Rust 程序设计语言 简体中文版

目录

Rust 程序设计语言 5
前言 6
简介 7
1. 入门指南 11
  1.1. 安装 12
  1.2. Hello, World! 15
  1.3. Hello, Cargo! 18
2. 编写一个猜数字游戏 22
3. 常见编程概念 38
  3.1. 变量与可变性 39
  3.2. 数据类型 43
  3.3. 函数 51
  3.4. 注释 57
  3.5. 控制流 58
4. 认识所有权 67
  4.1. 什么是所有权？ 68
  4.2. 引用与借用 81
  4.3. Slice 类型 88
5. 使用结构体组织相关联的数据 96
  5.1. 结构体的定义和实例化 97
  5.2. 结构体示例程序 103
  5.3. 方法语法 109
6. 枚举和模式匹配 114
  6.1. 枚举的定义 115
  6.2. match 控制流结构 122
  6.3. if let 和 let else 简洁控制流 128
7. 使用包、Crate 和模块管理不断增长的项目 131
  7.1. 包和 Crate 132
  7.2. 定义模块来控制作用域与私有性 133
  7.3. 引用模块树中项的路径 137
  7.4. 使用 use 关键字将路径引入作用域 144
  7.5. 将模块拆分成多个文件 150
8. 常见集合 152
  8.1. 使用 Vector 储存列表 153
  8.2. 使用字符串储存 UTF-8 编码的文本 158
  8.3. 使用 Hash Map 储存键值对 165
9. 错误处理 170
  9.1. 用 panic! 处理不可恢复的错误 171
  9.2. 用 Result 处理可恢复的错误 174
  9.3. 要不要 panic! 184
10. 泛型、Trait 和生命周期 188
  10.1. 泛型数据类型 191
  10.2. Trait：定义共同行为 199
  10.3. 生命周期确保引用有效 207

11. 编写自动化测试 219
11.1. 如何编写测试 220
11.2. 控制测试如何运行 236
11.3. 测试的组织结构 243
12. 一个 I/O 项目：构建命令行程序 249
12.1. 接受命令行参数 250
12.2. 读取文件 253
12.3. 重构以改进模块化与错误处理 255
12.4. 采用测试驱动开发完善库的功能 265
12.5. 处理环境变量 271
12.6. 将错误信息输出到标准错误而不是标准输出 276
13. 函数式语言特性：迭代器与闭包 278
13.1. 闭包：可以捕获其环境的匿名函数 279
13.2. 使用迭代器处理元素序列 289
13.3. 改进之前的 I/O 项目 294
13.4. 性能比较：循环对迭代器 298
14. 更多关于 Cargo 和 Crates.io 的内容 300
14.1. 采用发布配置自定义构建 301
14.2. 将 crate 发布到 Crates.io 302
14.3. Cargo 工作空间 311
14.4. 使用 cargo install 安装二进制文件 317
14.5. Cargo 自定义扩展命令 318
15. 智能指针 319
15.1. 使用 Box<T> 指向堆上数据 320
15.2. 使用 Deref Trait 将智能指针当作常规引用处理 326
15.3. 使用 Drop Trait 运行清理代码 331
15.4. Rc<T> 引用计数智能指针 335
15.5. RefCell<T> 与内部可变性模式 339
15.6. 引用循环会导致内存泄漏 347
16. 无畏并发 354
16.1. 使用线程同时地运行代码 355
16.2. 使用消息传递在线程间通信 361
16.3. 共享状态并发 367
16.4. 使用 Sync 与 Send Traits 的可扩展并发 373
17. Async 和 await 375
17.1. Futures 和 async 语法 378
17.2. 并发与 async 385
17.3. 使用任意数量的 futures 393
17.4. 流（Streams） 405
17.5. 深入理解 async 相关的 traits 414
17.6. future、任务和线程 422
18. 面向对象编程特性 425
18.1. 面向对象语言的特征 426
18.2. 顾及不同类型值的 trait 对象 429
18.3. 面向对象设计模式的实现 434
19. 模式与模式匹配 445

19.1. 所有可能会用到模式的位置 446
19.2. Refutability（可反驳性）：模式是否会匹配失效 451
19.3. 模式语法 453
20. 高级特性 465
20.1. 不安全 Rust 466
20.2. 高级 trait 477
20.3. 高级类型 488
20.4. 高级函数与闭包 494
20.5. 宏 498
21. 最后的项目：构建多线程 web server 507
21.1. 建立单线程 web server 508
21.2. 将单线程 server 变为多线程 server 517
21.3. 优雅停机与清理 533
22. 附录 541
22.1. A - 关键字 542
22.2. B - 运算符与符号 545
22.3. C - 可派生的 trait 552
22.4. D - 实用开发工具 555
22.5. E - 版本 558
22.6. F - 本书译本 559
22.7. G - Rust 是如何开发的与 “Nightly Rust” 560

Rust 程序设计语言

本书的英文原版作者为 Steve Klabnik 和 Carol Nichols，并由 Rust 社区补充完善。本简体中文译本由 Rust 中文社区翻译。

本书的当前版本假设你使用 Rust 1.85.0（2025-02-17 发布）或更高版本并在所有项目的 Cargo.toml 文件中通过 `edition = "2024"` 将其配置为使用 Rust 2024 edition 惯用法。请查看第一章的 “安装” 部分了解如何安装和升级 Rust。

本书的英文原版 HTML 格式可以在 https://doc.rust-lang.org/stable/book/ 在线阅读；使用 `rustup` 安装的 Rust 也包含一份英文离线版，运行 `rustup docs --book` 即可打开。

本书还有一些社区 翻译版本。（译者注：简体中文译本可以在 https://kaisery.github.io/trpl-zh-cn/ 在线阅读，PDF 版本请下载 Rust 程序设计语言 简体中文版.pdf）

本书也有由 No Starch Press 出版的纸质版和电子版。

> 🚨 **想要具有互动性的学习体验吗？试试 Rust Book 的另一个版本，其中包括测验、高亮、可视化等功能**：https://rust-book.cs.brown.edu

前言

Rust 程序设计语言的本质实际在于 **赋能**（*empowerment*）：无论你现在编写的是何种代码，Rust 能让你在更为广泛的编程领域走得更远，写出自信。（这一点并不显而易见）

举例来说，那些“系统层面”的工作涉及内存管理、数据表示和并发等底层细节。从传统角度来看，这是一个神秘的编程领域，只为浸润多年的极少数人所触及，也只有他们能避开那些臭名昭著的陷阱。即使谨慎的实践者，亦唯恐代码出现漏洞、崩溃或损坏。

Rust 破除了这些障碍：它消除了旧的陷阱，并提供了伴你一路同行的友好、精良的工具。想要 “深入” 底层控制的程序员可以使用 Rust，无需时刻担心出现崩溃或安全漏洞，也无需因为工具链不靠谱而被迫去了解其中的细节。更妙的是，语言设计本身会自然而然地引导你编写出可靠的代码，并且运行速度和内存使用上都十分高效。

已经在从事编写底层代码的程序员可以使用 Rust 来提升信心。例如，在 Rust 中引入并行是相对低风险的操作，因为编译器会替你找到经典的错误。同时你可以自信地采取更加激进的优化，而不会意外引入崩溃或漏洞。

但 Rust 并不局限于底层系统编程。它表达力强、写起来舒适，让人能够轻松地编写出命令行应用、网络服务器等各种类型的代码——在本书中就有这两者的简单示例。使用 Rust 能让你把在一个领域中学习的技能延伸到另一个领域：你可以通过编写网页应用来学习 Rust，接着将同样的技能应用到你的 Raspberry Pi（树莓派）上。

本书全面介绍了 Rust 为用户赋予的能力。其内容平易近人，致力于帮助你提升 Rust 的知识，并且提升你作为程序员整体的理解与自信。欢迎你加入 Rust 社区，让我们准备深入学习 Rust 吧！

—— Nicholas Matsakis 和 Aaron Turon

简介

> 注意：此书的英文原版与 No Starch Press 出版的《The Rust Programming Language》纸质版和电子版一致。

欢迎阅读《Rust 程序设计语言》，这是一本关于 Rust 的入门书籍。Rust 程序设计语言能帮助你编写更快、更可靠的软件。在编程语言设计中，高层的工程学与底层的控制往往是难以兼得的；而 Rust 则试图挑战这一矛盾。通过平衡强大的技术能力与优秀的开发者体验，Rust 为你提供了控制底层细节（如内存使用）的选项，而无需承受通常与此类控制相关的所有繁琐细节。

Rust 适合哪些人

Rust 因多种原因适合许多人。让我们看看几个最重要的群体。

开发者团队

Rust 已被证明是一个对于具有不同系统编程知识水平的大型开发团队协作而言，非常高效的工具。底层代码容易出现各种微妙的错误，在大多数其他语言中，这些错误只能通过广泛的测试和经验丰富的开发者的仔细审核代码来捕捉。在 Rust 中，编译器充当了守门员的角色，拒绝编译包含这些难以察觉的错误的代码，包括并发错误。通过与编译器合作，团队可以将时间集中在程序逻辑上，而不是追踪 bug。

Rust 也为系统编程世界带来了现代化的开发工具：

- Cargo 是内置的依赖管理器和构建工具，它能轻松增加、编译和管理依赖，并使依赖在 Rust 生态系统中保持一致。
- Rustfmt 格式化工具确保开发者遵循一致的代码风格。
- rust-analyzer 为集成开发环境（IDE）提供了强大的代码补全和内联错误信息功能。

通过使用 Rust 生态系统中丰富的工具，开发者在编写系统级代码时可以更加高效。

学生

Rust 适合学生群体，也适合有兴趣学习系统概念的人。许多人通过 Rust 学习了操作系统开发等主题。社区对学生问题非常欢迎并乐于回答。通过类似这本书以及其他内容的努力，Rust 团队希望使系统概念能为更多人所易于理解，特别是编程新手。

公司

数百家大小规模的公司在生产环境中使用 Rust 完成各种任务，包括命令行工具、Web 服务、DevOps 工具、嵌入式设备、音视频分析与转码、加密货币、生物信息学、搜索引擎、物联网（IOT）程序、机器学习，甚至是 Firefox 浏览器的重要部分。

开源开发者

Rust 适合那些希望构建 Rust 编程语言、社区、开发工具和库的开发者。我们非常欢迎你为 Rust 语言作出贡献。

重视速度和稳定性的开发者

Rust 适合那些渴望在编程语言中寻求速度与稳定性的开发者。对于速度来说，既是指 Rust 可以运行的多快，也是指编写 Rust 程序的速度。Rust 编译器的检查确保了增加功能和重构代码时的稳定性，这与那些缺乏这些检查的语言中脆弱的祖传代码形成了鲜明对比，开发者往往不敢去修改这些代码。通过追求零成本抽象（zero-cost abstractions）—— 将高级语言特性编译成底层代码，并且与手写的代码运行速度同样快。Rust 努力确保代码又安全又快速。

这里提到的只是几个较大的受益群体，Rust 语言也希望能支持更多其他用户。总的来说，Rust 最重要的目标是消除数十年来程序员习以为常的取舍，让安全和高效、速度和易读易用**可以兼得**。试试看 Rust，说不定它的选择就适合你。

本书适合哪些人

本书假设你已经有其他编程语言的经验，任何语言均可，我们力求让各种语言背景的人都能读懂。本书的重点不是程序设计**本身**，也不是程序设计思维。如果你完全没学过编程，建议你先阅读专门介绍程序设计的书籍。

如何阅读本书

本书大体上假设你按从头到尾的顺序阅读。后面的章节建立在前面章节概念的基础上。前面的章节可能不会深入介绍部分主题，而是留待后续章节重新讨论。

本书分为两类章节：概念章节和项目章节。在概念章节中，我们学习 Rust 的某个方面。在项目章节中，我们应用目前所学的知识一同构建小型程序。第二、十二和二十一章是项目章节；其余都是概念章节。

第一章介绍如何安装 Rust，如何编写一个 “Hello, world!” 程序，以及如何使用 Rust 的包管理器和构建工具 Cargo。第二章是一个编写 Rust 语言的实战介绍，我们会构建一个猜数字游戏。我们会站在较高的层次介绍一些概念，而后续章节将提供更多细节。如果你希望立刻就动手实践一下，第二章是开始的好地方。第三章介绍 Rust 中类似其他编程语言的特性，第四章会学习 Rust 的所有权系统。如果你是一个特别细致的学习者，喜欢在进入下一环节之前学习每一个细节，你可能会想要跳过第二章，直接阅读第三章，等到你想要通过项目应用所学到的细节时再回到第二章。

第五章讨论结构体（struct）和方法，第六章介绍枚举（enum）、`match` 表达式和 `if let` 控制流结构。在 Rust 中，创建自定义类型需要用到结构体和枚举。

第七章介绍 Rust 的模块（module）系统，其中的私有性规则用来组织代码和公开的 API（应用程序接口）。第八章讨论标准库提供的常见集合数据结构，例如 Vector（向量）、字符串和 Hash Map（散列表）。第九章探索 Rust 的错误处理的理念与技术。

第十章深入介绍泛型（generic）、Trait 和生命周期（lifetime），这些功能让你能够定义适用于多种类型的代码。第十一章全面讲述了测试，因为就算 Rust 有安全保证，也需要测试确保程序逻辑正确。第十二章中将会构建我们自己的 `grep` 命令行工具的功能子集实现，用于在文件中搜索文本。为此会用到之前章节讨论的很多概念。

第十三章探索闭包（closure）和迭代器（iterator），这两个 Rust 特性来自函数式编程语言。第十四章会深入探讨 Cargo 并介绍分享代码库的最佳实践。第十五章讨论标准库提供的智能指针以及相关的 trait。

第十六章将引导我们了解不同的并发编程模型，并探讨 Rust 如何帮助你无畏地进行多线程编程。第十七章将在此基础上进一步探索 Rust 的 async 和 await 语法，以及它们所支持的轻量级并发模型。

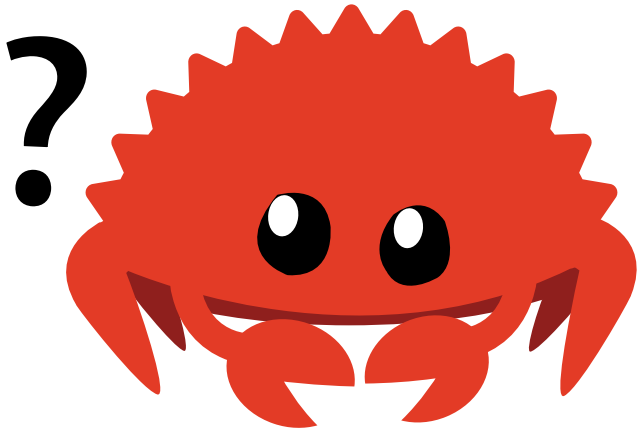
第十八章着眼于 Rust 风格与你可能比较熟悉的 OOP（面向对象编程）原则之间的比较。第十九章是一个模式和模式匹配的参考，它们是在 Rust 程序中表达思想的有效方式。第二十章是一个高级主题大杂烩，包括不安全 Rust（unsafe Rust）、宏（macro）和更多关于生命周期、Trait、类型、函数和闭包的内容。

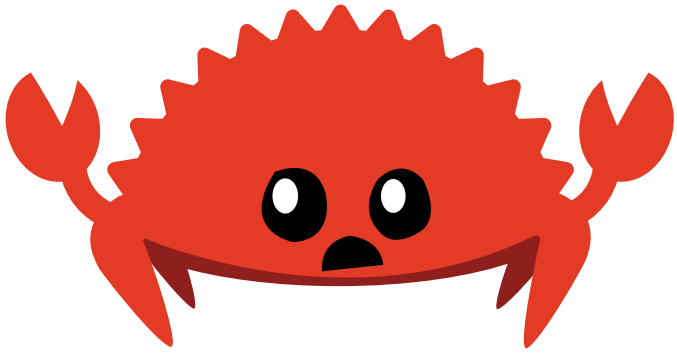
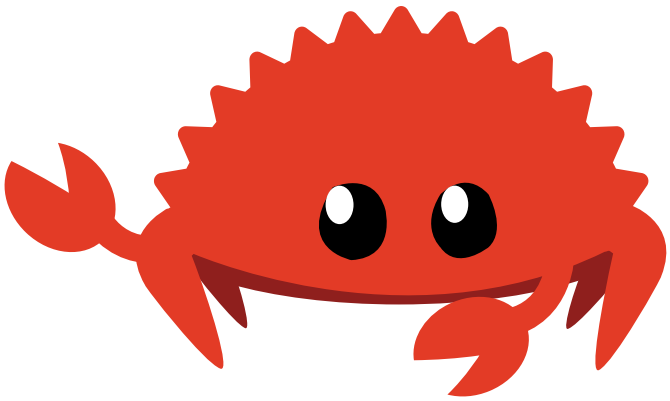
第二十一章我们将会完成一个项目，实现一个底层多线程的 Web 服务端！

最后的附录包含了一些关于该语言的实用信息，其格式更像是参考资料。附录 A 涵盖了 Rust 的关键字，附录 B 涵盖了 Rust 的运算符和符号，附录 C 涵盖了标准库提供的可派生 trait，附录 D 涵盖了一些有用的开发工具，而附录 E 解释了 Rust 版本。在附录 F 中，你可以找到本书的翻译版本，而在附录 G 中，我们将讨论 Rust 是如何制作的以及什么是 nightly Rust。

阅读本书没有错误的方式：如果你想跳过前面的内容，尽管跳过！如果你遇到任何困惑，可能需要回到前面的章节。请采取对你最有效的方式进行阅读。

学习 Rust 的一个重要部分是学会如何阅读编译器显示的错误信息：它们会指引你编写出能运行的代码。为此，我们将提供许多不能编译的示例，以及在每种情况下编译器将显示的错误信息。请知悉，如果你输入并运行一个随机示例，它可能无法编译！确保你阅读了示例周围的文本，以判断你尝试运行的示例是否出错。Ferris 也将帮助你区分那些不是意在工作的代码：

Ferris	**含义**
	这段代码无法通过编译！

	这段代码会 Panic!
	这段代码的运行结果不符合预期。

在大部分情况，我们会指导你将无法通过编译的代码修改为正确版本。

源代码

生成本书的源码可以在 GitHub 上找到。

> 译者注：此译本也有 GitHub 仓库，欢迎提交 Issue 和 PR :)

入门指南

让我们开始 Rust 之旅！有很多内容需要学习，但每次旅程总有起点。在本章中，我们会讨论：

- 在 Linux、macOS 和 Windows 上安装 Rust
- 编写一个打印 `Hello, world!` 的程序
- 使用 Rust 的包管理器和构建系统 `cargo`

安装

第一步是安装 Rust。我们会通过 `rustup` 下载 Rust，这是一个管理 Rust 版本和相关工具的命令行工具。下载时需要联网。

> 注意：如果你出于某些理由倾向于不使用 `rustup`，请到 Rust 的其他安装方法页面 查看其它安装选项。

接下来的步骤会安装最新的稳定版 Rust 编译器。Rust 的稳定性确保本书所有示例在最新版本的 Rust 中能够继续编译。不同版本的输出可能略有不同，因为 Rust 经常改进错误信息和警告。也就是说，任何通过这些步骤安装的最新稳定版 Rust，都应该能正常运行本书中的内容。

> **命令行标记**
>
> 本章和全书中，我们会展示一些在终端中使用的命令。所有需要输入到终端的行都以 `$` 开头。你不需要输入 `$` 字符；这里显示的 `$` 字符表示命令行提示符，仅用于提示每行命令的起点。不以 `$` 起始的行通常展示前一个命令的输出。另外，PowerShell 专用的示例会采用 `>` 而不是 `$`。

在 Linux 或 macOS 上安装 `rustup`

如果你使用 Linux 或 macOS，打开终端并输入如下命令：

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

此命令下载一个脚本并开始安装 `rustup` 工具，这会安装最新稳定版 Rust。过程中可能会提示你输入密码。如果安装成功，将会出现如下内容：

```
Rust is installed now. Great!
```

另外，你还需要一个 链接器（*linker*），这是 Rust 用来将其编译的输出连接成一个文件中的程序。很可能你已经有一个了。如果你遇到了链接器错误，请尝试安装一个 C 编译器，它通常包括一个链接器。C 编译器也很有用，因为一些常见的 Rust 包依赖于 C 代码，因此需要安装一个 C 编译器。

在 macOS 上，你可以通过运行以下命令获得 C 语言编译器：

```
$ xcode-select --install
```

Linux 用户通常需要根据发行版（distribution）文档安装 GCC 或 Clang。比如，如果你使用 Ubuntu，可以安装 `build-essential` 包。

在 Windows 上安装 `rustup`

在 Windows 上，前往 https://www.rust-lang.org/install.html 并按照说明安装 Rust。在安装过程的某个步骤，你会被提示要安装 Visual Studio。它提供了一个链接器和编译程序所需的原生库。如果你在此步骤需要更多帮助，请访问 https://rust-lang.github.io/rustup/installation/windows-msvc.html。

本书的余下部分会使用能同时运行于 *cmd.exe* 和 PowerShell 的命令。如果存在特定差异，我们会解释使用哪一个。

故障排除（Troubleshooting）

要检查是否正确安装了 Rust，打开命令行并输入：

```
$ rustc --version
```

你应该可以看到按照以下格式显示的最新稳定版本的版本号、对应的 Commit Hash 和 Commit 日期：

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

如果看到了这样的信息，就说明 Rust 已经安装成功了！如果没看到，请按照下面说明的方法检查 Rust 是否在您的 `%PATH%` 系统变量中。

在 Windows CMD 中，请使用命令：

```
> echo %PATH%
```

在 PowerShell 中，请使用命令：

```
> echo $env:Path
```

在 Linux 和 macOS 中，请使用命令：

```
$ echo $PATH
```

如果一切正确但 Rust 仍不能使用，有许多地方可以求助。您可以在社区页面查看如何与其他 Rustaceans（Rust 用户的称号，有自嘲意味）联系。

更新与卸载

通过 `rustup` 安装了 Rust 之后，更新到最新版本就很简单了。只需要在您对应的命令行中运行如下更新脚本：

```
$ rustup update
```

若要卸载 Rust 和 `rustup`，请在命令行中运行如下卸载脚本：

```
```

```
$ rustup self uninstall
```

本地文档

安装程序也自带一份文档的本地拷贝，可以离线阅读。运行 `rustup doc` 在浏览器中查看本地文档。

任何时候，如果你拿不准标准库中的类型或函数的用途和用法，请查阅应用程序接口（application programming interface，API）文档！

文本编辑器和集成开发环境（Integrated Development Environments, IDE）

本书不会假设你使用何种工具来编写 Rust 代码。几乎任何文本编辑器都可以搞定！然而，很多文本编辑器和集成开发环境（IDE）内置了 Rust 支持。你总是可以在 Rust 官网的工具页面找到很多相对流行的编辑器和 IDE 列表。

离线使用本书

在很多示例中，我们会使用多于标准库的 Rust 包。为了处理这些示例，要么需要网络连接要么需要提前下载这些依赖。为了提前下载这些依赖，可以运行如下命令。（我们稍后会详细解释 `cargo` 是什么以及这每一个命令在干什么。）

```
$ cargo new get-dependencies
$ cd get-dependencies
$ cargo add rand@0.8.5 trpl@0.2.0
```

这会缓存这些包的下载所以之后你不用再下载它们。一旦你运行了这些命令，就可以在本书之后所有的 `cargo` 命令中使用 `--offline` 参数来使用这些缓存的版本而不是尝试使用网络。

Hello, World!

既然安装好了 Rust，是时候来编写第一个 Rust 程序了。当学习一门新语言的时候，使用该语言在屏幕上打印 `Hello, world!` 是一项传统，我们将沿用这一传统！

> 注意：本书假设你熟悉基本的命令行操作。Rust 对于你的编辑器、工具，以及代码位于何处并没有特定的要求，如果你更倾向于使用集成开发环境（IDE），而不是命令行，请尽管使用你喜欢的 IDE。目前很多 IDE 都在一定程度上支持 Rust；查看 IDE 文档以了解更多细节。Rust 团队一直致力于借助 `rust-analyzer` 提供强大的 IDE 支持。详见附录 D。

创建项目目录

首先创建一个存放 Rust 代码的目录。Rust 并不关心代码的存放位置，不过对于本书的练习和项目来说，我们建议你在 home 目录中创建 *projects* 目录，并将你的所有项目存放在这里。

打开终端并输入如下命令创建 *projects* 目录，并在 *projects* 目录中为 “Hello, world!” 项目创建一个目录。

对于 Linux、macOS 和 Windows PowerShell，输入：

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

对于 Windows CMD，输入：

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

编写并运行 Rust 程序

接下来，新建一个源文件，命名为 *main.rs*。Rust 源文件总是以 *.rs* 扩展名结尾。如果文件名包含多个单词，那么按照命名习惯，应当使用下划线来分隔单词。例如命名为 *hello_world.rs*，而不是 *helloworld.rs*。

现在打开刚创建的 *main.rs* 文件，输入示例 1-1 中的代码。

文件名：main.rs

```
fn main() {
    println!("Hello, world!");
}
```

保存文件，并回到当前目录为 */projects/hello_world* 的终端窗口。在 Linux 或 macOS 上，输入如下命令，编译并运行文件：

```
$ rustc main.rs
$ ./main
Hello, world!
```

在 Windows 上，输入命令 `.\main.exe`，而不是 `./main`：

```
> rustc main.rs
> .\main
Hello, world!
```

不管使用何种操作系统，终端应该打印字符串 `Hello, world!`。如果没有看到这些输出，回到安装部分的 “故障排除” 小节查找有帮助的方法。

如果 `Hello, world!` 出现了，恭喜你！你已经正式编写了一个 Rust 程序。现在你成为一名 Rust 程序员，欢迎！

Rust 程序的结构

现在，让我们回过头来仔细看看这个 “Hello, world!” 程序。这是第一块拼图：

```
fn main() {

}
```

这几行定义了一个名叫 `main` 的函数。`main` 函数是一个特殊的函数：在可执行的 Rust 程序中，它总是最先运行的代码。第一行代码声明了一个叫做 `main` 的函数，它没有参数也没有返回值。如果有参数的话，它们的名称应该出现在小括号 `()` 中。

函数体被包裹在 `{}` 中。Rust 要求所有函数体都要用花括号包裹起来。一般来说，将左花括号与函数声明置于同一行并以空格分隔，是良好的代码风格。

> 注：如果你希望在 Rust 项目中保持一种标准风格，可以使用名为 `rustfmt` 的自动格式化工具将代码格式化为特定的风格（更多内容详见附录 D 中的 `rustfmt`）。Rust 团队已经在标准的 Rust 发行版中包含了这个工具，就像 `rustc` 一样。所以它应该已经安装在你的电脑中了！

在 `main` 函数体中有如下代码：

```
println!("Hello, world!");
```

这行代码完成这个简单程序的所有工作：在屏幕上打印文本。这里有三个重要的细节需要注意。

首先，`println!` 调用了一个 Rust 宏（macro）。如果是调用函数，则应输入 `println`（没有 `!`）。我们将在第二十章详细讨论宏。现在你只需记住，当看到符号 `!` 的时候，就意味着调用的是宏而不是普通函数，并且宏并不总是遵循与函数相同的规则。

第二，`"Hello, world!"` 是一个字符串。我们把这个字符串作为一个参数传递给 `println!`，字符串将被打印到屏幕上。

第三，该行以分号结尾（`;`），这代表一个表达式的结束和下一个表达式可以开始。大部分 Rust 代码行以分号结尾。

编译和运行是彼此独立的步骤

你刚刚运行了一个新创建的程序，那么让我们检查此过程中的每一个步骤。

在运行 Rust 程序之前，必须先使用 Rust 编译器编译它，即输入 `rustc` 命令并传入源文件名称，如下：

```
$ rustc main.rs
```

如果你有 C 或 C++ 背景，就会发现这与 `gcc` 和 `clang` 类似。编译成功后，Rust 会输出一个二进制的可执行文件。

在 Linux、macOS 或 Windows 的 PowerShell 上，在 shell 中输入 `ls` 命令可以看见这个可执行文件。

```
$ ls
main  main.rs
```

在 Linux 和 macOS，你会看到两个文件。在 Windows PowerShell 中，你会看到同使用 CMD 相同的三个文件。在 Windows 的 CMD 上，则输入如下命令：

```
> dir /B %= the /B option says to only show the file names =%
main.exe
main.pdb
main.rs
```

这展示了扩展名为 *.rs* 的源文件、可执行文件（在 Windows 下是 *main.exe*，其它平台是 *main*），以及当使用 CMD 时会有一个包含调试信息、扩展名为 *.pdb* 的文件。从这里开始运行 *main* 或 *main.exe* 文件，如下：

```
$ ./main # Windows 是 .\main.exe
```

如果这里的 *main.rs* 是上文所述的 “Hello, world!” 程序，那么在终端上就会打印出 `Hello, world!`。

如果你更熟悉动态语言，如 Ruby、Python 或 JavaScript，则可能不习惯将编译和运行分为两个单独的步骤。Rust 是一种 **预编译静态类型**（*ahead-of-time compiled*）语言，这意味着你可以编译程序，并将可执行文件送给其他人，他们甚至不需要安装 Rust 就可以运行。如果你给他人一个 *.rb*、*.py* 或 *.js* 文件，他们需要先分别安装 Ruby，Python，JavaScript 实现（运行时环境，VM）。不过在这些语言中，只需要一句命令就可以编译和运行程序。这一切都是语言设计上的权衡取舍。

仅仅使用 `rustc` 编译简单程序是没问题的，不过随着项目的增长，你可能需要管理你项目的方方面面，并让代码易于分享。接下来，我们要介绍一个叫做 Cargo 的工具，它会帮助你编写真实世界中的 Rust 程序。

Hello, Cargo!

Cargo 是 Rust 的构建系统和包管理器。大多数 Rustacean 们使用 Cargo 来管理他们的 Rust 项目，因为它可以为你处理很多任务，比如构建代码、下载依赖库并编译这些库。（我们把代码所需要的库叫做 **依赖**（*dependencies*））。

最简单的 Rust 程序，比如我们刚刚编写的，没有任何依赖。如果使用 Cargo 来构建 “Hello, world!” 项目，将只会用到 Cargo 构建代码的那部分功能。在编写更复杂的 Rust 程序时，你将添加依赖项，如果使用 Cargo 启动项目，则添加依赖项将更加容易。

由于绝大多数 Rust 项目使用 Cargo，本书接下来的部分假设你也使用 Cargo。如果使用 “安装” 部分介绍的官方安装包的话，则自带了 Cargo。如果通过其他方式安装的话，可以在终端输入如下命令检查是否安装了 Cargo：

```
$ cargo --version
```

如果你看到了版本号，说明已安装！如果看到类似 `command not found` 的错误，你应该查看相应安装文档以确定如何单独安装 Cargo。

使用 Cargo 创建项目

我们使用 Cargo 创建一个新项目，然后看看与上面的 “Hello, world!” 项目有什么不同。回到 *projects* 目录（或者你存放代码的目录）。接着，可在任何操作系统下运行以下命令：

```
$ cargo new hello_cargo
$ cd hello_cargo
```

第一行命令新建了名为 *hello_cargo* 的目录和项目。我们将项目命名为 *hello_cargo*，同时 Cargo 在一个同名目录中创建项目文件。

进入 *hello_cargo* 目录并列出文件。将会看到 Cargo 生成了两个文件和一个目录：一个 *Cargo.toml* 文件，一个 *src* 目录，以及位于 *src* 目录中的 *main.rs* 文件。

这也会在 *hello_cargo* 目录初始化了一个 git 仓库，以及一个 *.gitignore* 文件。如果在一个已经存在的 git 仓库中运行 `cargo new`，则这些 git 相关文件则不会生成；可以通过运行 `cargo new --vcs=git` 来覆盖这些行为。

> 注意：git 是一个常用的版本控制系统（version control system，VCS）。可以通过 `--vcs` 参数使 `cargo new` 切换到其它版本控制系统（VCS），或者不使用 VCS。运行 `cargo new --help` 查看可用的选项。

请自行选用文本编辑器打开 *Cargo.toml* 文件。它应该看起来与示例 1-2 中代码类似：

文件名：Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2024"
```

```
[dependencies]
```

这个文件使用 *TOML* (*Tom's Obvious, Minimal Language*) 格式，这是 Cargo 配置文件的格式。

第一行，`[package]`，是一个片段 section 标题，表明下面的语句用来配置一个包。随着我们在这个文件增加更多的信息，还将增加其他 section。

接下来的三行设置了 Cargo 编译程序所需的配置：项目的名称、项目的版本以及要使用的 Rust 版本。附录 E 会介绍 `edition` 的值。

最后一行，`[dependencies]`，是罗列项目依赖的 section 的开始。在 Rust 中，代码包被称为 *crates*。这个项目并不需要其他的 crate，不过在第二章的第一个项目会用到依赖，那时会用得上这个 section。

现在打开 *src/main.rs* 看看：

文件名：src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Cargo 为你生成了一个 "Hello, world!" 程序，正如我们之前编写的示例 1-1！目前为止，我们的项目与 Cargo 生成项目的区别是 Cargo 将代码放在 *src* 目录，同时项目根目录包含一个 *Cargo.toml* 配置文件。

Cargo 期望源文件存放在 *src* 目录中。项目根目录只存放 README、license 信息、配置文件和其他跟代码无关的文件。使用 Cargo 帮助你保持项目干净整洁。一切各得其所，井井有条。

如果没有使用 Cargo 开始项目，比如我们创建的 "Hello, world!" 项目，你可以将其转换为使用 Cargo 的项目。将项目代码移入 *src* 目录，并创建一个合适的 *Cargo.toml* 文件。一个简单的创建 *Cargo.toml* 文件的方法是运行 `cargo init`，它会自动为你创建该文件。

构建并运行 Cargo 项目

现在让我们看看通过 Cargo 构建和运行 "Hello, world!" 程序有什么不同！在 *hello_cargo* 目录下，输入下面的命令来构建项目：

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

这个命令会创建一个可执行文件 *target/debug/hello_cargo* （在 Windows 上是 *targetdebughello_cargo.exe*），而不是放在目前目录下。由于默认的构建方法是调试构建（debug build），Cargo 会将可执行文件放在名为 *debug* 的目录中。可以通过这个命令运行可执行文件：

```
$ ./target/debug/hello_cargo # 或者在 Windows 下为 .\target\debug\hello_cargo.exe
Hello, world!
```

如果一切顺利，终端上应该会打印出 `Hello, world!`。首次运行 `cargo build` 时，也会使 Cargo 在项目根目录创建一个新文件：*Cargo.lock*。这个文件记录项目依赖的实际版本。这个项目并没有依赖，所以其内容比较少。你永远也不需要手动编辑该文件；`Cargo` 会为你管理它。

我们刚刚使用 `cargo build` 构建了项目，并使用 `./target/debug/hello_cargo` 运行了程序，也可以使用 `cargo run` 在一个命令中同时编译并运行生成的可执行文件：

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

比起要记得运行 `cargo build` 之后再用可执行文件的完整路径来运行程序，使用 `cargo run` 更方便，所以大多数开发者会使用 `cargo run`。

注意这一次并没有出现表明 Cargo 正在编译 `hello_cargo` 的输出。Cargo 发现文件并没有被改变，所以它并没有重新构建，而是直接运行了二进制文件。如果修改了源文件的话，Cargo 会在运行之前重新构建项目，并会出现像这样的输出：

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo 还提供了一个叫 `cargo check` 的命令。该命令快速检查代码确保其可以编译，但并不产生可执行文件：

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

为什么你会不需要可执行文件呢？通常 `cargo check` 要比 `cargo build` 快得多，因为它省略了生成可执行文件的步骤。如果你在编写代码时持续的进行检查，`cargo check` 可以让你快速了解现在的代码能不能正常通过编译！为此很多 Rustaceans 编写代码时定期运行 `cargo check` 确保它们可以编译。当准备好使用可执行文件时才运行 `cargo build`。

我们回顾下已学习的 Cargo 内容：

- 可以使用 `cargo new` 创建项目。
- 可以使用 `cargo build` 构建项目。
- 可以使用 `cargo run` 一步构建并运行项目。
- 可以使用 `cargo check` 在不生成二进制文件的情况下构建项目来检查错误。
- 有别于将构建结果放在与源码相同的目录，Cargo 会将其放到 *target/debug* 目录。

使用 Cargo 的一个额外的优点是，不论你使用什么操作系统，其命令都是一样的。所以从现在开始本书将不再分别为 Linux 和 macOS 以及 Windows 提供相应的命令。

发布（release）构建

当项目最终准备好发布时，可以使用 `cargo build --release` 来优化编译项目。这会在 *target/release* 而不是 *target/debug* 下生成可执行文件。这些优化可以让 Rust 代码运行的更快，不过启用这些优化也需要消耗更长的编译时间。这也就是为什么会有两种不同的配置：一种是为了开发，你需要快速且频繁地重新构建；另一种是为用户构建最终程序，它们不会经常重新构建，并且希望程序运行得越快越好。如果你在基准测试代码的运行时间，请确保运行 `cargo build --release` 并使用 *target/release* 下的可执行文件进行测试。

把 Cargo 当作习惯

对于简单项目，Cargo 并不比 `rustc` 提供了更多的优势，但随着程序变得更复杂，其价值会逐渐显现。一旦程序壮大到由多个文件组成，亦或者是需要其他的依赖，让 Cargo 协调构建过程就会简单得多。

即便 `hello_cargo` 项目十分简单，它现在也使用了很多在你之后的 Rust 生涯将会用到的实用工具。其实，要在任何已存在的项目上工作时，可以使用如下命令通过 Git 检出代码，移动到该项目目录并构建：

```
$ git clone example.org/someproject
$ cd someproject
$ cargo build
```

关于更多 Cargo 的信息，请查阅 其文档。

总结

你已经准备好开启 Rust 之旅了！在本章中，你学习了如何：

- 使用 `rustup` 安装最新稳定版的 Rust
- 更新到新版的 Rust
- 打开本地安装的文档
- 直接通过 `rustc` 编写并运行 Hello, world! 程序
- 使用 Cargo 创建并运行新项目

是时候通过构建更实质性的程序来熟悉读写 Rust 代码了。所以在第二章我们会构建一个猜数字游戏程序。如果你更愿意从学习 Rust 常用的编程概念开始，请阅读第三章，接着再回到第二章。

编写一个猜数字游戏

让我们一起动手完成一个项目来快速上手 Rust！本章将介绍一些 Rust 中常见的概念，并通过真实的程序来展示如何运用它们。你将会学到 `let`、`match`、方法（methods）、关联函数（associated functions）、外部 crate 等知识！后续章节会深入探讨这些概念的细节。在这一章，我们将主要练习基础内容。

我们会实现一个经典的新手编程问题：猜数字游戏。游戏的规则如下：程序将会生成一个 1 到 100 之间的随机整数。然后提示玩家输入一个猜测值。输入后，程序会指示该猜测是太低还是太高。如果猜对了，游戏会打印祝贺信息并退出。

准备一个新项目

要创建一个新项目，进入第一章中创建的 *projects* 目录，使用 Cargo 新建一个项目，如下：

```
$ cargo new guessing_game
$ cd guessing_game
```

第一个命令，`cargo new`，它获取项目的名称（`guessing_game`）作为第一个参数。第二个命令进入到新创建的项目目录。

看看生成的 *Cargo.toml* 文件：

文件名：Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2024"

[dependencies]
```

正如第一章那样，`cargo new` 生成了一个 “Hello, world!” 程序。查看 *src/main.rs* 文件：

文件名：src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

现在使用 `cargo run` 命令，一步完成 “Hello, world!” 程序的编译和运行：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/guessing_game`
Hello, world!
```

当你需要在项目中快速迭代时，`run` 命令就能派上用场，正如我们在这个游戏项目中做的，在下一次迭代之前快速测试每一次迭代。

重新打开 *src/main.rs* 文件。我们将会在这个文件中编写全部的代码。

处理一次猜测

猜数字程序的第一部分请求和处理用户输入，并检查输入是否符合预期的格式。首先，我们会允许玩家输入一个猜测。在 *src/main.rs* 中输入示例 2-1 中的代码。

文件名：src/main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

这些代码包含很多信息，我们一行一行地讲解。为了获取用户输入并打印结果作为输出，我们需要将 `io` 输入/输出库引入当前作用域。`io` 库来自于标准库，也被称为 `std`：

```
use std::io;
```

默认情况下，Rust 设定了若干个会自动导入到每个程序作用域中的标准库内容，这组内容被称为 预导入（*prelude*） 内容。你可以在标准库文档中查看预导入的所有内容。

如果你需要的类型不在预导入内容中，就必须使用 `use` 语句显式地将其引入作用域。`std::io` 库提供很多有用的功能，包括接收用户输入的功能。

如第一章所提及，`main` 函数是程序的入口点：

```
fn main() {
```

`fn` 语法声明了一个新函数，小括号 `()` 表明没有参数，大括号 `{` 作为函数体的开始。

第一章也提及了 `println!` 是一个在屏幕上打印字符串的宏：

```
    println!("Guess the number!");

    println!("Please input your guess.");
```

这些代码仅仅打印提示，介绍游戏的内容然后请求用户输入。

使用变量储存值

接下来，创建一个 **变量**（*variable*）来储存用户输入，像这样：

```
    let mut guess = String::new();
```

现在程序开始变得有意思了！这一小行代码发生了很多事。我们使用 `let` 语句来创建变量。这里是另外一个例子：

```
let apples = 5;
```

这行代码新建了一个叫做 `apples` 的变量并把它绑定到值 `5` 上。在 Rust 中，变量默认是不可变的，这意味着一旦我们给变量赋值，这个值就不可以再修改了。我们将会在第三章的 “变量与可变性” 部分详细讨论这个概念。下面的例子展示了如何在变量名前使用 `mut` 来使一个变量可变：

```
let apples = 5; // 不可变
let mut bananas = 5; // 可变
```

> 注意：`//` 语法开始一个注释，持续到行尾。Rust 忽略注释中的所有内容，第三章将会详细介绍注释。

回到猜数字程序中。现在我们知道了 `let mut guess` 会引入一个叫做 `guess` 的可变变量。等号（`=`）告诉 Rust 我们现在想将某个值绑定在变量上。等号的右边是 `guess` 所绑定的值，它是 `String::new` 的结果，这个函数会返回一个 `String` 的新实例。`String` 是一个标准库提供的字符串类型，它是 UTF-8 编码的可增长文本块。

`::new` 那一行的 `::` 语法表明 `new` 是 `String` 类型的一个 **关联函数**（*associated function*）。关联函数是针对某个类型实现的函数，在这个例子中是 `String`。这个 `new` 函数创建了一个新的空字符串。你会发现许多类型上都有一个 `new` 函数，因为这是为某种类型创建新值的常用函数名。

总的来说，`let mut guess = String::new();` 这一行创建了一个可变变量，当前它绑定到一个新的 `String` 空实例上。呼！

接收用户输入

回忆一下，我们在程序的第一行使用 `use std::io;` 从标准库中引入了输入/输出功能。现在调用 `io` 库中的函数 `stdin`，这允许我们处理用户输入：

```
    io::stdin()
        .read_line(&mut guess)
```

如果程序的开头没有使用 `use std::io;` 引入 `io` 库，我们仍可以通过把函数调用写成 `std::io::stdin` 来使用该函数。`stdin` 函数返回一个 `std::io::Stdin` 的实例，这是一种代表终端标准输入句柄的类型。

接下来，代码中的 `.read_line(&mut guess)` 调用了标准输入句柄上的 `read_line` 方法，以获取用户输入。我们还将 `&mut guess` 作为参数传递给 `read_line` 函数，让其将用户输入储存到这个字符串中。`read_line` 的工作是，无论用户在标准输入中键入什么内容，都将其追加（不

会覆盖其原有内容）到一个字符串中，因此它需要字符串作为参数。这个字符串参数应该是可变的，以便 `read_line` 将用户输入附加上去。

`&` 表示这个参数是一个 **引用**（*reference*），它允许多处代码访问同一处数据，而无需在内存中多次拷贝。引用是一个复杂的特性，Rust 的一个主要优势就是安全而简单的操纵引用。完成当前程序并不需要了解如此多细节。现在，我们只需知道它像变量一样，默认是不可变的。因此，需要写成 `&mut guess` 来使其可变，而不是 `&guess`。（第四章会更全面地讲解引用。）

使用 `Result` 类型来处理潜在的错误

我们还没有完全分析完这行代码。虽然我们已经讲到了第三行代码，但要注意：它仍是逻辑行（虽然换行了但仍是语句）的一部分。后一部分是这个方法（method）：

```
        .expect("Failed to read line");
```

我们也可以将代码这样写：

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

不过，过长的代码行难以阅读，所以最好拆开来写。通常来说，当使用 `.method_name()` 语法调用方法时引入换行符和空格将长的代码行拆开是明智的。现在来看看这行代码干了什么。

之前提到了 `read_line` 会将用户输入附加到传递给它的字符串中，不过它也会返回一个类型为 `Result` 的值。`Result` 是一种枚举类型，通常也写作 *enum*，它可以是多种可能状态中的一个。我们把每种可能的状态称为一种 **枚举成员**（*variant*）。

第六章将介绍枚举的更多细节。这里的 `Result` 类型将用来编码错误处理的信息。

`Result` 的成员是 `Ok` 和 `Err`，`Ok` 成员表示操作成功，内部包含成功时产生的值。`Err` 成员则意味着操作失败，并且 `Err` 中包含有关操作失败的原因或方式的信息。

`Result` 类型的值，像其他类型一样，拥有定义于其实例上的方法。`Result` 的实例拥有 `expect` 方法。如果 `io::Result` 实例的值是 `Err`，`expect` 会导致程序崩溃，并输出当做参数传递给 `expect` 的信息。所以当 `read_line` 方法返回 `Err`，则可能是来源于底层操作系统错误的结果。如果 `Result` 实例的值是 `Ok`，`expect` 会获取 `Ok` 中的值并原样返回。在本例中，这个值是用户输入到标准输入中的字节数。

如果不调用 `expect`，程序也能编译，不过会出现一个警告：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
   |
10 |     let _ = io::stdin().read_line(&mut guess);
```

```
    |     +++++++

warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.59s
```

Rust 警告我们没有使用 `read_line` 的返回值 `Result`，说明有一个可能的错误没有处理。

消除警告的正确做法是实际去编写错误处理代码，不过由于我们就是希望程序在出现问题时立即崩溃，所以直接使用 `expect`。第九章 会学习如何从错误中恢复。

使用 `println!` 占位符打印值

除了位于结尾的右花括号，目前为止就只有这一行代码值得讨论一下了：

```
    println!("You guessed: {guess}");
```

这行代码现在打印了存储用户输入的字符串。`{}` 这对大括号是一个占位符：把 `{}` 想象成小蟹钳，可以夹住合适的值。当打印变量的值时，变量名可以写进大括号中。当打印表达式的执行结果时，格式化字符串（format string）中大括号中留空，格式化字符串后跟逗号分隔的需要打印的表达式列表，其顺序与每一个空大括号占位符的顺序一致。在一个 `println!` 调用中打印变量和表达式的值看起来像这样：

```
let x = 5;
let y = 10;

println!("x = {x} and y + 2 = {}", y + 2);
```

这行代码会打印出 `x = 5 and y + 2 = 12`。

测试第一部分代码

让我们来测试下猜数字游戏的第一部分。使用 `cargo run` 运行：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 6.44s
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

至此为止，游戏的第一部分已经完成：我们从键盘获取输入并打印了出来。

生成一个秘密数字

接下来，需要生成一个秘密数字，好让用户来猜。秘密数字应该每次都不同，这样重复玩才不会乏味；范围应该在 1 到 100 之间，这样才不会太困难。Rust 标准库中尚未包含随机数功能。然而，Rust 团队还是提供了一个包含上述功能的 `rand` crate。

使用 crate 来增加更多功能

记住，crate 是一组 Rust 源代码文件。我们正在构建的项目是一个 二进制 *crate*，它生成一个可执行文件。 `rand` crate 是一个 库 *crate*，库 crate 可以包含任意能被其他程序使用的代码，但是无法独立执行。

Cargo 对外部 crate 的运用是其真正的亮点所在。在我们使用 `rand` 编写代码之前，需要修改 *Cargo.toml* 文件，引入一个 `rand` 依赖。现在打开这个文件并将下面这一行添加到 `[dependencies]` section 标题之下。在当前版本下，请确保按照我们这里的方式指定 `rand`，否则本教程中的示例代码可能无法工作。

文件名：Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

在 *Cargo.toml* 文件中，标题以及之后的内容属同一个 section，直到遇到下一个标题才开始新的 section。`[dependencies]` section 告诉 Cargo 本项目依赖了哪些外部 crate 及其版本。本例中，我们使用语义化版本 `0.8.5` 来指定 `rand` crate。Cargo 理解 语义化版本（Semantic Versioning）（有时也称为 *SemVer*），这是一种定义版本号的标准。`0.8.5` 事实上是 `^0.8.5` 的简写，它表示任何至少是 `0.8.5` 但小于 `0.9.0` 的版本。

Cargo 认为这些版本与 `0.8.5` 版本的公有 API 相兼容，这样的版本指定确保了我们可以获取能使本章代码编译的最新的补丁（patch）版本。任何大于等于 `0.9.0` 的版本不能保证和接下来的示例采用了相同的 API。

现在，不修改任何代码，构建项目，如示例 2-2 所示。

```
$ cargo build
  Updating crates.io index
   Locking 15 packages to latest Rust 1.85.0 compatible versions
    Adding rand v0.8.5 (available: v0.9.0)
 Compiling proc-macro2 v1.0.93
 Compiling unicode-ident v1.0.17
 Compiling libc v0.2.170
 Compiling cfg-if v1.0.0
 Compiling byteorder v1.5.0
 Compiling getrandom v0.2.15
 Compiling rand_core v0.6.4
 Compiling quote v1.0.38
 Compiling syn v2.0.98
 Compiling zerocopy-derive v0.7.35
 Compiling zerocopy v0.7.35
 Compiling ppv-lite86 v0.2.20
 Compiling rand_chacha v0.3.1
 Compiling rand v0.8.5
 Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.48s
```

可能会出现不同的版本号（多亏了语义化版本，它们与代码是兼容的!），并且显示的行数可能会有所不同（取决于操作系统），行的顺序也可能会不同。

现在我们有了一个外部依赖，Cargo 从 *registry* 上获取所有包的最新版本信息，这是一份来自 Crates.io 的数据副本。Crates.io 是 Rust 生态系统中，人们发布其开源 Rust 项目的平台，供他人使用。

在更新完 *registry* 后，Cargo 检查 `[dependencies]` section 并下载列表中包含但还未下载的 crate。本例中，虽然只声明了 `rand` 一个依赖，然而 Cargo 还是额外获取了 `rand` 所需要的其他 crate，因为 `rand` 依赖它们来正常工作。下载完成后，Rust 编译依赖，然后使用这些依赖编译项目。

如果不做任何修改，立刻再次运行 `cargo build`，则不会看到任何除了 `Finished` 行之外的输出。Cargo 知道它已经下载并编译了依赖，同时 *Cargo.toml* 文件也没有变动。Cargo 还知道代码也没有任何修改，所以它不会重新编译代码。因为无事可做，它会简单地退出。

如果打开 *src/main.rs* 文件，做一些无关紧要的修改，保存并再次构建，你将只会看到两行输出：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
```

这一行表示 Cargo 只针对 *src/main.rs* 文件的微小修改而更新构建。依赖没有变化，所以 Cargo 知道它可以复用已经为此下载并编译的代码。

Cargo.lock 文件确保可重现构建

Cargo 有一个机制，确保无论是你还是其他人在任何时候重新构建代码，都会生成相同的构建产物：Cargo 只会使用你指定的依赖版本，除非你明确指定其他版本。例如，如果下周 `rand` crate 的 `0.8.6` 版本出来了，该版本包含了一个重要的 bug 修复，但同时也引入了一个会破坏你代码的回归问题。为了解决这个问题，Rust 在你第一次运行 `cargo build` 时创建了 *Cargo.lock* 文件，我们现在可以在 *guessing_game* 目录找到它。

当第一次构建项目时，Cargo 计算出所有符合要求的依赖版本并写入 *Cargo.lock* 文件。当将来构建项目时，Cargo 会发现 *Cargo.lock* 已存在并使用其中指定的版本，而不是再次计算所有的版本。这使得你拥有了一个自动化的可重现构建（reproducible build）。换句话说，项目会持续使用 `0.8.5` 直到你显式升级，多亏有了 *Cargo.lock* 文件。由于 *Cargo.lock* 文件对于可重现构建非常重要，因此它通常会和项目中的其余代码一样提交到版本控制系统中。

更新 crate 到一个新版本

当你 **确实** 需要升级 crate 时，Cargo 提供了这样一个命令，`update`，它会忽略 *Cargo.lock* 文件，并计算出所有符合 *Cargo.toml* 声明的最新版本。Cargo 接下来会把这些版本写入 *Cargo.lock* 文件。不过，Cargo 默认只会寻找大于 `0.8.5` 而小于 `0.9.0` 的版本。如果 `rand` crate 发布了两个新版本，`0.8.6` 和 `0.9.0`，在运行 `cargo update` 时会出现如下内容：

```
$ cargo update
    Updating crates.io index
     Locking 1 package to latest Rust 1.85.0 compatible version
    Updating rand v0.8.5 -> v0.8.6 (available: v0.9.0)
```

Cargo 忽略了 `0.9.0` 版本。这时，你也会注意到的 *Cargo.lock* 文件中的变化无外乎现在使用的 `rand` crate 版本是 `0.8.6` 。如果想要使用 `0.9.0` 版本的 `rand` 或是任何 `0.9.x` 系列的版本，必须像这样更新 *Cargo.toml* 文件：

```
[dependencies]
rand = "0.9.0"
```

下一次运行 `cargo build` 时，Cargo 会更新可用 crate 的 registry，并根据你指定的新版本重新评估 `rand` 的要求。

第十四章会讲到 Cargo 及其生态系统 的更多内容，不过目前你只需要了解这么多。通过 Cargo 复用库文件非常容易，因此 Rustacean 能够编写出由很多包组装而成的更轻巧的项目。

生成一个随机数

让我们开始使用 `rand` 来生成一个要猜测的数字。下一步是更新 *src/main.rs*，如示例 2-3 所示。

文件名：src/main.rs

```
use std::io;

use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

首先，我们新增了一行 `use rand::Rng;`。`Rng` 是一个 trait，它定义了随机数生成器应实现的方法，想使用这些方法的话，此 trait 必须在作用域中。第十章会详细介绍 trait。

接下来，我们在中间还新增加了两行。第一行调用了 `rand::thread_rng` 函数提供实际使用的随机数生成器：它位于当前执行线程的本地环境中，并从操作系统获取 seed。接着调用随机数生成器的 `gen_range` 方法。这个方法由 `use rand::Rng` 语句引入到作用域的 `Rng` trait 定义。`gen_range` 方法获取一个范围表达式（range expression）作为参数，并生成一个在此范围之间的随机数。这里使用的这类范围表达式使用了 `start..=end` 这样的形式，它对上下边界均为闭区间，所以需要指定 `1..=100` 来请求一个 1 和 100 之间的数。

> 注意：你不可能凭空就知道应该 use 哪个 trait 以及该从 crate 中调用哪个方法，因此每个 crate 有使用说明文档。Cargo 的另一个很棒的功能是运行 `cargo doc --open` 命令来构建所有本地依赖提供的文档并在浏览器中打开。例如，假设你对 `rand` crate 中的其他功能感兴趣，你可以运行 `cargo doc --open` 并点击左侧导航栏中的 `rand`。

新增加的第二行代码打印出了秘密数字。这在开发程序时很有用，因为可以测试它，不过在最终版本中会删掉它。如果游戏一开始就打印出结果就没什么可玩的了！

尝试运行程序几次：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

你应该能得到不同的随机数，同时它们应该都是在 1 和 100 之间的。干得漂亮！

比较猜测的数字和秘密数字

现在有了用户输入和一个随机数，我们可以比较它们。这个步骤如示例 2-4 所示。注意这段代码还不能通过编译，我们稍后会解释。

文件名：src/main.rs

```
use std::cmp::Ordering;
use std::io;

use rand::Rng;

fn main() {
    // --snip--

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
```

```
    }
}
```



首先我们增加了另一个 `use` 声明，从标准库引入了一个叫做 `std::cmp::Ordering` 的类型到作用域中。 `Ordering` 也是一个枚举，不过它的成员是 `Less`、`Greater` 和 `Equal`。这是比较两个值时可能出现的三种结果。

接着，底部的五行新代码使用了 `Ordering` 类型，`cmp` 方法用来比较两个值并可以在任何可比较的值上调用。它获取一个被比较值的引用：这里是把 `guess` 与 `secret_number` 做比较。然后它会返回一个刚才通过 `use` 引入作用域的 `Ordering` 枚举的成员。使用一个 `match` 表达式，根据对 `guess` 和 `secret_number` 调用 `cmp` 返回的 `Ordering` 成员来决定接下来做什么。

一个 `match` 表达式由 **分支（arms）** 构成。一个分支包含一个 **模式**（*pattern*）和表达式开头的值与分支模式相匹配时应该执行的代码。Rust 获取提供给 `match` 的值并挨个检查每个分支的模式。`match` 结构和模式是 Rust 中强大的功能，它体现了代码可能遇到的多种情形，并确保对所有情况作出处理。这些功能将分别在第六章和第十九章详细介绍。

让我们看看使用 `match` 表达式的例子。假设用户猜了 50，这时随机生成的秘密数字是 38。

比较 50 与 38 时，因为 50 比 38 要大，`cmp` 方法会返回 `Ordering::Greater`。`Ordering::Greater` 是 `match` 表达式得到的值。它检查第一个分支的模式，`Ordering::Less` 与 `Ordering::Greater`并不匹配，所以它忽略了这个分支的代码并来到下一个分支。下一个分支的模式是 `Ordering::Greater`，**正确** 匹配！这个分支关联的代码被执行，在屏幕打印出 `Too big!`。`match` 表达式会在第一次成功匹配后终止，因此在这种情况下不会查看最后一个分支。

然而，示例 2-4 的代码目前并不能编译，可以尝试一下：

```
$ cargo build
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.5
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                 --- ^^^^^^^^^^^^^^^ expected `&String`, found `&{integer}`
   |                 |
   |                 arguments to this method are incorrect
   |
   = note: expected reference `&String`
              found reference `&{integer}`
note: method defined here
  --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/core/src/
cmp.rs:964:8

For more information about this error, try `rustc --explain E0308`.
```

```
error: could not compile `guessing_game` (bin "guessing_game") due to 1 previous
error
```

错误的核心表明这里有 **不匹配的类型**（*mismatched types*）。Rust 有一个静态强类型系统，同时也有类型推断。当我们写出 `let guess = String::new()` 时，Rust 推断出 `guess` 应该是 `String` 类型，并不需要我们写出类型。另一方面，`secret_number`，是数字类型。几个数字类型拥有 1 到 100 之间的值：32 位数字 `i32`；32 位无符号数字 `u32`；64 位数字 `i64` 等等。Rust 默认使用 `i32`，所以它是 `secret_number` 的类型，除非增加类型信息，或任何能让 Rust 推断出不同数值类型的信息。这里错误的原因在于 Rust 不会比较字符串类型和数字类型。

所以我们必须把从输入中读取到的 `String` 转换为一个数字类型，才好与秘密数字进行比较。这可以通过在 `main` 函数体中增加如下代码来实现：

文件名：src/main.rs

```
    // --snip--

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse().expect("Please type a number!");

    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
```

这行新代码是：

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

这里创建了一个叫做 `guess` 的变量。不过等等，不是已经有了一个叫做 `guess` 的变量了吗？确实如此，不过 Rust 允许用一个新值来 **遮蔽**（*Shadowing*）`guess` 之前的值。这个功允许我们复用 `guess` 变量的名字，而不是被迫创建两个不同变量，诸如 `guess_str` 和 `guess` 之类。第三章会介绍 shadowing 的更多细节，现在只需知道这个功能经常用于将一个类型的值转换为另一个类型的值。

我们将这个新变量绑定到 `guess.trim().parse()` 表达式上。表达式中的 `guess` 指的是包含输入的字符串类型 `guess` 变量。`String` 实例的 `trim` 方法会去除字符串开头和结尾的空白字符，我们必须执行此方法才能将字符串与 `u32` 比较，因为 `u32` 只能包含数值型数据。用户必须输入 enter 键才能让 `read_line` 返回并输入他们的猜想，这将会在字符串中增加一个换行（newline）符。例如，用户输入 5 并按下 enter（在 Windows 上，按下 enter 键会得到一个回车符和一个换行符，`\r\n`），`guess` 看起来像这样：`5\n` 或者 `5\r\n`。`\n` 代表 “换行”，回车键；`\r` 代表 “回车”，回车键。`trim` 方法会消除 `\n` 或者 `\r\n`，结果只留下 `5`。

字符串的 `parse` 方法 将字符串转换成其他类型。这里用它来把字符串转换为数值。我们需要告诉 Rust 具体的数字类型，这里通过 `let guess: u32` 指定。`guess` 后面的冒号（`:`）告诉 Rust 我们指定了变量的类型。Rust 有一些内建的数字类型；`u32` 是一个无符号的 32 位整型。对于不大的正整数来说，它是不错的默认类型，第三章还会讲到其他数字类型。

另外，程序中的 `u32` 注解以及与 `secret_number` 的比较，意味着 Rust 会推断出 `secret_number` 也是 `u32` 类型。现在可以使用相同类型比较两个值了！

`parse` 方法只有在字符逻辑上可以转换为数字的时候才能工作所以非常容易出错。例如，字符串中包含 `A👍%`，就无法将其转换为一个数字。因此，`parse` 方法返回一个 `Result` 类型。像之前 “使用 `Result` 类型来处理潜在的错误” 讨论的 `read_line` 方法那样，再次按部就班的用 `expect` 方法处理即可。如果 `parse` 不能从字符串生成一个数字，返回一个 `Result` 的 `Err` 成员时，`expect` 会使游戏崩溃并打印附带的信息。如果 `parse` 成功地将字符串转换为一个数字，它会返回 `Result` 的 `Ok` 成员，然后 `expect` 会返回 `Ok` 值中的数字。

现在让我们运行程序！

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.26s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

漂亮！即便是在猜测之前添加了空格，程序依然能判断出用户猜测了 76。多运行程序几次，输入不同的数字来检验不同的行为：猜一个正确的数字，猜一个过大的数字和猜一个过小的数字。

现在游戏已经大体上能玩了，不过用户只能猜一次。增加一个循环来改变它吧！

使用循环来允许多次猜测

`loop` 关键字创建了一个无限循环。我们会增加循环来给用户更多机会猜数字：

文件名：src/main.rs

```
    // --snip--

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
```

```
        }
    }
}
```

如上所示，我们将提示用户猜测之后的所有内容移动到了循环中。确保 loop 循环中的代码多缩进四个空格，再次运行程序。注意这里有一个新问题，程序现在会不断地要求用户输入新的猜测。用户好像无法退出啊！

用户总能使用 ctrl-c 终止程序。不过还有另一个方法跳出无限循环，就是 “比较猜测与秘密数字” 部分提到的 `parse`：如果用户输入的答案不是一个数字，程序会崩溃。我们可以利用这一点来退出，如下所示：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.23s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit

thread 'main' panicked at src/main.rs:28:47:
Please type a number!: ParseIntError { kind: InvalidDigit }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

输入 `quit` 将会退出程序，同时你会注意到其他任何非数字输入也一样。这至少可以说是不理想的，我们想要当猜测正确的数字时游戏停止。

猜测正确后退出

让我们增加一个 `break` 语句，在用户猜对时退出游戏：

文件名：src/main.rs

```
        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
```

```
            }
        }
    }
}
```

通过在 `You win!` 之后增加一行 `break`，用户猜对了神秘数字后会退出循环。退出循环也意味着退出程序，因为循环是 `main` 的最后一部分。

处理无效输入

为了进一步改善游戏性，不要在用户输入非数字时崩溃，需要忽略非数字，让用户可以继续猜测。可以通过修改 `guess` 将 `String` 转化为 `u32` 那部分代码来实现，如示例 2-5 所示：

文件名：src/main.rs

```
        // --snip--

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        // --snip--
```

我们将 `expect` 调用换成 `match` 语句，以从遇到错误就崩溃转换为处理错误。须知 `parse` 返回一个 `Result` 类型，而 `Result` 是一个拥有 `Ok` 或 `Err` 成员的枚举。这里使用的 `match` 表达式，和之前处理 `cmp` 方法返回 `Ordering` 时用的一样。

如果 `parse` 能够成功地将字符串转换为一个数字，它会返回一个包含结果数字的 `Ok`。这个 `Ok` 值与 `match` 第一个分支的模式相匹配，该分支对应的动作返回 `Ok` 值中的数字 `num`，最后如愿变成新创建的 `guess` 变量。

如果 `parse` **不**能将字符串转换为一个数字，它会返回一个包含更多错误信息的 `Err`。`Err` 值不能匹配第一个 `match` 分支的 `Ok(num)` 模式，但是会匹配第二个分支的 `Err(_)` 模式：`_` 是一个通配符值，本例中用来匹配所有 `Err` 值，不管其中有何种信息。所以程序会执行第二个分支的动作，`continue` 意味着进入 `loop` 的下一次循环，请求另一个猜测。这样程序就有效的忽略了 `parse` 可能遇到的所有错误！

现在程序中的一切都应该如预期般工作了。让我们试试吧：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
```

```
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

太棒了！再有最后一个小的修改，就能完成猜数字游戏了：还记得程序依然会打印出秘密数字。在测试时还好，但正式发布时会毁了游戏体验。删掉打印秘密数字的 `println!`。示例 2-6 为最终代码：

文件名：src/main.rs

```rust
use std::cmp::Ordering;
use std::io;

use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
```

```
    }
}
```

此时此刻，你顺利完成了猜数字游戏。恭喜！

总结

本项目通过动手实践，向你介绍了 Rust 新概念：`let`、`match`、函数、使用外部 crate 等等，接下来的几章，你会继续深入学习这些概念。第三章介绍大部分编程语言都有的概念，比如变量、数据类型和函数，以及如何在 Rust 中使用它们。第四章探索所有权（ownership），这是一个 Rust 同其他语言大不相同的特性。第五章讨论结构体和方法的语法，而第六章解释枚举。

常见编程概念

本章介绍一些几乎所有编程语言都有的概念，以及它们在 Rust 中是如何工作的。很多编程语言的核心概念都是共通的，本章中展示的概念都不是 Rust 所特有的，不过我们会在 Rust 上下文中讨论它们，并解释使用这些概念的惯例。

具体来说，我们将会学习变量、基本类型、函数、注释和控制流。每一个 Rust 程序中都会用到这些基础知识，提早学习这些概念会让你在起步时就打下坚实的基础。

关键字

Rust 语言有一组保留的 **关键字**（*keywords*），就像大部分语言一样，它们只能由语言本身使用。记住，你不能使用这些关键字作为变量或函数的名称。大部分关键字有特殊的意义，你将在 Rust 程序中使用它们完成各种任务；一些关键字目前没有相应的功能，是为将来可能添加的功能保留的。可以在附录 A 中找到关键字的列表。

变量和可变性

正如第二章中“使用变量储存值” 部分提到的那样，变量默认是不可改变的（immutable）。这是 Rust 提供给你的众多优势之一，让你得以充分利用 Rust 提供的安全性和简单并发性来编写代码。不过，你仍然可以使用可变变量。让我们探讨一下 Rust 为何及如何鼓励你利用不可变性，以及何时你会选择禁用它。

当变量不可变时，一旦值被绑定一个名称上，你就不能改变这个值。为了对此进行说明，使用 `cargo new variables` 命令在 *projects* 目录生成一个叫做 *variables* 的新项目。

接着，在新建的 *variables* 目录，打开 *src/main.rs* 并将代码替换为如下代码，这些代码还不能编译，我们会首次检查到不可变错误（immutability error）：

文件名：src/main.rs



```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

保存并使用 `cargo run` 运行程序。应该会看到一条与不可变性有关的错误信息，如下输出所示：

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 5;
  |         +++

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

这个例子展示了编译器如何帮助你找出程序中的错误。虽然编译错误令人沮丧，但那只是表示程序不能安全的完成你想让它完成的工作；并**不能**说明你不是一个好程序员！经验丰富的 Rustacean 们一样会遇到编译错误。

错误信息指出错误的原因是 `不能对不可变变量 x 二次赋值`（`cannot assign twice to immutable variable `x``），因为你尝试对不可变变量 `x` 赋第二个值。

在尝试改变预设为不可变的值时，产生编译时错误是很重要的，因为这种情况可能导致 bug。如果一部分代码假设一个值永远也不会改变，而另一部分代码改变了这个值，第一部分代码就

有可能以不可预料的方式运行。不得不承认这种 bug 的起因难以跟踪，尤其是第二部分代码只是 **有时** 会改变值。

Rust 编译器保证，如果声明一个值不会变，它就真的不会变，所以你不必自己跟踪它。这意味着你的代码更易于推导。

不过可变性也是非常有用的，可以用来更方便地编写代码。尽管变量默认是不可变的，你仍然可以在变量名前添加 `mut` 来使其可变，正如在第二章所做的那样。`mut` 也向读者表明了其他代码将会改变这个变量值的意图。

例如，让我们将 *src/main.rs* 修改为如下代码：

文件名：src/main.rs

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

现在运行这个程序，会出现如下内容：

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

通过 `mut`，允许把绑定到 `x` 的值从 `5` 改成 `6`。是否让变量可变的最终决定权仍然在你，取决于在某个特定情况下，你是否认为变量可变会让代码更加清晰明了。

常量

类似于不可变变量，常量 *(constants)* 是绑定到一个名称的不允许改变的值，不过常量与变量还是有一些区别。

首先，不允许对常量使用 `mut`。常量不光默认不可变，它总是不可变。声明常量使用 `const` 关键字而不是 `let`，并且 必须 注明值的类型。在下一部分，“数据类型” 中会介绍类型和类型注解，现在无需关心这些细节，记住总是标注类型即可。

常量可以在任何作用域中声明，包括全局作用域，这在一个值需要被很多部分的代码用到时很有用。

最后一个区别是，常量只能被设置为常量表达式，而不可以是其他任何只能在运行时计算出的值。

下面是一个声明常量的例子：

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

常量的名称是 `THREE_HOURS_IN_SECONDS`，它的值被设置为 60（一分钟内的秒数）乘以 60（一小时内的分钟数）再乘以 3（我们在这个程序中要计算的小时数）的结果。Rust 对常量的命名

约定是在单词之间使用全大写加下划线。编译器能够在编译时计算一组有限的操作，这使我们可以选择以更容易理解和验证的方式写出此值，而不是将此常量设置为值 10,800。有关声明常量时可以使用哪些操作的详细信息，请参阅 Rust Reference 的常量求值部分。

在声明它的作用域之中，常量在整个程序生命周期中都有效，此属性使得常量可以作为多处代码使用的全局范围的值，例如一个游戏中所有玩家可以获取的最高分或者光速。

将遍布于应用程序中的硬编码值声明为常量，能帮助后来的代码维护人员了解值的意图。如果将来需要修改硬编码值，也只需修改汇聚于一处的硬编码值。

遮蔽

正如在第二章猜数字游戏中所讲，我们可以定义一个与之前变量同名的新变量。Rustacean 们称之为第一个变量被第二个 **遮蔽（Shadowing）** 了，这意味着当您使用变量的名称时，编译器将看到第二个变量。实际上，第二个变量遮蔽了第一个变量，此时任何使用该变量名的行为中都会视为是在使用第二个变量，直到第二个变量自己也被遮蔽或第二个变量的作用域结束。可以用相同变量名称来遮蔽一个变量，以及重复使用 `let` 关键字来多次遮蔽，如下所示：

文件名：src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

这个程序首先将 `x` 绑定到值 `5` 上。接着通过 `let x =` 创建了一个新变量 `x`，获取初始值并加 `1`，这样 `x` 的值就变成 `6` 了。然后，在使用花括号创建的内部作用域内，第三个 `let` 语句也遮蔽了 `x` 并创建了一个新的变量，将之前的值乘以 `2`，`x` 得到的值是 `12`。当该作用域结束时，内部遮蔽的作用域也结束了，`x` 又返回到 `6`。运行这个程序，它会有如下输出：

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

遮蔽与将变量标记为 `mut` 是有区别的。当不小心尝试对变量重新赋值时，如果没有使用 `let` 关键字，就会导致编译时错误。通过使用 `let`，我们可以用这个值进行一些计算，不过计算完之后变量仍然是不可变的。

`mut` 与遮蔽的另一个区别是，当再次使用 `let` 时，实际上创建了一个新变量，我们可以改变值的类型，并且复用这个名字。例如，假设程序请求用户输入空格字符来说明希望在文本之间显示多少个空格，接下来我们想将输入存储成数字（多少个空格）：

```
    let spaces = "   ";
    let spaces = spaces.len();
```

第一个 `spaces` 变量是字符串类型，第二个 `spaces` 变量是数字类型。遮蔽使我们不必使用不同的名字，如 `spaces_str` 和 `spaces_num`；相反，我们可以复用 `spaces` 这个更简单的名字。然而，如果尝试使用 `mut`，将会得到一个编译时错误，如下所示：

```
    let mut spaces = "   ";
    spaces = spaces.len();
```



这个错误说明，我们不能改变变量的类型：

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

现在我们已经了解了变量如何工作，让我们看看变量可以拥有的更多数据类型。

数据类型

在 Rust 中，每一个值都有一个特定 **数据类型**（*data type*），这告诉 Rust 它被指定为何种数据，以便明确数据处理方式。我们将看到两类数据类型子集：标量（scalar）和复合（compound）。

记住，Rust 是 **静态类型**（*statically typed*）语言，也就是说在编译时就必须知道所有变量的类型。根据值及其使用方式，编译器通常可以推断出我们想要用的类型。当多种类型均有可能时，比如第二章的 “比较猜测的数字和秘密数字” 使用 `parse` 将 `String` 转换为数字时，必须增加类型注解，像这样：

```
let guess: u32 = "42".parse().expect("Not a number!");
```

如果不像上面的代码这样添加类型注解 `: u32`，Rust 会显示如下错误，这说明编译器需要我们提供更多信息，来了解我们想要的类型：

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |
  = note: cannot satisfy `<_ as FromStr>::Err == _`
help: consider giving `guess` an explicit type
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due to
1 previous error
```

你会看到其它数据类型的各种类型注解。

标量类型

标量（*scalar*）类型代表一个单独的值。Rust 有四种基本的标量类型：整型、浮点型、布尔类型和字符类型。你可能在其他语言中见过它们。让我们深入了解它们在 Rust 中是如何工作的。

整型

整型 是一个没有小数部分的数字。我们在第二章使用过 `u32` 整数类型。该类型声明表明，它关联的值应该是一个占据 32 比特位的无符号整数（有符号整数类型以 `i` 开头而不是 `u`）。表格 3-1 展示了 Rust 内建的整数类型。我们可以使用其中的任一个来声明一个整数值的类型。

表格 3-1: Rust 中的整型

长度	有符号	无符号
8-bit	`i8`	`u8`

16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
架构相关	`isize`	`usize`

每一个变体都可以是有符号或无符号的，并有一个明确的大小。**有符号** 和 **无符号** 代表数字能否为负值，换句话说，这个数字是否有可能是负数（有符号数），或者永远为正而不需要符号（无符号数）。这有点像在纸上书写数字：当需要考虑符号的时候，数字以加号或减号作为前缀；然而，可以安全地假设为正数时，加号前缀通常省略。有符号数以二进制补码形式（two's complement representation） 存储。

每一个有符号的变体可以储存包含从 $-(2^{n-1})$ 到 $2^{n-1} - 1$ 在内的数字，这里 *n* 是变体使用的位数。所以 `i8` 可以储存从 $-(2^7)$ 到 $2^7 - 1$ 在内的数字，也就是从 -128 到 127。无符号的变体可以储存从 0 到 $2^n - 1$ 的数字，所以 `u8` 可以储存从 0 到 $2^8 - 1$ 的数字，也就是从 0 到 255。

另外，`isize` 和 `usize` 类型依赖运行程序的计算机架构：64 位架构上它们是 64 位的，32 位架构上它们是 32 位的。

可以使用表格 3-2 中的任何一种形式编写数字字面值。请注意可以是多种数字类型的数字字面值允许使用类型后缀，例如 `57u8` 来指定类型，同时也允许使用 `_` 做为分隔符以方便读数，例如`1_000`，它的值与你指定的 `1000` 相同。

表格 3-2: Rust 中的整型字面值

数字字面值	**例子**
Decimal (十进制)	`98_222`
Hex (十六进制)	`0xff`
Octal (八进制)	`0o77`
Binary (二进制)	`0b1111_0000`
Byte (单字节字符)(仅限于`u8`)	`b'A'`

那么该使用哪种类型的数字呢？如果拿不定主意，Rust 的默认类型通常是个不错的起点，整型默认是 `i32`。`isize` 或 `usize` 主要作为某些集合的索引。

整型溢出

比方说有一个 `u8` ，它可以存放从零到 `255` 的值。那么当你将其修改为 `256` 时就会发生 **整型溢出**（*integer overflow*），这会导致以下两种行为之一的发生。当在 debug

> 模式编译时，Rust 检查这类问题并使程序 *panic*。*panic* 这个术语被 Rust 用来表明程序因错误而退出。第九章 “`panic!` 与不可恢复的错误” 部分会详细介绍 panic。
>
> 使用 `--release` flag 在 release 模式中构建时，Rust **不会**检测会导致 panic 的整型溢出。相反发生整型溢出时，Rust 会进行一种被称为二进制补码 wrapping（*two's complement wrapping*）的操作。简而言之，比此类型能容纳最大值还大的值会回绕到最小值，值 `256` 变成 `0`，值 `257` 变成 `1`，依此类推。程序不会 panic，不过变量可能也不会是你所期望的值。依赖整型溢出 wrapping 的行为被认为是一种错误。
>
> 为了显式地处理溢出的可能性，可以使用这几类标准库提供的原始数字类型方法：
>
> - 所有模式下都可以使用 `wrapping_*` 方法进行 wrapping，如 `wrapping_add`
> - 如果 `checked_*` 方法出现溢出，则返回 `None` 值
> - 用 `overflowing_*` 方法返回值和一个布尔值，表示是否出现溢出
> - 用 `saturating_*` 方法在值的最小值或最大值处进行饱和处理

浮点型

Rust 也有两个原生的 **浮点数**（*floating-point numbers*）类型，它们是带小数点的数字。Rust 的浮点数类型是 `f32` 和 `f64`，分别占 32 位和 64 位。默认类型是 `f64`，因为在现代 CPU 中，它与 `f32` 速度几乎一样，不过精度更高。所有的浮点型都是有符号的。

文件名：src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

浮点数采用 IEEE-754 标准表示。（`f32` 是单精度浮点数，`f64` 是双精度浮点数。）

数值运算

Rust 中的所有数字类型都支持基本数学运算：加法、减法、乘法、除法和取余。整数除法会向零舍入到最接近的整数。下面的代码展示了如何在 `let` 语句中使用各种数值运算：

文件名：src/main.rs

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
```

```
    let truncated = -5 / 3; // 结果为 -1

    // remainder
    let remainder = 43 % 5;
}
```

这些语句中的每个表达式使用了一个数学运算符并计算出了一个值，然后绑定给一个变量。附录 B 包含 Rust 提供的所有运算符的列表。

布尔类型

正如其他大部分编程语言一样，Rust 中的布尔类型有两个可能的值：`true` 和 `false`。Rust 中的布尔类型使用 `bool` 表示。例如：

文件名：src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

使用布尔值的主要场景是条件表达式，例如 `if` 表达式。在 “控制流”（“Control Flow”） 部分将介绍 `if` 表达式在 Rust 中如何工作。

字符类型

Rust 的 `char` 类型是语言中最原始的字母类型。下面是一些声明 `char` 值的例子：

文件名：src/main.rs

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

注意，我们用单引号声明 `char` 字面值，而与之相反的是，使用双引号声明字符串字面值。Rust 的 `char` 类型的大小为四个字节 (four bytes)，并代表了一个 Unicode 标量值（Unicode Scalar Value），这意味着它可以比 ASCII 表示更多内容。在 Rust 中，带变音符号的字母（Accented letters），中文、日文、韩文等字符，emoji（绘文字）以及零长度的空白字符都是有效的 `char` 值。Unicode 标量值包含从 `U+0000` 到 `U+D7FF` 和 `U+E000` 到 `U+10FFFF` 在内的值。不过，“字符” 并不是一个 Unicode 中的概念，所以人直觉上的 “字符” 可能与 Rust 中的 `char` 并不符合。第八章的 “使用字符串储存 UTF-8 编码的文本” 中将详细讨论这个主题。

复合类型

复合类型（*Compound types*）可以将多个值组合成一个类型。Rust 有两个原生的复合类型：元组（tuple）和数组（array）。

元组类型

元组是一个将多个不同类型的值组合进一个复合类型的主要方式。元组长度固定：一旦声明，其长度不会增大或缩小。

我们使用包含在圆括号中的逗号分隔的值列表来创建一个元组。元组中的每一个位置都有一个类型，而且这些不同值的类型也不必是相同的。这个例子中使用了可选的类型注解：

文件名：src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

`tup` 变量绑定到整个元组上，因为元组是一个单独的复合元素。为了从元组中获取单个值，可以使用模式匹配（pattern matching）来解构（destructure）元组值，像这样：

文件名：src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

程序首先创建了一个元组并绑定到 `tup` 变量上。接着使用了 `let` 和一个模式将 `tup` 分成了三个不同的变量，`x`、`y` 和 `z`。这叫做 **解构**（*destructuring*），因为它将一个元组拆成了三个部分。最后，程序打印出了 `y` 的值，也就是 `6.4`。

我们也可以使用点号（`.`）后跟值的索引来直接访问所需的元组元素。例如：

文件名：src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

这个程序创建了一个元组，`x`，然后使用其各自的索引访问元组中的每个元素。跟大多数编程语言一样，元组的第一个索引值是 0。

不带任何值的元组有个特殊的名称，叫做 **单元（unit）** 元组。这种值以及对应的类型都写作 `()`，表示空值或空的返回类型。如果表达式不返回任何其他值，则会隐式返回单元值。

数组类型

另一个包含多个值的方式是 **数组**（*array*）。与元组不同，数组中的每个元素的类型必须相同。Rust 中的数组与一些其他语言中的数组不同，Rust 中的数组长度是固定的。

我们将数组的值写成在方括号内，用逗号分隔的列表：

文件名：src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

当你想要在栈（stack）而不是在堆（heap）上为数据分配空间（第四章将讨论栈与堆的更多内容），或者是想要确保总是有固定数量的元素时，数组非常有用。但是数组并不如 vector 类型灵活。vector 类型是标准库提供的一个 **允许** 增长和缩小长度的类似数组的集合类型。当不确定是应该使用数组还是 vector 的时候，那么很可能应该使用 vector。第八章会详细讨论 vector。

然而，当你确定元素个数不会改变时，数组会更有用。例如，当你在一个程序中使用月份名字时，你更应趋向于使用数组而不是 vector，因为你确定只会有 12 个元素。

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

可以像这样编写数组的类型：在方括号中包含每个元素的类型，后跟分号，再后跟数组元素的数量。

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

这里，`i32` 是每个元素的类型。分号之后，数字 `5` 表明该数组包含五个元素。

你还可以通过在方括号中指定初始值加分号再加元素个数的方式来创建一个每个元素都为相同值的数组：

```
let a = [3; 5];
```

变量名为 `a` 的数组将包含 `5` 个元素，这些元素的值最初都将被设置为 `3`。这种写法与 `let a = [3, 3, 3, 3, 3];` 效果相同，但更简洁。

访问数组元素

数组是可以在栈 (stack) 上分配的已知固定大小的单个内存块。可以使用索引来访问数组的元素，像这样：

文件名：src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
```

```
    let second = a[1];
}
```

在这个例子中，叫做 `first` 的变量的值是 `1`，因为它是数组索引 `[0]` 的值。变量 `second` 将会是数组索引 `[1]` 的值 `2`。

无效的数组元素访问

让我们看看如果我们访问数组结尾之后的元素会发生什么呢？比如你执行以下代码，它使用类似于第 2 章中的猜数字游戏的代码从用户那里获取数组索引：

文件名：src/main.rs



```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

此代码编译成功。如果您使用 `cargo run` 运行此代码并输入 `0`、`1`、`2`、`3` 或 `4`，程序将在数组中的索引处打印出相应的值。如果你输入一个超过数组末端的数字，如 10，你会看到这样的输出：

```
thread 'main' panicked at src/main.rs:19:19:
index out of bounds: the len is 5 but the index is 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

程序在索引操作中使用一个无效的值时导致 **运行时** 错误。程序带着错误信息退出，并且没有执行最后的 `println!` 语句。当尝试用索引访问一个元素时，Rust 会检查指定的索引是否小于数组的长度。如果索引超出了数组长度，Rust 会 *panic*，这是 Rust 术语，它用于程序因为错误而退出的情况。这种检查必须在运行时进行，特别是在这种情况下，因为编译器不可能知道用户在以后运行代码时将输入什么值。

这是第一个在实战中遇到的 Rust 安全原则的例子。在很多底层语言中，并没有进行这类检查，这样当提供了一个不正确的索引时，就会访问无效的内存。通过立即退出而不是允许内存访问

并继续执行，Rust 让你避开此类错误。第九章会更详细地讨论 Rust 的错误处理机制，以及如何编写可读性强而又安全的代码，使程序既不会 panic 也不会导致非法内存访问。

函数

函数在 Rust 代码中非常普遍。你已经见过语言中最重要的函数之一：`main` 函数，它是很多程序的入口点。你也见过 `fn` 关键字，它用来声明新函数。

Rust 代码中的函数和变量名使用 *snake case* 规范风格。在 snake case 中，所有字母都是小写并使用下划线分隔单词。这是一个包含函数定义示例的程序：

文件名：src/main.rs

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

我们在 Rust 中通过输入 `fn` 后面跟着函数名和一对圆括号来定义函数。大括号告诉编译器哪里是函数体的开始和结尾。

可以使用函数名后跟圆括号来调用我们定义过的任意函数。因为程序中已定义 `another_function` 函数，所以可以在 `main` 函数中调用它。注意，源码中 `another_function` 定义在 `main` 函数 **之后**；也可以定义在之前。Rust 不关心函数定义所在的位置，只要函数被调用时出现在调用之处可见的作用域内就行。

让我们新建一个叫做 *functions* 的二进制项目来进一步探索函数。将上面的 `another_function` 例子写入 *src/main.rs* 中并运行。你应该会看到如下输出：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/functions`
Hello, world!
Another function.
```

`main` 函数中的代码会按顺序执行。首先，打印 “Hello, world!” 信息，然后调用 `another_function` 函数并打印它的信息。

参数

我们可以定义为拥有 **参数**（*parameters*）的函数，参数是特殊变量，是函数签名的一部分。当函数拥有参数（形参）时，可以为这些参数提供具体的值（实参）。技术上讲，这些具体值被称为参数（*arguments*），但是在日常交流中，人们倾向于不区分使用 *parameter* 和 *argument* 来表示函数定义中的变量或调用函数时传入的具体值。

在这版 `another_function` 中，我们增加了一个参数：

文件名：src/main.rs

```
```

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

尝试运行程序，将会输出如下内容：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
The value of x is: 5
```

`another_function` 的声明中有一个命名为 `x` 的参数。`x` 的类型被指定为 `i32`。当我们将 `5` 传给 `another_function` 时，`println!` 宏会把 `5` 放在格式字符串中包含 `x` 的那对花括号的位置。

在函数签名中，**必须** 声明每个参数的类型。这是 Rust 设计中一个经过慎重考虑的决定：要求在函数定义中提供类型注解，意味着编译器再也不需要你在代码的其他地方注明类型来指出你的意图。而且，在知道函数需要什么类型后，编译器就能够给出更有用的错误消息。

当定义多个参数时，使用逗号分隔，像这样：

文件名：src/main.rs

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

这个例子创建了一个名为 `print_labeled_measurement` 的函数，它有两个参数。第一个参数名为 `value`，类型是 `i32`。第二个参数是 `unit_label` ，类型是 `char`。然后，该函数打印包含 `value` 和 `unit_label` 的文本。

尝试运行代码。使用上面的例子替换当前 *functions* 项目的 *src/main.rs* 文件，并用 `cargo run` 运行它：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

因为我们使用 `5` 作为 `value` 的值，`h` 作为 `unit_label` 的值来调用函数，所以程序输出包含这些值。

语句和表达式

函数体由一系列的语句和一个可选的结尾表达式构成。目前为止，我们提到的函数还不包含结尾表达式，不过你已经见过作为语句一部分的表达式。因为 Rust 是一门基于表达式（expression-based）的语言，这是一个需要理解的重要区别。其他语言并没有这样的区别，所以让我们看看语句与表达式有什么区别以及这些区别是如何影响函数体的。

- **语句**（*Statements*）是执行一些操作但不返回值的指令。
- **表达式**（*Expressions*）计算并产生一个值。

让我们看一些例子。

实际上，我们已经使用过语句和表达式。使用 `let` 关键字创建变量并绑定一个值是一个语句。在示例 3-1 中，`let y = 6;` 是一个语句。

文件名：src/main.rs

```
fn main() {
    let y = 6;
}
```

示例 3-1：包含一个语句的 `main` 函数定义

函数定义也是语句，上面整个例子本身就是一个语句。（不过，如我们将在下面看到，**调用**函数并不是语句。）

语句不返回值。因此，不能把 `let` 语句赋值给另一个变量，比如下面的例子尝试做的，会产生一个错误：

文件名：src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```



当运行这个程序时，会得到如下错误：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
```

```
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |

warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` (bin "functions") due to 1 previous error; 1
warning emitted
```

`let y = 6` 语句并不返回值，所以没有可以绑定到 `x` 上的值。这与其他语言不同，例如 C 和 Ruby，它们的赋值语句会返回所赋的值。在这些语言中，可以这么写 `x = y = 6`，这样 `x` 和 `y` 的值都是 `6`；Rust 中不能这样写。

表达式会计算出一个值，并且你将编写的大部分 Rust 代码是由表达式组成的。考虑一个数学运算，比如 `5 + 6`，这是一个表达式并计算出值 `11`。表达式可以是语句的一部分：在示例 3-1 中，语句 `let y = 6;` 中的 `6` 是一个表达式，它计算出的值是 `6`。函数调用是一个表达式。宏调用是一个表达式。用大括号创建的一个新的块作用域也是一个表达式，例如：

文件名：src/main.rs

```rust
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

这个表达式：

```rust
{
    let x = 3;
    x + 1
}
```

是一个代码块，它的值是 `4`。这个值作为 `let` 语句的一部分被绑定到 `y` 上。注意 `x + 1` 这一行在结尾没有分号，与你见过的大部分代码行不同。表达式的结尾没有分号。如果在表达式的结尾加上分号，它就变成了语句，而语句不会返回值。在接下来探索具有返回值的函数和表达式时要谨记这一点。

具有返回值的函数

函数可以向调用它的代码返回值。我们并不对返回值命名，但要在箭头（`->`）后声明它的类型。在 Rust 中，函数的返回值等同于函数体最后一个表达式的值。使用 `return` 关键字和指定值，可从函数中提前返回；但大部分函数隐式的返回最后的表达式。这是一个有返回值的函数的例子：

文件名：src/main.rs

```
```

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

在 `five` 函数中没有函数调用、宏、甚至没有 `let` 语句 —— 只有数字 `5`。这在 Rust 中是一个完全有效的函数。注意，也指定了函数返回值的类型，就是 `-> i32`。尝试运行代码；输出应该看起来像这样：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
The value of x is: 5
```

`five` 函数的返回值是 `5`，所以返回值类型是 `i32`。让我们仔细检查一下这段代码。有两个重要的部分：首先，`let x = five();` 这一行表明我们使用函数的返回值初始化一个变量。因为 `five` 函数返回 `5`，这一行与如下代码相同：

```
let x = 5;
```

其次，`five` 函数没有参数并定义了返回值类型，不过函数体只有单单一个 `5` 也没有分号，因为这是一个表达式，我们想要返回它的值。

让我们看看另一个例子：

文件名：src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

运行代码会打印出 `The value of x is: 6`。但如果在包含 `x + 1` 的行尾加上一个分号，把它从表达式变成语句，我们将看到一个错误。

文件名：src/main.rs

```
fn main() {
    let x = plus_one(5);
```

```
    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```



运行代码会产生一个错误，如下：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` (bin "functions") due to 1 previous error
```

主要的错误信息，“mismatched types”（类型不匹配），揭示了代码的核心问题。函数 `plus_one` 的定义说明它要返回一个 `i32` 类型的值，不过语句并不会返回值，使用单位类型 `()` 表示不返回值。因为不返回值与函数定义相矛盾，从而出现一个错误。在输出中，Rust 提供了一条信息，可能有助于纠正这个错误：它建议删除分号，这会修复这个错误。

注释

所有程序员都力求使其代码易于理解，不过有时还需要提供额外的解释。在这种情况下，程序员在源码中留下 **注释**（*comments*），编译器会忽略它们，不过阅读代码的人可能觉得有用。

这是一个简单的注释：

```
// hello, world
```

在 Rust 中，惯用的注释样式是以两个斜杠开始注释，并持续到本行的结尾。对于超过一行的注释，需要在每一行前都加上 `//`，像这样：

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
```

注释也可以放在包含代码的行的末尾：

文件名：src/main.rs

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today
}
```

不过你更经常看到的是以这种格式使用它们，也就是位于它所解释的代码行的上面一行：

文件名：src/main.rs

```
fn main() {
    // I'm feeling lucky today
    let lucky_number = 7;
}
```

Rust 还有另一种注释，称为文档注释，我们将在 14 章的 “将 crate 发布到 Crates.io” 部分讨论它。

控制流

根据条件是否为真来决定是否执行某些代码，以及根据条件是否为真来重复运行一段代码的能力是大部分编程语言的基本组成部分。Rust 代码中最常见的用来控制执行流的结构是 `if` 表达式和循环。

`if` 表达式

`if` 表达式允许根据条件执行不同的代码分支。你提供一个条件并表示 “如果条件满足，运行这段代码；如果条件不满足，不运行这段代码。”

在 *projects* 目录新建一个叫做 *branches* 的项目，来学习 `if` 表达式。在 *src/main.rs* 文件中，输入如下内容：

文件名：src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

所有的 `if` 表达式都以 `if` 关键字开头，其后跟一个条件。在这个例子中，条件检查变量 `number` 的值是否小于 5。在条件为 `true` 时希望执行的代码块位于紧跟条件之后的大括号中。`if` 表达式中与条件关联的代码块有时被叫做 *arms*，就像第二章 “比较猜测的数字和秘密数字” 部分中讨论到的 `match` 表达式中的分支一样。

也可以包含一个可选的 `else` 表达式来提供一个在条件为 `false` 时应当执行的代码块，这里我们就这么做了。如果不提供 `else` 表达式并且条件为 `false` 时，程序会直接忽略 `if` 代码块并继续执行下面的代码。

尝试运行代码，应该能看到如下输出：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was true
```

尝试改变 `number` 的值使条件为 `false` 时看看会发生什么：

```
    let number = 7;
```

再次运行程序并查看输出：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was false
```

另外值得注意的是代码中的条件**必须**是 `bool` 值。如果条件不是 `bool` 值，我们将得到一个错误。例如，尝试运行以下代码：

文件名：src/main.rs



```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

这里 `if` 条件的值是 `3`，Rust 抛出了一个错误：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

这个错误表明 Rust 期望一个 `bool` 却得到了一个整数。不像 Ruby 或 JavaScript 这样的语言，Rust 并不会尝试自动地将非布尔值转换为布尔值。必须总是显式地使用布尔值作为 `if` 的条件。例如，如果想要 `if` 代码块只在一个数字不等于 `0` 时执行，可以把 `if` 表达式修改成下面这样：

文件名：src/main.rs

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

运行代码会打印出 `number was something other than zero`。

使用 `else if` 处理多重条件

可以将 `else if` 表达式与 `if` 和 `else` 组合来实现多重条件。例如：

文件名：src/main.rs

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

这个程序有四个可能的执行路径。运行后应该能看到如下输出：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
number is divisible by 3
```

当执行这个程序时，它按顺序检查每个 `if` 表达式并执行第一个条件为 `true` 的代码块。注意即使 6 可以被 2 整除，也不会输出 `number is divisible by 2`，更不会输出 `else` 块中的 `number is not divisible by 4, 3, or 2`。原因是 Rust 只会执行第一个条件为 `true` 的代码块，并且一旦它找到一个以后，甚至都不会检查剩下的条件了。

使用过多的 `else if` 表达式会使代码显得杂乱无章，所以如果有多于一个 `else if` 表达式，最好重构代码。为此，第六章会介绍一个强大的 Rust 分支结构（branching construct），叫做 `match`。

在 `let` 语句中使用 `if`

因为 `if` 是一个表达式，我们可以在 `let` 语句的右侧使用它，例如在示例 3-2 中：

文件名：src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

示例 3-2：将 `if` 表达式的返回值赋给一个变量

`number` 变量将会绑定到表示 `if` 表达式结果的值上。运行这段代码看看会出现什么：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
The value of number is: 5
```

记住，代码块的值是其最后一个表达式的值，而数字本身就是一个表达式。在这个例子中，整个 `if` 表达式的值取决于哪个代码块被执行。这意味着 `if` 的每个分支的可能的返回值都必须是相同类型；在示例 3-2 中，`if` 分支和 `else` 分支的结果都是 `i32` 整型。如果它们的类型不匹配，如下面这个例子，则会出现一个错误：

文件名：src/main.rs



```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

当编译这段代码时，会得到一个错误。`if` 和 `else` 分支的值类型是不相容的，同时 Rust 也准确地指出在程序中的何处发现的这个问题：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                -           ^^^^^ expected integer, found
`&str`
  |                                |
  |                                expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

`if` 代码块中的表达式返回一个整数，而 `else` 代码块中的表达式返回一个字符串。这不可行，因为变量必须只有一个类型。Rust 需要在编译时就确切的知道 `number` 变量的类型，这样它就可以在编译时验证在每处使用的 `number` 变量的类型是有效的。如果`number`的类型仅在运行时确定，则 Rust 无法做到这一点；且编译器必须跟踪每一个变量的多种假设类型，那么它就会变得更加复杂，对代码的保证也会减少。

使用循环重复执行

多次执行同一段代码是很常用的，Rust 为此提供了多种 **循环**（*loops*）。一个循环执行循环体中的代码直到结尾并紧接着回到开头继续执行。为了实验一下循环，让我们新建一个叫做 *loops* 的项目。

Rust 有三种循环：`loop`、`while` 和 `for`。我们每一个都试试。

使用 `loop` 重复执行代码

`loop` 关键字告诉 Rust 一遍又一遍地执行一段代码直到你明确要求停止。

作为一个例子，将 *loops* 目录中的 *src/main.rs* 文件修改为如下：

文件名：src/main.rs

```
fn main() {
    loop {
        println!("again!");
    }
}
```

当运行这个程序时，我们会看到连续的反复打印 `again!`，直到我们手动停止程序。大部分终端都支持键盘快捷键 ctrl-c 来终止一个陷入无限循环的程序。尝试一下：

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

符号 `^C` 代表你在这按下了 ctrl-c。在 `^C` 之后你可能看到也可能看不到 `again!`，这取决于在接收到终止信号时代码执行到了循环的何处。

幸运的是，Rust 提供了一种从代码中跳出循环的方法。可以使用 `break` 关键字来告诉程序何时停止循环。回忆一下在第二章猜猜看游戏的 “猜测正确后退出” 部分使用过它来在用户猜对数字赢得游戏后退出程序。

我们在猜谜游戏中也使用了 `continue`。循环中的 `continue` 关键字告诉程序跳过这个循环迭代中的任何剩余代码，并转到下一个迭代。

从循环返回值

`loop` 的一个用例是重试可能会失败的操作，比如检查线程是否完成了任务。然而你可能会需要将操作的结果传递给其它的代码。要实现这一点，可以在用于停止循环的 `break` 表达式后添加你希望返回的值；这个值就会作为循环的返回值返回，这样你就可以使用它，如下所示：

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

在循环之前，我们声明了一个名为 `counter` 的变量并初始化为 `0`。接着声明了一个名为 `result` 来存放循环的返回值。在循环的每一次迭代中，我们将 `counter` 变量加 `1`，接着检查

计数是否等于 `10`。当相等时，使用 `break` 关键字返回值 `counter * 2`。循环之后，我们通过分号结束赋值给 `result` 的语句。最后打印出 `result` 的值，也就是 `20`。

循环标签：在多个循环之间消除歧义

如果存在嵌套循环，`break` 和 `continue` 应用于此时最内层的循环。你可以选择在一个循环上指定一个 **循环标签**（*loop label*），然后将标签与 `break` 或 `continue` 一起使用，使这些关键字应用于已标记的循环而不是最内层的循环。下面是一个包含两个嵌套循环的示例：

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

外层循环有一个标签 `counting_up`，它将从 0 数到 2。没有标签的内部循环从 10 向下数到 9。第一个没有指定标签的 `break` 将只退出内层循环。`break 'counting_up;` 语句将退出外层循环。这个代码打印：

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

`while` 条件循环

在程序中计算循环的条件也很常见。当条件为 `true`，执行循环。当条件不再为 `true`，调用 `break` 停止循环。这个循环类型可以通过组合 `loop`、`if`、`else` 和 `break` 来实现；如果你喜欢的话，现在就可以在程序中试试。然而，这个模式太常用了，Rust 为此内置了一个语言结构，

它被称为 `while` 循环。在示例 3-3 中，使用了 `while` 程序循环三次，每次数字都减一。接着，在循环结束后，打印出另一个信息并退出。

文件名：src/main.rs

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

示例 3-3: 当条件为 `true` 时，使用 `while` 循环运行代码

这种结构消除了很多使用 `loop`、`if`、`else` 和 `break` 时所必须的嵌套，这样更加清晰。当条件为 `true` 就执行，否则退出循环。

使用 `for` 遍历集合

可以使用 `while` 结构来遍历集合中的元素，比如数组。例如，示例 3-4 中的循环会打印数组 `a` 中的每一个元素。

文件名：src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

示例 3-4：使用 `while` 循环遍历集合中的每一个元素

这里，代码对数组中的元素进行计数。它从索引 `0` 开始，并接着循环直到遇到数组的最后一个索引（这时，`index < 5` 不再为 `true`）。运行这段代码会打印出数组中的每一个元素：

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

数组中的所有五个元素都如期出现在终端中。尽管 `index` 在某一时刻会到达值 `5`，不过循环在其尝试从数组获取第六个值（会越界）之前就停止了。

但这个过程很容易出错；如果索引长度或测试条件不正确会导致程序 panic。例如，如果将 `a` 数组的定义改为包含 4 个元素而忘记了更新条件 `while index < 4`，则代码会 panic。这也使程序更慢，因为编译器增加了运行时代码来对每次循环进行条件检查，以确定在循环的每次迭代中索引是否在数组的边界内。

作为更简洁的替代方案，可以使用 `for` 循环来对一个集合的每个元素执行一些代码。`for` 循环看起来如示例 3-5 所示：

文件名：src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

示例 3-5：使用 `for` 循环遍历集合中的元素

当运行这段代码时，将看到与示例 3-4 一样的输出。更为重要的是，我们增强了代码安全性，并消除了可能由于超出数组的结尾或遍历长度不够而缺少一些元素而导致的 bug。

例如，在示例 3-4 的代码中，如果你将 `a` 数组的定义改为有四个元素，但忘记将条件更新为 `while index < 4`，代码将会 panic。使用 `for` 循环的话，就不需要惦记着在改变数组元素个数时修改其他的代码了。

`for` 循环的安全性和简洁性使得它成为 Rust 中使用最多的循环结构。即使是在想要循环执行代码特定次数时，例如示例 3-3 中使用 `while` 循环的倒计时例子，大部分 Rustacean 也会使用 `for` 循环。这么做的方式是使用 `Range`，它是标准库提供的类型，用来生成从一个数字开始到另一个数字之前结束的所有数字的序列。

下面是一个使用 `for` 循环来倒计时的例子，它还使用了一个我们还未讲到的方法，`rev`，用来反转 range。

文件名：src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

这段代码看起来更帅气不是吗？

总结

你做到了！这是一个大章节：你学习了变量、标量和复合数据类型、函数、注释、 `if` 表达式和循环！如果你想要实践本章讨论的概念，尝试构建如下程序：

- 相互转换摄氏与华氏温度。
- 生成第 n 个斐波那契数。
- 打印圣诞颂歌 “The Twelve Days of Christmas” 的歌词，并利用歌曲中的重复部分（编写循环）。

当你准备好继续的时候，让我们讨论一个其他语言中**并不**常见的概念：所有权（ownership）。

认识所有权

所有权（系统）是 Rust 最为与众不同的特性，对语言的其他部分有着深刻含义。它让 Rust 无需垃圾回收（garbage collector）即可保障内存安全，因此理解 Rust 中所有权如何工作是十分重要的。本章，我们将讲到所有权以及相关功能：借用（borrowing）、slice 以及 Rust 如何在内存中布局数据。

什么是所有权?

所有权（*ownership*）是 Rust 用于如何管理内存的一组规则。所有程序都必须管理其运行时使用计算机内存的方式。一些语言中具有垃圾回收机制，在程序运行时有规律地寻找不再使用的内存；在另一些语言中，程序员必须亲自分配和释放内存。Rust 则选择了第三种方式：通过所有权系统管理内存，编译器在编译时会根据一系列的规则进行检查。如果违反了任何这些规则，程序都不能编译。在运行时，所有权系统的任何功能都不会减慢程序的运行。

因为所有权对很多程序员来说都是一个新概念，需要一些时间来适应。好消息是随着你对 Rust 和所有权系统的规则越来越有经验，你就越能自然地编写出安全和高效的代码。持之以恒!

当你理解了所有权，你将有一个坚实的基础来理解那些使 Rust 独特的功能。在本章中，你将通过完成一些示例来学习所有权，这些示例基于一个常用的数据结构：字符串。

> **栈（Stack）与堆（Heap）**
>
> 在很多语言中，你并不需要经常考虑到栈与堆。不过在像 Rust 这样的系统编程语言中，值是位于栈上还是堆上在更大程度上影响了语言的行为以及为何必须做出这样的抉择。我们会在本章的稍后部分描述所有权与栈和堆相关的内容，所以这里只是一个用来预热的简要解释。
>
> 栈和堆都是代码在运行时可供使用的内存，但是它们的结构不同。栈以放入值的顺序存储值并以相反顺序取出值。这也被称作 **后进先出**（*last in, first out*）。想象一下一叠盘子：当增加更多盘子时，把它们放在盘子堆的顶部，当需要盘子时，也从顶部拿走。不能从中间也不能从底部增加或拿走盘子！增加数据叫做 **入栈**（*pushing onto the stack*），而移出数据叫做 **出栈**（*popping off the stack*）。栈中的所有数据都必须占用已知且固定的大小。在编译时大小未知或大小可能变化的数据，要改为存储在堆上。
>
> 堆是缺乏组织的：当向堆放入数据时，你要请求一定大小的空间。内存分配器（memory allocator）在堆的某处找到一块足够大的空位，把它标记为已使用，并返回一个表示该位置地址的 **指针**（*pointer*）。这个过程称作 **在堆上分配内存**（*allocating on the heap*），有时简称为 “分配”（allocating）。（将数据推入栈中并不被认为是分配）。因为指向放入堆中数据的指针是已知的并且大小是固定的，你可以将该指针存储在栈上，不过当需要实际数据时，必须访问指针。想象一下去餐馆就座吃饭。当进入时，你说明有几个人，餐馆员工会找到一个够大的空桌子并领你们过去。如果有人来迟了，他们也可以通过询问来找到你们坐在哪。
>
> 入栈比在堆上分配内存要快，因为（入栈时）分配器无需为存储新数据去搜索内存空间；其位置总是在栈顶。相比之下，在堆上分配内存则需要更多的工作，这是因为分配器必须首先找到一块足够存放数据的内存空间，并接着做一些记录为下一次分配做准备。
>
> 访问堆上的数据比访问栈上的数据慢，因为必须通过指针来访问。现代处理器在内存中跳转越少就越快。继续类比，假设有一个服务员在餐厅里处理多个桌子的点菜。在一个桌子报完所有菜后再移动到下一个桌子是最有效率的。从桌子 A 听一个菜，接着桌子 B 听一个菜，然后再桌子 A，然后再桌子 B 这样的流程会更加缓慢。出于同样原

> 因，处理器在处理的数据彼此较近的时候（比如在栈上）比较远的时候（比如可能在堆上）更高效。
>
> 当你的代码调用一个函数时，传递给函数的值（包括可能指向堆上数据的指针）和函数的局部变量被压入栈中。当函数结束时，这些值被移出栈。
>
> 跟踪哪部分代码正在使用堆上的哪些数据，最大限度的减少堆上的重复数据的数量，以及清理堆上不再使用的数据确保不会耗尽空间，这些问题正是所有权系统要处理的。一旦理解了所有权，你就不需要经常考虑栈和堆了，不过明白了所有权的主要目的就是管理堆数据，能够帮助解释为什么所有权要以这种方式工作。

所有权规则

首先，让我们看一下所有权的规则。当我们通过举例说明时，请谨记这些规则：

1. Rust 中的每一个值都有一个 **所有者**（*owner*）。
2. 值在任一时刻有且只有一个所有者。
3. 当所有者离开作用域，这个值将被丢弃。

变量作用域

既然我们已经掌握了基本语法，将不会在之后的例子中包含 `fn main() {` 代码，所以如果你是一路跟过来的，必须手动将之后例子的代码放入一个 `main` 函数中。这样，例子将显得更加简明，使我们可以关注实际细节而不是样板代码。

在所有权的第一个例子中，我们看看一些变量的 **作用域**（*scope*）。作用域是一个项（item）在程序中有效的范围。假设有这样一个变量：

```
let s = "hello";
```

变量 `s` 绑定到了一个字符串字面值，这个字符串值是硬编码进程序代码中的。这个变量从声明的点开始直到当前**作用域**结束时都是有效的。示例 4-1 中的注释标明了变量 `s` 在何处是有效的。

```
    {                      // s 在这里无效，它尚未声明
        let s = "hello";   // 从此处起，s 是有效的

        // 使用 s
    }                      // 此作用域已结束，s 不再有效
```

示例 4-1：一个变量和其有效的作用域

换句话说，这里有两个重要的时间点：

- 当 `s` **进入作用域**时，它就是有效的。
- 这一直持续到它**离开作用域**为止。

目前为止，变量是否有效与作用域的关系跟其他编程语言是类似的。现在我们在此基础上介绍 `String` 类型。

`String` 类型

为了演示所有权的规则，我们需要一个比第三章 “数据类型” 中讲到的都要复杂的数据类型。前面介绍的类型都是已知大小的，可以存储在栈中，并且当离开作用域时被移出栈，如果代码的另一部分需要在不同的作用域中使用相同的值，可以快速简单地复制它们来创建一个新的独立实例。不过我们需要寻找一个存储在堆上的数据来探索 Rust 是如何知道该在何时清理数据的，而 `String` 类型就是一个很好的例子。

我们会专注于 `String` 与所有权相关的部分。这些方面也同样适用于标准库提供的或你自己创建的其他复杂数据类型。在第八章会更深入地讲解 `String`。

我们已经见过字符串字面值，即被硬编码进程序里的字符串值。字符串字面值是很方便的，不过它们并不适合使用文本的每一种场景。原因之一就是它们是不可变的。另一个原因是并非所有字符串的值都能在编写代码时就知道：例如，要是想获取用户输入并存储该怎么办呢？为此，Rust 有另一种字符串类型，`String`。这个类型管理被分配到堆上的数据，所以能够存储在编译时未知大小的文本。可以使用 `from` 函数基于字符串字面值来创建 `String`，如下：

这两个冒号 `::` 是运算符，允许将特定的 `from` 函数置于 `String` 类型的命名空间（namespace）下，而不需要使用类似 `string_from` 这样的名字。在第五章的 “方法语法”（“Method Syntax”） 部分会着重讲解这个语法，而且在第七章的 “路径用于引用模块树中的项” 中会讲到模块的命名空间。

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() 在字符串后追加字面值

println!("{s}"); // 将打印 `hello, world!`
```

那么这里有什么区别呢？为什么 `String` 可变而字面值却不行呢？区别在于两个类型对内存的处理上。

内存与分配

就字符串字面值来说，我们在编译时就知道其内容，所以文本被直接硬编码进最终的可执行文件中。这使得字符串字面值快速且高效。不过这些特性都只得益于字符串字面值的不可变性。不幸的是，我们不能为了每一个在编译时大小未知的文本而将一块内存放入二进制文件中，并且它的大小还可能随着程序运行而改变。

对于 `String` 类型，为了支持一个可变，可增长的文本片段，需要在堆上分配一块在编译时未知大小的内存来存放内容。这意味着：

- 必须在运行时向内存分配器（memory allocator）请求内存。
- 需要一个当我们处理完 `String` 时将内存返回给分配器的方法。

第一部分由我们完成：当调用 `String::from` 时，它的实现 (*implementation*) 请求其所需的内存。这在编程语言中是非常通用的。

然而，第二部分实现起来就各有区别了。在有 **垃圾回收**（*garbage collector*，*GC*）的语言中，GC 记录并清除不再使用的内存，而我们并不需要关心它。在大部分没有 GC 的语言中，识别出不再使用的内存并调用代码显式释放就是我们的责任了，跟请求内存的时候一样。从历史的角度上说正确处理内存回收曾经是一个困难的编程问题。如果忘记回收了会浪费内存。如果过

早回收了，将会出现无效变量。如果重复回收，这也是个 bug。我们需要精确的为一个 `allocate` 配对一个 `free`。

Rust 采取了一个不同的策略：内存在拥有它的变量离开作用域后就被自动释放。下面是示例 4-1 中作用域例子的一个使用 `String` 而不是字符串字面值的版本：

```
{
    let s = String::from("hello"); // 从此处起，s 是有效的

    // 使用 s
}                                  // 此作用域已结束，
                                   // s 不再有效
```

这是一个将 `String` 需要的内存返回给分配器的很自然的位置：当 `s` 离开作用域的时候。当变量离开作用域，Rust 为我们调用一个特殊的函数。这个函数叫做 `drop`，在这里 `String` 的作者可以放置释放内存的代码。Rust 在结尾的 `}` 处自动调用 `drop`。

> 注意：在 C++ 中，这种 item 在生命周期结束时释放资源的模式有时被称作 **资源获取即初始化**（*Resource Acquisition Is Initialization (RAII)*）。如果你使用过 RAII 模式的话应该对 Rust 的 `drop` 函数并不陌生。

这个模式对编写 Rust 代码的方式有着深远的影响。现在它看起来很简单，不过在更复杂的场景下代码的行为可能是不可预测的，比如当有多个变量使用在堆上分配的内存时。现在让我们探索一些这样的场景。

使用移动的变量与数据交互

在 Rust 中，多个变量可以采取不同的方式与同一数据进行交互。让我们看看示例 4-2 中一个使用整型的例子。

```
let x = 5;
let y = x;
```

示例 4-2：将变量 `x` 的整数值赋给 `y`

我们大致可以猜到这在干什么：“将 `5` 绑定到 `x`；接着生成一个值 `x` 的拷贝并绑定到 `y`”。现在有了两个变量，`x` 和 `y`，都等于 `5`。这也正是事实上发生了的，因为整数是有已知固定大小的简单值，所以这两个 `5` 被压入了栈中。

现在看看这个 `String` 版本：

```
let s1 = String::from("hello");
let s2 = s1;
```

这看起来与上面的代码非常类似，所以我们可能会假设它们的运行方式也是类似的：也就是说，第二行可能会生成一个 `s1` 的拷贝并绑定到 `s2` 上。但事实并非如此。

看看图 4-1 以了解 `String` 的底层会发生什么。`String` 由三部分组成，如图左侧所示：一个指向存放字符串内容内存的指针，一个长度，和一个容量。这一组数据存储在栈上。右侧则是堆上存放内容的内存部分。

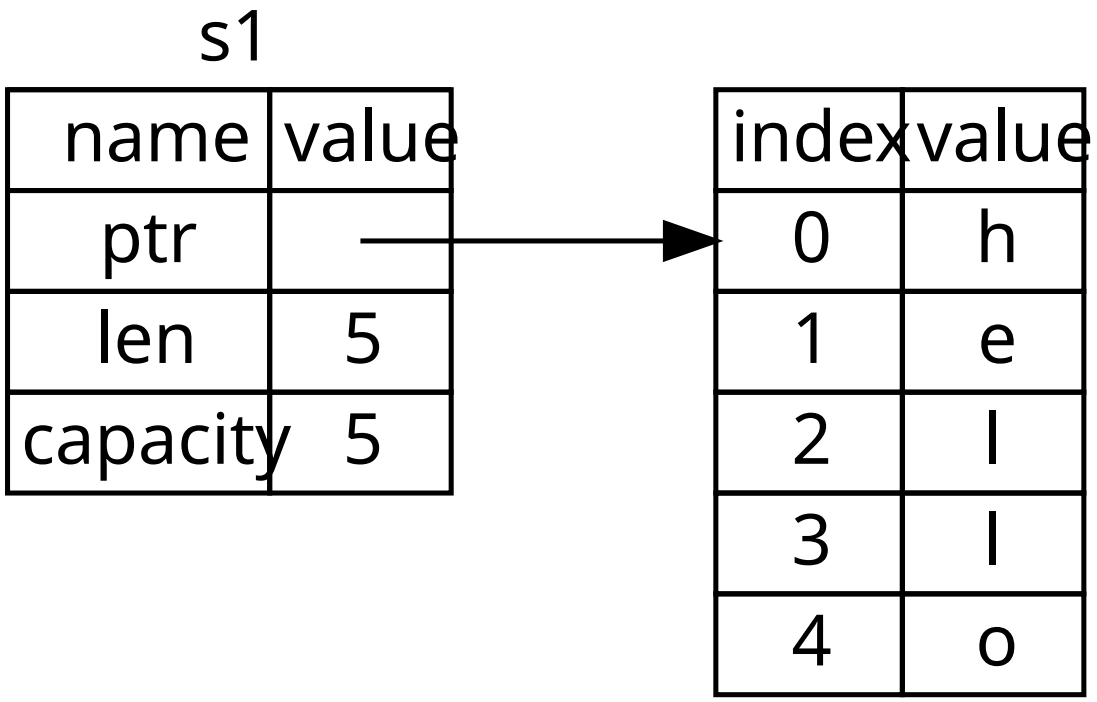


图 4-1：将值 `"hello"` 绑定给 `s1` 的 `String` 在内存中的表现形式

长度表示 `String` 的内容当前使用了多少字节的内存。容量是 `String` 从分配器总共获取了多少字节的内存。长度与容量的区别是很重要的，不过在当前上下文中并不重要，所以现在可以忽略容量。

当我们将 `s1` 赋值给 `s2`，`String` 的数据被复制了，这意味着我们从栈上拷贝了它的指针、长度和容量。我们并没有复制指针指向的堆上数据。换句话说，内存中数据的表现如图 4-2 所示。

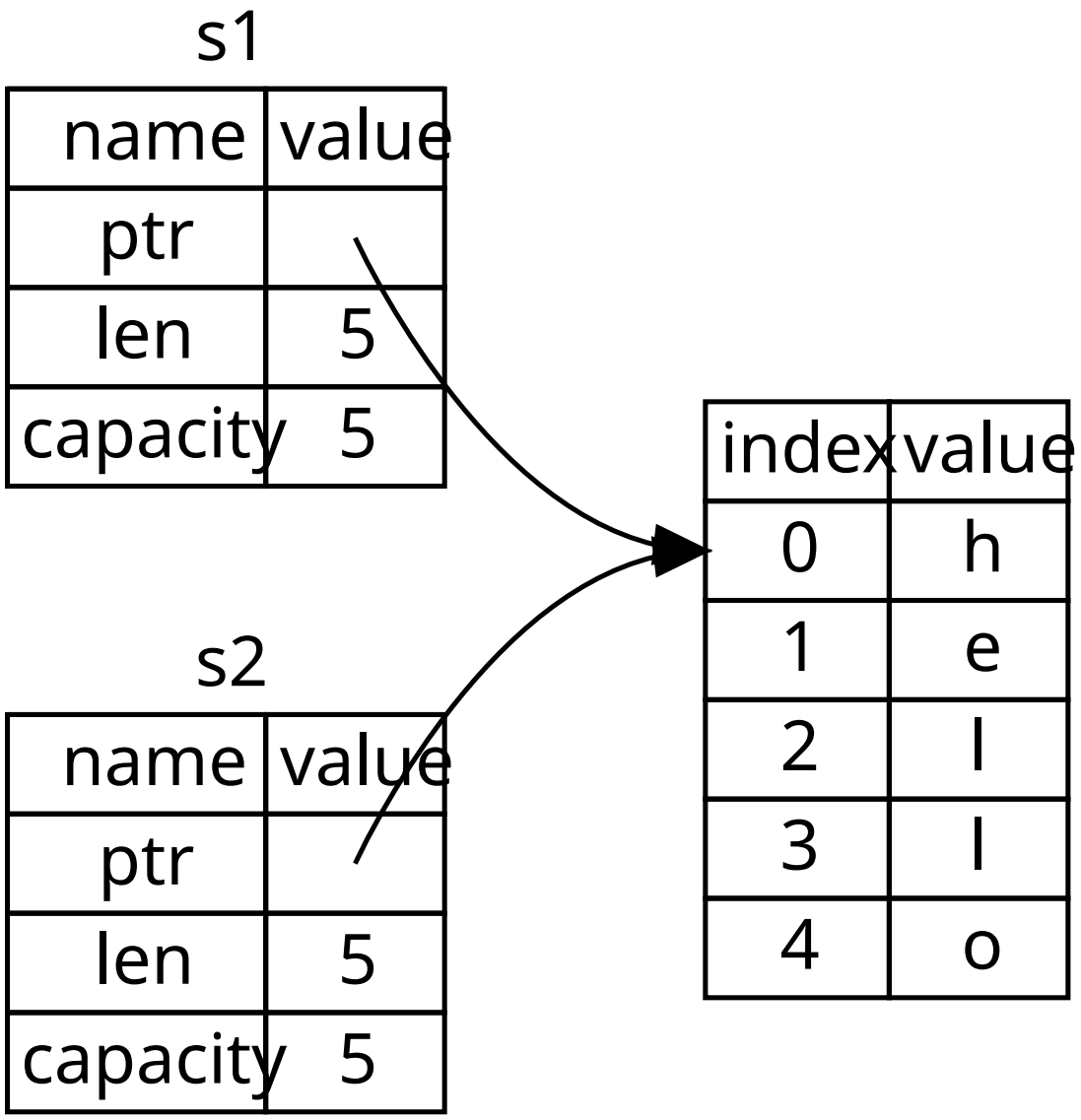


图 4-2：变量 `s2` 的内存表现，它有一份 `s1` 指针、长度和容量的拷贝

这个表现形式看起来**并不像**图 4-3 中的那样，如果 Rust 也拷贝了堆上的数据，那么内存看起来就是这样的。如果 Rust 这么做了，那么操作 `s2 = s1` 在堆上数据比较大的时候会对运行时性能造成非常大的影响。

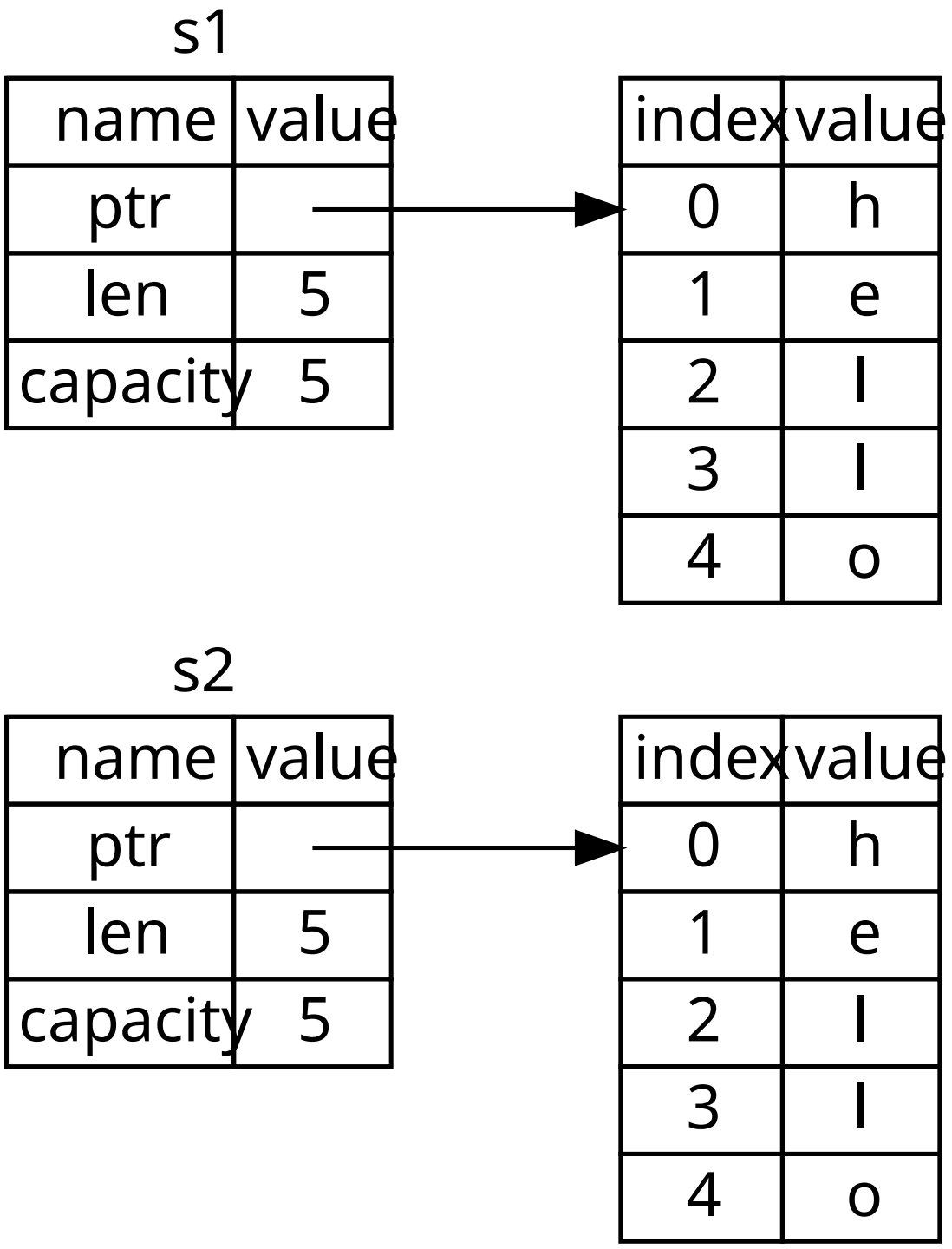


图 4-3：另一个 `s2 = s1` 时可能的内存表现，如果 Rust 同时也拷贝了堆上的数据的话

之前我们提到过当变量离开作用域后，Rust 自动调用 `drop` 函数并清理变量的堆内存。不过图 4-2 展示了两个数据指针指向了同一位置。这就有了一个问题：当 `s2` 和 `s1` 离开作用域，它们都会尝试释放相同的内存。这是一个叫做 **二次释放**（*double free*）的错误，也是之前提到过的内存安全性 bug 之一。两次释放（相同）内存会导致内存污染，它可能会导致潜在的安全漏洞。

为了确保内存安全，在 `let s2 = s1;` 之后，Rust 认为 `s1` 不再有效，因此 Rust 不需要在 `s1` 离开作用域后清理任何东西。看看在 `s2` 被创建之后尝试使用 `s1` 会发生什么；这段代码不能运行：

```
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{s1}, world!");
```



你会得到一个类似如下的错误，因为 Rust 禁止你使用无效的引用。

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:15
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not
implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{s1}, world!");
  |               ^^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes
from the expansion of the macro `println` (in Nightly builds, run with -Z macro-
backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

如果你在其他语言中听说过术语 **浅拷贝**（*shallow copy*）和 **深拷贝**（*deep copy*），那么拷贝指针、长度和容量而不拷贝数据可能听起来像浅拷贝。不过因为 Rust 同时使第一个变量无效了，这个操作被称为 **移动**（*move*），而不是叫做浅拷贝。上面的例子可以解读为 `s1` 被 **移动** 到了 `s2` 中。那么具体发生了什么，如图 4-4 所示。

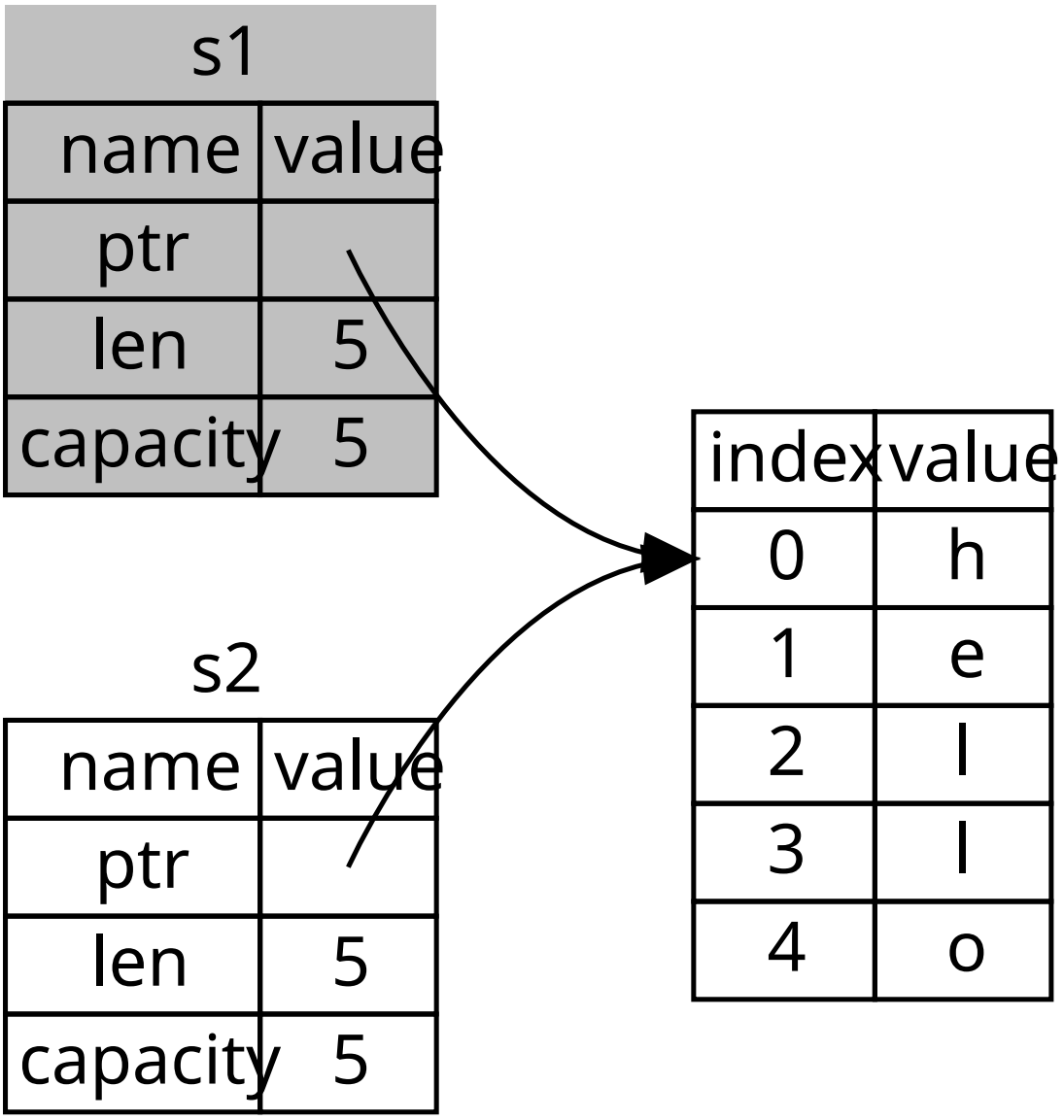


图 4-4：`s1` 无效之后的内存表现

这样就解决了我们的问题！因为只有 `s2` 是有效的，当其离开作用域，它就释放自己的内存，完毕。

另外，这里还隐含了一个设计选择：Rust 永远也不会自动创建数据的 “深拷贝”。因此，任何**自动**的复制都可以被认为是对运行时性能影响较小的。

作用域与赋值

作用域、所有权和通过 `drop` 函数释放内存之间的关系反过来也同样成立。当你给一个已有的变量赋一个全新的值时，Rust 将会立即调用 `drop` 并释放原始值的内存。例如，考虑如下代码：

```
    let mut s = String::from("hello");
    s = String::from("ahoy");
```

```
    println!("{s}, world!");
```

起初我们声明了变量 `s` 并绑定为一个 `"hello"` 值的 `String`。接着立即创建了一个值为 `"ahoy"` 的 `String` 并赋值给 `s`。在这里，完全没有任何内容指向了原始堆上的值。

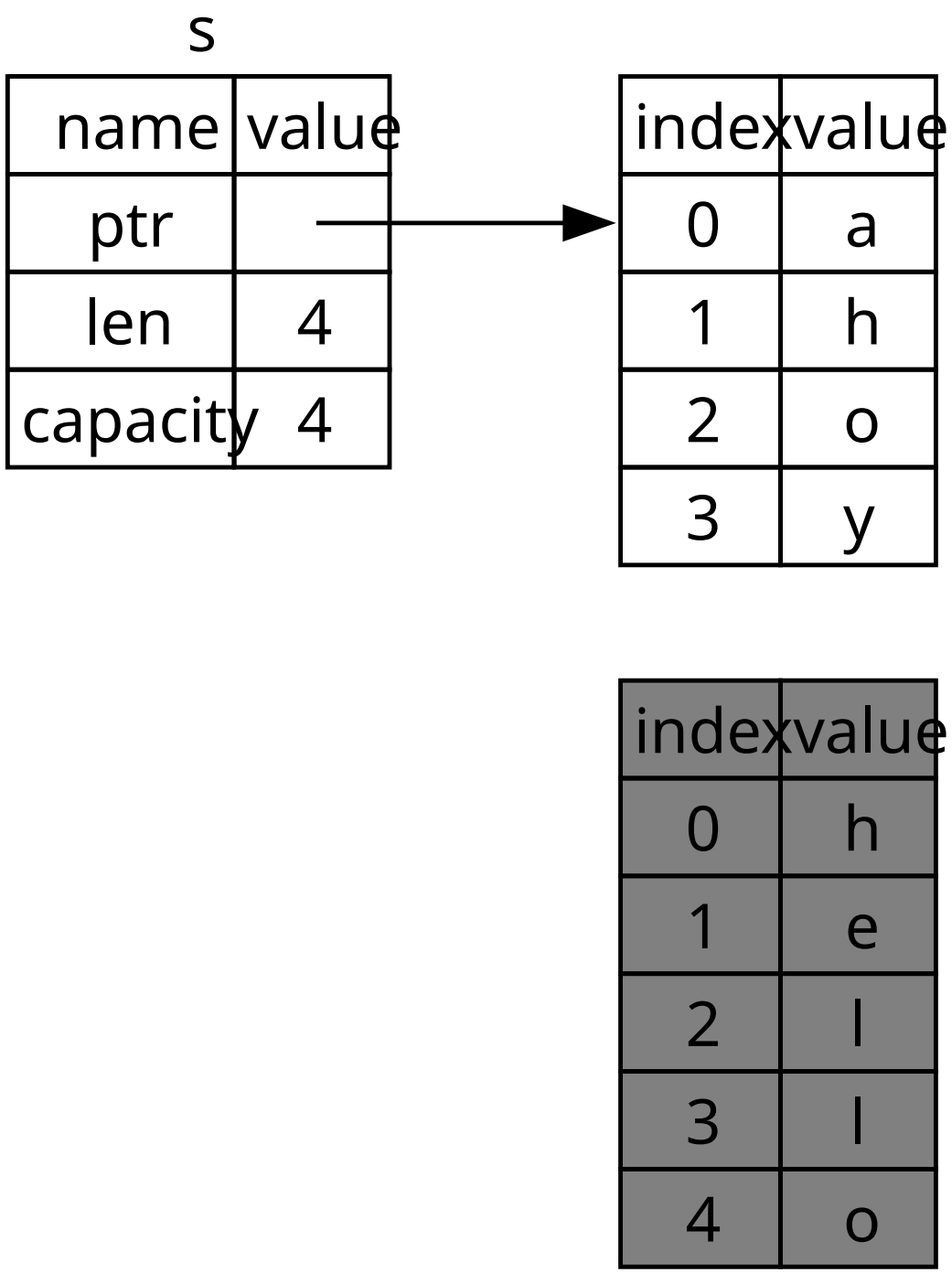


图 4-5: 当初始值被整体替换后的内存表现

因此原始的字符串立刻就离开了作用域。Rust 会在其上运行 `drop` 函数同时内存会马上释放。当结尾打印其值时，将会是 `"ahoy, world!"`。

使用克隆的变量与数据交互

如果我们 **确实** 需要深度复制 `String` 中堆上的数据，而不仅仅是栈上的数据，可以使用一个叫做 `clone` 的常用方法。第五章会讨论方法语法，不过因为方法在很多语言中是一个常见功能，所以之前你可能已经见过了。

这是一个实际使用 `clone` 方法的例子：

```
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("s1 = {s1}, s2 = {s2}");
```

这段代码能正常运行，并且明确产生图 4-3 中行为，这里堆上的数据**确实**被复制了。

当出现 `clone` 调用时，你知道一些特定的代码被执行而且这些代码可能相当消耗资源。你很容易察觉到一些不寻常的事情正在发生。

只在栈上的数据：拷贝

这里还有一个没有提到的细节。这些代码使用了整型并且是有效的，它们是示例 4-2 中的一部分：

```
    let x = 5;
    let y = x;

    println!("x = {x}, y = {y}");
```

但这段代码似乎与我们刚刚学到的内容相矛盾：没有调用 `clone`，不过 `x` 依然有效且没有被移动到 `y` 中。

原因是像整型这样的在编译时已知大小的类型被整个存储在栈上，所以拷贝其实际的值是快速的。这意味着没有理由在创建变量 `y` 后使 `x` 无效。换句话说，这里没有深浅拷贝的区别，所以这里调用 `clone` 并不会与通常的浅拷贝有什么不同，我们可以不用管它。

Rust 有一个叫做 `Copy` trait 的特殊注解，可以用在类似整型这样的存储在栈上的类型上（第十章将会详细讲解 trait）。如果一个类型实现了 `Copy` trait，那么一个旧的变量在将其赋值给其他变量后仍然有效。

Rust 不允许自身或其任何部分实现了 `Drop` trait 的类型使用 `Copy` trait。如果我们对其值离开作用域时需要特殊处理的类型使用 `Copy` 注解，将会出现一个编译时错误。要学习如何为你的类型添加 `Copy` 注解以实现该 trait，请阅读附录 C 中的 “可派生的 trait”。

那么哪些类型实现了 `Copy` trait 呢？你可以查看给定类型的文档来确认，不过作为一个通用的规则，任何一组简单标量值的组合都可以实现 `Copy`，任何不需要分配内存或某种形式资源的类型都可以实现 `Copy` 。如下是一些 `Copy` 的类型：

- 所有整数类型，比如 `u32`。
- 布尔类型，`bool`，它的值是 `true` 和 `false`。
- 所有浮点数类型，比如 `f64`。

- 字符类型，`char`。
- 元组，当且仅当其包含的类型也都实现 `Copy` 的时候。比如，`(i32, i32)` 实现了 `Copy`，但 `(i32, String)` 就没有。

所有权与函数

将值传递给函数与给变量赋值的原理相似。向函数传递值可能会移动或者复制，就像赋值语句一样。示例 4-3 使用注释展示变量何时进入和离开作用域：

文件名：src/main.rs

```
fn main() {
    let s = String::from("hello");  // s 进入作用域

    takes_ownership(s);             // s 的值移动到函数里 ...
                                    // ... 所以到这里不再有效

    let x = 5;                      // x 进入作用域

    makes_copy(x);                  // x 应该移动函数里，
                                    // 但 i32 是 Copy 的，
    println!("{}", x);              // 所以在后面可继续使用 x

} // 这里，x 先移出了作用域，然后是 s。但因为 s 的值已被移走，
  // 没有特殊之处

fn takes_ownership(some_string: String) { // some_string 进入作用域
    println!("{some_string}");
} // 这里，some_string 移出作用域并调用 `drop` 方法。
  // 占用的内存被释放

fn makes_copy(some_integer: i32) { // some_integer 进入作用域
    println!("{some_integer}");
} // 这里，some_integer 移出作用域。没有特殊之处
```

示例 4-3：带有所有权和作用域注释的函数

当尝试在调用 `takes_ownership` 后使用 `s` 时，Rust 会抛出一个编译时错误。这些静态检查使我们免于犯错。试试在 `main` 函数中添加使用 `s` 和 `x` 的代码来看看哪里能使用它们，以及所有权规则会在哪里阻止我们这么做。

返回值与作用域

返回值也可以转移所有权。示例 4-4 展示了一个返回了某些值的示例，与示例 4-3 一样带有类似的注释。

文件名：src/main.rs

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership 将它的返回值传递给 s1

    let s2 = String::from("hello");     // s2 进入作用域

    let s3 = takes_and_gives_back(s2); // s2 被传入 takes_and_gives_back,
```

```
                                        // 它的返回值又传递给 s3
} // 此处，s3 移出作用域并被丢弃。s2 被 move，所以无事发生
  // s1 移出作用域并被丢弃

fn gives_ownership() -> String {        // gives_ownership 将会把返回值传入
                                        // 调用它的函数

    let some_string = String::from("yours"); // some_string 进入作用域

    some_string                         // 返回 some_string 并将其移至调用函数
}

// 该函数将传入字符串并返回该值
fn takes_and_gives_back(a_string: String) -> String {
    // a_string 进入作用域

    a_string  // 返回 a_string 并移出给调用的函数
}
```

示例 4-4: 转移返回值的所有权

变量的所有权总是遵循相同的模式：将值赋给另一个变量时它会移动。当持有堆中数据值的变量离开作用域时，其值将通过 `drop` 被清理掉，除非数据被移动为另一个变量所有。

虽然这样是可以的，但是在每一个函数中都获取所有权并接着返回所有权有些啰嗦。如果我们想要函数使用一个值但不获取所有权该怎么办呢？如果我们还要接着使用它的话，每次都传进去再返回来就有点烦人了，除此之外，我们也可能想返回函数体中产生的一些数据。

我们可以使用元组来返回多个值，如示例 4-5 所示。

文件名：src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() 返回字符串的长度

    (s, length)
}
```

示例 4-5: 返回参数的所有权

但是这未免有些形式主义，而且这种场景应该很常见。幸运的是，Rust 对此提供了一个不用获取所有权就可以使用值的功能，叫做 **引用**（*references*）。

引用与借用

示例 4-5 中的元组代码有这样一个问题：我们必须将 `String` 返回给调用函数，以便在调用 `calculate_length` 后仍能使用 `String`，因为 `String` 被移动到了 `calculate_length` 内。相反我们可以提供一个 `String` 值的引用（reference）。**引用**（*reference*）像一个指针，因为它是一个地址，我们可以由此访问储存于该地址的属于其他变量的数据。与指针不同，引用在其生命周期内保证指向某个特定类型的有效值。

下面是如何定义并使用一个（新的） `calculate_length` 函数，它以一个对象的引用作为参数而不是获取值的所有权：

文件名：src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

首先，注意变量声明和函数返回值中的所有元组代码都消失了。其次，注意我们传递 `&s1` 给 `calculate_length`，同时在函数定义中，我们获取 `&String` 而不是 `String`。这些 & 符号就是**引用**，它们允许你使用值但不获取其所有权。图 4-6 展示了一张示意图。

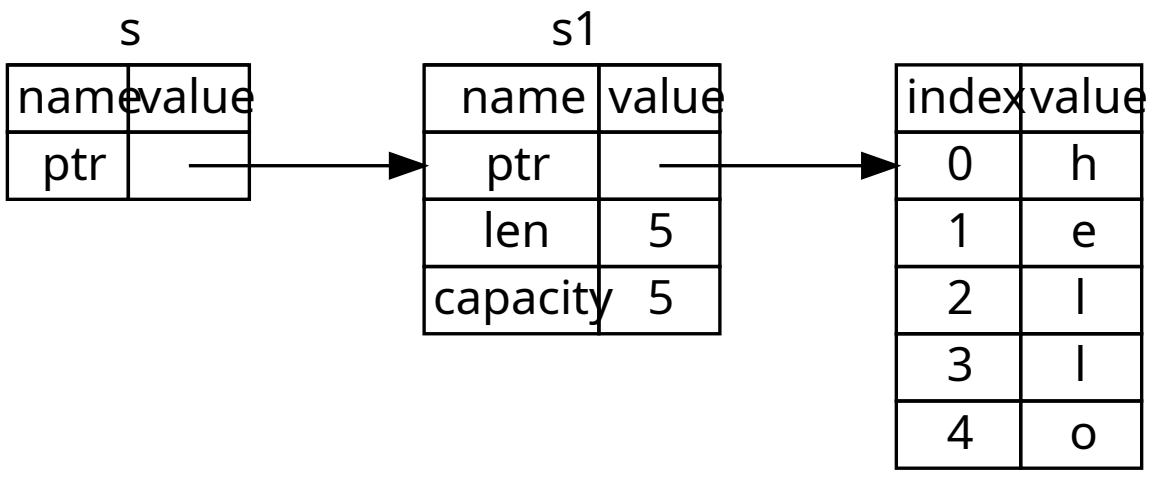


图 4-6：`&String s` 指向 `String s1` 示意图

> 注意：与使用 `&` 引用相反的操作是 **解引用**（*dereferencing*），它使用解引用运算符 `*` 实现。我们将会在第八章遇到一些解引用运算符，并在第十五章详细讨论解引用。

仔细看看这个函数调用：

```
    let s1 = String::from("hello");

    let len = calculate_length(&s1);
```

`&s1` 语法让我们创建一个**指向**值 `s1` 的引用，但是并不拥有它。因为并不拥有这个值，所以当引用停止使用时，它所指向的值也不会被丢弃。

同理，函数签名使用 `&` 来表明参数 `s` 的类型是一个引用。让我们增加一些解释性的注释：

```
fn calculate_length(s: &String) -> usize { // s 是 String 的引用
    s.len()
} // 这里，s 离开了作用域。但因为它并不拥有引用值的所有权，
  // 所以什么也不会发生
```

变量 `s` 有效的作用域与函数参数的作用域一样，不过当 `s` 停止使用时并不丢弃引用指向的数据，因为 `s` 并没有所有权。当函数使用引用而不是实际值作为参数，无需返回值来交还所有权，因为就不曾拥有所有权。

我们将创建一个引用的行为称为 **借用**（*borrowing*）。正如现实生活中，如果一个人拥有某样东西，你可以从他那里借来。当你使用完后，必须还回去。因为我们并不拥有它的所有权。

那如果我们尝试修改借用的变量呢？尝试示例 4-6 中的代码。剧透：这行不通！

文件名：src/main.rs



```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

示例 4-6：尝试修改借用的值

这里是错误：

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&`
reference
 --> src/main.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to
cannot be borrowed as mutable
  |
help: consider changing this to be a mutable reference
  |
7 | fn change(some_string: &mut String) {
  |                         +++
```

```
For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

正如变量默认是不可变的，引用也一样。（默认）不允许修改引用的值。

可变引用

我们通过一个小调整就能修复示例 4-6 代码中的错误，允许我们修改一个借用的值，这就是**可变引用**（*mutable reference*）：

文件名：src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

首先，我们必须将 `s` 改为 `mut`。然后在调用 `change` 函数的地方创建一个可变引用 `&mut s`，并更新函数签名以接受一个可变引用 `some_string: &mut String`。这就非常清楚地表明，`change` 函数将改变它所借用的值。

可变引用有一个很大的限制：如果你有一个对该变量的可变引用，你就不能再创建对该变量的引用。这些尝试创建两个 `s` 的可变引用的代码会失败：

文件名：src/main.rs



```
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
```

错误如下：

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here
```

```
For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

这个报错说这段代码是无效的，因为我们不能在同一时间多次将 `s` 作为可变变量借用。第一个可变的借入在 `r1` 中，并且必须持续到在 `println!` 中使用它，但是在那个可变引用的创建和它的使用之间，我们又尝试在 `r2` 中创建另一个可变引用，该引用借用与 `r1` 相同的数据。

这一限制以一种非常小心谨慎的方式允许可变性，防止同一时间对同一数据存在多个可变引用。新 Rustacean 们经常难以适应这一点，因为大部分语言中变量任何时候都是可变的。这个限制的好处是 Rust 可以在编译时就避免数据竞争。**数据竞争**（*data race*）类似于竞态条件，它可由这三个行为造成：

- 两个或更多指针同时访问同一数据。
- 至少有一个指针被用来写入数据。
- 没有同步数据访问的机制。

数据竞争会导致未定义行为，难以在运行时追踪，并且难以诊断和修复；Rust 通过拒绝编译存在数据竞争的代码来避免此问题！

一如既往，可以使用大括号来创建一个新的作用域，以允许拥有多个可变引用，只是不能**同时**拥有：

```
    let mut s = String::from("hello");

    {
        let r1 = &mut s;
    } // r1 在这里离开了作用域，所以我们完全可以创建一个新的引用

    let r2 = &mut s;
```

Rust 在同时使用可变与不可变引用时也强制采用类似的规则。这些代码会导致一个错误：



```
    let mut s = String::from("hello");

    let r1 = &s; // 没问题
    let r2 = &s; // 没问题
    let r3 = &mut s; // 大问题

    println!("{}, {}, and {}", r1, r2, r3);
```

错误如下：

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
```

```
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                 -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

呼！我们**也**不能在拥有不可变引用的同时拥有可变引用。

不可变引用的借用者可不希望在借用时值会突然发生改变！然而，多个不可变引用是可以的，因为没有哪个只能读取数据的引用者能够影响其他引用者读取到的数据。

注意一个引用的作用域从声明的地方开始一直持续到最后一次使用为止。例如，因为最后一次使用不可变引用的位置在 `println!`，它发生在声明可变引用之前，所以如下代码是可以编译的：

```
    let mut s = String::from("hello");

    let r1 = &s; // 没问题
    let r2 = &s; // 没问题
    println!("{r1} and {r2}");
    // 此位置之后 r1 和 r2 不再使用

    let r3 = &mut s; // 没问题
    println!("{r3}");
```

不可变引用 `r1` 和 `r2` 的作用域在 `println!` 最后一次使用之后结束，这发生在可变引用 `r3` 被创建之前。因为它们的作用域没有重叠，所以代码是可以编译的。编译器可以在作用域结束之前判断不再使用的引用。

尽管借用错误有时令人沮丧，但请牢记这是 Rust 编译器在提前指出一个潜在的 bug（在编译时而不是在运行时）并精准显示问题所在。这样你就不必去跟踪为何数据并不是你想象中的那样。

悬垂引用（Dangling References）

在具有指针的语言中，很容易通过释放内存时保留指向它的指针而错误地生成一个**悬垂指针**（*dangling pointer*）—— 指向可能已被分配给其他用途的内存位置的指针。相比之下，在 Rust 中编译器确保引用永远也不会变成悬垂引用：当你拥有一些数据的引用，编译器确保数据不会在其引用之前离开作用域。

让我们尝试创建一个悬垂引用，看看 Rust 如何通过通过一个编译时错误来防止它：

文件名：src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
```

```
    &s
}
```



这里是错误：

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
value for it to be borrowed from
help: consider using the `'static` lifetime, but this is uncommon unless you're
returning a borrowed value from a `const` or a `static`
  |
5 | fn dangle() -> &'static String {
  |                 +++++++
help: instead, you are more likely to want to return an owned value
  |
5 - fn dangle() -> &String {
5 + fn dangle() -> String {
  |

error[E0515]: cannot return reference to local variable `s`
 --> src/main.rs:8:5
  |
8 |     &s
  |     ^^ returns a reference to data owned by the current function

Some errors have detailed explanations: E0106, E0515.
For more information about an error, try `rustc --explain E0106`.
error: could not compile `ownership` (bin "ownership") due to 2 previous errors
```

错误信息引用了一个我们还未介绍的功能：生命周期（lifetimes）。第十章会详细介绍生命周期。不过，如果你不理会生命周期部分，错误信息中确实包含了为什么这段代码有问题的关键信息：

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from
```

让我们仔细看看我们的 `dangle` 代码的每个阶段到底发生了什么：

文件名：src/main.rs

```
fn dangle() -> &String { // dangle 返回一个字符串的引用

    let s = String::from("hello"); // s 是一个新字符串

    &s // 返回字符串 s 的引用
```

```
} // 这里 s 离开作用域并被丢弃。其内存被释放。
  // 危险!
```



因为 `s` 是在 `dangle` 函数内创建的，当 `dangle` 的代码执行完毕后，`s` 将被释放。不过我们尝试返回它的引用。这意味着这个引用会指向一个无效的 `String`，这可不对！Rust 不会允许我们这么做。

这里的解决方法是直接返回 `String`：

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

这样就没有任何错误了。所有权被移动出去，所以没有值被释放。

引用的规则

让我们概括一下之前对引用的讨论：

- 在任意给定时间，**要么**只能有一个可变引用，**要么**只能有多个不可变引用。
- 引用必须总是有效的。

接下来，我们来看看另一种不同类型的引用：slice。

Slice 类型

切片（*slice*）允许你引用集合中一段连续的元素序列，而不用引用整个集合。slice 是一种引用，所以它不拥有所有权。

这里有一个编程小习题：编写一个函数，该函数接收一个用空格分隔单词的字符串，并返回在该字符串中找到的第一个单词。如果函数在该字符串中并未找到空格，则整个字符串就是一个单词，所以应该返回整个字符串。

> 注意：出于介绍字符串 slice 的目的，本小节假设只使用 ASCII 字符集；一个关于 UTF-8 处理的更全面的讨论位于第八章“使用字符串储存 UTF-8 编码的文本”小节。

让我们推敲下如何不用 slice 编写这个函数的签名，来理解 slice 能解决的问题：

```
fn first_word(s: &String) -> ?
```

`first_word` 函数有一个参数 `&String`。因为我们不需要所有权，所以这没有问题。不过应该返回什么呢？我们并没有一个真正获取**部分**字符串的办法。不过，我们可以返回单词结尾的索引，结尾由一个空格表示。试试如示例 4-7 中的代码。

文件名：src/main.rs

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

示例 4-7：`first_word` 函数返回 `String` 参数的一个字节索引值

因为需要逐个元素的检查 `String` 中的值是否为空格，需要用 `as_bytes` 方法将 `String` 转化为字节数组。

```
    let bytes = s.as_bytes();
```

接下来，使用 `iter` 方法在字节数组上创建一个迭代器：

```
    for (i, &item) in bytes.iter().enumerate() {
```

我们将在第十三章详细讨论迭代器。现在，只需知道 `iter` 方法返回集合中的每一个元素，而 `enumerate` 包装了 `iter` 的结果，将这些元素作为元组的一部分来返回。`enumerate` 返回的元组中，第一个元素是索引，第二个元素是集合中元素的引用。这比我们自己计算索引要方便一些。

因为 `enumerate` 方法返回一个元组，我们可以使用模式来解构，我们将在第六章中进一步讨论有关模式的问题。所以在 `for` 循环中，我们指定了一个模式，其中元组中的 `i` 是索引而元组中的 `&item` 是单个字节。因为我们从 `.iter().enumerate()` 中获取了集合元素的引用，所以模式中使用了 `&`。

在 `for` 循环中，我们通过字节的字面值语法来寻找代表空格的字节。如果找到了一个空格，返回它的位置。否则，使用 `s.len()` 返回字符串的长度。

```
        if item == b' ' {
            return i;
        }
    }

    s.len()
```

现在有了一个找到字符串中第一个单词结尾索引的方法，不过这有一个问题。我们返回了一个独立的 `usize`，不过它只在 `&String` 的上下文中才是一个有意义的数字。换句话说，因为它是一个与 `String` 相分离的值，无法保证将来它仍然有效。考虑一下示例 4-8 中使用了示例 4-7 中 `first_word` 函数的程序。

文件名：src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word 的值为 5

    s.clear(); // 这清空了字符串，使其等于 ""

    // word 在此处的值仍然是 5，
    // 但是没有更多的字符串让我们可以有效地应用数值 5。word 的值现在完全无效！
}
```

示例 4-8：存储 `first_word` 函数调用的返回值并接着改变 `String` 的内容

这个程序编译时没有任何错误，而且在调用 `s.clear()` 之后使用 `word` 也不会出错。因为 `word` 与 `s` 状态完全没有联系，所以 `word` 仍然包含值 `5`。可以尝试用值 `5` 来提取变量 `s` 的第一个单词，不过这是有 bug 的，因为在我们将 `5` 保存到 `word` 之后 `s` 的内容已经改变。

我们不得不时刻担心 `word` 的索引与 `s` 中的数据不再同步，这既繁琐又易出错！如果编写这么一个 `second_word` 函数的话，管理索引这件事将更加容易出问题。它的签名看起来像这样：

```
fn second_word(s: &String) -> (usize, usize) {
```

现在我们要跟踪一个开始索引**和**一个结束索引，同时有了更多从数据的某个特定状态计算而来的值，但都完全没有与这个状态相关联。现在有三个飘忽不定的不相关变量需要保持同步。

幸运的是，Rust 为这个问题提供了一个解决方法：字符串 slice。

字符串 slice

字符串 slice（*string slice*）是 `String` 中一部分值的引用，它看起来像这样：

```
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```

不同于整个 `String` 的引用，`hello` 是一个部分 `String` 的引用，由一个额外的 `[0..5]` 部分指定。可以使用一个由中括号中的 `[starting_index..ending_index]` 指定的 range 创建一个 slice，其中 `starting_index` 是 slice 的第一个位置，`ending_index` 则是 slice 最后一个位置的后一个值。在其内部，slice 的数据结构存储了 slice 的开始位置和长度，长度对应于 `ending_index` 减去 `starting_index` 的值。所以对于 `let world = &s[6..11];` 的情况，`world` 将是一个包含指向 `s` 索引 6 的指针和长度值 5 的 slice。

图 4-7 展示了一个图例。

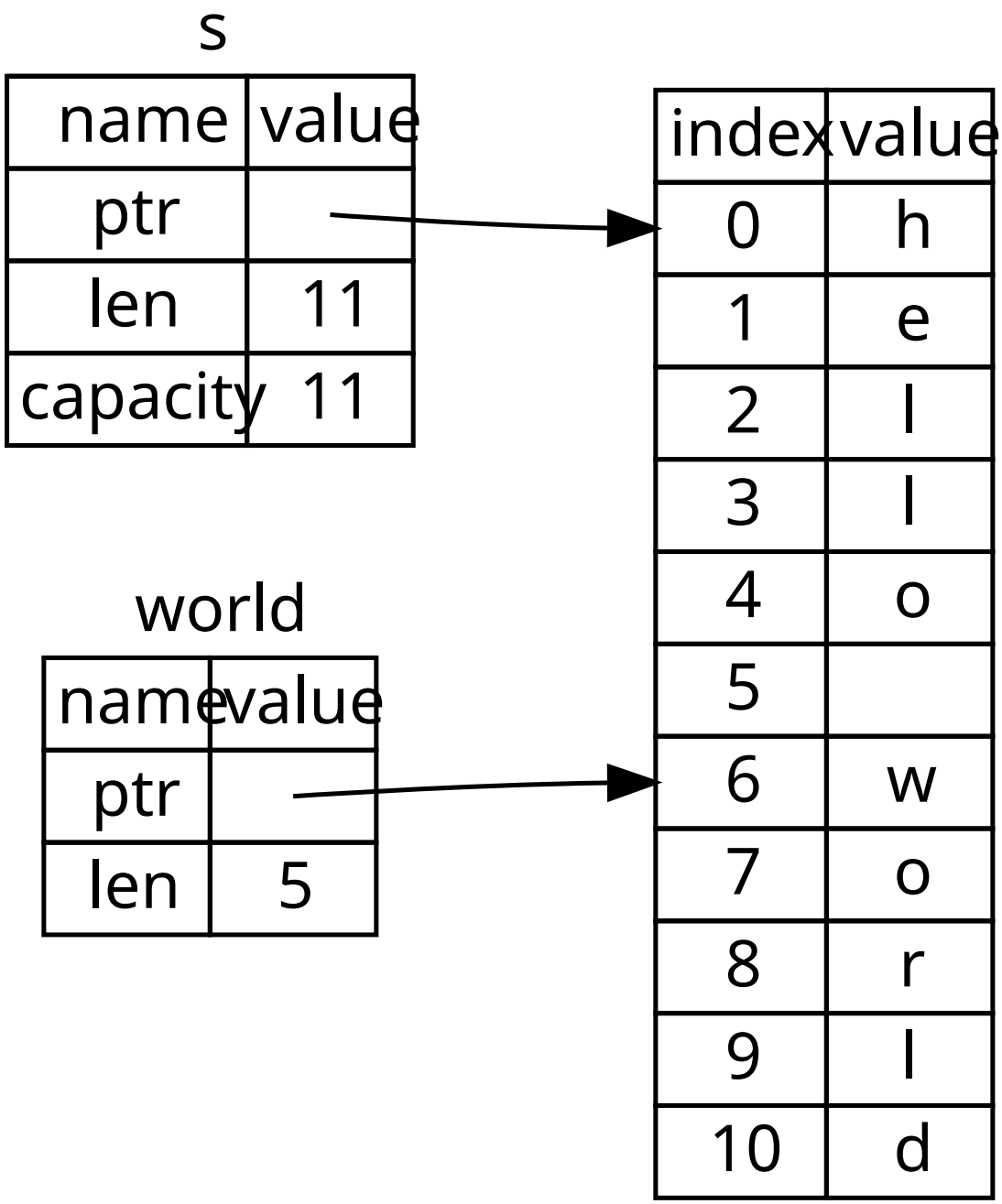


图 4-7：引用了部分 `String` 的字符串 slice

对于 Rust 的 `..` range 语法，如果想要从索引 0 开始，可以不写两个点号之前的值。换句话说，如下两个语句是相同的：

```
let s = String::from("hello");
```

```
let slice = &s[0..2];
let slice = &s[..2];
```

依此类推，如果 slice 包含 `String` 的最后一个字节，也可以舍弃尾部的数字。这意味着如下也是相同的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

也可以同时舍弃这两个值来获取整个字符串的 slice。所以如下亦是相同的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

> 注意：字符串 slice range 的索引必须位于有效的 UTF-8 字符边界内，如果尝试从一个多字节字符的中间位置创建字符串 slice，则程序将会因错误而退出。

在记住所有这些知识后，让我们重写 `first_word` 来返回一个 slice。“字符串 slice” 的类型声明写作 `&str`：

文件名：src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

我们使用跟示例 4-7 相同的方式获取单词结尾的索引，通过寻找第一个出现的空格。当找到一个空格，我们返回一个字符串 slice，它使用字符串的开始和空格的索引作为开始和结束的索引。

现在当调用 `first_word` 时，会返回与底层数据关联的单个值。这个值由一个 slice 开始位置的引用和 slice 中元素的数量组成。

`second_word` 函数也可以改为返回一个 slice：

```
fn second_word(s: &String) -> &str {
```

现在我们有了一个不易混淆且直观的 API 了，因为编译器会确保指向 `String` 的引用持续有效。还记得示例 4-8 程序中，那个当我们获取第一个单词结尾的索引后，接着就清除了字符串导致索引就无效的 bug 吗？那些代码在逻辑上是不正确的，但却没有显示任何直接的错误。问题会在之后尝试对空字符串使用第一个单词的索引时出现。slice 就不可能出现这种 bug 并让我们更早的知道出问题了。使用 slice 版本的 `first_word` 会抛出一个编译时错误：

文件名：src/main.rs



```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // 错误!

    println!("the first word is: {word}");
}
```

这里是编译错误：

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {word}");
   |                                   ------ immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

回忆一下借用规则，当拥有某值的不可变引用时，就不能再获取一个可变引用。因为 `clear` 需要清空 `String`，它尝试获取一个可变引用。在调用 `clear` 之后的 `println!` 使用了 `word` 中的引用，所以这个不可变的引用在此时必须仍然有效。Rust 不允许 `clear` 中的可变引用和 `word` 中的不可变引用同时存在，因此编译失败。Rust 不仅使得我们的 API 简单易用，也在编译时就消除了一整类的错误！

字符串字面值就是 slice

还记得我们讲到过字符串字面值被储存在二进制文件中吗？现在知道 slice 了，我们就可以正确地理解字符串字面值了：

```
```

```
let s = "Hello, world!";
```

这里 `s` 的类型是 `&str`：它是一个指向二进制程序特定位置的 slice。这也就是为什么字符串字面值是不可变的；`&str` 是一个不可变引用。

字符串 slice 作为参数

在知道了能够获取字面值和 `String` 的 slice 后，我们对 `first_word` 做了改进，这是它的签名：

```
fn first_word(s: &String) -> &str {
```

而更有经验的 Rustacean 会编写出示例 4-9 中的签名，因为它使得可以对 `&String` 值和 `&str` 值使用相同的函数：

```
fn first_word(s: &str) -> &str {
```

示例 4-9: 通过将 `s` 参数的类型改为字符串 slice 来改进 `first_word` 函数

如果有一个字符串 slice，可以直接传递它。如果有一个 `String`，则可以传递整个 `String` 的 slice 或对 `String` 的引用。这种灵活性利用了 *deref coercions* 的优势，这个特性我们将在"函数和方法的隐式 Deref 强制转换"章节中介绍。定义一个获取字符串 slice 而不是 `String` 引用的函数使得我们的 API 更加通用并且不会丢失任何功能：

文件名：src/main.rs

```
fn main() {
    let my_string = String::from("hello world");

    // `first_word` 适用于 `String`（的 slice），部分或全部
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // `first_word` 也适用于 `String` 的引用，
    // 这等价于整个 `String` 的 slice
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // `first_word` 适用于字符串字面值，部分或全部
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // 因为字符串字面值已经 **是** 字符串 slice 了，
    // 这也是适用的，无需 slice 语法！
    let word = first_word(my_string_literal);
}
```

其他类型的 slice

字符串 slice，正如你想象的那样，是针对字符串的。不过也有更通用的 slice 类型。考虑一下这个数组：

```
let a = [1, 2, 3, 4, 5];
```

就跟我们想要获取字符串的一部分那样，我们也会想要引用数组的一部分。我们可以这样做：

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

这个 slice 的类型是 `&[i32]`。它跟字符串 slice 的工作方式一样，通过存储第一个集合元素的引用和一个集合总长度。你可以对其他所有集合使用这类 slice。第八章讲到 vector 时会详细讨论这些集合。

总结

所有权、借用和 slice 这些概念让 Rust 程序在编译时确保内存安全。Rust 语言提供了跟其他系统编程语言相同的方式来控制你使用的内存，但拥有数据所有者在离开作用域后自动清除其数据的功能意味着你无须额外编写和调试相关的控制代码。

所有权系统影响了 Rust 中很多其他部分的工作方式，所以我们还会继续讲到这些概念，这将贯穿本书的余下内容。让我们开始第五章，来看看如何将多份数据组合进一个 `struct` 中。

使用结构体组织相关联的数据

结构体（*struct*），或者 *structure*，是一个自定义数据类型，允许你包装和命名多个相关的值，从而形成一个有意义的组合。如果你熟悉一门面向对象语言，*struct* 就像对象中的数据属性。在本章中，我们会对元组和结构体进行比较和对比，来全建立已有的知识，并演示在何种情况下结构体是组合数据的更好方式。

我们还将演示如何定义和实例化结构体，并讨论如何定义关联函数，特别是被称为 方法 的那种关联函数，以指定与结构体类型相关的行为。你可以在程序中基于结构体和枚举（*enum*）（在第六章介绍）创建新类型，以充分利用 Rust 的编译时类型检查。

结构体的定义和实例化

结构体和我们在“元组类型”部分论过的元组类似，它们都包含多个相关的值。和元组一样，结构体的每一部分可以是不同类型。但不同于元组，结构体需要命名各部分数据以便能清楚的表明其值的意义。由于有了这些名字，结构体比元组更灵活：不需要依赖顺序来指定或访问实例中的值。

定义结构体，需要使用 `struct` 关键字并为整个结构体提供一个名字。结构体的名字需要描述它所组合的数据的意义。接着，在大括号中，定义每一部分数据的名字和类型，我们称为 **字段**（*field*）。例如，示例 5-1 展示了一个存储用户账号信息的结构体：

文件名：src/main.rs

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

示例 5-1：`User` 结构体定义

一旦定义了结构体后，为了使用它，通过为每个字段指定具体值来创建这个结构体的**实例**。创建一个实例需要以结构体的名字开头，接着在大括号中使用 `key: value` 键 - 值对的形式提供字段，其中 key 是字段的名字，value 是需要存储在字段中的数据值。实例中字段的顺序不需要和它们在结构体中声明的顺序一致。换句话说，结构体的定义就像一个类型的通用模板，而实例则会在这个模板中放入特定数据来创建这个类型的值。例如，可以像示例 5-2 这样来声明一个特定的用户：

文件名：src/main.rs

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

示例 5-2：创建 `User` 结构体的实例

为了从结构体中获取某个特定的值，可以使用点号。举个例子，想要用户的邮箱地址，可以用 `user1.email`。如果结构体的实例是可变的，我们可以使用点号并为对应的字段赋值。示例 5-3 展示了如何改变一个可变的 `User` 实例中 `email` 字段的值：

文件名：src/main.rs

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
```

```
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

示例 5-3：改变 `User` 实例 `email` 字段的值

注意整个实例必须是可变的；Rust 并不允许只将某个字段标记为可变。另外需要注意同其他任何表达式一样，我们可以在函数体的最后一个表达式中构造一个结构体的新实例，来隐式地返回这个实例。

示例 5-4 显示了一个 `build_user` 函数，它返回一个带有给定的 email 和用户名的 `User` 结构体实例。`active` 字段的值为 `true`，并且 `sign_in_count` 的值为 `1`。

文件名：src/main.rs

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

示例 5-4：`build_user` 函数获取 email 和用户名并返回 `User` 实例

为函数参数起与结构体字段相同的名字是可以理解的，但是不得不重复 `email` 和 `username` 字段名称与变量有些啰嗦。如果结构体有更多字段，重复每个名称就更加烦人了。幸运的是，有一个方便的简写语法！

使用字段初始化简写语法

因为示例 5-4 中的参数名与字段名都完全相同，我们可以使用 **字段初始化简写语法**（*field init shorthand*）来重写 `build_user`，这样其行为与之前完全相同，不过无需重复 `username` 和 `email` 了，如示例 5-5 所示。

文件名：src/main.rs

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

示例 5-5：`build_user` 函数使用了字段初始化简写语法，因为 `username` 和 `email` 参数与结构体字段同名

这里我们创建了一个新的 `User` 结构体实例，它有一个叫做 `email` 的字段。我们想要将 `email` 字段的值设置为 `build_user` 函数 `email` 参数的值。因为 `email` 字段与 `email` 参数有着相同的名称，则只需编写 `email` 而不是 `email: email`。

使用结构体更新语法从其他实例创建实例

使用旧实例的大部分值但改变其部分值来创建一个新的结构体实例通常是很有用的。这可以通过 **结构体更新语法**（*struct update syntax*）实现。

首先，示例 5-6 展示了不使用更新语法时，如何在 `user2` 中创建一个新 `User` 实例。我们为 `email` 设置了新的值，其他值则使用了实例 5-2 中创建的 `user1` 中的同名值：

文件名：src/main.rs

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

示例 5-6：使用 `user1` 中的一个值创建一个新的 `User` 实例

使用结构体更新语法，我们可以通过更少的代码来达到相同的效果，如示例 5-7 所示。`..` 语法指定了剩余未显式设置值的字段应有与给定实例对应字段相同的值。

文件名：src/main.rs

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

示例 5-7：使用结构体更新语法为一个 `User` 实例设置一个新的 `email` 值，不过其余值来自 `user1` 变量中实例的字段

示例 5-7 中的代码也在 `user2` 中创建了一个新实例，但该实例中 `email` 字段的值与 `user1` 不同，而 `username`、 `active` 和 `sign_in_count` 字段的值与 `user1` 相同。`..user1` 必须放在最后，以指定其余的字段应从 `user1` 的相应字段中获取其值，但我们可以选择以任何顺序为任意字段指定值，而不用考虑结构体定义中字段的顺序。

请注意，结构更新语法就像带有 `=` 的赋值，因为它移动了数据，就像我们在“使用移动的变量与数据交互”部分讲到的一样。在这个例子中，总体上说我们在创建 `user2` 后就不能再使用 `user1` 了，因为 `user1` 的 `username` 字段中的 `String` 被移到 `user2` 中。如果我们给 `user2` 的 `email` 和 `username` 都赋予新的 `String` 值，从而只复用 `user1` 的 `active` 和 `sign_in_count` 值，那么 `user1` 在创建 `user2` 后仍然有效。`active` 和 `sign_in_count` 的类型是实现 `Copy` trait 的

类型，所以我们在“使用克隆的变量与数据交互”部分讨论的行为同样适用。在本例中我们也可以继续使用 `user1.email`，因为它的值并未从 `user1` 中移动出去。

使用没有命名字段的元组结构体来创建不同的类型

也可以定义与元组类似的结构体，称为 **元组结构体**（*tuple structs*）。元组结构体有着结构体名称提供的含义，但没有具体的字段名，只有字段的类型。当你想给整个元组取一个名字，并使元组成为与其他元组不同的类型时，元组结构体是很有用的，这时像常规结构体那样为每个字段命名就显得多余和形式化了。

要定义元组结构体，以 `struct` 关键字和结构体名开头并后跟元组中的类型。例如，下面是两个分别叫做 `Color` 和 `Point` 元组结构体的定义和用法：

文件名：src/main.rs

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

注意 `black` 和 `origin` 值的类型不同，因为它们是不同的元组结构体的实例。你定义的每一个结构体有其自己的类型，即使结构体中的字段可能有着相同的类型。例如，一个获取 `Color` 类型参数的函数不能接受 `Point` 作为参数，即便这两个类型都由三个 `i32` 值组成。除此之外，元组结构体实例类似于元组，你可以将它们解构为单独的部分，也可以使用 `.` 后跟索引来访问单独的值。与元组不同的是，解构元组结构体时必须写明结构体的类型。例如，我们可以写 `let Point(x, y, z) = origin;`，将 `origin` 的值解构到名为 `x`、`y` 和 `z` 的变量中。

没有任何字段的类单元结构体

我们也可以定义一个没有任何字段的结构体！它们被称为 **类单元结构体**（*unit-like structs*）因为它们类似于 `()`，即“元组类型”一节中提到的 unit 类型。类单元结构体常常在你想要在某个类型上实现 trait 但不需要在类型中存储数据的时候发挥作用。我们将在第十章介绍 trait。下面是一个声明和实例化一个名为 `AlwaysEqual` 的 unit 结构的示例：

文件名：src/main.rs

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

为了定义 `AlwaysEqual`，我们使用 `struct` 关键字，接着是我们想要的名称，然后是一个分号。不需要花括号或圆括号！然后，我们可以以类似的方式在 `subject` 变量中创建 `AlwaysEqual` 的实例：只需使用我们定义的名称，无需任何花括号或圆括号。设想我们稍后将为这个类型实现某种行为，使得每个 `AlwaysEqual` 的实例始终等于任何其它类型的实例，也许是为了获得一个已知的结果以便进行测试。我们不需要任何数据来实现这种行为！在第十章中，你会看到如何定义 trait 并在任何类型上实现它们，包括类单元结构体。

结构体数据的所有权

在示例 5-1 中的 `User` 结构体的定义中，我们使用了自身拥有所有权的 `String` 类型而不是 `&str` 字符串 slice 类型。这是一个有意而为之的选择，因为我们想要这个结构体拥有它所有的数据，为此只要整个结构体是有效的话其数据也是有效的。

可以使结构体存储被其他对象拥有的数据的引用，不过这么做的话需要用上 **生命周期**（*lifetimes*），这是一个第十章会讨论的 Rust 特性。生命周期确保结构体引用的数据有效性跟结构体本身保持一致。如果你尝试在结构体中存储一个引用而不指定生命周期将是无效的，比如这样：

文件名：src/main.rs



```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

编译器会抱怨它需要生命周期标识符：

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
```

```
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` (bin "structs") due to 2 previous
errors
```

第十章会讲到如何修复这个问题以便在结构体中存储引用，不过现在，我们会使用像 `String` 这类拥有所有权的类型来替代 `&str` 这样的引用以修正这个错误。

结构体示例程序

为了理解何时会需要使用结构体，让我们编写一个计算长方形面积的程序。我们会从单独的变量开始，接着重构程序直到使用结构体替代它们为止。

使用 Cargo 新建一个叫做 *rectangles* 的二进制程序，它获取以像素为单位的长方形的宽度和高度，并计算出长方形的面积。示例 5-8 显示了位于项目的 *src/main.rs* 中的小程序，它刚刚好实现此功能：

文件名：src/main.rs

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

示例 5-8：通过分别指定长方形的宽和高的变量来计算长方形面积

现在使用 `cargo run` 运行程序：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
```

这个示例代码在调用 `area` 函数时传入每个维度，虽然可以正确计算出长方形的面积，但我们仍然可以修改这段代码来使它的意义更加明确，并且增加可读性。

这些代码的问题突显在 `area` 的签名上：

```
fn area(width: u32, height: u32) -> u32 {
```

函数 `area` 本应该计算一个长方形的面积，不过函数却有两个参数。这两个参数是相关联的，不过程序本身却没有表现出这一点。将长度和宽度组合在一起将更易懂也更易处理。第三章的“元组类型”部分已经讨论过了一种可行的方法：元组。

使用元组重构

示例 5-9 展示了使用元组的另一个程序版本。

文件名：src/main.rs

```

```

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

示例 5-9：使用元组来指定长方形的宽高

在某种程度上说，这个程序更好一点了。元组帮助我们增加了一些结构性，并且现在只需传一个参数。不过在另一方面，这个版本却有一点不明确了：元组并没有给出元素的名称，所以计算变得更费解了，因为不得不使用索引来获取元组的每一部分。

在计算面积时将宽和高弄混倒无关紧要，不过当在屏幕上绘制长方形时就有问题了！我们必须牢记 `width` 的元组索引是 `0`，`height` 的元组索引是 `1`。如果其他人要使用这些代码，他们必须要搞清楚这一点，并也要牢记于心。很容易忘记或者混淆这些值而造成错误，因为我们没有在代码中传达数据的意图。

使用结构体重构：赋予更多意义

我们使用结构体为数据命名来为其赋予意义。我们可以将我们正在使用的元组转换成一个有整体名称而且每个部分也有对应名字的结构体，如示例 5-10 所示：

文件名：src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

示例 5-10：定义 `Rectangle` 结构体

这里我们定义了一个结构体并称其为 `Rectangle`。在大括号中定义了字段 `width` 和 `height`，类型都是 `u32`。接着在 `main` 中，我们创建了一个具体的 `Rectangle` 实例，它的宽是 `30`，高是 `50`。

函数 `area` 现在被定义为接收一个名叫 `rectangle` 的参数，其类型是一个结构体 `Rectangle` 实例的不可变借用。第四章讲到过，我们希望借用结构体而不是获取它的所有权，这样 `main` 函数就可以保持 `rect1` 的所有权并继续使用它，所以这就是为什么在函数签名和调用的地方会有 `&`。

`area` 函数访问 `Rectangle` 实例的 `width` 和 `height` 字段（注意，访问对结构体的引用的字段不会移动字段的所有权，这就是为什么你经常看到对结构体的引用）。`area` 的函数签名现在明确的阐述了我们的意图：使用 `Rectangle` 的 `width` 和 `height` 字段，计算 `Rectangle` 的面积。这表明宽高是相互联系的，并为这些值提供了描述性的名称而不是使用元组的索引值 `0` 和 `1`。这在可读性上是一个明显的提升。

通过派生 trait 增加实用功能

在调试程序时打印出 `Rectangle` 实例来查看其所有字段的值非常有用。示例 5-11 像前面章节那样尝试使用 `println!` 宏。但这并不行。

文件名：src/main.rs



```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

示例 5-11：尝试打印出 `Rectangle` 实例

当我们运行这个代码时，会出现带有如下核心信息的错误：

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

`println!` 宏能处理很多类型的格式，不过，`{}` 默认告诉 `println!` 使用被称为 `Display` 的格式：意在提供给直接终端用户查看的输出。目前为止见过的基本类型都默认实现了 `Display`，因为它就是向用户展示 `1` 或其他任何基本类型的唯一方式。不过对于结构体，`println!` 应该用来输出的格式是不明确的，因为这有更多显示的可能性：是否需要逗号？需要打印出大括号吗？所有字段都应该显示吗？由于这种不确定性，Rust 不会尝试猜测我们的意图，所以结构体并没有提供一个 `Display` 实现来使用 `println!` 与 `{}` 占位符。

但是如果我们继续阅读错误，将会发现这个有帮助的信息：

```
   = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
```

让我们来试试！现在 `println!` 宏调用看起来像 `println!("rect1 is {:?}", rect1);` 这样。在 `{}` 中加入 `:?` 指示符告诉 `println!` 我们想要使用叫做 `Debug` 的输出格式。`Debug` 是一个 trait，它允许我们以一种对开发者有帮助的方式打印结构体，以便当我们调试代码时能看到它的值。

这样调整后再次运行程序。见鬼了！仍然能看到一个错误：

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

不过编译器又一次给出了一个有帮助的信息：

```
   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`
```

Rust **确实** 包含了打印出调试信息的功能，不过我们必须为结构体显式选择这个功能。为此，在结构体定义之前加上外部属性 `#[derive(Debug)]`，如示例 5-12 所示：

文件名：src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {rect1:?}");
}
```

示例 5-12：增加属性来派生 `Debug` trait，并使用调试格式打印 `Rectangle` 实例

现在我们再运行这个程序时，就不会有任何错误，并会出现如下输出：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

好极了！这并不是最漂亮的输出，不过它显示这个实例的所有字段，毫无疑问这对调试有帮助。当我们有一个更大的结构体时，能有更易读一点的输出就好了，为此可以使用 `{:#?}` 替换 `println!` 字符串中的 `{:?}`。在这个例子中使用 `{:#?}` 风格将会输出如下：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

另一种使用 Debug 格式打印数值的方法是使用 dbg! 宏。dbg! 宏接收一个表达式的所有权（与 println! 宏相反，后者接收的是引用），打印出代码中调用 dbg! 宏时所在的文件和行号，以及该表达式的结果值，并返回该值的所有权。

> 注意：调用 dbg! 宏会打印到标准错误控制台流（stderr），与 println! 不同，后者会打印到标准输出控制台流（stdout）。我们将在第十二章“将错误信息写入标准错误而不是标准输出”一节中更多地讨论 stderr 和 stdout。

下面是一个例子，我们对分配给 width 字段的值以及 rect1 中整个结构的值感兴趣。

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

我们可以把 dbg! 放在表达式 30 * scale 周围，因为 dbg! 返回表达式的值的所有权，所以 width 字段将获得相同的值，就像我们在那里没有 dbg! 调用一样。我们不希望 dbg! 拥有 rect1 的所有权，所以我们在下一次调用 dbg! 时传递一个引用。下面是这个例子的输出结果：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10:16] 30 * scale = 60
[src/main.rs:14:5] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

我们可以看到第一点输出来自 *src/main.rs* 第 10 行，我们正在调试表达式 `30 * scale`，其结果值是 `60`（为整数实现的 `Debug` 格式化是只打印它们的值）。在 *src/main.rs* 第 14 行 的 `dbg!` 调用输出 `&rect1` 的值，即 `Rectangle` 结构。这个输出使用了更为易读的 `Debug` 格式。当你试图弄清楚你的代码在做什么时，`dbg!` 宏可能真的很有帮助！

除了 `Debug` trait，Rust 还为我们提供了很多可以通过 `derive` 属性来使用的 trait，它们可以为我们的自定义类型增加实用的行为。附录 C 中列出了这些 trait 和行为。第十章会介绍如何通过自定义行为来实现这些 trait，同时还有如何创建你自己的 trait。除了 `derive` 之外，还有很多属性；更多信息请参见 Rust Reference 的 Attributes 部分。

我们的 `area` 函数用途非常专一：它仅计算长方形的面积。如果这个行为与 `Rectangle` 结构体再结合得更紧密一些就更好了，因为它不能用于其他类型。现在让我们看看如何继续重构这些代码，来将 `area` 函数协调进 `Rectangle` 类型定义的 `area` **方法** 中。

方法语法

方法（method）与函数类似：它们使用 `fn` 关键字和名称声明，可以拥有参数和返回值，同时包含在某处调用该方法时会执行的代码。不过方法与函数是不同的，因为它们在结构体的上下文中被定义（或者是枚举或 trait 对象的上下文，将分别在第六章和第十八章讲解），并且它们第一个参数总是 `self`，它代表调用该方法的结构体实例。

定义方法

让我们把前面实现的获取一个 `Rectangle` 实例作为参数的 `area` 函数，改写成一个定义于 `Rectangle` 结构体上的 `area` 方法，如示例 5-13 所示：

文件名：src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

示例 5-13：在 `Rectangle` 结构体上定义 `area` 方法

为了使函数定义于 `Rectangle` 的上下文中，我们开始了一个 `impl` 块（`impl` 是 *implementation* 的缩写），这个 `impl` 块中的所有内容都将与 `Rectangle` 类型相关联。接着将 `area` 函数移动到 `impl` 大括号中，并将签名中的第一个（在这里也是唯一一个）参数和函数体中其他地方的对应参数改成 `self`。然后在 `main` 中将我们先前调用 `area` 方法并传递 `rect1` 作为参数的地方，改成使用 **方法语法**（*method syntax*）在 `Rectangle` 实例上调用 `area` 方法。方法语法获取一个实例并加上一个点号，后跟方法名、圆括号以及任何参数。

在 `area` 的签名中，使用 `&self` 来替代 `rectangle: &Rectangle`，`&self` 实际上是 `self: &Self` 的缩写。在一个 `impl` 块中，`Self` 类型是 `impl` 块的类型的别名。方法的第一个参数必须有一个名为 `self` 的`Self` 类型的参数，所以 Rust 让你在第一个参数位置上只用 `self` 这个名字来简化。注意，我们仍然需要在 `self` 前面使用 `&` 来表示这个方法借用了 `Self` 实例，就像我们在 `rectangle: &Rectangle` 中做的那样。方法可以选择获得 `self` 的所有权，或者像我们这里一样不可变地借用 `self`，或者可变地借用 `self`，就跟其他参数一样。

这里选择 `&self` 的理由跟在函数版本中使用 `&Rectangle` 是相同的：我们并不想获取所有权，只希望能够读取结构体中的数据，而不是写入。如果想要在方法中改变调用方法的实例，需要将第一个参数改为 `&mut self`。通过仅仅使用 `self` 作为第一个参数来使方法获取实例的所有权是很少见的；这种技术通常用在当方法将 `self` 转换成别的实例的时候，这时我们想要防止调用者在转换之后使用原始的实例。

使用方法替代函数，除了可使用方法语法和不需要在每个函数签名中重复 `self` 的类型之外，其主要好处在于组织性。我们将某个类型实例能做的所有事情都一起放入 `impl` 块中，而不是让将来的用户在我们的库中到处寻找 `Rectangle` 的功能。

请注意，我们可以选择将方法的名称与结构中的一个字段相同。例如，我们可以在 `Rectangle` 上定义一个方法，并命名为 `width`：

文件名：src/main.rs

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

在这里，我们选择让 `width` 方法在实例的 `width` 字段的值大于 `0` 时返回 `true`，等于 `0` 时则返回 `false`：我们可以出于任何目的，在同名的方法中使用同名的字段。在 `main` 中，当我们在 `rect1.width` 后面加上括号时。Rust 知道我们指的是方法 `width`。当我们不使用圆括号时，Rust 知道我们指的是字段 `width`。

通常，但并不总是如此，与字段同名的方法将被定义为只返回字段中的值，而不做其他事情。这样的方法被称为 *getters*，Rust 并不像其他一些语言那样为结构字段自动实现它们。Getters 很有用，因为你可以把字段变成私有的，但方法是公共的，这样就可以把对字段的只读访问作为该类型公共 API 的一部分。我们将在第七章中讨论什么是公有和私有，以及如何将一个字段或方法指定为公有或私有。

> **`->` 运算符到哪去了？**
>
> 在 C/C++ 语言中，有两个不同的运算符来调用方法：`.` 直接在对象上调用方法，而 `->` 在一个对象的指针上调用方法，这时需要先解引用（dereference）指针。换句话说，如果 `object` 是一个指针，那么 `object->something()` 就像 `(*object).something()` 一样。

Rust 并没有一个与 `->` 等效的运算符；相反，Rust 有一个叫 **自动引用和解引用**（*automatic referencing and dereferencing*）的功能。方法调用是 Rust 中少数几个拥有这种行为的地方。

它是这样工作的：当使用 `object.something()` 调用方法时，Rust 会自动为 `object` 添加 `&`、`&mut` 或 `*` 以便使 `object` 与方法签名匹配。也就是说，这些代码是等价的：

```
#
#
p1.distance(&p2);
(&p1).distance(&p2);
```

第一行看起来简洁的多。这种自动引用的行为之所以有效，是因为方法有一个明确的接收者———— `self` 的类型。在给出接收者和方法名的前提下，Rust 可以明确地计算出方法是仅仅读取（`&self`），做出修改（`&mut self`）或者是获取所有权（`self`）。事实上，Rust 对方法接收者的隐式借用让所有权在实践中更友好。

带有更多参数的方法

让我们通过实现 `Rectangle` 结构体上的另一方法来练习使用方法。这回，我们让一个 `Rectangle` 的实例获取另一个 `Rectangle` 实例，如果 `self` （第一个 `Rectangle`）能完全包含第二个长方形则返回 `true`；否则返回 `false`。一旦我们定义了 `can_hold` 方法，就可以编写示例 5-14 中的代码。

文件名：src/main.rs

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
```

```
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

示例 5-14：使用还未实现的 `can_hold` 方法

同时我们希望看到如下输出，因为 `rect2` 的两个维度都小于 `rect1`，而 `rect3` 比 `rect1` 要宽：

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

因为我们想定义一个方法，所以它应该位于 `impl Rectangle` 块中。方法名是 `can_hold`，并且它会获取另一个 `Rectangle` 的不可变借用作为参数。通过观察调用方法的代码可以看出参数是什么类型的：`rect1.can_hold(&rect2)` 传入了 `&rect2`，它是一个 `Rectangle` 的实例 `rect2` 的不可变借用。这是可以理解的，因为我们只需要读取 `rect2`（而不是写入，这意味着我们需要一个不可变借用），而且希望 `main` 保持 `rect2` 的所有权，这样就可以在调用这个方法后继续使用它。`can_hold` 的返回值是一个布尔值，其实现会分别检查 `self` 的宽高是否都大于另一个 `Rectangle`。让我们在示例 5-13 的 `impl` 块中增加这个新的 `can_hold` 方法，如示例 5-15 所示：

文件名：src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

示例 5-15：在 `Rectangle` 上实现 `can_hold` 方法，它获取另一个 `Rectangle` 实例作为参数

如果结合示例 5-14 的 `main` 函数来运行，就会看到期望的输出。在方法签名中，可以在 `self` 后增加多个参数，而且这些参数就像函数中的参数一样工作。

关联函数

所有在 `impl` 块中定义的函数被称为 **关联函数**（*associated functions*），因为它们与 `impl` 后面命名的类型相关。我们可以定义不以 `self` 为第一参数的关联函数（因此不是方法），因为它们并不作用于一个结构体的实例。我们已经使用了一个这样的函数：在 `String` 类型上定义的 `String::from` 函数。

不是方法的关联函数经常被用作返回一个结构体新实例的构造函数。这些函数的名称通常为 `new` ，但 `new` 并不是一个关键字。例如我们可以提供一个叫做 `square` 关联函数，它接受一个维度参数并且同时作为宽和高，这样可以更轻松的创建一个正方形 `Rectangle` 而不必指定两次同样的值：

文件名：src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

关键字 `Self` 在函数的返回类型和函数体中，都是对 `impl` 关键字后所示类型的别名，这里是 `Rectangle`。

要调用这个关联函数，我们使用结构体名和 `::` 语法；比如 `let sq = Rectangle::square(3);`。这个函数位于结构体的命名空间中：`::` 语法用于关联函数和模块创建的命名空间。第七章会讲到模块。

多个 `impl` 块

每个结构体都允许拥有多个 `impl` 块。例如，示例 5-15 中的代码等同于示例 5-16 中所示的代码，但后者每个方法有其自己的 `impl` 块。

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

示例 5-16：使用多个 `impl` 块重写示例 5-15

这里没有理由将这些方法分散在多个 `impl` 块中，不过这是有效的语法。第十章讨论泛型和 trait 时会看到实用的多 `impl` 块的用例。

总结

结构体让你可以创建出在你的领域中有意义的自定义类型。通过结构体，我们可以将相关联的数据片段联系起来并命名它们，这样可以使得代码更加清晰。在 `impl` 块中，你可以定义与你的类型相关联的函数，而方法是一种相关联的函数，让你指定结构体的实例所具有的行为。

但结构体并不是创建自定义类型的唯一方法：让我们转向 Rust 的枚举功能，为你的工具箱再添一个工具。

枚举和模式匹配

本章介绍**枚举**（*enumerations*），也被称作 *enums*。枚举允许你通过列举可能的 **变体**（*variants*）来定义一个类型。首先，我们会定义并使用一个枚举来展示它是如何连同数据一起编码信息的。接下来，我们会探索一个特别有用的枚举，叫做 `Option`，它代表一个值要么是某个值要么什么都不是。然后会讲到在 `match` 表达式中用模式匹配，针对不同的枚举值编写相应要执行的代码。最后会介绍 `if let`，另一个简洁方便处理代码中枚举的结构。

枚举的定义

结构体给予你将字段和数据聚合在一起的方法，像 `Rectangle` 结构体有 `width` 和 `height` 两个字段。而枚举给予你一个途径去声明某个值是一个集合中的一员。比如，我们想让 `Rectangle` 是一些形状的集合，包含 `Circle` 和 `Triangle` 。为此，Rust 允许我们将这些可能性编码为一个枚举类型。

让我们看看一个需要诉诸于代码的场景，来考虑为何此时使用枚举更为合适且实用。假设我们要处理 IP 地址。目前被广泛使用的两个主要 IP 标准：IPv4（version four）和 IPv6（version six）。这是我们的程序可能会遇到的所有可能的 IP 地址类型：所以可以**枚举**出所有可能的值，这也正是枚举一词的由来。

任何一个 IP 地址要么是 IPv4 的要么是 IPv6 的，而且不能两者都是。IP 地址的这个特性使得枚举数据结构非常适合这个场景，因为枚举值只可能是其中一个变体。IPv4 和 IPv6 从根本上讲仍是 IP 地址，所以当代码在处理适用于任何类型的 IP 地址的场景时应该把它们当作相同的类型。

可以通过在代码中定义一个 `IpAddrKind` 枚举来表现这个概念并列出可能的 IP 地址类型，`V4` 和 `V6`。这被称为枚举的**变体**（*variants*）：

```
enum IpAddrKind {
    V4,
    V6,
}
```

现在 `IpAddrKind` 就是一个可以在代码中使用的自定义数据类型了。

枚举值

可以像这样创建 `IpAddrKind` 两个不同变体的实例：

```
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
```

注意枚举的变体位于其标识符的命名空间中，并使用两个冒号分开。这么设计的益处是现在 `IpAddrKind::V4` 和 `IpAddrKind::V6` 都是 `IpAddrKind` 类型的。例如，接着可以定义一个函数来接收任何 `IpAddrKind`类型的参数：

```
fn route(ip_kind: IpAddrKind) {}
```

现在可以使用任一变体来调用这个函数：

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

使用枚举甚至还有更多优势。进一步考虑一下我们的 IP 地址类型，目前没有一个存储实际 IP 地址**数据**的方法；只知道它是什么**类型**的。考虑到已经在第五章学习过结构体了，你可能会像示例 6-1 那样尝试用结构体来解决这个问题：

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

示例 6-1：将 IP 地址的数据和 `IpAddrKind` 变体存储在一个 `struct` 中

这里我们定义了一个有两个字段的结构体 `IpAddr`：`IpAddrKind`（之前定义的枚举）类型的 `kind` 字段和 `String` 类型 `address` 字段。我们有这个结构体的两个实例。第一个，`home`，它的 `kind` 的值是 `IpAddrKind::V4` 与之相关联的地址数据是 `127.0.0.1`。第二个实例，`loopback`，`kind` 的值是 `IpAddrKind` 的另一个变体，`V6`，关联的地址是 `::1`。我们使用了一个结构体来将 `kind` 和 `address` 打包在一起，现在枚举变体就与值相关联了。

我们可以使用一种更简洁的方式来表达相同的概念，仅仅使用枚举并将数据直接放进每一个枚举变体而不是将枚举作为结构体的一部分。`IpAddr` 枚举的新定义表明了 `V4` 和 `V6` 变体都关联了 `String` 值：

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

我们直接将数据附加到枚举的每个变体上，这样就不需要一个额外的结构体了。这里也很容易看出枚举工作的另一个细节：每一个我们定义的枚举变体的名字也变成了一个构建枚举的实例的函数。也就是说，`IpAddr::V4()` 是一个获取 `String` 参数并返回 `IpAddr` 类型实例的函数调用。作为定义枚举的结果，这些构造函数会自动被定义。

用枚举替代结构体还有另一个优势：每个变体可以处理不同类型和数量的数据。IPv4 版本的 IP 地址总是含有四个值在 0 和 255 之间的数字部分。如果我们想要将 `V4` 地址存储为四个 `u8` 值而 `V6` 地址仍然表现为一个 `String`，这就不能使用结构体了。枚举则可以轻易的处理这个情况：

```
```

```
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
    }

    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
```

这些代码展示了使用枚举来存储两种不同 IP 地址的几种可能的选择。然而，事实证明存储和编码 IP 地址实在是太常见了以致标准库提供了一个开箱即用的定义！让我们看看标准库是如何定义 `IpAddr` 的：它正有着跟我们定义和使用的一样的枚举和变体，不过它将变体中的地址数据嵌入到了两个不同形式的结构体中，它们对不同的变体的定义是不同的：

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

这些代码展示了可以将任意类型的数据放入枚举变体中：例如字符串、数字类型或者结构体。甚至可以包含另一个枚举！另外，标准库中的类型通常并不比你设想出来的要复杂多少。

注意虽然标准库中包含一个 `IpAddr` 的定义，仍然可以创建和使用我们自己的定义而不会有冲突，因为我们并没有将标准库中的定义引入作用域。第七章会讲到如何导入类型。

来看看示例 6-2 中的另一个枚举的例子：它的变体中内嵌了多种多样的类型：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

示例 6-2：一个 `Message` 枚举，其每个变体都存储了不同数量和类型的值

这个枚举有四个含有不同类型的变体：

- `Quit` 没有关联任何数据。
- `Move` 类似结构体包含命名字段。
- `Write` 包含单独一个 `String`。
- `ChangeColor` 包含三个 `i32`。

定义一个如示例 6-2 中所示那样的有关联值的枚举的方式和定义多个不同类型的结构体的方式很相像，除了枚举不使用 `struct` 关键字以及其所有变体都被组合在一起位于 `Message` 类型下。如下这些结构体可以包含与之前枚举变体中相同的数据：

```
struct QuitMessage; // 类单元结构体
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 元组结构体
struct ChangeColorMessage(i32, i32, i32); // 元组结构体
```

不过，如果我们使用不同的结构体，由于它们都有不同的类型，我们将不能像使用示例 6-2 中定义的 `Message` 枚举那样，轻易的定义一个能够处理这些不同类型的结构体的函数，因为枚举是单独一个类型。

结构体和枚举还有另一个相似点：就像可以使用 `impl` 来为结构体定义方法那样，也可以在枚举上定义方法。这是一个定义于我们 `Message` 枚举上的叫做 `call` 的方法：

```
    impl Message {
        fn call(&self) {
            // 在这里定义方法体
        }
    }

    let m = Message::Write(String::from("hello"));
    m.call();
```

方法体使用了 `self` 来获取调用方法的值。这个例子中，创建了一个值为 `Message::Write(String::from("hello"))` 的变量 `m`，而且这就是当 `m.call()` 运行时 `call` 方法中的 `self` 的值。

让我们看看标准库中的另一个非常常见且实用的枚举：`Option`。

`Option` 枚举及其相对于空值的优势

这一部分会分析一个 `Option` 的案例，`Option` 是标准库定义的另一个枚举。`Option` 类型应用广泛因为它编码了一个非常普遍的场景，即一个值要么有值要么没值。

例如，如果请求一个非空列表的第一项，会得到一个值，如果请求一个空的列表，就什么也不会得到。从类型系统的角度来表达这个概念就意味着编译器需要检查是否处理了所有应该处理的情况，这样就可以避免在其他编程语言中非常常见的 bug。

编程语言的设计经常要考虑包含哪些功能，但考虑排除哪些功能也很重要。Rust 并没有很多其他语言中有的空值功能。**空值**（*Null*）是一个值，它代表没有值。在有空值的语言中，变量总是这两种状态之一：空值和非空值。

Tony Hoare，null 的发明者，在他 2009 年的演讲 “Null References: The Billion Dollar Mistake” 中曾经说到：

> I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> 我称之为我十亿美元的错误。当时，我在为一个面向对象语言设计第一个综合性的面向引用的类型系统。我的目标是通过编译器的自动检查来保证所有引用的使用都应该是绝对安全的。不过我未能抵抗住引入一个空引用的诱惑，仅仅是因为它是这么的容易实现。这引发了无数错误、漏洞和系统崩溃，在过去四十年里可能造成了价值十亿美元的痛苦和损失。

空值的问题在于当你尝试像一个非空值那样使用一个空值，会出现某种形式的错误。因为空和非空的属性无处不在，非常容易出现这类错误。

然而，空值尝试表达的概念仍然是有意义的：空值是一个因为某种原因目前无效或缺失的值。

问题不在于概念而在于具体的实现。为此，Rust 并没有空值，不过它确实拥有一个可以编码存在或不存在概念的枚举。这个枚举是 `Option<T>`，而且它定义于标准库中，如下：

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option<T>` 枚举是如此有用以至于它甚至被包含在了 prelude 之中，无需将其显式引入作用域。另外，它的变体也是如此：可以不需要 `Option::` 前缀来直接使用 `Some` 和 `None`。即便如此 `Option<T>` 也仍是常规的枚举，`Some(T)` 和 `None` 仍是 `Option<T>` 的变体。

`<T>` 语法是一个我们还未讲到的 Rust 功能。它是一个泛型类型参数，第十章会更详细的讲解泛型。目前，所有你需要知道的就是 `<T>` 意味着 `Option` 枚举的 `Some` 变体可以包含任意类型的数据，同时每一个用于 `T` 位置的具体类型使得 `Option<T>` 整体作为不同的类型。这里是一些包含数字类型和字符串类型 `Option` 值的例子：

```
    let some_number = Some(5);
    let some_char = Some('e');

    let absent_number: Option<i32> = None;
```

`some_number` 的类型是 `Option<i32>`。`some_char` 的类型是 `Option<char>`，是不同于 `some_number` 的类型。因为我们在 `Some` 变体中指定了值，Rust 可以推断其类型。对于 `absent_number`，Rust 需要我们指定 `Option` 整体的类型，因为编译器只通过 `None` 值无法推断出 `Some` 变体保存的值的类型。这里我们告诉 Rust 希望 `absent_number` 是 `Option<i32>` 类型的。

当有一个 `Some` 值时，我们就知道存在一个值，而这个值保存在 `Some` 中。当有个 `None` 值时，在某种意义上，它跟空值具有相同的意义：并没有一个有效的值。那么，`Option<T>` 为什么就比空值要好呢？

简而言之，因为 `Option<T>` 和 `T`（这里 `T` 可以是任何类型）是不同的类型，编译器不允许像一个肯定有效的值那样使用 `Option<T>`。例如，这段代码不能编译，因为它尝试将 `Option<i8>` 与 `i8` 相加：

```
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let sum = x + y;
```



如果运行这些代码，将得到类似这样的错误信息：

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
  = help: the following other types implement trait `Add<Rhs>`:
            `&i8` implements `Add<i8>`
            `&i8` implements `Add`
            `i8` implements `Add<&i8>`
            `i8` implements `Add`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

很好！事实上，错误信息意味着 Rust 不知道该如何将 `Option<i8>` 与 `i8` 相加，因为它们的类型不同。当在 Rust 中拥有一个像 `i8` 这样类型的值时，编译器确保它总是有一个有效的值。我们可以自信地使用而无需做空值检查。只有当使用 `Option<i8>`（或者任何用到的类型）的时候需要担心可能没有值，而编译器会确保我们在使用值之前处理了为空的情况。

换句话说，在对 `Option<T>` 进行运算之前必须将其转换为 `T`。通常这能帮助我们捕获到空值最常见的问题之一：假设某值不为空但实际上为空的情况。

消除了错误地假设一个非空值的风险，会让你对代码更加有信心。为了拥有一个可能为空的值，你必须要显式的将其放入对应类型的 `Option<T>` 中。接着，当使用这个值时，必须明确的处理值为空的情况。只要一个值不是 `Option<T>` 类型，你就**可以**安全的认定它的值不为空。这是 Rust 的一个经过深思熟虑的设计决策，来限制空值的泛滥以增加 Rust 代码的安全性。

那么当有一个 `Option<T>` 的值时，如何从 `Some` 变体中取出 `T` 的值来使用它呢？`Option<T>` 枚举拥有大量用于各种情况的方法：你可以查看它的文档。熟悉 `Option<T>` 的方法将对你的 Rust 之旅非常有用。

总的来说，为了使用 `Option<T>` 值，需要编写处理每个变体的代码。你想要一些代码只当拥有 `Some(T)` 值时运行，允许这些代码使用其中的 `T`。也希望一些代码只在值为 `None` 时运行，这

些代码并没有一个可用的 `T` 值。`match` 表达式就是这么一个处理枚举的控制流结构：它会根据枚举的变体运行不同的代码，这些代码可以使用匹配到的值中的数据。

`match` 控制流结构

Rust 有一个叫做 `match` 的极为强大的控制流运算符，它允许我们将一个值与一系列的模式相比较，并根据相匹配的模式执行相应代码。模式可由字面值、变量、通配符和许多其他内容构成；第十九章会涉及到所有不同种类的模式以及它们的作用。`match` 的力量来源于模式的表现力，以及编译器能够确认所有可能情况均已被覆盖。

可以把 `match` 表达式想象成某种硬币分类器：硬币滑入有着不同大小孔洞的轨道，每一个硬币都会掉入符合它大小的孔洞。同样地，值也会通过 `match` 的每一个模式，并且在遇到第一个“符合”的模式时，值会进入相关联的代码块并在执行中被使用。

因为刚刚提到了硬币，让我们用它们来作为一个使用 `match` 的例子！我们可以编写一个函数来获取一个未知的美国硬币，并以一种类似验钞机的方式，确定它是何种硬币并返回它的美分值，如示例 6-3 中所示。

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

示例 6-3：一个枚举和一个以枚举变体作为模式的 `match` 表达式

拆开 `value_in_cents` 函数中的 `match` 来看。首先，我们列出 `match` 关键字后跟一个表达式，在这个例子中是 `coin` 的值。这看起来非常像 `if` 所使用的条件表达式，不过这里有一个非常大的区别：对于 `if`，表达式必须返回一个布尔值，而这里它可以是任何类型的。例子中的 `coin` 的类型是示例 6-3 中定义的 `Coin` 枚举。

接下来是 `match` 的分支。一个分支有两个部分：一个模式和一些代码。第一个分支的模式是值 `Coin::Penny` 而之后的 `=>` 运算符将模式和将要运行的代码分开。这里的代码就仅仅是值 `1`。每一个分支之间使用逗号分隔。

当 `match` 表达式执行时，它将结果值按顺序与每一个分支的模式相比较。如果模式匹配了这个值，这个模式相关联的代码将被执行。如果模式并不匹配这个值，将继续执行下一个分支，非常类似一个硬币分类器。可以拥有任意多的分支：示例 6-3 中的 `match` 有四个分支。

每个分支相关联的代码是一个表达式，而表达式的结果值将作为整个 `match` 表达式的返回值。

如果分支代码较短的话通常不使用大括号，正如示例 6-3 中的每个分支都只是返回一个值。如果想要在分支中运行多行代码，可以使用大括号，而分支后的逗号是可选的。例如，如下代码在每次使用`Coin::Penny` 调用时都会打印出 “Lucky penny!”，同时仍然返回代码块最后的值，`1`：

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

绑定值的模式

匹配分支的另一个有用的功能是可以绑定匹配的模式的部分值。这也就是如何从枚举变体中提取值的。

作为一个例子，让我们修改枚举的一个变体来存放数据。1999 年到 2008 年间，美国在 25 美分的硬币的一侧为 50 个州的每一个都印刷了不同的设计。其他的硬币都没有这种区分州的设计，所以只有这些 25 美分硬币有特殊的价值。可以将这些信息加入我们的 `enum`，通过改变 `Quarter` 变体来包含一个 `State` 值，示例 6-4 中完成了这些修改：

```
#[derive(Debug)] // 这样可以立刻看到州的名称
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

示例 6-4：`Quarter` 变体也存放了一个 `UsState` 值的 `Coin` 枚举

想象一下我们的一个朋友尝试收集所有 50 个州的 25 美分硬币。在根据硬币类型分类零钱的同时，也可以报告出每个 25 美分硬币所对应的州名称，这样如果我们的朋友没有的话，他可以将其加入收藏。

在这些代码的匹配表达式中，我们在匹配 `Coin::Quarter` 变体的分支的模式中增加了一个叫做 `state` 的变量。当匹配到 `Coin::Quarter` 时，变量 `state` 将会绑定 25 美分硬币所对应州的值。接着在那个分支的代码中使用 `state`，如下：

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {state:?}!");
```

```
            25
        }
    }
}
```

如果调用 `value_in_cents(Coin::Quarter(UsState::Alaska))`，`coin` 将是 `Coin::Quarter(UsState::Alaska)`。当将值与每个分支相比较时，没有分支会匹配，直到遇到 `Coin::Quarter(state)`。这时，`state` 绑定的将会是值 `UsState::Alaska`。接着就可以在 `println!` 表达式中使用这个绑定了，像这样就可以获取 `Coin` 枚举的 `Quarter` 变体中内部的州的值。

匹配 `Option<T>`

我们在之前的部分中使用 `Option<T>` 时，是为了从 `Some` 中取出其内部的 `T` 值；我们还可以像处理 `Coin` 枚举那样使用 `match` 处理 `Option<T>`！只不过这回比较的不再是硬币，而是 `Option<T>` 的变体，但 `match` 表达式的工作方式保持不变。

比如我们想要编写一个函数，它获取一个 `Option<i32>`，如果其中含有一个值，将其加一。如果其中没有值，函数应该返回 `None` 值，而不尝试执行任何操作。

得益于 `match`，编写这个函数非常简单，它将看起来像示例 6-5 中这样：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

示例 6-5：一个在 `Option<i32>` 上使用 `match` 表达式的函数

让我们更仔细地检查 `plus_one` 的第一行操作。当调用 `plus_one(five)` 时，`plus_one` 函数体中的 `x` 将会是值 `Some(5)`。接着将其与每个分支比较。

```
            None => None,
```

值 `Some(5)` 并不匹配模式 `None`，所以继续进行下一个分支。

```
            Some(i) => Some(i + 1),
```

`Some(5)` 与 `Some(i)` 匹配吗？当然匹配！它们是相同的变体。`i` 绑定了 `Some` 中包含的值，所以 `i` 的值是 `5`。接着匹配分支的代码被执行，所以我们将 `i` 的值加一并返回一个含有值 `6` 的新 `Some`。

接着考虑下示例 6-5 中 `plus_one` 的第二个调用，这里 `x` 是 `None`。我们进入 `match` 并与第一个分支相比较。

```
            None => None,
```

匹配成功！这里没有值来加一，所以程序结束并返回 `=>` 右侧的值 `None`，因为第一个分支就匹配到了，其他的分支将不再比较。

将 `match` 与枚举相结合在很多场景中都是有用的。你会在 Rust 代码中看到很多这样的模式：`match` 一个枚举，绑定其中的值到一个变量，接着根据其值执行代码。这在一开始有点复杂，不过一旦习惯了，你会希望所有语言都拥有它！这一直是用户的最爱。

匹配是穷尽的

`match` 还有另一方面需要讨论：这些分支必须覆盖了所有的可能性。考虑一下 `plus_one` 函数的这个版本，它有一个 bug 并不能编译：

```
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
        }
    }
```



我们没有处理 `None` 的情况，所以这些代码会造成一个 bug。幸运的是，这是一个 Rust 知道如何处理的 bug。如果尝试编译这段代码，会得到这个错误：

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
 --> src/main.rs:3:15
  |
3 |         match x {
  |               ^ pattern `None` not covered
  |
note: `Option<i32>` defined here
 --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/core/src/
option.rs:572:1
 ::: /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/core/src/
option.rs:576:5
  |
  = note: not covered
  = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm with
a wildcard pattern or an explicit pattern as shown
  |
4 ~             Some(i) => Some(i + 1),
5 ~             None => todo!(),
  |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

Rust 知道我们没有覆盖所有可能的情况甚至知道哪些模式被忘记了！Rust 中的匹配是 **穷尽的**（*exhaustive*）：必须穷举到最后的可能性来使代码有效。特别的在这个 `Option<T>` 的例子中，Rust 防止我们忘记明确的处理 `None` 的情况，这让我们免于假设拥有一个实际上为空的值，从而使之前提到的价值亿万的错误不可能发生。

通配模式和 `_` 占位符

使用枚举，我们也可以针对少数几个特定值执行特殊操作，而对其他所有值采取默认操作。想象我们正在玩一个游戏，如果你掷出骰子的值为 3，角色不会移动，而是会得到一顶新奇的帽子。如果你掷出了 7，你的角色将失去一顶新奇的帽子。对于其他的数值，你的角色会在棋盘上移动相应的格子。这是一个实现了上述逻辑的 `match`，骰子的结果是硬编码而不是一个随机值，其他的逻辑部分使用了没有函数体的函数来表示，实现它们超出了本例的范围：

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

对于前两个分支，匹配模式是字面值 `3` 和 `7`，最后一个分支则涵盖了所有其他可能的值，模式是我们命名为 `other` 的一个变量。`other` 分支的代码通过将其传递给 `move_player` 函数来使用这个变量。

即使我们没有列出 `u8` 所有可能的值，这段代码依然能够编译，因为最后一个模式将匹配所有未被特殊列出的值。这种通配模式满足了 `match` 必须被穷尽的要求。请注意，我们必须将通配分支放在最后，因为模式是按顺序匹配的。如果我们在通配分支后添加其他分支，Rust 将会警告我们，因为此后的分支永远不会被匹配到。

Rust 还提供了一个模式，当我们不想使用通配模式获取的值时，请使用 `_`，这是一个特殊的模式，可以匹配任意值而不绑定到该值。这告诉 Rust 我们不会使用这个值，所以 Rust 也不会警告我们存在未使用的变量。

让我们改变游戏规则：现在，当你掷出的值不是 3 或 7 的时候，你必须再次掷出。这种情况下我们不需要使用这个值，所以我们改动代码使用 `_` 来替代变量 `other`：

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

这个例子也满足穷尽性要求，因为我们在最后一个分支中显式地忽略了其它值。我们没有忘记处理任何东西。

最后，让我们再次改变游戏规则，如果你掷出 3 或 7 以外的值，你的回合将无事发生。我们可以使用单元值（在“元组类型”一节中提到的空元组）作为 `_` 分支的代码：

```
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
```

在这里，我们明确告诉 Rust 我们不会使用与前面模式不匹配的值，并且这种情况下我们不想运行任何代码。

我们将在第十九章中介绍更多关于模式和匹配的内容。现在，让我们继续讨论 `if let` 语法，这在 `match` 表达式显得有些冗长时非常有用。

`if let` 和 `let else` 简洁控制流

`if let` 语法让我们以一种不那么冗长的方式结合 `if` 和 `let`，来处理只匹配一个模式的值而忽略其他模式的情况。考虑示例 6-6 中的程序，它匹配一个 `config_max` 变量中的 `Option<u8>` 值并只希望当值为 `Some` 变体时执行代码：

```
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {max}"),
        _ => (),
    }
```

示例 6-6：`match` 只关心当值为 `Some` 时执行代码

如果值是 `Some`，我们希望打印出 `Some` 变体中的值，这个值被绑定到模式中的 `max` 变量里。对于 `None` 值我们不希望做任何操作。为了满足 `match` 表达式（穷尽性）的要求，必须在处理完这唯一的变体后加上 `_ => ()`，这样也要增加很多繁琐的样板代码。

不过我们可以使用 `if let` 这种简洁的方式编写。如下代码与示例 6-6 中的 `match` 行为一致：

```
    let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {max}");
    }
```

`if let` 语法获取通过等号分隔的一个模式和一个表达式。它的工作方式与 `match` 相同，这里的表达式对应 `match` 而模式则对应第一个分支。在这个例子中，模式是 `Some(max)`，`max` 绑定为 `Some` 中的值。接着可以在 `if let` 代码块中使用 `max` 了，就跟在对应的 `match` 分支中一样。只有当值匹配该模式时，`if let` 块中的代码才会执行。

使用 `if let` 意味着编写更少代码，更少的缩进和更少的样板代码。然而，这样会失去 `match` 强制要求的穷尽性检查来确保你没有忘记处理某些情况。`match` 和 `if let` 之间的选择依赖特定的环境以及增加简洁度和失去穷尽性检查的权衡取舍。

换句话说，可以认为 `if let` 是 `match` 的一个语法糖，它当值匹配某一模式时执行代码而忽略所有其他值。

可以在 `if let` 中包含一个 `else`。`else` 块中的代码与 `match` 表达式中的 `_` 分支块中的代码相同，这样的 `match` 表达式就等同于 `if let` 和 `else`。回忆一下示例 6-4 中 `Coin` 枚举的定义，其 `Quarter` 变体也包含一个 `UsState` 值。如果想要计数所有不是 25 美分的硬币的同时也报告 25 美分硬币所属的州，可以使用这样一个 `match` 表达式：

```
    let mut count = 0;
    match coin {
        Coin::Quarter(state) => println!("State quarter from {state:?}!"),
        _ => count += 1,
    }
```

或者可以使用这样的 `if let` 和 `else` 表达式：

```
    let mut count = 0;
    if let Coin::Quarter(state) = coin {
        println!("State quarter from {state:?}!");
    } else {
        count += 1;
    }
```

使用 `let...else` 来保持在“愉快路径”（“Happy Path”）

当某个值存在时进行一些操作否则返回一个默认是一个常规操作。继续以处理 `UsState` 值的硬币例子来说，如果我们说一些有趣的事依赖于硬币的州有多老，我们可能会像这样在 `UsState` 上引入一个检查州龄的方法：

```
impl UsState {
    fn existed_in(&self, year: u16) -> bool {
        match self {
            UsState::Alabama => year >= 1819,
            UsState::Alaska => year >= 1959,
            // -- snip --
        }
    }
}
```

接着我们可能使用 `if let` 来匹配硬币的类型，在条件代码中引入一个 `state`，如示例 6-7 所示。

```
fn describe_state_quarter(coin: Coin) -> Option<String> {
    if let Coin::Quarter(state) = coin {
        if state.existed_in(1900) {
            Some(format!("{state:?} is pretty old, for America!"))
        } else {
            Some(format!("{state:?} is relatively new."))
        }
    } else {
        None
    }
}
```

这样固然可以完成任务，不过这将工作推进了 `if let` 语句中，如果需要完成的工作更为复杂，则可能难以追踪顶层分支是如何关联的。我们也可以利用这个表达式要么从 `if let` 中生成一个 `state` 要么提前返回的优势，如示例 6-8 所示。（使用 `match` 也可以实现类似效果。）

```
fn describe_state_quarter(coin: Coin) -> Option<String> {
    let state = if let Coin::Quarter(state) = coin {
        state
    } else {
        return None;
    };

    if state.existed_in(1900) {
        Some(format!("{state:?} is pretty old, for America!"))
```

```
    } else {
        Some(format!("{state:?} is relatively new."))
    }
}
```

不过这样写在某种程度上会让人觉得有些繁琐！`if let` 的一个分支产生一个值，而另一个分支则直接从函数中返回。

为了使这个通用模式更容易表达，Rust 提供了 `let...else`。`let...else` 语法左侧是一个模式，右侧是一个表达式，非常类似于 `if let`，不过它没有 `if` 分支，只有 `else` 分支。如果模式匹配，它会将匹配到的值绑定到外层作用域。如果模式**不**匹配，程序流会指向 `else` 分支，它必须从函数返回。

在示例 6-9 中，可以看到当在示例 6-8 中的 `if let` 替换为 `let...else` 时看起来如何。

```
fn describe_state_quarter(coin: Coin) -> Option<String> {
    let Coin::Quarter(state) = coin else {
        return None;
    };

    if state.existed_in(1900) {
        Some(format!("{state:?} is pretty old, for America!"))
    } else {
        Some(format!("{state:?} is relatively new."))
    }
}
```

注意它以这种方式在函数主体中保持了 “愉快路径”（“Happy Path”），而不用像 `if let` 那样在两个分支中拥有明显不同的控制流

如果你的程序遇到一个使用 `match` 表达起来过于冗长的逻辑，记住 `if let` 和 `let...else` 也在你的 Rust 工具箱中。

总结

现在我们涉及到了如何使用枚举来创建有一系列可列举值的自定义类型。我们也展示了标准库的 `Option<T>` 类型是如何帮助你利用类型系统来避免出错的。当枚举值包含数据时，你可以根据需要处理多少情况来选择使用 `match` 或 `if let` 来获取并使用这些值。

你的 Rust 程序现在能够使用结构体和枚举在自己的作用域内表现其内容了。在你的 API 中使用自定义类型保证了类型安全：编译器会确保你的函数只会得到它期望的类型的值。

为了向你的用户提供一个组织良好的 API，它使用起来很直观并且只向用户暴露他们确实需要的部分，那么现在就让我们转向 Rust 的模块系统吧。

使用包、Crate 和模块管理不断增长的项目

当你编写大型程序时，组织代码显得尤为重要。通过对相关功能进行分组和划分不同功能的代码，你可以清楚在哪里可以找到实现了特定功能的代码，以及在哪里可以改变一个功能的工作方式。

到目前为止，我们编写的程序都在一个文件的一个模块中。伴随着项目的增长，你应该通过将代码分解为多个模块和多个文件来组织代码。一个包（package）可以包含多个二进制 crate 项和一个可选的库 crate。伴随着包的增长，你可以将包中的部分代码提取出来，做成独立的 crate，这些 crate 则作为外部依赖项。本章将会涵盖所有这些概念。对于一个由一系列相互关联的包组成的超大型项目，Cargo 提供了**工作空间**（*workspaces*）这一功能，我们将在第十四章的 “Cargo Workspaces” 对此进行讲解。

我们也会讨论封装来实现细节，这可以让你在更高层面重用代码：你实现了一个操作后，其他的代码可以通过该代码的公共接口来进行调用，而不需要知道它是如何实现的。你在编写代码时可以定义哪些部分是其他代码可以使用的公共部分，以及哪些部分是你有权更改实现细节的私有部分。这是另一种减少你在脑海中记住项目内容数量的方法。

这里有一个需要说明的概念 “作用域（scope）”：代码所在的嵌套上下文有一组定义为 “in scope” 的名称。当阅读、编写和编译代码时，程序员和编译器需要知道特定位置的特定名称是否引用了变量、函数、结构体、枚举、模块、常量或者其他有意义的项。你可以创建作用域，以及改变哪些名称在作用域内还是作用域外。同一个作用域内不能拥有两个相同名称的项；可以使用一些工具来解决名称冲突。

Rust 有许多功能可以让你管理代码的组织，包括哪些细节可以被公开，哪些细节作为私有部分，以及程序中各个作用域中有哪些名称。这些特性，有时被统称为 “模块系统（the module system）”，包括：

- **包**（*Packages*）：Cargo 的一个功能，它允许你构建、测试和分享 crate。
- **Crates**：一个模块的树形结构，它形成了库或可执行文件项目。
- **模块**（*Modules*）和 **use**：允许你控制作用域和路径的私有性。
- **路径**（*path*）：一个为例如结构体、函数或模块等项命名的方式。

本章将会涵盖所有这些概念，讨论它们如何交互，并说明如何使用它们来管理作用域。到最后，你会对模块系统有深入的了解，并且能够像专业人士一样使用作用域！

包和 Crate

模块系统的第一部分，我们将介绍包和 crate。

crate 是 Rust 在编译时最小的代码单位。即使你用 `rustc` 而不是 `cargo` 来编译一个单独的源代码文件（正如我们在第 1 章“编写并运行 Rust 程序”中所做的那样），编译器还是会将那个文件视为一个 crate。crate 可以包含模块，模块可以定义在其他文件，然后和 crate 一起编译，我们会在接下来的章节中遇到。

crate 有两种形式：二进制 crate 和库 crate。**二进制 crate**（*Binary crates*）可以被编译为可执行程序，比如命令行程序或者服务端。它们必须有一个名为 `main` 函数来定义当程序被执行的时候所需要做的事情。目前我们所创建的 crate 都是二进制 crate。

库 crate（*Library crates*）并没有 `main` 函数，它们也不会编译为可执行程序。相反它们定义了可供多个项目复用的功能模块。比如 第二章 的 `rand` crate 就提供了生成随机数的功能。大多数时间 `Rustaceans` 说的 “crate” 指的都是库 crate，这与其他编程语言中 “library” 概念一致。

crate root 是一个源文件，Rust 编译器以它为起始点，并构成你的 crate 的根模块（我们将在 “定义模块来控制作用域与私有性” 一节深入解读）。

包（*package*）是提供一系列功能的一个或者多个 crate 的捆绑。一个包会包含一个 *Cargo.toml* 文件，阐述如何去构建这些 crate。Cargo 实际上就是一个包，它包含了用于构建你代码的命令行工具的二进制 crate。其他项目也依赖 Cargo 库来实现与 Cargo 命令行程序一样的逻辑。

包中可以包含至多一个库 crate(library crate)。包中可以包含任意多个二进制 crate(binary crate)，但是必须至少包含一个 crate（无论是库的还是二进制的）。

让我们来看看创建包的时候会发生什么。首先，我们输入命令 `cargo new my-project`：

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

运行了这条命令后，我们先用 `ls` （译者注：此命令为 Linux 平台的指令，Windows 下可用 dir）来看看 Cargo 给我们创建了什么，Cargo 会给我们的包创建一个 *Cargo.toml* 文件。查看 *Cargo.toml* 的内容，会发现并没有提到 *src/main.rs*，因为 Cargo 遵循的一个约定：*src/main.rs* 就是一个与包同名的二进制 crate 的 crate 根。同样的，Cargo 知道如果包目录中包含 *src/lib.rs*，则包带有与其同名的库 crate，且 *src/lib.rs* 是 crate 根。crate 根文件将由 Cargo 传递给 `rustc` 来实际构建库或者二进制项目。

在此，我们有了一个只包含 *src/main.rs* 的包，意味着它只含有一个名为 `my-project` 的二进制 crate。如果一个包同时含有 *src/main.rs* 和 *src/lib.rs*，则它有两个 crate：一个二进制的和一个库的，且名字都与包相同。通过将文件放在 *src/bin* 目录下，一个包可以拥有多个二进制 crate：每个 *src/bin* 下的文件都会被编译成一个独立的二进制 crate。

定义模块来控制作用域与私有性

在本节，我们将讨论模块和其它一些关于模块系统的部分，如允许你命名项的 路径（*paths*）；用来将路径引入作用域的 `use` 关键字；以及使项变为公有的 `pub` 关键字。我们还将讨论 `as` 关键字、外部包（external packages）和 glob 运算符（glob operator）。

首先，我们将从一系列的规则开始，在你未来组织代码的时候，这些规则可被用作简单的参考。接下来我们将会详细的解释每条规则。

模块小抄（Cheat Sheet）

在深入了解模块和路径的细节之前，这里提供一个简单的参考，用来解释模块、路径、`use`关键词和`pub`关键词如何在编译器中工作，以及大部分开发者如何组织他们的代码。我们将在本章中举例说明每条规则，但这是回顾模块工作原理的绝佳参考。

- **从 crate 根节点开始**: 当编译一个 crate, 编译器首先在 crate 根文件（通常，对于一个库 crate 而言是 *src/lib.rs*，对于一个二进制 crate 而言是 *src/main.rs*）中寻找需要被编译的代码。
- **声明模块**: 在 crate 根文件中，你可以声明一个新模块；比如，用 `mod garden;` 声明了一个叫做 `garden` 的模块。编译器会在下列路径中寻找模块代码：- 内联，用大括号替换 `mod garden` 后跟的分号
- 在文件 *src/garden.rs*
- 在文件 *src/garden/mod.rs*
- **声明子模块**: 在除了 crate 根节点以外的任何文件中，你可以定义子模块。比如，你可能在 *src/garden.rs* 中声明 `mod vegetables;`。编译器会在以父模块命名的目录中寻找子模块代码：- 内联，直接在 `mod vegetables` 后方不是一个分号而是一个大括号
- 在文件 *src/garden/vegetables.rs*
- 在文件 *src/garden/vegetables/mod.rs*
- **模块中的代码路径**: 一旦一个模块是你 crate 的一部分，你可以在隐私规则允许的前提下，从同一个 crate 内的任意地方，通过代码路径引用该模块的代码。举例而言，一个 garden vegetables 模块下的 `Asparagus` 类型可以通过 `crate::garden::vegetables::Asparagus` 访问。
- **私有 vs 公用**: 一个模块里的代码默认对其父模块私有。为了使一个模块公用，应当在声明时使用 `pub mod` 替代 `mod`。为了使一个公用模块内部的成员公用，应当在声明前使用`pub`。
- **`use` 关键字**: 在一个作用域内，`use`关键字创建了一个项的快捷方式，用来减少长路径的重复。在任何可以引用 `crate::garden::vegetables::Asparagus` 的作用域，你可以通过 `use crate::garden::vegetables::Asparagus;` 创建一个快捷方式，然后你就可以在作用域中只写 `Asparagus` 来使用该类型。

这里我们创建一个名为`backyard`的二进制 crate 来说明这些规则。该 crate 的路径同样命名为 `backyard`，该路径包含了这些文件和目录：

```
backyard
├── Cargo.lock
```

```
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

这个例子中的 crate 根文件是 *src/main.rs*，该文件包含了：

文件名：src/main.rs

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {plant:?}!");
}
```

`pub mod garden;` 行告诉编译器将 *src/garden.rs* 中发现的代码包含进来：

文件名：src/garden.rs

```
pub mod vegetables;
```

在此处，`pub mod vegetables;` 意味着在 *src/garden/vegetables.rs* 中的代码也应该被包含。这些代码是：

```
#[derive(Debug)]
pub struct Asparagus {}
```

现在让我们深入了解这些规则的细节并在实践中演示它们！

在模块中对相关代码进行分组

模块让我们可以将一个 crate 中的代码进行分组，以提高可读性与重用性。因为一个模块中的代码默认是私有的，所以还可以利用模块控制项的**私有性**（*privacy*）。私有项是不可为外部使用的内在详细实现。我们也可以将模块和它其中的项标记为公开的，这样，外部代码就可以使用并依赖于它们。

作为示例，让我们编写一个提供餐厅功能的库 `crate`。我们将定义函数的签名，但将其函数体留空以便将注意力集中在代码的组织结构上而不是餐厅实现的细节。

在餐饮业，餐馆中会有一些地方被称之为**前台**（*front of house*），还有另外一些地方被称之为**后台**（*back of house*）。前台是招待顾客的地方；这包括接待员为顾客安排座位、服务员接受点单和付款、调酒师制作饮品的地方。后台则是厨师和烹饪人员在厨房工作、洗碗工清理餐具，以及经理处理行政事务的区域。

为了以这种方式构建我们的 `crate`，我们可以将其功能组织到嵌套模块中。通过执行 `cargo new restaurant --lib` 来创建一个新的名为 `restaurant` 的库。然后将示例 7-1 中所罗列出来的代码放入 *src/lib.rs* 中，来定义一些模块和函数签名；这段代码即为前台部分。

文件名：src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

示例 7-1：一个包含了其他内置了函数的模块的 `front_of_house` 模块

我们使用 `mod` 关键字来定义模块，后跟模块名（本例中叫做 `front_of_house`），并且用花括号包围模块的主体。在模块内，我们还可以定义其它的模块，就像本例中的 `hosting` 和 `serving` 模块。模块还可以保存一些定义的其它项，比如结构体、枚举、常量、trait、或者如示例 7-1 所示的函数。

通过使用模块，我们可以将相关的定义分组到一起，并指出它们为什么相关。程序员可以通过使用这段代码，更加容易地找到他们想要的定义，因为他们可以基于分组来对代码进行导航，而不需要阅读所有的定义。程序员向这段代码中添加一个新的功能时，他们也会知道代码应该放置在何处，可以保持程序的组织性。

在前面我们提到了，`src/main.rs` 和 `src/lib.rs` 叫做 crate 根。之所以这样叫它们是因为这两个文件的内容都分别在 crate 模块结构的根组成了一个名为 `crate` 的模块，该结构被称为**模块树**（*module tree*）。

示例 7-2 展示了示例 7-1 中模块树的结构。

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

示例 7-2: 示例 7-1 中代码的模块树

这个树展示了一些模块是如何被嵌入到另一个模块的（例如，`hosting` 嵌套在 `front_of_house` 中）。这个树还展示了一些模块是互为**兄弟**（*siblings*）的，这意味着它们定义在同一模块中；`hosting` 和 `serving` 被一起定义在 `front_of_house` 中。继续沿用家庭关系的比喻，如果一个模块 A 被包含在模块 B 中，我们将模块 A 称为模块 B 的**子**（*child*）模块，模块 B 则是模块 A 的**父**（*parent*）模块。注意，整个模块树都植根于名为 `crate` 的隐式模块下。

这个模块树可能会令你想起电脑上文件系统的目录树；这是一个非常恰当的类比！就像文件系统的目录，你可以使用模块来组织你的代码。并且，就像目录中的文件，我们需要一种方法来找到模块。

引用模块树中项的路径

为了向 Rust 指示在模块树中从何处查找某个项，我们使用路径，就像在文件系统中使用路径一样。为了调用一个函数，我们需要知道它的路径。

路径有两种形式：

- **绝对路径**（*absolute path*）是以 crate 根（root）开头的完整路径；对于外部 crate 的代码，是以 crate 名开头的绝对路径，对于当前 crate 的代码，则以字面值 `crate` 开头。
- **相对路径**（*relative path*）从当前模块开始，以 `self`、`super` 或当前模块中的某个标识符开头。

绝对路径和相对路径都后跟一个或多个由双冒号（`::`）分割的标识符。

回到示例 7-1，假设我们希望调用 `add_to_waitlist` 函数。这相当于在问：`add_to_waitlist` 函数的路径是什么？在示例 7-3 中删除了示例 7-1 的一些模块和函数。

我们在 crate 根定义了一个新函数 `eat_at_restaurant`，并在其中展示调用 `add_to_waitlist` 函数的两种方法。这些路径都是正确的，不过因为存在另一个问题导致示例无法照原样编译。稍后我们会解释为什么。

`eat_at_restaurant` 函数是我们 crate 库的一个公共 API，所以我们使用 `pub` 关键字来标记它。在 “使用 `pub` 关键字暴露路径” 一节，我们将详细介绍 `pub`。

文件名：src/lib.rs



```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 绝对路径
    crate::front_of_house::hosting::add_to_waitlist();

    // 相对路径
    front_of_house::hosting::add_to_waitlist();
}
```

示例 7-3: 使用绝对路径和相对路径来调用 `add_to_waitlist` 函数

第一次在 `eat_at_restaurant` 中调用 `add_to_waitlist` 函数时，使用的是绝对路径。`add_to_waitlist` 函数与 `eat_at_restaurant` 被定义在同一 crate 中，这意味着我们可以使用 `crate` 关键字为起始的绝对路径。接着我们依次包含各级模块，直到我们找到 `add_to_waitlist`。你可以想象出一个相同结构的文件系统：我们通过指定路径 `/front_of_house/hosting/add_to_waitlist` 来执行 `add_to_waitlist` 程序。我们使用 `crate` 从 crate 根开始就类似于在 shell 中使用 `/` 从文件系统根开始。

第二次在 `eat_at_restaurant` 中调用 `add_to_waitlist` 时，使用的是相对路径。这个路径以 `front_of_house` 为起始，这个模块在模块树中与 `eat_at_restaurant` 定义在同一层级。与之等价的文件系统路径就是 `front_of_house/hosting/add_to_waitlist`。以模块名开头意味着该路径是相对路径。

选择使用相对路径还是绝对路径要取决于你的项目，也取决于你是更倾向于将项的定义代码与使用该项的代码分开来移动，还是一起移动。例如，如果我们要将 `front_of_house` 模块和 `eat_at_restaurant` 函数一起移动到一个名为 `customer_experience` 的模块中，我们需要更新 `add_to_waitlist` 的绝对路径，但是相对路径还是可用的。相反，如果我们要将 `eat_at_restaurant` 函数单独移到一个名为 `dining` 的模块中，还是可以使用原本的绝对路径来调用 `add_to_waitlist`，但是相对路径必须要更新。我们更倾向于使用绝对路径，因为把代码定义和项调用各自独立地移动是更常见的。

让我们试着编译一下示例 7-3，并查明其为何不能编译！示例 7-4 展示了这个错误。

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^  --------------- function `add_to_waitlist`
is not publicly re-exported
  |                            |
  |                            private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^  --------------- function `add_to_waitlist` is
not publicly re-exported
   |                     |
   |                     private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

示例 7-4: 构建示例 7-3 出现的编译器错误

错误信息说 `hosting` 模块是私有的。换句话说，我们拥有 `hosting` 模块和 `add_to_waitlist` 函数的正确路径，但是 Rust 不让我们使用，因为它不能访问私有片段。在 Rust 中，所有项（函数、方法、结构体、枚举、模块和常量）默认对父模块都是私有的。如果希望创建一个如函数或结构体的私有项，可以将其放入一个模块。

父模块中的项不能使用子模块中的私有项，但是子模块中的项可以使用它们父模块中的项。这是因为子模块封装并隐藏了它们的实现详情，但是子模块可以看到定义它们的上下文。继续我

们的比喻，把私有性规则想象成餐馆的后台办公室：后台的事务对餐厅顾客来说是不可知的，但办公室经理可以洞悉其经营的餐厅并在其中做任何事情。

Rust 选择以这种方式来实现模块系统功能，因此默认隐藏内部实现细节。这样一来，你就知道可以更改内部代码的哪些部分而不会破坏外部代码。不过 Rust 也确实提供了通过使用 `pub` 关键字来创建公共项，使子模块的内部部分暴露给上级模块。

使用 `pub` 关键字暴露路径

让我们回头看一下示例 7-4 的错误，它告诉我们 `hosting` 模块是私有的。我们想让父模块中的 `eat_at_restaurant` 函数可以访问子模块中的 `add_to_waitlist` 函数，因此我们使用 `pub` 关键字来标记 `hosting` 模块，如示例 7-5 所示。

文件名：src/lib.rs



```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

// -- snip --
```

示例 7-5: 使用 `pub` 关键字声明 `hosting` 模块使其可在 `eat_at_restaurant` 使用

不幸的是，示例 7-5 的代码编译仍然有错误，如示例 7-6 所示。

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:10:37
   |
10 |     crate::front_of_house::hosting::add_to_waitlist();
   |                                     ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:13:30
   |
13 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

示例 7-6: 构建示例 7-5 出现的编译器错误

发生了什么？在 `mod hosting` 前添加了 `pub` 关键字，使其变成公有的。伴随着这种变化，如果我们可以访问 `front_of_house`，那我们也可以访问 `hosting`。但是 `hosting` 的**内容**（*contents*）仍然是私有的；这表明使模块公有并不使其内容也是公有的。模块上的 `pub` 关键字只允许其父模块引用它，而不允许访问内部代码。因为模块是一个容器，只是将模块变为公有能做的其实并不太多；同时需要更深入地选择将一个或多个项变为公有。

示例 7-6 中的错误说，`add_to_waitlist` 函数是私有的。私有性规则不但应用于模块，还应用于结构体、枚举、函数和方法。

让我们继续将 `pub` 关键字放置在 `add_to_waitlist` 函数的定义之前，使其变成公有。如示例 7-7 所示。

文件名：src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

// -- snip --
```

示例 7-7: 为 `mod hosting` 和 `fn add_to_waitlist` 添加 `pub` 关键字使它们可以在 `eat_at_restaurant` 函数中被调用

现在代码可以编译通过了！为了了解为何增加 `pub` 关键字使得我们可以在 `eat_at_restaurant` 中调用这些路径与私有性规则有关，让我们看看绝对路径和相对路径。

在绝对路径，我们从 `crate` 也就是 crate 根开始。crate 根中定义了 `front_of_house` 模块。虽然 `front_of_house` 模块不是公有的，不过因为 `eat_at_restaurant` 函数与 `front_of_house` 定义于同一级模块中（即，`eat_at_restaurant` 和 `front_of_house` 是兄弟），我们可以从 `eat_at_restaurant` 中引用 `front_of_house`。接下来是使用 `pub` 标记的 `hosting` 模块。我们可以访问 `hosting` 的父模块，所以可以访问 `hosting`。最后，`add_to_waitlist` 函数被标记为 `pub` ，我们可以访问其父模块，所以这个函数调用是有效的！

在相对路径，其逻辑与绝对路径相同，除了第一步：不同于从 crate 根开始，路径从 `front_of_house` 开始。`front_of_house` 模块与 `eat_at_restaurant` 定义于同一级模块，所以从 `eat_at_restaurant` 中开始定义的该模块相对路径是有效的。接下来因为 `hosting` 和 `add_to_waitlist` 被标记为 `pub`，路径其余的部分也是有效的，因此函数调用也是有效的！

如果你计划共享你的库 crate 以便其它项目可以使用你的代码，公有 API 将是决定 crate 用户如何与你代码交互的契约。关于管理公有 API 的修改以便被人更容易依赖你的库有着很多考量。这些考量超出了本书的范畴；如果你对这些话题感兴趣，请查阅 The Rust API Guidelines。

> **二进制和库 crate 包的最佳实践**
>
> 我们提到过包（package）可以同时包含一个 *src/main.rs* 二进制 crate 根和一个 *src/lib.rs* 库 crate 根，并且这两个 crate 默认以包名来命名。通常，这种包含二进制

> crate 和库 crate 的模式的包，在二进制 crate 中只保留足以生成一个可执行文件的代码，并由可执行文件调用库 crate 的代码。又因为库 crate 可以共享，这使得其它项目从包提供的大部分功能中受益。
>
> 模块树应该定义在 *src/lib.rs* 中。这样通过以包名开头的路径，公有项就可以在二进制 crate 中使用。二进制 crate 就变得像一个一个完全外部的 crate 来使用库 crate 的用户一样：它只能使用 public API。你不仅仅是作者，也是用户！
>
> 在第十二章我们会通过一个同时包含二进制 crate 和库 crate 的命令行程序来展示这些组织上的实践。

`super` 开始的相对路径

我们可以通过在路径的开头使用 `super` ，从父模块开始构建相对路径，而不是从当前模块或者 crate 根开始。这类似以 `..` 语法开始一个文件系统路径。使用 `super` 允许我们引用父模块中的已知项，这使得当模块与父模块关联的很紧密，但某天父模块可能要移动到模块树的其它位置时重新组织模块树变得更容易。

考虑一下示例 7-8 中的代码，它模拟了厨师更正了一个错误订单并亲自将其提供给客户的情况。`back_of_house` 模块中的定义的 `fix_incorrect_order` 函数通过指定的 `super` 起始的 `deliver_order` 路径来调用父模块中的 `deliver_order` 函数。

文件名：src/lib.rs

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

示例 7-8: 使用以 `super` 开头的相对路径调用函数

`fix_incorrect_order` 函数在 `back_of_house` 模块中，所以我们可以使用 `super` 进入 `back_of_house` 父模块，也就是本例中的 `crate` 根。在这里，我们可以找到 `deliver_order`。成功！我们认为 `back_of_house` 模块和 `deliver_order` 函数之间可能保持某种关联关系并且如果我们要重新组织这个 crate 的模块树时，需要一起移动它们。因此，我们使用 `super`，这样一来，如果这些代码被移动到了其他模块，只需要更新很少的代码。

创建公有的结构体和枚举

我们还可以使用 `pub` 来设计公有的结构体和枚举，不过关于在结构体和枚举上使用 `pub` 还有一些额外的细节需要注意。如果我们在一个结构体定义的前面使用了 `pub`，这个结构体会变成公有的，但是这个结构体的字段仍然是私有的。我们可以根据情况决定每个字段是否公有。在示例 7-9 中，我们定义了一个公有结构体 `back_of_house::Breakfast`，其中有一个公有字段 `toast` 和私有字段 `seasonal_fruit`。这个例子模拟的情况是，在一家餐馆中，顾客可以选择随

餐面包的类型，但是厨师会根据季节和库存情况来决定随餐搭配的水果。餐馆可用的水果变化是很快的，所以顾客不能选择水果，甚至无法看到他们将会得到什么水果。

文件名：src/lib.rs

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // 在夏天订购一个黑麦土司作为早餐
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // 改变主意更换想要面包的类型
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // 如果取消下一行的注释代码不能编译；
    // 不允许查看或修改早餐附带的季节水果
    // meal.seasonal_fruit = String::from("blueberries");
}
```

示例 7-9: 带有公有和私有字段的结构体

因为 `back_of_house::Breakfast` 结构体的 `toast` 字段是公有的，所以我们可以在 `eat_at_restaurant` 中使用点号来读写 `toast` 字段。注意，我们不能在 `eat_at_restaurant` 中使用 `seasonal_fruit` 字段，因为 `seasonal_fruit` 是私有的。尝试去除那一行修改 `seasonal_fruit` 字段值的代码的注释，看看你会得到什么错误！

还请注意一点，因为 `back_of_house::Breakfast` 具有私有字段，所以这个结构体需要提供一个公共的关联函数来构造 `Breakfast` 的实例 (这里我们命名为 `summer`)。如果 `Breakfast` 没有这样的函数，我们将无法在 `eat_at_restaurant` 中创建 `Breakfast` 实例，因为我们不能在 `eat_at_restaurant` 中设置私有字段 `seasonal_fruit` 的值。

与之相反，如果我们将枚举设为公有，则它的所有变体都将变为公有。我们只需要在 `enum` 关键字前面加上 `pub`，就像示例 7-10 展示的那样。

文件名：src/lib.rs

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
```

```
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

示例 7-10: 设计公有枚举，使其所有成员公有

因为我们将 `Appetizer` 枚举声明为公有，所以可以在 `eat_at_restaurant` 中使用 `Soup` 和 `Salad` 变体。

如果枚举变体不是公有的，那么枚举会显得用处不大；给枚举的所有变体挨个添加 `pub` 是很令人恼火的，因此枚举变体默认就是公有的。结构体在许多情况下即使字段不可公有也能正常使用，所以结构体字段遵循默认私有的通用规则，除非使用 `pub` 关键字。

还有一个我们尚未介绍的与 `pub` 相关的情形，那就是模块系统的最后一个特性：`use` 关键字。我们将先单独介绍 `use`，然后展示如何结合使用 `pub` 和 `use`。

使用 `use` 关键字将路径引入作用域

不得不编写路径来调用函数显得繁琐且重复。在示例 7-7 中，无论我们选择 `add_to_waitlist` 函数的绝对路径还是相对路径，每次我们想要调用 `add_to_waitlist` 时，都必须指定 `front_of_house` 和 `hosting`。幸运的是，有一种方法可以简化这个过程。我们可以使用 `use` 关键字创建一个捷径，然后就可以在作用域中的任何地方使用这个更短的名字。

在示例 7-11 中，我们将 `crate::front_of_house::hosting` 模块引入了 `eat_at_restaurant` 函数的作用域，而我们只需要指定 `hosting::add_to_waitlist` 即可在 `eat_at_restaurant` 中调用 `add_to_waitlist` 函数。

文件名：src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

示例 7-11: 使用 `use` 将模块引入作用域

在作用域中增加 `use` 和路径类似于在文件系统中创建软连接（符号连接，symbolic link）。通过在 crate 根增加 `use crate::front_of_house::hosting`，现在 `hosting` 在作用域中就是有效的名称了，如同 `hosting` 模块被定义于 crate 根一样。通过 `use` 引入作用域的路径也会检查私有性，同其它路径一样。

注意 `use` 只能创建 `use` 所在的特定作用域内的捷径。示例 7-12 将 `eat_at_restaurant` 函数移动到了一个叫 `customer` 的子模块，这又是一个不同于 `use` 语句的作用域，所以函数体不能编译。

文件名：src/lib.rs



```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

示例 7-12: `use` 语句只适用于其所在的作用域

编译器错误显示捷径不再适用于 `customer` 模块中：

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`
   |
help: consider importing this module through its public re-export
   |
10 +     use crate::hosting;
   |

warning: unused import: `crate::front_of_house::hosting`
 --> src/lib.rs:7:5
  |
7 | use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` (lib) due to 1 previous error; 1 warning
emitted
```

注意这里还有一个警告说 `use` 在其作用域内不再被使用！为了修复这个问题，可以将 `use` 移动到 `customer` 模块内，或者在子模块 `customer` 内通过 `super::hosting` 引用父模块中的这个捷径。

创建惯用的 `use` 路径

在示例 7-11 中，你可能会比较疑惑，为什么我们是指定 `use crate::front_of_house::hosting`，然后在 `eat_at_restaurant` 中调用 `hosting::add_to_waitlist`，而不是通过指定一直到 `add_to_waitlist` 函数的 `use` 路径来得到相同的结果，如示例 7-13 所示。

文件名：src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

示例 7-13: 使用 `use` 将 `add_to_waitlist` 函数引入作用域，这并不符合习惯

虽然示例 7-11 和 7-13 都完成了相同的任务，但示例 7-11 是使用 `use` 将函数引入作用域的习惯用法。要想使用 `use` 将函数的父模块引入作用域，我们必须在调用函数时指定父模块，这样可以清晰地表明函数不是在本地定义的，同时使完整路径的重复度最小化。示例 7-13 中的代码不清楚 `add_to_waitlist` 是在哪里被定义的。

另一方面，使用 `use` 引入结构体、枚举和其他项时，习惯是指定它们的完整路径。示例 7-14 展示了将 `HashMap` 结构体引入二进制 crate 作用域的习惯用法。

文件名：src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

示例 7-14: 将 `HashMap` 引入作用域的习惯用法

这种习惯用法背后没有什么硬性要求：它只是一种惯例，人们已经习惯了以这种方式阅读和编写 Rust 代码。

这个习惯用法有一个例外，那就是我们想使用 `use` 语句将两个具有相同名称的项带入作用域，因为 Rust 不允许这样做。示例 7-15 展示了如何将两个具有相同名称但不同父模块的 `Result` 类型引入作用域，以及如何引用它们。

文件名：src/lib.rs

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

示例 7-15: 使用父模块将两个具有相同名称的类型引入同一作用域

如你所见，使用父模块可以区分这两个 `Result` 类型。如果我们是指定 `use std::fmt::Result` 和 `use std::io::Result`，我们将在同一作用域拥有了两个 `Result` 类型，当我们使用 `Result` 时，Rust 则不知道我们要用的是哪个。

使用 `as` 关键字提供新的名称

使用 `use` 将两个同名类型引入同一作用域这个问题还有另一个解决办法：在这个类型的路径后面，我们使用 `as` 指定一个新的本地名称或者**别名**。示例 7-16 展示了另一个编写示例 7-15 中代码的方法，通过 `as` 重命名其中一个 `Result` 类型。

文件名：src/lib.rs

```
```

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

示例 7-16: 使用 `as` 关键字重命名引入作用域的类型

在第二个 `use` 语句中，我们选择 `IoResult` 作为 `std::io::Result` 的新名称，它与从 `std::fmt` 引入作用域的 `Result` 并不冲突。示例 7-15 和示例 7-16 都是惯用写法，如何选择都取决于你！

使用 `pub use` 重导出名称

使用 `use` 关键字，将某个名称导入当前作用域后，该名称对此作用域之外还是私有的。若要让作用域之外的代码能够像在当前作用域中一样使用该名称，可以将 `pub` 与 `use` 组合使用。这种技术被称为**重导出**（*re-exporting*），因为在把某个项目导入当前作用域的同时，也将其暴露给其他作用域。

示例 7-17 将示例 7-11 根模块中的 `use` 改为 `pub use` 的代码。

文件名：src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

示例 7-17: 通过 `pub use` 使名称可从新作用域中被导入至任何代码

在这个修改之前，外部代码需要使用路径 `restaurant::front_of_house::hosting::add_to_waitlist()` 来调用 `add_to_waitlist` 函数，并且还需要将 `front_of_house` 模块标记为 `pub`。现在这个 `pub use` 从根模块重导出了 `hosting` 模块，外部代码现在可以使用路径 `restaurant::hosting::add_to_waitlist`。

当你代码的内部结构与调用你代码的程序员所想象的结构不同时，重导出会很有用。例如，在这个餐馆的比喻中，经营餐馆的人会想到“前台”和“后台”。但顾客在光顾一家餐馆时，可能不会以这些术语来考虑餐馆的各个部分。使用 `pub use`，我们可以使用一种结构编写代码，却将不同的结构形式暴露出来。这样做使我们的库井井有条，也使开发这个库的程序员和调用这个库的程序员都更加方便。在“使用 `pub use` 导出便捷的公有 API”部分让我们再看另一个 `pub use` 的例子来了解这如何影响 crate 的文档。

使用外部包

在第二章中我们编写了一个猜猜看游戏。那个项目使用了一个外部包 `rand` 来生成随机数。为了在项目中使用 `rand`，在 *Cargo.toml* 中加入了如下行：

文件名：Cargo.toml

```
rand = "0.8.5"
```

在 *Cargo.toml* 中加入 `rand` 依赖告诉了 Cargo 要从 crates.io 下载 `rand` 和其依赖，并使其可在项目代码中使用。

接着，为了将 `rand` 定义引入项目包的作用域，我们加入一行 `use` 起始的包名，它以 `rand` 包名开头并列出了需要引入作用域的项。回忆一下第二章的“生成一个随机数”部分，我们曾将 `Rng` trait 引入作用域并调用了 `rand::thread_rng` 函数：

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

crates.io 上有很多 Rust 社区成员发布的包，将其引入你自己的项目都需要一道相同的步骤：在 *Cargo.toml* 列出它们并通过 `use` 将其中定义的项引入项目包的作用域中。

注意 `std` 标准库对于你的包来说也是外部 crate。因为标准库随 Rust 语言一同分发，无需修改 *Cargo.toml* 来引入 `std`，不过需要通过 `use` 将标准库中定义的项引入项目包的作用域中来引用它们。例如，对于 `HashMap`，我们会使用以下语句：

```
use std::collections::HashMap;
```

这是一个以标准库 crate 名 `std` 开头的绝对路径。

使用嵌套路径来清理大量的 `use` 列表

当需要引入很多定义于相同包或相同模块的项时，为每一项单独列出一行会占用源码大量的垂直空间。例如猜猜看章节示例 2-4 中有两行 `use` 语句都从 `std` 引入项到作用域：

文件名：src/main.rs

```
// --snip--
use std::cmp::Ordering;
use std::io;
// --snip--
```

相反，我们可以使用嵌套路径将相同的项在一行中引入作用域。这么做需要指定路径的相同部分，接着是两个冒号，接着是大括号中的各自不同的路径部分，如示例 7-18 所示。

文件名：src/main.rs

```
```

```
// --snip--
use std::{cmp::Ordering, io};
// --snip--
```

示例 7-18: 指定嵌套的路径在一行中将多个带有相同前缀的项引入作用域

在较大的程序中，使用嵌套路径从相同包或模块中引入很多项，可以显著减少所需的独立 `use` 语句的数量！

我们可以在路径的任何层级使用嵌套路径，这在组合两个共享子路径的 `use` 语句时非常有用。例如，示例 7-19 中展示了两个 `use` 语句：一个将 `std::io` 引入作用域，另一个将 `std::io::Write` 引入作用域：

文件名：src/lib.rs

```
use std::io;
use std::io::Write;
```

示例 7-19: 通过两行 `use` 语句引入两个路径，其中一个是另一个的子路径

两个路径的相同部分是 `std::io`，这正是第一个路径。为了在一行 `use` 语句中引入这两个路径，可以在嵌套路径中使用 `self`，如示例 7-20 所示。

文件名：src/lib.rs

```
use std::io::{self, Write};
```

示例 7-20: 将示例 7-19 中部分重复的路径合并为一个 `use` 语句

这一行便将 `std::io` 和 `std::io::Write` 同时引入作用域。

glob 运算符

如果希望将一个路径下**所有**公有项引入作用域，可以指定路径后跟 `*` glob 运算符：

```
use std::collections::*;
```

这个 `use` 语句将 `std::collections` 中定义的所有公有项引入当前作用域。使用 glob 运算符时请多加小心！Glob 会使得我们难以推导作用域中有什么名称和它们是在何处定义的。

glob 运算符经常用于测试模块 `tests` 中，这时会将所有内容引入作用域；我们将在第十一章“如何编写测试”部分讲解。glob 运算符有时也用于 prelude 模式；查看标准库中的文档了解这个模式的更多细节。

将模块拆分成多个文件

到目前为止，本章所有的例子都在一个文件中定义多个模块。当模块变得更大时，你可能想要将它们的定义移动到单独的文件中，从而使代码更容易阅读。

例如，我们从示例 7-17 中包含多个餐厅模块的代码开始。我们会将模块提取到各自的文件中，而不是将所有模块都定义到 crate 根文件中。在这里，crate 根文件是 *src/lib.rs*，不过这个过程也适用于 crate 根文件是 *src/main.rs* 的二进制 crate。

首先将 `front_of_house` 模块提取到其自己的文件中。删除 `front_of_house` 模块的大括号中的代码，只留下 `mod front_of_house;` 声明，这样 *src/lib.rs* 会包含如示例 7-21 所示的代码。注意直到创建示例 7-22 中的 *src/front_of_house.rs* 文件之前代码都不能编译。

文件名：src/lib.rs



```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

示例 7-21: 声明 `front_of_house` 模块，其内容将位于 *src/front_of_house.rs*

接下来将之前大括号内的代码放入一个名叫 *src/front_of_house.rs* 的新文件中，如示例 7-22 所示。因为编译器找到了 crate 根中名叫 `front_of_house` 的模块声明，它就知道去搜寻这个文件。

文件名：src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

示例 7-22: 在 *src/front_of_house.rs* 中定义 `front_of_house` 模块

注意你只需在模块树中的某处使用一次 `mod` 声明就可以加载这个文件。一旦编译器知道了这个文件是项目的一部分（并且通过 `mod` 语句的位置知道了代码在模块树中的位置），项目中的其他文件应该使用其所声明的位置的路径来引用那个文件的代码，这在“引用模块项目的路径”部分有讲到。换句话说，`mod` **不是**你可能会在其他编程语言中看到的 “include” 操作。

接下来我们同样将 `hosting` 模块提取到自己的文件中。这个过程会有所不同，因为 `hosting` 是 `front_of_house` 的子模块而不是根模块。我们将 `hosting` 的文件放在与模块树中它的父模块同名的目录中，在这里是 *src/front_of_house/*。

为了移动 `hosting`，修改 *src/front_of_house.rs* 使之仅包含 `hosting` 模块的声明。

文件名：src/front_of_house.rs

```
pub mod hosting;
```

接着我们创建一个 *src/front_of_house* 目录和一个包含 `hosting` 模块定义的 *hosting.rs* 文件：

文件名：src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

如果将 *hosting.rs* 放在 *src* 目录，编译器会认为 `hosting` 模块中的 *hosting.rs* 的代码声明于 crate 根，而不是声明为 `front_of_house` 的子模块。编译器所遵循的哪些文件对应哪些模块的代码的规则，意味着目录和文件更紧密地贴合模块树。

> **另一种文件路径**
>
> 目前为止我们介绍了 Rust 编译器所最常用的文件路径，但 Rust 也支持一种更老的路径风格。
>
> 对于声明于 crate 根的 `front_of_house` 模块，编译器会在如下位置查找模块代码：
>
> - *src/front_of_house.rs*（我们所介绍的）
> - *src/front_of_house/mod.rs*（老风格，不过仍然支持）
>
> 对于 `front_of_house` 的子模块 `hosting`，编译器会在如下位置查找模块代码：
>
> - *src/front_of_house/hosting.rs*（我们所介绍的）
> - *src/front_of_house/hosting/mod.rs*（老风格，不过仍然支持）
>
> 如果你对同一模块同时使用这两种路径风格，会得到一个编译错误。在同一项目中的不同模块混用不同的路径风格是允许的，不过这会使他人感到疑惑。
>
> 使用 *mod.rs* 这一文件名的风格的主要缺点是会导致项目中出现很多 *mod.rs* 文件，当你在编辑器中同时打开它们时会感到疑惑。

我们将各个模块的代码移动到独立文件了，同时模块树保持不变。`eat_at_restaurant` 中的函数调用也无需修改继续保持有效，即便其定义存在于不同的文件中。这个技巧让你可以在模块代码增长时，将它们移动到新文件中。

注意，*src/lib.rs* 中的 `pub use crate::front_of_house::hosting` 语句也并未发生改变，`use` 也不会对哪些文件会被编译为 crate 的一部分有任何影响。`mod` 关键字声明了模块，而 Rust 会在与模块同名的文件中查找模块的代码。

总结

Rust 允许你将一个包拆分为多个 crate，并将一个 crate 拆分为若干模块，从而可以在一个模块中引用另一个模块中定义的项。你可以使用绝对路径或相对路径来实现这一点。你可以通过使用 `use` 语句将路径引入作用域，这样在多次使用时可以使用更短的路径。模块定义的代码默认是私有的，不过可以选择增加 `pub` 关键字使其定义变为公有。

接下来，让我们看看一些标准库提供的集合数据类型，你可以利用它们编写出漂亮整洁的代码。

常见集合

Rust 标准库中包含一系列被称为 **集合**（*collections*）的非常有用的数据结构。大部分其他数据类型都代表一个特定的值，不过集合可以包含多个值。不同于内建的数组和元组类型，这些集合指向的数据是储存在堆上的，这意味着数据的数量不必在编译时就已知，并且还可以随着程序的运行增长或缩小。每种集合都有着不同功能和开销，而根据当前情况选择合适的集合，这是一项需要逐渐掌握的技能。在这一章里，我们将详细的了解三个在 Rust 程序中被广泛使用的集合：

- **向量**（*vector*）允许我们一个挨着一个地储存一系列数量可变的值。
- **字符串**（*string*）是字符的集合。我们之前见过 `String` 类型，不过在本章我们将深入了解。
- **哈希 map**（*hash map*）允许我们将值与一个特定的键（key）相关联。这是一个叫做 *map* 的更通用的数据结构的特定实现。

对于标准库提供的其他类型的集合，请查看文档。

我们将讨论如何创建和更新 vector、字符串和哈希 map，以及它们有什么特别之处。

使用 Vector 储存列表

我们要讲到的第一个类型是 `Vec<T>`，也被称为 *vector*。vector 允许我们在一个单独的数据结构中储存多于一个的值，它在内存中彼此相邻地排列所有的值。vector 只能储存相同类型的值。它们在拥有一系列项的场景下非常实用，例如文件中的文本行或是购物车中商品的价格。

新建 vector

为了新建一个的空 vector，可以调用 `Vec::new` 函数，如示例 8-1 所示。

```
let v: Vec<i32> = Vec::new();
```

示例 8-1：新建一个的空 vector 来储存 `i32` 类型的值

注意这里我们增加了一个类型注解。因为没有向这个 vector 中插入任何值，Rust 并不知道我们想要储存什么类型的元素。这是一个非常重要的点。vector 是用泛型实现的，第十章会涉及到如何对你自己的类型使用它们。现在，所有你需要知道的就是 `Vec<T>` 是一个由标准库提供的类型，它可以存放任何类型，而当 `Vec` 存放某个特定类型时，那个类型位于尖括号中。在示例 8-1 中，我们告诉 Rust `v` 这个 `Vec<T>` 将存放 `i32` 类型的元素。

通常，我们会用初始值来创建一个 `Vec<T>` 而 Rust 会推断出储存值的类型，所以很少会需要这些类型注解。为了方便 Rust 提供了 `vec!` 宏，这个宏会根据我们提供的值来创建一个新的 vector。示例 8-2 新建一个拥有值 `1`、`2` 和 `3` 的 `Vec<i32>`。推断为 `i32` 是因为这是默认整型类型，第三章的 “数据类型” 讨论过：

```
let v = vec![1, 2, 3];
```

示例 8-2：新建一个包含初值的 vector

因为我们提供了 `i32` 类型的初始值，Rust 可以推断出 `v` 的类型是 `Vec<i32>`，因此类型注解就不是必须的。接下来让我们看看如何修改一个 vector。

更新 vector

对于新建一个 vector 并向其增加元素，可以使用 `push` 方法，如示例 8-3 所示：

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

示例 8-3：使用 `push` 方法向 vector 增加值

如第三章中讨论的任何变量一样，如果想要能够改变它的值，必须使用 `mut` 关键字使其可变。放入其中的所有值都是 `i32` 类型的，而且 Rust 也根据数据做出如此判断，所以不需要 `Vec<i32>` 注解。

读取 vector 的元素

有两种方法引用 vector 中储存的值：通过索引或使用 `get` 方法。在接下来的示例中，为了更加清楚的说明，我们已经标注了这些函数返回的值的类型。

示例 8-4 展示了访问 vector 中一个值的两种方式，索引语法或者 `get` 方法：

```
    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2];
    println!("The third element is {third}");

    let third: Option<&i32> = v.get(2);
    match third {
        Some(third) => println!("The third element is {third}"),
        None => println!("There is no third element."),
    }
```

列表 8-4：使用索引语法或 `get` 方法来访问 vector 中的项

这里有几个细节需要注意。我们使用索引值 `2` 来获取第三个元素，因为索引是从数字 0 开始的。使用 `&` 和 `[]` 会得到一个索引位置元素的引用。当使用索引作为参数调用 `get` 方法时，会得到一个可以用于 `match` 的 `Option<&T>`。

Rust 提供了两种引用元素的方法的原因是当尝试使用现有元素范围之外的索引值时可以选择让程序如何运行。举个例子，让我们看看使用这个技术，尝试在当有一个 5 个元素的 vector 接着访问索引 100 位置的元素会发生什么，如示例 8-5 所示：

```
    let v = vec![1, 2, 3, 4, 5];

    let does_not_exist = &v[100];
    let does_not_exist = v.get(100);
```



示例 8-5：尝试访问一个包含 5 个元素的 vector 的索引 100 处的元素

当运行这段代码，你会发现对于第一个 `[]` 方法，当引用一个不存在的元素时 Rust 会造成 panic。此方法适用于当你希望在尝试访问 vector 末尾之外的元素时让程序直接崩溃的场景。

当 `get` 方法被传递了一个数组外的索引时，它不会 panic 而是返回 `None`。当偶尔出现超过 vector 范围的访问属于正常情况的时候可以考虑使用它。接着你的代码可以有处理 `Some(&element)` 或 `None` 的逻辑，如第六章讨论的那样。例如，索引可能来源于用户输入的数字。如果它们不慎输入了一个过大的数字那么程序就会得到 `None` 值，你可以告诉用户当前 vector 元素的数量并再请求它们输入一个有效的值。这就比因为输入错误而使程序崩溃要友好的多！

一旦程序获取了一个有效的引用，借用检查器将会执行所有权和借用规则（第四章讲到）来确保 vector 内容的这个引用和任何其他引用保持有效。回忆一下不能在相同作用域中同时存在可变和不可变引用的规则。这个规则适用于示例 8-6，当我们获取了 vector 的第一个元素的不可变引用并尝试在 vector 末尾增加一个元素的时候，如果尝试在函数的后面再次引用这个元素是行不通的：

```
    let mut v = vec![1, 2, 3, 4, 5];
```

```
    let first = &v[0];

    v.push(6);

    println!("The first element is: {first}");
```



示例 8-6：尝试在拥有 vector 中项的引用的同时向其增加一个元素

编译会给出这个错误：

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("The first element is: {first}");
  |                                     ------- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` (bin "collections") due to 1 previous error
```

示例 8-6 中的代码看起来应该能够运行：为什么第一个元素的引用会关心 vector 结尾的变化？不能这么做的原因是由于 vector 的工作方式：在 vector 的结尾增加新元素时，在没有足够空间将所有元素依次相邻存放的情况下，可能会要求分配新内存并将老的元素拷贝到新的空间中。这时，第一个元素的引用就指向了被释放的内存。借用规则阻止程序陷入这种状况。

> 注意：关于 `Vec<T>` 类型的更多实现细节，请查看 “The Rustonomicon”

遍历 vector 中的元素

如果想要依次访问 vector 中的每一个元素，我们可以遍历其所有的元素而无需通过索引一次一个的访问。示例 8-7 展示了如何使用 `for` 循环来获取 `i32` 值的 vector 中的每一个元素的不可变引用并将其打印：

```
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{i}");
    }
```

示例 8-7：通过 `for` 循环遍历 vector 的元素并打印

我们也可以遍历可变 vector 的每一个元素的可变引用以便能改变它们。示例 8-8 中的 `for` 循环会给每一个元素加 `50`：

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

示例 8-8：遍历 vector 中元素的可变引用

为了修改可变引用所指向的值，在使用 `+=` 运算符之前必须使用解引用运算符（`*`）获取 `i` 中的值。第十五章的“通过解引用运算符追踪指针的值”部分会详细介绍解引用运算符。

由于借用检查器的规则，无论可变还是不可变地遍历一个 vector 都是安全的。如果尝试在示例 8-7 和 示例 8-8 的 `for` 循环体内插入或删除项，都会得到一个类似示例 8-6 代码中类似的编译错误。`for` 循环中获取的 vector 引用阻止了同时对整个 vector 进行修改。

使用枚举来储存多种类型

vector 只能储存相同类型的值。这是很不方便的；绝对会有需要储存一系列不同类型的值的用例。幸运的是，枚举的成员都被定义为相同的枚举类型，所以当需要在 vector 中储存不同类型值时，我们可以定义并使用一个枚举！

例如，假如我们想要从电子表格的一行中获取值，而这一行的有些列包含数字，有些包含浮点值，还有些是字符串。我们可以定义一个枚举，其成员会存放这些不同类型的值，同时所有这些枚举成员都会被当作相同类型：那个枚举的类型。接着可以创建一个储存该枚举值的 vector，这样最终就能够储存不同类型的值了。示例 8-9 展示了这个用法：

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

示例 8-9：定义一个枚举，以便能在 vector 中存放不同类型的数据

Rust 在编译时必须确切知道 vector 中的类型，这样它才能确定在堆上需要为每个元素分配多少内存。我们还必须明确这个 vector 中允许的类型。如果 Rust 允许 vector 存储任意类型，那么可能会因为一个或多个类型在对 vector 元素执行操作时导致（类型相关）错误。使用枚举加上 `match` 表达式意味着 Rust 会在编译时确保每种可能的情况都得到处理，正如第六章讲到的那样。

如果在编写程序时不能确切无遗地知道运行时会储存进 vector 的所有类型，枚举技术就行不通了。相反，你可以使用 trait 对象，第十八章会讲到它。

现在我们了解了一些使用 vector 的最常见的方式，请一定去看看标准库中 `Vec` 定义的很多其他实用方法的 API 文档。例如，除了 `push` 之外还有一个 `pop` 方法，它会移除并返回 vector 的最后一个元素。

丢弃 vector 时也会丢弃其所有元素

类似于任何其他的 `struct`，vector 在其离开作用域时会被释放，如示例 8-10 所标注的：

```
{
    let v = vec![1, 2, 3, 4];

    // 使用 v
} // <- 在这里 v 离开作用域并被释放
```

示例 8-10：展示 vector 和其元素于何处被丢弃

当 vector 被丢弃时，所有其内容也会被丢弃，这意味着这里它包含的整数将被清理。借用检查器确保了任何 vector 中内容的引用仅在 vector 本身有效时才可用。

让我们继续下一个集合类型：`String`！

使用字符串储存 UTF-8 编码的文本

第四章已经讲过一些字符串的内容，不过现在让我们更深入地了解它。字符串是新晋 Rustacean 们通常会被困住的领域，这是由于三方面理由的结合：Rust 倾向于确保暴露出可能的错误，字符串是比很多程序员所想象的要更为复杂的数据结构，以及 UTF-8。所有这些要素结合起来对于来自其他语言背景的程序员就可能显得很困难了。

在集合章节中讨论字符串的原因是，字符串就是作为字节的集合外加一些方法实现的，当这些字节被解释为文本时，这些方法提供了实用的功能。在本小节中，我们会讲到 `String` 中那些任何集合类型都有的操作，比如创建、更新和读取。也会讨论 `String` 与其他集合不一样的地方，例如索引 `String` 是很复杂的，由于人和计算机理解 `String` 数据方式的不同。

什么是字符串?

我们先定义一下**字符串**这一术语的具体意义。Rust 的核心语言中只有一种字符串类型，字符串 slice `str`，它通常以被借用的形式出现，`&str`。第四章讲到了**字符串 slices**：它们是一些对储存在别处的 UTF-8 编码字符串数据的引用。举例来说，由于字符串字面值被储存在程序的二进制输出中，因此它们也是字符串 slices。

字符串（`String`）类型由 Rust 标准库提供，而不是编入核心语言，它是一种可增长、可变、可拥有、UTF-8 编码的字符串类型。当 Rustaceans 提及 Rust 中的 “字符串 “时，他们可能指的是 `String` 或 string slice `&str` 类型，而不仅仅是其中一种类型。虽然本节主要讨论 `String`，但这两种类型在 Rust 的标准库中都有大量使用，而且 `String` 和 字符串 slices 都是 UTF-8 编码的。

新建字符串

很多 `Vec<T>` 上可用的操作在 `String` 中同样可用，事实上 `String` 被实现为一个带有一些额外保证、限制和功能的字节 vector 的封装。其中一个同样作用于 `Vec<T>` 和 `String` 函数的例子是用来新建一个实例的 `new` 函数，如示例 8-11 所示。

```
    let mut s = String::new();
```

示例 8-11：新建一个空的 `String`

这新建了一个叫做 `s` 的空的字符串，接着我们可以向其中加载数据。通常字符串会有初始数据，因为我们希望一开始就有这个字符串。为此，可以使用 `to_string` 方法，它能用于任何实现了 `Display` trait 的类型，比如字符串字面值。示例 8-12 展示了两个例子。

```
    let data = "initial contents";

    let s = data.to_string();

    // 该方法也可直接用于字符串字面值：
    let s = "initial contents".to_string();
```

示例 8-12：使用 `to_string` 方法从字符串字面值创建 `String`

这些代码会创建包含 `initial contents` 的字符串。

也可以使用 `String::from` 函数来从字符串字面值创建 `String`。示例 8-13 中的代码等同于使用 `to_string`。

```
    let s = String::from("initial contents");
```

示例 8-13：使用 `String::from` 函数从字符串字面值创建 `String`

因为字符串应用广泛，这里有很多不同的用于字符串的通用 API 可供选择。其中一些可能看起来多余，不过都有其用武之地！在这个例子中，`String::from` 和 `.to_string` 最终做了完全相同的工作，所以如何选择就是代码风格与可读性的问题了。

记住字符串是 UTF-8 编码的，所以可以包含任何经过正确编码的数据，如示例 8-14 所示。

```
    let hello = String::from("السلام عليكم");
    let hello = String::from("Dobrý den");
    let hello = String::from("Hello");
    let hello = String::from("שלום");
    let hello = String::from("नमस्ते");
    let hello = String::from("こんにちは");
    let hello = String::from("안녕하세요");
    let hello = String::from("你好");
    let hello = String::from("Olá");
    let hello = String::from("Здравствуйте");
    let hello = String::from("Hola");
```

示例 8-14：在字符串中储存不同语言的问候语

所有这些都是有效的 `String` 值。

更新字符串

`String` 的大小可以增加，其内容也可以改变，就像可以放入更多数据来改变 `Vec` 的内容一样。另外，可以方便的使用 `+` 运算符或 `format!` 宏来拼接 `String` 值。

使用 `push_str` 和 `push` 附加字符串

可以通过 `push_str` 方法来附加字符串 slice，从而使 `String` 变长，如示例 8-15 所示。

```
    let mut s = String::from("foo");
    s.push_str("bar");
```

示例 8-15：使用 `push_str` 方法向 `String` 附加字符串 slice

执行这两行代码之后，`s` 将会包含 `foobar`。`push_str` 方法采用字符串 slice，因为我们并不需要获取参数的所有权。例如，示例 8-16 中我们希望在将 `s2` 的内容附加到 `s1` 之后还能使用它。

```
    let mut s1 = String::from("foo");
    let s2 = "bar";
    s1.push_str(s2);
    println!("s2 is {s2}");
```

示例 8-16：将字符串 slice 的内容附加到 `String` 后使用它

如果 `push_str` 方法获取了 `s2` 的所有权，就不能在最后一行打印出其值了。好在代码如我们期望那样工作！

`push` 方法被定义为获取一个单独的字符作为参数，并附加到 `String` 中。示例 8-17 展示了使用 `push` 方法将字母 `l` 加入 `String` 的代码。

```
    let mut s = String::from("lo");
    s.push('l');
```

示例 8-17：使用 `push` 将一个字符加入 `String` 值中

执行这些代码之后，`s` 将会包含 `lol`。

使用 `+` 运算符或 `format!` 宏拼接字符串

通常你会希望将两个已知的字符串合并在一起。一种办法是像这样使用 `+` 运算符，如示例 8-18 所示。

```
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // 注意 s1 被移动了，不能继续使用
```

示例 8-18：使用 `+` 运算符将两个 `String` 值合并到一个新的 `String` 值中

执行完这些代码之后，字符串 `s3` 将会包含 `Hello, world!`。`s1` 在相加后不再有效的原因，和使用 `s2` 的引用的原因，与使用 `+` 运算符时调用的函数签名有关。`+` 运算符使用了 `add` 函数，这个函数签名看起来像这样：

```
fn add(self, s: &str) -> String {
```

在标准库中你会发现，`add` 的定义使用了泛型和关联类型。在这里我们替换为了具体类型，这也正是当使用 `String` 值调用这个方法会发生的。第十章会讨论泛型。这个签名提供了理解 `+` 运算那微妙部分的线索。

首先，`s2` 使用了 `&`，意味着我们使用第二个字符串的**引用**与第一个字符串相加。这是因为 `add` 函数的 `s` 参数：只能将 `&str` 和 `String` 相加，不能将两个 `String` 值相加。不过等一下 ——`&s2` 的类型是 `&String`，而不是 `add` 第二个参数所指定的 `&str`。那么为什么示例 8-18 还能编译呢？

之所以能够在 `add` 调用中使用 `&s2` 是因为 `&String` 可以被 **强转**（*coerced*）成 `&str`。当`add`函数被调用时，Rust 使用了一个被称为 **Deref 强制转换**（*deref coercion*）的技术，实际上会把 `&s2` 转换为 `&s2[..]`。第十五章会更深入的讨论 Deref 强制转换。因为 `add` 没有获取参数的所有权，所以在这个操作后 `s2` 仍然是有效的 `String`。

其次，可以发现签名中 `add` 获取了 `self` 的所有权，因为 `self` **没有**使用 `&`。这意味着示例 8-18 中的 `s1` 的所有权将被移动到 `add` 调用中，之后就不再有效。所以虽然 `let s3 = s1 + &s2;` 看起来就像它会复制两个字符串并创建一个新的字符串，而实际上这个语句会获取 `s1` 的所有权，附加上从 `s2` 中拷贝的内容，并返回结果的所有权。换句话说，它看起来好像生成了很多拷贝，不过实际上并没有：这个实现比拷贝要更高效。

如果想要级联多个字符串，`+` 运算符的行为就显得笨重了：

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = s1 + "-" + &s2 + "-" + &s3;
```

这时 `s` 的内容会是 `tic-tac-toe`。在有这么多 `+` 和 `"` 字符的情况下，很难理解具体发生了什么。对于更为复杂的字符串链接，可以使用 `format!` 宏：

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{s1}-{s2}-{s3}");
```

这些代码也会将 `s` 设置为 `tic-tac-toe`。`format!` 与 `println!` 的工作原理相同，不过不同于将输出打印到屏幕上，它返回一个带有结果内容的 `String`。这个版本就好理解的多，宏 `format!` 生成的代码使用引用因此不会获取任何参数的所有权。

索引字符串

在很多语言中，通过索引来引用字符串中的单独字符是有效且常见的操作。然而在 Rust 中，如果你尝试使用索引语法访问 `String` 的一部分，会出现一个错误。考虑一下如示例 8-19 中所示的无效代码。

```
    let s1 = String::from("hi");
    let h = s1[0];
```



示例 8-19：尝试对字符串使用索引语法

这段代码会导致如下错误：

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `str` cannot be indexed by `{integer}`
 --> src/main.rs:3:16
  |
3 |     let h = s1[0];
  |                ^ string indices are ranges of `usize`
  |
  = note: you can use `.chars().nth()` or `.bytes().nth()`
          for more information, see chapter 8 in The Book: <https://doc.rust-lang.
org/book/ch08-02-strings.html#indexing-into-strings>
  = help: the trait `SliceIndex<str>` is not implemented for `{integer}`
          but trait `SliceIndex<[_]>` is implemented for `usize`
  = help: for that trait implementation, expected `[_]`, found `str`
  = note: required for `String` to implement `Index<{integer}>`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` (bin "collections") due to 1 previous error
```

错误和提示说明了全部问题：Rust 的字符串不支持索引。那么，为什么会这样呢？为了回答这个问题，我们必须先聊一聊 Rust 是如何在内存中储存字符串的。

内部表现

`String` 是一个 `Vec<u8>` 的封装。让我们看看示例 8-14 中一些正确编码的字符串的例子。首先是这一例：

```
    let hello = String::from("Hola");
```

在这里，`len` 的值是 `4`，这意味着储存字符串 `"Hola"` 的 vector 的长度是四个字节：这里每一个字母的 UTF-8 编码都占用一个字节。下面这一行可能会让你感到意外（注意这个字符串中的首字母是西里尔字母的 *Ze* 而不是数字 3。）：

```
    let hello = String::from("Здравствуйте");
```

如果有人问及该字符串的长度，你可能会回答 12。然而，Rust 的回答是 24：这是使用 UTF-8 编码 “Здравствуйте” 所需要的字节数，这是因为在这个字符串中每个 Unicode 标量值需要两个字节存储。因此一个字符串字节值的索引并不总是对应一个有效的 Unicode 标量值。作为演示，考虑如下无效的 Rust 代码：

```
let hello = "Здравствуйте";
let answer = &hello[0];
```



我们已经知道 `answer` 不是第一个字符 `3`。当使用 UTF-8 编码时，`3` 的第一个字节是 `208`，第二个是 `151`，所以 `answer` 实际上应该是 `208`，不过 `208` 自身并不是一个有效的字母。返回 `208` 可不是一个请求字符串第一个字母的人所希望看到的，不过它是 Rust 在字节索引 0 位置所能提供的唯一数据。用户通常不会想要一个字节值被返回，即使这个字符串只有拉丁字母，如果 `&"hi"[0]` 是返回字节值的有效代码，它也会返回 `104` 而不是 `h`。

为了避免返回意外的值并造成不能立刻发现的 bug，Rust 根本不会编译这些代码，并在开发过程中及早杜绝了误会的发生。

字节、标量值和字形簇！天呐！

这引起了关于 UTF-8 的另外一个问题：从 Rust 的角度来讲，事实上有三种相关方式可以查看字符串：字节、标量值和字形簇（最接近人们眼中 **字母**（*letters*）的概念）。

比如这个用梵文书写的印度语单词 “नमस्ते”，最终它储存在 vector 中的 `u8` 值看起来像这样：

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

这里有 18 个字节，也就是计算机最终会储存的数据。如果从 Unicode 标量值的角度理解它们，也就像 Rust 的 `char` 类型那样，这些字节看起来像这样：

```
['न', 'म', 'स', '्', 'त', 'े']
```

这里有六个 `char`，不过第四个和第六个都不是字母，它们是发音符号本身并没有任何意义。最后，如果以字形簇的角度理解，就会得到人们所说的构成这个单词的四个字母：

```
["न", "म", "स्", "ते"]
```

Rust 提供了多种不同的方式来解释计算机储存的原始字符串数据，这样程序就可以选择它需要的表现方式，而无所谓是何种人类语言。

最后一个 Rust 不允许使用索引获取 `String` 字符的原因是，索引操作预期总是需要常数时间（O(1)）。但是对于 `String` 不可能保证这样的性能，因为 Rust 必须从开头到索引位置遍历来确定有多少有效的字符。

字符串 slice

索引字符串通常是一个坏点子，因为字符串索引应该返回的类型是不明确的：字节值、字符、字形簇或者字符串 slice。因此，如果你真的希望使用索引创建字符串 slice 时，Rust 会要求你更明确一些。为了更明确索引并表明你需要一个字符串 slice，相比使用 `[]` 和单个值的索引，可以使用 `[]` 和一个 range 来创建含特定字节的字符串 slice：

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

这里，`s` 会是一个 `&str`，它包含字符串的头四个字节。早些时候，我们提到了这些字母都是两个字节长的，所以这意味着 `s` 将会是 `Зд`。

如果尝试用类似 `&hello[0..1]` 的方式对字符的部分字节进行 slice，Rust 会在运行时 panic，就跟访问 vector 中的无效索引时一样：

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`

thread 'main' panicked at src/main.rs:4:19:
byte index 1 is not a char boundary; it is inside '3' (bytes 0..2) of
`Здравствуйте`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

在使用 range 来创建字符串 slice 时要格外小心，因为这么做可能会使你的程序崩溃。

遍历字符串的方法

操作字符串每一部分的最好的方法是明确表示需要字符还是字节。对于单独的 Unicode 标量值使用 `chars` 方法。对 “Зд” 调用 `chars` 方法会将其分开并返回两个 `char` 类型的值，接着就可以遍历其结果来访问每一个元素了：

```
for c in "Зд".chars() {
    println!("{c}");
}
```

这些代码会打印出如下内容：

```
3
д
```

另外 `bytes` 方法返回每一个原始字节，这可能会适合你的使用场景：

```
for b in "3д".bytes() {
    println!("{b}");
}
```

这些代码会打印出组成字符串的四个字节：

```
208
151
208
180
```

不过请务必记住有效的 Unicode 标量值可能会由不止一个字节组成。

从字符串中获取如同天城文这样的字形簇是很复杂的，所以标准库并没有提供这个功能。crates.io 上有些提供这样功能的 crate。

字符串并不简单

总而言之，字符串还是很复杂的。不同的语言选择了不同的向程序员展示其复杂性的方式。Rust 选择了以准确的方式处理 `String` 数据作为所有 Rust 程序的默认行为，这意味着程序员们必须更多的思考如何预先处理 UTF-8 数据。这种权衡相比其他语言更多地暴露出了字符串的复杂性，不过也使你在开发周期后期免于处理涉及非 ASCII 字符的错误。

好消息是标准库提供了很多围绕 `String` 和 `&str` 构建的功能，来帮助我们正确处理这些复杂场景。请务必查看这些使用方法的文档，例如 `contains` 来搜索一个字符串，和 `replace` 将字符串的一部分替换为另一个字符串。

现在让我们转向一些不太复杂的集合：哈希 map！

使用 Hash Map 储存键值对

最后介绍的常用集合类型是**哈希 map**（*hash map*）。`HashMap<K, V>` 类型储存了一个键类型 `K` 对应一个值类型 `V` 的映射。它通过一个**哈希函数**（*hashing function*）来实现映射，决定如何将键和值放入内存中。很多编程语言支持这种数据结构，不过通常有不同的名字：**哈希**、**map**、**对象**、**哈希表**、**字典**或者**关联数组**，仅举几例。

哈希 map 可以用于需要任何类型作为键来寻找数据的情况，而不是像 vector 那样通过索引。例如，在一个游戏中，你可以将每个团队的分数记录到哈希 map 中，其中键是队伍的名字而值是每个队伍的分数。给出一个队名，就能检索到该队的得分。

本章我们会介绍哈希 map 的基本 API，不过还有更多吸引人的功能隐藏于标准库在 `HashMap<K, V>` 上定义的函数中。一如既往请查看标准库文档来了解更多信息。

新建一个哈希 map

可以使用 `new` 创建一个空的 `HashMap`，并使用 `insert` 增加元素。在示例 8-20 中我们记录两支队伍的分数，分别是**蓝队**和**黄队**。蓝队开始有 10 分而黄队开始有 50 分：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

示例 8-20：新建一个哈希 map 并插入一些键值对

注意必须首先 `use` 标准库中集合部分的 `HashMap`。在这三个常用集合中，`HashMap` 是最不常用的，所以并没有被 prelude 自动引用。标准库中对 `HashMap` 的支持也相对较少，例如，并没有内建的构建宏。

像 vector 一样，哈希 map 将它们的数据储存在堆上，这个 `HashMap` 的键类型是 `String` 而值类型是 `i32`。类似于 vector，哈希 map 是同质的：所有的键必须是相同类型，值也必须都是相同类型。

访问哈希 map 中的值

可以通过 `get` 方法并提供对应的键来从哈希 map 中获取值，如示例 8-21 所示：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name).copied().unwrap_or(0);
```

示例 8-21：访问哈希 map 中储存的蓝队分数

这里，`score` 是与蓝队分数相关的值，应为 `10`。`get` 方法返回 `Option<&V>`，如果某个键在哈希 map 中没有对应的值，`get` 会返回 `None`。程序中通过调用 `copied` 方法来获取一个 `Option<i32>` 而不是 `Option<&i32>`，接着调用 `unwrap_or` 在 `scores` 中没有该键所对应的项时将其设置为零。

可以使用与 vector 类似的方式来遍历哈希 map 中的每一个键值对，也就是 `for` 循环：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{key}: {value}");
}
```

这会以任意顺序打印出每一个键值对：

```
Yellow: 50
Blue: 10
```

哈希 map 和所有权

对于像 `i32` 这样的实现了 `Copy` trait 的类型，其值可以拷贝进哈希 map。对于像 `String` 这样拥有所有权的值，其值将被移动而哈希 map 会成为这些值的所有者，如示例 8-22 所示：

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// 这里 field_name 和 field_value 不再有效，
// 尝试使用它们看看会出现什么编译错误！
```

示例 8-22：展示一旦键值对被插入后就为哈希 map 所拥有

当 `insert` 调用将 `field_name` 和 `field_value` 移动到哈希 map 中后，将不能使用这两个绑定。

如果将值的引用插入哈希 map，这些值本身将不会被移动进哈希 map。但是这些引用指向的值必须至少在哈希 map 有效时也是有效的。第十章 “生命周期确保引用有效” 部分将会更多的讨论这个问题。

更新哈希 map

尽管键值对的数量是可以增长的，每个唯一的键只能同时关联一个值（反之不一定成立：比如蓝队和黄队的 `scores` 哈希 map 中都可能存储有 10 这个值）。

当我们想要改变哈希 map 中的数据时，必须决定如何处理一个键已经有值了的情况。可以选择完全无视旧值并用新值代替旧值。可以选择保留旧值而忽略新值，并只在键**没有**对应值时增加新值。或者可以结合新旧两值。让我们看看这分别该如何实现！

覆盖一个值

如果我们插入了一个键值对，接着用相同的键插入一个不同的值，与这个键相关联的旧值将被替换。即便示例 8-23 中的代码调用了两次 `insert`，哈希 map 也只会包含一个键值对，因为两次都是对蓝队的键插入的值：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{scores:?}");
```

示例 8-23：替换以特定键储存的值

这会打印出 `{"Blue": 25}`。原始的值 `10` 则被覆盖了。

只在键尚不存在时插入键值对

我们经常会检查某个特定的键是否已经存在于哈希 map 中并进行如下操作：如果哈希 map 中键已经存在则不做任何操作；如果不存在则连同值一块插入。

为此哈希 map 有一个专用的 API，叫做 `entry`，它获取我们想要检查的键作为参数。`entry` 函数的返回值是一个枚举 `Entry` 它代表了可能存在也可能不存在的值。比如说我们想要检查黄队的键是否关联了一个值。如果没有，就插入值 `50`，对于蓝队也是如此。使用 `entry` API 的代码看起来如示例 8-24 所示。

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{scores:?}");
```

示例 8-24：使用 `entry` 方法只在键没有对应一个值时插入

`Entry` 的 `or_insert` 方法在键对应的值存在时就返回这个值的可变引用，如果不存在则将参数作为新值插入并返回新值的可变引用。这比编写自己的逻辑要简明的多，另外也与借用检查器结合得更好。

运行示例 8-24 的代码会打印出 `{"Yellow": 50, "Blue": 10}`。第一个 `entry` 调用会插入黄队的键和值 `50`，因为黄队并没有一个值。第二个 `entry` 调用不会改变哈希 map 因为蓝队已经有了值 `10`。

根据旧值更新一个值

另一个常见的哈希 map 的应用场景是找到一个键对应的值并根据旧的值更新它。例如，示例 8-25 中的代码计数一些文本中每一个单词分别出现了多少次。我们使用哈希 map 以单词作为键并递增其值来记录我们遇到过几次这个单词。如果是第一次看到某个单词，就先插入值 `0`。

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{map:?}");
```

示例 8-25：通过哈希 map 储存单词和计数来统计出现次数

这会打印出 `{"world": 2, "hello": 1, "wonderful": 1}`。你可能会看到相同的键值对以不同的顺序打印：回忆一下“访问哈希 map 中的值”部分中提到遍历哈希 map 会以任意顺序进行。

`split_whitespace` 方法返回一个由空格分隔 `text` 值子 slice 的迭代器。`or_insert` 方法返回这个键的值的一个可变引用（`&mut V`）。这里我们将这个可变引用储存在 `count` 变量中，所以为了赋值必须首先使用星号（`*`）解引用 `count`。这个可变引用在 `for` 循环的结尾离开作用域，这样所有这些改变都是安全的并符合借用规则。

哈希函数

`HashMap` 默认使用一种叫做 SipHash 的哈希函数，它可以抵御涉及哈希表（hash table） 的拒绝服务（Denial of Service, DoS）攻击。然而这并不是可用的最快的算法，不过为了更高的安全性值得付出一些性能的代价。如果性能监测显示此哈希函数非常慢，以致于你无法接受，你可以指定一个不同的 *hasher* 来切换为其它函数。hasher 是一个实现了 `BuildHasher` trait 的类型。第十章会讨论 trait 和如何实现它们。你并不需要从头开始实现你自己的 hasher；crates.io 有其他人分享的实现了许多常用哈希算法的 hasher 的库。

https://en.wikipedia.org/wiki/SipHash

总结

vector、字符串和哈希 map 会在你的程序需要储存、访问和修改数据时帮助你。这里有一些你应该能够解决的练习问题：

1. 给定一组整数，使用 vector 并返回这个列表的中位数（排列数组后位于中间的值）和众数（出现次数最多的值；在这里哈希 map 会很有帮助）。
2. 将字符串转换为 pig latin。也就是每一个单词的第一个辅音字母被移动到单词的结尾并增加 *ay*，所以 *first* 会变成 *irst-fay*。元音字母开头的单词则在结尾增加 *hay*（*apple* 会变成 *apple-hay*）。请注意 UTF-8 编码的细节！

3. 使用哈希 map 和 vector，创建一个文本接口来允许用户向公司的部门中增加员工的名字。例如，“Add Sally to Engineering” 或 “Add Amir to Sales”。接着让用户获取一个部门的所有员工的列表，或者公司每个部门的所有员工按照字典序排列的列表。

标准库 API 文档中描述的这些类型的方法将有助于你进行这些练习！

我们已经开始接触可能会有失败操作的复杂程序了，这也意味着接下来是一个了解错误处理的绝佳时机！接下来我们将讨论这一部分！

错误处理

错误是软件开发中不可避免的事实，所以 Rust 有一些处理出错情况的特性。在许多情况下，Rust 要求你承认错误的可能性，并在你的代码编译前采取一些行动。这一要求使你的程序更加健壮，因为它可以确保你在将代码部署到生产环境之前就能发现错误并进行适当的处理。

Rust 将错误分为两大类：**可恢复的**（*recoverable*）和 **不可恢复的**（*unrecoverable*）错误。对于一个可恢复的错误，比如文件未找到的错误，我们很可能只想向用户报告问题并重试操作。不可恢复的错误总是 bug 出现的征兆，比如试图访问一个超过数组末端的位置，因此我们要立即停止程序。

大多数语言并不区分这两种错误，并采用类似异常（exception）这样方式统一处理它们。Rust 没有异常。相反，它有 `Result<T, E>` 类型，用于处理可恢复的错误，还有 `panic!` 宏，在程序遇到不可恢复的错误时停止执行。本章首先介绍 `panic!` 调用，接着会讲到如何返回 `Result<T, E>`。此外，我们将探讨在决定是尝试从错误中恢复还是停止执行时的注意事项。

用 `panic!` 处理不可恢复的错误

突然有一天，代码出问题了，而你对此束手无策。对于这种情况，Rust 有 `panic!` 宏。在实践中有两种方法造成 panic：执行会造成代码 panic 的操作（比如访问超过数组结尾的内容）或者显式调用 `panic!` 宏。这两种情况都会使程序 panic。通常情况下这些 panic 会打印出一个错误信息，展开并清理栈数据，然后退出。通过一个环境变量，你也可以让 Rust 在 panic 发生时打印调用堆栈（call stack）以便于定位 panic 的原因。

> **响应 panic 时的栈展开或终止**
>
> 当出现 panic 时，程序默认会开始 **展开**（*unwinding*），这意味着 Rust 会回溯栈并清理它遇到的每一个函数的数据，不过这个回溯并清理的过程有很多工作。另一种选择是直接 **终止**（*abort*），这会不清理数据就退出程序。
>
> 那么程序所使用的内存需要由操作系统来清理。如果你需要项目的最终二进制文件越小越好，panic 时通过在 *Cargo.toml* 的 `[profile]` 部分增加 `panic = 'abort'`，可以由展开切换为终止。例如，如果你想要在 release 模式中 panic 时直接终止，可添加：
>
> ```
> [profile.release]
> panic = 'abort'
> ```

让我们在一个简单的程序中调用 `panic!`：

文件名：src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```



运行程序将会出现类似这样的输出：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`

thread 'main' panicked at src/main.rs:2:5:
crash and burn
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

最后两行包含 `panic!` 调用造成的错误信息。第一行显示了 panic 提供的信息并指明了源码中 panic 出现的位置：*src/main.rs:2:5* 表明这是 *src/main.rs* 文件的第二行第五个字符。

在这个例子中，被指明的那一行是我们代码的一部分，如果跳转到该行，就会发现 `panic!` 宏的调用。在其它情况下，`panic!` 可能会出现在我们的代码所调用的代码中。错误信息报告的文件名和行号可能指向别人代码中的 `panic!` 宏调用，而不是我们代码中最终导致 `panic!` 的那一行。

使用 `panic!` 的 backtrace

我们可以使用 `panic!` 被调用的函数的 backtrace 来寻找代码中出问题的地方。下面我们会详细介绍 backtrace 是什么。为了了解如何使用 `panic!` 的 backtrace，让我们来看另一个示例，我们代码中的 bug 引起的别的库中 `panic!` 的例子，而不是直接的宏调用看起来如何。示例 9-1 有一些尝试通过索引访问 vector 中超出有效范围元素的例子：

文件名：src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```



示例 9-1：尝试访问超越 vector 结尾的元素，这会造成 `panic!`

这里尝试访问 vector 的第 100 个元素（这里的索引是 99 因为索引从 0 开始），不过它只有三个元素。这种情况下 Rust 会 panic。`[]` 应当返回一个元素，不过如果传递了一个无效索引，就没有可供 Rust 返回的正确元素。

C 语言中，尝试读取数据结构之后的值是未定义行为（undefined behavior）。你会得到任何对应数据结构中这个元素的内存位置的值，甚至是这些内存并不属于这个数据结构的情况。这被称为 **缓存区过读**（*buffer overread*），并可能会导致安全漏洞，比如攻击者可以像这样操作索引来读取储存在数据结构之后未经授权的数据。

为了保护程序不受此类漏洞的影响，如果尝试读取一个索引不存在的元素，Rust 会停止执行并拒绝继续。让我们来试一试，看看结果：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`

thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

错误指向 *main.rs* 的第 4 行，这里我们试图访问向量 `v` 中的索引 `99`。

`note:` 告诉我们可以设置 `RUST_BACKTRACE` 环境变量来得到一个 backtrace。*backtrace* 是一个执行到目前位置所有被调用的函数的列表。Rust 的 backtrace 跟其他语言中的一样：阅读 backtrace 的关键是从头开始读直到发现你编写的文件。这就是问题的发源地。这一行往上是你的代码所调用的代码；往下则是调用你的代码的代码。这些行可能包含核心 Rust 代码，标准库代码或用到的 crate 代码。让我们将 `RUST_BACKTRACE` 环境变量设置为任何不是 `0` 的值来获取 backtrace 看看。示例 9-2 展示了与你看到类似的输出：

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
stack backtrace:
   0: rust_begin_unwind
```

```
             at /rustc/4d91de4e48198da2e33413efdcd9cd2cc0c46688/library/std/src/
panicking.rs:692:5
   1: core::panicking::panic_fmt
             at /rustc/4d91de4e48198da2e33413efdcd9cd2cc0c46688/library/core/src/
panicking.rs:75:14
   2: core::panicking::panic_bounds_check
             at /rustc/4d91de4e48198da2e33413efdcd9cd2cc0c46688/library/core/src/
panicking.rs:273:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at file:///home/.rustup/toolchains/1.85/lib/rustlib/src/rust/library/
core/src/slice/index.rs:274:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at file:///home/.rustup/toolchains/1.85/lib/rustlib/src/rust/library/
core/src/slice/index.rs:16:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at file:///home/.rustup/toolchains/1.85/lib/rustlib/src/rust/library/
alloc/src/vec/mod.rs:3361:9
   6: panic::main
             at ./src/main.rs:4:6
   7: core::ops::function::FnOnce::call_once
             at file:///home/.rustup/toolchains/1.85/lib/rustlib/src/rust/library/
core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

示例 9-2：当设置 `RUST_BACKTRACE` 环境变量时 `panic!` 调用所生成的 backtrace 信息

这里有大量的输出！你实际看到的输出可能因不同的操作系统和 Rust 版本而有所不同。为了获取带有这些信息的 backtrace，必须启用调试符号（debug symbols）。当不使用 `--release` 参数运行 cargo build 或 cargo run 时调试符号会默认启用，就像这里一样。

示例 9-2 的输出中，backtrace 的第 6 行指向了我们项目中造成问题的行：*src/main.rs* 的第 4 行。如果你不希望程序 panic，就应当从第一个提到我们自己编写的文件的那一行开始调查。在示例 9-1 中，我们故意编写了会导致 panic 的代码，修复这个 panic 的方法就是不要尝试在一个只包含三个项的 vector 中请求索引是 100 的元素。当将来你的代码出现了 panic，你需要搞清楚在这特定的场景下代码中执行了什么操作和什么值导致了 panic，以及应当如何处理才能避免该问题。

本章后面的小节 “要不要 panic!” 会再次回到 `panic!` 并讲解何时应该、何时不应该使用 `panic!` 来处理错误情况。接下来，我们来看看如何使用 `Result` 来从错误中恢复。

用 `Result` 处理可恢复的错误

大部分错误并没有严重到需要程序完全停止执行。有时函数失败的原因很容易理解并加以处理。例如，如果因为打开一个并不存在的文件而失败，此时我们可能想要创建这个文件，而不是终止进程。

回忆一下第二章 “使用 `Result` 类型来处理潜在的错误” 部分中的那个 `Result` 枚举，它定义有如下两个变体，`Ok` 和 `Err`：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`T` 和 `E` 是泛型类型参数；第十章会详细介绍泛型。现在你需要知道的就是 `T` 代表成功时返回的 `Ok` 变体中的数据的类型，而 `E` 代表失败时返回的 `Err` 变体中的错误的类型。因为 `Result` 有这些泛型类型参数，我们可以将 `Result` 类型和标准库中为其定义的函数用于很多不同的场景，这些情况中需要返回的成功值和失败值可能会各不相同。

让我们调用一个返回 `Result` 的函数，因为它可能会失败：如示例 9-3 所示尝试打开一个文件：

文件名：src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

示例 9-3：打开文件

`File::open` 的返回值是 `Result<T, E>`。泛型参数 `T` 会被 `File::open` 的实现放入成功返回值的类型 `std::fs::File`，这是一个文件句柄。错误返回值使用的 `E` 的类型是 `std::io::Error`。这些返回类型意味着 `File::open` 调用可能成功并返回一个可以读写的文件句柄。这个函数调用也可能会失败：例如，也许文件不存在，或者可能没有权限访问这个文件。`File::open` 函数需要一个方法在告诉我们成功与否的同时返回文件句柄或者错误信息。这些信息正好是 `Result` 枚举所代表的。

当 `File::open` 成功时，`greeting_file_result` 变量将会是一个包含文件句柄的 `Ok` 实例。当失败时，`greeting_file_result` 变量将会是一个包含了更多关于发生了何种错误的信息的 `Err` 实例。

我们需要在示例 9-3 的代码中增加根据 `File::open` 返回值进行不同处理的逻辑。示例 9-4 展示了一个使用基本工具处理 `Result` 的例子，第六章学习过的 `match` 表达式。

文件名：src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
```

```
    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {error:?}"),
    };
}
```

示例 9-4：使用 `match` 表达式处理可能会返回的 `Result` 变体

注意与 `Option` 枚举一样，`Result` 枚举和其变体也被导入到了 prelude 中，所以就不需要在 `match` 分支中的 `Ok` 和 `Err` 之前指定 `Result::`。

这里我们告诉 Rust 当结果是 `Ok` 时，返回 `Ok` 变体中的 `file` 值，然后将这个文件句柄赋值给变量 `greeting_file`。`match` 之后，我们可以利用这个文件句柄来进行读写。

`match` 的另一个分支处理从 `File::open` 得到 `Err` 值的情况。在这种情况下，我们选择调用 `panic!` 宏。如果当前目录没有一个叫做 *hello.txt* 的文件，当运行这段代码时会看到如下来自 `panic!` 宏的输出：

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`

thread 'main' panicked at src/main.rs:8:23:
Problem opening the file: Os { code: 2, kind: NotFound, message: "No such file or
directory" }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

一如既往，此输出准确地告诉了我们到底出了什么错。

匹配不同的错误

示例 9-4 中的代码不管 `File::open` 是因为什么原因失败都会 `panic!`。我们真正希望的是对不同的错误原因采取不同的行为：如果 `File::open` 因为文件不存在而失败，我们希望创建这个文件并返回新文件的句柄。如果 `File::open` 因为任何其他原因失败 – 例如没有打开文件的权限 – 我们仍然希望像示例 9-4 那样 `panic!`。为此，我们在示例 9-5 中添加了一个内部 `match` 表达式，如下所示：

文件名：src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {e:?}"),
            },
```

```
            _ => {
                panic!("Problem opening the file: {error:?}");
            }
        },
    };
}
```

示例 9-5：使用不同的方式处理不同类型的错误

`File::open` 返回的 `Err` 变体中的值类型 `io::Error`，它是一个标准库中提供的结构体。这个结构体有一个返回 `io::ErrorKind` 值的 `kind` 方法可供调用。`io::ErrorKind` 是一个标准库提供的枚举，它的变体对应 `io` 操作可能导致的不同错误类型。我们感兴趣的变体是 `ErrorKind::NotFound`，它代表尝试打开的文件并不存在。这样，`match` 就匹配完 `greeting_file_result` 了，不过对于 `error.kind()` 还有一个内层 `match`。

我们希望在内层 `match` 中检查的条件是 `error.kind()` 的返回值是否为 `ErrorKind`的 `NotFound` 变体。如果是，则通过 `File::create` 尝试创建该文件。然而因为 `File::create` 也可能会失败，还需要在内层 `match` 表达式中增加了第二个分支。当文件不能被创建，会打印出一个不同的错误信息。外层 `match` 的最后一个分支保持不变，这样对任何除了文件不存在的错误会使程序 panic。

使用 `match` 处理 `Result<T, E>` 的替代方案

这里有好多 `match`！`match` 确实很强大，不过也非常的原始。第十三章我们会介绍闭包（closure），它会和定义在 `Result<T, E>` 中的很多方法一起使用。在处理代码中的 `Result<T, E>` 值时，使用这些方法往往比直接写 `match` 更简洁。

例如，这是另一个编写与示例 9-5 逻辑相同但是使用闭包和 `unwrap_or_else` 方法的例子：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {error:?}");
            })
        } else {
            panic!("Problem opening the file: {error:?}");
        }
    });
}
```

虽然这段代码有着如示例 9-5 一样的行为，但并没有包含任何 `match` 表达式且更容易阅读。在阅读完第十三章后再回到这个例子，并查看标准库文档 `unwrap_or_else` 方法都做了什么操作。在处理错误时，还有很多这类方法可以消除大量嵌套的 `match` 表达式。

失败时 panic 的快捷方式：`unwrap` 和 `expect`

`match` 能够胜任它的工作，不过它可能有点冗长并且不总是能很好的表明其意图。`Result<T, E>` 类型定义了很多辅助方法来处理各种更为特定的任务。`unwrap` 方法是一个快捷方式，其内部实现与我们在 Listing 9-4 中编写的 `match` 表达式相同。如果 `Result` 值是变体 `Ok`，`unwrap` 会返回 `Ok` 中的值。如果 `Result` 是变体 `Err`，`unwrap` 会为我们调用 `panic!`。这里是一个实践 `unwrap` 的例子：

文件名：src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

如果调用这段代码时不存在 *hello.txt* 文件，我们将会看到一个 `unwrap` 调用 `panic!` 时提供的错误信息：

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:4:49
```

同样，`expect` 方法也允许我们自定义 `panic!` 的错误信息。使用 `expect` 而不是 `unwrap` 并提供一个好的错误信息可以表明你的意图并更易于追踪 panic 的根源。`expect` 的语法看起来像这样：

文件名：src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

`expect` 与 `unwrap` 的使用方式一样：返回文件句柄或调用 `panic!` 宏。`expect` 在调用 `panic!` 时使用的错误信息将是我们传递给 `expect` 的参数，而不像 `unwrap` 那样使用默认的 `panic!` 信息。它看起来像这样：

```
thread 'main' panicked at src/main.rs:5:10:
hello.txt should be included in this project: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
```

在生产级别的代码中，大部分 Rustaceans 选择 `expect` 而不是 `unwrap` 并提供更多关于为何操作期望是一直成功的上下文。如此如果该假设真的被证明是错的，你也有更多的信息来用于调试。

传播错误

当函数的实现中调用了可能会失败的操作时，除了在这个函数中处理错误外，还可以选择让调用者知道这个错误并决定该如何处理。这被称为**传播**（*propagating*）错误，这样能更好的控制代码调用，因为比起你代码所拥有的上下文，调用者可能拥有更多信息或逻辑来决定应该如何处理错误。

例如，示例 9-6 展示了一个从文件中读取用户名的函数。如果文件不存在或不能读取，这个函数会将这些错误返回给调用它的代码：

文件名：src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

示例 9-6：一个函数使用 `match` 将错误返回给代码调用者

这个函数可以编写成更加简短的形式，不过我们以大量手动处理开始以便探索错误处理；在最后我们会展示更简洁的形式。让我们看看函数的返回值：`Result<String, io::Error>`。这意味着函数返回一个 `Result<T, E>` 类型的值，其中泛型参数 `T` 的具体类型是 `String`，而 `E` 的具体类型是 `io::Error`。

如果这个函数没有出任何错误成功返回，函数的调用者会收到一个包含 `String` 的 `Ok` 值 —— 函数从文件中读取到的用户名。如果函数遇到任何错误，函数的调用者会收到一个 `Err` 值，它储存了一个包含更多这个问题相关信息的 `io::Error` 实例。这里选择 `io::Error` 作为函数的返回值是因为它正好是函数体中那两个可能会失败的操作的错误返回值：`File::open` 函数和 `read_to_string` 方法。

函数体以调用 `File::open` 函数开始。接着使用 `match` 处理返回值 `Result`，类似示例 9-4，如果 `File::open` 成功了，模式变量 `file` 中的文件句柄就变成了可变变量 `username_file` 中的值，接着函数继续执行。在 `Err` 的情况下，我们没有调用 `panic!`，而是使用 `return` 关键字提前结束整个函数，并将来自 `File::open` 的错误值（现在在模式变量 `e` 中）作为函数的错误值传回给调用者。

所以，如果在 `username_file` 中有一个文件句柄，该函数随后会在变量 `username` 中创建一个新的 `String` 并调用文件句柄 `username_file` 上的 `read_to_string` 方法，以将文件的内容读入 `username`。`read_to_string` 方法也返回一个 `Result`，因为它可能会失败，哪怕是 `File::open`

已经成功了。因此，我们需要另一个 `match` 来处理这个 `Result`：如果 `read_to_string` 执行成功，那么这个函数也就成功了，我们将从文件中读取的用户名返回，此时用户名位于被封装进 `Ok` 的 `username` 中。如果 `read_to_string` 执行失败，则像之前处理 `File::open` 的返回值的 `match` 那样返回错误值。然而，我们无需显式写出 `return`，因为这是函数的最后一个表达式。

调用这个函数的代码最终会得到一个包含用户名的 `Ok` 值，或者一个包含 `io::Error` 的 `Err` 值。我们无从得知调用者会如何处理这些值。例如，如果他们得到了一个 `Err` 值，他们可能会选择 `panic!` 并使程序崩溃、使用一个默认的用户名或者从文件之外的地方寻找用户名。我们没有足够的信息知晓调用者具体会如何尝试，所以将所有的成功或失败信息向上传播，让他们选择合适的处理方法。

这种传播错误的模式在 Rust 是如此的常见，以至于 Rust 提供了 `?` 问号运算符来来简化这一过程。

传播错误的快捷方式：`?` 运算符

示例 9-7 展示了一个 `read_username_from_file` 的实现，它实现了与示例 9-6 中的代码相同的功能，不过这个实现使用了 `?` 运算符：

文件名：src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

示例 9-7：一个使用 `?` 运算符向调用者返回错误的函数

`Result` 值之后的 `?` 被定义为与示例 9-6 中定义的处理 `Result` 值的 `match` 表达式有着几乎完全相同的工作方式。如果 `Result` 的值是 `Ok`，这个表达式将会返回 `Ok` 中的值而程序将继续执行。如果值是 `Err`，`Err` 将作为整个函数的返回值，就好像使用了 `return` 关键字一样，这样错误值就被传播给了调用者。

示例 9-6 中的 `match` 表达式与 `?` 运算符所做的有一点不同：`?` 运算符所使用的错误值被传递给了 `from` 函数，它定义于标准库的 `From` trait 中，其用来将错误从一种类型转换为另一种类型。当 `?` 运算符调用 `from` 函数时，收到的错误类型被转换为由当前函数返回类型所指定的错误类型。这在当函数返回单个错误类型来代表所有可能失败的方式时很有用，即使其可能会因很多种原因失败。

例如，我们可以将示例 9-7 中的 `read_username_from_file` 函数修改为返回一个自定义的 `OurError` 错误类型。如果我们也定义了 `impl From<io::Error> for OurError` 来从 `io::Error` 构造一个 `OurError` 实例，那么 `read_username_from_file` 函数体中的 `?` 运算符调用会调用 `from` 并转换错误而无需在函数中增加任何额外的代码。

在示例 9-7 的上下文中，`File::open` 调用结尾的 `?` 会将 `Ok` 中的值返回给变量 `username_file`。如果发生了错误，`?` 运算符会使整个函数提前返回并将任何 `Err` 值返回给调用代码。同理也适用于 `read_to_string` 调用结尾的 `?`。

`?` 运算符消除了大量样板代码并使得函数的实现更简单。我们甚至可以在 `?` 之后直接使用链式方法调用来进一步简化代码，如示例 9-8 所示：

文件名：src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

示例 9-8：问号运算符之后的链式方法调用

在 `username` 中创建新的 `String` 被放到了函数开头；这一部分没有变化。我们对 `File::open("hello.txt")?` 的结果直接链式调用了 `read_to_string`，而不再创建变量 `username_file`。仍然需要 `read_to_string` 调用结尾的 `?`，而且当 `File::open` 和 `read_to_string` 都成功没有失败时返回包含用户名 `username` 的 `Ok` 值，而不是返回错误。其功能再一次与示例 9-6 和示例 9-7 保持一致，不过这是一个与众不同且更符合工程学（ergonomic）的写法。

示例 9-9 展示了一个使用 `fs::read_to_string` 的更为简短的写法：

文件名：src/main.rs

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

示例 9-9: 使用 `fs::read_to_string` 而不是打开后读取文件

将文件读取到一个字符串是相当常见的操作，所以标准库提供了名为 `fs::read_to_string` 的函数，它会打开文件、新建一个 `String`、读取文件的内容，并将内容放入 `String`，接着返回它。当然，这样做就没有展示所有这些错误处理的机会了，所以我们最初就选择了艰苦的道路。

哪里可以使用 `?` 运算符

`?` 运算符只能被用于返回值与 `?` 作用的值相兼容的函数。因为 `?` 运算符被定义为从函数中提早返回一个值，这与示例 9-6 中的 `match` 表达式有着完全相同的工作方式。示例 9-6 中 `match` 作用于一个 `Result` 值，提早返回的分支返回了一个 `Err(e)` 值。函数的返回值必须是 `Result` 才能与这个 `return` 相兼容。

在示例 9-10 中，让我们看看在返回值不兼容的 `main` 函数中使用 `?` 运算符会得到什么错误：

文件名：src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```



示例 9-10: 尝试在返回 `()` 的 `main` 函数中使用 `?` 的代码不能编译

这段代码打开一个文件，这可能会失败。`?` 运算符作用于 `File::open` 返回的 `Result` 值，不过 `main` 函数的返回类型是 `()` 而不是 `Result`。当编译这些代码，会得到如下错误信息：

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `FromResidual`)
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
  |                                                ^ cannot use the `?` operator
in a function that returns `()`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`
help: consider adding return type
  |
3 ~ fn main() -> Result<(), Box<dyn std::error::Error>> {
4 |     let greeting_file = File::open("hello.txt")?;
5 +     Ok(())
  |

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` (bin "error-handling") due to 1 previous
error
```

这个错误指出只能在返回 `Result`、`Option` 或者其它实现了 `FromResidual` 的类型的函数中使用 `?` 运算符。

为了修复这个错误，有两个选择。一个是，如果没有限制的话将函数的返回值改为与你在 `?` 运算符所作用的值兼容的类型。另一个是使用 `match` 或者 `Result<T, E>` 类型的方法，以适当的方式处理 `Result<T, E>`。

错误信息也提到 `?` 也可用于 `Option<T>` 值。如同对 `Result` 使用 `?` 一样，只能在返回 `Option` 的函数中对 `Option` 使用 `?`。在 `Option<T>` 上调用 `?` 运算符的行为与 `Result<T, E>` 类似：如果值是 `None`，此时 `None` 会从函数中提前返回。如果值是 `Some`，`Some` 中的值作为表达式的返回值同时函数继续。示例 9-11 中有一个从给定文本中返回第一行最后一个字符的函数的例子：

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

示例 9-11: 在 `Option<T>` 值上使用 `?` 运算符

这个函数返回 `Option<char>` 因为它可能会在这个位置找到一个字符，也可能没有字符。这段代码获取 `text` 字符串 slice 作为参数并调用其 `lines` 方法，这会返回一个字符串中每一行的迭代器。因为函数希望检查第一行，所以调用了迭代器 `next` 来获取迭代器中第一个值。如果 `text` 是空字符串，`next` 调用会返回 `None`，此时我们可以使用 `?` 来停止并从 `last_char_of_first_line` 返回 `None`。如果 `text` 不是空字符串，`next` 会返回一个包含 `text` 中第一行的字符串 slice 的 `Some` 值。

`?` 会提取这个字符串 slice，然后可以在字符串 slice 上调用 `chars` 来获取字符的迭代器。我们感兴趣的是第一行的最后一个字符，所以可以调用 `last` 来返回迭代器的最后一项。这是一个 `Option`，因为有可能第一行是一个空字符串；例如 `text` 以一个空行开头而后面的行有文本，像是 `"\nhi"`。不过，如果第一行有最后一个字符，它会返回在一个 `Some` 变体中。`?` 运算符作用于其中给了我们一个简洁的表达这种逻辑的方式。如果我们不能在 `Option` 上使用 `?` 运算符，则不得不使用更多的方法调用或者 `match` 表达式来实现这些逻辑。

注意你可以在返回 `Result` 的函数中对 `Result` 使用 `?` 运算符，可以在返回 `Option` 的函数中对 `Option` 使用 `?` 运算符，但是不可以混合搭配。`?` 运算符不会自动将 `Result` 转化为 `Option`，反之亦然；在这些情况下，可以使用类似 `Result` 的 `ok` 方法或者 `Option` 的 `ok_or` 方法来显式转换。

目前为止，我们所使用的所有 `main` 函数都返回 `()`。`main` 函数是特殊的因为它是可执行程序的入口点和退出点，为了使程序能正常工作，其可以返回的类型是有限制的。

幸运的是 `main` 函数也可以返回 `Result<(), E>`，示例 9-12 中的代码来自示例 9-10 不过修改了 `main` 的返回值为 `Result<(), Box<dyn Error>>` 并在结尾增加了一个 `Ok(())` 作为返回值。这段代码就可以编译了。

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

示例 9-12: 修改 `main` 返回 `Result<(), E>` 允许对 `Result` 值使用 `?` 运算符

`Box<dyn Error>` 类型是一个 **trait 对象**（*trait object*）第十八章顾及不同类型值的 trait 对象”部分会做介绍。目前可以将 `Box<dyn Error>` 理解为 “任何类型的错误”。在返回 `Box<dyn Error>` 错误类型 `main` 函数中对 `Result` 使用 `?` 是允许的，因为它允许任何 `Err` 值提前返回。即便 `main` 函数体从来只会返回 `std::io::Error` 错误类型，通过指定 `Box<dyn Error>`，这个签名也仍是正确的，甚至当 `main` 函数体中增加更多返回其他错误类型的代码，这个函数签名依然保持正确。

当 `main` 函数返回 `Result<(), E>`，如果 `main` 返回 `Ok(())` 可执行程序会以 `0` 值退出，而如果 `main` 返回 `Err` 值则会以非零值退出；成功退出的程序会返回整数 `0`，运行错误的程序会返回非 `0` 的整数。Rust 也会从二进制程序中返回与这个惯例相兼容的整数。

`main` 函数也可以返回任何实现了 `std::process::Termination` trait 的类型，它包含了一个返回 `ExitCode` 的 `report` 函数。请查阅标准库文档了解更多为自定义类型实现 `Termination` trait 的细节。

现在我们讨论过了调用 `panic!` 或返回 `Result` 的细节，让我们回到在不同场景下如何决定使用哪种方式的问题。

要不要 `panic!`

那么，该如何决定何时应该 `panic!` 以及何时应该返回 `Result` 呢？如果代码 panic，就没有恢复的可能。你可以选择对任何错误场景都调用 `panic!`，不管是否有可能恢复，不过这样就是你代替调用者决定了这是不可恢复的。选择返回 `Result` 值的话，就将选择权交给了调用者，而不是代替他们做出决定。调用者可能会选择以符合他们场景的方式尝试恢复，或者也可能干脆就认为 `Err` 是不可恢复的，所以他们也可能会调用 `panic!` 并将可恢复的错误变成了不可恢复的错误。因此返回 `Result` 是定义可能会失败的函数的一个好的默认选择。

在一些类似示例、原型代码（prototype code）和测试中，panic 比返回 `Result` 更为合适，下文中会讨论合适的原因，紧接着讨论另外一种特殊情况，即有些场景编译器无法认识这个分支代码是不可能走到的，但是人类可以判断出来的，这种场景也可以用 panic!。另外章节最后会总结一些在库代码中如何决定是否要 panic 的通用指导原则。

示例、代码原型和测试

当你编写一个示例来展示一些概念时，在拥有健壮的错误处理代码的同时也会使得例子不那么明确。例如，调用一个类似 `unwrap` 这样可能 `panic!` 的方法可以被理解为一个你实际希望程序处理错误方式的占位符，它根据其余代码运行方式可能会各不相同。

类似地，在我们准备好决定如何处理错误之前，`unwrap`和`expect`方法在原型设计时非常方便。当我们准备好让程序更加健壮时，它们会在代码中留下清晰的标记。

如果方法调用在测试中失败了，我们希望这个测试都失败，即便这个方法并不是需要测试的功能。因为 `panic!` 会将测试标记为失败，此时调用 `unwrap` 或 `expect` 是恰当的。

当我们比编译器知道更多的情况

当你有一些其他的逻辑来确保 `Result` 会是 `Ok` 值时，调用 `unwrap` 或者 `expect` 也是合适的，虽然编译器无法理解这种逻辑。你仍然需要处理一个 `Result` 值：即使在你的特定情况下逻辑上是不可能的，你所调用的任何操作仍然有可能失败。如果通过人工检查代码来确保永远也不会出现 `Err` 值，那么调用 `unwrap` 也是完全可以接受的，更好的做法是在 `expect` 的提示文本中说明你认为永远不会出现 `Err` 的原因。下面是一个示例:。这里是一个例子：

```
    use std::net::IpAddr;

    let home: IpAddr = "127.0.0.1"
        .parse()
        .expect("Hardcoded IP address should be valid");
```

我们通过解析一个硬编码的字符来创建一个 `IpAddr` 实例。可以看出 `127.0.0.1` 是一个有效的 IP 地址，所以这里使用 `expect` 是可以接受的。然而，拥有一个硬编码的有效的字符串也不能改变 `parse` 方法的返回值类型：它仍然是一个 `Result` 值，而编译器仍然会要求我们处理这个 `Result`，好像还是有可能出现 `Err` 变体那样。这是因为编译器还没有智能到可以识别出这个字符串总是一个有效的 IP 地址。如果 IP 地址字符串来源于用户而不是硬编码进程序中的话，那么就**确实**有失败的可能性，这时确实需要我们以一种更健壮的方式处理 `Result`。提及这个 IP 地址是硬编码的假设会促使我们将来把 `expect` 替换为更好的错误处理，我们应该从其它代码获取 IP 地址。

错误处理指导原则

在当有可能会导致有害状态（bad state）的情况下建议使用 `panic!` —— 在这里，**有害状态**（*bad state*）是指当一些假设、保证、协议或不可变性被打破的状态，例如无效的值、自相矛盾的值或者被传递了不存在的值 —— 外加如下几种情况：

- 有害状态是非预期的行为，与偶尔会发生的行为相对，比如用户输入了错误格式的数据。
- 在此之后代码的运行依赖于不处于这种有害状态，而不是在每一步都检查是否有问题。
- 没有可行的手段来将有害状态信息编码进所使用的类型中的情况。我们会在第十八章"将状态和行为编码为类型"部分通过一个例子来说明我们的意思。

如果别人调用你的代码并传递了一个没有意义的值，尽最大可能返回一个错误，如此库的用户就可以决定在这种情况下该如何处理。然而在继续执行代码是不安全或有害的情况下，最好的选择可能是调用 `panic!` 并警告库的用户他们的代码中有 bug，这样他们就会在开发时进行修复。类似的，如果你正在调用不受你控制的外部代码，并且它返回了一个你无法修复的无效状态，那么 `panic!` 往往是合适的。

然而当错误预期会出现时，返回 `Result` 仍要比调用 `panic!` 更为合适。这样的例子包括解析器接收到格式错误的数据，或者 HTTP 请求返回了一个表明触发了限流的状态。在这些例子中，应该通过返回 `Result` 来表明失败预期是可能的，而调用者就必须决定该如何处理这个问题。

当你的代码在进行一个使用无效值进行调用时可能将用户置于风险中的操作时，代码应该首先验证值是有效的，并在其无效时 `panic!`。这主要是出于安全的原因：尝试操作无效数据会暴露代码漏洞，这就是标准库在尝试越界访问数组时会 `panic!` 的主要原因：尝试访问不属于当前数据结构的内存是一个常见的安全隐患。函数通常都遵循**契约**（*contracts*）：它们的行为只有在输入满足特定条件时才能得到保证。当违反契约时 panic 是有道理的，因为这通常代表调用方的 bug，而且这也不是那种你希望所调用的代码必须处理的错误。事实上所调用的代码也没有合理的方式来恢复，而是需要调用方的**开发者**修复其代码。函数的契约，尤其是当违反它会造成 panic 的契约，应该在函数的 API 文档中进行说明。

虽然在所有函数中都拥有许多错误检查是冗长而烦人的。幸运的是，可以利用 Rust 的类型系统（以及编译器的类型检查）为你进行很多检查。如果函数有一个特定类型的参数，可以在知晓编译器已经确保其拥有一个有效值的前提下进行你的代码逻辑。例如，如果你使用了一个并不是 `Option` 的类型，则程序期望它是**有值**的并且不是**空值**。你的代码无需处理 `Some` 和 `None` 这两种情况，它只会有一种情况就是绝对会有一个值。尝试向函数传递空值的代码甚至根本不能编译，所以你的函数在运行时没有必要判空。另外一个例子是使用像 `u32` 这样的无符号整型，也会确保它永远不为负。

创建自定义类型进行有效性验证

让我们使用 Rust 类型系统的思想来进一步确保值的有效性，并尝试创建一个自定义类型以进行验证。回忆一下第二章的猜猜看游戏，我们的代码要求用户猜测一个 1 到 100 之间的数字，在将其与秘密数字做比较之前我们从未验证用户的猜测是位于这两个数字之间的，我们只验证它是否为正。在这种情况下，其影响并不是很严重：“Too high” 或 “Too low” 的输出仍然是正确的。但是这是一个很好的引导用户得出有效猜测的辅助，例如当用户猜测一个超出范围的数字或者输入字母时采取不同的行为。

一种实现方式是将猜测解析成 `i32` 而不仅仅是 `u32`，来默许输入负数，接着检查数字是否在范围内，像这样：

文件名：src/main.rs

```
    loop {
        // --snip--

        let guess: i32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        if guess < 1 || guess > 100 {
            println!("The secret number will be between 1 and 100.");
            continue;
        }

        match guess.cmp(&secret_number) {
            // --snip--
    }
```

`if` 表达式检查了值是否超出范围，告诉用户出了什么问题，并调用 `continue` 开始下一次循环，请求另一个猜测。`if` 表达式之后，就可以在知道 `guess` 在 1 到 100 之间的情况下与秘密数字作比较了。

然而，这并不是一个理想的解决方案：如果让程序仅仅处理 1 到 100 之间的值是一个绝对需要满足的要求，而且程序中的很多函数都有这样的要求，在每个函数中都有这样的检查将是非常冗余的（并可能潜在地影响性能）。

相反我们可以在一个专用的模块中创建一个新类型来将验证放入创建其实例的函数中，而不是到处重复这些检查。这样就可以安全地在函数签名中使用新类型并相信它们接收到的值。示例 9-13 中展示了一个定义 `Guess` 类型的方法，只有在 `new` 函数接收到 1 到 100 之间的值时才会创建 `Guess` 的实例：

文件名：src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {value}.");
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

示例 9-13：一个 `Guess` 类型，它只在值位于 1 和 100 之间时才继续

首先，我们创建一个名为 `guessing_game` 的新模块。接着定义一个包含 `i32` 类型字段 `value` 的结构体 `Guess`。这里是储存猜测值的地方。

接着在 `Guess` 上实现了一个叫做 `new` 的关联函数来创建 `Guess` 的实例。`new` 定义为接收一个 `i32` 类型的参数 `value` 并返回一个 `Guess`。`new` 函数中代码的测试确保了其值是在 1 到 100 之间的。如果 `value` 没有通过测试则调用 `panic!`，这会警告调用这个函数的程序员有一个需要修改的 bug，因为创建一个 `value` 超出范围的 `Guess` 将会违反 `Guess::new` 所遵循的契约。`Guess::new` 会出现 panic 的条件应该在其公有 API 文档中被提及；第十四章会涉及到在 API 文档中表明 `panic!` 可能性的相关规则。如果 `value` 通过了测试，我们新建一个 `Guess`，其字段 `value` 将被设置为参数 `value` 的值，接着返回这个 `Guess`。

接着，我们实现了一个借用了 `self` 的方法 `value`，它没有任何其他参数并返回一个 `i32`。这类方法有时被称为 *getter*，因为它的目的就是返回对应字段的数据。这样的公有方法是必要的，因为 `Guess` 结构体的 `value` 字段是私有的。私有的字段 `value` 是很重要的，这样使用 `Guess` 结构体的代码将不允许直接设置 `value` 的值：调用者**必须**使用 `Guess::new` 方法来创建一个 `Guess` 的实例，这就确保了不会存在一个 `value` 没有通过 `Guess::new` 函数的条件检查的 `Guess`。

于是，一个接收或返回 1 到 100 之间数字的函数就可以声明为接收（或返回） `Guess` 的实例，而不是 `i32`，同时其函数体中也无需进行任何额外的检查。

总结

Rust 的错误处理功能被设计为帮助你编写更加健壮的代码。`panic!` 宏代表一个程序无法处理的状态，并停止执行而不是使用无效或不正确的值继续处理。Rust 类型系统的 `Result` 枚举代表操作可能会在一种可以恢复的情况下失败。可以使用 `Result` 来告诉代码调用者他需要处理潜在的成功或失败。在适当的场景使用 `panic!` 和 `Result` 将会使你的代码在面对不可避免的错误时显得更加可靠。

现在我们已经见识过了标准库中 `Option` 和 `Result` 泛型枚举的能力了，在下一章让我们聊聊泛型是如何工作的，以及如何在你的代码中使用它们。

泛型、Trait 和生命周期

每一个编程语言都有高效处理重复概念的工具。在 Rust 中其工具之一就是 **泛型**（*generics*）：具体类型或其他属性的抽象替代。我们可以表达泛型的属性，比如它们的行为或如何与其他泛型相关联，而不需要在编写和编译代码时知道它们在这里实际上代表什么。

函数可以获取一些不同于 `i32` 或 `String` 这样具体类型的泛型参数，就像一个获取未知类型值的函数可以对多种具体类型的值运行同一段代码一样。事实上我们已经使用过第六章的 `Option<T>`，第八章的 `Vec<T>` 和 `HashMap<K, V>`，以及第九章的 `Result<T, E>` 这些泛型了。本章会探索如何使用泛型定义我们自己的类型、函数和方法！

首先，我们将回顾一下提取函数以减少代码重复的机制。接下来，我们将使用相同的技术，从两个仅参数类型不同的函数中创建一个泛型函数。我们也会讲到结构体和枚举定义中的泛型。

之后，我们讨论如何使用 **trait** 定义泛型行为的方法。trait 可以与泛型结合来将泛型限制为只接受拥有特定行为的类型，而不是任意类型。

最后介绍 **生命周期**（*lifetimes*）：一类允许我们向编译器提供引用如何相互关联的泛型。Rust 的生命周期功能允许在更多场景下借用值的同时仍然使编译器能够检查这些引用的有效性而不用借助我们的帮助。

提取函数来减少重复

泛型允许我们使用一个可以代表多种类型的占位符来替换特定类型，以此来减少代码冗余。在深入了解泛型的语法之前，我们首先来看一种没有使用泛型的减少冗余的方法，即提取一个函数。在这个函数中，我们用一个可以代表多种值的占位符来替换具体的值。接着我们使用相同的技术来提取一个泛型函数！通过学习如何识别并提取可以整合进一个函数的重复代码，你也会开始识别出可以使用泛型的重复代码。

让我们从下面这个寻找列表中最大值的小程序开始，如示例 10-1 所示：

文件名：src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");
}
```

示例 10-1：在一个数字列表中寻找最大值的函数

这段代码获取一个整型列表，存放在变量 `number_list` 中。它将列表的第一个数字的引用放入了变量 `largest` 中。接着遍历了列表中的所有数字，如果当前值大于 `largest` 中储存的值，将 `largest` 替换为这个值。如果当前值小于或者等于目前为止的最大值，变量保持不变，并且代

码移动到列表中的下一个数字。当列表中所有值都被考虑到之后，`largest` 将会指向最大值，在这里也就是 100。

我们的任务是在两个不同的数字列表中寻找最大值。为此我们可以选择重复示例 10-1 中的代码在程序的两个不同位置使用相同的逻辑，如示例 10-2 所示：

文件名：src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {largest}");
}
```

示例 10-2：寻找**两个**数组最大值的代码

虽然代码能够执行，但是重复的代码是冗余且容易出错的，更新逻辑时我们不得不记住需要修改多处地方的代码。

为了消除重复，我们要创建一层抽象，定义一个处理任意整型列表作为参数的函数。这个方案使得代码更简洁，并且表现了寻找任意列表中最大值这一概念。

在示例 10-3 的程序中将寻找最大值的代码提取到了一个叫做 `largest` 的函数中。接着我们调用该函数来寻找示例 10-2 中两个列表中的最大值。之后也可以将该函数用于任何可能的 `i32` 值的列表。

文件名：src/main.rs

```
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }
```

```
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {result}");
}
```

示例 10-3：抽象后的寻找两个数字列表最大值的代码

`largest` 函数有一个参数 `list`，它代表会传递给函数的任何具体的 `i32`值的 slice。函数定义中的 `list` 代表任何 `&[i32]`。当调用 `largest` 函数时，其代码实际上运行于我们传递的特定值上。

总的来说，从示例 10-2 到示例 10-3 中代码的修改经历了如下几步：

1. 找出重复代码。
2. 将重复代码提取到了一个函数中，并在函数签名中指定了代码中的输入和返回值。
3. 将重复代码的两个实例，改为调用函数。

接下来我们会使用相同的步骤通过泛型来减少重复。与函数体可以处理任意的 `list` 而不是具体的值一样，泛型也允许代码处理任意类型。

如果我们有两个函数，一个寻找一个 `i32` 值的 slice 中的最大项而另一个寻找 `char` 值的 slice 中的最大项该怎么办？该如何消除重复呢？让我们拭目以待！

泛型数据类型

我们可以使用泛型为像函数签名或结构体这样的项创建定义，这样它们就可以用于多种不同的具体数据类型。让我们看看如何使用泛型定义函数、结构体、枚举和方法，然后我们将讨论泛型如何影响代码性能。

在函数定义中使用泛型

当使用泛型定义函数时，本来在函数签名中指定参数和返回值的类型的地方，会改用泛型来表示。采用这种技术，使得代码适应性更强，从而为函数的调用者提供更多的功能，同时也避免了代码的重复。

回到 `largest` 函数，示例 10-4 中展示了两个函数，它们的功能都是寻找 slice 中最大值。接着我们使用泛型将其合并为一个函数。

文件名：src/main.rs

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {result}");
}
```

示例 10-4：两个函数，不同点只是名称和签名类型

`largest_i32` 函数是从示例 10-3 中摘出来的，它用来寻找 slice 中最大的 `i32`。`largest_char` 函数寻找 slice 中最大的 `char`。因为两者函数体的代码是一样的，我们可以定义一个函数，再引进泛型参数来消除这种重复。

为了参数化这个新函数中的这些类型，我们需要为类型参数命名，道理和给函数的形参起名一样。任何标识符都可以作为类型参数的名字。这里选用 `T`，因为传统上来说，Rust 的类型参数名字都比较短，通常仅为一个字母，同时，Rust 类型名的命名规范是首字母大写驼峰式命名法（UpperCamelCase）。`T` 作为 “type” 的缩写是大部分 Rust 程序员的首选。

如果要在函数体中使用参数，就必须在函数签名中声明它的名字，好让编译器知道这个名字指代的是什么。同理，当在函数签名中使用一个类型参数时，必须在使用它之前就声明它。为了定义泛型版本的 `largest` 函数，类型参数声明位于函数名称与参数列表中间的尖括号 `<>` 中，像这样：

```
fn largest<T>(list: &[T]) -> &T {
```

可以这样理解这个定义：函数 `largest` 有泛型类型 `T`。它有个参数 `list`，其类型是元素为 `T` 的 slice。`largest` 函数会返回一个与 `T` 相同类型的引用。

示例 10-5 中的 `largest` 函数在它的签名中使用了泛型，统一了两个实现。该示例也展示了如何调用 `largest` 函数，把 `i32` 值的 slice 或 `char` 值的 slice 传给它。请注意这些代码还不能编译。

文件名：src/main.rs



```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {result}");
}
```

示例 10-5：一个使用泛型参数的 `largest` 函数定义，尚不能编译

如果现在就编译这段代码，会出现如下错误：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T` with trait `PartialOrd`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

帮助说明中提到了 `std::cmp::PartialOrd`，这是一个 *trait*。下一部分会讲到 trait。不过简单来说，这个错误表明 `largest` 的函数体不能适用于 `T` 的所有可能的类型。因为在函数体需要比较 `T` 类型的值，不过它只能用于我们知道如何排序的类型。为了开启比较功能，标准库中定义的 `std::cmp::PartialOrd` trait 可以实现类型的比较功能（查看附录 C 获取该 trait 的更多信息）。依照帮助说明中的建议，我们限制 `T` 只对实现了 `PartialOrd` 的类型有效后代码就可以编译了，因为标准库为 `i32` 和 `char` 实现了 `PartialOrd`。

结构体定义中的泛型

同样也可以用 `<>` 语法来定义结构体，它包含一个或多个泛型参数类型字段。示例 10-6 定义了一个可以存放任何类型的 `x` 和 `y` 坐标值的结构体 `Point`：

文件名：src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

示例 10-6：`Point` 结构体存放了两个 `T` 类型的值 `x` 和 `y`

在结构体定义中使用泛型的语法类似于函数定义中使用泛型。首先，必须在结构体名称后面的尖括号中声明泛型参数的名称。接着在结构体定义中可以指定具体数据类型的位置使用泛型类型。

注意 `Point<T>` 的定义中只使用了一个泛型类型，这个定义表明结构体 `Point<T>` 对于一些类型 `T` 是泛型的，而且字段 `x` 和 `y` **都是** 相同类型的，无论它具体是何类型。如果尝试创建一个有不同类型值的 `Point<T>` 的实例，像示例 10-7 中的代码就不能编译：

文件名：src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```



示例 10-7：字段 `x` 和 `y` 的类型必须相同，因为它们都有相同的泛型类型 `T`

在这个例子中，当把整型值 `5` 赋值给 `x` 时，就告诉了编译器这个 `Point<T>` 实例中的泛型 `T` 全是整型。接着指定 `y` 为浮点值 `4.0`，因为 `y` 被定义为与 `x` 相同类型，所以将会得到一个像这样的类型不匹配错误：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found floating-
point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

如果想要定义一个 `x` 和 `y` 可以有不同类型且仍然是泛型的 `Point` 结构体，我们可以使用多个泛型类型参数。在示例 10-8 中，我们修改 `Point` 的定义为拥有两个泛型类型 `T` 和 `U`。其中字段 `x` 是 `T` 类型的，而字段 `y` 是 `U` 类型的：

文件名：src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

示例 10-8：使用两个泛型的 `Point`，这样 `x` 和 `y` 可能是不同类型

现在所有这些 `Point` 实例都合法了！你可以在定义中使用任意多的泛型类型参数，不过太多的话，代码将难以阅读和理解。当你发现代码中需要很多泛型时，这可能表明你的代码需要重构分解成更小的结构。

枚举定义中的泛型

和结构体类似，枚举也可以在成员中存放泛型数据类型。第六章我们曾用过标准库提供的 `Option<T>` 枚举，这里再回顾一下：

```
enum Option<T> {
    Some(T),
    None,
}
```

现在这个定义应该更容易理解了。如你所见 `Option<T>` 是一个拥有泛型 `T` 的枚举，它有两个成员：`Some`，它存放了一个类型 `T` 的值，和不存在任何值的`None`。通过 `Option<T>` 枚举可以表达有一个可能的值的抽象概念，同时因为 `Option<T>` 是泛型的，无论这个可能的值是什么类型都可以使用这个抽象。

枚举也可以拥有多个泛型类型。第九章使用过的 `Result` 枚举定义就是一个这样的例子：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` 枚举有两个泛型类型，`T` 和 `E`。`Result` 有两个成员：`Ok`，它存放一个类型 `T` 的值，而 `Err` 则存放一个类型 `E` 的值。这个定义使得 `Result` 枚举能很方便的表达任何可能成功（返回 `T` 类型的值）也可能失败（返回 `E` 类型的值）的操作。实际上，这就是我们在示例 9-3 用来打开文件的方式：当成功打开文件的时候，`T` 对应的是 `std::fs::File` 类型；而当打开文件出现问题时，`E` 的值则是 `std::io::Error` 类型。

当你意识到代码中定义了多个结构体或枚举，它们不一样的地方只是其中的值的类型的时候，不妨通过泛型类型来避免重复。

方法定义中的泛型

在为结构体和枚举实现方法时（像第五章那样），一样也可以用泛型。示例 10-9 中展示了示例 10-6 中定义的结构体 `Point<T>`，和在其上实现的名为 `x` 的方法。

文件名：src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };
```

```
    println!("p.x = {}", p.x());
}
```

示例 10-9：在 `Point<T>` 结构体上实现方法 `x`，它返回 `T` 类型的字段 `x` 的引用

这里在 `Point<T>` 上定义了一个叫做 `x` 的方法用于返回字段 `x` 中数据的引用。

注意必须在 `impl` 后面声明 `T`，这样就可以在 `Point<T>` 上实现的方法中使用 `T` 了。通过在 `impl` 之后声明泛型 `T`，Rust 就知道 `Point` 的尖括号中的类型是泛型而不是具体类型。我们可以为泛型参数选择一个与结构体定义中声明的泛型参数所不同的名称，不过依照惯例使用了相同的名称。如果你在`impl`中编写一个声明泛型类型的方法，那么该方法将在任何类型的实例上定义，无论最终用什么具体类型来替换泛型类型。（译者注：以示例 10-9 为例，`impl` 中声明了泛型类型参数 `T`，`x` 是编写在 `impl` 中的方法，`x` 方法将会定义在 `Point<T>` 的任何实例上，无论最终替换泛型类型参数 `T` 的是何具体类型）。

定义方法时也可以为泛型指定限制（constraint）。例如，可以选择为 `Point<f32>` 实例实现方法，而不是为泛型 `Point` 实例。示例 10-10 展示了一个没有在 `impl` 之后（的尖括号）声明泛型的例子，这里使用了一个具体类型，`f32`：

文件名：src/main.rs

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

示例 10-10：构建一个只用于拥有泛型参数 `T` 的结构体的具体类型的 `impl` 块

这段代码意味着 `Point<f32>` 类型会有一个方法 `distance_from_origin`，而其他 `T` 不是 `f32` 类型的 `Point<T>` 实例则没有定义此方法。这个方法计算点实例与坐标 (0.0, 0.0) 之间的距离，并使用了只能用于浮点型的数学运算符。

结构体定义中的泛型类型参数并不总是与结构体方法签名中使用的泛型是同一类型。示例 10-11 中为 `Point` 结构体使用了泛型类型 `X1` 和 `Y1`，为 `mixup` 方法签名使用了 `X2` 和 `Y2` 来使得示例更加清楚。这个方法用 `self` 的 `Point` 类型的 `x` 值（类型 `X1`）和参数的 `Point` 类型的 `y` 值（类型 `Y2`）来创建一个新 `Point` 类型的实例：

文件名：src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
```

```
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

示例 10-11：方法使用了与结构体定义中不同类型的泛型

在 `main` 函数中，定义了一个有 `i32` 类型的 `x`（其值为 `5`）和 `f64` 的 `y`（其值为 `10.4`）的 `Point`。`p2` 则是一个有着字符串 slice 类型的 `x`（其值为 `"Hello"`）和 `char` 类型的 `y`（其值为 `c`）的 `Point`。在 `p1` 上以 `p2` 作为参数调用 `mixup` 会返回一个 `p3`，它会有一个 `i32` 类型的 `x`，因为 `x` 来自 `p1`，并拥有一个 `char` 类型的 `y`，因为 `y` 来自 `p2`。`println!` 会打印出 `p3.x = 5, p3.y = c`。

这个例子的目的是展示一些泛型通过 `impl` 声明而另一些通过方法定义声明的情况。这里泛型参数 `X1` 和 `Y1` 声明于 `impl` 之后，因为它们与结构体定义相对应。而泛型参数 `X2` 和 `Y2` 声明于 `fn mixup` 之后，因为它们只是相对于方法本身的。

泛型代码的性能

你可能会好奇使用泛型类型参数是否会有运行时消耗。好消息是泛型并不会使程序比具体类型运行得慢。

Rust 通过在编译时进行泛型代码的**单态化**（*monomorphization*）来保证效率。单态化是一个通过填充编译时使用的具体类型，将通用代码转换为特定代码的过程。

在这个过程中，编译器所做的工作正好与示例 10-5 中我们创建泛型函数的步骤相反。编译器寻找所有泛型代码被调用的位置并使用泛型代码针对具体类型生成代码。

让我们看看这在标准库的 `Option<T>` 枚举上是如何工作的：

```
let integer = Some(5);
let float = Some(5.0);
```

当 Rust 编译这些代码的时候，它会进行单态化。编译器会读取传递给 `Option<T>` 的值并发现有两种 `Option<T>`：一个对应 `i32` 另一个对应 `f64`。为此，它会将泛型定义 `Option<T>` 展开为两个针对 `i32` 和 `f64` 的定义，接着将泛型定义替换为这两个具体的定义。

编译器生成的单态化版本的代码看起来像这样（编译器会使用不同于如下假想的名字）：

文件名：src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
```

```
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

泛型 `Option<T>` 被编译器替换为了具体的定义。因为 Rust 会将每种情况下的泛型代码编译为具体类型，使用泛型没有运行时开销。当代码运行时，它的执行效率就跟好像手写每个具体定义的重复代码一样。这个单态化过程正是 Rust 泛型在运行时极其高效的原因。

Trait：定义共同行为

trait 定义了某个特定类型拥有可能与其他类型共享的功能。可以通过 trait 以一种抽象的方式定义共同行为。可以使用 *trait bounds* 指定泛型是任何拥有特定行为的类型。

> 注意：*trait* 类似于其他语言中的常被称为 **接口**（*interfaces*）的功能，虽然有一些不同。

定义 trait

一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。trait 定义是一种将方法签名组合起来的方法，目的是定义一个实现某些目的所必需的行为的集合。

例如，这里有多个存放了不同类型和属性文本的结构体：结构体 `NewsArticle` 用于存放发生于世界各地的新闻故事，而结构体 `SocialPost` 最多只能存放 280 个字符的内容，以及指示该帖子是新发布的、转发的还是对另一条帖子的回复的元数据。

我们想要创建一个名为 `aggregator` 的多媒体聚合库用来显示可能储存在 `NewsArticle` 或 `SocialPost` 实例中的数据摘要。为了实现功能，每个结构体都要能够获取摘要，这样的话就可以调用实例的 `summarize` 方法来请求摘要。示例 10-12 中展示了一个表现这个概念的公有 `Summary` trait 的定义：

文件名：src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

示例 10-12：`Summary` trait 定义，它包含由 `summarize` 方法提供的行为

这里使用 `trait` 关键字来声明一个 trait，后面是 trait 的名字，在这个例子中是 `Summary`。我们也声明 `trait` 为 `pub` 以便依赖这个 crate 的其它 crate 也可以使用这个 trait，正如我们见过的一些示例一样。在大括号中声明描述实现这个 trait 的类型所需要的行为的方法签名，在这个例子中是 `fn summarize(&self) -> String`。

在方法签名后跟分号，而不是在大括号中提供其实现。接着每一个实现这个 trait 的类型都需要提供其自定义行为的方法体，编译器也会确保任何实现 `Summary` trait 的类型都拥有与这个签名的定义完全一致的 `summarize` 方法。

trait 体中可以有多个方法：一行一个方法签名且都以分号结尾。

为类型实现 trait

现在我们定义了 `Summary` trait 的签名，接着就可以在多媒体聚合库中实现这个类型了。示例 10-13 中展示了 `NewsArticle` 结构体上 `Summary` trait 的一个实现，它使用标题、作者和创建的位置作为 `summarize` 的返回值。对于 `SocialPost` 结构体，我们选择将 `summarize` 定义为用户名后跟帖子全文作为返回值，并假设帖子内容已经被限制为 280 字符以内。

文件名：src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct SocialPost {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub repost: bool,
}

impl Summary for SocialPost {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

示例 10-13：在 `NewsArticle` 和 `SocialPost` 类型上实现 `Summary` trait

在类型上实现 trait 类似于实现常规方法。区别在于 `impl` 关键字之后，我们提供需要实现 trait 的名称，接着是 `for` 和需要实现 trait 的类型的名称。在 `impl` 块中，使用 trait 定义中的方法签名，不过不再后跟分号，而是需要在大括号中编写函数体来为特定类型实现 trait 方法所拥有的行为。

现在库在 `NewsArticle` 和 `SocialPost` 上实现了 `Summary` trait，crate 的用户可以像调用常规方法一样调用 `NewsArticle` 和 `SocialPost` 实例的 trait 方法了。唯一的区别是 trait 必须和类型一起引入作用域以便使用额外的 trait 方法。这是一个二进制 crate 如何利用 `aggregator` 库 crate 的例子：

```
use aggregator::{SocialPost, Summary};

fn main() {
    let post = SocialPost {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        repost: false,
    };

    println!("1 new post: {}", post.summarize());
}
```

这会打印出 `1 new post: horse_ebooks: of course, as you probably already know, people`。

其他依赖 `aggregator` crate 的 crate 也可以将 `Summary` 引入作用域以便为其自己的类型实现该 trait。需要注意的限制是，只有在 trait 或类型至少有一个属于当前 crate 时，我们才能对类型实现该 trait。例如，可以为 `aggregator` crate 的自定义类型 `SocialPost` 实现如标准库中的 `Display` trait，这是因为 `SocialPost` 类型位于 `aggregator` crate 本地的作用域中。类似地，也可以在 `aggregator` crate 中为 `Vec<T>` 实现 `Summary`，这是因为 `Summary` trait 位于 `aggregator` crate 本地作用域中。

但是不能为外部类型实现外部 trait。例如，不能在 `aggregator` crate 中为 `Vec<T>` 实现 `Display` trait。这是因为 `Display` 和 `Vec<T>` 都定义于标准库中，它们并不位于 `aggregator` crate 本地作用域中。这个限制是被称为**相干性**（*coherence*）的程序属性的一部分，或者更具体的说是 **孤儿规则**（*orphan rule*），其得名于不存在父类型。这条规则确保了其他人编写的代码不会破坏你的代码，反之亦然。没有这条规则的话，两个 crate 可以分别对相同类型实现相同的 trait，而 Rust 将无从得知应该使用哪一个实现。

默认实现

有时为 trait 中的某些或全部方法提供默认的行为，而不是在每个类型的每个实现中都定义自己的行为是很有用的。这样当为某个特定类型实现 trait 时，可以选择保留或重载每个方法的默认行为。

示例 10-14 中我们为 `Summary` trait 的 `summarize` 方法指定一个默认的字符串值，而不是像示例 10-12 中那样只是定义方法签名：

文件名：src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

示例 10-14：`Summary` trait 的定义，带有一个 `summarize` 方法的默认实现

如果想要对 `NewsArticle` 实例使用这个默认实现，可以通过 `impl Summary for NewsArticle {}` 指定一个空的 `impl` 块。

虽然我们不再直接为 `NewsArticle` 定义 `summarize` 方法了，但是我们提供了一个默认实现并且指定 `NewsArticle` 实现 `Summary` trait。因此，我们仍然可以对 `NewsArticle` 实例调用 `summarize` 方法，如下所示：

```
    let article = NewsArticle {
        headline: String::from("Penguins win the Stanley Cup Championship!"),
        location: String::from("Pittsburgh, PA, USA"),
        author: String::from("Iceburgh"),
        content: String::from(
            "The Pittsburgh Penguins once again are the best \
             hockey team in the NHL.",
        ),
    };

    println!("New article available! {}", article.summarize());
```

这段代码会打印 `New article available! (Read more...)`。

为 `summarize` 创建默认实现并不要求对示例 10-13 中 `SocialPost` 上的 `Summary` 实现做任何改变。其原因是重载一个默认实现的语法与实现没有默认实现的 trait 方法的语法一样。

默认实现允许调用相同 trait 中的其他方法，哪怕这些方法没有默认实现。如此，trait 可以提供很多有用的功能而只需要实现指定一小部分内容。例如，我们可以定义 `Summary` trait，使其具有一个需要实现的 `summarize_author` 方法，然后定义一个 `summarize` 方法，此方法的默认实现调用 `summarize_author` 方法：

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

为了使用这个版本的 `Summary`，只需在为类型实现 trait 时定义 `summarize_author` 即可：

```
impl Summary for SocialPost {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

一旦定义了 `summarize_author`，我们就可以对 `SocialPost` 结构体的实例调用 `summarize` 了，而 `summarize` 的默认实现会调用我们提供的 `summarize_author` 定义。因为实现了 `summarize_author`，`Summary` trait 就提供了 `summarize` 方法的功能，且无需编写更多的代码。

```
    let post = SocialPost {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        repost: false,
    };

    println!("1 new social post: {}", post.summarize());
```

这会打印出 `1 new post: (Read more from @horse_ebooks...)`。

注意无法从相同方法的重载实现中调用默认方法。

trait 作为参数

知道了如何定义 trait 和在类型上实现这些 trait 之后，我们可以探索一下如何使用 trait 来接受多种不同类型的参数。示例 10-13 中为 `NewsArticle` 和 `SocialPost` 类型实现了 `Summary` trait，用其来定义了一个函数 `notify` 来调用其参数 `item` 上的 `summarize` 方法，该参数是实现了 `Summary` trait 的某种类型。为此可以使用 `impl Trait` 语法，像这样：

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

对于 `item` 参数，我们指定了 `impl` 关键字和 trait 名称，而不是具体的类型。该参数支持任何实现了指定 trait 的类型。在 `notify` 函数体中，可以调用任何来自 `Summary` trait 的方法，比如 `summarize`。我们可以传递任何 `NewsArticle` 或 `SocialPost` 的实例来调用 `notify`。任何用其它如 `String` 或 `i32` 的类型调用该函数的代码都不能编译，因为它们没有实现 `Summary`。

Trait Bound 语法

`impl Trait` 语法更直观，但它实际上是更长形式的 *trait bound* 语法的语法糖。它看起来像：

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

这种更冗长的写法与上一节的示例等价，但更为冗长。trait bound 与泛型参数声明在一起，位于尖括号中的冒号后面。

`impl Trait` 很方便，适用于短小的例子。更长的 trait bound 则适用于更复杂的场景。例如，可以获取两个实现了 `Summary` 的参数。使用 `impl Trait` 的语法看起来像这样：

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

这适用于 `item1` 和 `item2` 允许是不同类型的情况（只要它们都实现了 `Summary`）。不过如果你希望强制它们都是相同类型呢？但如果我们希望强制两个参数必须具有相同类型，则必须使用 trait bound，如下所示：

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

泛型 `T` 被指定为 `item1` 和 `item2` 的参数限制，如此传递给参数 `item1` 和 `item2` 值的具体类型必须一致。

通过 `+` 指定多个 trait bound

我们也可以指定多个 trait bound。假设我们希望 `notify` 在 `item` 上既能使用格式化显示，又能使用 `summarize` 方法：在 `notify` 的定义中，指定 `item` 必须同时实现 `Display` 和 `Summary` 两个 trait。这可以通过 `+` 语法实现：

```
pub fn notify(item: &(impl Summary + Display)) {
```

`+` 语法也适用于泛型的 trait bound：

```
pub fn notify<T: Summary + Display>(item: &T) {
```

通过指定这两个 trait bound，`notify` 的函数体可以调用 `summarize` 并使用 `{}` 来格式化 `item`。

通过 `where` 简化 trait bound

然而，使用过多的 trait bound 也有缺点。每个泛型有其自己的 trait bound，所以有多个泛型参数的函数在名称和参数列表之间会有很长的 trait bound 信息，这使得函数签名难以阅读。为此，Rust 有另一个在函数签名之后的 `where` 从句中指定 trait bound 的语法。所以除了这么写：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

还可以像这样使用 `where` 从句：

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

这个函数签名就显得不那么杂乱，函数名、参数列表和返回值类型都离得很近，看起来跟没有那么多 trait bounds 的函数很像。

返回实现了 trait 的类型

也可以在返回值中使用 `impl Trait` 语法，来返回实现了某个 trait 的类型：

```
fn returns_summarizable() -> impl Summary {
    SocialPost {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        repost: false,
    }
}
```

通过使用 `impl Summary` 作为返回值类型，我们指定了 `returns_summarizable` 函数返回某个实现了 `Summary` trait 的类型，但是不确定其具体的类型。在这个例子中 `returns_summarizable` 返回了一个 `SocialPost`，不过调用方并不知情。

返回一个只是指定了需要实现的 trait 的类型的能力在闭包和迭代器场景十分的有用，第十三章会介绍它们。闭包和迭代器创建只有编译器知道的类型，或者是非常非常长的类型。`impl Trait` 允许你简单的指定函数返回一个 `Iterator` 而无需写出实际的冗长的类型。

不过这只适用于返回单一类型的情况。例如，这段代码的返回值类型指定为返回 `impl Summary`，但是返回了 `NewsArticle` 或 `SocialPost` 就行不通：

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
```



```
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        SocialPost {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            repost: false,
        }
    }
}
```

这里尝试返回 `NewsArticle` 或 `SocialPost` 是不被允许的，原因在于编译器中 `impl Trait` 语法的实现限制。第十八章的 “顾及不同类型值的 trait 对象” 部分会介绍如何编写这样一个函数。

使用 trait bound 有条件地实现方法

通过使用带有 trait bound 的泛型参数的 `impl` 块，可以有条件地只为那些实现了特定 trait 的类型实现方法。例如，示例 10-15 中的类型 `Pair<T>` 总是实现了 `new` 方法并返回一个 `Pair<T>` 的实例（回忆一下第五章的 “定义方法” 部分，`Self` 是一个 `impl` 块类型的类型别名（type alias），在这里是 `Pair<T>`）。不过在下一个 `impl` 块中，只有那些为 `T` 类型实现了 `PartialOrd` trait（来允许比较） **和** `Display` trait（来启用打印）的 `Pair<T>` 才会实现 `cmp_display` 方法：

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

示例 10-15：根据 trait bound 在泛型上有条件的实现方法

也可以对任何实现了特定 trait 的类型有条件地实现 trait。对任何满足特定 trait bound 的类型实现 trait 被称为 *blanket implementations*，它们被广泛的用于 Rust 标准库中。例如，标准库为任何实现了 `Display` trait 的类型实现了 `ToString` trait。这个 `impl` 块看起来像这样：

```
impl<T: Display> ToString for T {
    // --snip--
}
```

因为标准库有了这些 blanket implementation，我们可以对任何实现了 `Display` trait 的类型调用由 `ToString` 定义的 `to_string` 方法。例如，可以将整型转换为对应的 `String` 值，因为整型实现了 `Display`：

```
let s = 3.to_string();
```

Blanket implementation 会出现在 trait 文档的 “Implementers” 部分。

Trait 和 trait bound 让我们能够使用泛型类型参数来减少重复，而且能够向编译器明确指定泛型类型需要拥有哪些行为。然后编译器可以利用 trait bound 信息检查代码中所用到的具体类型是否提供了正确的行为。在动态类型语言中，如果我们调用了一个未定义的方法，会在运行时出现错误。Rust 将这些错误移动到了编译时，甚至在代码能够运行之前就强迫我们修复问题。另外，我们也无需编写运行时检查行为的代码，因为在编译时就已经检查过了。这样既提升了性能又不必放弃泛型的灵活性。

生命周期确保引用有效

生命周期是另一类我们已经使用过的泛型。不同于确保类型有期望的行为，生命周期用于保证引用在我们需要的整个期间内都是有效的。

当在第四章讨论 “引用和借用” 部分时，我们遗漏了一个重要的细节：Rust 中的每一个引用都有其**生命周期**（*lifetime*），也就是引用保持有效的作用域。大部分时候生命周期是隐含并可以推断的，正如大部分时候类型也是可以推断的一样。类似于当因为有多种可能类型的时候必须注明类型，也会出现引用的生命周期以一些不同方式相关联的情况，所以 Rust 需要我们使用泛型生命周期参数来注明它们的关系，这样就能确保运行时实际使用的引用绝对是有效的。

生命周期注解甚至不是一个大部分语言都有的概念，所以这可能感觉起来有些陌生。虽然本章不可能涉及到它全部的内容，我们会讲到一些通常你可能会遇到的生命周期语法以便你熟悉这个概念。

生命周期避免了悬垂引用

生命周期的主要目标是避免**悬垂引用**（*dangling references*），后者会导致程序引用了非预期引用的数据。考虑一下示例 10-16 中的程序，它有一个外部作用域和一个内部作用域。



```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {r}");
}
```

示例 10-16：尝试使用离开作用域的值的引用

> 注意：示例 10-16、10-17 和 10-23 中声明了没有初始值的变量，所以这些变量存在于外部作用域。这乍看之下好像和 Rust 不允许存在空值相冲突。然而如果尝试在给它一个值之前使用这个变量，会出现一个编译时错误，这就说明了 Rust 确实不允许空值。

外部作用域声明了一个没有初值的变量 `r`，而内部作用域声明了一个初值为 `5` 的变量`x`。在内部作用域中，我们尝试将 `r` 的值设置为一个 `x` 的引用。接着在内部作用域结束后，尝试打印出 `r` 的值。这段代码不能编译因为 `r` 引用的值在尝试使用之前就离开了作用域。如下是错误信息：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
5 |         let x = 5;
```

```
  |             - binding `x` declared here
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r: {r}");
  |                   --- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

变量 `x` 并没有 “存在的足够久”。其原因是 `x` 在到达第 7 行内部作用域结束时就离开了作用域。不过 `r` 在外部作用域仍是有效的；作用域越大我们就说它 “存在的越久”。如果 Rust 允许这段代码工作，`r` 将会引用在 `x` 离开作用域时被释放的内存，这时尝试对 `r` 做任何操作都不能正常工作。那么 Rust 是如何决定这段代码是不被允许的呢？这得益于借用检查器。

借用检查器

Rust 编译器有一个**借用检查器**（*borrow checker*），它比较作用域来确保所有的借用都是有效的。示例 10-17 展示了与示例 10-16 相同的例子不过带有变量生命周期的注释：



```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {r}");   //          |
}                         // ---------+
```

示例 10-17：`r` 和 `x` 的生命周期注解，分别叫做 `'a` 和 `'b`

这里将 `r` 的生命周期标记为 `'a` 并将 `x` 的生命周期标记为 `'b`。如你所见，内部的 `'b` 块要比外部的生命周期 `'a` 小得多。在编译时，Rust 比较这两个生命周期的大小，并发现 `r` 拥有生命周期 `'a`，不过它引用了一个拥有生命周期 `'b` 的对象。程序被拒绝编译，因为生命周期 `'b` 比生命周期 `'a` 要小：被引用的对象比它的引用者存在的时间更短。

让我们看看示例 10-18 中这个并没有产生悬垂引用且可以正确编译的例子：

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {r}");   //   |       |
                          // --+       |
}                         // ----------+
```

示例 10-18：一个有效的引用，因为数据比引用有着更长的生命周期

这里 `x` 拥有生命周期 `'b`，比 `'a` 要大。这就意味着 `r` 可以引用 `x`：Rust 知道 `r` 中的引用在 `x` 有效的时候也总是有效的。

现在我们已经在一个具体的例子中展示了引用的生命周期位于何处，并讨论了 Rust 如何分析生命周期来保证引用总是有效的，接下来让我们聊聊在函数的上下文中参数和返回值的泛型生命周期。

函数中的泛型生命周期

让我们来编写一个返回两个字符串 slice 中较长者的函数。这个函数获取两个字符串 slice 并返回一个字符串 slice。一旦我们实现了 `longest` 函数，示例 10-19 中的代码应该会打印出 `The longest string is abcd`：

文件名：src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {result}");
}
```

示例 10-19：`main` 函数调用 `longest` 函数来寻找两个字符串 slice 中较长的一个

注意这个函数获取作为引用的字符串 slice，而不是字符串，因为我们不希望 `longest` 函数获取参数的所有权。参考之前第四章中的"字符串 slice 作为参数"部分中更多关于为什么示例 10-19 的参数正符合我们期望的讨论。

如果尝试像示例 10-20 中那样实现 `longest` 函数，它并不能编译：

文件名：src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```



示例 10-20：一个 `longest` 函数的实现，它返回两个字符串 slice 中较长者，现在还不能编译

相应地会出现如下有关生命周期的错误：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
```

```
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

提示文本揭示了返回值需要一个泛型生命周期参数，因为 Rust 并不知道将要返回的引用是指向 `x` 或 `y`。事实上我们也不知道，因为函数体中 `if` 块返回一个 `x` 的引用而 `else` 块返回一个 `y` 的引用！

当我们定义这个函数的时候，并不知道传递给函数的具体值，所以也不知道到底是 `if` 还是 `else` 会被执行。我们也不知道传入的引用的具体生命周期，所以也就不能像示例 10-17 和 10-18 那样通过观察作用域来确定返回的引用是否总是有效。借用检查器自身同样也无法确定，因为它不知道 `x` 和 `y` 的生命周期是如何与返回值的生命周期相关联的。为了修复这个错误，我们将增加泛型生命周期参数来定义引用间的关系以便借用检查器可以进行分析。

生命周期注解语法

生命周期注解并不改变任何引用的生命周期的长短。相反它们描述了多个引用生命周期相互的关系，而不影响其生命周期。与当函数签名中指定了泛型类型参数后就可以接受任何类型一样，当指定了泛型生命周期后函数也能接受任何生命周期的引用。

生命周期注解有着一个不太常见的语法：生命周期参数名称必须以撇号（`'`）开头，其名称通常全是小写，类似于泛型其名称非常短。大多数人使用 `'a` 作为第一个生命周期注解。生命周期参数注解位于引用的 `&` 之后，并有一个空格来将引用类型与生命周期注解分隔开。

这里有一些例子：我们有一个没有生命周期参数的 `i32` 的引用，一个有叫做 `'a` 的生命周期参数的 `i32` 的引用，和一个生命周期也是 `'a` 的 `i32` 的可变引用：

```
&i32        // 引用
&'a i32     // 带有显式生命周期的引用
&'a mut i32 // 带有显式生命周期的可变引用
```

单个的生命周期注解本身没有多少意义，因为生命周期注解告诉 Rust 多个引用的泛型生命周期参数如何相互联系的。让我们在 `longest` 函数的上下文中理解生命周期注解如何相互联系。

函数签名中的生命周期注解

为了在函数签名中使用生命周期注解，需要在函数名和参数列表间的尖括号中声明泛型生命周期（*lifetime*）参数，就像泛型类型（*type*）参数一样。

我们希望函数签名表达如下限制：也就是这两个参数和返回的引用存活的一样久。（两个）参数和返回的引用的生命周期是相关的。就像示例 10-21 中在每个引用中都加上了 `'a` 那样。

文件名：src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

示例 10-21：`longest` 函数定义指定了签名中所有的引用必须有相同的生命周期 `'a`

这段代码能够编译并会产生我们希望得到的示例 10-19 中的 `main` 函数的结果。

现在函数签名表明对于某些生命周期 `'a`，函数会获取两个参数，它们都是与生命周期 `'a` 存在的至少一样长的字符串 slice。函数会返回一个同样也与生命周期 `'a` 存在的至少一样长的字符串 slice。它的实际含义是 `longest` 函数返回的引用的生命周期与函数参数所引用的值的生命周期的较小者一致。这些关系就是我们希望 Rust 分析代码时所使用的。

记住通过在函数签名中指定生命周期参数时，我们并没有改变任何传入值或返回值的生命周期，而是指出任何不满足这个约束条件的值都将被借用检查器拒绝。注意 `longest` 函数并不需要知道 `x` 和 `y` 具体会存在多久，而只需要知道有某个可以被 `'a` 替代的作用域将会满足这个签名。

当在函数中使用生命周期注解时，这些注解出现在函数签名中，而不存在于函数体中的任何代码中。生命周期注解成为了函数约定的一部分，非常像签名中的类型。让函数签名包含生命周期约定意味着 Rust 编译器的工作变得更简单了。如果函数注解有误或者调用方法不对，编译器错误可以更准确地指出代码和限制的部分。如果不这么做的话，Rust 编译会对我们期望的生命周期关系做更多的推断，这样编译器可能只能指出离出问题地方很多步之外的代码。

当具体的引用被传递给 `longest` 时，被 `'a` 所替代的具体生命周期是 `x` 的作用域与 `y` 的作用域相重叠的那一部分。换一种说法就是泛型生命周期 `'a` 的具体生命周期等同于 `x` 和 `y` 的生命周期中较小的那一个。因为我们用相同的生命周期参数 `'a` 标注了返回的引用值，所以返回的引用值就能保证在 `x` 和 `y` 中较短的那个生命周期结束之前保持有效。

让我们看看如何通过传递拥有不同具体生命周期的引用来限制 `longest` 函数的使用。示例 10-22 是一个很直观的例子。

文件名：src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {result}");
    }
}
```

示例 10-22：通过拥有不同的具体生命周期的 `String` 值调用 `longest` 函数

在这个例子中，`string1` 直到外部作用域结束都是有效的，`string2` 则在内部作用域中是有效的，而 `result` 则引用了一些直到内部作用域结束都是有效的值。借用检查器认可这些代码；它能够编译和运行，并打印出 `The longest string is long string is long`。

接下来，让我们尝试另外一个例子，该例子揭示了 `result` 的引用的生命周期必须是两个参数中较短的那个。以下代码将 `result` 变量的声明移动出内部作用域，但是将 `result` 和 `string2` 变量的赋值语句一同留在内部作用域中。接着，使用了变量 `result` 的 `println!` 也被移动到内部作用域之外。注意示例 10-23 中的代码不能通过编译：

文件名：src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
```



```
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {result}");
}
```

示例 10-23：尝试在 `string2` 离开作用域之后使用 `result`

如果尝试编译这段代码会出现如下错误：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
5 |         let string2 = String::from("xyz");
  |             ------- binding `string2` declared here
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^ borrowed value does not
live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {result}");
  |                                     -------- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

错误表明为了保证 `println!` 中的 `result` 是有效的，`string2` 需要直到外部作用域结束都是有效的。Rust 知道这些是因为（`longest`）函数的参数和返回值都使用了相同的生命周期参数 `'a`。

作为人类，我们可以直观地发现 `string1` 比 `string2` 更长，因此 `result` 会包含指向 `string1` 的引用。因为 `string1` 尚未离开作用域，对于 `println!` 来说 `string1` 的引用仍然是有效的。然而，编译器并不能识别出这种情况。我们通过生命周期参数告诉 Rust 的是： `longest` 函数返回的引用的生命周期应该与传入参数的生命周期中较短那个保持一致。因此，借用检查器不允许示例 10-23 中的代码，因为它可能会存在无效的引用。

请尝试更多采用不同的值和不同生命周期的引用作为 `longest` 函数的参数和返回值的实验。并在开始编译前猜想你的实验能否通过借用检查器，接着编译一下看看你的理解是否正确！

深入理解生命周期

指定生命周期参数的正确方式依赖函数实现的具体功能。例如，如果将 `longest` 函数的实现修改为总是返回第一个参数而不是最长的字符串 slice，就不需要为参数 `y` 指定一个生命周期。如下代码将能够编译：

文件名：src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

我们为参数 `x` 和返回值指定了生命周期参数 `'a`，不过没有为参数 `y` 指定，因为 `y` 的生命周期与参数 `x` 和返回值的生命周期没有任何关系。

当从函数返回一个引用，返回值的生命周期参数需要与一个参数的生命周期参数相匹配。如果返回的引用**没有**指向任何一个参数，那么唯一的可能就是它指向一个函数内部创建的值。然而它将会是一个悬垂引用，因为它将会在函数结束时离开作用域。尝试考虑这个并不能编译的 `longest` 函数实现：

文件名：src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```



即便我们为返回值指定了生命周期参数 `'a`，这个实现却编译失败了，因为返回值的生命周期与参数完全没有关联。这里是会出现的错误信息：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return value referencing local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ------^^^^^^^^^^
   |     |
   |     returns a value referencing data owned by the current function
   |     `result` is borrowed here

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` (bin "chapter10") due to 1 previous error
```

出现的问题是 `result` 在 `longest` 函数的结尾将离开作用域并被清理，而我们尝试从函数返回一个 `result` 的引用。无法指定生命周期参数来改变悬垂引用，而且 Rust 也不允许我们创建一个悬垂引用。在这种情况，最好的解决方案是返回一个有所有权的数据类型而不是一个引用，这样函数调用者就需要负责清理这个值了。

综上，生命周期语法是用于将函数的多个参数与其返回值的生命周期进行关联的。一旦它们形成了某种关联，Rust 就有了足够的信息来允许内存安全的操作并阻止会产生悬垂指针亦或是违反内存安全的行为。

结构体定义中的生命周期注解

目前为止，我们定义的结构体全都包含拥有所有权的类型。也可以定义包含引用的结构体，不过这需要为结构体定义中的每一个引用添加生命周期注解。示例 10-24 中有一个存放了一个字符串 slice 的结构体 `ImportantExcerpt`。

文件名：src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}
```

```
fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

示例 10-24：一个存放引用的结构体，所以其定义需要生命周期注解

这个结构体只有一个字段 `part`，它存放了一个字符串 slice，这是一个引用。类似于泛型参数类型，必须在结构体名称后面的尖括号中声明泛型生命周期参数，以便在结构体定义中使用生命周期参数。这个注解意味着 `ImportantExcerpt` 的实例不能比其 `part` 字段中的引用存在的更久。

这里的 `main` 函数创建了一个 `ImportantExcerpt` 的实例，它存放了变量 `novel` 所拥有的 `String` 的第一个句子的引用。`novel` 的数据在 `ImportantExcerpt` 实例创建之前就存在。另外，直到 `ImportantExcerpt` 离开作用域之后 `novel` 都不会离开作用域，所以 `ImportantExcerpt` 实例中的引用是有效的。

生命周期省略（Lifetime Elision）

现在我们已经知道了每一个引用都有一个生命周期，而且我们需要为那些使用了引用的函数或结构体指定生命周期。然而，第四章的示例 4-9 中有一个函数，再次展示为如示例 10-25 所示，它没有生命周期注解却能编译成功：

文件名：src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

示例 10-25：示例 4-9 定义了一个没有使用生命周期注解的函数，即便其参数和返回值都是引用

这个函数没有生命周期注解却能编译是由于一些历史原因：在早期版本（pre-1.0）的 Rust 中，这的确是不能编译的。每一个引用都必须有明确的生命周期。那时的函数签名将会写成这样：

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

在编写了很多 Rust 代码后，Rust 团队发现在特定情况下 Rust 程序员们总是重复地编写一模一样的生命周期注解。这些场景是可预测的并且遵循几个明确的模式。接着 Rust 团队就把这

些模式编码进了 Rust 编译器中，如此借用检查器在这些情况下就能推断出生命周期而不再强制程序员显式的增加注解。

这里我们提到一些 Rust 的历史是因为更多的明确的模式被合并和添加到编译器中是完全可能的。未来可能只需要更少的生命周期注解。

被编码进 Rust 引用分析的模式被称为 **生命周期省略规则**（*lifetime elision rules*）。这并不是需要程序员遵守的规则；这些规则是一系列特定的场景，此时编译器会考虑，如果代码符合这些场景，就无需明确指定生命周期。

省略规则并不提供完整的推断。如果 Rust 在明确遵守这些规则的前提下变量的生命周期仍然是模棱两可的话，它不会猜测剩余引用的生命周期应该是什么。编译器会在可以通过增加生命周期注解来解决错误问题的地方给出一个错误提示，而不是进行推断或猜测。

函数或方法的参数的生命周期被称为 **输入生命周期**（*input lifetimes*），而返回值的生命周期被称为 **输出生命周期**（*output lifetimes*）。

编译器采用三条规则来判断引用何时不需要明确的注解。第一条规则适用于输入生命周期，后两条规则适用于输出生命周期。如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误。这些规则适用于 `fn` 定义，以及 `impl` 块。

第一条规则是编译器为每一个引用参数都分配一个生命周期参数。换句话说就是，函数有一个引用参数的就有一个生命周期参数：`fn foo<'a>(x: &'a i32)`，有两个引用参数的函数就有两个不同的生命周期参数，`fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`，依此类推。

第二条规则是如果只有一个输入生命周期参数，那么将它赋予给所有输出生命周期参数：`fn foo<'a>(x: &'a i32) -> &'a i32`。

第三条规则是如果方法有多个输入生命周期参数并且其中一个参数是 `&self` 或 `&mut self`，说明这是个方法，那么所有输出生命周期参数被赋予 `self` 的生命周期。第三条规则使得方法更容易读写，因为只需更少的符号。

假设我们自己就是编译器。并应用这些规则来计算示例 10-25 中 `first_word` 函数签名中的引用的生命周期。开始时签名中的引用并没有关联任何生命周期：

```
fn first_word(s: &str) -> &str {
```

接着编译器应用第一条规则，也就是每个引用参数都有其自己的生命周期。我们像往常一样称之为 `'a`，所以现在签名看起来像这样：

```
fn first_word<'a>(s: &'a str) -> &str {
```

对于第二条规则，因为这里正好只有一个输入生命周期参数所以是适用的。第二条规则表明输入参数的生命周期将被赋予输出生命周期参数，所以现在签名看起来像这样：

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

现在这个函数签名中的所有引用都有了生命周期，如此编译器可以继续它的分析而无须程序员标记这个函数签名中的生命周期。

让我们再看看另一个例子，这次我们从示例 10-20 中没有生命周期参数的 `longest` 函数开始：

```
fn longest(x: &str, y: &str) -> &str {
```

再次假设我们自己就是编译器并应用第一条规则：每个引用参数都有其自己的生命周期。这次有两个参数，所以就有两个（不同的）生命周期：

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

再来应用第二条规则，因为函数存在多个输入生命周期，它并不适用于这种情况。再来看第三条规则，它同样也不适用，这是因为没有 `self` 参数。应用了三个规则之后编译器还没有计算出返回值类型的生命周期。这就是在编译示例 10-20 的代码时会出现错误的原因：编译器使用所有已知的生命周期省略规则，仍不能计算出签名中所有引用的生命周期。

因为第三条规则真正能够适用的就只有方法签名，现在就让我们看看那种情况中的生命周期，并看看为什么这条规则意味着我们经常不需要在方法签名中标注生命周期。

方法定义中的生命周期注解

当为带有生命周期的结构体实现方法时，其语法依然类似示例 10-11 中展示的泛型类型参数的语法。我们在哪里声明和使用生命周期参数，取决于它们是与结构体字段相关还是与方法参数和返回值相关。

（实现方法时）结构体字段的生命周期必须总是在 `impl` 关键字之后声明并在结构体名称之后被使用，因为这些生命周期是结构体类型的一部分。

`impl` 块里的方法签名中，引用可能与结构体字段中的引用相关联，也可能是独立的。另外，生命周期省略规则也经常让我们无需在方法签名中使用生命周期注解。让我们看看一些使用示例 10-24 中定义的结构体 `ImportantExcerpt` 的例子。

首先，这里有一个方法 `level`。其唯一的参数是 `self` 的引用，而且返回值只是一个 `i32`，并不引用任何值：

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

`impl` 之后和类型名称之后的生命周期参数是必要的，不过因为第一条生命周期规则我们并不必须标注 `self` 引用的生命周期。

这里是一个适用于第三条生命周期省略规则的例子：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {announcement}");
        self.part
    }
}
```

这里有两个输入生命周期，所以 Rust 应用第一条生命周期省略规则并给予 `&self` 和 `announcement` 它们各自的生命周期。接着，因为其中一个参数是 `&self`，返回值类型被赋予了 `&self` 的生命周期，这样所有的生命周期都被计算出来了。

静态生命周期

这里有一种特殊的生命周期值得讨论：`'static`，其生命周期**能够**存活于整个程序期间。所有的字符串字面值都拥有 `'static` 生命周期，我们也可以选择像下面这样标注出来：

```
let s: &'static str = "I have a static lifetime.";
```

这个字符串的文本被直接储存在程序的二进制文件中而这个文件总是可用的。因此所有的字符串字面值都是 `'static` 的。

你可能在错误信息的帮助文本中见过使用 `'static` 生命周期的建议，不过将引用指定为 `'static` 之前，思考一下这个引用是否真的在整个程序的生命周期里都有效，以及你是否希望它存在得这么久。大部分情况中，推荐 `'static` 生命周期的错误信息都是尝试创建一个悬垂引用或者可用的生命周期不匹配的结果。在这种情况下的解决方案是修复这些问题而不是指定一个 `'static` 的生命周期。

结合泛型类型参数、trait bounds 和生命周期

让我们简要的看一下在同一函数中指定泛型类型参数、trait bounds 和生命周期的语法！

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {ann}");
    if x.len() > y.len() { x } else { y }
}
```

这个是示例 10-21 中那个返回两个字符串 slice 中较长者的 `longest` 函数，不过带有一个额外的参数 `ann`。`ann` 的类型是泛型 `T`，它可以被放入任何实现了 `where` 从句中指定的 `Display` trait 的类型。这个额外的参数会使用 `{}` 打印，这也就是为什么 `Display` trait bound 是必须的。因为生命周期也是泛型，所以生命周期参数 `'a` 和泛型类型参数 `T` 都位于函数名后的同一尖括号列表中。

总结

这一章介绍了很多的内容！现在你知道了泛型类型参数、trait 和 trait bounds 以及泛型生命周期类型，你已经准备好编写既不重复又能适用于多种场景的代码了。泛型类型参数意味着代码可以适用于不同的类型。trait 和 trait bounds 保证了即使类型是泛型的，这些类型也会拥有所需要的行为。由生命周期注解所指定的引用生命周期之间的关系保证了这些灵活多变的代码不会出现悬垂引用。而所有的这一切发生在编译时所以不会影响运行时效率！

你可能不会相信，这个话题还有更多需要学习的内容：第十八章会讨论 trait 对象，这是另一种使用 trait 的方式。还有更多更复杂的涉及生命周期注解的场景，只有在非常高级的情况下才会需要它们；对于这些内容，请阅读 Rust Reference。不过接下来，让我们聊聊如何在 Rust 中编写测试，来确保代码的所有功能能像我们希望的那样工作！

编写自动化测试

Edsger W. Dijkstra 在其 1972 年的文章《谦卑的程序员》（“The Humble Programmer”）中说到 “软件测试是证明 bug 存在的有效方法，而证明其不存在时则显得令人绝望的不足。”（“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”）这并不意味着我们不应尽可能地测试软件！

程序的正确性意味着代码如我们期望的那样运行。Rust 是一个相当注重正确性的编程语言，不过正确性是一个难以证明的复杂主题。Rust 的类型系统在此问题上下了很大的功夫，不过类型系统不可能捕获所有问题。为此，Rust 包含了编写自动化软件测试的功能支持。

假设我们可以编写一个叫做 `add_two` 的将传递给它的值加二的函数。它的签名有一个整型参数并返回一个整型值。当实现和编译这个函数时，Rust 会进行所有目前我们已经见过的类型检查和借用检查，例如，这些检查会确保我们不会传递 `String` 或无效的引用给这个函数。Rust 所**不能**检查的是这个函数是否会准确的完成我们期望的工作：返回参数加二后的值，而不是比如说参数加 10 或减 50 的值！这正是测试的用武之地。

我们可以编写测试断言，比如说，当传递 `3` 给 `add_two` 函数时，返回值是 `5`。无论何时对代码进行修改，都可以运行测试来确保任何现存的正确行为没有被改变。

测试是一项复杂的技能：虽然不能在一个章节的篇幅中介绍如何编写好的测试的每个细节，但我们还是会讨论 Rust 测试功能的机制。我们会讲到编写测试时会用到的注解和宏，运行测试的默认行为和选项，以及如何将测试组织成单元测试和集成测试。

如何编写测试

Rust 中的测试函数是用来验证非测试代码是否是按照期望的方式运行的。测试函数体通常执行如下三种操作：

- 设置任何所需的数据或状态
- 运行需要测试的代码
- 断言其结果是我们所期望的

让我们看看 Rust 提供的专门用来编写测试的功能：`test` 属性、一些宏和 `should_panic` 属性。

测试函数剖析

作为最简单例子，Rust 中的测试就是一个带有 `test` 属性注解的函数。属性（attribute）是关于 Rust 代码片段的元数据；第五章中结构体中用到的 `derive` 属性就是一个例子。为了将一个函数变成测试函数，需要在 `fn` 行之前加上 `#[test]`。当使用 `cargo test` 命令运行测试时，Rust 会构建一个测试执行程序用来调用被标注的函数，并报告每一个测试是通过还是失败。

每次使用 Cargo 新建一个库项目时，它会自动为我们生成一个测试模块和一个测试函数。这个模块提供了一个编写测试的模板，为此每次开始新项目时不必去查找测试函数的具体结构和语法了。当然你也可以额外增加任意多的测试函数以及测试模块！

在实际编写测试代码之前，让我们先通过尝试那些自动生成的测试模版来探索测试是如何工作的。接着，我们会写一些真正的测试，调用我们编写的代码并断言它们的行为的正确性。

让我们创建一个新的库项目 `adder`，它会将两个数字相加：

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

`adder` 库中 *src/lib.rs* 的内容应该看起来如示例 11-1 所示：

文件名：src/lib.rs

```
pub fn add(left: u64, right: u64) -> u64 {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

示例 11-1：由 `cargo new` 自动生成的测试模块和函数

文件以一个示例 `add` 函数开头，这样我们就有东西可以测试。

现在让我们只关注 `it_works` 函数本身。注意 `fn` 行之前的 `#[test]`：这个属性表明这是一个测试函数，这样测试执行者就知道将其作为测试处理。`tests` 模块中也可以有非测试的函数来帮助我们建立通用场景或进行常见操作，必须每次都标明哪些函数是测试。

示例函数体通过使用 `assert_eq!` 宏来断言 `result`（其中包含 2 加 2 的结果）等于 4。这个断言示例展示了典型测试的格式。接下来运行就可以看到测试通过。

`cargo test` 命令会运行项目中所有的测试，如示例 11-2 所示：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests src/lib.rs (target/debug/deps/adder-01ad14159ff659ab)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

示例 11-2：运行自动生成测试的输出

Cargo 编译并运行了测试。可以看到 `running 1 test` 这一行。下一行显示了生成的测试函数的名称，`tests::it_works`，以及测试的运行结果，`ok`。接着可以看到全体测试运行结果的摘要：`test result: ok.` 意味着所有测试都通过了。`1 passed; 0 failed` 表示通过或失败的测试数量。

可以将一个测试标记为忽略以便在特定情况下它就不会运行；本章之后的“除非特别指定否则忽略某些测试”部分会介绍它。因为之前我们并没有将任何测试标记为忽略，所以摘要中会显示 `0 ignored`。

`0 measured` 统计是针对性能测试的。性能测试（benchmark tests）在编写本书时，仍只能用于 Rust 开发版（nightly Rust）。请查看 性能测试的文档 了解更多。

我们可以将参数传递给 `cargo test` 命令，以便只运行名称与字符串匹配的测试；这就是所谓的**过滤**（*filtering*），我们会在 “通过名称运行部分测试” 讨论这一点。这里我们没有过滤需要运行的测试，所以摘要中会显示`0 filtered out`。

测试输出中的以 `Doc-tests adder` 开头的这一部分是所有文档测试的结果。我们现在并没有任何文档测试，不过 Rust 会编译任何在 API 文档中的代码示例。这个功能帮助我们使文档和代码保持同步！在第十四章的 “文档注释作为测试” 部分会讲到如何编写文档测试。现在我们将忽略 `Doc-tests` 部分的输出。

让我们开始自定义测试来满足我们的需求。首先给 `it_works` 函数起个不同的名字，比如 `exploration`，像这样：

文件名：src/lib.rs

```
pub fn add(left: u64, right: u64) -> u64 {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn exploration() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

并再次运行 `cargo test`。现在输出中将出现 `exploration` 而不是 `it_works`：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

现在让我们增加另一个测试，不过这一次是一个会失败的测试！当测试函数中出现 panic 时测试就失败了。每一个测试都在一个新线程中运行，当主线程发现测试线程异常了，就将对应测试标记为失败。第九章讲到了最简单的造成 panic 的方法：调用 `panic!` 宏。写入新测试 `another` 后，*src/lib.rs* 现在看起来如示例 11-3 所示：

文件名：src/lib.rs

```
pub fn add(left: u64, right: u64) -> u64 {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn exploration() {
        let result = add(2, 2);
        assert_eq!(result, 4);
```

```
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```



示例 11-3：增加第二个因调用了 `panic!` 而失败的测试

再次 `cargo test` 运行测试。输出应该看起来像示例 11-4，它表明 `exploration` 测试通过了而 `another` 失败了：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----

thread 'tests::another' panicked at src/lib.rs:17:9:
Make this test fail
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

示例 11-4：一个测试通过和一个测试失败的测试结果

`test tests::another` 这一行是 `FAILED` 而不是 `ok` 了。在单独测试结果和摘要之间多了两个新的部分：第一个部分显示了测试失败的详细原因。在这个例子中，我们看到 `another` 因为在 *src/lib.rs* 的第 10 行 `panicked at 'Make this test fail'` 而失败的详细信息。下一部分列出了所有失败的测试，这在有很多测试和很多失败测试的详细输出时很有帮助。我们可以通过使用失败测试的名称来只运行这个测试，以便调试；下一部分 “控制测试如何运行” 会讲到更多运行测试的方法。

最后是摘要行：总体上讲，测试结果是 `FAILED`。有一个测试通过和一个测试失败。

现在我们见过不同场景中测试结果是什么样子的了，再来看看除 `panic!` 之外的一些在测试中有帮助的宏吧。

使用 `assert!` 宏来检查结果

`assert!` 宏由标准库提供，在希望确保测试中一些条件为 `true` 时非常有用。需要向 `assert!` 宏提供一个求值为布尔值的参数。如果值是 `true`，`assert!` 什么也不做，同时测试会通过。如果值为 `false`，`assert!` 调用 `panic!` 宏，这会导致测试失败。`assert!` 宏帮助我们检查代码是否以期望的方式运行。

回忆一下第五章中，示例 5-15 中有一个 `Rectangle` 结构体和一个 `can_hold` 方法，在示例 11-5 中再次使用它们。将它们放进 *src/lib.rs* 并使用 `assert!` 宏编写一些测试。

文件名：src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

示例 11-5：第五章中 `Rectangle` 结构体和其 `can_hold` 方法

`can_hold` 方法返回一个布尔值，这意味着它完美符合 `assert!` 宏的使用场景。在示例 11-6 中，让我们编写一个 `can_hold` 方法的测试来作为练习，这里创建一个宽为 8 高为 7 的 `Rectangle` 实例，并假设它可以放得下另一个宽为 5 高为 1 的 `Rectangle` 实例：

文件名：src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

示例 11-6：一个 `can_hold` 的测试，检查一个较大的矩形确实能放得下一个较小的矩形

注意在 `tests` 模块中新增加了一行：`use super::*;`。`tests` 是一个普通的模块，它遵循第七章 “路径用于引用模块树中的项” 部分介绍的可见性规则。因为 `tests` 模块是一个内部模块，

要测试外部模块中的代码，需要将其引入到内部模块的作用域中。这里选择使用 glob 全局导入，以便在 `tests` 模块中使用所有在外部模块定义的内容。

我们将测试命名为 `larger_can_hold_smaller`，并创建所需的两个 `Rectangle` 实例。接着调用 `assert!` 宏并传递 `larger.can_hold(&smaller)` 调用的结果作为参数。这个表达式预期会返回 `true`，所以测试应该通过。让我们拭目以待！

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

它确实通过了！再来增加另一个测试，这一回断言一个更小的矩形不能放下一个更大的矩形：

文件名：src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

因为这里 `can_hold` 函数的正确结果是 `false`，我们需要将这个结果取反后传递给 `assert!` 宏。因此 `can_hold` 返回 `false` 时测试就会通过：

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

两个测试通过了！现在让我们看看如果引入一个 bug 的话测试结果会发生什么。将 `can_hold` 方法中比较宽度时本应使用大于号的地方改成小于号：



```
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```

现在运行测试会产生以下结果：

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----

thread 'tests::larger_can_hold_smaller' panicked at src/lib.rs:28:9:
assertion failed: larger.can_hold(&smaller)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

```
error: test failed, to rerun pass `--lib`
```

我们的测试捕获了 bug！因为 `larger.width` 是 8 而 `smaller.width` 是 5，`can_hold` 中的宽度比较现在因为 8 不小于 5 而返回 `false`：8 并不小于 5。

使用 `assert_eq!` 和 `assert_ne!` 宏测试相等

测试功能的一个常用方法是将需要测试代码的值与期望值做比较，并检查是否相等。可以通过向 `assert!` 宏传递一个使用 `==` 运算符的表达式来做到。不过这个操作实在是太常见了，以至于标准库提供了一对宏来更方便的处理这些操作 —— `assert_eq!` 和 `assert_ne!`。这两个宏分别比较两个值是相等还是不相等。当断言失败时它们也会打印出这两个值具体是什么，以便于观察测试**为什么**失败，而 `assert!` 只会打印出它从 `==` 表达式中得到了 `false` 值，而不是打印导致 `false` 的具体值。

示例 11-7 中，让我们编写一个对其参数加二并返回结果的函数 `add_two`。接着使用 `assert_eq!` 宏测试这个函数。

文件名：src/lib.rs

```
pub fn add_two(a: usize) -> usize {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        let result = add_two(2);
        assert_eq!(result, 4);
    }
}
```

示例 11-7：使用 `assert_eq!` 宏测试 `add_two` 函数

测试通过了！

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests
```

```
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

我们创建一个名为 `result` 的变量，用于保存调用 `add_two(2)` 的结果。然后我们将 `result` 和 `4` 作为参数传递给 `assert_eq!`。测试中的这一行 `test tests::it_adds_two ... ok` 中 `ok` 表明测试通过！

在代码中引入一个 bug 来看看使用 `assert_eq!` 的测试失败是什么样的。修改 `add_two` 函数的实现使其加 `3`：

```
pub fn add_two(a: usize) -> usize {
    a + 3
}
```



再次运行测试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----

thread 'tests::it_adds_two' panicked at src/lib.rs:12:9:
assertion `left == right` failed
  left: 5
 right: 4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

测试捕获到了 bug！`it_adds_two` 测试失败，错误信息告诉我们断言失败了，它告诉我们 `assertion `left == right` failed` 以及 `left` 和 `right` 的值是什么。这个错误信息有助于我们开始调试：它说 `assert_eq!` 的 `left` 参数（也就是 `add_two(2)` 的结果）是 `5`，而 `right` 参数是 `4`。可以想象当有很多测试在运行时这些信息是多么的有用。

需要注意的是，在一些语言和测试框架中，断言两个值相等的函数的参数被称为 `expected` 和 `actual`，而且指定参数的顺序非常重要。然而在 Rust 中，它们则叫做 `left` 和 `right`，同时指定期望的值和被测试代码产生的值的顺序并不重要。这个测试中的断言也可以写成

`assert_eq!(add_two(2), result)`，这时失败信息仍同样是 `assertion failed: `(left == right)``。

`assert_ne!` 宏在传递给它的两个值不相等时通过，而在相等时失败。当我们不确定值**会**是什么，不过能确定值绝对**不会**是什么的时候，这个宏最有用处。例如，如果一个函数保证会以某种方式改变其输入，不过这种改变方式是由运行测试时是星期几来决定的，这时最好的断言可能就是函数的输出不等于其输入。

`assert_eq!` 和 `assert_ne!` 宏在底层分别使用了 `==` 和 `!=`。当断言失败时，这些宏会使用调试格式打印出其参数，这意味着被比较的值必须实现了 `PartialEq` 和 `Debug` trait。所有的基本类型和大部分标准库类型都实现了这些 trait。对于自定义的结构体和枚举，需要实现 `PartialEq` 才能断言它们的值是否相等。需要实现 `Debug` 才能在断言失败时打印它们的值。因为这两个 trait 都是派生 trait，如第五章示例 5-12 所提到的，通常可以直接在结构体或枚举上添加 `#[derive(PartialEq, Debug)]` 注解。附录 C “可派生 trait” 中有更多关于这些和其他派生 trait 的详细信息。

自定义失败信息

你也可以向 `assert!`、`assert_eq!` 和 `assert_ne!` 宏传递一个可选的失败信息参数，可以在测试失败时将自定义失败信息一同打印出来。任何在 `assert!` 的一个必需参数和 `assert_eq!` 和 `assert_ne!` 的两个必需参数之后指定的参数都会传递给 `format!` 宏（在第八章的 “使用 `+` 运算符或 `format!` 宏拼接字符串” 部分讨论过），所以可以传递一个包含 `{}` 占位符的格式字符串和需要放入占位符的值。自定义信息有助于记录断言的意义；当测试失败时就能更好的理解代码出了什么问题。

例如，比如说有一个根据人名进行问候的函数，而我们希望测试将传递给函数的人名显示在输出中：

文件名：src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {name}!")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

这个程序的需求还没有被确定，因此问候文本开头的 `Hello` 文本很可能会改变。然而我们并不想在需求改变时不得不更新测试，所以相比检查 `greeting` 函数返回的确切值，我们将仅仅断言输出的文本中包含输入参数。

让我们通过将 `greeting` 改为不包含 `name` 在代码中引入一个 bug 来测试失败时是怎样的：

```
```

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```



运行测试会产生：

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----

thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
assertion failed: result.contains("Carol")
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

结果仅仅告诉了我们断言失败了和失败的行号。一个更有用的失败信息应该打印出 `greeting` 函数的值。让我们为测试函数增加一个自定义失败信息参数：带占位符的格式字符串，以及 `greeting` 函数的值：

```
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{result}`"
        );
    }
```

现在如果再次运行测试，将会看到更有价值的信息：

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED
```

```
failures:

---- tests::greeting_contains_name stdout ----

thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
Greeting did not contain name, value was `Hello!`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

可以在测试输出中看到所取得的确切的值，这会帮助我们调试真正发生了什么，而不是期望发生什么。

使用 `should_panic` 检查 panic

除了检查返回值之外，检查代码是否按照期望处理错误也是很重要的。例如，考虑第九章示例 9-13 创建的 `Guess` 类型。其他使用 `Guess` 的代码都是基于 `Guess` 实例仅有的值范围在 1 到 100 的前提。可以编写一个测试来确保创建一个超出范围的值的 `Guess` 实例会 panic。

可以通过对函数增加另一个属性 `should_panic` 来实现这些。这个属性在函数中的代码 panic 时会通过，而在其中的代码没有 panic 时失败。

示例 11-8 展示了一个测试，检查 `Guess::new` 在错误条件下是否如我们所料那样。

文件名：src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {value}.");
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
```

```
        Guess::new(200);
    }
}
```

示例 11-8：测试会造成 `panic!` 的条件

`#[should_panic]` 属性位于 `#[test]` 之后，对应的测试函数之前。让我们看看测试通过时它是什么样子：

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/
guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

看起来不错！现在在代码中引入 bug，移除 `new` 函数在值大于 100 时会 panic 的条件：



```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {value}.");
        }

        Guess { value }
    }
}
```

如果运行示例 11-8 的测试，它会失败：

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/
guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:
```

```
---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

这回并没有得到非常有用的信息，不过一旦我们观察测试函数，会发现它标注了 `#[should_panic]`。这个错误意味着代码中测试函数 `Guess::new(200)` 并没有产生 panic。

然而 `should_panic` 测试结果可能会非常含糊不清。`should_panic` 甚至在一些不是我们期望的原因而导致 panic 时也会通过。为了使 `should_panic` 测试结果更精确，我们可以给 `should_panic` 属性增加一个可选的 `expected` 参数。测试工具会确保错误信息中包含其提供的文本。例如，考虑示例 11-9 中修改过的 `Guess`，这里 `new` 函数根据其值是过大还或者过小而提供不同的 panic 信息：

文件名：src/lib.rs

```
// --snip--

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {value}."
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {value}."
            );
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

示例 11-9：一个错误信息包含特定子字符串的 `panic!` 条件的测试

这个测试会通过，因为 `should_panic` 属性中 `expected` 参数提供的值是 `Guess::new` 函数 panic 信息的子串。我们可以指定期望的整个 panic 信息，在这个例子中是 `Guess value must be less than or equal to 100, got 200` 。信息的选择取决于 panic 信息有多独特或动态，和你希望测试有多准确。在这个例子中，错误信息的子字符串足以确保函数在 `else if value > 100` 的情况下运行。

为了观察带有 `expected` 信息的 `should_panic` 测试失败时会发生什么，让我们再次引入一个 bug，将 `if value < 1` 和 `else if value > 100` 的代码块对换：



```
        if value < 1 {
            panic!(
                "Guess value must be less than or equal to 100, got {value}."
            );
        } else if value > 100 {
            panic!(
                "Guess value must be greater than or equal to 1, got {value}."
            );
        }
```

这一次运行 `should_panic` 测试，它会失败：

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/
guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----

thread 'tests::greater_than_100' panicked at src/lib.rs:12:13:
Guess value must be greater than or equal to 1, got 200.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got 200."`,
 expected substring: `"less than or equal to 100"`

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

失败信息表明测试确实如期望 panic 了，不过 panic 信息中并没有包含期望的信息 `less than or equal to 100`。而我们得到的 panic 信息是 `Guess value must be greater than or equal to 1, got 200.`。这样就可以开始寻找 bug 在哪了！

在测试中使用 `Result<T, E>`

目前为止，我们编写的测试在失败时都会 panic。我们也可以使用 `Result<T, E>` 编写测试！这是一个延伸自示例 11-1 的测试，使用 `Result<T, E>` 重写，并在失败时返回 `Err` 而非 panic：

```
    #[test]
    fn it_works() -> Result<(), String> {
        let result = add(2, 2);

        if result == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
```

现在 `it_works` 函数的返回值类型为 `Result<(), String>`。在函数体中，不同于调用 `assert_eq!` 宏，而是在测试通过时返回 `Ok(())`，在测试失败时返回带有 `String` 的 `Err`。

这样编写测试来返回 `Result<T, E>` 就可以在函数体中使用问号运算符，如此可以方便的编写任何运算符会返回 `Err` 成员的测试。

不能对这些使用 `Result<T, E>` 的测试使用 `#[should_panic]` 注解。为了断言一个操作返回 `Err` 成员，**不要**对 `Result<T, E>` 值使用问号表达式（`?`）。而是使用 `assert!(value.is_err())`。

现在你知道了几种编写测试的方法，让我们看看运行测试时会发生什么，以及可以用于 `cargo test` 的不同选项。

控制测试如何运行

就像 `cargo run` 会编译代码并运行生成的二进制文件一样，`cargo test` 在测试模式下编译代码并运行生成的测试二进制文件。`cargo test` 产生的二进制文件的默认行为是并发运行所有的测试，并截获测试运行过程中产生的输出，阻止它们被显示出来，使得阅读测试结果相关的内容变得更容易。不过可以指定命令行参数来改变 `cargo test` 的默认行为。

可以将一部分命令行参数传递给 `cargo test`，而将另外一部分传递给生成的测试二进制文件。为了分隔这两种参数，需要先列出传递给 `cargo test` 的参数，接着是分隔符 `--`，再之后是传递给测试二进制文件的参数。运行 `cargo test --help` 会提示 `cargo test` 的有关参数，而运行 `cargo test -- --help` 可以提示在分隔符之后使用的有关参数。有关这些选项的说明，请参阅 the rustc book 的 “Tests” 一节。

并行或顺序运行测试

当运行多个测试时，Rust 默认使用线程来并行运行。这意味着测试会更快地运行完毕，所以你可以更快的得到代码能否工作的反馈。因为测试是在同时运行的，你应该确保测试不能相互依赖，或依赖任何共享的状态，包括依赖共享的环境，比如当前工作目录或者环境变量。

举个例子，每一个测试都运行一些代码，假设这些代码都在硬盘上创建一个 *test-output.txt* 文件并写入一些数据。接着每一个测试都读取文件中的数据并断言这个文件包含特定的值，而这个值在每个测试中都是不同的。因为所有测试都是同时运行的，一个测试可能会在另一个测试读写文件过程中修改了文件。那么第二个测试就会失败，并不是因为代码不正确，而是因为测试并行运行时相互干扰。一个解决方案是使每一个测试读写不同的文件；另一个解决方案是一次运行一个测试。

如果你不希望测试并行运行，或者想要更加精确的控制线程的数量，可以传递 `--test-threads` 参数和希望使用线程的数量给测试二进制文件。例如：

```
$ cargo test -- --test-threads=1
```

这里将测试线程设置为 `1`，告诉程序不要使用任何并行机制。这也会比并行运行花费更多时间，不过在有共享的状态时，测试就不会潜在的相互干扰了。

显示函数输出

默认情况下，当测试通过时，Rust 的测试库会捕获打印到标准输出的所有内容。比如在测试中调用了 `println!` 而测试通过了，我们将不会在终端看到 `println!` 的输出：只会看到说明测试通过的提示行。如果测试失败了，则会看到所有标准输出和其他错误信息。

例如，示例 11-10 有一个无意义的函数，它打印出其参数的值并接着返回 10。接着还有一个会通过的测试和一个会失败的测试：

文件名：src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {a}");
    10
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(value, 10);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(value, 5);
    }
}
```



示例 11-10：一个调用了 `println!` 的函数的测试

运行 `cargo test` 将会看到这些测试的输出：

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/
silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8

thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 10
 right: 5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

注意输出中不会出现测试通过时打印的内容，即 `I got the value 4`。因为当测试通过时，这些输出会被截获。失败测试的输出 `I got the value 8` ，则出现在输出的测试摘要部分，同时也显示了测试失败的原因。

如果你希望也能看到通过的测试中打印的值，也可以在结尾加上 `--show-output` 告诉 Rust 显示成功测试的输出。

```
$ cargo test -- --show-output
```

使用 `--show-output` 参数再次运行示例 11-10 中的测试会显示如下输出：

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/
silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4


successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8

thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 10
 right: 5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

通过名称运行部分测试

有时运行整个测试集会耗费很长时间。如果你负责特定位置的代码，你可能会希望只运行与这些代码相关的测试。你可以向 `cargo test` 传递所希望运行的测试名称的参数来选择运行哪些测试。

为了展示如何运行部分测试，示例 11-11 为 `add_two` 函数创建了三个测试，我们可以选择具体运行哪一个：

文件名：src/lib.rs

```
pub fn add_two(a: usize) -> usize {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        let result = add_two(2);
        assert_eq!(result, 4);
    }

    #[test]
    fn add_three_and_two() {
        let result = add_two(3);
        assert_eq!(result, 5);
    }

    #[test]
    fn one_hundred() {
        let result = add_two(100);
        assert_eq!(result, 102);
    }
}
```

示例 11-11：不同名称的三个测试

如果没有传递任何参数就运行测试，如你所见，所有测试都会并行运行：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

运行单个测试

可以向 `cargo test` 传递任意测试的名称来只运行这个测试：

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

只有名称为 `one_hundred` 的测试被运行了；因为其余两个测试并不匹配这个名称。测试输出在末尾显示了 `2 filtered out`，表明还有 2 个测试被过滤，未被运行。

不能像这样指定多个测试名称；只有传递给 `cargo test` 的第一个值才会被使用。不过有个运行多个测试的方法。

过滤运行多个测试

我们可以指定部分测试的名称，任何名称匹配这个名称的测试会被运行。例如，因为头两个测试的名称包含 `add`，可以通过 `cargo test add` 来运行这两个测试：

```
$ cargo test add
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

这运行了所有名字中带有 `add` 的测试，也过滤掉了名为 `one_hundred` 的测试。同时注意测试所在的模块也是测试名称的一部分，所以可以通过过滤模块名来运行一个模块中的所有测试。

除非特别指定否则忽略某些测试

有时一些特定的测试执行起来是非常耗费时间的，所以在大多数运行 `cargo test` 的时候希望能排除它们。虽然可以通过参数列举出所有希望运行的测试来做到，也可以使用 `ignore` 属性来标记耗时的测试并排除它们，如下所示：

文件名：src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
```

```
    }

    #[test]
    #[ignore]
    fn expensive_test() {
        // code that takes an hour to run
    }
}
```

对于想要排除的测试，我们在 `#[test]` 之后增加了 `#[ignore]` 行。现在如果运行测试，就会发现 `it_works` 运行了，而 `expensive_test` 没有运行：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::expensive_test ... ignored
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

`expensive_test` 被列为 `ignored`，如果我们只希望运行被忽略的测试，可以使用 `cargo test -- --ignored`：

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

通过控制运行哪些测试，你可以确保 `cargo test` 的结果能够快速返回。当你需要运行 `ignored` 的测试时，可以执行 `cargo test -- --ignored`。如果你希望不管是否忽略都要运行全部测试，可以运行 `cargo test -- --include-ignored`。

测试的组织结构

本章一开始就提到，测试是一个复杂的概念，而且不同的开发者也采用不同的术语和组织。Rust 社区倾向于根据测试的两个主要分类来考虑问题：**单元测试**（*unit tests*）与**集成测试**（*integration tests*）。单元测试倾向于更小而更集中，在隔离的环境中一次测试一个模块，并且可以测试私有接口。而**集成测试**对于你的库来说则完全是外部的。它们与其他外部代码一样，通过相同的方式使用你的代码，只测试公有接口而且每个测试都有可能会测试多个模块。

为了保证你的库能够按照你的预期运行，从独立和整体的角度编写这两类测试都是非常重要的。

单元测试

单元测试的目的是在与其他部分隔离的环境中测试每一个单元的代码，以便于快速而准确地验证某个单元的代码功能是否符合预期。单元测试与它们要测试的代码共同存放在位于 *src* 目录下相同的文件中。规范是在每个文件中创建包含测试函数的 `tests` 模块，并使用 `cfg(test)` 标注模块。

测试模块和 `#[cfg(test)]`

测试模块的 `#[cfg(test)]` 注解告诉 Rust 只在执行 `cargo test` 时才编译和运行测试代码，而在运行 `cargo build` 时不这么做。这在只希望构建库的时候可以节省编译时间，并且因为它们并没有包含测试，所以能减少编译产生的文件的大小。与之对应的集成测试因为位于另一个文件夹，所以它们并不需要 `#[cfg(test)]` 注解。然而单元测试位于与源码相同的文件中，所以你需要使用 `#[cfg(test)]` 来指定它们不应该被包含进编译结果中。

回忆本章第一部分新建的 `adder` 项目，Cargo 为我们生成了如下代码：

文件名：src/lib.rs

```
pub fn add(left: u64, right: u64) -> u64 {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

上述代码就是自动生成的测试模块。`cfg` 属性代表**配置**（*configuration*），它告诉 Rust 接下来的项只有在给定特定配置选项时，才会被包含。在这种情况下，配置选项是 `test`，即 Rust 所提供的用于编译和运行测试的配置选项。通过使用 `cfg` 属性，Cargo 只会在我们主动使用 `cargo test` 运行测试时才编译测试代码。这包括测试模块中可能存在的辅助函数，以及标注为 `#[test]` 的函数。

测试私有函数

测试社区中一直存在关于是否应该对私有函数直接进行测试的论战，而在其他语言中想要测试私有函数是一件困难的，甚至是不可能的事。不过无论你坚持哪种测试意识形态，Rust 的私有性规则确实允许你测试私有函数。考虑示例 11-12 中带有私有函数 `internal_adder` 的代码。

文件名：src/lib.rs

```
pub fn add_two(a: usize) -> usize {
    internal_adder(a, 2)
}

fn internal_adder(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        let result = internal_adder(2, 2);
        assert_eq!(result, 4);
    }
}
```

示例 11-12：测试私有函数

注意 `internal_adder` 函数并没有标记为 `pub`。测试也不过是 Rust 代码，同时 `tests` 也仅仅是另一个模块。正如"路径用于引用模块树中的项" 部分所说，子模块的项可以使用其上级模块的项。在测试中，我们通过 `use super::*` 将 `tests` 模块的父模块的所有项引入了作用域，接着测试调用了 `internal_adder`。如果你并不认为应该测试私有函数，Rust 也不会强迫你这么做。

集成测试

在 Rust 中，集成测试对于你需要测试的库来说完全是外部的。同其他使用库的代码一样使用库文件，也就是说它们只能调用一部分库中的公有 API。集成测试的目的是测试库的多个部分能否一起正常工作。一些单独能正确运行的代码单元集成在一起也可能会出现问题，所以集成测试的覆盖率也是很重要的。为了创建集成测试，你需要先创建一个 *tests* 目录。

tests 目录

为了编写集成测试，需要在项目根目录创建一个 *tests* 目录，与 *src* 同级。Cargo 知道如何去寻找这个目录中的集成测试文件。接着可以随意在这个目录中创建任意多的测试文件，Cargo 会将每一个文件当作单独的 crate 来编译。

让我们来创建一个集成测试。保留示例 11-12 中 *src/lib.rs* 的代码。创建一个 *tests* 目录，新建一个文件 *tests/integration_test.rs*。目录结构应该看起来像这样：

```
adder
├── Cargo.lock
├── Cargo.toml
```

```
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

将示例 11-13 中的代码输入到 *tests/integration_test.rs* 文件中。

文件名：tests/integration_test.rs

```
use adder::add_two;

#[test]
fn it_adds_two() {
    let result = add_two(2);
    assert_eq!(result, 4);
}
```

示例 11-13：一个 `adder` crate 中函数的集成测试

因为每一个 *tests* 目录中的测试文件都是完全独立的 crate，所以需要将库引入到每个测试 crate 的作用域中。为此与单元测试不同，我们需要在文件顶部添加 `use adder::add_two;`，这在单元测试中是不需要的。

并不需要将 *tests/integration_test.rs* 中的任何代码标注为 `#[cfg(test)]`。 `tests` 文件夹在 Cargo 中是一个特殊的文件夹，Cargo 只会在运行 `cargo test` 时编译这个目录中的文件。现在就运行 `cargo test` 试试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/
integration_test-1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

现在有了三个部分的输出：单元测试、集成测试和文档测试。注意如果一个部分的任何测试失败，之后的部分都不会运行。例如如果一个单元测试失败，则不会有任何集成测试和文档测试的输出，因为这些测试只会在所有单元测试都通过后才会执行。

第一部分单元测试与我们之前见过的一样：每个单元测试一行（示例 11-12 中有一个叫做 `internal` 的测试），接着是一个单元测试的摘要行。

集成测试部分以行 `Running tests/integration_test.rs`开头。接下来每一行是一个集成测试中的测试函数，以及一个位于 `Doc-tests adder` 部分之前的集成测试的摘要行。

每一个集成测试文件有对应的测试结果部分，所以如果在 *tests* 目录中增加更多文件，测试结果中就会有更多集成测试结果部分。

我们仍然可以通过指定测试函数的名称作为 `cargo test` 的参数来运行特定集成测试。也可以使用 `cargo test` 的 `--test` 后跟文件的名称来运行某个特定集成测试文件中的所有测试：

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs (target/debug/deps/
integration_test-82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

这个命令只运行 *tests/integration_test.rs* 文件中的测试。

集成测试中的子模块

随着集成测试的增加，你可能希望在 `tests` 目录创建更多文件以便更好地组织它们；例如根据测试的功能来将测试分组。如前所述，*tests* 目录中的每一个文件都被编译成一个单独的 crate，这有助于创建独立的作用域，以便更接近于最终用户使用你的 crate 的方式。但这意味着，*tests* 目录中的文件的行为，和你在第七章中学习如何将代码分为模块和文件时，学到的 *src* 中的文件的行为不一样。

当你有一些在多个集成测试文件都会用到的帮助函数，而你尝试按照第七章"将模块移动到其他文件"部分的步骤将它们提取到一个通用的模块中时，*tests* 目录中文件行为的不同就会凸显出来。例如，如果我们在 *tests/common.rs* 中创建一个名为 `setup` 的函数，并希望在多个测试文件的测试函数中调用它，就可以在 `setup` 中添加想要复用的代码：

文件名：tests/common.rs

```
pub fn setup() {
    // setup code specific to your library's tests would go here
}
```

如果再次运行测试，将会在测试结果中看到一个新的对应 *common.rs* 文件的测试结果部分，即便这个文件并没有包含任何测试函数，也没有任何地方调用了 `setup` 函数：

```
```

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/
integration_test-92948b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

我们并不想要 `common` 出现在测试结果中显示 `running 0 tests`。我们只是想与其他集成测试文件共享一些代码而已。为了不让 `common` 出现在测试输出中，我们将创建 *tests/common/mod.rs*，而不是创建 *tests/common.rs* 。现在项目目录结构看起来像这样：

```
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── common
    │   └── mod.rs
    └── integration_test.rs
```

这是一种老的命名规范，正如第七章“另一种文件路径”中提到的 Rust 仍然理解它们。这样命名告诉 Rust 不要将 `common` 看作一个集成测试文件。将 `setup` 函数代码移动到 *tests/common/mod.rs* 并删除 *tests/common.rs* 文件之后，测试输出中将不会出现这一部分。*tests* 目录中的子目录不会被作为单独的 crate 编译或作为一个测试结果部分出现在测试输出中。

一旦创建了 *tests/common/mod.rs*，就可以将其作为模块以便在任何集成测试文件中使用。这里是一个 *tests/integration_test.rs* 中调用 `setup` 函数的 `it_adds_two` 测试的示例：

文件名：tests/integration_test.rs

```
use adder::add_two;

mod common;

#[test]
fn it_adds_two() {
    common::setup();

    let result = add_two(2);
    assert_eq!(result, 4);
}
```

注意 `mod common;` 声明与示例 7-21 中展示的模块声明相同。接着在测试函数中就可以调用 `common::setup()` 了。

二进制 crate 的集成测试

如果项目是二进制 crate 并且只包含 *src/main.rs* 而没有 *src/lib.rs*，这样就不可能在 *tests* 目录创建集成测试并也无法通过 `use` 语句将 *src/main.rs* 中定义的函数引入作用域。只有库 crate 才会向其他 crate 暴露了可供调用和使用的函数；二进制 crate 只意在单独运行。

这就是许多 Rust 二进制项目使用一个简单的 *src/main.rs* 调用 *src/lib.rs* 中的逻辑的原因之一。因为通过这种结构，集成测试**就可以**通过 `use` 来测试库 crate 中的重要功能了。而如果这些重要的功能没有问题的话，*src/main.rs* 中的少量代码也就会正常工作且不需要测试。

总结

Rust 的测试功能提供了一个确保即使你改变了函数的实现方式，也能继续以期望的方式运行的途径。单元测试独立地验证库的不同部分，也能够测试私有函数实现细节。集成测试则检查多个部分是否能结合起来正确地工作，并像其他外部代码那样测试库的公有 API。即使 Rust 的类型系统和所有权规则可以帮助避免某些类型的 bug，不过测试对于减少代码中不符合期望行为的逻辑 bug 仍然是很重要的。

让我们将本章和其前面各章所学的知识组合起来，在下一章一起编写一个项目！

一个 I/O 项目：构建一个命令行程序

本章既是一个目前所学的多项技能的概括，也是一个更多标准库功能的探索。我们将构建一个与文件和命令行输入/输出交互的命令行工具来练习现在一些你已经掌握的 Rust 概念。

Rust 的运行速度、安全性、单二进制文件输出和跨平台支持使其成为创建命令行程序的理想语言，所以我们的项目将创建一个我们自己版本的经典命令行搜索工具：`grep`。grep 是“**G**lobally search a **R**egular **E**xpression and **P**rint.” 的首字母缩写。`grep` 最简单的使用场景是在特定文件中搜索指定字符串。为此，`grep` 获取一个文件路径和一个字符串作为参数，接着读取文件并找到其中包含字符串参数的行，然后打印出这些行。

在这个过程中，我们会展示如何让我们的命令行工具利用很多命令行工具中用到的终端功能。读取环境变量来使得用户可以配置工具的行为。打印到标准错误控制流（`stderr`）而不是标准输出（`stdout`），例如这样用户可以选择将成功输出重定向到文件中的同时仍然在屏幕上显示错误信息。

一位 Rust 社区的成员，Andrew Gallant，已经创建了一个功能完整且非常快速的 `grep` 版本，名为 `ripgrep`。相比之下，我们的版本将非常简单，本章将教会你一些帮助理解像 `ripgrep` 这样真实项目的背景知识。

我们的 `grep` 项目将会结合之前所学的一些概念：

- 代码组织（第七章）
- vector 和字符串（第八章）
- 错误处理（第九章）
- 合理地使用 trait 和生命周期（第十章）
- 编写测试（第十一章）

另外还会简要地讲到闭包、迭代器和 trait 对象，它们分别会在第十三章和第十八章中详细介绍。

接受命令行参数

一如既往使用 `cargo new` 新建一个项目，我们称之为 `minigrep` 以便与可能已经安装在系统上的 `grep` 工具相区分。

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

第一个任务是让 `minigrep` 能够接受两个命令行参数：文件路径和要搜索的字符串。也就是说我们希望能够使用 `cargo run`，两个连字符来表明后面的参数是要传递给程序而不是 `cargo`，要搜索的字符串和被搜索的文件的路径来运行程序，像这样：

```
$ cargo run -- searchstring example-filename.txt
```

现在 `cargo new` 生成的程序无法处理传递给它的参数。Crates.io 上有一些现成的库可以帮助我们接受命令行参数，不过我们正在学习这一概念，就让我们自己实现这个功能。

读取参数值

为了确保 `minigrep` 能够获取传递给它的命令行参数的值，我们需要一个 Rust 标准库提供的 `std::env::args` 函数。这个函数返回一个传递给程序的命令行参数的**迭代器**（*iterator*）。我们会在第十三章全面介绍迭代器。但是现在只需理解迭代器的两个细节：迭代器生成一系列的值，可以在迭代器上调用 `collect` 方法将其转换为一个集合，比如包含所有迭代器产生元素的 vector。

示例 12-1 中的代码允许 `minigrep` 程序读取任何传递给它的命令行参数并将其收集到一个 vector 中。

文件名：src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

示例 12-1：将命令行参数收集到一个 vector 中并打印出来

首先使用 `use` 语句来将 `std::env` 模块引入作用域以便可以使用它的 `args` 函数。注意 `std::env::args` 函数被嵌套进了两层模块中。正如第七章讲到的，当所需函数嵌套了多于一层模块时，通常将父模块引入作用域而不是其自身。这便于我们利用 `std::env` 中的其他函数。这比增加了 `use std::env::args;` 后仅仅使用 `args` 调用函数要更明确一些，因为 `args` 容易被错认成一个定义于当前模块的函数。

> **`args` 函数和无效的 Unicode**
>
> 注意 `std::env::args` 在其任何参数包含无效 Unicode 字符时会 panic。如果你需要接受包含无效 Unicode 字符的参数，使用 `std::env::args_os` 代替。这个函数返回 `OsString` 值而不是 `String` 值。这里出于简单考虑使用了 `std::env::args`，因为 `OsString` 值每个平台都不一样而且比 `String` 值处理起来更为复杂。

在 `main` 函数的第一行，我们调用了 `env::args`，并立即使用 `collect` 来创建了一个包含迭代器所有值的 vector。`collect` 可以被用来创建很多类型的集合，所以这里显式注明 `args` 的类型来指定我们需要一个字符串 vector。虽然在 Rust 中我们很少会需要注明类型，然而 `collect` 是一个经常需要注明类型的函数，因为 Rust 不能推断出你想要什么类型的集合。

最后，我们使用调试宏打印出 vector。让我们先在不传递任何参数的情况下运行一次代码，然后再传入两个参数运行一次：

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/minigrep`
[src/main.rs:5:5] args = [
    "target/debug/minigrep",
]
```

```
$ cargo run -- needle haystack
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/minigrep needle haystack`
[src/main.rs:5:5] args = [
    "target/debug/minigrep",
    "needle",
    "haystack",
]
```

注意 vector 的第一个值是 `"target/debug/minigrep"`，它是我们二进制文件的名称。这与 C 中的参数列表的行为相匹配，让程序使用在执行时调用它们的名称。如果要在消息中打印它或者根据用于调用程序的命令行别名更改程序的行为，通常可以方便地访问程序名称，不过考虑到本章的目的，我们将忽略它并只保存所需的两个参数。

将参数值保存进变量

目前程序可以访问指定为命令行参数的值。现在需要将这两个参数的值保存进变量这样就可以在程序的余下部分使用这些值了。让我们如示例 12-2 这样做：

文件名：src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
```

```
    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {query}");
    println!("In file {file_path}");
}
```

示例 12-2：创建变量来存放查询参数和文件路径参数

正如之前打印出 vector 时所看到的，程序的名称占据了 vector 的第一个值 `args[0]`，所以我们从索引为 `1` 的参数开始。`minigrep` 获取的第一个参数是需要搜索的字符串，所以将第一个参数的引用存放在变量 `query` 中。第二个参数将是文件路径，所以将第二个参数的引用放入变量 `file_path` 中。

我们将临时打印出这些变量的值来证明代码如我们期望的那样工作。使用参数 `test` 和 `sample.txt` 再次运行这个程序：

```
$ cargo run -- test sample.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

太好了，程序正常工作！我们将所需的参数值保存进了对应的变量中。之后会增加一些错误处理来应对类似用户没有提供参数的情况。不过现在我们将忽略它们并开始增加读取文件功能。

读取文件

现在我们要增加读取由 `file_path` 命令行参数指定的文件的功能。首先，需要一个用来测试的示例文件：我们会用一个拥有多行少量文本且有一些重复单词的文件。示例 12-3 是一首艾米莉·狄金森（Emily Dickinson）的诗，它正适合这个工作！在项目根目录创建一个文件 *poem.txt*，并输入诗 “I’m nobody! Who are you?”：

文件名：poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

示例 12-3：艾米莉·狄金森的诗 “I’m nobody! Who are you?”，一个好的测试用例

有了文本后，修改 *src/main.rs* 并增加如示例 12-4 所示的打开文件的代码：

文件名：src/main.rs

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {file_path}");

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

示例 12-4：读取第二个参数所指定的文件内容

首先，我们增加了一个 `use` 语句来引入标准库中的相关部分：我们需要 `std::fs` 来处理文件。

在 `main` 中新增了一行语句：`fs::read_to_string` 接受 `file_path`，打开文件，接着返回包含其内容的 `std::io::Result<String>`。

在这些代码之后，我们再次增加了临时的 `println!` 打印出读取文件之后 `contents` 的值，这样就可以检查目前为止的程序能否工作。

尝试运行这些代码，随意指定一个字符串作为第一个命令行参数（因为还未实现搜索功能的部分）而将 *poem.txt* 文件将作为第二个参数：

```
$ cargo run -- the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
```

```
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

好的！代码读取并打印出了文件的内容。虽然它还有一些瑕疵：此时 `main` 函数有着多个职能，通常函数只负责一个功能的话会更简洁并易于维护。另一个问题是没有尽可能的处理错误。虽然我们的程序还很小，这些瑕疵并不是什么大问题，不过随着程序功能的丰富，要干净地修复它们就会越来越困难。在开发程序时，及早开始重构是一个良好实践，因为重构少量代码时要容易的多。接下来我们就来进行重构。

重构改进模块性和错误处理

为了改善我们的程序这里有四个问题需要修复，而且它们都与程序的组织方式和如何处理潜在错误有关。第一，`main` 现在进行了两个任务：它解析了参数并打开了文件。然而随着 `main` 中的功能持续增加，`main` 函数处理的独立任务也会增加。当函数承担了更多责任，它就更难以推导，更难以测试，并且更难以在不破坏其他部分的情况下做出修改。最好能分离出功能以便每个函数各司其职。

这同时也关系到第二个问题：`query` 和 `file_path` 是程序中的配置变量，而像 `contents` 则用来执行程序逻辑。随着 `main` 函数的增长，就需要引入更多的变量到作用域中，而当作用域中有更多的变量时，将更难以追踪每个变量的目的。最好能将配置变量组织进一个结构，这样就能使它们的目的更明确了。

第三个问题是如果打开文件失败时我们使用 `expect` 来打印出错误信息，不过这个错误信息只是说 `Should have been able to read the file`。读取文件失败的原因有多种：例如文件可能不存在，或者没有打开此文件的权限。目前，无论哪种情况，都会显示相同的错误信息，无法为用户提供任何有用的线索！

第四，我们也使用 `expect` 来处理参数错误，如果用户没有指定足够的参数来运行程序，他们会从 Rust 得到 `index out of bounds` 错误，而这并不能明确地解释问题。如果所有的错误处理都位于一处，这样将来的维护者在需要修改错误处理逻辑时就只需要考虑这一处代码。将所有的错误处理都放在一处也有助于确保我们打印的错误信息对终端用户来说是有意义的。

让我们通过重构项目来解决这四个问题。

二进制项目的关注分离

`main` 函数负责多个任务的组织问题在许多二进制项目中很常见。所以 Rust 社区开发出一类在 `main` 函数开始变得庞大时进行二进制程序的关注分离的指南。这些过程包括如下步骤：

- 将程序拆分成 *main.rs* 和 *lib.rs* 并将程序的逻辑放入 *lib.rs* 中。
- 当命令行解析逻辑比较小时，可以保留在 *main.rs* 中。
- 当命令行解析开始变得复杂时，也同样将其从 *main.rs* 提取到 *lib.rs* 中。

经过这些过程之后保留在 `main` 函数中的责任应该被限制为：

- 使用参数值调用命令行解析逻辑
- 设置任何其他的配置
- 调用 *lib.rs* 中的 `run` 函数
- 如果 `run` 返回错误，则进行错误处理

这个模式的一切就是为了关注分离：*main.rs* 处理程序运行，而 *lib.rs* 处理所有的真正的任务逻辑。因为不能直接测试 `main` 函数，这个结构通过将所有的程序逻辑移动到 *lib.rs* 的函数中使得我们可以测试它们。仅仅保留在 *main.rs* 中的代码将足够小以便阅读就可以验证其正确性。让我们遵循这些步骤来重构程序。

提取参数解析器

首先，我们将解析参数的功能提取到一个 `main` 将会调用的函数中，为将命令行解析逻辑移动到 *src/lib.rs* 中做准备。示例 12-5 中展示了新 `main` 函数的开头，它调用了新函数 `parse_config`。目前它仍将定义在 *src/main.rs* 中：

文件名：src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];

    (query, file_path)
}
```

示例 12-5：从 `main` 中提取出 `parse_config` 函数

我们仍然将命令行参数收集进一个 vector，不过不同于在 `main` 函数中将索引 1 的参数值赋值给变量 `query` 和将索引 2 的值赋值给变量 `file_path`，我们将整个 vector 传递给 `parse_config` 函数。接着 `parse_config` 函数将包含决定哪个参数该放入哪个变量的逻辑，并将这些值返回到 `main`。我们仍然在 `main` 中创建变量 `query` 和 `file_path`，不过 `main` 不再负责处理命令行参数与变量如何对应。

这对重构我们这小程序可能有点大材小用，不过我们将采用小的、增量的步骤进行重构。在做出这些改变之后，再次运行程序并验证参数解析是否仍然正常。经常验证你的进展是一个好习惯，这样在遇到问题时能帮助你定位问题的成因。

组合配置值

我们可以采取另一个小的步骤来进一步改善 `parse_config` 函数。现在函数返回一个元组，不过立刻又将元组拆成了独立的部分。这是一个我们可能没有进行正确抽象的信号。

另一个表明还有改进空间的迹象是 `parse_config` 名称的 `config` 部分，它暗示了我们返回的两个值是相关的并都是一个配置值的一部分。目前除了将这两个值组合进元组之外并没有表达这个数据结构的意义；相反我们可以将这两个值放入一个结构体并给每个字段一个有意义的名字。这会让未来的维护者更容易理解不同的值如何相互关联以及它们的目的。

示例 12-6 展示了 `parse_config` 函数的改进。

文件名：src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    // --snip--
}
```

```
struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

    Config { query, file_path }
}
```

示例 12-6：重构 `parse_config` 返回一个 `Config` 结构体实例

我们添加了一个名为 `Config` 的结构体，其中包含 `query` 和 `file_path` 字段。`parse_config` 的签名表明它现在返回一个 `Config` 值。在之前的 `parse_config` 函数体中，我们返回了引用 `args` 中 `String` 值的字符串 slice，现在我们定义 `Config` 来包含拥有所有权的 `String` 值。`main` 中的 `args` 变量是参数值的所有者并只允许 `parse_config` 函数借用它们，这意味着如果 `Config` 尝试获取 `args` 中值的所有权将违反 Rust 的借用规则。

还有许多不同的方式可以处理 `String` 的数据，而最简单但有些不太高效的方式是调用这些值的 `clone` 方法。这会生成 `Config` 实例可以拥有的数据的完整拷贝，不过会比储存字符串数据的引用消耗更多的时间和内存。不过拷贝数据使得代码显得更加直白因为无需管理引用的生命周期，所以在这种情况下牺牲一小部分性能来换取简洁性的取舍是值得的。

> **使用 `clone` 的权衡取舍**
>
> 由于其运行时消耗，许多 Rustacean 之间有一个趋势是倾向于避免使用 `clone` 来解决所有权问题。在关于迭代器的第十三章中，我们将会学习如何更有效率的处理这种情况，不过现在，复制一些字符串来取得进展是没有问题的，因为只会进行一次这样的拷贝，而且文件路径和要搜索的字符串都比较短。在第一轮编写时拥有一个可以工作但有点低效的程序要比尝试过度优化代码更好一些。随着你对 Rust 更加熟练，将能更轻松的直奔合适的方法，不过现在调用 `clone` 是完全可以接受的。

我们更新 `main` 将 `parse_config` 返回的 `Config` 实例放入变量 `config` 中，并将之前分别使用 `query` 和 `file_path` 变量的代码更新为现在的使用 `Config` 结构体的字段的代码。

现在代码更明确的表现了我们的意图，`query` 和 `file_path` 是相关联的并且它们的目的是配置程序如何工作。任何使用这些值的代码就知道在 `config` 实例中对应目的的字段名中寻找它们。

创建 `Config` 的构造函数

目前为止，我们将负责解析命令行参数的逻辑从 `main` 提取到了 `parse_config` 函数中，这有助于我们看清值 `query` 和 `file_path` 是相互关联的并应该在代码中表现这种关系。接着我们增加了 `Config` 结构体来描述 `query` 和 `file_path` 的相关性，并能够从 `parse_config` 函数中将这些值的名称作为结构体字段名称返回。

所以现在 `parse_config` 函数的目的是创建一个 `Config` 实例，我们可以将 `parse_config` 从一个普通函数变为一个叫做 `new` 的与结构体关联的函数。做出这个改变使得代码更符合习惯：可

以像标准库中的 `String` 调用 `String::new` 来创建一个该类型的实例那样，将 `parse_config` 变为一个与 `Config` 关联的 `new` 函数。示例 12-7 展示了需要做出的修改：

文件名：src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

示例 12-7：将 `parse_config` 变为 `Config::new`

这里将 `main` 中调用 `parse_config` 的地方更新为调用 `Config::new`。我们将 `parse_config` 的名字改为 `new` 并将其移动到 `impl` 块中，这使得 `new` 函数与 `Config` 相关联。再次尝试编译并确保它可以工作。

修复错误处理

现在我们开始修复错误处理。回忆一下之前提到过如果 `args` vector 包含少于 3 个项并尝试访问 vector 中索引 `1` 或索引 `2` 的值会造成程序 panic。尝试不带任何参数运行程序；这将看起来像这样：

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`

thread 'main' panicked at src/main.rs:27:21:
index out of bounds: the len is 1 but the index is 1
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

`index out of bounds: the len is 1 but the index is 1` 是一个面向程序员的错误信息，然而这并不能真正帮助终端用户理解发生了什么和他们应该做什么。现在就让我们修复它吧。

改善错误信息

在示例 12-8 中，在 `new` 函数中增加了一个检查在访问索引 `1` 和 `2` 之前检查 slice 是否足够长。如果 slice 不够长，程序会打印一个更好的错误信息并 panic：

文件名：src/main.rs

```
    // --snip--
    fn new(args: &[String]) -> Config {
        if args.len() < 3 {
            panic!("not enough arguments");
        }
        // --snip--
```

示例 12-8：增加一个参数数量检查

这类似于示例 9-13 中的 `Guess::new` 函数，那里如果 `value` 参数超出了有效值的范围就调用 `panic!`。不同于检查值的范围，这里检查 `args` 的长度至少是 `3`，而函数的剩余部分则可以在假设这个条件成立的基础上运行。如果 `args` 少于 3 个项，则这个条件将为真，并调用 `panic!` 立即终止程序。

有了 `new` 中这几行额外的代码，再次不带任何参数运行程序并看看现在错误看起来像什么：

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`

thread 'main' panicked at src/main.rs:26:13:
not enough arguments
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

这个输出就更好了：现在有了一个合理的错误信息。然而，还是有一堆额外的信息我们不希望提供给用户。所以在这里使用示例 9-13 中的技术可能不是最好的；正如第九章所讲到的一样，`panic!` 的调用更趋向于程序上的问题而不是使用上的问题。相反我们可以使用第九章学习的另一个技术 —— 返回一个可以表明成功或错误的 `Result`。

返回 `Result` 而不是调用 `panic!`

我们可以选择返回一个 `Result` 值，它在成功时会包含一个 `Config` 的实例，而在错误时会描述问题。我们还将把函数名从 `new` 改为 `build`，因为许多程序员希望 `new` 函数永远不会失败。当 `Config::new` 与 `main` 交流时，可以使用 `Result` 类型来表明这里存在问题。接着修改 `main` 将 `Err` 成员转换为对用户更友好的错误，而不是 `panic!` 调用产生的关于 `thread 'main'` 和 `RUST_BACKTRACE` 的文本。

示例 12-9 展示了为了返回 `Result` 在 `Config::new` 的返回值和函数体中所需的改变。注意这还不能编译，直到下一个示例更新了 `main` 之后。

文件名：src/main.rs

```
impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();
```

```
        Ok(Config { query, file_path })
    }
}
```



示例 12-9：从 `Config::build` 中返回 `Result`

现在 `build` 函数返回一个 `Result`，在成功时带有一个 `Config` 实例而在出现错误时总是一个有着 `'static` 生命周期的字符串字面值。

`build` 函数体中有两处修改：当没有足够参数时不再调用 `panic!`，而是返回 `Err` 值。同时我们将 `Config` 返回值包装进 `Ok` 成员中。这些修改使得函数符合其新的类型签名。

通过让 `Config::build` 返回一个 `Err` 值，这就允许 `main` 函数处理 `build` 函数返回的 `Result` 值并在出现错误的情况更明确的结束进程。

调用 `Config::build` 并处理错误

为了处理错误情况并打印一个对用户友好的信息，我们需要像示例 12-10 那样更新 `main` 函数来处理现在 `Config::build` 返回的 `Result`。另外还需要手动实现原先由 `panic!` 负责的工作，即以非零错误码退出命令行工具的工作。非零的退出状态是一个惯例信号，用来告诉调用程序的进程：该程序以错误状态退出了。

文件名：src/main.rs

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
```

示例 12-10：如果新建 `Config` 失败则使用错误码退出

在上面的示例中，使用了一个之前没有详细说明的方法：`unwrap_or_else`，它定义于标准库的 `Result<T, E>` 上。使用 `unwrap_or_else` 可以进行一些自定义的非 `panic!` 的错误处理。当 `Result` 是 `Ok` 时，这个方法的行为类似于 `unwrap`：它返回 `Ok` 内部封装的值。然而，当其值是 `Err` 时，该方法会调用一个**闭包**（*closure*），也就是一个我们定义的作为参数传递给 `unwrap_or_else` 的匿名函数。第十三章会更详细地介绍闭包。现在你需要理解的是 `unwrap_or_else` 会将 `Err` 的内部值，也就是示例 12-9 中增加的 `not enough arguments` 静态字符串的情况，传递给闭包中位于两道竖线间的参数 `err`。闭包中的代码在其运行时可以使用这个 `err` 值。

我们新增了一个 `use` 行来从标准库中导入 `process`。在错误的情况闭包中将被运行的代码只有两行：我们打印出了 `err` 值，接着调用了 `std::process::exit`。`process::exit` 会立即停止程序并将传递给它的数字作为退出状态码。这类似于示例 12-8 中使用的基于 `panic!` 的错误处理，除了不会再得到所有的额外输出了。让我们试试：

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

非常好！现在输出对于用户来说就友好多了。

从 `main` 提取逻辑

现在我们完成了配置解析的重构，让我们转向程序的逻辑。正如"二进制项目的关注分离"部分所展开的讨论，我们将提取一个叫做 `run` 的函数来存放目前 `main` 函数中不属于设置配置或处理错误的所有逻辑。一旦完成这些，`main` 函数将简明得足以通过观察来验证，而我们将能够为所有其他逻辑编写测试。

示例 12-11 展示了提取出来的 `run` 函数。目前我们只进行小的增量式的提取函数的改进。我们仍将在 *src/main.rs* 中定义这个函数：

文件名：src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}

// --snip--
```

示例 12-11：提取 `run` 函数来包含剩余的程序逻辑

现在 `run` 函数包含了 `main` 中从读取文件开始的剩余的所有逻辑。`run` 函数获取一个 `Config` 实例作为参数。

从 `run` 函数返回错误

通过将剩余的逻辑分离进 `run` 函数中，就可以像示例 12-9 中的 `Config::build` 那样改进错误处理。不再通过 `expect` 允许程序 panic，`run` 函数将会在出错时返回一个 `Result<T, E>`。这让我们进一步以一种对用户友好的方式将处理错误的逻辑统一到 `main` 中。示例 12-12 展示了 `run` 签名和函数体中的改变：

文件名：src/main.rs

```
use std::error::Error;
```

```
// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}
```

示例 12-12：修改 `run` 函数返回 `Result`

这里我们做出了三个明显的修改。首先，将 `run` 函数的返回类型变为 `Result<(), Box<dyn Error>>`。之前这个函数返回 unit 类型 `()`，现在它仍然保持作为 `Ok` 时的返回值。

对于错误类型，使用 **trait 对象** `Box<dyn Error>`（在开头使用了 `use` 语句将 `std::error::Error` 引入作用域）。第十八章会涉及 trait 对象。目前只需知道 `Box<dyn Error>` 意味着函数会返回实现了 `Error` trait 的类型，不过无需指定具体将会返回的值的类型。这提供了在不同的错误场景可能有不同类型的错误返回值的灵活性。这也就是 `dyn`，它是“动态的”（“dynamic”）的缩写。

第二个改变是去掉了 `expect` 调用并替换为第九章讲到的 `?` 运算符。不同于遇到错误就 `panic!`，`?` 会从函数中返回错误值并让调用者来处理它。

第三个修改是现在成功时这个函数会返回一个 `Ok` 值。因为 `run` 函数签名中声明成功类型返回值是 `()`，这意味着需要将 unit 类型值包装进 `Ok` 值中。`Ok(())` 一开始看起来有点奇怪，不过这样使用 `()` 是惯用的做法，表明调用 `run` 函数只是为了它的副作用；函数并没有返回什么有意义的值。

运行上述代码时，它能够编译通过，但会显示一条警告：

```
$ cargo run -- the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
  --> src/main.rs:19:5
   |
19 |     run(config);
   |     ^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
   |
19 |     let _ = run(config);
   |     +++++++

warning: `minigrep` (bin "minigrep") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.71s
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
```

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Rust 提示我们的代码忽略了 `Result` 值，它可能表明这里存在一个错误。但我们却没有检查这里是否有一个错误，而编译器提醒我们这里应该有一些错误处理代码！现在就让我们修正这个问题。

在 `main` 中处理 `run` 返回的错误

我们将检查错误并使用类似示例 12-10 中 `Config::build` 处理错误的技术来处理它们，不过有一些细微的不同：

文件名：src/main.rs

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        println!("Application error: {e}");
        process::exit(1);
    }
}
```

我们使用 `if let` 来检查 `run` 是否返回一个 `Err` 值，不同于 `unwrap_or_else`，并在出错时调用 `process::exit(1)`。`run` 并不返回像 `Config::build` 返回的 `Config` 实例那样需要 `unwrap` 的值。因为 `run` 在成功时返回 `()`，而我们只关心检测错误，所以并不需要 `unwrap_or_else` 来返回未封装的值，因为它只会是 `()`。

不过两个例子中 `if let` 和 `unwrap_or_else` 的函数体都一样：打印出错误并退出。

将代码拆分到库 crate

现在我们的 `minigrep` 项目看起来好多了！现在我们将要拆分 *src/main.rs* 并将一些代码放入 *src/lib.rs*，这样就能测试它们并拥有一个含有更少功能的 `main` 函数。

让我们将所有不是 `main` 函数的代码从 *src/main.rs* 移动到新文件 *src/lib.rs* 中：

- `run` 函数定义
- 相关的 `use` 语句
- `Config` 的定义
- `Config::build` 函数定义

现在 *src/lib.rs* 的内容应该看起来像示例 12-13（为了简洁省略了函数体）。注意直到下一个示例修改完 *src/main.rs* 之后，代码还不能编译。

文件名：src/lib.rs



```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}
```

示例 12-13：将 `Config` 和 `run` 移动到 *src/lib.rs*

这里大量使用了 `pub` 关键字：在 `Config`、其字段和其 `build` 方法，以及 `run` 函数上。现在我们有了一个拥有可以测试的公有 API 的库 crate 了。

现在需要在 *src/main.rs* 中将移动到 *src/lib.rs* 的代码引入二进制 crate 的作用域中，如示例 12-14 所示：

文件名：src/main.rs

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}
```

示例 12-14：将 `minigrep` crate 引入 *src/main.rs* 的作用域中

我们添加了一行 `use minigrep::Config`，它将 `Config` 类型引入作用域，并使用 crate 名称作为 `run` 函数的前缀。通过这些重构，所有功能应该能够联系在一起并运行了。运行 `cargo run` 来确保一切都正确的衔接在一起。

呼！我们做了大量的工作，不过我们为将来的成功打下了基础。现在处理错误将更容易，同时代码也更加模块化。从现在开始几乎所有的工作都将在 *src/lib.rs* 中进行。

让我们利用这些新创建的模块的优势来进行一些在旧代码中难以展开的工作，这些工作在新代码中非常容易实现：那就是编写测试！

采用测试驱动开发完善库的功能

现在我们将逻辑提取到了 *src/lib.rs* 并将所有的参数解析和错误处理留在了 *src/main.rs* 中，为代码的核心功能编写测试将更加容易。我们可以直接使用多种参数调用函数并检查返回值而无需从命令行运行二进制文件了。

在这一部分，我们将遵循测试驱动开发（Test Driven Development, TDD）的模式来逐步增加 `minigrep` 的搜索逻辑。它遵循如下步骤：

1. 编写一个失败的测试，并运行它以确保它失败的原因是你所期望的。
2. 编写或修改足够的代码来使新的测试通过。
3. 重构刚刚增加或修改的代码，并确保测试仍然能通过。
4. 从步骤 1 开始重复！

虽然这只是众多编写软件的方法之一，不过 TDD 有助于驱动代码的设计。在编写能使测试通过的代码之前编写测试有助于在开发过程中保持高测试覆盖率。

我们将测试驱动实现实际在文件内容中搜索查询字符串并返回匹配的行示例的功能。我们将在一个叫做 `search` 的函数中增加这些功能。

编写失败测试

去掉 *src/lib.rs* 和 *src/main.rs* 中用于检查程序行为的 `println!` 语句，因为不再真正需要它们了。接着我们会像第十一章那样增加一个 `test` 模块和一个测试函数。测试函数指定了 `search` 函数期望拥有的行为：它会获取一个需要查询的字符串和用来查询的文本，并只会返回包含请求的文本行。示例 12-15 展示了这个测试，它还不能编译：

文件名：src/lib.rs



```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}
```

示例 12-15：创建一个我们期望的 `search` 函数的失败测试

这里选择使用 `"duct"` 作为这个测试中需要搜索的字符串。用来搜索的文本有三行，其中只有一行包含 `"duct"`。（注意双引号之后的反斜杠，这告诉 Rust 不要在字符串字面值内容的开头加入换行符）我们断言 `search` 函数的返回值只包含期望的那一行。

我们还不能运行这个测试并看到它失败，因为它甚至都还不能编译：`search` 函数还不存在呢！根据 TDD 的原则，我们将增加足够的代码来使其能够编译：一个总是会返回空 vector 的

`search` 函数定义，如示例 12-16 所示。然后这个测试应该能够编译并因为空 vector 并不匹配一个包含一行 `"safe, fast, productive."` 的 vector 而失败。

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

示例 12-16：刚好足够使测试通过编译的 `search` 函数定义

注意需要在 `search` 的签名中定义一个显式生命周期 `'a` 并用于 `contents` 参数和返回值。回忆一下第十章中讲到生命周期参数指定哪个参数的生命周期与返回值的生命周期相关联。在这个例子中，我们表明返回的 vector 中应该包含引用参数 `contents`（而不是参数`query`）slice 的字符串 slice。

换句话说，我们告诉 Rust 函数 `search` 返回的数据将与 `search` 函数中的参数 `contents` 的数据存在的一样久。这是非常重要的！为了使这个引用有效那么**被** slice 引用的数据也需要保持有效；如果编译器认为我们是在创建 `query` 而不是 `contents` 的字符串 slice，那么安全检查将是不正确的。

如果我们忘记添加生命周期注解而尝试编译此函数，就会得到如下错误：

```
$ cargo build
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
  --> src/lib.rs:28:51
   |
28 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
   |                      ----            ----         ^ expected named lifetime
parameter
   |
   = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `query` or `contents`
help: consider introducing a named lifetime parameter
   |
28 | pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
   |              ++++         ++                 ++              ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` (lib) due to 1 previous error
```

Rust 不可能知道我们需要的是哪一个参数，所以需要显式地告诉它。因为参数 `contents` 包含了所有的文本而且我们希望返回匹配的那部分文本，所以我们知道 `contents` 是应该要使用生命周期语法来与返回值相关联的参数。

其他语言中并不需要你在函数签名中将参数与返回值相关联。所以这么做可能仍然感觉有些陌生，随着时间的推移这将会变得越来越容易。你可能想要将这个例子与第十章中“生命周期确保引用有效”部分做对比。

现在运行测试：

```
```

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.97s
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----

thread 'tests::one_result' panicked at src/lib.rs:44:9:
assertion `left == right` failed
  left: ["safe, fast, productive."]
 right: []
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

很好，测试失败了，这正是我们所期望的。修改代码来让测试通过吧！

编写使测试通过的代码

目前测试之所以会失败是因为我们总是返回一个空的 vector。为了修复并实现 `search`，我们的程序需要遵循如下步骤：

1. 遍历内容的每一行文本。
2. 查看这一行是否包含要搜索的字符串。
3. 如果有，将这一行加入列表返回值中。
4. 如果没有，什么也不做。
5. 返回匹配到的结果列表。

让我们一步一步的来，从遍历每行开始。

使用 `lines` 方法逐行遍历

Rust 有一个有助于一行一行遍历字符串的方法，出于方便它被命名为 `lines`，它如示例 12-17 这样工作。注意这还不能编译：

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```



示例 12-17：遍历 `contents` 的每一行

`lines` 方法返回一个迭代器。第十三章会深入了解迭代器，不过我们已经在示例 3-5 中见过使用迭代器的方法了，在那里使用了一个 `for` 循环和迭代器在一个集合的每一项上运行了一些代码。

用查询字符串搜索每一行

接下来将会增加检查当前行是否包含查询字符串的功能。幸运的是，字符串类型为此也有一个叫做 `contains` 的实用方法！如示例 12-18 所示在 `search` 函数中加入 `contains` 方法调用。注意这仍然不能编译：

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // 对文本行进行操作
        }
    }
}
```



示例 12-18：增加检查文本行是否包含 `query` 中字符串的功能

目前，我们正在构建功能。为了让代码能够编译，需要从函数体返回一个我们在函数签名中所声明的值。

存储匹配的行

为了完成这个函数，我们需要一种方法来存储要返回的匹配行。为此可以在 `for` 循环之前创建一个可变的 vector 并调用 `push` 方法在 vector 中存放一个 `line`。在 `for` 循环之后，返回这个 vector，如示例 12-19 所示：

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

示例 12-19：储存匹配的行以便可以返回它们

现在 `search` 函数应该返回只包含 `query` 的那些行，而测试应该会通过。让我们运行测试：

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.22s
```

```
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

测试通过了，我们知道它可以工作了！

现在正是可以考虑重构的时机，在保证测试通过，保持功能不变的前提下重构 `search` 函数。`search` 函数中的代码并不坏，不过并没有利用迭代器的一些实用功能。第十三章将回到这个例子并深入探索迭代器并看看如何改进代码。

在 `run` 函数中使用 `search` 函数

现在 `search` 函数是可以工作并测试通过了的，我们需要实际在 `run` 函数中调用 `search`。需要将 `config.query` 值和 `run` 从文件中读取的 `contents` 传递给 `search` 函数。接着 `run` 会打印出 `search` 返回的每一行：

文件名：src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{line}");
    }

    Ok(())
}
```

这里仍然使用了 `for` 循环获取了 `search` 返回的每一行并打印出来。

现在整个程序应该可以工作了！让我们试一试，首先使用一个只会在艾米莉·狄金森的诗中返回一行的单词 “frog”：

```
$ cargo run -- frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.38s
```

```
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

不错！现在试试一个会匹配多行的单词，比如 “body”：

```
$ cargo run -- body poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

最后，让我们确保搜索一个在诗中哪里都没有的单词时不会得到任何行，比如 *monomorphization*：

```
$ cargo run -- monomorphization poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep monomorphization poem.txt`
```

非常好！我们创建了一个属于自己的迷你版经典工具，并学习了很多如何组织程序的知识。我们还学习了一些文件输入输出、生命周期、测试和命令行解析的内容。

为了使这个项目更丰满，我们将简要的展示如何处理环境变量和打印到标准错误，这两者在编写命令行程序时都很有用。

处理环境变量

我们将增加一个额外的功能来改进 `minigrep`：用户可以通过设置环境变量来设置搜索是否是大小写敏感的选项。当然，我们也可以将其设计为一个命令行参数并要求用户每次需要时都加上它，不过在这里我们将使用环境变量。这允许用户设置环境变量一次之后在整个终端会话中所有的搜索都将是大小写不敏感的。

编写一个大小写不敏感 `search` 函数的失败测试

首先我们希望增加一个新函数 `search_case_insensitive`，并将会在环境变量有值时调用它。这里将继续遵循 TDD 过程，其第一步是再次编写一个失败测试。我们将为新的大小写不敏感搜索函数新增一个测试函数，并将老的测试函数从 `one_result` 改名为 `case_sensitive` 来更清楚的表明这两个测试的区别，如示例 12-20 所示：

文件名：src/lib.rs



```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

示例 12-20：为准备添加的大小写不敏感函数新增失败测试

注意我们也改变了老测试中 `contents` 的值。还新增了一个含有文本 `"Duct tape."` 的行，它有一个大写的 D，这在大小写敏感搜索时不应该匹配 `"duct"`。我们修改这个测试以确保不会意外破坏已经实现的大小写敏感搜索功能；这个测试现在应该能通过并在处理大小写不敏感搜索时应该能一直通过。

大小写**不敏感**搜索的新测试使用 `"rUsT"` 作为其查询字符串。在我们将要增加的 `search_case_insensitive` 函数中，`"rUsT"` 查询应该包含带有一个大写 R 的 `"Rust:"` 还有 `"Trust me."` 这两行，即便它们与查询的大小写都不同。这个测试现在不能编译，因为还没有定义 `search_case_insensitive` 函数。请随意增加一个总是返回空 vector 的骨架实现，正如示例 12-16 中 `search` 函数为了使测试通过编译并失败时所做的那样。

实现 `search_case_insensitive` 函数

`search_case_insensitive` 函数，如示例 12-21 所示，将与 `search` 函数基本相同。唯一的区别是它会将 `query` 变量和每一 `line` 都变为小写，这样不管输入参数是大写还是小写，在检查该行是否包含查询字符串时都会是小写。

文件名：src/lib.rs

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

示例 12-21：定义 `search_case_insensitive` 函数，它在比较查询和每一行之前将它们都转换为小写

首先我们将 `query` 字符串转换为小写，并将其覆盖到同名的变量中，遮蔽原来的 `query`。对查询字符串调用 `to_lowercase` 是必需的，这样不管用户的查询是 `"rust"`、`"RUST"`、`"Rust"` 或者 `"rUsT"`，我们都将其当作 `"rust"` 处理并对大小写不敏感。虽然 `to_lowercase` 可以处理基本的 Unicode，但它不是 100% 准确。如果编写真实的程序的话，我们还需多做一些工作，不过这一部分是关于环境变量而不是 Unicode 的，所以这样就够了。

注意 `query` 现在是一个 `String` 而不是字符串 slice，因为调用 `to_lowercase` 是在创建新数据，而不是引用现有数据。如果查询字符串是 `"rUsT"`，这个字符串 slice 并不包含可供我们使用的小写的 `u` 或 `t`，所以必需分配一个包含 `"rust"` 的新 `String`。现在当我们将 `query` 作为一个参数传递给 `contains` 方法时，需要增加一个 & 因为 `contains` 的签名被定义为获取一个字符串 slice。

接下来我们对每一 `line` 都调用 `to_lowercase` 将其转为小写。现在我们将 `line` 和 `query` 都转换成了小写，这样就可以不管查询的大小写进行匹配了。

让我们看看这个实现能否通过测试：

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
```

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.33s
     Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

太好了！测试都通过了。现在，让我们在 `run` 函数中实际调用新 `search_case_insensitive` 函数。首先，我们将在 `Config` 结构体中增加一个配置项来切换大小写敏感和大小写不敏感搜索。增加这些字段会导致编译错误，因为我们还没有在任何地方初始化这些字段：

文件名：src/lib.rs



```
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
```

这里增加了 `ignore_case` 字段来存放一个布尔值。接着我们需要 `run` 函数检查 `case_sensitive` 字段的值并使用它来决定是否调用 `search` 函数或 `search_case_insensitive` 函数，如示例 12-22 所示。注意这还不能编译：

文件名：src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{line}");
    }
```

```
    Ok(())
}
```



示例 12-22：根据 `config.ignore_case` 的值调用 `search` 或 `search_case_insensitive`

最后需要实际检查环境变量。处理环境变量的函数位于标准库的 `env` 模块中，所以我们需要在 *src/lib.rs* 的开头将这个模块引入作用域中。接着使用 `env` 模块的 `var` 方法来检查一个叫做 `IGNORE_CASE` 的环境变量，如示例 12-23 所示：

文件名：src/lib.rs

```
use std::env;
// --snip--

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

示例 12-23：检查叫做 `IGNORE_CASE` 的环境变量

这里创建了一个新变量 `ignore_case`。为了设置它的值，需要调用 `env::var` 函数并传递我们需要寻找的环境变量名称，`IGNORE_CASE`。`env::var` 返回一个 `Result`，它在环境变量被设置时返回包含其值的 `Ok` 变体，并在环境变量未被设置时返回 `Err` 变体。

我们使用 `Result` 的 `is_ok` 方法来检查环境变量是否被设置，这也就意味着程序应该进行一个大小写不敏感的搜索。如果 `IGNORE_CASE` 环境变量没有被设置为任何值，`is_ok` 会返回 `false` 并将进行大小写敏感的搜索。我们并不关心环境变量所设置的**值**，只关心它是否被设置了，所以检查 `is_ok` 而不是 `unwrap`、`expect` 或任何我们已经见过的 `Result` 的方法。

我们将变量 `ignore_case` 的值传递给 `Config` 实例，这样 `run` 函数可以读取其值并决定是否调用示例 12-22 中实现的 `search_case_insensitive` 或者 `search`。

让我们试一试吧！首先不设置环境变量并使用查询 `to` 运行程序，这应该会匹配任何全小写的单词 “to” 的行：

```
$ cargo run -- to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

看起来程序仍然能够工作！现在将 `IGNORE_CASE` 设置为 `1` 并仍使用相同的查询 *to* 来运行程序：

```
$ IGNORE_CASE=1 cargo run -- to poem.txt
```

如果你使用 PowerShell，则需要用两个命令来分别设置环境变量并运行程序：

```
PS> $Env:IGNORE_CASE=1; cargo run -- to poem.txt
```

而这会让 `IGNORE_CASE` 的效果在当前 shell 会话中持续生效。可以通过 `Remove-Item` cmdlet 来取消设置：

```
PS> Remove-Item Env:IGNORE_CASE
```

这回应该得到包含可能有大写字母的 *to* 的行：

```
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

好极了，我们也得到了包含 *to* 的行！现在 `minigrep` 程序可以通过环境变量控制进行大小写不敏感搜索了。现在你知道了如何管理由命令行参数或环境变量设置的选项了！

一些程序允许对相同配置同时使用参数**和**环境变量。在这种情况下，程序来决定参数和环境变量的优先级。作为一个留给你的测试，尝试通过一个命令行参数或一个环境变量来控制大小写敏感搜索。并在运行程序时遇到矛盾值时决定命令行参数和环境变量的优先级。

`std::env` 模块还包含了更多处理环境变量的实用功能；请查看其文档来了解其可用的功能。

将错误信息输出到标准错误而不是标准输出

目前为止，我们将所有的输出都通过 `println!` 写到了终端。大部分终端都提供了两种输出：**标准输出**（*standard output*，`stdout`）对应一般信息，**标准错误**（*standard error*，`stderr`）则用于错误信息。这种区别允许用户选择将程序正常输出定向到一个文件中并仍将错误信息打印到屏幕上。

但是 `println!` 宏只能够打印到标准输出，所以我们必须使用其他方法来打印到标准错误。

检查错误写入何处

首先，让我们观察一下目前 `minigrep` 打印的所有内容是如何被写入标准输出的，包括那些应该被写入标准错误的错误信息。可以通过将标准输出流重定向到一个文件同时有意产生一个错误来做到这一点。我们没有重定向标准错误流，所以任何发送到标准错误的内容将会继续显示在屏幕上。

命令行程序被期望将错误信息发送到标准错误流，这样即便选择将标准输出流重定向到文件中时仍然能看到错误信息。目前我们的程序并不符合期望；相反我们将看到它将错误信息输出保存到了文件中！

我们通过 `>` 和文件路径 *output.txt* 来运行程序，我们期望重定向标准输出流到该文件中。在这里，我们没有传递任何参数，所以会产生一个错误：

```
$ cargo run > output.txt
```

`>` 语法告诉 shell 将标准输出的内容写入到 *output.txt* 文件中而不是屏幕上。我们并没有看到期望的错误信息打印到屏幕上，所以这意味着它一定被写入了文件中。如下是 *output.txt* 所包含的：

```
Problem parsing arguments: not enough arguments
```

是的，错误信息被打印到了标准输出中。像这样的错误信息被打印到标准错误中将会有用得多，这将使得只有成功运行所产生的输出才会写入文件。我们接下来就修改。

将错误打印到标准错误

让我们如示例 12-24 所示的代码改变错误信息是如何被打印的。得益于本章早些时候的重构，所有打印错误信息的代码都位于 `main` 一个函数中。标准库提供了 `eprintln!` 宏来打印到标准错误流，所以将两个调用 `println!` 打印错误信息的位置替换为 `eprintln!`：

文件名：src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
```

```
        eprintln!("Application error: {e}");
        process::exit(1);
    }
}
```

示例 12-24：使用 `eprintln!` 将错误信息写入标准错误而不是标准输出

现在我们再次尝试用同样的方式运行程序，不使用任何参数并通过 `>` 重定向标准输出：

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

现在我们看到了屏幕上的错误信息，同时 *output.txt* 里什么也没有，这正是命令行程序所期望的行为。

如果使用不会造成错误的参数再次运行程序，不过仍然将标准输出重定向到一个文件，像这样：

```
$ cargo run -- to poem.txt > output.txt
```

我们并不会在终端看到任何输出，同时 `output.txt` 将会包含其结果：

文件名：output.txt

```
Are you nobody, too?
How dreary to be somebody!
```

这一部分展示了现在我们适当地使用了成功时产生的标准输出和错误时产生的标准错误。

总结

在这一章中，我们回顾了目前为止的一些主要章节并涉及了如何在 Rust 环境中进行常规的 I/O 操作。通过使用命令行参数、文件、环境变量和打印错误的 `eprintln!` 宏，现在你已经准备好编写命令行程序了。通过结合前几章的知识，你的代码将会是组织良好的，并能有效的将数据存储到合适的数据结构中、更好的处理错误，并且还是经过良好测试的。

接下来，让我们探索一些 Rust 中受函数式编程语言影响的功能：闭包和迭代器。

函数式语言特性：迭代器与闭包

Rust 的设计灵感来源于很多现存的语言和技术。其中一个显著的影响就是**函数式编程**（*functional programming*）。函数式编程风格通常包含将函数作为参数值或其他函数的返回值、将函数赋值给变量以供之后执行等等。

本章我们不会讨论函数式编程是或不是什么的问题，而是展示 Rust 的一些在功能上与其他被认为是函数式语言类似的特性。

更具体地，我们将要涉及：

- **闭包**（*Closures*），一个可以储存在变量里的类似函数的结构
- **迭代器**（*Iterators*），一种处理元素序列的方式
- 如何使用闭包和迭代器来改进第十二章的 I/O 项目。
- 闭包和迭代器的性能。（**剧透警告：** 它们的速度超乎你的想象！）

我们已经介绍了其它受函数式风格影响的 Rust 功能，比如模式匹配和枚举，这些已经在其他章节中讲到过了。因为掌握闭包和迭代器是编写符合语言风格的高性能 Rust 代码的重要一环，所以我们将专门用一整章来讲解它们。

闭包：可以捕获环境的匿名函数

Rust 的 **闭包**（*closures*）是可以保存在变量中或作为参数传递给其他函数的匿名函数。你可以在一个地方创建闭包，然后在不同的上下文中执行闭包运算。不同于函数，闭包允许捕获其被定义时所在作用域中的值。我们将展示这些闭包特性如何支持代码复用和行为定制。

使用闭包捕获环境

我们首先了解如何通过闭包捕获定义它的环境中的值以便之后使用。考虑如下场景：我们的 T 恤公司偶尔会向邮件列表中的某位成员赠送一件限量版的独家 T 恤作为促销。邮件列表中的成员可以选择将他们的喜爱的颜色添加到个人信息中。如果被选中的成员设置了喜爱的颜色，他们将获得那个颜色的 T 恤。如果他没有设置喜爱的颜色，他们会获赠公司当前库存最多的颜色的款式。

有很多种方式来实现这一点。例如，使用有 `Red` 和 `Blue` 两个变体的 `ShirtColor` 枚举（出于简单考虑限定为两种颜色）。我们使用 `Inventory` 结构体来代表公司的库存，它有一个类型为 `Vec<ShirtColor>` 的 `shirts` 字段表示库存中的衬衫的颜色。`Inventory` 上定义的 `giveaway` 方法获取免费衬衫得主所喜爱的颜色（如有），并返回其获得的衬衫的颜色。初始代码如示例 13-1 所示：

文件名：src/main.rs

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}
```

```
fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref2, giveaway2
    );
}
```

示例 13-1：衬衫公司赠送场景

`main` 函数中定义的 `store` 还剩下两件蓝衬衫和一件红衬衫，可以在限量版促销活动中赠送。我们通过调用 `giveaway` 方法，为一个期望红衬衫的用户和一个没有特定偏好的用户进行赠送。

再次强调，这段代码有多种实现方式。这里为了专注于闭包，我们继续使用已经学习过的概念，除了 `giveaway` 方法体中使用了闭包。在 `giveaway` 方法中，我们将用户偏好作为 `Option<ShirtColor>` 类型的参数获取，并在 `user_preference` 上调用 `unwrap_or_else` 方法。`Option<T>` 上的 `unwrap_or_else` 方法由标准库定义。它接受一个无参闭包作为参数，该闭包返回一个 `T` 类型的值（与 `Option<T>` 的 `Some` 变体中存储的值类型相同，这里是 `ShirtColor`）。如果 `Option<T>` 是 `Some` 变体，则 `unwrap_or_else` 返回 `Some` 中的值。如果 `Option<T>` 是 `None` 变体，则 `unwrap_or_else` 调用闭包并返回闭包的返回值。

我们将闭包表达式 `|| self.most_stocked()` 作为 `unwrap_or_else` 的参数。这是一个本身不获取参数的闭包（如果闭包有参数，它们会出现在两道竖杠之间）。闭包体调用了 `self.most_stocked()`。我们在这里定义了闭包，而 `unwrap_or_else` 的实现会在之后需要其结果的时候执行闭包。

运行代码会打印出：

```
$ cargo run
   Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/shirt-company`
The user with preference Some(Red) gets Red
The user with preference None gets Blue
```

这里有一个有趣的地方是，我们传递了一个闭包，该闭包会在当前的 `Inventory` 实例上调用 `self.most_stocked()` 方法。标准库不需要了解我们定义的 `Inventory` 或 `ShirtColor` 类型，也不需要了解我们在这个场景中要使用的逻辑。闭包捕获了对 `self`（即 `Inventory` 实例）的不可变引用，并将其与我们指定的代码一起传递给 `unwrap_or_else` 方法。相比之下，函数无法以这种方式捕获其环境。

闭包类型推断和注解

函数与闭包还有更多区别。闭包通常不要求像 `fn` 函数那样对参数和返回值进行类型注解。函数需要类型注解是因为这些类型是暴露给用户的显式接口的一部分。严格定义这些接口对于确保所有人对函数使用和返回值的类型达成一致理解非常重要。与此相比，闭包并不用于这样暴露在外的接口：它们储存在变量中并被使用，不用命名它们或暴露给库的用户调用。

闭包通常较短，并且只与特定的上下文相关，而不是适用于任意情境。在这些有限的上下文中，编译器可以推断参数和返回值的类型，类似于它推断大多数变量类型的方式（尽管在某些罕见的情况下，编译器也需要闭包的类型注解）。

类似于变量，如果我们希望增加代码的明确性和清晰度，可以添加类型注解，但代价是会使代码变得比严格必要的更冗长。为示例 13-1 中定义的闭包标注类型看起来如示例 13-2 中的定义一样。这个例子中，我们定义了一个闭包并将它保存在变量中，而不是像示例 13-1 那样在传参的地方定义它。

文件名：src/main.rs

```
    let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };
```

示例 13-2：为闭包的参数和返回值增加可选的类型注解

有了类型注解，闭包的语法看起来就更像函数的语法了。如下是一个对其参数加一的函数的定义与拥有相同行为闭包语法的纵向对比。这里增加了一些空格来对齐相应部分。这展示了除了使用管道符（|，pipes）以及一些可选语法外，闭包语法与函数语法有多么地相似：

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

第一行展示了一个函数定义，第二行展示了一个完整标注的闭包定义。第三行闭包定义中省略了类型注解，而第四行去掉了可选的大括号，因为闭包体只有一个表达式，所以大括号是可选的。这些都是有效的闭包定义，并在调用时产生相同的行为。调用闭包是 `add_one_v3` 和 `add_one_v4` 能够编译的必要条件，因为类型将从其用法中推断出来。这类似于 `let v = Vec::new();`，Rust 需要类型注解或是某种类型的值被插入到 `Vec` 中，才能推断其类型。

对于闭包定义，编译器会为每个参数和返回值推断出一个具体类型。例如，示例 13-3 展示了一个简短的闭包定义，该闭包仅仅返回作为参数接收到的值。除了作为示例用途外，这个闭包并不是很实用。注意这个定义没有增加任何类型注解。因为没有类型注解，我们可以使用任意类型来调用这个闭包，我们在这里第一次调用时使用了 `String` 类型。但是如果我们接着尝试使用整数来调用 `example_closure`，就会得到一个错误。

文件名：src/main.rs

```
    let example_closure = |x| x;
```

```
    let s = example_closure(String::from("hello"));
    let n = example_closure(5);
```



示例 13-3：尝试调用一个被推断为两个不同类型的闭包

编译器给出如下错误：

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion method:
`.to_string()`
  |             |               |
  |             |               expected `String`, found integer
  |             arguments to this function are incorrect
  |
note: expected because the closure was earlier called with an argument of type
`String`
 --> src/main.rs:4:29
  |
4 |     let s = example_closure(String::from("hello"));
  |             --------------- ^^^^^^^^^^^^^^^^^^^^^ expected because this
argument is of type `String`
  |             |
  |             in this closure call
note: closure parameter defined here
 --> src/main.rs:2:28
  |
2 |     let example_closure = |x| x;
  |                            ^

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure-example` (bin "closure-example") due to 1
previous error
```

第一次使用 `String` 值调用 `example_closure` 时，编译器推断出 `x` 的类型以及闭包的返回类型为 `String`。接着这些类型被锁定进闭包 `example_closure` 中，如果尝试对同一闭包使用不同类型则就会得到类型错误。

捕获引用或移动所有权

闭包可以通过三种方式捕获其环境中的值，它们直接对应到函数获取参数的三种方式：不可变借用、可变借用和获取所有权。闭包将根据函数体中对捕获值的操作来决定使用哪种方式。

在示例 13-4 中定义了一个捕获名为 `list` 的 vector 的不可变引用的闭包，因为只需不可变引用就能打印其值：

文件名：src/main.rs

```
```

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let only_borrows = || println!("From closure: {list:?}");

    println!("Before calling closure: {list:?}");
    only_borrows();
    println!("After calling closure: {list:?}");
}
```

示例 13-4：定义并调用一个捕获不可变引用的闭包

这个示例也展示了变量可以绑定一个闭包定义，并且我们可以像使用函数名一样，使用变量名和括号来调用该闭包。

因为同时可以有多个 `list` 的不可变引用，所以在闭包定义之前，闭包定义之后调用之前，闭包调用之后代码仍然可以访问 `list`。该代码可以编译、运行并输出：

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

接下来在示例 13-5 中，我们修改闭包体让它向 `list` vector 增加一个元素。闭包现在捕获一个可变引用：

文件名：src/main.rs

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {list:?}");
}
```

示例 13-5：定义并调用一个捕获可变引用的闭包

代码可以编译、运行并打印：

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

注意在 `borrows_mutably` 闭包的定义和调用之间不再有 `println!`，这是因为当 `borrows_mutably` 被定义时，它捕获了对 `list` 的可变引用。闭包在被调用后就不再被使用，这时可变借用结束。因为当可变借用存在时不允许有其它的借用，所以在闭包定义和调用之间不能有不可变引用来进行打印。可以尝试在这里添加 `println!` 看看你会得到什么报错信息！

即使闭包体不严格需要所有权，如果希望强制闭包获取它在环境中所使用的值的所有权，可以在参数列表前使用 `move` 关键字。

当将闭包传递到一个新的线程时，这个技巧特别有用，因为它将数据的所有权移动到新线程中。我们将在第十六章讨论并发时详细讨论线程以及为什么你可能需要使用它们。不过现在，我们先简要探索一下如何使用需要 `move` 关键字的闭包来生成一个新线程。示例 13-6 展示了如何修改示例 13-4，以便在一个新线程中而不是在主线程中打印 vector：

文件名：src/main.rs

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    thread::spawn(move || println!("From thread: {list:?}"))
        .join()
        .unwrap();
}
```

示例 13-6：使用 `move` 来强制闭包为线程获取 `list` 的所有权

我们生成了一个新的线程，并给这个线程传递一个闭包作为参数来运行，闭包体打印出列表。在示例 13-4 中，闭包仅通过不可变引用捕获了 `list`，因为这是打印列表所需的最小访问权限。这个例子中，尽管闭包体依然只需要不可变引用，我们还是在闭包定义前写上 `move` 关键字，以确保 `list` 被移动到闭包中。新线程可能在主线程剩余部分执行完前执行完，也可能在主线程执行完之后执行完。如果主线程维护了 `list` 的所有权但却在新线程之前结束并且丢弃了 `list`，则在线程中的不可变引用将失效。因此，编译器要求 `list` 被移动到在新线程中运行的闭包中，这样引用就是有效的。试着移除 `move` 关键字，或者在闭包定义后在主线程中使用 `list`，看看你会得到什么编译器报错！

将捕获的值移出闭包和 `Fn` trait

一旦闭包捕获了定义它的环境中的某个值的引用或所有权（也就影响了什么会被移**进**闭包，如有），闭包体中的代码则决定了在稍后执行闭包时，这些引用或值将如何处理（也就影响了什么会被移**出**闭包，如有）。闭包体可以执行以下任一操作：将一个捕获的值移出闭包，修改捕获的值，既不移动也不修改值，或者一开始就不从环境中捕获任何值。

闭包捕获和处理环境中的值的方式会影响闭包实现哪些 trait，而 trait 是函数和结构体指定它们可以使用哪些类型闭包的方式。根据闭包体如何处理这些值，闭包会自动、渐进地实现一个、两个或全部三个 `Fn` trait。

1. `FnOnce` 适用于只能被调用一次的闭包。所有闭包至少都实现了这个 trait，因为所有闭包都能被调用。一个会将捕获的值从闭包体中移出的闭包只会实现 `FnOnce` trait，而不会实现其他 `Fn` 相关的 trait，因为它只能被调用一次。

2. `FnMut` 适用于不会将捕获的值移出闭包体，但可能会修改捕获值的闭包。这类闭包可以被调用多次。
3. `Fn` 适用于既不将捕获的值移出闭包体，也不修改捕获值的闭包，同时也包括不从环境中捕获任何值的闭包。这类闭包可以被多次调用而不会改变其环境，这在会多次并发调用闭包的场景中十分重要。

让我们来看示例 13-1 中使用的在 `Option<T>` 上的 `unwrap_or_else` 方法的定义：

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

回忆一下，`T` 是表示 `Option` 中 `Some` 变体中的值的类型的泛型。类型 `T` 也是 `unwrap_or_else` 函数的返回值类型：举例来说，在 `Option<String>` 上调用 `unwrap_or_else` 会得到一个 `String`。

接着注意到 `unwrap_or_else` 函数有额外的泛型参数 `F`。`F` 是参数 `f` 的类型，`f` 是调用 `unwrap_or_else` 时提供的闭包。

泛型 `F` 的 trait bound 是 `FnOnce() -> T`，这意味着 `F` 必须能够被调用一次，没有参数并返回一个 `T`。在 trait bound 中使用 `FnOnce` 表示 `unwrap_or_else` 最多只会调用 `f` 一次。在 `unwrap_or_else` 的函数体中可以看到，如果 `Option` 是 `Some`，`f` 不会被调用。如果 `Option` 是 `None`，`f` 将会被调用一次。由于所有的闭包都实现了 `FnOnce`，`unwrap_or_else` 接受所有三种类型的闭包，灵活性达到极致。

> 注意：如果我们要做的事情不需要从环境中捕获值，则可以在需要某种实现了 `Fn` trait 的东西时使用函数而不是闭包。举个例子，可以在 `Option<Vec<T>>` 的值上调用 `unwrap_or_else(Vec::new)`，以便在值为 `None` 时获取一个新的空的 vector。编译器会自动为函数定义实现适用的 `Fn` trait。

现在让我们来看定义在 slice 上的标准库方法 `sort_by_key`，看看它与 `unwrap_or_else` 的区别，以及为什么 `sort_by_key` 使用 `FnMut` 而不是 `FnOnce` 作为 trait bound。这个闭包以一个 slice 中当前被考虑的元素的引用作为参数，并返回一个可以排序的 `K` 类型的值。当你想按照 slice 中每个元素的某个属性进行排序时，这个函数非常有用。在示例 13-7 中，我们有一个 `Rectangle` 实例的列表，并使用 `sort_by_key` 按 `Rectangle` 的 `width` 属性对它们从低到高排序：

文件名：src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{list:#?}");
}
```

示例 13-7：使用 `sort_by_key` 对长方形按宽度排序

代码输出：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/rectangles`
[
    Rectangle {
        width: 3,
        height: 5,
    },
    Rectangle {
        width: 7,
        height: 12,
    },
    Rectangle {
        width: 10,
        height: 1,
    },
]
```

`sort_by_key` 被定义为接收一个 `FnMut` 闭包的原因是它会多次调用这个闭包：对 slice 中的每个元素调用一次。闭包 `|r| r.width` 不捕获、修改或将任何东西移出它的环境，所以它满足 trait bound 的要求。

相比之下，示例 13-8 展示了一个只实现了 `FnOnce` trait 的闭包的例子，因为它从环境中移出了一个值。编译器不允许我们在 `sort_by_key` 中使用这个闭包：

文件名：src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
```



```
fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("closure called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{list:#?}");
}
```

示例 13-8：尝试在 `sort_by_key` 上使用一个 `FnOnce` 闭包

这是一个刻意构造的、复杂且无效的方式，试图统计在对 `list` 进行排序时 `sort_by_key` 调用闭包的次数。该代码试图通过将闭包环境中的 `value`（一个 `String`）插入 `sort_operations` vector 来实现计数。闭包捕获了 `value`，然后通过将 `value` 的所有权转移给 `sort_operations` vector 的方式将其移出闭包。这个闭包只能被调用一次；尝试第二次调用它将无法工作，因为这时 `value` 已经不在闭包的环境中，无法被再次插入 `sort_operations` 中！因而，这个闭包只实现了 `FnOnce`。当我们尝试编译此代码时，会出现错误提示：`value` 不能从闭包中移出，因为闭包必须实现 `FnMut`：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut`
closure
  --> src/main.rs:18:30
   |
15 |     let value = String::from("closure called");
   |         ----- captured outer variable
16 |
17 |     list.sort_by_key(|r| {
   |                      --- captured by this `FnMut` closure
18 |         sort_operations.push(value);
   |                              ^^^^^ move occurs because `value` has type
`String`, which does not implement the `Copy` trait
   |
help: consider cloning the value if the performance cost is acceptable
   |
18 |         sort_operations.push(value.clone());
   |                                    ++++++++

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` (bin "rectangles") due to 1 previous error
```

报错指向了闭包体中将 `value` 移出环境的那一行。要修复此问题，我们需要修改闭包体，使其不会将值移出环境。在环境中维护一个计数器，并在闭包体中递增其值，是计算闭包被调用次

数的一个更直观的方法。示例 13-9 中的闭包可以在 `sort_by_key` 中使用，因为它只捕获了 `num_sort_operations` 计数器的可变引用，因此可以被多次调用：

文件名：src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{list:#?}, sorted in {num_sort_operations} operations");
}
```

示例 13-9：允许在 `sort_by_key` 上使用一个 `FnMut` 闭包

当定义或使用涉及闭包的函数或类型时，`Fn` trait 十分重要。在下个小节中，我们将讨论迭代器。许多迭代器方法都接收闭包参数，因此在继续前，请记住这些闭包的细节！

使用迭代器处理元素序列

迭代器模式允许你依次对一个序列中的项执行某些操作。**迭代器**（*iterator*）负责遍历序列中的每一项并确定序列何时结束的逻辑。使用迭代器时，你无需自己重新实现这些逻辑。

在 Rust 中，迭代器是**惰性的**（*lazy*），这意味着在调用消费迭代器的方法之前不会执行任何操作。例如，示例 13-10 中的代码通过调用定义于 `Vec<T>` 上的 `iter` 方法在一个 vector `v1` 上创建了一个迭代器。这段代码本身并没有执行任何有用的操作。

```
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();
```

示例 13-10：创建一个迭代器

迭代器被储存在 `v1_iter` 变量中。一旦创建迭代器之后，可以选择用多种方式利用它。在第三章的示例 3-5 中，我们使用 `for` 循环来遍历一个数组并在每一个项上执行了一些代码。在底层它隐式地创建并接着消费了一个迭代器，不过直到现在我们都一笔带过了它具体是如何工作的。

示例 13-11 中的例子将迭代器的创建和 `for` 循环中的使用分开。当 `for` 循环使用 `v1_iter` 中的迭代器时，迭代器中的每一个元素都会用于循环的一次迭代，并打印出每个值。

```
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {val}");
    }
```

示例 13-11：在一个 `for` 循环中使用迭代器

在标准库中没有提供迭代器的语言中，我们可能会使用一个从 0 开始索引的变量，使用这个变量索引 vector 中的值，并循环增加其值直到达到 vector 中的元素总量，以实现相同的功能。

迭代器为我们处理了所有这些逻辑，这减少了重复代码并消除了潜在的混乱。另外，迭代器的实现方式提供了对多种不同的序列使用相同逻辑的灵活性，而不仅仅是像 vector 这样可索引的数据结构。让我们看看迭代器是如何做到这些的。

`Iterator` trait 和 `next` 方法

迭代器都实现了名为 `Iterator` 的定义于标准库的 trait。这个 trait 的定义看起来像这样：

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // 此处省略了方法的默认实现
}
```

注意这里有一个我们还未讲到的新语法：`type Item` 和 `Self::Item`，它们定义了 trait 的**关联类型**（*associated type*）。第二十章会深入讲解关联类型，不过现在只需知道这段代码表明实现 `Iterator` trait 要求同时定义一个 `Item` 类型，这个 `Item` 类型被用作 `next` 方法的返回值类型。换句话说，`Item` 类型将是迭代器返回元素的类型。

`Iterator` trait 仅要求实现者定义一个方法：`next` 方法，该方法每次返回迭代器中的一个项，封装在 `Some` 中，并且当迭代完成时，返回 `None`。

可以直接调用迭代器的 `next` 方法；示例 13-12 展示了对由 vector 创建的迭代器重复调用 `next` 方法时返回的值。

文件名：src/lib.rs

```
    #[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];

        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

示例 13-12：在迭代器上（直接）调用 `next` 方法

注意我们需要将 `v1_iter` 声明为可变的：在迭代器上调用 `next` 方法会改变迭代器内部的状态，该状态用于跟踪迭代器在序列中的位置。换句话说，代码**消费**（consume）了，或者说用尽了迭代器。每一次 `next` 调用都会从迭代器中消费一个项。使用 `for` 循环时无需使 `v1_iter` 可变因为 `for` 循环会获取 `v1_iter` 的所有权并在后台使 `v1_iter` 可变。

还需要注意的是，从 `next` 调用中获取的值是对 vector 中值的不可变引用。`iter` 方法生成一个不可变引用的迭代器。如果我们需要一个获取 `v1` 所有权并返回拥有所有权的迭代器，则可以调用 `into_iter` 而不是 `iter`。类似地，如果我们希望迭代可变引用，可以调用 `iter_mut` 而不是 `iter`。

消费迭代器的方法

`Iterator` trait 有一系列不同的由标准库提供默认实现的方法；你可以在 `Iterator` trait 的标准库 API 文档中找到所有这些方法。一些方法在其定义中调用了 `next` 方法，这也就是为什么在实现 `Iterator` trait 时要求实现 `next` 方法的原因。

这些调用 `next` 方法的方法被称为**消费适配器**（*consuming adaptors*），因为调用它们会消耗迭代器。一个消费适配器的例子是 `sum` 方法，这个方法获取迭代器的所有权并反复调用 `next` 来遍历迭代器，从而消费迭代器。在遍历过程中，它将每个项累加到一个运行时总和中，并在迭代完成时返回这个总和。示例 13-13 有一个展示 `sum` 方法使用的测试：

文件名：src/lib.rs

```
    #[test]
    fn iterator_sum() {
        let v1 = vec![1, 2, 3];
```

```
        let v1_iter = v1.iter();

        let total: i32 = v1_iter.sum();

        assert_eq!(total, 6);
    }
```

示例 13-13：调用 `sum` 方法获取迭代器所有项的总和

调用 `sum` 之后不再允许使用 `v1_iter` 因为调用 `sum` 时它会获取迭代器的所有权。

产生其他迭代器的方法

`Iterator` trait 中定义了另一类方法，被称为**迭代器适配器**（*iterator adaptors*），它们不会消耗当前的迭代器，而是通过改变原始迭代器的某些方面来生成不同的迭代器。

示例 13-14 展示了一个调用迭代器适配器方法 `map` 的例子，该方法使用一个闭包对每个元素进行操作。`map` 方法返回一个新的迭代器，该迭代器生成经过修改的元素。这里的闭包创建了一个新的迭代器，其中 vector 中的每个元素都被加 1。

文件名：src/main.rs

```
    let v1: Vec<i32> = vec![1, 2, 3];

    v1.iter().map(|x| x + 1);
```



示例 13-14：调用迭代器适配器 `map` 来创建一个新迭代器

不过这些代码会产生一个警告：

```
$ cargo run
   Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: iterators are lazy and do nothing unless consumed
  = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
  |
4 |     let _ = v1.iter().map(|x| x + 1);
  |     +++++++

warning: `iterators` (bin "iterators") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

示例 13-14 中的代码实际上并没有做任何事；所指定的闭包从未被调用过。警告提醒了我们原因所在：迭代器适配器是惰性的，因此我们需要在此处消费迭代器。

为了修复这个警告并消费迭代器，我们将使用第十二章示例 12-1 结合 `env::args` 使用的 `collect` 方法。这个方法消费迭代器并将结果收集到一个集合数据类型中。

在示例 13-15 中，我们将遍历由 `map` 调用生成的迭代器结果收集到一个 vector 中。这个 vector 将包含原始 vector 中每个元素加 1 的结果。

文件名：src/main.rs

```
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

    assert_eq!(v2, vec![2, 3, 4]);
```

示例 13-15：调用 `map` 方法创建一个新迭代器，接着调用 `collect` 方法消费新迭代器并创建一个 vector

由于 `map` 接受一个闭包，因此我们可以指定希望在每个元素上执行的任何操作。这是一个很好的例子，展示了如何通过闭包来自定义某些行为，同时复用 `Iterator` trait 提供的迭代行为。

可以链式调用多个迭代器适配器来以一种可读的方式进行复杂的操作。不过因为所有的迭代器都是惰性的，你必须调用一个消费适配器方法，才能从这些迭代器适配器的调用中获取结果。

使用捕获其环境的闭包

很多迭代器适配器接受闭包作为参数，而我们通常会指定捕获其环境的闭包作为迭代器适配器的参数。

作为一个例子，我们使用 `filter` 方法来获取一个闭包。该闭包从迭代器中获取一项并返回一个 `bool`。如果闭包返回 `true`，其值将会包含在 `filter` 提供的新迭代器中。如果闭包返回 `false`，其值不会被包含。

示例 13-16 中使用 `filter` 和一个捕获环境中变量 `shoe_size` 的闭包来遍历一个 `Shoe` 结构体集合。它只会返回指定鞋码的鞋子。

文件名：src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
```

```
                size: 10,
                style: String::from("sneaker"),
            },
            Shoe {
                size: 13,
                style: String::from("sandal"),
            },
            Shoe {
                size: 10,
                style: String::from("boot"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
    }
}
```

示例 13-16：使用 `filter` 方法和一个捕获 `shoe_size` 的闭包

`shoes_in_size` 函数获取一个鞋子 vector 的所有权和一个鞋码作为参数。它返回一个只包含指定鞋码的鞋子的 vector。

`shoes_in_size` 函数体中调用了 `into_iter` 来创建一个获取 vector 所有权的迭代器。接着调用 `filter` 将这个迭代器适配成一个只含有那些闭包返回 `true` 的元素的新迭代器。

闭包从环境中捕获了 `shoe_size` 变量并使用其值与每一只鞋的大小作比较，只保留指定鞋码的鞋子。最终，调用 `collect` 将迭代器适配器返回的值收集进一个 vector 并返回。

这个测试展示当调用 `shoes_in_size` 时，返回的只会是与我们指定的鞋码相同的鞋子。

改进 I/O 项目

掌握了这些关于迭代器的新知识后，我们可以使用迭代器来改进第十二章中 I/O 项目的实现来使得代码更简洁明了。接下来，让我们看看迭代器如何改进 `Config::build` 函数和 `search` 函数的实现。

使用迭代器消除 `clone`

在示例 12-6 中，我们增加了一些代码获取一个 `String` 类型的 slice 并创建一个 `Config` 结构体的实例，它们索引 slice 中的值并克隆这些值以便 `Config` 结构体可以拥有这些值。在示例 13-17 中重现了第十二章结尾示例 12-23 中 `Config::build` 函数的实现：

文件名：src/lib.rs

```
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

示例 13-17：重现示例 12-23 的 `Config::build` 函数

当时我们说过不必担心低效的 `clone` 调用，因为我们以后会将其移除。好吧，就是现在！

起初这里需要 `clone` 的原因是参数 `args` 中有一个 `String` 元素的 slice，而 `build` 函数并不拥有 `args`。为了能够返回 `Config` 实例的所有权，我们不得不克隆 `Config` 中字段 `query` 和 `file_path` 的值，这样 `Config` 实例就能拥有这些值。

在学习了迭代器之后，我们可以将 `build` 函数改为获取一个有所有权的迭代器作为参数，而不是借用 slice。我们将使用迭代器功能代替之前检查 slice 长度和索引特定位置的代码。这样可以更清晰地表达 `Config::build` 函数的操作，因为迭代器会负责访问这些值。

一旦 `Config::build` 获取了迭代器的所有权并不再使用借用的索引操作，就可以将迭代器中的 `String` 值移动到 `Config` 中，而不是调用 `clone` 分配新的空间。

直接使用返回的迭代器

打开 I/O 项目的 *src/main.rs* 文件，它看起来应该像这样：

文件名：src/main.rs

```
```

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

首先我们修改第十二章结尾示例 12-24 中的 `main` 函数的开头为示例 13-18 中的代码。在更新 `Config::build` 之前这些代码还不能编译：

文件名：src/main.rs



```
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

示例 13-18：将 `env::args` 的返回值传递给 `Config::build`

`env::args` 函数返回一个迭代器！不同于将迭代器的值收集到一个 vector 中接着传递一个 slice 给 `Config::build`，现在我们直接将 `env::args` 返回的迭代器的所有权传递给 `Config::build`。

接下来需要更新 `Config::build` 的定义。在 I/O 项目的 *src/lib.rs* 中，将 `Config::build` 的签名改为如示例 13-19 所示。这仍然不能编译因为我们还需更新函数体。

文件名：src/lib.rs



```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        // --snip--
```

示例 13-19：以迭代器作为参数更新 `Config::build` 的签名

`env::args` 函数的标准库文档显示，它返回的迭代器的类型为 `std::env::Args`，并且这个类型实现了 `Iterator` trait 并返回 `String` 值。

我们已经更新了 `Config::build` 函数的签名，因此参数 `args` 有一个带有 trait bound `impl Iterator<Item = String>` 的泛型类型，而不是 `&[String]`。这里用到了第十章“trait 作为参数”部分讨论过的 `impl Trait` 语法，这意味着 `args` 可以是任何实现了 `Iterator` trait 并返回 `String` 项（item）的类型。

由于我们获取了 `args` 的所有权，并且将通过迭代来修改 `args`，因此我们可以在 `args` 参数的声明中添加 `mut` 关键字，使其可变。

使用 `Iterator` trait 方法代替索引

接下来，我们将修改 `Config::build` 的函数体。因为 `args` 实现了 `Iterator` trait，因此我们知道可以对其调用 `next` 方法！示例 13-20 更新了示例 12-23 中的代码，以使用 `next` 方法：

文件名：src/lib.rs

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

示例 13-20：修改 `Config::build` 的函数体来使用迭代器方法

请记住 `env::args` 返回值的第一个值是程序的名称。我们希望忽略它并获取下一个值，所以首先调用 `next` 且不对其返回值做任何操作。然后，我们再次调用 `next` 来获取要放入 `Config` 结构体的 `query` 字段的值。如果 `next` 返回 `Some`，使用 `match` 来提取其值。如果它返回 `None`，则意味着没有提供足够的参数并通过 `Err` 值提早返回。我们对 `file_path` 的值也进行同样的操作。

使用迭代器适配器让代码更清晰

I/O 项目中其他可以利用迭代器的地方是 `search` 函数，示例 13-21 中重现了第十二章结尾示例 12-19 中此函数的定义：

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
```

```
    }

    results
}
```

示例 13-21：示例 12-19 中 `search` 函数的定义

可以通过使用迭代器适配器方法来编写更简明的代码。这样做还可以避免使用一个可变的中间 `results` vector。函数式编程风格倾向于最小化可变状态的数量来使代码更清晰。去除可变状态可能会使未来的并行搜索优化变得更容易，因为我们不必管理对 `results` vector 的并发访问。示例 13-22 展示了这一变化：

文件名：src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

示例 13-22：在 `search` 函数实现中使用迭代器适配器

回忆一下，`search` 函数的目的是返回所有 `contents` 中包含 `query` 的行。类似于示例 13-16 中的 `filter` 例子，这段代码使用 `filter` 适配器来保留 `line.contains(query)` 返回 `true` 的行。接着使用 `collect` 将匹配行收集到另一个 vector 中。这样就容易多了！尝试对 `search_case_insensitive` 函数做出同样的使用迭代器方法的修改吧。

选择循环或迭代器

接下来的逻辑问题就是在代码中应该选择哪种风格，以及原因：是使用示例 13-21 中的原始实现还是使用示例 13-22 中使用迭代器的版本？大部分 Rust 程序员倾向于使用迭代器风格。开始这有点难以掌握，不过一旦你对不同迭代器的工作方式有了感觉之后，迭代器反而更容易理解。相比摆弄不同的循环并创建新 vector，（迭代器）代码则更关注循环的高层次目的。这抽象掉那些老生常谈的代码，这样就更容易看清代码所特有的概念，比如迭代器中每个元素必须满足的过滤条件。

不过这两种实现真的完全等价吗？直觉上的假设是更底层的循环会更快一些。让我们聊聊性能吧。

性能对比：循环 VS 迭代器

为了决定是否使用循环或迭代器，你需要了解哪个实现更快：使用显式 `for` 循环的 `search` 函数版本，还是使用迭代器的版本。

我们进行了一个基准测试，将阿瑟·柯南·道尔的《福尔摩斯探案集》全文加载到一个 `String` 中，并在内容中查找单词 *the*。以下是使用 `for` 循环版本和使用迭代器版本的 `search` 函数的基准测试结果：

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

两种实现具有相似的性能表现！这里我们不会解释性能测试的代码，我们的目的并不是为了证明它们是完全等同的，而是得出一个怎样比较这两种实现方式性能的基本思路。

对于一个更全面的性能测试，你应该使用不同大小的文本作为 `contents`，不同的单词以及长度各异的单词作为 `query`，以及各种其他变化进行检查。关键在于：迭代器，作为一个高级的抽象，被编译成了与手写的底层代码大体一致性能的代码。迭代器是 Rust 的**零成本抽象**（*zero-cost abstractions*）之一，它意味着抽象并不会引入额外的运行时开销，它与本贾尼·斯特劳斯特卢普（C++ 的设计和实现者）在《Foundations of C++》（2012）中所定义的**零开销**（*zero-overhead*）如出一辙：

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.
>
> 总的来说，C++ 的实现遵循了零开销原则：不使用的功能无需为其付出代价；而已经使用的功能，也不可能通过手写代码做得更好。

作为另一个例子，以下代码取自一个音频解码器。解码算法使用线性预测数学运算（linear prediction mathematical operation）来根据之前样本的线性函数预测将来的值。这些代码使用迭代器链对作用域中的三个变量进行某种数学计算：一个叫 `buffer` 的数据 slice、一个有 12 个元素的数组 `coefficients`、和一个代表位数据位移量的 `qlp_shift`。我们在这个例子中声明了这些变量，但没有为它们赋值；虽然这些代码在其上下文之外没有太多意义，不过仍是一个简明的现实例子，来展示 Rust 如何将高级概念转换为底层代码。

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

为了计算 `prediction` 的值，这段代码遍历了 `coefficients` 中的 12 个值，使用 `zip` 方法将系数与 `buffer` 的前 12 个值组合在一起。接着将每一对值相乘，再将所有结果相加，然后将总和右移 `qlp_shift` 位。

像音频解码器这样的程序通常最看重计算的性能。这里，我们创建了一个迭代器，使用了两个适配器，接着消费了其值。那么这段 Rust 代码将会被编译为什么样的汇编代码呢？好吧，在编写本书的这个时候，它被编译成与手写的相同的汇编代码。遍历 `coefficients` 的值完全用不到循环：Rust 知道这里会迭代 12 次，所以它“展开”（unroll）了循环。展开是一种将循环迭代转换为重复代码，并移除循环控制代码开销的代码优化技术。

所有的系数（coefficients）都被储存在了寄存器中，这意味着访问它们非常快。这里也没有运行时数组访问边界检查。所有这些 Rust 能够提供的优化使得结果代码极为高效。现在你知道了这些，请放心大胆的使用迭代器和闭包吧！它们使得代码看起来更高级，但并不为此引入运行时性能损失。

总结

闭包和迭代器是 Rust 受函数式编程语言观念所启发的特性。它们对 Rust 以高性能来明确的表达高级概念的能力有很大贡献。闭包和迭代器的实现达到了不影响运行时性能的程度。这正是 Rust 致力于提供零成本抽象的目标的一部分。

现在我们改进了 I/O 项目的（代码）表现力，那么让我们来看看 `cargo` 的更多功能，这些功能将帮助我们将项目分享给全世界。

进一步认识 Cargo 和 Crates.io

目前为止我们只使用过 Cargo 构建、运行和测试代码这些最基本的功能，不过它还可以做到更多。本章会讨论 Cargo 其他一些更为高级的功能，我们将展示如何：

- 使用发布配置（release profiles）来自定义构建
- 将库发布到 crates.io
- 使用工作空间（workspaces）来组织更大的项目
- 从 crates.io 安装二进制文件
- 使用自定义的命令来扩展 Cargo

Cargo 的功能不止本章所介绍的，关于其全部功能的详尽解释，请查看文档

采用发布配置自定义构建

在 Rust 中**发布配置**（*release profiles*）文件是预定义和可定制的，它们包含不同的配置，允许程序员更灵活地控制代码编译的多种选项。每一个配置都相互独立。

Cargo 有两个主要的配置：运行 `cargo build` 时采用的 `dev` 配置和运行 `cargo build --release` 的 `release` 配置。`dev` 配置为开发定义了良好的默认配置，`release` 配置则为发布构建定义了良好的默认配置。

这些配置名称可能很眼熟，因为它们出现在构建的输出中：

```
$ cargo build
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
$ cargo build --release
    Finished `release` profile [optimized] target(s) in 0.32s
```

构建输出中的 `dev` 和 `release` 表明编译器在使用不同的配置。

当项目的 *Cargo.toml* 文件中没有显式增加任何 `[profile.*]` 部分的时候，Cargo 会对每一个配置都采用默认设置。通过增加任何希望定制的配置对应的 `[profile.*]` 部分，我们可以选择覆盖任意默认设置的子集。例如，如下是 `dev` 和 `release` 配置的 `opt-level` 设置的默认值：

文件名：Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

`opt-level` 设置控制 Rust 会对代码进行何种程度的优化。这个配置的值从 0 到 3。越高的优化级别需要更多的时间编译，所以如果你在进行开发并经常编译，可能会希望在牺牲一些代码性能的情况下减少优化以便编译得快一些。因此 `dev` 的 `opt-level` 默认为 `0`。当你准备发布时，花费更多时间在编译上则更好。只需要在发布模式编译一次，而编译出来的程序则会运行很多次，所以发布模式用更长的编译时间换取运行更快的代码。这正是为什么 `release` 配置的 `opt-level` 默认为 `3`。

我们可以选择通过在 *Cargo.toml* 增加不同的值来覆盖任何默认设置。比如，如果我们想要在开发配置中使用级别 1 的优化，则可以在 *Cargo.toml* 中增加这两行：

文件名：Cargo.toml

```
[profile.dev]
opt-level = 1
```

这会覆盖默认的设置 `0`。现在运行 `cargo build` 时，Cargo 将会使用 `dev` 的默认配置加上定制的 `opt-level`。因为 `opt-level` 设置为 `1`，Cargo 会比默认进行更多的优化，但是没有发布构建那么多。

对于每个配置的设置和其默认值的完整列表，请参阅 Cargo 的文档。

将 crate 发布到 Crates.io

我们曾经在项目中使用 crates.io 上的包作为依赖，不过你也可以通过发布自己的包来向他人分享代码。crates.io 上的 crate 注册表会分发你包的源代码，因此它主要托管开源代码。

Rust 和 Cargo 有一些帮助他人更方便地找到和使用你发布的包的功能。我们将介绍一些这样的功能，接着讲到如何发布一个包。

编写有用的文档注释

准确的包文档有助于其他用户理解如何以及何时使用它们，所以花一些时间编写文档是值得的。第三章中我们讨论了如何使用双斜杠 `//` 注释 Rust 代码。Rust 也有特定的用于文档的注释类型，通常被称为**文档注释**（*documentation comments*），它们会生成 HTML 文档。这些 HTML 展示公有 API 文档注释的内容，它们意在让对库感兴趣的程序员理解如何**使用**这个 crate，而不是它是如何被**实现**的。

文档注释使用三斜杠 `///` 而不是双斜杠以支持 Markdown 注解来格式化文本。文档注释就位于需要文档的项的之前。示例 14-1 展示了一个 `my_crate` crate 中 `add_one` 函数的文档注释。

文件名：src/lib.rs

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

示例 14-1：一个函数的文档注释

这里，我们提供了一个 `add_one` 函数工作的描述，接着开始了一个标题为 `Examples` 的部分，和展示如何使用 `add_one` 函数的代码。可以运行 `cargo doc` 来生成这个文档注释的 HTML 文档。这个命令运行由 Rust 分发的工具 `rustdoc` 并将生成的 HTML 文档放入 *target/doc* 目录。

为了方便起见，运行 `cargo doc --open` 会构建当前 crate 文档（同时还有所有 crate 依赖的文档）的 HTML 并在浏览器中打开。导航到 `add_one` 函数将会发现文档注释的文本是如何渲染的，如图 14-1 所示：

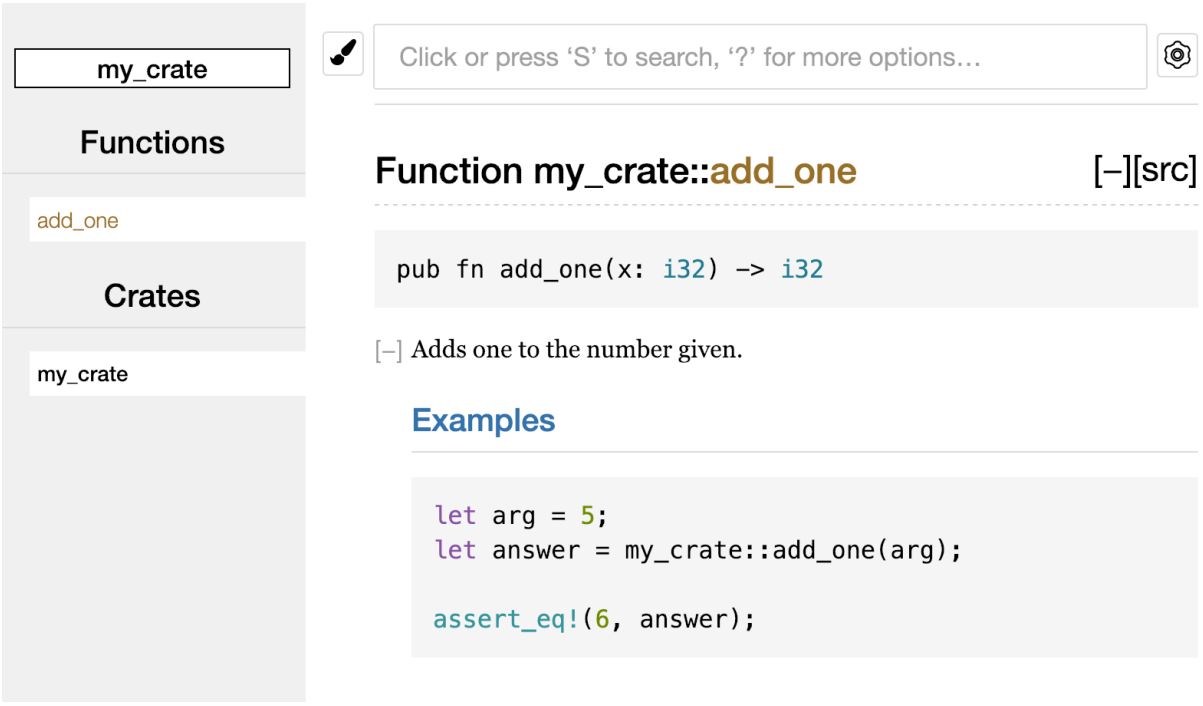


图 14-1：`add_one` 函数的文档注释 HTML

常用（文档注释）部分

示例 14-1 中使用了 `# Examples` Markdown 标题在 HTML 中创建了一个以 “Examples” 为标题的部分。其他一些 crate 作者经常在文档注释中使用的部分有：

- **Panics**：这个函数可能会 `panic!` 的场景。并不希望程序崩溃的函数调用者应该确保他们不会在这些情况下调用此函数。
- **Errors**：如果这个函数返回 `Result`，此部分描述可能会出现何种错误以及什么情况会造成这些错误，这有助于调用者编写代码来采用不同的方式处理不同的错误。
- **Safety**：如果这个函数使用 `unsafe` 代码（这会在第二十章讨论），这一部分应该会涉及到期望函数调用者支持的确保 `unsafe` 块中代码正常工作的不变条件（invariants）。

大部分文档注释不需要所有这些部分，不过这是一个提醒你检查调用你代码的用户有兴趣了解的内容的列表。

文档注释作为测试

在文档注释中增加示例代码块是一个清楚的表明如何使用库的方法，这么做还有一个额外的好处：`cargo test` 也会像测试那样运行文档中的示例代码！没有什么比有例子的文档更好的了，但最糟糕的莫过于写完文档后改动了代码，而导致例子不能正常工作。尝试 `cargo test` 运行像示例 14-1 中 `add_one` 函数的文档；应该在测试结果中看到像这样的部分：

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.27s
```

现在尝试改变函数或例子来使例子中的 `assert_eq!` 产生 panic。再次运行 `cargo test`，你将会看到文档测试捕获到了例子与代码不再同步！

注释包含项的结构

文档注释风格 `//!` 为包含注释的项，而不是位于注释之后的项增加文档。这通常用于 crate 根文件（通常是 *src/lib.rs*）或模块的根文件为 crate 或模块整体提供文档。

作为一个例子，为了增加描述包含 `add_one` 函数的 `my_crate` crate 目的的文档，可以在 *src/lib.rs* 开头增加以 `//!` 开头的注释，如示例 14-2 所示：

文件名：src/lib.rs

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing certain
//! calculations more convenient.

/// Adds one to the number given.
// --snip--
```

示例 14-2：`my_crate` crate 整体的文档

注意 `//!` 的最后一行之后没有任何代码。因为它们以 `//!` 开头而不是 `///`，这是属于包含此注释的项而不是注释之后项的文档。在这个情况下是 *src/lib.rs* 文件，也就是 crate 根文件。这些注释描述了整个 crate。

如果运行 `cargo doc --open`，将会发现这些注释显示在 `my_crate` 文档的首页，位于 crate 中公有项列表之上，如图 14-2 所示：

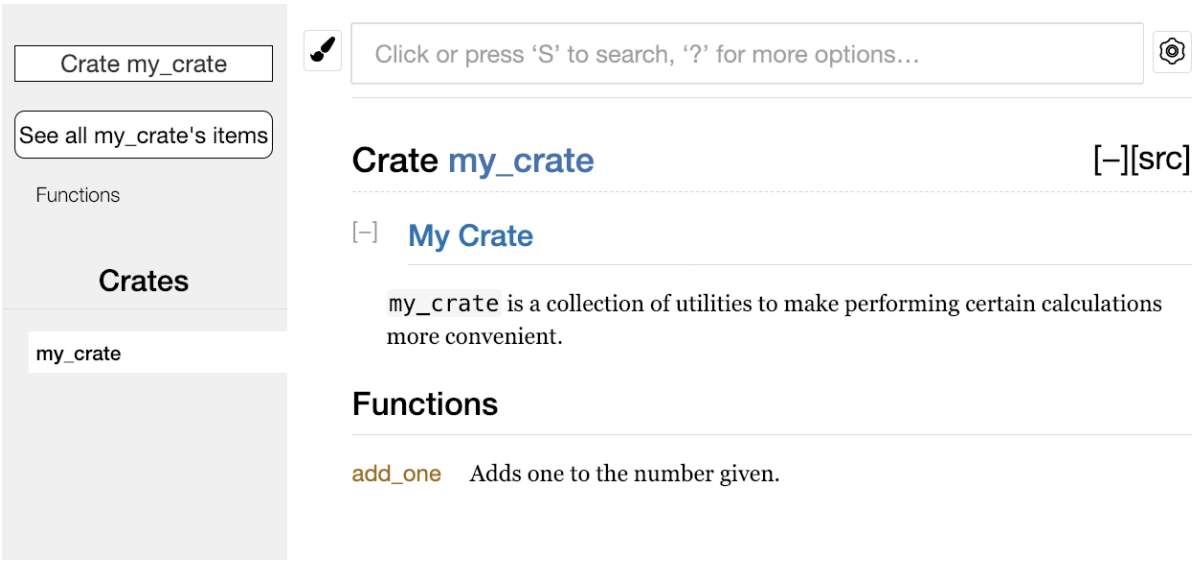


图 14-2：包含 `my_crate` 整体描述的注释所渲染的文档

位于项之中的文档注释对于描述 crate 和模块特别有用。使用它们描述其容器整体的目的来帮助 crate 用户理解你的代码组织。

使用 `pub use` 导出便捷的公有 API

公有 API 的结构是你发布 crate 时主要需要考虑的。crate 用户没有你那么熟悉其结构，并且如果模块层级过大他们可能会难以找到所需的部分。

第七章介绍了如何使用 `pub` 关键字将项变为公有，和如何使用 `use` 关键字将项引入作用域。然而你开发时候使用的文件架构可能并不方便用户使用。你的结构可能是一个包含多个层级的分层结构，不过这对于用户来说并不方便。这是因为想要使用被定义在很深层级中的类型的人可能很难发现这些类型的存在。他们也可能会厌烦要使用 `use my_crate::some_module::another_module::UsefulType;` 而不是 `use my_crate::UsefulType;` 来使用类型。

好消息是，即使文件结构对于用户来说**不是**很方便，你也无需重新安排内部组织：你可以选择使用 `pub use` 重导出（re-export）项来使公有结构不同于私有结构。重导出获取位于一个位置的公有项并将其公开到另一个位置，好像它就定义在这个新位置一样。

例如，假设我们创建了一个描述艺术概念的库 `art`。这个库中包含了一个有两个枚举 `PrimaryColor` 和 `SecondaryColor` 的模块 `kinds`，以及一个包含函数 `mix` 的模块 `utils`，如示例 14-3 所示：

文件名：src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

示例 14-3：一个库 `art` 其组织包含 `kinds` 和 `utils` 模块

`cargo doc` 所生成的 crate 文档首页如图 14-3 所示：

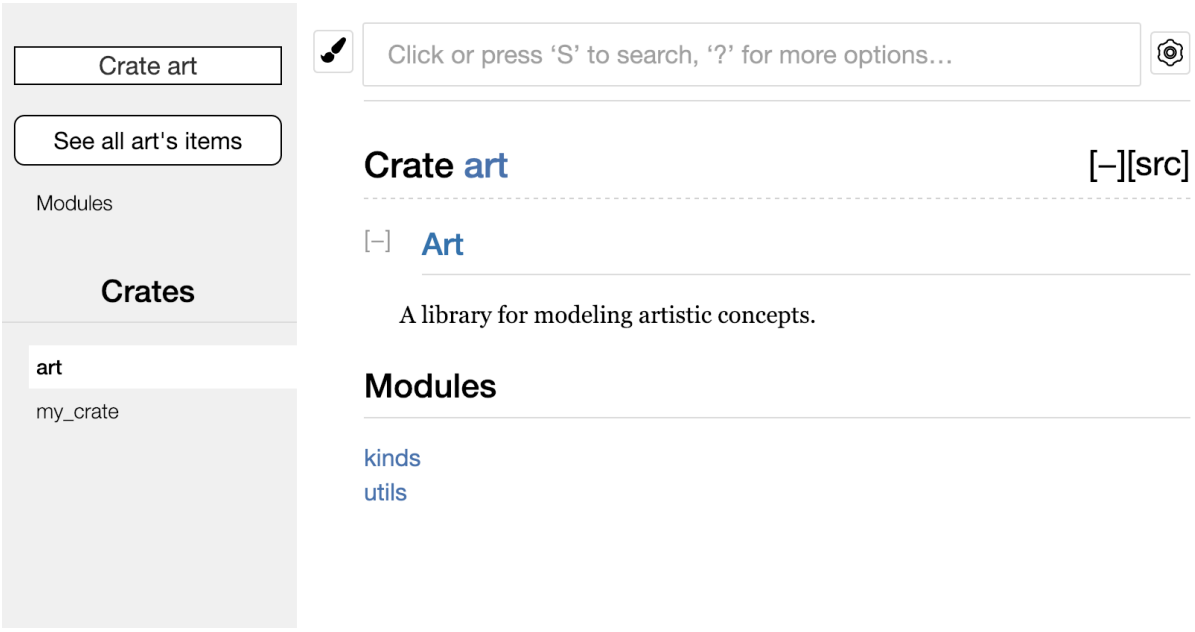


图 14-3：包含 `kinds` 和 `utils` 模块的库 `art` 的文档首页

注意 `PrimaryColor` 和 `SecondaryColor` 类型、以及 `mix` 函数都没有在首页中列出。我们必须点击 `kinds` 或 `utils` 才能看到它们。

另一个依赖这个库的 crate 需要 `use` 语句来导入 `art` 中的项，这包含指定其当前定义的模块结构。示例 14-4 展示了一个使用 `art` crate 中 `PrimaryColor` 和 `mix` 项的 crate 的例子：

文件名：src/main.rs

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

示例 14-4：一个通过导出内部结构使用 `art` crate 中项的 crate

示例 14-4 中使用 `art` crate 代码的作者不得不搞清楚 `PrimaryColor` 位于 `kinds` 模块而 `mix` 位于 `utils` 模块。`art` crate 的模块结构相比使用它的开发者来说对编写它的开发者更有意义。其内部结构并没有对尝试理解如何使用 `art` crate 的人提供任何有价值的信息，相反因为不得不搞清楚所需的内容在何处和必须在 `use` 语句中指定模块名称而显得混乱。

为了从公有 API 中去掉 crate 的内部组织，我们可以采用示例 14-3 中的 `art` crate 并增加 `pub use` 语句来重导出项到顶层结构，如示例 14-5 所示：

文件名：src/lib.rs

```
//! # Art
//!
//! A library for modeling artistic concepts.
```

```
pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

示例 14-5：增加 `pub use` 语句重导出项

现在此 crate 由 `cargo doc` 生成的 API 文档会在首页列出重导出的项以及其链接（re-exports），如图 14-4 所示，这使得 `PrimaryColor` 和 `SecondaryColor` 类型和 `mix` 函数更易于查找。

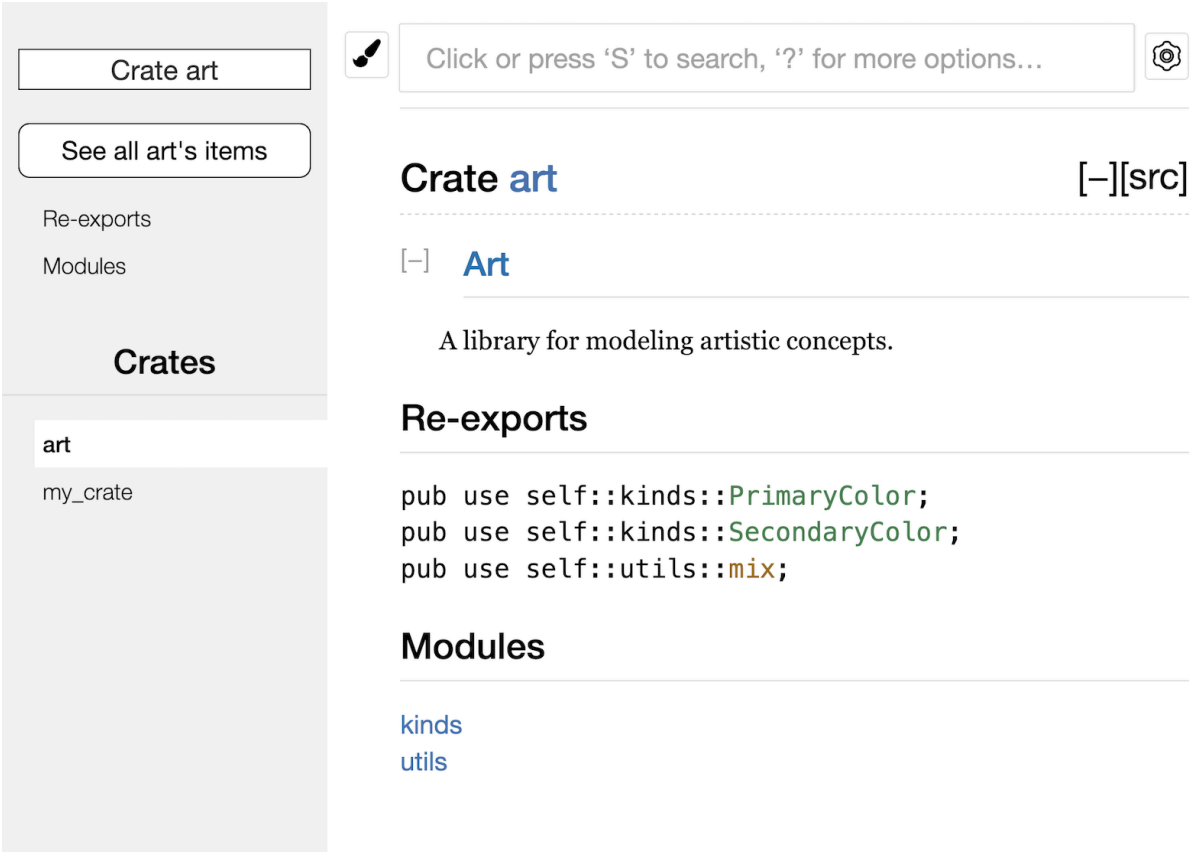


图 14-10：`art` 文档的首页，这里列出了重导出的项

`art` crate 的用户仍然可以看到并使用示例 14-3 中的内部结构，如示例 14-4 所示，或者可以使用示例 14-5 中更为方便的结构，如示例 14-6 所示：

文件名：src/main.rs

```
use art::PrimaryColor;
use art::mix;
```

```
fn main() {
    // --snip--
}
```

示例 14-6：一个使用 `art` crate 中重导出项的程序

对于有很多嵌套模块的情况，使用 `pub use` 将类型重导出到顶级结构对于使用 crate 的人来说将会是大为不同的体验。`pub use` 的另一个常见用法是重导出当前 crate 的依赖的定义使其 crate 定义变成你 crate 公有 API 的一部分。

创建一个有用的公有 API 结构更像是一门艺术而非科学，你可以反复检视它们来找出最适合用户的 API。`pub use` 提供了解耦组织 crate 内部结构和与终端用户体现的灵活性。观察一些你所安装的 crate 的代码来看看其内部结构是否不同于公有 API。

创建 Crates.io 账号

在你可以发布任何 crate 之前，需要在 crates.io 上注册账号并获取一个 API token。为此，访问位于 crates.io 的首页并使用 GitHub 账号登录。（目前 GitHub 账号是必须的，不过将来该网站可能会支持其他创建账号的方法）一旦登录之后，查看位于 https://crates.io/me/ 的账户设置页面并获取 API token。然后运行 `cargo login` 命令，并在提示时粘贴该 token，操作如下所示：

```
$ cargo login
abcdefghijklmnopqrstuvwxyz012345
```

这个命令会通知 Cargo 你的 API token 并将其储存在本地的 *~/.cargo/credentials* 文件中。注意这个 token 是一个**秘密**（**secret**）且不应该与其他人共享。如果因为任何原因与他人共享了这个信息，应该立即到 crates.io 撤销并重新生成一个 token。

向新 crate 添加元数据

比如说你已经有一个希望发布的 crate。在发布之前，你需要在 crate 的 *Cargo.toml* 文件的 `[package]` 部分增加一些本 crate 的元数据（metadata）。

首先 crate 需要一个唯一的名称。虽然在本地开发 crate 时，可以使用任何你喜欢的名称。不过 crates.io 上的 crate 名称遵守先到先得的分配原则。一旦某个 crate 名称被使用，其他人就不能再发布这个名称的 crate 了。请搜索你希望使用的名称来找出它是否已被使用。如果没有，修改 *Cargo.toml* 中 `[package]` 里的`name` 字段为你希望用于发布的名称，像这样：

文件名：Cargo.toml

```
[package]
name = "guessing_game"
```

即使你选择了一个唯一的名称，如果此时尝试运行 `cargo publish` 发布该 crate 的话，会得到一个警告接着是一个错误：

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file, documentation,
```

```
homepage or repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata for
more info.
--snip--
error: failed to publish to registry at https://crates.io

Caused by:
  the remote server responded with an error (status 400 Bad Request): missing or
empty metadata fields: description, license. Please see https://doc.rust-lang.org/
cargo/reference/manifest.html for more information on configuring these fields
```

这个错误是因为我们缺少一些关键信息：关于该 crate 用途的描述和用户可能在何种条款下使用该 crate 的 license。在 *Cargo.toml* 中添加通常是一两句话的描述，因为它将在搜索结果中和你的 crate 一起显示。对于 `license` 字段，你需要一个 **license 标识符值**（*license identifier value*）。Linux 基金会的 Software Package Data Exchange (SPDX) 列出了可以使用的标识符。例如，为了指定 crate 使用 MIT License，增加 `MIT` 标识符：

文件名：Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

如果你希望使用不存在于 SPDX 的 license，则需要将 license 文本放入一个文件，将该文件包含进项目中，接着使用 `license-file` 来指定文件名而不是使用 `license` 字段。

关于项目所适用的 license 指导超出了本书的范畴。很多 Rust 社区成员选择与 Rust 自身相同的 license，这是一个双许可的 `MIT OR Apache-2.0`。这个实践展示了也可以通过 `OR` 分隔为项目指定多个 license 标识符。

那么，有了唯一的名称、版本号、由 `cargo new` 新建项目时增加的作者信息、描述和所选择的 license，已经准备好发布的项目的 *Cargo.toml* 文件可能看起来像这样：

文件名：Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2024"
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

Cargo 的文档 描述了其他可以指定的元数据，它们可以帮助你的 crate 更容易被发现和使用！

发布到 Crates.io

现在我们创建了一个账号，保存了 API token，为 crate 选择了一个名字，并指定了所需的元数据，你已经准备好发布了！发布 crate 会上传特定版本的 crate 到 crates.io 以供他人使用。

发布 crate 时请多加小心，因为发布是**永久性的**（*permanent*）。对应版本不可能被覆盖，其代码也不可能被删除。crates.io 的一个主要目标是作为一个存储代码的永久文档服务器，这样

所有依赖 crates.io 中的 crate 的项目都能一直正常工作。而允许删除版本没办法达成这个目标。然而，可以被发布的版本号却没有限制。

再次运行 `cargo publish` 命令。这次它应该会成功：

```
$ cargo publish
    Updating crates.io index
   Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
   Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
   Compiling guessing_game v0.1.0
(file:///projects/guessing_game/target/package/guessing_game-0.1.0)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.19s
   Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

恭喜！你现在向 Rust 社区分享了代码，而且任何人都可以轻松的将你的 crate 加入他们项目的依赖。

发布现有 crate 的新版本

当你修改了 crate 并准备好发布新版本时，改变 *Cargo.toml* 中 `version` 所指定的值。请使用语义化版本规则来根据修改的类型决定下一个版本号。接着运行 `cargo publish` 来上传新版本。

使用 `cargo yank` 从 Crates.io 撤回版本

虽然你不能删除 crate 的历史版本，但是可以阻止任何将来的项目将它们加入到依赖中。这在某个版本因为这样或那样的原因被破坏的情况很有用。对于这种情况，Cargo 支持**撤回**（*yanking*）某个版本。

撤回某个版本会阻止新项目依赖此版本，不过所有现存此依赖的项目仍然能够下载和依赖这个版本。从本质上说，撤回意味着所有带有 *Cargo.lock* 的项目的依赖不会被破坏，同时任何新生成的 *Cargo.lock* 将不能使用被撤回的版本。

为了撤回一个版本的 crate，在之前发布 crate 的目录运行 `cargo yank` 并指定希望撤回的版本。例如，如果我们发布了一个名为 `guessing_game` 的 crate 的 1.0.1 版本并希望撤回它，在 `guessing_game` 项目目录运行：

```
$ cargo yank --vers 1.0.1
    Updating crates.io index
        Yank guessing_game@1.0.1
```

也可以撤销撤回操作，并允许项目可以再次开始依赖某个版本，通过在命令上增加 `--undo`：

```
$ cargo yank --vers 1.0.1 --undo
    Updating crates.io index
      Unyank guessing_game@1.0.1
```

撤回**并没有**删除任何代码。举例来说，撤回功能并不能删除不小心上传的秘密信息。如果出现了这种情况，请立即重新设置这些秘密信息。

Cargo 工作空间

第十二章中，我们构建一个包含二进制 crate 和库 crate 的包。你可能会发现，随着项目开发的深入，库 crate 持续增大，而你希望将其进一步拆分成多个库 crate。Cargo 提供了一个叫**工作空间**（*workspaces*）的功能，它可以帮助我们管理多个相关的协同开发的包。

创建工作空间

工作空间是一系列共享同样的 *Cargo.lock* 和输出目录的包。让我们使用工作空间创建一个项目——— 这里采用常见的代码以便可以关注工作空间的结构。有多种组织工作空间的方式，所以我们只展示一个常用方法。我们的工作空间有一个二进制项目和两个库。二进制项目会提供主要功能，并会依赖另两个库。一个库会提供 `add_one` 方法而第二个会提供 `add_two` 方法。这三个 crate 将会是相同工作空间的一部分。让我们以新建工作空间目录开始：

```
$ mkdir add
$ cd add
```

接着在 *add* 目录中，创建 *Cargo.toml* 文件，用于配置整个整个工作空间。它不会包含 `[package]` 部分。相反，它以 `[workspace]` 部分作为开始，允许我们向工作区添加成员。我们还通过将 `resolver` 设置为 `"3"`，在工作区中使用 Cargo 最新且最强大的解析算法。

文件名：Cargo.toml

```
[workspace]
resolver = "3"
```

接下来，在 *add* 目录运行 `cargo new` 新建 `adder` 二进制 crate：

```
$ cargo new adder
    Creating binary (application) `adder` package
      Adding `adder` as member of workspace at `file:///projects/add`
```

在工作空间中运行 `cargo new` 也会自动将新建包加入到工作空间 `Cargo.toml` 的 `[workspace]` 定义的 `members` 键中，像这样：

```
[workspace]
resolver = "3"
members = ["adder"]
```

到此为止，可以运行 `cargo build` 来构建工作空间。*add* 目录中的文件应该看起来像这样：

```
├── Cargo.lock
├── Cargo.toml
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

工作空间在顶级目录只有一个 *target* 目录，用于存放编译生成的产物；`adder` 包并没有自己的 *target* 目录。即使进入 *adder* 目录运行 `cargo build`，构建结果也位于 *add/target* 而不是 *add/adder/target*。工作空间中的 crate 之间相互依赖。如果每个 crate 有其自己的 *target* 目录，为了在自己的 *target* 目录中生成构建结果，工作空间中的每一个 crate 都不得不相互重新编译其他 crate。通过共享一个 *target* 目录，工作空间可以避免其他 crate 重复构建。

在工作空间中创建第二个包

接下来，让我们在工作空间中创建另一个成员包，并将其命名为 `add_one`。生成一个名为 `add_one` 的库 crate：

```
$ cargo new add_one --lib
    Creating library `add_one` package
      Adding `add_one` as member of workspace at `file:///projects/add`
```

现在顶层的 *Cargo.toml* 的 `members` 列表将会包含 add_one* 路径：

文件名：Cargo.toml

```
[workspace]
resolver = "3"
members = ["adder", "add_one"]
```

现在 *add* 目录应该有如下目录和文件：

```
├── Cargo.lock
├── Cargo.toml
├── add_one
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── adder
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

在 *add_one/src/lib.rs* 文件中，增加一个 `add_one` 函数：

文件名：add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

现在我们有了二进制 `adder` 包依赖库 crate `add_one` 包。首先需要在 *adder/Cargo.toml* 文件中增加 `add_one` 作为路径依赖：

文件名：adder/Cargo.toml

```
[dependencies]
add_one = { path = "../add_one" }
```

cargo 并不假定工作空间中的 Crates 会相互依赖，所以需要显式表明工作空间中 crate 的依赖关系。

接下来，在 `adder` crate 中使用（`add_one` crate 中的）函数 `add_one`。打开 *adder/src/main.rs* 在顶部增加一行 `use` 将新 `add_one` 库 crate 引入作用域。接着修改 `main` 函数来调用 `add_one` 函数，如示例 14-7 所示。

文件名：adder/src/main.rs

```
fn main() {
    let num = 10;
    println!("Hello, world! {num} plus one is {}!", add_one::add_one(num));
}
```

示例 14-7：在 `adder` crate 中使用 `add_one` 库 crate

在顶层 *add* 目录中运行 `cargo build` 来构建工作空间！

```
$ cargo build
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.22s
```

为了在顶层 *add* 目录运行二进制 crate，可以通过 `-p` 参数和包名称来运行 `cargo run` 指定工作空间中我们希望使用的包：

```
$ cargo run -p adder
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

这会运行 *adder/src/main.rs* 中的代码，其依赖 `add_one` crate

在工作空间中依赖外部包

还需注意的是工作空间只在根目录有一个 *Cargo.lock*，而不是在每一个 crate 目录都有 *Cargo.lock*。这确保了所有的 crate 都使用完全相同版本的依赖。如果在 *Cargo.toml* 和 *add_one/Cargo.toml* 中都增加 `rand` crate，则 Cargo 会将其都解析为同一版本并记录到唯一的 *Cargo.lock* 中。使得工作空间中的所有 crate 都使用相同的依赖意味着其中的 crate 都是相互兼容的。让我们在 *add_one/Cargo.toml* 中的 `[dependencies]` 部分增加 `rand` crate 以便能够在 `add_one` crate 中使用 `rand` crate：

文件名：add_one/Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

现在就可以在 *add_one/src/lib.rs* 中增加 `use rand;` 了，接着在 *add* 目录运行 `cargo build` 构建整个工作空间就会引入并编译 `rand` crate。我们会收到一个警告，因为我们并没有引用已导入作用域的 `rand`：

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.5
   --snip--
   Compiling rand v0.8.5
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
warning: unused import: `rand`
 --> add_one/src/lib.rs:1:5
  |
1 | use rand;
  |     ^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

warning: `add_one` (lib) generated 1 warning (run `cargo fix --lib -p add_one` to
apply 1 suggestion)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.95s
```

现在顶级的 *Cargo.lock* 包含了 `add_one` 的 `rand` 依赖的信息。然而，即使 `rand` 被用于工作空间的某处，也不能在其他 crate 中使用它，除非也在它们的 *Cargo.toml* 中加入 `rand`。例如，如果在顶级的 `adder` crate 的 *adder/src/main.rs* 中增加 `use rand;`，会得到一个错误：

```
$ cargo build
  --snip--
   Compiling adder v0.1.0 (file:///projects/add/adder)
error[E0432]: unresolved import `rand`
 --> adder/src/main.rs:2:5
  |
2 | use rand;
  |     ^^^^ no external crate `rand`
```

为了修复这个错误，修改顶级 `adder` crate 的 *Cargo.toml* 来表明 `rand` 也是这个 crate 的依赖。构建 `adder` crate 会将 `rand` 加入到 *Cargo.lock* 中 `adder` 的依赖列表中，但是这并不会下载 `rand` 的额外拷贝。Cargo 确保了工作空间中任何使用 `rand` 的 crate 都采用相同的版本，这节省了空间并确保了工作空间中的 crate 将是相互兼容的。

如果工作空间中的 crate 指定了不兼容的同一依赖的不同版本，Cargo 会解析它们，但仍会尽量减少解析的版本数量。

为工作空间增加测试

作为另一个改进，让我们为 `add_one` crate 中的 `add_one::add_one` 函数增加一个测试：

文件名：add_one/src/lib.rs

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

在顶级 *add* 目录运行 `cargo test`。在像这样的工作空间结构中运行 `cargo test` 会运行工作空间中所有 crate 的测试。：

```
$ cargo test
   Compiling add_one v0.1.0 (file:///projects/add/add_one)
   Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.20s
     Running unittests src/lib.rs (target/debug/deps/add_one-93c49ee75dc46543)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/adder-3a47283c568d2b6a)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests add_one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

输出的第一部分显示 `add_one` crate 的 `it_works` 测试通过了。下一个部分显示 `adder` crate 中找到了零个测试，最后一部分显示 `add_one` crate 中有零个文档测试。

也可以选择运行工作空间中特定 crate 的测试，通过在根目录使用 `-p` 参数并指定希望测试的 crate 名称：

```
$ cargo test -p add_one
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests src/lib.rs (target/debug/deps/add_one-93c49ee75dc46543)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests add_one
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

输出显示了 `cargo test` 只运行了 `add_one` crate 的测试而没有运行 `adder` crate 的测试。

如果你选择向 crates.io 发布工作空间中的 crate，每一个工作空间中的 crate 需要单独发布。就像 `cargo test` 一样，可以通过 `-p` 参数并指定期望发布的 crate 名来发布工作空间中的某个特定的 crate。

现在尝试以类似 `add_one` crate 的方式向工作空间增加 `add_two` crate 来作为更多的练习！

随着项目增长，考虑使用工作空间：每一个更小的组件比一大块代码要容易理解。如果它们经常需要同时被修改的话，将 crate 保持在工作空间中更易于协调 crate 的改变。

使用 `cargo install` 安装二进制文件

`cargo install` 命令用于在本地安装和使用二进制 crate。它并不打算替换系统中的包；它意在作为一个方便 Rust 开发者们安装其他人已经在 crates.io 上共享的工具的手段。只有拥有二进制目标文件的包能够被安装。**二进制目标** 文件是在 crate 有 *src/main.rs* 或者其他指定为二进制文件时所创建的可执行程序，这不同于自身不能执行但适合包含在其他程序中的库目标文件。通常 crate 的 *README* 文件中有该 crate 是库、二进制目标还是两者兼有的信息。

所有来自 `cargo install` 的二进制文件都安装到 Rust 安装根目录的 *bin* 文件夹中。如果你是使用 *rustup.rs* 来安装 Rust 且没有自定义任何配置，这个目录将是 *$HOME/.cargo/bin*。确保将这个目录添加到 `$PATH` 环境变量中就能够运行通过 `cargo install` 安装的程序了。

例如，第十二章提到的叫做 `ripgrep` 的用于搜索文件的 `grep` 的 Rust 实现。为了安装 `ripgrep` 运行如下：

```
$ cargo install ripgrep
    Updating crates.io index
  Downloaded ripgrep v14.1.1
  Downloaded 1 crate (213.6 KB) in 0.40s
  Installing ripgrep v14.1.1
--snip--
   Compiling grep v0.3.2
    Finished `release` profile [optimized + debuginfo] target(s) in 6.73s
  Installing ~/.cargo/bin/rg
   Installed package `ripgrep v14.1.1` (executable `rg`)
```

倒数第二行输出展示了安装的二进制文件的位置和名称，在这里 `ripgrep` 被命名为 `rg`。只要你像上面提到的那样将安装目录加入 `$PATH`，就可以运行 `rg --help` 并开始使用一个更快更 Rust 的工具来搜索文件了！

Cargo 自定义扩展命令

Cargo 的设计使得开发者可以通过新的子命令来对 Cargo 进行扩展，而无需修改其本身。如果 `$PATH` 中有类似 `cargo-something` 的二进制文件，就可以通过 `cargo something` 来像 Cargo 子命令一样运行它。像这样的自定义命令也可以运行 `cargo --list` 来展示出来。能够通过 `cargo install` 向 Cargo 安装扩展并可以如内建 Cargo 工具那样运行它们是 Cargo 设计上的一个非常方便的优点！

总结

通过 Cargo 和 crates.io 来分享代码是使得 Rust 生态环境可以用于许多不同的任务的重要组成部分。Rust 的标准库是小而稳定的，不过 crate 易于分享和使用，并采用一个不同于语言自身的时间线来提供改进。不要犹豫在 crates.io 上共享对你有用的代码，因为它很有可能对别人也很有用！

智能指针

指针（*pointer*）是一个包含内存地址的变量的通用概念。这个地址引用，或 “指向”（points at）一些其它数据。Rust 中最常见的指针是第四章介绍的**引用**（*reference*）。引用以 `&` 符号为标志并借用了它们所指向的值。除了引用数据没有任何其他特殊功能，也没有额外开销。

另一方面，**智能指针**（*smart pointers*）是一类数据结构，它们的表现类似指针，但是也拥有额外的元数据和功能。智能指针的概念并不为 Rust 所独有；其起源于 C++ 并存在于其它语言中。Rust 标准库中定义了多种不同的智能指针，它们提供了多于引用的额外功能。为了探索其基本概念，我们来看看一些智能指针的例子，这包括**引用计数** （*reference counting*）智能指针类型。这种指针允许数据有多个所有者，它会记录所有者的数量，当没有所有者时清理数据。

在 Rust 中因为引用和借用的概念，普通引用和智能指针有一个额外的区别：引用是一类只借用数据的指针，在大部分情况下，智能指针**拥有**它们指向的数据。

实际上本书中已经出现过一些智能指针，比如第八章的 `String` 和 `Vec<T>`，虽然当时并没有这样称呼它们。这些类型都属于智能指针，因为它们拥有一些数据，并允许你修改这些数据。它们也拥有元数据和额外的功能或保证。例如 `String` 存储了其容量作为元数据，并拥有额外的能力来确保其数据总是有效的 UTF-8 编码。

智能指针通常使用结构体实现。智能指针不同于结构体的地方在于其实现了 `Deref` 和 `Drop` trait。`Deref` trait 允许智能指针结构体实例表现的像引用一样，这样就可以编写既用于引用、又用于智能指针的代码。`Drop` trait 允许我们自定义当智能指针离开作用域时运行的代码。本章会讨论这些 trait 以及为什么对于智能指针来说它们很重要。

考虑到智能指针是一个在 Rust 经常被使用的通用设计模式，本章并不会覆盖所有现存的智能指针。很多库都有自己的智能指针而你也可以编写属于你自己的智能指针。这里将会讲到的是来自标准库中最常用的几种：

- `Box<T>`，用于在堆上分配值
- `Rc<T>`，一个引用计数类型，其数据可以有多个所有者
- `Ref<T>` 和 `RefMut<T>`，通过 `RefCell<T>` 访问。`RefCell<T>` 是一个在运行时而不是在编译时执行借用规则的类型。

另外我们会涉及**内部可变性**（*interior mutability*）模式，这是不可变类型暴露出改变其内部值的 API。我们也会讨论**引用循环**（*reference cycles*）会如何泄漏内存，以及如何避免它们。

让我们开始吧！

使用 `Box<T>` 指向堆上的数据

最简单直接的智能指针是 *box*，其类型是 `Box<T>`。box 允许你将一个值放在堆上而不是栈上。留在栈上的则是指向堆数据的指针。如果你想回顾一下栈与堆的区别请参考第四章。

除了数据被储存在堆上而不是栈上之外，box 没有性能损失。不过也没有很多额外的功能。它们多用于如下场景：

- 当有一个在编译时未知大小的类型，而又想要在需要确切大小的上下文中使用这个类型值的时候
- 当有大量数据并希望在确保数据不被拷贝的情况下转移所有权的时候
- 当希望拥有一个值并只关心它的类型是否实现了特定 trait 而不是其具体类型的时候

我们会在“box 允许创建递归类型”部分展示第一种场景。在第二种情况中，转移大量数据的所有权可能会花费很长的时间，因为数据在栈上进行了拷贝。为了改善这种情况下的性能，可以通过 box 将这些数据储存在堆上。接着，只有少量的指针数据在栈上被拷贝。第三种情况被称为 **trait 对象**（*trait object*），第十八章刚好有一整个部分“顾及不同类型值的 trait 对象”专门讲解这个主题。所以这里所学的内容会在第十八章再次应用！

使用 `Box<T>` 在堆上存储数据

在讨论 `Box<T>` 的堆存储用例之前，让我们熟悉一下语法以及如何与存储在 `Box<T>` 中的值进行交互。

示例 15-1 展示了如何使用 box 在堆上储存一个 `i32`：

文件名：src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {b}");
}
```

示例 15-1：使用 box 在堆上储存一个 `i32` 值

这里定义了变量 `b`，其值是一个指向被分配在堆上的值 `5` 的 `Box`。这个程序会打印出 `b = 5`；在这个例子中，我们可以像数据是储存在栈上的那样访问 box 中的数据。正如任何拥有数据所有权的值那样，当像 `b` 这样的 box 在 `main` 的末尾离开作用域时，它将被释放。这个释放过程作用于 box 本身（位于栈上）和它所指向的数据（位于堆上）。

将一个单独的值存放在堆上并不是很有意义，所以像示例 15-1 这样单独使用 box 并不常见。将像单个 `i32` 这样的值储存在栈上，也就是其默认存放的地方在大部分使用场景中更为合适。让我们看看一个不使用 box 时无法定义的类型的例子。

Box 允许创建递归类型

递归类型（*recursive type*）的值可以拥有另一个同类型的值作为其自身的一部分。但是这会产生一个问题，因为 Rust 需要在编译时知道类型占用多少空间。递归类型的值嵌套理论上可以无限地进行下去，所以 Rust 不知道递归类型需要多少空间。因为 box 有一个已知的大小，所以通过在循环类型定义中插入 box，就可以创建递归类型了。

作为一个递归类型的例子，让我们探索一下 *cons list*。这是一个函数式编程语言中常见的数据类型，来展示这个（递归类型）概念。除了递归之外，我们将要定义的 cons list 类型相当简单，所以这个例子中的概念，在任何遇到更为复杂的涉及到递归类型的场景时都很实用。

cons list 的更多信息

cons list 是一个来源于 Lisp 编程语言及其方言的数据结构，它由嵌套的列表组成。它的名字来源于 Lisp 中的 `cons` 函数（“construct function“ 的缩写），它利用两个参数来构造一个新的列表。通过对一个包含值的列表和另一个值调用 `cons`，可以构建由递归列表组成的 cons list。

例如这里有一个包含列表 `1, 2, 3` 的 cons list 的伪代码表示，其每个对在一个括号中：

```
(1, (2, (3, Nil)))
```

cons list 的每一项都包含两个元素：当前项的值和下一项。其最后一项值包含一个叫做 `Nil` 的值且没有下一项。cons list 通过递归调用 `cons` 函数产生。代表递归的归基条件（base case）的规范名称是 `Nil`，它宣布列表的终止。注意这不同于第六章中的 “null” 或 “nil” 的概念，它们代表无效或缺失的值。

cons list 并不是一个 Rust 中常见的类型。大部分在 Rust 中需要列表的时候，`Vec<T>` 是一个更好的选择。其他更为复杂的递归数据类型**确实**在 Rust 的很多场景中很有用，不过通过以 cons list 作为开始，我们可以探索如何使用 box 毫不费力地定义一个递归数据类型。

示例 15-2 包含一个 cons list 的枚举定义。注意这还不能编译因为这个类型没有已知的大小，之后我们会展示：

文件名：src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```



示例 15-2：第一次尝试定义一个代表 `i32` 值的 cons list 数据结构的枚举

> 注意：出于示例的需要我们选择实现一个只存放 `i32` 值的 cons list。也可以用泛型，正如第十章讲到的，来定义一个可以存放任何类型值的 cons list 类型。

使用这个 cons list 来储存列表 `1, 2, 3` 将看起来如示例 15-3 所示：

文件名：src/main.rs

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```



示例 15-3：使用 `List` 枚举储存列表 `1, 2, 3`

第一个 `Cons` 储存了 `1` 和另一个 `List` 值。这个 `List` 是另一个包含 `2` 的 `Cons` 值和下一个 `List` 值。接着又有另一个存放了 `3` 的 `Cons` 值和最后一个值为 `Nil` 的 `List`，非递归变体代表了列表的结尾。

如果尝试编译示例 15-3 的代码，会得到如示例 15-4 所示的错误：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +

error[E0391]: cycle detected when computing when `List` needs drop
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
  |
  = note: ...which immediately requires computing when `List` needs drop again
  = note: cycle used when computing whether `List` needs drop
  = note: see https://rustc-dev-guide.rust-lang.org/overview.html#queries and
https://rustc-dev-guide.rust-lang.org/query.html for more information

Some errors have detailed explanations: E0072, E0391.
For more information about an error, try `rustc --explain E0072`.
error: could not compile `cons-list` (bin "cons-list") due to 2 previous errors
```

示例 15-4：尝试定义一个递归枚举时得到的错误

这个错误表明这个类型 “有无限的大小”（“has infinite size”）。其原因是 `List` 的一个变体被定义为是递归的：它直接存放了另一个相同类型的值。这意味着 Rust 无法计算为了存放 `List` 值到底需要多少空间。让我们拆开来看为何会得到这个错误。首先了解一下 Rust 如何决定需要多少空间来存放一个非递归类型。

计算非递归类型的大小

回忆一下第六章讨论枚举定义时示例 6-2 中定义的 `Message` 枚举：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

当 Rust 需要知道要为 `Message` 值分配多少空间时，它可以检查每一个变体并发现 `Message::Quit` 并不需要任何空间，`Message::Move` 需要足够储存两个 `i32` 值的空间，依此类推。因为 enum 实际上只会使用其中的一个变体，所以 `Message` 值所需的空间等于储存其最大变体的空间大小。

与此相对当 Rust 编译器检查像示例 15-2 中的 `List` 这样的递归类型时会发生什么呢。编译器从 `Cons` 变体开始分析，其需要的空间等于 `i32` 的大小加上 `List` 的大小。为了计算 `List` 需要多少内存，编译器检查其变体，从 `Cons` 变体开始。`Cons` 变体储存了一个 `i32` 值和一个`List` 值，这样的计算将无限进行下去，如图 15-1 所示：

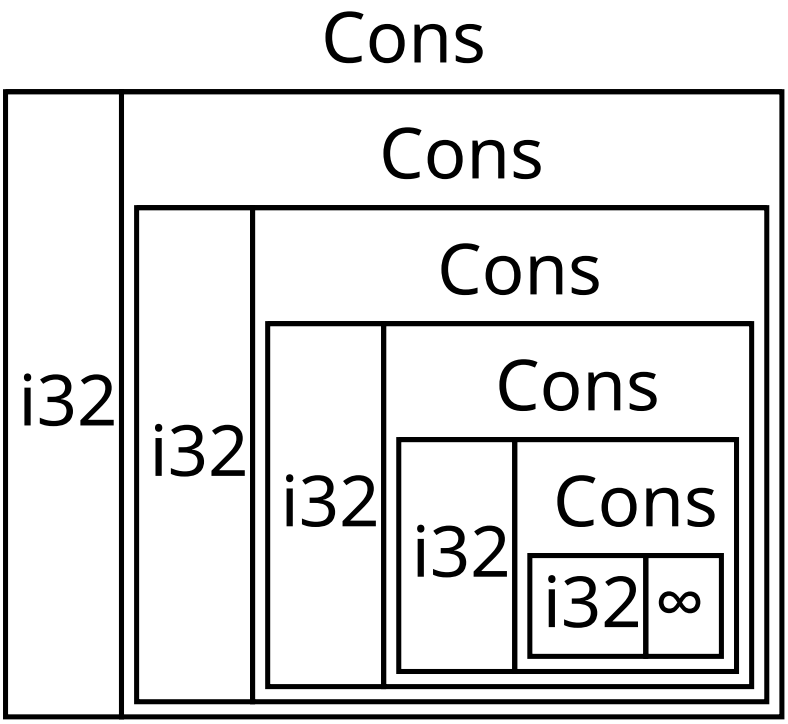


图 15-1：一个包含无限个 `Cons` 变体的无限 `List`

使用 `Box<T>` 给递归类型一个已知的大小

因为 Rust 无法计算出要为定义为递归的类型分配多少空间，所以编译器给出了一个包括了有用建议的错误：

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

在建议中，*indirection* 意味着不同于直接储存一个值，应该间接地存储一个指向值的指针。

因为 `Box<T>` 是一个指针，我们总是知道它需要多少空间：指针的大小并不会根据其指向的数据量而改变。这意味着可以将 `Box` 放入 `Cons` 变体中而不是直接存放另一个 `List` 值。`Box` 会指向下一个位于堆上的 `List` 值，而不是存放在 `Cons` 变体中。从概念上讲，我们仍然是在创

建一个包含其他列表的列表，不过现在实现这个概念的方式更像是一个项挨着另一项，而不是一项包含另一项。

我们可以修改示例 15-2 中 `List` 枚举的定义和示例 15-3 中对 `List` 的应用，如示例 15-5 所示，这是可以编译的：

文件名：src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

示例 15-5：为了拥有已知大小而使用 `Box<T>` 的 `List` 定义

`Cons` 变体将会需要一个 `i32` 的大小加上储存 box 指针数据的空间。`Nil` 变体不储存值，所以它比 `Cons` 变体需要更少的空间。现在我们知道了任何 `List` 值最多需要一个 `i32` 加上 box 指针数据的大小。通过使用 box，打破了这无限递归的连锁，这样编译器就能够计算出储存 `List` 值需要的大小了。图 15-2 展示了现在 `Cons` 变体看起来像什么：

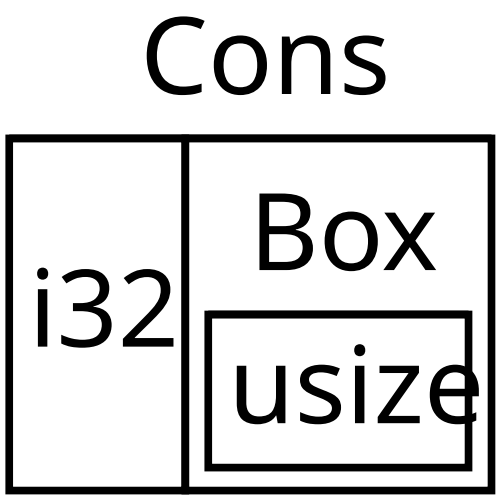


图 15-2：因为 `Cons` 存放一个 `Box` 所以 `List` 不是无限大小的了

box 只提供了间接存储和堆分配；它们并没有任何其他特殊的功能，比如我们将会见到的其他智能指针。它们也没有这些特殊功能带来的性能损失，所以它们可以用于像 cons list 这样间接存储是唯一所需特性的场景。我们还将在第十八章看到 box 的更多应用场景。

`Box<T>` 类型是一个智能指针，因为它实现了 `Deref` trait，它允许 `Box<T>` 值被当作引用对待。当 `Box<T>` 值离开作用域时，由于 `Box<T>` 类型 `Drop` trait 的实现，box 所指向的堆数据也会被

清除。这两个 trait 对于在本章余下讨论的其他智能指针所提供的功能中，将会更为重要。让我们更详细地探索一下这两个 trait。

使用 `Deref` Trait 将智能指针当作常规引用处理

实现 `Deref` trait 允许我们定制**解引用运算符**（*dereference operator*）`*`（不要与乘法运算符或通配符相混淆）。通过这种方式实现 `Deref` trait 的智能指针可以被当作常规引用来对待，可以编写操作引用的代码并同样适用于智能指针。

让我们首先看看解引用运算符如何处理常规引用，接着尝试定义我们自己的类似 `Box<T>` 的类型并看看为何解引用运算符不能像引用一样工作。我们会探索如何实现 `Deref` trait 使得智能指针以类似引用的方式工作变为可能。最后，我们会讨论 Rust 的 **Deref 强制转换**（*deref coercions*）特性以及它是如何处理引用或智能指针的。

> 我们将要构建的 `MyBox<T>` 类型与真正的 `Box<T>` 有一个很大的区别：我们的版本不会在堆上储存数据。这个例子重点关注 `Deref`，所以其数据实际存放在何处，相比其类似指针的行为来说不算重要。

追踪指针的值

常规引用是一个指针类型，一种理解指针的方式是将其看成指向储存在其他某处值的箭头。在示例 15-6 中，创建了一个 `i32` 值的引用，接着使用解引用运算符来跟踪所引用的值：

文件名：src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

示例 15-6：使用解引用运算符来跟踪 `i32` 值的引用

变量 `x` 存放了一个 `i32` 值 `5`。`y` 等于 `x` 的一个引用。可以断言 `x` 等于 `5`。然而，如果希望对 `y` 的值做出断言，必须使用 `*y` 来追踪引用所指向的值（也就是**解引用**），这样编译器就可以比较实际的值了。一旦解引用了 `y`，就可以访问 `y` 所指向的整型值并可以与 `5` 做比较。

相反如果尝试编写 `assert_eq!(5, y);`，则会得到如下编译错误：

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
  |
  = help: the trait `PartialEq<&{integer}>` is not implemented for `{integer}`
  = note: this error originates in the macro `assert_eq` (in Nightly builds, run
with -Z macro-backtrace for more info)
```

```
For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` (bin "deref-example") due to 1 previous
error
```

不允许比较数字的引用与数字，因为它们是不同的类型。必须使用解引用运算符追踪引用所指向的值。

像引用一样使用 `Box<T>`

可以使用 `Box<T>` 代替引用来重写示例 15-6 中的代码，示例 15-7 中 `Box<T>` 上使用的解引用运算符与示例 15-6 中引用上使用的解引用运算符有着一样的功能：

文件名：src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

示例 15-7：在 `Box<i32>` 上使用解引用运算符

示例 15-7 相比示例 15-6 主要不同的地方就是将 `y` 设置为一个指向 `x` 值拷贝的 `Box<T>` 实例，而不是指向 `x` 值的引用。在最后的断言中，可以使用解引用运算符以 `y` 为引用时相同的方式追踪 `Box<T>` 的指针。接下来让我们通过实现自己的类型来探索 `Box<T>` 能这么做有何特殊之处。

自定义智能指针

为了体会默认情况下智能指针与引用的不同，让我们创建一个类似于标准库提供的 `Box<T>` 类型的智能指针。接着学习如何增加使用解引用运算符的功能。

从根本上说，`Box<T>` 被定义为包含一个元素的元组结构体，所以示例 15-8 以相同的方式定义了 `MyBox<T>` 类型。我们还定义了 `new` 函数来对应定义于 `Box<T>` 的 `new` 函数：

文件名：src/main.rs

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

示例 15-8：定义 `MyBox<T>` 类型

这里定义了一个结构体 `MyBox` 并声明了一个泛型参数 `T`，因为我们希望其可以存放任何类型的值。`MyBox` 是一个包含 `T` 类型元素的元组结构体。`MyBox::new` 函数获取一个 `T` 类型的参数并返回一个存放传入值的 `MyBox` 实例。

尝试将示例 15-7 中的代码加入示例 15-8 中并修改 `main` 使用我们定义的 `MyBox<T>` 类型代替 `Box<T>`。示例 15-9 中的代码不能编译，因为 Rust 不知道如何解引用 `MyBox`：

文件名：src/main.rs



```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

示例 15-9：尝试以使用引用和 `Box<T>` 相同的方式使用 `MyBox<T>`

下面是相应的编译错误：

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^

For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` (bin "deref-example") due to 1 previous
error
```

`MyBox<T>` 类型不能解引用，因为我们尚未在该类型上实现这个功能。为了启用 `*` 运算符的解引用功能，需要实现 `Deref` trait。

实现 `Deref` trait

如第十章 “为类型实现 trait” 部分所讨论的，为了实现 trait，需要提供 trait 所需的方法实现。`Deref` trait，由标准库提供，要求实现名为 `deref` 的方法，其借用 `self` 并返回一个内部数据的引用。示例 15-10 包含定义于 `MyBox` 之上的 `Deref` 实现：

文件名：src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

示例 15-10：`MyBox<T>` 上的 `Deref` 实现

`type Target = T;` 语法定义了用于此 trait 的关联类型。关联类型是一个稍有不同的定义泛型参数的方式，现在还无需过多地担心它；第二十章会详细介绍。

`deref` 方法体中写入了 `&self.0`，这样 `deref` 返回了我希望通过 `*` 运算符访问的值的引用。回忆一下第五章 “使用没有命名字段的元组结构体来创建不同的类型” 部分 `.0` 用来访问元组结构体的第一个元素。示例 15-9 中的 `main` 函数中对 `MyBox<T>` 值的 `*` 调用现在可以编译并能通过断言了！

没有 `Deref` trait 的话，编译器只会解引用 `&` 引用类型。`deref` 方法向编译器提供了获取任何实现了 `Deref` trait 的类型的值，并且调用这个类型的 `deref` 方法来获取一个它知道如何解引用的 `&` 引用的能力。

当我们在示例 15-9 中输入 `*y` 时，Rust 事实上在底层运行了如下代码：

```
*(y.deref())
```

Rust 将 `*` 运算符替换为先调用 `deref` 方法再进行普通解引用的操作，如此我们便不用担心是否还需手动调用 `deref` 方法了。Rust 的这个特性可以让我们写出行为一致的代码，无论面对的是常规引用还是实现了 `Deref` 的类型。

`deref` 方法返回值的引用，以及 `*(y.deref())` 括号外边的普通解引用仍为必须的原因在于所有权。如果 `deref` 方法直接返回值而不是值的引用，其值将被移出 `self`。在这里以及大部分使用解引用运算符的情况下我们并不希望获取 `MyBox<T>` 内部值的所有权。

注意，每次当我们在代码中使用 `*` 时， `*` 运算符都被替换成了先调用 `deref` 方法再接着使用 `*` 解引用的操作，且只会发生一次，不会对 `*` 操作符无限递归替换，解引用出上面 `i32` 类型的值就停止了，这个值与示例 15-9 中 `assert_eq!` 的 `5` 相匹配。

函数和方法的隐式 Deref 强制转换

Deref 强制转换（*deref coercions*）将实现了 `Deref` trait 的类型的引用转换为另一种类型的引用。例如，Deref 强制转换可以将 `&String` 转换为 `&str`，因为 `String` 实现了 `Deref` trait 因此可以返回 `&str`。Deref 强制转换是 Rust 在函数或方法传参上的一种便利操作，并且只能作用于实现了 `Deref` trait 的类型。当这种特定类型的引用作为实参传递给和形参类型不同的函数或方法时将自动进行。这时会有一系列的 `deref` 方法被调用，把我们提供的类型转换成了参数所需的类型。

Deref 强制转换的加入使得 Rust 程序员编写函数和方法调用时无需增加过多显式使用 `&` 和 `*` 的引用和解引用。这个功能也使得我们可以编写更多同时作用于引用或智能指针的代码。

作为展示 Deref 强制转换的实例，让我们使用示例 15-8 中定义的 `MyBox<T>`，以及示例 15-10 中增加的 `Deref` 实现。示例 15-11 展示了一个有着字符串 slice 参数的函数定义：

文件名：src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}
```

示例 15-11：`hello` 函数有着 `&str` 类型的参数 `name`

可以使用字符串 slice 作为参数调用 `hello` 函数，比如 `hello("Rust");`。Deref 强制转换使得用 `MyBox<String>` 类型值的引用调用 `hello` 成为可能，如示例 15-12 所示：

文件名：src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

示例 15-12：因为 Deref 强制转换，使用 `MyBox<String>` 的引用调用 `hello` 是可行的

这里使用 `&m` 调用 `hello` 函数，其为 `MyBox<String>` 值的引用。因为示例 15-10 中在 `MyBox<T>` 上实现了 `Deref` trait，Rust 可以通过 `deref` 调用将 `&MyBox<String>` 变为 `&String`。标准库中提供了 `String` 上的 `Deref` 实现，其会返回字符串 slice，这可以在 `Deref` 的 API 文档中看到。Rust 再次调用 `deref` 将 `&String` 变为 `&str`，这就符合 `hello` 函数的定义了。

如果 Rust 没有实现 Deref 强制转换，为了使用 `&MyBox<String>` 类型的值调用 `hello`，则不得不编写示例 15-13 中的代码来代替示例 15-12：

文件名：src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

示例 15-13：如果 Rust 没有 Deref 强制转换则必须编写的代码

`(*m)` 将 `MyBox<String>` 解引用为 `String`。接着 `&` 和 `[..]` 获取了整个 `String` 的字符串 slice 来匹配 `hello` 的签名。没有 Deref 强制转换所有这些符号混在一起将更难以读写和理解。Deref 强制转换使得 Rust 自动的帮我们处理这些转换。

当所涉及到的类型定义了 `Deref` trait，Rust 会分析这些类型并使用任意多次 `Deref::deref` 调用以获得匹配参数的类型。这些解析都发生在编译时，所以利用 Deref 强制转换并没有运行时开销！

Deref 强制转换如何与可变性交互

类似于如何使用 `Deref` trait 重载不可变引用的 `*` 运算符，Rust 提供了 `DerefMut` trait 用于重载可变引用的 `*` 运算符。

Rust 在发现类型和 trait 实现满足三种情况时会进行 Deref 强制转换：

1. 当 `T: Deref<Target=U>` 时从 `&T` 到 `&U`。
2. 当 `T: DerefMut<Target=U>` 时从 `&mut T` 到 `&mut U`。
3. 当 `T: Deref<Target=U>` 时从 `&mut T` 到 `&U`。

头两个情况除了第二种实现了可变性之外是相同的：第一种情况表明如果有一个 `&T`，而 `T` 实现了返回 `U` 类型的 `Deref`，则可以透明地得到 `&U`。第二种情况表明对于可变引用也有着相同的行为。

第三个情况有些微妙：Rust 也会将可变引用强转为不可变引用。但反之是**不可能** 的：不可变引用永远也不能强转为可变引用。因为根据借用规则，如果有一个可变引用，其必须是这些数据的唯一引用（否则程序将无法编译）。将一个可变引用转换为不可变引用永远也不会打破借用规则。将不可变引用转换为可变引用则需要初始的不可变引用是数据唯一的不可变引用，而借用规则无法保证这一点。因此，Rust 无法假设将不可变引用转换为可变引用是可能的。

使用 `Drop` Trait 运行清理代码

对于智能指针模式来说第二个重要的 trait 是 `Drop`，其允许我们在值要离开作用域时自定义要执行的操作。可以为任何类型提供 `Drop` trait 的实现，同时所指定的代码被用于释放类似于文件或网络连接的资源。

我们在智能指针上下文中讨论 `Drop`，是因为在实现智能指针时几乎总会用到 `Drop` trait。例如，当 `Box<T>` 被丢弃时会释放 box 指向的堆空间。

在其他一些语言中的某些类型，我们不得不记住在每次使用完那些类型的智能指针实例后调用清理内存或资源的代码。常见示例包括文件句柄（file handles）、套接字（sockets）和锁（locks）。如果忘记的话，运行代码的系统可能会因为负荷过重而崩溃。在 Rust 中，可以指定每当值离开作用域时被执行的代码，编译器会自动插入这些代码。于是我们就不需要在程序中到处编写在实例结束时清理这些变量的代码 —— 而且还不会泄漏资源！

指定在值离开作用域时应该执行的代码的方式是实现 `Drop` trait。`Drop` trait 要求实现一个叫做 `drop` 的方法，它获取一个 `self` 的可变引用。为了能够看出 Rust 何时调用 `drop`，让我们暂时使用 `println!` 语句实现 `drop`。

示例 15-14 展示了唯一定制功能就是当其实例离开作用域时，打印出 `Dropping CustomSmartPointer!` 的结构体 `CustomSmartPointer`，这会演示 Rust 何时运行 `drop` 方法：

文件名：src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

示例 15-14：结构体 `CustomSmartPointer`，其实现了放置清理代码的 `Drop` trait

`Drop` trait 包含在 prelude 中，因此无需将其引入作用域。我们在 `CustomSmartPointer` 上实现了 `Drop` trait，并提供了一个调用 `println!` 的 `drop` 方法实现。`drop` 函数体是放置任何当类型实例离开作用域时期望运行的逻辑的地方。这里选择打印一些文本以可视化地展示 Rust 何时调用 `drop`。

在 `main` 中，我们新建了两个 `CustomSmartPointer` 实例并打印出了 `CustomSmartPointer created.`。在 `main` 的结尾，`CustomSmartPointer` 的实例会离开作用域，

而 Rust 会调用放置于 `drop` 方法中的代码，打印出最后的信息。注意无需显式调用 `drop` 方法：

当运行这个程序，会出现如下输出：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!
```

当实例离开作用域 Rust 会自动调用 `drop`，并调用我们指定的代码。变量以被创建时相反的顺序被丢弃，所以 `d` 在 `c` 之前被丢弃。这个例子的作用是给了我们一个 drop 方法如何工作的可视化指导，不过通常需要指定类型所需执行的清理代码而不是打印信息。

不幸的是，禁用自动 `drop` 功能并不是一件容易的事。通常也不需要禁用 `drop` ；整个 `Drop` trait 存在的意义在于其是自动处理的。然而，有时你可能需要提早清理某个值。一个例子是当使用智能指针管理锁时；你可能希望强制运行 `drop` 方法来释放锁以便作用域中的其他代码可以获取锁。Rust 并不允许我们主动调用 `Drop` trait 的 `drop` 方法；当我们希望在作用域结束之前就强制释放变量的话，我们应该使用的是由标准库提供的 `std::mem::drop` 函数。

如果我们像是示例 15-14 那样尝试调用 `Drop` trait 的 `drop` 方法，就会得到像示例 15-15 那样的编译错误：

文件名：src/main.rs



```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

示例 15-15：尝试手动调用 `Drop` trait 的 `drop` 方法提早清理

如果尝试编译代码会得到如下错误：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |       ^^^^ explicit destructor calls not allowed
   |
help: consider using `drop` function
   |
16 |     drop(c);
   |     +++++ ~
```

```
For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` (bin "drop-example") due to 1 previous
error
```

错误信息表明不允许显式调用 `drop`。错误信息使用了术语**析构函数**（*destructor*），这是一个清理实例的函数的通用编程术语。**析构函数** 对应创建实例的**构造函数**（*constructor*）。Rust 中的 `drop` 函数就是这么一个析构函数。

Rust 不允许我们显式调用 `drop` 因为 Rust 仍然会在 `main` 的结尾对值自动调用 `drop`，这会导致一个 *double free* 错误，因为 Rust 会尝试清理相同的值两次。

因为不能禁用当值离开作用域时自动插入的 `drop`，并且不能显式调用 `drop` 方法。如果我们需要强制提早清理值，可以使用 `std::mem::drop` 函数。

`std::mem::drop` 函数不同于 `Drop` trait 中的 `drop` 方法。可以通过传递希望强制丢弃的值作为参数。该函数位于 prelude，所以我们可以修改示例 15-15 中的 `main` 来调用 `drop` 函数。如示例 15-16 所示：

文件名：src/main.rs

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

示例 15-16: 在值离开作用域之前调用 `std::mem::drop` 显式清理

运行这段代码会打印出如下：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

`Dropping CustomSmartPointer with data `some data`!` 出现在 `CustomSmartPointer created.` 和 `CustomSmartPointer dropped before the end of main.` 之间，表明了 `drop` 方法被调用了并在此丢弃了 `c`。

`Drop` trait 实现中指定的代码可以用于多种场景，来使得清理变得方便和安全：比如可以用其创建我们自己的内存分配器！通过 `Drop` trait 和 Rust 所有权系统，你无需担心之后的代码清理，因为 Rust 会自动完成这些工作。

你也不必担心由于不小心清理仍在使用的值而导致的问题：所有权系统确保引用总是有效的，也会确保 `drop` 只会在值不再被使用时被调用一次。

现在我们学习了 `Box<T>` 和一些智能指针的特性，让我们聊聊标准库中定义的其他几种智能指针。

`Rc<T>` 引用计数智能指针

大部分情况下所有权是非常明确的：可以准确地知道哪个变量拥有某个值。然而，有些情况单个值可能会有多个所有者。例如，在图数据结构中，多个边可能指向相同的节点，而这个节点从概念上讲为所有指向它的边所拥有。节点在没有任何边指向它从而没有任何所有者之前，都不应该被清理掉。

为了启用多所有权需要显式地使用 Rust 类型 `Rc<T>`，其为**引用计数**（*reference counting*）的缩写。引用计数意味着记录一个值的引用数量来知晓这个值是否仍在被使用。如果某个值有零个引用，就代表没有任何有效引用并可以被清理。

可以将其想象为客厅中的电视。当一个人进来看电视时，他打开电视。其他人也可以进来看电视。当最后一个人离开房间时，他关掉电视因为它不再被使用了。如果某人在其他人还在看的时候就关掉了电视，正在看电视的人肯定会抓狂的！

`Rc<T>` 用于当我们希望在堆上分配一些内存供程序的多个部分读取，而且无法在编译时确定程序的哪一部分会最后结束使用它的时候。如果确实知道哪部分是最后一个结束使用的话，就可以令其成为数据的所有者，正常的所有权规则就可以在编译时生效。

注意 `Rc<T>` 只能用于单线程场景；在第十六章讨论并发时会涉及到如何在多线程程序中进行引用计数。

使用 `Rc<T>` 共享数据

让我们回到示例 15-5 中使用 `Box<T>` 定义 cons list 的例子。这一次，我们希望创建两个共享第三个列表所有权的列表，其概念将会看起来如图 15-3 所示：

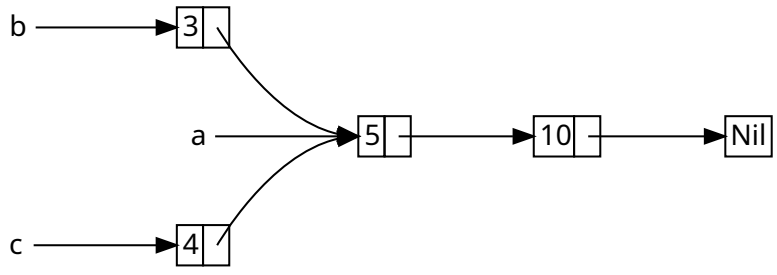


图 15-3: 两个列表，`b` 和 `c`, 共享第三个列表 `a` 的所有权

列表 `a` 包含 `5` 之后是 `10`，之后是另两个列表：`b` 从 `3` 开始而 `c` 从 `4` 开始。`b` 和 `c` 会接上包含 `5` 和 `10` 的列表 `a`。换句话说，这两个列表会尝试共享第一个列表所包含的 `5` 和 `10`。

尝试使用 `Box<T>` 定义的 `List` 实现并不能工作，如示例 15-17 所示：

文件名：src/main.rs



```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

示例 15-17: 展示不能用两个 `Box<T>` 的列表尝试共享第三个列表的所有权

编译会得出如下错误：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement
the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` (bin "cons-list") due to 1 previous error
```

`Cons` 成员拥有其储存的数据，所以当创建 `b` 列表时，`a` 被移动进了 `b` 这样 `b` 就拥有了 `a`。接着当再次尝试使用 `a` 创建 `c` 时，这不被允许，因为 `a` 的所有权已经被移动。

可以改变 `Cons` 的定义来存放一个引用，不过接着必须指定生命周期参数。通过指定生命周期参数，表明列表中的每一个元素都至少与列表本身存在的一样久。这是示例 15-17 中元素与列表的情况，但并不是所有情况都如此。

相反，我们修改 `List` 的定义为使用 `Rc<T>` 代替 `Box<T>`，如列表 15-18 所示。现在每一个 `Cons` 变量都包含一个值和一个指向 `List` 的 `Rc<T>`。当创建 `b` 时，不同于获取 `a` 的所有权，这里会克隆 `a` 所包含的 `Rc<List>`，这会将引用计数从 1 增加到 2 并允许 `a` 和 `b` 共享 `Rc<List>` 中数据的所有权。创建 `c` 时同样会克隆 `a`，这会将引用计数从 2 增加为 3。每次调用 `Rc::clone`，`Rc<List>` 中数据的引用计数都会增加，直到有零个引用之前其数据都不会被清理。

文件名：src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

示例 15-18: 使用 `Rc<T>` 定义的 `List`

需要使用 `use` 语句将 `Rc<T>` 引入作用域，因为它不在 prelude 中。在 `main` 中创建了存放 5 和 10 的列表并将其存放在 `a` 的新的 `Rc<List>` 中。接着当创建 `b` 和 `c` 时，调用 `Rc::clone` 函数并传递 `a` 中 `Rc<List>` 的引用作为参数。

也可以调用 `a.clone()` 而不是 `Rc::clone(&a)`，不过在这里 Rust 的习惯是使用 `Rc::clone`。`Rc::clone` 的实现并不像大部分类型的 `clone` 实现那样对所有数据进行深拷贝。`Rc::clone` 只会增加引用计数，这并不会花费多少时间。深拷贝可能会花费很长时间。通过使用 `Rc::clone` 进行引用计数，可以在视觉上区别深拷贝类的克隆和增加引用计数类的克隆。当查找代码中的性能问题时，只需考虑深拷贝类的克隆而无需考虑 `Rc::clone` 调用。

克隆 `Rc<T>` 会增加引用计数

让我们修改示例 15-18 的代码以便观察创建和丢弃 `a` 中 `Rc<List>` 的引用时引用计数的变化。

在示例 15-19 中，修改了 `main` 以便将列表 `c` 置于内部作用域中，这样就可以观察当 `c` 离开作用域时引用计数如何变化。

文件名：src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

示例 15-19：打印出引用计数

在程序中每个引用计数变化的点，会打印出引用计数，其值可以通过调用 `Rc::strong_count` 函数获得。这个函数叫做 `strong_count` 而不是 `count` 是因为 `Rc<T>` 也有 `weak_count`；在“避免引用循环：将 `Rc<T>` 变为 `Weak<T>`”部分会讲解 `weak_count` 的用途。

这段代码会打印出：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

我们能够看到 `a` 中 `Rc<List>` 的初始引用计数为 1，接着每次调用 `clone`，计数会增加 1。当 `c` 离开作用域时，计数减 1。不必像调用 `Rc::clone` 增加引用计数那样调用一个函数来减少计数；`Drop` trait 的实现当 `Rc<T>` 值离开作用域时自动减少引用计数。

从这个例子我们所不能看到的是，在 `main` 的结尾当 `b` 然后是 `a` 离开作用域时，此处计数会是 0，同时 `Rc<List>` 被完全清理。使用 `Rc<T>` 允许一个值有多个所有者，引用计数则确保只要任何所有者依然存在其值也保持有效。

通过不可变引用， `Rc<T>` 允许在程序的多个部分之间只读地共享数据。如果 `Rc<T>` 也允许多个可变引用，则会违反第四章讨论的借用规则之一：相同位置的多个可变借用可能造成数据竞争和不一致。不过可以修改数据是非常有用的！在下一部分，我们将讨论内部可变性模式和 `RefCell<T>` 类型，它可以与 `Rc<T>` 结合使用来处理不可变性的限制。

`RefCell<T>` 和内部可变性模式

内部可变性（*Interior mutability*）是 Rust 中的一个设计模式，它允许你即使在有不可变引用时也可以改变数据，这通常是借用规则所不允许的。为了改变数据，该模式在数据结构中使用 `unsafe` 代码来模糊 Rust 通常的可变性和借用规则。不安全代码表明我们在手动检查这些规则而不是让编译器替我们检查。第二十章会更详细地介绍不安全代码。

当可以确保代码在运行时会遵守借用规则，即使编译器不能保证的情况，可以选择使用那些运用内部可变性模式的类型。所涉及的 `unsafe` 代码将被封装进安全的 API 中，而外部类型仍然是不可变的。

让我们通过遵循内部可变性模式的 `RefCell<T>` 类型来开始探索。

通过 `RefCell<T>` 在运行时强制借用规则

不同于 `Rc<T>`，`RefCell<T>` 代表其数据的唯一的所有权。那么是什么让 `RefCell<T>` 不同于像 `Box<T>` 这样的类型呢？回忆一下第四章所学的借用规则：

- 在任意给定时刻，只能拥有一个可变引用或任意数量的不可变引用**之一**（而不是两者）。
- 引用必须始终有效。

对于引用和 `Box<T>`，借用规则的不可变性（invariants）在编译时就会被强制执行。对于 `RefCell<T>`，这些不可变性作用于**运行时**。对于引用，如果违反这些规则，会得到一个编译错误。而对于 `RefCell<T>`，如果违反这些规则程序会 panic 并退出。

在编译时检查借用规则的优势是这些错误将在开发过程的早期被捕获，同时对运行时没有性能影响，因为所有的分析都提前完成了。为此，在编译时检查借用规则是大部分情况的最佳选择，这也正是其为何是 Rust 的默认行为。

相反在运行时检查借用规则的好处则是允许出现特定内存安全的场景，而它们在编译时检查中是不允许的。静态分析，正如 Rust 编译器，是天生保守的。但代码的一些属性不可能通过分析代码发现：其中最著名的就是停机问题（Halting Problem），这超出了本书的范畴，不过如果你感兴趣的话这是一个值得研究的有趣主题。

因为一些分析是不可能的，如果 Rust 编译器不能通过所有权规则编译，它可能会拒绝一个正确的程序；从这种角度考虑它是保守的。如果 Rust 接受不正确的程序，那么用户也就不会相信 Rust 所做的保证了。然而，如果 Rust 拒绝正确的程序，虽然会给程序员带来不便，但不会带来灾难。`RefCell<T>` 正是用于当你确信代码遵守借用规则，而编译器不能理解和确定的时候。

类似于 `Rc<T>`，`RefCell<T>` 只能用于单线程场景。如果尝试在多线程上下文中使用 `RefCell<T>`，会得到一个编译错误。第十六章会介绍如何在多线程程序中实现 `RefCell<T>` 的功能。

如下为选择 `Box<T>`，`Rc<T>` 或 `RefCell<T>` 的理由：

- `Rc<T>` 允许相同数据有多个所有者；`Box<T>` 和 `RefCell<T>` 则只有单一所有者。
- `Box<T>` 允许在编译时执行不可变或可变借用检查；`Rc<T>` 仅允许在编译时执行不可变借用检查；`RefCell<T>` 允许在运行时执行不可变或可变借用检查。
- 因为 `RefCell<T>` 允许在运行时执行可变借用检查，所以我们可以在即便 `RefCell<T>` 自身是不可变的情况下修改其内部的值。

在不可变值内部改变值就是**内部可变性**（*interior mutability*）模式。让我们看看何时内部可变性是有用的，并讨论这是如何成为可能的。

内部可变性：不可变值的可变借用

借用规则的一个推论是当有一个不可变值时，不能可变地借用它。例如，如下代码不能编译：

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```



如果尝试编译，会得到如下错误：

```
$ cargo run
   Compiling borrowing v0.1.0 (file:///projects/borrowing)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable
 --> src/main.rs:3:13
  |
3 |     let y = &mut x;
  |             ^^^^^^ cannot borrow as mutable
  |
help: consider changing this to be mutable
  |
2 |     let mut x = 5;
  |         +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `borrowing` (bin "borrowing") due to 1 previous error
```

然而，特定情况下，令一个值在其方法内部能够修改自身，而在其他代码中仍视为不可变，是很有用的。值方法外部的代码就不能修改其值了。`RefCell<T>` 是一个获得内部可变性的方法。`RefCell<T>` 并没有完全绕开借用规则，编译器中的借用检查器允许内部可变性并相应地在运行时检查借用规则。如果违反了这些规则，会出现 panic 而不是编译错误。

让我们通过一个实际的例子来探索何处可以使用 `RefCell<T>` 来修改不可变值并看看为何这么做是有意义的。

内部可变性的用例：mock 对象

有时在测试中程序员会用某个类型替换另一个类型，以便观察特定的行为并断言它是被正确实现的。这个占位符类型被称为 **测试替身**（*test double*）。就像电影制作中的替身演员（*stunt double*）一样，替代演员完成高难度的场景。测试替身在运行测试时替代某个类型。**mock 对象** 是特定类型的测试替身，它们记录测试过程中发生了什么以便可以断言操作是正确的。

Rust 并不像其他语言那样在标准库中提供内建的对象模型，Rust 也没有像其他语言那样在标准库中内建 mock 对象功能，不过我们确实可以创建一个与 mock 对象有着相同功能的结构体。

如下是一个我们想要测试的场景：我们在编写一个记录某个值与最大值的差距的库，并根据当前值与最大值的差距来发送消息。例如，这个库可以用于记录用户所允许的 API 调用数量限额。

该库只提供记录与最大值的差距，以及何种情况发送什么消息的功能。使用此库的程序则期望提供实际发送消息的机制：程序可以选择记录一条消息、发送 email、发送短信等等。库本身无需知道这些细节；只需实现其提供的 `Messenger` trait 即可。示例 15-20 展示了库代码：

文件名：src/lib.rs

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}
```

示例 15-20：一个记录某个值与最大值差距的库，并根据此值的特定级别发出警告

这些代码中一个重要部分是拥有一个方法 `send` 的 `Messenger` trait，其获取一个 `self` 的不可变引用和文本信息。这个 trait 是 mock 对象所需要实现的接口库，这样 mock 就能像一个真正的对象那样使用了。另一个重要的部分是我们需要测试 `LimitTracker` 的 `set_value` 方法的行为。可以改变传递的 `value` 参数的值，不过 `set_value` 并没有返回任何可供断言的值。我们希望能够说，如果我们创建一个实现了 `Messenger` trait 和具有特定 `max` 值的 `LimitTracker` 时，当传递不同 `value` 值时，消息发送者应被告知发送合适的消息。

我们所需的 mock 对象是，调用 `send` 并不实际发送 email 或消息，而是只记录信息被通知要发送了。可以新建一个 mock 对象实例，用其创建 `LimitTracker`，调用 `LimitTracker` 的 `set_value` 方法，然后检查 mock 对象是否有我们期望的消息。示例 15-21 展示了一个如此尝试的 mock 对象实现，不过借用检查器并不允许：

文件名：src/lib.rs



```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

示例 15-21：尝试实现 `MockMessenger`，借用检查器不允许这么做

测试代码定义了一个 `MockMessenger` 结构体，其 `sent_messages` 字段为一个 `String` 值的 `Vec` 用来记录被告知发送的消息。我们还定义了一个关联函数 `new` 以便于新建从空消息列表开始的 `MockMessenger` 值。接着为 `MockMessenger` 实现 `Messenger` trait 这样就可以为 `LimitTracker` 提供一个 `MockMessenger`。在 `send` 方法的定义中，获取传入的消息作为参数并储存在 `MockMessenger` 的 `sent_messages` 列表中。

在测试中，我们测试了当 `LimitTracker` 被告知将 `value` 设置为超过 `max` 值 75% 的某个值。首先新建一个 `MockMessenger`，其从空消息列表开始。接着新建一个 `LimitTracker` 并传递新建 `MockMessenger` 的引用和 `max` 值 `100`。我们使用值 `80` 调用 `LimitTracker` 的 `set_value` 方法，这超过了 100 的 75%。接着断言 `MockMessenger` 中记录的消息列表应该有一条消息。

然而，这个测试存在一个问题，如下所示：

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.sent_messages` as mutable, as it is behind a `&`
reference
  --> src/lib.rs:58:13
   |
58 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ `self` is a `&` reference, so the data it
refers to cannot be borrowed as mutable
   |
help: consider changing this to be a mutable reference in the `impl` method and
the `trait` definition
   |
2  ~     fn send(&mut self, msg: &str);
3  | }
...
56 |     impl Messenger for MockMessenger {
57 ~         fn send(&mut self, message: &str) {
   |

For more information about this error, try `rustc --explain E0596`.
error: could not compile `limit-tracker` (lib test) due to 1 previous error
```

不能修改 `MockMessenger` 来记录消息，因为 `send` 方法接收的是对 `self` 的不可变引用。我们也不能采纳错误提示中将 `&self` 改为 `&mut self` 的建议，因为那样既要在 `impl` 方法中修改签名，也要在 `Messenger` trait 定义中修改签名。我们并不希望仅为了测试而改变 `Messenger` trait。相反，我们需要想办法让测试代码与现有设计兼容，正常工作。

这正是内部可变性的用武之地！我们将通过 `RefCell` 来储存 `sent_messages`，然后 `send` 将能够修改 `sent_messages` 并储存消息。示例 15-22 展示了代码：

文件名：src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }
```

```
    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

示例 15-22：使用 `RefCell<T>` 能够在外部值被认为是不可变的情况下修改内部值

现在 `sent_messages` 字段的类型是 `RefCell<Vec<String>>` 而不是 `Vec<String>`。在 `new` 函数中在空 vector 外层创建了一个 `RefCell<Vec<String>>` 实例。

对于 `send` 方法的实现，第一个参数仍为 `self` 的不可变借用，这是符合 trait 定义的。我们调用 `self.sent_messages` 中 `RefCell<Vec<String>>` 的 `borrow_mut` 方法来获取 `RefCell<Vec<String>>` 中值的可变引用，这是一个 vector。接着可以对 vector 的可变引用调用 `push` 以便记录测试过程中看到的消息。

最后必须做出的修改位于断言中：为了看到其内部 vector 中有多少个项，需要调用 `RefCell<Vec<String>>` 的 `borrow` 以获取 vector 的不可变引用。

现在我们见识了如何使用 `RefCell<T>`，让我们研究一下它怎样工作的！

`RefCell<T>` 在运行时记录借用

当创建不可变和可变引用时，我们分别使用 `&` 和 `&mut` 语法。对于 `RefCell<T>` 来说，则是 `borrow` 和 `borrow_mut` 方法，这属于 `RefCell<T>` 安全 API 的一部分。`borrow` 方法返回 `Ref<T>` 类型的智能指针，`borrow_mut` 方法返回 `RefMut<T>` 类型的智能指针。这两个类型都实现了 `Deref`，所以可以当作常规引用对待。

`RefCell<T>` 记录当前有多少个活动的 `Ref<T>` 和 `RefMut<T>` 智能指针。每次调用 `borrow`，`RefCell<T>` 将活动的不可变借用计数加一。当 `Ref<T>` 值离开作用域时，不可变借用计数减一。就像编译时借用规则一样，`RefCell<T>` 在任何时候只允许有多个不可变借用或一个可变借用。

如果我们尝试违反这些规则，相比引用时的编译时错误，`RefCell<T>` 的实现会在运行时出现 panic。示例 15-23 展示了对示例 15-22 中 `send` 实现的修改，这里我们故意尝试在相同作用域创建两个可变借用以便演示 `RefCell<T>` 不允许我们在运行时这么做。

文件名：src/lib.rs



```
    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            let mut one_borrow = self.sent_messages.borrow_mut();
            let mut two_borrow = self.sent_messages.borrow_mut();

            one_borrow.push(String::from(message));
            two_borrow.push(String::from(message));
        }
    }
```

示例 15-23：在同一作用域中创建两个可变引用并观察 `RefCell<T>` 将会 panic

这里为 `borrow_mut` 返回的 `RefMut` 智能指针创建了 `one_borrow` 变量。接着用相同的方式在变量 `two_borrow` 创建了另一个可变借用。这会在相同作用域中创建两个可变引用，这是不允许的。当运行库的测试时，示例 15-23 编译时不会有任何错误，不过测试会失败：

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/limit_tracker-
e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----

thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at src/
lib.rs:60:53:
already borrowed: BorrowMutError
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

注意代码 panic 和信息 `already borrowed: BorrowMutError`。这也就是 `RefCell<T>` 如何在运行时处理违反借用规则的情况。

像我们这里这样选择在运行时捕获借用错误而不是编译时意味着会发现在开发过程的后期才会发现的潜在错误，甚至有可能发布到生产环境才会发现。还会因为在运行时而不是编译时记录借用而导致少量的运行时性能惩罚。然而，使用 `RefCell` 使得在只允许不可变值的上下文中编写修改自身以记录消息的 mock 对象成为可能。虽然有取舍，但是我们可以选择使用 `RefCell<T>` 来获得比常规引用所能提供的更多的功能。

结合 `Rc<T>` 和 `RefCell<T>` 来拥有多个可变数据所有者

`RefCell<T>` 的一个常见用法是与 `Rc<T>` 结合。回忆一下 `Rc<T>` 允许对相同数据有多个所有者，不过只能提供数据的不可变访问。如果有一个储存了 `RefCell<T>` 的 `Rc<T>` 的话，就可以得到有多个所有者**并且**可以修改的值了！

例如，回忆示例 15-18 的 cons list 的例子中使用 `Rc<T>` 使得多个列表共享另一个列表的所有权。因为 `Rc<T>` 只存放不可变值，所以一旦创建了这些列表值后就不能修改。让我们加入 `RefCell<T>` 来获得修改列表中值的能力。示例 15-24 展示了通过在 `Cons` 定义中使用 `RefCell<T>`，我们就允许修改所有列表中的值了：

文件名：src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {a:?}");
    println!("b after = {b:?}");
    println!("c after = {c:?}");
}
```

示例 15-24：使用 `Rc<RefCell<i32>>` 创建可以修改的 `List`

这里创建了一个 `Rc<RefCell<i32>>` 实例并储存在变量 `value` 中以便之后直接访问。接着在 `a` 中用包含 `value` 的 `Cons` 变体创建了一个 `List`。需要克隆 `value` 以便 `a` 和 `value` 都能拥有其内部值 `5` 的所有权，而不是将所有权从 `value` 移动到 `a` 或者让 `a` 借用 `value`。

我们将列表 `a` 封装进了 `Rc<T>` 这样当创建列表 `b` 和 `c` 时，它们都可以引用 `a`，正如示例 15-18 一样。

一旦创建了列表 `a`、`b` 和 `c`，我们将 `value` 的值加 10。为此对 `value` 调用了 `borrow_mut`，这里使用了第五章讨论的自动解引用功能（“`->` 运算符到哪去了？”部分）来解引用 `Rc<T>` 以获取其内部的 `RefCell<T>` 值。`borrow_mut` 方法返回 `RefMut<T>` 智能指针，可以对其使用解引用运算符并修改其内部值。

当我们打印出 `a`、`b` 和 `c` 时，可以看到它们都拥有修改后的值 `15` 而不是 `5`：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.63s
     Running `target/debug/cons-list`
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

这个技巧非常巧妙！通过使用 `RefCell<T>`，我们可以拥有一个表面上不可变的 `List`，不过可以使用 `RefCell<T>` 中提供内部可变性的方法来在需要时修改数据。`RefCell<T>` 的运行时借用规则检查也确实保护我们免于出现数据竞争，有时为了数据结构的灵活性而付出一些性能是值得的。注意 `RefCell<T>` 不能用于多线程代码！`Mutex<T>` 是一个线程安全版本的 `RefCell<T>`，我们会在第十六章讨论 `Mutex<T>`。

引用循环与内存泄漏

Rust 的内存安全性保证使其难以意外地制造永远也不会被清理的内存（被称为 **内存泄漏**，*memory leak*），但并非不可能。Rust 并不保证完全防止内存泄漏，这意味着内存泄漏在 Rust 中被认为是内存安全的。这一点可以通过 `Rc<T>` 和 `RefCell<T>` 看出 Rust 允许出现内存泄漏：创建引用循环的可能性是存在的。这会造成内存泄漏，因为每一项的引用计数永远也到不了 0，持有的数据也就永远不会被释放。

制造引用循环

让我们看看引用循环是如何发生的以及如何避免它，以示例 15-25 中的 `List` 枚举和 `tail` 方法的定义开始：

文件名：src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

示例 15-25: 一个存放 `RefCell` 的 cons list 定义，这样可以修改 `Cons` 变体所引用的数据

这里采用了示例 15-5 中 `List` 定义的另一种变体。现在 `Cons` 变体的第二个元素是 `RefCell<Rc<List>>`，这意味着不同于像示例 15-24 那样能够修改 `i32` 的值，我们希望能够修改 `Cons` 变体所指向的 `List`。这里还增加了一个 `tail` 方法来方便我们在有 `Cons` 变体的时候访问其第二项。

在示例 15-26 中增加了一个 `main` 函数，其使用了示例 15-25 中的定义。这些代码在 `a` 中创建了一个列表，一个指向 `a` 中列表的 `b` 列表，接着修改 `a` 中的列表指向 `b` 中的列表，这会创建一个引用循环。在这个过程的多个位置有 `println!` 语句展示引用计数。

文件：src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
```

```
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack.
    // println!("a next item = {:?}", a.tail());
}
```

示例 15-26：创建一个引用循环：两个 `List` 值互相指向彼此

这里在变量 `a` 中创建了一个 `Rc<List>` 实例来存放初值为 `5, Nil` 的 `List` 值。接着在变量 `b` 中创建了存放包含值 `10` 和指向列表 `a` 的 `List` 的另一个 `Rc<List>` 实例。

下来修改 `a` 使其指向 `b` 而不是 `Nil`，这就创建了一个循环。为此需要使用 `tail` 方法获取 `a` 中 `RefCell<Rc<List>>` 的引用，并放入变量 `link` 中。接着使用 `RefCell<Rc<List>>` 的 `borrow_mut` 方法将其值从存放 `Nil` 的 `Rc<List>` 修改为 `b` 中的 `Rc<List>`。

如果保持最后的 `println!` 行注释并运行代码，会得到如下输出：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

可以看到将列表 `a` 修改为指向 `b` 之后， `a` 和 `b` 中的 `Rc<List>` 实例的引用计数都是 2。在 `main` 的结尾，Rust 丢弃 `b`，这会使 `b` `Rc<List>` 实例的引用计数从 2 减为 1。此时该 `Rc<List>` 实例并不会被回收，因为其引用计数是 1 而不是 0。接下来 Rust 会丢弃 `a` 将 `a` `Rc<List>` 实例的引用计数从 2 减为 1。这个实例也不能被回收，由于另一个 `Rc<List>` 实例依然引用它，所以其引用计数是 1。这些列表的内存将永远保持未被回收的状态。为了更直观地展示这一引用循环，我们创建了一个如图 15-4 所示的示意图：

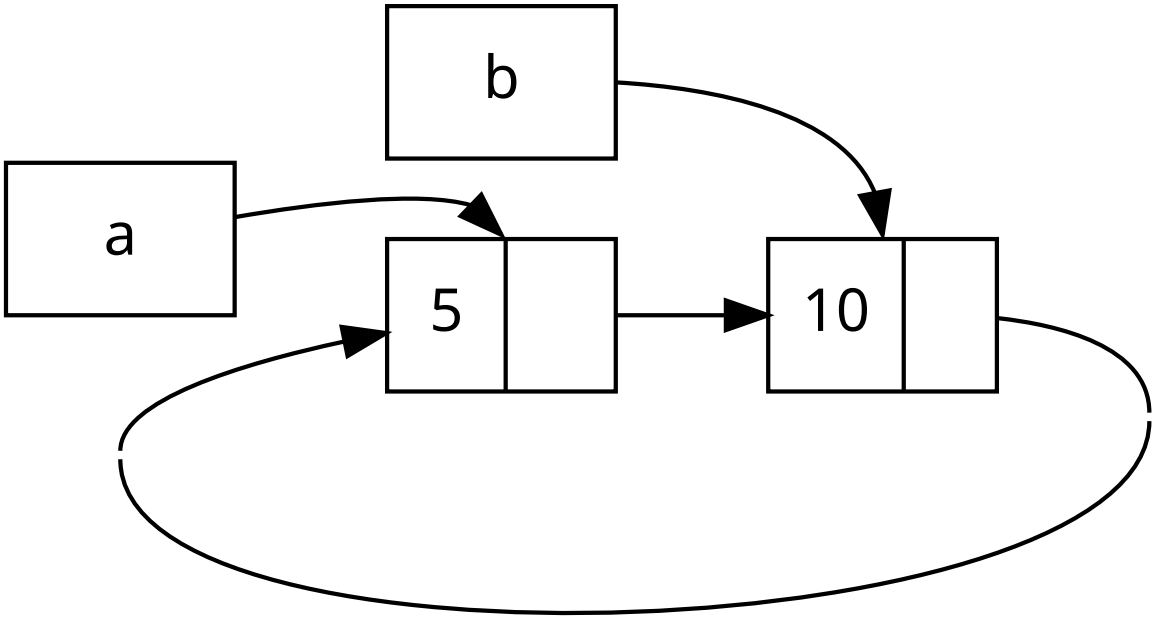


图 15-4: 列表 `a` 和 `b` 彼此互相指向形成引用循环

如果取消最后 `println!` 的注释并运行程序，Rust 会尝试打印出 `a` 指向 `b` 指向 `a` 这样的循环直到栈溢出。

相比真实世界的程序，这个例子中创建引用循环的结果并不可怕：创建了引用循环之后程序立刻就结束了。如果在更为复杂的程序中并在循环里分配了很多内存并占有很长时间，这个程序会使用多于它所需要的内存，并有可能压垮系统并造成没有内存可供使用。

创建引用循环并不容易，但也不是不可能。如果你有包含 `Rc<T>` 的 `RefCell<T>` 值或类似的嵌套结合了内部可变性和引用计数的类型，请务必小心确保你没有形成一个引用循环；你无法指望 Rust 帮你捕获它们。创建引用循环是一个程序上的逻辑 bug，你应该使用自动化测试、代码评审和其他软件开发最佳实践来使其最小化。

另一个解决方案是重新组织数据结构，使得一部分引用拥有所有权而另一部分没有。换句话说，循环将由一些拥有所有权的关系和一些无所有权的关系组成，只有所有权关系才能影响值是否可以被丢弃。在示例 15-25 中，我们总是希望 `Cons` 变体拥有其列表，所以重新组织数据结构是不可能的。让我们看看一个由父节点和子节点构成的图的例子，观察何时是使用无所有权的关系来避免引用循环的合适时机。

使用 `Weak<T>` 防止引用循环

到目前为止，我们已经展示了调用 `Rc::clone` 会增加 `Rc<T>` 实例的 `strong_count`，和只在其 `strong_count` 为 0 时 `Rc<T>` 实例才会被清理。你也可以通过调用 `Rc::downgrade` 并传递 `Rc<T>` 实例的引用来创建其值的**弱引用**（*weak reference*）。强引用代表如何共享 `Rc<T>` 实例的所有权；弱引用不表达所有权关系，当 `Rc<T>` 实例被清理时其计数没有影响。它们不会造成引用循环，因为任何涉及弱引用的循环会在其相关的值的强引用计数为 0 时被打断。

调用 `Rc::downgrade` 时会得到 `Weak<T>` 类型的智能指针。不同于将 `Rc<T>` 实例的 `strong_count` 加 1，调用 `Rc::downgrade` 会将 `weak_count` 加 1。`Rc<T>` 类型使用 `weak_count` 来记录其存在多少个 `Weak<T>` 引用，类似于 `strong_count`。其区别在于 `weak_count` 无需计数为 0 就能使 `Rc<T>` 实例被清理。

因为 `Weak<T>` 引用的值可能已经被丢弃了，为了使用 `Weak<T>` 所指向的值，我们必须确保其值仍然有效。为此可以调用 `Weak<T>` 实例的 `upgrade` 方法，这会返回 `Option<Rc<T>>`。如果 `Rc<T>` 值还未被丢弃，则结果是 `Some`；如果 `Rc<T>` 已被丢弃，则结果是 `None`。因为 `upgrade` 返回一个 `Option<Rc<T>>`，Rust 会确保处理 `Some` 和 `None` 的情况，所以它不会返回无效指针。

作为示例，我们不再使用只知道下一个元素的列表，而是创建一个既知道子节点又知道父节点的树结构。

创建树形数据结构：带有子节点的 `Node`

首先，我们将构建一个节点能够知道其子节点的树。创建一个用于存放其拥有所有权的 `i32` 值和对其子 `Node` 的引用：

文件名：src/main.rs

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

我们希望 `Node` 能够拥有其子节点，同时也希望能将所有权共享给变量，以便可以直接访问树中的每一个 `Node`，为此 `Vec<T>` 的项的类型被定义为 `Rc<Node>`。我们还希望能修改其他节点的子节点，所以 `children` 中 `Vec<Rc<Node>>` 被放进了 `RefCell<T>`。

接下来，使用此结构体定义来创建一个叫做 `leaf` 的带有值 `3` 且没有子节点的 `Node` 实例，和另一个带有值 5 并以 `leaf` 作为子节点的实例 `branch`，如示例 15-27 所示：

文件名：src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

示例 15-27：创建没有子节点的 `leaf` 节点和以 `leaf` 作为子节点的 `branch` 节点

这里克隆了 `leaf` 中的 `Rc<Node>` 并储存在 `branch` 中，这意味着 `leaf` 中的 `Node` 现在有两个所有者：`leaf` 和 `branch`。可以通过 `branch.children` 从 `branch` 中获得 `leaf`，不过无法从 `leaf` 得到 `branch`。`leaf` 没有到 `branch` 的引用且并不知道它们相互关联。我们希望 `leaf` 知道 `branch` 是其父节点。接下来我们会这么做。

增加从子到父的引用

为了使子节点知道其父节点，需要在 `Node` 结构体定义中增加一个 `parent` 字段。问题是 `parent` 的类型应该是什么。我们知道其不能包含 `Rc<T>`，因为这样 `leaf.parent` 将会指向 `branch` 而 `branch.children` 会包含 `leaf` 的指针，这会形成引用循环，会造成其 `strong_count` 永远也不会为 0。

现在换一种方式思考这个关系，父节点应该拥有其子节点：如果父节点被丢弃了，其子节点也应该被丢弃。然而子节点不应该拥有其父节点：如果丢弃子节点，其父节点应该依然存在。这正是弱引用的例子！

所以 `parent` 使用 `Weak<T>` 类型而不是 `Rc<T>`，具体来说是 `RefCell<Weak<Node>>`。现在 `Node` 结构体定义看起来像这样：

文件名：src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

这样，一个节点就能够引用其父节点，但不拥有其父节点。在示例 15-28 中，我们更新 `main` 来使用新定义以便 `leaf` 节点可以通过 `branch` 引用其父节点：

文件名：src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

示例 15-28：一个 `leaf` 节点，其拥有指向其父节点 `branch` 的弱引用

创建 `leaf` 节点类似于示例 15-27，除了 `parent` 字段有所不同：`leaf` 开始时没有父节点，所以我们新建了一个空的 `Weak<Node>` 引用实例。

此时，当尝试使用 `upgrade` 方法获取 `leaf` 的父节点引用时，会得到一个 `None` 值。如第一个 `println!` 输出所示：

```
leaf parent = None
```

当创建 `branch` 节点时，其也会新建一个 `Weak<Node>` 引用，因为 `branch` 并没有父节点。`leaf` 仍然作为 `branch` 的一个子节点。一旦在 `branch` 中有了 `Node` 实例，就可以修改 `leaf` 使其拥有指向父节点的 `Weak<Node>` 引用。这里使用了 `leaf` 中 `parent` 字段里的 `RefCell<Weak<Node>>` 的 `borrow_mut` 方法，接着使用了 `Rc::downgrade` 函数来从 `branch` 中的 `Rc<Node>` 值创建了一个指向 `branch` 的 `Weak<Node>` 引用。

当再次打印出 `leaf` 的父节点时，这一次将会得到存放了 `branch` 的 `Some` 值：现在 `leaf` 可以访问其父节点了！当打印出 `leaf` 时，我们也避免了如示例 15-26 中最终会导致栈溢出的循环：`Weak<Node>` 引用被打印为 `(Weak)`：

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } }] } })
```

没有无限的输出表明这段代码并没有造成引用循环。这一点也可以从观察 `Rc::strong_count` 和 `Rc::weak_count` 调用的结果看出。

可视化 `strong_count` 和 `weak_count` 的改变

让我们通过创建了一个新的内部作用域并将 `branch` 的创建放入其中，来观察 `Rc<Node>` 实例的 `strong_count` 和 `weak_count` 值的变化。这会展示当 `branch` 创建和离开作用域被丢弃时会发生什么。这些修改如示例 15-29 所示：

文件名：src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);
```

```
        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf strong = {}, weak = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

示例 15-29：在内部作用域创建 `branch` 并检查其强弱引用计数

一旦创建了 `leaf`，其 `Rc<Node>` 的强引用计数为 1，弱引用计数为 0。在内部作用域中创建了 `branch` 并与 `leaf` 相关联，此时 `branch` 中 `Rc<Node>` 的强引用计数为 1，弱引用计数为 1（因为 `leaf.parent` 通过 `Weak<Node>` 指向 `branch`）。这里 `leaf` 的强引用计数为 2，因为现在 `branch` 的 `branch.children` 中储存了 `leaf` 的 `Rc<Node>` 的拷贝，不过弱引用计数仍然为 0。

当内部作用域结束时，`branch` 离开作用域，`Rc<Node>` 的强引用计数减少为 0，所以其 `Node` 被丢弃。来自 `leaf.parent` 的弱引用计数 1 与 `Node` 是否被丢弃无关，所以并没有产生任何内存泄漏！

如果在内部作用域结束后尝试访问 `leaf` 的父节点，会再次得到 `None`。在程序的结尾，`leaf` 中 `Rc<Node>` 的强引用计数为 1，弱引用计数为 0，因为现在 `leaf` 又是 `Rc<Node>` 唯一的引用了。

所有这些管理计数和值的逻辑都内建于 `Rc<T>` 和 `Weak<T>` 以及它们的 `Drop` trait 实现中。通过在 `Node` 定义中指定从子节点到父节点的关系为一个 `Weak<T>` 引用，就能够拥有父节点和子节点之间的双向引用而不会造成引用循环和内存泄漏。

总结

这一章涵盖了如何使用智能指针来做出不同于 Rust 常规引用默认所提供的保证与取舍。`Box<T>` 有一个已知的大小并指向分配在堆上的数据。`Rc<T>` 记录了堆上数据的引用计数从而允许多个所有者。`RefCell<T>` 类型及其内部可变性允许我们在保持类型不可变的前提下更改其内部值；它也在运行时而非编译时执行借用规则检查。

我们还讨论了 trait `Deref` 和 `Drop`，它们实现了智能指针的许多功能。同时探索了会造成内存泄漏的引用循环，以及如何使用 `Weak<T>` 来避免它们。

如果本章内容引起了你的兴趣并希望现在就实现你自己的智能指针的话，请阅读 “The Rustonomicon” 来获取更多有用的信息。

接下来，让我们谈谈 Rust 的并发。届时甚至还会学习到一些新的对并发有帮助的智能指针。

无畏并发

安全且高效地处理并发编程是 Rust 的另一个主要目标。**并发编程**（*Concurrent programming*），代表程序的不同部分相互独立地执行，而**并行编程**（*parallel programming*）代表程序不同部分同时执行，这两个概念随着计算机越来越多的利用多处理器的优势而显得愈发重要。由于历史原因，在此类上下文中编程一直是困难且容易出错的：Rust 希望能改变这一现状。

起初，Rust 团队认为确保内存安全和防止并发问题是两个分别需要不同方法应对的挑战。随着时间的推移，团队发现所有权和类型系统是一系列解决内存安全**和**并发问题的强有力的工具！通过利用所有权和类型检查，在 Rust 中很多并发错误都是**编译时**错误，而非运行时错误。因此，相比花费大量时间尝试重现运行时并发 bug 出现的特定情况，不正确的代码会直接编译失败并提供解释问题的错误信息。因此，你可以在开发时修复代码，而不是在部署到生产环境后修复代码。我们给 Rust 的这一部分起了一个绰号**无畏并发**（*fearless concurrency*）。无畏并发令你的代码免于出现诡异的 bug 并可以轻松重构且无需担心会引入新的 bug。

> 注意：出于简洁的考虑，我们将很多问题归类为**并发**，而不是更准确的区分**并发和/或并行**。对于本章，当我们谈到**并发**时，请自行脑内替换为 **并发和/或并行**。在下一章中当区分二者更为重要时，我们会使用更准确的表述。

很多语言所提供的处理并发问题的解决方法都非常固有。例如，Erlang 有着优雅的消息传递（message-passing）并发功能，但只有模糊不清的在线程间共享状态的方法。对于高级语言来说，只实现可能解决方案的子集是一个合理的策略，因为高级语言所许诺的价值来源于牺牲一些控制来换取抽象。然而对于底层语言则期望提供在任何给定的情况下有着最高的性能且对硬件有更少的抽象。因此，Rust 提供了多种工具，以符合实际情况和需求的方式来为问题建模。

如下是本章将要涉及到的内容：

- 如何创建线程来同时运行多段代码。
- **消息传递**（*Message passing*）并发，其中信道（channel）被用来在线程间传递消息。
- **共享状态**（*Shared state*）并发，其中多个线程可以访问同一片数据。
- `Sync` 和 `Send` trait，将 Rust 的并发保证扩展到用户定义的以及标准库提供的类型中。

使用线程同时运行代码

在大部分现代操作系统中，已执行程序的代码在一个**进程**（*process*）中运行，操作系统则会负责管理多个进程。在程序内部，也可以拥有多个同时运行的独立部分。这些运行这些独立部分的功能被称为**线程**（*threads*）。例如，web 服务端可以有多个线程以便可以同时响应多个请求。

将程序中的计算拆分进多个线程可以改善性能，因为程序可以同时进行多个任务，不过这也会增加复杂性。因为线程是同时运行的，所以无法预先保证不同线程中的代码的执行顺序。这会导致诸如此类的问题：

- 竞态条件（Race conditions），多个线程以不一致的顺序访问数据或资源
- 死锁（Deadlocks），两个线程相互等待对方，这会阻止两者继续运行
- 只会发生在特定情况且难以稳定重现和修复的 bug

Rust 尝试减轻使用线程的负面影响。不过在多线程上下文中编程仍需格外小心，同时其所要求的代码结构也不同于运行于单线程的程序。

编程语言实现线程的方式各不相同，许多操作系统都提供了供语言调用以创建新线程的 API。Rust 标准库使用 *1:1* 模型的线程实现，这代表程序的每一个语言级线程使用一个系统线程。有一些 crate 实现了其他有着不同于 1:1 模型取舍的线程模型。（Rust 的 async 系统，我们将在下一章看到，也提供了另一种并发方式。）

使用 `spawn` 创建新线程

为了创建一个新线程，需要调用 `thread::spawn` 函数并传递一个闭包（第十三章学习了闭包），并在其中包含希望在新线程运行的代码。示例 16-1 中的例子在主线程打印了一些文本而另一些文本则由新线程打印：

文件名：src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

示例 16-1: 创建一个打印某些内容的新线程同时主线程打印其它内容

注意当 Rust 程序的主线程结束时，所有新线程也会结束，而不管其是否执行完毕。这个程序的输出可能每次都略有不同，不过它大体上看起来像这样：

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

`thread::sleep` 调用强制线程停止执行一小段时间，这会允许其他不同的线程运行。这些线程可能会轮流运行，不过并不保证如此：这依赖操作系统如何调度线程。在这里，主线程首先打印，即便新创建线程的打印语句位于程序的开头，甚至即便我们告诉新建的线程打印直到 `i` 等于 9，它在主线程结束之前也只打印到了 5。

如果运行代码只看到了主线程的输出，或没有出现重叠打印的现象，尝试增大区间 (变量 `i` 的范围) 来增加操作系统切换线程的机会。

使用 `join` 等待所有线程结束

由于主线程结束，示例 16-1 中的代码大部分时候不光会提早结束新建线程，因为无法保证线程运行的顺序，我们甚至不能实际保证新建线程会被执行！

可以通过将 `thread::spawn` 的返回值储存在变量中来修复新建线程部分没有执行或者完全没有执行的问题。`thread::spawn` 的返回值类型是 `JoinHandle<T>`。`JoinHandle<T>` 是一个拥有所有权的值，当对其调用 `join` 方法时，它会等待其线程结束。示例 16-2 展示了如何使用示例 16-1 中创建的线程的 `JoinHandle<T>` 并调用 `join` 来确保新建线程在 `main` 退出前结束运行。

文件名：src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

示例 16-2: 从 `thread::spawn` 保存一个 `JoinHandle<T>` 以确保该线程能够运行至结束

通过调用句柄的 `join` 会阻塞当前线程直到句柄所代表的线程结束。**阻塞**（*Blocking*）线程意味着阻止该线程执行工作或退出。因为我们将 `join` 调用放在了主线程的 `for` 循环之后，运行示例 16-2 应该会产生类似这样的输出：

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

这两个线程仍然会交替执行，不过主线程会由于 `handle.join()` 调用而不会结束直到新建线程执行完毕。

不过让我们看看将 `handle.join()` 移动到 `main` 中 `for` 循环之前会发生什么，如下：

文件名：src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

主线程会等待直到新建线程执行完毕之后才开始执行 `for` 循环，所以输出将不会交替出现，如下所示：

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
```

```
hi number 3 from the main thread!
hi number 4 from the main thread!
```

诸如将 `join` 放置于何处这样的小细节，会影响线程是否同时运行。

将 `move` 闭包与线程一同使用

`move` 关键字经常用于传递给 `thread::spawn` 的闭包，因为闭包会获取从环境中取得的值的所有权，因此会将这些值的所有权从一个线程传送到另一个线程。在第十三章"使用闭包捕获环境"部分讨论了闭包上下文中的 `move`。现在我们会更专注于 `move` 和 `thread::spawn` 之间的交互。

在第十三章中，我们讲到可以在参数列表前使用 `move` 关键字强制闭包获取其使用的环境值的所有权。这个技巧在创建新线程将值的所有权从一个线程移动到另一个线程时最为实用。

注意示例 16-1 中传递给 `thread::spawn` 的闭包并没有任何参数：并没有在新建线程代码中使用任何主线程的数据。为了在新建线程中使用来自于主线程的数据，需要新建线程的闭包获取它需要的值。示例 16-3 展示了一个尝试在主线程中创建一个 vector 并用于新建线程的例子，不过这么写还不能工作，如下所示：

文件名：src/main.rs



```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {v:?}");
    });

    handle.join().unwrap();
}
```

示例 16-3: 尝试在另一个线程使用主线程创建的 vector

闭包使用了 `v`，所以闭包会捕获 `v` 并使其成为闭包环境的一部分。因为 `thread::spawn` 在一个新线程中运行这个闭包，所以可以在新线程中访问 `v`。然而当编译这个例子时，会得到如下错误：

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows `v`, which
is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {v:?}");
  |                                     - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
```

```
  |
6 |       let handle = thread::spawn(|| {
  |  ____________________^
7 | |         println!("Here's a vector: {v:?}");
8 | |     });
  | |______^
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` (bin "threads") due to 1 previous error
```

Rust 会**推断**如何捕获 `v`，因为 `println!` 只需要 `v` 的引用，闭包尝试借用 `v`。然而这有一个问题：Rust 不知道这个新建线程会执行多久，所以无法知晓对 `v` 的引用是否一直有效。

示例 16-4 展示了一个 `v` 的引用很有可能不再有效的场景：

文件名：src/main.rs



```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {v:?}");
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

示例 16-4: 一个具有闭包的线程，尝试使用一个在主线程中被回收的引用 `v`

如果 Rust 允许这段代码运行，则新建线程可能会立刻被转移到后台并完全没有机会运行。新建线程内部有一个 `v` 的引用，不过主线程立刻就使用第十五章讨论的 `drop` 丢弃了 `v`。接着当新建线程开始执行，`v` 已不再有效，所以其引用也是无效的。噢，这太糟了！

为了修复示例 16-3 的编译错误，我们可以听取错误信息的建议：

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++
```

通过在闭包之前增加 `move` 关键字，我们强制闭包获取其使用的值的所有权，而不是任由 Rust 推断它应该借用值。示例 16-5 中展示的对示例 16-3 代码的修改，就能按预期编译并运行：

文件名：src/main.rs

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {v:?}");
    });

    handle.join().unwrap();
}
```

示例 16-5: 使用 `move` 关键字强制获取它使用的值的所有权

我们可能希望尝试同样的方法来修复示例 16-4 中的代码，其主线程使用 `move` 闭包调用了 `drop`。然而这个修复行不通，因为示例 16-4 所尝试的操作由于一个不同的原因而不被允许。如果为闭包增加 `move`，将会把 `v` 移动进闭包的环境中，如此将不能在主线程中对其调用 `drop` 了。我们会得到如下不同的编译错误：

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
4  |     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `Vec<i32>`, which does not
implement the `Copy` trait
5  |
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved into closure here
7  |         println!("Here's a vector: {v:?}");
   |                                     - variable moved due to use in closure
...
10 |     drop(v); // oh no!
   |          ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads` (bin "threads") due to 1 previous error
```

Rust 的所有权规则又一次帮助了我们！示例 16-3 中的错误是因为 Rust 是保守的并只会为线程借用 `v`，这意味着主线程理论上可能使新建线程的引用无效。通过告诉 Rust 将 `v` 的所有权移动到新建线程，我们向 Rust 保证主线程不会再使用 `v`。如果对示例 16-4 也做出如此修改，那么当在主线程中使用 `v` 时就会违反所有权规则。`move` 关键字覆盖了 Rust 默认保守的借用，但它不允许我们违反所有权规则。

现在我们已经了解了线程的概念以及线程 API 提供的方法，下面让我们看看在什么情况下可以使用线程。

使用消息传递在线程间传送数据

一个日益流行的确保安全并发的方式是**消息传递**（*message passing*），这里线程或 actor 通过发送包含数据的消息来相互沟通。这个思想来源于 Go 编程语言文档 中的口号：“不要通过共享内存来通讯；而要通过通讯来共享内存。”（“Do not communicate by sharing memory; instead, share memory by communicating.”）

为了实现消息传递并发，Rust 标准库提供了一个**信道**（*channel*）实现。信道是一个通用编程概念，表示数据从一个线程发送到另一个线程。

你可以将编程中的信道想象为一个水流的渠道，比如河流或小溪。如果你将诸如橡皮鸭之类的东西放入其中，它们会顺流而下到达下游。

信道有两个组成部分：一个发送端（transmitter）和一个接收端（receiver）。发送端位于上游位置，在这里可以将橡皮鸭放入河中，接收端则位于下游，橡皮鸭最终会漂流至此。代码中的一部分调用发送端的方法以及希望发送的数据，另一部分则检查接收端收到的消息。当发送端或接收端任一被丢弃时可以认为信道被**关闭**（*closed*）了。

这里，我们将开发一个程序，它会在一个线程生成值向信道发送，而在另一个线程会接收值并打印出来。这里会通过信道在线程间发送简单值来演示这个功能。一旦你熟悉了这项技术，你就可以将信道用于任何相互通信的任何线程，例如一个聊天系统，或利用很多线程进行分布式计算并将部分计算结果发送给一个线程进行聚合。

首先，在示例 16-6 中，创建了一个信道但没有做任何事。注意这还不能编译，因为 Rust 不知道我们想要在信道中发送什么类型：

文件名：src/main.rs



```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

示例 16-6: 创建一个信道，并将其两端赋值给 `tx` 和 `rx`

这里使用 `mpsc::channel` 函数创建一个新的信道；`mpsc` 是 **多生产者，单消费者**（*multiple producer, single consumer*）的缩写。简而言之，Rust 标准库实现信道的方式意味着一个信道可以有多个产生值的 **发送端**（*sending*），但只能有一个消费这些值的**接收端**（*receiving*）。想象一下多条小河小溪最终汇聚成大河：所有通过这些小河发出的东西最后都会来到下游的大河。目前我们以单个生产者开始，但是当示例可以工作后会增加多个生产者。

`mpsc::channel` 函数返回一个元组：第一个元素是发送侧 – 发送端，而第二个元素是接收侧 – 接收端。由于历史原因，`tx` 和 `rx` 通常作为**发送端**（*transmitter*）和 **接收端**（*receiver*）的传统缩写，所以这就是我们将用来绑定这两端变量的名字。这里使用了一个 `let` 语句和模式来解构了此元组；第十九章会讨论 `let` 语句中的模式和解构。现在只需知道使用 `let` 语句是一个方便提取 `mpsc::channel` 返回的元组中一部分的手段。

让我们将发送端移动到一个新建线程中并发送一个字符串，这样新建线程就可以和主线程通讯了，如示例 16-7 所示。这类似于在河的上游扔下一只橡皮鸭或从一个线程向另一个线程发送聊天信息：

文件名：src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

示例 16-7: 将 `tx` 移动到一个新建的线程中并发送 `"hi"`

这里再次使用 `thread::spawn` 来创建一个新线程并使用 `move` 将 `tx` 移动到闭包中这样新建线程就拥有 `tx` 了。新建线程需要拥有信道的发送端以便能向信道发送消息。信道的发送端有一个 `send` 方法用来获取需要放入信道的值。`send` 方法返回一个 `Result<T, E>` 类型，所以如果接收端已经被丢弃了，将没有发送值的目标，所以发送操作会返回错误。在这个例子中，出错的时候调用 `unwrap` 产生 panic。不过在一个真实应用中，需要合理地处理它：回到第九章复习正确处理错误的策略。

在示例 16-8 中，我们在主线程中从信道的接收端获取值。这类似于在河的下游捞起橡皮鸭或接收聊天信息：

文件名：src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```

示例 16-8: 在主线程中接收并打印内容 `"hi"`

信道的接收端有两个有用的方法：`recv` 和 `try_recv`。这里，我们使用了 `recv`，它是 *receive* 的缩写，这会阻塞主线程执行直到从信道中接收一个值。一旦发送了一个值，`recv` 会在一个 `Result<T, E>` 中返回它。当信道发送端关闭，`recv` 会返回一个错误表明不会再有新的值到来了。

`try_recv` 不会阻塞，相反它立刻返回一个 `Result<T, E>`：`Ok` 值包含可用的信息，而 `Err` 值代表此时没有任何消息。如果线程在等待消息过程中还有其他工作时使用 `try_recv` 很有用：可以编写一个循环来频繁调用 `try_recv`，在有可用消息时进行处理，其余时候则处理一会其他工作直到再次检查。

出于简单的考虑，这个例子使用了 `recv`；主线程中除了等待消息之外没有任何其他工作，所以阻塞主线程是合适的。

运行示例 16-8 中的代码时，我们将会看到主线程打印出这个值：

```
Got: hi
```

完美！

信道与所有权转移

所有权规则在消息传递中扮演了重要角色，其有助于我们编写安全的并发代码。防止并发编程中的错误是在 Rust 程序中考虑所有权的一大优势。现在让我们做一个实验来看看信道与所有权如何一同协作以避免产生问题：我们将尝试在新建线程中的信道中发送完 `val` 值**之后**再使用它。尝试编译示例 16-9 中的代码并看看为何这是不允许的：

文件名：src/main.rs



```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {val}");
    });

    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```

示例 16-9: 在我们已经发送到信道中后，尝试使用 `val` 引用

这里尝试在通过 `tx.send` 发送 `val` 到信道中之后将其打印出来。允许这么做是一个坏主意：一旦将值发送到另一个线程后，那个线程可能会在我们再次使用它之前就将其修改或者丢弃。其他线程对值可能的修改会由于不一致或不存在的数据而导致错误或意外的结果。然而，尝试编译示例 16-9 的代码时，Rust 会给出一个错误：

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:26
   |
8  |         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`, which does not
implement the `Copy` trait
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {val}");
   |                          ^^^^^ value borrowed here after move
```

```
   |
   = note: this error originates in the macro `$crate::format_args_nl` which comes
from the expansion of the macro `println` (in Nightly builds, run with -Z macro-
backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` (bin "message-passing") due to 1
previous error
```

我们的并发错误会造成一个编译时错误。`send` 函数获取其参数的所有权并移动这个值归接收端所有。这可以防止在发送后意外地再次使用这个值；所有权系统检查一切是否合乎规则。

发送多个值并观察接收端的等待

示例 16-8 中的代码可以编译和运行，不过它并没有明确的告诉我们两个独立的线程通过信道相互通讯。示例 16-10 则有一些改进会证明示例 16-8 中的代码是并发执行的：新建线程现在会发送多个消息并在每个消息之间暂停一秒钟。

文件名：src/main.rs

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {received}");
    }
}
```

示例 16-10: 发送多个消息，并在每次发送后暂停一段时间

这一次，在新建线程中有一个字符串 vector 希望发送到主线程。我们遍历它们，单独发送每一个字符串并通过一个 `Duration` 值调用 `thread::sleep` 函数来暂停一秒。

在主线程中，不再显式调用 `recv` 函数：而是将 `rx` 当作一个迭代器。对于每一个接收到的值，我们将其打印出来。当信道被关闭时，迭代器也将结束。

当运行示例 16-10 中的代码时，将看到如下输出，每一行都会暂停一秒：

```
Got: hi
Got: from
Got: the
Got: thread
```

因为主线程中的 `for` 循环里并没有任何暂停或等待的代码，所以可以说主线程是在等待从新建线程中接收值。

通过克隆发送端来创建多个生产者

之前我们提到了 `mpsc` 是 *multiple producer, single consumer* 的缩写。可以运用 `mpsc` 来扩展示例 16-10 中的代码来创建多个向同一接收端发送值的线程。这可以通过克隆发送端来做到，如示例 16-11 所示：

文件名：src/main.rs

```
    // --snip--

    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {received}");
    }

    // --snip--
```

示例 16-11: 从多个生产者发送多个消息

这一次，在创建新线程之前，我们对发送端调用了 `clone` 方法。这会给我们一个可以传递给第一个新建线程的发送端句柄。我们会将原始的信道发送端传递给第二个新建线程。这样就会有两个线程，每个线程将向信道的接收端发送不同的消息。

运行代码时，输出应该看起来类似如下：

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

虽然你可能会看到这些值以不同的顺序出现；这依赖于你的系统。这也就是并发既有趣又困难的原因。如果通过 `thread::sleep` 做实验，在不同的线程中提供不同的值，就会发现它们的运行更加不确定，且每次都会产生不同的输出。

现在我们见识过了信道如何工作，再看看另一种不同的并发方式吧。

共享状态的并发

消息传递是一个很好的处理并发的方式，但并不是唯一一个。另一种方式是让多个线程访问相同的共享数据。再考虑一下 Go 语言文档中的这句口号：“不要通过共享内存来通讯”。（“do not communicate by sharing memory.”）

通过共享内存进行通信，会是什么样的代码？此外，为什么喜欢消息传递的人会警告说谨慎使用内存共享？

在某种程度上，任何编程语言中的信道都类似于单所有权，因为一旦将一个值传送到信道中，将无法再使用这个值。共享内存类似于多所有权：多个线程可以同时访问相同的内存位置。在 15 章中，我们介绍了智能指针可以实现多所有权，然而这会增加额外的复杂性，因为需要管理多个所有者。Rust 的类型系统和所有权规则在正确管理这些问题上提供了极大的帮助：举个例子，让我们来看看互斥器（mutexes），较为常见的共享内存并发原语之一。

使用互斥器实现同一时刻只允许一个线程访问数据

互斥器（*mutex*）是互相排斥（*mutual exclusion*）的缩写，因为在同一时刻，它只允许一个线程访问数据。为了访问互斥器中的数据，线程首先需要通过获取互斥器的**锁**（*lock*）来表明其希望访问数据。锁是一个数据结构，作为互斥器的一部分，它记录谁有数据的专属访问权。因此我们讲，互斥器通过锁系统**保护**（*guarding*）其数据。

互斥器以难以使用著称（译注：原文指互斥器在其他编程语言中难以使用），因为你必须记住两个规则：

1. 在使用数据之前，必须获取锁。
2. 使用完被互斥器所保护的数据之后，必须解锁数据，这样其他线程才能够获取锁。

作为一个现实中互斥器的例子，想象一下在某个会议的一次小组座谈会中，只有一个麦克风。如果一位成员要发言，他必须请求或表示希望使用麦克风。得到了麦克风后，他可以畅所欲言，讲完后再将麦克风交给下一位希望讲话的成员。如果一位成员结束发言后忘记将麦克风交还，其他人将无法发言。如果对共享麦克风的管理出现了问题，座谈会将无法正常进行！

正确的管理互斥器异常复杂，这也是许多人之所以热衷于信道的原因。然而，在 Rust 中，得益于类型系统和所有权，你不可能在锁和解锁上出错。

`Mutex<T>`的 API

我们先从在单线程环境中使用互斥器开始，作为展示其用法的一个例子，如示例 16-12 所示：

文件名：src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {m:?}");
}
```

示例 16-12: 出于简单的考虑，在一个单线程上下文中探索 `Mutex<T>` 的 API

像很多类型一样，我们使用关联函数 `new` 来创建一个 `Mutex<T>`。使用 `lock` 方法来获取锁，从而可以访问互斥器中的数据。这个调用会阻塞当前线程，直到我们拥有锁为止。

如果另一个线程拥有锁，并且那个线程 panic 了，则 `lock` 调用会失败。在这种情况下，没人能够再获取锁，所以我们调用 `unwrap`，使当前线程 panic。

一旦获取了锁，就可以将返回值（命名为 `num`）视为一个其内部数据的可变引用了。类型系统确保了我们在使用 `m` 中的值之前获取锁。`m` 的类型是 `Mutex<i32>` 而不是 `i32`，所以**必须**调用 `lock` 才能使用这个 `i32` 值。我们不能忘记这么做，因为如果没有获取锁，类型系统就不允许访问内部的 `i32` 值。

正如你所猜想的，`Mutex<T>` 是一个智能指针。更准确的说，`lock` 调用会**返回**一个叫做 `MutexGuard` 的智能指针。`MutexGuard` 智能指针实现了 `Deref` 来指向其内部数据；它也实现了 `Drop`，当 `MutexGuard` 离开作用域时，自动释放锁（发生在示例 16-12 内部作用域的结尾）。这样一来，就不会有忘记释放锁从而导致互斥器阻塞无法被其他线程使用的潜在风险，因为锁的释放是自动发生的。

释放锁之后，我们可以打印出互斥器内部的 `i32` 值，并发现我们刚刚已经将其值改为 6。

在多个线程间共享 `Mutex<T>`

现在让我们尝试使用 `Mutex<T>` 在多个线程间共享同一个值。我们将启动 10 个线程，并在各个线程中对同一个计数器值加 1，这样计数器将从 0 累加到 10。示例 16-13 中的例子会出现编译错误，而我们将通过这些错误来学习如何使用 `Mutex<T>`，以及 Rust 又是如何帮助我们正确使用它的。

文件名：src/main.rs



```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

示例 16-13: 程序启动了 10 个线程，每个线程都通过 `Mutex<T>` 来增加计数器的值

这里创建了一个 `counter` 变量来存放内含 `i32` 的 `Mutex<T>`，类似示例 16-12 那样。接下来我们遍历一个整数 range 来创建了 10 个线程。我们使用了 `thread::spawn`，并为所有线程传入了相同的闭包：它们每一个都将计数器移动进线程，调用 `lock` 方法来获取 `Mutex<T>` 上的锁，接着将互斥器中的值加一。当一个线程结束执行，`num` 会离开闭包作用域并释放锁，这样另一个线程就可以获取它了。

在主线程中，我们像示例 16-2 那样收集了所有的 `JoinHandle`，并调用它们的 `join` 方法来等待所有线程结束。然后，主线程会获取锁，并打印出程序的结果。

之前提示过，这个例子不能编译。现在让我们看看为什么！

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: borrow of moved value: `counter`
  --> src/main.rs:21:29
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which
does not implement the `Copy` trait
...
8  |     for _ in 0..10 {
   |     -------------- inside of this loop
9  |         let handle = thread::spawn(move || {
   |                                    ------- value moved into closure here, in
previous iteration of loop
...
21 |     println!("Result: {}", *counter.lock().unwrap());
   |                            ^^^^^^^ value borrowed here after move
   |
help: consider moving the expression out of the loop so it is only moved once
   |
8  ~     let mut value = counter.lock();
9  ~     for _ in 0..10 {
10 |         let handle = thread::spawn(move || {
11 ~             let mut num = value.unwrap();
   |

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` (bin "shared-state") due to 1 previous
error
```

错误信息表明 `counter` 值在上一次循环中被移动了。所以 Rust 告诉我们，不能将 `counter` 锁的所有权移动到多个线程中。让我们通过一个第 15 章讨论过的多所有权手段，来修复这个编译错误。

多线程和多所有权

在第 15 章中，我们用智能指针 `Rc<T>` 来创建引用计数，使得一个值有了多个所有者。让我们做同样的事，看看会发生什么。将示例 16-14 中的 `Mutex<T>` 封装进 `Rc<T>` 中，并在将所有权移入线程之前克隆（clone）`Rc<T>`。

文件名：src/main.rs



```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

示例 16-14: 尝试使用 `Rc<T>` 来允许多个线程拥有 `Mutex<T>`

再一次编译并...出现了不同的错误！编译器真是教会了我们很多！

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
  --> src/main.rs:11:36
   |
11 |           let handle = thread::spawn(move || {
   |                        ------------- ^------
   |                        |             |
   |  ______________________|_____________within this `{closure@src/main.rs:11:36:
11:43}`
   | |                      |
   | |                      required by a bound introduced by this call
12 | |             let mut num = counter.lock().unwrap();
13 | |
14 | |             *num += 1;
15 | |         });
   | |_________^ `Rc<Mutex<i32>>` cannot be sent between threads safely
   |
   = help: within `{closure@src/main.rs:11:36: 11:43}`, the trait `Send` is not
implemented for `Rc<Mutex<i32>>`
note: required because it's used within this closure
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^
note: required by a bound in `spawn`
  --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/std/src/thread/
```

```
mod.rs:728:1

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` (bin "shared-state") due to 1 previous
error
```

哇哦，错误信息太长不看！下面是重点要关注的内容：
`` `Rc<Mutex<i32>>` cannot be sent between threads safely ``。编译器也指出了原因：
`` the trait `Send` is not implemented for `Rc<Mutex<i32>>` ``。下一节我们会讲到 `Send`：这是一个确保所使用的类型可以用于并发环境的 trait。

不幸的是，`Rc<T>` 并不能安全的在线程间共享。当 `Rc<T>` 管理引用计数时，它必须在每一个 `clone` 调用时增加计数，并在每一个克隆体被丢弃时减少计数。`Rc<T>` 并没有使用任何并发原语，无法确保改变计数的操作不会被其他线程打断。这可能使计数出错，并导致诡异的 bug，比如可能会造成内存泄漏，或在使用结束之前就丢弃一个值。我们所需要的是一个与 `Rc<T>` 完全一致，又以线程安全的方式改变引用计数的类型。

原子引用计数 `Arc<T>`

所幸 `Arc<T>` 正是这么一个类似 `Rc<T>` 并可以安全地用于并发环境的类型。字母 *a* 代表 **原子性**（*atomic*），所以这是一个**原子引用计数**（*atomically reference counted*）类型。**原子类型** (Atomics) 是另一类这里还未涉及到的并发原语：请查看标准库中 `std::sync::atomic` 的文档来获取更多细节。目前我们只需要知道：原子类型就像基本类型一样，可以安全地在线程间共享。

你可能会好奇，为什么不是所有的基本类型都是原子性的？为什么标准库中的类型没有全部默认使用 `Arc<T>` 实现？原因在于，线程安全会造成性能损失，我们希望只在必要时才为此买单。如果只是在单线程中对值进行操作，不必强制原子性所提供的保证可以使代码运行得更快。

回到之前的例子：`Arc<T>` 和 `Rc<T>` 有着相同的 API，所以我们只需修改程序中的 `use` 行、`new` 调用和 `clone` 调用。示例 16-15 中的代码最终可以编译并运行。

文件名：src/main.rs

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
```

```
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

示例 16-15: 使用 `Arc<T>` 包装一个 `Mutex<T>` 能够实现在多线程之间共享所有权

这段代码会打印出如下：

```
Result: 10
```

成功了！我们从 0 数到了 10，这好像没啥大不了的，不过一路上我们确实学习了很多关于 `Mutex<T>` 和线程安全的知识！这个例子中构建的结构可以用于比增加计数更为复杂的操作。使用这个策略，我们可将计算任务分成独立的部分，并分散到多个线程中，接着使用 `Mutex<T>` 使用各自的运算结果来更新最终的结果。

注意，对于简单的数值运算，标准库中 `std::sync::atomic` 模块 提供了比 `Mutex<T>` 更简单的类型。针对基本类型，这些类型提供了安全、并发、原子的操作。在上面的例子中，为了专注于讲明白 `Mutex<T>` 的用法，我们才选择在基本类型上使用 `Mutex<T>`。（译注：对于上面例子中出现的 `i32` 加法操作，更好的做法是使用 `AtomicI32` 类型来完成。具体参考文档。）

`RefCell<T>/Rc<T>` 与 `Mutex<T>/Arc<T>` 的相似性

你可能注意到了，尽管 `counter` 是不可变的，我们仍然可以获取其内部值的可变引用；这意味着 `Mutex<T>` 提供了内部可变性，就像 `Cell` 系列类型那样。使用 `RefCell<T>` 可以改变 `Rc<T>` 中内容（在 15 章中讲到过），同样地，使用 `Mutex<T>` 我们也可以改变 `Arc<T>` 中的内容。

另一个值得注意的细节是，Rust 不能完全避免使用 `Mutex<T>` 所带来的逻辑错误。回忆一下，第 15 章中讲过，使用 `Rc<T>` 就有造成引用循环的风险：两个 `Rc<T>` 值相互引用，造成内存泄漏。同理，`Mutex<T>` 也有造成**死锁**（*deadlock*）的风险：当某个操作需要锁住两个资源，而两个线程分别持有两个资源的其中一个锁时，它们会永远相互等待。如果你对这个话题感兴趣，尝试编写一个带有死锁的 Rust 程序，接着研究其它语言中使用互斥器的死锁规避策略，并尝试在 Rust 中实现它们。标准库中 `Mutex<T>` 和 `MutexGuard` 的 API 文档会提供有用的信息。

作为本章的最后，我们将讨论 `Send` 和 `Sync` trait，以及如何将它们用于自定义类型。

使用 Send 和 Sync trait 的可扩展并发

Rust 的并发模型中一个有趣的方面是：我们之前讨论的几乎所有内容，都属于标准库，而不是语言本身的内容。处理并发的方案并不受标准库或语言所限：我们可以编写自己的或使用他人编写的并发特性。

然而，有一些关键的并发概念是内嵌于语言本身而非标准库的，其中就包括 `std::marker` 的 `Send` 和 `Sync` trait。

通过 Send 允许在线程间转移所有权

`Send` 标记 trait 表明实现了 `Send` 的类型值的所有权可以在线程间传送。几乎所有的 Rust 类型都是`Send` 的，不过有一些例外，包括 `Rc<T>`：这是不能实现 `Send` 的，因为如果克隆了 `Rc<T>` 的值并尝试将克隆的所有权转移到另一个线程，这两个线程都可能同时更新引用计数。为此，`Rc<T>` 被实现为用于单线程场景，这时不需要为拥有线程安全的引用计数而付出性能代价。

因此，Rust 类型系统和 trait bound 确保永远也不会意外的将不安全的 `Rc<T>` 在线程间发送。当尝试在示例 16-14 中这么做的时候，会得到错误 `the trait Send is not implemented for Rc<Mutex<i32>>`。而使用实现了 `Send` 的 `Arc<T>` 时，代码就能成功编译。

任何完全由 `Send` 的类型组成的类型也会自动被标记为 `Send`。几乎所有基本类型都是 `Send` 的，除了第二十章将会讨论的裸指针（raw pointer）。

Sync 允许多线程访问

`Sync` 标记 trait 表明一个实现了 `Sync` 的类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 `T`，如果 `&T`（`T` 的不可变引用）实现了 `Send` 的话 `T` 就实现了 `Sync`，这意味着其引用就可以安全的发送到另一个线程。类似于 `Send` 的情况，基本类型都实现了 `Sync`，完全由实现了 `Sync` 的类型组成的类型也实现了 `Sync`。

智能指针 `Rc<T>` 也没有实现 `Sync`，出于其没有实现 `Send` 相同的原因。`RefCell<T>`（第十五章讨论过）和 `Cell<T>` 系列类型没有实现 `Sync`。`RefCell<T>` 在运行时所进行的借用检查也不是线程安全的。`Mutex<T>` 实现了 `Sync`，正如 “在多个线程间共享 `Mutex<T>`” 部分所讲的它可以被用来在多线程中共享访问。

手动实现 Send 和 Sync 是不安全的

通常并不需要手动实现 `Send` 和 `Sync` trait，因为完全由实现了 `Send` 和 `Sync` 的类型组成的类型，自动实现了 `Send` 和 `Sync`。因为它们是标记 trait，甚至都不需要实现任何方法。它们只是用来加强并发相关的不可变性的。

手动实现这些标记 trait 涉及到编写不安全的 Rust 代码，第二十章将会讲述具体的方法；当前重要的是，在创建新的由不是 `Send` 和 `Sync` 的部分构成的并发类型时需要多加小心，以确保维持其安全保证。“The Rustonomicon” 中有更多关于这些保证以及如何维持它们的信息。

总结

这不会是本书最后一个出现并发的章节：下一章我们会专注于异步编程，并且第二十一章的项目会在更现实的场景中使用这些概念，而不像本章中讨论的这些小例子。

正如之前提到的，因为 Rust 本身很少有处理并发的部分内容，有很多的并发方案都由 crate 实现。它们比标准库要发展的更快；请在网上搜索当前最新的用于多线程场景的 crate。

Rust 提供了用于消息传递的信道，和像 `Mutex<T>` 和 `Arc<T>` 这样可以安全的用于并发上下文的智能指针。类型系统和借用检查器会确保这些场景中的代码不会出现数据竞争和无效的引用。一旦代码可以编译了，我们就可以坚信这些代码可以正确地运行于多线程环境，而不会出现其他语言中经常出现的那些难以追踪的 bug。并发编程不再是什么可怕的概念：无所畏惧地并发起来吧！

Async 和 await

很多我们要求计算机处理的操作都需要一定的时间才能完成。例如，如果你使用视频编辑器来创建一个家庭聚会的视频，导出视频可能会花费几分钟到几小时不等。同样，从家庭成员那里下载共享的视频也可能需要很长时间。如果我们能在等待这些长时间运行的操作完成期间做点其他事情，那就太好了。

视频导出会尽可能使用所有的 CPU 和 GPU。如果你只有一个 CPU 核，同时操作系统在导出完成前也不会暂停，那么在其运行期间你无法使用计算机进行任何其他操作。这会是一个非常糟糕的体验。相反计算机的操作系统可以（也确实可以）隐式地中断导出过程，频率足够高，使你能够在导出进行的同时完成其他任务。

下载文件则有所不同。它不占用大量的 CPU 时间。相反 CPU 需要等待来自于网络的数据。虽然可以在部分数据就绪时就开始读取，但等待剩余数据可能还需要一段时间。即便数据全部就绪了，视频文件也可能非常大，因此加载所有数据也会花费一些时间。虽然这可能只需要一两秒，不过这对于一个现代处理器来说已经是非常长的时间了，因为它每秒可以执行数十亿次操作。因此，如果能让 CPU 在等待网络调用完成的同时去处理别的工作就再好不过了。所以同上操作系统会隐式地中断你的程序以便其它工作可以在网络操作进行的同时继续进行。

> 注意：视频导出这类操作通常被称为 “CPU 密集型”（“CPU-bound”）或者 “计算密集型”（“compute-bound”）操作。其受限于计算机 *CPU* 或 *GPU* 处理数据的速度，以及它所能利用的计算能力。而下载视频这类操作通常被称为 “IO 密集型”（“IO-bound”）操作，因为其受限于计算机的 输入输出 速度。下载的速度最多只能与通过网络传输数据的速度一致。

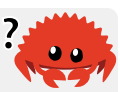
在上述两个例子中，操作系统的隐式中断提供了一种形式的并发。不过这种并发仅限于整个程序的级别：操作系统中断一个程序并让其它程序得以执行。在很多场景中，由于我们能比操作系统在更细粒度上理解我们的程序，因此我们可以观察到很多操作系统无法察觉的并发机会。

例如，如果我们在构建一个管理文件下载的工具，我们应当以一种不会因开始一个下载任务而锁定 UI 的方式来编写程序，并且用户应该能够同时开始多个下载任务。不过很多操作系统与网络交互的 API 都是 阻塞 的（*blocking*）。也就是说这些 API 会阻塞程序的进程，直到它们处理的数据完全就绪。

> 注意：如果你仔细思索一下，会发现这是 大部分 函数调用的工作方式！不过我们通常将 “阻塞” 这个术语保留给那些与文件、网络或其它计算机资源交互的函数调用，因为这些地方是单个程序可以从 非 阻塞操作中获益的地方。

我们可以新建专用的线程来下载每个文件以免阻塞主线程。然而，我们最终会发现这些线程的开销会成为一个问题。如果这些调用在一开始就是非阻塞的话那就更理想了。最后，如果我们能够像在阻塞代码中一样，以直接的风格编写非阻塞代码，那就更好了。比如这样：

```
let data = fetch_data_from(url).await;
println!("{data}");
```



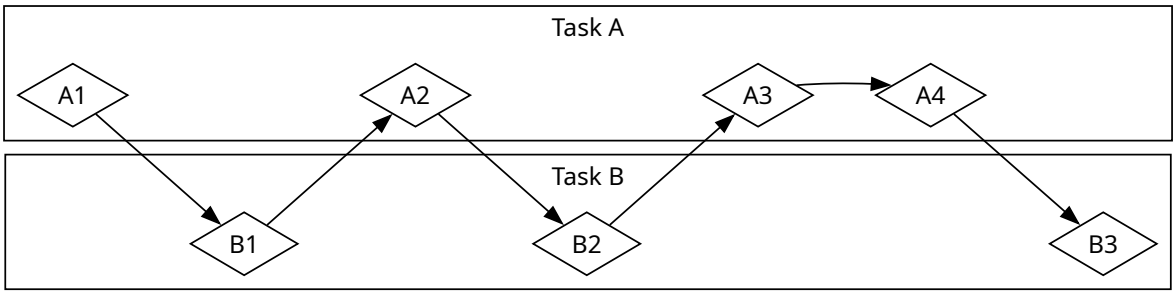
这正是 Rust 的 async 抽象所提供的。不过在讲解它们在实践中如何工作之前，让我们稍微绕个远路来了解一下并行（parallelism）和并发（concurrency）的区别。

并行与并发

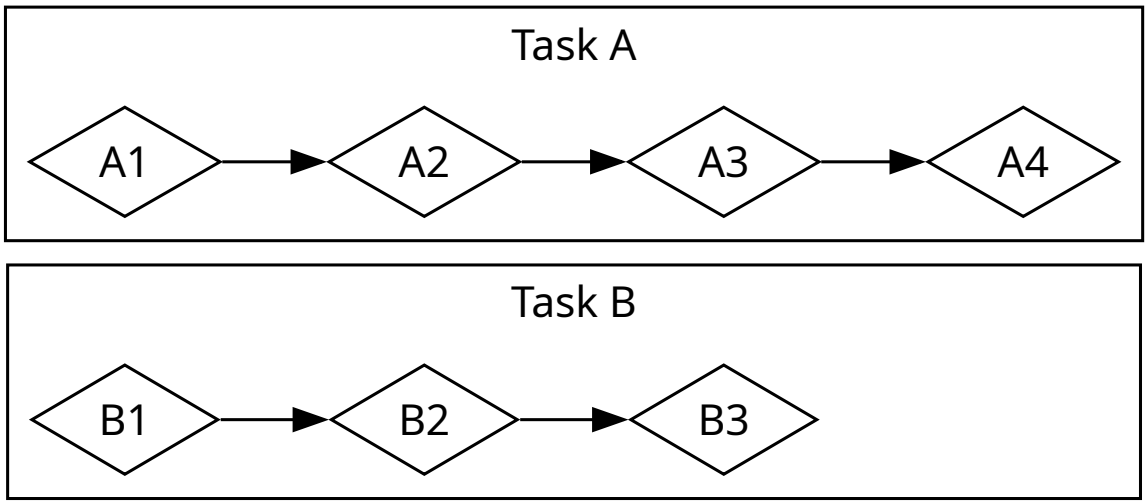
在上一章中，我们大致将并行和并发视为可以互换的概念。但现在我们需要更加精确地区分它们，因为它们的区别将在实际工作中显现出来。

思考一下不同的团队分割方法来开发一个软件项目。我们可以分配给一个个人多个任务，也可以每个团队成员各自负责一个任务，或者可以采用这两种方法的组合。

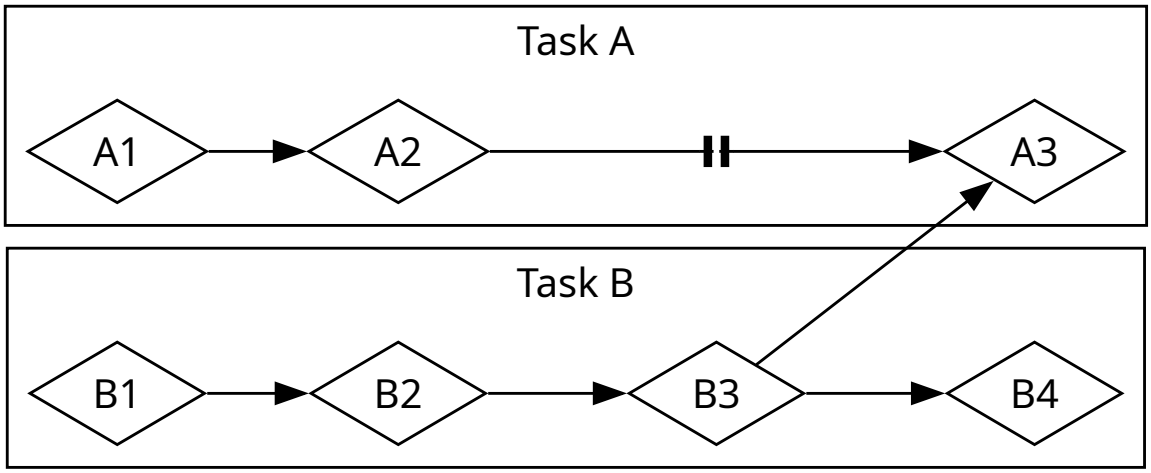
当一个个人在任何一个任务完成前同时处理多个任务，这就是 并发。你可能在计算机上同时运行两个项目，当你对其中一个项目感到厌倦或遇到困难时，可以切换到另一个项目。因为你是单独一个人，所以无法真正同时推进两个任务，但是你可以多任务处理，在不同任务之间切换以取得进展。



当你同意将一组任务在组员中分配，每一个组员分配一个任务并单独处理它，这就是 并行。每个组员可以真正同时进行工作。



在这两种场景中，你可能需要协调不同的任务。也许你 认为 某个人负责的任务与其他人的工作完全不相关，但实际上它确实依赖于团队中另一位成员的工作完成。一些工作可以并行进行，不过一些工作事实上是 串行 的：它们只能串行地发生，一个接着一个，如图 17-3 所示。



同理，你可能会意识到你自己的一个任务依赖另一个任务。现在并发任务也变成串行的了。

并行与并发也可能相互交叉（阻塞）。如果你得知某个同事卡在等待你的一个任务完成，你可能会集中所有精力在这个任务上来 “解锁” 你的同事。你和你的同事则不再能并行地工作了，同时你也不能够并发地处理自己的任务。

同样的基础动态也作用于软件与硬件。在一个单核的机器上，CPU 一次只能执行一个操作，不过它仍然可以并发工作。借助像线程、进程和异步（async）等工具，计算机可以暂停一个活动，并在最终切换回第一个活动之前切换到其它活动。在一个有多个 CPU 核心的机器上，它也可以并行工作。一个核心可以做一件工作的同时另一个核心可以做一些完全不相关的工作，而且这些工作实际上是同时发生的。

当使用 Rust 中的 async 时，我们总是在处理并发。取决于硬件、操作系统和所使用的异步运行时（async runtime）– 稍后会介绍更多的异步运行时！并发也可能在底层使用了并行。

现在让我们深入理解 Rust 的异步编程实际上是如何工作的！在接下来的章节中，我们将：

- 学习如何使用 Rust 的 `async` 和 `await` 语法
- 探索如何使用异步模型来解决第十六章中遇到的一些挑战
- 了解多线程和异步如何互补，在很多场景中你甚至可以同时使用两者

Futures 和 async 语法

Rust 异步编程的关键元素是 *futures* 和 Rust 的 `async` 与 `await` 关键字。

future 是一个现在可能还没有准备好但将在未来某个时刻准备好的值。（相同的概念也出现在很多语言中，有时被称为 “task” 或者 “promise”。）Rust 提供了 `Future` trait 作为基础组件，这样不同的异步操作就可以在不同的数据结构上实现。在 Rust 中，我们称实现了 `Future` trait 的类型为 futures。每一个实现了 `Future` 的类型会维护自己的进度状态信息和 “ready” 的定义。

`async` 关键字可以用于代码块和函数，表明它们可以被中断并恢复。在一个 async 块或 async 函数中，可以使用 `await` 关键字来等待一个 future 准备就绪，这一过程称为 等待一个 *future*。async 块或 async 函数中每一个等待 future 的地方都可能是一个 async 块或 async 函数中断并随后恢复的点。检查一个 future 并查看其值是否已经准备就绪的过程被称为 轮询（polling）。

其它一些语言也使用 `async` 和 `await` 关键字进行异步编程。如果你熟悉这些语言，则可能会注意到它们与 Rust 的处理方式有显著不同，包括语法上的差异。我们将看到，这样做是有充分理由的！

在大多数情况下，编写异步 Rust 代码时，我们使用 `async` 和 `await` 关键字。Rust 将其编译为等同于使用 `Future` trait 的代码，这非常类似于将 `for` 循环编译为等同于使用 `Iterator` trait 的代码。不过，由于 Rust 提供了 `Future` trait，你也可以在需要时为你自己的数据类型实现它。在整个章节中你会看到很多函数的返回值类型都有其自己的 `Future` 实现。我们会在本章结尾回到这个 trait 的定义，并深入了解它的工作原理，但现在这些细节已经足够让我们继续前进了。

这些内容可能有点抽象。让我们来编写第一个异步程序：一个小型网络爬虫。我们会从命令行传递两个 URL，并发地解析它们，并返回第一个完成解析的结果。这个示例会引入不少的新语法，不过不用担心。我们会逐步解释所有你需要了解的内容。

第一个异步程序

为了保持本章的内容专注于学习 async，而不是处理生态系统的部分组件，我们已经创建了一个 `trpl` crate（`trpl` 是 “The Rust Programming Language” 的缩写）。它重导出了你需要的所有类型、traits 和函数，它们主要来自于 `futures` 和 `tokio` crates。

- `futures` crate 是一个 Rust 异步代码实验的官方仓库，也正是 `Future` 最初设计的地方。
- Tokio 是目前 Rust 中应用最广泛的异步运行时（async runtime），特别是（但不仅是！）web 应用。这里还有其他优秀的运行时，它们可能更适合你的需求。我们在 `trpl` 的底层使用 Tokio 是因为它经过了充分测试且广泛使用。

在一些场景中，`trpl` 也会重命名或者封装原始 API 以便我们专注于与本章相关的细节。如果你想了解该 crate 的具体功能，我们鼓励你查看其源码。你可以看到每个重导出的内容来自哪个 crate，我们留下了大量注释来解释这个 crate 的用途。

创建一个名为 `hello-async` 的二进制项目并将 `trpl` crate 作为一个依赖添加：

```
$ cargo new hello-async
$ cd hello-async
$ cargo add trpl
```

现在我们可以利用 `trpl` 提供的多种组件来编写第一个异步程序。我们构建了一个小的命令行工具来抓取两个网页，拉取各自的 `<title>` 元素，并打印出第一个完成全部过程的标题。

让我们开始编写一个函数，它获取一个网页 URL 作为参数，请求该 URL 并返回标题元素的文本：

文件名：src/main.rs

```
use trpl::Html;

async fn page_title(url: &str) -> Option<String> {
    let response = trpl::get(url).await;
    let response_text = response.text().await;
    Html::parse(&response_text)
        .select_first("title")
        .map(|title_element| title_element.inner_html())
}
```

在示例 17-1 中，我们定义了一个名为 `page_title` 的函数，并使用了 `async` 关键字标记。接着我们使用 `trpl::get` 函数来获取传入的任意 URL，然后使用 `await` 关键字来等待响应。接着我们调用其 `text` 方法来获取响应的文本，这里再一次使用 `await` 关键字等待。这两个步骤都是异步的。对于 `get` 来说，我们需要等待服务器发送回其响应的第一部分，这会包含 HTTP 头（headers）、cookies 等。这部分响应可以独立于响应体发送。特别是在响应体非常大的时候，接收完整响应可能会花费一些时间。因此我们不得不等待响应 整体 返回，所以 `text` 方法也是异步。

我们必须显式地等待这两个 futures，因为 Rust 中的 futures 是 惰性（*lazy*）的：在你使用 `await` 请求之前它们不会执行任何操作。（事实上，如果你不使用一个 futures，Rust 会显示一个编译警告）这应该会让你想起之前第十三章关于迭代器的讨论。直到你调用迭代器的 `next` 方法（直接调用或者使用 `for` 循环或者类似 `map` 这类在底层使用 `next` 的方法）之前它们什么也不会做。对于 futures 来说，同样的基本理念也是适用的：除非你显式地请求，否则它们不会执行。惰性使得 Rust 可以避免提前运行异步代码，直到真正需要时才执行。

> 注意：这不同于上一章节中 `thread::spawn` 的行为，当时传递给另一个线程的闭包会立即开始运行。这也与许多其他语言处理异步的方式不同！但对于 Rust 而言，这一点非常重要。稍后我们会解释原因。

当我们有了 `response_text` 函数，就可以使用 `Html::parse` 将其解析为一个 `Html` 类型的实例。不同于原始字符串，现在我们有了一个可以将 HTML 作为更丰富数据结构来操作的数据类型。特别是我们可以使用 `select_first` 方法来找出给定 CSS 选择器（selector）中第一个匹配元素。通过传递字符串 `"title"`，我们会得到文档中的第一个 `<title>` 元素，如果它存在的话。由于可能没有任何匹配的元素，`select_first` 返回一个 `Option<ElementRef>`。最后我们使用 `Option::map` 方法，它允许我们在 `Option` 中有元素时对其进行处理，而在没有时则什么也不做。（这里也可以使用一个 `match` 表达式，但 `map` 更符合惯用的写法。）在传递给 `map` 的函数体中，我们调用了 `title_element` 上的 `inner_html` 来获取其内容，这是一个 `String`。当上面所讲的都完成后，我们会得到一个 `Option<String>`。

注意 Rust 的 `await` 关键字出现在需要等待的表达式之后而不是之前。也就是说，这是一个 后缀关键字（*postfix keyword*）。如果你在其它语言中使用过 async 的话，这可能与你所熟悉的有所不同。Rust 如此选择是因为这使得方法的链式调用更加简洁。因此，我们可以修改 `page_url_for` 的函数体来链式调用 `trpl::get` 和 `text` 并在其之间使用 `await`，如示例 17-2 所示：

文件名：src/main.rs

```
    let response_text = trpl::get(url).await.text().await;
```

这样我们就成功编写了第一个异步函数！在我们向 `main` 加入一些代码调用它之前，让我们再多了解下我们写了什么以及它的意义。

当 Rust 遇到一个 `async` 关键字标记的代码块时，会将其编译为一个实现了 `Future` trait 的唯一的、匿名的数据类型。当 Rust 遇到一个被标记为 `async` 的函数时，会将其编译进一个拥有异步代码块的非异步函数。异步函数的返回值类型是编译器为异步代码块所创建的匿名数据类型。

因此，编写 `async fn` 就等同于编写一个返回类型为 *future* 的函数。当编译器遇到类似示例 17-1 中 `async fn page_title` 的函数定义时，它等价于以下定义的非异步函数：

```
use std::future::Future;
use trpl::Html;

fn page_title(url: &str) -> impl Future<Output = Option<String>> + '_ {
    async move {
        let text = trpl::get(url).await.text().await;
        Html::parse(&text)
            .select_first("title")
            .map(|title| title.inner_html())
    }
}
```

让我们挨个看一下转换后版本的每一个部分：

- 它使用了之前第十章 “trait 作为参数” 部分讨论过的 `impl Trait` 语法
- 它返回的 trait 是一个 `Future`，它有一个关联类型 `Output`。注意 `Output` 的类型是 `Option<String>`，这与 `async fn` 版本的 `page_title` 的原始返回值类型相同。
- 所有原始函数中被调用的代码被封装进一个 `async move` 块。回忆一下，代码块是表达式。这整个块就是函数所返回的表达式
- 如上所述，这个异步代码块产生一个 `Option<String>` 类型的值。这个值与返回类型中的 `Output` 类型一致。这正类似于你已经见过的其它代码块。
- 新版函数的函数体是一个 `async move` 代码块，因为它如何使用 `url` 参数决定了这一点。（本章后续部分将更详细地讨论 `async` 和 `async move` 之间的区别。）
- 新版本的函数在返回类型中使用了一种我们之前未见过的生命周期标记：`'_`。因为函数返回的 `Future` 指向一个引用（在这个例子中是指向 `url` 参数的引用）我们需要告诉 Rust 引用的生命周期。这里无需命名该生命周期，因为 Rust 足够智能到能理解这里只涉及到唯一一个引用，不过我们 必须 明确指出返回的 `Future` 受该生命周期的约束。

现在我们可以在 `main` 中调用 `page_title`。首先，我们只会获取一个页面的标题。在示例 17-3 中，我们沿用了第十二章中获取命令行参数的相同模式。接着我们传递第一个 URL 给

`page_title`，并等待结果。因为 future 产生的值是一个 `Option<String>`，我们使用 `match` 表达式来根据页面是否有 `<title>` 来打印不同的信息。

文件名：src/main.rs



```
async fn main() {
    let args: Vec<String> = std::env::args().collect();
    let url = &args[1];
    match page_title(url).await {
        Some(title) => println!("The title for {url} was {title}"),
        None => println!("{url} had no title"),
    }
}
```

很不幸的是这还不能编译。唯一可以使用 `await` 关键字的地方是异步函数或者代码块中，同时 Rust 不允许将特殊的 `main` 函数标记为 `async`。

```
error[E0752]: `main` function is not allowed to be `async`
 --> src/main.rs:6:1
  |
6 | async fn main() {
  | ^^^^^^^^^^^^^^^ `main` function is not allowed to be `async`
```

`main` 不能标记为 `async` 的原因是异步代码需要一个 运行时：即一个管理执行异步代码细节的 Rust crate。一个程序的 `main` 函数可以 初始化 一个运行时，但是其 自身 并不是一个运行时。（稍后我们会进一步解释原因。）每一个执行异步代码的 Rust 程序必须至少有一个设置运行时并执行 futures 的地方。

大部分支持异步的语言会打包一个运行时在语言中，Rust 则不是。相反，这里有很多不同的异步运行时，每一个都有适合其目标的权衡取舍。例如，一个拥有很多核心和大量内存的高吞吐 web server 与一个单核、少量内存并且没有堆分配能力的微控制器相比有着截然不同的需求。提供这些运行时的 crate 通常也提供了例如文件或者网络 IO 这类常用功能的异步版本。

从这里到本章余下部分，我们会使用 `trpl` crate 的 `run` 函数，它获取一个 future 作为参数并运行到结束。在内部，调用 `run` 会设置一个运行时来运行传递的 future。一旦 future 完成，`run` 返回 future 返回的任何值。

我们可以将 `page_title` 返回的 future 直接传递给 `run`。一旦其完成，我们能够匹配返回的 `Option<String>`，正如示例 17-3 我们尝试的那样。然而，在本章的大部分示例中（以及大多数实际应用中的异步代码中!），我们会执行不止一次异步函数调用，所以相反我们会传递一个 `async` 块并显式地等待 `page_title` 调用的结果，如示例 17-4 所示。

文件名：src/main.rs

```
fn main() {
    let args: Vec<String> = std::env::args().collect();

    trpl::run(async {
        let url = &args[1];
        match page_title(url).await {
            Some(title) => println!("The title for {url} was {title}"),
            None => println!("{url} had no title"),
```

```
        }
    })
}
```

当我们运行代码，我们会得到最初预想的行为：

```
$ cargo run -- https://www.rust-lang.org
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.05s
     Running `target/debug/async_await 'https://www.rust-lang.org'`
The title for https://www.rust-lang.org was
            Rust Programming Language
```

我们终于有了一些可以正常工作的异步代码！现在它们可以成功编译并运行。在我们添加代码让两个网址进行竞争之前，让我们简要地回顾一下 future 是如何工作的。

每一个 *await point*，也就是代码使用 `await` 关键字的地方，代表将控制权交还给运行时的地方。为此 Rust 需要记录异步代码块中涉及的状态，这样运行时可以去执行其他工作，并在准备好时回来继续推进当前的任务。这就像你通过编写一个枚举来保存每一个 `await` point 的状态一样：

```
enum PageTitleFuture<'a> {
    Initial { url: &'a str },
    GetAwaitPoint { url: &'a str },
    TextAwaitPoint { response: trpl::Response },
}
```

编写代码来手动控制不同状态之间的转换是非常乏味且容易出错的，特别是之后增加了更多功能和状态的时候。相反，Rust 编译器自动创建并管理异步代码的状态机数据结构。如果你感兴趣的话：是的，正常的借用和所有权也全部适用于这些数据结构。幸运的是，编译器也会为我们处理这些检查，并提供友好的错误信息。本章稍后会讲解一些相关内容！

最终需要某个组件来执行状态机。这就是运行时。（这也是为什么在了解运行时的时候，你可能会看到 *executors* 这个词：executor 是运行时中负责执行异步代码的部分。）

现在我们能够理解之前示例 17-3 中为何编译器阻止我们将 `main` 本身标记为异步函数了。如果 `main` 是一个异步函数，需要有其它组件来管理 `main` future 返回的状态机，但是 `main` 是程序的入口点！为此我们在 `main` 函数中调用 `trpl::run`，它设置了一个运行时并运行 `async` 块返回的 future 并等待它返回 `Ready`。

> 注意：一些运行时提供了相关的宏，所以你 可以 编写一个异步 `main` 函数。这些宏将 `async fn main() { ... }` 重写为正常的 `fn main`，执行的逻辑与我们在示例 17-5 中手动实现的一样：像 `trpl::run` 一样调用一个函数运行 future 直到结束。

让我们将这些代码片段整理一下来看看如何编写并发代码，这里通过两个来自命令行的不同 URL 来调用 `page_title` 并使其相互竞争。

文件名：src/main.rs

```
use trpl::{Either, Html};

fn main() {
    let args: Vec<String> = std::env::args().collect();

    trpl::run(async {
        let title_fut_1 = page_title(&args[1]);
        let title_fut_2 = page_title(&args[2]);

        let (url, maybe_title) =
            match trpl::race(title_fut_1, title_fut_2).await {
                Either::Left(left) => left,
                Either::Right(right) => right,
            };

        println!("{url} returned first");
        match maybe_title {
            Some(title) => println!("Its page title is: '{title}'"),
            None => println!("Its title could not be parsed."),
        }
    })
}

async fn page_title(url: &str) -> (&str, Option<String>) {
    let text = trpl::get(url).await.text().await;
    let title = Html::parse(&text)
        .select_first("title")
        .map(|title| title.inner_html());
    (url, title)
}
```

示例 17-5 中以分别由用户提供的 URL 调用 `page_title` 开始。我们将调用 `page_title` 产生的 future 分别保存为 `title_fut_1` 和 `title_fut_2`。请记住，它们还没有进行任何工作，因为 future 是惰性的，并且我们还没有 `await` 它们。接着我们将 futures 传递给 `trpl::race`，它返回一个值表明哪个传递的 future 最先返回。

> 注意：在内部 `race` 构建在一个更通用的函数 `select` 之上，你会在真实的 Rust 代码中更常遇到它。`select` 函数可以做很多 `trpl::race` 函数做不了的事，不过它也有一些额外的复杂性，所以目前我们先略过介绍。

由于任何一个 future 都可以合理地 “获胜”，所以返回 `Result` 没有意义。相反 `race` 返回了一个我们之前没有见过的类型 `trpl::Either`。`Either` 类型有点类似于 `Result`，它也有两个成员。但是不同于 `Result`，`Either` 没有内置成功或者失败的概念。相反它使用 `Left` 和 `Right` 来表示 “一个或另一个”。

```
enum Either<A, B> {
    Left(A),
    Right(B),
}
```

如果第一个参数先完成，`race` 函数返回 `Left` 并包含该 future 的输出，如果第二个 future 先完成，则返回 `Right` 和第二个 future 的输出。这匹配调用函数时参数出现的顺序：第一个参数在第二个参数的左边。

我们还更新了 `page_title` 来返回与传递时相同的 URL。如此如果首先返回的页面没有可以解析的 `<title>`，仍然可以打印出有意义的信息。有了这些信息，我们对 `println!` 的输出进行了封装和更新，以表明哪个 URL 最先完成，并在页面有 `<title>` 时打印出它的内容。

现在我们完成一个小型网页爬虫的构建了！挑选一对 URL 并运行命令行工具。你会发现某些网站稳定地快于其它网站，而有些情况哪些网站会 赢 则每次都不同。更重要的是，你已经掌握了处理 futures 的基础知识，因此我们现在可以进一步探索更多异步操作的可能性了。

并发与 async

在这一部分，我们将使用异步来应对一些与第十六章中通过线程解决的相同的并发问题。因为之前我们已经讨论了很多关键理念了，这一部分我们会专注于线程与 future 的区别。

在很多情况下，使用异步处理并发的 API 与使用线程的非常相似。在其它的一些情况，它们则非常不同。即便线程与异步的 API 看起来 很类似，通常它们有着不同的行为，同时它们几乎总是有着不同的性能特点。

计数

第十六章中我们应付的第一个任务是在两个不同的线程中计数。让我们用异步来完成相同的任务。`trpl` crate 提供了一个 `spawn_task` 函数，它看起来非常像 `thread::spawn` API，和一个 `sleep` 函数，这是 `thread::sleep` API 的异步版本。我们可以将它们结合使用，实现与线程示例相同的计数功能，如示例 17-6 所示。

文件名：src/main.rs

```
use std::time::Duration;

fn main() {
    trpl::run(async {
        trpl::spawn_task(async {
            for i in 1..10 {
                println!("hi number {i} from the first task!");
                trpl::sleep(Duration::from_millis(500)).await;
            }
        });

        for i in 1..5 {
            println!("hi number {i} from the second task!");
            trpl::sleep(Duration::from_millis(500)).await;
        }
    });
}
```

作为开始，我们在 `main` 函数中使用 `trpl::run`，这样我们的顶层函数可以是异步的。

> 注意：本章从现在开始，每一个示例的 `main` 中都会包含几乎相同的 `trpl::run` 封装代码，所以我们经常会连同 `main` 一同省略。别忘了在你的代码中加入它们！

接着我们在代码块中编写了两个循环，每个其中都有一个 `trpl::sleep` 调用，每一个都在发送下一个信息之前等待半秒（500 毫秒）。我们将一个循环放到 `trpl::spawn_task` 中并将另一个放在顶层的 `for` 循环中。我们也在 `sleep` 调用之后加入了一个 `await`。

这个实现与基于线程的版本类似，包括在运行时，你可能会在终端中看到消息以不同顺序出现的情况。

```
hi number 1 from the second task!
hi number 1 from the first task!
```

```
hi number 2 from the first task!
hi number 2 from the second task!
hi number 3 from the first task!
hi number 3 from the second task!
hi number 4 from the first task!
hi number 4 from the second task!
hi number 5 from the first task!
```

这个版本在 main 中的异步代码块中 for 循环结束后就停止了，因为当 main 函数结束时 `spawn_task` 产生的任务就会关闭。如果运行该任务直到结束，你需要使用一个 join 句柄（join handle）来等待第一个任务完成。对于线程来说，可以使用 `join` 方法来 “阻塞” 直到线程结束运行。在示例 17-7 中，我们可以使用 `await` 来实现相同的效果，因为任务句柄本身是一个 future。它的 `Output` 类型是一个 `Result`，所以我们还需要 unwrap 来 await 它。

文件名：src/main.rs

```
        let handle = trpl::spawn_task(async {
            for i in 1..10 {
                println!("hi number {i} from the first task!");
                trpl::sleep(Duration::from_millis(500)).await;
            }
        });

        for i in 1..5 {
            println!("hi number {i} from the second task!");
            trpl::sleep(Duration::from_millis(500)).await;
        }

        handle.await.unwrap();
```

更新后的版本会运行 两个 循环直到结束。

```
hi number 1 from the second task!
hi number 1 from the first task!
hi number 2 from the first task!
hi number 2 from the second task!
hi number 3 from the first task!
hi number 3 from the second task!
hi number 4 from the first task!
hi number 4 from the second task!
hi number 5 from the first task!
hi number 6 from the first task!
hi number 7 from the first task!
hi number 8 from the first task!
hi number 9 from the first task!
```

目前为止，看起来异步和线程版本给出了基本一样的输出，它们只是使用了不同的语法：在 join 句柄上使用 `await` 而不是调用 `join`，和 await `sleep` 调用。

最大的区别在于无需再产生另一个操作系统线程来进行工作。事实上，我们甚至不需要产生一个任务。因为异步代码块会编译为匿名 future，我们可以将每一个循环放进一个异步代码块并使用 `trpl::join` 方法来让运行时将它们两个都运行至完成。

在第十六章中，我们展示了如何在 `std::thread::spawn` 调用返回的 `JoinHandle` 类型上调用 `join` 方法。`trpl::join` 函数也类似，不过它作用于 future。当你传递两个 future，它会产生单独一个 future 但它的输出是一个元组，当 两者 都完成时其中有每一个传递给它的 future 的输出。因此，在示例 17-8 中，我们使用 `trpl::join` 来等待 `fut1` 和 `fut2` 都结束。我们 没有 await `fut1` 和 `fut2`，而是等待 `trpl::join` 新产生的 future。我们忽略其输出，因为它只是一个包含两个单元值（unit value）的元组。

文件名：src/main.rs

```
        let fut1 = async {
            for i in 1..10 {
                println!("hi number {i} from the first task!");
                trpl::sleep(Duration::from_millis(500)).await;
            }
        };

        let fut2 = async {
            for i in 1..5 {
                println!("hi number {i} from the second task!");
                trpl::sleep(Duration::from_millis(500)).await;
            }
        };

        trpl::join(fut1, fut2).await;
```

当运行代码我们会看到两个 future 会运行至结束：

```
hi number 1 from the first task!
hi number 1 from the second task!
hi number 2 from the first task!
hi number 2 from the second task!
hi number 3 from the first task!
hi number 3 from the second task!
hi number 4 from the first task!
hi number 4 from the second task!
hi number 5 from the first task!
hi number 6 from the first task!
hi number 7 from the first task!
hi number 8 from the first task!
hi number 9 from the first task!
```

这里，你每次都会看到完全相同的顺序，这与我们在线程中看到的情况非常不同。这是因为 `trpl::join` 函数是 公平的（*fair*），这意味着它以相同的频率检查每一个 future，使它们交替执行，绝不会让一个任务在另一个任务准备好时抢先执行。对于线程来说，操作系统会决定该检查哪个线程和会让它运行多长时间。对于异步 Rust 来说，运行时决定检查哪一个任务。（在实践中，细节会更为复杂，因为异步运行时可能在底层使用操作系统线程来作为其并发管理的一部分，因此要保证公平性可能会增加运行时的工作量，但这仍然是可行的！）运行时无需为任何操作保证公平性，同时运行时也经常提供不同的 API 来让你选择是否需要公平性。

尝试这些不同的 await future 的变体来观察它们的效果：

- 去掉一个或者两个循环外的异步代码块。
- 在定义两个异步代码块后立刻 await 它们。

- 只将第一个循环封装进异步代码块，并在第二个循环体之后 await 作为结果的 future。

作为额外的挑战，看看你能否在运行代码 之前 想出每个情况下的输出！

消息传递

在 future 之间共享数据也与线程类似：我们会再次使用消息传递，不过这次使用的是异步版本的类型和函数。我们会采用与之前第十六章中使用的稍微不同的方法，来展示一些基于线程的并发与基于 future 的并发之间的关键差异。在示例 17-9 中，我们会从仅有一个异步代码块开始，不像 之前产生独立线程那样产生一个独立的任务。

文件名：src/main.rs

```
        let (tx, mut rx) = trpl::channel();

        let val = String::from("hi");
        tx.send(val).unwrap();

        let received = rx.recv().await.unwrap();
        println!("Got: {received}");
```

这里我们使用了 `trpl::channel`，一个第十六章用于线程的多生产者、单消费者信道 API 的异步版本。异步版本的 API 与基于线程的版本只有一点微小的区别：它使用一个可变的而不是不可变的 `rx`，并且它的 `recv` 方法产生一个需要 await 的 future 而不是直接返回值。现在我们可以发送端向接收端发送消息了。注意我们无需产生一个独立的线程或者任务；只需等待（await） `rx.recv` 调用。

`std::mpsc::channel` 中的同步 `Receiver::recv` 方法阻塞执行直到它接收一个消息。`trpl::Receiver::recv` 则不会阻塞，因为它是异步的。不同于阻塞，它将控制权交还给运行时，直到接收到一个消息或者信道的发送端关闭。相比之下，我们不用 await `send`，因为它不会阻塞。也无需阻塞，因为信道的发送端的数量是没有限制的。

> 注意：因为所有这些异步代码都运行在一个 `trpl::run` 调用的异步代码块中，其中的所有代码可以避免阻塞。然而，外面 的代码会阻塞到 `run` 函数返回。这正是 `trpl::run` 函数的全部意义：它允许你 选择 在何处阻塞一部分异步代码，也就是在何处进行同步和异步代码的转换。这正是在大部分运行时中 `run` 实际上被命名为 `block_on` 的原因。

请注意这个示例中的两个地方：首先，消息立刻就会到达！其次，虽然我们使用了 future，但是这里还没有并发。示例中的所有事情都是顺序发生的，就像没涉及到 future 时一样。

让我们通过发送一系列消息并在之间休眠来解决第一个问题，如示例 17-10 所示：

文件名：src/main.rs

```
        let (tx, mut rx) = trpl::channel();

        let vals = vec![
            String::from("hi"),
            String::from("from"),
```

```
            String::from("the"),
            String::from("future"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            trpl::sleep(Duration::from_millis(500)).await;
        }

        while let Some(value) = rx.recv().await {
            println!("received '{value}'");
        }
```

除了发送消息之外，我们还需要接收它们。在这个例子中我们可以手动接收，就是调用四次 `rx.recv().await`，因为我们知道进来了多少条消息。然而，在现实世界中，我们通常会等待未知 数量的消息。这时我们需要一直等待直到可以确认没有更多消息了为止。

在示例 16-10 中，我们使用了 `for` 循环来处理从异步信道接收的所有消息。然而，Rust 目前还没有在 异步 序列上编写 `for` 循环的方法。取而代之的是，我们需要一个我们还没有见过的新循环类型，即 `while let` 条件循环。`while let` 循环是我们在第六章中见过的 `if let` 结构的循环版本。只要其指定的模式持续匹配循环就会一直执行。

`rx.recv` 调用产生一个 `Future`，我们会 await 它。运行时会暂停 `Future` 直到它就绪。一旦消息到达，future 会解析为 `Some(message)`，每次消息到达时都会如此。当信道关闭时，不管是否有 任何 消息到达，future 都会解析为 `None` 来表明没有更多的值了，我们也就应该停止轮询，也就是停止等待。

`while let` 循环将上述逻辑整合在一起。如果 `rx.recv().await` 调用的结果是 `Some(message)`，我们会得到消息并可以在循环体中使用它，就像使用 `if let` 一样。如果结果是 `None`，则循环停止。每次循环执行完毕，它会再次触发 await point，如此运行时会再次暂停直到另一条消息到达。

现在代码可以成功发送和接收所有的消息了。不幸的是，这里还有一些问题。首先，消息并不是按照半秒的间隔到达的。它们在程序启动后两秒（2000 毫秒）后立刻一起到达。其次，程序永远也不会退出！相反它会永远等待新消息。你会需要使用 ctrl-c 来关闭它。

让我们开始理解为何消息在全部延迟后立刻一起到达，而不是逐个在延迟后到达。在一个给定的异步代码块，`await` 关键字在代码中出现的顺序也就是程序执行时其发生的顺序。

示例 17-10 中只有一个异步代码块，所以所有的代码线性地执行。这里仍然没有并发。所有 `tx.send` 调用与 `trpl::sleep` 调用及其相关的 await point 是依次进行的。只有在此之后 `while let` 循环才开始执行 `recv` 调用上的 `await` point。

为了得到我们需要的行为，在接收每条消息之间引入休眠延迟，我们需要将 `tx` 和 `rx` 操作放置于它们各自的异步代码块中。这样运行时就可以使用 `trpl::join` 来分别执行它们，就像在计数示例中一样。我们再一次 await `trpl::join` 调用的结果，而不是它们各自的 future。如果我们顺序地 await 单个 future，则就又回到了一个顺序流，这正是我们 不 希望做的。

文件名：src/main.rs

```
        let tx_fut = async {
            let vals = vec![
```

```
                String::from("hi"),
                String::from("from"),
                String::from("the"),
                String::from("future"),
            ];

            for val in vals {
                tx.send(val).unwrap();
                trpl::sleep(Duration::from_millis(500)).await;
            }
        };

        let rx_fut = async {
            while let Some(value) = rx.recv().await {
                println!("received '{value}'");
            }
        };

        trpl::join(tx_fut, rx_fut).await;
```

采用示例 17-11 中的更新后的代码，消息会以 500 毫秒的间隔打印，而不是在两秒后就全部一起打印。

但是程序仍然永远也不会退出，这是由于 `while let` 循环与 `trpl::join` 的交互方式所致：

- `trpl::join` 返回的 future 只会完成一次，即传递的 两个 future 都完成的时候。
- `tx` future 在发送 `vals` 中最后一条消息之后的休眠结束后立刻完成。
- `rx` future 直到 `while let` 循环结束之前都不会完成。
- 当信道的另一端关闭后 await `rx.recv` 将只会返回 `None`。
- 信道只有在调用 `rx.close` 或者发送端 `tx` 被丢弃时才会关闭。
- 我们没有在任何地方调用 `rx.close`，并且 `tx` 直到传递给 `trpl::run` 的最外层异步代码块结束前都不会被丢弃。
- 代码块不能结束是因为它阻塞在了等待 `trpl::join` 完成，这就又回到了列表的开头！

我们可以在代码的某处调用 `rx.close` 来手动关闭 `rx`，不过这并没有太多意义。在处理了任意数量的消息后停止可以使程序停止，但是可能会丢失消息。我们需要其它的手段来确保 `tx` 在函数的结尾 之前 被丢弃。

目前发送消息的异步代码块只是借用了 `tx`，因为发送消息并不需要其所有权，但是如果我们可以将 `tx` 移动（move）进异步代码块，它会在代码块结束后立刻被丢弃。在第十三章中我们学习了如何在闭包上使用 `move` 关键字，在第十六章中，我们知道了使用线程时经常需要移动数据进闭包。同样的基本原理也适用于异步代码块，因此 `move` 关键字也能像闭包那样作用于异步代码块。

在示例 17-12 中，我们将发送消息的异步代码块从普通的 `async` 代码块修改为 `async move` 代码块。当运行 这个 版本的代码时，它会在发送和接收完最后一条消息后优雅地关闭。

文件名：src/main.rs

```
        let (tx, mut rx) = trpl::channel();

        let tx_fut = async move {
            let vals = vec![
```

```
                String::from("hi"),
                String::from("from"),
                String::from("the"),
                String::from("future"),
            ];

            for val in vals {
                tx.send(val).unwrap();
                trpl::sleep(Duration::from_millis(500)).await;
            }
        };

        let rx_fut = async {
            while let Some(value) = rx.recv().await {
                println!("received '{value}'");
            }
        };

        trpl::join(tx_fut, rx_fut).await;
```

这个异步信道也是一个多生产者信道，所以如果希望从多个 future 发送消息可以调用 `tx` 上的 `clone` 方法。在示例 17-13 中，我们克隆了 `tx`，在第一个异步代码块外面创建 `tx1`。我们像第一个 `tx` 一样将 `tx1` 移动进代码块。接下来，将原始的 `tx` 移动进一个 新的 异步代码块，其中会用一个稍微更长的延迟发送更多的消息。我们碰巧将新代码块放在接收消息的异步代码块之后，不过也可以放在之前。关键在于 future 被 await 的顺序，而不是它们创建的顺序。

两个发送消息的异步代码块需要是 `async move` 代码块，如此 `tx` 和 `tx1` 都会在代码块结束后被丢弃。否则我们就会陷入到开始时同样的无限循环。最后，我们从 `trpl::join` 切换到 `trpl::join3` 来处理额外的 future。

文件名：src/main.rs

```
        let (tx, mut rx) = trpl::channel();

        let tx1 = tx.clone();
        let tx1_fut = async move {
            let vals = vec![
                String::from("hi"),
                String::from("from"),
                String::from("the"),
                String::from("future"),
            ];

            for val in vals {
                tx1.send(val).unwrap();
                trpl::sleep(Duration::from_millis(500)).await;
            }
        };

        let rx_fut = async {
            while let Some(value) = rx.recv().await {
                println!("received '{value}'");
            }
        };
```

```
        let tx_fut = async move {
            let vals = vec![
                String::from("more"),
                String::from("messages"),
                String::from("for"),
                String::from("you"),
            ];

            for val in vals {
                tx.send(val).unwrap();
                trpl::sleep(Duration::from_millis(1500)).await;
            }
        };

        trpl::join3(tx1_fut, tx_fut, rx_fut).await;
```

现在我们会看到所有来自两个发送 future 的消息。因为发送 future 采用了稍微不同的发送延迟，消息也会以这些不同的延迟接收。

```
received 'hi'
received 'more'
received 'from'
received 'the'
received 'messages'
received 'future'
received 'for'
received 'you'
```

这是一个良好的开始，不过它将我们限制到少数几个 future：`join` 两个，或者 `join3` 三个。让我们看下如何处理更多的 future。

使用任意数量的 futures

当我们在上一部分从使用两个 future 到三个 future 的时候，我们也必须从使用 `join` 切换到 `join3`。每次我们想要改变 join 的 future 数量时都不得不调用一个不同的函数是很烦人的。令人高兴的是，我们有一个宏版本的 `join` 可以传递任意数量的参数。它还会自行处理 await 这些 future。因此，我们可以重写示例 17-13 中的代码来使用 `join!` 而不是 `join3`，如示例 17-14 所示：

文件名：src/main.rs

```
        trpl::join!(tx1_fut, tx_fut, rx_fut);
```

相比于需要在 `join` 和 `join3` 和 `join4` 等等之间切换来说这绝对是一个进步！然而，即便是这个宏形式也只能用于我们提前知道 future 的数量的情况。不过，在现实世界的 Rust 中，将 futures 放进一个集合并接着等待集合中的一些或者全部 future 完成是一个常见的模式。

为了检查一些集合中的所有 future，我们需要遍历并 join 全部 的 future。`trpl::join_all` 函数接受任何实现了 `Iterator` trait 的类型，我们在之前的第十三章中学习过它们，所以这正是我们需要的。让我们将 futures 放进一个向量，并将 `join!` 替换为 `join_all`。

文件名：src/main.rs

```
        let futures = vec![tx1_fut, rx_fut, tx_fut];

        trpl::join_all(futures).await;
```



不幸的是这还不能编译。相反我们会得到这个错误：

```
error[E0308]: mismatched types
  --> src/main.rs:43:37
   |
8  |           let tx1_fut = async move {
   |  _________________________-
9  | |             let vals = vec![
10 | |                 String::from("hi"),
11 | |                 String::from("from"),
...  |
19 | |             }
20 | |         };
   | |_________- the expected `async` block
21 |
22 |           let rx_fut = async {
   |  _________________________-
23 | |             while let Some(value) = rx.recv().await {
24 | |                 println!("received '{value}'");
25 | |             }
26 | |         };
   | |_________- the found `async` block
...
43 |           let futures = vec![tx1_fut, rx_fut, tx_fut];
   |                                       ^^^^^^ expected `async` block, found a
different `async` block
   |
```

```
= note: expected `async` block `{async block@src/main.rs:8:23: 20:10}`
           found `async` block `{async block@src/main.rs:22:22: 26:10}`
= note: no two async blocks, even if identical, have the same type
= help: consider pinning your async block and and casting it to a trait object
```

这可能有点令人惊讶。毕竟没有一个 future 返回了任何值，所以每个代码块都会产生一个 `Future<Output = ()>`。然而，`Future` 是一个 trait，而不是一个具体类型。其具体类型是编译器为各个异步代码块生成的（不同的）数据结构。你不能将两个不同的手写的 struct 放进同一个 `Vec`，同样的原理也适用于编译器生成的不同 struct。

为了使代码能够正常工作，我们需要使用 *trait objects*，正如我们在第十二章的 "从 `run` 函数中返回错误" 中做的那样。（第十八章会详细介绍 trait objects。）使用 trait objects 允许我们将这些类型所产生的不同的匿名 future 视为相同的类型，因为它们都实现了 `Future` trait。

> 注意：在第八章中，我们讨论过另一种将多种类型包含进一个 `Vec` 的方式：使用一个枚举来代表每个可以出现在向量中的不同类型。不过这里我们不能这么做。一方面，没有方法来命名这些不同的类型，因为它们是匿名的。另一方面，我们最开始采用向量和 `join_all` 的原因是为了处理一个直到运行时之前都不知道是什么的 future 的动态集合。

我们以将 `vec!` 中的每个 future 用 `Box::new` 封装来作为开始，如示例 17-16 所示。

文件名：src/main.rs



```
        let futures =
            vec![Box::new(tx1_fut), Box::new(rx_fut), Box::new(tx_fut)];

        trpl::join_all(futures).await;
```

不幸的是，代码仍然不能编译。事实上，我们遇到了与之前相同的基本错误，不过这次我们会在第二个和第三个 `Box::new` 调用处各得到一个错误，同时还会得到一个提及 `Unpin` trait 的新错误。我们一会再回到 `Unpin` 错误上。首先，让我们通过显式标注 `futures` 的类型来修复 `Box::new` 调用的类型错误：

文件名：src/main.rs



```
        let futures: Vec<Box<dyn Future<Output = ()>>> =
            vec![Box::new(tx1_fut), Box::new(rx_fut), Box::new(tx_fut)];
```

这里必须编写的类型有一点复杂，让我们逐步过一遍：

- 最内层的类型是 future 本身。我们显式地指出 future 的输出类型是单元类型 `()`，其编写为 `Future<Output = ()>`。
- 接着使用 `dyn` 将 trait 标记为动态的。
- 整个 trait 引用被封装进一个 `Box`。
- 最后，我们显式表明 `futures` 是一个包含这些项的 `Vec`。

这已经有了很大的区别。现在当我们运行编译器时，就只会有提到 `Unpin` 的错误了。虽然这里有三个错误，但请注意它们每个的内容都非常相似。

```
error[E0277]: `{async block@src/main.rs:8:23: 20:10}` cannot be unpinned
   --> src/main.rs:46:24
    |
46  |         trpl::join_all(futures).await;
    |         -------------- ^^^^^^^ the trait `Unpin` is not implemented for
`{async block@src/main.rs:8:23: 20:10}`, which is required by `Box<{async
block@src/main.rs:8:23: 20:10}>: std::future::Future`
    |         |
    |         required by a bound introduced by this call
    |
    = note: consider using the `pin!` macro
            consider using `Box::pin` if you need to access the pinned value
outside of the current scope
    = note: required for `Box<{async block@src/main.rs:8:23: 20:10}>` to implement
`std::future::Future`
note: required by a bound in `join_all`
   --> /Users/chris/.cargo/registry/src/index.crates.io-6f17d22bba15001f/futures-
util-0.3.30/src/future/join_all.rs:105:14
    |
102 | pub fn join_all<I>(iter: I) -> JoinAll<I::Item>
    |        -------- required by a bound in this function
...
105 |     I::Item: Future,
    |              ^^^^^^ required by this bound in `join_all`

error[E0277]: `{async block@src/main.rs:8:23: 20:10}` cannot be unpinned
  --> src/main.rs:46:9
   |
46 |         trpl::join_all(futures).await;
   |         ^^^^^^^^^^^^^^^^^^^^^^^ the trait `Unpin` is not implemented for
`{async block@src/main.rs:8:23: 20:10}`, which is required by `Box<{async
block@src/main.rs:8:23: 20:10}>: std::future::Future`
   |
   = note: consider using the `pin!` macro
           consider using `Box::pin` if you need to access the pinned value
outside of the current scope
   = note: required for `Box<{async block@src/main.rs:8:23: 20:10}>` to implement
`std::future::Future`
note: required by a bound in `JoinAll`
  --> /Users/chris/.cargo/registry/src/index.crates.io-6f17d22bba15001f/futures-
util-0.3.30/src/future/join_all.rs:29:8
   |
27 | pub struct JoinAll<F>
   |            ------- required by a bound in this struct
28 | where
29 |     F: Future,
   |        ^^^^^^ required by this bound in `JoinAll`

error[E0277]: `{async block@src/main.rs:8:23: 20:10}` cannot be unpinned
  --> src/main.rs:46:33
   |
46 |         trpl::join_all(futures).await;
   |                                 ^^^^^ the trait `Unpin` is not implemented
for `{async block@src/main.rs:8:23: 20:10}`, which is required by `Box<{async
```

```
block@src/main.rs:8:23: 20:10}>: std::future::Future`
   |
   = note: consider using the `pin!` macro
           consider using `Box::pin` if you need to access the pinned value
outside of the current scope
   = note: required for `Box<{async block@src/main.rs:8:23: 20:10}>` to implement
`std::future::Future`
note: required by a bound in `JoinAll`
  --> /Users/chris/.cargo/registry/src/index.crates.io-6f17d22bba15001f/futures-
util-0.3.30/src/future/join_all.rs:29:8
   |
27 | pub struct JoinAll<F>
   |            ------- required by a bound in this struct
28 | where
29 |     F: Future,
   |        ^^^^^^ required by this bound in `JoinAll`

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
```

这里有 很多 内容需要分析，所以让我们拆开来看。信息的第一部分告诉我们第一个异步代码块（`src/main.rs:8:23: 20:10`）没有实现 `Unpin` trait，并建议使用 `pin!` 或 `Box::pin` 来修复，在本章的稍后部分我们会深入 `Pin` 和 `Unpin` 的一些更多细节。不过现在我们可以仅仅遵循编译器的建议来解困！在示例 17-18 中，我们以更新 `futures` 的类型声明作为开始，用 `Pin` 来封装每个 `Box`。其次，我们使用 `Box::pin` 来 pin 住 futures 自身。

文件名：src/main.rs

```
use std::pin::Pin;

// -- snip --

        let futures: Vec<Pin<Box<dyn Future<Output = ()>>>> =
            vec![Box::pin(tx1_fut), Box::pin(rx_fut), Box::pin(tx_fut)];
```

如果编译并运行代码，我们终于会得到我们期望的输出：

```
received 'hi'
received 'more'
received 'from'
received 'messages'
received 'the'
received 'for'
received 'future'
received 'you'
```

（长舒一口气!）

这里还有一些我们可以进一步探索的内容。首先，因为通过 `Box` 来将这些 futures 放到堆上，使用 `Pin<Box<T>>` 会带来少量的额外开销，而我们这么做仅仅是为了使类型对齐。毕竟这里实际上并不 需要 堆分配：这些 futures 对于这个特定的函数来说是本地的。如上所述，`Pin` 本身是一个封装类型，因此我们可以在 `Vec` 中拥有单一类型的好处（也就是使用 `Box` 的初始原因）而不用堆分配。我们可以通过 `std::pin::pin` 宏来直接对每个 future 使用 `Pin`。

然而，我们仍然必须显式地知道被 pin 的引用的类型：否则 Rust 仍然不知道如何将它们解释为动态 trait objects，这是将它们放进 `Vec` 所需的。因此我们在定义每个 future 的时候使用 `pin!`，并将 `futures` 定义为一个包含被 pin 的动态 `Future` 类型的可变引用的 `Vec`，如示例 17-19 所示。

文件名：src/main.rs

```
use std::pin::{Pin, pin};

// -- snip --

        let tx1_fut = pin!(async move {
            // --snip--
        });

        let rx_fut = pin!(async {
            // --snip--
        });

        let tx_fut = pin!(async move {
            // --snip--
        });

        let futures: Vec<Pin<&mut dyn Future<Output = ()>>> =
            vec![tx1_fut, rx_fut, tx_fut];
```

目前为止我们一直忽略了可能有不同 `Output` 类型的事实。例如，在示例 17-20 中，匿名 future `a` 实现了 `Future<Output = u32>`，匿名 future `b` 实现了 `Future<Output = &str>`，而匿名 future `c` 实现了 `Future<Output = bool>`。

文件名：src/main.rs

```
        let a = async { 1u32 };
        let b = async { "Hello!" };
        let c = async { true };

        let (a_result, b_result, c_result) = trpl::join!(a, b, c);
        println!("{a_result}, {b_result}, {c_result}");
```

我们可以使用 `trpl::join!` 来 await 它们，因为它允许你传递多个 future 类型并产生一个这些类型的元组。我们 不能 使用 `trpl::join_all`，因为它要求传递的 future 都拥有相同的类型。请记住，那个错误正是我们开启 `Pin` 探索之旅的原因！

这是一个基础的权衡取舍：要么我们可以使用 `join_all` 处理动态数量的 future，只要它们都有相同的类型；要么我们可以使用 `join` 函数或者 `join!` 宏来处理固定数量的 future，哪怕它们有着不同的类型。不过这与 Rust 处理任何其它类型是一样的。Future 并不特殊，即便我们采用了一些友好的语法来处理它们，而这其实是好事。

future 竞争

当我们使用 `join` 系列函数和宏来 “join” future 时，我们要求它们 全部 结束才能继续。虽然有时我们只需要 部分 future 结束就能继续，这有点像一个 future 与另一个 future 竞争。

在示例 17-21 中，我们再次使用 `trpl::race` 来运行 `slow` 和 `fast` 两个 future 并相互竞争。它们每一个都会在开始运行时打印一条消息，并通过调用 await `sleep` 暂停一段时间，接着在其结束时打印另一条消息。然后我们将它们传递给 `trpl::race` 并等待其中一个结束。（结果不会令人意外：`fast` 会赢！）不同于我们在第一个异步程序中使用 `race` 的时候，这里忽略了其返回的 `Either` 实例，因为所有有趣的行为都发生在异步代码块中。

文件名：src/main.rs

```
        let slow = async {
            println!("'slow' started.");
            trpl::sleep(Duration::from_millis(100)).await;
            println!("'slow' finished.");
        };

        let fast = async {
            println!("'fast' started.");
            trpl::sleep(Duration::from_millis(50)).await;
            println!("'fast' finished.");
        };

        trpl::race(slow, fast).await;
```

请注意如果你反转 `race` 参数的顺序，“started” 消息的顺序会改变，即使 `fast` future 总是第一个结束。这是因为这个特定的 `race` 函数实现并不是公平的。它总是以传递的参数的顺序来运行传递的 futures。其它的实现 是 公平的，并且会随机选择首先轮询的 future。不过无论我们使用的 race 实现是否公平，其中 一个 future 会在另一个任务开始之前一直运行到异步代码块中第一个 `await` 为止。

回忆一下第一个异步程序中提到在每一个 await point，如果被 await 的 future 还没有就绪，Rust 会给运行时一个机会来暂停该任务并切换到另一个任务。反过来也是正确的：Rust 只会在一个 await point 暂停异步代码块并将控制权交还给运行时。await points 之间的一切都是同步的。

这意味着如果你在异步代码块中做了一堆工作而没有一个 await point，则那个 future 会阻塞其它任何 future 继续进行。有时你可能会听说这称为一个 future 导致其它 future 饥饿（starving）。在一些情况中，这可能不是什么大问题。不过，如果你在进行某种昂贵的设置或者长时间运行的任务，亦或有一个 future 会无限持续运行某些特定任务的话，你会需要思考在何时何地将控制权交还运行时。

同样地，如果你有长时间运行的阻塞操作，异步可能是一个提供了将程序的不同部分相互关联起来的实用工具。

不过在这种情况下 如何 将控制权交还运行时呢?

Yielding

让我们模拟一个长时间运行的操作。示例 17-22 引入了一个 `slow` 函数。它使用 `std::thread::sleep` 而不是 `trpl::sleep` 因此 `slow` 调用会阻塞当前线程若干毫秒。我们可以用 `slow` 来代表现实世界中的长时间运行并且会阻塞的操作。

文件名：src/main.rs

```
fn slow(name: &str, ms: u64) {
    thread::sleep(Duration::from_millis(ms));
    println!("'{name}' ran for {ms}ms");
}
```

在示例 17-22 中，我们使用 `slow` 在几个 future 中模拟这类 CPU 密集型工作。首先，每个 future 只会在进行了一系列缓慢操作 之后 才将控制权交还给运行时。

文件名：src/main.rs

```
        let a = async {
            println!("'a' started.");
            slow("a", 30);
            slow("a", 10);
            slow("a", 20);
            trpl::sleep(Duration::from_millis(50)).await;
            println!("'a' finished.");
        };

        let b = async {
            println!("'b' started.");
            slow("b", 75);
            slow("b", 10);
            slow("b", 15);
            slow("b", 350);
            trpl::sleep(Duration::from_millis(50)).await;
            println!("'b' finished.");
        };

        trpl::race(a, b).await;
```

如果运行代码，你会看到这些输出：

```
'a' started.
'a' ran for 30ms
'a' ran for 10ms
'a' ran for 20ms
'b' started.
'b' ran for 75ms
'b' ran for 10ms
'b' ran for 15ms
'b' ran for 350ms
'a' finished.
```

与上一个示例一样，`race` 仍然在 `a` 完成后就立刻结束了。两个 future 之间没有交替运行。`a` future 一直进行其工作直到 `trpl::sleep` 调用被 await，然后 `b` future 一直进行其工作直到它自己的 `trpl::sleep` 调用被 await，再然后 `a` future 才完成。为了使两个 future 在各自缓慢任务之间都能有所进展，我们需要 await point 才能将控制权交还给运行时。这意味着我们需要一些可以 await 的东西！

我们已经在示例 17-23 中见过这类交接发生：如果去掉 `a` future 结尾的 `trpl::sleep`，那么当它完成时 `b` future 完全 不会运行。也许我们可以使用 `sleep` 函数来作为开始呢？

文件名：src/main.rs

```
        let one_ms = Duration::from_millis(1);

        let a = async {
            println!("'a' started.");
            slow("a", 30);
            trpl::sleep(one_ms).await;
            slow("a", 10);
            trpl::sleep(one_ms).await;
            slow("a", 20);
            trpl::sleep(one_ms).await;
            println!("'a' finished.");
        };

        let b = async {
            println!("'b' started.");
            slow("b", 75);
            trpl::sleep(one_ms).await;
            slow("b", 10);
            trpl::sleep(one_ms).await;
            slow("b", 15);
            trpl::sleep(one_ms).await;
            slow("b", 350);
            trpl::sleep(one_ms).await;
            println!("'b' finished.");
        };
```

在示例 17-24 中，我们在 `slow` 调用之间增加了 `trpl::sleep` 调用和 await points。现在两个 future 的工作会相互交替运行：

```
'a' started.
'a' ran for 30ms
'b' started.
'b' ran for 75ms
'a' ran for 10ms
'b' ran for 10ms
'a' ran for 20ms
'b' ran for 15ms
'a' finished.
```

`a` future 仍然会在交还控制权给 `b` 之前运行一会儿，因为它在调用 `trpl::sleep` 之前就调用了 `slow`，不过在这之后两个 future 会在触发 await point 时来回切换。在这个例子中，我们在 `slow` 之后这么做，不过我们可以在任何合适的地方拆分任务。

不过我们并不是真的想在这里 休眠：我们希望尽可能快地取得进展。我们仅仅是需要交还控制权给运行时。我们可以使用 `yield_now` 函数来直接这么做。在示例 17-25 中，我们将所有的 `sleep` 调用替换为 `yield_now`。

文件名：src/main.rs

```
        let a = async {
            println!("'a' started.");
            slow("a", 30);
            trpl::yield_now().await;
            slow("a", 10);
```

```
            trpl::yield_now().await;
            slow("a", 20);
            trpl::yield_now().await;
            println!("'a' finished.");
        };

        let b = async {
            println!("'b' started.");
            slow("b", 75);
            trpl::yield_now().await;
            slow("b", 10);
            trpl::yield_now().await;
            slow("b", 15);
            trpl::yield_now().await;
            slow("b", 350);
            trpl::yield_now().await;
            println!("'b' finished.");
        };
```

这不仅更为清楚地表明了实际的意图而且更显著地快于使用 `sleep`，因为像这样使用 `sleep` 的定时器通常受限于其控制粒度。例如我们使用的 `sleep` 版本，会至少休眠一毫秒，哪怕我们传递一纳秒的 `Duration`。而且，现代计算机非常 快速：它们可以在一毫秒内完成很多工作！

你可以自行设置一些基准测试来验证这一点，例如示例 17-26 中的这个。（这并不是一个特别严谨的进行性能测试的方法，不过用来展示这里的区别是足够的。）这里，我们省略了所有的状态打印，传递一纳秒的 `Duration` 给 `trpl::sleep`，并让每一个 future 各自运行，不在 future 之间切换。接着我们运行 999 次迭代并对比下使用 `trpl::sleep` 的 future 和使用 `trpl::yield_now` 的 future 的运行时间。

文件名：src/main.rs

```
        let one_ns = Duration::from_nanos(1);
        let start = Instant::now();
        async {
            for _ in 1..1000 {
                trpl::sleep(one_ns).await;
            }
        }
        .await;
        let time = Instant::now() - start;
        println!(
            "'sleep' version finished after {} seconds.",
            time.as_secs_f32()
        );

        let start = Instant::now();
        async {
            for _ in 1..1000 {
                trpl::yield_now().await;
            }
        }
        .await;
        let time = Instant::now() - start;
        println!(
```

```
            "'yield' version finished after {} seconds.",
            time.as_secs_f32()
        );
```

使用 `yield_now` 的版本要 快得多！

这意味着取决于程序所作的其它工作，异步操作甚至在计算密集型任务中也有用处，因为它提供了一个结构化程序中不同部分之间关系的实用工具。这是一种形式的 协同多任务处理（*cooperative multitasking*），每个 future 有权通过 await point 来决定何时交还控制权。因此每个 future 也有责任避免长时间阻塞。在一些基于 Rust 的嵌入式系统中，这是 唯一 的多任务处理类型！

当然，在真实代码中，你通常不会在每一行上都交替使用 await 点来调用函数。虽然这样控制 yielding 相对来说更为廉价，但也不是毫无代价的！在很多情况下，尝试将计算密集型任务拆分可能会显著降低其速度，所以有时为了 整体 性能简单地让一个操作阻塞是更好的选择。你应该总是通过测量来观察代码真正的性能瓶颈是什么。不过其底层的考量在于重要的是要牢记你是否 确实 观察到了很多期望并发进行的工作在串行地进行。

构建我们自己的异步抽象

我们也可以将 futures 组合起来形成一个新模式。例如，我们可以使用已有的异步代码块构建一个 `timeout` 函数。当我们完成时，其结果将是另一个可以用来构建进一步异步抽象的代码块。

示例 17-27 展示了我们预期 `timeout` 如何处理一个缓慢运行的 future。

文件名：src/main.rs

```
        let slow = async {
            trpl::sleep(Duration::from_millis(100)).await;
            "I finished!"
        };

        match timeout(slow, Duration::from_millis(10)).await {
            Ok(message) => println!("Succeeded with '{message}'"),
            Err(duration) => {
                println!("Failed after {} seconds", duration.as_secs())
            }
        }
```

让我们来实现它！首先，让我们考虑一下 `timeout` 的 API：

- 它需要是一个 async 函数以便可以 await。
- 它的第一个参数应该是需要运行的 future。我们可以使用泛型以便可以处理任意 future。
- 它的第二个参数将是需要等待的最大时间。如果我们使用 `Duration` 的话，将会使得将其直接传递给 `trpl::sleep` 变得简单。
- 它应该返回一个 `Result`。如果 `future` 成功完成，`Result` 将会是包含 `future` 所产生的值的 `Ok`。如果超时先发生，`Result` 将会是包含超时等待的持续时间的 `Err`。

示例 17-28 展示了这个抽象。

文件名：src/main.rs

```
async fn timeout<F: Future>(
    future_to_try: F,
    max_time: Duration,
) -> Result<F::Output, Duration> {
    // Here is where our implementation will go!
}
```

这满足了我们对类型的目标。现在让我们思考下所需的 行为：我们需要传递进来的 future 在持续时间内相互竞争。我们可以使用 `trpl::sleep` 和 duration 来创建一个定时器 future，并使用 `trpl::race` 来运行定时器 future 和调用者传递进来的 future。

我们还知道 `race` 是不公平的，并按照传递的顺序轮询参数。因此，我们首先传递 `future_to_try` 给 `race` 以便哪怕 `max_time` 是一个非常短的持续时间它也能有机会完成。如果 `future_to_try` 首先完成，`race` 会返回 `Left` 和 `future` 的输出。如果 `timer` 首先完成，`race` 会返回 `Right` 和定时器的输出 `()`。

在示例 17-29 中，我们匹配 await `trpl::race` 的结果。如果 `future_to_try` 成功并得到一个 `Left(output)`，我们返回 `Ok(output)`。相反如果休眠定时器超时了并得到一个 `Right(())`，则我们通过 `_` 忽略 `()` 并返回 `Err(max_time)`。

文件名：src/main.rs

```
use trpl::Either;

// --snip--

fn main() {
    trpl::run(async {
        let slow = async {
            trpl::sleep(Duration::from_secs(5)).await;
            "Finally finished"
        };

        match timeout(slow, Duration::from_secs(2)).await {
            Ok(message) => println!("Succeeded with '{message}'"),
            Err(duration) => {
                println!("Failed after {} seconds", duration.as_secs())
            }
        }
    });
}

async fn timeout<F: Future>(
    future_to_try: F,
    max_time: Duration,
) -> Result<F::Output, Duration> {
    match trpl::race(future_to_try, trpl::sleep(max_time)).await {
        Either::Left(output) => Ok(output),
        Either::Right(_) => Err(max_time),
    }
```

于是我们有了一个由另外两个帮助函数构成的可以工作的 `timeout`。如果我们运行代码，它会在超时之后打印失败模式：

```
Failed after 2 seconds
```

由于 future 可以和其他 future 组合，你可以使用更小的异步代码块来构建非常强力的工具。例如，可以使用相同的方式来组合超时和重试，并转而将其用于类似网络调用的工作，这正是本章开头的一个示例！

在实践中，你会直接处理 `async` 和 `await`，其次才是类似 `join`、`join_all`、`race` 等函数和宏，在使用这些 API 时你只会偶尔遇到 `pin`。

现在我们见过了一系列同时处理多个 future 的方法了。接下来，我们来看看如何通过（流）*streams* 处理一个时间序列的多个 future。不过，在此之前，这里有几个你可能想要先考虑的问题：

- 我们在 `Vec` 上使用了 `join_all` 来等待一组中的所有 future 完成。相反该如何使用 `Vec` 来依次处理一个序列的 future 呢？这么做有哪些权衡取舍呢？
- 仔细观察 `futures` crate 中的 `futures::stream::FuturesUnordered` 类型。使用它与使用 `Vec` 又有什么区别呢？（不用担心它来自与 crate 的 `stream` 部分的事实；它刚好能处理任何 future 的集合。）

流（Streams）：顺序的 Futrues

到本章的目前为止，我们大部分时间都专注于单个的 future 上。一个重要的例外就是我们用过的异步信道。回忆一下在本章之前的 “消息传递” 中我们如何使用异步信道接收端的。异步 `recv` 方法随着时间的推移产生一个序列的项。这是一个更通用的模式的实例，通常被称为 流（*stream*）。

我们之前在第十三章的 Iterator trait 和 `next` 方法 部分已经见过项的序列，不过迭代器和异步信道接收端有两个区别。第一个区别是时间维度：迭代器是同步的，而信道接收端是异步的。第二个区别是 API。当直接处理 `Iterator` 时，我们会调用其同步 `next` 方法。对于这个特定的 `trpl::Receiver` 流，我们调用一个异步的 `recv` 方法。除此之外，这两种 API 在使用上感觉十分相似，这种相似性并非巧合。流类似于一种异步形式的迭代器。不过鉴于 `trpl::Receiver` 专门等待接收消息，多用途的流 API 则更为通用：它像 `Iterator` 一样提供了下一个项，但采用异步的方式。

Rust 中迭代器和流的相似性意味着我们实际上可以从任何迭代器上创建流。就迭代器而言，可以通过调用其 `next` 方法并 await 输出来使用流，如示例 17-30 所示。

文件名：src/main.rs



```
        let values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        let iter = values.iter().map(|n| n * 2);
        let mut stream = trpl::stream_from_iter(iter);

        while let Some(value) = stream.next().await {
            println!("The value was: {value}");
        }
```

我们以一组数字作为开始，将其转换为一个迭代器并接着调用 `map` 将其所有值翻倍。然后使用 `trpl::stream_from_iter` 函数将迭代器转换为流。随后，我们使用 `while let` 循环在项到达时对流中的每个项进行循环处理。

遗憾的是当我们尝试运行代码时，代码无法编译，而是报告没有可用的 `next` 方法。

```
error[E0599]: no method named `next` found for struct `Iter` in the current scope
  --> src/main.rs:10:40
   |
10 |         while let Some(value) = stream.next().await {
   |                                        ^^^^
   |
   = note: the full type name has been written to '/Users/chris/dev/rust-lang/
book/main/listings/ch17-async-await/listing-17-30/target/debug/deps/
async_await-575db3dd3197d257.long-type-14490787947592691573.txt'
   = note: consider using `--verbose` to print the full type name to the console
   = help: items from traits can only be used if the trait is in scope
help: the following traits which provide `next` are implemented but not in scope;
perhaps you want to import one of them
   |
1  + use crate::trpl::StreamExt;
   |
1  + use futures_util::stream::stream::StreamExt;
   |
1  + use std::iter::Iterator;
```

```
   |
1  + use std::str::pattern::Searcher;
   |
help: there is a method `try_next` with a similar name
   |
10 |         while let Some(value) = stream.try_next().await {
   |                                        ~~~~~~~~
```

正如输出中所建议的，编译器错误的原因是我们需要在作用域中有正确的 trait 以便能够使用 `next` 方法。鉴于目前为止的讨论，你可能会合理地推测是 `Stream`，但实际上需要的是 `StreamExt`。这里的 `Ext` 是 “extension”：在 Rust 社区中这是用另一个 trait 扩展 trait 的常见模式。

我们稍后会在本章末尾更详细地介绍 `Stream` 和 `StreamExt` trait，目前你只需知道 `Stream` trait 定义了一个底层接口用于有效地组合 `Iterator` 和 `Future` trait。

为什么我们需要 `StreamExt` 而不是 `Stream`，而 `Stream` trait 本身又是做什么的呢？简单来说，答案是贯穿整个 Rust 生态系统，`Stream` trait 定义了一个底层接口用于有效地结合 `Iterator` 与 `Future` trait。`StreamExt` trait 在 `Stream` 之上提供了一组高层 API，其中包括了 `next` 和其它类似于 `Iterator` trait 提供的工具方法。`Stream` 和 `StreamExt` 目前尚未被纳入 Rust 的标准库，但大多数生态系统 crate 都使用相同的定义。

对编译器错误的修复是增加一个 `trpl::StreamExt` 的 `use` 语句，如示例 17-31 所示。

文件名：src/main.rs

```rust
use trpl::StreamExt;

fn main() {
    trpl::run(async {
        let values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        let iter = values.iter().map(|n| n * 2);
        let mut stream = trpl::stream_from_iter(iter);

        while let Some(value) = stream.next().await {
            println!("The value was: {value}");
        }
    });
}
```

将所有这些代码片段拼凑在一起，这段代码如我们预期般运行！更重要的是，现在我们将 `StreamExt` 引入了作用域，就可以像使用迭代器一样使用它的所有工具方法。例如在示例 17-32 中，我们使用 `filter` 方法来过滤出仅为 3 或 5 的倍数的项。

文件名：src/main.rs

```rust
use trpl::StreamExt;

fn main() {
    trpl::run(async {
        let values = 1..101;
        let iter = values.map(|n| n * 2);
        let stream = trpl::stream_from_iter(iter);
```

```
        let mut filtered =
            stream.filter(|value| value % 3 == 0 || value % 5 == 0);

        while let Some(value) = filtered.next().await {
            println!("The value was: {value}");
        }
    });
}
```

当然这并不是非常的有趣。我们完全可以使用普通的迭代器而不用任何异步操作来做到这些。所以让我们看看流能实现的一些独特功能。

组合流

很多概念天然地适合用流来表示：队列中陆续可用的项、数据量超过计算机内存限制时逐步从文件系统拉取的数据块，或者随时间推移通过网络逐渐到达的数据。因为流本身也是 future，我们也可以将其用于任何其它类型的 future，并以一些非常有趣的方式组合它们。例如，我们可以批量处理事件来避免触发过多的网络调用，为一系列的长时间运行的任务设置超时，或者对用户接口事件限速来避免进行不必要的工作。

让我们构建一个小的消息流作为开始，将其作为一个可能从 WebSocket 或者其它现实世界中的通信协议中遇到的数据流的替代，如示例 17-33 所示。

在示例 17-33 中，作为其实现，我们创建了一个异步信道，循环英文字母表的前十个字符，并通过信道发送它们。

文件名：src/main.rs

```
use trpl::{ReceiverStream, Stream, StreamExt};

fn main() {
    trpl::run(async {
        let mut messages = get_messages();

        while let Some(message) = messages.next().await {
            println!("{message}");
        }
    });
}

fn get_messages() -> impl Stream<Item = String> {
    let (tx, rx) = trpl::channel();

    let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];
    for message in messages {
        tx.send(format!("Message: '{message}'")).unwrap();
    }

    ReceiverStream::new(rx)
}
```

首先，我们创建了一个返回 `impl Stream<Item = String>` 的 `get_messages` 函数。作为其实现，我们创建了一个异步信道，循环遍历英文字母表的前 10 个字母，并通过信道发送它们。

我们还使用了一个新类型：`ReceiverStream`，它将 `trpl::channel` 的 `rx` 接收端转换为一个带有 `next` 方法的 `Stream`。回到 `main`，我们使用了一个 `while let` 循环来打印来自流中的所有消息。

运行这段代码时，我们将得到与预期完全一致的结果：

```
Message: 'a'
Message: 'b'
Message: 'c'
Message: 'd'
Message: 'e'
Message: 'f'
Message: 'g'
Message: 'h'
Message: 'i'
Message: 'j'
```

不过，我们可以使用常规的 `Receiver` API 甚至是 `Iterator` API 来做到这些，所以让我们增加一个需要流的功能：增加一个适用于流中所有项的超时，和一个发送项的延时，如示例 17-34 所示。

文件名：src/main.rs

```
use std::{pin::pin, time::Duration};
use trpl::{ReceiverStream, Stream, StreamExt};

fn main() {
    trpl::run(async {
        let mut messages =
            pin!(get_messages().timeout(Duration::from_millis(200)));

        while let Some(result) = messages.next().await {
            match result {
                Ok(message) => println!("{message}"),
                Err(reason) => eprintln!("Problem: {reason:?}"),
            }
        }
    })
}
```

我们通过 `timeout` 方法在流上增加超时来作为开始，它来自 `StreamExt` trait。接着我们更新 `while let` 循环体，因为现在流返回一个 `Result`。`Ok` 变体表明消息及时到达；`Err` 变体表明任何消息到达前就触发超时了。我们 `match` 其结果要么在成功接收时打印消息要么打印一个超时的提示。最后，请注意我们在加上超时之后 pin 住了这些消息，因为超时辅助函数产生了一个需要 pin 住才能轮询的流。

然后，因为消息之间没有延时，超时并不会改变程序的行为。让我们为发送的消息增加一个延时变量，如示例 17-35 所示。

文件名：src/main.rs

```
fn get_messages() -> impl Stream<Item = String> {
    let (tx, rx) = trpl::channel();
```

```
    trpl::spawn_task(async move {
        let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];
        for (index, message) in messages.into_iter().enumerate() {
            let time_to_sleep = if index % 2 == 0 { 100 } else { 300 };
            trpl::sleep(Duration::from_millis(time_to_sleep)).await;

            tx.send(format!("Message: '{message}'")).unwrap();
        }
    });

    ReceiverStream::new(rx)
}
```

在 `get_messages` 中，我们在 `messages` 数组上使用 `enumerate` 迭代器方法以便能够同时获得项本身和其索引。然后我们为偶数索引的项引入 100 毫秒的延时并为奇数索引的项引入 300 毫秒的延时来模拟真实世界的消息流中可能出现的不同的延时。因为我们的延时为 200 毫秒，这应该会影响到其中一半的消息。

为了在 `get_messages` 函数中实现消息间的延迟且不造成阻塞，我们需要使用异步。然而，我们不能将 `get_messages` 函数本身变为异步函数，因为这样它会返回一个 `Future<Output = Stream<Item = String>>` 而不是 `Stream<Item = String>`。调用者则不得不 await `get_messages` 本身来获取流。不过请记住：在一个给定的 future 中的一切都是线性发生的；并发发生在 futures **之间**。await `get_messages` 会要求其在返回接收端流之前发送所有的消息，包括消息之间的休眠延时。其结果是，超时将毫无用处。流本身没有任何的延时；它们甚至全都发生在流可用之前。

相反，我们保持 `get_messages` 为一个返回流的常规函数，并 spawn 一个任务来处理异步 `sleep` 调用。

> 注意：像这样调用 `spawn_task` 可以工作是因为我们已经设置了运行时；如果没有，则会造成 panic。其它的实现则选择了不同的权衡策略：它们可能会产生一个新的运行时来避免 panic 不过最终会有一些额外开销，有的则可能根本不提供一种独立的、脱离运行时引用的方式来 spawn 任务。请务必理解你的运行时所选择的权衡策略来编写相应的代码！

现在我们的代码有了一个更为有趣的结果。每隔一对消息会有一个 `Problem: Elapsed(())` 错误。

```
Message: 'a'
Problem: Elapsed(())
Message: 'b'
Message: 'c'
Problem: Elapsed(())
Message: 'd'
Message: 'e'
Problem: Elapsed(())
Message: 'f'
Message: 'g'
```

```
Problem: Elapsed(())
Message: 'h'
Message: 'i'
Problem: Elapsed(())
Message: 'j'
```

超时最终并不会阻止消息到达。我们仍然能够得到所有原始的消息，因为我们的信道是 **无限的**（**unbounded**）：它可以存储内存所允许的所有消息。如果消息在超时之前没有到达，流处理器会做出相应处理，不过当再次轮询流时，消息现在可能已经到达了。

如果需要的话可以通过使用不同的信道或者其他更通用的流来实现不同行为。让我们看一个实际的将一个表示时间间隔的流和这个消息流合并的例子。

合并流

首先，让我们创建另一个流，如果直接运行它的话它会每毫秒发送一个项。为了简单起见，我们可以使用 `sleep` 函数来延迟发送一个消息并采用与 `get_messages` 函数中从信道创建流时相同的方式来合并它们。区别是这一次，我们将发送已经过去的间隔次数，所以返回值类型将会是 `impl Stream<Item = u32>`，函数可以命名为 `get_intervals`（如示例 17-36 所示）。

文件名：src/main.rs

```
fn get_intervals() -> impl Stream<Item = u32> {
    let (tx, rx) = trpl::channel();

    trpl::spawn_task(async move {
        let mut count = 0;
        loop {
            trpl::sleep(Duration::from_millis(1)).await;
            count += 1;
            tx.send(count).unwrap();
        }
    });

    ReceiverStream::new(rx)
}
```

我们以在任务中定义一个 `count` 作为开始。（我们也可以在任务外面定义它，不过限定任何变量的作用域会更明确。）接着我们创建一个无限循环。循环的每一次迭代会异步休眠一毫秒，递增计数器，并接着通过信道发送该值。因为这些全都封装在 `spawn_task` 创建的任务中，因此它们（包括无限循环）都会随着运行时的销毁而被清理。

这类在运行时被回收时才会结束的无限循环，在异步 Rust 中相当常见：很多程序需要无限地运行下去。通过异步编程，这不会阻塞任何其它内容，只要循环的每次迭代中有至少一个 await point。

现在回到 main 函数的异步代码块，我们可以尝试合并 `messages` 和 `intervals` 流，如示例 17-37 所示。

文件名：src/main.rs

```
```

```
        let messages = get_messages().timeout(Duration::from_millis(200));
        let intervals = get_intervals();
        let merged = messages.merge(intervals);
```



我们以调用 `get_intervals` 作为开始。接着通过 `merge` 方法合并 `messages` 和 `intervals` 流，它将多个流合并为一个从任何一个来源流的项可用时返回项的流，并且不会保持任何特定顺序。最后循环遍历合并后的流而不是 `messages`。

此时，`messages` 和 `intervals` 都不需要被 pin 住或是可变的，因为它们都会被合并进一个单一的 `merged` 流。然而，这个 `merge` 调用并不能编译！（`while let` 循环中的 `next` 调用也不行，稍后我们会回到这里。）这是因为两个流有着不同的类型。`messages` 流有着 `Timeout<impl Stream<Item = String>>` 类型，其中 `Timeout` 是在调用 `timeout` 时实现了 `Stream` 的类型。`intervals` 有着 `impl Stream<Item = u32>` 类型。为了合并这两个类型，我们需要将其中一个流转换以适配另一个流。我们将重构 `intervals` 流，因为 `messages` 流已经有了我们期望的基本形态而且我们必须处理超时错误（如示例 17-38 所示）。

文件名：src/main.rs

```
        let messages = get_messages().timeout(Duration::from_millis(200));
        let intervals = get_intervals()
            .map(|count| format!("Interval: {count}"))
            .timeout(Duration::from_secs(10));
        let merged = messages.merge(intervals);
        let mut stream = pin!(merged);
```

首先，我们可以使用 `map` 辅助方法将 `intervals` 转换为字符串。再次，我们需要匹配 `messages` 中的 `Timeout`。但是因为我们不 **希望** `intervals` 有超时，因此可以直接创建一个比其他超时时长更长的超时。这里通过 `Duration::from_secs(10)` 创建了一个十秒的超时。最后我们需要将 `stream` 变为可变，这样 `while let` 循环的 `next` 调用可以遍历流，并且需要 pin 住它才能安全地执行。这 **几乎** 到了我们需要的地方。每一个类型都检查正确了。但是，如果你运行它，这会有两个问题。第一，它永远也不会停止！你需要使用 ctrl-c 来停止它。第二，来自英文字母表的消息会淹没在所有的间隔计数消息之中：

```
--snip--
Interval: 38
Interval: 39
Interval: 40
Message: 'a'
Interval: 41
Interval: 42
Interval: 43
--snip--
```

示例 17-39 展示了一种解决最后两个问题的方法。

文件名：src/main.rs

```
        let messages = get_messages().timeout(Duration::from_millis(200));
        let intervals = get_intervals()
            .map(|count| format!("Interval: {count}"))
```

```
            .throttle(Duration::from_millis(100))
            .timeout(Duration::from_secs(10));
        let merged = messages.merge(intervals).take(20);
        let mut stream = pin!(merged);
```

首先，我们在 `intervals` 流上使用 `throttle` 方法以便其不会淹没 `messages`。**限流**（**Throttling**）是一种限制函数被调用速率的方式，或者在本例中是限制流被轮询的频率。每 100 毫秒一次较为合适。因为这大概是消息到达的间隔。

为了限制我们从流接收的项的数量，可以在 `merged` 流上调用 `take` 方法，因为我们希望限制最终输出，而不仅仅是两个流中的某一个。

现在当我们运行程序时，它在从流中轮询 20 个项后停止，同时间隔不会淹没消息。我们也不会看到 `Interval: 100` 或 `Interval: 200` 等信息，而是 `Interval: 1`、`Interval: 2` 等等，即便来源流**可以**每毫秒产生一个事件。这是因为 `throttle` 调用产生了一个封装了原始流的新流，这样原始流只会在限制速率下而不是其 “原生” 速率下轮询。我们不会有大量未处理的间隔消息来选择性地丢弃，我们最开始就从未产生这些间隔消息！这又是 Rust 的 future 所固有的 “惰性” 在起作用，它允许我们自主选择程序的性能特点。

```
Interval: 1
Message: 'a'
Interval: 2
Interval: 3
Problem: Elapsed(())
Interval: 4
Message: 'b'
Interval: 5
Message: 'c'
Interval: 6
Interval: 7
Problem: Elapsed(())
Interval: 8
Message: 'd'
Interval: 9
Message: 'e'
Interval: 10
Interval: 11
Problem: Elapsed(())
Interval: 12
```

还有最后一个需要处理的问题：错误！有了这两个基于信道的流，当信道的另一端关闭时 `send` 方法可能会失败，这取决于运行时如何执行组成流的 future。直到现在为止，我们通过 `unwrap` 调用忽略了这种可能性。但在一个行为良好的应用程序中，我们应明确地处理该错误，至少应终止循环，以避免继续尝试发送消息。示例 17-40 展示了一个简单的错误处理策略：打印问题并从循环 `break` 出来。

文件名：src/main.rs

```
fn get_messages() -> impl Stream<Item = String> {
    let (tx, rx) = trpl::channel();

    trpl::spawn_task(async move {
```

```
        let messages = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"];

        for (index, message) in messages.into_iter().enumerate() {
            let time_to_sleep = if index % 2 == 0 { 100 } else { 300 };
            trpl::sleep(Duration::from_millis(time_to_sleep)).await;

            if let Err(send_error) = tx.send(format!("Message: '{message}'")) {
                eprintln!("Cannot send message '{message}': {send_error}");
                break;
            }
        }
    });

    ReceiverStream::new(rx)
}

fn get_intervals() -> impl Stream<Item = u32> {
    let (tx, rx) = trpl::channel();

    trpl::spawn_task(async move {
        let mut count = 0;
        loop {
            trpl::sleep(Duration::from_millis(1)).await;
            count += 1;

            if let Err(send_error) = tx.send(count) {
                eprintln!("Could not send interval {count}: {send_error}");
                break;
            };
        }
    });

    ReceiverStream::new(rx)
}
```

同往常一样，正确处理消息发送失败的方式会有所不同：只要确保你有一个策略即可。

现在我们已经看过了很多异步实践，让我们稍作回顾，更深入地探讨一下 Rust 中用于实现异步的 `Future`、`Stream` 和其它关键 trait 的一些细节。

深入理解 async 相关的 traits

贯穿本章，我们通过多种方式使用了 `Future`、`Pin`、`Unpin`、`Stream` 和 `StreamExt` trait。但是直到目前为止，我们避免过多地了解它们如何工作或者如何组合在一起的细节，这对你日常的 Rust 开发而言通常是没问题的。不过有时你会遇到需要了解更多细节的场景。在本小节，我们会足够深入以便理解这些场景，并仍会将 真正 有深度的内容留给其它文档。

`Future` trait

让我们以更深入地了解 `Future` trait 作为开始。这里是 Rust 中其如何定义的：

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

trait 定义中包含一些的新类型和我们之前没有见过的新语法，所以让我们逐步详细地解析一下这个定义。

首先， `Future` 的关联类型 `Output` 表明 future 最终解析出的类型。这类似于 `Iterator` trait 的关联类型 `Item`。其次，`Future` 还有一个 `poll` 方法，其有一个特殊的 `self` 参数的 `Pin` 引用和一个 `Context` 类型的可变引用，并返回一个 `Poll<Self::Output>`。稍后我们再细说 `Pin` 和 `Context`。现在让我们专注于方法返回的 `Poll` 类型：

```
enum Poll<T> {
    Ready(T),
    Pending,
}
```

`Poll` 类型类似于一个 `Option`。它有一个包含值的变体 `Ready(T)`，和一个没有值的变体 `Pending`。不过 `Poll` 所代表的意义与 `Option` 非常不同！`Pending` 变体表明 future 仍然还有工作要进行，所有调用者稍后需要再次检查。`Ready` 变体表明 future 已经完成了其工作并且 `T` 的值是可用的。

> 注意：对于大部分功能，调用者不应在 future 返回 `Ready` 后再次调用 `poll`。很多 future 在完成后再次轮询会 panic。可以安全地再次轮询的 future 会在文档中显式地说明。这类似于 `Iterator::next` 的行为。

当你见到使用 `await` 的代码时，Rust 会在底层将其编译为调用 `poll` 的代码。如果你回头看下示例 17-4，其在一个单个 URL 解析完成后打印出页面标题，Rust 将其编译为一些类似（虽然不完全是）这样的代码：

```
match page_title(url).poll() {
    Ready(page_title) => match page_title {
```

```
        Some(title) => println!("The title for {url} was {title}"),
        None => println!("{url} had no title"),
    }
    Pending => {
        // 但这里运行什么呢?
    }
}
```

如果 future 仍然是 `Pending` 的话我们应该做什么呢？我们需要某种方式不断重试，直到 future 最终准备好。换句话说，我们需要一个循环：

```
let mut page_title_fut = page_title(url);
loop {
    match page_title_fut.poll() {
        Ready(value) => match page_title {
            Some(title) => println!("The title for {url} was {title}"),
            None => println!("{url} had no title"),
        }
        Pending => {
            // continue
        }
    }
}
```

不过，如果 Rust 真的将代码精确地编译成那样，那么每一个 `await` 都会变成阻塞操作 – 这恰恰与我们的目标相反！相反，Rust 确保循环可以将控制权交给一些可以暂停当前 future 转而去处理其它 future 并在之后再次检查当前 future 的内容。如你所见，这就是异步运行时，这种安排和协调的工作是其主要工作之一。

在本章前面的内容中，我们描述了等待 `rx.recv`。`recv` 调用返回一个 future，并 await 轮询它的 future。我们注意到当信道关闭时运行时会暂停 future 直到它就绪并返回 `Some(message)` 或 `None` 为止。随着我们对 `Future` trait，尤其是 `Future::poll` 的理解的深入，我们可以看出其是如何工作的。运行时知道 future 返回 `Poll::Pending` 时它还没有完成。反过来说，当 `poll` 返回 `Poll::Ready(Some(message))` 或 `Poll::Ready(None)` 时运行时知道 future **已经**完成了并继续运行。

运行时如何工作的具体细节超出了本书的范畴。不过关键在于理解 future 的基本机制：运行时**轮询**其所负责的每一个 future，在它们还没有完成时使其休眠。

`Pin` 和 `Unpin` traits

当我们在示例 17-16 中引入 pin 的概念时，我们遇到了一个很不友好的错误信息。这里再次展示其中相关的部分：

```
error[E0277]: `{async block@src/main.rs:10:23: 10:33}` cannot be unpinned
  --> src/main.rs:48:33
   |
48 |         trpl::join_all(futures).await;
   |                                 ^^^^^ the trait `Unpin` is not implemented
for `{async block@src/main.rs:10:23: 10:33}`
   |
   = note: consider using the `pin!` macro
```

```
           consider using `Box::pin` if you need to access the pinned value
outside of the current scope
   = note: required for `Box<{async block@src/main.rs:10:23: 10:33}>` to implement
`Future`
note: required by a bound in `futures_util::future::join_all::JoinAll`
  --> file:///home/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/futures-
util-0.3.30/src/future/join_all.rs:29:8
   |
27 | pub struct JoinAll<F>
   |            ------- required by a bound in this struct
28 | where
29 |     F: Future,
   |        ^^^^^^ required by this bound in `JoinAll`
```

这个错误信息不仅告诉我们需要对这些值进行 pin 操作，还解释了为什么 pin 是必要的。`trpl::join_all` 函数返回一个叫做 `JoinAll` 的结构体。这个结构体是一个 `F` 类型的泛型，它被限制为需要实现 `Future` trait。通过 `await` 直接 await 一个 future 会隐式地 pin 住这个函数。这也就是为什么我们不需要在任何想要 await future 的地方使用 `pin!`。

然而，这里我们没有直接 await 一个 future。相反我们通过向 `join_all` 函数传递一个 future 集合来构建了一个新 future `JoinAll`。`join_all` 的签名要求集合中项的类型都要实现 `Future` trait，而 `Box<T>` 只有在其封装的 `T` 是一个实现了 `Unpin` trait 的 future 时才会实现 `Future`。

这有很多需要吸收的知识！为了真正地理解它，让我们稍微深入理解 `Future` 实际上是如何工作的，特别是 *pinning* 那一部分。

再次观察 `Future` trait 的定义：

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

这里的 `cx` 参数及其 `Context` 类型，是运行时如何在仍保持 lazy 的情况下实际知道何时去检查任何给定的 future 的关键。同样，它们是如何工作的细节超出了本章的范畴，通常你只有在编写自定义 `Future` 实现时才需要思考它。相反我们将关注 `self` 的类型，因为这是第一次见到 `self` 有类型注解的方法。`self` 的类型注解与其它参数的类型注解类似，但有两个关键区别：

- 它告诉 Rust 在调用该方法时 `self` 必须具备的类型。
- 它不能是任意类型。这限制了实现了该方法的类型，是一个该类型的引用或者智能指针，或者一个封装了该类型引用的 `Pin`。

我们会在 第十八章 更多地看到这个语法。现在，知道如果要轮询一个 future 来检查它是 `Pending` 或者 `Ready(Output)`，我们需要一个 `Pin` 封装的该类型的可变引用就够了。

`Pin` 是一个类指针类型的封装，比如 `&`，`&mut`，`Box` 和 `Rc`。（从技术上来说，`Pin` 适用于实现了 `Deref` 或 `DerefMut` trait 的类型，不过这实际上等同于只能适用于指针。）`Pin` 本身并不是一

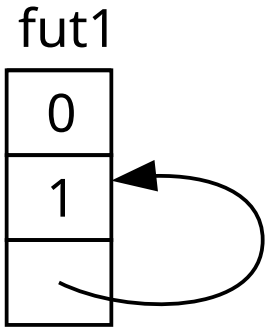
个指针并且也不具备类似 `Rc` 和 `Arc` 那样引用技术的功能；它单纯地是一个编译器可以用来约束指针使用的工具。

回忆一下 `await` 的实现是基于对 `poll` 的调用，这有助于解释之前见到的错误信息，不过那是关于 `Unpin` 的。所以 `Pin` 具体与 `Unpin` 有何关联，又为什么 `Future` 需要 `self` 在一个 `Pin` 类型中才能调用 `poll`？

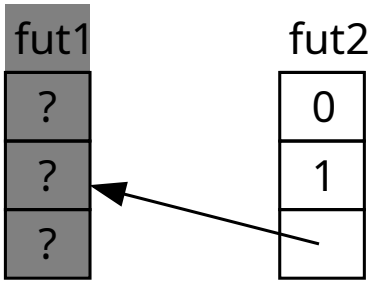
回忆一下本章之前 future 中一系列的 await point 被编译为一个状态机，而编译器确保这个状态机遵守 Rust 在安全方面的所有常规规则，包括借用和所有权。为了使其正常工作，Rust 检查在一个 await point 和另一个 await point 之间或到异步代码块结尾之间什么数据是需要的。编译器接着在生成的状态机中创建一个相应的变体。每一个变体获得其在源码中对应片段所用到的数据的访问权限，要么获取数据的所有权要么获取其可变或不可变引用。

到目前为止，一切正常：如果你在给定异步代码块中搞错了所有权或者引用，借用检查器会告诉你。当我们需要移动对应代码块的 future – 例如将其移动到 `Vec` 中或者传递给 `join_all` – 问题就会变得棘手。

当我们移动一个 future – 将其移动进一个数据结构通过 `join_all` 将其用作迭代器或者将其从函数返回 – 这实际上意味着要移动 Rust 为我们创建的状态机。而与 Rust 中大多数其他类型不同，Rust 为异步代码块创建的 future 可能最终会在任意给定变体的字段中包含对其自身的引用，如图 17-4 中的简化图所示。

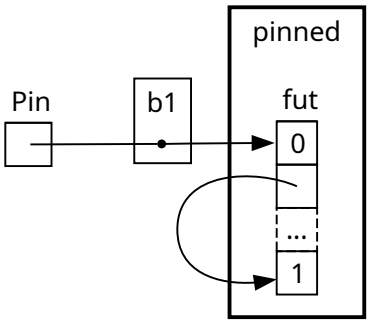


但是，默认情况下移动任何拥有其自身引用的对象是不安全（unsafe）的，因为引用总是会指向任何其引用数据的实际内存地址（见图 17-5）。如果我们移动数据结构本身，这些内部引用会停留在指向老的地址。然而，这些内存地址现在是无效的。一方面，当修改这些数据结构时这些值不会被更新。更重要的是，计算机现在可以随意将这块内存重新用于其他用途！之后你最终可能会读取到完全不相关的数据。

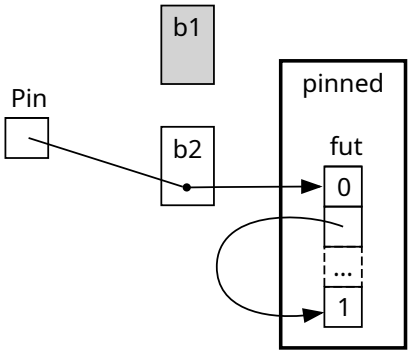


理论上来说，Rust 编译器也可以在对象被移动时尝试更新其所有的引用，不过这会增加很多性能开销，特别是在有一整个网状的引用需要更新的时候。相反如果我们确保相关的数据结构**不会再内存中移动**，就无需更新任何引用。这正是 Rust 的借用检查器所要求的：在安全代码中，禁止移动任何有自身活动引用的项。

而 `Pin` 正是在此基础上，为我们提供了所需的保证。当我们通过 `Pin` 封装一个值的引用来 **pin** 住它时，它就无法再移动了。也就是说，如果有 `Pin<Box<SomeType>>`，你实际上 pin 住了 `SomeType` 的值，而**不是** `Box` 指针。图 17-6 解释了这一过程。



事实上，`Box` 指针仍然可以随意移动。请记住：我们关心确保最终被引用的数据保持不动。如果指针移动了，**但是它指向的数据还在相同的位置**，就像图 17-7 一样，就不会有潜在的问题。（作为一个独立的练习，可以查看相关类型以及 `std::pin` 模块的文档，尝试推导出如何使用一个包裹 `Box` 的 `Pin` 来实现这一点。）其关键在于自引用类型本身不可移动，因为它仍然是被 pin 住的。



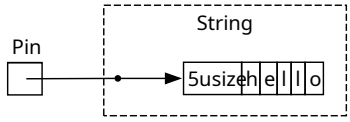
然而，大多数类型即使被封装在 `Pin` 后面，也完全可以安全地移动。只有当项中含有内部引用的时候才需要考虑 pin。像数字或者布尔值这样的基本类型值是安全的因为很明显它们没有任何内部引用。大多数你在 Rust 中常用的类型也同样如此。例如你可以移动一个 `Vec` 而不用担心。考虑到目前我们所见到的，如果有一个 `Pin<Vec<String>>`，即便在没有其他引用的情况下 `Vec<String>` 始终可以安全移动，你仍然必须通过 `Pin` 提供的安全但有限的 API 来进行所有操作。我们需要一个方法来告诉编译器在类似这种情况下移动项是可以的 – 这就是 `Unpin` 的用武之地了。

`Unpin` 是一个标记 trait（marker trait），类似于我们在第十六章见过的 `Send` 和 `Sync` trait，因此它们自身没有功能。标记 trait 的存在只是为了告诉编译器在给定上下文中可以安全地使用实现了给定 trait 的类型。`Unpin` 告知编译器这个给定类型**无需**维护被提及的值是否可以安全地移动的任何保证。

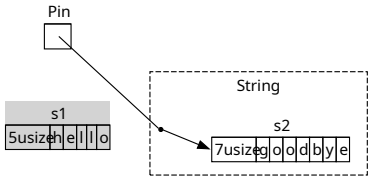
正如 `Send` 和 `Sync` 一样，编译器自动为所有被证明为安全的类型实现 `Unpin`。同样类似于 `Send` 和 `Sync`，有一个特殊的例子**不会**为类型实现 `Unpin`。这个例子的符号是 impl !Unpin for SomeType，这里 SomeType 指的是这样的一种类型：为了确保内存安全，当一个指向它的指针被用于 `Pin` 时，无论何时它都**必须**维护其不被移动的安全保证。

换句话说，关于 `Pin` 与 `Unpin` 的关系有两点需要牢记。首先，`Unpin` 用于 “正常” 情况，而 `!Unpin` 用于特殊情况。其次，不管一个类型是实现了 `Unpin` 或者实现了 `!Unpin`，它**只在**你使用了一个被 pin 住的指向类似 Pin<&mut SomeType> 类型的指针时才会产生影响。

更具体地说，考虑一下 `String`：它包含一个长度和构成它的 Unicode 字符。我们可以将 `String` 封装进 `Pin` 中，如图 17-8 所示。然而，就像 Rust 中大部分其它类型一样，`String` 自动实现了 `Unpin`。



因此，对于 `String` 来说，我们可以像图 17-9 这样在完全相同的内存位置将一个字符串替换为另一个字符串，而如果 `String` 实现的是 `!Unpin`，这个操作本该是非法的。这并不违反 `Pin` 的规则，因为 `String` 没有内部引用这使得它可以安全地移动！这是为何它实现了 `Unpin` 而不是 `!Unpin`。



现在我们已经掌握足够的知识来理解示例 17-17 中对 `join_all` 调用所报告的错误了。最初我们尝试将异步代码块产生的 future 移动进 `Vec<Box<dyn Future<Output = ()>>>` 中，不过正如之前所见，这些 future 可能包含内部引用，因此它们并未实现 `Unpin`。它们需要被 pin 住，接下来就可以将 `Pin` 类型传入 `Vec`，并确信 future 底层的数据**不会**被移动。

`Pin` 和 `Unpin` 在编写底层代码库或你自己编写运行时的时候最为重要，而不是在日常的 Rust 代码中。不过，现在当你在错误信息中看到这些 trait 时，就能想出更好的方式来修复代码了！

> 注意：`Pin` 与 `Unpin` 的组合使得可以安全地实现在 Rust 中原本因自引用而难以实现的一整类复杂类型。要求 `Pin` 的类型在如今的异步 Rust 中最为常见，不过偶尔你也会在其它上下文中见到它们。
>
> `Pin` 和 `Unpin` 如何工作的细节，以及它们要求维护的规则，在 `std::pin` 的 API 文档中有详尽的介绍，所以如果你有兴趣学习更多，这是一个很好的起点。
>
> 如果你希望更深入地理解底层是如何实现的细节，请查看 *Asynchronous Programming in Rust* 的第二章和第四章。

`Stream` trait

现在你对 `Future`、`Pin` 和 `Unpin` trait 有更深刻的理解了，我们可以将注意力转向 `Stream` trait。如你在本章之前所学的，流类似于异步迭代器。但是不同于 `Iterator` 和 `Future`，截至撰写本书时 `Stream` 在标准库中并无定义，不过在 `futures` crate 中**存在**一个在整个生态系统中广泛使用的常见定义。

在学习 `Stream` trait 如何能够将 `Iterator` 和 `Future` trait 结合在一起之前，让我们先回顾一下它们的定义。从 `Iterator` 中我们引入了序列的概念：其 `next` 方法提供一个 `Option<Self::Item>`。从 `Future` 中我们学习到随时间就绪的概念：其 `poll` 方法提供一个 `Poll<Self::Output>`。为了表示一个随着时间就绪的项的序列，我们定义了一个将这些功能结合到一起的 `Stream` trait：

```
use std::pin::Pin;
use std::task::{Context, Poll};

trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

`Stream` trait 定义了一个名为 `Item` 的关联类型来作为流所产生项的类型。这类似于 `Iterator`，其中可能含有零个到多个项，而有别于 `Future`，后者总是只有一个 `Output`，即使它是 unit 类型 `()`。

`Stream` 也定义了一个获取这些项的方法。名为 `poll_next`，来明确它以 `Future::poll` 同样的方式轮询并以 `Iterator::next` 同样的方式产生一系列的项。其返回类型用 `Option` 组合了 `Poll`。外部类型是 `Poll`，因为它必须检查可用性，就像 future 一样。内部类型是 `Option`，因为它需要表明是否有更多消息，就像迭代器一样。

与此定义非常相似的实现很可能最终会成为 Rust 标准库的一部分。目前，它是大部分运行时工具箱的一部分，所以你可以依赖它，并且接下来所讲一切应该也是适用的！

不过，在这一部分我们之前见过的关于流的示例中，我们没有使用 `poll_next` **或** `Stream`，相反我们使用了 `next` 和 `StreamExt`。当然，我们**可以**通过手写自己的 `Stream` 状态机来直接处理 `poll_next` API，就像**可以**通过 `poll` 方法直接处理 future 一样。不过，使用 `await` 更加优雅，同时 `StreamExt` trait 提供了 `next` 方法以便我们可以这样做：

```
trait StreamExt: Stream {
    async fn next(&mut self) -> Option<Self::Item>
    where
        Self: Unpin;

    // other methods...
}
```

> 注意：本章之前用到的实际定义与这个看起来略有不同，因为它需要支持还不支持在 trait 中使用异步函数的 Rust 版本。因此，它看起来像这样：
>
> ```
> fn next(&mut self) -> Next<'_, Self> where Self: Unpin;
> ```
>
> `Next` 类型是一个实现了 `Future` 并通过 `Next<'_, Self>` 允许我们命名 `self` 引用生命周期的 `struct`，因此 `await` 可以处理这个方法。

`StreamExt` trait 也是所有可用于流的有趣方法所在的 trait。`StreamExt` 自动为所有实现了 `Stream` 的方法实现，不过这些 trait 是分别定义的以便社区可以迭代便利的工具而不会影响基础 trait。

在 `trpl` crate 所用到的 `StreamExt` 版本中，该 trait 不仅定义了 `next` 方法而且提供了一个正确处理 `Stream::poll_next` 细节的 `next` 方法默认实现。这意味着即便当你编写自己的流数据类型时，**只需**实现 `Stream`，接着任何使用你数据类型的人就自动地可以使用 `StreamExt` 及其方法。

这就是我们要涉及的这些 trait 的底层细节的全部了。最后，让我们来思考 futures（包括 streams）、任务和线程如何协同配合！

结合使用 future、任务和线程

正如我们在第十六章所见，线程提供了一种并发的方式。在这一章节我们见过了另一种方式：通过 future 和流来使用异步。如果你好奇何时选择一个而不是另一个，答案是：视具体情况而定！同时在很多场景下，我们不应只选择线程**或**异步，而应同时考虑线程**和**异步两者。

几十年来很多操作系统已经提供了基于线程的并发模型，因此很多编程语言也对其提供了支持。然而这些模型并非没有取舍。在很多操作系统中，它们为每一个线程使用了不少的内存，同时启动和停止带来了一些开销。线程也只有当你的操作系统和硬件支持它们的时候才是一个选项。不同于主流的桌面和移动电脑，一些嵌入式系统根本没有操作系统，因此也就没有线程。

异步模型提供了一个不同的 – 最终也是互补的 – 权衡取舍。在异步模型中，并发操作无需各自独立的线程。相反，它们运行在任务上，正如流小节中我们用 `trpl::spawn_task` 从异步函数中开始工作一样。任务类似于线程，但不是由操作系统管理，而是由库级别的代码管理：也就是运行时。

在上一小节中，我们看到可以通过异步信道来构建一个流并产生一个可以在异步代码中调用的异步任务。我们也可以用线程来做到完全相同的事情。在示例 17-40 中使用了 `trpl::spawn_task` 和 `trpl::sleep`。在示例 17-41 中，我们将 `get_intervals` 函数中的代码替换为标准库中的 `thread::spawn` 和 `thread::sleep` API。

文件名：src/main.rs

```
fn get_intervals() -> impl Stream<Item = u32> {
    let (tx, rx) = trpl::channel();

    // 这里 *不是* `trpl::spawn`，是 `std::thread::spawn`！
    thread::spawn(move || {
        let mut count = 0;
        loop {
            // 同样，这里 *不是* `trpl::sleep`，是 `std::thread::sleep`！
            thread::sleep(Duration::from_millis(1));
            count += 1;

            if let Err(send_error) = tx.send(count) {
                eprintln!("Could not send interval {count}: {send_error}");
                break;
            };
        }
    });

    ReceiverStream::new(rx)
}
```

如果你运行这段代码，其输入与示例 17-40 的一样。同时请注意从调用代码的角度来说改变是多么的微小。而且，即便一个函数在运行时上产生一个异步任务而另一个产生一个系统线程，其返回的流不受该区别的影响。

尽管它们是相似的，这两种方式的行为非常不同，尽管在这个非常简单的例子中我们可能很难进行测量。我们可以在任何现代计算机中产生数以百万计的异步任务。如果尝试用线程来这样做，我们实际上会耗尽内存！

然而，这些 API 如此相似是有理由的。线程作为同步操作集的边界；线程**之间**的并发是可能的。任务作为**异步**操作集的边界，任务**之间**和**之内**的并发是可能的，因为任务可以在其内部切换 future。最后，future 是 Rust 中最细粒度的并发单位，同时每一个 future 可能代表一棵由其它 future 组成的树。其运行时 – 更具体地说，其执行器（executor）– 管理任务，任务则管理 future。在这一点上，任务类似于轻量的、运行时管理的线程，并具有由运行时而非操作系统管理所赋予的额外能力。

这并不意味着异步任务总是优于线程（反之亦然）。基于线程的并发在某种程度上来说是一个比基于 `async` 的并发更简单的编程模型。这既可以是优点，也可以是缺点。线程有点像 “射后不理”（“fire and forget”）；它们没有原生等同于 future 的机制，所以它们简单地运行到结束而不会被中断，除非操作系统本身介入。也就是说，它们没有像 future 那样内建**任务内并发**（**intratask concurrency**）支持。Rust 中的线程也没有提供取消机制 – 本章虽未明确讨论此主题，但当我们结束一个 future 时，其状态能够被正确清理就隐含了这一事实。

这些限制也使得线程比 future 更加难以组合。例如，更加难以使用线程构建类似于本章之前的 `timeout` 和 `throttle` 辅助方法。正如我们所见，future 更加丰富的数据结构的事实意味着它们可以更自然地组合在一起。

接下来，任务在 future 之上提供了**额外**控制，允许我们选择在何处如何组合它们。同时事实证明线程和 future 常常能很好地协同工作，因为任务可以（至少是在一些运行时中）在线程间移动。事实上，在底层我们使用的运行时 – 包括 `spawn_blocking` 和 `spawn_task` 函数 – 默认就是多线程的！很多运行时采用一种被称为**工作窃取**（**work stealing**）的方式来透明地在线程间移动任务，它基于当前线程是如何被利用的，以提高系统的整体性能。这个方式实际上需要线程**和**任务，因此也需要 future。

当思考何时采用哪种方法时，考虑这些经验法则：

- 如果工作是**非常可并行的**，例如处理大量数据其中每一部分数据都可以单独处理时，线程是更佳的选择。
- 如果工作是**非常并发的**，例如处理大量不同来源的消息，它们可能有着不同的间隔或者速率，异步是更佳的选择。

同时如果你同时需要并行和并发，也无需在线程和异步间做出选择。你可以随意同时使用它们，让它们各自处理最擅长的工作。例如，示例 17-42 展示了一个这样的真实世界 Rust 代码中非常常见的组合示例。

文件名：src/main.rs

```
use std::{thread, time::Duration};

fn main() {
    let (tx, mut rx) = trpl::channel();

    thread::spawn(move || {
        for i in 1..11 {
            tx.send(i).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    trpl::run(async {
        while let Some(message) = rx.recv().await {
            println!("{message}");
```

```
        }
    });
}
```

我们以创建异步信道作为开始，接着产生一个线程来获取信道发送端的所有权。在线程中，我们发送数字 1 到 10，每个之间休眠一秒。最后，就像贯穿本章的那样将一个 async 代码块创建的 future 传递给 `trpl::run` 运行。在 future 中，我们 await 这些信息，就像我们见过的其它消息传递的示例那样。

为了回到本章开头提出的场景，想象一下用一个专门的线程来运行一系列视频解码任务（因为视频解码是计算密集型任务）不过通知 UI 这些任务完成了是通过异步信道完成的。在现实世界的用例中有无数这类组合的例子。

总结

这并不会是本书中你最后一次接触并发。第二一章 中的项目会在一个更加真实的场景中运用这些概念，而不是这里讨论的简单示例，同时会更直接地比较线程和任务的解决问题方式。

无论你选择何种方式，Rust 提供了编写安全、快速和并发代码的工具 – 无论是用于高吞吐量 web 服务器或是用于嵌入式操作系统。

接下来，我们会讨论随着 Rust 程序增大时如何以惯用的方式对问题进行建模和对解决方案进行结构化。此外我们还会讨论 Rust 的惯用写法如何与你可能已经熟悉的面向对象编程惯例相对应。

面向对象编程特性

面向对象编程（Object-Oriented Programming，OOP）是一种对程序进行建模的方式。对象（Object）作为一个编程概念来源于 20 世纪 60 年代的 Simula 编程语言。这些对象影响了 Alan Kay 的编程架构，该架构中对象之间互相传递消息。他于 1967 年创造了**面向对象编程**（*object-oriented programming*）这一术语。对于 OOP 的定义众说纷纭；在一些定义下，Rust 是面向对象的；在其他定义下，Rust 不是。在本章节中，我们会探索一些被普遍认为是面向对象的特性和这些特性是如何体现在 Rust 语言习惯中的。接着会展示如何在 Rust 中实现面向对象设计模式，并讨论这么做与利用 Rust 自身的一些优势实现的方案相比有什么取舍。

面向对象语言的特征

关于一门语言必须具备哪些特征才能被视为面向对象，目前在编程社区中并没有共识。Rust 受到了许多编程范式的影响，包括面向对象编程（OOP）；例如，在第 13 章中，我们探讨了来自函数式编程的特性。可以说，面向对象的语言共有一些共同的特征，即对象、封装和继承。我们将会讨论这些特征分别是什么，以及 Rust 是否支持它们。

对象包含数据和行为

由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides（Addison-Wesley Professional, 1994）编写的书 *Design Patterns: Elements of Reusable Object-Oriented Software* ，通称 *The Gang of Four*，是一本面向对象设计模式的目录。它这样定义面向对象编程：

> Object-oriented programs are made up of objects. An *object* packages both data and the procedures that operate on that data. The procedures are typically called *methods* or *operations*.
>
> 面向对象的程序由对象组成。一个**对象**包含数据和操作这些数据的过程。这些过程通常被称为**方法** 或**操作**。

在这个定义下，Rust 是面向对象的：结构体和枚举包含数据而 `impl` 块提供了在结构体和枚举之上的方法。虽然带有方法的结构体和枚举并不被**称为**对象，但是参考 *The Gang of Four* 中对象的定义，它们提供了与对象相同的功能。

封装隐藏了实现细节

另一个通常与面向对象编程关联的概念是 **封装**（*encapsulation*）：一个对象的实现细节对使用该对象的代码不可见。因此，对象交互的唯一方式是通过其公有 API；使用对象的代码不应能直接触及对象的内部并改变数据或行为。这使得程序员能够更改和重构一个对象的内部实现，而无需改变使用该对象的代码。

我们在第七章讨论了如何控制封装：我们可以使用 `pub` 关键字来决定代码中的哪些模块、类型、函数和方法是公有的，而默认情况下其他所有内容都是私有的。例如，我们可以定义一个 `AveragedCollection` 结构体，其中有一个存有 `Vec<i32>` 的字段。该结构体还可以有一个字段存储向量中值的平均值，从而无需在每次需要时重新计算。换句话说，`AveragedCollection` 会为我们缓存已计算的平均值。示例 18-1 给出了 `AveragedCollection` 结构体的定义：

文件名：src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

示例 18-1: `AveragedCollection` 结构体维护了一个整型列表及其所有元素的平均值。

该结构体被标记为 `pub`，这样其他代码就可以使用它，但结构体内的字段仍保持私有。这在这种情况下很重要，因为我们想确保每当列表中添加或删除值时，平均值也会更新。我们通过实现结构体上的 `add`、`remove` 和 `average` 方法来做到这一点，如示例 18-2 所示：

文件名：src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

示例 18-2: 在 `AveragedCollection` 结构体上实现了 `add`、`remove` 和 `average` 公有方法

公有方法 `add`、`remove` 和 `average` 是访问或修改 `AveragedCollection` 实例中数据的唯一途径。当使用 `add` 方法把一个元素加入到 `list` 或者使用 `remove` 方法来删除时，这些方法的实现同时会调用私有的 `update_average` 方法来更新 `average` 字段。

`list` 和 `average` 是私有的，所以没有其他方式来使得外部的代码直接向 `list` 增加或者删除元素，否则 `list` 改变时可能会导致 `average` 字段不同步。`average` 方法返回 `average` 字段的值，这使得外部的代码只能读取 `average` 而不能修改它。

因为我们已经封装了 `AveragedCollection` 的实现细节，改动数据结构等内部实现非常简单。例如，可以使用 `HashSet<i32>` 代替 `Vec<i32>` 作为 `list` 字段的类型。只要 `add`、`remove` 和 `average` 这些公有方法的签名保持不变，使用 `AveragedCollection` 的代码就无需改变。如果我们将 `list` 设为公有，情况就未必如此：`HashSet<i32>` 和 `Vec<i32>` 使用不同的方法增加或移除项，所以如果外部代码直接修改 `list`，很可能需要进行更改。

如果封装被认为是面向对象语言所必要的特征，那么 Rust 满足这个要求。在代码中不同的部分控制 `pub` 的使用来封装实现细节。

继承作为类型系统与代码共享

继承（*Inheritance*）是一种机制：一个对象可以从另一个对象的定义中继承元素，从而获得父对象的数据和行为，无需再次定义。

如果一种语言必须具有继承才能被认为是面向对象语言，那么 Rust 不是其中之一。Rust 不支持定义一个结构体时继承父结构体的字段和方法，除非使用宏。

然而，如果您习惯于在编程过程中使用继承，那么根据运用继承的原因，Rust 提供了其他解决方案。

选择继承有两个主要的原因。其一是代码复用：您可以为一种类型实现特定的行为，继承可将其复用到不同的类型上。在 Rust 代码中可以使用默认 trait 方法实现来进行有限的代码复用，就像示例 10-14 中在 `Summary` trait 上增加的 `summarize` 方法的默认实现。任何实现了 `Summary` trait 的类型都可以使用 `summarize` 方法而无须进一步实现。这类似于父类有一个方法的实现，继承的子类也拥有这个方法的实现。当实现 `Summary` trait 时也可以选择覆盖 `summarize` 的默认实现，这类似于子类覆盖从父类继承方法的实现。

其二与类型系统有关：子类型可以用于父类型被使用的地方。这也被称为**多态**（*polymorphism*）：如果多个对象共享某些特征，可以在运行时将它们互相替代。

> **多态（Polymorphism）**
>
> 对很多人来说，多态性与继承同义。但它实际上是一个更广义的概念，指的是可以处理多种类型数据的代码。对继承而言，这些类型通常是子类。
>
> Rust 使用泛型来抽象不同可能的类型，并通过 trait bound 来约束这些类型所必须提供的内容。这有时被称为 *bounded parametric polymorphism*。

作为一种语言设计的解决方案，继承在许多新的编程语言中逐渐不被青睐，因为它经常有分享过多代码的风险。子类不应总是共享父类的所有特征，但是继承始终如此。这会降低程序设计的灵活性。它还引入了在子类上调用方法的可能性，这些方法可能没有意义，或因为方法不适用于子类而导致错误。此外，一些语言只允许单一继承（意味着子类只能从一个类继承），进一步限制了程序设计的灵活性。

出于这些原因，Rust 使用 trait 对象而非继承。接下来我们会讨论 Rust 如何使用 trait 对象实现多态性。

顾及不同类型值的 trait 对象

在第八章中，我们谈到了 vector 只能存储同种类型元素的局限性。示例 8-9 中提供了一个替代方案，通过定义 `SpreadsheetCell` 枚举，来储存整型、浮点型或文本类型的变体。这意味着，我们可以在每个单元中储存不同类型的数据，并仍能拥有一个代表一排单元的 vector。只要我们需存储的值由一组固定的类型组成，并且在代码编译时就知道具体会有哪些类型，那么这种使用枚举的办法是完全可行的。

然而有时我们希望库用户在特定情况下能够扩展有效的类型集合。为了展示如何实现这一点，这里将创建一个图形用户接口（Graphical User Interface，GUI）工具的例子，它通过遍历列表并调用每一个项目的 `draw` 方法来将其绘制到屏幕上 —— 此乃一个 GUI 工具的常见技术。我们将要创建一个叫做 `gui` 的库 crate，它含一个 GUI 库的结构。这个 GUI 库包含一些可供开发者使用的类型，比如 `Button` 或 `TextField`。在此之上，`gui` 的用户希望创建自定义的可以绘制于屏幕上的类型：比如，一个程序员可能会增加 `Image`，另一个可能会增加 `SelectBox`。

这个例子中并不会实现一个功能完善的 GUI 库，不过会展示其中各个部分是如何结合在一起的。编写库的时候，我们不可能知晓并定义所有其他程序员希望创建的类型。我们所知晓的是 `gui` 需要记录一系列不同类型的值，并需要能够对其中每一个值调用 `draw` 方法。这里无需知道调用 `draw` 方法时具体会发生什么，只要该值会有那个方法可供我们调用即可。

在拥有继承的语言中，可以定义一个名为 `Component` 的类，该类上有一个 `draw` 方法。其他的类比如 `Button`、`Image` 和 `SelectBox` 会从 `Component` 派生并因此继承 `draw` 方法。它们各自都可以重写 `draw` 方法来定义自己的行为，但是框架会把所有这些类型当作是 `Component` 的实例，并在其上调用 `draw`。不过 Rust 并没有继承，我们需要寻找另一种方式来设计 `gui` 库，以便用户能够使用新类型进行扩展。

定义通用行为的 trait

为了实现 `gui` 所期望的行为，让我们定义一个 `Draw` trait，其中包含名为 `draw` 的方法。接着可以定义一个存放 **trait 对象**（*trait object*）的 vector。trait 对象指向一个实现了我们指定 trait 的类型的实例，以及一个用于在运行时查找该类型的 trait 方法的表。我们通过指定某种指针来创建 trait 对象，例如 `&` 引用或 `Box<T>` 智能指针，还有 `dyn` 关键字，以及指定相关的 trait（第二十章 “动态大小类型和 `Sized` trait” 部分会介绍 trait 对象必须使用指针的原因）。我们可以使用 trait 对象代替泛型或具体类型。任何使用 trait 对象的位置，Rust 的类型系统会在编译时确保任何在此上下文中使用的值会实现其 trait 对象的 trait。如此便无需在编译时就知晓所有可能的类型。

之前提到过，Rust 刻意不将结构体与枚举称为 “对象”，以便与其他语言中的对象相区别。在结构体或枚举中，结构体字段中的数据和 `impl` 块中的行为是分开的，不同于其他语言中将数据和行为组合进一个称为对象的概念中。trait 对象将数据和行为两者相结合，从这种意义上说**则**更类似其他语言中的对象。不过 trait 对象不同于传统的对象，因为不能向 trait 对象添加数据。trait 对象并不像其他语言中的对象那么通用：其具体的作用是允许对通用行为进行抽象。

示例 18-3 展示了如何定义一个带有 `draw` 方法的 trait `Draw`：

文件名：src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

示例 18-3：`Draw` trait 的定义

因为第十章已经讨论过如何定义 trait，其语法看起来应该比较眼熟。接下来就是一些新语法：示例 18-4 定义了一个存放了名叫 `components` 的 vector 的结构体 `Screen`。这个 vector 的类型是 `Box<dyn Draw>`，此为一个 trait 对象：它是 `Box` 中任何实现了 `Draw` trait 的类型的替身。

文件名：src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

示例 18-4: 一个 `Screen` 结构体的定义，它带有一个字段 `components`，其包含实现了 `Draw` trait 的 trait 对象的 vector

在 `Screen` 结构体上，我们将定义一个 `run` 方法，该方法会对其 `components` 上的每一个组件调用 `draw` 方法，如示例 18-5 所示：

文件名：src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

示例 18-5：在 `Screen` 上实现一个 `run` 方法，该方法在每个 component 上调用 `draw` 方法

这与定义使用了带有 trait bound 的泛型类型参数的结构体不同。泛型类型参数一次只能替代一个具体类型，而 trait 对象则允许在运行时替代多种具体类型。例如，可以定义 `Screen` 结构体来使用泛型和 trait bound，如示例 18-6 所示：

文件名：src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where
    T: Draw,
{
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

示例 18-6: 一种 `Screen` 结构体的替代实现，其 `run` 方法使用泛型和 trait bound

这限制了 `Screen` 实例必须拥有一个全是 `Button` 类型或者全是 `TextField` 类型的组件列表。如果只需要同质（相同类型，homogeneous）集合，则倾向于使用泛型和 trait bound，因为其定义会在编译时采用具体类型进行单态化（monomorphized）。

另一方面，通过使用 trait 对象的方法，一个 `Screen` 实例可以存放一个既能包含 `Box<Button>`，也能包含 `Box<TextField>` 的 `Vec<T>`。让我们看看它是如何工作的，接着会讲到其运行时性能影响。

实现 trait

现在来增加一些实现了 `Draw` trait 的类型。我们将提供 `Button` 类型。再一次重申，真正实现 GUI 库超出了本书的范畴，所以 `draw` 方法体中不会有任何有意义的实现。为了想象一下这个实现看起来像什么，一个 `Button` 结构体可能会拥有 `width`、`height` 和 `label` 字段，如示例 18-7 所示：

文件名：src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // 实际绘制按钮的代码
    }
}
```

示例 18-7: 一个实现了 `Draw` trait 的 `Button` 结构体

在 `Button` 上的 `width`、`height` 和 `label` 字段会和其他组件不同；比如 `TextField` 可能有 `width`、`height`、`label` 以及 `placeholder` 字段。每一个我们希望能在屏幕上绘制的类型都会使用不同的代码来实现 `Draw` trait 的 `draw` 方法来定义如何绘制特定的类型，像这里的 `Button` 类型（如上提到的并不包含任何实际的 GUI 代码）。除了实现 `Draw` trait 之外，比如 `Button` 还可能有另一个包含按钮点击如何响应的方法的 `impl` 块。这类方法并不适用于像 `TextField` 这样的类型。

如果一些库的使用者决定实现一个包含 `width`、`height` 和 `options` 字段的结构体 `SelectBox`，并且也为其实现了 `Draw` trait，如示例 18-8 所示：

文件名：src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
```

```
        // code to actually draw a select box
    }
}
```

示例 18-8: 另一个使用 `gui` 的 crate，其在 `SelectBox` 结构体上实现 `Draw` trait

库使用者现在可以在他们的 `main` 函数中创建一个 `Screen` 实例。至此可以通过将 `SelectBox` 和 `Button` 放入 `Box<T>` 转变为 trait 对象再放入 `Screen` 实例中。接着可以调用 `Screen` 的 `run` 方法，它会调用每个组件的 `draw` 方法。示例 18-9 展示了这个实现：

文件名：src/main.rs

```
use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

示例 18-9: 使用 trait 对象来存储实现了相同 trait 的不同类型的值

当编写库的时候，我们不知道何人会在何时增加 `SelectBox` 类型，不过 `Screen` 的实现能够操作并绘制这个新类型，因为 `SelectBox` 实现了 `Draw` trait，这意味着它实现了 `draw` 方法。

这个概念 —— 只关心值所反映的信息而不是其具体类型 —— 类似于动态类型语言中称为**鸭子类型**（*duck typing*）的概念：如果它走起来像一只鸭子，叫起来像一只鸭子，那么它就是一只鸭子！在示例 18-5 中 `Screen` 上的 `run` 实现中，`run` 并不需要知道各个组件的具体类型是什么。它并不检查组件是 `Button` 或者 `SelectBox` 的实例，而是直接调用组件的 `draw` 方法。通过指定 `Box<dyn Draw>` 作为 `components` vector 中值的类型，我们就定义了 `Screen` 为需要可以在其上调用 `draw` 方法的值。

使用 trait 对象和 Rust 类型系统来进行类似鸭子类型操作的优势是无需在运行时检查一个值是否实现了特定方法或者担心在调用时因为值没有实现方法而产生错误。如果值没有实现 trait 对象所需的 trait 则 Rust 不会编译这些代码。

例如，示例 18-10 展示了当创建一个使用 `String` 做为其组件的 `Screen` 时发生的情况：

文件名：src/main.rs



```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };

    screen.run();
}
```

示例 18-10: 尝试使用一种没有实现 trait 对象的 trait 的类型

我们会遇到这个错误因为 `String` 没有实现 `Draw` trait：

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
 --> src/main.rs:5:26
  |
5 |         components: vec![Box::new(String::from("Hi"))],
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is not
implemented for `String`
  |
  = help: the trait `Draw` is implemented for `Button`
  = note: required for the cast from `Box<String>` to `Box<dyn Draw>`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` (bin "gui") due to 1 previous error
```

这告诉了我们，要么是我们传递了并不希望传递给 `Screen` 的类型并应该提供其他类型，要么应该在 `String` 上实现 `Draw` 以便 `Screen` 可以调用其上的 `draw`。

trait 对象执行动态分发

回忆一下第十章 “泛型代码的性能” 部分讨论过的，当对泛型使用 trait bound 时编译器所执行的单态化处理：编译器为每一个被泛型类型参数代替的具体类型生成了函数和方法的非泛型实现。单态化产生的代码在执行**静态分发**（*static dispatch*），也就是说编译器在编译时就知晓要调用什么方法。这与**动态分发**（*dynamic dispatch*）相对，这时编译器在编译时无法知晓要调用哪个方法。在动态分发的场景下，编译器会生成负责在运行时确定该调用什么方法的代码。

当使用 trait 对象时，Rust 必须使用动态分发。编译器无法知晓所有可能用于 trait 对象代码的类型，所以它也不知道应该调用哪个类型的哪个方法实现。为此，Rust 在运行时使用 trait 对象中的指针来知晓需要调用哪个方法。这种查找会带来在静态分发中不会产生的运行时开销。动态分发也阻止编译器有选择地内联方法代码，这会相应地禁用一些优化，Rust 还定义了一些规则，称为 **dyn 兼容性**（*dyn compatibility*），用于规定可以和不可以在哪些地方使用动态分发。这些规则超出了本讨论范围，但你可以在参考资料中详细了解。尽管在编写示例 18-5 和可以支持示例 18-9 中的代码的过程中确实获得了额外的灵活性，但仍然需要权衡取舍。

面向对象设计模式的实现

状态模式（*state pattern*）是一个面向对象设计模式。该模式的关键在于定义值的一系列内含状态。这些状态体现为一系列的**状态对象**（*state objects*），同时值的行为随着其内部状态而改变。我们将编写一个博客发布结构体的例子，它拥有一个包含其状态的字段，该字段可以是“draft”、“review” 或 “published” 状态对象之一。

状态对象共享功能：当然，在 Rust 中使用结构体和 trait 而不是对象和继承。每一个状态对象负责其自身的行为，以及该状态何时应当转移至另一个状态。持有一个状态对象的值对于不同状态的行为以及何时状态转移毫不知情。

使用状态模式的优点在于，程序的业务需求改变时，无需改变值持有状态或者使用值的代码。我们只需更新某个状态对象中的代码来改变其规则，或者是增加更多的状态对象。

首先我们将以一种更加传统的面向对象的方式实现状态模式，接着使用一种在 Rust 中更自然的方式。让我们使用状态模式来增量式地实现一个发布博文的工作流以探索这个概念。

最终功能看起来像这样：

1. 博文从空白的草稿开始。
2. 一旦草稿完成，请求审核博文。
3. 一旦博文过审，它将被发表。
4. 只有被发表的博文的内容会被打印，这样就不会意外打印出没有被审核的博文的文本。

任何其他对博文的修改尝试都不会生效。例如，如果尝试在请求审核之前通过一个草稿博文，博文应该保持未发布的状态。

示例 18-11 展示这个工作流的代码形式：这是一个我们将要在一个叫做 `blog` 的库 crate 中实现的 API 的示例。这段代码还不能编译，因为还未实现 `blog` crate。

文件名：src/main.rs



```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

示例 18-11: 展示了 `blog` crate 期望行为的代码

我们希望允许用户使用 `Post::new` 创建一个新的博文草稿。也希望能在草稿阶段为博文编写一些文本。如果在审批之前尝试立刻获取博文的内容，不应该获取到任何文本因为博文仍然是草稿。出于演示目的我们在代码中添加了 `assert_eq!`。一个好的单元测试将是断言草稿博文的 `content` 方法返回空字符串，不过我们并不准备为这个例子编写单元测试。

接下来，我们希望能够请求审核博文，而在等待审核的阶段 `content` 应该仍然返回空字符串。最后当博文审核通过，它应该被发表，这意味着当调用 `content` 时博文的文本将被返回。

注意我们与 crate 交互的唯一的类型是 `Post`。这个类型会使用状态模式并会存放处于三种博文所可能的状态之一的值 —— 草稿，审核和发布。状态上的改变由 `Post` 类型内部进行管理。状态依库用户对 `Post` 实例调用的方法而改变，但是不能直接管理状态变化。这也意味着用户不会在状态上犯错，比如在过审前发布博文。

定义 `Post` 并新建一个草稿状态的实例

让我们开始实现这个库吧！我们知道需要一个公有 `Post` 结构体来存放一些文本，所以让我们从结构体的定义和一个创建 `Post` 实例的公有关联函数 `new` 开始，如示例 18-12 所示。还需定义一个私有 trait `State` 用于定义 `Post` 的状态对象所必须有的行为。

`Post` 将在私有字段 `state` 中存放一个 `Option<T>` 类型的 trait 对象 `Box<dyn State>`。稍后将会看到为何 `Option<T>` 是必须的。

文件名：src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

示例 18-12: `Post` 结构体的定义和新建 `Post` 实例的 `new` 函数，`State` trait 和结构体 `Draft`

`State` trait 定义了所有不同状态的博文所共享的行为，这个状态对象是 `Draft`、`PendingReview` 和 `Published`，它们都会实现 `State` trait。现在这个 trait 并没有任何方法，同时开始将只定义 `Draft` 状态因为这是我们希望博文的初始状态。

当创建新的 `Post` 时，我们将其 `state` 字段设置为一个存放了 `Box` 的 `Some` 值。这个 `Box` 指向一个 `Draft` 结构体新实例。这确保了无论何时新建一个 `Post` 实例，它都会从草稿开始。因为 `Post` 的 `state` 字段是私有的，也就无法创建任何其他状态的 `Post` 了！ `Post::new` 函数中将 `content` 设置为新建的空 `String`。

存放博文内容的文本

在示例 18-11 中，展示了我们希望能够调用一个叫做 `add_text` 的方法并向其传递一个 `&str` 来将文本增加到博文的内容中。选择实现为一个方法而不是将 `content` 字段暴露为 `pub` 。这意

味着之后可以实现一个方法来控制 `content` 字段如何被读取。`add_text` 方法是非常直观的，让我们在示例 18-13 的 `impl Post` 块中增加一个实现：

文件名：src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

示例 18-13: 实现方法 `add_text` 来向博文的 `content` 增加文本

`add_text` 获取一个 `self` 的可变引用，因为需要改变调用 `add_text` 的 `Post` 实例。接着调用 `content` 中的 `String` 的 `push_str` 并传递 `text` 参数来将其追加到已保存的 `content` 中。这不是状态模式的一部分，因为它的行为并不依赖博文所处的状态。`add_text` 方法完全不与 `state` 字段交互，不过这是我们希望支持的行为的一部分。

确保博文草稿的内容是空的

即使调用 `add_text` 并向博文增加一些内容之后，我们仍然希望 `content` 方法返回一个空字符串 slice，因为博文仍然处于草稿状态，如示例 18-11 的第 7 行所示。现在让我们使用能满足要求的最简单的方式来实现 `content` 方法：总是返回一个空字符串 slice。当实现了将博文状态改为发布的能力之后将改变这一做法。但是目前博文只能是草稿状态，这意味着其内容应该总是空的。示例 18-14 展示了这个占位符实现。

文件名：src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

列表 18-14: 增加一个 `Post` 的 `content` 方法的占位实现，它总是返回一个空字符串 slice

通过增加这个 `content` 方法，示例 18-11 中直到第 7 行的代码能如期运行。

请求审核来改变博文的状态

接下来需要增加请求审核博文的功能，这应当将其状态由 `Draft` 改为 `PendingReview`。示例 18-15 展示了这个代码：

文件名：src/lib.rs

```
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
```

```
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

示例 18-15: 实现 `Post` 和 `State` trait 的 `request_review` 方法

这里为 `Post` 增加一个获取 `self` 可变引用的公有方法 `request_review`。接着在 `Post` 的当前状态下调用内部的 `request_review` 方法，并且第二个 `request_review` 方法会消费当前的状态并返回一个新状态。

这里给 `State` trait 增加了 `request_review` 方法；所有实现了这个 trait 的类型现在都需要实现 `request_review` 方法。注意不同于使用 `self`、 `&self` 或者 `&mut self` 作为方法的第一个参数，这里使用了 `self: Box<Self>`。这个语法意味着该方法只可在持有这个类型的 `Box` 上被调用。这个语法获取了 `Box<Self>` 的所有权使老状态无效化，以便 `Post` 的状态值可转换为一个新状态。

为了消费老状态，`request_review` 方法需要获取状态值的所有权。这就是 `Post` 的 `state` 字段中 `Option` 的来历：调用 `take` 方法将 `state` 字段中的 `Some` 值取出并留下一个 `None`，因为 Rust 不允许结构体实例中存在未初始化的字段。这使得我们将 `state` 的值移出 `Post` 而不是借用它。接着我们将博文的 `state` 值设置为这个操作的结果。

我们需要将 `state` 临时设置为 `None` 来获取 `state` 值，即老状态的所有权，而不是使用 `self.state = self.state.request_review();` 这样的代码直接更新状态值。这确保了当 `Post` 被转换为新状态后不能再使用老 `state` 值。

`Draft` 的 `request_review` 方法需要返回一个新的，装箱的 `PendingReview` 结构体的实例，其用来代表博文处于等待审核状态。结构体 `PendingReview` 同样也实现了 `request_review` 方法，不过它不进行任何状态转换。相反它返回自身，因为当我们请求审核一个已经处于 `PendingReview` 状态的博文，它应该继续保持 `PendingReview` 状态。

现在我们能看出状态模式的优势了：无论 `state` 是何值，`Post` 的 `request_review` 方法都是一样的。每个状态只负责它自己的规则。

我们将继续保持 `Post` 的 `content` 方法实现不变，返回一个空字符串 slice。现在我们可以拥有 `PendingReview` 状态和 `Draft` 状态的 `Post` 了，不过我们希望在 `PendingReview` 状态下 `Post` 也有相同的行为。现在示例 18-11 中直到 10 行的代码是可以执行的！

添加 `approve` 以改变 `content` 的行为

`approve` 方法将与 `request_review` 方法类似：它会将 `state` 设置为审核通过时应处于的状态，如示例 18-16 所示。

文件名：src/lib.rs

```
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

示例 18-16: 为 `Post` 和 `State` trait 实现 `approve` 方法

这里为 State trait 增加了 approve 方法，并新增了一个实现了 State 的结构体，Published 状态。

类似于 PendingReview 中 request_review 的工作方式，如果对 Draft 调用 approve 方法，并没有任何效果，因为它会返回 self。当对 PendingReview 调用 approve 时，它返回一个新的、装箱的 Published 结构体的实例。Published 结构体实现了 State trait，同时对于 request_review 和 approve 两方法来说，它返回自身，因为在这两种情况博文应该保持 Published 状态。

现在需要更新 Post 的 content 方法。我们希望 content 根据 Post 的当前状态返回值，所以需要 Post 代理一个定义于 state 上的 content 方法，如示例 18-17 所示：

文件名：src/lib.rs



```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(self)
    }
    // --snip--
}
```

示例 18-17: 更新 Post 的 content 方法来委托调用 State 的 content 方法

因为目标是将所有像这样的规则保持在实现了 State 的结构体中，我们将调用 state 中的值的 content 方法并传递博文实例（也就是 self）作为参数。接着返回 state 值的 content 方法的返回值。

这里调用 Option 的 as_ref 方法是因为需要 Option 中值的引用而不是获取其所有权。因为 state 是一个 Option<Box<dyn State>>，调用 as_ref 会返回一个 Option<&Box<dyn State>>。如果不调用 as_ref，将会得到一个错误，因为不能将 state 移动出借用的 &self 函数参数。

接着调用 unwrap 方法，这里我们知道它永远也不会 panic，因为 Post 的所有方法都确保在它们返回时 state 会有一个 Some 值。这就是一个第十二章 “当我们比编译器知道更多的情况” 部分讨论过的我们知道 None 是不可能的而编译器却不能理解的情况之一。

接着我们就有了一个 &Box<dyn State>，当调用其 content 时，解引用强制转换会作用于 & 和 Box ，这样最终会调用实现了 State trait 的类型的 content 方法。这意味着需要为 State trait 定义增加 content，这也是放置根据所处状态返回什么内容的逻辑的地方，如示例 18-18 所示：

文件名：src/lib.rs

```
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}
```

```
impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

示例 18-18: 为 `State` trait 增加 `content` 方法

这里增加了一个 `content` 方法的默认实现来返回一个空字符串 slice。这意味着无需为 `Draft` 和 `PendingReview` 结构体实现 `content` 了。`Published` 结构体会重写 `content` 方法并会返回 `post.content` 的值。

注意这个方法需要生命周期注解，如第十章所讨论的。这里获取 `post` 的引用作为参数，并返回 `post` 一部分的引用，所以返回的引用的生命周期与 `post` 参数相关。

现在示例完成了 —— 现在示例 18-11 中所有的代码都能工作！我们通过发布博文工作流的规则实现了状态模式。围绕这些规则的逻辑都存在于状态对象中而不是分散在 `Post` 之中。

> **为什么不用枚举?**
>
> 你可能会好奇为什么不用包含不同可能的博文状态变体的 `enum` 作为变量。这确实是一个可能的方案；尝试实现并对比最终结果来看看哪一种更适合你！使用枚举的一个缺点是每一个检查枚举值的地方都需要一个 `match` 表达式或类似的代码来处理所有可能的变体。这相比 trait 对象模式可能显得更重复。

状态模式的权衡取舍

我们展示了 Rust 是能够实现面向对象的状态模式的，以便能根据博文所处的状态来封装不同类型的行为。`Post` 的方法并不知道这些不同类型的行为。通过这种组织代码的方式，要找到所有已发布博文的不同行为只需查看一处代码：`Published` 的 `State` trait 的实现。

如果要创建一个不使用状态模式的替代实现，则可能会在 `Post` 的方法中，或者甚至于在 `main` 代码中用到 `match` 语句，来检查博文状态并在这里改变其行为。这意味着需要查看很多位置来理解处于发布状态的博文的所有逻辑！这在增加更多状态时会变得更糟：每一个 `match` 语句都会需要另一个分支。

对于状态模式来说，`Post` 的方法和使用 `Post` 的位置无需 `match` 语句，同时增加新状态只涉及到增加一个新 `struct` 和为其实现 trait 的方法。

这个实现易于扩展增加更多功能。为了体会使用此模式维护代码的简洁性，请尝试如下一些建议：

- 增加 `reject` 方法将博文的状态从 `PendingReview` 变回 `Draft`。
- 在将状态变为 `Published` 之前要求两次 `approve` 调用。
- 只允许博文处于 `Draft` 状态时增加文本内容。提示：让状态对象负责内容可能发生什么改变，但不负责修改 `Post`。

状态模式的一个缺点是因为状态实现了状态之间的转换，一些状态会相互联系。如果在 `PendingReview` 和 `Published` 之间增加另一个状态，比如 `Scheduled`，则不得不修改

`PendingReview` 中的代码来转移到 `Scheduled`。如果 `PendingReview` 无需因为新增的状态而改变就更好了，不过这意味着切换到另一种设计模式。

另一个缺点是我们会发现一些重复的逻辑。为了消除它们，可以尝试为 `State` trait 中返回 `self` 的 `request_review` 和 `approve` 方法增加默认实现；然而这样做行不通：当将 `State` 用作 trait 对象时，trait 并不知道 `self` 具体是什么类型，因此无法在编译时确定返回类型。（这是前面提到的 dyn 兼容性规则之一。）

另一个重复是 `Post` 中 `request_review` 和 `approve` 这两个类似的实现。它们都会对 `Post` 的 `state` 字段调用 `Option::take`，如果 `state` 为 `Some`，就将调用委托给封装值的同名方法，并将返回结果重新赋值给 `state` 字段。如果 `Post` 中的很多方法都遵循这个模式，我们可能会考虑定义一个宏来消除重复（查看第二十章的 “宏” 部分）。

完全按照面向对象语言的定义实现这个模式并没有尽可能地利用 Rust 的优势。让我们看看一些代码中可以做出的修改，来将无效的状态和状态转移变为编译时错误。

将状态和行为编码为类型

我们将展示如何稍微反思状态模式来进行一系列不同的权衡取舍。不同于完全封装状态和状态转移使得外部代码对其毫不知情，我们将状态编码进不同的类型。如此，Rust 的类型检查就会将任何在只能使用发布博文的地方使用草稿博文的尝试变为编译时错误。

让我们考虑一下示例 18-11 中 `main` 的第一部分：

文件名：src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

我们仍然希望能够使用 `Post::new` 创建一个新的草稿博文，并能够增加博文的内容。不过不同于存在一个草稿博文时返回空字符串的 `content` 方法，我们将使草稿博文完全没有 `content` 方法。这样如果尝试获取草稿博文的内容，将会得到一个方法不存在的编译错误。这使得我们不可能在生产环境意外显示出草稿博文的内容，因为这样的代码甚至就不能编译。示例 18-19 展示了 `Post` 结构体、`DraftPost` 结构体以及各自的方法的定义：

文件名：src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
```

```
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

示例 18-19: 带有 `content` 方法的 `Post` 和没有 `content` 方法的 `DraftPost`

`Post` 和 `DraftPost` 结构体都有一个私有的 `content` 字段来储存博文的文本。这些结构体不再有 `state` 字段因为我们将状态编码改为结构体类型本身。`Post` 将代表发布的博文，它有一个返回 `content` 的 `content` 方法。

仍然有一个 `Post::new` 函数，不过不同于返回 `Post` 实例，它返回 `DraftPost` 的实例。现在不可能创建一个 `Post` 实例，因为 `content` 是私有的同时没有任何函数返回 `Post`。

`DraftPost` 上定义了一个 `add_text` 方法，这样就可以像之前那样向 `content` 增加文本，不过注意 `DraftPost` 并没有定义 `content` 方法！如此现在程序确保了所有博文都从草稿开始，同时草稿博文没有任何可供展示的内容。任何绕过这些限制的尝试都会产生编译错误。

实现状态转移为不同类型的转换

那么如何得到发布的博文呢？我们希望强制执行的规则是草稿博文在可以发布之前必须被审核通过。等待审核状态的博文应该仍然不会显示任何内容。让我们通过增加另一个结构体 `PendingReviewPost` 来实现这个限制，在 `DraftPost` 上定义 `request_review` 方法来返回 `PendingReviewPost`，并在 `PendingReviewPost` 上定义 `approve` 方法来返回 `Post`，如示例 18-20 所示：

文件名：src/lib.rs

```
impl DraftPost {
    // --snip--
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
```

```
        }
    }
}
```

列表 18-20: `PendingReviewPost` 通过调用 `DraftPost` 的 `request_review` 创建，`approve` 方法将 `PendingReviewPost` 变为发布的 `Post`

`request_review` 和 `approve` 方法获取 `self` 的所有权，因此会消费 `DraftPost` 和 `PendingReviewPost` 实例，并分别转换为 `PendingReviewPost` 和发布的 `Post`。这样在调用 `request_review` 之后就不会遗留任何 `DraftPost` 实例，后者同理。`PendingReviewPost` 并没有定义 `content` 方法，所以尝试读取其内容会导致编译错误，`DraftPost` 同理。因为唯一得到定义了 `content` 方法的 `Post` 实例的途径是调用 `PendingReviewPost` 的 `approve` 方法，而得到 `PendingReviewPost` 的唯一办法是调用 `DraftPost` 的 `request_review` 方法，现在我们就将发博文的工作流编码进了类型系统。

这也意味着不得不对 `main` 做出一些小的修改。因为 `request_review` 和 `approve` 返回新实例而不是修改被调用的结构体，所以我们需要增加更多的 `let post =` 遮蔽赋值来保存返回的实例。也不再能断言草稿和等待审核的博文的内容为空字符串了，我们也不再需要它们：不能编译尝试使用这些状态下博文内容的代码。更新后的 `main` 的代码如示例 18-21 所示。

文件名：src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

示例 18-21: `main` 中使用新的博文工作流实现的修改

不得不修改 `main` 来重新赋值 `post` 使得这个实现不再完全遵守面向对象的状态模式：状态间的转换不再完全封装在 `Post` 实现中。然而，得益于类型系统和编译时类型检查，我们得到的收获是无效状态是不可能的了！这确保了某些特定的 bug，比如显示未发布博文的内容，将在部署到生产环境之前被发现。

尝试为示例 18-21 之后的 `blog` crate 实现这一部分开始所建议的任务来体会使用这个版本的代码是何感觉。注意在这个设计中一些需求可能已经完成了。

我们已经看到，虽然 Rust 能够实现面向对象设计模式，但 Rust 还提供了诸如将状态编码进类型系统之类的其他模式。这些模式有着不同的权衡取舍。虽然你可能非常熟悉面向对象模式，重新思考这些问题来利用 Rust 提供的像在编译时避免一些 bug 这样的有益功能。在 Rust 中面向对象模式并不总是最好的解决方案，因为 Rust 拥有像所有权这样的面向对象语言所没有的特性。

总结

阅读本章后，不管你是否认为 Rust 是一个面向对象语言，现在你都见识了 trait 对象是一个 Rust 中获取部分面向对象功能的方法。动态分发可以通过牺牲少量运行时性能来为你的代码提供一些灵活性。这些灵活性可以用来实现有助于代码可维护性的面向对象模式。Rust 也有像所有权这样不同于面向对象语言的特性。面向对象模式并不总是利用 Rust 优势的最好方式，但也是一个可用选项。

接下来，让我们看看另一个提供了多样灵活性的 Rust 功能：模式。我们在全书中已多次简要提及它们，但尚未充分领略它们的全部威力。让我们开始探索吧！

模式与模式匹配

模式（*Patterns*）是 Rust 中一种特殊的语法，它用来匹配类型的结构，无论类型是简单还是复杂。结合使用模式和 `match` 表达式以及其他结构可以提供更多对程序控制流的支配权。模式由如下一些内容组合而成：

- 字面值
- 已解构的数组、枚举、结构体或者元组
- 变量
- 通配符
- 占位符

一些模式的例子包括`x`，`(a, 3)` 和 `Some(Color::Red)`。在模式为有效的上下文中，这些部分描述了数据的形状。接着可以用其匹配值来决定程序是否拥有正确的数据来运行特定部分的代码。

我们通过将一些值与模式相比较来使用它。如果模式匹配这些值，我们对值部分进行相应处理。回忆一下第六章讨论 `match` 表达式时像硬币分类器那样使用模式。如果数据符合这个形状，就可以使用这些命名的片段。如果不符合，与该模式相关的代码则不会运行。

本章是所有模式相关内容的参考。我们将涉及到使用模式的有效位置，*refutable* 与 *irrefutable* 模式的区别，和你可能会见到的不同类型的模式语法。到本章末尾，你就能掌握如何利用模式以清晰的方式表达多种概念。

所有可能会用到模式的位置

模式出现在 Rust 的很多地方。你已经在不经意间使用了很多模式！本节将介绍所有模式有效的位置。

`match` 分支

如第六章所讨论的，一个模式常用的位置是 `match` 表达式的分支。在形式上 `match` 表达式由 `match` 关键字、用于匹配的值和一个或多个分支构成，这些分支包含一个模式和在值匹配分支的模式时运行的表达式，如下所示：

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

例如这是一个来自示例 6-5 中匹配变量 `x` 中 `Option<i32>` 值的 `match` 表达式：

```
match x {
    None => None,
    Some(i) => Some(i + 1),
}
```

这个 `match` 表达式中的模式为每个箭头左边的 `None` 和 `Some(i)`。

`match` 表达式的一个要求是它们必须是**穷尽**（*exhaustive*）的，意为 `match` 表达式所有可能的值都必须被考虑到。一个确保覆盖每个可能值的方法是在最后一个分支使用**捕获所有**的模式：比如，一个匹配任何值的名称永远也不会失败，因此可以覆盖所有匹配剩下的情况。

有一个特定的模式 `_` 可以匹配所有情况，不过它从不绑定任何变量。例如这在希望忽略任何未指定值的情况很有用。本章之后的 “忽略模式中的值” 部分会详细介绍 `_` 模式的更多细节。

`if let` 条件表达式

第六章讨论过了 `if let` 表达式，以及它是如何主要用于编写等同于只关心一个情况的 `match` 语句简写的。`if let` 可以对应一个可选的带有代码的 `else` 在 `if let` 中的模式不匹配时运行。

示例 19-1 展示了也可以组合并匹配 `if let`、`else if` 和 `else if let` 表达式。这相比 `match` 表达式一次只能将一个值与模式比较提供了更多灵活性。并且 Rust 并不要求一系列 `if let`、`else if`、`else if let` 分支的条件相互关联。

示例 19-1 中的代码展示了一系列针对不同条件的检查来决定背景颜色应该是什么。为了达到这个例子的目的，我们创建了硬编码值的变量，真实程序中这些值可能来源于用户输入。

文件名：src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();
```

```
    if let Some(color) = favorite_color {
        println!("Using your favorite color, {color}, as the background");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

示例 19-1: 结合 `if let`、`else if`、`else if let` 以及 `else`

如果用户指定了中意的颜色，将使用其作为背景颜色。如果没有指定中意的颜色且今天是星期二，背景颜色将是绿色。如果用户指定了他们的年龄字符串并能够成功将其解析为数字的话，我们将根据这个数字使用紫色或者橙色。最后，如果没有一个条件符合，背景颜色将是蓝色。

这个条件结构允许我们支持复杂的需求。使用这里硬编码的值，例子会打印出 `Using purple as the background color`。

注意 `if let` 也可以像 `match` 分支那样引入并遮蔽现有变量：`if let Ok(age) = age` 引入了一个新的 `age` 变量，包含 `Ok` 变体中的值，从而遮蔽了之前的 `age` 变量。这意味着 `if age > 30` 条件需要位于这个代码块内部：不能将两个条件组合为 `if let Ok(age) = age && age > 30`，因为我们想与 30 比较的新 `age` 只有在大括号开启的新作用域内才有效。

`if let` 表达式的缺点在于其穷尽性没有为编译器所检查，而 `match` 表达式则检查了。如果去掉最后的 `else` 块而遗漏处理一些情况，编译器也不会警告这类可能的逻辑错误。

`while let` 条件循环

一个与 `if let` 结构类似的是 `while let` 条件循环，它允许只要模式匹配就一直进行 `while` 循环。在示例 19-2 展示了一个 `while let` 循环等待跨线程发送的消息，不过在这个示例中它检查一个 `Result` 而非 `Option`。

```
    let (tx, rx) = std::sync::mpsc::channel();
    std::thread::spawn(move || {
        for val in [1, 2, 3] {
            tx.send(val).unwrap();
        }
    });

    while let Ok(value) = rx.recv() {
        println!("{value}");
    }
```

示例 19-2: 使用 `while let` 循环只要 `rx.recv()` 返回 `Ok` 就打印出其值

这个例子会打印出 `1`、`2` 和 `3`。`recv` 方法从信道的接收端取出第一条消息并返回一个 `Ok(value)`。当在第十六章遇到 `recv` 时，我们直接 unwrap 了错误，或者使用 `for` 循环将其

视为迭代器处理。不过如示例 19-2 所示，我们也可以使用 `while let`，因为 `recv` 方法只要发送端持续产生消息它就一直返回 `Ok`，并在发送端断开连接后产生一个 `Err`。

`for` 循环

在 `for` 循环中，模式是 `for` 关键字直接跟随的值。例如，在 `for x in y` 中，`x` 就是这个模式。示例 19-3 中展示了如何使用 `for` 循环来解构，或拆开一个元组作为 `for` 循环的一部分：

```
    let v = vec!['a', 'b', 'c'];

    for (index, value) in v.iter().enumerate() {
        println!("{value} is at index {index}");
    }
```

列表 19-3: 在 `for` 循环中使用模式来解构元组

示例 19-3 的代码会打印出：

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.52s
     Running `target/debug/patterns`
a is at index 0
b is at index 1
c is at index 2
```

这里使用 `enumerate` 方法适配一个迭代器来产生一个值和其在迭代器中的索引，它们位于一个元组中。第一个产生的值是元组 `(0, 'a')`。当这个值匹配模式 `(index, value)`，`index` 将会是 `0` 而 `value` 将会是 `'a'`，并打印出第一行输出。

`let` 语句

在本章之前，我们只明确讨论过通过 `match` 和 `if let` 使用模式，不过事实上也在别的地方使用过模式，包括 `let` 语句。例如，考虑一下这个直白的 `let` 变量赋值：

```
let x = 5;
```

不过你可能没有发觉，每一次像这样使用 `let` 语句就是在使用模式！`let` 语句更为正式的样子如下：

```
let PATTERN = EXPRESSION;
```

像 `let x = 5;` 这样的语句中变量名位于 `PATTERN` 位置，变量名不过是形式特别朴素的模式。我们将表达式与模式比较，并为任何找到的名称赋值。所以例如 `let x = 5;` 的情况，`x` 是一个代表 “将匹配到的值绑定到变量 x” 的模式。同时因为名称 `x` 是整个模式，这个模式实际上等于 “将任何值绑定到变量 `x`，不管值是什么”。

为了更清楚的理解 `let` 的模式匹配方面的内容，考虑示例 19-4 中使用 `let` 和模式解构一个元组：

```

```

```
    let (x, y, z) = (1, 2, 3);
```

示例 19-4: 使用模式解构元组并一次创建三个变量

这里将一个元组与模式匹配。Rust 会比较值 `(1, 2, 3)` 与模式 `(x, y, z)`，并发现二者具有相同的元素数量，因此匹配成功，于是将 `1` 绑定到 `x`，将 `2` 绑定到 `y`，将 `3` 绑定到 `z`。你可以将这个元组模式看作是将三个独立的变量模式结合在一起。

如果模式中元素的数量不匹配元组中元素的数量，则整个类型不匹配，并会得到一个编译时错误。例如，示例 19-5 展示了尝试用两个变量解构三个元素的元组，这是不行的：

```
    let (x, y) = (1, 2, 3);
```



示例 19-5: 一个错误的模式结构，其中变量的数量不符合元组中元素的数量

尝试编译这段代码会给出如下类型错误：

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^   --------- this expression has type `({integer}, {integer},
{integer})`
  |         |
  |         expected a tuple with 3 elements, found one with 2 elements
  |
  = note: expected tuple `({integer}, {integer}, {integer})`
             found tuple `(_, _)`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

为了修复这个错误，可以使用 `_` 或 `..` 来忽略元组中一个或多个值，如 “忽略模式中的值” 部分所示。如果问题是模式中有太多的变量，则解决方法是通过去掉变量使得变量数与元组中元素数相等。

函数参数

函数参数也可以是模式。列表 19-6 中的代码声明了一个叫做 `foo` 的函数，它获取一个 `i32` 类型的参数 `x`，现在这看起来应该很熟悉：

```
fn foo(x: i32) {
    // code goes here
}
```

列表 19-6: 在参数中使用模式的函数签名

`x` 部分就是一个模式！类似于之前对 `let` 所做的，可以在函数参数中匹配元组。列表 19-7 将传递给函数的元组拆分为各个值：

文件名：src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({x}, {y})");
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

列表 19-7: 一个在参数中解构元组的函数

这会打印出 `Current location: (3, 5)`。值 `&(3, 5)` 会匹配模式 `&(x, y)`，如此 `x` 得到了值 `3`，而 `y` 得到了值 `5`。

因为如第十三章所讲闭包类似于函数，也可以在闭包参数列表中使用模式。

现在我们见过了很多使用模式的方式了，不过模式在每个使用它的地方并不以相同的方式工作；在一些地方，模式必须是 *irrefutable* 的，意味着它们必须匹配所提供的任何值。在另一些情况，它们则可以是 refutable 的。接下来让我们讨论这两个概念。

Refutability（可反驳性）：模式是否会匹配失效

模式有两种形式：refutable（可反驳的）和 irrefutable（不可反驳的）。能匹配任何传递的可能值的模式被称为是**不可反驳的**（*irrefutable*）。一个例子就是 `let x = 5;` 语句中的 `x`，因为 `x` 可以匹配任何值所以不可能会失败。对某些可能的值进行匹配会失败的模式被称为是**可反驳的**（*refutable*）。一个这样的例子便是 `if let Some(x) = a_value` 表达式中的 `Some(x)`；如果变量 `a_value` 中的值是 `None` 而不是 `Some`，那么 `Some(x)` 模式不能匹配。

函数参数、`let` 语句和 `for` 循环只能接受不可反驳的模式，因为当值不匹配时，程序无法进行有意义的操作。`if let` 和 `while let` 表达式可以接受可反驳和不可反驳的模式，但编译器会对不可反驳的模式发出警告，因为根据定义它们旨在处理可能的失败：条件表达式的功能在于它能够根据成功或失败来执行不同的操作。

通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。遇到这些情况，根据代码行为的意图，需要修改模式或者使用模式的结构。

让我们看看一个尝试在 Rust 要求不可反驳模式的地方使用可反驳模式以及相反情况的例子。在示例 19-8 中，有一个 `let` 语句，不过模式被指定为可反驳模式 `Some(x)`。如你所见，这不能编译：

```
    let Some(x) = some_option_value;
```



示例 19-8: 尝试在 `let` 中使用可反驳模式

如果 `some_option_value` 的值是 `None`，其不会成功匹配模式 `Some(x)`，表明这个模式是可反驳的。然而，因为 `let` 对于 `None` 匹配不能产生任何合法的代码，所以 `let` 语句只能接受不可反驳模式。Rust 会在编译时抱怨我们尝试在要求不可反驳模式的地方使用可反驳模式：

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding
 --> src/main.rs:3:9
  |
3 |     let Some(x) = some_option_value;
  |         ^^^^^^^ pattern `None` not covered
  |
  = note: `let` bindings require an "irrefutable pattern", like a `struct` or an
`enum` with only one variant
  = note: for more information, visit https://doc.rust-lang.org/book/ch19-02-
refutability.html
  = note: the matched value is of type `Option<i32>`
help: you might want to use `let else` to handle the variant that isn't matched
  |
3 |     let Some(x) = some_option_value else { todo!() };
  |                                     ++++++++++++++++

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

因为我们没有覆盖（也不可能覆盖！）到模式 `Some(x)` 的每一个可能的值，所以 Rust 会合理地抗议。

为了修复在需要不可反驳模式的地方使用可反驳模式的情况，可以修改使用模式的代码：不同于使用 `let`，可以使用 `if let`。如此，如果模式不匹配，大括号中的代码将被忽略，其余代码保持有效。示例 19-9 展示了如何修复示例 19-8 中的代码。

```
    if let Some(x) = some_option_value else {
        return;
    };
```

示例 19-9: 使用 `if let` 和一个带有可反驳模式的代码块来代替 `let`

我们给代码留了一条后路！现在这段代码已经完全有效了。然而，如果我们给 `if let` 提供一个不可反驳模式（即总会匹配的模式），例如示例 19-10 中的 `x`，编译器就会给出警告：

```
    if let x = 5 {
        println!("{x}");
    };
```

示例 19-10: 尝试把不可反驳模式用到 `if let` 上

Rust 会抱怨将不可反驳模式用于 `if let` 是没有意义的：

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
warning: irrefutable `if let` pattern
 --> src/main.rs:2:8
  |
2 |     if let x = 5 {
  |        ^^^^^^^^^
  |
  = note: this pattern will always match, so the `if let` is useless
  = help: consider replacing the `if let` with a `let`
  = note: `#[warn(irrefutable_let_patterns)]` on by default

warning: `patterns` (bin "patterns") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/patterns`
```

基于此，`match` 匹配分支必须使用可反驳模式，除了最后一个分支需要使用能匹配任何剩余值的不可反驳模式。Rust 允许我们在只有一个匹配分支的`match` 中使用不可反驳模式，不过这么做不是特别有用，并可以被更简单的 `let` 语句替代。

目前我们已经讨论了所有可以使用模式的地方，以及可反驳模式与不可反驳模式的区别，下面让我们一起去把可以用来创建模式的语法过目一遍吧。

模式语法

在本节中，我们收集了模式中所有有效的语法，并讨论为什么以及何时你可能要使用这些语法。

匹配字面值

如第六章所示，可以直接匹配字面值模式。如下代码给出了一些例子：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这段代码会打印 `one` 因为 `x` 的值是 1。如果希望代码获得特定的具体值，则该语法很有用。

匹配命名变量

命名变量（Named variables）是匹配任何值的不可反驳模式，这在之前已经使用过数次。然而，当在 `match`、`if let` 或 `while let` 表达式中使用命名变量时，会出现一些复杂情况。由于这些表达式会开始一个新作用域，作为模式一部分在表达式内部声明的变量会遮蔽外部同名变量，这与所有变量的遮蔽规则一致。在示例 19-11 中，声明了一个值为 `Some(5)` 的变量 `x` 和一个值为 `10` 的变量 `y`。接着在值 `x` 上创建了一个 `match` 表达式。观察匹配分支中的模式和结尾的 `println!`，并在运行此代码或进一步阅读之前推断这段代码会打印什么。

文件名：src/main.rs

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {y}"),
    _ => println!("Default case, x = {x:?}"),
}

println!("at the end: x = {x:?}, y = {y}");
```

示例 19-11: 一个 `match` 语句其中一个分支引入了遮蔽变量 `y`

让我们看看当 `match` 语句运行的时候发生了什么。第一个匹配分支的模式并不匹配 `x` 中定义的值，所以代码继续执行。

第二个匹配分支中的模式引入了一个新变量 `y`，它会匹配任何 `Some` 中的值。因为我们在 `match` 表达式的新作用域中，这是一个新变量，而不是开头声明为值 10 的那个 `y`。这个新的 `y` 绑定会匹配任何 `Some` 中的值，在这里是 `x` 中的值。因此这个 `y` 绑定了 `x` 中 `Some` 内部的值。这个值是 5，所以这个分支的表达式将会执行并打印出 `Matched, y = 5`。

如果 `x` 的值是 `None` 而不是 `Some(5)`，头两个分支的模式不会匹配，所以会匹配下划线。这个分支的模式中没有引入变量 `x`，所以此时表达式中的 `x` 会是外部没有被遮蔽的 `x`。在这个假想的例子中，`match` 将会打印 `Default case, x = None`。

一旦 `match` 表达式执行完毕，其作用域也就结束了，同理内部 `y` 的作用域也结束了。最后的 `println!` 会打印 `at the end: x = Some(5), y = 10`。

为了创建能够比较外部 `x` 和 `y` 的值，又不引入新的变量去遮蔽已有 `y` 的 `match` 表达式，我们需要相应地使用带有条件的匹配守卫（match guard）。我们稍后将在 “匹配守卫提供的额外条件” 这一小节讨论匹配守卫。

多个模式

在 `match` 表达式中，可以使用 `|` 语法匹配多个模式，它代表 **或**（*or*）运算符模式。例如，如下代码将 `x` 的值与匹配分支相比较，第一个分支有**或**选项，意味着如果 `x` 的值匹配此分支的任一个值，它就会运行：

```
    let x = 1;

    match x {
        1 | 2 => println!("one or two"),
        3 => println!("three"),
        _ => println!("anything"),
    }
```

上面的代码会打印 `one or two`。

通过 `..=` 匹配值范围

`..=` 语法允许你匹配一个闭区间范围（range）内的值。在如下代码中，当模式匹配任何在给定范围内的值时，该分支会执行：

```
    let x = 5;

    match x {
        1..=5 => println!("one through five"),
        _ => println!("something else"),
    }
```

如果 `x` 是 1、2、3、4 或 5，第一个分支就会匹配。这个语法在匹配多个值时相比使用 `|` 运算符来表达相同的意思更为方便；如果使用 `|` 则不得不指定 `1 | 2 | 3 | 4 | 5`。相反指定范围就简短的多，特别是在希望匹配比如从 1 到 1000 的数字的时候！

编译器会在编译时检查范围不为空，而 `char` 和数字值是 Rust 仅有的可以判断范围是否为空的类型，所以范围只允许用于数字或 `char` 值。

如下是一个使用 `char` 类型值范围的例子：

```
    let x = 'c';

    match x {
        'a'..='j' => println!("early ASCII letter"),
```

```
        'k'..='z' => println!("late ASCII letter"),
        _ => println!("something else"),
    }
```

Rust 知道 `'c'` 位于第一个模式的范围内，并会打印出 `early ASCII letter`。

解构并分解值

也可以使用模式来解构结构体、枚举和元组，以便使用这些值的不同部分。让我们来分别看一看。

解构结构体

示例 19-12 展示带有两个字段 `x` 和 `y` 的结构体 `Point`，可以通过带有模式的 `let` 语句将其分解：

文件名：src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

示例 19-12: 解构一个结构体的字段为单独的变量

这段代码创建了变量 `a` 和 `b` 来匹配结构体 `p` 中的 `x` 和 `y` 字段。这个例子展示了模式中的变量名不必与结构体中的字段名一致。不过通常希望变量名与字段名一致以便于理解变量来自于哪些字段。因为变量名匹配字段名是常见的，同时因为 `let Point { x: x, y: y } = p;` 包含了很多重复，所以对于匹配结构体字段的模式存在简写：只需列出结构体字段的名称，则模式创建的变量会有相同的名称。示例 19-13 展示了与示例 19-12 有着相同行为的代码，不过 `let` 模式创建的变量为 `x` 和 `y` 而不是 `a` 和 `b`：

文件名：src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
```

```
    assert_eq!(7, y);
}
```

示例 19-13: 使用结构体字段简写来解构结构体字段

这段代码创建了变量 `x` 和 `y`，与变量 `p` 中的 `x` 和 `y` 相匹配。其结果是变量 `x` 和 `y` 包含结构体 `p` 中的值。

也可以使用字面值作为结构体模式的一部分进行解构，而不是为所有的字段创建变量。这允许我们测试一些字段为特定值的同时创建其他字段的变量。

示例 19-14 展示了一个 `match` 语句将 `Point` 值分成了三种情况：直接位于 `x` 轴上（此时 `y = 0` 为真）、位于 `y` 轴上（`x = 0`）或不在任何轴上的点。

文件名：src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {x}"),
        Point { x: 0, y } => println!("On the y axis at {y}"),
        Point { x, y } => {
            println!("On neither axis: ({x}, {y})");
        }
    }
}
```

示例 19-14: 解构和匹配模式中的字面值

第一个分支通过指定字段 `y` 匹配字面值 `0` 来匹配任何位于 `x` 轴上的点。此模式仍然创建了变量 `x` 以便在分支的代码中使用。

类似的，第二个分支通过指定字段 `x` 匹配字面值 `0` 来匹配任何位于 `y` 轴上的点，并为字段 `y` 创建了变量 `y`。第三个分支没有指定任何字面值，所以其会匹配任何其他的 `Point` 并为 `x` 和 `y` 两个字段创建变量。

在这个例子中，值 `p` 因为其 `x` 包含 `0` 而匹配第二个分支，因此会打印出 `On the y axis at 7`。

记住 `match` 表达式一旦找到一个匹配的模式就会停止检查其它分支，所以即使 `Point { x: 0, y: 0}` 在 `x` 轴上也在 `y` 轴上，这些代码也只会打印 `On the x axis at 0`。

解构枚举

本书之前曾经解构过枚举（例如第六章示例 6-5），不过当时没有明确提到解构枚举的模式需要对应枚举所定义的储存数据的方式。让我们以示例 6-2 中的 `Message` 枚举为例，编写一个 `match` 使用模式解构每一个内部值，如示例 19-15 所示：

文件名：src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
```

```
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.");
        }
        Message::Move { x, y } => {
            println!("Move in the x direction {x} and in the y direction {y}");
        }
        Message::Write(text) => {
            println!("Text message: {text}");
        }
        Message::ChangeColor(r, g, b) => {
            println!("Change color to red {r}, green {g}, and blue {b}");
        }
    }
}
```

示例 19-15: 解构包含不同类型值变体的枚举

这段代码会打印出 `Change the color to red 0, green 160, and blue 255`。尝试改变 `msg` 的值来观察其他分支代码的运行。

对于像 `Message::Quit` 这样没有任何数据的枚举变体，不能进一步解构其值。只能匹配其字面值 `Message::Quit`，因此模式中没有任何变量。

对于像 `Message::Move` 这样的类结构体枚举变体，可以采用类似于匹配结构体的模式。在变体名称后，使用大括号并列出字段变量以便将其分解以供此分支的代码使用。这里使用了示例 19-13 所展示的简写。

对于像 `Message::Write` 这样的包含一个元素，以及像 `Message::ChangeColor` 这样包含三个元素的类元组枚举变体，其模式则类似于用于解构元组的模式。模式中变量的数量必须与变体中元素的数量完全一致。

解构嵌套的结构体和枚举

目前为止，所有的例子都只匹配了深度为一级的结构体或枚举，不过当然也可以匹配嵌套的项！例如，我们可以重构列表 19-15 的代码在 `ChangeColor` 消息中同时支持 RGB 和 HSV 色彩模式，如示例 19-16 所示：

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
```

```
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!("Change color to red {r}, green {g}, and blue {b}");
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!("Change color to hue {h}, saturation {s}, value {v}");
        }
        _ => (),
    }
}
```

示例 19-16: 匹配嵌套的枚举

`match` 表达式第一个分支的模式匹配一个包含 `Color::Rgb` 枚举变体的 `Message::ChangeColor` 枚举变体，然后模式绑定了三个内部的 `i32` 值。第二个分支的模式也匹配一个 `Message::ChangeColor` 枚举变体，但是其内部的枚举会匹配 `Color::Hsv` 枚举变体。我们可以在一个 `match` 表达式中指定这些复杂条件，即使会涉及到两个枚举。

解构结构体和元组

甚至可以用复杂的方式来混合、匹配和嵌套解构模式。如下是一个复杂结构体的例子，其中结构体和元组嵌套在元组中，并将所有的原始类型解构出来：

```
    let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

这将复杂的类型分解成部分组件以便可以单独使用我们感兴趣的值。

通过模式解构是一个方便将值的各个片段分离开来单独使用的方式，比如结构体中每个单独字段的值。

忽略模式中的值

有时忽略模式中的一些值是有用的，比如 `match` 中最后捕获全部情况的分支实际上没有做任何事，但是它确实负责匹配了所有剩余的可能值。有一些方法可以忽略模式中全部或部分值：使用 `_` 模式（我们已经见过了），在另一个模式中使用 `_` 模式，使用一个以下划线开始的名称，或者使用 `..` 忽略所剩部分的值。让我们来分别探索如何以及为什么要这么做。

使用 `_` 忽略整个值

我们已经使用过下划线作为匹配但不绑定任何值的通配符模式了。虽然这作为 `match` 表达式最后的分支特别有用，也可以将其用于任意模式，包括函数参数中，如示例 19-17 所示：

文件名：src/main.rs

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {y}");
}
```

```
fn main() {
    foo(3, 4);
}
```

示例 19-17: 在函数签名中使用 `_`

这段代码会完全忽略作为第一个参数传递的值 `3`，并会打印出 `This code only uses the y parameter: 4`。

大部分情况当你不再需要特定函数参数时，最好修改签名不再包含无用的参数。在一些情况下忽略函数参数会变得特别有用，比如实现 trait 时，当你需要特定类型签名但是函数实现并不需要某个参数时。这样可以避免一个存在未使用的函数参数的编译警告，就跟使用命名参数一样。

使用嵌套的 `_` 忽略部分值

也可以在一个模式内部使用 `_` 忽略部分值，例如，当只需要测试部分值但在期望运行的代码中没有用到其他部分时。示例 19-18 展示了负责管理设置值的代码。业务需求是用户不允许覆盖现有的自定义设置，但是可以取消设置，也可以在当前未设置时为其提供一个值。

```
    let mut setting_value = Some(5);
    let new_setting_value = Some(10);

    match (setting_value, new_setting_value) {
        (Some(_), Some(_)) => {
            println!("Can't overwrite an existing customized value");
        }
        _ => {
            setting_value = new_setting_value;
        }
    }

    println!("setting is {setting_value:?}");
```

示例 19-18: 当不需要 `Some` 中的值时在模式内使用下划线来匹配 `Some` 变体

这段代码会打印出 `Can't overwrite an existing customized value` 接着是 `setting is Some(5)`。在第一个匹配分支，我们不需要匹配或使用任一个 `Some` 变体中的值，但需要检测 `setting_value` 和 `new_setting_value` 是否均为 `Some` 变体。在这种情况下，我们打印出为何不改变 `setting_value`，并且不会改变它。

对于所有其他情况（`setting_value` 或 `new_setting_value` 任一为 `None`），这由第二个分支的 `_` 模式体现，这时确实希望允许 `new_setting_value` 变为 `setting_value`。

也可以在一个模式中的多处使用下划线来忽略特定值，如示例 19-19 所示，这里忽略了一个五元元组中的第二和第四个值：

```
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, _, third, _, fifth) => {
            println!("Some numbers: {first}, {third}, {fifth}");
```

```
        }
    }
```

示例 19-19: 忽略元组的多个部分

这会打印出 `Some numbers: 2, 8, 32`，值 `4` 和 `16` 会被忽略。

通过在变量名开头加 `_` 来忽略未使用的变量

如果你创建了一个变量却不在任何地方使用它，Rust 通常会给你一个警告，因为未使用的变量可能会是个 bug。但是有时创建一个还未使用的变量是有用的，比如你正在设计原型或刚刚开始一个项目。这时你希望告诉 Rust 不要警告未使用的变量，为此可以用下划线作为变量名的开头。示例 19-20 中创建了两个未使用变量，不过当编译代码时只会得到其中一个的警告：

文件名：src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

示例 19-20: 以下划线开始变量名以便去掉未使用变量警告

这里得到了警告说未使用变量 `y`，不过没有警告说未使用 `_x`。

注意，只使用 `_` 和使用以下划线开头的名称有些微妙的不同：比如 `_x` 仍会将值绑定到变量，而 `_` 则完全不会绑定。为了展示这个区别的意义，示例 19-21 会产生一个错误。



```
    let s = Some(String::from("Hello!"));

    if let Some(_s) = s {
        println!("found a string");
    }

    println!("{s:?}");
```

示例 19-21: 以下划线开头的未使用变量仍然会绑定值，它可能会获取值的所有权

我们会得到一个错误，因为 `s` 的值仍然会移动进 `_s`，并阻止我们再次使用 `s`。然而只使用下划线本身，并不会绑定值。示例 19-22 能够无错编译，因为 `s` 没有被移动进 `_`：

```
    let s = Some(String::from("Hello!"));

    if let Some(_) = s {
        println!("found a string");
    }

    println!("{s:?}");
```

示例 19-22: 单独使用下划线不会绑定值

上面的代码能很好的运行；因为没有把 `s` 绑定到任何变量；它没有被移动。

用 `..` 忽略剩余值

对于有多个部分的值，可以使用 `..` 语法来只使用特定部分并忽略其它值，从而避免不得不每一个忽略值列出下划线。`..` 模式会忽略模式中剩余的任何没有显式匹配的值部分。在示例 19-23 中，有一个 `Point` 结构体存放了三维空间中的坐标。在 `match` 表达式中，我们希望只操作 `x` 坐标并忽略 `y` 和 `z` 字段的值：

```
    struct Point {
        x: i32,
        y: i32,
        z: i32,
    }

    let origin = Point { x: 0, y: 0, z: 0 };

    match origin {
        Point { x, .. } => println!("x is {x}"),
    }
```

示例 19-23: 通过使用 `..` 来忽略 `Point` 中除 `x` 以外的字段

这里列出了 `x` 值，接着仅仅包含了 `..` 模式。这比不得不列出 `y: _` 和 `z: _` 要来得简单，特别是在处理有很多字段的结构体，但只涉及一到两个字段时的情形。

`..` 会扩展为所需要的值的数量。示例 19-24 展示了如何在元组中使用 `..`：

文件名：src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {first}, {last}");
        }
    }
}
```

示例 19-24: 只匹配元组中的第一个和最后一个值并忽略掉所有其它值

这里用 `first` 和 `last` 来匹配第一个和最后一个值。`..` 将匹配并忽略中间的所有值。

然而使用 `..` 必须是无歧义的。如果期望匹配和忽略的值是不明确的，Rust 会报错。示例 19-25 展示了一个带有歧义的 `..` 例子，因此其不能编译：

文件名：src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {second}")
        },
```

```
    }
}
```



示例 19-25: 尝试以有歧义的方式运用 `..`

当编译这个示例时，会得到如下错误：

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |          --          ^^ can only be used once per tuple pattern
  |          |
  |          previously used here

error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

Rust 不可能决定在元组中匹配 `second` 值之前应该忽略多少个值，以及在之后忽略多少个值。这段代码可能表明我们意在忽略 `2`，绑定 `second` 为 `4`，接着忽略 `8`、`16` 和 `32`；抑或是意在忽略 `2` 和 `4`，绑定 `second` 为 `8`，接着忽略 `16` 和 `32`，以此类推。变量名 `second` 对于 Rust 来说并没有任何特殊意义，所以会得到编译错误，因为在这两个地方使用 `..` 是有歧义的。

匹配守卫提供的额外条件

匹配守卫（*match guard*）是一个指定于 `match` 分支模式之后的额外 `if` 条件，它也必须被满足才能选择此分支。匹配守卫用于表达比单独的模式所能允许的更为复杂的情况。但是注意，它们仅在 `match` 表达式中可用，不能用于 `if let` 或 `while let` 表达式。

这个条件可以使用模式中创建的变量。示例 19-26 展示了一个 `match`，其中第一个分支有模式 `Some(x)` 还有匹配守卫 `if x % 2 == 0` (当 `x` 是偶数时为真)：

```
    let num = Some(4);

    match num {
        Some(x) if x % 2 == 0 => println!("The number {x} is even"),
        Some(x) => println!("The number {x} is odd"),
        None => (),
    }
```

示例 19-26: 在模式中加入匹配守卫

上例会打印出 `The number 4 is even`。当 `num` 与模式中第一个分支比较时，因为 `Some(4)` 匹配 `Some(x)` 所以可以匹配。接着匹配守卫检查 `x` 除以 `2` 的余数是否等于 `0`，因为它等于 `0`，所以第一个分支被选择。

相反如果 `num` 为 `Some(5)`，因为 `5` 除以 `2` 的余数是 `1` 不等于 `0` 所以第一个分支的匹配守卫为 `false`。接着 Rust 会前往第二个分支，这次匹配因为它没有匹配守卫所以会匹配任何 `Some` 变体。

无法在模式中表达类似 `if x % 2 == 0` 的条件，所以通过匹配守卫提供了表达类似逻辑的能力。这种替代表达方式的缺点是，编译器不会尝试为包含匹配守卫的模式检查穷尽性。

在示例 19-11 中，我们提到可以使用匹配守卫来解决模式中变量遮蔽的问题，那里 `match` 表达式的模式中新建了一个变量而不是使用 `match` 之外的同名变量。新变量意味着不能够测试外部变量的值。示例 19-27 展示了如何使用匹配守卫修复这个问题。

文件名：src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {n}"),
        _ => println!("Default case, x = {x:?}"),
    }

    println!("at the end: x = {x:?}, y = {y}");
}
```

示例 19-27: 使用匹配守卫来测试与外部变量的相等性

现在这会打印出 `Default case, x = Some(5)`。现在第二个匹配分支中的模式不会引入一个遮蔽外部 `y` 的新变量 `y`，这意味着可以在匹配守卫中使用外部的 `y`。相比指定会遮蔽外部 `y` 的模式 `Some(y)`，这里指定为 `Some(n)`。此新建的变量 `n` 并没有覆盖任何值，因为 `match` 外部没有变量 `n`。

匹配守卫 `if n == y` 并不是一个模式所以没有引入新变量。这个 `y` **正是**外部的 `y` 而不是新的遮蔽变量 `y`，这样就可以通过比较 `n` 和 `y` 来表达寻找一个与外部 `y` 相同的值了。

也可以在匹配守卫中使用**或**运算符 `|` 来指定多个模式，同时匹配守卫的条件会作用于所有的模式。示例 19-28 展示了结合匹配守卫与使用了 `|` 的模式的优先级。这个例子中重要的部分是匹配守卫 `if y` 作用于 `4`、`5` **和** `6`，即使这看起来好像 `if y` 只作用于 `6`：

```
    let x = 4;
    let y = false;

    match x {
        4 | 5 | 6 if y => println!("yes"),
        _ => println!("no"),
    }
```

示例 19-28: 结合多个模式与匹配守卫

这个匹配条件表明此分支值匹配 `x` 值为 `4`、`5` 或 `6` **同时** `y` 为 `true` 的情况。运行这段代码时会发生的是第一个分支的模式因 `x` 为 `4` 而匹配，不过匹配守卫 `if y` 为 `false`，所以第一个分支不会被选择。代码移动到第二个分支，这会匹配，此程序会打印出 `no`。这是因为 `if` 条件作用于整个 `4 | 5 | 6` 模式，而不仅是最后的值 `6`。换句话说，匹配守卫与模式的优先级关系看起来像这样：

```
(4 | 5 | 6) if y => ...
```

而不是：

```
4 | 5 | (6 if y) => ...
```

运行代码后，优先级行为就很明显了：如果匹配守卫只作用于由 `|` 运算符指定的值列表的最后一个值，这个分支就会匹配且程序会打印出 `yes`。

`@` 绑定

at 运算符（`@`）允许我们在创建一个存放值的变量的同时测试其值是否匹配模式。示例 19-29 展示了一个例子，这里我们希望测试 `Message::Hello` 的 `id` 字段是否位于 `3..=7` 范围内，同时也希望能将其值绑定到 `id_variable` 变量中以便此分支相关联的代码可以使用它。可以将 `id_variable` 命名为 `id`，与字段同名，不过出于示例的目的这里选择了不同的名称。

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable @ 3..=7,
    } => println!("Found an id in range: {id_variable}"),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {id}"),
}
```

示例 19-29: 使用 `@` 在模式中绑定值的同时测试它

上例会打印出 `Found an id in range: 5`。通过在 `3..=7` 之前指定 `id_variable @`，我们捕获了任何匹配此范围的值并同时测试其值匹配这个范围模式。

第二个分支只在模式中指定了一个范围，分支相关代码没有一个包含 `id` 字段实际值的变量。`id` 字段的值可以是 10、11 或 12，不过这个模式的代码并不知情也不能使用 `id` 字段中的值，因为没有将 `id` 值保存进一个变量。

最后一个分支指定了一个没有范围的变量，此时确实拥有可以用于分支代码的变量 `id`。因为这里使用了结构体字段简写语法。不过此分支中没有像头两个分支那样对 `id` 字段的值进行测试：任何值都会匹配该模式。

使用 `@` 可以在一个模式中同时测试和保存变量值。

总结

模式是 Rust 中一个很有用的功能，它有助于我们区分不同类型的数据。当用于 `match` 语句时，Rust 确保模式会包含每一个可能的值，否则程序将不能编译。`let` 语句和函数参数的模式使得这些结构更强大，可以在将值解构为更小部分的同时为变量赋值。可以创建简单或复杂的模式来满足我们的要求。

接下来，在本书倒数第二章中，我们将介绍一些 Rust 众多功能中较为高级的部分。

高级特性

现在我们已经学习了 Rust 编程语言中最常用的部分。在第二十一章开始另一个新项目之前，我们先来了解一些你偶尔可能会遇到，但并不会每天都用的语言特性。你可以将本章作为不经意间遇到未知的内容时的参考。本章将要学习的功能在一些非常特定的场景下非常有用。虽然很少会碰到它们，我们希望确保你了解 Rust 提供的所有功能。

本章将涉及如下内容：

- 不安全 Rust：用于当需要舍弃 Rust 的某些保证并负责手动维持这些保证
- 高级 trait：与 trait 相关的关联类型，默认类型参数，完全限定语法（fully qualified syntax），超（父）trait（supertraits）模式 newtype 模式
- 高级类型：关于 newtype 模式的更多内容，类型别名，never 类型和动态大小类型
- 高级函数和闭包：函数指针和返回闭包
- 宏：定义在编译时定义更多代码的方式

对所有人而言，这都是一个介绍 Rust 迷人特性的宝典！让我们开始探索吧！

不安全 Rust

目前为止讨论过的代码都有 Rust 在编译时会强制执行的内存安全保证。然而，Rust 还隐藏有第二种语言，它不会强制执行这类内存安全保证：这被称为 **不安全 Rust**（*unsafe Rust*）。它与常规 Rust 代码无异，但是会提供额外的超能力。

不安全 Rust 之所以存在，是因为静态分析本质上是保守的。当编译器尝试确定一段代码是否支持某个保证时，拒绝一些合法的程序比接受无效的程序要好一些。这必然意味着有时代码**可能**是合法的，但如果 Rust 编译器没有足够的信息来确定，它将拒绝该代码。在这种情况下，可以使用不安全代码告诉编译器，“相信我，我知道自己在干什么。” 不过千万注意，使用不安全 Rust 风险自担：如果不安全代码出错了，比如解引用空指针，可能会导致不安全的内存使用。

另一个 Rust 存在不安全一面的原因是底层计算机硬件固有的不安全性。如果 Rust 不允许进行不安全操作，那么有些任务则根本完成不了。Rust 需要能够进行像直接与操作系统交互甚至于编写你自己的操作系统这样的底层系统编程。底层系统编程也是 Rust 语言的目标之一。让我们看看不安全 Rust 能做什么，和怎么做。

不安全的超能力

要切换到 `unsafe Rust`，可以使用 `unsafe` 关键字，然后开启一个包含不安全代码的新块。这里有五类可以在不安全 Rust 中进行而不能用于安全 Rust 的操作，它们称之为**不安全的超能力**（**unsafe superpowers**）。这些超能力包括：

- 解引用裸指针
- 调用不安全的函数或方法
- 访问或修改可变静态变量
- 实现不安全 trait
- 访问 `union` 的字段

有一点很重要，`unsafe` 并不会关闭借用检查器或禁用任何其他 Rust 安全检查：如果在不安全代码中使用引用，它仍会被检查。`unsafe` 关键字只是提供了那五个不会被编译器检查内存安全的功能。你仍然能在不安全块中获得某种程度的安全。

再者，`unsafe` 不意味着块中的代码就一定是危险的或者必然导致内存安全问题：其意图在于作为程序员，你将会确保 `unsafe` 块中的代码以有效的方式访问内存。

人难免出错，错误总会发生，不过通过要求这五类不安全操作必须位于标记为 `unsafe` 的块中，就能够知道任何与内存安全相关的错误必定位于 `unsafe` 块内。保持 `unsafe` 块尽可能小；如此当之后调查内存 bug 时就会感谢你自己了。

为了尽可能隔离不安全代码，最好将不安全代码封装进一个安全的抽象并提供安全 API，当我们学习不安全函数和方法时会讨论到。标准库的一部分被实现为在被评审过的不安全代码之上的安全抽象。这个技术防止了 `unsafe` 泄露到所有你或者用户希望使用由 `unsafe` 代码实现的功能的地方，因为使用其安全抽象是安全的。

让我们按顺序依次介绍上述五类超能力，同时我们会看到一些提供不安全代码的安全接口的抽象。

解引用裸指针

回到第四章的"悬垂引用"一节，那里提到了编译器会确保引用总是有效的。不安全 Rust 有两个被称为 **裸指针**（*raw pointers*）的类似于引用的新类型。和引用一样，裸指针是不可变或可变的，分别写作 `*const T` 和 `*mut T`。这里的星号不是解引用运算符；它是类型名称的一部分。在裸指针的上下文中，**不可变** 意味着指针解引用之后不能直接赋值。

裸指针与引用和智能指针的区别在于

- 允许忽略借用规则，可以同时拥有不可变和可变的指针，或多个指向相同位置的可变指针
- 不保证指向有效的内存
- 允许为空
- 不能实现任何自动清理功能

通过去掉 Rust 强加的保证，你可以放弃安全保证以换取性能或使用另一个语言或硬件接口的能力，此时 Rust 的保证并不适用。

示例 20-1 展示了如何创建一个不可变裸指针和一个可变裸指针。

```
    let mut num = 5;

    let r1 = &raw const num;
    let r2 = &raw mut num;
```

示例 20-1: 通过引用创建裸指针

注意这段代码中没有引入 `unsafe` 关键字。可以在安全代码中创建裸指针；只是不能在不安全块之外解引用裸指针，稍后便会看到。

我们通过使用裸指针借用操作符（raw borrow operators）创建裸指针：`&raw const num` 会创建一个 `*const i32` 的不可变裸指针。因为由于我们是直接从一个局部变量创建它们的，因此可以确定这些特定的裸指针是有效的，但是不能对任何裸指针都做出如此假设。

为了演示这一点，接下来我们将创建一个有效性无法确定的裸指针，使用 `as` 进行类型转换而不是使用裸指针借用操作符。示例 20-2 展示了如何创建一个指向任意内存地址的裸指针。尝试使用任意内存是未定义行为：此地址可能有数据也可能没有，编译器可能会优化掉这个内存访问，或者程序可能因段错误（segmentation fault）而终止。通常在有裸指针借用操作符可用的情况下，没有充分的理由编写这样的代码，但这确实是可行的。

```
    let address = 0x012345usize;
    let r = address as *const i32;
```

示例 20-2: 创建指向任意内存地址的裸指针

记得我们说过可以在安全代码中创建裸指针，但不能 **解引用** 裸指针和读取其指向的数据。示例 20-3 中，我们在裸指针上使用了解引用运算符 `*`，该操作需要一个 `unsafe` 块：

```
    let mut num = 5;

    let r1 = &raw const num;
    let r2 = &raw mut num;
```

```
    unsafe {
        println!("r1 is: {}", *r1);
        println!("r2 is: {}", *r2);
    }
```

示例 20-3: 在 `unsafe` 块中解引用裸指针

创建一个指针不会造成任何危害；只有当访问其指向的值时才有可能遇到无效的值。

还需注意示例 20-1 和 20-3 中创建了同时指向相同内存位置 `num` 的裸指针 `*const i32` 和 `*mut i32`。相反如果尝试同时创建 `num` 的不可变和可变引用，代码将无法通过编译，因为 Rust 的所有权规则不允许在拥有任何不可变引用的同时再创建可变引用。通过裸指针，就能够同时创建同一地址的可变指针和不可变指针，若通过可变指针修改数据，则可能造成潜在数据竞争。请多加小心！

既然存在这么多的危险，为何还要使用裸指针呢？一个主要的应用场景便是调用 C 代码接口，这在下一部分 “调用不安全函数或方法” 中会讲到。另一个场景是构建借用检查器无法理解的安全抽象。让我们先介绍不安全函数，接着看一看使用不安全代码的安全抽象的示例。

调用不安全函数或方法

第二类可以在不安全块中进行的操作是调用不安全函数。不安全函数和方法与常规函数方法十分类似，除了其开头有一个额外的 `unsafe`。在此上下文中，关键字 `unsafe` 表示该函数具有调用时需要满足的要求，而 Rust 不会保证满足这些要求。通过在 `unsafe` 块中调用不安全函数，表明我们已经阅读过此函数的文档并对其是否满足函数自身的契约负责。

如下是一个没有做任何操作的不安全函数 `dangerous` 的例子：

```
    unsafe fn dangerous() {}

    unsafe {
        dangerous();
    }
```

必须在一个单独的 `unsafe` 块中调用 `dangerous` 函数。如果尝试不使用 `unsafe` 块调用 `dangerous`，则会得到一个错误：

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function `dangerous` is unsafe and requires unsafe
block
 --> src/main.rs:4:5
  |
4 |     dangerous();
  |     ^^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how to avoid
undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` (bin "unsafe-example") due to 1 previous
error
```

通过 `unsafe` 块，我们向 Rust 断言我们已经阅读过函数的文档，理解如何正确使用它，并核实我们履行了该函数的契约。

在不安全函数的函数体内部执行不安全操作时，同样需要使用 `unsafe` 块，就像在普通函数中一样，如果忘记了编译器会发出警告（warning）。这有助于将 `unsafe` 块保持得尽可能小，因为 `unsafe` 操作并不一定需要覆盖整个函数体。

> 译注：不安全函数体也是有效的 `unsafe` 块，所以在不安全函数中进行另一个不安全操作时可以不新增额外的 `unsafe` 块，但从 2024 edition 开始，`#[warn(unsafe_op_in_unsafe_fn)]` 是默认开启的，所以此时会产生警告（warning）。参考 RFC 2585，出于尽量缩小 `unsafe` 块的动机，不再建议在不安全函数体中直接进行不安全操作，因为未来可能将这种行为视为错误（error）。

创建不安全代码的安全抽象

仅仅因为函数包含不安全代码并不意味着整个函数都需要标记为不安全的。事实上，将不安全代码封装进安全函数是一种常见的抽象方式。作为一个例子，了解一下标准库中的函数 `split_at_mut`，它需要一些不安全代码，让我们探索可以如何实现它。这个安全函数定义于可变 slice 之上：它获取一个 slice 并从给定的索引参数开始将其分割为两个 slice。示例 20-4 展示了如何使用 `split_at_mut`。

```
    let mut v = vec![1, 2, 3, 4, 5, 6];

    let r = &mut v[..];

    let (a, b) = r.split_at_mut(3);

    assert_eq!(a, &mut [1, 2, 3]);
    assert_eq!(b, &mut [4, 5, 6]);
```

示例 20-4: 使用安全的 `split_at_mut` 函数

这个函数无法只通过安全 Rust 实现。一个尝试可能看起来像示例 20-5，它不能编译。出于简单考虑，我们将 `split_at_mut` 实现为函数而不是方法，并只处理 `i32` 值而非泛型 `T` 的 slice。

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```



示例 20-5: 尝试只使用安全 Rust 来实现 `split_at_mut`

此函数首先获取 slice 的长度，然后通过检查参数是否小于或等于这个长度来断言参数所给定的索引位于 slice 当中。该断言意味着如果传入的索引比要分割的 slice 的索引更大，此函数在尝试使用这个索引前 panic。

之后我们在一个元组中返回两个可变的 slice：一个从原始 slice 的开头直到 `mid` 索引，另一个从 `mid` 直到原 slice 的结尾。

如果尝试编译示例 20-5 的代码，会得到一个错误：

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
 --> src/main.rs:6:31
  |
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
  |                         - let's call the lifetime of this reference `'1`
...
6 |     (&mut values[..mid], &mut values[mid..])
  |     -------------------------^^^^^^--------
  |     |     |                  |
  |     |     |                  second mutable borrow occurs here
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*values` is borrowed for `'1`
  |
  = help: use `.split_at_mut(position)` to obtain two mutable non-overlapping sub-
slices

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` (bin "unsafe-example") due to 1 previous
error
```

Rust 的借用检查器无法理解我们要借用这个 slice 的两个不同部分：它只知道我们借用了同一个 slice 两次。本质上借用 slice 的不同部分是可以的，因为这两段 slice 不会重叠，不过 Rust 还没有智能到能够理解这些。当我们知道某些事是可以的而 Rust 不知道的时候，就是触及不安全代码的时候了

示例 20-6 展示了如何使用 `unsafe` 块，裸指针和一些不安全函数调用来实现 `split_at_mut`：

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

示例 20-6: 在 `split_at_mut` 函数的实现中使用不安全代码

回忆第四章的“Slice 类型”部分，slice 是一个指向一些数据的指针，并带有该 slice 的长度。可以使用 `len` 方法获取 slice 的长度，使用 `as_mut_ptr` 方法访问 slice 的裸指针。在这个例子

中，因为有一个 `i32` 值的可变 slice，`as_mut_ptr` 返回一个 `*mut i32` 类型的裸指针，并将其存储在 `ptr` 变量中。

我们保持索引 `mid` 位于 slice 中的断言。接着是不安全代码：`slice::from_raw_parts_mut` 函数获取一个裸指针和一个长度来创建一个 slice。这里使用此函数从 `ptr` 中创建了一个有 `mid` 个项的 slice。之后在 `ptr` 上调用 `add` 方法并使用 `mid` 作为参数来获取一个从 `mid` 开始的裸指针，使用这个裸指针并以 `mid` 之后项的数量为长度创建另一个 slice。

`slice::from_raw_parts_mut` 函数是不安全的因为它获取一个裸指针，并必须确信这个指针是有效的。裸指针上的 `add` 方法也是不安全的，因为其必须确信此地址偏移量也是有效的指针。因此必须将 `slice::from_raw_parts_mut` 和 `add` 放入 `unsafe` 块中以便能调用它们。通过观察代码，和增加 `mid` 必然小于等于 `len` 的断言，我们可以说 `unsafe` 块中所有的裸指针将是有效的 slice 中数据的指针。这是一个可以接受的 `unsafe` 的恰当用法。

注意无需将 `split_at_mut` 函数的结果标记为 `unsafe`，并可以在安全 Rust 中调用此函数。我们创建了一个不安全代码的安全抽象，其代码以一种安全的方式使用了 `unsafe` 代码，因为其只从这个函数访问的数据中创建了有效的指针。

与此相对，示例 20-7 中的 `slice::from_raw_parts_mut` 在使用 slice 时很有可能会崩溃。这段代码获取任意内存地址并创建了一个长度为一万的 slice：

```
    use std::slice;

    let address = 0x01234usize;
    let r = address as *mut i32;

    let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

示例 20-7: 通过任意内存地址创建 slice

我们并不拥有这个任意地址的内存，也不能保证这段代码创建的 slice 包含有效的 `i32` 值。试图使用臆测为有效的 `values` 会导致未定义的行为。

使用 `extern` 函数调用外部代码

有时你的 Rust 代码可能需要与其他语言编写的代码交互。为此 Rust 有一个关键字，`extern`，有助于创建和使用 **外部函数接口**（*Foreign Function Interface*，FFI）。外部函数接口是一个编程语言用以定义函数的方式，其允许不同（外部）编程语言调用这些函数。

示例 20-8 展示了如何集成 C 标准库中的 `abs` 函数。`extern` 块中声明的函数在 Rust 代码中通常是不安全的因此 `extern` 块本身也必须标注 `unsafe`。之所以如此，是因为其他语言不会强制执行 Rust 的规则，Rust 也无法检查这些约束，因此程序员有责任确保调用的安全性。

文件名：src/main.rs

```
unsafe extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
```

```
    }
}
```

示例 20-8: 声明并调用另一个语言中定义的 `extern` 函数

在 `unsafe extern "C"` 块中，我们列出了希望能够调用的另一个语言中的外部函数的签名和名称。`"C"` 部分定义了外部函数所使用的 **应用二进制接口**（*application binary interface*，ABI）—— ABI 定义了如何在汇编语言层面调用此函数。`"C"` ABI 是最常见的，并遵循 C 编程语言的 ABI。有关 Rust 支持的所有 ABI 的信息请参见 the Rust Reference。

`unsafe extern` 中声明的任何项都隐式地是 `unsafe` 的。然而，一些 FFI 函数**可以**安全地调用。例如，C 标准库中的 `abs` 函数没有任何内存安全方面的考量并且我们知道它可以使用任何 `i32` 调用。在类似这样的例子中，我们可以使用 `safe` 关键字来表明这个特定的函数即便是在 `unsafe extern` 块中也是可以安全调用的。一旦我们做出这个修改，调用它不再需要 `unsafe` 块，如示例 20-9 所示。

文件名：src/main.rs

```
unsafe extern "C" {
    safe fn abs(input: i32) -> i32;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

将一个函数标记为 `safe` 并不会固有地使其变得安全！相反，这像是一个对 Rust 的承诺表明它**是**安全的。确保履行这个承诺仍然是你的责任！

从其它语言调用 Rust 函数

也可以使用 `extern` 来创建一个允许其它语言调用 Rust 函数的接口。不同于创建整个 `extern` 块，就在 `fn` 关键字之前增加 `extern` 关键字并为相关函数指定所用到的 ABI。还需增加 `#[no_mangle]` 注解来告诉 Rust 编译器不要 mangle 此函数的名称。*Mangling* 指编译器将我们命名的函数名更改为包含更多供其他编译过程使用的信息的名称，不过可读性较差。每一个编程语言的编译器都会以稍微不同的方式 mangle 函数名，所以为了使 Rust 函数能在其他语言中指定，必须禁用 Rust 编译器的 name mangling。这是不安全的因为在没有内置 mangling 的时候在库之间可能有命名冲突，所以确保所选的名称可以不用 mangling 地安全导出是我们的责任。

在如下的例子中，一旦其编译为动态库并从 C 语言中链接，`call_from_c` 函数就能够在 C 代码中访问：

```
#[unsafe(no_mangle)]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

> 这种 `extern` 用法只在属性中需要 `unsafe`，而不需要在 `extern` 块本身使用 `unsafe`。

访问或修改可变静态变量

在本书中，我们尚未讨论过 **全局变量**（*global variables*），Rust 确实支持它们，不过这对于 Rust 的所有权规则来说是有问题的。如果有两个线程访问相同的可变全局变量，则可能会造成数据竞争。

全局变量在 Rust 中被称为 **静态**（*static*）变量。示例 20-10 展示了一个拥有字符串 slice 值的静态变量的声明和使用：

文件名：src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {HELLO_WORLD}");
}
```

示例 20-10: 定义和使用一个不可变静态变量

静态（`static`）变量类似于第三章 “变量和常量的区别” 部分讨论的常量。通常静态变量的名称采用 `SCREAMING_SNAKE_CASE` 写法。静态变量只能储存拥有 `'static` 生命周期的引用，这意味着 Rust 编译器可以自己计算出其生命周期而无需显式标注。访问不可变静态变量是安全的。

常量与不可变静态变量的一个微妙的区别是静态变量中的值有一个固定的内存地址。使用这个值总是会访问相同的地址。另一方面，常量则允许在任何被用到的时候复制其数据。另一个区别在于静态变量可以是可变的。访问和修改可变静态变量都是 **不安全** 的。示例 20-11 展示了如何声明、访问和修改名为 `COUNTER` 的可变静态变量：

文件名：src/main.rs

```
static mut COUNTER: u32 = 0;

/// SAFETY: 同时在多个线程调用这个方法是未定义的行为，所以你*必须*保证同一时间只
/// 有一个线程在调用它。
unsafe fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    unsafe {
        // SAFETY: 它只在 `main` 这一个线程被调用。
        add_to_count(3);
        println!("COUNTER: {}", *(&raw const COUNTER));
    }
}
```

示例 20-11: 读取或修改一个可变静态变量是不安全的

就像常规变量一样，我们使用 `mut` 关键字来指定可变性。任何读写 `COUNTER` 的代码都必须位于 `unsafe` 块中。这段代码可以编译并如期打印出 `COUNTER: 3`，因为这是单线程的。拥有多个线程访问 `COUNTER` 则可能导致数据竞争，所以这是未定义行为。因此，我们需要将整个函数标记为 `unsafe`，并在文档注释中说明其安全性限制，以便调用者明确哪些操作是安全的、哪些是不安全的。

每当我们编写一个不安全函数，惯常做法是编写一个以 `SAFETY` 开头的注释并解释调用者需要做什么才可以安全地调用该方法。同理，当我们进行不安全操作时，惯常做法是编写一个以 `SAFETY` 开头并解释安全性规则是如何维护的。

另外，编译器不会允许你创建一个可变静态变量的引用。你只能通过用裸指针解引用操作符创建的裸指针访问它。这包括引用的创建是不可见的情况，例如这个代码示例中用于 `println!` 的情况。可变静态变量只能通过裸指针创建的要求有助于确保使用它们的安全要求更为明确。

拥有可以全局访问的可变数据，难以保证不存在数据竞争，这就是为何 Rust 认为可变静态变量是不安全的。在任何可能的情况下，请优先使用第十六章讨论的并发技术和线程安全智能指针，这样编译器就能检测不同线程间的数据访问是否是安全的。

实现不安全 trait

我们可以使用 `unsafe` 来实现一个不安全 trait。当 trait 中至少有一个方法中包含编译器无法验证的不变式（invariant）时该 trait 就是不安全的。可以在 `trait` 之前增加 `unsafe` 关键字将 trait 声明为 `unsafe`，同时 trait 的实现也必须标记为 `unsafe`，如示例 20-12 所示：

```
unsafe trait Foo {
    // 方法在这里
}

unsafe impl Foo for i32 {
    // 方法实现在这里
}
```

示例 20-12: 定义并实现不安全 trait

通过 `unsafe impl`，我们承诺将保证编译器所不能验证的不变式。

作为一个例子，回忆第十六章 “使用 `Sync` 和 `Send` trait 的可扩展并发” 部分中的 `Sync` 和 `Send` 标记 trait：如果我们的类型完全由实现了 `Send` 与 `Sync` 的其他类型组成，编译器会自动为其实现这些 trait。如果我们定义的类型包含某些未实现 `Send` 或 `Sync` 的类型，例如裸指针，但又想将该类型标记为 `Send` 或 `Sync`，就必须使用 `unsafe`。Rust 不能验证我们的类型保证可以安全的跨线程发送或在多线程间访问，所以需要我们自己进行检查并通过 `unsafe` 表明。

访问联合体中的字段

最后一个只能在 `unsafe` 块中执行的操作是访问（union）中的字段。`union` 和 `struct` 类似，但是在一个实例中同时只能使用一个已声明的字段。联合体主要用于和 C 代码中的联合体进行交互。访问联合体的字段是不安全的，因为 Rust 无法保证当前存储在联合体实例中数据的类型。可以查看 the Rust Reference 了解有关联合体的更多信息。

使用 miri 检查不安全代码

当编写不安全代码时，你可能会想要检查编写的代码是否真的安全正确。最好的方式之一是使用 Miri，一个用来检测未定义行为的 Rust 官方工具。鉴于借用检查器是一个在编译时工作的**静态**工具，Miri 是一个在运行时工作的**动态**工具。它通过运行程序，或者测试集来检查代码，并检测你是否违反了它理解的 Rust 应该如何工作的规则。

使用 Miri 要求使用 nightly 版本的 Rust（我们在附录 G：Rust 是如何开发的与 “Nightly Rust”中有更多讨论）。你可以通过输入 `rustup +nightly component add miri` 来同时安装 nightly 版本的 Rust 和 Miri。这并不会改变你项目正在使用的 Rust 版本；它只是为你的系统增加了这个工具所以你可以在需要的时候使用它。你可以通过输入 `cargo +nightly miri run` or `cargo +nightly miri test` 在项目中使用 Miri。

作为一个它是如何有用的例子，考虑一下对示例 20-11 运行它时会发生什么。

```
$ cargo +nightly miri run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
     Running `/Users/chris/.rustup/toolchains/nightly-aarch64-apple-darwin/bin/
cargo-miri runner target/miri/aarch64-apple-darwin/debug/unsafe-example`
warning: creating a shared reference to mutable static is discouraged
  --> src/main.rs:14:33
   |
14 |         println!("COUNTER: {}", COUNTER);
   |                                 ^^^^^^^ shared reference to mutable static
   |
   = note: for more information, see <https://doc.rust-lang.org/nightly/edition-
guide/rust-2024/static-mut-references.html>
   = note: shared references to mutable statics are dangerous; it's undefined
behavior if the static is mutated or if a mutable reference is created for it
while the shared reference lives
   = note: `#[warn(static_mut_refs)]` on by default

COUNTER: 3
```

Miri 正确地警告了我们共享了可变数据的引用。这里，Miri 只是发出了一个警告因为在这个例子中并不能保证是未定义行为，它也没有告诉我们如何修复问题。但是至少我们知道这里有未定义行为的风险并接着可以思考如何使代码变得安全。在一些例子中，Miri 也可以检测真正的错误 – **确定**是错误的代码模式 – 并提出如何修复这些错误的推荐方案。

Miri 并不能捕获编写不安全代码时可能出现的所有错误。Miri 是一个动态分析工具，因此它只能捕获代码实际运行时出现的问题。这意味着需要将其与良好的测试技术相结合以增强你对所编写的不安全代码的信心。Miri 也不能覆盖代码所有的不可靠的地方。

换句话说：如果 Miri **可以**捕获一个问题，你知道这里有个 bug，不过仅仅是因为 miri **没有**捕获一个 bug 并不意味着这里没有问题。但是它可以捕获很多问题。尝试对本章中的其它不安全代码示例运行它来看看它会说些什么！

你可以在 Miri 的 GitHub 仓库了解更多信息。

何时使用不安全代码

使用 `unsafe` 来进行这五个操作（超能力）之一是没有问题的，甚至是不需要深思熟虑的，不过使得 `unsafe` 代码正确也实属不易，因为编译器不能帮助保证内存安全。当有理由使用

`unsafe` 代码时，是可以这么做的，通过使用显式的 `unsafe` 标注可以更容易地在错误发生时追踪问题的源头。每当编写不安全代码时，都可以借助 Miri 来更加自信地验证所写代码是否遵循 Rust 的规则。

若想更深入地了解如何高效使用不安全 Rust，请阅读 Rust 关于该主题的官方指南 Rustonomicon。

高级 trait

在第十章"trait：定义共同行为"部分，我们第一次涉及到了 trait，不过我们并没有覆盖一些较为高级的细节。现在你对 Rust 已经有了更多了解，我们可以深入探究了。

关联类型

关联类型（*associated types*）将一个类型占位符与 trait 相关联，使得该 trait 的方法定义可以在签名中使用这些占位符类型。该 trait 的实现者会为每个具体实现指定要使用的具体类型来替代占位符类型。这样，我们就能在定义 trait 时使用占位符类型，而无需预先知道这些类型的具体内容，直到实现该 trait 时再进行指定。

我们之前提到，本章所讨论的大多数高级特性都很少需要。关联类型则比较适中：它们的使用频率低于本书其他部分讲解的特性，但又高于本章中许多其他特性。

一个带有关联类型的 trait 的例子是标准库提供的 `Iterator` trait。它有一个叫做 `Item` 的关联类型来替代遍历的值的类型。`Iterator` trait 的定义如示例 20-13 所示：

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

示例 20-13: `Iterator` trait 的定义中带有关联类型 `Item`

`Item` 是一个占位符类型，同时 `next` 方法的定义表明它返回 `Option<Self::Item>` 类型的值。`Iterator` trait 的实现者会指定 `Item` 的具体类型，于是 `next` 方法就会返回一个包含该具体类型值的 `Option`。

关联类型可能看起来与泛型类似，后者允许我们在定义函数时不必指定它可以处理的类型。为了体现这两者的区别，我们来看一个名为 `Counter` 的类型上的 `Iterator` trait 实现，其中指定 `Item` 的类型为 `u32`：

文件名：src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

这种语法看起来与泛型类似。那么为什么不直接像示例 20-14 那样，用泛型来定义 `Iterator` trait 呢？

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

示例 20-14: 一个使用泛型的 `Iterator` trait 假想定义

区别在于当如示例 20-14 那样使用泛型时，则不得不在每一个实现中标注类型；这是因为我们也可以实现为 `Iterator<String> for Counter`，或任何其他类型，这样就可以有多个针对 `Counter` 的 `Iterator` 的实现。换句话说，当 trait 有泛型参数时，可以多次实现这个 trait，每次都使用不同的具体泛型参数类型。当我们在 `Counter` 上调用 `next` 方法时，就必须通过类型注解来指明要使用哪一个 `Iterator` 的实现。

使用关联类型后，则无需标注类型，因为不能对同一个类型多次实现该 trait。在示例 20-13 中使用关联类型的定义里，我们只能为 `Item` 选择一次具体类型，因为只能有一个 `impl Iterator for Counter`。当调用 `Counter` 的 `next` 时不必每次指定我们需要 `u32` 值的迭代器。

关联类型也会成为 trait 契约的一部分：trait 的实现必须提供一个类型来替代关联类型占位符。关联类型通常以它的用途来命名，在 API 文档中对关联类型进行说明也是一种良好实践。

默认泛型类型参数和运算符重载

当使用泛型类型参数时，可以为泛型指定一个默认的具体类型。如果默认类型就足够的话，这消除了为具体类型实现 trait 的需要。为泛型类型指定默认类型的语法是在声明泛型类型时使用 `<PlaceholderType=ConcreteType>`。

这种技术的一个很好的示例是 **运算符重载** (*operator overloading*)，即在特定情况下自定义运算符（比如 `+`）行为的操作。

Rust 并不允许创建自定义运算符或重载任意运算符，但可以通过实现 `std::ops` 中列出的运算符相关 trait 来重载它们。例如，在示例 20-15 中我们重载 `+` 运算符来将两个 `Point` 实例相加。我们通过在 `Point` 结构体上实现 `Add` trait 来实现这一点。

文件名：src/main.rs

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(
        Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
        Point { x: 3, y: 3 }
    );
}
```

示例 20-15: 实现 `Add` trait 重载 `Point` 实例的 `+` 运算符

`add` 方法将两个 `Point` 实例的 `x` 值和 `y` 值分别相加来创建一个新的 `Point`。`Add` trait 有一个叫做 `Output` 的关联类型，它用来决定 `add` 方法的返回值类型。

这里默认泛型类型位于 `Add` trait 中。这里是其定义：

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

这些代码看来应该很熟悉：一个带有一个方法和一个关联类型的 trait。新增的部分是 `Rhs=Self`：这个语法叫做 **默认类型参数**（*default type parameters*）。`Rhs` 是一个泛型类型参数（“right-hand side” 的缩写），它用于定义 `add` 方法中的 `rhs` 参数。如果实现 `Add` trait 时不指定 `Rhs` 的具体类型，`Rhs` 的类型将默认为 `Self`，即正在实现 `Add` 的类型。

当为 `Point` 实现 `Add` 时，使用了默认的 `Rhs`，因为我们希望将两个 `Point` 实例相加。让我们看看一个实现 `Add` trait 时希望自定义 `Rhs` 类型而不是使用默认类型的例子。

这里有两个存放不同单元值的结构体，`Millimeters` 和 `Meters`。这种将现有类型简单封装进另一个结构体的方式被称为 **newtype 模式**（*newtype pattern*），之后的“使用 newtype 模式在外部类型上实现外部 trait”部分会做详细介绍。我们希望能够将毫米值与米值相加，并让 `Add` 的实现正确处理单位转换。可以为 `Millimeters` 实现 `Add` 并以 `Meters` 作为 `Rhs`，如示例 20-16 所示。

文件名：src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

示例 20-16: 在 `Millimeters` 上实现 `Add`，以便能够将 `Millimeters` 与 `Meters` 相加

为了使 `Millimeters` 和 `Meters` 能够相加，我们指定 `impl Add<Meters>` 来设定 `Rhs` 类型参数的值而不是使用默认的 `Self`。

默认参数类型主要用于如下两个方面：

- 扩展类型而不破坏现有代码。
- 在大部分用户都不需要的特定情况进行自定义。

标准库的 `Add` trait 就是第二个目的的一个例子：大部分时候你会将两个相似的类型相加，但 `Add` trait 也提供了自定义额外行为的能力。在 `Add` trait 定义中使用默认类型参数意味着大部

分时候无需指定额外的参数。换句话说，一小部分实现的样板代码是不必要的，这样使用 trait 就更容易了。

第一个目的与第二个相似但方向相反：如果需要为现有 trait 增加类型参数，为其提供一个默认类型将允许我们在不破坏现有实现代码的基础上扩展 trait 的功能。

在同名方法之间消歧义

Rust 既不能避免一个 trait 与另一个 trait 拥有相同名称的方法，也不能阻止为同一类型同时实现这两个 trait。同时还可以直接在类型上实现一个与 trait 方法同名的方法。

当调用这些同名方法时，需要告诉 Rust 我们想要使用哪一个。考虑一下示例 20-17 中的代码，这里我们定义了两个 trait，`Pilot` 和 `Wizard`，它们都拥有名为 `fly` 的方法。接着在一个本身已经实现了名为 `fly` 方法的类型 `Human` 上实现这两个 trait。每一个 `fly` 方法都进行了不同的操作：

文件名：src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

示例 20-17: 两个 trait 定义为拥有 `fly` 方法，并在直接定义有 `fly` 方法的 `Human` 类型上实现这两个 trait

当调用 `Human` 实例的 `fly` 时，编译器默认调用直接实现在该类型上的方法，如示例 20-18 所示。

文件名：src/main.rs

```
```

```
fn main() {
    let person = Human;
    person.fly();
}
```

示例 20-18: 调用一个 `Human` 实例的 `fly`

运行这段代码会打印出 `*waving arms furiously*`，这表明 Rust 调用了直接实现在 `Human` 上的 `fly` 方法。

为了能够调用 `Pilot` trait 或 `Wizard` trait 的 `fly` 方法，需要使用更明确的语法来指定具体要调用的 `fly` 方法。示例 20-19 演示了这种语法。

文件名：src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

示例 20-19: 指定我们希望调用哪一个 trait 的 `fly` 方法

在方法名前指定 trait 名称可让 Rust 明确我们想调用哪个 `fly` 实现。也可以选择写成 `Human::fly(&person)`，这等同于示例 20-19 中的 `person.fly()`，不过如果无需消歧义的话这么写就有点冗长了。

运行这段代码会打印出如下内容：

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

因为 `fly` 方法获取一个 `self` 参数，如果有两个**类型**都实现了同一 **trait**，Rust 可以根据 `self` 的类型计算出应该使用哪一个 trait 实现。

然而，关联函数中非方法的函数不带有 `self` 参数。当存在多个类型或者 trait 定义了相同函数名的非方法函数时，Rust 就不总是能计算出我们期望的是哪一个类型，除非使用 **完全限定语法**（*fully qualified syntax*）。例如示例 20-20 中的创建了一个希望将所有小狗叫做 *Spot* 的动物收容所的 trait。`Animal` trait 有一个关联非方法函数 `baby_name`。结构体 `Dog` 实现了 `Animal`，同时又直接提供了关联非方法函数 `baby_name`。

文件名：src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}
```

```
struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

示例 20-20: 一个带有关联函数的 trait 和一个带有同名关联函数并实现了此 trait 的类型

在 `Dog` 类型上定义的关联函数 `baby_name` 中，我们实现了将所有小狗命名为 Spot 的功能。`Dog` 类型还实现了 `Animal` trait，它描述了所有动物所共有的特征。小狗被称为 puppy，这表现为 `Dog` 的 `Animal` trait 实现中与 `Animal` trait 相关联的函数 `baby_name`。

在 `main` 调用了 `Dog::baby_name` 函数，它直接调用了定义于 `Dog` 之上的关联函数。这段代码会打印出：

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.54s
     Running `target/debug/traits-example`
A baby dog is called a Spot
```

这不是我们想要的输出。我们希望调用的是 `Dog` 上 `Animal` trait 实现那部分的 `baby_name` 函数，这样能够打印出 `A baby dog is called a puppy`。我们在示例 20-19 中使用的指定 trait 名称的技巧在这里不起作用；如果将 `main` 改为示例 20-21 中的代码，就会得到编译错误：

文件名：src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```



示例 20-21: 尝试调用 `Animal` trait 的 `baby_name` 函数，不过 Rust 并不知道该使用哪一个实现

因为 `Animal::baby_name` 没有 `self` 参数，而且可能有其他类型实现了 `Animal` trait，Rust 无法确定我们想调用哪一个 `Animal::baby_name` 的实现。此时会得到如下编译错误：

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0790]: cannot call associated function on trait without specifying the
corresponding `impl` type
```

```
  --> src/main.rs:20:43
   |
2  |     fn baby_name() -> String;
   |     ------------------------ `Animal::baby_name` defined here
...
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                           ^^^^^^^^^^^^^^^^^^ cannot call
associated function of trait
   |
help: use the fully-qualified path to the only available implementation
   |
20 |     println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
   |                                           +++++++       +

For more information about this error, try `rustc --explain E0790`.
error: could not compile `traits-example` (bin "traits-example") due to 1 previous
error
```

为了消歧义并告诉 Rust 我们希望使用的是 `Dog` 的 `Animal` 实现而不是其它类型的 `Animal` 实现，需要使用**完全限定语法**。示例 20-22 演示了如何使用完全限定语法：

文件名：src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

示例 20-22: 使用完全限定语法来指定我们希望调用的是 `Dog` 上 `Animal` trait 实现中的 `baby_name` 函数

我们在尖括号中向 Rust 提供了类型注解，这表明我们希望在此次函数调用中将 `Dog` 类型视为 `Animal`，从而调用在 `Dog` 上实现的 `Animal` trait 中的 `baby_name` 方法。现在这段代码将打印出我们期望的结果：

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/traits-example`
A baby dog is called a puppy
```

通常，完全限定语法定义为如下：

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

对于不是方法的关联函数，并没有一个 `receiver`：故只会有其他参数的列表。可以选择在任何函数或方法调用处使用完全限定语法。然而，允许省略任何 Rust 能够从程序中的其他信息中计算出的部分。只有当存在多个同名实现而 Rust 需要帮助以便知道我们希望调用哪个实现时，才需要使用这个较为冗长的语法。

使用超 trait

有时我们可能会需要编写一个依赖另一个 trait 的 trait 定义：对于一个实现了第一个 trait 的类型，你希望要求这个类型也实现了第二个 trait。如此就可使 trait 定义使用第二个 trait 的关联项。这个所需的 trait 是我们实现的 trait 的 **超（父）trait**（*supertrait*）。

例如我们希望创建一个带有 `outline_print` 方法的 trait `OutlinePrint`，它会将给定的值格式化为带有星号框。也就是说，给定一个实现了标准库 `Display` trait 的并返回 `(x, y)` 的 `Point`，当我们对一个 `x` 为 `1`、`y` 为 `3` 的 `Point` 实例调用 `outline_print` 时，它应该打印出如下内容：

```
**********
*        *
* (1, 3) *
*        *
**********
```

在 `outline_print` 的实现中，我们希望使用 `Display` trait 的功能。因此，需要说明 `OutlinePrint` trait 仅适用于那些同时实现了 `Display` 并提供 `OutlinePrint` 所需功能的类型。可以通过在 trait 定义中指定 `OutlinePrint: Display` 来做到这一点。这种技术类似于为 trait 增加 trait bound。示例 20-23 展示了一个 `OutlinePrint` trait 的实现：

文件名：src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {output} *");
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

示例 20-23: 实现 `OutlinePrint` trait，它要求来自 `Display` 的功能

因为我们已经指定 `OutlinePrint` 需要 `Display` trait，因而可以使用自动为任何实现了 `Display` 的类型提供的 `to_string` 方法。如果我们在没有在 trait 名称后添加冒号并指定 `Display` trait 的情况下尝试使用 `to_string`，就会出现错误，提示在当前作用域中未为类型 `&Self` 找到名为 `to_string` 的方法。

让我们看看如果尝试在一个没有实现 `Display` 的类型上实现 `OutlinePrint` 会发生什么，比如 `Point` 结构体：

文件名：src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}
```

```
impl OutlinePrint for Point {}
```



这样会得到一个错误说 `Display` 是必须的而未被实现：

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:20:23
   |
20 | impl OutlinePrint for Point {}
   |                       ^^^^^ `Point` cannot be formatted with the default
formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
note: required by a bound in `OutlinePrint`
  --> src/main.rs:3:21
   |
3  | trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in `OutlinePrint`

error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:24:7
   |
24 |     p.outline_print();
   |       ^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
note: required by a bound in `OutlinePrint::outline_print`
  --> src/main.rs:3:21
   |
3  | trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in
`OutlinePrint::outline_print`
4  |     fn outline_print(&self) {
   |        ------------- required by a bound in this associated function

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` (bin "traits-example") due to 2 previous
errors
```

为了修复这个问题，我们在 `Point` 上实现 `Display` 并满足 `OutlinePrint` 要求的限制，比如这样：

文件名：src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
```

```
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

那么在 `Point` 上实现 `OutlinePrint` trait 就能成功编译，并可以在 `Point` 实例上调用 `outline_print` 将其显示在由星号组成的边框内。

使用 newtype 模式在外部类型上实现外部 trait

在第十章的 “为类型实现 trait” 部分，我们提到了孤儿规则（orphan rule），它规定只有当 trait 或类型至少有一方或两者都对于当前 crate 是本地时，才能在该类型上实现该 trait。一个绕开这个限制的方法是使用 **newtype 模式**（*newtype pattern*），它涉及到在一个元组结构体（第五章 “用没有命名字段的元组结构体来创建不同的类型” 部分介绍了元组结构体）中创建一个新类型。这个元组结构体带有一个字段作为希望实现 trait 的类型的简单封装。由于这个封装类型对于 crate 是本地的，这样就可以在这个封装上实现 trait。*Newtype* 是一个源自 Haskell 编程语言的概念。使用这个模式没有运行时性能惩罚，这个封装类型在编译时就被省略了。

例如，如果想要在 `Vec<T>` 上实现 `Display`，而孤儿规则阻止我们直接这么做，因为 `Display` trait 和 `Vec<T>` 都定义于我们的 crate 之外。可以创建一个包含 `Vec<T>` 实例的 `Wrapper` 结构体，接着可以如列表 20-24 那样在 `Wrapper` 上实现 `Display` 并使用 `Vec<T>` 的值：

文件名：src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {w}");
}
```

示例 20-24: 创建 `Wrapper` 类型封装 `Vec<String>` 以便能够实现 `Display`

`Display` 的实现使用 `self.0` 来访问其内部的 `Vec<T>`，因为 `Wrapper` 是元组结构体而 `Vec<T>` 是结构体总位于索引 0 的项。接着就可以使用 `Wrapper` 中 `Display` 的功能了。

这种做法的缺点在于因为 `Wrapper` 是一个新类型，它并不具备其所封装值的方法。必须直接在 `Wrapper` 上实现 `Vec<T>` 的所有方法，这样就可以代理到 `self.0` 上，这就允许我们完全像 `Vec<T>` 那样对待 `Wrapper`。如果希望新类型拥有其内部类型的每一个方法，为封装类型实现 `Deref` trait（第十五章 “使用 `Deref` Trait 将智能指针当作常规引用处理” 部分讨论过）并返回其内部类型是一种解决方案。如果不希望封装类型拥有所有内部类型的方法 —— 比如为了限制封装类型的行为 —— 则只需自行实现所需的方法即可。

甚至当不涉及 trait 时 newtype 模式也很有用。现在让我们将关注点转移到一些与 Rust 类型系统交互的高级方式上来吧。

高级类型

Rust 的类型系统有一些我们曾经提到但尚未讨论过的特性。首先我们将从一般意义上讨论 newtype 并探讨它们作为类型为何有用。接着会转向类型别名（type aliases），一个类似于 newtype 但有着稍微不同的语义的功能。我们还会讨论 `!` 类型和动态大小类型。

使用 newtype 模式实现类型安全和抽象

本小节假设你已经阅读了之前的 “使用 newtype 模式在外部类型上实现外部 trait” 部分。

newtype 模式还可用于我们到目前为止尚未讨论的其他任务，包括静态地确保值不会混淆以及标注值的单位。你在示例 20-16 中已经看到了一个使用 newtype 来表示单位的例子：`Millimeters` 和 `Meters` 结构体都在 newtype 中封装了 `u32` 值。如果编写了一个有 `Millimeters` 类型参数的函数，不小心使用 `Meters` 或普通的 `u32` 值来调用该函数的程序是不能编译的。

newtype 模式也可以用于抽象掉某个类型的部分实现细节：新的类型可以暴露与其私有内部类型不同的共有 API。

newtype 模式还可以隐藏内部实现。例如，可以提供一个封装了 `HashMap<i32, String>` 的 `People` 类型，用来储存人名以及相应的 ID。使用 `People` 的代码只需与我们提供的公有 API 交互即可，比如向 `People` 集合增加名字字符串的方法；这样这些代码就无需知道在内部我们将一个 `i32` ID 赋予了这个名字了。newtype 模式是一种实现第十八章 “封装隐藏了实现细节” 中讨论的隐藏实现细节的轻量级封装方法。

使用类型别名创建类型同义词

Rust 提供了声明 **类型别名**（*type alias*）的能力，使用 `type` 关键字为现有类型赋予另一个名字。例如，可以像这样创建 `i32` 的别名 `Kilometers`：

```
type Kilometers = i32;
```

这意味着 `Kilometers` 是 `i32` 的 **同义词**（*synonym*）；不同于示例 20-16 中创建的 `Millimeters` 和 `Meters` 类型。`Kilometers` 并不是一个新的、单独的类型。`Kilometers` 类型的值将被完全当作 `i32` 类型值来对待：

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

因为 `Kilometers` 是 `i32` 的别名，它们是同一类型，可以将 `i32` 与 `Kilometers` 相加，也可以将 `Kilometers` 传递给获取 `i32` 参数的函数。但通过这种手段无法获得上一部分讨论的 newtype 模式所提供的类型检查的好处。换句话说，如果在某处混用 `Kilometers` 和 `i32` 的值，编译器也不会给出一个错误。

类型别名的主要用途是减少重复。例如，可能会有这样很长的类型：

```
```

```
Box<dyn Fn() + Send + 'static>
```

在函数签名和类型注解中到处书写这个冗长的类型既乏味又容易出错。想象一下有一个项目，到处都是像 Listing 20-25 那样的代码。

```
    let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

    fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
        // --snip--
    }

    fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
        // --snip--
    }
```

示例 20-25: 在很多地方使用名称很长的类型

类型别名通过减少重复使代码更易于管理。在示例 20-26 中，我们为这个冗长的类型引入了名为 `Thunk` 的别名，并可以使用更简洁的 `Thunk` 来替换所有使用该类型的地方。

```
    type Thunk = Box<dyn Fn() + Send + 'static>;

    let f: Thunk = Box::new(|| println!("hi"));

    fn takes_long_type(f: Thunk) {
        // --snip--
    }

    fn returns_long_type() -> Thunk {
        // --snip--
    }
```

示例 20-26: 引入类型别名 `Thunk` 来减少重复

这样阅读和编写代码都容易多了！为类型别名选择一个好名字也可以帮助你表达意图（单词 *thunk* 表示会在之后被计算的代码，所以这是一个存放闭包的合适的名字）。

类型别名也经常与 `Result<T, E>` 结合使用来减少重复。考虑一下标准库中的 `std::io` 模块。I/O 操作通常会返回一个 `Result<T, E>` 来处理操作失败的情况。标准库中的 `std::io::Error` 结构体代表了所有可能的 I/O 错误。`std::io` 中的许多函数都会返回 `Result<T, E>`，其中 `E` 是 `std::io::Error`，比如 `Write` trait 中的这些函数：

```
use std::fmt;
use std::io::Error;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

这里重复出现了很多次 `Result<..., Error>`。为此，`std::io` 有这个类型别名声明：

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

该声明位于 `std::io` 模块中，因此我们可以使用完全限定的别名 `std::io::Result<T>`；也就是说，`Result<T, E>` 中 `E` 放入了 `std::io::Error`。`Write` trait 中的函数最终看起来像这样：

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

类型别名在两个方面有帮助：易于编写**并**在整个 `std::io` 中提供了一致的接口。因为这是一个别名，它只是另一个 `Result<T, E>`，这意味着可以在其上使用 `Result<T, E>` 的任何方法，以及像 `?` 这样的特殊语法。

从不返回的 never type

Rust 有一个叫做 `!` 的特殊类型。在类型理论术语中被称为 *empty type*，因为它没有值。我们更倾向于称之为 *never type*。这个名字描述了它的作用：在函数从不返回的时候充当返回值。下面是一个示例：

```
fn bar() -> ! {
    // --snip--
}
```

这段代码可以读作 “函数 `bar` 从不返回”，而从不返回的函数被称为 **发散函数**（*diverging functions*）。不能创建 `!` 类型的值，所以 `bar` 也不可能返回值。

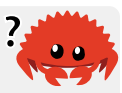
不过一个不能创建值的类型有什么用呢？回想一下示例 2-5 中猜数字游戏的代码；我们在示例 20-27 中重现了其中的一小部分：

```
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

示例 20-27: `match` 语句和一个以 `continue` 结束的分支

当时我们忽略了代码中的一些细节。在第六章 “`match` 控制流运算符” 部分，我们学习了 `match` 的分支必须返回相同的类型。如下代码不能工作：

```
    let guess = match guess.trim().parse() {
        Ok(_) => 5,
        Err(_) => "hello",
    };
```



这里的 `guess` 必须既是整型 **也是** 字符串，而 Rust 要求 `guess` 只能是一个类型。那么 `continue` 返回了什么呢？为什么在示例 20-27 中，一个分支返回 `u32`，而另一个分支却以 `continue` 结束呢？

正如你可能猜到的，`continue` 的值是 `!`。也就是说，当 Rust 要计算 `guess` 的类型时，它会查看这两个分支。前者是 `u32` 值，而后者是 `!` 值。因为 `!` 类型永远不会有值，Rust 决定 `guess` 的类型是 `u32`。

描述这种行为的正式方式是，类型为 `!` 的表达式可以被强制转换为任意其他类型。之所以允许 `match` 分支以 `continue` 结束是因为 `continue` 并不真正返回值；相反它把控制权交回上层循环，所以在 `Err` 的情况，事实上并未对 `guess` 进行赋值。

never type 在 `panic!` 宏中也很有用。还记得 `Option<T>` 上的 `unwrap` 函数吗？它产生一个值或 panic。这里是它的定义：

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

这里与示例 20-27 中的 `match` 发生了相同的情况：Rust 知道 `val` 是 `T` 类型，`panic!` 是 `!` 类型，所以整个 `match` 表达式的结果是 `T` 类型。这能工作是因为 `panic!` 并不产生一个值；它会终止程序。对于 `None` 的情况，`unwrap` 并不返回一个值，所以这些代码是有效的。

最后一个有着 `!` 类型的表达式是 `loop`：

```
    print!("forever ");

    loop {
        print!("and ever ");
    }
```

这里，循环永远也不结束，所以此表达式的值是 `!`。但是如果引入 `break` 这就不为真了，因为循环在执行到 `break` 后就会终止。

动态大小类型和 `Sized` trait

Rust 需要知道有关类型的某些细节，例如为特定类型的值需要分配多少空间。这便是起初留下的一个类型系统中令人迷惑的角落：即 **动态大小类型**（*dynamically sized types*）的概念。这有时被称为 “DST” 或 “unsized types”，它们让我们能够编写使用那些只有在运行时才能知道大小的值的代码。

让我们深入研究我们在整本书中一直在使用的动态大小类型 `str` 的细节。没错，不是 `&str`，而是单独的 `str` 就是一个 DST。直到运行时我们都不知道字符串有多长。我们无法在编译时知道字符串的长度，这意味着我们无法创建 `str` 类型的变量，也不能获取 `str` 类型的参数。考虑一下这些代码，它们不能工作：

```
```

```
    let s1: str = "Hello there!";
    let s2: str = "How's it going?";
```



Rust 需要知道应该为特定类型的值分配多少内存，同时所有同一类型的值必须使用相同数量的内存。如果允许编写这样的代码，也就意味着这两个 `str` 需要占用完全相同大小的空间。不过它们有着不同的长度：`s1` 需要 12 字节存储，而 `s2` 需要 15 字节。这也就是为什么不可能创建一个存放动态大小类型的变量的原因。

那么该怎么办呢？在这种情况下，你已经知道答案：`s1` 和 `s2` 的类型是 `&str` 而不是 `str`。如果你回想第四章 “字符串 slice” 中提到，slice 数据结构仅仅储存了开始位置和 slice 的长度。所以虽然 `&T` 是一个储存了 `T` 所在的内存位置的单个值，`&str` 则是**两个**值：`str` 的地址和其长度。这样，`&str` 就有了一个在编译时可以知道的大小：它是 `usize` 长度的两倍。也就是说，无论所引用的字符串多长，我们总是知道 `&str` 的大小。一般来说，这就是 Rust 使用动态大小类型的方式：它们有一些额外的元信息来储存动态信息的大小。这引出了动态大小类型的黄金法则：必须将动态大小类型的值置于某种指针之后。

可以将 `str` 与所有类型的指针结合：比如 `Box<str>` 或 `Rc<str>`。事实上，之前我们已经见过了，不过是另一个动态大小类型：trait。每一个 trait 都是一个可以通过 trait 名称来引用的动态大小类型。在第十八章 “顾及不同类型值的 trait 对象” 中，我们提到了为了将 trait 用于 trait 对象，必须将它们放入指针之后，比如 `&dyn Trait` 或 `Box<dyn Trait>`（`Rc<dyn Trait>` 也可以）。

为了处理 DST，Rust 提供了 `Sized` trait 来决定一个类型的大小是否在编译时可知。该 trait 会自动为所有在编译时大小已知的类型实现。此外，Rust 隐式地为每一个泛型函数增加了 `Sized` bound。也就是说，对于如下泛型函数定义：

```
fn generic<T>(t: T) {
    // --snip--
}
```

实际上，这会被当作我们写了如下内容来处理：

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

默认情况下，泛型函数只能作用于在编译时大小已知的类型。然而，你可以使用如下特殊语法来放宽这一限制：

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

`?Sized` 这个 trait bound 表示 “`T` 可以是 `Sized`，也可以不是 `Sized`” 同时这个注解会覆盖泛型类型必须在编译时拥有固定大小的默认规则。具有该含义的 `?Trait` 语法仅适用于 `Sized`，而不适用于其他任何 trait。

另外注意我们将 `t` 参数的类型从 `T` 变为了 `&T`：因为其类型可能不是 `Sized` 的，所以需要将其置于某种指针之后。在这个例子中选择了引用。

接下来，我们将讨论函数和闭包！

高级函数与闭包

本部分将探索一些有关函数和闭包的高级特性，这包括函数指针以及返回闭包。

函数指针

我们讨论过了如何向函数传递闭包；也可以将普通函数传递给函数！这个技术在我们希望传递已经定义的函数而不是重新定义闭包作为参数时很有用。函数会被强制转换为 `fn` 类型（小写的 f），不要与闭包 trait 的 `Fn` 相混淆。`fn` 被称为 **函数指针**（*function pointer*）。通过函数指针允许我们使用函数作为其它函数的参数。

指定参数为函数指针的语法类似于闭包，如示例 20-28 所示，这里定义了一个 `add_one` 函数用于将其参数加一。`do_twice` 函数获取两个参数：一个指向任何获取一个 `i32` 参数并返回一个 `i32` 的函数指针，和一个 `i32` 值。`do_twice` 函数传入 `arg` 参数调用 `f` 函数两次，接着将两次函数调用的结果相加。`main` 函数使用 `add_one` 和 `5` 作为参数调用 `do_twice`。

文件名：src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {answer}");
}
```

示例 20-28: 使用 `fn` 类型接受函数指针作为参数

这段代码会打印出 `The answer is: 12`。`do_twice` 中的 `f` 被指定为一个接受一个 `i32` 参数并返回 `i32` 的 `fn`。接着就可以在 `do_twice` 函数体中调用 `f`。在 `main` 中，可以将函数名 `add_one` 作为第一个参数传递给 `do_twice`。

不同于闭包，`fn` 是一个类型而不是一个 trait，所以直接指定 `fn` 作为参数而不是声明一个带有 `Fn` 作为 trait bound 的泛型参数。

函数指针实现了所有三个闭包 trait（`Fn`、`FnMut` 和 `FnOnce`），所以总是可以在调用期望闭包的函数时传递函数指针作为参数。倾向于编写使用泛型和闭包 trait 的函数，这样它就能接受函数或闭包作为参数。

尽管如此，一个只期望接受 `fn` 而不接受闭包的情况的例子是与不存在闭包的外部代码交互时：C 语言的函数可以接受函数作为参数，但 C 语言没有闭包。

作为一个既可以使用内联定义的闭包又可以使用命名函数的例子，让我们看看一个标准库中 `Iterator` trait 提供的 `map` 方法的应用。使用 `map` 函数将一个数字 vector 转换为一个字符串 vector，就可以使用闭包，如示例 20-29 所示：

```
    let list_of_numbers = vec![1, 2, 3];
    let list_of_strings: Vec<String> =
        list_of_numbers.iter().map(|i| i.to_string()).collect();
```

或者可以将函数作为 `map` 的参数来代替闭包，如示例 20-30 所示：

```
    let list_of_numbers = vec![1, 2, 3];
    let list_of_strings: Vec<String> =
        list_of_numbers.iter().map(ToString::to_string).collect();
```

注意这里必须使用 “高级 trait” 部分讲到的完全限定语法，因为存在多个叫做 `to_string` 的函数。

这里使用了定义于 `ToString` trait 的 `to_string` 函数，标准库为所有实现了 `Display` 的类型实现了这个 trait。

回忆一下第六章 “枚举值” 部分中定义的每一个枚举成员也变成了一个构造函数。我们可以使用这些构造函数作为实现了闭包 trait 的函数指针，这意味着可以指定构造函数作为接受闭包的方法的参数，如示例 20-31 所示：

```
    enum Status {
        Value(u32),
        Stop,
    }

    let list_of_statuses: Vec<Status> = (0u32..20).map(Status::Value).collect();
```

这里，我们通过 `Status::Value` 的初始化函数，对 `map` 所作用的范围内每个 `u32` 值创建 `Status::Value` 实例。一些人倾向于函数式风格，一些人喜欢闭包。它们会编译成相同的代码，因此请选择对你来说更清晰的那一种。

返回闭包

闭包表现为 trait，这意味着不能直接返回闭包。对于大部分需要返回 trait 的场景中，可以使用实现了期望返回的 trait 的具体类型来替代函数的返回值。但是这不能用于闭包，因为它们没有一个可返回的具体类型；例如，当闭包从其作用域捕获任何值时，就不允许使用函数指针 `fn` 作为返回类型。

相反，可以正常地使用第十章所学的 `impl Trait` 语法。可以使用 `Fn`、`FnOnce` 和 `FnMut` 返回任何函数类型。例如，示例 20-32 中的代码就可以正常工作。

```
fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

然而，如我们在 “闭包类型推断和注解” 中所注意到的，每一个闭包也有其独立的类型。如果你需要处理多个拥有相同签名但是不同实现的函数，就需要使用 trait 对象。考虑一下如果编写类似示例 20-33 中所示代码会发生什么。

```

```

```
fn main() {
    let handlers = vec![returns_closure(), returns_initialized_closure(123)];
    for handler in handlers {
        let output = handler(5);
        println!("{output}");
    }
}

fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}

fn returns_initialized_closure(init: i32) -> impl Fn(i32) -> i32 {
    move |x| x + init
}
```



这里有两个函数，`returns_closure` 和 `returns_initialized_closure`，它们都返回 `impl Fn(i32) -> i32`。注意它们返回的闭包是不同的，即使它们实现了相同的类型。如果尝试编译这段代码，Rust 会告诉我们这不可行：

```
$ cargo build
   Compiling functions-example v0.1.0 (file:///projects/functions-example)
error[E0308]: mismatched types
  --> src/main.rs:2:44
   |
2  |     let handlers = vec![returns_closure(), returns_initialized_closure(123)];
   |                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
expected opaque type, found a different opaque type
...
9  | fn returns_closure() -> impl Fn(i32) -> i32 {
   |                         ------------------- the expected opaque type
...
13 | fn returns_initialized_closure(init: i32) -> impl Fn(i32) -> i32 {
   |                                              ------------------- the found
opaque type
   |
   = note: expected opaque type `impl Fn(i32) -> i32` (opaque type at <src/
main.rs:9:25>)
              found opaque type `impl Fn(i32) -> i32` (opaque type at <src/
main.rs:13:46>)
   = note: distinct uses of `impl Trait` result in different opaque types

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions-example` (bin "functions-example") due to 1
previous error
```

错误信息告诉我们每当返回一个 `impl Trait` Rust 会创建一个独特的**不透明类型**（*opaque type*），这是一个无法看清 Rust 为我们构建了什么细节的类型。所以即使这些函数都返回了实现了相同 trait（`Fn(i32) -> i32`）的闭包，Rust 为我们生成的不透明类型也是不同的。这类似于 Rust 如何为不同的异步代码块生成不同的具体类型，即使它们有着相同的输出类型，如第十七章 “使用任意数量的 futures” 所示。我们已经多次看到这个问题的解决方案：我们可以使用 trait 对象，如示例 20-34 所示。

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}

fn returns_initialized_closure(init: i32) -> Box<dyn Fn(i32) -> i32> {
    Box::new(move |x| x + init)
}
```

这段代码正好可以编译。关于 trait 对象的更多内容，请回顾第十八章的 顾及不同类型值的 trait 对象” 部分。

接下来让我们学习宏！

宏

我们已经在本书中使用过像 `println!` 这样的宏了，不过尚未深入探讨什么是宏以及它是如何工作的。**宏**（*Macro*）指的是 Rust 中一系列的功能：使用 `macro_rules!` 的 **声明宏**（*declarative macro*），和三种 **过程宏**（*procedural macro*）：

- 自定义 `#[derive]` 宏，用于在结构体和枚举上通过添加 `derive` 属性生成代码
- 类属性宏，定义可用于任意项的自定义属性
- 类函数宏，看起来像函数，但操作的是作为其参数传递的 token

我们会依次讨论每一种宏，不过首要的是，让我们看看为什么已经有了函数还需要宏呢？

宏和函数的区别

从根本上来说，宏是一种为写其他代码而写代码的方式，即所谓的 **元编程**（*metaprogramming*）。在附录 C 中会探讨 `derive` 属性，其生成各种 trait 的实现。我们也在本书中一直使用 `println!` 宏和 `vec!` 宏。所有的这些宏以 **展开** 的方式来生成比你所手写出的更多的代码。

元编程对于减少大量编写和维护的代码是非常有用的，它也扮演了函数所扮演的角色。但宏有一些函数所没有的附加能力。

一个函数签名必须声明函数参数的数量和类型。相比之下，宏能够接收可变数量的参数：用一个参数调用 `println!("hello")` 或用两个参数调用 `println!("hello {}", name)` 。而且，宏可以在编译器解析代码前展开，例如，宏可以在一个给定类型上实现 trait。而函数则不行，因为函数是在运行时被调用，而 trait 需要在编译时实现。

实现宏的缺点是与函数的定义相比宏的定义更复杂，因为你正在编写生成 Rust 代码的 Rust 代码。由于这样的间接性，宏定义通常要比函数定义更难阅读、理解和维护。

宏和函数的最后一个重要的区别是：在一个文件里调用宏 **之前** 必须定义它，或将其引入作用域，而函数则可以在任何地方定义和调用。

使用 `macro_rules!` 的声明宏用于通用元编程

Rust 最常用的宏形式是 **声明宏**（*declarative macros*）。它们有时也被称为 “macros by example”、“`macro_rules!` 宏” 或者就是 “macros”。其核心概念是，声明宏允许我们编写一些类似 Rust `match` 表达式的代码。正如在第六章讨论的那样，`match` 表达式是一种控制结构，其接收一个表达式，与表达式的结果进行模式匹配，然后根据模式匹配执行相关代码。宏也将一个值和包含相关代码的模式进行比较：此种情况下，该值是传递给宏的 Rust 源代码字面值；模式用于和前面提到的源代码字面值进行比较，一旦匹配成功，每个模式的相关代码会替换传递给宏的代码。所有这一切都发生于编译时。

可以使用 `macro_rules!` 来定义宏。让我们通过查看 `vec!` 宏定义来探索如何使用 `macro_rules!` 结构。第八章讲述了如何使用 `vec!` 宏来生成一个给定值的新 vector。例如，下面的宏用三个整数创建一个 vector：

```
let v: Vec<u32> = vec![1, 2, 3];
```

也可以使用 `vec!` 宏来构造两个整数的 vector 或五个字符串 slice 的 vector。但却无法使用函数做相同的事情，因为我们无法预先知道参数值的数量和类型。

在示例 20-35 中展示了 `vec!` 宏的一个稍微简化的定义。

文件名：src/lib.rs

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

示例 20-35: 一个 `vec!` 宏定义的简化版本

> 注意：标准库中实际定义的 `vec!` 包括预分配适正确数量内存的代码。这部分为代码优化，为了让示例简化，此处并没有包含在内。

`#[macro_export]` 注解表明只要导入了定义这个宏的 crate，该宏就应该是可用的。如果没有该注解，这个宏不能被引入作用域。

接着使用 `macro_rules!` 和宏名称开始宏定义，且所定义的宏并 **不带** 感叹号。名字后跟大括号表示宏定义体，在该例中宏名称是 `vec` 。

`vec!` 宏的结构和 `match` 表达式的结构类似。此处有一个分支模式 `( $( $x:expr ),* )` ，后跟 `=>` 以及和模式相关的代码块。如果模式匹配，该相关代码块将被展开。鉴于这个宏只有一个模式，那就只有一个有效匹配方式，其他任何模式方向（译者注：不匹配这个模式）都会导致错误。更复杂的宏会有不止一个分支。

宏定义中有效模式语法和在第十九章提及的模式语法是不同的，因为宏模式所匹配的是 Rust 代码结构而不是值。回过头来检查下示例 20-29 中模式片段什么意思。有关完整的宏模式语法，请查阅 Rust 参考。

首先，一对括号包含了整个模式。我们使用美元符号（`$`）在宏系统中声明一个变量来包含匹配该模式的 Rust 代码。美元符号明确表明这是一个宏变量而不是普通 Rust 变量。之后是一对括号，其捕获了符合括号内模式的值用以在替代代码中使用。`$()` 内则是 `$x:expr` ，其匹配 Rust 的任意表达式，并将该表达式命名为 `$x`。

在 `$()` 之后的逗号表示在每个与 `$()` 内代码匹配的实例之间必须出现一个字面量逗号分隔符。紧随逗号之后的 `*` 说明该模式匹配零个或更多个 `*` 之前的任何模式。

当以 `vec![1, 2, 3];` 调用宏时，`$x` 模式与三个表达式 `1`、`2` 和 `3` 对应进行了三次匹配。

现在让我们来看看与此分支模式相关联的代码块中的模式：在 `$()*` 部分，`temp_vec.push($x)` 会针对模式中每次匹配到 `$()` 的部分，生成零次或多次，取决于模式匹配到多少次。`$x` 由每个与之相匹配的表达式所替换。当以 `vec![1, 2, 3];` 调用该宏时，替换该宏调用所生成的代码会是下面这样：

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

我们已经定义了一个宏，其可以接收任意数量和类型的参数，同时可以生成能够创建包含指定元素的 vector 的代码。

要了解更多关于如何编写宏的信息，请查阅在线文档或其他资源，如由 Daniel Keep 发起、Lukas Wirth 继续维护的 “The Little Book of Rust Macros”。

用于从属性生成代码的过程宏

第二种形式的宏被称为 **过程宏**（*procedural macros*），因为它们更像函数（一种类型的过程）。过程宏接收 Rust 代码作为输入，在这些代码上进行操作，然后产生另一些代码作为输出，而非像声明式宏那样匹配对应模式然后以另一部分代码替换当前代码。有三种类型的过程宏，自定义派生（derive），类属性和类函数，它们的工作原理都类似。

创建过程宏时，其定义必须驻留在它们自己的具有特殊 crate 类型的 crate 中。这么做出于一些复杂的技术原因，将来我们希望能够消除这些限制。在示例 20-36 中展示了如何定义过程宏，其中 `some_attribute` 是一个使用特定宏变体的占位符。

文件名：src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

示例 20-36: 一个定义过程宏的例子

定义过程宏的函数接收一个 `TokenStream` 作为输入并生成 `TokenStream` 作为输出。`TokenStream` 是定义于 `proc_macro` crate 里代表一系列 token 的类型，Rust 默认携带了 `proc_macro` crate。这就是宏的核心：宏所处理的源代码组成了输入 `TokenStream`，宏生成的代码是输出 `TokenStream`。函数上还有一个属性；这个属性指明了我们创建的过程宏的类型。在同一 crate 中可以有多种的过程宏。

让我们看看不同种类的程序宏。我们将从一个自定义的派生宏开始，然后解释使其他形式不同的小差异。

如何编写自定义 `derive` 宏

让我们创建一个 `hello_macro` crate，其包含名为 `HelloMacro` 的 trait 和关联函数 `hello_macro`。不同于让用户为其每一个类型实现 `HelloMacro` trait，我们将会提供一个过程式宏以便用户可以使用 `#[derive(HelloMacro)]` 注解它们的类型来得到 `hello_macro` 函数的默认实现。该默认实现会打印 `Hello, Macro! My name is TypeName!`，其中 `TypeName` 为定义了 trait 的类型名。换言之，我们会创建一个 crate，使程序员能够写类似示例 20-37 中的代码。

文件名：src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```



示例 20-37: 我们 crate 的用户所写的能够使用过程式宏的代码

运行该代码将会打印 `Hello, Macro! My name is Pancakes!` 第一步是像下面这样新建一个库 crate：

```
$ cargo new hello_macro --lib
```

接下来，会定义 `HelloMacro` trait 以及其关联函数：

文件名：src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

示例 20-38: 一个我们会用于 `derive` 宏的简单 trait

现在有了一个 trait 及其相关函数。此时，crate 用户可以像示例 20-39 那样实现该 trait 来达到期望的功能，像这样：

文件名：src/main.rs

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

示例 20-39: 如果用户手动编写了一个 `HelloMacro` trait 实现看起来如何

然而，他们需要为每一个想要与 `hello_macro` 一同使用的类型编写实现的代码块。我们希望免去他们的这份工作。

另外，我们也无法为 `hello_macro` 函数提供一个能够打印实现了该 trait 的类型的名字的默认实现：Rust 没有反射能力，因此其无法在运行时获取类型名。我们需要一个在编译时生成代码的宏。

下一步是定义过程宏。在编写本部分时，过程式宏必须在其自己的 crate 内。该限制最终可能被取消。crate 及其宏 crate 的结构惯例如下：对于一个名为 `foo` 的 crate，其自定义 derive 过程宏 crate 通常命名为 `foo_derive`。让我们在 `hello_macro` 项目中，新建一个名为 `hello_macro_derive` 的 crate。

```
$ cargo new hello_macro_derive --lib
```

由于两个 crate 紧密相关，因此在 `hello_macro` 包的目录下创建过程式宏的 crate。如果改变在 `hello_macro` 中定义的 trait，同时也必须改变在 `hello_macro_derive` 中过程宏的实现。这两个包需要分别发布，编程人员如果使用这些包，则需要同时添加这两个依赖并将其引入作用域。我们也可以只用 `hello_macro` 包而将 `hello_macro_derive` 作为一个依赖，并重导出过程式宏的代码。但现在我们组织项目的方式使编程人员在无需 `derive` 功能时也能够单独使用 `hello_macro`。

我们需要声明 `hello_macro_derive` crate 为过程宏 (proc-macro) crate。我们还需要 `syn` 和 `quote` crate 中的功能，正如你即将看到的，需要将它们加到依赖中。将下面的代码加入到 `hello_macro_derive` 的 *Cargo.toml* 文件中。

文件名：hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "2.0"
quote = "1.0"
```

为定义一个过程式宏，请将示例 20-40 中的代码放在 `hello_macro_derive` crate 的 *src/lib.rs* 文件里面。注意这段代码在我们添加 `impl_hello_macro` 函数的定义之前是无法编译的。

文件名：hello_macro_derive/src/lib.rs



```
use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // 将 Rust 代码构建成我们可以操作的语法树。
    let ast = syn::parse(input).unwrap();

    // 生成 trait 的实现。
    impl_hello_macro(&ast)
}
```

示例 20-40: 大多数过程式宏处理 Rust 代码时所需的代码

注意我们将代码分成了 `hello_macro_derive` 和 `impl_hello_macro` 两个函数，前者负责解析 `TokenStream`，后者负责转换语法树：这使得编写过程宏更加方便。几乎你看到或者创建的每一个过程宏的外部函数（这里是 `hello_macro_derive`）中的代码都跟这里是一样的。你放入内部函数（这里是 `impl_hello_macro`）中的代码根据你的过程宏的设计目的会有所不同。

现在，我们已经引入了三个新的 crate：`proc_macro` 、 `syn` 和 `quote` 。Rust 自带 `proc_macro` crate，因此无需将其加到 *Cargo.toml* 文件的依赖中。`proc_macro` crate 是编译器提供用来读取和操作我们 Rust 代码的 API。

`syn` crate 将字符串中的 Rust 代码解析成为一个可以操作的数据结构。`quote` crate 则将 `syn` 解析的数据结构转换回 Rust 代码。这些 crate 让解析任何我们所要处理的 Rust 代码变得更加简单：为 Rust 编写完整的解析器并不是一件简单的工作。

当用户在一个类型上指定 `#[derive(HelloMacro)]` 时，`hello_macro_derive` 函数将会被调用。我们已使用 `proc_macro_derive` 注解该函数并指定名称 `HelloMacro`，该名称与我们的 trait 名称相匹配；这是大多数过程宏遵循的惯例。

该函数首先将来自 `TokenStream` 的 `input` 转换为一个我们可以解释和操作的数据结构。这正是 `syn` 派上用场的地方。`syn` 中的 `parse` 函数获取一个 `TokenStream` 并返回一个表示解析出的 Rust 代码的 `DeriveInput` 结构体。示例 20-41 展示了从字符串 `struct Pancakes;` 中解析出来的 `DeriveInput` 结构体的相关部分：

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

示例 20-41: 解析示例 20-37 中带有宏属性的代码时得到的 `DeriveInput` 实例

该结构体的字段展示了我们解析的 Rust 代码是一个类单元结构体，其 `ident`（identifier，表示名字）为 `Pancakes`。该结构体里面有更多字段描述了所有类型的 Rust 代码，查阅 `syn` 中 `DeriveInput` 的文档 以获取更多信息。

很快我们将定义 `impl_hello_macro` 函数，其用于构建所要包含在内的 Rust 新代码。但在此之前，注意其输出也是 `TokenStream`。所返回的 `TokenStream` 会被加到我们的 crate 用户所写的代码中，因此，当用户编译他们的 crate 时，他们会通过修改后的 `TokenStream` 获取到我们所提供的额外功能。

你可能也注意到了，当调用 `syn::parse` 函数失败时，我们用 `unwrap` 来使 `hello_macro_derive` 函数 panic。在错误时 panic 对过程宏来说是必须的，因为 `proc_macro_derive` 函数必须返回 `TokenStream` 而不是 `Result`，以此来符合过程宏的 API。这里选择用 `unwrap` 来简化了这个例子；在生产代码中，则应该通过 `panic!` 或 `expect` 来提供关于发生何种错误的更加明确的错误信息。

现在我们有了将注解的 Rust 代码从 `TokenStream` 转换为 `DeriveInput` 实例的代码，让我们来创建在注解类型上实现 `HelloMacro` trait 的代码，如示例 20-42 所示。

文件名：hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let generated = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    generated.into()
}
```

示例 20-42: 使用解析过的 Rust 代码实现 `HelloMacro` trait

我们得到一个包含以 `ast.ident` 作为注解类型名字（标识符）的 `Ident` 结构体实例。示例 20-33 中的结构体表明当 `impl_hello_macro` 函数运行于示例 20-31 中的代码上时 `ident` 字段的值是 `"Pancakes"`。因此，示例 20-34 中 `name` 变量会包含一个 `Ident` 结构体的实例，当打印时，会是字符串 `"Pancakes"`，也就是示例 20-37 中结构体的名称。

`quote!` 宏能让我们编写希望返回的 Rust 代码。`quote!` 宏执行的直接结果并不是编译器所期望的所以需要转换为 `TokenStream`。为此需要调用 `into` 方法，它会消费这个中间表示（intermediate representation，IR）并返回所需的 `TokenStream` 类型值。

这个宏也提供了一些非常酷的模板机制；我们可以写 `#name` ，然后 `quote!` 会以名为 `name` 的变量值来替换它。你甚至可以做一些类似常用宏那样的重复代码的工作。查阅 `quote` crate 的文档 来获取完整的介绍。

我们期望我们的过程式宏能够为通过 `#name` 获取到的用户注解类型生成 `HelloMacro` trait 的实现。该 trait 的实现有一个函数 `hello_macro` ，其函数体包括了我们期望提供的功能：打印 `Hello, Macro! My name is` 和注解的类型名。

此处所使用的 `stringify!` 为 Rust 内置宏。其接收一个 Rust 表达式，如 `1 + 2` ，然后在编译时将表达式转换为一个字符串常量，如 `"1 + 2"` 。这与计算表达式并接着将结果转换为 `String` 的 `format!` 或 `println!` 不同。有一种可能的情况是，所输入的 `#name` 可能是一个需要打印的表达式，因此我们用 `stringify!` 。`stringify!` 也能通过在编译时将 `#name` 转换为字符串字面值来节省一次内存分配。

此时，`cargo build` 应该都能成功编译 `hello_macro` 和 `hello_macro_derive` 。我们将这些 crate 连接到示例 20-31 的代码中来看看过程宏的行为！在 *projects* 目录下用 `cargo new pancakes` 命令新建一个二进制项目。需要将 `hello_macro` 和 `hello_macro_derive` 作为依赖加到 `pancakes` 包的 *Cargo.toml* 文件中去。如果你正将 `hello_macro` 和 `hello_macro_derive` 的版本发布到 crates.io 上，它们将是常规依赖；否则，则可以像下面这样将其指定为 `path` 依赖：

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

把示例 20-37 中的代码放在 *src/main.rs* ，然后执行 `cargo run`：其应该打印 `Hello, Macro! My name is Pancakes!`。其包含了该过程宏中 `HelloMacro` trait 的实现，而无需 `pancakes` crate 实现它；`#[derive(HelloMacro)]` 增加了该 trait 实现。

接下来，让我们探索一下其他类型的过程宏与自定义 `derive` 宏有何区别。

类属性宏

类属性（Attribute-Like）宏与自定义 `derive` 宏相似，不同之处在于它们不是为 `derive` 属性生成代码，而是允许你创建新的属性。它们也更为灵活；`derive` 只能用于结构体和枚举；属性还可以用于其它的项，比如函数。作为一个使用类属性宏的例子，可以创建一个名为 `route` 的属性用于注解 web 应用程序框架（web application framework）的函数：

```
#[route(GET, "/")]
fn index() {
```

`#[route]` 属性将由框架本身定义为一个过程宏。其宏定义的函数签名看起来像这样：

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

这里有两个 `TokenStream` 类型的参数；第一个用于属性内容本身，也就是 `GET, "/"` 部分。第二个是属性所标记的项：在本例中，是 `fn index() {}` 和剩下的函数体。

除此之外，类属性宏与自定义派生宏工作方式一致：创建 `proc-macro` crate 类型的 crate 并实现希望生成代码的函数！

类函数宏

类函数（Function-like）宏的定义看起来像函数调用的宏。类似于 `macro_rules!`，它们比函数更灵活；例如，可以接受未知数量的参数。然而 `macro_rules!` 宏只能使用之前 “使用 `macro_rules!` 的声明宏用于通用元编程” 介绍的类匹配的语法定义。类函数宏获取 `TokenStream` 参数，其定义使用 Rust 代码操纵 `TokenStream`，就像另两种过程宏一样。一个类函数宏例子是可以像这样被调用的 `sql!` 宏：

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

这个宏会解析其中的 SQL 语句并检查其是否是句法正确的，这是比 `macro_rules!` 可以做到的更为复杂的处理。`sql!` 宏会被定义为类似如此：

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

这类似于自定义 `derive` 宏的签名：获取括号中的 token，并返回希望生成的代码。

总结

呼！现在你的工具箱中有了一些 Rust 特性，虽然你可能不会经常使用它们，但在非常特定的情况下你会知道它们可用。我们介绍了几个复杂的主题，以便当你在错误信息建议或他人代码中遇到它们时，能够识别这些概念和语法。本章可作为查找解决方案的参考。

接下来，我们将再开始一个项目，将本书所学的所有内容付诸实践！

最后的项目：构建多线程 web server

这是一次漫长的旅途，不过我们已经抵达了本书的结尾。在本章中，我们将一同构建另一个项目，来展示最后几章所学，同时复习更早的章节。

作为最后的项目，我们将要实现一个返回 “hello” 的 web server，它在浏览器中看起来就如图 21-1 所示：

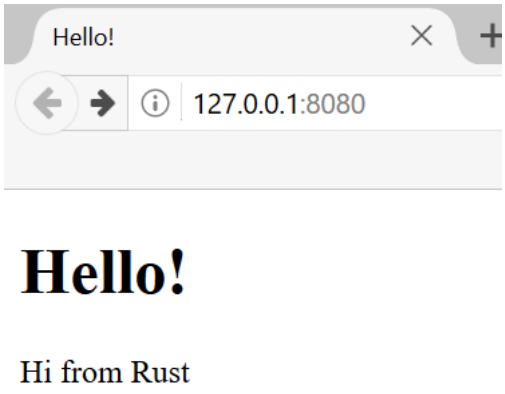


图 21-1: 我们最后将一起分享的项目

如下是构建 web server 的计划：

1. 学习一些 TCP 与 HTTP 知识
2. 在套接字（socket）上监听 TCP 请求
3. 解析少量的 HTTP 请求
4. 创建一个合适的 HTTP 响应
5. 通过线程池改善 server 的吞吐量

在开始之前，我们先提两点说明。首先，这里使用的方法并不是使用 Rust 构建 web server 的最佳方式。crates.io 上有很多可用于生产环境的 crate，它们提供了比我们所要编写的更为完整的 web server 和线程池实现。然而，本章的目的在于学习，而不是走捷径。因为 Rust 是一个系统编程语言，我们能够选择处理什么层次的抽象，并能够选择比其他语言可能或可用的层次更低的层次。

其次，我们不会在这里使用 async 和 await。构建线程池本身已经是一个相当大的挑战，无需再加入构建异步运行时的复杂度！不过，我们会指出 async 和 await 在本章中会遇到的一些问题上的可能应用。

因此我们将手动编写一个基础的 HTTP server 和线程池，以便学习将来可能用到的 crate 背后的通用理念和技术。

构建单线程 web server

首先让我们创建一个可运行的单线程 web server。在开始之前，我们将快速了解一下构建 web server 所涉及到的协议。这些协议的细节超出了本书的范畴，不过一个简单的概括会提供我们所需的信息。

web server 中涉及到的两个主要协议是 **超文本传输协议**（*Hypertext Transfer Protocol*，*HTTP*）和 **传输控制协议**（*Transmission Control Protocol*，*TCP*）。这两者都是 **请求 - 响应**（*request-response*）协议，也就是说，有 **客户端**（*client*）来初始化请求，并有 **服务端**（*server*）监听请求并向客户端提供响应。请求与响应的内容由协议本身定义。

TCP 是一个底层协议，它描述了信息如何从一个 server 到另一个的细节，不过其并不指定信息是什么。HTTP 构建于 TCP 之上，它定义了请求和响应的内容。从技术上讲可将 HTTP 用于其他协议之上，不过对于绝大部分情况，HTTP 通过 TCP 传输数据。我们将要做的就是处理 TCP 和 HTTP 请求与响应的原始字节数据。

监听 TCP 连接

我们的 web server 所需做的第一件事，就是监听 TCP 连接。标准库提供了 `std::net` 模块处理这些功能。让我们像往常一样新建一个项目：

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

现在，在 `src/main.rs` 输入示例 21-1 中的代码，作为一个开始。这段代码会在地址 `127.0.0.1:7878` 上监听传入的 TCP 流。当获取到传入的流，它会打印出 `Connection established!`：

文件名：src/main.rs

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

示例 21-1: 监听传入的流并在接收到流时打印信息

`TcpListener` 用于监听 TCP 连接。我们选择监听本地地址 `127.0.0.1:7878`。将这个地址拆开来看，冒号之前的部分是一个代表本机的 IP 地址（在每台计算机上，这个地址都指本机，并不特指作者的计算机），而 `7878` 是端口。选择这个端口出于两个原因：通常 HTTP 服务器不在这个端口上接受请求，所以它不太可能与你机器上运行的其它 web server 的端口冲突；而且 7878 在电话上打出来就是 “rust”（译者注：九宫格键盘上的英文）。

在这个场景中 `bind` 函数类似于 `new` 函数，在这里它返回一个新的 `TcpListener` 实例。这个函数叫做 `bind` 是因为，在网络领域，连接到要监听的端口称为“绑定到端口”（“binding to a port”）

`bind` 函数返回 `Result<T, E>`，这表明绑定可能会失败。例如，监听 80 端口需要管理员权限（非管理员用户只能监听大于 1023 的端口），所以如果尝试监听 80 端口而没有管理员权限，则会绑定失败。再比如，如果我们运行这个程序的两个实例，并因此有两个实例监听同一个端口，那么绑定也将失败。我们是出于学习目的来编写一个基础的服务器，不用关心处理这类错误，而仅仅使用 `unwrap` 在出现这些情况时直接停止程序。

`TcpListener` 的 `incoming` 方法返回一个迭代器，它提供了一系列的流（更准确的说是 `TcpStream` 类型的流）。**流**（*stream*）代表一个客户端和服务端之间打开的连接。**连接**（*connection*）代表客户端连接服务端、服务端生成响应以及服务端关闭连接的整个请求 / 响应过程。为此，我们会从 `TcpStream` 读取客户端发送了什么并接着向流发送响应以向客户端发回数据。总体来说，这个 `for` 循环会依次处理每个连接并产生一系列的流供我们处理。

目前，处理流的代码中也有一个 `unwrap` 调用，如果 `stream` 出现任何错误会终止程序；如果没有任何错误，程序会打印一条消息。下一个示例中，我们将为成功的情况增加更多功能。当客户端连接到服务端时，`incoming` 方法是可能返回错误的，因为我们实际上不是在遍历连接，而是遍历 **连接尝试**（*connection attempts*）。连接的尝试可能会因为多种原因不能成功，其中大部分是操作系统相关的。例如，很多系统限制它所能支持的同时打开的连接数，超出数量限制的新连接尝试会产生错误，直到一些现有的连接关闭为止。

让我们试试这段代码！首先在终端执行 `cargo run`，接着在浏览器中打开 `127.0.0.1:7878`。浏览器会显示出看起来类似于“连接重置”（“Connection reset”）的错误信息，因为 server 目前并没响应任何数据。如果我们观察终端，会发现当浏览器连接我们的服务端时，会打印出一系列的信息！

```
     Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

有时，对于一次浏览器请求可能会打印出多条信息；原因可能是浏览器不仅请求页面，还请求其他资源，比如出现在浏览器标签页中的 *favicon.ico* 图标。

这也可能是因为浏览器尝试多次连接服务端，因为服务端没有响应任何数据。当 `stream` 在循环结束时离开作用域并被丢弃，其连接将作为 `drop` 实现的一部分被关闭。浏览器有时通过重连来处理关闭的连接，因为这些问题可能是暂时的。

浏览器有时还会在不发送任何请求的情况下打开多个连接，以便在**稍后**发送请求时能够更快地开始。出现这种情况时，我们的服务端会看到每个连接，而不管该连接上是否有请求。例如，许多基于 Chrome 的浏览器版本都会这样做；你可以通过使用私人浏览模式或更换其他浏览器来禁用该优化。

重要的是，我们已经成功获取了一个 TCP 连接的句柄！

记得当运行完特定版本的代码后，使用 ctrl-c 来停止程序。然后在你完成每次代码修改后，通过运行 `cargo run` 命令重新启动程序，以确保你正在运行最新的代码。

读取请求

让我们实现读取来自浏览器请求的功能！为了将首先获取连接和接着对连接采取操作两项职责分离，我们将开始写一个新函数来处理连接。在这个新的 `handle_connection` 函数中，我们从 TCP 流中读取数据，并打印出来，以便观察浏览器发送过来的数据。将代码修改为如示例 21-2 所示：

文件名：src/main.rs

```
use std::{
    io::{BufReader, prelude::*},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("Request: {http_request:#?}");
}
```

示例 21-2: 读取 `TcpStream` 并打印数据

这里将 `std::io::prelude` 和 `std::io::BufReader` 引入作用域，来获取读写流所需的 trait 和类型。在 `main` 函数的 `for` 循环中，相比获取到连接时打印信息，现在调用新的 `handle_connection` 函数并向其传递 `stream`。

在 `handle_connection` 中，我们新建了一个 `BufReader` 实例来封装一个 `stream` 的引用。`BufReader` 通过替我们管理 `std::io::Read` trait 方法的调用增加了缓冲。

我们创建了一个 `http_request` 变量来收集浏览器发送给服务端的请求行。这里增加了 `Vec<_>` 类型注解表明希望将这些行收集到一个 vector 中。

`BufReader` 实现了 `std::io::BufRead` trait，它提供了 `lines` 方法。`lines` 方法通过遇到换行符（newline）字节就切分数据流来返回一个 `Result<String, std::io::Error>` 的迭代器。为了获取每一个 `String`，我们通过 map 并 `unwrap` 每一个 `Result`。如果数据不是有效的 UTF-8 编码或者读取流遇到问题时，`Result` 可能是一个错误。同理，用于生产环境的程序应该更优雅地处理这些错误，不过出于简单的目的我们选择在错误情况下停止程序。

浏览器通过连续发送两个换行符来代表一个 HTTP 请求的结束，所以为了从流中获取一个请求，我们会读取行直到遇到一个空字符串的行。一旦将这些行收集进 vector，就可以使用友好的 debug 格式化打印它们，以便看看浏览器发送给服务端的指令。

让我们试一试！启动程序并再次在浏览器中发起请求。注意浏览器中仍然会出现错误页面，不过终端中程序的输出现在看起来像这样：

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/hello`
Request: [
    "GET / HTTP/1.1",
    "Host: 127.0.0.1:7878",
    "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:99.0)
Gecko/20100101 Firefox/99.0",
    "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,*/*;q=0.8",
    "Accept-Language: en-US,en;q=0.5",
    "Accept-Encoding: gzip, deflate, br",
    "DNT: 1",
    "Connection: keep-alive",
    "Upgrade-Insecure-Requests: 1",
    "Sec-Fetch-Dest: document",
    "Sec-Fetch-Mode: navigate",
    "Sec-Fetch-Site: none",
    "Sec-Fetch-User: ?1",
    "Cache-Control: max-age=0",
]
```

根据不同的浏览器，可能会出现稍微不同的输出。现在我们打印出了请求数据，可以通过观察第一行 `GET` 之后的路径就可以看出为何会从浏览器得到多个连接。如果重复的连接都是请求 `/`，就知道了浏览器尝试重复获取 `/` 因为它没有从程序得到响应。

让我们拆开请求数据来理解浏览器向程序请求了什么。

仔细观察 HTTP 请求

HTTP 是一个基于文本的协议，同时一个请求有如下格式：

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

第一行叫做 **请求行**（*request line*），它存放了客户端请求了什么的信息。请求行的第一部分是所使用的 *method*，比如 `GET` 或 `POST`，这描述了客户端如何进行请求。这里客户端使用了 `GET` 请求，表明它在请求信息。

请求行接下来的部分是 `/`，它代表客户端请求的 **统一资源标识符**（*Uniform Resource Identifier*，*URI*）：URI 大体上但也不完全类似于 URL（**统一资源定位符**，*Uniform Resource Locators*）。URI 和 URL 之间的区别对于本章的目的来说并不重要，不过 HTTP 规范使用术语 URI，所以这里可以简单的将 URL 理解为 URI。

最后一部分是客户端使用的 HTTP 版本，然后请求行以 **CRLF 序列** （CRLF 代表回车和换行，*carriage return line feed*，这是打字机时代的术语！）结束。CRLF 序列也可以写成`\r\n`，其中`\r`是回车符，`\n`是换行符。CRLF 序列将请求行与其余请求数据分开。请注意，打印 CRLF 时，我们会看到一个新行开始，而不是`\r\n`。

观察目前为止运行程序所接收到的请求行数据，可以看到`GET`是 method，`/`是请求 URI，而`HTTP/1.1`是版本。

从`Host:`开始的其余的行是 headers；`GET`请求没有 body。

如果你希望的话，可以尝试用不同的浏览器发送请求，或请求不同的地址，比如`127.0.0.1:7878/test`，来观察请求数据如何变化。

现在我们知道了浏览器请求了什么。让我们返回一些数据！

编写响应

我们将实现在客户端请求的响应中发送数据的功能。响应具有如下格式：

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

第一行叫做 **状态行**（*status line*），它包含响应的 HTTP 版本、一个数字状态码（status code）用以总结请求的结果和一个描述之前状态码的文本原因短语（reason phrase）。CRLF 序列之后是任意 header，另一个 CRLF 序列，和响应的 body。

这里是一个使用 HTTP 1.1 版本的响应例子，其状态码为 200，原因短语为 OK，没有 header，也没有 body：

```
HTTP/1.1 200 OK\r\n\r\n
```

状态码 200 是一个标准的成功响应。这些文本是一个微型的成功 HTTP 响应。让我们将这些文本写入流作为成功请求的响应！在`handle_connection`函数中，我们需要去掉打印请求数据的`println!`，并替换为示例 21-3 中的代码：

文件名：src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write_all(response.as_bytes()).unwrap();
}
```

示例 21-3: 将一个微型成功 HTTP 响应写入流

新代码中的第一行定义了变量 `response` 来存放成功消息的数据。接着，我们在 `response` 上调用 `as_bytes` 将字符串数据转换为字节数组。因为 `stream` 的 `write_all` 方法获取一个 `&[u8]` 并直接将这些字节发送给连接。因为 `write_all` 操作可能会失败，所以像之前那样对任何错误结果使用 `unwrap`。同理，在真实世界的应用中这里需要添加错误处理。

有了这些修改，运行我们的代码并进行请求。由于不再向终端打印任何数据，所以不会再看到除了 Cargo 以外的任何输出。不过当在浏览器中加载 *127.0.0.1:7878* 时，会得到一个空页面而不是错误。我们刚刚手写了接收 HTTP 请求并发送响应！

返回真正的 HTML

让我们实现不只是返回空页面的功能。在项目根目录创建一个新文件，*hello.html*，不是在 *src* 目录。在此可以放入任何你期望的 HTML 内容；示例 21-4 展示了一个可能的文本：

文件名：hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

示例 21-4: 一个示例 HTML 文件作为响应返回

这是一个极简 HTML5 文档包含一个标题和一小段文本。为了在服务端接收请求时返回它，需要如示例 21-5 所示修改 `handle_connection` 来读取 HTML 文件，将其加入到响应的 body 中并发送：

文件名：src/main.rs

```
use std::{
    fs,
    io::{BufReader, prelude::*},
    net::{TcpListener, TcpStream},
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();
```

```
    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

示例 21-5: 将 *hello.html* 的内容作为响应 body 发送

我们在开头 `use` 语句将标准库的文件系统模块 `fs` 引入作用域。打开和读取文件的代码应该看起来很熟悉；在第十二章 `I/O` 项目中的 Listing 12-4 就用到了类似的代码。

接下来，使用 `format!` 将文件内容加入到将要写入流的成功响应的 body 中。为了确保构造出有效的 HTTP 响应，我们添加了 `Content-Length` header，其值设为响应 `body` 的大小，在这里也就是 `hello.html` 文件的大小。

使用 `cargo run` 运行程序，在浏览器加载 *127.0.0.1:7878*，你应该会看到渲染后的 HTML！

目前忽略了 `http_request` 中的请求数据并无条件的发送了 HTML 文件的内容。这意味着如果尝试在浏览器中请求 *127.0.0.1:7878/something-else* 也会得到同样的 HTML 响应。目前我们的 server 的作用是非常有限的，也不是大部分 server 所做的那样；让我们检查请求并只对格式良好（well-formed）的请求 `/` 发送 HTML 文件。

验证请求并有选择的进行响应

目前我们的 web server 不管客户端请求什么都会返回相同的 HTML 文件。让我们增加在返回 HTML 文件前检查浏览器是否请求 */*，并在其请求任何其他内容时返回错误的功能。为此需要如示例 21-6 那样修改 `handle_connection`。新代码接收到的请求的内容与已知的 */* 请求做比较，并增加了 `if` 和 `else` 块来区别处理请求：

文件名：src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

示例 21-6: 以不同于其它请求的方式处理 */* 请求

我们只看 HTTP 请求的第一行，所以不同于将整个请求读取进 vector 中，这里调用 `next` 从迭代器中获取第一项。第一个 `unwrap` 负责处理 `Option` 并在迭代器没有项时停止程序。第二个 `unwrap` 处理 `Result` 并与示例 21-2 中增加的 `map` 中的 `unwrap` 有着相同的效果。

接下来检查 `request_line` 是否等于一个 / 路径的 GET 请求。如果是，`if` 代码块返回 HTML 文件的内容。

如果 `request_line` **不** 等于一个 / 路径的 GET 请求，就说明接收的是其它请求。我们之后会在 `else` 块中增加代码来响应所有其他请求。

现在如果运行代码并请求 *127.0.0.1:7878*，就会得到 *hello.html* 中的 HTML。如果进行任何其他请求，比如 *127.0.0.1:7878/something-else*，则会得到像运行示例 21-1 和 21-2 中代码那样的连接错误。

现在向示例 21-7 的 `else` 块增加代码来返回一个带有 404 状态码的响应，这代表了所请求的内容没有找到。接着也会返回一个 HTML 向浏览器终端用户渲染该响应。

文件名：src/main.rs

```
    // --snip--
    } else {
        let status_line = "HTTP/1.1 404 NOT FOUND";
        let contents = fs::read_to_string("404.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    }
```

示例 21-7: 对于任何不是 / 的请求返回 404 状态码的响应和错误页面

这里，响应的状态行有状态码 404 和原因短语 `NOT FOUND`。仍然没有返回任何 header，而其 body 将是 *404.html* 文件中的 HTML。需要在 *hello.html* 同级目录创建 *404.html* 文件作为错误页面；这一次也可以随意使用任何 HTML 或使用示例 21-8 中的示例 HTML：

文件名：404.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

示例 21-8: 任何 404 响应所返回错误页面内容样例

有了这些修改，再次运行服务端。请求 *127.0.0.1:7878* 应该会返回 *hello.html* 的内容，而对于任何其他请求，比如 *127.0.0.1:7878/foo*，应该会返回 *404.html* 中的错误 HTML。

稍加重构

目前 `if` 和 `else` 块中的代码存在大量重复：他们都读取文件并将其内容写入流。唯一的区别是状态行和文件名。为使代码更简洁，将这些区别分别提取到各自的 `if` 和 `else` 中，对状态行和文件名变量赋值；然后在读取文件和写入响应的代码中无条件地使用这些变量。重构后取代了大段 `if` 和 `else` 块代码后的结果如示例 21-9 所示：

文件名：src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

示例 21-9: 重构使得 `if` 和 `else` 块中只包含两个情况所不同的代码

现在 `if` 和 `else` 块所做的唯一的事就是在一个元组中返回合适的状态行和文件名的值；接着使用第十九章讲到的使用模式的 `let` 语句通过解构元组的两部分给 `filename` 和 `header` 赋值。

之前读取文件和写入响应的冗余代码现在位于 `if` 和 `else` 块之外，并会使用变量 `status_line` 和 `filename`。这样更易于观察这两种情况真正有何不同，还意味着如果需要改变如何读取文件或写入响应时只需要更新一处的代码。示例 21-9 中代码的行为与示例 21-8 完全相同。

好极了！我们现在有了一个 40 行左右 Rust 代码的小而简单的服务端，它对一个请求返回页面内容而对所有其他请求返回 404 响应。

目前服务端运行于单线程中，这意味着它一次只能处理一个请求。让我们模拟一些慢请求来看看这为何会成为一个问题。然后我们将修复它使得服务端能够同时处理多个请求。

将单线程 server 变为多线程 server

目前服务端会依次处理每一个请求，意味着它在完成第一个连接的处理之前不会处理第二个连接。如果服务端正接收越来越多的请求，这类串行操作会使性能越来越差。如果一个请求花费很长时间来处理，随后而来的请求则不得不等待这个长请求结束，即便这些新请求可以很快就处理完。我们需要修复这种情况，不过首先让我们实际尝试一下这个问题。

在当前服务端实现中模拟慢请求

让我们看看一个慢请求如何影响当前服务端实现中的其他请求。示例 21-10 通过模拟慢响应实现了 */sleep* 请求处理，它会使服务端在响应之前休眠五秒。

文件名：src/main.rs

```
use std::{
    fs,
    io::{BufReader, prelude::*},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --snip--
}
```

示例 21-10: 通过休眠五秒来模拟慢请求

从 `if` 切换到 `match` 后现在有三个分支了。我们需要显式地匹配一个 slice 的 `request_line` 以模式匹配字符串字面值。`match` 不会像相等方法那样自动引用和解引用。

第一个分支与示例 21-9 中的 `if` 代码块相同。第二个分支匹配一个 */sleep* 请求。当接收到这个请求时，server 在渲染成功 HTML 页面之前会先休眠五秒。第三个分支与示例 21-9 中的 `else` 代码块相同。

现在就可以真切的看出我们的服务端有多么的原始：真实的库将会以更简洁的方式处理多个请求的识别！

使用 `cargo run` 启动服务端，并接着打开两个浏览器窗口：一个请求 *http://127.0.0.1:7878/_* 而另一个请求 http://127.0.0.1:7878/sleep_ 。如果像之前一样多次请求 */*，会发现响应的比较快速。不过如果请求 */sleep* 之后再请求 */*，就会看到 */* 会等待直到 `sleep` 休眠完五秒之后才响应。

有多种技术可以用来避免所有请求都排在慢请求之后，包括我们在第十七章中所使用的异步；我们将要实现的一个便是线程池。

使用线程池改善吞吐量

线程池（*thread pool*）是一组预先分配的等待或准备处理任务的线程。当程序收到一个新任务，线程池中的一个线程会被分配该任务，并负责处理它。其余线程在该线程处理任务的同时可以处理任何其他接收到的任务。当第一个线程处理完任务时，它会返回空闲线程池中等待处理新任务。线程池允许我们并发处理连接，提高服务端的吞吐量。

我们会将池中线程限制为较少的数量，以防拒绝服务（Denial of Service，DoS）攻击；如果程序为每一个接收的请求都新建一个线程，某人向服务端发起千万级的请求时会耗尽服务器的资源并导致请求处理陷入停滞。

不同于分配无限的线程，线程池中将有固定数量的等待线程。当新进请求时，将请求发送到线程池中做处理。线程池会维护一个接收请求的队列。每一个线程会从队列中取出一个请求，处理请求，接着向队列获取下一个请求。通过这种设计，则可以并发处理 `N` 个请求，其中 `N` 为线程数。如果每一个线程都在响应慢请求，之后的请求仍然会阻塞队列，不过相比之前已经增加了能处理的慢请求的数量。

这个设计仅仅是多种改善 web 服务端吞吐量的方法之一。其他可供探索的方法有 fork/join 模型（fork/join model）、单线程异步 I/O 模型（single-threaded async I/O model）或者多线程异步 I/O 模型（multi-threaded async I/O model）。如果你对这个主题感兴趣，则可以阅读更多关于其他解决方案的内容并尝试实现它们；对于一个像 Rust 这样的底层语言，所有这些方法都是可行的。

在开始之前，让我们讨论一下线程池应用看起来如何。当尝试设计代码时，首先编写客户端接口（client interface）有助于指导代码设计。以期望的调用方式来构建 API 代码的结构，接着在这个结构之内实现功能，而不是先实现功能再设计公有 API。

类似于第十二章项目中使用的测试驱动开发。这里将要使用编译器驱动开发（compiler-driven development）。我们将编写调用所期望的函数的代码，接着观察编译器错误告诉我们接下来需要修改什么使得代码可以工作。不过在开始之前，我们将探索不会作为起点使用的技术。

为每一个请求分配线程

首先，让我们探索一下如果为每一个连接都创建一个线程的代码看起来如何。这并不是最终方案，因为正如之前讲到的它会潜在的分配无限的线程，不过这是一个可用的多线程服务端的起点。接着我们会增加线程池作为改进，这样比较两个方案将会更容易。示例 21-11 展示了 `main` 的改变，它在 `for` 循环中为每一个流分配了一个新线程进行处理：

文件名：src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
```

```
        });
    }
}
```

示例 21-11: 为每一个流新建一个线程

正如第十六章讲到的，`thread::spawn` 会创建一个新线程并在其中运行闭包中的代码。如果运行这段代码并在在浏览器中加载 */sleep*，接着在另两个浏览器标签页中加载 */*，确实会发现 */* 请求不必等待 */sleep* 结束。不过正如之前提到的，这最终会使系统崩溃因为我们会无限制地创建新线程。

你可能也会回想起第十七章中正是这一类情况才是 async 和 await 真正闪光的地方！在我们用线程池构建项目时请记住并思考这与异步有什么不同或相同的地方。

创建有限数量的线程

我们期望线程池以类似且熟悉的方式工作，以便从线程切换到线程池并不会对使用该 API 的代码做出大幅修改。示例 21-12 展示我们希望用来替换 `thread::spawn` 的 `ThreadPool` 结构体的假想接口：

文件名：src/main.rs



```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

示例 21-12: 假想的 `ThreadPool` 接口

我们使用 `ThreadPool::new` 创建一个具有可配置线程数的新线程池，在这里是四。这样在 `for` 循环中，`pool.execute` 有着类似 `thread::spawn` 的接口，它获取一个线程池运行于每一个流的闭包。我们需要实现 `pool.execute`，使其能够接收闭包并将其传递给线程池中的线程执行。这段代码还不能编译，但我们可以尝试让编译器指导我们如何修复它。

采用编译器驱动开发构建 `ThreadPool`

继续并对示例 21-12 中的 *src/main.rs* 做出修改，并利用来自 `cargo check` 的编译器错误来驱动开发。下面是我们得到的第一个错误：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
  --> src/main.rs:11:16
   |
11 |     let pool = ThreadPool::new(4);
```

```
   |                ^^^^^^^^^^ use of undeclared type `ThreadPool`

For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` (bin "hello") due to 1 previous error
```

太好了！这个错误告诉我们需要一个 `ThreadPool` 类型或模块，所以我们现在就来构建一个。`ThreadPool` 的实现会与 web 服务端的特定工作相独立。所以让我们从 `hello` crate 切换到存放 `ThreadPool` 实现的新库 crate。切换为库 crate 之后，我们就可以在任何工作中使用这个单独的线程池库，而不仅仅是处理网络请求。

创建 *src/lib.rs* 文件，它包含了目前可用的最简单的 `ThreadPool` 定义：

文件名：src/lib.rs

```
pub struct ThreadPool;
```

接着编辑 *main.rs* 文件通过在 *src/main.rs* 的开头增加如下代码将 `ThreadPool` 从库 crate 引入作用域：

文件名：src/main.rs

```
use hello::ThreadPool;
```

这仍然不能工作，再次尝试运行来得到下一个需要解决的错误：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for struct
`ThreadPool` in the current scope
  --> src/main.rs:12:28
   |
12 |     let pool = ThreadPool::new(4);
   |                            ^^^ function or associated item not found in
`ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` (bin "hello") due to 1 previous error
```

此错误表明下一步是为 `ThreadPool` 创建一个叫做 `new` 的关联函数。我们还知道 `new` 需要有一个参数可以接受 `4`，而且 `new` 应该返回 `ThreadPool` 实例。让我们实现拥有此特征的最小化 `new` 函数：

文件夹：src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

这里选择 `usize` 作为 `size` 参数的类型，因为我们知道线程数为负没有意义。我们还知道将使用 `4` 作为线程集合的元素数量，这也就是使用 `usize` 类型的原因，如第三章 “整型” 部分所讲。

再次编译检查这段代码：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for struct `ThreadPool` in the
current scope
  --> src/main.rs:17:14
   |
17 |         pool.execute(|| {
   |         -----^^^^^^^ method not found in `ThreadPool`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` (bin "hello") due to 1 previous error
```

这里发生错误是因为并没有 `ThreadPool` 上的 `execute` 方法。回忆 “创建有限数量的线程” 部分我们决定线程池应该有与 `thread::spawn` 类似的接口，同时我们将实现 `execute` 函数来获取传递的闭包并将其传递给池中的空闲线程执行。

我们会在 `ThreadPool` 上定义 `execute` 函数来获取一个闭包参数。回忆第十三章的 “将捕获的值移出闭包和 `Fn` trait” 部分，闭包作为参数时可以使用三个不同的 trait：`Fn`、`FnMut` 和 `FnOnce`。我们需要决定这里应该使用哪种闭包。最终需要实现的类似于标准库的 `thread::spawn`，所以我们可以观察 `thread::spawn` 的签名在其参数中使用了何种 bound。查看文档会发现：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

`F` 是这里我们关心的参数；`T` 与返回值有关所以我们并不关心。考虑到 `spawn` 使用 `FnOnce` 作为 `F` 的 trait bound，这可能也是我们需要的，因为最终会将传递给 `execute` 的参数传给 `spawn`。因为处理请求的线程只会执行闭包一次，这也进一步确认了 `FnOnce` 是我们需要的 trait，这里符合 `FnOnce` 中 `Once` 的意思。

`F` 还有 trait bound `Send` 和生命周期绑定 `'static`，这对我们的情况也是有意义的：需要 `Send` 来将闭包从一个线程转移到另一个线程，而 `'static` 是因为并不知道线程会执行多久。让我们编写一个使用带有这些 bound 的泛型参数 `F` 的 `ThreadPool` 的 `execute` 方法：

文件名：src/lib.rs

```
impl ThreadPool {
    // --snip--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

`FnOnce` trait 仍然需要之后的 `()`，因为这里的 `FnOnce` 代表一个没有参数也没有返回值的闭包。正如函数的定义，返回值类型可以从签名中省略，不过即便没有参数也需要括号。

这里再一次增加了 `execute` 方法的最小化实现：它没有做任何工作，只是尝试让代码能够编译。再次进行检查：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.24s
```

现在就只有警告了！这意味着能够编译了！注意如果尝试 `cargo run` 运行程序并在浏览器中发起请求，仍会在浏览器中出现在本章开始时那样的错误。这个库实际上还没有调用传递给 `execute` 的闭包！

> 一个你可能听说过的关于像 Haskell 和 Rust 这样有严格编译器的语言的说法是 “如果代码能够编译，它就能工作”。不过这个说法并不是普适的。我们的项目可以编译，不过它完全没有做任何工作！如果构建一个真实且功能完整的项目，则需花费大量的时间来开始编写单元测试来检查代码能否编译 **并且** 拥有期望的行为。

思考一下：如果这里要执行的是一个 `future` 而不是闭包会有什么不同?

在 `new` 中验证线程池的线程数量

这里并没有对 `new` 和 `execute` 的参数做任何操作。让我们用期望的行为来实现这些函数。以考虑 `new` 作为开始。之前选择使用无符号类型作为 `size` 参数的类型，因为线程数为负的线程池没有意义。然而，线程数为零的线程池同样没有意义，不过零是一个完全有效的 `usize` 值。让我们增加在返回 `ThreadPool` 实例之前检查 `size` 是否大于零的代码，并使用 `assert!` 宏在得到零时 panic，如示例 21-13 所示：

文件名：src/lib.rs

```
impl ThreadPool {
    /// 创建一个新的线程池。
    ///
    /// size 是池中线程的数量。
    ///
    /// # Panics
    ///
    /// 如果 size 为 0，`new` 方法会 panic。
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}
```

示例 21-13: 实现 `ThreadPool::new` 在 `size` 为零时 panic

这里也用文档注释为 `ThreadPool` 增加了一些文档。注意这里遵循了良好的文档实践并增加了一个部分来提示函数会 panic 的情况，正如第十四章所讨论的。尝试运行 `cargo doc --open` 并点击 `ThreadPool` 结构体来查看生成的 `new` 的文档看起来如何！

相比像这里使用 `assert!` 宏，也可以让 `new` 像之前 I/O 项目中示例 12-9 中 `Config::build` 那样将 `new` 更改为 `build` 并返回一个 `Result`，不过在这里我们选择创建一个没有任何线程的线程池应该是不可恢复的错误。如果你想做的更好，尝试编写一个采用如下签名的名为 `build` 的函数来对比一下 `new` 函数：

```
pub fn build(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

分配空间以存储线程

现在我们已经有了一种方法来确保线程池中的线程数有效，就可以实际创建这些线程并在返回结构体之前将它们存储在 `ThreadPool` 结构体中。不过如何 “存储” 一个线程？让我们再看看 `thread::spawn` 的签名：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

`spawn` 返回 `JoinHandle<T>`，其中 `T` 是闭包返回的类型。尝试使用 `JoinHandle` 来看看会发生什么。在我们的情况中，传递给线程池的闭包会处理连接并不返回任何值，所以 `T` 将会是单元类型 `()`。

示例 21-14 中的代码可以编译，不过实际上还并没有创建任何线程。我们改变了 `ThreadPool` 的定义来存放一个 `thread::JoinHandle<()>` 的 vector 实例，使用 `size` 容量来初始化，并设置一个 `for` 循环来运行创建线程的代码，并返回包含这些线程的 `ThreadPool` 实例：

文件名：src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // 创建一些线程并将它们存入 vector 中。
        }

        ThreadPool { threads }
    }
```

```
    // --snip--
}
```



示例 21-14: 为 `ThreadPool` 创建一个 vector 来存放线程

这里将 `std::thread` 引入库 crate 的作用域，因为使用了 `thread::JoinHandle` 作为 `ThreadPool` 中 vector 元素的类型。

在得到了有效的数量之后，`ThreadPool` 新建一个存放 `size` 个元素的 vector。`with_capacity` 函数与 `Vec::new` 做了同样的工作，不过有一个重要的区别：它为 vector 预先分配空间。因为已经知道了 vector 中需要 `size` 个元素，预先进行分配比仅仅 `Vec::new` 要稍微有效率一些，因为 `Vec::new` 随着插入元素而重新改变大小。

如果再次运行 `cargo check`，它应该会成功。

`Worker` 结构体负责将代码从 `ThreadPool` 传递给线程

示例 21-14 的 `for` 循环中留下了一个关于创建线程的注释。这里，我们来看看如何实际创建线程。标准库提供了 `thread::spawn` 作为创建线程的方法，`thread::spawn` 期望获取一些一旦创建线程就应该执行的代码。然而，我们希望开始线程并使其等待稍后传递的代码。标准库的线程实现并没有包含这么做的方法；我们必须手动实现。

我们将要实现的行为是创建线程并稍后发送代码，这会在 `ThreadPool` 和线程间引入一个新数据类型来管理这种新行为。这个数据结构称为 *Worker*，这是一个池实现中的常见概念。`Worker` 会获取需要运行的代码，并在该 worker 的线程中运行该代码。

想象一下在餐馆厨房工作的员工：员工等待来自顾客的订单，他们负责接单并完成它们。

不同于在线程池中储存一个 `JoinHandle<()>` 实例的 vector，我们会储存 `Worker` 结构体的实例。每一个 `Worker` 会储存一个单独的 `JoinHandle<()>` 实例。接着会在 `Worker` 上实现一个方法，该方法将闭包发送到已经运行的线程中执行。我们还会赋予每个 worker 一个 `id`，这样就可以在日志和调试中区别线程池中的不同 `Worker` 的实例。

如下是创建 `ThreadPool` 时会发生的新过程。在通过如下方式设置完 `Worker` 之后，我们会实现向线程发送闭包的代码：

1. 定义存放 `id` 和 `JoinHandle<()>` 的 `Worker` 结构体。
2. 修改 `ThreadPool` 存放一个 `Worker` 实例的 vector。
3. 定义 `Worker::new` 函数，它获取一个 `id` 数字并返回一个带有 `id` 和用空闭包分配的线程的 `Worker` 实例。
4. 在 `ThreadPool::new` 中，使用 `for` 循环计数生成 `id`，使用这个 `id` 新建 `Worker`，并储存进 vector 中。

如果你渴望挑战，在查示例 21-15 中的代码之前尝试自己实现这些修改。

准备好了吗？示例 21-15 就是一个做出了上述修改的例子：

文件名：src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
```

```
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}
```

示例 21-15: 修改 `ThreadPool` 存放 `Worker` 实例而不是直接存放线程

这里将 `ThreadPool` 中字段名从 `threads` 改为 `workers`，因为它现在存储 `Worker` 而不是 `JoinHandle<()>`。使用 `for` 循环中的计数作为 `Worker::new` 的参数，并将每一个新建的 `Worker` 存储在叫做 `workers` 的 vector 中。

`Worker` 结构体和其 `new` 函数是私有的，因为外部代码（比如 *src/main.rs* 中的 server）并不需要知道关于 `ThreadPool` 中使用 `Worker` 结构体的实现细节。`Worker::new` 函数使用 `id` 参数并存储了使用一个空闭包创建的 `JoinHandle<()>` 实例。

> 注意：如果操作系统因为没有足够的系统资源而无法创建线程时，`thread::spawn` 会 panic。这会导致整个 server panic，即使一些线程可能创建成功了。出于简单的考虑，这个行为是可行的，不过在一个生产级别的线程池实现中，你可能会希望使用 `std::thread::Builder` 和其 `spawn` 方法来返回一个 `Result`。

这段代码能够编译并用指定给 `ThreadPool::new` 的参数创建存储了一系列的 `Worker` 实例，不过 **仍然** 没有处理 `execute` 中得到的闭包。让我们聊聊接下来怎么做。

使用信道向线程发送请求

下一个需要解决的问题是传递给 `thread::spawn` 的闭包完全没有做任何工作。目前，我们在 `execute` 方法中获得期望执行的闭包，不过在创建 `ThreadPool` 的过程中创建每一个 `Worker` 时需要向 `thread::spawn` 传递一个要运行的闭包。

我们希望刚创建的 `Worker` 结构体能够从 `ThreadPool` 的队列中获取需要执行的代码，并发送到线程中执行。

在第十六章，我们学习了 **信道** —— 一个沟通两个线程的简单手段 —— 对于这个例子来说则是绝佳的选择。这里信道将充当任务队列的作用，`execute` 将通过 `ThreadPool` 向其中线程正在寻找工作的 `Worker` 实例发送任务。计划如下：

1. `ThreadPool` 会创建一个信道并持有发送端。
2. 每个 `Worker` 将持有接收端。
3. 新建一个 `Job` 结构体来存放用于向信道中发送的闭包。
4. `execute` 方法会在发送者发出期望执行的工作。
5. 在线程中，`Worker` 会遍历接收者并执行任何接收到的工作。

让我们以在 `ThreadPool::new` 中创建信道并让 `ThreadPool` 实例充当发送者开始，如示例 21-16 所示。`Job` 结构体目前为空，但它将作为我们通过通道发送的类型：

文件名：src/lib.rs

```
use std::{sync::mpsc, thread};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}
```

示例 21-16: 修改 `ThreadPool` 来储存一个传输 `Job` 实例的发送者

在 `ThreadPool::new` 中，新建了一个信道，并接着让线程池在接收端等待。这段代码能够成功编译。

让我们尝试在线程池创建每个 worker 时将接收端传递给它们。须知我们希望在 worker 所分配的线程中使用接收者，所以将在闭包中引用 `receiver` 参数。示例 21-17 中展示的代码还不能编译：

文件名：src/lib.rs



```
impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker { id, thread }
    }
}
```

示例 21-17: 将信道的接收端传递给 worker

这是一些简单而直观的修改：将接收端传递进了 `Worker::new`，并接着在闭包中使用它。

如果尝试 check 代码，会得到这个错误：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:26:42
   |
21 |         let (sender, receiver) = mpsc::channel();
   |                      -------- move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
...
25 |         for id in 0..size {
   |         ----------------- inside of this loop
26 |             workers.push(Worker::new(id, receiver));
   |                                      ^^^^^^^^ value moved here, in
previous iteration of loop
```

```
   |
note: consider changing this parameter type in method `new` to borrow instead if
owning the value isn't necessary
  --> src/lib.rs:47:33
   |
47 |     fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
   |        --- in this method       ^^^^^^^^^^^^^^^^^^^^ this parameter takes
ownership of the value
help: consider moving the expression out of the loop so it is only moved once
   |
25 ~         let mut value = Worker::new(id, receiver);
26 ~         for id in 0..size {
27 ~             workers.push(value);
   |

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` (lib) due to 1 previous error
```

这段代码尝试将 `receiver` 传递给多个 `Worker` 实例。这是不行的，回忆第十六章：Rust 所提供的信道实现是多 **生产者**，单 **消费者** 的。这意味着不能简单的克隆信道的消费端来解决问题。我们也不希望将一个消息向多个消费者发送多次；我们希望有一个消息列表和多个 worker 这样每个消息就只会处理一次。

另外，从信道队列中取出任务涉及到修改 `receiver`，所以这些线程需要一个能安全的共享和修改 `receiver` 的方式，否则可能导致竞争状态（参考第十六章）。

回忆一下第十六章讨论的线程安全智能指针，为了在多个线程间共享所有权并允许线程修改其值，需要使用 `Arc<Mutex<T>>`。`Arc` 使得多个 `Worker` 实例拥有接收端，而 `Mutex` 则确保一次只有一个 `Worker` 能从接收端得到任务。示例 21-18 展示了所需的修改：

文件名：src/lib.rs

```
use std::{
    sync::{Arc, Mutex, mpsc},
    thread,
};
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }
```

```
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
    }
}
```

示例 21-18: 使用 `Arc` 和 `Mutex` 在 worker 间共享接收者

在 `ThreadPool::new` 中，将接收端放入 `Arc` 和 `Mutex` 中。对于每一个新 `Worker` `Arc` 来增加引用计数，如此这些 `Worker` 实例就可以共享接收者的所有权了。

通过这些修改，代码可以编译了！我们已经快完成了！

实现 `execute` 方法

最后让我们实现 `ThreadPool` 上的 `execute` 方法。同时也要修改 `Job` 结构体：它将不再是结构体，`Job` 将是一个有着 `execute` 接收到的闭包类型的 trait 对象的类型别名。第二十章 “使用类型别名创建类型同义词” 部分提到过，类型别名允许将长的类型变短。观察示例 21-19：

文件名：src/lib.rs

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

示例 21-19: 为存放每一个闭包的 `Box` 创建一个 `Job` 类型别名，接着在信道中发出任务

在使用 `execute` 得到的闭包新建 `Job` 实例之后，将这些任务从信道的发送端发出。这里调用 `send` 上的 `unwrap`，因为发送可能会失败，这可能发生于例如停止了所有线程执行的情况，这意味着接收端停止接收新消息了。不过目前我们无法停止线程执行；只要线程池存在它们就会一直执行。使用 `unwrap` 是因为我们知道失败不可能发生，不过编译器不知道这些。

不过到此事情还没有结束！在 `Worker` 中，传递给 `thread::spawn` 的闭包仍然还只是 **引用** 了信道的接收端。相反我们需要闭包一直循环，向信道的接收端请求任务，并在得到任务时执行它们。如示例 21-20 对 `Worker::new` 做出修改：

文件名：src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {id} got a job; executing.");

                job();
            }
        });

        Worker { id, thread }
    }
}
```

示例 21-20: 在 worker 线程中接收并执行任务

这里，首先在 `receiver` 上调用了 `lock` 来获取互斥器，接着 `unwrap` 在出现任何错误时 panic。如果互斥器处于一种叫做 **被污染**（*poisoned*）的状态时获取锁可能会失败，这可能发生于其他线程在持有锁时 panic 了且没有释放锁。在这种情况下，调用 `unwrap` 使其 panic 是正确的行为。请随意将 `unwrap` 改为包含有意义错误信息的 `expect`。

如果锁定了互斥器，接着调用 `recv` 从信道中接收 `Job`。最后的 `unwrap` 也绕过了一些错误，这可能发生于持有信道发送端的线程停止的情况，类似于如果接收端关闭时 `send` 方法如何返回 `Err` 一样。

调用 `recv` 会阻塞当前线程，所以如果还没有任务，其会等待直到有可用的任务。`Mutex<T>` 确保一次只有一个 `Worker` 线程尝试请求任务。

现在线程池处于可以运行的状态了！执行 `cargo run` 并发起一些请求：

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
warning: field `workers` is never read
 --> src/lib.rs:7:5
  |
6 | pub struct ThreadPool {
  |            ---------- field in this struct
7 |     workers: Vec<Worker>,
  |     ^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: fields `id` and `thread` are never read
  --> src/lib.rs:48:5
   |
```

```
47 | struct Worker {
   |        ------ fields in this struct
48 |     id: usize,
   |     ^^
49 |     thread: thread::JoinHandle<()>,
   |     ^^^^^^

warning: `hello` (lib) generated 2 warnings
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 4.91s
     Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

成功了！现在我们有了一个可以异步执行连接的线程池！它绝不会创建超过四个线程，所以当服务端收到大量请求时系统也不会负担过重。如果请求 */sleep*，server 也能够通过另外一个线程处理其他请求。

> 注意如果同时在多个浏览器窗口打开 */sleep*，它们可能会彼此间隔地加载 5 秒，因为一些浏览器出于缓存的原因会顺序执行相同请求的多个实例。这些限制并不是由于我们的 web 服务端造成的。

在学习了第十七章和第十八章的 `while let` 循环之后，你可能会好奇为何不能如此编写 worker 线程，如示例 21-21 所示：

文件名：src/lib.rs



```rust
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {id} got a job; executing.");

                job();
            }
        });

        Worker { id, thread }
    }
}
```

示例 21-21: 一个使用 `while let` 的 `Worker::new` 替代实现

这段代码可以编译和运行，但是并不会产生所期望的线程行为：一个慢请求仍然会导致其他请求等待执行。其原因有些微妙：`Mutex` 结构体没有公有 `unlock` 方法，因为锁的所有权依赖 `lock` 方法返回的 `LockResult<MutexGuard<T>>` 中 `MutexGuard<T>` 的生命周期。这允许借用检查器在编译时确保绝不会在没有持有锁的情况下访问由 `Mutex` 守护的资源，不过如果没有认真的思考 `MutexGuard<T>` 的生命周期的话，也可能会导致比预期更久的持有锁。

示例 21-20 中的代码使用的 `let job = receiver.lock().unwrap().recv().unwrap();` 之所以可以工作是因为对于 `let` 来说，当 `let` 语句结束时任何表达式中等号右侧使用的临时值都会立即被丢弃。然而 `while let`（`if let` 和 `match`）直到相关的代码块结束都不会丢弃临时值。在示例 21-21 中，`job()` 调用期间锁一直持续，这也意味着其他的 `Worker` 实例无法接收任务。

优雅停机与清理

示例 21-20 中的代码如期通过使用线程池异步的响应请求。这里有一些警告说 `workers`、`id` 和 `thread` 字段没有直接被使用，这提醒了我们并没有清理所有的内容。当使用不那么优雅的 ctrl-c 终止主线程时，所有其他线程也会立刻停止，即便它们正处于处理请求的过程中。

现在我们要为 `ThreadPool` 实现 `Drop` trait 对线程池中的每一个线程调用 `join`，这样这些线程在关闭前将会执行完它们的请求。接着会为 `ThreadPool` 实现一个告诉线程它们应该停止接收新请求并结束的方式。为了实践这些代码，修改服务端在优雅停机（graceful shutdown）之前只接受两个请求。

在我们开始时需要注意的是：这一切都不会影响处理执行闭包的那部分代码因此如果我们在异步运行时中使用线程池，所有操作也完全相同。

为 `ThreadPool` 实现 `Drop` Trait

现在开始为线程池实现 `Drop`。当线程池被丢弃时，应该 join 所有线程以确保它们完成其操作。示例 21-22 展示了 `Drop` 实现的第一次尝试；这些代码还不能够编译：

文件名：src/lib.rs



```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

示例 21-22: 当线程池离开作用域时 join 每个线程

这里首先遍历线程池中的每个 `workers`。这里使用了 `&mut` 因为 `self` 本身是一个可变引用而且也需要能够修改 `worker`。对于每一个线程，会打印出说明信息表明此特定 `Worker` 实例正在关闭，接着在 `Worker` 实例的线程上调用 `join`。如果 `join` 调用失败，通过 `unwrap` 使得 panic 并进行不优雅的关闭。

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a mutable
reference
  --> src/lib.rs:52:13
   |
52 |             worker.thread.join().unwrap();
   |             ^^^^^^^^^^^^^ ------ `worker.thread` moved due to this method
call
   |             |
   |             move occurs because `worker.thread` has type `JoinHandle<()>`,
which does not implement the `Copy` trait
   |
note: `JoinHandle::<T>::join` takes ownership of the receiver `self`, which moves
`worker.thread`
```

```
  --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/std/src/thread/
mod.rs:1876:17

For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` (lib) due to 1 previous error
```

这里的错误告诉我们并不能调用 `join`，因为我们只有每一个 `worker` 的可变借用，而 `join` 需要获取其参数的所有权。为了解决这个问题，需要一个方法将 `thread` 移动出拥有其所有权的 `Worker` 实例以便 `join` 可以消费这个线程。示例 18-15 中我们曾见过这么做的方法：如果 `Worker` 存放的是 `Option<thread::JoinHandle<()>>`，就可以在 `Option` 上调用 `take` 方法将值从 `Some` 成员中移动出来而对 `None` 成员不做处理。换句话说，正在运行的 `Worker` 的 `thread` 将是 `Some` 成员值，而当需要清理 worker 时，将 `Some` 替换为 `None`，这样 worker 就没有可以运行的线程了。

然而，这种情况**只**会在丢弃 `Worker` 时出现。相应地，我们必须在任何访问 `worker.thread` 时处理 `Option<thread::JoinHandle<()>>`。在惯用的 Rust 代码中 `Option` 用的很多，但当你发现自己总是知道 `Option` 中一定会有值，却还要将其包装在 `Option` 中来应对这一场景时，就应该考虑其他更优雅的方法了。

在这个例子中，存在一个更好的替代方案：`Vec::drain` 方法。它接受一个 range 参数来指定哪些项要从 `Vec` 中移除，并返回一个这些项的迭代器。使用 `..` range 语法会从 `Vec` 中移除**所有**值。

因此我们需要像下面这样更新 `ThreadPool` 的 `drop` 实现：

文件名：src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in self.workers.drain(..) {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

这解决了编译器错误且不需要对我们的代码做其它更改。

向线程发送信号使其停止接收任务

有了所有这些修改，代码就能编译且没有任何警告。然而，坏消息是，这些代码还不能以我们期望的方式运行。问题的关键在于 `Worker` 实例中分配的线程所运行的闭包中的逻辑：此时，调用 `join` 并不会关闭线程，因为它们一直 `loop` 来寻找任务。如果采用这个实现来尝试丢弃 `ThreadPool`，则主线程会永远阻塞在等待第一个线程结束上。

为了修复这个问题，我们将修改 `ThreadPool` 的 `drop` 实现并修改 `Worker` 循环。

首先修改 `ThreadPool` 的 `drop` 实现在等待线程结束前显式地丢弃 `sender`。示例 21-23 展示了 `ThreadPool` 显式丢弃 `sender` 所作的修改。与处理线程时不同，这里**确实**需要使用 `Option`，以便能够使用 `Option::take` 将 `sender` 从 `ThreadPool` 中移出。

文件名：src/lib.rs



```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --snip--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --snip--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in self.workers.drain(..) {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

示例 21-23: 在 join `Worker` 线程之前显式丢弃 `sender`

丢弃 `sender` 会关闭信道，这表明不会有更多的消息被发送。这时 `Worker` 实例中的无限循环中的所有 `recv` 调用都会返回错误。在示例 21-24 中，我们修改 `Worker` 循环在这种情况下优雅地退出，这意味着当 `ThreadPool` 的 `drop` 实现调用 `join` 时线程会结束。

文件名：src/lib.rs

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let message = receiver.lock().unwrap().recv();

                match message {
                    Ok(job) => {
                        println!("Worker {id} got a job; executing.");
```

```
                        job();
                    }
                    Err(_) => {
                        println!("Worker {id} disconnected; shutting down.");
                        break;
                    }
                }
            }
        });

        Worker { id, thread }
    }
}
```

示例 21-24：当 `recv` 返回错误时显式退地出循环

为了实践这些代码，如示例 21-25 所示修改 `main` 在优雅停机服务端之前只接受两个请求：

文件名：src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

示例 21-25: 在处理两个请求之后通过退出循环来停止服务端

你不会希望真实世界的 web 服务端只处理两次请求就停机了，这只是为了展示优雅停机和清理处于正常工作状态。

`take` 方法定义于 `Iterator` trait，这里限制循环最多头 2 次。`ThreadPool` 会在 `main` 的结尾离开作用域， `drop` 实现会运行。

使用 `cargo run` 启动服务端，并发起三个请求。第三个请求应该会失败，而终端的输出应该看起来像这样：

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/hello`
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
```

```
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

可能会出现不同顺序的 `Worker` ID 和信息输出。可以从信息中看到服务是如何运行的：`Worker` 实例 0 和 3 获取了头两个请求。server 会在头第二个请求后停止接受请求，`ThreadPool` 的 `Drop` 实现甚至会在 `Worker` 3 开始工作之前就开始执行。丢弃 `sender` 会断开所有 `Worker` 实例的连接并让它们关闭。每个 `Worker` 实例在断开时会打印出一个信息，接着线程池调用 `join` 来等待每一个 `Worker` 线程结束。

注意在这个特定的运行过程中一个有趣的地方在于：`ThreadPool` 丢弃 `sender`，而在任何 `Worker` 收到消息之前，就尝试 join `Worker` 0 `Worker` 0 还没有从 `recv` 获得一个错误，所以主线程阻塞直到 `Worker` 0 结束。与此同时，`Worker` 3 接收到一个任务接着所有线程会收到一个错误。一旦 `Worker` 0 结束，主线程就等待余下其他 worker 结束。此时它们都退出了循环并停止。

恭喜！现在我们完成了这个项目，也有了一个使用线程池异步响应请求的基础 web 服务端。我们能对服务端执行优雅停机，它会清理线程池中的所有线程。

如下是完整的代码参考：

文件名：src/main.rs

```rust
use hello::ThreadPool;
use std::{
    fs,
    io::{BufReader, prelude::*},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();
```

```
    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

文件名：src/lib.rs

```
use std::{
    sync::{Arc, Mutex, mpsc},
    thread,
};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender: Some(sender),
```

```
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let message = receiver.lock().unwrap().recv();

                match message {
                    Ok(job) => {
                        println!("Worker {id} got a job; executing.");

                        job();
                    }
                    Err(_) => {
                        println!("Worker {id} disconnected; shutting down.");
                        break;
                    }
                }
            }
        });

        Worker {
            id,
            thread: Some(thread),
        }
```

```
    }
}
```

我们还能做得更多！如果你希望继续增强这个项目，如下是一些点子：

- 为 `ThreadPool` 和其公有方法增加更多文档
- 为库的功能增加测试
- 将 `unwrap` 调用改为更健壮的错误处理
- 使用 `ThreadPool` 进行其他不同于处理网络请求的任务
- 在 crates.io 上寻找一个线程池 crate 并使用它实现一个类似的 web 服务端，将其 API 和鲁棒性与我们的实现做对比

总结

好极了！你已经完成了本书的学习！由衷感谢你与我们一道踏上这段 Rust 之旅。现在你已经准备好实现自己的 Rust 项目并帮助他人了。请不要忘记我们的社区，这里有其他 Rustaceans 正乐于帮助你迎接 Rust 之路上的任何挑战。

附录

附录部分包含一些在你的 Rust 之旅中可能用到的参考资料。

附录 A：关键字

下面的列表包含 Rust 中正在使用或者将来会用到的关键字。因此，这些关键字不能被用作标识符（除了 “原始标识符” 部分介绍的原始标识符），这包括函数、变量、参数、结构体字段、模块、crate、常量、宏、静态值、属性、类型、trait 或生命周期 的名字。

目前正在使用的关键字

如下为目前正在使用的关键字及其功能描述的列表。

- `as` - 原始类型转换，消除特定包含项的 trait 的歧义，或者对 `use` 语句中的项重命名
- `async` - 返回一个 `Future` 而不是阻塞当前线程
- `await` - 暂停执行直到 `Future` 的结果就绪
- `break` - 立刻退出循环
- `const` - 定义常量或常量裸指针（constant raw pointer）
- `continue` - 继续进入下一次循环迭代
- `crate` - 在模块路径中，代指 crate root
- `dyn` - 动态分发 trait 对象
- `else` - 作为 `if` 和 `if let` 控制流结构的 fallback
- `enum` - 定义一个枚举
- `extern` - 链接一个外部函数或变量
- `false` - 布尔字面值 `false`
- `fn` - 定义一个函数或 **函数指针类型** (*function pointer type*)
- `for` - 遍历一个迭代器或实现一个 trait 或者指定一个更高级的生命周期
- `if` - 基于条件表达式的结果分支
- `impl` - 实现自有或 trait 功能
- `in` - `for` 循环语法的一部分
- `let` - 绑定一个变量
- `loop` - 无条件循环
- `match` - 模式匹配
- `mod` - 定义一个模块
- `move` - 使闭包获取其所捕获项的所有权
- `mut` - 表示引用、裸指针或模式绑定的可变性
- `pub` - 表示结构体字段、`impl` 块或模块的公有可见性
- `ref` - 通过引用绑定
- `return` - 从函数中返回
- `Self` - 定义或实现 trait 的类型的类型别名
- `self` - 表示方法本身或当前模块
- `static` - 表示全局变量或在整个程序执行期间保持其生命周期
- `struct` - 定义一个结构体
- `super` - 表示当前模块的父模块
- `trait` - 定义一个 trait
- `true` - 布尔字面值 `true`
- `type` - 定义一个类型别名或关联类型
- `union` - 定义一个 union；仅在 union 声明中作为关键字
- `unsafe` - 表示不安全的代码、函数、trait 或实现
- `use` - 将符号引入作用域；为泛型和生命周期约束指定精确捕获

- `where` - 表示一个约束类型的从句
- `while` - 根据表达式的结果进行条件循环

为将来使用保留的关键字

以下关键字目前尚无任何功能，但已被 Rust 保留以备将来使用。

- `abstract`
- `become`
- `box`
- `do`
- `final`
- `gen`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

原始标识符

原始标识符（*Raw identifiers*）是一种允许你使用通常不能使用的关键字的语法。通过在关键字前加上 `r#` 前缀来使用原始标识符。

例如，`match` 是关键字。如果尝试编译如下使用 `match` 作为名字的函数：

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```



会得到这个错误：

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
4 | fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

该错误表示你不能将关键字 `match` 用作函数标识符。要将 `match` 用作函数名称，需要使用原始标识符语法，如下所示：

文件名：src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
```

```
    assert!(r#match("foo", "foobar"));
}
```

此代码编译没有任何错误。注意 `r#` 前缀需同时用于函数名定义和 `main` 函数中的调用。

原始标识符允许使用你选择的任何单词作为标识符，即使该单词恰好是保留关键字。这给予了我们更大的自由来选择名字，这样与其他语言交互式就不用考虑到关键字问题，在要交互的语言中这个名字不是关键字。此外，原始标识符允许你使用以不同于你的 crate 使用的 Rust 版本编写的库。比如，`try` 在 2015 edition 中不是关键字，而在 2018、2021 和 2024 edition 则是。所以如果用 2015 edition 编写的库中带有 `try` 函数，在 2018 edition 中调用时就需要使用原始标识符语法，在这里是 `r#try`。有关版本的更多信息，请参见附录 E。

附录 B：运算符与符号

该附录包含了 Rust 语法的词汇表，包括运算符以及其它符号，这些符号单独出现或出现在路径、泛型、trait bounds、宏、属性、注释、元组以及大括号上下文中。

运算符

表 B-1 包含了 Rust 中的运算符、运算符如何出现在上下文中的示例、简短解释以及该运算符是否可重载。如果一个运算符是可重载的，则该运算符上用于重载的相关 trait 也会列出。

表 B-1: 运算符

运算符	示例	解释	是否可重载
`!`	`ident!(...)`, `ident!{...}`, `ident![...]`	宏展开	
`!`	`!expr`	按位非或逻辑非	`Not`
`!=`	`expr != expr`	不等比较	`PartialEq`
`%`	`expr % expr`	算术取余	`Rem`
`%=`	`var %= expr`	算术取余与赋值	`RemAssign`
`&`	`&expr`, `&mut expr`	借用	
`&`	`&type`, `&mut type`, `&'a type`, `&'a mut type`	借用指针类型	
`&`	`expr & expr`	按位与	`BitAnd`
`&=`	`var &= expr`	按位与及赋值	`BitAndAssign`
`&&`	`expr && expr`	短路（Short-circuiting）逻辑与	
`*`	`expr * expr`	算术乘法	`Mul`
`*=`	`var *= expr`	算术乘法与赋值	`MulAssign`
`*`	`*expr`	解引用	`Deref`
`*`	`*const type`, `*mut type`	裸指针	
`+`	`trait + trait`, `'a + trait`	复合类型限制	
`+`	`expr + expr`	算术加法	`Add`

`+=`	`var += expr`	算术加法与赋值	`AddAssign`
`,`	`expr, expr`	参数以及元素分隔符	
`-`	`- expr`	算术取负	`Neg`
`-`	`expr - expr`	算术减法	`Sub`
`-=`	`var -= expr`	算术减法与赋值	`SubAssign`
`->`	`fn(...) -> type`, |…| -> type	函数与闭包的返回类型	
`.`	`expr.ident`	字段访问	
`.`	`expr.ident(expr, ...)`	方法调用	
`.`	`expr.0`, `expr.1`, etc.	元组索引	
`..`	`..`, `expr..`, `..expr`, `expr..expr`	右开区间范围字面值	`PartialOrd`
`..=`	`..=expr`, `expr..=expr`	右闭区间范围字面值	`PartialOrd`
`..`	`..expr`	结构体更新语法	
`..`	`variant(x, ..)`, `struct_type { x, .. }`	“与剩余部分” 的模式绑定	
`...`	`expr...expr`	（Deprecated，请使用 `..=`） 在模式中：闭区间范围模式	
`/`	`expr / expr`	算术除法	`Div`
`/=`	`var /= expr`	算术除法与赋值	`DivAssign`
`:`	`pat: type`, `ident: type`	约束	
`:`	`ident: expr`	结构体字段初始化	
`:`	`'a: loop {...}`	循环标志	
`;`	`expr;`	语句和语句结束符	
`;`	`[...; len]`	固定大小数组语法的部分	
`<<`	`expr << expr`	左移	`Shl`

`<<=`	`var <<= expr`	左移与赋值	`ShlAssign`
`<`	`expr < expr`	小于比较	`PartialOrd`
`<=`	`expr <= expr`	小于等于比较	`PartialOrd`
`=`	`var = expr`, `ident = type`	赋值/等值	
`==`	`expr == expr`	等于比较	`PartialEq`
`=>`	`pat => expr`	匹配分支语法的部分	
`>`	`expr > expr`	大于比较	`PartialOrd`
`>=`	`expr >= expr`	大于或等于比较	`PartialOrd`
`>>`	`expr >> expr`	右移	`Shr`
`>>=`	`var >>= expr`	右移与赋值	`ShrAssign`
`@`	`ident @ pat`	模式绑定	
`^`	`expr ^ expr`	按位异或	`BitXor`
`^=`	`var ^= expr`	按位异或与赋值	`BitXorAssign`
|	pat | pat	模式替代	
|	expr | expr	按位或	`BitOr`
|=	var |= expr	按位或与赋值	`BitOrAssign`
||	expr || expr	短路（Short-circuiting）逻辑或	
`?`	`expr?`	错误传播	

非运算符符号

下面的列表中包含了所有和运算符不一样功能的符号；也就是说，它们不表现为函数或方法调用。

表 B-2 展示了以其自身出现以及出现在合法其他各个地方的符号。

表 B-2：独立语法

符号	**解释**
`'ident`	命名生命周期或循环标签

`...u8`, `...i32`, `...f64`, `...usize` 等	指定类型的数值常量
`"..."`	字符串字面值
`r"..."`, `r#"..."#`, `r##"..."##`, etc.	原始字符串字面值，未处理的转义字符
`b"..."`	字节字符串字面值; 构造一个字节数组类型而非字符串
`br"..."`, `br#"..."#`, `br##"..."##` 等	原始字节字符串字面值，原始和字节字符串字面值的结合
`'...'`	字符字面值
`b'...'`	ASCII 码字节字面值
|...| expr	闭包
`!`	离散函数的总是为空的类型
`_`	“忽略” 模式绑定；也用于增强整型字面值的可读性

表 B-3 展示了出现在从模块结构到项的路径上下文中的符号

表 B-3：路径相关语法

符号	**解释**
`ident::ident`	命名空间路径
`::path`	与 extern prelude 相对的路径，其他所有 crate 都以此为根（即一个包含 crate 名称的显式绝对路径）
`self::path`	与当前模块相对的路径（即一个显式相对路径）
`super::path`	与当前模块的父模块相对的路径
`type::ident`, `<type as trait>::ident`	关联常量、函数以及类型
`<type>::...`	不可以被直接命名的关联项类型（如 `<&T>::...`，`<[T]>::...`，等）
`trait::method(...)`	通过命名定义的 trait 来消除方法调用的二义性
`type::method(...)`	通过命名定义的类型来消除方法调用的二义性

`<type as trait>::method(...)`	通过命名 trait 和类型来消除方法调用的二义性

表 B-4 展示了用于泛型类型参数上下文中的符号。

表 B-4：泛型

符号	解释
`path<...>`	为一个类型中的泛型指定具体参数（如 `Vec<u8>`）
`path::<...>`, `method::<...>`	为一个泛型、函数或表达式中的方法指定具体参数，通常指 turbofish（如 `"42".parse::<i32>()`）
`fn ident<...> ...`	泛型函数定义
`struct ident<...> ...`	泛型结构体定义
`enum ident<...> ...`	泛型枚举定义
`impl<...> ...`	定义泛型实现
`for<...> type`	高级生命周期限制
`type<ident=type>`	泛型，其一个或多个相关类型必须被指定为特定类型（如 `Iterator<Item=T>`）

表 B-5 展示了出现在使用 trait bound 约束泛型参数上下文中的符号。

表 B-5: Trait Bound 约束

符号	解释
`T: U`	泛型参数 `T` 约束于实现了 `U` 的类型
`T: 'a`	泛型 `T` 的生命周期必须长于 `'a`（意味着该类型不能传递包含生命周期短于 `'a` 的任何引用）
`T: 'static`	泛型 T 不包含除 `'static` 之外的借用引用
`'b: 'a`	泛型 `'b` 生命周期必须长于泛型 `'a`
`T: ?Sized`	使用一个不定大小的泛型类型
`'a + trait`, `trait + trait`	复合类型限制

表 B-6 展示了在调用或定义宏以及在项上指定属性的上下文中出现的符号。

表 B-6: 宏与属性

符号	解释
`#[meta]`	外部属性
`#![meta]`	内部属性
`$ident`	宏替换
`$ident:kind`	宏捕获
`$(…)…`	宏重复
`ident!(...)`, `ident!{...}`, `ident![...]`	宏调用

表 B-7 展示了创建注释的符号。

表 B-7: 注释

符号	注释
`//`	行注释
`//!`	内部行文档注释
`///`	外部行文档注释
`/*...*/`	块注释
`/*!...*/`	内部块文档注释
`/**...*/`	外部块文档注释

表 B-8 展示了出现在使用圆括号上下文中的符号。

表 B-8: 圆括号

符号	解释
`()`	空元组（亦称单元），即是字面值也是类型
`(expr)`	括号表达式
`(expr,)`	单一元素元组表达式
`(type,)`	单一元素元组类型
`(expr, ...)`	元组表达式

`(type, ...)`	元组类型
`expr(expr, ...)`	函数调用表达式；也用于初始化元组结构体 `struct` 以及元组枚举 `enum` 变体

表 B-9 展示了使用大括号的上下文。

表 B-9: 大括号

符号	**解释**
`{...}`	块表达式
`Type {...}`	`struct` 字面值

表 B-10 展示了使用方括号的上下文。

表 B-10: 方括号

符号	**解释**
`[...]`	数组字面值
`[expr; len]`	复制了 `len` 个 `expr` 的数组
`[type; len]`	包含 `len` 个 `type` 类型的数组
`expr[expr]`	集合索引。重载（`Index`, `IndexMut`）
`expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`	集合索引，使用 `Range`，`RangeFrom`，`RangeTo` 或 `RangeFull` 作为索引来代替集合 slice

附录 C：可派生的 trait

在本书的各个部分中，我们讨论了可应用于结构体和枚举定义的 `derive` 属性。`derive` 属性会在使用 `derive` 语法标记的类型上生成对应 trait 的默认实现的代码。

在本附录中提供了标准库中所有可以使用 `derive` 的 trait 的参考。这些部分涉及到：

- 该 trait 将会派生什么样的操作符和方法
- 由 `derive` 提供什么样的 trait 实现
- 实现该 trait 对类型意味着什么
- 在何种条件下允许或不允许实现该 trait
- 需要 trait 操作的例子

如果你希望不同于 `derive` 属性所提供的行为，请查阅标准库文档 中每个 trait 的细节以了解如何手动实现它们。

这里列出的 trait 是仅有的在标准库中定义且能通过 `derive` 在类型上实现。标准库中定义的其它 trait 不能通过 `derive` 在类型上实现。这些 trait 不存在有意义的默认行为，所以由你负责以合理的方式实现它们。

一个无法被派生的 trait 的例子是为终端用户处理格式化的 `Display` 。你应该时常考虑使用合适的方法来为终端用户显示一个类型。终端用户应该看到类型的什么部分？他们会找出相关部分吗？对他们来说最相关的数据格式是什么样的？Rust 编译器没有这样的洞察力，因此无法为你提供合适的默认行为。

本附录所提供的可派生 trait 列表并不全面：库可以为其自己的 trait 实现 `derive`，可以使用 `derive` 的 trait 列表事实上是无限的。实现 `derive` 涉及到过程宏的应用，这在第二十章的 “宏” 一节中有介绍。

用于程序员输出的 `Debug`

`Debug` trait 用于开启格式化字符串中的调试格式，其通过在 `{}` 占位符中增加 `:?` 表明。

`Debug` trait 允许以调试目的来打印一个类型的实例，所以使用该类型的程序员可以在程序执行的特定时间点观察其实例。

例如，在使用 `assert_eq!` 宏时，`Debug` trait 是必需的。如果等式断言失败，这个宏就把给定实例的值作为参数打印出来，如此程序员可以看到两个实例为什么不相等。

等值比较的 `PartialEq` 和 `Eq`

`PartialEq` trait 可以比较某个类型的实例以检查是否相等，并开启了 `==` 和 `!=` 运算符的功能。

派生的 `PartialEq` 实现了 `eq` 方法。当 `PartialEq` 在结构体上派生时，只有**所有**的字段都相等时两个实例才相等，同时只要有任何字段不相等则两个实例就不相等。当在枚举上派生时，每一个变体都和其自身相等，且和其它变体都不相等。

例如，当使用 `assert_eq!` 宏时，需要比较一个类型的两个实例是否相等，则 `PartialEq` trait 是必须的。

`Eq` trait 没有方法。其作用是表明每一个被标记类型的值等于其自身。`Eq` trait 只能应用于那些实现了 `PartialEq` 的类型，但并非所有实现了 `PartialEq` 的类型都可以实现 `Eq`。浮点类型就是一个例子：浮点数的实现表明两个非数字（`NaN`，not-a-number）值是互不相等的。

例如，对于一个 `HashMap<K, V>` 中的键（key）来说，`Eq` 是必须的，这样 `HashMap<K, V>` 就可以知道两个键是否相等了。

排序比较的 `PartialOrd` 和 `Ord`

`PartialOrd` trait 可以基于排序的目的而比较一个类型的实例。实现了 `PartialOrd` 的类型可以使用 `<`、 `>`、`<=` 和 `>=` 操作符。但只能在同时实现了 `PartialEq` 的类型上使用 `PartialOrd`。

派生 `PartialOrd` 实现了 `partial_cmp` 方法，其返回一个 `Option<Ordering>`，但当给定值无法产生顺序时将返回 `None`。尽管大多数类型的值都可以比较，但一个无法产生顺序的例子是：浮点类型的非数字值（not-a-number，`NaN`）。对任何浮点数与 `NaN` 调用 `partial_cmp` 都会返回 `None`。

当在结构体上派生时，`PartialOrd` 按照结构体定义中字段出现的顺序，依次比较每个字段的值，以此来比较两个实例。当在枚举上派生时，认为在枚举定义中声明较早的枚举变体小于其后的变体。

例如，对于来自于 `rand` crate 中的 `gen_range` 方法来说，当在一个范围表达式指定的范围内生成一个随机值时，`PartialOrd` trait 是必须的。

`Ord` trait 也让你知道在一个带注解类型上的任意两个值存在有效顺序。`Ord` trait 实现了 `cmp` 方法，它返回一个 `Ordering` 而不是 `Option<Ordering>`，因为总存在一个合法的顺序。只可以在实现了 `PartialOrd` 和 `Eq`（`Eq` 依赖 `PartialEq`）的类型上使用 `Ord` trait。当在结构体或枚举上派生时，`cmp` 的行为与 `PartialOrd` 派生实现的 `partial_cmp` 相同。

例如，将值存储到 `BTreeSet<T>` 中时，需要 `Ord` trait，因为该数据结构基于值的排序顺序来存储数据。

复制值的 `Clone` 和 `Copy`

`Clone` trait 可以明确地创建一个值的深拷贝（deep copy），复制过程可能包含任意代码的执行以及堆上数据的复制。查阅第四章 “使用克隆的变量与数据交互” 以获取有关 `Clone` 的更多信息。

派生 `Clone` 实现了 `clone` 方法，当其为整个类型实现时，会在类型的每一部分上调用 `clone` 方法。这意味着类型中所有字段或值也必须实现了 `Clone`，这样才能够派生 `Clone` 。

例如，当在一个 slice 上调用 `to_vec` 方法时，`Clone` 是必须的。slice 并不拥有其包含的实例，但是从 `to_vec` 中返回的 vector 需要拥有它们的实例，因此 `to_vec` 在每个元素上调用 `clone`。所以存储在切片中的类型必须实现 `Clone`。

`Copy` trait 允许你通过只拷贝存储在栈上的位来复制值；无需执行额外的代码。查阅第四章 “只在栈上的数据：拷贝” 的部分来获取有关 `Copy` 的更多信息。

`Copy` trait 并未定义任何方法来阻止编程人员重写这些方法或违反无需执行额外代码的假设。这样，所有程序员都可以假定复制值会非常快速。

可以在类型内部全部实现 `Copy` trait 的任意类型上派生 `Copy`。一个实现了 `Copy` 的类型必须也实现了 `Clone`，因为一个实现了 `Copy` 的类型也简单地实现了 `Clone`，其执行和 `Copy` 相同的任务。

`Copy` trait 很少是必需的；实现 `Copy` 的类型是有优化的，这意味着你无需调用 `clone`，这让代码更简洁。

任何使用 `Copy` 的代码都可以通过 `Clone` 实现，但代码可能会稍慢，或者不得不在代码中的许多位置上使用 `clone`。

固定大小的值到值映射的 `Hash`

`Hash` trait 可以实例化一个任意大小的类型，并且能够用哈希（hash）函数将该实例映射到一个固定大小的值上。派生 `Hash` 实现了 `hash` 方法。`hash` 方法的派生实现结合了在类型的每部分调用 `hash` 的结果，这意味着所有的字段或值也必须实现了 `Hash`，这样才能够派生 `Hash`。

例如，在 `HashMap<K, V>` 上存储数据，存放 key 的时候，`Hash` 是必须的。

一个 `Hash` 是必须的例子是在 `HashMap<K, V>` 中存储键来高效地存储数据。

默认值的 `Default`

`Default` trait 使你创建一个类型的默认值。派生 `Default` 实现了 `default` 函数。`default` 函数的派生实现调用了类型每部分的 `default` 函数，这意味着类型中所有的字段或值也必须实现了 `Default`，这样才能够派生 `Default` 。

`Default::default` 函数通常结合结构体更新语法一起使用，这在第五章的 “使用结构体更新语法从其他实例中创建实例” 部分有讨论。可以自定义一个结构体的一小部分字段而剩余字段则使用 `..Default::default()` 来设置默认值。

例如，当你在 `Option<T>` 实例上使用 `unwrap_or_default` 方法时，`Default` trait 是必须的。如果 `Option<T>` 是 `None` 的话，`unwrap_or_default` 方法将返回存储在 `Option<T>` 中 `T` 类型的 `Default::default` 的结果。

附录 D：实用开发工具

在本附录中，我们将讨论 Rust 项目提供的一些有助于开发 Rust 代码的工具。我们将介绍自动格式化、快速应用警告修复、linter 以及与 IDE 的集成。

通过 `rustfmt` 自动格式化

`rustfmt` 工具根据社区代码风格格式化代码。很多项目使用 `rustfmt` 来避免编写 Rust 代码风格的争论：所有人都用这个工具格式化代码！

安装 `rustfmt`：

Rust 安装默认已包含 rustfmt，因此你的系统上应该已经有 `rustfmt` 和 `cargo-fmt` 程序了。这两个命令类似于 `rustc` 和 `cargo`，其中 `rustfmt` 提供了更细粒度的控制，而 `cargo-fmt` 则理解使用 Cargo 的项目约定。要格式化任何 Cargo 项目，请输入以下命令：

```
$ cargo fmt
```

运行此命令会格式化当前 crate 中所有的 Rust 代码。这应该只会改变代码风格，而不是代码语义。

该命令会为你提供 `rustfmt` 和 `cargo-fmt`，类似于 Rust 同时提供 `rustc` 和 `cargo`。要格式化任何 Cargo 项目，请执行以下命令：

```
$ cargo fmt
```

运行此命令会格式化当前 crate 中所有的 Rust 代码。这应该只会改变代码风格，而不是代码语义。有关 `rustfmt` 的更多信息，请参阅 该文档。

通过 `rustfix` 修复代码

`rustfix` 工具已随 Rust 安装一并提供，可以自动修复那些具有明确修复方式的编译器警告，这通常正是你所需要的。你可能已经见过类似的编译器警告。例如，考虑如下代码：

文件名：src/main.rs

```
fn main() {
    let mut x = 42;
    println!("{x}");
}
```

这里定义变量 `x` 为可变，但是我们从未修改它。Rust 会警告说：

这里调用了 `do_something` 函数 100 次，不过从未在 `for` 循环体中使用变量 `i`。Rust 会警告说：

```
$ cargo build
   Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: variable does not need to be mutable
 --> src/main.rs:2:9
  |
2 |     let mut x = 0;
```

```
  |         ----^
  |         |
  |         help: remove this `mut`
  |
  = note: `#[warn(unused_mut)]` on by default
```

警告中建议移除 `mut` 关键字。我们可以通过运行 `cargo fix` 命令使用 `rustfix` 工具来自动采用该建议：

```
$ cargo fix
    Checking myprogram v0.1.0 (file:///projects/myprogram)
      Fixing src/main.rs (1 fix)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

如果再次查看 *src/main.rs*，会发现 `cargo fix` 修改了代码：

文件名：src/main.rs

```
fn main() {
    let x = 42;
    println!("{x}");
}
```

变量 `x` 现在是不可变的了，警告也不再出现。

`cargo fix` 命令可以用于在不同 Rust 版本间迁移代码。版本在附录 E 中介绍。

使用 Clippy 获取更多 lint

Clippy 工具是一组 lints 的集合，用于分析你的代码，帮助你捕捉常见错误并改进 Rust 代码。Clippy 已包含在 Rust 的标准安装中。

要对任何 Cargo 项目运行 Clippy 的 lint，请输入以下命令：

```
$ cargo clippy
```

例如，你编写了一个程序使用了数学常数，例如 pi，的一个近似值，如下所示，：

文件名：src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

在此项目上运行 `cargo clippy` 会导致这个错误：

```
error: approximate value of `f{32, 64}::consts::PI` found
 --> src/main.rs:2:13
  |
2 |     let x = 3.1415;
```

```
  |             ^^^^^^
  |
  = note: `#[deny(clippy::approx_constant)]` on by default
  = help: consider using the constant directly
  = help: for further information visit https://rust-lang.github.io/rust-clippy/
master/index.html#approx_constant
```

该错误提示你 Rust 已经定义了一个更精确的 `PI` 常量，如果使用该常量，你的程序会更为正确。你可以将代码改为使用 `PI` 常量。如下代码就不会引发 Clippy 的任何错误或警告：

文件名：src/main.rs

```
fn main() {
    let x = std::f64::consts::PI;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

有关 Clippy 的更多信息，请参阅 其文档。

使用 `rust-analyzer` 的 IDE 集成

为了帮助 IDE 集成，Rust 社区建议使用 `rust-analyzer`。这个工具是一组以编译器为中心的实用程序，它实现了 Language Server Protocol，一个 IDE 与编程语言之间的通信规范。`rust-analyzer` 可以用于不同的客户端，比如 Visual Studio Code 的 Rust analyzer 插件。

访问 `rust-analyzer` 项目的主页来了解如何安装它，然后为你的 IDE 安装 language server 支持。如此你的 IDE 便会获得如自动补全、跳转到定义和 inline error 之类的功能。

附录 E：版本

早在第一章，我们见过 `cargo new` 在 *Cargo.toml* 中增加了一些有关 `edition` 的元数据。本附录将解释其意义！

Rust 语言和编译器有一个为期六周的发布循环，这意味着用户会稳定得到新功能的更新。其他编程语言发布大更新但不甚频繁；Rust 选择更为频繁的发布小更新。一段时间之后，所有这些小更新会日积月累。不过随着小更新逐次的发布，或许很难回过头来感叹：“哇，从 Rust 1.10 到 Rust 1.31，Rust 的变化真大！”

大约每两到三年，Rust 团队会生成一个新的 Rust **版本**（*edition*）。每一个版本会结合已经落地的功能，并提供一个清晰的带有完整更新文档和工具的功能包。新版本会作为常规的六周发布过程的一部分发布。

新的版本对不同人群具有不同意义：

- 对于活跃的 Rust 用户，新版本将这些增量改进整合成一个易于理解的包。
- 对于非 Rust 用户，它表明发布了一些重大进展，这意味着 Rust 可能变得值得一试。
- 对于 Rust 自身开发者，它提供了项目整体的集合点。

在本文档编写时，Rust 有四个可用版本：Rust 2015、Rust 2018、Rust 2021 和 Rust 2024。本书基于 Rust 2024 edition 惯用法编写。

Cargo.toml 中的 `edition` 字段表明代码应该使用哪个版本编译。如果该字段不存在，其默认为 `2015` 以提供后向兼容性。

每个项目都可以选择不同于默认的 2015 edition 的版本。这样，版本可能会包含不兼容的修改，比如新增关键字可能会与代码中的标识符冲突并导致错误。不过除非选择兼容这些修改，（旧）代码仍将能够编译，即便升级了 Rust 编译器的版本。

所有 Rust 编译器都支持任何之前存在的编译器版本，并可以链接任何支持版本的 crate。编译器修改只影响最初的解析代码的过程。因此，如果你使用 Rust 2015 而某个依赖使用 Rust 2018，你的项目仍旧能够编译并使用该依赖。反之，若项目使用 Rust 2018 而依赖使用 Rust 2015 亦可工作。

有一点需要明确：大部分功能在所有版本中都能使用。开发者使用任何 Rust 版本将能继续接收最新稳定版的改进。然而在一些情况，主要是增加了新关键字的时候，则可能出现了只能用于新版本的功能。只需切换版本即可利用新版本的功能。

请查看 *Edition Guide* 了解更多细节，这是一个全面介绍不同版本之间差异的书籍，包括如何通过 `cargo fix` 自动将代码迁移到新版本。

附录 F：本书译本

一些非英语语言的资源。多数仍在翻译中；请查阅翻译标签来帮助翻译，或者添加译本链接！

- Português (BR)
- Português (PT)
- 简体中文：KaiserY/trpl-zh-cn, gnu4cn/rust-lang-Zh_CN
- 正體中文
- Українська
- Español, alternate, Español por RustLangES
- Русский
- 한국어
- 日本語
- Français
- Polski
- Cebuano
- Tagalog
- Esperanto
- ελληνική
- Svenska
- Farsi, Persian (FA)
- Deutsch
- हिंदी
- ไทย
- Danske

附录 G：Rust 是如何开发的与 “Nightly Rust”

本附录介绍 Rust 是如何开发的以及这对你作为 Rust 开发者的影响。

无停滞稳定

作为一门语言，Rust **十分**注重代码的稳定性。我们希望 Rust 成为你可依赖的坚实基础，假如事务持续地在变化，这个希望就实现不了。但与此同时，如果不能实验新功能的话，在发布之前我们又无法发现其中重大的缺陷，而一旦发布便再也没有修改的机会了。

对于这个问题我们的解决方案被称为 “无停滞稳定”（“stability without stagnation”），其指导性原则是：无需担心升级到最新的稳定版 Rust。每次升级应该是无痛的，并应带来新功能，更少的 bug 和更快的编译速度。

Choo, Choo! 发布通道和发布时刻表（Riding the Trains）

Rust 开发运行于一个**发布时刻表**（*train schedule*）之上。也就是说，所有的开发工作都位于 Rust 仓库的 `master` 分支。发布采用 software release train 模型，其被用于思科 IOS 等其它软件项目。Rust 有三个**发布通道**（*release channel*）：

- Nightly
- Beta
- Stable（稳定版）

大部分 Rust 开发者主要采用稳定版通道，不过希望实验新功能的开发者可能会使用 nightly 或 beta 版。

如下是一个开发和发布过程如何运转的例子：假设 Rust 团队正在进行 Rust 1.5 的发布工作。该版本发布于 2015 年 12 月，不过这里只是为了提供一个真实的版本。Rust 新增了一项功能：一个 `master` 分支的新提交。每天晚上，会产生一个新的 nightly 版本。每天都是发布版本的日子，而这些发布由发布基础设施自动完成。所以随着时间推移，发布轨迹看起来像这样，版本每晚一发：

```
nightly: * - - * - - *
```

每六周时间，是准备发布新版本的时候了！Rust 仓库的 `beta` 分支会从用于 nightly 的 `master` 分支产生。现在，有了两个发布渠道：

```
nightly: * - - * - - *
                     |
beta:                *
```

大部分 Rust 用户不会主要使用 beta 版本，不过在 CI 系统中对 beta 版本进行测试能够帮助 Rust 发现可能的回归缺陷（regression）。同时，每晚仍产生 nightly 发布：

```
nightly: * - - * - - * - - * - - *
                     |
beta:                *
```

比如我们发现了一个回归缺陷。好消息是在这些回归缺陷流入稳定发布之前还有一些时间来测试 beta 版本！fix 被合并到 `master`，为此 nightly 版本得到了修复，接着这些 fix 将 backport 到 `beta` 分支，一个新的 beta 发布就产生了：

```
nightly: * - - * - - * - - * - - * - - *
                  |
beta:             * - - - - - - - - - *
```

第一个 beta 版的六周后，是发布稳定版的时候了！`stable` 分支从 `beta` 分支生成：

```
nightly: * - - * - - * - - * - - * - - * - * - *
                  |
beta:             * - - - - - - - - - *
                                      |
stable:                               *
```

好的！Rust 1.5 发布了！然而，我们忘了些东西：因为又过了六周，我们还需发布**下一个** Rust 的 beta 版，Rust 1.6。所以从 `beta` 生成 `stable` 分支后，新版的 `beta` 分支也再次从 `nightly` 生成：

```
nightly: * - - * - - * - - * - - * - - * - * - *
                  |                           |
beta:             * - - - - - - - - - *       *
                                      |
stable:                               *
```

这被称为 “train model”，因为每六周，一个版本 “离开车站”（“leaves the station”），不过从 beta 通道到达稳定通道还需历经一段旅程。

Rust 每六周发布一个版本，如时钟般准确。如果你知道了某个 Rust 版本的发布时间，就可以知道下个版本的时间：六周后。每六周发布版本的一个好的方面是下一班车会来得更快。如果特定版本碰巧缺失某个功能也无需担心：另一个版本很快就会到来！这有助于减少因临近发版时间而偷偷释出未经完善的功能的压力。

多亏了这个过程，你总是可以切换到下一版本的 Rust 并验证是否可以轻易的升级：如果 beta 版不能如期工作，你可以向 Rust 团队报告并在发布稳定版之前得到修复！beta 版造成的破坏是非常少见的，不过 `rustc` 也不过是一个软件，难免会有 bug。

维护时间

Rust 项目仅对最近的稳定版本提供支持。当发布新稳定版本时，旧版本即达到生命周期终止（EOL, end of life），这意味着每个版本的支持期为六周。

不稳定功能

这个发布模型中另一个值得注意的地方：不稳定功能（unstable features）。Rust 使用一个被称为 “功能标记”（“feature flags”）的技术来确定给定版本的某个功能是否启用。如果新功能正在积极地开发中，其提交到了 `master`，因此会出现在 nightly 版中，不过会位于一个 **功能标记** 之后。作为用户，如果你希望尝试这个正在开发的功能，必须使用 nightly 版并在源码中使用合适的标记来开启。

如果使用的是 beta 或稳定版 Rust，则不能使用任何功能标记。这是在新功能被宣布为永久稳定之前让大家提前实际使用它们的关键。这既满足了希望使用最尖端技术的同学，那些坚持稳定版的同学也知道其代码不会被破坏。这就是无停滞稳定。

本书只包含稳定的功能，因为还在开发中的功能仍可能改变，当其进入稳定版时肯定会与编写本书的时候有所不同。你可以在网上获取只存在 nightly 版中功能的文档。

Rustup 和 Rust Nightly 的职责

Rustup 使得改变不同发布通道的 Rust 更为简单，其在全局或分项目的层次工作。其默认会安装稳定版 Rust。例如，为了安装 nightly：

```
$ rustup toolchain install nightly
```

你会发现 `rustup` 也安装了所有的**工具链**（*toolchains*，Rust 和其相关组件）。如下是一位作者的 Windows 计算机上的例子：

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

如你所见，默认是稳定版。大部分 Rust 用户在大部分时间使用稳定版。你可能也会这么做，不过如果你关心最新的功能，可以为特定项目使用 nightly 版。为此，可以在项目目录使用 `rustup override` 来设置当前目录 `rustup` 使用 nightly 工具链：

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

现在，每次在 */projects/needs-nightly* 中调用 `rustc` 或 `cargo`，`rustup` 会确保使用 nightly 版 Rust 而非默认的稳定版。在你有很多 Rust 项目时大有裨益！

RFC 流程和团队

那么你如何了解这些新功能呢？Rust 开发模式遵循一个 **Request For Comments (RFC) 流程**。如果你希望改进 Rust，可以编写一个提案，也就是 RFC。

任何人都可以编写 RFC 来改进 Rust，同时这些 RFC 会被 Rust 团队评审和讨论，他们由很多不同分工的子团队组成。这里是 Rust 官网上 所有团队的总列表，其包含了项目中每个领域的团队：语言设计、编译器实现、基础设施、文档等。各个团队会阅读相应的提议和评论，发表自己的意见，并最终达成接受或回绝功能的一致。

如果功能被接受了，在 Rust 仓库会打开一个 issue，人们就可以实现它。实现功能的人当然可能不是最初提议功能的人！当实现完成后，其会合并到 `master` 分支并位于一个功能开关（feature gate）之后，正如 “不稳定功能” 部分所讨论的。

在稍后的某个时间，一旦使用 nightly 版的 Rust 团队能够尝试这个功能了，团队成员会讨论这个功能，它如何在 nightly 中工作，并决定是否应该进入稳定版。如果决定继续推进，功能开关会移除，然后这个功能就被认为是稳定的了！乘着“发布列车”，最终在新的稳定版 Rust 中出现。