

THE RUST
PROGRAMMING LANGUAGE

Rust权威指南

[美] Steve Klabnik Carol Nichols 著 毛靖凯 唐刚 沙渺 译

涵盖
Rust
2018



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Rust权威指南

THE RUST PROGRAMMING LANGUAGE

[美] Steve Klabnik Carol Nichols 著

毛靖凯 唐刚 沙渺 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书由Rust核心团队成员编写而成，由浅入深地探讨了Rust语言的方方面面。从创建函数、选择数据类型及绑定变量等基础内容着手，逐步介绍所有权、生命周期、trait、安全保证等高级概念，错误处理、模式匹配、包管理、并发机制、函数式特性等实用工具，以及完整的项目开发实战案例。

作为开源的系统级编程语言，Rust可以帮助你编写出更有效率且更加可靠的软件，在给予开发者底层控制能力的同时，通过高水准的工程设计避免了传统语言带来的诸多麻烦。

本书适合所有希望评估、入门、提高和研究Rust语言的软件开发人员阅读。

Copyright © 2019 by Mozilla Corporation and the Rust Project Developers. Title of English-language original: The Rust Programming Language, ISBN 978-1-71850-044-0, published by No Starch Press. Simplified Chinese-language edition copyright ©2020 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由No Starch Press授予电子工业出版社。

专有出版权受法律保护。

版权贸易合同登记号 图字: 01-2019-4420

图书在版编目 (CIP) 数据

Rust权威指南 | (美) 史蒂夫· 克拉伯尼克 (Steve Klabnik) ,
(美) 卡罗尔· 尼科尔斯 (Carol Nichols) 著; 毛靖凯, 唐刚, 沙渺
译. —北京: 电子工业出版社, 2020. 6

书名原文: The Rust Programming Language

ISBN 978-7-121-38706-7

I . ①R… II . ①史… ②卡… ③毛… ④唐… ⑤沙… III. ①程
序语言-程序设计-指南 IV. ①TP312-62

中国版本图书馆CIP数据核字 (2020) 第039475号

责任编辑: 刘恩惠

印刷:

装订:

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编： 100036

开本： 787×980 1| 16 印张： 44.75 字数： 716千字

版次： 2020年6月第1版

印次： 2020年6月第1次印刷

定价： 159.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。
若书店售缺，请与本社发行部联系，联系及邮购电话：（010）
88254888，88258888。

质量投诉请发邮件至z1ts@phei.com.cn，盗版侵权举报请发邮件
至dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

译者序

作为系统级语言事实上的标杆，C/C++语言诞生至今已经四十余年了。四十年历史的积累从某种角度上讲亦是四十年的负担。为了开发出运行正确的软件，我们需要投入数年的时间来学会如何避免臭名昭著的漏洞，但即便是最为谨慎的开发者，也无法保证自己的程序万无一失。这些漏洞不仅会导致计算机崩溃，还会带来许多意想不到的安全性问题。特别是随着互联网技术的飞速发展，所有人的私密信息都有可能因为这类安全性问题而赤裸裸地暴露在陌生人的面前。

有些语言，比如C#等，试图使用庞大的运行时系统来解决这一问题，其中最常见的解决方案便是垃圾回收（Garbage Collection）机制。这种机制在保证了内存安全的同时，却在某种程度上剥夺了程序员对底层的控制能力，并往往伴随着性能上的额外损耗。

正是在这样的背景之下，Rust应运而生。

Rust站在了前人的肩膀上，借助于最近几十年的语言研究成果，创造出了所有权与生命周期等崭新的概念。相对于C/C++等传统语言，它具有天生的安全性；换句话说，你无法在安全的Rust代码中执行任何非法的内存操作。相对于C#等带有垃圾回收机制的语言来讲，它遵循了零开销抽象（Zero-Cost Abstraction）规则，并为开发者保留了最大的底层控制能力。

Rust从设计伊始便致力于提供高水准的人体工程学体验。你可以在Rust中看到代数数据类型、卫生宏、迭代器等饱经证明的优秀语言设计，这些刻意的设计能够帮助你自然而然地编写出高效且安全的代

码。在语言本身之外，Rust核心开发团队还规划并实现了一系列顶尖的工具链——从集成的包管理器到带有依赖管理的构建工具，再到跨越编辑器的自动补全、类型推导及自动格式化等服务工具。

Rust由开源基金会Mozilla推动开发，它的背后有一个完善且热情的社区。年轻的Rust正在众人合力之下不断进步，许许多多像你我一样的开发者共同决定着Rust的前进方向。你能够在Rust的托管网站GitHub上追踪到最新的源代码及开发进展，甚至是参与到Rust本身的开发之中。

但不得不承认的是，Rust独特的创新性也给我们带来了突兀的学习曲线。这些概念与传统语言雕刻在我们脑海中的回路是如此的不同，以至于使众多的初学者望而却步。这让人无比遗憾。为了解决这个问题，Rust核心团队的Steve Klabnik和Carol Nichols共同撰写了本书。他们由浅入深地介绍了Rust语言的方方面面——从基本的通用概念开始，到模式匹配、函数式特性、并发机制等实用工具，再到所有权、生命周期等特有概念。除此之外，本书还穿插了众多的代码片段及3个完整的项目开发实践案例。我们相信本书能够帮助所有期望评估、入门、提高及研究Rust语言的软件开发人员。

最后，我们非常高兴能够参与此次的翻译工作。在长久以来的学习过程中，社区内热情的Rust爱好者们提供了许多无法言尽的帮助，而这次的工作则给予了我们回馈社区的机会。感谢电子工业出版社牵头引进了这样一本官方图书，感谢编辑刘恩惠在翻译过程中的包容和理解，并在后期进行了大量的编辑工作。没有他们，就没有本书最终的完成。

碍于能力有限，对于本书中可能出现的错误，还望读者海涵；我们会随着Rust的迭代升级，不断地对本书进行更新与勘误。

序

虽然不是那么明显，但Rust编程语言的核心在于赋能：无论你正在编写什么样的代码，Rust赋予的能力都可以帮助你走得更远，并使你可以在更为广阔的领域中充满自信地编写程序。

例如，完成某些“系统层面”的工作需要处理内存管理、数据布局及并发的底层细节。我们习惯于将这些领域内的编程视作某种神秘的魔法，只有少部分被选中的专家才能真正深入其中。他们需要投入数年的时间来学习如何避免该领域内那些臭名昭著的陷阱，但即便是最为谨慎的实践者，也无法避免自己的代码出现漏洞、崩溃或损坏。

通过消灭这些陈旧的缺陷并提供一系列友好、精良的开发工具，Rust极大地降低了相关领域的门槛。需要“深入”底层控制的程序员可以使用Rust来完成任务，而无须承受那些常见的崩溃或安全性风险，也无须持续学习那些不断更新的工具链。更妙的是，这门语言旨在引导你自然而然地编写出可靠的代码，这些代码可以高效地运行并运用内存。

拥有底层代码编写经验的开发者可以使用Rust来实现“更具野心”的项目。例如，在Rust中引入并行是一种相对低风险的操作：编译器会为你捕捉那些常见的经典错误。你可以在代码中采用更为激进的优化策略，而无须担心意外地引发崩溃或引入漏洞。

但Rust的用途并不单单局限于底层系统编程，它极强的表达能力及工作效率足以帮助你轻松地编写出CLI应用、Web服务器及许多其他类型的代码——你会在本书中看到前两个领域内的简单示例。使用

Rust还意味着你能够在不同的领域中构建相同的技能体系；你可以编写Web应用来学习Rust，并将这些技能应用到树莓派（Raspberry Pi）上。

本书全面地介绍了Rust赋予用户的诸多可能性，它采用了通俗易懂的语言以期帮助你理解有关Rust的知识。除此之外，本书还能从整体上提升你对编程的理解和信心。让我们一起来打开新世界的大门吧！欢迎加入Rust社区！

Nicholas Matsakis和Aaron Turon

前言

欢迎阅读《Rust权威指南》，我们会在本书中深入浅出地向你介绍Rust语言！

Rust是一门可以帮助你开发出高效率、高可靠性软件的编程语言。以往的编程语言往往无法同时兼顾高水准的工程体验与底层的控制能力，而Rust则被设计出来挑战这一目标，它力图同时提供强大的工程能力及良好的开发体验，在给予开发者控制底层细节能力（比如内存操作）的同时，避免传统语言带来的诸多麻烦。

谁是Rust的目标用户

基于各种各样的原因，Rust对于许多人来讲都是一门相当理想的语言。现在让我们看一看其中最重要的一些群体。

开发团队

Rust已经被证明可以高效地应用于大规模的、拥有不同系统编程背景的开发团队。底层代码总是容易出现各种各样隐晦的错误，对于大部分编程语言来说，想要发现这些错误，要么通过海量的测试样例，要么通过优秀程序员细致的代码评审。而在Rust的世界里，大部分的错误（甚至包括并发环境中产生的错误）都可以在编译阶段被编译器发现并拦截。得益于编译器这种类似于守门员的角色，开发团队可以在更多的时间内专注于业务逻辑而非错误调试。

当然，Rust也附带了一系列面向系统级编程的现代化开发工具：

- Cargo提供了一套内置的依赖管理与构建工具。通过Cargo，你可以在Rust生态系统中一致地、轻松地增加、编译及管理依赖。
- Rustfmt用于约定一套统一的编码风格。
- The Rust Language Server则为集成开发环境（IDE）提供了可供集成的代码补全和错误提示工具。

通过使用上述工具，开发者可以有效率地进行系统级编程。

学生

对于那些有兴趣接触系统编程的学生而言，Rust也是一个非常好的选择，已经有不少人基于Rust来学习诸如操作系统开发之类的课程。另外，我们拥有一个非常热情的社区，社区成员们总是乐于回答来自初学者的各种问题。Rust开发团队希望通过本书让更多的人，特别是学生，能更加轻松地接触、学习系统编程的各种概念。

企业

目前已经有数百家或大或小的企业，将Rust用于生产环境并用它来处理各式各样的任务。这些任务包括命令行工具开发、Web服务开发、DevOps工具开发、嵌入式设备开发、音频图像分析转码、数字货币交易、生物信息提取、搜索引擎开发、物联网开发、机器学习算法研究，以及Firefox网络浏览器中的大部分功能开发。

开源开发者

当然，我们欢迎所有愿意参与构建Rust编程语言本身，或者周边社区、开发工具及第三方库的开发者。你们的贡献对于构建一个良好的Rust语言生态环境非常重要！

重视速度与稳定性的开发者

Rust适用于那些重视速度与稳定性的开发者。当谈论到速度时，我们不仅是指Rust程序可以拥有良好的运行时效率，而且还期望Rust可以提供良好的开发时效率。得益于Rust编译器的静态检查能力，我们可以稳定地在开发过程中增添功能或重构代码。与此形成鲜明对比的是，在缺少这些检查能力的语言中，开发者往往恐惧于修改那些脆弱的遗留代码。此外，得益于对零开销抽象这一概念的追求，开发者可以在无损耗的前提下使用高级语言特性。Rust力图使安全的代码也同样高效。

当然，这里提到的只是Rust使用场景中最有代表性的一部分用户，Rust语言也希望能够服务于尽可能多的其他开发者群体。总的来说，Rust最大的目标在于通过同时保证安全与效率、运行速度与编程体验，消除数十年来程序员们不得不接受的那些取舍。不妨给Rust一个机会，让我们一起来看一看它是否适合你。

谁是本书的目标读者

对于本书的读者，我们假设你已经使用过某种其他编程语言。虽然我们努力使本书的内容能够被具有不同编程背景的读者所接受，但我们不会花太多时间去讨论一些基本的编程概念。如果你对于编程是完全陌生的，那么你最好先阅读一些入门类的编程图书。

如何阅读本书

通常而言，我们假定读者按顺序从头到尾阅读本书。一开始我们会简单地介绍一些概念，接着在随后的章节中逐步深入，并有针对性地对其中的细节进行讨论。后面章节的讨论建立在前面章节引入的概念之上。

在本书中，你会发现两种类型的章节：概念讨论类章节和项目实践类章节。在概念讨论类章节中，你会接触到Rust的某些特性；在项目实践类章节中，我们会利用之前已经讲解过的Rust特性来共同构建一些小程序。第2章、第12章、第20章属于项目实践类章节，其余章节属于概念讨论类章节。

第1章会介绍如何安装Rust，如何编写“Hello, World!”程序，以及如何使用Cargo来对它进行管理及构建。

第2章会从实践的角度对Rust语言进行介绍，这里我们会从较高的层次去覆盖一系列概念，并在之后的章节中逐步深入研究细节。如果你是一个实践派，想要立即动手编写代码，那么第2章正好适合你。第3章会介绍Rust中类似于其他语言的那些特性，心急的人也许会尝试跳过这一章，并直接阅读第4章中关于Rust所有权系统的内容。相反，如果你是一个特别重视细节的学习者，期望一步一步了解清楚每一个角落，那么我建议你跳过第2章，从第3章开始按顺序阅读，并在想要通过实践来巩固知识点时再返回第2章进行阅读。

第5章会讨论结构体和方法，第6章会包含枚举、match表达式及if let控制流结构的相关内容。你将学会在Rust中使用结构体及枚举来创建自定义类型。

在第7章中，你会了解到Rust中的模块系统及私有性规则，并学会如何使用它们来组织代码和设计公共接口（API）。第8章会介绍一些标准库中提供的常用数据结构，比如Vec（动态数组）、String（字符串）及HashMap（哈希表）。第9章会讨论Rust中关于错误处理的一些设计理念和工具。

第10章会深入讲解关于泛型、trait（特征）和生命周期的概念，它们赋予了你复用代码的能力。第11章则是关于如何在Rust中构建测试系统的内容。即便是有Rust的安全检查，我们也要通过测试来保障业务逻辑上的正确性。在第12章中，我们会实现命令行工具grep的一些功能子集，用于在文件中搜索某些特定文本，为此我们会用到很多前面章节中讨论的概念。

第13章会讨论Rust中与函数式编程相关的概念：闭包与迭代器。在第14章中，我们会更加深入地了解Cargo，以及与他人共享代码库的一些最佳实践。第15章会讨论标准库中的智能指针，以及它们所实现的相关trait。

在第16章中，我们会讨论多个不同的并发编程模型，并看一看Rust是如何让多线程编程变得不那么恐怖的。第17章则着眼于比较Rust与常见的面向对象编程范式的不同风格。

第18章是关于模式及模式匹配的介绍，它们给Rust语言带来了异常强大的表达能力。第19章则会覆盖一些有趣的高级主题，包括对不安全Rust、宏、trait、类型、函数及闭包的更深入的讨论。

终于，在第20章中，我们将从底层开始实现一个完整的多线程Web服务器！

最后的附录内会包含一系列有关语言的实用参考资料。附录A会列举Rust中全部的关键字，附录B会列举Rust中所有的运算符及其他符号，附录C会包含标准库中提供的可派生trait，附录D会介绍一些有用的开发工具，附录E会解释Rust中的版本机制。

当然，不管你怎样阅读本书都是可以的。假如你想要跳过某个特定的章节，那就跳过吧，你可以在感到疑惑的时候再返回略过的那些部分。用你觉得最舒服的方式去阅读本书就好！

在学习Rust的过程中，掌握如何阅读编译器显示的错误提示信息是一项尤为重要的能力：它们能够引导你编写出可用的代码。为此，我们会故意提供许多无法通过编译的示例，进而展示相关情境下编译器输出的错误提示信息。所以，在本书中随意挑选出来的示例代码也许根本就无法通过编译！请仔细阅读上下文来确定你尝试运行的示例代码是否是一段故意写错的代码。在大部分情况下，我们会指引你将不能编译的代码纠正为正确版本。

致谢

我们想要感谢那些参与了Rust开发的人们，这样一门令人惊叹的语言绝对值得去编写一本书。我们感谢Rust社区中的所有人，你们的热情构建了一个值得更多伙伴参与进来的伟大社区。

我们要特别感谢那些阅读过本书早期版本并提供了众多反馈、错误报告及修改请求的读者。还要特别感谢Eduard-Mihai Burtescu与Alex Crichton提供的技术审查，以及Karen Rustad Tölva设计的封面。感谢我们在No Starch的编辑团队，Bill Pollock、Liz Chadwick与Janelle Ludowise协助完善并完成了本书的出版工作。

Steve想要感谢一位异常出色的合著者Carol，她使本书能够更快、更好地完成。另外，还要感谢Ashley Williams，她对本书的整个编写过程提供了难以想象的支持。

Carol想要感谢Steve激起了自己对Rust的兴趣，并给予了自己共同编写本书的机会。感谢家人长久的爱与支持，特别是丈夫Jake Goulding及女儿Vivian。

关于技术审校者

黄东旭，PingCAP联合创始人兼CTO，资深基础软件工程师、架构师，曾就职于微软亚洲研究院、网易有道及豌豆荚，是Go和Rust语言的早期实践者。擅长分布式系统及数据库开发，在分布式存储领域有丰富的经验和独到的见解。作为狂热的开源爱好者及开源软件作者，代表作品有分布式Redis缓存方案Codis，以及分布式关系型数据库TiDB。2015年开始创业，成立PingCAP，在PingCAP的主要工作是从零开始设计并研发开源NewSQL数据库TiDB，目前该项目在GitHub上累积的Star数已超过22000，成为本领域全球顶级的开源项目，其中的底层分布式存储引擎TiKV是Rust社区中的知名项目。

张汉东，资深软件工程师、企业独立咨询师、技术作者和译者、创业者。爱好读书、写作，喜欢研究技术、学习之道、思维认知等领域。曾在互联网行业沉浮十余载，先后效力于电商、社交游戏、广告和众筹领域。作为企业独立咨询师，先后为华为、思科、平安科技、闪迪等公司提供过咨询服务。2015年开始学习Rust语言，并参与了国内Rust社区的管理和运营。在2018年打造了《Rust日报》频道，深受Rustacean们的喜爱。为初学者精心打造了Rust必学第一课：《如何系统地学习Rust语言》（知乎Live），获得五星好评。2019年初出版了技术畅销书《Rust编程之道》，深受好评。目前正在青少年思维心智领域开疆拓土，努力打造属于自己的教育品牌。

目录

[内容简介](#)

[译者序](#)

[序](#)

[前言](#)

[致谢](#)

[关于技术审校者](#)

[第1章 入门指南](#)

[安装](#)

[在Linux或macOS环境中安装Rust](#)

[在Windows环境中安装Rust](#)

[更新与卸载](#)

[常见问题](#)

[本地文档](#)

[Hello, World!](#)

[创建一个文件夹](#)

[编写并运行一个Rust程序](#)

[对这个程序的剖析](#)

[编译与运行是两个不同的步骤](#)

[Hello, Cargo!](#)

[使用Cargo创建一个项目](#)

[使用Cargo构建和运行项目](#)

[以Release模式进行构建](#)

[学会习惯Cargo](#)

[总结](#)

[第2章 编写一个猜数游戏](#)

[创建一个新的项目](#)

[处理一次猜测](#)

[使用变量来存储值](#)

[使用Result类型来处理可能失败的情况](#)

[通过println! 中的占位符输出对应的值](#)

[尝试运行代码](#)

[生成一个保密数字](#)

[借助包来获得更多功能](#)

[生成一个随机数](#)
[比较猜测数字与保密数字](#)
[使用循环来实现多次猜测](#)
[在猜测成功时优雅地退出](#)
[处理非法输入](#)
[总结](#)
[第3章 通用编程概念](#)
[变量与可变性](#)
[变量与常量之间的不同](#)
[隐藏](#)
[数据类型](#)
[标量类型](#)
[复合类型](#)
[函数](#)
[函数参数](#)
[函数体中的语句和表达式](#)
[函数的返回值](#)
[注释](#)
[控制流](#)
[if表达式](#)
[使用循环重复执行代码](#)
[总结](#)
[第4章 认识所有权](#)
[什么是所有权](#)
[所有权规则](#)
[变量作用域](#)
[String类型](#)
[内存与分配](#)
[所有权与函数](#)
[返回值与作用域](#)
[引用与借用](#)
[可变引用](#)
[悬垂引用](#)
[引用的规则](#)
[切片](#)
[字符串切片](#)
[其他类型的切片](#)

总结

第5章 使用结构体来组织相关联的数据

定义并实例化结构体

在变量名与字段名相同时使用简化版的字段初始化方法

使用结构体更新语法根据其他实例创建新实例

使用不需要对字段命名的元组结构体来创建不同的类型

没有任何字段的空结构体

一个使用结构体的示例程序

使用元组来重构代码

使用结构体来重构代码：增加有意义的描述信息

通过派生 trait 增加实用功能

方法

定义方法

带有更多参数的方法

关联函数

多个impl块

总结

第6章 枚举与模式匹配

定义枚举

枚举值

Option枚举及其在空值处理方面的优势

控制流运算符match

绑定值的模式

匹配Option<T>

匹配必须穷举所有的可能

通配符

简单控制流if let

总结

第7章 使用包、单元包及模块来管理日渐复杂的项目

包与单元包

通过定义模块来控制作用域及私有性

用于在模块树中指明条目的路径

使用pub关键字来暴露路径

使用super关键字开始构造相对路径

将结构体或枚举声明为公共的

用use关键字将路径导入作用域

创建use路径时的惯用模式

[使用as关键字来提供新的名称](#)
[使用pub use重导出名称](#)
[使用外部包](#)
[使用嵌套的路径来清理众多use语句](#)
[通配符](#)
[将模块拆分为不同的文件](#)
[总结](#)
[第8章 通用集合类型](#)
[使用动态数组存储多个值](#)
[创建动态数组](#)
[更新动态数组](#)
[销毁动态数组时也会销毁其中的元素](#)
[读取动态数组中的元素](#)
[遍历动态数组中的值](#)
[使用枚举来存储多个类型的值](#)
[使用字符串存储UTF-8编码的文本](#)
[字符串是什么](#)
[创建一个新的字符串](#)
[更新字符串](#)
[字符串索引](#)
[字符串切片](#)
[遍历字符串的方法](#)
[字符串的确没那么简单](#)
[在哈希映射中存储键值对](#)
[创建一个新的哈希映射](#)
[哈希映射与所有权](#)
[访问哈希映射中的值](#)
[更新哈希映射](#)
[哈希函数](#)
[总结](#)
[第9章 错误处理](#)
[不可恢复错误与panic!](#)
[使用panic! 产生的回溯信息](#)
[可恢复错误与Result](#)
[匹配不同的错误](#)
[失败时触发panic的快捷方式: unwrap和expect](#)
[传播错误](#)

[要不要使用panic!](#)

[示例、原型和测试](#)

[当你比编译器拥有更多信息时](#)

[错误处理的指导原则](#)

[创建自定义类型来进行有效性验证](#)

[总结](#)

[第10章 泛型、trait与生命周期](#)

[通过将代码提取为函数来减少重复工作](#)

[泛型数据类型](#)

[在函数定义中](#)

[在结构体定义中](#)

[在枚举定义中](#)

[在方法定义中](#)

[泛型代码的性能问题](#)

[trait: 定义共享行为](#)

[定义trait](#)

[为类型实现trait](#)

[默认实现](#)

[使用trait作为参数](#)

[返回实现了trait的类型](#)

[使用trait约束来修复largest函数](#)

[使用trait约束来有条件地实现方法](#)

[使用生命周期保证引用的有效性](#)

[使用生命周期来避免悬垂引用](#)

[借用检查器](#)

[函数中的泛型生命周期](#)

[生命周期标注语法](#)

[函数签名中的生命周期标注](#)

[深入理解生命周期](#)

[结构体定义中的生命周期标注](#)

[生命周期省略](#)

[方法定义中的生命周期标注](#)

[静态生命周期](#)

[同时使用泛型参数、trait约束与生命周期](#)

[总结](#)

[第11章 编写自动化测试](#)

[如何编写测试](#)

测试函数的构成
使用assert! 宏检查结果
使用assert_eq! 宏和assert_ne! 宏判断相等性
添加自定义的错误提示信息
使用should_panic检查panic
使用Result<T, E>编写测试
控制测试的运行方式
并行或串行地进行测试
显示函数输出
只运行部分特定名称的测试
通过显式指定来忽略某些测试
测试的组织结构
单元测试
集成测试
总结
第12章 I/O项目：编写一个命令行程序
接收命令行参数
读取参数值
将参数值存入变量
读取文件
重构代码以增强模块化程度和错误处理能力
二进制项目的关注点分离
修复错误处理逻辑
从main中分离逻辑
将代码分离为独立的代码包
使用测试驱动开发来编写库功能
编写一个会失败的测试
编写可以通过测试的代码
处理环境变量
为不区分大小写的search函数编写一个会失败的测试
实现search_case_insensitive函数
将错误提示信息打印到标准错误而不是标准输出
确认错误被写到了哪里
将错误提示信息打印到标准错误
总结
第13章 函数式语言特性：迭代器与闭包
闭包：能够捕获环境的匿名函数

使用闭包来创建抽象化的程序行为
闭包的类型推断和类型标注
使用泛型参数和Fn trait来存储闭包
Cacher实现的局限性
使用闭包捕获上下文环境
使用迭代器处理元素序列
Iterator trait和next方法
消耗迭代器的方法
生成其他迭代器的方法
使用闭包捕获环境
使用Iterator trait来创建自定义迭代器
改进I/O项目
使用迭代器代替clone
使用迭代器适配器让代码更加清晰
比较循环和迭代器的性能
总结
第14章 进一步认识Cargo及crates.io
使用发布配置来定制构建
将包发布到crates.io上
编写有用的文档注释
使用pub use来导出合适的公共API
创建crates.io账户
为包添加元数据
发布到crates.io
发布已有包的新版本
使用cargo yank命令从cargo.io上移除版本
Cargo工作空间
创建工作空间
在工作空间中创建第二个包
使用cargo install从crates.io上安装可执行程序
使用自定义命令扩展Cargo的功能
总结
第15章 智能指针
使用Box<T>在堆上分配数据
使用Box<T>在堆上存储数据
使用装箱定义递归类型
通过Deref trait将智能指针视作常规引用

使用解引用运算符跳转到指针指向的值

把Box<T>当成引用来操作

定义我们自己的智能指针

通过实现Deref trait来将类型视作引用

函数和方法的隐式解引用转换

解引用转换与可变性

借助Drop trait在清理时运行代码

使用std::mem::drop提前丢弃值

基于引用计数的智能指针Rc<T>

使用Rc<T>共享数据

克隆Rc<T>会增加引用计数

RefCell<T>和内部可变性模式

使用RefCell<T>在运行时检查借用规则

内部可变性：可变地借用一个不可变的值

将Rc<T>和RefCell<T>结合起来实现一个拥有多重所有权的可变数据

循环引用会造成内存泄漏

创建循环引用

使用Weak<T>代替Rc<T>来避免循环引用

总结

第16章 无畏并发

使用线程同时运行代码

使用spawn创建新线程

使用join句柄等待所有线程结束

在线程中使用move闭包

使用消息传递在线程间转移数据

通道和所有权转移

发送多个值并观察接收者的等待过程

通过克隆发送者创建多个生产者

共享状态的并发

互斥体一次只允许一个线程访问数据

RefCell<T> Rc<T>和Mutex<T> Arc<T>之间的相似性

使用Sync trait和Send trait对并发进行扩展

允许线程间转移所有权的Send trait

允许多线程同时访问的Sync trait

手动实现Send和Sync是不安全的

总结

第17章 Rust的面向对象编程特性

面向对象语言的特性
对象包含数据和行为
封装实现细节

作为类型系统和代码共享机制的继承
使用trait对象来存储不同类型的值

为共有行为定义一个trait

实现trait

trait对象会执行动态派发

trait对象必须保证对象安全

实现一种面向对象的设计模式

定义Post并新建一个处于草稿状态下的新实例

存储文章内容的文本

确保草稿的可读内容为空

请求审批文章并改变其状态

添加approve方法来改变content的行为

状态模式的权衡取舍

总结

第18章 模式匹配

所有可以使用模式的场合

match分支

if let条件表达式

while let条件循环

for循环

let语句

函数的参数

可失败性：模式是否会匹配失败

模式语法

匹配字面量

匹配命名变量

多重模式

使用... 来匹配值区间

使用解构来分解值

忽略模式中的值

使用匹配守卫添加额外条件

@绑定

总结

第19章 高级特性

不安全Rust

不安全超能力

解引用裸指针

调用不安全函数或方法

访问或修改一个可变静态变量

实现不安全trait

使用不安全代码的时机

高级trait

在trait的定义中使用关联类型指定占位类型

默认泛型参数和运算符重载

用于消除歧义的完全限定语法：调用相同名称的方法

用于在trait中附带另外一个trait功能的超trait

使用newtype模式在外部类型上实现外部trait

高级类型

使用newtype模式实现类型安全与抽象

使用类型别名创建同义类型

永不返回的Never类型

动态大小类型和Sized trait

高级函数与闭包

函数指针

返回闭包

宏

宏与函数之间的差别

用于通用元编程的macro_rules! 声明宏

基于属性创建代码的过程宏

如何编写一个自定义derive宏

属性宏

函数宏

总结

第20章 最后的项目：构建多线程Web服务器

构建单线程Web服务器

监听TCP连接

读取请求

仔细观察HTTP请求

编写响应

返回真正的HTML

验证请求有效性并选择性地响应

[少许重构](#)

[把单线程服务器修改为多线程服务器](#)

[在现有的服务器实现中模拟一个慢请求](#)

[使用线程池改进吞吐量](#)

[优雅地停机与清理](#)

[为ThreadPool实现Drop trait](#)

[通知线程停止监听任务](#)

[总结](#)

[附录A 关键字](#)

[当前正在使用的关键字](#)

[将来可能会使用的保留关键字](#)

[原始标识符](#)

[附录B 运算符和符号](#)

[运算符](#)

[非运算符符号](#)

[附录C 可派生trait](#)

[面向程序员格式化输出的Debug](#)

[用于相等性比较的PartialEq和Eq](#)

[使用PartialOrd和Ord进行次序比较](#)

[使用Clone和Copy复制值](#)

[用于将值映射到另外一个长度固定的值的Hash](#)

[用于提供默认值的Default](#)

[附录D 有用的开发工具](#)

[使用rustfmt自动格式化代码](#)

[使用rustfix修复代码](#)

[使用Clippy完成更多的代码分析](#)

[使用Rust语言服务器来集成IDE](#)

[附录E 版本](#)

第1章

入门指南



好了！现在让我们开始正式了解Rust的旅程。千里之行，始于足下。我们会在本章讨论如下议题：

- 在Linux、macOS及Windows环境中安装Rust。
- 编写一个输出“Hello, World! ”字符串的小程序。
- 使用Rust附带的包管理和构建工具cargo。

安装

学习Rust的第一步自然是安装它。我们会通过一个叫作rustup的命令行工具来完成Rust的下载与安装，这个工具还被用来管理不同的Rust发行版本及其附带的工具链。当然，下载时需要你有一个顺畅的网络连接。

注意

假如你因为某种原因而不愿意使用rustup，那么请前往Rust官方网站寻找其他可用的安装方式。

接下来的步骤会安装最新的Rust稳定版本。值得一提的是，Rust的稳定性保证了所有发行版本都是向后兼容的，这意味着本书中所有可编译的示例都将可以在更新的Rust版本中编译通过。在不同的版本下，示例在编译时的输出内容也许会有些许细微的差异，这是因为Rust在升级的过程中改进了编译器的错误提示信息和警告信息。换句话说，任何通过以下步骤安装的最新Rust版本都能够顺利运行本书中的所有内容。

命令行标记

在本书中，我们演示了一些将会在终端中使用的命令行程序。所有需要被输入终端的命令行都会以字符\$开头。这并不代表你需要实际去输入这个字符，它只是被用来标记每个命令行的起始位置。那些没有\$前置标记的行，则是之前命令的输出结果。另外，一些特定于PowerShell的演示将会使用>来代替\$作为标记。

在Linux或macOS环境中安装Rust

假如你使用的操作系统是Linux或macOS，那么请打开命令行终端，并且输入命令：

```
$ curl https://sh.rustup.rs -sSf | sh
```

这条命令会下载并执行一个脚本来安装rustup工具，进而安装最新的Rust稳定版本。该脚本可能会在执行过程中请求输入你的密码。一旦安装成功，你将能够看到如下所示的输出：

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

当然，你也可以独立下载这个脚本，并在执行前检查一下其中的内容。

上面的安装过程会自动将Rust工具链添加到环境变量PATH中，并在下一次登录终端时生效。假如你想要立即开始使用Rust而不用重新启动终端，那么你可以在终端中运行如下所示的命令来让配置立即生效：

```
$ source $HOME/.cargo/env
```

或者，你也可以向`~/.bash_profile`文件中添加下面的语句，手动将Rust添加到环境变量PATH中：

```
$ export PATH="$HOME/.cargo/bin:$PATH"
"
```

另外，为了正常地编译执行Rust程序，你还需要一个链接器（linker）。虽然你的系统内极有可能已经配备了链接器，但假如你在编译Rust程序的过程中出现了链接器无法正常使用的错误，那么你也可以自行安装一个。由于C语言编译器通常都会附带运行正常的链接器，所以你可以查询当前平台的相关文档来安装一个C语言编译器。除此之外，一部分常用的Rust包会依赖于使用C语言编写的代码，即便是为了编译这些Rust代码，你也需要安装一个C语言编译器。

在Windows环境中安装Rust

假如你使用的是Windows操作系统，那么最好前往Rust官方网站的安装页面，并根据网页上的说明来安装Rust。你也许会在安装的过程中发现这样一条警告信息，它要求你同时安装Visual Studio 2013或更高版本的C++构建工具。解决这个问题最简单的方式就是前往Visual Studio官方网站的下载页面，并在其他工具和框架页面中下载需要的内容。

本书中使用的大部分命令行程序都可以同时运行于 *cmd.exe* 和 PowerShell上。如果出现特殊情形，我们会单独进行说明。

更新与卸载

在使用rustup成功地安装了Rust后，你可以非常简单地通过如下所示的命令来更新Rust版本：

```
$ rustup update
```

当然，你也可以通过如下所示的命令卸载rustup及Rust工具链：

```
$ rustup self uninstall
```

常见问题

你可以在终端中输入如下所示的命令来检查Rust是否已经被正确地安装了：

```
$ rustc  
--version
```

一切顺利的话，你应该可以在命令输出中以如下所示的格式依次看到最新稳定版本的版本号、当前版本的哈希码及版本的提交日期：

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

假如你无法看到这样的输出信息，并且使用的是Windows系统，那么你可以尝试检查Rust工具链是否已经被添加到了环境变量%PATH%中。假如你已经看到了这条输出信息，但Rust依然不能正常工作，那么你可能需要到另外的地方寻求帮助。最简单的方式是通过Discord访问Rust官方讨论组的#beginners频道，你可以在这里与其他的Rustaceans（这是我们内部对Rust用户的昵称）进行实时交流并找到愿意帮助你的伙伴。除此之外，你还可以通过Rust用户论坛或Stack Overflow来获得帮助。

本地文档

安装工具在执行的过程中会在本地生成一份离线的文档，你可以通过命令rustup doc在网页浏览器中打开它。

当你在标准库中发现了某个自己并不清楚用途或使用方式的类型或函数时，可以通过离线文档在任何时刻查询对应的应用程序接口（API）来获得相关信息！

Hello, World!

现在，你应该已经成功安装好了Rust。让我们遵从传统，从编写一个可以打印出“Hello, world!”的小程序开始正式的学习旅程。

注意

本书假定你已经熟悉了基本的终端操作与常用命令。开发Rust程序并不会对你所使用的编辑工具有任何的要求，如果你喜欢使用某个IDE（Integrated Development Environment，集成开发环境），那么就用你喜欢的IDE好了。许多常用的IDE都已经针对Rust实现了某种程度上的支持，你可以通过相应的IDE文档来了解更多的细节。值得高兴的是，Rust开发团队在集中精力提供流畅、舒适的IDE支持，不断优化编码体验！

创建一个文件夹

首先，我们需要创建一个文件夹来存储编写的Rust代码。通常而言，Rust不会限制我们存储代码的位置，但是针对本书中的各种练习和项目，我们建议你创建一个可以集合所有项目的根文件夹，然后将本书中所有的项目放在里面。

现在，你可以打开终端并输入相应命令，来创建我们的文件夹及第一个“Hello, world!”项目了。

对于Linux系统、macOS系统，以及Windows系统的PowerShell终端来说，输入的命令如下所示：

```
$ mkdir ~/projects
```

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

对于Windows系统的CMD终端，输入的命令如下所示：

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

编写并运行一个Rust程序

接下来，我们需要创建一个名为*main.rs* 的源文件。在命名规则上，Rust文件总是以*.rs* 扩展名结尾。如果在名字中使用了多个单词，那么你可以使用下画线来隔开它们。比如你最好使用*hello_world.rs* 作为文件名而不是*helloworld.rs*。

现在，你可以打开刚刚创建的*main.rs* 文件，并键入示例1-1中的代码。

main.rs

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

示例1-1：一个输出“Hello, world!”的程序

然后保存文件并回到终端窗口。在Linux或macOS系统中，你可以通过输入如下所示的命令来编译并运行这个文件：

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

在Windows系统中，你需要将上面命令中的`.\main`替换成`.\main.exe`:

```
> rustc main.rs  
> .\main.exe
```

Hello, world!

无论使用哪种操作系统，你都应该能看到终端中输出的“Hello, world!”字符串结果。如果没有看到此输出结果，你最好回到本章的“常见问题”一节寻求帮助。

假如一切顺利，那么恭喜你！你已经完成了第一个Rust程序，并正式成为了Rust开发者！欢迎来到Rust的世界！

对这个程序的剖析

现在，让我们回过头来仔细看看“Hello, world!”程序中到底发生了什么。第一个值得注意的部分如下所示：

```
fn main() {  
}
```

这部分代码定义了Rust中的一个函数。这里的`main`函数会比较特殊：当你运行一个可执行Rust程序的时候，所有的代码都会从这个入口函数开始运行。这段代码的第一行声明了一个名为`main`的、没有任何参数和返回值的函数。如果某天你需要给函数声明参数的话，那么就必须把它们放置在圆括号`()`中。

另外，那对花括号`{}`被用来标记函数体，Rust要求所有的函数体都要被花括号包裹起来。按照惯例，我们推荐把左花括号与函数声明置于同一行并以空格分隔。

在本书撰写的过程中，一个名为`rustfmt`的工具正处于开发状态。假如你希望在不同的项目中保持同样的编码风格，那么`rustfmt`可以帮助你将代码自动格式化为约定的风格。Rust开发团队正计划着将这个工具包含进Rust的发行版本中（就像`rustc`一样）。在你阅读本书时，也许它已经被安装到了你的计算机中！你可以阅读在线文档来获得更多关于它的信息。

再来看一看main函数体中的代码：

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

这一行代码完成了整个程序的所有工作：将字符串输出到终端上。这里有4个需要注意的细节。首先，标准Rust风格使用4个空格而不是Tab来实现缩进。

其次，我们调用了一个被叫作println! 的宏。假如我们调用的是一个普通函数，那么这里会以去掉! 符号的println来进行标记。我们会在第19章对Rust宏进行深入地讨论，现在你只需要记住，Rust中所有以! 结尾的调用都意味着你正在使用一个宏而不是普通函数。

再次，你可以看到"Hello, world!"字符串本身。我们把这个字符串作为参数传入了println!，并最终将它显示到了终端屏幕上。

最后，我们使用了一个分号(;)作为这一行的结尾，它表明当前的表达式已经结束，而下一个表达式将要开始。大部分的Rust代码行都会以分号来结尾。

编译与运行是两个不同的步骤

你应该已经运行过刚刚编写的程序了，让我们来详细地讨论一下这个过程中的每一个步骤。

在运行一段Rust程序之前，你必须输入rustc命令及附带的源文件名参数来编译它：

```
$ rustc main.rs
```

假如你曾经有过C/C++开发的背景，那么你就会发现这个步骤与gcc或clang编译十分相似。一旦编译成功，我们就会获得一个二进制的可执行文件。

在Linux系统、macOS系统，以及Windows系统的PowerShell中，我们可以通过输入如下所示的ls命令看到刚刚生成的可执行文件。在Linux及macOS系统中，你将在输出中看到两个文件；在Windows系统的PowerShell中，你将看到与CMD输出结果相同的3个文件。

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

在Windows系统的CMD中，你需要输入如下所示的命令：

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

显示的文件里面有我们刚刚创建的、以.*rs* 为后缀的源代码文件，还有生成的可执行文件（也就是Windows系统下的*main.exe*，或其余系统下的*main*）。如果你使用的是Windows系统，那么你还会看到一个附带调试信息、以.*pdb* 为后缀的文件。现在，我们可以通过如下所示的方式运行*main* 或*main.exe* 文件了：

```
$ ./main # or .\main.exe on Windows
```

如果*main.rs* 还是我们刚刚创建的“Hello, world!” 程序，那么你就会在终端中看到Hello, world! 的字符串输出。

假如你更加熟悉某种类似于Ruby、Python或JavaScript之类的动态语言，你可能还不太习惯在运行之前需要先进行编译。Rust是一种预编译语言，这意味着当你编译完Rust程序之后，便可以将可执行文件交付于其他人，并运行在没有安装Rust的环境中。而如果你交付给其他人的是一份.*rb*、.*py* 或.*js* 文件，那么他们就必须要拥有对应的Ruby、Python或JavaScript实现来执行程序。当然，这些语言只需要用简单的一句命令就可以完成程序的编译和运行。这也算是语言设计上的权衡与取舍吧。

仅仅使用rustc编译简单的程序并不会造成太大的麻烦，但随着项目的规模越来越大，协同开发的人员越来越多，管理项目依赖、代码构建这样的事情就会变得越来越复杂和琐碎。下面将介绍一个帮助我们简化问题，并能够实际运用于生产的Rust构建工具：Cargo。

Hello, Cargo!

Cargo是Rust工具链中内置的构建系统及包管理器。由于它可以处理众多诸如构建代码、下载编译依赖库等琐碎但重要的任务，所以绝大部分的Rust用户都会选择它来管理自己的Rust项目。

因为我们编写的简单程序不会依赖于任何外部库，所以当我们通过Cargo来构建这个“Hello, world!”项目时，它只会用到Cargo中负责构建代码的那部分功能。初看上去，它和rustc并没有太大的区别，但当你开始尝试编写更加复杂的Rust程序时，Cargo会让添加、管理依赖这件事变得十分轻松。

由于绝大部分的Rust项目都使用了Cargo，所以我们将在本书剩余的章节中假设你也会基于Cargo来进行项目管理。假如你使用了本章中“安装”一节提到的标准程序来安装Rust，那么Cargo就已经被附带在了当前的Rust工具链里。而假如你选择了其他方式安装Rust，那么你最好先在终端输入如下所示的命令来检查Cargo是否已经被安装妥当：

```
$ cargo --version
```

当你看到上面的命令输出了一串版本号时，那么就表示一切正常，Cargo可以正常使用了。但如果你看到了类似于command not found的错误提示信息，那么你最好重新阅读安装Rust时附带的文档来单独安装Cargo。

使用Cargo创建一个项目

现在，让我们使用Cargo创建一个新的项目，并与之前的“Hello, world!”项目做一个对比，来看一看它们之间有何异同。将当前的目

录跳转至 *projects* 文件夹（或者你用来存储项目的任意位置），然后运行如下所示的命令：

```
$ cargo new hello_cargo
```

```
$ cd hello_cargo
```

第一条命令会创建一个名为 *hello_cargo* 的项目。由于我们将这个项目命名为 *hello_cargo*，所以 Cargo 会以同样的名字创建项目目录并放置它生成的文件。

现在，让我们进入 *hello_cargo* 文件夹，你可以看到 Cargo 刚刚生成的两个文件与一个目录：一个名为 *Cargo.toml* 的文件，以及一个名为 *main.rs* 的源代码文件，该源代码文件被放置在 *src* 目录下。与此同时，Cargo 还会初始化一个新的 Git 仓库并生成默认的 *.gitignore* 文件。

注意

Git 是一种常见的版本管理系统。你也可以在创建项目时，通过使用 *--vcs* 参数来选择不使用版本控制系统，或者使用某个特定的版本控制系统。运行命令 *cargo new --help* 可以获得关于命令参数的更多说明。

Cargo.toml 中的内容如示例 1-2 所示，你可以使用文本编辑器打开它。

Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

示例 1-2：通过 *cargo new* 生成的 *Cargo.toml* 文件中的内容

Cargo使用TOML（Tom's Obvious, Minimal Language）作为标准的配置格式，正如这里的*Cargo.toml*一样。

首行文本中的[`package`]是一个区域标签，它表明接下来的语句会被用于配置当前的程序包。随着我们在这个文件中增加更多的信息，你还会见识到更多其他的区域（section）。

紧随标签后的3行语句提供了Cargo编译这个程序时需要的配置信息，它们分别是程序名、版本号及作者信息。在Cargo生成*Cargo.toml*的过程中，它会尝试着从环境变量中获得你的名字与电子邮箱，但如果这些生成的信息与实际情况不符，你也可以直接修改并保存这个文件。我们会在附录E中讨论这里的edition字段。

最后一行文本中的[`dependencies`]同样是一个区域标签，它表明随后的区域会被用来声明项目的依赖。在Rust中，我们把代码的集合称作包（crate）[\[1\]](#)。虽然目前的项目暂时还不需要使用任何的第三方包，但你可以在第2章的第一个实践项目中看到这个配置区域的用法。

好了，现在让我们打开*src/main.rs*来看一下吧：

src/main.rs

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

是的，正如示例1-1中所写的一样，Cargo帮我们生成了一个输出“Hello, world!”的小程序。到目前为止，Cargo生成的项目与我们在上一节中手动生成的项目相比，其区别就是源文件*main.rs*被放置到了*src*目录下，并且在项目目录下多了一个叫作*Cargo.toml*的配置文件。

同样按照惯例，Cargo会默认把所有的源代码文件保存到*src*目录下，而项目根目录只被用来存放诸如README文档、许可声明、配置文件等与源代码无关的文件。使用Cargo可以帮助你合理并一致地组织自己的项目文件，从而使一切井井有条。

如果你想要使一个手动创建的项目，比如上面创建的“Hello, world!”项目，转为使用Cargo管理的项目，那么你只需把源代码文件放置到*src* 目录下，并且创建一个对应的*Cargo.toml* 配置文件即可。

使用Cargo构建和运行项目

那么使用Cargo来构建和运行项目与手动使用rustc相比又有哪些异同呢？在当前的*hello_cargo* 项目目录下，Cargo可以通过下面的命令来完成构建任务：

```
$ cargo build

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

与之前不同，这个命令会将可执行程序生成在路径*target/debug/hello_cargo*（或者 Windows 系统下的*target\debug\hello_cargo.exe*）下。你可以通过如下所示的命令运行这个可执行程序试试看：

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows

Hello, world!
```

一切正常的话，Hello, world! 应该能够被打印到终端上。首次使用命令cargo build构建的时候，它还会在项目根目录下创建一个名为*Cargo.lock* 的新文件，这个文件记录了当前项目所有依赖库的具体版本号。由于当前的项目不存在任何依赖，所以这个文件中还没有太多东西。你最好不要手动编辑其中的内容，Cargo可以帮助你自动维护它。

我们刚刚使用命令 cargo build 构建好了一个项目，并通过*target\debug\hello_cargo* 完成了运行，但我们也简单地使用 cargo run 命令来依次完成编译和运行任务：

```
$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in
```

```
0.0 secs
    Running `target/debug/hello_cargo`
Hello, world!
```

你可能会注意到，这次的输出里没有提示我们编译hello_cargo的信息。这是因为Cargo发现源代码并没有被修改，所以它就直接运行了生成的二进制可执行文件。如果我们修改了源代码，那么Cargo便会在运行之前重新构建项目，并输出如下所示的内容：

```
$ 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!
```

另外，Cargo还提供了一个叫作cargo check的命令，你可以使用这个命令来快速检查当前的代码是否可以通过编译，而不需要花费额外的时间去真正生成可执行程序：

```
$ cargo check

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

你也许会问，我们为什么需要这样一个命令？通常来讲，由于cargo check跳过了生成可执行程序的步骤，所以它的运行速度要远远快于cargo build。假如你在编码的过程中需要不断通过编译器检查错误，那么使用cargo check就会极大地加速这个过程。事实上，大部分Rust用户在编写程序的过程中都会周期性地调用cargo check以保证自己的程序可以通过编译，只有真正需要生成可执行程序时才会调用cargo build。

好了，让我们回顾一下目前接触到的关于Cargo的知识点：

- 我们可以通过cargo build或cargo check来构建一个项目。
- 我们可以通过cargo run来构建并运行一个项目。

- 构建产生的结果会被Cargo存储在 *target/debug* 目录下，而非代码所处的位置。

另外一个使用Cargo的优势在于，它的命令在不同的操作系统中都是相同的。因此，从现在开始，我们不会再引入不同系统（Linux和macOS系统，Windows系统）下的特定操作指令了。

以Release模式进行构建

当准备好发布自己的项目时，你可以使用命令 `cargo build --release` 在优化模式下构建并生成可执行程序。它生成的可执行文件会被放置在 *target/release* 目录下，而不是之前的 *target/debug* 目录下。这种模式会以更长的编译时间为代价来优化代码，从而使代码拥有更好的运行时性能。这也是存在两种不同的构建模式的原因。一种模式用于开发，它允许你快速地反复执行构建操作。而另一种模式则用于构建交付给用户的最终程序，这种构建场景不会经常发生，但却需要生成的代码拥有尽可能高效的运行时表现。值得指出的是，假如你想要对代码的运行效率进行基准测试，那么请确保你会通过 `cargo run --release` 命令进行构建，并使用 *target/release* 目录下的可执行程序完成基准测试。

学会习惯Cargo

你也许无法在较为简单的项目中意识到Cargo相对于rustc的优势，但随着程序变得越来越复杂，Cargo最终一定会证明自己的价值。对于那些由多个包构成的复杂项目而言，使用Cargo来协调整个构建过程要比手动操作简单得多。

另外，即便 *hello_cargo* 项目是如此简单，你也在创建它的过程中接触到了相当多的工具，这些工具会在你使用Rust的生涯中派上不小的用场。事实上，对于大部分现有的项目而言，你都可以通过下面几行简单的命令来从Git中检出代码、将当前目录移动到该项目的目录下及执行构建操作。

```
$ git clone someurl.com/someproject
```

```
$ cd someproject
```

```
$ cargo build
```

你可以参考Cargo的官方文档来获得更多有关它的信息。

[1] 译者注：crate是Rust中最小的编译单元，package是单个或多个crate的集合，crate和package都可以被叫作包，因为单个crate也是一个package，但package通常倾向于多个crate的组合。本书中，crate和package统一被翻译为包，只在两者同时出现且需要区别对待时，将crate译为单元包，将package译为包。

总结

你已经在自己的Rust旅程上迈出了坚实的一步！我们在本章学会了：

- 如何使用rustup安装Rust最新的稳定版本。
- 如何将Rust更新到最新版本。
- 如何打开本地安装的文档。
- 如何编写一个“Hello, world!”程序，并使用rustc来直接构建编译它。
- 如何通过Cargo来创建并运行一个新的项目。

为了帮助你逐渐习惯阅读和编写Rust代码，现在也许是时候构建一个更为复杂的项目了。因此，我们会在第2章共同编写一个“猜数游戏”。但假如你希望从更为基础的部分开始，优先学习那些常用编程概念在Rust中的作用机制，那么你也可以在阅读完第3章后再来阅读第2章。

第2章

编写一个猜数游戏



现在，让我们来共同编写一个简单的程序并快速熟悉Rust! 本章会在实际编码的过程中介绍常见的Rust概念。你可以接触到诸如let、match、类型方法、关联函数及外部依赖库等一系列知识。当然，我们只会在本章练习一些基本的使用技巧，有关这些概念背后的细节会在随后的章节中进行讨论。

我们将完成一个经典的初学者编程挑战：猜数游戏，它会首先生成一个1到100之间的随机整数，并紧接着请求玩家对这个数字进行猜测。假如玩家输入的数字与随机数不同，那么程序将给出数字偏大或偏小的提示。而假如玩家猜中了我们准备的数字，那么程序就会打印出一段祝贺信息并随之退出。

创建一个新的项目

现在，让我们前往第1章中创建的*projects* 文件夹，并使用Cargo来开始一个新的项目：

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

正如我们在第1章中了解的那样，第一行命令cargo new以项目名(guessing_game)作为首个参数；第二行命令则将当前目录修改为了新的项目文件夹目录。

现在，让我们打开新生成的*Cargo.toml* 文件：

Cargo.toml

```
[package]  
name = "guessing_game"  
version = "0.1.0"  
authors = ["Your Name <you@example.com>"]  
  
[dependencies]
```

Cargo会尝试从你的系统环境中获得当前的作者信息并将其填充至authors字段。假如这个字段存在问题，你也可以手动修正并保存它。

正如在第1章中所看到的那样，cargo new会自动生成一段输出“Hello, world!”的程序，这段程序被放置在文件*src/main.rs* 中：

src/main.rs

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

现在，让我们使用 cargo run 命令来编译并运行 “Hello, world!” 这段程序：

```
$ cargo run
```

```
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
    Running `target/debug/guessing_game`
Hello, world!
```

run命令可以在你需要快速迭代一个项目的时候派上用场，我们会在开发游戏的迭代过程中反复使用该命令来测试代码是否能够通过编译。

重新打开 *src/main.rs* 文件，我们将在这个文件中编写本章的所有代码。

处理一次猜测

猜数游戏的第一部分会请求用户进行输入，并检查该输入是否满足预期的格式。现在，让我们将示例2-1中的代码输入*src/main.rs*。

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);
}
```

示例2-1：从用户处获得一个输入并将其打印出来

由于这段代码包含了不少新鲜内容，所以让我们来一行一行地分析它们。为了获得用户的输入并将其打印出来，我们需要把标准库（也就是所谓的std）中的io模块引入当前的作用域中：

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

作为默认行为，Rust会将预导入（prelude）模块内的条目自动引入每一段程序的作用域中，它包含了一小部分相当常用的类型。但假如你需要的类型不在预导入模块内，那么我们就必须使用use语句来显式地进行导入声明。std::io库包含了许多有用的功能，我们可以使用它来获得用户的输入数据。

正如你在第1章中见到的那样，main函数是一段程序开始的地方：

```
fn main() {
```

上面的fn语法声明了一个新的函数，而紧随名称后的圆括号()则意味着当前函数没有任何参数，最后的花括号{}被用来标识函数体的开始。

函数体中的前几行使用了我们在第1章学过的println! 宏，它被用来将字符串打印到屏幕上：

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

这段代码输出的信息会向玩家展示当前的游戏内容并请求他们输入数据。

使用变量来存储值

接下来，我们创建了一个存储用户输入数据的地方：

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

这一行出现了不少新东西，我们的程序要开始变得有意思起来了！这个以let开头的语句创建了一个新的变量（variable）。再来看一看下面的例子：

```
let foo = bar;
```

这行代码创建了一个名为foo的新变量，并将它绑定到了变量bar的值上。在Rust中，变量都是默认不可变的，我们会在第3章的“变量与可变性”一节中深入讨论这一概念。现在，你只需要知道我们必须使用mut关键字来声明一个变量是可变的：

```
let foo = 5; // foo是不可变的
```

```
let mut bar = 5; // bar是可变的
```

注意

上面的`|`语法意味着当前位置到本行结尾的所有内容都是注释。Rust会在编译的过程中忽略注释，你可以在随后的第3章中看到有关注释的详细介绍。

让我们回到猜数游戏中。你现在知道`let mut guess`语句会创建出一个名为`guess`的可变变量了。在这行语句中，等号`(=)`的右边是`guess`被绑定的值，也就是调用函数`String::new`后返回的结果：一个新的`String`实例。`String`是标准库中的一个字符串类型，它在内部使用了UTF-8格式的编码并可以按照需求扩展自己的大小。

`String::new`中的`::`语法表明`new`是`String`类型的一个关联函数（associated function）。我们会针对类型本身来定义关联函数，比如本例中的`String`，而不会针对`String`的某个特定实例。关联函数在某些语言中也被称为静态方法（static method）。

这个`new`函数会创建一个新的空白字符串。你会在许多类型上发现`new`函数，因为这是创建类型实例的惯用函数名称。

总的来说，语句`let mut guess = String::new();`会创建出一个可变的变量，并在它身上绑定一个新的空白字符串。

为了引入标准库中的输入输出功能，我们在程序的第一行使用了语句`use std::io`。现在，我们将调用`io`模块中的关联函数`stdin`：

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

假如你没有在程序的开始处添加`use std::io;`行，那么就需要将这个函数调用修改为`std::io::stdin`。`stdin`函数会返回类型`std::io::Stdin`的实例，它被用作句柄来处理终端中的标准输入。

这行代码随后的部分，`.read_line(&mut guess)`，调用了标准输入句柄的`read_line`方法来获得用户输入。另外，`read_line`还在调用的过程中使用了一个参数：`&mut guess`。

由于`read_line`方法会将当前用户输入的数据不加区分地存储在字符串中，所以它需要接收一个传入的字符串作为参数。我们传入的变量还需要是可变的，因为这一方法会在记录用户输入的过程中修改字符串。

参数前面的`&`意味着当前的参数是一个引用。你的代码可以通过引用在不同的地方访问同一份数据，而无须付出多余的拷贝开销。不得不说，引用是一个较为复杂的概念，而Rust的核心竞争力之一，就是它保证了我们可以简单并安全地使用引用功能。我们会在第4章深入地讨论这一概念，现在你只需要知道：引用与变量一样，默认情况下也是不可变的。因此，你需要使用`&mut guess`而不是`&guess`来声明一个可变引用。

使用Result类型来处理可能失败的情况

虽然我们讨论完了文本中这一行的所有内容，但当前的语句还没有结束，它仅仅是当前逻辑行中的第一部分，逻辑行中的第二部分调用了下面的方法：

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

你可以在使用`.foo()`语法调用函数时引入换行和缩进来格式化一些较长的代码。我们当然可以将前面的语句写为：

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

但通常而言，过分长的语句会显得有些难以阅读。因此，我们将它链式调用的两个方法拆分为了不同的文本行。现在，让我们来看一看第二部分的方法究竟干了些什么事情。

前面提到过，`read_line`会将用户输入的内容存储到我们传入的字符串中，但与此同时，它还会返回一个`io::Result`值。在Rust标准库中，你可以找到许多以`Result`命名的类型，它们通常是各个子模块中`Result`泛型的特定版本，比如这里的`io::Result`。

`Result`是一个枚举类型。枚举类型由一系列固定的值组合而成，这些值被称作枚举的变体。我们会在第6章详细地讨论枚举类型。

对于`Result`而言，它拥有`Ok`和`Err`两个变体。其中的`Ok`变体表明当前的操作执行成功，并附带代码产生的结果值。相应地，`Err`变体则表明当前的操作执行失败，并附带引发失败的具体原因。

这些Result类型会被用来编码可能出现的错误处理信息。和其他类型的值相同，Result类型的值也定义了一系列的方法，我们刚刚调用的expect就是其中之一。假如io::Result实例的值是Err，那么expect方法就会中断当前的程序，并将传入的字符串参数显示出来。read_line方法有可能因为底层操作系统的错误而返回一个Err结果。相应地，假如io::Result实例的值是Ok，那么expect就会提取出Ok中附带的值，并将它作为结果返回给用户。在我们的例子中，这个值就是用户输入内容的字节数。

即便我们没有在语句末尾调用expect，这段程序也能够编译通过，但你会在编译过程中看到如下所示的警告信息：

```
$ cargo build

Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
--> src/main.rs:10:5
  |
10    |
  io::stdin().read_line(&mut guess);
  |
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |

= note: #[warn(unused_must_use)] on by default
```

Rust编译器借由这段信息提醒我们read_line方法返回的Result值还没有被处理，这通常意味着我们的程序没有对潜在的错误进行处理。

消除警告最正确的方法当然是编写对应的错误处理代码，为了简单起见，我们在这里选择使用expect方法，它会让程序在出现错误时直接终止运行并退出。你可以在第9章学习到有关错误处理的更多内容。

通过println! 中的占位符输出对应的值

现在，除了结尾的花括号，整个函数就只剩最后这一行代码了：

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

它可以将我们存储的用户输入打印出来。这段宏调用的第一个参数是用于格式化的字符串，而字符串中的那对花括号{}则是一个占位符，它用于将后面的参数值插入自己预留的特定位置。你也可以使用花括号来同时打印多个值：第一对花括号对应格式化字符串后的第一个参数，第二对花括号对应格式化字符串后的第二个参数，以此类推。下面的代码展示了如何调用println! 来同时打印多个值：

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

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

一切顺利的话，运行这段代码将会输出x = 5 and y = 10。

尝试运行代码

现在，让我们借助cargo run命令来尝试运行一下这段代码：

```
$ cargo run

Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in
2.53 secs
    Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6

You guessed: 6
```

到目前为止，我们已经完成了猜数游戏的第一部分：可以从用户的键盘获得输入并将它们打印出来。

生成一个保密数字

下一步，我们需要生成一个保密数字来供玩家进行猜测。为了保证一定的可玩性，并使每局游戏都有不同的体验，这个生成的保密数字将会是随机的。为了让游戏不会过分困难，我们可以把这个随机数字限制在1到100之间。Rust团队并没有把类似的随机数字生成功能内置到标准库中，而是选择将它作为rand包（`rand crate`）提供给用户。

借助包来获得更多功能

要记住，Rust中的包（crate）代表了一系列源代码文件的集合。我们当前正在构建的项目是一个用于生成可执行程序的二进制包（binary crate），而我们引用的rand包则是一个用于复用功能的库包（library crate，代码包）。

Cargo最主要的功能就是帮助我们管理和使用第三方库。在使用rand编写代码之前，我们需要修改*Cargo.toml*文件来将rand包声明为依赖。现在让我们打开文件，并在Cargo生成的[dependencies]区域下方添加依赖：

Cargo.toml

```
[dependencies]
rand = "0.3.14"
```

在*Cargo.toml*文件中，从一个标题到下一个标题之间的所有内容都属于同一区域。这里的[dependencies]区域被用来声明项目中需要用到的全部依赖包及其版本号。我们在本例中声明了一个rand包，并将它的版本号指定为0.3.14。Cargo会按照标准的语义化版本系统

(Semantic Versioning, SemVer) 来理解所有的版本号。这里的数字 0.3.14 实际上是 ^0.3.14 的一个简写，它表示“任何与 0.3.14 版本公共 API 相兼容的版本”。

现在先不要修改任何代码，直接重新构建这个项目，如示例 2-2 所示。

```
$ cargo build
```

```
Updating registry
`https://github.com/rust-lang/crates.io-index`
Downloading rand v0.3.14
Downloading libc v0.2.14
Compiling libc v0.2.14
Compiling rand v0.3.14
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
```

示例 2-2：将 rand 包添加为依赖后，运行 cargo build 可能产生的输出

这里显示的编译顺序可能会有所变化，显示的版本号也可能会与我们指定的有所不同，但多亏了 SemVer 的约定，它们会一直与我们的代码保持兼容。

现在，我们的程序有了一个外部依赖，Cargo 可以从注册表（registry）中获取所有可用库的最新版本信息，而这些信息通常是从 crates.io 上复制过来的。在 Rust 的生态中，crates.io 是人们用于分享各种各样开源 Rust 项目的网站。

Cargo 会在更新完注册表后开始逐条检查 [dependencies] 区域中的依赖，并下载当前缺失的依赖包。你可能会注意到，虽然我们只将 rand 引用为依赖，但 Cargo 却额外下载了一份 libc 的数据，这是因为 rand 本身是依赖于 libc 来完成工作的。在下载完所需的包后，Rust 就会开始编译它们，并基于这些依赖编译我们自己的项目。

现在，如果你没有做出任何改变，立即重新运行 cargo build，那么除了 Finished 提示，你应该看不到其他部分。Cargo 会自动分析当前已经下载或编译过的内容，并跳过无须重复的步骤。由于我们既没有修改代码，也没有修改 *Cargo.toml* 文件，所以它不需要重新进行编译，既然无事可做，便随即退出了。

假如你打开 `src/main.rs` 文件，随便做一些无关紧要的修改，保存并再次编译，那么你就可以观察到如下所示的输出结果：

```
$ cargo build

Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
```

这些输出说明Cargo只针对 `src/main.rs` 文件的小修改进行了构建操作。由于我们的依赖没有发生任何变化，所以Cargo自动跳过了下载、编译第三方库的过程，而只重新构建了我们修改过的部分代码。

通过 `Cargo.lock` 文件确保我们的构建是可重现的

Cargo提供了一套机制来确保构建结果是可以重现的，任何人在任何时候重新编译我们的代码都会生成相同的产物：Cargo会一直使用某个特定版本的依赖直到你手动指定了其他版本。打个比方，假如我们使用的rand包在下周发布了0.3.15版本，它修复了一个重要的bug，但这个修复会破坏我们现有的代码，这时，我们重新构建项目会发生什么呢？

回答这个问题的关键就是之前一直被忽略的 `Cargo.lock` 文件，它在我们第一次使用 `cargo build` 时便在当前的项目目录 `guessing_game` 下生成了。当你第一次构建项目时，Cargo会依次遍历我们声明的依赖及其对应的语义化版本，找到符合要求的具体版本号，并将它们写入 `Cargo.lock` 文件中。随后再次构建项目时，Cargo就会优先检索 `Cargo.lock`，假如文件中存在已经指明具体版本的依赖库，那么它就会跳过计算版本号的过程，并直接使用文件中指明的版本。这使得我们拥有了一个自动化的、可重现的构建系统。换句话说，在 `Cargo.lock` 文件的帮助下，当前的项目将会一直使用0.3.14版本的rand包，直到我们手动升级至其他版本。

升级依赖包

当你确实想要升级某个依赖包时，Cargo提供了一个专用命令：`update`，它会强制Cargo忽略 `Cargo.lock` 文件，并重新计算出所有依赖包中符合 `Cargo.toml` 声明的最新版本。假如命令运行成功，Cargo就会将更新后的版本号写入 `Cargo.lock` 文件，并覆盖之前的内容。

基于语义化版本的规则，Cargo在自动升级时只会寻找大于0.3.0并小于0.4.0的最新版本。假如rand包发布了两个新版本：0.3.15和0.4.0，那么当你运行cargo update时，会看到如下所示的输出：

```
$ cargo update

Updating registry `https://github.com/rust-lang/crates.io-index`
Updating rand v0.3.14 -> v0.3.15
```

这时，你也可以在*Cargo.lock*文件中观察到rand包的版本已经被更新为0.3.15了。

假如你想要使用0.4.0或0.4.x系列的版本，那么就必须像下面这样修改*Cargo.toml*文件：

Cargo.toml

```
[dependencies]
rand = "0.4.0"
```

当你下一次运行cargo build时，Cargo就会自动更新注册表中所有可用包的最新版本信息，并根据指定的新版本来重新评估你对rand的需求。

Cargo及其背后的生态系统还有许多可供讨论学习的地方，我们会在第14章深入讨论这些话题，但就目前而言，你已经接触到了够用的基础知识。总的来说，Cargo简化了我们复用代码的诸多流程，以至于Rust的开发者们可以轻松地基于第三方库编写出更为轻巧的项目。

生成一个随机数

现在，你应该已经在*Cargo.toml*中添加了rand包，我们可以正式地在代码中使用它了。接下来，让我们将示例2-3中的代码更新至*src/main.rs*文件中。

```
use std::io;
❶ use rand::Rng;

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

```
②     let secret_number = rand::thread_rng().gen_range(1, 101);

        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);
}
```

示例2-3：添加生成随机数的代码

首先，我们额外增加了一行use语句：use rand::Rng①。这里的Rng是一个trait（特征），它定义了随机数生成器需要实现的方法集合。为了使用这些方法，我们需要显式地将它引入当前的作用域中。第10章会详细介绍有关trait的诸多细节。

另外，我们还在中间新增了两行代码②。第一行中的函数rand::thread_rng会返回一个特定的随机数生成器：它位于本地线程空间，并通过操作系统获得随机数种子。随后，我们调用了这个随机数生成器的方法gen_range。这个方法是在刚刚引入作用域的Rng trait中定义的，它接收两个数字作为参数，并生成一个范围在两者之间的随机数。值得指出的是，它的随机数空间包含下限但不包含上限，所以我们可以指定1和101来获得1到100之间的随机整数。

注意

你当然无法在使用第三方包时凭空知晓自己究竟需要使用什么样的trait或什么样的函数，而是需要在各类包的文档中找到相关的使用说明。值得一提的是，Cargo提供了一个特别有用命令：cargo doc --open，它可以为你在本地构建一份有关所有依赖的文档，并自动地在浏览器中将文档打开来供你查阅。假如你对rand包中的其他功能也颇有兴趣，那么就可以运行cargo doc--open命令，并点击左侧边栏的rand按钮来浏览它的详细文档。

这里添加的第二行代码会将生成的保密数字打印出来，这当然不是游戏的一部分，提前知道了答案哪里还有可玩性。我们只把它作为

开发过程中的调试手段，并会在最终版本中删掉这行代码。

下面我们可以反复尝试运行这段程序：

```
$ cargo run
```

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

```
You guessed: 4
$ cargo run
```

```
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
```

```
You guessed: 5
```

每次运行你都会获得一个不同的随机保密数字，它们会如同我们预料的一样处于1至100的区间内。

比较猜测数字与保密数字

现在，我们有了一个随机生成的保密数字，还有一个用户输入的猜测数字。接下来，在示例2-4中的代码将比较这两个数字。注意，这段用于展示的代码暂时还无法通过编译。

src/main.rs

```
use std::io;
①use std::cmp::Ordering;
use rand::Rng;

fn main() {
    // ---略
    ---

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

    match② guess.cmp(&secret_number)③ {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

示例2-4：比较两个数字并对可能的结果做出响应

这里出现了一行新的use声明①，它从标准库中引入了一个名为std::cmp::Ordering的类型。与Result相同，Ordering也是一个枚举类型，它拥有Less、Greater及Equal这3个变体。它们分别被用来表示比较两个数字之后可能产生的3种结果。

接着，我们在底部增加了5行使用Ordering类型的新代码。其中的cmp方法③能够为任何可比较的值类型计算出它们比较后的结果。本例中的cmp方法接收了被比较值secret_number的引用作为参数来与guess进行比较，它会返回一个我们刚刚引入作用域的Ordering枚举类型的

变体。然后，我们会基于该返回值的具体内容使用match表达式❷来决定下一步执行的代码。

match表达式由数个分支（arm）组成，每个分支都包含一个用于匹配的模式（pattern），以及匹配成功后要执行的相应的代码。Rust会尝试用我们传入match表达式的值去依次匹配每个分支的模式，一旦匹配成功，它就会执行当前分支中的代码。Rust中的match结构及模式是一类非常强大的工具，它们提供了依据不同条件执行不同代码的能力，并能够确保你不会遗漏任何分支条件。我们将在第6章和第18章分别对这两个功能进行详细的介绍。

现在先来简单分析一下这段match表达式的执行过程。假设某次运行生成的随机保密数字是38，而玩家输入了一个猜测数字50。当我们的代码比较50和38时，由于50大于38，所以cmp方法将返回对应的 Ordering:: Greater 变体。随后，match 表达式就以这个变体 Ordering::Greater作为输入，并开始依次去匹配每个分支的模式。第一个分支的模式是Ordering::Less，与当前的输入无法匹配，所以我们会跳过第一个分支及其相应的代码，与下一个分支进行匹配。第二个分支的模式 Ordering::Greater 正好匹配上当前的输入值 Ordering::Greater，当前分支中的代码因此得到执行，进而在屏幕上打印出Too big! 消息。随后，由于已经产生了成功的匹配，所以match表达式也就随之结束了。

上面曾经提到过示例2-4还暂时无法通过编译，那么让我们先试试看：

```
$ cargo build  
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 struct  
`std::string::String`, found integral variable  
|
```

```
= note: expected type `&std::string::String`  
= note:     found type `&{integer}`  
  
error: aborting due to previous error  
Could not compile `guessing_game`.
```

该错误的核心在于示例中的代码存在不匹配的类型。Rust有一个静态强类型系统，同时，它还拥有自动进行类型推导的能力。当我们编写`let guess = String::new()`时，虽然我们没有做出任何显式的声明，但Rust会自动将变量`guess`的类型推导为`String`。另一方面，`secret_number`是一个数值类型。有许多数值类型可以包含从1到100之间的整数，比如`i32`（32位整数）、`u32`（32位无符号整数）、`i64`（64位整数）等。除非我们在代码中增加更多用于推导类型的信息，否则Rust会默认将`secret_number`视作`i32`类型。总而言之，编译器指出的错误就是：Rust无法将字符串类型（`String`类型）和数值类型直接进行对比。

为了正常进行比较操作，我们需要将程序中读取的输入从`String`类型转换为数值类型。这一转换可以通过在`main`函数中增加两行代码来完成：

src/main.rs

```
// --略  
-  
  
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!"),  
}  
}
```

我们在这里创建了一个新的变量`guess`，不过等等，我们不是已经使用过这个名字了吗？没错，但Rust允许使用同名的新变量`guess`来隐藏（shadow）旧变量的值。这一特性通常被用在需要转换值类型的场

景中，它在本例中允许我们重用guess这个变量名，而无须强行创造出guess_str之类的不同名字。我们会在第3章详细介绍隐藏机制。

新创建的guess变量被绑定到了表达式guess.trim().parse()产生的结果上。在这个表达式中，guess指代我们之前通过输入获得的字符串值，而它调用的trim方法则会返回一个去掉了首尾所有空白字符的新字符串实例。之所以需要额外调用trim方法，是因为u32类型只能通过数字字符转换而来，而用户在输入过程中敲击的回车键（Enter键）会导致我们获得的输入字符串额外多出一个换行符。例如，用户在游戏中输入了5并敲击回车键确认，变量guess中存储的字符串将会是5\n。这里的n来自用户敲击的回车键，它是一个换行符，代表“新的一行”，而trim方法则会抹掉n，只留下5在新的字符串中。

最后，字符串的parse方法会尝试将当前的字符串解析为某种数值。由于这个方法可以处理不同的数值类型，所以我们需要通过语句let guesss: u32来显式地声明我们需要的数值类型。guess后面的冒号(:)告诉Rust我们将手动指定当前变量的类型。而这里的u32则是一个32位无符号整型，它是Rust内置的数值类型之一。对于不大的正整数来说，u32已经完全可以满足需求了，我们会在第3章介绍其他可供选择的数值类型。值得指出的是，由于我们将guess手动标记为了u32，并且将它和secret_number进行了比较，所以Rust会将secret_number也推导为相同的u32类型。现在，我们终于可以比较两个相同类型的值了。

一般来说，调用parse非常容易产生错误。假如用户输入的字符串中包含A或%，那么它便无法转换为一个正常的数字。类似于本章“使用Result类型来处理可能失败的情况”一节中提到过的read_line，正是因为parse方法存在失败的可能性，所以它会返回一个Result类型的值。这里，我们依然简单地使用expect方法即可。假如parse无法将字符串转换为一个数字，并返回了Result的Err变体，那么expect就会使游戏崩溃退出并打印出我们设定的提示信息。假如parse成功地将字符串转换为了数字，并返回了Result的Ok变体，那么expect就会直接返回Ok中附带的数值。

现在让我们运行程序试试：

```
$ cargo run
```

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

0.43 secs
    Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76

You guessed: 76
Too big!
```

真棒！尽管我们在输入猜测数字时额外地添加了几个空白键，但程序依然正确地识别出了用户的输入是76。你可以尝试反复运行这段程序来检验它在不同输入条件下的不同行为，分别观察用户在猜测一个正确的数字、一个偏大的数字和一个偏小的数字时程序会产生什么样的输出。

这个游戏已经大体成型了，但玩家只能做出一次猜测，这显然是不够的。接下来，我们会加入一个循环来完善这个游戏。

使用循环来实现多次猜测

在Rust中，loop关键字会创建一个无限循环。我们可以将它加入当前的程序中，进而允许玩家反复地进行猜测抉择：

```
src/main.rs
// --略

-- 

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

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

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

正如你看到的一样，我们将提示用户做出猜测决定之后的所有内容都移动到了loop中。当然，出于美观的考虑，循环表达式中的代码都额外缩进了4个字符。再次运行这段代码，你应该可以看到程序忠实地执行了我们的要求：无限地反复请求用户做出猜测抉择。这可不太对，玩家永远都没办法正常地结束游戏了！

当然，用户总是可以通过诸如Ctrl+C之类的快捷键强制终止程序。另外，正如我们在本章的“比较猜测数字与保密数字”一节中提到过的，用户可以输入一段非法的字符串来触发parse转换错误，并最终导致程序崩溃退出，如下所示：

```
$ cargo run
```

```

Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
    Running `target/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 'Please type a number!: ParseIntError { kind: InvalidDigit
}', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for

a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)

```

输入**quit**确实让我们退出了游戏，当然，你输入其他任何无法转换为数字的字符串都会得到类似的结果。但这并不是我们想要的行为，我们希望游戏能够在玩家正确地猜出保密数字时优雅地退出。

在猜测成功时优雅地退出

现在让我们给程序增加一条break语句，使得玩家在猜对数字后能够正常退出游戏。

src/main.rs

```
// --略--

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

```
        break;
    }
}
}
```

输出You win! 后添加的break语句会让程序在玩家猜对保密数字后退出当前循环。由于程序中的循环是main函数的最后一部分代码，所以退出循环也就意味着退出程序。

处理非法输入

为了进一步改善游戏的可玩性，我们可以在用户输入了一个非数字数据时简单地忽略这次猜测行为，并使用户可以继续进行猜测，从而避免程序发生崩溃。还记得我们把guess变量从String类型转换为u32类型的语句吗？我们可以将它改进为示例2-5中的形式：

```
src/main.rs

// --略

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);

// --略

--
```

示例2-5：忽略非数字数据并再次请求玩家猜数，从而避免程序发生崩溃

我们使用了match表达式来替换之前的expect方法，这是我们处理错误行为的一种惯用手段。如之前所提到过的，parse会返回一个Result类型，而Result则包含了Ok与Err两个变体。这里使用match表达式，就像之前处理cmp方法的返回值Ordering一样。

假如parse成功地将字符串解析为数字，那么它将返回一个包含了该数字的Ok值。这个Ok值会匹配到match表达式的第一分支模式，而match表达式则会返回parse产生的放置在Ok中的num值。最终这个数字会被绑定到我们创建的guess变量中。

假如parse没能将字符串解析为数字，那么它将返回一个包含了具体错误信息的Err值。这个值会因为无法匹配Ok(num)模式而跳过match表达式的第一分支，并匹配上第二个分支中的Err(_)模式。这里的下划线_是一个通配符，它可以在本例中匹配所有可能的Err值，而不管其中究竟有何种错误信息。因此，程序会继续执行第二个分支中的代码：continue，这条语句会使程序直接跳转至下一次循环，并再次请求玩家猜测保密数字。这样，程序便忽略了parse可能会触发的那些错误。

万事俱备，让我们运行这个项目试试看：

```
$ cargo run
```

```
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Running `target/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!
```

真棒！只剩下最后一小处修改，我们就能完成这个猜数游戏了。还记得程序会在最开始的时候将保密数字打印出来吗？这种行为会在正式发布时毁掉我们的游戏，它只能被用于测试。现在，让我们删除

可以输出保密数字的println! 语句，最终的完整代码将如示例2-6所示。

src/main.rs

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

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

    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;
            }
        }
    }
}
```

示例2-6：完整的猜数游戏代码

总结

恭喜你！到这里为止，你已经成功完成了猜数游戏！

这是一个用来向你介绍Rust相关概念的实践项目。我们在本章接触到了let、match、方法、关联函数，以及外部包的使用等不同概念。我们将在接下来的章节中针对这些概念进行更为细致的讨论。第3章会涉及一些常见的编程语言概念，例如变量、数据类型、函数等，当然，你也会学习到如何在Rust中使用它们。第4章会深入浅出地向你介绍Rust区别于其他语言的一个重要特性：所有权。第5章会讨论结构体和方法。第6章会向你介绍枚举类型的工作机制。

第3章

通用编程概念



本章将会介绍一些编程领域中常见的概念，以及它们在Rust中的实现方式。许多语言都有着类似的核心特性。本章涉及的所有概念都不是Rust所独有的，但我们在Rust的上下文环境中讨论它们，并演示这些概念常见的使用方式。

更具体一些来讲，你可以在本章学到关于变量、基本类型、函数、注释和控制流等概念。这些基础概念几乎会出现在每一个Rust程序中，尽早地了解它们可以为你学习编程语言打下坚实的基础。

关键字

与其他编程语言类似，Rust也拥有一系列只能被用于语言本身的保留关键字。要记住，你不能使用这些关键字来命名自定义的变量或函数。大部分关键字都有特殊的意义，你会使用它们来完成Rust程序中各式各样的任务；还有一些关键字目前没有任何功能，

但它们被预留给了未来可能会添加的功能来使用。你可以在附录A中看到一份关键字的详细列表。

变量与可变性

正如第2章所提到过的，Rust中的变量默认是不可变的。Rust语言提供这一概念是为了能够让你安全且方便地写出复杂、甚至是并行的代码。当然，Rust也提供了让你可以使用可变变量的方法，我们会在这一节中讨论有关可变性的设计取舍。

当一个变量是不可变的时，一旦它被绑定到某个值上面，这个值就再也无法被改变。为了演示这一点，让我们在*projects* 目录下使用命令`cargo new variables`来新建一个叫作*variables* 的项目。

现在，打开新项目*variables* 中的*src/main.rs* 文件，并用下面的代码替换原来的代码（注意，这段代码还无法通过编译）：

src/main.rs

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

保存并通过命令`cargo run`来运行这段程序，你将看到如下所示的错误提示信息：

```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
 |
2 | 
3 |     let x = 5;
 |     |
 |         - first assignment to `x`
```

```
    println!("The value of x is: {}", x);
4 |
    x = 6;
|
^^^^^ cannot assign twice to immutable variable
```

这个示例同时也展示了编译器会如何帮助你定位程序中的错误。没错，在编写Rust程序的过程中，编译器的错误提示信息可能会让人感到有些沮丧，但这却并不能说明你是一个失败的程序员！即便是经验丰富的Rust程序员，也需要通过编译器的错误提示信息来确保程序能够按照自己的指示、安全地执行任务。

这里的错误提示信息提示我们 `cannot assign twice to immutable variable`（不能对不可变变量进行二次赋值），因为我们 在第4行尝试了对不可变变量x进行二次赋值。

编译时的错误提示信息可以帮助我们避免修改一个不可变变量。这种情形非常容易导致一些难以察觉的bug，因为我们的代码逻辑可能会依赖于绑定在这个变量上的不可变的值，所以一旦这个值发生变化，程序就无法继续按照期望的方式运行下去。这种bug往往难以追踪，特别是修改操作只在某些条件下偶然发生的时候。在类似的情形下，编译时的错误提示信息就显得相当重要了。

Rust的编译器能够保证那些声明为不可变的值一定不会发生改变。这也意味着你无须在阅读和编写代码时追踪一个变量会如何变化，从而使代码逻辑更加易于理解和推导。

不过，可变性也同样非常有用。如同第2章所介绍的那样，变量默认是不可变的，但你可以通过在声明的变量名称前添加`mut`关键字来使其可变。除了使变量的值可变，`mut`还会向阅读代码的人暗示其他代码可能会改变这个变量的值。

例如，我们可以将 `src/main.rs` 修改为如下所示的样子：

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);
}
```

运行上面的程序，你会看到如下所示的输出结果：

```
$ cargo run

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

正是因为`mut`出现在了变量绑定的过程中，所以我们现在可以合法地将`x`绑定的值从5修改为6了。在某些情况下，相较于不可变变量而言，可变变量会让代码变得更加易于编写。

除了避免出现bug，设计一个变量的可变性还需要考量许多因素。例如当你在使用某些重型数据结构时，适当地使用可变性去修改一个实例，可能比赋值和重新返回一个新分配的实例要更有效率；而当数据结构较为轻量的时候，采用更偏向函数式的风格，通过创建新变量来进行赋值，可能会使代码更加易于理解。在类似这样的情形下，为了可读性而损失少许的性能也许是值得的。

变量与常量之间的不同

变量的不可变性可能会让你联想到另外一个常见的编程概念：常量（constant）。就像不可变变量一样，绑定到常量上的值无法被其他代码修改，但常量和变量之间还是存在着一些细微的差别的。

首先，我们不能用`mut`关键字来修饰一个常量。常量不仅是默认不可变的，它还总是不可变的。

其次，你需要使用`const`关键字而不是`let`关键字来声明一个常量。在声明的同时，你必须 显式地标注值的类型。我们将在接下来的“数据类型”一节中接触到类型及类型标注，你可以先暂时不用理会类型标注的具体含义，只要在脑海中记住常量总是需要标注类型即可。

再次，常量可以被声明在任何作用域中，甚至包括全局作用域。这在一个值需要被不同部分的代码共同引用时十分有用。

最后，你只能将常量绑定到一个常量表达式上，而无法将一个函数的返回值，或其他需要在运行时计算的值绑定到常量上。

下面是一个常量声明的例子，数值100 000被绑定到了常量MAX_POINTS上。在Rust程序中，我们约定俗成地使用以下画线分隔的全大写字母来命名一个常量，并在数值中插入下画线来提高可读性。

```
const MAX_POINTS: u32 = 100_000;
```

常量在整个程序运行的过程中都在自己声明的作用域内有效，这使得常量可以被用于在程序的不同代码之间共享值，例如一个游戏中所有玩家可以获取的最高分数，或者光速之类的东西。

将整个程序中硬编码的值声明为常量并给予其有意义的名字，可以帮助后来的维护者去理解这些值的意义，而使用同一常量来索引相同的硬编码值也能为将来的修改提供方便。

隐藏

在第2章的“比较猜测数字与保密数字”一节中，我们曾经看到一个新声明的变量可以覆盖掉旧的同名变量。在Rust世界中，我们把这一现象描述为：第一个变量被第二个变量隐藏（shadow）了。这意味着我们随后使用这个名称时，它指向的将会是第二个变量。我们可以重复使用let关键字并配以相同的名称来不断地隐藏变量：

src/main.rs

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

    let x = x + 1;

    let x = x * 2;

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

这段程序首先将x绑定到值5上。随后它又通过重复let x =语句隐藏了第一个x变量，并将第一个x变量值加上1的运算结果绑定到新的变量x上，这时x的值是6。第三个let语句同样隐藏了第二个x变量，并将第二个x变量值乘以2的结果12绑定到第三个x变量上。你可以在运行这段程序后看到如下所示的结果：

```
$ cargo run

Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in
0.31 secs
Running `target/debug/variables`
The value of x is: 12
```

隐藏机制不同于将一个变量声明为mut，因为如果不是在使用let关键字的情况下重新为这个变量赋值，则会导致编译错误。通过使用let，我们可以对这个值执行一系列的变换操作，并允许这个变量在操作完成后保持自己的不可变性。

隐藏机制与mut的另一个区别在于：由于重复使用let关键字会创建出新的变量，所以我们可以复用变量名称的同时改变它的类型。例如，假设程序需要根据用户输入的空格数量来决定文本之间的距离，那么我们可能会把输入的空格存储为一个独立的数值：

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

这段代码之所以能够生效，是因为声明的第一个变量spaces是字符串类型，而第二个spaces变量虽然拥有与第一个变量相同的名称，但它却是一个崭新的数值变量。隐藏机制允许我们复用spaces这个简单的名字，而不需要做出诸如spaces_str和spaces_num之类的区分。然而，尝试使用mut来模拟类似的效果（如下所示）就会在编译时报错：

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

编译器会拒绝我们修改变量的类型：

```
error[E0308]: mismatched types
--> src/main.rs:3:14
|
```

3 |

```
| spaces = spaces.len();  
|  
|     ^^^^^^^^^^^^ expected &str, found usize  
|  
|= note: expected type `&str`  
  found type `usize`
```

现在你应该已经清楚了变量的工作方式，让我们接着来看那些变量可以使用的常见数据类型。

数据类型

Rust中的每一个值都有其特定的数据类型，Rust会根据数据的类型来决定应该如何处理它们。我们会讨论两种不同的数据类型子集：标量类型（scalar）和复合类型（compound）。

要记住，Rust是一门静态类型语言，这意味着它在编译程序的过程中需要知道所有变量的具体类型。在大部分情况下，编译器都可以根据我们如何绑定、使用变量的值来自动推导出变量的类型。但在某些时候，比如在第2章的“比较猜测数字与保密数字”一节中，当我们需要使用parse将一个String类型转换为数值类型时，就必须像下面这样显式地添加一个类型标注：

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

假如我们移除这里的类型标注，Rust就会在编译的过程中输出如下所示的错误提示信息：

```
error[E0282]: type annotations needed
--> src/main.rs:2:9
 |
2 | let guess = "42".parse().expect("Not a number!");
|           ^^^^^^
|
|           |
|
|           cannot infer type for `guess`  
consider giving `guess` a type
```

这段信息表明当前的编译器无法自动推导出变量的类型，为了避免混淆，它需要我们手动地添加类型标注。接下来，你会看到不同类型的数据类型的类型标注方式。

标量类型

标量 类型是单个值类型的统称。Rust中内建了4种基础的标量类型：整数、浮点数、布尔值及字符。你应该在其他语言中也接触过类似的基础类型，让我们来看一看这些标量类型是如何在Rust中工作的吧。

整数类型

整数 是指那些没有小数部分的数字。我们曾经在第2章使用过一个叫作u32的整数类型，这个类型表明它关联的值是一个无符号的32位整数（有符号整数类型的名称会以i代替u来开头）。表3-1展示了Rust中内建的那些整数类型，每一个长度不同的值都存在有符号和无符号两种变体，它们可以被用来描述不同类型的整数。

表3-1 Rust中的整数类型

长度	有符号	无符号
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
arch	isize	usize

每一个整数类型的变体都会标明自身是否存在符号，并且拥有一个明确的大小。有符号 和无符号 代表了一个整数类型是否拥有描述负数的能力。换句话说，对于有符号的整数类型来讲，数值需要一个符号来表示当前是否为正，而对于无符号的整数类型来讲，数值永远为正，不需要符号。这与在纸上书写数字类似：当数字需要考虑正负性的时候，我们会使用一个加号或减号作为前缀进行标识；而当数字

可以被安全地视作永远为正数时，就不需要使用加号作为前缀了。有符号数是通过二进制补码的形式来存储的。

对于一个位数为 n 的有符号整数类型，它可以存储从 $-(2^{n-1})$ 到 $2^{n-1}-1$ 范围内的所有整数。比如对于i8来讲，它可以存储从 $-(2^7)$ 到 2^7-1 ，也就是从-128到127之间的所有整数。而对于无符号整数类型而言，则可以存储从0到 2^n-1 范围内的所有整数。以u8为例，它可以存储从0到 2^8-1 ，也就是从0到255之间的所有整数。

除了指明位数的类型，还有 isize和 usize两种特殊的整数类型，它们的长度取决于程序运行的目标平台。在64位架构上，它们就是64位的，而在32位架构上，它们就是32位的。

你可以使用表3-2中列出的所有方式在代码中书写整数字面量。注意，除了Byte，其余所有的字面量都可以使用类型后缀，比如57u8，代表一个使用了u8类型的整数57。同时你也可以使用_作为分隔符以方便读数，比如1_000。

表3-2 Rust中的整数字面量

整数字面量	示例
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

在这么多的整数类型中，你怎么确定自己需要使用哪一种呢？如果拿不定主意，Rust对于整数字面量的默认推导类型i32通常就是一个很好的选择：它在大部分情形下都是运算速度最快的那一个，即便是在64位系统上也是如此。较为特殊的两个整数类型usize和isize则主要用作某些集合的索引。

整数溢出

假设你有一个u8类型的变量，它可以存储从0到255的数字。当你尝试将该变量修改为某个超出范围的值（比如256）时，就会发生整数溢出。Rust在这一行为中拥有某些有趣的规则。如果你在调试(debug)模式下进行编译，那么Rust就会在程序中包含整数溢出的运行时检测代码，并在整数溢出发生时触发程序的panic。Rust使用术语panic来描述程序因为错误而退出的情形；我们会在第9章的“不可恢复错误与panic!”一节中讨论更多有关panic的内容。

如果你在编译时使用了带有--release标记的发布(release)模式，那么Rust就不会包含那些可能会触发panic的检查代码。作为替代，Rust会在溢出发生时执行二进制补码环绕。简而言之，任何超出类型最大值的数值都会被“环绕”为类型最小值。以u8为例，256会变为0，257会变为1，以此类推。虽然程序不会发生panic，但变量中实际存储的值也许会让你大吃一惊。那些依赖于整数溢出时环绕行为的代码应该被视作错误代码。假如你确实希望显式地进行环绕行为，那么你可以使用标准库中的类型Wrapping。

浮点数类型

除了整数，Rust还提供了两种基础的浮点数类型，浮点数也就是带小数的数字。这两种类型是f32和f64，它们分别占用32位和64位空间。由于在现代CPU中f64与f32的运行效率相差无几，却拥有更高的精度，所以在Rust中，默认会将浮点数字面量的类型推导为f64。

下面展示了实际代码中的浮点数声明：

src/main.rs

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

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

Rust的浮点数使用了IEEE-754标准来进行表述，f32和f64类型分别对应着标准中的单精度浮点数和双精度浮点数。

数值运算

对于所有的数值类型，Rust都支持常见的数学运算：加法、减法、乘法、除法及取余。下面的代码展示了如何在let语句中使用这些运算进行求值：

src/main.rs

```
fn main() {
    // 加法

    let sum = 5 + 10;

    // 减法

    let difference = 95.5 - 4.3;

    // 乘法

    let product = 4 * 30;

    // 除法

    let quotient = 56.7 / 32.2;

    // 取余

    let remainder = 43 % 5;
}
```

这些语句中的每条表达式都使用了一个数学运算符，并将运算产生的结果绑定到了左侧的变量上。附录B中有一份列表，完整地包含了Rust支持的所有运算符。

布尔类型

正如其他大部分编程语言一样，Rust的布尔类型只拥有两个可能的值：true和false，它会占据单个字节的空间大小。你可以使用bool来表示一个布尔类型，例如：

src/main.rs

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

    let f: bool = false; // 附带了显式类型标注的语句
```

}

布尔类型最主要的用途是在if表达式内作为条件使用，我们将在本章后面的“控制流”一节中详细介绍Rust的if表达式是如何工作的。

字符类型

到目前为止，我们接触到的大部分类型都只与数字有关，但Rust也同样提供了相应的字符类型支持。在Rust中，char类型被用于描述语言中最基础的单个字符。下面的代码展示了它的使用方式，但需要注意的是，char类型使用单引号指定，而不同于字符串使用双引号指定。

src/main.rs

```
fn main() {
    let c = 'z';
    let z = 'ℤ
';
    let heart_eyed_cat = '😻
';
}
```

Rust中的char类型占4字节，是一个Unicode标量值，这也意味着它可以表示比ASCII多得多的字符内容。拼音字母、中文、日文、韩文、零长度空白字符，甚至是emoji表情都可以作为一个有效的char类型值。实际上，Unicode标量可以描述从U+0000到U+D7FF以及从U+E000到U+10FFFF范围内的所有值。由于Unicode中没有“字符”的概念，所以你现在从直觉上认为的“字符”也许与Rust中的概念并不相符。我们将在第8章的“使用字符串存储UTF-8编码的文本”一节详细地讨论这个主题。

复合类型

复合类型（compound type）可以将多个不同类型的值组合为一个类型。Rust提供了两种内置的基础复合类型：元组（tuple）和数组（array）。

元组类型

元组是一种相当常见的复合类型，它可以将其他不同类型的多个值组合进一个复合类型中。元组还拥有一个固定的长度：你无法在声明结束后增加或减少其中的元素数量。

为了创建元组，我们需要把一系列的值使用逗号分隔后放置到一对圆括号中。元组每个位置的值都有一个类型，这些类型不需要是相同的。为了演示，下面的例子中手动添加了不必要的类型注解：

src/main.rs

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

由于一个元组也被视作一个单独的复合元素，所以这里的变量tup被绑定到了整个元组上。为了从元组中获得单个的值，我们可以像下面这样使用模式匹配来解构元组：

src/main.rs

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

    let (x, y, z) = tup;

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

这段程序首先创建了一个元组，并将其绑定到了变量tup上。随后，let关键字的右侧使用了一个模式将tup拆分为3个不同的部分：x、y和z，这个操作也被称为解构（destructuring）。最后，程序将变量y的值，也就是6.4打印了出来。

除了解构，我们还可以通过索引并使用点号（.）来访问元组中的值：

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;  
}
```

这段程序首先创建了一个元组x，随后又通过索引访问元组的各个元素，并将它们的值绑定到新的变量上。和大多数编程语言一样，元组的索引从0开始。

数组类型

我们同样可以在数组中存储多个值的集合。与元组不同，数组中的每一个元素都必须是相同的类型。Rust中的数组拥有固定的长度，一旦声明就再也不能随意更改大小，这与其他某些语言有所不同。

在Rust中，你可以将以逗号分隔的值放置在一对方括号内来创建一个数组：

src/main.rs

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

通常而言，当你想在栈上而不是堆上为数据分配空间时，或者想要确保总有固定数量的元素时，数组是一个非常有用的工具（如果你还不太清楚栈和堆的区别，那么请不要着急，我们会在第4章详细地讨论它们）。当然，Rust标准库也提供了一个更加灵活的动态数组（vector）类型。动态数组是一个类似于数组的集合结构，但它允许用户自由地调整数组长度。假如你还不确定什么时候应该使用数组，什么时候应该使用动态数组，那就先使用动态数组好了。在第8章会深入讨论有关动态数组的更多细节。

在下面这种情形中，你也许会选择使用数组而非动态数组。假设在某个程序中需要知道一年中每个月份的名字，我们就可以使用数组来存储这个名字列表。因为我们知道它有且仅有12个元素，且不太可能添加或删除月份。

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

为了写出数组的类型，你需要使用一对方括号，并在方括号中填写数组内所有元素的类型、一个分号及数组内元素的数量，如下所示：

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

示例中的i32便是数组内所有元素的类型，而分号之后的5则表明当前的数组包含5个元素。

这样撰写数组类型的方式有些类似于另一种初始化数组的语法，即假如你想要创建一个含有相同元素的数组，那么你可以在方括号中指定元素的值，并接着填入一个分号及数组的长度，如下所示：

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

以a命名的数组将会拥有5个元素，而这些元素全部拥有相同的初始值3。这一写法等价于`let a = [3, 3, 3, 3, 3];`，但却更加精简。

访问数组的元素

数组由一整块分配在栈上的内存组成，你可以通过索引来访问一个数组中的所有元素，就像下面演示的一样：

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，first变量会被赋值为1，这正是数组中索引[0]对应的那个值。同样，second变量将获得数组中索引[1]对应的那个值，也就是2。

非法的数组元素访问

尝试访问数组结尾之后的元素会发生些什么呢？我们可以将例子修改为下面的样子：

src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

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

使用cargo run运行这段代码，我们会发现程序顺利通过了编译，却会在运行时因为错误而崩溃退出：

```
$ cargo run
```

```
Compiling arrays v0.1.0 (file:///projects/arrays)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/arrays`
thread '' panicked at 'index out of bounds: the len is 5 but the index is
10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

编译并没有产生任何错误提示，但是程序却因为一个运行时 错误而不正确地终止了运行。实际上，每次通过索引来访问一个元素时，Rust都会检查这个索引是否小于当前数组的长度。假如索引超出了当前数组的长度，Rust就会发生panic。

这应该是我们遇到的第一个涉及Rust安全原则的示例。有许多底层语言没有提供类似的检查，一旦尝试使用非法索引，你就会访问到某块无效的内存。在这种情况下，逻辑上的错误常常会蔓延至程序的其他部分，进而产生无法预料的结果。通过立即中断程序而不是自作主张地去继续运行，Rust帮助我们避开了此类错误。你可以在第9章接触到更多的错误处理机制。

函数

函数在Rust中有着非常广泛的应用。你应该已经见过Rust中最为重要的main函数了，它是大部分程序开始的地方。你应该也对fn关键字有印象，我们可以用它来声明一个新的函数。

Rust代码使用蛇形命名法（snake case）来作为规范函数和变量名称的风格。蛇形命名法只使用小写的字母进行命名，并以下划线分隔单词。下面就是一个包含函数定义的示例：

src/main.rs

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

    another_function();
}

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

在Rust中，函数定义以fn关键字开始并紧随函数名称与一对圆括号，另外还有一对花括号用于标识函数体开始和结尾的地方。

我们可以使用函数名加圆括号的方式来调用函数。在上面的示例中，由于another_function被定义为了函数，所以我们在main函数体内调用它。需要注意的是，我们在这个例子中将another_function函数定义在了main函数之后，但把它放到main函数之前其实也没有什么影响。Rust不关心你在何处定义函数，只要这些定义对于使用区域是可见的即可。

现在，让我们创建一个新的二进制项目*functions* 来实践一下函数的相关功能。将上面another_function示例中的内容复制到文件src/main.rs中并运行它，你可以观察到如下所示的输出结果：

```
$ cargo run

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

正如我们预料的那样，代码以它们出现在main函数中的顺序依次执行了出来。首先，“Hello, world!”这条信息被打印出来，紧接着another_function函数得到执行，其函数体内的信息也被打印出来。

函数参数

你也可以在函数声明中定义参数（parameter），它们是一种特殊的变量，并被视作函数签名的一部分。当函数存在参数时，你需要在调用函数时为这些变量提供具体的值。在英语技术文档中，参数变量和传入的具体参数值有自己分别对应的名称*parameter* 和 *argument*，但我们通常会混用两者并将它们统一地称为参数而不加以区别。

下面重写后的another_function函数展示了Rust中参数的样子：

src/main.rs

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

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

尝试运行这段程序可以得到如下所示的输出：

```
$ cargo run

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

这里定义的another_function有一个名为x且类型为i32的参数。当5被传入another_function时，println! 宏会将5放入格式化字符串中的特定位置并打印出来。

在函数签名中，你必须 显式地声明每个参数的类型。这是在Rust设计中设计者们经过慎重考虑后做出的决定：由于类型被显式地注明了，因此编译器不需要通过其他部分的代码进行推导就能明确地知道你的意图。

另外，你可以像下面一样，通过使用逗号分隔符来为函数声明多个参数：

src/main.rs

```
fn main() {
    another_function(5, 6);
}

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

这里的示例创建了一个拥有两个参数的函数，这个函数会依次打印出这两个参数。需要注意的是，函数参数可以是不同类型的，本例中只是恰好使用了两个i32类型的参数而已。

让我们用这段代码替换*functions*项目的src/main.rs文件中的内容，并使用cargo run来运行试试看：

```
$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in
0.31 secs
    Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

由于我们在调用函数时，将5和6分别作为x和y的值传入了函数，所以这两个字符串与它们的值被相应地打印了出来。

函数体中的语句和表达式

函数体由若干条语句组成，并可以以一个表达式作为结尾。虽然我们在语句中见到了许多表达式，但到目前为止，我们都还没有使用过表达式来结束一个函数。由于Rust是一门基于表达式的语言，所以它将语句（statement）与表达式（expression）区别为两个不同的概念，这与其他某些语言不同。因此，让我们首先来看一看语句和表达式究竟是什么，接着再进一步讨论它们之间的区别会如何影响函数体的定义过程。

虽然之前没有明确地说明过，但我们在示例中已经使用过很多次语句和表达式了。语句指那些执行操作但不返回值的指令，而表达式则是指会进行计算并产生一个值作为结果的指令。这么说起来可能会有些抽象，让我们结合例子来看一看它们之间的区别。

使用`let`关键字创建变量并绑定值时使用的指令是一条语句。在示例3-1中，`let y = 6;` 就是一条语句。

`src/main.rs`

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

示例3-1：包含一条语句的`main`函数

这里的函数定义同样是语句，甚至上面整个例子本身也是一条语句。

要记住，语句不会返回值。因此，在Rust中，你不能将一条`let`语句赋值给另一个变量，如下所示的代码会产生编译时错误：

`src/main.rs`

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

尝试运行上面这段程序将产生如下所示的错误提示信息：

```

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
--> src/main.rs:2:14
 |
2 |
    let x = (let y = 6);
 |
    ^^^
 |
= note: variable declaration using `let` is a statement

```

由于`let y = 6`语句没有返回任何值，所以变量`x`就没有可以绑定的东西。这里的行为与某些语言不同，例如C语言或Ruby语言中的赋值语句会返回所赋的值。在这些语言中，你可以编写类似于`x = y = 6`这样的语句，并使得`x`和`y`变量同时拥有6这个值，但这在Rust中可行不通。

与语句不同，表达式会计算出某个值来作为结果。你在Rust中编写的大部分代码都会是表达式。以简单的数学运算`5 + 6`为例，这就是一个表达式，并且会计算出值11。另外，表达式本身也可以作为语句的一部分。在示例3-1中，语句`let y = 6;`中的字面量6就是一个表达式，它返回6作为自己的计算结果。调用函数是表达式，调用宏是表达式，我们用来创建新作用域的花括号（`{}`）同样也是表达式，例如：

src/main.rs

```

fn main() {
    let x = 5;

① let y = {②
    let x = 3;
    ③ x + 1
};

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

```

表达式②是一个代码块。在这个例子中，它会计算出4作为结果。而这个结果会作为`let`语句①的一部分被绑定到变量`y`上。注意结尾处③的表达式`x + 1`没有添加分号，这与我们之前见过的大部分代码不

同。假如我们在表达式的末尾加上了分号，这一段代码就变为了语句而不会返回任何值。记住这一点，你会在接下来的章节中用到相关内容。

函数的返回值

函数可以向调用它的代码返回值。虽然你不用为这个返回值命名，但需要在箭头符号(\rightarrow)的后面声明它的类型。在Rust中，函数的返回值等同于函数体最后一个表达式的值。你可以使用return关键字并指定一个值来提前从函数中返回，但大多数函数都隐式地返回了最后的表达式。下面是一个带有返回值的函数示例：

src/main.rs

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

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

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

在以上的five函数中，除了数字5，没有任何其他的函数调用、宏调用，甚至是let语句，但它在Rust中确实是一个有效的函数。注意，这个函数的返回值类型通过 \rightarrow i32被指定了。尝试运行这段代码，你会看到如下所示的输出：

\$ cargo run

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

five函数中的5就是函数的输出值，这也是它的返回类型会被声明为i32的原因。这段代码中有两处需要注意的地方。首先，语句let x = five(); 使用函数的返回值来初始化左侧的变量。由于five函数总是返回5，所以该行代码等价于：

```
let x = 5;
```

其次，这里的five函数没有参数，仅仅定义了返回值的类型。函数体中除了孤零零的、不带分号的5，没有任何东西，而它也正是我们想要用来作为结果返回的表达式。

再来看另外一个例子：

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
}
```

运行这段代码会输出The value of x is: 6。假如我们给函数plus_one结尾处的x + 1加上分号（如下所示），那么这个表达式就会变为语句并进而导致编译时错误。

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;
}
```

尝试编译这段代码会产生如下所示的错误提示信息：

```
error[E0308]: mismatched types
--> src/main.rs:7:28
  |
7 |     fn plus_one(x: i32) -> i32 {
  |     ^
  |     |
8 |     |     x + 1;
  |     |     - help: consider removing this semicolon
  |     |
9 |     }
  |     ^ expected i32, found ()
  |
= note: expected type `i32`
        found type `()`
```

这里的错误提示信息“mismatched types”（类型不匹配）揭示了上面代码中的核心问题。我们在定义plus_one的过程中声明它会返回一个i32类型的值，但由于语句并不会产生值，所以Rust默认返回了一个空元组，也就是上面描述中的()。实际的返回值类型与函数定义产生了矛盾，进而触发了编译时错误。另外，Rust编译器在错误提示信息中还提示了一个修正错误的可能方案：它建议我们尝试去掉函数末尾的分号来解决这个问题。

注释

没错，所有的程序员都应该致力于让自己的代码通俗易懂，但有些时候，额外的说明也是必不可少的。在这些情形下，程序员可以在源代码中留下一些记录，或者说是注释（comment）。虽然编译器会忽略掉这些注释，但其他阅读代码的人也许会因为它们而能够更加轻松地理解你的意图。

这里是一个简单的例子：

```
// Hello, world.
```

在Rust中，注释必须使用两道斜杠开始，并持续到本行结尾。对于那些超过一行的注释，你需要像下面这样在每一行前面都加上//。

```
// 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.
```

注释也可以被放置在代码行的结尾处：

src/main.rs

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

不过你可能会更常见到下面这种格式，在需要说明的代码上方单独放置一行注释：

src/main.rs

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

Rust中还有一种被称为文档注释的注释格式，我们将在第14章介绍它。

控制流

通过条件来执行或重复执行某些代码是大部分编程语言的基础组成部分。在Rust中用来控制程序执行流的结构主要就是if表达式与循环表达式。

if表达式

if表达式允许我们根据条件执行不同的代码分支。我们提供一个条件，并且做出声明：“假如这个条件满足，则运行这段代码。假如条件没有被满足，则跳过相应的代码。”

现在，让我们在*projects* 目录下创建一个新的*branches* 项目来学习与if表达式相关的知识。打开*src/main.rs* 文件，并输入如下所示的内容：

src/main.rs

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

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

所有的if表达式都会使用if关键字来开头，并紧随一个判断条件。在上面的例子中，我们的条件会检查变量number绑定的值是否小于5。其后的花括号中放置了条件为真时需要执行的代码片段。if表达式中与条件相关联的代码块也被称作分支（arm），就和我们在第2章的“比较猜测数字与保密数字”一节中接触到的match表达式一样。

示例中的代码还为if表达式添加了一个可选的else表达式，你可以用它来指定一段条件为假时希望执行的代码块。假如我们没有提供else表达式，且条件被判定为假，那么程序会简单地跳过if表达式并继续执行之后的代码。

尝试运行这段代码，你会看到如下所示的输出提示信息：

```
$ cargo run
```

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

让我们尝试修改number的值，使判断条件变为false，并观察随后发生的情形：

```
let number = 7;
```

再次运行这个程序，输出如下：

```
$ cargo run
```

```
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/branches`
condition was false
```

值得注意的是，代码中的条件表达式必须产生一个bool类型的值，否则就会触发编译错误。例如，尝试运行下面的代码：

src/main.rs

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

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

这一次，if的条件表达式计算结果为3，因而Rust在编译过程中抛出了如下所示的错误：

```
error[E0308]: mismatched types
--> src/main.rs:4:8
|
```

```
4 |  
|   if number {  
|  
|     ^^^^^^ expected bool, found integral variable  
|  
|= note: expected type `bool`  
|       found type `'{integer}`
```

这个错误表明Rust期望在条件表达式中获得一个bool值，而不是一个整数。与Ruby或JavaScript等语言不同，Rust不会自动尝试将非布尔类型的值转换为布尔类型。你必须显式地在if表达式中提供一个布尔类型作为条件。假如你想要if代码块只在数字不等于0时运行，那么我们可以将if表达式修改为如下所示的样子：

src/main.rs

```
fn main() {  
    let number = 3;  
  
    if number != 0 {  
        println!("number was something other than zero");  
    }  
}
```

运行这段代码将会输出number was something other than zero。

使用else if 实现多重条件判断

你可以组合if、else以及else if表达式来实现多重条件判断。例如：

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");
}
}
```

这段程序拥有4条可能的执行路径，运行后可以看到如下所示的输出：

```
$ cargo run
```

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

当这段程序运行时，它会依次检查每一个if表达式，并执行条件首先被判断为真的代码片段。尽管6可以被2整除，但我们既没有看到输出number is divisible by 2，也没有看到else代码块中的number is not divisible by 4, 3, or 2。这是因为Rust会且仅会执行第一个条件为真的代码块，一旦发现满足的条件，它便不会再继续检查剩余的那些条件分支了。

当然，过多的else if表达式可能会使我们的代码变得杂乱无章。在第6章会介绍Rust中另外一个强大的分支结构语法match，它可以被用来应对这种情况。

在let语句中使用if

由于if是一个表达式，所以我们可以在let语句的右侧使用它来生成一个值，如示例3-2所示。

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：将变量绑定到if表达式的结果上

这里的number变量被绑定到了if表达式的输出结果上面。运行这段代码可以看到如下所示的结果：

```
$ cargo run
```

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

记住，代码块输出的值就是其中最后一个表达式的值，另外，数字本身也可以作为一个表达式使用。在上面的例子中，整个if表达式的值取决于究竟哪一个代码块得到了执行。这也意味着，所有if分支可能返回的值都必须是一种类型的；在示例3-2中，if分支与else分支的结果都是i32类型的整数。假如分支表达式产生的类型无法匹配，那么就会触发编译错误，如下所示：

src/main.rs

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

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

运行这段代码会导致编译时错误，因为if与else分支产生了不同类型的值。Rust在错误提示信息中指出了程序出现问题的地方：

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
 |
4 |     let number = if condition {
|     |
5 |         ^~~~~~
|         |
6 |         5
```

这段代码中的if表达式会返回一个整数，而else表达式则会返回一个字符串。因为变量只能拥有单一的类型，所以这段代码无法通过编译。为了对其他使用number变量的代码进行编译时类型检查，Rust需要在编译时确定number的具体类型。如果Rust能够使用运行时确定的number类型，那么它就不得不记录变量所有可能出现的类型，这会使得编译器的实现更加复杂，并丧失许多代码安全保障。

使用循环重复执行代码

我们常常需要重复执行同一段代码，针对这种场景，Rust提供了多种循环（loop）工具。一个循环会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。同样地，我们新创立一个叫作*loops* 的项目来进行与循环相关的实验。

Rust提供了3种循环：loop、while和for。下面让我们来逐一讲解一下。

使用loop重复执行代码

我们可以使用`loop`关键字来指示Rust反复执行某一块代码，直到我们显式地声明退出为止。

例如，把 *loops* 目录中 *src/main.rs* 文件的内容修改为如下所示的样子：

src/main.rs

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

运行这段程序时，除非我们手动强制退出程序，否则 *again!* 会被反复地输出到屏幕中。大部分终端都支持使用键盘快捷键 Ctrl+C 来终止这种陷入无限循环的程序：

```
$ cargo run
```

```
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs
Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

这里的符号 ^C 表示我们按下了快捷键 Ctrl+C。在 ^C 后面，你可能能看到，也可能看不到 *again!*。这取决于程序在接收到退出信号时执行到了循环的哪一步。

当然，Rust 提供了另外一种更加可靠的循环退出方式。你可以在循环中使用 `break` 关键字来通知程序退出循环。如果你还记得的话，我们在第 2 章的“在猜测成功时优雅地退出”一节中曾经使用过 `break`，它帮助我们在用户猜对数字时退出了循环代码。

从 loop 循环中返回值

loop 循环可以被用来反复尝试一些可能会失败的操作，比如检查某个线程是否完成了自己的工作。不管怎么样，你也许会需要将该操作的结果传递给余下的代码。为了实现这一目的，我们可以将需要返回的值添加到 `break` 表达式后面，也就是我们用来终止循环的表达式后面。接着，你就可以在代码中使用这个从循环中返回的值了，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);
}

```

我们在循环前声明了变量counter并将其初始化为0。接着，我们声明了一个名为result的变量来存储循环中返回的值。该循环会在每次迭代时给counter变量中的值加1，并检查计数器是否已经增加至10。一旦条件符合，我们便使用break关键字返回counter * 2。在循环之后，我们还使用了一个分号来结束当前的语句，这会将循环的返回结果赋值给result。最终，我们会打印出result内存储的值，在本例中，也就是20。

while条件循环

另外一种常见的循环模式是在每次执行循环体之前都判断一次条件，假如条件为真则执行代码片段，假如条件为假或在执行过程中碰到break就退出当前循环。这种模式可以通过loop、if、else及break关键字的组合使用来实现。假如你有兴趣的话，可以试着自行完成这一功能。

由于这种模式太过于常见，所以Rust为此提供了一个内置的语言结构：while条件循环。示例3-3中演示了while的使用方式：这段程序会循环执行3次，每次将数字减1，在循环结束后打印出特定消息并退出。

src/main.rs

```

fn main() {
    let mut number = 3;

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

        number = number - 1;
    }
}

```

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

示例3-3：while会在条件为真时重复执行代码

假如你尝试了使用loop、if、else及break来模拟条件循环，那么就会发现这个结构省去了很多冗余的内容，代码整体上会显得更加清晰。当条件为真时，执行循环体中的代码；否则，退出循环。

使用for来循环遍历集合

你可以使用while结构来遍历诸如数组之类的集合中的元素，如示例3-4所示。

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 = index + 1;  
    }  
}
```

示例3-4：使用while结构来遍历集合中的每个元素

在这段程序中，代码会对数组中的所有元素进行计数。它从索引0开始循环，直到数组的最后一个索引（这时，条件index < 5不再为真）。运行这段代码会将数组中的每一个元素都打印出来：

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

如同我们预料的那样，数组中的5个元素都被输出到了终端上。尽管index会在某个时候变为5，但是循环会在我们尝试越界去访问数组的第六个数值之前停止。

需要指出的是，类似的代码非常容易出错，可能会因为使用了不正确的索引长度而使程序崩溃。而且，由于我们增加了运行时的代码来对每一次遍历做出条件判断，所以这段代码的运行效率会比较低。

你可以使用for循环这种更简明的方法来遍历集合中的每一个元素。示例3-5演示了for循环的使用方法。

src/main.rs

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

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

示例3-5：使用for循环遍历集合中的每个元素

运行这段代码，我们会看到与示例3-4同样的输出。但更重要的是，我们增强了代码的安全性，不会出现诸如越界访问或漏掉某些元素之类的问题。

例如，在示例3-4中，假如我们从a数组中移除了某个元素，却忘记将循环中的条件更新为`while index < 4`，那么再次运行代码就会发生崩溃。而使用for循环的话，就不需要时常惦记着在更新数组元素数量时还要去修改代码的其他部分。

for循环的安全性和简捷性使它成为了Rust中最为常用的循环结构。即便是为了实现示例3-3中循环特定次数的任务，大部分的Rust开发者也会选择使用for循环。我们可以配合标准库中提供的Range来实现这一目的，它被用来生成从一个数字开始到另一个数字结束之前的所有数字序列。

下面我们借助for循环来重构示例3-3中的代码，下面的代码使用了一个还未介绍过的方法rev来翻转Range生成的序列：

src/main.rs

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

现在的代码看上去更加整洁了，不是吗？

总结

让我们喘口气，本章可介绍了不少内容！我们在本章学习了变量、标量和复合数据类型、函数、注释、if表达式及循环。假如你想要通过实践来强化自己对这些概念的理解，那么你可以尝试着编写下面的这些程序：

- 摄氏温度与华氏温度的相互转换。
- 生成一个 n 阶的斐波那契数列。
- 打印圣诞颂歌 *The Twelve Days of Christmas* 的歌词，并利用循环处理其中重复的内容。

当你准备好进一步学习时，我们会接着讨论一个在其他编程语言中非常罕见 的概念：所有权。

第4章

认识所有权



所有权可以说是Rust中最为独特的一个功能了。正是所有权概念和相关工具的引入，Rust才能够在没有垃圾回收机制的前提下保障内存安全。因此，正确地了解所有权概念及其在Rust中的实现方式，对于所有Rust开发者来讲都是十分重要的。在本章中，我们会详细地讨论所有权及其相关功能：借用、切片，以及Rust在内存中布局数据的方式。

什么是所有权

所有权 概念本身的含义并不复杂，但作为Rust语言的核心功能，它对语言的其他部分产生了十分深远的影响。

一般来讲，所有的程序都需要管理自己在运行时使用的计算机内存空间。某些使用垃圾回收机制的语言会在运行时定期检查并回收那些没有被继续使用的内存；而在另外一些语言中，程序员需要手动地分配和释放内存。Rust采用了与众不同的第三种方式：它使用包含特定规则的所有权系统来管理内存，这套规则允许编译器在编译过程中执行检查工作，而不会产生任何的运行时开销。

你可能需要一些时间来消化所有权概念，因为它对于大部分程序员来讲是一个非常新鲜的事物。但只要你持之以恒地坚持下去，就可以基于Rust和所有权系统越来越自然地编写出安全且高效的代码！

理解所有权概念还可以帮助你理解Rust中其余那些独有的特性，它会为你接下来的学习打下坚实的基础！在本章中，我们会通过一些示例来学习所有权，这些示例将聚焦于一个十分常用的数据结构：字符串。

栈与堆

在许多编程语言中，程序员不需要频繁地考虑栈空间和堆空间的区别。但对于Rust这样的系统级编程语言来说，一个值被存储在栈上还是被存储在堆上会极大地影响到语言的行为，进而影响到我们编写代码时的设计抉择。由于所有权的某些内容会涉及栈与堆，所以让我们先来简单地了解一下它们。

栈和堆都是代码在运行时可以使用的内存空间，不过它们通常以不同的结构组织而成。栈会以我们放入值时的顺序来存储它们，并以相反的顺序将值取出。这也就是所谓的“后进先出”策略。你可以把栈上的操作想象成堆放盘子：当你需要放置盘子时，你只能将它们放置在栈的顶部，而当你需要取出盘子时，你也只能从顶部取出。换句话说，你没有办法从中间或底部插入或移除盘子。用术语来讲，添加数据这一操作被称作入栈，移除数据则被称作出栈。

所有存储在栈中的数据都必须拥有一个已知且固定的大小。对于那些在编译期无法确定大小的数据，你就只能将它们存储在堆中。堆空间的管理是较为松散的：当你希望将数据放入堆中时，你就可以请求特定大小的空间。操作系统会根据你的请求在堆中找到一块足够大的可用空间，将它标记为已使用，并把指向这片空间地址的指针返回给我们。这一过程就是所谓的堆分配，它也常常被简称为分配。将值压入栈中不叫分配。由于指针的大小是固定的且可以在编译期确定，所以可以将指针存储在栈中。当你想要访问指针所指向的具体数据时，可以通过指针指向的地址来访问。

你可以把这个过程想象为到餐厅聚餐。当你到达餐厅表明自己需要的座位数后，服务员会找到一张足够大的空桌子，并将你们领过去入座。即便这时有小伙伴来迟了，他们也可以通过询问你们就座的位置来找到你们。向栈上推入数据要比在堆上进行分配更有效率一些，因为操作系统省去了搜索新数据存储位置的工作；这个位置永远处于栈的顶端。除此之外，操作系统在堆上分配空间时还必须首先找到足够放下对应数据的空间，并进行某些记录工作来协调随后进行的其余分配操作。

由于多了指针跳转的环节，所以访问堆上的数据要慢于访问栈上的数据。一般来说，现代处理器在进行计算的过程中，由于缓存的缘故，指令在内存中跳转的次数越多，性能就越差。继续使用上面的餐厅来作类比。假设现在同时有许多桌的顾客正在等待服务员的处理。那么最高效的处理方式自然是报完一张桌子所有的订单后再接着服务下一张桌子的顾客。而一旦服务员每次在单个桌子前只处理单个订单，那么他就不得不浪费较多的时间往返于不同的桌子之间。出于同样的原因，处理器在操作排布紧密的数据（比如在栈上）时要比操作排布稀疏的数据（比如在堆上）有效率得多。另外，分配命令本身也可能消耗不少时钟周期。

许多系统编程语言都需要你记录代码中分配的堆空间，最小化堆上的冗余数据，并及时清理堆上的无用数据以避免耗尽空间。而

所有权概念则解决了这些问题。一旦你熟练地掌握了所有权及其相关工具，就可以将这些问题交给Rust处理，减轻用于思考栈和堆的心智负担。不过，知晓如何使用和管理堆内存可以帮助我们理解所有权存在的意义及其背后的工作原理。

所有权规则

现在，让我们来具体看一看所有权规则。你最好先将这些规则记下来，我们会在随后的章节中通过示例来解释它们：

- Rust中的每一个值都有一个对应的变量作为它的所有者。
- 在同一时间内，值有且仅有一个所有者。
- 当所有者离开自己的作用域时，它持有的值就会被释放掉。

变量作用域

由于我们在第2章完整地编写了一个Rust示例程序，所以接下来的示例代码会略过那些基本的语法，比如`fn main() {`等语句，你可以手动将下面的示例代码放置在`main`函数中来完成编译运行任务。这样处理后的示例会更加简单明了，使我们把注意力集中到具体的细节而不是冗余的代码上。

作为所有权的第一个示例，我们先来了解一下变量的作用域。简单来讲，作用域是一个对象在程序中有效的范围。假设有这样一个变量：

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

这里的变量`s`指向了一个字符串字面量，它的值被硬编码到了当前的程序中。变量从声明的位置开始直到当前作用域结束都是有效的。示例4-1中的注释对变量的有效范围给出了具体的说明：

```
{ // 由于变量 s 还未被声明，所以它在这里是不可用的
    let s = "hello"; // 从这里开始变量 s 变得可用
    // 执行与 s 相关的操作
} // 作用域到这里结束，变量 s 再次不可用
```

示例4-1：一个变量及其有效范围的说明

换句话说，这里有两个重点：

- s在进入作用域 后变得有效。
- 它会保持自己的有效性直到自己离开作用域 为止。

到目前为止，Rust语言中变量的有效性与作用域之间的关系跟其他编程语言中的类似。现在，让我们继续在作用域的基础上学习String类型。

String类型

为了演示所有权的相关规则，我们需要一个特别的数据类型，它要比第3章的“数据类型”一节中涉及的类型都更加复杂。之前接触的那些类型会将数据存储在栈上，并在离开自己的作用域时将数据弹出栈空间。我们需要一个存储在堆上的数据类型来研究Rust是如何自动回收这些数据的。

我们将以String类型为例，并将注意力集中到String类型与所有权概念相关的部分。这些部分同样适用于标准库中提供的或你自己创建的其他复杂数据类型。我们会在第8章更加深入地讲解String类型。

你已经在上面的示例中接触过字符串字面量了，它们是那些被硬编码进程序的字符串值。字符串字面量的的确是很方便，但它们并不能满足所有需要使用文本的场景。原因之一在于字符串字面量是不可变的。而另一个原因则在于并不是所有字符串的值都能够在编写代码时确定：假如我们想要获取用户的输入并保存，应该怎么办呢？为了应对这种情况，Rust提供了第二种字符串类型String。这个类型会在堆上分配到自己需要的存储空间，所以它能够处理在编译时未知大小的

文本。你可以调用from函数根据字符串字面量来创建一个String实例：

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

这里的双冒号（::）运算符允许我们调用置于String命名空间下面的特定from函数，而不需要使用类似于string_from这样的名字。我们会在第5章的“方法”一节着重讲解这个语法，并在第7章的“用于在模块树中指明条目的路径”一节中讨论基于模块的命名空间。

上面定义的字符串对象能够 被声明为可变的：

```
let mut s = String::from("hello");
s.push_str(", world!"); // push_str() 函数向String空间的尾部添加了一段字面量

println!("{}", s); // 这里会输出完整的
hello, world!
```

你也许会问：为什么String是可变的，而字符串字面量不是？这是因为它们采用了不同的内存处理方式。

内存与分配

对于字符串字面量而言，由于我们在编译时就知道其内容，所以这部分硬编码的文本被直接嵌入到了最终的可执行文件中。这就是访问字符串字面量异常高效的原因，而这些性质完全得益于字符串字面量的不可变性。不幸的是，我们没有办法将那些未知大小的文本在编译期统统放入二进制文件中，更何况这些文本的大小还可能随着程序的运行而发生改变。

对于String类型而言，为了支持一个可变的、可增长的文本类型，我们需要在堆上分配一块在编译时未知大小的内存来存放数据。这同时也意味着：

- 我们使用的内存是由操作系统在运行时动态分配出来的。

- 当使用完String时，我们需要通过某种方式来将这些内存归还给操作系统。

这里的第一步由我们，也就是程序的编写者，在调用String::from时完成，这个函数会请求自己需要的内存空间。在大部分编程语言中都有类似的设计：由程序员来发起堆内存的分配请求。

然而，对于不同的编程语言来说，第二步实现起来就各有区别了。在某些拥有垃圾回收（Garbage Collector，GC）机制的语言中，GC会代替程序员来负责记录并清除那些不再使用的内存。而对于那些没有GC的语言来说，识别不再使用的内存并调用代码显式释放的工作就依然需要由程序员去完成，正如我们请求分配时一样。按照以往的经验来看，正确地完成这些任务往往是十分困难的。假如我们忘记释放内存，那么就会造成内存泄漏；假如我们过早地释放内存，那么就会产生一个非法变量；假如我们重复释放同一块内存，那么就会产生无法预知的后果。为了程序的稳定运行，我们必须严格地将分配和释放操作一一对应起来。

与这些语言不同，Rust提供了另一套解决方案：内存会自动地在拥有它的变量离开作用域后进行释放。下面的代码类似于示例4-1中的代码，不过我们将字符串字面量换成了String类型：

```
{  
    let s = String::from("hello"); // 从这里开始，变量 s 变得有效  
  
    // 执行与 s 相关的操作  
}  
                                // 作用域到这里结束，变量 s 失效
```

审视上面的代码，有一个很适合用来回收内存给操作系统的地方：变量s离开作用域的地方。Rust在变量离开作用域时，会调用一个叫作drop的特殊函数。String类型的作者可以在这个函数中编写释放内存的代码。记住，Rust会在作用域结束的地方（即}处）自动调用drop函数。

注意

在C++中，这种在对象生命周期结束时释放资源的模式有时也被称作资源获取即初始化（Resource Acquisition Is Initialization, RAII）。假如你使用过类似的模式，那么你应该对Rust中的特殊函数drop并不陌生。

这种模式极大地影响了Rust中的许多设计抉择，并最终决定了我们现在编写Rust代码的方式。在上面的例子中，这套释放机制看起来也许还算简单，然而一旦把它放置在某些更加复杂的环境中，代码呈现出来的行为往往会展现出乎你的意料，特别是当我们拥有多个指向同一处堆内存的变量时。让我们接着来看一看其中一些可能的使用场景。

变量和数据交互的方式：移动

Rust中的多个变量可以采用一种独特的方式与同一数据进行交互。让我们看一看示例4-2中的代码，这里使用了一个整型作为数据：

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

示例4-2：将变量x绑定的整数值重新绑定到变量y上

你也许能够猜到这段代码的执行效果：将整数值5绑定到变量x上；然后创建一个x值的拷贝，并将它绑定到y上。结果我们有了两个变量x和y，它们的值都是5。这正是实际发生的情形，因为整数是已知固定大小的简单值，两个值5会同时被推入当前的栈中。

现在，让我们看一看这段程序的String版本：

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

以上两段代码非常相似，你也许会假设它们的运行方式也是一致的。也就是说，第二行代码可能会生成一个s1值的拷贝，并将它绑定到s2上。不过，事实并非如此。

图4-1展示了String的内存布局，它实际上由3部分组成，如图左侧所示：一个指向存放字符串内容的指针(ptr)、一个长度(len)及一个容量(capacity)，这部分的数据存储在了栈中。图片右侧显示了字符串存储在堆上的文本内容。

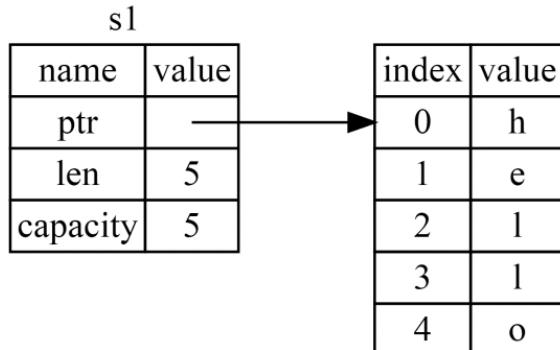


图4-1 绑定到变量s1上、拥有值“hello”的String的内存布局

长度字段被用来记录当前String中的文本使用了多少字节的内存。而容量字段则被用来记录String向操作系统总共获取到的内存字节数量。长度和容量之间的区别十分重要，但我们先不去讨论这个问题，简单地忽略容量字段即可。

当我们把s1赋值给s2时，便复制了一次String的数据，这意味着我们复制了它存储在栈上的指针、长度及容量字段。但需要注意的是，我们没有复制指针指向的堆数据。换句话说，此时的内存布局应该类似于图4-2。

由于Rust不会在复制值时深度地复制堆上的数据，所以这里的布局不会像图4-3中所示的那样。假如Rust依照这样的模式去执行赋值，那么当堆上的数据足够大时，类似于 $s2 = s1$ 这样的指令就会造成相当可观的运行时性能消耗。

s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

The diagram illustrates the state of variables s1 and s2. In s1, the 'ptr' field is initially empty. After copying, it points to the same heap memory as s2's 'ptr' field, which also remains empty. Both s1 and s2 have their 'len' and 'capacity' fields set to 5.

The diagram shows the character data copied into s2. The 'index' column lists indices 0 through 4, and the 'value' column lists the characters 'h', 'e', 'l', 'l', and 'o' respectively. This represents the string "hello".

图4-2 变量s2在复制了s1的指针、长度及容量后的内存布局

s1

name	value
ptr	
len	5
capacity	5

The diagram illustrates the state of variables s1 and s2. In s1, the 'ptr' field is initially empty. After copying, it points to the same heap memory as s2's 'ptr' field, which also remains empty. Both s1 and s2 have their 'len' and 'capacity' fields set to 5.

The diagram shows the character data copied into s2. The 'index' column lists indices 0 through 4, and the 'value' column lists the characters 'h', 'e', 'l', 'l', and 'o' respectively. This represents the string "hello".

s2

name	value
ptr	
len	5
capacity	5

图4-3 当Rust也复制了堆上的数据时，执行完s2 = s1语句后可能产生的内存布局

前面我们提到过，当一个变量离开当前的作用域时，Rust会自动调用它的drop函数，并将变量使用的堆内存释放回收。不过，图4-2中

展示的内存布局里有两个指针指向了同一个地址，这就导致了一个问题：当s2和s1离开自己的作用域时，它们会尝试去重复释放相同的内存。这也就是我们之前提到过的内存错误之一，臭名昭著的二次释放。重复释放内存可能会导致某些正在使用的数据发生损坏，进而产生潜在的安全隐患。

为了确保内存安全，同时也避免复制分配的内存，Rust在这种场景下会简单地将s1废弃，不再视其为一个有效的变量。因此，Rust也不需要在s1离开作用域后清理任何东西。试图在s2创建完毕后使用s1（如下所示）会导致编译时错误。

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

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

为了阻止你使用无效的引用，Rust会产生类似于下面的错误提示信息：

```
error[E0382]: use of moved value: `s1`
--> src/main.rs:5:28
  |
3 | 
  |     let s2 = s1;
  |
  |     -- value moved here
4 | 

5 | 
  |     println!("{} world!", s1);
  |
  |     ^^^ value used here after move
  |

= note: move occurs because `s1` has type `std::string::String`, which does
not implement the `Copy` trait
```

假如你在其他语言中接触过浅度拷贝（shallow copy）和深度拷贝（deep copy）这两个术语，那么你也许会将这里复制指针、长度及容量字段的行为视作浅度拷贝。但由于Rust同时使第一个变量无效了，所以我们使用了新的术语移动（move）来描述这一行为，而不再

使用浅度拷贝。在上面的示例中，我们可以说s1被移动到了s2中。在这个过程中所发生的操作如图4-4所示。

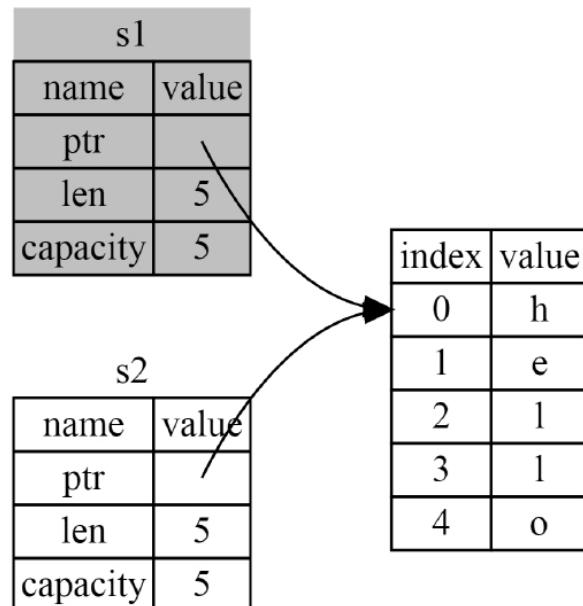


图4-4 s1变为无效之后的内存布局

这一语义完美地解决了我们的问题！既然只有s2有效，那么也就只有它会在离开自己的作用域时释放空间，所以再也没有二次释放的可能性了。

另外，这里还隐含了另外一个设计原则：Rust永远不会自动地创建数据的深度拷贝。因此在Rust中，任何自动的赋值操作都可以被视为高效的。

变量和数据交互的方式：克隆

当你确实需要去深度拷贝String堆上的数据，而不仅仅是栈数据时，就可以使用一个名为clone的方法。我们将在第5章讨论类型方法的语法，但你应该在其他语言中见过类似的东西。

下面是一个实际使用clone方法的例子：

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

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

这段代码在Rust中完全合法，它显式地生成了图4-3中的行为：复制了堆上的数据。

当你看到某处调用了clone时，你就应该知道某些特定的代码将会被执行，而且这些代码可能会相当消耗资源。你可以很容易地在代码中察觉到一些不寻常的事情正在发生。

栈上数据的复制

上面的讨论中遗留了一个没有提及的知识点。我们在示例4-2中曾经使用整型编写出了如下所示的合法代码：

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

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

这与我们刚刚学到的内容似乎有些矛盾：即便代码没有调用clone，x在被赋值给y后也依然有效，且没有发生移动现象。

这是因为类似于整型的类型可以在编译时确定自己的大小，并且能够将自己的数据完整地存储在栈中，对于这些值的复制操作永远都是非常快速的。这也同样意味着，在创建变量y后，我们没有任何理由去阻止变量x继续保持有效。换句话说，对于这些类型而言，深度拷贝与浅度拷贝没有任何区别，调用clone并不会与直接的浅度拷贝有任何行为上的区别。因此，我们完全不需要在类似的场景中考虑上面的问题。

Rust提供了一个名为Copy的trait，它可以用于整数这类完全存储在栈上的数据类型（我们会在第10章详细地介绍trait）。一旦某种类型拥有了Copy这种trait，那么它的变量就可以在赋值给其他变量之后保持可用性。如果一种类型本身或这种类型的任意成员实现了Drop这种trait，那么Rust就不允许其实现Copy这种trait。尝试给某个需要在离开作用域时执行特殊指令的类型实现Copy这种trait会导致编译时错误。附录C中有关于如何给类型添加Copy注解的详细信息。

那么究竟哪些类型是Copy的呢？你可以查看特定类型的文档来确定，不过一般来说，任何简单标量的组合类型都可以是Copy的，任何

需要分配内存或某种资源的类型都不会是Copy的。下面是一些拥有Copy这种trait的类型：

- 所有的整数类型，诸如u32。
- 仅拥有两种值（true和false）的布尔类型：bool。
- 字符类型：char。
- 所有的浮点类型，诸如f64。
- 如果元组包含的所有字段的类型都是Copy的，那么这个元组也是Copy的。例如，(i32, i32)是Copy的，但(i32, String)则不是。

所有权与函数

将值传递给函数在语义上类似于对变量进行赋值。将变量传递给函数将会触发移动或复制，就像是赋值语句一样。示例4-3展示了变量在这种情况下作用域的变化过程。

src/main.rs

```

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

    takes_ownership(s);                    // s 的值被移动进了函数
    // 所以它从这里开始不再有效

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

    makes_copy(x);                      // 变量 x 同样被传递进了函数,
    // 但由于 i32 是 Copy 的, 所以我们依然可以在这之后使用 x

} // x 首先离开作用域, 随后是 s。
// 但由于 s 的值已经发生了移动, 所以没有什么特别的事情会发生

fn takes_ownership(some_string: String) { // some_string 进入作用域
    println!("{}", some_string);
} // some_string 在这里离开作用域, drop 函数被自动调用,
// some_string 所占用的内存也就随之被释放了

fn makes_copy(some_integer: i32) {        // some_integer 进入作用域
    println!("{}", some_integer);
} // some_integer 在这里离开了作用域, 没有什么特别的事情发生

```

示例4-3：函数中所有权和作用域的变化过程

尝试在调用`takes_ownership`后使用变量`s`会导致编译时错误。这类静态检查可以使我们免于犯错。你可以尝试在`main`函数中使用`s`和`x`变量，来看一看在所有权规则的约束下能够在哪些地方合法地使用它们。

返回值与作用域

函数在返回值的过程中也会发生所有权的转移。示例4-4像示例4-3一样详细地标注了这种情况下变量所有权和作用域的变化过程：

`src/main.rs`

```

fn main() {
    let s1 = gives_ownership();           // gives_ownership 将它的返
                                            // 回值移动至 s1 中

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

    let s3 = takes_and_gives_back(s2);   // s2 被移动进函数
    // takes_and_gives_back 中，而这个函数的返回值又被移动到了变量 s3 上
} // s3 在这里离开作用域并被销毁。由于 s2 已经移动了，
// 所以它不会在离开作用域时发生任何事情。s1 最后离开作用域并被销毁。

fn gives_ownership() -> String {      // gives_ownership 会将它的
// 返回值移动至调用它的函数内

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

    some_string                         // some_string 作为返回值移动
                                         // 至调用函数
}

// takes_and_gives_back 将取得一个 String 的所有权并将它作为结果返回
fn takes_and_gives_back(a_string: String) -> String {
    // a_string 进入作用域

    a_string                           // a_string 作为返回值移动至调用函数
}

```

示例4-4：函数在返回值时所有权的转移过程

变量所有权的转移总是遵循相同的模式：将一个值赋值给另一个变量时就会转移所有权。当一个持有堆数据的变量离开作用域时，它的数据就会被drop清理回收，除非这些数据的所有权移动到了另一个变量上。

在所有的函数中都要获取所有权并返回所有权显得有些烦琐。假如你希望在调用函数时保留参数的所有权，那么就不得不将传入的值作为结果返回。除了这些需要保留所有权的值，函数还可能会返回它们本身的结果。

当然，你可以利用元组来同时返回多个值，如示例4-5所示。

src/main.rs

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

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

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

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

    (s, length)
}
```

示例4-5：返回参数的所有权

但这种写法未免太过笨拙了，类似的概念在编程工作中相当常见。幸运的是，Rust针对这类场景提供了一个名为引用 的功能。

引用与借用

在示例4-5中，由于调用calculate_length会导致String移动到函数体内部，而我们又希望在调用完毕后继续使用该String，所以我们不得不使用元组将String作为元素再次返回。

下面的示例重新定义了一个新的calculate_length函数。与之前不同的是，新的函数签名使用了String的引用作为参数而没有直接转移值的所有权：

src/main.rs

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

    let len = calculate_length(&s1);

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

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

首先需要注意的是，变量声明及函数返回值中的那些元组代码都消失了。其次，我们在调用calculate_length函数时使用了&s1作为参数，且在该函数的定义中，我们使用&String替代了String。

这些&代表的就是引用语义，它们允许你在不获取所有权的前提下使用值。图4-5所展示的是该过程的一个图解。

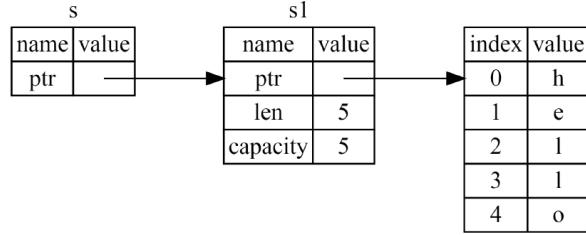


图4-5 &String s指向String s1的图解

注意

与使用`&`进行引用相反的操作被称为解引用 (dereferencing)，它使用`*`作为运算符。我们会在第8章接触到解引用的一些使用场景，并在第15章详细地介绍它们。

现在，让我们仔细观察一下这个函数的调用过程：

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

这里的`&s1`语法允许我们在不转移所有权的前提下，创建一个指向 `s1` 值的引用。由于引用不持有值的所有权，所以当引用离开当前作用域时，它指向的值也不会被丢弃。

同理，函数签名中的`&`用来表明参数 `s` 的类型是一个引用。下面的注释给出了更详细的解释：

```
fn calculate_length(s: &String) -> usize { // s 是一个指向 String 的引用
    s.len()
} // 到这里，s 离开作用域。但是由于它并不持有自己所指向值的所有权，
// 所以没有什么特殊的事情会发生
```

此处，变量 `s` 的有效作用域与其他任何函数参数一样，唯一不同的是，它不会在离开自己的作用域时销毁其指向的数据，因为它并不拥有该数据的所有权。当一个函数使用引用而不是值本身作为参数时，我们便不需要为了归还所有权而特意去返回值，毕竟在这种情况下，我们根本没有取得所有权。

这种通过引用传递参数给函数的方法也被称为借用（borrowing）。在现实生活中，假如一个人拥有某件东西，你可以从他那里把东西借过来。但是当你使用完毕时，就必须将东西还回去。

如果我们尝试着修改借用的值又会发生什么呢？不妨尝试一下示例4-6中的代码。剧透：这段代码无法通过编译！

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

出现的错误如下所示：

```
error[E0596]: cannot borrow immutable borrowed content
`*some_string` as mutable
--> error.rs:8:5
 |

7 |
fn change(some_string: &String) {
|
    ----- use `&mut String` here to make mutable
8 |
    some_string.push_str(", world");
|
    ^^^^^^^^^^^^ cannot borrow as mutable
```

与变量类似，引用是默认不可变的，Rust不允许我们去修改引用指向的值。

可变引用

我们可以通过进行一个小小的调整来修复示例4-6中出现的编译错误：

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，我们需要将变量s声明为mut，即可变的。其次，我们使用&mut s来给函数传入一个可变引用，并将函数签名修改为some_string: &mut String来使其可以接收一个可变引用作为参数。

但可变引用在使用上有一个很大的限制：对于特定作用域中的特定数据来说，一次只能声明一个可变引用。以下代码尝试违背这一限制，则会导致编译错误：

src/main.rs

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

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

出现的错误如下所示：

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
--> borrow_twice.rs:5:19
|
4 | 
5 |     let r1 = &mut s;
| 
|         - first mutable borrow occurs here
5 | 
6 |     let r2 = &mut s;
| 
|         ^ second mutable borrow occurs here
6 | 
```

```
}

|
- first borrow ends here
```

这个规则使得引用的可变性只能以一种受到严格限制的方式来使用。许多刚刚接触Rust的开发者会反复地与它进行斗争，因为大部分的语言都允许你随意修改变量。

但另一方面，在Rust中遵循这条限制性规则可以帮助我们在编译时避免数据竞争。数据竞争（data race）与竞态条件十分类似，它会在指令满足以下3种情形时发生：

- 两个或两个以上的指针同时访问同一空间。
- 其中至少有一个指针会向空间中写入数据。
- 没有同步数据访问的机制。

数据竞争会导致未定义的行为，由于这些未定义的行为往往难以在运行时进行跟踪，也就使得出现的bug更加难以被诊断和修复。Rust则完美地避免了这种情形的出现，因为存在数据竞争的代码连编译检查都无法通过！

与大部分语言类似，我们可以通过花括号来创建一个新的作用域范围。这就使我们可以创建多个可变引用，当然，这些可变引用不会同时存在：

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

{
    let r1 = &mut s;

} // 由于 r1 在这里离开了作用域，所以我们可以合法地再创建一个可变引用

.

let r2 = &mut s;
```

在结合使用可变引用与不可变引用时，还有另外一条类似的限制规则，它会导致下面的代码编译失败：

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

let r1 = &s;           // 没问题
let r2 = &s;           // 没问题
let r3 = &mut s;        // 错误!
```

出现的错误如下所示：

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
as
immutable
--> borrow_thrice.rs:6:19
|
4 |     let r1 = &s;      // 没问题
|         - immutable borrow occurs here
5 |     let r2 = &s;      // 没问题
6 |     let r3 = &mut s; // 错误!
|             ^ mutable borrow occurs here
7 | }
| - immutable borrow ends here
```

哇！发现了吗？我们不能 在拥有不可变引用的同时创建可变引用。不可变引用的用户可不会希望他们眼皮底下的值突然发生变化！不过，同时存在多个不可变引用是合理合法的，对数据的只读操作不会影响到其他读取数据的用户。

尽管这些编译错误会让人不时地感到沮丧，但是请牢记这一点：Rust编译器可以为我们提早（在编译时而不是运行时）暴露那些潜在的bug，并且明确指出出现问题的地方。你不再需要去追踪调试为何数据会在运行时发生了非预期的变化。

悬垂引用

使用拥有指针概念的语言会非常容易错误地创建出悬垂指针。这类指针指向曾经存在的某处内存地址，但该内存已经被释放掉甚至是被重新分配另作他用了。而在Rust语言中，编译器会确保引用永远不会进入这种悬垂状态。假如我们当前持有某个数据的引用，那么编译器可以保证这个数据不会在引用被销毁前离开自己的作用域。

让我们试着来创建一个悬垂引用，并看一看Rust是如何在编译期发现这个错误的：

src/main.rs

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

出现的错误如下所示：

```
error[E0106]: missing lifetime specifier  
--> dangle.rs:5:16  
|  
5 |  
fn dangle() -> &String {  
|  
|  
    ^ expected 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 giving it a 'static lifetime
```

这段错误提示信息包含了一个我们还没有接触到的新概念：生命周期，我们会在第10章详细地讨论它。不过，即使我们先将生命周期放置不管，这条错误提示信息也准确地指出了代码中的问题：

```
this function's return type contains a borrowed value, but there is no value for it  
to be borrowed from.  
\[1\].
```

回过头来仔细看一看我们的dangle函数中究竟发生了些什么：

src/main.rs

```
fn dangle() -> &String {           // dangle会返回一个指向String的引用

    let s = String::from("hello");   // s被绑定到新的String上

    &s // 我们将指向s的引用返回给调用者

} // 变量s在这里离开作用域并随之被销毁，它指向的内存自然也不再有效。

// 危险！
```

由于变量s创建在函数dangle内，所以它会在dangle执行完毕时随之释放。但是，我们的代码依旧尝试返回一个指向s的引用，这个引用指向的是一个无效的String，这可不对！Rust成功地拦截了我们的危险代码。

解决这个问题的方法也很简单，直接返回String就好：

src/main.rs

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

    s
}
```

这种写法没有任何问题，所有权被转移出函数，自然也就不会涉及释放操作了。

引用的规则

让我们简要地概括一下本节对引用的讨论：

- 在任何一段给定的时间里，你要么只能拥有一个可变引用，要么只能拥有任意数量的不可变引用。
- 引用总是有效的。

接下来，我们会继续讨论另外一种特殊的引用形式：切片。

[1] 译者注：这条错误提示信息的意思是，该函数的返回类型包含了一个借用，但却不存在可供其借用的值。

切片

除了引用，Rust还有另外一种不持有所有权的数据类型：切片（slice）。切片允许我们引用集合中某一段连续的元素序列，而不是整个集合。

考虑这样一个小问题：编写一个搜索函数，它接收字符串作为参数，并将字符串中的首个单词作为结果返回。如果字符串中不存在空格，那么就意味着整个字符串是一个单词，直接返回整个字符串作为结果即可。

让我们来看一下这个函数的签名应该如何设计：

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

由于我们不需要获得传入值的所有权，所以这个函数first_word采用了&String作为参数。但它应该返回些什么呢？我们还没有一个获取部分字符串的方法。当然，你可以将首个单词结尾处的索引返回给调用者，如示例4-7所示：

src/main.rs

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

    for (i, &item)❷ in bytes.iter()❸.enumerate() {
        ❹ if item == b' ' {
            return i;
        }
    }
    ❺ s.len()
}
```

示例4-7：first_word函数会返回String参数中首个单词结尾处的索引作为结果

这段代码首先使用as_bytes方法❶将String转换为字节数组，因为我们的算法需要依次检查String中的字节是否为空格。接着，我们通过iter方法❷创建了一个可以遍历字节数组的迭代器。

我们会在第13章详细讨论这里新出现的迭代器。目前，你只需要知道iter方法会依次返回集合中的每一个元素即可。随后的enumerate则将iter的每个输出作为元素逐一封装在对应的元组中返回。元组的第一个元素是索引，第二个元素是指向集合中字节的引用。使用enumerate可以较为方便地获得迭代索引。

既然enumerate方法返回的是一个元组，那么我们就可以使用模式匹配来解构它，就像Rust中其他使用元组的地方一样。在for循环的遍历语句中，我们指定了一个解构模式，其中i是元组中的索引部分，而&item ❸则是元组中指向集合元素的引用。由于我们从iter().enumerate()中获取的是产生引用元素的迭代器，所以我们在模式中使用了&。

在for循环的代码块中，我们使用了字节字面量语法来搜索数组中代表着空格的字节❹。这段代码会在搜索到空格时返回当前位置索引，并在搜索失败时返回传入字符串的长度s.len()❺。

现在，我们初步实现了期望的功能，它能够成功地搜索并返回字符串中第一个单词结尾处的位置索引。但这里依然存在一个设计上的缺陷。我们将一个usize值作为索引独立地返回给调用者，但这个值在脱离了传入的&String的上下文之后便毫无意义。换句话说，由于这个值独立于String而存在，所以在函数返回值后，我们就再也无法保证它的有效性了。示例4-8中使用first_word函数演示了这种返回值失效的情形：

src/main.rs

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

    let word = first_word(&s); // 索引5会被绑定到变量word上

    s.clear(); // 这里的clear方法会清空当前字符串，使之变为""
```

```
// 虽然word依然拥有5这个值，但因为我们用于搜索的字符串发生了改变，  
  
//所以这个索引也就没有任何意义了，word到这里便失去了有效性  
  
}
```

示例4-8：保存first_word产生的返回值并改变其中String的内容

上面的程序在编译器看来没有任何问题，即便我们在调用 `s.clear()` 之后使用 `word` 变量也是没有问题的。同时由于 `word` 变量本身与 `s` 没有任何关联，所以 `word` 的值始终都是 5。但当我们再次使用 5 去从变量 `s` 中提取单词时，一个 bug 就出现了：此时 `s` 中的内容早已在我们将 5 存入 `word` 后发生了改变。

这种 API 的设计方式使我们需要随时关注 `word` 的有效性，确保它与 `s` 中的数据是一致的，类似的工作往往相当烦琐且易于出错。这种情况对于另一个函数 `second_word` 而言更加明显。这个函数被设计来搜索字符串中的第二个单词，它的签名也许会被设计为下面这样：

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

现在，我们需要同时维护起始和结束两个位置的索引，这两个值基于数据的某个特定状态计算而来，却没有跟数据产生任何程度上的联系。于是我们有了 3 个彼此不相关的变量需要被同步，这可不妙。

幸运的是，Rust 为这个问题提供了解决方案：字符串切片。

字符串切片

字符串切片是指向 `String` 对象中某个连续部分的引用，它的使用方式如下所示：

```
let s = String::from("hello world");  
  
let hello = &s[0..5];  
❶ let world = &s[6..11];
```

这里的语法与创建指向整个 `String` 对象的引用有些相似，但不同的是，新语法在结尾的地方多出了一段 `[0..5]`。这段额外的声明告诉

编译器我们正在创建一个String的切片引用，而不是对整个字符串本身的引用。

我们可以在一对方括号中指定切片的范围区间[`starting_index..ending_index`]，其中的`starting_index`是切片起始位置的索引值，`ending_index`是切片终止位置的下一个索引值。切片数据结构在内部存储了指向起始位置的引用和一个描述切片长度的字段，这个描述切片长度的字段等价于`ending_index`减去`starting_index`。所以在上面示例的❶中，`world`是一个指向变量`s`第七个字节并且长度为5的切片。

图4-6所展示的是字符串切片的图解。

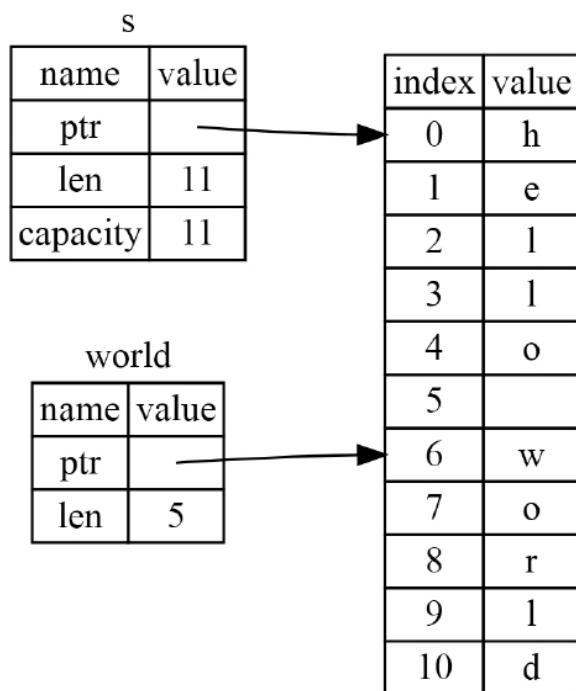


图4-6 指向String对象中某个连续部分的字符串切片

Rust的范围语法..有一个小小的话语糖：当你希望范围从第一个元素（也就是索引值为0的元素）开始时，则可以省略两个点号之前的值。换句话说，下面两个创建切片的表达式是等价的：

```
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[..2];
```

同样地，假如你的切片想要包含String中的最后一个字节，你也可以省略双点号之后的值。下面的切片表达式依然是等价的：

```
let s = String::from("hello");
let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];
```

你甚至可以同时省略首尾的两个值，来创建一个指向整个字符串所有字节的切片：

```
let s = String::from("hello");
let len = s.len();
let slice = &s[0..len];
let slice = &s[..];
```

注意

字符串切片的边界必须位于有效的UTF-8字符边界内。尝试从一个多字节字符的中间位置创建字符串切片会导致运行时错误。为了将问题简化，我们只会在本节中使用ASCII字符集；你可以在第8章的“使用字符串存储UTF-8编码的文本”一节中找到更多有关UTF-8的讨论。

基于所学到的这些知识，让我们开始重构first_word函数吧！该函数可以返回一个切片作为结果。字符串切片的类型写作&str：

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[..]
}
```

这个新函数中搜索首个单词索引的方式类似于示例4-7中的方式。一旦搜索成功，就返回一个从首字符开始到这个索引位置结束的字符串切片。

调用新的first_word函数会返回一个与底层数据紧密联系的切片作为结果，它由指向起始位置的引用和描述元素长度的字段组成。

当然，我们也可以用同样的方式重构second_word函数：

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

由于编译器会确保指向String的引用持续有效，所以我们新设计的接口变得更加健壮且直观了。还记得在示例4-8中故意构造出的错误吗？那段代码在搜索完成并保存索引后清空了字符串的内容，这使得我们存储的索引不再有效。它在逻辑上明显是有问题的，却不会触发任何编译错误，这个问题只会在我们使用第一个单词的索引去读取空字符串时暴露出来。切片的引入使我们可以在开发早期快速地发现此类错误。在示例4-8中，新的first_word函数在编译时会抛出一个错误，尝试运行以下代码：

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);
}
```

编译错误如下所示：

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
--> src/main.rs:6:5
|
4 |
5 |     let word = first_word(&s);
|     |
|         - immutable borrow occurs here
6 |
7 |     s.clear(); // 错误
|
```

```
|  
|     ^ mutable borrow occurs here  
7 | }  
|  
- immutable borrow ends here
```

回忆一下借用规则，当我们拥有了某个变量的不可变引用时，我们就无法同时取得该变量的可变引用。由于clear需要截断当前的String实例，所以调用clear需要传入一个可变引用。这就是编译失败的原因。Rust不仅使我们的API更加易用，它还在编译过程中帮助我们避免了此类错误。

字符串字面量就是切片

还记得我们讲过字符串字面量被直接存储在了二进制程序中吗？在学习了切片之后，我们现在可以更恰当地理解字符串字面量了：

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

在这里，变量s的类型其实就是&str：它是一个指向二进制程序特定位置的切片。正是由于&str是一个不可变的引用，所以字符串字面量自然才是不可变的。

将字符串切片作为参数

既然我们可以分别创建字符串字面量和String的切片，那么就能够进一步优化first_word函数的接口，下面是它目前的签名：

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

比较有经验的Rust开发者往往会采用下面的写法，这种改进后的签名使函数可以同时处理String与&str：

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

示例4-9：使用字符串切片作为参数s的类型来改进first_word函数

当你持有字符串切片时，你可以直接调用这个函数。而当你持有String时，你可以创建一个完整String的切片来作为参数。在定义函

数时使用字符串切片来代替字符串引用会使我们的API更加通用，且不会损失任何功能，尝试运行以下代码：

src/main.rs

```
fn main() {
    let my_string = String::from("hello world");
    // first_word 可以接收String对象的切片作为参数

    let word = first_word(&my_string[..]);
    let my_string_literal = "hello world";
    // first_word 可以接收字符串字面量的切片作为参数

    let word = first_word(&my_string_literal[..]);
    // 由于字符串字面量本身就是切片，所以我们可以在那里直接将它传入函数，

    // 而不需要使用额外的切片语法！

    let word = first_word(my_string_literal);
}
```

其他类型的切片

从名字上就可以看出来，字符串切片是专门用于字符串的。但实际上，Rust还有其他更加通用的切片类型，以下面的数组为例：

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

就像我们想要引用字符串的某个部分一样，你也可能会希望引用数组的某个部分。这时，我们可以这样做：

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

这里的切片类型是`&[i32]`，它在内部存储了一个指向起始元素的引用及长度，这与字符串切片的工作机制完全一样。你将在各种各样的集合中接触到此类切片，而我们会在第8章讨论动态数组时再来介绍那些常用的集合。

总结

所有权、借用和切片的概念是Rust可以在编译时保证内存安全的关键所在。像其他系统级语言一样，Rust语言给予了程序员完善的内存使用控制能力。除此之外，借助于本章学习的这些工具，Rust还能够自动清除那些所有者离开了作用域的数据。这极大地减轻了使用者的心智负担，也不需要专门去编写销毁代码和测试代码。

所有权影响了Rust中绝大部分功能的运作机制，有关这些概念的深入讨论会贯穿本书剩余的章节。在接下来的第5章中，我们会学习如何使用struct来组装不同的数据。

第5章

使用结构体来组织相关联的数据



结构，或者说结构体，是一种自定义数据类型，它允许我们命名多个相关的值并将它们组成一个有机的结合体。假如你曾经有过面向对象的编程经历，那么你可以把结构体视作对象中的数据属性。在本章中，我们会首先对比元组与结构体之间的异同，并演示如何使用结构体。接着，我们还会讨论如何定义方法和关联函数，它们可以指定那些与结构体数据相关的行为。结构体与枚举类型（将在第6章学习）是用来创建新类型的基本工具，这些特定领域中的新类型同样可以享受到Rust编译时类型检查系统的所有优势。

定义并实例化结构体

结构体与我们在第3章讨论过的元组有些相似。和元组一样，结构体中的数据可以拥有不同的类型。而和元组不一样的是，结构体需要给每个数据赋予名字以便清楚地表明它们的意义。正是由于有了这些名字，结构体的使用要比元组更加灵活：你不再需要依赖顺序索引来指定或访问实例中的值。

关键字`struct`被用来定义并命名结构体，一个良好的结构体名称应当能够反映出自身数据组合的意义。除此之外，我们还需要在随后的花括号中声明所有数据的名字及类型，这些数据也被称作字段。示例5-1中展示了一个用于存储账户信息的结构体定义：

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

示例5-1：User结构体的定义

为了使用定义好的结构体，我们需要为每个字段赋予具体的值来创建结构体实例。可以通过声明结构体名称，并使用一对花括号包含键值对来创建实例。其中的键对应字段的名字，而值则对应我们想要在这些字段中存储的数据。这里的赋值顺序并不需要严格对应我们在结构体中声明它们的顺序。换句话说，结构体的定义就像类型的通用模板一样，当我们把具体的数据填入模板时就创建出了新的实例。例如，我们可以像示例5-2这样来声明一个特定的用户。

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

示例5-2：创建一个User结构体的实例

在获得了结构体实例后，我们可以通过点号来访问实例中的特定字段。如果你想获得某个用户的电子邮件地址，那么可以使用`user1.email`来获取。另外，假如这个结构体的实例是可变的，那么我们还可以通过点号来修改字段中的值。示例5-3展示了如何修改一个可变User实例中`email`字段的值。

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

示例5-3：修改User实例中email字段的值

需要注意的是，一旦实例可变，那么实例中的所有字段都将是可变的。Rust不允许我们单独声明某一部分字段的可变性。如同其他表达式一样，我们可以在函数体的最后一个表达式中构造结构体实例，来隐式地将这个实例作为结果返回。

示例5-4中的`build_user`函数会使用传入的邮箱和用户名参数构造并返回User实例。另外两个字段`active`和`sign_in_count`则分别被赋予了值`true`和`1`。

```
fn build_user(email: String, username: String) -> User {  
    User {  
        email: email,  
        username: username,  
        active: true,  
        sign_in_count: 1,  
    }  
}
```

示例5-4：一个接收邮箱和用户名作为参数并返回User实例的函数 build_user

在函数中使用与结构体字段名相同的参数名可以让代码更加易于阅读，但分别两次书写`email`和`username`作为字段名与变量名则显得有些繁琐了，特别是当结构体拥有较多字段时。Rust为此提供了一个简便的写法。

在变量名与字段名相同时使用简化版的字段初始化方法

由于示例5-4中的参数与结构体字段拥有完全一致的名称，所以我们可以使用名为字段初始化简写（field init shorthand）的语法来重构build_user函数。这种语法不会改变函数的行为，但却能让我们免于在代码中重复书写email和username，如示例5-5所示。

```
fn build_user(email: String, username: String) -> User {  
    User {  
        email,  
        username,  
        active: true,  
        sign_in_count: 1,  
    }  
}
```

示例5-5：build_user函数中使用了相同的参数名与字段名，并采用了字段初始化简写语法进行编写

上面的代码首先创建了一个拥有email字段的User结构体实例。我们希望使用build_user函数的email参数来初始化这个实例的email字段。由于字段email与参数email拥有相同的名字，所以我们不用书写完整的email: email语句，只保留email即可。

使用结构体更新语法根据其他实例创建新实例

在许多情形下，在新创建的实例中，除了需要修改的小部分字段，其余字段的值与旧实例中的完全相同。我们可以使用结构体更新语法来快速实现此类新实例的创建。

首先，示例5-6展示了如何在不使用更新语法的前提下创建新的User实例user2。除了email和username这两个字段，其余的值都与在示例5-2中创建的user1实例中的值一样。

```
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername567"),  
    active: user1.active,  
    sign_in_count: user1.sign_in_count,  
};
```

示例5-6：使用user1中的某些值来创建一个新的User实例

通过结构体更新语法，我们可以使用更少的代码来实现完全相同的效果，如示例5-7所示。这里的双点号`..`表明剩下的那些还未被显式赋值的字段都与给定实例拥有相同的值。

```
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername567"),  
    ..user1  
};
```

示例5-7：使用结构体更新语法来为一个User实例设置新的email和username字段的值，并从user1实例中获取剩余字段的值

示例5-7中的代码新创建了一个实例user2，它的email和username字段的值与实例user1中的不同，但是active和sign_in_count字段的值与user1中的相同。

使用不需要对字段命名的元组结构体来创建不同的类型

除了上面的方法，你还可以使用另外一种类似于元组的方式定义结构体，这种结构体也被称作元组结构体。元组结构体同样拥有用于表明自身含义的名称，但你无须在声明它时对其字段进行命名，仅保留字段的类型即可。一般来说，当你想要给元组赋予名字，并使其区别于其他拥有同样定义的元组时，你就可以使用元组结构体。在这种情况下，像常规结构体那样为每个字段命名反而显得有些烦琐和形式化了。

定义元组结构体时依然使用`struct`关键字开头，并由结构体名称及元组中的类型定义组成。下面的代码中展示了两个分别叫作Color和Point的元组结构体定义：

```
struct Color(i32, i32, i32);  
struct Point(i32, i32, i32);  
  
let black = Color(0, 0, 0);  
let origin = Point(0, 0, 0);
```

注意，这里的black和origin是不同的类型，因为它们两个分别是不同元组结构体的实例。你所定义的每一个结构体都拥有自己的类型，即便结构体中的字段拥有完全相同的类型。例如，一个以Color类型作为参数的函数不能合法地接收Point类型的参数，即使它们都是由3个i32值组成的。除此之外，元组结构体实例的行为就像元组一样：你可以通过模式匹配将它们解构为单独的部分，你也可以通过. 及索引来访问特定字段。

没有任何字段的空结构体

也许会出乎你的意料，Rust允许我们创建没有任何字段的结构体！因为这种结构体与空元组()十分相似，所以它们也被称为空结构体。当你想要在某些类型上实现一个trait，却不需要在该类型中存储任何数据时，空结构体就可以发挥相应的作用。我们将会在第10章讨论trait。

结构体数据的所有权

在示例5-1的User结构体定义中，我们使用了自持所有权的String类型而不是&str字符串切片类型。这是一个有意为之的选择，因为我们希望这个结构体的实例拥有自身全部数据的所有权。在这种情形下，只要结构体是有效的，那么它携带的全部数据也就是有效的。

当然，我们也可以在结构体中存储指向其他数据的引用，不过这需要用到Rust中独有的生命周期功能，关于它的详细讨论会在第10章进行。生命周期保证了结构体实例中引用数据的有效期不短于实例本身。你也许会尝试在没有生命周期的情形下，直接在结构体中声明引用字段：

src/main.rs

```
struct User {  
    username: &str,  
    email: &str,  
    sign_in_count: u64,  
    active: bool,
```

```
}

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

但这段代码可没办法通过检查，Rust会在编译过程中报错，提示我们应该指定生命周期：

```
errorE0106: missing lifetime specifier
-->
|
1

2 |
    username: &str,
    |

        ^ expected lifetime parameter
errorE0106: missing lifetime specifier
-->
|
1

3 |
    email: &str,
    |

        ^ expected lifetime parameter
```

不用着急，我们会在第10章学习如何解决上面这些错误，并合法地在结构体中存储引用字段。现在，我们先使用持有自身所有权的String而不是像&str一样的引用来解决这个问题。

一个使用结构体的示例程序

为了能够了解结构体的使用时机，让我们来共同编写一个计算长方形面积的程序。我们会从使用变量开始，并逐渐将它重构为使用结构体的版本。

使用Cargo创建一个叫作*rectangles* 的二进制项目。这个程序会接收以像素为单位的宽度和高度作为输入，并计算出对应的长方形面积。示例5-8展示了文件*src/main.rs* 中实现的一段简单程序。

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：分别指定宽度和高度变量来计算长方形的面积

现在，使用指令cargo run来运行这段程序：

```
The area of the rectangle is 1500 square pixels.
```

尽管示例5-8中的程序成功地计算出了长方形的面积，但它还有可以改进的空间。这里的宽度和高度是相互关联的两个数据，它们两个组合在一起才能定义一个长方形。

示例5-8中的问题可以在area的签名中看到：

```
fn area(width: u32, height: u32) -> u32 {
```

area函数被编写出来计算长方形的面积，但它却有两个不同的参数。这两个参数是相互关联的，但程序中却没有任何地方可以表现出这一点。将宽度和高度放到一起能够使我们的代码更加易懂，也更加易于维护。我们曾经在第3章的“元组类型”一节中讨论过一种可行的组织方式：元组。

使用元组来重构代码

示例5-9中展示了使用元组重构后的代码版本：

src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        ① area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    ② dimensions.0 * dimensions.1
}
```

示例5-9：通过元组来指定长方形的宽度和高度

新的程序从某种程度上来说要更好一些。元组使输入的参数结构化了，我们现在只需要传递一个参数①便可以调用函数area了。但从另一方面来讲，这个版本的程序变得难以阅读了。元组并不会给出其中元素的名字，我们可能会对使用索引获取的元组值产生困惑和混淆②。

在计算面积时，混淆宽度和高度的使用似乎没有什么问题，但是当我们需要将这个长方形绘制到屏幕上时，这样的混淆就会出问题了！我们必须牢牢地记住，元素的索引0对应了宽度width，而索引1则对应了高度height。如果有其他人想要接手这部分代码，那么他也不得不搞清楚并牢记这些规则。在实际工作中，由于没有在代码里表明

数据的意义，我们总是会因为忘记或弄混这些不同含义的值而导致各种程序错误。

使用结构体来重构代码：增加有意义的描述信息

我们可以使用结构体来为这些数据增加有意义的标签。在重构元组为结构体的过程中，我们会分别给结构体本身及它们的每个字段赋予名字，如示例5-10所示。

src/main.rs

```
❶struct Rectangle {
    ❷width: u32,
    height: u32,
}

fn main() {
    ❸let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

❹ fn area(rectangle: &Rectangle) -> u32 {
    ❺ rectangle.width * rectangle.height
}
```

示例5-10：定义Rectangle结构体

在上面的代码中，我们首先定义了结构体并将它命名为Rectangle❶。随后，在花括号中依次定义了u32类型的字段width和height❷。接着，在main函数中创建了一个宽度为30和高度为50的Rectangle实例❸。

现在，用于计算面积的area函数在被定义时只需要接收一个rectangle参数，它是结构体Rectangle实例的不可变借用❹。正如我们在第4章提到过的，在函数签名和调用过程中使用&进行引用是因为我们希望借用结构体，而不是获取它的所有权，这样main函数就可以保留rect1的所有权并继续使用它。

area函数会在执行时访问Rectangle实例的width和height字段❸。此时，area的函数签名终于准确无误地明白了我们的意图：使用width和height这两个字段计算出Rectangle的面积。Rectangle结构体表明了宽度和高度是相互关联的两个值，并为这些值提供了描述性的名字，而无须使用类似于元组索引的0或1。如此，我们的代码看起来就更加清晰了。

通过派生trait增加实用功能

如果我们可以打印出Rectangle实例及其每个字段的值，那么调试代码的过程就会变得简单许多。你也许会试着使用之前接触过的println! 宏来达到这个目的，如示例5-11所示，但它暂时还无法通过编译。

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

尝试运行上面这段代码会产生含有如下核心信息的错误：

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

println! 宏可以执行多种不同的文本格式化命令，而作为默认选项，格式化文本中的花括号会告知println! 使用名为Display的格式化方法：这类输出可以被展示给直接的终端用户。我们目前接触过的所有基础类型都默认地实现了Display，因为当你想要给用户展示1或其他基础类型时没有太多可供选择的方式。但对于结构体而言，println! 则无法确定应该使用什么样的格式化内容：在输出的时候是否需要逗号？需要打印花括号吗？所有的字段都应当被展示吗？正是由于这种不确定性，Rust没有为结构体提供默认的Display实现。

假如我们继续阅读上面的编译器错误提示信息，则会发现一条有用的帮助信息：

```
= 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
```

这好像给我们指明了解决问题的方法，让我们赶紧试一试！修改过的`println!`宏调用会类似于`println!("rect1 is {:?}", rect1);`。我们把标识符号`:?`放入了花括号中，它会告知`println!`当前的结构体需要使用名为`Debug`的格式化输出。`Debug`是另外一种格式化trait，它可以让我们在调试代码时以一种对开发者友好的形式打印出结构体。

修改完代码后再次尝试运行程序。让人沮丧的是，我们还是触发了一个错误：

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Debug`
```

不过编译器再次给出了一条有用的帮助信息：

```
= help: the trait `std::fmt::Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
```

Rust确实包含了打印调试信息的功能，但我们必须为自己的结构体显式地选择这一功能。为了完成该声明，我们在结构体定义前添加了`#[derive(Debug)]`注解，如示例5-12所示。

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：添加注解来派生`Debug` trait，并使用调试格式打印出`Rectangle`实例

现在，让我们再次运行程序。这下应该不会有任何错误了，我们将在程序成功运行后观察到如下所示的输出内容：

```
rect1 is Rectangle { width: 30, height: 50 }
```

真棒！这也许还不是最漂亮的输出，但它展示了实例中所有字段的值，这毫无疑问会对调试有帮助。而对于某些更为复杂的结构体，你可能会希望调试的输出更加易读一些，为此我们可以将`println!`字符串中的`{:?}`替换为`{:#?}`。修改后的输出会变成下面的样子：

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

实际上，Rust提供了许多可以通过`derive`注解来派生的trait，它们可以为自定义的类型增加许多有用的功能。所有这些trait及它们所对应的行为都可以在附录C中找到。我们会在第10章学习如何通过自定义行为来实现这些trait，以及创建新的trait。

这里的`area`函数其实是非常有针对性的：它只会输出长方形的面积。既然它不能被用于其他类型，那么将其行为与`Rectangle`结构体本身结合得更加紧密一些可以帮助我们理解它的含义。接下来，我们会把`area`函数转变为`Rectangle`的方法来继续重构当前的代码。

方法

方法与函数十分相似：它们都使用fn关键字及一个名称来进行声明；它们都可以拥有参数和返回值；另外，它们都包含了一段在调用时执行的代码。但是，方法与函数依然是两个不同的概念，因为方法总是被定义在某个结构体（或者枚举类型、trait对象，我们会在第6章和第17章分别介绍它们）的上下文中，并且它们的第一个参数永远都是self，用于指代调用该方法的结构体实例。

定义方法

现在，让我们把那个以Rectangle实例作为参数的area函数，改写为定义在Rectangle结构体中的area方法，如示例5-13所示。

src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

❶impl Rectangle {
❷    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.",
        ❸ rect1.area()
    );
}
```

示例5-13：在Rectangle结构体中定义area方法

为了在Rectangle的上下文环境中定义这个函数，我们需要将area函数移动到一个由impl (implementation) 关键字①起始的代码块中②，并把签名中的第一个参数（也是唯一的那个参数）和函数中使用该参数的地方改写为self。除此之外，我们还需要把main函数中调用area函数的地方，用方法调用的语法进行改写。前者是将rect1作为参数传入area函数，而后者则直接在Rectangle实例上调用area方法③。方法调用是通过在实例后面加点号，并跟上方法名、括号及可能的参数来实现的。

由于方法的声明过程被放置在impl Rectangle块中，所以Rust能够将self的类型推导为Rectangle。也正是因为这样，我们才可以在area的签名中使用&self来代替rectangle: &Rectangle。但我们依然需要在self之前添加&，就像&Rectangle一样。方法可以在声明时选择获取self的所有权，也可以像本例一样采用不可变的借用&self，或者采用可变的借用&mut self。总之，就像是其他任何普通的参数一样。

在这里，选择&self签名的原因和之前选择使用&Rectangle的原因差不多：我们既不用获得数据的所有权也不需要写入数据，而只需要读取数据即可。假如我们想要在调用方法时改变实例的某些数据，那么就需要将第一个参数改写为&mut self。通常来说，将第一个参数标记为self并在调用过程中取得实例的所有权的方法并不常见。这种技术有可能会被用于那些需要将self转换为其他类型，且在转换后想要阻止调用者访问原始实例的场景。

使用方法替代函数不仅能够避免在每个方法的签名中重复编写self的类型，还有助于我们组织代码的结构。我们可以将某个类型的实例需要的功能放置在同一个impl块中，从而避免用户在代码库中盲目地自行搜索它们。

运算符->到哪里去了？

在C和C++中调用方法时有两个不同的运算符：它们分别是直接用于对象本身的. 及用于对象指针的->。之所以有这样的区别，是因为我们在调用指针的方法时首先需要对该指针进行解引用。换句

话说，假如object是一个指针，那么object->something()的写法实际上等价于(* object). something()。

虽然Rust没有提供类似的->运算符，但作为替代，我们设计了一种名为自动引用和解引用的功能。方法调用是Rust中少数几个拥有这种行为的地方之一。

它的工作模式如下：当你使用object. something() 调用方法时，Rust会自动为调用者object添加&、&mut或*，以使其能够符合方法的签名。换句话说，下面两种方法调用是等价的：

```
p1.distance(&p2);  
(&p1).distance(&p2);
```

第一种调用看上去要简捷得多。这种自动引用行为之所以能够行得通，是因为方法有一个明确的作用对象：self的类型。在给出调用者和方法名的前提下，Rust可以准确地推导出方法是否是只读的(&self)，是否需要修改数据(&mut self)，是否会获取数据的所有权(self)。这种针对方法调用者的隐式借用在实践中可以让所有权系统更加友好且易于使用。

带有更多参数的方法

现在，让我们通过实现Rectangle结构体的第二个方法来继续练习使用这种方法。这次我们要实现的是：检测当前的Rectangle实例是否能完整包含传入的另外一个Rectangle实例，如果是的话就返回true，否则返回false。也就是说，一旦我们完成了这个方法(can_hold)，我们就能像示例5-14中所示的那样去使用它。

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：使用还没有编写好的can_hold方法

因为rect2的两个维度都要小于rect1，而rect3的宽度要大于rect1，所以如果一切正常的话，它们应当能够输出如下所示的结果：

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

因为我们想要定义的是方法，所以我们会把新添加的代码放置到impl Rectangle块中。另外，这个名为can_hold的方法需要接收另一个Rectangle的不可变借用作为参数。通过观察调用方法时的代码便可以推断出此处的参数类型：语句rect1.can_hold(&rect2)中传入了一个&rect2，也就是指向Rectangle实例rect2的不可变借用。为了计算包容关系，我们只需要去读取rect2的数据（而不是写入，写入意味着需要一个可变借用）。main函数还应该在调用can_hold方法后继续持有rect2的所有权，从而使得我们可以在随后的代码中继续使用这个变量。can_hold方法在实现时会依次检查self的宽度和长度是否大于传入的Rectangle实例的宽度和长度，并返回一个布尔类型作为结果。现在，让我们在示例5-13里出现过的impl块中添加can_hold方法，如示例5-15所示。

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：基于Rectangle实现can_hold方法，该方法可以接收另外一个Rectangle作为参数

当你将这段代码与示例5-14中的main函数合并运行后，就可以得到预期的输出结果。实际上，方法同样可以在self参数后增加签名来接收多个参数，就如同函数一样。

关联函数

除了方法，`impl`块还允许我们定义不用接收`self`作为参数的函数。由于这类函数与结构体相互关联，所以它们也被称为关联函数（associated function）。我们将其命名为函数而不是方法，是因为它们不会作用于某个具体的结构体实例。你曾经接触过的`String::from`就是关联函数的一种。

关联函数常常被用作构造器来返回一个结构体的新实例。例如，我们可以编写一个接收一个维度参数的关联函数，它会将输入的参数同时用作长度与宽度来构造正方形的`Rectangle`实例：

src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}
```

我们可以在类型名称后添加`::`来调用关联函数，就像`let sq = Rectangle::square(3);`一样。这个函数位于结构体的命名空间中，这里的`::`语法不仅被用于关联函数，还被用于模块创建的命名空间。我们会在第7章讨论此处的模块概念。

多个`impl`块

每个结构体可以拥有多个`impl`块。例如，示例5-15中的代码等价于示例5-16中的，下面的代码将方法放置到了不同的`impl`块中。

rc/main.rs

```
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`块来重写示例5-15

虽然这里没有采用多个impl块的必要，但它仍然是合法的。我们会在第10章讨论泛型和trait时看到多个impl块的实际应用场景。

总结

结构体可以让我们基于特定领域创建有意义的自定义类型。通过使用结构体，你可以将相关联的数据组合起来，并为每条数据赋予名字，从而使代码变得更加清晰。方法可以让我们为结构体实例指定行为，而关联函数则可以将那些不需要实例的特定功能放置到结构体的命名空间中。

但结构体并不是创建自定义类型的唯一方法，接下来我们会继续学习Rust中另外一个十分常用的工具：枚举。

第6章

枚举与模式匹配



枚举类型，通常也被简称为枚举，它允许我们列举所有可能的值来定义一个类型。在本章中，我们首先会定义并使用一个枚举，以向你展示枚举是如何连同数据来一起编码信息的。接着，我们会讨论一个特别有用的枚举：Option，它常常被用来描述某些可能不存在的值。随后，我们将学会如何在match表达式中使用模式匹配，并根据不同的枚举值来执行不同的代码。最后，我们还会介绍另外一种常用的结构if let，它可以在某些场景下简化我们处理枚举的代码。

你可以找到许多拥有枚举特性的语言，但它们提供的具体功能却不尽相同。如果一定要比较的话，Rust中的枚举更类似于F#、OCaml和Haskell这类函数式编程语言中的代数数据类型（algebraic data type）。

定义枚举

现在，让我们来尝试处理一个实际的编码问题，并接着讨论在这种情形下，为什么使用枚举要比结构体更加合适。假设我们需要对IP地址进行处理，那么目前有两种被广泛使用的IP地址标准：IPv4和IPv6。因为我们只需要处理这两种情形，所以可以将所有可能的值枚举出来，这也正是枚举名字的由来。

另外，一个IP地址要么是IPv4的，要么是IPv6的，没有办法同时满足两种标准。这个特性使得IP地址非常适合使用枚举结构来进行描述，因为枚举的值也只能是变体中的一个成员。无论是IPv4还是IPv6，它们都属于基础的IP地址协议，所以当我们需要在代码中处理IP地址时，应该将它们视作同一种类型。

我们可以通过定义枚举IpAddrKind来表达这样的概念，声明该枚举需要列举出所有可能的IP地址种类—V4和V6，这也就是所谓的枚举变体（variant）：

```
enum IpAddrKind {
    V4,
    V6,
}
```

现在，IpAddrKind就是一个可以在代码中随处使用的自定义数据类型了。

枚举值

我们可以像下面的代码一样分别使用IpAddrKind中的两个变体来创建实例：

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

需要注意的是，枚举的变体全都位于其标识符的命名空间中，并使用两个冒号来将标识符和变体分隔开来。由于IpAddrKind::V4和IpAddrKind::V6拥有相同的类型IpAddrKind，所以我们可以定义一个接收IpAddrKind类型参数的函数来统一处理它们：

```
fn route(ip_type: IpAddrKind) { }
```

现在，我们可以使用任意一个变体来调用这个函数了：

```
route(IpAddrKind::V4);  
route(IpAddrKind::V6);
```

除此之外，使用枚举还有很多优势。让我们继续考察这个IP地址类型，到目前为止，我们只能知道IP地址的种类，却还没有办法去存储实际的IP地址数据。考虑到你刚刚在第5章学习了结构体，所以你也许会像示例6-1所示的那样去解决这个问题。

```
①enum IpAddrKind {  
    V4,  
    V6,  
}  
  
②struct IpAddr {  
    ③ kind: IpAddrKind,  
    ④ address: String,  
}  
  
⑤let home = IpAddr {  
    kind: IpAddrKind::V4,  
    address: String::from("127.0.0.1"),  
};  
  
⑥let loopback = IpAddr {  
    kind: IpAddrKind::V6,  
    address: String::from("::1"),  
};
```

示例6-1：使用struct来存储IP地址的数据和IpAddrKind变体

上面的代码定义了拥有两个字段的结构体IpAddr②：一个IpAddrKind类型（也就是我们之前定义的枚举①）的字段kind③，以及一个String类型的字段address④。另外，我们还分别创建了两个不同的结构体实例。第一个实例，home⑤，使用了IpAddrKind::V4作为字段kind的值，并存储了关联的地址数据127.0.0.1。第二个实例，loopback⑥，存储了IpAddrKind的另外一个变体V6作为kind的值，并

存储了关联的地址`::1`。新结构体组合了`kind`和`address`的值，现在，变体就和具体数据关联起来了。

实际上，枚举允许我们直接将其关联的数据嵌入枚举变体内。我们可以使用枚举来更简捷地表达出上述概念，而不用将枚举集成至结构体中。在新的`IpAddr`枚举定义中，`V4`和`V6`两个变体都被关联上了一个`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了枚举的每个变体中，这样便不需要额外地使用结构体。

另外一个使用枚举代替结构体的优势在于：每个变体可以拥有不同类型和数量的关联数据。还是以IP地址为例，IPv4地址总是由4个0～255之间的整数部分组成。假如我们希望使用4个`u8`值来代表`V4`地址，并依然使用`String`值来代表`V6`地址，那么结构体就无法轻易实现这一目的了，而枚举则可以轻松地处理此类情形：

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

目前，我们已经为存储IPv4地址及IPv6地址的数据结构给出了好几种不同的方案。但实际上，由于存储和编码IP地址的工作实在太常见了，因此标准库为我们内置了一套可以开箱即用的定义！让我们来看一看标准库是如何设计`IpAddr`的。它采用了和我们自定义一样的枚举和变体定义，但将两个变体中的地址数据各自组装到了两个独立的结构体中：

```
struct Ipv4Addr {
    // --略
}
```

```
struct Ipv6Addr {  
    // --略  
  
    --  
}  
  
enum IpAddr {  
    V4(Ipv4Addr),  
    V6(Ipv6Addr),  
}
```

在这段代码中，你可以在枚举的变体中嵌入任意类型的数据，无论是字符串、数值，还是结构体，甚至可以嵌入另外一个枚举！另外，标准库中的类型通常不会比我们设想的实现要复杂多少。

需要注意的是，虽然标准库中包含了一份IpAddr的定义，但由于我们没有把它引入当前的作用域，所以可以无冲突地继续创建和使用自己定义的版本。我们会在第7章深入讨论作用域引入。

继续来看示例6-2中另外一个关于枚举的例子，它的变体中内嵌了各式各样的数据类型。

```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
    Write(String),  
    ChangeColor(i32, i32, i32),  
}
```

示例6-2：枚举Message的变体拥有不同数量和类型的内嵌数据

这个枚举拥有4个内嵌了不同类型数据的变体：

- Quit没有任何关联数据。
- Move包含了一个匿名结构体。
- Write包含了一个String。
- ChangeColor包含了3个i32值。

定义示例6-2中的枚举有些类似于定义多个不同类型的结构体。但枚举除了不会使用struct关键字，还将变体们组合到了同一个Message类型中。下面代码中的结构体可以存储与这些变体完全一样的数据：

```

    struct QuitMessage; // 空结构体
    struct MoveMessage {
        x: i32,
        y: i32,
    }
    struct WriteMessage(String); // 元组结构体
    struct ChangeColorMessage(i32, i32, i32); // 元组结构体

```

两种实现方式之间的差别在于，假如我们使用了不同的结构体，那么每个结构体都会拥有自己的类型，我们无法轻易定义一个能够统一处理这些类型数据的函数，而我们定义在示例6-2中的Message枚举则不同，因为它是单独的一个类型。

枚举和结构体还有一点相似的地方在于：正如我们可以使用`impl`关键字定义结构体的方法一样，我们同样可以定义枚举的方法。下面的代码在Message枚举中实现了一个名为`call`的方法：

```

impl Message {
    fn call(&self) {
        ① // 方法体可以在这里定义
    }
}

②let m = Message::Write(String::from("hello"));
m.call();

```

方法定义中的代码同样可以使用`self`来获得调用此方法的实例。在这个例子中，我们创建了一个变量 `m②`，并为其赋予了值 `Message::Write(String::from("hello"))`，而该值也就是执行 `m.call()` 指令时传入`call`方法①的`self`。

让我们再来看一看标准库中提供的另外一个非常常见且实用的枚举：`Option`。

Option枚举及其在空值处理方面的优势

在前面几节中，我们看到了`IpAddr`枚举是如何利用Rust的类型系统来将更多的信息，而不仅仅是数据，编码到程序中去的。而本节则会针对性地研究一个定义于标准库中的枚举：`Option`。由于这里的

Option类型描述了一种值可能不存在的情形，所以它被非常广泛地应用在各种地方。将这一概念使用类型系统描述出来意味着，编译器可以自动检查我们是否妥善地处理了所有应该被处理的情况。使用这一功能可以避免某些在其他语言中极其常见的错误。

在设计编程语言时往往会展现出各式各样的功能，但思考应当避免设计哪些功能也是一门非常重要的功课。Rust并没有像许多其他语言一样支持空值。空值（Null）本身是一个值，但它的含义却是没有值。在设计有空值的语言中，一个变量往往处于这两种状态：空值或非空值。

Tony Hoare，空值的发明者，曾经在2009年的一次演讲 *Null References: The Billion Dollar Mistake* 中提到：

这是一个价值数十亿美金的错误设计。当时，我正在为一门面向对象语言中的引用设计一套全面的类型系统。我的目标是，通过编译器自动检查来确保所有关于引用的操作都是百分之百安全的。但是我却没有抵挡住引入一个空引用概念的诱惑，仅仅是因为这样会比较容易去实现这套系统。这导致了无数的错误、漏洞和系统崩溃，并在之后的40多年中造成了价值数10亿美金的损失。

空值的问题在于，当你尝试像使用非空值那样使用空值时，就会触发某种程度上的错误。因为空或非空的属性被广泛散布在程序中，所以你很难避免引起类似的问题。

但是不管怎么说，空值本身所尝试表达的概念仍然是有意义的：它代表了因为某种原因而变为无效或缺失的值。

引发这些问题的关键并不是概念本身，而是那些具体的实现措施。因此，Rust中虽然没有空值，但却提供了一个拥有类似概念的枚举，我们可以用它来标识一个值无效或缺失。这个枚举就是 Option<T>，它在标准库中被定义为如下所示的样子：

```
enum Option<T> {
    Some(T),
    None,
}
```

由于Option<T>枚举非常常见且很有用，所以它也被包含在了预导入模块中，这意味着我们不需要显式地将它引入作用域。另外，它的变体也是这样的：我们可以在不加Option::前缀的情况下直接使用Some或None。但Option<T>枚举依然只是一个普通的枚举类型，Some(T)和None也依然只是Option<T>类型的变体。

这里的语法<T>是一个我们还没有学到的Rust功能。它是一个泛型参数，我们将会在第10章讨论关于泛型的更多细节。现在，你只需要知道<T>意味着Option枚举中的Some变体可以包含任意类型的数据即可。下面是一些使用Option值包含数值类型和字符串类型的示例：

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

假如我们使用了None而不是Some变体来进行赋值，那么我们需要明确地告知Rust这个Option<T>的具体类型。这是因为单独的None变体值与持有数据的Some变体不一样，编译器无法根据这些信息来正确推导出值的完整类型。

当我们有了一个Some值时，我们就可以确定值是存在的，并且被Some所持有。而当我们有了一个None值时，我们就知道当前并不存在一个有效的值。这看上去与空值没有什么差别，那为什么Option<T>的设计就比空值好呢？

简单来讲，因为Option<T>和T（这里的T可以是任意类型）是不同的类型，所以编译器不会允许我们像使用普通值一样去直接使用Option<T>的值。例如，下面的代码在尝试将i8与Option<i8>相加时无法通过编译：

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

运行这段代码，我们可以看到类似下面的错误提示信息：

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
-->
|
```

```
let sum = x + y;
|
|     ^ no implementation for `i8 + std::option::Option<i8>`
```

哇！这段错误提示信息实际上指出了Rust无法理解i8和Option<T>相加的行为，因为它们拥有不同的类型。当我们在Rust中拥有一个i8类型的值时，编译器就可以确保我们所持有的值是有效的。我们可以充满信心地去使用它而无须在使用前进行空值检查。而只有当我们持有的类型是Option<i8>（或者任何可能用到的值）时，我们才必须要考虑值不存在的情况，同时编译器会迫使我们在使用值之前正确地做出处理操作。

换句话说，为了使用Option<T>中可能存在的T，我们必须将它转换为T。一般而言，这能帮助我们避免使用空值时最常见的一个问题：假设某个值存在，实际上却为空。

在编写代码的过程中，不必再去考虑一个值是否为空可以极大地增强我们对自己代码的信心。为了持有一个可能为空的值，我们总是需要将它显式地放入对应类型的Option<T>值中。当我们随后使用这个值的时候，也必须显式地处理它可能为空的情况。无论在什么地方，只要一个值的类型不是Option<T>的，我们就可以安全地假设这个值不是非空的。这是Rust为了限制空值泛滥以增加Rust代码安全性而做出的一个有意为之的设计决策。

那么，当你持有了一个Option<T>类型的Some变体时，你应该怎样将其中的T值取出来使用呢？Option<T>枚举针对不同的使用场景提供了大量的实用方法，你可以在官方文档中找到具体的使用说明。熟练掌握Option<T>的这些方法将为你的Rust之旅提供巨大的帮助。

总的来说，为了使用一个Option<T>值，你必须要编写处理每个变体的代码。某些代码只会在持有Some(T)值时运行，它们可以使用变体中存储的T。而另外一些代码则只会在持有None值时运行，这些代码将没有可用的T值。match表达式就是这么一个可以用来处理枚举的控制流结构：它允许我们基于枚举拥有的变体来决定运行的代码分支，并允许代码通过匹配值来获取变体内的数据。

控制流运算符match

Rust中有一个异常强大的控制流运算符：match，它允许将一个值与一系列的模式相比较，并根据匹配的模式执行相应代码。模式可由字面量、变量名、通配符和许多其他东西组成；第18章会详细介绍所有不同种类的模式及它们的工作机制。match的能力不仅来自模式丰富的表达力，也来自编译器的安全检查，它确保了所有可能的情况都会得到处理。

你可以将match表达式想象成一台硬币分类机：硬币滑入有着不同大小孔洞的轨道，并且掉入第一个符合大小的孔洞。同样，值也会依次通过match中的模式，并且在遇到第一个“符合”的模式时进入相关联的代码块，并在执行过程中被代码所使用。

由于我们正好提到了硬币，所以就用它们来编写一个使用match的示例！示例中的函数会接收一个美国的硬币作为输入，并以一种类似于验钞机的方式，确定硬币的类型并返回它的分值，如示例6-3所示。

```
①enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    ② match coin {
        ③ Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

示例6-3：一个枚举以及一个以枚举变体作为模式的match表达式

让我们先来逐步分析一下函数value_in_cents中的match块。首先，我们使用的match关键字后面会跟随一个表达式，也就是本例中的coin值②。初看上去，这与if表达式的使用十分相似，但这里有个巨大的区别：在if语句中，表达式需要返回一个布尔值，而这里的表达式则可以返回任何类型。例子中coin的类型正是我们在首行①中定义的Coin枚举。

接下来是match的分支，一个分支由模式和它所关联的代码组成。第一个分支采用了值Coin::Penny作为模式，并紧跟着一个=>运算符用于将模式和代码区分开来③。这里的代码简单地返回了值1。不同分支之间使用了逗号分隔。

当这个match表达式执行时，它会将产生的结果值依次与每个分支中的模式相比较。假如模式匹配成功，则与该模式相关联的代码就会被继续执行。而假如模式匹配失败，则会继续执行下一个分支，就像上面提到过的硬币分类机一样。分支可以有任意多个，在示例6-3中，match有4个分支。

每个分支所关联的代码同时也是一个表达式，而这个表达式运行所得到的结果值，同时也会被作为整个match表达式的结果返回。

如果分支代码足够短，就像示例6-3中仅返回一个值的话，那么通常不需要使用花括号。但是，假如我们想要在一个匹配分支中包含多行代码，那么就可以使用花括号将它们包裹起来。例如，下面的代码会在每次给函数传入Coin::Penny时打印“Lucky penny!”，同时仍然返回代码块中最后的值1：

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

绑定值的模式

匹配分支另外一个有趣的地方在于它们可以绑定被匹配对象的部分值，而这也正是我们用于从枚举变体中提取值的方法。

下面举一个例子，让我们修改上面的枚举变体来存放数据。在1999年到2008年之间，美国在25美分硬币的一侧为50个州采用了不同的设计。其他类型的硬币都没有类似的各州的设计，所以只有25美分拥有这个特点。我们可以通过在Quarter变体中添加一个UsState值，来将这些信息添加至枚举中，如示例6-4所示。

```
# [derive(Debug)] // 使我们能够打印并观察各州的设计
```

```
enum UsState {
    Alabama,
    Alaska,
    // --略

    --
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

示例6-4：Coin枚举中的Quarter变体存放了一个UsState值

假设我们有一个朋友正在尝试收集所有50个州的25美分硬币。当我们在根据硬币类型进行大致分类的时候，也可以打印出每个25美分硬币所对应的州的名字。一旦这个朋友发现了没有的硬币，就可以将其加入自己的收藏中。

在这份代码的匹配表达式中，我们在模式中加入了一个叫作state的变量用于匹配变体Coin::Quarter中的值。当匹配到Coin::Quarter时，变量state就会被绑定到25美分所包含的值上。接着，我们就可以在这个分支中像下面一样使用state了：

```
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}", state);
            25
        },
    }
}
```

```
    }  
}
```

如果我们在代码中调用 `value_in_cents(Coin::Quarter(UsState::Alaska))`，`Coin::Quarter(UsState::Alaska)` 就会作为 `coin` 的值传入函数。这个值会依次与每个分支进行匹配，一直到 `Coin::Quarter(state)` 模式才会终止匹配。这时，值 `UsState::Alaska` 就会被绑定到变量 `state` 上。接着，我们就可以在 `println!` 表达式中使用这个绑定了，这就是从 `Coin` 枚举的变体 `Quarter` 中获取值的方法。

匹配 Option<T>

在上一节中，我们曾经想要在使用 `Option<T>` 时，从 `Some` 中取出内部的 `T` 值；现在我们就可以如同操作 `Coin` 枚举一样，使用 `match` 来处理 `Option<T>` 了！除了使用 `Option<T>` 的变体而不是 `Coin` 的变体来进行比较，`match` 表达式的大部分工作流程完全一致。

比如，我们想要编写一个接收 `Option<i32>` 的函数，如果其中有值存在，则将这个值加1。如果其中不存在值，那么这个函数就直接返回 `None` 而不进行任何操作。

得益于 `match` 方法的使用，编写这个函数将会非常简单，它看起来会如示例6-5所示：

```
fn plus_one(x: Option<i32>) -> Option<i32> {  
    match x {  
        ① None => None,  
        ② Some(i) => Some(i + 1),  
    }  
}  
  
let five = Some(5);  
let six = plus_one(five); ❸  
let none = plus_one(None); ❹
```

示例6-5：一个对 `Option<i32>` 使用 `match` 表达式的函数

让我们来分析一下首次执行 `plus_one` 的过程中究竟发生了些什么。当我们调用 `plus_one(five)` ❸ 时，`plus_one` 函数体中的变量 `x` 被绑定为值 `Some(5)`。随后我们会将这个值与各个分支进行比较。

自然，`Some(5)`没办法匹配上模式`None①`，所以我们继续尝试与下一个分支进行比较。`②`这里`Some(5)`会匹配上`Some(i)`吗？答案是肯定的！匹配的两端拥有相同的变体。这里的*i*绑定了`Some`所包含的值，也就是5。接着，这个匹配分支中的代码得到执行，我们将*i*中的值加1，并返回一个新的包含了结果为6的`Some`值。

现在，再让我们来看一看示例6-5中`plus_one`的第二次调用，这一次，`x`变成了`None④`。依然继续进入`match`表达式，并将它与第一个分支`①`进行比较。

它们匹配上了！这里我们没有可用于增加的对象，所以`=>`右侧的程序会简单地终止并返回`None`值。由于第一个分支匹配成功，因此其他的分支会被跳过。

将`match`与枚举相结合在许多情形下都是非常有用的。你会在Rust代码中看到许多类似的套路：使用`match`来匹配枚举值，并将其中的值绑定到某个变量上，接着根据这个值执行相应的代码。这初看起来可能会有些复杂，不过一旦你习惯了它的用法，就会希望在所有的语言中都有这个特性。这一特性一直以来都是社区用户的最爱。

匹配必须穷举所有的可能

`match`表达式中还有另外一个需要注意的特性。你可以先来看下面这个存在bug、无法编译的`plus_one`函数版本：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

此段代码的问题在于我们忘记了处理值是`None`的情形。幸运的是，这是一个Rust可以轻松捕获的问题。假如我们尝试去编译这段代码，就会看到如下所示的错误提示信息：

```
error[E0004]: non-exhaustive patterns: `None` not covered
-->
|
```

```
match x {
|
    ^ pattern `None` not covered
```

Rust知道我们没有覆盖所有可能的情形，甚至能够确切地指出究竟是哪些模式被我们漏掉了！Rust中的匹配是穷尽的(exhaustive)：我们必须穷尽所有的可能性，来确保代码是合法有效的。特别是在这个Option<T>的例子中，Rust会强迫我们明确地处理值为None的情形。这使得我们不需要去怀疑所持有值的存在性，因而可以有效地避免前面提到过的10亿美金的错误。

通配符

有的时候，我们可能并不想要处理所有可能的值，Rust同样也提供了一种模式用于处理这种需求。例如，一个u8可以合法地存储从0到255之间的所有整数。但假设我们只关心值为1、3、5或7的情形，我们就没有必要去列出0、2、4、6、8、9直到255等其余的值。所幸我们也确实可以避免这种情形，即通过使用一个特殊的模式_来替代其余的值：

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

这里的_模式可以匹配任何值。通过将它放置于其他分支后，可以使其帮我们匹配所有没有被显式指定出来的可能的情形。与它对应的代码块里只有一个()空元组，所以在_匹配下什么都不会发生。使用它也就暗示了，我们并不关心那些在_通配符前没有显式列出的情形，且不想为这些情形执行任何操作。

不过，在只关心某一种特定可能的情形下，使用match仍然会显得有些烦琐。为此，Rust提供了if let语句。

简单控制流if let

if let能让我们通过一种不那么烦琐的语法结合使用if与let，并处理那些只用关心某一种匹配而忽略其他匹配的情况。思考一下示例6-6中的程序，它会匹配一个Option<u8>的值，并只在值为3时执行代码。

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

示例6-6：这里的match只在值为Some(3)时执行特定的代码

我们想要对Some(3)的匹配执行某些操作，并忽略其他Some<u8>或None值。为了满足match表达式穷尽性的需求，我们不得不在处理完这唯一的变体后额外加上一句`_ => ()`，这显得十分多余。

不过，我们可以使用if let以一种更加简短的方式实现这段代码。下面的代码与示例6-6中的match拥有完全一致的行为：

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

这里的if let语法使用一对以=隔开的模式与表达式。它们所起的作用与match中的完全相同，表达式对应match中的输入，而模式则对应第一个分支。

使用if let意味着你可以编写更少的代码，使用更少的缩进，使用更少的模板代码。但是，你也放弃了match所附带的穷尽性检查。究竟应该使用match还是if let取决于你当时所处的环境，这是一个在代码简捷性与穷尽性检查之间取舍的过程。

换句话说，你可以将if let视作match的语法糖。它只在值满足某一特定模式时运行代码，而忽略其他所有的可能性。

我们还可以在if let中搭配使用else。else所关联的代码块在if let语句中扮演的角色，就如同match中_模式所关联的代码块一样。还记得我们曾经在示例6-4中定义的Coin枚举吗？里面的Quarter变体包含了一个UsState值。假如我们想要在打印25美分硬币中的信息的同时，对处理过的所有非25美分的硬币进行计数，我们就可以像下面一样使用match表达式：

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

或者我们可以像下面这样使用if let与else表达式：

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

如果你在编写程序的过程中，觉得在某些情形下使用match会过份繁琐，要记得在Rust工具箱中还有if let的存在。

总结

在本章中，我们学会了如何使用枚举来创建自定义类型，它可以包含一系列可被列举的值。我们同时也展示了如何使用标准库中的Option<T>类型，以及它会如何帮助我们利用类型系统去避免错误。当枚举中包含数据时，我们可以使用match或if let来抽取并使用这些值。具体应该使用哪个工具则取决于我们想要处理的情形有多少。

你的Rust程序现在应该可以使用结构体与枚举来表达自己领域中特定的概念了。在API中使用自定义类型同样也可以保证类型安全：编译器会确保函数只会得到它所期望类型的值。

为了向用户提供一个组织良好、使用直观并且只暴露必要部分的API，现在是时候开始学习Rust中的模块系统了。

第7章

使用包、单元包及模块来管理日渐复杂的项目



在编写较为复杂的项目时，合理地对代码进行组织与管理很重要，因为我们不太可能记住代码中所有的细枝末节。只有按照不同的特性来组织或分割相关功能的代码，我们才能够清晰地找到实现指定功能的代码片段，或确定哪些地方需要修改。

到目前为止，我们编写的程序都被放置在了同一个文件下的一个模块中。但随着项目的成熟，你可以将代码拆分为不同的模块并使用不同的文件来管理它们。一个包（package）可以拥有多个二进制单元包及一个可选的库单元包。而随着包内代码规模的增长，你还可以将部分代码拆分到独立的单元包（crate）中，并将它作为外部依赖进行引用。本章便会讲解这些技术。对于那些特别巨大的、拥有多个相互关联的包的项目，Cargo 提供了另外一种解决方案：工作空间（workspace），我们会在第14章的“Cargo工作空间”一节中详细地讨论它。

除了对功能进行分组，对实现的细节进行封装可以使你在更高的层次上复用代码：一旦你实现了某个操作，其他代码就可以通过公共接口来调用这个操作，而无须了解具体的实现过程。我们编写代码的方式决定了哪些部分会作为公共接口供他人使用，而哪些部分又会作为私有的细节实现，使你可以保留进一步修改的权利。这一过程同样使你可以减轻需要记忆在脑海中的心智负担。

另外一个与组织和封装密切相关的概念被称为作用域（scope）：在编写代码的嵌套上下文中有一系列被定义在“作用域内”的名字。当程序员阅读、撰写或编译器编译代码时，都需要借用作用域来确定某个特定区域中的特定名字是否指向了某个变量、函数、结构体、枚举、模块、常量或其他条目，以及这些条目的具体含义。你可以创建作用域并决定某个名字是否处于该作用域中，但是不能在同一作用域中使用相同的名字指向两个不同的条目；有一些工具可以被用来解决命名冲突。

Rust提供了一系列的功能来帮助我们管理代码，包括决定哪些细节是暴露的、哪些细节是私有的，以及不同的作用域内存在哪些名称。这些功能有时被统称为模块系统（module system），它们包括：

- 包（package）：一个用于构建、测试并分享单元包的Cargo功能。
- 单元包（crate）：一个用于生成库或可执行文件的树形模块结构。
- 模块（module）及use关键字：它们被用于控制文件结构、作用域及路径的私有性。
- 路径（path）：一种用于命名条目的方法，这些条目包括结构体、函数和模块等。

我们会在本章介绍上述所有功能，讨论它们之间进行交互的方式，并演示如何使用它们来管理作用域。通过阅读本章，你应该会对模块系统有一个深入的理解，并能够像专家一样熟练地使用作用域！

包与单元包

让我们先来看一看模块系统中有关包与单元包的部分。单元包可以被用于生成二进制程序或库。我们将Rust编译时所使用的入口文件称作这个单元包的根节点，它同时也是单元包的根模块（我们会在随后的“通过定义模块来控制作用域及私有性”一节中详细讨论模块）。而包则由一个或多个提供相关功能的单元包集合而成，它所附带的配置文件*Cargo.toml* 描述了如何构建这些单元包的信息。

有几条规则决定了包可以包含哪些东西。首先，一个包中只能拥有最多一个库单元包。其次，包可以拥有任意多个二进制单元包。最后，包内必须存在至少一个单元包（库单元包或二进制单元包）。

现在，让我们输入命令cargo new，并观察创建一个包时会发生哪些事情：

```
$ cargo new my-project
Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

当我们执行这条命令时，Cargo会生成一个包并创建相应的*Cargo.toml* 文件。观察*Cargo.toml* 中的内容，你也许会奇怪它居然没有提到*src/main.rs*，这是因为Cargo会默认将*src/main.rs* 视作一个二进制单元包的根节点而无须指定，这个二进制单元包与包拥有相同的名称。同样地，假设包的目录中包含文件*src/lib.rs*，Cargo也会自动将其视作与包同名的库单元包的根节点。Cargo会在构建库和二进制程序时将这些单元包的根节点文件作为参数传递给rustc。

最初生成的包只包含源文件 `src/main.rs`，这也意味着它只包含一个名为 `my-project` 的二进制单元包。而假设包中同时存在 `src/main.rs` 及 `src/lib.rs`，那么其中就会分别存在一个二进制单元包与一个库单元包，它们拥有与包相同的名称。我们可以在路径 `src/bin` 下添加源文件来创建出更多的二进制单元包，这个路径下的每个源文件都会被视作单独的二进制单元包。

单元包可以将相关功能分组，并放到同一作用域下，这样便可以使这些功能轻松地在多个项目中共享。例如，我们在第2章使用过的 `rand` 包（`rand crate`）提供了生成随机数的功能。而为了使用这些功能，我们只需要将 `rand` 包引入当前项目的作用域中即可。所有由 `rand` 包提供的功能都可以通过单元包的名称 `rand` 来访问。

将单元包的功能保留在它们自己的作用域中有助于指明某个特定功能来源于哪个单元包，并避免可能的命名冲突。例如，`rand` 包提供了一个名为 `Rng` 的 `trait`，我们同样也可以在自己的单元包中定义一个名为 `Rng` 的 `struct`。正是因为这些功能被放置在了各自的作用域中，当我们将 `rand` 添加为依赖时，编译器才不会为某个 `Rng` 的具体含义是什么而困惑。在我们的单元包中，它指向刚刚定义的 `struct Rng`。我们可以通过 `rand::Rng` 来访问 `rand` 包中的 `Rng trait`。

接着，让我们来聊一聊模块系统。

通过定义模块来控制作用域及私有性

接下来，我们将会讨论模块及模块系统中的其他部分，它们包括可以为条目命名的路径，可以将路径引入作用域的use关键字，以及能够将条目标记为公开的pub关键字。另外，我们还会学习如何使用as关键字、外部项目及通配符。现在，先让我们把注意力集中到模块上！

模块允许我们将单元包内的代码按照可读性与易用性来进行分组。与此同时，它还允许我们控制条目的私有性。换句话说，模块决定了一个条目是否可以被外部代码使用（公共），或者仅仅只是一个内部的实现细节而不对外暴露（私有）。

下面举一个例子，让我们编写一个提供就餐服务的库单元包。为了将注意力集中到代码组织而不是实现细节上，这个示例只会定义函数的签名而省略函数体中的具体内容。

在餐饮业中，店面往往会被划分为前厅与后厨两个部分。其中，前厅会被用于服务客户、处理订单、结账及调酒，而后厨则主要用于厨师与职工们制作料理，以及进行其他一些管理工作。

为了按照餐厅的实际工作方式来组织单元包，可以将函数放置到嵌套的模块中。运行命令cargo new --lib restaurant来创建一个名为restaurant的库，并将示例7-1中的代码输入*src/lib.rs*中来定义一些模块与函数签名。

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7-1：一个含有其他功能模块的front_of_house模块

我们以mod关键字开头来定义一个模块，接着指明这个模块的名字（也就是本例中的front_of_house），并在其后使用一对花括号来包裹模块体。模块内可以继续定义其他模块，如本例中的hosting与serving模块。模块内同样也可以包含其他条目的定义，比如结构体、枚举、常量、trait或如示例7-1中所示的函数。

通过使用模块，我们可以将相关的定义分到一组，并根据它们的关系指定有意义的名称。开发者可以轻松地在此类代码中找到某个定义，因为他们可以根据分组来进行搜索而无须遍历所有定义。开发者可以把新功能的代码按这些模块进行划分并放入其中，从而保持程序的组织结构不变。

我们前面提到过，*src/main.rs* 与 *src/lib.rs* 被称作单元包的根节点，因为这两个文件的内容各自组成了一个名为crate的模块，并位于单元包模块结构的根部。这个模块结构也被称为模块树（module tree）。

示例7-2展示了示例7-1中的树状模块结构。

```
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中代码的树状模块结构

这个树状图展示了模块之间的嵌套关系（比如，hosting被嵌套在front_of_house内）。你还可以观察到，某些模块与其他一些模块是同级的，这也就意味着它们被定义在相同的模块中（比如，hosting与

serving被定义在front_of_house中）。继续使用家庭关系来描述这一现象，当模块A被包含在模块B内时，我们将模块A称作模块B的子节点（child），并将模块B称作模块A的父节点（parent）。注意，整个模块树都被放置在一个名为crate的隐式根模块下。

模块树也许会让你想起文件系统的目录树，实际上这是一个非常恰当的对比！正如文件系统中的目录一样，我们可以使用模块来组织代码；也正如目录中的文件一样，我们也需要对应的方法来定位模块。

用于在模块树中指明条目的路径

类似于在文件系统中使用路径进行导航的方式，为了在Rust的模块树中找到某个条目，我们同样需要使用路径。比如，在调用某个函数时，我们必须要知道它的路径。

路径有两种形式：

- 使用单元包名或字面量crate从根节点开始的绝对路径。
- 使用self、super或内部标识符从当前模块开始的相对路径。

绝对路径与相对路径都由至少一个标识符组成，标识符之间使用双冒号 (::) 分隔。

回到示例7-1中的例子，我们应该如何调用add_to_waitlist函数呢？这个问题实际上等价于：add_to_waitlist函数的路径是什么呢？示例7-3中新定义了一个位于根模块的eat_at_restaurant函数，并在函数体内展示了两种调用add_to_waitlist的方法。因为eat_at_restaurant函数属于公共接口的一部分，所以我们使用了pub关键字来标记它。我们会在“使用pub关键字来暴露路径”一节中详细讨论有关pub的细节。注意，这段代码还无法通过编译，稍后可以看到具体的原因。

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：分别使用绝对路径和相对路径来调用add_to_waitlist函数

eat_at_restaurant第一次调用add_to_waitlist函数时使用了绝对路径。因为add_to_waitlist函数与eat_at_restaurant被定义在相同的单元包中，所以我们可以使用crate关键字来开始一段绝对路径。

在crate之后，我们还填写了一系列连续的模块名称，直到最终的add_to_waitlist。你可以想象一个拥有相同结构的文件系统，这个过程类似于指定路径 front_of_house hosting add_to_waitlist来运行add_to_waitlist程序。使用crate从根节点开始类似于在shell中使用从文件系统根开始。

eat_at_restaurant第二次调用add_to_waitlist时使用了相对路径。这个路径从front_of_house开始，也就是从与eat_at_restaurant定义的模块树级别相同的那个模块名称开始。此时的路径类似于文件系统中的front_of_house hosting add_to_waitlist。以名称开头意味着这个路径是相对的。

你可以基于项目中的实际情况来决定使用相对路径还是绝对路径。这个决定通常取决于你是否会移动条目的定义代码并使用该条目的代码。例如，当我们将front_of_house模块和eat_at_restaurant函数同时移动至一个新的customer_experience模块时，我们就需要更新指向add_to_waitlist的绝对路径，而相对路径则依然有效。而当我们单独将 eat_at_restaurant 移动至 dining 模块时，指向add_to_waitlist的绝对路径会保持不变，但对应的相对路径则需要手动更新。大部分的Rust开发者会更倾向于使用绝对路径，因为我们往往彼此独立地移动代码的定义与调用代码。

现在，让我们试着编译示例7-3中的代码并找出它无法编译的原因！此时产生的错误如示例7-4所示。

```
$ 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();
  |     ^^^^^^
  |
error[E0603]: module `hosting` is private
--> src/lib.rs:12:21
  |
12 |     front_of_house::hosting::add_to_waitlist();
  |     ^^^^^^
```

示例7-4：构建示例7-3中的代码后产生的编译错误

这段错误提示信息指出，模块hosting是私有的。换句话说，虽然我们拥有指向hosting模块及add_to_waitlist函数的正确路径，但由于缺少访问私有域的权限，所以Rust依然不允许我们访问它们。

模块不仅仅被用于组织代码，同时还定义了Rust中的私有边界（privacy boundary）：外部代码无法知晓、调用或依赖那些由私有边界封装了的实现细节。因此，当你想要将一个条目（比如函数或结构体）声明为私有时，你可以将它放置到某个模块中。

Rust中的所有条目（函数、方法、结构体、枚举、模块及常量）默认都是私有的。处于父级模块中的条目无法使用子模块中的私有条目，但子模块中的条目可以使用它所有祖先模块中的条目。虽然子模块包装并隐藏了自身的实现细节，但它却依然能够感知当前定义环境中的上下文。还是使用餐厅作为比喻，你可以将私有性规则想象为餐厅的后勤办公室：其中的工作细节对于餐厅的客户而言自然是不可见的，但后勤经理却依然能够观察并使用自己餐厅中的任何东西。

Rust之所以选择让模块系统这样运作，是因为我们希望默认隐藏内部的实现细节。这样，你就能够明确地知道修改哪些内部实现不会破坏外部代码。同时，你也可以使用pub关键字来将某些条目标记为公共的，从而使子模块中的这些部分被暴露到祖先模块中。

使用pub关键字来暴露路径

让我们回到示例7-4中的错误，它指出hosting模块是私有的。为了让父模块中的eat_at_restaurant函数正常访问子模块中的

`add_to_waitlist`函数，我们可以使用`pub`关键字来标记`hosting`模块，如示例7-5所示。

src/lib.rs

```
mod front_of_house {
    pub 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7-5：将`hosting`模块标记为`pub`以便在`eat_at_restaurant`中使用它

不幸的是，编译示例7-5中的代码依然会导致错误，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:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
--> src/lib.rs:12:30
  |
12 |     front_of_house::hosting::add_to_waitlist();
   |                                     ^^^^^^^^^^^^^^
```

示例7-6：构建示例7-5中的代码后产生的编译错误

究竟发生了什么？在`mod hosting`前面添加`pub`关键字使得这个模块公开了。这一修改使我们在访问`front_of_house`时，可以正常访问`hosting`。但`hosting`中的内容却依旧是私有的。将模块变为公开状态

并不会影响到它内部条目的状态。模块之前的pub关键字仅仅意味着祖先模块拥有了指向该模块的权限。

示例7-6中的错误指出，add_to_waitlist函数是私有的。私有性规则不仅作用于模块，也同样作用于结构体、枚举、函数及方法。

让我们以同样的方式为add_to_waitlist函数添加pub关键字，如示例7-7所示。

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub 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7-7：为mod hosting与fn add_to_waitlist添加的pub关键字使我们可以在eat_at_restaurant中调用这一函数

现在，代码可以通过编译了！在了解了私有性规则后，让我们再来看一看这里的绝对路径与相对路径，并重新检查一下为什么添加的pub关键字能够使我们使用指向add_to_waitlist的路径。

在绝对路径中，我们从crate，也就是单元包的模块树的根节点开始。接着，在根节点中定义front_of_house模块。虽然front_of_house模块并没有被公开，但是因为eat_at_restaurant函数被定义在与front_of_house相同的模块中（也就是说eat_at_restaurant与front_of_house属于同级节点），所以我们可以直接在eat_at_restaurant中引用front_of_house。随后，hosting模块被pub关键字标记。由于我们拥有访问hosting父模块的权利，所以我们也可以访问hosting。最后，add_to_waitlist函数被pub关键字标

记，同样因为我们能够访问它的父模块，所以这个函数能够被正常地访问并调用。

在相对路径中，除了第一步，大部分逻辑都与绝对路径中的相同：相对路径从`front_of_house`开始而不是从单元包的根节点开始。因为`front_of_house`模块被定义在与`eat_at_restaurant`相同的模块下，所以相对路径能够在`eat_at_restaurant`中从这个模块开始寻址。接着，由于`hosting`和`add_to_waitlist`都被标记为了`pub`，所以路径中的其余部分也同样合法，并最终保证函数调用的有效性。

使用super关键字开始构造相对路径

我们同样也可以从父模块开始构造相对路径，这一方式需要在路径起始处使用`super`关键字。它有些类似于在文件系统中使用`..`语法开始一段路径。我们为什么想要这样做呢？

考虑一下示例7-8中涉及的情形：某个大厨需要修正一份错误的订单，并亲自将它送给外面的客户。其中的函数`fix_incorrect_order`通过`super`关键字来指定路径并调用`serve_order`函数。

src/lib.rs

```
fn serve_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }

    fn cook_order() {}
}
```

示例7-8：使用super开头构建相对路径来调用函数

由于`fix_incorrect_order`函数处于`back_of_house`模块内，所以我们能够使用`super`关键字来跳转至`back_of_house`的父模块，也就是根模块处。从它开始，可以成功地找到`serve_order`。考虑到`back_of_house`模块与`serve_order`函数联系较为紧密，当我们需要重新组织单元包的模块树时应该会同时移动它们，所以本例使用了

super。当未来需要将代码移动至其他模块时，可以避免更新这部分相对路径。

将结构体或枚举声明为公共的

结构体与枚举都可以使用pub来声明为公共的，但需要注意其中存在一些细微差别。当我们在结构体定义前使用pub时，结构体本身就成了公共结构体，但它的字段依旧保持了私有状态。我们可以逐一决定是否将某个字段公开。在示例7-9中，我们定义了一个公共的back_of_house::Breakfast结构体，并使它的toast字段公开，而使seasonal_fruit字段保持私有。这段代码描述了餐厅中的早餐模型，客户可以自行选择想要的面包，但只有厨师才能根据季节与存货决定配餐水果。这是因为当前可用的水果总是处于变化中，客户无法选择甚至无法知晓他们能够获得的水果种类。

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() {
    // 选择黑麦面包作为夏季早餐

    let mut meal = back_of_house::Breakfast::summer("Rye");
    // 修改我们想要的面包类型

    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // 接下来的这一行无法通过编译，我们不能看到或更换随着食物附带的季节性水果
}
```

```
// meal.seasonal_fruit = String::from("blueberries");
}
```

示例7-9：一个拥有部分公共字段、部分私有字段的结构体

因为back_of_house::Breakfast结构体中的toast字段是公共的，所以我们才能够在eat_at_restaurant中使用点号读写toast字段。同样由于seasonal_fruit是私有的，所以我们依然不能在eat_at_restaurant中使用它。试着取消上面的那段修改seasonal_fruit字段的代码注释，并看一下会得到什么样的编译错误！

另外还需要注意的是，因为back_of_house::Breakfast拥有了一个私有字段，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（也就是本例中的summer）。如果缺少了这样的函数，我们将无法在eat_at_restaurant中创建任何的Breakfast实例，因为我们不能在eat_at_restaurant中设置私有seasonal_fruit字段的值。

相对应地，当我们将一个枚举声明为公共的时，它所有的变体都自动变为了公共状态。我们仅需要在enum关键字前放置pub，如示例7-10所示。

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：公开一个枚举会同时将它的所有字段公开

因为Appetizer枚举具有公共属性，所以我们能够在eat_at_restaurant中使用Soup与Salad变体。枚举与结构体之所以不同，是由于枚举只有在所有变体都公共可用时才能实现最大的功效，而必须为所有枚举变体添加pub则显得烦琐了一些，因此所有的枚举变

体默认都是公共的。对于结构体而言，即便部分字段是私有的也不会影响到它自身的使用，所以结构体字段遵循了默认的私有性规则，除非被标记为pub，否则默认是私有的。

除了上述情形，本节还遗留了一处与pub有关的使用场景没有介绍，它涉及模块系统的最后一个功能：use关键字。我们会首先介绍use本身，然后再演示如何组合使用pub与use。

用use关键字将路径导入作用域

基于路径来调用函数的写法看上去会有些重复与冗长。例如在示例 7-7 中，无论我们使用绝对路径还是相对路径来指定 add_to_waitlist 函数，都必须在每次调用 add_to_waitlist 的同时指定路径上的节点 front_of_house 与 hosting。幸运的是，有一种方法可以简化该步骤。我们可以借助 use 关键字来将路径引入作用域，并像使用本地条目一样来调用路径中的条目。

示例 7-11 中的代码将 crate::front_of_house::hosting 模块引入了 eat_at_restaurant 函数所处的作用域，从而使我们可以在 eat_at_restaurant 中通过指定 hosting::add_to_waitlist 来调用 add_to_waitlist 函数。

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();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
# fn main() {}
```

示例 7-11：使用 use 将模块引入作用域

在作用域中使用 use 引入路径有些类似于在文件系统中创建符号链接。通过在单元包的根节点下添加 use crate::front_of_house::hosting，hosting 成为了该作用域下的一个

有效名称，就如同hosting模块被定义在根节点下一样。当然，使用use将路径引入作用域时也需要遵守私有性规则。

使用use来指定相对路径稍有一些不同。我们必须在传递给use的路径的开始处使用关键字self，而不是从当前作用域中可用的名称开始。示例7-12中的代码演示了如何使用相对路径来获得与示例7-11中代码相同的行为。

src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use self::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

示例7-12：使用use与以self开头的相对路径来将模块引入作用域

需要注意的是，Rust开发者们正在尝试去掉self前缀，也许在不久的将来我们能够避免在代码中使用它。

创建use路径时的惯用模式

在示例7-11中，你也许会好奇为什么我们使用了use crate::front_of_house::hosting并接着调用hosting::add_to_waitlist，而没有直接使用use来指向add_to_waitlist函数的完整路径，正如示例7-13所示。

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();
    add_to_waitlist();
    add_to_waitlist();
}
```

示例7-13：使用use将add_to_waitlist函数引入作用域的非惯用方式

尽管示例7-11与示例7-13都完成了相同的工作，但相对而言，示例7-11中将函数引入作用域的方式要更加常用一些。使用use将函数的父模块引入作用域意味着，我们必须在调用函数时指定这个父模块，从而更清晰地表明当前函数没有被定义在当前作用域中。当然，这一方式同样也尽可能地避免了重复完整路径。示例7-13中的代码则无法清晰地传达出add_to_waitlist的定义区域。

另一方面，当使用use将结构体、枚举和其他条目引入作用域时，我们习惯于通过指定完整路径的方式引入。示例7-14中的二进制单元包展示了将标准库HashMap结构体引入作用域时的惯用方式。

src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

示例7-14：通过惯用方式将HashMap引入作用域

我们并没有特别强有力的论据来支持这一写法，但它已经作为一种约定俗成的习惯被开发者们接受并应用在阅读和编写Rust代码中了。

当然，假如我们需要将两个拥有相同名称的条目引入作用域，那么就应该避免使用上述模式，因为Rust并不支持这样的情形。示例7-15展示了如何将来自不同模块却拥有相同名称的两个Result类型引入作用域，并分别指向不同的Result。

src/lib.rs

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

fn function1() -> fmt::Result {
    // --略

    --
}

fn function2() -> io::Result<()> {
    // --略

    --
}
```

示例7-15：将两个拥有相同名称的类型引入作用域时需要使用它们的父模块

正如以上代码所示，我们可以使用父模块来区分两个不同的Result类型。但是，假设我们直接指定了use std::fmt::Result与use std::io::Result，那么同一作用域内就会出现两个Result类型，这时Rust便无法在我们使用Result时确定使用的是哪一个Result。

使用as关键字来提供新的名称

使用use将同名类型引入作用域时所产生的问题还有另外一种解决办法：我们可以在路径后使用as关键字为类型指定一个新的本地名称，也就是别名。示例7-16使用了这种方法来编写示例7-15中的代码，它使用as将其中一个Result类型进行了重命名。

src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --略

    --
}

fn function2() -> IoResult<()> {
    // --略

    --
}
```

示例7-16：使用as关键字将引入作用域的类型进行重命名

在第二段use语句中，我们为std::io::Result类型选择了新的名称IoResult，避免了它与同样引入该作用域的std::fmt::Result发生冲突。示例7-15与示例7-16中的写法都是惯用的方法，你可以根据自己的喜好进行选择。

使用pub use重导出名称

当我们使用use关键字将名称引入作用域时，这个名称会以私有的方式在新的作用域中生效。为了让外部代码能够访问到这些名称，我们可以通过组合使用pub与use实现。这项技术也被称作重导出（re-exporting），因为我们不仅将条目引入了作用域，而且使该条目可以被外部代码从新的作用域引入自己的作用域。

示例7-17将示例7-11中根模块下的use修改为了pub use。

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();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

示例7-17：通过pub use使一个名称可以在新作用域中被其他任意代码使用

通过使用pub use，外部代码现在也能够借助路径hosting::add_to_waitlist来调用add_to_waitlist函数了。假设我们没有指定pub use，那么虽然eat_at_restaurant函数能够在自己的作用域中调用hosting::add_to_waitlist，但外部代码则无法访问这一新路径。

当代码的内部结构与外部所期望的访问结构不同时，重导出技术会显得非常有用。例如，在这个餐厅的比喻中，餐厅的员工会以“前厅”和“后厨”来区分工作区域，但访问餐厅的顾客则不会以这样的术语来考虑餐厅的结构。通过使用pub use，我们可以在编写代码时使用一种结构，而在对外部暴露时使用另外一种不同的结构。这一方法可以让我们的代码库对编写者与调用者同时保持良好的组织结构。

使用外部包

我们在第2章编写过一个猜数游戏，并在程序中使用了外部包rand来获得随机数。为了在项目中使用rand，我们需要在*Cargo.toml*中添加下面的内容：

Cargo.toml

```
[dependencies]
rand = "0.5.5"
```

在*Cargo.toml*中添加rand作为依赖会指派Cargo从crates.io上下载rand及相关的依赖包，并使rand对当前的项目可用。

接着，为了将rand定义引入当前包的作用域，我们以包名rand开始添加了一行use语句，并在包名后列出了我们想要引入作用域的条目。回忆一下第2章中的“生成一个随机数”一节，我们当时引入了Rng trait，接着又调用了rand::thread_rng函数：

```
use rand::Rng;
fn main() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

Rust社区的成员已经在crates.io上上传了许多可用的包，你可以按照类似的步骤将它们引入自己的项目：首先将它们列入*Cargo.toml*文件，接着使用use来将特定条目引入作用域。

注意，标准库（std）实际上也同样被视作当前项目的外部包。由于标准库已经被内置到了Rust语言中，所以我们不需要特意修改*Cargo.toml*来包含std。但是，我们同样需要使用use来将标准库中特

定的条目引入当前项目的作用域。例如，我们可以通过如下所示的语句来引入HashMap：

```
use std::collections::HashMap;
```

这段绝对路径以std开头，std是标准库单元包的名称。

使用嵌套的路径来清理众多use语句

当我们想要使用同一个包或同一个模块内的多个条目时，将它们逐行列出会占据较多的纵向空间。例如，猜数游戏中的示例2-4使用了两行use语句来将std中的条目引入作用域：

src/main.rs

```
use std::cmp::Ordering;
use std::io;
// ---略
---
```

然而，我们还可以在同一行内使用嵌套路径来将上述条目引入作用域。这一方法需要我们首先指定路径的相同部分，再在后面跟上两个冒号，接着用一对花括号包裹路径差异部分的列表，如示例7-18所示。

src/main.rs

```
use std::{cmp::Ordering, io};
// ---略
---
```

示例7-18：指定嵌套的路径来将拥有共同路径前缀的条目引入作用域

在一些更复杂的项目里，使用嵌套路径来将众多条目从同一个包或同一个模块引入作用域可以节省大量的独立use语句！

我们可以在路径的任意层级使用嵌套路径，这一特性对于合并两行共享子路径的use语句十分有用。例如，示例7-19展示了两行use语

句：其中一行用于将 std::io 引入作用域，而另一行则用于将 std::io::Write 引入作用域。

src/lib.rs

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

示例7-19：两行使用了use的语句，其中一行是另一行的子路径

这两条路径拥有共同的std::io前缀，该前缀还是第一条路径本身。为了将这两条路径合并至一行use语句中，我们可以在嵌套路途中使用self，如示例7-20所示。

src/lib.rs

```
use std::io::{self, Write};
```

示例7-20：将示例7-19中的路径合并至一行use语句中

上述语句会将std::io与std::io::Write引入作用域。

通配符

假如你想要将所有定义在某个路径中的公共条目都导入作用域，那么可以在指定路径时在后面使用* 通配符：

```
use std::collections::*;


```

上面这行use语句会将定义在std::collections内的所有公共条目都导入当前作用域。请小心谨慎地使用这一特性！通配符会使你难以确定作用域中存在哪些名称，以及某个名称的具体定义位置。

测试代码常常会使用通配符将所有需要测试的东西引入tests模块，我们会在第11章的“如何编写测试”一节来讨论这个话题。通配符还经常被用于预导入模块，你可以阅读官方网站的标准库文档中有 关预导入模块的内容来获得更多信息。

将模块拆分为不同的文件

到目前为止，本章所有的示例都被定义于同一文件内的不同模块中。当模块规模逐渐增大时，我们可以将它们的定义移动至新的文件，从而使代码更加易于浏览。

下面来举一个例子，让我们将示例7-17中的`front_of_house`模块移动至它自己的文件`src/front_of_house.rs`中。这一过程需要修改根节点文件中的代码，如示例7-21所示。在本例中，根节点文件也就是`src/lib.rs`，但这一方法同样也可以被应用到以`src/main.rs`为根节点文件的二进制单元包中。

src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

示例7-21：声明`front_of_house`模块，其代码位于
`src/front_of_house.rs`文件中

将`front_of_house`模块体中的定义移动至`src/front_of_house.rs`文件中，如示例7-22所示。

src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

示例7-22: *src/front_of_house.rs* 文件中front_of_house模块的定义

在mod front_of_house后使用分号而不是代码块会让Rust前往与当前模块同名的文件中加载模块内容。我们可以继续在该示例中进行修改，将hosting模块也移动至它自己的文件中。修改后的*src/front_of_house.rs* 仅仅包含了hosting模块的声明：

src/front_of_house.rs

```
pub mod hosting;
```

接着，创建一个*src/front_of_house* 目录，以及一个名为*src/front_of_house/hosting.rs* 的文件来存放hosting模块中的定义：

src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

所有的修改都没有改变原有的模块树结构，尽管这些定义被放置到了不同的文件中，但eat_at_restaurant中的函数调用依旧有效。该方法使我们可以在模块规模逐渐增大时将它们移动至新的文件中。

注意，*src/lib.rs* 中的 pub use crate::front_of_house::hosting语句同样没有发生变化，use本身也不会影响到编译单元包时会使用的那些文件。我们使用mod关键字声明模块，并指示Rust在同名文件中搜索模块内的代码。

总结

Rust允许你将包拆分为不同的单元包，并将单元包拆分为不同的模块，从而使你能够在其他模块中引用某个特定模块内定义的条目。为了引用外部条目，你需要指定它们的绝对路径或相对路径。我们可以通过use语句将这些路径引入作用域，接着在该作用域中使用较短的路径来多次使用对应的条目。模块中的代码是默认私有的，但你可以通过添加pub关键字来将定义声明为公共的。

在接下来的章节中，我们将会接触到一些来自标准库中的集合数据结构，你可以将它们应用到那些拥有良好组织结构的代码中去。

第8章

通用集合类型



Rust标准库包含了一系列非常有用的数据结构。大部分的数据结构都代表着某个特定的值，但集合却可以包含多个值。与内置的数组与元组类型不同，这些集合将自己持有的数据存储在了堆上。这意味着数据的大小不需要在编译时确定，并且可以随着程序的运行按需扩大或缩小数据占用的空间。不同的集合类型有着不同的性能特性与开销，你需要学会如何为特定的场景选择合适的集合类型。在本章中，我们将讨论以下3个被广泛使用在Rust程序中的集合：

- 动态数组（vector）可以让你连续地存储任意多个值。
- 字符串（string）是字符的集合。我们之前提到过String类型，本章会更为深入地讨论它。
- 哈希映射（hash map）可以让你将值关联到一个特定的键上，它是另外一种数据结构—映射（map）的特殊实现。

对于标准库中的其他集合类型，你可以通过在Rust官方网站查询相关文档来学习。

我们会讨论如何创建和更新动态数组、字符串及哈希映射，并研究它们之间的异同。

使用动态数组存储多个值

我们要学习的第一个集合类型叫作`Vec<T>`，也就是所谓的动态数组。动态数组允许你在单个数据结构中存储多个相同类型的值，这些值会彼此相邻地排布在内存中。动态数组非常适合在需要存储一系列相同类型值的场景中使用，例如文本中由字符组成的行或购物车中的物品价格等。

创建动态数组

我们可以调用函数`Vec::new`来创建一个空动态数组，如示例8-1所示。

```
let v: Vec<i32> = Vec::new();
```

示例8-1：创建一个用来存储i32数据的空动态数组

注意，这段代码显式地增加了一个类型标记。因为我们还没有在这个动态数组中插入任何值，所以Rust无法自动推导出我们想要存储的元素类型。这一点非常重要。动态数组在实现中使用了泛型；我们将第10章学习如何为自定义类型添加泛型。但就目前而言，你只需要知道，标准库中的`Vec<T>`可以存储任何类型的元素，而当你希望某个动态数组持有某个特定的类型时，可以通过一对尖括号来显式地进行声明。示例8-1中的语句向Rust传达了这样的含义：`v`变量绑定的`Vec<T>`会持有`i32`类型的元素。

在实际的编码过程中，只要你向动态数组内插入了数据，Rust便可以在绝大部分情形下推导出你希望存储的元素类型。我们只需要在极少数的场景中对类型进行声明。另外，使用初始值去创建动态数组的场景也十分常见，为此，Rust特意提供了一个用于简化代码的`vec!`

宏。这个宏可以根据我们提供的值来创建一个新的动态数组。示例8-2创建了一个持有初始值1、2、3的Vec<i32>。

```
let v = vec![1, 2, 3];
```

示例8-2：创建一个包含了值的新动态数组

由于Rust可以推断出我们提供的是i32类型的初始值，并可以进一步推断出v的类型是Vec<i32>，所以在这条语句中不需要对类型进行声明。接下来，我们会介绍如何修改一个动态数组。

更新动态数组

为了在创建动态数组后将元素添加至其中，我们可以使用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方法将值添加到动态数组中

正如第3章讨论过的，对于任何变量，只要我们想要改变它的值，就必须使用关键字mut来将其声明为可变的。由于Rust可以从数据中推断出我们添加的值都是i32类型的，所以此处同样不需要添加Vec<i32>的类型声明。

销毁动态数组时也会销毁其中的元素

和其他的struct一样，动态数组一旦离开作用域就会被立即销毁，如示例8-4中的注释所示。

```
{
    let v = vec![1, 2, 3, 4];
    // 执行与v相关的操作
```

```
} // <- v在这里离开作用域并随之被销毁
```

示例8-4：展示了动态数组及其元素销毁的地方

动态数组中的所有内容都会随着动态数组的销毁而销毁，其持有的整数将被自动清理干净。这一行为看上去也许较为直观，但却会在你接触到指向动态数组元素的引用时变得有些复杂。让我们接着来处理这种情况！

读取动态数组中的元素

现在，你应该已经学会了如何去创建、更新及销毁动态数组，接下来就该了解如何读取其中的内容了。有两种方法可以引用存储在动态数组中的值。为了更加清晰地说明问题，我们在下面的示例中标记出了函数返回值的类型。

示例8-5展示了两种访问动态数组的方式，它们分别是使用索引和get方法。

```
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2];

println!("The third element is {}", third);
match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

示例8-5：使用索引或get方法来访问动态数组中的元素

这里有两个需要注意的细节。首先，我们使用索引值2获得的是第三个值：动态数组使用数字进行索引，索引值从零开始。其次，使用`&`与`[]`会直接返回元素的引用；而接收索引作为参数的`get`方法则会返回一个`Option<&T>`。

当你尝试使用对应元素不存在的索引值去读取动态数组时，因为Rust提供了两种不同的元素引用方式，所以你能够自行选择程序的响应方式。比如，示例8-6中创建的动态数组持有5个元素，但它却尝试

着访问数组中索引值为100的元素，让我们来看一下这种行为会导致什么样的后果。

```
let v = vec![1, 2, 3, 4, 5];  
let does_not_exist = &v[100];  
let does_not_exist = v.get(100);
```

示例8-6：尝试在只有5个元素的动态数组中访问索引值为100的元素

当我们运行这段代码时，[]方法会因为索引指向了不存在的元素而导致程序触发panic。假如你希望在尝试越界访问元素时使程序直接崩溃，那么这个方法就再适合不过了。

get方法会在检测到索引越界时简单地返回None，而不是使程序直接崩溃。当偶尔越界访问动态数组中的元素是一个正常行为时，你就应该使用这个方法。另外，正如在第6章讨论的那样，你的代码应该合乎逻辑地处理Some (&element) 与None两种不同的情形。例如，索引可能来自一个用户输入的数字。当这个数字意外地超出边界时，程序就会得到一个None值。而我们也应该将这一信息反馈给用户，告诉他们当前动态数组的元素数量，并再度请求用户输入有效的值。这就比因为输入错误而使程序崩溃要友好得多！

如同在第4章讨论过的那样，一旦程序获得了一个有效的引用，借用检查器就会执行所有权规则和借用规则，来保证这个引用及其他任何指向这个动态数组的引用始终有效。回忆一下所有权规则，我们不能在同一个作用域中同时拥有可变引用与不可变引用。示例8-7便遵循了该规则。在这个例子中，我们持有了一个指向动态数组中首个元素的不可变引用，但却依然尝试向这个动态数组的结尾处添加元素，该尝试是不会成功的。

```
let mut v = vec![1, 2, 3, 4, 5];  
let first = &v[0];  
  
v.push(6);  
println!("The first element is: {}", first);
```

示例8-7：在存在指向动态数组元素的引用时尝试向动态数组中添加元素

编译这段代码将会导致下面的错误：

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
-->
|
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
```

你也许不会觉得示例8-7中的代码有什么问题：为什么对第一个元素的引用需要关心动态数组结尾处的变化呢？此处的错误是由动态数组的工作原理导致的：动态数组中的元素是连续存储的，插入新的元素后也许会没有足够多的空间将所有元素依次相邻地放下，这就需要分配新的内存空间，并将旧的元素移动到新的空间上。在本例中，第一个元素的引用可能会因为插入行为而指向被释放的内存。借用规则可以帮助我们规避这类问题。

注意

你可以查看 *The Rustonomicon* 中的相关内容来了解更多`Vec<T>`的实现细节。

遍历动态数组中的值

假如你想要依次访问动态数组中的每一个元素，那么可以直接遍历其所有元素，而不需要使用索引来一个一个地访问它们。示例8-8展示了如何使用`for`循环来获得动态数组中每一个`i32`元素的不可变引用，并将它们打印出来。

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

示例8-8：使用for循环遍历并打印出动态数组中的所有元素

我们同样也可以遍历可变的动态数组，获得元素的可变引用，并修改其中的值。示例8-9中的for循环会让动态数组中的所有元素的值增加50。

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

示例8-9：遍历动态数组中所有元素的可变引用

为了使用`+=`运算符来修改可变引用指向的值，我们首先需要使用解引用运算符`(*)`来获得`i`绑定的值。我们会在第15章的“使用解引用运算符跳转到指针指向的值”一节中进一步讨论解引用运算符。

使用枚举来存储多个类型的值

在本章开始的时候，我们曾经提到过动态数组只能存储相同类型的值。这个限制可能会带来不小的麻烦，实际工作中总是会碰到需要存储一些不同类型值的情况。幸运的是，当我们需要在动态数组中存储不同类型的元素时，可以定义并使用枚举来应对这种情况，因为枚举中的所有变体都被定义为了同一种枚举类型。

假设我们希望读取表格中的单元值，这些单元值可能是整数、浮点数或字符串，那么就可以使用枚举的不同变体来存放不同类型的值。所有的这些枚举变体都会被视作统一的类型：也就是这个枚举类型。接着，我们便可以创建一个持有该枚举类型的动态数组来存放不同类型的值，如示例8-10所示。

```
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8-10：在动态数组中使用定义的枚举来存储不同类型的值

为了计算出元素在堆上使用的存储空间，Rust需要在编译时确定动态数组的类型。使用枚举的另一个好处在于它可以显式地列举出所有可以被放入动态数组的值类型。假如Rust允许动态数组存储任意类型，那么在对动态数组中的元素进行操作时，就有可能会因为一个或多个不当的类型处理而导致错误。将枚举和match表达式搭配使用意味着，Rust可以在编译时确保所有可能的情形都得到妥当的处理，正如在第6章讨论过的那样。

假如你没有办法在编写程序时穷尽所有可能出现在动态数组中的值类型，那么就无法使用枚举。为了解决这一问题，我们需要用到在第17章会介绍的动态trait。

现在，我们已经学会了一些常见的使用动态数组的方法，但请你一定要去看一下标准库中有关Vec<T>的API文档，它包含了Vec<T>所有方法的详细说明。例如，除了push，还有一个pop方法可以移除并返回末尾的元素。接下来，让我们来继续学习下一个集合类型：String！

使用字符串存储UTF-8编码的文本

我们曾经在第4章提到过字符串，现在终于可以来深入地讨论它了。刚刚接触Rust的开发者们十分容易在使用字符串时出现错误，这是由3个因素共同作用造成的：首先，Rust倾向于暴露可能的错误；其次，字符串是一个超乎许多编程者想象的复杂数据结构；最后，Rust中的字符串使用了UTF-8编码。假如你曾经使用过其他编程语言，那么这些因素组合起来也许会让你感到有些困惑。

之所以要将字符串放在集合章节中来学习，是因为字符串本身就是基于字节的集合，并通过功能性的方法将字节解析为文本。本节将会介绍一些常见的基于String的集合类型的操作，比如创建、更新及访问等。我们也会讨论String与其他集合类型不同的地方，比如，尝试通过索引访问String中的字符往往是十分复杂的，这是因为人和计算机对String数据的解释方式不同。

字符串是什么

我们先来定义一下术语字符串 的具体含义。Rust在语言核心部分只有一种字符串类型，那就是字符串切片str，它通常以借用的形式(&str)出现。正如在第4章讨论的那样，字符串切片是一些指向存储在别处的UTF-8编码字符串的引用。例如，字符串字面量的数据被存储在程序的二进制文件中，而它们本身也是字符串切片的一种。

String类型被定义在了Rust标准库中而没有被内置在语言的核心部分。当Rust开发者们提到“字符串”时，他们通常指的是String与字符串切片&str这两种类型，而不仅仅只是其中的一种。虽然本节会着重介绍String，但是这两种类型都广泛地被应用于Rust标准库中，并且都采用了UTF-8编码。

Rust的标准库中同时包含了其他一系列的字符串类型，比如OsString、OsStr、CString及CStr。某些第三方库甚至还提供了更多用于存储字符串数据的选择。注意到这些名字全都以String或Str结尾了吗？这用来表明类型提供的是所有者版本还是借用者版本，正如你之前所看到的String和str类型一样。这些字符串类型可以使用不同的编码，或者不同的内存布局来存储文本。我们不会在本章讨论这些类型，但你可以通过查看它们的API文档来学习如何使用这些字符串，并了解各自最佳的使用场景。

创建一个新的字符串

许多对于Vec<T>可用的操作也同样可用于String，我们可以从new函数开始来创建一个字符串，如示例8-11所示。

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

示例8-11：创建一个新的空字符串

这行代码创建了一个叫作s的空字符串，之后我们可以将数据填入该字符串。但是一般而言，字符串在创建的时候都会有一些初始数据。对于这种情况，我们可以对那些实现了Display trait的类型调用to_string方法，如同字符串字面量一样。示例8-12中展示了两个例子。

```
let data = "initial contents";
let s = data.to_string();
// 这个方法同样也可以直接作用于字面量
:
let s = "initial contents".to_string();
```

示例8-12：使用to_string方法基于字符串字面量创建String

这段代码所创建的字符串会拥有initial contents作为内容。

我们同样也可以使用函数String::from来基于字符串字面量生成String。示例8-13中的代码等价于示例8-12中使用to_string的代码。

```
let s = String::from("initial contents");
```

示例8-13：使用String::from函数基于字符串字面量创建String

由于字符串被如此广泛地使用，因此在它的实现中提供了许多不同的通用API供我们选择。某些函数初看起来也许会有些多余，但是请相信它们自有妙用。在以上的例子中，String::from和to_string实际上完成了相同的工作，你可以根据自己的喜好来选择使用哪种方法。

记住，字符串是基于UTF-8编码的，我们可以将任何合法的数据编码进字符串中，如示例8-14所示。

```
let hello = String::from("عليكم السلام");
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：存储在字符串中的不同语言的问候

所有这些问候短语都是合法的String值。

更新字符串

String的大小可以增减，其中的内容也可以修改，正如我们将数据推入其中时Vec<T>内部数据所发生的变化一样。此外，我们还可以方便地使用+运算符或format! 宏来拼接String。

使用push_str 或push 向字符串中添加内容

我们可以使用push_str方法来向String中添加一段字符串切片，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中添加字符串切片

执行完上面的代码后，s中的字符串会被更新为foobar。由于我们并不需要取得参数的所有权，所以这里的push_str方法只需要接收一个字符串切片作为参数。你可以想象一下，在示例8-16中，如果s2在拼接至s1后再也无法使用了该是多么不方便。

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}", s2);
```

示例8-16：在将字符串切片附加至String后继续使用它

假如push_str方法取得了s2的所有权，那么我们就无法在最后一行打印出它的值了。好在这些代码如期运行了！

push方法接收单个字符作为参数，并将它添加到String中。示例8-17展示了如何使用push方法向String的尾部添加字符l。

```
let mut s = String::from("lo");
s.push('l');
```

示例8-17：使用push方法将一个字符添加到String中

这段代码执行完毕后，s中的内容会变为lol。

使用+运算符或format! 宏来拼接字符串

你也许经常需要在代码中将两个已经存在的字符串组合在一起。一种办法是像示例8-18那样使用+运算符。

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // 注意这里的s1已经被移动且再也不能被使用了
```

示例8-18：使用+运算符将两个String合并到一个新的String中

执行完这段代码后，字符串s3中的内容会变为Hello, world!。值得注意的是，我们在加法操作中仅对s2采用了引用，而s1在加法操作之后则不再有效。产生这一现象的原因与使用+运算符时所调用的方法签名有关。这里的+运算符会调用一个add方法，它的签名看起来像下面一样：

```
fn add(self, s: &str) -> String {
```

当然，这与标准库中实际的签名有些许差别：在标准库中，add函数使用了泛型来进行定义。此处展示的add函数将泛型替换为了具体的类型，这是我们使用String值调用add时使用的签名。我们将在第10章继续讨论泛型。这个签名应该能够帮助你理解+运算符中的微妙之处。

首先，代码中的s2使用了&符号，这意味着我们实际上是将第二个字符串的引用与第一个字符串相加了，正如add函数中的s参数所指明的那样：我们只能将&str与String相加，而不能将两个String相加。但是等等，&s2的类型是&String，而add函数中的第二个参数的类型则是&str。为什么示例8-18依然能够通过编译呢？

我们能够使用&s2来调用add函数的原因在于：编译器可以自动将&String类型的参数强制转换为&str类型。当我们调用add函数时，Rust使用了一种被称作解引用强制转换的技术，将&s2转换为了&s2[...]。我们将在第15章更加深入地讨论解引用强制转换这一概念。由于add并不会取得函数签名中参数s的所有权，因此变量s2将在执行这一操作后依旧保留一个有效的String值。

其次，我们可以看到add函数签名中的self并没有&标记，所以add函数会取得self的所有权。这也意味着示例8-18中的s1将会被移动至add函数调用中，并在调用后失效。所以，即便let s3 = s1 + &s2;看起来像是复制两个字符串并创建一个新的字符串，但实际上这条语句会取得s1的所有权，再将s2中的内容复制到其中，最后再将s1的所有权作为结果返回。换句话说，它看起来好像进行了很多复制，但实际上并没有，这种实现要比单纯的复制更加高效。

假如你需要拼接多个字符串，那么使用+运算符可能就会显得十分笨拙了：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
```

```
let s = s1 + "-" + &s2 + "-" + &s3;
```

本例中s的内容将是tic-tac-toe。在有这么多+及”字符的情况下，你很难去分析其中的具体实现。对于这种复杂一些的字符串合并，我们可以使用format! 宏：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3);
```

这段代码同样也会在s中生成tic-tac-toe。format! 宏与println! 宏的工作原理完全相同，不过不同于println! 将结果打印至屏幕，format! 会将结果包含在一个String中返回。这段使用format! 的代码要更加易读，并且不会夺取任何参数的所有权。

字符串索引

在许多编程语言中，往往可以合法地通过索引来引用字符串中每一个单独的字符。但不管怎样，假如你在Rust中尝试使用同样的索引语法去访问String中的内容，则会收到一个错误提示。下面来看一下示例8-19中的这段非法代码。

```
let s1 = String::from("hello");
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：

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>` is
not satisfied
-->
|
3 | 
|   let h = s1[0];
| 
|     ^^^^^^ the type `std::string::String` cannot be indexed by `{integer}`
```

```
= help: the trait `std::ops::Index<{integer}>` is not implemented for  
`std::string::String`
```

这里的错误日志和提示信息说明了其中的缘由：Rust中的字符串并不支持索引。但是为什么不支持呢？为了回答这个问题，我们接着来看一下Rust是如何在内存中存储字符串的。

内部布局

String实际上是一个基于Vec<u8>的封装类型。下面来看一些示例8-14中的UTF-8编码的字符串的例子。首先来看下面这个：

```
let len = String::from("Hola").len();
```

在这行代码中，len方法将会返回4，这意味着动态数组所存储的字符串Hola占用了4字节。在编码为UTF-8时，每个字符都分别占用1字节。那么，下面这个例子是否也符合这样的规律呢？（注意，这个字符串中的首字母是西里尔字母中的Ze，而不是阿拉伯数字3。）

```
let len = String::from("Здравствуйте").len();
```

首先来猜一下这个字符串的长度，你给出的答案也许是12。但实际上，Rust返回的结果是24：这就是使用UTF-8编码来存储“З д р а в с т в у й т е”所需要的字节数，因为这个字符串中的每个Unicode标量值都需要占据2字节。发现了吧，对字符串中字节的索引并不总是能对应到一个有效的Unicode标量值。为了演示这一行为，让我们来看一看下面这段非法的Rust代码：

```
let hello = "Здравствуйте";  
let answer = &hello[0];
```

这段代码中的answer值会是多少呢？它应该是首字母3吗？当使用UTF-8编码时，3依次使用了208、151两字节空间，所以这里的answer应该是208吧，但208本身却又不是一个合法的字符。请求字符串中首字母的用户可不会希望获得一个208的返回值，可这又偏偏是Rust在索引0处取到的唯一字节数据。用户想要的结果通常不会是一个字节值，即便这个字符串只由拉丁字母组成：如果我们将“hello”[0]视作合法的代码，那么它会返回一个字节值104，而不是h。为了避免返回意想不到的值，以及出现在运行时才会暴露的错误，Rust会直接拒绝编译这段代码，在开发阶段提前杜绝可能的误解。

字节、标量值及字形簇！天呐！

使用UTF-8编码还会引发另外一个问题。在Rust中，我们实际上可以通过3种不同的方式来看待字符串中的数据：字节、标量值和字形簇（最接近人们眼中字母的概念）。

假如我们尝试存入一个使用梵文书写的印度语单词“नमस्ते”，那么该单词在动态数组中存储的u8值看起来会像下面一样：

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165,  
135]
```

这里有18字节，也是计算机最终存储数据的样子。假如我们将它们视作Unicode标量值，也就是Rust中的char类型，那么这些字节看起来会像是：

```
['न', 'म', 'स', '्', 'त', 'े']
```

这里有6个char值，但实际上第四个与第六个并不能算作字母：它们本身没有任何意义，只是作为音标存在。最后，假如我们将它们视作字形簇，就会得到通常意义上的印度语字符：

```
["न", "म", "स्", "ते"]
```

Rust中提供了不同的方式来解析存储在计算机中的字符串数据，以便于程序员们自行选择所需的解释方式，而不用关心具体的语言类型。

Rust不允许我们通过索引来获得String中的字符还有最后一个原因，那就是索引操作的复杂度往往会被预期为常数时间($O(1)$)。但在String中，我们无法保障这种做法的性能，因为Rust必须要遍历从头至索引位置的整个内容来确定究竟有多少合法的字符存在。

字符串切片

尝试通过索引引用字符串通常是一个坏主意，因为字符串索引操作应当返回的类型是不明确的：究竟应该是字节，还是字符，或是字形簇，甚至是字符串切片呢？因此，如果真的想要使用索引来创建字符串切片，Rust会要求你做出更加明确的标记。为了明确表明需要一个字符串切片，你需要在索引的[]中填写范围来指定所需的字节内容，而不是在[]中使用单个数字进行索引：

```
let hello = "Здравствуйте";  
let s = &hello[0..4];
```

在这段代码中，s将会是一个包含了字符串前4字节的&str。前面曾提到过，这里的每个字符都会占据2字节，这也意味着s中的内容将是 З д。

假如我们在这里尝试使用&hello[0..1]会发生什么呢？答案是，Rust会如同我们在动态数组中使用非法索引时一样，在运行时发生 panic。

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside 'З'  
(bytes 0..2) of `Здравствуйте`',  
src/libcore/str/mod.rs:2188:4
```

切记要小心谨慎地使用范围语法创建字符串切片，因为错误的指令会导致程序崩溃。

遍历字符串的方法

幸运的是，还有其他访问字符串中元素的方法。

假如你想要对每一个Unicode标量值都进行处理，那么最好的办法就是使用chars方法。针对字符串“नमस्ते”调用chars会分别返回6个类型为char的值，接着就可以遍历这个结果来访问每个元素了：

```
for c in "नमस्ते"  
.chars() {  
    println!("{}", c);  
}
```

这段代码的输出如下所示：

न
म
स

、
त

而bytes方法则会依次返回每个原始字节，这在某些场景下可能会有用：

```
for b in "नमस्ते"  
    .bytes() {  
        println!("{}", b);  
    }
```

这段代码会打印出组成这个String的18个字节值：

```
224  
164  
// --略  
  
--  
165  
135
```

但是请记住，合法的Unicode标量值可能会需要占用1字节以上的空间。

从字符串中获取字形簇相对复杂一些，所以标准库中也没有提供这个功能。如果你有这方面的需求，那么可以在crates.io上获取相关的开源库。

字符串的确没那么简单

总而言之，字符串确实是挺复杂的。不同的编程语言会做出不同的设计抉择，来确定将何种程度的复杂性展现给程序员。Rust选择了将正确的String数据处理方法作为所有Rust程序的默认行为，这也就意味着程序员需要提前理解UTF-8数据的处理流程。与某些编程语言相比，这一设计暴露了字符串中更多的复杂性，但它也避免了我们在开发周期临近结束时再去处理那些涉及非ASCII字符的错误。

下面学习的这个集合要稍微简单一些，它就是哈希映射！

在哈希映射中存储键值对

我们将要学习的最后一个集合类型就是哈希映射：`HashMap<K, V>`，它存储了从`K`类型键到`V`类型值之间的映射关系。哈希映射在内部实现中使用了哈希函数，这同时决定了它在内存中存储键值对的方式。许多编程语言都支持这种类型的数据结构，只是使用了不同的名字，例如：哈希（hash）、映射（map）、对象（object）、哈希表（hash table）、字典（dictionary）或关联数组（associative array）等，这只是其中的一部分而已。

当你不仅仅满足于使用索引—就像是动态数组那样，而需要使用某些特定的类型作为键来搜索数据时，哈希映射就会显得特别有用。例如，在一个游戏中，你可以将团队的名字作为键，将团队获得的分数作为值，并将所有队伍的分数存放在哈希映射中。随后只要给出一个队伍的名称，你就可以获得当前的分数值。

我们会在本节介绍一些哈希映射的常用API，但是，此处无法覆盖标准库为`HashMap<K, V>`定义的全部有趣的功能。通常，你可以通过查阅标准库文档来获得更多信息。

创建一个新的哈希映射

你可以使用`new`来创建一个空哈希映射，并通过`insert`方法来添加元素。在示例8-20中，我们记录了两支队伍的分数，它们分别被称作蓝队和黄队。蓝队的起始分数为10分，而黄队的起始分数为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：创建一个新的哈希映射并插入一些键值对

注意，我们首先需要使用use将HashMap从标准库的集合部分引入当前作用域。由于哈希映射的使用频率相比于本章介绍的其他两个集合低一些，所以它没有被包含在预导入模块内。标准库对哈希映射的支持也不如另外两个集合，例如它没有提供一个可以用于构建哈希映射的内置宏。

和动态数组一样，哈希映射也将其数据存储在堆上。上面例子中的HashMap拥有类型为String的键，以及类型为i32的值。依然和动态数组一样，哈希映射也是同质的：它要求所有的键必须拥有相同的类型，所有的值也必须拥有相同的类型。

另外一个构建哈希映射的方法是，在一个由键值对组成的元组动态数组上使用collect方法。这里的collect方法可以将数据收集到很多数据结构中，这些数据结构也包括HashMap。例如，假设我们在两个不同的动态数组里分别存储了队伍的名字和分数，那么我们就可以使用zip方法来创建一个元组的数组，其中第一个元组由"Blue"与10组成，以此类推。接着，我们还可以使用collect方法来将动态数组转换为哈希映射，如示例8-21所示。

```
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let scores: HashMap<_, _> =
teams.iter().zip(initial_scores.iter()).collect();
```

示例8-21：使用队伍列表和分数列表创建哈希映射

这里的类型标记HashMap<_, _>不能被省略，因为collect可以作用于许多不同的数据结构，如果不指明类型的话，Rust就无法知道我们具体想要的类型。但是对于键值的类型参数，我们则使用了下画线占位，因为Rust能够根据动态数组中的数据类型来推导出哈希映射所包含的类型。

哈希映射与所有权

对于那些实现了Copy trait的类型，例如i32，它们的值会被简单地复制到哈希映射中。而对于String这种持所有权的值，其值将会转移且所有权会转移给哈希映射，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);
// field_name和field_value从这一刻开始失效，若尝试使用它们则会导致编译错误！
```

示例8-22：一旦键值对被插入，其所有权就会转移给哈希映射

在调用insert方法后，field_name和field_value变量被移动到哈希映射中，我们再也没有办法使用这两个变量了。

假如我们只是将值的引用插入哈希映射，那么这些值是不会被移动到哈希映射中的。这些引用所指向的值必须要保证，在哈希映射有效时自己也是有效的。我们会在第10章的“使用生命周期保证引用的有效性”一节中详细地讨论这个问题。

访问哈希映射中的值

我们可以通过将键传入get方法来获得哈希映射中的值，如示例8-23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);
```

示例8-23：访问存储在哈希映射中的蓝队分数

上面这段代码中的score将会是与蓝队相关联的值，也就是Some(&10)。因为get返回的是一个Option<&V>，所以这里的结果被封装到了Some中；假如这个哈希映射中没有键所对应的值，那么get就会

返回None。接下来，程序需要使用我们在第6章讨论过的方法来处理这个Option。

类似于动态数组，我们同样可以使用一个for循环来遍历哈希映射中所有的键值对：

```
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);
}
```

这段代码会将每个键值对以不特定的顺序打印出来：

```
Yellow: 50
Blue: 10
```

更新哈希映射

尽管键值对的数量是可以增长的，但是在任意时刻，每个键都只能对应一个值。当你想要修改哈希映射中的数据时，你必须处理某些键已经被关联到值的情况。你可以完全忽略旧值，并用新值去替换它。你也可以保留旧值，只在键没有对应值时添加新值。或者，你还可以将新值与旧值合并到一起。让我们来看一看应该如何分别处理这些情况！

覆盖旧值

当我们将在一个键值对插入哈希映射后，接着使用同样的键并配以不同的值来继续插入，之前的键所关联的值就会被替换掉。即便示例8-24中的代码调用了两次insert，这里的哈希映射也依然只会包含一个键值对，因为我们插入值时所用的键是一样的。

```
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8-24：替换使用特定键存储的值

原来的值10已经被覆盖掉了，这段代码会打印出{"Blue": 25}。

只在键没有对应值时插入数据

在实际工作中，我们常常需要检测一个键是否存在对应值，如果不存在，则为它插入一个值。哈希映射中提供了一个被称为entry的专用API来处理这种情形，它接收我们想要检测的键作为参数，并返回一个叫作Entry的枚举作为结果。这个枚举指明了键所对应的值是否存在。比如，我们想要分别检查黄队、蓝队是否拥有一个关联的分数值，如果该分数值不存在，就将50作为初始值插入。使用entry API的代码如示例8-25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8-25：通过使用entry方法在键不存在对应值时插入数据

Entry的or_insert方法被定义为返回一个Entry键所指向值的可变引用，假如这个值不存在，就将参数作为新值插入哈希映射中，并把这个新值的可变引用返回。使用这个功能要比我们自己编写逻辑代码更加简单，使代码更加整洁，另外也可以与借用检查器结合得更好。

运行示例8-25中的代码将会打印出{"Yellow": 50, "Blue": 10}。由于黄队的比分还不存在，所以第一个对entry的调用会将分数50插入哈希映射中；而由于蓝队已经存储了比分10，所以第二个对entry的调用不会改变哈希映射。

基于旧值来更新值

哈希映射的另外一个常见用法是查找某个键所对应的值，并基于这个值来进行更新。比如，示例8-26中的代码用于计算一段文本中每个单词所出现的次数。我们使用了一个以单词作为键的哈希映射来记录它们所出现的次数。在遍历的过程中，假如出现了一个新的单词，我们就先将值0插入哈希映射中。

```
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8-26：使用哈希映射来存储并计算单词出现的次数

运行这段代码会输出 {"world": 2, "hello": 1, "wonderful": 1}。代码中的方法`or_insert`实际上为我们传入的键返回了一个指向关联值的可变引用（`&mut V`）。这个可变引用进而被存储到变量`count`上，为了对这个值进行赋值操作，我们必须首先使用星号（*）来对`count`进行解引用。由于这个可变引用会在`for`循环的结尾处离开作用域，所以我们在代码中的所有修改都是安全且满足借用规则的。

哈希函数

为了提供抵御拒绝服务攻击（DoS，Denial of Service）的能力，`HashMap`默认使用了一个在密码学上安全的哈希函数。这确实不是最快的哈希算法，不过为了更高的安全性付出一些性能代价通常是值得的。假如你在对代码进行性能分析的过程中，发现默认哈希函数成为了你的性能热点并导致性能受损，你也可以通过指定不同的哈希计算工具 来使用其他函数。这里的哈希计算工具特指实现了`BuildHasher trait`的类型。我们会在第10章讨论`trait`，以及如何实现它们。你并不一定非要从头实现自己的哈希工具，Rust开发者们已经在crates.io上分享了许多基于不同哈希算法的开源项目。

总结

动态数组、字符串及哈希映射为我们提供了很多用于存储、访问或修改数据的功能，你可以非常方便地将它们应用到自己的程序中。这里给出了一些小问题，你可以尝试独立解决它们来练习在本章中学到的知识：

- 给定一组整数，使用动态数组来计算该组整数中的平均数、中位数（对数组进行排序后位于中间的值）及众数（出现次数最多的值；哈希映射可以在这里帮上忙）。
- 将给定字符串转换为Pig Latin格式。在这个格式中，每个单词的第一个辅音字母会被移动到单词的结尾并增加“ay”后缀，例如“first”就会变为“irst-fay”。元音字母开头的单词则需要在结尾拼接上“hay”（例如，“apple”就会变为“apple-hay”）。要牢记我们讨论的关于UTF-8编码的内容！
- 使用哈希映射和动态数组来创建一个添加雇员名字到公司部门的文本接口。例如，“添加Sally至项目部门”或“添加Amir至销售部门”。除此之外，该文本接口还应该允许用户获得某个部门所有员工或公司中所有部门员工的列表，列表按照字母顺序进行排序。

这里有个小提示：标准库中关于动态数组、字符串和哈希映射的API文档会有助于你解决这些问题！

我们已经开始接触到一些可能会导致操作失败的复杂程序了，现在正是讨论如何进行错误处理的绝佳时机。让我们继续学习下一章吧！

第9章

错误处理



Rust对可靠性的执着同样延伸到了错误处理领域。为了应对软件中那些几乎无法避免的错误，Rust提供了许多特性来处理这类出了问题的场景。在大部分情形下，Rust会迫使你意识到可能出现错误的地方，并在编译阶段确保它们得到妥善的处理。这些特性使你能够在将代码最终部署到生产环境之前，发现并合理地处理错误，从而使程序更加健壮！

在Rust中，我们将错误分为两大类：可恢复 错误与不可恢复 错误。对于可恢复错误，比如文件未找到等，一般需要将它们报告给用户并再次尝试进行操作。而不可恢复错误往往就是bug的另一种说法，比如尝试访问超出数组结尾的位置等。

其他大部分的编程语言都没有刻意地区分这两种错误，而是通过异常之类的机制来统一处理它们。虽然Rust没有类似的异常机制，但它提供了用于可恢复错误的类型Result<T, E>，以及在程序出现不可恢复错误时中止运行的panic! 宏。本章会依次介绍调用panic! 宏及返回Result<T, E>类型的值。另外，我们还会讨论什么时候应该尝试从错误中恢复，而什么时候应该终止运行。

不可恢复错误与panic!

代码里总是会出现一些令你束手无策的糟糕情形。为了应对这样的场景，Rust提供了一个特殊的panic! 宏。程序会在panic! 宏执行时打印出一段错误提示信息，展开并清理当前的调用栈，然后退出程序。这种情况大部分都发生在某个错误被检测到，但程序员却不知该如何处理的时候。

panic中的栈展开与终止

当panic发生时，程序会默认开始栈展开。这意味着Rust会沿着调用栈的反向顺序遍历所有调用函数，并依次清理这些函数中的数据。但是为了支持这种遍历和清理操作，我们需要在二进制中存储许多额外信息。除了展开，我们还可以选择立即终止程序，它会直接结束程序且不进行任何清理工作，程序所使用过的内存只能由操作系统来进行回收。假如项目需要使最终二进制包尽可能小，那么你可以通过在*Cargo.toml* 文件中的[profile] 区域添加panic = 'abort' 来将panic的默认行为从展开切换为终止。例如，如果你想要在发布模式中使用终止模式，那么可以在配置文件中加入：

```
[profile.release]
panic = 'abort'
```

让我们先来尝试一下在一段简单的程序中调用panic!：

src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

当你运行这段程序时，会看到如下所示的输出：

```
$ cargo run

Compiling panic v0.1.0 (file:///projects/panic)
Finished dev [unoptimized + debuginfo] target(s) in 0.25 secs
Running `target/debug/panic'
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

由于调用了panic!，因此输出了最后两行错误提示信息。第一行显示了我们向panic所提供的信息，并指出了源代码中panic所发生的位置：*src/main.rs:2:5* 表明panic发生在文件*src/main.rs* 中第二行的第五个字符处。

在本例中，日志所指出的位置正处于我们自己的代码中，假如我们跳转到这一行，就可以看到对应的panic! 宏调用。而在其他某些情况下，panic! 调用可能会出现在我们所依赖的某些代码里，这段错误提示信息所指明的文件名和行号也会对应那些被依赖代码中发生panic! 调用的地方。我们依然可以通过查看panic! 调用函数的回溯信息来定位代码出现问题的地方。那么回溯信息又是什么呢？让我们接着来详细地了解一下。

使用panic! 产生的回溯信息

下面来看一个例子，它没有直接在代码中调用panic!，但会因为其中代码的bug而导致标准库中产生panic!。示例9-1中的代码会尝试使用索引来访问动态数组中的元素。

src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}
```

示例9-1：尝试越界访问动态数组中的元素，这会导致panic!

在这段代码中，动态函数只持有3个元素，但我们却在尝试访问它的第100个元素（由于索引从0开始，所以第100个元素的索引为99）。

在这种情况下，Rust会触发panic。使用[]意味着可以返回一个元素，但如果我们传入了一个非法的索引，那么它所指向的位置就没有可供Rust返回的合法元素了。

在类似于C这样的语言中，程序在这种情况下依然会尝试返回你所请求的值，即便这可能会与你所期望的并不相符：你会得到动态数组中对应这个索引位置的内存，而这个内存可能存储了其他数据，甚至都不属于动态数组本身。这种情形也被称为缓冲区溢出（buffer overread），并可能导致严重的安全性问题。攻击者可以通过操纵索引来访问存储在数组后面的、那些不被允许读取的数据。

为了保护我们的程序，避免出现类似的漏洞，当你尝试读取一个非法索引指向的元素时，Rust会拒绝继续执行代码，并终止程序。让我们尝试运行一下吧：

```
$ cargo run
```

```
Compiling panic v0.1.0 (file:///projects/panic)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
libcore/slice/mod.rs:2448:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这段错误提示信息指向了一个我们没有写过的文件*libcore/slice/mod.rs*。它是Rust源代码中实现slice的地方，我们对动态数组v使用[]运算符时所运行的代码就放置在*libcore/slice/mod.rs*中，这也是上面触发panic!的地方。

随后的输出行还提示我们可以通过设置环境变量RUST_BACKTRACE来得到回溯信息，进而确定触发错误的原因。回溯中包含了到达错误点的所有调用函数列表。在Rust中使用回溯的方式与其他语言中的使用方式类似：从头开始查看回溯列表，直至定位到自己所编写代码的文件，而这也正是产生问题的地方。从定位到文件的那一行往上是我们代码所调用的代码，往下则是调用了我们代码的代码。这些调用中可能会包含Rust核心库、标准库，以及你所使用的第三方库。让我们来将环境变量RUST_BACKTRACE设置为一个非0值，从而获得回溯信息。输出如示例9-2所示。

```
$ RUST_BACKTRACE=1 cargo run
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
libcore/slice/mod.rs:2448:10
stack backtrace:
 0: std::sys::unix::backtrace::tracing::imp:: unwind_backtrace
    at libstd/sys/unix/backtrace/tracing/gcc_s.rs:49
 1: std::sys_common::backtrace::print
    at libstd/sys_common/backtrace.rs:71
    at libstd/sys_common/backtrace.rs:59
 2: std::panicking::default_hook::{closure}
    at libstd/panicking.rs:211
 3: std::panicking::default_hook
    at libstd/panicking.rs:227
 4: <std::panicking::begin_panic::PanicPayload<A> as
    core::panic::BoxMeUp>::get
    at libstd/panicking.rs:476
 5: std::panicking::continue_panic_fmt
    at libstd/panicking.rs:390
 6: std::panicking::try::do_call
    at libstd/panicking.rs:325
 7: core::ptr::drop_in_place
    at libcore/panicking.rs:77
 8: core::ptr::drop_in_place
    at libcore/panicking.rs:59
 9: <usize as core::slice::SliceIndex<[T]>>::index
    at libcore/slice/mod.rs:2448
10: core::slice::<impl core::ops::index::Index<I> for [T]>::index
    at libcore/slice/mod.rs:2316
11: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
    at liballoc/vec.rs:1653
12: panic::main
    at src/main.rs:4
13: std::rt::lang_start::{closure}
    at libstd/rt.rs:74
14: std::panicking::try::do_call
    at libstd/rt.rs:59
    at libstd/panicking.rs:310
15: macho_symbol_search
    at libpanic_unwind/lib.rs:102
16: std::alloc::default_alloc_error_hook
    at libstd/panicking.rs:289
    at libstd/panic.rs:392
    at libstd/rt.rs:58
17: std::rt::lang_start
    at libstd/rt.rs:74
18: panic::main
```

示例9-2：当环境变量RUST_BACKTRACE被设置好后，通过调用panic!所生成的回溯

这里输出的日志可包含不少内容！当然，你所看到的信息可能会因操作系统不同或Rust版本不同而产生一些区别。另外，为了获取这些带有调试信息的回溯，你必须启用调试符号（debug symbol）。在运行cargo build或cargo run命令时，如果没有附带--release标志，那么调试符号就是默认开启的，正如我们这里一样。

在示例9-2的输出中，回溯的第12行指向了项目中导致错误的地方：文件 `src/main.rs` 的第四行。假如我们并不想让程序出现这种 `panic!`，就应该从我们所编写的代码中首个被提到的文件开始着手调查。在示例9-1中，我们特意编写了可能会导致 `panic!` 的代码来演示如何使用回溯，而修复这个 `panic!` 的方式就是避免在只拥有3个元素的动态数组中尝试引用第100个元素。如果将来代码发生了 `panic!`，你就需要自己去搞清楚代码中的哪些操作或哪些值导致了 `panic!`，并且思考应该如何修改代码以避免出现问题。

我们将在本章后面的“要不要使用 `panic!`”一节来继续讨论使用 `panic!` 进行错误处理的最佳时机。接下来，让我们继续学习如何使用 `Result` 从错误中恢复。

可恢复错误与Result

大部分的错误其实都没有严重到需要整个程序停止运行的地步。函数常常会由于一些可以简单解释并做出响应的原因而运行失败。例如，尝试打开文件的操作会因为文件不存在而失败。你也许会在这种情形下考虑创建该文件而不是终止进程。

还记得我们在第2章的“使用Result类型来处理可能失败的情况”一节所讨论的内容吗？里面的Result枚举定义了两个变体—Ok和Err，如下所示：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

这里的T和E是泛型参数：我们将在第10章深入地讨论泛型。现在你只需要知道：T代表了Ok变体中包含的值类型，该变体中的值会在执行成功时返回；而E则代表了Err变体中包含的错误类型，该变体中的值会在执行失败时返回。正是因为Result拥有这些泛型参数，我们才得以将Result类型及标准库中为它编写的函数应用于众多场景中，这些场景往往需要返回不同的成功值与错误值。

让我们调用一个可能会运行失败的函数，它会返回Result值作为运行结果。在示例9-3中，我们尝试着去打开一个文件。

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

示例9-3：打开一个文件

我们怎么知道File::open会返回一个Result呢？除了翻阅标准库API文档，我们还可以直接向编译器索要答案！假如我们赋予f一个错误的类型标记并尝试编译这段代码，编译器就会通知我们发生了类型不匹配的错误。这条错误提示信息会直接指出f的正确类型。让我们试试看吧！我们可以猜到File::open的返回类型不会是u32，进而可以将let f语句改为下面这样：

```
let f: u32 = File::open("hello.txt");
```

尝试编译这段代码就会得到如下所示的输出：

```
error[E0308]: mismatched types
--> src/main.rs:4:18
|
4 |     let f: u32 = File::open("hello.txt");
|     ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
|
= note: expected type `u32`
      found type `std::result::Result<std::fs::File, std::io::Error>`
```

上面的输出表明，File::open函数的返回类型是Result<T, E>。这里的泛型参数T被替换为了成功值的类型std::fs::File，也就是文件的句柄，而错误值所对应的类型E则被替换为了std::io::Error。

这个返回类型意味着File::open的调用可能成功，并会返回用于读写文件的句柄。它的调用也同样可能失败，例如，当文件不存在或我们没有访问文件的权限时。File::open函数需要能够通过某种方法在通知用户是否调用成功的同时，返回文件句柄或错误提示信息。这也是Result枚举所能够提供的功能。

当File::open函数运行成功时，变量f中的值将会是一个包含了文件句柄的Ok实例。当它运行失败时，变量f中的值则会是一个包含了用于描述错误种类信息的Err实例。

现在，我们需要基于File::open函数的返回值，向示例9-3中的代码添加不同的处理逻辑。示例9-4使用了我们在第6章讨论过的match表达式作为工具来处理Result。

src/main.rs

```
use std::fs::File;
fn main() {
```

```
let f = File::open("hello.txt");

let f = match f {
    Ok(file) => file,
    Err(error) => {
        panic!("There was a problem opening the file: {:?}", error)
    },
};

}
```

示例9-4：使用match表达式来处理所有可能的Result变体

注意，与Option枚举一样，Result枚举及其变体已经通过预导入模块被自动地引入当前作用域中，所以我们不需要在使用Ok变体与Err变体之前在match分支中显式地声明Result::。

我们的代码告诉Rust，当结果是Ok的时候，将Ok变体内部的file值移出，并将这个文件句柄重新绑定至变量f。这样在执行完match表达式之后，我们就能够使用这个句柄来进行读写操作了。

而match的另一个分支则处理了File::open返回Err值的情形。在本例中，我们选择通过调用panic! 宏来处理该情形。当我们运行代码，且当前目录中还不存在一个名为*hello.txt* 的文件时，就会看到来自panic! 宏的输出，如下所示：

```
thread 'main' panicked at 'There was a problem opening the file: Error
  { repr:
  Os { code: 2, message: "No such file or directory" } }',
src/main.rs:9:12
```

输出通常都会明确告诉我们错误的原因。

匹配不同的错误

不管File::open是因为何种原因而运行失败的，示例9-4中的代码都会触发panic!。但我们想要的其实是根据不同的失败原因做出不同的反应：当File::open因为文件不存在而运行失败时，我们可以创建这个文件并返回这个文件的句柄；而当File::open因为诸如没有访问权限之类的原因而运行失败时，我们才会像示例9-4一样直接触发panic!。示例9-5中的代码增加了一个内部match表达式。

src/main.rs

```

use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was a problem:
{:?}", e),
            },
            other_error => panic!("There was a problem opening the file: {:?}",
other_error),
        },
    };
}

```

示例9-5：用不同的方式处理不同的错误类型

File::open返回的Err变体中的错误值类型，是定义在某个标准库中的结构体类型：io::Error。这个结构体拥有一个被称作kind的方法，我们可以通过调用它来获得io::ErrorKind值。这个io::ErrorKind枚举是由标准库提供的，它的变体被用于描述io操作所可能导致的不同错误。这里使用的变体是ErrorKind::NotFound，它用于说明我们尝试打开的文件不存在。所以，我们不但对变量f使用了match表达式，还在内部对error.kind()使用了match表达式。

在这个匹配分支中，我们需要检查error.kind()返回的值是不是ErrorKind枚举的NotFound变体。如果是的话，我们就接着使用函数File::create来创建这个文件。

然而，由于File::create本身也有可能会运行失败，所以我们也需要对它的返回值添加一个match表达式。如果文件创建失败，那么就可以打印出一条不同的错误提示信息。外部match的最后一个分支保持不变，用于在出现其余错误时让程序触发panic。

这里出现了很多match! match表达式确实非常有用，但它同时也十分基础。Result<T,E>通过使用match表达式实现了许多接收闭包的方法；我们会在第13章开始学习闭包。一个更有经验的Rust开发者可能会像下面这样实现示例9-5中的代码：

src/main.rs

```

use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").map_err(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Tried to create file but there was a problem: {:?}", error);
            })
        } else {
            panic!("There was a problem opening the file: {:?}", error);
        }
    });
}

```

虽然这段代码与示例9-5拥有完全一致的行为，但它却没有使用任何的match表达式，并且更为清晰易读。你可以在阅读完第13章后再回到这个例子，并到标准库文档中查一下unwrap_or_else方法所起的作用。在处理错误时，有许多类似的方法可以简化嵌套的match表达式。

失败时触发panic的快捷方式：unwrap和expect

虽然使用match运行得很不错，但使用它所编写出来的代码可能会显得有些冗长，且无法较好地表明其意图。类型Result<T, E>本身也定义了许多辅助方法来应对各式各样的任务。其中一个被称为unwrap的方法实现了我们在示例9-4中编写的match表达式的效果。当Result的返回值是Ok变体时，unwrap就会返回Ok内部的值。而当Result的返回值是Err变体时，unwrap则会替我们调用panic! 宏。下面是一个在实际代码中使用unwrap的例子：

src/main.rs

```

use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}

```

假如我们在不存在`hello.txt`文件的前提下运行这段代码，就会触发unwrap方法中产生的panic! 调用，返回的错误提示信息如下所示：

```

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
    repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4

```

还有另外一个被称作expect的方法，它允许我们在unwrap的基础上指定panic! 所附带的错误提示信息。使用expect并附带上一段清晰的错误提示信息可以阐明你的意图，并使你更容易追踪到panic的起源。下面演示了expect的使用语法：

src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

我们使用expect所实现的功能与unwrap完全一样：要么返回指定文件句柄，要么触发panic! 宏调用。唯一的区别在于，expect触发panic! 时会将传入的参数字符串作为错误提示信息输出，而unwrap触发的panic! 则只会携带一段简短的默认信息。这段信息如下所示：

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr:
Os { code:
2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
```

因为这段错误提示信息包含了指定的文本—Failed to open hello.txt，所以我们能够更轻松地定位到代码中产生这段错误提示信息的地方。而因为所有unwrap触发的panic都会打印出同样的消息，所以假如我们同时在多个地方使用了unwrap，可能就需要付出额外的时间来分析一下究竟是哪一个unwrap导致了panic。

传播错误

当你编写的函数中包含了一些可能会执行失败的调用时，除了可以在函数中处理这个错误，还可以将这个错误返回给调用者，让他们决定应该如何做进一步处理。这个过程也被称作传播错误，在调用代码时它给了用户更多的控制能力。与编写代码时的上下文环境相比，调用者可能会拥有更多的信息和逻辑来决定应该如何处理错误。

示例9-6展示了一个从文件中读取用户名的函数。当文件不存在或无法读取时，这个函数会将错误作为结果返回给自己的调用者：

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> ❶ {
    ❷ let f = File::open("hello.txt");

    ❸ let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    ❹ let mut s = String::new();

    ❺ match f.read_to_string(&mut s) ❻ {
        Ok(_) => Ok(s) ❼,
        Err(e) => Err(e) ❽,
    }
}
```

示例9-6：使用match将错误返回给调用者的函数

这个函数其实能够以更加简捷的方式编写出来，但我们刻意保留了其中的冗余代码来解释错误处理流程；我们会在后面展示它的简便写法。首先，让我们将注意力放到这个函数的返回类型上：

`Result<String, io::Error>`❶。它意味着这个函数的返回值的类型为`Result<T, E>`，其中的泛型参数`T`被替换为具体的`String`类型，而泛型`E`则被替换为具体的`io::Error`类型。当这个函数顺利运行时，调用这个函数的代码将会获得一个包裹在`Ok`中的`String`值，也就是这个函数从文件中读取的用户名❷。而假如这个函数碰到了某个问题，函数的调用者就会获得一个包含了`io::Error`实例的`Err`值，这个实例中会包含问题的相关信息。我们之所以选择`io::Error`作为函数的返回类型，是因为函数中另外两个可能会失败的操作，`File::open`函数及`read_to_string`方法，恰好同样使用了`io::Error`作为错误类型。

函数体中的代码从调用`File::open`函数开始❷。接着，我们采用类似于示例9-4中的方式使用`match`表达式来处理返回的`Result`值❸。只是在这个例子中，我们选择了在`Err`情况下提前将`File::open`产生的错误作为结果返回，而不是调用`panic!`❹。假如函数`File::open`运行成功，我们就将生成的文件句柄存储到变量`f`中并继续执行下一步。

接着，我们基于变量s创建一个新的String⑤，然后调用文件句柄f中的read_to_string方法来将文件内容读取到s中⑥。即便File::open调用成功，这里的read_to_string方法同样可能会执行失败，所以它也返回了一个Result。为了处理这个Result，我们还需要一个match：假如read_to_string运行成功，我们就可以成功地将从文件中读取的用户名s封装到Ok中，并返回给调用者⑦。假如read_to_string运行失败，我们就可以像之前处理File::open时一样，将这个错误值作为结果返回⑧。但需要注意的是，由于这里是函数的最后一个表达式，所以我们不再需要显式地添加return。

调用这段代码的用户将需要处理包含了用户名的Ok值，或者包含了io::Error实例的Err值。我们无从得知调用者处理这些值的方式。当调用者获得了一个Err值时，他们可能会调用panic! 并直接终止程序，也可能会使用一个默认用户名，或者从另外的文件中尝试查找用户名。我们没有足够的上下文信息去知晓调用者会如何处理返回值，所以我们将成功信息和错误信息都向上传播，让调用者自行决定自己的处理方式。

传播错误的模式在Rust编程中非常常见，所以Rust专门提供了一个问号运算符（?）来简化它的语法。

传播错误的快捷方式：?运算符

示例9-7展示了一个与示例9-6中函数read_username_from_file拥有相同功能但使用了? 运算符的版本。

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

示例9-7：一个使用? 运算符来将错误返回给调用者的函数

通过将`?` 放置于`Result`值之后，我们实现了与示例9-6中使用`match`表达式来处理`Result`时一样的功能。假如这个`Result`的值是`Ok`，那么包含在`Ok`中的值就会作为这个表达式的结果返回并继续执行程序。假如值是`Err`，那么这个值就会作为整个程序的结果返回，如同使用了`return`一样将错误传播给调用者。

不过，我们还是需要指出示例9-6中的`match`表达式与`?` 运算符的一个区别：被`?` 运算符所接收的错误值会隐式地被`from`函数处理，这个函数定义于标准库的`From` trait中，用于在错误类型之间进行转换。当`?` 运算符调用`from`函数时，它就开始尝试将传入的错误类型转换为当前函数的返回错误类型。当一个函数拥有不同的失败原因，却使用了统一的错误返回类型来同时进行表达时，这个功能会十分有用。只要每个错误类型都实现了转换为返回错误类型的`from`函数，`?` 运算符就会自动帮我们处理所有的转换过程。

在示例9-7的上下文中，位于`File::open`句尾的`?` 会将存储在`Ok`内部的值返回给变量`f`。如果出现了错误，`?` 就会提前结束整个函数的执行，并将任何可能的`Err`值返回给函数调用者。这个规则同样也作用于调用`read_to_string`时句尾的`?` 。

`?` 运算符帮助我们消除了大量模板代码，使函数实现更为简单。我们甚至还可以通过链式方法调用进一步简化这些代码，如示例9-8所示。

src/main.rs

```
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

示例9-8：`?` 运算符后面的链式方法调用

我们将创建新`String`并赋值给`s`的语句移动到了函数开始的地方，这一部分没有任何改变。接下来，我们并没有创建变量`f`，而是直接将

`read_to_string`链接至`File::open ("hello.txt")?`所产生的结果处来进行调用。我们依然在`read_to_string`调用的尾部保留了`?`，并依然会在`File::open`和`read_to_string`都运行成功时，返回一个包含了用户名`s`的`Ok`值。这段函数所实现的功能与示例9-6和示例9-7完全一致，不过它使用了一个更符合项目实践的写法。

如果只是单纯地想要缩短代码，那么示例9-9中的写法可以使代码更短。

src/main.rs

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

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`读取文件

从文件中读取字符串可以说是一种相当常见的操作了，所以Rust提供了一个函数`fs::read_to_string`，用于打开文件，创建一个新`String`，并将文件中的内容读入这个`String`，接着返回给调用者。当然，直接使用这个函数无法给我们展示所有错误处理的机会，所以我们一开始选择了较为复杂的方法。

?运算符只能被用于返回Result的函数

因为?运算符的功能类似于示例9-6中定义的`match`表达式，所以它只能被用于那些拥有`Result`返回类型的函数。在`match`表达式中，`return Err(e)`部分产生的返回类型是`Result`，所以函数的返回类型也必须是`Result`，才能与此处的`return`兼容。

你应该还记得`main`函数的返回类型是`()`吧？让我们试试看在`main`函数中使用?运算符会发生些什么：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

当我们编译这段代码时，会看到如下所示的错误提示信息：

```
error[E0277]: the `?` operator can only be used in a function that
returns `Result` or `Option` (or another type that implements
`std::ops::Try`)
--> src/main.rs:4:13
|
4 |     let f = File::open("hello.txt")?;
|           ^^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a function
that returns `()``
|
= help: the trait `std::ops::Try` is not implemented for `()``
= note: required by `std::ops::Try::from_error`
```

这段错误提示信息指出，使用了`?` 运算符的函数必须返回`Result`、`Option`或任何实现了`std::ops::Try`的类型。在那些没有返回上述类型的函数里，一旦调用的其他函数返回了`Result<T, E>`，就需要使用`match`或`Result<T, E>`自身的方法来对`Result<T, E>`进行恰当的处理。当然，你也可以选择在合适的条件下将函数的返回类型修改为`Result<T, E>`。

对于特殊的`main`函数而言，可用的返回类型除了`()`，还有更加方便的`Result<T, E>`，如下所示：

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box
```

这里的`Box<dyn Error>`被称作trait对象，我们将在第17章讨论它。现在，你可以简单地将`Box<dyn Error>`理解为“任何可能的错误类型”。在拥有这种返回类型的`main`函数中使用`?` 运算符是合法的。

我们已经讨论了足够多关于调用`panic!`或返回`Result`的内容。让我们再来看一看它们各自的适用场景吧。

要不要使用panic!

什么时候应该使用panic!，而什么时候又应该返回Result呢？代码一旦发生panic，就再也没有恢复的可能了。只要你认为自己可以代替调用者决定某种情形是不可恢复的，那么就可以使用panic!，而不用考虑错误是否存在可以恢复的机会。当你选择返回一个Result值时，你就将这种选择权交给了调用者。调用者可以根据自己的实际情况来决定是否要尝试进行恢复，或者干脆认为Err是不可恢复的，并使用panic!来将可恢复错误转变为不可恢复错误。因此，我们会在定义一个可能失败的函数时优先考虑使用Result方案。

但对于某些不太常见的场景，直接触发panic要比返回Result更为合适一些。下面，我们会首先讨论为什么panic适用于示例、原型和测试等情形。接着，我们会讨论某些编程者确信错误不会发生但编译器却无法做出合理推断的场景。最后，我们会总结一些在库代码中是否应当使用panic的通用指导原则。

示例、原型和测试

当你编写示例用于演示某些概念时，为了增强健壮性而添加的错误处理代码往往会影响示例的可读性。在示例代码中，大家能够约定俗成地将unwrap之类可能会导致panic的方法理解为某种占位符，用来标明那些需要由应用程序进一步处理的错误，根据上下文环境的不同，具体的处理方法也会不同。

类似地，在原型中使用unwrap与expect方法也会非常方便，此时往往还无法决定具体的错误处理方式。当你准备好开始增强程序健壮性时，就可以使用它们在代码中留下的那些明显记号作为参考。

假如测试代码中的某个方法调用失败了，那么即便这个方法并不是需要测试的功能，我们也可以认为整个测试都失败了。测试的失败状态正是通过panic来进行标记的，所以这种场景也是我们应该调用unwrap或expect的场景。

当你比编译器拥有更多信息时

当你拥有某些逻辑可以确保Result是一个Ok值时，调用unwrap也是非常合理的，虽然编译器无法理解这种逻辑。你总是会拥有一个Result值需要处理：即便在某些特定的场景下，逻辑上不可能出现错误，但总的来说，你所调用的操作仍然有失败的可能。假如你可以通过人工检查确保代码永远不会出现Err变体，那就放心大胆地使用unwrap吧。下面就是一个例子：

```
use std::net::IpAddr;  
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

在这段代码中，我们通过解析一个硬编码的字符串来创建IpAddr实例。可以看到，127.0.0.1是一个有效的IP地址，所以在这里使用unwrap是合理的。但是，拥有一个硬编码、合法的字符串并不能改变parse方法的返回类型：我们依然会得到一个Result值，编译器依然会要求我们处理Err变体可能会出现的情形，编译器可没聪明到能够直接判断出这个字符串是一个合法的IP地址的程度。当这个IP地址字符串来自用户输入而不是硬编码，进而存在解析失败的可能时，我们就需要用一种更加健壮的方式来处理Result了。

错误处理的指导原则

当某个错误可能会导致代码处于损坏状态时，我们推荐你在代码中使用panic来处理错误。在这种情形下，损坏状态意味着设计中的一些假设、保证、约定或不可变性出现了被打破的情形。比如，当某些非法的值、自相矛盾的值或不存在的值被传入代码中，且满足下列某个条件时：

- 损坏状态并不包括预期中会偶尔发生的事情。

- 随后的代码无法在出现损坏状态后继续正常运行。
- 没有合适的方法来将“处于损坏状态”这一信息编码至我们所使用的类型中。

假如用户在使用你的代码时传入了一些毫无意义的值，最好的办法也许就是调用panic! 来警告他们代码中出现了bug，以便用户提前在开发过程中发现并解决这些问题。类似地，当你调用某些不可控制的外部代码，且这些代码出现了无法修复的非法状态时，也可以直接调用panic! 。

但是，假如错误是可预期的，那么就应该返回一个Result而不是调用panic! 。这样的例子包括解析器接收到错误数据的场景，以及HTTP请求返回限流状态的场景。在这些例子中，返回Result作为结果表明失败是一种可预期的状态，调用者必须决定如何处理这些失败。

当你的代码基于某些值来执行操作时，应该首先验证值的有效性，并在其无效时触发panic。这主要是出于安全性的考虑：尝试基于某些非法值去进行操作可能会暴露代码中的漏洞。这也是标准库会在我们尝试进行越界访问时触发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普遍的安全性问题。函数通常都有某种约定：它们只在输入数据满足某些特定条件时才能正常运行。在约定被违反时触发panic是合理的，因为破坏约定往往预示着调用产生了bug，而这不是我们希望用户显式去处理的错误类型。事实上，调用者很难用合理的方式对程序进行恢复；调用代码的程序员需要自行解决这些问题。函数的这些约定，尤其是在违反时会触发panic的那些约定，应该在API文档中被详细注明。

在所有的函数中都进行错误检测和处理可能会有些冗长和麻烦。但幸运的是，你可以借助于Rust的类型系统（也就是编译器所做的类型检查）来自动完成某些检测工作。假如你的函数拥有某个特定类型的参数，那么在知道编译器会确保值的有效性后，你就可以安全地基于它来继续编写代码了。例如，当你拥有一个不同于Option的类型，而你的程序期望接收一个非空值时，你的代码就无须处理Some和None两种变体的状态：你永远只会面对确定有值这一种情形。那些尝试传递空值给函数的代码根本就无法通过编译，所以你没有必要去编写代码用于运行时空值检测。另外一个例子是，使用u32这样的无符号整型时可以保证参数永远不会为负。

创建自定义类型来进行有效性验证

Rust的类型系统可以确保我们获得一个有效值。现在让我们更进一步，看一看如何创建一个自定义类型来进行有效性验证。还记得我们在第2章编写的猜数游戏吗？在那里，我们曾经请求用户输入一个1~100之间的数字。在将这个数字与保密数字进行比较之前，我们从未验证过用户的猜测是否处于这两个数字之间，而只是检查了数字是否为正。在这个场景中，缺少值检查的后果还没有那么严重：最终产生的“Too hight”或“Too low”输出依然是正确的。但是，这种值检测可以被用于引导玩家做出正确的选择，并在玩家尝试越界猜测或输入字符时给出不同的响应。

值检测的一种实现方式是将玩家的输入解析为i32（而不仅仅只是u32）来允许玩家输入负数，并接着检查数字是否处于1~100之间：

```
loop {
    // --略

    --
    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) {
        // --略
    }
}
```

这里的if表达式被用于检测传入的值是否处于1~100之间，告知用户出现的问题，并调用continue来继续请求玩家输入并开始下一次的循环迭代。在if表达式执行结束后，我们就可以在确保guess处于1~100之间的前提下，进行guess与保密数字的比较了。

不过，这并不是一个完美的解决方案：假设程序中有许多函数都强制要求参数值处于1~100之间，那么在每个对应的函数中都编写检查代码可能会相当麻烦（并可能影响性能）。

相比于到处重复验证代码，我们可以创建一个新的类型，并在创建的类型实例的函数中对值进行有效性检查。这样就可以在函数签名中安全地使用新类型，而无须担心我们所接收的值的有效性了。示例9-10展示了定义Guess类型的一种方法，它只有在new函数接收到一个1~100之间的数字时才会创建Guess实例。

```
❶ pub struct Guess {
    value: i32,
}

impl Guess {
❷ pub fn new(value: i32) -> Guess {
   ❸ if value < 1 || value > 100 {
       ❹ panic!("Guess value must be between 1 and 100, got {}.", value);
    }

❺ Guess {
    value
}
}

❻ pub fn value(&self) -> i32 {
    self.value
}
}
```

示例9-10：只有在值位于1~100之间时才创建Guess实例

首先，我们定义了一个名为Guess的结构体，其中包含了一个类型为i32的value字段❶，用于存储数字。

接着，我们为Guess实现了一个关联函数new，用于创建新的Guess实例❷。根据这个new函数的定义，它会接收一个i32类型的参数value并返回Guess。处于new函数体中的代码则会测试value是否处于1~100之间❸。假如value没有通过测试，我们就触发panic! 调用❹。这会警告调用这段代码的程序员出现了一个需要修复的bug，因为使用超出范围的value来创建Guess违反了Guess::new所依赖的约定。这会使得Guess::new触发panic的条件被详细地注释在这个函数所对应的公共API文档中。我们将在第14章讨论API文档中一些用于标明panic! 触发条件的习惯用法。假如value通过了这个测试，那么我们就会创建一个新的Guess，并将其字段value设置为对应的value参数，最后将Guess类型的实例返回给调用者❺。

最后，我们实现了一个value方法，它仅有一个参数用于借用self，并返回一个i32类型的值❶。这类方法有时也被称作读取接口（getter），因为它的功能就在于读取相应字段内的数据并返回。因为Guess中的value字段是私有的，所以我们有必要提供这类公共方法用于访问数据。而之所以将value字段设置为私有的，是因为我们不允许使用Guess结构体的代码随意修改value中的值：模块外的代码必须使用Guess::new函数来创建新的Guess实例，这就确保了所有Guess实例中的value都可以在Guess::new函数中进行有效性检查。

现在，如果一个函数需要将1~100之间的数字作为参数或返回值，那么它就可以在自己的签名中使用Guess（而不是i32），并且再也不需要在函数体内做任何额外的检查了。

总结

Rust 中的错误处理功能被设计出来帮助我们编写更加健壮的代码。`panic!` 宏表示程序正处于一个无法处理的状态下，你需要终止进程运行，而不是基于无效或非法的值继续执行命令。`Result` 枚举可以借助 Rust 的类型系统表明某个操作有失败的可能，并且代码能够从这种失败中恢复过来。你也可以使用 `Result` 来强制代码的调用者对可能的成功或失败情形都做出处理。合理地搭配使用 `panic!` 和 `Result` 可以让我们的代码在面对无法避免的错误时显得更加可靠。

到目前为止，你已经通过标准库中的 `Option` 与 `Result` 枚举见识了泛型的一些使用场景，我们会在下一章详细介绍泛型的工作机制，以及如何在代码中使用它们。

第10章

泛型、trait与生命周期



所有的编程语言都会致力于高效地处理重复概念，并为此提供各种各样的工具。在Rust中，泛型（generics）就是这样一种工具。泛型是具体类型或其他属性的抽象替代。在编写代码时，我们可以直接描述泛型的行为，或者它与其他泛型产生的联系，而无须知晓它在编译和运行代码时采用的具体类型。

函数可以使用参数中未知的具体值来执行相同的代码，与之类似地，函数也可以使用泛型参数而不是i32或String之类的具体类型。事实上，我们已经在不少地方使用过泛型了，它们包括第6章中的Option<T>、第8章中的Vec<T>与Hash<K, V>，以及第9章中的Result<T, E>。在本章中，你将进一步学会如何在声明自定义类型、函数与方法时使用泛型！

首先，我们会复习一下如何将代码提取为函数来减少代码重复。接着，我们将使用同样的技术来从两个仅仅是参数类型不同的函数中提取出泛型函数。另外，我们还会介绍如何在结构体与枚举中使用泛型。

之后，你将学会如何使用trait来定义通用行为。在定义泛型时，使用trait可以将其限制为拥有某些特定行为的类型，而不是任意类型。

最后，我们还会讨论生命周期。这类泛型可以向编译器提供引用之间的相互关系，它允许我们在借用值时通过编译器来确保这些引用的有效性。

通过将代码提取为函数来减少重复工作

在介绍泛型语法之前，让我们首先复习一下如何将代码提取为函数以减少重复工作。虽然这一过程不会涉及泛型概念，但在随后的一节中，我们会用同样的技术来将代码提取为泛型函数！如同你识别出可以提取为函数的重复代码一样，你也将会识别出能够使用泛型的重复代码。

如示例10-1所示，这段代码用来在一个数字列表中找到最大值。

src/main.rs

```
fn main() {
    ❶ let number_list = vec![34, 50, 25, 100, 65];

    ❷ let mut largest = number_list[0];

    ❸ for number in number_list {
        ❹ if number > largest {
            ❺ largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

示例10-1：在一个数字列表中找到最大值

这段代码首先在变量number_list中保存了一个整数列表❶，并将列表中的首个数字赋值给了变量largest❷。随后它又遍历了列表中的所有数字❸，若是当前数字大于largest中的数字❹，就将largest中的数字替换为当前的数字❺；而如果当前数字小于largest中的数字，就保持largest中的数字不变，并移动至列表中的下一个数字继续执行代码。在比较完列表中的全部数字后，存储在largest中的就是最大值了，在本例中也就是100。

如果需要在两个不同的列表中搜索各自的最大值，那么可以复制示例10-1中的代码，并在两个不同的地方使用相同的逻辑，如示例10-2所示。

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：在两个数字列表中分别找到最大值

尽管这段代码能够正常工作，但重复的代码总是乏味且易于出错的。一旦我们想要修改任何逻辑，就需要同时更新多个地方的代码。

为了消除这种重复代码，我们可以通过定义函数来创建抽象，它可以接收任意整数列表作为参数并进行求值。这将使我们的代码更加整洁，并可以让我们更加抽象地表达在整数列表中找到最大值的概念。

如示例10-3所示，我们把在整数列表中搜索最大值的代码提取到了函数largest中。与示例10-1中只能作用于特定列表的代码不同，这段程序可以为两个不同的整数列表分别找到最大值。

src/main.rs

```

fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        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：提取在两个列表中搜索最大值的代码

这里的largest函数拥有一个list参数，它代表了所有可能会传递给函数的具体i32值切片。因此，当我们调用函数时，这段代码会运行在我们所传入的特定值上。

总的来说，为了将示例10-2中的代码修改为示例10-3中的，我们大致经历了下面几步：

1. 定位到重复的代码。
2. 将重复的代码提取至函数体中，并在函数签名中指定代码的输入和返回值。
3. 将两段重复代码实例改为调用函数。

接下来，我们将会针对泛型使用同样的步骤，以一种不同的方式来减少代码重复。与这个函数体作用于抽象list而不是具体值一样，泛型允许代码作用于抽象的类型。

假设我们拥有两个不同的函数：一个用于在i32切片中搜索最大值，而另一个用于在char切片中搜索最大值。这里的重复性应当怎样消除呢？让我们拭目以待！

泛型数据类型

我们可以在声明函数签名或结构体等元素时使用泛型，并在随后搭配不同的具体类型来使用这些元素。本节会首先介绍如何使用泛型定义函数、结构体、枚举及方法，最后再来讨论泛型对代码性能所产生的影响。

在函数定义中

当使用泛型来定义一个函数时，我们需要将泛型放置在函数签名中通常用于指定参数和返回值类型的地方。以这种方式编写的代码更加灵活，并可以在不引入重复代码的同时向函数调用者提供更多的功能。

回到`largest`函数，示例10-4展示了两个同样被用于在切片中找到最大值的函数。

src/main.rs

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```

        }
    }

    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10-4：两个只在名称和签名类型上有所区别的函数

这里的largest_i32函数正是我们在示例10-3中归纳出来用于寻找i32切片中最大值的函数。而largest_char函数则是largest函数作用于char切片的版本。因为这两个函数拥有完全相同的代码，所以我们可以通过在一个函数中使用泛型来消除重复代码。

为了参数化这个新函数所使用的类型，我们首先需要给类型参数命名，就像我们为函数中的值参数命名一样。你可以使用任何合法的标识符作为类型参数名称。但出于惯例，我们选择了T。在Rust中，我们倾向于使用简短的泛型参数名称，通常仅仅是一个字母。另外，Rust采用了驼峰命名法（CamelCase）作为类型的命名规范。T作为“type”的缩写，往往是大部分Rust程序员在命名类型参数时的默认选择。

当我们需要在函数体中使用参数时，我们必须要在签名中声明对应的参数名称，以便编译器知晓这个名称的含义。类似地，当我们需要在函数签名中使用类型参数时，也需要在使用前声明这个类型参数的名称。为了定义泛型版本的largest函数，类型名称的声明必须被放置在函数名与参数列表之间的一对尖括号<>中，如下所示：

```
fn largest<T>(list: &[T]) -> T {
```

这段定义可以被理解为：函数largest拥有泛型参数T，它接收一个名为list的T值切片作为参数，并返回一个同样拥有类型T的值作为结果。

示例10-5展示了如何在签名中使用泛型数据类型并合并不同的largest函数。该示例同样也向我们展示了如何分别使用i32切片与char切片来调用函数。注意，这段代码目前还无法通过编译，我们会在本章后面的部分来修复它。

src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        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：使用泛型参数定义的largest函数，目前还无法通过编译

假如我们立即尝试编译这段代码，就会出现如下错误提示信息：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
  |
5 |         if item > largest {
  |         ^^^^^^^^^^^^^^
  |
= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

这段错误提示信息中提到的std::cmp::PartialOrd是一个trait，我们将在下一节来讨论它。简单来讲，这个错误表明largest函数中的代码不能适用于T所有可能的类型。因为函数体中的相关语句需要比较类型T的值，这个操作只能被用于可排序的值类型。我们可以通过实现标准库中的std::cmp::PartialOrd trait来为类型实现比较功能。你将在“使用trait作为参数”一节中学习如何为泛型参数指派特定的trait，现在先让我们看一看其他可能会用到泛型参数的地方。

在结构体定义中

同样地，我们也可以使用`< >`语法来定义在一个或多个字段中使用泛型的结构体。示例10-6展示了如何定义一个可以存储任意类型坐标的`Point<T>`结构体。

src/main.rs

```
struct Point<T>❶ {
    x: T,❷
    y: T,❸
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

示例10-6：存储了T类型值x与y的`Point<T>`结构体

在结构体定义中使用泛型语法的方式与在函数定义中的使用方式类似。在结构名后的一对尖括号中声明泛型参数❶后，我们就可以在结构体定义中那些通常用于指定具体数据类型的位置使用泛型了❷❸。

注意，我们在定义`Point<T>`时仅使用了一个泛型，这个定义表明`Point<T>`结构体对某个类型T是通用的。而无论具体的类型是什么，字段x与y都同时 属于这个类型。假如我们像示例10-7一样使用不同的值类型来创建`Point<T>`实例，那么代码是无法通过编译的。

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必须是相同的类型，因为它们拥有相同的泛型T

在这个例子中，当我们将整数5赋值给x时，编译器就会将这个Point<T>实例中的泛型T识别为整数类型。但是，我们接着为y指定了浮点数4.0，而这个变量被定义为与x拥有相同类型，因此这段代码就会触发一个类型不匹配错误：

```
error[E0308]: mismatched types
--> src/main.rs:7:38
|
7 |     let wont_work = Point { x: 5, y: 4.0 };
|                                     ^^^ expected integral variable, found
floating-point variable
|
|= note: expected type `'{integer}`
          found type `'{float}'
```

为了在保持泛型状态的前提下，让Point结构体中的x和y能够被实例化为不同的类型，我们可以使用多个泛型参数。例如，在示例10-8中，我们使Point的定义中拥有两个泛型参数T与U，其中x字段属于类型T，而y字段则属于类型U。

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：使用了两个泛型的Point<T, U>，x和y可以拥有不同类型的值

现在，所有的这些Point实例都是合法的了！你可以在定义中使用任意多个泛型参数，但要注意，过多的泛型会使代码难以阅读。通常来讲，当你需要在代码中使用很多泛型时，可能就意味着你的代码需要重构为更小的片段。

在枚举定义中

类似于结构体，枚举定义也可以在它们的变体中存放泛型数据。让我们再来看一看标准库中提供的Option<T>枚举，我们曾经在第6章使用过它：

```
enum Option<T> {
    Some(T),
    None,
}
```

你现在应该能够理解这个定义了。正如你所见，Option<T>是一个拥有泛型T的枚举。它拥有两个变体：持有T类型值的Some变体，以及一个不持有任何值的None变体。Option<T>被我们用来表示一个值可能存在的抽象概念。也正是因为Option<T>使用了泛型，所以无论这个可能存在的值是什么类型，我们都可以通过Option<T>来表达这一抽象。

枚举同样也可以使用多个泛型参数。我们在第9章使用过的Result枚举就是一个非常好的例子：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Result枚举拥有两个泛型：T和E。它也同样拥有两个变体：持有T类型值的Ok，以及一个持有E类型值的Err。这个定义使得Result枚举可以很方便地被用在操作可能成功（返回某个T类型的值），也可能失败（返回某个E类型的错误）的场景。实际上，我们在示例9-3中打开文件时就曾经使用过它；其中的泛型参数T被替换为std::fs::File类型，用来在文件成功打开时返回；而泛型参数E则被替换为了std::io::Error类型，用来描述打开文件过程中触发的问题。

当你意识到自己的代码拥有多个结构体或枚举定义，且仅仅只有值类型不同时，你就可以通过使用泛型来避免重复代码。

在方法定义中

如同第5章所介绍的，我们可以为结构体或枚举实现方法，而方法也可以在自己的定义中使用泛型。示例10-9基于示例10-6中定义的结构体Point<T>实现了一个名为x的方法。

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：为结构体Point<T>实现名为x的方法，它会返回一个指向x字段中T类型值的引用

在上面的代码中，我们为结构体Point<T>定义了一个名为x的方法，它会返回一个指向字段x中数据的引用。

注意，我们必须紧跟着impl关键字声明T，以便能够在实现方法时指定类型Point<T>。通过在impl之后将T声明为泛型，Rust能够识别出Point尖括号内的类型是泛型而不是具体类型。

打个比方，我们可以单独为Point<f32>实例而不是所有的Point<T>泛型实例来实现方法。在示例10-10中，我们使用了这个具体的类型f32，这也意味着我们无须在impl之后声明任何类型了。

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

示例10-10：这里的impl代码块只作用于使用具体类型替换了泛型参数T的结构体

这段代码意味着，类型Point<f32>将会拥有一个名为distance_from_origin的方法，而其他的Point<T>实例则没有该方法的定义。方法本身被用于计算当前点与原点坐标(0.0, 0.0)的距离，它使用了只能被用于浮点数类型的数学操作。

结构体定义中的泛型参数并不总是与我们在方法签名上使用的类型参数一致。例如，示例10-11中为来自示例10-8中的Point<T, U>结构体定义了一个方法mixup。这个方法会接收另外一个Point作为参数，而它与self参数所代表的Point之间有可能拥有不同的类型。方法在运行结束后会创建一个新的Point实例，这个实例的x值来自self所绑定的Point（拥有类型T），而y值则来自传入的Point（拥有类型W）。

src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U>❶ Point<T, U> {
    fn mixup<V, W>❷(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
❸ let p1 = Point { x: 5, y: 10.4 };
❹ let p2 = Point { x: "Hello", y: 'c' };

❺ let p3 = p1.mixup(p2);

❻ println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

示例10-11：方法使用了与结构体定义不同的泛型参数

在main中，我们定义了一个Point，它的x拥有类型为i32的值5，而y则拥有类型为f64的值10.4❸。接下来的p2变量同样是一个Point结构体，其中x的类型为字符串切片（值为"Hello"），而y的类型则是char（值为' c'）❹。在p1上调用mixup并传入p2作为参数，返回值为p3❺。p3会拥有类型为i32的字段x，因为x来自p1；它还会拥有类型为char的字段y，因为y来自p2。最后调用的println! 宏会输出p3. x = 5, p3. y = c❻。

这个例子说明，在某些情况下可能会有一部分泛型参数声明于impl关键字后，而另一部分则声明于方法定义中。在这里，泛型参数T

与U被声明在impl之后❶，因为它们是结构体定义的一部分。而泛型参数V与W则被定义在fn mixup中❷，因为它们仅仅与方法本身相关。

泛型代码的性能问题

当你使用泛型参数时，你也许会好奇这种机制是否存在一定的运行时消耗。好消息是，Rust实现泛型的方式决定了使用泛型的代码与使用具体类型的代码相比不会有任何速度上的差异。

为了实现这一点，Rust会在编译时执行泛型代码的单态化（monomorphization）。单态化是一个在编译期将泛型代码转换为特定代码的过程，它会将所有使用过的具体类型填入泛型参数从而得到有具体类型的代码。

在这个过程中，编译器所做的工作与我们在示例10-5中创建泛型函数时相反：它会寻找所有泛型代码被调用过的地方，并基于该泛型代码所使用的具体类型生成代码。

让我们看一看这套机制是怎么在标准库的Option<T>枚举上生效的：

```
let integer = Some(5);
let float = Some(5.0);
```

当Rust编译这段代码时，就会开始执行单态化。编译器会首先读取在Option<T>实例中被使用过的值，进而确定存在两种Option<T>：一种是i32，另一种是f32。因此，它会将Option<T>的泛型定义展开为Option_i32与Option_f64，接着再将泛型定义替换为这两个具体类型定义。

单态化后的代码如下所示。Option<T>被替换为了编译器所生成的特定定义：

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);
}
```

正是由于Rust会将每一个实例中的泛型代码编译为特定类型的代码，所以我们无须为泛型的使用付出任何运行时的代价。当运行泛型代码时，其运行效果和我们手动重复每个定义的运行效果一样。单态化使Rust的泛型代码在运行时极其高效。

trait：定义共享行为

trait（特征）被用来向Rust编译器描述某些特定类型拥有的且能够被其他类型共享的功能，它使我们可以以一种抽象的方式来定义共享行为。我们还可以使用trait约束来将泛型参数指定为实现了某些特定行为的类型。

注意

trait与其他语言中常被称为接口（interface）的功能类似，但也不尽相同。

定义trait

类型的行为由该类型本身可供调用的方法组成。当我们可以在不同的类型上调用相同的方法时，我们就称这些类型共享了相同的行为。trait提供了一种将特定方法签名组合起来的途径，它定义了为达成某种目的所必需的行为集合。

打个比方，假如我们拥有多个结构体，它们分别持有不同类型、不同数量的文本字段：其中的NewsArticle结构体存放了某地发生的新闻故事，而Tweet结构体则包含了最多280个字符的推文，以及用于描述该推文是一条新推文、一条转发推文还是一条回复的元数据。

此时，我们想要创建一个多媒体聚合库，用来显示存储在NewsArticle或Tweet结构体实例中的数据摘要。为了达到这一目标，我们需要为每个类型都实现摘要行为，从而可以在实例上调用统一的summarize方法来请求摘要内容。示例10-12展示了用于表达这一行为的Summary trait定义。

src/lib.rs

```
pub trait Summary {  
    fn summarize(&self) -> String;  
}
```

示例10-12：Summary trait由summarize方法所提供的行为组成

这里，我们使用了trait关键字来声明trait，紧随关键字的是该trait的名字，在本例中也就是Summary。在其后的花括号中，我们声明了用于定义类型行为的方法签名，也就是本例中的fn summarize(&self) -> String。

在方法签名后，我们省略了花括号及具体的实现，直接使用分号终结了当前的语句。任何想要实现这个trait的类型都需要为上述方法提供自定义行为。编译器会确保每一个实现了Summary trait的类型都定义了与这个签名完全一致的summarize方法。

一个trait可以包含多个方法：每个方法签名占据单独一行并以分号结尾。

为类型实现trait

我们基于Summary trait定义了所期望的行为，现在就可以在多媒体聚合中依次为每个类型实现这个trait了。示例10-13展示了NewsArticle结构体的Summary trait实现，该结构体使用了标题、作者及位置来创建summarize方法的返回值。而对于Tweet结构体，我们则选择了将用户名和全部推文作为summarize返回值，并假设推文内容已经被限制在了280个字符以内。

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 Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}

```

示例10-13：为NewsArticle与Tweet类型实现Summary trait

为类型实现trait与实现普通方法的步骤十分类似。它们的区别在于我们必须在impl关键字后提供我们想要实现的trait名，并紧接for关键字及当前的类型名。在impl代码块中，我们同样需要填入trait中的方法签名。但在每个签名的结尾不再使用分号，而是使用花括号并在其中编写函数体来为这个特定类型实现该trait的方法所应具有的行为。

一旦实现了trait，我们便可以基于NewsArticle和Tweet的实例调用该trait的方法了，正如我们调用普通方法一样：

```

let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());

```

这段代码会打印出 1 new tweet: horse_ebooks: of course, as you probably already know, people。

注意，示例10-13将Summary trait以及NewsArticle和Tweet结构体定义在了同一个lib.rs文件中，所以它们都处于相同的作用域中。假设这个lib.rs属于某个名为aggregator的库，当第三方开发者想要为他们自定义的结构体实现Summary trait并使用相关功能时，就必须将这个trait引入自己的作用域中。使用use aggregator::Summary; 语句就可以完成引入操作，进而调用相关方法或为自定义类型实现Summary。另外，示例10-12中的trait使用了pub关键字作为前缀，这

是因为我们必须要将Summary trait声明为公共的才能被其他库用于具体实现。

注意，实现trait有一个限制：只有当trait或类型定义于我们的库中时，我们才能为该类型实现对应的trait。例如，基于我们的aggregator库所提供的功能，我们可以为自定义类型，比如Tweet，实现标准库中的Display trait。能这么做的原因在于，类型Tweet定义在我们的aggregator库中。同样地，因为Summary trait也定义在我们的aggregator库中，所以也可以在aggregator库中为Vec<T>实现Summary trait。

但是，我们不能为外部类型实现外部trait。例如，我们不能在aggregator库内为Vec<T>实现Display trait，因为Display与Vec<T>都被定义在标准库中，而没有定义在aggregator库中。这个限制被称为孤儿规则（orphan rule），之所以这么命名是因为它的父类型没有定义在当前库中。这一规则也是程序一致性（coherence）的组成部分，它确保了其他人所编写的内容不会破坏到你的代码，反之亦然。如果没有这条规则，那么两个库可以分别对相同的类型实现相同的trait，Rust将无法确定应该使用哪一个版本。

默认实现

有些时候，为trait中的某些或所有方法都提供默认行为非常有用，它使我们无须为每一个类型的实现都提供自定义行为。当我们在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展示了如何为Summary trait中的summarize方法指定一个默认的字符串返回值，而不是如同示例10-12一样仅仅定义方法签名本身。

src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

示例10-14：拥有默认summarize方法实现的Summary trait定义

假如我们决定在NewsArticle的实例中使用这种默认实现而不是自定义实现，那么我们可以指定一个空的impl代码块：impl Summary for NewsArticle {}。

即便此时没有直接为NewsArticle定义summarize方法，我们也可以提供一个默认实现，并指定NewsArticle实现Summary trait。于是，我们依然可以在NewsArticle的实例上调用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());
```

这段代码会打印出New article available! (Read more...)。

为summarize提供一个默认实现并不会影响示例10-13中为Tweet实现Summary时所编写的代码。这是因为重载默认实现与实现trait方法的语法完全一致。

我们还可以在默认实现中调用相同trait中的其他方法，哪怕这些方法没有默认实现。基于这一规则，trait可以在只需要实现一小部分方法的前提下，提供许多有用的功能。例如，我们可以为Summary trait定义一个需要被实现的方法summarize_author，这样就可以通过调用summarize_author来为summarize方法提供一个默认实现：

```
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，我们只需要在为类型实现这一trait时定义summarize_author：

```
impl Summary for Tweet {  
    fn summarize_author(&self) -> String {
```

```
        format!("@{}", self.username)
    }
}
```

在定义了`summarize_author`之后，我们就可以在`Tweet`结构体的实例上调用`summarize`了。这个`summarize`的默认实现会进一步调用我们提供的`summarize_author`定义。因为我们实现了`summarize_author`，所以`Summary trait`可以为我们提供`summarize`方法的行为，而无须编写额外的代码。

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

这段代码会打印出`1 new tweet: (Read more from @horse_ebooks...)`。

注意，我们是无法在重载方法实现的过程中调用该方法的默认实现的。

使用trait作为参数

现在，你应该已经学会如何去定义`trait`并为类型实现`trait`了。接下来，我们会继续讨论如何使用`trait`来定义接收不同类型参数的函数。

例如在示例10-13中，我们为`NewsArticle`与`Tweet`类型实现了`Summary trait`。我们可以定义一个`notify`函数来调用其`item`参数的`summarize`方法，这里的参数`item`可以是任何实现了`Summary trait`的类型。为了达到这一目的，我们需要像下面一样使用`impl Trait`语法：

```
pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

我们没有为`item`参数指定具体的类型，而是使用了`impl`关键字及对应的`trait`名称。这一参数可以接收任何实现了指定`trait`的类型。

在notify的函数体内，我们可以调用来自Summary trait的任何方法，当然也包括summarize。我们可以在调用notify时向其中传入任意一个NewsArticle或Tweet实例。尝试使用其他类型（诸如String或i32）来调用函数则无法通过编译，因为这些类型没有实现Summary。

trait约束

这里的impl Trait常被用在一些较短的示例中，但它其实只是trait约束（trait bound）的一种语法糖。它的完整形式如下所示：

```
pub fn notify<T: Summary>(item: T) {  
    println!("Breaking news! {}", item.summarize());  
}
```

这种较长的形式完全等价于之前的示例，只是后面的写法会稍显臃肿一些。我们将泛型参数与trait约束同时放置在尖括号中，并使用冒号分隔。

那么，什么时候才应该使用完整形式而不是impl Trait呢？简单来说，impl Trait更适用于短小的示例，而trait约束则更适用于复杂情形。例如，假设我们需要接收两个都实现了Summary的参数，那么使用impl Trait的写法如下所示：

```
pub fn notify(item1: impl Summary, item2: impl Summary) {
```

只要item1和item2可以使用不同的类型（同时都实现了Summary），这段代码就没有任何问题。但是，如果你想强迫两个参数使用同样的类型，又应当怎么处理呢？此时你就只能使用trait约束了：

```
pub fn notify<T: Summary>(item1: T, item2: T) {
```

泛型T指定了参数item1与item2的类型，它同时也决定了函数为item1与item2接收的参数值必须拥有相同的类型。

通过+语法来指定多个trait约束

假如notify函数需要在调用summarize方法的同时显示格式化后的item，那么item就必须实现两个不同的trait：Summary和Display。我们可以使用+语法做到这一点：

```
pub fn notify(item: impl Summary + Display) {
```

这一语法在泛型的trait约束中同样有效：

```
pub fn notify<T: Summary + Display>(item: T) {
```

通过指定的两个 trait 约束，可以在 notify 函数体中调用 summarize，并使用 {} 来格式化 item。

使用where从句来简化trait约束

使用过多的 trait 约束也有一些缺点。因为每个泛型都拥有自己的 trait 约束，定义有多个泛型参数的函数可能会有大量的 trait 约束信息需要被填写在函数名与参数列表之间。这往往会使函数签名变得难以理解。为了解决这一问题，Rust 提供了一个替代语法，使我们可以在函数签名之后使用 where 从句来指定 trait 约束。所以，相比于下面的代码：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

我们可以使用 where 从句改写为：

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

这时的函数签名就没有那么杂乱了。函数名、参数列表及返回类型的排布要紧密得多，与没有 trait 约束的函数相差无几。

返回实现了 trait 的类型

我们同样可以在返回值中使用 impl Trait 语法，用于返回某种实现了 trait 的类型：

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know, people"),
        reply: false,
        retweet: false,
    }
}
```

通过在返回类型中使用 `impl Summary`，我们指定 `returns_summarizable` 函数返回一个实现了 `Summary trait` 的类型作为结果，而无须显式地声明具体的类型名称。在本例中，`returns_summarizable` 返回了一个 `Tweet`，但调用者却无法知晓这一信息。

我们为什么需要这样的功能呢？在第13章中，我们将会学习两个重度依赖于 `trait` 的功能：闭包（closure）与迭代器（iterator）。这些功能会创建出只有编译器才知道的或签名长到难以想象的类型。`impl Trait` 使你可以精练地声明函数会返回实现了 `Iterator trait` 的类型，而不需要写出具体的类型。

但是，你只能在返回一个类型时使用 `impl Trait`。例如，下面这段代码中返回的 `NewsArticle` 和 `Tweet` 都实现了 `impl Summary`，却依然无法通过编译：

```
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 {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know,
people"),
            reply: false,
            retweet: false,
        }
    }
}
```

在上面的代码中，我们尝试返回 `NewsArticle` 或 `Tweet` 类型，但碍于 `impl Trait` 工作方式的限制，Rust 并不支持这样的写法。在第17章的“使用 `trait` 对象来存储不同类型的值”一节中，我们会讲到如何编写类似功能的函数。

使用 `trait` 约束来修复 `largest` 函数

现在，你已经学会了如何使用泛型参数约束来指定想要使用的行为，让我们回到示例10-5中，来尝试修复那个使用了泛型参数的largest函数的定义！如果你尝试运行那段代码，就会看到如下所示的错误：

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
|
5 |         if item > largest {
|             ^^^^^^^^^^^^^^
|
= note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

在largest的函数体中，我们想要使用大于(>)运算符来比较两个T类型的值。由于这一运算符被定义为标准库trait std::cmp::PartialOrd的一个默认方法，所以我们需要在T的trait约束中指定PartialOrd，才能够使largest函数用于任何可比较类型的切片上。由于PartialOrd位于预导入模块内，所以我们不需要手动将其引入作用域。先让我们把largest的签名改为下面这样：

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

当我们再次编译这段代码时，会触发另外一些错误：

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
--> src/main.rs:2:23
|
2 |     let mut largest = list[0];
|         ^^^^^^^
|
|         |
|         cannot move out of here
|         help: consider using a reference instead: `&list[0]`
```



```
error[E0507]: cannot move out of borrowed content
--> src/main.rs:4:9
|
4 |     for &item in list.iter() {
|         ^----
|         ||
|         |hint: to prevent move, use `ref item` or `ref mut item`
```



```
|         cannot move out of borrowed content
```

这段错误提示信息的核心在于cannot move out of type [T]，a non-copy slice（无法从不可复制的切片[T]中移出元素）。当我们编写largest函数的非泛型版本时，我们只尝试过搜索i32和char类型的最大值。正如在第4章讨论过的那样，i32或char这样拥有确定大小并被存储在栈上的类型，已经实现了Copy trait。但是当我们尝试将largest函数泛型化时，list参数中的类型有可能是没有实现Copy

trait的。这也就意味着，我们无法将list[0]中的值移出并绑定到largest变量上，进而会导致上面的错误。

为了确保这个函数只会被那些实现了Copy trait的类型所调用，我们可以把Copy加入T的trait约束中！示例10-15展示了largest函数泛型版本的完整代码，只要我们传入函数的切片值类型实现了PartialOrd和Copy这两个trait，该示例中的代码就能正常通过编译。

src/main.rs

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        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10-15：largest函数可以被用于任何实现了PartialOrd与Copy这两个trait的泛型

假如我们不希望使largest函数只能使用那些实现了Copy trait的类型，那么可以用Clone来替换T trait约束中的Copy。接着，当需要在largest函数中取得切片中某个值的所有权时，我们就可以使用克隆方法。当然，一旦搜索对象是类似于String之类的存储在堆上的类型时，使用clone函数就意味着我们会执行更多堆分配操作，而当需要处理大量数据时，执行堆分配可能会相当缓慢。

另一种可能的largest实现方式是返回切片中T值的引用。假如将返回类型从T修改为&T，并修改函数体使其返回一个引用，那么我们就

不再需要Clone或Copy来进行trait约束了，同时可以避免执行堆分配操作。不妨自己尝试着实现一下这种方案吧！

使用trait约束来有条件地实现方法

通过在带有泛型参数的impl代码块中使用trait约束，我们可以单独为实现了指定trait的类型编写方法。例如，示例10-16中的类型Pair<T>都会实现new函数，但只有在内部类型T实现了PartialOrd（用于比较）与Display（用于打印）这两个trait的前提下，才会实现cmd_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10-16：根据泛型的trait约束来有条件地实现方法

我们同样可以为实现了某个trait的类型有条件地实现另一个trait。对满足trait约束的所有类型实现trait也被称作覆盖实现（blanket implementation），这一机制被广泛地应用于Rust标准库中。例如，标准库对所有满足Display trait约束的类型实现了ToString trait。标准库中的impl代码块如下所示：

```
impl<T: Display> ToString for T {
    // --略
```

```
--  
}
```

由于标准库提供了上面的覆盖实现，所以我们可以为任何实现了 `Display` trait 的类型调用 `ToString` trait 中的 `to_string` 方法。例如，我们可以像下面一样将整数转换为对应的 `String` 值，因为整数实现了 `Display`：

```
let s = 3.to_string();
```

有关覆盖实现的描述信息在对应 trait 文档中的“implementors”部分可以找到。

借助于 trait 和 trait 约束，我们可以在使用泛型参数来消除重复代码的同时，向编译器指明自己希望泛型拥有的功能。而编译器则可以利用这些 trait 约束信息来确保代码中使用的具体类型提供了正确的行为。在动态语言中，尝试调用一个类型没有实现的方法会导致在运行时出现错误。但是，Rust 将这些错误出现的时期转移到了编译期，并迫使我们在运行代码之前修复问题。我们无须编写那些用于在运行时检查行为的代码，因为这些工作已经在编译期完成了。这一机制在保留泛型灵活性的同时提升了代码的性能。

生命周期是另外一种你已经接触过的泛型。普通泛型可以确保类型拥有期望的行为，与之不同的是，生命周期能够确保引用在我们的使用过程中一直有效。让我们接着来看一看生命周期是如何做到这一点的。

使用生命周期保证引用的有效性

我们在第4章的“引用与借用”一节中有意地跳过了一些细节：Rust的每个引用都有自己的生命周期（lifetime），它对应着引用保持有效性的作用域。在大多数时候，生命周期都是隐式且可以被推导出来的，就如同大部分时候类型也是可以被推导的一样。当出现了多个可能的类型时，我们就必须手动声明类型。类似地，当引用的生命周期可能以不同的方式相互关联时，我们就必须手动标注生命周期。Rust需要我们注明泛型生命周期参数之间的关系，来确保运行时实际使用的引用一定是有效的。

生命周期的概念不同于其他编程语言中的工具，从某种意义上说，它也是Rust最与众不同的特性。尽管我们无法在本章介绍所有与生命周期相关的内容，但我们会讨论一些常见的生命周期语法来帮助你熟悉这一概念。

使用生命周期来避免悬垂引用

生命周期最主要的目标在于避免悬垂引用，进而避免程序引用到非预期的数据。看一下示例10-17中的程序，它包含了一个外部作用域及一个内部作用域。

```
{
① let r;
{
② let x = 5;
③ r = &x;
④ }
⑤ println!("r: {}", r);
}
```

示例10-17：尝试在值离开作用域时使用指向它的引用

注意

示例10-17、10-18及10-24中的代码声明了一些未被初始化的变量，以便这些变量名可以存在于外部作用域中。初看起来，这好像与Rust中不存在空值的设计相矛盾。但实际上，只要我们尝试在赋值前使用这些变量就会触发编译时错误。Rust中确实不允许空值存在！

上面的代码在外部作用域中声明了一个名为r的未初始化变量❶，而内部作用域则声明了一个初始值为5的变量x❷。在内部作用域中，我们尝试将r的值设置为指向x的引用❸。接着，当内部作用域结束时❹，尝试去打印出r所指向的值❺。这段代码将无法通过编译，因为在我们使用r时，它所指向的值已经离开了作用域。下面是相关的错误提示信息：

```
error[E0597]: `x` does not live long enough
--> src/main.rs:7:5
  |
6 |         r = &x;
  |             - borrow occurs here
7 |     }
  |     ^ `x` dropped here while still borrowed
...
10| }
   | - borrowed value needs to live until here
```

上面的错误提示信息指出，变量x的存活周期不够长。这是因为x在到达第7行，也就是内部作用域结束时离开了自己的作用域。而r对于整个外部作用域始终是有效的，它的作用域要更大一些，也就是我们所说的“存活得更久一些”。假如Rust允许这段代码运行，r就会引用到在x离开作用域时已经释放的内存，这时任何基于r所进行的操作都无法正确地进行。那么，Rust是如何确定这段代码并不合法的呢？它使用了一个叫作借用检查器的工具。

借用检查器

Rust编译器拥有一个借用检查器（borrow checker），它被用于比较不同的作用域并确定所有借用的合法性。示例10-18针对示例10-17中的代码增加了用于说明变量生命周期的注释。

```

{
    let r;                      // -----
                                // |
{
    let x = 5;                  // -+-- 'b |
    r = &x;                     // | |
}
                                // -+
                                // |
println!("r: {}", r);      // |
}
                                // -----

```

示例10-18：r与x的生命周期的标注，它们分别对应'a与'b

在这里，我们将r的生命周期标注为了'a，并将x的生命周期标注为了'b。如你所见，内部的'b代码块要小于外部的'a生命周期代码块。在编译过程中，Rust会比较两段生命周期的大小，并发现r拥有生命周期'a，但却指向了拥有生命周期'b的内存。这段程序会由于'b比'a短而被拒绝通过编译：被引用对象的存在范围短于引用者。

示例10-19修复了这段代码中可能产生悬垂引用的问题，使代码可以成功通过编译。

```

{
    let x = 5;                  // -----
                                // |
    let r = &x;                  // --+-- 'a |
                                // | |
    println!("r: {}", r);      // | |
                                // --+ |
}
                                // -----

```

示例10-19：这里的引用是有效的，因为数据的生命周期要比引用更长

这里的x拥有长于'a的生命周期'b。这也意味着r可以引用x了，因为Rust知道r中的引用在x有效时会始终有效。

现在，你应该已经清楚引用的生命周期所存在的范围，以及Rust会如何通过分析生命周期来确保引用的合法性了。接下来，让我们看一看在函数上下文中那些被用于参数和返回值的泛型生命周期。

函数中的泛型生命周期

让我们来编写一个函数，用于返回两个字符串切片中较长的一个。这个函数会接收两个字符串切片作为参数，并返回一个字符串切片作为结果。当我们实现了longest函数之后，示例10-20中的代码应该会打印出The longest string is abcd。

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10-20：main函数会调用longest函数来找到两个字符串切片中较长的一个

需要注意的是，因为我们并不希望longest取得参数的所有权，所以它应该可以接收字符串切片（也就是引用）作为参数。同时，我们还希望这个函数既能处理String切片（也就是变量string1的类型），又能处理字符串字面量（也就是变量string2所存储的）。

如果你还不太清楚我们为什么对使用的参数有如上要求，那么你可以参考第4章的“将字符串切片作为参数”一节。

不过，假如试着像示例10-21一样实现longest函数，那么它将无法通过编译。

src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
```

```
        y  
    }  
}
```

示例10-21：用于返回两个字符串切片中较长的那个的longest函数，但目前还无法通过编译

在编译过程中会触发涉及生命周期的错误：

```
error[E0106]: missing lifetime specifier  
--> src/main.rs:1:33  
|  
1 | fn longest(x: &str, y: &str) -> &str {  
| | ^ expected 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`
```

帮助文本解释了具体的错误原因：我们需要给返回类型标注一个泛型生命周期参数，因为Rust并不能确定返回的引用会指向x还是指向y。实际上，即便是编写代码的我们也无法做出这个判断。因为函数体中的if代码块返回了x的引用，而else代码块则返回了y的引用。

在我们定义这个函数的时候，我们并不知道会被传入函数的具体值，所以也不能确定到底是if分支还是else分支会得到执行。我们同样也无法知晓传入的引用的具体生命周期，所以就无法像示例10-18和10-19那样通过分析作用域来确定返回的引用是否有效。借用检查器自然也无法确定这一点，因为它不知道x与y的生命周期是如何与返回值的生命周期相关联的。为了解决这个问题，我们会添加一个泛型生命周期参数，并用它来定义引用之间的关系，进而使借用检查器可以正常地进行分析。

生命周期标注语法

生命周期的标注并不会改变任何引用的生命周期长度。如同使用了泛型参数的函数可以接收任何类型一样，使用了泛型生命周期的函数也可以接收带有任何生命周期的引用。在不影响生命周期的前提下，标注本身会被用于描述多个引用生命周期之间的关系。

生命周期的标注使用了一种明显不同的语法：它们的参数名称必须以撇号（'）开头，且通常使用全小写字符。与泛型一样，它们的名

称通常也会非常简短。’ a被大部分开发者选择作为默认使用的名称。我们会将生命周期参数的标注填写在&引用运算符之后，并通过一个空格符来将标注与引用类型区分开来。

这里有一些例子：一个指向i32且不带生命周期参数的引用，一个指向i32且带有名为’ a的生命周期参数的引用，以及一个同样拥有生命周期’ a的指向i32的可变引用。

```
&i32           // 引用  
&'a i32        // 拥有显式生命周期的引用  
&'a mut i32    // 拥有显式生命周期的可变引用
```

单个生命周期的标注本身并没有太多意义，标注之所以存在是为了向Rust描述多个泛型生命周期参数之间的关系。例如，假设我们编写了一个函数，这个函数的参数first是一个指向i32的引用，并且拥有生命周期’ a。它的另一个参数second同样也是指向i32且拥有生命周期’ a的引用。这样的标注就意味着：first和second的引用必须与这里的泛型生命周期存活一样长的时间。

函数签名中的生命周期标注

现在，让我们回过头来看一看longest函数上下文中的生命周期标注。如同泛型参数一样，我们同样需要在函数名与参数列表之间的尖括号内声明泛型生命周期参数。在这个签名中我们所表达的意思是：参数与返回值中的所有引用都必须拥有相同的生命周期。我们将这个生命周期命名为’ a并将它添加至每个引用中，如示例10-22所示。

src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

示例10-22：longest函数的定义指定了签名中所有的引用都必须拥有相同的生命周期’ a

这段代码现在能够正常编译，结果与在示例10-20的main函数中使用它们时的结果相同。

这段代码的函数签名向Rust表明，函数所获取的两个字符串切片参数的存活时间，必须不短于给定的生命周期' a。这个函数签名同时也意味着，从这个函数返回的字符串切片也可以获得不短于' a的生命周期。而这些正是我们需要Rust所保障的约束。记住，当我们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。我们只是向借用检查器指出了一些可以用于检查非法调用的约束。注意，longest函数本身并不需要知道x与y的具体存活时长，只要某些作用域可以被用来替换' a并满足约束就可以了。

当我们在函数中标注生命周期时，这些标注会出现在函数签名而不是函数体中。Rust可以独立地完成对函数内代码的分析。但是，当函数开始引用或被函数外部的代码所引用时，想要单靠Rust自身来确定参数或返回值的生命周期，就几乎是不可能的了。函数所使用的生命周期可能在每次调用中都会发生变化。这也正是我们需要手动对生命周期进行标注的原因。

当我们具体的引用传入longest时，被用于替代' a的具体生命周期就是作用域x与作用域y重叠的那一部分。换句话说，泛型生命周期' a会被具体化为x与y两者中生命周期较短的那一个。因为我们将返回的引用也标记为了生命周期参数' a，所以返回的引用在具化后的生命周期范围内都是有效的。

让我们通过一个示例来看一看生命周期标注是如何对longest函数的调用进行限制的。在示例10-23中，我们向函数中传入了拥有不同具体生命周期的引用。

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10-23：使用具有不同生命周期的String来调用longest函数

在这个示例中，`string1`直到外部作用域结束都会是有效的，而`string2`的有效性则只持续到内部作用域结束的地方。运行这段代码，它可以正常地通过借用检查器进行编译，并最终输出The longest string is long string is long。

接下来的这个示例被用于演示`result`引用中的生命周期必须要小于两个参数的生命周期。我们会将对`result`变量的声明移出内部作用域，只将`result`变量的赋值操作与`string2`一同保留在内部作用域中。接着，我们将使用`result`的`println!`移动到内部作用域结束后的地 方。示例10-24中的代码无法通过编译。

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10-24：尝试在string2离开作用域后使用result

当我们尝试编译这段代码时，就会出现如下所示的错误：

```
error[E0597]: `string2` does not live long enough
--> src/main.rs:15:5
|
14 |         result = longest(string1.as_str(), string2.as_str());
|                         ----- borrow occurs here
15 |     }
|     ^ `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
17 | }
```

这里错误提示信息的意思是，为了使`println!`语句中的`result`是有效的，`string2`需要一直保持有效，直到外部作用域结束的地方。因为我们在函数参数与返回值中使用了同样的生命周期参数'a，所以Rust才会指出这些问题。

对人类而言，我们可以确定string1中的字符要长于string2，进而确定result中将会持有指向string1的引用。由于string1在我们使用println!语句时还没有离开自己的作用域，所以这个指向string1的引用应该是完全合法的才对。但是，编译器无法在这种情形下得出引用一定有效的结论。不过，我们曾经告诉过Rust，longest函数返回的引用的生命周期与传入的引用的生命周期中较短的那个相同。仅在这一约束下，还是有可能出现非法引用的，因此借用检查器拒绝编译示例10-12中的代码。

在开始下一节之前，请尝试将不同的值、具有不同生命周期的引用传入longest函数，并改变返回引用的使用方式；接着，提前对代码能否通过借用检查器的编译做出判断；最后，借助编译来验证自己的猜想！

深入理解生命周期

指定生命周期的方式往往取决于函数的具体功能。打个比方，假如将longest函数的实现修改为返回第一个而不是最长的那个字符串切片参数，那么我们就无须再为y参数指定生命周期。下面的代码是可以通过编译的：

src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

在这个例子中，我们为参数x与返回类型指定了相同的生命周期参数'a，却有意忽略了参数y，这是因为y的生命周期与x和返回值的生命周期没有任何相互关系。

当函数返回一个引用时，返回类型的生命周期参数必须要与其中一个参数的生命周期参数相匹配。当返回的引用没有指向任何参数时，那么它只可能是指向了一个创建于函数内部的值，由于这个值会因为函数的结束而离开作用域，所以返回的内容也就变成了悬垂引用。下面来看一个无法通过编译的longest函数实现：

src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

即便我们在上面的代码中为返回类型指定了生命周期参数'a，这个实现也依然无法通过编译，因为返回值的生命周期没有与任何参数的生命周期产生关联。下面是编译后产生的错误提示信息：

```
error[E0597]: `result` does not live long enough
--> src/main.rs:3:5
|
3 |     result.as_str()
|     ^^^^^^ does not live long enough
4 | }
| - borrowed value only lives until here
|
note: borrowed value must be valid for the lifetime 'a as defined on the
function body at 1:1...
--> src/main.rs:1:1
|
1 | / fn longest<'a>(x: &str, y: &str) -> &'a str {
2 | |     let result = String::from("really long string");
3 | |     result.as_str()
4 | | }
```

这里的问题在于result在longest函数结束时就离开了作用域，并被清理。但我们依然在尝试从函数中返回一个指向result的引用。无论我们怎么改变生命周期参数，都无法阻止悬垂引用的产生，而Rust并不允许创建悬垂引用。在本例中，最好的解决办法就是返回一个持有自身所有权的数据类型而不是引用，这样就可以将清理值的责任转移给函数调用者了。

从根本上说，生命周期语法就是用来关联一个函数中不同参数及返回值的生命周期的。一旦它们形成了某种联系，Rust就获得了足够的信息来支持保障内存安全的操作，并阻止那些可能会导致悬垂指针或其他违反内存安全的行为。

结构体定义中的生命周期标注

到目前为止，我们只在结构体中定义过自持有类型。实际上，我们也可以在结构体中存储引用，不过需要为结构体定义中的每一个引用都添加生命周期标注。示例10-25定义了一个存放字符串切片的ImportantExcerpt结构体。

src/main.rs

```
①struct ImportantExcerpt<'a> {
    ② part: &'a str,
}

fn main() {
    ③ let novel = String::from("Call me Ishmael. Some years ago...");
    ④ let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    ⑤ let i = ImportantExcerpt { part: first_sentence };
}
```

示例10-25：结构体中持有了引用，所以它的定义中需要添加生命周期标注

这个结构体仅有一个字段part，用于存储一个字符串切片，也就是一个引用②。如同泛型数据类型一样，为了在结构体定义中使用生命周期参数，我们需要在结构体名称后的尖括号内声明泛型生命周期参数的名字①。这个标注意味着ImportantExcerpt实例的存活时间不能超过存储在part字段中的引用的存活时间。

在main函数中，我们首先创建了一个String实例novel③，接着又创建了一个ImportantExcerpt结构体的实例⑤，它存放了变量novel中第一个句子的引用④。在ImportantExcerpt实例创建之前，novel中的数据就已经生成了，而且novel会在ImportantExcerpt离开作用域后才离开作用域，所以ImportantExcerpt实例中的引用总是有效的。

生命周期省略

到目前为止，你应该已经知道，任何引用都有一个生命周期，并且需要为使用引用的函数或结构体指定生命周期参数。然而，在第4章的示例4-9中我们曾经编写过一个函数，它在没有任何生命周期标注的情况下正常地通过了编译，示例10-26展示了该函数的详细版本。

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10-26：即便参数和返回类型都是引用，示例4-9中定义的这个函数依然没有使用生命周期标注

这个函数之所以能够在没有生命周期标注的情况下通过编译是出于一些历史原因：在Rust的早期版本（pre-1.0）中，这样的代码确实无法通过编译，因为每个引用都必须有一个显式的生命周期。当时的函数签名会被写为：

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

在编写了相当多的Rust代码后，Rust团队发现，在某些特定情况下Rust程序员总是在一遍又一遍地编写同样的生命周期标注。这样的场景是可预测的，而且有一些明确的模式。于是，Rust团队决定将这些模式直接写入编译器代码中，使借用检查器在这些情况下可以自动对生命周期进行推导而无须显式标注。

了解这段Rust历史是有必要的，因为随着Rust自身的开发，可能会有更多确定性的模式被添加到编译器中。在未来，需要手动标注的生命周期也许会越来越少。

这些被写入Rust引用分析部分的模式也就是所谓的生命周期省略规则。这些规则并不需要程序员去遵守；它们只是指明了编译器会考虑的某些场景，当你的代码符合这些场景时，就无须再显式地为代码注明相关生命周期了。

省略规则并不能提供完整的推断。假如Rust在确定性地应用了规则后仍然对引用的生命周期存在歧义的话，那么编译器不会去猜测剩余引用所拥有的生命周期是怎样的。在这种情况下，编译器会直接抛出错误而不是进行随意猜测。你可以通过添加生命周期标注，显式地注明引用之间的关系，来解决这些错误。

函数参数或方法参数中的生命周期被称为输入生命周期（input lifetime），而返回值的生命周期则被称为输出生命周期（output

`lifetime`)。

在没有显式标注的情况下，编译器目前使用了3种规则来计算引用的生命周期。第一条规则作用于输入生命周期，第二条和第三条规则作用于输出生命周期。当编译器检查完这3条规则后仍有无法计算出生命周期的引用时，编译器就会停止运行并抛出错误。这些规则不但对fn定义生效，也对impl代码块生效。

第一条规则是，每一个引用参数都会拥有自己的生命周期参数。换句话说，单参数函数拥有一个生命周期参数：`fn foo<'a>(x: &'a i32)`；双参数函数拥有两个不同的生命周期参数：`fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`；以此类推。

第二条规则是，当只存在一个输入生命周期参数时，这个生命周期会被赋予给所有输出生命周期参数，例如`fn foo<'a>(x: &'a i32) -> &'a i32`。

第三条规则是，当拥有多个输入生命周期参数，而其中一个是指向self或mut self时，self的生命周期会被赋予给所有的输出生命周期参数。这条规则使方法更加易于阅读和编写，因为它省略了一些不必要的符号。

现在，让我们假设自己就是编译器。我们会尝试应用这些规则来计算出示例10-26中的`first_word`函数签名中引用的生命周期。这段签名中的引用刚开始时还没有关联任何生命周期：

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

接着，编译器开始应用第一条规则，为每个参数指定生命周期。我们按照惯例使用'`a`'，所以签名如下所示：

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

因为这里只有一个输入生命周期，所以第二条规则也是适用的。根据第二条规则，输入参数的生命周期将被赋予给输出生命周期参数，也就是：

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

现在，函数签名中所有的引用都已经有了生命周期，因此编译器可以继续分析代码，而无须程序员标注这个函数签名中的生命周期。

让我们再看一个例子。这次，我们用示例10-21中没有生命周期参数的longest函数来分析：

```
fn longest(x: &str, y: &str) -> &str {
```

依然使用第一条规则，即每一个参数都有自己的生命周期。因为这次我们有两个参数，所以产生了两个生命周期：

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

这时你会发现，由于函数中的输入生命周期个数超过一个，所以第二条规则不再适用。此外，由于longest是一个函数而不是方法，其中并没有self参数，所以第三条规则也不再适用。在遍历完所有的3条规则后，我们依然无法计算出返回类型的生命周期。这也是当我们尝试去编译示例10-21中的代码时会出现错误的原因：编译器已经使用了全部生命周期省略规则，却依然无法计算出签名中所有引用的生命周期。

因为第三条规则实际上只适用于方法签名，所以我们会接着来学习这一上下文环境中的生命周期，并看一看为什么第三条规则可以让我们在大部分的方法签名中省略生命周期标注。

方法定义中的生命周期标注

当我们需要为某个拥有生命周期的结构体实现方法时，可以使用与示例10-11中展示的与泛型参数相似的语法。声明和使用生命周期参数的位置取决于它们是与结构体字段相关，还是与方法参数、返回值相关。

结构体字段中的生命周期名字总是需要被声明在impl关键字之后，并被用于结构体名称之后，因为这些生命周期是结构体类型的一部分。

在impl代码块的方法签名中，引用可能是独立的，也可能会与结构体字段中的引用的生命周期相关联。另外，生命周期省略规则在大

部分情况下都可以帮我们免去方法签名中的生命周期标注。让我们来看一些使用了示例10-25中结构体ImportantExcerpt的例子。

首先，我们定义一个名为level的方法，它仅有一个指向self的参数，并返回i32类型的值作为结果，这个结果并不会引用任何东西：

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

声明在impl及类型名称之后的生命周期是不能省略的，但根据第一条省略规则，我们可以不用为方法中的self引用标注生命周期。

下面是一个应用了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通过应用第一条生命周期省略规则给了&self和announcement各自的生命周期。接着，由于其中一个参数是&self，返回类型被赋予了&self的生命周期，因此所有的生命周期就都被计算出来了。

静态生命周期

Rust中还存在一种特殊的生命周期' static，它表示整个程序的执行期。所有的字符串字面量都拥有' static生命周期，我们可以像下面一样显式地把它们标注出来：

```
let s: &'static str = "I have a static lifetime.;"
```

字符串的文本被直接存储在二进制程序中，并总是可用的。因此，所有字符串字面量的生命周期都是' static。

你可能会在错误提示信息中看到过关于使用' static生命周期的建议。不过，在将引用的生命周期指定为' static之前，记得要思考一下

你所持有的引用是否真的可以在整个程序的生命周期内都有效。即便它可以，你也需要考虑一下它是否真的需要存活那么长时间。大部分情况下，错误的原因都在于尝试创建一个悬垂引用或可用生命周期不匹配。这时，应该去解决这些问题，而不是指定¹ static生命周期。

同时使用泛型参数、trait约束与生命周期

让我们来简单地看一下在单个函数中同时指定泛型参数、trait约束及生命周期的语法：

```
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
    }
}
```

这是示例10-22中用于返回两个字符串切片中较长者的longest函数。但是现在，它多了一个额外的ann参数，这个参数的类型为泛型T。根据where从句中的约束，该参数的类型可以被替换为任何实现了Display trait的类型。这个额外的参数会在函数比较字符串切片长度之前被打印出来，所以我们需要Display来作为trait约束。因为生命周期也是泛型的一种，所以生命周期参数'a和泛型参数T都被放置到了函数名后的尖括号列表中。

总结

我们在这一章学习了不少内容！现在，你应该对泛型参数、trait与trait约束，以及泛型生命周期参数等概念比较熟悉了，也应该可以在没有重复代码的前提下编写出适用于多种场景的代码了。泛型参数可以使你将代码应用于不同的类型，而trait与trait约束则可以用来在代码中指定泛型的行为。除此之外，你还学到了如何使用生命周期来确保这些灵活的代码不会产生任何悬垂引用。所有的这些分析都将在编译过程中，而不会对运行时性能造成任何影响！

无论你是否相信，我们在本章讨论的内容都还有更多值得深入的细节：第17章将会讨论trait对象，这是另外一种使用trait的方式；第19章将会讨论某些涉及生命周期标注的高级类型系统功能。不过接下来，你会先学习如何在Rust中编写测试，它们可以确保你的代码能够按照预期的方式运行。

第11章

编写自动化测试



Edsger W. Dijkstra [\[1\]](#) 在1972年发表的文章《谦逊的程序员》(*The Humble Programmer*)中指出：“虽然测试可以高效地暴露程序中的bug，但在证明bug不存在方面却无能为力。”尽管测试有着这样的局限，但是我们作为开发者，仍然应该竭尽全力地去进行测试！

程序中的正确性被用来衡量一段代码的实际行为与设计目标之间的一致程度。从设计之初，Rust就将程序正确性作为一项非常优先的考量因素，但是一个程序最终是否正确，终究是复杂并且难以证明的。虽然Rust的类型系统为我们提供了相当多的安全保障，但还是不足以防止所有的错误。因此，Rust在语言层面内置了编写测试代码、执行自动化测试任务的功能。

例如，我们需要编写一个给任意数值加2的函数add_two。这个函数的签名会接收一个整型作为参数，并返回一个整型作为结果。当我们编译这个函数时，Rust会按照前面章节所介绍的规则进行完整的类型检查和借用检查。这样可以杜绝将String值或无效引用误传入函数中这样的错误。但是，Rust却无法确定这个函数是否能够按照我们的

意图去运行。它可能会返回输入值加2，也可能会返回输入值加10，甚至是输入值减50！这种场景正是测试的用武之地。

我们可以编写测试用例进行断言，例如，只要给add_two函数传入3，那么必定返回5。然后我们就可以在每次修改代码时运行测试，并利用断言确保所有已经存在的正确行为不会受到改动的影响。

测试是一门复杂的技术：虽然我们无法在本章覆盖关于如何编写优秀测试的每一个细节，但是会讨论Rust测试工具的运行机制。我们会向你介绍编写测试时常用的标注和宏、运行测试的默认行为和选项参数，以及如何将测试用例组织为单元测试与集成测试。

[1] 译者注：艾兹赫尔·韦伯·戴克斯特拉（1930年-2002年），荷兰计算机科学家，1972年获得图灵奖。

如何编写测试

Rust语言中的测试是一个函数，它被用于验证非测试代码是否按照期望的方式运行。测试函数的函数体中一般包含3个部分：

1. 准备所需的数据或状态。
2. 调用需要测试的代码。
3. 断言运行结果与我们所期望的一致。

接下来，我们会一起学习用于编写测试代码的相关功能，它们包含`test`属性、一些测试宏及`should_panic`属性。

测试函数的构成

在最简单的情形下，Rust中的测试就是一个标注有`test`属性的函数。属性（attribute）是一种用于修饰Rust代码的元数据；我们在第5章为结构体标注的`derive`就是一种属性。你只需要将`#[test]`添加到关键字`fn`的上一行便可以将函数转变为测试函数。当测试编写完成后，我们可以使用`cargo test`命令来运行测试。这个命令会构建并执行一个用于测试的可执行文件，该文件在执行的过程中会逐一调用所有标注了`test`属性的函数，并生成统计测试运行成功或失败的相关报告。

当我们使用Cargo新建一个库项目时，它会自动为我们生成一个带有测试函数的测试模块。这使你可以在启动新项目时立即开始编写测试代码，而无须查阅与测试相关的具体结构和语法。当然，你也可以额外增加任意多的测试函数与测试模块。

我们会先在这个生成的模板测试上进行实验，并介绍一些有关测试的基本概念。接着，我们会编写一些真实场景下的测试，它们会调用相关代码并对行为的正确性做出断言。

让我们来新建一个名为adder的库项目：

```
$ cargo new adder --lib
Created library `adder` project
$ cd adder
```

这个adder库会自动生成一个*src/lib.rs* 文件，其中的内容如示例11-1所示。

src/lib.rs

```
# [cfg(test)]
mod tests {
   ❶ #[test]
    fn it_works() {
       ❷ assert_eq!(2 + 2, 4);
    }
}
```

示例11-1：运行cargo new命令自动生成的测试模块和测试函数

让我们先忽略最上方的两行代码，并将注意力集中到测试函数部分。你可以看到❶这一行出现了#[test] 标注：它将当前的函数标记为一个测试，并使该函数可以在测试运行过程中被识别出来。要知道，即便是在tests模块中也可能会存在普通的非测试函数，它们通常被用来执行初始化操作或一些常用指令，所以我们必须要将测试函数标记为#[test]。

函数体中使用了assert_eq! 宏❷断言 $2+2$ 和4相等，这是一个典型的测试用例编写方式。让我们运行这段显然会通过的测试试试看。

执行命令cargo test会运行项目中的所有测试，如示例11-2所示。

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs
Running target/debug/deps/adder-ce99bcc2479f4607

①running 1 test
②test tests::it_works ... ok

③test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

④ Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

示例11-2：运行生成的模板测试后所输出的结果

Cargo成功编译并运行了这段测试。在运行结果的Compiling、Finished、Running这3行后面紧接着输出了running 1 test①，表示当前正在执行1个测试。接下来显示的是所生成的测试函数名称it_works，以及相对应的测试结果ok②。再下一行是该测试集的摘要。test result: ok. ③表示该集合中的所有测试均成功通过，1 passed; 0 failed则统计了通过和失败的测试总数。

由于我们没有将任何测试标记为忽略，所以摘要中出现了信息0 ignored。同样，由于我们没有对运行的测试进行过滤，所以摘要的末尾处输出了0 filtered out。我们会在“控制测试的运行方式”一节中讨论忽略和过滤测试的相关手段。

另外一处信息0 measured则统计了用于测量性能的测试数量。在编写此书时，性能测试（benchmark test）还只能用于Rust的nightly版本，请参阅Rust官方文档来了解更多关于性能测试的信息。

接下来以Doc-tests adder ④开头的部分是文档测试（documentation test）的结果。虽然我们还未编写过这种测试，但是要知道Rust能够编译在API文档中出现的任何代码示例。这一特性可以帮助我们保证文档总会和实际代码同步！我们将在第14章的“将文档注释用作测试”一节中讨论这部分内容，现在先暂时忽略与Doc-tests相关的输出即可。

让我们修改测试函数的名称来看一看输出结果会有怎样的变化。下面的代码将lib.rs文件中的it_works函数重命名为exploration：

src/lib.rs

```
# [cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

再次运行 cargo test，输出中的测试名从 it_works 变为了 exploration：

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

现在让我们添加一个新的测试，并故意使它成为一个会导致失败的案例！在Rust中，一旦测试函数触发panic，该测试就被视作执行失败。每个测试在运行时都处于独立的线程中，主线程在监视测试线程时，一旦发现测试线程意外终止，就会将对应的测试标记为失败。而触发panic最简单的方法就是调用我们在第9章讨论过的panic! 宏。增加一个新的测试another，如示例11-3所示。

src/lib.rs

```
# [cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

示例11-3：增加一个新的测试，它会因为调用panic! 宏而运行失败

再次使用cargo test运行测试，输出的结果如示例11-4所示。它表明exploration通过了测试，而another却失败了：

```
running 2 tests
test tests::exploration ... ok
❶test tests::another ... FAILED

❷failures:
```

```
---- tests::another stdout ----
thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

③failures:
tests::another

④test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
error: test failed
```

示例11-4：测试结果显示一个测试通过、一个测试失败

与之前不同，结果中`test tests::another`字段后出现了FAILED❶而不是ok。另外，在测试结果与摘要之间新增了两段信息。第一段❷展示了每个测试失败的详细原因。在本例中，测试`another`因为在`src/lib.rs`文件的第十行发生了`panicked at 'Make this test fail'`而导致失败。第二段❸则列出了所有失败测试的名称，它可以帮助我们在输出的众多信息中定位到具体的失败测试。我们可以通过指定名称来单独运行对应的测试，以便更容易地定位错误。我们会在“控制测试的运行方式”一节中讨论这部分内容。

测试摘要依旧显示在输出结尾处❹：总的来说，我们的测试结果为FAILED。其中有1个测试成功，1个测试失败。

好了，现在你已经见过不同场景下可能输出的测试结果。接下来让我们继续讨论除`panic!`之外的一些在测试工作中十分有用的宏。

使用`assert!`宏检查结果

`assert!`宏由标准库提供，它可以确保测试中某些条件的值为`true`。`assert!`宏可以接收一个能够被计算为布尔类型的值作为参数。当这个值为`true`时，`assert!`宏什么都不用做并正常通过测试。而当值为`false`时，`assert!`宏就会调用`panic!`宏，进而导致测试失败。使用`assert!`宏可以检查代码是否按照我们预期的方式运行。

在第5章的示例5-15中，我们曾经使用过`Rectangle`结构体及其`can_hold`方法。让我们将这些代码加入`src/lib.rs`文件中，并利用`assert!`宏来为它编写一些测试，如示例11-5所示。

src/lib.rs

```
# [derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

示例11-5：使用第5章中的Rectangle结构体及其can_hold方法

can_hold方法会返回一个布尔值，这意味着它完美地符合使用assert! 宏的场景。在示例11-6中，我们使用can_hold方法编写了一个测试。它会创建一个长为8、宽为7的Rectangle实例，并断言自身可以容纳另外一个长为5、宽为1的Rectangle实例。

src/lib.rs

```
# [cfg(test)]
mod tests {
    ❶ use super::*;

    #[test]
    ❷ fn larger_can_hold_smaller() {
        ❸ let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        ❹ assert!(larger.can_hold(&smaller));
    }
}
```

示例11-6：这个测试会调用can_hold来检查一个矩形是否可以容纳另外一个小的矩形

注意，我们在tests模块中新增加了一行：use super::*;

tests模块与其他模块没有任何区别，它同样遵循第7章的“用于在模块树中指明条目的路径”一节中介绍的可见性规则。因为tests是一个内部模块，所以我们必须将外部模块中的代码导入内部模块的作用域中。这里使用了通配符（*）让外层模块所定义的全部内容在tests模块中都可用。

我们将这个测试命名为larger_can_hold_smaller❷，并在测试中按需创建了两个Rectangle实例❸。接着，我们又将表达式larger.can_hold(&smaller)的结果作为参数传递给了assert!宏❹。由于这个表达式理论上会返回true，所以示例11-6中的测试应该可以顺利地通过。让我们试试看吧：

```
running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

它顺利通过测试了！我们不妨再来增加一个测试用例，并断言较小的矩形不能容纳较大的矩形：

src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        // --略

        --
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(!smaller.can_hold(&larger));
    }
}
```

由于新测试用例中的can_hold函数应当返回false作为结果，所以我们需要在将它传递给assert!宏之前执行取反操作。这段测试会在can_hold返回false时顺利通过：

```
running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

两个测试都通过了！但如果测试结果出现错误的话，输出日志又会是怎样的呢？修改can_hold方法的实现，将代码中用于比较长度的大于号更换为小于号：

```
// --略  
--  
  
impl Rectangle {  
    pub fn can_hold(&self, other: &Rectangle) -> bool {  
        self.length < other.length && self.width > other.width  
    }  
}
```

再次运行测试，你将会看到如下所示的输出：

```
running 2 tests  
test tests::smaller_cannot_hold_larger ... ok  
test tests::larger_can_hold_smaller ... FAILED  
  
failures:  
  
---- tests::larger_can_hold_smaller stdout ----  
thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed:  
    larger.can_hold(&smaller)', src/lib.rs:22:8  
note: Run with `RUST_BACKTRACE=1` for a backtrace.  
  
failures:  
    tests::larger_can_hold_smaller  
  
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

我们的测试成功捕捉到了代码错误！因为larger.length等于8，而smaller.length等于5，所以can_hold中比较长度的判断条件8<5不成立，从而返回了false并导致断言失败。

使用assert_eq! 宏和assert_ne! 宏判断相等性

在对功能进行测试时，我们常常需要将被测试代码的结果与我们所期望的结果相比较，并检查它们是否相等。你可以利用assert!宏，向其中传入一个使用==运算符的判断表达式来完成这项测试。因为这项测试比较常见，所以标准库中专门提供了一对可以简化编程的宏：assert_eq! 和assert_ne!。这两个宏分别用于比较并断言两个参数相等或不相等。在断言失败时，它们还可以自动打印出两个参数的值，从而方便我们观察测试失败的原因；相反，使用assert!宏则只能得知==判断表达式失败的事实，而无法知晓被用于比较的值。

在示例11-7中，我们编写了一个名为add_two的函数，它会将输入的参数加2并返回结果。接下来，我们使用assert_eq! 宏对这个函数

进行测试。

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

示例11-7：使用assert_eq! 宏对add_two函数进行测试

让我们检查一下测试结果是否通过：

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

我们传入assert_eq! 宏的第一个参数是4，而它和第二个参数，也就是add_two(2)的返回值相等。输出日志test tests::it_adds_two ... ok中的ok表明，这条测试顺利地通过了检查！

接下来，让我们在代码中引入错误，并看一看assert_eq! 宏断言失败后的结果。修改add_two函数的实现，让它加3：

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

再次运行测试：

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
❶ thread 'tests::it_adds_two' panicked at 'assertion failed:
` (left == right)`
❷ left: `4`,
❸ right: `5`, src/lib.rs:11:8
```

```
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

我们的测试成功捕捉到了代码错误！这段信息指出`it_adds_two`测试失败，它不但显示了失败原因`assertion failed: `(left == right)``❶，还将对应的参数值打印了出来：`left`为4❷，`right`为5❸。这样的日志可以帮助我们立即开始调试工作：它意味着传递给`assert_eq!`宏的`left`参数为4，而`right`参数，也就是`add_two(2)`的值，是5。

注意，在某些语言或测试框架中，这两个被用于相等性判断的参数常常被命名为`expected`（期待值）和`actual`（实际值），我们需要在指定参数时留意先后顺序。不过在Rust中，我们将它们称为`left`（左值）和`right`（右值），你无须在意指定期望值和实际值时的具体顺序。上面代码中的断言写成`assert_eq!(add_two(2), 4)`也没有任何问题，它在运行时依然会输出错误提示信息`assertion failed: `(left == right)``，并指明`left`为5，而`right`为4。

相对应地，`assert_ne!`宏在两个值不相等时通过，相等时失败。当我们无法预测程序的运行结果，却可以确定它绝不可能是某些值的时候，就可以使用这个宏来进行断言。例如，假设被测试的函数保证自己会以某种方式修改输入的值，但这种修改方式是由运行代码时所处的日期来决定的，那么在这种情形下最好的测试方式就是断言函数的输出结果和输入的值不相等。

从本质上来看，`assert_eq!`和`assert_ne!`宏分别使用了`==`和`!=`运算符来进行判断，并在断言失败时使用调试输出格式`({:?})`将参数值打印出来。这意味着它们的参数必须同时实现`PartialEq`和`Debug`这两个trait。所有的基本类型和绝大多数标准库定义的类型都是符合这一要求的。而对于自定义的结构体和枚举来说，你需要自行实现`PartialEq`来判断两个值是否相等，并实现`Debug`来保证值可以在断言失败时被打印出来。第5章的示例5-12中曾提到过，由于这两个trait都是可派生trait，所以它们一般可以通过在自定义的结构体或枚举的定义上方添加`#[derive(PartialEq, Debug)]`标注来自动实现这两个trait。你可以参阅附录C来了解有关自动实现trait的更多细节。

添加自定义的错误提示信息

你也可以添加自定义的错误提示信息，将其作为可选的参数传入 assert!、assert_eq! 或 assert_ne! 宏。实际上，任何在 assert!、assert_eq! 或 assert_ne! 的必要参数之后出现的参数都会一起被传递给 format! 宏（我们曾经在第8章的“使用+运算符或 format! 宏来拼接字符串”一节中讨论过）。因此，你甚至可以将一个包含 {} 占位符的格式化字符串及相对应的填充值作为参数一起传递给这些宏。自定义的错误提示信息可以很方便地记录当前断言的含义；这样在测试失败时，我们就可以更容易地知道代码到底出了什么问题。

例如，假设有一个函数会接收客人姓名作为参数，并返回拼接的问候语作为结果。现在，我们需要通过测试来确定姓名确实出现在了问候语中：

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"));
    }
}
```

这个程序的需求还未被最终确定，问候语起始处的Hello文本极有可能会发生改变。我们希望在每次修改需求时避免修改对应的测试用例，因此在测试的断言问候语中仅包含了正确的姓名，而不要求输出结果和某个正确答案完全相等。

接下来，让我们在代码中引入bug，并观察测试失败时会发生什么。修改代码，把姓名从问候语中去掉：

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

运行测试，得到如下所示的结果：

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
    thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains("Carol")', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::greeting_contains_name
```

这个测试结果仅仅表明了在代码的某一行发生了断言失败。在本例中，一个更加友好的错误提示信息应该将我们从greeting函数中获得的结果值打印出来。现在让我们修改一下测试函数，指定自定义的错误提示信息。该信息由一个包含占位符的格式化字符串，以及greeting函数的实际返回值组成：

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{:?}`", result
    );
}
```

再次运行测试，我们应该可以看到更具有实际意义的错误提示信息：

```
---- tests::greeting_contains_name stdout ----
    thread 'tests::greeting_contains_name' panicked at 'Greeting did not
contain name, value was `Hello!`', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这次的测试输出中包含了实际的值，它能帮助我们观察程序真正发生的行为，并迅速定位与预期产生差异的地方。

使用should_panic检查panic

除了检查代码是否返回了正确的结果，确认代码能否按照预期处理错误状况同样重要。以第9章示例9-9中的Guess类型为例：使用Guess类型的相关代码的前提是Guess实例只会包含处于1至100范围内

的值。那么，我们可以编写一个测试来检查使用了非法值的Guess的创建过程是否会如期发生panic。

我们需要为测试函数添加一个额外的新属性：should_panic。标记了这个属性的测试函数会在代码发生panic时顺利通过，而在代码不发生panic时执行失败。

示例11-8展示了一段用于检测Guess::new是否按照预期处理错误的测试用例。

src/lib.rs

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> 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： 测试一个应当引发panic! 的条件

我们将#[should_panic]属性放在了#[test]属性之后、对应的测试函数之前。让我们看一看测试顺利通过时的样子：

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

非常好！接下来让我们在代码中引入bug，删除new函数中值大于100时发生panic的判断条件：

```
// --略
--



impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }
}
```

当我们再次运行示例11-8中的测试时，它应该会输出测试失败的结果：

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:
failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

这次测试的输出中似乎并没有包含太多有用的信息，但当我们观察测试函数时，会发现它被标注了#[`should_panic`]。这也就意味着测试函数中的代码并没有如期地产生一个panic。

使用`should_panic`进行的测试可能会有些含糊不清，因为它们仅仅能够说明被检查的代码会发生panic。即便函数中发生panic的原因与我们预期的不同，使用`should_panic`进行的测试也会顺利通过。为了让`should_panic`测试更加精确一些，我们可以在`should_panic`属性中添加可选参数`expected`。它会检查panic发生时输出的错误提示信息是否包含了指定的文字。仍然以`Guess`类型为例，让我们稍微修改一下`new`函数，使`new`函数根据其参数值过大或过小而提供不同的panic信息，如示例11-9所示。

src/lib.rs

```

// --略
--



impl Guess {
    pub fn new(value: u32) -> 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 = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}

```

示例11-9：测试某个条件会触发带有特定错误提示信息的panic!

因为 Guess::new 函数在发生 panic 的输出消息中包含了 should_panic 属性的 expected 参数指定的文本，所以该示例中的测试会顺利通过。实际上，我们在测试时匹配完整的 panic 信息（Guess value must be less than or equal to 100, got 200）也是可以的。一般来说，expected 参数中的内容既取决于 panic 信息是明确的还是易变的，也取决于测试本身需要准确到何种程度。在本例中，panic 信息的子字符串就足以确保测试函数中的代码运行的是 else if value > 100 分支下的了。

为了观察指定了 expected 参数的 should_panic 测试在失败时会发生什么，我们再次向代码中引入 bug，将 if value < 1 与 else if value > 100 两个分支中的代码块交换一下：

```

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);
}

```

这时再次运行should_panic测试就会失败：

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
    thread 'tests::greater_than_100' panicked at 'Guess value must be
greater than or equal to 1, got 200.', src/lib.rs:11:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string 'Guess value must be 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
```

这段错误提示信息表明，测试确实如期地发生了panic，但panic所附带的消息却没有包含期望的字符串'Guess value must be less than or equal to 100'。实际上，本例中我们所获得的panic信息是'Guess value must be greater than or equal to 1, got 200'。我们可以从这些信息着手来排查bug！

使用Result<T, E>编写测试

到目前为止，我们编写的测试都会在运行失败时触发panic。不过我们也可以用Result<T, E>来编写测试！我们在这里使用Result<T, E>重写示例11-1中的测试，让它在运行失败时返回一个Err值而不是触发panic：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

it_works函数现在会返回一个Result<(), String>类型的值。在函数体中，我们不再调用assert_eq! 宏，而是在测试通过时返回Ok(())，在失败时返回一个带有String的Err值。

像这样编写返回Result<T, E>的测试，就可以在测试函数体中使用问号运算符了。这样可以方便地编写一个测试，该测试在任意一个步骤返回Err值时都会执行失败。

不要在使用Result<T, E>编写的测试上标注#[should_panic]。在测试运行失败时，我们应当直接返回一个Err值。

在本节中，我们学习了几种编写测试的方法。接下来，我们还会讨论运行测试时会发生什么，并向你介绍更多可用于cargo test命令的选项。

控制测试的运行方式

如同cargo run会编译代码并运行生成的二进制文件一样，cargo test同样会在测试模式下编译代码，并运行生成的测试二进制文件。你可以通过指定命令行参数来改变cargo test的默认行为。例如，cargo test生成的二进制文件默认会并行执行所有的测试，并截获测试运行过程中产生的输出来让与测试结果相关的内容更加易读。

我们既可以为cargo test指定命令行参数，也可以为生成的测试二进制文件指定参数。为了区分两种不同类型的参数，你需要在传递给cargo test的参数后使用分隔符--，并在其后指定需要传递给测试二进制文件的参数。例如，运行cargo test --help会显示出cargo test的可用参数，而运行cargo test -- --help则会显示出所有可以用在--之后的参数。

并行或串行地进行测试

当你尝试运行多个测试时，Rust会默认使用多线程来并行执行它们。这样可以让测试更快地运行完毕，从而尽早得到代码是否能正常工作的反馈。但由于测试是同时进行的，所以开发者必须保证测试之间不会互相依赖，或者依赖到同一个共享的状态或环境上，例如当前工作目录、环境变量等。

举个例子，假设当前所有测试都会运行代码去创建名为*test-output.txt*的文本文件并写入不同的数据，紧接着它们又会读取文件中的内容，并断言该内容与自己写入的数据相等。如果我们并行运行这些测试，那么可能就会出现测试A覆盖了测试B所写入的数据，进而导致测试B在随后的指令中发生断言失败。但这并不是因为测试B的代码真的有错，而是因为多个测试并行运行时互相产生了影响。一种解

解决方案是使不同的测试指向不同的文件，另一种解决方案则是顺序执行这些测试。

如果你不想并行运行测试，或者希望精确地控制测试时所启动的线程数量，那么可以通过给测试二进制文件传入`--test-threads`标记及期望的具体线程数量来控制这一行为。来看下面的例子：

```
$ cargo test -- --test-threads=1
```

在上面的命令中，我们将线程数量限制为1，这也就意味着程序不会使用任何并行操作。使用单线程执行测试会比并行执行花费更多的时间，但顺序执行的测试不会再因为共享状态而出现可能的干扰情形了。

显示函数输出

默认情况下，Rust的测试库会在测试通过时捕获所有被打印至标准输出中的消息。例如，即便我们在测试中调用了`println!`，但只要测试顺利通过，它所打印的内容就无法显示在终端上；我们只能看到一条用于指明测试通过的消息。只有在测试失败时，我们才能在错误提示信息的上方观察到打印至标准输出中的内容。

举一个例子，示例11-10中包含了一个没有实际用处的函数，它会打印出输入的参数值并返回一个固定值10，另外还包含两个测试，一个会顺利通过，而另一个则会失败。

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!(10, value);
    }
}
```

```

#[test]
fn this_test_will_fail() {
    let value = prints_and_returns_10(8);
    assert_eq!(5, value);
}
}

```

示例11-10： 测试一个调用了println! 的函数

使用cargo test运行测试，会得到如下所示的结果：

```

running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
❶ I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left == right)`
  left: `5`,
  right: `10``, src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

```

注意，我们无法在这段输出中找到信息I got the value 4，它虽然在通过的测试样例中被打印了出来，但却被Rust截获后丢弃了。而I got the value 8❶则正常出现在了测试失败的摘要中，和测试失败的原因一起显示出来了。

假如你希望在测试通过时也将值打印出来，那么可以传入--nocapture标记来禁用输出截获功能：

```
$ cargo test -- --nocapture
```

在示例11-10中传入--nocapture标记后，再次运行代码，输出的结果如下所示：

```

running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed: `(left == right)`
  left: `5`,
  right: `10``, src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

```

```
failures:  
  
failures:  
    tests::this_test_will_fail  
  
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

值得注意的是，测试的输出和测试结果相互交叉在了一起；这是因为测试是并行运行的。你可以自行尝试使用`--test-threads=1`选项和`--nocapture`标记来运行测试，并观察输出的结果。

只运行部分特定名称的测试

执行全部的测试用例有时会花费很长时间。而在编写某个特定部分的代码时，你也许只需要运行和代码相对应的那部分测试。我们可以通过向`cargo test`中传递测试名称来指定需要运行的测试。

为了演示如何运行部分测试，我们在示例11-11中为`add_two`函数创建了3个测试。

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {  
    a + 2  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;

    #[test]  
    fn add_two_and_two() {  
        assert_eq!(4, add_two(2));  
    }

    #[test]  
    fn add_three_and_two() {  
        assert_eq!(5, add_two(3));  
    }

    #[test]  
    fn one_hundred() {  
        assert_eq!(102, add_two(100));  
    }
}
```

示例11-11：3个不同名称的测试

如果我们在运行测试时不传递任何参数，那么正如之前看到的一样，所有的测试都会并行运行：

```
running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

运行单个测试

我们可以给cargo test传递一个测试函数的名称来单独运行该测试：

```
$ cargo test one_hundred

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

只有名为one_hundred的测试得到了运行，因为其余两个测试的名称无法匹配我们传入的参数。同时，测试输出还通过摘要一行中的2 filtered out表明部分测试被过滤掉了。

需要注意的是，我们不能指定多个参数来运行多个测试；只有传递给cargo test的第一个参数才会生效。运行多个测试需要使用其他方法。

通过过滤名称来运行多个测试

实际上，我们可以指定测试名称的一部分来作为参数，任何匹配这一名称的测试都会得到执行。例如，因为有两个测试名称中都包含add，所以我们可以命令cargo test add来运行它们。

```
$ cargo test add

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
```

```
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

这个命令运行了所有名称中带有 add 的测试，并将名为 one_hundred 的测试过滤掉了。另外要注意，测试所在的模块的名称也是测试名称的一部分，所以我们可以使用模块名来运行特定模块内的所有测试。

通过显式指定来忽略某些测试

有时，一些特定的测试执行起来会非常耗时，所以你可能会想要在大部分的 cargo test 命令中忽略它们。除了手动将想要运行的测试列举出来，你也可以使用 ignore 属性来标记这些耗时的测试，将这些测试排除在正常的测试运行之外，如下所示：

src/lib.rs

```
# [test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // 需要运行一个小时的代码
}
```

对于想要剔除的测试，我们会在 #[test] 标记的下方添加 #[ignore] 行。现在，当我们运行测试时，只有 it_works 得到了执行，而 expensive_test 则被跳过了：

```
$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

`expensive_test` 函数被放到了 `ignored` 类别下。我们可以使用 `cargo test -- --ignored` 来单独运行这些被忽略的测试：

```
$ cargo test -- --ignored

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

通过控制测试的运行，我们可以保证每次执行 `cargo test` 都能迅速得到结果。而对于忽略的测试，我们则可以在时间充裕时通过执行 `cargo test -- --ignored` 来运行。

测试的组织结构

正如本章一开始就提到过的，测试是一门复杂的学科，测试的技术名词和组织方法也因人而异。Rust社区主要从以下两个分类来讨论测试：单元测试（unit test）和集成测试（integration test）。单元测试小而专注，每次只单独测试一个模块或私有接口。而集成测试完全位于代码库之外，和正常从外部调用代码库一样使用外部代码，只能访问公共接口，并且在一次测试中可能会联用多个模块。

为了确保代码库无论是独立的还是作为一个整体都能如期运行，编写单元测试和集成测试是非常重要的工作。

单元测试

单元测试的目的在于将一小段代码单独隔离出来，从而迅速地确定这段代码的功能是否符合预期。我们一般将单元测试与需要测试的代码存放在*src* 目录下的同一文件中。同时也约定俗成地在每个源代码文件中都新建一个*tests*模块来存放测试函数，并使用`cfg(test)`对该模块进行标注。

测试模块和`[cfg(test)]`

在*tests*模块上标注`[cfg(test)]`可以让Rust只在执行`cargo test`命令时编译和运行该部分测试代码，而在执行`cargo build`时剔除它们。这样就可以在正常编译时不包含测试代码，从而节省编译时间和产出物所占用的空间。我们不需要对集成测试标注`[cfg(test)]`，因为集成测试本身就放置在独立的目录中。但是，由于单元测试是和业务代码并列放置在同一文件中的，所以我们必须使用`[cfg(test)]`进行标注才能将单元测试的代码排除在编译产出物之外。

回忆一下本章开始时创建的adder项目，Cargo为我们自动生成了如下所示的代码：

src/lib.rs

```
# [cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

上述代码就是自动生成的测试模块。其中的`cfg`属性是配置(`configuration`)一词的英文缩写，它告知Rust接下来的条目只有在处于特定配置时才需要被包含进来。本例中指定的`test`就是Rust中用来编译、运行测试的配置选项。通过使用`cfg`属性，Cargo只在运行`cargo test`时才会将测试代码纳入编译范围。这一约定不止针对那些标注了`#[test]`属性的测试函数，还针对该模块内的其余辅助函数。

测试私有函数

软件测试社区对于是否应当直接测试私有函数一直存在争议。在某些语言中，测试私有函数往往是困难的，甚至是不可能的。不过无论你在软件测试上持有何种观点，Rust都通过私有性规则的设计，允许测试私有函数。示例11-12中的代码编写并测试了一个私有函数`internal_adder`。

src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

# [cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

示例11-12：测试一个私有函数

注意，上面代码中的internal_adder函数没有被标注为pub，但因为测试本身就是Rust代码，并且tests模块就是Rust模块，所以你可以正常地将internal_adder导入测试作用域并调用它。当然，如果你认为不应当测试私有函数，那么Rust也不会强迫你做这些事情。

集成测试

在Rust中，集成测试是完全位于代码库之外的。集成测试调用库的方式和其他的代码调用方式没有任何不同，这也意味着你只能调用对外公开提供的那部分接口。集成测试的目的在于验证库的不同部分能否协同起来正常工作。能够独立正常工作的单元代码在集成运行时也会发生各种问题，所以集成测试的覆盖率同样是非常重要的。为了创建集成测试，你首先需要建立一个*tests* 目录。

tests目录

我们需要在项目根目录下创建*tests* 文件夹，它和*src* 文件夹并列。Cargo会自动在这个目录下寻找集成测试文件。我们可以在这个目录下创建任意多个测试文件，Cargo在编译时会将每个文件都处理为一个独立的包。

现在让我们开始创建一个集成测试。保留示例11-12中的*src/lib.rs* 文件，创建一个*tests* 文件夹，并创建文件*tests/integration_test.rs*，将示例11-13中的代码输入其中。

tests/integration_test.rs

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

示例11-13：adder包中某个函数的集成测试

与单元测试不同，集成测试需要在代码顶部添加语句`use adder`。这是因为`tests` 目录下的每一个文件都是一个独立的包，所以我们需要将目标库导入每一个测试包中。

我们不需要为`tests/integration_test.rs` 中的任何代码标注`# [cfg(test)]`。Cargo对`tests` 目录进行了特殊处理，它只会在执行`cargo test`命令时编译这个目录下的文件。让我们执行`cargo test`并观察输出结果：

```
$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running target/debug/deps/adder-abcabcabc

❶running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

❷ Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
❸test it_adds_two ... ok

❹test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

上面的输出中出现了单元测试、集成测试和文档测试这3部分。单元测试部分❶与我们之前见到的一样：每行输出一个单元测试结果（这个被称为`internal`的测试是我们在示例11-12中添加的），并在后面给出单元测试的摘要。

集成测试部分从输出行`Running target| debug| deps| integration-test-ce99bcc2479f4607`❷（末尾的哈希值会有所不同）开始。接着，它会为当前集成测试中的每一个测试函数使用单独一行输出结果❸，并在`Doc-tests adder`部分开始前给出集成测试的摘要❹。

我们添加的单元测试函数越多，在输出中的单元测试部分产生的结果行就越多。与之类似，集成测试文件中添加的测试函数越多，对

应的集成测试文件部分产生的结果行就越多。每一个集成测试文件都会在输出中有自己独立的区域，所以我们在 *tests* 目录下添加的文件越多，出现的集成测试区域就越多。

我们仍然可以在 cargo test 命令中指定测试函数名称作为参数，来运行特定的集成测试函数。另外，在执行 cargo test 时使用 --test 并指定文件名，可以单独运行某个特定集成测试文件下的所有测试函数：

```
$ cargo test --test integration_test

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

这条命令只运行了 *tests/integration_test.rs* 文件中的测试。

在集成测试中使用子模块

随着集成测试的增加，你也许希望把 *tests* 目录下的代码分离到多个文件中，以便更好地管理它们。例如，你可以根据待测函数的功能将测试函数分组。就像之前提到的一样，每一个 *tests* 目录中的文件都会被编译为各自独立的包。

将每个集成测试的文件编译成独立的包有助于隔离作用域，并使集成测试环境更加贴近于用户的使用场景。但是，这同时意味着我们在第7章学习如何将代码分离为模块和文件时，所学到的 *src* 目录下的文件的处理规则并不完全适用于 *tests* 目录。

当你有一些可用于多个集成测试文件的辅助函数，且想要尝试按照第7章的“将模块拆分为不同的文件”一节中的步骤将它们提取到通用的模块中时，*tests* 目录的这种特殊行为就会显得异常明显。例如，假设我们创建了一个新文件 *tests/common.rs*，在该文件中编写了一个名为 *setup* 的函数，并希望在多个不同的集成测试文件中调用它：

tests/common.rs

```
pub fn setup() {  
    // 一些测试工作中可能会用到的初始化代码  
}
```

即便我们没有在这个文件中包含任何测试函数，也没有在任何地方调用过setup函数，也依然会在运行测试后的测试输出中观察到一段与*common.rs* 文件相关的区域：

```
running 1 test  
test tests::internal ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
    Running target/debug/deps/common-b8b07b6f1be2db70  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
    Running target/debug/deps/integration_test-d993c68b431d39df  
  
running 1 test  
test it_adds_two ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
    Doc-tests adder  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

让*common*出现在测试输出中，并显示毫无意义的running 0 tests 消息可不是我们想要的，我们只是想要在多个集成测试文件之间共享一些代码而已。

为了避免*common*出现在测试结果中，我们可以创建*tests/common/mod.rs*，而不再创建*tests/common.rs*。这是另一种可以被Rust理解的命名规范。通过采用这种文件命名方式，Rust就不会再将*common*模块视作一个集成测试文件了。当我们把*setup*函数移动至*tests/common/mod.rs*中并删除*tests/common.rs*文件后，测试输出中就再也不会出现与*common*相关的区域了。*tests*子目录中的文件不会被视作单独的包进行编译，更不会在测试输出中拥有自己的区域。

当 *tests/common/mod.rs* 创建完毕后，我们就可以将其视作一个普通的模块并应用到不同的集成测试文件中了。接下来的示例在 *tests/integration_test.rs* 文件的集成测试函数 `it_adds_two` 中调用了 `setup` 函数：

`tests/integration_test.rs`

```
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

就像示例 7-21 中的模块声明一样，这段代码中的 `mod common;` 语句声明了需要引用的模块。接着，我们就可以在测试函数中正常调用 `common::setup()` 函数了。

二进制包的集成测试

如果我们的项目是一个只有 *src/main.rs* 文件而没有 *src/lib.rs* 文件的二进制包，那么我们就无法在 *tests* 目录中创建集成测试，也无法使用 `use` 语句将 *src/main.rs* 中定义的函数导入作用域。只有代码包（library crate）才可以将函数暴露给其他包来调用，而二进制包只被用于独立执行。

这就是 Rust 的二进制项目经常会把逻辑编写在 *src/lib.rs* 文件中，而只在 *src/main.rs* 文件中进行简单调用的原因。这种组织结构使得集成测试可以将我们的项目视作一个代码包，并能够使用 `use` 访问包中的核心功能。只要我们能够保证核心功能一切正常，*src/main.rs* 中的少量胶水代码就能够工作，无须进行测试。

总结

Rust的测试功能提供了一种可以指定函数行为的方式。即便实现方式发生了改变，它也能够保证函数会继续按照我们期望的方式去工作。单元测试可以独立地验证库中的不同部分，并可以测试私有实现细节。集成测试则可以检查库内的各个部分能否正确地协同工作，它们与外部代码一样，只会访问库中的公共API。尽管Rust的类型系统和所有权规则能够帮助我们避免一些bug，但测试依旧必不可少，它对于减少代码中的逻辑错误并避免不符合预期的行为非常重要。

接下来，让我们结合在本章和前几章中学到的内容来编写一个实际的项目吧！

第12章

I/O项目：编写一个命令行程序



本章不仅会回顾此前学习过的众多知识，还会向你介绍一些新的标准库功能。我们将开发一个能够和文件系统交互并处理命令行输入、输出的工具。你会在这个过程中不断地复习到那些已经接触过的Rust概念。

Rust语言非常适合用来编写命令行工具，因为它具有快速、安全、跨平台及产出物为单一二进制文件的特点。在本章的实践项目中，我们会重新实现经典的命令行工具grep（globally search a regular expression and print，全局正则搜索与输出），而它最简单的使用场景就是在特定文件中搜索指定的字符串。为此，grep会接收一个文件名和一个字符串作为参数，然后在执行时读取文件来搜索包含指定字符串的行，并最终将这些行打印输出。

在本章中，我们会演示如何像其他的命令行工具一样使用各种终端特性。我们会读取环境变量，使用户可以对工具的行为进行配置。我们还会学习如何将信息打印至标准错误流（stderr）而不是标准输出流（stdout），这一功能使用户可以将正常输出重定向到文件的同时仍然可以在屏幕上看到错误提示信息。

值得一提的是，Rust社区中的成员Andrew Gallant已经实现了一个功能完备且性能极佳的grep替代品：`ripgrep`。相比之下，本章所编写的grep要简单得多，但我们会试图让你接触到足够多的背景知识，为理解现实世界中像`ripgrep`这样复杂的项目做好准备。

我们的grep项目将会包含目前为止学习过的一些概念：

- 组织代码（通过使用在第7章接触到的模块）
- 使用动态数组和字符串（第8章，集合类型）
- 错误处理（第9章）
- 合理地使用trait和生命周期（第10章）
- 编写测试（第11章）

我们还会简要地介绍闭包、迭代器和trait对象，有关这些知识的详细内容可以在第13章和第17章找到。

接收命令行参数

让我们一如既往地使用cargo new来建立一个新的项目。为了避免和系统中可能已经内置的grep相混淆，我们将这个项目命名为minigrep。

```
$ cargo new minigrep
$ cd minigrep
     Created binary (application) `minigrep` project
```

实现这一工具的首要任务是让minigrep接收两个命令行参数：文件名和用于搜索的字符串。也就是说，我们希望通过依次输入cargo run、用于搜索的字符串及文件路径的命令行来运行程序，例如：

```
$ cargo run searchstring example-filename.txt
```

通过cargo new自动生成出来的初始程序不会处理任何传递给它的参数。crates.io上有一些现成的库可以帮助开发者编写接收命令行参数的程序，但是由于你刚刚开始学习这些概念，所以我们会从零开始自行实现这些功能。

读取参数值

为了使minigrep可以读取传递给它的命令行参数值，我们需要使用Rust标准库提供的std::env::args函数。这个函数会返回一个传递给minigrep的命令行参数迭代器(iterator)。我们会在第13章深入介绍迭代器，目前，你只需要知道两个有关它的细节：迭代器会产生一系列的值，而我们可以通过调用迭代器的collect方法来生成一个包含所有产出值的集合，比如动态数组。

示例12-1中的代码使minigrep程序可以读取所有传递给它的命令行参数值，并将它们收集到一个动态数组中。

src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{} {}", args);
}
```

示例12-1：将命令行参数收集到一个动态数组中并打印出来

首先，使用use语句将std::env模块引入当前作用域，以便我们调用其中的args函数。注意，std::env::args函数被嵌套于两层模块内。正如在第7章所讨论的，当所需函数被嵌套于不止一层模块中时，我们通常只将其父模块引入作用域，而不将函数本身引入。这便于我们使用std::env模块中的其他函数。另外，使用use std::env::args;之后直接调用args函数的做法也容易引发歧义，因为单独的args容易被误认为定义于当前模块中的函数。

args函数与非法的Unicode字符

注意，std::env::args函数会因为命令行参数中包含了非法的Unicode字符而发生panic。如果你确实需要在程序中接收包含非法Unicode字符的参数，那么请使用std::env::args_os函数。这个函数会返回一个产出OsString值（而不是String值）的迭代器。我们在本章选择使用std::env::args是为了简单起见，因为OsString值会因平台而异，处理起来也会比String值更加复杂。

我们在main函数的第一行调用了env::args，并立刻使用collect函数将迭代器转换成一个包含所有迭代器产出值的动态数组。由于collect函数可以被用来创建多种不同的集合，所以我们显式地标注了args的类型来获得一个包含字符串的动态数组。尽管在Rust中我们极少需要标注类型，但因为Rust无法推断出想要的具体集合类型，所以我们常常需要为collect函数进行手动标注。

最后，我们使用了调试格式`:? 来打印动态数组中的内容。现在，让我们分别试一试在不加参数并附带2个参数的情形下运行示例12-1中的代码：`

```
$ cargo run  
--略  
--  
["target/debug/minigrep"]  
$ cargo run needle haystack  
--略  
--  
["target/debug/minigrep", "needle", "haystack"]
```

注意，动态数组中的第一个值是“`target debug minigrep`”，也就是当前执行的二进制文件名称。这和C语言处理参数列表时的行为是一致的，程序可以通过这个参数在运行时获得自己的名称。这一功能可以让我们方便地在输出信息中打印程序名称，或者根据程序名称的不同而改变行为等。但是考虑到本章的目的，我们将会忽略这个参数并只存储我们需要的两个参数。

将参数值存入变量

将动态数组打印出来表明当前程序能够获取命令行参数指定的值。现在，将这两个参数的值保存至变量中，以便我们在程序的其余部分使用，如示例12-2所示。

src/main.rs

```
use std::env;  
  
fn main() {  
    let args: Vec<String> = env::args().collect();  
  
    let query = &args[1];  
    let filename = &args[2];  
  
    println!("Searching for {}", query);  
    println!("In file {}", filename);  
}
```

示例12-2：创建变量来存储查询参数和文件名参数

正如打印动态数组时所观察到的，程序名占据了动态数组中的第一个元素，也就是`args[0]`，所以我们需要从1开始计算数组下标。`minigrep`接收的第一个参数是待搜索的字符串，我们将它的引用存入了变量`query`。第二个参数是文件名，我们将它的引用存入了变量`filename`。

我们临时将这两个变量的值打印出来，以便检查程序工作是否正常。让我们使用`test`和`sample.txt`作为参数，再次运行这个程序：

```
$ cargo run test sample.txt

Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

很好，程序工作正常！这些必要的参数值已经被存入对应的变量中。我们会在本章稍后一些的地方来增加错误处理以应对可能出现的异常情况，比如用户没有输入任何参数的情况等。现在，让我们暂时忽略这类问题，开始为程序添加读取文件内容的功能。

读取文件

在获取了指定文件的命令行参数filename后，我们现在可以来编写读取文件的功能了。首先，我们需要一个可供测试的样例文件。对于开发minigrep来说，这个文件最好拥有多行文本但字符量不要太大，且各行文本中存在重复的单词。示例12-3中是一首Emily Dickinson [1] 的诗，它恰好满足了我们对样例文件的所有要求。在项目根目录下创建poem.txt文件，并将I'm Nobody! Who are you?这首诗的内容输入其中。

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：Emily Dickinson的诗，同时也是一个不错的测试用例

有了测试文本之后，就可以开始编辑src/main.rs并添加读取文件的代码了，如示例12-4所示。

src/main.rs

```
use std::env;  
①use std::fs;  
  
fn main() {  
    // --略  
  
    --  
    println!("In file {}", filename);  
  
    ② let contents = fs::read_to_string(filename)
```

```
.expect("Something went wrong reading the file");

❸ println!("With text:\n{}", contents);
}
```

示例12-4：读取第二个参数所指定文件中的内容

这段代码额外地增加了一条use语句来引入标准库中的std::fs模块，它被用来处理与文件相关的事务❶。

随后，我们在main中新增了一条语句：其中的fs::read_to_string函数接收filename作为参数，它会打开对应文件并使用Result<String>类型返回文件的内容❷。

最后，为了检查程序工作是否正常，我们增加了一条临时的println!语句，它会在读取文件后打印出contents变量中的值❸。

尝试运行这段程序，先随便指定一个字符串作为命令行的第一个参数（因为我们还没有实现搜索功能），并指定文件名poem.txt作为第二个参数：

```
$ cargo run the poem.txt

Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
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!
```

很好！这段代码成功地读取并打印出了文件中的内容。但需要注意的是，它依然存在不少瑕疵。目前，main函数中实现了多个功能，但通常而言，只负责单个功能的函数会更加简捷并易于维护一些。另外，我们没有尽可能地处理各种错误。虽然现有的程序还比较小，这些瑕疵处理起来也不算棘手，但随着程序规模逐渐增长，我们将会越

越来越难以用简单的方式去修复它们。尽早重构是软件开发中的最佳实践，毕竟代码越少，重构就越简单。接下来，我们就要做这件事情。

[1] 译者注：艾米莉·狄金森（1830年—1886年），美国诗人。

重构代码以增强模块化程度和错误处理能力

为了改进当前的程序，我们计划修复4个涉及程序架构及错误处理的问题。

首先，当前的main函数会同时负责解析命令行参数和读取文件两项工作。这对于一个如此小巧的函数也许还算不上什么大问题，但如果我们继续在main函数中增加功能，它所处理的独立任务就会越来越多。而随着函数功能的增多，它也会变得越来越令人难以理解，难以测试，也难以在不破坏其他部分的情况下修改代码。因此，我们最好将函数拆分开来，让一个函数只负责一项任务。

这同时也关系到第二个问题：虽然query和filename变量是用来存储程序配置的，但与它们同为变量的contents等却是用于业务逻辑的。随着main中的代码越来越长，我们需要引入的变量势必越来越多。而当作用域中的变量越来越多时，我们就越难以追踪每个变量的实际含义。因此，我们最好将多个配置变量合并至一个结构体内，从而让它们的用途变得更加清晰。

第三个问题是，我们在处理文件读取失败时选择了使用expect输出错误提示信息，但它只语焉不详地打印出了Something went wrong reading the file。读取文件的操作会因为许多不同的原因而失败，例如文件不存在或缺少相关权限等。但就目前而言，无论发生了什么情况，我们都只能打印出Something went wrong reading the file这条错误提示信息，它并没有办法给用户提供任何有用的排错信息！

第四，我们广泛地使用expect来处理不同的错误，当用户运行程序却没有指定参数时，他们只会得到来自Rust语言内部的错误提示信息：index out of bounds，却无法清晰地理解问题本身。我们最好将用于错误处理的代码集中放置，从而使将来的维护者在需要修改错误处理相关逻辑时只用考虑这一处代码。另外，将它们放置到一处也能

确保我们为终端用户打印出的错误提示信息是有意义的、便于理解的。

让我们针对这4个问题开始重构项目。

二进制项目关注点分离

很多二进制项目都会面临同样的组织结构问题：它们将过多的功能、过多的任务放到了main函数中。对此，Rust社区开发了一套为将会逐渐臃肿的二进制程序进行关注点分离的指导性原则：

- 将程序拆分为*main.rs* 和 *lib.rs*，并将实际的业务逻辑放入 *lib.rs*。
- 当命令行解析逻辑相对简单时，将它留在*main.rs* 中也无妨。
- 当命令行解析逻辑开始变得复杂时，同样需要将它从*main.rs* 提取至 *lib.rs* 中。

经过这样的拆分之后，保留在main函数中的功能应当只有：

- 调用命令行解析的代码处理参数值。
- 准备所有其他的配置。
- 调用 *lib.rs* 中的run函数。
- 处理run函数可能出现的错误。

这种模式正是关注点分离思想的体现：*main.rs* 负责运行程序，而 *lib.rs* 则负责处理所有真正的业务逻辑。虽然你无法直接测试main函数，但因为我们将大部分代码都移动到了*lib.rs* 中，所以我们依然可以测试几乎所有的程序逻辑。依然保留在*main.rs* 中的代码量应该小到可以直接通过阅读来进行正确性检查。下面让我们按照以上原则来重构程序。

提取解析参数的代码

首先，我们需要把解析参数的功能提取成单独的函数以便main函数调用，并为随后将它转移至 *src/lib.rs* 做好准备。示例12-5中展示了新的main函数的开头部分，它调用了一个暂时定义在 *src/main.rs* 文件中的新函数parse_config。

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, filename) = parse_config(&args);

    // --略

    --
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

示例12-15：将main函数中的部分代码提取成parse_config函数

这段代码依然将所有的命令行参数收集到了一个动态数组中，但不同于在main函数中将索引为1的参数赋值给query、将索引为2的参数赋值给filename，这里直接将整个动态数组都传递给了parse_config函数。接着，再由parse_config函数中的逻辑来决定将哪个参数赋值给哪个变量，并将结果传给main函数。虽然我们还是在main函数中定义了query和filename变量，但main函数已经不需要再关心变量和命令行参数之间的关系了。

这样的重写步骤对于我们的小程序来说也许会有些大材小用，但重构工作正是要这样小步、递进地完成。在修改之后，请记得重新运行程序并确认参数解析的功能仍然能够正常工作。经常验证你的工作进展是一个好习惯，它可以帮助你在发生问题时迅速定位到具体原因。

组合配置值

我们还可以再稍微改进一下parse_config函数。目前的函数返回了一个元组，但我们在使用时又立即将元组拆分为了独立的变量。这

种迹象说明当前程序中建立的抽象结构也许是不对的。

另外值得注意的是parse_config名称中的config部分，它暗示我们返回的两个值是彼此相关的，并且都是配置值的一部分。单纯地将这两个值放置在元组中并不足以表达出这些意义。我们可以将这两个值放在一个结构体中，并给予每个字段一个有意义的名字。这样可以让未来的维护者更加方便地理解不同值之间的关系及其各自的用处。

注意

在使用复杂类型更合适时偏偏坚持使用基本类型，是一种叫作基本类型偏执（primitive obsession）的反模式（anti-pattern）。

示例12-6展示了改进后的parse_config函数。

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

❶    let config = parse_config(&args);

    println!("Searching for {}", config.query❷);
    println!("In file {}", config.filename❸);

    let contents = fs::read_to_string(config.filename❹)
        .expect("Something went wrong reading the file");

    // --略

    --
}

❺ struct Config {
    query: String,
    filename: String,
}

❻ fn parse_config(args: &[String]) -> Config {
    ❼    let query = args[1].clone();
    ❽    let filename = args[2].clone();

    Config { query, filename }
}
```

示例12-6：重构parse_config函数以返回一个Config结构体的实例

这段代码新增了一个包含 query 和 filename 字段的结构体 Config❸。函数 parse_config 的签名意味着它现在会返回一个 Config 类型的值❹。在 parse_config 的函数体内，我们之前返回的是指向 args 中 String 值的字符串切片，但我们现在定义的 Config 却包含了拥有自身所有权的 String 值。这是因为 main 函数中的 args 变量是程序参数值的所有者，而 parse_config 函数只是借用了这个值。如果 Config 试图在运行过程中夺取 args 中某个值的所有权，那么就会违反 Rust 的借用规则。

有许多不同的方法可以用来处理 String 类型的输入值，但其中最简单的莫过于调用 clone 方法进行复制，尽管它可能会有些低效❻❼。这个方法会将输入值完整复制一份，从而方便 Config 实例取得新值的所有权。这样做确实比存储字符串的引用消耗了更多的时间和内存，但同时也省去了管理引用的生命周期的麻烦，从而让代码更加简单直接。在这个场景中，用少许的性能交换更多的简捷性是非常值得的取舍。

使用 clone 的取舍

许多 Rust 爱好者由于担心增加运行时代价，从而会避免使用 clone 来解决所有权问题。我们确实会在第 13 章学习如何更有效率地处理这种情形。但是对于本例来说，用复制字符串的方式来改进程序是没有任何问题的，因为我们只会执行一次相关的复制操作，并且文件名和搜索字符串都只会占用相当小的空间。在首次编写程序时，先完成一个运行正常但效率有待改进的程序比尝试过度优化代码更好一些。另外，随着你越来越熟悉 Rust，你也会越来越容易地一次性写出高效率的代码，但是，目前使用 clone 是完全可以接受的。

我们更新了 main 函数，它会将 parse_config 返回的 Config 实例放入 config 变量中❶，并将我们之前独立使用 query 和 filename 的地方相应地修改为了使用 Config 结构体中的字段❷❸❹。

更新后的代码清晰地表明了我们的意图：query和filename是相关联的，它们被用于控制程序的运作方式。那些用到这些值的代码现在知道该去config实例中寻找对应名称的字段了。

为Config创建一个构造器

到目前为止，我们先是将解析命令行参数的逻辑从main函数中分离出来并放于parse_config函数中。这一过程帮我们厘清了query与filename之间的关系，并将这种关系在代码中体现出来。接着，我们增加了一个名为Config的结构体来描述query和filename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的字段名返回。

实际上，parse_config函数的功能正是创建一个新的Config实例，所以我们可以用一种更符合Rust惯例的方式，把parse_config从一个普通函数改写成一个与Config结构体相关联的new函数。对于标准库中像String这样的类型，我们可以通过调用String::new来创建新的实例。同样，如果将parse_config改写成Config的新函数，我们也能通过调用Config::new来创建新的Config实例。示例12-7展示了需要对代码做出的修改。

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

❶ let config = Config::new(&args);
    // --略

    --
}

// --略

--

❷ impl Config {
❸     fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

示例12-7：将parse_config函数改写为Config::new

这段代码将main函数中调用parse_config的地方改为了调用Config::new❶，而parse_config函数的名字则被改写为了new❷，并被关联到了Config的impl块中❸。现在可以尝试编译运行这段代码以确保它可以正常运行了。

修复错误处理逻辑

现在，我们开始修复错误处理的相关逻辑。之前，我们曾经尝试在动态数组args的元素不足3个时使用索引1或索引2来访问其中的值，从而导致代码产生了panic。你可以再试一下不带任何参数来运行这段程序，运行结果会如下所示：

```
$ cargo run

Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', src/main.rs:25:21
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

其中，index out of bounds: the len is 1 but the index is 1这一行是用于提醒程序员的错误提示信息，它不会帮助终端用户了解发生了什么或接下来应该怎么做。现在，让我们来修复一下这条逻辑。

改进错误提示信息

在示例12-8中，我们在new函数中添加了一段用于确认切片长度是否充足的语句，以便在访问索引为1或索引为2的数据之前进行检查。如果切片长度不足，则程序就会引发panic并显示出一条比index out of bounds更为有用错误提示信息。

src/main.rs

```
// --略
```

```
-
```

```
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
    // --略
--
```

示例12-8：增加对参数数量的检查

这段代码和示例9-10中编写的Guess::new函数有些相似，当时我们在value参数超出有效值范围时调用了panic!。不过在本例中，我们检查的不再是数值的范围而是args的长度是否达到了3，从而使函数的剩余部分可以在满足该条件的基础上继续运行。假设args中的元素数量不足3，那么条件为真，我们会调用panic!立刻终止程序。

在new函数中添加完上面的错误处理代码后，再次在不输入任何参数的情况下运行程序并观察会出现怎样的错误：

```
$ cargo run
```

```
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这次的输出就好多了，我们得到了一段合理的错误提示信息。但是，输出中仍然残留了一些我们不希望暴露给用户的信息。实际上，示例9-10中使用过的这种方法已经不再适用于当前的场景了：正如在第9章讨论的那样，我们更倾向于使用panic!来暴露程序的内部问题而非用法问题。因此我们改为使用在第9章学过的另一种方法：返回一个可以表明结果成功或失败的Result。

从new中返回Result而不是调用panic!

我们可以返回一个Result值，它会在成功的情况下包含Config实例，并在失败的情况下携带具体的问题描述。当我们在main函数中调用Config::new时，就可以使用Result类型来表明当前是否存在问题。接着，我们还可以在main函数中将可能出现的Err变体转换为一种更加友好的形式来通知用户。使用这种方法可以避免调用panic!时在错误提示信息前后产生thread 'main' 和RUST_BACKTRACE等内部信息。

示例12-9中的代码将Config::new的返回值修改为了Result。注意，因为我们还没有对main函数做出对应的修改，所以这段代码暂时无法通过编译。

src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

示例12-9：让Config::new返回一个Result

现在的new函数会返回Result，它在运行成功时带有一个Config实例，而在运行失败时带有一个&' static str。我们在第10章的“静态生命周期”一节中曾经指出&' static str是字符串字面量的类型，这也正是我们目前使用的错误提示信息类型。

我们还在new函数体中做出了两处改动：一是当用户输入参数不足时，返回Err值而不是调用panic!；二是将Config返回值放于Ok变体中。这样就让函数的实现符合了新修改的函数签名。

Config::new在运行失败时返回的Err值使main函数可以对Result值做进一步处理，以便它能够在出错时更加干净地退出进程。

调用Config::new并处理错误

为了处理错误情形并打印出对用户友好的信息，我们需要修改main函数来处理Config::new返回的Result值，如示例12-10所示。另外，我们还需要取代之前由panic!实现的退出命令行工具并返回一个非0的错误码的功能。程序在退出时向调用者（父进程）返回非0的状态码是一种惯用的信号，它表明当前程序的退出是由某种错误状态导致的。

src/main.rs

```
❶use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

❷    let config = Config::new(&args).unwrap_or_else(|❸| err❹| {
        ❺    println!("Problem parsing arguments: {}", err);
        ❻    process::exit(1);
    });
}

// --略

--
```

示例12-10：在创建Config实例失败时使用错误码退出程序

在这段代码中，我们使用了一个尚未接触过的`unwrap_or_else`方法，它被定义于标准库的`Result<T, E>`中❷。使用`unwrap_or_else`可以让我们执行一些自定义的且不会产生`panic!`的错误处理策略。当`Result`的值是`Ok`时，这个方法的行为与`unwrap`相同：它会返回`Ok`中的值。但是，当值为`Err`时，这个方法则会调用闭包（closure）中编写的代码，也就是我们定义出来并通过参数传入`unwrap_or_else`的这个匿名函数❸。我们将在第13章学习有关闭包的更多知识。目前，你只需要知道闭包的参数被写在两条竖线之间，而`unwrap_or_else`则会将`Err`中的值，也就是示例12-9中添加的`not enough arguments`，作为参数`err`传递给闭包❹。闭包中的代码可以在随后运行时使用参数`err`中的值。

新增的那一行`use`语句被用来将标准库中的`process`引入作用域中❺。只会在错误情形下调用的闭包代码仅有两行：打印`err`的值❻并接着调用`process::exit`函数❼。调用`process::exit`函数会立刻中止程序运行，并将我们指定的错误码返回给调用者。这类似于示例12-8中基于`panic!`的处理流程，但此时的错误提示信息中再也不会出现之前的一些额外信息了。让我们试试看：

```
$ cargo run

Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

非常棒！现在的错误提示信息对用户友好多了。

从main中分离逻辑

现在，我们已经完成了对配置解析的重构工作，接下来就轮到程序的逻辑了。如同在“二进制项目的关注点分离”一节中讨论的那样，我们会把main函数中除配置解析和错误处理之外的所有逻辑都提取到单独的run函数中。一旦完成这项工作，main函数本身就会精简得足以通过阅读来检查正确性，而其他几乎所有的逻辑则能够通过测试代码进行检验。

分离出来的run函数如示例12-11所示。目前只是做了一些较小的、增量式的提取改进，所以我们仍然要在*src/main.rs*中定义这个函数。

src/main.rs

```
fn main() {
    // --略

    --
    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.filename)
        .expect("something went wrong reading the file");

    println!("With text:\n{}", contents);
}
// --略

--
```

示例12-11：将其他所有的逻辑分离为run函数

这个run函数包含了main函数中从读取文件处开始的所有逻辑，它会接收一个Config实例作为参数。

从run函数中返回错误

通过将程序逻辑全部提取到run函数中，我们现在可以像示例12-9中的Config::new函数那样来改进错误处理了。run函数应当在发生错误时返回Result<T, E>，而不是调用expect引发panic。这让我们可以进一步在main函数中统一处理错误情形，从而给用户一个友好的反馈。示例12-12展示了对run函数签名和函数体做出的修改。

src/main.rs

```
❶use std::error::Error;
// --略
--❷fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?❸;
    println!("With text:\n{}", contents);
❹    Ok(())
}
```

示例12-12：修改run函数使其返回Result

这段代码中主要有3处改动。首先，我们将run函数的返回值修改为了Result<(), Box<dyn Error>>❷。之前，函数的返回值是空元组()，它被保留在Ok情形中作为返回值使用。

而对于错误情形，我们则使用了trait对象（trait object）Box<dyn Error>（我们已经通过use语句将std::error::Error引入了作用域❶）。第17章将对trait对象进行详细的讨论。现在，你只需要知道Box<dyn Error>意味着函数会返回一个实现了Error trait的类型，但我们并不需要指定具体的类型是什么。这意味着我们可以在不同的错误场景下返回不同的错误类型，语句中的dyn关键字所表达的正是这种“动态”（dynamic）的含义。

其次，我们用在第9章讨论过的?运算符取代了expect❸。不同于panic!宏对错误的处理方式，?运算符可以将错误值返回给函数的调用者来进行处理。

最后，修改后的run函数会在运行成功时返回Ok④。由于函数签名中指定了运行成功时的数据类型是()，所以我们需要把空元组的值包裹在Ok变体中。初看Ok()的写法可能会有些奇怪，但这样使用()其实可以更清楚地表明函数的编写意图：调用run函数只是为了产生函数副作用，而不是为了返回任何有用的值。

假如我们现在运行这段代码，你会发现它虽然能够成功通过编译，但是却输出了一条警告消息：

```
warning: unused `std::result::Result` that must be used
--> src/main.rs:17:5
  |
17 |     run(config);
  | ^^^^^^^^^^^^^^
  |
= note: #[warn(unused_must_use)] on by default
= note: this `Result` may be an `Err` variant, which should be handled
```

Rust告诉我们代码中忽略了对Result值的处理。一个函数返回Result值，表明它在运行时可能发生了错误。但是，我们没有检查错误是否发生，于是编译器通过警告提醒我们需要在该处添加错误处理代码！现在就让我们来修复这个问题。

在main中处理run函数返回的错误

我们将检查并处理错误，此处用到的技术有些类似于示例12-10中处理Config::new返回值的方法，当然也会有少许差异：

src/main.rs

```
fn main() {
    // --略
    --
    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

我们使用了if let而不是unwrap_or_else来检查run的返回值，并在返回Err值的情况下调用了process::exit(1)。和Config::new返回一个Config实例不同，run函数并不会返回一个需要进行unwrap的值。因为run函数在运行成功时返回的是()，而我们只关注产生错误时的情形，所以没有必要调用unwrap_or_else把这个必定是()的值取出来。

不过在这两个例子中，if let和unwrap_or_else的函数体是一样的：打印错误并退出程序。

将代码分离为独立的代码包

现在，我们的minigrep项目看起来好多了！接下来，我们需要拆分*src/main.rs*文件并将部分代码移入*src/lib.rs*，这使我们可以正常进行测试并减少*src/main.rs*中负责的功能。

让我们将所有非main函数的代码从*src/main.rs*转移至*src/libs.rs*，它们包括：

- run函数的定义
- 相关的use语句
- Config的定义
- Config::new函数的定义

转移完毕后，文件*src/lib.rs*中应该包含示例12-13所示的各种签名（为了使代码看起来比较简捷，这里省略了函数体）。注意，直到在示例12-14中修改完*src/main.rs*后，整个项目才能编译通过。

src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub filename: String,
}
```

```

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        // --略

        --
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --略

    --
}

```

示例12-13：将Config和run转移至 *src/lib.rs* 中

在新的代码中，我们广泛地在Config结构体、结构体内各个字段、new方法及run函数上使用了pub关键字。现在，我们拥有一个可以进行测试的公共API代码包了！

我们还需要将那些转移至 *src/libs.rs* 的代码导入二进制包的 *src/main.rs* 的作用域中，如示例12-14所示。

src/main.rs

```

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --略

    --
    if let Err(e) = minigrep::run(config) {
        // --略
    }
}

```

示例12-14：将minigrep包引入 *src/main.rs* 的作用域中

为了将代码包中的Config类型引入二进制包的作用域中，我们增加了use minigrep::Config这行语句。另外，我们还将包的名称作为前缀添加到了run函数前。现在，所有的功能组件应该都可以连接到一块并顺利运行了。让我们使用cargo run来运行程序并确保一切正常。

这可真是一个大工程！但这是值得的，因为它为我们将来的成功打下了基础。现在的代码要更加模块化一些，也更容易对错误情形做出响应。剩下的几乎所有的工作都可以只在 `src/lib.rs` 中进行了。

接下来，让我们利用模块化的便利来完成一个曾经很难做到，但现在却轻而易举的任务：编写测试！

使用测试驱动开发来编写库功能

现在，我们已经把主要的逻辑提取到了 `src/lib.rs` 中，并将参数解析和错误处理留在了 `src/main.rs` 中。这为编写测试去验证程序的核心功能提供了很大的便利。我们可以直接使用不同的参数来调用功能函数并检验其返回值，而不需要在命令行下运行二进制程序。你现在就可以自行行为 `Config::new` 和 `run` 函数中的功能编写几个测试。

在本节中，我们会按照测试驱动开发（test-driven development, TDD）的流程来为 `minigrep` 程序添加搜索逻辑。这一软件开发技术需要遵循如下步骤：

1. 编写一个会失败的测试，运行该测试，确保它会如期运行失败。
2. 编写或修改刚好足够多的代码来让新测试通过。
3. 在保证测试始终通过的前提下重构刚刚编写的代码。
4. 返回步骤1，进行下一轮开发。

虽然测试驱动开发只是众多软件开发技术中的一个，但它能对代码的设计工作起到指导和帮助的作用。优先编写测试，然后再编写能够通过测试的代码也有助于在开发过程中保持较高的测试覆盖率。

我们将通过测试驱动来实现具体的搜索功能，它会在文件内容中搜索指定字符串，并生成一个包含所有匹配行的列表。这些代码会被放置在一个名为 `search` 的函数中。

编写一个会失败的测试

让我们移除 *src/libs.rs* 和 *src/main.rs* 中的那些用来检查程序行为的 `println!` 语句，因为新的程序不再需要它们了。接着，我们会像在11章做过的那样在 *src/lib.rs* 中添加一个附带测试函数的 `tests` 模块。这个测试函数指定了我们期望 `search` 函数所拥有的行为：它会接收一个查询字符串和一段用于查询的文本，并返回文本中包含查询字符串的所有行。示例12-15展示了这个暂时还无法通过编译的测试：

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：基于我们对 `search` 函数行为的预期，创建一个暂时会失败的测试

这个测试要求搜索字符串“`duct`”。因为在被搜索的3行文本中只有第二行包含“`duct`”，所以我们断言 `search` 函数的返回值只会包含这一行。

我们现在无法运行并观察到测试失败的结果，因为它调用的 `search` 函数还没有被编写，此时的程序甚至连编译都无法通过！为了使测试能够正常地编译和运行，我们会添加一个返回空动态数组的 `search` 函数定义，如示例12-16所示。这一修改恰好使测试可以编译运行。因为新函数的返回值是一个空动态数组，它并不会包含我们期待的“`safe, fast, productive.`”行，所以示例中的测试暂时会运行失败。

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

示例12-16： 定义一个恰好能让测试编译通过的search函数

注意，search函数的签名中需要一个显式生命周期'a，它被用来和contents参数与返回值一起使用。我们在第10章曾经说过，生命周期参数指定了哪一个参数的生命周期会和返回值的生命周期产生关联。在本例中，我们指定返回的动态数组应当包含从contents参数（而不是query参数）中取得的字符串切片。

也就是说，我们告诉Rust，search函数返回的数据将与contents参数中的数据有同样的生命周期。这一点非常重要！只有当切片引用的数据有效时，引用本身才是有效的。如果编译器误认为我们在获取query的字符串切片而不是contents的字符串切片，那么就无法进行正确的安全检查。

假如我们忘记标记生命周期并直接尝试编译这个函数，那么就会产生如下所示的错误：

```
error[E0106]: missing lifetime specifier
--> src/lib.rs:5:51
  |
5 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
  |                                     ^ expected 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`
```

Rust不可能得知返回值究竟需要哪一个参数，我们必须明确告知编译器这个信息。因为contents参数中包含了所有待查找及返回的文本内容，所以我们知道contents正是可以通过生命周期语法与返回值相关联的那个参数。

其他编程语言并不需要你在签名中关联参数与返回值。尽管这一设计初看上去会有些奇怪，但通过不断地练习，相信你一定会逐渐习惯它。你可以将这个例子与第10章的“使用生命周期保证引用的有效性”一节相互参照着学习。

现在让我们运行这个测试：

```
$ cargo test

Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
Running target/debug/deps/minigrep-abcabcabc

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
    thread 'tests::one_result' panicked at 'assertion failed: `left == right)`'
left: `["safe, fast, productive."]`,
right: `[]`, src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
error: test failed, to rerun pass '--lib'
```

很好！测试正如期待的那样运行失败了。让我们来修复这个测试吧！

编写可以通过测试的代码

目前的测试之所以会失败是因为我们总是返回一个空动态数组。我们需要按照以下的步骤来修复并真正实现search函数：

1. 遍历内容的每一行。
2. 检查当前行是否包含搜索字符串。
3. 如果包含，则将其添加到返回值列表中。
4. 如果不包含，则忽略。
5. 返回匹配到的结果列表。

我们会从遍历开始，依次编写上面每一步的代码。

使用lines方法逐行遍历文本

Rust有一个可以逐行遍历字符串的方法，被命名为lines，其用法如示例12-17所示。请注意，这段代码暂时还不能通过编译。

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：逐行遍历contents中的内容

lines方法会返回一个迭代器。我们会在第13章深入地讨论迭代器，但回忆一下，我们已经在示例3-5中见识过类似的迭代器使用方法了，那时我们配合迭代器与for循环遍历了集合中的每一个元素。

在每一行中搜索字符串

接下来，我们会检查当前行是否包含待搜索的字符串。幸运的是，字符串类型有一个名为contains的实用方法可以帮助我们完成这项工作！在search函数中加入调用contains方法的代码，如示例12-18所示。请注意，此时的代码依然无法通过编译。

src/lib.rs

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // 使用line执行某些操作
        }
    }
}
```

示例12-18：添加判断当前行是否包含query参数指定的字符串的功能

存储匹配的行

最后，我们需要将包含目标字符串的行存储起来。为此，我们可以在for循环之前创建一个可变的动态数组，并在循环过程中用push方法将line变量存入其中。在for循环结束之后，我们就直接返回这个动态数组，如示例12-19所示。

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：存储匹配的行并返回

现在，我们的测试应该可以通过了，search函数的返回值中包含了所有与query相匹配的行。运行测试：

```
$ cargo test

--略

-- 
running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

测试顺利通过，也就是说我们的程序可以正常运行！

此时，我们可以考虑一下search函数的实现是否还存在有待重构的余地。在重构的过程中只要始终确保测试通过，就可以保证功能不会受到影响。search函数中的代码看上去还不算太坏，但它还没有用到迭代器的一些实用功能。我们会在第13章更深入地讨论迭代器时再来看这段示例，并研究一下如何改进这段代码。

在run函数中调用search函数

search函数经过测试可以正常运行了，现在让我们来调用它。我们需要向search函数中传入config.query的值，以及run函数从文件中读取的contents文本。接着，run函数还会打印出search函数返回的每一行内容：

src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;
    for line in search(&config.query, &contents) {
        println!("{}", line);
    }
    Ok(())
}
```

这段代码再次使用了for循环去获取并打印search返回值中的每一行。

我们终于编写完了所有代码！现在你可以使用“frog”来试着运行程序，这个单词只会匹配到Emily Dickinson的诗中的一行。

```
$ cargo run frog poem.txt
```

```
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

棒！接下来，搜索一个反复出现在很多行中的单词“body”：

```
$ cargo run body poem.txt
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

最后，搜索一个诗中没有出现过单词“monomorphization”：

```
$ cargo run monomorphization poem.txt
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep monomorphization poem.txt`
```

非常好！我们已经实现了迷你版的经典命令行工具，同时也掌握了如何搭建一个应用程序的方法。另外，我们还了解了一些有关文件输入输出、生命周期、测试和命令行参数解析的知识。

为了进一步完善这个项目，我们接下来还会简要地演示如何处理环境变量，以及如何将信息输出到标准错误流。这两个技巧都是在编写命令行工具时经常用到的。

处理环境变量

我们将增加一项额外的功能来继续完善minigrep：用户可以通过设置环境变量来进行不区分大小写的搜索。我们当然可以将这个选项做成命令行参数，并要求用户在每次运行minigrep时手动添加这一参数。但出于教学的目的，我们在本节选择使用环境变量来实现这项功能。另外，这样也可以允许用户只配置一次环境变量，就能让配置的选项在整个终端会话中一直有效。

为不区分大小写的search函数编写一个会失败的测试

为了应对设置环境变量后的情形，我们计划增加一个新的search_case_insensitive函数。为了继续遵循测试驱动开发的流程，我们首先需要为search_case_insensitive函数编写一个暂时会失败的测试用例。示例12-20为search_case_insensitive函数添加了一个新的测试，并将旧测试的名称从one_result改为case_sensitive以凸显两个测试之间的区别。

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：为我们计划添加的不区分大小写的函数编写一个暂时失败的测试

注意，我们同时修改了旧测试中contents的值，它新增了一行包含大写D的文本“Duct tape.”，该行文本无法在区分大小写的模式下匹配到搜索值duct。这样修改测试用例可以帮助我们确保已经实现的区分大小写的搜索功能不会遭到意外损坏。在我们编写新功能的过程中，这个测试应当是一直保持通过的。

为不区分大小写的搜索编写的新测试使用了rUsT作为搜索字符串。在我们即将添加的search_case_insensitive函数中，这一搜索字符串应该会匹配带有大写R的“Rust:”以及“Trust me.”两行，即便它们都拥有与搜索字符串不一样的大小写字母。这个测试暂时还不能通过编译，因为我们尚未定义search_case_insensitive函数。你可以像在示例12-16中编写search函数那样，定义一个返回空动态数组的假实现，并观察测试编译和运行失败的过程。

实现search_case_insensitive函数

search_case_insensitive函数的实现和之前的search函数几乎一致，如示例12-21所示。唯一的区别在于我们将query和每一行的line都转换成了小写，这样一来，无论输入的参数是大写还是小写，当我

们在检查某行文本中是否包含目标字符串时，它们都会拥有相同的大写模式。

src/lib.rs

```
pub fn search_case_insensitive<'a>(query: &str, contents: &'a str)
-> Vec<&'a str> {
❶ let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase()❷.contains(&query❸) {
            results.push(line);
        }
    }

    results
}
```

示例12-21：在比较搜索字符串和文本前，将它们转换为小写，以实现search_case_insensitive函数

首先，我们将query字符串转换为小写，并把结果存储到同名变量中❶。在将字符串转换为小写后，无论用户搜索的是rust、RUST、Rust还是rUsT，我们都可以将它们统一视作rust来处理，而不区分子字符串中的字母是大写还是小写。

需要注意，现在的query是一个拥有数据所有权的String，而不再是一个字符串切片。因为调用to_lowercase函数必定会创建新的数据，而不可能去引用现有数据。以测试用的rUsT为例，现有的字符串切片中并没有小写的u和t可以使用，所以我们必须分配新的String才能存储rust这个结果。当我们将新的query作为参数传递给contains时必须添加一个&符号❸，因为函数contains的签名只会接收一个字符串切片作为参数。

接着，在每次检查行文本是否包含query前，我们同样使用to_lowercase将line转换为小写字符串❷。由于line和query都被转换为了小写，所以随后进行的匹配操作就不再区分大小写了。

让我们看一下这个实现是否能够通过测试：

```
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
```

很好，测试通过了！现在，我们需要在run函数中调用新的search_case_insensitive函数。首先，我们将为Config结构体增加一个新的配置选项，以切换区分大小写的搜索和不区分大小写的搜索。只是简单地将这个字段添加到代码中会引起编译错误，因为这个字段尚未在任何地方被初始化：

src/lib.rs

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

注意，我们增加的这个字段case_sensitive是一个布尔类型。接下来，我们要在run函数中根据这个字段的值来决定调用search函数还是search_case_insensitive函数，如示例12-22所示。请注意，这段代码目前还无法编译。

src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

示例12-22：根据config.case_sensitive的值决定调用search函数还是search_case_insensitive函数

最后，我们还需要检查当前设置的环境变量。因为用于处理环境变量的相关函数被放置在标准库的env模块中，所以我们需要在src/libs.rs的起始处添加use std::env;语句来将该模块引入当前作

用域。接着，我们会使用 env 模块中的 var 函数来检查名为 CASE_INSENSITIVE 的环境变量是否存在，如示例12-23所示。

src/lib.rs

```
use std::env;

// --略

--



impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
        Ok(Config { query, filename, case_sensitive })
    }
}
```

示例12-23：检查环境变量CASE_INSENSITIVE

这段代码创建了一个新的变量case_sensitive。为了给它赋值，我们调用了env::var函数，并将环境变量CASE_INSENSITIVE的名称作为参数传递给该函数。env::var函数会返回一个Result作为结果，只有在环境变量被设置时，该结果才会是包含环境变量值的Ok变体，而在环境变量未被设置时，该结果则会是一个Err变体。

我们使用了Result的is_err方法来检查结果是否为错误，也就是环境变量是否未被设置，应当进行区分大小写搜索的情况。如果CASE_INSENSITIVE环境变量被设置为了某个值，那么is_err就会返回假，也就意味着程序会进行不区分大小写的搜索。因为我们不关心环境变量的具体值，只关心其存在与否，所以我们直接使用了is_err而不是unwrap、expect或其他曾经接触过的Result的方法。

我们随后将case_sensitive变量的值传递给了Config实例，从而使run函数可以读取这个值，以此决定是否调用search或示例12-22中实现的search_case_insensitive。

让我们试一试吧！首先，不设置环境变量并使用to作为查询字符串运行程序，我们会看到程序找出了所有带有小写to的行：

```
$ cargo run to poem.txt
```

```
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

看起来程序依旧能够正常工作！现在，将CASE_INSENSITIVE设置为1并继续运行程序搜索字符串to。

如果你现在正在使用PowerShell，那么你就需要使用两条命令来分别设置环境变量和执行程序：

```
$ $env:CASE_INSENSITIVE=1
```

```
$ cargo run to poem.txt
```

即便是文本中包含了大写字母的to，我们也应该能够将它们全部搜索出来：

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

非常好！我们匹配到了包含To的文本行，新的minigrep程序可以在环境变量的控制下进行不区分大小写的搜索。现在，你就知道如何通过命令行参数或环境变量来控制程序选项了。

某些程序允许用户同时使用命令行参数和环境变量来设置同一个选项。在这种情况下，程序需要确定不同配置方式的启用优先级。作为练习，你可以同时使用命令行参数和环境变量来配置不区分大小写的选项，并在两种配置方式不一致时决定命令行参数和环境变量的优先级。

`std::env`模块中还有很多用于处理环境变量的实用功能，你可以查看它的文档来了解这些可用的功能。

将错误提示信息打印到标准错误而不是标准输出

目前，我们将所有的输出信息通过`println!` 宏打印到了终端上。大多数的终端都提供两种输出：用于输出一般信息的标准输出（`stdout`），以及用于输出错误提示信息的标准错误（`stderr`）。这种区分可以使用户将正常输出重定向到文件的同时仍然将错误提示信息打印到屏幕上。

`println!` 宏只能用来打印到标准输出，我们需要使用其他工具才能将信息打印到标准错误中。

确认错误被写到了哪里

首先，让我们观察一下`minigrep`输出的那些信息是如何被打印至标准输出的，包括那些应该被写入标准错误的错误提示信息。我们可以将标准输出重定向到一个文件，并故意触发错误来观察这一现象。由于我们没有重定向标准错误，所以打印到标准错误上的那些内容仍然会输出到屏幕上。

命令行程序本应该将错误提示信息输出到标准错误，这样我们就能够在将标准输出重定向到文件的同时，仍然在屏幕上看到错误提示信息。由于目前我们的程序行为还不够标准，所以我们将会看到错误提示信息也输出并保存到了文件中。

为了演示这一行为，我们可以在运行程序时使用`>`运算符与文件名 `output.txt`，这个文件名指定了标准输出重定向的目标。由于我们没有传入任何参数，所以程序应该会在执行时引发一个错误：

```
$ cargo run > output.txt
```

这里的>语法会告知终端将标准输出中的内容写入*output.txt*文件中而不是打印到屏幕上。运行程序后，屏幕上没有出现我们期待的错误提示信息，这意味着错误提示信息可能被写入了文件中。现在，*output.txt*文件中应该会包含以下内容：

```
Problem parsing arguments: not enough arguments
```

没错，我们的错误提示信息被打印到了标准输出中。将类似的错误提示信息打印到标准错误可能会更加实用，这样可以让文件内容保持整洁，只包含正常的运行结果数据。接下来我们就要改变这种行为。

将错误提示信息打印到标准错误

我们将使用示例12-24中的代码来演示如何修改错误提示信息的输出方式，它使用了一个由标准库提供的eprintln! 宏来向标准错误打印信息。在本章前面的重构中，我们已经把所有打印错误提示信息的代码都放到了main函数中，因此我们只需要将打印错误提示信息的两处println! 改为eprintln! 即可。

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&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! 将错误提示信息打印到标准错误而不是标准输出

将println! 修改为eprintln! 之后，让我们再次以同样的方式运行这段程序：

```
$ cargo run > output.txt  
  
Problem parsing arguments: not enough arguments
```

现在，我们可以看到错误提示信息被打印到了屏幕上，而 *output.txt* 中则没有任何内容，这才是一个符合我们期望的命令行程序的行为。

接下来，让我们使用正常的参数运行程序，依然将标准输出重定向到文件：

```
$ cargo run to poem.txt > output.txt
```

我们可以看到，终端上没有打印出任何信息，而 *output.txt* 中则包含了正确的输出结果：

```
output.txt  
Are you nobody, too?  
How dreary to be somebody!
```

这就意味着我们合理地使用了标准输出和标准错误来区分正常结果和错误提示信息。

总结

在本章中，我们回顾了曾经学习过的一部分主要概念，并掌握了如何在Rust环境中进行常用的I/O操作。通过使用命令行参数、文件、环境变量及打印错误的`eprintln!`宏，你现在已经为编写命令行程序做好了一切准备。通过结合前几章的相关知识，你将能够有序地组织代码、有效率地运用数据结构存储数据、优雅地处理错误并保证程序会经过充分的测试。

接下来，我们将要开始讨论Rust中那些受到函数式编程语言影响的功能：闭包和迭代器。

第13章

函数式语言特性：迭代器与闭包



Rust在设计过程中从许多现有的语言和技术中获得启发，函数式编程（functional programming）理念就是其中之一，它对Rust产生了非常显著的影响。常见的函数式风格编程通常包括将函数当作参数、将函数作为其他函数的返回值或将函数赋给变量以备之后执行等。

在本章中，我们不会去争论究竟什么才是函数式编程，而会将讨论的重点放在Rust与其他函数式语言相似的特性上。

具体来说，本章会涉及以下几方面内容：

- 闭包（closure），一个类似于函数且可以存储在变量中的结构。
- 迭代器（iterator），一种处理一系列元素的方法。
- 使用闭包和迭代器来改善第12章中的I/O项目。

- 讨论闭包和迭代器的运行时性能。（悄悄透露一下：它们比你想象的还要快！）

其他一些Rust特性其实也同样深受函数式风格的影响，例如我们在其他章节中提到过的模式匹配和枚举。掌握闭包和迭代器对于编写风格地道、运行迅速的Rust程序相当重要，所以我们专门投入了这一整章的内容来详细讲解它们。

闭包：能够捕获环境的匿名函数

Rust中的闭包是一种可以存入变量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环境中调用该闭包来完成运算。和一般的函数不同，闭包可以从定义它的作用域中捕获值。我们将展示如何运用闭包的这些特性来实现代码复用和行为自定义。

使用闭包来创建抽象化的程序行为

闭包在接下来的示例场景中十分有用，我们会合理地存储闭包并在后面执行它。我们还会在完善这个示例的过程中逐步讨论闭包的语法、类型推断及一些相关的trait。

假设有这样一个场景：我们身处的初创公司正在开发一个为用户提供健身计划的应用。使用Rust编写的后端程序在生成计划的过程中需要考虑到年龄、身体质量指数（BMI）、健身偏好、运动历史、指定强度值等因素。具体的算法究竟长什么样子在这个例子中并不重要，重要的是这个计算过程会消耗掉数秒钟时间。我们希望只在必要的时候调用算法，并且只调用一次，以免让用户等待过久。

我们会在函数`simulated_expensive_calculation`中模拟这一假设的算法的计算过程，如示例13-1所示，它会依次打印出`calculating slowly...`，等待两秒钟，并接着返回传递给它的数字。

src/main.rs

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: u32) -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

```
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

示例13-1：一个用来代替假设计算的函数，它大约会执行2秒钟

接下来是main函数，它包含了这个健身应用中较为重要的一部分内容。用户会在生成健身计划时调用该函数。由于应用前端的交互与闭包使用的关系不大，所以我们硬编码了一些代表输入数据的值并将输出打印出来。

需要的输入数据包含下面这些：

- 一个来自用户的强度值，它会在用户请求健身计划时被要求指定，以便确定用户想要低强度训练还是高强度训练。
- 一个随机数，它会让输出的健身计划产生些许变化。

最后输出的内容就是程序生成的健身计划。示例13-2展示了我们将要使用的main函数。

src/main.rs

```
fn main() {
    let simulated_user_specified_value = 10;
    let simulated_random_number = 7;

    generate_workout(
        simulated_user_specified_value,
        simulated_random_number
    );
}
```

示例13-2：包含硬编码的main函数，这些硬编码用于模拟用户的输入和生成的随机数

为了简单起见，我们将变量simulated_user_specified_value硬编码为10，将变量simulated_random_number硬编码为7；在真正的程序中，强度值应该是在应用前端输入的，而随机数则是使用rand包来生成的，正如我们在第2章的猜数游戏中做的那样。随后，main函数会使用这些模拟的输入数值调用generate_workout函数。

现在，上下文环境一切就绪，让我们开始编写算法部分。示例13-3中的generate_workout函数包含了这个应用的业务逻辑，这也是本例中我们最为关心的部分。本例中剩下的所有代码修改都会在这个函数内进行。

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    ❶ if intensity < 25 {
        println!(
            "Today, do {} pushups!",           // 今天做 {} 个俯卧撑!
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Next, do {} situps!",           // 接下来做 {} 个仰卧起
            simulated_expensive_calculation(intensity)
        );
    } else {
        ❷ if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
            // 今天休息一下吧！别忘了补充水分!
        } ❸ else {
            println!(
                "Today, run for {} minutes!", // 今天跑步 {} 分钟!
                simulated_expensive_calculation(intensity)
            );
        }
    }
}
```

示例13-3：根据输入数据打印健身计划的业务逻辑，它多次调用了 simulated_expensive_calculation函数

示例13-3 中有多处代码调用了耗时的计算函数。其中，第一个if块❶调用了simulated_expensive_calculation两次，而第二个else块❸内部的代码又调用了它一次。

generate_workout函数在执行时会首先判断用户想要的是低强度训练（由小于25的数字表示）还是高强度训练（由大于等于25的数字表示）。

低强度训练计划会根据我们模拟的复杂算法来推荐用户需要完成的俯卧撑和仰卧起坐的个数。

如果用户想要高强度训练，那么这个函数会执行一些额外的逻辑：如果应用生成的随机数恰好是3，那么应用会推荐用户休息并补充水分；否则，我们仍然根据复杂算法的计算结果来推荐用户进行几分钟的跑步训练。

这段代码可以满足目前的业务需求，但我们必须考虑到数据科学团队需要不断地改进计算方法，他们很可能会在未来要求我们修改调用 simulated_expensive_calculation 函数的方式。为了在遇到这种情况时简化更新流程，我们想通过重构代码来使它只调用 simulated_expensive_calculation 函数一次。另外，我们还希望在不引入其他函数调用的前提下，优化那处调用了两次复杂算法函数的代码块。换句话说，我们希望在不必要时避免调用这个耗时的计算函数，在必要时也最多只调用一次。

使用函数来进行重构

有许多方法可以用来重构这个程序。首先，我们可以把重复调用 simulated_expensive_calculation 的地方提取为变量，如示例 13-4 所示。

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_result =
        simulated_expensive_calculation(intensity);

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result
            );
        }
    }
}
```

```
    }
}
}
```

示例13-4：将调用simulated_expensive_calculation的代码提取至一处，并将结果存入expensive_result变量

这个修改统一了所有针对simulated_expensive_calculation的调用，并修复了第一个if块中不必要的两次函数调用。但不幸的是，我们在所有调用了这个函数的情况下都要等待结果，这对于无须结果值的内层if代码块来讲显得异常浪费。

我们希望在程序中将代码定义在一处，但只在真正需要结果时才执行相关代码。而这正是闭包的用武之地！

使用闭包存储代码来进行重构

相较于每次在if块之前调用simulated_expensive_calculation函数，我们可以定义一个闭包，并将闭包而不是函数的计算结果存储在变量中，如示例13-5所示。实际上，我们可以直接将simulated_expensive_calculation的整个函数体都移动到闭包中。

src/main.rs

```
let expensive_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

示例13-5：定义一个闭包并将它存入expensive_closure变量中

闭包的定义放置在=之后，它会被赋值给语句左侧的expensive_closure变量。为了定义闭包，我们需要以一对竖线(|)开始，并在竖线之间填写闭包的参数；之所以选择这样的写法是因为它与Smalltalk及Ruby中的闭包定义类似。这个闭包仅有一个名为num的参数，而当闭包需要多个参数时，我们需要使用逗号来分隔它们，例如| param1, param2|。

在参数后面，我们使用了一对花括号来包裹闭包的函数体。如果这个闭包只是单行表达式，你也可以选择省略花括号。在闭包结束

后，也就是右花括号的后边，我们需要用一个分号来结束当前的let语句。因为闭包代码中的最后一行（num）没有以分号结尾，所以该行产生的值会被当作闭包的结果返回给调用者，其行为与普通函数的完全一致。

注意，这条let语句意味着expensive_closure变量存储了一个匿名函数的定义，而不是调用该匿名函数而产生的返回值。回忆一下，我们之所以使用闭包是因为想要在一个地方定义要调用的代码，将其存储起来，并在稍后的地方调用它。现在，这些可以被调用的代码已经存储在expensive_closure中了。

定义完闭包后，我们就可以修改if块中的代码来调用闭包，执行代码并求得结果了。调用闭包的方式类似于调用普通函数：先指定存储闭包定义的变量名，再跟上一对包含传入参数的括号，如示例13-6所示。

src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}
```

示例13-6：调用我们定义的expensive_closure闭包

现在，耗时的计算操作只会在一个地方被调用，而具体的代码只会在需要结果的地方得到执行。

但是，这样的改动让示例13-3中的老问题重新出现了：我们依然在第一个if块中调用了两次闭包，也就是执行了两次耗时的计算操作，进而导致用户的等待时间不合理地被延长了两倍。当然，在if块中定义一个局部变量来存储闭包结果可以解决这个问题，但你也可以利用闭包的特性提供另一种解决方案。我们稍后再来讨论它。现在，先来看一看为什么闭包定义及其相关trait中都没有出现任何的类型标注。

闭包的类型推断和类型标注

和fn定义的函数不同，闭包并不强制要求你标注参数和返回值的类型。Rust之所以要求我们在函数定义中进行类型标注，是因为类型信息是暴露给用户的显式接口的一部分。严格定义接口有助于所有人们对参数和返回值的类型取得明确共识。但是，闭包并不会被用于这样的暴露接口：它们被存储在变量中，在使用时既不需要命名，也不会被暴露给代码库的用户。

闭包通常都相当短小，且只在狭窄的代码上下文中使用，而不会被应用在广泛的场景下。在这种限定环境下，编译器能够可靠地推断出闭包参数的类型及返回值的类型，就像是编译器能够推断出大多数变量的类型一样。

强制程序员为这些短小的匿名函数标注类型会显得非常烦人，况且这些信息往往已经被编译器自动推断出来了。

不过，就和变量一样，假如你愿意为了明确性而接受不必要的繁杂作为代价，那么你仍然可以为闭包手动添加类型标注。示例13-7中的代码为示例13-5中的闭包添加了类型标注。

src/main.rs

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
}
```

```
    num  
};
```

示例13-7：为闭包中的参数和返回值添加可选的类型标注

添加类型标注之后，闭包的语法就和函数的语法更加相似了。下面的列表纵向对比了函数和闭包的定义语法，它们都实现了为参数加1并返回的行为。我们额外添加了一些空格来对齐相关的部分。你可以从这段展示中看到，除了使用竖线及省略某些语法的部分，闭包与函数的语法是多么类似：

```
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 ;
```

第一行展示的是函数定义，而第二行则是一个完整标注了类型的闭包定义。第三行省去了闭包定义中的类型标注，而第四行更是在闭包块只有一个表达式的前提下省去了花括号。这些定义都是合法且完全等效的。

闭包定义中的每一个参数及返回值都会被推导为对应的具体类型。例如，示例13-8中展示了一个直接将参数作为结果返回的闭包。当然，这个闭包除了可以用来演示并不是非常实用。注意，我们并没有为它添加类型标注：如果调用闭包两次，第一次使用String类型的参数，而第二次使用u32类型的参数，那么就会发生编译错误。

src/main.rs

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("hello"));  
let n = example_closure(5);
```

示例13-8：试图使用两种不同的类型调用同一个需要类型推导的闭包

编译器会报告如下所示的错误：

```
error[E0308]: mismatched types  
--> src/main.rs  
|  
| let n = example_closure(5);  
| ^ expected struct `std::string::String`, found  
integral variable  
|
```

```
= note: expected type `std::string::String`  
      found type `'{integer}'`
```

当我们首先使用String值调用example_closure时，编译器将闭包的参数x的类型和返回值的类型都推导为了String类型。接着，这些类型信息就被绑定到了example_closure闭包中，当我们尝试使用其他类型调用这一闭包时就会触发类型不匹配的错误。

使用泛型参数和Fn trait来存储闭包

让我们回到健身计划生成应用。在示例13-6中，代码依然不必要的多次调用了耗时的计算闭包。这个问题的一个解决方案是将耗时闭包的结果存储至变量中，并在随后需要结果的地方使用该变量而不是继续调用闭包。但需要注意的是，这种方法可能会造成大量的代码重复。

幸运的是，我们还有另一种可用的解决方案：创建一个同时存放闭包及闭包返回值的结构体。这个结构体只会在我们需要获得结果值时运行闭包，并将首次运行闭包时的结果缓存起来，这样余下的代码就不必再负责存储结果，而可以直接复用该结果。这种模式一般被称作 *记忆化* (memoization) 或 *惰性求值* (lazy evaluation)。

为了将闭包存储在结构体中，我们必须明确指定闭包的类型，因为结构体各个字段的类型在定义时就必须确定。但需要注意的是，每一个闭包实例都有它自己的匿名类型。换句话说，即便两个闭包拥有完全相同的签名，它们的类型也被认为是不一样的。为了在结构体、枚举或函数参数中使用闭包，我们需要使用在第10章讨论过的泛型及trait约束。

标准库中提供了一系列Fn trait，而所有的闭包都至少实现了Fn、FnMut及FnOnce中的一个trait。我们将在“使用闭包捕获上下文环境”一节中讨论这3种trait之间的区别。在本例中，我们可以使用Fn trait。

我们会在Fn的trait约束中添加代表了闭包参数和闭包返回值的类型。在这个例子中，闭包有一个u32类型的参数并返回一个u32值，因此我们指定的trait约束就是Fn(u32) -> u32。

示例13-9定义了一个Cacher结构体，它存储了一个闭包和一个可选结果值。

src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

示例13-9：存储了一个闭包calculation和一个可选结果值value的结构体Cacher

Cacher结构体拥有一个泛型T的calculation字段，而trait约束规定的这个T代表一个使用Fn trait的闭包。另外，我们存储在calculation中的闭包必须有一个u32参数（在Fn后面的括号中指定），同时必须返回一个u32值（在->后面指定）。

注意

函数同样也可以实现这3个Fn trait。假如代码不需要从环境中捕获任何值，那么我们也可以使用实现了Fn trait的函数而不是闭包。

另外一个字段value的类型是Option<u32>。在运行闭包之前，value会被初始化为None。而当使用Cacher的代码请求闭包的执行结果时，Cacher会运行闭包并将结果存储在value的Some变体中。之后，如果代码重复请求闭包的结果，Cacher就可以避免再次运行闭包，而将缓存在Some变体中的结果返回给调用者。

示例13-10展示了刚刚讨论的与value相关的逻辑。

src/main.rs

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }
}
```

```

        }
    }

❸ fn value(&mut self, arg: u32) -> u32 {
    match self.value {
        ❹ Some(v) => v,
        ❺ None => {
            let v = (self.calculation)(arg);
            self.value = Some(v);
            v
        }
    }
}

```

示例13-10：Cacher的缓存逻辑

我们希望Cacher自行管理结构体中的各个字段，从而避免调用代码意外地直接修改这些字段中的值，因此这些字段是私有的。

Cacher::new函数会接收一个泛型参数❶，它与Cacher结构体有着相同的trait约束❷。调用Cacher::new会返回一个在calculation字段中存储了闭包的Cacher实例❸。因为我们还未执行过这个闭包，所以value字段的值被设置为了None。

当调用代码需要获得闭包的执行结果时，它们将会调用value方法而不是调用闭包本身❹。这个方法会检查self.value中是否已经拥有了一个属于Some变体的返回值，如果有的话，它会直接返回Some中的值作为结果而无须再次执行闭包❺。

而如果self.value是None的话，则代码会先执行self.calculation中的闭包并将返回值存储在self.value中以便将来使用，最后再把结果返回给调用者❻。

示例13-11展示了如何在示例13-6的generate_workout函数中使用Cacher结构体。

src/main.rs

```

fn generate_workout(intensity: u32, random_number: u32) {
❶ let mut expensive_result = Cacher::new(|num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
});

```

```

if intensity < 25 {
    println!(
        "Today, do {} pushups!",
        ② expensive_result.value(intensity)
    );
    println!(
        "Next, do {} situps!",
        ③ expensive_result.value(intensity)
    );
} else {
    if random_number == 3 {
        println!("Take a break today! Remember to stay hydrated!");
    } else {
        println!(
            "Today, run for {} minutes!",
            ④ expensive_result.value(intensity)
        );
    }
}
}

```

示例13-11：在generate_workout函数中使用Cacher来将其中的缓存逻辑抽象出来

上面的代码没有直接将闭包存储在变量中，而是将闭包存储在了新创建的Cacher实例中①。接着，在每一个需要结果的地方②③④，我们都调用了Cacher实例的value方法。无论我们调用多少次value方法，或者一次都不调用，真正耗时的计算操作都最多只会执行一次。

请尝试使用示例13-2中的main函数来运行这段代码。你可以通过修改simulated_user_specified_value和simulated_random_number变量的值来分别验证不同条件分支的运行结果，并且确认calculating slowly... 提示只在需要进行计算操作时出现且仅出现一次。因为Cacher实现了必要的逻辑来确保不会执行多于需求的耗时计算操作，所以我们现在可以专注于generate_workout中的业务逻辑了。

Cacher实现的局限性

缓存值其实是一种相当通用且有效的策略，你可能会想要在其他部分的闭包代码中使用它。但必须指出的是，当前的Cacher实现存在两个问题，导致我们很难在不同的上下文环境中复用它。

第一个问题是，Cacher实例假设value方法会为不同的arg参数返回相同的值。也就是说，类似于下面的Cacher测试将会失败：

```
# [test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 2);
}
```

这个测试中创建的Cacher实例会存储一个原样返回参数值的闭包。它分别使用了1和2作为arg参数来调用Cacher实例的value方法。我们期望在参数为2时调用value方法会返回2。

使用示例13-9与示例13-10中的Cacher实现来运行这段测试，会在执行assert_eq! 指令时失败并输出如下所示的信息：

```
thread 'call_with_different_values' panicked at 'assertion failed: `(left == right)`
  left: `1`,
  right: `2`, src/main.rs
```

这里的问题在于我们第一次使用1作为参数来执行c.value时，Cacher实例就将Some(1)存储在了self.value中。在这之后，无论我们在调用value方法时传入的值是什么，它都只会返回1。

解决这个问题的方法是让Cacher存储一个哈希表而不是单一的值。这个哈希表使用传入的arg值作为关键字，并将关键字调用闭包后的结果作为对应的值。相应地，value方法不再简单地判断self.value的值是Some还是None，而是会检查哈希映射里是否存在arg这个关键字。如果存在的话，Cacher就直接返回对应的值；如果不存在的话，则调用闭包，使用arg关键字将结果存入哈希表之后再返回。

这个Cacher实现的第二个问题是它只能接收一个获取u32类型参数并返回u32类型的值的闭包。但我们可能想要缓存的是一个获取字符串切片参数并返回usize值的闭包。为了修复这一问题，你可以自行尝试引入更多的泛型参数来提升Cacher功能的灵活性。

使用闭包捕获上下文环境

在健身计划生成应用中，我们只把闭包视作一个内部的匿名函数来使用。但除此之外，闭包还有一项函数所不具备的功能：它们可以捕获自己所在的环境并访问自己被定义时的作用域中的变量。

示例13-12中有一个存储在equal_to_x变量中的闭包，它使用了自己所处环境内的变量x。

src/main.rs

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;
    let y = 4;
    assert!(equal_to_x(y));
}
```

示例13-12：这个闭包引用了自身封闭作用域中的变量

在上面的代码中，即使x不是equal_to_x的参数，equal_to_x闭包也可以使用定义在同一个作用域中的变量x。

这个功能是函数所不具备的，类似于下面的代码是无法通过编译的：

src/main.rs

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;
    assert!(equal_to_x(y));
}
```

错误结果如下所示：

```
error[E0434]: can't capture dynamic environment in a fn item; use the || { ...
} closure form instead
--> src/main.rs
|
4 |     fn equal_to_x(z: i32) -> bool { z == x }
|           ^
```

编译器甚至会提醒我们这一特性只能被用于闭包！

当闭包从环境中捕获值时，它会使用额外的空间来存储这些值以便在闭包体内使用。在大多数情况下，我们都不要在执行代码时捕获环境，也不想要为这种场景产生额外的内存开销。因为函数不被允许从环境中捕获变量，所以定义和使用函数永远不会产生这类开销。

闭包可以通过3种方式从它们的环境中捕获值，这和函数接收参数的3种方式是完全一致的：获取所有权、可变借用及不可变借用。这3种方式被分别编码在如下所示的3种Fn系列的 trait中：

- FnOnce意味着闭包可以从它的封闭作用域中，也就是闭包所处的环境中，消耗捕获的变量。为了实现这一功能，闭包必须在定义时取得这些变量的所有权并将它们移动至闭包中。这也是名称FnOnce中Once一词的含义：因为闭包不能多次获取并消耗掉同一变量的所有权，所以它只能被调用一次。
- FnMut可以从环境中可变地借用值并对它们进行修改。
- Fn可以从环境中不可变地借用值。

当你创建闭包时，Rust会基于闭包从环境中使用值的方式来自动推导出它需要使用的trait。所有闭包都自动实现了FnOnce，因为它们至少都可以被调用一次。那些不需要移动被捕获变量的闭包还会实现FnMut，而那些不需要对被捕获变量进行可变访问的闭包则同时实现了Fn。在示例13-12中，因为equal_to_x闭包只需要读取x中的值，所以它仅仅不可变地借用了x并实现了Fn trait。

假如你希望强制闭包获取环境中值的所有权，那么你可以在参数列表前添加move关键字。这个特性在把闭包传入新线程时相当有用，它可以将捕获的变量一并移动到新线程中去。

我们会在第16章讨论并发的时候接触到更多move闭包的用法。现在先来看一看下面的代码，它是在示例13-12中代码的基础上修改的：首先在闭包定义中添加了move关键字，接着用动态数组替代了被捕获的整型，因为整型只会被复制而不会被移动。请注意，这段代码还无法通过编译。

src/main.rs

```
fn main() {
    let x = vec![1, 2, 3];

    let equal_to_x = move |z| z == x;
    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];
    assert!(equal_to_x(y));
}
```

运行代码，我们会看到如下所示的错误：

```
error[E0382]: use of moved value: `x`
--> src/main.rs:6:40
|
4 |     let equal_to_x = move |z| z == x;
|           ----- value moved (into closure) here
5 |
6 |     println!("can't use x here: {:?}", x);
|           ^ value used here after move
|
= note: move occurs because `x` has type `std::vec::Vec<i32>`, which does not
       implement the `Copy` trait
```

因为我们添加了move关键字，所以x的值会在定义闭包时移动至闭包中。由于闭包拥有了x的所有权，所以main函数就无法在println!语句中使用x了。移除println! 一行将会修复这个示例。

在大部分情形下，当你需要指定某一个Fn系列的 trait时，可以先尝试使用Fn trait，编译器会根据闭包体中的具体情况来告诉你是否需要FnMut或FnOnce。

可捕获环境的闭包在作为函数参数传递时非常有用，为了演示这一情形，让我们接着来讨论下一个主题：迭代器。

使用迭代器处理元素序列

迭代器模式允许你依次为序列中的每一个元素执行某些任务。迭代器会在这个过程中负责遍历每一个元素并决定序列何时结束。只要使用了迭代器，我们就可以避免手动去实现这些逻辑。

在Rust中，迭代器是惰性的（lazy）。这也就意味着创建迭代器后，除非你主动调用方法来消耗并使用迭代器，否则它们不会产生任何的实际效果。例如，示例13-13中的代码通过调用`Vec<T>`的`iter`方法创建了一个用于遍历动态数组`v1`的迭代器。这段代码本身并不会产生任何影响。

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
```

示例13-13：创建一个迭代器

一旦创建好迭代器，我们就可以用多种方式来使用它。在第3章的示例3-5中，我们就曾经在`for`循环内使用迭代器来依次遍历元素并执行相关的任务，虽然我们当时一笔带过了`iter`函数的具体用途。

示例13-14中的代码将创建迭代器从在`for`循环中使用迭代器中分离出来。迭代器被存储在`v1_iter`变量中，此时还没有出现任何的遍历。只有当使用了迭代器`v1_iter`的`for`循环开始执行时，迭代器才开始为每一次循环产生一个元素，并将每个值打印出来。

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {}", val);
}
```

示例13-14：在`for`循环中使用迭代器

某些语言没有在标准库中提供迭代器特性，为了实现类似的功能，你通常都需要定义一个从0开始的变量作为索引来获得动态数组中的值，并在循环中逐次递增这个变量的值，直到它达到动态数组的总长度为止。

迭代器会为我们处理所有上述的逻辑，这减少了重复代码并消除了潜在的混乱。另外，迭代器还可以用统一的逻辑来灵活处理各种不同种类的序列，而不仅仅只是像动态数组一样可以进行索引的数据结构。让我们来看一看迭代器是如何做到这一点的。

Iterator trait和next方法

所有的迭代器都实现了定义于标准库中的Iterator trait。该trait的定义类似于下面这样：

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // 这里省略了由Rust给出的默认实现方法  
}
```

注意，这个定义使用了两种新语法：type Item和Self::Item，它们定义了trait的关联类型（associated type）。我们会在第19章深入讨论关联类型。现在，你只需要知道，这段代码表明，为了实现Iterator trait，我们必须定义一个具体的Item类型，而这个Item类型会被用作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只要求实现者手动定义一个方法：next方法，它会在每次被调用时返回一个包裹在Some中的迭代器元素，并在迭代结束时返回None。

我们可以直接在迭代器上调用next方法。示例13-15中创建了一个动态数组的迭代器，并演示了重复调用迭代器的next方法会得到怎样的返回值。

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13-15：手动调用迭代器的next方法

注意，这里的v1_iter必须是可变的，因为调用next方法改变了迭代器内部用来记录序列位置的状态。换句话说，这段代码消耗或使用了迭代器，每次调用next都吃掉了迭代器中的一个元素。在刚才的for循环中我们之所以不要求v1_iter可变，是因为循环取得了v1_iter的所有权并在内部使得它可变了。

另外还需要注意到，iter方法生成的是一个不可变引用的迭代器，我们通过next取得的值实际上是指向动态数组中各个元素的不可变引用。如果你需要创建一个取得v1所有权并返回元素本身的迭代器，那么你可以使用into_iter方法。类似地，如果你需要可变引用的迭代器，那么你可以使用iter_mut方法。

消耗迭代器的方法

标准库为Iterator trait提供了多种包含默认实现的方法，你可以在标准库的API文档中查询Iterator trait相关的页面来进一步了解它们。这些方法中的一部分会在它们的定义中调用next方法，这也是我们需要在实现Iterator trait时手动定义next方法的原因。

这些调用next的方法也被称为消耗适配器（consuming adaptor），因为它们同样消耗了迭代器本身。以sum方法为例，这个方法会获取迭代器的所有权并反复调用next来遍历元素，进而导致迭代器被消耗。在迭代过程中，它会对所有元素进行求和并在迭代结束后将总和作为结果返回。示例13-16中的测试展示了sum方法的使用场景。

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13-16：调用sum方法来得到迭代器中所有元素的总和

由于我们在调用sum的过程中获取了迭代器v1_iter的所有权，所以该迭代器无法继续被随后的代码使用。

生成其他迭代器的方法

Iterator trait还定义了另外一些被称为迭代器适配器(iterator adaptor)的方法，这些方法可以使你将已有的迭代器转换成其他不同类型的迭代器。你可以链式地调用多个迭代器适配器完成一些复杂的操作，同时保持代码易于阅读。但因为所有的迭代器都是惰性的，所以你必须调用一个消耗适配器的方法才能从迭代器适配器中获得结果。

示例13-17展示了一个名为map的迭代器适配器方法，它接收一个用来处理所有元素的闭包作为参数并会生成一个新的迭代器。新的迭代器同样会遍历动态数组中的所有元素并返回经过闭包处理后增加了1的值。这段代码目前会产生一个警告。

src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

示例13-17：调用迭代器适配器map来创建新的迭代器

编译器给出的警告如下所示：

```
warning: unused `std::iter::Map` which must be used: iterator adaptors are lazy
and do nothing unless consumed
--> src/main.rs:4:5
|
4 |     v1.iter().map(|x| x + 1);
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^
|
= note: #[warn(unused_must_use)] on by default
```

实际上，示例13-17中的代码没有做任何工作，我们定义的闭包一次都没有被调用过。编译器通过警告提示我们：迭代器适配器是惰性的，除非我们消耗迭代器，否则什么事情都不会发生。

为了修复这一问题并消耗迭代器，我们可以使用`collect`方法。这个方法曾经在第12章的示例12-1中配合`env::args`使用过，它会消耗迭代器并将结果值收集到某种集合数据类型中。

在示例13-18中，我们遍历了通过`map`方法生成的新迭代器并将返回的结果收集到一个动态数组中。最终，这个动态数组会包含原数组中的所有元素加1之后的值。

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13-18：调用`map`方法创建新迭代器，接着再调用`collect`方法将其消耗掉并得到一个动态数组

由于`map`接收一个闭包作为参数，所以我们可以对每个元素指定想要执行的任何操作。这一示例很好地演示了如何既能复用`Iterator trait`提供的迭代功能，又能通过闭包来自定义部分具体行为。

使用闭包捕获环境

在介绍了迭代器之后，我们现在可以通过`filter`迭代器适配器来演示闭包捕获环境的一种常见用法了。迭代器的`filter`方法会接收一个闭包作为参数，这个闭包会在遍历迭代器中的元素时返回一个布尔

值，而每次遍历的元素只有在闭包返回true时才会被包含在filter生成的新迭代器中。

在示例13-19中，我们传入一个从环境中捕获了变量shoe_size的闭包来使用filter方法，这个闭包会遍历一个由Shoe结构体实例组成的集合，并返回集合中拥有特定尺寸的鞋子。

src/lib.rs

```
# [derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

❶ fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
❷     shoes.into_iter()
        ❸     .filter(|s| s.size == shoe_size)
        ❹     .collect()
}

#[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_my_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13-19：传入一个捕获了变量shoe_size的闭包来使用filter方法

shoes_in_my_size函数接收一个由鞋子组成的动态数组和一个鞋子的尺寸作为参数❶，它会返回一个只包含指定尺寸鞋子的动态数组。

在shoes_in_my_size函数体中，我们调用了into_iter来创建可以获取动态数组所有权的迭代器❷。接着，我们调用filter来将这个迭

代器适配成一个新的迭代器，新的迭代器只会包含闭包返回值为true的那些元素③。

闭包从环境中捕获了shoe_size参数并将它的值与每只鞋子的尺寸进行比较，这一过程会过滤掉所有不符合尺寸的鞋子。最后，调用collect来将迭代器适配器返回的值收集到动态数组中，并将其作为函数的结果返回④。

最后的测试表明，我们在调用shoes_in_my_size时只会得到符合指定尺寸的鞋子。

使用Iterator trait来创建自定义迭代器

我们已经向你展示了如何通过调用动态数组的iter、into_iter及iter_mut方法来创建迭代器。你也可以采用类似的方法为标准库中的其他集合类型（例如哈希表）创建迭代器。除此之外，你还可以通过实现Iterator trait来创建拥有自定义行为的迭代器。正如之前所提到的，你只需要提供一个next方法的定义即可实现Iterator trait。一旦完成该方法定义，你就可以使用其他一切拥有默认实现的Iterator trait提供的方法。

为了方便演示，我们会创建一个从1遍历到5的迭代器。首先，我们会创建一个结构体，以便存储迭代过程中所需的数值。接着我们会运用这些数值，为这个结构体实现Iterator trait，从而使它成为迭代器。

示例13-20定义了Counter结构体，并定义了一个关联函数new用于创建Counter的实例。

src/lib.rs

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

示例13-20：定义Counter结构体及其new函数，new函数会以0作为count的初始值创建Counter的实例

Counter结构体只有一个名为count的字段，它存储的u32值被用来记录迭代器从1遍历到5这个过程中的状态。count字段被设置为私有的，以便Counter能够独立管理其中的值。new函数则确保了任何一个新实例中的count字段的值都会从0开始。

接下来，我们会为Counter类型实现Iterator trait，并通过定义next方法的函数体来指定迭代器被使用时的具体行为，如示例13-21所示。

src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

示例13-21：为Counter结构体实现Iterator trait

我们将迭代器的关联类型Item指定为了u32，这也就意味着迭代器将返回一个u32序列。再次说明一下，不用担心此处出现的关联类型，我们会在第19章讨论它。

因为我们希望返回值的序列从1开始，而这个迭代器在每次迭代时都会对其内部的状态加1，所以count的值被初始化为了0。当count的值小于6时，next会将当前值包裹在Some中返回，而当count大于或等于6时，迭代器则会返回None。

使用Counter迭代器的next方法

一旦实现了Iterator trait，我们就拥有了一个迭代器！示例13-22中的测试借助next方法来直接地使用Counter结构体的迭代器功能，

它的用法与示例13-15中基于动态数组创建出来的迭代器完全一致。

src/lib.rs

```
# [test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

示例13-22： 测试next方法实现的功能

这个测试首先在counter变量中创建了一个新的Counter实例，接着反复调用next来验证实现的迭代器行为是否符合我们的期望，也就是返回从1到5的值。

使用其他的Iterator trait方法

我们只需要提供next方法的定义便可以使用标准库中那些拥有默认实现的Iterator trait方法，因为这些方法都依赖于next方法的功能。

例如，假设我们希望将一个Counter实例产生的值与另一个Counter实例跳过首元素后的值一一配对，接着将配对的两个值相乘，最后再对乘积中能被3整除的那些数字求和。示例13-23中的测试演示了这一过程。

src/lib.rs

```
# [test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();

    assert_eq!(18, sum);
}
```

示例13-23： 在Counter迭代器上使用不同的Iterator trait方法

注意，`zip`方法只会产生4对值，它在两个迭代器中的任意一个返回`None`时结束迭代，所以理论上的第五对值(5, `None`)永远不会被生成出来。

因为我们指定了`next`方法的具体行为，而标准库又对其他调用`next`的方法提供了默认实现，所以我们能够合法地使用所有这些方法。

改进I/O项目

在了解了迭代器方面的知识后，我们现在可以使用迭代器来改进第12章中的I/O项目了，它会使项目中的代码变得更加简单明了。让我们看一看迭代器会如何改进Config::new函数和search函数的实现。

使用迭代器代替clone

在示例12-6中，我们获取了String序列值的一个切片，随后又利用索引访问并克隆这些值来创建新的Config结构体实例，从而使Config结构体可以拥有这些值的所有权。示例13-24重现了示例12-23中Config::new函数的实现。

src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
        Ok(Config { query, filename, case_sensitive })
    }
}
```

示例13-24：在示例12-23中实现的Config::new函数

我们在编写这个函数时曾经让你不要在意clone引发的性能损耗，因为我们在将来改进这一行为。好吧，就是现在了！

之所以需要在这里使用clone是因为new函数并不持有args参数内元素的所有权，我们获得的仅仅是一个String序列的切片。为了返回Config实例的所有权，我们必须克隆Config的query字段和filename字段中的值，只有这样，Config才能拥有这些值的所有权。

在学习了迭代器之后，我们现在可以在new函数中直接使用迭代器作为参数来获取其所有权，而无须再借用切片。我们还可以使用迭代器附带的功能来进行长度检查和索引。这将使Config::new函数的责任范围更加明确，因为我们通过迭代器将读取具体值的工作分离了出去。

只要Config::new能够获取迭代器的所有权，我们就可以将迭代器产生的String值移动到Config中，而无须调用clone进行二次分配。

直接使用返回的迭代器

打开I/O项目中的*src/main.rs*文件，其中的代码应该如下所示：

src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --略

    --
}
```

我们会将示例12-24中main函数的起始部分改写为示例13-25中的样子。这段代码在我们修改完Config::new之前暂时还不能通过编译。

src/main.rs

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --略
```

```
--  
}
```

示例13-25：将env::args的返回值传递给Config::new

env::args函数的返回值其实就是一个迭代器！与其将迭代器产生的值收集至动态数组后再作为切片传入Config::new中，不如选择直接传递迭代器本身。

接下来，我们需要更新Config::new的定义。在Hello项目的src/lib.rs文件中，将Config::new的签名修改为示例13-26中的样子。注意，在函数体修改完毕之前这段代码仍然不能通过编译。

src/lib.rs

```
impl Config {  
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {  
        // --略  
  
--
```

示例13-26：修改Config::new的签名来接收一个迭代器

env::args函数的标准库文档表明，它会返回一个类型为std::env::Args的迭代器。据此，我们将Config::new函数签名中的args参数的类型从&[String]类型改为了std::env::Args类型。由于我们获得了args的所有权并会在函数体中通过迭代来改变它，所以我们需要在args参数前指定mut关键字来使其可变。

使用Iterator trait方法来替代索引

接下来，我们会对应地修复Config::new函数体。因为标准库文档指出std::env::Args实现了Iterator trait，所以我们能够基于它的实例调用next方法。示例13-27使用next方法更新了示例12-23中的代码。

src/lib.rs

```
impl Config {  
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {  
        args.next();  
  
        let query = match args.next() {
```

```

        Some(arg) => arg,
        None => return Err("Didn't get a query string"),
    };

    let filename = match args.next() {
        Some(arg) => arg,
        None => return Err("Didn't get a file name"),
    };

    let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

    Ok(Config { query, filename, case_sensitive })
}
}
}

```

示例13-27：使用迭代器的方法重新实现Config::new函数

请记住，env::args的返回值的第一个值是程序本身的名称。为了忽略它，我们必须先调用一次next并忽略返回值。随后，我们再次调用next来取得用于Config中query字段的值。如果next返回一个Some变体，我们就会使用match来提取这个值；而如果它返回的是None，则表明用户没有提供足够的参数，我们需要让整个函数提前返回Err值。接下来，对filename字段进行类似的处理。

使用迭代器适配器让代码更加清晰

我们还可以在这个I/O项目的search函数中使用迭代器，示例13-28重现了示例12-19中的代码。

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13-28：示例12-19中的search函数实现

我们可以通过迭代器适配器的相关方法来更加简单明了地编写这段代码。另外，这样做还能避免使用可变的临时变量`results`。函数式编程风格倾向于在程序中最小化可变状态的数量来使代码更加清晰。消除可变状态也使我们可以在未来通过并行化来提升搜索效率，因为我们不再需要考虑并发访问`results`动态数组时的安全问题了。修改后的代码如示例13-29所示。

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13-29：使用迭代器适配器实现search函数

`search`函数被用来返回`contents`中包含`query`的所有行。与示例13-19中使用`filter`的例子类似，这段代码使用了`filter`适配器来进行过滤，从而只保留满足`line.contains(query)`条件的行。接着，我们使用`collect`方法将所有匹配的行收集成一个动态数组。这样，代码就简单多了！类似地，你可以自行将`search_case_insensitive`函数也修改成使用迭代器适配器的形式。

你会喜欢什么样的代码风格呢？是示例13-28中平铺直叙的原始实现，还是示例13-29中使用迭代器的版本？在这个问题上，大多数Rust开发者都更倾向于使用迭代器风格。初学时，你也许会觉得迭代器有些难以理解，而一旦你了解并习惯了各种迭代器适配器的使用方法，那么理解迭代器就会变得相当简单。迭代器可以让开发者专注于高层的业务逻辑，而不必陷入编写循环、维护中间变量这些具体的细节中。通过高层抽象去消除一些惯例化的模板代码，也可以让代码的重点逻辑（例如`filter`方法的过滤条件）更加突出。

不过，这两种实现真的等价吗？仅从直觉上看，你也许会觉得更接近底层的循环实现要快一些。接下来，就让我们来讨论一下性能问题。

比较循环和迭代器的性能

为了决定使用循环还是迭代器，你需要知道哪个版本的search函数要更快一些：是直接使用for循环的版本，还是使用迭代器的版本。

我们使用 Sir Arthur Conan [1] 的小说 *The Adventures of Sherlock Holmes* (《福尔摩斯探案集》) 来进行一次性能测试，这个测试会将整本小说读入一个String中，并搜索所有包含了单词 *the* 的文本行。使用for循环和迭代器分别实现的search函数的测试结果如下：

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

性能测试显示出来的结果竟然是迭代器版本要稍微快了一些！我们就不在这里深入分析测试代码本身了，因为我们的目的并不是为了证明两个版本完全相等，而只是想让你了解评判这两种实现的基本方法。

为了让性能测试更加全面，你也可以使用不同内容的文本、不同的搜索单词和其他所有的可变情况来检验比较结果。这里的重点在于：尽管迭代器是一种高层次的抽象，但它在编译后生成了与手写底层代码几乎一样的产物。迭代器是Rust语言中的一种零开销抽象（zero-cost abstraction），这个词意味着我们在使用这些抽象时不会引入额外的运行时开销。它与Bjarne Strostrup，也就是C++最初的设计者和实现者，在*Foundations of C++* (2012) 中定义的零开销（zero-overhead）如出一辙：

C++的实现大体上遵从了零开销原则：你无须为你没有使用过的功能付出代价。甚至更进一步地，你无法为你使用的那些功能编写出更好的代码。

还有一个例子，其代码如下所示，这段代码来自一个实际的音频解码器。这个解码算法基于线性预测来将之前的样本拟合成一个线性函数，并用它去预测未来可能出现的样本。这段代码使用了链式的迭代器来对作用域中的以下3个变量进行数学计算：buffer是一段数据的切片，coefficients 是一个长度为12的数组，qlp_shift代表需要移动的二进制位数。注意，我们只在例子中声明了变量而没有赋值。尽管这段代码在脱离了原有的上下文环境后没有太大的意义，但它仍然可以作为一个简捷、真实的案例来演示Rust是如何将高层概念转换为底层代码的。

```
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中所有的12个元素，并用zip方法将其与buffer的前12个值一一配对。接着，将每一对数值相乘并对所有得到的乘积求和，最后将总和向右移qlp_shift位得到结果。

音频解码器这类程序往往非常看重计算过程的性能表现。我们在这里创建了1个迭代器和2个适配器，并消耗了它们产出的值。Rust会将这段代码编译成什么样的汇编代码呢？好吧，在我们编写本书的时候，它已经能够被编译成与手写汇编几乎一样的产出物。遍历coefficients根本不会用到循环：因为Rust知道这里会迭代12次，所以它直接“展开”（unroll）了循环。展开是一种优化策略，它通过将循环代码展开成若干份重复的代码来消除循环控制语句带来的性能开销。

这样能让所有coefficients中的值都存储在寄存器中，进而使得对它们的访问变得异常快速。同时，我们也就无须在运行时浪费时间对数组访问进行边界检查了。Rust引入的所有这些优化使最终产出的代码极为高效。现在你知道了，我们完全可以无所畏惧地使用迭代器

和闭包！它们既能够让代码在观感上保持高层次的抽象，又不会因此带来任何运行时性能损失。

[1] 译者注：阿瑟·柯南·道尔（1859年—1930年），英国作家、医生。创造了著名侦探人物“夏洛克·福尔摩斯”。

总结

闭包和迭代器是Rust受函数式编程语言启发而实现的功能。它们帮助Rust在清晰地表达出高层次抽象概念的同时兼顾了底层性能。闭包和迭代器的实现保证了运行时性能不会受到影响。这是Rust努力实现零开销抽象这个目标的重要一环。

现在，我们已经提高了I/O项目中代码的表达力，接下来我们会开始讨论cargo工具的一些高级特性，并利用它来帮助我们更方便地和世界分享这个项目。

第14章

进一步认识Cargo及crates.io



到目前为止，我们仅仅使用过一些基础的Cargo特性来构建、运行及测试代码，但其实它还有相当多的其他功能。我们将在本章讨论这些更为高级的特性，并向你展示如何做下面这些事情：

- 使用发布配置来定制构建。
- 将代码库发布到crates.io上。
- 使用工作空间来组织更大的项目。
- 下载安装crates.io提供的二进制文件。
- 使用自定义命令来扩展Cargo。

当然，Cargo还有更多本书没有机会覆盖到的特性，你可以在官方网站查看它的文档来获得更为全面细致的介绍。

使用发布配置来定制构建

Rust中的发布配置（release profile）是一系列预定义好的配置方案，它们的配置选项各有不同，但都允许程序员对细节进行定制修改。这些配置方案使得程序员可以更好地来控制各种编译参数。另外，每一套配置都是相互独立的。

Cargo最常用的配置有两种：执行cargo build时使用的dev配置，以及执行cargo build --release时使用的release配置。dev配置中的默认选项适合在开发过程中使用，而release配置中的默认选项则适合在正式发布时使用。

你也许会觉得这些配置的名称非常眼熟，因为我们已经在构建的输出中多次见过它们了：

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

以上输出中的dev和release表明了编译器正在使用不同的配置。

当项目的*Cargo.toml*文件中没有任何[profile.*]区域时，Cargo针对每个配置都会有一套可以应用的默认选项。通过为任意的配置添加[profile.*]区域，我们可以覆盖默认设置的任意子集。例如，下面是opt-level选项分别在dev与release配置中的默认值：

Cargo.toml

```
[profile.dev]
opt-level = 0
```

```
[profile.release]
opt-level = 3
```

选项 opt-level 决定了 Rust 在编译时会对代码执行何种程度的优化，从 0 到 3 都是合法的配置值。越高级的优化需要消耗越多的编译时间，当你处于开发阶段并常常需要编译代码时，你也许宁可牺牲编译产出物的运行速度，也想要尽可能地缩短编译时间。这就是 dev 配置下的默认 opt-level 值为 0 的原因。而当你准备好最终发布产品时，则最好花费更多的时间来编译程序。因为你只需要在发布时编译一次，但却会多次运行编译后的程序，所以发布模式会使用更长的编译时间来交换更佳的运行时性能。这就是 release 配置下的默认 opt-level 值为 3 的原因。

你可以在 *Cargo.toml* 中指定不同的编译选项来覆盖它们的默认设置。例如，假设你希望将 dev 配置中的优化级别修改为 1，那么我们可以在 *Cargo.toml* 文件中添加下面两行：

Cargo.toml

```
[profile.dev]
opt-level = 1
```

这段配置覆盖了对应选项的默认值 0。当你再次运行 cargo build 时，Cargo 会使用我们指定的 opt-level 值并在其他选项上保持 dev 的默认配置。将 opt-level 设置为 1 会让 Cargo 比在默认配置下多执行一些优化，但仍然没有发布时使用的优化那么多。

你可以在 Rust 官方网站参阅 Cargo 的在线文档来获得所有的可用选项及它们在各个配置中的默认值。

将包发布到crates.io上

我们在之前的项目中曾经使用过来自crates.io的包作为依赖，但与此同时，你也可以发布自己的包来与他人分享代码。由于crates.io的包注册表会以源代码的形式来分发你的包，所以由它托管的包大部分都是开源的。

Rust和Cargo提供了一些功能来帮助人们更轻松地找到并使用你所发布的包。接下来，我们就开始讨论一下这些功能并演示如何发布一个包。

编写有用的文档注释

准确无误的包文档有助于用户理解这个包的用途及具体的使用方法，编写文档是一件值得你投入时间去做的工作。我们在第3章学习过如何使用双斜线（//）来为代码编写注释，除此之外，Rust还提供了一种特殊的文档注释（documentation comment）。以这种方式编写的注释内容可以被生成为HTML文档。这些HTML文档会向感兴趣的用户展示公共API的文档注释内容，它的作用在于描述当前包的使用方法而不是包内部的实现细节。

我们可以使用三斜线（///）而不是双斜线来编写文档注释，并且可以在文档注释中使用Markdown语法来格式化内容。文档注释被放置在它所说明的条目之前。示例14-1展示了my_crate包中为add_one函数编写的文档注释：

src/lib.rs

```
/// 将传入的数字加
```

```
///  
/// # 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：为函数编写的文档注释

在上面的代码中，我们首先描述了add_one函数的用途，接着开始了一段名为Examples的区域并提供了一段演示add_one函数使用方式的代码。我们可以通过运行cargo doc命令来基于这段文档注释生成HTML文档。这条命令会调用Rust内置的rustdoc工具在 *target/doc* 路径下生成HTML文档。

为了方便，你也可以调用cargo doc --open来生成并自动在浏览器中打开当前的包的文档（以及所有依赖包的文档）。打开浏览器后，导航到add_one函数，你应该能够看到文档注释被渲染出来的效果，如图14-1所示。

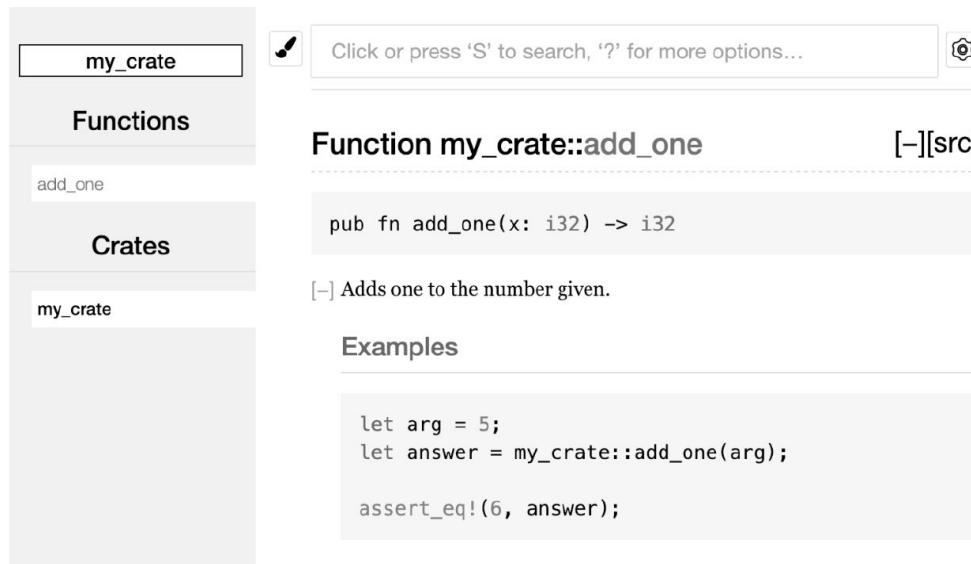


图14-1 add_one函数的HTML文档

常用的文档注释区域

示例14-1中使用Markdown标题语法# Examples在HTML文档中创建了标题为“Examples”的区域。除此之外，包的作者还经常会在文档中使用下面一些区域：

- **Panics**，指出函数可能引发panic的场景。不想触发panic的调用者应当确保自己的代码不会在这些场景下调用该函数。
- **Errors**，当函数返回Result作为结果时，这个区域会指出可能出现的错误，以及造成这些错误的具体原因，它可以帮助调用者在编写代码时为不同的错误采取不同的措施。
- **Safety**，当函数使用了unsafe关键字（在第19章讨论）时，这个区域会指出当前函数不安全的原因，以及调用者应当确保的使用前提。

大部分的文档注释都不需要拥有全部这些区域，但你可以将它作为一个检查列表来提醒自己需要在文档中编写哪几部分。

将文档注释用作测试

在文档注释中增加示例可以帮助用户理解代码库的使用方式。除此之外，cargo test会在执行时将文档注释中的代码示例作为测试去运行。没有什么比一个附带示例的文档对开发者更为友好了，但也没有什么比无法正常工作的示例更为糟糕了，要知道代码可能在文档编写完毕之后发生改动并破坏示例的有效性。假如我们为示例14-1中add_one函数所在的文档运行cargo test，你将会在测试结果中看到如下所示的内容：

```
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
```

如果现在改变函数实现或示例代码来使示例中的assert_eq!触发panic，那么再次运行cargo test，我们就会看到文档测试捕捉到了示例与文档不再同步的问题！

在条目内部编写注释

还有一种文档注释形式：`||!`，它可以为包裹当前注释的外层条目（而不是紧随注释之后的条目）添加文档。这种文档注释通常被用在包的根文件（也就是惯例上的 `src/libs.rs`）或模块的根文件上，分别为整个包或整个模块提供文档。

例如，假设我们需要为含有`add_one`函数的`my_crate`包添加描述性文档，那么我们就可以在 `src/libs.rs` 文件的起始处增加以`||!`开头的文档注释，如示例14-2所示。

src/lib.rs

```
///! # My Crate
///!
///! my_crate是一系列工具的集合
///
///! 这些工具被用来简化特定的计算操作
///
///! 将传入的数字加
1
// --略
--
```

示例14-2：为整个my_crate包编写的文档

注意，最后一个`||!`注释行的后面没有任何可供注释的代码。因为我们在注释时使用了`||!`而不是`||`，所以该注释是为了包含这段注释的条目而编写的，而不是为了紧随注释之后的条目编写的。在本例中，包含这段注释的条目就是 `src/libs.rs` 文件，也就是包的根文件。也就是说这段注释是描述整个包的。

当我们运行`cargo doc --open`时，这些新添加的注释就会出现在 `my_crate` 文档的首页，显示在包的所有公共条目上方，如图14-2所示。

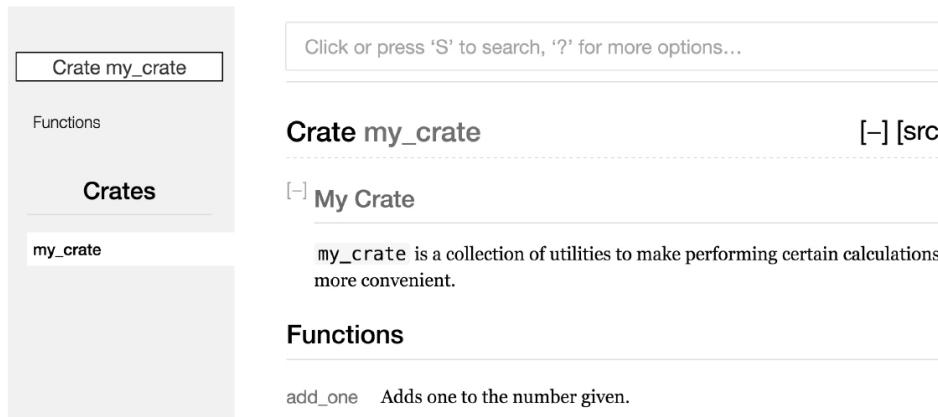


图14-2 my_crate文档的渲染效果中显示了整个包的文档注释

在条目内部的文档注释对于描述包或模块特别有用，通过它们来描述外部条目的整体意图可以帮助用户理解包的组织结构。

使用pub use来导出合适的公共API

在第7章中，我们介绍了如何使用mod关键字来将代码组织为模块、如何使用pub关键字来将条目声明为公共的，以及如何使用use关键字来将条目引入作用域。然而对于用户来讲，这些在开发过程中建立起来的组织结构也许并不是特别友好。你的代码模块可能是一个包含多个层次的树状结构，但当用户想要使用某个较深层次中的类型时就会在查找过程中遇到麻烦。另外，在引入数据时需要输入use `my_crate::some_module:: another_module::UsefulType;`，这比输入简单的use `my_crate:: UsefulType;` 要烦人得多。

在决定发布一个包时，我们必须考虑好如何组织公共API。包的使用者可没有你那样熟悉代码的内部结构，一旦包的层次结构过于复杂，用户就可能会难以找到他们真正需要的部分。

幸运的是，即便代码的内部结构对于用户来讲不是 特别友好，你也不必为了解决问题而重新组织代码。我们可以使用pub use来重新导出部分条目，从而建立一套和你的内部结构不同的对外结构。重新导出操作会取得某个位置的公共条目并将其公开到另一个位置，就好像这个条目原本就定义在新的位置上一样。

例如，假设我们编写了一个对美术概念进行建模的art库。这个库由两个模块组成：其中的kinds模块包含了PrimaryColor和SecondaryColor两个枚举类型，而另一个utils模块则包含了一个mix函数，如示例14-3所示。

src/lib.rs

```
///! # Art
///!
///! 一个用来建模艺术概念的代码库

pub mod kinds {
    /// RYB颜色模型的三原色

    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// RYB模型的调和色

    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// 将两种等量的原色混合生成调和色

    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --略
    }
}
```

示例14-3：将内部条目组织为kinds模块和utils模块的art库

运行cargo doc命令为这个包生成的文档首页如图14-3所示。

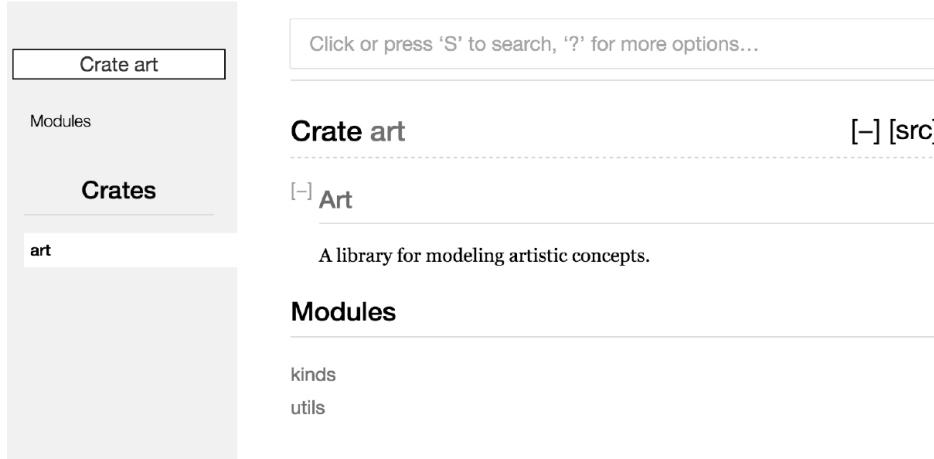


图14-3 列出了kinds模块和utils模块的art文档首页

注意，PrimaryColor类型、SecondaryColor类型及mix函数都没有被显示在首页上，我们必须通过点击kinds和 utils才能进入相应的页面看到它们。

如果用户想要在其他的包中依赖这个代码库，那么他们就需要使用use语句来将art中的条目引入作用域。示例14-4演示了如何使用art包中的PrimaryColor和mix条目。

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：通过指定导出的内部结构来使用art包中的条目

为了使用art包编写出示例14-4中的代码，我们必须搞清楚PrimaryColor位于kinds模块，而mix则位于utils模块。但此处的模块结构其实只是为了方便art包的开发者进行维护，对用户却没有什么太大的用处。这些用于将条目组织到kinds模块和utils模块的内部结构并不能对用户理解art包的使用方式提供任何有用的信息。相反地，art包的模块结构还会使用户产生困惑，因为他们不得不搞清楚功能的

实现路径。另外，由于用户需要在使用use语句时指定完整的模块名称，所以这一结构本身也让代码变得更加冗长了。

为了从公共API中移除内部结构，我们可以修改示例14-3中的art包代码，使用pub use语句将需要公开的条目重新导出到顶层结构中，如示例14-5所示。

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 {
    // --略

    --
}

pub mod utils {
    // --略

    --
}
```

示例14-5：使用pub use语句重新导出一些条目

再次使用cargo doc为art包生成API文档，新的文档首页会列出重新导出的条目及指向它们的链接，如图14-4所示。这就使得PrimaryColor类型、SecondaryColor类型及mix函数更加易于查找了。

此时的用户依然可以看到并使用示例14-3中定义的art包内部结构，就如示例14-4所演示的那样，但他们也可以选择使用示例14-5中的更为方便的结构，如示例14-6所示。

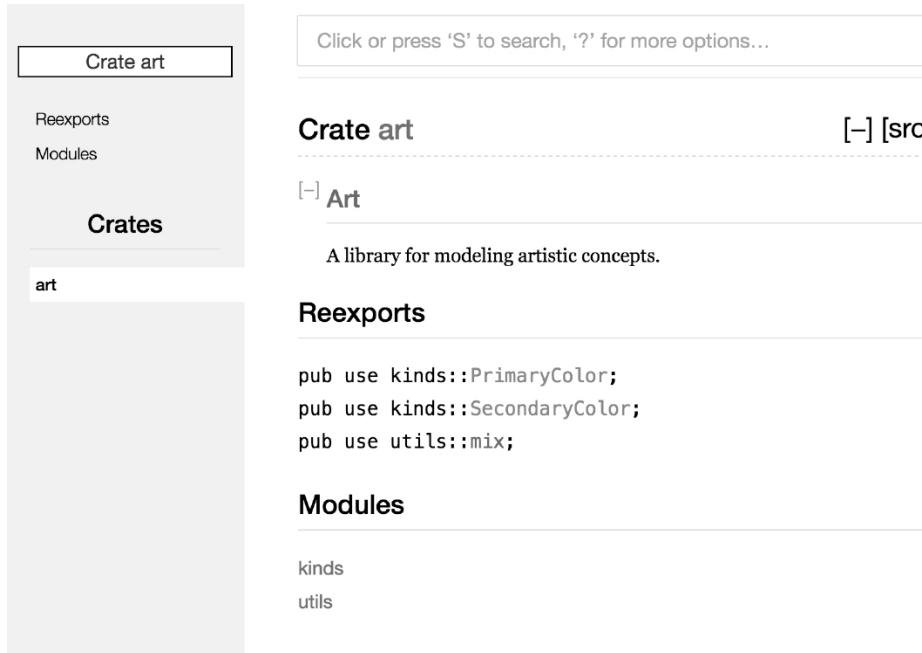


图14-4 art包的文档首页列出了重新导出的条目

src/main.rs

```
use art::PrimaryColor;
use art::mix;

fn main() {
    // --略
}

--
```

示例14-6：这段程序使用了art包中重新导出的条目

当存在较多嵌套模块时，使用pub use将类型重新导出到顶层模块可以显著地改善用户体验。

设计公共API结构这项工作与其说是科学，倒不如说更像是一门艺术。你可以通过不断地迭代实验来找到最适合用户的API风格。使用pub use可以让你在设计内部结构时拥有更大的灵活性，因为它将内部结构与外部表现进行了解耦。你可以浏览一些已经安装过的第三方包，并看一看他们的内部结构是否不同于公共API。

创建crates.io账户

在发布包之前，我们需要在crates.io上注册一个账户并获取一个API令牌（API token）。你可以访问crates.io主页并使用GitHub账户登录来完成注册。（目前，你只能使用GitHub账户来进行登录与注册，未来也许会支持其他认证方式。）登录之后，访问账户设置页面即可获取API令牌。接着，再像下面一样使用API令牌执行cargo login命令：

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

这个命令会让Cargo将你的API令牌存入`~/.cargo/credentials`文件中。请小心地保护令牌，不要将它轻易分享给别人。假如你无意间向他人泄露了令牌，那么你应该立即到crates.io上废除该令牌并重新生成一个新的令牌。

为包添加元数据

你应该已经创建好账户了，假设我们现在有一个正要准备发布的包。在发布之前，我们需要在*Cargo.toml*文件的[package]区域中为这个包添加一些元数据（metadata）。

首先，包需要有一个独一无二的名称。当在本地对包进行开发时，你可以使用任何你喜欢的名称。但是，托管到crates.io平台上的包就必须按照先来先得的规则取名了。一旦某个包的名称被占用，其他包就不能再使用这个名称了。你可以在尝试发布包之前在网站上搜索一下你想要的名字，如果这个名字已经存在了，你就必须重新起一个名字并修改*Cargo.toml*文件[package]区域中的name值，以便使用新的名字进行发布，如下所示：

Cargo.toml

```
[package]
name = "guessing_game"
```

即便包的名称已经是独一无二的了，你仍然会在运行cargo publish来生成包时触发一个警告并导致一个错误：

```
$ cargo publish
```

```
Updating registry `https://github.com/rust-lang/crates.io-index`
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
--略

--
error: api errors: missing or empty metadata fields: description, license.
```

这里出现错误的原因是我们缺少了某些关键信息：一个用于介绍包用途的描述（description），以及一个声明使用条款的许可协议（license）。我们需要在*Cargo.toml* 文件中添加对应的信息来修复这个错误。

因为描述被用在包的搜索结果或对应页面中，所以它通常只有一两句话的长度。对于license字段，你需要填入一个合法的许可协议标识符的值（license identifier value）。Linux基金会的Software Package Data Exchange（ SPDX ）中给出了所有可用的许可协议标识符。假如你想要采用MIT协议的话，那么就需要添加MIT标识符：

Cargo.toml

```
[package]
name = "guessing_game"
license = "MIT"
```

假如你希望使用一个SPDX文档范围之外的许可证，那么就需要将许可协议的文本以文件形式放置在项目目录中，并使用license-file字段指定文件名称，而不要使用license字段来指定。

究竟应该在项目中使用什么样的许可协议已经超出了本书的讨论范畴。许多Rust社区中的开发者会选择在他们的项目中使用与Rust完全一致的许可协议，也就是双许可的MIT OR Apache-2.0。这个例子同时演示了使用OR语法分隔多个许可协议标识符的情形。

拥有了唯一的名称、版本信息、使用cargo new创建包时自动添加的作者信息、描述及许可协议，一切准备就绪的*Cargo.toml* 文件如下所示：

Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

```
edition = "2018"
description = "A fun game where you guess what number the computer has chosen."
license = "MIT OR Apache-2.0"

[dependencies]
```

Cargo的官方文档中列出了其他可添加的元数据，它们可以让你的包更容易被其他人发现与使用。

发布到crates.io

现在，你已经完成了创建账户、存储API令牌、为包选择名称等任务，并指定了必要的元数据，正式发布前的一切准备工作都已经就绪。发布过程会将一个特定版本的包上传到crates.io以供他人使用。

请在发布包的过程中多加小心，因为这一操作是永久性的。已经上传的版本将无法被覆盖，对应的代码也不能被删除。这种行为正是crates.io的一个主要设计目标，它希望能够成为一个永久的代码文档服务器，并保证所有依赖于crates.io的包都能一直被正常构建。如果允许开发者删除已经发布的版本，则根本无法达成这一目的。不过，crates.io对于开发者上传不同版本的包没有数量上的限制。

再次运行cargo publish命令，现在应该能够运行成功了：

```
$ cargo publish

Updating registry `https://github.com/rust-lang/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 [unoptimized + debuginfo] target(s) in 0.19 secs
Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

恭喜你！你已经与Rust社区分享了自己的代码，任何人都可以轻松地将你的包作为依赖来使用了。

发布已有包的新版本

为了在修改代码后发布新的版本，我们需要修改 *Cargo.toml* 文件中的 `version` 字段并重新发布。你应当根据语义化版本规则来基于修改的内容决定下一个合理的版本号，然后执行 `cargo publish` 上传新的版本。

使用 `cargo yank` 命令从 `cargo.io` 上移除版本

尽管你不能移除某一个老版本的包，但我们仍然可以阻止未来的新项目将它们引用为依赖。这在包的版本因为异常问题而损坏时十分有用。对于此类场景，Cargo 支持撤回（yank）某个特定版本。

撤回版本会阻止新的项目来依赖这个版本的包，但对于现存的那些依赖于当前版本的项目则依旧能够下载和依赖它。更具体地说，所有已经产生 *Cargo.lock* 的项目将不会受到撤回操作的影响，而未来所有产生的新 *Cargo.lock* 文件将不会再使用已经撤回的版本。

运行 `cargo yank` 时，指定对应版本号即可撤回指定版本：

```
$ cargo yank --vers 1.0.1
```

通过在命令中添加 `--undo` 参数，你也可以取消撤回操作，从而允许项目再次开始依赖这个版本：

```
$ cargo yank --vers 1.0.1 --undo
```

总之，撤回操作不会删除任何代码。例如，假设你意外地将秘钥发布到了版本中，那么撤回操作并不能帮助你删除这个秘钥，你只能选择立即重置它们。

Cargo工作空间

在第12章中，我们构建了一个既有二进制包，又有代码包的项目。但随着项目规模逐渐增长，你也许会发现自己的代码包越来越臃肿，并想要将它进一步拆分为多个代码包。针对这种需求，Cargo提供了一个叫作工作空间（workspace）的功能，它可以帮助开发者管理多个相互关联且需要协同开发的包。

创建工作空间

工作空间是由共用同一个*Cargo.lock* 和输出目录的一系列包所组成的。现在，让我们使用工作空间来创建一个项目，我们会在这个示例中使用一些刻意简化的代码，并将注意力集中到工作空间的结构本身。组织工作空间的方法有许多种，让我们先从最常见的着手。这个工作空间最终包含一个二进制包和两个代码包，二进制包依赖于另外两个代码包来实现自己的主要功能。其中一个代码包会提供add_one函数，而另一个代码包则会提供add_two函数。这3个包将会共处于同一个工作空间中。让我们先来创建出工作空间的目录：

```
$ mkdir add
```

```
$ cd add
```

随后，在`add` 目录中添加一个用于配置工作空间的*Cargo.toml* 文件，它与我们曾经见过的其他*Cargo.toml* 文件有所不同，它既不包含`[package]` 区域，也不包含之前使用过的那些元数据。这个文件会以`[workspace]` 区域开始，该区域允许我们指定二进制包的路径来为工作空间添加成员。在本例中，这个路径也就是`adder`：

`Cargo.toml`

```
[workspace]

members = [
    "adder",
]
```

接下来，我们将使用cargo new命令在add目录下创建这个adder二进制包：

```
$ cargo new adder
```

```
Created binary (application) `adder` project
```

现在，我们已经可以使用cargo build来构建整个工作空间了。此时，add目录下的文件应该有如下所示的文件结构：

```
├── Cargo.lock
├── Cargo.toml
└── adder
    ├── Cargo.toml
    └── src
        └── main.rs
└── target
```

工作空间在根目录下有一个target目录用来存放所有成员的编译产出物，相对应地，adder包也就没有了自己独立的target目录。即使我们进入adder目录中运行cargo build，编译产出物依然会输出到add/target而不是add/adder/target中。Cargo之所以会将不同的target目录集中到一处是因为工作空间中的包往往是互相依赖的。如果每个包都有自己的target目录，那么它们就不得不在执行各自的构建过程中反复编译工作空间下的其余包。而通过共享一个target目录，不同的包就可以避免这些不必要的重复编译过程。

在工作空间中创建第二个包

现在，让我们来创建工作空间的另一个成员包add-one。打开根目录下的Cargo.toml文件，并向members列表中添加add-one路径：

Cargo.toml

```
[workspace]

members = [
    "adder",
]
```

```
        "add-one",
]
```

接着生成一个名为add-one的新代码包：

```
$ cargo new add-one --lib
```

```
Created library `add-one` project
```

此时，*add* 目录下应该有如下所示的目录和文件：

```
└── Cargo.lock
└── Cargo.toml
└── add-one
    ├── Cargo.toml
    └── src
        └── lib.rs
└── adder
    ├── Cargo.toml
    └── src
        └── main.rs
└── target
```

在*add-one/src/lib.rs* 文件中添加一个add_one函数：

add-one/src/lib.rs

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

创建好新的代码包后，我们可以让二进制包adder依赖于代码包add-one。首先，我们需要在*adder/Cargo.toml* 中添加add-one的路径作为依赖：

adder/Cargo.toml

```
[dependencies]
add-one = { path = "../add-one" }
```

由于Cargo不会主动去假设工作空间中的包会彼此依赖，所以我们必须要显式地指明包与包之间的依赖关系。

接下来，让我们在adder包中使用来自add-one包的add_one函数。打开*adder/src/main.rs* 文件，并在文件顶部使用use语句将新的add-

one包引入作用域。随后修改main函数来调用add_one函数，如示例14-7所示。

adder/src/main.rs

```
use add_one;

fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}
```

示例14-7：在adder包中使用add-one代码包

在add 根目录下运行cargo build来构建整个工作空间：

```
$ cargo build
```

```
Compiling add-one v0.1.0 (file:///projects/add/add-one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs
```

为了在add 根目录下运行二进制包，我们需要在调用cargo run时通过-p参数来指定需要运行的包名：

```
$ cargo run -p adder
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

上面的命令运行了adder/src/main.rs 中的代码，而这段代码则依赖了add-one包。

在工作空间中依赖外部包

需要注意的是，整个工作空间只在根目录下有一个Cargo.lock 文件，而不是在每个包的目录下都有一个Cargo.lock 文件。这一规则确保了所有的内部包都会使用完全相同的依赖版本。假设我们将rand包同时添加到了adder/Cargo.toml 与 add-one/Cargo.toml 文件中，那么Cargo会将这两个依赖解析为同一版本的rand包，并将此信息记录在唯一的Cargo.lock 文件中。确保工作空间中所有的包使用相同的依赖意味着这些包将会是彼此兼容的。让我们在add-one/Cargo.toml 文件

的[dependencies] 区域中加入rand包，以便可以在add-one包中使用rand包内的功能：

add-one/Cargo.toml

```
[dependencies]
rand = "0.3.14"
```

接着在 *add-one/src/lib.rs* 文件中添加use rand;，并在 *add* 目录下运行cargo build来构建整个工作空间。此时，Cargo就会引入并编译rand包：

```
$ cargo build

    Updating registry
`https://github.com/rust-lang/crates.io-index`
  Downloading rand v0.3.14
  --略

-- 
  Compiling rand v0.3.14
  Compiling add-one v0.1.0 (file:///projects/add/add-one)
  Compiling adder v0.1.0 (file:///projects/add/adder)
  Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

现在，根目录下的 *Cargo.lock* 文件包含了add-one依赖于rand的记录。但需要注意的是，虽然当前的工作空间已经引用了rand，但工作空间内其余的包依然不能直接使用它，除非我们将rand添加到这些包对应的 *Cargo.toml* 中去。例如，在 *adder/src/main.rs* 文件中为 *adder* 包添加use rand; 语句将导致编译时错误：

```
$ cargo build

  Compiling adder v0.1.0 (file:///projects/add/adder)
error: use of unstable library feature 'rand': use `rand` from crates.io (see
issue #27703)
--> adder/src/main.rs:1:1
  |
1 | use rand;
```

为了修复这个问题，我们只需要在 *adder* 包的 *Cargo.toml* 文件中添加rand依赖即可。再次构建adder包时，rand就会被添加至 *Cargo.lock* 中adder的依赖列表中了。但是，构建时不会重复下载并编译rand包，因为Cargo保证了工作空间中使用的所有的rand包都是同

一个版本。统一的rand版本不仅避免了多余的拷贝从而节约了磁盘空间，也确保了工作空间中的包是彼此兼容的。

为工作空间增加测试

接下来进行另一处改进，让我们来为add_one包的add_one::add_one函数添加一个测试：

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add根目录下执行cargo test命令：

```
$ cargo test

Compiling add-one v0.1.0 (file:///projects/add/add-one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
Running target/debug/deps/add_one-f0253159197f7841

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

第一段输出表明add-one包中的it_works测试通过了。第二段输出表明指令没有在adder包中发现可用的测试，第三段输出则表明指令没有在add-one包中发现可用的文档测试。在这样的结构中调用cargo test会一次性执行工作空间中所有包的测试。

我们同样可以在工作空间根目录下，使用参数-p及指定的包名称来运行某一个特定包的测试：

```
$ cargo test -p add-one

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

这段新的输出消息意味着只有add-one包的测试得到了执行，而adder包的测试则没有被执行。

当你想要将工作空间中的各个包发布到crates.io上时，你必须要将它们分别发布。cargo publish命令并没有提供类似于--all或-p之类的标记，你必须手动切换到每个包的目录，并对每个包分别执行cargo publish来完成发布任务。

作为练习，请试着模仿我们添加add-one包的方式来将add-two包添加到这个工作空间中。

你可以在项目规模逐渐增长时考虑使用工作空间：独立短小的组件要比繁冗长的代码更容易理解一些。另外，当多个包经常需要同时修改时，将它们放于同一工作空间下也有助于协调同步。

使用cargo install从crates.io上安装可执行程序

cargo install命令使我们可以在自己的计算机设备中安装和使用二进制包。但需要注意的是，它不能被用来替换操作系统的包管理器。这一命令只是为了便于Rust开发者们获得其他人在crates.io上分享的工具。另外，你只能安装那些带有二进制目标（binary target）的包。二进制目标其实就是一段可执行的程序，它们只有在包内存在 `src/main.rs` 或其他被指定为二进制入口的文件时才会生成。这个概念和库目标（library target）相对应，库目标本身无法单独执行但非常适合被包含在其他程序中。大部分的包都会在 `README` 文件中说明自己是否拥有库目标，是否拥有二进制目标，又或者是否两者皆有。

所有通过cargo install命令安装的二进制文件都会被存储在Rust安装根目录下的 `bin` 文件夹中。假如你在安装Rust的过程中使用了rustup且没有指定任何自定义配置，那么 `bin` 的路径就是 `$HOME/.cargo/bin`。为了能够直接运行cargo install安装的工具程序，我们需要将该路径添加到环境变量\$PATH中。

例如，我们在第12章曾经提到过一个用Rust实现的grep工具ripgrep（用于搜索文件的工具）。你可以运行如下所示的命令来安装ripgrep：

```
$ cargo install ripgrep

Updating registry `https://github.com/rust-lang/crates.io-index`
Downloaded ripgrep v0.3.2
--略

--
Compiling ripgrep v0.3.2
Finished release [optimized + debuginfo] target(s) in 97.91 secs
Installing ~/.cargo/bin/rg
```

输出结果的最后一行显示了二进制文件的安装路径和名称，本例中的`ripgrep`被命名为了`rg`。只要你像上面提到过的那样将安装目录加入到了`$PATH`中，就可以接着运行`rg - help`，来开始使用一个更快、更具Rust风格的文件搜索工具。

使用自定义命令扩展Cargo的功能

Cargo允许我们添加子命令来扩展它的功能而无须修改Cargo本身。只要你的\$PATH路径中存在二进制文件cargo-something，就可以通过运行cargo something来运行该二进制文件，就好像它是Cargo的子命令一样。运行cargo --list可以列出所有与此类似的自定义命令。借助于这一设计，我们可以使用cargo install来安装扩展，并把这些扩展视作内建的Cargo命令来运行。

总结

因为有了Cargo和crates. io共同构建出的代码分享机制，Rust的生态系统才能够应对许多不同类型的任务。虽然Rust的标准库小巧且稳定，但是我们依然可以借助包机制来轻松地分享与使用代码，并随着时间不断地演化进步而不必拘泥于语言本身的更新频率。请勇敢地将那些对你有用的代码分享到crates. io上吧，因为它们同样会帮助到许许多多和你一样的开发者！

第15章

智能指针



指针（pointer）是一个通用概念，它指代那些包含内存地址的变量。这个地址被用于索引，或者说用于“指向”内存中的其他数据。Rust中最常用的指针就是你在第4章学习过的引用。引用是用`&`符号表示的，会借用它所指向的值。引用除了指向数据外没有任何其他功能，也没有任何额外的开销，它是Rust中最为常见的一种指针。

而智能指针（smart pointer）则是一些数据结构，它们的行为类似于指针但拥有额外的元数据和附加功能。智能指针的概念并不是Rust所独有的，它最初起源于C++并被广泛地应用在多种语言中。Rust标准库中不同的智能指针提供了比引用更为强大的功能。本章将会介绍的是引用计数（reference counting）智能指针类型。这种指针会通过记录所有者的数量来使一份数据被多个所有者同时持有，并在没有任何所有者时自动清理数据。

在拥有所有权和借用概念的Rust中，引用和智能指针之间还有另外一个差别：引用是只借用数据的指针；而与之相反地，大多数智能指针本身就拥有它们指向的数据。

实际上，我们已经在本书中接触过好几种不同的智能指针了，例如第8章中的String与Vec<T>。尽管我们没有刻意地提及智能指针这个称呼，但这两种类型都可以被算作智能指针，因为它们都拥有一片内存区域并允许用户对其进行操作。它们还拥有元数据（例如容量等），并提供额外的功能或保障（例如String会保障其中的数据必定是合法的UTF-8编码）。

我们通常会使用结构体来实现智能指针，但区别于一般结构体的地方在于它们会实现Deref与Drop这两个trait。Deref trait使得智能指针结构体的实例拥有与引用一致的行为，它使你可以编写出能够同时用于引用和智能指针的代码。Drop trait则使你可以自定义智能指针离开作用域时运行的代码。在本章中，我们会依次讨论这两个trait，并通过演示来说明它们对于智能指针的重要性。

由于智能指针作为一种设计模式被相当频繁地应用到了Rust中，所以我们无法在本书中涉及所有现存的智能指针类型。事实上，许多代码库都会提供它们自己的智能指针，你也可以选择自己编写满足特定用途的智能指针类型。接下来，我们会将讨论的重点集中到标准库中最为常见的那些智能指针上：

- Box<T>，可用于在堆上分配值。
- Rc<T>，允许多重所有权的引用计数类型。
- Ref<T>和RefMut<T>，可以通过RefCell<T>访问，是一种可以在运行时而不是编译时执行借用规则的类型。

另外，我们会在本章介绍内部可变性（interior mutability）模式，使用了这一模式的不可变类型会暴露出能够改变自己内部值的API。我们还会讨论循环引用导致内存泄漏的原因，并研究如何来规避类似的问题。

让我们开始吧！

使用Box<T>在堆上分配数据

装箱（box）是最为简单直接的一种智能指针，它的类型被写作Box<T>。装箱使我们可以将数据存储在堆上，并在栈中保留一个指向堆数据的指针。你可以回顾第4章来复习一下栈与堆的区别。

除了将它们的数据存储在堆上而不是栈上，装箱没有其他任何的性能开销。当然，它们也无法提供太多的额外功能。装箱常常被用于下面的场景中：

- 当你拥有一个无法在编译时确定大小的类型，但又想要在一个要求固定尺寸的上下文环境中使用这个类型的值时。
- 当你需要传递大量数据的所有权，但又不希望产生大量数据的复制行为时。
- 当你希望拥有一个实现了指定trait的类型值，但又不关心具体的类型时。

我们会在“使用装箱定义递归类型”一节中演示第一种场景的应用示例。在第二种场景中，转移大量数据的所有权可能会花费较多的时间，因为这些数据需要在栈上进行逐一复制。为了提高性能，你可以借助装箱将这些数据存储到堆上。通过这种方式，我们只需要在转移所有权时复制指针本身即可，而不必复制它指向的全部堆数据。第三种场景也被称作trait对象（trait object），我们会在第17章的“使用trait对象来存储不同类型的值”一节来详细讨论它。本节介绍的内容将在第17章再次用到！

使用Box<T>在堆上存储数据

在开始讨论Box<T>的使用场景前，先让我们来了解一下它的语法及如何与存储在其中的值进行交互。

示例15-1展示了如何使用装箱在堆上存储一个i32值。

src/main.rs

```
fn main() {  
    let b = Box::new(5);  
    println!("b = {}", b);  
}
```

示例15-1：使用装箱在堆上存储一个i32值

我们在这个示例中定义了一个持有Box的值的变量b，它指向了堆上的值5。这段程序会在运行时输出b = 5。代码中用来访问装箱数据的语法与访问栈数据的语法非常类似。另外，和其他任何拥有所有权的值一样，装箱会在离开自己的作用域时（也就是b到达main函数的结尾时）被释放。装箱被释放的东西除了有存储在栈上的指针，还有它指向的那些堆数据。

将单一值存放在堆上并没有太大的用处，因此你也不会经常这样使用装箱。在大部分情况下，我们都可以将类似的单个i32值默认放置在栈上。现在，让我们再来看一下另一个案例，在该场景下我们只有使用装箱才能定义出期望的类型。

使用装箱定义递归类型

Rust必须在编译时知道每一种类型占据的空间大小，但有一种被称作递归（recursive）的类型却无法在编译时被确定具体大小。递归类型的值可以在自身中存储另一个相同类型的值，因为这种嵌套在理论上可以无穷无尽地进行下去，所以Rust根本无法计算出一个递归类型需要的具体空间大小。但是，装箱有一个固定的大小，我们只需要在递归类型的定义中使用装箱便可以创建递归类型了。

下面来看一个递归类型的例子，一个在函数式编程语言中相当常见的数据类型：链接列表（cons list）。除了递归部分，我们将使用

较为直接的方式来定义这个链接列表类型。本例中用到的概念对于设计一些更为复杂的递归类型也是同样适用的。

有关链接列表的更多信息

链接列表是一种来自Lisp编程语言与其方言的数据结构。在Lisp中，`cons`函数（也就是构造函数的缩写）会将两个参数组成一个二元组，而这个元组通常由一个值与另一个二元组组成。通过这种不断嵌套元组的形式可以最终组成一个列表。

函数式编程语言中甚至有一个用来描述`cons`函数的通用术语：“将 x 链接至 y ”，它意味着这个函数会将元素 x 链接到容器 y 来构造出一个新的容器实例。

链接列表的每一项都包含了两个元素：当前项的值及下一项。列表中的最后一项是一个被称作`Nil`且不包含下一项的特殊值。我们通过反复调用`cons`函数来生成链接列表，并使用规范名称`Nil`来作为列表的终止标记。注意，这不同于在第6章讨论过的“`null`”概念，`Nil`并不是一个无效或缺失的值。

尽管你会在函数式编程语言中非常高频率地用到链接列表，但它在Rust中其实并不常见。当你需要在Rust中持有一系列的元素时，`Vec<T>`在大部分情况下都会是一个更好的选择。确实有一些比链接列表更具有实用价值的递归数据类型，但它们的具体实现细节也更加复杂。为了简单起见，就让我们从链接列表着手，并将注意力集中到如何使用装箱来定义递归数据类型上。

示例15-2尝试使用枚举来定义一个链接列表。注意，这段代码暂时无法通过编译，因为我们不能确定`List`类型的大小。

src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

示例15-2：尝试使用枚举来表达一个持有`i32`值的链接列表数据类型

注意

作为示例，上面的代码仅仅实现了一个可以持有i32值的链接列表。但是，实际上我们可以使用在第10章讨论过的泛型来实现这一数据结构，并使它可以存储任意类型的值。

示例15-3演示了使用这个List类型来存储列表1, 2, 3的方法。

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枚举存储列表1, 2, 3

第一个Cons变体包含了1和另外一个List值。这个List值作为另外一个Cons变体包含了2和另外一个List值。这个List依然是一个Cons变体，它包含了3与一个特殊的List值，也就是最终的非递归变体Nil，它代表了列表的结束。

如果你试图编译示例15-3中的代码，则会观察到示例15-4中出现的错误提示信息。

```
error[E0072]: recursive type `List` has infinite size
--> src/main.rs:1:1
|
1 | enum List {
| ^^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
|           ----- recursive without indirection
|
|= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

示例15-4：试图定义带有递归的枚举类型时发生的错误

上面的错误提示信息指出这个类型“拥有无限大小”，这是因为我们在定义List时引入了一个递归的变体，它直接持有了另一个相同类型的值。这意味着Rust无法计算出存储一个List值需要消耗多大的

空间。为了更好地理解这一问题，让我们先来看一看Rust会如何计算非递归类型所需占用的存储空间大小。

计算一个非递归类型的大小

回忆一下我们在第6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),
}
```

为了计算出Message值需要多大的存储空间，Rust会遍历枚举中的每一个成员来找到需要最大空间的那个变体。在Rust眼中，Message::Quit不需要占用任何空间，Message::Move需要两个存储i32值的空间，以此类推。因为在每个时间点只会有一个变体存在，所以Message值需要的空间大小也就是能够存储得下最大变体的空间大小。

与此类似，让我们模拟一下Rust在确定递归类型大小时发生的运算过程。以示例15-2中的List为例，编译器会首先检查Cons变体，并发现它持有一个i32类型的值及另外一个List类型的值。因此，Cons变体需要的空间也就等于一个i32值的大小加上一个List值的大小。为了确定List值所需的空间大小，编译器又会从Cons开始遍历其下的所有变体，这样的检查过程将永无穷尽地进行下去，如图15-1所示。

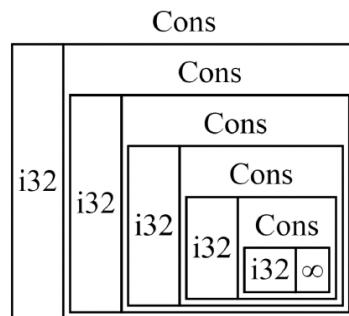


图15-1 一个包含无限多Cons变体的无穷List

使用Box<T>将递归类型的大小固定下来

虽然Rust无法推断出递归类型需要的空间大小，但在示例15-4的错误提示信息中，编译器也给出了一条有用的建议：

```
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
make `List` representable
```

建议中的indirection（间接）意味着，我们应该改变数据结构来存储指向这个值的指针，而不是直接地存储这个值。

因为Box<T>是一个指针，所以Rust总是可以确定一个Box<T>的具体大小。指针的大小总是恒定的，它不会因为指向数据的大小而产生变化。这也意味着我们可以在Cons变体中存放一个Box<T>而不是直接存放另外一个List值。而Box<T>则会指向下一个List并存储在堆上，而不是直接存放在Cons变体中。理论上讲，我们仍然拥有一个“持有”其他列表的列表，但现在的实现更像是一项挨着一项，而不是一项包含另一项。

修改示例15-2中关于List枚举的定义及示例15-3中有关List的用法，如示例15-5所示。现在，代码可以通过编译了。

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：为了拥有固定大小而使用Box<T>的List定义

新的Cons变体需要一部分存储i32的空间和一部分存储装箱指针数据的空间。另外，由于Nil变体没有存储任何值，所以它需要的空间比Cons变体小。现在，我们知道任意的List值都只需要占用一个i32值加上一个装箱指针的大小。通过使用装箱，我们打破了无限递归的过程，进而使编译器可以计算出存储一个List值需要多大的空间。现在，Cons变体的结构如图15-2所示。

Cons

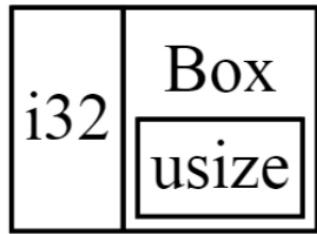


图15-2 由于Cons持有Box，所以现在的List不再具有无限大小了

与我们即将学习的其他智能指针相比，除了间接访问内存和堆分配，装箱没有提供其他任何特殊功能，也自然没有这些特殊功能附带的性能开销。因此，装箱正好能够被用在类似于链接列表这类仅仅需要间接访问的场景中。我们会在第17章见到有关装箱的更多应用实例。

Box<T>属于智能指针的一种，因为它实现了Deref trait，并允许我们将Box<T>的值当作引用来对待。当一个Box<T>值离开作用域时，因为它实现了Drop trait，所以Box<T>指向的堆数据会自动地被清理释放掉。这两个trait也同样被用到了随后讨论的其他智能指针中，它们对实现某些功能起到了至关重要的作用。因此，让我们先来更深入地了解一下这两个trait。

通过Deref trait将智能指针视作常规引用

实现Deref trait使我们可以自定义解引用运算符（dereference operator）*的行为（这一符号也同时被用作乘法运算符和通配符）。通过实现Deref，我们可以将智能指针视作常规引用来进行处理。这也就意味着，原本用于处理引用的代码可以不加修改地用于处理智能指针。

首先，让我们来看一看解引用运算符作用于常规引用时的效果。接着，我们会尝试编写一个与Box<T>拥有类似行为的自定义类型，并进一步分析为什么无法对这个自定义类型进行解引用操作。然后，我们还会学习如何通过实现Deref trait来使智能指针拥有类似于引用的行为。最后，我们将讨论Rust的解引用转换（deref coercion）功能，并观察它会如何影响我们使用引用或智能指针。

注意

本节将要构建的MyBox<T>并不会将数据存储在堆上，它与实际的Box<T>有着显著的差异。这是因为我们希望在这个示例中专注于讨论有关Deref的细节，并模拟类似于指针的行为，而至于将数据存储在何处则没有那么重要。

使用解引用运算符跳转到指针指向的值

常规引用就是一种类型的指针。你可以将指针形象地理解为一个箭头，它会指向存储在别处的某个值。我们在示例15-6中创建了一个i32值的引用，并接着通过解引用运算符跟踪该数据的引用。

src/main.rs

```
fn main() {
    ❶ let x = 5;
    ❷ let y = &x;

    ❸ assert_eq!(5, x);
    ❹ assert_eq!(5, *y);
}
```

示例15-6：使用解引用运算符跟踪i32值的引用

这段代码中的变量x存储了一个i32值5❶，并在变量y中存储了x的引用❷。我们可以直接断言，这里的x与5相等❸。但是，当你想要断言变量y中的值时，我们就必须使用*y来跟踪引用并跳转到它指向的值（也就是解引用）❹。在对y进行了解引用后，我们才可以得到y指向的整数值，并将它与5进行比较。

假如你将上面的代码改写为assert_eq! (5, y);，则会触发编译错误：

```
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 `std::cmp::PartialEq<&<integer>>` is not implemented for
`<integer>`
```

由于数值和引用是两种不同的类型，所以你不能直接比较这两者。我们必须使用解引用运算符来跳转到引用指向的值。

把Box<T>当成引用来操作

我们可以使用Box<T>来代替示例15-6中的引用，此时的解引用运算符能够正常工作，如示例15-7所示。

src/main.rs

```
fn main() {
    let x = 5;
    ❶ let y = Box::new(x);

    ❸ assert_eq!(5, x);
    ❹ assert_eq!(5, *y);
}
```

示例15-7：对Box<i32>进行解引用

示例15-7与示例15-6的唯一区别就在于我们将y设置为了一个指向x值的装箱指针，而不是一个指向x值的引用❶。在最后的断言中❷，我们依然可以使用解引用运算符来跟踪装箱指针，正如我们跟踪引用一样。接下来，我们会实现一个自定义的装箱类型，并借此来研究为什么Box<T>能够进行解引用操作。

定义我们自己的智能指针

让我们来构建一个类似于Box<T>类型的智能指针，并体会默认行为下智能指针与常规引用之间的差异。接着，我们再来学习如何使它可以使用解引用运算符。

Box<T>类型最终被定义为一个拥有单元素的元组结构体，示例15-8以相同的方式定义了MyBox<T>类型。除此之外，我们还定义了一个与Box<T>的new函数作用类似的new函数。

src/main.rs

```
❶struct MyBox<T>(T);

impl<T> MyBox<T> {
   ❷ fn new(x: T) -> MyBox<T> {
       ❸ MyBox(x)
    }
}
```

示例15-8：定义一个MyBox<T>类型

上面的代码定义了一个名为MyBox的结构体。结构体的定义中附带了泛型参数T❶，因为我们希望它能够存储任意类型的值。MyBox是一个拥有T类型单元素的元组结构体。它的关联函数MyBox::new接收一个T类型的参数❷，并返回一个存储有传入值的MyBox实例作为结果❸。

让我们试着将示例15-7中的main函数添加至示例15-8中，并使用新定义的MyBox<T>类型替换Box<T>。示例15-9中的代码暂时无法通过编译，因为Rust还不知道应该如何去解引用MyBox。

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：以类似于引用和Box<T>的方法来使用MyBox<T>

编译后出现如下所示的错误：

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:14:19
 |
14 |     assert_eq!(5, *y);
   |     ^^^
```

因为我们没有为MyBox<T>类型实现解引用功能，所以这个解引用操作还无法生效。为了使用*完成解引用操作，我们需要实现Deref trait。

通过实现Deref trait来将类型视作引用

正如在第10章讨论的那样，为了实现某个trait，我们需要为该trait的方法指定具体的行为。而标准库中的Deref trait则要求我们实现一个deref方法，该方法会借用self并返回一个指向内部数据的引用。示例15-10为MyBox实现了Deref。

src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    ❶ type Target = T;

    fn deref(&self) -> &T {
        ❷ &self.0
    }
}
```

示例15-10：为MyBox<T>实现Deref

type Target = T; 语法❶定义了Deref trait的一个关联类型。关联类型是一种稍微有些不同的泛型参数定义方式，我们会在第19章对这一特性进行深入的讨论，现在先忽略它就好。

我们在deref的方法体中填入了`&self.0`，这意味着deref会返回一个指向值的引用，进而允许调用者通过`*`运算符访问值❷。示例15-9的main函数中对`MyBox<T>`值调用`*`的操作，现在可以正常通过编译并断言成功了！

在没有Deref trait的情形下，编译器只能对`&`形式的常规引用执行解引用操作。deref方法使编译器可以从任何实现了Deref的类型中获取值，并能够调用deref方法来获得一个可以进行解引用操作的引用。

我们在示例15-9中编写的`*y`会被Rust隐式地展开为：

```
* (y.deref())
```

Rust使用`*`运算符来替代deref方法和另外一个朴素的解引用操作，这样我们就不用考虑是否需要调用deref方法了。这一特性使我们可以用完全相同的方式编写代码来处理常规引用及实现了Deref trait的类型。

所有权系统决定了deref方法需要返回一个引用，而`*(y.deref())`的最外层依然需要一个朴素的解引用操作。假设deref方法直接返回了值而不是指向值的引用，那么这个值就会被移出`self`。在大多数使用解引用运算符的场景下，我们并不希望获得`MyBox<T>`内部值的所有权。

需要注意的是，这种将`*`运算符替换为deref方法和另外一个朴素`*`运算符的过程，对代码中的每个`*`都只会进行一次。因为`*`运算符的替换不会无穷尽地递归下去，所以我们才能在代码中得到`i32`类型的值，并与示例15-9中`assert_eq!`的5相匹配。

函数和方法的隐式解引用转换

解引用转换（deref coercion）是Rust为函数和方法的参数提供的一种便捷特性。当某个类型T实现了Deref trait时，它能够将T的引用转换为T经过Deref操作后生成的引用。当我们将来某个特定类型的值引用作为参数传递给函数或方法，但传入的类型与参数类型不一致时，解引用转换就会自动发生。编译器会插入一系列的deref方法调用将我们提供的类型转换为参数所需的类型。

Rust通过实现解引用转换功能，使程序员在调用函数或方法时无须多次显式地使用`&`和`*`运算符来进行引用和解引用操作。这一特性还使我们可以更多地编写出能够同时作用于常规引用和智能指针的代码。

为了观察解引用转换的实际效果，让我们使用示例15-8中的`MyBox<T>`类型及示例15-10中的Deref实现来进行演示。示例15-11展示了一个接收字符串切片作为参数的函数定义。

src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

示例15-11：接收一个类型为`&str`的参数`name`的`hello`函数

借助于解引用转换特性，我们既可以将字符串切片作为参数传入`hello`函数，例如`hello("Rust")`，也可以将`MyBox<String>`值的引用传入`hello`函数，如示例15-12所示。

src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

示例15-12：解引用转换特性使我们可以将`MyBox<String>`值的引用传入`hello`函数

我们在上面的代码中将参数`&m`传入了`hello`函数，而`&m`正是一个指向`MyBox<String>`值的引用。因为我们在示例15-10中为`MyBox<T>`实现了Deref trait，所以Rust可以通过调用`deref`来将`&MyBox<String>`转

换为`&String`。因为标准库为`String`提供的`Deref`实现会返回字符串切片（你可以在`Deref`的API文档中看到这一信息），所以Rust可以继续调用`deref`来将`&String`转换为`&str`，并最终与`hello`函数的定义相匹配。

如果Rust没有解引用转换功能，那么为了将`&MyBox<String>`类型的值传入`hello`函数，就不得不使用示例15-13中的代码来代替示例15-12中的代码。

src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m) [...]);
}
```

示例15-13：如果Rust没有解引用转换功能，就必须编写这样的代码

代码中的`(*m)`首先将`MyBox<String>`进行解引用得到`String`，然后，通过`&`和`[...]`来获取包含整个`String`的字符串切片以便匹配`hello`函数的签名。缺少了解引用转换的代码会充斥着这类符号，从而变得更加难以阅读、编写和理解。解引用转换使Rust可以为我们自动处理这些转换过程。

只要代码涉及的类型实现了`Deref trait`，Rust就会自动分析类型并不断尝试插入`Deref::deref`来获得与参数类型匹配的引用。因为这一分析过程会在编译时完成，所以解引用转换不会在运行时产生任何额外的性能开销！

解引用转换与可变性

使用`Deref trait`能够重载不可变引用的`*`运算符。与之类似，使用`DerefMut trait`能够重载可变引用的`*`运算符。

Rust会在类型与trait满足下面3种情形时执行解引用转换：

- 当`T: Deref<Target=U>`时，允许`&T`转换为`&U`。

- 当T: DerefMut<Target=U>时，允许&mut T转换为&mut U。
- 当T: Deref<Target=U>时，允许&mut T转换为&U。

前两种情形除可变性之外是完全相同的。其中，情形一意味着，如果T实现了类型U的Deref trait，那么&T就可以被直接转换为&U。情形二意味着，同样的解引用转换过程会作用于可变引用。

情形三则有些微妙：Rust会将一个可变引用自动地转换为一个不可变引用。但这个过程绝对不会逆转，也就是说不可变引用永远不可能转换为可变引用。因为按照借用规则，如果存在一个可变引用，那么它就必须是唯一的引用（否则程序将无法通过编译）。将一个可变引用转换为不可变引用肯定不会破坏借用规则，但将一个不可变引用转换为可变引用则要求这个引用必须是唯一的，而借用规则无法保证这一点。因此，Rust无法将不可变引用转换为可变引用视作一个合理的操作。

借助Drop trait在清理时运行代码

另一个对智能指针十分重要的trait就是Drop，它允许我们在变量离开作用域时执行某些自定义操作。你可以为任意类型实现一个Drop trait，它常常被用来释放诸如文件、网络连接等资源。我们之所以选择在智能指针的上下文中介绍Drop，是因为几乎每一种智能指针的实现都会用到这一trait。例如，Box<T>通过自定义Drop来释放装箱指针指向的堆内存空间。

在某些语言中，开发者必须在使用完智能指针后手动地释放内存或资源。一旦他们忘记这件事情，系统就可能会出现资源泄漏并最终引发过载崩溃。而在Rust中，我们可以为值指定离开作用域时需要执行的代码，而编译器则会自动将这些代码插入到合适的地方。因此，你不用在程序中众多的实例销毁处放置清理代码，也不会产生任何的资源泄漏。

我们可以通过实现Drop trait来指定值离开作用域时需要运行的代码。Drop trait要求实现一个接收self可变引用作为参数的drop函数。为了观察Rust在何时会调用drop，让我们先来实现一个带有println!输出的drop函数。

示例15-14定义了一个CustomSmartPointer结构体，它唯一的功能是在离开作用域时打印一行文字：Dropping CustomSmartPointer!。通过这个示例，我们可以观察到Rust调用drop函数的时间。

src/main.rs

```
struct CustomSmartPointer {
    data: String,
}

①impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        ② println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
```

```

    }
}

fn main() {
③ let c = CustomSmartPointer { data: String::from("my stuff") };
④ let d = CustomSmartPointer { data: String::from("other stuff") };
⑤ println!("CustomSmartPointers created.");
⑥}

```

示例15-14：为CustomSmartPointer结构体实现存放清理代码的Drop trait

这段代码没有显式地将Drop trait引入作用域，因为它已经被包含在了预导入模块中。我们为CustomSmartPointer结构体实现了Drop trait①，并在drop方法中调用了println! ②，这些打印出来的文本可以用来展示Rust调用drop函数的时间。实际上，任何你想要在类型实例离开作用域时运行的逻辑都可以放在drop函数体内。

我们在main函数中创建了两个CustomSmartPointer实例③④并打印了一行文本：CustomSmartPointers created. ⑤。在main函数的结尾处⑥，当两个CustomSmartPointer实例离开作用域时，Rust会自动调用我们在drop方法中放置的代码②来打印出最终的信息，而无须显式地调用drop方法。

运行这段程序可以看到如下所示的输出结果：

```

CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!

```

Rust在实例离开作用域时自动调用了我们编写的drop代码。因为变量的丢弃顺序与创建顺序相反，所以d在c之前被丢弃。这个实例应该能够较为直观地演示出drop方法的运行机制；当然在实际的开发中，你通常需要为指定类型执行清理逻辑而不是打印文本。

使用std::mem::drop提前丢弃值

遗憾的是，我们无法直接禁用自动drop功能。当然，禁用drop通常也没有任何必要，因为Drop trait存在的意义就是为了完成自动释放的逻辑。不过，我们倒是常常会碰到需要提前清理一个值的情形。

其中一个例子就是使用智能指针来管理锁时：你也许会希望强制运行drop方法来提前释放锁，从而允许同一作用域内的其他代码来获取它。Rust并不允许我们手动调用Drop trait的drop方法；但是，你可以调用标准库中的std::mem::drop函数来提前清理某个值。

假如你修改了示例15-14中的main函数，以便手动调用Drop trait的drop方法，如示例15-15所示，那么这段代码就会在编译时出现错误。

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：试图调用Drop trait的drop方法来提前清理一个值

编译这段代码会产生如下所示的错误：

```
error[E0040]: explicit use of destructor method
--> src/main.rs:14:7
|
14 |     c.drop();
|     ^^^^^ explicit destructor calls not allowed
```

这条错误提示信息表明我们不能显式地调用drop。信息中使用了一个专有名词析构函数（destructor），这个通用的编程概念被用来指代可以清理实例的函数，它与创建实例的构造函数（constructor）相对应。而Rust中的drop函数正是这样一个析构函数。

因为Rust已经在main函数结尾的地方自动调用了drop，所以它不允许我们再次显式地调用drop。这种行为会试图对同一个值清理两次而导致重复释放（double free）错误。

我们既不能在一个值离开作用域时禁止自动插入drop，也不能显式地调用drop方法。因此，如果必须要提前清理一个值，我们就需要使用std::mem::drop函数。

std::mem::drop函数不同于Drop trait中的drop方法。我们需要手动调用这个函数，并将需要提前丢弃的值作为参数传入。因为该函

数被放置在了预导入模块中，所以我们可以修改示例15-15中的main函数来直接调用drop函数，如示例15-16所示。

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：在值离开作用域前调用std::mem::drop来显式地丢弃它

运行这段代码会输出如下所示的内容：

```
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

文本消息 Dropping CustomSmartPointer with data `some data`! 被打印在了 CustomSmartPointer created. 和 CustomSmartPointer dropped before the end of main. 之间，这说明drop方法的确被调用了，c在预期的位置被丢弃了。

你可以使用不同的方式来实现Drop trait，从而使清理工作更为方便和安全。你甚至可以使用它来实现自定义的内存分配器！借助Drop trait和Rust的所有权系统，开发者可以将清理现场的工作完全交由Rust执行，它会自动处理好这类琐碎的任务。

我们也无须担心正在使用的值会被意外地清理掉：所有权系统会保证所有的引用有效，而drop只会在确定不再使用这个值时被调用一次。

现在，我们已经学习了Box<T>和智能指针的部分特点，接下来让我们来看一看标准库中提供的一些其他智能指针。

基于引用计数的智能指针Rc<T>

所有权在大多数情况下都是清晰的：对于一个给定的值，你可以准确地判断出哪个变量拥有它。但在某些场景中，单个值也可能同时被多个所有者持有。例如，在图数据结构中，多个边可能会指向相同的节点，而这个节点从概念上来讲就同时属于所有指向它的边。一个节点只要在任意指向它的边还存在时就不应该被清理掉。

Rust提供了一个名为Rc<T>的类型来支持多重所有权，它名称中的Rc是Reference counting（引用计数）的缩写。Rc<T>类型的实例会在内部维护一个用于记录值引用次数的计数器，从而确认这个值是否仍在使用。如果对一个值的引用次数为零，那么就意味着这个值可以被安全地清理掉，而不会触发引用失效的问题。

你可以将Rc<T>想象成客厅中的电视。在第一个人进入客厅并打开电视后，其余所有进入的人就都可以直接观看电视。电视会一直保持开启状态并在最后一个人离开时关闭，因为我们不再需要使用电视了。假如你在其他人观看节目时关闭电视，那么就一定会被其余的观众声讨！

当你希望将堆上的一些数据分享给程序的多个部分同时使用，而又无法在编译期确定哪个部分会最后释放这些数据时，我们就可以使用Rc<T>类型。相反地，如果我们能够在编译期确定哪一部分会最后释放数据，那么就只需要让这部分代码成为数据的所有者即可，仅仅靠编译期的所有权规则也可以保证程序的正确性。

需要注意的是，Rc<T>只能被用于单线程场景中。我们会在第16章讨论并发时再来研究如何在多线程程序中使用引用计数。

使用Rc<T>共享数据

我们曾经在示例15-5的链接列表程序中使用了Box<T>。这一次我们会创建出两个列表，并让它们同时持有第三个列表的所有权，结构如图15-3所示。

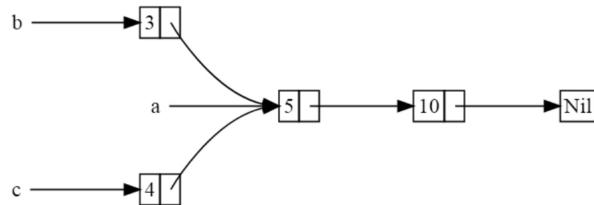


图15-3 b和c两个列表同时持有第三个列表a的所有权

我们会首先创建一个包含5和10的列表a，并接着创建另外两个列表：以3开始的b和以4开始的c。b和c两个列表会连接至包含了5和10的列表a。换句话说，这两个列表将会共享第一个列表中的5和10。

基于Box<T>实现的List无法实现这样的场景，示例15-17中的代码无法正常运行。

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))));

① let b = Cons(3, Box::new(a));
② let c = Cons(4, Box::new(a));
}
```

示例15-17：Box<T>无法让两个列表同时持有另一列表的所有权

尝试编译这段代码会出现如下所示的错误：

```
error[E0382]: use of moved value: `a`
--> src/main.rs:13:30
|
12 |     let b = Cons(3, Box::new(a));
|           - value moved here
13 |     let c = Cons(4, Box::new(a));
|           ^ value used here after move
```

```
|  
= note: move occurs because `a` has type `List`, which does not implement  
the `Copy` trait
```

Cons变体持有它存储的数据。因此，整个a列表会在我们创建b列表时❶被移动至b中。换句话说，b列表拥有了a列表的所有权。当我们随后再次尝试使用a来创建c列表时❷就会出现编译错误，因为a已经被移走了。

我们当然可以改变Cons的定义来让它持有一个引用而不是所有权，并为其指定对应的生命周期参数。但这个生命周期参数会要求列表中所有元素的存活时间都至少要和列表本身一样长。换句话说，借用检查器最终会阻止我们编译类似于let a = Cons(10, &Nil); 这样的代码，因为此处临时创建的Nil变体值会在a取得其引用前被丢弃。

另外一种解决方案是，我们可以将List中的Box<T>修改为Rc<T>，如示例15-18所示。在这段新的代码中，每个Cons变体都会持有一个值及一个指向List的Rc<T>。我们只需要在创建b的过程中克隆a的Rc<List>智能指针即可，而不再需要获取a的所有权。这会使a和b可以共享Rc<List>数据的所有权，并使智能指针中的引用计数从1增加到2。随后，我们在创建c时也会同样克隆a并将引用计数从2增加到3。每次调用Rc::clone都会使引用计数增加，而Rc<List>智能指针中的数据只有在引用计数器减少到0时才会被真正清理掉。

src/main.rs

```
enum List {  
    Cons(i32, Rc<List>),  
    Nil,  
}  
  
use crate::List::{Cons, Nil};  
❶use std::rc::Rc;  
  
fn main() {  
    ❷ let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));  
    ❸ let b = Cons(3, Rc::clone(&a));  
    ❹ let c = Cons(4, Rc::clone(&a));  
}
```

示例15-18：使用Rc<T>定义List

由于Rc<T>没有被包含在预导入模块中，所以我们必须使用use语句来将它引入作用域❶。我们在main函数中首先创建了一个包含5和10

的列表，并将这个新建的Rc<List>存入了a❷。随后，我们在创建b❸和c❹时调用的Rc::clone函数会接收a中Rc<List>的引用作为参数。

你可以在这里调用a.clone()而不是Rc::clone(&a)来实现同样的效果，但Rust的惯例是在此场景下使用Rc::clone，因为Rc::clone不会执行数据的深度拷贝操作，这与绝大多数类型实现的clone方法明显不同。调用Rc::clone只会增加引用计数，而这不会花费太多时间。但与此相对的是，深度拷贝则常常需要花费大量时间来搬运数据。因此，在引用计数上调用Rc::clone可以让开发者一眼就区分开“深度拷贝”与“增加引用计数”这两种完全不同的克隆行为。当你需要定位存在性能问题的代码时，就可以忽略Rc::clone而只需要审查剩余的深度拷贝克隆行为即可。

克隆Rc<T>会增加引用计数

接下来，让我们继续修改示例15-18中的代码来观察Rc<List>在创建和丢弃引用时的计数变化情形。

示例15-19在main函数中创建了一个被包裹在内部作用域中的c，让我们来看一看c离开作用域时引用计数会产生怎样的变化。

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：打印引用计数

我们在每一个引用计数发生变化的地方调用Rc::strong_count函数来读取引用计数并将它打印出来。这个函数之所以被命名为strong_count（强引用计数）而不是count（计数），是因为Rc<T>类

型还拥有一个weak_count（弱引用计数）函数。我们会在随后的“使用Weak<T>代替Rc<T>来避免循环引用”一节中详细介绍在什么情况下使用weak_count。

运行代码可以观察到如下所示的输出结果：

```
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存储的Rc<List>拥有初始引用计数1，并在随后每次调用clone时增加1。而当c离开作用域被丢弃时，引用计数减少1。我们不需要像调用Rc::clone来增加引用计数一样手动调用某个函数来减少引用计数：Rc<T>的Drop实现会在Rc<T>离开作用域时自动将引用计数减1。

我们没有在这段输出中观察到b和a在main函数末尾离开作用域时的情形，但它们会让计数器的值减少到0并使Rc<List>被彻底地清理掉。使用Rc<T>可以使单个值拥有多个所有者，而引用计数机制则保证了这个值会在其所有的所有者存活时一直有效，并在所有者全部离开作用域时被自动清理。

Rc<T>通过不可变引用使你可以在程序的不同部分之间共享只读数据。如果Rc<T>也允许你持有多个可变引用的话，那么它就会违反在第4章讨论过的其中一个借用规则：多个指向同一区域的可变借用会导致数据竞争及数据不一致。但在实际开发中，允许数据可变无疑是非常有用的！因此，我们接下来将要讨论内部可变性模式及RefCell<T>类型，该类型可以与Rc<T>联合使用来绕开不可变的限制。

RefCell<T>和内部可变性模式

内部可变性（interior mutability）是Rust的设计模式之一，它允许你在只持有不可变引用的前提下对数据进行修改；通常而言，类似的行为会被借用规则所禁止。为了能够改变数据，内部可变性模式在它的数据结构中使用了unsafe（不安全）代码来绕过Rust正常的可变性和借用规则。我们会在第19章学习如何使用不安全代码。假如我们能够保证自己的代码在运行时符合借用规则，那么就可以在即使编译器无法在编译阶段保证符合借用规则的前提下，也能使用那些采取了内部可变性模式的类型。实现过程中涉及的那些不安全代码会被妥善地封装在安全的API内，而类型本身从外部看来依然是不可变的。

接下来，我们会讨论一个使用了内部可变性模式的类型：`RefCell<T>`。

使用RefCell<T>在运行时检查借用规则

与`Rc<T>`不同，`RefCell<T>`类型代表了其持有数据的唯一所有权。那么，`RefCell<T>`和`Box<T>`的区别究竟在哪里呢？让我们回忆一下在第4章学习的借用规则：

- 在任何给定的时间里，你要么只能拥有一个可变引用，要么只能拥有任意数量的不可变引用。
- 引用总是有效的。

对于使用一般引用和`Box<T>`的代码，Rust会在编译阶段强制代码遵守这些借用规则。而对于使用`RefCell<T>`的代码，Rust则只会在运行时 检查这些规则，并在出现违反借用规则的情况下触发panic来提前中止程序。

将借用规则的检查放在编译阶段有许多优势：它不仅会帮助我们在开发阶段尽早地暴露问题，而且不会带来任何运行时的开销，因为所有检查都已经提前执行完毕。因此，在编译期检查借用规则对于大多数场景而言都是最佳的选择，这也正是Rust将编译期检查作为默认行为的原因。

在运行时检查借用规则则可以使我们实现某些特定的内存安全场景，即便这些场景无法通过编译时检查。静态分析（static analysis），正如Rust编译器一样，从本质上讲是保守的。并不是程序中所有的属性都能够通过分析代码来得出：其中最为经典的例子莫过于停机问题（Halting Problem）。有关它的讨论超出了本书的范畴，但这是一个非常值得研究的有趣的话题。

因为某些分析是根本无法完成的，所以Rust编译器会简单地拒绝掉所有不符合所有权规则的代码，哪怕这些代码根本没有任何问题。Rust编译器的保守正是体现于此。一旦Rust放行了某个有问题的程序，那么Rust对安全性的保证就将直接破产，进而失去用户的信任！虽然拒绝掉某些正确的程序会对开发者造成不便，但至少这样不会产生什么灾难性的后果。在这类编译器无法理解代码，但开发者可以保证借用规则能够满足的情况下，`RefCell<T>`便有了它的用武之地。

与`Rc<T>`相似，`RefCell<T>`只能被用于单线程场景中。强行将它用于多线程环境中会产生编译时错误。我们在第16章会继续讨论如何在多线程程序中使用`RefCell<T>`的功能。

下面是选择使用`Box<T>`、`Rc<T>`还是`RefCell<T>`的依据：

- `Rc<T>`允许一份数据有多个所有者，而`Box<T>`和`RefCell<T>`都只有一个所有者。
- `Box<T>`允许在编译时检查的可变或不可变借用，`Rc<T>`仅允许编译时检查的不可变借用，`RefCell<T>`允许运行时检查的可变或不可变借用。
- 由于`RefCell<T>`允许我们在运行时检查可变借用，所以即便`RefCell<T>`本身是不可变的，我们仍然能够更改其中存储的值。

内部可变性模式允许用户更改一个不可变值的内部数据。下面我们会讨论一个具有实际作用的内部可变性场景，并研究一下它的工作机制。

内部可变性：可变地借用一个不可变的值

借用规则的一个推论是，你无法可变地借用一个不可变的值。例如，下面这段代码就无法通过编译：

```
fn main() {  
    let x = 5;  
    let y = &mut x;  
}
```

尝试编译这段代码会产生如下所示的错误：

```
error[E0596]: cannot borrow immutable local variable `x` as mutable  
--> src/main.rs:3:18  
|  
2 |     let x = 5;  
|     - consider changing this to `mut x`  
3 |     let y = &mut x;  
|           ^ cannot borrow mutably
```

然而，在某些特定情况下，我们也会需要一个值在对外保持不可变性的同时能够在方法内部修改自身。除了这个值本身的方法，其余的代码则依然不能修改这个值。使用RefCell<T>就是获得这种内部可变性的一种方法。不过，RefCell<T>并没有完全绕开借用规则：我们虽然使用内部可变性通过了编译阶段的借用检查，但借用检查的工作仅仅是被延后到了运行阶段。如果你违反了借用规则，那么就会得到一个panic! 而不再只是编译时的错误。

让我们来编写一个实际运用RefCell<T>修改不可变值的例子，并观察它在其中起到的作用。

内部可变性的应用场景：模拟对象

测试替代（test double）是一个通用的编程概念，它代表了那些在测试工作中被用作其他类型替代品的类型。而模拟对象（mock object）则指代了测试替代中某些特定的类型，它们会承担起记录测

试过程的工作。我们可以利用这些记录来断言测试工作的运行是否正确。

Rust没有和其他语言中类似的对象概念，也同样没有在标准库中提供模拟对象的测试功能。但是，我们可以自行定义一个结构体来实现与模拟对象相同的功能。

设计的测试场景如下：我们希望开发一个记录并对比当前值与最大值的库，它会基于当前值与最大值之间的接近程度向外传递信息。例如，这个库可以记录用户调用不同API的次数，并将它们与设置的调用限额进行比较。

我们只会在这个库中记录当前值与最大值的接近程度，以及决定何时显示何种信息。使用库的应用程序需要自行实现发送消息的功能，例如在应用程序中打印信息、发送电子邮件、发送文字短信等。我们会提供一个Messenger trait供外部代码来实现这些功能，而使库本身不需要关心这些细节。这个库的源代码如示例15-20所示。

src/lib.rs

```
pub trait Messenger {
   ❶ fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }
}

❷ 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!");
    }
}
```

```
        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：我们的库会记录当前值与最大值的接近程度并根据不同的程度输出警告信息

这段代码的一个重点是Messenger trait，它唯一的方法send可以接收self的不可变引用及一条文本消息作为参数❶。我们创建的模拟对象就需要拥有这样的接口。另外一个重点则是LimitTracker的set_value方法，我们需要对这个方法的行为进行测试❷。你也许会尝试着改变value参数的值来进行测试，但set_value并不会返回任何可供断言的结果。实际上，我们需要在测试中确定的是，当某段程序使用一个实现了Messenger trait的值与一个max值来创建LimitTracker实例时，传入的不同value值能够触发messenger发送不同的信息。

我们的模拟对象在调用send时只需要将收到的信息存档记录即可，而不需要真的去发送邮件或短信。使用模拟对象来创建LimitTracker实例后，我们便可以通过调用set_value方法检查模拟对象中是否存储了我们希望见到的消息。按照这一思路实现的模拟对象如示例15-21所示，注意，这段代码还无法通过借用检查。

src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

① struct MockMessenger {
    ② sent_messages: Vec<String>,
}

impl MockMessenger {
    ③ fn new() -> MockMessenger {
        MockMessenger { sent_messages: vec![] }
    }
}

④ impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        ⑤ self.sent_messages.push(String::from(message));
    }
}

#[test]
```

```

⑥ 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：尝试实现的MockMessenger在编译时无法通过借用检查

这段测试代码定义的MockMessenger结构体①拥有一个sent_messages字段，它用携带String值的动态数组②来记录所有接收到的信息。我们还定义了关联函数new③来方便地创建一个不包含任何消息的新MockMessenger实例。接着，我们为MockMessenger实现了Messenger trait④，从而使它可以被用于创建LimitTracker。在send方法的定义中⑤，参数中的消息文本会被存入sent_messages的MockMessenger列表。

在测试函数中，我们希望检查LimitTracker在当前值value超过最大值max的75%时的行为⑥。函数体中的代码首先创建了一个信息列表为空的MockMessenger实例，并使用它的引用及最大值100作为参数来创建LimitTracker。随后，我们调用了LimitTracker的set_value方法，并将值80传入该方法，这个值超过了最大值100的75%。最后，我们断言MockMessenger的信息列表中存在一条被记录下来的信息。

尝试编译这段测试代码会出现如下所示的错误：

```

error[E0596]: cannot borrow immutable field `self.sent_messages` as mutable
--> src/lib.rs:52:13
|
51 |         fn send(&self, message: &str) {
|             ----- use `&mut self` here to make mutable
52 |             self.sent_messages.push(String::from(message));
|             ^^^^^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable fie
|

```

由于send方法接收了self的不可变引用，所以我们无法修改MockMessenger的内容来记录消息。我们也无法按照编译器在错误提示信息中给出的建议来将函数签名修改为&mut self，因为修改后的签名与Messenger trait定义的send的签名不符（你可以自行尝试进行这样的修改并观察出现的错误）。

这就是一个内部可变性能够大显身手的场景！只要在RefCell<T>中存入sent_messages，send方法就可以修改sent_messages来存储我们看到的信息了！修改后的代码如示例15-22所示。

src/lib.rs

```
# [cfg(test)]
mod tests {
    use super::*;

    use std::cell::RefCell;

    struct MockMessenger {
        ❶ sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            ❷ MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            ❸ self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --略

        --
        ❹ assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

示例15-22：在保持外部值不可变的前提下，使用RefCell<T>来修改内部存储的值

sent_messages字段的类型变为了RefCell<Vec<String>>**❶**，而不是Vec<String>。在new函数中，我们使用了一个空的动态数组来创建新的RefCell<Vec<String>>实例**❷**。

对于send方法的实现，其第一个参数依然是self的不可变借用，以便与 trait 的定义维持一致。随后的代码调用了RefCell<Vec<String>>类型的self.sent_messages的borrow_mut方法**❸**，来获取RefCell<Vec<String>>内部值（也就是动态数组）的可变

引用。接着，我们便可以在动态数组的可变引用上调用push方法来存入数据，从而将已发送消息记录在案。

最后，我们还需要稍微修改一下断言语句。为了查看内部动态数组的长度，我们需要先调用RefCell<Vec<String>>的borrow方法来取得动态数组的不可变引用④。

在了解了如何使用RefCell<T>后，让我们来接着研究一下它是如何工作的吧！

使用RefCell<T>在运行时记录借用信息

我们会在创建不可变和可变引用时分别使用语法`&`与`&mut`。对于RefCell<T>而言，我们需要使用borrow与borrow_mut方法来实现类似的功能，这两者都被作为RefCell<T>的安全接口来提供给用户。borrow方法和borrow_mut方法会分别返回Ref<T>与RefMut<T>这两种智能指针。由于这两种智能指针都实现了Deref，所以我们可以把它们当作一般的引用来对待。

RefCell<T>会记录当前存在多少个活跃的Ref<T>和RefMut<T>智能指针。每次调用borrow方法时，RefCell<T>会将活跃的不可变借用计数加1，并且在任何一个Ref<T>的值离开作用域被释放时，不可变借用计数将减1。RefCell<T>会基于这一技术来维护和编译器同样的借用检查规则：在任何一个给定的时间里，它只允许你拥有多个不可变借用或一个可变借用。

当我们违背借用规则时，相比于一般引用导致的编译时错误，RefCell<T>的实现会在运行时触发panic。示例15-23稍微修改了一下示例15-22中的send函数。这段新的代码故意在同一个作用域中创建两个同时有效的可变借用，以便演示RefCell<T>在运行时会如何阻止这一行为。

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：在同一个作用域中创建两个可变引用，这会使RefCell<T>引发panic

我们首先创建了一个RefMut<T>类型的one_borrow变量来存储从borrow_mut返回的结果，并在随后用同样的方法在two_borrow变量中创建另外一个可变借用。这段代码实现了一个不被允许的情形：同一个作用域中出现了两个可变引用。示例15-23中的测试代码可以顺利地通过编译，但却会在测试运行时运行失败：

```
---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at
'already borrowed: BorrowMutError', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

注意，这段代码触发了panic并输出信息already borrowed: BorrowMutError，这是RefCell<T>在运行时处理违反借用规则代码的方法。

在运行时而不是编译时捕获借用错误意味着，开发者很有可能到研发后期才得以发现问题，甚至是将问题暴露到生产环境中。另外，代码也会因为运行时记录借用的数量而产生些许性能损失。但不管怎么样，使用RefCell<T>都能够使我们在不可变的环境中修改自身数据，从而成功地编写出能够记录消息的不可变模拟对象。只要能够做出正确的取舍，你就可以借助RefCell<T>来完成某些常规引用无法完成的功能。

将Rc<T>和RefCell<T>结合使用来实现一个拥有多重所有权的可变数据

将RefCell<T>和Rc<T>结合使用是一种很常见的用法。Rc<T>允许多个所有者持有同一数据，但只能提供针对数据的不可变访问。如果我们在Rc<T>内存储了RefCell<T>，那么就可以定义出拥有多个所有者且能够进行修改的值了。

让我们以示例15-18中定义的链接列表为例，它使用Rc<T>来让多个列表共享同一个列表的所有权。由于Rc<T>只能存储不可变值，所以列表一经创建，其中的值就无法被再次修改了。现在，让我们在Cons定义中使用RefCell<T>来实现修改现有列表内数值的功能，如示例15-24所示。

src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    ❶ let value = Rc::new(RefCell::new(5));

    ❷ let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    ❸ *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>>创建一个可变的List

main函数中的代码首先创建了一个Rc<RefCell<i32>>实例，并将它暂时存入了value变量中❶以便之后可以直接访问。接着，我们使用含有value的Cons变体创建一个List类型的a变量❷。为了确保a和value同时持有内部值5的所有权，这里的代码还克隆了value，而不仅仅只是将value的所有权传递给a，或者让a借用value。

与示例15-18类似，为了让随后创建的b和c能够同时指向a，我们将a封装到了Rc<T>中。

创建完a、b、c这3个列表后，我们通过调用borrow_mut来将value指向的值增加10❸。注意，这里使用了自动解引用功能（在第5章讨论

过) 来将 Rc<T> 解引用为 RefCell<T>。borrow_mut 方法会返回一个 RefMut<T> 智能指针, 我们可以使用解引用运算符来修改其内部值。

打印 a、b、c 这 3 个列表可以看到它们存储的值都从 5 变为了 15:

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

这种实现方法非常简单明了! 通过使用 RefCell<T>, 我们拥有的 List 保持了表面上的不可变状态, 并能够在必要时借由 RefCell<T> 提供的方法来修改其内部存储的数据。运行时的借用规则检查同样能够帮助我们避免数据竞争, 在某些场景下为了必要的灵活性而牺牲一些运行时性能也是值得的。

标准库还提供了其他一些类型来实现内部可变性, 例如与 RefCell<T> 十分类似的 Cell<T>, 但相比于前者通过借用来实现内部数据的读写, Cell<T> 选择了通过复制来访问数据。另外还有在第 16 章会讨论到的 Mutex<T>, 它被用于实现跨线程情形下的内部可变性模式。请参考标准库文档来了解有关这些类型有哪些区别的更多信息。

循环引用会造成内存泄漏

Rust提供的内存安全保障使我们很难在程序中意外地制造出永远不会得到释放的内存空间（也就是所谓的内存泄漏），但这也并非是不可能的。与数据竞争不同，在编译期彻底防止内存泄漏并不是Rust作出的保证之一，这也意味着内存泄漏在Rust中是一种内存安全行为。你可以通过使用Rc<T>和RefCell<T>看到Rust是允许内存泄漏的：我们能够创建出互相引用成环状的实例。由于环中每一个指针的引用计数都不可能减少到0，所以对应的值也不会被释放丢弃，这就造成了内存泄漏。

创建循环引用

让我们来看一看循环引用是如何发生的，再来学习如何才能避免它。示例15-25中的代码定义了一个List枚举，以及它的tail方法。

src/main.rs

```
# fn main() {}
use std::rc::Rc;
use std::cell::RefCell;
use crate::List::{Cons, Nil};

#[derive(Debug)]
enum List {
    ❶ Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    ❷ fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

示例15-25：一个使用RefCell<T>定义的链接列表，使我们可以修改Cons变体指向的内容

这里的List枚举与示例15-5中的稍微有些区别。Cons变体的第二项元素变为了RefCell<Rc<List>>**①**，这也意味着我们现在可以灵活修改Cons变体指向的下一个List值，而不再像示例15-24一样修改i32值了。为了能够较为方便地访问Cons变体中的第二项元素，我们还专门添加了tail方法**②**。

示例15-26为示例15-25定义的代码添加了一个main函数。这段代码首先建立了一个普通的列表a与一个指向a的列表b；随后，它又将列表a修改为指向b，如此便可以形成一个循环引用。中间添加的那些println! 可以让你观察到代码在运行至各个阶段后引用计数的具体数值。

src/main.rs

```
fn main() {
    ❶ 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());

    ❷ 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());

    ❸ if let Some(link) = a.tail() {
        ❹ *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));

    // 取消下面的注释行便可以观察到循环引用；它会造成栈的溢出。

    // println!("a next item = {:?}", a.tail());
}
```

示例15-26：构造出一个循环引用，它由两个互相指向对方的List组成

这段代码首先创建出一个Rc<List>实例并存储至变量a，其中的List被赋予了初始值5, Nil**①**。随后，我们又创建出一个Rc<List>实

例并存储至变量b，其中的List包含数值10及指向列表a的指针②。

接下来，我们将a指向的下一个元素Nil修改为b来创建出循环。为了实现这一修改，我们需要调用tail方法来得到a的RefCell< Rc< List >>值的引用并将它暂存在link变量中③。接着，我们使用RefCell< Rc< List >>的borrow_mut方法来将Rc< List >中存储的值由Nil修改为b中存储的Rc< List >④。

保留最后一行println! 的注释并运行程序，你会看到如下所示的结果：

```
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
```

在完成a指向b的操作后，这两个Rc< List >实例的引用计数就都变为了2。而在main函数结尾处，Rust会首先释放b，并使b存储的Rc< List >实例的引用计数减少1。

但由于a仍然持有一个指向b中Rc< List >的引用，所以这个Rc< List >的引用计数仍然是1而不是0。因此，该Rc< List >在堆上的内存不会得到释放。这块内存会永远以引用计数为1的状态保留在堆上。我们绘制了图15-4来图形化地演示这一循环引用的情形。

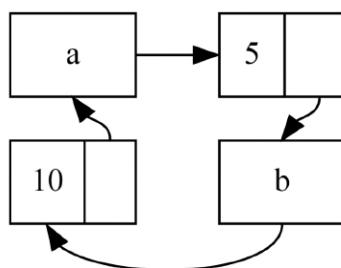


图15-4 列表a和b互相指向的循环引用

假如去除最后一行println! 的注释并再次运行程序，那么Rust会在尝试将这个循环引用打印出来的过程中反复地从a跳转至b，再从b跳转至a；整个程序会一直处于这样的循环中直到发生栈溢出为止。

在这个示例中，由于程序在创建出循环引用后就立即结束运行了，所以它不会造成特别严重的后果。但对于一个逻辑更复杂的程序而言，在循环引用中分配并长时间持有大量内存会让程序不断消耗掉超过业务所需的内存，这样的漏洞可能会导致内存逐步消耗殆尽并最终拖垮整个系统。

在Rust中创建出循环引用并不是特别容易，但也绝非不可能。如果你的程序中存在RefCell<T>包含Rc<T>或其他联用了内部可变性与引用计数指针的情形，那么你就需要自行确保不会在代码中创建出循环引用；Rust的特性对这样的场景无能为力。创造出循环引用意味着你的代码逻辑出现了bug，而这些bug可以通过自动化测试、代码评审及其他软件开发手段来尽可能地避免。

另外一种用于解决循环引用的方案需要重新组织数据结构，它会将引用拆分为持有所有权和不持有所有权两种情形。因此，你可以在形成的环状实例中让某些指向关系持有所有权，并让另外某些指向关系不持有所有权。只有持有所有权的指向关系才会影响到某个值是否能够被释放。接下来，让我们来观察一个由父子节点组成的图状数据结构，并思考非所有权关系是如何帮助我们避免循环引用的。

使用Weak<T>代替Rc<T>来避免循环引用

目前，我们已经演示了如何通过调用Rc::clone来增加Rc<T>实例的strong_count引用计数，并指出Rc<T>实例只有在strong_count为0时才会被清理。不过除此之外，我们还可以通过调用Rc::downgrade函数来创建出Rc<T>实例中值的弱引用。使用Rc<T>的引用来调用Rc::downgrade函数会返回一个类型为Weak<T>的智能指针，这一操作会让Rc<T>中weak_count的计数增加1，而不会改变strong_count的状态。Rc<T>类型使用weak_count来记录当前存在多少个Weak<T>引用，这与strong_count有些类似。它们之间的差别在于，Rc<T>并不会在执行清理操作前要求weak_count必须减为0。

强引用可以被我们用来共享一个Rc<T>实例的所有权，而弱引用则不会表达所有权关系。一旦强引用计数减为0，任何由弱引用组成的循环就会被打破。因此，弱引用不会造成循环引用。

由于我们无法确定Weak<T>引用的值是否已经被释放了，所以我们需要在使用Weak<T>指向的值之前确保它依然存在。你可以调用Weak<T>实例的upgrade方法来完成这一验证。此函数返回的Option<Rc<T>>会在Rc<T>值依然存在时表达为Some，而在Rc<T>值被释放时表达为None。由于upgrade返回的是Option<T>类型，所以Rust能够保证Some和None两个分支都得到妥善的处理，而不会产生无效指针之类的问题。

为了举例，我们放弃了仅仅指向下一个元素的列表结构，而会在接下来的示例中创建一棵树，它的每个节点都能够指向自己的父节点与全部的子节点。

创建树状数据结构体：带有子节点的Node

首先，我们会创建出一个能够指向子节点的Node结构体，它可以存储一个i32值及指向所有子节点的引用：

src/main.rs

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>,
}
```

我们希望Node持有自身所有的子节点并通过变量来共享它们的所用权，从而使我们可以直接访问树中的每个Node。因此，我们将Vec<T>的元素定义为Rc<Node>类型的值。由于我们还希望能够灵活修改节点的父子关系，所以我们在children字段中使用RefCell<T>包裹Vec<Rc<Node>>来实现内部可变性。

接着，我们将使用这个结构体定义一个值为3且没有子节点的Node实例，并将它作为叶子节点存入leaf变量。随后，我们还会再定义一个值为5且将leaf作为子节点的branch实例，如示例15-27所示。

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leaf叶子节点和包含leaf子节点的branch节点

我们克隆了leaf的Rc<Node>实例，并将它存入branch。这意味着leaf中的Node现在分别拥有了leaf与branch两个所有者。我们可以使用branch. children来从branch访问leaf，但反之则暂时还不行。这是因为leaf并不持有branch的引用，它甚至对两个节点之间存在父子关系的事实一无所知。接下来，我们会修改代码来让leaf指向自己的父节点。

增加子节点指向父节点的引用

为了让子节点意识到父节点的存在，我们为Node结构体添加了一个parent字段。这里的麻烦在于决定parent究竟应该使用哪种类型。Rc<T>这种类型肯定不是一个好的选择，因为它会创建出循环引用：在branch. children指向leaf的同时使leaf. parent指向branch会导致两者的strong_count都无法归0。

现在换一种思路来考虑此处的父子节点关系：父节点自然应该拥有子节点的所有权，因为当父节点被丢弃时，子节点也应当随之被丢弃。但子节点却不应该拥有父节点，父节点的存在性不会因为丢弃子节点而受到影响。这正是应当使用弱引用的场景！

因此，我们会采用Weak<T>而不是Rc<T>来定义parent，也就是本例中的RefCell<Weak<Node>>类型。新的Node结构体定义如下所示：

src/main.rs

```

use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
}

```

```
        children: RefCell<Vec<Rc<Node>>>,
    }
```

这样，节点便可以指向父节点却不持有它的所有权了。示例15-28根据这段定义更新了main函数，使leaf节点指向了自己的父节点branch。

src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
    ❶ parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    ❷ println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
    ❸ parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    ❹ *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    ❺ println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

示例15-28：leaf节点持有一个指向父节点branch的弱引用

除了parent字段，创建leaf节点的代码与示例15-27中的区别不大。由于leaf一开始不存在父节点，所以我们创建了一个空的Weak<Node>引用实例来初始化parent字段❶。

如果在这个时候使用upgrade方法来获得指向leaf父节点的引用，那么我们就会得到一个None值。我们可以从第一个println!语句的输出中观察到这一现象❷：

```
leaf parent = None
```

因为branch没有父节点，所以我们在创建branch时将parent字段同样设置为了一个空的Weak<Node>引用❸。随后的代码依然将leaf用作了branch的子节点。当branch创建完毕后，我们就可以修改leaf来增加指向父节点的Weak<Node>引用了❹。为了实现这一目的，我们通过RefCell<Weak<Node>>的borrow_mut方法取出leaf中parent字段的可

变借用。随后，我们使用Rc::downgrade函数来获取branch中Rc<Node>的Weak<Node>引用，并将它存入leaf的parent字段中。

当我们再次打印leaf的父节点时⑤，便可以看到一个包含了branch实际内容的Some变体。这意味着leaf现在可以访问父节点了！另外，现在打印leaf还可以避免示例15-26中因循环引用而导致的栈溢出故障，因为Weak<Node>引用会被直接打印为(Weak)。

```
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) },
    children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
        children: RefCell { value: [] } }] } })
```

有限的输出意味着代码中没有产生循环引用。这一结论同样可以通过观察Rc::strong_count和Rc::weak_count的计数值来得出。

显示strong_count和weak_count计数值的变化

接下来，我们会将branch的创建过程移动至一个新创建的内部作用域中，让我们来看一看Rc<Node>实例的strong_count与weak_count计数值会发生些什么样的变化。我们可以通过这一实验观察到branch在创建和丢弃时发生的操作。修改后的代码如示例15-29所示。

src/main.rs

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    ❶ println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    ❷ {
        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);

        ❸ println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );
    }
}
```

```

        Rc::strong_count(&branch),
        Rc::weak_count(&branch),
    );
}

④ println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
    ⑤ );
}

⑥ println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
⑦ println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
);
}

```

示例15-29：在内层作用域中创建branch并观察强引用和弱引用的计数

leaf中的Rc<Node>在创建完毕后，其强引用计数为1，弱引用计数为0①。随后，我们在内部作用域②中创建了branch并将它与leaf关联起来，此时③branch中Rc<Node>的强引用计数为1，弱引用计数也为1（因为leaf.parent通过Weak<Node>指向了branch）。当我们打印leaf的计数时④可以观察到，它的强引用计数变为了2，因为branch在创建过程中克隆了leaf变量的Rc<Node>，并将它存入了自己的branch.children中。此时，leaf的弱引用仍然为0。

当内部作用域结束时⑤，branch会离开作用域并使Rc<Node>的强引用计数减为0，从而导致该Node被丢弃。虽然此时branch的弱引用计数因为leaf.parent的指向依然为1，但这并不会影响到Node是否会被丢弃。这段代码没有产生任何内存泄漏！

试图在作用域结束后访问leaf的父节点会得到一个None值⑥。当程序结束时⑦，由于只有leaf变量指向了存储在自身中的Rc<Node>，所以这个Rc<Node>的强引用计数为1。

所有这些用于管理引用计数及值释放的逻辑都被封装到了Rc<T>与Weak<T>类型，以及它们对Drop trait的具体实现中。通过在Node定义中将子节点指向父节点的关系定义为一个Weak<T>引用，可以使父子节点在指向彼此的同时避免产生循环引用或内存泄漏。

总结

本章涉及了如何使用智能指针来实现不同于Rust常规引用的功能保障与取舍。Box<T>类型拥有固定的大小并指向一段分配于堆上的数据。Rc<T>类型通过记录堆上数据的引用次数使该数据可以拥有多个所有者。RefCell<T>类型则通过其内部可变性模式使我们可以修改一个不可变类型的内部值；它会在运行时而不是编译时承担起维护借用规则的责任。

我们还讨论了实现智能指针功能不可或缺的Deref和Drop这两个trait。最后，我们研究了会触发内存泄漏的循环引用问题，以及如何使用Weak<T>来避免它们。

如果本章的内容引起了你的兴趣并希望立即开始实现智能指针的话，那么你可以参考Rust官方网站上的*The Rustonomicon* 来获得更多有用的信息。

接下来，我们会开始讨论Rust中的并发。你甚至能够在其中学习到几种新的智能指针。

第16章

无畏并发



安全并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）与并行编程（parallel programming）这两种概念随着计算机设备的多核心化而变得越来越重要。前者允许程序中的不同部分相互独立地运行，而后者则允许程序中的不同部分同时执行。从历史上看，在这类场景下进行编程往往是非常困难且易于出错的，而Rust则希望改变这种情形。

Rust团队曾经认为保证内存安全和防止并发问题是两个截然不同的挑战，我们需要使用不同的方法来解决它们。但是随着时间的推移，开发团队发现所有权和类型系统这套强有力的工具集能够同时帮助我们管理内存安全及并发问题！借助所有权和类型检查，许多并发问题可以在Rust中暴露为编译时错误而不是运行时错误。因此，相比于在运行时遭遇并发缺陷后花费大量时间来重现特定的问题场景，Rust编译器会直接拒绝不正确的代码并给出解释问题的错误提示信息。这使得代码中的并发缺陷可以在开发过程中被及时修复，而不必等到它们被发布至生产环境后暴露出来。我们为Rust的这一特性起了一个昵称：无畏并发（fearless concurrency）。无畏并发可以让你编写出没有诡异缺陷的代码，并且易于重构而不会引入新的缺陷。

注意

为了简单起见，我们将很多问题概括地称作并发，而不是更精确地分为并发和并行。如果本书是一本专门讨论并发和并行问题的书，我们会将两者明确地区分开来。但就本章而言，请读者在见到并发一词时自行按照并发或并行来进行理解。

许多语言用来解决并发问题的方案都是较为教条的。例如，Erlang提供了一套优雅的消息传递并发特性，但却没有提供可以在线程间共享状态的简单方法。对于高级语言来说，只支持全部解决方案的一部分是完全可以理解的设计策略。因为高级语言往往通过放弃部分控制能力来获得有益于用户的抽象。但是，底层语言则被期望在任意场景下都可以提供一套性能最佳的解决方案，并对硬件建立尽可能少的抽象。因此，Rust提供了多种建模问题的工具来应对不同的场景和需求。

我们会在本章讨论以下话题：

- 如何创建线程来同时运行多段代码。
- 使用通道在线程间发送消息的消息传递式并发。
- 允许多个线程访问同一片数据的共享状态式并发。
- Sync trait与Send trait，能够将Rust的并发保证从标准库中提供的类型扩展至用户自定义类型。

使用线程同时运行代码

在大部分现代操作系统中，执行程序的代码会运行在进程（process）中，操作系统会同时管理多个进程。类似地，程序内部也可以拥有多个同时运行的独立部分，用来运行这些独立部分的就叫作线程（thread）。

由于多个线程可以同时运行，所以将程序中的计算操作拆分至多个线程可以提高性能。但这也增加了程序的复杂度，因为不同线程在执行过程中的具体顺序是无法确定的。这可能会导致一系列的问题，比如：

- 当多个线程以不一致的顺序访问数据或资源时产生的竞争状态（race condition）。
- 当两个线程同时尝试获取对方持有的资源时产生的死锁（deadlock），它会导致这两个线程无法继续运行。
- 只会出现在特定情形下且难以稳定重现和修复的bug。

尽管Rust试图减轻使用线程带来的负面影响，但在多线程场景下进行编程依然需要格外小心。这种编程模型使用的代码结构不同于运行在单线程中的程序。

现有的编程语言采用了不同的方式来实现线程。许多操作系统都提供了用于创建新线程的API。这种直接利用操作系统API来创建线程的模型常常被称作*1:1* 模型，它意味着一个操作系统线程对应一个语言线程。

也有许多编程语言提供了它们自身特有的线程实现，这种由程序语言提供的线程常常被称为绿色线程（green thread），使用绿色线程的语言会在拥有不同数量系统线程的环境下运行它们。为此，绿色

线程也被称为*M:N* 模型，它表示M个绿色线程对应着N个系统线程，这里的M与N不必相等。

每一种模型都有其自身的优势和取舍。对于Rust而言，设计过程中最重要的权衡因素在于是否需要提供运行时支持。运行时（runtime）是一个容易令人迷惑的术语，它在不同的上下文中拥有不同的含义。

在当前语境下，运行时指语言中那些被包含在每一个可执行文件中的代码。不同的语言拥有不同大小的运行时代码。除汇编语言之外，编程语言总是会包含一定数量的运行时代码。因此，当人们提到某种语言“没有运行时”的时候，他们想要表达的其实是该语言的“运行时非常小”。较小的运行时拥有较少的功能，但却可以生成较小的二进制文件，并可以使该语言能够方便地在众多场景下与其他语言组合使用。尽管许多语言选择了增加运行时来提供更多的功能，但Rust会尽可能地保持几乎没有运行时的状态，这使我们可以方便地与C语言进行交互并获得较高的性能。

由于绿色线程的*M:N*模型需要一个较大的运行时来管理线程，所以Rust标准库只提供了1:1线程模型的实现。但得益于Rust良好的底层抽象能力，Rust社区中涌现出了许多支持*M:N*线程模型的第三方包。你可以选择付出一定开销来获得期望的特性，诸如更强的线程控制能力、更低的线程上下文切换开销等。

接下来，让我们来看一看如何使用标准库中的线程API。

使用spawn创建新线程

我们可以调用`thread::spawn`函数来创建线程，它接收一个闭包（在第13章中讨论过）作为参数，该闭包会包含我们想要在新线程（生成线程）中运行的代码。示例16-1中的代码可以在主线程和新线程中各自打印出一些文本。

src/main.rs

```
use std::thread;  
  
use std::time::Duration;
```

```

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}

```

示例16-1：创建新线程来打印部分信息，并由主线程打印出另外一部分信息

需要注意的是，只要这段程序中的主线程运行结束，创建出的新线程就会相应停止，而不管它的打印任务是否完成。每次运行这段程序都有可能产生不同的输出，但它们都会类似于下面的样子：

```

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!

```

调用`thread::sleep`会强制当前的线程停止执行一小段时间，并允许一个不同的线程继续运行。这些线程可能会交替执行，但我们无法对它们的执行顺序做出任何保证：执行顺序由操作系统的线程调度策略决定。在上面这次运行中，主线程首先打印出了文本，即便新线程的打印语句要出现得更早一些。另外，虽然我们要求新线程不停地打印文本直到`i`迭代到9，但它在主线程停止前仅仅迭代到了5。

如果你在运行这段代码时只观察到了主线程中的输出，或者没有看到任何交替出现的打印，那么你可以试着增加循环中表示范围的数字来为操作系统创造出更多进行线程切换的机会。

使用`join`句柄等待所有线程结束

由于主线程的停止，示例16-1中的代码会在大部分情形下提前终止新线程，它甚至不能保证新线程一定会得到执行。这同样是因为我们无法对线程的执行顺序做出任何保证而导致的！

我们可以通过将`thread::spawn`返回的结果保存在一个变量中，来避免新线程出现不执行或不能完整执行的情形。`thread::spawn`的返回值类型是一个自持有所有权的`JoinHandle`，调用它的`join`方法可以阻塞当前线程直到对应的新线程运行结束。示例16-2中的代码展示了如何使用示例16-1中新线程的`JoinHandle`，并通过调用`join`方法来保证新线程能够在`main`函数退出前执行完毕。

src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

示例16-2：保存`thread::spawn`的`JoinHandle`来保证新线程能够执行完毕

在线程句柄上调用`join`函数会阻塞当前线程，直到句柄代表的线程结束。阻塞线程意味着阻止一个线程继续运行或使其退出。由于我们将`join`函数放置到了主线程的`for`循环之后，所以运行示例16-2中的代码会产生如下所示的输出：

```
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!
```

这两个线程依然交替地打印出了信息，但由于我们调用了`handle.join()`，所以主线程只会在新线程运行结束后退出。

如果将`handle.join()`放置到`main`函数的`for`循环之前会发生什么呢？代码如下所示：

src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

在这段代码中，由于主线程会等待新线程执行完毕后才开始执行自己的`for`循环，所以它的输出将不再出现交替的情形，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!
```

在并发编程中，诸如在哪里调用`join`等微小的细节也会影响到多个线程是否能够同时运行。

在线程中使用move闭包

move闭包常常被用来与`thread::spawn`函数配合使用，它允许你在某个线程中使用来自另一个线程的数据。

我们在第13章曾经提到过，你可以在闭包的参数列表前使用`move`关键字来强制闭包从外部环境中捕获值的所有权。这一技术在我们创建新线程时尤其有用，它可以跨线程地传递某些值的所有权。

注意，示例16-1中传递给`thread::spawn`的闭包没有捕获任何参数，因为新线程的代码并不依赖于主线程中的数据。但是，为了使用主线程中的数据，新线程的闭包必须捕获它所需要的值。示例16-3中的代码试图在主线程中创建一个动态数组，并接着在新线程中使用它。但稍后我们会看到，这种写法是行不通的。

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：尝试在另外的线程中使用主线程中创建的动态数组

由于代码中的闭包使用了`v`，所以它会捕获`v`并使其成为闭包环境的一部分。又因为`thread::spawn`会在新线程中运行这个闭包，所以我们应当能够在新线程中访问`v`。但是，当我们编译这段示例代码时却会出现如下所示的错误：

```
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
|
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 || {
|
```

Rust在推导出如何捕获v后决定让闭包借用v，因为闭包中的`println!`只需要使用v的引用。但这就出现了一个问题：由于Rust不知道新线程会运行多久，所以它无法确定v的引用是否一直有效。

示例16-4中的代码展示了这样一个场景：新线程捕获的v的引用在使用时极有可能不再有效了。

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); // 情况不妙
    !
    handle.join().unwrap();
}
```

示例16-4：新线程的闭包尝试从主线程中捕获的v的引用会在随后被丢弃掉

如果Rust允许我们运行这段代码，那么新线程有极大的概率会在创建后被立即置入后台，不再被执行。此时的新线程在内部持有了v的引用，但主线程却已经通过在第15章介绍过的`drop`函数将v丢弃了。当新线程随后开始执行时，v和指向它的引用全部失效了。这可不妙！

为了修复示例16-3中的编译错误，我们可以参考错误提示信息中给出的建议：

```
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 || {
|
```

通过在闭包前添加move关键字，我们会强制闭包获得它所需值的所有权，而不仅仅是基于Rust的推导来获得值的借用。对示例16-3中的代码进行修改后，代码如示例16-5所示，新的代码能够正常通过编译并按照预期运行了。

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关键字来强制闭包获得它所需值的所有权

假如我们在示例16-4中添加了move闭包，那么随后在主线程中调用drop会发生什么呢？而添加move又是否能够修复该示例中的编译错误呢？遗憾的是，当我们在闭包上添加move后，示例16-4中的代码会因为其他原因而编译失败。由于move将v移动到了闭包的环境中，所以我们无法在主线程中继续使用它来调用drop函数了。尝试编译这段代码会得到如下所示的错误：

```
error[E0382]: use of moved value: `v`
--> src/main.rs:10:10
   |
6 |     let handle = thread::spawn(move || {
   |                         ----- value moved (into closure) here
...
10|         drop(v); // 情况不妙!
   |
   |             ^ value used here after move
   |
= note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
       not implement the `Copy` trait
```

Rust的所有权规则又一次帮助了我们！示例16-3中出现的错误是因为Rust只会在新线程中保守地借用v，这也就意味着主线程可以从理论上让新线程持有的引用失效。通过将v的所有权转移给新线程，我们就可以向Rust保证主线程不会再次使用v。如果我们采用类似的方法来修改示例16-4中的代码，那么就会在主线程继续使用v时违反所有权规

则。`move`关键字覆盖了Rust的默认借用规则；当然，这并不意味着它会允许我们去违反任何的所有权规则。

基于本节对线程和线程API的基本讨论，接下来让我们看一看线程可以用来完成一些什么样的任务。

使用消息传递在线程间转移数据

使用消息传递（message passing）机制来保证并发安全正在变得越来越流行。在这种机制中，线程或actor之间通过给彼此发送包含数据的消息来进行通信。Go编程语言文档中的口号正体现了这样的思路：不要通过共享内存来通信，而是通过通信来共享内存。

Rust在标准库中实现了一个名为通道（channel）的编程概念，它可以被用来实现基于消息传递的并发机制。你可以将它想象为有活水流动的通道，比如小溪或河流。只要你将橡皮鸭或小船这样的东西放入其中，它就会顺流而下抵达水路的终点。

编程中的通道由发送者（transmitter）和接收者（receiver）两个部分组成。发送者位于通道的上游，也就是你放置橡皮鸭的地方；而接收者则位于通道的下游，也就是橡皮鸭到达的地方。某一处代码可以通过调用发送者的方法来传送数据，而另一处代码则可以通过检查接收者来获取数据。当你丢弃了发送者或接收者的任何一端时，我们就称相应的通道被关闭（closed）了。

接下来我们编写的程序会拥有两个线程，其中一个线程会产生一些值并将它们传入通道，而另外一个线程则会接收这些值并将它们打印出来。为了在演示该功能时尽可能地保持简单，我们只会使用通道来跨线程地传递一些非常简单的值。但只要熟悉了这项技术，你就利用通道来实现更复杂一些的聊天系统，甚至是可以在多个线程中执行计算并最终汇总至单一线程的分布式计算系统。

首先，我们在示例16-6中创建了一个不执行任何操作的通道。注意，这段代码还无法通过编译，因为Rust不能推导出我们希望在通道中传递的值类型。

src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

示例16-6： 创建一个通道，并将两端分别赋给tx和rx

上面的代码使用`mpsc::channel`函数创建了一个新的通道。路径中的`mpsc`是英文“multiple producer, single consumer”（多个生产者，单个消费者）的缩写。简单来讲，Rust标准库中特定的实现方式使得通道可以拥有多个生产内容的发送端，但只能拥有一个消耗内容的接收端。想象一下多股水流汇入大河的场景：任何被放入水流的东西最终都会到达大河。我们会从单个生产者开始编写程序，并在这个示例运行成功后再扩展至拥有多个生产者的场景。

函数`mpsc::channel`会返回一个含有发送端与接收端的元组。代码中用来绑定它们的变量名称为`tx`和`rx`，这也是在许多场景下发送者与接收者的惯用简写。这里还使用了带有模式的`let`语句对元组进行解构，我们会在第18章讨论带有模式的`let`语句与有关解构的具体知识。如此使用`let`语句只是为了方便地从`mpsc::channel`函数的返回值中提取元组的各个部分。

接下来，让我们将发送端移动到新线程中，并接着发送一个字符串来完成新线程与主线程的通信，如示例16-7所示。这就像将橡皮鸭放入河流上游，或是将聊天消息从一个线程发往另外一个线程。

src/main.rs

```
use std::thread;
use std::sync::mpsc;

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移动到新线程中并发送值“hi”

我们再次使用`thread::spawn`生成了一个新线程。为了让新线程拥有`tx`的所有权，我们使用`move`关键字将`tx`移动到了闭包的环境中。新线程必须拥有通道发送端的所有权才能通过通道来发送消息。

发送端提供了`send`方法来接收我们想要发送的值。这个方法会返回`Result<T, E>`类型的值作为结果；当接收端已经被丢弃而无法继续传递内容时，执行发送操作便会返回一个错误。在这个示例中，我们在出现错误时直接调用了`unwrap`来触发`panic`。但是在实际应用中，我们应该更为妥善地处理类似错误：可以回到第9章来复习有关错误处理的恰当策略。

在示例16-8的主线程中，我们会从通道的接收端获得传入的值。这就像是在河流的终点拾起了橡皮鸭，或者说是接收到一段聊天信息。

src/main.rs

```
use std::thread;
use std::sync::mpsc;

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值“hi”

通道的接收端有两个可用于获取消息的方法：`recv`和`try_recv`。我们使用的`recv`（也就是`receive`的缩写）会阻塞主线程的执行直到有值被传入通道。一旦有值被传入通道，`recv`就会将它包裹在`Result<T, E>`中返回。而如果通道的发送端全部关闭了，`recv`则会返回一个错误来表明当前通道再也没有可接收的值。

`try_recv`方法不会阻塞线程，它会立即返回`Result<T, E>`：当通道中存在消息时，返回包含该消息的`Ok`变体；否则便返回`Err`变体。当某个线程需要一边等待消息一边完成其他工作时，`try_recv`方法会非

常有用。我们可以编写出一个不断调用try_recv方法的循环，并在有消息到来时对其进行处理，而在没有消息时执行其他指令。

为了简单起见，我们在本例中使用了recv；由于示例中的主线程除等待消息之外没有其他任何工作可做，所以阻塞主线程是合适的。

运行示例16-8中的代码，你可以观察到主线程打印出了值的内容：

```
Got: hi
```

完美！

通道和所有权转移

所有权规则在消息传递的过程中扮演了至关重要的角色，因为它可以帮助你写出安全的并发代码。通过不断地在编写Rust代码时思考所有权问题，我们可以有效地避免并发编程中的常见错误。下面的实验演示了通道和所有权规则是如何通过协作来规避问题的：我们会尝试在新线程中使用一个已经发送给通道的val值。尝试编译示例16-9中的代码，并观察它会出现怎样的编译错误。

src/main.rs

```
use std::thread;
use std::sync::mpsc;

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：将val发送给通道后再尝试使用它

上面的代码首先通过调用`tx.send`将`val`值发送给通道，接着又继续尝试打印这个值。允许这样的操作可不是什么好主意：一旦这个值被发送到了另外一个线程中，那个线程就可以在我们尝试重新使用这个值之前修改或丢弃它。这些修改极有可能造成不一致或产生原本不存在的数据，最终导致错误或出乎意料的结果。幸运的是，Rust会在编译示例16-9中的代码时给出如下所示的错误提示信息：

```
error[E0382]: use of moved value: `val`
--> src/main.rs:10:31
|
9 |         tx.send(val).unwrap();
|             --- value moved here
10|         println!("val is {}", val);
|                     ^^^ value used here after move
|
= note: move occurs because `val` has type `std::string::String`, which does
not implement the `Copy` trait
```

我们的并发缺陷造成了一个编译时错误。`send`函数会获取参数的所有权，并在参数传递时将所有权转移给接收者。这可以阻止我们意外地使用已经发送的值，所有权系统会在编译时确保程序的每个部分都是符合规则的。

发送多个值并观察接收者的等待过程

虽然示例16-8中的代码可以编译运行，但你却很难观察出那两个独立的线程是否正在基于通道相互通信。因此，我们在示例16-10中修改了部分代码来证明示例16-8中的代码是并发执行的：新线程现在会发送多条信息，并在每次发送后暂停1秒钟。

src/main.rs

```
use std::thread;
use std::sync::mpsc;
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_millis(1));
        }
    });
}

fn main() {
    let (tx, rx) = mpsc::channel();

    let mut messages = rx.try_iter();
    while let Some(message) = messages.next() {
        println!("{}!", message);
    }
}
```

```

        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：发送多条消息并在每次发送后暂停1秒钟

这段代码在新线程中创建了一个用于存储字符串的动态数组。我们会迭代动态数组来逐个发送其中的字符串，并在每次发送后调用Duration值为1秒的thread::sleep函数来稍作暂停。

在主线程中，我们会将rx视作迭代器，而不再显式地调用recv函数。迭代中的代码会打印出每个接收到的值，并在通道关闭时退出循环。

运行示例16-10中的代码，你应该会观察到如下所示的输出，并体验到每次打印后出现的1秒钟的时间间隔：

```

Got: hi
Got: from
Got: the
Got: thread

```

我们并没有在主线程的for循环中执行暂停或延迟指令，这也就表明主线程确实在等待接收新线程中传递过来的值。

通过克隆发送者创建多个生产者

前面曾经提到过，mpsc是英文“multiple producer, single consumer”（多个生产者，单个消费者）的缩写。现在，让我们继续扩展示例16-10中的代码来实现多重生产者的模式。我们会通过克隆通道的发送端来创建出多个能够发送值到同一个接收端的线程，如示例16-11所示。

`src/main.rs`

```

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
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);
}
// --略
--
```

示例16-11：用多个生产者发送多条消息

我们在创建第一个新线程前调用了通道发送端的`clone`方法，这会为我们生成可以传入首个新线程的发送端句柄。随后，我们又将原始的通道发送端传入第二个新线程。这两个线程会各自发送不同的消息到通道的接收端。

你应该会在执行这段代码时观察到如下所示的输出：

```

Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
```

```
Got: thread  
Got: you
```

根据你所使用的操作系统的不同，这些字符串的打印顺序也许会有所不同。这也正是并发编程有趣且充满挑战的地方。如果你在实验时为不同的线程调用了含有不同参数的`thread::sleep`函数，那么输出结果的差异有可能更为显著且难以确定。

现在，我们知道通道是如何工作的了，接下来让我们看一看另外一种实现并发的方式。

共享状态的并发

消息传递确实是一种不错的并发通信机制，但它并不是唯一的解决方案。再次思考一下Go编程语言文档口号中前半段所说的：通过共享内存来通信。

通过共享内存来通信究竟是什么样子的？另外，为什么消息传递的拥护者会尽量避免使用这种方法并做出相反的选择？

从某种程度上来说，任何编程语言中的通道都有些类似于单一所有权的概念，因为你不应该在值传递给通道后再次使用它。而基于共享内存的并发通信机制则更类似于多重所有权概念：多个线程可以同时访问相同的内存地址。正如我们在第15章讨论的那样，我们可以通过智能指针实现多重所有权，但由于需要同时管理多个所有者，所以这会为系统增加额外的复杂性。当然，Rust的类型系统和所有权规则能够帮助我们正确地管理这些所有权。为了举例，我们会先来讨论共享内存领域中一个较为常见的并发原语：互斥体（mutex）。

互斥体一次只允许一个线程访问数据

互斥体（mutex）是英文mutual exclusion的缩写。也就是说，一个互斥体在任意时刻只允许一个线程访问数据。为了访问互斥体中的数据，线程必须首先发出信号来获取互斥体的锁（lock）。锁是互斥体的一部分，这种数据结构被用来记录当前谁拥有数据的唯一访问权。通过锁机制，互斥体守护（guarding）了它所持有的数据。

互斥体是出了名的难用，因为你必须牢记下面两条规则：

- 必须在使用数据前尝试获取锁。

- 必须在使用完互斥体守护的数据后释放锁，这样其他线程才能继续完成获取锁的操作。

在现实世界中可以对互斥体进行这样一个隐喻，你可以将它想象成一场仅有单个话筒的座谈会议。每个人在讲话前都必须发出信号来试图获取这个话筒的使用权。演讲者在拿到话筒后可以使用任意长的时间，并接着将话筒递给下一个请求发言者。如果某个演讲者在发言完成后忘记将话筒移交出去，其他人便无法再次开口。一旦针对共享话筒的管理出现了失误，整个座谈会就无法按照计划继续进行下去！

正是因为管理互斥体是一件非常棘手的工作，所以才会有那么多通道机制的拥护者。然而在Rust中，由于类型系统和所有权规则的帮助，我们可以保证自己不会在加锁和解锁这两个步骤中出现错误。

Mutex<T>的接口

为了便于演示，我们会首先在单线程环境中使用互斥体，如示例16-12所示。

src/main.rs

```
use std::sync::Mutex;

fn main() {
    ❶ let m = Mutex::new(5);

    {
        ❷ let mut num = m.lock().unwrap();
        ❸ *num = 6;
    }
    ❹ println!("m = {:?}", m);
}
```

示例16-12：简单地探索单线程场景下的Mutex<T>接口

与许多其他类型一样，我们可以使用关联函数new❶来创建Mutex<T>实例。为了访问Mutex<T>实例中的数据，我们首先需要调用它的lock方法来获取锁❷。这个调用会阻塞当前线程直到我们取得锁为止。

当前线程对于lock函数的调用会在其他某个持有锁的线程发生panic时失败。实际上，任何获取锁的请求都会在这种场景里以失败告终，所以示例中的代码选择使用unwrap在意外发生时触发当前线程的panic。

一旦获取了锁，我们便可以将它的返回值num视作一个指向内部数据的可变引用③。Rust的类型系统会确保我们在使用m的值之前执行加锁操作：因为Mutex<i32>并不是i32的类型，所以我们必须获取锁才能使用i32值。我们无法忘记或忽略这一步骤，因为类型系统并不允许我们以其他方式访问内部的i32值。

正如你可能会猜到的那样，Mutex<T>是一种智能指针。更准确地说，对lock的调用会返回一个名为MutexGuard的智能指针。这个智能指针通过实现Deref来指向存储在内部的数据，它还会通过实现Drop来完成自己离开作用域时的自动解锁操作。在示例16-12中，这种释放过程会发生在内部作用域的结尾处④。因此，我们不会因为忘记释放锁而导致其他线程无法继续使用该互斥体。锁的释放过程是自动发生的。

在释放完锁之后，我们打印出了这个互斥体的值。你可以观察到内部的i32值确实被修改为了6⑤。

在多个线程间共享Mutex<T>

现在，让我们试着在多线程环境中使用Mutex<T>来共享数据。在接下来的例子中，我们会依次启动10个线程，并在每个线程中分别为共享的计数器的值加1。一切顺利的话，这最终会让计数器的值从0累计到10。注意，接下来的几段示例代码都无法通过编译。我们需要借助示例中出现的错误来学习Mutex<T>，并观察Rust会如何帮助我们正确地使用它。示例16-13便是我们的第一个例子。

src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    ① let counter = Mutex::new(0);
    ② let mut handles = vec![];
```

```

for _ in 0..10 {
    ③ let handle = thread::spawn(move || {
        ④ let mut num = counter.lock().unwrap();

        ⑤ *num += 1;
    });
    ⑥ handles.push(handle);
}

for handle in handles {
    ⑦ handle.join().unwrap();
}

⑧ println!("Result: {}", *counter.lock().unwrap());
}

```

示例16-13：在10个线程中分别为Mutex<T>守护的计数器的值加1

与示例16-12类似，上面的代码首先创建了一个名为counter的变量来存储持有i32值的Mutex<T>①。随后，我们通过迭代数字范围创建出了10个线程②。在调用thread::spawn创建线程的过程中，我们给所有创建的线程传入了同样的闭包。这个闭包会把计数器移动至线程中③，它还会调用Mutex<T>的lock方法来进行加锁④并为互斥体中的值加1⑤。而当线程执行完毕后，num会在离开作用域时释放锁，从而让其他线程得到获取锁的机会。

与示例16-2类似，我们还在主线程中收集了所有的线程句柄⑥，并通过逐一调用句柄的join方法来确保所有生成的线程执行完毕⑦。最后，主线程会获取锁并打印出程序的结果⑧。

现在，让我们来看一看这个例子为什么无法通过编译：

```

error[E0382]: capture of moved value: `counter`
--> src/main.rs:10:27
|
9 |         let handle = thread::spawn(move || {
|                         ----- value moved (into closure) here
10|             let mut num = counter.lock().unwrap();
|                         ^^^^^^^ value captured here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:21:29
|
9 |         let handle = thread::spawn(move || {
|                         ----- value moved (into closure) here
...

```

```

21 |     println!("Result: {}", *counter.lock().unwrap());
|           ^^^^^^^^ value used here after move
|
|= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

这段错误提示信息指出，`counter`被移动进了闭包中并在调用`lock`时被捕获了。这一描述与我们的设计思路似乎完全相符，但它却是不被允许的！

为了厘清问题所在，让我们先来简化一下示例16-13中的程序。我们不再通过`for`循环来创建10个线程，而是去掉循环，并手动地生成两个线程。修改后的代码如下所示：

```

let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();

    *num += 1;
});
handles.push(handle);

let handle2 = thread::spawn(move || {
    let mut num2 = counter.lock().unwrap();

    *num2 += 1;
});
handles.push(handle2);

```

新的代码创建了两个线程，第二个线程用到的相关变量被命名为`handle2`与`num2`。再次运行代码，编译器的错误提示信息会变为：

```

error[E0382]: capture of moved value: `counter`
--> src/main.rs:16:24
|
8 |     let handle = thread::spawn(move || {
|           ----- value moved (into closure) here
...
16 |         let mut num2 = counter.lock().unwrap();
|             ^^^^^^^ value captured here after move
|
|= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:26:29
|
8 |     let handle = thread::spawn(move || {
|           ----- value moved (into closure) here
...
26 |     println!("Result: {}", *counter.lock().unwrap());
|           ^^^^^^^ value used here after move
|
|= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,

```

```
which does not implement the `Copy` trait
error: aborting due to 2 previous errors
```

啊哈！第一条错误提示信息指出，counter被移动到了handle指代的线程中。而这一移动行为阻止了我们在第二个线程中调用lock来再次捕获counter。Rust提醒我们不应该将counter的所有权移动到多个线程中。这个问题很难在前一个示例中被发现，因为我们使用了循环来创建线程，而Rust无法在提示信息中指出迭代过程中创建哪一个线程时出了问题。接下来，让我们使用在第15章讨论过的多重所有权方法来修复这一编译错误。

多线程与多重所有权

在第15章中，我们借助于智能指针Rc<T>提供的引用计数为单个值赋予了多个所有者。接下来，我们会尝试用相同的方法来解决当前的问题。示例16-14中的代码使用Rc<T>来包裹Mutex<T>，并在每次需要移动所有权至线程时克隆Rc<T>。另外，鉴于我们已经发现了错误的原因，所以下面的代码重新使用了for循环，并且依然为闭包使用了move关键字。

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>来允许多个线程持有Mutex<T>

再次编译代码，居然出现了另外一个错误！编译器可真能教会我们不少东西：

```
①error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>`:
    std::marker::Send` is not satisfied in `[closure@src/main.rs:11:36:
15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`  
--> src/main.rs:11:22  
|  
11 |         let handle = thread::spawn(move || {  
② |             ^^^^^^^^^^  
`std::rc::Rc<std::sync::Mutex<i32>>`  
cannot be sent between threads safely  
|  
= help: within `[closure@src/main.rs:11:36: 15:10  
counter:std::rc::Rc<std::sync::Mutex<i32>>]`, the trait  
`std::marker::Send` is  
not implemented for `std::rc::Rc<std::sync::Mutex<i32>>`  
= note: required because it appears within the type  
`[closure@src/main.rs:11:36: 15:10  
counter:std::rc::Rc<std::sync::Mutex<i32>>]  
= note: required by `std::thread::spawn`
```

这段错误提示信息的内容可真丰富！这里的重点在于第一段内嵌的错误：``std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads safely` ②。这意味着我们新创建的 `std::rc::Rc<std::sync::Mutex<i32>>` 类型无法安全地在线程间传递。这个错误的原因被指明在随后的错误描述中：the trait bound `Send` is not satisfied①，该类型不满足 trait 约束 `Send`。我们会在下一节中再来讨论 `Send`，它确保了我们在线程中使用的类型能够在并发环境下正常工作。

不幸的是，`Rc<T>` 在跨线程使用时并不安全。当 `Rc<T>` 管理引用计数时，它会在每次调用 `clone` 的过程中增加引用计数，并在克隆出的实例被丢弃时减少引用计数，但它并没有使用任何并发原语来保证修改计数的过程不会被另一个线程所打断。这极有可能导致计数错误并产生诡异的 bug，比如内存泄漏或值在使用时被莫名其妙地提前释放。我们需要的是一个行为与 `Rc<T>` 一致，且能够保证线程安全的引用计数类型。

原子引用计数Arc<T>

幸运的是，我们拥有一种被称为Arc<T>的类型，它既拥有类似于Rc<T>的行为，又保证了自己可以被安全地用于并发场景。它名称中的A代表着原子（atomic），表明自己是一个原子引用计数（atomically reference counted）类型。原子是一种新的并发原语，我们可以参考标准库文档中的std::sync::atomic部分来获得更多相关信息。你现在只需要知道：原子和原生类型的用法十分相似，并且可以安全地在多个线程间共享。

你也许会疑惑的是：为什么不将所有的原生类型实现为原子？标准库中的类型为什么不默认使用Arc<T>来实现呢？这是因为我们需要付出一定的性能开销才能够实现线程安全，而我们只应该在必要时为这种开销买单。如果你只是在单线程中对值进行操作，那么我们的代码可以因为无须原子的安全保障而运行得更快。

让我们回到示例中。由于Arc<T>与Rc<T>的接口完全一致，所以我们只需要简单地修改use代码行、对new的调用及对clone的调用即可。最终能够正常编译并运行的代码如示例16-15所示。

src/main.rs

```
use std::sync::{Mutex, Arc};
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>包裹Mutex<T>来实现多线程共享所有权

这段代码将会打印出下面的结果：

```
Result: 10
```

终于成功了！我们的计数器从0变为了10。虽然这个例子看上去非常平凡，但我们确实在这个过程中掌握了许多有关Mutex<T>与线程安全的知识。你可以使用本节中的程序结构去完成比计数更为复杂的工作。基于这个策略，你可以将计算分割为多个独立的部分，并将它们分配至不同的线程中，然后使用Mutex<T>来允许不同的线程更新计算结果中与自己有关的那一部分。

RefCell<T>/Rc<T>和Mutex<T>/Arc<T>之间的相似性

你可能会注意到，虽然counter本身不可变，但我们仍然能够获取其内部值的可变引用。这意味着，Mutex<T>与Cell系列类型有着相似的功能，它同样提供了内部可变性。我们在第15章使用了RefCell<T>来改变Rc<T>中的内容，而本节按照同样的方式使用Mutex<T>来改变Arc<T>中的内容。

另外还有一个值得注意的细节是，Rust并不能使你完全避免使用Mutex<T>过程中所有的逻辑错误。回顾第15章中讨论的内容，使用Rc<T>会有产生循环引用的风险。两个Rc<T>值在互相指向对方时会造成内存泄漏。与之类似，使用Mutex<T>也会有产生死锁（deadlock）的风险。当某个操作需要同时锁住两个资源，而两个线程分别持有其中一个锁并相互请求另外一个锁时，这两个线程就会陷入无穷尽的等待过程。如果你对死锁感兴趣，不妨试着编写一个可能导致死锁的Rust程序。然后，你还可以借鉴其他语言中规避互斥体死锁的策略，并在Rust中实现它们。标准库API文档的Mutex<T>和MutexGuard页面为此提供了许多有用的信息。

为了圆满地结束本章，我们会接着讨论Send与Sync这两个trait，并演示如何在自定义类型中使用它们。

使用Sync trait和Send trait对并发进行扩展

有趣的是，Rust语言本身内置的并发特性非常少。到目前为止，我们在本章讨论的几乎每一个并发特性都是标准库的一部分，而非语言本身内置的。你能够用来处理并发的解决方案也仅仅只局限于语言本身或标准库。我们既可以编写自己的并发功能，也可以使用他人写好的并发框架。

但不管怎样，仍然有两个并发概念被内嵌在了Rust语言中，它们是std::marker模块内的Sync trait与Send trait。

允许线程间转移所有权的Send trait

只有实现了Send trait的类型才可以安全地在线程间转移所有权。除了Rc<T>等极少数的类型，几乎所有的Rust类型都实现了Send trait：如果你将克隆后的Rc<T>值的所有权转移到了另外一个线程中，那么两个线程就有可能同时更新引用计数值并进而导致计数错误。因此，Rc<T>只被设计在单线程场景中使用，它也无须为线程安全付出额外的性能开销。

因此，Rust的类型系统与trait约束能够阻止我们意外地跨线程传递Rc<T>实例。当我们在示例16-14中试图执行这类操作时立马触发了编译时错误：the trait Send is not implemented for Rc<T>; Mutex<i32>。而当我们切换到实现了Send的Arc<T>后，那段代码就顺利地编译通过了。

任何完全由Send类型组成的复合类型都会被自动标记为Send。除了我们在第19章将会讨论到的裸指针，几乎所有的原生类型都满足Send约束。

允许多线程同时访问的Sync trait

只有实现了Sync trait的类型才可以安全地被多个线程引用。换句话说，对于任何类型T，如果&T（也就是T的引用）满足约束Send，那么T就是满足Sync的。这意味着T的引用能够被安全地传递至另外的线程中。与Send类似，所有原生类型都满足Sync约束，而完全由满足Sync的类型组成的复合类型也都会被自动识别为满足Sync的类型。

智能指针Rc<T>同样不满足Sync约束，其原因与它不满足Send约束类似。在第15章讨论过的RefCell<T>类型及Cell<T>系列类型也不满足Sync约束。RefCell<T>实现的运行时借用检查并没有提供有关线程安全的保证。而正如“在多个线程间共享Mutex<T>”一节中讨论的那样，智能指针Mutex<T>是Sync的，可以被多个线程共享访问。

手动实现Send和Sync是不安全的

当某个类型完全由实现了Send与Sync的类型组成时，它就会自动实现Send与Sync。因此，我们并不需要手动地为此种类型实现相关trait。作为标签trait，Send与Sync甚至没有任何可供实现的方法。它们仅仅被用来强化与并发相关的不可变性。

手动实现这些trait涉及使用特殊的不安全Rust代码。我们将在第19章讨论这一概念，目前你需要注意的是，当你构建的自定义并发类型包含了没有实现Send或Sync的类型时，你必须要非常谨慎地确保设计能够满足线程间的安全性要求。Rust官方网站中的*The Rustonomicon* 文档详细地讨论了此类安全性保证及如何满足安全性要求的具体技术。

总结

你还会在本书中看到更多有关并发的内容：第20章中的实践项目将在一个更加真实的场景中用到本章介绍的诸多概念。

Rust内置在语言中的并发特性相当少，几乎所有的并发解决方案都被实现为了不同的代码包。它们的迭代演化速度要远快于标准库，当你需要使用多线程时，请不要忘记到网络上搜索最新的、具有最高水准的第三方包。

Rust在标准库中提供了用于实现消息传递的通道，也提供了可以在并发场景中安全使用的智能指针：`Mutex<T>`与`Arc<T>`。类型系统与借用检查器则确保了使用这些组件的代码不会产生数据竞争或无效引用。只要我们的代码能够顺利通过编译，你就可以相信它能够正确地运行在多线程环境中，而不会出现其他语言中常见的那些难以解决的bug。并发编程在Rust中不再是一个令人望而生畏的概念：请无所畏惧地使用并发吧！

在接下来的章节中，我们会讨论一些符合语言习惯的建模方式及结构化解决方案，它们可以被用在那些逐渐变得臃肿的Rust项目中。另外，我们还会讨论面向对象编程的诸多常见概念，并研究它们和Rust风格之间的异同。

第17章

Rust的面向对象编程特性



面向对象编程（Object-Oriented Programming, OOP）是一种程序建模的方法。对象这个概念最初来源于20世纪60年代的Simula语言。随后，这一概念又催生出在对象中彼此传递消息的Alan Kay编程架构。面向对象编程正是Alan Kay在1967年为了描述这种架构所发明的一个专用术语。面向对象编程有很多种相互矛盾的定义，其中一部分定义能够把Rust归类为面向对象语言，而另外一部分定义则并不这样认为。我们会在本章讨论一些形成了普遍共识的面向对象特性，并学习如何在Rust语言的习惯下实现这些特性。然后，我们还会展示如何使用Rust来实现面向对象的设计模式，并讨论这一模式与常用的Rust实现方案之间的权衡取舍。

面向对象语言的特性

一门语言究竟需要包含哪些特性才能算作面向对象的编程语言呢？编程社区对此始终没有给出一个共识性的结论。Rust在开发过程中受到了众多编程范式的影响（例如在第13章讨论的函数式编程特性），这其中就包含了面向对象编程。我们认为面向对象的语言通常都包含以下这些特性：命名对象、封装及继承。让我们来看一看这些概念背后的含义，并研究一下Rust是否能够支持它们。

对象包含数据和行为

被称为“设计模式四人帮”的Erich Gamma、Richard Helm、Ralph Johnson及John Vlissides编写过一本名为《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*）的经典书籍，你可以在这本书中找到各式各样面向对象的设计模式。他们在书中给面向对象编程做出了这样的定义：

面向对象的程序由对象组成。对象包装了数据和操作这些数据的流程。这些流程通常被称作方法或操作。

基于这个定义，Rust是面向对象的：结构体和枚举包含数据，而impl块则提供了可用于结构体和枚举的方法。虽然带有方法的结构体和枚举并没有被称为对象，但它们确实满足“设计模式四人帮”对对象定义的所有功能。

封装实现细节

另外一个常常伴随着面向对象编程的思想便是封装（encapsulation）：调用对象的外部代码无法直接访问对象内部的实现细节，而唯一可以与对象进行交互的方法便是通过它公开的接口。使用对象的代码不应当深入对象的内部去改变数据或行为。封装使得开发者在修改或重构对象的内部实现时无须改变调用这个对象的外部代码。

我们曾经在第7章介绍过如何控制封装：我们可以使用pub关键字来决定代码中哪些模块、类型、函数和方法是公开的，而默认情况下其他所有内容都是私有的。例如，我们可以定义一个名为AveragedCollection的结构体，它的字段中包含了一个存储i32元素的动态数组。除此之外，为了避免在每次读取元素平均值的时候重复计算，我们添加了一个用于存储动态数组平均值的字段。换句话说，AveragedCollection会缓存计算出的平均值。这个结构体的定义如示例17-1所示。

src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

示例17-1：维护一个整数列表及集合平均值的AveragedCollection结构体

结构体本身被标记为pub来使其他代码可以使用自己，但其内部字段则依然保持私有。这一封装在本例中十分重要，因为我们希望在每次增加或删除值的时候平均值能够相应地得到更新。通过在结构体中实现add、remove和average方法便可以完成这些需求，如示例17-2所示。

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17-2：在AveragedCollection结构体中实现公共方法add、remove和average

公共方法add、remove和average是仅有的几个可以访问或修改AveragedCollection实例中数据的方法。当用户调用add方法向list中增加元素，或者调用remove方法从list中删除元素时，方法内部的实现都会调用私有方法update_average来更新average字段。

由于list和average字段是私有的，所以外部代码无法直接读取list字段来增加或删除其中的元素。一旦缺少了这样的封装，average字段便无法在用户私自更新list字段时保持同步更新。另外，用户可以通过average方法来读取average字段的值，却不能修改它。

因为结构体AveragedCollection封装了内部的实现细节，所以我们能够在未来轻松地改变数据结构等内部实现。例如，我们可以在list字段上使用HashSet<i32>代替Vec<i32>。只要add、remove和average这几个公共方法的签名保持不变，正在使用AveragedCollection的外部代码就无须进行任何修改；而假如我们将list声明为pub，那么就必然会失去这一优势：由于HashSet<i32>与Vec<i32>在增加或删除元素时使用的具体方法有所不同，因此如果直接修改list，那么外部代码将不得不发生变化。

如果封装是考察一门语言是否能够被算作面向对象语言的必要条件，那么Rust就是满足要求的。在不同的代码区域选择是否添加pub关键字可以实现对细节的封装。

作为类型系统和代码共享机制的继承

继承（inheritance）机制使得对象可以沿用另一个对象的数据与行为，而无须重复定义代码。

如果一门语言必须拥有继承才能算作面向对象语言，那么Rust就不是。你无法在Rust中定义一个继承父结构体字段和方法实现的子结构体。但不管怎样，如果你已经习惯了在编程中使用继承特性，那么你也可以根据使用继承时希望达成的效果来选择其他的Rust解决方案。

选择使用继承有两个主要原因。其一是实现代码复用：你可以为某个类型实现某种行为，并接着通过继承来让另一个类型直接复用这一实现。作为替代解决方案，你可以使用Rust中的默认 trait方法来进行代码共享。我们曾经在示例10-14中演示过这一特性，示例中的代码为Summary trait的summarize方法提供了一个默认实现。任何实现了Summary trait的类型都会自动拥有这个summarize方法，而无须添加额外的重复代码。这与继承十分相似，父类中的实现方法可以被继承它的子类所拥有。另外，我们还可以在实现Summary trait时选择覆盖summarize方法的默认实现，正如子类覆盖父类中的方法一样。

另外一个使用继承的原因与类型系统有关：希望子类型能够被应用在一个需要父类型的地方。这也就是所谓的多态（polymorphism）：如果一些对象具有某些共同的特性，那么这些对象就可以在运行时相互替换使用。

多态

许多人将“多态”视作“继承”的同义词。但实际上多态是一个更为通用的概念，它指代所有能够适应多种数据类型的代码。对于继承概念而言，这些类型就是所谓的子类。

你可以在Rust中使用泛型来构建不同类型的抽象，并使用trait约束来决定类型必须提供的具体特性。这一技术有时也被称作限定参数化多态（bounded parametric polymorphism）。

许多较为新潮的语言已经不太喜欢将继承作为内置的程序设计方案了，因为使用继承意味着你会在无意间共享出比所需内容更多的代码。子类并不应该总是共享父类的所有特性，但使用继承机制却会始终产生这样的结果，进而使程序设计缺乏灵活性。子类在继承的过程中有可能会引入一些毫无意义甚至根本就不适用于子类的方法。另外，某些语言强制要求子类只能继承自单个父类，这进一步限制了程序设计的灵活性。

考虑到这些弊端，Rust选择了trait对象来代替继承。让我们一起来看一看trait对象是如何在Rust中实现多态的。

使用trait对象来存储不同类型的值

我们曾经在第8章提及过动态数组的使用限制：它只能存储同一类型的元素。接着，我们还在示例8-10中实现了相应的变通方案，这个方案定义的枚举SpreadsheetCell同时包含了可以持有整数、浮点数和文本的变体。这意味着我们可以在每个单元格中存储不同的数据类型，并依然能够用一个动态数组来表示一整行单元格。只要可能会上出现的元素类型是固定的且能够在编译时准确得知，那么这就是一个非常不错的解决方案。

但是总有某些时候，我们希望用户能够在特定的应用场景下为这个类型的集合进行扩展。为了展示如何实现该特性，我们会在示例中创建一个图形用户界面（Graphical User Interface, GUI）工具。这个工具会遍历某个元素列表，并依次调用元素的draw方法来将其绘制到屏幕上，这是GUI工具最为基本的功能之一。我们会创建一个含有GUI库构架的gui包，并在包中提供一些可供用户使用的具体类型，比如Button或TextField等。此外，gui的用户也应当能够创建支持绘制的自定义类型，例如，某些开发者可能会添加Image，而另外某些开发者则可能会添加SelectBox。

本节的示例不会创建一个功能完善的GUI库，但它能够恰当地展示出各个部分会如何被组织到一起。虽然我们无法在编写库的时候预先推测出用户想要创建的类型，但我们知道gui需要记录一系列不同类型的值，并为这些不同类型的值逐一调用相同的draw方法。这个库不会关心调用draw方法后发生的具体事务，它只需要确保这些值都有一个可供调用的draw方法即可。

在那些支持继承的语言中，我们也许会定义出一个拥有draw方法的Component类。而其他类，比如Button、Image及SelectBox等，则需要继承Component类来获得draw方法。虽然它们可以选择覆盖draw方法来实现自定义行为，但框架会在处理过程中将它们全部视作Component

类型的实例，并以此调用draw方法。由于Rust没有继承功能，所以我们需要使用另外的方式去构建gui库，从而赋予用户扩展自定义类型的能力。

为共有行为定义一个trait

为了在gui中实现期望的行为，我们首先要定义一个拥有draw方法的Draw trait。接着，我们便可以定义一个持有trait对象的动态数组。trait对象能够指向实现了指定trait的类型实例，以及一个用于在运行时查找trait方法的表。我们可以通过选用一种指针，例如&引用或Box<T>智能指针等，并添加dyn关键字与指定相关trait来创建trait对象。至于为什么必须使用指针来创建trait对象，我们会在第19章的“动态大小类型和Sized trait”一节中讨论。trait对象可以被用在泛型或具体类型所处的位置。无论我们在哪里使用trait对象，Rust类型系统都会在编译时确保出现在相应位置上的值实现trait对象指定的trait。因此，我们无须在编译时知晓所有可能的具体类型。

我们曾经提到过，Rust有意避免将结构体和枚举称为“对象”，以便与其他语言中的对象概念区分开来。对于结构体或枚举而言，它们字段中的数据与impl块中的行为是分开的；而在其他语言中，数据和行为往往被组合在名为对象的概念中。trait对象则有些类似于其他语言中的对象，因为它也在某种程度上组合了数据与行为。但trait对象与传统对象不同的地方在于，我们无法为trait对象添加数据。由于trait对象被专门用于抽象某些共有行为，所以它没有其他语言中的对象那么通用。

示例17-3展示了如何定义一个拥有draw方法的Draw trait：

src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

示例17-3：Draw trait的定义

由于我们曾经在第10章介绍过如何定义trait，所以你应该对这段语法比较熟悉。接下来的示例17-4定义了一个持有components动态数

组的Screen结构体。这个动态数组的元素类型使用了新语法Box<dyn Draw>来定义trait对象，它被用来代表所有被放置在Box中且实现了Draw trait的具体类型。

src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

示例17-4：持有components字段的Screen结构体定义，components字段存储了实现了Draw trait的trait对象动态数组

这个Screen结构体还定义了一个名为run的方法，它会逐一调用components中每个元素的draw方法，如示例17-5所示。

src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

示例17-5：在Screen中实现的run方法会逐一调用components中每个元素的draw方法

我们同样可以使用带有trait约束的泛型参数来定义结构体，但它与此处示例代码的工作机制截然不同。泛型参数一次只能被替代为一个具体的类型，而trait对象则允许你在运行时填入多种不同的具体类型。例如，使用泛型参数与trait约束来定义Screen结构体，如示例17-6所示。

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17-6：使用泛型参数与trait约束定义的Screen结构体及run方法

为了使用新定义的Screen实例，我们被限制在list中存储完全由Button类型组成的列表，或完全由TextField类型组成的列表。如果你需要的仅仅是同质集合（homogeneous collection），那么使用泛型和trait约束就再好不过了，因为这段定义会在编译时被多态化以便使用具体类型。

另一方面，借助于在方法中使用trait对象，单个Screen实例持有的Vec可以同时包含Box<Button>与Box<TextField>。让我们来看一看trait对象背后的运行机制，以及它的运行时性能开销。

实现trait

现在，让我们来添加一些实现了Draw trait的具体类型。接下来的示例会提供Button类型的实现。需要再次重申的是，draw方法不会包含任何有意义的内容，因为我们并不打算编写一个完整的GUI库。但我们可以想象来猜测出Button结构体中可能会出现的字段，比如width、height与label等，如示例17-7所示。

src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // 实际绘制一个按钮的代码
    }
}
```

示例17-7：实现了Draw trait的Button结构体

Button中持有的width、height和label字段也许会不同于其他组件中的字段，例如，TextField类型就有可能在这些字段额外持有一个placeholder字段。每一个希望绘制在屏幕上的类型都应当实现Draw trait，并在draw方法中使用不同的代码来自定义具体的绘制行为，就像上面代码中的Button那样（这里省略了具体的绘制代码，因为它超出了本章的讨论范围）。除了实现Draw trait，我们的Button类型也许还会在另外的impl块中实现响应用户点击按钮时的行为，而这些方法并不适用于TextField等其他类型。

如果某个用户决定实现一个带有width、height和options字段的SelectBox结构体，那么他也同样可以为SelectBox类型实现Draw trait，如示例17-8所示。

src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // 实际绘制一个选择框的代码
    }
}
```

示例17-8：在另外某个依赖gui库的包中，定义一个实现了Draw trait的SelectBox结构体

用户已经可以在编写main函数时创建Screen实例了。另外，他们还可以使用Box<T>来生成SelectBox或Button的trait对象，并将这些trait对象添加到Screen实例中。接着，他们便可以运行Screen实例的run方法来依次调用所有组件的draw实现，如示例17-9所示。

src/main.rs

```
use gui::{Screen, Button};

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17-9：使用trait对象来存储实现了相同trait的不同类型值

我们在编写库的时候无法得知用户是否会添加自定义的SelectBox类型。但我们的Screen实现依然能够接收新的类型并顺利完成绘制工作，因为SelectBox实现了Draw trait及其draw方法。

run方法中的代码只关心值对行为的响应，而不在意值的具体类型。这一概念与动态类型语言中的“鸭子类型”（duck typing）十分相似：如果某个东西走起来像鸭子，叫起来也像鸭子，那么它就是一只鸭子！示例17-5在实现run方法的过程中并不需要知晓每个组件的具体类型，它仅仅调用了组件的draw方法，而不会去检查某个组件究竟是Button实例还是SelectBox实例。通过在定义动态数组components时指定Box<dyn Draw>元素类型，Screen实例只会接收那些能够调用draw方法的值。

使用trait对象与类型系统来实现“鸭子类型”有一个明显的优势：我们永远不需要在运行时检查某个值是否实现了指定的方法，或者担心出现“调用未定义方法”等运行时错误。Rust根本就不会允许这样的代码通过编译。

例如，我们可以尝试将String类型用作Screen的组件，如示例17-10所示。

src/main.rs

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };

    screen.run();
}
```

示例17-10：尝试使用一个没有实现指定trait的类型

由于String没有实现Draw trait，所以你会在编译时观察到如下所示的错误：

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not satisfied
--> src/main.rs:7:13
|
7 |         Box::new(String::from("Hi")),
|             ^^^^^^^^^^^^^^^^^^^^^^^^^ the trait gui::Draw is not
| implemented for `std::string::String`
|
= note: required for the cast to the object type `gui::Draw`
```

上面的错误提示信息指出了出现错误的原因：要么是我们给Screen传入了错误的类型，要么是我们没有为String实现对应的Draw trait。为了解决前者引起的错误，我们可以修改代码，将传入值替换为正确的类型。对于后者而言，我们则需要给String实现Draw trait，从而使Screen能够调用它的draw方法。

trait对象会执行动态派发

在第10章的“泛型代码的性能问题”一节中，我们曾经介绍过Rust编译器会在泛型使用trait约束时执行单态化：编译器会为每一个具体类型生成对应泛型函数和泛型方法的非泛型实现，并使用这些具体的类型来替换泛型参数。通过单态化生成的代码会执行静态派发（static dispatch），这意味着编译器能够在编译过程中确定你调用的具体方法。这个概念与动态派发（dynamic dispatch）相对应，动态派发下的编译器无法在编译过程中确定你调用的究竟是哪一个方法。在进行动态派发的场景中，编译器会生成一些额外的代码以便在运行时找出我们希望调用的方法。

Rust必然会在我们使用trait对象时执行动态派发。因为编译器无法知晓所有能够用于trait对象的具体类型，所以它无法在编译时确定需要调用哪个类型的哪个具体方法。不过，Rust会在运行时通过trait对象内部的指针去定位具体调用哪个方法。该定位过程会产生一些不可避免的运行时开销，而这并不会出现在静态派发中。动态派发还会阻止编译器内联代码，进而使得部分优化操作无法进行。但不管怎么样，动态派发确实能够为示例17-5中的代码带来额外的灵活性，它同时支撑了示例17-9中的代码。你可以基于这些对项目的考虑来决定是否使用trait对象。

trait对象必须保证对象安全

需要注意的是，你只能把满足对象安全（object-safe）的trait转换为trait对象。Rust采用了一套较为复杂的规则来决定某个trait是否对象安全。但在实际应用中，我们只需要关注其中两条规则即可。如果一个trait中定义的所有方法满足下面两条规则，那么这个trait就是对象安全的：

- 方法的返回类型不是Self。
- 方法中不包含任何泛型参数。

关键字Self是一个别名，它指向了实现当前trait或方法的具体类型。trait对象必须是对象安全的，因为Rust无法在我们使用trait对象时确定实现这个trait的具体类型究竟是什么。由于trait对象忘记了Self的具体类型，所以编译器无法在trait方法返回Self时使用原来的具体类型。同理，对于trait方法中的泛型参数而言，我们会在使用时将具体类型填入泛型所处的位置，这些具体类型会被视作当前类型的一部分。由于trait对象忘记了类型信息，所以我们无法确定被填入泛型参数处的类型究竟是哪一个。

标准库中的Clone trait就是一个不符合对象安全的例子。Clone trait中的clone方法拥有这样的签名：

```
pub trait Clone {  
    fn clone(&self) -> Self;  
}
```

由于String类型实现了Clone trait，所以我们在String实例上调用clone方法来获得一个新的String实例。类似地，我们也可以在Vec<T>实例上调用clone来获得新的Vec<T>实例。clone方法的签名需要知道Self究竟代表了哪一种具体类型，因为这是它作为结果返回的类型。

编译器会在你使用trait对象时，指出违反了对象安全规则的地方。以示例17-4中的代码为例，让我们在Screen结构体中存储实现了Clone trait的类型：

```
pub struct Screen {  
    pub components: Vec<Box<dyn Clone>>,  
}
```

编译这段代码会出现如下所示的错误：

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object  
--> src/lib.rs:2:5  
|  
2 |     pub components: Vec<Box<dyn Clone>>,  
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone` cannot be  
made into an object  
|= note: the trait cannot require that `Self : Sized`
```

上面的错误提示信息表明我们不能按照这种方式将Clone trait用作trait对象。如果你想了解更多有关对象安全的信息，请参考Rust官方网站的RFC 255文档。

实现一种面向对象的设计模式

状态模式（state pattern）是一种面向对象的设计模式，它的关键特点是，一个值拥有的内部状态由数个状态对象（state object）表达而成，而值的行为则随着内部状态的改变而改变。这种设计模式会通过状态对象来共享功能：相对应地，Rust使用了结构体与trait而不是对象与继承来实现这一特性。每个状态对象都会负责自己的行为并掌控自己转换为其他状态的时机。而持有状态对象的值则对状态的不同行为和状态转换的时机一无所知。

使用状态模式意味着在业务需求发生变化时我们不需要修改持有状态对象的值，或者使用这个值的代码。我们只需要更新状态对象的代码或增加一些新的状态对象，就可以改变程序的运转规则。接下来，我们会演示一个使用了状态模式的示例并讨论如何使用Rust来实现它。

这个示例会采用增量式的开发过程来实现一个用于发布博客的工作流程。这个博客最终的工作流程如下：

1. 在新建博客文章时生成一份空白的草稿文档。
2. 在草稿撰写完毕后，请求对这篇草稿状态的文章进行审批。
3. 在文章通过审批后正式对外发布。
4. 仅返回并打印成功发布后的文章，而不能意外地发布没有通过审批的文章。

除了上面描述的流程，任何其他对文章的修改行为都应当是无效的。例如，假设某人试图跳过审批过程来直接发布草稿状态的文章，那么我们的程序应当阻止这一行为并保持文章的草稿状态。

示例17-11展示了上述工作流程的代码实现。我们会在随后实现的blog代码包中提供该示例使用的各种API。由于暂时还没有实现blog代码包，所以这段代码还无法通过编译。

src/main.rs

```
use blog::Post;

fn main() {
    ① let mut post = Post::new();

    ② post.add_text("I ate a salad for lunch today");
    ③ assert_eq!("", post.content());

    ④ post.request_review();
    ⑤ assert_eq!("", post.content());

    ⑥ post.approve();
    ⑦ assert_eq!("I ate a salad for lunch today", post.content());
}
```

示例17-11：演示blog包预期行为的代码示例

我们希望用户通过Post::new来创建一篇新的文章草稿①。接着，我们还应该使用户可以在文章处于草稿状态时自由地将文字添加到文章中②。假如用户试图立即（也就是在发布前）获得文章中的内容，那么他什么也获取不到，因为文章依然处于草稿状态。出于演示的目的，我们在代码中添加了用于检查这一行为的assert_eq! 断言③。这种情况下一个非常不错的单元测试是，断言文章在处于草稿状态时调用content方法必然会返回一个空字符串，不过我们这里不会为这个例子编写测试。

接着，我们希望用户可以发出审批文章的请求④，而处于等待阶段的content方法则依然会在调用时返回空字符串⑤。当文章获得审批⑥并能够正式对外发布时，调用content方法则应当返回完整的文章内容⑦。

需要注意的是，用户与这个库进行交互时涉及的数据类型只有Post类型。这个类型会采用状态模式，它持有的值会是3种不同的文章状态对象中的一个：草稿、等待审批或已发布。Post类型会在内部管理状态与状态之间的变化过程。虽然状态变化的行为会在用户调用Post实例的对应方法时发生，但用户并不需要直接对这一过程进行管

理。另外，这同样意味着用户不会因为状态而出错，比如在审批未完成前发布文章。

定义Post并新建一个处于草稿状态下的新实例

现在，让我们开始来实现这个发布博文的代码库！显而易见，我们需要一个用来存储内容的公共结构体Post，所以我们开始定义这个结构体，并声明一个用于创建Post实例的公共关联函数new，如示例17-12所示。另外，我们还需要创建一个私有的State trait。Post类型会在私有的state字段中持有包裹在Option<T>内的trait对象Box<dyn State>。你会在稍后了解到Option<T>的必要性。

src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            ❶ state: Some(Box::new(Draft {})),
            ❷ content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

示例17-12：Post结构体的定义，以及用于创建Post实例的new函数、State trait和Draft结构体

State trait定义了所有文章状态共享的行为，状态Draft、PendingReview及Published都会实现State trait。目前，我们还没有为trait提供任何方法，只暂时给出了Draft这一个状态的定义，因为它是文章创建时所处的初始状态。

这段代码在创建Post实例时把它的state字段设置为了持有Box的Some值❶，而该Box则指向了Draft结构体的一个实例。这保证了任何

创建出来的Post实例都会从草稿状态开始，因为Post的state字段是私有的，所以用户无法采用其他状态来创建Post。Post::new函数将content字段设置为了新的空String❷。

存储文章内容的文本

示例17-11调用了一个名为add_text的方法来将传入的&str参数添加至文章中。之所以将这个功能作为方法来实现而不是通过pub关键字来直接暴露content字段，是因为我们需要控制用户访问content字段中的数据时的具体行为。add_text方法的实现过程一目了然，你只需要简单地把它添加至impl Post块中即可，如示例17-13所示。

src/lib.rs

```
impl Post {  
    // --略  
  
    --  
    pub fn add_text(&mut self, text: &str) {  
        self.content.push_str(text);  
    }  
}
```

示例17-13：add_text方法的实现使用户可以将字符串添加至文章的content中

由于调用add_text方法会修改对应的Post实例，所以该方法需要接收self的可变引用作为参数。接着，我们调用content中基于String类型的push_str方法来将text参数中的字符串添加到content字段中。由于该行为不依赖于文章当前所处的状态，所以它不是状态模式的一部分。虽然add_text方法没有与state字段进行交互，但它仍然是我们希望对外提供的行为之一。

确保草稿的可读内容为空

即便用户调用add_text方法为文章添加了一些内容，但只要文章处于草稿状态，我们就需要在用户调用content方法时返回一个空的字符串，正如示例17-11中❸行所演示的那样。目前，我们暂时将

content方法实现为最简单的形式：它永远返回一个空的字符串切片。我们会在后续实现状态转换的功能后再来修改这一方法。由于文章目前只能处于草稿状态，所以用户获取的文章内容也应该总是空的。这个临时的实现如示例17-14所示。

src/lib.rs

```
impl Post {
    // --略

    -- pub fn content(&self) -> &str {
        ""
    }
}
```

示例17-14：临时的Post::content方法会总是返回一个空的字符串切片

通过添加这个content方法，示例17-11中❸行之前的代码就可以如期运行了。

请求审批文章并改变其状态

接下来，我们会添加一个请求审批文章的功能。这个功能会将文章的状态从Draft变为PendingReview，如示例17-15所示。

src/lib.rs

```
impl Post {
    // --略

    ❶ pub fn request_review(&mut self) {
        ❷ if let Some(s) = self.state.take() {
            ❸ self.state = Some(s.request_review())
        }
    }

    trait State {
        ❹ fn request_review(self: Box<Self>) -> Box<dyn State>;
    }

    struct Draft {}
}
```

```

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        ⑤ Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        ⑥ self
    }
}

```

示例17-15：基于Post和State trait实现request_review方法

我们给Post添加了一个名为request_review的公共方法，它会接收self的可变引用①并调用当前state的request_review方法③。后面这个request_review方法会消耗当前的状态并返回一个新的状态。

我们为State trait添加了一个request_review方法④，所有实现了这个trait的类型都必须实现这个request_review方法。值得注意的是，我们选择了self: Box<Self>来作为方法的第一个参数，而不是self、&self或&mut self。这个语法意味着该方法只能被包裹着当前类型的Box实例调用，它会在调用过程中获取Box<Self>的所有权并使旧的状态失效，从而将Post的状态值转换为一个新的状态。

为了消耗旧的状态，request_review方法需要获取状态值的所有权。这也正是Post的state字段引入Option的原因：Rust不允许结构体中出现未被填充的值②。我们可以通过Option<T>的take方法来取出state字段的Some值，并在原来的位置留下一个None。这样做使我们能够将state的值从Post中移出来，而不单单只是借用它。接着，我们又将这个方法的结果赋值给了文章的state字段。

我们需要临时把state设置为None来取得state值的所有权，而不能直接使用self.state = self.state.request_review(); 这种代码。这可以确保Post无法在我们完成状态转换后再次使用旧的state值。

Draft实现的request_review方法需要在Box中包含一个新的PendingReview结构体实例⑤，这一状态意味着文章正在等待审批。PendingReview结构体同样实现了request_review方法，但它没有执行任何状态转移过程，仅仅是返回了自己⑥。对于一篇已经处在

PendingReview状态下的文章，发起审批请求并不会改变该文章的当前状态。

现在，你可以看到使用状态模式的优势了：无论state的值是什么，Post的request_review方法都不需要发生改变。每个状态都会负责维护自己的运行规则。

我们先暂时不去修改Post的content方法，仍然让它返回一个空的字符串切片。现在，我们的Post实例除了可以处于Draft状态，还能够被转变为PendingReview状态。示例17-11中的前11行代码现在可以正常工作了❸。

添加approve方法来改变content的行为

approve方法与request_review方法类似：它会执行状态的审批流程，并将state设置为当前状态审批后返回的值，如示例17-16所示。

src/lib.rs

```
impl Post {
    // --略

    --
    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 {
    // --略

    --
    fn approve(self: Box<Self>) -> Box<dyn State> {
        ❶ self
    }
}

struct PendingReview {}
```

```

impl State for PendingReview {
    // --略

    --
    fn approve(self: Box<Self>) -> Box<dyn State> {
        ② 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17-16：基于Post和State trait实现approve方法

这段代码为State trait添加了一个名为approve的方法。接着，我们创建了新的Published结构体来添加已发布状态，并为它实现State trait。

与request_review类似，为Draft实例调用approve方法会简单地返回self而不会产生任何作用①。PendingReview实例会在调用approve时返回一个包裹在Box内的Published结构体的新实例②。Published结构体同样实现了State trait，它的request_review和approve方法都只会返回它们本身，因为处于Published状态下的文章不应当被这些操作改变状态。

接下来，我们需要更新Post的content方法：它会在文章的状态为Published时返回content字段的值，并在其他情形下继续返回一个空的字符串切片，如示例17-17所示。

src/lib.rs

```

impl Post {
    // --略

    --
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(&self)
    }
    // --略
}

```

```
--  
}
```

示例17-17：更新Post的content方法，在该方法中委托调用State的content方法

因为我们希望使所有的规则在State相关结构体的内部实现，所以我们会调用state值的content方法，并将Post实例本身（也就是self）作为参数传入，最后将这个方法返回的值作为结果。

这段代码调用了Option的as_ref方法，因为我们需要的只是Option中值的引用，而不是它的所有权。由于state的类型是Option<Box<dyn State>>，所以我们在调用as_ref时得到Option<&Box<dyn State>>。如果这段代码中没有调用as_ref，那么就会导致编译时错误，因为我们不能将state从函数参数的借用&self中移出。

我们接着调用了unwrap方法。由于Post的具体实现保证了方法调用结束时的state总会是一个有效的Some值，所以我们可以确信调用unwrap不会发生panic。我们曾经在第9章的“当你比编译器拥有更多信息时”一节中讨论过类似的情形。即便编译器无法理解这样的逻辑，我们也可以知道state字段中的值永远不会出现None。

随后，我们又调用了&Box<dyn State>的content方法。由于解引用转换会依次作用于&与Box，所以我们最终调用的content方法来自实现了State trait的具体类型。这意味着我们需要在State trait的定义中添加content方法，并在这个方法的实现中基于当前状态来决定究竟返回哪些内容，如示例17-18所示。

src/lib.rs

```
trait State {  
    // --略  
  
    fn content<'a>(&self, post: &'a Post) -> &'a str {  
        ① ""  
    }  
}  
// --略  
--
```

```
struct Published {}

impl State for Published {
    // --略

    -->
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ② &post.content
    }
}
```

示例17-18：在State trait中添加content方法

我们为content方法添加了默认的trait实现，它会返回一个空的字符串切片①。这使得我们可以不必在Draft和PendingReview结构体中重复实现content。Published结构体会覆盖content方法并返回post.content的值②。

注意，我们需要在这个方法上添加相关的生命周期标注，正如在第10章讨论过的那样。这个方法的实现需要接收post的引用作为参数，并返回post中某一部分的引用作为结果，因此，该方法中返回值的生命周期应该与post参数的生命周期相关。

现在，所有工作均已完成，示例17-11中的代码现在可以正常工作了！我们按照发布博客工作流程的规则实现了一套状态模式。与规则相关的具体逻辑被封装在了状态对象中，而没有分散在整个Post代码内。

状态模式的权衡取舍

我们演示了如何使用Rust来实现面向对象的状态模式，它将一篇博客文章可能拥有的各种行为封装到了不同的状态中，而Post自身的方法则对这些行为一无所知。通过这种组织代码的方式，我们只需要查看一个地方便能知晓已发布文章的行为差异：Published结构体中的State trait的具体实现。

如果你采用其他的实现来替代状态模式，那么我们就可能需要在Post甚至是main函数的代码中使用match表达式来检查文章的状态，并根据状态执行不同的行为。当你希望了解文章处于已发布状态的具体行为时，采用这种实现就意味着我们不得不查看好几个不同的地方。

另外，这种代码的复杂度还会随着状态数量的增加而增加：每增加一个状态，所有的match表达式就需要对应地增加一个分支。

基于状态模式，我们可以免于在Post的方法或使用Post的代码中添加match表达式。当业务需要新增状态时，我们也只需要创建一个新的结构体并为它实现trait的各种方法即可。

使用状态模式实现的程序可以较为容易地扩展功能。为了更好地体验状态模式给维护代码带来的便捷，你可以试着自行完成下面这些需求：

- 添加reject方法，它可以将文章的状态从PendingReview修改为Draft。
- 为了将文章状态修改为Published，用户需要调用两次approve。
- 用户只有在文章处于Draft状态时才能够修改文本内容（提示：将改变内容的职责从Post转移至状态对象）。

状态模式的其中一个缺点在于：因为状态实现了状态间的转移，所以某些状态之间是相互耦合的。如果我们希望在PendingReview和Published之间添加一个Scheduled状态，那么我们就需要修改PendingReview中的代码来转移到Scheduled状态。假如我们能够在新增状态时避免修改PendingReview，那么虽然这会更加方便，但也意味着我们需要选用另外一种设计模式。

状态模式的另外一个缺点在于：我们需要重复实现一些代码逻辑。你也许会试着提供默认实现，让State trait的request_review和approve方法默认返回self；但这样的代码违背了对象安全规则，因为trait无法确定self的具体类型究竟是什么。如果我们希望将State当作trait对象来使用，那么它的方法就必须全部是对象安全的。

其他重复的地方还包括Post中的request_review和approve方法，它们在具体实现上具有高度的相似性。两个方法都将实现细节委托给了Option的state字段中值的同名方法实现，并将这个方法的返回结果设置为新的state值。如果Post中还有其他更多类似的方法，那么我们可以考虑使用宏来消除这种重复（参见第19章的“宏”一节）。

严格按照面向对象语言的定义来实现一套状态模式自然是可行的，但这并不能发挥出Rust的全部威力。接下来，我们会修改部分代码来使blog库可以将无效的状态和状态转移暴露为编译时错误。

将状态和行为编码成类型

我们会向你演示如何反思状态模式来获得一系列不同的取舍。我们会将状态编码为不同的类型，而不是完全封装状态与转移过程以使外部对其一无所知。结果，Rust的类型检查系统将会通过编译时错误来阻止用户使用无效的状态，比如在需要使用已发布文章的场合误用处于草稿状态的文章。

下面来看一下示例17-11中main函数的第一部分：

src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

我们仍然希望通过Post::new来创建出状态为草稿的新文章，并保留向文章中添加内容的能力。但我们不是让草稿的content方法返回一个空字符串，而是根本就不会为草稿提供content方法。基于这样的设计，用户会在试图读取草稿内容时得到方法不存在的编译错误。这使得我们不可能在产品中意外地暴露出草稿内容，因为这样的代码连编译都无法通过。示例17-19中包含了Post结构体和DraftPost结构体的定义，以及它们的方法实现。

src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    ① pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }
}
```

```

        }
    }

❷ pub fn content(&self) -> &str {
    &self.content
}
}

impl DraftPost {
❸ pub fn add_text(&mut self, text: &str) {
    self.content.push_str(text);
}
}

```

示例17-19：带有content方法的Post和不带content方法的DraftPost

Post 和 DraftPost 结构体都有一个用来存储文本的私有字段 content。由于我们将状态直接编码为了结构体类型，所以这两个结构体不再拥有之前的state字段。新的Post结构体将会代表一篇已发布的文章，它的content方法被用来返回内部content字段的值❷。

Post结构体仍然定义了自己的关联函数Post::new，但它现在会返回一个DraftPost实例，而不再是Post实例❶。由于content字段是私有的，且没有任何直接返回Post的函数，所以我们暂时无法创建出Post实例。

因为DraftPost结构体具有一个add_text方法，所以我们可以像以前一样为content添加文本❸，但是请注意，DraftPost根本就没有定义content方法！现在，程序能够保证所有文章都从草稿状态开始，并且处于草稿状态的文章无法对外展示自己的内容了。任何绕过这些限制的尝试都会导致编译时错误。

将状态转移实现为不同类型之间的转换

那么，我们应该如何得到一篇已发布的文章呢？我们依然希望草稿状态的文章能够在得到审批后发布，而一篇处于待审批状态的文章则不应该对外显示任何内容。让我们添加新的结构体PendingReviewPost来实现这一规则。新的代码还会在DraftPost中定义返回 PendingReviewPost 实例的 request_review 方法，并在 PendingReviewPost 中定义一个返回Post实例的 approve 方法，如示例 17-20 所示。

src/lib.rs

```
impl DraftPost {
    // --略

    --
    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,
        }
    }
}
```

示例17-20：可以通过调用DraftPost的request_review方法创建Pending ReviewPost，而PendingReviewPost的approve方法则能够把自己转换为已发布的Post

由于request_review和approve方法获取了self的所有权，所以它们会消耗DraftPost和PendingReviewPost实例，并分别将自己转换为PendingReviewPost和已发布的Post。通过这种写法，我们不可能在调用request_review方法后遗漏任何DraftPost实例，调用approve方法与此同理。尝试读取PendingReviewPost的内容同样会导致编译错误，因为它没有定义content方法。含有content方法的Post实例只能够通过PendingReviewPost的approve方法来获得，而用户只能通过调用DraftPost的request_review方法来获得PendingReviewPost实例。我们现在已经成功地将发布博客的工作流程编码到了类型系统中。

但是，我们不得不在main函数中做出一些小修改。因为request_review和approve方法会返回新的实例，而不是修改调用方法的结构体本身，所以我们需要添加一些let post =绑定来保存返回的新实例。接着，我们还删除了那些用来进行检查的断言，因为保证处于草稿状态或待审批状态的文章一定会返回空字符串不再有意义了：任何试图非法读取内容的操作都会导致编译错误。更新后的main函数代码如示例17-21所示。

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17-21：使用新的发布博客的工作流程实现来修改main函数

既然我们需要修改main函数来重新为post赋值，那么新的实现就不再是完全面向对象的状态模式了：状态之间的转换过程不再被完整地封装在Post实现中。但不管怎样，我们的目标是借助于类型系统和编译时类型检查彻底地杜绝无效状态！这将确保某些bug能够在进入生产环境之前暴露出来，例如显示出未经发布的文章等。

你可以试着按照示例17-20的思路来完成本节开始时为blog库提出的额外需求，并思考新版本代码中的设计模式会给实现需求带来哪些不一样的体验。需要注意的是，其中的部分需求也许已经随着设计的变化而解决了。

你应该可以观察到，Rust不仅能够实现面向对象的设计模式，它还可以支持其他更多的模式，比如将状态编码到类型系统等。不同的模式有着不同的取舍。尽管你可能会更加熟悉面向对象模式，但充分利用Rust的特性来重新思考问题依然能够带来不少好处，例如将部分错误暴露在编译期等。面向对象的经典模式并不总是Rust编程实践中的最佳选择，因为Rust具有所有权等其他面向对象语言所没有的特性。

总结

不管你是否将Rust视作面向对象的语言，你都可以在阅读完本章后学会如何使用trait对象来实现部分面向对象的特性。动态派发通过牺牲些许的运行时性能赋予了代码更多的灵活性。你可以利用这种灵活性来实现有助于改善代码可维护性的面向对象模式。由于Rust具有所有权等其他面向对象语言没有的特性，所以面向对象的模式仅仅是一种可用的选项，而并不总是最佳实践方式。

我们会在下一章学习模式，它是另外一个能够带来极强灵活性的Rust特性。虽然还没有完整地了解过模式，但我们已经在本书中多次接触过它了。现在让我们继续前进吧！

第18章

模式匹配



模式是Rust中一种用来匹配类型结构的特殊语法，它时而复杂，时而简单。将模式与match表达式或其他工具配合使用可以更好地控制程序流程。一个模式通常由以下组件组合而成：

- 字面量
- 解构的数组、枚举、结构体或元组
- 变量
- 通配符
- 占位符

这些组件可以描述我们将要处理的数据形状，而数据形状则可以被用来匹配值，进而判断出程序能否获得可供后续代码处理的正确数据。

模式被用来与某个特定的值进行匹配。如果模式与值匹配成功，那么我们就可以在代码中使用这个值的某些部分。回忆一下我们在第6章运用模式匹配编写的match表达式，尤其是那个“硬币分拣机”的例子：如果值与模式在形状上相符，那么我们就可以在随后的代码块中使用模式中命名的各种标识符；而如果不相符，那么模式对应的代码就会被简单地略过。

本章中的材料包含了所有与模式相关的内容。我们会讨论所有可以使用模式匹配的场景、不可失败模式与可失败模式之间的区别，以及代码中可能出现的各种模式匹配语法。通过阅读本章，你应当能够学会如何运用模式匹配来更加清晰地表达各种概念。

所有可以使用模式的场合

实际上，我们已经不知不觉地使用过许多次模式了，它会出现在相当多不同的Rust语法中！本节将系统地介绍所有可以使用模式的场合。

match分支

正如在第6章讨论的那样，模式可以被应用在match表达式的分支中。match表达式在形式上由match关键字、待匹配的值，以及至少一个匹配分支组合而成，而分支则由一个模式及匹配模式成功后应当执行的表达式组成：

```
match 值
{
    模式 => 表达式,
    模式 => 表达式,
    模式 => 表达式,
}
```

match表达式必须穷尽（exhaustive）匹配值的所有可能性。为了确保代码满足这一要求，我们可以在最后的分支处使用全匹配模式，例如，变量名可以被用来覆盖所有剩余的可能性，一个能够匹配任何值的变量名永远不会失败。

另外，还有一个特殊的_模式可以被用来匹配所有可能的值，且不将它们绑定到任何一个变量上，因此，这个模式常常被用作匹配列表中的最后一个分支。当你想要忽略所有未被指定的值时，_模式会非常

有用。我们将在本章的“忽略模式中的值”一节中更为详细地讨论`_`模式。

if let条件表达式

在第6章讨论如何使用`if let`表达式时，我们曾经将它当作只匹配单个分支的`match`表达式来使用。但实际上`if let`还能够添加一个可选的`else`分支，如果`if let`对应的模式没有匹配成功，那么`else`分支的代码就会得到执行。

另外，我们同样可以混合使用`if let`、`else if`及`else if let`表达式来进行匹配，如示例18-1所示。相较于单次只能将一个值与模式比较的`match`表达式来说，这种混合语法可以提供更多的灵活性，并且一系列`if let`、`else if`、`else if let`分支中的条件也不需要彼此相关。

示例18-1中的代码通过执行一系列的条件检查来决定需要使用的背景颜色。为了简单起见，我们为本例中的变量赋予了硬编码值，但真正的程序应当从用户的输入中获得这些值。

src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    ❶ if let Some(color) = favorite_color {
        ❷ println!("Using your favorite color, {}, as the background", color);
    } else if is_tuesday {
        ❸ println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        ❹ if age > 30 {
            ❺ println!("Using purple as the background color");
        } else {
            ❻ println!("Using orange as the background color");
        }
    } ❼ else {
        ❽ println!("Using blue as the background color");
    }
}
```

示例18-1：混合使用`if let`、`else if`、`else if let`和`else`

如果用户明确指定了一个偏爱的颜色①，那么我们就将它直接用作背景色②；否则，我们会继续判断当天是否是星期二③，并在条件满足时采用绿色作为背景色④。如果条件匹配再次失败，就进而判断用户给出的字符串是否能够被成功解析为数字⑤。当数字解析成功时，我们会根据数字的大小⑥选用紫色⑦或橙色⑧。如果以上所有条件均不满足⑨，那么我们就选择蓝色⑩作为背景色。

这种条件结构使我们可以支持较为复杂的需求。通过将本例中的硬编码值代入代码中执行，我们可以推断出这个示例最终打印出的结果为Using purple as the background color。

你也许注意到了，和match分支类似，if let分支能够以同样的方式对变量进行覆盖。if let Ok(age) = age这条语句⑤中引入了新的变量age来存储Ok变体中的值，而它覆盖了右侧的同名变量。这意味着我们必须把判断条件if age > 30⑥放置到匹配成功后执行的代码块中，而不能把这两个条件组合成if let Ok(age) = age && age > 30。因为覆盖了同名变量的age只有在花括号后的新作用域中才会变得有效。

与match表达式不同，if let表达式的不利之处在于它不会强制开发者穷尽值的所有可能性。即便我们省略了随后可选的else块⑨，并因此遗漏了某些需要处理的情形，编译器也不会在这里警告我们存在可能的逻辑性缺陷。

while let条件循环

条件循环while let的构造与if let十分类似，但它会反复执行同一个模式匹配直到出现失败的情形。示例18-2中的代码将一个动态数组用作栈，并使用while let依次将栈内的值按照与入栈相反的顺序打印出来。

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);
```

```
while let Some(top) = stack.pop() {  
    println!("{}", top);  
}
```

示例18-2：只要stack.pop()返回的值是Some变体，那么while let循环就会不断地进行打印

上面的示例会依次打印出3、2、1。其中的pop方法会试图取出动态数组的最后一个元素并将它包裹在Some(value)中返回。如果动态数组为空，则pop返回None。while循环会在pop返回Some时迭代执行循环体中的代码，并在pop返回None时结束循环。使用while let便可以将栈中的每个元素逐一弹出了。

for循环

正如我们在第3章介绍的那样，for循环是Rust代码中最为常用的循环结构，而你同样可以在for循环内使用模式。for语句中紧随关键字for的值就是一个模式，比如for x in y中的x。

示例18-3展示了如何在for循环中使用模式来解构元组。

```
let v = vec!['a', 'b', 'c'];  
  
for (index, value) in v.iter().enumerate() {  
    println!("{} is at index {}", value, index);  
}
```

示例18-3：在for循环中使用模式来解构元组

示例18-3中的代码会打印出如下所示的内容：

```
a is at index 0  
b is at index 1  
c is at index 2
```

上面的代码使用了enumerate方法来作为迭代器的适配器，它会在每次迭代过程中生成一个包含值本身及值索引的元组。例如，首次调用enumerate会产生元组(0, 'a')。当我们把这个值与模式(index, value)进行匹配时，index就会被赋值为0，而value则会被赋值为'a'，这也就是第一行输出中的内容。

let语句

我们只在前面的章节中明确地讨论过如何在match和if let表达式中使用模式，但实际上，我们在其他的许多语句（甚至是最基本的let语句）中也同样用到了模式。例如，考虑下面这个使用let来直接为变量赋值的语句：

```
let x = 5;
```

类似于这样的用法在本书中出现了数百次，虽然你可能没有意识到，但我们已经在上面的语句中使用到模式了！更正式的let语句的定义如下所示：

```
let PATTERN = EXPRESSION;
```

在类似于let x = 5; 的语句中，单独的变量名作为最朴素的模式被放于PATTERN对应的位置。Rust会将表达式与模式进行比较，并为所有找到的名称赋值。因此，在let x = 5; 这个示例中，x作为模式表达的含义是“将此处匹配到的所有内容绑定至变量x”。因为x就是整个模式本身，所以它实际上意味着“无论表达式会返回什么样的值，我们都可以将它绑定至变量x中”。

为了更清晰地理解let语句中的模式匹配，我们在示例18-4中演示了一条使用let模式来解构元组的语句。

```
let (x, y, z) = (1, 2, 3);
```

示例18-4：使用模式来解构元组并一次性创建出3个变量

我们在这个示例中使用模式来匹配一个元组。由于Rust在比较值(1, 2, 3)与模式(x, y, z)时发现它们是一一对应的，所以Rust会最终将1、2、3分别绑定至x、y、z上。你可以将这个元组模式理解为嵌套的3个独立变量模式。

如果模式中元素的数量与元组中元素的数量不同，那么整个类型就会匹配失败，进而导致编译错误。示例18-5中的代码试图用两个变量来解构拥有3个元素的元组，这当然是行不通的。

```
let (x, y) = (1, 2, 3);
```

示例18-5：一个错误的模式，其中变量的数量与元组中元素的数量不匹配

尝试编译这段代码会出现如下所示的错误：

```
error[E0308]: mismatched types
--> src/main.rs:2:9
|
2 |     let (x, y) = (1, 2, 3);
|          ^^^^^^ expected a tuple with 3 elements, found one with 2 elements
|
= note: expected type `({integer}, {integer}, {integer})`
        found type `(_, _)`
```

如果你需要忽略元组中的某一个或多个值，那么我们可以使用`_`或`..`语法，如随后的“忽略模式中的值”一节会提到的。假如你在模式中编写了过多的变量，那么我们只需要移除那些额外的变量并使变量的个数与元组中元素的个数相等，便可以保持类型的匹配。

函数的参数

函数的参数同样也是模式。示例18-6中的代码声明了一个名为`foo`的函数，它接收一个名为`x`的`i32`类型参数。你应该对这段代码中使用的语法相当熟悉了。

```
fn foo(x: i32) {
    // 在此编写函数代码
}
```

示例18-6：在参数中使用了模式的函数签名

签名中的`x`部分就是一个模式！与`let`语句类似，我们同样可以在函数参数中使用模式去匹配元组。示例18-7将我们传递给函数的元组拆分为了不同的值。

src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
```

```
let point = (3, 5);
print_coordinates(&point);
}
```

示例18-7：在参数中解构元组的函数

这段代码会打印出字符串Current location: (3, 5)。由于模式`&(x, y)`能够和值`&(3, 5)`匹配，所以`x`的值为3，`y`的值为5。

类似于函数的参数列表，我们同样可以在闭包的参数列表中使用模式。因为闭包和函数是非常类似的，正如我们在第13章讨论过的那样。

虽然你已经见识了许多不同的模式用法，但模式在不同上下文中的运作机制却不尽相同。在某些场合下，模式必须是不可失败的形式；而在另外一些场合下，模式却被允许是可失败的形式。我们会在接下来的一节中详细讨论这两个概念。

可失败性：模式是否会匹配失败

模式可以被分为不可失败（irrefutable）和可失败（refutable）两种类型。不可失败的模式能够匹配任何传入的值。例如，语句`let x = 5;`中的`x`便是不可失败模式，因为它能够匹配表达式右侧所有可能的返回值。可失败模式则可能因为某些特定的值而匹配失败。例如，表达式`if let Some(x) = a_value`中的`Some(x)`便是可失败模式。如果`a_value`变量的值是`None`而不是`Some`，那么表达式左侧的`Some(x)`模式就会发生不匹配的情况。

函数参数、`let`语句及`for`循环只接收不可失败模式，因为在这些场合下，我们的程序无法在值不匹配时执行任何有意义的行为。`if let`和`while let`表达式则只接收可失败模式，因为它们在被设计时就将匹配失败的情形考虑在内了：条件表达式的功能就是根据条件的成功与否执行不同的操作。

一般而言，我们不用在编写代码时过多地考虑模式的可失败性，但你还是需要熟悉可失败性这个概念本身，因为你需要能够识别出错误提示信息中有关它的描述，进而做出正确的应对。在遇到此类问题时，要么改变用于匹配的模式，要么改变被模式匹配的值的构造，这取决于代码期望实现的行为。

假如我们试图在需要不可失败模式的场合中使用可失败模式会发生些什么呢？示例18-8中的`let`语句使用了一个可失败的`Some(x)`模式。正如你可能猜想的那样，这段代码无法通过编译。

```
let Some(x) = some_option_value;
```

示例18-8：试图在`let`中使用一个可失败的模式

如果`some_option_value`的值是`None`，那么我们就无法成功地匹配模式`Some(x)`，这也意味着这个模式本身是可失败的。然而，`let`语句

只能接收一个不可失败模式，因为这段代码无法通过None值执行任何有效的操作。Rust会在编译时指出这一错误，即该代码试图在需要不可失败模式的场合中使用可失败模式：

```
error[E0005]: refutable pattern in local binding: `None` not covered
-->
|
3 | let Some(x) = some_option_value;
|     ^^^^^^ pattern `None` not covered
```

因为模式Some(x)无法（也不可能）覆盖表达式右侧的值的所有可能的情形，所以Rust产生了一个合理的编译错误。

为了修复上面示例中的问题，我们可以使用if let来代替涉及模式的那一部分let代码。新的代码能够在我们遇到模式不匹配时跳过花括号中的代码块，并给予程序一个合法的方式继续执行。将示例18-8中的代码修复后得到的代码如示例18-9所示。

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

示例18-9：将let替换为支持可失败模式的if let及对应的代码块

我们通过上面的方式给代码添加了一个合法的出口！你可以顺利地运行这段代码，尽管这意味着我们必须在此时使用可失败模式。假如我们在if let中使用了一个不可失败模式，那么这段代码是无法通过编译的，如示例18-10所示。

```
if let x = 5 {
    println!("{}", x);
};
```

示例18-10：尝试在if let表达式中使用一个不可失败模式

Rust会在编译错误中告诉我们，同时使用if let与不可失败模式没有任何意义：

```
error[E0162]: irrefutable if-let pattern
--> <anon>:2:8
|
2 | if let x = 5 {
|     ^ irrefutable pattern
```

因此，在match表达式的匹配分支中，除了最后一个，其他必须全部使用可失败模式，而最后的分支则应该使用不可失败模式，因为它需要匹配值的所有剩余的情形。Rust允许你在仅有一个分支的match表达式中使用不可失败模式，但这种语法几乎没有任何用处，它可以被简单的let语句所代替。

现在，你已经知道了所有可以使用模式的场合，以及不可失败模式与可失败模式之间的区别，那么接着就让我们来学习所有可以被用于构建模式的语法吧。

模式语法

在本书中，你可以看到许多不同种类的模式示例。本节则会系统地整理所有可用的模式语法，并讨论每一种语法的用武之地。

匹配字面量

正如在第6章介绍的那样，你可以直接使用模式来匹配字面量，如下所示：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这段代码会因为x的值是1而打印出one。当你需要根据特定的具体值来决定下一步行为时，就可以在代码中使用这一语法。

匹配命名变量

命名变量（named variable）是一种可以匹配任何值的不可失败模式，我们在本书中相当频繁地使用了这一模式。值得一提的是，当你在match表达式中使用命名变量时，情况可能会变得稍微有些复杂。由于match开启了一个新的作用域，所以被定义在match表达式内作为模式一部分的变量会覆盖掉match结构外的同名变量，正如覆盖其他普通变量一样。在示例18-11中，我们声明的变量x与y分别存储了Some(5)和10。接着，我们编写了一个match表达式来匹配x的值。请留意这个表达式分支中的模式及最后的println!语句，并试着在运行代码前预测一下最终的输出结果会是什么。

src/main.rs

```
fn main() {
    ❶ let x = Some(5);
    ❷ let y = 10;

    match x {
        ❸ Some(50) => println!("Got 50"),
        ❹ Some(y) => println!("Matched, y = {:?}", y),
        ❺ _ => println!("Default case, x = {:?}", x),
    }

    ❻ println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

示例18-11：match表达式的一个分支引入了一个覆盖变量y

让我们来逐步分析一下执行这段match表达式时究竟会发生些什么。由于第一个匹配分支❸的模式与x中的值❶不匹配，所以我们简单地跳过该分支即可。

第二个匹配分支❹的模式引入了新的变量y，它会匹配Some变体中携带的任意值。因为我们处在match表达式创建的新作用域中，所以这里的y是一个新的变量，而不是我们在程序起始处声明的那个存储了10的y❷。这个新的y的绑定能够匹配Some中的任意值，而x正是一个Some。因此，新的y会被绑定到x变量中Some内部的值。由于这个值是5，所以当前分支的表达式会在执行后打印出Matched, y = 5。

如果x不是Some(5)而是None，那么它会在前两个分支的模式匹配中匹配失败，进而与最后的那个下画线模式❺相匹配。由于我们没有在下画线模式的分支内引入x变量，所以这个表达式使用的x没有被任何变量所覆盖，它依然是外部作用域中的x。这个假想的match运行过程最终会打印出Default case, x = None。

match表达式创建出来的作用域会随着当前表达式的结束而结束，而它内部的y自然也无法幸免，代码最后的println! ❻会打印出at the end: x = Some(5), y = 10。

如果你希望在match表达式中比较外部的x与y，而不是引入新的覆盖变量，那么我们就需要使用带有条件的匹配守卫。本章随后的“使用匹配守卫添加额外条件”一节会详细介绍这一概念。

多重模式

你可以在match表达式的分支匹配中使用| 来表示或 (or) 的意思，它可以被用来一次性地匹配多个模式。例如，下面的代码在第一个分支中采用了或语法，即只要x的值匹配到了分支中的任意模式，当前分支中的代码就会得到执行：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这段代码会打印出one or two。

使用... 来匹配值区间

我们可以使用... 来匹配闭区间的值。例如，在下面的代码中，只要匹配的值处于这个区间，这个模式对应的分支就会得到执行：

```
let x = 5;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("something else"),
}
```

当上面代码中的x是1、2、3、4或5时，它就会被匹配到第一个分支。这个语法在表达类似含义时要比使用| 运算符更为方便。相比于1 ... 5，我们需要在使用| 时将模式修改为1 | 2 | 3 | 4 | 5。指定范围的代码要简短得多，特别是当你需要匹配1到1000之间任意数字的时候。

范围模式只被允许使用数值或char值来进行定义，因为编译器需要在编译时确保范围的区间不为空，而char和数值正是Rust仅有的可以判断区间是否为空的类型。

下面是一个使用char值区间的例子：

```
let x = 'c';

match x {
```

```
'a' ..= 'j' => println!("early ASCII letter"),
'k' ..= 'z' => println!("late ASCII letter"),
_ => println!("something else"),
}
```

由于Rust判断出c位于第一个模式的区间内，所以它最终会打印出early ASCII letter。

使用解构来分解值

我们可以使用模式来分解结构体、枚举、元组或引用，从而使用这些值中的不同部分。让我们分别来看一看这些用法。

解构结构体

示例18-12展示了一个由x和y两个字段组成的结构体Point。我们可以使用带有模式的let语句来拆分出这些字段。

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18-12：将结构体中的字段解构为独立的变量

这段代码创建了a和b两个变量，它们分别匹配了p结构体中字段x和y的值。这个例子说明模式中的变量名并不需要与结构体的字段名相同。但我们常常倾向于采用与字段名相同的变量名，因为这样可以方便我们记住哪一个变量来自哪一个字段。

采用与字段名相同的变量名在实践中相当常见，为了避免写出类似于let Point { x: x, y: y } = p这样冗余的代码，Rust允许我们在使用模式匹配分解结构体字段时采用一种较为简便的写法：你只需要列出结构体字段中的名称，模式就会自动创建出拥有相同名称的变量。示例

18-13中的代码与示例18-12中的代码拥有完全一致的行为，但它在let语句中使用模式创建的变量从a和b变为了x和y。

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18-13：使用结构体字段的简便写法来解构结构体字段

上述代码中创建的x和y变量会分别匹配到结构体变量p中的x与y字段。结果是，变量x和y中会存储结构体p中的值。

除了为所有字段创建变量，我们还可以在结构体模式中使用字面量来进行解构。这一技术使我们可以在某些字段符合要求的前提下再对其他字段进行解构。

示例18-14中展示的match表达式将Point值分为了3种不同的情况：位于x轴上的点（即y = 0）、位于y轴上的点（即x = 0），以及不在任意一个轴上的点。

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18-14：对模式中的字面量进行解构和匹配

通过在第一个分支中要求y字段匹配到字面量0，我们会匹配到所有位于x轴上的点。这个模式还同时创建了一个可以在随后代码块中使用

的x变量。

类似地，通过在第二个分支中要求x字段匹配到字面量0，我们能够匹配到所有位于y轴上的点，并为y字段的值创建变量y。由于第三个分支没有指定任何字面量，所以它可以匹配所有剩余的那些Point，并为x和y字段创建变量。

在本例中，因为p实例的x字段的值为0，所以它会匹配到match表达式的第二个分支，并最终打印出On the y axis at 7。

解构枚举

我们在本书前面的部分中已经完成过解构枚举的操作了，例如第6章中的示例6-5就曾经解构了Option<i32>。但仍有一个未被提及的细节需要注意：用于解构枚举的模式必须要对应枚举定义中存储数据的方式。下面来看一个例子，示例18-15中的代码使用了在示例6-2中的Message枚举，这里的match表达式会基于模式来解构枚举中的所有内部值。

src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    ❶ let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        ❷ Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        },
        ❸ Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        },
        ❹ Message::Write(text) => println!("Text message: {}", text),
        ❺ Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

```
        }
    }
}
```

示例18-15：解构含有不同种类值的枚举变体

执行这段代码最终会打印出Change the color to red 0, green 160, and blue 255。你可以试着改变msg的值❶来观察枚举实例匹配到其他分支时的代码运行情况。

对于不含有数据的枚举变体而言，比如Message::Quit❷，我们无法从其内部进一步解构出其他值。因此，针对这种变体的模式不会创建出任何变量，它只能被用于匹配字面量Message::Quit的值。

对类似于结构体的枚举变体而言，比如Message::Move❸，我们可以采用类似于匹配结构体的模式，在变体名后使用花括号包裹那些列出的字段变量名，而这些分解出来的部分同样可以被用于匹配分支的代码块中。示例18-15使用了与示例18-13相同的简写形式。

对类似于元组的枚举变体而言，比如在元组中持有一个元素的Message::Write ❹，以及在元组中持有3个元素的Message::ChangeColor❺，我们使用的模式与匹配元组时用到的模式非常相似。模式中的变量数目必须与目标变体中的元素数目完全一致。

解构嵌套的结构体和枚举

到目前为止，我们所有的示例都只匹配了单层的结构体或枚举，但匹配语法还可以被用于嵌套的结构中！

例如，我们可以重构示例18-15中的代码，从而使得ChangeColor消息同时支持RGB和HSV颜色空间，如示例18-16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 the color to red {}, green {}, and blue {}", r, g, b)
        },
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!("Change the color to hue {}, saturation {}, and value {}", h, s,
v)
        }
        _ => ()
    }
}

```

示例18-16：匹配嵌套的枚举

在这段match表达式中，第一个分支的模式匹配了含有Color::Rgb变体的Message::ChangeColor枚举变体，并绑定了3个内部的i32值；第二个分支的模式则匹配了含有Color::Hsv变体的Message::ChangeColor枚举变体。你可以在单个match表达式中指定这些较为复杂的条件，即便它们需要同时匹配两个不同的枚举类型。

解构结构体和元组

我们甚至可以按照某种更为复杂的方式来将模式混合、匹配或嵌套在一起。下面示例的元组中嵌套了结构体与其他元组，但我们依然能够同时解构出这个类型所有的基本元素：

```
let ((feet, inches), Point {x, y}) = ((3, 10), Point {x: 3, y: -10});
```

这段代码能够将复杂的类型值分解为不同的组成部分，以便使我们可以分别使用自己感兴趣的值。

基于模式的解构使我们可以较为方便地将值分解为不同部分，比如结构体中不同的字段，并相对独立地使用它们。

忽略模式中的值

在某些场景下忽略模式中的值是有意义的，例如在match表达式的最后一个分支中，代码可以匹配剩余所有可能的值而又不需要执行什么操作。有几种不同的方法可以让我们在模式中忽略全部或部分值：使用`_`模式、在另一个模式中使用`_`模式、使用以下画线开头的名称，或者使

用`..`来忽略值的剩余部分。让我们来逐一讨论一下这些模式的用法及目的。

使用`_`忽略整个值

我们曾经将下画线`_`作为通配符模式来匹配任意可能的值而不绑定值本身的内容。虽然`_`模式最常被用在`match`表达式的最后一个分支中，但实际上我们可以把它用于包括函数参数在内的一切模式中，如示例18-17所示。

src/main.rs

```
fn foo(_ : i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

示例18-17：在函数签名中使用`_`

上述代码会忽略传给第一个参数的值3，并打印出`This code only uses the y parameter: 4`。

当不再需要函数中的某个参数时，你可以修改函数签名来移除那个不会被使用的参数。忽略函数参数在某些情形下会变得相当有用。例如，假设你正在实现一个trait，而这个trait的方法包含了你不需要的某些参数。在这种情形下，可以借助忽略模式来避免编译器产生未使用变量的警告。

使用嵌套的`_`忽略值的某些部分

我们还可以在另一个模式中使用`_`来忽略值的某些部分。例如，在要运行的代码中，当你需要检查值的某一部分且不会用到其他部分时，就可以使用这一模式。示例18-18展示了一段用于管理选项的代码。该业务不允许用户覆盖某个设置中已经存在的自定义选项，但它允许用户重置选项并在选项未初始化时进行设置。

```
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18-18：当我们不需要使用Some中的值时，在模式中使用下画线来匹配Some变体

上述代码会打印出Can't overwrite an existing customized value与setting is Some(5)。虽然这段代码在第一个匹配分支中忽略了Some变体中的值，但我们可以通过它来确定setting_value和new_setting_value是否都是Some变体。我们希望在这种情形下保持setting_value的值不变，并打印出拒绝此次修改请求的理由。

剩余的所有情形（也就是setting_value或new_setting_value中任意一个是None时）都可以与第二个分支中的模式匹配。我们希望在这个分支中将setting_value的值修改为new_setting_value。

类似地，我们也可以在一个模式中多次使用下画线来忽略多个特定的值。示例18-19中的代码在匹配拥有5个元素的元组时忽略了其中第二个与第四个元素的值。

```

let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}

```

示例18-19：忽略一个元组中的多个部分

这段代码会忽略值4与16，并打印出Some numbers: 2, 8, 32。

通过以_开头的名称来忽略未使用的变量

Rust会在你创建出一个变量却又没有使用过它时给出相应的警告，因为这有可能是程序中的bug。但在某些场景下，创建一个暂时不会用到的变量仍然是合理的，比如进行原型开发时或开始一个新的项目时。为了避免Rust在这些场景中因为某些未使用的变量而抛出警告，我们可

以在这些变量的名称前添加下画线。示例18-20创建了两个未被使用的变量，但我们只会在运行这段代码时得到一条警告信息。

src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

示例18-20：以下画线开始的变量名可以避免触发变量未使用警告

编译这段代码会警告我们没有使用过变量y，但却不会警告我们没有使用那个以下画线开头的变量_x。

值得注意的是，使用以下画线开头的变量名与仅仅使用_作为变量名存在一个细微的差别：_x语法仍然将值绑定到了变量上，而_则完全不会进行绑定。为了展示这一区别的意义，我们在示例18-21中编写的代码会有意地触发编译错误。

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

println!("{:?}", s);
```

示例18-21：以下画线开头的未使用变量仍然绑定了值，这会导致值的所有权发生转移

由于变量s中的值被移动到了变量_s中，所以我们在随后使用s时违背所有权规则。然而，仅仅使用下画线本身则不会发生任何绑定操作。示例18-22中的代码可以顺利地通过编译，因为s的值不会被移动至—°。

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{:?}", s);
```

示例18-22：单独使用下画线不会绑定值

由于我们没有将s绑定到任何物体上，所以这段代码可以正常地运行。

使用..忽略值的剩余部分

对于拥有多个部分的值，我们可以使用`..`语法来使用其中的某一部分并忽略剩余的那些部分。这使我们不必为每一个需要忽略的值都添加对应的`_`模式来进行占位。`..`模式可以忽略一个值中没有被我们显式匹配的那些部分。示例18-23中有一个用来描述三维坐标的Point结构体。在这段代码的match表达式中，我们只需要使用三维坐标中的x字段，并可以忽略剩下的y和z字段。

```
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18-23：使用`..`忽略Point中除x之外的所有字段

这段代码在分支模式中首先列出了x变量，并接着列出了一个`..`模式。这种语法要比具体地写出`y: _`和`z: _`稍微便捷一些，尤其是当你需要操作某个拥有大量字段的结构体，却只需要使用其中的某一两个字段时。

`..`语法则会自动展开并填充任意多个所需的值。示例18-24演示了如何在元组中使用`..`。

src/main.rs

```
fn main() {  
    let numbers = (2, 4, 8, 16, 32);  
  
    match numbers {  
        (first, .., last) => {  
            println!("Some numbers: {}, {}", first, last);  
        },  
    }  
}
```

示例18-24：只匹配元组中的第一个值和最后一个值，而忽略其他值

这段代码使用first和last来分别匹配了元组中的第一个值和最后一个值，而它们之间的..模式则会匹配并忽略中间的所有值。

但不管怎么样，使用..必须不能出现任何歧义。如果模式中需要匹配的值或需要忽略的值是无法确定的，那么Rust就会产生一个编译时错误。示例18-25中的代码在使用..时便产生了歧义，因此它无法正常地通过编译。

src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        .., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

示例18-25：试图以存在歧义的方式使用..

编译这段代码会出现如下所示的错误：

```
error: `..` can only be used once per tuple or tuple struct pattern
--> src/main.rs:5:22
  |
5 |     .., second, ..) => {
  |           ^^^
```

Rust无法知道在匹配过程中需要在second之前和之后忽略多少个值。我们在这段代码中表达出的含义既有可能是忽略2，然后将second绑定到4，最后忽略8、16和24；也有可能是忽略2和4，然后将second绑定到8，最后忽略16和32；以此类推。由于变量名second在Rust中没有任何特殊的含义，所以我们会因为..模式中出现的歧义导致编译失败。

使用匹配守卫添加额外条件

匹配守卫（match guard）是附加在match分支模式后的if条件语句，分支中的模式只有在该条件被同时满足时才能匹配成功。相比于单独使用模式，匹配守卫可以表达出更为复杂的意图。

匹配守卫的条件可以使用模式中创建的变量。示例18-26中的match表达式在使用模式Some(x)的同时附带了额外的匹配守卫if $x < 5$ 。

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{} {}", x),
    None => (),
}
```

示例18-26：在模式上添加一个匹配守卫

上面的代码会在运行时打印出less than five: 4。num能够与第一个分支中的模式匹配成功，因为Some(4)与Some(x)匹配。随后的匹配守卫则会检查模式中创建的变量x是否小于5。由于num同样满足这一条件，所以我们最终执行了第一个分支中的代码。

假设num的值是Some(10)，那么第一个分支中的匹配守卫则无法成立，因为10大于5。Rust会接着进入第二个分支继续比较并最终匹配成功。因为第二个分支中没有匹配守卫，所以它能够匹配包含任意值的Some变体。

我们无法通过模式表达出类似于 $x < 5$ 这样的条件，匹配守卫增强了语句中表达相关逻辑的能力。

在示例18-11中，我们曾经提到匹配守卫可以用来解决模式中变量覆盖的问题。回忆一下当时的场景，那个match表达式在模式中创建了一个新的变量，而没有使用表达式外部的变量。这个新变量使我们无法在模式中使用外部变量的值来进行比较。示例18-27使用匹配守卫修复了这一问题。

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);
}
```

示例18-27：使用匹配守卫来测试Some变体内的值是否与外部变量相等

修改后的代码会打印出Default case, x = Some(5)。由于第二个分支的模式中没有引入新的变量y，所以随后的匹配守卫可以正常地在条件判断中使用外部变量y。这个分支使用了Some(n)而不是Some(y)来避免覆盖y变量。这里新创建出来的n变量不会覆盖外部的任何东西，因为match外部没有与n同名的变量。

匹配守卫if n == y不是一个模式，所以它不会引入新的变量。因为这个条件中的y就是来自表达式外部的y，而不是之前示例中覆盖后的y，所以我们才能够比较n和y的值是否相同。

我们同样可以在匹配守卫中使用或运算符|来指定多重模式。示例18-28演示了如何将匹配守卫及带有|的模式组合使用。另外，你还能从这个示例中观察到它们作用的优先级：if y匹配守卫同时作用于4、5及6这3个值，尽管你可能会误以为if y仅仅对6有效。

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

示例18-28：将匹配守卫与多重模式组合使用

第一个分支中的匹配条件要求x的值等于4、5或6，并且要求y为true。当你运行这段代码时，虽然x存储的4满足第一个分支中的模式要求，但却无法满足匹配守卫的条件if y，所以第一个分支的匹配失败。接着，代码会在第二个分支处匹配成功，并打印出no。之所以会出现这样的结果，是因为if条件对于整个模式4 | 5 | 6都是有效的，而不仅仅只针对最后的那个值6。换句话说，匹配守卫与模式之间的优先级关系是：

(4 | 5 | 6) if y => ...

而不是：

4 | 5 | (6 if y) => ...

运行示例代码便能够观察到它们之间的优先级关系：假如匹配守卫只对|分隔的最后一个值有效，那么第一个分支就应当匹配成功并打印

出yes。

@绑定

@运算符允许我们在测试一个值是否匹配模式的同时创建存储该值的变量。我们希望在示例18-29中测试Message::Hello的id字段是否在区间3...7中。另外，我们还想要将这个字段中的值绑定到变量id_variable上，以便我们在随后的分支代码块中使用它。这个绑定变量可以被命名为id，与字段同名，但本例出于演示目的使用了一个不同的名称。

```
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18-29：在模式中测试一个值的同时使用@来绑定它

运行该示例会打印出Found an id in range: 5。通过在3...7之前使用id_variable @，我们在测试一个值是否满足区间模式的同时可以捕获到匹配成功的值。

第二个分支仅仅在模式中指定了值的区间，而与这个分支关联的代码块中却没有一个包含了id字段的值的可用变量。id字段的值可以是10、11或12，但随后的代码却无法得知匹配值具体是哪一个。由于我们没有将这个值存储在某个变量中，所以该模式分支的代码无法使用id字段中的值。

最后一个分支的模式则指定了一个没有区间约束的变量，这个变量可以被用于随后的分支代码中，因为这里的代码使用了结构体字段简写

语法。这个分支的匹配没有像前两个分支那样对id值执行任何测试，因此所有的值都可以匹配这个模式。

@使我们可以在模式中测试一个值的同时将它保存到变量中。

总结

Rust中的模式可以有效地帮助我们区分不同种类的数据。当你在match表达式中使用模式时，Rust会在编译时检查你的分支模式是否覆盖了所有可能的情况，未满足条件的程序无法通过编译。let语句和函数参数中的模式使这些结构变得更加富有表达力，它们允许你将值解构为较小的部分并同时赋值给变量。我们可以根据不同需求来编写或简单或复杂的不同模式。

接下来，我们会在下一章讨论众多Rust特性里的一些高级特性。

第19章

高级特性



到目前为止，我们接触到了Rust编程语言中最常用的那些部分。在开始第20章的另外一个实践项目前，先让我们来聊一聊你可能会遇到的一些高级特性。你可以只把本章的内容当作参考，并在遇到相关的Rust未知问题时回来查阅。本章将要讨论的特性在一些特定场景下非常有用，虽然你很少会用到它们，但我们还是希望你能够了解Rust提供的所有功能。

本章将涉及以下内容：

- **不安全Rust:** 舍弃Rust的某些安全保障并负责手动维护相关规则。
- **高级 trait:** 关联类型、默认类型参数、完全限定语法 (fully qualified syntax)、超 trait (supertrait)，以及与 trait相关的newtype模式。
- **高级类型:** 更多关于newtype模式的内容、类型别名、never类型和动态大小类型。

- 高级函数和闭包： 函数指针与返回闭包。
- 宏： 在编译期生成更多代码的方法。

本章包含了一系列迷人的Rust特性，里面总会有你需要用到的东西！现在就让我们开始吧！

不安全Rust

到目前为止，我们讨论过的所有代码都拥有编译期强制实施的内存安全保障。然而，在Rust的内部还隐藏了另外一种不会强制实施内存安全保障的语言：不安全 Rust（unsafe Rust）。它与常规的Rust代码无异，但会给予我们一些额外的超能力。

不安全Rust之所以存在是因为静态分析从本质上讲是保守的。当编译器在判断一段代码是否拥有某种安全保障时，它总是宁可错杀一些合法的程序也不会接受可能非法的代码。尽管某些代码也许是安全的，但目前的Rust编译器却可能会做出相反的结论！在这种情况下，你可以使用不安全代码来告知编译器：“相信我，我知道自己在干些什么。”这样做的缺点在于你需要为自己的行为负责：如果你错误地使用了不安全代码，那么就可能会引发不安全的内存问题，比如空指针解引用等。

另外一个需要不安全Rust的原因在于底层计算机硬件固有的不安全性。如果Rust不允许进行不安全的操作，那么某些底层任务可能根本就完成不了。Rust作为一门系统语言需要能够进行底层编程，它应当允许你直接与操作系统打交道甚至是编写你自己的操作系统，这正是Rust语言的目标之一。现在，让我们来看一看不安全Rust能够完成哪些任务，我们又应当如何使用它。

不安全超能力

你可以在代码块前使用关键字unsafe来切换到不安全模式，并在被标记后的代码块中使用不安全代码。不安全Rust允许你执行4种在安全Rust中不被允许的操作，而它们也就是所谓的不安全超能力（unsafe superpower）。这些能力包括：

- 解引用裸指针。
- 调用不安全的函数或方法。
- 访问或修改可变的静态变量。
- 实现不安全trait。

需要注意的是，`unsafe`关键字并不会关闭借用检查器或禁用任何其他Rust安全检查：如果你在不安全代码中使用引用，那么该引用依然会被检查。`unsafe`关键字仅仅让你可以访问这4种不会被编译器进行内存安全检查的特性。因此，即便是身处于不安全的代码块中，你也仍然可以获得一定程度的安全性。

另外，`unsafe`并不意味着块中的代码一定就是危险的或一定会导致内存安全问题，它仅仅是将责任转移到了程序员的肩上，你需要手动确定`unsafe`块中的代码会以合法的方式访问内存。

人无完人，错误总是会在不经意间发生。但通过将这4种不安全操作约束在拥有`unsafe`标记的代码块中，我们可以在出现内存相关的错误时快速地将问题定位到`unsafe`代码块中。你应当尽可能地避免使用`unsafe`代码块，这会使你在最终排查内存错误时感激自己。

为了尽可能地隔离不安全代码，你可以将不安全代码封装在一个安全的抽象中并提供一套安全的API，我们会在随后学习不安全函数与方法时再来讨论这一技术。实际上，某些标准库功能同样使用了审查后的不安全代码，并以此为基础提供了安全的抽象接口。这种技术可以有效地防止`unsafe`代码泄漏到任何调用它的地方，因为使用安全抽象总会是安全的。

接下来，我们会依次介绍这4种不安全的超能力。同时，你也会在这个过程中发现一些在不安全代码上提供安全接口的抽象实例。

解引用裸指针

我们在第4章的“悬垂引用”一节中曾经提到过编译器会对引用的有效性做出保障。不安全Rust的世界里拥有两种类似于引用的新指针

类型，它们都被叫作裸指针（raw pointer）。与引用类似，裸指针要么是可变的，要么是不可变的，它们分别被写作`*const T`和`*mut T`。这里的星号是类型名的一部分而不是解引用操作。在裸指针的上下文中，不可变意味着我们不能直接对解引用后的指针赋值。

裸指针与引用、智能指针的区别在于：

- 允许忽略借用规则，可以同时拥有指向同一个内存地址的可变和不可变指针，或者拥有指向同一个地址的多个可变指针。
- 不能保证自己总是指向了有效的内存地址。
- 允许为空。
- 没有实现任何自动清理机制。

在避免Rust强制执行某些保障后，你就能够以放弃安全保障为代价来换取更好的性能，或者换取与其他语言、硬件进行交互的能力（Rust的保障在这些领域本来就不起作用）。

示例19-1演示了如何从一个引用中同时创建出不可变的和可变的裸指针。

```
let mut num = 5;  
  
let r1 = &num as *const i32;  
let r2 = &mut num as *mut i32;
```

示例19-1：通过引用创建裸指针

注意，我们没有在这段代码中使用`unsafe`关键字。你可以在安全代码内合法地创建裸指针，但不能在不安全代码块外解引用裸指针，稍后我们就会看到这一点。

在创建裸指针的过程中，我们使用了`as`来分别将不可变引用和可变引用强制转换为了对应的裸指针类型。由于这两个裸指针来自有效的引用，所以我们能够确认它们的有效性。但要记住，这一假设并不是对任意一个裸指针都成立。

接下来，我们会创建一个无法确定其有效性的裸指针。示例19-2创建了一个指向内存中任意地址的裸指针。尝试使用任意内存地址的行为是未定义的：这个地址可能有数据，也可能没有数据，编译器可能会通过优化代码来去掉该次内存访问操作，否则程序可能会在运行时出现段错误（segmentation fault）。我们一般不会编写出如示例19-2所示的代码，但它确实是合法的语句。

```
let address = 0x012345usize;
let r = address as *const i32;
```

示例19-2：创建一个指向任意内存地址的裸指针

刚刚提到，我们可以在安全代码中创建裸指针，但却不能通过解引用 裸指针来读取其指向的数据。为了使用*解引用裸指针，我们需要添加一个unsafe块，如示例19-3所示。

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

示例19-3：在unsafe块中解引用裸指针

创建一个指针并不会产生任何危害，只有当我们试图访问它指向的值时才可能因为无效的值而导致程序异常。

值得注意的是：我们在示例19-1和示例19-3中同时创建出了指向同一个内存地址num的*const i32和*mut i32裸指针。如果我们尝试同时创建一个指向num的可变引用和不可变引用，那么就会因为Rust的所有权规则而导致编译失败。但在使用裸指针时，我们却可以创建同时指向同一地址的可变指针和不可变指针，并能够通过可变指针来修改数据。这一修改操作会导致潜在的数据竞争，请在使用时多加小心！

既然存在这些危险，那么为什么我们还需要使用裸指针呢？它的一个主要用途便是与C代码接口进行交互，我们会在下一节“调用不安全函数或方法”中看到。另外，它还可以被用来构造一些借用检查器

无法理解的安全抽象。随后我们会先讨论不安全函数，并接着展示一个使用不安全代码块的安全抽象实例。

调用不安全函数或方法

第二种需要使用不安全代码块的操作便是调用不安全函数（unsafe function）。除了在定义前面要标记unsafe，不安全函数或方法看上去与正常的函数或方法几乎一模一样。此处的unsafe关键字意味着我们需要在调用该函数时手动满足并维护一些先决条件，因为Rust无法对这些条件进行验证。通过在unsafe代码块中调用不安全函数，我们向Rust表明自己确实理解并实现了相关的约定。

下面的示例中有一个不执行任何操作的dangerous函数：

```
unsafe fn dangerous() {}  
  
unsafe {  
    dangerous();  
}
```

我们必须在单独的unsafe代码块中调用dangerous函数。假设你试图在unsafe代码块外调用它则会产生如下所示的错误：

```
error[E0133]: call to unsafe function requires unsafe function or block  
-->  
|  
4 |     dangerous();  
|     ^^^^^^^^^^ call to unsafe function
```

通过在调用dangerous的代码外插入unsafe代码块，我们向Rust表明自己已经阅读过函数的文档，能够理解正确使用它的方式，并确认满足了它所要求的约定。

因为不安全函数的函数体也是unsafe代码块，所以你可以在一个不安全函数中执行其他不安全操作而无须添加额外的unsafe代码块。

创建不安全代码的安全抽象

函数中包含不安全代码并不意味着我们需要将整个函数都标记为不安全的。实际上，将不安全代码封装在安全函数中是一种十分常见的抽象。下面，让我们通过示例来观察标准库中使用了不安全代码的

`split_at_mut`函数，并思考应该如何实现它。这个安全方法被定义在可变切片上：它接收一个切片并从给定的索引参数处将其分割为两个切片。示例19-4展示了`split_at_mut`的相关使用方法。

```
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19-4：使用安全的`split_at_mut`函数

我们无法仅仅使用安全Rust来实现这个函数。示例19-5中展示了一个可能的尝试，但它却无法通过编译。为了简单起见，我们将`split_at_mut`实现为函数而不是方法，并只处理特定类型*i32*的切片而非泛型T的切片。

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    assert!(mid <= len);
    (&mut slice[..mid],
     &mut slice[mid..])
}
```

示例19-5：尝试仅仅使用安全Rust来实现`split_at_mut`

这个函数会首先取得整个切片的长度，并通过断言检查给定的参数是否小于或等于当前切片的长度。如果给定的参数大于切片的长度，那么函数就会在尝试使用该索引前触发panic。

接着，我们会返回一个包含两个可变切片的元组：一个从原切片的起始位置到mid索引的位置，另一个则从mid索引的位置到原切片的末尾。

尝试编译示例19-5中的代码会出现如下所示的错误：

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
-->
|
6 |     (&mut slice[..mid],
|         ----- first mutable borrow occurs here
7 |     &mut slice[mid..])
```

```
8 | } ^^^^^^ second mutable borrow occurs here
| - first borrow ends here
```

Rust的借用检查器无法理解我们正在借用一个切片的不同部分，它只知道我们借用了两次同一个切片。借用一个切片的不同部分从原理上来讲应该是没有任何问题的，因为两个切片并没有交叉的地方，但Rust并没有足够智能到理解这些信息。当我们能够确定某段代码的正确性而Rust却不能时，不安全代码就可以登场了。

示例19-6展示了如何使用unsafe代码块、裸指针及一些不安全函数来实现split_at_mut。

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
① let len = slice.len();
② let ptr = slice.as_mut_ptr();

③ assert!(mid <= len);

④ unsafe {
    ⑤ (slice::from_raw_parts_mut(ptr, mid),
        ⑥ slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
}
}
```

示例19-6：在实现split_at_mut函数时使用不安全代码

回忆一下第4章的“其他类型的切片”一节中的内容，切片由一个指向数据的指针与切片长度组成。我们可以使用len方法来得到切片的长度①，并使用as_mut_ptr方法来访问切片包含的裸指针②。在本例中，由于我们使用了可变的i32类型的切片，所以as_mut_ptr会返回一个类型为*mut i32的裸指针。而这个指针被存储到了变量ptr中。

随后的断言语句保证了mid索引一定会位于合法的切片长度内③。继续往下的部分就是不安全代码④：slice::from_raw_parts_mut函数接收一个裸指针和长度来创建一个切片。这里的代码使用该函数从ptr处创建了一个拥有mid个元素的切片⑤，接着我们又在ptr上使用mid作为偏移量参数调用offset方法得到了一个从mid处开始的裸指针，并基于它创建了另外一个起始于mid处且拥有剩余所有元素的切片⑥。

由于函数slice::from_raw_parts_mut接收一个裸指针作为参数并默认该指针的合法性，所以它是不安全的。裸指针的offset方法也是

不安全的，因为它必须默认此地址的偏移量也是一个有效的指针。因此，我们必须在unsafe代码块中调用slice::from_raw_parts_mut和offset函数。通过审查代码并添加mid必须小于等于len的断言，我们可以确认unsafe代码块中的裸指针都会指向有效的切片数据且不会产生任何的数据竞争。这便是一个恰当的unsafe使用场景。

因为代码没有将split_at_mut函数标记为unsafe，所以我们可以安全Rust中调用该函数。我们创建了一个对不安全代码的安全抽象，并在实现时以安全的方式使用了unsafe代码，因为它仅仅创建了指向访问数据的有效指针。

与之相反，示例19-7中对slice::from_raw_parts_mut的调用则很有可能导致崩溃。这段代码试图用一个随意的内存地址来创建拥有10000个元素的切片。

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let slice : &[i32] = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

示例19-7：基于任意内存地址创建一个切片

由于我们不拥有这个随意地址的内存，所以就无法保证这段代码的切片中包含有效的i32值，尝试使用该slice会导致不确定的行为。

使用extern函数调用外部代码

在某些场景下，你的Rust代码可能需要与另外一种语言编写的代码进行交互。Rust为此提供了extern关键字来简化创建和使用外部函数接口（Foreign Function Interface, FFI）的过程。FFI是编程语言定义函数的一种方式，它允许其他（外部的）编程语言来调用这些函数。

示例19-8集成了C标准库中的abs函数。任何在extern块中声明的函数都是不安全的。因为其他语言并不会强制执行Rust遵守的规则，而Rust又无法对它们进行检查，所以在调用外部函数的过程中，保证安全的责任也同样落在了开发者的肩上。

src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

示例19-8：声明并调用在另外一种语言中定义的extern函数

这段代码在extern "C"块中列出了我们想要调用的外部函数名称及签名，其中的"C"指明了外部函数使用的应用二进制接口（Application Binary Interface, ABI）：它被用来定义函数在汇编层面的调用方式。我们使用的"C"ABI正是C编程语言的ABI，它也是最常见的ABI格式之一。

在其他语言中调用Rust函数

我们同样可以使用extern来创建一个允许其他语言调用Rust函数的接口。但不同于使用extern标注的代码块，我们需要将extern关键字及对应的ABI添加到函数签名的fn关键字前，并为该函数添加#[no_mangle]注解来避免Rust在编译时改变它的名称。Mangling是一个特殊的编译阶段，在这个阶段，编译器会修改函数名称来包含更多可用于后续编译步骤的信息，但通常也会使得函数名称难以阅读。几乎所有程序语言的编译器都会以稍微不同的方式来改变函数名称，为了让其他语言正常地识别Rust函数，我们必须禁用Rust编译器的改名功能。

在下面的示例中，我们编写了一个可以在编译并链接后被C语言代码访问的call_from_c函数：

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

这一类型的extern功能不需要使用unsafe。

访问或修改一个可变静态变量

到目前为止，我们一直都没有讨论全局变量（global variable）。Rust确实是支持全局变量的，但在使用它们的过程中可能会因为Rust的所有权机制而产生某些问题。如果两个线程同时访问一个可变的全局变量，那么就会造成数据竞争。

在Rust中，全局变量也被称为静态（static）变量。示例19-9声明并使用了一个静态变量，它的值是一个字符串切片。

src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

示例19-9：定义并使用一个不可变静态变量

静态变量类似于第3章的“变量与常量之间的不同”一节中讨论过的常量。静态变量的名称会约定俗成地被写作SCREAMING_SNAKE_CASE的形式，并且必须标注变量的类型（也就是本例中的`& static str`）。静态变量只能存储拥有`static`生命周期的引用，这意味着Rust编译器可以自己计算出它的生命周期而无须手动标注。访问一个不可变静态变量是安全的。

常量和不可变静态变量看起来可能非常相似，但它们之间存在一个非常微妙的区别：静态变量的值在内存中拥有固定的地址，使用它的值总是会访问到同样的数据。与之相反的是，常量则允许在任何被使用到的时候复制其数据。

常量和静态变量之间的另外一个区别在于静态变量是可变的。需要注意的是，访问和修改可变的静态变量是不安全的。示例19-10展示了如何声明、访问与修改一个名为COUNTER的可变静态变量。

src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
```

```

        unsafe {
            COUNTER += inc;
        }
    }

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}

```

示例19-10：从一个可变静态变量中读或写都是不安全的

和正常变量一样，我们使用`mut`关键字来指定静态变量的可变性。任何读写`COUNTER`的代码都必须位于`unsafe`代码块中。上述代码可以顺利地通过编译并如期打印出`COUNTER: 3`，因为它是单线程的。当有多个线程同时访问`COUNTER`时，则可能会出现数据竞争。

在拥有可全局访问的可变数据时，我们很难保证没有数据竞争发生，这也是Rust会将可变静态变量当作不安全的原因。你应当尽可能地使用在第16章讨论过的并发技术或线程安全的智能指针，从而使编译器能够对线程中的数据访问进行安全性检查。

实现不安全trait

最后一个只能在`unsafe`中执行的操作是实现某个不安全trait。当某个trait中存在至少一个方法拥有编译器无法校验的不安全因素时，我们就称这个trait是不安全的。你可以在trait定义的前面加上`unsafe`关键字来声明一个不安全trait，同时该trait也只能在`unsafe`代码块中实现，如示例19-11所示。

```

unsafe trait Foo {
    // 某些方法
}

unsafe impl Foo for i32 {
    // 对应的方法实现
}

```

示例19-11：定义和实现一个不安全trait

通过使用`unsafe impl`，我们向Rust保证我们会手动维护好那些编译器无法验证的不安全因素。

回忆一下我们在第16章的“使用`Sync trait`和`Send trait`对并发进行扩展”一节中讨论过的`Sync`与`Send`标签`trait`：当我们的类型完全由实现了`Send`与`Sync`的类型组成时，编译器会自动为它实现`Send`与`Sync`。假如我们的类型包含了某个没有实现`Send`或`Sync`的字段（比如裸指针等），而又希望把这个类型标记为`Send`或`Sync`，那么我们就必须使用`unsafe`。Rust无法验证我们的类型是否能够安全地跨线程传递，或安全地从多个线程中访问。因此，我们需要手动执行这些审查并使用`unsafe`关键字来实现这些`trait`。

使用不安全代码的时机

使用`unsafe`来执行刚刚讨论过的4种操作（超能力）并没有什么问题，执行的时候甚至都不用皱眉头。但是由于它们缺少编译器提供的强制内存安全保障，所以想要始终保持`unsafe`代码的正确性也并不是一件简单的事情。你可以在拥有充足理由时使用`unsafe`，并在出现问题时通过显式标记的`unsafe`关键字来较为轻松地定位到它们。

高级trait

虽然你早在第10章的“trait：定义共享行为”一节中就正式接触过了trait，但我们当时忽略了一些较为高级的细节。在对Rust有了更多的了解后，现在是时候来深入地研究它们了。

在trait的定义中使用关联类型指定占位类型

关联类型（associated type）是trait中的类型占位符，它可以被用于trait的方法签名中。trait的实现者需要根据特定的场景来为关联类型指定具体的类型。通过这一技术，我们可以定义出包含某些类型的trait，而无须在实现前确定它们的具体类型是什么。

我们在本章讨论的大部分高级主题都较少被用到，但关联类型却处于某种中间状态：虽然它比本书中介绍的其他特性用得更少一些，但却比本章中出现的诸多高级特性更为常用。

标准库中的Iterator就是一个带有关联类型的trait示例，它拥有一个名为Item的关联类型，并使用该类型来替代迭代中出现的值类型。我们在第13章的“Iterator trait和next方法”一节中曾经提到过，Iterator trait的定义如示例19-12所示。

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

示例19-12：含有关联类型Item的Iterator trait的定义

这里的类型Item是一个占位类型，而next方法的定义则表明它会返回类型为Option<Self::Item>的值。Iterator trait的实现者需要

为Item指定具体的类型，并在实现的next方法中返回一个包含该类型值的Option。

关联类型看起来与泛型的概念有些类似，后者允许我们在不指定具体类型的前提下定义函数。那么我们为什么需要使用关联类型呢？

让我们通过一个在第13章出现过的例子来观察它们两者之间的区别，这个例子在Counter结构体上实现了Iterator trait。在示例13-21中，我们将Item类型指定为了u32：

src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --略
    }
}
```

这里的语法似乎和泛型语法差不多，那么我们为什么不直接使用泛型来定义Iterator trait呢？如示例19-13所示。

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

示例19-13：一个使用泛型的假想Iterator trait定义

其中的区别在于，如果我们使用了示例19-13中的泛型版本，那么就需要在每次实现该trait的过程中标注类型；因为我们既可以实现Iterator<String> for Counter，也可以实现其他任意的迭代类型，从而使得Counter可以拥有多个不同版本的Iterator实现。换句话说，当trait拥有泛型参数时，我们可以为一个类型同时多次实现trait，并在每次实现中改变具体的泛型参数。那么当我们在Counter上使用next方法时，也必须提供类型标注来指明想要使用的Iterator实现。

借助关联类型，我们不需要在使用该trait的方法时标注类型，因为我们不能为单个类型多次实现这样的trait。对于示例19-12中使用了关联类型的trait定义，由于我们只能实现一次impl Iterator for Counter，所以Counter就只能拥有一个特定的Item类型。我们不需要

在每次调用Counter的next方法时来显式地声明这是一个u32类型的迭代器。

默认泛型参数和运算符重载

我们可以在使用泛型参数时为泛型指定一个默认的具体类型。当使用默认类型就能工作时，该trait的实现者可以不用再指定另外的具体类型。你可以在定义泛型时通过语法<PlaceholderType=ConcreteType>来为泛型指定默认类型。

这个技术常常被应用在运算符重载中。运算符重载（operator overloading）使我们可以在某些特定的情形下自定义运算符（比如+）的具体行为。

虽然Rust不允许你创建自己的运算符及重载任意的运算符，但你可以实现std::ops中列出的那些trait来重载一部分相应的运算符。例如，在示例19-14中，我们为Point结构体实现的Add trait重载了+运算符，它允许代码对两个Point实例执行加法操作。

src/main.rs

```
use std::ops::Add;

#[derive(Debug, 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19-14：通过实现Add trait来重载Point实例的+运算符

add方法将两个Point实例的x与y分别相加来创建出一个新的Point。Add trait拥有一个名为Output的关联类型，它被用来确定add方法的返回类型。

这里的Add trait使用了默认泛型参数，它的定义如下所示：

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

你应该对这段代码中的大部分语法都较为熟悉，它定义的trait中带有一个方法和一个关联类型。那段新的语法RHS=Self就是所谓的默认类型参数（default type parameter）。泛型参数RHS（也就是“right-hand side”的缩写）定义了add方法中rhs参数的类型。假如我们在实现Add trait的过程中没有为RHS指定一个具体的类型，那么RHS的类型就会默认为Self，也就是我们正在为其实现Add trait的那个类型。

因为我们希望将两个Point实例相加，所以代码在为Point实现Add时使用了默认的RHS。现在让我们来看另外一个例子，这个新的例子会在实现Add trait时自定义RHS的类型而不使用其默认类型。

这里有两个以不同单位存放值的结构体：Millimeters与Meters。我们希望可以将毫米表示的值与米表示的值相加，并在Add的实现中添加正确的转换计算。我们可以为Millimeters实现Add，并将Meters作为RHS，如示例19-15所示。

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19-15：为Millimeters实现Add trait，从而使Millimeters和Meters可以相加

为了将Millimeters和Meters的值加起来，我们指定impl Add< Meters >来设置RHS类型参数的值，而没有使用默认的Self。

默认类型参数主要被用于以下两种场景：

- 扩展一个类型而不破坏现有代码。
- 允许在大部分用户都不需要的特定场合进行自定义。

标准库中的Add trait就是第二种场景的例子：通常你只需要将两个同样类型的值相加，但Add trait也同时提供了自定义额外行为的能力。在Add trait的定义中使用默认类型参数意味着，在大多数情况下你都不需要指定额外的参数。换句话说，就是可以避免一小部分重复的代码模块，从而可以更加轻松地使用trait。

第一种场景与第二种场景有些相似，但却采用了相反的思路：当你想要为现有的trait添加一个类型参数来扩展功能时，你可以给它设定一个默认值来避免破坏已经实现的代码。

用于消除歧义的完全限定语法：调用相同名称的方法

Rust既不会阻止两个trait拥有相同名称的方法，也不会阻止你为同一个类型实现这样的两个trait。你甚至可以在这个类型上直接实现与trait方法同名的方法。

当你调用这些同名方法时，你需要明确地告诉Rust你期望调用的具体对象。思考示例19-16中的代码，它定义了两个拥有同名方法fly的trait：Pilot和Wizard，并为类型Human实现了这两个trait，而Human本身也正好实现了fly方法。每个fly方法都执行了不同的操作。

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19-16：定义了两个拥有同名方法fly的trait，并为本就拥有fly方法的Human类型实现了这两个trait

当我们在Human的实例上调用fly时，编译器会默认调用直接实现类型上的方法，如示例19-17所示。

src/main.rs

```

fn main() {
    let person = Human;
    person.fly();
}

```

示例19-17：在Human实例上调用fly

运行这段代码会打印出*waving arms furiously*，它表明Rust调用了直接实现在Human类型上的fly方法。

为了调用实现在Pilot trait或Wizard trait中的fly方法，我们需要使用更加显式的语法来指定具体的fly方法，如示例19-18所示。

src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

示例19-18：指定我们想要调用哪个trait的fly方法

在方法名的前面指定trait名称向Rust清晰地表明了我们想要调用哪个fly实现。另外，你也可以使用类似的Human::fly(&person)语句，它与示例19-18中使用的person.fly()在行为上等价，但会稍微冗长一些。

运行这段代码会打印出如下所示的内容：

```
This is your captain speaking.
Up!
*waving arms furiously*
```

当你拥有两种实现了同一trait的类型时，对于fly等需要接收self作为参数的方法，Rust可以自动地根据self的类型推导出具体的trait实现。

然而，因为trait中的关联函数没有self参数，所以当在同一作用域下有两个实现了此种trait的类型时，Rust无法推导出你究竟想要调用哪一个具体类型，除非使用完全限定语法（fully qualified syntax）。例如，示例19-19中的Animal trait拥有关联函数baby_name，而示例中定义的Dog结构体在拥有独立关联函数baby_name的同时实现了Animal trait。

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19-19：一个带关联函数的trait和一个带同名关联函数的类型，并且这个类型还实现了该trait

使用这段代码的动物收容所希望将所有的小狗都叫作Spot，他们在Dog的关联函数baby_name中实现了这一需求。另外，Dog类型还同时实现了用于描述动物的通用 trait: Animal。Dog在实现该trait的baby_name函数时将小狗称为puppy。

随后的代码在main函数中使用语句Dog::baby_name来直接调用了Dog的关联函数，它会打印出如下所示的内容：

```
A baby dog is called a Spot
```

这与我们预期的结果有些出入，我们希望的是调用在Dog上实现的Animal trait的baby_name函数来打印出A baby dog is called a puppy。示例19-18中指定trait名称的技术无法解决这一需求，将main函数修改为示例19-20中的代码会导致编译时错误。

src/main.rs

```

fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}

```

示例19-20：尝试调用Animal trait中的baby_name函数，但Rust并不知道应该使用哪一个实现

由于Animal::baby_name是一个没有self参数的关联函数而不是方法，所以Rust无法推断出我们想要调用哪一个Animal::baby_name的实现。尝试编译这段代码会出现如下所示的错误：

```

error[E0283]: type annotations required: cannot resolve `_: Animal`
--> src/main.rs:20:43
  |
20 |     println!("A baby dog is called a {}", Animal::baby_name());
  |                                         ^^^^^^^^^^^^^^^^^^
  |
  = note: required by `Animal::baby_name`
```

为了消除歧义并指示Rust使用Dog为Animal trait实现的baby_name函数，我们需要使用完全限定语法。它在本例中的具体使用方法如示例19-21所示。

src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

示例19-21：使用完全限定语法来调用Dog为Animal trait实现的baby_name函数

这段代码在尖括号中提供的类型标注表明我们希望将Dog类型视作Animal，并调用Dog为Animal trait实现的baby_name函数。修改后的代码能够打印出我们期望的结果了：

```
A baby dog is called a puppy
```

一般来说，完全限定语法被定义为如下所示的形式：

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

对于关联函数而言，上面的形式会缺少receiver而只保留剩下的参数列表。你可以在任何调用函数或方法的地方使用完全限定语法，而Rust允许你忽略那些能够从其他上下文信息中推导出来的部分。只有当代码中存在多个同名实现，且Rust也无法区分出你期望调用哪个具体实现时，你才需要使用这种较为烦琐的显式语法。

用于在trait中附带另外一个trait功能的超trait

有时，你会需要在一个trait中使用另外一个trait的功能。在这种情况下，我们需要使当前trait的功能依赖于另外一个同时被实现的trait。这个被依赖的trait也就是当前trait的超trait(supertype)。

例如，假设我们希望创建一个拥有outline_print方法的OutlinePrint trait，这个方法会在调用时打印出带有星号框的实例

值。换句话说，给定一个实现了Display trait的Point结构体，如果它会将自己的值显示为(x, y)，那么当x和y分别是1和3时，调用outline_print就会打印出如下所示的内容：

```
*****  
*      *  
* (1, 3) *  
*      *  
*****
```

由于我们想要在outline_print的默认实现中使用Display trait的功能，所以OutlinePrint trait必须注明自己只能用于那些提供了Display功能的类型。我们可以在定义trait时指定OutlinePrint: Display来完成该声明，这有些类似于为泛型添加trait约束。示例19-22展示了OutlinePrint trait的实现。

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19-22：实现使用了Display功能的OutlinePrint trait

由于这段定义注明了OutlinePrint依赖于Display trait，所以我们能够在随后的方法中使用to_string函数，任何实现了Display trait的类型都会自动拥有这一函数。如果你尝试去掉trait名后的冒号与Display trait并继续使用to_string，那么Rust就会因为无法在当前作用域中找到&Self的to_string方法而抛出错误。

让我们看一看在没有实现Display的类型上实现OutlinePoint时（如下所示）会发生些什么。

src/main.rs

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl OutlinePrint for Point {}
```

编译后出现的错误提示信息指出了Point类型没有实现必要的Display trait约束：

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied  
--> src/main.rs:20:6  
|  
20 |     impl OutlinePrint for Point {}  
|     ^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;  
try using `{:?}` instead if you are using a format string  
|= help: the trait `std::fmt::Display` is not implemented for `Point`
```

为了解决这一问题，让我们为Point类型实现Display来满足OutlinePrint要求的约束，如下所示：

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)  
    }  
}
```

接着再为Point实现OutlinePrint trait便可以顺利地通过编译了。现在，我们可以调用Point实例的outline_print方法来打印出包含在星号框中的值了。

使用newtype模式在外部类型上实现外部trait

我们曾经在第10章的“为类型实现trait”一节中提到过孤儿规则：只有当类型和对应trait中的任意一个定义在本地包内时，我们才能够为该类型实现这一trait。但实际上，你还可以使用newtype模式来巧妙地绕过这个限制，它会利用元组结构体创建出一个新的类型（我们曾经在第5章的“使用不需要对字段命名的元组结构体来创建不同的类型”一节中讨论过元组结构体）。这个元组结构体只有一个字段，是我们想要实现trait的类型的瘦封装（thin wrapper）。由于封

装后的类型位于本地包内，所以我们可以为这个壳类型实现对应的 trait。newtype是一个来自Haskell编程语言的术语。值得注意的是，使用这一模式不会导致任何额外的运行时开销，封装后的类型会在编译过程中被优化掉。

例如，孤儿规则会阻止我们直接为Vec<T>实现Display，因为Display trait与Vec<T>类型都被定义在外部包中。为了解决这一问题，我们可以首先创建一个持有Vec<T>实例的Wrapper结构体。接着，我们便可以为Wrapper实现Display并使用Vec<T>值了，如示例19-23所示。

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19-23：创建一个包含Vec<String>的Wrapper类型，并为其实现Display

这段代码在实现Display的过程中使用了self.0来访问内部的Vec<T>，因为Wrapper是一个元组结构体，而Vec<T>是元组中序号为0的那个元素。接着，我们就可以使用Wrapper中的Display功能了。

这项技术仍然有它的不足之处。因为Wrapper是一个新的类型，所以它没有自己内部值的方法。为了让Wrapper的行为与Vec<T>完全一致，我们需要在Wrapper中实现所有Vec<T>的方法，并将这些方法委托给self.0。假如我们希望新类型具有内部类型的所有方法，那么我们也可以为Wrapper实现Deref trait（在第15章的“通过Deref trait将智能指针视作常规引用”一节曾经讨论过这一技术）来直接返回内部的类型。假如我们不希望Wrapper类型具有内部类型的所有方法，比如

在需要限制Wrapper类型的行为时，我们就只能手动实现需要的那部分方法了。

现在，你知道newtype模式是如何与trait配合使用的了；但即便不涉及trait概念，它也是一个非常有用的模式。接下来，让我们把焦点转移到一些更为高级的类型系统交互方式上来。

高级类型

本书在之前的章节里曾经粗略地提及过一些比较高级的类型系统特性，但却碍于种种原因没有立即深入地进行研究。在本节中，我们会首先讨论更为通用的newtype模式，该模式作为类型在某些场景下十分有用。接着，我们会把目光转移至类型别名，它与newtype类似但拥有不同的语义。最后，我们还会讨论! 类型及动态大小类型。

注意

接下来的内容会假定你已经阅读过了“使用newtype模式在外部类型上实现外部trait”一节。

使用newtype模式实现类型安全与抽象

newtype模式在一些我们还没有介绍过的任务中同样有用，它可以被用来静态地保证各种值之间不会被混淆及表明值使用的单位。你曾经在示例19-15中见到过使用newtype来标注单位的例子，回忆一下当时的场景，我们分别使用Millimeters结构体和Meters结构体封装了u32的值，这就是典型的newtype模式。假如我们编写了一个接收Millimeters值作为参数的函数，那么Rust就会在我们意外传入Meters值或u32值的时候出现编译错误。

newtype模式的另外一个用途是为类型的某些细节提供抽象能力。例如，新类型可以暴露出一个与内部私有类型不同的公共API，从而限制用户可以访问的功能。

newtype模式还可以被用来隐藏内部实现。例如，我们可以提供People类型来封装一个用于存储人物ID及其名称的HashMap<i32,

`String`。`People`类型的用户只能使用我们提供的公共API，比如一个添加名称字符串到`People`集合的方法；而调用该方法的代码不需要知道我们在内部赋予了名称一个对应的`i32` ID。`newtype`模式通过轻量级的封装隐藏了实现细节，正如我们在第17章的“封装实现细节”一节中讨论过的那样。

使用类型别名创建同义类型

除了`newtype`模式，Rust还提供了创建类型别名（`type alias`）的功能，它可以为现有的类型生成另外的名称。这一特性需要用到`type`关键字。例如，我们可以像下面一样创建`i32`的别名`Kilometers`：

```
type Kilometers = i32;
```

现在，别名`Kilometers`被视作了`i32`的同义词；不同于我们在示例19-15中创建的`Millimeters`与`Meters`类型，`Kilometers`并不是一个独立的新类型。`Kilometers`类型的值实际上等价于`i32`类型的值：

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

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

也正是由于`Kilometers`和`i32`是同一种类型，所以我们可以把两个类型的值相加起来，甚至是将`Kilometers`类型的值传递给以`i32`类型作为参数的函数。但无论如何，当你使用这种方式时，你就无法享有`newtype`模式附带的类型检查的便利。

类型别名最主要的用途是减少代码字符重复。例如，我们可能会拥有一个如下所示的较长的类型：

```
Box<dyn Fn() + Send + 'static>
```

在函数签名中插入或在代码中通篇标注这样的类型不但令人生厌，也非常容易出错。与示例19-24类似的代码充斥着整个项目会是一幅怎样的场景？

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
```

```

// --略

-- 
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --略

-- 
#     Box::new(|| ())
}

```

示例19-24：在多个地方使用长类型

类型别名通过减少字符重复可以使得代码更加易于管理。在示例19-25中，我们引入了一个别名Thunk来替换所有冗长的类型标注。

```

type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --略

-- 
}

fn returns_long_type() -> Thunk {
    // --略

-- 
#     Box::new(|| ())
}

```

示例19-25：引入类型别名Thunk以减少重复

新的代码明显更加易读！为类型别名选择一个有意义的名字可以帮助你清晰地表达自己的意图。此处的Thunk指代一段可以延后执行的代码，它对于存储的闭包来说是一个较为合适的名字。

Result<T, E>类型常常使用类型别名来减少代码重复。考虑一下标准库中的std::io模块，该模块下的I/O操作常常会返回Result<T, E>来处理操作失败时的情形。另外，该代码库使用了一个std::io::Error结构体来表示所有可能的I/O错误，而大部分std::io模块下的函数都会将返回类型Result<T, E>中的E替换为std::io::Error，比如Write trait中的这些函数：

```

use std::io::Error;
use std::fmt;

```

```

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>;
}

```

这里重复出现了许多Result<... , Error>。为此，std::io有了如下所示的类型别名：

```
type Result<T> = Result<T, std::io::Error>;
```

由于该声明被放置在std::io模块中，所以我们可以使用完全限定别名std::io::Result<T>来指向它，即指向将std::io::Error填入E的Result<T, E>。简化后的Write trait函数如下所示：

```

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: Arguments) -> Result<()>;
}

```

使用类型别名可以从两个方面帮助我们：它让编写代码更加轻松，并且为整个std::io提供了一致的接口。另外，由于它仅仅是别名，也就是另外一个Result<T, E>，所以我们在它的实例上调用Result<T, E>拥有的任何方法，甚至是? 运算符。

永不返回的Never类型

Rust有一个名为! 的特殊类型，它在类型系统中的术语为空类型(empty type)，因为它没有任何的值。我们倾向于叫它never类型，因为它在从不返回的函数中充当返回值的类型。例如：

```

fn bar() -> ! {
    // --略
    --
}

```

这段代码可以读作“函数bar永远不会返回值”。不会返回值的函数也被称作发散函数(diverging function)。我们不可能创建出类型为! 的值来让bar返回。

但是一个不能创建值的类型究竟有什么用处呢？回忆一下示例2-5中的代码，我们将其中的部分代码重现在示例19-26中。

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

示例19-26：拥有一个以continue结尾的分支的match语句

当时，我们略过了这段代码中的某些细节。随后，我们在第6章的“控制流运算符match”一节中指出所有的match分支都必须返回相同的类型。因此，类似于如下所示的代码是无法工作的：

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

上面代码中guess的类型既可以是整数，也可以是字符串，而Rust则明确要求guess只能是单一的类型。那么示例19-26中的continue究竟返回了什么呢？我们为何可以在一个分支中返回u32，而在另一个分支中以continue结束呢？

正如你可能会猜到的，continue的返回类型是!。当Rust计算guess的类型时，它会发现在可用于匹配的两个分支中，前者的返回类型为u32而后的返回类型为!。因为!无法产生一个可供返回的值，所以Rust采用了u32作为guess的类型。

对于此类行为，还有另外一种更加正式的说法：类型!的表达式可以被强制转换为其他的任意类型。我们之所以能够使用continue来结束match分支，是因为continue永远不会返回值；相反地，它会将程序的控制流转移至上层循环。因此，这段代码在输入值为Err的情况下不会对guess进行赋值。

panic! 宏的实现同样使用了never类型。还记得我们在Option<T>值上调用unwrap函数吗？它会生成一个值或触发panic。下面便是这个函数的定义：

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

```
    }
}
}
```

这段代码中发生的行为类似于示例19-26中match的行为：Rust注意到val拥有类型T，而panic!则拥有返回类型!，所以整个match表达式的返回类型为T。这段代码之所以可以正常工作，是因为panic!只会中断当前的程序而不会产生值。因为我们不会在None的情况下为unwrap返回一个值，所以这段代码是合法的。

最后一个以!作为返回类型的表达式是loop：

```
print!("forever ");
loop {
    print!("and ever ");
}
```

由于loop循环永远不会结束，所以这个表达式以!作为自己的返回类型。当然，循环中也可能会存在break指令，并会在逻辑执行至break时中止。

动态大小类型和Sized trait

通常而言，Rust需要在编译时获取一些特定的信息来完成自己的工作，比如应该为一个特定类型的值分配多少空间等。但Rust的类型系统中又同时存在这样一处令人疑惑的角落：动态大小类型（Dynamically Sized Type, DST）的概念，它有时也被称作不确定大小类型（unsized type），这些类型使我们可以在编写代码时使用只有在运行时才能确定大小的值。

让我们来深入研究一个叫作str的动态大小类型，这个类型几乎贯穿了本书的所有章节。没错，我们会在这里讨论str本身而不是&str，str正好是一个动态大小类型。我们只有在运行时才能确定字符串的长度，这也意味着我们无法创建一个str类型的变量，或者使用str类型来作为函数的参数。如下所示的代码无法正常工作：

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust需要在编译时确定某个特定类型的值究竟会占据多少内存，而同一类型的所有值都必须使用等量的内存。假如Rust允许我们写出上面这样的代码，那么这两个str的值就必须要占据等量的空间。但它们确实具有不同的长度：s1需要12字节的存储空间，而s2则需要15字节。这也是我们无法创建出动态大小类型变量的原因。

那么我们应该怎么处理类似的需求呢？你应该已经非常熟悉本例中出现的情形了：我们会把s1与s2的类型从str修改为&str。回忆一下第4章的“字符串切片”一节，我们当时指出，切片的数据结构中会存储数据的起始位置及切片的长度。

因此，尽管&T被视作存储了T所在内存地址的单个值，但&str实际上是由两个值组成的：str的地址与它的长度。这也使我们可以在编译时确定&str值的大小：其长度为usize长度的两倍。换句话说，无论&str指向了什么样的字符串，我们总是能够知道&str的大小。这就是Rust中使用动态大小类型的通用方式：它们会附带一些额外的元数据来存储动态信息的大小。我们在使用动态大小类型时总是会把它的值放在某种指针的后面。

我们可以将str与所有种类的指针组合起来，例如Box<str>或Rc<str>等。事实上，你在之前的章节就已经见到过类似的用法了，只不过当时使用了另外一种动态大小类型：trait。每一个trait都是一个可以通过其名称来进行引用的动态大小类型。在第17章的“使用trait对象来存储不同类型的值”一节中曾经提到过，为了将trait用作trait对象，我们必须将它放置在某种指针之后，比如dyn Trait或Box<dyn Trait>（Rc<dyn Trait>也可以）之后。

为了处理动态大小类型，Rust还提供了一个特殊的Sized trait来确定一个类型的大小在编译时是否可知。编译时可计算出大小的类型会自动实现这一trait。另外，Rust还会为每一个泛型函数隐式地添加Sized约束。也就是说，下面定义的泛型函数：

```
fn generic<T>(t: T) {  
    // --略  
}  
--
```

实际上会被隐式地转换为：

```
fn generic<T: Sized>(t: T) {  
    // --略  
  
    --  
}
```

在默认情况下，泛型函数只能被用于在编译时已经知道大小的类型。但是，你可以通过如下所示的特殊语法来解除这一限制：

```
fn generic<T: ?Sized>(t: &T) {  
    // --略  
  
    --  
}
```

?Sized trait约束表达了与Sized相反的含义，我们可以将它读作“T可能是也可能不是Sized的”。这个语法只能被用在Sized上，而不能被用于其他trait。

另外还需要注意的是，我们将t参数的类型由T修改为了&T。因为类型可能不是Sized的，所以我们需要将它放置在某种指针的后面。在本例中，我们选择使用引用。

接下来，让我们继续讨论函数与闭包！

高级函数与闭包

我们终于可以来讨论一些有关函数与闭包的高级特性了，它们包括函数指针及闭包返回。

函数指针

我们曾经讨论过如何将闭包传递给函数，但实际上你同样可以将普通函数传递至其他函数！这一技术可以帮助你将已经定义好的函数作为参数，而无须重新定义新的闭包。函数会在传递的过程中被强制转换成fn类型，注意这里使用了小写字符f从而避免与Fn闭包trait相混淆。fn类型也就是所谓的函数指针（function pointer），将参数声明为函数指针时使用的语法与闭包类似，如示例19-27所示。

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19-27：使用fn类型来接收函数指针作为参数

这段代码会打印出The answer is: 12，其中函数do_twice的参数f被指定为了fn类型，它会接收i32类型作为参数，并返回一个i32作为

结果。随后do_twice函数体中的代码调用了两次f。在main函数中，我们将函数add_one作为第一个参数传递给了do_twice。

与闭包不同，fn是一个类型而不是一个trait。因此，我们可以直接指定fn为参数类型，而不用声明一个以Fn trait为约束的泛型参数。

由于函数指针实现了全部3种闭包 trait（Fn、FnMut 以及 FnOnce），所以我们总是可以把函数指针用作参数传递给一个接收闭包的函数。也正是出于这一原因，我们倾向于使用搭配闭包trait的泛型来编写函数，这样的函数可以同时处理闭包与普通函数。

当然，在某些情形下，我们可能只想接收fn而不想接收闭包，比如与某种不支持闭包的外部代码进行交互时：C函数可以接收函数作为参数，但它却没有闭包。

下面让我们来看一个既可以使用闭包也可以使用命名函数的例子，即map方法的相关应用。为了使用map函数来将一个整型动态数组转换为一个字符串动态数组，我们可以像下面一样使用闭包：

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

我们也可以使用一个函数作为map的参数，如下所示：

```
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”一节中提到的完全限定语法，因为此作用域中存在多个可用的to_string函数。本例使用的是ToString trait中的to_string函数，而标准库已经为所有实现了Display的类型都自动实现了这一trait。

另外还有一种十分有用的模式，它利用了元组结构体和元组结构枚举变体的实现细节。这些类型的初始化语法()与调用函数有些相似。实际上，它们的构造器也确实被实现为了函数，该函数会接收它

们的参数并返回一个新的实例。因此，我们可以把构造器视作实现了闭包trait的函数指针，并在那些接收闭包的方法中使用它们：

```
enum Status {
    Value(u32),
    Stop,
}

let list_of_statuses: Vec<Status> =
    (0u32..20)
        .map(Status::Value)
        .collect();
```

这段代码使用Status::Value的构造器调用了map方法，从而为范围中的每一个u32值创建了对应的Status::Value实例。在实际编程中，有一些人倾向于使用这种风格，而另外一些人则喜欢使用闭包。这两种形式最终都会编译出同样的代码，你完全可以按照自己的喜好决定使用哪种风格。

返回闭包

由于闭包使用了trait来进行表达，所以你无法在函数中直接返回一个闭包。在大多数希望返回trait的情形下，你可以将一个实现了该trait的具体类型作为函数的返回值。但你无法对闭包执行同样的操作，因为闭包没有一个可供返回的具体类型；例如，你无法把函数指针fn用作返回类型。

下面的代码试图直接返回一个闭包，但它却无法通过编译：

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

编译后出现的错误如下所示：

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static:
std::marker::Sized` is not satisfied
-->
|
1 | fn returns_closure() -> Fn(i32) -> i32 {
|           ^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static`
does not have a constant size known at compile-time
|
= help: the trait `std::marker::Sized` is not implemented for
`std::ops::Fn(i32) -> i32 + 'static`
= note: the return type of a function must have a statically known size
```

这段错误提示信息再次指向了Sized trait！Rust无法推断出自己需要多大的空间来存储此处返回的闭包。幸运的是，我们已经在之前的章节中接触过了解决这一问题的方法，那就是使用trait对象：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

现在的代码可以正常编译了。如果你想要了解更多有关trait对象的信息，请参考第17章的“使用trait对象来存储不同类型的值”一节。

让我们接着来看一看有关宏的高级特性！

宏

虽然我们在本书中大量地使用了与`println!`类似的宏，但我们始终没有正式地研究过它究竟是什么，以及它是怎样工作的。术语宏（macro）其实是Rust中的某一组相关功能的集合称谓，其中包括使用`macro_rules!`构造的声明宏（declarative macro）及另外3种过程宏（procedural macro）：

- 用于结构体或枚举的自定义#[derive]宏，它可以指定随derive属性自动添加的代码。
- 用于为任意条目添加自定义属性的属性宏。
- 看起来类似于函数的函数宏，它可以接收并处理一段标记（token）序列。

我们会依次讨论这些功能，但首先需要弄清楚的是，既然已经有了函数概念，为什么还需要宏呢？

宏与函数之间的差别

从根本上来说，宏是一种用于编写其他代码的代码编写方式，也就是所谓的元编程范式（metaprogramming）。附录C中会讨论的`derive`属性是一种宏，它会自动为你生成各种trait的实现。我们在本书中一直使用`println!`宏与`vec!`宏。这些宏会通过展开来生成比你手写代码更多的内容。

元编程可以极大程度地减少你需要编写和维护的代码数量，虽然这也是函数的作用之一，但宏却有一些函数所不具备的能力。

函数在定义签名时必须声明自己参数的个数与类型，而宏则能够处理可变数量的参数：我们可以使用单一参数调用 `println!("hello")`，也可以使用两个参数调用 `println!("hello {}", name)`。另外，由于编译器会在解释代码前展开宏，所以宏可以被用来执行某些较为特殊的任务，比如为类型实现 trait 等。之所以函数无法做到这一点，是因为 trait 需要在编译时实现，而函数则是在运行时调用执行的。

编写一个宏来实现功能相较于函数也有它自己的缺点：宏的定义要比函数定义复杂得多，因为你需要编写的是用于生成 Rust 代码的 Rust 代码。正是由于这种间接性，宏定义通常要比函数定义更加难以阅读、理解及维护。

宏和函数间的最后一个区别在于：当你在某个文件中调用宏时，你必须提前 定义宏或将宏引入当前作用域中，而函数则可以在任意位置定义并在任意位置使用。

用于通用元编程的macro_rules! 声明宏

Rust 中最常用的宏形式是声明宏，它们有时也被称作“模板宏”（macros by example）“macro_rules! 宏”，或者直白的“宏”。从核心形式上来讲，声明宏要求你编写出类似于 `match` 表达式的东西。正如在第 6 章讨论过的那样，`match` 表达式是一种接收其他表达式的控制结构，它会将表达式的结果值与模式进行比较，并在匹配成功时执行对应分支中的代码。类似地，宏也会将输入的值与带有相关执行代码的模式进行比较：此处的值是传递给宏的字面 Rust 源代码，而此处的模式则是可以用来匹配这些源代码的结构。当某个模式匹配成功时，该分支下的代码就会被用来替换传入宏的代码。所有的这一切都会发生在编译时期。

为了定义一个宏，你需要用到 `macro_rules!`。接下来，让我们学习 `vec!` 宏的定义方式来了解如何使用 `macro_rules!`。在第 8 章提到过，`vec!` 宏可以被用来创建一个具有特定元素的动态数组。例如，下面的宏创建出了一个包含 3 个整数的动态数组：

```
let v: Vec<u32> = vec![1, 2, 3];
```

当然，我们也可以使用`vec!` 宏来创建出包含2个整数的动态数组或包含5个字符串切片的动态数组。而函数则无法完成同样的事情，因为我们无法提前确定值的类型与数量。

示例19-28展示了一个稍微简化后的`vec!` 宏定义。

src/lib.rs

```
❶#[macro_export]
❷macro_rules! vec {
    ❸ ($($x:expr),*) => {
        {
            let mut temp_vec = Vec::new();
            ❹ $(
                ❺ temp_vec.push($x❻);
            )*
            ❻ temp_vec
        }
    };
}
```

示例19-28：`vec!` 宏定义的简化版本

注意

标准库中实际定义的`vec!` 宏包含了预先分配内存的代码。为了让例子更为简单，我们移除了这部分用于优化的代码。

代码中标注的`#[macro_export]` ❶意味着这个宏会在它所处的包被引入作用域后可用。缺少了这个标注的宏则不能被引入作用域。

接着，我们使用了`macro_rules!` 及不带感叹号的名称来开始定义宏❷。宏的名称（也就是本例中的`vec`）后面是一对包含了宏定义体的花括号。

`vec!` 代码块中的结构与`match`表达式的结构相似。这段实现中存在一个模式为`($($x:expr),*)`的分支，模式后紧跟着的是`=>`及对应的代码块❸，这些关联代码会在模式匹配成功时触发。由于这是这个宏中仅有的模式，所以整个宏只存在一种有效的匹配方法；任何其他模式都会导致编译时错误。某些更加复杂的宏会包含多个分支。

宏定义中的有效模式语法与在第18章讲到的模式语法不同，因为宏模式匹配的是Rust代码结构，而不是值。让我们一步一步来看一看示例19-28中的模式片段的意思是什么。如果想要了解完整的宏模式语法，请参考Rust官方网站的相关文档。

我们首先使用了一对圆括号来把整个模式包裹起来。接着是一个\$符号，以及另外一对包裹着匹配模式的圆括号，这些被匹配并捕获的值最终会被用于生成替换代码。\$()中的\$x:expr可以匹配任意的Rust表达式，并将其命名为\$x。

\$()之后的逗号意味着一个可能的字面逗号分隔符会出现在捕获代码的后面，而逗号后的*则意味着这个模式能够匹配零个或多个*之前的东西。

当我们使用指令vec![1, 2, 3];调用这个宏时，\$x模式会分别匹配3个表达式：1、2及3。

现在，让我们把目光转移到该分支对应的代码中：它会为模式中匹配到的每一个\$()生成\$()*^{④⑦}中对应的temp_vec.push()^⑤代码；这一展开过程会重复零次还是多次，取决于匹配成功的表达式数量。而\$x^⑥则会被每个匹配到的表达式替代。使用vec![1, 2, 3];调用宏会生成如下所示的代码来替换调用语句：

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

我们定义的这个宏可以接收任意数量、任意类型的参数，并创建出一个包含指定元素的动态数组。

不得不承认，macro_rules! 中存在一些奇怪的技术细节。Rust开发团队正在致力于推出使用macro关键字的第二种声明宏，它与现有宏的工作方式类似但修复了某些可能的极端情况。更新后，macro_rules! 会被标记为弃用。由于大多数Rust程序员都只是单纯地使用宏，而不会编写宏，所以我们就不再深入讨论macro_rules! 了。如果你想要学习更多有关编写宏的知识，请参考在线文档或其他资源，比如*The Little Book of Rust Macros* 等。

基于属性创建代码的过程宏

第二种形式的宏更像函数（某种形式的过程）一些，所以它们被称为过程宏。过程宏会接收并操作输入的Rust代码，并生成另外一些Rust代码作为结果，这与声明宏根据模式匹配来替换代码的行为有所不同。

虽然过程宏存在3种不同的类型（自定义派生宏、属性宏及函数宏），但它们都具有非常类似的工作机制。

当创建过程宏时，宏的定义必须单独放在它们自己的包中，并使用特殊的包类型。这完全是因为技术上的原因，我们希望未来能够消除这种限制。使用过程宏的代码如示例19-29所示，其中的`some_attribute`是一个用来指定过程宏类型的占位符。

src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream { }
```

示例19-29：使用过程宏的一个例子

这个定义了过程宏的函数接收一个`TokenStream`作为输入，并产生一个`TokenStream`作为输出。`TokenStream`类型在`proc_macro`包（Rust自带）中定义，表示一段标记序列。这也是过程宏的核心所在：需要被宏处理的源代码组成了输入的`TokenStream`，而宏生成的代码则组成了输出的`TokenStream`。函数附带的属性决定了我们究竟创建的是哪一种过程宏。同一个包中可以拥有多种不同类型的过程宏。

考虑到不同类型的过程宏之间是如此相似，我们会从自定义派生宏开始讨论，并在随后介绍它和其余过程宏之间的细微差别。

如何编写一个自定义`derive`宏

让我们创建一个名为hello_macro的包，并在其中定义一个拥有关联函数hello_macro的HelloMacro trait。为了避免用户在他们的每一个类型上逐一实现HelloMacro trait，我们会提供一个能够自动实现trait的过程宏。这使用户可以在他们的类型上标注#[derive(HelloMacro)]，进而得到hello_macro函数的默认实现，即将文本Hello Macro! My name is TypeName! 中的TypeName替换为当前类型的名称后打印出来。换句话说，我们提供的包可以使其他程序员编写出如示例19-30所示的代码。

src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

示例19-30：包的用户可以使用我们提供的过程宏来编写出这样的代码

这段代码会在运行完毕后打印出Hello, Macro! My name is Pancakes! 这样的信息。首先，我们需要创建一个新的代码包：

```
$ cargo new hello_macro -lib
```

接下来，我们会定义HelloMacro trait及其关联函数：

src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

在拥有了trait与相应的函数后，我们的用户可以直接实现该trait来达成期望的功能，如下所示：

```
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();
}
```

但是，他们必须为每一个希望使用hello_macro功能的类型编写出类似的实现代码，而我们则想要将用户从这些烦琐的工作中解放出来。

另外，我们无法提供hello_macro函数，该函数在默认情况下可以实现打印出正在实现一个trait的类型的名称的功能：因为Rust没有提供反射功能，所以它无法在运行时查找到类型的名称。因此，我们需要的是一个能够在编译时生成代码的宏。

下一步便是定义过程宏了。在我们编写本书的过程中，过程宏依然需要被单独放置到它们自己的包内，Rust开发团队也许会在未来去掉这一限制。就目前而言，组织主包和宏包的惯例是，对于一个名为foo的包，我们会生成一个用于放置自定义派生过程宏的包foo_derive。现在，让我们在hello_macro的项目中创建一个名为hello_macro_derive的包：

```
$ cargo new hello_macro_derive --lib
```

由于这两个包紧密相关，所以我们将它们放置到了同一目录中。如果我们改变了hello_macro中的trait定义，那么我们也需要同时修改hello_macro_derive中有关过程宏的实现。这两个包需要被独立地公开发行，使用它们的程序员应当分别添加这两个依赖并将它们导入作用域中。我们也可以让hello_macro包依赖于hello_macro_derive并重新导出过程宏的代码。但不管怎么样，目前使用的项目结构都可以使用户在不引入derive功能的前提下继续使用hello_macro。

我们需要声明hello_macro_derive包是一个含有过程宏的包。正如你稍后会看到的，我们还需要使用syn和quote包中的功能，所以我们应该将它们声明为依赖。将如下所示的内容添加至hello_macro_derive包的*Cargo.toml*文件中：

hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "0.14.4"
quote = "0.6.3"
```

为了开始定义过程宏，将示例19-31中的代码放入hello_macro_derive包的*src/lib.rs*文件中。注意，在我们为impl_hello_macro函数添加定义前，这段代码还无法通过编译。

hello_macro_derive/src/lib.rs

```
extern crate proc_macro;

use crate::proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // 将Rust代码转换为我们能够进行处理的语法树

    let ast = syn::parse(input).unwrap();
    // 构造对应的trait实现

    impl_hello_macro(&ast)
}
```

示例19-31：大部分的过程宏包都需要这些逻辑来处理Rust代码

注意，我们将负责解析TokenStream的代码提取到了单独的函数hello_macro_derive中，而impl_hello_macro函数则只负责转换语法树：这一实践方式会使得编写过程宏更加方便。这段代码中的外部函数（也就是本例中的hello_macro_derive）会出现在你能看到的大部分拥有过程宏的包中。你仅仅需要根据特定目标来定制内部函数（也就是本例中的impl_hello_macro）的具体实现。

这段代码还引入了3个新的外部包：proc_macro、syn及quote。我们可以借助proc_macro包提供的编译器接口在代码中读取和操作Rust代码，由于它已经被内置在Rust中了，所以我们不需要将它添加至Cargo.toml的依赖中。

syn包被用来将Rust代码从字符串转换为可供我们进一步操作的数据结构体。最后的quote包则能够将syn包产生的数据结构重新转换为Rust代码。这些工具包使得解析Rust代码的任务变得相当轻松：要知道编写一个完整的Rust代码解析器可不是一件简单的事情。

当包的用户在某个类型上标注#[derive(HelloMacro)]时，hello_macro_derive函数就会被自动调用。之所以会发生这样的操作，是因为我们在 hello_macro_derive 函数 上 标 注 了 proc_macro_derive，并在该属性中指定了可以匹配到 trait 的名称 HelloMacro；这是大多数过程宏都需要遵循的编写惯例。

hello_macro_derive函数会首先把input参数从TokenStream转换为一个可供我们解释和操作的数据结构，这也正是syn发挥作用的地方。syn的parse函数接收一个TokenStream作为输入，并返回一个DeviceInput结构体作为结果，这个结构体代表了解析后的Rust代码。示例19-32展示了字符串struct Pancakes; 被解析为DeviceInput结构体后的产出结果。

```
DeriveInput {
    // --略

    --
    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

示例19-32：解析示例19-30中带有宏属性的代码后得到的DeviceInput实例

这个结构体中的字段表明刚刚解析的Rust代码是一个单位结构体，它的ident (identifier, 也就是标识符的意思) 是Pancakes。这个结构体中可用的字段远多于此处示例中的，它能够被用来描述所有

种类的Rust代码，你可以查看syn中有关DeviceInput的文档来获取更多信息。

我们会紧接着开始定义impl_hello_macro函数，这也正是我们用来生成新Rust代码的地方。但在这之前，你需要注意到这个宏函数的产出物也是一个TokenStream。返回的TokenStream会被添加到使用这个宏的用户代码中，并使用户在编译自己的包时获得我们提供的额外功能。

注意，我们在使用syn::parse函数后调用了unwrap，函数hello_macro_derive会在出现解析错误时直接触发panic。在失败时立即中止程序对于编写过程宏来说是必要的，因为proc_macro_derive函数必须遵循过程宏的API规范来返回一个TokenStream，而不是Result。我们在这里选择了使用unwrap来简化示例；但在产品级的代码中，你应该使用panic! 或expect来添加更多用于指明错误原因的信息。

我们现在已经把被标注的Rust代码从TokenStream转换为了DeviceInput实例。接下来，我们添加的代码将为被标注类型实现HelloMacro trait，如示例19-33所示。

hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput)
-> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        };
        gen.into()
    }
}
```

示例19-33：使用解析后的Rust代码实现HelloMacro trait

这段代码首先取得了一个Ident结构体实例，它包含了被标注类型的名称ast.ident。根据示例19-32中所展示的，impl_hello_macro函数作用于示例19-30时产生的Ident实例会包含一个值为“Pancakes”的ident字段。因此，示例19-33中name变量包含的Ident结构体实例会在打印时输出字符串“Pancakes”，也就是示例19-30中结构体的名字。

其中的`quote!` 宏允许我们定义那些希望返回的Rust代码。由于`quote!` 宏的执行结果是一种编译器无法直接理解的类型，所以我们还需要将执行结果转换为`TokenStream`类型。我们可以通过调用`into`方法来实现这样的转换，该方法可以将这段中间代码的返回值类型转换为符合要求的`TokenStream`类型。

另外，`quote!` 宏还提供了一些非常酷的模板机制：它会将我们输入的`#name`替换为变量`name`中的值。你甚至可以在这个宏的代码块中编写一些类似于常规宏的重复操作。请查阅`quote`包的官方文档来获得关于它的更全面的介绍。

我们希望编写的过程宏能够为用户标注的类型生成一份`HelloMacro trait`的实现，而这个类型的名称可以通过使用`#name`得到。该trait的实现只有一个`hello_macro`函数，它的函数体内会包含我们想要提供的功能：打印出`Hello, Macro! My name is`及被标注类型的名称。

这里使用的`stringify!` 宏是内置在Rust中的，它接收一个Rust表达式，比如`1 + 2`，并在编译时将这个表达式转换成字符串字面量，比如“`1 + 2`”。这种行为与`format!` 或`println!` 的行为可不相同，后者会计算出表达式的值并将其返回为`String`。代码中输入的`#name`有可能是一个表达式，而因为我们希望直接打印出这个值的字面量，所以这里使用了`stringify!`。使用`stringify!` 还可以省去内存分配的开销，因为它在编译时就已经将`#name`转换为了字符串字面量。

`cargo build`此时应该能够在`hello_macro`和`hello_macro_derive`上顺利地通过编译了。让我们把这两个包连接到示例19-30的代码中来看一看过程宏会产生什么样的效果！使用`cargo new pancakes`在你的项目 目录中创建一个新的可执行程序，接着将`hello_macro` 和`hello_macro_derive`添加到`pancakes`包的`Cargo.toml` 中作为依赖。假如你将`hello_macro`与`hello_macro_derive`发布到了`crates.io`上，那么你可以按照常用的方式来依赖它们；而假如没有的话，则应该使用`path`依赖按照如下所示的方式来指定它们：

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

把示例19-30中的代码复制到`src/main.rs`中并执行`cargo run`，它应该会打印出Hello, Macro! My name is Pancakes!。我们在过程宏里提供的`HelloMacro` trait实现已经被成功地包含在了代码中，而无须`pancakes`包单独实现它；`#[derive(HelloMacro)]`自动地添加了这个trait的实现。

接下来，让我们看一看其他过程宏与自定义派生宏之间的区别。

属性宏

属性宏与自定义派生宏类似，它们允许你创建新的属性，而不是为`derive`属性生成代码。属性宏在某种程度上也更加灵活：`derive`只能被用于结构体和枚举，而属性则可以同时被用于其他条目，比如函数等。下面便是一个使用了属性宏的例子，即假设你拥有一个名为`route`的属性，那么就可以在编写Web应用框架时为函数添加标记：

```
#[route(GET, "/")]
fn index() {
```

这个`#[route]`属性是由框架本身作为一个过程宏来定义的，这个宏定义的函数签名如下所示：

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream
{
```

上面的代码中出现了两个类型为`TokenStream`的参数。前者是属性本身的内容，也就是本例中的`Get, "/"`部分，而后者则是这个属性所附着的条目，也就是本例中的`fn index() {}`及剩下的函数体。

除此之外，属性宏与自定义派生宏的工作方式几乎一样：它们都需要创建一个`proc-macro`类型的包并提供生成相应代码的函数。

函数宏

函数宏可以定义出类似于函数调用的宏，但它们远比普通函数更为灵活。例如，与`macro_rules!`宏类似，函数宏也能接收未知数量的参数。但是，`macro_rules!`宏只能使用类似于`match`的语法来进行定

义，而函数宏则可以接收一个TokenStream作为参数，并与另外两种过程宏一样在定义中使用Rust代码来操作TokenStream。例如，我们可能会这样调用一个名为sql! 的函数宏：

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

这个宏会解析圆括号内的SQL语句并检验它在语法上的正确性，这一处理过程所做的比macro_rules! 宏可以完成的任务要复杂得多。此处的sql! 可以被定义为如下所示的样子：

```
# [proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

这里的定义与自定义派生宏的签名十分类似：我们接收括号内的标记序列作为参数，并返回一段执行相应功能的生成代码。

总结

哇！我们在本章学习了不少生僻的Rust特性，你也许不会经常用到它们，但你应该能够意识到这些功能在某些特定场景下的作用。当你在错误提示信息或别人的代码中碰见这些稍显复杂的主题时，至少能够识别出这些概念与语法。你可以把本章当作参考材料，并在遇到问题时返回来寻找解决方案。

接下来，我们会把本书讨论过的所有内容用于实践，并完成一个全新的项目！

第20章

最后的项目：构建多线程Web服务器

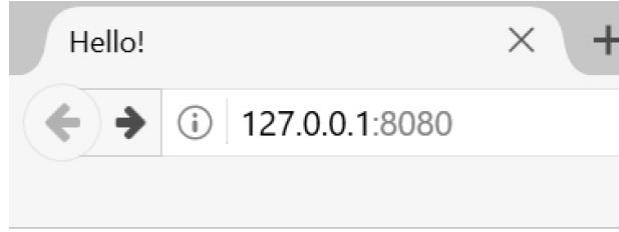


这可真是一段漫长的旅程，但我们可以快要接近尾声了。我们会在本章开发一个新的实践项目来展示最后几章中涉及的概念，并顺带复习一些之前章节提到的知识点。

我们将在本章的终极项目中实现一个能够返回“Hello！”的Web服务器，它在浏览器中的显示如图20-1所示。

为了构建Web服务器，我们会依次完成如下所示的计划：

1. 学习一些有关TCP和HTTP的知识。
2. 在套接字（socket）上监听TCP连接。
3. 解析少量的HTTP请求。
4. 创建一个合适的HTTP响应。
5. 使用线程池改进服务器的吞吐量。



Hello!

Hi from Rust

图20-1 我们共同编写的最后一个项目

值得注意的是，我们在本章采用的技术并不是构建Web服务器的最佳实践，你可以在crates.io中找到一些更为优秀的Web服务器或线程池实现，它们中的一部分甚至可以被直接应用在生产环境下。

然而，我们的目标终究是巩固学习成果而不是寻找捷径。由于Rust是一个系统级编程语言，所以我们能够按需选择代码的抽象层次，这些可用的抽象手段要比其他某些语言能够提供的机制触及的层次更低。因此，我们选择手动编写一个基本的HTTP服务器与线程池，以便你学习到代码背后的通用技术与思路并将它们应用到未来的实际代码中。

构建单线程Web服务器

首先，我们需要让一个单线程的Web服务器工作起来。在此之前，我们会快速地了解一下构建Web服务器需要使用的相关协议。有关这些协议的详细讨论超出了本书的范畴，但是简要的介绍应该就可以提供足够的背景信息了。

Web服务器涉及的两个主要协议分别是超文本传输协议（HTTP）和传输控制协议（TCP）。它们两者都是基于请求-响应（request-response）的协议，也就是说，这个协议由客户端发起请求，再由服务器监听并响应客户端。请求和响应的内容会由协议本身定义。

TCP是一种底层协议，它描述了信息如何从一个服务器传送到另外一个服务器的细节，但它并不指定信息的具体内容。HTTP协议建立在TCP之上，它定义了请求和响应的内容。从技术上来说，基于其他底层协议使用HTTP也是可以的，但在绝大多数情况下，HTTP都是通过TCP发送数据的。我们将会处理TCP中的原始字节并与HTTP请求及响应打交道。

监听TCP连接

由于Web服务器需要监听TCP连接，所以让我们从这里开始着手。标准库提供了一个可以完成该任务的std::net模块。下面还是按照惯例创建一个新项目：

```
$ cargo new hello
Created binary (application) `hello` project
$ cd hello
```

将示例20-1中的代码输入 `src/main.rs` 中。这段代码会在地址 `127.0.0.1:7878` 上监听传入的TCP流，并在获取到新的TCP流时打印出 `Connection established!`。

src/main.rs

```
use std::net::TcpListener;

fn main() {
    ❶ let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    ❷ for stream in listener.incoming() {
        ❸ let stream = stream.unwrap();
        ❹ println!("Connection established!");
    }
}
```

示例20-1：监听传入的TCP流，并在接收到流时打印信息

通过使用 `TcpListener`，我们得以在地址 `127.0.0.1:7878` 上监听 TCP 连接❶。这个地址中冒号前面的部分是一个代表了当前设备的 IP 地址（这一地址在每台计算机上都是相同的，并不特指作者的计算机），随后的部分则是端口号 7878。我们出于两个原因选择了这个端口号：HTTP 可以正常地监听这个端口，而 7878 恰好是 `rust` 这 4 个字母在（9 宫格）电话上打出时的按键。

与 `new` 函数类似，代码中的 `bind` 函数会返回一个新的 `TcpListener` 实例。之所以选择 `bind` 作为函数的名称是因为在网络领域中，连接到端口这一行为也被称作“绑定到端口”（binding to a port）。

`bind` 函数的返回值类型 `Result<T, E>` 意味着绑定操作是有可能失败的。比如，连接到端口 80 需要管理员权限（非管理员只能监听大于 1024 的端口），当我们以非管理员身份尝试连接到 80 端口时就会被系统拒绝从而失败。另外，假如我们运行了 2 个监听到同一地址上的程序实例，那么绑定也不会成功。你可以先暂时忽略这些错误，因为本章的目标只在于学习并编写一个基本可用的服务器；而我们使用的 `unwrap` 函数会在错误发生时简单地结束程序。

`TcpListener` 上的 `incoming` 方法会返回一个产生流序列的迭代器❷（更准确地说，是 `TcpStream` 类型的流）。单个流（`stream`）代表了一个在客户端和服务器之间打开的连接。而连接（`connection`）则代

表了客户端连接服务器、服务器生成响应，以及服务器关闭连接的全部请求与响应过程。为此，`TcpStream`会读取自身的数据来观察客户端发送的内容，并允许我们将响应写回到流上去。简单来说，上面代码中的`for`循环会依次处理每个连接，并生成一系列的流供我们处理。

在目前的流处理过程中，我们选择在出现任何错误的情形下调用`unwrap`来结束程序③；而在程序成功的情形下打印出一段信息④。随后的示例会为成功情形添加更多的功能。`incoming`方法之所以会在客户端连接服务器时产生错误，是因为我们并没有对连接本身进行遍历，而仅仅只是遍历了连接尝试（connection attempt）。连接过程可能会因为相当多的原因而失败，其中大部分都与操作系统相关。例如，许多操作系统都会限制同时打开的连接数，试图创建超过这个数目的新连接就会产生错误，直到某些已经打开的连接关闭为止。

让我们运行这段代码试试看！先在终端调用`cargo run`，然后使用网页浏览器打开地址`127.0.0.1:7878`。因为服务器现在还没有返回任何数据，所以此时的浏览器应该会显示出类似于“Connection reset”的错误提示信息。但当你把目光转移到终端时，应该会在浏览器连接到服务器时看到数条打印出的信息：

```
Running `target/debug/hello`  
Connection established!  
Connection established!  
Connection established!
```

单次的浏览器访问有时会产生多条信息输出，这是因为浏览器在请求一个页面的同时还会试图请求其他资源，比如显示在浏览器标签上的`favicon.ico`图标文件等。

当然，这也有可能是因为浏览器没有接收到服务器返回的任何数据而尝试进行了多次连接导致的。`stream`的连接会在它离开作用域（也就是本例中循环结束的地方）时关闭，而浏览器则有可能会在连接关闭后尝试重新连接，因为导致连接断开的问题有可能是临时的。但不管怎样，重要的是我们现在已经成功地处理了TCP连接！

记得在运行完特定版本的代码后在终端按下`CTRL+C`组合键来结束程序，并在完成代码更新后重新使用`cargo run`启动服务。

读取请求

接下来，让我们开始实现从浏览器读取请求的功能。为了把处理连接的代码和其他工作分开，我们可以用一个单独的函数来处理连接。在这个新的handle_connection函数中，我们会从TCP流内读取数据并将它们打印出来，以便你观察浏览器发送过来的这些数据。将代码修改为示例20-2中的样子。

src/main.rs

```
❶use std::io::prelude::*;
use std::net::TcpStream;
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();

        ❷ handle_connection(stream);
    }
}

fn handle_connection(❸mut stream: TcpStream) {
    ❹ let mut buffer = [0; 512];

    ❺ stream.read(&mut buffer).unwrap();
    ❻ println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}
```

示例20-2：从TcpStream中读取并打印数据

为了使用读写流相关的trait，我们首先需要将std::io::prelude内的条目引入作用域❶。然后，我们会使用stream来调用新的handle_connection函数❷，而不是在main函数的for循环中简单地打印连接信息。

handle_connection函数中的stream参数被声明为了可变的❸，因为TcpStream实例的内部记录了返回给我们的数据，它可能会读取多于我们请求的数据，并将这些数据保存下来以备下次请求时使用。因为TcpStream的内部状态可能会被改变，所以我们需要将它标记为mut。虽然一般的读取操作不需要可变性，但此处是个例外。

接下来可以从流中读取实际的数据了，我们会通过两个步骤来完成这一任务：首先，我们在栈上声明了一个用于存放数据的buffer④。这是一个512字节的缓冲区，它足以存放基本的请求数据并满足本章的需要。如果你想要处理任意大小的请求，那么相关的缓存管理会更为复杂一些；我们暂时维持现状就好。接着，我们使用缓冲区调用了stream.read，它会从TcpStream中读取数据并将其存储至缓冲区中⑤。

第二步，我们将缓冲区中的字节转换成字符串并打印了出来⑥。函数String:: from_utf8_lossy可以接收一个&[u8]并产生对应的String。它名字中的“lossy”部分暗示了这个函数遇到无效UTF-8序列时的行为：它会用◆（U+FFFD REPLACEMENT CHARACTER）来替换所有无效的序列。你可能会在缓冲区中那部分没有被请求数据占据的地方看到这种替代字符。

让我们尝试运行这段新代码！启动程序并接着在网页浏览器中发起一个请求。注意，浏览器中仍然会出现错误页面，但程序在终端的输出会变为如下所示的样子：

```
$ cargo run

Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
????????????????????????????????????????????????????
```

不同的浏览器会产生些许不同的输出结果。现在我们已经打印出了请求数据，你可以通过观察Request: GET后面的路径来解释为何会从浏览器处得到多个连接。如果重复的连接都是请求 |，那么我们就可以猜测浏览器由于没有收到服务器响应而反复地请求获取 |。

让我们接着来分解这份请求数据，并尝试理解浏览器究竟在要求我们提供哪些内容。

仔细观察HTTP请求

HTTP是一个基于文本的协议，它的请求采用了如下所示的格式：

```
Method Request-URI HTTP-Version CRLF  
headers CRLF  
message-body
```

第一行被称作请求行（request line），它存放了客户端请求的信息。其中的第一部分表明了当前请求使用的方法，比如GET或POST，它描述了客户端请求数据的方式。这里的客户端使用GET请求。

接下来的部分| 代表了客户端正在请求的统一资源标识符（Uniform Resource Identifier, URI）：URI大体上类似于统一资源定位符（Uniform Resource Locator, URL），但完全一样。它们之间的差异对于本章要达到的目的不是那么重要，但由于HTTP标准使用了专门的术语URI，所以你可以将URI简单地理解为URL。

最后一部分是客户端使用的HTTP版本，接着，请求行就以CRLF序列结束了。CR与LF分别代表回车（Carriage Return）与换行（Line Feed），它们是从打字机时代传承下来的术语。CRLF序列也被写作|r|n，其中|r代表回车，|n代表换行。CRLF序列会将请求行和请求数据的其他部分区别开来。需要注意的是，我们会在打印CRLF时看到一个新行而不是字符|r|n。

观察示例中出现的请求行数据，我们可以看到方法是GET，请求的URI是|，版本号是HTTP| 1. 1。

在请求行结束之后，剩下那些从Host:开始的部分是HTTP附带的消息头。另外，GET请求还省略了自己的消息体。

你可以尝试使用一个不同的浏览器来发起请求，或者是更换一个不同的地址，比如127.0.0.1:7878/test，来看一看请求数据会发生什么样的改变。

在理解了浏览器的请求消息后，我们就可以来返回一些数据了！

编写响应

为了响应客户端请求，我们需要实现发送数据的功能。HTTP响应的格式如下所示：

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

第一行被称作状态行（status line），它包含了当前响应的HTTP版本、一个汇总了请求结果的数字状态码，以及一段提供了状态码文本描述的原因短语。状态行的CRLF序列之后是任意数量的消息头、另一个CRLF序列，以及响应消息体。

下面示例中的响应使用了HTTP 1.1版本，状态码为200，原因短语为OK，没有消息头与消息体：

```
HTTP/1.1 200 OK\r\n\r\n
```

状态码200被用作标准的成功响应码，而紧随其后的则是一段用于表示成功的袖珍HTTP响应。让我们把这些数据作为成功请求的响应写入流中！从handle_connection函数中移除打印请求数据的println!函数，并将它替换为示例20-3中的代码。

src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

① let response = "HTTP/1.1 200 OK\r\n\r\n";
② stream.write(response.as_bytes()③).unwrap();
④ stream.flush().unwrap();
}
```

示例20-3：向流中写入一个成功的HTTP响应

新增的第一行代码定义了包含成功响应数据的response变量①。由于stream的write方法只接收&[u8]类型值作为参数③，所以我们需要调用response的as_bytes方法来将它的字符串转换为字节，并将这些字节发送到连接中去②。

因为write操作可能会失败，所以我们如同往常一样使用了unwrap，它会在出现错误时简单地中止程序。当然，你需要在实际应

用中依据上下文添加恰当的错误处理逻辑。最后的flush调用会等待并阻止程序继续运行直到所有字节都被写入连接中❸；为了减少对底层操作系统的调用，`TcpStream`的实现中包含了一个内部缓冲区。

完成上述修改后，让我们再次运行代码并发起请求。由于新的代码不再向终端打印任何数据，所以我们除Cargo之外不会看到任何额外的输出。当你在浏览器中加载 `127.0.0.1:7878` 时，应该会获得一个空页面而不是错误，这也就意味着我们成功地编写了一段响应HTTP请求的代码！

返回真正的HTML

接下来，我们会实现返回更多内容的功能，而不仅仅只是返回简单的空白页面。创建一个名为 `hello.html` 的文件，并将它放置到项目根目录中（注意不是 `src` 目录）。你可以在其中输入任何你想要返回的HTML代码；示例20-4中展示了HTML文件中的一种可能的写法。

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20-4：一个用于在响应中返回的简单HTML文件

上面的示例展示了一个最小化的HTML5文档，它包含一个标题和一小段文本。为了在服务器处理请求时返回它，我们需要按照示例20-5所示的来修改 `handle_connection` 函数。新的函数会读取这个HTML文件，将其中的内容添加到响应中并作为消息体一起发送。

src/main.rs

```
❶use std::fs;
// --略

--



fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let contents = fs::read_to_string("hello.html").unwrap();

❷    let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

示例20-5：将文件hello.html中的内容作为消息体发送

我们在顶部增加的use语句会将标准库中的文件系统模块引入当前作用域中❶。你应该对随后这段打开和读取文件的代码比较熟悉，因为我们曾经在第12章的示例12-4中使用过它们。

接着，我们使用format! 把文件的内容添加为成功响应的消息体❷。

通过使用cargo run运行代码并在浏览器中加载127.0.0.1:7878，你应该就能够看到渲染出来的HTML页面了。

目前，我们忽略了buffer中的请求数据并无条件地返回了HTML文件中的内容。即便浏览器尝试请求的地址是127.0.0.1:7878/something-else，它也仍然会得到同样的HTML响应。我们服务器的功能非常有限且不同于大部分的Web服务器。接下来，我们会根据请求来自定义返回的响应数据，并只对格式正确的请求返回之前的HTML文件。

验证请求有效性并选择性地响应

目前的Web服务器会统一返回HTML文件中的内容，而不关心客户端请求的具体数据。现在，让我们在返回数据前添加检测功能：只在浏览器请求时返回HTML文件中的内容，而在其他情形下返回错误提示信息。为了达到这一目的，我们需要修改handle_connection函数，如示

例20-6所示。新的代码会将接收到的请求内容与已知的 请求进行对比，并在随后的if与else块中做出相对应的处理。

src/main.rs

```
// --略
--



fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

❶ let get = b"GET / HTTP/1.1\r\n";
❷ if buffer.starts_with(get) {
    let contents = fs::read_to_string("hello.html").unwrap();
    let response = format!("HTTP/1.1 200 OK\r\n{}\r\n", contents);
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
❸ } else {
    // 一些其他的请求
}

}
```

示例20-6：匹配和处理请求，对 的处理要与其他请求不同

这段代码首先将 请求的相关数据硬编码到了变量get中❶。由于缓冲区中接收的数据是原始字节，所以我们使用字节字符串语法b""将get的文本内容转换为字节字符串。接着，我们开始检查buffer中的数据是否以get中的字节开头❷。如果答案是肯定的，那么就意味着我们接收到了一个符合规范的 请求。随后的if块中放置了处理此种情形的代码，它会返回HTML文件中的内容。

如果buffer没有 以get中的字节开头，那么就意味着我们接收到了其他请求。else块❸中的代码会对这些异常请求做出处理。

再次运行代码并访问 `127.0.0.1:7878`，你应该会获得 `hello.html` 文件中的 HTML 内容。当你请求其他地址时，比如 `127.0.0.1:7878/something-else`，则会得到类似于示例20-1或示例20-2的连接错误。

现在，向示例20-7的else块中添加代码来返回一个带有状态码404的响应，它表明当前请求的内容没有找到。接着，我们还会返回一个可渲染在浏览器中的HTML页面来提示终端用户。

src/main.rs

```
// --略  
--  
} else {  
❶ let status_line = "HTTP/1.1 404 NOT FOUND\r\n\r\n";  
❷ let contents = fs::read_to_string("404.html").unwrap();  
  
let response = format!("{}{}", status_line, contents);  
  
stream.write(response.as_bytes()).unwrap();  
stream.flush().unwrap();  
}
```

示例20-7：在请求其他路径时返回状态码404与一个错误页面

我们在此种情形下的响应会包含状态码404，以及原因短语NOT FOUND❶。它依然没有消息头，但会在消息体中附带文件*404.html*中的内容❷。你需要在*hello.html*的同级目录下创建一个新的*404.html*文件。你依然可以在这个文件中使用任何HTML代码或示例20-8中的HTML范本。

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20-8：404响应的示例内容

基于这些修改重新运行你的代码。现在请求*127.0.0.1:7878*依然会返回*hello.html*的内容，但在其他情形下，比如请求*127.0.0.1:7878/foo*，则会返回*404.html*文件中的HTML内容。

少许重构

目前的if与else块中存在不少重复代码：除了状态行和文件名不同，它们都在读取文件并把其内容写入流中。为了使代码变得更加紧凑一些，我们可以把存在差异的部分提取至独立的if与else块中，并将它们赋值给相应的变量。随后，我们就可以在读取文件和写入响应时无条件地使用这些变量了。重构后的代码如示例20-9所示。

src/main.rs

```
// --略  
--  
fn handle_connection(mut stream: TcpStream) {  
    // --略  
    --  
    let (status_line, filename) = if buffer.starts_with(get) {  
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")  
    } else {  
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")  
    };  
    let contents = fs::read_to_string(filename).unwrap();  
    let response = format!("{}{}", status_line, contents);  
    stream.write(response.as_bytes()).unwrap();  
    stream.flush().unwrap();  
}
```

示例20-9：重构if和else块，只在分支代码中包含有区别的部分

新的if和else块返回了一个由状态行和文件名组成的元组，并通过在第18章讨论的let语句将它们分别解构到了变量status_line和filename中。

之前重复的代码现在位于if与else块外，并使用了status_line和filename变量。这样的写法使我们更容易观察两种情况的不同之处；而当你想要修改读取文件或写入响应的逻辑时，只需要修改其中的一处地方即可。示例20-9中的代码行为与示例20-8中的完全一致。

非常棒！我们仅用了大约40行Rust代码就实现了一个简易的Web服务器，它对某个请求返回特定页面，并对其余所有请求返回404。

目前的服务器运行在单线程上，这意味着它一次只能处理一个请求。接下来，我们会通过模拟一些较慢的请求来暴露出这种处理方式可能发生的问题。当然，我们最终会修复这些问题并使服务器能够同时处理多个请求。

把单线程服务器修改为多线程服务器

目前，我们的服务器会依次处理各个请求，这也就意味着它在处理完第一个请求前不会处理第二个连接。服务器接收到的请求越多，这类串行操作就会使整体性能越差。当服务器接收到某个需要处理很长时间的请求时，其余的请求就不得不排队进行等待，即便新请求可以被快速处理完毕。我们最终会修复这一问题，但在这之前，让我们先来观察一下这一问题的具体行为。

在现有的服务器实现中模拟一个慢请求

我们会在现有的服务器实现中模拟一个慢请求，并观察它会如何影响到后续的其他请求。示例20-10在实现 sleep请求的处理逻辑中模拟了一段较慢的响应，它会让服务器在完成响应前休眠5秒钟。

src/main.rs

```
use std::thread;
use std::time::Duration;
// --略

fn handle_connection(mut stream: TcpStream) {
    // --略

    let get = b"GET / HTTP/1.1\r\n";
    ① let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        ③ thread::sleep(Duration::from_secs(5));
        ④ ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    }
}
```

```
};

// --略

--
```

示例20-10：通过识别 sleep请求并休眠5秒钟来模拟慢请求

这段代码稍微有些散乱，但已经足以实现模拟的目的了。我们在代码中创建了第二个请求sleep❶，其数据可被服务器识别，然后又在if块的后面添加了一个else if来检查| sleep请求❷。服务器会在接收到这一请求后休眠5秒钟❸，并接着渲染响应成功的HTML页面❹。

你现在能够看到我们的服务器有多么初级了，真实的库会以更简捷的方式来识别不同的请求！

使用cargo run启动服务器，接着打开两个浏览器窗口：一个请求 `http://127.0.0.1:7878/`，另一个请求 `http://127.0.0.1:7878/sleep`。如果你和之前一样反复地输入URI，那么你应该会非常迅速地获得响应结果。但如果你在加载页面之前加载了sleep，那么你就会观察到需要花费至少5秒钟才能渲染出成功响应的HTML页面。

有许多方式可以避免慢请求阻塞随后的请求队列，我们选择通过实现线程池来解决这一问题。

使用线程池改进吞吐量

线程池（thread pool）是一组预先分配出来的线程，它们被用于等待并随时处理可能的任务。当程序接到一个新任务时，它会将线程池中的一个线程分配给这个任务，并让该线程处理这个任务。线程池中其余可用的线程能够在第一个线程处理任务时接收其他到来的任务。当第一个线程处理完它的任务后，我们会将它放回线程池，并使其变为空闲状态以准备处理新的任务。一个线程池允许你并发地处理连接，从而增加服务器的吞吐量。

我们会将池中线程的数量限制为一个较小的值，以避免受到拒绝服务（Denial of Service, DoS）攻击。如果程序为每一个接收的请求都创建了相应的线程，那么恶意攻击者就可以同时创建出成千上万个请求来耗尽服务器资源并最终导致所有请求中断。

不同于无限制地创建线程，线程池中只会有固定数量的等待线程。新连接进来的请求会被发送至线程池中处理，而线程池则维护了一个接收请求的队列。池中可用的线程会从这个请求队列中取出请求并处理，然后再向队列索要下一个请求。基于这种设计，我们可以同时处理 N 个请求，这里的 N 也就是线程数量。当所有的线程都在处理慢请求时，随后的请求依然会被阻塞在等待队列中。虽然没能完全避免阻塞的出现，但我们增加了可同时处理的慢请求数量。

这种用来改进服务器吞吐量的技术仅仅是众多可用方案中的一种。其他可供你深入研究的方向包括fork/join模型与单线程异步I/O模型。假如你对这个主题感兴趣，你可以尝试阅读这些方案的相关材料并使用Rust来实现它们。对于Rust这样一个底层语言来讲，所有这些模型都是可实现的。

在开始实现一个线程池前，让我们先讨论一下线程池的使用方式应该是什么样子的。提前编写客户端接口有助于指导代码设计。你可以先以期望的调用方式来组织构建API，并接着实现具体的功能，而不是先实现功能然后再设计公共API。

类似于在第12章使用的测试驱动开发，我们会在那里用到编译器驱动开发（compiler-driven development）。我们将编写代码来调用期望中的函数，并依据编译器的错误提示信息来修改代码直到一切正常。

为每个请求创建独立线程时的代码结构

首先，让我们来看一看为每个连接创建独立线程时可能编写出的代码。正如之前提到过的，这种方案具有潜在的风险：它可能会导致系统无止境地创建线程。因此，我们不会把这一方案视作最终的实现目标，而是一个起点。示例20-11展示了main函数中的改动，它在for循环中为每个流创建了独立的新线程来进行连接处理。

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20-11：为每个流创建新线程

我们在第16章曾经讨论过，`thread::spawn`会创建一个新线程并在新线程中执行闭包内的代码。当你运行这段代码，并接着在浏览器中依次打开 `| sleep` 页面与 `|` 页面时，你会观察到 页面非常快速地响应了我们的请求，没有等待 `| sleep` 页面加载完毕。但正如之前提到过的，这种方案可能会导致系统崩溃，因为它对新线程的数量没有任何限制。

用有限数量的线程创建类似接口

我们希望采用线程池的方案也能用类似的方式运行，以避免在切换方案时对使用该API的代码做出较大的修改。示例20-12展示了一个`ThreadPool`结构体的假想接口，它被用来替换`thread::spawn`。

src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
❶    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

❷        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

示例20-12：假想的`ThreadPool`接口

我们在上面的代码中使用了`ThreadPool::new`来创建一个可配置线程数量的线程池，并在本例中将线程数量配置为4①。在随后的`for`循环中，`pool.execute`的接口与`thread::spawn`的完全一致，它会接收一个处理所有流的闭包②。我们需要实现`pool.execute`来接收闭包并将它分配给池中的线程执行。虽然这段代码还无法通过编译，但我们仍然可以不断地尝试，编译器的错误提示信息会帮助我们逐步修复错误。

使用编译器驱动开发来构建`ThreadPool`结构体

按照示例20-12修改代码后，我们可以使用`cargo check`来查看编译错误并驱动下一步的开发。下面是我们得到的第一条错误提示信息：

```
$ cargo check
```

```
Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module `ThreadPool`
--> src\main.rs:10:16
   |
10 |     let pool = ThreadPool::new(4);
   |           ^^^^^^^^^^^^^^ Use of undeclared type or module
   |           `ThreadPool`'
error: aborting due to previous error
```

很好！这段错误提示信息指出代码缺少了对应的`ThreadPool`类型或模块，现在就让我们来创建一个。因为`ThreadPool`的实现是Web服务器正在做的这类工作中的独立部分，所以我们可以把`hello`包从二进制模式切换为库模式来存放`ThreadPool`实现。这也意味着我们可以在更多的工作中用到这一独立的线程池，而不仅仅是在处理网络请求时使用。

创建一个含有下列代码的`src/lib.rs`文件，它包含了`ThreadPool`结构体的极简化定义：

src/lib.rs

```
pub struct ThreadPool;
```

接着创建一个新目录`src/bin`，并将二进制包的根目录从`src/main.rs`移动至`src/bin/main.rs`。这一操作会使得代码包成为

hello 目录中的主包，我们依然可以使用 cargo run 来运行 *src/bin/main.rs* 中的二进制文件。移动完 *main.rs* 文件后，在 *src/bin/main.rs* 的顶部添加如下所示的语句来引入代码包及其中的 ThreadPool：

src/bin/main.rs

```
use hello::ThreadPool;
```

这段代码依然无法通过编译，让我们继续运行 cargo check 并观察出现的错误提示信息：

```
$ cargo check
```

```
Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for type
`hello::ThreadPool` in the current scope
--> src/bin/main.rs:13:16
   |
13 |     let pool = ThreadPool::new(4);
   |           ^^^^^^^^^^^^^^^^^ function or associated item not
found in
`hello::ThreadPool`
```

新的编译错误提示信息指出了我们接下来需要完成的工作：为 ThreadPool 创建一个名为 new 的关联函数，它应当能够接收 4 作为参数并返回新的 ThreadPool 实例。让我们来实现一个满足此功能的最简化的 new 函数：

src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

由于负的线程数量没有任何意义，所以我们选择了 `usize` 作为 `size` 参数的类型。另外，我们知道调用函数的代码会传入数字 4 作为线程集合的元素数量，所以采用 `usize` 类型是合适的，正如第 3 章的“整数类型”一节中所讨论的那样。

让我们再次运行指令检查代码：

```

$ cargo check

     Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
--> src/lib.rs:4:16
|
4 |     pub fn new(size: usize) -> ThreadPool {
|           ^^^^
|
|= note: #[warn(unused_variables)] on by default
= note: to avoid this warning, consider using `'_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in the
current scope
--> src/bin/main.rs:18:14
|
18 |         pool.execute(|| {
|           ^^^^^^
|

```

新的编译输出中出现了一个警告和一个错误。先暂时忽略警告不管，此处发生的错误指出ThreadPool结构体中不存在可用的execute方法。前面的“用有限数量的线程创建类似接口”一节中曾经提到过，我们希望让线程池的接口与thread::spawn尽量保持一致。另外，我们的execute函数会接收一个闭包作为参数并在内部将它分配给池中空闲的线程去执行。

我们会在ThreadPool上定义execute方法，使其可以接收一个闭包作为参数。正如第13章的“使用泛型参数和Fn trait来存储闭包”一节中讨论的那样，在将闭包作为参数时我们可以选择使用3种不同的trait: Fn、FnMut、FnOnce。由于最终的execute实现会类似于标准库中的thread::spawn实现，所以我们可以参考thread::spawn的函数签名来决定究竟使用哪一种约束：

```

pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static

```

将注意力集中到签名中的类型参数F，另外的那个类型参数T仅仅与返回值有关，先暂时忽略它就好。你可以观察到spawn使用了FnOnce作为F的trait约束。这极有可能也是我们需要使用的trait，因为execute最终会把自己获得的参数传递给spawn。另外，处理请求的线程只会执行一次闭包，它符合FnOnce中Once的含义，这进一步确认了FnOnce就是我们需要的trait。

除了要满足FnOnce trait约束，类型参数F还需要满足Send trait约束及生命周期' static。这也是我们需要为当前场景添加的约束条件：满足Send约束的闭包才可以从一个线程传递至另一个线程；而由于我们不知道线程究竟会执行多长时间，所以闭包必须是' static的。现在，让我们在ThreadPool结构体中实现一个带有泛型参数F的execute方法，并在参数上添加相应的约束条件：

src/lib.rs

```
impl ThreadPool {
    // --略

    --
    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce()① + Send + 'static
    {
    }
}
```

FnOnce①后的()意味着传入的闭包既没有参数，也不返回结果。就像函数定义一样，我们可以省略签名中的返回值，但却不能省略函数名后的圆括号，即便括号中没有任何参数。

再次声明，我们仅仅实现了最简单的execute方法：它能够让我们的代码通过编译，但不会执行任何指令。让我们再次运行检查命令：

```
$ cargo check
```

```
Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
--> src/lib.rs:4:16
|
4 |     pub fn new(size: usize) -> ThreadPool {
|           ^^^^
|
|= note: #[warn(unused_variables)] on by default
|= note: to avoid this warning, consider using `\_size` instead

warning: unused variable: `f`
--> src/lib.rs:8:30
|
8 |     pub fn execute<F>(&self, f: F)
|           ^
|
|= note: to avoid this warning, consider using `\_f` instead
```

新的输出中只剩下警告了，这也就意味着我们的代码顺利通过了编译。但需要注意的是，当你尝试执行cargo run并在浏览器中发起请求时，你会在浏览器中观察到在本章开头看到的那个错误，因为我们还没有调用过传递给execute的闭包！

注意

你也许听到过这样的说法：对于Haskell和Rust这样拥有严格编译检查的语言来讲，“只要编译通过，它就可以正常工作”。这段论述并不完全正确，上面的代码通过了编译，但却什么都没做！假设你正在构建一个完整的真实项目，那么现在就是编写单元测试最好的时机，我们需要借助它来检查代码能否编译通过并且拥有预期的行为。

在new中验证线程数量

我们之所以会看到编译警告，是因为new和execute的参数还没有被任何地方使用过。现在，让我们接着在函数体中实现期望的行为。先将注意力集中到new函数上。之前我们为size参数选择了无符号整数类型，因为一个线程数量为负的线程池结构毫无意义。然而，线程数为0的线程池同样也没有意义，但0却是一个合法的usize值。因此，我们需要在返回ThreadPool实例前检查size的值是否大于0，并在接收到0时调用assert! 宏中断程序，如示例20-13所示。

src/lib.rs

```

impl ThreadPool {
    /// 创建线程池。
    ///
    /// 线程池中线程的数量。
    ///
❶    /// # Panics
    ///
    /// `new`函数会在 size 为 0 时触发 panic。
    pub fn new(size: usize) -> ThreadPool {
❷        assert!(size > 0);

        ThreadPool
    }

    // --略--
}

```

示例20-13：让ThreadPool::new在size为0时中断程序

这段代码使用文档注释语法为ThreadPool添加了一些文档。通过添加一个文档区域来列举函数可能触发panic的情形，我们进行了较为良好的文档实践❶，正如在第14章讨论的那样。你可以试着运行cargo doc --open，并点击ThreadPool结构体来查看生成的new函数文档。

另外，我们也可以在new函数中返回一个Result而不再使用assert!宏❷，正如I/O项目中示例12-9为Config::new设计的那样。但按照目前的设计来看，试图创建没有任何线程的线程池是一个不可恢复错误。当然，你也可以试着去编写拥有如下签名的new函数，并比较它与当前版本之间的异同：

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

创建用于存放线程的空间

基于合法的线程数目，我们可以在返回ThreadPool前创建这些线程，并将它们存储到ThreadPool结构体中。但究竟应该如何“存储”一个线程呢？让我们再来看一看thread::spawn的签名：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static
```

spawn函数会返回一个JoinHandle<T>，其中的T是闭包的返回值类型。我们可以试着使用JoinHandle来存储线程并看一看会发生些什么。由于线程池中的闭包只会被用来处理连接而没有返回值，所以JoinHandle<T>中的T就是单元类型()。

示例20-14中的代码能够正常通过编译，但依然没有创建任何线程。新修改的ThreadPool定义包含了一个thread::JoinHandle<()>的动态数组实例，我们会使用参数size来初始化这个动态数组的容量。随后，我们还会使用for循环来创建线程，并最终返回包含它们的ThreadPool实例。

src/lib.rs

```
❶use std::thread;

pub struct ThreadPool {
   ❷    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --略

    --
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

   ❸    let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // 创建线程并将它们存储至动态数组中

        }

        ThreadPool {
            threads
        }
    }
    // --略
}
```

示例20-14：为ThreadPool创建一个动态数组来存放线程

上面的代码还将std::thread引入了作用域❶，因为我们需要使用thread::JoinHandle来作为ThreadPool中动态数组的元素类型❷。

一旦得到了合法的数量参数，ThreadPool就可以创建出包含size个元素的动态数组③。此处用到的with_capacity函数还没有在本书中出现过，它与Vec::new有些类似，但区别在于with_capacity会为动态数组预分配出指定的空间。在知晓存储大小的前提下预先分配存储空间要比使用Vec::new在插入时动态扩展大小更有效率一些。

再次运行cargo check，虽然还能看到一些警告消息，但你应该可以成功编译这段代码。

负责将代码从ThreadPool传递给线程的Worker结构

示例20-14的for循环中留下了一行关于创建线程的注释。现在，让我们来看一看如何真正地创建线程。标准库提供了一个用于创建线程的thread::spawn函数，它会在线程创建完毕后立即执行自己接收到的代码参数。然而在当前情形下，我们需要线程在创建后进入等待状态并执行随后传递给它的代码。标准库中的线程并没有包含这些功能，我们必须手动实现它们。

我们会在ThreadPool与线程之间引入一个新的数据结构来实现并管理上述行为。我们选择了一个线程池实现中的通用术语Worker来命名这一数据结构。想象一下在餐厅厨房中工作的人们：员工们(Workers)会持续地等待顾客的订单，并在订单出现后负责接收并完成它们。

新的代码会在线程池的动态数组中存储Worker结构体实例，而不再是JoinHandle<()>实例。每个Worker都会在内部维护自己的JoinHandle<()>实例。接着，我们将在Worker结构体上实现一个接收闭包的方法，它会将闭包发送到已经在运行的线程中去执行。为了便于在记录日志和调试时区分不同的Worker实例，我们为每个Worker都赋予了独立的id。

让我们在创建ThreadPool时首先完成下面的修改。在按照如下方式设置完Worker后，我们再来实现发送闭包到线程中去的代码：

1. 定义持有id和JoinHandle<()>的Worker结构体。
2. 修改ThreadPool的实现来存放一个Worker实例的动态数组。

3. 定义一个接收id数字的Worker::new函数，它会返回一个持有该id的Worker实例，这个实例中还附带了一个由空闭包创建而成的线程。

4. 在ThreadPool::new中使用for语句循环创建id并生成相应的Worker，再将Worker实例存储到动态数组中。

如果你渴望挑战的话，那么你可以在查看示例20-15中的代码前自行完成这些修改。

准备好了吗？示例20-15中的代码完成了上述规划中的那些改动。

src/lib.rs

```
use std::thread;

pub struct ThreadPool {
    ❶ workers: Vec<Worker>,
}

impl ThreadPool {
    // --略

    --
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        ❷ for id in 0..size {
            ❸ workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
    }
    // --略

    --
}

❹ struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    ❺ fn new(id: usize) -> Worker {
        ❻ let thread = thread::spawn(|| {});

        Worker {
            ❼ id,
```

```
    ⑧ thread,  
}  
}  
}
```

示例20-15：修改ThreadPool的实现来存放Worker实例，而不是直接持有线程

由于修改后的ThreadPool持有Worker实例而不是JoinHandle<()>实例，所以我们将对应的字段名称从threads改为了workers①。代码中还使用了for循环②中的计数器作为Worker::new的参数，并将创建出来的Worker实例逐一存储到了名为workers的动态数组中③。

因为外部代码（比如*src/bin/main.rs*中的服务器）在使用ThreadPool时，并不需要知道Worker的具体实现细节，所以Worker结构体④和它的new函数⑤都保持了私有性。Worker::new函数接收传递给它的id作为参数⑦，并存储了一个由空闭包⑥创建而成的JoinHandle<()>实例⑧。

这段代码可以通过编译，并基于传递给ThreadPool::new的参数来创建对应数目的Worker实例，但我们仍然没有处理execute函数中的闭包。接下来让我们看一看如何实现这一需求。

使用通道把请求发送给线程

虽然我们在execute方法中获得了期望执行的闭包，但代码在创建ThreadPool并进一步创建Worker时给thread::spawn传入的闭包实际上并没有执行任何指令。现在就让我们来解决这一问题。

我们希望刚刚创建的Worker结构体能够从存储在ThreadPool的队列中获取需要执行的代码，并将它们发送到线程中运行。

第16章曾经介绍过一个用于线程间通信的简单方式：通道，它在当前的场景中非常适用。我们会将通道用作一个普通的任务队列，由execute方法将任务从ThreadPool发送到Worker实例，并最终发送到具体的线程中去。具体的计划如下所示：

1. 由ThreadPool创建通道并持有通道的发送端。

2. 生成的每个Worker都会持有通道的接收端。
3. 创建一个新的Job结构体来持有需要发送到通道中的闭包。
4. 在execute方法中将它想要执行的任务传递给通道的发送端。
5. Worker会在自己的线程中不断地查询通道接收端，并执行收取到的闭包任务。

让我们首先在Thread::new中创建一个通道，并将通道的发送端存储在ThreadPool实例中，如示例20-16所示。通道使用了Job结构体作为传递数据的类型，虽然这段代码中的Job结构体还没有添加任何内容。

src/lib.rs

```
// --略

use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --略

    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ① let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            ② sender,
        }
    }
    // --略
}

--
```

示例20-16：修改ThreadPool来存储一个用于发送Job实例的通道发送端

上面的ThreadPool::new中创建了一个新的通道❶，并在线程池中持有了这个通道的发送端❷。这段代码能够通过编译，但仍然会产生一些警告信息。

接下来，让我们试着在创建通道时将它的接收端传递给每一个工作线程。由于我们希望在工作线程中使用这些接收端，所以闭包中直接引用了receiver参数。示例20-17中的代码还无法通过编译。

src/lib.rs

```
impl ThreadPool {
    // --略

    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 {
            ❶ workers.push(Worker::new(id, receiver));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --略

    --
}

// --略

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            ❷ receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}
```

```
    }
}
```

示例20-17：将通道的接收端传递给工作线程

我们在这段代码中做出了一些细微但直接的修改：将通道的接收端传递给了`Worker::new`❶并接着在闭包中使用了接收端❷。

运行命令检查这段代码，你会得到如下所示的编译错误：

```
$ cargo check

Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:27:42
  |
27 |         workers.push(Worker::new(id, receiver));
  |                           ^^^^^^^^^^ value moved here in
previous iteration of loop
  |
= note: move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
```

我们的代码会尝试将`receiver`传递给多个`Worker`实例，这可行不通。回忆一下第16章中的内容：Rust提供的通道是多生产者、单消费者的，这也意味着你不能简单地通过克隆接收端来解决上述问题。即便可以，那也不是我们想要使用的技术；我们希望在所有的工作线程中共享同一个`receiver`，从而能够在线程间分发任务。

另外，从通道队列中取出任务意味着`receiver`是可变的，所以线程需要一个安全的方式来共享和修改`receiver`，否则我们就可能会触发竞争状态（参考第16章）。

再回忆一下在第16章讨论过的线程安全智能指针：为了在多个线程中共享所有权并允许线程修改共享值，我们可以使用`Arc<Mutex<T>>`。`Arc`类型允许多个工作线程拥有同一个接收者，而`Mutex`则保证了一次只有一个工作线程能够从接收端得到任务。示例20-18展示了我们所做的修改。

src/lib.rs

```
use std::sync::Arc;
use std::sync::Mutex;
// --略
```

```

-->
impl ThreadPool {
    // --略

-->
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        ① 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)②));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    // --略
}

-->
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --略
    }
}

```

示例20-18：使用Arc和Mutex在所有工作线程中共享通道的接收端

ThreadPool中的代码将通道的接收端放入了Arc和Mutex中①，并在创建新的Worker时克隆Arc来增加引用计数，从而使所有的工作线程可以共享接收端的所有权②。

经过修改，代码终于能够通过编译了！我们做到了！

实现execute方法

最后，让我们来实现ThreadPool中的execute方法。同时，我们也将Job从结构体修改为了一个trait对象的类型别名，它的实例能够在内部持有传递给execute的闭包。第19章的“使用类型别名创建同义类

型”一节中曾经提到过，类型别名允许我们简化一个较长的类型名称，如示例20-19所示。

src/lib.rs

```
// --略
--



type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --略
    --



    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        ❶ let job = Box::new(f);

        ❷ self.sender.send(job).unwrap();
    }
}

// --略
--
```

示例20-19：为存放闭包的Box创建一个类型别名，并接着在通道中发送任务

execute方法在得到闭包后会创建出一个新的Job实例❶，并将这个任务传递给通道的发送端❷。为了应对发送失败的情形，我们在send后直接调用了unwrap。发送失败确实有可能会出现，比如当所有执行线程停止运行时，这意味着接收端停止了接收新的消息。但就目前来讲，我们无法中断运行的线程：只要线程池存在，池中的线程就会持续地执行。即便我们知道这种失败情形不会发生也仍然需要使用unwrap，因为编译器不知道这些业务相关的信息。

事情到此还没有结束！Worker传递给thread::spawn的闭包仅仅引用了通道的接收端，而我们需要这个闭包不断地查询通道的接收端，并在获得任务时立即执行。示例20-20展示了Worker::new中的相关修改。

src/lib.rs

```
// --略

-->

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock()①.unwrap()②.recv()③.unwrap()④;

                println!("Worker {} got a job; executing.", id);

                job();
            }
        });
        Worker {
            id,
            thread,
        }
    }
}
```

示例20-20：在Worker的线程中接收并执行任务

这段代码首先调用了receiver的lock方法来请求互斥锁①，并接着使用unwrap来处理可能出现的错误情形②。请求获取锁的操作会在互斥体被污染时出错，而互斥体会在某个持有锁的线程崩溃而锁没有被正常释放时被污染。在这种情形下，调用unwrap触发当前线程的panic是非常恰当的行为。当然，你也可以将unwrap修改为expect来附带一个有意义的错误提示信息。

在互斥体上得到锁以后，我们就可以通过调用recv来从通道中接收Job③了。与发送端的send方法类似，recv会在持有通道发送端的线程关闭时出现错误，所以我们同样使用了unwrap来拦截所有错误④。

调用recv会阻塞当前线程，当通道中不存在任务时，当前线程就会一直处于等待状态。而Mutex<T>则保证了一次只有一个Worker线程尝试请求任务。

基于这一巧妙的实现，我们的线程池现在可以正常工作了！执行cargo run并发起一些请求：

```
$ cargo run
```

```
Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers`
--> src/lib.rs:7:5
|
7 |     workers: Vec<Worker>,
|     ^^^^^^^^^^^^^^^^^^^^^^
|
|= note: #[warn(dead_code)] on by default

warning: field is never used: `id`
--> src/lib.rs:61:5
|
61 |     id: usize,
|     ^^^^^^^
|
|= note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
--> src/lib.rs:62:5
|
62 |     thread: thread::JoinHandle<()>,
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|
|= note: #[warn(dead_code)] on by default

Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
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！我们现在拥有了一个可以异步执行请求的线程池。由于它最多只会创建4个线程，所以即便服务器接收到了大量的请求也不会导致系统负载超过极限。当我们请求`sleep`时，服务器可以同时响应新的请求并启用其他线程来执行它们。

注意

如果你在多个浏览器窗口中同时打开`sleep`，它们可能会彼此间隔地加载5秒钟，这是因为一些网页浏览器出于缓存的原因会顺序地执行相同请求的多个实例。我们的Web服务器不会产生这些限制。

在学习了第18章中介绍的while let循环后，你也许会好奇为什么我们不把工作线程编写成示例20-21中所示的样子。

src/lib.rs

```
// --略

-->
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 {} got a job; executing.", id);

                job();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

示例20-21：使用while let实现的Worker::new

这段代码能够顺利通过编译并运行，但却不会产生我们期望的线程行为：一个慢请求依旧会导致其他请求被阻塞等待。其原因有些微妙：Mutex结构体不存在公开的unlock方法，因为锁的所有权依赖于MutexGuard<T>的生命周期，而你只能在lock方法返回的LockResult<MutexGuard<T>>中得到它。这使得编译器能够在编译过程中确保我们只有在持有锁时才能访问由Mutex守护的资源。但假如你没有妥当地设计好MutexGuard<T>的生命周期，那么这种实现也可能会让我们意外地逾期持有锁。在本例中，由于while表达式内的值会把整个代码块视作自己的作用域，所以我们在调用job()的过程中仍然持有锁，这也就意味着其他工作线程无法正常地接收任务。

通过使用loop并在循环代码块内部而不是外部请求锁和任务，lock方法中返回的MutexGuard会在let job语句结束后被立即丢弃。这确保了我们只会在调用recv的过程中持有锁，并能够在调用job()之前将锁释放。因此，我们的服务器才可以同时响应多个请求。

优雅地停机与清理

正如我们期望的那样，示例20-20中的代码能够通过线程池来异步地响应请求。但Rust依然会在编译时警告我们没有直接使用过workers、id及thread字段，这意味着我们还没有清理完所有的东西。当你使用不那么优雅的Ctrl+C方法使主线程停止运行时，所有的其他线程也会立即停止，即便它们正处于处理请求的过程中。

现在，我们将为线程池实现Drop trait来调用池中每个线程的join11方法，从而使它们能够在关闭前完成当前正在处理的工作。接着，我们还需要通过某种方式来避免线程接收新的请求并为停机做好准备。在接下来的实践中，修改后的服务器代码会在接收到两个请求后优雅地关闭线程池。

为ThreadPool实现Drop trait

让我们开始为线程池实现Drop。所有的线程都应当在线程池被丢弃时调用join，从而确保它们能够在结束前完成自己的工作。示例20-22中的代码是实现Drop的第一次尝试，这段代码还无法通过编译。

src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        ① for worker in &mut self.workers {
            ② println!("Shutting down worker {}", worker.id);

            ③ worker.thread.join().unwrap();
        }
    }
}
```

示例20-22：在线程池离开作用域时等待每个线程

上面的代码首先遍历了线程池中所有的workers①。这里使用了`&mut`，因为我们需要修改worker且正好self本身是一个可变引用。针对遍历中的每一个worker，代码会打印出信息来表明当前的worker正在停止运行②，并会接着在它的线程上调用join③。假如join调用失败，随后的unwrap就会触发panic并进入不那么优雅的关闭过程。

尝试编译代码会出现如下所示的错误：

```
error[E0507]: cannot move out of borrowed content
--> src/lib.rs:65:13
|
65 |         worker.thread.join().unwrap();
|         ^^^^^^ cannot move out of borrowed content
```

这个错误意味着我们不能调用join，因为当前的代码仅仅是可变借用了worker，而join方法则要求取得其参数的所有权。为了解决这一问题，我们需要把线程移出拥有其所有权的Worker实例，以便join可以消耗掉它。示例17-15曾经完成过类似的工作：如果Worker持有的是一个`Option<thread::JoinHandle<()>>`，那么我们就可以在Option上调用take方法来将Some变体的值移出来，并在原来的位置留下None变体。换句话说，正在运行中的Worker会在thread中持有一个Some变体，当我们希望清理Worker时，就可以使用None来替换掉Some，从而使Worker失去可以运行的线程。

更新后的Worker定义如下所示：

src/lib.rs

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

让我们再次依赖编译器来找出其他需要修改的地方。运行指令来检查代码，会显示出如下所示的两个错误：

```
error[E0599]: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>>` in the current scope
--> src/lib.rs:65:27
|
65 |         worker.thread.join().unwrap();
|         ^^^^
```



```
error[E0308]: mismatched types
--> src/lib.rs:89:13
```

```

89 |         thread,
|         ^^^^^^
|         |
|         expected enum `std::option::Option`, found struct
`std::thread::JoinHandle`
|             help: try using a variant of the expected type: `Some(thread)`
|
= note: expected type `std::option::Option<std::thread::JoinHandle<()>>`
        found type `std::thread::JoinHandle<_>`

```

先将注意力集中到第二个错误上，它指向了Worker::new结束部分的代码，新建Worker时需要将thread值包裹在Some中。接下来的修改可以修复这一问题：

src/lib.rs

```

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --略

        --
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

第一个错误出现在Drop实现中，因为我们还没有在Option值上调用take方法来把thread从worker中移出。如下所示的修改会修复这一问题：

src/lib.rs

```

impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            ① if let Some(thread) = worker.thread.take() {
                ② thread.join().unwrap();
            }
        }
    }
}

```

正如在第17章讨论的那样，为Option值调用take方法会将Some变体的值移出并在原来的位置留下None变体。上面的代码使用了if let来解构Some从而得到线程①，并接着在这个线程上调用了join②。当

某个Worker的线程值是None时，我们就知道worker已经清理了这个线程而无须进行任何操作。

通知线程停止监听任务

在完成了所有这些改进后，新的代码应该能够在没有任何警告的前提下成功通过编译了，但这段代码依旧没有实现我们想要的功能。问题的关键就在于工作线程运行的闭包逻辑：调用join并不会真正关停线程，因为它们还在loop循环中持续地等待任务。假如我们尝试使用当前的drop实现去丢弃ThreadPool，主线程就会被永远地阻塞住以等待第一个线程结束。

为了修复这一问题，我们需要修改线程中的逻辑。它需要在接收到Job时运行任务，并在接收到特定的结束信号时退出无限循环。新的代码将在通道中传递如下所示的两个枚举变体，而不再是Job实例：

src/lib.rs

```
enum Message {
    NewJob(Job),
    Terminate,
}
```

Message枚举的值要么是持有需要运行的Job的NewJob变体，要么是会导致线程退出循环并停止的Terminate变体。

与此同时，我们还需要修改通道来使用Message类型而不是Job类型，如示例20-23所示。

src/lib.rs

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    ● sender: mpsc::Sender<Message>,
}

// --略

--



impl ThreadPool {
    // --略
}
```

```

-->

pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static
{
    let job = Box::new(f);

    ❷ self.sender.send(Message::NewJob(job)).unwrap();
}

}

// --略

-->

impl Worker {
    ❸ fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>) ->
        Worker {
        let thread = thread::spawn(move || {
            loop {
                ❹ let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    ❺ Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);

                        ❻ job.();
                    },
                    ❼ Message::Terminate => {
                        println!("Worker {} was told to terminate.", id);
                    }
                    ❽ break;
                },
            }
        });
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

示例20-23：发送和接收Message值，并在工作线程收到Message::Terminate时退出循环

为了配合Message枚举，我们需要将ThreadPool的定义❶及Worker::new的签名❸中的Job修改为Message。ThreadPool的execute方法会发送包裹着任务的Message::NewJob变体❷。接着，Worker::new中的代码会从通道中接收并处理Message❹，在收到NewJob变体❺时处理任务❻，在收到Terminate变体❼时退出循环❽。

进行这些修改后，代码可以顺利地通过编译了，并可以使函数功能保持与示例20-20中的一样。但我们仍然会在编译过程中观察到一个新的警告消息，它提示我们还没有使用过Terminate变体。我们会修改Drop实现来解决这一问题，如示例20-24所示。

src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            ❶ self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                ❷ thread.join().unwrap();
            }
        }
    }
}
```

示例20-24：在对工作线程调用join之前向它们发送Message::Terminate

这段代码迭代了两次workers动态数组：第一次向每个worker发送了Terminate消息❶，第二次则在每个worker的线程上调用了join❷。如果我们尝试在同一个循环中发送消息并立即调用join，那么就无法保证当前正在迭代的worker就是从通道中获得消息的那个。

为了更好地理解为何需要两个循环，你可以想象一下拥有两个worker的场景。如果我们使用单个循环来迭代所有的worker，那么在进行首次迭代时，代码会将一个结束信息发送到通道中后接着在第一个worker线程上调用join。假设这个worker正好忙于处理其他请求，那么第二个worker就会从通道中获取这个结束信息并退出自己的循环。由于结束信号被第二个线程截取了，所以我们等待的第一个worker线程永远不会停止。一次死锁事件发生了！

为了阻止这种情况的发生，我们首先用一个循环把全部Terminate消息发送到通道中，随后再到另一个循环中等待所有的进程结束。由

于worker会在收到结束信号后停止接收请求，所以只要我们在调用join之前发送了与workers数目相等的结束消息，就可以确保每一个worker都能够收到自己的结束信号。

为了在实践中观察代码，让我们修改main函数来接收固定的两个请求并优雅地关闭服务器，如示例20-25所示。

src/bin/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20-25：处理两个请求后退出循环并关闭服务器

当然，这部分代码仅仅被用于确认停机与清理过程能够以正确的顺序执行，现实世界中的Web服务器可不会仅仅只处理两个请求就停止工作。

定义在Iterator trait中的take方法限制了我们的迭代过程最多只会进行两次。而ThreadPool则会在main函数结束时离开作用域，并调用自己的drop实现。

使用cargo run启动服务器，然后发起3个请求。第3个请求应该会出现错误，并会在终端输出如下所示的信息：

```
$ cargo run

Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
```

```
Shutting down all workers.  
Shutting down worker 0  
Worker 1 was told to terminate.  
Worker 2 was told to terminate.  
Worker 0 was told to terminate.  
Worker 3 was told to terminate.  
Shutting down worker 1  
Shutting down worker 2  
Shutting down worker 3
```

输出信息中的Worker序号与执行顺序也许会有所不同，但我们依然可以从这些信息中看出Worker是如何工作的：首先，编号为0和3的工作线程获得了前2个请求；接着，服务器在接收第3个请求前停止了接收连接；最后，ThreadPool在main函数的末尾离开作用域时调用了自己的Drop实现，它会通知池内的所有线程结束运行。工作线程会在收到结束信息后打印出一段信息，而线程池则会逐个调用join来使所有的工作线程停止运行。

注意，在这个特定的执行顺序中有一个有趣的地方：虽然ThreadPool向通道中发送了结束信息，但在工作线程接收到消息前，主线程就已经开始等待工作线程0的结束了。而因为此时的工作线程0还没有收到结束信号，所以主线程会被阻塞着直到工作线程0结束。当工作线程0结束时，主线程就会开始等待其他工作线程结束。由于其余线程全都在这个过程中收到了结束信号，所以服务器紧接着就顺利停机了。

恭喜！你终于完成了这个项目；我们现在拥有了一个基本的Web服务器，它使用线程池来异步地对请求做出响应。这个服务器可以优雅地停机，并会在停机前清理线程池中的工作线程。完整的参考代码可见No Starch出版社的官方网站。

实际上，这个项目还有许多可以改进的地方！如果你想要继续完善它，下面是一些可供你参考的方向：

- 为ThreadPool及其公共方法添加更多的文档。
- 为代码库的功能添加测试。
- 将unwrap调用修改为更健壮的错误处理方式。
- 使用ThreadPool完成其他一些不同于网页请求的任务。

- 在crates.io上寻找一个线程池包并使用它实现类似的Web服务器，接着再比较它们的API与鲁棒性之间的差别。

[1] 译者注：join 方法会阻塞当前的线程，并持续到它所属的线程执行完毕为止。

总结

干得不错！终于到了说再见的时候！由衷地感谢你同我们一道经历这趟Rust之旅。现在的你应该已经准备好实现自己的Rust项目并为他人提供帮助了。要始终记住的是，Rust拥有一个相当友善的社区，社区中其他的Rustacean总是乐于帮助你迎接Rust之路上出现的任何挑战。

附录A 关键字



下面的列表包含了Rust当前正在使用或将来可能会使用的关键字。也正是因为如此，它们通常不能被用作标识符出现在函数、变量、参数、结构体字段、模块、包、常量、宏、静态变量、属性、类型、trait或生命周期的名称中。稍后的“原始标识符”一节会讨论一种使用这些关键字作为标识符的特殊方法。

当前正在使用的关键字

下面的关键字目前可以完成对应描述中的功能。

- `as`: 执行基础类型转换，消除包含条目的指定trait的歧义，在`use`与`extern crate`语句中对条目进行重命名。
- `break`: 立即退出一个循环。
- `const`: 定义常量元素或不可变裸指针。
- `continue`: 继续下一次循环迭代。
- `crate`: 连接一个外部包或一个代表了当前包的宏变量。
- `dyn`: 表示trait对象可以进行动态分发。
- `else`: `if`和`if let`控制流结构的回退分支。
- `enum`: 定义一个枚举。
- `extern`: 连接外部包、函数或变量。
- `false`: 字面量布尔假。
- `fn`: 定义一个函数或函数指针类型。
- `for`: 在迭代器元素上进行迭代，实现一个trait，指定一个高阶生命周期。
- `if`: 基于条件表达式结果的分支。
- `impl`: 实现类型自有的功能或trait定义的功能。

- `in`: for循环语法的一部分。
- `let`: 绑定一个变量。
- `loop`: 无条件循环。
- `match`: 用模式匹配一个值。
- `mod`: 定义一个模块。
- `move`: 让一个闭包获得全部捕获变量的所有权。
- `mut`: 声明引用、裸指针或模式绑定的可变性。
- `pub`: 声明结构体字段、`impl`块或模块的公共性。
- `ref`: 通过引用绑定。
- `return`: 从函数中返回。
- `Self`: 指代正在其上实现trait的类型别名。
- `self`: 指代方法本身或当前模块。
- `static`: 全局变量或持续整个程序执行过程的生命周期。
- `struct`: 定义一个结构体。
- `super`: 当前模块的父模块。
- `trait`: 定义一个trait。
- `true`: 字面量布尔真。
- `type`: 定义一个类型别名或关联类型。
- `unsafe`: 声明不安全的代码、函数、trait或实现。
- `use`: 把符号引入作用域中。

- where: 声明一个用于约束类型的从句。
- while: 基于一个表达式结果的条件循环。

将来可能会使用的保留关键字

下面的关键字目前还没有任何功能，但它们被Rust保留下来以备将来使用。

- abstract
- async
- become
- box
- do
- final
- macro
- override
- priv
- try
- typeof
- unsized
- virtual
- yield

原始标识符

原始标识符（raw identifier）作为一种特殊的语法，允许我们使用那些通常不被允许使用的关键字作为标识符。这一语法需要为关键字添加前缀r#。

比如，match是一个关键字。假如你尝试编译下面这个以match作为名称的函数：

src/main.rs

```
fn match(needle: &str, haystack: &str) -> bool {  
    haystack.contains(needle)  
}
```

你将会得到如下所示的错误：

```
error: expected identifier, found keyword `match`  
--> src/main.rs:4:4  
|  
4 | fn match(needle: &str, haystack: &str) -> bool {  
|     ^^^^^^ expected identifier, found keyword
```

这个错误表明你不能将关键字match用作函数标识符。为了使用match作为函数名称，我们需要使用原始标识符语法，如下所示：

src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {  
    haystack.contains(needle)  
}  
  
fn main() {  
    assert!(r#match("foo", "foobar"));  
}
```

这段代码可以毫无问题地通过编译。注意，函数名称的前缀r#同样也出现在了调用这个函数的地方。

原始标识符允许我们使用任意的单词作为标识符，即便这个单词恰好是保留的关键字。另外，原始标识符也使我们能够调用基于不同Rust版本编写的外部库。例如，try在2018版本中作为新关键字被引入Rust。假设你所依赖的库基于2015版本并正好拥有一个名为try的函数，那么你就需要用到原始标识符语法，也就是r#try，以便在2018版本的代码中调用这个函数。你可以在附录E中找到有关版本差异的更多信息。

附录B 运算符和符号



本附录中给出了Rust语法的术语表，它们包括运算符与其他符号。这些符号要单独出现，要么出现在路径、泛型、trait约束、宏、属性、注释、元组或括号中。

运算符

表B-1包含了Rust中的所有运算符，每一行分别包含运算符本身、运算符出现在上下文中的示例、一个简短的说明及当前运算符是否可重载。如果运算符是可重载的，我们还会列出重载运算符所涉及的 trait。

表B-1 运算符

运算符	示例	说明	是否可重载
!	ident!(...), ident!{...}, ident![...]	宏展开	
!	!expr	按位或逻辑非	Not

续表

运算符	示例	说明	是否可重载
<code>!=</code>	<code>var != expr</code>	不相等比较	<code>PartialEq</code>
<code>%</code>	<code>expr % expr</code>	算术求余	<code>Rem</code>
<code>%=</code>	<code>var %= expr</code>	算术求余并赋值	<code>RemAssign</code>
<code>&</code>	<code>&expr, &mut expr</code>	借用	
<code>&</code>	<code>&type, &mut type, &'a type, &'a mut type</code>	借用指针类型	
<code>&</code>	<code>expr & expr</code>	按位与	<code>BitAnd</code>
<code>&=</code>	<code>var &= expr</code>	按位与并赋值	<code>BitAndAssign</code>
<code>&&</code>	<code>expr && expr</code>	逻辑与	
<code>*</code>	<code>expr * expr</code>	算术乘法	<code>Mul</code>
<code>*=</code>	<code>var *= expr</code>	算术乘法并赋值	<code>MulAssign</code>
<code>*</code>	<code>*expr</code>	解引用	
<code>*</code>	<code>*const type, *mut type</code>	裸指针	
<code>+</code>	<code>trait + trait, 'a + trait</code>	复合类型限制	
<code>+</code>	<code>expr + expr</code>	算术加法	<code>Add</code>
<code>+=</code>	<code>var += expr</code>	算术加法并赋值	<code>AddAssign</code>
<code>,</code>	<code>expr, expr</code>	参数和元素分隔符	
<code>-</code>	<code>- expr</code>	算术取负	<code>Neg</code>
<code>-</code>	<code>expr - expr</code>	算术减法	<code>Sub</code>
<code>--=</code>	<code>var -= expr</code>	算术减法并赋值	<code>SubAssign</code>
<code>-></code>	<code>fn(...) -> type, [...] -> type</code>	函数和闭包返回类型	
<code>.</code>	<code>expr.ident</code>	成员访问	
<code>...</code>	<code>..., expr..., ...expr, expr..expr</code>	左闭右开区间字面量	
<code>...=</code>	<code>...=expr, expr...=expr</code>	左闭右闭区间字面量	
<code>...</code>	<code>..expr</code>	结构体字面量更新语法	
<code>..</code>	<code>variant(x, ...), struct_type</code>	“余下所有”模式绑定	
<code>..</code>	<code>{ x, ... }</code>		
<code>...</code>	<code>expr...expr</code>	模式: 范围包含模式	
<code>/</code>	<code>expr / expr</code>	算术除法	<code>Div</code>
<code>/=</code>	<code>var /= expr</code>	算术除法并赋值	<code>DivAssign</code>
<code>:</code>	<code>pat: type, ident: type</code>	限制	
<code>:</code>	<code>ident: expr</code>	结构体字段初始化	
<code>:</code>	<code>'a: loop {...}</code>	循环标签	
<code>;</code>	<code>expr;</code>	语句或元素结束符	
<code>;</code>	<code>[...; len]</code>	固定大小数组语法的一部分	
<code><<</code>	<code>expr << expr</code>	左移	<code>Shl</code>
<code><<=</code>	<code>var <<= expr</code>	左移并赋值	<code>ShlAssign</code>
<code><</code>	<code>expr < expr</code>	小于比较	<code>PartialOrd</code>

续表

运算符	示例	说明	是否可重载
<code><=</code>	<code>expr <= expr</code>	小于等于比较	PartialOrd
<code>=</code>	<code>var = expr, ident = type</code>	赋值/等值	
<code>==</code>	<code>expr == expr</code>	相等性比较	PartialEq
<code>=></code>	<code>pat => expr</code>	匹配分支语法的一部分	
<code>></code>	<code>expr > expr</code>	大于比较	PartialOrd
<code>>=</code>	<code>expr >= expr</code>	大于等于比较	PartialOrd
<code>>></code>	<code>expr >> expr</code>	右移	Shr
<code>>>=</code>	<code>var >>= expr</code>	右移并赋值	ShrAssign
<code>@</code>	<code>ident @ pat</code>	模式绑定	
<code>^</code>	<code>expr ^ expr</code>	按位异或	BitXor
<code>^=</code>	<code>var ^= expr</code>	按位异或并赋值	BitXorAssign
<code> </code>	<code>pat pat</code>	模式或	
<code> </code>	<code>expr expr</code>	按位或	BitOr
<code> =</code>	<code>var = expr</code>	按位或并赋值	BitOrAssign
<code> </code>	<code>expr expr</code>	逻辑或	
<code>?</code>	<code>expr?</code>	错误传播	

非运算符符号

接下来的列表包含了所有非运算符符号，换句话说，这些符号的行为不同于函数或方法调用。

表B-2展示了可以独立出现的符号，以及它们在不同场景下合法出现时的样子。

表B-2 独立语法

符号	说明
'ident	命名生命周期或循环标签
...u8, ...i32, ...f64, ...usiz e, etc.	指定类型的数字字面量
"..."	字符串字面量
r"...", r#"..."#, r##"..."##, etc.	原始字符串字面量，其中的转义字符不会被处理
b"..."	字节字符串字面量；构建一个[u8]，而不是一个字符串
br"...", br#"..."#, br##"..."##, etc.	原始字节字符串字面量，由原始字符串字面量和字节字符串字面量组成
'...''	字符字面量
b'...''	ASCII 码字节字面量
[...] expr	闭包
!	离散函数中总是为空的返回类型
_	忽略模式绑定，也可用于增强整数字面量的可读性

表B-3展示了出现在路径上下文（从模块层级到具体条目）中的所有符号。

表B-3 与路径相关的语法

符号	说明
ident::ident	命名空间路径
::path	从根模块开始的相对路径（比如，一个显式的绝对路径）
self::path	从当前模块开始的相对路径（比如，一个显式的相对路径）
super::path	从当前模块的父节点开始的相对路径
type::ident, <type as trait>::ident	关联常数、函数和类型
<type>::...	不能直接被命名的类型的关联条目 (<&T>::..., <[T]>::... 等)
trait::method(...)	通过命名定义该方法的 trait 来消除方法调用的歧义
type::method(...)	通过命名定义该方法的类型来消除方法调用的歧义
<type as trait>::method(...)	通过命名定义该方法的 trait 和类型来消除方法调用的歧义

表B-4展示了出现在泛型参数上下文中的符号。

表B-4 泛型

符号	说明
path<...>	为类型中的泛型指定参数（比如，Vec<u8>）
path::<...>, method::<...>	为表达式中的泛型、函数或方法指定参数，这一语法常被称作 <i>turbofish</i> （比如，"42".parse::<i32>()）
fn ident<...> ...	定义泛型函数
struct ident<...> ...	定义泛型结构体
enum ident<...> ...	定义泛型枚举
impl<...> ...	定义泛型实现
for<...> type	高阶生命周期限定
type<ident=type>	为泛型中的一个或多个关联类型指定具体类型（比如，Iterator<Item=T>）

表B-5展示了使用 trait 约束来限制泛型参数时可能出现的符号。

表B-5 trait约束

符号	说明
T: U	将泛型 T 限制为实现了 U 的类型
T: 'a	将泛型 T 限制为生命周期长于'a 的类型（意味着这种类型不能传递性地包含生命周期短于'a 的引用）
T : 'static	泛型 T 不包含除'static 之外的借用引用
'b: 'a	泛型'b 的生命周期必须要长于'a
T: ?Sized	允许泛型参数是一个动态大小的类型
'a + trait, trait + trait	复合类型限制

表B-6展示了在调用宏、定义宏或在条目上指定属性时可能出现的符号。

表B-6 宏和属性

符号	解释
# [meta]	外部属性
#! [meta]	内部属性
\$ident	宏替代
\$ident:kind	宏捕获
\$ (...) ...	宏重复

表B-7展示了创建注释时可能出现的符号。

表B-7：注释

符号	说明
//	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展示了出现在元组上下文中的符号

表B-8 元组

符号	说明
()	空元组（也叫单元），它既是字面量也是类型
(expr)	圆括号表达式
(expr,)	单个元素元组表达式
(type,)	单个元素元组类型
(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 struct 及元组 enum 变体
ident!(...), ident!{...}, ident![...]	宏调用
expr.0, expr.1, etc.	元组索引

表B-9展示了使用花括号时的上下文。

表B-9 花括号

上下文	说明
{...}	块表达式
Type {...}	struct 字面量

表B-10展示了使用方括号时的上下文。

表B-10：方括号

上下文	说明
[...]	数组字面量
[expr; len]	包含 len 个 expr 的数组字面量
[type; len]	包含 len 个 type 的实例的数组类型
expr[expr]	集合索引，可重载 (Index, IndexMut)
expr[..], expr[a..], expr[..b], expr[a..b]	使用 Range、RangeFrom、RangeTo 或 RangeFull 进行集合切片索引

附录C 可派生trait



我们在本书中的许多地方都提到过derive属性，它可以被用在一个结构体或枚举定义上。当你在某个类型中声明derive属性时，它会为你在当前derive语法中声明的trait自动生成一份默认实现。

本附录会列举出标准库中所有可用于配合derive使用的trait作为参考。其中每一节都会涉及以下几个方面：

- 派生trait会重载哪些运算符或提供哪些方法。
- derive为trait提供了什么样的默认实现。
- trait的实现对目标类型意味着什么。
- 是否允许实现trait的相关条件。
- 使用这个trait的操作示例。

假如你需要的行为不同于derive属性的默认实现，那么你可以参考标准库文档中相关trait的细节来了解如何手动实现它们。

标准库中余下的那些trait无法通过derive来基于你的类型实现，它们通常都不存在有意义的默认行为。因此，你需要基于所处的具体环境来手动选择有意义的实现方式。

Display就是一个典型的不可派生trait，它被用来实现面向终端用户的文本格式化。你应该总是考虑为终端用户选择适当的方式来显示类型。类型中的哪些部分能够允许被终端用户看到？哪些部分可能会对终端用户起到作用？哪种格式对于终端用户最为友好？Rust编译器可没有这样的洞察力，它在这种场景下无法为你提供一个合适的默认行为。

本附录中并没有列出所有可以被派生的trait：代码库可以为它们自己的trait实现derive功能，这使得能够使用derive的trait实际上是无穷无尽的。实现derive会用到第19章的“宏”一节中介绍的过程宏。

面向程序员格式化输出的Debug

`Debug trait`被用于在格式化字符串中提供用于调试的格式，你可以在{}占位符中添加`:?来指定使用这一格式。`

`Debug trait`允许我们为了调试来打印出某个类型的实例，这也就意味着使用该类型的开发者可以在程序执行的某个特定时间点上查看实例。

例如，使用`assert_eq!` 宏时需要用到`Debug trait`。这个宏会在相等性检查失败时打印出参数中实例的值，从而使得程序员可以观察到实例不相等的具体原因。

用于相等性比较的PartialEq和Eq

PartialEq trait允许我们比较类型实例的相等性，并允许我们使用==与!=运算符。

派生的PartialEq实现了eq方法。当在某个结构体上派生PartialEq时，两个结构体实例在所有字段都相等时相等。换句话说，只要存在任意不相等的字段，两个实例都会被视作不相等。当在某个枚举上派生时，每个变体都只与自身相等，而和其余变体不相等。

例如，assert_eq!宏需要使用PartialEq trait来比较类型的两个实例是否相等。

Eq trait本身没有方法，它的作用在于表明被标记类型的每一个值都与自身相等。Eq trait只能被应用在同时实现了PartialEq的类型上，尽管并不是所有实现了PartialEq的类型都能够实现Eq。一个典型的例子就是浮点数类型：浮点数类型的实现规范里明确指出两个非数(not-a-number, Nan)值的实例是互不相等的。

例如，HashMap<K, V>中的键需要实现Eq trait，从而使得HashMap<K, V>可以判定两个键是否相同。

使用PartialOrd和Ord进行次序比较

PartialOrd trait允许我们对类型实例进行次序比较。任何实现了PartialOrd的类型都可以使用<、>、<=与>=运算符。PartialOrd trait只能被应用在同时实现了PartialEq的类型上。

派生的PartialOrd实现了一个返回Option<Ordering>作为结果的partial_cmp方法，它会在给定值无法分出次序时返回None。一个无法给定比较次序的例子就是浮点数中的非数，尽管浮点数中的大部分值都是可以比较的。使用一个浮点数和一个Nan浮点数调用partial_cmp会返回None。

当我们在结构体上派生PartialOrd时，为了比较两个实例的次序，PartialOrd会按照字段出现在结构体定义中的顺序逐个对比字段的值。而当我们在枚举上派生它时，变体在枚举中的排列次序决定了不同变体之间的大小关系，在枚举定义中，声明在前的变体要小于声明在后的变体。

例如，rand包中的gen_range方法需要用到PartialOrd trait，这个方法在指定低值和高值的区间内产生随机数。

Ord trait表明被标注类型的任意两个值都存在一个有效的次序。它所实现的方法cmp会返回一个Ordering而不是Option<Ordering>，因为总是存在一个有效的次序。Ord trait只能被应用在同时实现了PartialOrd和Eq（并且，要实现Eq，必须先实现PartialEq）的类型上。当我们在结构体或枚举上派生它时，cmp与PartialOrd的partial_cmp方法拥有相同的行为。

例如，BTreeSet<T>在存储值时会用到Ord trait，这一数据结构需要基于值的次序来存储数据。

使用Clone和Copy复制值

Clone trait允许我们显式地创建一个值的深度拷贝，这一过程可能包含执行任意的代码及复制堆数据。你可以查阅第4章的“变量和数据交互的方式：克隆”一节来获得更多关于Clone的信息。

当我们在完整类型上派生Clone时，它会实现相应的clone方法来依次克隆类型中的每一部分。这也意味着派生Clone需要类型中所有的字段或值都同样实现了Clone。

例如，当我们在切片上调用to_vec方法时就会用到Clone。因为切片本身并不拥有它内部类型实例的所有权，但to_vec返回的动态数组却拥有它的实例，所以执行to_vec会在每一个元素上调用clone。因此，存储在切片中的类型必须要实现Clone（才拥有to_vec方法）。

Copy trait允许我们通过复制存储在栈上的位数据来创建一个值的浅度拷贝，这一过程不会涉及其他的任意代码。你可以查阅第4章的“栈上数据的复制”一节来获得更多关于Copy的信息。

由于Copy trait没有定义任何可供程序员重载的方法，所以不会有额外的代码在这一过程中得到执行。这也就是说，所有的开发者都可以假设复制值会非常快。

你可以在所有内部元素都实现了Copy的类型上派生Copy。另外，Copy trait只能被应用在同样实现了Clone的类型上，因为实现了Copy的类型总是存在一个Clone的实现来执行与Copy相同的工作。

很少有地方会强制要求使用Copy trait。一个实现了Copy的类型是可以优化的，这意味着你不需要显式调用clone，从而使代码更加简捷。

每一个需要使用Copy的地方都可以使用Clone来代替完成，但代码可能会损失一些性能或需要在适当的位置调用clone。

用于将值映射到另外一个长度固定的值的Hash

Hash trait允许我们使用哈希函数将一个任意大小的类型实例映射至一个固定大小的值对应的实例。派生Hash会实现对应的hash方法，hash方法的派生实现会逐次对类型的每个部分求hash结果，并将这些结果组合起来作为最终映射值。这也就意味着，派生Hash类型的所有字段或值也必须同样实现了Hash。

例如，HashMap<K, V>为了有效地存储数据会要求自己的键实现Hash。

用于提供默认值的Default

Default trait允许我们为某个类型创建默认值。派生的Default实现了一个default函数，它会对类型的每个部分依次调用相应的default函数。这也就意味着，派生Default类型的所有字段或值也必须同样实现了Default。

Default::default函数常常被组合用于结构体更新语法中（第5章的“使用结构体更新语法根据其他实例创建新实例”一节中有详细介绍）。你可以自定义结构体中某一小部分字段的值，然后再使用.. Default::default()为剩余部分的字段提供默认值。

例如，Option<T>实例的unwrap_or_default方法需要用到Default trait。当Option<T>为None时，unwrap_or_default方法就会调用Option<T>中类型T的Default::default方法，并将这一方法的返回值作为自己的结果。

附录D

有用的开发工具



本附录会介绍一些由Rust项目提供的有用的开发工具，它们包括自动格式化、警告的快速修复、代码分析及IDE的集成。

使用rustfmt自动格式化代码

rustfmt工具会根据社区约定的风格来重新将你的代码格式化。许多协作完成的项目都会选择使用rustfmt来避免产生对于Rust代码风格的争论：所有人都使用统一的工具来将代码格式化。

你可以通过如下所示的命令来安装rustfmt：

```
$ rustup component add rustfmt
```

上面的命令会同时安装rustfmt与cargo-fmt，与Rust为你安装rustc与cargo时类似。你可以通过如下所示的命令来格式化任意的Cargo项目：

```
$ cargo fmt
```

运行这条命令会格式化当前包中的所有Rust代码。当然，这只会修改代码风格而不会导致代码语义产生变化。你可以在Rust官方网站阅读rustfmt的文档来获得更多信息。

使用rustfix修复代码

rustfix工具被包含在Rust安装包中，它可以自动地修复一些编译器警告。假如你编写过Rust代码，那么你应该见识过编译器警告了。例如，考虑如下所示的代码：

src/main.rs

```
fn do_something() {}

fn main() {
    for i in 0..100 {
        do_something();
    }
}
```

我们在上面的代码中试图调用do_something函数100次，但是我们实际上并没有在for循环体中用到变量i。Rust会给出如下所示的警告：

```
$ cargo build

Compiling myprogram v0.1.0 (file:///projects/myprogram)
warning: unused variable: `i`
--> src/main.rs:4:9
|
4 |     for i in 1..100 {
|         ^ help: consider using `_i` instead
|
= note: #[warn(unused_variables)] on by default

Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

这段警告建议使用_i作为替代名称：变量名前方的下划线会表明我们是故意不使用该变量的。我们可以通过执行cargo fix来调用rustfix工具自动地采用这一建议：

```
$ 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
```

再次观察 *src/main.rs*，我们会发现 cargo fix 确实改变了代码：

src/main.rs

```
fn do_something() {}

fn main() {
    for _i in 0..100 {
        do_something();
    }
}
```

for 循环中的变量被重命名为了 *_i*，警告不会再出现了。

你也同样可以使用 cargo fix 命令来将代码翻译为不同的 Rust 版本，有关版本的更多信息可以参见附录 E。

使用Clippy完成更多的代码分析

Clippy工具中包含了一系列的代码分析工具（lint），它被用来捕捉常见的错误并提升Rust代码质量。

你可以通过如下所示的命令安装Clippy：

```
$ rustup component add clippy
```

并通过如下所示的命令在任意Cargo项目中运行Clippy来进行代码分析：

```
$ cargo clippy
```

例如，假设你的程序中使用了一个与数学常量pi近似的值，如下所示：

src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("the area of the circle is {}", x * r * r);
}
```

在这个项目中执行cargo clippy会产生如下所示的错误：

```
error: approximate value of `f{32, 64}::consts::PI` found. Consider using it
directly
--> src/main.rs:2:13
  |
2 |     let x = 3.1415;
  |           ^^^^^^
  |
= note: #[deny(clippy::approx_constant)] on by default
= help: for further information visit
  https://rust-lang-nursery.github.io/rust-clippy/master/index.html#approx_constant
```

这个错误指出Rust中存在更为精确的常量定义，你可以通过替换常量来获得更为准确的代码执行结果。当你将代码修改为使用PI常量后就不会产生任何来自Clippy的错误和警告了：

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);  
}
```

你可以在Rust官方网站阅读Clippy的文档来获得更多信息。

使用Rust语言服务器来集成IDE

为了帮助开发者实现IDE集成，Rust项目提供了自己的Rust语言服务器（Rust Language Server, rls）。这一工具实现了语言服务器协议（Language Server Protocol），该协议作为一份通用规范被用于IDE与编程语言的相互通信。这也意味着rls可以被用于不同的客户端来完成IDE集成，比如用于Visual Studio Code的Rust插件（<https://marketplace.visualstudio.com/items?itemName=rust-lang.rust>）。

你可以通过如下所示的命令安装rls：

```
$ rustup component add rls
```

接着安装特定IDE中的语言服务器支持。随后你就可以获得诸如自动补全、定义跳转、内联错误提示等功能。

你可以在Rust官方网站上阅读rls的文档来获得更多信息。

附录E 版本



当你在第1章中使用 `cargo new` 创建项目时，你应该已经在 `Cargo.toml` 文件中见到过有关版本的元数据了。本附录会更加深入地讨论它所蕴含的意义！

Rust语言与编译器以6周作为一个发布循环，这意味着用户可以持续稳定地获得功能更新。某些编程语言选择在更长的时间周期后发布大规模修改，但Rust则选择了更为频繁地发布小规模更新。在一段时间后，所有这些小更新会日积月累地增多。随着版本的迭代，普通用户将会越来越难以回顾并发出类似于这样的感叹：“哇，Rust 1.10到Rust 1.31的变化可真大！”

每隔两到三年，Rust团队都会生成一个新的Rust版本。每个版本都会将当前已经落地至对应包中的功能集合到一起，这些功能都拥有完善的文档与工具。新版本会作为6周发布循环中的一部分被提交给用户。

版本对于不同的人群拥有不同的意义：

- 对于活跃的Rust用户而言，一个新的版本会将增加的修改引入易于理解的包中。
- 对于还没有开始使用Rust的用户而言，一个新的版本表明我们发布了某些重大的进展，此时的Rust可能值得一试。
- 对于Rust本身的开发者而言，一个新的版本提供了整个项目的里程碑。

在编写本书时，Rust已经提供了两个可用的版本：Rust 2015与Rust 2018。本书基于Rust 2018编写而成。

Cargo.toml 文件中的edition表明白代码应该使用哪个版本的编译器。当这个字段不存在时，Rust会出于向后兼容目的默认采用2015作为编译版本。

每个项目都可以自由地选择版本，而无须拘泥于默认的2015版本。版本与版本之间会包含一些不兼容的修改，比如引入一个会与当前标识符冲突的新关键字等。但不管怎样，除非你主动选择新的版本来兼容这些修改，否则你之前的代码都应当能够继续通过编译，即便你升级了系统中的Rust编译器版本。

所有的Rust编译器都会兼容之前存在的任意版本，并能够链接采用这些支持版本的包。版本之间产生的变化仅仅会影响到编译器最初解析代码时的过程。因此，即便你正在使用Rust 2015编写代码，也可以将一个使用Rust 2018的包作为依赖，项目不会在编译时出现任何问题。相反，当你使用Rust 2018编写代码时，你也可以依赖于Rust 2015的包。

需要注意的是，大部分功能在所有的版本中都是可用的。使用任何Rust版本的开发者都应该能够持续地接收稳定版本中的改进。但在某些情况下，主要是当某些新关键字被引入时，某些新功能将只会在较新的版本中可用。你需要切换到新的版本才能体验到这些功能。

请在Rust官方网站查阅 *Edition Guide* 来获得更多相关信息，它被专门用于介绍版本及版本之间的差异，并解释了如何利用cargo fix来自动地将你的代码升级至新的版本。

Broadview 博文视点·IT出版旗舰品牌
www.broadview.com.cn 技术凝聚实力·专业创新出版

Rust权威指南



《Rust权威指南》是一本介绍Rust语言的官方图书。作为开源的系统编程语言，Rust可以帮助你编写出更有效率且更加可靠的软件。Rust在给予开发者控制底层细节（比如内存操作）能力的同时，通过高水准的工程设计避免了传统底层语言中的诸多麻烦。

在《Rust权威指南》中，两位Rust核心团队成员将会向你演示如何利用Rust的各种特性——从安装过程到编写出健壮且可扩展的程序。你会从创建函数、选择数据类型及绑定变量等基础内容着手，然后深入学习更多高级主题，例如：

- 所有权、借用、生命周期及trait
- 使用Rust的内存安全保证来构建快速、安全的程序
- 测试、错误处理及高效的重构
- 泛型、智能指针、多线程、trait对象及高级模式匹配
- 使用Cargo（Rust内置的包管理器）来构建、测试、将代码生成文档及管理依赖
- 利用Rust先进的编译器来完成编译驱动编程

除了贯穿全书的众多示例代码，你还会看到3章项目实践方面的内容。这3章会通过构建3个特定的完整项目来测试你所学到的知识，这3个完整的项目包括一个猜数游戏、一个用Rust实现的命令行工具及一个多线程服务器。

关于作者

Steve Klabnik既是Rust的核心开发者，也是Rust文档团队的负责人。他时常参与各种演讲，并且是一位十分多产的开源贡献者。他曾经开发过的项目包括Ruby及Ruby on Rails。

Carol Nichols既是Rust核心团队中的成员，也是Integer 32公司的联合创始人，以及Rust Belt Rust（Rust铁锈地带）会议的组织者。



读者服务

- 微信扫码回复：38706
- 获取博文视点学院20元付费内容抵扣券
- 获取本书涉及的相关网站链接
- 获得更多技术专家分享的视频与学习资源
- 加入读者交流群，与更多读者互动



责任编辑：刘恩惠
封面设计：李玲



上架建议：编程语言/Rust

ISBN 978-7-121-38706-7



9 787121 387067

定价：159.00元