HZ BOOKS
华章科技

Rust 连续多年荣膺 Stack Overflow 网站最受程序员欢迎的编程语言。

深入浅出Rust

DIVE INTO RUST

范长春 ◎ 著

本书使用通俗易懂的语言，辅以大量的代码示例，高屋建瓴地总结阐释了Rust 的主要概念以及使用方法，并对背后的设计思路和原理做了深入浅出的剖析，全面深入地提炼了 Rust 的设计精华。

机械工业出版社
China Machine Press

深入浅出Rust

范长春　著

ISBN：978-7-111-60642-0

本书纸版由机械工业出版社于2018年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

前言
第一部分　基础知识
　第1章　与君初相见
　　1.1　版本和发布策略
　　1.2　安装开发环境
　　1.3　Hello　World
　　1.4　Prelude
　　1.5　Format格式详细说明
　第2章　变量和类型
　　2.1　变量声明
　　2.2　基本数据类型
　　2.3　复合数据类型
　第3章　语句和表达式
　　3.1　语句
　　3.2　表达式
　　3.3　if-else
　第4章　函数
　　4.1　简介
　　4.2　发散函数
　　4.3　main函数
　　4.4　const　fn
　　4.5　函数递归调用
　第5章　trait
　　5.1　成员方法
　　5.2　静态方法
　　5.3　扩展方法
　　5.4　完整函数调用语法
　　5.5　trait约束和继承
　　5.6　Derive
　　5.7　trait别名
　　5.8　标准库中常见的trait简介
　　5.9　总结
　第6章　数组和字符串
　　6.1　数组
　　6.2　字符串

第7章　模式解构
7.1　简介
7.2　match
7.3　if-let和while-let
7.4　函数和闭包参数做模式解构
7.5　总结
第8章　深入类型系统
8.1　代数类型系统
8.2　Never　Type
8.3　再谈Option类型
第9章　宏
9.1　简介macro
9.2　示范型宏
9.3　宏1.1
第二部分　内存安全
第10章　内存管理基础
10.1　堆和栈
10.2　段错误
10.3　内存安全
第11章　所有权和移动语义
11.1　什么是所有权
11.2　移动语义
11.3　复制语义
11.4　Box类型
11.5　Clone　VS.Copy
11.6　析构函数
第12章　借用和生命周期
12.1　生命周期
12.2　借用
12.3　借用规则
12.4　生命周期标记
12.5　省略生命周期标记
第13章　借用检查
13.1　编译错误示例
13.2　内存不安全示例：修改枚举
13.3　内存不安全示例：迭代器失效
13.4　内存不安全示例：悬空指针

13.5 小结
第14章 NLL（Non-Lexical-Lifetime）
14.1 NLL希望解决的问题
14.2 NLL的原理
14.3 小结
第15章 内部可变性
15.1 Cell
15.2 RefCell
15.3 UnsafeCell
第16章 解引用
16.1 自定义解引用
16.2 自动解引用
16.3 自动解引用的用处
16.4 有时候需要手动处理
16.5 智能指针
16.6 小结
第17章 泄漏
17.1 内存泄漏
17.2 内存泄漏属于内存安全
17.3 析构函数泄漏
第18章 Panic
18.1 什么是panic
18.2 Panic实现机制
18.3 Panic Safety
18.4 小结
第19章 Unsafe
19.1 unsafe关键字
19.2 裸指针
19.3 内置函数
19.4 分割借用
19.5 协变
19.6 未定义行为
19.7 小结
第20章 Vec源码分析
20.1 内存申请
20.2 内存扩容
20.3 内存释放

20.4 不安全的边界
20.5 自定义解引用
20.6 迭代器
20.7 panic safety
第三部分 高级抽象
第21章 泛型
21.1 数据结构中的泛型
21.2 函数中的泛型
21.3 impl块中的泛型
21.4 泛型参数约束
21.5 关联类型
21.6 何时使用关联类型
21.7 泛型特化
第22章 闭包
22.1 变量捕获
22.2 move关键字
22.3 Fn/FnMut/FnOnce
22.4 闭包与泛型
22.5 闭包与生命周期
第23章 动态分派和静态分派
23.1 trait object
23.2 object safe
23.3 impl trait
第24章 容器与迭代器
24.1 容器
24.2 迭代器
第25章 生成器
25.1 简介
25.2 对比迭代器
25.3 对比立即求值
25.4 生成器的原理
25.5 协程简介
第26章 标准库简介
26.1 类型转换
26.2 运算符重载
26.3 I/O
26.4 Any

第四部分　线程安全
第27章　线程安全
27.1　什么是线程
27.2　启动线程
27.3　免数据竞争
27.4　Send　&　Sync
第28章　详解Send和Sync
28.1　什么是Send
28.2　什么是Sync
28.3　自动推理
28.4　小结
第29章　状态共享
29.1　Arc
29.2　Mutex
29.3　RwLock
29.4　Atomic
29.5　死锁
29.6　Barrier
29.7　Condvar
29.8　全局变量
29.9　线程局部存储
29.10　总结
第30章　管道
30.1　异步管道
30.2　同步管道
第31章　第三方并行开发库
31.1　threadpool
31.2　scoped-threadpool
31.3　parking_lot
31.4　crossbeam
31.5　rayon
第五部分　实用设施
第32章　项目和模块
32.1　cargo
32.2　项目依赖
32.3　模块管理
第33章　错误处理

33.1 基本错误处理
33.2 组合错误类型
33.3 问号运算符
33.4 main函数中使用问号运算符
33.5 新的Failure库
第34章 FFI
34.1 什么是FFI
34.2 从C调用Rust库
34.3 从Rust调用C库
34.4 更复杂的数据类型
第35章 文档和测试
35.1 文档
35.2 测试
附录 词汇表

前言

A language that doesn’t affect the way you think about programming is not worth knowing.

——Alan Perlis

Rust简介

Rust是一门新的编程语言。

我想，大部分读者看到本书，估计都会不约而同地想到同样的问题：现存的编程语言已经多得数不清了，再发明一种新的编程语言有何意义？难道现存的那么多编程语言还不够用吗，发明一种新的编程语言能解决什么新问题？

俗话说，工欲善其事，必先利其器。在程序员平时最常用的工具排行榜中，编程语言当仁不让的是最重要的“器”。编程语言不仅是给程序设计者使用的工具，反过来，它也深刻地影响了设计者本身的思维方式和开发习惯。

卓越的编程语言，可以将优秀的设计、先进的思想、成功的经验，自然而然地融入其中，使更多的使用者开阔眼界、拓展思路，受益无穷。

A programming language is a tool that has profound influence on our thinking habits.

——Edsger Dijkstra

所以说关于这个问题，我认为，如果与现有的各种语言相比，新设计的语言有所进步、有所发展、有所创新，那么它的出现就很有意义。

最近这些年，的确涌现出了一大批编程语言，可以说是百花争艳、繁华似锦。但是在表面的繁荣之下，我们是否可以自满地说，编程语言的设计和发展已经基本成熟、趋于完美了呢？恐怕不尽然吧！

那些优秀的编程语言中，不少都有自己的“绝活”。有的性能非常高，有的表达力非常强，有的擅长组织大型程序，有的适合小巧的脚本，有的专注于并发，有的偏重于科学计算，等等，不一而足。即便如此，新兴的Rust语言面市后依旧展现出了它的独特魅力，矫矫不群，非常值得大家关注。

作为多年来鲜有的新一代系统编程语言，它的设计准则是“安全，并发，实用”。Rust的设计者是这样定位这门语言的：

Rust is a system’s programming language that runs blazingly fast，prevents segfaults，and guarantees thread safety.

安全

是的，安全性很重要。Rust最重要的特点就是可以提供内存安全保证，而且没有额外的性能损失。

在传统的系统级编程语言（C/C++）的开发过程中，经常出现因各种内存错误引起的崩溃或bug。比如空指针、野指针、内存泄漏、内存越界、段错误、数据竞争、迭代器失效等，血泪斑斑，数不胜数。这些问题不仅在教科书中被无数次提起，而且在实践中也极其常见。因此，各种高手辛苦地总结了大量的编程经验，许多代码检查和调试工具被开发出来，各种代码开发流程和规范被制定出来，无数人呕心沥血就是为了系统性地防止各类bug的出现。尽管如此，我们依然无法彻底解决这些问题。

教科书解决不了问题，因为教育不是强制性的；静态代码检查工具解决不了问题，因为传统的C/C++对静态代码检查不友好，永远只能查出一部分问题；软件工程解决不了问题，因为规范依赖于执行者的素质，任何人都会犯错误。事后debug也不是办法，解决bug的代价更高。

鉴于手动内存管理非常容易出问题，因此先辈们发明了一种自动垃圾回收的机制（Garbage Collection），故而程序员在绝大多数情况下不用再操心内存释放的问题。新发明的绝大多数编程语言都使用了基于各种高级算法的自动垃圾回收机制，因为它确实方便，解放了程序员的大脑，使大家能更专注于业务逻辑的部分。但是到目前为止，不管使用哪种算法的GC系统，在性能上都要付出比较大的代价。要么需要较大的运行时占用较大内存，要么需要暂停整个程序，要么具备不确定性的时

延。当然，在现实的许多业务场景中，这点开销是微不足道的，因此问题不大。可是如果在性能敏感的领域，这是完全不可接受的。

很遗憾，到目前为止，在系统级编程语言中，我们依然被各种内存安全问题所困扰。这些年来，许多新的语言特性被发明出来，许多优秀的编程范式被总结出来，许多高质量的代码库被开发出来。但是内存安全问题依然像一个幽灵一样，一直徘徊在众多程序员的头顶，无法摆脱。再多的努力，也只能减少它出现的机会，很难保证完整地解决掉这一类错误。

Rust对自己的定位是接近芯片硬件的系统级编程语言，因此，它不可能选择使用自动垃圾回收的机制来解决问题。事实证明，要想解决内存安全问题，小修小补是不够的，必须搞清楚导致内存错误的根本原因，从源头上解决。Rust就是为此而生的。Rust语言是可以保证内存安全的系统级编程语言。这是它的独特的优势。本书将用大量的篇幅详细介绍“内存安全”。

并发

在计算机单核性能越来越接近瓶颈的今天，多核并行成了提高软件执行效率的发展趋势。一些编程语言已经开始从语言层面支持并发编程，把“并发”的概念植入到了编程语言的血液中。然而，在传统的系统级编程语言中，并行代码很容易出错，而且有些问题很难复现，难以发现和解决问题，debug的成本非常高。线程安全问题一直以来都是非常令人头痛的问题。

Rust当然也不会在这一重要领域落伍，它也非常好地支持了并发编程。更重要的是，在强大的内存安全特性的支持下，Rust一举解决了并发条件下的数据竞争（Data Race）问题。它从编译阶段就将数据竞争解决在了萌芽状态，保障了线程安全。

Rust在并发方面还具有相当不错的可扩展性。所有跟线程安全相关的特性，都不是在编译器中写死的。用户可以用库的形式实现各种高效且安全的并发编程模型，进而充分利用多核时代的硬件性能。

实用

Rust并不只是实验室中的研究型产品，它的目标是解决目前软件行

业中实实在在的各种问题。它的实用性体现在方方面面。

Rust编译器的后端是基于著名的LLVM完成机器码生成和优化的，它只需要一个非常小巧的运行时即可工作，执行效率上可与C语言相媲美，具备很好的跨平台特性。

Rust摈弃了手动内存管理带来的各种不安全的弊端，同时也避免了自动垃圾回收带来的效率损失和不可控性。在绝大部分情况下，保持了“无额外性能损失”的抽象能力。

Rust具备比较强大的类型系统，借鉴了许多现代编程语言的历史经验，包含了众多方便的语法特性。其中包括代数类型系统、模式匹配、闭包、生成器、类型推断、泛型、与C库ABI兼容、宏、模块管理机制、内置开源库发布和管理机制、支持多种编程范式等。它吸收了许多其他语言中优秀的抽象能力，海纳百川，兼容并蓄。在不影响安全和效率的情况下，拥有不俗的抽象表达力。

有意思的地方是，在程序语言设计领域，按照传统思路，有些设计目标是互相冲突的。而Rust的优异之处在于，它能游刃有余地游走在各种设计目标之间，扬长避短，保持良好的妥协和平衡。

本书结构

本书将详细描述Rust语言的基本语法，穿插讲解一部分高级使用技巧，并尽量以更容易理解的方式向读者解释其背后的设计思想。语法只是基础，并非本书的重点，笔者更希望读者能理解到这些语法设计背后的理念，读完本书之后，可以从中受到一点启发，对程序语言有更多的认识，从而对编程本身有更深的理解。

Learning a language that is significantly different than you are used to is certainly tough at first，but it’s a great way to expand your horizons a bit.

在本书中，笔者尽量避免要求读者有很多的基础知识。当然，如果读者对其他的一种或多种编程语言有所了解更佳，其中包括C/C++的基础知识、内存错误、手动内存管理、自动垃圾回收、多线程并发和同步、操作系统相关的基础概念等。

本书共分为五个部分。

第一部分介绍Rust基本语法。因为对任何程序设计语言来说，语法都是基础，学习这部分是理解其他部分的前提。

第二部分介绍属于Rust独一无二的内存管理方式。它设计了一组全新的机制，既保证了安全性，又保持了强大的内存布局控制力，而且没有额外性能损失。这部分是本书的重点和核心所在，是Rust语言的思想内核精髓之处。

第三部分介绍Rust的抽象表达能力。它支持多种编程范式，以及较为强大的抽象表达能力。

第四部分介绍并发模型。在目前这个阶段，对并行编程的支持是新一代编程语言无法绕过的重要话题。Rust也吸收了业界最新的发展成果，对并发有良好支持。

第五部分介绍一些实用设施。Rust语言有许多创新，但它绝不是高高在上、孤芳自赏的类型。设计者们在设计过程中充分考虑了语言的工程实用性。众多在其他语言中被证明过的优秀实践被吸收了进来，有利于提升实际工作效率。

为了内容的完整性，本书并没有严格按照知识点顺序组织内容，少数地方会直接使用后续章节中的知识点。笔者相信对读者来说，这不是一个很大的障碍，各位读者在碰到这种情况的时候，可以自行前后参照来理解。

总结和勘误

在计算机程序设计语言的领域中，一代又一代的语言潮起潮落，其兴起和衰落的节奏往往并非取决于技术本身的发展。对于Rust这门新出现的语言来说，以后究竟会有多大的影响，是否会成为取代某种语言的“新时代的宠儿”，实在难以预测，而且毫无必要预测。

笔者认为，Rust语言是最近若干年内系统级编程语言领域的集大成者之一。不论其最终发展如何，它的许多设计思想和令人惊叹的特性都值得大家学习。在本人的学习过程中，也时常为某些精彩的设计发出由

衷的赞叹。

Rust语言是一门优秀的语言，同时也是门槛较高的一门语言，要完全掌握它不是一件很容易的事。因此，笔者并不希望将本书写成语言特性的逐一简单罗列，而更希望向读者解释清楚这些语言特性背后的设计思想。

所幸的是，Rust语言是完全开源的，不仅代码是开源的，而且整个设计过程、思辩讨论都是对社区完全开放的。它的许多非常有价值的学习资料，如同星星点点，散落在各个地方，包括官方文档、邮件列表、讨论组、GitHub、个人博客等。在学习和写作的过程中，能有幸一窥新语言创造者们的心路历程，也是难得的机缘。

要想把Rust语言的方方面面讲好、讲透，实在是一个无比繁重的任务。动笔之际，方知“看人挑担不吃力，自家挑担压断脊”，诚惶诚恐，战战兢兢，生怕有误人子弟之嫌。笔者水平有限，如有错漏，在所难免，欢迎读者批评指正。笔者将会在GitHub上发布最新的勘误列表，网址为https://github.com/F001/rust_book_feedback 。读者可以在这个项目中新建bug提交问题，也可以通过邮件（rust-lang@qq.com）与笔者联系。同时也欢迎读者关注微信公众号：Rust编程，后面还会发布更多关于Rust的文章。

致谢

感谢Rust设计组，为软件开发行业创造了一份宝贵的财富。

感谢我所在的公司synopsys给予的大力支持。

感谢梁自泽导师对我的培养。

感谢林春晓博士拨冗为本书做了最后一轮审校。

感谢妻子的包容和呵护，否则本书不可能面世。

感谢杨绣国编辑细致的工作。

感谢各位老师、同学和同事对我的支持，正是因为你们的帮助，才

使我技术水平更上一层楼。

范长春（F001）

中国，武汉，2018年3月

第一部分　基础知识

在这一部分中，我们将对Rust语言的主要语法特性做一个循序渐进的介绍。Rust语言的基本语法特性并不复杂，它也并没有贪多求全地堆砌大量华而不实的语法特性。相反，它在吸收各种优秀语法规则的同时也做了裁剪，去芜存菁，张弛有度。

第1章　与君初相见

Rust编程语言的官方网站是https://www.rust-lang.org/ 。在官网主页上，我们可以看到，在最显眼的位置，写着Rust语言最重要的特点：

Rust is a systems programming language that runs blazingly fast，prevents segfaults，and guarantees thread safety.

Rust语言是一门系统编程语言，它有三大特点：运行快 、防止段错误 、保证线程安全 。

系统级编程是相对于应用级编程而言。一般来说，系统级编程意味着更底层的位置，它更接近于硬件层次，并为上层的应用软件提供支持。系统级编程语言一般具有以下特点：

·可以在资源非常受限的环境下执行；

·运行时开销很小，非常高效；

·很小的运行库，甚至于没有；

·可以允许直接的内存操作。

目前，C和C++应该是业界最流行的系统编程语言。Rust的定位与它们类似，但是增加了安全性。C和C++都是编译型语言，无须规模庞大的运行时（runtime）支持，也没有自动内存回收（Garbage Collection）机制。

本章主要对Rust做一个简单的介绍，准备好一些基本概念以及开发环境。

1.1 版本和发布策略

Rust编程语言是开源的，编译器的源码位于https://github.com/rust-lang/rust 项目中，语言设计和相关讨论位于https://github.com/rust-lang/rfcs 项目中。对于想深入研究这门语言的读者来说，这是一个非常好的消息，大家可以通过研读开放的源代码和技术文档了解到很多书本上没有讲解过的知识。任何一个开发者都可以直接给这个项目提bug，或者直接贡献代码。Rust项目是完全由开源社区管理和驱动的，社区的氛围非常友好。

Rust编译器的版本号采用了“语义化版本号”（Semantic Versioning）规划。在这个规则之下，版本格式为：主版本号.次版本号.修订号。版本号递增规则如下。

·主版本号：当你做了不兼容的API修改

·次版本号：当你做了向下兼容的功能性新增

·修订号：当你做了向下兼容的问题修正

Rust的第一个正式版本号是1.0，是2015年5月发布的。从那以后，只要版本没有出现大规模的不兼容的升级，大版本号就一直维持在“1”，而次版本号会逐步升级。Rust一般以6个星期更新一个正式版本的速度进行迭代。

为了兼顾更新速度以及稳定性，Rust使用了多渠道发布的策略：

·nightly版本

·beta版本

·stable版本

nightly版本是每天在主版本上自动创建出来的版本，这个版本上的功能最多，更新最快，但是某些功能存在问题的可能性也更大。因为新功能会首先在这个版本上开启，供用户试用。beta版本是每隔一段时

间，将一些在nightly版本中验证过的功能开放给用户使用。它可以被看作stable版本的“预发布”版本。而stable版本则是正式版，它每隔6个星期发布一个新版本，一些实验性质的新功能在此版本上无法使用。它也是最稳定、最可靠的版本。stable版本是保证向前兼容的。

在nightly版本中使用试验性质的功能，必须手动开启feature gate。也就是说要在当前项目的入口文件中加入一条#！[feature（…name…）]语句。否则是编译不过的。等到这个功能最终被稳定了，再用新版编译器编译的时候，它会警告你这个feature gate现在是多余的了，可以去掉了。

Rust语言相对重大的设计，必须经过RFC（Request For Comments）设计步骤。这个步骤主要是用于讨论如何“设计”语言。这个项目存在于 https://github.com/rust-lang/rfcs 。所有大功能必须先写好设计文档，讲清楚设计的目标、实现方式、优缺点等，让整个社区参与讨论，然后由“核心组”（Core Team）的成员参与定夺是否接受这个设计。笔者强烈建议各位读者多读一下RFC文档，许多深层次的设计思想问题可以在这个项目中找到答案。在Rust社区，我们不仅可以看到最终的设计结果，还能看到每一步设计的过程，对我们来说非常有教育意义。

Rust语言每个相对复杂一点的新功能，都要经历如下步骤才算真正稳定可用：

RFC → Nightly → Beta → Stable

先编写一份RFC，其中包括这个功能的目的、详细设计方案、优缺点探讨等。如果这个RFC被接受了，下一步就是在编译器中实现这个功能，在nightly版本中开启。经过几个星期甚至几个月的试用之后，根据反馈结果来决定撤销、修改或者接受这个功能。如果表现不错，它就会进入beta版本，继续过几个星期后，如果确实没发现什么问题，最终会进入stable版本。至此，这个功能才会被官方正式定为“稳定的”功能，在后续版本中要确保兼容性的。

这个发布策略非常成功，它保证了新功能可以持续、快速地进入到编译器中。在这个发布策略的支持下，Rust语言以及编译器的进化速度非常了不起，成功实践了快速迭代、敏捷交付以及重视用户反馈的特点，同时也保证了核心设计的稳定性——用户可以根据自己的需要和风险偏好，选择合适的版本。本书假定读者安装的是nightly版本，因为我

们的目标是学习，目前有许多重要的功能只存在于nightly版本。

在2017年下半年，Rust设计组又提出了一个基于epoch的演进策略（后来也被称为edition）。它要解决的问题是，如何让Rust更平稳地进化。比如，有时某些新功能确实需要一定程度上破坏兼容性。为了最大化地减少这些变动给用户带来的影响，Rust设计组又设计了一个所谓的edition的方案。简单来说就是让Rust的兼容性保证是一个有时限的长度，而不是永久。Rust设计组很可能会在不久的将来发布一个2018 edition，把之前的版本叫作2015 edition。在这个版本的进化过程中，就可以实施一些不兼容的改变。当然了，Rust设计组不会突然让前一个edition的代码到了后一个edition就不能编译了。他们采用了一种平滑过渡的方案。

我们举个例子。假设我们要添加一个功能，比如增加一个关键字。这件事情肯定是不兼容的改变，因为用户写的代码中很可能包含用这个关键字命名的变量、函数、类型等，直接把这个单词改成关键字会直接导致这些遗留代码出现编译错误。那怎么办呢？首先会在下一个edition中做出警告，提示用户这个单词已经不适合作为变量名了，请用户修改。但是这个阶段代码依然能编译通过。然后到再下一个edition的时候，这个警告就会变成真正的编译错误，此时这个关键字就可以真正启用了。先编译警告，再编译错误，这个过程可能会持续好几年，所以Rust的稳定性还是基本上有保证的。毕竟，如果要维持百分之百的兼容性，Rust语言就很难再继续进化了。如果让极少一部分受影响的遗留代码，完全锁死整个语言的进步空间，对于那些特别需要某些新功能的用户来说也是不公平的。通过这个缓慢过渡的策略，基本可以让所有Rust的使用者平滑、无痛地过渡到新版本。几年的过渡时间也是足够充分的。

Rust的标准库文档位于https://doc.rust-lang.org/std/ 。学会查阅标准库文档，是每个Rust使用者的必备技能之一。

1.2 安装开发环境

Rust编译器的下载和安装方法在官网上有文档说明，点击官网上的Install链接可以查看。Rust官方已经提供了预编译好的编译器供我们下载，支持Windows平台、Linux平台以及Mac平台。但是一般我们不单独下载Rust的编译器，而是使用一个叫rustup的工具安装Rust相关的一整套工具链，包括编译器、标准库、cargo等。使用这个工具，我们还可以轻易地更新版本、切换渠道、多工具链管理等。

在官网上下载rustup-init程序，打开命令行工具，执行这个程序，按照提示选择合适的选项即可。不论在Windows、Linux还是Mac操作系统上，安装步骤都是差不多的。

在Windows平台下的选项要稍微麻烦一点。在Windows平台上，Rust支持两种形式的ABI（Application Binary Interface），一种是原生的MSVC版本，另一种是GNU版本。如果你需要跟MSVC生成的库打交道，就选择MSVC版本；如果你需要跟MinGW生成的库打交道，就选择GNU版本。一般情况下，我们选择MSVC版本。在这种情况下，Rust编译器还需要依赖MSVC提供的链接器，因此还需要下载VisualC++的工具链。到Visual Studio官网下载VS2015或者VS2017社区版，安装C++开发工具即可。

安装完成之后，在$HOME/.cargo/bin文件夹下可以看到一系列的可执行程序，比如Rust 1.19版本的时候，在Windows平台上安装的程序如图1-1所示。

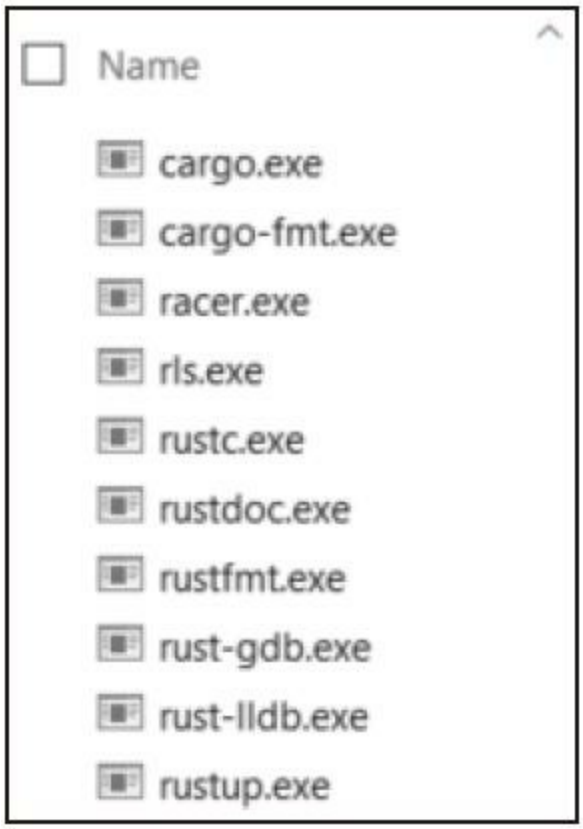


图　1-1

其中，rustc.exe是编译器，cargo.exe是包管理器，cargo-fmt.exe和rustfmt.exe是源代码格式化工具，rust-gdb.exe和rust-lldb.exe是调试器，rustdoc.exe是文档生成器，rls.exe和racer.exe是为编辑器准备的代码提示工具，rustup.exe是管理这套工具链下载更新的工具。

我们可以使用rustup工具管理工具链。

```
// 更新rustup本身
$ rustup self update
// 卸载rust所有程序
$ rustup self uninstall
// 更新工具链
$ rustup update
```

我们还可以使用它轻松地在stable/beta/nightly渠道中切换，比如：

```
// 安装nightly版本的编译工具链
$ rustup install nightly
// 设置默认工具链是nightly版本
```

```
$ rustup default nightly
```

为了提高访问速度，中国科技大学Linux用户协会（USTC LUG）提供了一个代理服务，官方网址为https://lug.ustc.edu.cn/wiki/mirrors/help/rust-static ，建议国内用户设置好以下环境变量再使用rustup：

```
export RUSTUP_DIST_SERVER=https://mirrors.ustc.edu.cn/rust-static
export RUSTUP_UPDATE_ROOT=https://mirrors.ustc.edu.cn/rust-static/rustup
```

Rust官方工具链还提供了重要的包管理工具cargo.exe，我们可以通过这个工具轻松导入或者发布开源库。官方的管理仓库在https://crates.io/ ，大家可以登录这个网站浏览一下Rust社区热门的开源库都有哪些。大型项目往往需要依赖这些开源库，cargo会帮我们自动下载编译。同样，为了解决网络问题，需要利用USTC提供的代理服务，使用方式为：在$HOME/.cargo目录下创建一个名为config的文本文件，其内容为：

```
[source.crates-io]
registry = "https://github.com/rust-lang/crates.io-index"
replace-with = 'ustc'
[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

这样，在编译需要依赖crates.io的项目时，不会由于网络问题导致依赖库下载失败。

RLS（Rust Language Server）是官方提供的一个标准化的编辑器增强工具。它也是开源的，项目地址在https://github.com/rust-lang-nursery/rls 。它是一个单独的进程，通过进程间通信给编辑器或者集成开发环境提供一些信息，实现比较复杂的功能，比如代码自动提示、跳转到定义、显示函数签名等。安装最新的RLS的方法为：

```
// 更新rustup到最新
rustup self update
// 更新rust编译器到最新的nightly版本
rustup update nightly
// 安装RLS
rustup component add rls --toolchain nightly
rustup component add rust-analysis --toolchain nightly
```

```
rustup component add rust-src --toolchain nightly
```

有了这些准备，大家就可以在Visual Studio Code中下载支持Rust的插件，提升编辑体验。理论上来说，RLS可以跟任何编辑器或者集成开发环境配合使用，只要这个编辑器实现了它们之间的通信协议即可。

有了上面这些准备工作，我们就可以正式开始Rust编程之旅了。首先，打开命令行工具，看看rustc编译器能否正常运行，使用-V命令查看rustc的版本：

```
$ rustc -V
rustc 1.20.0-nightly (f85579d4a 2017-07-12)
```

如果看到类似的输出，说明编译器已经可以正常工作。接下来，请大家探索一下这些工具的简明使用帮助：

1）使用rustc-h命令查看rustc的基本用法；

2）使用cargo-h命令查看cargo的基本用法；

3）使用rustc-C help命令查看rustc的一些跟代码生成相关的选项；

4）使用rustc-W help命令查看rustc的一些跟代码警告相关的选项；

5）使用rustc-Z help命令查看rustc的一些跟编译器内部实现相关的选项；

6）使用rustc–help-V命令查看rustc的更详细的选项说明。

1.3 Hello World

编程语言入门第一课，必须得是hello world程序。我们先来看看Rust的hello world是什么样子：

```
// hello_world.rs
fn main() {
    let s = "hello world!";
    println!("{}", s);
}
```

对于这样一个简单的示例程序，我们并没有使用cargo创建工程，因为没有复杂的依赖关系。编译就直接使用rustc即可，其他所有选项使用默认值：

```
rustc hello_world.rs
```

可看到本地文件夹中生成了一个名为hello_world的可执行程序。执行./hello_world程序，可以看见控制台上输出了hello world！字符串。恭喜读者，第一个Rust程序已经运行成功了！

我们来分析一下这个最简单的程序。

1）一般Rust源代码的后缀名使用.rs表示。源码一定要注意使用utf-8编码。

2）第一行是注释语句，Rust的注释是C语言系列风格的，行注释采用//开头，块注释使用/*和*/包围。它还支持更高级的文档注释，将在后文中详细展开说明。

3）fn是一个关键字（key word），函数定义必须以这个关键字开头。函数体使用大括号来包含。fn是单词function的缩写，在Rust中，设计者比较偏向使用单词缩写，即使是关键字也不例外。在代码风格上，某些读者可能开始会有点不习惯。但总体而言，这只是个审美偏好而已，不必过于纠结，习惯就好。

4）默认情况下，main函数是可执行程序的入口点，它是一个无参数，无返回值的函数。如果我们要定义的函数有参数和返回值，可以使用以下语法（参数列表使用逗号分开，冒号后面是类型，返回值类型使用->符号分隔）：

```
fn Foo( input1 : i32, input2 : u32) -> i32 {
    ...
}
```

5）局部变量声明使用let关键字开头，用双引号包含起来的部分是字符串常量。Rust是静态强类型语言，所有的变量都有严格的编译期语法检查。关于Rust的变量和类型系统将在后文详细说明。

6）每条语句使用分号结尾。语句块使用大括号。空格、换行和缩进不是语法规则的一部分。这都是明显的C语言系列的风格。

最简单的标准输出是使用println！宏来完成。请大家一定注意println后面的感叹号，它代表这是一个宏，而不是一个函数。Rust中的宏与C/C++中的宏是完全不一样的东西。简单点说，可以把它理解为一种安全版的编译期语法扩展。这里之所以使用宏，而不是函数，是因为标准输出宏可以完成编译期格式检查，更加安全。

从这个小程序的惊鸿一瞥中，大家可以看到，Rust的语法主要还是C系列的语法风格。对于熟悉C/C++/Java/C#/PHP/JavaScript等语言的读者来说，会看到许多熟悉的身影。

1.4 Prelude

Rust的代码从逻辑上是分crate和mod管理的。所谓crate大家可以理解为“项目”。每个crate是一个完整的编译单元，它可以生成为一个lib或者exe可执行文件。而在crate内部，则是由mod这个概念管理的，所谓mod大家可以理解为namespace。我们可以使用use语句把其他模块中的内容引入到当前模块中来。关于Rust模块系统的详细说明，可参见本书第五部分。

Rust有一个极简标准库，叫作std，除了极少数嵌入式系统下无法使用标准库之外，绝大部分情况下，我们都需要用到标准库里面的东西。为了给大家减少麻烦，Rust编译器对标准库有特殊处理。默认情况下，用户不需要手动添加对标准库的依赖，编译器会自动引入对标准库的依赖。除此之外，标准库中的某些type、trait、function、macro等实在是太常用了。每次都写use语句确实非常无聊，因此标准库提供了一个std：：prelude模块，在这个模块中导出了一些最常见的类型、trait等东西，编译器会为用户写的每个crate自动插入一句话：

```
use std::prelude::*;
```

这样，标准库里面的这些最重要的类型、trait等名字就可以直接使用，而无须每次都写全称或者use语句。

Prelude模块的源码在src/libstd/prelude/文件夹下。我们可以看到，目前的mod.rs中，直接导出了v1模块中的内容，而v1.rs中，则是编译器为我们自动导入的相关trait和类型。

1.5 Format格式详细说明

在后面的内容中，我们还会大量使用println！宏，因此提前介绍一下这个宏的基本用法。跟C语言的printf函数类似，这个宏也支持各种格式控制，示例如下：

```
fn main() {
    println!("{}", 1);             // 默认用法,打印Display
    println!("{:o}", 9);           // 八进制
    println!("{:x}", 255);         // 十六进制 小写
    println!("{:X}", 255);         // 十六进制 大写
    println!("{:p}", &0);          // 指针
    println!("{:b}", 15);          // 二进制
    println!("{:e}", 10000f32);    // 科学计数(小写)
    println!("{:E}", 10000f32);    // 科学计数(大写)

    println!("{:?}", "test");      // 打印Debug
    println!("{:#?}", ("test1", "test2"));        // 带换行和缩进的Debug打印

    println!("{a} {b} {b}", a = "x", b = "y");   // 命名参数
}
```

Rust中还有一系列的宏，都是用的同样的格式控制规则，如format！write！writeln！等。详细文档可以参见标准库文档中std：：fmt模块中的说明。

Rust标准库中之所以设计了这么一个宏来做标准输出，主要是为了更好地错误检查。大家可以试试，如果出现参数个数、格式等各种原因不匹配会直接导致编译错误。而函数则不具备字符串格式化的静态检查功能，如果出现了不匹配的情况，只能是运行期错误。这个宏最终还是调用了std：：io模块内提供的一些函数来完成的。如果用户需要更精细地控制标准输出操作，也可以直接调用标准库来完成。

第2章　变量和类型

2.1　变量声明

Rust的变量必须先声明后使用。对于局部变量，最常见的声明语法为：

```
let variable : i32 = 100;
```

与传统的C/C++语言相比，Rust的变量声明语法不同。这样设计主要有以下几个方面的考虑。

1.语法分析更容易

从语法分析的角度来说，Rust的变量声明语法比C/C++语言的简单，局部变量声明一定是以关键字let开头，类型一定是跟在冒号：的后面。语法歧义更少，语法分析器更容易编写。

2.方便引入类型推导功能

Rust的变量声明的一个重要特点是：要声明的变量前置，对它的类型描述后置。这也是吸取了其他语言的教训后的结果。因为在变量声明语句中，最重要的是变量本身，而类型其实是个附属的额外描述，并非必不可少的部分。如果我们可以通过上下文环境由编译器自动分析出这个变量的类型，那么这个类型描述完全可以省略不写。Rust一开始的设计就考虑了类型自动推导功能，因此类型后置的语法更合适。

3.模式解构

let语句不光是局部变量声明语句，而且具有pattern destructure（模式解构）的功能。关于“模式解构”的内容在后面的章节会详细描述。

实际上，包括C++/C#/Java等传统编程语言都开始逐步引入这种声明语法，目的是相似的。

Rust中声明变量缺省是“只读”的，比如如下程序：

```
fn main() {
    let x = 5;
    x = 10;
}
```

会得到“re-assignment of immutable variable`x`”这样的编译错误。

如果我们需要让变量是可写的，那么需要使用mut关键字：

```
let mut x = 5; // mut x: i32
x = 10;
```

此时，变量x才是可读写的。

实际上，let语句在此处引入了一个模式解构，我们不能把let mut视为一个组合，而应该将mut x视为一个组合。

mut x是一个“模式”，我们还可以用这种方式同时声明多个变量：

```
let (mut a, mut b) = (1, 2);
let Point { x: ref a, y: ref b} = p;
```

其中，赋值号左边的部分是一个“模式”，第一行代码是对tuple的模式解构，第二行代码是对结构体的模式解构。所以，在Rust中，一般把声明的局部变量并初始化的语句称为“变量绑定”，强调的是“绑定”的含义，与C/C++中的“赋值初始化”语句有所区别。

Rust中，每个变量必须被合理初始化之后才能被使用。使用未初始化变量这样的错误，在Rust中是不可能出现的（利用unsafe做hack除外）。如下这个简单的程序，也不能编译通过：

```
fn main() {
    let x: i32;
    println!("{}", x);
}
```

错误信息为：

```
error: use of possibly uninitialized variable: `x`
```

编译器会帮我们做一个执行路径的静态分析，确保变量在使用前一定被初始化：

```
fn test(condition: bool) {
    let x: i32; // 声明 x,不必使用 mut 修饰
    if condition {

        x = 1;  // 初始化 x,不需要 x 是 mut 的,因为这是初始化,不是修改

        println!("{}", x);
    }
    // 如果条件不满足,x 没有被初始化

    // 但是没关系,只要这里不使用 x 就没事
}
```

类型没有“默认构造函数”，变量没有“默认值”。对于let x：i32；如果没有显式赋值，它就没有被初始化，不要想当然地以为它的值是0。

Rust里的合法标识符（包括变量名、函数名、trait名等）必须由数字、字母、下划线组成，且不能以数字开头。这个规定和许多现有的编程语言是一样的。Rust将来会允许其他Unicode字符做标识符，只是目前这个功能的优先级不高，还没有最终定下来。另外还有一个raw identifier功能，可以提供一个特殊语法，如r#self，让用户可以以关键字作为普通标识符。这只是为了应付某些特殊情况时迫不得已的做法。

Rust里面的下划线是一个特殊的标识符，在编译器内部它是被特殊处理的。它跟其他标识符有许多重要区别。比如，以下代码就编译不过：

```
fn main() {
    let _ = "hello";
    println!("{}", _);
}
```

我们不能在表达式中使用下划线来作为普通变量使用。下划线表达的含义是“忽略这个变量绑定，后面不会再用到了”。在后面讲析构的时

候，还会提到这一点。

2.1.1 变量遮蔽

Rust允许在同一个代码块中声明同样名字的变量。如果这样做，后面声明的变量会将前面声明的变量“遮蔽”（Shadowing）起来。

```
fn main() {
    let x = "hello";
    println!("x is {}", x);

    let x = 5;
    println!("x is {}", x);
}
```

上面这个程序是可以编译通过的。请注意第5行的代码，它不是x=5；，它前面有一个let关键字。如果没有这个let关键字，这条语句就是对x的重新绑定（重新赋值）。而有了这个let关键字，就是又声明了一个新的变量，只是它的名字恰巧与前面一个变量相同而已。

但是这两个x代表的内存空间完全不同，类型也完全不同，它们实际上是两个不同的变量。从第5行开始，一直到这个代码块结束，我们没有任何办法再去访问前一个x变量，因为它的名字已经被遮蔽了。

变量遮蔽在某些情况下非常有用，比如，我们需要在同一个函数内部把一个变量转换为另一个类型的变量，但又不想给它们起不同的名字。再比如，在同一个函数内部，需要修改一个变量绑定的可变性。例如，我们对一个可变数组执行初始化，希望此时它是可读写的，但是初始化完成后，我们希望它是只读的。可以这样做：

```
// 注意：这段代码只是演示变量遮蔽功能,并不是Vec类型的最佳初始化方法
fn main() {
    let mut v = Vec::new();         // v 必须是mut修饰,因为我们需要对它写入数据
    v.push(1);
    v.push(2);
    v.push(3);

    let v = v;                      // 从这里往下,v成了只读变量,可读写变量v已经被遮蔽,无法再i
    for i in &v {
        println!("{}", i);
    }
}
```

反过来，如果一个变量是不可变的，我们也可以通过变量遮蔽创建一个新的、可变的同名变量。

```
fn main() {
    let v = Vec::new();
    let mut v = v;
    v.push(1);
    println!("{:?}", v);
}
```

请注意，这个过程是符合“内存安全”的。“内存安全”的概念一直是Rust关注的重点，我们将在第二部分详细讲述。在上面这个示例中，我们需要理解的是，一个“不可变绑定”依然是一个“变量”。虽然我们没办法通过这个“变量绑定”修改变量的值，但是我们重新使用“可变绑定”之后，还是有机会修改的。这样做并不会产生内存安全问题，因为我们对这块内存拥有完整的所有权，且此时没有任何其他引用指向这个变量，对这个变量的修改是完全合法的。Rust的可变性控制规则与其他语言不一样。更多内容请参阅本书第二部分内存安全。

实际上，传统编程语言C/C++中也存在类似的功能，只不过它们只允许嵌套的区域内部的变量出现遮蔽。而Rust在这方面放得稍微宽一点，同一个语句块内部声明的变量也可以发生遮蔽。

2.1.2 类型推导

Rust的类型推导功能是比较强大的。它不仅可以从变量声明的当前语句中获取信息进行推导，而且还能通过上下文信息进行推导。

```
fn main() {
    // 没有明确标出变量的类型,但是通过字面量的后缀,
    // 编译器知道elem的类型为u8
    let elem = 5u8;

    // 创建一个动态数组,数组内包含的是什么元素类型可以不写
    let mut vec = Vec::new();
    vec.push(elem);
    // 到后面调用了push函数,通过elem变量的类型,
    // 编译器可以推导出vec的实际类型是 Vec<u8>

    println!("{:?}", vec);
}
```

我们甚至还可以只写一部分类型，剩下的部分让编译器去推导，比如下面的这个程序，我们只知道players变量是Vec动态数组类型，但是里面包含什么元素类型并不清楚，可以在尖括号中用下划线来代替：

```
fn main() {
    let player_scores = [
        ("Jack", 20), ("Jane", 23), ("Jill", 18), ("John", 19),
    ];

    // players 是动态数组,内部成员的类型没有指定,交给编译器自动推导
    let players : Vec<_> = player_scores
        .iter()
        .map(|&(player, _score)| {
            player
        })
        .collect();

    println!("{:?}", players);
}
```

自动类型推导和“动态类型系统”是两码事。Rust依然是静态类型的。一个变量的类型必须在编译阶段确定，且无法更改，只是某些时候不需要在源码中显式写出来而已。这只是编译器给我们提供的一个辅助工具。

Rust只允许“局部变量/全局变量”实现类型推导，而函数签名等场景下是不允许的，这是故意这样设计的。这是因为局部变量只有局部的影响，全局变量必须当场初始化而函数签名具有全局性影响。函数签名如果使用自动类型推导，可能导致某个调用的地方使用方式发生变化，它的参数、返回值类型就发生了变化，进而导致远处另一个地方的编译错误，这是设计者不希望看到的情况。

2.1.3 类型别名

我们可以用type关键字给同一个类型起个别名（type alias）。示例如下：

```
type Age = u32;

fn grow(age: Age, year: u32) -> Age {
    age + year
}
```

```
fn main() {
    let x : Age = 20;
    println!("20 years later: {}", grow(x, 20));
}
```

类型别名还可以用在泛型场景，比如：

```
type Double<T> = (T, Vec<T>); // 小括号包围的是一个 tuple,请参见后文中的复合数据类型
```

那么以后使用Double<i32>的时候，就等同于（i32，Vec<i32>），可以简化代码。

2.1.4 静态变量

Rust中可以用static关键字声明静态变量。如下所示：

```
static GLOBAL: i32 = 0;
```

与let语句一样，static语句同样也是一个模式匹配。与let语句不同的是，用static声明的变量的生命周期是整个程序，从启动到退出。static变量的生命周期永远是'static，它占用的内存空间也不会在执行过程中回收。这也是Rust中唯一的声明全局变量的方法。

由于Rust非常注重内存安全，因此全局变量的使用有许多限制。这些限制都是为了防止程序员写出不安全的代码：

·全局变量必须在声明的时候马上初始化；

·全局变量的初始化必须是编译期可确定的常量，不能包括执行期才能确定的表达式、语句和函数调用；

·带有mut修饰的全局变量，在使用的时候必须使用unsafe关键字。

示例如下：

```
fn main() {
```

```
//局部变量声明,可以留待后面初始化,只要保证使用前已经初始化即可
    let x;
    let y = 1_i32;
    x = 2_i32;
    println!("{} {}", x, y);

//全局变量必须声明的时候初始化,因为全局变量可以写到函数外面,被任意一个函数使用
    static G1 : i32 = 3;
    println!("{}", G1);

//可变全局变量无论读写都必须用 unsafe修饰
    static mut G2 : i32 = 4;
    unsafe {
        G2 = 5;
        println!("{}", G2);
    }

//全局变量的内存不是分配在当前函数栈上,函数退出的时候,并不会销毁全局变量占用的内存空间,程序退出才会
}
```

Rust禁止在声明static变量的时候调用普通函数，或者利用语句块调用其他非const代码：

```
// 这样是允许的
static array : [i32; 3] = [1,2,3];
// 这样是不允许的
static vec : Vec<i32> = { let mut v = Vec::new();  v.push(1); v  };
```

调用const fn是允许的：

```
#![feature(const_fn)]
fn main() {
    use std::sync::atomic::AtomicBool;
    static FLAG: AtomicBool = AtomicBool::new(true);
}
```

因为const fn是编译期执行的。这个功能在编写本书的时候目前还没有stable，因此需要使用nightly版本并打开feature gate才能使用。

Rust不允许用户在main函数之前或者之后执行自己的代码。所以，比较复杂的static变量的初始化一般需要使用lazy方式，在第一次使用的时候初始化。在Rust中，如果用户需要使用比较复杂的全局变量初始化，推荐使用lazy_static库。

2.1.5 常量

在Rust中还可以用const关键字做声明。如下所示：

```
const GLOBAL: i32 = 0;
```

使用const声明的是常量，而不是变量。因此一定不允许使用mut关键字修饰这个变量绑定，这是语法错误。常量的初始化表达式也一定要是一个编译期常量，不能是运行期的值。它与static变量的最大区别在于：编译器并不一定会给const常量分配内存空间，在编译过程中，它很可能会被内联优化。因此，用户千万不要用hack的方式，通过unsafe代码去修改常量的值，这么做是没有意义的。以const声明一个常量，也不具备类似let语句的模式匹配功能。

2.2 基本数据类型

2.2.1 bool

布尔类型（bool）代表的是“是”和“否”的二值逻辑。它有两个值：true和false。一般用在逻辑表达式中，可以执行“与”“或”“非”等运算。

```
fn main() {
    let x = true;
    let y: bool = !x;       // 取反运算

    let z = x && y;         // 逻辑与,带短路功能
    println!("{}", z);

    let z = x || y;         // 逻辑或,带短路功能
    println!("{}", z);

    let z = x & y;          // 按位与,不带短路功能
    println!("{}", z);

    let z = x | y;          // 按位或,不带短路功能
    println!("{}", z);

    let z = x ^ y;          // 按位异或,不带短路功能
    println!("{}", z);
}
```

一些比较运算表达式的类型就是bool类型：

```
fn logical_op(x: i32, y: i32) {
    let z : bool = x < y;
    println!("{}", z);
}
```

bool类型表达式可以用在if/while等表达式中，作为条件表达式。比如：

```
if a >= b {
    ...
} else {
    ...
}
```

2.2.2 char

字符类型由char表示。它可以描述任何一个符合unicode标准的字符值。在代码中，单个的字符字面量用单引号包围。

```
let love = '❤';        // 可以直接嵌入任何 unicode 字符
```

字符类型字面量也可以使用转义符：

```
let c1 = '\n';          // 换行符
let c2 = '\x7f';        // 8 bit 字符变量
let c3 = '\u{7FFF}';    // unicode字符
```

因为char类型的设计目的是描述任意一个unicode字符，因此它占据的内存空间不是1个字节，而是4个字节。

对于ASCII字符其实只需占用一个字节的空间，因此Rust提供了单字节字符字面量来表示ASCII字符。我们可以使用一个字母b在字符或者字符串前面，代表这个字面量存储在u8类型数组中，这样占用空间比char型数组要小一些。示例如下：

```
let x :u8 = 1;
let y :u8 = b'A';
let s :&[u8;5] = b"hello";
let r :&[u8;14] = br#"hello \n world"#;
```

2.2.3 整数类型

Rust有许多的数字类型，主要分为整数类型和浮点数类型。本节讲解整数类型。各种整数类型之间的主要区分特征是：有符号/无符号，占据空间大小。具体见表2-1。

表 2-1

整数类型	有符号	无符号
8 bits	i8	u8
16 bits	i16	u16
32 bits	i32	u32
64 bits	i64	u64
128 bits	i128	u128
Pointer size	isize	usize

所谓有符号/无符号，指的是如何理解内存空间中的bit表达的含义。如果一个变量是有符号类型，那么它的最高位的那一个bit就是“符号位”，表示该数为正值还是负值。如果一个变量是无符号类型，那么它的最高位和其他位一样，表示该数的大小。比如对于一个byte大小（8 bits）的数据来说，如果存的是无符号数，那么它的表达范围是0～255，如果存的是有符号数，那么它的表达范围是-128～127。

关于各个整数类型所占据的空间大小，在名字中就已经表现得很明确了，Rust原生支持了从8位到128位的整数。需要特别关注的是isize和usize类型。它们占据的空间是不定的，与指针占据的空间一致，与所在的平台相关。如果是32位系统上，则是32位大小；如果是64位系统上，则是64位大小。在C++中与它们相对应的类似类型是int_ptr和uint_ptr。Rust的这一策略与C语言不同，C语言标准中对许多类型的大小并没有做强制规定，比如int、long、double等类型，在不同平台上都可能是不同的大小，这给许多程序员带来了不必要的麻烦。相反，在语言标准中规定好各个类型的大小，让编译器针对不同平台做适配，生成不同的代码，是更合理的选择。

数字类型的字面量表示可以有许多方式：

```
let var1 : i32 = 32;          // 十进制表示
let var2 : i32 = 0xFF;        // 以0x开头代表十六进制表示
let var3 : i32 = 0o55;        // 以0o开头代表八进制表示
let var4 : i32 = 0b1001;      // 以0b开头代表二进制表示
```

注意！在C/C++/JavaScript语言中以0开头的数字代表八进制坑过不少人，Rust中设计不一样。

在所有的数字字面量中，可以在任意地方添加任意的下划线，以方便阅读：

```
let var5 = 0x_1234_ABCD;          //使用下划线分割数字,不影响语义,但是极大地提升了阅读体验。
```

字面量后面可以跟后缀，可代表该数字的具体类型，从而省略掉显示类型标记：

```
let var6 = 123usize;          // i6变量是usize类型
let var7 = 0x_ff_u8;          // i7变量是u8类型
let var8 = 32;                // 不写类型,默认为 i32 类型
```

在Rust中，我们可以为任何一个类型添加方法，整型也不例外。比如在标准库中，整数类型有一个方法是pow，它可以计算n次幂，于是我们可以这么使用：

```
let x : i32 = 9;
println!("9 power 3 = {}", x.pow(3));
```

同理，我们甚至可以不使用变量，直接对整型字面量调用函数：

```
fn main() {
    println!("9 power 3 = {}", 9_i32.pow(3));
}
```

我们可以看到这是非常方便的设计。

对于整数类型，如果Rust编译器通过上下文无法分析出该变量的具体类型，则自动默认为i32类型。比如：

```
fn main() {
    let x = 10;
    let y = x * x;
    println!("{}", y);
}
```

在此例中，编译器只知道x是一个整数，但是具体是i8 i16 i32或者u8 u16 u32等，并没有足够的信息判断，这些都是有可能的。在这种情况下，编译器就默认把x当成i32类型处理。这么做的好处是，很多时候，我们不想在每个地方都明确地指定数字类型，这么做很麻烦。给编译器指定一个在信息不足情况下的“缺省”类型会更方便一点。

2.2.4 整数溢出

在整数的算术运算中，有一个比较头疼的事情是“溢出”。在C语言中，对于无符号类型，算术运算永远不会overflow，如果超过表示范围，则自动舍弃高位数据。对于有符号类型，如果发生了overflow，标准规定这是undefined behavior，也就是说随便怎么处理都可以。

未定义行为有利于编译器做一些更激进的性能优化，但是这样的规定有可能导致在程序员不知情的某些极端场景下，产生诡异的bug。

Rust的设计思路更倾向于预防bug，而不是无条件地压榨效率，Rust设计者希望能尽量减少“未定义行为”。比如彻底杜绝“Segment Fault”这种内存错误是Rust的一个重要设计目标。当然还有其他许多种类的bug，即便是无法完全解决，我们也希望能尽量避免。整数溢出就是这样的一种bug。

Rust在这个问题上选择的处理方式为：默认情况下，在debug模式下编译器会自动插入整数溢出检查，一旦发生溢出，则会引发panic；在release模式下，不检查整数溢出，而是采用自动舍弃高位的方式。示例如下：

```
fn arithmetic(m: i8, n: i8) {
    // 加法运算,有溢出风险
    println!("{}", m + n);
}

fn main() {
    let m : i8 = 120;
    let n : i8 = 120;
    arithmetic(m, n);
}
```

如果我们编译debug版本：

```
rustc test.rs
```

执行这个程序，结果为：

```
thread 'main' panicked at 'attempt to add with overflow', test.rs:3:20
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

可以看到，程序执行时发生了panic。有关panic的详细解释，需要参见第18章，此处无须深入细节。

如果编译一个优化后的版本，加上-O选项：

```
rustc -O test.rs
```

执行时没有错误，而是使用了自动截断策略：

```
$ ./test
-16
```

Rust编译器还提供了一个独立的编译开关供我们使用，通过这个开关，可以设置溢出时的处理策略：

```
$ rustc -C overflow-checks=no test.rs
```

“-C overflow-checks=”可以写“yes”或者“no”，打开或者关闭溢出检查。如果我们用上面这个命令编译，执行可见：

```
$ ./test
-16
```

虽然它还是debug版本，但我们依然有办法关闭溢出检查。

如果在某些场景下，用户确实需要更精细地自主控制整数溢出的行为，可以调用标准库中的checked_*、saturating_*和wrapping_*系列函数。

```
fn main() {
    let i = 100_i8;
    println!("checked {:?}", i.checked_add(i));
    println!("saturating {:?}", i.saturating_add(i));
    println!("wrapping {:?}", i.wrapping_add(i));
}
```

输出结果为：

```
checked None
saturating 127
wrapping -56
```

可以看到：checked_*系列函数返回的类型是Option<_>，当出现溢出的时候，返回值是None；saturating_*系列函数返回类型是整数，如果溢出，则给出该类型可表示范围的“最大/最小”值；wrapping_*系列函数则是直接抛弃已经溢出的最高位，将剩下的部分返回。在对安全性要求非常高的情况下，强烈建议用户尽量使用这几个方法替代默认的算术运算符来做数学运算，这样表意更清晰。在Rust标准库中就大量使用了这几个方法，而不是简单地使用算术运算符，值得大家参考。

在很多情况下，整数溢出应该被处理为截断，即丢弃最高位。为了方便用户，标准库还提供了一个叫作std：：num：：Wrapping<T>的类型。它重载了基本的运算符，可以被当成普通整数使用。凡是被它包裹起来的整数，任何时候出现溢出都是截断行为。常见使用示例如下：

```
use std::num::Wrapping;

fn main() {
    let big = Wrapping(std::u32::MAX);
    let sum = big + Wrapping(2_u32);
    println!("{}", sum.0);
}
```

不论用什么编译选项，上述代码都不会触发panic，任何情况下执行结果都是一致的。

标准库中还提供了许多有用的方法，在此不一一赘述，请大家参考标准API文档。

2.2.5 浮点类型

Rust提供了基于IEEE 754-2008标准的浮点类型。按占据空间大小区分，分别为f32和f64，其使用方法与整型差别不大。浮点数字面量表示方式有如下几种：

```
let f1 = 123.0f64;        // type f64
let f2 = 0.1f64;          // type f64
let f3 = 0.1f32;          // type f32
let f4 = 12E+99_f64;      // type f64 科学计数法
let f5 : f64 = 2.;        // type f64
```

与整数类型相比，Rust的浮点数类型相对复杂得多。浮点数的麻烦之处在于：它不仅可以表达正常的数值，还可以表达不正常的数值。

在标准库中，有一个std：：num：：FpCategory枚举，表示了浮点数可能的状态：

```
enum FpCategory {
    Nan,
    Infinite,
    Zero,
    Subnormal,
    Normal,
}
```

其中Zero表示0值、Normal表示正常状态的浮点数。其他几个就需要特别解释一下了。

在IEEE 754标准中，规定了浮点数的二进制表达方式：x=（-1）^s*（1+M）*2^e。其中s是符号位，M是尾数，e是指数。尾数M是一个[0，1）范围内的二进制表示的小数。以32位浮点为例，如果只有normal形式的话，0表示为所有位数全0，则最小的非零正数将是尾数最后一位为1的数字，就是（1+2^（-23））*2^（-127），而次小的数字为（1+2^（-22））*2^（-127），这两个数字的差距为2^（-23）*2^（-127）=2^（-150），然而最小的数字和0之间的差距有

（1+2^（-23））*2^（-127），约等于2^（-127），也就是说，数字在渐渐减少到0的过程中突然降到了0。为了减少0与最小数字和最小数字与次小数字之间步长的突然下跌，subnormal规定：当指数位全0的时候，指数表示为-126而不是-127（和指数为最低位为1一致）。然而公式改成（-1）^s*M*2^e，M不再+1，这样最小的数字就变成2^（-23）*2^（-126），次小的数字变成2^（-22）*2^（-126），每两个相邻subnormal数字之差都是2^（-23）*2^（-126），避免了突然降到0。在这种状态下，这个浮点数就处于了Subnormal状态，处于这种状态下的浮点数表示精度比Normal状态下的精度低一点。我们用一个示例来演示一下什么是Subnormal状态的浮点数：

```
fn main() {
    // 变量 small 初始化为一个非常小的浮点数
    let mut small = std::f32::EPSILON;
    // 不断循环,让 small 越来越趋近于 0,直到最后等于0的状态
    while small > 0.0 {
        small = small / 2.0;
        println!("{} {:?}", small, small.classify());
    }
}
```

编译，执行，发现循环几十次之后，数值就小到了无法在32bit范围内合理表达的程度，最终收敛到了0，在后面表示非常小的数值的时候，浮点数就已经进入了Subnormal状态。

Infinite和Nan是带来更多麻烦的特殊状态。Infinite代表的是“无穷大”，Nan代表的是“不是数字”（not a number）。

什么情况会产生“无穷大”和“不是数字”呢？举例说明：

```
fn main() {
    let x = 1.0f32 / 0.0;
    let y = 0.0f32 / 0.0;
    println!("{} {}", x, y);
}
```

编译执行，打印出来的结果分别为inf NaN。非0数除以0值，得到的是inf，0除以0得到的是NaN。

对inf做一些数学运算的时候，它的结果可能与你期望的不一致：

```
fn main() {
    let inf = std::f32::INFINITY;
    println!("{} {} {}", inf * 0.0, 1.0 / inf, inf / inf);
}
```

编译执行，结果为：

```
NaN 0 NaN
```

NaN这个特殊值有个特殊的麻烦，主要问题还在于它不具备“全序”的特点。示例如下：

```
fn main() {
    let nan = std::f32::NAN;
    println!("{} {} {}", nan < nan, nan > nan, nan == nan);
}
```

编译执行，输出结果为：

```
false false false
```

这就很麻烦了，一个数字可以不等于自己。因为NaN的存在，浮点数是不具备“全序关系”（total order）的。关于“全序”和“偏序”的问题，本节就不展开讲解了，后面讲到trait的时候，再给大家介绍PartialOrd和Ord这两个trait。

2.2.6 指针类型

无GC的编程语言，如C、C++以及Rust，对数据的组织操作有更多的自由度，具体表现为：

·同一个类型，某些时候可以指定它在栈上，某些时候可以指定它在堆上。内存分配方式可以取决于使用方式，与类型本身无关。

·既可以直接访问数据，也可以通过指针间接访问数据。可以针对任何一个对象取得指向它的指针。

·既可以在复合数据类型中直接嵌入别的类型的实体，也可以使用指针，间接指向别的类型。

·甚至可能在复合数据类型末尾嵌入不定长数据构造出不定长的复合数据类型。

Rust里面也有指针类型，而且不止一种指针类型。常见的几种指针类型见表2-2。

表　2-2

类型名	简介
Box<T>	指向类型 T 的、具有所有权的指针，有权释放内存
&T	指向类型 T 的借用指针，也称为引用，无权释放内存，无权写数据
&mut T	指向类型 T 的 mut 型借用指针，无权释放内存，有权写数据
*const T	指向类型 T 的只读裸指针，没有生命周期信息，无权写数据
*mut T	指向类型 T 的可读写裸指针，没有生命周期信息，有权写数据

除此之外，在标准库中还有一种封装起来的可以当作指针使用的类型，叫“智能指针”（smart pointer）。常见的智能指针见表2-3。

表　2-3

类型名	简介
Rc<T>	指向类型 T 的引用计数指针，共享所有权，线程不安全
Arc<T>	指向类型 T 的原子型引用计数指针，共享所有权，线程安全
Cow<'a, T>	Clone-on-write，写时复制指针。可能是借用指针，也可能是具有所有权的指针

有关这几种指针的使用方法和设计原理，请参见本书第二部分。

2.2.7　类型转换

Rust对不同类型之间的转换控制得非常严格。即便是下面这样的程序，也会出现编译错误：

```
fn main() {
    let var1 : i8 = 41;
    let var2 : i16 = var1;
}
```

编译结果为mismatched types！i8类型的变量竟然无法向i16类型的变量赋值！这可能对很多用户来说都是一个意外。

Rust提供了一个关键字as，专门用于这样的类型转换：

```
fn main() {
    let var1 : i8 = 41;
    let var2 : i16 = var1 as i16;
}
```

也就是说，Rust设计者希望在发生类型转换的时候不是偷偷摸摸进行的，而是显式地标记出来，防止隐藏的bug。虽然在许多时候会让代码显得不那么精简，但这也算是一种合理的折中。

as关键字也不是随便可以用的，它只允许编译器认为合理的类型转换。任意类型转换是不允许的：

```
let a = "some string";
let b = a as u32; // 编译错误
```

有些时候，甚至需要连续写多个as才能转成功，比如&i32类型就不能直接转换为*mut i32类型，必须像下面这样写才可以：

```
fn main() {
    let i = 42;
    // 先转为 *const i32,再转为 *mut i32
    let p = &i as *const i32 as *mut i32;
    println!("{:p}", p);
}
```

as表达式允许的类型转换如表2-4所示。对于表达式e as U，e是表达式，U是要转换的目标类型，表2-4中所示的类型转换是允许的。

表　2-4

Type of `e`	`U`
Integer or Float type	Integer or Float type
C-like enum	Integer type
`bool` or `char`	Integer type
`u8`	`char`
`*T`	`*V` where `V: Sized *`
`*T` where `T: Sized`	Numeric type
Integer type	`*V` where `V: Sized`
`&[T; n]`	`*const T`
Function pointer	`*V` where `V: Sized`
Function pointer	Integer

如果需要更复杂的类型转换，一般是使用标准库的From Into等trait，请参见第26章。

2.3 复合数据类型

复合数据类型可以在其他类型的基础上形成更复杂的组合关系。

本章介绍tuple、struct、enum等几种复合数据类型。数组留到第6章介绍。

2.3.1 tuple

tuple指的是“元组”类型，它通过圆括号包含一组表达式构成。tuple内的元素没有名字。tuple是把几个类型组合到一起的最简单的方式。比如：

```
let a = (1i32, false);                  // 元组中包含两个元素,第一个是i32类型,第二个是bool类型
let b = ("a", (1i32, 2i32));            // 元组中包含两个元素,第二个元素本身也是元组,它又包含了两
```

如果元组中只包含一个元素，应该在后面添加一个逗号，以区分括号表达式和元组：

```
let a = (0,);       // a是一个元组,它有一个元素
let b = (0);        // b是一个括号表达式,它是i32类型
```

访问元组内部元素有两种方法，一种是“模式匹配”（pattern destructuring），另外一种是“数字索引”：

```
let p = (1i32, 2i32);
let (a, b) = p;

let x = p.0;
let y = p.1;
println!("{} {} {} {}", a, b, x, y);
```

在第7章中会对“模式匹配”做详细解释。

元组内部也可以一个元素都没有。这个类型单独有一个名字，叫

unit（单元类型）：

```
let empty : () = ();
```

可以说，unit类型是Rust中最简单的类型之一，也是占用空间最小的类型之一。空元组和空结构体struct Foo；一样，都是占用0内存空间。

```
fn main() {
    println!("size of i8 {}" , std::mem::size_of::<i8>());
    println!("size of char {}" , std::mem::size_of::<char>());
    println!("size of '()' {}" , std::mem::size_of::<()>());
}
```

上面的程序中，std：：mem：：size_of函数可以计算一个类型所占用的内存空间。可以看到，i8类型占用1 byte，char类型占用4 bytes，空元组占用0 byte。

与C++中的空类型不同，Rust中存在实打实的0大小的类型。在C++标准中，有明确的规定，是这么说的：

Complete objects and member subobjects of class type shall have nonzero size.

```
class Empty {};
Empty emp;
assert(sizeof(emp) != 0);
```

2.3.2 struct

结构体（struct）与元组类似，也可以把多个类型组合到一起，作为新的类型。区别在于，它的每个元素都有自己的名字。举个例子：

```
struct Point {
    x: i32,
    y: i32,
}
```

每个元素之间采用逗号分开，最后一个逗号可以省略不写。类型依旧跟在冒号后面，但是不能使用自动类型推导功能，必须显式指定。struct类型的初始化语法类似于json的语法，使用“成员–冒号–值”的格式。

```
fn main() {
    let p = Point { x: 0, y: 0};
    println!("Point is at {} {}", p.x, p.y);
}
```

有些时候，Rust允许struct类型的初始化使用一种简化的写法。如果有局部变量名字和成员变量名字恰好一致，那么可以省略掉重复的冒号初始化：

```
fn main() {
    // 刚好局部变量名字和结构体成员名字一致
    let x = 10;
    let y = 20;
    // 下面是简略写法,等同于 Point { x: x, y: y },同名字的相对应
    let p = Point { x, y };
    println!("Point is at {} {}", p.x, p.y);
}
```

访问结构体内部的元素，也是使用“点”加变量名的方式。当然，我们也可以使用“模式匹配”功能：

```
fn main() {
    let p = Point { x: 0, y: 0};
    // 声明了px 和 py,分别绑定到成员 x 和成员 y
    let Point { x : px, y : py } = p;
    println!("Point is at {} {}", px, py);
    // 同理,在模式匹配的时候,如果新的变量名刚好和成员名字相同,可以使用简写方式
    let Point { x, y } = p;
    println!("Point is at {} {}", x, y);
}
```

Rust设计了一个语法糖，允许用一种简化的语法赋值使用另外一个struct的部分成员。比如：

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
```

```
}

fn default() -> Point3d {
    Point3d { x: 0, y: 0, z: 0 }
}

// 可以使用default()函数初始化其他的元素
// ..expr 这样的语法,只能放在初始化表达式中,所有成员的最后最多只能有一个
let origin = Point3d { x: 5, ..default()};
let point = Point3d { z: 1, x: 2, ..origin };
```

如前所说，与tuple类似，struct内部成员也可以是空：

```
//以下三种都可以,内部可以没有成员
struct Foo1;
struct Foo2();
struct Foo3{}
```

2.3.3 tuple struct

Rust有一种数据类型叫作tuple struct，它就像是tuple和struct的混合。区别在于，tuple struct有名字，而它们的成员没有名字：

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
```

它们可以被想象成这样的结构体：

```
struct Color{
    0: i32,
    1: i32,
    2: i32,
}
struct Point {
    0: i32,
    1: i32,
    2: i32,
}
```

因为这两个类型都有自己的名字，虽然它们的内部结构是一样的，但是它们是完全不同的两个类型。有时候我们不需要特别关心结构体内部成员的名字，可以采用这种语法。

tuple、struct、struct tuple起的作用都是把几个不同类型的成员打包组合成一个类型。它们的区别如表2-5所示。

表　2-5

类型名称	tuple	struct	tuple struct
语法	没名字圆括号	名字加大括号	名字加圆括号
类型名字	没有单独的名字	有单独的名字	有单独的名字
成员名字	没有单独的名字	有单独的名字	没单独的名字

它们除了在取名上有这些区别外，没有其他区别。它们有一致的内存对齐策略、一致的占用空间规则，也有类似的语法。从下面这个例子可以看出它们的语法是很一致的：

```
// define struct
struct T1 {
    v: i32
}
// define tuple struct
struct T2(i32);

fn main() {
    let v1 = T1 { v: 1 };
    let v2 = T2(1);              // init tuple struct
    let v3 = T2 { 0: 1 };        // init tuple struct

    let i1 = v1.v;
    let i2 = v2.0;
    let i3 = v3.0;
}
```

tuple struct有一个特别有用的场景，那就是当它只包含一个元素的时候，就是所谓的newtype idiom。因为它实际上让我们非常方便地在一个类型的基础上创建了一个新的类型。举例如下：

```
fn main() {
    struct Inches(i32);

    fn f1(value : Inches) {}
    fn f2(value : i32) {}

    let v : i32 = 0;
    f1(v);  // 编译不通过,'mismatched types'
    f2(v);
}
```

以上程序编译不通过，因为Inches类型和i32是不同的类型，函数调用参数不匹配。

但是，如果我们把以上程序改一下，使用type alias（类型别名）实现，那么就可以编译通过了：

```
fn type_alias() {
    type I = i32;

    fn f1(v : I) {}
    fn f2(v : i32) {}

    let v : i32 = 0;
    f1(v);
    f2(v);
}
```

从上面的讲解可以看出，通过关键字type，我们可以创建一个新的类型名称，但是这个类型不是全新的类型，而只是一个具体类型的别名。在编译器看来，这个别名与原先的具体类型是一模一样的。而使用tuple struct做包装，则是创造了一个全新的类型，它跟被包装的类型不能发生隐式类型转换，可以具有不同的方法，满足不同的trait，完全按需而定。

2.3.4 enum

如果说tuple、struct、tuple struct在Rust中代表的是多个类型的“与”关系，那么enum类型在Rust中代表的就是多个类型的“或”关系。

与C/C++中的枚举相比，Rust中的enum要强大得多，它可以为每个成员指定附属的类型信息。比如说我们可以定义这样的类型，它内部可能是一个i32型整数，或者是f32型浮点数：

```
enum Number {
    Int(i32),
    Float(f32),
}
```

Rust的enum中的每个元素的定义语法与struct的定义语法类似。可以像空结构体一样，不指定它的类型；也可以像tuple struct一样，用圆

括号加无名成员；还可以像正常结构体一样，用大括号加带名字的成员。

用enum把这些类型包含到一起之后，就组成了一个新的类型。

要使用enum，一般要用到“模式匹配”。模式匹配是很重要的一部分，用第7章来详细讲解。这里我们给出一个用match语句读取enum内部数据的示例：

```
enum Number {
    Int(i32),
    Float(f32),
}

fn read_num(num: &Number) {
    match num {
        // 如果匹配到了 Number::Int 这个成员,那么value的类型就是 i32
        &Number::Int(value) => println!("Integer {}", value),
        // 如果匹配到了 Number::Float 这个成员,那么value的类型就是 f32
        &Number::Float(value) => println!("Float {}", value),
    }
}
fn main() {
    let n: Number = Number::Int(10);
    read_num(&n);
}
```

Rust的enum与C/C++的enum和union都不一样。它是一种更安全的类型，可以被称为“tagged union”。从C语言的视角来看Rust的enum类型，重写上面这段代码，它的语义类似这样：

```
#include <stdio.h>
#include <stdint.h>

// C 语言模拟 Rust 的 enum
struct Number {
    enum {Int, Float} tag;
    union {
        int32_t int_value;
        float   float_value;
    } value;
};

void read_num(struct Number * num) {
    switch(num->tag) {
        case Int:
            printf("Integer %d", num->value.int_value);
            break;
        case Float:
            printf("Float %f", num->value.float_value);
```

```
            break;
        default:
            printf("data error");
            break;
    }
}

int main() {
    struct Number n = { tag : Int, value: { int_value: 10} };
    read_num(&n);
    return 0;
}
```

Rust的enum类型的变量需要区分它里面的数据究竟是哪种变体，所以它包含了一个内部的“tag标记”来描述当前变量属于哪种类型。这个标记对用户是不可见的，通过恰当的语法设计，保证标记与类型始终是匹配的，以防止用户错误地使用内部数据。如果我们用C语言来模拟，就需要程序员自己来保证读写的时候标记和数据类型是匹配的，编译器无法自动检查。当然，上面这个模拟只是为了通俗地解释Rust的enum类型的基本工作原理，在实际中，enum的内存布局未必是这个样子，编译器有许多优化，可以保证语义正确的同时减少内存使用，并加快执行速度。如果是在FFI场景下，要保证Rust里面的enum的内存布局和C语言兼容的话，可以给这个enum添加一个#[repr（C，Int）]属性标签（目前这个设计已经通过，但是还未在编译器中实现）。

我们可以试着把前面定义的Number类型占用的内存空间大小打印出来看看：

```
fn main() {
    // 使用了泛型函数的调用语法,请参考第21章泛型
    println!("Size of Number: {}", std::mem::size_of::<Number>());
    println!("Size of i32:    {}", std::mem::size_of::<i32>());
    println!("Size of f32:    {}", std::mem::size_of::<f32>());
}
```

编译执行可见：

```
Size of Number: 8
Size of i32:    4
Size of f32:    4
```

Number里面要么存储的是i32，要么存储的是f32，它存储数据需要的空间应该是max（sizeof（i32），sizeof（f32））=max（4 byte，4

byte）=4 byte。而它总共占用的内存是8 byte，多出来的4 byte就是用于保存类型标记的。之所以用4 byte，是为了内存对齐。

Rust里面也支持union类型，这个类型与C语言中的union完全一致。但在Rust里面，读取它内部的值被认为是unsafe行为，一般情况下我们不使用这种类型。它存在的主要目的是为了方便与C语言进行交互。

在Rust中，enum和struct为内部成员创建了新的名字空间。如果要访问内部成员，可以使用：：符号。因此，不同的enum中重名的元素也不会互相冲突。例如在下面的程序中，两个枚举内部都有Move这个成员，但是它们不会有冲突。

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}
let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

我们也可以手动指定每个变体自己的标记值：

```
fn main() {
    enum Animal {
        dog = 1,
        cat = 200,
        tiger,
    }
    let x = Animal::tiger as isize;
    println!("{}", x);
}
```

Rust标准库中有一个极其常用的enum类型Option<T>，它的定义如下：

```
enum Option<T> {
    None,
    Some(T),
}
```

由于它实在是太常用，标准库将Option以及它的成员Some、None都加入到了Prelude中，用户甚至不需要use语句声明就可以直接使用。它表示的含义是“要么存在、要么不存在”。比如Option<i32>表达的意思就是“可以是一个i32类型的值，或者没有任何值”。

Rust的enum实际上是一种代数类型系统（Algebraic Data Type，ADT），本书第8章简要介绍什么是ADT。enum内部的variant只是一个名字而已，恰好我们还可以将这个名字作为类型构造器使用。意思是说，我们可以把enum内部的variant当成一个函数使用，示例如下：

```
fn main() {
    let arr = [1,2,3,4,5];
    // 请注意这里的map函数
    let v: Vec<Option<&i32>> = arr.iter().map(Some).collect();
    println!("{:?}", v);
}
```

有关迭代器的知识，请各位读者参考第24章的内容。在这里想说明的问题是，Some可以当成函数作为参数传递给map。这里的Some其实是作为一个函数来使用的，它输入的是&i32类型，输出为Option<&i32>类型。可以用如下方式证明Some确实是一个函数类型，我们把Some初始化给一个unit变量，产生一个编译错误：

```
fn main() {
    let _ : () = Some;
}
```

编译错误是这样写的：

```
error[E0308]: mismatched types
 --> test.rs:3:18
  |
3 |     let _ : () = Some;
  |                  ^^^^ expected (), found fn item
  |
  = note: expected type `()`
             found type `fn(_) -> std::option::Option<_> {std::option::Option<_>::Som
```

可见，enum内部的variant的类型确实是函数类型。

2.3.5 类型递归定义

Rust里面的复合数据类型是允许递归定义的。比如struct里面嵌套同样的struct类型，但是直接嵌套是不行的。示例如下：

```
struct Recursive {
    data: i32,
    rec: Recursive,
}
```

使用rustc--crate-type=lib test.rs命令编译，可以看到如下编译错误：

```
error[E0072]: recursive type `Recursive` has infinite size
 --> test.rs:2:1
  |
2 | struct Recursive {
  | ^^^^^^^^^^^^^^^^ recursive type has infinite size
3 |     data: i32,
4 |     rec: Recursive,
  |     -------------- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make `Re
```

以上编译错误写得非常人性化，不仅写清楚了错误原因，还给出了可能的修复办法。Rust是允许用户手工控制内存布局的语言。直接使用类型递归定义的问题在于，当编译器计算Recursive这个类型大小的时候：

```
size_of::<Recursive>() == 4 + size_of::<Recursive>()
```

这个方程在实数范围内无解。

解决办法很简单，用指针间接引用就可以了，因为指针的大小是固定的，比如：

```
struct Recursive {
    data: i32,
    rec: Box<Recursive>,
}
```

我们把产生了递归的那个成员类型改为了指针，这个类型就非常合理了。

第3章　语句和表达式

语句和表达式是Rust语言实现控制逻辑的基本单元。

3.1 语句

一个Rust程序，是从main函数开始执行的。而函数体内，则是由一条条语句组成的。

Rust程序里，表达式（Expression）和语句（Statement）是完成流程控制、计算求值的主要工具，也是本节要讲的核心部分。在Rust程序里面，表达式可以是语句的一部分，反过来，语句也可以是表达式的一部分。一个表达式总是会产生一个值，因此它必然有类型；语句不产生值，它的类型永远是（）。如果把一个表达式加上分号，那么它就变成了一个语句；如果把语句放到一个语句块中包起来，那么它就可以被当成一个表达式使用。

3.2 表达式

在Rust Reference中有这样一句话：

Rust is primarily an expression language.

Rust基本上就是一个表达式语言。“表达式”在Rust程序中占据着重要位置，表达式的功能非常强大。Rust中的表达式语法具有非常好的“一致性”，每种表达式都可以嵌入到另外一种表达式中，组成更强大的表达式。

Rust的表达式包括字面量表达式、方法调用表达式、数组表达式、索引表达式、单目运算符表达式、双目运算符表达式等。Rust表达式又可以分为“左值”（lvalue）和“右值”（rvalue）两类。所谓左值，意思是这个表达式可以表达一个内存地址。因此，它们可以放到赋值运算符左边使用。其他的都是右值。

3.2.1 运算表达式

Rust的算术运算符包括：加（+）、减（-）、乘（*）、除（/）、求余（%），示例如下：

```
fn main() {
    let x = 100;
    let y = 10;
    println!("{} {} {} {} {}", x + y, x - y, x * y, x / y, x % y);
}
```

在上面例子中，x+y、x-y这些都是算术运算表达式，它们都有自己的值和类型。常见的整数、浮点数类型都支持这几种表达式。它们还可以被重载，让自定义的类型也支持这几种表达式。运算符重载相关的内容会在第26章介绍标准库的时候会详细说明。

Rust的比较运算符包括：等于（==）、不等于（！=）、小于（<）、大于（>）、小于等于（<=）、大于等于（>=）。比较运算符的两边必须是同类型的，并满足PartialEq约束。比较表达式的类型是

bool。另外，Rust禁止连续比较，示例如下：

```
fn f(a: bool, b: bool, c: bool) -> bool {
    a == b == c
}
```

编译时，编译器提示“连续比较运算符必须加上括号”：

```
$ rustc --crate-type rlib test.rs
error: chained comparison operators require parentheses
 --> test.rs:2:7
  |
2 |     a == b == c
  |       ^^^^^^^^^

error: aborting due to previous error
```

这也是故意设计的，避免不同知识背景的用户对这段代码有不同的理解。

Rust的位运算符具体见表3-1。

表　3-1

运算符	作用
!	按位取反（注意不是～符号）
&	按位与
|	按位或
^	按位异或
<<	左移
>>	右移

示例如下：

```
fn main() {
    let num1 : u8 = 0b_1010_1010;
```

```
    let num2 : u8 = 0b_1111_0000;

    println!("{:08b}", !num1);
    println!("{:08b}", num1 & num2);
    println!("{:08b}", num1 | num2);
    println!("{:08b}", num1 ^ num2);
    println!("{:08b}", num1 << 4);
    println!("{:08b}", num1 >> 4);
}
```

执行结果为：

```
$ ./test
01010101
10100000
11111010
01011010
10100000
00001010
```

Rust的逻辑运算符具体见表3-2。

表　3-2

运算符	作用
&&	逻辑与
||	逻辑或
!	逻辑取反

取反运算符既支持“逻辑取反”也支持“按位取反”，它们是同一个运算符，根据类型决定执行哪个操作。如果被操作数是bool类型，那么就是逻辑取反；如果被操作数是其他数字类型，那么就是按位取反。

bool类型既支持“逻辑与”、“逻辑或”，也支持“按位与”、“按位或”。它们的区别在于，“逻辑与”、“逻辑或”具备“短路”功能。示例如下：

```
fn f1() -> bool {
    println!("Call f1");
    true
}
```

```
fn f2() -> bool {
    println!("Call f2");
    false
}

fn main() {
    println!("Bit and: {}\n", f2() & f1());
    println!("Logic and: {}\n", f2() && f1());

    println!("Bit or: {}\n", f1() | f2());
    println!("Logic or: {}\n", f1() || f2());
}
```

执行结果为：

```
$ ./test
Call f2
Call f1
Bit and: false

Call f2
Logic and: false

Call f1
Call f2
Bit or: true

Call f1
Logic or: true
```

可以看到，所谓短路的意思是：

·对于表达式A&&B，如果A的值是false，那么B就不会执行求值，直接返回false。

·对于表达式A||B，如果A的值是true，那么B就不会执行求值，直接返回true。

而“按位与”、“按位或”在任何时候都会先执行左边的表达式，再执行右边的表达式，不会省略。

另外需要提示的一点是，Rust里面的运算符优先级与C语言里面的运算符优先级设置是不一样的，有些细微的差别。不过这并不是很重要。不论在哪种编程语言中，我们都建议，如果碰到复杂一点的表达式，尽量用小括号明确表达计算顺序，避免依赖语言默认的运算符优先

级。因为不同知识背景的程序员对运算符优先级顺序的记忆是不同的。

3.2.2 赋值表达式

一个左值表达式、赋值运算符（=）和右值表达式，可以构成一个赋值表达式。示例如下：

```
// 声明局部变量,带 mut 修饰
let mut x : i32 = 1;

// x 是 mut 绑定,所以可以为它重新赋值
x = 2;
```

上例中，x=2是一个赋值表达式，它末尾加上分号，才能组成一个语句。赋值表达式具有“副作用”：当它执行的时候，会把右边表达式的值“复制或者移动”（copy or move）到左边的表达式中。关于复制和移动的语义区别，请参见第11章的内容。赋值号左右两边表达式的类型必须一致，否则是编译错误。

赋值表达式也有对应的类型和值。这里不是说赋值表达式左操作数或右操作数的类型和值，而是说整个表达式的类型和值。Rust规定，赋值表达式的类型为unit，即空的tuple（）。示例如下：

```
fn main() {
    let x = 1;
    let mut y = 2;
    // 注意这里专门用括号括起来了
    let z = (y = x);
    println!("{:?}", z);
}
```

编译，执行，结果为：（）。

Rust这么设计是有原因的，比如说可以防止连续赋值。如果你有x：i32、y：i32以及z：i32，那么表达式z=y=x会发生编译错误。因为变量z的类型是i32但是却用（）对它初始化了，编译器是不允许通过的。

C语言允许连续赋值，但这个设计没有带来任何性能提升，反而在某些场景下给用户带来了代码不够清晰直观的麻烦。举个例子：

```
#include <stdio.h>

int main() {
    int x = 300;
    char y;
    int z;
    z = y = x;
    printf("%d %d %d", x, y, z);
}
```

在这种情况下，如果变量x、y、z的类型不一样，而且在赋值的时候可能发生截断，那么用户很难一眼看出最终变量z的值是与x相同，还是与y相同。

这个设计同样可以防止把==写成=的错误。比如，Rust规定，在if表达式中，它的条件表达式类型必须是bool类型，所以if x=y{}这样的代码是无论如何都编译不过的，哪怕x和y的类型都是bool也不行。赋值表达式的类型永远是（），它无法用于if条件表达式中。

Rust也支持组合赋值表达式，+、-、*、/、%、&、|、^、<<、>>这几个运算符可以和赋值运算符组合成赋值表达式。示例如下：

```
fn main() {
    let x = 2;
    let mut y = 4;
    y += x;
    y *= x;
    println!("{} {}", x, y);
}
```

LEFT OP=RIGHT这种写法，含义等同于LEFT=LEFT OP RIGHT。所以，y+=x的意义相当于y=y+x，依此类推。

Rust不支持++、--运算符，请使用+=1、-=1替代。

3.2.3 语句块表达式

在Rust中，语句块也可以是表达式的一部分。语句和表达式的区分方式是后面带不带分号（；）。如果带了分号，意味着这是一条语句，它的类型是（）；如果不带分号，它的类型就是表达式的类型。示例如下：

```
// 语句块可以是表达式,注意后面有分号结尾,x的类型是()
let x : () = { println!("Hello."); };

// Rust将按顺序执行语句块内的语句,并将最后一个表达式类型返回,y的类型是 i32
let y : i32 = { println!("Hello."); 5 };
```

同理，在函数中，我们也可以利用这样的特点来写返回值：

```
fn my_func() -> i32 {
    // ... blablabla 各种语句
    100
}
```

注意，最后一条表达式没有加分号，因此整个语句块的类型就变成了i32，刚好与函数的返回类型匹配。这种写法与return 100；语句的效果是一样的，相较于return语句来说没有什么区别，但是更加简洁。特别是用在后面讲到的闭包closure中，这样写就方便轻量得多。

3.3 if-else

Rust中if-else表达式的作用是实现条件分支。if-else表达式的构成方式为：以if关键字开头，后面跟上条件表达式，后续是结果语句块，最后是可选的else块。条件表达式的类型必须是bool。

示例如下：

```
fn func(i : i32) -> bool {

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
}
```

在if语句中，后续的结果语句块要求一定要用大括号包起来，不能省略，以便明确指出该if语句块的作用范围。这个规定是为了避免“悬空else”导致的bug。比如下面这段C代码：

```
if (condition1)
    if (condition2) {

    }
    else {

    }
```

请问，这个else分支是与第一个if相匹配的，还是与第二个if相匹配的呢？从可读性上来说，答案是不够明显，容易出bug。规定if和else后面必须有大括号，可读性会好很多。

相反，条件表达式并未强制要求用小括号包起来；如果加上小括号，编译器反而会认为这是一个多余的小括号，给出警告。

更重要的是，if-else结构还可以当表达式使用，比如：

```
let x : i32 = if condition { 1 } else { 10 };
//------------------        ^ -------- ^
//------------------ 这两个地方不要加分号
```

在这里，if-else结构成了表达式的一部分。在if和else后面的大括号内，最后一条表达式不要加分号，这样一来，这两个语句块的类型就都是i32，与赋值运算符左边的类型刚好匹配。所以，在Rust中，没有必要专门设计像C/C++那样的三元运算符（？：）语法，因为通过现有的设计可以轻松实现同样的功能。而且笔者认为这样的语法一致性、扩展性、可读性更好。

如果使用if-else作为表达式，那么一定要注意，if分支和else分支的类型必须一致，否则就不能构成一个合法的表达式，会出现编译错误。如果else分支省略掉了，那么编译器会认为else分支的类型默认为（）。所以，下面这种写法一定会出现编译错误：

```
fn invalid_expr(cond: bool) -> i32 {
    if cond {
        42
    }
}
```

编译器提示信息是：

```
1= note: expected type `()`
            found type `i32`
```

这看起来像是类型不匹配的错误，实际上是漏写了else分支造成的。如果此处编译器不报错，放任程序编译通过，那么在执行到else分支的时候，就只能返回一个未初始化的值，这在Rust中是不允许的。

3.3.1 loop

在Rust中，使用loop表示一个无限死循环。示例如下：

```
fn main() {
    let mut count = 0u32;
    println!("Let's count until infinity!");
```

```
    // 无限循环
    loop {
        count += 1;
        if count == 3 {
            println!("three");

            // 不再继续执行后面的代码,跳转到loop开头继续循环
            continue;
        }

        println!("{}", count);
        if count == 5 {
            println!("OK, that's enough");

            // 跳出循环
            break;
        }
    }
}
```

其中，我们可以使用continue和break控制执行流程。continue；语句表示本次循环内，后面的语句不再执行，直接进入下一轮循环。break；语句表示跳出循环，不再继续。

另外，break语句和continue语句还可以在多重循环中选择跳出到哪一层的循环。

```
fn main() {
    // A counter variable
    let mut m = 1;
    let n = 1;

    'a: loop {
        if m < 100 {
            m += 1;
        } else {
            'b: loop {
                if m + n > 50 {
                    println!("break");
                    break 'a;
                } else {
                    continue 'a;
                }
            }
        }
    }
}
```

我们可以在loop while for循环前面加上“生命周期标识符”。该标识符以单引号开头，在内部的循环中可以使用break语句选择跳出到哪一层。

与if结构一样，loop结构也可以作为表达式的一部分。

```
fn main() {
    let v = loop {
        break 10;
    };
    println!("{}", v);
}
```

在loop内部break的后面可以跟一个表达式，这个表达式就是最终的loop表达式的值。如果一个loop永远不返回，那么它的类型就是“发散类型”。示例如下：

```
fn main() {
    let v = loop {};
    println!("{}", v);
}
```

编译器可以判断出v的类型是发散类型，而后面的打印语句是永远不会执行的死代码。

3.3.2 while

while语句是带条件判断的循环语句。其语法是while关键字后跟条件判断语句，最后是结果语句块。如果条件满足，则持续循环执行结果语句块。示例如下：

```
fn main() {
    // A counter variable
    let mut n = 1;
    // Loop while `n` is less than 101
    while n < 101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
        // Increment counter
        n += 1;
    }
```

```
}
```

同理，while语句中也可以使用continue和break来控制循环流程。

看到这里，读者可能会产生疑惑：loop{}和while true{}循环有什么区别，为什么Rust专门设计了一个死循环，loop语句难道不是完全多余的吗？

实际上不是。主要原因在于，相比于其他的许多语言，Rust语言要做更多的静态分析。loop和while true语句在运行时没有什么区别，它们主要是会影响编译器内部的静态分析结果。比如：

```
let x;
loop { x = 1; break; }
println!("{}", x)
```

以上语句在Rust中完全合理。因为编译器可以通过流程分析推理出x=1；必然在println！之前执行过，因此打印变量x的值是完全合理的。而下面的代码是编译不过的：

```
let x;
while true { x = 1; break; }
println!("{}", x);
```

因为编译器会觉得while语句的执行跟条件表达式在运行阶段的值有关，因此它不确定x是否一定会初始化，于是它决定给出一个错误：use of possibly uninitialized variable，也就是说变量x可能没有初始化。

3.3.3 for循环

Rust中的for循环实际上是许多其他语言中的for-each循环。Rust中没有类似C/C++的三段式for循环语句。举例如下：

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

    for i in array {
        println!("The number is {}", i);
```

```
    }
}
```

for循环的主要用处是利用迭代器对包含同样类型的多个元素的容器执行遍历，如数组、链表、HashMap、HashSet等。在Rust中，我们可以轻松地定制自己的容器和迭代器，因此也很容易使for循环也支持自定义类型。

for循环内部也可以使用continue和break控制执行流程。

有关for循环的原理以及迭代器相关内容，参见第24章。

第4章　函数

4.1　简介

Rust的函数使用关键字fn开头。函数可以有一系列的输入参数，还有一个返回类型。函数体包含一系列的语句（或者表达式）。函数返回可以使用return语句，也可以使用表达式。Rust编写的可执行程序的入口就是fn main（）函数。以下是一个函数的示例：

```
fn add1(t : (i32,i32)) -> i32 {
    t.0 + t.1
}
```

这个函数有一个输入参数，其类型是tuple（i32，i32）。它有一个返回值，返回类型是i32。函数的参数列表与let语句一样，也是一个“模式解构”。模式结构的详细解释请参考第7章。上述函数也可以写成下面这样：

```
fn add2((x,y) : (i32,i32)) -> i32 {
    x + y
}
```

函数体内部是一个表达式，这个表达式的值就是函数的返回值。也可以写return x+y；这样的语句作为返回值，效果是一样的。

函数也可以不写返回类型，在这种情况下，编译器会认为返回类型是unit（）。此处和表达式的规定是一致的。

函数可以当成头等公民（first class value）被复制到一个值中，这个值可以像函数一样被调用。示例如下：

```
fn main() {
    let p = (1, 3);

    // func 是一个局部变量
    let func = add2;
    // func 可以被当成普通函数一样被调用
```

```
    println!("evaluation output {}", func(p));
}
```

在Rust中，每一个函数都具有自己单独的类型，但是这个类型可以自动转换到fn类型。示例如下：

```
fn main() {
    // 先让 func 指向 add1
    let mut func = add1;
    // 再重新赋值,让 func 指向 add2
    func = add2;
}
```

编译，会出现编译错误，如下：

```
error[E0308]: mismatched types
  --> test.rs:11:12
   |
11 |     func = add2;
   |            ^^^^ expected fn item, found a different fn item
   |
   = note: expected type `fn((i32, i32)) -> i32 {add1}`
              found type `fn((i32, i32)) -> i32 {add2}`
```

虽然add1和add2有同样的参数类型和同样的返回值类型，但它们是不同类型，所以这里报错了。修复方案是让func的类型为通用的fn类型即可：

```
// 写法一,用 as 类型转换
let mut func = add1 as fn((i32,i32))->i32;
// 写法二,用显式类型标记
let mut func : fn((i32,i32))->i32 = add1;
```

以上两种写法都能修复上面的编译错误。但是，我们不能在参数、返回值类型不同的情况下作类型转换，比如：

```
fn add3(x: i32, y: i32) -> i32 {
    x + y
}

fn main() {
    let mut func : fn((i32,i32))->i32 = add1;
    func = add2;
    func = add3;
```

```
}
```

这里再加了一个add3函数，它接受两个i32参数，这就跟add1和add2有了本质区别。add1和add2是一个参数，类型是tuple包含两个i32成员，而add3是两个i32参数。三者完全不一样，它们之间是无法进行类型转换的。

另外需要提示的就是，Rust的函数体内也允许定义其他item，包括静态变量、常量、函数、trait、类型、模块等。比如：

```
fn test_inner() {
    static INNER_STATIC: i64 = 42;

    // 函数内部定义的函数
    fn internal_incr(x: i64) -> i64 {
        x + 1
    }

    struct InnerTemp(i64);

    impl InnerTemp {
        fn incr(&mut self) {
            self.0 = internal_incr(self.0);
        }
    }

    // 函数体,执行语句
    let mut t = InnerTemp(INNER_STATIC);
    t.incr();
    println!("{}", t.0);
}
```

当你需要一些item仅在此函数内有用的时候，可以把它们直接定义到函数体内，以避免污染外部的命名空间。

4.2 发散函数

Rust支持一种特殊的发散函数（Diverging functions），它的返回类型是感叹号！。如果一个函数根本就不能正常返回，那么它可以这样写：

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

因为panic！会直接导致栈展开，所以这个函数调用后面的代码都不会继续执行，它的返回类型就是一个特殊的！符号，这种函数也叫作发散函数。发散类型的最大特点就是，它可以被转换为任意一个类型。比如：

```
let x : i32 = diverges();
let y : String = diverges();
```

我们为什么需要这样的一种返回类型呢？先看下面的例子：

```
let p = if x {
    panic!("error");
} else {
    100
};
```

上面这条语句中包含一个if-else分支结构的表达式。我们知道，对于分支结构的表达式，它的每条分支的类型必须一致。那么这条panic！宏应该生成一个什么类型呢？这就是！类型的作用了。因为它可以与任意类型相容，所以编译器的类型检查才能通过。

在Rust中，有以下这些情况永远不会返回，它们的类型就是！。

·panic！以及基于它实现的各种函数/宏，比如unimplemented！、unreachable！；

·死循环loop{}；

·进程退出函数std：：process：：exit以及类似的libc中的exec一类函数。

关于这个！类型，第8章在对类型系统做更深入分析的时候还会再提到。

4.3 main函数

在大部分主流操作系统上，一个进程开始执行的时候可以接受一系列的参数，退出的时候也可以返回一个错误码。许多编程语言也因此为main函数设计了参数和返回值类型。以C语言为例，主函数的原型一般允许定义成以下几种形式：

```
int main(void);
int main();

int main(int argc, char **argv);
int main(int argc, char *argv[]);
int main(int argc, char **argv, char **env);
```

Rust的设计稍微有点不一样，传递参数和返回状态码都由单独的API来完成，示例如下：

```
fn main() {
    for arg in std::env::args() {
        println!("Arg: {}", arg);
    }

    std::process::exit(0);
}
```

编译，执行并携带几个参数，可以看到：

```
$ test -opt1 opt2 -- opt3
Arg: test
Arg: -opt1
Arg: opt2
Arg: --
Arg: opt3
```

每个被空格分开的字符串都是一个参数。进程可以在任何时候调用exit（）直接退出，退出时候的错误码由exit（）函数的参数指定。

如果要读取环境变量，可以用std：：env：：var（）以及std：：env：：vars（）函数获得。示例如下：

```
fn main() {
    for arg in std::env::args() {
        match std::env::var(&arg) {
            Ok(val) => println!("{}: {:?}", &arg, val),
            Err(e) => println!("couldn't find environment {}, {}", &arg, e),
        }
    }

    println!("All environment varible count {}", std::env::vars().count());
}
```

var（）函数可以接受一个字符串类型参数，用于查找当前环境变量中是否存在这个名字的环境变量，vars（）函数不携带参数，可以返回所有的环境变量。

此前，Rust的main函数只支持无参数、无返回值类型的声明方式，即main函数的签名固定为：fn main（）->（）。但是，在引入了？符号作为错误处理语法糖之后，就变得不那么优雅了，因为？符号要求当前所在的函数返回的是Result类型，这样一来，问号就无法直接在main函数中使用了。为了解决这个问题，Rust设计组扩展了main函数的签名，使它变成了一个泛型函数，这个函数的返回类型可以是任何一个满足Terminationtrait约束的类型，其中（）、bool、Result都是满足这个约束的，它们都可以作为main函数的返回类型。关于这个问题，可以参见第33章。

4.4 const fn

函数可以用const关键字修饰，这样的函数可以在编译阶段被编译器执行，返回值也被视为编译期常量。示例如下：

```
#![feature(const_fn)]

const fn cube(num: usize) -> usize {
    num * num * num
}

fn main() {
    const DIM : usize = cube(2);
    const ARR : [i32; DIM] = [0; DIM];

    println!("{:?}", ARR);
```

cube函数接受数字参数，它会返回一个数字，而且这个返回值本身可以用于给一个const常量做初始化，const常量又可以当成一个常量数组的长度使用。

const函数是在编译阶段执行的，因此相比普通函数有许多限制，并非所有的表达式和语句都可以在其中使用。鉴于目前这个功能还没有完全稳定，const函数具体有哪些限制规则，本书就不在此问题上详细展开了，后面也许还会有调整。

4.5 函数递归调用

Rust允许函数递归调用。所谓递归调用，指的是函数直接或者间接调用自己。下面用经典的Fibonacci数列来举例：

```
fn fib(index: u32) -> u64 {
    if index == 1 || index == 2 {
        1
    } else {
        fib(index - 1) + fib(index - 2)
    }
}

fn main() {
    let f8 = fib(8);
    println!("{}", f8);
}
```

这个fib函数就是典型的递归调用函数，因为在它的函数体内又调用了它自己。

谈到递归调用，许多读者都会自然联想到“尾递归优化”这个概念。可惜的是，当前版本的Rust暂时还不支持尾递归优化，因此如果递归调用层次太多的话，是有可能撑爆栈空间的。不过这个问题已经在设计讨论之中，各位读者可以从最新的RFC项目中了解进度。

第5章　trait

Rust语言中的trait是非常重要的概念。在Rust中，trait这一个概念承担了多种职责。在中文里，trait可以翻译为“特征”“特点”“特性”等。由于这些词区分度并不明显，在本书中一律不翻译trait这个词，以避免歧义。

trait中可以包含：函数、常量、类型等。

5.1 成员方法

trait中可以定义函数。用例子来说明，我们定义如下的trait：

```
trait Shape {
    fn area(&self) -> f64;
}
```

上面这个trait包含了一个方法，这个方法只有一个参数，这个&self参数是什么意思呢？

所有的trait中都有一个隐藏的类型Self（大写S），代表当前这个实现了此trait的具体类型。trait中定义的函数，也可以称作关联函数（associated function）。函数的第一个参数如果是Self相关的类型，且命名为self（小写s），这个参数可以被称为“receiver”（接收者）。具有receiver参数的函数，我们称为“方法”（method），可以通过变量实例使用小数点来调用。没有receiver参数的函数，我们称为“静态函数”（static function），可以通过类型加双冒号：：的方式来调用。在Rust中，函数和方法没有本质区别。

Rust中Self（大写S）和self（小写s）都是关键字，大写S的是类型名，小写s的是变量名。请大家一定注意区分。self参数同样也可以指定类型，当然这个类型是有限制的，必须是包装在Self类型之上的类型。对于第一个self参数，常见的类型有self：Self、self：&Self、self：&mut Self等类型。对于以上这些类型，Rust提供了一种简化的写法，我们可以将参数简写为self、&self、&mut self。self参数只能用在第一个参数的位置。请注意“变量self”和“类型Self”的大小写不同。示例如下：

```
trait T {
    fn method1(self: Self);
    fn method2(self: &Self);
    fn method3(self: &mut Self);
}
// 上下两种写法是完全一样的
trait T {
    fn method1(self);
    fn method2(&self);
    fn method3(&mut self);
}
```

所以，回到开始定义的那个Shape trait，上面定义的这个area方法的参数的名字为self，它的类型是&Self类型。我们可以把上面这个方法的声明看成：

```
trait Shape {
    fn area(self: &Self) -> f64;
}
```

我们可以为某些具体类型实现（impl）这个trait。

假如我们有一个结构体类型Circle，它实现了这个trait，代码如下：

```
struct Circle {
    radius: f64,
}

impl Shape for Circle {
    // Self 类型就是 Circle
    // self 的类型是 &Self,即 &Circle
    fn area(&self) -> f64 {
        // 访问成员变量,需要用 self.radius
        std::f64::consts::PI * self.radius * self.radius
    }
}

fn main() {
    let c = Circle { radius : 2f64};
    // 第一个参数名字是 self,可以使用小数点语法调用
    println!("The area is {}", c.area());
}
```

在上面的例子中可以看到，如果有一个Circle类型的实例c，我们就可以用小数点调用函数，c.area（）。在方法内部，我们可以通过self.radius的方式访问类型的内部成员。

另外，针对一个类型，我们可以直接对它impl来增加成员方法，无须trait名字。比如：

```
impl Circle {
    fn get_radius(&self) -> f64 { self.radius }
}
```

我们可以把这段代码看作是为Circle类型impl了一个匿名的trait。用

这种方式定义的方法叫作这个类型的“内在方法”（inherent methods）。

trait中可以包含方法的默认实现。如果这个方法在trait中已经有了方法体，那么在针对具体类型实现的时候，就可以选择不用重写。当然，如果需要针对特殊类型作特殊处理，也可以选择重新实现来“override”默认的实现方式。比如，在标准库中，迭代器Iterator这个trait中就包含了十多个方法，但是，其中只有fn next（&mut self）->Option<Self：：Item>是没有默认实现的。其他的方法均有其默认实现，在实现迭代器的时候只需挑选需要重写的方法来实现即可。

self参数甚至可以是Box指针类型self：Box<Self>。另外，目前Rust设计组也在考虑让self变量的类型放得更宽，允许更多的自定义类型作为receiver，比如MyType<Self>。示例如下：

```
trait Shape {
    fn area(self: Box<Self>) -> f64;
}


struct Circle {
    radius: f64,
}

impl Shape for Circle {
    // Self 类型就是 Circle
    // self 的类型是 Box<Self>,即 Box<Circle>
    fn area(self : Box<Self>) -> f64 {
        // 访问成员变量,需要用 self.radius
        std::f64::consts::PI * self.radius * self.radius
    }
}

fn main() {
    let c = Circle { radius : 2f64};
    // 编译错误
    // c.area();

    let b = Box::new(Circle {radius : 4f64});
    // 编译正确
    b.area();
}
```

impl的对象甚至可以是trait。示例如下：

```
trait Shape {
    fn area(&self) -> f64;
}
trait Round {
```

```
    fn get_radius(&self) -> f64;
}

struct Circle {
    radius: f64,
}

impl Round for Circle {
    fn get_radius(&self) -> f64 { self.radius }
}

// 注意这里是 impl Trait for Trait
impl Shape for Round {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.get_radius() * self.get_radius()
    }
}


fn main() {
    let c = Circle { radius : 2f64};
    // 编译错误
    // c.area();

    let b = Box::new(Circle {radius : 4f64}) as Box<Round>;
    // 编译正确
    b.area();
}
```

注意这里的写法，impl Shape for Round和impl<T：Round>Shape for T是不一样的。在前一种写法中，self是&Round类型，它是一个trait object，是胖指针。而在后一种写法中，self是&T类型，是具体类型。前一种写法是为trait object增加一个成员方法，而后一种写法是为所有的满足T：Round的具体类型增加一个成员方法。所以上面的示例中，我们只能构造一个trait object之后才能调用area（）成员方法。trait object和“泛型”之间的区别请参考本书第三部分。

题外话，impl Shape for Round这种写法确实是很让初学者纠结的，Round既是trait又是type。在将来，trait object的语法会被要求加上dyn关键字，所以在Rust 2018 edition以后应该写成impl Shape for dyn Round才合理。关于trait object的内容，请参考本书第三部分第23章。

5.2 静态方法

没有receiver参数的方法（第一个参数不是self参数的方法）称作“静态方法”。静态方法可以通过Type：：FunctionName（）的方式调用。需要注意的是，即便我们的第一个参数是Self相关类型，只要变量名字不是self，就不能使用小数点的语法调用函数。

```
struct T(i32);

impl T {
    // 这是一个静态方法
    fn func(this: &Self) {
        println!{"value {}", this.0};
    }
}

fn main() {
    let x = T(42);
    // x.func(); 小数点方式调用是不合法的
    T::func(&x);
}
```

在标准库中就有一些这样的例子。Box的一系列方法Box：：into_raw（b：Self）Box：：leak（b：Self），以及Rc的一系列方法Rc：：try_unwrap（this：Self）Rc：：downgrade（this：&Self），都是这种情况。它们的receiver不是self关键字，这样设计的目的是强制用户用Rc：：downgrade（&obj）的形式调用，而禁止obj.downgrade（）形式的调用。这样源码表达出来的意思更清晰，不会因为Rc<T>里面的成员方法和T里面的成员方法重名而造成误解问题（这又涉及Deref trait的内容，读者可以把第16章读完再回看这一段）。

trait中也可以定义静态函数。下面以标准库中的std：：default：：Default trait为例，介绍静态函数的相关用法：

```
pub trait Default {
    fn default() -> Self;
}
```

上面这个trait中包含了一个default（）函数，它是一个无参数的函数，返回的类型是实现该trait的具体类型。Rust中没有“构造函数”的概

念。Default trait实际上可以看作一个针对无参数构造函数的统一抽象。

比如在标准库中，Vec：：default（）就是一个普通的静态函数。

```
// 这里用到了“泛型”,请参阅第21章
impl<T> Default for Vec<T> {
    fn default() -> Vec<T> {
        Vec::new()
    }
}
```

跟C++相比，在Rust中，定义静态函数没必要使用static关键字，因为它把self参数显式在参数列表中列出来了。作为对比，C++里面成员方法默认可以访问this指针，因此它需要用static关键字来标记静态方法。Rust不采取这个设计，主要原因是self参数的类型变化太多，不同写法语义差别很大，选择显式声明self参数更方便指定它的类型。

5.3 扩展方法

我们还可以利用trait给其他的类型添加成员方法，哪怕这个类型不是我们自己写的。比如，我们可以为内置类型i32添加一个方法：

```
trait Double {
    fn double(&self) -> Self;
}

impl Double for i32 {
    fn double(&self) -> i32 { *self * 2 }
}

fn main() {
    // 可以像成员方法一样调用
    let x : i32 = 10.double();
    println!("{}", x);
}
```

这个功能就像C#里面的“扩展方法”一样。哪怕这个类型不是在当前的项目中声明的，我们依然可以为它增加一些成员方法。但我们也不是随随便便就可以这么做的，Rust对此有一个规定。

在声明trait和impl trait的时候，Rust规定了一个Coherence Rule（一致性规则）或称为Orphan Rule（孤儿规则）：impl块要么与trait的声明在同一个的crate中，要么与类型的声明在同一个crate中。

也就是说，如果trait来自于外部crate，而且类型也来自于外部crate，编译器不允许你为这个类型impl这个trait。它们之中必须至少有一个是在当前crate中定义的。因为在其他的crate中，一个类型没有实现一个trait，很可能是有意的设计。如果我们在使用其他的crate的时候，强行把它们“拉郎配”，是会制造出bug的。比如说，我们写了一个程序，引用了外部库lib1和lib2，lib1中声明了一个trait T，lib2中声明了一个struct S，我们不能在自己的程序中针对S实现T。这也意味着，上游开发者在给别人写库的时候，尤其要注意，一些比较常见的标准库中的trait，如Display Debug ToString Default等，应该尽可能地提供好。否则，使用这个库的下游开发者是没办法帮我们把这些trait实现的。

同理，如果是匿名impl，那么这个impl块必须与类型本身存在于同一个crate中。

更多关于“一致性规则”的解释，可以参见编译器的详细错误说明：

```
rustc --explain E0117
rustc --explain E0210
```

当类型和trait涉及泛型参数的时候，一致性规则实际上是很复杂的，用户如果需要了解所有的细节，还需要参考对应的RFC文档。

许多初学者会用自带GC的语言中的“Interface”、抽象基类来理解trait这个概念，但是实际上它们有很大的不同。

Rust是一种用户可以对内存有精确控制能力的强类型语言。我们可以自由指定一个变量是在栈里面，还是在堆里面，变量和指针也是不同的类型。类型是有大小（Size）的。有些类型的大小是在编译阶段可以确定的，有些类型的大小是编译阶段无法确定的。目前版本的Rust规定，在函数参数传递、返回值传递等地方，都要求这个类型在编译阶段有确定的大小。否则，编译器就不知道该如何生成代码了。

而trait本身既不是具体类型，也不是指针类型，它只是定义了针对类型的、抽象的“约束”。不同的类型可以实现同一个trait，满足同一个trait的类型可能具有不同的大小。因此，trait在编译阶段没有固定大小，目前我们不能直接使用trait作为实例变量、参数、返回值。

有一些初学者特别喜欢写这样的代码：

```
let x: Shape = Circle::new(); // Shape 不能做局部变量的类型
fn use_shape(arg : Shape) {}  // Shape 不能直接做参数的类型
fn ret_shape() -> Shape {}    // Shape 不能直接做返回值的类型
```

这样的写法是错误的。请一定要记住，trait的大小在编译阶段是不固定的。那怎样写才是对的呢？后面我们讲到泛型的时候再说。

5.4 完整函数调用语法

Fully Qualified Syntax提供一种无歧义的函数调用语法，允许程序员精确地指定想调用的是那个函数。以前也叫UFCS（universal function call syntax），也就是所谓的“通用函数调用语法”。这个语法可以允许使用类似的写法精确调用任何方法，包括成员方法和静态方法。其他一切函数调用语法都是它的某种简略形式。它的具体写法为<T as TraitName>：：item。示例如下：

```
trait Cook {
    fn start(&self);
}

trait Wash {
    fn start(&self);
}

struct Chef;

impl Cook for Chef {
    fn start(&self) { println!("Cook::start");}
}

impl Wash for Chef {
    fn start(&self) { println!("Wash::start");}
}

fn main() {
    let me = Chef;
    me.start();
}
```

我们定义了两个trait，它们的start（）函数有同样方法签名。

如果一个类型同时实现了这两个trait，那么如果我们使用variable.start（）这样的语法执行方法调用的话，就会出现歧义，编译器不知道你具体想调用哪个方法，编译错误信息为“multiple applicable items in scope”。

这时候，我们就有必要使用完整的函数调用语法来进行方法调用，只有这样写，才能清晰明白且无歧义地表达清楚期望调用的是哪个函数：

```
fn main() {
    let me = Chef;
// 函数名字使用更完整的path来指定,同时,self参数需要显式传递
    <Cook>::start(&me);
    <Chef as Wash>::start(&me);
}
```

由此我们也可以看到，所谓的“成员方法”也没什么特殊之处，它跟普通的静态方法的唯一区别是，第一个参数是self，而这个self只是一个普通的函数参数而已。只不过这种成员方法也可以通过变量加小数点的方式调用。变量加小数点的调用方式在大部分情况下看起来更简单更美观，完全可以视为一种语法糖。

需要注意的是，通过小数点语法调用方法调用，有一个“隐藏着”的“取引用”步骤。虽然我们看起来源代码长的是这个样子me.start（），但是大家心里要清楚，真正传递给start（）方法的参数是&me而不是me，这一步是编译器自动帮我们做的。不论这个方法接受的self参数究竟是Self、&Self还是&mut Self，最终在源码上，我们都是统一的写法：variable.method（）。而如果用UFCS语法来调用这个方法，我们就不能让编译器帮我们自动取引用了，必须手动写清楚。

下面用一个示例演示一下成员方法和普通函数其实没什么本质区别。

```
struct T(usize);

impl T {
    fn get1(&self) -> usize {self.0}

    fn get2(&self) -> usize {self.0}
}

fn get3(t: &T) -> usize { t.0 }

fn check_type( _ : fn(&T)->usize ) {}

fn main() {
    check_type(T::get1);
    check_type(T::get2);
    check_type(get3);
}
```

可以看到，get1、get2和get3都可以自动转成fn（&T）→usize类型。

5.5 trait约束和继承

Rust的trait的另外一个大用处是，作为泛型约束使用。关于泛型，本书第三部分还会详细解释。下面用一个简单示例演示一下trait如何作为泛型约束使用：

```
use std::fmt::Debug;

fn my_print<T : Debug>(x: T) {
    println!("The value is {:?}.", x);
}

fn main() {
    my_print("China");
    my_print(41_i32);
    my_print(true);
    my_print(['a', 'b', 'c'])
}
```

上面这段代码中，my_print函数引入了一个泛型参数T，所以它的参数不是一个具体类型，而是一组类型。冒号后面加trait名字，就是这个泛型参数的约束条件。它要求这个T类型实现Debug这个trait。这是因为我们在函数体内，用到了println！格式化打印，而且用了{：？}这样的格式控制符，它要求类型满足Debug的约束，否则编译不过。

在调用的时候，凡是满足Debug约束的类型都可以是这个函数的参数，所以我们可以看到以上四种调用都是可以编译通过的。假如我们自定义一个类型，而它没有实现Debug trait，我们就会发现，用这个类型作为my_print的参数的话，编译就会报错。

所以，泛型约束既是对实现部分的约束，也是对调用部分的约束。

泛型约束还有另外一种写法，即where子句。示例如下：

```
fn my_print<T>(x: T) where T: Debug {
    println!("The value is {:?}.", x);
}
```

对于这种简单的情况，两种写法都可以。但是在某些复杂的情况

下，泛型约束只有where子句可以表达，泛型参数后面直接加冒号的写法表达不出来，比如涉及关联类型的时候，请参见第21章。

trait允许继承。类似下面这样：

```
trait Base { ... }
trait Derived : Base { ... }
```

这表示Derived trait继承了Base trait。它表达的意思是，满足Derived的类型，必然也满足Base trait。所以，我们在针对一个具体类型impl Derived的时候，编译器也会要求我们同时impl Base。示例如下：

```
trait Base {}

trait Derived : Base {}

struct T;

impl Derived for T {}

fn main() {
}
```

编译，出现错误，提示信息为：

```
--> test.rs:7:6
  |
7 | impl Derived for T {}
  |      ^^^^^^^ the trait `Base` is not implemented for `T`
```

我们再加上一句

```
impl Base for T {}
```

编译器就不再报错了。

实际上，在编译器的眼中，trait Derived：Base{}等同于trait Derived where Self：Base{}。这两种写法没有本质上的区别，都是给Derived这个trait加了一个约束条件，即实现Derived trait的具体类型，也必须满足

Base trait的约束。

在标准库中，很多trait之间都有继承关系，比如：

```
trait Eq: PartialEq<Self> {}
trait Copy: Clone {}
trait Ord: Eq + PartialOrd<Self> {}
trait FnMut<Args>: FnOnce<Args> {}
trait Fn<Args>: FnMut<Args> {}
```

读完本书后，读者应该能够理解这些trait是用来做什么的，以及为什么这些trait之间会有这样的继承关系。

5.6 Derive

Rust里面为类型impl某些trait的时候，逻辑是非常机械化的。为许多类型重复而单调地impl某些trait，是非常枯燥的事情。为此，Rust提供了一个特殊的attribute，它可以帮我们自动impl某些trait。示例如下：

```
#[derive(Copy, Clone, Default, Debug, Hash, PartialEq, Eq, PartialOrd, Ord)]
struct Foo {
    data : i32
}
fn main() {
    let v1 = Foo { data : 0 };
    let v2 = v1;
    println!("{:?}", v2);
}
```

如上所示，它的语法是，在你希望impl trait的类型前面写#[derive（...）]，括号里面是你希望impl的trait的名字。这样写了之后，编译器就帮你自动加上了impl块，类似这样：

```
impl Copy for Foo { ... }
impl Clone for Foo { ... }
impl Default for Foo { ... }
impl Debug for Foo { ... }
impl Hash for Foo { ... }
impl PartialEq for Foo { ... }
......
```

这些trait都是标准库内部的较特殊的trait，它们可能包含有成员方法，但是成员方法的逻辑有一个简单而一致的“模板”可以使用，编译器就机械化地重复这个模板，帮我们实现这个默认逻辑。当然我们也可以手动实现。

目前，Rust支持的可以自动derive的trait有以下这些：

```
Debug   Clone   Copy   Hash   RustcEncodable   RustcDecodable   PartialEq   Eq
    ParialOrd   Ord   Default   FromPrimitive   Send   Sync
```

5.7 trait别名

跟type alias类似的，trait也可以起别名（trait alias）。假如在某些场景下，我们有一个比较复杂的trait：

```
pub trait Service {
    type Request;
    type Response;
    type Error;
    type Future: Future<Item=Self::Response, Error=Self::Error>;
    fn call(&self, req: Self::Request) -> Self::Future;
}
```

每次使用这个trait的时候都需要携带一堆的关联类型参数。为了避免这样的麻烦，在已经确定了关联类型的场景下，我们可以为它取一个别名，比如：

```
trait HttpService = Service<Request = http::Request,
        Response = http::Response,
        Error = http::Error>;
```

5.8 标准库中常见的trait简介

标准库中有很多很有用的trait，本节挑几个特别常见的给大家介绍一下。

5.8.1 Display和Debug

这两个trait在标准库中的定义是这样的：

```
// std::fmt::Display
pub trait Display {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error>;
}
// std::fmt::Debug
pub trait Debug {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error>;
}
```

它们的主要用处就是用在类似println！这样的地方：

```
use std::fmt::{Display, Formatter, Error};

#[derive(Debug)]
struct T {
    field1: i32,
    field2: i32,
}

impl Display for T {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error> {
        write!(f, "{{ field1:{}, field2:{} }}", self.field1, self.field2)
    }
}

fn main() {
    let var = T { field1: 1, field2: 2 };
    println!("{}", var);
    println!("{:?}", var);
    println!("{:#?}", var);
}
```

只有实现了Display trait的类型，才能用{}格式控制打印出来；只有实现了Debug trait的类型，才能用{：？}{：#？}格式控制打印出来。它

们之间更多的区别如下。

·Display假定了这个类型可以用utf-8格式的字符串表示，它是准备给最终用户看的，并不是所有类型都应该或者能够实现这个trait。这个trait的fmt应该如何格式化字符串，完全取决于程序员自己，编译器不提供自动derive的功能。

·标准库中还有一个常用trait叫作std：：string：：ToString，对于所有实现了Display trait的类型，都自动实现了这个ToString trait。它包含了一个方法to_string（&self）->String。任何一个实现了Display trait的类型，我们都可以对它调用to_string（）方法格式化出一个字符串。

·Debug则是主要为了调试使用，建议所有的作为API的“公开”类型都应该实现这个trait，以方便调试。它打印出来的字符串不是以“美观易读”为标准，编译器提供了自动derive的功能。

5.8.2 PartialOrd/Ord/PartialEq/Eq

在前文中讲解浮点类型的时候提到，因为NaN的存在，浮点数是不具备“total order（全序关系）”的。在这里，我们详细讨论一下什么是全序、什么是偏序。Rust标准库中有如下解释。

对于集合X中的元素a，b，c，

·如果a<b则一定有！（a>b）；反之，若a>b，则一定有！（a<b），称为反对称性。

·如果a<b且b<c则a<c，称为传递性。

·对于X中的所有元素，都存在a<b或a>b或者a==b，三者必居其一，称为完全性。

如果集合X中的元素只具备上述前两条特征，则称X是“偏序”。同时具备以上所有特征，则称X是“全序”。

从以上定义可以看出，浮点数不具备“全序”特征，因为浮点数中特殊的值NaN不满足完全性。这就导致了一个问题：浮点数无法排序。对

于任意一个不是NaN的数和NaN之间做比较，无法分出先后关系。示例如下：

```
fn main() {
    let nan = std::f32::NAN;
    let x = 1.0f32;
    println!("{}", nan < x);
    println!("{}", nan > x);
    println!("{}", nan == x);
}
```

以上不论是NaN<x，NaN>x还是NaN==x，结果都是false。这是IEEE754标准中规定的行为。

因此，Rust设计了两个trait来描述这样的状态：一个是std：：cmp：：PartialOrd，表示“偏序”，一个是std：：cmp：：Ord，表示“全序”。它们的对外接口是这样定义的：

```
pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs> {
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    fn lt(&self, other: &Rhs) -> bool { //... }
    fn le(&self, other: &Rhs) -> bool { //... }
    fn gt(&self, other: &Rhs) -> bool { //... }
    fn ge(&self, other: &Rhs) -> bool { //... }
}
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

从以上代码可以看出，partial_cmp函数的返回值类型是Option<Ordering>。只有Ord trait里面的cmp函数才能返回一个确定的Ordering。f32和f64类型都只实现了PartialOrd，而没有实现Ord。

因此，如果我们写出下面的代码，编译器是会报错的：

```
let int_vec = [1_i32, 2, 3];
let biggest_int = int_vec.iter().max();

let float_vec = [1.0_f32, 2.0, 3.0];
let biggest_float = float_vec.iter().max();
```

对整数i32类型的数组求最大值是没问题的，但是对浮点数类型的数组求最大值是不对的，编译错误为：

```
the trait 'core::cmp::Ord' is not implemented for the type 'f32'
```

笔者认为，这个设计是优点，而不是缺点，它让我们尽可能地在更早的阶段发现错误，而不是留到运行时再去debug。假如说编译器无法静态检查出这样的问题，那么就可能发生下面的情况，以Python为例：

```
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> v = [1.0, float("nan")]
>>> max(v)
1.0
>>> v = [float("nan"), 1.0]
>>> max(v)
nan
```

上面这个示例意味着，如果数组v中有NaN，对它求最大值，跟数组内部元素的排列顺序有关。

Rust中的PartialOrd trait实际上就是C++20中即将加入的three-way comparison运算符<=>。

同理，PartialEq和Eq两个trait也就可以理解了，它们的作用是比较相等关系，与排序关系非常类似。

5.8.3 Sized

Sized trait是Rust中一个非常重要的trait，它的定义如下：

```
#[lang = "sized"]
#[rustc_on_unimplemented = "`{Self}` does not have a constant size known at compile·
#[fundamental] // for Default, for example, which requires that `[T]: !Default` be e
pub trait Sized {
    // Empty.
}
```

这个trait定义在std：：marker模块中，它没有任何的成员方法。它有#[lang="sized"]属性，说明它与普通trait不同，编译器对它有特殊的处理。用户也不能针对自己的类型impl这个trait。一个类型是否满足Sized

约束是完全由编译器推导的，用户无权指定。

我们知道，在C/C++这一类的语言中，大部分变量、参数、返回值都应该是编译阶段固定大小的。在Rust中，但凡编译阶段能确定大小的类型，都满足Sized约束。那还有什么类型是不满足Sized约束的呢？比如C语言里的不定长数组（Variable-length Array）。不定长数组的长度在编译阶段是未知的，是在执行阶段才确定下来的。Rust里面也有类似的类型[T]。在Rust中VLA类型已经通过了RFC设计，只是暂时还没有实现而已。不定长类型在使用的时候有一些限制，比如不能用它作为函数的返回类型，而必须将这个类型藏到指针背后才可以。但它作为一个类型，依然是有意义的，我们可以为它添加成员方法，用它实例化泛型参数，等等。

Rust中对于动态大小类型专门有一个名词Dynamic Sized Type。我们后面将会看到的[T]，str以及dyn Trait都是DST。

5.8.4 Default

Rust里面并没有C++里面的“构造函数”的概念。大家可以看到，它只提供了类似C语言的各种复合类型各自的初始化语法。主要原因在于，相比普通函数，构造函数本身并没有提供什么额外的抽象能力。所以Rust里面推荐使用普通的静态函数作为类型的“构造器”。比如，常见的标准库中提供的字符串类型String，它包含的可以构造新的String的方法不完全列举都有这么多：

```
fn new() -> String
fn with_capacity(capacity: usize) -> String
fn from_utf8(vec: Vec<u8>) -> Result<String, FromUtf8Error>
fn from_utf8_lossy<'a>(v: &'a [u8]) -> Cow<'a, str>
fn from_utf16(v: &[u16]) -> Result<String, FromUtf16Error>
fn from_utf16_lossy(v: &[u16]) -> String
unsafe fn from_raw_parts(buf: *mut u8, length: usize, capacity: usize) -> String
unsafe fn from_utf8_unchecked(bytes: Vec<u8>) -> String
```

这还不算Default：：default（）、From：：from（s：&’a str）、FromIte-r-a-t-or：：from_iter<I：IntoIterator<Item=char>>（iter：I）、Iter-a-t-or：：collect等相对复杂的构造方法。这些方法接受的参数各异，错误处理方式也各异，强行将它们统一到同名字的构造函数重载中不是什么好主意（况且Rust坚决反对ad hoc式的函数重载）。

不过，对于那种无参数、无错误处理的简单情况，标准库中提供了Default trait来做这个统一抽象。这个trait的签名如下：

```
trait Default {
    fn default() -> Self;
}
```

它只包含一个“静态函数”default（）返回Self类型。标准库中很多类型都实现了这个trait，它相当于提供了一个类型的默认值。

在Rust中，单词new并不是一个关键字。所以我们可以看到，很多类型中都使用了new作为函数名，用于命名那种最常用的创建新对象的情况。因为这些new函数差别甚大，所以并没有一个trait来对这些new函数做一个统一抽象。

5.9 总结

本章对trait这个概念做了基本的介绍。除了上面介绍的之外，trait还有许多用处：

·trait本身可以携带泛型参数；

·trait可以用在泛型参数的约束中；

·trait可以为一组类型impl，也可以单独为某一个具体类型impl，而且它们可以同时存在；

·trait可以为某个trait impl，而不是为某个具体类型impl；

·trait可以包含关联类型，而且还可以包含类型构造器，实现高阶类型的某些功能；

·trait可以实现泛型代码的静态分派，也可以通过trait object实现动态分派；

·trait可以不包含任何方法，用于给类型做标签（marker），以此来描述类型的一些重要特性；

·trait可以包含常量。

trait这个概念在Rust语言中扮演了非常重要的角色，承担了各种各样的功能，在写代码的时候会经常用到。本章还远没有把trait相关的知识讲解完整，更多关于trait的内容，请参阅本书后文中与泛型、trait object线程安全有关的章节。

第6章　数组和字符串

6.1　数组

数组是一个容器，它在一块连续空间内存中，存储了一系列的同样类型的数据。数组中元素的占用空间大小必须是编译期确定的。数组本身所容纳的元素个数也必须是编译期确定的，执行阶段不可变。如果需要使用变长的容器，可以使用标准库中的Vec/LinkedList等。数组类型的表示方式为[T；n]。其中T代表元素类型；n代表元素个数；它必须是编译期常量整数；中间用分号隔开。下面看一个基本的示例：

```
fn main() {
    // 定长数组
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // 所有的元素,如果初始化为同样的数据,可以使用如下语法
    let ys: [i32; 500] = [0; 500];
}
```

在Rust中，对于两个数组类型，只有元素类型和元素个数都完全相同，这两个数组才是同类型的。数组与指针之间不能隐式转换。同类型的数组之间可以互相赋值。示例如下：

```
fn main() {

    let mut xs: [i32; 5] = [1, 2, 3, 4, 5];
    let     ys: [i32; 5] = [6, 7, 8, 9, 10];
    xs = ys;
    println!("new array {:?}", xs);
}
```

把数组xs作为参数传给一个函数，这个数组并不会退化成一个指针。而是会将这个数组完整复制进这个函数。函数体内对数组的改动不会影响到外面的数组。

对数组内部元素的访问，可以使用中括号索引的方式。Rust支持usize类型的索引的数组，索引从0开始计数。

```
fn main() {
    let v : [i32; 5] = [1,2,3,4,5];
    let x = v[0] + v[1];        // 把第一个元素和第二个元素的值相加
    println!("sum is {}", x);
}
```

6.1.1 内置方法

与其他所有类型一样，Rust的数组类型拥有一些内置方法，可以很方便地完成一些任务。比如，我们可以直接实现数组的比较操作，只要它包含的元素是可以比较的：

```
fn main() {
    let v1 = [1, 2, 3];
    let v2 = [1, 2, 4];
    println!("{:?}", v1 < v2 );
}
```

我们也可以对数组执行遍历操作，如：

```
fn main() {

    let v = [0_i32; 10];

    for i in &v {
        println!("{:?}", i);
    }
}
```

在目前的标准库中，数组本身没有实现IntoIterator trait，但是数组切片是实现了的。所以我们可以直接在for in循环中使用数组切片，而不能直接使用数组本身。更详细的内容请参阅后文中关于迭代器的解释。

6.1.2 多维数组

既然[T；n]是一个合法的类型，那么它的元素T当然也可以是数组类型，因此[[T；m]；n]类型自然也是合法类型。示例如下：

```
fn main() {
    let v : [[i32; 2]; 3] = [[0, 0], [0, 0], [0, 0]];
```

```
    for i in &v {
        println!("{:?}", i);
    }
}
```

6.1.3 数组切片

对数组取借用borrow操作，可以生成一个“数组切片”（Slice）。数组切片对数组没有“所有权”，我们可以把数组切片看作专门用于指向数组的指针，是对数组的另外一个“视图”。比如，我们有一个数组[T；n]，它的借用指针的类型就是&[T；n]。它可以通过编译器内部魔法转换为数组切片类型&[T]。数组切片实质上还是指针，它不过是在类型系统中丢弃了编译阶段定长数组类型的长度信息，而将此长度信息存储为运行期的值。示例如下：

```
fn main() {
    fn mut_array(a : &mut [i32]) {
        a[2] = 5;
    }

    println!("size of &[i32; 3] : {:?}", std::mem::size_of::<&[i32; 3]>());
    println!("size of &[i32]    : {:?}", std::mem::size_of::<&[i32]>());

    let mut v :  [i32; 3] = [1,2,3];
    {
        let s : &mut [i32; 3] = &mut v;
        mut_array(s);
    }
    println!("{:?}", v);
}
```

变量v是[i32；3]类型；变量s是&mut[i32；3]类型，占用的空间大小与指针相同。它可以自动转换为&mut[i32]数组切片类型传入函数mut_array，占用的空间大小等于两个指针的空间大小。通过这个指针，在函数内部，修改了外部的数组v的值。

6.1.4 DST和胖指针

从前面的示例中可以看到，数组切片是指向一个数组的指针，而它比指针又多了一点东西——它不止包含有一个指向数组的指针，切片本身还含带长度信息。

Slice与普通的指针是不同的，它有一个非常形象的名字：胖指针（fat pointer）。与这个概念相对应的概念是“动态大小类型”（Dynamic Sized Type，DST）。所谓的DST指的是编译阶段无法确定占用空间大小的类型。为了安全性，指向DST的指针一般是胖指针。

比如：对于不定长数组类型[T]，有对应的胖指针&[T]类型；对于不定长字符串str类型，有对应的胖指针&str类型；以及在后文中会出现的Trait Object；等等。

由于不定长数组类型[T]在编译阶段是无法判断该类型占用空间的大小的，目前我们不能在栈上声明一个不定长大小数组的变量实例，也不能用它作为函数的参数、返回值。但是，指向不定长数组的胖指针的大小是确定的，&[T]类型可以用做变量实例、函数参数、返回值。

通过前面的示例我们可以看到，&[T]类型占用了两个指针大小的内存空间。我们可以利用unsafe代码把这个胖指针内部的数据打印出来看看：

```
fn raw_slice(arr: &[i32]) {
    unsafe {
        let (val1, val2) : (usize, usize) = std::mem::transmute(arr);
        println!("Value in raw pointer:");
        println!("value1: {:x}", val1);
        println!("value2: {:x}", val2);
    }
}

fn main() {
    let arr : [i32; 5] = [1, 2, 3, 4, 5];
    let address : &[i32; 5] = &arr;
    println!("Address of arr: {:p}", address);

    raw_slice(address as &[i32]);
}
```

在这个示例中，我们arr是长度为5的i32类型的数组。address是一个普通的指向arr的借用指针。我们可以用as关键字把address转换为一个胖指针&[i32]，并传递给raw_slice函数。在raw_slice函数内部，我们利用了unsafe的transmute函数。我们可以把它看作一个强制类型转换，类似reinterpret_cast，通过这个函数，我们把胖指针的内部数据转换成了两个usize大小的整数来看待。编译，执行，结果为：

```
$ ./test
Address of arr: 0xe2e236f6cc
Value in raw pointer:
value1: e2e236f6cc
value2: 5
```

由此可见，胖指针内部的数据既包含了指向源数组的地址，又包含了该切片的长度。

对于DST类型，Rust有如下限制：

·只能通过指针来间接创建和操作DST类型，&[T]Box<[T]>可以，[T]不可以；

·局部变量和函数参数的类型不能是DST类型，因为局部变量和函数参数必须在编译阶段知道它的大小因为目前unsized rvalue功能还没有实现；

·enum中不能包含DST类型，struct中只有最后一个元素可以是DST，其他地方不行，如果包含有DST类型，那么这个结构体也就成了DST类型。

Rust设计出DST类型，使得类型暂时系统更完善，也有助于消除一些C/C++中容易出现的bug。这一设计的好处有：

·首先，DST类型虽然有一些限制条件，但我们依然可以把它当成合法的类型看待，比如，可以为这样的类型实现trait、添加方法、用在泛型参数中等；

·胖指针的设计，避免了数组类型作为参数传递时自动退化为裸指针类型，丢失了长度信息的问题，保证了类型安全；

·这一设计依然保持了与“所有权”“生命周期”等概念相容的特点。

数组切片不只是提供了“数组到指针”的安全转换，配合上Range功能，它还能提供数组的局部切片功能。

6.1.5 Range

Rust中的Range代表一个“区间”，一个“范围”，它有内置的语法支持，就是两个小数点..。示例如下：

```
fn main() {
    let r = 1..10;   // r是一个Range<i32>,中间是两个点,代表[1,10)这个区间
    for i in r {
        print!("{:?}\t", i);
    }
}
```

编译，执行，结果为：

```
$ ./test
1       2       3       4       5       6       7       8       9
```

需要注意的是，在begin..end这个语法中，前面是闭区间，后面是开区间。这个语法实际上生成的是一个std：：ops：：Range<_>类型的变量。该类型在标准库中的定义如下：

```
pub struct Range<Idx> {
    /// The lower bound of the range (inclusive).
    pub start: Idx,
    /// The upper bound of the range (exclusive).
    pub end: Idx,
}
```

所以，上面那段示例代码实质上等同于下面这段代码：

```
use std::ops::Range;

fn main() {
    let r = Range {start: 1, end: 10};   // r是一个Range<i32>
    for i in r {
            print!("{:?}\t", i);
    }
}
```

两个小数点的语法仅仅是一个“语法糖”而已，用它构造出来的变量是Range类型。

这个类型本身实现了Iterator trait，因此它可以直接应用到循环语句

中。Range具有迭代器的全部功能，因此它能调用迭代器的成员方法。比如，我们要实现从100递减到10，中间间隔为10的序列，可以这么做（具体语法请参考后文中的迭代器、闭包等章节）：

```
fn main() {
    use std::iter::Iterator;
// 先用rev方法把这个区间反过来,然后用map方法把每个元素乘以10
    let r = (1i32..11).rev().map(|i| i * 10);

    for i in r {
        print!("{:?}\t", i);
    }
}
```

执行结果为：

```
$ ./test
100     90      80      70      60      50      40      30      20      10
```

在Rust中，还有其他的几种Range，包括

·std：：ops：：RangeFrom代表只有起始没有结束的范围，语法为start..，含义是[start，+∞）；

·std：：ops：：RangeTo代表没有起始只有结束的范围，语法为..end，对有符号数的含义是（-∞，end），对无符号数的含义是[0，end）；

·std：：ops：：RangeFull代表没有上下限制的范围，语法为..，对有符号数的含义是（-∞，+∞），对无符号数的含义是[0，+∞）。

数组和Range之间最常用的配合就是使用Range进行索引操作。示例如下：

```
fn print_slice(arr: &[i32]) {
    println!("Length: {}", arr.len());

    for item in arr {
        print!("{}\t", item);
    }
    println!("");
}
```

```
fn main() {
    let arr : [i32; 5] = [1, 2, 3, 4, 5];
    print_slice(&arr[..]);    // full range

    let slice = &arr[2..];    // RangeFrom
    print_slice(slice);

    let slice2 = &slice[..2]; // RangeTo
    print_slice(slice2);
}
```

编译，执行，结果为：

```
Length: 5
1       2       3       4       5
Length: 3
3       4       5
Length: 2
3       4
```

第一次打印，内容为整个arr的所有区间。第二次打印，是从arr的index为2的元素开始算起，一直到最后。注意数组是从index为0开始计算的。第三次打印，是从slice的头部开始，长度为2，因此只打印出了3、4两个数字。

在许多时候，使用数组的一部分切片作为被操作对象在函数间传递，既保证了效率（避免直接复制大数组），又能保证将所需要执行的操作限制在一个可控制的范围内（有长度信息，有越界检查），还能控制其读写权限，非常有用。

虽然左闭右开区间是最常用的写法，然而，在有些情况下，这种语法不足以处理边界问题。比如，我们希望产生一个i32类型的从0到i32：：MAX的范围，就无法表示。因为按语法，我们应该写0..（i32：：MAX+1），然而（i32：：MAX+1）已经溢出了。所以，Rust还提供了一种左闭右闭区间的语法，它使用这种语法来表示..=。

闭区间对应的标准库中的类型是：

·std：：ops：：RangeInclusive，语法为start..=end，含义是[start，end]。

·std：：ops：：RangeToInclusive，语法为..=end，对有符号数的含

义是（-∞，end]，对无符号数的含义是[0，end]

6.1.6 边界检查

在前面的示例中，我们的“索引”都是一个合法的值，没有超过数组的长度。如果我们给“索引”一个非法的值会怎样呢：

```
fn main() {
    let v = [10i32, 20, 30, 40, 50];
    let index : usize =std :: env :: args(). nth(1). map(|x|x.parse().unwrap_or(0)).
    println!("{:?}", v[index]);
}
```

编译通过，执行thread'main'panicked at'index out of bounds：the len is 5 but the index is 10'。可以看出，如果用/test 10，则会出现数组越界，Rust目前还无法任意索引执行编译阶段边界检查，但是在运行阶段执行了边界检查。下面我们分析一下边界检查背后的故事。

在Rust中，“索引”操作也是一个通用的运算符，是可以自行扩展的。如果希望某个类型可以执行“索引”读操作，就需要该类型实现std：：ops：：Index trait，如果希望某个类型可以执行“索引”写操作，就需要该类型实现std：：ops：：IndexMut trait。

对于数组类型，如果使用usize作为索引类型执行读取操作，实际执行的是标准库中的以下代码：

```
impl<T> ops::Index<usize> for [T] {
    type Output = T;

    fn index(&self, index: usize) -> &T {
        assert!(index < self.len());
        unsafe { self.get_unchecked(index) }
    }
}
```

代码中使用的assert！宏定义在libcore/macros.rs中，源码是这样的：

```
macro_rules! assert {
    ($cond:expr) => (
        if !$cond {
```

```
            panic!(concat!("assertion failed: ", stringify!($cond)))
        }
    );
    ($cond:expr, $($arg:tt)+) => (
        if !$cond {
            panic!($($arg)+)
        }
    );
}
```

也就是说，如果index超过了数组的真实长度范围，会执行panic！操作，导致线程abort。使用Range等类型做Index操作的执行流程与此类似。

为了防止索引操作导致程序崩溃，如果我们不确定使用的“索引”是否合法，应该使用get（）方法调用来获取数组中的元素，这个方法不会引起panic！，它的返回类型是Option<T>，示例如下：

```
fn main() {
    let v = [10i32, 20, 30, 40, 50];
    let first = v.get(0);
    let tenth = v.get(10);
    println!("{:?} {:?}", first, tenth);
}
```

输出结果为：“Some（10）None”。

Rust宣称的优点是“无GC的内存安全”，那么数组越界会直接导致程序崩溃这件事情是否意味着Rust不够安全呢？不能这么理解。Rust保证的“内存安全”，并非意味着“永不崩溃”。Rust中关于数组越界的行为，定义得非常清晰。相比于C/C++，Rust消除的是“未定义行为”（Undefined Behaviour）。

对于明显的数组越界行为，在Rust中可以通过lint检查来发现。大家可以参考“clippy”这个项目，它可以检查出这种明显的常量索引越界的现象。然而，总体来说，在Rust里面，靠编译阶段静态检查是无法消除数组越界的行为的。

一般情况下，Rust不鼓励大量使用“索引”操作。正常的“索引”操作都会执行一次“边界检查”。从执行效率上来说，Rust比C/C++的数组索引效率低一点，因为C/C++的索引操作是不执行任何安全性检查的，它们对应的Rust代码相当于调用get_unchecked（）函数。在Rust中，更加

地道的做法是尽量使用“迭代器”方法。“迭代器”非常重要，本书将在第24章专门详细分析，下面是使用迭代器操作数组的一些简单示例：

```
fn main() {
    use std::iter::Iterator;

    let v = &[10i32, 20, 30, 40, 50];

    // 如果我们同时需要index和内部元素的值,调用enumerate()方法
    for (index, value) in v.iter().enumerate() {
        println!("{} {}", index, value);
    }

    // filter方法可以执行过滤,nth函数可以获取第n个元素
    let item = v.iter().filter(|&x| *x % 2 == 0).nth(2);
    println!("{:?}", item);
}
```

Iterator还有许多有用的方法，合理地组合使用它们，能使程序表达能力强，可读性好，安全高效，可以满足我们绝大多数的需求。

6.2 字符串

字符串是非常重要的常见类型。相比其他很多语言，Rust的字符串显得有点复杂，主要是跟所有权有关。Rust的字符串涉及两种类型，一种是&str，另外一种是String。

6.2.1 &str

str是Rust的内置类型。&str是对str的借用。Rust的字符串内部默认是使用utf-8编码格式的。而内置的char类型是4字节长度的，存储的内容是Unicode Scalar Value。所以，Rust里面的字符串不能视为char类型的数组，而更接近u8类型的数组。实际上str类型有一种方法：fn as_ptr（&self）->*const u8。它内部无须做任何计算，只需做一个强制类型转换即可：

```
self as *const str as *const u8
```

这样设计有一个缺点，就是不能支持O（1）时间复杂度的索引操作。如果我们要找一个字符串s内部的第n个字符，不能直接通过s[n]得到，这一点跟其他许多语言不一样。在Rust中，这样的需求可以通过下面的语句实现：

```
s.chars().nth(n)
```

它的时间复杂度是O（n），因为utf-8是变长编码，如果我们不从头开始过一遍，根本不知道第n个字符的地址在什么地方。

但是，综合来看，选择utf-8作为内部默认编码格式是缺陷最少的一种方式了。相比其他的编码格式，它有相当多的优点。比如：它是大小端无关的，它跟ASCII码兼容，它是互联网上的首选编码，等等。关于各种编码格式之间的详细优劣对比，强烈建议大家参考下面这个网站：

http://utf8everywhere.org/

跟上一章讲过的数组类似，[T]是DST类型，对应的str是DST类型。&[T]是数组切片类型，对应的&str是字符串切片类型。示例如下：

```
fn main() {
    let greeting : &str = "Hello";
    let substr : &str = &greeting[2..];
    println!("{}", substr);
}
```

编译，执行，可见它跟数组切片的行为很相似。

&str类型也是一个胖指针，可以用下面的示例证明：

```
fn main() {
    println!("Size of pointer: {}", std::mem::size_of::<*const ()>());
    println!("Size of &str   : {}", std::mem::size_of::<&str>());
}
```

编译，执行，结果为：

```
Size of pointer: 8
Size of &str   : 16
```

它内部实际上包含了一个指向字符串片段头部的指针和一个长度。所以，它跟C/C++的字符串不同：C/C++里面的字符串以'\0'结尾，而Rust的字符串是可以中间包含'\0'字符的。

6.2.2 String

接下来讲String类型。它跟&str类型的主要区别是，它有管理内存空间的权力。关于“所有权”和“借用”的关系，在本书第二部分会详细讲解。&str类型是对一块字符串区间的借用，它对所指向的内存空间没有所有权，哪怕&mut str也一样。比如：

```
let greeting : &str = "Hello";
```

我们没办法扩大greeting所引用的范围，在它后面增加内容。但是

String类型可以。示例如下：

```
fn main() {
    let mut s = String::from("Hello");
    s.push(' ');
    s.push_str("World.");
    println!("{}", s);
}
```

这是因为String类型在堆上动态申请了一块内存空间，它有权对这块内存空间进行扩容，内部实现类似于std：：Vec<u8>类型。所以我们可以把这个类型作为容纳字符串的容器使用。

这个类型实现了Deref<Target=str>的trait。所以在很多情况下，&String类型可以被编译器自动转换为&str类型。关于Deref大家可以参考本书第二部分“解引用”章节。我们写个小示例演示一下：

```
fn capitalize(substr: &mut str) {
    substr.make_ascii_uppercase();
}

fn main() {
    let mut s = String::from("Hello World");
    capitalize(&mut s);
    println!("{}", s);
}
```

在这个例子中，capitalize函数调用的时候，形式参数要求是&mut str类型，而实际参数是&mut String类型，这里编译器给我们做了自动类型转换。在capitalize函数内部，它有权修改&mut str所指向的内容，但是无权给这个字符串扩容或者释放内存。

Rust的内存管理方式和C++有很大的相似之处。如果用C++来对比，Rust的String类型类似于std：：string，而Rust的&str类型类似于std：：string_view。示例如下：

```
#include <iostream>
#include <string>
#include <string_view>

int main() {
    std::string s = "Hello world";
    std::string_view v(&s[5], 5);
```

```
    std::cout << "Size of string_view:" << sizeof(v) << "\n"
            << "Value: " << v << std::endl;
}
```

这样的对比可能会让有C++背景的读者更容易理解一些。

第7章　模式解构

7.1　简介

“Pattern Destructure”是Rust中一个重要且实用的设计。笔者暂且将其翻译为“模式解构”。注意这里的“Destructure”和“Destructor”是完全不同的两个单词，代表完全不同的含义。“Destructure”的意思是把原来的结构肢解为单独的、局部的、原始的部分；“Destructor”是指“析构器”，是一个与“构造器”相对应的概念，是在对象被销毁的时候调用的。

下面举例说明什么是模式解构：

```
let tuple = (1_i32, false, 3f32);
let (head, center, tail) = tuple;
```

以上的第二句代码就是一个典型的“模式解构”。我们可以这么理解，第一句话是“构造”，它把三个元素组合到了一起，形成了一个tuple。而第二句代码则是刚好反过来，把一个组合数据结构，拆解开来，分成了三个不同的变量。在let语句中，赋值号左边的内容就是本节中我们所说的“模式”，赋值号右边的内容就是我们需要被“解构”的内容。这个“模式”中引入了三个新的变量head、center、tail，它们分别绑定了这个tuple的三个成员。

Rust中模式解构功能设计得非常美观，它的原则是：构造和解构遵循类似的语法，我们怎么把一个数据结构组合起来的，我们就怎么把它拆解开来。为了更好地说明这个问题，我们再来看一个更复杂一点的例子。比如，我们有一个struct，里面包含另外一个struct类型：

```
struct T1 (i32, char);

struct T2 {
    item1: T1,
    item2: bool,
}

fn main()
{
    let x = T2 {
```

```
        item1: T1(0, 'A'),
        item2: false,
    };

    let T2 {
        item1: T1(value1, value2),
        item2: value3,
    } = x;

    println!("{} {} {}", value1, value2, value3);
}
```

如前所述，我们首先构造了一个T2类型的变量x，它内部又嵌套包含了其他的结构体。实际上，我们完全可以一次性解构多个层次，直接把这个对象内部深处的元素拆解出来。第二条let语句，就是一个比较复杂的“模式解构”，赋值号的左边不仅仅是一个变量名，还是一个完整的“模式”，在这个模式中引入了三个变量value1、value2、value3，分别绑定到了item1内部的两个成员以及item2上。

编译，执行，打印出来的结果为"0 A false"。

Rust的“模式解构”功能不仅出现在let语句中，还可以出现在match、if let、while let、函数调用、闭包调用等情景中。而match具有功能最强大的模式匹配。下面首先介绍match语法。

7.2 match

首先，我们看看使用match的最简单的示例：

```
enum Direction {
    East, West, South, North
}

fn print(x: Direction)
{
    match x {
        Direction::East  => {
            println!("East");
        }
        Direction::West  => {
            println!("West");
        }
        Direction::South => {
            println!("South");
        }
        Direction::North => {
            println!("North");
        }
    }
}

fn main() {
    let x = Direction::East;
    print(x);
}
```

当一个类型有多种取值可能性的时候，特别适合使用match表达式。对于这个示例，我们也可以用多个if-else表达式完成类似的功能，但是match表达式还有更强大的功能，下面我们继续说明。

7.2.1 exhaustive

如果我们把上例中的Direction：：North对应的分支删除：

```
match x {
    Direction::East  => {
        println!("East");
    }
    Direction::West  => {
        println!("West");
    }
```

```
    Direction::South => {
        println!("South");
    }
}
```

编译，出现了编译错误：

```
error[E0004]: non-exhaustive patterns: `North` not covered
```

这是因为Rust要求match需要对所有情况做完整的、无遗漏的匹配，如果漏掉了某些情况，是不能编译通过的。exhaustive意思是无遗漏的、穷尽的、彻底的、全面的。exhaustive是Rust模式匹配的重要特点。

有些时候我们不想把每种情况一一列出，可以用一个下划线来表达“除了列出来的那些之外的其他情况”：

```
match x {
    Direction::East  => {
        println!("East");
    }
    Direction::West  => {
        println!("West");
    }
    Direction::South => {
        println!("South");
    }
    _ => {
        println!("Other");
    }
}
```

正因为如此，在多个项目之间有依赖关系的时候，在上游的一个库中对enum增加成员，是一个破坏兼容性的改动。因为增加成员后，很可能会导致下游的使用者match语句编译不过。为解决这个问题，Rust提供了一个叫作non_exhaustive的功能（目前还没有稳定）。示例如下：

```
#[non_exhaustive]
pub enum Error {
    NotFound,
    PermissionDenied,
    ConnectionRefused,
}
```

上游库作者可以用一个叫作“non_exhaustive”的attribute来标记一个enum或者struct，这样在另外一个项目中使用这个类型的时候，无论如何都没办法在match表达式中通过列举所有的成员实现完整匹配，必须使用下划线才能完成编译。这样，以后上游库里面为这个类型添加新成员的时候，就不会导致下游项目中的编译错误了因为它已经存在一个默认分支匹配其他情况。

7.2.2 下划线

下划线还能用在模式匹配的各种地方，用来表示一个占位符，虽然匹配到了但是忽略它的值的情况：

```
struct P(f32, f32, f32);

fn calc(arg: P) -> f32 {
    // 匹配 tuple struct,但是忽略第二个成员的值
    let P(x, _, y) = arg;
    x * x + y * y
}

fn main() {
    let t = P(1.0, 2.0, 3.0);
    println!("{}", calc(t));
}
```

对于上例，实际上我们还能写得更简略一点。因为函数参数本身就具备“模式解构”功能，我们可以直接在参数中完成解构：

```
struct P(f32, f32, f32);
// 参数类型是 P,参数本身是一个模式,解构之后,变量x、y分别绑定了第一个和第三个成员
fn calc(P(x, _, y): P) -> f32 {
    x * x + y * y
}
fn main() {
    let t = P(1.0, 2.0, 3.0);
    println!("{}", calc(t));
}
```

另外需要提醒的一点是，下划线更像是一个“关键字”，而不是普通的“标识符”（identifier），把它当成普通标识符使用是会有问题的。举例如下：

```
fn main() {
    let _ = 1_i32;
    let x = _ + _;
    println!("{}", x);
}
```

编译可见，编译器并不会把单独的下划线当成一个正常的变量名处理：

```
error: expected expression, found `_`
 --> test.rs:4:13
  |
4 |     let x = _ + _;
  |             ^

error[E0425]: cannot find value `x` in this scope
```

如果把下划线后面跟上字母、数字或者下划线，那么它就可以成为一个正常的标识符了。比如，连续两个下划线__，就是一个合法的、正常的“标识符”。

let_=x；和let_y=x；具有不一样的意义。这一点在后面的“析构函数”部分还会继续强调。如果变量x是非Copy类型，let_=x；的意思是“忽略绑定”，此时会直接调用x的析构函数，我们不能在后面使用下划线_读取这个变量的内容；而let_y=x；的意思是“所有权转移”，_y是一个正常的变量名，x的所有权转移到了_y上，_y在后面可以继续使用。

下划线在Rust里面用处很多，比如：在match表达式中表示“其他分支”，在模式中作为占位符，还可以在类型中做占位符，在整数和小数字面量中做连接符，等等。

除了下划线可以在模式中作为“占位符”，还有两个点..也可以在模式中作为“占位符”使用。下划线表示省略一个元素，两个点可以表示省略多个元素。比如：

```
fn main() {
    let x = (1, 2, 3);
    let (a, _) = x;    // 模式解构
    println!("{}", a);
}
```

如果我们希望只匹配tuple中的第一个元素，其他的省略，那么用一个下划线是不行的，因为这样写，左边的tuple和右边的tuple不匹配。修改方案有两种。一种是：

```
let (a, _, _) = x; // 用下划线,那么个数要匹配
```

另一种是：

```
let (a, ..) = x;   // 用两个点,表示其他的全部省略
let (a, .., b) = x;// 用两个点,表示只省略所有元素也是可以的
```

7.2.3 match也是表达式

跟Rust中其他流程控制语法一样，match语法结构也同样可以是表达式的一部分。示例如下：

```
enum Direction {
    East, West, South, North
}

fn direction_to_int(x: Direction) -> i32
{
    match x {
        Direction::East  => 10,
        Direction::West  => 20,
        Direction::South => 30,
        Direction::North => 40,
    }
}

fn main() {
    let x = Direction::East;
    let s = direction_to_int(x);
    println!("{}", s);
}
```

match表达式的每个分支可以是表达式，它们要么用大括号包起来，要么用逗号分开。每个分支都必须具备同样的类型。在此例中，这个match表达式的类型为i32，在match后面没有分号，因此这个表达式的值将会作为整个函数的返回值传递出去。

match除了匹配“结构”，还可以匹配“值”：

```
fn category(x: i32) {
    match x {
        -1 => println!("negative"),
        0  => println!("zero"),
        1  => println!("positive"),
        _  => println!("error"),
    }
}
fn main() {
    let x = 1;
    category(x);
}
```

我们可以使用或运算符|来匹配多个条件，比如：

```
fn category(x: i32) {
    match x {
        -1 | 1 => println!("true"),
        0  => println!("false"),
        _  => println!("error"),
    }
}

fn main() {
    let x = 1;
    category(x);
}
```

我们还可以使用范围作为匹配条件，使用..表示一个前闭后开区间范围，使用..=表示一个闭区间范围：

```
let x = 'X';

match x {
    'a' ..= 'z' => println!("lowercase"),
    'A' ..= 'Z' => println!("uppercase"),
    _ => println!("something else"),
}
```

7.2.4 Guards

可以使用if作为“匹配看守”（match guards）。当匹配成功且符合if条件，才执行后面的语句。示例如下：

```
enum OptionalInt {
```

```
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

在对变量的“值”进行匹配的时候，编译器依然会保证“完整无遗漏”检查。但是这个检查目前做得并不是很完美，某些情况下会发生误报的情况，因为毕竟编译器内部并没有一个完整的数学解算功能：

```
fn main() {
    let x = 10;

    match x {
        i if i > 5 => println!("bigger than five"),
        i if i <= 5 => println!("small or equal to five"),
    }
}
```

从if条件中可以看到，实际上我们已经覆盖了所有情况，可惜还是出现了编译错误。编译器目前还无法完美地处理这样的情况。我们只能再加入一条分支，单纯为了避免编译错误：

```
_ => unreachable!(),
```

编译器会保证match的所有分支合起来一定覆盖了目标的所有可能取值范围，但是并不会保证各个分支是否会有重叠的情况（毕竟编译器不想做成一个完整的数学解算器）。如果不同分支覆盖范围出现了重叠，各个分支之间的先后顺序就有影响了：

```
fn intersect(arg: i32) {
    match arg {
        i if i < 0 => println!("case 1"),
        i if i < 10 => println!("case 2"),
        i if i * i < 1000 => println!("case 3"),
        _ => println!("default case"),
    }
}
```

```
fn main() {
    let x = -1;
    intersect(x);
}
```

如果我们进行匹配的值同时符合好几条分支，那么总会执行第一条匹配成功的分支，忽略其他分支。

7.2.5 变量绑定

我们可以使用@符号绑定变量。@符号前面是新声明的变量，后面是需要匹配的模式：

```
let x = 1;

match x {
    e @ 1 ..= 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

当一个Pattern嵌套层次比较多的时候，如果我们需要匹配更深层次作为条件，希望绑定上面一层的数据，就需要像下面这样写：

```
#![feature(exclusive_range_pattern)]

fn deep_match(v: Option<Option<i32>>) -> Option<i32> {
    match v {
        // r 绑定到的是第一层 Option 内部,r 的类型是 Option<i32>
        // 与这种写法含义不一样：Some(Some(r)) if (1..10).contains(r)
        Some(r @ Some(1..10)) => r,
        _ => None,
    }
}

fn main() {
    let x = Some(Some(5));
    println!("{:?}", deep_match(x));

    let y = Some(Some(100));
    println!("{:?}", deep_match(y));
}
```

如果在使用@的同时使用|，需要保证在每个条件上都绑定这个名字：

```
```

```
let x = 5;

match x {
    e @ 1 .. 5 | e @ 8 .. 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

7.2.6 ref和mut

如果我们需要绑定的是被匹配对象的引用，则可以使用ref关键字：

```
let x = 5_i32;

match x {
    ref r => println!("Got a reference to {}", r), // 此时 r 的类型是 `&i32`
}
```

之所以在某些时候需要使用ref，是因为模式匹配的时候有可能发生变量的所有权转移，使用ref就是为了避免出现所有权转移。

那么ref关键字和引用符号&有什么关系呢？考虑以下代码中变量绑定x分别是什么类型？

```
let x = 5_i32;      // i32
let x = &5_i32;     // &i32
let ref x = 5_i32;  // ???
let ref x = &5_i32; // ???
```

注意：ref是“模式”的一部分，它只能出现在赋值号左边，而&符号是借用运算符，是表达式的一部分，它只能出现在赋值号右边。

为了搞清楚这些变量绑定的分别是什么类型，我们可以把变量的类型信息打印出来看看。有两种方案：

·利用编译器的错误信息来帮我们理解；

·利用标准库里面的intrinsic函数打印。

方案一，示例如下：

```
// 这个函数接受一个 unit 类型作为参数
fn type_id(_: ()) {}

fn main() {
    let ref x = 5_i32;
    // 实际参数的类型肯定不是 unit,此处必定有编译错误,通过编译错误,我们可以看到实参的具体类型
    type_id(x);
}
```

这里我们写了一个不做任何事情的函数type_id。它接收一个参数，类型是（），我们在main函数中调用这个函数，肯定会出现类型不匹配的编译错误。错误信息为：

```
error[E0308]: mismatched types
 --> test.rs:5:13
  |
5 |     type_id(x);
  |             ^ expected (), found &i32
  |
  = note: expected type `()`
             found type `&i32`
```

这个错误信息正是我们想要的内容。从中我们可以看到我们感兴趣的变量类型。

方案二，示例如下：

```
#![feature(core_intrinsics)]

fn print_type_name<T>(_arg: &T) {
    unsafe {
        println!("{}", std::intrinsics::type_name::<T>());
    }
}

fn main() {
    let ref x = 5_i32;
    print_type_name(&x);
}
```

利用标准库里面提供的type_name函数，可以打印出变量的类型信息。

从以上方案可以看到，let ref x=5_i32；和let x=&5_i32；这两条语句中，变量x是同样的类型&i32。

同理，我们可以试验得出，let ref x=&5_i32；语句中，变量x绑定的类型是&&i32。对于更复杂的情况，读者可以用类似的办法做试验，多看看各种情况下具体的类型是什么。

ref关键字是“模式”的一部分，不能修饰赋值号右边的值。let x=ref 5_i32；这样的写法是错误的语法。

mut关键字也可以用于模式绑定中。mut关键字和ref关键字一样，是“模式”的一部分。Rust中，所有的变量绑定默认都是“不可更改”的。只有使用了mut修饰的变量绑定才有修改数据的能力。最简单的例子如下：

```
fn main() {
    let x = 1;
    x = 2;
}
```

编译错误，错误信息为：error：re-assignment of immutable variable x。我们必须使用let mut x=1；，才能在以后的代码中修改变量绑定x。

使用了mut修饰的变量绑定，可以重新绑定到其他同类型的变量。

```
fn main() {
    let mut v = vec![1i32, 2, 3];
    v = vec![4i32, 5, 6];          // 重新绑定到新的Vec
    v = vec![1.0f32, 2, 3];        // 类型不匹配,不能重新绑定
}
```

重新绑定与前面提到的“变量遮蔽”（shadowing）是完全不同的作用机制。“重新绑定”要求变量本身有mut修饰，并且不能改变这个变量的类型。“变量遮蔽”要求必须重新声明一个新的变量，这个新变量与老变量之间的类型可以毫无关系。

Rust在“可变性”方面，默认为不可修改。与C++的设计刚好相反。C++默认为可修改，使用const关键字修饰的才变成不可修改。

mut关键字不仅可以在模式用于修饰变量绑定，还能修饰指针（引用），这里是很多初学者常常搞混的地方。mut修饰变量绑定，与&mut型引用，是完全不同的意义。

```
let mut x: &mut i32;
//  ^1      ^2
```

以上两处的mut含义是不同的。第1处mut，代表这个变量x本身可变，因此它能够重新绑定到另外一个变量上去，具体到这个示例来说，就是指针的指向可以变化。第2处mut，修饰的是指针，代表这个指针对于所指向的内存具有修改能力，因此我们可以用*x=1；这样的语句，改变它所指向的内存的值。

mut关键字不像想象中那么简单，我们会经常碰到与mut有关的各种编译错误。在本节中，只是简单介绍了它的语法，关于mut关键字代表的深层含义，及其正确使用方法，在本书第二部分还会有详细描述。

至于为什么有些场景下，我们必须使用ref来进行变量绑定，背后的原因跟“move”语义有关。关于变量的生命周期/所有权/借用/move等概念，请参考本书第二部分。下面举个例子：

```
fn main() {
    let mut x : Option<String> = Some("hello".into());
    match x {
        Some(i) => i.push_str("world"),
        None => println!("None"),
    }

    println!("{:?}", x);
}
```

这段代码编译器会提示编译错误，第一个原因是，局部变量i是不可变的，所以它无权调用push_str方法。我们可以修改为Some（mut i）再次编译。还是会发生编译错误。这次提示的是，“use of partially moved value`x`”。因为编译器认为这个match语句把内部的String变量移动出来了，所以后续的打印x的值是错误的行为。为了保证这个match语句不发生移动，我们需要把这个模式继续修改为Some（ref mut i），这一次，编译通过了。

这个问题还有更简单的修复方式，就是把match x改为match &mut x：

```
fn main() {
    let mut x : Option<String> = Some("hello".into());
```

```
    match &mut x {
        Some(i) => i.push_str("world"),
        None => println!("None"),
    }

    println!("{:?}", x);
}
```

在这种情况下，编译器没有报错，是因为我们对指针做了模式匹配，编译器很聪明地推理出来了，变量i必须是一个指针类型，所以它帮我们自动加了ref mut模式，它通过自动类型推导自己得出了结论，认为i的类型是&mut String。这是编译器专门做的一个辅助功能。在很多时候，特别是类型嵌套层次很多的时候，处处都要关心哪个pattern是不是要加个mut或者ref，其实是个很烦人的事情。有了这个功能，用户就不用每次都写麻烦的mut或者ref，在一些显而易见的情况下，编译器自动来帮我们合理地使用mut或者ref。读者可以用我们前面实现的print_type_name试试，用Some（i）模式，以及用Some（ref mut i）模式，变量i的类型分别是什么。结论是类型一样。因为在我们不明确写出来ref mut模式的时候，编译器帮我们做了更合理的自动类型推导。

7.3 if-let和while-let

Rust不仅能在match表达式中执行“模式解构”，在let语句中，也可以应用同样的模式。Rust还提供了if-let语法糖。它的语法为if let PATTERN=EXPRESSION{BODY}。后面可以跟一个可选的else分支。

比如，我们有一个类型为Option<T>的变量optVal，如果我们需要取出里面的值，可以采用这种方式：

```
match optVal {
    Some(x) => {
        doSomethingWith(x);
    }
    _ => {}
}
```

这样做语法比较冗长，从变量optVal到执行操作的函数有两层缩进，我们还必须写一个不做任何操作的语句块才能满足语法要求。换一种方式，通过Option类型的方法，我们可以这么做：

```
if optVal.is_some() {                // 首先判断它一定是 Some(_)
    let x = optVal.unwrap();         // 然后取出内部的数据
    doSomethingWith(x);
}
```

从视觉上来看，代码缩进层次减少到了一层。但是它在运行期实际上判断了两次optVal里面是否有值：第一次是is_some（）函数，第二次是unwrap（）函数。从执行效率上来说是降低了的。而使用if-let语法，则可以这么做：

```
if let Some(x) = optVal {
    doSomethingWith(x);
}
```

这其实是一个简单的语法糖，其背后执行的代码与match表达式相比，并无效率上的差别。它跟match的区别是：match一定要完整匹配，if-let只匹配感兴趣的某个特定的分支，这种情况下的写法比match简单

点。同理，while-let与if-let一样，提供了在while语句中使用“模式解构”的能力，此处就不再举例。

if-let和while-let还支持模式的“或”操作（此功能目前尚未在编译器中实现）。比如，我们有如下enum定义：

```
enum E<T> {
    A(T), B(T), C, D, E, F
}
```

如果我们需要匹配“C或者D”，那么可以这样写：

```
let r = if let C | D = x { 1 } else { 2 };
```

这段代码等同于：

```
let r = match x {
    C | D => 1,
    _ => 2,
}
```

在这个匹配过程中还可以有变量绑定，比如：

```
while let A(x) | B(x) = expr {
    do_something(x);
}
```

这段代码等同于：

```
match expr {
    A(x) | B(x) => do_something(x),
    _ => {},
}
```

7.4 函数和闭包参数做模式解构

示例如下。一个函数接受一个结构体参数，可以直接在参数这里做模式解构：

```
struct T {
    item1: char,
    item2: bool,
}

fn test( T{item1: arg1, item2: arg2} : T) {
    println!("{} {}", arg1, arg2);
}

fn main()
{
    let x = T {
        item1: 'A',
        item2: false,
    };

    test(x);
}
```

7.5 总结

“模式解构”是Rust中较为复杂的一个功能，但是非常实用。

·Rust的“模式解构”功能在语法上具有良好的一致性和扩展性；

·Rust的“模式解构”功能不仅出现在match语句中，还可以出现在let、if-let、while-let、函数调用、闭包调用等情景中；

·Rust的“模式解构”功能可以应用于各种数据类型，包括但不限于tuple、struct、enum等，暂时在稳定版中不支持slice的模式匹配；

·Rust的“模式解构”功能要求“无遗漏”的分析（exhaustive case analysis），确保不会因为不小心而漏掉某些情况；

·Rust的“模式解构”与Rust的核心所有权管理功能完全相容。

第8章　深入类型系统

Rust的类型系统实际上是一种代数类型系统（Algebraic data type）。它在数学上是有严格定义的，非常严谨的一套理论。本章试图用一种比较通俗的语言，简单地解释一下什么是代数类型系统，我们应该如何理解它，以及它给Rust带来了哪些优势。

8.1 代数类型系统

代数，我们以前都学过。比如，给定一个整数x，我们可以对它执行一些数学运算，像加法、乘法之类的：

```
x + 1
2 * x
```

我们还可以从中总结出一些规律，比如，任何一个整数与0之和等于它本身，任何数与0之积等于0，任何一个整数与1之积等于它本身。用数学语言描述，可以这样写：

```
0 + x = x
0 * x = 0
1 * x = x
```

在代数里面，我们的变量x代表的是某个集合内的数字，执行的操作一般是加减乘除一类的数学运算。而对应到代数类型系统上，我们可以把类型类比为代数中的变量，把类型之间的组合关系类比为代数中的数学运算。

我们可以做这样一个假定，一个类型所有取值的可能性叫作这个类型的“基数”（cardinality）。那么在此基础上，我们可以得出结论：最简单的类型unit（）的基数就是1，它可能的取值范围只能是（）。再比如说，bool类型的基数就是2，可能的取值范围有两个，分别是true和false。对于i32类型，它的取值范围是232，我们用Cardinality（i32）来代表i32的基数。

我们把多个类型组合到一起形成新的复合类型，这个新的类型就会有新的基数。如果两个类型的基数是一样的，那么我们可以说它们携带的信息量其实是一样的，我们也可以说它们是“同构”的。下面是一个典型的例子：

```
type T1 = [i32; 2];
type T2 = (i32, i32);
struct T3(i32, i32);
```

```
struct T4 {
    field1: i32,
    field2: i32,
}
```

上面出现了四个类型，在实践中，它们不是同一个类型，是无法通用的。但是从数学上讲，这四个类型表达出来的信息量是完全一样的，它们都只能装下两个i32类型的成员。它们的基数都是Cardinality（i32）*Cardinality（i32）。

tuple、struct、tuple struct这几种类型，实质上是同样的内存布局，区别仅仅在于是否给类型及成员起了名字。这几个类型都可以类比为代数中的“求积”运算。没有成员的tuple类型，它的基数就是1。同理，任意一个空struct类型，基数也是1，它们都可以类比为代数运算中的数字1。

那么，如果struct里面有多个成员，比如：

```
struct R {
    var1 : bool,
    var2 : bool,
}
```

R类型包括了两个成员。分别是var1和var2。它的基数是：

```
Cardinality(R) = Cardinality(var1) * Cardinality(var2) = 2 * 2 = 4
```

如果我们在结构体里面加入一个unit类型的成员：

```
struct Prod {
    field1: i32,
    field2: (),
}
```

这个field2成员实际上对这个结构体没有带来什么贡献，它并没有带来额外的信息量：

```
Cardinality(Prod) = Cardinality(field1) * 1 = Cardinality(i32)
```

对于数组类型，可以对应为每个成员类型都相同的tuple类型（或者struct是一样的）。用数学公式类比，则比较像乘方运算。

Rust中的enum类型就相当于代数中的“求和”运算。比如，某个类型可以代表“东南西北”四个方向，我们可以如下设计：

```
enum Direction {
    North, East, South, West
}
```

它可能的取值范围是四种可能性，它的基数就是4。我们可以把它看作是四个无成员的struct的“或”关系。

```
Cardinality(Direction) = Cardinality(North) + Cardinality(East)
                                    + Cardinality(South) + Cardinality(West)
                                    = 4
```

实际上，我们甚至可以将内置的bool类型定义为一个标准库中的enum：

```
enum Bool { True, False }
```

这样定义的Bool类型和目前的内置bool类型表达能力上是没有什么差别的。

标准库中有一个极其常见的类型叫作Option<T>，它是这么定义的：

```
enum Option<T> {
    None,
    Some(T),
}
```

由于它实在是太常用，标准库将Option以及它的成员Some、None都加入到了Prelude中，用户甚至不需要use语句声明就可以直接使用。其中T是一个泛型参数，在使用的时候可以被替换为实际类型。例如，Option<i32>类型实际上等同于：

```
enum Option<i32> {
    Some(i32),
    None
}
```

因此它的基数是：

```
Cardinality(Option<i32>) = Cardinality(i32) + 1
```

同理，我们可以得出一个空的enum，基数是0。

通过上文中的示例，我们可以为“代数类型系统”和“代数”建立一个直观的类比关系。空的enum可以类比为数字0；unit类型或者空结构体可以类比为数字1；enum类型可以类比为代数运算中的求和；tuple、struct可以类比为代数运算中的求积；数组可以类比为代数运算中的乘方。同理，我们还能继续做更多的类比。

enum类型的每个成员还允许包含更多关联数据：

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 }
}
```

这就好比enum的每个成员还可以是tuple或者struct。类比到代数，相当于加法乘法混合运算：

```
Cardinality(Message) = Cardinality(Quit) + Cardinality(ChangeColor) + Cardinality(Mo
          = 1 + Cardinality(i32)^3 + Cardinality(i32)^2
// 此处用 ^ 代表乘方,不是异或
```

加法具有交换率，同理，enum中的成员交换位置，也不会影响它的表达能力；乘法具有交换率，同理，struct中的成员交换位置，也不影响它的表达能力。

乘法具有结合律：x*（y*z）=（x*y）*z，同理，对于tuple类型（A，（B，C））和（（A，B），C）的表达能力是一样的。

继续列下去，我们还能做出更多的类比，这是一个非常庞大的话题。读者可以到网上搜索更多、更详尽的关于ADT的讲解文章。下面我们讲一下Never type这个类型。

8.2 Never Type

如果我们考虑一个类型在机器层面的表示方式，一个类型占用的bit位数可以决定它能携带多少信息。假设我们用bits_of（T）代表类型T占用的bit位数，那么2^bits_of（T）=Cardinality（T）。反过来，我们可以得出结论，存储一个类型需要的bit位数等于bits_of（T）=log2（Cardinality（T））。比如说，bool类型的基数为2，那么在内存中表示这个类型需要的bit位应该为bits_of（bool）=log2（2）=1，也就是1个bit就足够表达。

我们前面已经看到了，在代数类型系统中有一些比较特殊的类型。

像unit类型和没有成员的空struct类型，都可以类比为代数中的数字1。这样的类型在内存中实际需要占用的空间为bits_of（（））=log2（1）=0。也就是说，这样的类型实际上是0大小的类型。这样的性质有很多好处，比如，Rust里面HashSet的定义方式为：

```
pub struct HashSet<T, S = RandomState> {
    map: HashMap<T, (), S>,
}
```

也就是说，只要把HashMap中存的键值对中的“值”指定为unit类型就可以了。这个设计和我们的思维模型是一致的。所谓的HashSet，就是只有key没有value的HashMap。如果我们没有真正意义上的0大小类型，这个设计是无法做到如此简洁的。

没有任何成员的空enum类型，都可以类比为代数中的数字0，例如：

```
enum Never {}
```

这个Never没有任何成员。如果声明一个这种类型的变量，let e=Never：：？？？；，我们都不知道该怎么初始化，因为Rust根本就没有提供任何语法为这样的类型构造出变量。这样的类型在Rust类型系统中的名字叫作never type，它们有一些属性是其他类型不具备的：

·它们在运行时根本不可能存在，因为根本没有什么语法可以构造出这样的变量；

·Cardinality（Never）=0；

·考虑它需要占用的内存空间bits_of（Never）=log2（0）=-∞，也就是说逻辑上是不可能存在的东西；

·处理这种类型的代码，根本不可能执行；

·返回这种类型的代码，根本不可能返回；

·它们可以被转换为任意类型。

这些有什么意义呢，我们来看以下代码：

```
loop {
    ...
    let x : i32 = if cond { 1 } else { continue; };
    ...
}
```

我们知道，在Rust中，if-else也是表达式，而且每个分支表达式类型必须一致。这种有continue的情况，类型检查是怎样通过的呢？如果我们把continue语句的类型指定为never类型，那么一切就都顺理成章了。因为never类型可以转换为任意类型，所以，它可以符合与if分支的类型相一致的规定。它根本不可能返回，所以执行到else分支的时候，接下来根本不会执行对变量x的赋值操作，会进入下一次的循环。如果这个分支大括号内部continue后面还有其他代码，编译器可以很容易判断出来，它后面的代码是永远不可能执行的死代码。一切都在类型系统层面得到了统一。

所以说，never类型是Rust类型系统中不可缺少的一部分。与unit类型类似，一般我们用空tuple（）代表unit类型，Rust里面其实也有一个专门的类型来表示never，也就是我们前面提到过的感叹号！。所谓的“diverging function”就是一个返回never type的函数。在早期版本中，Rust的做法是把diverging function做特殊处理，使得它在分支结构表达式的类型检查能够通过，而没有把它当成真正意义上的类型。后来，有一个RFC 1216对这个never type做了完整的设计，才真正将它提升为一

个类型。而且，直到编写本书时候的1.19版本，这个功能的完整实现也还没有完全做完。

除了在数学形式上的统一，以及显而易见的对分支结构表达式的类型检查有好处之外，一个完整的never type对于Rust还有一些其他的现实意义。下面举几个例子来说明。

场景一：可以使得泛型代码兼容diverging function

比如，一个这样的泛型方法接受一个泛型函数类型作为参数，可是：

```
fn main() {
    fn call_fn<T, F: Fn(i32)->T> (f: F, arg: i32) -> T { f(arg) }
// 如果不把 ! 当成一个类型,那么下面这句话会出现编译错误,因为只有类型才能替换类型参数
    call_fn(std::process::exit, 0);
}
```

场景二：更好的死代码检查

```
let t = std::thread::spawn(|| panic!("nope"));
t.join().unwrap();
println!("hello");
```

如果我们有完整的never类型支持，那么编译器应该可以推理出闭包的返回类型是！，而不是（），因此t.join（）.unwrap（）会产生一个！类型，编译器因此可以检查出println永远不可能执行。

场景三：可以用更好的方式表达“不可能出现的情况”

标准库中有一个trait叫FromStr，它有一个关联类型代表错误：

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

如果某些类型调用from_str方法永远不会出错，那么这个Err类型可以指定为！。

```
struct T(String);

impl FromStr for T {
    type Err = !;

    fn from_str(s: &str) -> Result<T, !> {
        Ok(T(String::from(s)))
    }
}
```

对于错误处理，我们可以让Result退化成没有错误的情况：

```
use std::str::FromStr;
use std::mem::{size_of, size_of_val};

struct T(String);

impl FromStr for T {
    type Err = !;

    fn from_str(s: &str) -> Result<T, !> {
        Ok(T(String::from(s)))
    }
}


fn main() {
    let r: Result<T, !> = T::from_str("hello");
    println!("Size of T: {}", size_of::<T>());
    println!("Size of Result: {}", size_of_val(&r));
    // 将来甚至应该可以直接用 let 语句进行模式匹配而不发生编译错误
    // 因为编译器有能力推理出 Err 分支没必要存在
    // let Ok(T(ref s)) = r;
    // println!("{}", s);
}
```

这里其实根本不需要考虑Err的情况，因为Err的类型是！，所以哪怕match语句中只有Ok分支，编译器也可以判定其为“完整匹配”。

8.3 再谈Option类型

Rust中的Option类型解决了许多编程语言中的“空指针”问题。

在目前的许多编程语言中，都存在一个很有意思的特殊指针（或者引用），它代表指向的对象为“空”，名字一般叫作null、nil、None、Nothing、nullptr等。这个空指针看似简单，但它引发的问题却一点也不少。空指针错误对许多朋友来说都不陌生，它在许多编程语言中都是极为常见的。以Java为例。我们有一个String类型的引用，String str=null；。如果它的值为null，那么接下来用它调用成员函数的时候，程序就会抛出一个NullPointerException。如果不catch住这个异常，整个程序就会crash掉。据说，这一类问题已经造成了业界无法估量的巨大损失。

在2009年的某次会议中，著名的“快速排序”算法的发明者Tony Hoare向全世界道歉，忏悔他曾经发明了“空指针”这个玩意儿。他是这样说的：

I call it my billion-dollar mistake.It was the invention of the null reference in 1965.At that time，I was designing the first comprehensive type system for references in an object oriented language（ALGOL W）.My goal was to ensure that all use of references should be absolutely safe，with checking performed automatically by the compiler.But I couldn’t resist the temptation to put in a null reference，simply because it was so easy to implement.This has led to innumerable errors，vulnerabilities，and system crashes，which have probably caused a billion dollars of pain and damage in the last forty years.

原来，在程序语言中加入空指针设计，其实并非是经过深思熟虑的结果，而仅仅是因为它很容易实现而已。这个设计的影响是如此深远，以至于后来许多编程语言都不假思索地继承了这一设计，范围几乎包括了目前业界所有流行的编程语言。

空指针最大的问题在于，它违背了类型系统的初衷。我们再来回忆一下什么是“类型”。按维基百科的定义，类型是：

Type is a classification identifying one of various types of data，that determines the possible values for that type，the operations that can be done on values of that type，the meaning of the data，and the way values of that type can be stored.

“类型”规定了数据可能的取值范围，规定了在这些值上可能的操作，也规定了这些数据代表的含义，还规定了这些数据的存储方式。

我们如果有一个类型Thing，它有一个成员函数doSomeThing（），那么只要是这个类型的变量，就一定应该可以调用doSomeThing（）函数，完成同样的操作，返回同样类型的返回值。

但是，null违背了这样的约定。一个正常的指针和一个null指针，哪怕它们是同样的类型，做同样的操作，所得到的结果也是不一样的。那么，凭什么说null指针和普通指针是一个类型呢？null实际上是在类型系统上打开了一个缺口，引入一个必须在运行期特殊处理的特殊“值”。它就像一个全局的、无类型的singleton变量一样，可以无处不在，可以随意与任意指针实现自动类型转换。它让编译器的类型检查在此处失去了意义。

那么，Rust里面的解决方案是什么呢？那就是，利用类型系统（ADT）将空指针和非空指针区别开来，分别赋予它们不同的操作权限，禁止针对空指针执行解引用操作。编译器和静态检查工具不可能知道一个变量在运行期的“值”，但是可以检查所有变量所属的“类型”，来判断它是否符合了类型系统的各种约定。如果我们把null从一个“值”上升为一个“类型”，那么静态检查就可以发挥其功能了。实际上早就已经有了这样的设计，叫作Option Type，并在scala、haskell、Ocaml、F#等许多程序设计语言中存在了许多年。

下面我们以Rust为例，介绍一下Option是如何解决空指针问题的。在Rust中，Option实际上只是一个标准库中普通的enum：

```
pub enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T)
}
```

Rust中的enum要求，在使用的时候必须“完整匹配”。意思是说，enum中的每一种可能性都必须处理，不能遗漏。对于一个可空的Option<T>类型，我们没有办法直接调用T类型的成员函数，要么用模式匹配把其中的类型T的内容“拆”出来使用，要么调用Option类型的成员方法。

而对于普通非空类型，Rust不允许赋值为None，也不允许不初始化就使用。在Rust中，也没有null这样的关键字。另外，Option类型参数可以是常见的指针类型，如Box和&等，也可以是非指针类型，它的表达能力其实已经超过了“可空的指针”这一种类型。

实际上，C++/C#等语言也发现了初始设计中的缺点，并开发了一些补救措施。C++标准库中加入了std：：optional<T>类型，C#中加入了System.Nullable<T>类型。可惜的是，受限于早期版本的兼容性，这些设计已经不能作为强制要求使用，因此其作用也就弱化了许多。

Option类型有许多非常方便的成员函数可供使用，另外我们还可以利用if-let、while-let等语法糖。许多情况下，没必要每次都动用match语句。

比如，map方法可以把一个Option<U>类型转为另外一个Option<V>类型：

```
let maybe_some_string = Some(String::from("Hello, World!"));
let maybe_some_len = maybe_some_string.map(|s| s.len());
assert_eq!(maybe_some_len, Some(13));
```

再比如，and_then方法可以把一系列操作串联起来：

```
fn sq(x: u32) -> Option<u32> { Some(x * x) }
fn nope(_: u32) -> Option<u32> { None }

assert_eq!(Some(2).and_then(sq).and_then(sq), Some(16));
```

而unwrap方法则是从Option<T>中提取出T。如果当前状态是None，那么这个函数会执行失败导致panic。正因为如此，除非是写不重要的小工具之类的，否则这个方法是不推荐使用的。哪怕你很确定此时Option的状态一定是Some，最好也用expect方法代替，至少它在发生

panic的时候还会打印出一个字符串，方便我们查找原因。

这个类型还有许多有用的方法，各位读者应该去查阅一下标准库的文档，对它的成员方法烂熟于心。

Option类型不仅在表达能力上非常优秀，而且运行开销也非常小。在这里我们还可以再次看到“零性能损失的抽象”能力。示例如下：

```
use std::mem::size_of;

fn main() {
    println!("size of isize            : {}",
        size_of::<isize>() );
    println!("size of Option<isize>    : {}",
        size_of::<Option<isize>>() );

    println!("size of &isize           : {}",
        size_of::<&isize>() );
    println!("size of Box<isize>       : {}",
        size_of::<Box<isize>>() );

    println!("size of Option<&isize>     : {}",
        size_of::<Option<&isize>>() );
    println!("size of Option<Box<isize>> : {}",
        size_of::<Option<Box<isize>>>() );

    println!("size of *const isize     : {}",
        size_of::<* const isize>() );
    println!("size of Option<*const isize> : {}",
        size_of::<Option<*const isize>>() );
}
```

这个示例分析了Option类型在执行阶段所占用的内存空间大小，结果为：

```
size of isize                    : 8
size of Option<isize>            : 16
size of &isize                   : 8
size of Box<isize>               : 8
size of Option<&isize>           : 8
size of Option<Box<isize>>       : 8
size of *const isize             : 8
size of Option<*const isize>     : 16
```

其中，不带Option的类型大小完全在意料之中。isize&isizeBox<isize>这几个类型占用空间的大小都等于该系统平台上一个指针占用的空间大小，不足为奇。Option<isize>类型实际表示的含义

是“可能为空的整数”，因此它除了需要存储一个isize空间的大小之外，还需要一个标记位（至少1bit）来表示该值存在还是不存在的状态。这里的结果是16，猜测可能是因为内存对齐的原因。

最让人惊奇的是，那两个“可空的指针”类型占用的空间竟然和一个指针占用的空间相同，并未多浪费一点点的空间来表示“指针是否为空”的状态。这是因为Rust在这里做了一个小优化：根据Rust的设计，借用指针&和所有权指针Box从语义上来说，都是不可能为“0”的状态。有些数值是不可能成为这几个指针指向的地址的，它们的取值范围实际上小于isize类型的取值范围。因此Option<&isize>和Option<Box<isize>>类型可以利用这个特点，使用“0”值代表当前状态为“空”。这意味着，在编译后的机器码层面，使用Option类型对指针的包装，与C/C++的指针完全没有区别。

Rust是如何做到这一点的呢？在目前的版本中，Rust设计了NonZero这样一个struct。（这个设计目前还处于不稳定状态，将来很可能会有变化，但是基本思路应该是不变的。）

```
/// A wrapper type for raw pointers and integers that will never be
/// NULL or 0 that might allow certain optimizations.
#[lang = "non_zero"]
#[derive(Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Debug, Hash)]
pub struct NonZero<T: Zeroable>(T);
```

它有一个attribute#[lang="..."]表明这个结构体是Rust语言的一部分。它是被编译器特殊处理的，凡是被这个结构体包起来的类型，编译器都将其视为“不可能为0”的。

我们再看一下Box<T>的定义：

```
#[lang = "owned_box"]
#[fundamental]
pub struct Box<T: ?Sized>(Unique<T>);
```

其中，Unique<T>的定义是：

```
pub struct Unique<T: ?Sized> {
    pointer: NonZero<*const T>,
    _marker: PhantomData<T>,
```

```
}
```

其中PhantomData<T>是一个零大小的类型pub struct Phantom-Data<T：？Sized>；，它的作用是在unsafe编程的时候辅助编译器静态检查，在运行阶段无性能开销，此处暂时略过。

把以上代码综合起来可以发现，Option<Box<T>>的实际内部表示形式是Option<NonZero<*const T>>。因此编译器就有能力将这个类型的占用空间压缩到与*const T类型占用的空间一致。

而对于*const T类型，它本身是有可能取值为0的，因此这种类型无法执行优化，Option<*const T>的大小就变成了两个指针大小。大家搞明白这一点后，我们自定义的类型如果也符合同样的条件，也可以利用这个特性来完成优化。

总结起来，对于Rust中的Option类型，读者需要注意以下几点。

1）如果从逻辑上说，我们需要一个变量确实是可空的，那么就应该显式标明其类型为Option<T>，否则应该直接声明为T类型。从类型系统的角度来说，这二者有本质区别，切不可混为一谈。

2）不要轻易使用unwrap方法。这个方法可能会导致程序发生panic。对于小工具来说无所谓，在正式项目中，最好是使用lint工具强制禁止调用这个方法。

3）相对于裸指针，使用Option包装的指针类型的执行效率不会降低，这是“零开销抽象”。

4）不必担心这样的设计会导致大量的match语句，使得程序可读性变差。因为Option<T>类型有许多方便的成员函数，再配合上闭包功能，实际上在表达能力和可读性上要更胜一筹。

第9章　宏

9.1　简介macro

“宏”（macro）是Rust的一个重要特性。Rust的“宏”（macro）是一种编译器扩展，它的调用方式为some_macro！（...）。宏调用与普通函数调用的区别可以一眼区分开来，凡是宏调用后面都跟着一个感叹号。宏也可以通过some_macro！[...]和some_macro！{...}两种语法调用，只要括号能正确匹配即可。我们在本书一开始就已经使用了“宏”，大家一定记得println！这个宏，它可以用于向标准输出打印字符串。

与C/C++中的宏不一样的是，Rust中的宏是一种比较安全的“卫生宏”（hygiene）。首先，Rust的宏在调用的时候跟函数有明显的语法区别；其次，宏的内部实现和外部调用者处于不同名字空间，它的访问范围严格受限，是通过参数传递进去的，我们不能随意在宏内访问和改变外部的代码。C/C++中的宏只在预处理阶段起作用，因此只能实现类似文本替换的功能。而Rust中的宏在语法解析之后起作用，因此可以获取更多的上下文信息，而且更加安全。

我们可以把“宏”视为“元编程”的一种方式。它是一种“生成程序的程序”。宏有很多用处。

9.1.1　实现编译阶段检查

比如我们用下面的方式调用println！宏：

```
fn main() {
    println!("number1 {} number2 {}");
}
```

编译器会产生一个编译错误“invalid reference to argument`0`（no arguments given）”。这是因为我们的第一个参数是一个字符串模板，它应该接受两个参数用于内部填充，可是我们在调用的时候，后面没有提供足够的参数，因此出错。这个功能如果使用普通函数来实现，是不可

能在编译阶段实现这样的错误检查功能的。使用宏，我们可以在编译阶段分析这个字符串常量和对应参数，确保它符合约定。另外一个常见的场景是，利用宏来检查正则表达式的正确性。

9.1.2 实现编译期计算

比如以下代码可以打印出当前源代码的文件名，以及当前代码的行数。这些信息都是纯编译阶段的信息。

```
fn main() {
    println!("file {} line {} ", file!(), line!());
}
```

在某些场景下，利用宏来完成一些编译期计算也是一种可行的选择。

9.1.3 实现自动代码生成

有些情况下，许多代码具有同样的“模式”，但是它们不能用现有的语法工具，如“函数”“泛型”“trait”等对其进行合理抽象。如果这样的boilerplate代码数量很多，实际上意味着代码违反了“Don’t Repeat Yourself”原则，那么我们可以用“宏”来精简代码，消除重复。

比如，在标准库中就有许多类似的用法。在core/ops.rs代码中，内置类型对各种运算符trait的支持就使用了宏。

```
add_impl! { usize u8 u16 u32 u64 isize i8 i16 i32 i64 f32 f64 }
```

这是各个内置类型实现std：：ops：：Add这个trait的办法。因为这些代码非常相似，所以可以将它们提取到一个“宏”里面，以避免无聊的重复。

9.1.4 实现语法扩展

某些情况下，我们可以使用宏来设计比较方便的“语法糖”，而不必使用编译器内部硬编码来实现。比如初始化一个动态数组，我们可以使用方便的vec！宏：

```
let v = vec![1, 2, 3, 4, 5];
```

简洁、直观、明了，而且不是编译器内部的“黑魔法”。我们可以充分发挥自己的想象力，通过自定义宏来增加语言的表达能力，甚至自定义DSL（Domain Specific Language）。

9.2 示范型宏

自定义宏有两种实现方式：

·通过标准库提供的macro_rules！宏实现。

·通过提供编译器扩展来实现。

编译器扩展只能在不稳定版本中使用。它的API正在重新设计中，还没有正式定稿，这就是所谓的macro 2.0。在后面，我们会体验macro 1.1，它就是macro 2.0的缩微版。

下面我们来使用一个例子讲解如何使用macro_rules！实现自定义宏。macro_rules！是标准库中为我们提供的一个编写简单宏的小工具，它本身也是用编译器扩展来实现的。它可以提供一种“示范型”（by example）宏编写方式。

举个例子，我们考虑一下这样的需求：提供一个hashmap！宏，实现如下初始化HashMap的功能：

```
let counts = hashmap!['A' => 0, 'C' => 0, 'G' => 0, 'T' => 0];
```

首先，定义hashmap这样一个宏名字：

```
macro_rules! hashmap {
}
```

在大括号里面，我们定义宏的使用语法，以及它展开后的形态。定义方式类似match语句的语法，expander=>{transcriber}。左边的是宏扩展的语法定义，后面是宏扩展的转换机制。语法定义的标识符以$开头，类型支持item、block、stmt、pat、expr、ty、itent、path、tt。我们的需求是需要一个表达式，一个“=>”标识符，再跟一个表达式，因此，宏可以写成这样：

```
macro_rules! hashmap {
```

```
    ($key: expr => $val: expr) => {
        // 暂时空着
        ()
    }
}
```

现在我们已经实现了一个hashmap！{'A'=>'1'}；这样的语法了。我们希望这个宏扩展开后的类型是HashMap，而且进行了合理的初始化，那么我们可以使用“语句块”的方式来实现：

```
macro_rules! hashmap {
    ($key: expr => $val: expr) => {
        {
        let mut map = ::std::collections::HashMap::new();
        map.insert($key, $val);
        map
        }
    }
}
```

现在我们希望在宏里面，可以支持重复多个这样的语法元素。我们可以使用+模式和*模式来完成。类似正则表达式的概念，+代表一个或者多个重复，*代表零个或者多个重复。因此，我们需要把需要重复的部分用括号括起来，并加上逗号分隔符：

```
macro_rules! hashmap {
    ($( $key: expr => $val: expr ),*) => {{
        let mut map = ::std::collections::HashMap::new();
        map.insert($key, $val);
        map
    }}
}
```

最后，我们在语法扩展的部分也使用*符号，将输入部分扩展为多条insert语句。最终的结果如下所示：

```
macro_rules! hashmap {
    ($( $key: expr => $val: expr ),*) => {{
        let mut map = ::std::collections::HashMap::new();
        $( map.insert($key, $val); )*
        map
    }}
}

fn main() {
    let counts = hashmap!['A' => 0, 'C' => 0, 'G' => 0, 'T' => 0];
```

```
    println!("{:?}", counts);
}
```

一个自定义宏就诞生了。如果我们想检查一下宏展开的情况是否正确，可以使用如下rustc的内部命令：

```
rustc -Z unstable-options --pretty=expanded temp.rs
```

可以看到，hashmap！宏展开后的结果是：

```
let counts =
        {
            let mut map = ::std::collections::HashMap::new();
            map.insert('A', 0);
            map.insert('C', 0);
            map.insert('G', 0);
            map.insert('T', 0);
            map
        };
```

很大一部分宏的需求我们都可以通过这种方式实现，它比较适合写那种一个模子套出来的重复代码。

9.3 宏1.1

对于一些简单的宏，这种“示例型”（by example）的方式足够使用了。但是更复杂的逻辑则需要通过更复杂的方式来实现，这就是所谓的“过程宏”（procedural macro）。它是直接用Rust语言写出来的，相当于一个编译器插件。但是编译器插件的最大问题是，它依赖于编译器的内部实现方式。一旦编译器内部有所变化，那么对应的宏就有可能出现编译错误，需要修改。因此，Rust中的“宏”一直难以稳定。

所以，Rust设计者希望提供一套相对稳定一点的API，它基本跟rustc的内部数据结构解耦。这个设计就是macro 2.0。这个功能目前暂时还没完成。但是，Rust提前推出了一个macro 1.1版本。我们在1.15正式版中可以体验一下这个功能的概貌。所谓的macro 1.1是按照2.0的思路专门为自动derive功能设计的，是一个缩微版的macro 2.0。

在Rust中，attribute也是一种特殊的宏。在编译器内部，attribute和macro并没有本质的区别，它们都是所谓的编译器扩展。在以后的macro 2.0中，我们也可以用类似的API设计自定义attribute。目前有一个叫作derive的attribute是最常用的，最需要支持自定义扩展。专门为支持自定义derive的功能，就是macro 1.1。derive功能我们在trait一章中已经讲过了，attribute可以让编译器帮我们自动impl某些trait。

目前，编译器的derive只支持一小部分固定的trait。但我们可以通过自定义宏实现扩展derive。下面，我们用一个示例来演示一下如何使用macro 1.1完成自定义#[derive（HelloWorld）]功能。

首先，需要把编译工具升级到最新的nightly版本。然后创建两个项目：一个是实现宏，一个使用宏。

```
$ cargo new --bin hello-world
$ cd hello-world
$ cargo new hello-world-derive
```

在hello-world-derive项目的Cargo.toml中，加上项目属性设置：

```
[lib]
```

```
proc-macro=true
```

在hello-world项目的Cargo.toml中，设置项目依赖：

```
[dependencies]
hello-world-derive = { path = "hello-world-derive" }
```

宏项目编译完成后，会生成一个动态链接库。这个库会被编译器在编译主项目的过程中调用。在主项目代码中写上如下测试代码：

```
#[macro_use]
extern crate hello_world_derive;

trait THelloWorld {
    fn hello();
}

#[derive(HelloWorld)]
struct FrenchToast;


fn main() {
    FrenchToast::hello();
}
```

接下来，我们来实现这个宏。它的代码骨架如下所示：

```
extern crate proc_macro;

use proc_macro::TokenStream;
use std::str::FromStr;

#[proc_macro_derive(HelloWorld)]
pub fn hello_world(input: TokenStream) -> TokenStream {
    // Construct a string representation of the type definition
    let s = input.to_string();

    TokenStream::from_str("").unwrap()
}
```

我们的主要逻辑就写在hello_world函数中，它需要用proc_macro_derive修饰。它的签名是，输入一个TokenStream，输出一个TokenStream。这个TokenStream类型目前还没实现什么有用的成员方法，暂时只提供了和字符串类型之间的转换方式。我们在函数中把input

的值打印出来：

```
let s = input.to_string();
println!("{}", s);
```

编译可见，输出值为struct FrenchToast；。由此可见，编译器将#[derive（）]宏修饰的部分作为参数，传递给了我们这个编译器扩展函数。我们需要对这个参数进行分析，然后将希望自动生成的代码作为返回值传递出去。

在这里，我们引入regex库来辅助实现逻辑。在项目文件中，加入以下代码：

```
[dependencies]
regex = "0.2"
```

然后写一个函数，把类型名字从输入参数中提取出来：

```
fn parse_struct_name(s: &str) -> String {
    let r = Regex::new(r"(?:struct\s+)([\w\d_]+)").unwrap();
    let caps = r.captures(s).unwrap();
    caps[1].to_string()
}

#[test]
fn test_parse_struct_name() {
    let input = "struct Foo(i32);";
    let name = parse_struct_name(input);
    assert_eq!(&name, "Foo");
}
```

接下来，就可以自动生成我们的impl代码了：

```
#[proc_macro_derive(HelloWorld)]
pub fn hello_world(input: TokenStream) -> TokenStream {
    let s = input.to_string();
    let name = parse_struct_name(&s);
    let output = format!(r#"
impl THelloWorld for {0} {{
    fn hello() {{ println!(" {0} says hello "); }}
}}"#, name);

    TokenStream::from_str(&output).unwrap()
}
```

我们构造了一个字符串，然后将这个字符串转化为TokenStream类型返回。

编译主项目可见，FrenchToast类型已经有了一个hello（）方法，执行结果为：

```
FrenchToast says hello
```

在macro 1.1版本中，只提供了这么一点简单的API。在接下来的macro 2.0版本中，会为TokenStream添加一些更有用的方法，或许那时候就没必要把TokenStream转成字符串再自己解析一遍了。

第二部分　内存安全

不带自动内存回收（Garbage Collection）的内存安全是Rust语言最重要的创新，是它与其他语言最主要的区别所在，是Rust语言设计的核心。

Rust希望通过语言的机制和编译器的功能，把程序员易犯错、不易检查的问题解决在编译期，避免运行时的内存错误。这一部分主要探讨Rust是如何达到内存安全特性的。

Languages shape the way we think，or don’t.

——Erik Naggum

第10章　内存管理基础

本章主要介绍没有垃圾回收机制下的内存管理基础知识。

10.1 堆和栈

一个进程在执行的时候，它所占用的内存的虚拟地址空间一般被分割成好几个区域，我们称为“段”（Segment）。常见的几个段如下。

·代码段。编译后的机器码存在的区域。一般这个段是只读的。

·bss段。存放未初始化的全局变量和静态变量的区域。

·数据段。存放有初始化的全局变量和静态变量的区域。

·函数调用栈（call stack segment）。存放函数参数、局部变量以及其他函数调用相关信息的区域。

·堆（heap）。存放动态分配内存的区域。

函数调用栈（call stack）也可以简称为栈（stack）。因为函数调用栈本来就是基于栈这样一个数据结构实现的。它具备“后入先出”（LIFO）的特点。最先进入的数据也是最后出来的数据。一般来说，CPU有专门的指令可以用于入栈或者出栈的操作。当一个函数被调用时，就会有指令把当前指令的地址压入栈内保存起来，然后跳转到被调用的函数中执行。函数返回的时候，就会把栈里面先前的指令地址弹出来继续执行，如图10-1所示。

堆是为动态分配预留的内存空间，如图10-2所示。和栈不一样，从堆上分配和重新分配块没有固定模式，用户可以在任何时候分配和释放它。这样就使得跟踪哪部分堆已经被分配和被释放变得异常复杂；有许多定制的堆分配策略用来为不同的使用模式下调整堆的性能。堆是在内存中动态分配的内存，是无序的。每个线程都有一个栈，但是每一个应用程序通常都只有一个堆。在堆上的变量必须要手动释放，不存在作用域的问题。

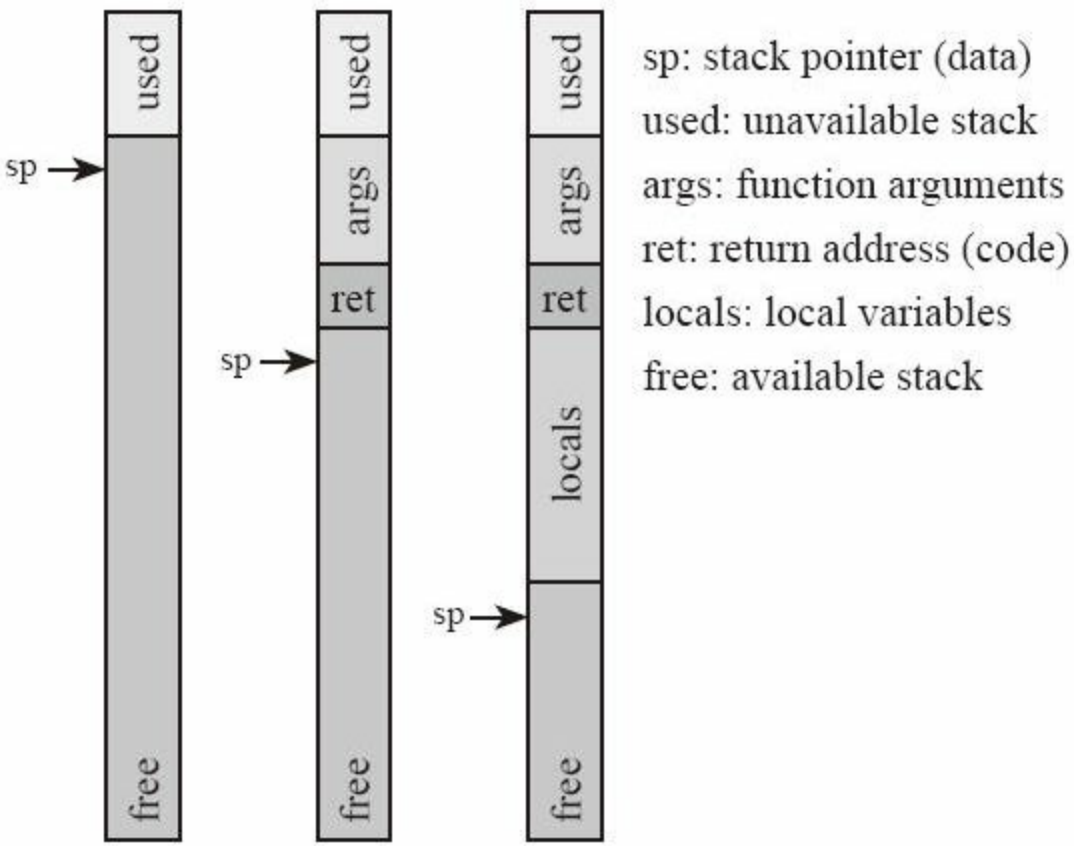

used
sp
free
used
args
ret
sp
free
used
args
ret
locals
sp
free
sp: stack pointer (data)
used: unavailable stack
args: function arguments
ret: return address (code)
locals: local variables
free: available stack

图　10-1 [1]

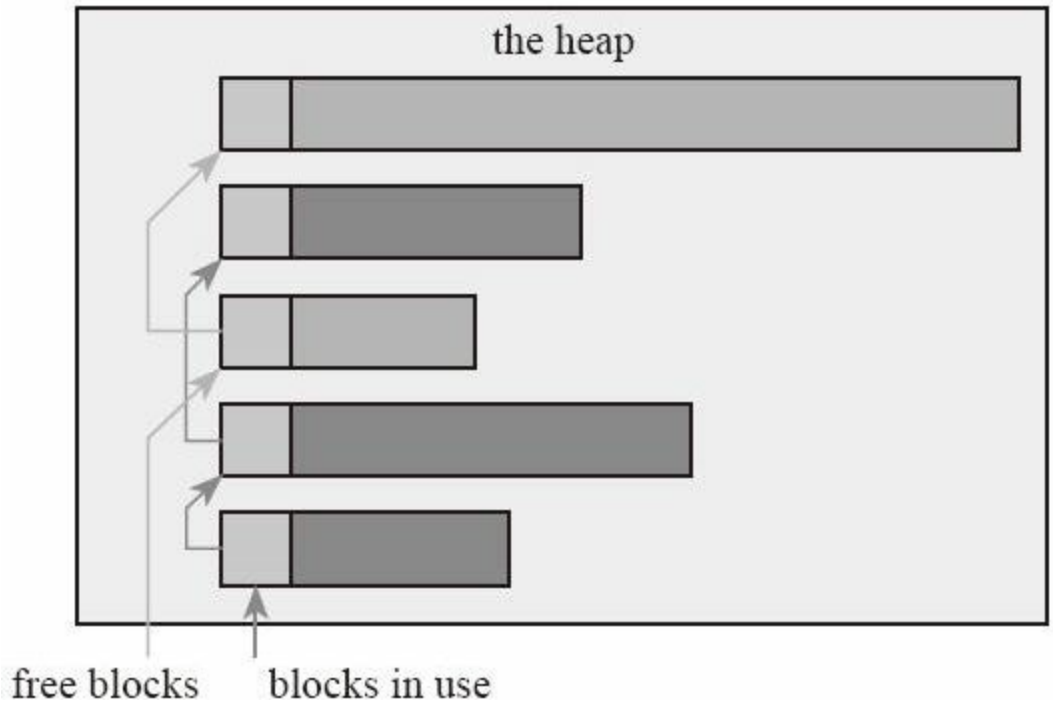

the heap
free blocks
blocks in use

图　10-2 [2]

一般来说，操作系统提供了在堆上分配和释放内存的系统调用，但是用户不是直接使用这个系统调用，而是使用封装的更好的“内存分配器”（Allocator）。比如，在C语言里面，运行时（runtime）就提供了malloc和free这两个函数可以管理堆内存。

堆和栈之间的区别有：

·栈上保存的局部变量在退出当前作用域的时候会自动释放；

·堆上分配的空间没有作用域，需要手动释放；

·一般栈上分配的空间大小是编译阶段就可以确定的（C语言里面的VLA除外）；

·栈有一个确定的最大长度，超过了这个长度会产生“栈溢出”（stack overflow）；

·堆的空间一般要更大一些，堆上的内存耗尽了，就会产生“内存分配不足”（out of memory）。

[1] 此图源于https://stackoverfl ow.com/questions/79923/what-and-where-are-the-stack-and-heap。
[2] 此图源于https://stackoverfl ow.com/questions/79923/what-and-where-are-the-stack-and-heap。

10.2 段错误

我们再回忆一下Rust的主要特点。Rust官方网站给自己的介绍是：

Rust is a systems programming language that runs blazingly fast，prevents segfaults，and guarantees thread safety.

本节就来讲一下什么是segfault，以及什么是Rust所谈论的“内存安全”概念。

segfault实际上是“segmentation fault”的缩写形式，我们可以翻译为“段错误”。segfault是这样形成的：进程空间中的每个段通过硬件MMU映射到真正的物理空间；在这个映射过程中，我们还可以给不同的段设置不同的访问权限，比如代码段就是只能读不能写；进程在执行过程中，如果违反了这些权限，CPU会直接产生一个硬件异常；硬件异常会被操作系统内核处理，一般内核会向对应的进程发送一条信号；如果没有实现自己特殊的信号处理函数，默认情况下，这个进程会直接非正常退出；如果操作系统打开了core dump功能，在进程退出的时候操作系统会把它当时的内存状态、寄存器状态以及各种相关信息保存到一个文件中，供用户以后调试使用。

在传统系统级编程语言C/C++里面，制造segfault是很容易的。程序员需要非常小心才能避免这种错误，这也是为什么会有那么多的代码标准来规范程序员的行为。而另外一类编程语言规避segfault的办法是使用自动垃圾回收机制。在这些编程语言中，指针的能力被大幅限制，内存分配和释放都在一个运行时环境中被严格管理。当然，这么做也付出了一定的代价。某些应用场景下用这样的代价换取开发效率和安全性是非常划算的，而在某些应用场景下这样的代价是不可接受的。

Rust的主要设计目标之一，是在不用自动垃圾回收机制的前提下避免产生segfault。从这个意义上来说，它是独一无二的。至于它是如何做到的，本书将会详细剖析。

10.3 内存安全

在谈到Rust的时候，经常会提到的一个概念，那就是“内存安全”（Memory safety）。内存安全是Rust设计的主要目标之一，因此我们有必要把这个概念做一个澄清，让大家能更清楚地理解Rust为什么要这么设计。

在Wikipedia上，内存安全是这么定义的：

Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access，such as buffer overflows and dangling pointers.

下面列举一系列的“内存不安全”的例子。以下这些情况，就是Rust想要避免的问题。

·空指针

解引用空指针是不安全的。这块地址空间一般是受保护的，对空指针解引用在大部分平台上会产生segfault。

·野指针

野指针指的是未初始化的指针。它的值取决于它这个位置以前遗留下来的是什么值。所以它可能指向任意一个地方。对它解引用，可能会造成segfault，也可能不会，纯粹凭运气。但无论如何，这个行为都不会是你预期内的行为，是一定会产生bug的。

·悬空指针

悬空指针指的是内存空间在被释放了之后，继续使用。它跟野指针类似，同样会读写已经不属于这个指针的内容。

·使用未初始化内存

不只是指针类型，任何一种类型不初始化就直接使用都是危险的，造成的后果我们完全无法预测。

·非法释放

内存分配和释放要配对。如果对同一个指针释放两次，会制造出内存错误。如果指针并不是内存分配器返回的值，对其执行释放操作，也是危险的。

·缓冲区溢出

指针访问越界了，结果也是类似于野指针，会读取或者修改临近内存空间的值，造成危险。

·执行非法函数指针

如果一个函数指针不是准确地指向一个函数地址，那么调用这个函数指针会导致一段随机数据被当成指令来执行，是非常危险的。

·数据竞争

在有并发的场景下，针对同一块内存同时读写，且没有同步措施。

以上这些问题都是极度危险的，而且它们并不一定会在发生的时候就被发现并立即终止。它们不一定会直接触发core dump，有可能程序一直带病运行，只是结果一直有bug但却无法找到原因，因为真正的原因与表现之间没有任何肉眼可见的关联关系。它们有可能造成非常随机的、难以复现和难以调试的诡异bug，就像武林高手一样神出鬼没，行踪不定。它们也可能在经过许多步骤之后最终触发core dump，可惜此时早已不是案发第一现场，修复这种bug的难度极高。

在Rust语境中，还有一些内存错误是不算在“内存安全”范畴内的，比如内存泄漏以及内存耗尽。内存泄漏显然是一种bug，但是它不会直接造成非常严重的后果，至少比上面列出的那些错误危险性要低一些，解决的办法也是完全不一样的。同样，内存耗尽也不是事关安全性的问题，出现内存耗尽的时候，Rust程序的行为依然是确定性的和可控的（目前版本下，如果内存耗尽则发生panic，也有人认为在这种情况发生的时候，应该给个机会由用户自己处理，这种情况后面应该会有改进）。

另外，panic也不属于内存安全相关的问题。在后面我们会花很多篇

幅来讲解panic。panic和core dump之间有重要区别。panic是发生不可恢复错误后，程序主动执行的一种错误处理机制；而core dump则是程序失控之后，触发了操作系统的保护机制而被动退出的。发生panic的时候，此处就是确定性的第一现场，我们可以根据call stack信息很快找到事发地点，然后修复。panic是防止更严重内存安全错误的重要机制。

第11章　所有权和移动语义

11.1　什么是所有权

拿C语言的代码来打个比方。我们可能会在堆上创建一个对象，然后使用一个指针来管理这个对象：

```
Foo *p = make_object("args");
```

接下来，我们可能需要使用这个对象：

```
use_object(p);
```

然而，这段代码之后，谁能猜得到，指针p指向的对象究竟发生了什么？它是否被修改过了？它还存在吗，是否已经被释放？是否有另外一个指针现在也同时指向这个对象？我们还能继续读取、修改或者释放这个对象吗？实际上，除了去了解use_object的内部实现之外，我们没办法回答以上问题。

对此，C++进行了一个改进，即通过“智能指针”来描述“所有权”（Ownership）概念。这在一定程度上减少了内存使用bug，实现了“半自动化”的内存管理。而Rust在此基础上更进一步，将“所有权”的理念直接融入到了语言之中。

“所有权”代表着以下意义：

·每个值在Rust中都有一个变量来管理它，这个变量就是这个值、这块内存的所有者；

·每个值在一个时间点上只有一个管理者；

·当变量所在的作用域结束的时候，变量以及它代表的值将会被销毁。

拿前面已经讲过的字符串String类型来举例：

```
fn main() {
    let mut s = String::from("hello");
    s.push_str(" world");
    println!("{}", s);
}
```

当我们声明一个变量s，并用String类型对它进行初始化的时候，这个变量s就成了这个字符串的“所有者”。如果我们希望修改这个变量，可以使用mut修饰s，然后调用String类型的成员方法来实现。当main函数结束的时候，s将会被析构，它管理的内存（不论是堆上的，还是栈上的）则会被释放。

我们一般把变量从出生到死亡的整个阶段，叫作一个变量的“生命周期”。比如这个例子中的局部变量s，它的生命周期就是从let语句开始，到main函数结束。

在上述示例的基础上，若做一点修改：

```
fn main() {
    let s = String::from("hello");
    let s1 = s;
    println!("{}", s);
}
```

编译，可见：

```
error[E0382]: use of moved value: `s`
 --> test.rs:5:20
  |
4 |     let s1 = s;
  |         -- value moved here
5 |     println!("{}", s);
  |                    ^ value used here after move
  |
  = note: move occurs because `s` has type `std::string::String`, which does not imp
```

这里出现了编译错误。编译器显示，在let s1=s；语句中，原本由s拥有的字符串已经转移给了s1这个变量。所以，后面继续使用s是不对的。

也就是前面所说的每个值只有一个所有者。变量s的生命周期从声明开始，到move给s1就结束了。变量s1的生命周期则是从它声明开始，到函数结束。而字符串本身，由String：：from函数创建出来，到函数结束的时候就会销毁。中间所有权的转换，并不会将这个字符串本身重新销毁再创建。在任意时刻，这个字符串只有一个所有者，要么是s，要么是s1。

请注意，Rust的赋值和C++的赋值有重大区别。如果我们用C++来实现上面这个例子的话，程序如下：

```
#include <iostream>

using namespace std;

int main() {
    string s("hello");
    string s1 = s;
    cout << s << endl;
    cout << s1 << endl;
    return 0;
}
```

在用变量s初始化s1的时候，并不会造成s的生命周期结束。这里只会调用string类型的复制构造函数复制出一个新的字符串，于是在后面s1和s都是合法变量。在Rust中，我们要模拟这一行为，需要手动调用clone（）方法来完成：

```
fn main() {
    let s = String::from("hello");
    let s1 = s.clone();
    println!("{} {}", s, s1);
}
```

在Rust里面，不可以做“赋值运算符重载”，若需要“深复制”，必须手工调用clone方法。这个clone方法来自于std：：clone：：Clone这个trait。clone方法里面的行为是可以自定义的。

11.2 移动语义

一个变量可以把它拥有的值转移给另外一个变量，称为“所有权转移”。赋值语句、函数调用、函数返回等，都有可能导致所有权转移。

比如：

```
fn create() -> String {
    let s = String::from("hello");
    return s; // 所有权转移,从函数内部移动到外部
}

fn consume(s: String) { // 所有权转移,从函数外部移动到内部
    println!("{}", s);
}

fn main() {
    let s = create();
    consume(s);
}
```

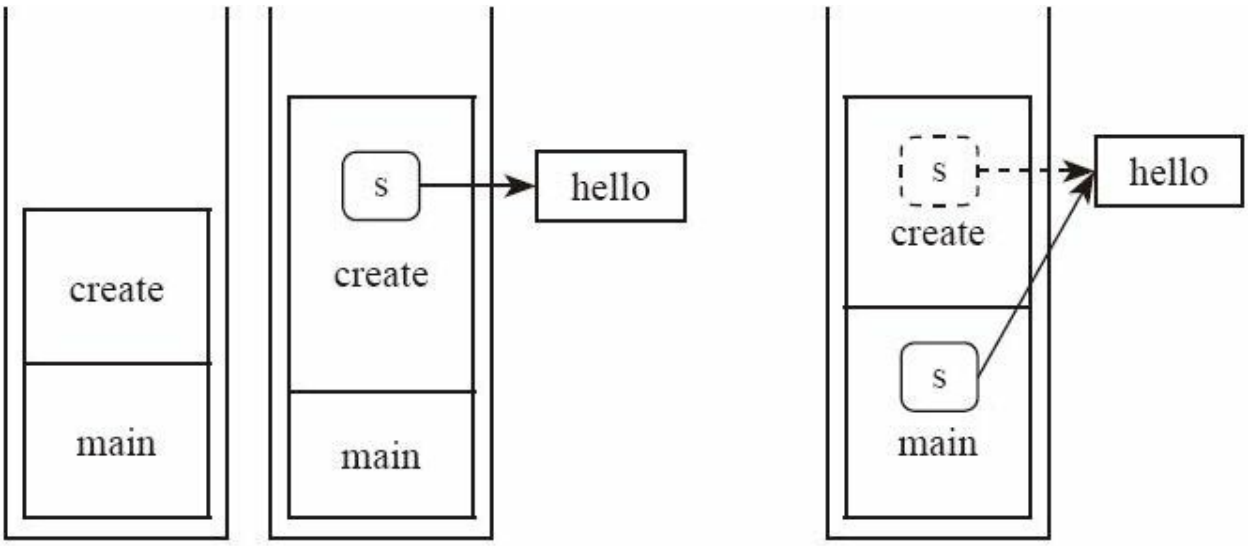
在上面这个例子中，函数参数、函数返回都发生了所有权转移，转移的过程可以用图11-1表示。

所有权转移的步骤分解如下。

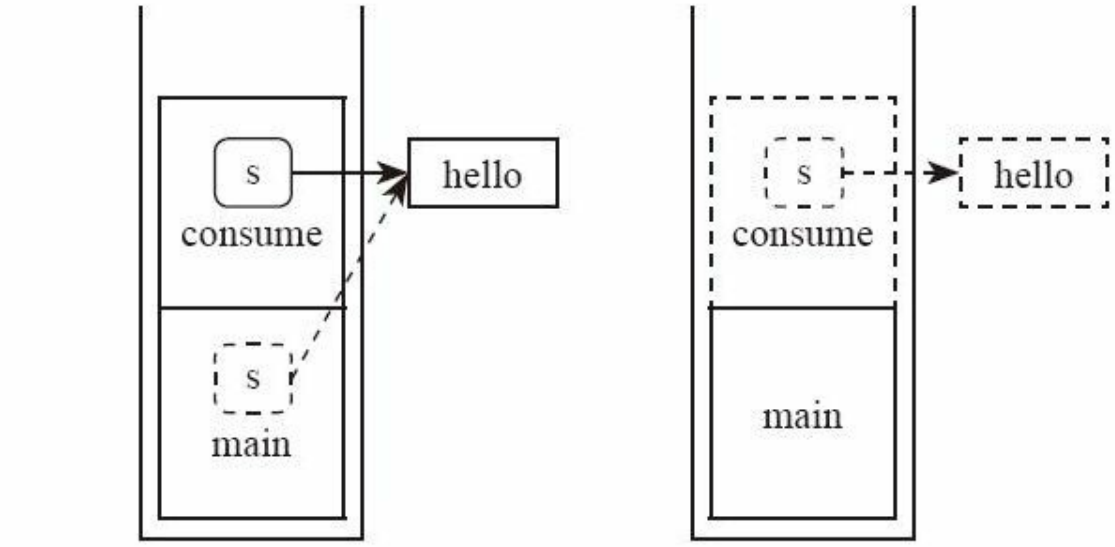
1）main函数调用create函数。

2）在调用create函数的时候创建了字符串，在栈上和堆上都分配有内存。局部变量s是这些内存的所有者。

3）create函数返回的时候，需要将局部变量s移动到函数外面，这个过程就是简单地按字节复制memcpy。



a）函数返回所有权转移



b）函数参数所有权转移

图　11-1

4）同理，在调用consume函数的时候，需要将main函数中的局部变量转移到consume函数，这个过程也是简单地按字节复制memcpy。

5）当consume函数结束的时候，它并没有把内部的局部变量再转移出来，这种情况下，consume内部局部变量的生命周期就该结束了。这个局部变量s生命周期结束的时候，会自动释放它所拥有的内存，因此字符串也就被释放了。

Rust中所有权转移的重要特点是，它是所有类型的默认 语义。这是许多读者一开始不习惯的地方。这里再重复一遍，请大家牢牢记住，Rust中的变量绑定操作，默认是move语义，执行了新的变量绑定后，原来的变量就不能再被使用！一定要记住！

Rust的这一规定非常有利于编译器静态检查。与之相对的，C++的做法就不一样了，它允许赋值构造函数、赋值运算符重载，因此在出现“构造”或者“赋值”操作的时候，有可能表达的是完全不同的含义，这取决于程序员如何实现重载。C++的这个设计具有巨大的灵活性，但是不恰当的实现也可能造成内存不安全。而Rust的这一设计大幅降低了语言的复杂度，“移动语义”不可能执行用户的自定义代码，没有任何内存安全风险，而且满足异常安全。在C++里面，std：：vector<int>v1=v2；是复制语义，而Rust里面的let v1：Vec<i32>=v2；是移动语义。如果要在Rust里面实现复制语义，需要显式写出函数调用let v1：Vec<i32>=v2.clone（）；。如果我们在C++中实现移动语义，则需要用户自定义实现移动构造函数及移动赋值运算符。

对于“移动语义”，最后还需要强调的一点是，“语义”不代表最终的执行效率。“语义”只是规定了什么样的代码是编译器可以接受的，以及它执行后的效果可以用怎样的思维模型去理解。编译器有权在不改变语义的情况下做任何有利于执行效率的优化。语义和优化是两个阶段的事情。我们可以把移动语义想象成执行了一个memcpy，但真实的汇编代码未必如此。比如：

```
fn create() -> BigObject {
    let local = …;
    return local;
}
let v = create();
```

完全可能被优化为类似如下的效果：

```
fn create(p: &mut BigObject) {
    ptr::write(p, …);
}
let mut v: BigObject = uninitialized();
create(&mut v);
```

编译器可以提前在当前调用栈中把大对象的空间分配好，然后把这

个对象的指针传递给子函数，由子函数执行这个变量的初始化。这样就避免了大对象的复制工作，参数传递只是一个指针而已。这么做是完全满足移动语义要求的，而且编译器还有权利做更多类似的优化。

11.3 复制语义

默认的move语义是Rust的一个重要设计，但是任何时候需要复制都去调用clone函数会显得非常烦琐。对于一些简单类型，比如整数、bool，让它们在赋值的时候默认采用复制操作会让语言更简单。

比如下面这个程序就可以正常编译通过：

```
fn main() {
    let v1 : isize = 0;
    let v2 = v1;
    println!("{}", v1);
}
```

编译器并没有阻止v1被使用，这是为什么呢？

因为在Rust中有一部分“特殊照顾”的类型，其变量绑定操作是copy语义。所谓的copy语义，是指在执行变量绑定操作的时候，v2是对v1所属数据的一份复制。v1所管理的这块内存依然存在，并未失效，而v2是新开辟了一块内存，它的内容是从v1管理的内存中复制而来的。和手动调用clone方法效果一样，let v2=v1；等效于let v2=v1.clone（）；。

使用文件系统来打比方。copy语义就像“复制、粘贴”操作。操作完成后，原来的数据依然存在，而新的数据是原来数据的复制品。move语义就像“剪切、粘贴”操作。操作完成后，原来的数据就不存在了，被移动到了新的地方。这两个操作本身是一样的，都是简单的内存复制，区别在于复制完以后，原先那个变量的生命周期是否结束。

Rust中，在普通变量绑定、函数传参、模式匹配等场景下，凡是实现了std：：marker：：Copy trait的类型，都会执行copy语义。基本类型，比如数字、字符、bool等，都实现了Copy trait，因此具备copy语义。

对于自定义类型，默认是没有实现Copy trait的，但是我们可以手动添上。示例如下：

```
struct Foo {
    data : i32
}

impl Copy for Foo {}

fn main() {
    let v1 = Foo { data : 0 };
    let v2 = v1;
    println!("{:?}", v1.data);
}
```

编译错误。错误信息是“the trait`core：：clone：：Clone`is not implemented for the type`Foo`”。查一下文档发现，原来Copy继承了Clone，我们要实现Copy trait必须同时实现Clone trait。把代码改成：

```
struct Foo {
    data : i32
}

impl Clone for Foo {
    fn clone(&self) -> Foo {
        Foo { data : self.data }
    }
}

impl Copy for Foo {}

fn main() {
    let v1 = Foo { data : 0 };
    let v2 = v1;
    println!("{:?}", v1.data);
}
```

编译通过。现在Foo类型也拥有了复制语义。在执行变量绑定、函数参数传递的时候，原来的变量不会失效，而是会新开辟一块内存，将原来的数据复制过来。

绝大部分情况下，实现Copy trait和Clone trait是一个非常机械化的、重复性的工作，clone方法的函数体要对每个成员调用一下clone方法。Rust提供了一个编译器扩展derive attribute，来帮我们写这些代码，其使用方式为#[derive（Copy，Clone）]。只要一个类型的所有成员都具有Clone trait，我们就可以使用这种方法来让编译器帮我们实现Clone trait了。

示例如下：

```
#[derive(Copy, Clone)]
struct Foo {
    data : i32
}

fn main() {
    let v1 = Foo { data : 0 };
    let v2 = v1;
    println!("{:?}", v1.data);
}
```

11.4 Box类型

Box类型是Rust中一种常用的指针类型。它代表“拥有所有权的指针”，类似于C++里面的unique_ptr（严格来说，unique_ptr<T>更像Option<Box<T>>）。用法如下：

```
struct T{
    value: i32
}

fn main() {
    let p = Box::new(T{value: 1});
    println!("{}", p.value);
}
```

Box类型永远执行的是move语义，不能是copy语义。原因大家想想就可以明白，Rust中的copy语义就是浅复制。对于Box这样的类型而言，浅复制必然会造成二次释放问题。

对于Rust里面的所有变量，在使用前一定要合理初始化，否则会出现编译错误。对于Box<T>/&T/&mut T这样的类型，合理初始化意味着它一定指向了某个具体的对象，不可能是空。如果用户确实需要“可能为空的”指针，必须使用类型Option<Box<T>>。

Rust里面还有一个保留关键字box（注意是小写）。它可以用于把变量“装箱”到堆上。目前这个语法依然是unstable状态，需要打开feature gate才能使用，示例如下：

```
#![feature(box_syntax)]

struct T{
    value: i32
}

fn main() {
    let p : Box<T> = box T{value: 1};
    println!("{}", p.value);
}
```

这种写法和Box：：new（）函数调用并没有本质区别。将来box关

键字可能会同样支持各种智能指针，从而根据上下文信息自动判断执行。比如let p：Rc<T>=box T{value：1}；就可以构造一个Rc指针。

11.5 Clone VS.Copy

Rust中的Copy是一个特殊的trait，它给类型提供了“复制”语义。在Rust标准库里面，还有一个跟它很相近的trait，叫作Clone。很多人容易把这两者混淆，本节专门来谈一谈这两个trait。

11.5.1 Copy的含义

Copy的全名是std：：marker：：Copy。请大家注意，std：：marker模块里面所有的trait都是特殊的trait。目前稳定的有四个，它们是Copy、Send、Sized、Sync。它们的特殊之处在于：它们是跟编译器密切绑定的，impl这些trait对编译器的行为有重要影响。在编译器眼里，它们与其他的trait不一样。这几个trait内部都没有方法，它们的唯一任务是给类型打一个“标记”，表明它符合某种约定——这些约定会影响编译器的静态检查以及代码生成。

Copy这个trait在编译器的眼里代表的是什么意思呢？简单点总结就是，如果一个类型impl了Copy trait，意味着任何时候，我们都可以通过简单的内存复制（在C语言里按字节复制memcpy）实现该类型的复制，并且不会产生任何内存安全问题。

一旦一个类型实现了Copy trait，那么它在变量绑定、函数参数传递、函数返回值传递等场景下，都是copy语义，而不再是默认的move语义。

下面用最简单的赋值语句x=y来说明move语义和copy语义的根本区别。move语义是“剪切、粘贴”操作，变量y把所有权递交给了x之后，y就彻底失效了，后面继续使用y就会出编译错误。而copy语义是“复制、粘贴”操作，变量y把所有权递交给了x之后，它自己还留了一个副本，在这句赋值语句之后，x和y依然都可以继续使用。

在Rust里，move语义和copy语义具体执行的操作，是不允许由程序员自定义的，这是它和C++的巨大区别。这里没有赋值构造函数或者赋值运算符重载。move语义或者copy语义都是执行的memcpy，无法更改，这个过程中绝对不存在其他副作用。当然，这里一直谈的是“语

义”，而没有涉及编译器优化。从语义的角度，我们要讲清楚，什么样的代码在编译器看来是合法的，什么样的代码是非法的。如果考虑后端优化，在许多情况下，不必要的内存复制实际上已经彻底优化掉了，大家不必担心执行效率的问题。也没有必要每次都把move或者copy操作与具体的汇编代码联系起来，因为场景不同，优化结果不同，生成的代码也是不同的。大家只需记住的是语义。

11.5.2 Copy的实现条件

并不是所有的类型都可以实现Copy trait。Rust规定，对于自定义类型，只有所有成员都实现了Copy trait，这个类型才有资格实现Copy trait。

常见的数字类型、bool类型、共享借用指针&，都是具有Copy属性的类型。而Box、Vec、可写借用指针&mut等类型都是不具备Copy属性的类型。

对于数组类型，如果它内部的元素类型是Copy，那么这个数组也是Copy类型。

对于元组tuple类型，如果它的每一个元素都是Copy类型，那么这个tuple也是Copy类型。

struct和enum类型不会自动实现Copy trait。只有当struct和enum内部的每个元素都是Copy类型时，编译器才允许我们针对此类型实现Copy trait。比如下面这个类型，虽然它的成员是Copy类型，但它本身不是Copy类型：

```
struct T(i32);

fn main() {
    let t1 = T(1);
    let t2 = t1;
    println!("{} {}", t1.0, t2.0);
}
```

编译可见编译错误。原因是在let t2=t1；这条语句中执行的是move语义。但是我们可以手动为它impl Copy trait，这样它就具备了copy语

义。

我们可以认为，Rust中只有POD（C++语言中的Plain Old Data）类型才有资格实现Copy trait。在Rust中，如果一个类型只包含POD数据类型的成员，并且没有自定义析构函数，那它就是POD类型。比如：整数、浮点数、只包含POD类型的数组等，都属于POD类型；而Box String Vec等不能按字节复制的类型，都不属于POD类型。但是，反过来讲，也并不是所有满足POD的类型都应该实现Copy trait，是否实现Copy取决于业务需求。

11.5.3 Clone的含义

Clone的全名是std：：clone：：Clone。它的完整声明如下：

```
pub trait Clone : Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

它有两个关联方法，分别是clone_from和clone，clone_from是有默认实现的，依赖于clone方法的实现。clone方法没有默认实现，需要手动实现。

clone方法一般用于“基于语义的复制”操作。所以，它做什么事情，跟具体类型的作用息息相关。比如，对于Box类型，clone执行的是“深复制”；而对于Rc类型，clone做的事情就是把引用计数值加1。

虽然Rust中的clone方法一般是用来执行复制操作的，但是如果在自定义的clone函数中做点别的什么工作，编译器也没办法禁止。你可以根据需要在clone函数中编写任意的逻辑。

但是有一条规则需要注意：对于实现了copy的类型，它的clone方法应该跟copy语义相容，等同于按字节复制。

11.5.4 自动derive

绝大多数情况下，实现Copy Clone这样的trait都是一个重复而无聊的工作。因此，Rust提供了一个attribute，让我们可以利用编译器自动生成这部分代码。示例如下：

```
#[derive(Copy, Clone)]
struct MyStruct(i32);
```

这里的derive会让编译器帮我们自动生成impl Copy和impl Clone这样的代码。自动生成的clone方法，会依次调用每个成员的clone方法。

通过derive方式自动实现Copy和手工实现Copy有微小的区别。当类型具有泛型参数的时候，比如struct MyStruct<T>{}，通过derive自动生成的代码会自动添加一个T：Copy的约束。

目前，只有一部分固定的特殊trait可以通过derive来自动实现。将来Rust会允许自定义的derive行为，让我们自己的trait也可以通过derive的方式自动实现。

11.5.5 总结

Copy和Clone两者的区别和联系如下。

·Copy内部没有方法，Clone内部有两个方法。

·Copy trait是给编译器用的，告诉编译器这个类型默认采用copy语义，而不是move语义。Clone trait是给程序员用的，我们必须手动调用clone方法，它才能发挥作用。

·Copy trait不是想实现就能实现的，它对类型是有要求的，有些类型不可能impl Copy。而Clone trait则没有什么前提条件，任何类型都可以实现（unsized类型除外，因为无法使用unsized类型作为返回值）。

·Copy trait规定了这个类型在执行变量绑定、函数参数传递、函数返回等场景下的操作方式。即这个类型在这种场景下，必然执行的是“简单内存复制”操作，这是由编译器保证的，程序员无法控制。Clone trait里面的clone方法究竟会执行什么操作，则是取决于程序员自己写的逻辑。一般情况下，clone方法应该执行一个“深复制”操作，但这

不是强制性的，如果你愿意，在里面启动一个人工智能程序都是有可能的。

·如果你确实不需要Clone trait执行其他自定义操作（绝大多数情况都是这样），编译器提供了一个工具，我们可以在一个类型上添加#[derive（Clone）]，来让编译器帮我们自动生成那些重复的代码。编译器自动生成的clone方法非常机械，就是依次调用每个成员的clone方法。

·Rust语言规定了在T：Copy的情况下，Clone trait代表的含义。即：当某变量t：T符合T：Copy时，它调用t.clone（）方法的含义必须等同于“简单内存复制”。也就是说，clone的行为必须等同于let x=std：：ptr：：read（&t）；，也等同于let x=t；。当T：Copy时，我们不要在Clone trait里面乱写自己的逻辑。所以，当我们需要指定一个类型是Copy的时候，最好使用#[derive（Copy，Clone）]方式，避免手动实现Clone导致错误。

11.6 析构函数

所谓“析构函数”（destructor），是与“构造函数”（constructor）相对应的概念。“构造函数”是对象被创建的时候调用的函数，“析构函数”是对象被销毁的时候调用的函数。

Rust中没有统一的“构造函数”这个语法，对象的构造是直接对每个成员进行初始化完成的，我们一般将对象的创建封装到普通静态函数中。

相对于构造函数，析构函数有更重要的作用。它会在对象消亡之前由编译器自动调用，因此特别适合承担对象销毁时释放所拥有的资源的作用。比如，Vec类型在使用的过程中，会根据情况动态申请内存，当变量的生命周期结束时，就会触发该类型的析构函数的调用。在析构函数中，我们就有机会将所拥有的内存释放掉。在析构函数中，我们还可以根据需要编写特定的逻辑，从而达到更多的目的。析构函数不仅可以用于管理内存资源，还能用于管理更多的其他资源，如文件、锁、socket等。

在C++中，利用变量生命周期绑定资源的使用周期，已经是一种常用的编程惯例。此手法被称为RAII（Resource Acquisition Is Initialization）。在变量生命周期开始时申请资源，在变量生命周期结束时利用析构函数释放资源，从而达到自动化管理资源的作用，很大程度上减少了资源的泄露和误用。

在Rust中编写“析构函数”的办法是impl std：：ops：：Drop。Drop trait的定义如下：

```
trait Drop {
    fn drop(&mut self);
}
```

Drop trait允许在对象即将消亡之时，自行调用指定代码。我们来写一个自带析构函数的类型。示例如下：

```
use std::ops::Drop;
```

```
struct D(i32);
impl Drop for D {
    fn drop(&mut self) {
        println!("destruct {}", self.0);
    }
}
fn main() {
    let _x = D(1);
    println!("construct 1");
    {
        let _y = D(2);
        println!("construct 2");
        println!("exit inner scope");
    }
    println!("exit main function");
}
```

编译，执行结果为：

```
construct 1
construct 2
exit inner scope
destruct 2
exit main function
destruct 1
```

从上面这段程序可以看出析构函数的调用时机。变量_y的生存期是内部的大括号包围起来的作用域（scope），待这个作用域中的代码执行完之后，它的析构函数就被调用；变量_x的生存期是整个main函数包围起来的作用域，待这个函数的最后一条语句执行完之后，它的析构函数就被调用。

对于具有多个局部变量的情况，析构函数的调用顺序是：先构造的后析构，后构造的先析构。因为局部变量存在于一个“栈”的结构中，要保持“先进后出”的策略。

11.6.1 资源管理

在创建变量的时候获取某种资源，在变量生命周期结束的时候释放资源，是一种常见的设计模式。这里的资源，不仅可以包括内存，还可以包括其他向操作系统申请的资源。比如我们经常用到的File类型，会在创建和使用的过程中向操作系统申请打开文件，在它的析构函数中就会去释放文件。所以，RAII手法是比GC更通用的资源管理手段，GC只

能管理内存，RAII可以管理各种资源。

下面用Rust标准库中的“文件”类型，来展示一下RAII手法。示例如下：

```
use std::fs::File;
use std::io::Read;

fn main() {
    let f = File::open("/target/file/path");
    if f.is_err() {
        println!("file is not exist.");
        return;
    }
    let mut f = f.unwrap();
    let mut content = String::new();
    let result = f.read_to_string(&mut content);
    if result.is_err() {
        println!("read file error.");
        return;
    }
    println!("{}", result.unwrap());
}
```

除去那些错误处理的代码以后，整个逻辑实际上相当清晰：首先使用open函数打开文件，然后使用read_to_string方法读取内容，最后关闭文件，这里不需要手动关闭文件，因为在File类型的析构函数中已经自动处理好了关闭文件这件事情。

再比如标准库中的各种复杂数据结构（如Vec LinkedList HashMap等），它们管理了很多在堆上动态分配的内存。它们也是利用“析构函数”这个功能，在生命终结之前释放了申请的内存空间，因此无须像C语言那样手动调用free函数。

11.6.2 主动析构

一般情况下，局部变量的生命周期是从它的声明开始，到当前语句块结束。然而，我们也可以手动提前结束它的生命周期。请注意，用户主动调用析构函数是非法的，示例如下：

```
fn main() {
    let p = Box::new(42);
    p.drop();
```

```
    println!("{}", p);
}
```

编译，可见错误信息：

```
error[E0040]: explicit use of destructor method
 --> test.rs:5:7
  |
5 |     p.drop();
  |       ^^^^ explicit destructor calls not allowed
```

这说明编译器不允许手动调用析构函数。那么，我们怎样才能让局部变量在语句块结束前提前终止生命周期呢？办法是调用标准库中的std：：mem：：drop函数：

```
use std::mem::drop;

fn main() {
    let mut v = vec![1, 2, 3];          // <--- v的生命周期开始
    drop(v);                            // ---> v的生命周期结束
    v.push(4);                          // 错误的调用
}
```

这段代码会编译出错，是因为调用drop方法的时候，v的生命周期就结束了，后面继续使用变量v就会发生编译错误。

那么，标准库中的std：：mem：：drop函数是怎样实现的呢？可能许多人想不到的是，这个函数是Rust中最简单的函数，因为它的实现为“空”：

```
#[inline]
pub fn drop<T>(_x: T) { }
```

drop函数不需要任何的函数体，只需要参数为“值传递”即可。将对象的所有权移入函数中，什么都不用做，编译器就会自动释放掉这个对象了。

因为这个drop函数的关键在于使用move语义把参数传进来，使得变量的所有权从调用方移动到drop函数体内，参数类型一定要是T，而不是&T或者其他引用类型。函数体本身其实根本不重要，重要的是把变

量的所有权move进入这个函数体中，函数调用结束的时候该变量的生命周期结束，变量的析构函数会自动调用，管理的内存空间也会自然释放。这个过程完全符合前面讲的生命周期、move语义，无须编译器做特殊处理。事实上，我们完全可以自己写一个类似的函数来实现同样的效果，只要保证参数传递是move语义即可。

因此，对于Copy类型的变量，对它调用std：：mem：：drop函数是没有意义的。下面以整数类型作为示例来说明：

```
use std::mem::drop;

fn main() {
    let x = 1_i32;
    println!("before drop {}", x);
    drop(x);
    println!("after drop {}", x);
}
```

这种情况很容易理解。因为Copy类型在函数参数传递的时候执行的是复制语义，原来的那个变量依然存在，传入函数中的只是一个复制品，因此原变量的生命周期不会受到影响。

变量遮蔽（Shadowing）不会导致变量生命周期提前结束，它不等同于drop。示例如下：

```
use std::ops::Drop;

struct D(i32);

impl Drop for D {
    fn drop(&mut self) {
        println!("destructor for {}", self.0);
    }
}

fn main() {
    let x = D(1);
    println!("construct first variable");
    let x = D(2);
    println!("construct second variable");
}
```

编译，执行，输出的结果为：

```
construct first variable
construct second variable
destructor for 2
destructor for 1
```

这里函数调用的顺序为：先创建第一个x，再创建第二个x，退出函数的时候，先析构第二个x，再析构第一个x。由此可见，在第二个x出现的时候，虽然将第一个x遮蔽起来了，但是第一个x的生命周期并未结束，它依然存在，直到函数退出。这也说明了，虽然这两个变量绑定了同一个名字，但在编译器内部依然将它们视为两个不同的变量。

另外还有一个小问题需要提醒读者注意，那就是下划线这个特殊符号。请注意：如果你用下划线来绑定一个变量，那么这个变量会当场执行析构，而不是等到当前语句块结束的时候再执行。下划线是特殊符号，不是普通标识符。示例如下：

```
use std::ops::Drop;

struct D(i32);

impl Drop for D {
    fn drop(&mut self) {
        println!("destructor for {}", self.0);
    }
}

fn main() {
    let _x = D(1);
    let _  = D(2);
    let _y = D(3);
}
```

执行结果为：

```
destructor for 2
destructor for 3
destructor for 1
```

之所以是这样的结果，是因为用下划线绑定的那个变量当场就执行了析构，而其他两个变量等到语句块结束了才执行析构，而且析构顺序和初始化顺序刚好相反。所以，如果大家需要利用RAII实现某个变量的析构函数在退出作用域的时候完成某些功能，千万不要用下划线来绑定这个变量。

最后，请大家注意区分，std：：mem：：drop（）函数和std：：ops：：Drop：：drop（）方法。

1）std：：mem：：drop（）函数是一个独立的函数，不是某个类型的成员方法，它由程序员主动调用，作用是使变量的生命周期提前结束；std：：ops：：Drop：：drop（）方法是一个trait中定义的方法，当变量的生命周期结束的时候，编译器会自动调用，手动调用是不允许的。

2）std：：mem：：drop<T>（_x：T）的参数类型是T，采用的是move语义；std：：ops：：Drop：：drop（&mut self）的参数类型是&mut Self，采用的是可变借用。在析构函数调用过程中，我们还有机会读取或者修改此对象的属性。

11.6.3 Drop VS.Copy

前面已经讲了，要想实现Copy trait，类型必须满足一定条件。这个条件就是：如果一个类型可以使用memcpy的方式执行复制操作，且没有内存安全问题，那么它才能被允许实现Copy trait。反过来，所有满足Copy trait的类型，在需要执行move语义的时候，使用memcpy复制一份副本，不删除原件是完全不会产生安全问题的。

本节中需要强调的是，带有析构函数的类型都是不能满足Copy语义的。因为我们不能保证，对于带析构函数的类型，使用memcpy复制一个副本一定不会有内存安全问题。所以对于这种情况，编译器是直接禁止的。

同样，下面还是用示例来说明：

```
use std::ops::Drop;

struct T;

impl Drop for T {
    fn drop(&mut self){}
}

impl Copy for T {}

fn main() {}
```

编译，可见编译错误：

```
error[E0184]: the trait `Copy` may not be implemented for this type; the type has a
 --> test.rs:9:1
  |
9 | impl Copy for T {}
  | ^^^^^^^^^^^^^^^^^ Copy not allowed on types with destructors
```

错误信息说得清清楚楚，带有析构函数的类型不能是Copy。这两个身份是不能同时存在于一个类型上的。

11.6.4 析构标记

在Rust里面，析构函数是在变量生命周期结束的时候被调用的。然而，既然我们可以手动提前终止变量的生命周期，那么就说明，变量的生命周期并不是简单地与某个代码块一致，生命周期何时结束，很可能是由运行时的条件决定的。下面用一个示例来说明变量的析构函数调用时机是有可能在运行阶段发生改变的：

```
use std::ops::Drop;
use std::mem::drop;

struct D(&'static str);
impl Drop for D {
    fn drop(&mut self) {
        println!("destructor {}", self.0);
    }
}

// 获取 DROP 环境变量的值,并转换为整数
fn condition() -> Option<u32> {
    std::env::var("DROP")
        .map(|s| s.parse::<u32>().unwrap_or(0))
        .ok()
}

fn main() {
    let var = (D("first"), D("second"), D("third"));
    match condition() {
        Some(1) => drop(var.0),
        Some(2) => drop(var.1),
        Some(3) => drop(var.2),
        _ => {},
    }
    println!("main end");
}
```

在上面这段示例代码中，我们在变量的析构函数中写了一条打印语句，用于判断析构函数的调用顺序。在主函数里面，则通过判断当前的环境变量信息来决定是否提前终止某个变量的生命周期。

编译执行，如果我们没有设置DROP环境变量，输出结果为：

```
main end
destructor first
destructor second
destructor third
```

如果我们设置了export DROP=2这个环境变量，不重新编译，执行同样的代码，输出结果为：

```
destructor second
main end
destructor first
destructor third
```

我们还可以将DROP环境变量的值分别改为“1”、“2”、“3”，结果会导致析构函数的调用顺序发生变化。

然而，问题来了，前面说过，析构函数的调用是在编译阶段就确定好了的，调用析构函数是编译器自动插入的代码做的。而且示例又表明，析构函数的具体调用时机还是跟运行时的情况相关的。那么编译器是怎么做到的呢？

编译器是这样完成这个功能的：首先判断一个变量是否可能会在多个不同的路径上发生析构，如果是这样，那么它会在当前函数调用栈中自动插入一个bool类型的标记，用于标记该对象的析构函数是否已经被调用，生成的代码逻辑像下面这样：

```
// 以下为伪代码,仅仅是示意
fn main() {
    let var = (D("first"), D("second"), D("third"));
    // 当函数中有拥有所有权的对象时,需要有析构自动标记
    let drop_flag_0 = false;  // ---
    let drop_flag_1 = false;  // ---
    let drop_flag_2 = false;  // ---

    // 退出语句块时,对当前block内拥有所有权的对象调用析构函数,并设置标记
    match condition() {
```

```
            Some(1) => {
                drop(var.0);
                if (!drop_flag_0) {      // ---
                    drop_flag_0 = true;  // ---
                }                        // ---
            }
            Some(2) => {
                drop(var.1);
                if (!drop_flag_1) {      // ---
                    drop_flag_1 = true;  // ---
                }                        // ---
            }
            Some(3) => {
                drop(var.2);
                if (!drop_flag_2) {      // ---
                    drop_flag_2 = true;  // ---
                }                        // ---
            }
            _ => {},
        }

        println!("main end");
        // 退出语句块时,对当前block内拥有所有权的对象调用析构函数,并设置标记
        if (!drop_flag_0) {              // ---
            drop(var.0);                 // ---
            drop_flag_0 = true;          // ---
        }                                // ---
        if (!drop_flag_1) {              // ---
            drop(var.1);                 // ---
            drop_flag_1 = true;          // ---
        }                                // ---
        if (!drop_flag_2) {              // ---
            drop(var.2);                 // ---
            drop_flag_2 = true;          // ---
        }                                // ---
    }
```

编译器生成的代码类似于上面的示例，可能会有细微差别。原理是在析构函数被调用的时候，就把标记设置一个状态，在各个可能调用析构函数的地方都先判断一下状态再调用析构函数。这样，编译阶段确定生命周期和执行阶段根据情况调用就统一起来了。

第12章　借用和生命周期

12.1　生命周期

一个变量的生命周期就是它从创建到销毁的整个过程。其实我们在前面已经注意到了这样的现象：

```
fn main() {
    let v = vec![1,2,3,4,5];        // --> v 的生命周期开始
    {
        let center = v[2];          // --> center 的生命周期开始
        println!("{}", center);
    }                               // <-- center 的生命周期结束
    println!("{:?}", v);
}                                   // <-- v 的生命周期结束
```

然而，如果一个变量永远只能有唯一一个入口可以访问的话，那就太难使用了。因此，所有权还可以借用。

12.2 借用

变量对其管理的内存拥有所有权。这个所有权不仅可以被转移（move），还可以被借用（borrow）。本节讲解一下如何实现所有权借用。

借用指针的语法使用&符号或者&mut符号表示。前者表示只读借用，后者表示可读写借用。借用指针（borrow pointer）也可以称作“引用”（reference）。借用指针与普通指针的内部数据是一模一样的，唯一的区别是语义层面上的。它的作用是告诉编译器，它对指向的这块内存区域没有所有权。

示例如下：

```
fn foo(v: &Vec<i32>) {
    v.push(5);
}

fn main() {
    let v = vec![];
    foo(&v);
}
```

这里会出现编译错误，信息为“cannot borrow immutable borrowed content`*v`as mutable”。

原因在于Vec：：push函数。它的作用是对动态数组添加元素，它的签名是

```
pub fn push(&mut self, value: T)
```

它要求self参数是一个&mut Self类型。而我们给foo传递的参数是&Vec类型，因此会报错。修复方式如下：

```
// 我们需要“可变的”借用指针,因此函数签名需要改变
fn foo(v: &mut Vec<i32>) {
    v.push(5);
}
```

```
fn main() {
    // 我们需要这个动态数组本身是“可变的”,才能获得它的“可变借用指针”
    let mut v = vec![];

    // 在函数调用的时候,同时也要显示获取它的“可变借用指针”
    foo(&mut v);

    // 打印结果,可以看到v已经被改变
    println!("{:?}", v);
}
```

对于&mut型指针，请大家注意不要混淆它与变量绑定之间的语法。如果mut修饰的是变量名，那么它代表这个变量可以被重新绑定；如果mut修饰的是“借用指针&”，那么它代表的是被指向的对象可以被修改。示例如下：

```
fn main() {
    let mut var = 0_i32;
    {
        let p1 = &mut var; // p1 指针本身不能被重新绑定,我们可以通过p1改变变量var的值

        *p1 = 1;
    }
    {
        let temp = 2_i32;
        let mut p2 = &var; // 我们不能通过p2改变变量var的值,但p2指针本身指向的位置可以被改变
        p2 = &temp;

    }
    {
        let mut temp = 3_i32;
        let mut p3 = &mut var; // 我们既可以通过p3改变变量var的值,而且p3指针本身指向的位置也
        *p3 = 3;
        p3 = &mut temp;
    }
}
```

借用指针在编译后，实际上就是一个普通的指针，它的意义只能在编译阶段的静态检查中体现。

12.3 借用规则

关于借用指针，有以下几个规则：

·借用指针不能比它指向的变量存在的时间更长。

·&mut型借用只能指向本身具有mut修饰的变量，对于只读变量，不可以有&mut型借用。

·&mut型借用指针存在的时候，被借用的变量本身会处于“冻结”状态。

·如果只有&型借用指针，那么能同时存在多个；如果存在&mut型借用指针，那么只能存在一个；如果同时有其他的&或者&mut型借用指针存在，那么会出现编译错误。

借用指针只能临时地拥有对这个变量读或写的权限，没有义务管理这个变量的生命周期。因此，借用指针的生命周期绝对不能大于它所引用的原来变量的生命周期，否则就是悬空指针，会导致内存不安全。示例如下：

```
// 这里的参数采用的“引用传递”,意味着实参本身并未丢失对内存的管理权
fn borrow_semantics(v : &Vec<i32>) {

    // 打印参数占用空间的大小,在64位系统上,结果为8,表明该指针与普通裸指针的内部表示方法相同
    println!("size of param: {}", std::mem::size_of::<&Vec<i32>>());
    for item in v {
        print!("{} ", item);
    }
    println!("");
}

// 这里的参数采用的“值传递”,而Vec没有实现Copy trait,意味着它将执行move语义
fn move_semantics(v : Vec<i32>) {

    // 打印参数占用空间的大小,结果为24,表明实参中栈上分配的内存空间复制到了函数的形参中
    println!("size of param: {}", std::mem::size_of::<Vec<i32>>());
    for item in v {
        print!("{} ", item);
    }
    println!("");
}

fn main() {
    let array = vec![1, 2, 3];
```

```
    // 需要注意的是,如果使用引用传递,不仅在函数声明的地方需要使用&标记
    // 函数调用的地方同样需要使用&标记,否则会出现语法错误
    // 这样设计主要是为了显眼,不用去阅读该函数的签名就知道这个函数调用的时候发生了什么
    // 而小数点方式的成员函数调用,对于self参数,会“自动转换”,不必显式借用,这里有个区别
    borrow_semantics(&array);

    // 在使用引用传递给上面的函数后,array本身依然有效,我们还能在下面的函数中使用
    move_semantics(array);

    // 在使用move语义传递后,array在这个函数调用后,它的生命周期已经完结
}
```

在这里给大家提个醒：一般情况下，函数参数使用引用传递的时候，不仅在函数声明这里要写上类型参数，在函数调用这里也要显式地使用引用运算符。但是，有一个例外，那就是当参数为self &self &mut self等时，若使用小数点语法调用成员方法，在函数调用这里不能显式写出借用运算符。以常见的String类型来举例：

```
fn main() {
    // 创建了一个可变的 String 类型实例
    let mut x : String = "hello".into();

    // 调用 len(&self) -> usize 函数。 self的类型是 &Self
    // x.len() 等同于 String::len(&x)
    println!("length of String {}", x.len());

    // 调用fn push(&mut self, ch: char) 函数。self的类型是 &mut Self,因此它有权对字符串做修
    // x.push('!') 等同于 String::push(&mut x, '!')
    x.push('!');

    println!("length of String {}", x.len());

    // 调用 fn into_bytes(self) -> Vec<u8> 函数。注意self的类型,此处发生了所有权转移
    // x.into_bytes() 等同于 String::into_bytes(x)
    let v = x.into_bytes();

    // 再次调用len(),编译失败,因为此处已经超过了 x 的生命周期
    //println!("length of String {}", x.len());
}
```

在这个示例中，所有的函数调用都是同样的语法，比如x.len（）、x.push（'！'）、x.into_bytes（）等，但它们背后对self参数的传递类型完全不同，因此也就出现了不同的语义。这是需要提醒大家注意的地方。当然，如果我们使用统一的完整函数调用语法，那么所有的参数传递类型在调用端都是显式写出来的。

任何借用指针的存在，都会导致原来的变量被“冻结”（Frozen）。

示例如下：

```
fn main() {
    let mut x = 1_i32;
    let p = &mut x;
    x = 2;
    println!("value of pointed : {}", p);
}
```

编译结果为：

```
error: cannot assign to `x` because it is borrowed
```

因为p的存在，此时对x的改变被认为是非法的。至于为什么会有这样的规定，请参考下一章。

12.4 生命周期标记

对一个函数内部的生命周期进行分析，Rust编译器可以很好地解决。但是，当生命周期跨函数的时候，就需要一种特殊的生命周期标记符号了。

12.4.1 函数的生命周期标记

示例如下：

```
01| struct T {
02|     member: i32,
03| }
04|
05| fn test<'a>(arg: &'a T) -> &'a i32
06| {
07|     &arg.member
08| }
09|
10| fn main() {
11|     let t = T { member : 0 };  //----- 't -|
12|     let x = test(&t);          //-- 'x ---|      |
13|     println!("{:?}", x);       //         |      |
14| }                              //-- 'x ----- 't -
```

生命周期符号使用单引号开头，后面跟一个合法的名字。生命周期标记和泛型类型参数是一样的，都需要先声明后使用。在上面这段代码中，尖括号里面的'a是声明一个生命周期参数，它在后面的参数和返回值中被使用。

前面提到的借用指针类型都有一个生命周期泛型参数，它们的完整写法应该是&'a T&'a mut T，只不过在做局部变量的时候，生命周期参数是可以省略的。

生命周期之间有重要的包含关系。如果生命周期'a比'b更长或相等，则记为'a：'b，意思是'a至少不会比'b短，英语读做“lifetime a outlives lifetime b”。对于借用指针类型来说，如果&'a是合法的，那么'b作为'a的一部分，&'b也一定是合法的。

另外，'static是一个特殊的生命周期，它代表的是这个程序从开始到结束的整个阶段，所以它比其他任何生命周期都长。这意味着，任意一个生命周期'a都满足'static：'a。

在上面这个例子中，如果我们把变量t的真实生命周期记为't，那么这个生命周期't实际上是变量t从“出生”到“死亡”的区间，即从第11行到第14行。在函数被调用的时候，它传入的实际参数是&t，它是指向t的引用。那么可以说，在调用的时候，这个泛型参数'a被实例化为了't。根据函数签名，基于返回类型的生命周期与参数是一致的，可以推理出test函数的返回类型是&'t i32。如果我们把x的生命周期记为'x，那么'x代表的就是从第12行到第14行。这条let x=text（&t）；语句实际上是把&'t i32类型的变量赋值给&'x i32类型的变量。这个赋值是否合理呢？它应该是合理的。因为这两个生命周期的关系是't：'x。test返回的那个指针在't这个生命周期范围内都是合法的，在一个被't包围的更小范围的生命周期内，它当然也是合法的。所以，上面这个例子可以编译通过。

接下来，我们把上面这个例子稍作修改，让test函数有两个生命周期参数，其中一个给函数参数使用，另外一个给返回值使用：

```
fn test<'a, 'b>(arg: &'a T) -> &'b i32
{
    &arg.member
}
```

编译时果然出了问题，在&arg.member这一行，报了生命周期错误。这是为什么呢？因为这一行代码是把&'a i32类型赋值给&'b i32类型。'a和'b有什么关系？答案是什么关系都没有。所以编译器觉得这个赋值是错误的。怎么修复呢？指定'a：'b就可以了。'a比'b“活”得长，自然，&'a i32类型赋值给&'b i32类型是没问题的。验证如下：

```
fn test<'a, 'b>(arg: &'a T) -> &'b i32
    where 'a:'b
{
    &arg.member
}
```

经过这样的改写后，我们可以认为，在test函数被调用的时候，生命周期参数'a和'b被分别实例化为了't和'x。它们刚好满足了where条件中的't：'x约束。而&arg.member这条表达式的类型是&'t i32，返回值要求

的是&'x i32类型，可见这也是合法的。所以test函数的生命周期检查可以通过。

上述示例是读者比较难理解的地方。以下两种写法都是可行的：

```
fn test<'a>(arg: &'a T) -> &'a i32
fn test<'a, 'b>(arg: &'a T) -> &'b i32    where 'a:'b
```

这里的关键是，Rust的引用类型是支持“协变”的。在编译器眼里，生命周期就是一个区间，生命周期参数就是一个普通的泛型参数，它可以被特化为某个具体的生命周期。

我们再看一个例子。它有两个引用参数，共享同一个生命周期标记：

```
fn select<'a>(arg1: &'a i32, arg2: &'a i32) -> &'a i32 {
    if *arg1 > *arg2 {
        arg1
    } else {
        arg2
    }
}

fn main() {
    let x = 1;
    let y = 2;
    let selected = select(&x, &y);
    println!("{}", selected);
}
```

上述示例中，select这个函数引入了一个生命周期标记，两个参数以及返回值都是用的这个生命周期标记。同时我们注意到，在调用的时候，传递的实参其实是具备不同的生命周期的。x的生命周期明显大于y的生命周期，&x可存活的范围要大于&y可存活的范围，我们把它们的实际生命周期分别记录为'x和'y。select函数的形式参数要求的是同样的生命周期，而实际参数是两个不同生命周期的引用，这个类型之所以可以匹配成功，就是因为生命周期的协变特性。编译器可以把&x和&y的生命周期都缩小到某个生命周期'a以内，且满足'x：'a，'y：'a。返回的selected变量具备'a生命周期，也并没有超过'x和'y的范围。所以，最终的生命周期检查可以通过。

12.4.2 类型的生命周期标记

如果自定义类型中有成员包含生命周期参数，那么这个自定义类型也必须有生命周期参数。示例如下：

```
struct Test<'a> {
    member: &'a str
}
```

在使用impl的时候，也需要先声明再使用：

```
impl<'t> Test<'t> {
    fn test<'a>(&self, s: &'a str) {

    }
}
```

impl后面的那个't是用于声明生命周期参数的，后面的Test<'t>是在类型中使用这个参数。如果有必要的话，方法中还能继续引入新的泛型参数。

如果在泛型约束中有where T：'a之类的条件，其意思是，类型T的所有生命周期参数必须大于等于'a。要特别说明的是，若是有where T：'static的约束，意思则是，类型T里面不包含任何指向短生命周期的借用指针，意思是要么完全不包含任何借用，要么可以有指向'static的借用指针。

12.5 省略生命周期标记

在某些情况下，Rust允许我们在写函数的时候省略掉显式生命周期标记。在这种时候，编译器会通过一定的固定规则为参数和返回值指定合适的生命周期，从而省略一些显而易见的生命周期标记。比如我们可以写这样的代码：

```
fn get_str(s: &String) -> &str {
    s.as_ref()
}
```

实际上，它等同于下面这样的代码，只是把显式的生命周期标记省略掉了而已：

```
fn get_str<'a>(s: &'a String) -> &'a str {
    s.as_ref()
}
```

若把以上代码稍微修改一下，返回的指针将并不指向参数传入的数据，而是指向一个静态常量，代码如下：

```
fn get_str(s: &String) -> &str {
    println!("call fn {}", s);
    "hello world"
}
```

这时，我们期望返回的指针实际上是&'static str类型。测试代码如下：

```
fn main() {
    let c = String::from("haha");
    let x: &'static str = get_str(&c);
    println!("{}", x);
}
```

可以看到，在get_str函数中，返回的是一个指向静态字符串的指针。在主函数的调用方，我们希望变量x指向一个“静态变量”。可是这

一次，我们发现了编译错误：

```
error: `c` does not live long enough
```

按照分析，变量x理应指向一个'static生命周期的变量，根本不是指向C变量，它的存活时间足够长，为什么编译器没发现这一点呢？这是因为，编译器对于省略掉的生命周期，不是用的“自动推理”策略，而是用的几个非常简单的“固定规则”策略。这跟类型自动推导不一样，当我们省略变量的类型时，编译器会试图通过变量的使用方式推导出变量的类型，这个过程叫“type inference”。而对于省略掉的生命周期参数，编译器的处理方式就简单粗暴得多，它完全不管函数内部的实现，并不尝试找到最合适的推理方案，只是应用几个固定的规则而已，这些规则被称为“lifetime elision rules”。以下就是省略的生命周期被自动补全的规则：

·每个带生命周期参数的输入参数，每个对应不同的生命周期参数；

·如果只有一个输入参数带生命周期参数，那么返回值的生命周期被指定为这个参数；

·如果有多个输入参数带生命周期参数，但其中有&self、&mut self，那么返回值的生命周期被指定为这个参数；

·以上都不满足，就不能自动补全返回值的生命周期参数。

这时再回头去看前面的例子，可以知道，对于这个函数：

```
fn get_str(s: &String) -> &str {
    println!("call fn {}", s);
    "hello world"
}
```

编译器会自动补全生命周期参数：

```
fn get_str<'a>(s: &'a String) -> &'a str {
    println!("call fn {}", s);
    "hello world"
```

```
}
```

所以，当我们调用

```
let x: &'static str = get_str(&c);
```

这句代码的时候，就发生了编译错误。了解了这些，修复方案也就很简单了。在这种情况下，我们不能省略生命周期参数，让编译器给我们自动补全，自己手写就对了：

```
fn get_str<'a>(s: &'a String) -> &'static str {
    println!("call fn {}", s);
    "hello world"
}
```

或者只手写返回值的生命周期参数，输入参数靠编译器自动补全：

```
fn get_str(s: &String) -> &'static str { ... }
```

最后，一句话总结，elision！=inference，省略生命周期参数和类型自动推导的原理是完全不同的。

第13章　借用检查

在前文中，我们已经讨论了Rust在内存管理方面的语法。本文将主要探讨Rust实现无性能损失的“内存安全”的原理。

Rust语言的核心特点是：在没有放弃对内存的直接控制力的情况下，实现了内存安全。所谓对内存的直接控制能力，前文已经有所展示：可以自行决定内存布局，包括在栈上分配内存，还是在堆上分配内存；支持指针类型；可以对一个变量实施取地址操作；有确定性的内存释放；等等。

另一方面，从安全性的角度来说，我们可以看到，Rust有所有权概念、借用指针、生命周期分析等这些内容。初学者在刚开始碰到这些概念的时候，往往会觉得无所适从，感觉太麻烦、太复杂了。随便写个小程序都编译不通过，学习曲线非常陡峭。那么，Rust设计者究竟是如何考虑的这个问题，为什么要设计这样复杂的规则？Rust语言的这一系列安全规则，背后的指导思想究竟是什么呢？

总的来说，Rust的设计者们在一系列的“内存不安全”的问题中观察到了这样的一个结论：

Danger arises from　Aliasing+Mutation

首先我们介绍一下这两个概念Alias和Mutation。

（1）Alias的意思是“别名”。如果一个变量可以通过多种Path来访问，那它们就可以互相看作alias。Alias意味着“共享”，我们可以通过多个入口访问同一块内存。

（2）Mutation的意思是“改变”。如果我们通过某个变量修改了一块内存，就是发生了mutation。Mutation意味着拥有“修改”权限，我们可以写入数据。

Rust保证内存安全的一个重要原则就是，如果能保证alias和mutation不同时出现，那么代码就一定是安全的。

在本书中，笔者将此规则总结为：共享不可变，可变不共享。

13.1 编译错误示例

Rust的编译错误列表（https://doc.rust-lang.org/error-index.html ）中，从E0499到E0509，所有的这些编译错误，其实都在讲同一件事情。它们主要关心的是共享和可变之间的关系。“共享不可变，可变不共享”是所有这些编译错误遵循的同样的法则。

下面我们通过几个简单的示例来直观地感受一下这个规则究竟是什么意思。

示例一

```
fn main() {
    let i = 0;
    let p1 = & i;
    let p2 = & i;
    println!("{} {} {}", i, p1, p2);
}
```

以上这段代码是可以编译通过的。其中变量绑定i、p1、p2指向的是同一个变量，我们可以通过不同的Path访问同一块内存p，*p1，*p2，所以它们存在“共享”。而且它们都只有只读的权限，所以它们存在“共享”，不存在“可变”。因此它一定是安全的。

示例二

我们让变量绑定i是可变的，然后在存在p1的情况下，通过i修改变量的值：

```
fn main() {
    let mut i = 0;
    let p1 = & i;
    i = 1;
}
```

编译，出现了错误，错误信息为：

```
error: cannot assign to `i` because it is borrowed [E0506]
```

这个错误可以这样理解：在存在只读借用的情况下，变量绑定i和p1已经互为alias，它们之间存在“共享”，因此必须避免“可变”。这段代码违反了“共享不可变”的原则。

示例三

如果我们把上例中的借用改为可变借用的话，其实是可以通过它修改原来变量的值的。以下代码可以编译通过：

```
fn main() {
    let mut i = 0;
    let p1 = &mut i;
    *p1 = 1;
}
```

那我们是不是说，它违反了“共享不可变”的原则呢？其实不是。因为这段代码中不存在“共享”。在可变借用存在的时候，编译器认为原来的变量绑定i已经被冻结（frozen），不可通过i读写变量。此时有且仅有p1这一个入口可以读写变量。证明如下：

```
fn main() {
    let mut i = 0;
    let p1 = &mut i;
    *p1 = 1;
    let x = i; // 通过i读变量
}
```

在存在p1的情况下，我们再通过i做读操作是错误的：

```
error: cannot use `i` because it was mutably borrowed [E0503]
```

同理，如果我们改成下面这样，一样会出错：

```
fn main() {
    let mut i = 0;
    let p1 = &mut i;
    i = 1; // 通过i写变量
}
```

在p1存在的情况下，不可通过i写变量。如果这种情况可以被允许，那就会出现多个入口可以同时访问同一块内存，且都具有写权限，这就违反了Rust的“共享不可变，可变不共享”的原则。错误信息为：

```
error: cannot assign to `i` because it is borrowed [E0506]
```

示例四

同时创建两个可变借用的情况：

```
fn main() {
    let mut i = 0;
    let p1 = &mut i;
    let p2 = &mut i;
}
```

编译错误信息为：

```
error: cannot borrow `i` as mutable more than once at a time [E0499]
```

因为p1、p2都是可变借用，它们都指向了同一个变量，而且都有修改权限，这是Rust不允许的情况，因此这段代码无法编译通过。

正因如此，&mut型借用也经常被称为“独占指针”，&型借用也经常被称为“共享指针”。

13.2 内存不安全示例：修改枚举

Rust设计的这个原则，究竟有没有必要呢？它又是如何在实际代码中起到“内存安全”检查作用的呢？

第一个示例，我们用enum来说明。假如我们有一个枚举类型：

```
enum StringOrInt {
    Str(String),
    Int(i64),
}
```

它有两个元素，分别可以携带String类型的信息以及i64类型的信息。假如我们有一个引用指向了它的内部数据，同时再修改这个变量，大家猜想会发生什么情况？这样做可能会出现内存安全问题，因为我们有机会用一个String类型的指针指向i64类型的数据，或者用一个i64类型的指针指向String类型的数据。完整示例如下：

```
use std::fmt::Debug;

#[derive(Debug)]
enum StringOrInt {
        Str(String),
        Int(i64),
}

fn main() {
    use StringOrInt::{Str, Int};
    let mut x = Str("Hello world".to_string());

    if let Str(ref insides) = x {
        x = Int(1);
        println!("inside is {}, x says: {:?}", insides, x);
    }
}
```

在这段代码中，我们用if let语法创建了一个指向内部String的指针，然后在此指针的生命周期内，再把x内部数据变成i64类型。这是典型的内存不安全的场景。

幸运的是，这段代码编译不通过，错误信息为：

```
error: cannot assign to `x` because it is borrowed [E0506]
```

这个例子给了我们一个直观的感受，为什么Rust需要“可变性和共享性不能同时存在”的规则？保证当前只有一个访问入口，这是保证安全的可靠做法。

13.3 内存不安全示例：迭代器失效

如果在遍历一个数据结构的过程中修改这个数据结构，会导致迭代器失效。比如在C++里面，我们可能写出这样的代码：

```
#include <vector>

using namespace std;
int main() {
    vector<int> v(10,10);

    for (vector<int>::iterator i = v.begin(); i != v.end(); i++) {
        if (*i % 2 == 0) { // when arbitrary condition satisfied
            v.clear();
        }
    }
    return 0;
}
```

编译，执行，发现程序崩溃了。原因就在于我们在迭代的过程中，数组v直接被清空了，而迭代器并不知道这个信息，它还在继续进行迭代，于是出现了“野指针”现象，此时迭代器实际上指向了已经被释放的内存。迭代器失效这样的问题在C++中是“未定义行为”，也就是说可能发生什么后果都是未知的。这是一种典型的内存不安全行为。

然而，在Rust里面，下面这样的代码是不允许编译通过的：

```
fn main() {
    let mut arr = vec!["ABC", "DEF", "GHI"];
    for item in &arr {
        arr.clear();
    }
}
```

为什么Rust可以避免这个问题呢？因为Rust里面的for循环实质上是生成了一个迭代器，它一直持有一个指向容器的引用，在迭代器的生命周期内，任何对容器的修改都是无法编译通过的。类似这样：

```
{   //以下是伪代码
    // 在iter变量的生命周期内,它都持有一个指向arr的引用
    let iter<'a> = into_iter(&'a arr);
```

```
    loop  {
        match iter.next() {
// 如果需要使用 arr 的 &mut 指针,则会发生冲突

// &mut arr 和 &arr 不能同时存在,它违反了Rust内存安全的原则
            Some(i) => { arr.clear(); }
            None => break ,
        }
    }
}
```

在整个for循环的范围内，这个迭代器的生命周期都一直存在。而它持有一个指向容器的引用，&型或者&mut型，根据情况而定。迭代器的API设计是可以修改当前指向的元素，没办法修改容器本身的。当我们想在这里对容器进行修改的时候，必然需要产生一个新的针对容器的&mut型引用，（clear方法的签名是Vec：：clear（&mut self），调用clear必然产生对原Vec的&mut型引用）。这是与Rust的“alias+mutation”规则相冲突的，所以编译不通过。

为什么在Rust中永远不会出现迭代器失效这样的错误？因为通过“mutation+alias”规则，就可以完全杜绝这样的现象，这个规则是Rust内存安全的根，是解决内存安全问题的灵魂。Rust不是针对各式各样的场景，用case by case的方式来解决内存安全问题。而是通过一种统一的机制，高屋建瓴地解决这一类问题，快刀斩乱麻，直击要害。

面对类似迭代器失效这一类的、指针指向非法地址的内存安全问题，许多语言都无法做到静态检查出来。比如在Java中出现这样的问题的时候，编译器是没法检查出来的，在执行阶段，程序会抛出一个异常“Exception in thread“main”java.util.ConcurrentModificationException”。因为我们在for循环内部对容器本身做了修改，Java容器探测到了这种修改，然后就阻止了逻辑的继续执行，抛出了异常。Java的这个设计相比C++要好很多，因为即便出现了迭代器失效，最多引发异常，而并不会有“野指针”这样的内存安全问题，因为迭代器没有机会访问已经被释放的非法内存。然而“抛出异常”并不是一个完美的设计，只是不得已而为之罢了。因为异常本来的设计目的是为了处理外部环境难以预计的错误的，而现在的这个错误实际上是程序的逻辑错误，即便抛出了异常，外部逻辑捕获了这个异常，也没什么好办法来处理。唯一合理的修复方案是，发现这样的异常之后，回过头来修复代码错误。这样的问题如果在编译阶段就能得到发现和解决，才是最合适的解决方案。在遍历容器的时候同时对容器做修改，可能出现在多线程场景，也可能出现在单线程场景。

类似这样的问题依靠GC也没办法处理。GC只关心内存的分配和释放，对于变量的读写权限是不关心的。GC在此处发挥不了什么作用。

而Rust依靠我们前面强调的“alias+mutation”规则就可以很好地解决该问题。这个思路的核心就是：如果存在多个只读的引用，是允许的；如果存在可写的引用，那么就一定不能同时存在其他的只读或者可写的引用。大家看到这个逻辑，是不是马上联想到多线程环境下的“ReadWriteLocker”？事实也确实如此。Rust检查内存安全的核心逻辑可以理解为一个在编译阶段执行的读写锁。多个读同时存在是可以的，存在一个写的时候，其他的读写都不能同时存在。

大家还记不记得，Rust设计者总结的Rust的三大特点：一是快，二是内存安全，三是免除数据竞争。由上面的分析可见，Rust所说的“免除数据竞争”，实际上和“内存安全”是一回事。“免除数据竞争”可以看作多线程环境下的“内存安全”。单线程环境下的“内存安全”靠的是编译阶段的类似读写锁的机制，与多线程环境下其他语言常用的读写锁机制并无太大区别。也正因为Rust编译器在设计上打下的良好基础，“内存安全”才能轻松地扩展到多线程环境下的“免除数据竞争”。这两个概念其实只有一箭之隔。所以我们可以理解Java将此异常命名为“Concurrent”的真实含义——这里的“Concurrent”并不是单指多线程并发。

13.4 内存不安全示例：悬空指针

我们再使用一个例子，来继续说明为什么Rust的“mutation+alias”规则是有必要的。我们这次通过制造一个悬空指针来解释。以下为一段合理的C++代码，它创建了一个动态数组，然后使用了一个指针，指向了动态数组的内部元素，然后我们向动态数组内添加内容，然后发现原先的指针“悬空”了，它指向了一个非法的地址：

```
// 以下仅仅为了示例而已,不代表推荐的C++编码风格
#include <vector>
#include <iostream>

using namespace std;
int main() {
    vector<int> v(100, 5);

    // 指针指向内部第一个元素
    int * p0 = &v[0];
    cout << *p0 << endl;
    // 为了确保v发生扩容,多插入一些数据
    for (int i = 0; i<100; i++) {
        v.push_back(10);
    }
    // 打印p0的内容
    cout << *p0 << endl;
    return 0;
}
```

编译通过，执行结果为：

```
5
-72140872
```

熟悉STL的朋友肯定知道这里究竟发生了什么。动态数组是自行管理内存空间的，在向动态数组内部添加元素的时候，如果超过了当前的最大容量，这个动态数组会申请一块更大的连续内存空间，将原来的元素移动过去，释放掉之前的内存空间，然后继续往后面添加元素。

我们的指针一开始是指向动态数组的第一个元素的，但是当往动态数组内部添加多个元素之后，之前的那块内存已经不够用了，动态数组在这个过程中已经将原来的内存空间释放，并申请了新的内存空间。于

是，原本应该指向数组第一个元素的指针从一个合法的指针变成了指向已回收内存区域的悬空指针，它现在指向的数据是与原来的意图不同的。而这种情况正是属于Rust希望解决的“内存安全”问题。

我们来看看用Rust写会发生什么。同样，使用动态数组类型，使用一个指针指向它的第一个元素，然后在原来的动态数组中插入数据：

```
fn main() {
    let mut arr : Vec<i32> = vec![1,2,3,4,5];
    let p : &i32 = &arr[0];
    for i in 1..100 {
        arr.push(i);
    }
}
```

编译不通过，错误信息为：

```
error: cannot borrow `arr` as mutable because it is also borrowed as immutable
```

我们可以看到，“mutation+alias”规则再次起了作用。在存在一个不可变引用的情况下，我们不能修改原来变量的值。写Rust代码的时候，经常会有这样的感觉：Rust编译器极其严格，甚至到了“不近人情”的地步。但是大部分时候却又发现，它指出来的问题的确是对我们编程有益的。对它使用越熟练，越觉得它是一个好帮手。

13.5 小结

Rust在内存安全方面的设计方案的核心思想是“共享不可变，可变不共享”。

在可变性控制方面，如果说，C语言和函数式编程语言分属一个天平的两端，那么Rust就处于这个天平的中央。C语言的思想是：尽量不对程序员做限制，尽量接近机器底层，类型安全、可变性、共享性都由程序员自由掌控，语言本身不提供太多的限制和规定。安全与否，也完全取决于程序员。而函数式编程的思想是：尽量使用不可变绑定，在可变性上有严格限制，在共享性方面没有限制。函数式编程特别强调无副作用的函数以及不可变类型，以此来达到提高安全性的目的。

而Rust则是选择了折中的方案，既允许可变性，也允许共享性，只要这两者不是同时出现即可。“共享不可变，可变不共享”，是Rust保证内存安全和线程安全的“法宝”。而我们可以看到，Rust的这个设计并不是首鼠两端、和稀泥式的中庸之道，而是经过了仔细的观察总结、严谨的设计之后的产物。

其一，相比函数式设计方式，Rust并没有本质上牺牲安全性。函数式编程强调的“不可变”特性，极大地提升了安全性的同时，也极大地提高了学习门槛。而Rust在“不可变”要求上的理性妥协，实现了在不损失安全性的同时，一定程度上也降低了学习成本。从C/C++背景转为使用Rust无需做太大的思维转变。相比函数式的设计方式，Rust的入门门槛更低。虽然对于习惯了无拘无束自由挥洒的C/C++编程语言的朋友来说，还是有诸多不习惯，但毕竟比Haskell要容易得多。

其二，Rust针对传统C/C++做了大幅改进，设计了一系列静态检查规则，来防止一些潜在的bug。“共享不可变，可变不共享”就是其中一项重要的规则。在传统的C/C++中，所有的指针都是同一个类型。从功能性来说，这样设计是非常强大的，但它缺少的恰恰是一定程度的取舍，以提高安全性。相对来说，Rust对程序员的限制更多，有所为、有所不为。鼓励用户使用的功能应当越容易越优雅越好；避免用户滥用的功能应当越困难越复杂越好。二者不可偏废。

其三，Rust的这套内存安全体系，不需要依赖GC。虽然现在GC的

性能越来越好，但是没有GC在某些场景下依然是很重要的。没有GC、编译型语言的特点，是Rust执行性能的潜力保证。这就是为什么Rust设计组有底气说Rust的运行性能与C语言处于同一个档次的原因。当然，目前的Rust还很年轻，许多优化还没有实现，但这不要紧，单从技术层面上看，还有许多优化在可行性上是没问题的，唯一需要的是时间和工作量。另外，没有GC就可以使得它只依赖一个非常轻量级的runtime。理论上来说，它可以用于许多嵌入式平台，甚至可以在无操作系统的裸机上执行，使用Rust编写操作系统也是完全可行的。这就使得Rust拥有与C/C++相似的系统级编程特性，大幅扩展了Rust的应用场景。

其四，Rust的核心思想“共享不可变，可变不共享”，具有极好的一致性和扩展性。它不仅可以解决内存安全的问题，还是解决线程安全的基础。在后文中我们会看到，所谓的线程安全，实质上就是内存安全在多线程情况下的自然延伸。反过来，我们也可以把Rust的内存安全解决方案视为传统的线程安全机制Read Write Locker的编译阶段执行的版本。大家应该都能联想到，在多线程环境下，数据竞争问题是怎么出现的。如果多个线程对同一个共享变量都是只读的，它是安全的；如果有一个线程对共享变量写操作，那它就必须是独占的，不可有其他线程继续读写，否则就会出现数据竞争。在第四部分中我们还会发现，Rust里面的许多线程安全的类型，与一些非线程安全的类型，具有非常有趣的对称性。

由此我们可以看出，Rust的这套设计方案的确是有创新性的。它走出了一条前无古人的道路。Rust在其他方面的功能，都不能被称作原创设计，都是从其他编程语言中学过来的。唯独安全性方面的设计是独一无二的。只要我们保证了“共享不可变，可变不共享”，我们就可以保证内存安全。那么它这套设计方案，究竟能不能被大众所接受呢？我们拭目以待。

另外，这个规定是否是过于严苛了呢？会不会大幅削弱代码的表达能力？后面我们还需要进一步分析。

第14章　NLL（Non-Lexical-Lifetime）

Rust防范“内存不安全”代码的原则极其清晰明了。如果你对同一块内存存在多个引用，就不要试图对这块内存做修改；如果你需要对一块内存做修改，就不要同时保留多个引用。只要保证了这个原则，我们就可以保证内存安全。它在实践中发挥了强大的作用，可以帮助我们尽早发现问题。这个原则是Rust的立身之本、生命之基、活力之源。

这个原则是没问题的，但是，初始的实现版本有一个主要问题，那就是它让借用指针的生命周期规则与普通对象的生命周期规则一样，是按作用域来确定的。所有的变量、借用的生命周期就是从它的声明开始，到当前整个语句块结束。这个设计被称为Lexical Lifetime，因为生命周期是严格和词法中的作用域范围绑定的。这个策略实现起来非常简单，但它可能过于保守了，某些情况下借用的范围被过度拉长了，以至于某些实质上是安全的代码也被阻止了。在某些场景下，限制了程序员的发挥。

因此，Rust核心组又决定引入Non Lexical Lifetime，用更精细的手段调节借用真正起作用的范围。这就是NLL。

注意：在编写本书时，该功能在编译器中只实现了一小部分。

14.1 NLL希望解决的问题

首先，我们来看几个简单的示例。

```
use std::ascii::AsciiExt;

fn foo() -> Vec<char> {
    let mut data = vec!['a', 'b', 'c']; // --+ 'scope
    capitalize(&mut data[..]);          //   |
//  ^~~~~~~~~~~~~~~~~~~~~~~~~ 'lifetime //   |
    data.push('d');                     //   |
    data.push('e');                     //   |
    data.push('f');                     //   |
    data                                //   |
//  <---------------------------------------+
}

fn capitalize(data: &mut [char]) {
    for c in data {
        c.make_ascii_uppercase();
    }
}

fn main() {
    let v = foo();
    println!("{:?}", v);
}
```

这段代码是没有问题的。我们的关注点是foo（）这个函数，它在调用capitalize函数的时候，创建了一个临时的&mut型引用，在它的调用结束后，这个临时的借用就终止了，因此，后面我们就可以再用data去修改数据。注意，这个临时的&mut引用存在的时间很短，函数调用结束，它的生命周期就结束了。

但是，如果我们把这段代码稍作修改，问题就出现了：

```
fn foo() -> Vec<char> {
    let mut data = vec!['a', 'b', 'c']; // --+ 'scope
    let slice = &mut data[..];// <-----------+ 'lifetime
    capitalize(slice);                  //   |
    data.push('d');  //ERROR            //   |
    data.push('e');  //ERROR            //   |
    data.push('f');  //ERROR            //   |
    data             //ERROR            //   |
//  <---------------------------------------+
}
```

在这段代码中，我们创建了一个临时变量slice，保存了一个指向data的&mut型引用，然后再调用capitalize函数，就出问题了。编译器提示为：

```
error[E0499]: cannot borrow `data` as mutable more than once at a time
```

这是因为，Rust规定“共享不可变，可变不共享”，同时出现两个&mut型借用是违反规则的。在编译器报错的地方，编译器认为slice依然存在，然而又使用data去调用fn push（&mut self，value：T）方法，必然又会产生一个&mut型借用，这违反了Rust的原则。在目前这个版本中，如果我们要修复这个问题，只能这样做：

```
fn foo() -> Vec<char> {
    let mut data = vec!['a', 'b', 'c']; // --+ 'scope
    {
        let slice = &mut data[..];      // <-------+ 'lifetime
        capitalize(slice);              //   |
    }// <-----------------------------------+
    data.push('d');
    data.push('e');
    data.push('f');
    data
}
```

我们手动创建了一个代码块，让slice在这个子代码块中创建，后面就不会产生生命周期冲突问题了。这是因为，在早期的编译器内部实现里面，所有的变量，包括引用，它们的生命周期都是从声明的地方开始，到当前语句块结束（不考虑所有权转移的情况）。

这样的实现方式意味着每个引用的生命周期都是跟代码块（scope）相关联的，它总是从声明的时候被创建，在退出这个代码块的时候被销毁，因此可以称为Lexical lifetime。而本章所说的Non-Lexical lifetime，意思就是取消这个关联性，引用的生命周期，我们用另外的、更智能的方式分析。有了这个功能，上例中手动加入的代码块就不需要了，编译器应该能自动分析出来，slice这个引用在capitalize函数调用后就再没有被使用过了，它的生命周期完全可以就此终止，不会对程序的正确性有任何影响，后面再调用push方法修改数据，其实跟前面的slice并没有什么冲突关系。

看了上面这个例子，可能有人还会觉得，显式的用一个代码块来规

定局部变量的生命周期是个更好的选择，Non-Lexical-Lifetime的意义似乎并不大。那我们再继续看看更复杂的例子。我们可以发现，Non-Lexical-Lifetime可以打开更多的可能性，让用户有机会用更直观的方式写代码。比如下面这样的一个分支结构的程序：

```
fn process_or_default<K,V:Default>
    (map: &mut HashMap<K,V>, key: K)
{
    match map.get_mut(&key) { // -------------+ 'lifetime
        Some(value) => process(value),     // |
        None => {                          // |
            map.insert(key, V::default()); // |
            //  ^~~~~ ERROR.               // |
        }                                  // |
    } // <-----------------------------------+
}
```

这段代码从一个HashMap中查询某个key是否存在。如果存在，就继续处理，如果不存在，就插入一个新的值。目前这段代码是编译不过的，因为编译器会认为在调用get_mut（&key）的时候，产生了一个指向map的&mut型引用，而且它的返回值也包含了一个引用，返回值的生命周期是和参数的生命周期一致的。这个方法的返回值会一直存在于整个match语句块中，所以编译器判定，针对map的引用也是一直存在于整个match语句块中的。于是后面调用insert方法会发生冲突。

当然，如果我们从逻辑上来理解这段代码，就会知道，这段代码其实是安全的。因为在None分支，意味着map中没有找到这个key，在这条路径上自然也没有指向map的引用存在。但是可惜，在老版本的编译器上，如果我们希望让这段代码编译通过，只能绕一下。我们试一下做如下的修复：

```
fn get_default1<'m,K,V:Default>(
    map: &'m mut HashMap<K,V>,key: K)
  -> &'m mut V
{
    match map.get_mut(&key) { // -------------+ 'm
        Some(value) => return value,       // |
        None => { }                        // |
    }                                      // |
    map.insert(key, V::default());         // |
    //  ^~~~~ ERROR (still)                   |
    map.get_mut(&key).unwrap()             // |
}                                          // v
```

实际上这个改动依然会编译失败。原因就在于return语句，get_mut时候对map的借用传递给了Some（value），在Some这个分支内存在一个引用，指向map的某个部分，而我们又把value返回了，这意味着编译器认为，这个借用从match开始一直到退出这个函数都存在。因此后面的insert调用依然发生了冲突。接下来我们再做一次修复：

```
fn get_default2<'m,K,V:Default>(
    map: &'m mut HashMap<K,V>,
    key: K)
  -> &'m mut V
{
    if map.contains(&key) {
    // ^~~~~~~~~~~~~~~~~~ 'n
        return match map.get_mut(&key) { // + 'm
            Some(value) => value,        // |
            None => unreachable!()       // |
        };                               // v
    }

    // At this point, `map.get_mut` was never
    // called! (As opposed to having been called,
    // but its result no longer being in use.)
    map.insert(key, V::default()); // OK now.
    map.get_mut(&key).unwrap()
}
```

这次的区别在于，get_mut发生在一个子语句块中。在这种情况下，编译器会认为这个借用跟if外面的代码没什么关系。通过这种方式，我们终于绕过了borrow checker。但是，为了绕过编译器的限制，我们付出了一些代价。这段代码，我们需要执行两次hash查找，一次在contains方法，一次在get_mut方法，因此它有额外的性能开销。这也是为什么标准库中的HashMap设计了一个叫作entry的api，如果用entry来写这段逻辑，可以这么做：

```
fn get_default3<'m,K,V:Default>(
    map: &'m mut HashMap<K,V>,
    key: K)
  -> &'m mut V
{
    map.entry(key)
        .or_insert_with(|| V::default())
}
```

这个设计既清晰简洁，也没有额外的性能开销，而且不需要Non-Lexical-Lifetime的支持。这说明，虽然老版本的生命周期检查确实有点

过于严格，但至少在某些场景下，我们其实还是有办法绕过去的，不一定要在“良好的抽象”和“安全性”之间做选择。但是它付出了其他的代价，那就是设计难度更高，更不容易被掌握。标准库中的entry API也是很多高手经过很长时间才最终设计出来的产物。对于普通用户而言，如果在其他场景下出现了类似的冲突，恐怕大部分人都没有能力想到一个最佳方案，可以既避过编译器限制，又不损失性能。所以在实践中的很多场景下，普通用户做不到“零开销抽象”。

让编译器能更准确地分析借用指针的生命周期，不要简单地与scope相绑定，不论对普通用户还是高阶用户都是一个更合理、更有用的功能。如果编译器能有这么聪明，那么它应该能理解下面这段代码其实是安全的：

```
match map.get_mut(&key) {
        Some(value) => process(value), // 找到了就继续处理这个值
        None => {
            map.insert(key, V::default()); // 没找到key就插入一个新的值
        }
    }
```

这段代码既符合用户直观思维模式，又没有破坏Rust的安全原则。以前的编译器无法编译通过，实际上是对正确程序的误伤，是一种应该修复的缺陷。NLL的设计目的就是让Rust的安全检查更加准确，减少误报，使得编译器对程序员的掣肘更少。

打开NLL功能，以下代码就可以编译通过了：

```
#![feature(nll)]

use std::collections::HashMap;

fn process_or_default4(map: &mut HashMap<String, String>, key: String)
{
    match map.get_mut(&key) {
        Some(value) => println!("{}", value),
        None => {
            map.insert(key, String::new());
        }
    }
}

fn main() {
    let mut map = HashMap::<String, String>::new();
    let key = String::from("abc");
    process_or_default4(&mut map, key);
```

```
}
```

14.2 NLL的原理

NLL的设计目的是让“借用”的生命周期不要过长，适可而止，避免不必要的编译错误，把实际上正确的代码也一起拒绝掉。但是实现方法不能是简单地在AST上找以下某个引用最后一次在哪里使用，就让它的生命周期结束算了。我们用例子来说明：

```
fn baz() {
    let mut data = vec!['a', 'b', 'c'];
    let slice = &mut data[..]; // <-+ lifetime if we ignored
    loop {                     //   | variables altogether
        capitalize(slice);     //   |
        // <-----------------------+
        data.push('d'); // Should be error, but would not be.
    }
    data.push('e');     // OK
    data.push('f');     // OK
}
```

在这个示例中，我们引入了一个循环结构。如果我们只是分析AST的结构的话，很可能会觉得capitalize函数结束后，slice的生命周期就结束了，因此data.push（）方法调用是合理的。但这个结论是错误的，因为这里有一个循环结构。大家想想看，如果执行了push（）方法后，引发了Vec数据结构的扩容，它把以前的空间释放掉，申请了新的空间，进入下一轮循环的时候，slice就会指向一个非法地址，会出现内存不安全。以上这段代码理应出现编译错误。

因此，新版本的借用检查器将不再基于AST的语句块来设计，而是将AST转换为另外一种中间表达形式MIR（middle-level intermediate representation）之后，在MIR的基础上做分析。这是因为前面已经分析过了，对于复杂一点的程序逻辑，基于AST来做生命周期分析是费力不讨好的事情，而MIR则更适合做这种分析。读者可以用以下编译器命令打印出MIR的文本格式：

```
rustc --emit=mir test.rs
```

不过在一般情况下，MIR在编译器内部的表现形式是内存中的一组数据结构。这些数据结构描述的是一个叫作“控制流图”（control flow

graph）的概念。所谓控制流图，就是用“图”这种数据结构，描述程序的执行流程。每个函数都有一个MIR来描述它，比如对于以下这段代码：

```
fn send_if2(data: Vec<Data>) {
    if some_condition(&data) {
        send_to_other_thread(data);
        return;
    }

    process(&data);
}
```

生成的控制流图如图14-1所示。

图上面有节点，也有边。节点代表一条或者一组语句，边代表分支跳转。有了这个图，引用的生命周期就可以用这个图上的节点来表示了。编译器最后会分析出来，引用在这个图上的哪些节点上还是活着的，在哪些节点上可以看作已经死掉了。相比于以前，一个引用的生命周期直接充满整个语句块，现在的表达方式明显要精细得多，这样我们就可以保证引用的生命周期不会被过分拉长。

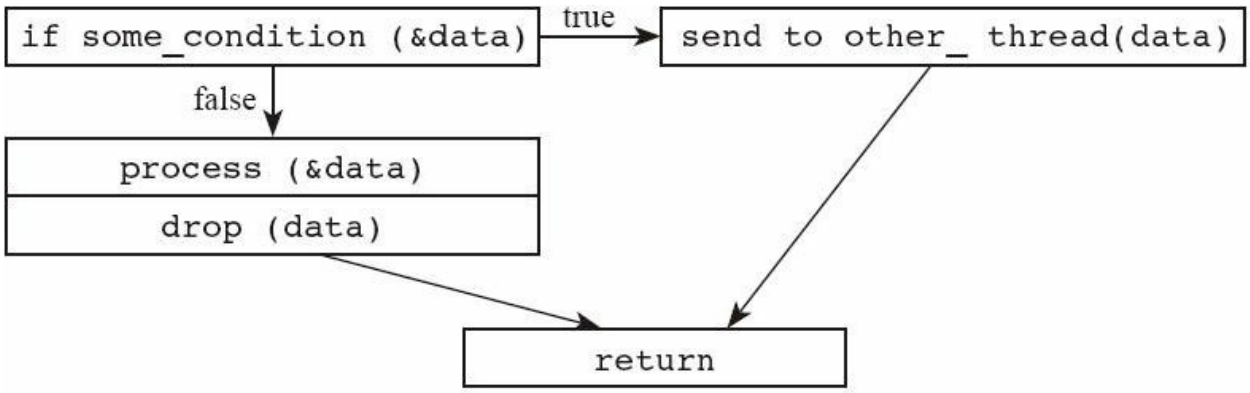


图　14-1

这个新版的借用分析器，会允许下面的代码编译成功，比如：

```
#![feature(nll)]
fn main() {
    let mut v = vec![1,2,3,4,5];
    v.push(v.len()); // 同一行,既有 & 型借用,也有 &mut 型借用。但逻辑上是安全的
    println!("{:?}", v);
}
```

目前版本中，如果去掉#！[feature（nll）]就会出现编译错误。

再比如：

```
#![feature(nll)]
fn main() {
    let mut data = 100_i32;
    let mut p = &data; // p is live
    println!("{}", p); // p is dead

    data = 101;
    p = &data;         // p is live again
    println!("{}", p); // p is dead again
}
```

上面这个示例启用了新的生命周期分析器后，也可以编译成功。它不会再把p指针的生命周期当成从声明到语句块结束，而是聪明地查出了到第一次println！就可以结束了，后面的data又重新赋值的时候不会跟它冲突。

另外需要强调的是：

·这个功能只影响静态分析结果，不影响程序的执行情况；

·以前能编译通过的程序以后依然会编译通过，不会影响以前的代码；

·它依然保证了安全性，只是将以前过于保守的检查规则适当放宽；

·它依赖的依然是静态检查规则，不会涉及任何动态检查规则；

·它只影响“引用类型”的生命周期，不影响“对象”的生命周期，即维持现有的析构函数调用时机不变；

·它不会影响RAII语义。

在编写本书之时，此功能还没有完全实现，但是鉴于该功能的重要性，笔者依然觉得非常有必要向各位读者提前介绍。希望该功能尽快完成，以减少不必要的编译错误对新手的困扰。

14.3 小结

内存安全是需要一些代码规范来约束才能实现的。这就好比交通安全是需要交通法规来约束才能实现一样。我们不能因为某个具体的人在某个具体的路段上不能直达目标，不得不绕路而行，而否定整个交通法规的意义。交通法规本身肯定还有值得优化的地方，但它优化的方向应该是让社会平均交通事故率下降，提高社会平均通行效率，不能着眼于特定时间特定位置。NLL的设计就是朝这个方向前进的。Rust中的生命周期是一个初学者不易理解的难点，而且也确实存在一些情况损害了语言的表达能力。但我们不应该轻易地否定生命周期这个设计，而是应该做一些更精细的、准确的调整，使它尽可能接近“安全”与“不安全”的那条分界线，不偏不倚，否则宽严皆误。

在C/C++的领域，为了提高软件可靠性而设计的代码规范其实有很多，这些先贤总结出来的C/C++的设计范式和惯用法是非常有利于提高代码质量、降低安全风险的。但是可惜的是，这些良好的设计范式并不是所有人都能遵守的，就好比大街上总是有人试图抄近路，违反交通规则而导致交通事故一样。这些人不理解一个统一的代码规范对于整个项目代码质量的意义。一旦在项目中掺入了一些带有“坏味道”的代码，那它就不一定在哪天给项目带来一个出其不意的问题，哪怕只是一个简单的赋值、函数调用，都可能触发完全超过预期的结果。

A rule is worthless if it is not enforced.如果规则不能强制执行，那么这个规则就是没有价值的。因此，C/C++的领域内也出现了一大批静态代码检查工具。这些工具可以帮助我们提高代码规范的严肃性和可执行性，从而间接增强代码的可靠程度。但是可惜的是，C/C++的静态代码检查工具远称不上完美，它总会有误报或者漏报的情况发生。而Rust则在这个方向上更进了一步，保证了segment fault这一类内存安全问题，可以在静态代码检查阶段“完整无遗漏”地被检查出来。

Rust之所以能达到这么好的检查效果，并不是因为设计者发明了什么高深的算法，或者比做C/C++静态检查的人更聪明。主要原因是设计者们“作弊”了，他们直接简化了被研究对象。C/C++由于本质上的灵活性太强，使得某些安全问题检查极其难以实现。而Rust在设计的时候就一直将这些问题考虑在内，所有的功能都不能影响“内存安全”的设计。

当然，Rust的这种设计自然就带来了Rust独有的新的设计范式。有些C/C++风格的代码在Rust里面就很难写出来，但这并不一定意味着“表达能力低”，往往是因为用地道的Rust处理同样的问题是另外一种你不熟悉的风格而已。要搞清楚这种Rust风格的代码究竟是怎样的，就需要多读高质量的开源的Rust代码来培养感觉。

第15章　内部可变性

Rust的borrow checker的核心思想是“共享不可变，可变不共享”。但是只有这个规则是不够的，在某些情况下，我们的确需要在存在共享的情况下可变。为了让这种情况是可控的、安全的，Rust还设计了一种“内部可变性”（interior mutability）。

“内部可变性”的概念，是与“承袭可变性”（inherited mutability）相对应的。大家应该注意到了，Rust中的mut关键字不能在声明类型的时候使用，只能跟变量一起使用。类型本身不能规定自己是否是可变的。一个变量是否是可变的，取决于它的使用环境，而不是它的类型。可变还是不可变取决于变量的使用方式，这就叫作“承袭可变性”。如果我们用let var：T；声明，那么var是不可变的，同时，var内部的所有成员也都是不可变的；如果我们用let mut var：T；声明，那么var是可变的，相应的，它的内部所有成员也都是可变的。我们不能在类型声明的时候指定可变性，比如在struct中对某部分成员使用mut修饰，这是不合法的。我们只能在变量声明的时候指定可变性。我们也不能针对变量的某一部分成员指定可变性，其他部分保持不变。

常见的具备内部可变性特点的类型有Cell、RefCell、Mutex、RwLock、Atomic*等。其中Cell和RefCell是只能用在单线程环境下的具备内部可变性的类型。下面就来讲解何为“内部可变性”。

15.1 Cell

按照前面的理论，如果我们有共享引用指向一个对象，那么这个对象就不会被更改了。因为在共享引用存在的期间，不能有可变引用同时指向它，因此它一定是不可变的。其实在Rust中，这种想法是不准确的。下面给出一个示例：

```
use std::rc::Rc;

fn main() {
    let r1 = Rc::new(1);
    println!("reference count {}", Rc::strong_count(&r1));
    let r2 = r1.clone();
    println!("reference count {}", Rc::strong_count(&r2));
}
```

编译，执行，结果为：

```
reference count 1
reference count 2
```

Rc是Rust里面的引用计数智能指针，在后文中我们还会继续讲解。多个Rc指针可以同时指向同一个对象，而且有一个共享的引用计数值在记录总共有多少个Rc指针指向这个对象。

注意Rc指针提供的是共享引用，按道理它没有修改共享数据的能力。但是我们用共享引用调用clone方法，引用计数值发生了变化。这就是我们要说的“内部可变性”。如果没有内部可变性，标准库中的Rc类型是无法正确实现出来的。具备内部可变性的类型，最典型的就是Cell。

现在用一个更浅显的例子来演示一下Cell的能力：

```
use std::cell::Cell;

fn main() {
    let data : Cell<i32> = Cell::new(100);
    let p = &data;
    vdata.set(10);
    println!("{}", p.get());
```

```
    p.set(20);
    println!("{:?}", data);
}
```

这次编译通过，执行，结果是符合我们的预期的：

```
10
Cell { value: 20 }
```

请注意这个例子最重要的特点。需要注意的是，这里的“可变性”问题跟我们前面见到的情况不一样了。data这个变量绑定没有用mut修饰，p这个指针也没有用&mut修饰，然而不可变引用竟然可以调用set函数，改变了变量的值，而且还没有出现编译错误。

这就是所谓的内部可变性——这种类型可以通过共享指针修改它内部的值。虽然粗略一看，Cell类型似乎违反了Rust的“唯一修改权”原则。我们可以存在多个指向Cell类型的不可变引用，同时我们还能利用不可变引用改变Cell内部的值。但实际上，这个类型是完全符合“内存安全”的。我们再想想，为什么Rust要尽力避免alias和mutation同时存在？因为假如我们同时有可变指针和不可变指针指向同一块内存，有可能出现通过一个可变指针修改内存的过程中，数据结构处于被破坏状态的情况下，被其他的指针观测到。Cell类型是不会出现这样的情况的。因为Cell类型把数据包裹在内部，用户无法获得指向内部状态的指针，这意味着每次方法调用都是执行的一次完整的数据移动操作。每次方法调用之后，Cell类型的内部都处于一个正确的状态，我们不可能观察到数据被破坏掉的状态。

多个共享指针指向Cell类型的状态就类似图15-1所示的这样，Cell就是一个“壳”，它把数据严严实实地包裹在里面，所有的指针只能指向Cell，不能直接指向数据。修改数据只能通过Cell来完成，用户无法创造一个直接指向数据的指针。

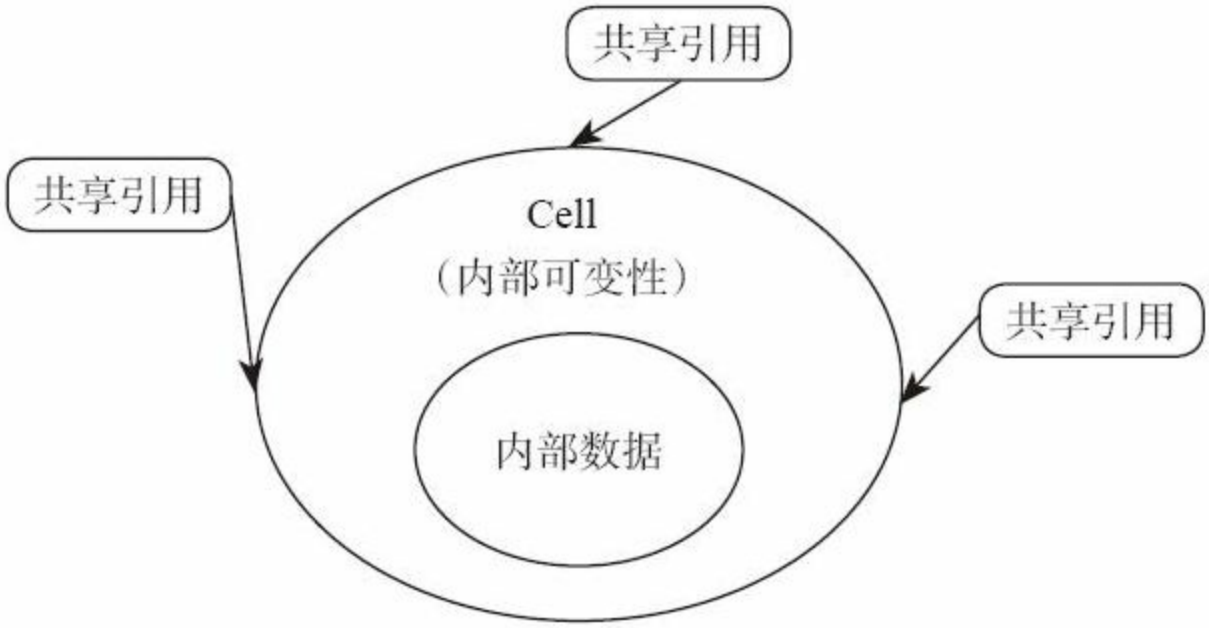


图　15-1

我们来仔细观察一下Cell类型提供的公开的API，就能理解Cell类型设计的意义了。下面是Cell类型提供的几个主要的成员方法：

```
impl<T> Cell<T> {

    pub fn get_mut(&mut self) -> &mut T {    }

    pub fn set(&self, val: T) {    }

    pub fn swap(&self, other: &Self) {    }

    pub fn replace(&self, val: T) -> T {    }

    pub fn into_inner(self) -> T {    }
}

impl<T:Copy> Cell<T> {

    pub fn get(&self) -> T {    }
}
```

·get_mut方法可以从&mut Cell<T>类型制造出一个&mut T型指针。因为&mut型指针具有“独占性”，所以这个函数保证了调用前，有且仅有一个“可写”指针指向Cell，调用后有且仅有一个“可写”指针指向内部数据。它不存在制造多个引用指向内部数据的可能性。

·set方法可以修改内部数据。它是把内部数据整个替换掉，不存在

多个引用指向内部数据的可能性。

·swap方法也是修改内部数据。跟set方法一样，也是把内部数据整体替换掉。与std：：mem：：swap函数的区别在于，它仅要求&引用，不要求&mut引用。

·replace方法也是修改内部数据。跟set方法一样，它也是把内部数据整体替换，唯一的区别是，换出来的数据作为返回值返回了。

·into_inner方法相当于把这个“壳”剥掉了。它接受的是Self类型，即move语义，原来的Cell类型的变量会被move进入这个方法，会把内部数据整体返回出来。

·get方法接受的是&self参数，返回的是T类型，它可以在保留之前Cell类型不变的情况下返回一个新的T类型变量，因此它要求T：Copy约束。每次调用它的时候，都相当于把内部数据memcpy了一份返回出去。

正因为上面这些原因，我们可以看到，Cell类型虽然违背了“共享不可变，可变不共享”的规则，但它并不会造成内存安全问题。它把“共享且可变”的行为放在了一种可靠、可控、可信赖的方式下进行。它的API是经过仔细设计过的，绝对不可能让用户有机会通过&Cetl<T>获得&T或者&mut T。它是对alias+mutation原则的有益补充，而非完全颠覆。大家可以尝试一下用更复杂的例子（如Cell<Vec<i32>>）试试，看能不能构造出内存不安全的场景。

15.2 RefCell

RefCell是另外一个提供了内部可变性的类型。它提供的方式与Cell类型有点不一样。Cell类型没办法制造出直接指向内部数据的指针，而RefCell可以。我们来看一下它的API：

```
impl<T: ?Sized> RefCell<T> {

    pub fn borrow(&self) -> Ref<T> {    }

    pub fn try_borrow(&self) -> Result<Ref<T>, BorrowError> {    }

    pub fn borrow_mut(&self) -> RefMut<T> {    }

    pub fn try_borrow_mut(&self) -> Result<RefMut<T>, BorrowMutError> {    }

    pub fn get_mut(&mut self) -> &mut T {    }
}
```

get_mut方法与Cell：：get_mut一样，可以通过&mut self获得&mut T，这个过程是安全的。除此之外，RefCell最主要的两个方法就是borrow和borrow_mut，另外两个try_borrow和try_borrow_mut只是它们俩的镜像版，区别仅在于错误处理的方式不同。

我们还是用示例来演示一下RefCell怎样使用：

```
use std::cell::RefCell;

fn main() {
    let shared_vec: RefCell<Vec<isize>> = RefCell::new(vec![1, 2, 3]);
    let shared1 = &shared_vec;
    let shared2 = &shared1;

    shared1.borrow_mut().push(4);
    println!("{:?}", shared_vec.borrow());

    shared2.borrow_mut().push(5);
    println!("{:?}", shared_vec.borrow());
}
```

在这个示例中，我们用一个RefCell包了一个Vec，并且制造了另外两个共享引用指向同一个RefCell。这时，我们可以通过任何一个共享引用调用borrow_mut方法，获得指向内部数据的“可写”指针，通过这个指

针调用了push方法，修改内部数据。同时，我们也可以通过调用borrow方法获得指向内部数据的“只读”指针，读取Vec里面的值。

编译，执行，结果为：

```
$ ./test
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
```

这里有一个小问题需要跟大家解释一下：在函数的签名中，borrow方法和borrow_mut方法返回的并不是&T和&mut T，而是Ref<T>和RefMut<T>。它们实际上是一种“智能指针”，完全可以当作&T和&mut T的等价物来使用。标准库之所以返回这样的类型，而不是原生指针类型，是因为它需要这个指针生命周期结束的时候做点事情，需要自定义类型包装一下，加上自定义析构函数。至于包装起来的类型为什么可以直接当成指针使用，它的原理可以参考下一章“解引用”。

那么问题来了：如果borrow和borrow_mut这两个方法可以制造出指向内部数据的只读、可读写指针，那么它是怎么保证安全性的呢？像前几章讲的那样，如果同时构造了只读引用和可读写引用指向同一个Vec，那不是很容易就构造出悬空指针么？答案是，RefCell类型放弃了编译阶段的alias+mutation原则，但依然会在执行阶段保证alias+mutation原则。示例如下：

```
use std::cell::RefCell;

fn main() {
    let shared_vec: RefCell<Vec<isize>> = RefCell::new(vec![1, 2, 3]);
    let shared1 = &shared_vec;
    let shared2 = &shared1;

    let p1 = shared1.borrow();
    let p2 = &p1[0];

    shared2.borrow_mut().push(4);
    println!("{}", p2);
}
```

上面这个示例的意图是：我们先调用borrow方法，并制造一个指向数组第一个元素的指针，接着再调用borrow_mut方法，修改这个数组。这样，就构造出了同时出现alias和mutation的场景。

编译，通过。执行，问题来了，程序出现了panic：

```
$ ./test
thread 'main' panicked at 'already borrowed: BorrowMutError', src\libcore\result.rs:
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

出现panic的原因是，RefCell探测到同时出现了alias和mutation的情况，它为了防止更糟糕的内存不安全状态，直接使用了panic来拒绝程序继续执行。如果我们用try_borrow方法的话，就会发现返回值是Result：：Err，这是另外一种更友好的错误处理风格。

那么RefCell是怎么探测出问题的呢？原因是，RefCell内部有一个“借用计数器”，调用borrow方法的时候，计数器里面的“共享引用计数”值就加1。当这个borrow结束的时候，会将这个值自动减1（如图15-2所示）。同样，borrow_mut方法被调用的时候，它就记录一下当前存在“可变引用”。如果“共享引用”和“可变引用”同时出现了，就会报错。

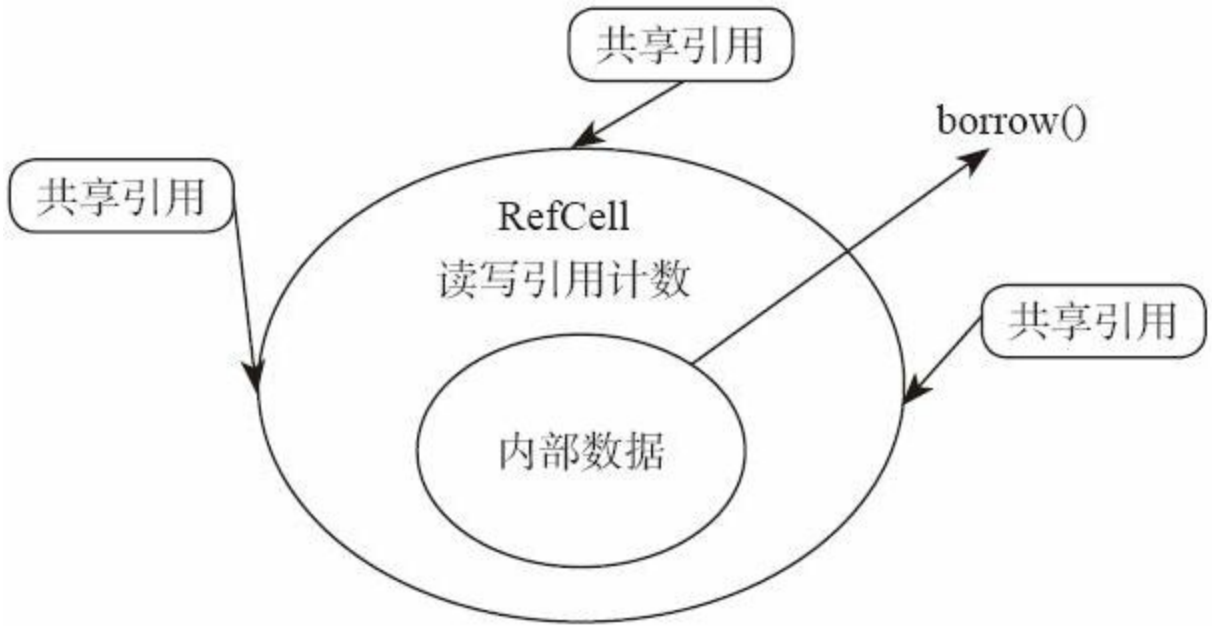


图　15-2

从原理上来说，Rust默认的“借用规则检查器”的逻辑非常像一个在编译阶段执行的“读写锁”（read-write-locker）。如果同时存在多个“读”的锁，是没问题的；如果同时存在“读”和“写”的锁，或者同时存在多个“写”的锁，就会发生错误。RefCell类型并没有打破这个规则，只

不过，它把这个检查逻辑从编译阶段移到了执行阶段。RefCell让我们可以通过共享引用&修改内部数据，逃过编译器的静态检查。但是它依然在兢兢业业地尽可能保证“内存安全”。我们需要的借用指针必须通过它提供的API borrow（）borrow_mut（）来获得，它实际上是在执行阶段，在内部维护了一套“读写锁”检查机制。一旦出现了多个“写”或者同时读写，就会在运行阶段报错，用这种办法来保证写数据时候的执行过程中的内部状态不会被观测到，任何时候，开始读或者开始写操作开始的时候，共享的变量都处于一个合法状态。因此在执行阶段，RefCell是有少量开销的，它需要维护一个借用计数器来保证内存安全。

所以说，我们一定不要过于滥用RefCell这样的类型。如果确有必要使用，请一定规划好动态借用出来的指针存活时间，否则会在执行阶段有问题。

Cell和RefCell用得最多的场景是和多个只读引用相配合。比如，多个&引用或者Rc引用指向同一个变量的时候。我们不能直接通过这些只读引用修改变量，因为既然存在alias，就不能提供mutation。为了让存在多个alias共享的变量也可以被修改，那我们就需要使用内部可变性。Rust中提供了只读引用的类型有&、Rc、Arc等指针，它们可以提供alias。Rust中提供了内部可变性的类型有Cell、RefCell、Mutex、RwLock以及Atomic*系列类型等。这两类类型经常需要配合使用。

如果你需要把一个类型T封装到内部可变性类型中去，要怎样选择Cell和RefCell呢？原则就是，如果你只需要整体性地存入、取出T，那么就选Cell。如果你需要有个可读写指针指向这个T修改它，那么就选RefCell。

15.3 UnsafeCell

接下来，我们来分析Cell/RefCell的实现原理。我们先来考虑两个问题，标准库中的Cell类型是怎样实现的？假如让我们自己来实现一遍，是否可行呢？

模仿标准库中的Cell类型的公开方法（只考虑最简单的new、get、set这三个方法），我们先来一个最简单的版本V1：

```
struct CellV1<T> {
    value: T
}

impl<T> CellV1<T> {

    fn new(v: T) -> Self where T: Copy {
        CellV1 { value: v}
    }

    fn set(&self, v: T) {
        self.value = v;
    }

    fn get(&self) -> T where T: Copy {
        self.value
    }
}
```

这个版本是一个new type类型，内部包含了一个T类型的成员。成员方法对类型T都有恰当的约束。这些都没错。只有一个关键问题需要注意：对于set方法，直接这样写是肯定行不通的，因为self是只读引用，我们不可能直接对self.value赋值。而且，Cell类型最有用的地方就在于，它可以通过不可变引用改变内部的值。那么这个问题怎么解决呢？可以使用unsafe关键字。后面还有一章专门讲解unsafe，此处只需知道，用unsafe包起来的代码块可以突破编译器的一些限制，做一些平常不能做的事情。

以下是修正版：

```
struct CellV2<T> {
    value: T
}
```

```
impl<T> CellV2<T> {

    fn new(v: T) -> Self where T: Copy {
        CellV2 { value: v}
    }

    fn set(&self, v: T) where T: Copy {
        unsafe {
            let p = &(self.value) as *const T as *mut T;//此处实际上引入了未定义行为
            *p = v;
        }
    }

    fn get(&self) -> T where T: Copy {
        self.value
    }
}
```

在使用unsafe语句块之后，这段代码可以编译通过了。这里的关键是，在unsafe代码中，我们可以把*const T类型强制转换为*mut T类型。这是初学者最直观的解决方案，但这个方案是错误的。通过这种方式，我们获得了写权限。通过下面简单的示例可以看到，这段代码是符合我们的预期的：

```
fn main() {
    let c = CellV2::new(1_isize);
    let p = &c;
    p.set(2);
    println!("{}", c.get());
}
```

从以上代码可以看出，这正是内部可变性类型的特点，即通过共享指针，修改了内部的值。

事情就这么简单么？很可惜，有这种想法的人都过于naive了。下面这个示例会给大家泼一盆冷水：

```
struct Table<'arg> {
    cell: CellV2<&'arg isize>
}

fn evil<'long,'short>(t: &Table<'long>, s: &'short isize)
    where 'long : 'short
{
    // The following assignment is not legal, but it escapes from lifetime checking
    let u: &Table<'short> = t;
    u.cell.set(s);
}
```

```
fn innocent<'long>(t: &Table<'long>) {
    let foo: isize = 1;
    evil(t, &foo);
}

fn main() {
    let local = 100;
    let table = Table { cell: CellV2::new(&local) };
    innocent(&table);
    // reads `foo`, which has been destroyed
    let p = table.cell.get();
    println!("{}", p);
}
```

如果我们用rustc temp.rs编译debug版本，可以看到执行结果为1。

如果我们用rustc-O temp.rs编译release版本，可以看到执行结果为140733369053192。

这是怎么回事呢？因为这段代码中出现了野指针。我们来分析一下这段测试代码。在这段测试代码中，我们在CellV2类型里面保存了一个引用。main函数调用了innocent函数，继而又调用了evil函数。这里需要特别注意的是：在evil函数中，我们调用了CellV2类型的set方法，改变了它里面存储的指针。修改后的指针指向的谁呢？是innocent函数内部的一个局部变量。最后在main函数中，innocent函数返回后，再把这个CellV2里面的指针拿出来使用，就得到了一个野指针。

我们继续从生命周期的角度深入分析，这个野指针的成因。在main函数的开始，table.cell变量保存了一个指向local变量的指针。这是没问题的，因为local的生命周期比table更长，table.cell指向它肯定不会有问题。有问题的是table.cell在evil函数中被重新赋值。这个赋值导致了table.cell保存了一个指向局部调用栈上的变量。也就是这里出的问题：

```
// t: &Table<'long>
let u: &Table<'short> = t;
// s: &'short isize
u.cell.set(s);
```

我们知道，在'long：'short的情况下，&'long类型的指针向&'short类型赋值是没问题的。但是这里的&Table<'long>类型的变量赋值给&Table<'short>类型的变量合理吗？事实证明，不合理。证明如下。我们把上例中的CellV2类型改用标准库中的Cell类型试试：

```
type CellV2<T> = std::cell::Cell<T>;
```

其他测试代码不变。编译，提示错误为：

```
error[E0308]: mismatched types
  --> temp.rs:11:29
   |
11 |   let u: &Table<'short> = t;
   |                         ^ lifetime mismatch
   |
   = note: expected type `&Table<'short>`
   = note:    found type `&Table<'long>`
```

果然是这里的问题。使用我们自己写的CellV2版本，这段测试代码可以编译通过，并制造出了内存不安全。使用标准库中的Cell类型，编译器成功发现了这里的生命周期问题，给出了提示。

这说明了CellV2的实现依然是错误的。虽然最基本的测试用例通过了，但是碰到复杂的测试用例，它还是不够“健壮”。而Rust对于“内存不安全”问题是绝对禁止的。不像C/C++，在Rust语言中，如果有机会让用户在不用unsafe的情况下制造出内存不安全，这个责任不是由用户来承担，而是应该归因于写编译器或者写库的人。在Rust中，写库的人不需要去用一堆文档来向用户保证内存安全，而是必须要通过编译错误来保证。这个示例中的内存安全问题，不能归因于测试代码写得不对，因为在测试代码中没有用到任何unsafe代码，用户是正常使用而已。这个问题出现的根源还是CellV2的实现有问题，具体来说就是那段unsafe代码有问题。按照Rust的代码质量标准，CellV2版本是完全无法接受的垃圾代码。

那么，这个bug该如何修正呢？为什么&'long类型的指针可以向&'short类型赋值，而&Cell<'long>类型的变量不能向&Cell<'short>类型的变量赋值？因为对于具有内部可变性特点的Cell类型而言，它里面本来是要保存&'long型指针的，结果我们给了它一个&'short型指针，那么在后面取出指针使用的时候，这个指针所指向的内容已经销毁，就出现了野指针。这个bug的解决方案是，禁止具有内部可变性的类型，针对生命周期参数具有“协变/逆变”特性。这个功能是通过标准库中的UnsafeCell类型实现的：

```
#[lang = "unsafe_cell"]
#[stable(feature = "rust1", since = "1.0.0")]
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

请注意这个类型上面的标记#[lang=...]。这个标记意味着这个类型是个特殊类型，是被编译器特别照顾的类型。这个类型的说明文档需要特别提示读一下：

```
The core primitive for interior mutability in Rust.

UnsafeCell<T> is a type that wraps some T and indicates unsafe interior operations c

Types like Cell<T> and RefCell<T> use this type to wrap their internal data.
```

所有具有内部可变性特点的类型都必须基于UnsafeCell来实现，否则必然出现各种问题。这个类型是唯一合法的将&T类型转为&mut T类型的办法。绝对不允许把&T直接转换为&mutT而获得可变性。这是未定义行为。

大家可以自行读一下Cell和RefCell的源码，可以发现，它们能够正常工作的关键在于它们都是基于UnsafeCell实现的，而UnsafeCell本身是编译器特殊照顾的类型。所以我们说“内部可变性”这个概念是Rust语言提供的一个核心概念，而不是通过库模拟出来的。

实际上，上面那个CellV2示例也正说明了写unsafe代码的困难之处。许多时候，我们的确需要使用unsafe代码来完成功能，比如调用C代码写出来的库等。但是却有可能一不小心违反了Rust编译器的规则。比如，你没读过上面这段文档的话，不大可能知道简单地通过裸指针强制类型转换实现&T到&mut T的类型转换是错误的。这么做会在编译器的生命周期静态检查过程中制造出一个漏洞，而且这个漏洞用简单的测试代码测不出来，只有在某些复杂场景下才会导致内存不安全。Rust代码中写unsafe代码最困难的地方其实就在这样的细节中，有些人在没有完全理解掌握Rust的safe代码和unsafe代码之间的界限的情况下，乱写unsafe代码，这是不负责任的。本书后面还会有一章专门讲解unsafe关键字。

第16章　解引用

“解引用”（Deref）是“取引用”（Ref）的反操作。取引用，我们有&、&mut等操作符，对应的，解引用，我们有*操作符，跟C语言是一样的。示例如下：

```
fn main() {
    let v1 = 1;
    let p = &v1; //取引用操作
    let v2 = *p;  //解引用操作
    println!("{} {}", v1, v2);
}
```

比如说，我们有引用类型p：&i32；，那么可以用*符号执行解引用操作。上例中，v1的类型是i32，p的类型是&i32，*p的类型又返回i32。

16.1 自定义解引用

解引用操作可以被自定义。方法是，实现标准库中的std：：ops：：Deref或者std：：ops：：DerefMut这两个trait。

Deref的定义如下所示。DerefMut的唯一区别是返回的是&mut型引用都是类似的，因此不过多介绍了。

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

这个trait有一个关联类型Target，代表解引用之后的目标类型。

比如，标准库中实现了String向str的解引用转换：

```
impl ops::Deref for String {
    type Target = str;

    #[inline]
    fn deref(&self) -> &str {
        unsafe { str::from_utf8_unchecked(&self.vec) }
    }
}
```

请大家注意这里的类型，deref（）方法返回的类型是&Target，而不是Target。

如果说有变量s的类型为String，*s的类型并不等于s.deref（）的类型。

*s的类型实际上是Target，即str。&*s的类型才是&str。

s.deref（）的类型为&Target，即&str。它们的关系见表16-1。

表　16-1

表达式	类型	表达式	类型
s	String	s.deref()	&str
&s	&String	*s	str
Deref::Target	str	&*s	&str
Deref::deref()	&str		

以上关系有点绕。关键是要理解，*expr的类型是Target，而deref（）方法返回的类型却是&Target。

标准库中有许多我们常见的类型实现了这个Deref操作符。比如Vec<T>、String、Box<T>、Rc<T>、Arc<T>等。它们都支持“解引用”操作。从某种意义上来说，它们都可以算做特种形式的“指针”（像胖指针一样，是带有额外元数据的指针，只是元数据不限制在usize范围内了）。我们可以把这些类型都称为“智能指针”。

比如我们可以这样理解这几个类型：

·Box<T>是“指针”，指向一个在堆上分配的对象；

·Vec<T>是“指针”，指向一组同类型的顺序排列的堆上分配的对象，且携带有当前缓存空间总大小和元素个数大小的元数据；

·String是“指针”，指向的是一个堆上分配的字节数组，其中保存的内容是合法的utf8字符序列。且携带有当前缓存空间总大小和字符串实际长度的元数据。

以上几个类型都对所指向的内容拥有所有权，管理着它们所指向的内存空间的分配和释放。

·Rc<T>和Arc<T>也是某种形式的、携带了额外元数据的“指针”，它们提供的是一种“共享”的所有权，当所有的引用计数指针都销毁之后，它们所指向的内存空间才会被释放。

自定义解引用操作符可以让用户自行定义各种各样的“智能指针”，完成各种各样的任务。再配合上编译器的“自动”解引用机制，非常有用。下面我们讲解什么是“自动解引用”。

16.2 自动解引用

Rust提供的“自动解引用”机制，是在某些场景下“隐式地”“自动地”帮我们做了一些事情。什么是自动解引用呢？下面用一个示例来说明：

```
fn main() {
    let s = "hello";
    println!("length: {}", s.len());
    println!("length: {}", (&s).len());
    println!("length: {}", (&&&&&&&&&&&&&s).len());
}
```

编译，成功。查文档我们可以知道，len（）这个方法的签名是：

```
fn len(&self) -> usize
```

它接受的receiver参数是&str，因此我们可以用UFCS语法调用：

```
println!("length: {}", str::len(&s));
```

但是，如果我们使用&&&&&&&&&&&str类型来调用成员方法，也是可以的。原因就是，Rust编译器帮我们做了隐式的deref调用，当它找不到这个成员方法的时候，会自动尝试使用deref方法后再找该方法，一直循环下去。

编译器在&&&str类型里面找不到len方法；尝试将它deref，变成&&str类型后再寻找len方法，还是没找到；继续deref，变成&str，现在找到len方法了，于是就调用这个方法。

自动deref的规则是，如果类型T可以解引用为U，即T：Deref<U>，则&T可以转为&U。

16.3 自动解引用的用处

用Rc这个“智能指针”举例。Rc实现了Deref：

```
impl<T: ?Sized> Deref for Rc<T> {
    type Target = T;

    #[inline(always)]
    fn deref(&self) -> &T {
        &self.inner().value
    }
}
```

它的Target类型是它的泛型参数T。这么设计有什么好处呢？我们看下面的用法：

```
use std::rc::Rc;

fn main() {
    let s = Rc::new(String::from("hello"));
    println!("{:?}", s.bytes());
}
```

我们创建了一个指向String类型的Rc指针，并调用了bytes（）方法。这里是不是有点奇怪？

这里的机制是这样的：Rc类型本身并没有bytes（）方法，所以编译器会尝试自动deref，试试s.deref（）.bytes（）。

String类型其实也没有bytes（）方法，但是String可以继续deref，于是再试试s.deref（）.deref（）.bytes（）。

这次在str类型中找到了bytes（）方法，于是编译通过。

我们实际上通过Rc类型的变量调用了str类型的方法，让这个智能指针透明。这就是自动Deref的意义。

实际上以下写法在编译器看起来是一样的：

```
use std::rc::Rc;
use std::ops::Deref;

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

    println!("length: {}", s.len());
    println!("length: {}", s.deref().len());
    println!("length: {}", s.deref().deref().len());

    println!("length: {}", (*s).len());
    println!("length: {}", (&*s).len());
    println!("length: {}", (&**s).len());
}
```

这就是为什么String需要实现Deref trait，是为了让&String类型的变量可以在必要的时候自动转换为&str类型。所以String类型的变量可以直接调用str类型的方法。比如：

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

虽然s的类型是String，但它在调用bytes（）方法的时候，编译器会自动查找并转换为s.deref（）.bytes（）调用。所以String类型的变量就可以直接调用str类型的方法了。

同理：Vec<T>类型也实现了Deref trait，目标类型是[T]，&Vec<T>类型的变量就可以在必要的时候自动转换为&[T]数组切片类型；Rc<T>类型也实现了Deref trait，目标类型是T，Rc<T>类型的变量就可以直接调用T类型的方法。

注意：&*两个操作符连写跟分开写是不同的含义。以下两种写法是不同的：

```
fn joint() {
    let s = Box::new(String::new());
    let p = &*s;
    println!("{} {}", p, s);
}

fn separate() {
    let s = Box::new(String::new());
    let tmp = *s;
    let p = &tmp;
    println!("{} {}", p, s);
}
```

```
fn main() {
    joint();
    separate();
}
```

---

fn joint（）是可以直接编译通过的，而fn separate（）是不能编译通过的。因为编译器很聪明，它看到&*这两个操作连在一起的时候，会直接把&*s表达式理解为s.deref（），这时候p只是s的一个借用而已。而如果把这两个操作分开写，会先执行*s把内部的数据move出来，再对这个临时变量取引用，这时候s已经被移走了，生命周期已经结束。

同样的，let p=&{*s}；这种写法也编译不过。这个花括号的存在创建了一个临时的代码块，在这个临时代码块内部先执行解引用，同样是move语义。

从这里我们也可以看到，默认的“取引用”、“解引用”操作是互补抵消的关系，互为逆运算。但是，在Rust中，只允许自定义“解引用”，不允许自定义“取引用”。如果类型有自定义“解引用”，那么对它执行“解引用”和“取引用”就不再是互补抵消的结果了。先&后*以及先*后&的结果是不同的。

16.4 有时候需要手动处理

如果智能指针中的方法与它内部成员的方法冲突了怎么办呢？编译器会优先调用当前最匹配的类型，而不会执行自动deref，在这种情况下，我们就只能手动deref来表达我们的需求了。

比如说，Rc类型和String类型都有clone方法，但是它们执行的任务不同。Rc：：clone（）做的是把引用计数指针复制一份，把引用计数加1。String：：clone（）做的是把字符串深复制一份。示例如下：

```
use std::rc::Rc;
use std::ops::Deref;
fn type_of(_: ()) { }

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

    let s1 = s.clone();        // (1)
    //type_of(s1);
    let ps1 = (*s).clone();    // (2)
    //type_of(ps1);
    let pps1 = (**s).clone();  // (3)
    //type_of(pps1);
}
```

在以上的代码中，位置（1）处s1的类型为Rc<Rc<String>>，位置（2）处ps1的类型为Rc<String>，位置（3）处pps1的类型为String。

一般情况下，在函数调用的时候，编译器会帮我们尝试自动解引用。但在某些情况下，编译器不会为我们自动插入自动解引用的代码。以String和&str类型为例，在match表达式中：

```
fn main() {
    let s = String::new();
    match &s {
        "" => {}
        _ => {}
    }
}
```

这段代码编译会发生错误，错误信息为：

```
mismatched types:
    expected `&collections::string::String`,
        found `&'static str`
```

match后面的变量类型是&String，匹配分支的变量类型为&'static str，这种情况下就需要我们手动完成类型转换了。手动将&String类型转换为&str类型的办法如下。

1）match s.deref（）。这个方法通过主动调用deref（）方法达到类型转换的目的。此时我们需要引入Deref trait方可通过编译，即加上代码use std：：ops：：Deref；。

2）match &*s。我们可以通过*s运算符，也可以强制调用deref（）方法，与上面的做法一样。

3）match s.as_ref（）。这个方法调用的是标准库中的std：：convert：：AsRef方法，这个trait存在于prelude中，无须手工引入即可使用。

4）match s.borrow（）。这个方法调用的是标准库中的std：：borrow：：Borrow方法。要使用它，需要加上代码use std：：borrow：：Borrow；。

5）match &s[..]。这个方案也是可以的，这里利用了String重载的Index操作。

16.5 智能指针

Rust语言提供了所有权、默认move语义、借用、生命周期、内部可变性等基础概念。但这些并不是Rust全部的内存管理方式，在这些概念的基础上，我们还能继续抽象、封装更多的内存管理方式，而且保证内存安全。

16.5.1 引用计数

到目前为止，我们接触到的示例中都是一块内存总是只有唯一的一个所有者。当这个变量绑定自身消亡的时候，这块内存就会被释放。引用计数智能指针给我们提供了另外一种选择：一块不可变内存可以有多个所有者，当所有的所有者消亡后，这块内存才会被释放。

Rust中提供的引用计数指针有std：：rc：：Rc<T>类型和std：：sync：：Arc<T>类型。Rc类型和Arc类型的主要区别是：Rc类型的引用计数是普通整数操作，只能用在单线程中；Arc类型的引用计数是原子操作，可以用在多线程中。这一点是通过编译器静态检查保证的。Arc类型的讲解可以参见第四部分相关章节，本章主要关注Rc类型。

首先我们用示例展示Rc智能指针的用法：

```
use std::rc::Rc;

struct SharedValue {
    value : i32
}

fn main() {
    let shared_value : Rc<SharedValue> = Rc::new(SharedValue { value : 42 });

    let owner1 = shared_value.clone();
    let owner2 = shared_value.clone();

    println!("value : {} {}", owner1.value, owner2.value);
    println!("address : {:p} {:p}", &owner1.value, &owner2.value);
}
```

编译运行，结果显示：

```
$ ./test
value : 42 42
address : 0x13958abdf20 0x13958abdf20
```

这说明，owner1 owner2里面包含的数据不仅值是相同的，而且地址也是相同的。这正是Rc的意义所在。

从示例中可以看到，Rc指针的创建是调用Rc：：new静态函数，与Box类型一致（将来会允许使用box关键字创建）。如果要创建指向同样内存区域的多个Rc指针，需要显式调用clone函数。请注意，Rc指针是没有实现Copy trait的。如果使用直接赋值方式，会执行move语义，导致前一个指针失效，后一个指针开始起作用，而且引用计数值不变。如果需要创造新的Rc指针，必须手工调用clone（）函数，此时引用计数值才会加1。当某个Rc指针失效，会导致引用计数值减1。当引用计数值减到0的时候，共享内存空间才会被释放。

这没有违反我们前面讲的“内存安全”原则，它内部包含的数据是“不可变的”，每个Rc指针对它指向的内部数据只有读功能，和共享引用&一致，因此，它是安全的。区别在于，共享引用对数据完全没有所有权，不负责内存的释放，Rc指针会在引用计数值减到0的时候释放内存。Rust里面的Rc<T>类型类似于C++里面的shared_ptr<const T>类型，且强制不可为空。

从示例中我们还可以看到，使用Rc访问被包含的内部成员时，可以直接使用小数点语法来进行，与T &T Box<T>类型的使用方法一样。原因我们在前面已经讲过了，这是因为编译器帮我们做了自动解引用。我们查一下Rc的源码就可以知道：

```
impl<T: ?Sized> Deref for Rc<T> {
    type Target = T;

    #[inline(always)]
    fn deref(&self) -> &T {
        &self.inner().value
    }
}
```

可见，Rc类型重载了“解引用”运算符，而且恰好Target类型指定的是T。这就意味着编译器可以将Rc<T>类型在必要的时候自动转换为&T

类型，于是它就可以访问T的成员变量，调用T的成员方法了。因此，它可以被归类为“智能指针”。

下面我们继续分析Rc类型的实现原理。它的源代码在src/liballoc/rc.rs中，Rc类型的定义如下所示：

```
pub struct Rc<T: ?Sized> {
    _ptr: Shared<RcBox<T>>,
}
```

其中RcBox是这样定义的：

```
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}
```

其中Shared类型我们暂时可以不用管它，当它是一个普通指针就好。目前它还没有稳定，后续可能设计上还会有变化，因此本书就不对它深究了。

同时，它实现了Clone和Drop这两个trait。在clone方法中，它没有对它内部的数据实行深复制，而是将强引用计数值加1，如下所示：

```
impl<T: ?Sized> Clone for Rc<T> {

    #[inline]
    fn clone(&self) -> Rc<T> {
        self.inc_strong();
        Rc { ptr: self.ptr }
    }
}

fn inc_strong(&self) {
        self.inner().strong.set(self.strong().checked_add(1)
.unwrap_or_else(|| unsafe { abort() }));
}
```

在drop方法中，也没有直接把内部数据释放掉，而是将强引用计数值减1，当强引用计数值减到0的时候，才会析构掉共享的那块数据。当弱引用计数值也减为0的时候，才说明没有任何Rc/Weak指针指向这块

内存，它占用的内存才会被彻底释放。如下所示：

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            let ptr = self.ptr.as_ptr();

            self.dec_strong();
            if self.strong() == 0 {
                // destroy the contained object
                ptr::drop_in_place(self.ptr.as_mut());

                // remove the implicit "strong weak" pointer now that we've
                // destroyed the contents.
                self.dec_weak();

                if self.weak() == 0 {
                    Heap.dealloc(ptr as *mut u8, Layout::for_value(&*ptr));
                }
            }
        }
    }
}
```

从上面代码中我们可以看到，Rc智能指针所指向的数据，内部包含了强引用和弱引用的计数值。这两个计数值都是用Cell包起来的。为什么这两个数字一定要用Cell包起来呢？

我们假设，如果不用Cell<usize>，而是直接用usize的话，在执行clone方法时会出现什么情况。

```
fn clone(&self) -> Rc<T> {}
```

大家需要注意的是，这个self的类型是&Self，不是&mut Self。但我们同时还需要使用这个共享引用self来修改引用计数的值。

所以这个成员必须是具有内部可变性的。反之，如果它们是普通的整数，那么我们就要求使用&mut Self类型来调用clone方法，然而一般情况下，我们都会需要多个Rc指针指向同一块内存区域，引用计数值是共享的。如果存在多个&mut型指针指向引用计数值的话，则违反了Rust内存安全的规则。

因此，Rc智能指针的实现，必须使用“内部可变性”功能。Cell类型提供了一种类似C++的mutable关键字的能力，使我们可以通过不可变指

针修改复合数据类型内部的某一个成员变量。

所以，我们可以总结出最适合使用“内部可变性”的场景是：当逻辑上不可变的方法的实现细节又要求某部分成员变量具有可变性的时候，我们可以使用“内部可变性”。Rc内部的引用计数变量就是绝佳的例子。

多个Rc指针指向的共享内存区域如果需要修改的话，也必须用内部可变性。如在下面的例子中，如果我们需要多个Rc指针指向一个Vec，而且具备修改权限的话，那我们必须用RefCell把Vec包起来：

```
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let shared_vec: Rc<RefCell<Vec<isize>>> = Rc::new(RefCell::new(vec![1, 2, 3]));
    let shared1 = shared_vec.clone();
    let shared2 = shared1.clone();

    shared1.borrow_mut().push(4);
    println!("{:?}", shared_vec.borrow());

    shared2.borrow_mut().push(5);
    println!("{:?}", shared_vec.borrow());
}
```

16.5.2 Cow

在C++语境中，Cow代表的是Copy-On-Write，即“写时复制技术”。它是一种高效的资源管理手段。假设我们有一份比较昂贵的资源，当我们需要复制的时候，我们可以采用“浅复制”的方式，而不需要重新克隆一份新的资源。而如果要修改复制之后的值，这时候再执行深复制，在此基础上修改。因此，它的优点是把克隆这个操作推迟到真正需要“复制并写操作”的时候发生。

在Rust语境中，因为Copy和Clone有比较明确的语义区分，一般把Cow解释为Clone-On-Write。它对指向的数据可能“拥有所有权”，或者可能“不拥有所有权”。

当它只需要对所指向的数据进行只读访问的时候，它就只是一个借用指针；当它需要写数据功能时，它会先分配内存，执行复制操作，再对自己拥有所有权的内存进行写入操作。Cow在标准库中是一个enum：

```
pub enum Cow<'a, B: ?Sized + 'a> where B: ToOwned {
    /// Borrowed data.
    Borrowed(&'a B),

    /// Owned data.
    Owned(<B as ToOwned>::Owned)
}
```

它可以是Borrowed或者Owned两种状态。如果是Borrowed状态，可以通过调用to_mut函数获取所有权。在这个过程中，它实际上会分配一块新的内存，并将原来Borrowed状态的数据通过调用to_owned（）方法构造出一个新的拥有所有权的对象，然后对这块拥有所有权的内存执行操作。

Cow类型最常见的是跟字符串配合使用：

```
use std::borrow::Cow;

fn remove_spaces<'a>(input: &'a str) -> Cow<'a, str> {
    if input.contains(' ') {
        let mut buf = String::with_capacity(input.len());

        for c in input.chars() {
            if c != ' ' {
                buf.push(c);
            }
        }

        return Cow::Owned(buf);
    }

    return Cow::Borrowed(input);
}

fn main() {
    let s1 = "no_spaces_in_string";
    let result1 = remove_spaces(s1);

    let s2 = "spaces in string";
    let result2 = remove_spaces(s2);

    println!("{}\n{}", result1, result2);
}
```

在这个示例中，我们使用Cow类型最主要的目的是优化执行效率。remove_spaces函数的输入参数是&str类型。如果输入的参数本来就不包含空格，那么我们最好是直接返回参数本身，无须分配新的内存；如果输入参数包含空格，我们就只能在函数体内部创建一个新的String对

象，用于存储去除掉空格的结果，然后再返回去。

这样一来，就产生了一个小矛盾，这个函数的返回值类型用&str类型和String类型都不大合适。

·如果返回类型指定为&str类型，那么需要新分配内存的时候，会出现生命周期编译错误。因为函数内部新分配的字符串的引用不能在函数调用结束后继续存在。

·如果返回类型指定为String类型，那么对于那种不需要对输入参数做修改的情况，有一些性能损失。因为输入参数&str类型转为String类型需要分配新的内存空间并执行复制，性能开销较大。

这种时候使用Cow类型就是不二之选。既能满足编译器的生命周期要求，也避免了无谓的数据复制。Cow类型，就是优秀的“零性能损失抽象”的设计范例。

C++implementations obey the zero-overhead principle：What you don’t use，you don’t pay for.And further：What you do use，you couldn’t hand code any better.

——Stroustrup

由于Rust中有这套所有权、生命周期的基础，在Rust中使用Cow这种类型是完全没有风险的，任何可能的内存安全问题，编译器都可以帮我们查出来。所以，有些时候，自由和不自由是可以相互转化的，语法方面的不自由，反而可能造就抽象水平的更自由。

Cow类型还实现了Deref trait，所以当我们需要调用类型T的成员函数的时候，可以直接调用，完全无须考虑后面具体是“借用指针”还是“拥有所有权的指针”。所以我们也可以把它当成是一种“智能指针”。

16.6 小结

Rust中允许一部分运算符可以由用户自定义行为，即“操作符重载”。其中“解引用”是一个非常重要的操作符，它允许重载。

而需要提醒大家注意的是，“取引用”操作符，如&、&mut，是不允许重载的。因此，“取引用”和“解引用”并非对称互补关系。*&T的类型一定是T，而&*T的类型未必就是T。

更重要的是，读者需要理解，在某些情况下，编译器帮我们插入了自动deref的调用，简化代码。

在Deref的基础上，我们可以封装出一种自定义类型，它可以直接调用其内部的其他类型的成员方法，我们可以把这种类型称为智能指针类型。

第17章　泄漏

熟悉C++的朋友应该知道，在C++中，如果引用计数智能指针出现了循环引用，就会导致内存泄漏。而Rust中也一样存在引用计数智能指针Rc，那么Rust中是否可能制造出内存泄漏呢？

下面我们来通过一步步的尝试，看看如何才能构造一个内存泄漏的例子。

17.1 内存泄漏

首先，我们设计一个Node类型，它里面包含一个指针，可以指向其他的Node实例：

```
struct Node {
    next : Box<Node>
}
```

接下来我们尝试一下创建两个实例，将它们首尾相连：

```
fn main() {
    let node1 = Node { next : Box::new(...) }
}
```

到这里写不下去了，Rust中要求，Box指针必须被合理初始化，而初始化Box的时候又必须先传入一个Node实例，这个Node的实例又要求创建一个Box指针。这成了“鸡生蛋蛋生鸡”的无限循环。

要打破这个循环，我们需要使用“可空的指针”。在初始化Node的时候，指针应该是“空”状态，后面再把它们连接起来。我们把代码改进，为了能修改node的值，还需要使用mut：

```
struct Node {
    next : Option<Box<Node>>
}

fn main() {
    let mut node1 = Box::new (Node { next : None });
    let mut node2 = Box::new (Node { next : None });

    node1.next = Some(node2);
    node2.next = Some(node1);
}
```

编译，发生错误：“error：use of moved value：`node2`”。

从编译信息中可以看到，在node1.next=Some（node2）；这条语句中发生了move语义，从此句往后，node2变量的生命周期已经结束了。

因此后面一句中使用node2的时候发生了错误。那我们需要继续改进，不使用node2，换而使用node1.next，代码改成下面这样：

```
fn main() {
    let mut node1 = Box::new (Node { next : None });
    let mut node2 = Box::new (Node { next : None });

    node1.next = Some(node2);
    match node1.next {
        Some(mut n) => n.next = Some(node1),
        None => {}
    }
}
```

编译又发生了错误，错误信息为：“error：use of partially moved value：`node1`”。

这是因为在match语句中，我们把node1.next的所有权转移到了局部变量n中，这个n实际上就是node2的实例，在执行赋值操作n.next=Some（node1）的过程中，编译器认为此时node1的一部分已经被转移出去了，它不能再被用于赋值号的右边。

看来，这是因为我们选择使用的指针类型不对，Box类型的指针对所管理的内存拥有所有权，只使用Box指针没有办法构造一个循环引用的结构出来。于是，我们想到使用Rc指针。同时，我们还用了Drop trait来验证这个对象是否真正被释放了：

```
use std::rc::Rc;

struct Node {
    next : Option<Rc<Node>>
}

impl Drop for Node {
    fn drop(&mut self) {
        println!("drop");
    }
}

fn main() {
    let mut node1 = Node { next : None };
    let mut node2 = Node { next : None };
    let mut node3 = Node { next : None };

    node1.next = Some(Rc::new(node2));
    node2.next = Some(Rc::new(node3));
    node3.next = Some(Rc::new(node1));
}
```

编译依然没有通过，错误信息为：“error：partial reinitialization of uninitialized structure`node2`”，还是没有达到目的。继续改进，我们将原来“栈”上分配内存改为在“堆”上分配内存：

```
use std::rc::Rc;

struct Node {
    next : Option<Rc<Node>>
}

impl Drop for Node {
    fn drop(&mut self) {
        println!("drop");
    }
}

fn main() {
    let mut node1 = Rc::new(Node { next : None });
    let mut node2 = Rc::new(Node { next : None });
    let mut node3 = Rc::new(Node { next : None });

    node1.next = Some(node2);
    node2.next = Some(node3);
    node3.next = Some(node1);
}
```

编译再次不通过，错误信息为：“error：cannot assign to immutable field”。通过这个错误信息，我们现在应该能想到，Rc类型包含的数据是不可变的，通过Rc指针访问内部数据并做修改是不可行的，必须用RefCell把它们包裹起来才可以。继续修改：

```
use std::rc::Rc;
use std::cell::RefCell;

struct Node {
    next : Option<Rc<RefCell<Node>>>
}

impl Node {
    fn new() -> Node {
        Node { next : None}
    }
}
impl Drop for Node {
    fn drop(&mut self) {
        println!("drop");
    }
}

fn alloc_objects() {
```

```
    let node1 = Rc::new(RefCell::new(Node::new()));
    let node2 = Rc::new(RefCell::new(Node::new()));
    let node3 = Rc::new(RefCell::new(Node::new()));

    node1.borrow_mut().next = Some(node2.clone());
    node2.borrow_mut().next = Some(node3.clone());
    node3.borrow_mut().next = Some(node1.clone());
}

fn main() {
    alloc_objects();
    println!("program finished.");
}
```

因为我们使用了RefCell，对Node内部数据的修改不再需要mut关键字。编译通过，执行，这一次屏幕上没有打印任何输出，说明了析构函数确实没有被调用。

至此，终于实现了使用Rc指针构造循环引用，制造了内存泄漏。

本节花费这么多笔墨一步步地向大家演示如何构造内存泄漏，主要是为了说明，虽然构造循环引用非常复杂，但是可能性还是存在的，Rust无法从根本上避免内存泄漏 。通过循环引用构造内存泄漏，需要同时满足三个条件：1）使用引用计数指针；2）存在内部可变性；

3）指针所指向的内容本身不是'static的。

当然，这个示例也说明，通过构造循环引用来制造内存泄漏是比较复杂的，不是轻而易举就能做到的。构造循环引用的复杂性可能也刚好符合我们的期望，毕竟从设计原则上来说：鼓励使用的功能应该设计得越易用越好；不鼓励使用的功能，应该设计得越难用越好。

Easy to Use，Easy to Abuse.

对于上面这个例子，要想避免内存泄漏，需要程序员手动把内部某个地方的Rc指针替换为std：：rc：：Weak弱引用来打破循环。这是编译器无法帮我们静态检查出来的。

17.2 内存泄漏属于内存安全

在编程语言设计这个层面，“内存泄漏”是一个基本上无法在编译阶段彻底解决的问题。在许多场景下，什么是“内存泄漏”、什么不是“内存泄漏”，本身就没有一个完全客观的评判标准。它实质上跟程序员的“意图”有关。程序很难自动判定出哪些变量是以后还会继续用的，哪些是不再被使用的。

即便是在使用GC做内存管理的环境下，程序员也有可能不小心将不应该被使用的变量错误引用，造成无法自动回收的问题。因为GC判定一个对象是否可回收的标准是，这个对象有没有被“根”对象直接或者间接引用。假如一个对象本来是应该被释放的，可是因为逻辑问题，没有把指向它的有效引用全部释放，那么GC依旧把它判定为不可回收。我们可能在不经意的情况下，造成了不再需要继续使用的对象被生命周期更长的对象所引用。面对这样的情况，GC也会显得无能为力。比如，在android编程领域，我们可能会在注册回调函数的时候把一个较大的activity引用传递过去，结果activity应该被销毁的时候，由于还有其他变量继续持有指向它的引用，从而导致该activity变量无法正常被释放，这种现象被称为“Context泄漏”。解决这个问题的办法只能是，在必要的地方使用“弱引用”（Weak Reference），避免“强引用”对变量生命周期的影响。解决引用计数的循环引用的办法与此类似，也是一样用“弱引用”来打破循环，避免“强引用”对生命周期的影响。再比如在javascript中注册一个定时器，而定时器不小心引用了许多大对象，这些对象会随着闭包加入到主事件循环队列中，也会造成类似的结果。在绝大部分情况下，GC给我们带来了方便。但是，程序员也千万不能因为有GC的辅助，而忽略对变量的生命周期的设计考量。

在C++和Rust中是一样的，如果出现了循环引用，那么只能通过手动打破循环的方式来解决内存泄漏的问题。编译器无法通过静态检查来保证你不会犯这个错误。

内存泄漏显然是一种bug。但它跟“内存不安全”这种bug的性质不一样。“内存泄漏”是对“正常数据”的“应该执行但是没有执行”的操作，“内存不安全”是对“不正常数据”的“不应该执行但是执行了”的操作。从后果上说，“内存不安全”导致的后果比“内存泄漏”要严重得多，如表17-1所示。

表　17-1

	正常数据	不正常数据
执行了	√	内存不安全
没执行	内存泄漏	√

语言的设计者当然是希望能彻底解决内存泄漏的问题。但是很可惜，这个问题恐怕不是在语言层面能彻底解决的问题。所谓“彻底解决”的意思是，用户无论使用何种技巧，永远无法构造出内存泄漏的情况 。Rust语言无法给出这样的保证。笔者也不认为GC“彻底解决”了内存泄漏的问题。内存泄漏当然是不好的事情，作为开发者，我们应该尽可能避免内存泄漏现象的发生。然而，需要强调的是，内存泄漏不属于内存安全的范畴，Rust也不会在语言设计层面给出一个“免除内存泄漏”的承诺。

To put it another way，Rust gives you a lot of safety guarantees，but it doesn’t protect you from memory leaks（or deadlocks，which turns out to be a very similar problem）.

17.3 析构函数泄漏

上面的例子展现了如何在Rust中不使用unsafe代码制造内存泄漏。在Rust中，在不经意间不小心制造内存泄漏的可能性是很低的。但是这个可能性还是存在的。

然而，内存泄漏并非最可怕的情况，因为内存泄漏只造成资源浪费，毕竟没有造成野指针等更为严重的内存安全问题。上面的例子实际上还暗示了另外一种危险性，即析构函数泄漏。在Rust中，RAII手法用得非常普遍，它实际上要求程序的正确性依赖于析构函数的确定性调用。然而让我们担心的事情是，析构函数是有可能永远不会被调用的。

除了前面展示的通过循环引用导致的析构函数泄漏之外，还有许多种方式可以产生同样的效果。比如，我们构造两个首尾相连的channel，发送端和接收端连到一起，那么在这两个channel里面传递的对象就进入了死循环，就永远不会被析构了。

析构函数泄漏是比内存泄漏更严重的情况。因为析构函数是可以“自定义”的，析构函数里面可能调用“任意的”代码。

我们一直在强调，Rust给了我们一个非常强的保证，即“内存安全”。这个保证是非常严肃认真的。这个保证意味着，只要不使用unsafe，用户永远无法构造出“内存不安全”的情况。然而，对于泄漏问题，Rust做不到像内存安全这种程度的保证。所以，Rust设计者不得不痛苦地承认，析构函数并不能被保证调用。大家不要误解了这段话，这并不是意味着Rust会轻轻松松、时时刻刻造成泄漏，它只是意味着，编译器没办法自动检查出所有可能的资源泄漏问题，并给出编译错误或警告。

承认析构函数可能不会被调用（即便在不使用unsafe代码情况下），并不会造成特别严重的问题——除非它违反了“内存安全”。“内存安全”一直是Rust坚持的原则和底线，这条原则是永远不能被破坏的，否则Rust就失去了存在的意义。这个结论直接导致了下面几个比较重要的后果。

其一，标准库中的std：：mem：：forget函数去掉了unsafe标记。

其二，允许带有析构函数的类型，作为static变量和const变量。全局变量的析构函数最后是泄漏掉了的，不会被调用。以前曾经规定带析构函数的类型不允许作为全局变量，后来放宽了规定，允许作为全局变量，但是析构函数无法调用。

其三，标准库中不安全代码需要依赖析构函数调用的逻辑得到修改，其中涉及Vec：：drain_range和Thread：：scoped等方法。

Rust标准库中有一个std：：mem：：forget函数，这个方法的签名是fn forget<T>（t：T）。它接受的参数不是引用类型，而是将参数move进入函数体中，类似于std：：mem：：drop。但它与drop最大的区别是，它会阻止编译器调用这个变量的析构函数，也不会释放它在堆上申请的内存。它的作用就是制造泄漏。原来这个函数是unsafe的，但是，当设计者发现完全可以用安全代码写一个同样效果的forget函数，那么，它的unsafe标记也就没有什么意义了。因此，大家决定，去掉forget函数前面的unsafe标记。这个函数不再被标记为unsafe，只是因为设计者意识到了泄漏并非内存安全问题，unsafe关键字只能用于标记跟内存安全相关的问题，并非意味着鼓励用户随意使用这个函数。那么它有什么用呢？我们可以举几个例子：

·我们有一个未初始化的值，我们不希望它执行析构函数；

·在用FFI跟外部函数打交道的时候。

即便析构函数泄漏，也不应造成内存不安全。这个结论，直接导致了thread：：scoped函数从标准库中移除。scoped函数是这样设计的：scoped函数可以创建一个线程，跟spawn函数不一样，它保证在当前函数退出以前，这个线程必定已经退出。这样一来，我们就可以直接使用引用&来读父线程读局部变量，或者用&mut来写局部变量，避免了Arc的运行效率损失，是非常有用的scoped函数与spawn函数的区别就在于，它保证子线程一定会在当前函数退出之前退出，所以它的生命周期比当前函数的生命周期短。

```
// 以下示例目前无法编译通过,scoped已经被移除
use std::thread;

fn main() {
    let mut vec = vec![0, 1, 2, 3, 4, 5, 6, 7];
    {
        let mut guards = Vec::new();
```

```
            for x in &mut vec {
                let guard = thread::scoped(move || {
                        *x += 1;
                    });
                guards.push(guard);
            }
            // guards 析构,在析构函数中等待子线程被销毁
        }
        // 子线程已经全部退出
        println!("{:?}", vec);
    }
```

这个scoped函数的实现原理是，它返回一个JoinGuard类型，在这个类型的析构函数中阻塞当前线程，等待子线程结束。所以，函数退出之前，子线程必定已经被销毁。子线程中用到的指向当前函数栈的指针，也不会成为野指针。

粗看起来以上这个设计是不错的，而且它也的确在早期版本的Rust标准库中存在了一段时间。然而可惜的是，这个设计是有bug的，它会造成安全代码中的“内存不安全”现象。问题在哪里呢？问题在于“析构函数泄漏”。我们知道，Rust无法保证“析构函数必定被调用”。如果有一个用户，想办法将这个JoinGuard传递到当前函数外面去了，或者用循环引用使得这个类型的析构函数永不调用，就出现了析构泄漏。如果这个类型出现了析构泄漏，那么会导致这个子线程的生命周期并不会限制在父线程的当前函数执行周期之内，父线程的当前函数退出，子线程却并未结束，父线程的函数调用栈已经发生变化，而子线程依然有能力访问以前指向的那块内存。这是一个典型的内存安全问题。

Rust对“内存安全”是不允许的。虽然上面叙述的情况在正式代码中出现的几率很小，但是这依旧是一个潜在的问题。Rust对库代码的质量标准是：不论使用何种奇巧，只要用户有可能在不使用unsafe构造出内存安全，那这个库就是不安全的、不可接受的。因此，scoped函数必须从标准库中去掉，它是不能被接受的。它违反了Rust的安全承诺，将安全性建立在“析构函数必定被调用”的假设之上，而这个假设是不成立的。它有可能导致不使用unsafe的情况下，也能制造出“内存不安全”。这是一个设计失败的API。

那么用什么办法来解决这个问题呢？可以通过改变API的风格来实现。如果说RAII式的风格是外向式的，那么对应的“回调函数”式的风格就是内向式的。什么是外向式的和内向式的API风格？我们拿迭代器来打比方。Rust的迭代器是典型的外向式的风格，它暴露了next（）方

法，由使用者决定何时、何处、如何调用。我们还可以设计内向式的迭代风格，它的写法是for_each（||{this is a closure}）。在这种方式下，使用者只能传递一个闭包进去，而无权管理迭代器的调用。内向式的API对使用者来说灵活性就比较差，比如，我们没办法实现针对两个容器的两个迭代器，分别轮流调用next（）方法，或者在迭代过程中提前中止等。

相比之下，RAII式的API暴露给使用者的灵活性更强，而回调函数式的API对使用者的约束性更强。我们如果把scoped函数变一种风格，它就可以变成安全的了。这个尝试，在第三方库中已经实现，如果大家需要这个功能，可以搜索crossbeam或者scoped_threadpool这两个库。我们来看看scoped_threadpool是如何使用的：

```
extern crate scoped_threadpool;
use scoped_threadpool::Pool;

fn main() {
    let mut pool = Pool::new(4);

    let mut vec = vec![0, 1, 2, 3, 4, 5, 6, 7];

    pool.scoped(|scope| {
        for e in &mut vec {
            scope.execute(move || {
                *e += 1;
            });
        }
    });

    println!("{:?}", vec);
}
```

关于如何利用cargo工具引用外部库，在本章就不详细解释了。在这里我们只关心代码逻辑。线程内部直接使用了&mut vec形式访问了父线程“栈”上的变量。这个scoped函数的使用方式跟前面介绍的版本相比更复杂；然而，它的优点是安全性并不依赖外部使用者确保“析构函数”的调用。因为这个改变，使得“等待线程结束”这个逻辑从库的使用者那边移动到了库的编写者那边。库的编写者当然可以保证这个逻辑必然被调用，如果我们把它暴露出来，交给使用者来调用，就不一定了。所以说，我们能从中学到的一点是：当你写一个库的时候，如果希望能确保某个方法一定会被调用，请保证这段代码在你自己的控制之中，不要只在文档中描述，要求使用者主动去调用。

我们比较一下scoped函数和spawn函数的签名规则：

```
fn scoped<'pool, 'scope, F, R>(&'pool mut self, f: F) -> R
    where F: FnOnce(&Scope<'pool, 'scope>) -> R
    {}
fn spawn<F, T>(f: F) -> JoinHandle<T>
    where F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
    {}
```

我们可以看到，对于闭包参数F类型的约束，spawn有'static生命周期限制，而scoped无需'static生命周期限制。之所以有这样的区别，原因就是在scoped的内部实现中保证了子线程一定会在父线程当前函数退出前结束，这个约束条件不是随便能修改的。在它们的内部都使用了unsafe代码作为实现细节，在它们的外部都使用了合理的约束条件来保证线程安全。所以我们需要再向大家提醒一下safe和unsafe的边界究竟在哪里。哪些是编译器可以保证的，哪些是编译器无法保证的，不是简单就说得清楚的事情。千万不要自以为是地滥用unsafe，如果暴露的外部接口和内部实现不匹配，就会给下游用户“挖坑”，很容易导致某些初学者误以为Rust的内存安全保证是骗人的。

泄漏是一个麻烦的问题，Rust在这个问题上的设计涉及许多的妥协和平衡，其间引发了大量的纠结、讨论甚至争吵。正是：

曾虑多情损梵行，

入山又恐别倾城。

世间安得双全法，

不负如来不负卿。

第18章　Panic

18.1　什么是panic

在Rust中，有一类错误叫作panic。示例如下：

```
fn main() {
    let x : Option<i32> = None;
    x.unwrap();
}
```

编译，没有错误，执行这段程序，输出为：

```
thread '<main>' panicked at 'called `Option::unwrap()` on a `None` value', ../src/li
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

这种情况就引发了一个panic。在这段代码中，我们调用了Option：：unwrap（）方法，正是这个方法有可能导致panic。根据提示，我们设置一个环境变量RUST_BACKTRACE=1之后再执行这个程序，可以看到这个程序在发生panic时候的函数调用栈。

执行RUST_BACKTRACE=1./test，结果为：

```
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', ../src/libc
stack backtrace:
1: 0x10af488f8 - std::sys::backtrace::tracing::imp::write::h6f1d53a70916b90d
2: 0x10af4a3af - std::panicking::default_hook::{{closure}}::h137e876f7d3b5850
3: 0x10af49945 - std::panicking::default_hook::h0ac3811ec7cee78c
4: 0x10af49e96 - std::panicking::rust_panic_with_hook::hc303199e04562edf
5: 0x10af49d34 - std::panicking::begin_panic::h6ed03353807cf54d
6: 0x10af49c52 - std::panicking::begin_panic_fmt::hc321cece241bb2f5
7: 0x10af49bb7 - rust_begin_unwind
8: 0x10af6f0b0 - core::panicking::panic_fmt::h27224b181f9f037f
9: 0x10af6efb4 - core::panicking::panic::h53676c30b3bd95eb
10: 0x10af44804 - <core::option::Option<T>>::unwrap::h3478e42c3c27faa3
11: 0x10af44880 - test::main::h8a7a35fa594c0174
12: 0x10af4a96a - __rust_maybe_catch_panic
13: 0x10af49486 - std::rt::lang_start::h538f8960e7644c80
14: 0x10af448b9 - main
```

我们去查一下Option：：unwrap（）的文档，其中是这么说的：

```
Moves the value v out of the Option<T> if it is Some(v).
Panics
    Panics if the self value equals None.
Safety note
    In general, because this function may panic, its use is discouraged.
    Instead, prefer to use pattern matching and handle the None case explicitly.
```

当Option内部的数据是Some时，它可以成功地将内部的数据move出来返回。

当Option内部的数据是None时，它会发生panic。panic如果没有被处理，它会导致整个程序崩溃。

在Rust中，正常的错误处理应该尽量使用Result类型。Panic则是作为一种“fail fast”机制，处理那种万不得已的情况。

比如，上面例子中的unwrap方法，试图把Option<i32>转换为i32类型，当参数是None的时候，这个转换是没办法做到的，这种时候就只能使用panic。所以，一般情况下，用户应该使用unwrap_or等不会制造panic的方法。

18.2 Panic实现机制

在Rust中，Panic的实现机制有两种方式：unwind和abort。

·unwind方式在发生panic的时候，会一层一层地退出函数调用栈，在此过程中，当前栈内的局部变量还可以正常析构。

·abort方式在发生panic的时候，会直接退出整个程序。

在常见的操作系统上，默认情况下，编译器使用的是unwind方式。所以在发生panic的时候，我们可以通过一层层地调用栈找到发生panic的第一现场，就像前面例子展示的那样。

但是，unwind并不是在所有平台上都能获得良好支持的。在某些嵌入式系统上，unwind根本无法实现，或者占用的资源太多。在这种时候，我们可以选择使用abort方式实现panic。

编译器提供了一个选项，供用户指定panic的实现方式。如下所示：

```
rustc -C panic=unwind test.rs
rustc -C panic=abort test.rs
```

读者可以试试上面两个编译命令，做一下对比。可以看到它们生成的代码，panic时的行为是不一样的，生成的可执行程序大小也不同。

Rust中，通过unwind方式实现的panic，其内部的实现方式基本与C++的异常是一样的。而且，Rust提供了一些工具函数，可以让用户在代码中终止栈展开。示例如下：

```
use std::panic;

fn main() {
    panic::catch_unwind(|| {
        let x : Option<i32> = None;
        x.unwrap();
        println!("interrupted.");
    }).ok();

    println!("continue to execute.");
```

```
}
```

编译执行可见，在unwrap语句后面的这条打印语句并未执行。因为在上一条语句中触发了panic，这个函数调用栈开始销毁。但是我们有一句catch_unwind阻止了函数调用栈的继续展开，就像C++里面的try-catch机制一样。因此，main方法并没有继续被销毁，最后那条语句可以正常打印输出。

如果我们尝试使用“-C panic=abort”选项编译上面的代码，可以看到这个catch_unwind起不了什么作用，最后那条语句无法正常打印输出。

但是，请大家注意，这个catch_unwind机制绝对不是设计用于模拟“try-catch”机制的。请大家永远不要利用这个机制来做正常的流程控制。Rust推荐的错误处理机制是用返回值，第33章讲解Rust的错误处理机制。panic出现的场景一般是：如果继续执行下去就会有极其严重的内存安全问题，这种时候让程序继续执行导致的危害比崩溃更严重，此时panic就是最后的一种错误处理机制。它的主要用处参考下面的情况：

·在FFI场景下的时候，当C语言调用了Rust的函数，在Rust内部出现了panic，如果这个panic在Rust内部没有处理好，直接扔到C代码中去，会导致C语言产生“未定义行为”（undefined behavior）。

·某些高级抽象机制需要阻止栈展开，比如线程池。如果一个线程中出现了panic，我们希望只把这个线程关闭，而不至于将整个线程池“拖下水”。

18.3 Panic Safety

C++中引入了“异常”这个机制之后，同时也带入了一个“异常安全”（exception safety）的概念。对这个概念不熟悉的读者，建议阅读以下文档：

http://www.stroustrup.com/except.pdf

异常安全存在四种层次的保证：

·No-throw——这种层次的安全性保证了所有的异常都在内部正确处理完毕，外部毫无影响；

·Strong exception safety——强异常安全保证可以保证异常发生的时候，所有的状态都可以“回滚”到初始状态，不会导致状态不一致的问题；

·Basic exception safety——基本异常安全保证可以保证异常发生的时候不会导致资源泄漏；

·No exception safety——没有任何异常安全保证。

当我们在系统中使用了“异常”的时候，就一定要想清楚，每个组件应该提供哪种层级的异常安全保证。在Rust中，这个问题同样存在，但是一般叫作panic safety，与“异常”说的是同一件事情。

下面我们来用代码来示例“异常安全”问题会如何影响我们的代码实现。这次，我们用标准库中的一段代码来演示。下面的代码是从src/liballoc/boxed.rs中复制出来的，clone（）方法目的是复制一份新的Box<[T]>：

```
impl<T: Clone> Clone for Box<[T]> {
    fn clone(&self) -> Self {
        let mut new = BoxBuilder {
            data: RawVec::with_capacity(self.len()),
            len: 0,
        };

        let mut target = new.data.ptr();
```

```
            for item in self.iter() {
                unsafe {
                    ptr::write(target, item.clone());
                    target = target.offset(1);
                };

                new.len += 1;
            }

            return unsafe { new.into_box() };

            // Helper type for responding to panics correctly.
            struct BoxBuilder<T> {
                data: RawVec<T>,
                len: usize,
            }

            impl<T> BoxBuilder<T> {
                unsafe fn into_box(self) -> Box<[T]> {
                    let raw = ptr::read(&self.data);
                    mem::forget(self);
                    raw.into_box()
                }
            }

            impl<T> Drop for BoxBuilder<T> {
                fn drop(&mut self) {
                    let mut data = self.data.ptr();
                    let max = unsafe { data.offset(self.len as isize) };

                    while data != max {
                        unsafe {
                            ptr::read(data);
                            data = data.offset(1);
                        }
                    }
                }
            }
        }
    }
```

这段代码的实现稍微有点复杂，但是逻辑还是很清楚的。我们可以先不考虑标准实现，先想想如果我们自己来实现会怎么做。要clone（）一份Box<[T]>，就要新分配一份内存空间，循环把每个元素clone（）一遍即可。有个小技巧是，构造Box<[T]>可以用RawVec：：into_box（）来完成，因此我们需要一个RawVec来做中转。但是标准库却用了一个更麻烦的实现方式，大家需要注意BoxBuilder这个类型的变量是干什么的。

为什么明明可以直接在一个方法里写完的代码，还要引入一个新的类型呢？原因就在于panic safety问题。注意我们这里调用了T类型的clone方法。T是一个泛型参数，谁能保证clone方法不会产生panic？没有

谁能保证，我们只能尽可能让clone发生panic的时候，RawVec的状态不会乱掉。

所以，标准库的实现利用了RAII机制，即便在clone的时候发生了panic，这个BoxBuilder类型的局部变量的析构函数依然会正确执行，并在析构函数中做好清理工作。上面这段代码之所以搞这么复杂，就是为了保证在发生panic的时候逻辑依然是正确的。

大家可以去翻一下标准库中的代码，有大量类似的模式存在，都是因为需要考虑panic safety问题。Rust的标准库在编写的时候有这样一个目标：即便发生了panic，也不会产生“内存不安全”和“线程不安全”的情况。

在Rust中，什么情况下panic会导致bug呢？这种情况的产生需要两个条件：

·panic导致了数据结构内部的状态错误；

·这个错误的状态会在以后被观测到。

在unsafe代码中，这种情况非常容易出现。所以，在写unsafe代码的时候，需要对这种情况非常敏感小心，一不小心就可能因为这个原因制造出“内存不安全”。

在不用unsafe的情况下，Panic Safety是基本有保障的。考虑一种场景、假如我们有两个数据结构，我们希望每次在更新其中一个的时候，也要对另外一个同步更新，如果不一致就会有问题。万一更新了其中一个，发生了panic，第二个没有正常更新怎么办？代码示例如下：

```
use std::panic;

fn main() {
    let mut x : Vec<i32> = vec![1];
    let mut y : Vec<i32> = vec![2];
    panic::catch_unwind(|| {
        x.push(10);
        panic!("user panic");
        y.push(100);
    }).ok();

    println!("Observe corruptted data. {:?} {:?}", x, y);
}
```

这里我们必须使用catch_unwind来阻止栈展开，否则这两个数据结构就一起被销毁了，无法观测到panic引发的错误状态。编译可见，这段代码是无法编译通过的，错误如下：

```
error[E0277]: the trait bound `&mut std::vec::Vec<i32>: std::panic::UnwindSafe` is r
```

这是什么原因呢？因为catch_unwind的签名是这样的：

```
pub fn catch_unwind<F: FnOnce() -> R + UnwindSafe, R>(f: F) -> Result<R>
```

它要求闭包参数满足UnwindSafe条件，而标准库中早就标记好了&mut型指针是不满足UnwindSafe trait的。有些类型，天生就不适合在catch_unwind的外部和内部同时存在。

有了这个约束条件，被panic破坏掉的数据结构被外部继续观测、使用的几率就小了许多。

当然，编译器是永远不知道用户的真实意图的，可能在某些场景下，用户就是要这样写，而且不认为这些数据结构是“被破坏”的状态。怎么修复上面这个编译错误呢？示例如下：

```
use std::panic;
use std::panic::AssertUnwindSafe;

fn main() {
    let mut x : Vec<i32> = vec![1];
    let mut y : Vec<i32> = vec![2];
    panic::catch_unwind(AssertUnwindSafe(|| {
        x.push(10);
        panic!("user panic");
        y.push(100);
    })).ok();

    println!("Observe corruptted data. {:?} {:?}", x, y);
}
```

我们可以用AssertUnwindSafe把这个闭包包一层，就可以强制突破编译器的限制了。我们也可以单独为某个变量来包一层，可以起到一样的效果。AssertUnwindSafe这个类型，不管内部包含的是什么数据，它都是满足catch_unwind函数约束的。这个设计至少能保证catch_unwind

可能造成的风险是显式标记出来的。

同理，在多线程情况下也有类似的问题。比如，我们把第29章经常用的示例的多线程修改全局变量的程序改改，在其中某个线程中制造一个panic：

```
use std::sync::Arc;
use std::sync::Mutex;
use std::thread;

const COUNT: u32 = 1000000;

fn main() {
    let global = Arc::new(Mutex::new(0));

    let clone1 = global.clone();
    let thread1 = thread::spawn(move|| {
        for _ in 0..COUNT {
            match clone1.lock(){
                Ok(mut value) => *value +=1,
                Err(poisoned) => {
                    let mut value = poisoned.into_inner();
                    *value += 1;
                }
            }
        }
    });

    let clone2 = global.clone();
    let thread2 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone2.lock().unwrap();
            *value -= 1;
            if *value < 100000 {
                println!("make a panic");
                panic!("");
            }
        }
    });

    thread1.join().ok();
    thread2.join().ok();
    println!("final value: {:?}", global);
}
```

在thread2中，在达到某个条件的情况下会发生panic。这个panic是在Mutex锁定的状态下发生的。这时，标准库会将Mutex设置为一个特殊的称为poisoned状态。处在这个状态下的Mutex，再次调用lock，会返回Err状态。它里面依然包含了原来的数据，只不过用户需要显式调用into_inner才能使用它。这种方式防止了用户在不小心的情况下产生异常不安全的风险。

18.4 小结

Rust语言是跟C++非常相似的一门语言。它们的定位相似，目标一致，都没有GC，都面向底层硬件，都希望提供比较好的高级抽象能力，都对性能有非常高的要求。而这也意味着，在许多情况下，它们会碰到同样的问题，有着类似的设计思路。

Rust跟C++最大的区别在于，它彻底摆脱了C/C++遗留下来的“前向兼容性”包袱，可以大刀阔斧地引入一些在其他语言中早已被证明了的优秀设计。Rust在保持底层定位的同时，把关注点主要放在了“安全性”问题上。除了前面介绍的“内存安全”、“线程安全”，Rust在“异常安全”方面的设计也非常不错，只是比较少用于拿出来宣传而已，毕竟只有C++是跟Rust遭遇了类似的问题，其他自带GC的语言，在这个问题上基本没什么需要特别关注的。在这个问题上，Rust通过一系列设计，将这个问题基本控制在了unsafe代码中。在safe代码部分，编译器会将可能发生异常安全的部分提示出来，让用户显式处理，用户基本不会由于不小心被这个问题“坑”到。

虽然Rust抛弃了“兼容性”这样一个巨大的负担，但是，这并不意味着设计这样一门语言是件轻松的事情。在“系统级”编程语言这个“紧箍咒”之下，很多看似简单的方面都需要权衡和妥协。比如，“易用性”“简洁性”总是永远排在“安全性”“性能”这样的目标后面。这也注定了Rust的内在复杂性并不低。

最后要强调的是，panic并不意味着“内存不安全”，恰恰相反，它是阻止“内存不安全”的利器。内存不安全造成的问题比程序突然退出要严重得多。在即将发生内存不安全现象的时候，如果当时已经没有任何其他选择，panic至少可以避免最坏情况的发生。然后我们可以很快发现事故的第一现场，从而修复代码，使其继续执行。

第19章　Unsafe

到目前为止，Rust的设计让人觉得非常放心。利用类型系统消除空指针，简洁明了的“唯一修改权”原则，消除了野指针，还有各种智能指针可以使用，甚至可以利用同样的规则消除多线程环境下的数据竞争，这一切就像一组简洁的数学定理一样，构建了一整套清晰的“内存安全”代码的“世界观”。

但是这只是Rust的一部分。还有一些情况，编译器的静态检查是不够用的，它没办法自动推理出来这段代码究竟是不是安全的。这种时候，我们就需要使用unsafe关键字来保证代码的安全性。

本章简单介绍一下Rust的unsafe代码。

19.1 unsafe关键字

Rust的unsafe关键字有以下几种用法：

·用于修饰函数fn；

·用于修饰代码块；

·用于修饰trait；

·用于修饰impl。

当一个fn是unsafe的时候，意味着我们在调用这个函数的时候需要非常小心。它可能要求调用者满足一些其他的重要约束，而这些约束条件无法由编译器自动检查来保证。有unsafe修饰的函数，要么使用unsafe语句块调用，要么在unsafe函数中调用。因此需要注意，unsafe函数是具有“传递性”的，unsafe函数的“调用者”也必须用unsafe修饰。

比如，String：：from_raw_parts就是一个unsafe函数，它的签名如下：

```
pub unsafe fn from_raw_parts(buf: *mut u8, length: usize, capacity: usize) -> String
```

它之所以是unsafe的，是因为String类型对所有者有一个保证：它内部存储的是合法的utf-8字符串。而这个函数没有检查传递进来的这个缓冲区是否满足这个条件，所以使用者必须这样调用：

```
// 自己保证这个缓冲区包含的是合法的 utf-8 字符串
let s = unsafe { String::from_raw_parts(ptr as *mut _, len, capacity) } ;
```

上面这个写法就是unsafe代码块的用法。使用unsafe关键字包围起来的语句块，里面可以做一些一般情况下做不了的事情。但是，它也是有规矩的。与普通代码比起来，它多了以下几项能力：

·对裸指针执行解引用操作；

·读写可变静态变量；

·读union或者写union的非Copy成员；

·调用unsafe函数。

在Rust中，有些地方必须使用unsafe才能实现。比如标准库提供的一系列intrinsic函数，很多都是unsafe的，再比如调用extern函数必须在unsafe中实现。另外，一些重要的数据结构内部也使用了unsafe来实现一些功能。

当unsafe修饰一个trait的时候，那么意味着实现这个trait也需要使用unsafe。比如在后面讲线程安全的时候会着重讲解的Send、Sync这两个trait。因为它们很重要，是实现线程安全的根基，如果由程序员来告诉编译器，强制指定一个类型是否满足Send、Sync，那么程序员自己必须很谨慎，必须很清楚地理解这两个trait代表的含义，编译器是没有能力推理验证这个impl是否正确的。这种impl对程序的安全性影响很大。

19.2 裸指针

Rust提供了两种裸指针供我们使用，*const T和*mut T。我们可以通过*mut T修改所指向的数据，而*const T不能。在unsafe代码块中它们俩可以互相转换。

裸指针相对于其他的指针，如Box，&，&mut来说，有以下区别：

·裸指针可以为空，而且编译器不保证裸指针一定指向一个合法的内存地址；

·不会执行任何自动化清理工作，比如自动释放内存等；

·裸指针赋值操作执行的是简单的内存浅复制，并且不存在borrow checker的限制。

创建裸指针是完全安全的行为，只有对裸指针执行“解引用”才是不安全的行为，必须在unsafe语句块中完成。

示例代码如下：

```
fn main() {
    let x = 1_i32;
    let mut y : u32 = 1;

    let raw_mut = &mut y as *mut u32 as *mut i32 as *mut i64; // 这是安全的

    unsafe {
        *raw_mut = -1;   // 这是不安全的,必须在 unsafe 块中才能通过编译
    }

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

我们可以把裸指针通过as运算符执行类型转换。转换类型之后，它就可以把它所指向的数据当成另外一个类型来操作了。原本变量y的类型是u32，但是我们对它取地址后，最后将指针类型转换成了i64。此时，我们对该指针所指向的地址进行修改会发生“类型安全”问题以及“内存安全”问题。编译，执行，这段代码的执行结果为：

```
FFFFFFFF FFFFFFFF
```

可见，x原本是一个在栈上存在的不可变绑定，在我们通过裸指针对y做了修改之后，x的值也发生了变化。原因就是，我们对指向y的指针类型做了转换，让它以为自己指向的是i64类型，恰巧x就在y旁边，城门失火，殃及池鱼，x就被顺带一起修改了。从这个示例我们可以看到，unsafe代码中可以做很多危险的事情。上面这个例子就是一个错误的unsafe用法。

再比如：

```
fn raw_to_ref<'a>(p: *const i32) -> &'a i32 {
    unsafe {
        &*p
    }
}
fn main() {
    let p : &i32 = raw_to_ref(std::ptr::null::<i32>());
    println!("{}", p);
}
```

编译，执行，可以看到发生了core dump。为什么呢？因为unsafe代码写错了。这段代码里面直接用unsafe功能把一个裸指针转换为了一个共享引用，忽略了Rust里面共享引用必须遵循的规则。在Rust中，&型引用、&mut型引用以及Box指针，全部要求是合法的非空指针。在unsafe代码中，我们必须自己从逻辑上保证这一点，否则就是不可容忍的严重bug。（注意在safe代码中是没办法构造出这样的场景的。）有些初学者可能会在写FFI，封装C代码的时候犯这样的错误。改正方法如下：

```
fn raw_to_ref<'a>(p: *const i32) -> Option<&'a i32> {
    if p.is_null() {
        None
    } else {
        unsafe { Some(&*p) }
    }
}
fn main() {
    let p : Option<&i32> = raw_to_ref(std::ptr::null::<i32>());
    println!("{:?}", p);
}
```

Rust的各种指针还有一些重要约束，比如&mut型指针最多只能同时存在一个。这些约束条件，在unsafe场景下是很容易被打破的，而编译器并没有能力帮我们自动检查出来。我们之所以需要unsafe，只是因为某些代码只有在特定条件下才是安全的，而这个条件我们没有办法利用类型系统表达出来，所以这时候需要依靠我们自己来保证。

大家千万不要到处滥用unsafe。当你不得不使用unsafe的时候，请一定注意，这并不意味着你就可以乱写不安全的代码，相反，它的意思是“编译器请相信我，这段代码依然是安全的，它的安全性由我自己负责”。

裸指针也有很多有用的成员方法，读者可以参考标准文档中的“Primitive Type Pointer”。比如，裸指针并不直接支持算术运算，而是提供了一系列成员方法offset wrapping_offset等来实现指针的偏移运算。

19.3 内置函数

在标准库中，有一个std：：intrinsics模块，它里面包含了一系列的编译器内置函数。这些函数都有一个extern"rust-intrinsic"修饰，它们看起来都像一种特殊的FFI外部函数，大家打开标准库的源代码src/core/intrinsics.rs，可以看到这些函数根本没有函数体，因为它们的实现是在编译器内部，而不是在标准库内部。调用它们的时候都必须使用unsafe才可以。编译器见到这些函数，就知道应该生成什么样的代码，而不是像普通函数调用一样处理。另外，intrinsics是藏在一个feature gate后面的，这个feature可能永远不会稳定，这些函数就不是准备直接提供给用户使用的。一般标准库会在这些函数基础上做一个更合适的封装给用户使用。

下面就在这些函数中挑一部分作介绍。

19.3.1 transmute

fn transmute<T，U>（e：T）->U函数可以执行强制类型转换。把一个T类型参数转换为U类型返回值，转换过程中，这个参数的内部二进制表示不变。但是有一个约束条件，即size_of：：<T>（）==size_of：：<U>（）。如果不符合这个条件，会发生编译错误。transmute_copy的作用跟它类似，区别是，参数类型是一个借用为&T。

一般情况下，我们也可以用as做类型转换，把&T类型指针转换为裸指针，然后再转换为&U类型的指针。这样也可以实现类似的功能。但是用户自己实现的泛型函数有一个缺陷，即无法在where条件中自己表达size_of：：<T>（）==size_of：：<U>（）。而transmute作为一个内置函数就可以实现这样的约束。

transmute和transmute_copy在std：：mem模块中重新导出。用户如果需要，请使用这个模块，而不是std：：intrinsics模块。

下面用一个示例演示一下Vec类型的二进制表示是怎样的：

```
fn main() {
```

```
    let x = vec![1,2,3,4,5];

    unsafe {
        let t : (usize, usize, usize) = std::mem::transmute_copy(&x);
        println!("{} {} {}", t.0, t.1, t.2);
    }
}
```

上面的例子中，我们调用了transmute_copy，因此参数类型是&Vec。假如我们用transmute函数，参数类型就必须是Vec，区别在于，参数会被move进入这个函数中，在后面就不能继续使用了。在调用transmute_copy函数的时候，必须显示指定返回值类型，因为它是泛型函数，返回值类型可以有多种多样的无穷变化，只要满足size_of：：<T>（）==size_of：：<U>（）条件，都可以完成类型转换。所以编译器自己是无法自动推理出返回值类型的。在上例中，我们的返回值类型是包含三个usize的tuple类型。这是因为，Vec中实际包含了3个成员，一个是指向堆上的指针，一个是指向内存空间的总大小，还有一个是实际使用了的元素个数，因此这个类型转换从编译器看来是满足“占用内存空间相同”这一条件的。

编译执行，我们就可以看到Vec内部的具体内存表示了。执行结果为：

```
6393920 5 5
```

19.3.2 内存读写

intrinsics模块里面有几个与内存读写相关的函数，比如copy、copy_nonoverla-pping、write_bytes、move_val_init、volatile_load等。这些函数又在std：：ptr/std：：mem模块中做了个简单封装，然后暴露出来给用户使用。下面挑其中几个重要的函数介绍。

1.copy

copy的完整签名是unsafe fn copy<T>（src：*const T，dst：*mut T，count：usize）。它做的就是把src指向的内容复制到dst中去。它跟C语言里面的memmove类似，都假设src和dst指向的内容可能有重叠。区别是memmove的参数是void*类型，第三个参数是字节长度，而ptr：：

copy的指针是带类型的，第三个参数是对象的个数。

这个模块中还提供了ptr：：copy_nonoverlapping。它跟C语言里面的memcpy很像，都假设用户已经保证了src和dst指向的内容不可重叠。所以ptr：：copy_nonoverlapping的执行速度比ptr：：copy要快一些。

2.write

在ptr模块中，write的签名是unsafe fn write<T>（dst：*mut T，src：T），作用是把变量src写入到dst所指向的内存中。注意它的参数src是使用的类型T，执行的是move语义。查看源码可知，它是基于intrinsics：：move_val_init实现的。注意，在写的过程中，不管dst指向的内容是什么，都会被直接覆盖掉。而src这个对象也不会执行析构函数。

写内存还有ptr：：write_bytes、ptr：：write_unaligned、ptr：：write_vol-a-tile等函数。

3.read

在ptr模块中，read的签名是unsafe fn read<T>（src：*const T）->T，作用是把src指向的内容当成类型T返回去。查看它的内部源码，可见它就是基于ptr：：copy_nonoverlapping实现的。

读内存还有ptr：：read_unaligned以及ptr：：read_volatile两个函数，大同小异。

以上这些内存读写函数，都是不管语义，直接操作内存中的字节。所以它们都是用unsafe函数。

4.swap

在ptr模块中，swap的签名是unsafe fn swap<T>（x：*mut T，y：*mut T），作用是把两个指针指向的内容做交换。两个指针所指向的对象都只是被修改，而不会被析构。

这个函数在mem模块中又做了一次封装，变成了unsafe fn swap<T>（x：&mut T，y：&mut T）供用户使用。签名中的&mut型引用可以保证这两个指针是当前唯一指向该对象的指针。

某些特殊类型还有自己的swap成员函数。比如Cell：：swap（&self，other：&Cell<T>）。这个函数跟其他swap函数最大的区别在于，它的参数只要求共享引用，不要求可变引用。这是因为Cell本身的特殊性。它具备内部可变性，所以这么设计是完全安全的。我们可以从源码看到，它就是简单地调用了ptr：：swap。

5.drop_in_place

在ptr模块中，drop_in_place的签名是unsafe fn drop_in_place<T：？Sized>（to_drop：*mut T）。它的作用是执行当前指向对象的析构函数，如果没有就不执行。

6.uninitialized

在mem模块中，uninitialized的签名是unsafe fn uninitialized<T>（）->T。它是基于intrinsics：：uninit函数实现的。我们知道，Rust编译器要求每个变量必须在初始化之后再使用，如果在某些情况下，你确实需要未初始化的变量，那么必须使用unsafe才能做到。注意，任何时候读取未初始化变量都是未定义行为，请大家不要这么做。即便你在unsafe代码中创造了未初始化变量，也需要自己在逻辑上保证，读取这个变量之前先为它合理地赋过值。

另外，这个函数有点像std：：mem：：forget，调用这个函数，不仅不会在程序中增加代码，反而会减少可执行代码。调用forget，会导致编译器不再插入析构函数调用的代码，调用uninitialized会导致缺少初始化。它们没有任何运行开销。

uninitialized函数也还没有稳定，它有一些无法克服的缺陷，将来标准库会废弃掉这个函数，而使用一个新的类型让用户在unsafe代码中创建未初始化变量。

19.3.3 综合示例

下面我们用一个示例来演示一下这些unsafe函数的用途，以及怎样才能正确调用unsafe代码。示例很简单，就是实现标准库中的内存交换函数std：：mem：：swap。

我们可以确定这个函数的签名是fn swap<T>（x：&mut T，y：&mut T）。关于泛型的解释在第21章中有，此处略过不提。先试一个最简单的实现：

```
fn swap<T>(x: &mut T, y: &mut T) {
    let z : T = *x;
    *x = *y;
    *y = z;
}
```

编译不通过。因为let z=*x；执行的是move语义，编译器不允许我们把x指向的内容move出来，这只是一个借用而已。如果允许执行这样的操作，会导致原来的借用指针x指向非法数据。但是我们知道，我们这个函数整体上是可以保证安全的，因为我们把x指向的内容move出来之后，会用其他的正确数据填回去，最终可以保证函数执行完之后，x和y都是一个正常的状态。这种时候，我们就需要动用unsafe了，代码如下：

```
fn swap<T>(x: &mut T, y: &mut T) {
    unsafe {
        let mut t: T = mem::uninitialized();

        ptr::copy_nonoverlapping(&*x, &mut t, 1);
        ptr::copy_nonoverlapping(&*y, x, 1);
        ptr::copy_nonoverlapping(&t, y, 1);

        mem::forget(t);
    }
}
```

代码逻辑的意思如下。

首先，我们依然需要一个作为中转的局部变量。这个局部变量该怎么初始化呢？其实我们不希望它执行初始化，因为我们只需要这部分内存空间而已，它里面的内容马上就会被覆盖掉，做初始化是浪费性能。况且，我们也不知道用什么通用的办法初始化一个泛型类型，它连Default约束都未必满足。所以我们要用mem：：uninitialized函数。

接下来，我们可以直接通过内存复制来交换两个变量。因为在Rust中，所有的类型、所有的move操作，都是简单的内存复制，不涉及其他的语义。Rust语言已经假定任何一个类型的实例，随时都可以被move

到另外的地方，不会产生任何问题。所以，我们可以直接使用ptr：：copy系列函数来完成。再加上在safe代码中，&mut型指针具有排他性，我们可以确信，x和y一定指向不同的变量。所以可以使用ptr：：copy_nonoverlapping函数，比ptr：：copy要快一点。

最后，一定要记得，要阻止临时的局部变量t执行析构函数。因为t本身并未被合理地初始化，它内部的值是直接通过内存复制获得的。在复制完成后，它内部的指针（如果有的话）会和y指向的变量是相同的。如果我们不阻止它，那么在函数结束的时候它的析构函数就会被自动调用，这样y指向的变量就变成非法的了。

这样我们才能正确地完成这个功能。虽然源代码看起来比较长，但是实际生成的代码并不多，就是3次内存块的复制。假设执行的时候泛型参数T被实例化为Vec<i32>，这个swap函数的执行流程如图19-1所示。

在新版本的标准库的源码中，做法比这个更复杂，主要是为了更好地优化执行效率。它并没有在内存中分配一个临时对象，而是尽可能利用寄存器做数据交换。具体细节就不展开了，大家可以自己去读源码，自己尝试做性能测试。

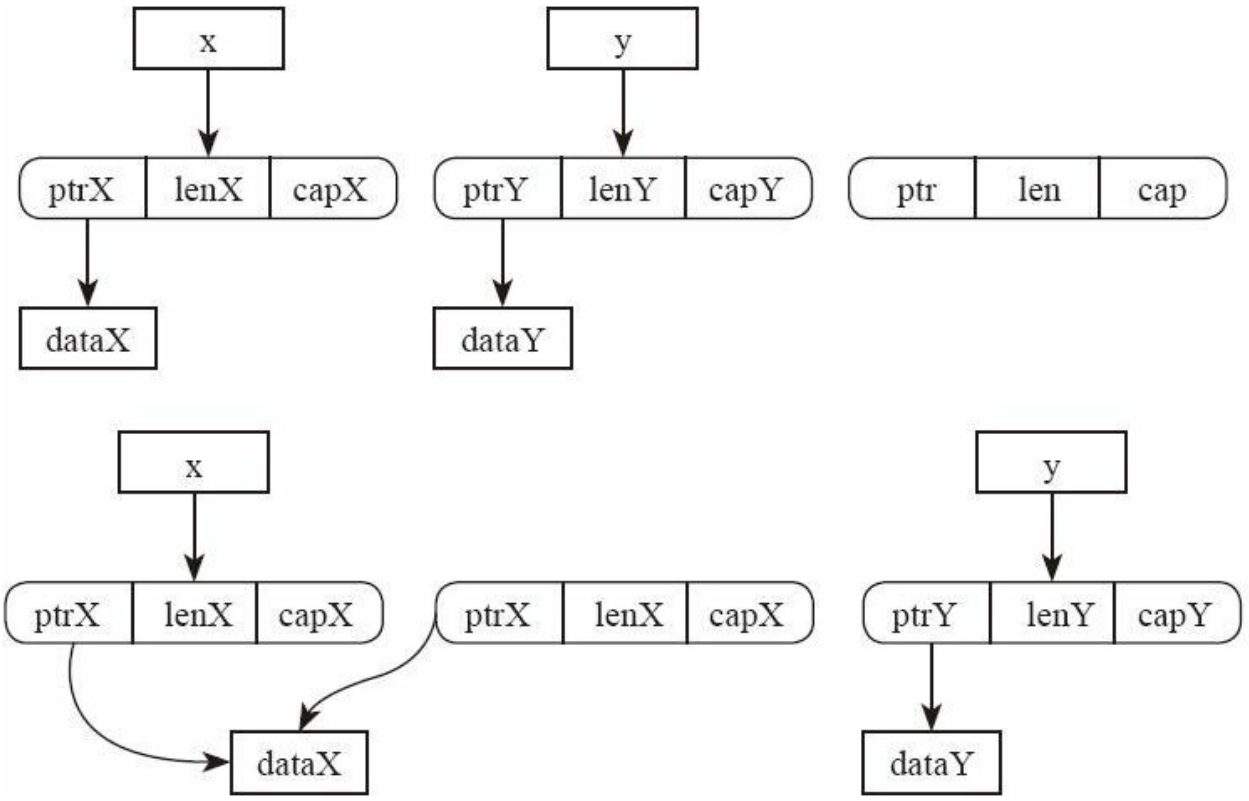

x
ptrX
lenX
capX
dataX
y
ptrY
lenY
capY
dataY
ptr
len
cap
x
ptrX
lenX
capX
dataX
ptrX
lenX
capX
y
ptrY
lenY
capY
dataY

图　19-1

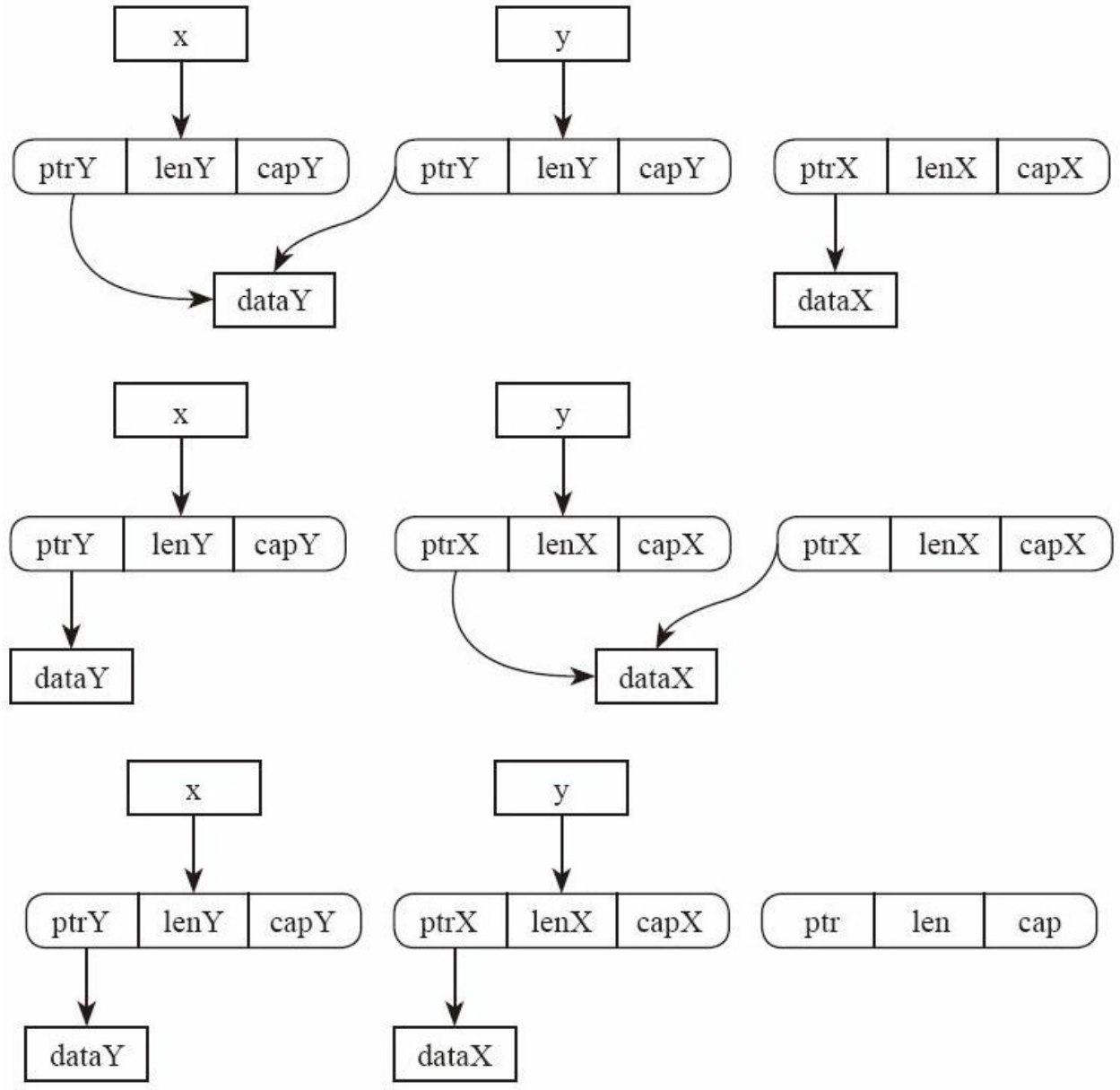

x
y
ptrY
lenY
capY
ptrY
lenY
capY
ptrX
lenX
capX
dataY
dataX
x
y
ptrY
lenY
capY
ptrX
lenX
capX
ptrX
lenX
capX
dataY
dataX
x
y
ptrY
lenY
capY
ptrX
lenX
capX
ptr
len
cap
dataY
dataX

图19-1 （续）

19.4 分割借用

“alias+mutation”规则非常有用。然而，alias分析在碰到复合数据类型的时候也会非常无奈。我们看看下面的示例：

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

fn main() {
    let mut x = Foo {a: 0, b: 0, c: 0};
    let pa = &mut x.a;
    let pb = &mut x.b;
    let pc = &x.c;
    *pb += 1;
    let pc2 = &x.c;
    *pa += 10;
    println!("{} {} {} {}", pa, pb, pc, pc2);
}
```

这种情况，Rust编译器可以愉快地通过编译，因为它知道，指针pa、pb、pc分别指向的是不同的内存区域，它们之间不是alias关系，所以这些指针完全可以共存。但是，我们把数据类型从结构体转为数组之后，情况就变了：

```
fn main() {
    let mut x = [1_i32, 2, 3];
    let pa = &mut x[0];
    let pb = &mut x[1];
    let pc = &x[2];
    *pb += 1;
    let pc2 = &x[2];
    *pa += 10;
    println!("{} {} {} {}", pa, pb, pc, pc2);
}
```

这段代码所做的事情其实跟上面一段代码基本一样。编译，发生错误，错误信息为：

```
error: cannot borrow `x[..]` as mutable more than once at a time
```

这时候，Rust编译器判定pa、pb、pc存在alias关系。它没办法搞清楚&x[A]、&x[B]、&x[C]之间是否有可能有重叠。为什么编译器没办法搞清楚它们之间是否有重叠呢？

首先，要考虑到实际程序中作为索引的变量很可能不是编译期常量，而是根据运行期的值决定，A、B、C可以是任意表达式。

其次，索引操作运算符其实是在标准库中实现的，除了数组，还有许多类型也能支持索引操作，比如HashMap。即便编译器知道了A、B、C三个索引没有重叠，它也无法直接推理出&x[A]、&x[B]、&x[C]之间是否有重叠。两个不同字符串做索引实际上指向了同一个值也有可能。

所以，对于结构体类型，编译器可以很轻松地知道，指向不同成员的指针一定不重叠，而对于数组切片，编译器的推理结果是将x[_]视为一个整体，&x[A]、&x[B]、&x[C]之间都算重叠。虽然读者可以看出来，&mut x[0..2]和&mut x[3..4]根本就是指向两块独立的内存区域，它们同时存在是完全安全的。但是编译器却觉得，&mut x[A]和&mut x[B]一定不能同时存在，否则就违反了alias+mutation的设计原则。

那么面对这样的情况，Rust该如何解决呢？如果说我们可以确定两块数组切片确实不存在重叠，我们应该怎么告诉编译器呢？

这就需要用到标准库中的split_at以及split_at_mut方法。首先，我们看看它的使用方式：

```
fn main() {
    let mut x = [1_i32, 2, 3];
    {
        let (first, rest)  : (&mut [i32], &mut [i32]) = x.split_at_mut(1);
        let (second, third): (&mut [i32], &mut [i32]) = rest.split_at_mut(1);
        first[0]  += 2;
        second[0] += 4;
        third[0]  += 8;
        println!("{:?} {:?} {:?}", first, second, third);
    }
    println!("{:?}", &x);
}
```

执行结果为：

```
[3] [6] [11]
[3, 6, 11]
```

使用split_at_mut方法，可以将一个Slice切分为两个部分返回，返回值中包括的两个值分别都是指向原Slice的&mut[T]型切片。这样可以保证这两个数组切片一定不会发生重叠，因此它们可以同时存在两个&mut型指针，同时修改原来的数组，而不会制造内存不安全。

那么split_at_mut方法内部实现是怎么做的呢？它的源码如下所示：

```
#[inline]
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();

    unsafe {
        assert!(mid <= len);

        (from_raw_parts_mut(ptr, mid),
        from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

整体的逻辑并不复杂，只是需要一点unsafe代码来辅助。这说明，Rust的这套alias+mutation规则，虽然威力巨大，但也误伤了一些“好人”。存在一些情况是，实际上内存安全但是无法被编译器接受。面对这样的情况，我们可以利用Rust的unsafe代码，case by case地解决问题。

另外需要强调的是，unsafe一定不能滥用。应该尽量把unsafe代码封装在一个较小的范围，对外公开的是完全safe的API。如果不懂得如何抽象，把unsafe散落在业务逻辑的各个角落中，那么这就相当于退化成了C语言，甚至更糟糕。

还需要特别强调的一点是，用户更加不要做自欺欺人的事情，把一个明明是unsafe的函数声明成普通函数，仅仅为了调用的时候稍微方便一点。请大家一定要注意：如果一个函数有可能在某些场景下制造出内存不安全，那么它必须用unsafe标记，不能偷懒。哪怕这个可能性微乎其微也不行。在调用unsafe函数的地方，调用者必须反复确认被调用的这个unsafe函数的前置条件和后置条件是否满足，不能简单地用unsafe代码块框起来。

19.5 协变

19.5.1 什么是协变

Rust的生命周期参数是一种泛型类型参数。比如，我们可以这样理解共享引用：

```
type StrRef<'a> = &'a str;
```

这是一个指向字符串的借用指针。它是一个泛型类型，接受一个泛型参数，之后形成一个完整类型。它跟Vec<T>很像，只不过Rust里面泛型类型参数既有生命周期，又有普通类型。下面是一个示例：

```
type StrRef<'a> = &'a str;

fn print_str<'b>(s: StrRef<'b>) {
    println!("{}", s);
}

fn main() {
    let s : StrRef<'static> = "hello";
    print_str(s);
}
```

这个例子中演示了一种有意思的现象。大家看一下，print_str接受的参数类型是Str-Ref<'b>，而实际上传进来的参数类型是StrRef<'static>，这两个类型并不完全一致，因为'b！='static。但是Rust可以接受。这种现象在类型系统中被称为“协变”（covariance）和“逆变”（contravariance）。

协变和逆变的定义如下。我们用<：符号记录子类型关系，对于泛型类型C<T>，

·协变

若T1<：T2时满足C<T1><：C<T2>，则C对于参数T是协变关系。

·逆变

若T1<：T2时满足C<T2><：C<T1>，则C对于参数T是逆变关系。

·不变

上述两种都不成立。

总结起来就是，如果类型构造器保持了参数的子类型关系，就是协变；如果逆转了参数的子类型关系，就是逆变。其他情况，就是不变。

Rust不支持普通泛型参数类型的协变和逆变，只对生命周期泛型参数存在协变和逆变。

在Rust中，泛型类型支持针对生命周期的协变是一个重要功能。大家试想一下，下面这条语句为什么能成立：

```
let s : &str = "hello";
```

"hello"是一个字符串字面量，它的类型是&'static str。而s是一个局部变量，它的类型是&'s str。其中泛型参数在源码中省略掉了，这个生命周期泛型参数代表的是这个局部变量从声明到结束的这段区域。在这句话中，我们把一个生命周期更长的引用&'static str，赋值给了一个生命周期更短的引用&'a str，这是没问题的。原因在于，既然这边被指向的目标在更长生命周期内都是合法的，那么它在一个较短生命周期内也一定是合法的。所以，我们可以说引用类型&对生命周期参数具有协变关系。（此处有些争论，有人认为这里应该理解为逆变关系，主要的争议来自于我们很难说清两个生命周期，究竟准是准的子类型。本书中为了行文的方便，继续使用“协变”，但主要意思是“协变or逆变”，是跟“不变”的概念相对立的。）

接下来，我们可以通过一些示例继续理解其他一些泛型类型的协变关系。示例如下：

```
fn test<'a>(s : &'a &'static str) {
    let local : &'a &'a str = s;
}
```

从这个示例我们可以看到，&'a &'static str类型可以安全地赋值给&'a &'a str类型。由于&'static str<：&'a str以及&'a &'static str<：&'a &'a str关系成立，这说明引用类型针对泛型参数T也是具备协变关系的。

把上面的示例改一下，试试&'a mut T型指针：

```
fn test<'a>(s : &'a mut &'static str) {
    let local : &'a mut &'a str = s;
}
```

编译，可见出现了生命周期错误。这说明从&'a mut &'static str类型到&'a mut &'a str类型的转换是不安全的。此事可以说明，&mut型指针针对泛型T参数是不变的。

下面再试试Box类型：

```
fn test<'a>(s : Box<&'static str>) {
    let local : Box<&'a str> = s;
}
```

这段代码可以编译通过，说明从Box<&'static str>类型到Box<&'a str>类型的转换是安全的。所以Box<T>类型针对T参数是具备协变关系的。

下面再试试函数fn类型。注意fn类型有两个地方可以使用泛型参数，一个是参数那里，一个是返回值那里。我们写两个测试用例：

```
fn test_arg<'a>(f : fn(&'a str)) {
    let local : fn(&'static str) = f;
}

fn test_ret<'a>(f : fn()->&'a str) {
    let local : fn()->&'static str = f;
}
```

test_arg可以通过编译，test_ret不能通过。意思是，fn（&'a str）类型可以转换为fn（&'static str）类型，而fn（）->&'a str类型不能转换为fn（）->&'static str类型。这意味着类型fn（T）->U对于泛型参数T具备协变关系，对于U不具备协变关系。如果我们把这个测试改一下，尝试

把生命周期参数换个位置：

```
fn test_ret<'a>(f : fn()->&'a str) {
    f();
}

fn main() {
    fn s() -> &'static str { return ""; }

    test_ret(s);
}
```

上面这段代码可以编译通过。这意味着fn（）->&'static str类型可以安全地转换为fn（）->&'a str类型。那我们可以说，类型fn（T）->U对于参数U具备逆变关系。

再换成具备内部可变性的类型试验：

```
use std::cell::Cell;

fn test<'a>(s : Cell<&'static str>) {
    let local : Cell<&'a str> = s;
}
```

编译出现了生命周期不匹配的错误。这说明Cell<T>类型针对T参数不具备协变关系。至于为什么要这样设计，前面已经讲过了，如果具备内部可变性的类型还有生命周期协变关系，可以构造出悬空指针的情况。所以需要编译器提供的UnsafeCell来表达针对类型参数具备“不变”关系的泛型类型。

同样，我们可以试试裸指针：

```
fn test<'a>(s : *mut &'static str) {
    let local : *mut &'a str = s;
}
```

可以得出结论，*const T针对T参数具备协变关系，而*mut T针对T参数是不变关系。比如标准库里面的Box<T>。它的内部包含了一个裸指针，这个裸指针就是用的*const T而不是*mut T。这是因为我们希望Box<T>针对T参数具备协变关系，而*mut T无法提供。

在写unsafe代码的时候，特别是涉及泛型的时候，往往需要我们手动告诉编译器，这个类型的泛型参数究竟应该是什么协变关系。很多情况下，我们需要使用PhantomData类型来表达这个信息。

19.5.2 PhantomData

在写unsafe代码的时候，我们经常会碰到一种情况，那就是一个类型是带有生命周期参数的，它表达的是一种借用关系。可是它内部是用裸指针实现的。请注意，裸指针是不带生命周期参数的。于是就发生了下面这样的情况：

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

然而，在Rust中，如果一个类型的泛型参数从来没有被使用过，那么就是一个编译错误。请参考RFC 0738-variance。如果一个泛型参数从来没有使用过，那么编译器就不知道这个泛型参数对于这个类型是否具备协变逆变关系，那么就可能在生命周期分析的时候做出错误的结论。所以编译器禁止未使用的泛型参数。在这种情况下，我们可以使用PhantomData类型来告诉编译器协变逆变方面的信息。

PhantomData没有运行期开销，它只在类型系统层面有意义。比如，一个自定义类型有一个泛型参数'a没有被使用，为了表达这个类型对于泛型参数'a具备协变关系，那我们可以为它加一个成员，并且把类型制定为PhantomData<'aT>即可。因为前面我们已经说了&'a T类型对于泛型参数'a具备协变关系，所以编译器就可以推理出来这个自定义类型对于泛型参数'a具备协变关系。其他用法与此类似，你要表达一个什么样的协变逆变关系，就找一个现存的类似的类型，拿它当作PhantomData的泛型参数即可。

PhantomData定义在std：：marker模块中：

```
#[lang = "phantom_data"]
#[stable(feature = "rust1", since = "1.0.0")]
pub struct PhantomData<T:?Sized>;
```

它是一个0大小的、特殊的泛型类型。它上面有#[lang=…]属性标记，这说明它是编译器特殊照顾的类型。它主要是用于写unsafe代码时，告诉编译器这个类型的语义信息。

如果你想表达这个类型对T类型成员有拥有关系，那么可以使用PhantomData<T>。例如std：：core：：ptr：：Unique：

```
pub struct Unique<T: ?Sized> {
    pointer: NonZero<*const T>,
    _marker: PhantomData<T>,
}
```

如果你想表达这个类型对T类型成员有借用关系，那么可以使用PhantomData<&'a T>。

你还可以用它来表明当前这个类型不可Send、Sync，示例如下：

```
struct MyStruct {
    data: String,
    _marker: PhantomData<*mut ()>,
}
```

下面同样用比较完整的示例来演示一下这个类型的具体作用。假设我们现在有两个类型：

```
use std::fmt::Debug;

#[derive(Clone, Debug)]
struct S;

#[derive(Debug)]
struct R<T: Debug> {
    x: *const T
}
```

其中R类型想表达一种借用关系，它内部需要用裸指针实现。上面这种简单的写法是有问题的，因为我们可以很容易制造出悬空指针：

```
fn main() {
    let mut r = R { x: std::ptr::null() };
    {
        let local = S{};
```

```
            r.x = &local;
        }
        // r.x now is dangling pointer
    }
```

为了让编译器使用borrow checker检查这种内存错误，我们可以给R类型添加一个生命周期参数，并且利用PhantomData使用这个生命周期参数，避免“未使用泛型参数”的错误。同时给R类型增加一个成员方法，在成员方法中改变指针的地址，并且通过模块系统禁止外部用户直接访问R的内部成员。完整代码如下所示：

```
use std::fmt::Debug;
use std::ptr::null;
use std::marker::PhantomData;

#[derive(Clone, Debug)]
struct S;

#[derive(Debug)]
struct R<'a, T: Debug + 'a> {
    x: *const T,
    marker: PhantomData<&'a T>,
}

impl<'a, T: Debug> Drop for R<'a, T> {
    fn drop(&mut self) {
        unsafe { println!("Dropping R while S {:?}", *self.x) }
    }
}

impl<'a, T: Debug + 'a> R<'a, T> {
    pub fn ref_to<'b: 'a>(&mut self, obj: &'b T) {
        self.x = obj;
    }
}
fn main() {
    let mut r = R { x: null(), marker: PhantomData };
    {
        let local = S { };
        r.ref_to(&local);
    }
}
```

再编译，我们可以看到，这次编译器就可以成功发现生命周期错误，禁止悬挂指针的产生。在写FFI给C代码做封装的时候，需要经常使用裸指针，这时就可以用类似的技巧来处理生命周期的问题。

19.6 未定义行为

在C/C++等语言中，未定义行为（undefined behavior，简称UB）指的是，在某些情况下语言标准允许编译器做任何事情，无论发生什么后果都是正常的，不属于编译器的bug。比如，对空指针做“解引用”操作，在C/C++里面就是未定义行为，编译器可以决定做任何事情。

在C/C++标准里面，很多情况下，都有充分的理由将某些行为定义为UB，这是语言本身的定位决定的。它可以简化编译器的设计，也可以最大化执行效率，还可以最大化跨平台，等等。但是我们也应该承认，过多的UB是对用户极其不友好的。在Rust里面，UB被限制在了一个较小的范围内，只有unsafe代码有可能制造出UB，这也是在写unsafe代码的时候需要注意的。

下面列举一些undefined behavior，摘抄自Rust的Reference文档：

·数据竞争

·解引用空指针或者悬空指针

·使用未对齐的指针读写内存而不是使用read_unaligned或者write_unaligned

·读取未初始化内存

·破坏了指针别名规则

·通过共享引用修改变量（除非数据是被UnsafeCell包裹的）

·调用编译器内置函数制造UB

·给内置类型赋予非法值

·给引用或者Box赋值为空或者悬空指针

·给bool类型赋值为0和1之外的数字

·给enum类型赋予类型定义之外的tag标记

·给char类型赋予超过char：：MAX的值

·给str类型赋予非utf-8编码的值

以上只是一些典型的问题，完整列表请大家参考官方文档。这些问题只可能在写unsafe代码的时候出现，这都是需要读者注意的地方。

19.7 小结

Rust的unsafe关键字是一个难点，也是很多初学者困惑的地方。很多人有这样的疑惑：既然Rust允许使用unsafe来完成许多危险的操作，那么Rust的安全性保证是不是就没什么意义了？

这件事情不能这么理解。unsafe的存在不是来故意破坏安全性的，它只是一种面向更底层操作的接口。不同的高级语言对于什么是底层的定义是不同的，但是所有的高级语言，只要不断往底层探究，总会碰到safe与unsafe之间的分界线。比如，Java有自己的JNI机制，C#也有unsafe关键字，Python也可以调用C模块，甚至C/C++语言都可以内嵌汇编。当你在高级语言中与底层操作交互的时候，必须确保高级语言中的一些规则和约定。Java、C#这类语言，利用GC实现了内存安全，但是用户同样可以使用JNI/unsafe实现不安全的操作，但这件事情并不意味着Java、C#语言本身有安全性缺陷。同理，在C语言里面用内嵌汇编搞乱了堆栈，也不能说是C语言的设计缺陷。只不过是用户使用这些机制的时候，没有一个自动检查工具来保证安全性，而是必须由自己来保证上层代码和下层代码之间交互的正确性。

Rust的unsafe最大的问题在于，到目前为止，依然没有一份官方文档来明确哪些东西是用户可以依赖的、哪些是编译器实现相关的、哪些是以后永远不变的、哪些是将来可能会有变化的。所以，哪怕用户能确保自己写出来的unsafe代码在目前版本上是完全正确的，也没办法确保不会在以后的版本中出问题。如果以后编译器的实现发生了变化，导致了unsafe代码无法正常工作，究竟算是编译器的bug还是用户错误地依赖了某些特性，还说不清楚。正式的unsafe guideline还在继续编写过程中。（当然这种错误情况几率是很低的，绝大多数用户使用unsafe的时候都是在FFI场景下，不会涉及那些精微细密的语义规则。）

我们既不能过于滥用unsafe，也不该对它心怀恐惧。它只是表明，某些代码的安全性依赖于某些条件，而我们无法清晰地在代码中表达这些约束条件，因此无法由编译器帮我们自动检查。

unsafe是Rust的一块重要拼图，充分理解unsafe的意义和作用，才能让我们更好地理解safe的来源和可贵。

不尽知用兵之害者，则不能尽知用兵之利也。——孙子兵法

第20章　Vec源码分析

本节将通过一个比较完整的例子，把内存安全问题分析一遍。本节选择的例子是标准库中的基本数据结构Vec<T>源代码分析。之所以选择这个模块作为示例，其一是因为这个类型作为非常基础的数据结构，平时用得很多，大家都很熟悉；第二个原因是，恰好它的内部实现又完全展现了Rust内存安全的方方面面，深入剖析它的内部实现非常有利于加深我们对Rust内存安全的认识。本章中用于分析的代码是1.23 nightly版本，Vec的内部实现源码在此之前一直有所变化，以后也很可能还会有变化，请读者注意这一点。

我们先从使用者的角度分析一下Vec是如何自动管理内存空间的：

```
fn main() {
    let mut v1 = Vec::<i32>::new();
    println!("Start: length {} capacity {}", v1.len(), v1.capacity());

    for i in 1..10 {
        v1.push(i);
        println!("[Pushed {}] length {} capacity {}", i, v1.len(), v1.capacity());
    }

    let mut v2 = Vec::<i32>::with_capacity(1);
    println!("Start: length {} capacity {}", v2.len(), v2.capacity());

    v2.reserve(10);
    for i in 1..10 {
        v2.push(i);
        println!("[Pushed {}] length {} capacity {}", i, v2.len(), v2.capacity());
    }
}
```

编译，执行，从结果中可以看出，如果用new方法构造出来，一开始的时候是没有分配内存空间的，capacity为0。我们也可以使用with_capacity来构造新的Vec，可以自行指定预留空间大小，还可以对已有的Vec调用reserve方法扩展预留空间。在向容器内部插入数据的时候，如果当前容量不够用了，它会自动申请更多的空间。当变量生命周期结束的时候，它会自动释放它管理的内存空间。

20.1 内存申请

首先，我们知道Vec是一个动态数组，它会根据情况动态扩展当前的空间大小。在Rust以及C++这样的语言中，动态数组这样的类型是由库来实现的，而不是像某些带GC的语言一样由运行时环境内置提供。这是因为Rust和C++一样，都具备对内存的直接控制力，其中一个表现就是，可以手动调用内存分配器，自己管理内存分配和释放的策略。动态数组类型的基本思想是：它不像内置数组一样直接把数据保存到栈上，而是在堆上开辟一块空间来保存数据，在向Vec中插入数据的时候，如果当前已分配的空间不够用了，它会重新分配更大的内存空间，把原有的数据复制过去，然后继续执行插入操作。

在目前版本的标准库中，Vec类型的源码存在于liballoc/vec.rs文件中。它的定义很简单：

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

它只有两个成员：一个是RawVec<T>类型，管理内存空间的分配和释放；另外一个是usize类型，记录当前包含的元素个数。Vec的new和capacity方法都很简单：

```
pub fn new() -> Vec<T> {
    Vec {
        buf: RawVec::new(),
        len: 0,
    }
}
pub fn with_capacity(capacity: usize) -> Vec<T> {
    Vec {
        buf: RawVec::with_capacity(capacity),
        len: 0,
    }
}
```

我们继续深入查看RawVec这个类型的new和capacity方法是如何实现的。它的源码在liballoc/raw_vec.rs文件中：

```
pub struct RawVec<T, A: Alloc = Heap> {
    ptr: Unique<T>,
    cap: usize,
    a: A,
}

impl<T> RawVec<T, Heap> {
    pub fn new() -> Self {
        Self::new_in(Heap)
    }
    pub fn with_capacity(cap: usize) -> Self {
        RawVec::allocate_in(cap, false, Heap)
    }
}
```

从这里可以看到，RawVec的泛型参数比Vec多了一个，这个泛型参数代表的是内存分配器allocator。这个设计的目的是让Rust的标准容器能像C++中的一样，可以由用户自行指定内存分配器。这个功能目前还处于设计过程中，因此只有RawVec中有这个功能，而Vec还没有，以后Vec同样也会有这样一个泛型参数的。这个泛型参数有一个默认值，叫作Heap，是标准库给我们提供的默认内存分配器。一般来说，内存分配器都是0大小的类型，它没有任何成员，不保存任何状态信息。比如Heap这个类型的定义：

```
#[derive(Copy, Clone, Default, Debug)]
pub struct Heap;
```

继续分析RawVec的new方法。它调用了它自己的new_in方法，并将allocator作为参数传递进去。因为Heap类型没有成员，所以它可以直接用它的名字当作一个对象实例来使用。这个new_in方法是这样实现的：

```
pub fn new_in(a: A) -> Self {
    // !0 is usize::MAX. This branch should be stripped at compile time.
    let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

    // Unique::empty() doubles as "unallocated" and "zero-sized allocation"
    RawVec {
        ptr: Unique::empty(),
        cap,
        a,
    }
}
```

对于成员ptr以及成员a，都是简单的默认构造。只有成员cap的处理

稍微麻烦一点。主要是考虑到0大小类型的问题，如果类型参数T的大小是0，那么显然Vec即便不申请任何内存，也可以存下任意多的T类型成员。因为不管你往Vec中插入多少数据，总大小依然是0。所以这里的处理逻辑就是，当size_of：：<T>（）==0的时候，cap的取值是usize：：MAX。这里的！0的写法，实际上是对0按位取反。

再回看RawVec的with_capacity方法。它调用了它自己的allocate_in方法。这个方法的实现如下所示：

```
fn allocate_in(cap: usize, zeroed: bool, mut a: A) -> Self {
    unsafe {
        let elem_size = mem::size_of::<T>();

        let alloc_size = cap.checked_mul(elem_size).expect("capacity overflow");
        alloc_guard(alloc_size);

        // handles ZSTs and `cap = 0` alike
        let ptr = if alloc_size == 0 {
            mem::align_of::<T>() as *mut u8
        } else {
            let align = mem::align_of::<T>();
            let result = if zeroed {
                a.alloc_zeroed(Layout::from_size_align(alloc_size, align).unwrap())
            } else {
                a.alloc(Layout::from_size_align(alloc_size, align).unwrap())
            };
            match result {
                Ok(ptr) => ptr,
                Err(err) => a.oom(err),
            }
        };

        RawVec {
            ptr: Unique::new_unchecked(ptr as *mut _),
            cap,
            a,
        }
    }
}
```

首先计算需要分配的内存大小。它使用了checked_mul处理溢出问题。然后考虑0大小的问题。此时无须调用内存分配器的方法，而是直接返回一个数值很小的指针（一般情况下这个值就是1）。为了有利于编译器后端优化，这个指针保证了与T类型字节对齐。此时不是直接用数值0，主要是为了和“空指针”做区分。因为RawVec已经假定了它的成员ptr永远不会是空指针，所以它用了Unique类型。这种设计可以让Option<Vec<T>>拥有和Vec<T>相同的大小，而无须浪费空间。

最后我们再来分析一下RawVec里面的ptr成员。它是Unique<T>这个类型。这个类型的定义如下：

```
pub struct Unique<T: ?Sized> {
    pointer: NonZero<*const T>,
    _marker: PhantomData<T>,
}
```

这个类型是在裸指针基础上做了一点封装。

·它通过PhantomData<T>方式，向编译器表达了“它是T类型对象的拥有者”这样一个概念。PhantomData这个类型是一个0大小的、被编译器特殊对待的类型，它有一个attribute做修饰#[lang="phantom_data"]，凡是被#[lang=…]修饰的东西，都是被编译器特殊处理的，跟普通用户自己定义的不一样。

·因为从逻辑上说这个指针不应该为空，因此它使用NonZero做了一个包装。NonZero这个类型也是一个特殊类型，它也有一个attribute是#[lang="non_zero"]。在编译器内部，会认为这个类型的取值永远不可能为0。这样在某些情况下，编译器可以根据这个信息优化存储空间。比如Option<Box<T>>占据的空间大小跟Box<T>就一模一样，无须额外空间，这里的关键就是Box<T>内部也使用了NonZero这个类型。

·Unique<T>还实现了Send和Sync这两个trait。unsafe impl<T：Send+？Sized>Send for Unique<T>{}unsafe impl<T：Sync+？Sized>Sync for Unique<T>{}

标准库中很多底层的数据结构都需要基于裸指针，使用unsafe代码才能实现。而很多数据结构都需要表达一种“对成员拥有所有权”这样一个概念，因此它们有一些共同的代码可以复用。这就是为什么RawVec是基于Unique类型而不是直接基于裸指针来实现的原因。因为抽象出的这样一个Unique类型，不止在实现动态数组的时候有用，还可以在实现Box<T>HashMap<K，V>等类型的时候有用。

与之对应的，标准库中还有一个叫作Shared<T>的类型。它也是在裸指针基础上做了一点封装。它跟Unique<T>之间的主要区别在于，Shared<T>适合用于表达那种“共享所有权的引用”的情况。比如Rc<T>Arc<T>LinkedList<T>等类型，都是基于Shared<T>实现的。

20.2 内存扩容

接下来我们分析一下Vec：：push这个方法是如何实现的。源码如下：

```
pub fn push(&mut self, value: T) {
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    unsafe {
        let end = self.as_mut_ptr().offset(self.len as isize);
        ptr::write(end, value);
        self.len += 1;
    }
}
```

首先判断当前是否还有空余容量，如果不够，就调用RawVec的double方法；如果足够，直接走下面的逻辑。接下来就是把数据插入Vec的做法。这里直接使用了ptr：：write方法，它做的事情其实就是简单地把数据按位复制到目标位置。请注意，在Rust中这么做是完全正确的，因为它没有“复制构造函数”“移动构造函数”“复制运算符重载”之类的东西，如果我们要把一个对象move到另外一个地方，那就只需要把这个对象按位复制到目的地址即可。当然我们还要防止对象在原来那个地方调用析构函数，恰好ptr：：write方法可以满足这个要求。

接下来继续看一下RawVec：：double方法是怎么实现的：

```
pub fn double(&mut self) {
  unsafe {
      let elem_size = mem::size_of::<T>();

      let (new_cap, uniq) = match self.current_layout() {
          Some(cur) => {
              let new_cap = 2 * self.cap;
              let new_size = new_cap * elem_size;
              let new_layout = Layout::from_size_align_unchecked(new_size, cur. alig
              alloc_guard(new_size);
              let ptr_res = self.a.realloc(self.ptr.as_ptr() as *mut u8,
                                           cur,
                                           new_layout);
              match ptr_res {
                  Ok(ptr) => (new_cap, Unique::new_unchecked(ptr as *mut T)),
                  Err(e) => self.a.oom(e),
              }
          }
```

```
            None => {
                let new_cap = if elem_size > (!0) / 8 { 1 } else { 4 };
                match self.a.alloc_array::<T>(new_cap) {
                    Ok(ptr) => (new_cap, ptr),
                    Err(e) => self.a.oom(e),
                }
            }
        };
        self.ptr = uniq;
        self.cap = new_cap;
    }
}
```

其中RawVec：：current_layout方法的实现如下：

```
fn current_layout(&self) -> Option<Layout> {
    if self.cap == 0 {
        None
    } else {
        unsafe {
            let align = mem::align_of::<T>();
            let size = mem::size_of::<T>() * self.cap;
            Some(Layout::from_size_align_unchecked(size, align))
        }
    }
}
```

由此可见，如果当前capacity是0，即一开始用Vec：：new（）方法初始化的情况下，新的容量一般设置为4，除非这个元素特别大。对于当前capacity不是0的情况，会调用allocator的realloc方法，申请一个两倍于当前大小的空间。

20.3 内存释放

20.3.1 Vec的析构函数

在Rust中，RAII手法是非常常用的资源管理方式。Vec就是利用RAII来进行资源管理的。因此，接下来我们需要分析Vec的析构函数：

```
unsafe impl<#[may_dangle] T> Drop for Vec<T> {
    fn drop(&mut self) {
        unsafe {
            // use drop for [T]
            ptr::drop_in_place(&mut self[..]);
        }
        // RawVec handles deallocation
    }
}
```

目前版本的Vec，在impl Drop trait的时候使用了unsafe关键字，而且使用了#[may_dangle]这个attribute。它是跟drop check相关的，下一节会继续分析。现在继续看这个析构函数的逻辑，它调用了libcore里面的ptr：：drop_in_place函数：

```
#[lang = "drop_in_place"]
#[allow(unconditional_recursion)]
pub unsafe fn drop_in_place<T: ?Sized>(to_drop: *mut T) {
    // Code here does not matter - this is replaced by the
    // real drop glue by the compiler.
    drop_in_place(to_drop);
}
```

而这个函数有#[lang="drop_in_place"]attribute，这说明它是编译器提供的特殊实现，所以它的函数体我们就不再继续深究了，这里写的不是它的真实逻辑。总之它就是告诉编译器，调用这个指针指向对象的析构函数。需要注意的是约束T：？Sized，这意味着这个泛型参数可以是定长类型，也可以是变长类型，即DST。当T是变长类型的时候，这个指针*mut T实际上是一个“胖指针”，这种情况它也是可以处理的。

所以我们看到在Vec的析构函数里面，传递进去的实际参数是一个数组切片slice。编译器会逐个调用这个slice里面每个对象的析构函数。

Vec的析构函数调用完之后，编译器还会自动调用它所有成员的析构函数。我们再看一下RawVec类型的析构函数：

```
unsafe impl<#[may_dangle] T, A: Alloc> Drop for RawVec<T, A> {
    /// Frees the memory owned by the RawVec *without* trying to Drop its contents.
    fn drop(&mut self) {
        unsafe { self.dealloc_buffer(); }
    }
}
```

它做的事情就是回收内存，无须调用析构函数。其中dealloc_buffer函数的实现为：

```
impl<T, A: Alloc> RawVec<T, A> {
    /// Frees the memory owned by the RawVec *without* trying to Drop its contents.
    pub unsafe fn dealloc_buffer(&mut self) {
        let elem_size = mem::size_of::<T>();
        if elem_size != 0 {
            if let Some(layout) = self.current_layout() {
                let ptr = self.ptr() as *mut u8;
                self.a.dealloc(ptr, layout);
            }
        }
    }
}
```

当size_of：：<T>（）是0的时候，根本没有在堆上分配内存，所以无须处理；否则，调用allocator的dealloc函数即可。

20.3.2 Drop Check

下面来详细说明一下什么是drop check。Vec的析构函数中出现的#[may_dangle]究竟是什么呢？

请大家注意，'a：'b这个标记代表的含义是'a比'b长或者相等。什么情况下，它们可以相等呢？当两个变量声明在同一条语句的时候，它们的生命周期是相等的。

也就是说，假如我们按顺序声明两个变量：

```
let a = default();
let b = default();
```

那么a的生命周期一定严格大于b的生命周期。如果我们记录a的生命周期为'a，b的生命周期为'b，那么'a：'b成立，而'b：'a不成立。因此，在a里面引用b一定是行不通的。

但是，假如我们把它们在一条语句中一起声明：

```
let (a, b) = (default(), default());
```

它们的生命周期是相等的。如果我们记录a的生命周期为'a，b的生命周期为'b，那么'a：'b成立，而'b：'a也成立。我们可以用示例来证明：

```
fn main() {
    {
        let (a, mut b) : (i32, Option<&i32>) = (1, None);
        b = Some(&a);
    }
    {
        let (mut a, b) : (Option<&i32>, i32) = (None, 1);
        a = Some(&b);
    }
}
```

上面的代码可以编译通过，正是说明了以上的状况。

Rust之所以这么规定，是因为在同一条语句中声明出来的不同变量绑定，无法根据先后关系确定出哪个严格包含于哪个。tuple里面的两个成员的生命周期不存在严格大于和小于关系，struct里面不同成员的生命周期也不存在严格大于和小于关系，数组里面不同成员的生命周期同样不存在严格大于和小于关系。它们的生命周期都是相等的。

这就引出了一个问题。两个不同的变量在析构的时候，总会出现一个先一个后，不可能同时析构。如果同一条语句中声明的不同变量可以存在引用关系，那么如果我们在析构函数中，试图访问另外一个变量，会出现什么情况呢？我们写一个示例：

```
struct T { dropped: bool }

impl T {
```

```
    fn new() -> Self {
        T { dropped: false }
    }
}

impl Drop for T {
    fn drop(&mut self) {
        self.dropped = true;
    }
}

struct R<'a> {
    inner: Option<&'a T>
}

impl<'a> R<'a> {
    fn new() -> Self {
        R { inner: None }
    }
    fn set_ref<'b :'a>(&mut self, ptr: &'b T) {
        self.inner = Some(ptr);
    }
}

impl<'a> Drop for R<'a> {
    fn drop(&mut self) {
        if let Some(ref inner) = self.inner {
            println!("droppen R when T is {}", inner.dropped);
        }
    }
}

fn main() {
    {
        let (a, mut b) : (T, R) = (T::new(), R::new());
        b.set_ref(&a);
    }
    {
        let (mut a, b) : (R, T) = (R::new(), T::new());
        a.set_ref(&b);
    }
}
```

这个示例保持了上个示例的代码结构，只是把基本类型替换成了带有析构函数的自定义类型。编译之后出现了生命周期编译错误：

```
error[E0597]: `a` does not live long enough
```

这样看来，我们原来想象的，在析构函数中访问相同生命周期的变量，制造内存不安全的想法是行不通的。

为什么前面的代码使用基本i32和&i32类型可以编译通过，而我们

换成自定义类型就通不过了呢？这就是所谓的drop checker。Rust在涉及析构函数的时候有个特殊规定，即如果两个变量具有析构函数，且有互相引用的关系，那么它们的生命周期必须满足“严格大于”的关系。这个关系目前在源码中表达不出来，但是为了防止析构函数中出现内存安全问题，编译器内部对此专门做了检查。

但是这种检查又有点过于严格。因为在很多情况下，虽然它们有引用关系，但是并没有在析构函数中做数据访问。此事取决于析构函数具体做了什么。如果析构函数没有做什么危险的事情，那么它们之间的生命周期满足普通的大于等于关系就够了。所以设计者决定，暂时留一个后门，让用户告诉编译器这个析构函数究竟危险还是不危险，这就是#[may_dangle]attribute的由来。在上例中，我们如果把R类型的析构函数改为：

```
unsafe impl<#[may_dangle] 'a> Drop for R<'a> {
    fn drop(&mut self) {
    }
}
```

再打开相应的feature gate：

```
#![feature(generic_param_attrs, dropck_eyepatch)]
```

以上代码就可以编译通过，生命周期冲突问题就消失了。这就是为什么Vec<T>的析构函数用了#[may_dangle]的原因。加了这个attribute可以让Vec类型容纳生命周期不满足“严格大于”关系的元素。

关于此事的详细解释，请参考RFC-1327-dropck-param-eyepatch。这个功能也只是临时措施，关于drop check的部分，后面还会有改进。

20.4 不安全的边界

Vec有一个成员方法叫作set_len，可以用于改变动态数组的大小。它的源码如下：

```
pub unsafe fn set_len(&mut self, len: usize) {
    self.len = len;
}
```

关于这个方法，需要请大家注意的是，它有unsafe标记。它的内部只是一个usize类型的赋值而已，怎么会是unsafe呢？

因为，我们的Vec内部实现非常依赖于self.len这个值的合法性。如果说这个方法不是unsafe，外部的使用者可以随意设置动态数组的大小，那么用户可以将其大小突然变很大，然后就可以通过这个Vec访问本不该属于它的内容，这就造成了“内存不安全”的情况。

所以，我们一定要注意的是：判断一个函数是否应该是一个unsafe函数，不该看它表面的逻辑，而应该判断用户使用它的时候造成的影响。

当我们使用unsafe代码块的时候，很可能需要一些对safe代码的隐含的假设和依赖，这些依赖关系既不能通过类型系统向编译器清楚表达，也未必能在代码中明显地表现出来。safe代码有义务维持unsafe代码相对应的假设，unsafe代码中也要注意保持一致性。如果这些假设一旦被破坏，那这个库的安全性也就功亏一篑了。只要你在某个函数内部使用了unsafe代码块，你需要关注的就不只是这个函数的正确性，还有这个类型中，甚至是这个模块中，其他所有函数的正确性，它们是互相影响互相搭配的。

对于Vec类型，我们需要保证任何时候self.len这个成员都应该准确地反映它内部的成员个数。这个要求，我们无法利用类型系统或者别的什么语言特性表达出来。因为这个成员赋值，可能是安全的，也可能是不安全的，取决于上下文逻辑。所以，这个方法必须用unsafe标记。而用户在使用的时候，如果已经在逻辑上保证了这个赋值是安全的，那么就可以在那个地方利用unsafe代码块调用这个方法，否则就不该调用。

再举个例子，我们使用unsafe代码来访问数组内部的数据：

```
fn index(arr: &[u8], idx: usize) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            let p = arr.as_ptr().offset(idx as isize);
            Some(*p)
        }
    } else {
        None
    }
}

fn main() {
    let arr = [1,2,3,4,5];
    println!("{:?}", index(&arr, 3));
}
```

基本逻辑很简单，通过裸指针的算术运算，指向我们需要的目标，然后将数据读出来。这段代码将不安全的内部实现和安全的外部API良好地结合在了一起，是符合Rust的设计思路的。

在这个例子中，如果我们把if条件稍作改动，变成：

```
if idx <= arr.len()
```

那么这个函数就变成了“不安全”的代码，外部用户有机会通过安全代码读取不属于这个数组的内容，而且编译器检查不出来。需要注意的是，我们这里只修改了安全代码，但它制造了不安全现象。这是因为我们内部的unsafe语句块中，已经假定了idx的数值是合法的。我们在unsafe块的外部，就需要通过逻辑来保证这一点，否则就是bug。

所以，基于unsafe代码写库很难，难就难在你不仅要测试正常情况下功能的正确性，还必须考虑用户可能的各种行为是否有可能在safe代码中利用你这个库制造内存不安全。在C/C++中，如果用户可以通过一个库制造内存不安全，那是用户的问题，作为库只需要提供功能就足够了，无须保证安全性，当然你也保证不了，顶多也就“防君子不防小人”。跟C/C++相比，Rust提供的保证要严格得多，如果用户有机会利用你的库在不使用unsafe关键字的情况下制造出“内存不安全”，那这个库就有严重bug，是低质量的、不可接受的。

20.5 自定义解引用

继续分析Vec的源码。Vec是一个“动态数组”，它的行为应该尽量与默认的定长数组一致。语言内置的定长数组支持数组切片功能，可以使用一个Slice作为指向数组某个部分的“视图”。那我们也应该为Vec实现这样的功能。这种情况我们需要利用Deref，代码如下：

```
impl<T> ops::Deref for Vec<T> {
    type Target = [T];

    fn deref(&self) -> &[T] {
        unsafe {
            let p = self.buf.ptr();
            assume(!p.is_null());
            slice::from_raw_parts(p, self.len)
        }
    }
}
```

Target类型是[T]，这意味着*Vec<T>的类型为[T]，所以&*Vec<T>的类型为&[T]。

当碰到可以“隐式自动deref”的场景时，&Vec<T>类型如果不匹配，编译器就会继续尝试&*Vec<T>，即&[T]类型来匹配。所以，我们就可以在需要&[T]类型的时候，直接使用&Vec<T>。

Deref在许多时候都很有用。因为Deref的存在，实现vec[..]这样的功能很简单，因为编译器会帮我们实现类型自动转换：

```
impl<T> ops::Index<ops::RangeFull> for Vec<T> {
    type Output = [T];

    #[inline]
    fn index(&self, _index: ops::RangeFull) -> &[T] {
        self
    }
}
```

我们知道self的类型是&Vec<T>，而函数定义的返回类型是&[T]，因为有Deref的存在，编译器会帮我们做这个自动类型转换。

再比如索引Index功能，其内部实现方式就是先deref为原生数组类型，然后利用内置数组的Index功能实现：

```
impl<T> Index<usize> for Vec<T> {
    type Output = T;

    #[inline]
    fn index(&self, index: usize) -> &T {
        // NB built-in indexing via `&[T]`
        &(**self)[index]
    }
}
```

读者看到（**self）的时候不要惊慌，我们慢慢分析。self类型是&Vec<T>，因此*self类型是Vec<T>，**self类型是[T]。因此这句话的意思是：对[T]执行Index操作后，再把引用返回。

20.6 迭代器

我们知道，可以通过Vec：：iter（）方法创建一个动态数组的迭代器。但是我们在源码中却没有见到这个方法的存在。这是因为这个方法实际上是slice类型的方法，Vec只是自动deref后调用了原生数组的迭代器而已。

但是，Vec类型本身也是可以用于for循环中的：

```
fn main() {
    let x = vec![0_i32, 1, 2];
    for item in x { println!("{}", item); }
}
```

这是因为Vec实现了IntoIterator trait。标准库中的IntoIterator就是编译器留下来的一个扩展内置for循环语法的接口。任何自定义类型，只要合理地实现了这个trait，就可以被用在内置的for循环里面。关于迭代器的更多内容，在本书第三部分继续讲述。

关于迭代器，有一个Vec：：drain方法实现得比较特殊，这里专门拿出来讲一下。它的功能是从动态数组中把一个范围的数据“移除”出去，返回的还是一个“迭代器”。我们还可以遍历一次这个迭代器，使用已经被移除的那些元素。示例如下：

```
fn main() {
    let mut origin = vec![0, 1, 2, 3, 4, 5];

    println!("Removed:");
    for i in origin.drain(1..3) {
        println!("{}", i);
    }

    println!("Left:");
    for i in origin.iter() {
        println!("{}", i);
    }
}
```

drain（）方法返回的类型就是一个普通的迭代器，在标准库中，这个方法的源码如下所示：

```
pub fn drain<R>(&mut self, range: R) -> Drain<T>
        where R: RangeArgument<usize>
    {
        let len = self.len();
        let start = match range.start() {
            Included(&n) => n,
            Excluded(&n) => n + 1,
            Unbounded    => 0,
        };
        let end = match range.end() {
            Included(&n) => n + 1,
            Excluded(&n) => n,
            Unbounded    => len,
        };
        assert!(start <= end);
        assert!(end <= len);

        unsafe {
            self.set_len(start);
            let range_slice = slice::from_raw_parts_mut(self.as_mut_ptr().offset (st

            Drain {
                tail_start: end,
                tail_len: len - end,
                iter: range_slice.iter(),
                vec: Shared::from(self),
            }
        }
    }
```

返回的这个Drain类型实现了Iterator trait，具体源码就不详细列出了。总之遍历Drain这个迭代器，会把所有应该被删除的元素遍历一遍。而Drain类型还实现了一个析构函数。当它自己被销毁的时候，它会去修改原始的Vec的内容，把这些应该被删除的元素从原始数组中真正删掉。

大致原理就是这样。特别需要注意的是，在Vec：：drain方法中创建迭代器之前，先调用了self.set_len（start）方法。那么这个设置的目的是什么呢？

这个设计实际上是为了防止另一种情况下的内存不安全。

我们假设用户写了这样的代码：

```
fn main() {
    let mut origin = vec![
        "0".to_string(), "1".to_string(), "2".to_string(),
        "3".to_string(), "4".to_string(), "5".to_string()];
    {
```

```
            let mut d = origin.drain(1..3);
            let s: Option<String> = d.next();
            println!("{:?}", s);
            let s: Option<String> = d.next();
            println!("{:?}", s);
            let s: Option<String> = d.next();
            println!("{:?}", s);
            std::mem::forget(d);
        }

        println!("Left:");
        for i in origin.iter() {
            println!("{:?}", i);
        }
    }
```

前面讲泄露的时候已经说过了，std：：mem：：forget函数是不带unsafe修饰的。它可以阻止一个类型的析构函数调用，析构函数是不能保证一定会被调用的。在这种情况下，Drain类型没有机会执行它的析构函数，所以它没有机会修改原始的Vec，把数据从Vec中移除。

假设没有self.set_len（start）；这个函数调用，在上面的例子中会出现某些字符串元素已经被Drain迭代器取出来消费掉了，但是Vec中还存有一份“副本”，而这个副本本身处于一种“未初始化状态”，它们从逻辑上已经被移走了，但依然被认为是Vec的正常数据。这是典型的内存不安全的情况。

所以，在标准库中，drain（）方法内部在返回迭代器之前，先把当前Vec的大小设置为一个比较小的绝对安全的值。如果说这个drain（）方法返回的迭代器因为某种原因未能成功析构，那么最坏的结果也就是，原数组中仅剩下了start之前的元素。至少我们可以肯定，任何情况下，数组中的数据都是符合“内存安全”的。如果这个迭代器的析构函数成功执行了，那么end之后的元素会向前移动，数组的长度会被重置，这时候这个逻辑才算执行完整。

析构函数泄漏绝对不是我们期望发生的事情，我们只是无法阻止这种情况而已。所以，在写库的时候要注意，我们的底线是，即便析构函数泄漏会导致逻辑错误，也不会发生“内存不安全”。

20.7 panic safety

在利用unsafe写库的时候，还需要注意的一点是“panic安全”。panic在什么情况下发生是难以预测的，我们要做的是，即便在发生panic的时候，也能保证“内存安全”。

我们以Vec：：truncate方法为例来说明“panic安全”是怎么做到的。这个方法用于把数组切掉一部分，只保留前面的部分，后面的部分扔掉。所以，我们可以想到的实现逻辑应该是针对被切掉的部分，每个元素调用一下析构函数，最后重新设置一下数组的长度大小即可。

可惜这么做是不对的。因为“对象的析构函数”是用户自定义的行为，在这个方法里面会执行什么逻辑是无法提前确定的。所以，我们应该假设它有可能发生panic。如果有对象已经执行了析构，但是还继续把它留在数组里面，等待数组最后来重新设置长度，是有风险的。所以标准库里面的代码是这么做的：每次执行析构前先把数组长度减1，从逻辑上将元素从数组中移除，然后执行析构函数。源码如下所示：

```
pub fn truncate(&mut self, len: usize) {
    unsafe {
        // drop any extra elements
        while len < self.len {
            // decrement len before the drop_in_place(), so a panic on Drop
            // doesn't re-drop the just-failed value.
            self.len -= 1;
            let len = self.len;
            ptr::drop_in_place(self.get_unchecked_mut(len));
        }
    }
}
```

所以，大家在读源码的时候，不仅要看到别人是这样写的，更要看到别人为什么不会那样写。这段代码看起来好像不够优化，在每次循环的时候减1，却没有在循环前或者后面一次性减掉len。这样做是有原因的。请读者仔细理解源码中的那条注释。同样的道理，类似的保障异常安全的设计，我们还可以在extend_with等方法中看到。这个方法是实现resize、resize_default等方法的基础。

```
impl<T> Vec<T> {
```

```
    /// Extend the vector by `n` values, using the given generator.
    fn extend_with<E: ExtendWith<T>>(&mut self, n: usize, value: E) {
        self.reserve(n);
        unsafe {
            let mut ptr = self.as_mut_ptr().offset(self.len() as isize);
            // Use SetLenOnDrop to work around bug where compiler
            // may not realize the store through `ptr` through self.set_len()
            // don't alias.
            let mut local_len = SetLenOnDrop::new(&mut self.len);

            // Write all elements except the last one
            for _ in 1..n {
                ptr::write(ptr, value.next());
                ptr = ptr.offset(1);
                // Increment the length in every step in case next() panics
                local_len.increment_len(1);
            }

            if n > 0 {
                // We can write the last element directly without cloning needlessly
                ptr::write(ptr, value.last());
                local_len.increment_len(1);
            }

            // len set by scope guard
        }
    }
}
```

这个方法是往Vec后面继续扩展n个元素，这n个元素可以是通过一个元素clone（）而来，也可以是调用Default：：default（）构造而来。

大家可以注意到，在真正写入数据之前，先创建了一个SetLenOnDrop类型的变量local_len，另外每次写入一个新的元素之后，都会将这个变量重新设置长度。而这个SetLenOnDrop类型的主要功能，就是在析构的时候修改Vec的真正长度。因为在这段unsafe代码中，需要调用value.next（）方法，而这个方法最后会调用到元素的T：：default（）或者T：：clone（）方法。这些方法的实现，取决于外部元素类型的实现，它们会不会导致panic，写容器的作者是不知道的。因此，Vec容器的作者只能假定这些外部方法是有panic风险的。为了保证在发生panic之后Vec内部包含的依然是合法数据，一定要每次成功写入一个元素之后，马上更新长度信息。

第三部分　高级抽象

Rust既有面向硬件、面向底层、执行效率高的特点，也有面向封装、面向抽象、表达能力强的特点。本部分主要讲解Rust提供的一系列高级抽象工具。

第21章　泛型

泛型（Generics）是指把类型抽象成一种“参数”，数据和算法都针对这种抽象的类型参数来实现，而不针对具体类型。当我们需要真正使用的时候，再具体化、实例化类型参数。

21.1 数据结构中的泛型

有时候，我们需要针对多种类型进行统一的抽象，这就是泛型。泛型可以将“类型”作为参数，在函数或者数据结构中使用。

再以我们熟悉的Option类型举例。它就是一个泛型enum类型，其参数声明在尖括号<>中。

```
enum Option<T> {
    Some(T),
    None,
}
```

这里的<T>实际上是声明了一个“类型”参数。在这个Option内部，Some（T）是一个tuple struct，包含一个元素类型为T。这个泛型参数类型T，可以在使用时指定具体类型。

```
let x: Option<i32> = Some(42);
let y: Option<f64> = None;
```

在上述第一行代码中，泛型参数T被具体化成了i32，x的类型，在这里是Option<i32>；在第二行代码中，泛型参数T被具体化成了f64，y的类型，在这里是Option<f64>。

泛型参数可以有多个也可以有默认值。比如：

```
struct S<T=i32> {
    data: T
}

fn main() {
    let v1 = S { data: 0};
    let v2 = S::<bool> { data: false};
    println!("{} {}", v1.data, v2.data);
}
```

对于上例中的泛型参数T，我们可以在使用的时候不指定类型参数。如果不指定的话，参数默认为i32，也可以在使用的时候指定为其

他类型。

使用不同类型参数将泛型类型具体化后，获得的是完全不同的具体类型。如Option<i32>和Option<i64>是完全不同的类型，不可通用，也不可相互转换。当编译器生成代码的时候，它会为每一个不同的泛型参数生成不同的代码。

各种自定义复合类型都可以携带任意的泛型参数。Rust规定，所有的泛型参数必须是真的被使用过的。下面的代码就会报错：

```
struct Num<T> {
    data: i32
}
```

这个结构体声明了一个泛型参数，但是却并没有使用它。编译器会对这个问题报错，说有泛型参数从来没有使用过。按下面这样写就没有问题了：

```
struct Num<T> {
    data: Option<T>
}
```

21.2 函数中的泛型

泛型可以使用在函数中，语法类似：

```
fn compare_option<T>(first: Option<T>, second: Option<T>) -> bool
{
    match(first, second) {
        (Some(..), Some(..)) => true,
        (None, None) => true,
        _ => false
    }
}
```

在上面这个例子中，函数compare_option有一个泛型参数T，两个形参类型均为Option<T>。这意味着这两个参数必须是完全一致的类型。如果我们在参数中传入两个不同的Option，会导致编译错误：

```
fn main() {
    println!("{}", compare_option(Some(1i32), Some(1.0f32))); // 类型不匹配编译错误
}
```

编译器在看到这个函数调用的时候，会进行类型检查：first的形参类型是Option<T>、实参类型是Option<i32>，second的形参类型是Option<T>、实参类型是Option<f32>。这时编译器的类型推导功能会进行一个类似解方程组的操作：由Option<T>==Option<i32>可得T==i32，而由Option<T>==Option<f32>又可得T==f32。这两个结论产生了矛盾，因此该方程组无解，出现编译错误。

如果我们希望参数可以接受两个不同的类型，那么需要使用两个泛型参数：

```
fn compare_option<T1, T2>(first: Option<T1>, second: Option<T2>) -> bool { ... }
```

一般情况下，调用泛型函数可以不指定泛型参数类型，编译器可以通过类型推导自动判断。某些时候，如果确实需要手动指定泛型参数类型，则需要使用function_name：：<type params>（function params）的语法：

```
compare_option::<i32, f32>(Some(1), Some(1.0));
```

泛型函数在很大程度上实现了C++的“函数重载”功能。比如，str类型有一个contains方法，使用示例如下：

```
fn main() {
    let s = "hello";
    println!("{}", s.contains('a'));
    println!("{}", s.contains("abc"));
    println!("{}", s.contains(&['H'] as &[char]));
    println!("{}", s.contains(|c : char| c.len_utf8() > 2));
}
```

我们可以看到，这个contains方法可以接受很多种不同的参数类型，使用起来很方便。那么它是怎么实现的呢？主要技术就是泛型。它的签名如下：

```
fn contains<'a, P: Pattern<'a>>(&'a self, pat: P) -> bool
```

可见，它的第二个参数不是某个具体类型，而是一个泛型类型，而且这个泛型参数满足Pattern trait的约束。这意味着，所有实现了Pattern trait的类型，都可以作为参数使用。我们希望这个参数接受哪些类型，就针对这个类型实现这个trait即可。

Rust没有C++那种无限制的ad hoc式的函数重载功能。现在没有，将来也不会有。主要原因是，这种随意的函数重载对于代码的维护和可读性是一种伤害。通过泛型来实现类似的功能是更好的选择。如果说，不同的参数类型，没有办法用trait统一起来，利用一个函数体来统一实现功能，那么它们就没必要共用同一个函数名。它们的区别已经足够大，所以理应使用不同的名字。强行使用同一个函数名来表示区别非常大的不同函数逻辑，并不是好的设计。

我们还有另外一种方案，可以把不同的类型统一起来，那就是enum。通过enum的不同成员来携带不同的类型信息，也可以做到类似“函数重载”的功能。但这种做法跟“函数重载”有本质区别，因为它是有运行时开销的。enum会在执行阶段判断当前成员是哪个变体，而“函数重载”以及泛型函数都是在编译阶段静态分派的。同样，Rust也不鼓

励大家仅仅为了省去命名的麻烦，而强行把不同类型用enum统一起来用一个函数来实现。如果一定要这么做，那么最好是有一个好的理由，而不仅是因为懒得给函数命名而已。

21.3 impl块中的泛型

impl的时候也可以使用泛型。在impl<Trait>for<Type>{}这个语法结构中，泛型类型既可以出现在<Trait>位置，也可以出现在<Type>位置。

与其他地方的泛型一样，impl块中的泛型也是先声明再使用。在impl块中出现的泛型参数，需要在impl关键字后面用尖括号声明。

当我们希望为某一组类型统一impl某个trait的时候，泛型就非常有用了。有了这个功能，很多时候就没必要单独为每个类型去重复impl了。以标准库中的代码为例：

```
impl<T, U> Into<U> for T
    where U: From<T>
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

上面这段代码中，impl关键字后面的尖括号<T，U>意思是先声明两个泛型参数，后面会使用它们。这跟类型、函数中的泛型参数规则一样，先声明、后使用。

标准库中的Into和From是一对功能互逆的trait。如果A：Into<B>，意味着B：From<A>。因此，标准库中写了这样一段代码，意思是针对所有类型T，只要满足U：From<T>，那么就针对此类型impl Into<U>。有了这样一个impl块之后，我们如果想为自己的两个类型提供互相转换的功能，那么只需impl From这一个trait就可以了，因为反过来的Into trait标准库已经帮忙实现好了。

21.4 泛型参数约束

Rust的泛型和C++的template非常相似，但也有很大不同。它们的最大区别在于执行类型检查的时机。在C++里面，模板的类型检查是延迟到实例化的时候做的。而在Rust里面，泛型的类型检查是当场完成的。示例如下：

```
// C++ template 示例
template <class T>
const T& max (const T& a, const T& b) {
    return (a<b)?b:a;
}

void instantiateInt() {
    int m = max(1, 2);    // 实例化1
}

struct T {
    int value;
};

void instantiateT() {
    T t1 { value: 1};
    T t2 { value: 2};
    //T m = max(t1, t2);  // 实例化2
}

int main() {

    instantiateInt();
    instantiateT();
    return 0;
}
```

在这个例子中：如果我们把“实例化2”处的代码先注释掉，使用g++-std=c++11 test.cpp命令编译，可以通过；如果取消注释，则编译不通过。编译错误为：

```
error: no match for 'operator<' (operand types are 'const T' and 'const T')
```

出现编译错误的原因是我们没有给T类型提供比较运算符重载。此处的关键在于，如果我们用int类型来实例化max函数，它就可以通过；如果我们用自定义的T类型来实例化max函数，它就通不过。max函数本

身一直都是没有问题的。也就是说，编译器在处理max函数的时候，根本不去管a<b是不是一个合理的运算，而是将这个检查留给后面实例化的时候再分析。

Rust采取了不同的策略，它会在分析泛型函数的时候当场检查类型的合法性。这个检查是怎样做的呢？它要求用户提供合理的“泛型约束”。在Rust中，trait可以用于作为“泛型约束”。在这一点上，Rust跟C#的设计是类似的。上例用Rust来写，大致是这样的逻辑：

```
fn max<T>(a: T, b: T) -> T {
    if a < b {
        b
    } else {
        a
    }
}

fn main() {
    let m = max(1, 2);
}
```

编译，出现编译错误：

```
error[E0369]: binary operation `<` cannot be applied to type `T`
--> test.rs:2:8
    |
  2 |     if a < b {
    |        ^^^^^
    |
    = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

这个编译错误说得很清楚了，由于泛型参数T没有任何约束，因此编译器认为a<b这个表达式是不合理的，因为它只能作用于支持比较运算符的类型。在Rust中，只有impl了PartialOrd的类型，才能支持比较运算符。修复方案为泛型类型T添加泛型约束。

泛型参数约束有两种语法：

（1）在泛型参数声明的时候使用冒号：指定；

（2）使用where子句指定。

```
use std::cmp::PartialOrd;

// 第一种写法：在泛型参数后面用冒号约束
fn max<T: PartialOrd>(a: T, b: T) -> T {

// 第二种写法,在后面单独用 where 子句指定
fn max<T>(a: T, b: T) -> T
    where T: PartialOrd
```

在上面的示例中，这两种写法达到的目的是一样的。但是，在某些情况下（比如存在下文要讲解的关联类型的时候），where子句比参数声明中的冒号约束具有更强的表达能力，但它在泛型参数列表中是无法表达的。我们以Iterator trait中的函数为例：

```
trait Iterator {
    type Item; // Item 是一个关联类型

    // 此处的where子句没办法在声明泛型参数的时候写出来
    fn max(self) -> Option<Self::Item>
        where Self: Sized, Self::Item: Ord
    {
        ...
    }
    ...
}
```

它要求Self类型满足Sized约束，同时关联类型Self：：Item要满足Ord约束，这是用冒号语法写不出来的。在声明的时候使用冒号约束的地方，一定都能换作where子句来写，但是反过来不成立。另外，对于比较复杂的约束条件，where子句的可读性明显更好。

在有了“泛型约束”之后，编译器不仅会在声明泛型的地方做类型检查，还会在实例化泛型的地方做类型检查。接上例，如果向我们上面实现的那个max函数传递自定义类型参数：

```
struct T {
    value: i32
}

fn main() {
    let t1 = T { value: 1};
    let t2 = T { value: 2};
    let m = max(t1, t2);
}
```

编译，可见编译错误：

```
error[E0277]: the trait bound `T: std::cmp::PartialOrd` is not satisfied
    --> test.rs:20:13
     |
  20 |     let m = max(t1, t2);
     |             ^^^ can't compare `T` with `T`
     |
     = help: the trait `std::cmp::PartialOrd` is not implemented for `T`
     = note: required by `max`
```

这说明，我们在调用max函数的时候也要让参数符合“泛型约束”。因此我们需要impl PartialOrd for T。完整代码如下：

```
use std::cmp::PartialOrd;
use std::cmp::Ordering;

fn max<T>(a: T, b: T) -> T
    where T: PartialOrd
{
    if a < b {
        b
    } else {
        a
    }
}

struct T {
    value: i32
}

impl PartialOrd for T {
    fn partial_cmp(&self, other: &T) -> Option<Ordering> {
        self.value.partial_cmp(&other.value)
    }
}

impl PartialEq for T {
    fn eq(&self, other: &T) -> bool {
        self.value == other.value
    }
}

fn main() {
    let t1 = T { value: 1};
    let t2 = T { value: 2};
    let m = max(t1, t2);
}
```

由于标准库中的PartialOrd继承了PartialEq，因此单独实现PartialOrd还是会产生编译错误，必须同时实现PartialEq才能编译通过。

21.5 关联类型

trait中不仅可以包含方法（包括静态方法）、常量，还可以包含“类型”。比如，我们常见的迭代器Iterator这个trait，它里面就有一个类型叫Item。其源码如下：

```
pub trait Iterator {
    type Item;
    ...
}
```

这样在trait中声明的类型叫作“关联类型”（associated type）。关联类型也同样是这个trait的“泛型参数”。只有指定了所有的泛型参数和关联类型，这个trait才能真正地具体化。示例如下（在泛型函数中，使用Iterator泛型作为泛型约束）：

```
use std::iter::Iterator;
use std::fmt::Debug;

fn use_iter<ITEM, ITER>(mut iter: ITER)
    where ITER: Iterator<Item=ITEM>,
            ITEM: Debug
{
    while let Some(i) = iter.next() {
        println!("{:?}", i);
    }
}

fn main() {
    let v: Vec<i32> = vec![1,2,3,4,5];
    use_iter(v.iter());
}
```

可以看到，我们希望参数是一个泛型迭代器，可以在约束条件中写Iterator<Item=ITEM>。跟普通泛型参数比起来，关联类型参数必须使用名字赋值的方式。那么，关联类型跟普通泛型参数有哪些不同点呢？我们为什么需要关联参数呢？

1.可读性可扩展性

从上面这个例子中我们可以看到，虽然我们的函数只接受一个参数

iter，但是它却需要两个泛型参数：一个用于表示迭代器本身的类型，一个用于表示迭代器中包含的元素类型。这是相对冗余的写法。实际上，在有关联类型的情况下，我们可以将上面的代码简化，示例如下：

```
use std::iter::Iterator;
use std::fmt::Debug;
fn use_iter<ITER>(mut iter: ITER)
    where ITER: Iterator,
            ITER::Item: Debug
{
    while let Some(i) = iter.next() {
        println!("{:?}", i);
    }
}

fn main() {
    let v: Vec<i32> = vec![1,2,3,4,5];
    use_iter(v.iter());
}
```

这个版本的写法相对于上一个版本来说，泛型参数明显简化了，我们只需要一个泛型参数即可。在泛型约束条件中，可以写上ITER符合Iterator约束。此时，我们就已经知道ITER存在一个关联类型Item，可以针对这个ITER：：Item再加一个约束即可。如果我们的Iterator中的Item类型不是关联类型，而是普通泛型参数，就没办法进行这样的简化了。

我们再看另一个例子。假如我们想设计一个泛型的“图”类型，它包含“顶点”和“边”两个泛型参数，如果我们把它们作为普通的泛型参数设计，那么看起来就是：

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    ...
}
```

现在如果有一个泛型函数，要计算一个图中两个顶点的距离，它的签名会是：

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> uint {
    ...
}
```

我们可以看到，泛型参数比较多，也比较麻烦。对于指定的Graph类型，它的顶点和边的类型应该是固定的。在函数签名中再写一遍其实没什么道理。如果我们把普通的泛型参数改为“关联类型”设计，那么数据结构就成了：

```
trait Graph {
    type N;
    type E;
    fn has_edge(&self, &N, &N) -> bool;
    ...
}
```

对应的，计算距离的函数签名可以简化成：

```
fn distance<G>(graph: &G, start: &G::N, end: &G::N) -> uint
    where G: Graph
{
    ...
}
```

由此可见，在某些情况下，关联类型比普通泛型参数更具可读性。

2.trait的impl匹配规则

泛型的类型参数，既可以写在尖括号里面的参数列表中，也可以写在trait内部的关联类型中。这两种写法有什么区别呢？我们用一个示例来演示一下。

假如我们要设计一个trait，名字叫作ConvertTo，用于类型转换。那么，我们就有两种选择。一种是使用泛型类型参数：

```
trait ConvertTo<T> {
    fn convert(&self) -> T;
}
```

另一种是使用关联类型：

```
trait ConvertTo {
    type DEST;
    fn convert(&self) -> Self::DEST;
}
```

如果我们想写一个从i32类型到f32类型的转换，在这两种设计下，代码分别是：

```
impl ConvertTo<f32> for i32 {
    fn convert(&self) -> f32 { *self as f32 }
}
```

以及：

```
impl ConvertTo for i32 {
    type DEST = f32;
    fn convert(&self) -> f32 { *self as f32 }
}
```

到目前为止，这两种设计似乎没什么区别。但是，假如我们想继续增加一种从i32类型到f64类型的转换，使用泛型参数来实现的话，可以编译通过：

```
impl ConvertTo<f64> for i32 {
    fn convert(&self) -> f64 { *self as f64 }
}
```

如果用关联类型来实现的话，就不能通过编译了：

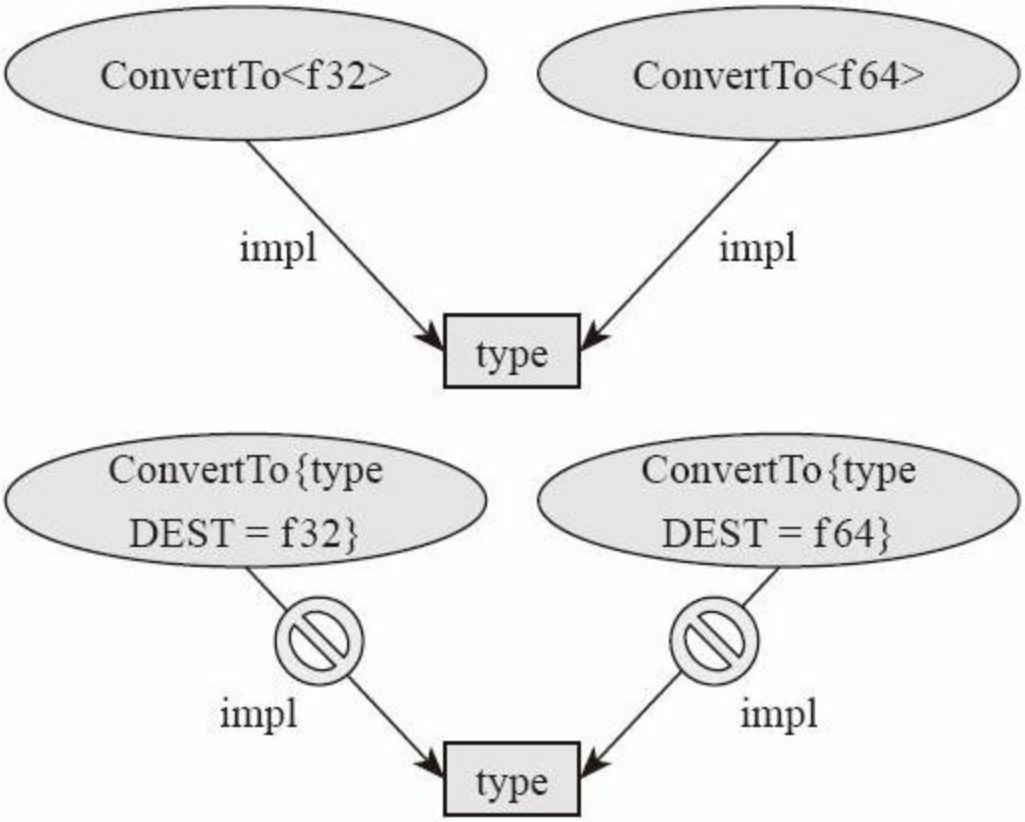
```
impl ConvertTo for i32 {
    type DEST = f64;
    fn convert(&self) -> f64 { *self as f64 }
}
```

错误信息为：

```
error: conflicting implementations of trait `ConvertTo` for type `i32`
```

由此可见，如果我们采用了“关联类型”的设计方案，就不能针对这个类型实现多个impl。在编译器的眼里，如果trait有类型参数，那么给定不同的类型参数，它们就已经是不同的trait，可以同时针对同一个类

型实现impl。如果trait没有类型参数，只有关联类型，给关联类型指定不同的类型参数是不能用它们针对同一个类型实现impl的。



21.6 何时使用关联类型

从前文中大家可以看到，虽然关联类型也是类型参数的一种，但它与泛型类型参数列表是不同的。我们可以把这两种泛型类型参数分为两个类别：

·输入类型参数

·输出类型参数

在尖括号中存在的泛型参数，是输入类型参数；在trait内部存在的关联类型，是输出类型参数。输入类型参数是用于决定匹配哪个impl版本的参数；输出类型参数则是可以由输入类型参数和Self类型决定的类型参数。

继续以上面的例子为例，用泛型参数实现的版本如下：

```
trait ConvertTo<T> {
    fn convert(&self) -> T;
}

impl ConvertTo<f32> for i32 {
    fn convert(&self) -> f32 { *self as f32 }
}

impl ConvertTo<f64> for i32 {
    fn convert(&self) -> f64 { *self as f64 }
}

fn main() {
    let i = 1_i32;
    let f = i.convert();
    println!("{:?}", f);
}
```

编译的时候，编译器会报错：

```
error: unable to infer enough type information about `_`; type annotations or generi
```

因为编译器不知道选择使用哪种convert方法，需要我们为它指定一

个类型参数，比如：

```
let f : f32 = i.convert();
// 或者
let f = ConvertTo::<f32>::convert(&i);
```

这很像C++/Java等语言中存在的“函数重载”规则。我们可以用不同的参数类型实现重载，但是不能用不同的返回类型来做重载，因为编译器是根据参数类型来判断调用哪个版本的重载函数的，而不是依靠返回值的类型。

在标准库中，何时使用泛型参数列表、何时使用关联类型，实际上有非常好的示范。

以标准库中的AsRef为例。我们希望String类型能实现这个trait，而且既能实现String：：as_ref：：<str>（）也能实现String：：as_ref：：<[u8]>（）。因此AsRef必须有一个类型参数，而不是关联类型。这样impl AsRef<str>for String和impl AsRef<[u8]>for String才能同时存在，互不冲突。如果我们把目标类型设计为关联类型，那么针对任何一个类型，最多只能impl一次，这就失去AsRef的意义了。

我们再看标准库中的Deref trait。我们希望一个类型实现Deref的时候，最多只能impl一次，解引用的目标类型是唯一固定的，不要让用户在调用obj.deref（）方法的时候指定返回类型。因此Deref的目标类型应该设计为“关联类型”。否则，我们可以为一个类型实现多次Deref，比如impl Deref<str>for String和impl Deref<char>for String，那么针对String类型做解引用操作，在不同场景下可以有不同的结果，这显然不是我们希望看到的。解引用的目标类型应该由Self类型唯一确定，不应该在被调用的时候被其他类型干扰。这种时候应使用关联类型，而不是类型参数。关联类型是在impl阶段确定下来的，而不是在函数调用阶段。这样才是最符合我们需求的写法。

还有一些情况下，我们既需要类型参数，也需要关联类型。比如标准库中的各种运算符相关的trait。以加法运算符为例，它对应的trait为std：：ops：：Add，定义为：

```
trait Add<RHS=Self> {
    type Output;
```

```
    fn add(self, rhs: RHS) -> Self::Output;
}
```

在这个trait中，“加数”类型为Self，“被加数”类型被设计为类型参数RHS，它有默认值为Self，求和计算结果的类型被设计为关联类型Output。用前面所讲解的思路来分析可以发现，这样的设计是最合理的方式。“被加数”类型在泛型参数列表中，因此我们可以为不同的类型实现Add加法操作，类型A可以与类型B相加，也可以与类型C相加。而计算结果的类型不能是泛型参数，因为它是被Self和RHS所唯一固定的。它需要在impl阶段就确定下来，而不是等到函数调用阶段由用户指定，是典型的“输出类型参数”。

21.7 泛型特化

Rust语言支持泛型特化，但还处于开发过程中，目前只能在最新的nightly版本中试用一些基本功能。

Rust不支持函数和结构体的特化。它支持的是针对impl块的特化。我们可以为一组类型impl一个trait，同时为其中一部分更特殊的类型impl同一个trait。

示例如下：

```
#![feature(specialization)]

use std::fmt::Display;

trait Example {
    fn call(&self);
}

impl<T> Example for T
{
    default fn call(&self) {
        println!("most generic");
    }
}

impl<T> Example for T
    where T: Display
{
    default fn call(&self) {
        println!("generic for Display, {}", self);
    }
}

impl Example for str {
    fn call(&self) {
        println!("specialized for str, {}", self);
    }
}

fn main() {
    let v1 = vec![1i32,2,3];
    let v2 = 1_i32;
    let v3 = "hello";

    v1.call();
    v2.call();
    v3.call();
}
```

用nightly版本编译，执行，结果为：

```
most generic
generic for Display, 1
specialized for str, hello
```

这段代码中有三个impl块。第一个是针对所有类型实现Example。第二个是针对所有的满足T：Display的类型实现Example。第三个是针对具体的str类型实现Example。一个比一个更具体，更特化。

对于主程序中v1.call（）；调用，因为Vec<i32>类型只能匹配第一个impl块，因此它调用的是最基本的版本。对于主程序中的v2.call（）；调用，因为i32类型满足Display约束，所以它同时满足第一个或者第二个impl块的实现版本，而第二个impl块比第一个更具体、更匹配，所以编译器选择了调用第二个impl块的版本。而对于v3.call（）；这句代码，实际上三个impl块都能和str类型相匹配，但是第三个impl块明显比其他两个impl块更特化，因此在主程序中调用的时候，选择了执行第三个impl块提供的版本。

在这个示例中，前面的impl块针对的类型范围更广，后面的impl块针对的类型更具体，它们针对的类型集合是包含关系，这就是特化。当编译器发现，针对某个类型，有多个impl能满足条件的时候，它会自动选择使用最特殊、最匹配的那个版本。

21.7.1 特化的意义

在RFC 1210中，作者列出了泛型特化的三个意义：

1.性能优化。泛型特化可以为某些情况提供统一抽象下的特殊实现。

2.代码重用。泛型特化可以提供一些默认（但不完整的）实现，某些情况下可以减少重复代码。

3.为“高效继承”铺路。泛型特化其实跟OOP中的继承很像。

下面拿标准库中的代码举例说明。

标准库中存在一个ToString trait，定义如下：

```
pub trait ToString {
    fn to_string(&self) -> String;
}
```

凡是实现了这个trait的类型，都可以调用to_string来得到一个String类型的结果。同时，标准库中还存在一个std：：fmt：：Display trait，也可以做到类似的事情。而且Display是可以通过#[derive（Display）]由编译器自动实现的。所以，我们可以想到，针对所有满足T：Display的类型，我们可以为它们提供一个统一的实现：

```
impl<T: fmt::Display + ?Sized> ToString for T {
    #[inline]
    fn to_string(&self) -> String {
        use core::fmt::Write;
        let mut buf = String::new();
        let _ = buf.write_fmt(format_args!("{}", self));
        buf.shrink_to_fit();
        buf
    }
}
```

这样一来，我们就没必要针对每一个具体类型来实现ToString。这么做，非常有利于代码重用，所有满足T：Display的类型，都自动拥有了to_string方法，不必一个个地手动实现。这样做代码确实简洁了，但是，对于某些类型，比如&str类型想调用to_string方法，效率就有点差强人意了。因为这段代码针对的是所有满足Display约束的类型来实现的，它调用的是fmt模块的功能，内部实现非常复杂而烦琐。如果我们用&str类型调用to_string方法的话，还走这么复杂的一套逻辑，略显多余。这也是为什么在早期的Rust代码中，&str转为String类型比较推荐以下方式：

```
// 推荐
let x : String = "hello".into();
// 推荐
let x : String = String::from("hello");
// 不推荐,因为效率低
let x : String = "hello".to_string();
```

现在有了泛型特化，这个性能问题就可以得到修复了，方案如下：

```
impl<T: fmt::Display + ?Sized> ToString for T {
    #[inline]
    default fn to_string(&self) -> String {
        ...
    }
}

impl ToString for str {
    #[inline]
    fn to_string(&self) -> String {
        String::from(self)
    }
}
```

我们可以为那个更通用的版本加一个default标记，然后再为str类型专门写一个特殊的实现。这样，对外接口依然保持统一，但内部实现有所区别，尽可能地提高了效率，满足了“零开销抽象”的原则。

21.7.2 default上下文关键字

我们可以看到，在使用泛型特化功能的时候，我们在许多方法前面加上了default关键字做修饰。这个default不是全局关键字，而是一个“上下文相关”关键字，它只在这种场景下是特殊的，在其他场景下，我们依然可以用它作为合法的变量/函数名字。比如标准库的Default trait中，就有一个方法名字叫作default（），这并不冲突。

为什么需要这样的一个关键字呢？这是因为，泛型特化其实很像传统OOP中的继承重写override功能。在传统的支持重写功能的语言中，一般都有一个类似的标记：

（1）C++中使用了virtual关键字来让一个方法可以在子类型中重写。Modern C++还支持final和override限定符。

（2）C#要求使用virtual关键字定义虚函数，用override关键字标记重写方法。

（3）Java默认让所有方法都是虚方法，但它也支持用final关键字让方法不可被重写。

之所以虚函数需要这样的标记，主要是因为重写方法在某种意义上是一种非局部交互。调用虚函数的时候，有可能调用的是在另外一个子

类中被重写后的版本。在这种情况下，最好是用一种方式表示出来，这个方法是有可能在其他地方被重写的。

虽然模板特化跟虚函数重写不是一个东西，但它们很相似。所以，一个可以被特化的方法，最好也用一个明显的标签显式标记出来。这个设计也可以保证代码的前向兼容性。假如没有这样的语法规定，那么很可能出现的场景是：以前写的一个库明明执行起来没问题，现在有了泛型特化之后，跟其他库一起用到一个项目中，就有了问题。比如，在某个项目中，我们针对一组类型实现了某个trait，而且存在一个变量调用了这个trait内部的方法。但是如果我们引入一个新的库，这个库针对某具体类型实现了同样的trait，就意味着存在了一个更精确的、更特化的实现，于是原来项目中调用的方法就被悄无声息地改变了。如果我们规定trait的设计者本身有权规定自己的这个trait（以及trait内部的每个方法）是否支持特化，就不会出现这个问题。原有trait内部的方法没有default修饰的话，对它进行特化只会导致编译错误。

如果没有default修饰的话，会产生的新问题。Rust里impl块中的方法默认是不可被“重写”特化的。比如我们已经有一个这样的impl代码：

```
impl<T> Example for T {
    type Output = Box<T>;
    fn generate(self) -> Box<T> { Box::new(self) }
}
```

此时，下面这段调用应该是可以编译通过的：

```
fn test(t: bool) -> Box<bool> {
    Example::generate(t)
}
```

我们假设一个针对bool类型的特化版本：

```
impl Example for bool {
    type Output = bool;
    fn generate(self) -> bool { self }
}
```

这样就会导致原来的test方法编译失败，因为generate方法的返回类

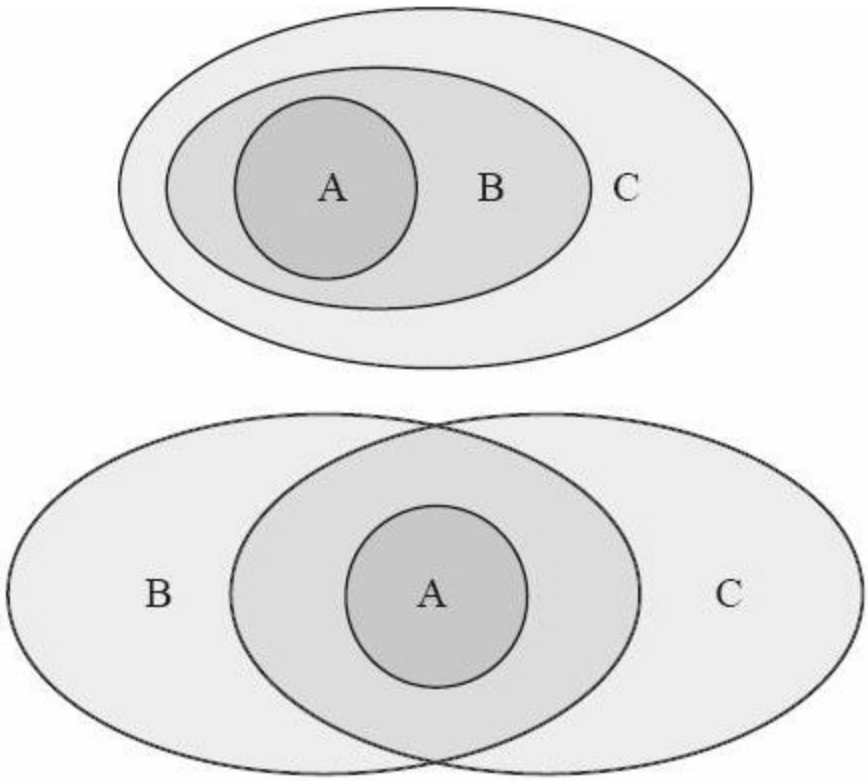
型变成了bool，而不是Box<bool>。如果我们要求使用default关键字来标记是否可以被特化的话，这个问题就可以解决了：

```
impl<T> Example for T {
    default type Output = Box<T>;
    default fn generate(self) -> Box<T> { Box::new(self) }
}

// 编译器应该会认为下面的返回类型不匹配
fn test(t: bool) -> Box<bool> {
    Example::generate(t)
}
```

编译器可以制定一个这样的规则：如果关联类型前面有default修饰，那么调用generate方法的返回值不能直接当成Box<T>类型处理。这就可以解决代码兼容性问题。

21.7.3 交叉impl

前面我们已经演示了什么是泛型特化。我们可以利用泛型针对一组类型写impl，还可以继续针对这个集合中的某一个部分写更特殊的impl。用集合的韦恩图表示如右图所示：



但是，如果我们写impl的时候针对的两个集合并非真子集关系，而是交集关系，怎么办？就像右图这样：

用代码表示：

```
trait Foo {}
trait B {}
trait C {}

// 第一个 impl
impl<T> Foo for T where T: B {}
// 第二个 impl
impl<T> Foo for T where T: C {}
```

此时就出现了交叉的情况。万一有一个类型满足T：B+C呢？在调用Foo中的方法的时候，究竟选B版本还是选C版本？B和C之间不存在继承关系，它们不是真包含关系，所以判断不出究竟哪个更匹配、更特殊、更吻合。所以在这种情况下，编译器理应报错。

为了解决这个问题，Rust设计者又提出了一个交集规则：如果两个impl之间存在交集，而且又不是真包含关系，那么可以为这个交集再写

一个impl，这样这个impl就是最特化的那个版本。即：

```
// 第三个 impl
impl<T> Foo for T where T: B+C {}
```

如果一个类型既满足B约束，又满足C约束，那么与它最匹配的impl版本就是新加入的第三个impl。

交集规则看起来既简洁又直观，可惜它并没有完全解决问题，还有一个遗留问题——跨项目交互。

下面有一个比较复杂的例子，来源于Niko的博客。假设我们的项目中设计了一个RichDisplay，它很像标准库中的Display trait，但是提供了更多的新功能。我们会为所有已经实现了Display的类型来impl我们的RichDisplay：

```
// crate 1
trait RichDisplay {}
// impl 1
impl<T> RichDisplay for T where T: Display {}
```

现在假设在另一个项目widget里面有个类型Widget<T>。它没有实现Display，但是我们希望为它实现RichDisplay。这没有什么问题，因为RichDisplay是我们自己设计的：

```
// crate 1
// impl 2
impl<T> RichDisplay for Widget<T> where T: RichDisplay {}
```

到这里我们碰到问题了：第一个impl和第二个impl存在“潜在的”交叉的可能性。第一个impl是为所有满足Display约束的类型实现RichDisplay；第二个impl是为Widget类型实现RichDisplay。这两个impl针对的类型集合是有交叉的，因此应该出现编译错误。

这里的问题在于，Widget类型是在另外一个项目中定义的。虽然它现在没有impl Display，但它存在这样的可能性。也就是说，widget项目加入了以下的代码，也不应该是一个破坏性扩展：

```
// crate widget
// impl 3
impl<T> Display for Widget<T> {}
```

如果这段代码存在，那么前面的第一个impl和第二个impl就冲突了。所以现在就有一个困境：上游的widget crate可能在将来加入或者不加入这个impl，所以目前第一个impl和第二个impl之间的关系，既不能被认为是完全无关，也不能认定为泛型特化。如果设计不当，上游项目中一个简单的impl块，有可能引起下游用户的breaking change，这是不应该出现的。

所以，如果我们要为某些trait添加默认实现，而且还想继续保持代码的前向兼容性，这个问题就必须考虑。仅仅靠“真子集”规则是不够的。

目前，泛型特化的完整规则依然处于酝酿之中，它的功能短时间内也还不能稳定，请大家关注RFC项目中的相关提案，跟踪最新的讨论结果。

第22章　闭包

闭包（closure）是一种匿名函数，具有“捕获”外部变量的能力。闭包有时候也被称作lambda表达式。它有两个特点：（1）可以像函数一样被调用；（2）可以捕获当前环境中的变量。Rust中的闭包，基本语法如下：

```
fn main() {

    let add = | a :i32, b:i32 | -> i32 { return a + b; } ;

    let x = add(1,2);
    println!("result is {}", x);
}
```

可以看到，以上闭包有两个参数，以两个|包围。执行语句包含在{}中。闭包的参数和返回值类型的指定与普通函数的语法相同。闭包的参数和返回值类型都是可以省略的，因此以上闭包可省略为：

```
let add = |a , b| {return a + b;};
```

和普通函数一样，返回值也可以使用语句块表达式完成，与return语句的作用一样。因此以上闭包可省略为：

```
let add = |a, b| { a + b };
```

更进一步，如果闭包的语句体只包含一条语句，那么外层的大括号也可以省略；如果有多条语句则不能省略。因此以上闭包可以省略为：

```
let add = |a, b| a + b;
```

closure看起来和普通函数很相似，但实际上它们有许多区别。最主要的区别是，closure可以“捕获”外部环境变量，而fn不可以。示例如下：

```
fn main() {
    let x = 1_i32;

    fn inner_add() -> i32 {
        x + 1
    }

    let x2 = inner_add();
    println!("result is {}", x2);
}
```

编译，出现编译错误：

```
error: can't capture dynamic environment in a fn item; use the || { ... } closure fo
```

由此可见，函数inner_add是不能访问变量x的。那么根据编译器的提示，我们改为闭包试试：

```
fn main() {
    let x = 1_i32;

    let inner_add = || x + 1;

    let x2 = inner_add();
    println!("result is {}", x2);
}
```

编译通过。

对于不需要捕获环境变量的场景，普通函数fn和closure是可以互换使用的：

```
fn main() {
    let option = Some(2);
    let new: Option<i32> = option.map(multiple2);
    println!("{:?}", new);

    fn multiple2(val: i32) -> i32{ val*2 }
}
```

在上面示例中，map方法的签名是：

```
fn map<U, F>(self, f: F) -> Option<U>
    where F: FnOnce(T) -> U
```

这里的FnOnce在下文中会详细解释。它在此处的含义是：f是一个闭包参数，类型为FnOnce（T）->U，根据上下文类型推导，实际上是FnOnce（i32）->i32。我们定义了一个普通函数，类型为fn（i32）->i32，也可以用于该参数中。如果我们用闭包来写，下面这样写也可以：

```
let new: Option<i32> = option.map(|val| val * 2);
```

普通函数和闭包之间最大的区别是，普通函数不可以捕获环境变量。在上面的例子中，虽然我们的multiple2函数定义在main函数体内，但是它无权访问main函数内的局部变量。其次，fn定义和调用的位置并不重要，Rust中是不需要前向声明的。只要函数定义在当前范围内是可以观察到的，就可以直接调用，不管在源码内的相对位置如何。相对而言，closure更像是一个的变量，它具有和变量同样的“生命周期”。

22.1 变量捕获

接下来我们研究一下closure的原理。Rust目前的closure实现，又叫作unboxed closure，它的原理与C++11的lambda非常相似。当一个closure创建的时候，编译器帮我们生成了一个匿名struct类型，通过自动分析closure的内部逻辑，来决定该结构体包括哪些数据，以及这些数据该如何初始化。

考虑以下例子：

```
fn main() {
    let x = 1_i32;
    let add_x = | a | x + a;
    let result = add_x( 5 );
    println!("result is {}", result);
}
```

我们来思考一下：如果不使用闭包来实现以上逻辑，该怎么做？方法如下：

```
struct Closure {
    inner1: i32
}

impl Closure {
    fn call(&self, a: i32) -> i32 {
        self.inner1 + a
    }
}

fn main() {
    let x = 1_i32;
    let add_x = Closure{ inner1: x};
    let result = add_x.call(5);
    println!("result is {}", result);
}
```

上面这个例子，我们模拟了一个闭包的原理，实际上Rust编译器就是用类似的手法来处理闭包语法的。对比一下使用闭包语法的版本和手动实现的版本，我们可以看到，创建闭包的时候，就相当于创建了一个结构体，我们把需要捕获的环境变量存到这个结构体中。闭包调用的时候，相当于调用了跟这个结构体相关的一个成员函数。

但是，还有几个问题没有解决。当编译器把闭包语法糖转换为普通的类型和函数调用的时候：

（1）结构体内部的成员应该用什么类型，如何初始化？应该用i32或是&i32还是&mut i32？

（2）函数调用的时候self应该用什么类型？应该写self或是&self还是&mut self？

理解了这两个问题的答案，就能完全理解了Rust的闭包的原理。

关于第一个问题，Rust主要是通过分析外部变量在闭包中的使用方式，通过一系列的规则自动推导出来的。主要规则如下：

（1）如果一个外部变量在闭包中，只通过借用指针&使用，那么这个变量就可通过引用&的方式捕获；

（2）如果一个外部变量在闭包中，通过&mut指针使用过，那么这个变量就需要使用&mut的方式捕获；

（3）如果一个外部变量在闭包中，通过所有权转移的方式使用过，那么这个变量就需要使用“by value”self的方式捕获。

简单点总结规则是，在保证能编译通过的情况下，编译器会自动选择一种对外部影响最小的类型存储。对于被捕获的类型为T的外部变量，在匿名结构体中的存储方式选择为：尽可能先选择&T类型，其次选择&mut T类型，最后选择T类型。示例如下：

```
struct T(i32);

fn by_value(_: T) {}
fn by_mut(_: &mut T) {}
fn by_ref(_: &T) {}

fn main() {
    let x: T = T(1);
    let y: T = T(2);
    let mut z: T = T(3);

    let closure = || {
        by_value(x);
        by_ref(&y);
        by_mut(&mut z);
    };
```

```
    closure();
}
```

以上闭包捕获了外部的三个变量x y z。其中，y通过&T的方式被使用了；z通过&mut T的方式被使用了；x通过T的方式被使用了。编译器会根据这些信息，自动生成结构类似下面这样的匿名结构体：

```
// 实际类型名字是编译器按某些规则自动生成的
struct ClosureEnvironment<'y, 'z> {
    x: T,
    y: &'y T,
    z: &'z mut T,
}
```

而原示例中的closure这个局部变量，就是这个类型的实例。对我们来说，这个类型是匿名的。

22.2 move关键字

以上变量捕获的规则都是针对只作为局部变量的闭包而准备的。有些时候，我们的闭包的生命周期可能会超过一个函数的范围。比如，我们可以将此闭包存储到某个数据结构中，在当前函数返回之后继续使用。这样一来，就可能出现更复杂的情况：在闭包被创建的时候，它通过引用的方式捕获了某些局部变量，而在闭包被调用的时候，它所指向的一些外部变量已经被释放了。示例如下：

```
fn make_adder(x: i32) -> Box<Fn(i32) -> i32> {
    Box::new(|y| x + y)
}

fn main() {
    let f = make_adder(3);

    println!("{}", f(1)); // 4
    println!("{}", f(10)); // 13
}
```

大家可以看到，函数make_adder中有一个局部变量x，按照前面所述的规则，它被闭包所捕获，而且可以使用引用&的方式完成闭包内部的逻辑，因此它是被引用捕获的。而闭包则作为函数返回值被传递出去了。于是，闭包被调用的时候，它内部的引用所指向的内容已经被释放了。这种情况，应该会出现典型的野指针问题，属于内存不安全的范畴。幸运的是，该程序在Rust中根本无法编译通过，错误信息为：

```
error: closure may outlive the current function, but it borrows `x`, which is owned
```

信息提示得非常清晰，我们又可以感谢Rust帮我们发现了一个问题。

那么这种情况，我们应该怎样写才对呢？这里要介绍一个新的关键字move，它的功能是用于修饰一个闭包。示例如下：

```
fn make_adder(x: i32) -> Box<Fn(i32) -> i32> {
    Box::new(move |y| x + y)
    // 注意这里 ^^^
```

```
}
```

加上move关键字之后，所有的变量捕获全部使用by value的方式。也就是说，编译器生成的匿名结构体内部看起来像是下面这样：

```
struct ClosureEnvironment {
    x: TYPE1,  //
    y: TYPE2,  // 这里没有 &TYPE,&mut TYPE,所有被捕获的外部变量所有权一律转移进闭包
    z: TYPE3,  //
}
```

所以，move关键字可以改变闭包捕获变量的方式，一般用于闭包需要传递到函数外部（escaping closure）的情况。

22.3 Fn/FnMut/FnOnce

外部变量捕获的问题解决了，我们再看看第二个问题，闭包被调用的方式。我们注意到，闭包被调用的时候，不需要执行某个成员函数，而是采用类似函数调用的语法来执行。这是因为它自动实现了编译器提供的几个特殊的trait，Fn或者FnMut或者FnOnce。

注意：小写fn是关键字，用于声明函数；大写的Fn不是关键字，只是定义在标准库中的一个trait。

它们的定义如下：

```
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
```

这几个trait的主要区别在于，被调用的时候self参数的类型。FnOnce被调用的时候，self是通过move的方式传递的，因此它被调用之后，这个闭包的生命周期就已经结束了，它只能被调用一次；FnMut被调用的时候，self是&mut Self类型，有能力修改当前闭包本身的成员，甚至可能通过成员中的引用，修改外部的环境变量；Fn被调用的时候，self是&Self类型，只有读取环境变量的能力。

目前这几个trait还处于unstable状态。在目前的稳定版编译器中，我们不能针对自定义的类型实现这几个trait，只能在nightly版本中开启#![feature（fn_traits）]功能。

那么，对于一个闭包，编译器是如何选择impl哪个trait呢？答案是，编译器会都尝试一遍，实现能让程序编译通过的那几个。闭包调用的时候，会尽可能先选择调用fn call（&self，args：Args）函数，其次

尝试选择fn call_mut（&self，args：Args）函数，最后尝试使用fn call_once（self，args：Args）函数。这些都是编译器自动分析出来的。

还是用示例来讲解比较清晰：

```
fn main() {
    let v: Vec<i32> = vec![];
    let c = || std::mem::drop(v);
    c();
}
```

对于上例，drop函数的签名是fn drop<T>（_x：T），它接受的参数类型是T。因此，在闭包中使用该函数会导致外部变量v通过move的方式被捕获。编译器为该闭包自动生成的匿名类型，类似下面这样：

```
struct ClosureEnvironment {
    v: Vec<i32> // 这里不是引用
}
```

对于这样的结构体，我们来尝试实现FnMut trait：

```
impl FnMut<Vec<i32>> for ClosureEnvironment {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output {
        drop(self.v)
    }
}
```

当然，这是行不通的，因为函数体内需要一个Self类型，但是函数参数只提供了&mut Self类型。因此，编译器不会为这个闭包实现FnMut trait。唯一能实现的trait就只剩下了FnOnce。

这个闭包被调用的时候，当然就会调用call_once方法。我们知道，fn call_once（self，arg：Args）这个函数被调用的时候，self参数是move进入函数体的，会“吃掉”self变量。在此函数调用后，这个闭包的生命周期就结束了。所以，FnOnce类型的闭包只能被调用一次。FnOnce也是得名于此。我们自己来试一下：

```
fn main() {

    let v: Vec<i32> = vec![];
```

```
    let c = || drop(v); // 闭包使用捕获变量的方式,决定了这个闭包的类型。它只实现了`FnOnce tra
    c();
    c(); // 再调用一次试试,编译错误 use of moved value: `c`。`c`是怎么被move走的?
}
```

编译器在处理上面这段代码的时候，做了一个下面这样的展开：

```
fn main() {
    struct ClosureEnvironment {
        _v: Vec<i32>
    }
    let v: Vec<i32> = vec![];
    let c = ClosureEnvironment { _v: v }; // v move 进入了c的成员中
    c.call_once(); // c move 进入了 call_once 方法中
    c.call_once(); // c 的生命周期已经结束了,这里的调用会发生编译错误
}
```

同样的道理，我们试试Fn的情况：

```
fn main() {
    let v: Vec<i32> = vec![1,2,3];
    let c = || for i in &v { println!("{}", i); };
    c();
    c();
}
```

可以看到，上面这个闭包捕获的环境变量在使用的时候，只需要&Vec<i32>类型即可，因此它只需要捕获环境变量v的引用。因此它能实现Fn trait。闭包在被调用的时候，执行的是fn call（&self）函数，所以，调用多次也是没问题的。

我们如果给上面的程序添加move关键字，依然可以通过：

```
fn main() {
    let v: Vec<i32> = vec![1,2,3];
    let c = move || for i in &v { println!("{}", i); };
    c();
    c();
}
```

可以看到，move关键字只是影响了环境变量被捕获的方式。第三行，创建闭包的时候，变量v被move进入了闭包中，闭包中捕获变量包括了一个拥有所有权的Vec<i32>。第四行，闭包调用的时候，根据推断

规则，它依然是Fn型的闭包，使用的是fn call（&self）函数，因此闭包变量c可以被多次调用。

22.4 闭包与泛型

我们已经知道，闭包是依靠trait来实现的。跟普通trait一样，我们不能直接用Fn FnMut FnOnce作为变量类型、函数参数、函数返回值。

跟其他的trait相比，闭包相关的trait语法上有特殊之处。比如，如果我们想让闭包作为一个参数传递到函数中，可以这样写：

```
fn call_with_closure<F>(some_closure: F) -> i32
        where F : Fn(i32) -> i32 {
        some_closure(1)
}

fn main() {
    let answer = call_with_closure(|x| x + 2);
    println!("{}", answer);
}
```

其中泛型参数F的约束条件是F：Fn（i32）->i32。这里Fn（i32）->i32是针对闭包设计的专门的语法，而不是像普通trait那样使用Fn<i32，i32>来写。这样设计为了让它们看起来跟普通函数类型fn（i32）->i32更相似。除了语法之外，Fn FnMut FnOnce其他方面都跟普通的泛型一致。

一定要注意的是：每个闭包，编译器都会为它生成一个匿名结构体类型；即使两个闭包的参数和返回值一致，它们也是完全不同的两个类型，只是都实现了同一个trait而已。下面我们用一个示例演示：

```
fn main() {
// 同一个变量绑定了两次
    let mut closure = |x : i32| -> i32 { x + 2 };
    closure = |x: i32| -> i32 { x - 2 } ;
    println!("{}", closure());
}
```

编译，结果出错，错误信息为：

```
error: mismatched types:
expected `[closure@temp.rs:3:21: 3:47]`,
    found `[closure@temp.rs:4:13: 4:38]`
```

```
(expected closure,
    found a different closure) [E0308]
```

可以看到，我们用同一个变量来绑定两个闭包的时候发生了类型错误。请大家牢牢记住，不同的闭包是不同的类型。

既然如此，跟普通的trait一样，如果我们需要向函数中传递闭包，有下面两种方式。

·通过泛型的方式。这种方式会为不同的闭包参数类型生成不同版本的函数，实现静态分派。

·通过trait object的方式。这种方式会将闭包装箱进入堆内存中，向函数传递一个胖指针，实现运行期动态分派。

关于动态分派和静态分派的内容，将在下一章中详细说明。此处只做一个简单示例：

```
fn static_dispatch<F>(closure: &F)  // 这里是泛型参数。对于每个不同类型的参数,编译器将会生成
    where F: Fn(i32) -> i32
{
    println!("static dispatch {}", closure(42));
}

fn dynamic_dispatch(closure: &Fn(i32)->i32) // 这里是 `trait object``Box<Fn(i32)->i32
{
    println!("dynamic dispatch {}", closure(42));
}

fn main() {
    let closure1 = | x | x * 2;
    let closure2 = | x | x * 3;
    fn function_ptr(x: i32)->i32 { x * 4 };

    static_dispatch(&closure1);
    static_dispatch(&closure2);
    static_dispatch(&function_ptr); // 普通`fn`函数也实现了`Fn trait`,它可以与此参数类型匹

    dynamic_dispatch(&closure1);
    dynamic_dispatch(&closure2);
    dynamic_dispatch(&function_ptr);
}
```

如果我们希望一个闭包作为函数的返回值，那么就不能使用泛型的方式了。因为如果泛型类型不在参数中出现，而仅在返回类型中出现的话，会要求在调用的时候显式指定类型，编译器才能完成类型推导。可

是调用方根本无法指定具体类型，因为闭包类型是匿名类型，用户无法显式指定。所以下面这样的写法是编译不过的：

```
fn test<F>() -> F
    where F: Fn(i32)->i32
{
    return | i | i * 2;
}

fn main() {
    let closure = test();
}
```

修复这段代码有两种方案，一种是静态分派，一种是动态分派。

·静态分派。我们可以用一种新的语法fn test（）->impl Fn（i32）->i32来实现。在后面的章节中有这个语法糖的详细介绍。

·动态分派。就是把闭包装箱进入堆内存中，使用Box<dyn Fn（i32）->i32>这种trait object类型返回。关于trait object的内容可参见下一章。

```
fn test() -> Box<dyn Fn(i32)->i32 >
{
    let c = | i: i32 | i * 2;
    Box::new(c)
}

fn main() {
    let closure = test();
    let r = closure(2);
    println!("{}", r);
}
```

22.5 闭包与生命周期

当使用闭包做参数或返回值的时候，生命周期会变得更加复杂。假设有下面这段代码：

```
fn calc_by<'a, F>(var: &'a i32, f: F) -> i32
    where F: Fn(&'a i32) -> i32
{
    f(var)
}

fn main() {
    let local = 10;
    let result = calc_by(&local, |i| i*2);
    println!("{}", result);
}
```

这段代码可以编译通过。但是，假如我们把calc_by的函数体稍微改一下，变成下面这样：

```
let local = *var;
f(&local)
```

就不能编译通过了。

但是，如果我们把所有的生命周期标记去掉，变成这样：

```
fn calc_by<F>(var: &i32, f: F) -> i32
    where F: Fn(&i32) -> i32
{
    let local = *var;
    f(&local)
}

fn main() {
    let local = 10;
    let result = calc_by(&local, |i| i*2);
    println!("{}", result);
}
```

它就又可以编译通过了。这说明，我们前面对这个calc_by函数手写的生命周期标记有问题。

对于上面这个例子，所有的借用生命周期都是由编译器自动补全的，假如我们手动来补全这些标记，应该怎么做呢？

在这里我们只能使用“高阶生命周期”的表示方法：

```
fn calc_by<'a, F>(var: &'a i32, f: F) -> i32
    where F: for<'f> Fn(&'f i32) -> i32
{
    let local = *var;
    f(&local)
}
```

注意F的约束条件，这样写表示的意思是，Fn的输入参数可作用于任意的生命周期'f，这个生命周期和另外一个参数var的生命周期没有半点关系。

这才是这段代码正确的生命周期标记方式。如果我们不手动标记出来，编译器为我们自动推导的生命周期关系就是这样的高阶生命周期。

谈到“高阶”这两个字，很多朋友会想到高阶类型（higher kinded types）。这里的高阶生命周期确实跟高阶类型有很多相似之处，Rust也确实在思考如何引入高阶类型这个问题，但还没有做出最终决定。到目前为止，for<'a>Fn（&'a Arg）->&'a Ret这样的语法，只能用于生命周期参数，不能用于任意泛型类型。

第23章　动态分派和静态分派

Rust可以同时支持“静态分派”（static dispatch）和“动态分派”（dynamic dispatch）。

所谓“静态分派”，是指具体调用哪个函数，在编译阶段就确定下来了。Rust中的“静态分派”靠泛型以及impl trait来完成。对于不同的泛型类型参数，编译器会生成不同版本的函数，在编译阶段就确定好了应该调用哪个函数。

所谓“动态分派”，是指具体调用哪个函数，在执行阶段才能确定。Rust中的“动态分派”靠Trait Object来完成。Trait Object本质上是指针，它可以指向不同的类型；指向的具体类型不同，调用的方法也就不同。

我们用一个示例来说明。假设我们有一个trait Bird，有另外两个类型都实现了这个trait，我们要设计一个函数，既可以接受Duck作为参数，也可以接受Swan作为参数。

```
trait Bird {
    fn fly(&self);
}

struct Duck;
struct Swan;

impl Bird for Duck {
    fn fly(&self) { println!("duck duck"); }
}

impl Bird for Swan {
    fn fly(&self) { println!("swan swan");}
}
```

首先，大家需要牢牢记住的一件事情是，trait是一种DST类型，它的大小在编译阶段是不固定的。这意味着下面这样的代码是无法编译通过的：

```
fn test(arg: Bird) {}
fn test() -> Bird  {}
```

因为Bird是一个trait，而不是具体类型，它的size无法在编译阶段确定，所以编译器是不允许直接使用trait作为参数类型和返回类型的。这也是trait跟许多语言中的“interface”的一个区别。

这种时候我们有两种选择。一种是利用泛型：

```
fn test<T: Bird>(arg: T) {
    arg.fly();
}
```

这样，test函数的参数既可以是Duck类型，也可以是Swan类型。实际上，编译器会根据实际调用参数的类型不同，直接生成不同的函数版本，类似C++中的template：

```
// 伪代码示意
fn test_Duck(arg: Duck) {
    arg.fly();
}
fn test_Swan(arg: Swan) {
    arg.fly();
}
```

所以，通过泛型函数实现的“多态”，是在编译阶段就已经确定好了调用哪个版本的函数，因此被称为“静态分派”。除了泛型之外，Rust还提供了一种impl Trait语法，也能实现静态分派。

我们还有另外一种办法来实现“多态”，那就是通过指针。虽然trait是DST类型，但是指向trait的指针不是DST。如果我们把trait隐藏到指针的后面，那它就是一个trait object，而它是可以作为参数和返回类型的。

```
// 根据不同需求,可以用不同的指针类型,如 Box/&/&mut 等
fn test(arg: Box<dyn Bird>) {
    arg.fly();
}
```

在这种方式下，test函数的参数既可以是Box<Duck>类型，也可以是Box<Swan>类型，一样实现了“多态”。但在参数类型这里已经将“具体类型”信息抹掉了，我们只知道它可以调用Bird trait的方法。而具体调

用的是哪个版本的方法，实际上是由这个指针的值来决定的。这就是“动态分派”。

23.1 trait object

什么是trait object呢？指向trait的指针就是trait object。假如Bird是一个trait的名称，那么dyn Bird就是一个DST动态大小类型。&dyn Bird、&mut dyn Bird、Box<dyn Bird>、*const dyn Bird、*mut dyn Bird以及Rc<dyn Bird>等等都是Trait Object。

注意　dyn是一个新引入的上下文关键字（Contextual keyword），目前还没有稳定。用户需要手动打开#![feature（dyn_trait）]才能使用。目前的trait object默认的语法是没有dyn关键字的。

trait object这个名字以后也会被改为dynamic trait type。impl Trait for Trait这样的语法同样会被改为impl Trait for dyn Trait。这样能更好地跟impl Trait语法对应起来。

当指针指向trait的时候，这个指针就不是一个普通的指针了，变成一个“胖指针”。请大家回忆一下前文所讲解的DST类型：数组类型[T]是一个DST类型，因为它的大小在编译阶段是不确定的，相对应的，&[T]类型就是一个“胖指针”，它不仅包含了指针指向数组的其中一个元素，同时包含一个长度信息。它的内部表示实质上是Slice类型。

同理，Bird只是一个trait的名字，符合这个trait的具体类型可能有多种，这些类型并不具备同样的大小，因此使用dyn Bird来表示满足Bird约束的DST类型。指向DST的指针理所当然也应该是一个“胖指针”，它的名字就叫trait object。比如Box<dyn Bird>，它的内部表示可以理解成下面这样：

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

它里面包含了两个成员，都是指向单元类型的裸指针。在这里声明的指针指向的类型并不重要，我们只需知道它里面包含了两个裸指针即可。由上可见，和Slice一样，Trait Object除了包含一个指针之外，还带

有另外一个“元数据”，它就是指向“虚函数表”的指针。这里用的是裸指针，指向unit类型的指针*mut（）实际上类似于C语言中的void*。我们来尝试一下使用unsafe代码，如果把它里面的数值当成整数拿出来会是什么结果：

```
#![feature(dyn_trait)]

use std::mem;

trait Bird {
    fn fly(&self);
}

struct Duck;
struct Swan;

impl Bird for Duck {
    fn fly(&self) { println!("duck duck"); }
}

impl Bird for Swan {
    fn fly(&self) { println!("swan swan");}
}
// 参数是 trait object 类型,p 是一个胖指针
fn print_traitobject(p: &dyn Bird) {

    // 使用transmute执行强制类型转换,把变量p的内部数据取出来
    let (data, vtable) : (usize, * const usize) = unsafe {mem::transmute(p)};
    println!("TraitObject    [data:{}, vtable:{:p}]", data, vtable);
    unsafe {
        // 打印出指针 v 指向的内存区间的值
        println!("data in vtable [{}, {}, {}, {}]",
            *vtable, *vtable.offset(1), *vtable.offset(2), *vtable.offset(3));
    }
}

fn main() {
    let duck = Duck;
    let p_duck = &duck;
    let p_bird = p_duck as &dyn Bird;
    println!("Size of p_duck {}, Size of p_bird {}", mem::size_of_val(&p_duck), mem

    let duck_fly : usize = Duck::fly as usize;
    let swan_fly : usize = Swan::fly as usize;
    println!("Duck::fly {}", duck_fly);
    println!("Swan::fly {}", swan_fly);

    print_traitobject(p_bird);
    let swan = Swan;
    print_traitobject(&swan as &dyn Bird);
}
```

执行结果为：

```
Size of p_duck 8, Size of p_bird 16
Duck::fly 139997348684016
Swan::fly 139997348684320
TraitObject    [data:140733800916056, vtable:139997351089872]
data in vtable [139997348687008, 0, 1, 139997348684016]
TraitObject    [data:140733800915512, vtable:139997351089952]
data in vtable [139997348687008, 0, 1, 139997348684320]
```

我们可以看到，直接针对对象取指针，得到的是普通指针，它占据64 bit的空间。如果我们把这个指针使用as运算符转换为trait object，它就成了胖指针，携带了额外的信息。这个额外信息很重要，因为我们还需要使用这个指针调用函数。如果指向trait的指针只包含了对象的地址，那么它就没办法实现针对不同的具体类型调用不同的函数了。所以，它不仅要包含一个指向真实对象的指针，还要有一个指向所谓的“虚函数表”的指针。我们把虚函数表里面的内容打印出来可以看到，里面有我们需要被调用的具体函数的地址。

从这里的分析结果可以看到，Rust的动态分派和C++的动态分派，内存布局有所不同。在C++里，如果一个类型里面有虚函数，那么每一个这种类型的变量内部都包含一个指向虚函数表的地址。而在Rust里面，对象本身不包含指向虚函数表的指针，这个指针是存在于trait object指针里面的。如果一个类型实现了多个trait，那么不同的trait object指向的虚函数表也不一样。

23.2 object safe

既然谈到trait object就不得不说一下object safe的概念。trait object的构造是受到许多约束的，当这些约束条件不能满足的时候，会产生编译错误。

我们来看看在哪些条件下trait object是无法构造出来的。

1.当trait有Self：Sized约束时

一般情况下，我们把trait当作类型来看的时候，它是不满足Sized条件的。因为trait只是描述了公共的行为，并不描述具体的内部实现，实现这个trait的具体类型是可以各种各样的，占据的空间大小也不是统一的。Self关键字代表的类型是实现该trait的具体类型，在impl的时候，针对不同的类型，有不同的具体化实现。如果我们给Self加上Sized约束：

```
#![feature(dyn_trait)]
trait Foo where Self: Sized {
    fn foo(&self);
}

impl Foo for i32 {
    fn foo(&self) { println!("{}", self); }
}

fn main() {
    let x = 1_i32;
    x.foo();
    //let p = &x as &dyn Foo;
    //p.foo();
}
```

我们可以看到，直接调用函数foo依然是可行的。可是，当我们试图创建trait object的时候，编译器阻止了我们：

```
error: the trait `Foo` cannot be made into an object [E0038]
```

所以，如果我们不希望一个trait通过trait object的方式使用，可以为它加上Self：Sized约束。

同理，如果我们想阻止一个函数在虚函数表中出现，可以专门为该函数加上Self：Sized约束：

```
#![feature(dyn_trait)]
trait Foo {
    fn foo1(&self);
    fn foo2(&self) where Self: Sized;
}

impl Foo for i32 {
    fn foo1(&self) { println!("foo1 {}", self); }
    fn foo2(&self) { println!("foo2 {}", self); }
}

fn main() {
    let x = 1_i32;
    x.foo2();
    let p = &x as &dyn Foo;
    p.foo2();
}
```

编译以上代码，可以看到，如果我们针对foo2函数添加了Self：Sized约束，那么就不能通过trait object来调用这个函数了。

2.当函数中有Self类型作为参数或者返回类型时

Self类型是个很特殊的类型，它代表的是impl这个trait的当前类型。比如说，Clone这个trait中的clone方法就返回了一个Self类型：

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

我们可以想象一下，假如我们创建了一个Clone trait的trait object，并调用clone方法：

```
let p: &dyn Clone = if from_input() { &obj1 as &dyn Clone } else { &obj2 as &dyn Clo
let o = p.clone();
```

变量o应该是什么类型？编译器不知道，因为它在编译阶段无法确定。p指向的具体对象，它的类型是什么只能在运行阶段确定，无法在编译阶段确定。在编译阶段，我们知道的仅仅是这个类型实现了Clone

trait，其他的就一无所知了。而这个clone（）方法又要求返回一个与p指向的具体类型一致的返回类型，所以o的类型是无法确定的。对编译器来说，这是无法完成的任务。所以，std：：clone：：Clone这个trait就不是object safe的，我们不能利用&dyn Clone构造trait object来实现虚函数调用。

编译下面的代码：

```
#![feature(dyn_trait)]
fn main() {
    let s = String::new();
    let p : &dyn Clone = &s as &dyn Clone();
}
```

编译器会提示错误：

```
error: the trait `std::clone::Clone` cannot be made into an object
```

Rust规定，如果函数中除了self这个参数之外，还在其他参数或者返回值中用到了Self类型，那么这个函数就不是object safe的。这样的函数是不能使用trait object来调用的。这样的方法是不能在虚函数表中存在的。

这样的规定在某些情况下会给我们造成一定的困扰。假如我们有下面这样一个trait，它里面的一部分方法是满足object safe的，而另外一部分是不满足的：

```
#![feature(dyn_trait)]
trait Double {
    fn new() -> Self;
    fn double(&mut self);
}

impl Double for i32 {
    fn new() -> i32 { 0 }
    fn double(&mut self) { *self *= 2; }
}

fn main() {
    let mut i = 1;
    let p : &mut dyn Double = &mut i as &mut dyn Double;
    p.double();
}
```

编译会出错，因为new（）这个方法是不满足object safe条件的。但是我们其实只想在trait object中调用double方法，并不指望通过trait object调用new（）方法，但可惜编译器还是直接禁止了这个trait object的创建。

面对这样的情况，我们应该怎么处理呢？我们可以通过下面的写法，把new（）方法从trait object的虚函数表中移除：

```
fn new() -> Self where Self: Sized;
```

把这个方法加上Self：Sized约束，编译器就不会在生成虚函数表的时候考虑它了。生成trait object的时候，只需考虑double（）这一个方法，编译器就会很愉快地创建这样的虚函数表和trait object。通过这种方式，我们就可以解决掉一个trait中一部分方法不满足object safe的烦恼。

3.当函数第一个参数不是self时

意思是，如果有“静态方法”，那这个“静态方法”是不满足object safe条件的。这个条件几乎是显然的，编译器没有办法把静态方法加入到虚函数表中。

与上面讲解的情况类似，如果一个trait中存在静态方法，而又希望通过trait object来调用其他的方法，那么我们需要在这个静态方法后面加上Self：Sized约束，将它从虚函数表中剔除。

4.当函数有泛型参数时

假如我们有下面这样的trait：

```
trait SomeTrait {
    fn generic_fn<A>(&self, value: A);
}
```

这个函数带有一个泛型参数，如果我们使用trait object调用这个函数：

```
fn func(x: &dyn SomeTrait) {
    x.generic_fn("foo"); // A = &str
    x.generic_fn(1_u8);  // A = u8
}
```

这样的写法会让编译器特别犯难，本来x是trait object，通过它调用成员的方法是通过vtable虚函数表来进行查找并调用。现在需要被查找的函数成了泛型函数，而泛型函数在Rust中是编译阶段自动展开的，generic_fn函数实际上有许多不同的版本。这里有一个根本性的冲突问题。Rust选择的解决方案是，禁止使用trait object来调用泛型函数，泛型函数是从虚函数表中剔除了的。这个行为跟C++是一样的。C++中同样规定了类的虚成员函数不可以是template方法。

23.3 impl trait

本节所讲的impl trait是一个全新的语法。比如下这样：

```
fn foo(n: u32) -> impl Iterator<Item=u32> {
    (0..n).map(|x| x * 100)
}
```

下面我们来讲解impl Iterator<Item=u32>。

我们注意到，在写泛型函数的时候，参数传递方式可以有：静态分派或动态分派两种选择：

```
fn consume_iter_static<I: Iterator<Item=u8>>(iter: I)
fn consume_iter_dynamic(iter: Box<dyn Iterator<Item=u8>>)
```

不论选用哪种方式，都可以写出针对一组类型的抽象代码，而不是针对某一个具体类型的。在consume_iter_static版本中，每次调用的时候，编译器都会为不同的实参类型实例化不同版本的函数。在consume_iter_dynamic版本中，每次调用的时候，实参的具体类型隐藏在了trait object的后面，通过虚函数表，在执行阶段选择调用正确的函数版本。

这两种方式都可以在函数参数中正常使用。但是，如果我们考虑函数的返回值，目前只有这样一种方式是合法的：

```
fn produce_iter_dynamic() -> Box<dyn Iterator<Item=u8>>
```

以下这种方式是不合法的：

```
fn produce_iter_static() -> Iterator<Item=u8>
```

目前版本中，Rust只支持返回“具体类型”，而不能返回一个trait。由于缺少了“不装箱的抽象返回类型”这样一种机制，导致了以下这些问

题。

·我们返回一个复杂的迭代器的时候，会让返回类型过于复杂，而且泄漏了具体实现。比如，如果我们需要返回一个栈上的迭代器，可能需要为函数写复杂的返回类型：

```
Chain<Map<'a, (int, u8), u16, Enumerate<Filter<'a, u8, vec::MoveItems<u8>>>>, SkipWh
```

函数内部的逻辑稍微有点变化，这个返回类型就要跟着改变，远不如泛型函数参数T：Iterator的抽象程度好。

·函数无法直接返回一个闭包。因为闭包的类型是编译器自动生成的一个匿名类型，我们没办法在函数的返回类型中手工指定，所以返回一个闭包一定要“装箱”到堆内存中，然后把胖指针返回回去，这样是有性能开销的。

```
fn multiply(m: i32) -> Box<dyn Fn(i32)->i32> {
    Box::new(move |x|x*m)
}

fn main() {
    let f = multiply(5);
    println!("{}", f(2));
}
```

请注意，这种时候引入一个泛型参数代表这个闭包是行不通的：

```
fn multiply<T>(m: i32) -> T where T:Fn(i32)->i32 {
    move |x|x*m
}

fn main() {
    let f = multiply(5);
    println!("{}", f(2));
}
```

编译出错，编译错误为：

```
note: expected type `T`
    found type `[closure@test.rs:3:5: 3:16 m:_]`
```

因为泛型这种语法实际的意思是，泛型参数T由“调用者”决定。比如std：：iter：：Iterator：：collect这个函数就非常适合这样实现：

```
let a = [1, 2, 3];
let doubled = a.iter()
        .map(|&x| x * 2)
        .collect::<???>::();
```

使用者可以在？？？这个地方填充不同的类型，如Vec<i32>、VecDeque<i32>、LinkedList<i32>等。这个collect方法的返回类型是一个抽象的类型集合，调用者可以随意选择这个集合中的任意一个具体类型。

这跟我们上面想返回一个内部的闭包情况不同，上面的程序想表达的是返回一个“具体类型”，这个类型是由被调用的函数自行决定的，只是调用者不知道它的名字而已。

为了解决上面的问题，aturon提出了impl trait这个方案。此方案引入了一个新的语法，可以表达一个不用装箱的匿名类型，以及它所满足的基本接口。

示例如下：

```
#![feature(conservative_impl_trait)]

fn multiply(m: i32) -> impl Fn(i32)->i32 {
    move |x|x*m
}

fn main() {
    let f = multiply(5);
    println!("{}", f(2));
}
```

这里的impl Fn（i32）->i32表示，这个返回类型，虽然我们不知道它的具体名字，但是知道它满足Fn（size）->isize这个trait的约束。因此，它解决了“返回不装箱的抽象类型”问题。

它跟泛型函数的主要区别是：泛型函数的类型参数是函数的调用者指定的；impl trait的具体类型是函数的实现体指定的。

为什么这个功能开关名称是#！[feature（conservative_impl_trait）]呢？因为目前为止，它的使用场景非常保守，只允许这个语法用于普通函数的返回类型，不能用于参数类型等其他地方。实际上设计组已经通过了另外一个RFC，将这个功能扩展到了更多的场景，但是这些功能目前在编译器中还没有实现。

·让impl trait用在函数参数中：

```
fn test(f: impl Fn(i32)->i32){}
```

·让impl trait用在类型别名中：

```
type MyIter = impl Iterator<Item=i32>;
```

·让impl trait用在trait中的方法参数或返回值中：

```
trait Test {
    fn test() -> impl MyTrait;
}
```

·让impl Trait用在trait中的关联类型中：

```
trait Test {
    type AT = impl MyTrait;
}
```

在某些场景下，impl trait这个语法具有明显的优势，因为它可以提高语言的表达能力。但是，要把它推广到各个场景下使用，还需要大量的设计和实现工作。目前的这个RFC将目标缩小为了：先推进这个语法在函数参数和返回值场景下使用，其他的情况后面再考虑。

最后需要跟各位读者提醒一点的是，不要过于激进地使用这个功能，如在每个可以使用impl trait的地方都用它替换原来的具体类型。它更多地倾向于简洁性，而牺牲了一部分表达能力。比如拿前文那个复杂的迭代器类型来说，

```
fn test() -> Chain<Map<...>>
```

我们可能希望将函数返回类型写成下面这样：

```
fn test() -> impl Iterator<Item=u16>
```

在绝大多数应用场景下，这样写更精简、更清晰。但是，这样写实际上是降低了表达能力。因为，使用前一种写法，用户可以拿到这个迭代器之后再调用clone（）方法，而使用后一种写法，就不可以了。如果希望支持clone，那么需要像下面这样写

```
fn test() -> impl Iterator<Item=u16> + Clone
```

而这两个trait依然不是原来那个具体类型的所有对外接口。在某些场景下，需要罗列出各种接口才能完整替代原来的写法，类似下面这样：

```
fn test() -> impl Iterator<Item=u16> +
        Clone +
        ExactSizeIterator+
        TrustedLen
```

先不管这种写法是否可行，单说这个复杂程度，就已经完全失去了impl trait功能的意义了。所以，什么时候该用这个功能，什么时候不该用，应该仔细权衡一下。

第24章　容器与迭代器

24.1　容器

跟C++的STL类似，Rust的标准库也给我们提供了一些比较常用的容器以及相关的迭代器。目前实现了的容器有：

容　器	描　述
Vec	可变长数组，连续存储
VecDeque	双向队列，适用于从头部和尾部插入删除数据
LinkedList	双向链表，非连续存储
HashMap	基于 Hash 算法存储一系列键值对
BTreeMap	基于 B 树存储一系列键值对
HashSet	基于 Hash 算法的集合，相当于没有值的 HashMap
BTreeSet	基于 B 树的集合，相当于没有值的 BTreeMap
BinaryHeap	基于二叉堆实现的优先级队列

下面选择几个常见的命令来讲解它们的用法及特点。

24.1.1　Vec

Vec是最常用的一个容器，对应C++里面的vector。它就是一个可以自动扩展容量的动态数组。它重载了Index运算符，可以通过中括号取下标的形式访问内部成员。它还重载了Deref/DerefMut运算符，因此可以自动被解引用为数组切片。

常见用法示例如下：

```
fn main() {

    // 常见的几种构造Vec的方式
    // 1. new() 方法与 default() 方法一样,构造一个空的Vec
    let v1 = Vec::<i32>::new();
    // 2. with_capacity() 方法可以预先分配一个较大空间,避免插入数据的时候动态扩容
    let v2 : Vec<String> = Vec::with_capacity(1000);
    // 3. 利用宏来初始化,语法跟数组初始化类似
    let v3 = vec![1,2,3];

    // 插入数据
```

```
    let mut v4 = Vec::new();
    // 多种插入数据的方式
    v4.push(1);
    v4.extend_from_slice(&[10,20,30,40,50]);
    v4.insert(2, 100);
    println!("capacity: {} length: {}", v4.capacity(), v4.len());

    // 访问数据
    // 调用 IndexMut 运算符,可以写入数据
    v4[5] = 5;
    let i = v4[5];
    println!("{}", i);
    // Index 运算符直接访问,如果越界则会造成 panic,而 get 方法不会,因为它返回一个 Option<T>
    if let Some(i) = v4.get(6) {
        println!("{}", i);
    }
    // Index 运算符支持使用各种 Range 作为索引
    let slice = &v4[4..];
    println!("{:?}", slice);
}
```

以上示例包含了Vec中最常见的一些操作，还有许多有用的方法，不方便在本书一一罗列，各位读者可以参考标准文档。

一个Vec中能存储的元素个数最多为std：：usize：：MAX个，超过了会发生panic。因为它记录元素个数，用的就是usize类型。如果我们指定元素的类型是0大小的类型，那么，这个Vec根本不需要在堆上分配任何空间。另外，因为Vec里面存在一个指向堆上的指针，它永远是非空的状态，编译器可以据此做优化，使得size_of：：<Option<Vec<T>>>（）==size_of：：<Vec<T>>（）。示例如下：

```
struct ZeroSized{}

fn main() {
    let mut v = Vec::<ZeroSized>::new();
    println!("capacity:{} length:{}", v.capacity(), v.len());

    v.push(ZeroSized{});
    v.push(ZeroSized{});
    println!("capacity:{} length:{}", v.capacity(), v.len());

// p 永远指向 align_of::<ZeroSized>(),不需要调用 allocator
    let p = v.as_ptr();
    println!("ptr:{:p}", p);

    let size1 = std::mem::size_of::<Vec<i32>>();
    let size2 = std::mem::size_of::<Option<Vec<i32>>>();
    println!("size of Vec:{} size of option vec:{}", size1, size2);
}
```

编译执行，可得结果为：

```
capacity:18446744073709551615 length:0
capacity:18446744073709551615 length:2
ptr:0x1
size of Vec:24 size of option vec:24
```

将来类似Vec的这些容器，还会像C++一样，支持一个新的泛型参数，允许用户自定义allocator。这部分目前还没有定下来，因此就不深入讨论了。

24.1.2 VecDeque

VecDeque是一个双向队列。在它的头部或者尾部执行添加或者删除操作，都是效率很高的。它的用法和Vec非常相似，主要是多了pop_front（）push_front（）等方法。

基本用法示例如下：

```
use std::collections::VecDeque;

fn main() {
    let mut queue = VecDeque::with_capacity(64);
// 向尾部按顺序插入一堆数据
    for i in 1..10 {
        queue.push_back(i);
    }
// 从头部按顺序一个个取出来
    while let Some(i) = queue.pop_front() {
        println!("{}", i);
    }
}
```

24.1.3 HashMap

HashMap<K，V，S>是基于hash算法的存储一组键值对（key-value-pair）的容器。其中，泛型参数K是键的类型，V是值的类型，S是哈希算法的类型。S这个泛型参数有一个默认值，平时我们使用的时候不需要手动指定，如果有特殊需要，则可以自定义哈希算法。

hash算法的关键是，将记录的存储地址和key之间建立一个确定的对应关系。这样，当想查找某条记录时，我们根据记录的key，通过一次函数计算，就可以得到它的存储地址，进而快速判断这条记录是否存在、存储在哪里。因此，Rust的HashMap要求，key要满足Eq+Hash的约束。Eq trait代表这个类型可以作相等比较，并且一定满足下列三个性质：

·自反性——对任意a，满足a==a；

·对称性——如果a==b成立，则b==a成立；

·传递性——如果a==b且b==c成立，则a==c成立。

而Hash trait更重要，它的定义如下：

```
trait Hash {
    fn hash<H: Hasher>(&self, state: &mut H);
    ...
}

trait Hasher {
    fn finish(&self) -> u64;
    fn write(&mut self, bytes: &[u8]);
    ...
}
```

如果一个类型，实现了Hash，给定了一种哈希算法Hasher，就能计算出一个u64类型的哈希值。这个哈希值就是HashMap中计算存储位置的关键。

一般来说，手动实现Hash trait的写法类似下面这样：

```
struct Person {
    first_name: String,
    last_name: String,
}

impl Hash for Person {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.first_name.hash(state);
        self.last_name.hash(state);
    }
}
```

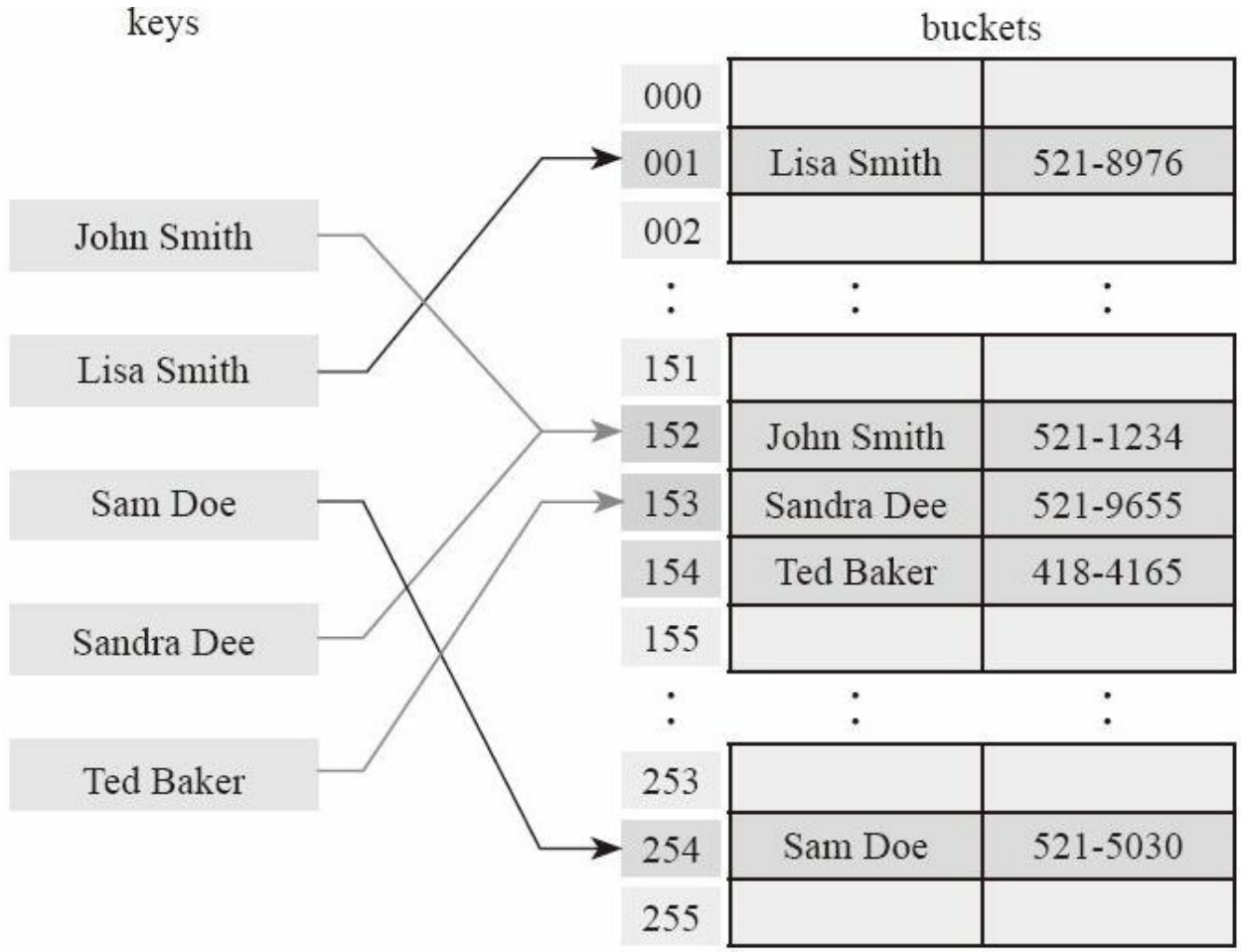
这个hash方法基本上就是重复性的代码，因此编译器提供了自动derive来帮我们实现。下面这种写法才是平时见得最多的：

```
#[derive(Hash)]
struct Person {
    first_name: String,
    last_name: String,
}
```

一个完整的使用HashMap的示例如下：

```
use std::collections::HashMap;

#[derive(Hash, Eq, PartialEq, Debug)]
struct Person {
    first_name: String,
    last_name: String,
}

impl Person {
    fn new(first: &str, last: &str) -> Self {
        Person {
            first_name: first.to_string(),
            last_name: last.to_string(),
        }
    }
}
fn main() {
    let mut book = HashMap::new();
    book.insert(Person::new("John", "Smith"), "521-8976");
    book.insert(Person::new("Sandra", "Dee"), "521-9655");
    book.insert(Person::new("Ted", "Baker"), "418-4165");

    let p = Person::new("John", "Smith");

    // 查找键对应的值
    if let Some(phone) = book.get(&p) {
        println!("Phone number found: {}", phone);
    }

    // 删除
    book.remove(&p);

    // 查询是否存在
    println!("Find key: {}", book.contains_key(&p));
}
```

HashMap的查找、插入、删除操作的平均时间复杂度都是O（1）。在这个例子中，HashMap内部的存储状态类似下图所示：



除了上面例子中演示的这些方法外，HashMap还设计了一种叫作entry的系列API。考虑这样的一种场景，我们需要先查看某个key是否存在，然后再做插入或删除操作。这种时候，如果我们用传统的API，那么就需要执行两遍查找的操作：

```
if map.contains_key(key) { // 执行了一遍hash查找的工作
    map.insert(key, value);  // 又执行了一遍hash查找的工作
}
```

如果我们用entry API，则可以提高效率，而且代码也更流畅：

```
map.entry(key).or_insert(value);
```

HashMap也实现了迭代器，使我们可以直接遍历整个容器，这部分内容放到后面迭代器部分讲解。

HashMap里面，key存储的位置跟它本身的值密切相关，如果key本身变了，那么它存放的位置也需要相应变化。所以，HashMap设计的各种API中，指向key的借用一般是只读借用，防止用户修改它。但是，只读借用并不能完全保证它不被修改，读者应该能想到，只读借用依然可以改变具备内部可变性特点的类型。下面这个示例演示了，如果我们动态修改了HashMap中的key值，会出现什么后果：

```
use std::hash::{Hash, Hasher};
use std::collections::HashMap;
use std::cell::Cell;

#[derive(Eq, PartialEq)]
struct BadKey {
    value: Cell<i32>
}

impl BadKey {
    fn new(v: i32) -> Self {
        BadKey { value: Cell::new(v) }
    }
}

impl Hash for BadKey {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.value.get().hash(state);
    }
}

fn main() {
    let mut map = HashMap::new();
    map.insert(BadKey::new(1), 100);
    map.insert(BadKey::new(2), 200);

    for key in map.keys() {
        key.value.set(key.value.get() * 2);
    }

    println!("Find key 1:{:?}", map.get(&BadKey::new(1)));
    println!("Find key 2:{:?}", map.get(&BadKey::new(2)));
    println!("Find key 4:{:?}", map.get(&BadKey::new(4)));
}
```

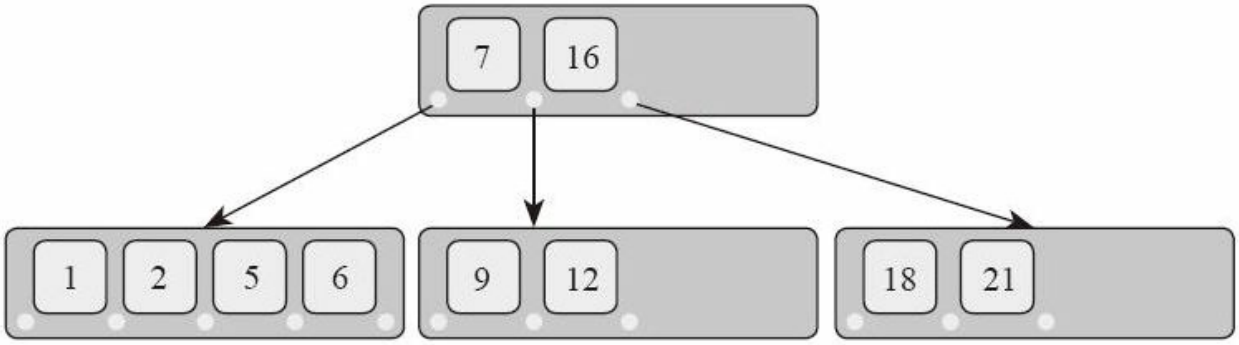
在上面的示例中，我们设计了一个具备内部可变性的类型作为key。然后直接在容器内部把它的值改变，接下来继续做查找。可以看到，我们再也找不到这几个key了，不论是用修改前的key值，还是用修改后的key值，都找不到。这种错误属于逻辑错误，是编译器无法静态检查出来的。所有关联容器，比如HashMap、HashSet、BTreeMap、BTreeSet等都存在这样的情况，使用者需要自己避免。

标准库中的HashSet类型就不再展开讲解了，它跟HashMap非常类似，主要区别在于它只有key没有value，从源码定义我们也可以看到：

```
struct HashSet<T, S = RandomState> {
    map: HashMap<T, (), S>,
}
```

24.1.4 BTreeMap

BTreeMap<K，V>是基于B树数据结构的存储一组键值对（key-value-pair）的容器。它跟HashMap的用途相似，但是内部存储的机制不同。B树的每个节点包含多个连续存储的元素，以及多个子节点。B树的结构如下图所示：



BTreeMap对key的要求是满足Ord约束，即具备“全序”特征。前文中关于Hash-Map的示例也可以用BTreeMap重写，从中我们可以看到它的基本用法与HashMap很相似：

```
use std::collections::BTreeMap;

#[derive(Ord, PartialOrd, Eq, PartialEq, Debug)]
struct Person {
    first_name: String,
    last_name: String,
}

impl Person {
    fn new(first: &str, last: &str) -> Self {
        Person {
            first_name: first.to_string(),
            last_name: last.to_string(),
        }
    }
```

```
}
fn main() {
    let mut book = BTreeMap::new();
    book.insert(Person::new("John", "Smith"), "521-8976");
    book.insert(Person::new("Sandra", "Dee"), "521-9655");
    book.insert(Person::new("Ted", "Baker"), "418-4165");

    let p = Person::new("John", "Smith");

    // 查找键对应的值
    if let Some(phone) = book.get(&p) {
        println!("Phone number found: {}", phone);
    }

    // 删除
    book.remove(&p);

    // 查询是否存在
    println!("Find key: {}", book.contains_key(&p));
}
```

同样，BTreeMap也实现了entry API。BTreeMap也实现了迭代器，同样可以直接遍历。但是HashMap在遍历的时候，是不保证遍历结果顺序的，而BTreeMap自动把数据排好序了，它遍历结果一定是按固定顺序的。

BTreeMap比HashMap多的一项功能是，它不仅可以查询单个key的结果，还可以查询一个区间的结果，示例如下：

```
use std::collections::BTreeMap;

fn main() {
    let mut map = BTreeMap::new();
    map.insert(3, "a");
    map.insert(5, "b");
    map.insert(8, "c");
    for (k, v) in map.range(2..6) {
        println!("{} : {}", k, v);
    }
}
```

执行结果是：

```
3 : a
5 : b
```

当然，我们还可以使用其他的Range类型，如RangeInclusive等，在

此就不赘述了。

对应的，标准库中的BTreeSet类型就不再展开讲解了，它跟BTreeMap非常类似，主要区别在于它只有key没有value，从源码定义我们也可以看到：

```
struct BTreeSet<T> {
    map: BTreeMap<T, ()>,
}
```

24.2 迭代器

迭代器是Rust的一项重要功能。Rust的迭代器是指实现了Iterator trait的类型。这个Iterator trait的定义如下：

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

它最主要的一个方法就是next（），返回一个Option<Item>。一般情况返回Some（Item）；如果迭代完成，就返回None。

24.2.1 实现迭代器

接下来我们试一下如何实现一个迭代器。假设我们的目标是，这个迭代器会生成一个从1到100的序列。我们需要创建一个类型，这里使用struct，它要实现Iterator trait。注意到每次调用next方法的时候，它都返回不同的值，所以它一定要有一个成员，能记录上次返回的是什么。完整代码如下：

```
use std::iter::Iterator;

struct Seq {
    current : i32
}

impl Seq {
    fn new() -> Self {
        Seq { current: 0 }
    }
}

impl Iterator for Seq {
    type Item = i32;

    fn next(&mut self) -> Option<i32> {
        if self.current < 100 {
            self.current += 1;
            return Some(self.current);
        } else {
            return None;
```

```
        }
    }
}

fn main() {
    let mut seq = Seq::new();
    while let Some(i) = seq.next() {
        println!("{}", i);
    }
}
```

编译执行，可见结果和预期一样。

24.2.2 迭代器的组合

Rust标准库有一个命名规范，从容器创造出迭代器一般有三种方法：

·iter（）创造一个Item是&T类型的迭代器；

·iter_mut（）创造一个Item是&mut T类型的迭代器；

·into_iter（）创造一个Item是T类型的迭代器。

比如，用Vec示例如下：

```
fn main() {
    let v = vec![1,2,3,4,5];
    let mut iter = v.iter();
    while let Some(i) = iter.next() {
        println!("{}", i);
    }
}
```

如果迭代器就是这么简单，那么它的用处基本就不大了。Rust的迭代器有一个重要特点，那它就是可组合的（composability）。我们在前面演示的迭代器的定义，实际上省略了很大一部分内容，去官方文档去查一下，可以看到Iterator trait里面还有一大堆的方法，比如nth、map、filter、skip_while、take等等，这些方法都有默认实现，它们可以统称为adapters（适配器）。它们有个共性，返回的是一个具体类型，而这个类型本身也实现了Iterator trait。这意味着，我们调用这些方法可以从一个迭代器创造出一个新的迭代器。

我们用示例来演示一下这些适配器的威力：

```
fn main() {
    let v = vec![1,2,3,4,5,6,7,8,9];
    let mut iter = v.iter()
        .take(5)
        .filter(|&x| x % 2 == 0)
        .map(|&x| x * x)
        .enumerate();
    while let Some((i, v)) = iter.next() {
        println!("{} {}", i, v);
    }
}
```

这种写法很有点“声明式”编程的味道。函数名本身就代表了用户的“意图”，它表达的重点是“我想做什么”，而不是具体怎么做。这个跟C#的linq很相似。如果我们用传统的循环来写这些逻辑，这段代码类似下面这样：

```
fn main() {
    let v = vec![1,2,3,4,5,6,7,8,9];
    let mut iter = v.iter();
    let mut count = 0;
    let mut index = 0;
    while let Some(i) = iter.next() {
        if count < 5 {
            count += 1;
            if (*i) % 2 == 0 {
                let s = (*i) * (*i);
                println!("{} {}", index, s);
                index += 1;
            }
        } else {
            break;
        }
    }
}
```

上面这种写法，源代码更倾向于实现细节。两个版本相比较，迭代器的可读性是不言而喻的。这种抽象相比于直接在传统的循环内部写各种逻辑是有优势的，特别是在后文“并行”的章节中我们可以看到，如果我们想把迭代器改成并行执行是非常容易的事情。而传统的写法涉及细节太多，不太容易改成并行执行。（一个题外话，迭代器的可组合性是一个非常大的优点，新版C++标准中引入了ranges这个库，主要就是为了解决这个问题。）

通过上面分析迭代器的实现原理我们也可以知道：构造一个迭代器本身，是代价很小的行为，因为它只是初始化了一个对象，并不真正产生或消费数据。不论迭代器内部嵌套了多少层，最终消费数据还是要通过调用next（）方法实现的。这个特点，也被称为惰性求值（lazy evaluation）。也就是说，如果用户写了下面这样的代码：

```
let v = vec![1, 2, 3, 4, 5];
v.iter().map(|x| println!("{}", x));
```

实际上是什么事都没做。因为map方法只是把前面一个迭代器包装一下，构造一个新的迭代器而已，没有真正读取容器内部的数据。

24.2.3 for循环

在前面的示例中，我们都是手工直接调用迭代器的next（）方法，然后使用while let语法来做循环。实际上，Rust里面更简洁、更自然地使用迭代器的方式是使用for循环。本质上来说，for循环就是专门为迭代器设计的一个语法糖。for循环可以对针对数组切片、字符串、Range、Vec、LinkedList、HashMap、BTreeMap等所有具有迭代器的类型执行循环，而且还允许我们针对自定义类型实现循环。

```
use std::collections::HashMap;

fn main() {
    let v = vec![1,2,3,4,5,6,7,8,9];
    for i in v {
        println!("{}", i);
    }

    let map : HashMap<i32, char> =
        [(1, 'a'), (2, 'b'), (3, 'c')].iter().cloned().collect();
    for (k, v) in &map {
        println!("{} : {}", k, v);
    }
}
```

那么for循环是怎么做到这一点的呢？原因就是下面这个trait：

```
trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item=Self::Item>;
```

```
    fn into_iter(self) -> Self::IntoIter;
}
```

只要某个类型实现了IntoIterator，那么调用into_iter（）方法就可以得到对应的迭代器。这个into_iter（）方法的receiver是self，而不是&self，执行的是move语义。这么做，可以同时支持Item类型为T、&T或者&mut T，用户有选择的权力。来看看常见的容器是怎样实现这个trait的就明白了：

```
impl<K, V> IntoIterator for BTreeMap<K, V> {
    type Item = (K, V);
    type IntoIter = IntoIter<K, V>;
}
impl<'a, K: 'a, V: 'a> IntoIterator for &'a BTreeMap<K, V> {
    type Item = (&'a K, &'a V);
    type IntoIter = Iter<'a, K, V>;
}
impl<'a, K: 'a, V: 'a> IntoIterator for &'a mut BTreeMap<K, V> {
    type Item = (&'a K, &'a mut V);
    type IntoIter = IterMut<'a, K, V>;
}
```

对于一个容器类型，标准库里面对它impl了三次IntoIterator。当Self类型为BTreeMap的时候，Item类型为（K，V），这意味着，每次next（）方法都是把内部的元素move出来了；当Self类型为&BTreeMap的时候，Item类型为（&K，&V），每次next（）方法返回的是借用；当Self类型为&mut BTreeMap的时候，Item类型为（&K，&mut V），每次next（）方法返回的key是只读的，value是可读写的。

所以，如果有个变量m，其类型为BTreeMap，那么用户可以选择使用m.into_iter（）或者（&m）.into_iter（）或者（&mut m）.into_iter（），分别达到不同的目的。

那么for循环和IntoIterator trait究竟是什么关系呢？下面我们写一个简单的for循环示例：

```
fn do_something(e : &i32) {}

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

    for i in array {
        do_something(i);
    }
```

```
}
```

使用以下编译命令：

```
rustc --unpretty=hir -Z unstable-options test.rs
```

可以看到输出结果为：

```
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
fn do_something(e: &i32) { }

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

    {
        let _result =
            match ::std::iter::IntoIterator::into_iter(array) {
                mut iter =>
                loop  {
                    let mut __next;
                    match ::std::iter::Iterator::next(&mut iter) {
                        ::std::option::Option::Some(val) =>
                        __next = val,
                        ::std::option::Option::None => break ,
                    }
                    let i = __next;
                    { do_something(i); }
                },
            };
        _result
    }
}
```

这说明Rust的for<item>in<container>{<body>}语法结构就是一个语法糖。这个语法的原理其实就是调用<container>.into_iter（）方法来获得迭代器，然后不断循环调用迭代器的next（）方法，将返回值解包，赋值给<item>，然后调用<body>语句块。

所以在使用for循环的时候，我们可以自主选择三种使用方式：

```
// container在循环之后生命周期就结束了,循环过程中的每个item是从container中move出来的
for item in container {}
// 迭代器中只包含container的&型引用,循环过程中的每个item都是container中元素的借用
for item in &container {}
```

```
// 迭代器中包含container的&mut型引用,循环过程中的每个item都是指向container中元素的可变借用
for item in &mut container {}
```

Rust的IntoIterator trait实际上就是for语法的扩展接口。如果我们需要让各种自定义容器也能在for循环中使用，那就可以借鉴标准库中的写法，自行实现这个trait即可。这跟其他语言的设计思路是一样的。比如：C#的foreach语句也可以对自定义类型使用，它的扩展接口就是标准库中定义的IEnumerable接口；Java的for循环的扩展接口是标准库中的Iterable接口；C++的Range-based-for循环也可以使用自定义容器，它约定的是调用容器的begin（）/end（）成员方法。

第25章　生成器

在Rust里面，协程（Coroutine）是编写高性能异步程序的关键设施，生成器（Generator）是协程的基础。本节主要讲解什么是生成器，并简要介绍一下协程。

25.1 简介

生成器的语法很像前面讲过的闭包，但它与闭包有一个区别，即yield关键字。当闭包中有yield关键字的时候，它就不是一个闭包，而是一个生成器。

依然用示例来说话。假设我们要生成一个Fibonacci数列，用生成器可以这样写：

```
// 方案一
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};

fn main() {
    let mut g = || {
        let mut curr : u64 = 1;
        let mut next : u64 = 1;
        loop {
            let new_next = curr.checked_add(next);

            if let Some(new_next) = new_next {
                curr = next;
                next = new_next;
                yield curr; // <-- 新的关键字
            } else {
                return;
            }
        }
    };

    loop {
        unsafe {
            match g.resume() {
                GeneratorState::Yielded(v) => println!("{}", v),
                GeneratorState::Complete(_) => return,
            }
        }
    }
}
```

在这段代码中，构造了一个生成器，它长得跟闭包的样子差不多，区别只是它内部用到了yield关键字。它与closure类似的地方在于，编译器同样会为它生成一个匿名结构体，并实现一些trait，添加一些成员方法。跟closure不同的地方在于，它的成员变量不一样，它实现的trait也不一样。在后面调用它时，不是采用类似闭包的那种调用方式，而是使

用编译器自动生成的成员方法resume（）。resume（）返回结果有两种可能性：一种是Yielded表示生成器内部yield关键字返回出来的东西，此时还可以继续调用resume，还有数据可以继续生成出来；另一种是Complete状态，表示这个生成器已经调用完了，它的值是内部return关键字返回出来的内容，返回了Complete之后就不能再继续调用resume了，否则会触发panic。

生成器最大的特点就是，程序的执行流程可以在生成器和调用者之间来回切换。当我们需要暂时从生成器中返回的时候，就使用yield关键字；当调用者希望再次进入生成器的时候，就调用resume（）方法，这时程序执行的流程是从上次yield返回的那个点继续执行。

上述程序的执行流程很有意思，它是这样的：

·let g=||{...yield...}；这句话是初始化了一个局部变量，它是一个生成器，此时并不执行生成器内部的代码；

·调用g.resume（）方法，此时会调用生成器内部的代码；

·执行到yieldcurr；这条语时，curr变量的值为1，生成器的方法此时会退出，g.resume（）方法的返回值是GeneratorState：：Yielded（1），在main函数中，程序会打印出1；

·循环调用g.resume（）方法，此时再次进入到生成器内部的代码中；

·此时生成器会直接从上次退出的那个地方继续执行，跳转到loop循环的开头，计算curr next new_next这几个变量新的值，然后再到yield curr；这条语句返回；

·如此循环往复，一直到加法计算溢出，生成器调用了return；语句，此时main函数那边会匹配上GeneratorState：：Complete这个分支，程序返回，执行完毕。

25.2 对比迭代器

生成器本质上跟迭代器是很像的。在Rust中，迭代器指的是实现了std：：iter：：Iterator trait的类型：

```
pub trait Iterator {

    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    ...
}
```

这个trait最主要的一个方法就是next方法。每次调用，它都会返回下一个元素，迭代完成，就返回None。使用方法如下：

```
// 假设 it 是一个迭代器变量
while let Some(item) = it.next() {
    do_something(item);
}
```

next方法接受的参数是&mut Self类型。因为它每次调用的时候，都要修改内部的状态，只有这样，下一次调用的时候才会返回不同的内容。如果内部是指针，需要把指针指向容器的下一个元素；如果内部是索引，就需要更新索引的值。

迭代器也可以不指向任何容器，只要它满足Iterator trait这个接口即可。比如std：：ops：：Range这个类型，它代表一个前闭后开的区间，也可以进行迭代，只是每次调用next后它代表的区间就变了。

任何一个生成器，总能找到某种办法改写为功能相同的迭代器。还是以前面的Fibonacci数列为例，如果改成迭代器的样子，该像下面这样写：

```
// 方案二
struct Fibonacci {
    curr: u64,
    next: u64,
```

```
}

impl Iterator for Fibonacci {
    type Item = u64;

    fn next(&mut self) -> Option<u64> {
        // 判断是否会溢出
        let new_next = self.curr.checked_add(self.next);

        if let Some(new_next) = new_next {
            // 先更新内部状态,再返回
            self.curr = self.next;
            self.next = new_next;
            Some(self.curr)
        } else {
            // 加法溢出,停止迭代
            None
        }
    }
}

fn fibonacci() -> Fibonacci {
    Fibonacci { curr: 1, next: 1 }
}

fn main() {
    let mut it = fibonacci();

    while let Some(i) = it.next() {
        println!("{}", i);
    }
}
```

这段代码同样也能实现打印Fibonacci数列的功能。请读者逐行逐字读一下next方法的逻辑，看清楚它是如何记录状态的，理解为什么每次调用next方法都会返回不同的值。这个示例在后文还会继续使用。

迭代器模式是一种典型的“拉”模式，它也经常被称为“惰性求值”（lazy evaluation）。生成器在这一点上与迭代器是一样的，也需要使用者调用方法把数据拉出来。它们一个用的是next方法，一个用的是resume方法，虽然方法的签名有所不同，但使用上差不多。

25.3 对比立即求值

实际上，从代码组织逻辑上来说，迭代器模式已经是相对高阶一点的写法。对于一个刚刚接触编程的初学者来说，用下面这种写法才是最常见的：

```
// 方案三
fn collector() -> Vec<u64> {
    let mut res = vec![];
    let mut curr : u64 = 1;
    let mut next : u64 = 1;
    loop {
        let new_next = curr.checked_add(next);

        if let Some(new_next) = new_next {
            curr = next;
            next = new_next;
            res.push(curr);
        } else {
            break;
        }
    }
    return res;
}

fn main() {
    let collected = collector();
    let mut it = collected.iter();
    while let Some(i) = it.next() {
        println!("{}", i);
    }
}
```

在这个方案中，我们用一个循环把Fibonacci数列提前生成出来了，存储在一个动态数组里，然后再去使用。这种做法可以看作是惰性求值的反向操作，叫作“立即求值”（eager evaluation）。不过，它有性能上的缺点，方案三提前把数据收集起来，缺少了灵活性。如果使用者只需要使用这个序列的前10个数据呢？如果是方案二迭代器的那种写法，使用者可以选择遍历10个元素后就提前break；后面的数据既不需要生产，也不需要消费，还节省了一个临时的占用很大内存空间的容器，这就是“惰性求值”的好处。如果我们把方案三改成方案二迭代器的写法，性能和灵活性更佳，但是需要人工推理：哪些数据是需要存储在迭代器成员中的，哪些是不需要的，进入next方法时如何读取上一次的状态，退出next方法时如何保存这一次的状态等。这些都是“心智负担”。业务

逻辑越复杂，这个负担越严重。

25.4 生成器的原理

25.4.1 生成器原理简介

再回过头来看一下生成器。它实际上是迭代器和立即求值的“杂交”。一方面，它写起来更接近人的思维模型，代码流程清晰，逻辑上更符合直觉；另一方面，它在执行的时候又具备惰性求值的性能优势。那么编译器是如何实现生成器的呢？yield关键字在背后究竟做了什么？

一句话总结，就是编译器把生成器自动转换成了一个匿名类型，然后对这个类型实现了Generator这个trait。这种处理手法和闭包非常相似。和闭包一样，生成器也可以捕获当前环境中的局部变量，并且可以用move做修饰，捕获的环境变量都是当前生成器的成员，捕获规则也与闭包一样。Generator trait是这么定义的：

```
trait Generator {
    type Yield;
    type Return;
// 至少到目前为止,resume方法还不能接受额外参数,这个限制条件以后可能会放宽
// 目前的resume方法还是不稳定版本,以后应该会去掉unsafe,self 的类型也会有所变化,
// 具体参见下一节的自引用类型
    unsafe fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>;
}
```

但是，生成器内部还有额外的成员，那就是跨yield语句存在的局部变量，也都会当成成员变量。这是因为，生成器内部的语句会成为成员函数resume（）的方法体，源代码中的yield语句都被替换成了普通的return语句，且返回的是Generator-State：：Yielded（_）。源代码中的return语句依然是return语句，但返回的是GeneratorState：：Complete（_）。注意生成器有一个特点，就是每次yield退出之后，当前的局部变量会保持当前的值不变，下一次被调用resume再进来执行的时候，会继续从上次yield的那个地方继续执行，局部变量是无须再次初始化的。这就意味着，对于在yield前和yield后都出现过的局部变量，务必要保存它的状态，它的值要存到匿名类型的成员中。

我们再看看最开始那段示例：

```
let mut g = || {
    let mut curr : u64 = 1;
    let mut next : u64 = 1;
    loop {
        let new_next = curr.checked_add(next); // 下轮循环的时候要继续使用 curr next 的值

        if let Some(new_next) = new_next {
            curr = next;
            next = new_next;
            yield curr; // <-- 此处退出
        } else {
            return;
        }
    }
};
```

可以看到，再进入生成器的时候，局部变量curr next的值是马上就需要使用的，因此变量g里面无论如何都要给这两个变量留下位置，保存它们的值。否则，下次再调用resume方法的时候，它们就无法恢复到上次退出时的状态了。而另外一个局部变量new_next则无须保存，因为它没有跨yield存在，所以这个局部变量可以作为成员方法resume内部的局部变量，无须提升为g的局部变量。

编译器把这个生成器处理之后，逻辑如下：

```
// 编译器实际上不是在源码级别做的转换,而是在MIR做的转换,以下代码只是为了说明原理,
// 与真实的编译器转换后的代码并不一致
// 编译器是如何做这个转换的,请参考源码 librustc_mir/transform/generator.rs
// 目前编译器实际上是转换为 struct,此处使用 enum 是为了更方便地演示大概的逻辑
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};

fn main() {
    let mut g = {
        enum __AnonymousGenerator {
            Start{curr : u64, next : u64},
            Yield1{curr : u64, next : u64},
            Done,
        }

        impl Generator for __AnonymousGenerator {
            type Yield = u64;
            type Return = ();

            unsafe fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return>
            {
                use std::mem;
                match mem::replace(self, __AnonymousGenerator::Done) {
                    __AnonymousGenerator::Start{curr, next}
                    | __AnonymousGenerator::Yield1{curr, next} => {
                        let new_next = curr.checked_add(next);
```

```
                        if let Some(new_next) = new_next {
                            *self = __AnonymousGenerator::Yield1{curr: next, next: r
                            return GeneratorState::Yielded(curr);
                        } else {
                            *self = __AnonymousGenerator::Done;
                            return GeneratorState::Complete(());
                        }
                    }

                    __AnonymousGenerator::Done => {
                        panic!("generator resumed after completion")
                    }
                }
            }
        }

        __AnonymousGenerator::Start{ curr: 1, next: 1}
    };

    loop {
        unsafe {
            match g.resume() {
                GeneratorState::Yielded(v) => println!("{}", v),
                GeneratorState::Complete(_) => return,
            }
        }
    }
}
```

可以看到，转换后的代码实际上和迭代器非常相似。所以，生成器实际上是让编译器帮我们自动管理状态：哪些状态应该放到成员变量里面，哪些不需要；退出前如何保存状态，重新进入的时候如何读取上次的状态等，都是编译器帮我们自动做好了的。

如果生成器内部存在多个yield语句呢？比如下面这样：

```
let mut g = || {
    yield 1_i32;
    yield 2_i32;
    yield 3_i32;
    return 4_i32;
};
```

那我们就再引入一个状态，来表达上次已经执行到哪条语句了，下次调用应该从哪条语句开始执行。在进入resume方法的时候，先判断这个状态，然后再跳转即可。

```
let mut g = {
        struct __AnonymousGenerator {
```

```
            state: u32
        }

    impl Generator for __AnonymousGenerator {
        type Yield = i32;
        type Return = i32;

        unsafe fn resume(&mut self) -> GeneratorState<Self::Yield, Self::Return> {
            match self.state {
                0 => { // 从初始状态开始执行
                    self.state = 1;
                    return GeneratorState::Yielded(1);
                }

                1 => { // 上一次返回的是yield 1
                    self.state = 2;
                    return GeneratorState::Yielded(2);
                }

                2 => { // 上一次返回的是yield 2
                    self.state = 3;
                    return GeneratorState::Yielded(3);
                }

                3 => { // 上一次返回的是yield 3
                    self.state = 4;
                    return GeneratorState::Complete(4);
                }

                _ => { // 上一次返回的是return 4
                    panic!("generator resumed after completion")
                }
            }
        }
    }

    __AnonymousGenerator{ state: 0}
};
```

总之，任何一个生成器，总能找到办法将它自动转换为类似迭代器的样子。之所以说是类似，是因为生成器的功能更强大，它的resume（）方法实际上可以设计为携带更多的参数，只是目前的Rust还没有实现，这个需求并不是很紧急而已。

25.4.2 自引用类型

目前的生成器只是一个在nightly版本中存在的、实验性质的功能，它还有一些问题没有解决。最主要的一个问题是如何使得借用跨yield存在。示例如下：

```
#![feature(generators, generator_trait)]

fn main() {
    let _g = || {
        let local = 1;
        let ptr = &local;
        yield local;
        yield *ptr;
    };
}
```

编译，出现编译错误：

```
error[E0626]: borrow may still be in use when generator yields
```

这个错误究竟是什么意思呢？我们可以通过分析生成器的原理来理解这个错误的含义。可以尝试看看这个生成器剥掉语法糖之后的样子。注意到，第一个yield之后变量ptr依然被使用，且local这个变量也还存在，那么意味着我们要在生成的匿名类型的成员中，保存ptr和local这两个变量。再加上一个成员变量记录yield的位置信息，我们可以设计下面这样的匿名结构体：

```
struct __Generator__ {
    local: i32,
    ptr: &i32,
    state: u32,
}
```

针对这个类型实现Generator这个trait，基本上就等同于上面那段程序剥掉语法糖之后的效果。

现在就可以更清楚地看到具体问题在哪里了。这里的关键点是：一个结构体类型内部出现了一个成员引用另外一个成员的现象。这种类型被称为“自引用类型”（Self-Referential Type）。目前的Rust，对自引用类型有很多限制。因为这个类型会破坏Rust的一个基本假设：任何类型都是可移动的。这个假设让Rust的移动语义变得非常清晰简单（主要跟C++对比）。但是自引用类型在移动的时候会出问题。原本成员ptr是指向成员变量local的，如果这个结构体整体发生了移动，ptr指针的值保持不变，local的位置却发生了变化，那么就会制造出悬空指针。所以，目前的Rust是不允许这种情况出现的，这种代码会被生命周期检查禁止掉。这就是上面那段示例代码无法编译通过的深层原因。

但是自引用现象未必就一定不安全。假如构成自引用之后这个对象就永远不再移动，那么它其实是没问题的，也不会有悬空指针之类的情况出现。在写生成器的时候会很容易出现自引用对象，如果完全禁止这种行为，会非常影响用户体验。如何让用户有权创建自引用的生成器，同时又能避免安全性问题呢？Rust设计组通过巧妙的设计做到了这一点。主要想法是：

·应该允许用户创建自引用生成器，因为在调用resume方法之前的移动都是没问题的，毕竟这个时候它内部的许多成员都是未初始化状态；

·一旦resume被调用过了，以后就不能再移动这个对象了，因为这时候指针和被指向的对象很可能已经初始化好了，再发生移动就会造成内存不安全。

具体来说，设计组会做以下改变。

·标准库引入一个新的智能指针类型PinMut<'a，T>，它可以指向一个T类型的对象。它的作用是，当这个指针存在的时候，它所指向的对象是不可移动的。

·允许更多的智能指针类型作为self变量的类型，这样我们可以指定resume方法的第一个参数是self：PinMut<Self>类型，而不是&mut self了。

这样，就可以从逻辑上保证用户调用resume方法之前，一定先构造出一个PinMut<XXGenerator>的指针变量。这样，在这个变量存在的期间，生成器就无法移动，调用resume必须通过这个指针来完成。有了这个保证，resume方法前面的unsafe修饰也就可以去掉了。预计这个设计到2018年下半年就可以稳定下来。

另外，生成器本身并不是直接面向广大用户的接口。用户真正需要的是完成异步任务。实际上，“协程”才是最终用户用得最多的东西。生成器只是实现协程的一个底层工具。最终，协程库会把所有这些PinMut指针之类的事情封装管理起来。

25.5 协程简介

Rust设计这个生成器，主要目的在于，基于生成器设计一套协程（Coroutine）的方案，从而方便编写大规模高性能异步程序。这个功能也是Rust设计组2018年要解决的主要问题之一，预计要到2018年年底才能正式稳定下来。到目前为止，依然只是一个实验性质的不稳定功能。

所谓协程，指的是一种用户态的非抢占式的多任务机制。它也可以实现多任务并行。跟线程相比，它的最大特点是它不是被内核调度的，而是由任务自己进行协作式的调度。协程的实现方案一般可以分为stackful以及stackless两种。Rust的协程采用的是stackless coroutine的设计思路。

在Rust语言和标准库中，只引入了极少数的关键字、trait和类型。async和await关键字是目前许多语言都采用的主流方案，使用关键字而不是用宏来做API，有助于社区的统一性，避免不同的异步方案使用完全不一样的用户API。引入关键字使用的是edition方案，所以不会造成代码不兼容问题。标准库中只有极少数必须的类型，这也是Rust一贯的设计思路。但凡是可以在第三方库中实现的，一律在第三方库中实现，哪怕这个库本来就是官方核心组维护的，这样做可以让这个库的版本升级更灵活，有助于标准库的稳定性。

Rust的协程设计，核心是async和await两个关键字，以及Future这个trait：

```
pub trait Future {
    type Output;
    fn poll(self: PinMut<Self>, cx: &mut Context) -> Poll<Self::Output>;
    ......
}
```

Future这个trait代表的是异步执行。它里面最重要的一个方法是poll，意思是，查看当前Future的状态，这个方法的返回类型如下：

```
pub enum Poll<T> {
    Ready(T),
    Pending,
```

```
}
```

对于一个实现了Future Trait的类型，每次调用这个poll方法，其实就是查看一下这个对象当前的状态是什么，该状态可以为正在执行或者已经执行完毕。

Future可以组合，一个Future可以由其他的一个或者多个Future包装而成。跟我们已经见过的迭代器Iterator很像。比如，我们可以实现一个新的Future，它的结果是多个Future按顺序执行得到的。或者，实现一个Future，它的结果是两个子Future中先返回的那个。Future组合的方式可以非常灵活。

然后我们还需要一个调度器Executor，标准库中有一个Executor的trait。它的具体实现可以由第三方库来实现。它应该有一个主事件循环，不断调用最外层每个收到了事件通知的Future的poll方法，外层的Future的poll方法被调用时，它就会调用内层Future的poll方法，不断嵌套。如果这个Future处于Pending状态，那么这个Future就应该设置好自己需要监听的事件信息，然后马上返回，放弃占用CPU。等到合适的事件发生时，调度器则应该再次调用这个Future的poll方法，驱动这个Future从上次退出的那里继续往下执行。

大家可以看到，Future跟Generator一样，具备同样的特性，也就是说可以在某个地方主动中断执行，待下一次再进来的时候，刚好可以从上次退出的地方恢复执行。这就是为什么Rust的Future最终是基于Generator实现的。在Rust里面，Generator是Future的基础设施。

关于这个Future trait，另外一个需要注意的点是，它的self参数是PinMut<Self>类型，而Generator的resume方法的self参数是&mut Self类型。这个PinMut类型，也是一个智能指针，它的目的主要就是保证它所指向的对象无法被move。而&mut Self类型是无法保证这一点的。举个例子，大家还记得std：：mem：：swap方法吗？

```
pub fn swap<T>(x: &mut T, y: &mut T)
```

对于任意两个T类型的对象，如果我们拥有指向它们的&mut T型指针，就可以把它们互换位置。这个操作就相当于把这两个对象都move到了其他地方。如果这两个对象存在自引用的现象，那么这个swap操作

就可以造成它们内部出现野指针。而PinMut<T>类型就不存在这样的问题。PinMut<T>还实现了DerefMut trait，所以它依然有权调用那些需要&mut Self的成员方法。正因为PinMut保证了指向的对象不可move，所以这个poll方法就可以不用unsafe修饰了。

一般情况下，实现任务调度以及为通过各种异步操作实现Future trait并不是最终用户关注的问题，这些应该都已经被网络开发框架完成，比如tokio。大部分用户需要关注的是如何利用这些框架完成业务逻辑。用户此时用得最多的是async和await关键字。在最新的nightly版本中，只有async关键字的实现已完成，await关键字还存在争议，它目前依然是使用宏来实现的。一个基本的使用示例如下所示：

```
async fn async_fn(x: u8) -> u8 {
    let msg = await!(read_from_network());
    let result = await!(calculate(msg, x));
    result
}
```

在这个示例中，假设read_from_network（）以及calculate（）函数都是异步的。最外层的async_fn函数当然也是异步的。当代码执行到await！（read_from_network（））里面的时候，发现异步操作还没有完成，它会直接退出当前这个函数，把CPU让给其他任务执行。当这个数据从网络上传输完成了，调度器会再次调用这个函数，它会从上次中断的地方恢复执行。所以用async/await的语法写代码，异步代码的逻辑在源码组织上跟同步代码的逻辑差别并不大。这里面状态保存和恢复这些琐碎的事情，都由编译器帮我们完成了。

下面给大家解释一下async和await分别做了什么事情。

async关键字可以修饰函数、闭包以及代码块。对于函数：

```
async fn f1(arg: u8) -> u8 {}
```

实际上等同于：

```
fn f1(arg: u8) -> impl Future<Output = u8> {}
```

这两种写法实际上是一模一样的。凡是被async修饰的函数，返回的都是一个实现了Future trait的类型。由async修饰的闭包也是一样的。async代码块同样类似。它相当于创建了一个语句块表达式，这个表达式的返回类型是impl Future。async关键字不仅对函数签名做了一个改变，而且对函数体也自动做了一个包装，被async关键字包起来的部分，会自动产生一个Generator，并把这个Generator包装成一个满足Future约束的结构体。在函数体中用户需要返回的是Future：：Output类型。

对于await这个宏，我们可以在标准库中看到它的实现：

```
macro_rules! await {
    ($e:expr) => { {
        let mut pinned = $e;
        let mut pinned = unsafe { $crate::mem::PinMut::new_unchecked(&mut pinned) };
        loop {
            match $crate::future::poll_in_task_cx(&mut pinned) {
                $crate::task::Poll::Pending => yield,
                $crate::task::Poll::Ready(x) => break x,
            }
        }
    } }
}
```

从语法上来讲，await一定只能在async函数或代码块中出现，所以它实际上是被包在一个Generator里面的。await这个宏的逻辑也很简单，await！（another_future）所做的事情就是，先构造一个PinMut指针指向another_future，然后调用another_future的poll方法。如果其处于Pending状态则yield，暂时退出这个Future。每当调度器恢复它的执行时，它都会继续调用poll方法，直到处于Ready状态，这时候这个await就算是执行完毕了，继续执行后面的语句。

下面我们看一下，如果基于async/await写程序，看起来会是什么样子：

```
async fn fetch_rust_lang(client: hyper::Client) -> io::Result<String> {
    let response = await!(client.get("https://www.rust-lang.org"))?;
    if !response.status().is_success() {
        return Err(io::Error::new(io::ErrorKind::Other, "request failed"))
    }
    let body = await!(response.body().concat())?;
    let string = String::from_utf8(body)?;
    Ok(string)
}
```

可以看到，使用async/await来写异步逻辑，一方面可以保证高效率，另一方面，代码流程还是跟普通的同步逻辑类似，比较符合直觉。当然，我们也可以不用这个语法，通过Future的各种adapter的组合来完成同样的功能，这就跟上文中的对比一样，其与async/await的区别，就与手写Iterator和Generator之间的区别一样。

第26章　标准库简介

除了前面介绍过的容器、迭代器之外，标准库还提供了一系列有用的类型、函数、trait等。本章挑选其中比较常见的一部分简单介绍。

26.1 类型转换

Rust给我们提供了一个关键字as用于基本类型的转换。但是除了基本类型之外，还有更多的自定义类型，它们之间也经常需要做类型转换。为此，Rust标准库给我们提供了一系列的trait来辅助抽象。

26.1.1 AsRef/AsMut

AsRef这个trait代表的意思是，这个类型可以通过调用as_ref方法，得到另外一个类型的共享引用。它的定义如下：

```
pub trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}
```

同理，AsMut有一个as_mut方法，可以得到另外一个类型的可读写引用：

```
pub trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) -> &mut T;
}
```

比如说，标准库里面的String类型，就针对好几个类型参数实现了AsRef trait：

```
impl AsRef<str> for String
impl AsRef<[u8]> for String
impl AsRef<OsStr> for String
impl AsRef<Path> for String
```

AsRef这样的trait很适合用在泛型代码中，为一系列类型做统一抽象。比如，我们可以写一个泛型函数，它接受各种类型，只要可以被转换为&[u8]即可：

```
    fn iter_bytes<T: AsRef<[u8]>>(arg: T) {
```

```
        for i in arg.as_ref() {
            println!("{}", i);
        }
    }

    fn main() {
        let s: String = String::from("this is a string");
        let v: Vec<u8> = vec![1,2,3];
        let c: &str = "hello";
// 相当于函数重载。只不过基于泛型实现的重载,一定需要重载的参数类型满足某种共同的约束
        iter_bytes(s);
        iter_bytes(v);
        iter_bytes(c);
    }
```

26.1.2 Borrow/BorrowMut

Borrow这个trait设计得与AsRef非常像。它是这样定义的：

```
pub trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

可以说，除了名字之外，它和AsRef长得一模一样。但它们的设计意图不同。比如，针对String类型，它只实现了一个Borrow<str>：

```
impl Borrow<str> for String
```

这是因为这个trait一般被用于实现某些重要的数据结构，比如HashMap：

```
impl HashMap {
pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
        where K: Borrow<Q>,
            Q: Hash + Eq
        {}
}
```

和BTreeMap：

```
impl BTreeMap {
pub fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
```

```
        where K: Borrow<Q>,
            Q: Ord
        {}
}
```

所以，它要求borrow（）方法返回的类型，必须和原来的类型具备同样的hash值，以及排序。这是一个约定，如果实现Borrow trait的时候违反了这个约定，那么把这个类型放到HashMap或者BTreeMap里面的时候就可能出现问题。

26.1.3 From/Into

AsRef/Borrow做的类型转换都是从一种引用&T到另一种引用&U的转换。而From/Into做的则是从任意类型T到U的类型转换：

```
pub trait From<T> {
    fn from(T) -> Self;
}

pub trait Into<T> {
    fn into(self) -> T;
}
```

显然，From和Into是互逆的一组转换。如果T实现了From<U>，那么U理应实现Into<T>。因此，标准库里面提供了这样一个实现：

```
impl<T, U> Into<U> for T where U: From<T>
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

用自然语言描述，意思就是：如果存在U：From<T>，则实现T：Into<U>。

正是因为标准库中已经有了这样一个默认实现，我们在需要给两个类型实现类型转换的trait的时候，写一个From就够了，Into不需要自己手写。

比如，标准库里面已经给我们提供了这样的转换：

```
impl<'a> From<&'a str> for String
```

这意味着&str类型可以转换为String类型。我们有两种调用方式：一种是通过String：：from（&str）来使用，一种是通过&str：：into（）来使用。它们的意思一样：

```
fn main() {
    let s: &'static str = "hello";
    let str1: String = s.into();
    let str2: String = String::from(s);
}
```

另外，由于这几个trait很常用，因此Rust已经将它们加入到prelude中。在使用的时候我们不需要写use std：：convert：：From；这样的语句了，包括AsRef、AsMut、Into、From、ToOwned等。具体可以参见libstd/prelude/v1.rs源代码的内容。

标准库中还有一组对应的TryFrom/TryInto两个trait，它们是为了处理那种类型转换过程中可能发生转换错误的情况。因此，它们的方法的返回类型是Result类型。

26.1.4 ToOwned

ToOwned trait提供的是一种更“泛化”的Clone的功能。Clone一般是从&T类型变量创造一个新的T类型变量，而ToOwned一般是从一个&T类型变量创造一个新的U类型变量。

在标准库中，ToOwned有一个默认实现，即调用clone方法：

```
impl<T> ToOwned for T
    where T: Clone
{
    type Owned = T;
    fn to_owned(&self) -> T {
        self.clone()
    }

    fn clone_into(&self, target: &mut T) {
```

```
            target.clone_from(self);
        }
    }
```

但是，它还对一些特殊类型实现了这个trait。比如：

```
impl<T: Clone> ToOwned for [T] {
    type Owned = Vec<T>;
}
impl ToOwned for str {
    type Owned = String;
}
```

而且，很有用的类型Cow也是基于ToOwned实现的：

```
pub enum Cow<'a, B>
where
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

26.1.5 ToString/FromStr

ToString trait提供了其他类型转换为String类型的能力。

```
pub trait ToString {
    fn to_string(&self) -> String;
}
```

一般情况下，我们不需要自己为自定义类型实现ToString trait。因为标准库中已经提供了一个默认实现：

```
impl<T: fmt::Display + ?Sized> ToString for T {
    #[inline]
    default fn to_string(&self) -> String {
        use core::fmt::Write;
        let mut buf = String::new();
        buf.write_fmt(format_args!("{}", self))
           .expect("a Display implementation return an error unexpectedly");
        buf.shrink_to_fit();
        buf
```

```
    }
}
```

这意味着，任何一个实现了Display trait的类型，都自动实现了ToString trait。而Display trait是可以自动derive的，我们只需要为类型添加一个attribute即可。

FromStr则提供了从字符串切片&str向其他类型转换的能力。

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

因为这个转换过程可能出错，所以from_str方法的返回类型被设计为Result。

正是因为有了这个trait，所以str类型才有了一个成员方法parse：

```
pub fn parse<F: FromStr>(&self) -> Result<F, F::Err> {  …    }
```

所以我们可以写下面这样非常清晰直白的代码：

```
let four = "4".parse::<u32>();
assert_eq!(Ok(4), four);
```

26.2 运算符重载

Rust允许一部分运算符重载，用户可以让这些运算符支持自定义类型。运算符重载的方式是：针对自定义类型，impl一些在标准库中预定义好的trait，这些trait都存在于std：：ops模块中。比如前面已经讲过了的Deref trait就属于运算符重载。

本章我们以最基本的Add trait来做讲解。Add代表的是加法运算符+重载。它的定义是：

```
trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

它具备一个泛型参数RHS和一个关联类型Output。其中RHS有一个默认值Self。

标准库早已经为基本数字类型实现了这个trait。比如：

```
impl Add<i32> for i32 {
    type Output = i32;
}
```

而且还有：

```
impl<'a> Add<i32> for &'a i32
    type Output = <i32 as Add<i32>>::Output;
impl<'a> Add<&'a i32> for i32
    type Output = <i32 as Add<i32>>::Output;
impl<'a, 'b> Add<&'a i32> for &'b i32
    type Output = <i32 as Add<i32>>::Output;
```

这意味着，不仅i32+i32是允许的，而且i32+&i32、&i32+i32、&i32+&i32这几种形式也都是允许的。它们的返回类型都是i32。

假如我们现在自己定义了一个复数类型，想让它支持加法运算符，

示例如下：

```
use std::ops::Add;

#[derive(Copy, Clone, Debug, PartialEq)]
struct Complex {
    real : i32,
    imaginary : i32,
}

impl Add for Complex {
    type Output = Complex;

    fn add(self, other: Complex) -> Complex {
        Complex {
            real: self.real + other.real,
            imaginary: self.imaginary + other.imaginary,
        }
    }
}

fn main() {
    let c1 = Complex { real: 1, imaginary: 2};
    let c2 = Complex { real: 2, imaginary: 4};
    println!("{:?}", c1 + c2);
}
```

在这个实现中，我们没有指定泛型参数RHS，所以它就采用了默认值，在此示例中就相当于Complex这个类型。同理，如果我们希望让这个复数能支持与更多的类型求和，可以继续写多个impl：

```
impl<'a> Add<&'a Complex> for Complex {
    type Output = Complex;

    fn add(self, other: &'a Complex) -> Complex {
        Complex {
            real: self.real + other.real,
            imaginary: self.imaginary + other.imaginary,
        }
    }
}

impl Add<i32> for Complex {
    type Output = Complex;

    fn add(self, other: i32) -> Complex {
        Complex {
            real: self.real + other,
            imaginary: self.imaginary,
        }
    }
}
```

26.3 I/O

标准库中也提供了一系列I/O相关的功能。虽然功能比较基础，但好在是跨平台的。如果用户需要更丰富的功能，可以去寻求外部的开源库。

26.3.1 平台相关字符串

要跟操作系统打交道，首先需要介绍的是两个字符串类型：OsString以及它所对应的字符串切片类型OsStr。它们存在于std：：ffi模块中。

Rust标准的字符串类型是String和str。它们的一个重要特点是保证了内部编码是统一的utf-8。但是，当我们和具体的操作系统打交道时，统一的utf-8编码是不够用的，某些操作系统并没有规定一定是用的utf-8编码。所以，在和操作系统打交道的时候，String/str类型并不是一个很好的选择。比如在Windows系统上，字符一般是用16位数字来表示的。

为了应付这样的情况，Rust在标准库中又设计了OsString/OsStr来处理这样的情况。这两种类型携带的方法跟String/str非常类似，用起来几乎没什么区别，它们之间也可以相互转换。

举个需要用到OsStr场景的例子：

```
use std::path::PathBuf;

fn main() {
    let mut buf = PathBuf::from("/");
    buf.set_file_name("bar");

    if let Some(s) = buf.to_str() {
        println!("{}", s);
    } else {
        println!("invalid path");
    }
}
```

上面这个例子是处理操作系统中的路径，就必须用OsString/OsStr这

两个类型。PathBuf的set_file_name方法的签名是这样的：

```
fn set_file_name<S: AsRef<OsStr>>(&mut self, file_name: S)
```

它要求，第二个参数必须满足AsRef<OsStr>的约束。而查看str类型的文档，我们可以看到：

```
impl AsRef<OsStr> for str
```

所以，&str类型可以直接作为参数在这个方法中使用。

另外，当我们想把&PathBuf转为&str类型的时候，使用了to_str方法，返回的是一个Option<&str>类型。这是为了错误处理。因为PathBuf内部是用OsString存储的字符串，它未必能成功转为utf-8编码。而想要把&PathBuf转为&OsStr则简单多了，这种转换不需要错误处理，因为它们是同样的编码。

26.3.2 文件和路径

Rust标准库中用PathBuf和Path两个类型来处理路径。它们之间的关系就类似String和str之间的关系：一个对内部数据有所有权，还有一个只是借用。实际上，读源码可知，PathBuf里面存的是一个OsString，Path里面存的是一个OsStr。这两个类型定义在std：：path模块中。

Rust对文件操作主要是通过std：：fs：：File来完成的。这个类型定义了一些成员方法，可以实现打开、创建、复制、修改权限等文件操作。std：：fs模块下还有一些独立函数，比如remove_file、soft_link等，也是非常有用的。

对文件的读写，则需要用到std：：io模块了。这个模块内部定义了几个重要的trait，比如Read/Write。File类型也实现了Read和Write两个trait，因此它拥有一系列方便读写文件的方法，比如read、read_to_end、read_to_string等。这个模块还定义了BufReader等类型。我们可以把任何一个满足Read trait的类型再用BufReader包一下，实现有缓冲的读取。

下面用一个示例来演示说明这些类型的使用方法：

```
use std::io::prelude::*;
use std::io::BufReader;
use std::fs::File;

fn test_read_file() -> Result<(), std::io::Error> {

    let mut path = std::env::home_dir().unwrap();
    path.push(".rustup");
    path.push("settings");
    path.set_extension("toml");

    let file = File::open(&path)?;
    let reader = BufReader::new(file);

    for line in reader.lines() {
        println!("Read a line: {}", line?);
    }

    Ok(())
}

fn main() {
    match test_read_file() {
        Ok(_) => {}

        Err(e) => {
            println!("Error occured: {}", e);
        }
    }
}
```

26.3.3 标准输入输出

前面我们已经多次使用了println！宏输出一些信息。这个宏很方便，特别适合在小程序中随手使用。但是如果你需要对标准输入输出作更精细的控制，则需要使用更复杂一点的办法。

在C++里面，标准输入输出流cin、cout是全局变量。在Rust中，基于线程安全的考虑，获取标准输入输出的实例需要调用函数，分别为std：：io：：stdin（）和std：：io：：stdout（）。stdin（）函数返回的类型是Stdin结构体。这个结构体本身已经实现了Read trait，所以，可以直接在其上调用各种读取方法。但是这样做效率比较低，因为为了线程安全考虑，每次读取的时候，它的内部都需要上锁。提高执行效率的办法是手动调用lock（）方法，在这个锁的期间内多次调用读取操作，来避免多次上锁。

示例如下：

```
use std::io::prelude::*;
use std::io::BufReader;

fn test_stdin() -> Result<(), std::io::Error> {
    let stdin = std::io::stdin();
    let handle = stdin.lock();
    let reader = BufReader::new(handle);

    for line in reader.lines() {
        let line = line?;
        if line.is_empty() {
            return Ok(());
        }
        println!("Read a line: {}", line);
    }

    Ok(())
}

fn main() {
    match test_stdin() {
        Ok(_) => {}

        Err(e) => {
            println!("Error occured: {}", e);
        }
    }
}
```

26.3.4 进程启动参数

大家应该注意到了，Rust的main函数的签名和C/C++不一致。在C/C++里面，一般进程启动参数是直接用指针传递给main函数的，进程返回值是通过main函数的返回值来决定的。

在Rust中，进程启动参数是调用独立的函数std：：env：：args（）来得到的，或者使用std：：env：：args_os（）来得到，进程返回值也是调用独立函数std：：process：：exit（）来指定的。示例如下：

```
fn main() {
    if std::env::args().any(|arg| arg == "-kill") {
        std::process::exit(1);
    }
    for arg in std::env::args() {
        println!("{}", arg);
    }
}
```

同样，标准库只提供最基本的功能。如果读者需要功能更强大、更容易使用的命令行参数解析器，可以到crates.io上搜索相关开源库，clap或者getopts都是很好的选择。

26.4 Any

Rust标准库中提供了一个乞丐版的“反射”功能，那就是std：：any模块。这个模块内，有个trait名字叫作Any。所有的类型都自动实现了Any这个trait，因此我们可以把任何一个对象的引用转为&Any这个trait object，然后调用它的方法。

它可以判断这个对象是什么类型，以及强制转换&Any为某个具体类型。另外，成员函数get_type_id（）暂时要求'static约束，这个限制条件以后会放宽。

基本用法示例如下：

```
#![feature(get_type_id)]

use std::fmt::Display;
use std::any::Any;

fn log<T: Any + Display>(value: &T) {
    let value_any = value as &Any;

    if let Some(s) = value_any.downcast_ref::<String>() {
        println!("String: {}", s);
    }
    else if let Some(i) = value_any.downcast_ref::<i32>() {
        println!("i32: {}", i);
    } else {
        let type_id = value_any.get_type_id();
        println!("unknown type {:?}: {}", type_id, value);
    }
}

fn do_work<T: Any + Display>(value: &T) {
    log(value);
}

fn main() {
    let my_string = "Hello World".to_string();
    do_work(&my_string);

    let my_i32: i32 = 100;
    do_work(&my_i32);

    let my_char: char = '❤';
    do_work(&my_char);
}
```

第四部分　线程安全

Rust不仅在没有自动垃圾回收（Garbage Coll-ection）的条件下实现了内存安全，而且实现了线程安全。Rust编译器可以在编译阶段避免所有的数据竞争（Data Race）问题。这也是Rust的核心竞争力之一。本部分主要讲解Rust是如何实现免疫数据竞争的。

第27章　线程安全

27.1　什么是线程

线程是操作系统能够进行调度的最小单位，它是进程中的实际运作单位，每个进程至少包含一个线程。在多核处理器越来越普及的今天，多线程编程也用得越来越广泛。多线程的优势有：

·容易利用多核优势；

·比单线程反应更敏捷，比多进程资源共享更容易。

多线程编程在许多领域是不可或缺的。但是，多线程并行，非常容易引发数据竞争，而且还非常不容易被发现和debug。下面，我们用C++语言来演示一下什么是数据竞争（这里是故意写的有bug的版本）：

```
#include <iostream>
#include <stdlib.h>
#include <thread>
#include <string>

#define COUNT 1000000
volatile int g_num = 0;

void thread1()
{
    for (int i=0; i<COUNT; i++){
        g_num++;
    }
}

void thread2()
{
    for (int i=0; i<COUNT; i++){
        g_num--;
    }
}

int main(int argc, char* argv[])
{
    std::thread t1(thread1);
    std::thread t2(thread2);
    t1.join();
    t2.join();
```

```
    std::cout << "final value:" << g_num << std::endl;
    return 0;
}
```

我们可以使用g++-pthread-std=c++11 temp.cpp命令编译这段代码。

在这段代码中，我们创建了两个线程。一个线程去修改全局变量global，循环1000000次加1。另外一个线程也去修改全局变量global，循环1000000次减1。如果没有数据竞争的话，这两个线程执行完毕后，数据最终一定是回到初始值0。然而，我们尝试运行后发现，每次执行的结果都不是0，而且每次的结果都不一样。

为什么会发生这样的现象呢？这是因为，为普通变量加1减1这样的操作并非“原子”操作。我们简化一下这个过程，可以将它分为三个步骤：读数据、执行计算、写数据。理想情况下，我们期望的执行流程应该是下面这样的：

Thread 1	Thread 2	读 / 写	变量值
			0
读数据		←	0
加 1			0
写数据		→	1
	读数据	←	1
	减 1		1
	写数据	→	0

然而，线程的调度是不受我们控制的，即便线程1和线程2内部的执行流程不变，只要调度时机发生了变化，结果也会不同。比如实际的执行过程中，有可能是这样的情况：

Thread 1	Thread 2	读 / 写	变量值
			0
读数据		←	0
	读数据	←	0
加 1			0
	减 1		0
写数据		→	1
	写数据	→	−1

根据调度情况的不同，最终的结果也会有所差异，所以我们可以看

到这个程序的执行结果不是0，而且循环次数越多，发生数据竞争的机会也越大。

在传统的系统级编程语言中，写多线程代码很容易出错。而Rust的一大特点就是消除了数据竞争，保证了线程安全。下面介绍Rust中的线程。

27.2 启动线程

Rust标准库中与线程相关的内容在std：：thread模块中。Rust中的线程是对操作系统线程的直接封装。

创建线程的方法为：

```
use std::thread;

thread::spawn(move || {
    // 这里是新建线程的执行逻辑
});
```

默认情况下，新创建的子线程与原来的父线程是分离的关系。也就是说，子线程可以在父线程结束后继续存在，除非父线程是主线程。因为我们知道，如果一个进程的主线程也退出了，这个进程就会终止，其他所有的线程也会随之结束。

如果我们需要等待子线程执行结束，那么可以使用join方法：

```
use std::thread;
// child 的类型是 JoinHandle<T>,这个T是闭包的返回类型
let child = thread::spawn(move || {
    // 子线程的逻辑
});
// 父线程等待子线程结束
let res = child.join();
```

如果我们需要为子线程指定更多的参数信息，那么在创建的时候可以使用Builder模式：

```
use std::thread;

thread::Builder::new().name("child1".to_string()).spawn(move || {
    println!("Hello, world!");
});
```

thread模块还提供了下面几个工具函数。

（1）thread：：sleep（dur：Duration）

使得当前线程等待一段时间继续执行。在等待的时间内，线程调度器会调度其他的线程来执行。

（2）thread：：yield_now（）

放弃当前线程的执行，要求线程调度器执行线程切换。

（3）thread：：current（）

获得当前的线程。

（4）thread：：park（）

暂停当前线程，进入等待状态。当thread：：Thread：：unpark（&self）方法被调用的时候，这个线程可以被恢复执行。

（5）thread：：Thread：：unpark（&self）

恢复一个线程的执行。

以上函数的综合使用见如下示例：

```
use std::thread;
use std::time::Duration;

fn main() {
    let t = thread::Builder::new()
        .name("child1".to_string())
        .spawn(move || {
            println!("enter child thread.");
            thread::park();
            println!("resume child thread");
        }).unwrap();
    println!("spawn a thread");
    thread::sleep(Duration::new(5,0));
    t.thread().unpark();
    t.join();
    println!("child thread finished");
}
```

27.3 免数据竞争

粗看起来，Rust的多线程API很简单。但其实，其表面的简洁之下隐藏着关键的创新设计。正可谓：

胸有激雷而面如平湖者，可拜上将军也！

为了说明Rust在多线程方面的威力，我们来做几个实验，看看如果用多个线程读写同一个变量会发生什么情况。

我们创建一个子线程，用它修改一个外部变量：

```
use std::thread;
fn main() {
    let mut health = 12;

    thread::spawn( || {
        health *= 2;
    });
    println!("{}", health);
}
```

编译，发生错误。错误信息为：

```
error: closure may outlive the current function, but it borrows health, which is owr
```

根据前面的知识可以知道，spawn函数接受的参数是一个闭包。我们在闭包里面引用了函数体内的局部变量，而这个闭包是运行在另外一个线程上，编译器无法肯定局部变量health的生命周期一定大于闭包的生命周期，于是发生了错误。

那我们对这个程序做一个修改，把闭包加上move修饰。再次编译，可见编译错误已经消失。但是执行发现，变量health的值并未发生改变。为什么呢？因为health是Copy类型，在遇到move型闭包的时候，闭包内的health实际上是一份新的复制，外面的变量并没有被真正修改。

如果我们使用的是非Copy类型，又会怎样呢？

```
use std::thread;
fn main() {
    let mut v : Vec<i32> = vec![];

    thread::spawn( || {
        v.push(1);
    });

    println!("{:?}", v);
}
```

编译，出现同样的错误。再次尝试给闭包加上move，还是出现编译错误：

```
error: use of moved value: v
```

这个错误也好理解：既然我们已经把v移动到了闭包里面，那么它在本函数内就不能再继续使用了，因为其所有权已经移走了。

以上这几个试验全部失败了，那我们究竟怎样做才能让一个变量在不同线程中共享呢？

答案是：我们没有办法在多线程中直接读写普通的共享变量，除非使用Rust提供的线程安全相关的设施 。

也就是说，Rust给我们提供了一个重要的安全保证：

The compiler prevents all data races.

“data race”即数据竞争，意思是在多线程程序中，不同线程在没有使用同步的条件下并行访问同一块数据，且其中至少有一个是写操作的情况。

在笔者看来，这是一项革命性 的进步，非常值得关注。

在许多传统（非函数式）的编程语言中，并行程序设计是非常困难的，原因就在于代码中存在大量的共享状态和很多隐藏的数据依赖。程序员必须非常清楚代码的流程，使用合适的策略正确实现并发控制。而

万一某人在某个地方犯了一个小错误，那么这个程序就成了不安全的，而且没有什么静态检查工具可以保证完整无遗漏地将此类问题检查出来。对于一份规模比较大的C/C++源代码，我们没有什么好办法“证明”一个程序是不是“线程安全”的。况且，人非圣贤，孰能无过，就像墨菲定律说的那样：

Anything that can go wrong，will go wrong.——Murphey’s Law

有很多人尝试过很多办法，来从根源上解决数据竞争（Data race）的问题。根据数据竞争的定义，它的发生需要三个条件：

·数据共享——有多个线程同时访问一份数据；

·数据修改——至少存在一个线程对数据做修改；

·没有同步——至少存在一个线程对数据的访问没有使用同步措施。

我们只要让这三个条件无法同时发生即可：

·可以禁止数据共享，比如actor-based concurrency，多线程之间的通信仅靠发送消息来实现，而不是通过共享数据来实现；

·可以禁止数据修改，比如functional programming，许多函数式编程语言严格限制了数据的可变性，而对共享性没有限制。

然而以上设计在许多情况下都有一些性能上的缺陷，无法达到“零开销抽象”的目的。

Rust并没有盲目跟随传统语言的脚步设计。Rust允许存在可变变量，允许存在状态共享，同时也做到了完整无遗漏的 线程安全检查。因为Rust设计的一个核心思想就是“共享不可变，可变不共享”，然后再加上类型系统和合理的API设计，就可以保证共享数据在访问时一定使用了同步措施。Rust既可以支持多线程数据共享的风格，也可以支持消息通信的风格。无论选择哪种方案，编译器都能保证没有数据竞争。

请注意这个区别：我们不是说传统C/C++就无法做到线程安全，而是说，传统C/C++需要依赖程序员不犯错误来保证线程安全；而Rust是由工具自动保证的，这个保证更稳定、更可靠、更有底气。虽然

C/C++里面也有许多静态检查工具，可以辅助我们自动发现一些线程安全问题，但是由于C/C++灵活度太大、自由度太高，因此可以肯定的是没有任何一款静态检查工具可以保证百分百“无遗漏、无误报”的线程安全检查。现在没有，将来也不可能有。所以不得不依赖程序员的水平。在代码规模大到一定程度之后，这种做法是不可靠的，不论一个人实力有多强，总有马虎的时候、疲惫的时候、情绪不良的时候，偶尔犯错是不可避免的，更何况大规模的团队存在管理配合问题、人员流动交接问题等，一不小心就会埋下一个隐患。作为对比，Rust对线程的安全检查是稳定的、可靠的，不因时因地而有所波动，不因代码量的多少或复杂程度而懈怠。这个特点，对于某些对安全性要求很高的场景具有非同寻常的意义。

这个区别赋予了Rust一种特殊的能力，它使Rust的使用者有了更强大的自信，让Rust的使用者有胆量使用更激进的并行优化。

27.4 Send & Sync

下面来简单讲解一下Rust是如何实现免疫数据竞争的。Rust线程安全背后的功臣是两个特殊的trait。

·std：：marker：：Sync

如果类型T实现了Sync类型，那说明在不同的线程中使用&T访问同一个变量是安全的。

·std：：marker：：Send

如果类型T实现了Send类型，那说明这个类型的变量在不同的线程中传递所有权是安全的。

Rust把类型根据Sync和Send做了分类。这样做起什么作用呢？当然是用在“泛型约束”中。Rust中所有跟多线程有关的API，会根据情况，要求类型必须满足Sync或者Send的约束。这样一来，“孙猴子就永远也逃不出如来佛的手掌心”了。你不可能随意在多线程之间共享变量，也不可能在使用多线程共享的时候忘记加锁。除非你使用unsafe，否则不可能写出存在“数据竞争”的代码来。

比如我们最常见的创建线程的函数spawn，它的完整函数签名是这样的：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
```

我们需要注意的是，参数类型F有重要的约束条件F：Send+'static，T：Send+'static。但凡在线程之间传递所有权会发生安全问题的类型，都无法在这个参数中出现，否则就是编译错误。另外，Rust对全局变量也有很多限制，你不可能简单地通过全局变量在多线程中随意共享状态。这样，编译器就会禁止掉可能有危险的线程间共享数据的行为。

在Rust中，线程安全是默认行为，大部分类型在单线程中是可以随意共享的，但是没办法直接在多线程中共享。也就是说，只要程序员不

滥用unsafe，Rust编译器就可以检查出所有具有“数据竞争”潜在风险的代码。凡是通过了编译检查的代码，Rust可以保证，绝对不会出现“线程不安全”的行为。如此一来，多线程代码和单线程代码就有了严格的分野。一般情况下，我们不需要考虑多线程的问题。即便是万一不小心在多线程中访问了原本只设计为单线程使用的代码，编译器也会报错。

第28章　详解Send和Sync

在上文中我们已经提到，Rust实现免疫数据竞争的关键是Send和Sync这两个trait。那么这两个trait究竟表达了什么意思？它们背后是什么原理？我们在本章详细分析。

28.1 什么是Send

根据定义：如果类型T实现了Send trait，那说明这个类型的变量在不同线程中传递所有权是安全的。但这句话对于初学者并不是那么容易理解的。究竟具备什么特点的类型才满足Send约束？本节就来详细分析一下。

如果一个类型可以安全地从一个线程move进入另一个线程，那它就是Send类型。比如：普通的数字类型是Send，因为我们把数字move进入另一个线程之后，两个线程同时执行也不会造成什么安全问题。

更进一步，内部不包含引用的类型，都是Send。因为这样的类型跟外界没有什么关联，当它被move进入另一个线程之后，它所有的部分都跟原来的线程没什么关系了，不会出现并发访问的情况。比如String类型。

稍微复杂一点的，具有泛型参数的类型，是否满足Send大多是取决于参数类型是否满足Send。比如Vec<T>，只要我们能保证T：Send，那么Vec<T>肯定也是Send，把它move进入其他线程是没什么问题的。再比如Cell<T>、RefCell<T>、Option<T>、Box<T>，也都是这种情况。

还有一些类型，不论泛型参数是否满足Send，都是满足Send的。这种类型，可以看作一种“构造器”，把不满足Send条件的类型用它包起来，就变成了满足Send条件的类型。比如Mutex<T>就是这种。Mutex<T>这个类型实际上不关心它内部类型是怎样的，反正要访问内部数据，一定要调用lock（）方法上锁，它的所有权在哪个线程中并不重要，所以把它move到其他线程也是没有问题的。

那么什么样的类型是！Send呢？典型的如Rc<T>类型。我们知道，Rc是引用计数指针，把Rc类型的变量move进入另外一个线程，只是其中一个引用计数指针move到了其他线程，这样会导致不同的线程中的Rc变量引用同一块数据，Rc内部实现没有做任何线程同步处理，这是肯定有问题的。所以标准库中早已指定Rc是！Send。当我们试图在线程边界传递这个类型的时候，就会出现编译错误。

但是相对的是，Arc<T>类型是符合Send的（当然需要T：Send）。

为什么呢？因为Arc类型内部的引用计数用的是“原子计数”，对它进行增减操作，不会出现多线程数据竞争。所以，多个线程拥有指向同一个变量的Arc指针是可以接受的。

28.2 什么是Sync

对应的，Sync的定义是，如果类型T实现了Sync trait，那说明在不同的线程中使用&T访问同一个变量是安全的。这句话也不好理解。下面我们仔细分析一下哪些类型是满足Sync约束的。

显然，基本数字类型肯定是Sync。假如不同线程都拥有指向同一个i32类型的只读引用&i32，这是没什么问题的。因为这个类型引用只能读，不能写。多个线程读同一个整数是安全的。

大部分具有泛型参数的类型是否满足Sync，很多都是取决于参数类型是否满足Sync。像Box<T>、Vec<T>Option<T>这种也是Sync的，只要其中的参数T是满足Sync的。

也有一些类型，不论泛型参数是否满足Sync，它都是满足Sync的。这种类型把不满足Sync条件的类型用它包起来，就变成了满足Sync条件的。Mutex<T>就是这种。多个线程同时拥有&Mutex<T>型引用，指向同一个变量是没问题的。

那么什么样的类型是！Sync呢？所有具有“内部可变性”而又没有多线程同步考虑的类型都不是Sync的。比如，Cell<T>和RefCell<T>就不能是Sync的。按照定义，如果我们多个线程中都持有指向同一个变量的&Cell<T>型指针，那么在多个线程中，都可以执行Cell：：set方法来修改它里面的数据。而我们知道，这个方法在修改内部数据的时候，是没有考虑多线程同步问题的。所以，我们必须把它标记为！Sync。

还有一些特殊的类型，它们既具备内部可变性，又满足Sync约束，比如前面提到的Mutex<T>类型。为什么说Mutex<T>具备内部可变性？大家查一下文档就会知道，这个类型可以通过不可变引用调用lock（）方法，返回一个智能指针MutexGuard<T>类型，而这个智能指针有权修改内部数据。这个做法就跟RefCell<T>的try_borrow_mut（）方法非常类似。区别只是：Mutex：：lock（）方法的实现，使用了操作系统提供的多线程同步机制，实现了线程同步，保证了异步安全；而RefCell的内部实现就是简单的普通数字加减操作。因此，Mutex<T>既具备内部可变性，又满足Sync约束。除了Mutex<T>，标准库中还有RwLock<T>、AtomicBool、AtomicIsize、AtomicUsize、AtomicPtr等类

型，都提供了内部可变性，而且满足Sync约束。

28.3 自动推理

Send和Sync是marker trait。在Rust中，有一些trait是在std：：marker模块中的特殊trait。它们有一个共同的特点，就是内部都没有任何的方法，它们只用于给类型做“标记”。在std：：marker这个模块中的trait，都是给类型做标记的trait。每一种标记都将类型严格切分成了两个组。

我们可以从源码中的src/libcore/marker.rs中看到：

```
unsafe impl Send for .. { }
unsafe impl Sync for .. { }
```

这是一个临时的、特殊的语法，它的含义是：针对所有类型，默认实现了Send/Sync。使用了这种特殊语法的trait叫作OIBIT（Opt-in built-in trait），后来改称为Auto Trait。注意：这个语法是不稳定的，以后会改变。不管怎样，编译器留了一个后门，可以让我们定义Auto Trait。

Auto Trait有一个重要特点，就是编译器允许用户不用手写impl，自动根据这个类型的成员“推理”出这个类型是否满足这个trait。

我们可以手动指定这个类型满足这个trait约束，也可以手动指定它不满足这个trait约束，但是手动指定的时候，一定要用unsafe关键字。

比如，在标准库中就有这样的代码：

```
unsafe impl<T: ?Sized> !Send for *const T { }
unsafe impl<T: ?Sized> !Send for *mut T { }
unsafe impl<'a, T: Sync + ?Sized> Send for &'a T {}
unsafe impl<'a, T: Send + ?Sized> Send for &'a mut T {}
    // 等等
```

使用！Send这种写法表示“取反”操作，这些类型就一定不满足Send约束。

请大家一定要注意unsafe关键字。这个关键字在这里的意思是，编译器自己并没有能力正确地、智能地理解每一个类型的内部实现原理，

并由此判断它是否满足Send或者Sync。它需要程序员来提供这个信息。此时，编译器选择相信程序员的判断。但同时，这两个trait对于“线程安全”至关重要，如果程序员自己在这里判断错了，就可能制造出“线程不安全”的问题。

所以，这里的规则和前面讲的“内存安全”的情况是一样的。某些情况下，程序员需要做底层操作的时候，编译器没有能力判断这部分是不是满足内存安全，就需要程序员把这部分代码用unsafe关键字包起来，由程序员去负责安全性。unsafe关键字的意义不是说这段代码“不安全”，而是说这段代码的安全性编译器自己无法智能检查出来，需要由程序员来保证。

标准库中把所有基本类型，以及标准库中定义的类型，都做了合适的Send/Sync标记。

同时，由于Auto trait这个机制的存在，绝大部分用户创建的自定义类型，本身都已经有了合理的Send/Sync标记，用户不需要手动修改它。只有一种情况例外：用户用了unsafe代码的时候，有些类型就很可能需要手动实现Send/Sync。比如做FFI，在Rust项目中调用C的代码。这种时候，类型内部很可能会包含一些裸指针，各种方法调用也会有许多unsafe代码块。此时，一个类型是否满足Send/Sync就不能依赖Auto Trait机制由编译器推理了，因为它推理出来的结论很可能是错的。程序员需要根据Send/Sync所表达的概念去理解这个类型的逻辑，然后自己判断出它是否满足Send/Sync的约束。在这种情况下，写这个库的程序员就成了实现“线程安全”目标的重要一环。如果写错了，就会对下游用户造成致命的影响，所有依赖于这个库的代码都有可能引发线程不安全。

28.4 小结

Rust语言本身并不知晓“线程”“并发”具体是什么，而是抽象出了一些更高级的概念Send/Sync，用来描述类型在并发环境下的特性。std：：thread：：spawn函数就是一个普通函数，编译器没有对它做任何特殊处理。它能保证线程安全的关键是，它对参数有合理的约束条件。这样的设计使得Rust在线程安全方面具备非常好的扩展性。很多高级的并发模型，在Rust中都可以通过第三方库的形式实现，而且可以保证线程安全特性。只要对外API设计得合理，客户就可以随便使用这些并行库，而不会有数据竞争的风险。

Rust的这个设计实际上将开发者分为了两个阵营，一部分是核心库的开发者，一部分是业务逻辑开发者。对于一般的业务开发者来说，完全没有必要写任何unsafe代码，更没有必要为自己的自定义类型去实现Sync Send，直接依赖编译器的自动推导即可。这个阵营中的程序员可以完全享受编译器和基础库带来的安全性保证，无须投入太多精力去管理细节，极大地减轻脑力负担。

而对于核心库的开发者，则必须对Rust的这套机制非常了解。比如，他们可能需要设计自己的“无锁数据类型”“线程池”“管道”等各种并行编程的基础设施。这种时候，就有必要对这些类型使用unsafe impl Send/Sync设计合适的接口。这些库本身的内部实现是基于unsafe代码做的，它享受不到编译器提供的各种安全检查。相反，这些库本身才是保证业务逻辑代码安全性的关键。

这个区分对整个生态圈是有好处的。跟前面讲的“内存安全”问题类似，需要使用unsafe的代码总是很小的一部分，把这部分代码抽象到独立的库里面，是比较容易测试和验证它的正确性的。而业务逻辑代码才是千变万化的，我们千万不要在这部分代码中用unsafe做各种hack。因此，大部分普通人就能充分享受到少部分精英给我们提供的各种安全性保证（编译器加基础库），放心大胆地做各种并行优化，完全不必担心会制造线程安全问题。

第29章　状态共享

前面我们已经看到，Rust可以阻止在多线程之间不安全的共享状态。本章我们来讲解一下，在Rust中，如何使用标准库安全地实现多线程访问共享变量。

29.1 Arc

Arc是Rc的线程安全版本。它的全称是“Atomic reference counter”。注意第一个单词代表的是atomic而不是automatic。它强调的是“原子性”。它跟Rc最大的区别在于，引用计数用的是原子整数类型。Arc使用方法示例如下：

```
use std::sync::Arc;
use std::thread;

fn main() {
    let numbers: Vec<_> = (0..100u32).collect();
    // 引用计数指针,指向一个 Vec
    let shared_numbers = Arc::new(numbers);

    // 循环创建 10 个线程
    for _ in 0..10 {
    // 复制引用计数指针,所有的 Arc 都指向同一个 Vec
        let child_numbers = shared_numbers.clone();
    // move修饰闭包,上面这个 Arc 指针被 move 进入了新线程中
        thread::spawn(move || {
    // 我们可以在新线程中使用 Arc,读取共享的那个 Vec
            let local_numbers = &child_numbers[..];
    // 继续使用 Vec 中的数据
        });
    }
}
```

这段代码可以正常编译通过。

如果我们把上面代码中的Arc改为Rc类型，就会发生下面的编译错误：

```
error: the trait `std::marker::Send` is not implemented for the type `std::rc::Rc<st
```

因为Rc类型内部的引用计数是普通整数类型，如果多个线程中分别同时持有指向同一块内存的Rc指针，是线程不安全的。这个错误是通过spawn函数的签名检查出来的。spawn要求闭包参数类型满足Send条件，闭包是没有显式impl Send或者Sync的，按照auto trait的推理规则，编译器会检查这个类型所有的成员是否满足Send或者Sync。目前这个闭包参数“捕获”了一个Rc类型，而Rc类型是不满足Send条件的，因此编译器推

理出来这个闭包类型是不满足Send条件的，与spawn函数的约束条件发生了冲突。

查看源码我们可以发现，Rc和Arc这两个类型之间的区别，除了引用计数值的类型之外，主要如下：

```
unsafe impl<T: ?Sized + Sync + Send> Send for Arc<T> {}
unsafe impl<T: ?Sized + Sync + Send> Sync for Arc<T> {}

impl<T: ?Sized> !marker::Send for Rc<T> {}
impl<T: ?Sized> !marker::Sync for Rc<T> {}
```

编译器的推理过程为：u32是Send，得出Unique<u32>是Send，接着得出Vec<u32>是Send，然后得出Arc<Vec<u32>>是Send，最后得出闭包类型是Send。它能够符合spawn函数的签名约束，可以穿越线程边界。如果把共享变量类型变成Cell<u32>，那么Arc<Cell<u32>>依然是不符合条件的。因为Cell类型是不满足Sync的。

这就是为什么有底气Rust给用户提供了两种“引用计数”智能指针。因为用户不可能用错。如果不小心把Rc用在了多线程环境，直接是编译错误，根本不会引发多线程同步的问题。如果不小心把Arc用在了单线程环境也没什么问题，不会有bug出现，只是引用计数增加或减少的时候效率稍微有一点降低。

29.2 Mutex

与前面讲解的Rc类似，根据Rust的“共享不可变，可变不共享”原则，Arc既然提供了共享引用，就一定不能提供可变性。所以，Arc也是只读的，它对外API和Rc是一致的。如果我们要修改怎么办？同样需要“内部可变性”。当然，在多线程环境下，Cell和RefCell已经不能用了，它们的内部实现根本没考虑过多线程的问题，所以它们不满足Sync。但是没关系，反正即使不小心误用了，也是编译错误。这种时候，我们需要用线程安全版本的“内部可变性”，如Mutex和RwLock。

下面我们用一个示例来演示一下Arc和Mutex配合。使用多线程修改共享变量：

```
use std::sync::Arc;
use std::sync::Mutex;
use std::thread;

const COUNT: u32 = 1000000;

fn main() {
    let global = Arc::new(Mutex::new(0));

    let clone1 = global.clone();
    let thread1 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone1.lock().unwrap();
            *value += 1;
        }
    });

    let clone2 = global.clone();
    let thread2 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone2.lock().unwrap();
            *value -= 1;
        }
    });

    thread1.join().ok();
    thread2.join().ok();
    println!("final value: {:?}", global);
}
```

因为我们的闭包用了move关键字修饰，为了避免把global这个引用计数指针move进入闭包，所以在外面先提前复制一份，然后将复制出来的这个指针传入闭包中。这样两个线程就都拥有了指向同一个变量的

Arc指针。

在这个程序中，我们使用两个线程修改同一个整数：一个线程对它进行多次加1，另一个线程对它进行多次减1。这次，我们使用Arc来实现多线程之间的共享，使用Mutex来提供内部可变性。每次需要修改的时候，我们需要调用lock（）方法（或者try_lock）获得锁，然后才能对内部的数据进行读/写操作。因为锁的存在，我们就可以保证整个“读/写”是一个完整的transaction。对于Mutex类型，标准库中有：

```
unsafe impl<T: ?Sized + Send> Send for Mutex<T> { }
unsafe impl<T: ?Sized + Send> Sync for Mutex<T> { }
```

因此，Arc<Mutex<isize>>可以满足Send要求。

Mutex：：lock（）方法的返回类型是LockResult<MutexGuard<T>>：

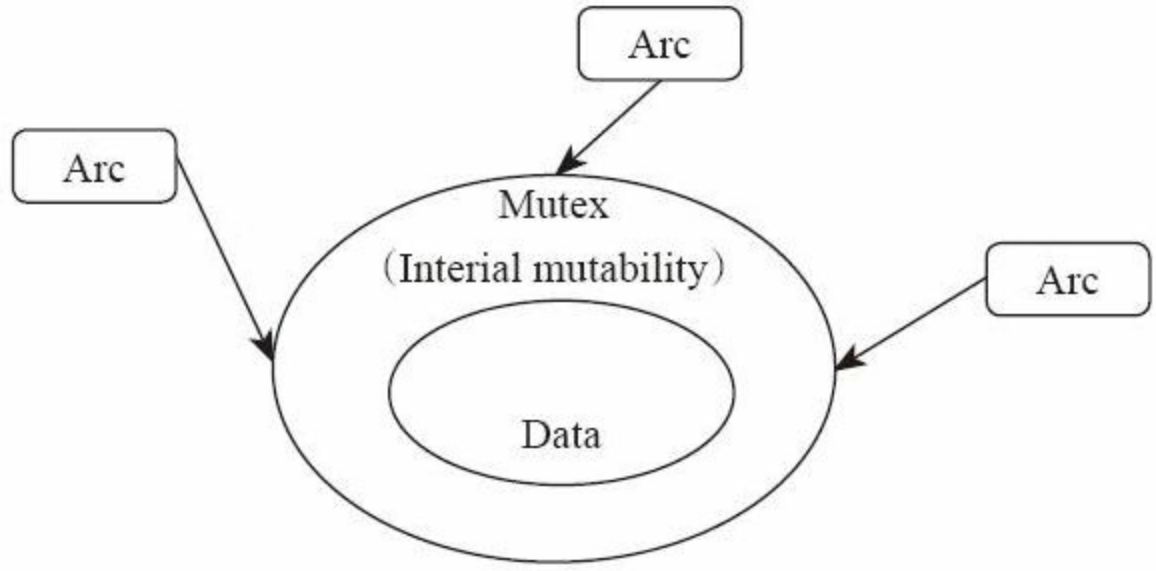
```
pub fn lock(&self) -> LockResult<MutexGuard<T>>
```

其中LockResult就是Result类型的一个别名，是用于错误处理的：

```
type LockResult<Guard> = Result<Guard, PoisonError<Guard>>;
```

如果当前Mutex已经是“有毒”（Poison）的状态，它返回的就是错误。什么情况会导致Mutex有毒呢？当Mutex在锁住的同时发生了panic，就会将这个Mutex置为“有毒”的状态，以后再调用lock（）都会失败。这个设计是为了panic safety而考虑的，主要就是考虑到在锁住的时候发生panic可能导致Mutex内部数据发生混乱。所以这个设计防止再次进入Mutex内部的时候访问了被破坏掉的数据内容。如果有需要的话，用户依然可以手动调用PoisonError：：into_inner（）方法获得内部数据。

而MutexGuard类型则是一个“智能指针”类型，它实现了DerefMut和Deref这两个trait，所以它可以被当作指向内部数据的普通指针使用。MutexGuard实现了一个析构函数，通过RAII手法，在析构函数中调用了unlock（）方法解锁。因此，用户是不需要手动调用方法解锁的。



Rust的这个设计，优点不在于它“允许你做什么”，而在于它“不允许你做什么”。 如果我们误用了Rc<isize>来实现线程之间的共享，就是编译错误。根据编译错误，我们将指针改为Arc类型，然后又会发现，它根本没有提供可变性。它的API只能共享读，根本没有写数据的方法存在。此时，我们会想到加入内部可变性来允许多线程共享读写。如果我们使用了Arc<RefCell<_>>类型，依然是编译错误。因为RefCell类型不满足Sync。而Arc<T>需要内部的T参数必须满足T：Sync，才能使Arc满足Sync。把这些综合起来，我们可以推理出Arc<RefCell<_>>是！Sync。

最终，编译器把其他的路都堵死了，唯一可以编译通过的就是使用那些满足Sync条件的类型，比如Arc<Mutex<_>>。在使用的时候，我们也不可能忘记调用lock方法，因为Mutex把真实数据包裹起来了，只有调用lock方法才有机会访问内部数据。我们也不需要记得调用unlock方法，因为lock方法返回的是一个MutexGuard类型，这个类型在析构的时候会自动调用unlock。

所以，编译器在逼着用户用正确的方式写代码。

29.3 RwLock

RwLock就是“读写锁”。它跟Mutex很像，主要区别是对外暴露的API不一样。对Mutex内部的数据读写，RwLock都是调用同样的lock方法；而对RwLock内部的数据读写，它分别提供了一个成员方法read/write来做这个事情。其他方面基本和Mutex一致。示例如下：

```
use std::sync::Arc;
use std::sync::RwLock;
use std::thread;

const COUNT: u32 = 1000000;

fn main() {
    let global = Arc::new(RwLock::new(0));

    let clone1 = global.clone();
    let thread1 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone1.write().unwrap();
            *value += 1;
        }
    });

    let clone2 = global.clone();
    let thread2 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone2.write().unwrap();
            *value -= 1;
        }
    });

    thread1.join().ok();
    thread2.join().ok();
    println!("final value: {:?}", global);
}
```

29.4 Atomic

Rust标准库还为我们提供了一系列的“原子操作”数据类型，它们在std：：sync：：atomic模块里面。它们都是符合Sync的，可以在多线程之间共享。比如，我们有AtomicIsize类型，顾名思义，它对应的是isize类型的“线程安全”版本。我们知道，普通的整数读取再写入，这种操作是非原子的。而原子整数的特点是，可以把“读取”“计算”“再写入”这样的操作编译为特殊的CPU指令，保证这个过程是原子操作。

我们来看一个示例：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicIsize, Ordering};
use std::thread;

const COUNT: u32 = 1000000;
fn main() {
    // Atomic 系列类型同样提供了线程安全版本的内部可变性
    let global = Arc::new(AtomicIsize::new(0));

    let clone1 = global.clone();
    let thread1 = thread::spawn(move|| {
        for _ in 0..COUNT {
            clone1.fetch_add(1, Ordering::SeqCst);
        }
    });

    let clone2 = global.clone();
    let thread2 = thread::spawn(move|| {
        for _ in 0..COUNT {
            clone2.fetch_sub(1, Ordering::SeqCst);
        }
    });

    thread1.join().ok();
    thread2.join().ok();
    println!("final value: {:?}", global);
}
```

这个示例我们很熟悉：两个线程修改同一个整数，一个线程对它进行多次加1，另外一个线程对它多次减1。这次我们发现，使用了Atomic类型后，我们可以保证最后的执行结果一定会回到0。

我们还可以把这段代码改动一下：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicIsize, Ordering};
use std::thread;

const COUNT: u32 = 1000000;

fn main() {
    let global = Arc::new(AtomicIsize::new(0));

    let clone1 = global.clone();
    let thread1 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone1.load(Ordering::SeqCst);
            value += 1;
            clone1.store(value, Ordering::SeqCst);
        }
    });


    let clone2 = global.clone();
    let thread2 = thread::spawn(move|| {
        for _ in 0..COUNT {
            let mut value = clone2.load(Ordering::SeqCst);
            value -= 1;
            clone2.store(value, Ordering::SeqCst);
        }
    });

    thread1.join().ok();
    thread2.join().ok();
    println!("final value: {:?}", global);
}
```

与上一个版本相比，这段代码的区别在于：我们没有使用原子类型自己提供的fetch_add fetch_sub方法，而是使用了load把里面的值读取出来，然后执行加/减，操作完成后，再用store存储回去。编译程序我们看到，是可以编译通过的。再执行，出现了问题：这次的执行结果就不是保证为0了。

大家应该很容易看明白问题在哪里。原来的那种写法，“读取/计算/写入”是一个完整的“原子操作”，中间不可被打断，它是一个“事务”（transaction）。而后面的写法把“读取”作为了一个“原子操作”，“写入”又作为了一个“原子操作”，把一个transaction分成了两段来执行。上面的那个程序，其逻辑类似于“lock=>读数据=>加/减运算=>写数据=>unlock”。下面的程序，其逻辑类似于“lock=>读数据=>unlock=>加/减运算=>lock=>写数据=>unlock”。虽然每次读写共享变量都保证了唯一性，但逻辑还是错的。

所以，编译器只能防止基本的数据竞争问题。如果程序里面有逻辑

错误，工具是没有办法帮我们发现的。上面这个示例中并不存在数据竞争问题，是完全的“业务逻辑bug”。

29.5 死锁

既然Rust中提供了“锁”机制，那么Rust中是否有可能出现“死锁”的现象呢？我们用经典的“哲学家就餐问题”来演示一下。

问题：假设有5个哲学家，共享一张放有5把椅子的桌子，每人分得一把椅子，但是，桌子上共有5支筷子，在每人两边各放一支，哲学家们在肚子饥饿时才试图分两次从两边拿起筷子就餐。

条件：

·拿到两支筷子时哲学家才开始吃饭；

·如果筷子已在他人手上，则该哲学家必须等他人吃完之后才能拿到筷子；

·任一哲学家在自己未拿到两只筷子前却不放下自己手中的筷子。

代码如下：

```
use std::thread;
use std::sync::{Mutex, Arc};
use std::time::Duration;

struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}


impl Philosopher {
    fn new(name: &str, left: usize, right: usize) -> Philosopher {
        Philosopher {
            name: name.to_string(),
            left: left,
            right: right,
        }
    }


    fn eat(&self, table: &Table) {
        let _left = table.forks[self.left].lock().unwrap();
        println!("{} take left fork.", self.name);
        thread::sleep(Duration::from_secs(2));
        let _right = table.forks[self.right].lock().unwrap();
```

```
        println!("{} take right fork.", self.name);
        thread::sleep(Duration::from_secs(1));
        println!("{} is done eating.", self.name);
    }
}

struct Table {
    forks: Vec<Mutex<()>>,
}

fn main() {
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});

    let philosophers = vec![
        Philosopher::new("Judith Butler", 0, 1),
        Philosopher::new("Gilles Deleuze", 1, 2),
        Philosopher::new("Karl Marx", 2, 3),
        Philosopher::new("Emma Goldman", 3, 4),
        Philosopher::new("Michel Foucault", 4, 0),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
        let table = table.clone();

        thread::spawn(move || {
            p.eat(&table);
        })
    }).collect();
    for h in handles {
        h.join().unwrap();
    }
}
```

编译执行，我们可以发现，5个哲学家都拿到了他左边的那支筷子，而都在等待他右边的那支筷子。在没等到右边筷子的时候，每个人都不会释放自己已经拿到的那支筷子。于是，大家都进入了无限的等待之中，程序无法继续执行了。这就是“死锁”。

在Rust中，“死锁”问题是没有办法在编译阶段由静态检查来解决的。就像前面提到的“循环引用制造内存泄漏”一样，编译器无法通过静态检查来完全避免这个问题，需要程序员自己注意。

29.6 Barrier

除了“锁”之外，Rust标准库还提供了一些其他线程之间的通信方式，比如Barrier等。Barrier是这样的一个类型，它使用一个整数做初始化，可以使得多个线程在某个点上一起等待，然后再继续执行。示例如下：

```
use std::sync::{Arc, Barrier};
use std::thread;

fn main() {
    let barrier = Arc::new(Barrier::new(10));
    let mut handlers = vec![];
    for _ in 0..10 {
        let c = barrier.clone();
        // The same messages will be printed together.
        // You will NOT see any interleaving.
        let t = thread::spawn(move|| {
            println!("before wait");
            c.wait();
            println!("after wait");
        });
        handlers.push(t);
    }

    for h in handlers {
        h.join().ok();
    }
}
```

这个程序创建了一个多个线程之间共享的Barrier，它的初始值是10。我们创建了10个子线程，每个子线程都有一个Arc指针指向了这个Barrier，并在子线程中调用了Barrier：：wait方法。这些子线程执行到wait方法的时候，就开始进入等待状态，一直到wait方法被调用了10次，10个子线程都进入等待状态，此时Barrier就通知这些线程可以继续了。然后它们再开始执行下面的逻辑。

所以最终的执行结果是：先打印出10条before wait，再打印出10条after wait，绝不会错乱。如果我们把c.wait（）这条语句删除掉可以看到，执行结果中，before wait和after wait是混杂在一起的。

29.7 Condvar

Condvar是条件变量，它可以用于等待某个事件的发生。在等待的时候，这个线程处于阻塞状态，并不消耗CPU资源。在常见的操作系统上，Condvar的内部实现是调用的操作系统提供的条件变量。它调用wait方法的时候需要一个MutexGuard类型的参数，因此Condvar总是与Mutex配合使用的。而且我们一定要注意，一个Condvar应该总是对应一个Mutex，不可混用，否则会导致执行阶段的panic。

Condvar的一个常见使用模式是和一个Mutex<bool>类型结合使用。我们可以用Mutex中的bool变量存储一个旧的状态，在条件发生改变的时候修改它的状态。通过这个状态值，我们可以决定是否需要执行等待事件的操作。示例如下：

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;
fn main() {
    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    thread::spawn(move|| {
        thread::sleep(Duration::from_secs(1));
        let &(ref lock, ref cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();
        println!("child thread {}", *started);
    });


    // wait for the thread to start up
    let &(ref lock, ref cvar) = &*pair;
    let mut started = lock.lock().unwrap();

    println!("before wait {}", *started);
    while !*started {
        started = cvar.wait(started).unwrap();
    }
    println!("after wait {}", *started);
}
```

这段代码中存在两个线程之间的共享变量，包括一个Condvar和一个Mutex封装起来的bool类型。我们用Arc类型把它们包起来。在子线程中，我们做完了某件工作之后，就将共享的bool类型变量设置为true，

并使用Condvar：：notify_one通知事件发生。

在主线程中，我们首先判定这个bool变量是否为true：如果已经是true，那就没必要进入等待状态了；否则，就进入阻塞状态，等待子线程完成任务。

29.8 全局变量

Rust中允许存在全局变量。在基本语法章节讲过，使用static关键字修饰的变量就是全局变量。全局变量有一个特点：如果要修改全局变量，必须使用unsafe关键字：

```
static mut G: i32 = 1;

fn main() {
    G = 2;
}
```

编译，可见错误信息为：

```
error[E0133]: use of mutable static requires unsafe function or block
```

这个规定显然是有助于线程安全的。如果允许任何函数可以随便读写全局变量的话，线程安全就无从谈起了。只有不用mut修饰的全局变量才是安全的，因为它只能被读取，不能被修改。

我们又可以想到，有些类型的变量不用mut修饰，也是可以做修改的。比如具备内部可变性的Cell等类型。我们可以试验一下，如果有一个全局的、具备内部可变性的变量，会发生什么情况：

```
#![feature(const_fn)]
use std::cell::Cell;
use std::thread;

static G: Cell<i32> = Cell::new(1);

fn f1() {
    G.set(2);
}

fn f2() {
    G.set(3);
}

fn main() {
    thread::spawn( || { f1() } );
    thread::spawn( || { f2() } );
}
```

试着编译一下上面这个例子，可见编译错误为：

```
error[E0277]: the trait bound `std::cell::Cell<i32>: std::marker::Sync` is not satis
--> test.rs:5:1
    |
5 | static G: Cell<i32> = Cell::new(1);
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `std::cell::Cell<i32>` cannot be shared be
    |
    = help: the trait `std::marker::Sync` is not implemented for `std::cell:: Cell<i
    = note: shared static variables must have a type that implements `Sync`
```

对于上面这个例子，我们可以推理一下，现在有两个线程同时修改一个全局变量，而且修改过程没有做任何线程同步，这里肯定是有线程安全的问题。但是，注意这里传递给spawn函数的闭包，实际上没有捕获任何局部变量，所以，它是满足Send条件的。在这种情况下，线程不安全类型并没有直接穿越线程的边界，spawn函数这里指定的约束条件是查不出问题来的。

但是，编译器还设置了另外一条规则，即共享又可变的全局变量必须满足Sync约束。根据Sync的定义，满足这个条件的全局变量显然是线程安全的。因此，编译器把这条路也堵死了，我们不可以简单地通过全局变量共享状态来构造出线程不安全的行为。对于那些满足Sync条件且具备内部可变性的类型，比如Atomic系列类型，作为全局变量共享是完全安全且合法的。

29.9 线程局部存储

线程局部（Thread Local）的意思是，声明的这个变量看起来是一个变量，但它实际上在每一个线程中分别有自己独立的存储地址，是不同的变量，互不干扰。在不同线程中，只能看到与当前线程相关联的那个副本，因此对它的读写无须考虑线程安全问题。

在Rust中，线程独立存储有两种使用方式。

·可以使用#[thread_local]attribute来实现。这个功能目前在稳定版中还不支持，只能在nightly版本中开启#！[feature（thread_local）]功能才能使用。

·可以使用thread_local！宏来实现。这个功能已经在稳定版中获得支持。

示例如下：

```
use std::cell::RefCell;
use std::thread;

fn main() {

    thread_local!{
        static FOO: RefCell<u32> = RefCell::new(1)
    };

    FOO.with(|f| {
        println!("main thread value1 {:?}", *f.borrow());
        *f.borrow_mut() = 2;
        println!("main thread value2 {:?}", *f.borrow());
    });

    let t = thread::spawn(move|| {
        FOO.with(|f| {
            println!("child thread value1 {:?}", *f.borrow());
            *f.borrow_mut() = 3;
            println!("child thread value2 {:?}", *f.borrow());
        });
    });
    t.join().ok();

    FOO.with(|f| {
        println!("main thread value3 {:?}", *f.borrow());
    });
}
```

用thread_local！声明的变量，使用的时候要用with（）方法加闭包来完成。以上代码编译、执行的结果为：

```
main thread value1 1
main thread value2 2
child thread value1 1
child thread value2 3
main thread value3 2
```

我们可以看到，在主线程中将FOO的值修改为2，但是进入子线程后，它看到的初始值依然是1。在子线程将FOO的值修改为3之后回到主线程，主线程看到的值还是2。

这说明，在子线程中和主线程中看到的FOO其实是两个完全独立的变量，互不影响。

29.10 总结

从前面的分析可见，Rust的设计一方面禁止了我们在线程之间随意共享变量，另一方面提供了一些工具类型供我们使用，使我们可以安全地在线程之间共享变量。这既提供了完整的功能，又避免了数据竞争一类的bug。Rust之所以这么设计，是因为设计者观察到了发生“数据竞争”的根源是什么。简单总结就是：

Alias+Mutation+No ordering

实际上我们可以看到，Rust保证内存安全的思路和线程安全的思路是一致的。在多线程中，我们要保证没有数据竞争，一般是通过下面的方式：

（1）多个线程可以同时读共享变量；

（2）只要存在一个线程在写共享变量，则不允许其他线程读/写共享变量。

这是不是与第二部分讲的“内存安全”的模型一模一样？这两个设计实际上是一脉相承的。如果没有“默认内存安全”打下的良好基础，Rust就没办法做到“线程安全”；正因为在“内存安全”问题上的一系列基础性设计，才导致了“线程安全”基本就是水到渠成的结果。我们甚至可以观察到一些“线程安全类型”和“非线程安全类型”之间有趣的对应关系，比如：

（1）Rc是非线程安全的，Arc则是与它对应的线程安全版本。当然还有弱指针Weak也是一一对应的。Rc无须考虑多线程场景下的问题，因此它内部只需普通整数做引用计数即可。Arc要用在多线程场景，因此它内部必须使用“原子整数”来做引用计数。

（2）RefCell是非线程安全的，它不能在跨线程场景使用。Mutex/RwLock则是与它相对应的线程安全版本。它们都提供了“内部可变性”，RefCell无须考虑多线程问题，所以它内部只需一个普通整数做借用计数即可。Mutex/RwLock可以用在多线程环境，所以它们内部需要使用操作系统提供的原语来完成“锁”功能。它们有相似之处，也有不

同之处。Mutex/RwLock在加锁的时候返回的是Result类型，是因为它们需要考虑“异常安全”问题——在多线程环境下，很可能出现一个线程发生了panic，导致Mutex内部的数据已经被破坏，而在另外一个线程中依然有可能观察到这个被破坏的数据结构。RefCell则相对简单，只需考虑AssertUnwindSafe即可。

（3）Cell是非线程安全的，Atomic*系列类型则是与它对应的线程安全版本。它们之间的相似之处在于，都提供了“内部可变性”，而且都不提供指向内部数据的方法。它们对内部数据的读写，都是整体读出、整体写入，不可能制造出直接指向内部数据的指针。它们的不同之处在于，Cell的条件更宽松。而标准库提供的Atomic*系列类型则受限于CPU提供的原子指令，内部存储的数据类型是有限的，无法推广到所有类型。其实我们完全可以仿造Cell类型，设计一个可以应用于所有类型的通用型Atomic<T>类型——内部用Mutex实现，提供get/set方法作为对外API。这个尝试已经在第三方开源库中实现，如需了解，上GitHub搜索“atomic-rs”即可。

Rust的这套线程安全设计有以下好处：

·免疫一切数据竞争；

·无额外性能损耗；

·无须与编译器紧耦合。

我们可以观察到一个有趣的现象：Rust语言实际上并不知晓“线程”这个概念，相关类型都是写在标准库中的，与其他类型并无二致。Rust语言提供的仅仅只是Sync、Send这样的一般性概念，以及生命周期分析、“borrow check”分析这样的机制。Rust编译器本身并未与“线程安全”“数据竞争”等概念深度绑定，也不需要一个runtime来辅助完成功能。然而，通过这些基本概念和机制，它却实现了完全通过编译阶段静态检查实现“免除数据竞争”这样的目标。这样的设计正是Rust的魅力所在。

正因为解耦合如此彻底，Rust语言才会如此精简，它只提供了非常基本的并行开发相关的基本抽象。而且标准库中实现的这些基本功能，其实都可以完全由第三方来实现。理论上来讲，其他语言中出现了的更高级的并行程序开发的抽象机制，一般都可以通过第三方库的方式来提

供，没必要与Rust编译器深度绑定。在下一章中，我们再来介绍一些更高级的并行开发模式，它们都由第三方库来实现，无须编译器特殊支持。

第30章　管道

在上一章中，我们主要讲述了如何在多线程中共享变量。Rust标准库中还提供了另外一种线程之间的通信方式：mpsc。这部分的库存储在std：：sync：：mpsc这个模块中。mpsc代表的是Multi-producer，single-consumer FIFO queue，即多生产者单消费者先进先出队列。这种线程之间的通信方式是在不同线程之间建立一个通信“管道”（channel），一边发送消息，一边接收消息，完成信息交流。

Do not communicate by sharing memory；instead，share memory by communicating.

——Effective Go

既然共享数据存在很多麻烦，那在某些场景下，用发消息的方式完成线程之间的通信是更合适的选择。

30.1 异步管道

异步管道是最常用的一种管道类型。它的特点是：发送端和接收端之间存在一个缓冲区，发送端发送数据的时候，是先将这个数据扔到缓冲区，再由接收端自己去取。因此，每次发送，立马就返回了，发送端不用管数据什么时候被接收端处理。

我们先用一个简单示例来演示一下管道的基本用法：

```
use std::thread;
use std::sync::mpsc::channel;

fn main() {
    let (tx, rx) = channel();
    thread::spawn(move|| {
        for i in 0..10 {
            tx.send(i).unwrap();
        }
    });

    while let Ok(r) = rx.recv() {
        println!("received {}", r);
    }
}
```

在这个示例中，我们首先创建了一个管道。channel函数的签名是这样的：

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>)
```

它返回了一个tuple，里面包括一个发送者（Sender）和一个接收者（Receiver）。

接下来我们创建一个子线程，然后将这个发送者move进入了子线程中。

子线程中的发送者不断循环调用send方法，发送数据。在主线程中，我们使用接收者不断调用recv方法接收数据。

我们可以注意到，channel（）是一个泛型函数，Sender和Receiver

都是泛型类型，且一组发送者和接收者必定是同样的类型参数，因此保证了发送和接收端都是同样的类型。因为Rust中的类型推导功能的存在，使我们可以在调用channel的时候不指定具体类型参数，而通过后续的方法调用，推导出正确的类型参数。

Sender和Receiver的泛型参数必须满足T：Send约束。这个条件是显而易见的：被发送的消息会从一个线程转移到另外一个线程，这个约束是为了满足线程安全。如果用户指定的泛型参数没有满足条件，在编译的时候会发生错误，提醒我们修复bug。

发送者调用send方法，接收者调用recv方法，返回类型都是Result类型，用于错误处理，因为它们都有可能调用失败。当发送者已经被销毁的时候，接收者调用recv则会返回错误；同样，当接收者已经销毁的时候，发送者调用send也会返回错误。

在管道的接收端，如果调用recv方法的时候还没有数据，它会进入等待状态阻塞当前线程，直到接收到数据才继续往下执行。

管道还可以是多发送端单接收端。做法很简单，只需将发送端Sender复制多份即可。复制方式是调用Sender类型的clone（）方法。这个库不支持多接收端的设计，因此Receiver类型没有clone（）方法。在上例的基础上我们稍做改动，创建多个线程，每个线程发送一个数据到接收端。代码如下：

```
use std::thread;
use std::sync::mpsc::channel;

fn main() {
    let (tx, rx) = channel();

    for i in 0..10 {
        let tx = tx.clone();  // 复制一个新的 tx,将这个复制的变量 move 进入子线程
        thread::spawn(move|| {
            tx.send(i).unwrap();
        });
    }
    drop(tx);

    while let Ok(r) = rx.recv() {
        println!("received {}", r);
    }
}
```

以上代码编译执行，可以发现它打印的结果与前面的例子不同了。

在前面示例中，这些数字呈顺序排列，因为发送端是按顺序发送的，接收端会保持同样的顺序。但在这个示例中，这些数字呈乱序排列，因为它们来自不同的线程，哪个先执行哪个后执行并不是确定的，取决于操作系统的调度。

目前我们用的这个管道是“异步”的，标准库还提供了另外一种“同步”管道供我们使用。同步管道和异步管道在接收端是一样的逻辑，区别在于发送端。

30.2 同步管道

异步管道内部有一个不限长度的缓冲区，可以一直往里面填充数据，直至内存资源耗尽。异步管道的发送端调用send方法不会发生阻塞，只要把消息加入到缓冲区，它就马上返回。

同步管道的特点是：其内部有一个固定大小的缓冲区，用来缓存消息。如果缓冲区被填满了，继续调用send方法的时候会发生阻塞，等待接收端把缓冲区内的消息拿走才能继续发送。缓冲区的长度可以在建立管道的时候设置，而且0是有效数值。如果缓冲区的长度设置为0，那就意味着每次的发送操作都会进入等待状态，直到这个消息被接收端取走才能返回。示例如下：

```
use std::thread;
use std::sync::mpsc::sync_channel;

fn main() {
    let (tx, rx) = sync_channel(1);
    tx.send(1).unwrap();
    println!("send first");
    thread::spawn(move|| {
        tx.send(2).unwrap();
        println!("send second");
    });

    println!("receive first {}", rx.recv().unwrap());
    println!("receive second {}", rx.recv().unwrap());
}
```

我们可以看到，程序执行结果永远是：发送一个并接收一个之后，才会出现发送第二个接收第二个。

我们讲的这两种管道都是单向通信，一个发送一个接收，不能反过来。Rust没有在标准库中实现管道双向通信。双向管道也不是不可能的，在第三方库中已经有了实现。

第31章　第三方并行开发库

在前面的章节中，我们介绍了Rust标准库提供的多线程相关工具。相比于当下许多流行的编程语言，Rust标准库的设计是属于比较基础、比较原始的。但是，Rust在线程安全方面的设计具有非常好的扩展性，包括标准库在内的很多实现并没有与编译器深度绑定。很多高级的抽象都可以通过第三方库的方式提供。而且，只要设计合理，高质量的第三方库可以拥有和标准库同样的“线程安全”特性。

本章就挑选一些社区内评价较高的并行开发方面的库，做简单介绍。

31.1 threadpool

threadpool是一个基本的线程池实现。在标准库中，我们可以用std：：thread：：spawn（）方法创建新线程。而这个库可以让我们指定一组固定数量的线程，它自动将每个任务分配到线程中去执行。使用方法如下：

```
use threadpool::ThreadPool;
use std::sync::mpsc::channel;

fn main() {
    let n_workers = 4;
    let n_jobs = 8;
    let pool = ThreadPool::new(n_workers);

    let (tx, rx) = channel();
    for _ in 0..n_jobs {
        let tx = tx.clone();
        pool.execute(move|| {
            tx.send(1).expect("channel will be there waiting for the pool");
        });
    }

    assert_eq!(rx.iter().take(n_jobs).fold(0, |a, b| a + b), 8);
```

ThreadPool：：execute与std：：thread：：spawn的区别就是，它需要先创建一个对象，然后调用：execute方法，其他都差不多。

31.2 scoped-threadpool

在前面的章节中，我们已经知道，如果要在多线程之间共享变量，必须使用Arc这样的保证线程安全的智能指针。然而，Arc是有运行期开销的（虽然很小）。假如我们有时候需要子线程访问当前调用栈中的局部变量，而且能保证当前函数的生命周期一定大于子线程的生命周期，子线程一定先于当前函数退出，那我们能不能直接在子线程中使用最简单的借用指针&来访问父线程栈上的局部对象呢？

至少标准库中的spawn函数是不行的。spawn的签名是：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
```

注意这里的闭包要满足f：'static约束。这意味着闭包中存在不能捕获短生命周期的变量，比如指向当前局部调用栈的指针。这是因为spawn函数会将闭包传递给一个新的子线程，这个子线程的生命周期很可能大于当前函数调用生命周期。如果我们希望在子线程中访问当前函数中的局部变量，怎么办呢？可以使用第三方库scoped_threadpool。我们来看看scoped_threadpool是如何使用的：

```
extern crate scoped_threadpool;
use scoped_threadpool::Pool;

fn main() {
    let mut pool = Pool::new(4);

    let mut vec = vec![0, 1, 2, 3, 4, 5, 6, 7];

    pool.scoped(|scope| {
        for e in &mut vec {
            scope.execute(move || {
                *e += 1;
            });
        }
    });

    println!("{:?}", vec);
}
```

在这里，线程内部直接使用了&mut vec形式访问了父线程“栈”上的

变量，而不必使用Arc。我们可以注意到，scoped方法的签名是这样的：

```
fn scoped<'pool, 'scope, F, R>(&'pool mut self, f: F) -> R
        where F: FnOnce(&Scope<'pool, 'scope>) -> R
```

这里，参数闭包的约束条件没有'static这一项。所以我们上面的调用是可以成功的。scoped_threadpool库的源码并不复杂，只需一个文件即可，各位读者可以自己去GitHub上阅读它的源码，看看它是怎么实现的。

31.3 parking_lot

Rust标准库帮我们封装了一些基本的操作系统的同步原语，比如Mutex Condvar等。一般情况下这些够我们使用了。但是还有一些对性能有极致要求的开发者对标准库的实现并不满意，于是社区里又有人开发出来了一套替代品，在性能和易用性方面，都比标准库更好，这就是parking_lot库。下面的示例展示了这个库提供的Mutex，它的用法与标准库的Mutex差别不大：

```
use std::sync::Arc;
use parking_lot::Mutex;
use std::thread;
use std::sync::mpsc::channel;

fn main() {
    const N: usize = 10;

    let data = Arc::new(Mutex::new(0));

    let (tx, rx) = channel();
    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());
        thread::spawn(move || {
            let mut data = data.lock();
            *data += 1;
            if *data == N {
                tx.send(()).unwrap();
            }
        });
    }

    println!("{}", rx.recv().unwrap());
}
```

这个库也给我们展现了Rust在并发方面的高度可扩展性，想要实现什么功能，基本不会因为编译器的限制而无法做到。

31.4 crossbeam

crossbeam是Rust核心组的另外一位重要成员Aaron Turon牵头开发的。它包含了并行开发的很多方面的功能，比如无锁数据类型，以及重新设计实现的管道。

我们知道标准库给了一份mpsc（多生产者单消费者）管道的实现，但是它有许多缺陷。crossbeam-channel这个库给我们提供了另外一套管道的实现方式。不仅包括mpsc，还包括mpmc（多生产者多消费者），而且使用便捷，执行效率也很高。

下面是一个双端管道的使用示例。它基本实现了go语言的内置管道功能，在执行效率上甚至有过之而无不及。大家可以到GitHub上跟踪一下这个项目的进展。

```
extern crate crossbeam;
#[macro_use]
extern crate crossbeam_channel as channel;

use channel::{Receiver, Sender};

fn main() {
    let people = vec!["Anna", "Bob", "Cody", "Dave", "Eva"];
    let (tx, rx) = channel::bounded(1); // Make room for one unmatched send.
    let (tx, rx) = (&tx, &rx);

    crossbeam::scope(|s| {
        for name in people {
            s.spawn(move || seek(name, tx, rx));
        }
    });

    if let Ok(name) = rx.try_recv() {
        println!("No one received {}’s message.", name);
    }
}

// Either send my name into the channel or receive someone else's, whatever happens
fn seek<'a>(name: &'a str, tx: &Sender<&'a str>, rx: &Receiver<&'a str>) {
    select_loop! {
        recv(rx, peer) => println!("{} received a message from {}.", name, peer),
        send(tx, name) => {}, // Wait for someone to receive my message.
    }
}
```

31.5 rayon

C#语言有一个很厉害的PLinq扩展，可以轻松地将linq语句并行化。比如：

```
string[] words = new[] { "Welcome", "to", "Beijing","OK","Hua","Ying","Ni" ,"2008"};
var lazyBeeQuery = from word in words.AsParallel() select word;
lazyBeeQuery.ForAll<string>(word => { Console.WriteLine(word); });
```

在新的C++17中，标准库也支持了一些并行算法：

```
std::experimental::parallel::for_each(
                        std::experimental::parallel::par, // 并行执行
                        v.begin(), v.end(), functor);
```

在Rust中，迭代器基本已经与linq的功能差不多。那我们能不能做个类似的扩展，让普通迭代器轻松变成并行迭代器呢？Rayon的设计目标就是这个。

Rayon是Rust核心组成员Nicholas Matsakis开发的一个并行迭代器项目。它可以把一个按顺序执行的任务轻松变成并行执行。它非常轻量级，效率极高，而且使用非常简单。而且它保证了无数据竞争的线程安全。

Rayon的API主要有两种：

·并行迭代器——对一个可迭代的序列调用par_iter方法，就可以产生一个并行迭代器；

·join函数——它可以把一个递归的分治算法问题变成并行执行。

照例，我们用示例来说明它的基本用法。比如，我们想对一个整数数组执行平方和计算，可以这样做：

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()  // iter() 换成 par_iter()
```

```
        .map(|&i| i * i)
        .sum()
}
```

这个问题是可以并行计算的，每个元素的平方操作互不干扰，如果能让它们在不同线程计算，最后再一起求和，可以提高执行效率。用Rayon来解决这个问题很简单，只需将单线程情况下的iter（）方法改为par_iter（）即可。Rayon会在后台启动一个线程池，自动分配任务，将多个元素的map操作分配到不同的线程中并行执行，最后把所有的执行结果汇总再相加。

类似的，这个迭代器也有mut版本。假如我们想并行修改某个数组，可以这样做：

```
use rayon::prelude::*;
fn increment_all(input: &mut [i32]) {
    input.par_iter_mut()
        .for_each(|p| *p += 1);
}
```

Rayon的另外一种使用方式是调用join函数。这个函数特别适合于分治算法。一个典型的例子是写一个快速排序算法：

```
fn partition<T:PartialOrd+Send>(v: &mut [T]) -> usize {
    let pivot = v.len() - 1;
    let mut i = 0;
    for j in 0..pivot {
        if v[j] <= v[pivot] {
            v.swap(i, j);
            i += 1;
        }
    }
    v.swap(i, pivot);
    i
}

fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() <= 1 {
        return;
    }

    let mid = partition(v);
    let (lo, hi) = v.split_at_mut(mid);
    rayon::join(|| quick_sort(lo), || quick_sort(hi));
}

fn main() {
    let mut v = vec![10,9,8,7,6,5,4,3,2,1];
```

```
    quick_sort(&mut v);
    println!("{:?}", v);
}
```

在快速排序算法中，我们可以先把数组切分为两部分，然后分别再对这两部分执行快速排序。在这里，我们使用了rayon：：join函数。

需要注意的是，并行迭代器和join函数并不是简单地新建线程，然后在两个线程上分别执行。它内部实际上使用了“work steal”策略。它后台的线程是由一个线程池管理的，join函数只是把这两个闭包作为两个任务分发出去了，并不保证这两个闭包一定会并行执行或者串行执行。如果现在有空闲线程，那么空闲线程就会执行这个任务。总之，哪个线程有空，就在哪个线程上执行，它不会让某些线程早早结束而让某些任务在其他线程里面等待。所以，它的开销是非常小的。Rayon这个库在性能测试的benchmark上的表现也是非常不错的，具体数据大家可以查看官方网站，或者自行测试。

同时，我还要强调一点，Rayon同样保证了“线程安全”。比如，我们如果想在两个任务中同时修改一个数组，编译器会阻止我们：

```
fn increment_all(slice: &mut [i32]) {
    rayon::join(|| process(slice), || process(slice));
}
```

我们应该能猜想到，这里的API肯定用到了Send、Sync之类的约束，就跟标准库中的spawn函数一样。因此第三方库也一样能享受到“线程安全”的优点。

有关这个库的使用方法以及其内部实现原理，在Nicholas Matsakis的博客有详细描述，本书篇幅有限就不再展开了。从这个库我们可以看到，Rust为各种并行开发的模式提供了无限的可能性。虽然标准库在这方面提供的直接选择不多，但并没有阻碍我们实现各种各样的第三方库。Rust在并行开发方面同时实现了执行效率高、安全性好、扩展性好的特点。

第五部分　实用设施

第32章　项目和模块

本书中的绝大部分代码示例都是很短的，一个文件就可以搞定。但是，任何一个规模稍微大一点的项目都不能这么写。我们需要一个机制，把一个项目切分成若干小部分，每个部分又可以切分成更小的部分，层层抽象，通过这种方式来管理复杂的代码。这就是很多编程语言中都有的“模块系统”。

Rust用了两个概念来管理项目：一个是crate，一个是mod。

·crate简单理解就是一个项目。crate是Rust中的独立编译单元。每个crate对应生成一个库或者可执行文件（如.lib.dll.so.exe等）。官方有一个crate仓库https://crates.io/ ，可以供用户发布各种各样的库，用户也可以直接使用这里面的开源库。

·mod简单理解就是命名空间。mod可以嵌套，还可以控制内部元素的可见性。

crate和mod有一个重要区别是：crate之间不能出现循环引用；而mod是无所谓的，mod1要使用mod2的内容，同时mod2要使用mod1的内容，是完全没问题的。在Rust里面，crate才是一个完整的编译单元（compile unit）。也就是说，rustc编译器必须把整个crate的内容全部读进去才能执行编译，rustc不是基于单个的.rs文件或者mod来执行编译的。作为对比，C/C++里面的编译单元是单独的.c/.cpp文件以及它们所有的include文件。每个.c/.cpp文件都是单独编译，生成.o文件，再把这些.o文件链接起来。

本章我们详细讲解一下crate和mod。

32.1 cargo

Cargo是Rust的包管理工具，是随着编译器一起发布的。在使用rustup安装了官方发布的Rust开发套装之后，Cargo工具就已经安装好了，无须单独安装。我们可以使用cargo命令来查看它的基本用法。Cargo的官方使用文档在这个地址：https://doc.rust-lang.org/cargo/ 。

Cargo可以用于创建和管理项目、编译、执行、测试、管理外部下载的包和可执行文件等。

下面我们使用cargo来创建一个项目，一步步带领大家学习cargo的基本用法。

我们创建一个新的工程，这个工程会生成一个可执行程序。步骤如下。

（1）进入项目文件夹后，使用如下命令：

```
cargo new hello_world --bin
```

后面的–-bin选项意味着我们希望生成的是可执行程序，cargo会帮助我们生成一个main.rs的模板文件。

如果我们的工程是一个library，则可以使用--lib选项，此时自动生成的是一个lib.rs的模板文件。

（2）使用tree.命令查看当前的文件夹结构。可以看到：

```
.
└── hello_world
    ├── Cargo.toml
    └── src
        └── main.rs

2 directories, 2 files
```

其中，Cargo.toml是我们的项目管理配置文件，这里记录了该项目

相关的元信息。关于这个文件的详细格式定义，可以参考官方网站上的帮助文档：https://doc.rust-lang.org/cargo/ 。

src文件夹内是源代码。

（3）进入hello_world文件夹，使用cargo build命令，编译项目。生成的可执行文件在./target/debug/文件夹内。如果我们使用cargo build--release命令，则可以生成release版的可执行文件，它比debug版本优化得更好。

（4）使用./target/debug/hello_world命令，或者cargo run命令，可以执行我们刚生成的这个可执行程序。

在Rust中，一个项目对应着一个目标文件，可能是library，也可能是可执行程序。现在我们试试给我们的程序添加依赖项目。

进入hello_world的上一层文件夹，新建一个library项目：

```
cargo new good_bye
```

lib.rs文件是库项目的入口，打开这个文件，写入以下代码：

```
pub fn say() {
    println!("good bye");
}
```

使用cargo build，编译通过。现在我们希望hello_world项目能引用good_bye项目。打开hello_world项目的Cargo.toml文件，在依赖项下面添加对good_bye的引用。

```
[dependencies]
good_bye = { path = "../good_bye" }
```

这个写法是引用本地路径中的库。如果要引用官方仓库中的库更简单，比如：

```
[dependencies]
```

```
lazy_static = "1.0.0"
```

现在在应用程序中调用这个库。打开main.rs源文件，修改代码为：

```
extern crate good_bye;

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

再次使用cargo run编译执行，就可以看到我们正确调用了good_bye项目中的代码。

cargo只是一个包管理工具，并不是编译器。Rust的编译器是rustc，使用cargo编译工程实际上最后还是调用的rustc来完成的。如果我们想知道cargo在后面是如何调用rustc完成编译的，可以使用cargo build--verbose选项查看详细的编译命令。

cargo在Rust的生态系统中扮演着非常重要的角色。除了最常用的cargo new、cargo build、cargo run等命令之外，还有很多有用的命令。我们可以用cargo-h来查看其他用法。现在挑选一部分给大家介绍：

·check

check命令可以只检查编译错误，而不做代码优化以及生成可执行程序，非常适合在开发过程中快速检查语法、类型错误。

·clean

清理以前的编译结果。

·doc

生成该项目的文档。

·test

执行单元测试。

·bench

执行benchmark性能测试。

·update

升级所有依赖项的版本，重新生成Cargo.lock文件。

·install

安装可执行程序。

·uninstall

删除可执行程序。

其中，cargo install是一个非常有用的命令，它可以让用户自己扩展cargo的子命令，为它增加新功能。比如我们可以使用

```
cargo install cargo-tree
```

安装一个新的cargo子命令，接下来就可以使用

```
cargo tree
```

把所有依赖项的树型结构打印出来。

在crates.io网站上，用subcommand关键字可以搜出许多有用的子命令，用户可以按需安装。

32.2 项目依赖

在Cargo.toml文件中，我们可以指定一个crate依赖哪些项目。这些依赖既可以是来自官方的crates.io，也可以是某个git仓库地址，还可以是本地文件路径。示例如下：

```
[dependencies]
lazy_static = "1.0.0"
rand = { git = https://github.com/rust-lang-nursery/rand, branch = "master" }
my_own_project = { path = "/my/local/path", version = "0.1.0" }
```

Rust里面的crate都是自带版本号的。版本号采用的是语义版本的思想（参考http://semver.org/ ）。基本意思如下：

·1.0.0以前的版本是不稳定版本，如果出现了不兼容的改动，升级次版本号，比如从0.2.15升级到0.3.0；

·在1.0.0版本之后，如果出现了不兼容的改动，需要升级主版本号，比如从1.2.3升级到2.0.0；

·在1.0.0版本之后，如果是兼容性的增加API，虽然不会导致下游用户编译失败，但是增加公开的API情况，应该升级次版本号。

下面详细讲一下在[dependencies]里面的几种依赖项的格式：

（1）来自crates.io的依赖

绝大部分优质开源库，作者都会发布到官方仓库中，所以我们大部分的依赖都是来自于这个地方。在crates.io中，每个库都有一个独一无二的名字，我们要依赖某个库的时候，只需指定它的名字及版本号即可，比如：

```
[dependencies]
lazy_static = "1.0"
```

指定版本号的时候，可以使用模糊匹配的方式。

·^符号，如^1.2.3代表1.2.3<=version<2.0.0；

·~符号，如~1.2.3代表1.2.3<=version<1.3.0；

·*符号，如1.*代表1.0.0<=version<2.0.0；

·比较符号，比如>=1.2.3、>1.2.3、<2.0.0、=1.2.3含义基本上一目了然，还可以把多个限制条件合起来用逗号分开，比如version=">1.2，<1.9"。

直接写一个数字的话，等同于^符号的意思。所以lazy_static="1.0"等同于lazy_static="^1.0"，含义是1.0.0<=version<2.0.0。cargo会到网上找到当前符合这个约束条件的最新的版本下载下来。

（2）来自git仓库的依赖

除了最简单的git="…"指定repository之外，我们还可以指定对应的分支：

```
rand = { git = https://github.com/rust-lang-nursery/rand, branch = "next" }
```

或者指定当前的commit号：

```
rand = { git = https://github.com/rust-lang-nursery/rand, branch = "master", rev = '
```

还可以指定对应的tag名字：

```
rand = { git = https://github.com/rust-lang-nursery/rand, tag = "0.3.15" }
```

（3）来自本地文件路径的依赖

指定本地文件路径，既可以使用绝对路径也可以使用相对路径。

当我们使用cargo build编译完项目后，项目文件夹内会产生一个新文件，名字叫Cargo.lock。它实际上是一个纯文本文件，同样也是toml格式。它里面记录了当前项目所有依赖项目的具体版本。每次编译项目

的时候，如果该文件存在，cargo就会使用这个文件中记录的版本号编译项目；如果该文件不存在，cargo就会使用Cargo.toml文件中记录的依赖项目信息，自动选择最合适的版本。

一般来说：如果我们的项目是库，那么最好不要把Cargo.lock文件纳入到版本管理系统中，避免依赖库的版本号被锁死；如果我们的项目是可执行程序，那么最好要把Cargo.lock文件纳入到版本管理系统中，这样可以保证，在不同机器上编译使用的是同样的版本，生成的是同样的可执行程序。

对于依赖项，我们不仅要在Cargo.toml文件中写出来，还要在源代码中写出来。目前版本中，必须在crate的入口处（对库项目就是lib.rs文件，对可执行程序项目就是main.rs文件）写上：

```
extern crate hello;    // 声明外部依赖
extern crate hello as hi; // 可以重命名
```

32.2.1 配置

cargo也支持配置文件。配置文件可以定制cargo的许多行为，就像我们给git设置配置文件一样。类似的，cargo的配置文件可以存在多份，它们之间有优先级关系。你可以为某个文件夹单独提供一份配置文件，放置到当前文件夹的.cargo/config位置，也可以提供一个全局的默认配置，放在$HOME/.cargo/config位置。下面是一份配置示例：

```
[cargo-new]
// 可以配置默认的名字和email,这些会出现在新项目的 Cargo.toml 中
name = "..."
email = "..."
[build]
jobs = 1                    // 并行执行的rustc程序数量
rustflags = ["..", ".."]  // 编译时传递给 rustc 的额外命令行参数
[term]
verbose = false             // 执行命令时是否打印详细信息
color = 'auto'              // 控制台内的彩色显示
[alias]                     // 设置命令别名
b = "build"
t = "test"
r = "run"
rr = "run --release"
```

更详细的信息请参考官方文档。

32.2.2 workspace

cargo的workspace概念，是为了解决多crate的互相协调问题而存在的。假设现在我们有一个比较大的项目。我们把它拆分成了多个crate来组织，就会面临一个问题：不同的crate会有各自不同的Cargo.toml，编译的时候它们会各自产生不同的Cargo.lock文件，我们无法保证所有的crate对同样的依赖项使用的是同样的版本号。

为了让不同的crate之间能共享一些信息，cargo提供了一个workspace的概念。一个workspace可以包含多个项目；所有的项目共享一个Cargo.lock文件，共享同一个输出目录；一个workspace内的所有项目的公共依赖项都是同样的版本，输出的目标文件都在同一个文件夹内。

workspace同样是用Cargo.toml来管理的。我们可以把所有的项目都放到一个文件夹下面。在这个文件夹下写一个Cargo.toml来管理这里的所有项目。Cargo.toml文件中要写一个[workspace]的配置，比如：

```
[workspace]

members = [
    "project1", "lib1"
]
```

整个文件夹的目录结构如下：

```
├── Cargo.lock
├── Cargo.toml
├── project1
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── lib1
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── target
```

我们可以在workspace的根目录执行cargo build等命令。请注意，虽

然每个crate都有自己的Cargo.toml文件，可以各自配置自己的依赖项，但是每个crate下面不再会各自生成一个Cargo.lock文件，而是统一在workspace下生成一个Cargo.lock文件。如果多个crate都依赖一个外部库，那么它们必然都是依赖的同一个版本。

32.2.3 build.rs

cargo工具还允许用户在正式编译开始前执行一些自定义的逻辑。方法是在Cargo.toml中配置一个build的属性，比如：

```
[package]
# ...
build = "build.rs"
```

自定义逻辑就写在build.rs文件里面。在执行cargo build的时候，cargo会先把这个build.rs编译成一个可执行程序，然后运行这个程序，做完后再开始编译真正的crate。build.rs一般用于下面这些情况：

·提前调用外部编译工具，比如调用gcc编译一个C库；

·在操作系统中查找C库的位置；

·根据某些配置，自动生成源码；

·执行某些平台相关的配置。

build.rs里面甚至可以再依赖其他的库。可以在build-dependencies里面指定：

```
[build-dependencies]
rand = "1.0"
```

build.rs里面如果需要读取当前crate的一些信息，可以通过环境变量来操作。cargo在执行这个程序之前就预先设置好了一些环境变量，比较常用的有下面几种。

·CARGO_MANIFEST_DIR

当前crate的Cargo.toml文件的路径。

·CARGO_PKG_NAME

当前crate的名字。

·OUT_DIR

build.rs的输出路径。如果要在build.rs中生成代码，那么生成的代码就要存在这个文件夹下。

·HOST

当前rustc编译器的平台特性。

·OPT_LEVEL

优化级别。

·PROFILE

判断是release还是debug版本。

更多的环境变量请参考cargo的标准文档。

下面还是用一个完整的示例演示一下build.rs功能如何使用。假设我们现在要把当前项目最新的commit id记录到可执行程序里面。这种需求就必须使用自动代码生成来完成。首先新建一个项目with_commit_hash：

```
cargo new –bin with_commit_hash
```

然后，到Cargo.toml里面加上：

```
build = "build.rs"
```

当然，对应的，要在项目文件夹下创建一个build.rs的文件。

我们希望能在编译过程中生成一份源代码文件，里面记录了一个常量，类似这样：

```
const CURRENT_COMMIT_ID : &’static str = "123456789ABCDEF";
```

查找当前git的最新commit id可以通过命令git rev-parse HEAD来完成。所以，我们的build.rs可以这样实现：

```
use std::env;
use std::fs::File;
use std::io::Write;
use std::path::Path;
use std::process::Command;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("commit_id.rs");
    let mut f = File::create(&dest_path).unwrap();

    let commit = Command::new("git")
        .arg("rev-parse")
        .arg("HEAD")
        .output()
        .expect("Failed to execute git command");
    let commit = String::from_utf8(commit.stdout).expect("Invalid utf8 string");

    let output = format!(r#"pub const CURRENT_COMMIT_ID : &'static str = "{}";" #, c

    f.write_all(output.as_bytes()).unwrap();
}
```

输出路径是通过读取OUT_DIR环境变量获得的。利用标准库里面的Command类型，我们可以调用外部的进程，并获得它的标准输出结果。最后再构造出我们想要的源码字符串，写入到目标文件中。

生成了这份代码之后，我们怎么使用呢？在main.rs里面，可以通过宏直接把这部分源码包含到项目中来：

```
include!(concat!(env!("OUT_DIR"), "/commit_id.rs"));

fn main() {
    println!("Current commit id is: {}", CURRENT_COMMIT_ID);
}
```

这个include！宏可以直接把目标文件中的内容在编译阶段复制到当

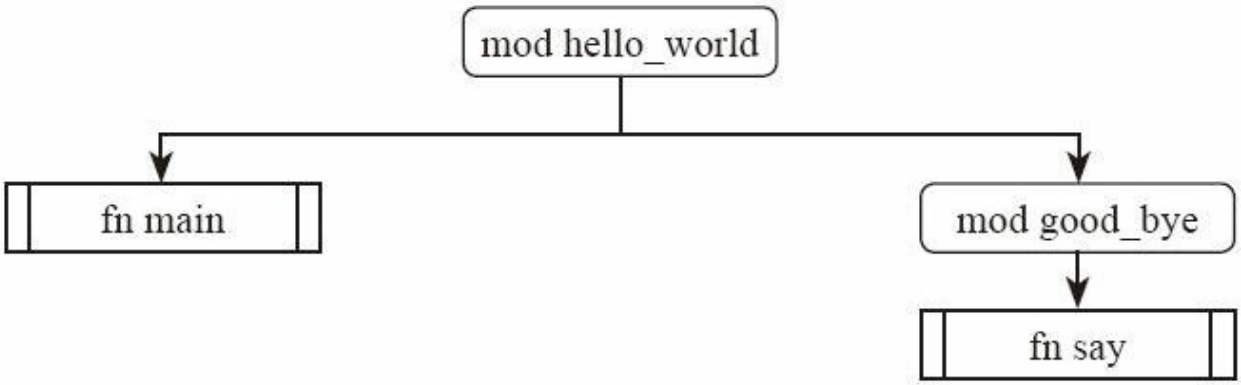
前位置。这样main函数就可以访问CURRENT_COMMIT_ID这个常量了。大家要记得在当前项目使用git命令新建几个commit。然后编译，执行，可见在可执行程序中包含最新commit id这个任务就完全自动化起来了。

32.3 模块管理

前面我们讲解了如何使用cargo工具管理crate。接下来还要讲解一个crate内部如何管理模块。可惜的是，Rust设计组觉得目前的模块系统还有一些瑕疵，准备继续改进，在编写本书的时候这部分内容正处在热火朝天的讨论过程中。改进的目标是思维模型更简洁、更加具备一致性、方便各个层次的用户。所以本书在这部分不会强调太多的细节，因为目前一些看起来比较繁复的细节将来很可能会得到简化。

32.3.1 文件组织

mod（模块）是用于在crate内部继续进行分层和封装的机制。模块内部又可以包含模块。Rust中的模块是一个典型的树形结构。每个crate会自动产生一个跟当前crate同名的模块，作为这个树形结构的根节点。比如在前面使用cargo创建多个项目的示例中，项目hello_world依赖于项目good_bye，我们要调用good_bye中的函数，需要写good_bye：：say（）；，这是因为say方法存在于good_bye这个mod中。它们组成的树形关系如下图所示：



在一个crate内部创建新模块的方式有下面几种。

·一个文件中创建内嵌模块。直接使用mod关键字即可，模块内容包含到大括号内部。

```
mod name { fn items() {}  … }
```

·独立的一个文件就是一个模块。文件名即是模块名。

·一个文件夹也可以创建一个模块。文件夹内部要有一个mod.rs文件，这个文件就是这个模块的入口。

使用哪种方式编写模块取决于当时的场景。如果我们需要创建一个小型子模块，比如单元测试模块，那么直接写到一个文件内部就非常简单而且直观；如果一个模块内容相对有点多，那么把它单独写到一个文件内是更容易维护的；如果一个模块的内容太多了，那么把它放到一个文件夹中就更合理，因为我们可以把真正的内容继续分散到更小的子模块中，而在mod.rs中直接重新导出（re-export）。这样mod.rs的源码就大幅简化，不影响外部的调用者。

可以这样理解：模块是一种更抽象的概念，文件是承载这个概念的实体。但是模块和文件并不是简单的一一对应关系，用户可以自己维护这个映射关系。

比如，我们有一个crate内部包含了两个模块，一个是caller一个是worker。我们可以有几种方案来实现。

方案一：直接把所有代码都写到lib.rs里面：

```
// <lib.rs>
mod caller {
    fn call() {}
}

mod worker {
    fn work1() {}
    fn work2() {}
    fn work3() {}
}
```

方案二：把这两个模块分到两个不同的文件中，分别叫作caller.rs和worker.rs。那么我们的项目就有了三个文件，它们的内容分别是：

```
// <lib.rs>
mod caller;
mod worker;
// <caller.rs>
fn call() {}
```

```
// <worker.rs>
fn work1() {}
fn work2() {}
fn work3() {}
```

因为lib.rs是这个crate的入口，我们需要在这里声明它的所有子模块，否则caller.rs和worker.rs都不会被当成这个项目的源码编译。

方案三：如果worker.rs这个文件包含的内容太多，我们还可以继续分成几个文件：

```
// <lib.rs>
mod caller;
mod worker;
// <caller.rs>
fn call() {}
// <worker/mod.rs>
mod worker1;
mod worker2;
mod worker3;
// <worker/worker1.rs>
fn work1() {}
// <worker/worker2.rs>
fn work2() {}
// <worker/worker3.rs>
fn work3() {}
```

这样就把一个模块继续分成了几个小模块。而且worker模块的拆分其实是不影响caller模块的，只要我们在worker模块中把它子模块内部的东西重新导出（re-export）就可以了。这个是可见性控制的内容，下面我们继续介绍可见性控制。

32.3.2 可见性

我们可以给模块内部的元素指定可见性。默认都是私有，除了两种例外情况：一是用pub修饰的trait内部的关联元素（associated item），默认是公开的；二是pub enum内部的成员默认是公开的。公开和私有的访问权限是这样规定的：

·如果一个元素是私有的，那么只有本模块内的元素以及它的子模块可以访问；

·如果一个元素是公开的，那么上一层的模块就有权访问它。

示例如下：

```
mod top_mod1 {
    pub fn method1() {}

    pub mod inner_mod1 {
        pub fn method2() {}

        fn method3() {}
    }

    mod inner_mod2 {
        fn method4() {}

        mod inner_mod3 {
            fn call_fn_inside() {
                super::method4();
            }
        }
    }
}

fn call_fn_outside() {
    ::top_mod1::method1();
    ::top_mod1::inner_mod1::method2();
}
```

在这个示例中，top_mod1外部的函数call_fn_outside（），有权访问method1（），因为它是用pub修饰的。同样也可以访问method2（），因为inner_mod1是pub的，而且method2也是pub的。而inner_mod2不是pub的，所以外部的函数是没法访问method4的。但是call_fn_inside是有权访问method4的，因为它在method4所处模块的子模块中。

模块内的元素可以使用pub use重新导出（re-export）。这也是Rust模块系统的一个重要特点。示例如下：

```
mod top_mod1 {
    pub use self::inner_mod1::method1;

    mod inner_mod1 {
        pub use self::inner_mod2::method1;

        mod inner_mod2 {
            pub fn method1() {}
        }
    }
```

```
}

fn call_fn_outside() {
    ::top_mod1::method1();
}
```

在call_fn_outside函数中，我们调用了top_mod1中的函数method1。可是我们注意到，method1其实不是在top_mod1内部实现的，它只是把它内部inner_mod1里面的函数重新导出了而已。pub use就是起这样的作用，可以把元素当成模块的直接成员公开出去。我们继续往下看还可以发现，这个函数在inner_mod1里面也只是重新导出的，它的真正实现是在inner_mod2里面。

这个机制可以让我们轻松做到接口和实现的分离。我们可以先设计好一个模块的对外API，这个固定下来之后，它的具体实现是可以随便改，不影响外部用户的。我们可以把具体实现写到任何一个子模块中，然后在当前模块重新导出即可。对外部用户来说，这没什么区别。

不过这个机制有个麻烦之处就是，如果具体实现嵌套在很深层次的子模块中的话，要把它导出到最外面来，必须一层层地转发，任何一层没有重新导出，都是无法达到目标的。

Rust里面用pub标记了的元素最终可能在哪一层可见，并不能很简单地得出结论。因为它有可能被外面重新导出。为了更清晰地限制可见性，Rust设计组又给pub关键字增加了下面的用法，可以明确地限定元素的可见性：

```
pub(crate)
pub(in xxx_mod)
pub(self)  或者 pub(in self)
pub(super) 或者 pub(in super)
```

示例如下：

```
mod top_mod {
    pub mod inner_mod1 {
        pub mod inner_mod2 {
            pub(self) fn method1() {}
            pub(super) fn method2() {}
            pub(crate) fn method3() {}
```

```
        }

        // Error:
        // pub use self::inner_mod2::method1;
        fn caller1() {
            // Error:
            // self::inner_mod2::method1();
        }
    }

    fn caller2() {
        // Error:
        // self::inner_mod1::inner_mod2::method2();
    }
}

// Error:
// pub use ::top_mod::inner_mod1::inner_mod2::method3;
```

在inner_mod2模块中定义了几个函数。method1用了pub（self）限制，那么它最多只能被这个模块以及子模块使用，在模块外部调用或者重新导出都会出错。method2用了pub（super）限制，那么它的可见性最多就只能到inner_mod1这一层，在这层外面不能被调用或者重新导出。而method3用了pub（crate）限制，那么它的可见性最多就只能到当前crate这一层，再继续往外重新导出，就会出错。

32.3.3 use关键字

前面我们用了很多完整路径来访问元素。比如：

```
::top_mod1::inner_mod1::method2();
```

Rust里面的路径有不同写法，它们代表的含义如下：

·以：：开头的路径，代表全局路径。它是从crate的根部开始算的：

```
mod top_mod1 {
    pub fn f1() {}
}

mod top_mod2 {
    pub fn call() {
        ::top_mod1::f1(); // 当前crate下的top_mod1
    }
```

```
}
```

·以super关键字开头的路径是相对路径。它是从上层模块开始算的：

```
mod top_mod1 {
    pub fn f1() {}

    mod inner_mod1 {
        pub fn call() {
            super::f1(); // 当前模块 inner_mod1 的父级模块中的f1函数
        }
    }
}
```

·以self关键字开头的路径是相对路径。它是从当前模块开始算的：

```
mod top_mod1 {
    pub fn f1() {}

    pub fn call() {
        self::f1();  // 当前模块 top_mod1中的f1函数
    }
}
```

如果我们需要经常重复性地写很长的路径，那么可以使用use语句把相应的元素引入到当前的作用域中来。

·use语句可以用大括号，一句话引入多个元素：

```
use std::io::{self, Read, Write}; // 这句话引入了io / Read / Write 三个名字
```

·use语句的大括号可以嵌套使用：

```
use a::b::{c, d, e::{f, g::{h, i}} };
```

·use语句可以使用星号，引入所有的元素：

```
use std::io::prelude::*;  // 这句话引入了 std::io::prelude下面所有的名字
```

·use语句不仅可以用在模块中，还可以用在函数、trait、impl等地方：

```
fn call() {
    use std::collections::HashSet;

    let s = HashSet::<i32>::new();
}
```

·use语句允许使用as重命名，避免名字冲突：

```
use std::result::Result as StdResult;
use std::io::Result as IoResult;
```

第33章　错误处理

错误处理指的是处理程序的非正常执行流程。比如，我们要打开一个文件，就不能只考虑文件正常打开情况，在实际中有可能因为各种原因，这个文件无法正常打开。这种时候我们就需要处理这些非正常的执行流程。

Rust把错误分成了两大类。一类是不可恢复错误，建议使用panic来处理。对于不可恢复错误，本质上没有办法在程序执行阶段做好处理的，那么就应该用panic让程序主动退出，由开发者来修复源码，这是唯一合理的方案。另外一类错误是可恢复错误，一般使用返回值来处理。比如打开文件出错这种问题，应该是设计阶段能预计到的，可以在执行阶段更好处理的问题，就适合采用这种方案。本章主要关注这一类的错误处理。

33.1 基本错误处理

Rust的错误处理机制，主要还是基于返回值的方案。不过因为拥有代数类型系统这套机制，所以它比C语言那种原始的错误码方案表达能力更强一点。Rust用于错误处理的最基本的类型就是我们常见的Option<T>类型。比如，内置字符串类型有一个find方法，查找一个子串：

```
impl str {
    pub fn find<'a, P: Pattern<'a>>(&'a self, pat: P) -> Option<usize> {}
}
```

这个方法当然是可能失败的，它有可能找不到。为了表达“成功返回了一个值”以及“没有返回值”这两种情况，Option<usize>就是一个非常合理的选择。而在传统的C语言里面，由于缺乏代数类型系统，往往会选择使用返回类型中的某些特殊值来表示非正常的情况。比如，C标准库中的子串查找函数的签名是这样的：

```
char *strstr( const char* str, const char* substr ) {}
```

返回的就是char*指针，使用空指针代表没找到的情况。

对于这种简单的错误处理，这么凑合一下问题也不大。如果错误信息更复杂一些，比如既需要用错误码表示错误的原因，又需要返回正常的返回值，就麻烦一些了。一般C语言的做法是，用返回值代表错误码，把真正需要返回的内容使用指针通过参数传递出去。比如C99里面打开文件的函数是这样设计的：

```
FILE *fopen( const char *restrict filename, const char *restrict mode );
```

到了C11又新增了一个支持多个错误码的打开文件的函数：

```
errno_t fopen_s(FILE *restrict *restrict streamptr,
                const char *restrict filename,
```

```
                const char *restrict mode);
```

这种方式明显是牺牲了可读性和使用方便性的。

在Rust里面就不用这么麻烦了。我们可以使用Result<T，E>类型来处理这种情况，干净利落：

```
impl File {
    pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File> {}
}
```

上面这个例子使用了std：：io：：Result类型，而不是std：：result：：Result类型。但是实际上它们是一回事，因为在io模块中有个类型别名的定义：

```
pub type Result<T> = result::Result<T, Error>;
```

这就是说，std：：io：：Result<T>等于std：：result：：Result<T，std：：io：：Error>。只是把泛型中的E参数定成了std：：io：：Error这个具体类型而已。

代数类型系统是错误处理的利器。我们再看一个例子。标准库中有一个FromStr trait：

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

如果我们想从一个字符串构造出一个类型的实例，就可以对这个类型实现这个trait。当然这个转换是有可能出错的，所以from_str方法一定要返回一个Result类型。比如针对bool类型实现的这个trait：

```
impl FromStr for bool {
    type Err = ParseBoolError;
    #[inline]
    fn from_str(s: &str) -> Result<bool, ParseBoolError> {
        match s {
            "true"  => Ok(true),
```

```
            "false" => Ok(false),
            _       => Err(ParseBoolError { _priv: () }),
        }
    }
}
```

正常情况是bool类型，异常情况是ParseBoolError类型。这个ParseBoolError不需要携带其他额外信息，所以它是一个空结构体就够了。

我们再看看FromStr针对f32的实现。可以看到，从字符串解析浮点数可能出现的错误种类更多，所以错误类型被设计成了一个enum：

```
enum FloatErrorKind {
    Empty,
    Invalid,
}
```

最后我们再看看FromStr针对String的实现。因为从&str到String的这个转换一定是可以成功的，不存在失败的可能，所以这种情况下错误类型被设计成了空的enum：

```
pub enum ParseError {}
```

前面我们已经说过了，空的enum就是bottom type，等同于发散类型！。所以这个错误实际上没有任何额外性能开销，证明如下：

```
use std::str::FromStr;
use std::string::ParseError;
use std::mem::{size_of, size_of_val};

fn main() {
    let r : Result<String, ParseError> = FromStr::from_str("hello");
    println!("Size of String: {}", size_of::<String>());
    println!("Size of `r`: {}", size_of_val(&r));
}
```

这个返回类型Result<String，ParseError>实际上和String是同构的。

所以说，Rust的这套错误处理机制既具备良好的抽象性，也具备无额外性能损失的优点。

33.2 组合错误类型

利用代数类型系统做错误处理的另外一大好处是可组合性（composability）。比如Result类型有这样的一系列成员方法：

```
fn map<U, F>(self, op: F) -> Result<U, E> where F: FnOnce(T) -> U
fn map_err<F, O>(self, op: O) -> Result<T, F> where O: FnOnce(E) -> F
fn and<U>(self, res: Result<U, E>) -> Result<U, E>
fn and_then<U, F>(self, op: F) -> Result<U, E> where F: FnOnce(T) -> Result<U, E>
fn or<F>(self, res: Result<T, F>) -> Result<T, F>
fn or_else<F, O>(self, op: O) -> Result<T, F> where O: FnOnce(E) -> Result<T, F>
```

这些方法的签名稍微有点复杂，涉及许多泛型参数。它们之间的区别也就表现在方法签名中。我们可以用下面的方式去掉语法干扰之后，来阅读函数签名，从而理解这些方法之间的区别：

方　法	方法类型	参数类型
map	`Result<T, E> -> Result<U, E>`	`T -> U`
map_err	`Result<T, E> -> Result<T, F>`	`E -> F`
and	`Result<T, E> -> Result<U, E>`	`Result<U, E>`
and_then	`Result<T, E> -> Result<U, E>`	`T -> Result<U, E>`
or	`Result<T, E> -> Result<T, F>`	`Result<T, F>`
or_else	`Result<T, E> -> Result<T, F>`	`E -> Result<T, F>`

通过这个表格的对比，我们可以很容易看出它们之间的区别。比如map和and_then的主要区别是闭包参数：map的参数是做的T->U的转换，而and_then的参数是T->Result的转换。Option类型也有类似的对应的方法，读者可以自己建一个表格，对比一下这些方法签名之间的区别。

下面用一个示例演示一下这些组合函数的用法：

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(
            |err| err.to_string()))
        .map(|n| 2 * n)
```

```
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

33.3 问号运算符

Result类型的组合调用功能很强大，但是它有一个缺点，就是经常会发生嵌套层次太多的情况，不利于可读性。比如下面这个示例：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
        .map_err(|err| err.to_string())
        .and_then(|mut file| {
            let mut contents = String::new();
            file.read_to_string(&mut contents)
                .map_err(|err| err.to_string())
                .map(|_| contents)
        })
        .and_then(|contents| {
            contents.trim().parse::<i32>()
                    .map_err(|err| err.to_string())
        })
        .map(|n| 2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

这说明我们还有继续改进的空间。为了方便用户，Rust设计组在前面这套系统的基础上，又加入了一个问号运算符，用来简化源代码。这个问号运算符完全是建立在前面这套错误处理机制上的语法糖。

问号运算符意思是，如果结果是Err，则提前返回，否则继续执行。

使用问号运算符，我们可以把file_double函数简化成这样：

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = File::open(file_path).map_err(|e| e.to_string())?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .map_err(|err| err.to_string())?;
```

```
    let n = contents.trim().parse::<i32>()
            .map_err(|err| err.to_string())?;

    Ok(2 * n)
}
```

这里依然有不少的map_err调用，主要原因是返回类型限制成了Result<i32，String>。如果改一下返回类型，代码还能继续精简。

因为这段代码总共有两种错误：一种是io错误，用std：：io：：Error表示；另外一种是字符串转整数错误，用std：：num：：ParseIntError表示。我们要把这两种类型统一起来，所以使用了一个自定义的enum类型，这样map_err方法调用就可以省略了。我们再补充这两种错误类型到自定义错误类型之间的类型转换，问题就解决了。完整源码如下所示：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

#[derive(Debug)]
enum MyError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
}

impl From<std::io::Error> for MyError {
    fn from(error: std::io::Error) -> Self {
        MyError::Io(error)
    }
}

impl From<std::num::ParseIntError> for MyError {
    fn from(error: std::num::ParseIntError) -> Self {
        MyError::Parse(error)
    }
}

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, MyError> {
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n = contents.trim().parse::<i32>()?;
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

这样一来，这个file_double函数就精简太多了。它只需管理正常逻辑，对于可能出错的分支，直接一个问号操作符提前返回，错误处理和正常逻辑互不干扰，清晰易读。

下面继续讲解一下问号运算符背后做了什么事情。跟其他很多运算符一样，问号运算符也对应着标准库中的一个trait std：：ops：：Try。它的定义如下：

```
trait Try {
    type Ok;
    type Error;
    fn into_result(self) -> Result<Self::Ok, Self::Error>;
    fn from_error(v: Self::Error) -> Self;
    fn from_ok(v: Self::Ok) -> Self;
}
```

编译器会把expr？这个表达式自动转换为以下语义（不在catch块内的情况）：

```
match Try::into_result(expr) {
    Ok(v) => v,

    // here, the `return` presumes that there is
    // no `catch` in scope:
    Err(e) => return Try::from_error(From::from(e)),
}
```

哪些类型支持这个问号表达式呢？标准库中已经为Option、Result两个类型impl了这个trait：

```
impl<T> ops::Try for Option<T> {
    type Ok = T;
    type Error = NoneError;

    fn into_result(self) -> Result<T, NoneError> {
        self.ok_or(NoneError)
    }

    fn from_ok(v: T) -> Self {
        Some(v)
    }

    fn from_error(_: NoneError) -> Self {
        None
    }
```

```
}

impl<T,E> ops::Try for Result<T, E> {
    type Ok = T;
    type Error = E;

    fn into_result(self) -> Self {
        self
    }

    fn from_ok(v: T) -> Self {
        Ok(v)
    }

    fn from_error(v: E) -> Self {
        Err(v)
    }
}
```

把这些综合起来，我们就能理解对于Result类型，执行问号运算符做了什么了。其实就是碰到Err的话，调用From trait做个类型转换，然后中断当前逻辑提前返回。

和Try trait一起设计的，还有一个临时性的do catch语法。在使用do catch的情况下，问号运算符就不是直接退出函数，而是退出do catch块。示例如下：

```
fn file_double<P: AsRef<Path>>(file_path: P) {
    let r : Result<i32, MyError> = do catch {
        let mut file = File::open(file_path)?;
        let mut contents = String::new();
        file.read_to_string(&mut contents)?;
        let n = contents.trim().parse::<i32>()?;
        Ok(2 * n)
    };
    match r {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

目前使用这个语法需要打开#！[feature（catch_expr）]这个feature gate。而且语法也是do catch{}这样的写法。这么做是为了避免导致代码不兼容的问题。以前的版本中，catch这个单词不是关键字，可能就有用户使用了catch这个名字作为标识符名字。如果我们直接把这个单词升级为关键字，必然导致某些现存代码编译错误。所以引入关键字这种事情，一定要通过edition的版本更迭来完成。在Rust 2018 edition中，所有使用catch作为标识符的代码都会生成一个警告，但依然编译通过。再下

一个edition的时候，catch就可以提升为正式关键字了，到那时，就不需要do catch{}语法，而是直接使用catch{}了。（以后也可能选择try作为关键字，目前还没有定论。）

大家可能又发现，如果使用问号运算符，主要逻辑那部分确实已经简化得非常干净，但是其他部分的代码量又有了增长。我们需要定义新的错误类型，实现一些trait，才能让它工作起来。这部分能不能更简化一点呢？答案是肯定的。

其中一个方案是，使用trait object来替换enum。实际上trait object和enum有很大的相似性，它们都可以把一系列的类型统一成一个类型。恰好标准库内部给我们提供了一个trait，来统一抽象所有的错误类型，它就是std：：error：：Error：

```
pub trait Error: Debug + Display {
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&dyn Error> { ... }
}
```

所有标准库里面定义的错误类型都已经实现了这个trait。所以，我们可以想象，错误类型其实可以表示成Box<dyn Error>。下面用这种方式来精简一下file_double这个例子：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) ->
            Result<i32, Box<dyn std::error::Error>> {
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n = contents.trim().parse::<i32>()?;
    Ok(2 * n)
}


fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

上面这段代码也可以编译通过。因为std：：io：：Error和std：：num：：Parse-Int-Error类型都实现了std：：error：：Error trait，都可以转换为Box<dyn std：：error：：Error>这个trait object。

统一用trait object来接收所有错误是最简单的写法，但它也有缺点。最大的问题是，它不方便向下转型。如果外面的调用者希望针对某些类型做特殊的错误处理，就很难办。除非你不需要对任何错误类型做任何有区分的处理。这种写法适合一些简单的小工具。

使用enum表达错误类型，可以最精确地表达错误信息。当然带来的一个后果是，被调用者的enum错误类型发生变化的时候（比如给enum增加一个成员），会导致调用者那边编译失败，这是由类型系统保证的。很多情况下，这其实是设计者愿意看到的结果，改变错误类型本质上就是改变了API，此事不该在调用者完全不知情的条件下默默进行。当然，有些情况下设计者的本意如果就是希望新增加一种错误类型但不影响下游用户的兼容性。这也是有办法的，那就是最开始的版本就给这个enum类型加上#[non_exhaustive]标签。这样调用者那边的代码在做模式匹配的时候，无论如何都要写一条默认分支。以后给enum新加一个成员，就不会造成编译错误，调用者那边的流程会执行最开始的那条默认分支。具体要不要使用这个标签，就取决于设计者的意图了。

33.4 main函数中使用问号运算符

新加入的问号运算符给main函数带来了一个挑战。因为问号运算符会return一个Result类型，如果它所处的函数签名不是返回的Result类型，一定会出现类型匹配错误。而main函数一开始的时候是定义成fn（）->（）类型的，所以问号运算符不能在main函数中使用。这显然是一个问题，解决这个问题的办法就是——修改main函数的签名类型。

我们希望：main函数既可以返回unit类型，不破坏以前的旧代码；又可以返回Result类型，支持使用问号运算符。所以，最简单的办法就是使用泛型，兼容这两种类型。Rust在标准库中引入了一个新的trait：

```
pub trait Termination {
    /// Is called to get the representation of the value as status code.
    /// This status code is returned to the operating system.
    fn report(self) -> i32;
}
```

main函数的签名就对应地改成了fn<T：Termination>（）->T。标准库为Result类型、（）类型、bool类型以及发散类型！实现了这个trait。所以这些类型都可以作为main函数的返回类型了。

它是怎么启动起来的呢？是因为Runtime库中有这样的一个函数，它调用了用户写的main函数：

```
#[lang = "start"]
fn lang_start<T: ::termination::Termination + 'static>
    (main: fn() -> T, argc: isize, argv: *const *const u8) -> isize
{
    lang_start_internal(&move || main().report(), argc, argv)
}
```

在最终的可执行代码中，程序刚启动的时候要先执行一些Runtime自带的逻辑，然后才会进入用户写的main函数中去。

33.5 新的Failure库

标准库中现存的Error trait有几个明显问题：

·description方法基本没什么用；

·无法回溯，它没有记录一层层的错误传播的过程，不方便debug；

·Box<Error>不是线程安全的。

Failure这个库就是为了进一步优化错误处理而设计的。它主要包含三个部分。

·新的failure：：Fail trait，是为了取代std：：error：：Error trait而设计的。它包含了更丰富的成员方法，且继承于Send+Sync，具备线程安全特性。

·自动derive机制，主要是让编译器帮用户写一些重复性的代码。

·failure：：Error结构体。所有其他实现了Fail trait的错误类型，都可以转换成这个类型，而且它提供了向下转型的方法。

使用failure来实现前面那个示例，代码如下：

```
#[macro_use]
extern crate failure;

use std::fs::File;
use std::io::Read;
use std::path::Path;

#[derive(Debug, Fail)]
enum MyError {
    #[fail(display = "IO error {}.", _0)]
    Io(#[cause] std::io::Error),
    #[fail(display = "Parse error {}.", _0)]
    Parse(#[cause] std::num::ParseIntError),
}

impl From<std::io::Error> for MyError {
    fn from(error: std::io::Error) -> Self {
        MyError::Io(error)
    }
```

```
}

impl From<std::num::ParseIntError> for MyError {
    fn from(error: std::num::ParseIntError) -> Self {
        MyError::Parse(error)
    }
}

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, MyError> {
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n = contents.trim().parse::<i32>()?;
    Ok(2 * n)
}


fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

现在社区里已经有一些库转向使用failure做错误处理。它将来可能是Rust生态系统中主流的错误处理方式。

第34章　FFI

Rust有一个非常好的特性，就是它支持与C语言的ABI兼容。什么是ABI呢？维基百科是这么解释的：

In computer software，an application binary interface（ABI）is an interface between two program modules；often，one of these modules is a library or operating system facility，and the other is a program that is being run by a user.

An ABI defines how data structures or computational routines are accessed in machine code，which is a low-level，hardware-dependent format.

——Wikipedia

所以，我们可以用Rust写一个库，然后直接把它当成C写的库来使用。或者反过来，用C写的库，可以直接在Rust中被调用。而且这个过程是没有额外性能损失的。C语言的ABI是这个世界上最通用的ABI，大部分编程语言都支持与C的ABI兼容。这也意味着，Rust与其他语言之间的交互是没问题的，比如用Rust为Python/Node.js/Ruby写一个模块等。

本章主要讲解Rust如何与其他语言进行交互，为了让示例尽可能地简洁、清晰，本章主要关注的是Rust如何与C语言进行交互。至于其他语言，搞清楚它们如何与C语言交互就完全可以掌握它们与Rust的交互办法，所以无须一个个地分别讲解。

34.1 什么是FFI

所谓的FFI指的是：

A foreign function interface（FFI）is a mechanism by which a program written in one programming language can call routines or make use of services written in another.

——Wikipedia

Rust不支持源码级别与其他语言的交互，因为这么做代价很大、限制太多，所以需要用库的方式来互相调用。这就要求我们有能力用Rust生成与C的ABI兼容的库。通过rustc-h命令我们可以看到，Rust编译器支持生成这样一些种类的库：

```
--crate-type [bin|lib|rlib|dylib|cdylib|staticlib|proc-macro]
            Comma separated list of types of crates for the
            compiler to emit
```

其中，cdylib和staticlib就是与C的ABI兼容的。分别代表动态链接库和静态链接库。在编译的时候，我们需要指定这样的选项才能生成合适的目标文件。指定目标文件类型有两种方式：

·在编译命令行中指定，如rustc--crate-type=staticlib test.rs；

·在源代码入口中指定，如#！[crate_type="staticlib"]。

另外，我们还需要注意，C的ABI以及运行时库也不是完全统一的。此事是由rustup工具管理的。执行rustup show可以看到当前使用的工具链是什么，比如笔者当前的工具链是：

```
Default host: x86_64-unknown-linux-gnu
```

这意味着用这套工具链生成的C库是和gcc工具链的ABI兼容的。如果读者需要生成与MSVC的ABI兼容的库，那么需要使用：

```
rustup target add x86_64-pc-windows-msvc
```

我们还可以用rustup来下载Android系统的工具链，实现交叉编译，等等。关于工具链以及C运行时库的链接方式的问题，读者可以参考rustup的官方网站。

除了指定目标文件、工具链之外，更重要的是需要注意接口的设计。不是所有的Rust的语言特性都适合放到交互接口中的。比如，Rust中有泛型，C语言里面没有，所以泛型这种东西是不可能暴露出来给C语言使用的，这就不是C语言的ABI的一部分。只有符合C语言的调用方式的函数，才能作为FFI的接口。这样的函数有以下基本要求：

·使用extern"C"修饰，在Rust中extern fn默认等同于extern"C"fn；

·使用#[no_mangle]修饰函数，避免名字重整；

·函数参数、返回值中使用的类型，必须在Rust和C里面具备同样的内存布局。

下面我们用示例来说明如何实现FFI。

34.2 从C调用Rust库

假设我们要在Rust中实现一个把字符串从小写变大写的函数，然后由C语言调用这个函数。实现代码如下：

```
#[no_mangle]
pub extern "C" fn rust_capitalize(s: *mut c_char)
{
    unsafe {
        let mut p = s as *mut u8;
        while *p != 0 {
            let ch = char::from(*p);
            if ch.is_ascii() {
                let upper = ch.to_ascii_uppercase();
                *p = upper as u8;
            }
            p = p.offset(1);
        }
    }
}
```

我们在Rust中实现这个函数，考虑到C语言调用的时候传递的是char*类型，所以在Rust中我们对应的参数类型是*mut std：：os：：raw：：c_char。这样两边就对应起来了。

这个函数是要被外部的C代码调用的，所以一定要用extern"C"修饰。用#[no_mangle]修饰主要是为了保证导出的函数名字和源码中的一致。这个并不是必须的，我们还可以使用#[export_name="my_whatever_name"]来指定导出名字。在某些时候，我们需要导出的函数名恰好在Rust中跟某个关键字发生了冲突，就可以用这种方式来规避。

使用如下编译命令，可以生成一个与C的ABI兼容的静态库。

```
rustc --crate-type=staticlib capitalize.rs
```

下面我们再写一个调用这个函数的C程序：

```
#include <stdlib.h>
#include <stdio.h>
```

```
// declare
extern void rust_capitalize(char *);

int main() {
    char str[] = "hello world";
    rust_capitalize(str);
    printf("%s\n", str);
    return 0;
}
```

使用如下命令编译链接：

```
gcc -o main main.c -L. -l:libcapitalize.a -Wl,--gc-sections -lpthread -ldl
```

可以正确生成可执行程序。执行代码，可见该程序完成了预期中的功能。

34.3 从Rust调用C库

这个例子我们反过来，从Rust中调用C写的库。C的实现如下所示：

```
int add_square(int a, int b)
{
    return a * a + b * b;
}
```

使用如下命令可以生成对应的静态库：

```
gcc -c -Wall -Werror -fpic simple_math.c
ar  rcs libsimple_math.a simple_math.o
```

现在我们到Rust中调用这个静态库：

```
use std::os::raw::c_int;

#[link(name = "simple_math")]
extern "C" {
    fn add_square(a: c_int, b: c_int) -> c_int;
}

fn main() {
    let r = unsafe { add_square(2, 2) };
    println!("{}", r);
}
```

使用如下命令编译链接：

```
rustc -L . call_math.rs
```

参数-L可以指定依赖库的查找路径，具体的名字可以通过#[link（name="library_name"）]来指定。

34.4 更复杂的数据类型

对于交互接口中的简单类型，我们直接使用标准库中定义好的std：：os：：raw里面的类型就够了。而更复杂的类型就需要我们手动封装了。比如结构体就需要用#[repr（C）]修饰，以保证这个结构体在Rust和C双方的内存布局是一致的。

如果我们需要做的是跟常见的操作系统交互，许多常用的数据结构都已经有人封装好了，可以在crates.io找libc库直接使用。接下来我们用一个示例演示一下在接口中包含结构体该怎样做。这个示例同时也使用了cargo来管理Rust项目，且使用动态链接库的方式执行。

Rust项目Cargo.toml如下所示：

```
[package]
name = "log"
version = "0.1.0"
authors = ["F001"]

[dependencies]
libc = "0.2"

[lib]
name = "rust_log"
crate-type = ["cdylib"]
```

src/lib.rs文件内容如下所示：

```
#![crate_type = "cdylib"]

extern crate libc;

use libc::{c_int, c_char};
use std::ffi::CStr;

// this struct is used as an public interface
#[repr(C)]
#[no_mangle]
pub struct RustLogMessage {
    id: c_int,
    msg: *const c_char
}

#[no_mangle]
pub extern "C" fn rust_log(msg: RustLogMessage) {
```

```
    let s = unsafe { CStr::from_ptr(msg.msg) };
    println!("id:{} message:{:?}", msg.id, s);
}
```

使用cargo build就可以编译出对应的动态库。

C语言的调用代码如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
struct RustLogMessage {
    int id;
    char *msg;
};

int main()
{
    void *rust_log_lib;

    void (*rust_log_fn)(struct RustLogMessage msg);

    rust_log_lib = dlopen("./rust_log/target/debug/librust_log.so", RTLD_LAZY);
    if ( rust_log_lib != NULL ) {
        rust_log_fn = dlsym(rust_log_lib, "rust_log");
    } else {
        printf("load so library failed.\n");
        return 1;
    }

    for (int i = 0; i < 10; i++) {
        struct RustLogMessage msg = {
            id : i,
            msg : "string in C\n",
        };
        rust_log_fn(msg);
    }

    if (rust_log_lib != NULL ) dlclose(rust_log_lib);

    return EXIT_SUCCESS;
}
```

编译命令为gcc main.c-ldl。执行程序之前，要保证动态链接库和可执行程序之间的相对路径关系是正确的，然后执行。可见，参数正确地在Rust和C之间传递了。

第35章　文档和测试

35.1　文档

Rust也支持使用注释来编写规范的文档。可以使用rustdoc工具把源码中的文档提取出来，生成易读的HTML等格式。在cargo里面可以用cargo doc命令生成文档。

普通的注释有两种，一种是用//开头的，是行注释，一种是/**/，是块注释。这些注释不会被视为文档的一部分。特殊的文档注释是///、//！、/**…*/、/*！…*/，它们会被视为文档。

跟attribute的规则类似：用///开头的文档被视为是给它后面的那个元素做的说明；//！开头的文档被视为是给包含这块文档的元素做的说明。/**…*/和/*！…*/也是类似的。示例如下：

```
mod foo {
    //! 这块文档是给 `foo` 模块做的说明

    /// 这块文档是给函数 `f` 做的说明
    fn f() {
    // 这块注释不是文档的一部分
    }
}
```

文档内部支持markdown格式。可以使用#作为一级标题。比如标准库中常用的几种标题：

```
/// # Panics
/// # Errors
/// # Safety
/// # Examples
```

文档中的代码部分要用``符号把它们括起来。代码块应该用```括起来。比如：

```
    /// ```
    /// let mut vec = Vec::with_capacity(10);
```

```
    ///
    /// // The vector contains no items, even though it has capacity for more
    /// assert_eq!(vec.len(), 0);
    /// ```
```

Rust文档里面的代码块，在使用cargo test命令时，也是会被当做测试用例执行的。这个设计可以在很多时候检查出文档和代码不对应的情况。

如果文档太长，也可以写在单独的markdown文件中。如果在单独的文件中写文档，就不需要再用///或者//！开头了，直接写内容就可以。然后再用一个attribute来指定给对应的元素：

```
#![feature(external_doc)]

#[doc(include = "external-doc.md")]
pub struct MyAwesomeType;
```

35.2 测试

Rust内置了一套单元测试框架。单元测试是一种目前业界广泛使用的，可以显著提升代码可靠性的工程管理手段。Rust里面的单元测试代码可以直接和业务代码写在一个文件中，非常有利于管理，方便更新。执行单元测试也非常简单，一条cargo test命令即可。

一般情况下，如果我们新建一个library项目，cargo工具会帮我们在src/lib.rs中自动生成如下代码：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

这就是最基本的单元测试框架。下面详细介绍一下这里面的各个要素。

首先，Rust里面有一个特殊的attribute，叫作#[cfg]。它主要是用于实现各种条件编译。比如#[cfg（test）]意思是，这部分代码只在test这个开关打开的时候才会被编译。它还有更高级的用法，比如

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

我们还可以自定义一些功能开关。比如在Cargo.toml中加入这样的代码：

```
[features]
# 默认开启的功能开关
default = []

# 定义一个新的功能开关,以及它所依赖的其他功能
# 我们定义的这个功能不依赖其他功能,默认没有开启
```

```
my_feature_name = []
```

之后就可以在代码中使用这个功能开关，某部分代码可以根据这个开关的状态决定编译还是不编译：

```
#[cfg(feature = "my_feature_name")]
mod sub_module_name {
}
```

这个开关究竟是开还是关，可以通过编译选项传递进去：

```
cargo build --features "my_feature_name"
```

当我们使用cargo test命令的时候，被#[cfg（test）]标记的代码就会被编译执行；否则直接被忽略。

我们还是用一个示例来说明。我们现在准备实现一个辗转相除法求最大公约数的功能。新建一个名叫gcd的项目：

```
cargo new --lib gcd
```

辗转相除法的细节就不展开了。实现代码如下所示：

```
pub fn gcd(a: u64, b: u64) -> u64
{
    let (mut l, mut g) = if a < b {
        (a, b)
    } else {
        (b, a)
    };

    while l != 0 {
        let m = g % l;
        g = l;
        l = m;
    }
    return g;
}
```

接下来添加一个最基本的测试：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(gcd(2, 3), 1);
    }
}
```

使用cargo test命令执行这个测试。这一次发生了编译错误，编译器找不到gcd这个函数。这是因为我们把测试用例写在了一个单独的模块中，在子模块中并不能直接访问父模块中的内容。在mod内部加一句use gcd；或者use super：：*；可以解决这个问题。

```
      Compiling gcd v0.1.0 (file:///projects/gcd)
          Finished dev [unoptimized + debuginfo] target(s) in 2.33 secs
              Running target/debug/deps/gcd-1658b34b1de16a01

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

    Doc-tests gcd

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

打印出来的结果非常清晰易读。前面一部分是执行测试模块中的测试用例的结果，后面一部分是执行文档中的测试用例的结果。1 passed代表通过了1个测试。0 failed代表失败了0个测试。0 ignored代表忽略了0个测试。

用户可以用#[ignore]标记测试用例，暂时忽略这个测试。比如：

```
#[test]
#[ignore]
fn it_works() {
    assert_eq!(gcd(2, 3), 1);
}
```

0 measured代表跑了0个benchmark性能测试。我们可以用#[bench]添加性能测试用例：

```
#[cfg(test)]
mod tests {
    use super::*;
    use self::test::Bencher;

    #[bench]
    fn big_num(b: &mut Bencher) {
        b.iter(|| gcd(12345, 67890) )
    }
}
```

这个功能目前还没有稳定，需要用户在当前crate中开启feature gate：

```
#![feature(test)]
extern crate test;
```

然后使用cargo bench就可以执行这个性能测试。这时就可以看到测试结果中有1 measured的结果。

测试结果中还有0 filtered out统计数据。这个数据代表的是用户跑测试的时候过滤出来了多少测试用例。比如我们有许多测试用例，但是只想执行某一个具体的测试用例it_works，可以这样做：cargo test it_works。或者可以用开头几个字母过滤出多个测试用例，比如cargo test it。还有更多的用法可以参见cargo test-h。

在测试用例内部，我们一般用assert_eq！宏来检查真实结果和预期结果是否一致。除此之外，也还有其他的检查方法。assert！宏可以用于检查一个bool类型结果是否为true。assert_ne！宏可以用于检查两个数据是否不相等。另外，我们还可以在这些宏里面自定义错误信息。比如我们用0来测试上面这个gcd函数。因为0作为除数没有意义，所以我们希望任何一个参数为0的时候，输出结果都是0，可以写这样的测试用例：

```
#[test]
fn with_zero() {
    assert_eq!(gcd(10, 0), 0, "division by zero has no meaning");
}
```

如果测试失败，失败消息中会显示我们指定的那条信息。

另外，有些时候测试结果无法用“等于不等于”这种条件表达，比如发生panic。测试框架也允许我们用#[should_panic]做这种测试。假设，我们修改一下上面的gcd函数的逻辑，不允许参数为0，如果参数是0直接发生panic：

```
pub fn gcd(a: u64, b: u64) -> u64
{
    if a == 0 || b == 0 {
        panic!("Parameter should not be zero");
    }
    …
}
```

为了测试这种情况，我们可以写如下测试用例：

```
#[test]
#[should_panic]
fn with_zero() {
    gcd(10, 0);
}
```

一般我们都把测试用例放到单独的mod里面，打上#[cfg（test）]条件编译的标签，这样编译正常代码的时候就可以把测试相关的整个模块忽略掉。这个测试模块一般直接放在被测试代码的同一个文件中：一方面是比较直观容易管理；另一方面，根据Rust的模块可见性规则，这个测试模块有权访问它父模块的私有元素，这样比较方便测试。

用户也可以自己组织测试用例的代码结构。比如单独使用一个新的文件夹来管理测试用例，这都是没问题的。毕竟测试代码也不过就是一个普通的模块而已，我们可以自由选择如何管理这个模块。

Rust默认的测试框架毕竟还只是一个轻量级的框架。功能比许多其他语言中的大型测试框架要差一些。这也是目前Rust设计组比较关心的一个领域。他们正在设计一个方案，使用户可以比较方便地实现自定义测试框架。这样可以由社区开发一些功能更强大的测试框架作为替代品，供大家使用。

附录　词汇表

ABI	Application Binary Interface
Alias	别名
Arc	Atomic Reference Counter
Associated Item	关联条目，包括关联类型、关联方法、关联常量等
Atomic	原子的
Attribute	属性，在 Rust 中跟宏性质是一样的东西，语法外观不同
Borrow Check	借用检查
Box	具有所有权的智能指针
Cargo	Rust 的官方包管理工具
Closure	闭包
Compile Unit	编译单元
Constructor	构造器
Copy	特殊 trait，代表类型默认是复制语义
Crate	Rust 的基本编译单元
Data Race	数据竞争
Deref	解引用
Destructor	析构函数
Destructure	解构
diverge function	发散函数，永远不会返回的函数
DST	Dynamic Sized Type 编译阶段无法确定大小的类型
Dynamic Trait Type	Trait Object 的新名字
Fat Pointer	胖指针，即还携带额外信息的指针
Feature Gate	功能开关
FFI	Foreign Function Interface

（续）

Fn/FnMut/FnOnce	闭包相关的系列 trait
Generic	泛型
Higher Rank Lifetime	高阶生命周期
Higher Rank Trait Bounds	高阶 trait 约束
Higher Rank Type	高阶类型系统
ICE	Internal Compiler Error 编译器内部错误
inherited mutability	承袭可变性
Interior Mutability	内部可变性
intrinsics	编译器内置函数
Iterator	迭代器
Lifetime	生命周期
Lifetime Elision	生命周期省略
Lint	可自定义扩展的编译阶段检查
LLVM	Low Level Virtual Machine
Macro	宏
Memory Safe	内存安全，Rust 的内存安全主要指没有段错误 Segmentation fault
MIR	Middle-level IR
Module	模块
Move	移动
Mutability	可变性
NLL	Non Lexical Lifetime，非词法生命周期
Object Safe	能正确构造 trait object 的规则
OIBIT	opt-in builtin traits，新名字为 Auto trait
Orphan Rules	孤儿规则
Ownership	所有权
Panic	恐慌，在 Rust 中用于不可恢复错误处理
Pattern Match	模式匹配
Placement New	在用户指定的内存位置上构建新的对象
Playpen	指的是 http://play.rust-lang.org 网站。这个网站提供了方便的编写、编译、执行 Rust 代码的能力
POD	Plain Old Data
Prelude	预先声明的自动被包含到每个源码中的内容
Race Condition	竞态条件
RAII	Resource Acquisition Is Initialization 是 C++ 等编程语言常用的管理资源方法
Rc	Reference Counted 引用计数智能指针
Release Channel	发布渠道
RFC	Request For Comments 语言设计提案，FCP 指 Final Comment Period
Rustc	Rust 官方编译器的可执行文件名字
RVO	Return Value Optimization

（续）

Self/self	小写 s 是特殊变量，大写 S 是特殊类型
Send	特殊的 trait，代表变量可以跨线程转移所有权
Shadowing	遮蔽，变量允许遮蔽，类型和生命周期不允许
SIMD	Single Instruction Multiple Data 单指令多数据流
Sized	特殊的 trait，代表编译阶段类型的大小是已知的
Slice	数组切片
Specialization	泛型特化
Stack Unwind	栈展开
STL	standard template library 是 C++ 的标准模板库
Sync	特殊 trait，代表变量可以跨线程共享
TLS	Thread Local Storage 线程局部存储
Toml	一种文本文件格式
Trait Object	指向对象及其虚表的胖指针，以后会改名为 dynamic trait type
TWiR	This Week in Rust，一个很有信息量的网站
Type Inference	类型自动推导
UFCS	Universal Function Call Syntax, 通用函数调用语法， 后来改为 Fully Qualified Syntax
Unit Type	单元类型，即空 tuple，记为 ( )
Unsized Type	不满足 Sized 约束的类型
VTable	虚函数表
ZST	零大小数据类型