



Rust 语言真的好：连续八年成为全世界最受欢迎的语言、没有 GC 也无需手动内存管理、性能比肩 C++/C 还能直接调用它们的代码、安全性极高 - 总有公司说使用 Rust 后以前的大部分 bug 都将自动消失、全世界最好的包管理工具 Cargo 等等。但...

### 有人说: "Rust 太难了, 学了也没用"

对于后面一句话我们持保留意见, 如果以找工作为标准, 那国内环境确实还不好, 但如果你想成为更优秀的程序员或者是玩转开源, 那 Rust 还真是不错的选择, 具体原因见[下一章](#)。

至于 Rust 难学, 那正是本书要解决的问题, 如果看完后, 你觉得没有学会 Rust, 可以找我们退款, 哟抱歉, 这是开源书, 那就退 吧 :)

如果看到这里, 大家觉得这本书的介绍并没有吸引到你, 不要立即放弃, 强烈建议读一下[进入 Rust 编程世界](#), 那里会有不一样的精彩。

## 配套练习题

对于学习编程而言, 读一篇文章不如做几道练习题, 此话虽然夸张, 但是也不无道理。既然如此, 即读书又做练习题, 效果会不会更好? 再加上练习题是书本的配套呢? :P

- [Rust 语言实战](#), Rust 语言圣经配套习题, 支持中英双语, 可以在右上角切换

## 创作感悟

截至目前, Rust 语言圣经已写了 170 余章, 110 余万字, 历经 1000 多个小时, 每一个章节都是手动写就, 没有任何机翻和质量上的妥协( 相信深入阅读过的读者都能体会到这一点 )。

曾经有读者问过 "这么好的书为何要开源, 而不是出版?", 原因很简单: **只有完全开源才能完美地呈现出我想要的教学效果。**

总之, Rust 要在国内真正发展起来, 必须得有一些追逐梦想的人在做着不计付出的事情, 而我希望自己能贡献一份微薄之力。

但是要说完全无欲无求, 那也是不可能的, 看到项目多了一颗 , 那感觉...棒极了, 因为它代表了读者的认可和称赞。

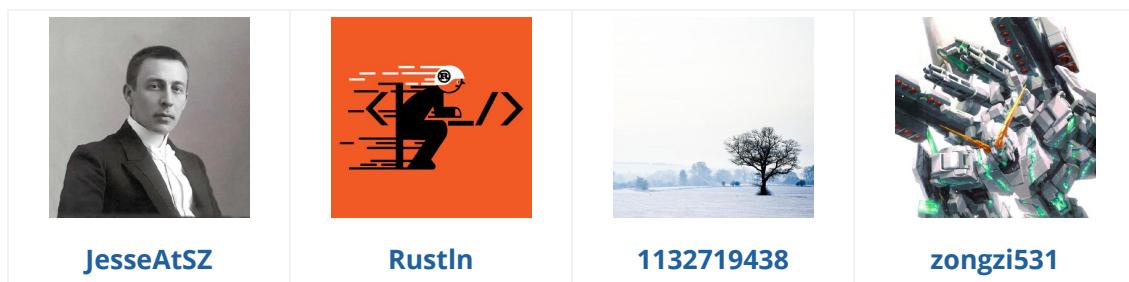
你们用指尖绘制的星空, 那里繁星点点, 每一颗都在鼓励着怀揣着开源梦想的程序员披荆斩棘、不断前行, 不夸张的说, 没有你们, 开源世界就没有星光, 自然也就不会有今天的开源盛世。

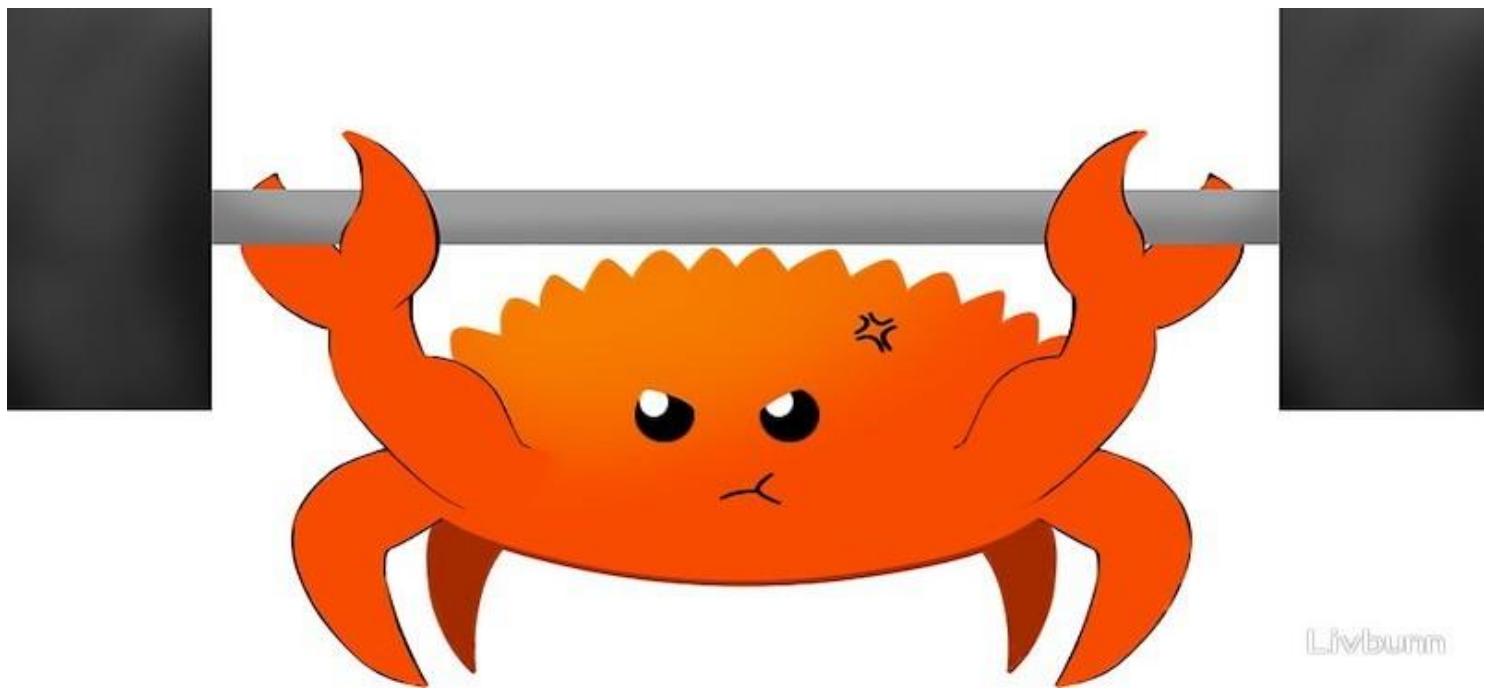
因此，我恳请大家，如果觉得书还可以，就在你的指尖星空绘制一颗新的 ，指引我们继续砥砺前行。  
这个人世间，因善意而美好。

最后，能通过开源在茫茫人海中与大家相识，这感觉真好 :D

## 🏆 贡献者

非常感谢本教程的[所有贡献者](#)，正是有了你们，才有了现在的高质量 Rust 教程！





Livburn

正在努力学 Rust 的 Ferris

## Rust 发展历程

Rust 最早是 Mozilla 雇员 Graydon Hoare 的个人项目。从 2009 年开始，得到了 Mozilla 研究院的资助，2010 年项目对外公布，2010 ~ 2011 年间实现自举。自此以后，Rust 在部分重构 -> 崩溃的边缘反复横跳（历程极其艰辛），终于，在 2015 年 5 月 15 日发布 1.0 版。

在紧锣密鼓的开发过程中，Rust 建立了一个强大且活跃的社区，形成一整套完善稳定的项目贡献机制（Rust 能够飞速发展，与这一点密不可分）。Rust 现在由 [Rust 项目开发者社区](#) 维护，[Rust 基金会](#) 赞助支持。

大家可能疑惑 Rust 为啥用了这么久才到 1.0 版本？与之相比，Go 语言 2009 年发布，却在 2012 年仅用 3 年就发布了 1.0 版本。

- 首先，Rust 语言特性较为复杂，所以需要全盘考虑的问题非常多；
- 其次，Rust 当时的参与者太多，七嘴八舌的声音很多，众口难调，而 Rust 开发团队又非常重视社区的意见；
- 最后，一旦 1.0 快速发布，那绝大部分语言特性就无法再被修改，对于有完美强迫症的 Rust 开发者团队来说，某种程度上的不完美是不可接受的。

因此，Rust 语用了足足 6 年时间，才发布了尽善尽美的 1.0 版本。

---

大家知道 Rust 的作者到底因为何事才痛下决心开发一门新的语言吗？

说来挺有趣，在 2006 年的某天，作者工作到精疲力尽后，本想回公寓享受下生活，结果发现电梯的程序出 Bug 崩溃了，要知道在国外，修理工可不像在中国那样随时待岗，还要知道，他家在 20 多楼！

最后，他选择了妥协，去酒店待几天等待电梯的修理。

当然，一般人可能就这样算了，毕竟忍几天就过去了嘛。但是这名伟大的程序员显然也不是一般人，他面对害他流离失所的电梯拿起了屠龙宝刀 - Rust。

自此，劈开一个全新的编程世界。

---

## 为何又来一门新语言？

简而言之，**因为还缺一门无 GC 且无需手动内存管理、性能高、工程性强、语言级安全性以及能同时得到工程派和学院派认可的语言**，而 Rust 就是这样的语言。你也可以回忆下熟悉的语言，看是不是有另外一门可以同时满足这些需求：）

至于 Rust 最为人诟病的点，那也就一个：学习曲线陡峭。不过当语言生态起来后，这都不算问题。

## 缓解内卷

有人说 Rust 作为新语言会增加内卷，其实恰恰相反，Rust 可以缓解内卷。为何不说 C++ 内卷，而说 Java、Python、JS 内卷？不就是后几个相对简单、上手容易嘛？而 Rust 怎么看也是 C++ 级别的上手难度。

其实从我内心不可告人的角度出发，并不希望 Rust 大众化，因为这样可以保饭碗、保薪资，还能拥有行业内的地位。但是从对 Rust 的喜爱角度出发，我还是希望能卷一些。不过，目前来看真的卷不动，现在全世界范围内 Rust 的需求都大于供给，特别是优秀的 Rust 程序员更是难寻。

与 Go 语言相比，成为一名优秀的 Rust 程序员所需的门槛高得多，例如融汇贯通 Rust 语言各种中高级特性、闭着眼睛趟过各种坑、不用回忆无需查找就能立刻写出最合适的包/模块/方法、性能/安全/工程性的权衡选择信手拈来、深层性能优化易如反掌、异步编程小菜一碟，更别说 Rust 之外的操作系统、网络、算法等等相关知识。

所以，Rust 可以缓解内卷，而不是增加内卷。可以说是程序员的福音，不再是被随意替换的螺丝钉。

## 效率

下面从三个角度来谈谈 Rust 的效率：学习、运行、开发。

## 学习效率

众所周知，Rust 的学习曲线相当陡峭。起初，对此说法我还嗤之以鼻，随着不断的深入，现在也很认可这个说法。Rust 之难，不在于语言特性，这些都可以很容易学到，而在于：

- 实践中如何融会贯通的运用
- 遇到了坑时（生命周期、借用错误，自引用等）如何迅速、正确的解决
- 大量的标准库方法记忆及熟练使用，这些是保证开发效率的关键
- 心智负担较重，特别是初中级阶段

好在这本书就是干这个的，包君满意，不满意退... 要不还是骂我吧 :D

## 运行效率

得益于各种零开销抽象、深入到底层的优化潜力、优质的标准库和第三方库实现，Rust 具备非常优秀的性能，和 C、C++ 是一个级别。

同时 Rust 有一个极大的优点：只要按照正确的方式使用 Rust，无需性能优化，就能有非常优秀的表现，不可谓不惊艳。

现在有不少用 Rust 重写的工具、平台都超过了原来用 C、C++ 实现的版本，将老前辈拍死在沙滩上，俨然成为一种潮流～～

## 开发效率

Rust 的开发效率可以用先抑后扬来形容。在最初上手写项目时，你的开发速度将显著慢于 Go、Java 等语言，不过，一旦开始熟悉标准库、熟悉生命周期和所有权的常用解决方法，开发效率将大幅提升，甚至当形成肌肉记忆后，开发效率将不会慢于这些语言，而且原生就能写出高质量、安全、高效的代码，可以说中高级 Rust 程序员就是高效程序员的代名词。

## 个人的好处

学习 Rust 对个人也有极大的好处。

## 成为更好的程序员

要学好 Rust，你需要深入理解内存、堆栈、引用、变量作用域等这些其它高级语言往往不会深入接触的内容。另外，Rust 会通过语法、编译器和 clippy 这些静态检查工具半帮助半强迫的让你成为更优秀的程序员，写出更好的代码。

同时，当你掌握 Rust 后，就会自发性的想要去做一些更偏底层的开发，这些都可以帮助你更加了解操作系统、网络、性能优化等底层知识，也会间接或者直接的接触到各种算法和数据结构的实现。

慢慢的，你就在成为那个更好的程序员，也是更优秀的自己。

## 增加不可替代性

语言难学也有好处，一旦掌握，你将具备较强的不可替代性，不再是一个简单的工具人角色。看看现在内卷严重的 Java，工具人有多少！一个人离职，另外一个人很快就能替补上。

当然，我不是说 Rust 会给公司带来这种隐形的维护成本，毕竟这其实是一种双赢，公司收获了更优秀的程序员（不可否认的是 Rust 程序员普遍来说水平确实更高，毕竟都是有很好的编程基础、也很有追求的自驱型人才），而你也收获了更稳定的工作环境，甚至是更高的收入。

## 团队的好处

先不说安全、可靠性等对公司团队非常友好的特性，就说 Rust 程序只要能跑起来，那代码质量其实就是相当不错的，因为 Rust 编译器实在是一名严师厉友，甚至有些鸡毛。

正因为这较高的质量下限，我们在代码 review 时不用过于担心潜在的各种坑，得益于此，可以实现更加高效的开发、review、merge 流程。

由于 Rust 语言拥有异常强大的编译器和语言特性，Rust 的代码天然就会比其它语言拥有更少的 Bug。同时 Rust 拥有非常完善的工具链、最好的包管理工具，决定了 Rust 非常适合大型团队的协作开发。

也许 Rust 在开发速度上不是最快的，但是从开发 + 维护的角度来看，这个成本在各个语言中绝对是很小的。当然，如果你的公司就追求做出来能用就行，那 Rust 确实有些灰姑娘的感觉。

还有一点很重要，现在的 Rust 程序员往往拥有更出众的能力和学习自驱性，因此团队招到的人天然就保持了较高的底线。如果你有幸招到一个优秀的 Rust 程序员，那真是捡到宝了，他也会同时带动周围的人一起慢慢优秀（优秀的 Rust 程序员比较好辨别，门槛低的语言就并没有那么好辨别）。总之，一个这样的程序员会给团队带来远超他薪资的潜在回报和长远收益。

## 开源

目前 Rust 的主战场是在开源上，Go 的成功也证明了农村包围城市( 开源包围商业 )的可行性。

- UI 层开发，Rust 的 WASM 发展的如火如荼，隐隐有王者风范，在 JS 的基础设施领域，Rust 也是如鱼得水，例如 `swc`、`deno` 等。同时 `nextjs` 也是押宝 Rust，可以说 Rust 在前端的成功完全是无心插柳柳成荫。
- 基础设施层、数据库、搜索引擎、网络设施、云原生等都在出现 Rust 的身影，而且还不少。

- 系统开发，目前 Linux 已经将 Rust 语言纳入内核，是继 C 语言后第二门支持内核开发的语言，不过刚开始将主要支持驱动开发。
- 系统工具，现在最流行的就是用 Rust 重写之前 C、C++ 写的一票系统工具，还都获得了挺高的关注和很好的效果，例如 sd, exa, ripgrep, fd, bat 等。
- 操作系统，正在使用 Rust 开发的操作系统有好几个，其中最有名的可能就是谷歌的 Fuchsia，Rust 在其中扮演非常重要的角色。
- 区块链，如果 Rust 的份额说第二，应该没人敢说自己是第一吧？

类似的还有很多，我们就不一一列举。总之，现在有大量的项目正在被 Rust 重写，同时还有海量的项目在等待被重写，这些都是赚取github 星星和认可的好机会。在其它语言杀成一片红海时，Rust 还留了一大片蓝海等待大家的探索！

## 相比其他语言 Rust 的优势

由于篇幅有限，我们这里不会讲述详细的对比，就是简单介绍下 Rust 的优势，并不是说 Rust 优于这些语言，大家轻喷：）

### Go

Rust 语言表达能力更强，性能更高。同时线程安全方面 Rust 也更强，不容易写出错误的代码。包管理 Rust 也更好，Go 虽然在 1.10 版本后提供了包管理，但是目前还比不上 Rust 。

### C++

Rust 与 C++ 的性能旗鼓相当，但是在安全性方面 Rust 会更优，特别是使用第三方库时，Rust 的严格要求会让三方库的质量明显高很多。

语言本身的学习，Rust 的前中期学习曲线会更陡峭，但是在实际的项目开发过程中，C++ 会更难，代码也更难以维护。

### Java

除了极少数纯粹的数字计算性能，Rust 的性能全面领先于 Java 。同时 Rust 占用内存小的多，因此实现同等规模的服务，Rust 所需的硬件成本会显著降低。

### Python

性能自然是 Rust 完胜，同时 Rust 对运行环境要求较低，这两点差不多就足够抉择了。不过 Python 和 Rust 的彼此适用面其实也不太冲突。

## 使用现状

- AWS 从 2017 年开始就用 Rust 实现了无服务器计算平台：AWS Lambda 和 AWS Fargate，并且用 Rust 重写了 Bottlerocket OS 和 AWS Nitro 系统，这两个是弹性计算云 (EC2) 的重要服务
- Cloudflare 是 Rust 的重度用户，DNS、无服务计算、网络包监控等基础设施都与 Rust 密不可分
- Dropbox 的底层存储服务完全由 Rust 重写，达到了数万 PB 的规模
- Google 除了在安卓系统的部分模块中使用 Rust 外，还在它最新的操作系统 Fuchsia 中重度使用 Rust
- Facebook 使用 Rust 来增强自己的网页端、移动端和 API 服务的性能，同时还写了 Hack 编程语言的虚拟机
- Microsoft 使用 Rust 为 Azure 平台提供一些组件，其中包括 IoT 的核心服务
- GitHub 和 npmjs.com，使用 Rust 提供高达每天 13 亿次的 npm 包下载
- Rust 目前已经成为全世界区块链平台的首选开发语言
- TiDB，国内最有名的开源分布式数据库

尤其值得一提的是，AWS 实际上在押宝 Rust，内部对 Rust 的使用已经上升到头等公民 **first-class** 的地位。

## Rust 语言版本更新

与其它语言相比，Rust 的更新迭代较为频繁（得益于精心设计过的发布流程以及 Rust 语言开发者团队的严格管理）：

- 每 6 周发布一个迭代版本
- 2-3 年发布一个新的大版本，例如 Rust 2018 edition，Rust 2021 edition

好处在于，可以满足不同的用户群体的需求：

- 对于活跃的 Rust 用户，他们总是能很快获取到新的语言内容，毕竟，尝鲜是技术爱好者的共同特点：）
- 对于一般的用户，edition 大版本的发布会告诉他们：Rust 语言相比上次大版本发布，有了重大的改进，值得一看
- 对于 Rust 语言开发者，可以让他们的工作成果更快的被世人所知，不必锦衣夜行

## 总结

连续 6 年最受欢迎的语言当然不是浪得虚名。无 GC、效率高、工程性强、强安全性以及能同时得到工程派和学院派认可，这些令 Rust 拥有了自己的特色和生存空间。社区的友善，生态的快速发展，大公司的重仓跟进，一切的一切都在说明 Rust 的璀璨未来。

当然，语言毕竟只是工具，我们不能神话它，但是可以给它一个机会，也许，你最终能收获自己的真爱：）

相信大家听了这么多 Rust 的优点，已经迫不及待想要开始学习旅程，那么容我引用一句 CS (Counter-Strike: 反恐精英) 的经典台词：Ok, Let's Rust.

---

本书是完全开源的，但是并不意味着质量上的妥协，这里的每一个章节都花费了大量的心血和时间才能完成，为此牺牲了陪伴家人、日常娱乐的时间，虽然我们并不后悔，但是如果能得到读者您的鼓励，我们将感激不尽。

既然是开源，那最大的鼓励不是 money，而是 star:) **如果大家觉得这本书作者真的用心了，就帮我们点一个  吧，这将是我们继续前行的最大动力**

---

# 避免从入门到放弃

很多人都在学 Rust ing，也有很多人在放弃 ing。想要顺利学完 Rust，大家需要谨记本文列出的内容，否则这极有可能是又双叒叕从入门到放弃之旅。

Rust 是一门全新的语言，它会带给你前所未有的体验，提升你的通用编程水平，甚至于赋予你全新的编程思想。在此时此刻，大家可能还半信半疑，但是当学完它再回头看时，你肯定也会认同这些貌似浮夸的赞美。

## 避免试一试的心态

在学习 Go、Python 等编程语言时，你可能会一边工作、一边轻松愉快的学习它们，但是 Rust 不行。原因如文章开头所说，在学习 Rust 的同时你会收获很多语言之外的知识，因此 Rust 在入门阶段比很多编程语言要更难，但是一旦入门，你将收获一个全新的自己，成为一个更加优秀的程序员。

在学习过程中，一开始可能会轻松愉快，但是在开始接触 Rust 核心概念时(所有权、借用、生命周期、智能指针等)，难度会陡然提升，此时就需要认真对待起来，否则会为后面埋下难以填补的坑：结果最后你可能只有两个选择 - 重新学 or 放弃。

因此，在学习过程中，给大家三点建议：

- 要提前做好会遇到困难的准备，因为如上所说，学习 Rust 不仅仅是在学习一门编程语言
- 不要抱着试一试的心态去试一试，否则是浪费时间和消耗学习激情，作为连续七年荣获全世界最受欢迎桂冠的语言，Rust 不仅仅是值得试一试：)
- 深入学习一本好书或教程

总之，Rust 入门难，但是在你一次次克服艰难险阻的同时，也一次次收获了与众不同的编程经验，最后历经九九八十一难，立地成大佬。给自己一个机会，也给 Rust 一个机会：）

## 深入学习一本好书

Rust 跟其它语言不同，你无法看了一遍语法，然后就能上手写代码，对，我说的就是对比 Go 语言，后者的简单易用是有目共睹的。

这些年，我遇到过太多在网上看了一遍菜鸟教程(或其它简易教程)就上手写 demo 甚至项目的同学，无一例外，都各种碰壁、趟坑，最后要么放弃，要么回炉重造，之前的时间和精力基本等同浪费。

因此，大家一定要舍得投入时间，沉下心去读一本好书，这本书会带你深入浅出地学习使用 Rust 所需的各种知识，还会带你提前趟坑，这些坑往往是需要大量的时间才能领悟的。

在以前我可能会推荐看 Rust Book + async book + nomicon 这几本英文书的组合，但是现在有了一本更适合中国用户的书籍，那就是...你们猜，内容好坏大家一读即知，光就文字而言，那绝对是行云流水般的阅读体验，可以极大提升学习效率，也不再因为反复读也读不懂一句话而烦闷不堪。

## 千万别从链表或图开始练手

CS (Computer Science：计算机科学) 课程中咱们会学习大量的常用数据结构和算法，因此大家都养成了一种好习惯：学习一门新语言，先用它写个链表或图试试。

我的天，在 Rust 中**千万别这么干**，你是在扼杀自己之前的努力！因为不像其它语言，链表在 Rust 中简直是地狱一般的难度，我见过太多英雄好汉难过链表关，最终黯然退幕。我不希望正在阅读此文的你也成为其中一个：(

这些自引用类型（一种数据结构，它内部的某个字段又引用了其自身），它们堪称恶魔：不仅仅在蹂躏着新手，还在折磨着老手。有意思的是，它们的难恰恰是 Rust 的优点导致的：无 GC 也无手动内存管理还要做到内存安全。

这些优点并不是凭空产生，而是来源于 Rust 那一套强大、优美的机制，这些机制一旦你学到，就会被它巧妙的构思和设计征服，进而被 Rust 深深吸引！但是一切选择都有利弊，这种机制的弊端就在于实现链表这类数据结构时，会变得非常非常复杂。

你需要糅合各种知识，才能解决这个问题，但是这显然不是一个新手应该独自去面对的。总之，不会链表对于 Rust 的学习和写项目，真的没有任何影响，直接使用大神已经写好的数据结构就可以。

如果想要练手，我们可以换个方向开始，例如书中的入门和进阶实战项目都是非常好的选择。当然如果你就是喜欢征服困难，那没问题，就从链表开始。但是无论选择哪个，本书都将给你莫大的帮助，包括如何实现一个链表！

## 仔细阅读编译错误

在一些编程语言中，你可能习惯了编译器给出的错误只看前面（或后面）几行，毕竟大部分是没啥大用的堆栈信息，在此过程中，`how stupid the 编译器 is` 的感想时不时会迸发出来。

但是 Rust 不是，它为我们提供了一个强大无比的编译器，而且会提示我们该如何修改代码以解决错误，简直就是一名优秀的老师！

因此在使用 Rust 过程中，如果你不知该如何解决错误，不妨仔细阅读下编译器或者 IDE 给出的错误提示，绝大多数时候，都可以通过这些提示顺利的解决问题。

同时也不要忽略编译器给出的警告信息(warnings)，因为里面包含了 cargo clippy 给出的 lint 提示，这些提示不仅仅包含代码风格，甚至包含了一些隐藏很深的错误！至于这些错误为何不是 error 形式出现，随着学习的深入，你将逐渐理解 Rust 的各种设计选择，包括这个问题。

## 不要强制自己使用其它编程语言的最佳实践来写 Rust

大多数其它编程语言适用的最佳实践在 Rust 中也可以很好的使用，但是 Rust 并不是一门专门的面向对象或者函数式语言，因此在使用自己喜欢的编程风格时，也要考虑遵循 Rust 应有的实践。

例如纯面向对象或纯函数式编程，在 Rust 中就并不是一个很好的选择。如果你有过 Go 语言的编程经验，相信能更加理解我这里想表达的含义。

不过大家也不用担心，在书中我们会以专题的形式专门讲解 Rust 的最佳实践，看完后自然就明白了。

## 总结

对于新手而言，最应该避免的就是从**链表开始练手**，最应该做的就是认真仔细地学习一本优秀而全面的书。

总之，认真学 Rust，既然选择了，就**相信自己，你的前方会是星辰大海！**

# Rust语言中文网



跟我来吧，学完 Rust，你的前面就是星辰大海！

---

这个社区与其它 Rust 社区有点不一样：我们聚焦于 Rust 语言的学习研究和实战应用上，不搞花活！

- QQ交流群: 1009730433
- 公众号: Rust语言中文网



## Rusty Book( 锈书 )



在 Rust 元宇宙，最优秀的项目可以称之为 `rusty`，用咱中国话来说，就是够锈( 秀 )。

如果你有以下需求，可以来看看锈书，它绝对不会让你失望：

- 想要知道现在优秀的、关注度高的 Rust 项目有哪些
- 发现一些好玩、有趣、酷炫的开源库
- 需要寻找某个类型的库，例如，一个 HTTP 客户端或 ProtoBuffer 编码库，要求是好用、更新活跃、高质量
- 想要寻找常用操作的代码片段，用于熟悉 Rust 或者直接复制粘贴到自己的项目中，例如文件操作、数据库操作、HTTP 请求、排序算法、正则等

在线阅读锈书：[Github地址](#)

# Datav: 可编程的数据可视化平台和可观测性平台

经常关注新技术的同学，这两年应该都听说过可观测性这个概念，它包含了对 **Metrics 指标、Log 日志以及 Trace 链路的监控**，特别是针对这三个指标的**深度关联**，让监控数据不再孤立。

目前整个业界百花齐放，Prometheus、Jaeger、Opentelemetry 各种优秀的世界级开源产品覆盖了可观测性的方方面面，但是在可观测性的可视化方面还缺少一个大杀器，我们期待它能拥有以下特性：

- 覆盖各种可观测性场景
- 拥有丰富可定制选项的图表组件和数据源
- 图表、页面之间的深度可定义交互
- 企业级的多租户、权限管理、导航菜单和全局状态管理
- 优秀的图表性能和交互体验
- 现代化的 UI 设计，支持数据大屏，完美支持移动端
- 强大的可编程性及二次开发友好性
- 宽松的开源协议、丰富的文档和快速的社区响应支持

这些特性每一个都不简单， Datav 就是为了解决这些问题而生。

---

Datav 不是全世界最好的可观测性和数据可视化平台，这毫无疑问，至少目前不是。但是它对开发者最友好，并且支持丰富的可编程性的平台

当一个产品它的代码结构清晰简洁、架构干净、使用的技术符合时代主流、文档齐全，当一个产品在各种使用细节上都为开发者专门设计过，当一个产品允许各种深度可定制时，我们可以称其开发者友好

---

## Datav

Datav 是一个专为开发者打造的、可编程的可观测性平台，同时它还是 Grafana 的数据可视化平替。可以帮助用户快速在线构建监控、日志和链路跟踪等可观测性场景，可以不夸张的说，你想要的几乎任何监控可视化场景，Datav 都能帮你实现（如果不行，大家可以提 issue，合理的需求秒支持）。

- 开源地址：<https://github.com/data-observe/datav>
- 官方网站：<https://datav.io>
- 在线 Demo: <https://play.datav.io>

## 开发语言

Datav 目前由以下部分组成:

- **UI:** React + Typescript + Vitejs
- **API、数据处理和 UI 静态文件服务:** Go
- **插件管理:** Go
- **数据采集 Agent:** Rust

## 写在最后的感悟

Rust course 开源迄今快 2 年了，我没有收过一分钱的赞赏，曾经有过这个想法，但最终还是被自己否定了。不是因为有多高尚，只是真心希望能为国内的开源做一些微不足道的、不那么金钱相关的贡献。

回头看，现在能获得这么多 Star，已经远远超出了我当初的预期。说句真心话，这本书的质量配不上这么多 star，毕竟有那么多非常优秀的开源产品都没能获得配得上它们质量的认可度和曝光度。

但可能因为天时地利人和，Rust course 走到了今天这一步，无论如何，非常感谢每一个贡献者，也感谢每一个点了 star 支持本书的读者，你们是最棒的！

正因为对开源的坚持，尽管是拥有近百万行代码的复杂平台，Datav 依然选择了 Apache2.0 作为开源协议，我不希望大家在使用它时，还担心未来商业化潜在的风险。

总之，开源这条路我会坚定走下去，为国为民的大话不敢说，但至少我会尽自己的一份力量，为国内的开源做一些力所能及的贡献。

# 寻找牛刀，以便小试

其实对于写这种章节，我内心是拒绝的，因为真的很无趣。对于一本书而言，这也更像是一种浪费纸张的行为（好在咱无纸化：-D）。不过没有办法，如果不安装 Rust 环境，总不能让大家用空气运行吧，so，我恶趣味的起了一个这样的章节名。

在本章中，你将学习以下内容：

1. 在 macOS、Linux、Windows 上安装 Rust 以及相关工具链
2. 搭建 VSCode 所需的环境
3. 简单介绍 Cargo
4. 实现一个酷炫多国语言版本的“世界，你好”的程序，并且谈谈对 Rust 语言的初印象

# 安装 Rust

`rustup` 是 Rust 的安装程序，也是它的版本管理程序。强烈建议使用 `rustup` 来安装 Rust，当然如果你有异心，请寻找其它安装方式，然后再从下一节开始阅读。

haha，开个玩笑。读者乃大大，怎么能弃之不顾。

注意：如果你不想用或者不能用 `rustup`，请参见 [Rust 其它安装方法](#)。

至于版本，现在 Rust 稳定版特性越来越全了，因此下载最新稳定版本即可。由于你用的 Rust 版本可能跟本书写作时不一样，一些编译错误和警告可能也会有所不同。

## 在 Linux 或 macOS 上安装 `rustup`

打开终端并输入下面命令：

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

这个命令将下载一个脚本并开始安装 `rustup` 工具，此工具将安装 Rust 的最新稳定版本。可能会提示你输入管理员密码。

如果安装成功，将出现下面这行：

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

OK，这样就已经完成 Rust 安装啦。

## 安装 C 语言编译器：（非必需）

Rust 对运行环境的依赖和 Go 语言很像，几乎所有环境都可以无需安装任何依赖直接运行。但是，Rust 会依赖 `libc` 和链接器 `linker`。所以如果遇到了提示链接器无法执行的错误，你需要再手动安装一个 C 语言编译器：

**macOS 下：**

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

**Linux 下：**

Linux 用户一般应按照相应发行版的文档来安装 `GCC` 或 `Clang`。

例如，如果你使用 Ubuntu，则可安装 `build-essential`。

## 在 Windows 上安装 rustup

Windows 上安装 Rust 需要有 C++ 环境，以下为安装的两种方式：

### 1. x86\_64-pc-windows-msvc (官方推荐)

先安装 Microsoft C++ Build Tools，勾选安装 C++ 环境即可。安装时可自行修改缓存路径与安装路径，避免占用过多 C 盘空间。安装完成后，Rust 所需的 msvc 命令行程序需要手动添加到环境变量中，否则安装 Rust 时 `rustup-init` 会提示未安装 Microsoft C++ Build Tools，其位于：`%Visual Studio 安装位置%\VC\Tools\MSVC\%version%\bin\Hostx64\x64`（请自行替换其中的 %Visual Studio 安装位置%、%version% 字段）下。

如果你不想这么做，可以选择安装 Microsoft C++ Build Tools 新增的“定制”终端 `Developer Command Prompt for %Visual Studio version%` 或 `Developer PowerShell for %Visual Studio version%`，在其中运行 `rustup-init.exe`。

准备好 C++ 环境后开始安装 Rust：

在 [RUSTUP-INIT](#) 下载系统相对应的 Rust 安装程序，一路默认即可。

```
PS C:\Users\Hehongyuan> rustup-init.exe
.....
Current installation options:

    default host triple: x86_64-pc-windows-msvc
    default toolchain: stable (default)
        profile: default
    modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
```

### 2. x86\_64-pc-windows-gnu

相比于 MSVC 版本来说，GNU 版本具有更轻量，更靠近 Linux 的优势。

首先，根据 [MSYS2 官网](#) 配置 MSYS。

若您觉得下载太慢，可以试试由 Caviar-X 提供的 [代理](#)。

在安装 mingw-toolchain 后，请将 %MSYS 安装路径%\mingw64\bin 添加到系统变量 PATH 中。

配置好后，在 MSYS 中输入下面的命令来安装 rustup。

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

之后，根据以下输出进行配置。

Current installation options:

```
default host triple: x86_64-pc-windows-msvc
default toolchain: stable (default)
    profile: default
modify PATH variable: yes
```

- 1) Proceed with installation (default)
  - 2) Customize installation
  - 3) Cancel installation
- >2

I'm going to ask you the value of each of these installation options.  
You may simply press the Enter key to leave unchanged.

```
Default host triple? [x86_64-pc-windows-msvc]
x86_64-pc-windows-gnu
```

```
Default toolchain? (stable/beta/nightly/none) [stable]
stable
```

```
Profile (which tools and data to install)? (minimal/default/complete) [default]
complete
```

```
Modify PATH variable? (Y/n)
Y
```

Current installation options:

```
default host triple: x86_64-pc-windows-gnu
default toolchain: stable
    profile: complete
modify PATH variable: yes
```

- 1) Proceed with installation (default)
  - 2) Customize installation
  - 3) Cancel installation
- >

再之后，按下 1，等待。完成后，您就已经安装了 Rust 和 rustup。

## 更新

要更新 Rust，在终端执行以下命令即可更新：

```
$ rustup update
```

## 卸载

要卸载 Rust 和 rustup，在终端执行以下命令即可卸载：

```
$ rustup self uninstall
```

## 检查安装是否成功

检查是否正确安装了 Rust，可打开终端并输入下面这行，此时能看到最新发布的稳定版本的版本号、提交哈希值和提交日期：

```
$ rustc -V  
rustc 1.56.1 (59eed8a2a 2021-11-01)  
  
$ cargo -V  
cargo 1.57.0 (b2e52d7ca 2021-10-21)
```

---

注：若发现版本号不同，以您的版本号为准

---

恭喜，你已成功安装 Rust！

如果没看到此信息：

1. 如果你使用的是 Windows，请检查 Rust 或 %USERPROFILE%\cargo\bin 是否在 %PATH% 系统变量中。
2. 如果你使用的是 Windows 下的 Linux 子系统，请关闭并重新打开终端，再次执行以上命令。

如果都正确，但 Rust 仍然无法正常工作，那么你可以在很多地方获得帮助。最简单的是加入 **Rust 编程学院这个大家庭，QQ 群：1009730433**。

## 本地文档

安装 Rust 的同时也会在本地安装一个文档服务，方便我们离线阅读：运行 `rustup doc` 让浏览器打开本地文档。

每当遇到标准库提供的类型或函数不知道怎么用时，都可以在 API 文档中查找到！具体参见 [在标准库寻找你想要的内容](#)。

## 墙推 VSCode!

VSCode 从 15 年刚开始推出，我就在使用了。做为第一个吃螃蟹的人，可以说见证了它一路的快速发展，直到现在它已经成为开源世界最火的 IDE 之一（弱弱的说一句，之一也许可以去掉）。

顺便歪楼说一句：我预言过三件事：

1. 在 13 年预言 Golang 会火遍全世界。同时创建了 14-19 年最火的 Golang 隐修会社区，可惜因为某些原因被封停了，甚是遗憾。
2. 在 15 年预言 VSCode 会成为世界上最好的 IDE；同时我还是 jaeger tracing 项目的第一 star 用户（是的，比作者还早），当时就很看好这个项目的后续发展。
3. 现在呢，我在这里正式预言：**未来 Rust 会成为主流编程语言之一，在几乎所有开发领域都将大放光彩**。总之牛逼已吹下，希望不要被打脸。：（

下面继续简单介绍下 VSCode，以下内容引用于官网：

---

Visual Studio Code(VSCode) 是微软 2015 年推出的一个轻量但功能强大的源代码编辑器，基于 Electron 开发，支持 Windows、Linux 和 macOS 操作系统。它内置了对 JavaScript, TypeScript 和 Node.js 的支持并且具有丰富的其它语言和扩展的支持，功能超级强大。Visual Studio Code 是一款免费开源的现代化轻量级代码编辑器，支持几乎所有主流的开发语言的语法高亮、智能代码补全、自定义快捷键、括号匹配和颜色区分、代码片段、代码对比 Diff、GIT 命令等特性，支持插件扩展，并针对网页开发和云端应用开发做了优化。

---

## 安装 VSCode 的 Rust 插件

在 VSCode 的左侧扩展目录里，搜索 `rust`，你能看到两个 Rust 插件，如果没有意外，这两个应该分别排名第一和第二：

1. 官方的 `Rust`，作者是 `The Rust Programming Language`，官方出品，牛逼就完了，但是.....我们并不推荐（事实上已经不再维护了，官方收编了第二个插件，现在第二个插件的作者也是 `The Rust Programming Language`），这个插件有几个问题：
  - 首先是在代码跳转上支持的很烂，只能在自己的代码库中跳转，一旦跳到别的三方库，那就无法继续跳转，对于查看标准库和三方库的源码带来了极大的困扰
  - 其次，不支持类型自动标注，对于 Rust 语言而言，类型说明是非常重要的，特别是在你不知道给变量一个什么类型时，这种 IDE 的自动提示就变得弥足珍贵

- 代码提示不太好用，有些方法既不会提示，也不能跳转
2. 社区驱动的 `rust-analyzer`，非常推荐，上面说的所有问题，在这个插件上都得到了解决，不得不说，Rust 社区 yyds!

所以，综上所述，我们选择 `rust-analyzer` 作为 Rust 语言的插件，具体的安装很简单，点击插件，选择安装即可，根据提示可能需要重新加载 IDE。

---

在搜索 VSCode 插件时，报错：提取扩展出错，`XHR failed`，这个报错是因为网络原因导致，很可能你是你的网络不行或者翻墙工具阻拦你的访问，试着关掉翻墙，再进行尝试。

安装完成后，在第一次打开 Rust 项目时，需要安装一些依赖，具体的状态在左下角会进行提示，包括下载、代码构建、building 等。

当插件使用默认设置时，每一次保存代码，都会出进行一次重新编译。

---

如果你的电脑慢，有一点一定要注意：

在编译器构建代码的同时，不要在终端再运行 `cargo run` 等命令进行编译，不然会获得一个报错提示，大意是当前文件目录已经被锁定，等待其它使用者释放。如果等了很久 IDE 还是没有释放（虽然我没遇到过，但是存在这个可能性），你可以关掉 IDE，并手动 `kill` 掉 `rust-analyzer`，然后重新尝试。

---

## 安装其它好用的插件

在此，再推荐大家几个好用的插件：

1. Even Better TOML，支持 `.toml` 文件完整特性
2. Error Lens，更好的获得错误展示
3. One Dark Pro，非常好看的 VSCode 主题
4. CodeLLDB，Debugger 程序

好了，至此，VSCode 的配置就已经全部结束，是不是很简单？下面让我们来用 `Cargo` 创建一个 Rust 项目，然后用 VSCode 打开。

## 认识 Cargo

但凡经历过 C/C++ 或 Go 语言 1.10 版本之前的用户都知道，一个好的包管理工具有多么的重要！！我那个时候是如此的渴望类似 nodejs 的 npm 包管理工具，但是却求而不得。

包管理工具最重要的意义就是**任何用户拿到你的代码，都能运行起来**，而不会因为各种包版本依赖焦头烂额。

Go 语言在 1.10 版本之前，所有的包都是在 `github.com` 下存放，导致了所有的项目都公用一套依赖代码，在本地项目复杂后，这简直是一种灾难。

说多了都是泪，笔者目前还有一个早期 Go 的项目（15 年写的），用到了 `iris`（一个坑爹 HTTP 服务），结果现在运行不起来了，因为找不到 `iris` 当时的那个版本了！！

作为一门现代化语言，Rust 吸收了多个语言的包管理优点，为大家提供超级大杀器：`cargo`，真的，再挑剔的开发者，都对它赞不绝口。👍

总而言之，`cargo` 提供了一系列的工具，从项目的建立、构建到测试、运行直至部署，为 Rust 项目的管理提供尽可能完整的手段。同时，与 Rust 语言及其编译器 `rustc` 紧密结合，可以说用了后就忘不掉，如同初恋般的感觉。

## 创建一个"你好，世界"项目

又见"你好，世界"，肯定有读者在批评了：你就不能有点创意吗？"世界，你好"难道不配？你是读者，你说了算，那我们就来创建一个"世界，你好"。

上文提到，Rust 语言的包管理工具是 `cargo`。不过，我们无需再手动安装，之前安装 Rust 的时候，就已经一并安装了。

终于到了紧张刺激的 new new new 环节：

```
$ cargo new world_hello  
$ cd world_hello
```

上面的命令使用 `cargo new` 创建一个项目，项目名是 `world_hello`（向读者势力低头的项目名称，泪奔），该项目的结构和配置文件都是由 `cargo` 生成，意味着**我们的项目被 cargo 所管理**。

---

如果你在终端无法使用这个命令，考虑一下 环境变量 是否正确的设置：把 cargo 可执行文件所在的目录添加到环境变量中。

如果是在 Windows 的 WSL2 子系统下，出现以下错误：

```
error: command failed: 'rustc' error: caused by: Permission denied (os error 13)
```

可尝试先卸载，再使用 sudo 命令进行安装：`$ sudo curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -s$Sf | sh`

---

早期的 cargo 在创建项目时，必须添加 `--bin` 的参数，如下所示：

```
$ cargo new world_hello --bin
$ cd world_hello
```

现在的版本，已经无需此参数，cargo 默认就创建 bin 类型的项目，顺便说一句，Rust 项目主要分为两个类型：bin 和 lib，前者是一个可运行的项目，后者是一个依赖库项目。

下面来看看创建的项目结构：

```
$ tree
.
├── .git
├── .gitignore
└── Cargo.toml
src
└── main.rs
```

是的，连 git 都给你创建了，不禁令人感叹，不是女儿，胜似女儿，比小棉袄还体贴。

## 运行项目

有两种方式可以运行项目：

1. `cargo run`
2. 手动编译和运行项目

首先来看看第一种方式，一码胜似千言，在之前创建的 `world_hello` 目录下运行：

```
$ cargo run
Compiling world_hello v0.1.0 (/Users/sunfei/development/rust/world_hello)
  Finished dev [unoptimized + debuginfo] target(s) in 0.43s
    Running `target/debug/world_hello`
Hello, world!
```

好了，你已经看到程序的输出：“Hello, world”。

如果你安装的 Rust 的 host triple 是 x86\_64-pc-windows-msvc 并确认 Rust 已经正确安装，但在终端上运行上述命令时，出现类似如下的错误摘要 linking with `link.exe` failed: exit code: 1181，请使用 Visual Studio Installer 安装 Windows SDK。

可能有读者不愿意了，说好了“世界，你好”呢？别急，在下一节，我们再对代码进行修改。（认真想来，“你好，世界”强调的是我对世界说你好，而“世界，你好”是世界对我说你好，明显是后者更有包容性和国际范儿，读者真·好眼光。）

上述代码，cargo run 首先对项目进行编译，然后再运行，因此它实际上等同于运行了两个指令，下面我们手动试一下编译和运行项目：

编译

```
$ cargo build
  Finished dev [unoptimized + debuginfo] target(s) in 0.00s
```

运行

```
$ ./target/debug/world_hello
Hello, world!
```

行云流水，但谈不上一气呵成。细心的读者可能已经发现，在调用的时候，路径 ./target/debug/world\_hello 中有一个明晃晃的 debug 字段，没错我们运行的是 debug 模式，在这种模式下，**代码的编译速度会非常快**，可是福兮祸所伏，**运行速度就慢了**。原因是，在 debug 模式下，Rust 编译器不会做任何的优化，只为了尽快的编译完成，让你的开发流程更加顺畅。

作为尊贵的读者，咱自然可以要求更多，比如你想要高性能的代码怎么办？简单，添加 --release 来编译：

- cargo run --release
- cargo build --release

试着运行一下我们高性能的 release 程序：

```
$ ./target/release/world_hello  
Hello, world!
```

## cargo check

当项目大了后，`cargo run` 和 `cargo build` 不可避免的会变慢，那么有没有更快的方式来验证代码的正确性呢？大杀器来了，接着！

`cargo check` 是我们在代码开发过程中最常用的命令，它的作用很简单：快速的检查一下代码能否编译通过。因此该命令速度会非常快，能节省大量的编译时间。

```
$ cargo check  
Checking world_hello v0.1.0 (/Users/sunfei/development/rust/world_hello)  
Finished dev [unoptimized + debuginfo] target(s) in 0.06s
```

---

Rust 虽然编译速度还行，但是还是不能与 Go 语言相提并论，因为 Rust 需要做很多复杂的编译优化和语言特性解析，甚至连如何优化编译速度都成了一门学问：[优化编译速度](#)。

## Cargo.toml 和 Cargo.lock

`Cargo.toml` 和 `Cargo.lock` 是 `cargo` 的核心文件，它的所有活动均基于此二者。

- `Cargo.toml` 是 `cargo` 特有的**项目数据描述文件**。它存储了项目的所有元配置信息，如果 Rust 开发者希望 Rust 项目能够按照期望的方式进行构建、测试和运行，那么，必须按照合理的方式构建 `Cargo.toml`。
- `Cargo.lock` 文件是 `cargo` 工具根据同一项目的 `toml` 文件生成的**项目依赖详细清单**，因此我们一般不用修改它，只需要对着 `Cargo.toml` 文件撸就行了。

---

什么情况下该把 `Cargo.lock` 上传到 git 仓库里？很简单，当你的项目是一个可运行的程序时，就上传 `Cargo.lock`，如果是一个依赖库项目，那么请把它添加到 `.gitignore` 中。

---

现在用 VSCode 打开上面创建的"世界，你好"项目，然后进入根目录的 `Cargo.toml` 文件，可以看到该文件包含不少信息：

## package 配置段落

package 中记录了项目的描述信息，典型的如下：

```
[package]
name = "world_hello"
version = "0.1.0"
edition = "2021"
```

`name` 字段定义了项目名称，`version` 字段定义当前版本，新项目默认是 `0.1.0`，`edition` 字段定义了我们使用的 Rust 大版本。因为本书很新（不仅仅是现在新，未来也将及时修订，跟得上 Rust 的小步伐），所以使用的是 Rust `edition 2021` 大版本，详情见 [Rust 版本详解](#)

## 定义项目依赖

使用 `cargo` 工具的最大优势就在于，能够对该项目的各种依赖项进行方便、统一和灵活的管理。

在 `Cargo.toml` 中，主要通过各种依赖段落来描述该项目的各种依赖项：

- 基于 Rust 官方仓库 `crates.io`，通过版本说明来描述
- 基于项目源代码的 git 仓库地址，通过 URL 来描述
- 基于本地项目的绝对路径或者相对路径，通过类 Unix 模式的路径来描述

这三种形式具体写法如下：

```
[dependencies]
rand = "0.3"
hammer = { version = "0.5.0" }
color = { git = "https://github.com/bjz/color-rs" }
geometry = { path = "crates/geometry" }
```

相信聪明的读者已经能看懂该如何引入外部依赖库，这里就不再赘述。详细的说明参见此章：[Cargo 依赖管理](#)，但是不建议大家现在去看，只要按照目录浏览，拨云见日指日可待。

## 基于 cargo 的项目组织结构

前文有提到 `cargo` 默认生成的项目结构，真实的项目肯定会有所不同，但是在目前的学习阶段，还无需关注。感兴趣的同学可以移步：[Cargo 项目结构](#)

至此，大家对 Rust 项目的创建和管理已经有了初步的了解，那么来完善刚才的“世界，你好”项目吧。

# 不仅仅是 Hello world

几乎所有教程中安装的最后一个环节都是 `hello world`，我们也不能免俗。但是，在 `hello world` 之后，还有一个相亲，啊呸，Rust 初印象环节，希望大家喜欢。

## 多国语言的"世界，你好"

还记得大明湖畔等你的 VSCode IDE 和通过 Cargo 创建的 [世界，你好](#) 工程吗？

现在使用 VSCode 打开 [上一节](#) 中创建的 `world_hello` 工程，然后进入 `main.rs` 文件。（此文件是当前 Rust 工程的入口文件，和其它语言几无区别。）

接下来，对世界友人给予热切的问候：

```
fn greet_world() {
    let southern_germany = "Grüß Gott!";
    let chinese = "世界，你好";
    let english = "World, hello";
    let regions = [southern_germany, chinese, english];
    for region in regions.iter() {
        println!("{}", &region);
    }
}

fn main() {
    greet_world();
}
```

打开终端，进入 `world_hello` 工程根目录，运行该程序。（你也可以在 VSCode 中打开终端，方法是点击 VSCode 上方菜单栏中的终端->新建终端，或者直接使用快捷键打开。）

```
$ cargo run
Compiling world_hello v0.1.0 (/Users/sunfei/development/rust/world_hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.21s
Running `target/debug/world_hello`
Grüß Gott!
世界，你好
World, hello
```

你的热情，就像一把火，燃烧了整个世界~ 花点时间来看看上面的代码：

首先，Rust 原生支持 UTF-8 编码的字符串，这意味着你可以很容易的使用世界各国文字作为字符串内容。

其次，关注下 `println` 后面的 `!`，如果你有 Ruby 编程经验，那么你可能会认为这是解构操作符，但是在 Rust 中，这是 宏 操作符，你目前可以认为宏是一种特殊类型函数。

对于 `println` 来说，我们没有使用其它语言惯用的 `%s`、`%d` 来做输出占位符，而是使用 `{}`，因为 Rust 在底层帮我们做了大量工作，会自动识别输出数据的类型，例如当前例子，会识别为 `String` 类型。

最后，和其它语言不同，Rust 的集合类型不能直接进行循环，需要变成迭代器（这里是通过 `.iter()` 方法），才能用于迭代循环。在目前来看，你会觉得这一点好像挺麻烦，不急，以后就知道这么做的好处所在。

---

实际上这段代码可以简写，在 2021 edition 及以后，支持直接写 `for region in regions`，原因会在迭代器章节的开头提到，是因为 for 隐式地将 `regions` 转换成迭代器。

---

至于函数声明、调用、数组的使用，和其它语言没什么区别，So Easy！

## Rust 语言初印象

Rust 这门语言对于 Haskell 和 Java 开发者来说，可能会觉得很熟悉，因为它们在高阶表达方面都很优秀。简而言之，就是可以很简洁的写出原本需要一大堆代码才能表达的含义。但是，Rust 又有所不同：它的性能是底层语言级别的性能，可以跟 C/C++ 相媲美。

上面的 So Easy 的余音仍在绕梁，我希望它能继续下去，可是... 人总是要面对现实，因此让我们来点狠活：

```

fn main() {
    let penguin_data = "\
common name,length (cm)
Little penguin,33
Yellow-eyed penguin,65
Fiordland penguin,60
Invalid,data
";

    let records = penguin_data.lines();

    for (i, record) in records.enumerate() {
        if i == 0 || record.trim().len() == 0 {
            continue;
        }

        // 声明一个 fields 变量，类型是 Vec
        // Vec 是 vector 的缩写，是一个可伸缩的集合类型，可以认为是一个动态数组
        // <_>表示 Vec 中的元素类型由编译器自行推断，在很多场景下，都会帮我们省却不少功夫
        let fields: Vec<_> = record
            .split(',')
            .map(|field| field.trim())
            .collect();
        if cfg!(debug_assertions) {
            // 输出到标准错误输出
            eprintln!("debug: {:?} -> {:?}", record, fields);
        }
    }

    let name = fields[0];
    // 1. 尝试把 fields[1] 的值转换为 f32 类型的浮点数，如果成功，则把 f32 值赋给 length 变量
    //
    // 2. if let 是一个匹配表达式，用来从=右边的结果中，匹配出 length 的值：
    //   1) 当=右边的表达式执行成功，则会返回一个 Ok(f32) 的类型，若失败，则会返回一个 Err(e)
    //      类型，if let 的作用就是仅匹配 Ok 也就是成功的情况，如果是错误，就直接忽略
    //   2) 同时 if let 还会做一次解构匹配，通过 Ok(length) 去匹配右边的 Ok(f32)，最终把相应的 f32 值赋给 length
    //
    // 3. 当然你也可以忽略成功的情况，用 if let Err(e) = fields[1].parse::<f32>() {...} 匹配出错误，然后打印出来，但是没啥卵用
    if let Ok(length) = fields[1].parse::<f32>() {
        // 输出到标准输出
        println!("{} , {}cm", name, length);
    }
}
}

```

看完这段代码，不知道你的余音有没有戛然而止，反正我已经在颤抖了。这就是传说中的下马威吗？ 😱

上面代码中，值得注意的 Rust 特性有：

- 控制流：`for` 和 `continue` 连在一起使用，实现循环控制。
- 方法语法：由于 Rust 没有继承，因此 Rust 不是传统意义上的面向对象语言，但是它却从 oo 语言那里偷师了方法的使用 `record.trim()`, `record.split(',')` 等。
- 高阶函数编程：函数可以作为参数也能作为返回值，例如 `.map(|field| field.trim())`，这里 `map` 方法中使用闭包函数作为参数，也可以称呼为 匿名函数、`lambda` 函数。
- 类型标注：`if let Ok(length) = fields[1].parse::<f32>()`，通过 `::<f32>` 的使用，告诉编译器 `length` 是一个 `f32` 类型的浮点数。这种类型标注不是很常用，但是在编译器无法推断出你的数据类型时，就很有用了。
- 条件编译：`if cfg!(debug_assertions)`，说明紧跟其后的输出（打印）只在 `debug` 模式下生效。
- 隐式返回：Rust 提供了 `return` 关键字用于函数返回，但是在很多时候，我们可以省略它。因为 Rust 是 **基于表达式的语言**。

在终端中运行上述代码时，会看到很多 `debug: ...` 的输出，上面有讲，这些都是 条件编译 的输出，那么该怎么消除掉这些输出呢？

读者大大普遍冰雪聪明，肯定已经想到：是的，在 [认识 Cargo](#) 中，曾经介绍过 `--release` 参数，因为 `cargo run` 默认是运行 `debug` 模式。因此想要消灭那些 `debug:` 输出，需要更改为其它模式，其中最常用的模式就是 `--release` 也就是生产发布的模式。

具体运行代码就不给了，留给大家作为一个小练习，建议亲自动手尝试下。

至此，Rust 安装入门就已经结束。相信看到这里，你已经发现了本书与其它书的区别，其中最大的区别就是：**这本书就像优秀的国外课本一样，不太枯燥。也希望这本不太枯燥的书，能伴你长行，犹如一杯奶茶，细细品之，唇齿留香。**

# 下载依赖很慢或卡住?

在目前，大家还不需要自己搭建的镜像下载服务，因此只需知道下载依赖库的地址是 [crates.io](https://crates.io)，是由 Rust 官方搭建的镜像下载和管理服务。

但悲剧的是，它的默认镜像地址是在国外，这就导致了某些时候难免会遇到下载缓慢或者卡住的情况，下面我们一起来看看。

## 下载很慢?

作为国外的语言，下载慢是正常的，隔壁的那位还被墙呢:)

解决下载缓慢有两种途径：

### 开启命令行或者全局翻墙

经常有同学反馈，我明明开启翻墙了，但是下载依然还是很慢，无论是命令行中下载还是 VSCode 的 rust-analyzer 插件自动拉取。

事实上，翻墙工具默认开启的仅仅是浏览器的翻墙代理，对于命令行或者软件中的访问，并不会代理流量，因此这些访问还是通过正常网络进行的，自然会失败。

因此，大家需要做的是在你使用的翻墙工具中 复制终端代理命令 或者开启全局翻墙。由于每个翻墙软件的使用方式不同，因此具体的还是需要自己研究下。以我使用的 ClashX 为例，点击 复制终端代理命令后，会自动复制一些 export 文本，将这些文本复制到命令行终端中，执行一下，就可以自动完成代理了。

```
export https_proxy=http://127.0.0.1:7890 http_proxy=http://127.0.0.1:7890  
all_proxy=socks5://127.0.0.1:7891
```

### 修改 Rust 的下载镜像为国内的镜像地址

这个效果最直接，一劳永逸，但是就是配置起来略微麻烦。

为了使用 crates.io 之外的注册服务，我们需要对 \$HOME/.cargo/config.toml (\$CARGO\_HOME 下) 文件进行配置，添加新的服务提供商，有两种方式可以实现：增加新的镜像地址和覆盖默认的镜像地址。

## 新增镜像地址

首先是在 `crates.io` 之外添加新的注册服务，在 `$HOME/.cargo/config.toml`（如果文件不存在则手动创建一个）中添加以下内容：

```
[registries]
ustc = { index = "https://mirrors.ustc.edu.cn/crates.io-index/" }
```

这种方式只会新增一个新的镜像地址，因此在引入依赖的时候，需要指定该地址，例如在项目中引入 `time` 包，你需要在 `Cargo.toml` 中使用以下方式引入：

```
[dependencies]
time = { registry = "ustc" }
```

在重新配置后，初次构建可能要较久的时间，因为要下载更新 `ustc` 注册服务的索引文件，由于文件比较大，需要等待较长的时间。

此处有两点需要注意：

1. cargo 1.68 版本开始支持稀疏索引，不再需要完整克隆 `crates.io-index` 仓库，可以加快获取包的速度，如：

```
[source.ustc]
registry = "sparse+https://mirrors.ustc.edu.cn/crates.io-index/"
```

2. cargo search 无法使用镜像

## 科大镜像

上面使用的是科大提供的注册服务，也是 Rust 最早期的注册服务，感谢大大的贡献。除此之外，大家还可以选择下面的镜像服务：

## 字节跳动

最大的优点就是不限速，当然，你的网速如果能跑到 1000Gbps，我们也可以认为它无情的限制了你，咳咳。

```
[source.crates-io]
replace-with = 'rsproxy'

[source.rsproxy]
registry = "https://rsproxy.cn/crates.io-index"

# 稀疏索引, 要求 cargo >= 1.68
[source.rsproxy-sparse]
registry = "sparse+https://rsproxy.cn/index/"

[registries.rsproxy]
index = "https://rsproxy.cn/crates.io-index"

[net]
git-fetch-with-cli = true
```

## 覆盖默认的镜像地址

事实上，我们更推荐第二种方式，因为第一种方式在项目大了后，实在是很麻烦，全部修改后，万一以后不用这个镜像了，你又要全部修改成其它的。

而第二种方式，则不需要修改 `Cargo.toml` 文件，**因为它是直接使用新注册服务来替代默认的 `crates.io`。**

在 `$HOME/.cargo/config.toml` 添加以下内容：

```
[source.crates-io]
replace-with = 'ustc'

[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

首先，创建一个新的镜像源 `[source.ustc]`，然后将默认的 `crates.io` 替换成新的镜像源：`replace-with = 'ustc'`。

简单吧？只要这样配置后，以往需要去 `crates.io` 下载的包，会全部从科大的镜像地址下载，速度刷刷的.. 我的 300M 大刀( 宽带 )终于有了用武之地。

**这里强烈推荐大家在学习完后面的基本章节后，看一下 [Cargo 使用指南章节](#)，对于你的 Rust 之旅会有莫大的帮助！**

# 下载卡住

下载卡住其实就一个原因：下载太慢了。

根据经验来看，卡住不动往往发生在更新索引时。毕竟 Rust 的包越来越多，索引也越来越大，如果不使用国内镜像，卡住还蛮正常的，好在，我们也无需经常更新索引:P

## Blocking waiting for file lock on package cache

不过这里有一个坑，需要大家注意，如果你同时打开了 VS CODE 和命令行，然后修改了 `Cargo.toml`，此时 VS CODE 的 `rust-analyzer` 插件会自动检测到依赖的变更，去下载新的依赖。

在 VS CODE 下载的过程中(特别是更新索引，可能会耗时很久)，假如你又在命令行中运行类似 `cargo run` 或者 `cargo build` 的命令，就会提示一行有些看不太懂的内容：

```
$ cargo build
  Blocking waiting for file lock on package cache
  Blocking waiting for file lock on package cache
```

其实这个报错就是因为 VS CODE 的下载太慢了，而且该下载构建还锁住了当前的项目，导致你无法在另一个地方再次进行构建。

解决办法也很简单：

- 增加下载速度，见前面内容
- 耐心等待持有锁的用户构建完成
- 强行停止正在构建的进程，例如杀掉 IDE 使用的 `rust-analyzer` 插件进程，然后删除 `$HOME/.cargo/.package_cache` 目录

# Rust 基本概念

从现在开始，我们正式踏入了 Rust 大陆，这片广袤而神秘的世界，在这个世界中，将接触到很多之前都没有听过概念：

- 所有权、借用、生命周期
- 宏编程
- 模式匹配

类似的还有很多，不过不用怕，引用武林外传一句话：咱上面有人。有本书在，一切虚妄终将烟消云散。

本章主要介绍 Rust 的基础语法、数据类型、项目结构等，学完本章，你将对 Rust 代码有一个清晰、完整的认识。

开始之前先通过一段代码来简单浏览下 Rust 的语法：

```
// Rust 程序入口函数，跟其它语言一样，都是 main，该函数目前无返回值
fn main() {
    // 使用let来声明变量，进行绑定，a是不可变的
    // 此处没有指定a的类型，编译器会默认根据a的值为a推断类型：i32，有符号32位整数
    // 语句的末尾必须以分号结尾
    let a = 10;
    // 主动指定b的类型为i32
    let b: i32 = 20;
    // 这里有两点值得注意：
    // 1. 可以在数值中带上类型：30i32表示数值是30，类型是i32
    // 2. c是可变的，mut是mutable的缩写
    let mut c = 30i32;
    // 还能在数值和类型中间添加一个下划线，让可读性更好
    let d = 30_i32;
    // 跟其它语言一样，可以使用一个函数的返回值来作为另一个函数的参数
    let e = add(add(a, b), add(c, d));

    // println!是宏调用，看起来像是函数但是它返回的是宏定义的代码块
    // 该函数将指定的格式化字符串输出到标准输出中（控制台）
    // {}是占位符，在具体执行过程中，会把e的值代入进来
    println!("( a + b ) + ( c + d ) = {}", e);
}

// 定义一个函数，输入两个i32类型的32位有符号整数，返回它们的和
fn add(i: i32, j: i32) -> i32 {
    // 返回相加值，这里可以省略return
    i + j
}
```

---

注意 在上面的 `add` 函数中，不要为 `i+j` 添加 `；`，这会改变语法导致函数返回 `()` 而不是 `i32`，具体参见[语句和表达式](#)。

---

有几点可以留意下：

- 字符串使用双引号 `" "` 而不是单引号 `' '`，Rust 中单引号是留给单个字符类型（`char`）使用的
- Rust 使用 `{}` 来作为格式化输出占位符，其它语言可能使用的是 `%s`，`%d`，`%p` 等，由于 `println!` 会自动推导出具体的类型，因此无需手动指定

# 变量绑定与解构

鉴于本书的目标读者（别慌，来到这里就说明你就是目标读者）已经熟练掌握其它任意一门编程语言，因此这里就不再对何为变量进行赘述，让我们开门见山来谈谈，为何 Rust 选择了手动设定变量的可变性。

## 为什么要手动设置变量的可变性？

在其它大多数语言中，要么只支持声明可变的变量，要么只支持声明不可变的变量（例如函数式语言），前者为编程提供了灵活性，后者为编程提供了安全性，而 Rust 比较野，选择了两者我都要，既要灵活性又要安全性。

能想要学习 Rust，说明我们的读者都是相当有水平的程序员了，你们应该能理解**一切选择皆是权衡**，那么两者都要的权衡是什么呢？这就是 Rust 开发团队为我们做出的贡献，两者都要意味着 Rust 语言底层代码的实现复杂度大幅提升，因此 Salute to The Rust Team!

除了以上两个优点，还有一个很大的优点，那就是运行性能上的提升，因为将本身无需改变的变量声明为不可变在运行期会避免一些多余的 `runtime` 检查。

## 变量命名

在命名方面，和其它语言没有区别，不过当给变量命名时，需要遵循 [Rust 命名规范](#)。

---

Rust 语言有一些**关键字** (*keywords*)，和其他语言一样，这些关键字都是被保留给 Rust 语言使用的，因此，它们不能被用作变量或函数的名称。在 [附录 A](#) 中可找到关键字列表。

## 变量绑定

在其它语言中，我们用 `var a = "hello world"` 的方式给 `a` 赋值，也就是把等式右边的 `"hello world"` 字符串赋值给变量 `a`，而在 Rust 中，我们这样写：`let a = "hello world"`，同时给这个过程起了另一个名字：**变量绑定**。

为何不用赋值而用绑定呢（其实你也可以称之为赋值，但是绑定的含义更清晰准确）？这里就涉及 Rust 最核心的原则——**所有权**，简单来讲，任何内存对象都是有主人的，而且一般情况下完全属于它的主人，

绑定就是把这个对象绑定给一个变量，让这个变量成为它的主人（聪明的读者应该能猜到，在这种情况下，该对象之前的主人就会丧失对该对象的所有权），像极了我们的现实世界，不是吗？

那为什么要引进“所有权”这个新的概念呢？请稍安勿躁，时机一旦成熟，我们就回来继续讨论这个话题。

## 变量可变性

Rust 的变量在默认情况下是**不可变的**。前文提到，这是 Rust 团队为我们精心设计的语言特性之一，让我们编写的代码更安全，性能也更好。当然你可以通过 `mut` 关键字让变量变为**可变的**，让设计更灵活。

如果变量 `a` 不可变，那么一旦为它绑定值，就不能再修改 `a`。举个例子，在我们的工程目录下使用 `cargo new variables` 新建一个项目，叫做 `variables`。

然后在新建的 `variables` 目录下，编辑 `src/main.rs`，改为下面代码：

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

保存文件，再使用 `cargo run` 运行它，迎面而来的是一条错误提示：

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

error: aborting due to previous error
```

具体的错误原因是 `cannot assign twice to immutable variable x`（无法对不可变的变量进行重复赋值），因为我们想为不可变的 `x` 变量再次赋值。

这种错误是为了避免无法预期的错误发生在我们的变量上：一个变量往往被多处代码所使用，其中一部分代码假定该变量的值永远不会改变，而另外一部分代码却无情的改变了这个值，在实际开发过程中，这个错误是很难被发现的，特别是在多线程编程中。

这种规则让我们的代码变得非常清晰，只有你想让你的变量改变时，它才能改变，这样就不会造成心智上的负担，也给别人阅读代码带来便利。

但是可变性也非常 important，否则我们就要像 ClojureScript 那样，每次要改变，就要重新生成一个对象，在拥有大量对象的场景，性能会变得非常低下，内存拷贝的成本异常的高。

在 Rust 中，可变性很简单，只要在变量名前加一个 `mut` 即可，而且这种显式的声明方式还会给后来人传达这样的信息：嗯，这个变量在后面代码部分会发生改变。

为了让变量声明为可变，将 `src/main.rs` 改为以下内容：

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

运行程序将得到下面结果：

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

选择可变还是不可变，更多的还是取决于你的使用场景，例如不可变可以带来安全性，但是丧失了灵活性和性能（如果你要改变，就要重新创建一个新的变量，这里涉及到内存对象的再分配）。而可变变量最大的好处就是使用上的灵活性和性能上的提升。

例如，在使用大型数据结构或者热点代码路径（被大量频繁调用）的情形下，在同一内存位置更新实例可能比复制并返回新分配的实例要更快。使用较小的数据结构时，通常创建新的实例并以更具函数式的风格来编写程序，可能会更容易理解，所以值得以较低的性能开销来确保代码清晰。

## 使用下划线开头忽略未使用的变量

如果你创建了一个变量却不在任何地方使用它，Rust 通常会给你一个警告，因为这可能会是个 BUG。但是有时创建一个不会被使用的变量是有用的，比如你正在设计原型或刚刚开始一个项目。这时**你希望告诉**

Rust 不要警告未使用的变量，为此可以用下划线作为变量名的开头：

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

使用 cargo run 运行下试试：

```
warning: unused variable: `y`
--> src/main.rs:3:9
3 |     let y = 10;
|         ^ help: 如果 y 故意不被使用, 请添加一个下划线前缀: `_y`
|
= note: `#[warn(unused_variables)]` on by default
```

可以看到，两个变量都是只有声明，没有使用，但是编译器却独独给出了 y 未被使用的警告，充分说明了 \_ 变量名前缀在这里发挥的作用。

值得注意的是，这里编译器还很善意的给出了提示( Rust 的编译器非常强大，这里的提示只是小意思 )：将 y 修改 \_y 即可。这里就不再给出代码，留给大家手动尝试并观察下运行结果。

更多关于 \_x 的使用信息，请阅读后面的[模式匹配章节](#)。

## 变量解构

let 表达式不仅仅用于变量的绑定，还能进行复杂变量的解构：从一个相对复杂的变量中，匹配出该变量的一部分内容：

```
fn main() {
    let (a, mut b): (bool, bool) = (true, false);
    // a = true, 不可变； b = false, 可变
    println!("a = {:?}", a, b);

    b = true;
    assert_eq!(a, b);
}
```

## 解构式赋值

在 Rust 1.59 版本后，我们可以在赋值语句的左式中使用元组、切片和结构体模式了。

```

struct Struct {
    e: i32
}

fn main() {
    let (a, b, c, d, e);

    (a, b) = (1, 2);
    // _ 代表匹配一个值，但是我们不关心具体的值是什么，因此没有使用一个变量名而是使用了 _
    [c, .., d, _] = [1, 2, 3, 4, 5];
    Struct { e, .. } = Struct { e: 5 };

    assert_eq!([1, 2, 1, 4, 5], [a, b, c, d, e]);
}

```

这种使用方式跟之前的 `let` 保持了一致性，但是 `let` 会重新绑定，而这里仅仅是对之前绑定的变量进行再赋值。

需要注意的是，使用 `+=` 的赋值语句还不支持解构式赋值。

这里用到了模式匹配的一些语法，如果大家看不懂没关系，可以在学完模式匹配章节后，再回头来看。

## 变量和常量之间的差异

变量的值不能更改可能让你想起其他另一个很多语言都有的编程概念：**常量(constant)**。与不可变变量一样，常量也是绑定到一个常量名且不允许更改的值，但是常量和变量之间存在一些差异：

- 常量不允许使用 `mut`。**常量不仅仅默认不可变，而且自始至终不可变**，因为常量在编译完成后，已经确定它的值。
- 常量使用 `const` 关键字而不是 `let` 关键字来声明，并且值的类型**必须标注**。

我们将在下一节[数据类型](#)中介绍，因此现在暂时无需关心细节。

下面是一个常量声明的例子，其常量名为 `MAX_POINTS`，值设置为 `100,000`。（Rust 常量的命名约定是全部字母都使用大写，并使用下划线分隔单词，另外对数字字面量可插入下划线以提高可读性）：

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

常量可以在任意作用域内声明，包括全局作用域，在声明的作用域内，常量在程序运行的整个过程中都有效。对于需要在多处代码共享一个不可变的值时非常有用，例如游戏中允许玩家赚取的最大点数或光速。

---

在实际使用中，最好将程序中用到的硬编码值都声明为常量，对于代码后续的维护有莫大的帮助。如果将来需要更改硬编码的值，你也只需要在代码中更改一处即可。

---

## 变量遮蔽(shadowing)

Rust 允许声明相同的变量名，在后面声明的变量会遮蔽掉前面声明的，如下所示：

```
fn main() {
    let x = 5;
    // 在main函数的作用域内对之前的x进行遮蔽
    let x = x + 1;

    {
        // 在当前的花括号作用域内，对之前的x进行遮蔽
        let x = x * 2;
        println!("The value of x in the inner scope is: {}", x);
    }

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

这个程序首先将数值 5 绑定到 `x`，然后通过重复使用 `let x =` 来遮蔽之前的 `x`，并取原来的值加上 1，所以 `x` 的值变成了 6。第三个 `let` 语句同样遮蔽前面的 `x`，取之前的值并乘上 2，得到的 `x` 最终值为 12。当运行此程序，将输出以下内容：

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
...
The value of x in the inner scope is: 12
The value of x is: 6
```

这和 `mut` 变量的使用是不同的，第二个 `let` 生成了完全不同的新变量，两个变量只是恰好拥有同样的名称，涉及一次内存对象的再分配，而 `mut` 声明的变量，可以修改同一个内存地址上的值，并不会发生内存对象的再分配，性能要更好。

变量遮蔽的用处在于，如果你在某个作用域内无需再使用之前的变量（在被遮蔽后，无法再访问到之前的同名变量），就可以重复的使用变量名字，而不用绞尽脑汁去想更多的名字。

例如，假设有一个程序要统计一个空格字符串的空格数量：

```
// 字符串类型
let spaces = "    ";
// usize数值类型
let spaces = spaces.len();
```

这种结构是允许的，因为第一个 `spaces` 变量是一个字符串类型，第二个 `spaces` 变量是一个全新的变量且和第一个具有相同的变量名，且是一个数值类型。所以变量遮蔽可以帮助我们节省些脑细胞，不用去想如 `spaces_str` 和 `spaces_num` 此类的变量名；相反我们可以重复使用更简单的 `spaces` 变量名。如果你不用 `let`：

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

运行一下，你就会发现编译器报错：

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

显然，Rust 对类型的要求很严格，不允许将整数类型 `usize` 赋值给字符串类型。`usize` 是一种 CPU 相关的整数类型，在[数值类型](#)中有详细介绍。

万事开头难，到目前为止，都进展很顺利，那下面开始，咱们正式进入 Rust 的类型世界，看看有哪些挑战在前面等着大家。

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 基本类型

当一门语言不谈类型时，你得小心，这大概率是动态语言(别拍我，我承认是废话)。但是把类型大张旗鼓的用多个章节去讲的，Rust 是其中之一。

Rust 每个值都有其确切的数据类型，总的来说可以分为两类：基本类型和复合类型。基本类型意味着它们往往是一个最小化原子类型，无法解构为其它类型(一般意义上来说)，由以下组成：

- 数值类型: 有符号整数 (`i8, i16, i32, i64, isize`)、无符号整数 (`u8, u16, u32, u64, usize`)、浮点数 (`f32, f64`)、以及有理数、复数
- 字符串: 字符串字面量和字符串切片 `&str`
- 布尔类型: `true` 和 `false`
- 字符类型: 表示单个 Unicode 字符，存储为 4 个字节
- 单元类型: 即 `()`，其唯一的值也是 `()`

## 类型推导与标注

与 Python、JavaScript 等动态语言不同，Rust 是一门静态类型语言，也就是编译器必须在编译期知道我们所有变量的类型，但这不意味着你需要为每个变量指定类型，因为 **Rust 编译器很聪明，它可以根据变量的值和上下文中的使用方式来自动推导出变量的类型**，同时编译器也不够聪明，在某些情况下，它无法推导出变量类型，需要手动去给予一个类型标注，关于这一点在 [Rust 语言初印象](#) 中有过展示。

来看段代码：

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

先忽略 `.parse().expect..` 部分，这段代码的目的是将字符串 "42" 进行解析，而编译器在这里无法推导出我们想要的类型：整数？浮点数？字符串？因此编译器会报错：

```
$ cargo build
Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
--> src/main.rs:2:9
2 |     let guess = "42".parse().expect("Not a number!");
|     ^^^^^ consider giving `guess` a type
```

因此我们需要提供给编译器更多的信息，例如给 `guess` 变量一个**显式的类型标注**：`let guess: i32 = ...` 或者 `"42".parse::<i32>()`。

# 数值类型

我朋友有一个领导（读者：你朋友？黑人问号）说过一句话：所有代码就是 0 和 1，简单的很。咱不评价这句话的正确性，但是计算机底层由 01 组成倒是真的。

计算机和数值关联在一起的时间，远比我们想象的要长，因此数值类型可以说是有计算机以来就有的类型，下面内容将深入讨论 Rust 的数值类型以及相关的运算符。

Rust 使用一个相对传统的语法来创建整数（1，2，...）和浮点数（1.0，1.1，...）。整数、浮点数的运算和你在其它语言上见过的一致，都是通过常见的运算符来完成。

---

不仅仅是数值类型，Rust 也允许在复杂类型上定义运算符，例如在自定义类型上定义 + 运算符，这种行为被称为运算符重载，Rust 具体支持的可重载运算符见[附录 B](#)。

---

## 整数类型

**整数**是没有小数部分的数字。之前使用过的 `i32` 类型，表示有符号的 32 位整数（`i` 是英文单词 *integer* 的首字母，与之相反的是 `u`，代表无符号 `unsigned` 类型）。下表显示了 Rust 中的内置的整数类型：

长度	有符号类型	无符号类型
8 位	<code>i8</code>	<code>u8</code>
16 位	<code>i16</code>	<code>u16</code>
32 位	<code>i32</code>	<code>u32</code>
64 位	<code>i64</code>	<code>u64</code>
128 位	<code>i128</code>	<code>u128</code>
视架构而定	<code>isize</code>	<code>usize</code>

类型定义的形式统一为：有无符号 + 类型大小(位数)。无符号数表示数字只能取正数和0，而有符号则表示数字可以取正数、负数还有0。就像在纸上写数字一样：当要强调符号时，数字前面可以带上正号或负号；然而，当很明显确定数字为正数时，就不需要加上正号了。有符号数字以[补码](#)形式存储。

每个有符号类型规定的数字范围是  $-(2^{n-1}) \sim 2^{n-1} - 1$ ，其中  $n$  是该定义形式的位长度。因此 `i8` 可存储数字范围是  $-(2^7) \sim 2^7 - 1$ ，即  $-128 \sim 127$ 。无符号类型可以存储的数字范围是  $0 \sim 2^n - 1$ ，所以 `u8` 能够存储的数字为  $0 \sim 2^8 - 1$ ，即  $0 \sim 255$ 。

此外，`isize` 和 `usize` 类型取决于程序运行的计算机 CPU 类型：若 CPU 是 32 位的，则这两个类型是 32 位的，同理，若 CPU 是 64 位，那么它们则是 64 位。

整形字面量可以用下表的形式书写：

数字字面量	示例
十进制	98_222
十六进制	0xff
八进制	0o77
二进制	0b1111_0000
字节(仅限于 <code>u8</code> )	<code>b'A'</code>

这么多类型，有没有一个简单的使用准则？答案是肯定的，Rust 整型默认使用 `i32`，例如 `let i = 1`，那 `i` 就是 `i32` 类型，因此你可以首选它，同时该类型也往往是性能最好的。`isize` 和 `usize` 的主要应用场景是用作集合的索引。

## 整型溢出

假设有一个 `u8`，它可以存放从 0 到 255 的值。那么当你将其修改为范围之外的值，比如 256，则会发 **生整型溢出**。关于这一行为 Rust 有一些有趣的规则：当在 debug 模式编译时，Rust 会检查整型溢出，若存在这些问题，则使程序在编译时 *panic*(崩溃,Rust 使用这个术语来表明程序因错误而退出)。

在当使用 `--release` 参数进行 release 模式构建时，Rust 不检测溢出。相反，当检测到整型溢出时，Rust 会按照补码循环溢出 (*two's complement wrapping*) 的规则处理。简而言之，大于该类型最大值的数值会被补码转换成该类型能够支持的对应数字的最小值。比如在 `u8` 的情况下，256 变成 0，257 变成 1，依此类推。程序不会 *panic*，但是该变量的值可能不是你期望的值。依赖这种默认行为的代码都应该被认为是错误的代码。

要显式处理可能的溢出，可以使用标准库针对原始数字类型提供的这些方法：

- 使用 `wrapping_*` 方法在所有模式下都按照补码循环溢出规则处理，例如 `wrapping_add`
- 如果使用 `checked_*` 方法时发生溢出，则返回 `None` 值
- 使用 `overflowing_*` 方法返回该值和一个指示是否存在溢出的布尔值
- 使用 `saturating_*` 方法使值达到最小值或最大值

下面是一个演示 `wrapping_*` 方法的示例：

```
fn main() {
    let a : u8 = 255;
    let b = a.wrapping_add(20);
    println!("{}", b); // 19
}
```

## 浮点类型

**浮点类型数字** 是带有小数点的数字，在 Rust 中浮点类型数字也有两种基本类型：`f32` 和 `f64`，分别为 32 位和 64 位大小。默认浮点类型是 `f64`，在现代的 CPU 中它的速度与 `f32` 几乎相同，但精度更高。

下面是一个演示浮点数的示例：

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

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

浮点数根据 IEEE-754 标准实现。`f32` 类型是单精度浮点型，`f64` 为双精度。

## 浮点数陷阱

浮点数由于底层格式的特殊性，导致了如果在使用浮点数时不够谨慎，就可能造成危险，有两个原因：

1. **浮点数往往是你想要数字的近似表达** 浮点数类型是基于二进制实现的，但是我们想要计算的数字往往是基于十进制，例如 `0.1` 在二进制上并不存在精确的表达形式，但是在十进制上就存在。这种不匹配性导致一定的歧义性，更多的，虽然浮点数能代表真实的数值，但是由于底层格式问题，它往往受限于定长的浮点数精度，如果你想要表达完全精准的真实数字，只有使用无限精度的浮点数才行
2. **浮点数在某些特性上是反直觉的** 例如大家都会觉得浮点数可以进行比较，对吧？是的，它们确实可以使用 `>`，`>=` 等进行比较，但是在某些场景下，这种直觉上的比较特性反而会害了你。因为 `f32`，`f64` 上的比较运算实现的是 `std::cmp::PartialEq` 特征（类似其他语言的接口），但是并没有实现 `std::cmp::Eq` 特征，但是后者在其它数值类型上都有定义，说了这么多，可能大家还是云里雾里，用一个例子来举例：

Rust 的 `HashMap` 数据结构，是一个 KV 类型的 Hash Map 实现，它对于 `K` 没有特定类型的限制，但是要求能用作 `K` 的类型必须实现了 `std::cmp::Eq` 特征，因此这意味着你无法使用浮点数作为 `HashMap` 的 `Key`，来存储键值对，但是作为对比，Rust 的整数类型、字符串类型、布尔类型都实现了该特征，因此可以作为 `HashMap` 的 `Key`。

为了避免上面说的两个陷阱，你需要遵守以下准则：

- 避免在浮点数上测试相等性
- 当结果在数学上可能存在未定义时，需要格外的小心

来看个小例子：

```
fn main() {
    // 断言0.1 + 0.2与0.3相等
    assert!(0.1 + 0.2 == 0.3);
}
```

你可能以为，这段代码没啥问题吧，实际上它会 *panic*(程序崩溃，抛出异常)，因为二进制精度问题，导致了  $0.1 + 0.2$  并不严格等于  $0.3$ ，它们可能在小数点 N 位后存在误差。

那如果非要进行比较呢？可以考虑用这种方式  $(0.1\_f64 + 0.2 - 0.3).abs() < 0.00001$ ，具体小于多少，取决于你对精度的需求。

讲到这里，相信大家基本已经明白了，为什么操作浮点数时要格外的小心，但是还不够，下面再来一段代码，直接震撼你的灵魂：

```
fn main() {
    let abc: (f32, f32, f32) = (0.1, 0.2, 0.3);
    let xyz: (f64, f64, f64) = (0.1, 0.2, 0.3);

    println!("abc (f32)");
    println!("    0.1 + 0.2: {:?}", (abc.0 + abc.1).to_bits());
    println!("    0.3: {:?}", (abc.2).to_bits());
    println!();

    println!("xyz (f64)");
    println!("    0.1 + 0.2: {:?}", (xyz.0 + xyz.1).to_bits());
    println!("    0.3: {:?}", (xyz.2).to_bits());
    println!();

    assert!(abc.0 + abc.1 == abc.2);
    assert!(xyz.0 + xyz.1 == xyz.2);
}
```

运行该程序，输出如下：

```

abc (f32)
0.1 + 0.2: 3e99999a
0.3: 3e99999a

xyz (f64)
0.1 + 0.2: 3fd333333333334
0.3: 3fd333333333333

thread 'main' panicked at 'assertion failed: xyz.0 + xyz.1 == xyz.2',
→ch2-add-floats.rs.rs:14:5
note: run with `RUST_BACKTRACE=1` environment variable to display
→a backtrace

```

仔细看，对 f32 类型做加法时， $0.1 + 0.2$  的结果是 3e99999a，0.3 也是 3e99999a，因此 f32 下的  $0.1 + 0.2 == 0.3$  通过测试，但是到了 f64 类型时，结果就不一样了，因为 f64 精度高很多，因此在小数点非常后面发生了一点微小的变化， $0.1 + 0.2$  以 4 结尾，但是 0.3 以 3 结尾，这个细微区别导致 f64 下的测试失败了，并且抛出了异常。

是不是**blow your mind away?** 没关系，在本书的后续章节中类似的直击灵魂的地方还很多，这就是敢号称 Rust 语言圣经 (Rust Course) 的底气！

## NaN

对于数学上未定义的结果，例如对负数取平方根 `-42.0.sqrt()`，会产生一个特殊的结果：Rust 的浮点数类型使用 NaN (not a number) 来处理这些情况。

**所有跟 NaN 交互的操作，都会返回一个 NaN**，而且 NaN 不能用来比较，下面的代码会崩溃：

```

fn main() {
    let x = (-42.0_f32).sqrt();
    assert_eq!(x, x);
}

```

出于防御性编程的考虑，可以使用 `is_nan()` 等方法，可以用来判断一个数值是否是 NaN：

```

fn main() {
    let x = (-42.0_f32).sqrt();
    if x.is_nan() {
        println!("未定义的数学行为")
    }
}

```

## 数字运算

Rust 支持所有数字类型的基本数学运算：加法、减法、乘法、除法和取模运算。下面代码各使用一条 `let` 语句来说明相应运算的用法：

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

    // 减法
    let difference = 95.5 - 4.3;

    // 乘法
    let product = 4 * 30;

    // 除法
    let quotient = 56.7 / 32.2;

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

这些语句中的每个表达式都使用了数学运算符，并且计算结果为一个值，然后绑定到一个变量上。[附录 B](#) 中给出了 Rust 提供的所有运算符的列表。

再来看一个综合性的示例：

```

fn main() {
    // 编译器会进行自动推导，给予twenty i32的类型
    let twenty = 20;
    // 类型标注
    let twenty_one: i32 = 21;
    // 通过类型后缀的方式进行类型标注：22是i32类型
    let twenty_two = 22i32;

    // 只有同样类型，才能运算
    let addition = twenty + twenty_one + twenty_two;
    println!("{} + {} + {} = {}", twenty, twenty_one, twenty_two, addition);

    // 对于较长的数字，可以用_进行分割，提升可读性
    let one_million: i64 = 1_000_000;
    println!("{}", one_million.pow(2));

    // 定义一个f32数组，其中42.0会自动被推导为f32类型
    let forty_twos = [
        42.0,
        42f32,
        42.0_f32,
    ];
    // 打印数组中第一个值，并控制小数位为2位
    println!(" {:.2}", forty_twos[0]);
}

```

## 位运算

Rust的位运算基本上和其他语言一样

运算符	说明
& 位与	相同位置均为1时则为1，否则为0
位或	相同位置只要有1时则为1，否则为0
^ 异或	相同位置不相同则为1，相同则为0
! 位非	把位中的0和1相互取反，即0置为1，1置为0
<< 左移	所有位向左移动指定位数，右位补0
>> 右移	所有位向右移动指定位数，带符号移动（正数补0，负数补1）

```

fn main() {
    // 二进制为00000010
    let a:i32 = 2;
    // 二进制为00000011
    let b:i32 = 3;

    println!("(a & b) value is {}", a & b);
    println!("(a | b) value is {}", a | b);
    println!("(a ^ b) value is {}", a ^ b);
    println!("(!b) value is {} ", !b);
    println!("(a << b) value is {}", a << b);
    println!("(a >> b) value is {}", a >> b);

    let mut a = a;
    // 注意这些计算符除了!之外都可以加上=进行赋值（因为!=是用来判断不等于）
    a <=> b;
    println!("(a << b) value is {}", a);
}

```

## 序列(Range)

Rust 提供了一个非常简洁的方式，用来生成连续的数值，例如 `1..5`，生成从 1 到 4 的连续数字，不包含 5；`1..=5`，生成从 1 到 5 的连续数字，包含 5，它的用途很简单，常常用于循环中：

```

for i in 1..=5 {
    println!("{}" , i);
}

```

最终程序输出：

```

1
2
3
4
5

```

序列只允许用于数字或字符类型，原因是：它们可以连续，同时编译器在编译期可以检查该序列是否为空，字符和数字值是 Rust 中仅有的可以用于判断是否为空的类型。如下是一个使用字符类型序列的例子：

```
for i in 'a'..='z' {
    println!("{}", i);
}
```

## 使用 As 完成类型转换

Rust 中可以使用 As 来完成一个类型到另一个类型的转换，其最常用于将原始类型转换为其他原始类型，但是它也可以完成诸如将指针转换为地址、地址转换为指针以及将指针转换为其他指针等功能。你可以在[这里](#)了解更多相关的知识。

## 有理数和复数

Rust 的标准库相比其它语言，准入门槛较高，因此有理数和复数并未包含在标准库中：

- 有理数和复数
- 任意大小的整数和任意精度的浮点数
- 固定精度的十进制小数，常用于货币相关的场景

好在社区已经开发出高质量的 Rust 数值库：[num](#)。

按照以下步骤来引入 num 库：

1. 创建新工程 `cargo new complex-num && cd complex-num`
2. 在 `Cargo.toml` 中的 `[dependencies]` 下添加一行 `num = "0.4.0"`
3. 将 `src/main.rs` 文件中的 `main` 函数替换为下面的代码
4. 运行 `cargo run`

```
use num::complex::Complex;

fn main() {
    let a = Complex { re: 2.1, im: -1.2 };
    let b = Complex::new(11.1, 22.2);
    let result = a + b;

    println!("{} + {}i", result.re, result.im)
}
```

## 总结

之前提到了过 Rust 的数值类型和运算跟其他语言较为相似，但是实际上，除了语法上的不同之外，还是存在一些差异点：

- **Rust 拥有相当多的数值类型.** 因此你需要熟悉这些类型所占用的字节数，这样就知道该类型允许的大小范围以及你选择的类型是否能表达负数
- **类型转换必须是显式的.** Rust 永远也不会偷偷把你的 16bit 整数转换成 32bit 整数
- **Rust 的数值上可以使用方法.** 例如你可以用以下方法来将 13.14 取整：`13.14_f32.round()`，在这里我们使用了类型后缀，因为编译器需要知道 13.14 的具体类型

数值类型的讲解已经基本结束，接下来，来看看字符和布尔类型。

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 字符、布尔、单元类型

这三个类型所处的地位比较尴尬，你说它们重要吧，确实出现的身影不是很多，说它们不重要吧，有时候也是不可或缺，而且这三个类型都有一个共同点：简单，因此我们统一放在一起讲。

## 字符类型(char)

字符，对于没有其它编程经验的新手来说可能不太好理解（没有编程经验敢来学 Rust 的绝对是好汉），但是你可以把它理解为英文中的字母，中文中的汉字。

下面的代码展示了几个颇具异域风情的字符：

```
fn main() {
    let c = 'z';
    let z = ' ZX';
    let g = '国';
    let heart_eyed_cat = '😻';
}
```

如果大家是从有年代感的编程语言过来，可能会大喊一声：这 XX 叫字符？是的，在 Rust 语言中这些都是字符，Rust 的字符不仅仅是 ASCII，所有的 Unicode 值都可以作为 Rust 字符，包括单个的中文、日文、韩文、emoji 表情符号等等，都是合法的字符类型。Unicode 值的范围从 U+0000 ~ U+D7FF 和 U+E000 ~ U+10FFFF。不过“字符”并不是 Unicode 中的一个概念，所以人在直觉上对“字符”的理解和 Rust 的字符概念并不一致。

由于 Unicode 都是 4 个字节编码，因此字符类型也是占用 4 个字节：

```
fn main() {
    let x = '中';
    println!("字符'中'占用了{}字节的内存大小", std::mem::size_of_val(&x));
}
```

输出如下：

```
$ cargo run
Compiling ...
字符'中'占用了4字节的内存大小
```

---

注意，我们还没开始讲字符串，但是这里提前说一下，和一些语言不同，Rust 的字符只能用 `''` 来表示，`""` 是留给字符串的。

---

## 布尔(bool)

Rust 中的布尔类型有两个可能的值： `true` 和 `false`，布尔值占用内存的大小为 1 个字节：

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

    let f: bool = false; // 使用类型标注, 显式指定f的类型

    if f {
        println!("这是段毫无意义的代码");
    }
}
```

使用布尔类型的场景主要在于流程控制，例如上述代码中的 `if` 就是其中之一。

## 单元类型

单元类型就是 `()`，对，你没看错，就是 `()`，唯一的值也是 `()`，一些读者读到这里可能就不愿意了，你也太敷衍了吧，管这叫类型？

只能说，再不起眼的东西，都有其用途，在目前为止的学习过程中，大家已经看到过很多次 `fn main()` 函数的使用吧？那么这个函数返回什么呢？

没错，`main` 函数就返回这个单元类型 `()`，你不能说 `main` 函数无返回值，因为没有返回值的函数在 Rust 中是有单独的定义的：发散函数(`diverge function`)，顾名思义，无法收敛的函数。

例如常见的 `println!()` 的返回值也是单元类型 `()`。

再比如，你可以用 `()` 作为 `map` 的值，表示我们不关注具体的值，只关注 `key`。这种用法和 Go 语言的 `struct{}` 类似，可以作为一个值用来占位，但是完全**不占用**任何内存。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 语句和表达式

Rust 的函数体是由一系列语句组成，最后由一个表达式来返回值，例如：

```
fn add_with_extra(x: i32, y: i32) -> i32 {  
    let x = x + 1; // 语句  
    let y = y + 5; // 语句  
    x + y // 表达式  
}
```

语句会执行一些操作但是不会返回一个值，而表达式会在求值后返回一个值，因此在上述函数体的三行代码中，前两行是语句，最后一行是表达式。

对于 Rust 语言而言，**这种基于语句（statement）和表达式（expression）的方式是非常重要的，你需要能明确的区分这两个概念**，但是对于很多其它语言而言，这两个往往无需区分。基于表达式是函数式语言的重要特征，**表达式总要返回值**。

其实，在此之前，我们已经多次使用过语句和表达式。

## 语句

```
let a = 8;  
let b: Vec<f64> = Vec::new();  
let (a, c) = ("hi", false);
```

以上都是语句，它们完成了一个具体的操作，但是并没有返回值，因此是语句。

由于 `let` 是语句，因此不能将 `let` 语句赋值给其它值，如下形式是错误的：

```
let b = (let a = 8);
```

错误如下：

```

error: expected expression, found statement (`let`) // 期望表达式, 却发现`let`语句
--> src/main.rs:2:13
2 |     let b = let a = 8;
   |           ^^^^^^
   |
= note: variable declaration using `let` is a statement `let`是一条语句

error[E0658]: `let` expressions in this position are experimental
              // 下面的 `let` 用法目前是试验性的, 在稳定版中尚不能使用
--> src/main.rs:2:13
2 |     let b = let a = 8;
   |           ^^^^^^
   |
= note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667> for more
information
= help: you can write `matches!(<expr>, <pattern>)` instead of `let <pattern> =
<expr>`
```

以上的错误告诉我们 `let` 是语句, 不是表达式, 因此它不返回值, 也就不能给其它变量赋值。但是该错误还透漏了一个重要的信息, `let` 作为表达式已经是试验功能了, 也许不久的将来, 我们在 [stable rust](#) 下可以这样使用。

## 表达式

表达式会进行求值, 然后返回一个值。例如 `5 + 6`, 在求值后, 返回值 `11`, 因此它就是一条表达式。

表达式可以成为语句的一部分, 例如 `let y = 6` 中, `6` 就是一个表达式, 它在求值后返回一个值 `6` (有些反直觉, 但是确实是表达式)。

调用一个函数是表达式, 因为会返回一个值, 调用宏也是表达式, 用花括号包裹最终返回一个值的语句块也是表达式, 总之, 能返回值, 它就是表达式:

```

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

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

上面使用一个语句块表达式将值赋给 `y` 变量, 语句块长这样:

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

该语句块是表达式的原因是：它的最后一行是表达式，返回了 `x + 1` 的值，注意 `x + 1` 不能以分号结尾，否则就会从表达式变成语句，**表达式不能包含分号**。这一点非常重要，一旦你在表达式后加上分号，它就会变成一条语句，再也不会返回一个值，请牢记！

最后，表达式如果不返回任何值，会隐式地返回一个 `()`。

```
fn main() {  
    assert_eq!(ret_unit_type(), ())  
}  
  
fn ret_unit_type() {  
    let x = 1;  
    // if 语句块也是一个表达式，因此可以用于赋值，也可以直接返回  
    // 类似三元运算符，在Rust里我们可以这样写  
    let y = if x % 2 == 1 {  
        "odd"  
    } else {  
        "even"  
    };  
    // 或者写成一行  
    let z = if x % 2 == 1 { "odd" } else { "even" };  
}
```

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

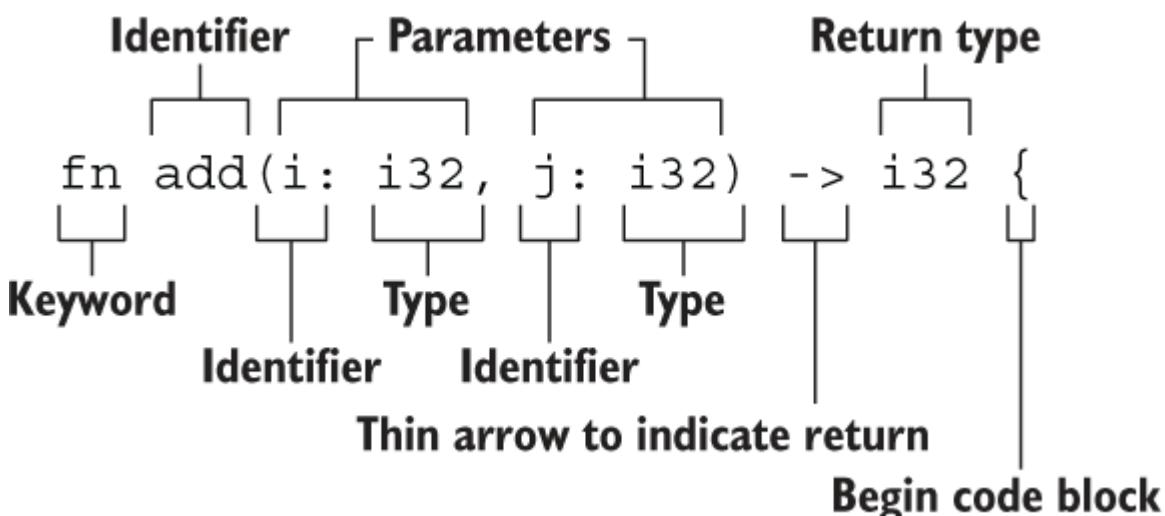
# 函数

Rust 的函数我们在之前已经见过不少，跟其他语言几乎没有什么区别。因此本章的学习之路将轻松和愉快，骚年们，请珍惜这种愉快，下一章你将体验到不一样的 Rust。

在函数界，有一个函数只闻其名不闻其声，可以止小孩啼！在程序界只有 `hello,world!` 可以与之媲美，它就是 `add` 函数：

```
fn add(i: i32, j: i32) -> i32 {  
    i + j  
}
```

该函数如此简单，但是又是如此的五脏俱全，声明函数的关键字 `fn`，函数名 `add()`，参数 `i` 和 `j`，参数类型和返回值类型都是 `i32`，总之一切那么的普通，但是又那么的自信，直到你看到了下面这张图：



当你看懂了这张图，其实就等于差不多完成了函数章节的学习，但是这么短的章节显然对不起读者老爷们的厚爱，所以我们来展开下。

## 函数要点

- 函数名和变量名使用[蛇形命名法\(snake case\)](#)，例如 `fn add_two() -> {}`
- 函数的位置可以随便放，Rust 不关心我们在哪里定义了函数，只要有定义即可
- 每个函数参数都需要标注类型

## 函数参数

Rust 是强类型语言，因此需要你为每一个函数参数都标识出它的具体类型，例如：

```
fn main() {
    another_function(5, 6.1);
}

fn another_function(x: i32, y: f32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

another\_function 函数有两个参数，其中 x 是 i32 类型，y 是 f32 类型，然后在该函数内部，打印出这两个值。这里去掉 x 或者 y 的任何一个的类型，都会报错：

```
fn main() {
    another_function(5, 6.1);
}

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

错误如下：

```
error: expected one of `:`, `@`, or `|`, found `)`
--> src/main.rs:5:30
|
5 | fn another_function(x: i32, y) {
|                                ^ expected one of `:`, `@`, or `|` // 期待以下符号之一
|                                `:`, `@`, or `|`
|
= note: anonymous parameters are removed in the 2018 edition (see RFC 1685)
// 匿名参数在 Rust 2018 edition 中就已经移除
help: if this is a parameter name, give it a type // 如果y是一个参数名，请给予它一个类型
|
5 | fn another_function(x: i32, y: TypeName) {
|                                ~~~~~
help: if this is a type, explicitly ignore the parameter name // 如果y是一个类型，请使用_
忽略参数名
|
5 | fn another_function(x: i32, _: y) {
|                                ~~~~
```

## 函数返回

在上一章节语句和表达式中，我们有提到，在 Rust 中函数就是表达式，因此我们可以把函数的返回值直接赋给调用者。

函数的返回值就是函数体最后一条表达式的返回值，当然我们也可以使用 `return` 提前返回，下面的函数使用最后一条表达式来返回一个值：

```
fn plus_five(x:i32) -> i32 {
    x + 5
}

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

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

`x + 5` 是一条表达式，求值后，返回一个值，因为它是函数的最后一行，因此该表达式的值也是函数的返回值。

再来看两个重点：

1. `let x = plus_five(5)`，说明我们用一个函数的返回值来初始化 `x` 变量，因此侧面说明了在 Rust 中函数也是表达式，这种写法等同于 `let x = 5 + 5;`
2. `x + 5` 没有分号，因为它是一条表达式，这个在上一节中我们也有详细介绍

再来看一段代码，同时使用 `return` 和表达式作为返回值：

```
fn plus_or_minus(x:i32) -> i32 {
    if x > 5 {
        return x - 5
    }

    x + 5
}

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

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

`plus_or_minus` 函数根据传入 `x` 的大小来决定是做加法还是减法，若 `x > 5` 则通过 `return` 提前返回 `x - 5` 的值，否则返回 `x + 5` 的值。

## Rust 中的特殊返回类型

### 无返回值()

对于 Rust 新手来说，有些返回类型很难理解，而且如果你想通过百度或者谷歌去搜索，都不好查询，因为这些符号太常见了，根本难以精确搜索到。

例如单元类型 ()，是一个零长度的元组。它没啥作用，但是可以用来表达一个函数没有返回值：

- 函数没有返回值，那么返回一个 ()
- 通过 ; 结尾的表达式返回一个 ()

例如下面的 report 函数会隐式返回一个 ()：

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

fn report<T: Debug>(item: T) {
    println!("{}:?", item);

}
```

与上面的函数返回值相同，但是下面的函数显式的返回了 ()：

```
fn clear(text: &mut String) -> () {
    *text = String::from("");
}
```

在实际编程中，你会经常在错误提示中看到该 () 的身影出没，假如你的函数需要返回一个 u32 值，但是如果你不幸的以 表达式；的方式作为函数的最后一行代码，就会报错：

```
fn add(x:u32,y:u32) -> u32 {
    x + y;
}
```

错误如下：

```
error[E0308]: mismatched types // 类型不匹配
--> src/main.rs:6:24
|
6 | fn add(x:u32,y:u32) -> u32 {
|     ---                                ^^^ expected `u32` , found `()` // 期望返回u32,却返回()
|     |
|     implicitly returns `()` as its body has no tail or `return` expression
7 |     x + y;
|             - help: consider removing this semicolon
```

还记得我们在[语句与表达式](#)中讲过的吗？只有表达式能返回值，而；结尾的是语句，在Rust中，一定要严格区分**表达式**和**语句**的区别，这个在其它语言中往往是被忽视的点。

### 永不返回的发散函数！

当用`!`作函数返回类型的时候，表示该函数永不返回( diverge function )，特别的，这种语法往往用做会导致程序崩溃的函数：

```
fn dead_end() -> ! {
    panic!("你已经到了穷途末路，崩溃吧！");
}
```

下面的函数创建了一个无限循环，该循环永不跳出，因此函数也永不返回：

```
fn forever() -> ! {
    loop {
        //...
    };
}
```

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 所有权和借用

Rust 之所以能成为万众瞩目的语言，就是因为其内存安全性。在以往，内存安全几乎都是通过 GC 的方式实现，但是 GC 会引来性能、内存占用以及 Stop the world 等问题，在高性能场景和系统编程上是不可接受的，因此 Rust 采用了与(错)众(误)不(之)同(源)的方式：**所有权系统**。

理解**所有权和借用**，对于 Rust 学习是至关重要的，因此我们把本章提到了非常靠前的位置，So，骚年们，准备好迎接狂风暴雨了嘛？

从现在开始，鉴于大家已经掌握了非常基本的语法，有些时候，在示例代码中，将省略 `fn main() {}` 的模版代码，只要将相应的示例放在 `fn main() {}` 中，即可运行。

# 所有权

所有的程序都必须和计算机内存打交道，如何从内存中申请空间来存放程序的运行内容，如何在不需要的时候释放这些空间，成了重中之重，也是所有编程语言设计的难点之一。在计算机语言不断演变过程中，出现了三种流派：

- **垃圾回收机制(GC)**, 在程序运行时不断寻找不再使用的内存，典型代表：Java、Go
- **手动管理内存的分配和释放**, 在程序中，通过函数调用的方式来申请和释放内存，典型代表：C++
- **通过所有权来管理内存**, 编译器在编译时会根据一系列规则进行检查

其中 Rust 选择了第三种，最妙的是，这种检查只发生在编译期，因此对于程序运行期，不会有任何性能上的损失。

由于所有权是一个新概念，因此读者需要花费一些时间来掌握它，一旦掌握，海阔天空任你飞跃，在本章，我们将通过 `字符串` 来引导讲解所有权的相关知识。

## 一段不安全的代码

先来看看一段来自 C 语言的糟糕代码：

```
int* foo() {
    int a;           // 变量a的作用域开始
    a = 100;
    char *c = "xyz"; // 变量c的作用域开始
    return &a;
}                   // 变量a和c的作用域结束
```

这段代码虽然可以编译通过，但是其实非常糟糕，变量 `a` 和 `c` 都是局部变量，函数结束后将局部变量 `a` 的地址返回，但局部变量 `a` 存在栈中，在离开作用域后，`a` 所申请的栈上内存都会被系统回收，从而造成了 悬空指针(Dangling Pointer) 的问题。这是一个非常典型的内存安全问题，虽然编译可以通过，但是运行的时候会出现错误，很多编程语言都存在。

再来看变量 `c`，`c` 的值是常量字符串，存储于常量区，可能这个函数我们只调用了一次，也可能我们不再会使用这个字符串，但 `"xyz"` 只有当整个程序结束后系统才能回收这片内存。

所以内存安全问题，一直都是程序员非常头疼的问题，好在，在 Rust 中这些问题即将成为历史，因为 Rust 在编译的时候就可以帮助我们发现内存不安全的问题，那 Rust 如何做到这一点呢？

在正式进入主题前，先来一个预热知识。

# 栈(Stack)与堆(Heap)

栈和堆是编程语言最核心的数据结构，但是在很多语言中，你并不需要深入了解栈与堆。但对于 Rust 这样的系统编程语言，值是位于栈上还是堆上非常重要，因为这会影响程序的行为和性能。

栈和堆的核心目标就是为程序在运行时提供可供使用的内存空间。

## 栈

栈按照顺序存储值并以相反顺序取出值，这也被称作**后进先出**。想象一下一叠盘子：当增加更多盘子时，把它们放在盘子堆的顶部，当需要盘子时，再从顶部拿走。不能从中间也不能从底部增加或拿走盘子！

增加数据叫做**进栈**，移出数据则叫做**出栈**。

因为上述的实现方式，栈中的所有数据都必须占用已知且固定大小的内存空间，假设数据大小是未知的，那么在取出数据时，你将无法取到你想要的数据。

## 堆

与栈不同，对于大小未知或者可能变化的数据，我们需要将它存储在堆上。

当向堆上放入数据时，需要请求一定大小的内存空间。操作系统在堆的某处找到一块足够大的空位，把它标记为已使用，并返回一个表示该位置地址的**指针**，该过程被称为**在堆上分配内存**，有时简称为“分配”(allocating)。

接着，该指针会被推入**栈**中，因为指针的大小是已知且固定的，在后续使用过程中，你将通过栈中的**指针**，来获取数据在堆上的实际内存位置，进而访问该数据。

由上可知，堆是一种缺乏组织的数据结构。想象一下去餐馆就座吃饭：进入餐馆，告知服务员有几个人，然后服务员找到一个够大的空桌子（堆上分配的内存空间）并领你们过去。如果有人来迟了，他们也可以通过桌号（栈上的指针）来找到你们坐在哪。

## 性能区别

写入方面：入栈比在堆上分配内存要快，因为入栈时操作系统无需分配新的空间，只需要将新数据放入栈顶即可。相比之下，在堆上分配内存则需要更多的工作，这是因为操作系统必须首先找到一块足够存放数据的内存空间，接着做一些记录为下一次分配做准备。

读取方面：得益于 CPU 高速缓存，使得处理器可以减少对内存的访问，高速缓存和内存的访问速度差异在 10 倍以上！栈数据往往可以直接存储在 CPU 高速缓存中，而堆数据只能存储在内存中。访问堆上的数据比访问栈上的数据慢，因为必须先访问栈再通过栈上的指针来访问内存。

因此，处理器处理分配在栈上数据会比在堆上的数据更加高效。

## 所有权与堆栈

当你的代码调用一个函数时，传递给函数的参数（包括可能指向堆上数据的指针和函数的局部变量）依次被压入栈中，当函数调用结束时，这些值将被从栈中按照相反的顺序依次移除。

因为堆上的数据缺乏组织，因此跟踪这些数据何时分配和释放是非常重要的，否则堆上的数据将产生内存泄漏——这些数据将永远无法被回收。这就是 Rust 所有权系统为我们提供的强大保障。

对于其他很多编程语言，你确实无需理解堆栈的原理，但是在 **Rust 中，明白堆栈的原理，对于我们理解所有权的工作原理会有很大的帮助。**

## 所有权原则

理解了堆栈，接下来看一下关于所有权的规则，首先请谨记以下规则：

- 
1. Rust 中每一个值都被一个变量所拥有，该变量被称为值的所有者
  2. 一个值同时只能被一个变量所拥有，或者说一个值只能拥有一个所有者
  3. 当所有者(变量)离开作用域范围时，这个值将被丢弃(drop)
- 

## 变量作用域

作用域是一个变量在程序中有效的范围，假如有这样一个变量：

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

变量 s 绑定到了一个字符串字面值，该字符串字面值是硬编码到程序代码中的。 s 变量从声明的点开始直到当前作用域的结束都是有效的：

```
{                                // s 在这里无效，它尚未声明
    let s = "hello";      // 从此处起，s 是有效的
    // 使用 s
}                                // 此作用域已结束，s 不再有效
```

简而言之， s 从创建开始就有效，然后有效期持续到它离开作用域为止，可以看出，就作用域来说，Rust 语言跟其他编程语言没有区别。

## 简单介绍 String 类型

之前提到过，本章会用 `String` 作为例子，因此这里会进行一下简单的介绍，具体的 `String` 学习请参见 [String 类型](#)。

我们已经见过字符串字面值 `let s = "hello"`，`s` 是被硬编码进程序里的字符串值（类型为 `&str`）。字符串字面值是很方便的，但是它并不适用于所有场景。原因有二：

- **字符串字面值是不可变的**，因为被硬编码到程序代码中
- 并非所有字符串的值都能在编写代码时得知

例如，字符串是需要程序运行时，通过用户动态输入然后存储在内存中的，这种情况，字符串字面值就完全无用武之地。为此，Rust 为我们提供动态字符串类型：`String`，该类型被分配到堆上，因此可以动态伸缩，也就能存储在编译时大小未知的文本。

可以使用下面的方法基于字符串字面量来创建 `String` 类型：

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

`::` 是一种调用操作符，这里表示调用 `String` 中的 `from` 方法，因为 `String` 存储在堆上是动态的，你可以这样修改它：

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

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

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

言归正传，了解 `String` 内容后，一起来看看关于所有权的交互。

## 变量绑定背后的的数据交互

### 转移所有权

先来看一段代码：

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

代码背后的逻辑很简单，将 `5` 绑定到变量 `x`；接着拷贝 `x` 的值赋给 `y`，最终 `x` 和 `y` 都等于 `5`，因为整数是 Rust 基本数据类型，是固定大小的简单值，因此这两个值都是通过自动拷贝的方式来赋值的，都

被存在栈中，完全无需在堆上分配内存。

可能有同学会有疑问：这种拷贝不消耗性能吗？实际上，这种栈上的数据足够简单，而且拷贝非常非常快，只需要复制一个整数大小（`i32`，4个字节）的内存即可，因此在这种情况下，拷贝的速度远比在堆上创建内存来得快的多。实际上，上一章我们讲到的 Rust 基本类型都是通过自动拷贝的方式来赋值的，就像上面代码一样。

然后再来看一段代码：

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

此时，可能某个大聪明(善意昵称)已经想到了：嗯，把 `s1` 的内容拷贝一份赋值给 `s2`，实际上，并不是这样。之前也提到了，对于基本类型（存储在栈上），Rust 会自动拷贝，但是 `String` 不是基本类型，而且是存储在堆上的，因此不能自动拷贝。

实际上，`String` 类型是一个复杂类型，由**存储在栈中的堆指针、字符串长度、字符串容量共同组成**，其中**堆指针**是最重要的，它指向了真实存储字符串内容的堆内存，至于长度和容量，如果你有 Go 语言的经验，这里就很好理解：容量是堆内存分配空间的大小，长度是目前已经使用的大小。

总之 `String` 类型指向了一个堆上的空间，这里存储着它的真实数据，下面对上面代码中的 `let s2 = s1` 分成两种情况讨论：

1. 拷贝 `String` 和存储在堆上的字节数组 如果该语句是拷贝所有数据(深拷贝)，那么无论是 `String` 本身还是底层的堆上数据，都会被全部拷贝，这对于性能而言会造成非常大的影响
2. 只拷贝 `String` 本身 这样的拷贝非常快，因为在 64 位机器上就拷贝了 8字节的指针、8字节的长度、8字节的容量，总计 24 字节，但是带来了新的问题，还记得我们之前提到的所有权规则吧？其中有一条就是：**一个值只允许有一个所有者**，而现在这个值（堆上的真实字符串数据）有了两个所有者：`s1` 和 `s2`。

好吧，就假定一个值可以拥有两个所有者，会发生什么呢？

当变量离开作用域后，Rust 会自动调用 `drop` 函数并清理变量的堆内存。不过由于两个 `String` 变量指向了同一位置。这就有了一个问题：当 `s1` 和 `s2` 离开作用域，它们都会尝试释放相同的内存。这是一个叫做**二次释放 (double free)** 的错误，也是之前提到过的内存安全性 BUG 之一。两次释放（相同）内存会导致内存污染，它可能会导致潜在的安全漏洞。

因此，Rust 这样解决问题：**当 `s1` 赋予 `s2` 后，Rust 认为 `s1` 不再有效，因此也无需在 `s1` 离开作用域后 `drop` 任何东西，这就是把所有权从 `s1` 转移给了 `s2`，`s1` 在被赋予 `s2` 后就马上失效了。**

再来看看，在所有权转移后再来使用旧的所有者，会发生什么：

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

println!("{}， world!", s1);
```

由于 Rust 禁止你使用无效的引用，你会看到以下的错误：

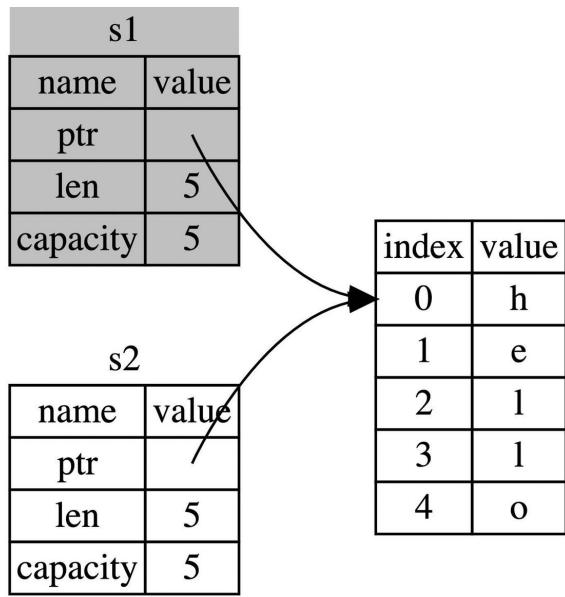
```
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:5:28
|
2 |     let s1 = String::from("hello");
|         -- move occurs because `s1` has type `String`, which does not implement
the `Copy` trait
3 |     let s2 = s1;
|             -- value moved here
4 |
5 |     println!("{}， world!", s1);
|             ^^ value borrowed here after move
|
= note: this error originates in the macro `$crate::format_args_nl` which comes
from the expansion of the macro `println` (in Nightly builds, run with -Z macro-
backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
|
3 |     let s2 = s1.clone();
|             ++++++++
For more information about this error, try `rustc --explain E0382`.
```

现在再回头看看之前的规则，相信大家已经有了更深刻的理解：

- 
1. Rust 中每一个值都被一个变量所拥有，该变量被称为值的所有者
  2. 一个值同时只能被一个变量所拥有，或者说一个值只能拥有一个所有者
  3. 当所有者(变量)离开作用域范围时，这个值将被丢弃(drop)

---

如果你在其他语言中听说过术语 **浅拷贝(shallow copy)** 和 **深拷贝(deep copy)**，那么拷贝指针、长度和容量而不拷贝数据听起来就像浅拷贝，但是又因为 Rust 同时使第一个变量 `s1` 无效了，因此这个操作被称为 **移动(move)**，而不是浅拷贝。上面的例子可以解读为 `s1` 被**移动到了** `s2` 中。那么具体发生了什么，用一张图简单说明：



这样就解决了我们之前的问题，`s1` 不再指向任何数据，只有 `s2` 是有效的，当 `s2` 离开作用域，它就会释放内存。相信此刻，你应该明白了，为什么 Rust 称呼 `let a = b` 为**变量绑定**了吧？

再来看一段代码：

```
fn main() {
    let x: &str = "hello, world";
    let y = x;
    println!("{} , {}", x, y);
}
```

这段代码，大家觉得会否报错？如果参考之前的 `String` 所有权转移的例子，那这段代码也应该报错才是，但是实际上呢？

这段代码和之前的 `String` 有一个本质上的区别：在 `String` 的例子中 `s1` 持有了通过 `String::from("hello")` 创建的值的所有权，而这个例子中，`x` 只是引用了存储在二进制中的字符串 `"hello, world"`，并没有持有所有权。

因此 `let y = x` 中，仅仅是对该引用进行了拷贝，此时 `y` 和 `x` 都引用了同一个字符串。**如果还不理解也没关系，当学习了下一章节“引用与借用”后，大家自然而然言就会理解。**

## 克隆(深拷贝)

首先，**Rust 永远也不会自动创建数据的“深拷贝”**。因此，任何**自动**的复制都不是深拷贝，可以被认为对运行时性能影响较小。

如果我们**确实**需要深度复制 `String` 中堆上的数据，而不仅仅是栈上的数据，可以使用一个叫做 `clone` 的方法。

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

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

这段代码能够正常运行，因此说明 `s2` 确实完整的复制了 `s1` 的数据。

如果代码性能无关紧要，例如初始化程序时，或者在某段时间只会执行一次时，你可以使用 `clone` 来简化编程。但是对于执行较为频繁的代码(热点路径)，使用 `clone` 会极大的降低程序性能，需要小心使用！

## 拷贝(浅拷贝)

浅拷贝只发生在栈上，因此性能很高，在日常编程中，浅拷贝无处不在。

再回到之前看过的例子：

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

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

但这段代码似乎与我们刚刚学到的内容相矛盾：没有调用 `clone`，不过依然实现了类似深拷贝的效果——没有报所有权的错误。

原因是像整型这样的基本类型在编译时是已知大小的，会被存储在栈上，所以拷贝其实际的值是快速的。这意味着没有理由在创建变量 `y` 后使 `x` 无效（`x`、`y` 都仍然有效）。换句话说，这里没有深浅拷贝的区别，因此这里调用 `clone` 并不会与通常的浅拷贝有什么不同，我们可以不用管它（可以理解成在栈上做了深拷贝）。

Rust 有一个叫做 `Copy` 的特征，可以用在类似整型这样在栈中存储的类型。如果一个类型拥有 `Copy` 特征，一个旧的变量在被赋值给其他变量后仍然可用。

那么什么类型是可 `Copy` 的呢？可以查看给定类型的文档来确认，不过作为一个通用的规则：**任何基本类型的组合可以 `Copy`，不需要分配内存或某种形式资源的类型是可以 `Copy` 的**。如下是一些 `Copy` 的类型：

- 所有整数类型，比如 `u32`
- 布尔类型，`bool`，它的值是 `true` 和 `false`
- 所有浮点数类型，比如 `f64`
- 字符类型，`char`
- 元组，当且仅当其包含的类型也都是 `Copy` 的时候。比如，`(i32, i32)` 是 `Copy` 的，但 `(i32, String)` 就不是

- 不可变引用 `&T`，例如[转移所有权](#)中的最后一个例子，**但是注意：可变引用 `&mut T` 是不可以 Copy 的**

## 函数传值与返回

将值传递给函数，一样会发生 移动 或者 复制，就跟 `let` 语句一样，下面的代码展示了所有权、作用域的规则：

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

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

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

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

} // 这里，x 先移出了作用域，然后是 s。但因为 s 的值已被移走，
// 所以不会有特殊操作

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

fn makes_copy(some_integer: i32) { // some_integer 进入作用域
    println!("{}", some_integer);
} // 这里，some_integer 移出作用域。不会有特殊操作
```

你可以尝试在 `takes_ownership` 之后，再使用 `s`，看看如何报错？例如添加一行 `println!("在move进函数后继续使用s: {}", s);`。

同样的，函数返回值也有所有权，例如：

```
fn main() {
    let s1 = gives_ownership();           // gives_ownership 将返回值
                                         // 移给 s1

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

    let s3 = takes_and_gives_back(s2);   // s2 被移动到
                                         // takes_and_gives_back 中,
                                         // 它也将返回值移给 s3
} // 这里, s3 移出作用域并被丢弃。s2 也移出作用域, 但已被移走,
  // 所以什么也不会发生。s1 移出作用域并被丢弃

fn gives_ownership() -> String {        // gives_ownership 将返回值移动给
                                         // 调用它的函数

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

    some_string                         // 返回 some_string 并移出给调用的函数
}

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

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

所有权很强大，避免了内存的不安全性，但是也带来了一个新麻烦：**总是把一个值传来传去来使用它**。传入一个函数，很可能还要从该函数传出去，结果就是语言表达变得非常啰嗦，幸运的是，Rust 提供了新功能解决这个问题。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 引用与借用

上节中提到，如果仅仅支持通过转移所有权的方式获取一个值，那会让程序变得复杂。Rust 能否像其它编程语言一样，使用某个变量的指针或者引用呢？答案是可以。

Rust 通过 借用(Borrowing) 这个概念来达成上述的目的，**获取变量的引用，称之为借用(borrowing)**。正如现实生活中，如果一个人拥有某样东西，你可以从他那里借来，当使用完毕后，也必须要物归原主。

## 引用与解引用

常规引用是一个指针类型，指向了对象存储的内存地址。在下面代码中，我们创建一个 `i32` 值的引用 `y`，然后使用解引用运算符来解出 `y` 所使用的值：

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

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

变量 `x` 存放了一个 `i32` 值 `5`。`y` 是 `x` 的一个引用。可以断言 `x` 等于 `5`。然而，如果希望对 `y` 的值做出断言，必须使用 `*y` 来解出引用所指向的值（也就是**解引用**）。一旦解引用了 `y`，就可以访问 `y` 所指向的整型值并可以与 `5` 做比较。

相反如果尝试编写 `assert_eq!(5, y);`，则会得到如下编译错误：

```
error[E0277]: can't compare `'{integer}` with `&{integer}`
--> src/main.rs:6:5
|
6 |     assert_eq!(5, y);
|     ^^^^^^^^^^^^^^^^^ no implementation for `'{integer} == &{integer}` // 无法比较整
数类型和引用类型
|
= help: the trait `std::cmp::PartialEq<&{integer}>` is not implemented for
`'{integer}'
```

不允许比较整数与引用，因为它们是不同的类型。必须使用解引用运算符解出引用所指向的值。

## 不可变引用

下面的代码，我们用 `s1` 的引用作为参数传递给 `calculate_length` 函数，而不是把 `s1` 的所有权转移给该函数：

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

    let len = calculate_length(&s1);

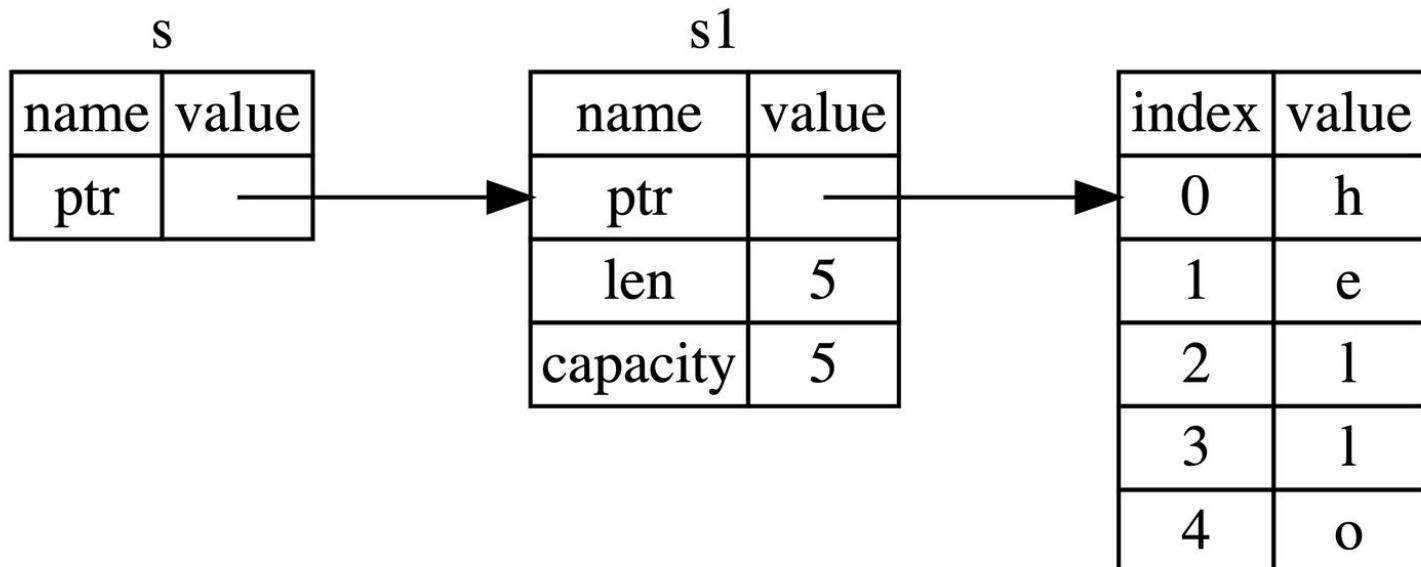
    println!("The length of '{}' is {}.", s1, len);
}

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

能注意到两点：

1. 无需像上章一样：先通过函数参数传入所有权，然后再通过函数返回来传出所有权，代码更加简洁
2. `calculate_length` 的参数 `s` 类型从 `String` 变为 `&String`

这里，`&` 符号即是引用，它们允许你使用值，但是不获取所有权，如图所示：



通过 `&s1` 语法，我们创建了一个**指向 `s1` 的引用**，但是并不拥有它。因为并不拥有这个值，当引用离开作用域后，其指向的值也不会被丢弃。

同理，函数 `calculate_length` 使用 `&` 来表明参数 `s` 的类型是一个引用：

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

人总是贪心的，可以拉女孩小手了，就想着抱抱柔软的身子（读者中的某老司机表示，这个流程完全不对），因此光借用已经满足不了我们了，如果尝试修改借用的变量呢？

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

    change(&s);
}

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

很不幸，妹子你没抱到，哦口误，你修改错了：

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
--> src/main.rs:8:5
|
7 | fn change(some_string: &String) {
|     ----- help: consider changing this to be a mutable
reference: `&mut String`
|             ----- 帮助：考虑将该参数类型修改为可变的引用：`&mut String`  

8 |     some_string.push_str(", world");
|     ^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot  

be borrowed as mutable
|             `some_string` 是一个`&`类型的引用，因此它指向的数据无法进行修改
```

正如变量默认不可变一样，引用指向的值默认也是不可变的，没事，来一起看看如何解决这个问题。

## 可变引用

只需要一个小调整，即可修复上面代码的错误：

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

    change(&mut s);
}

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

首先，声明 `s` 是可变类型，其次创建一个可变的引用 `&mut s` 和接受可变引用参数 `some_string: &mut String` 的函数。

## 可变引用同时只能存在一个

不过可变引用并不是随心所欲、想用就用的，它有一个很大的限制：**同一作用域，特定数据只能有一个可变引用：**

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

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

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

以上代码会报错：

```
error[E0499]: cannot borrow `s` as mutable more than once at a time 同一时间无法对 `s`  
进行两次可变借用  
--> src/main.rs:5:14  
|  
4 |     let r1 = &mut s;  
|         ----- first mutable borrow occurs here 首个可变引用在这里借用  
5 |     let r2 = &mut s;  
|         ^^^^^^ second mutable borrow occurs here 第二个可变引用在这里借用  
6 |  
7 |     println!("{} , {}", r1, r2);  
|             -- first borrow later used here 第一个借用在这里使用
```

这段代码出错的原因在于，第一个可变借用 `r1` 必须要持续到最后一次使用的位置 `println!`，在 `r1` 创建和最后一次使用之间，我们又尝试创建第二个可变借用 `r2`。

对于新手来说，这个特性绝对是一大拦路虎，也是新人们谈之色变的编译器 `borrow checker` 特性之一，不过各行各业都一样，限制往往是出于安全的考虑，Rust 也一样。

这种限制的好处就是使 Rust 在编译期就避免数据竞争，数据竞争可由以下行为造成：

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

数据竞争会导致未定义行为，这种行为很可能超出我们的预期，难以在运行时追踪，并且难以诊断和修复。而 Rust 避免了这种情况的发生，因为它甚至不会编译存在数据竞争的代码！

很多时候，大括号可以帮我们解决一些编译不通过的问题，通过手动限制变量的作用域：

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

{
    let r1 = &mut s;

} // r1 在这里离开了作用域，所以我们完全可以创建一个新的引用

let r2 = &mut s;
```

## 可变引用与不可变引用不能同时存在

下面的代码会导致一个错误：

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

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

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

错误如下：

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
              // 无法借用可变 `s` 因为它已经被借用了不可变
--> src/main.rs:6:14
|
4 |     let r1 = &s; // 没问题
|         -- immutable borrow occurs here 不可变借用发生在这里
5 |     let r2 = &s; // 没问题
6 |     let r3 = &mut s; // 大问题
|         ^^^^^^ mutable borrow occurs here 可变借用发生在这里
7 |
8 |     println!("{} , {} , and {}", r1, r2, r3);
|         -- immutable borrow later used here 不可变借用在这里使用
```

其实这个也很好理解，正在借用不可变引用的用户，肯定不希望他借用的东西，被另外一个人莫名其妙改变了。多个不可变借用被允许是因为没有人会去试图修改数据，每个人都只读这一份数据而不做修改，因此不用担心数据被污染。

---

注意，引用的作用域 `s` 从创建开始，一直持续到它最后一次使用的地方，这个跟变量的作用域有所不同，变量的作用域从创建持续到某一个花括号 `}`

---

Rust 的编译器一直在优化，早期的时候，引用的作用域跟变量作用域是一致的，这对日常使用带来了很大的困扰，你必须非常小心的去安排可变、不可变变量的借用，免得无法通过编译，例如以下代码：

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

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);
    // 新编译器中，r1, r2作用域在这里结束

    let r3 = &mut s;
    println!("{}", r3);
} // 老编译器中，r1、r2、r3作用域在这里结束
// 新编译器中，r3作用域在这里结束
```

在老版本的编译器中（Rust 1.31 前），将会报错，因为 `r1` 和 `r2` 的作用域在花括号 `}` 处结束，那么 `r3` 的借用就会触发 **无法同时借用可变和不可变** 的规则。

但是在新的编译器中，该代码将顺利通过，因为 **引用作用域的结束位置从花括号变成最后一次使用的位  
置**，因此 `r1` 借用和 `r2` 借用在 `println!` 后，就结束了，此时 `r3` 可以顺利借用到可变引用。

## NLL

对于这种编译器优化行为，Rust 专门起了一个名字——**Non-Lexical Lifetimes(NLL)**，专门用于找到某个引用在作用域(`}`)结束前就不再被使用的代码位置。

虽然这种借用错误有的时候会让我们很郁闷，但是你只要想想这是 Rust 提前帮你发现了潜在的 BUG，其实就开心了，虽然减慢了开发速度，但是从长期来看，大幅减少了后续开发和运维成本。

## 悬垂引用(Dangling References)

悬垂引用也叫做悬垂指针，意思为指针指向某个值后，这个值被释放掉了，而指针仍然存在，其指向的内  
存可能不存在任何值或已被其它变量重新使用。在 Rust 中编译器可以确保引用永远也不会变成悬垂状

态：当你获取数据的引用后，编译器可以确保数据不会在引用结束前被释放，要想释放数据，必须先停止其引用的使用。

让我们尝试创建一个悬垂引用，Rust 会抛出一个编译时错误：

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

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

    &s
}
```

这里是错误：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:5:16
|
5 | fn dangle() -> &String {
|             ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but there is no
value for it to be borrowed from
help: consider using the ``static`` lifetime
|
5 | fn dangle() -> &'static String {
|             ~~~~~~
```

错误信息引用了一个我们还未介绍的功能：[生命周期\(lifetimes\)](#)。不过，即使你不理解生命周期，也可以通过错误信息知道这段代码错误的关键信息：

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

该函数返回了一个借用的值，但是已经找不到它所借用值的来源

仔细看看 `dangle` 代码的每一步到底发生了什么：

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

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

    &s // 返回字符串 s 的引用
} // 这里 s 离开作用域并被丢弃。其内存被释放。
// 危险!
```

因为 `s` 是在 `dangle` 函数内创建的，当 `dangle` 的代码执行完毕后，`s` 将被释放，但是此时我们又尝试去返回它的引用。这意味着这个引用会指向一个无效的 `String`，这可不对！

其中一个很好的解决方法是直接返回 `String`：

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

这样就没有任何错误了，最终 `String` 的 **所有权被转移给外面的调用者**。

## 借用规则总结

总的来说，借用规则如下：

- 同一时刻，你只能拥有要么一个可变引用，要么任意多个不可变引用
- 引用必须总是有效的

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 复合类型

行百里者半九十，欢迎大家来到这里，虽然还不到中点，但是已经不远了。如果说之前学的基础数据类型是原子，那么本章将讲的数据类型可以认为是分子。

本章的重点在复合类型上，顾名思义，复合类型是由其它类型组合而成的，最典型的就是结构体 `struct` 和枚举 `enum`。例如平面上的一个点 `point(x, y)`，它由两个数值类型的值 `x` 和 `y` 组合而来。我们无法单独去维护这两个数值，因为单独一个 `x` 或者 `y` 是含义不完整的，无法标识平面上的一个点，应该把它们看作一个整体去理解和处理。

来看一段代码，它使用我们之前学过的内容来构建文件操作：

```
#![allow(unused_variables)]
type File = String;

fn open(f: &mut File) -> bool {
    true
}
fn close(f: &mut File) -> bool {
    true
}

#[allow(dead_code)]
fn read(f: &mut File, save_to: &mut Vec<u8>) -> ! {
    unimplemented!()
}

fn main() {
    let mut f1 = File::from("f1.txt");
    open(&mut f1);
    //read(&mut f1, &mut vec![]);
    close(&mut f1);
}
```

接下来我们的学习非常类似原型设计：有的方法只提供 API 接口，但是不提供具体实现。此外，有的变量在声明之后并未使用，因此在这个阶段我们需要排除一些编译器噪音（Rust 在编译的时候会扫描代码，变量声明后未使用会以 `warning` 警告的形式进行提示），引入 `#![allow(unused_variables)]` 属性标记，该标记会告诉编译器忽略未使用的变量，不要抛出 `warning` 警告，具体的常见编译器属性你可以在这里查阅：[编译器属性标记](#)。

`read` 函数也非常有趣，它返回一个 `!` 类型，这个表明该函数是一个发散函数，不会返回任何值，包括 `()`。`unimplemented!()` 告诉编译器该函数尚未实现，`unimplemented!()` 标记通常意味着我们期望快速完成主要代码，回头再通过搜索这些标记来完成次要代码，类似的标记还有 `todo!()`，当代码执行到

这种未实现的地方时，程序会直接报错。你可以反注释 `read(&mut f1, &mut vec![]);` 这行，然后再观察下结果。

同时，从代码设计角度来看，关于文件操作的类型和函数应该组织在一起，散落得到处都是，是难以管理和使用的。而且通过 `open(&mut f1)` 进行调用，也远没有使用 `f1.open()` 来调用好，这就体现出了只使用基本类型的局限性：**无法从更高的抽象层次去简化代码。**

接下来，我们将引入一个高级数据结构——结构体 `struct`，来看看复合类型是怎样更好的解决这类问题。开始之前，先来看看 Rust 的重点也是难点：字符串 `String` 和 `&str`。

# 字符串

在其他语言中，字符串往往是送分题，因为实在是太简单了，例如 "hello, world" 就是字符串章节的几乎全部内容了，但是如果你带着同样的想法来学 Rust，我保证，绝对会栽跟头，**因此这一章大家一定要重视，仔细阅读，这里有很多其它 Rust 书籍中没有的内容。**

首先来看段很简单的代码：

```
fn main() {
    let my_name = "Pascal";
    greet(my_name);
}

fn greet(name: String) {
    println!("Hello, {}!", name);
}
```

`greet` 函数接受一个字符串类型的 `name` 参数，然后打印到终端控制台中，非常好理解，你们猜猜，这段代码能否通过编译？

```
error[E0308]: mismatched types
--> src/main.rs:3:11
|
3 |     greet(my_name);
|     ^^^^^^^^
|     |
|     expected struct `std::string::String`, found `&str`
|     help: try using a conversion method: `my_name.to_string()`

error: aborting due to previous error
```

Bingo，果然报错了，编译器提示 `greet` 函数需要一个 `String` 类型的字符串，却传入了一个 `&str` 类型的字符串，相信读者心中现在一定有几头草泥马呼啸而过，怎么字符串也能整出这么多花活？

在讲解字符串之前，先来看看什么是切片？

## 切片(slice)

切片并不是 Rust 独有的概念，在 Go 语言中就非常流行，它允许你引用集合中部分连续的元素序列，而不是引用整个集合。

对于字符串而言，切片就是对 `String` 类型中某一部分的引用，它看起来像这样：

```

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

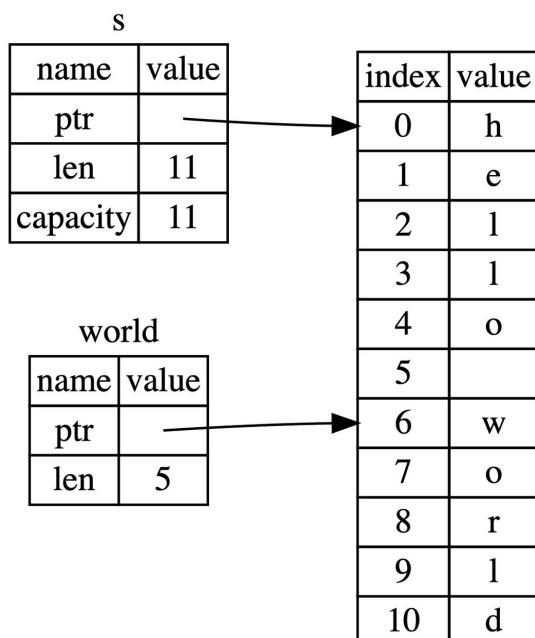
let hello = &s[0..5];
let world = &s[6..11];

```

`hello` 没有引用整个 `String s`，而是引用了 `s` 的一部分内容，通过 `[0..5]` 的方式来指定。

这就是创建切片的语法，使用方括号包括的一个序列：[开始索引..终止索引]，其中开始索引是切片中第一个元素的索引位置，而终止索引是最后一个元素后面的索引位置，也就是这是一个右半开区间。在切片数据结构内部会保存开始的位置和切片的长度，其中长度是通过 终止索引 - 开始索引 的方式计算得来的。

对于 `let world = &s[6..11];` 来说，`world` 是一个切片，该切片的指针指向 `s` 的第 7 个字节(索引从 0 开始, 6 是第 7 个字节)，且该切片的长度是 5 个字节。



在使用 Rust 的 .. range 序列语法时，如果你想从索引 0 开始，可以使用如下的方式，这两个是等效的：

```

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

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

```

同样的，如果你的切片想要包含 `String` 的最后一个字节，则可以这样使用：

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

let len = s.len();

let slice = &s[4..len];
let slice = &s[4..];
```

你也可以截取完整的 `String` 切片：

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

let len = s.len();

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

---

在对字符串使用切片语法时需要格外小心，切片的索引必须落在字符之间的边界位置，也就是 UTF-8 字符的边界，例如中文在 UTF-8 中占用三个字节，下面的代码就会崩溃：

```
let s = "中国人";
let a = &s[0..2];
println!("{}", a);
```

因为我们只取 `s` 字符串的前两个字节，但是本例中每个汉字占用三个字节，因此没有落在边界处，也就是连 `中` 字都取不完整，此时程序会直接崩溃退出，如果改成 `&s[0..3]`，则可以正常通过编译。因此，当你需要对字符串做切片索引操作时，需要格外小心这一点，关于该如何操作 UTF-8 字符串，参见[这里](#)。

---

字符串切片的类型标识是 `&str`，因此我们可以这样声明一个函数，输入 `String` 类型，返回它的切片：  
`fn first_word(s: &String) -> &str`。

有了切片就可以写出这样的代码：

```

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

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
fn first_word(s: &String) -> &str {
    &s[..1]
}

```

编译器报错如下：

```

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

```

回忆一下借用的规则：当我们已经有了可变借用时，就无法再拥有不可变的借用。因为 `clear` 需要清空改变 `String`，因此它需要一个可变借用（利用 VSCode 可以看到该方法的声明是 `pub fn clear(&mut self)`，参数是对自身的可变借用）；而之后的 `println!` 又使用了不可变借用，也就是在 `s.clear()` 处可变借用与不可变借用试图同时生效，因此编译无法通过。

从上述代码可以看出，Rust 不仅让我们的 API 更加容易使用，而且也在编译期就消除了大量错误！

## 其它切片

因为切片是对集合的部分引用，因此不仅仅字符串有切片，其它集合类型也有，例如数组：

```

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

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

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

```

该数组切片的类型是 `&[i32]`，数组切片和字符串切片的工作方式是一样的，例如持有一个引用指向原始数组的某个元素和长度。

## 字符串字面量是切片

之前提到过字符串字面量，但是没有提到它的类型：

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

实际上，`s` 的类型是 `&str`，因此你也可以这样声明：

```
let s: &str = "Hello, world!";
```

该切片指向了程序可执行文件中的某个点，这也是为什么字符串字面量是不可变的，因为 `&str` 是一个不可变引用。

了解完切片，可以进入本节的正题了。

## 什么是字符串？

顾名思义，字符串是由字符组成的连续集合，但是在上一节中我们提到过，Rust 中的字符是 Unicode 类型，因此每个字符占据 4 个字节内存空间，但是在字符串中不一样，字符串是 UTF-8 编码，也就是字符串中的字符所占的字节数是变化的(1 - 4)，这样有助于大幅降低字符串所占用的内存空间。

Rust 在语言级别，只有一种字符串类型：`str`，它通常是以引用类型出现 `&str`，也就是上文提到的字符串切片。虽然语言级别只有上述的 `str` 类型，但是在标准库里，还有多种不同用途的字符串类型，其中使用最广的即是 `String` 类型。

`str` 类型是硬编码进可执行文件，也无法被修改，但是 `String` 则是一个可增长、可改变且具有所有权的 UTF-8 编码字符串，当 Rust 用户提到字符串时，往往指的就是 `String` 类型和 `&str` 字符串切片类型，这两个类型都是 UTF-8 编码。

除了 `String` 类型的字符串，Rust 的标准库还提供了其他类型的字符串，例如 `OsString`，`OsStr`，`CsString` 和 `CsStr` 等，注意到这些名字都以 `String` 或者 `Str` 结尾了吗？它们分别对应的是具有所有权和被借用的变量。

## String 与 `&str` 的转换

在之前的代码中，已经见到好几种从 `&str` 类型生成 `String` 类型的操作：

- `String::from("hello,world")`

- "hello,world".to\_string()

那么如何将 `String` 类型转为 `&str` 类型呢？答案很简单，取引用即可：

```
fn main() {
    let s = String::from("hello,world!");
    say_hello(&s);
    say_hello(&s[..]);
    say_hello(s.as_str());
}

fn say_hello(s: &str) {
    println!("{}", s);
}
```

实际上这种灵活用法是因为 `deref` 隐式强制转换，具体我们会在 [Deref 特征](#) 进行详细讲解。

## 字符串索引

在其它语言中，使用索引的方式访问字符串的某个字符或者子串是很正常的行为，但是在 Rust 中就会报错：

```
let s1 = String::from("hello");
let h = s1[0];
```

该代码会产生如下错误：

```
3 |     let h = s1[0];
|          ^^^^^ `String` cannot be indexed by `{integer}`
|
= help: the trait `Index<{integer}>` is not implemented for `String`
```

## 深入字符串内部

字符串的底层的数据存储格式实际上是 `[ u8 ]`，一个字节数组。对于 `let hello = String::from("Hola");` 这行代码来说，`Hola` 的长度是 4 个字节，因为 `"Hola"` 中的每个字母在 UTF-8 编码中仅占用 1 个字节，但是对于下面的代码呢？

```
let hello = String::from("中国人");
```

如果问你该字符串多长，你可能会说 3，但是实际上是 9 个字节的长度，因为大部分常用汉字在 UTF-8 中的长度是 3 个字节，因此这种情况下对 `hello` 进行索引，访问 `&hello[0]` 没有任何意义，因为你取不到 中 这个字符，而是取到了这个字符三个字节中的第一个字节，这是一个非常奇怪而且难以理解的返回值。

## 字符串的不同表现形式

现在看一下用梵文写的字符串 “नमस्ते”，它底层的字节数组如下形式：

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]
```

长度是 18 个字节，这也是计算机最终存储该字符串的形式。如果从字符的形式去看，则是：

```
['न', 'म', 'स', '्', 'त', 'े']
```

但是这种形式下，第四和六两个字母根本就不存在，没有任何意义，接着再从字母串的形式去看：

```
["न", "म", "स", "ते"]
```

所以，可以看出来 Rust 提供了不同的字符串展现方式，这样程序可以挑选自己想要的方式去使用，而无需去管字符串从人类语言角度看长什么样。

还有一个原因导致了 Rust 不允许去索引字符串：因为索引操作，我们总是期望它的性能表现是 O(1)，然而对于 `String` 类型来说，无法保证这一点，因为 Rust 可能需要从 0 开始去遍历字符串来定位合法的字符。

## 字符串切片

前文提到过，字符串切片是非常危险的操作，因为切片的索引是通过字节来进行，但是字符串又是 UTF-8 编码，因此你无法保证索引的字节刚好落在字符的边界上，例如：

```
let hello = "中国人";  
let s = &hello[0..2];
```

运行上面的程序，会直接造成崩溃：

```
thread 'main' panicked at 'byte index 2 is not a char boundary; it is inside '中'  
(bytes 0..3) of `中国人`', src/main.rs:4:14  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

这里提示的很清楚，我们索引的字节落在了 中 字符的内部，这种返回没有任何意义。

因此在通过索引区间来访问字符串时，**需要格外的小心**，一不注意，就会导致你程序的崩溃！

## 操作字符串

由于 `String` 是可变字符串，下面介绍 Rust 字符串的修改，添加，删除等常用方法：

### 追加 (Push)

在字符串尾部可以使用 `push()` 方法追加字符 `char`，也可以使用 `push_str()` 方法追加字符串字面量。这两个方法都是**在原有的字符串上追加，并不会返回新的字符串**。由于字符串追加操作要修改原来的字符串，则该字符串必须是可变的，即**字符串变量必须由 mut 关键字修饰**。

示例代码如下：

```
fn main() {  
    let mut s = String::from("Hello ");  
  
    s.push_str("rust");  
    println!("追加字符串 push_str() -> {}", s);  
  
    s.push('!');  
    println!("追加字符 push() -> {}", s);  
}
```

代码运行结果：

```
追加字符串 push_str() -> Hello rust  
追加字符 push() -> Hello rust!
```

### 插入 (Insert)

可以使用 `insert()` 方法插入单个字符 `char`，也可以使用 `insert_str()` 方法插入字符串字面量，与 `push()` 方法不同，这两方法需要传入两个参数，第一个参数是字符（串）插入位置的索引，第二个参数

是要插入的字符（串），索引从 0 开始计数，如果越界则会发生错误。由于字符串插入操作要修改原来的字符串，则该字符串必须是可变的，即字符串变量必须由 `mut` 关键字修饰。

示例代码如下：

```
fn main() {
    let mut s = String::from("Hello rust!");
    s.insert(5, ',');
    println!("插入字符 insert() -> {}", s);
    s.insert_str(6, " I like");
    println!("插入字符串 insert_str() -> {}", s);
}
```

代码运行结果：

```
插入字符 insert() -> Hello, rust!
插入字符串 insert_str() -> Hello, I like rust!
```

## 替换 (Replace)

如果想要把字符串中的某个字符串替换成其它的字符串，那可以使用 `replace()` 方法。与替换有关的方法有三个。

1、`replace`

该方法可适用于 `String` 和 `&str` 类型。`replace()` 方法接收两个参数，第一个参数是要被替换的字符串，第二个参数是新的字符串。该方法会替换所有匹配到的字符串。**该方法是返回一个新的字符串，而不是操作原来的字符串。**

示例代码如下：

```
fn main() {
    let string_replace = String::from("I like rust. Learning rust is my favorite!");
    let new_string_replace = string_replace.replace("rust", "RUST");
    dbg!(new_string_replace);
}
```

代码运行结果：

```
new_string_replace = "I like RUST. Learning RUST is my favorite!"
```

2、`replacen`

该方法可适用于 `String` 和 `&str` 类型。`replacen()` 方法接收三个参数，前两个参数与 `replace()` 方法一样，第三个参数则表示替换的个数。该方法是返回一个新的字符串，而不是操作原来的字符串。

示例代码如下：

```
fn main() {
    let string_replace = "I like rust. Learning rust is my favorite!";
    let new_string_replacen = string_replace.replacen("rust", "RUST", 1);
    dbg!(new_string_replacen);
}
```

代码运行结果：

```
new_string_replacen = "I like RUST. Learning rust is my favorite!"
```

### 3、`replace_range`

该方法仅适用于 `String` 类型。`replace_range` 接收两个参数，第一个参数是要替换字符串的范围 (Range)，第二个参数是新的字符串。该方法是直接操作原来的字符串，不会返回新的字符串。该方法需要使用 `mut` 关键字修饰。

示例代码如下：

```
fn main() {
    let mut string_replace_range = String::from("I like rust!");
    string_replace_range.replace_range(7..8, "R");
    dbg!(string_replace_range);
}
```

代码运行结果：

```
string_replace_range = "I like Rust!"
```

## 删除 (Delete)

与字符串删除相关的方法有 4 个，他们分别是 `pop()`，`remove()`，`truncate()`，`clear()`。这四个方法仅适用于 `String` 类型。

### 1、`pop` —— 删除并返回字符串的最后一个字符

该方法是直接操作原来的字符串。但是存在返回值，其返回值是一个 `Option` 类型，如果字符串为空，则返回 `None`。示例代码如下：

```
fn main() {
    let mut string_pop = String::from("rust pop 中文!");
    let p1 = string_pop.pop();
    let p2 = string_pop.pop();
    dbg!(p1);
    dbg!(p2);
    dbg!(string_pop);
}
```

代码运行结果：

```
p1 = Some(
    '!',
)
p2 = Some(
    '文',
)
string_pop = "rust pop 中"
```

## 2、remove —— 删除并返回字符串中指定位置的字符

**该方法是直接操作原来的字符串。**但是存在返回值，其返回值是删除位置的字符串，只接收一个参数，表示该字符起始索引位置。`remove()` 方法是按照字节来处理字符串的，如果参数所给的位置不是合法的字符边界，则会发生错误。

示例代码如下：

```
fn main() {
    let mut string_remove = String::from("测试remove方法");
    println!(
        "string_remove 占 {} 个字节",
        std::mem::size_of_val(string_remove.as_str())
    );
    // 删除第一个汉字
    string_remove.remove(0);
    // 下面代码会发生错误
    // string_remove.remove(1);
    // 直接删除第二个汉字
    // string_remove.remove(3);
    dbg!(string_remove);
}
```

代码运行结果：

```
string_remove 占 18 个字节
string_remove = "试remove方法"
```

### 3、 truncate —— 删除字符串中从指定位置开始到结尾的全部字符

**该方法是直接操作原来的字符串。**无返回值。该方法 `truncate()` 方法是按照字节来处理字符串的，如果参数所给的位置不是合法的字符边界，则会发生错误。

示例代码如下：

```
fn main() {
    let mut string_truncate = String::from("测试truncate");
    string_truncate.truncate(3);
    dbg!(string_truncate);
}
```

代码运行结果：

```
string_truncate = "测"
```

### 4、 clear —— 清空字符串

**该方法是直接操作原来的字符串。**调用后，删除字符串中的所有字符，相当于 `truncate()` 方法参数为 0 的时候。

示例代码如下：

```
fn main() {
    let mut string_clear = String::from("string clear");
    string_clear.clear();
    dbg!(string_clear);
}
```

代码运行结果：

```
string_clear = ""
```

## 连接 (Concatenate)

### 1、 使用 + 或者 += 连接字符串

使用 + 或者 += 连接字符串，要求右边的参数必须为字符串的切片引用 (Slice) 类型。其实当调用 + 的操作符时，相当于调用了 `std::string` 标准库中的 `add()` 方法，这里 `add()` 方法的第二个参数是一个引用的类型。因此我们在使用 + 时，必须传递切片引用类型。不能直接传递 `String` 类型。+ 是返回一个新的字符串，所以变量声明可以不需要 `mut` 关键字修饰。

示例代码如下：

```
fn main() {
    let string_append = String::from("hello ");
    let string_rust = String::from("rust");
    // &string_rust会自动解引用为&str
    let result = string_append + &string_rust;
    let mut result = result + "!"; // `result + "!"` 中的 `result` 是不可变的
    result += "!!!";

    println!("连接字符串 + -> {}", result);
}
```

代码运行结果：

```
连接字符串 + -> hello rust!!!!
```

add() 方法的定义：

```
fn add(self, s: &str) -> String
```

因为该方法涉及到更复杂的特征功能，因此我们这里简单说明下：

```
fn main() {
    let s1 = String::from("hello,");
    let s2 = String::from("world!");
    // 在下句中，s1的所有权被转移走了，因此后面不能再使用s1
    let s3 = s1 + &s2;
    assert_eq!(s3, "hello,world!");
    // 下面的语句如果去掉注释，就会报错
    // println!("{}",s1);
}
```

self 是 String 类型的字符串 s1，该函数说明，只能将 &str 类型的字符串切片添加到 String 类型的 s1 上，然后返回一个新的 String 类型，所以 let s3 = s1 + &s2；就很好解释了，将 String 类型的 s1 与 &str 类型的 s2 进行相加，最终得到 string 类型的 s3。

由此可推，以下代码也是合法的：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

// String = String + &str + &str + &str + &str
let s = s1 + "-" + &s2 + "-" + &s3;
```

`String + &str` 返回一个 `String`，然后再继续跟一个 `&str` 进行 `+` 操作，返回一个 `String` 类型，不断循环，最终生成一个 `s`，也是 `String` 类型。

`s1` 这个变量通过调用 `add()` 方法后，所有权被转移到 `add()` 方法里面，`add()` 方法调用后就被释放了，同时 `s1` 也被释放了。再使用 `s1` 就会发生错误。这里涉及到[所有权转移（Move）](#)的相关知识。

## 2、使用 `format!` 连接字符串

`format!` 这种方式适用于 `String` 和 `&str`。`format!` 的用法与 `print!` 的用法类似，详见[格式化输出](#)。

示例代码如下：

```
fn main() {
    let s1 = "hello";
    let s2 = String::from("rust");
    let s = format!("{} {}", s1, s2);
    println!("{}", s);
}
```

代码运行结果：

```
hello rust!
```

## 字符串转义

我们可以通过转义的方式 \ 输出 ASCII 和 Unicode 字符。

```

fn main() {
    // 通过 \ + 字符的十六进制表示，转义输出一个字符
    let byte_escape = "I'm writing \x52\x75\x73\x74!";
    println!("What are you doing\x3F (\\"x3F means ?) {}", byte_escape);

    // \u 可以输出一个 unicode 字符
    let unicode_codepoint = "\u{211D}";
    let character_name = "\"DOUBLE-STRUCK CAPITAL R\"";

    println!(
        "Unicode character {} (U+211D) is called {}",
        unicode_codepoint, character_name
    );

    // 换行了也会保持之前的字符串格式
    // 使用\忽略换行符
    let long_string = "String literals
                      can span multiple lines.
                      The linebreak and indentation here ->\
                      <- can be escaped too!";
    println!("{}", long_string);
}

```

当然，在某些情况下，可能你会希望保持字符串的原样，不要转义：

```

fn main() {
    println!("{}", "hello \x52\x75\x73\x74");
    let raw_str = r"Escapes don't work here: \x3F \u{211D}";
    println!("{}", raw_str);

    // 如果字符串包含双引号，可以在开头和结尾加 #
    let quotes = r#"And then I said: "There is no escape!"#";
    println!("{}", quotes);

    // 如果还是有歧义，可以继续增加，没有限制
    let longer_delimiter = r###"A string with "#" in it. And even "##!"###;
    println!("{}", longer_delimiter);
}

```

## 操作 UTF-8 字符串

前文提到了几种使用 UTF-8 字符串的方式，下面来一一说明。

## 字符

如果你想要以 Unicode 字符的方式遍历字符串，最好的办法是使用 `chars` 方法，例如：

```
for c in "中国人".chars() {  
    println!("{}" , c);  
}
```

输出如下

```
中  
国  
人
```

## 字节

这种方式是返回字符串的底层字节数组表现形式：

```
for b in "中国人".bytes() {  
    println!("{}" , b);  
}
```

输出如下：

```
228  
184  
173  
229  
155  
189  
228  
186  
186
```

## 获取子串

想要准确的从 UTF-8 字符串中获取子串是较为复杂的事情，例如想要从 `holla中国人नमस्ते` 这种变长的字符串中取出某一个子串，使用标准库你是做不到的。你需要在 `crates.io` 上搜索 `utf8` 来寻找想要的功能。

可以考虑尝试下这个库：[utf8\\_slice](#)。

## 字符串深度剖析

那么问题来了，为啥 `String` 可变，而字符串字面值 `str` 却不可以？

就字符串字面值来说，我们在编译时就知道其内容，最终字面值文本被直接硬编码进可执行文件中，这使得字符串字面值快速且高效，这主要得益于字符串字面值的不可变性。不幸的是，我们不能为了获得这种性能，而把每一个在编译时大小未知的文本都放进内存中（你也做不到！），因为有的字符串是在程序运行得过程中动态生成的。

对于 `String` 类型，为了支持一个可变、可增长的文本片段，需要在堆上分配一块在编译时未知大小的内存来存放内容，这些都是在程序运行时完成的：

- 首先向操作系统请求内存来存放 `String` 对象
- 在使用完成后，将内存释放，归还给操作系统

其中第一部分由 `String::from` 完成，它创建了一个全新的 `String`。

重点来了，到了第二部分，就是百家齐放的环节，在有**垃圾回收 GC** 的语言中，GC 来负责标记并清除这些不再使用的内存对象，这个过程都是自动完成，无需开发者关心，非常简单好用；但是在无 GC 的语言中，需要开发者手动去释放这些内存对象，就像创建对象需要通过编写代码来完成一样，未能正确释放对象造成的后果简直不可估量。

对于 Rust 而言，安全和性能是写到骨子里的核心特性，如果使用 GC，那么会牺牲性能；如果使用手动管理内存，那么会牺牲安全，这该怎么办？为此，Rust 的开发者想出了一个无比惊艳的办法：变量在离开作用域后，就自动释放其占用的内存：

```
{  
    let s = String::from("hello"); // 从此处起，s 是有效的  
  
    // 使用 s  
}  
                                // 此作用域已结束，  
                                // s 不再有效，内存被释放
```

与其它系统编程语言的 `free` 函数相同，Rust 也提供了一个释放内存的函数：`drop`，但是不同的是，其它语言要手动调用 `free` 来释放每一个变量占用的内存，而 Rust 则在变量离开作用域时，自动调用 `drop` 函数：上面代码中，Rust 在结尾的 `}` 处自动调用 `drop`。

---

其实，在 C++ 中，也有这种概念：*Resource Acquisition Is Initialization (RAII)*。如果你使用过 RAII 模式的话应该对 Rust 的 `drop` 函数并不陌生。

---

这个模式对编写 Rust 代码的方式有着深远的影响，在后面章节我们会进行更深入的介绍。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

- 字符串
    - 习题解答
  - 切片
    - 习题解答
  - String
    - 习题解答
- 

## 引用资料

1. <https://blog.csdn.net/a1595901624/article/details/119294443>

# 元组

元组是由多种类型组合到一起形成的，因此它是复合类型，元组的长度是固定的，元组中元素的顺序也是固定的。

可以通过以下语法创建一个元组：

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

变量 `tup` 被绑定了一个元组值 `(500, 6.4, 1)`，该元组的类型是 `(i32, f64, u8)`，看到没？元组是用括号将多个类型组合到一起，简单吧？

可以使用模式匹配或者 `.` 操作符来获取元组中的值。

## 用模式匹配解构元组

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

    let (x, y, z) = tup;

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

上述代码首先创建一个元组，然后将其绑定到 `tup` 上，接着使用 `let (x, y, z) = tup;` 来完成一次模式匹配，因为元组是 `(n1, n2, n3)` 形式的，因此我们用一模一样的 `(x, y, z)` 形式来进行匹配，元组中对应的值会绑定到变量 `x, y, z` 上。这就是解构：用同样的形式把一个复杂对象中的值匹配出来。

## 用 `.` 来访问元组

模式匹配可以让我们一次性把元组中的值全部或者部分获取出来，如果只想要访问某个特定元素，那模式匹配就略显繁琐，对此，Rust 提供了 `.` 的访问方式：

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

    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
}
```

和其它语言的数组、字符串一样，元组的索引从 0 开始。

## 元组的使用示例

元组在函数返回值场景很常用，例如下面的代码，可以使用元组返回多个值：

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

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

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

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

    (s, length)
}
```

`calculate_length` 函数接收 `s1` 字符串的所有权，然后计算字符串的长度，接着把字符串所有权和字符串长度再返回给 `s2` 和 `len` 变量。

在其他语言中，可以用结构体来声明一个三维空间中的点，例如 `Point(10, 20, 30)`，虽然使用 Rust 元组也可以做到：`(10, 20, 30)`，但是这样写有个非常重大的缺陷：

**不具备任何清晰的含义**，在下一章节中，会提到一种与元组类似的结构体，元组结构体，可以解决这个问题。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 结构体

上一节中提到需要一个更高级的数据结构来帮助我们更好的抽象问题，结构体 `struct` 恰恰就是这样的复合数据结构，它是由其它数据类型组合而来。其它语言也有类似的数据结构，不过可能有不同的名称，例如 `object`、`record` 等。

结构体跟之前讲过的[元组](#)有些相像：都是由多种类型组合而成。但是与元组不同的是，结构体可以为内部的每个字段起一个富有含义的名称。因此结构体更加灵活更加强大，你无需依赖这些字段的顺序来访问和解析它们。

## 结构体语法

天下无敌的剑士往往也因为他有一柄无双之剑，既然结构体这么强大，那么我们就需要给它配套一套强大的语法，让用户能更好的驾驭。

### 定义结构体

一个结构体由几部分组成：

- 通过关键字 `struct` 定义
- 一个清晰明确的结构体 名称
- 几个有名字的结构体 字段

例如，以下结构体定义了某网站的用户：

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

该结构体名称是 `User`，拥有 4 个字段，且每个字段都有对应的字段名及类型声明，例如 `username` 代表了用户名，是一个可变的 `String` 类型。

### 创建结构体实例

为了使用上述结构体，我们需要创建 `User` 结构体的**实例**：

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

有几点值得注意:

1. 初始化实例时，**每个字段**都需要进行初始化
2. 初始化时的字段顺序**不需要**和结构体定义时的顺序一致

## 访问结构体字段

通过 . 操作符即可访问结构体实例内部的字段值，也可以修改它们:

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

需要注意的是，必须要将结构体实例声明为可变的，才能修改其中的字段，Rust 不支持将某个结构体某个字段标记为可变。

## 简化结构体创建

下面的函数类似一个构建函数，返回了 User 结构体的实例:

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

它接收两个字符串参数: email 和 username，然后使用它们来创建一个 User 结构体，并且返回。可以注意到这两行: email: email 和 username: username，非常的扎眼，因为实在有些啰嗦，如果你从 TypeScript 过来，肯定会鄙视 Rust 一番，不过好在，它也不是无可救药:

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

如上所示，当函数参数和结构体字段同名时，可以直接使用缩略的方式进行初始化，跟 TypeScript 中一模一样。

## 结构体更新语法

在实际场景中，有一种情况很常见：根据已有的结构体实例，创建新的结构体实例，例如根据已有的 `user1` 实例来构建 `user2`：

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

老话重提，如果你从 TypeScript 过来，肯定觉得啰嗦爆了：竟然手动把 `user1` 的三个字段逐个赋值给 `user2`，好在 Rust 为我们提供了 结构体更新语法：

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

因为 `user2` 仅仅在 `email` 上与 `user1` 不同，因此我们只需要对 `email` 进行赋值，剩下的通过结构体更新语法 `..user1` 即可完成。

`..` 语法表明凡是是我们没有显式声明的字段，全部从 `user1` 中自动获取。需要注意的是 `..user1` 必须在结构体的尾部使用。

---

结构体更新语法跟赋值语句 `=` 非常相像，因此在上面代码中，`user1` 的部分字段所有权被转移到 `user2` 中：`username` 字段发生了所有权转移，作为结果，`user1` 无法再被使用。

聪明的读者肯定要发问了：明明有三个字段进行了自动赋值，为何只有 `username` 发生了所有权转移？

仔细回想一下[所有权](#)那一节的内容，我们提到了 Copy 特征：实现了 Copy 特征的类型无需所有权转移，可以直接在赋值时进行数据拷贝，其中 bool 和 u64 类型就实现了 Copy 特征，因此 active 和 sign\_in\_count 字段在赋值给 user2 时，仅仅发生了拷贝，而不是所有权转移。

值得注意的是：username 所有权被转移给了 user2，导致了 user1 无法再被使用，但是并不代表 user1 内部的其它字段不能被继续使用，例如：

---

```
let user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};  
let user2 = User {  
    active: user1.active,  
    username: user1.username,  
    email: String::from("another@example.com"),  
    sign_in_count: user1.sign_in_count,  
};  
println!("{}" , user1.active);  
// 下面这行会报错  
println!("{:?}" , user1);
```

## 结构体的内存排列

先来看以下代码：

```

#[derive(Debug)]
struct File {
    name: String,
    data: Vec<u8>,
}

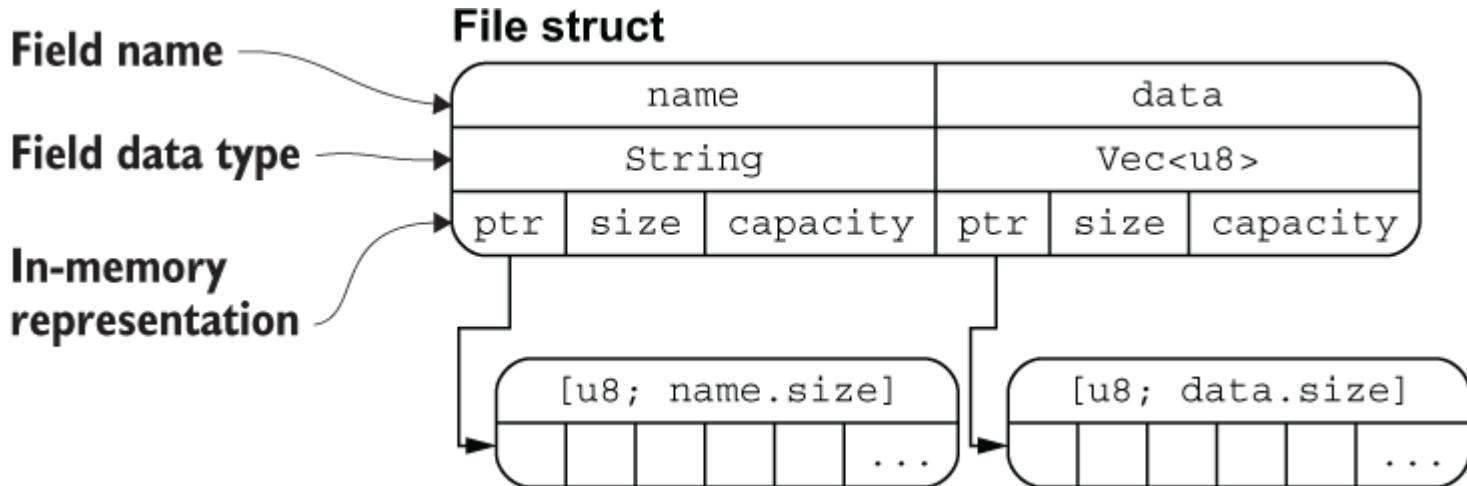
fn main() {
    let f1 = File {
        name: String::from("f1.txt"),
        data: Vec::new(),
    };

    let f1_name = &f1.name;
    let f1_length = &f1.data.len();

    println!("{}: {:?}", f1);
    println!("{} is {} bytes long", f1_name, f1_length);
}

```

上面定义的 `File` 结构体在内存中的排列如下图所示：



从图中可以清晰地看出 `File` 结构体两个字段 `name` 和 `data` 分别拥有底层两个 `[u8]` 数组的所有权 (`String` 类型的底层也是 `[u8]` 数组), 通过 `ptr` 指针指向底层数组的内存地址, 这里你可以把 `ptr` 指针理解为 Rust 中的引用类型。

该图片也侧面印证了：把结构体中具有所有权的字段转移出去后，将无法再访问该字段，但是可以正常访问其它的字段。

## 元组结构体(Tuple Struct)

结构体必须要有名称，但是结构体的字段可以没有名称，这种结构体长得很像元组，因此被称为元组结构体，例如：

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

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

元组结构体在你希望有一个整体名称，但是又不关心里面字段的名称时将非常有用。例如上面的 `Point` 元组结构体，众所周知 3D 点是  $(x, y, z)$  形式的坐标点，因此我们无需再为内部的字段逐一命名为：  
`x, y, z`。

## 单元结构体(Unit-like Struct)

还记得之前讲过的基本没啥用的[单元类型](#)吧？单元结构体就跟它很像，没有任何字段和属性，但是好在，它还挺有用。

如果你定义一个类型，但是不关心该类型的内容，只关心它的行为时，就可以使用 单元结构体：

```
struct AlwaysEqual;

let subject = AlwaysEqual;

// 我们不关心 AlwaysEqual 的字段数据，只关心它的行为，因此将它声明为单元结构体，然后再为它实现某个
// 特征
impl SomeTrait for AlwaysEqual {
}
```

## 结构体数据的所有权

在之前的 `User` 结构体的定义中，有一处细节：我们使用了自身拥有所有权的 `String` 类型而不是基于引用的 `&str` 字符串切片类型。这是一个有意而为之的选择：因为我们想要这个结构体拥有它所有的数据，而不是从其它地方借用数据。

你也可以让 `User` 结构体从其它对象借用数据，不过这么做，就需要引入[生命周期\(lifetimes\)](#)这个新概念（也是一个复杂的概念），简而言之，生命周期能确保结构体的作用范围要比它所借用的数据的作用范围要小。

总之，如果你想在结构体中使用一个引用，就必须加上生命周期，否则就会报错：

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

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

编译器会抱怨它需要生命周期标识符：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:2:15
|
2 |     username: &str,
|             ^ expected named lifetime parameter // 需要一个生命周期
|
help: consider introducing a named lifetime parameter // 考虑像下面的代码这样引入一个生命周期
|
1 ~ struct User<'a> {
2 ~     username: &'a str,
| 

error[E0106]: missing lifetime specifier
--> src/main.rs:3:12
|
3 |     email: &str,
|             ^ expected named lifetime parameter
|
help: consider introducing a named lifetime parameter
|
1 ~ struct User<'a> {
2 |     username: &str,
3 ~     email: &'a str,
| 
```

未来在[生命周期](#)中会讲到如何修复这个问题以便在结构体中存储引用，不过在那之前，我们会避免在结构体中使用引用类型。

## 使用 #[derive(Debug)] 来打印结构体的信息

在前面的代码中我们使用 `#[derive(Debug)]` 对结构体进行了标记，这样才能使用 `println!("{}: {}")` 的方式对其进行打印输出，如果不加，看看会发生什么：

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
fn main() {  
    let rect1 = Rectangle {  
        width: 30,  
        height: 50,  
    };  
  
    println!("rect1 is {}", rect1);  
}
```

首先可以观察到，上面使用了 `{}` 而不是之前的 `{: ?}`，运行后报错：

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

提示我们结构体 `Rectangle` 没有实现 `Display` 特征，这是因为如果我们使用 `{}` 来格式化输出，那对应的类型就必须实现 `Display` 特征，以前学习的基本类型，都默认实现了该特征：

```
fn main() {  
    let v = 1;  
    let b = true;  
  
    println!("{} , {}", v, b);  
}
```

上面代码不会报错，那么结构体为什么不默认实现 `Display` 特征呢？原因在于结构体较为复杂，例如考虑以下问题：你想要逗号对字段进行分割吗？需要括号吗？加在什么地方？所有的字段都应该显示？类似的还有很多，由于这种复杂性，Rust 不希望猜测我们想要的是什么，而是把选择权交给我们自己来实现：如果要用 `{}` 的方式打印结构体，那就自己实现 `Display` 特征。

接下来继续阅读报错：

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`  
= note: in format strings you may be able to use `{: ?}` (or `{: #?}` for pretty-print)  
instead
```

上面提示我们使用 `{: ?}` 来试试，这种方式我们在本文的前面也见过，下面来试试：

```
    println!("rect1 is {:?}", rect1);
```

可是依然无情报错了:

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

好在， 聪明的编译器又一次给出了提示:

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

让我们实现 `Debug` 特征， Oh No， 就是不想实现 `Display` 特征， 才用的 `{:?}",` 怎么又要实现 `Debug`， 但是仔细看， 提示中有一行： `add #[derive(Debug)] to Rectangle`， 哦？ 这不就是我们前文一直在使用的吗？

首先， Rust 默认不会为我们实现 `Debug`， 为了实现， 有两种方式可以选择：

- 手动实现
- 使用 `derive` 派生实现

后者简单的多， 但是也有限制， 具体见[附录 D](#)， 这里我们就不再深入讲解， 来看看该如何使用：

```
#[derive(Debug)]  
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
fn main() {  
    let rect1 = Rectangle {  
        width: 30,  
        height: 50,  
    };  
  
    println!("rect1 is {:?}", rect1);  
}
```

此时运行程序， 就不再有错误， 输出如下：

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

这个输出格式看上去也不赖嘛， 虽然未必是最好的。这种格式是 Rust 自动为我们提供的实现， 看上基本就跟结构体的定义形式一样。

当结构体较大时，我们可能希望能够有更好的输出表现，此时可以使用 `{:#?}` 来替代 `{:?:}`，输出如下：

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

此时结构体的输出跟我们创建时候的代码几乎一模一样了！当然，如果大家还是不满足，那最好还是自己实现 `Display` 特征，以向用户更美的展示你的私藏结构体。关于格式化输出的更多内容，我们强烈推荐看看这个[章节](#)。

还有一个简单的输出 debug 信息的方法，那就是使用 `dbg!` 宏，它会拿走表达式的所有权，然后打印出相应的文件名、行号等 debug 信息，当然还有我们需要的表达式的求值结果。**除此之外，它最终还会把表达式值的所有权返回！**

---

`dbg!` 输出到标准错误输出 `stderr`，而 `println!` 输出到标准输出 `stdout`。

---

下面的例子中清晰的展示了 `dbg!` 如何在打印出信息的同时，还把表达式的值赋给了 `width`：

```
#[derive(Debug)]  
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
fn main() {  
    let scale = 2;  
    let rect1 = Rectangle {  
        width: dbg!(30 * scale),  
        height: 50,  
    };  
  
    dbg!(&rect1);  
}
```

最终的 debug 输出如下：

```
$ cargo run  
[src/main.rs:10] 30 * scale = 60  
[src/main.rs:14] &rect1 = Rectangle {  
    width: 60,  
    height: 50,  
}
```

可以看到，我们想要的 debug 信息几乎都有了：代码所在的文件名、行号、表达式以及表达式的值，简直完美！

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 枚举

枚举(enum 或 enumeration)允许你通过列举可能的成员来定义一个**枚举类型**, 例如扑克牌花色:

```
enum PokerSuit {  
    Clubs,  
    Spades,  
    Diamonds,  
    Hearts,  
}
```

如果在此之前你没有在其它语言中使用过枚举, 那么可能需要花费一些时间来理解这些概念, 一旦上手, 就会发现枚举的强大, 甚至对它爱不释手, 枚举虽好, 可不要滥用哦。

再回到之前创建的 `PokerSuit`, 扑克总共有四种花色, 而这里我们枚举出所有的可能值, 这也正是 枚举名称的由来。

任何一张扑克, 它的花色肯定会落在四种花色中, 而且也只会落在其中一个花色上, 这种特性非常适合枚举的使用, 因为**枚举值**只可能是其中某一个成员。抽象来看, 四种花色尽管是不同的花色, 但是它们都是扑克花色这个概念, 因此当某个函数处理扑克花色时, 可以把它们当作相同的类型进行传参。

细心的读者应该注意到, 我们对之前的 **枚举类型** 和 **枚举值** 进行了重点标注, 这是因为对于新人来说容易混淆相应的概念, 总而言之: **枚举类型是一个类型, 它会包含所有可能的枚举成员, 而枚举值是该类型中的具体某个成员的实例。**

## 枚举值

现在来创建 `PokerSuit` 枚举类型的两个成员实例:

```
let heart = PokerSuit::Hearts;  
let diamond = PokerSuit::Diamonds;
```

我们通过 `::` 操作符来访问 `PokerSuit` 下的具体成员, 从代码可以清晰看出, `heart` 和 `diamond` 都是 `PokerSuit` 枚举类型的, 接着可以定义一个函数来使用它们:

```

fn main() {
    let heart = PokerSuit::Hearts;
    let diamond = PokerSuit::Diamonds;

    print_suit(heart);
    print_suit(diamond);
}

fn print_suit(card: PokerSuit) {
    // 需要在定义 enum PokerSuit 的上面添加上 #[derive(Debug)], 否则会报 card 没有实现 Debug
    println!("{}:{}", card);
}

```

`print_suit` 函数的参数类型是 `PokerSuit`，因此我们可以把 `heart` 和 `diamond` 传给它，虽然 `heart` 是基于 `PokerSuit` 下的 `Hearts` 成员实例化的，但是它是货真价实的 `PokerSuit` 枚举类型。

接下来，我们想让扑克牌变得更加实用，那么需要给每张牌赋予一个值：A(1)-K(13)，这样再加上花色，就是一张真实的扑克牌了，例如红心A。

目前来说，枚举值还不能带有值，因此先用结构体来实现：

```

enum PokerSuit {
    Clubs,
    Spades,
    Diamonds,
    Hearts,
}

struct PokerCard {
    suit: PokerSuit,
    value: u8
}

fn main() {
    let c1 = PokerCard {
        suit: PokerSuit::Clubs,
        value: 1,
    };
    let c2 = PokerCard {
        suit: PokerSuit::Diamonds,
        value: 12,
    };
}

```

这段代码很好的完成了它的使命，通过结构体 `PokerCard` 来代表一张牌，结构体的 `suit` 字段表示牌的花色，类型是 `PokerSuit` 枚举类型，`value` 字段代表扑克牌的数值。

可以吗？可以！好吗？说实话，不咋地，因为还有简洁得多的方式来实现：

```

enum PokerCard {
    Clubs(u8),
    Spades(u8),
    Diamonds(u8),
    Hearts(u8),
}

fn main() {
    let c1 = PokerCard::Spades(5);
    let c2 = PokerCard::Diamonds(13);
}

```

直接将数据信息关联到枚举成员上，省去近一半的代码，这种实现是不是更优雅？

不仅如此，同一个枚举类型下的不同成员还能持有不同的数据类型，例如让某些花色打印 1-13 的字样，另外的花色打印上 A-K 的字样：

```

enum PokerCard {
    Clubs(u8),
    Spades(u8),
    Diamonds(char),
    Hearts(char),
}

fn main() {
    let c1 = PokerCard::Spades(5);
    let c2 = PokerCard::Diamonds('A');
}

```

回想一下，遇到这种不同类型的情况，再用我们之前的结构体实现方式，可行吗？也许可行，但是会复杂很多。

再来看一个来自标准库中的例子：

```

struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}

```

这个例子跟我们之前的扑克牌很像，只不过枚举成员包含的类型更复杂了，变成了结构体：分别通过 `Ipv4Addr` 和 `Ipv6Addr` 来定义两种不同的 IP 数据。

从这些例子可以看出，**任何类型的数据都可以放入枚举成员中**：例如字符串、数值、结构体甚至另一个枚举。

增加一些挑战？先看以下代码：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let m1 = Message::Quit;
    let m2 = Message::Move{x:1,y:1};
    let m3 = Message::ChangeColor(255,255,0);
}
```

该枚举类型代表一条消息，它包含四个不同的成员：

- `Quit` 没有任何关联数据
- `Move` 包含一个匿名结构体
- `Write` 包含一个 `String` 字符串
- `ChangeColor` 包含三个 `i32`

当然，我们也可以用结构体的方式来定义这些消息：

```
struct QuitMessage; // 单元结构体
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 元组结构体
struct ChangeColorMessage(i32, i32, i32); // 元组结构体
```

由于每个结构体都有自己的类型，因此我们无法在需要同一类型的地方进行使用，例如某个函数它的功能是接受消息并进行发送，那么用枚举的方式，就可以接收不同的消息，但是用结构体，该函数无法接受 4 个不同的结构体作为参数。

而且从代码规范角度来看，枚举的实现更简洁，代码内聚性更强，不像结构体的实现，分散在各个地方。

## 同一化类型

最后，再用一个实际项目中的简化片段，来结束枚举类型的语法学习。

例如我们有一个 WEB 服务，需要接受用户的长连接，假设连接有两种：`TcpStream` 和 `TlsStream`，但是我们希望对这两个连接的处理流程相同，也就是用同一个函数来处理这两个连接，代码如下：

```
fn new (stream: TcpStream) {
    let mut s = stream;
    if tls {
        s = negotiate_tls(stream)
    }

    // websocket是一个WebSocket<TcpStream>或者
    // WebSocket<native_tls::TlsStream<TcpStream>>类型
    websocket = WebSocket::from_raw_socket(
        s, ....)
}
```

此时，枚举类型就能帮上大忙：

```
enum Websocket {
    Tcp(WebSocket<TcpStream>),
    Tls(WebSocket<native_tls::TlsStream<TcpStream>>),
}
```

## Option 枚举用于处理空值

在其它编程语言中，往往都有一个 `null` 关键字，该关键字用于表明一个变量当前的值为空（不是零值，例如整型的零值是 0），也就是不存在值。当你对这些 `null` 进行操作时，例如调用一个方法，就会直接抛出**null 异常**，导致程序的崩溃，因此我们在编程时需要格外的小心去处理这些 `null` 空值。

---

Tony Hoare，`null` 的发明者，曾经说过一段非常有名的话：

我称之为我十亿美元的错误。当时，我在使用一个面向对象语言设计第一个综合性的面向引用的类型系统。我的目标是通过编译器的自动检查来保证所有引用的使用都应该是绝对安全的。不过在设计过程中，我未能抵抗住诱惑，引入了空引用的概念，因为它非常容易实现。就是因为这个决策，引发了无数错误、漏洞和系统崩溃，在之后的四十多年中造成了数十亿美元的苦痛和伤害。

---

尽管如此，空值的表达依然非常有意义，因为空值表示当前时刻变量的值是缺失的。有鉴于此，Rust 吸取了众多教训，决定抛弃 `null`，而改为使用 `Option` 枚举变量来表述这种结果。

`Option` 枚举包含两个成员，一个成员表示含有值：`Some(T)`，另一个表示没有值：`None`，定义如下：

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

其中 `T` 是泛型参数，`Some(T)` 表示该枚举成员的数据类型是 `T`，换句话说，`Some` 可以包含任何类型的数据。

`Option<T>` 枚举是如此有用以至于它被包含在了 `prelude`（`prelude` 属于 Rust 标准库，Rust 会将最常用的类型、函数等提前引入其中，省得我们再手动引入）之中，你不需要将其显式引入作用域。另外，它的成员 `Some` 和 `None` 也是如此，无需使用 `Option::` 前缀就可直接使用 `Some` 和 `None`。总之，不能因为 `Some(T)` 和 `None` 中没有 `Option::` 的身影，就否认它们是 `Option` 下的卧龙凤雏。

再来看以下代码：

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

如果使用 `None` 而不是 `Some`，需要告诉 Rust `Option<T>` 是什么类型的，因为编译器只通过 `None` 值无法推断出 `Some` 成员保存的值的类型。

当有一个 `Some` 值时，我们就知道存在一个值，而这个值保存在 `Some` 中。当有个 `None` 值时，在某种意义上，它跟空值具有相同的意义：并没有一个有效的值。那么，`Option<T>` 为什么就比空值要好呢？

简而言之，因为 `Option<T>` 和 `T`（这里 `T` 可以是任何类型）是不同的类型，例如，这段代码不能编译，因为它尝试将 `Option<i8>` (`Option<T>`) 与 `i8(T)` 相加：

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

如果运行这些代码，将得到类似这样的错误信息：

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
-->
|
5 |     let sum = x + y;
|         ^ no implementation for `i8 + std::option::Option<i8>`
```

很好！事实上，错误信息意味着 Rust 不知道该如何将 `Option<i8>` 与 `i8` 相加，因为它们的类型不同。当在 Rust 中拥有一个像 `i8` 这样类型的值时，编译器确保它总是有一个有效的值，我们可以放心使用而无需做空值检查。只有当使用 `Option<i8>`（或者任何用到的类型）的时候才需要担心可能没有值，而编译器会确保我们在使用值之前处理了为空的情况。

换句话说，在对 `Option<T>` 进行 `T` 的运算之前必须将其转换为 `T`。通常这能帮助我们捕获到空值最常见的问题之一：期望某值不为空但实际上为空的情况。

不再担心会错误的使用一个空值，会让你对代码更加有信心。为了拥有一个可能为空的值，你必须要显式的将其放入对应类型的 `Option<T>` 中。接着，当使用这个值时，必须明确的处理值为空的情况。只要一个值不是 `Option<T>` 类型，你就 **可以** 安全的认定它的值不为空。这是 Rust 的一个经过深思熟虑的设计决策，来限制空值的泛滥以增加 Rust 代码的安全性。

那么当有一个 `Option<T>` 的值时，如何从 `Some` 成员中取出 `T` 的值来使用它呢？`Option<T>` 枚举拥有大量用于各种情况的方法：你可以查看[它的文档](#)。熟悉 `Option<T>` 的方法将对你的 Rust 之旅非常有用。

总的来说，为了使用 `Option<T>` 值，需要编写处理每个成员的代码。你想要一些代码只当拥有 `Some(T)` 值时运行，允许这些代码使用其中的 `T`。也希望一些代码在值为 `None` 时运行，这些代码并没有一个可用的 `T` 值。`match` 表达式就是这么一个处理枚举的控制流结构：它会根据枚举的成员运行不同的代码，这些代码可以使用匹配到的值中的数据。

这里先简单看一下 `match` 的大致模样，在[模式匹配](#)中，我们会详细讲解：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

`plus_one` 通过 `match` 来处理不同 `Option` 的情况。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 数组

在日常开发中，使用最广的数据结构之一就是数组，在 Rust 中，最常用的数组有两种，第一种是速度很快但是长度固定的 `array`，第二种是可动态增长的但是有性能损耗的 `Vector`，在本书中，我们称 `array` 为数组，`Vector` 为动态数组。

不知道你们发现没，这两个数组的关系跟 `&str` 与 `String` 的关系很像，前者是长度固定的字符串切片，后者是可动态增长的字符串。其实，在 Rust 中无论是 `String` 还是 `Vector`，它们都是 Rust 的高级类型：集合类型，在后面章节会有详细介绍。

对于本章节，我们的重点还是放在数组 `array` 上。数组的具体定义很简单：将多个类型相同的元素依次组合在一起，就是一个数组。结合上面的内容，可以得出数组的三要素：

- 长度固定
- 元素必须有相同的类型
- 依次线性排列

这里再啰嗦一句，**我们这里说的数组是 Rust 的基本类型，是固定长度的，这点与其他编程语言不同，其它编程语言的数组往往是可变长度的，与 Rust 中的动态数组 `Vector` 类似**，希望读者大大牢记此点。

## 创建数组

在 Rust 中，数组是这样定义的：

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

数组语法跟 JavaScript 很像，也跟大多数编程语言很像。由于它的元素类型大小固定，且长度也是固定，因此**数组 `array` 是存储在线上**，性能也会非常优秀。与此对应，**动态数组 `Vector` 是存储在堆上**，因此长度可以动态改变。当你不确定是使用数组还是动态数组时，那就应该使用后者，具体见[动态数组 `Vector`](#)。

举个例子，在需要知道一年中各个月份名称的程序中，你很可能希望使用的是数组而不是动态数组。因为月份是固定的，它总是只包含 12 个元素：

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

在一些时候，还需要为数组声明类型，如下所示：

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

这里，数组类型是通过方括号语法声明，`i32` 是元素类型，分号后面的数字 `5` 是数组长度，数组类型也从侧面说明了**数组的元素类型要统一，长度要固定**。

还可以使用下面的语法初始化一个**某个值重复出现 N 次的数组**：

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

`a` 数组包含 `5` 个元素，这些元素的初始化值为 `3`，聪明的读者已经发现，这种语法跟数组类型的声明语法其实是保持一致的：`[3; 5]` 和 `[类型; 长度]`。

在元素重复的场景，这种写法要简单的多，否则你就得疯狂敲击键盘：`let a = [3, 3, 3, 3, 3];`，不过老板可能很喜欢你的这种疯狂编程的状态。

## 访问数组元素

因为数组是连续存放元素的，因此可以通过索引的方式来访问存放其中的元素：

```
fn main() {
    let a = [9, 8, 7, 6, 5];

    let first = a[0]; // 获取a数组第一个元素
    let second = a[1]; // 获取第二个元素
}
```

与许多语言类似，数组的索引下标是从 `0` 开始的。此处，`first` 获取到的值是 `9`，`second` 是 `8`。

## 越界访问

如果使用超出数组范围的索引访问数组元素，会怎么样？下面是一个接收用户的控制台输入，然后将其作为索引访问数组元素的例子：

```
use std::io;

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

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

    let mut index = String::new();
    // 读取控制台的输出
    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

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

    let element = a[index];

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

使用 `cargo run` 来运行代码，因为数组只有 5 个元素，如果我们试图输入 5 去访问第 6 个元素，则会访问到不存在的数组元素，最终程序会崩溃退出：

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

这就是数组访问越界，访问了数组中不存在的元素，导致 Rust 运行时错误。程序因此退出并显示错误消息，未执行最后的 `println!` 语句。

当你尝试使用索引访问元素时，Rust 将检查你指定的索引是否小于数组长度。如果索引大于或等于数组长度，Rust 会出现 ***panic***。这种检查只能在运行时进行，比如在上面这种情况下，编译器无法在编译期知道用户运行代码时将输入什么值。

这种就是 Rust 的安全特性之一。在很多系统编程语言中，并不会检查数组越界问题，你会访问到无效的内存地址获取到一个风马牛不相及的值，最终导致在程序逻辑上出现大问题，而且这种问题会非常难以检查。

## 数组元素为非基础类型

学习了上面的知识，很多朋友肯定觉得已经学会了Rust的数组类型，但现实会给我们一记重锤，实际开发中还会碰到一种情况，就是**数组元素是非基本类型的**，这时候大家一定会这样写。

```
let array = [String::from("rust is good!"); 8];  
println!("{:?}", array);
```

然后你会惊喜的得到编译错误。

```
error[E0277]: the trait bound `String: std::marker::Copy` is not satisfied  
--> src/main.rs:7:18  
|  
7 |     let array = [String::from("rust is good!"); 8];  
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::marker::Copy` is  
not implemented for `String`  
|  
= note: the `Copy` trait is required because this value will be copied for each  
element of the array
```

有些还没有看过特征的小伙伴，有可能不太明白这个报错，不过这个目前可以不提，我们就拿之前所学的**所有权**知识，就可以思考明白，前面几个例子都是Rust的基本类型，而**基本类型在Rust中赋值是以Copy的形式**，这时候你就懂了吧，`let array=[3;5]` 底层就是不断的Copy出来的，但很可惜复杂类型都没有深拷贝，只能一个个创建。

接着就有小伙伴会这样写。

```
let array = [String::from("rust is good!"), String::from("rust is  
good!"), String::from("rust is good!")];  
  
println!("{:?}", array);
```

作为一个追求极致完美的Rust开发者，怎么能容忍上面这么难看的代码存在！

**正确的写法**，应该调用 `std::array::from_fn`

```
let array: [String; 8] = std::array::from_fn(|_i| String::from("rust is good!"));  
  
println!("{:?}", array);
```

## 数组切片

在之前的[章节](#)，我们有讲到 [切片](#) 这个概念，它允许你引用集合中的部分连续片段，而不是整个集合，对于数组也是，数组切片允许我们引用数组的一部分：

```
let a: [i32; 5] = [1, 2, 3, 4, 5];  
let slice: &[i32] = &a[1..3];  
  
assert_eq!(slice, &[2, 3]);
```

上面的数组切片 `slice` 的类型是 `&[i32]`，与之对比，数组的类型是 `[i32;5]`，简单总结下切片的特点：

- 切片的长度可以与数组不同，并不是固定的，而是取决于你使用时指定的起始和结束位置
- 创建切片的代价非常小，因为切片只是针对底层数组的一个引用
- 切片类型`[T]`拥有不固定的大小，而切片引用类型`&[T]`则具有固定的大小，因为 Rust 很多时候都需要固定大小数据类型，因此`&[T]`更有用，`&str` 字符串切片也同理

## 总结

最后，让我们以一个综合性使用数组的例子，来结束本章节的学习：

```

fn main() {
    // 编译器自动推导出one的类型
    let one = [1, 2, 3];
    // 显式类型标注
    let two: [u8; 3] = [1, 2, 3];
    let blank1 = [0; 3];
    let blank2: [u8; 3] = [0; 3];

    // arrays是一个二维数组，其中每一个元素都是一个数组，元素类型是[u8; 3]
    let arrays: [[u8; 3]; 4] = [one, two, blank1, blank2];

    // 借用arrays的元素用作循环中
    for a in &arrays {
        print!("{:?}: ", a);
        // 将a变成一个迭代器，用于循环
        // 你也可以直接用for n in a {}来进行循环
        for n in a.iter() {
            print!("\t{} + 10 = {}", n, n+10);
        }

        let mut sum = 0;
        // 0..a.len,是一个 Rust 的语法糖，其实就等于一个数组，元素是从0,1,2一直增加到到a.len-1
        for i in 0..a.len() {
            sum += a[i];
        }
        println!("\t{:?} = {}", a, sum);
    }
}

```

做个总结，数组虽然很简单，但是其实还是存在几个要注意的点：

- **数组类型容易跟数组切片混淆**，[T;n]描述了一个数组的类型，而[T]描述了切片的类型，因为切片是运行期的数据结构，它的长度无法在编译期得知，因此不能用[T;n]的形式去描述
- [u8; 3] 和 [u8; 4] 是不同的类型，数组的长度也是类型的一部分
- **在实际开发中，使用最多的是数组切片[T]**，我们往往通过引用的方式去使用 &[T]，因为后者有固定的类型大小

至此，关于数据类型部分，我们已经全部学完了，对于 Rust 学习而言，我们也迈出了坚定的第一步，后面将开始更高级特性的学习。未来如果大家有疑惑需要检索知识，一样可以继续回顾过往的章节，因为本书不仅仅是一门 Rust 的教程，还是一本厚重的 Rust 工具书。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 流程控制

80 后应该都对学校的小混混记忆犹新，在那个时代，小混混们往往都认为自己是地下王者，管控着地下事务的流程，在我看来，他们就像代码中的流程控制一样，无处不在，很显眼，但是又让人懒得重视。

言归正传，Rust 程序是从上而下顺序执行的，在此过程中，我们可以通过循环、分支等流程控制方式，更好的实现相应的功能。

## 使用 if 来做分支控制

---

if else 无处不在 -- 鲁迅

---

但凡你能找到一门编程语言没有 `if else`，那么一定更要反馈给鲁迅，反正不是我说的:) 总之，只要你拥有其它语言的编程经验，就一定会有以下认知：`if else 表达式`根据条件执行不同的代码分支：

```
if condition == true {  
    // A...  
} else {  
    // B...  
}
```

该代码读作：若 `condition` 的值为 `true`，则执行 A 代码，否则执行 B 代码。

先看下面代码：

```
fn main() {  
    let condition = true;  
    let number = if condition {  
        5  
    } else {  
        6  
    };  
  
    println!("The value of number is: {}", number);  
}
```

以上代码有以下几点要注意：

- **if 语句块是表达式**，这里我们使用 `if` 表达式的返回值来给 `number` 进行赋值：`number` 的值是 5

- 用 `if` 来赋值时，要保证每个分支返回的类型一样(事实上，这种说法不完全准确，见[这里](#))，此处返回的 5 和 6 就是同一个类型，如果返回类型不一致就会报错

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
|
4 |     let number = if condition {
|     -----^
5 |     |         5
6 |     |     } else {
7 |     |         "six"
8 |     |     };
|     |-----^ expected integer, found &str // 期望整数类型，但却发现&str字符串切片
|
= note: expected type `<integer>`
        found type `&str`
```

## 使用 `else if` 来处理多重条件

可以将 `else if` 与 `if`、`else` 组合在一起实现更复杂的条件分支判断：

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

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

程序执行时，会按照自上至下的顺序执行每一个分支判断，一旦成功，则跳出 `if` 语句块，最终本程序会匹配执行 `else if n % 3 == 0` 的分支，输出 "number is divisible by 3"。

有一点要注意，就算有多个分支能匹配，也只有第一个匹配的分支会被执行！

如果代码中有大量的 `else if` 会让代码变得极其丑陋，不过不用担心，下一章的 `match` 专门用以解决多分支模式匹配的问题。

# 循环控制

循环无处不在，上到数钱，下到数年，你能想象的很多场景都存在循环，因此它也是流程控制中最重要的组成部分之一。

在 Rust 语言中有三种循环方式：`for`、`while` 和 `loop`，其中 `for` 循环是 Rust 循环王冠上的明珠。

## for 循环

`for` 循环是 Rust 的大杀器：

```
fn main() {
    for i in 1..=5 {
        println!("{}", i);
    }
}
```

以上代码循环输出一个从 1 到 5 的序列，简单粗暴，核心就在于 `for` 和 `in` 的联动，语义表达如下：

```
for 元素 in 集合 {
    // 使用元素干一些你懂我不懂的事情
}
```

这个语法跟 JavaScript 还蛮像，应该挺好理解。

注意，使用 `for` 时我们往往使用集合的引用形式，除非你不想在后面的代码中继续使用该集合（比如我们这里使用了 `container` 的引用）。如果不使用引用的话，所有权会被转移（move）到 `for` 语句块中，后面就无法再使用这个集合了）：

```
for item in &container {
    // ...
}
```

---

对于实现了 `copy` 特征的数组(例如 `[i32; 10]`)而言，`for item in arr` 并不会把 `arr` 的所有权转移，而是直接对其进行了拷贝，因此循环之后仍然可以使用 `arr`。

---

如果想在循环中，修改该元素，可以使用 `mut` 关键字：

```

for item in &mut collection {
    // ...
}

```

总结如下：

使用方法	等价使用方式	所有权
for item in collection	for item in IntoIterator::into_iter(collection)	转移所有权
for item in &collection	for item in collection.iter()	不可变借用
for item in &mut collection	for item in collection.iter_mut()	可变借用

如果想在循环中获取元素的索引：

```

fn main() {
    let a = [4, 3, 2, 1];
    // `<T>.iter()` 方法把 `a` 数组变成一个迭代器
    for (i, v) in a.iter().enumerate() {
        println!("第{}个元素是{}", i + 1, v);
    }
}

```

有同学可能会想到，如果我们想用 `for` 循环控制某个过程执行 10 次，但是又不想单独声明一个变量来控制这个流程，该怎么写？

```

for _ in 0..10 {
    // ...
}

```

可以用 `_` 来替代 `i` 用于 `for` 循环中，在 Rust 中 `_` 的含义是忽略该值或者类型的意思，如果不使用 `_`，那么编译器会给你一个 变量未使用的 警告。

## 两种循环方式优劣对比

以下代码，使用了两种循环方式：

```
// 第一种
let collection = [1, 2, 3, 4, 5];
for i in 0..collection.len() {
    let item = collection[i];
    // ...
}

// 第二种
for item in collection {
```

第一种方式是循环索引，然后通过索引下标去访问集合，第二种方式是直接循环集合中的元素，优劣如下：

- **性能**: 第一种使用方式中 `collection[index]` 的索引访问，会因为边界检查(Bounds Checking)导致运行时的性能损耗——Rust 会检查并确认 `index` 是否落在集合内，但是第二种直接迭代的方式就不会触发这种检查，因为编译器会在编译时就完成分析并证明这种访问是合法的
- **安全**: 第一种方式里对 `collection` 的索引访问是非连续的，存在一定可能性在两次访问之间，`collection` 发生了变化，导致脏数据产生。而第二种直接迭代的方式是连续访问，因此不存在这种风险(由于所有权限制，在访问过程中，数据并不会发生变化)。

由于 `for` 循环无需任何条件限制，也不需要通过索引来访问，因此是最安全也是最常用的，通过与下面的 `while` 的对比，我们能看到为什么 `for` 会更加安全。

## continue

使用 `continue` 可以跳过当前当次的循环，开始下次的循环：

```
for i in 1..4 {
    if i == 2 {
        continue;
    }
    println!("{}", i);
```

上面代码对 1 到 3 的序列进行迭代，且跳过值为 2 时的循环，输出如下：

```
1
3
```

## break

使用 `break` 可以直接跳出当前整个循环：

```
for i in 1..4 {
    if i == 2 {
        break;
    }
    println!("{}!", i);
}
```

上面代码对 1 到 3 的序列进行迭代，在遇到值为 2 时的跳出整个循环，后面的循环不再执行，输出如下：

```
1
```

## while 循环

如果你需要一个条件来循环，当该条件为 `true` 时，继续循环，条件为 `false`，跳出循环，那么 `while` 就非常适用：

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

    while n <= 5 {
        println!("{}!", n);

        n = n + 1;
    }

    println!("我出来了!");
}
```

该 `while` 循环，只有当 `n` 小于等于 5 时，才执行，否则就立刻跳出循环，因此在上述代码中，它会先从 0 开始，满足条件，进行循环，然后是 1，满足条件，进行循环，最终到 6 的时候，大于 5，不满足条件，跳出 `while` 循环，执行 `我出来了!` 的打印，然后程序结束：

```
0!  
1!  
2!  
3!  
4!  
5!  
我出来了!
```

当然，你也可以用其它方式组合实现，例如 `loop`（无条件循环，将在下面介绍）+ `if` + `break`：

```
fn main() {  
    let mut n = 0;  
  
    loop {  
        if n > 5 {  
            break  
        }  
        println!("{} {}", n);  
        n += 1;  
    }  
  
    println!("我出来了!");  
}
```

可以看出，在这种循环场景下，`while` 要简洁的多。

## while vs for

我们也能用 `while` 来实现 `for` 的功能：

```
fn main() {  
    let a = [10, 20, 30, 40, 50];  
    let mut index = 0;  
  
    while index < 5 {  
        println!("the value is: {}", a[index]);  
  
        index = index + 1;  
    }  
}
```

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

```
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

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

但这个过程很容易出错；如果索引长度不正确会导致程序 **panic**。这也使程序更慢，因为编译器增加了运行时代码来对每次循环的每个元素进行条件检查。

`for` 循环代码如下：

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

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

可以看出，`for` 并不会使用索引去访问数组，因此更安全也更简洁，同时避免 运行时的边界检查，性能更高。

## loop 循环

对于循环而言，`loop` 循环毋庸置疑，是适用面最高的，它可以适用于所有循环场景（虽然能用，但是在很多场景下，`for` 和 `while` 才是最优选择），因为 `loop` 就是一个简单的无限循环，你可以在内部实现逻辑通过 `break` 关键字来控制循环何时结束。

使用 `loop` 循环一定要打起精神，否则你会写出下面的跑满你一个 CPU 核心的疯子代码：

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

该循环会不停的在终端打印输出，直到你使用 `ctrl-C` 结束程序：

```
again!
again!
again!
again!
^Cagain!
```

**注意**，不要轻易尝试上述代码，如果你电脑配置不行，可能会死机！！！

因此，当使用 `loop` 时，必不可少的伙伴是 `break` 关键字，它能让循环在满足某个条件时跳出：

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

    let result = loop {
        counter += 1;

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

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

以上代码当 `counter` 递增到 10 时，就会通过 `break` 返回一个 `counter * 2` 的值，最后赋给 `result` 并打印出来。

这里有几点值得注意：

- **break 可以单独使用，也可以带一个返回值**，有些类似 `return`
- **loop 是一个表达式**，因此可以返回一个值

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

# 模式匹配

模式匹配，这个词，对于非函数语言编程来说，真的还蛮少听到，因为它经常出现在函数式编程里，用于为复杂的类型系统提供一个轻松的解构能力。

曾记否？在枚举和流程控制那章，我们遗留了两个问题，都是关于 `match` 的，第一个是如何对 `Option` 枚举进行进一步处理，另外一个是用 `match` 来替代 `else if` 这种丑陋的多重分支使用方式。那么让我们先一起来揭开 `match` 的神秘面纱。

# match 和 if let

在 Rust 中，模式匹配最常用的就是 `match` 和 `if let`，本章节将对两者及相关的概念进行详尽介绍。

先来看一个关于 `match` 的简单例子：

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

fn main() {
    let dire = Direction::South;
    match dire {
        Direction::East => println!("East"),
        Direction::North | Direction::South => {
            println!("South or North");
        },
        _ => println!("West"),
    };
}
```

这里我们想去匹配 `dire` 对应的枚举类型，因此在 `match` 中用三个匹配分支来完全覆盖枚举变量 `Direction` 的所有成员类型，有以下几点值得注意：

- `match` 的匹配必须要穷举出所有可能，因此这里用 `_` 来代表未列出的所有可能性
- `match` 的每一个分支都必须是一个表达式，且所有分支的表达式最终返回值的类型必须相同
- `X | Y`，类似逻辑运算符 `或`，代表该分支可以匹配 `X` 也可以匹配 `Y`，只要满足一个即可

其实 `match` 跟其他语言中的 `switch` 非常像，`_` 类似于 `switch` 中的 `default`。

## match 匹配

首先来看看 `match` 的通用形式：

```
match target {  
    模式1 => 表达式1,  
    模式2 => {  
        语句1;  
        语句2;  
        表达式2  
    },  
    _ => 表达式3  
}
```

该形式清晰的说明了何为模式，何为模式匹配：将模式与 `target` 进行匹配，即为模式匹配，而模式匹配不仅仅局限于 `match`，后面我们会详细阐述。

`match` 允许我们将一个值与一系列的模式相比较，并根据相匹配的模式执行对应的代码，下面让我们来一一详解，先看一个例子：

```
enum Coin {  
    Penny,  
    Nickel,  
    Dime,  
    Quarter,  
}  
  
fn value_in_cents(coin: Coin) -> u8 {  
    match coin {  
        Coin::Penny => {  
            println!("Lucky penny!");  
            1  
        },  
        Coin::Nickel => 5,  
        Coin::Dime => 10,  
        Coin::Quarter => 25,  
    }  
}
```

`value_in_cents` 函数根据匹配到的硬币，返回对应的美分数值。`match` 后紧跟着的是一个表达式，跟 `if` 很像，但是 `if` 后的表达式必须是一个布尔值，而 `match` 后的表达式返回值可以是任意类型，只要能跟后面的分支中的模式匹配起来即可，这里的 `coin` 是枚举 `Coin` 类型。

接下来是 `match` 的分支。一个分支有两个部分：**一个模式和针对该模式的处理代码**。第一个分支的模式是 `Coin::Penny`，其后的 `=>` 运算符将模式和将要运行的代码分开。这里的代码就仅仅是表达式 `1`，不同分支之间使用逗号分隔。

当 `match` 表达式执行时，它将目标值 `coin` 按顺序依次与每一个分支的模式相比较，如果模式匹配了这个值，那么模式之后的代码将被执行。如果模式并不匹配这个值，将继续执行下一个分支。

每个分支相关联的代码是一个表达式，而表达式的结果值将作为整个 `match` 表达式的返回值。如果分支有多行代码，那么需要用 `{}` 包裹，同时最后一行代码需要是一个表达式。

## 使用 `match` 表达式赋值

还有一点很重要，`match` 本身也是一个表达式，因此可以用它来赋值：

```
enum IpAddr {
    Ipv4,
    Ipv6
}

fn main() {
    let ip1 = IpAddr::Ipv6;
    let ip_str = match ip1 {
        IpAddr::Ipv4 => "127.0.0.1",
        _ => "::1",
    };

    println!("{}", ip_str);
}
```

因为这里匹配到 `_` 分支，所以将 `::1` 赋值给了 `ip_str`。

## 模式绑定

模式匹配的另外一个重要功能是从模式中取出绑定的值，例如：

```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState), // 25美分硬币
}
```

其中 `Coin::Quarter` 成员还存放了一个值：美国的某个州（因为在 1999 年到 2008 年间，美国在 25 美分(Quarter)硬币的背后为 50 个州印刷了不同的标记，其它硬币都没有这样的设计）。

接下来，我们希望在模式匹配中，获取到 25 美分硬币上刻印的州的名称：

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}", state);
            25
        },
    }
}
```

上面代码中，在匹配 `Coin::Quarter(state)` 模式时，我们把它内部存储的值绑定到了 `state` 变量上，因此 `state` 变量就是对应的 `UsState` 枚举类型。

例如有一个印了阿拉斯加州标记的 25 分硬币： `Coin::Quarter(UsState::Alaska)`，它在匹配时，`state` 变量将被绑定 `UsState::Alaska` 的枚举值。

再来看一个更复杂的例子：

运行后输出：

```
$ cargo run
Compiling world_hello v0.1.0 (/Users/sunfei/development/rust/world_hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.16s
    Running `target/debug/world_hello`
Hello Rust
point from (0, 0) move to (1, 2)
change color into '(r:255, g:255, b:0)', 'b' has been ignored
```

## 穷尽匹配

在文章的开头，我们简单总结过 `match` 的匹配必须穷尽所有情况，下面来举例说明，例如：

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

fn main() {
    let dire = Direction::South;
    match dire {
        Direction::East => println!("East"),
        Direction::North | Direction::South => {
            println!("South or North");
        },
    };
}
```

我们没有处理 `Direction::West` 的情况，因此会报错：

```

error[E0004]: non-exhaustive patterns: `West` not covered // 非穷尽匹配, `West` 没有被覆盖
--> src/main.rs:10:11
|
1 / enum Direction {
2 | |   East,
3 | |   West,
4 | |   ---- not covered
5 | |   North,
6 | |   South,
7 | |
8 | | }
9 | |_- `Direction` defined here
...
10|     match dire {
|     |     ^^^^ pattern `West` not covered // 模式 `West` 没有被覆盖
|     |
|     = help: ensure that all possible cases are being handled, possibly by adding
|     wildcards or more match arms
|     = note: the matched value is of type `Direction`

```

不禁想感叹，Rust 的编译器**真强大**，忍不住想爆粗口了，sorry，如果你以后进一步深入使用 Rust 也会像我这样感叹的。Rust 编译器清晰地知道 `match` 中有哪些分支没有被覆盖，这种行为能强制我们处理所有的可能性，有效避免传说中价值**十亿美金**的 `null` 陷阱。

## \_ 通配符

当我们不想在匹配时列出所有值的时候，可以使用 Rust 提供的一个特殊**模式**，例如，`u8` 可以拥有 0 到 255 的有效的值，但是我们只关心 1、3、5 和 7 这几个值，不想列出其它的 0、2、4、6、8、9 一直到 255 的值。那么，我们不必一个一个列出所有值，因为可以使用特殊的模式 `_` 替代：

```

let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}

```

通过将 `_` 其放置于其他分支后，`_` 将会匹配所有遗漏的值。`()` 表示返回**单元类型**与所有分支返回值的类型相同，所以当匹配到 `_` 后，什么也不会发生。

除了 `_` 通配符，用一个变量来承载其他情况也是可以的。

```

#[derive(Debug)]
enum Direction {
    East,
    West,
    North,
    South,
}

fn main() {
    let dire = Direction::South;
    match dire {
        Direction::East => println!("East"),
        other => println!("other direction: {:?}", other),
    };
}

```

然而，在某些场景下，我们其实只关心**某一个值是否存在**，此时 `match` 就显得过于啰嗦。

## if let 匹配

有时会遇到只有一个模式的值需要被处理，其它值直接忽略的场景，如果用 `match` 来处理就要写成下面这样：

```

let v = Some(3u8);
match v {
    Some(3) => println!("three"),
    _ => (),
}

```

我们只想要对 `Some(3)` 模式进行匹配，不想处理任何其他 `Some<u8>` 值或 `None` 值。但是为了满足 `match` 表达式（穷尽性）的要求，写代码时必须在处理完这唯一的成员后加上 `_ => ()`，这样会增加不少无用的代码。

俗话说“杀鸡焉用牛刀”，我们完全可以用 `if let` 的方式来实现：

```

if let Some(3) = v {
    println!("three");
}

```

这两种匹配对于新手来说，可能有些难以抉择，但是只要记住一点就好：**当你只要匹配一个条件，且忽略其他条件时就用 if let，否则都用 match。**

## matches!宏

Rust 标准库中提供了一个非常实用的宏：`matches!`，它可以将一个表达式跟模式进行匹配，然后返回匹配的结果 `true` or `false`。

例如，有一个动态数组，里面有以下枚举：

```
enum MyEnum {
    Foo,
    Bar
}

fn main() {
    let v = vec![MyEnum::Foo, MyEnum::Bar, MyEnum::Foo];
}
```

现在如果想对 `v` 进行过滤，只保留类型是 `MyEnum::Foo` 的元素，你可能想这么写：

```
v.iter().filter(|x| x == MyEnum::Foo);
```

但是，实际上这行代码会报错，因为你无法将 `x` 直接跟一个枚举成员进行比较。好在，你可以使用 `match` 来完成，但是会导致代码更为啰嗦，是否有更简洁的方式？答案是使用 `matches!`：

```
v.iter().filter(|x| matches!(x, MyEnum::Foo));
```

很简单也很简洁，再来看看更多的例子：

```
let foo = 'f';
assert!(matches!(foo, 'A'..='Z' | 'a'..='z'));

let bar = Some(4);
assert!(matches!(bar, Some(x) if x > 2));
```

## 变量遮蔽

无论是 `match` 还是 `if let`，这里都是一个新的代码块，而且这里的绑定相当于新变量，如果你使用同名变量，会发生变量遮蔽：

```

fn main() {
    let age = Some(30);
    println!("在匹配前, age是{:?}", age);
    if let Some(age) = age {
        println!("匹配出来的age是{}", age);
    }

    println!("在匹配后, age是{:?}", age);
}

```

cargo run 运行后输出如下：

```

在匹配前, age是Some(30)
匹配出来的age是30
在匹配后, age是Some(30)

```

可以看出在 `if let` 中，`=` 右边 `Some(i32)` 类型的 `age` 被左边 `i32` 类型的新 `age` 遮蔽了，该遮蔽一直持续到 `if let` 语句块的结束。因此第三个 `println!` 输出的 `age` 依然是 `Some(i32)` 类型。

对于 `match` 类型也是如此：

```

fn main() {
    let age = Some(30);
    println!("在匹配前, age是{:?}", age);
    match age {
        Some(age) => println!("匹配出来的age是{}", age),
        _ => ()
    }
    println!("在匹配后, age是{:?}", age);
}

```

需要注意的是，**`match` 中的变量遮蔽其实不是那么容易看出**，因此要小心！其实这里最好不要使用同名，避免难以理解，如下。

```

fn main() {
    let age = Some(30);
    println!("在匹配前, age是{:?}", age);
    match age {
        Some(x) => println!("匹配出来的age是{}", x),
        _ => ()
    }
    println!("在匹配后, age是{:?}", age);
}

```

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 解构 Option

在枚举那章，提到过 `Option` 枚举，它用来解决 Rust 中变量是否有值的问题，定义如下：

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

简单解释就是：**一个变量要么有值： `Some(T)`，要么为空： `None`。**

那么现在的问题就是该如何去使用这个 `Option` 枚举类型，根据我们上一节的经验，可以通过 `match` 来实现。

---

因为 `Option`，`Some`，`None` 都包含在 `prelude` 中，因此你可以直接通过名称来使用它们，而无需以 `Option::Some` 这种形式去使用，总之，千万不要因为调用路径变短了，就忘记 `Some` 和 `None` 也是 `Option` 底下的枚举成员！

---

## 匹配 `Option<T>`

使用 `Option<T>`，是为了从 `Some` 中取出其内部的 `T` 值以及处理没有值的情况，为了演示这一点，下面一起来编写一个函数，它获取一个 `Option<i32>`，如果其中含有一个值，将其加一；如果其中没有值，则函数返回 `None` 值：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

`plus_one` 接受一个 `Option<i32>` 类型的参数，同时返回一个 `Option<i32>` 类型的值(这种形式的函数在标准库内随处可见)，在该函数的内部处理中，如果传入的是一个 `None`，则返回一个 `None` 且不做任

何处理；如果传入的是一个 `Some(i32)`，则通过模式绑定，把其中的值绑定到变量 `i` 上，然后返回 `i+1` 的值，同时用 `Some` 进行包裹。

为了进一步说明，假设 `plus_one` 函数接受的参数值 `x` 是 `Some(5)`，来看看具体的分支匹配情况：

### 传入参数 `Some(5)`

`None => None,`

首先是匹配 `None` 分支，因为值 `Some(5)` 并不匹配模式 `None`，所以继续匹配下一个分支。

`Some(i) => Some(i + 1),`

`Some(5)` 与 `Some(i)` 匹配吗？当然匹配！它们是相同的成员。`i` 绑定了 `Some` 中包含的值，因此 `i` 的值是 5。接着匹配分支的代码被执行，最后将 `i` 的值加一并返回一个含有值 6 的新 `Some`。

### 传入参数 `None`

接着考虑下 `plus_one` 的第二个调用，这次传入的 `x` 是 `None`，我们进入 `match` 并与第一个分支相比 较。

`None => None,`

匹配上了！接着程序继续执行该分支后的代码：返回表达式 `None` 的值，也就是返回一个 `None`，因为第一个分支就匹配到了，其他的分支将不再比较。

# 模式适用场景

## 模式

模式是 Rust 中的特殊语法，它用来匹配类型中的结构和数据，它往往和 `match` 表达式联用，以实现强大的模式匹配能力。模式一般由以下内容组合而成：

- 字面值
- 解构的数组、枚举、结构体或者元组
- 变量
- 通配符
- 占位符

## 所有可能用到模式的地方

### `match` 分支

```
match VALUE {  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
}
```

如上所示，`match` 的每个分支就是一个模式，因为 `match` 匹配是穷尽式的，因此我们往往需要一个特殊的模式 `_`，来匹配剩余的所有情况：

```
match VALUE {  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
    _ => EXPRESSION,  
}
```

### `if let` 分支

`if let` 往往用于匹配一个模式，而忽略剩下的所有模式的场景：

```
if let PATTERN = SOME_VALUE {  
}
```

## while let 条件循环

一个与 `if let` 类似的结构是 `while let` 条件循环，它允许只要模式匹配就一直进行 `while` 循环。下面展示了一个使用 `while let` 的例子：

```
// Vec是动态数组  
let mut stack = Vec::new();  
  
// 向数组尾部插入元素  
stack.push(1);  
stack.push(2);  
stack.push(3);  
  
// stack.pop从数组尾部弹出元素  
while let Some(top) = stack.pop() {  
    println!("{}", top);  
}
```

这个例子会打印出 3、2 接着是 1。`pop` 方法取出动态数组的最后一个元素并返回 `Some(value)`，如果动态数组是空的，将返回 `None`，对于 `while` 来说，只要 `pop` 返回 `Some` 就会一直不停的循环。一旦其返回 `None`，`while` 循环停止。我们可以使用 `while let` 来弹出栈中的每一个元素。

你也可以用 `loop + if let` 或者 `match` 来实现这个功能，但是会更加啰嗦。

## for 循环

```
let v = vec!['a', 'b', 'c'];  
  
for (index, value) in v.iter().enumerate() {  
    println!("{} is at index {}", value, index);  
}
```

这里使用 `enumerate` 方法产生一个迭代器，该迭代器每次迭代会返回一个（索引，值）形式的元组，然后用 `(index,value)` 来匹配。

## let 语句

```
let PATTERN = EXPRESSION;
```

是的，该语句我们已经用了无数次了，它也是一种模式匹配：

```
let x = 5;
```

这其中，`x` 也是一种模式绑定，代表将匹配的值绑定到变量 `x` 上。因此，在 Rust 中，**变量名也是一种模式**，只不过它比较朴素很不起眼罢了。

```
let (x, y, z) = (1, 2, 3);
```

上面将一个元组与模式进行匹配(**模式和值的类型必需相同！**)，然后把 `1, 2, 3` 分别绑定到 `x, y, z` 上。

模式匹配要求两边的类型必须相同，否则就会导致下面的报错：

```
let (x, y) = (1, 2, 3);
```

```
error[E0308]: mismatched types
--> src/main.rs:4:5
|
4 | let (x, y) = (1, 2, 3);
|     ^^^^^^ ----- this expression has type `({integer}, {integer},
{integer})`
|         |
|         expected a tuple with 3 elements, found one with 2 elements
|
= note: expected tuple `({integer}, {integer}, {integer})`  

          found tuple `(_, _)`  

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

error: could not compile `playground` due to previous error
```

对于元组来说，元素个数也是类型的一部分！

## 函数参数

函数参数也是模式：

```
fn foo(x: i32) {
    // 代码
}
```

其中 `x` 就是一个模式，你还可以在参数中匹配元组：

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

&(3, 5) 会匹配模式 &(x, y) , 因此 x 得到了 3 , y 得到了 5 。

## let 和 if let

对于以下代码，编译器会报错：

```
let Some(x) = some_option_value;
```

因为右边的值可能不为 Some , 而是 None , 这种时候就不能进行匹配，也就是上面的代码遗漏了 None 的匹配。

类似 let , for 和 match 都必须要求完全覆盖匹配，才能通过编译( 不可驳模式匹配 )。

但是对于 if let , 就可以这样使用：

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

因为 if let 允许匹配一种模式，而忽略其余的模式( 可驳模式匹配 )。

# 全模式列表

在本书中我们已领略过许多不同类型模式的例子，本节的目标就是把这些模式语法都罗列出来，方便大家检索查阅（模式匹配在我们的开发中会经常用到）。

## 匹配字面值

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这段代码会打印 `one` 因为 `x` 的值是 1，如果希望代码获得特定的具体值，那么这种语法很有用。

## 匹配命名变量

在 `match` 中，我们有讲过变量遮蔽的问题，这个在**匹配命名变量**时会遇到：

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

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}", x, y);
}
```

让我们看看当 `match` 语句运行的时候发生了什么。第一个匹配分支的模式并不匹配 `x` 中定义的值，所以代码继续执行。

第二个匹配分支中的模式引入了一个新变量 `y`，它会匹配任何 `Some` 中的值。因为这里的 `y` 在 `match` 表达式的作用域中，而不是之前 `main` 作用域中，所以这是一个新变量，不是开头声明为值 10 的那个

`y`。这个新的 `y` 绑定会匹配任何 `Some` 中的值，在这里是 `x` 中的值。因此这个 `y` 绑定了 `x` 中 `Some` 内部的值。这个值是 5，所以这个分支的表达式将会执行并打印出 `Matched, y = 5`。

如果 `x` 的值是 `None` 而不是 `Some(5)`，头两个分支的模式不会匹配，所以会匹配模式 `_`。这个分支的模式中没有引入变量 `x`，所以此时表达式中的 `x` 会是外部没有被遮蔽的 `x`，也就是 `None`。

一旦 `match` 表达式执行完毕，其作用域也就结束了，同理内部 `y` 的作用域也结束了。最后的 `println!` 会打印 `at the end: x = Some(5), y = 10`。

如果你不想引入变量遮蔽，可以使用另一个变量名而非 `y`，或者使用匹配守卫(match guard)的方式，稍后在[匹配守卫提供的额外条件](#)中会讲解。

## 单分支多模式

在 `match` 表达式中，可以使用 `|` 语法匹配多个模式，它代表 或 的意思。例如，如下代码将 `x` 的值与匹配分支相比较，第一个分支有 或 选项，意味着如果 `x` 的值匹配此分支的任何一个模式，它就会运行：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

上面的代码会打印 `one or two`。

## 通过序列 `..=` 匹配值的范围

在[数值类型](#)中我们有讲到一个序列语法，该语法不仅可以用于循环中，还能用于匹配模式。

`..=` 语法允许你匹配一个闭区间序列内的值。在如下代码中，当模式匹配任何在此序列内的值时，该分支会执行：

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}
```

如果 `x` 是 1、2、3、4 或 5，第一个分支就会匹配。这相比使用 `|` 运算符表达相同的意思更为方便；相比 `1..=5`，使用 `|` 则不得不指定 `1 | 2 | 3 | 4 | 5` 这五个值，而使用 `..=` 指定序列就简短的多，比如希望匹配比如从 1 到 1000 的数字的时候！

序列只允许用于数字或字符类型，原因是：它们可以连续，同时编译器在编译期可以检查该序列是否为空，字符和数字值是 Rust 中仅有的可以用于判断是否为空的类型。

如下是一个使用字符类型序列的例子：

```
let x = 'c';

match x {
    'a'..'j' => println!("early ASCII letter"),
    'k'..'z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Rust 知道 '`c`' 位于第一个模式的序列内，所以会打印出 `early ASCII letter`。

## 解构并分解值

也可以使用模式来解构结构体、枚举、元组、数组和引用。

### 解构结构体

下面代码展示了如何用 `let` 解构一个带有两个字段 `x` 和 `y` 的结构体 `Point`：

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

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

这段代码创建了变量 `a` 和 `b` 来匹配结构体 `p` 中的 `x` 和 `y` 字段，这个例子展示了**模式中的变量名不必与结构体中的字段名一致**。不过通常希望变量名与字段名一致以便于理解变量来自于哪些字段。

因为变量名匹配字段名是常见的，同时因为 `let Point { x: x, y: y } = p;` 中 `x` 和 `y` 重复了，所以对于匹配结构体字段的模式存在简写：只需列出结构体字段的名称，则模式创建的变量会有相同的名称。下例与上例有着相同行为的代码，不过 `let` 模式创建的变量为 `x` 和 `y` 而不是 `a` 和 `b`：

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

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

这段代码创建了变量 `x` 和 `y`，与结构体 `p` 中的 `x` 和 `y` 字段相匹配。其结果是变量 `x` 和 `y` 包含结构体 `p` 中的值。

也可以使用字面值作为结构体模式的一部分进行解构，而不是为所有的字段创建变量。这允许我们测试一些字段为特定值的同时创建其他字段的变量。

下文展示了固定某个字段的匹配方式：

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y),
    }
}
```

首先是 `match` 第一个分支，指定匹配 `y` 为 `0` 的 `Point`；然后第二个分支在第一个分支之后，匹配 `y` 不为 `0`，`x` 为 `0` 的 `Point`；最后一个分支匹配 `x` 不为 `0`，`y` 也不为 `0` 的 `Point`。

在这个例子中，值 `p` 因为其 `x` 包含 `0` 而匹配第二个分支，因此会打印出 `On the y axis at 7`。

## 解构枚举

下面代码以 `Message` 枚举为例，编写一个 `match` 使用模式解构每一个内部值：

```

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.")
        }
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}

```

这里老生常谈一句话，模式匹配一样要类型相同，因此匹配 `Message::Move{1,2}` 这样的枚举值，就必须用 `Message::Move{x,y}` 这样的同类型模式才行。

这段代码会打印出 `Change the color to red 0, green 160, and blue 255`。尝试改变 `msg` 的值来观察其他分支代码的运行。

对于像 `Message::Quit` 这样没有任何数据的枚举成员，不能进一步解构其值。只能匹配其字面值 `Message::Quit`，因此模式中没有任何变量。

对于另外两个枚举成员，就用相同类型的模式去匹配出对应的值即可。

## 解构嵌套的结构体和枚举

目前为止，所有的例子都只匹配了深度为一级的结构体或枚举。`match` 也可以匹配嵌套的项！

例如使用下面的代码来同时支持 RGB 和 HSV 色彩模式：

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!(
                "Change the color to hue {}, saturation {}, and value {}",
                h,
                s,
                v
            )
        }
        _ => ()
    }
}
```

match 第一个分支的模式匹配一个 Message::ChangeColor 枚举成员，该枚举成员又包含了一个 Color::Rgb 的枚举成员，最终绑定了 3 个内部的 i32 值。第二个，就交给亲爱的读者来思考完成。

## 解构结构体和元组

我们甚至可以用复杂的方式来混合、匹配和嵌套解构模式。如下是一个复杂结构体的例子，其中结构体和元组嵌套在元组中，并将所有的原始类型解构出来：

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 });
```

这种将复杂类型分解匹配的方式，可以让我们单独得到感兴趣的某个值。

## 解构数组

对于数组，我们可以用类似元组的方式解构，分为两种情况：

### 定长数组

```
let arr: [u16; 2] = [114, 514];  
let [x, y] = arr;  
  
assert_eq!(x, 114);  
assert_eq!(y, 514);
```

### 不定长数组

```
let arr: &[u16] = &[114, 514];  
  
if let [x, ..] = arr {  
    assert_eq!(x, &114);  
}  
  
if let &[.., y] = arr {  
    assert_eq!(y, 514);  
}  
  
let arr: &[u16] = &[];  
  
assert!(matches!(arr, [..]));  
assert!(!matches!(arr, [x, ..]));
```

## 忽略模式中的值

有时忽略模式中的一些值是很有用的，比如在 `match` 中的最后一个分支使用 `_` 模式匹配所有剩余的值。你也可以在另一个模式中使用 `_` 模式，使用一个以下划线开始的名称，或者使用 `..` 忽略所剩部分的值。

## 使用 \_ 忽略整个值

虽然 \_ 模式作为 match 表达式最后的分支特别有用，但是它的作用还不限于此。例如可以将其用于函数参数中：

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

这段代码会完全忽略作为第一个参数传递的值 3，并会打印出 This code only uses the y parameter: 4。

大部分情况当你不再需要特定函数参数时，最好修改签名不再包含无用的参数。在一些情况下忽略函数参数会变得特别有用，比如实现特征时，当你需要特定类型签名但是函数实现并不需要某个参数时。此时编译器就**不会警告说存在未使用的函数参数**，就跟使用命名参数一样。

## 使用嵌套的 \_ 忽略部分值

可以在一个模式内部使用 \_ 忽略部分值：

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

这段代码会打印出 Can't overwrite an existing customized value 接着是 setting is Some(5)。

第一个匹配分支，我们不关心里面的值，只关心元组中两个元素的类型，因此对于 Some 中的值，直接进行忽略。剩下的形如 (Some(\_), None) , (None, Some(\_)) , (None, None) 形式，都由第二个分支 \_ 进行分配。

还可以在一个模式中的多处使用下划线来忽略特定值，如下所示，这里忽略了一个五元元组中的第二和第四个值：

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}
```

老生常谈：模式匹配一定要类型相同，因此匹配 `numbers` 元组的模式，也必须有五个值（元组中元素的数量也属于元组类型的一部分）。

这会打印出 `Some numbers: 2, 8, 32`，值 4 和 16 会被忽略。

## 使用下划线开头忽略未使用的变量

如果你创建了一个变量却不在任何地方使用它，Rust 通常会给你一个警告，因为这可能会是个 BUG。但是有时创建一个不会被使用的变量是有用的，比如你正在设计原型或刚刚开始一个项目。这时你希望告诉 Rust 不要警告未使用的变量，为此可以用下划线作为变量名的开头：

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

这里得到了警告说未使用变量 `y`，至于 `x` 则没有警告。

注意，只使用 `_` 和使用以下划线开头的名称有些微妙的不同：比如 `_x` 仍会将值绑定到变量，而 `_` 则完全不会绑定。

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

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

`s` 是一个拥有所有权的动态字符串，在上面代码中，我们会得到一个错误，因为 `s` 的值会被转移给 `_s`，在 `println!` 中再次使用 `s` 会报错：

```
error[E0382]: borrow of partially moved value: `s`
--> src/main.rs:8:22
|
4 |     if let Some(_s) = s {
|         -- value partially moved here
...
8 |     println!("{}:?", s);
|         ^ value borrowed here after partial move
```

只使用下划线本身，则并不会绑定值，因为 `s` 没有被移动进 `_`：

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

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

## 用 .. 忽略剩余值

对于有多个部分的值，可以使用 `..` 语法来只使用部分值而忽略其它值，这样也不用再为每一个被忽略的值都单独列出下划线。`..` 模式会忽略模式中剩余的任何没有显式匹配的值部分。

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

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

这里列出了 `x` 值，接着使用了 `..` 模式来忽略其它字段，这样的写法要比一一列出其它字段，然后用 `_` 忽略简洁的多。

还可以用 `..` 来忽略元组中间的某些值：

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}
```

这里用 `first` 和 `last` 来匹配第一个和最后一个值。`..` 将匹配并忽略中间的所有值。

然而使用 `..` 必须是无歧义的。如果期望匹配和忽略的值是不明确的，Rust 会报错。下面代码展示了一个带有歧义的 `..` 例子，因此不能编译：

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}
```

如果编译上面的例子，会得到下面的错误：

```
error: `..` can only be used once per tuple pattern // 每个元组模式只能使用一个 `..`
--> src/main.rs:5:22
|
5 |     (.., second, ..) => {
|     --           ^^^ can only be used once per tuple pattern
|     |
|     previously used here // 上一次使用在这里
|
error: could not compile `world_hello` due to previous error
```

Rust 无法判断，`second` 应该匹配 `numbers` 中的第几个元素，因此这里使用两个 `..` 模式，是有很大歧义的！

## 匹配守卫提供的额外条件

**匹配守卫** (*match guard*) 是一个位于 `match` 分支模式之后的额外 `if` 条件，它能为分支模式提供更进一步的匹配条件。

这个条件可以使用模式中创建的变量：

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}!", x),
    None => (),
}
```

这个例子会打印出 `less than five: 4`。当 `num` 与模式中第一个分支匹配时，`Some(4)` 可以与 `Some(x)` 匹配，接着匹配守卫检查 `x` 值是否小于 5，因为 4 小于 5，所以第一个分支被选择。

相反如果 `num` 为 `Some(10)`，因为 10 不小于 5，所以第一个分支的匹配守卫为假。接着 Rust 会前往第二个分支，因为这里没有匹配守卫所以会匹配任何 `Some` 成员。

模式中无法提供类如 `if x < 5` 的表达能力，我们可以通过匹配守卫的方式来实现。

在之前，我们提到可以使用匹配守卫来解决模式中变量覆盖的问题，那里 `match` 表达式的模式中新建了一个变量而不是使用 `match` 之外的同名变量。内部变量覆盖了外部变量，意味着此时不能够使用外部变量的值，下面代码展示了如何使用匹配守卫修复这个问题。

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

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {}", n),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}", x, y);
}
```

现在这会打印出 `Default case, x = Some(5)`。现在第二个匹配分支中的模式不会引入一个覆盖外部 `y` 的新变量 `y`，这意味着可以在匹配守卫中使用外部的 `y`。相比指定会覆盖外部 `y` 的模式 `Some(y)`，这里指定为 `Some(n)`。此新建的变量 `n` 并没有覆盖任何值，因为 `match` 外部没有变量 `n`。

匹配守卫 `if n == y` 并不是一个模式所以没有引入新变量。这个 `y` 正是外部的 `y` 而不是新的覆盖变量 `y`，这样就可以通过比较 `n` 和 `y` 来表达寻找一个与外部 `y` 相同的值的概念了。

也可以在匹配守卫中使用 **或** 运算符 `|` 来指定多个模式，同时匹配守卫的条件会作用于所有的模式。下面代码展示了匹配守卫与 `|` 的优先级。这个例子中看起来好像 `if y` 只作用于 6，但实际上匹配守卫 `if y` 作用于 4、5 和 6，在满足 `x` 属于 `4 | 5 | 6` 后才会判断 `y` 是否为 `true`：

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

这个匹配条件表明此分支只匹配 `x` 值为 4、5 或 6 同时 `y` 为 `true` 的情况。

虽然在第一个分支中，`x` 匹配了模式 4，但是对于匹配守卫 `if y` 来说，因为 `y` 是 `false`，因此该守卫条件的值永远是 `false`，也意味着第一个分支永远无法被匹配。

下面的文字图解释了匹配守卫作用于多个模式时的优先级规则，第一张是正确的：

`(4 | 5 | 6) if y => ...`

而第二张图是错误的

`4 | 5 | (6 if y) => ...`

可以通过运行代码时的情况看出这一点：如果匹配守卫只作用于由 `|` 运算符指定的值列表的最后一个值，这个分支就会匹配且程序会打印出 yes。

## @绑定

`@` (读作 at) 运算符允许为一个字段绑定另外一个变量。下面例子中，我们希望测试 `Message::Hello` 的 `id` 字段是否位于 `3..=7` 范围内，同时也希望能将其值绑定到 `id_variable` 变量中以便此分支中相关的代码可以使用它。我们可以将 `id_variable` 命名为 `id`，与字段同名，不过出于示例的目的这里选择了不同的名称。

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3..=7 } => {
        println!("Found an id in range: {}", id_variable)
    },
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```

上例会打印出 `Found an id in range: 5`。通过在 `3..=7` 之前指定 `id_variable @`，我们捕获了任何匹配此范围的值并同时将该值绑定到变量 `id_variable` 上。

第二个分支只在模式中指定了一个范围，`id` 字段的值可以是 `10`、`11` 或 `12`，不过这个模式的代码并不知情也不能使用 `id` 字段中的值，因为没有将 `id` 值保存进一个变量。

最后一个分支指定了一个没有范围的变量，此时确实拥有可以用于分支代码的变量 `id`，因为这里使用了结构体字段简写语法。不过此分支中没有像头两个分支那样对 `id` 字段的值进行测试：任何值都会匹配此分支。

当你既想要限定分支范围，又想要使用分支的变量时，就可以用 `@` 来绑定到一个新的变量上，实现想要的功能。

## @前绑定后解构(Rust 1.56 新增)

使用 `@` 还可以在绑定新变量的同时，对目标进行解构：

```

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // 绑定新变量 `p`，同时对 `Point` 进行解构
    let p @ Point {x: px, y: py} = Point {x: 10, y: 23};
    println!("x: {}, y: {}", px, py);
    println!("{:?}", p);

    let point = Point {x: 10, y: 5};
    if let p @ Point {x: 10, y} = point {
        println!("x is 10 and y is {} in {:?}", y, p);
    } else {
        println!("x was not 10 :(");
    }
}

```

## @新特性(Rust 1.53 新增)

考慮下面一段代码:

```

fn main() {
    match 1 {
        num @ 1 | 2 => {
            println!("{} ", num);
        }
        _ => {}
    }
}

```

编译不通过，是因为 `num` 没有绑定到所有的模式上，只绑定了模式 `1`，你可能会试图通过这种方式来解决：

```
num @ (1 | 2)
```

但是，如果你用的是 Rust 1.53 之前的版本，那这种写法会报错，因为编译器不支持。

至此，模式匹配的内容已经全部完结，复杂但是详尽，想要一次性全部记住属实不易，因此读者可以先留一个印象，等未来需要时，再来翻阅寻找具体的模式实现方式。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 方法 Method

从面向对象语言过来的同学对于方法肯定不陌生，`class` 里面就充斥着方法的概念。在 Rust 中，方法的概念也大差不差，往往和对象成对出现：

```
object.method()
```

例如读取一个文件写入缓冲区，如果用函数的写法 `read(f, buffer)`，用方法的写法 `f.read(buffer)`。不过与其它语言 `class` 跟方法的联动使用不同（这里可能要修改下），Rust 的方法往往跟结构体、枚举、特征(Trait)一起使用，特征将在后面几章进行介绍。

## 定义方法

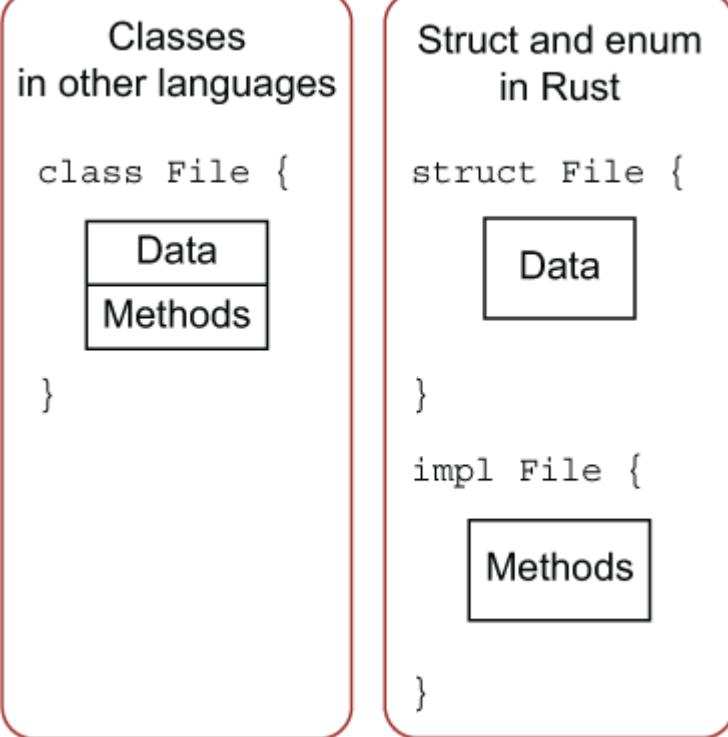
Rust 使用 `impl` 来定义方法，例如以下代码：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    // new是Circle的关联函数，因为它的第一个参数不是self，且new并不是关键字
    // 这种方法往往用于初始化当前结构体的实例
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }

    // Circle的方法，&self表示借用当前的Circle结构体
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

我们这里先不详细展开讲解，只是先建立对方法定义的大致印象。下面的图片将 Rust 方法定义与其它语言的方法定义做了对比：



可以看出，其它语言中所有定义都在 `class` 中，但是 Rust 的对象定义和方法定义是分离的，这种数据和使用分离的方式，会给予使用者极高的灵活度。

再来看一个例子：

```

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

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

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

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

```

该例子定义了一个 `Rectangle` 结构体，并且在其上定义了一个 `area` 方法，用于计算该矩形的面积。

`impl Rectangle {}` 表示为 `Rectangle` 实现方法(`impl` 是实现 *implementation* 的缩写)，这样的写法表明 `impl` 语句块中的一切都是跟 `Rectangle` 相关联的。

## **self、&self 和 &mut self**

接下来的内容非常重要，请大家仔细看。在 `area` 的签名中，我们使用 `&self` 替代 `rectangle: &Rectangle`，`&self` 其实是 `self: &Self` 的简写（注意大小写）。在一个 `impl` 块内，`Self` 指代被实现方法的结构体类型，`self` 指代此类型的实例，换句话说，`self` 指代的是 `Rectangle` 结构体实例，这样的写法会让我们的代码简洁很多，而且非常便于理解：我们为哪个结构体实现方法，那么 `self` 就是指代哪个结构体的实例。

需要注意的是，`self` 依然有所有权的概念：

- `self` 表示 `Rectangle` 的所有权转移到该方法中，这种形式用的较少
- `&self` 表示该方法对 `Rectangle` 的不可变借用
- `&mut self` 表示可变借用

总之，`self` 的使用就跟函数参数一样，要严格遵守 Rust 的所有权规则。

回到上面的例子中，选择 `&self` 的理由跟在函数中使用 `&Rectangle` 是相同的：我们并不想获取所有权，也无需去改变它，只是希望能够读取结构体中的数据。如果想要在方法中去改变当前的结构体，需要将第一个参数改为 `&mut self`。仅仅通过使用 `self` 作为第一个参数来使方法获取实例的所有权是很少见的，这种使用方式往往用于把当前的对象转成另外一个对象时使用，转换完后，就不再关注之前的对象，且可以防止对之前对象的误调用。

简单总结下，使用方法代替函数有以下好处：

- 不用在函数签名中重复书写 `self` 对应的类型
- 代码的组织性和内聚性更强，对于代码维护和阅读来说，好处巨大

## **方法名跟结构体字段名相同**

在 Rust 中，允许方法名跟结构体的字段名相同：

```

impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

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

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}

```

当我们使用 `rect1.width()` 时，Rust 知道我们调用的是它的方法，如果使用 `rect1.width`，则是访问它的字段。

一般来说，方法跟字段同名，往往适用于实现 `getter` 访问器，例如：

```

pub struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    pub fn new(width: u32, height: u32) -> Self {
        Rectangle { width, height }
    }
    pub fn width(&self) -> u32 {
        return self.width;
    }
}

fn main() {
    let rect1 = Rectangle::new(30, 50);

    println!("{}", rect1.width());
}

```

用这种方式，我们可以把 `Rectangle` 的字段设置为私有属性，只需把它的 `new` 和 `width` 方法设置为公开可见，那么用户就可以创建一个矩形，同时通过访问器 `rect1.width()` 方法来获取矩形的宽度，因为 `width` 字段是私有的，当用户访问 `rect1.width` 字段时，就会报错。注意在此例中，`Self` 指代的就是被实现方法的结构体 `Rectangle`。

---

## -> 运算符到哪去了？

在 C/C++ 语言中，有两个不同的运算符来调用方法：. 直接在对象上调用方法，而 -> 在一个对象的指针上调用方法，这时需要先解引用指针。换句话说，如果 `object` 是一个指针，那么 `object->something()` 和 `(*object).something()` 是一样的。

Rust 并没有一个与 -> 等效的运算符；相反，Rust 有一个叫 **自动引用和解引用** 的功能。方法调用是 Rust 中少数几个拥有这种行为的地方。

他是这样工作的：当使用 `object.something()` 调用方法时，Rust 会自动为 `object` 添加 &、`&mut` 或 \* 以便使 `object` 与方法签名匹配。也就是说，这些代码是等价的：

```
p1.distance(&p2);  
(&p1).distance(&p2);
```

第一行看起来简洁的多。这种自动引用的行为之所以有效，是因为方法有一个明确的接收者—— `self` 的类型。在给出接收者和方法名的前提下，Rust 可以明确地计算出方法是仅仅读取 (`&self`)，做出修改 (`&mut self`) 或者是获取所有权 (`self`)。事实上，Rust 对方法接收者的隐式借用让所有权在实践中更友好。

---

## 带多个参数的方法

方法和函数一样，可以使用多个参数：

```

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };
    let rect3 = Rectangle { width: 60, height: 45 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}

```

## 关联函数

现在大家可以思考一个问题，如何为一个结构体定义一个构造器方法？也就是接受几个参数，然后构造并返回该结构体的实例。其实答案在开头的代码片段中就给出了，很简单，参数中不包含 `self` 即可。

这种定义在 `impl` 中且没有 `self` 的函数被称之为**关联函数**：因为它没有 `self`，不能用 `f.read()` 的形式调用，因此它是一个函数而不是方法，它又在 `impl` 中，与结构体紧密关联，因此称为关联函数。

在之前的代码中，我们已经多次使用过关联函数，例如 `String::from`，用于创建一个动态字符串。

```

impl Rectangle {
    fn new(w: u32, h: u32) -> Rectangle {
        Rectangle { width: w, height: h }
    }
}

```

---

Rust 中有一个约定俗成的规则，使用 `new` 来作为构造器的名称，出于设计上的考虑，Rust 特地没有用 `new` 作为关键字。

---

因为是函数，所以不能用 `.` 的方式来调用，我们需要用 `::` 来调用，例如 `let sq = Rectangle::new(3, 3);`。这个方法位于结构体的命名空间中：`::` 语法用于关联函数和模块创建的命名空间。

## 多个 impl 定义

Rust 允许我们为一个结构体定义多个 `impl` 块，目的是提供更多的灵活性和代码组织性，例如当方法多了后，可以把相关的方法组织在同一个 `impl` 块中，那么就可以形成多个 `impl` 块，各自完成一块儿目标：

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

当然，就这个例子而言，我们没必要使用两个 `impl` 块，这里只是为了演示方便。

## 为枚举实现方法

枚举类型之所以强大，不仅仅在于它好用、可以同一化类型，还在于，我们可以像结构体一样，为枚举实现方法：

```
#![allow(unused)]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message {
    fn call(&self) {
        // 在这里定义方法体
    }
}

fn main() {
    let m = Message::Write(String::from("hello"));
    m.call();
}
```

除了结构体和枚举，我们还能为特征(trait)实现方法，这将在下一章进行讲解，在此之前，先来看看泛型。

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 泛型和特征

泛型和特征是 Rust 中最最重要的抽象类型，也是你在学习 Rust 路上的拦路虎，但是挑战往往与乐趣并存，一旦学会，在后面学习 Rust 的路上，你将一往无前。

# 泛型 Generics

Go 语言在 2022 年，就要正式引入泛型，被视为在 1.0 版本后，语言特性发展迈出的一大步，为什么泛型这么重要？到底什么是泛型？Rust 的泛型有几种？本章将一一为你讲解。

我们在编程中，经常有这样的需求：用同一功能的函数处理不同类型的数据，例如两个数的加法，无论是整数还是浮点数，甚至是自定义类型，都能进行支持。在不支持泛型的编程语言中，通常需要为每一种类型编写一个函数：

```
fn add_i8(a:i8, b:i8) -> i8 {
    a + b
}
fn add_i32(a:i32, b:i32) -> i32 {
    a + b
}
fn add_f64(a:f64, b:f64) -> f64 {
    a + b
}

fn main() {
    println!("add i8: {}", add_i8(2i8, 3i8));
    println!("add i32: {}", add_i32(20, 30));
    println!("add f64: {}", add_f64(1.23, 1.23));
}
```

上述代码可以正常运行，但是很啰嗦，如果你要支持更多的类型，那么会更繁琐。程序员或多或少都有强迫症，一个好程序员的公认特征就是——懒，这么勤快的写一大堆代码，显然不是咱们的优良传统，是不是？

在开始讲解 Rust 的泛型之前，先来看看什么是多态。

在编程的时候，我们经常利用多态。通俗的讲，多态就是好比坦克的炮管，既可以发射普通弹药，也可以发射制导炮弹（导弹），也可以发射贫铀穿甲弹，甚至发射子母弹，没有必要为每一种炮弹都在坦克上分别安装一个专用炮管，即使生产商愿意，炮手也不愿意，累死人啊。所以在编程开发中，我们也需要这样“通用的炮管”，这个“通用的炮管”就是多态。

实际上，泛型就是一种多态。泛型主要目的是为程序员提供编程的便利，减少代码的臃肿，同时可以极大地丰富语言本身的表达能力，为程序员提供了一个合适的炮管。想想，一个函数，可以代替几十个，甚至数百个函数，是一件多么让人兴奋的事情：

```
fn add<T>(a:T, b:T) -> T {
    a + b
}

fn main() {
    println!("add i8: {}", add(2i8, 3i8));
    println!("add i32: {}", add(20, 30));
    println!("add f64: {}", add(1.23, 1.23));
}
```

将之前的代码改成上面这样，就是 Rust 泛型的初印象，这段代码虽然很简洁，但是并不能编译通过，我们会在后面进行详细讲解，现在只要对泛型有个大概的印象即可。

## 泛型详解

上面代码的 `T` 就是**泛型参数**，实际上在 Rust 中，泛型参数的名称你可以任意起，但是出于惯例，我们都用 `T` (`T` 是 `type` 的首字母) 来作为首选，这个名称越短越好，除非需要表达含义，否则一个字母是最完美的。

使用泛型参数，有一个先决条件，必需在使用前对其进行声明：

```
fn largest<T>(list: &[T]) -> T {
```

该泛型函数的作用是从列表中找出最大的值，其中列表中的元素类型为 `T`。首先 `largest<T>` 对泛型参数 `T` 进行了声明，然后才在函数参数中进行使用该泛型参数 `list: &[T]` (还记得 `&[T]` 类型吧？这是**数组切片**！)。

总之，我们可以这样理解这个函数定义：函数 `largest` 有泛型类型 `T`，它有个参数 `list`，其类型是元素为 `T` 的数组切片，最后，该函数返回值的类型也是 `T`。

下面是一个错误的泛型函数的实现：

```

fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}

```

运行后报错：

```

error[E0369]: binary operation `>` cannot be applied to type `T` // `>`操作符不能用于类型`T`
--> src/main.rs:5:17
|
5 |         if item > largest {
|             ^ ----- T
|             |
|             T
|
help: consider restricting type parameter `T` // 考虑对T进行类型上的限制 :
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
|           ++++++ ++++++ ++++++

```

因为 `T` 可以是任何类型，但不是所有的类型都能进行比较，因此上面的错误中，编译器建议我们给 `T` 添加一个类型限制：使用 `std::cmp::PartialOrd` 特征（Trait）对 `T` 进行限制，特征在下一节会详细介绍，现在你只要理解，该特征的目的就是让**类型实现可比较的功能**。

还记得我们一开始的 `add` 泛型函数吗？如果你运行它，会得到以下的报错：

```

error[E0369]: cannot add `T` to `T` // 无法将 `T` 类型跟 `T` 类型进行相加
--> src/main.rs:2:7
2 |     a + b
|     - ^ - T
|     |
|     T
|
help: consider restricting type parameter `T`
1 | fn add<T: std::ops::Add<Output = T>>(a:T, b:T) -> T {
|         ++++++-----+

```

同样的，不是所有 `T` 类型都能进行相加操作，因此我们需要用 `std::ops::Add<Output = T>` 对 `T` 进行限制：

```

fn add<T: std::ops::Add<Output = T>>(a:T, b:T) -> T {
    a + b
}

```

进行如上修改后，就可以正常运行。

## 结构体中使用泛型

结构体中的字段类型也可以用泛型来定义，下面代码定义了一个坐标点 `Point`，它可以存放任何类型的坐标值：

```

struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}

```

这里有两点需要特别的注意：

- **提前声明**，跟泛型函数定义类似，首先我们在使用泛型参数之前必需要进行声明 `Point<T>`，接着就可以在结构体的字段类型中使用 `T` 来替代具体的类型
- **x 和 y 是相同的类型**

第二点非常重要，如果使用不同的类型，那么它会导致下面代码的报错：

```

struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let p = Point{x: 1, y :1.1};
}

```

错误如下：

```

error[E0308]: mismatched types //类型不匹配
--> src/main.rs:7:28
|
7 |     let p = Point{x: 1, y :1.1};
|                         ^^^ expected integer, found floating-point number //期
望y是整数，但是却是浮点数

```

当把 1 赋值给 x 时，变量 p 的 T 类型就被确定为整数类型，因此 y 也必须是整数类型，但是我们却给它赋予了浮点数，因此导致报错。

如果想让 x 和 y 既能类型相同，又能类型不同，就需要使用不同的泛型参数：

```

struct Point<T,U> {
    x: T,
    y: U,
}
fn main() {
    let p = Point{x: 1, y :1.1};
}

```

切记，所有的泛型参数都要提前声明：Point<T,U>！但是如果你的结构体变成这鬼样：struct Woo<T,U,V,W,X>，那么你需要考虑拆分这个结构体，减少泛型参数的个数和代码复杂度。

## 枚举中使用泛型

提到枚举类型，Option 永远是第一个应该被想起来的，在之前的章节中，它也多次出现：

```

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

```

`Option<T>` 是一个拥有泛型 `T` 的枚举类型，它第一个成员是 `Some(T)`，存放了一个类型为 `T` 的值。得益于泛型的引入，我们可以在任何一个需要返回值的函数中，去使用 `Option<T>` 枚举类型来做为返回值，用于返回一个任意类型的值 `Some(T)`，或者没有值 `None`。

对于枚举而言，卧龙凤雏永远是绕不过去的存在：如果是 `Option` 是卧龙，那么 `Result` 就一定是凤雏，得两者可得天下：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

这个枚举和 `Option` 一样，主要用于函数返回值，与 `Option` 用于值的存在与否不同，`Result` 关注的主要是值的正确性。

如果函数正常运行，则最后返回一个 `Ok(T)`，`T` 是函数具体的返回值类型，如果函数异常运行，则返回一个 `Err(E)`，`E` 是错误类型。例如打开一个文件：如果成功打开文件，则返回 `Ok(std::fs::File)`，因此 `T` 对应的是 `std::fs::File` 类型；而当打开文件时出现问题时，返回 `Err(std::io::Error)`，`E` 对应的就是 `std::io::Error` 类型。

## 方法中使用泛型

上一章中，我们讲到什么是方法以及如何在结构体和枚举上定义方法。方法上也可以使用泛型：

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

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

使用泛型参数前，依然需要提前声明：`impl<T>`，只有提前声明了，我们才能在 `Point<T>` 中使用它，这样 Rust 就知道 `Point` 的尖括号中的类型是泛型而不是具体类型。需要注意的是，这里的 `Point<T>` 不

再是泛型声明，而是一个完整的结构体类型，因为我们定义的结构体就是 `Point<T>` 而不再是 `Point`。

除了结构体中的泛型参数，我们还能在该结构体的方法中定义额外的泛型参数，就跟泛型函数一样：

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

这个例子中，`T, U` 是定义在结构体 `Point` 上的泛型参数，`V, W` 是单独定义在方法 `mixup` 上的泛型参数，它们并不冲突，说白了，你可以理解为，一个是结构体泛型，一个是函数泛型。

## 为具体的泛型类型实现方法

对于 `Point<T>` 类型，你不仅能定义基于 `T` 的方法，还能针对特定的具体类型，进行方法定义：

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

这段代码意味着 `Point<f32>` 类型会有一个方法 `distance_from_origin`，而其他 `T` 不是 `f32` 类型的 `Point<T>` 实例则没有定义此方法。这个方法计算点实例与坐标  $(0.0, 0.0)$  之间的距离，并使用了只能用于浮点型的数学运算符。

这样我们就能针对特定的泛型类型实现某个特定的方法，对于其它泛型类型则没有定义该方法。

## const 泛型 (Rust 1.51 版本引入的重要特性)

在之前的泛型中，可以抽象为一句话：针对类型实现的泛型，所有的泛型都是为了抽象不同的类型，那有没有针对值的泛型？可能很多同学感觉很难理解，值怎么使用泛型？不急，我们先从数组讲起。

在[数组](#)那节，有提到过很重要的一点：`[i32; 2]` 和 `[i32; 3]` 是不同的数组类型，比如下面的代码：

```
fn display_array(arr: [i32; 3]) {
    println!("{:?}", arr);
}

fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(arr);

    let arr: [i32; 2] = [1, 2];
    display_array(arr);
}
```

运行后报错：

```
error[E0308]: mismatched types // 类型不匹配
--> src/main.rs:10:19
 |
10 |     display_array(arr);
|           ^^^ expected an array with a fixed size of 3 elements, found
one with 2 elements
// 期望一个长度为3的数组，却发现一个长度为2的
```

结合代码和报错，可以很清楚的看出，`[i32; 3]` 和 `[i32; 2]` 确实是两个完全不同的类型，因此无法用同一个函数调用。

首先，让我们修改代码，让 `display_array` 能打印任意长度的 `i32` 数组：

```
fn display_array(arr: &[i32]) {
    println!("{:?}", arr);
}

fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(&arr);

    let arr: [i32; 2] = [1, 2];
    display_array(&arr);
}
```

很简单，只要使用数组切片，然后传入 `arr` 的不可变引用即可。

接着，将 `i32` 改成所有类型的数组：

```

fn display_array<T: std::fmt::Debug>(arr: &[T]) {
    println!("{:?}", arr);
}

fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(&arr);

    let arr: [i32; 2] = [1, 2];
    display_array(&arr);
}

```

也不难，唯一要注意的是需要对 `T` 加一个限制 `std::fmt::Debug`，该限制表明 `T` 可以用在 `println!("{:?}", arr)` 中，因为 `{:?}` 形式的格式化输出需要 `arr` 实现该特征。

通过引用，我们可以很轻松的解决处理任何类型数组的问题，但是如果在某些场景下引用不适宜用或者干脆不能用呢？你们知道为什么以前 Rust 的一些数组库，在使用的时候都限定长度不超过 32 吗？因为它会为每个长度都单独实现一个函数，简直。。。毫无人性。难道没有什么办法可以解决这个问题吗？

好在，现在咱们有了 `const` 泛型，也就是针对值的泛型，正好可以用于处理数组长度的问题：

```

fn display_array<T: std::fmt::Debug, const N: usize>(arr: [T; N]) {
    println!("{:?}", arr);
}

fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    display_array(arr);

    let arr: [i32; 2] = [1, 2];
    display_array(arr);
}

```

如上所示，我们定义了一个类型为 `[T; N]` 的数组，其中 `T` 是一个基于类型的泛型参数，这个和之前讲的泛型没有区别，而重点在于 `N` 这个泛型参数，它是一个基于值的泛型参数！因为它用来替代的是数组的长度。

`N` 就是 `const` 泛型，定义的语法是 `const N: usize`，表示 `const` 泛型 `N`，它基于的值类型是 `usize`。

在泛型参数之前，Rust 完全不适合复杂矩阵的运算，自从有了 `const` 泛型，一切即将改变。

## const 泛型表达式

假设我们某段代码需要在内存很小的平台上工作，因此需要限制函数参数占用的内存大小，此时就可以使用 `const` 泛型表达式来实现：

```

// 目前只能在nightly版本下使用
#![allow(incomplete_features)]
#![feature(generic_const_exprs)]

fn something<T>(val: T)
where
    Assert<{ core::mem::size_of::<T>() < 768 }>: IsTrue,
    //           ^-----^ 这里是一个 const 表达式, 换成其它的 const 表达式也可以
{
    //
}

fn main() {
    something([0u8; 0]); // ok
    something([0u8; 512]); // ok
    something([0u8; 1024]); // 编译错误, 数组长度是1024字节, 超过了768字节的参数长度限制
}

// ---

pub enum Assert<const CHECK: bool> {
    //
}

pub trait IsTrue {
    //
}

impl IsTrue for Assert<true> {
    //
}

```

## const fn

@todo

## 泛型的性能

在 Rust 中泛型是零成本的抽象，意味着你在使用泛型时，完全不用担心性能上的问题。

但是任何选择都是权衡得失的，既然我们获得了性能上的巨大优势，那么又失去了什么呢？Rust 是在编译期为泛型对应的多个类型，生成各自的代码，因此损失了编译速度和增大了最终生成文件的大小。

具体来说：

Rust 通过在编译时进行泛型代码的 **单态化**(monomorphization)来保证效率。单态化是一个通过填充编译时使用的具体类型，将通用代码转换为特定代码的过程。

编译器所做的工作正好与我们创建泛型函数的步骤相反，编译器寻找所有泛型代码被调用的位置并针对具体类型生成代码。

让我们看看一个使用标准库中 `Option` 枚举的例子：

```
let integer = Some(5);
let float = Some(5.0);
```

当 Rust 编译这些代码的时候，它会进行单态化。编译器会读取传递给 `Option<T>` 的值并发现有两种 `Option<T>`：一种对应 `i32` 另一种对应 `f64`。为此，它会将泛型定义 `Option<T>` 展开为 `Option_i32` 和 `Option_f64`，接着将泛型定义替换为这两个具体的定义。

编译器生成的单态化版本的代码看起来像这样：

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

我们可以使用泛型来编写不重复的代码，而 Rust 将会为每一个实例编译其特定类型的代码。这意味着在使用泛型时没有运行时开销；当代码运行，它的执行效率就跟好像手写每个具体定义的重复代码一样。这个单态化过程正是 Rust 泛型在运行时极其高效的原因。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

- [泛型](#)
  - [习题解答](#)

- `const` 泛型
    - 习题解答
-

# 特征 Trait

如果我们想定义一个文件系统，那么把该系统跟底层存储解耦是很重要的。文件操作主要包含四个：`open`、`write`、`read`、`close`，这些操作可以发生在硬盘，可以发生在内存，还可以发生在网络IO甚至（...我实在编不下去了，大家来帮帮我）。总之如果你要为每一种情况都单独实现一套代码，那这种实现将过于繁杂，而且也没那个必要。

要解决上述问题，需要把这些行为抽象出来，就要使用 Rust 中的特征 `trait` 概念。可能你是第一次听说这个名词，但是不要怕，如果学过其他语言，那么大概率你听说过接口，没错，特征跟接口很类似。

在之前的代码中，我们也多次见过特征的使用，例如 `#[derive(Debug)]`，它在我们定义的类型(`struct`)上自动派生 `Debug` 特征，接着可以使用 `println!("{}:?", x)` 打印这个类型；再例如：

```
fn add<T: std::ops::Add<Output = T>>(a:T, b:T) -> T {  
    a + b  
}
```

通过 `std::ops::Add` 特征来限制 `T`，只有 `T` 实现了 `std::ops::Add` 才能进行合法的加法操作，毕竟不是所有的类型都能进行相加。

这些都说明一个道理，特征定义了一组可以被共享的行为，只要实现了特征，你就能使用这组行为。

## 定义特征

如果不同的类型具有相同的行为，那么我们就可以定义一个特征，然后为这些类型实现该特征。**定义特征**是把一些方法组合在一起，目的是定义一个实现某些目标所必需的行为的集合。

例如，我们现在有文章 `Post` 和微博 `Weibo` 两种内容载体，而我们想对相应的内容进行总结，也就是无论是文章内容，还是微博内容，都可以在某个时间点进行总结，那么总结这个行为就是共享的，因此可以用特征来定义：

```
pub trait Summary {  
    fn summarize(&self) -> String;  
}
```

这里使用 `trait` 关键字来声明一个特征，`Summary` 是特征名。在大括号中定义了该特征的所有方法，在这个例子中是：`fn summarize(&self) -> String`。

特征只定义行为看起来是什么样的，而不定义行为具体是怎么样的。因此，我们只定义特征方法的签名，而不进行实现，此时方法签名结尾是 ;，而不是一个 {}。

接下来，每一个实现这个特征的类型都需要具体实现该特征的相应方法，编译器也会确保任何实现 Summary 特征的类型都拥有与这个签名的定义完全一致的 summarize 方法。

## 为类型实现特征

因为特征只定义行为看起来是什么样的，因此我们需要为类型实现具体的特征，定义行为具体是怎么样的。

首先来为 Post 和 Weibo 实现 Summary 特征：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Post {
    pub title: String, // 标题
    pub author: String, // 作者
    pub content: String, // 内容
}

impl Summary for Post {
    fn summarize(&self) -> String {
        format!("文章{}, 作者是{}", self.title, self.author)
    }
}

pub struct Weibo {
    pub username: String,
    pub content: String
}

impl Summary for Weibo {
    fn summarize(&self) -> String {
        format!("{}发表了微博{}", self.username, self.content)
    }
}
```

实现特征的语法与为结构体、枚举实现方法很像：impl Summary for Post，读作“为 Post 类型实现 Summary 特征”，然后在 impl 的花括号中实现该特征的具体方法。

接下来就可以在这个类型上调用特征的方法：

```
fn main() {
    let post = Post{title: "Rust语言简介".to_string(),author: "Sunface".to_string(),
content: "Rust棒极了!".to_string()};
    let weibo = Weibo{username: "sunface".to_string(),content: "好像微博没Tweet好用".to_string()};

    println!("{}" ,post.summarize());
    println!("{}" ,weibo.summarize());
}
```

运行输出：

```
文章 Rust 语言简介，作者是Sunface
sunface发表了微博好像微博没Tweet好用
```

说实话，如果特征仅仅如此，你可能会觉得花里胡哨没啥用，接下来就让你见识下 trait 真正的威力。

### 特征定义与实现的位置(孤儿规则)

上面我们将 Summary 定义成了 pub 公开的。这样，如果他人想要使用我们的 Summary 特征，则可以引入到他们的包中，然后再进行实现。

关于特征实现与定义的位置，有一条非常重要的原则：**如果你想要为类型 A 实现特征 T，那么 A 或者 T 至少有一个是在当前作用域中定义的！** 例如我们可以为上面的 Post 类型实现标准库中的 Display 特征，这是因为 Post 类型定义在当前的作用域中。同时，我们也可以在当前包中为 String 类型实现 Summary 特征，因为 Summary 定义在当前作用域中。

但是你无法在当前作用域中，为 String 类型实现 Display 特征，因为它们俩都定义在标准库中，其定义所在的位置都不在当前作用域，跟你半毛钱关系都没有，看看就行了。

该规则被称为**孤儿规则**，可以确保其它人编写的代码不会破坏你的代码，也确保了你不会莫名其妙就破坏了风马牛不相及的代码。

### 默认实现

你可以在特征中定义具有**默认实现**的方法，这样其它类型无需再实现该方法，或者也可以选择重载该方法：

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

上面为 `Summary` 定义了一个默认实现，下面我们编写段代码来测试下：

```
impl Summary for Post {}

impl Summary for Weibo {
    fn summarize(&self) -> String {
        format!("{}发表了微博{}", self.username, self.content)
    }
}
```

可以看到，`Post` 选择了默认实现，而 `Weibo` 重载了该方法，调用和输出如下：

```
println!("{}", post.summarize());
println!("{}", weibo.summarize());
```

```
(Read more...)
sunface发表了微博好像微博没Tweet好用
```

默认实现允许调用相同特征中的其他方法，哪怕这些方法没有默认实现。如此，特征可以提供很多有用的功能而只需要实现指定的一小部分内容。例如，我们可以定义 `Summary` 特征，使其具有一个需要实现的 `summarize_author` 方法，然后定义一个 `summarize` 方法，此方法的默认实现调用 `summarize_author` 方法：

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

为了使用 `Summary`，只需要实现 `summarize_author` 方法即可：

```
impl Summary for Weibo {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
println!("1 new weibo: {}", weibo.summarize());
```

`weibo.summarize()` 会先调用 `Summary` 特征默认实现的 `summarize` 方法，通过该方法进而调用 `Weibo` 为 `Summary` 实现的 `summarize_author` 方法，最终输出：1 new weibo: (Read more from @horse\_ebooks...)。

## 使用特征作为函数参数

之前提到过，特征如果仅仅是用来实现方法，那真的有些大材小用，现在我们来讲下，真正可以让特征大放光彩的地方。

现在，先定义一个函数，使用特征作为函数参数：

```
pub fn notify(item: &impl Summary) {  
    println!("Breaking news! {}", item.summarize());  
}
```

`impl Summary`，只能说想出这个类型的人真的是起名鬼才，简直太贴切了，顾名思义，它的意思是 **实现了 Summary 特征的 item 参数**。

你可以使用任何实现了 `Summary` 特征的类型作为该函数的参数，同时在函数体内，还可以调用该特征的方法，例如 `summarize` 方法。具体的说，可以传递 `Post` 或 `Weibo` 的实例来作为参数，而其它类如 `String` 或者 `i32` 的类型则不能用做该函数的参数，因为它们没有实现 `Summary` 特征。

## 特征约束(trait bound)

虽然 `impl Trait` 这种语法非常好理解，但是实际上它只是一个语法糖：

```
pub fn notify<T: Summary>(item: &T) {  
    println!("Breaking news! {}", item.summarize());  
}
```

真正的完整书写形式如上所述，形如 `T: Summary` 被称为**特征约束**。

在简单的场景下 `impl Trait` 这种语法糖就足够使用，但是对于复杂的场景，特征约束可以让我们拥有更大的灵活性和语法表现能力，例如一个函数接受两个 `impl Summary` 的参数：

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {}
```

如果函数两个参数是不同的类型，那么上面的方法很好，只要这两个类型都实现了 `Summary` 特征即可。但是如果我们要强制函数的两个参数是同一类型呢？上面的语法就无法做到这种限制，此时我们只能使特征约束来实现：

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {}
```

泛型类型 `T` 说明了 `item1` 和 `item2` 必须拥有同样的类型，同时 `T: Summary` 说明了 `T` 必须实现 `Summary` 特征。

## 多重约束

除了单个约束条件，我们还可以指定多个约束条件，例如除了让参数实现 `Summary` 特征外，还可以让参数实现 `Display` 特征以控制它的格式化输出：

```
pub fn notify(item: &(impl Summary + Display)) {}
```

除了上述的语法糖形式，还能使用特征约束的形式：

```
pub fn notify<T: Summary + Display>(item: &T) {}
```

通过这两个特征，就可以使用 `item.summarize` 方法，以及通过 `println!("{}"`, `item`) 来格式化输出 `item`。

## Where 约束

当特征约束变得很多时，函数的签名将变得很复杂：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
```

严格来说，上面的例子还是不够复杂，但是我们还是能对其做一些形式上的改进，通过 `where`：

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{}
```

## 使用特征约束有条件地实现方法或特征

特征约束，可以让我们在指定类型 + 指定特征的条件下实现方法，例如：

```

use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}

```

`cmp_display` 方法，并不是所有的 `Pair<T>` 结构体对象都可以拥有，只有 `T` 同时实现了 `Display + PartialOrd` 的 `Pair<T>` 才可以拥有此方法。该函数可读性会更好，因为泛型参数、参数、返回值都在一起，可以快速的阅读，同时每个泛型参数的特征也在新的代码行中通过**特征约束**进行了约束。

**也可以有条件地实现特征**，例如，标准库为任何实现了 `Display` 特征的类型实现了 `Tostring` 特征：

```

impl<T: Display> ToString for T {
    // --snip--
}

```

我们可以对任何实现了 `Display` 特征的类型调用由 `ToString` 定义的 `to_string` 方法。例如，可以将整型转换为对应的 `String` 值，因为整型实现了 `Display`：

```
let s = 3.to_string();
```

## 函数返回中的 `impl Trait`

可以通过 `impl Trait` 来说明一个函数返回了一个类型，该类型实现了某个特征：

```

fn returns_summarizable() -> impl Summary {
    Weibo {
        username: String::from("sunface"),
        content: String::from(
            "m1 max太厉害了，电脑再也不会卡",
        )
    }
}

```

因为 Weibo 实现了 Summary，因此这里可以用它来作为返回值。要注意的是，虽然我们知道这里是一个 Weibo 类型，但是对于 returns\_summarizable 的调用者而言，他只知道返回了一个实现了 Summary 特征的对象，但是并不知道返回了一个 Weibo 类型。

这种 impl Trait 形式的返回值，在一种场景下非常非常有用，那就是返回的真实类型非常复杂，你不知道该怎么声明时(毕竟 Rust 要求你必须标出所有的类型)，此时就可以用 impl Trait 的方式简单返回。例如，闭包和迭代器就是很复杂，只有编译器才知道那玩意的真实类型，如果让你写出来它们的具体类型，估计内心有一万只草泥马奔腾，好在你可以用 impl Iterator 来告诉调用者，返回了一个迭代器，因为所有迭代器都会实现 Iterator 特征。

但是这种返回值方式有一个很大的限制：只能有一个具体的类型，例如：

```

fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        Post {
            title: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                hockey team in the NHL.",
            ),
        }
    } else {
        Weibo {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
        }
    }
}

```

以上的代码就无法通过编译，因为它返回了两个不同的类型 Post 和 Weibo。

```

`if` and `else` have incompatible types
expected struct `Post`, found struct `Weibo`

```

报错提示我们 `if` 和 `else` 返回了不同的类型。如果想要实现返回不同的类型，需要使用下一章节中的特征对象。

## 修复上一节中的 `largest` 函数

还记得上一节中的[例子](#)吧，当时留下一个疑问，该如何解决编译报错：

```
error[E0369]: binary operation `>` cannot be applied to type `T` // 无法在 `T` 类型上应用`>`运算符
--> src/main.rs:5:17
|
5 |     if item > largest {
|     ^----- T
|     |
|     T
|
help: consider restricting type parameter `T` // 考虑使用以下的特征来约束 `T`
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
|           ^^^^^^
```

在 `largest` 函数体中我们想要使用大于运算符 (`>`) 比较两个 `T` 类型的值。这个运算符是标准库中特征 `std::cmp::PartialOrd` 的一个默认方法。所以需要在 `T` 的特征约束中指定 `PartialOrd`，这样 `largest` 函数可以用于内部元素类型可比较大小的数组切片。

由于 `PartialOrd` 位于 `prelude` 中所以并不需要通过 `std::cmp` 手动将其引入作用域。所以可以将 `largest` 的签名修改为如下：

```
fn largest<T: PartialOrd>(list: &[T]) -> T {}
```

但是此时编译，又会出现新的错误：

```

error[E0508]: cannot move out of type `[T]`, a non-copy slice
--> src/main.rs:2:23
2 |     let mut largest = list[0];
  |     ^^^^^^^^
  |     |
  |     cannot move out of here
  |     help: consider using a reference instead: `&list[0]`


error[E0507]: cannot move out of borrowed content
--> src/main.rs:4:9
4 |     for &item in list.iter() {
  |     ^
  |     ||
  |     |hint: to prevent move, use `ref item` or `ref mut item`
  |     cannot move out of borrowed content

```

错误的核心是 `cannot move out of type [T]`, a non-copy slice, 原因是 `T` 没有实现 `Copy` 特性, 因此我们只能把所有权进行转移, 毕竟只有 `i32` 等基础类型才实现了 `Copy` 特性, 可以存储在栈上, 而 `T` 可以指代任何类型 (严格来说是实现了 `PartialOrd` 特征的所有类型)。

因此, 为了让 `T` 拥有 `Copy` 特性, 我们可以增加特征约束:

```

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}

```

如果并不希望限制 `largest` 函数只能用于实现了 `Copy` 特征的类型，我们可以在 `T` 的特征约束中指定 `Clone` 特征 而不是 `Copy` 特征。并克隆 `list` 中的每一个值使得 `largest` 函数拥有其所有权。使用 `clone` 函数意味着对于类似 `String` 这样拥有堆上数据的类型，会潜在地分配更多堆上空间，而堆分配在涉及大量数据时可能会相当缓慢。

另一种 `largest` 的实现方式是返回在 `list` 中 `T` 值的引用。如果我们将函数返回值从 `T` 改为 `&T` 并改变函数体使其能够返回一个引用，我们将不需要任何 `Clone` 或 `Copy` 的特征约束而且也不会有任何的堆分配。尝试自己实现这种替代解决方式吧！

## 通过 `derive` 派生特征

在本书中，形如 `#[derive(Debug)]` 的代码已经出现了很多次，这种是一种特征派生语法，被 `derive` 标记的对象会自动实现对应的默认特征代码，继承相应的功能。

例如 `Debug` 特征，它有一套自动实现的默认代码，当你给一个结构体标记后，就可以使用 `println!("{}:{}")`, `s` 的形式打印该结构体的对象。

再如 `Copy` 特征，它也有一套自动实现的默认代码，当标记到一个类型上时，可以让这个类型自动实现 `Copy` 特征，进而可以调用 `copy` 方法，进行自我复制。

总之，`derive` 派生出来的是 Rust 默认给我们提供的特征，在开发过程中极大的简化了自己手动实现相应特征的需求，当然，如果你有特殊的需求，还可以自己手动重载该实现。

详细的 `derive` 列表参见[附录-派生特征](#)。

## 调用方法需要引入特征

在一些场景中，使用 `as` 关键字做类型转换会有比较大的限制，因为你想要在类型转换上拥有完全的控制，例如处理转换错误，那么你将需要 `TryInto`：

```
use std::convert::TryInto;

fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    let b_ = b.try_into()
        .unwrap();

    if a < b_ {
        println!("Ten is less than one hundred.");
    }
}
```

上面代码中引入了 `std::convert::TryInto` 特征，但是却没有使用它，可能有些同学会为此困惑，主要原因在于**如果你要使用一个特征的方法，那么你需要将该特征引入当前的作用域中**，我们在上面用到了 `try_into` 方法，因此需要引入对应的特征。

但是 Rust 又提供了一个非常便利的办法，即把最常用的标准库中的特征通过 `std::prelude` 模块提前引入到当前作用域中，其中包括了 `std::convert::TryInto`，你可以尝试删除第一行的代码 `use ...`，看看是否会报错。

## 几个综合例子

### 为自定义类型实现 + 操作

在 Rust 中除了数值类型的加法，`String` 也可以做加法，因为 Rust 为该类型实现了 `std::ops::Add` 特征，同理，如果我们为自定义类型实现了该特征，那就可以自己实现 `Point1 + Point2` 的操作：

```

use std::ops::Add;

// 为Point结构体派生Debug特征，用于格式化输出
#[derive(Debug)]
struct Point<T: Add<T, Output = T>> { //限制类型T必须实现了Add特征，否则无法进行+操作。
    x: T,
    y: T,
}

impl<T: Add<T, Output = T>> Add for Point<T> {
    type Output = Point<T>;
    fn add(self, p: Point<T>) -> Point<T> {
        Point{
            x: self.x + p.x,
            y: self.y + p.y,
        }
    }
}

fn add<T: Add<T, Output=T>>(a:T, b:T) -> T {
    a + b
}

fn main() {
    let p1 = Point{x: 1.1f32, y: 1.1f32};
    let p2 = Point{x: 2.1f32, y: 2.1f32};
    println!("{:?}", add(p1, p2));

    let p3 = Point{x: 1i32, y: 1i32};
    let p4 = Point{x: 2i32, y: 2i32};
    println!("{:?}", add(p3, p4));
}

```

## 自定义类型的打印输出

在开发过程中，往往只要使用 `#[derive(Debug)]` 对我们的自定义类型进行标注，即可实现打印输出的功能：

```

#[derive(Debug)]
struct Point{
    x: i32,
    y: i32
}
fn main() {
    let p = Point{x:3,y:3};
    println!("{:?}", p);
}

```

但是在实际项目中，往往需要对我们的自定义类型进行自定义的格式化输出，以让用户更好的阅读理解我们的类型，此时就要为自定义类型实现 `std::fmt::Display` 特征：

```
#![allow(dead_code)]  
  
use std::fmt;  
use std::fmt::{Display};  
  
#[derive(Debug,PartialEq)]  
enum FileState {  
    Open,  
    Closed,  
}  
  
#[derive(Debug)]  
struct File {  
    name: String,  
    data: Vec<u8>,  
    state: FileState,  
}  
  
impl Display for FileState {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        match *self {  
            FileState::Open => write!(f, "OPEN"),  
            FileState::Closed => write!(f, "CLOSED"),  
        }  
    }  
}  
  
impl Display for File {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        write!(f, "{} ({})",  
              self.name, self.state)  
    }  
}  
  
impl File {  
    fn new(name: &str) -> File {  
        File {  
            name: String::from(name),  
            data: Vec::new(),  
            state: FileState::Closed,  
        }  
    }  
}  
  
fn main() {  
    let f6 = File::new("f6.txt");  
    //...  
    println!("{}:?", f6);  
    println!("{}:", f6);  
}
```

以上两个例子较为复杂，目的是为读者展示下真实的使用场景长什么样，因此需要读者细细阅读，最终消化这些知识对于你的 Rust 之路会有莫大的帮助。

最后，特征和特征约束，是 Rust 中极其重要的概念，如果你还是没搞懂，强烈建议回头再看一遍，或者寻找相关的资料进行补充学习。如果已经觉得掌握了，那么就可以进入下一节的学习。

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 特征对象

在上一节中有一段代码无法通过编译：

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        Post {
            // ...
        }
    } else {
        Weibo {
            // ...
        }
    }
}
```

其中 Post 和 Weibo 都实现了 Summary 特征，因此上面的函数试图通过返回 `impl Summary` 来返回这两个类型，但是编译器却无情地报错了，原因是 `impl Trait` 的返回值类型并不支持多种不同的类型返回，那如果我们想返回多种类型，该怎么办？

再来考虑一个问题：现在在做一款游戏，需要将多个对象渲染在屏幕上，这些对象属于不同的类型，存储在列表中，渲染的时候，需要循环该列表并顺序渲染每个对象，在 Rust 中该怎么实现？

聪明的同学可能已经能想到一个办法，利用枚举：

```
#[derive(Debug)]
enum UiObject {
    Button,
    SelectBox,
}

fn main() {
    let objects = [
        UiObject::Button,
        UiObject::SelectBox
    ];

    for o in objects {
        draw(o)
    }
}

fn draw(o: UiObject) {
    println!("{}: {:?}", o);
}
```

Bingo，这个确实是一个办法，但是问题来了，如果你的对象集合并不能事先明确地知道呢？或者别人想要实现一个 UI 组件呢？此时枚举中的类型是有些缺少的，是不是还要修改你的代码增加一个枚举成员？

总之，在编写这个 UI 库时，我们无法知道所有的 UI 对象类型，只知道的是：

- UI 对象的类型不同
- 需要一个统一的类型来处理这些对象，无论是作为函数参数还是作为列表中的一员
- 需要对每一个对象调用 `draw` 方法

在拥有继承的语言中，可以定义一个名为 `Component` 的类，该类上有一个 `draw` 方法。其他的类比如 `Button`、`Image` 和 `SelectBox` 会从 `Component` 派生并因此继承 `draw` 方法。它们各自都可以覆盖 `draw` 方法来定义自己的行为，但是框架会把所有这些类型当作是 `Component` 的实例，并在其上调用 `draw`。不过 Rust 并没有继承，我们得另寻出路。

## 特征对象定义

为了解决上面的所有问题，Rust 引入了一个概念——**特征对象**。

在介绍特征对象之前，先来为之前的 UI 组件定义一个特征：

```
pub trait Draw {  
    fn draw(&self);  
}
```

只要组件实现了 `Draw` 特征，就可以调用 `draw` 方法来进行渲染。假设有一个 `Button` 和 `SelectBox` 组件实现了 `Draw` 特征：

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // 绘制按钮的代码
    }
}

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // 绘制SelectBox的代码
    }
}
```

此时，还需要一个动态数组来存储这些 UI 对象：

```
pub struct Screen {
    pub components: Vec<?>,
}
```

注意到上面代码中的 `?` 吗？它的意思是：我们应该填入什么类型，可以说就之前学过的内容里，你找不到哪个类型可以填入这里，但是因为 `Button` 和 `SelectBox` 都实现了 `Draw` 特征，那我们是不是可以把 `Draw` 特征的对象作为类型，填入到数组中呢？答案是肯定的。

**特征对象**指向实现了 `Draw` 特征的类型的实例，也就是指向了 `Button` 或者 `SelectBox` 的实例，这种映射关系是存储在一张表中，可以在运行时通过特征对象找到具体调用的类型方法。

可以通过 `&` 引用或者 `Box<T>` 智能指针的方式来创建特征对象。

---

`Box<T>` 在后面章节会[详细讲解](#)，大家现在把它当成一个引用即可，只不过它包裹的值会被强制分配在堆上。

---

```

trait Draw {
    fn draw(&self) -> String;
}

impl Draw for u8 {
    fn draw(&self) -> String {
        format!("u8: {}", *self)
    }
}

impl Draw for f64 {
    fn draw(&self) -> String {
        format!("f64: {}", *self)
    }
}

// 若 T 实现了 Draw 特征，则调用该函数时传入的 Box<T> 可以被隐式转换成函数参数签名中的 Box<dyn Draw>
fn draw1(x: Box) {
    // 由于实现了 Deref 特征，Box 智能指针会自动解引用为它所包裹的值，然后调用该值对应的类型上定义的 `draw` 方法
    x.draw();
}

fn draw2(x: &dyn Draw) {
    x.draw();
}

fn main() {
    let x = 1.1f64;
    // do_something(&x);
    let y = 8u8;

    // x 和 y 的类型 T 都实现了 `Draw` 特征，因为 Box<T> 可以在函数调用时隐式地被转换为特征对象 Box<dyn Draw>
    // 基于 x 的值创建一个 Box<f64> 类型的智能指针，指针指向的数据被放置在了堆上
    draw1(Box::new(x));
    // 基于 y 的值创建一个 Box<u8> 类型的智能指针
    draw1(Box::new(y));
    draw2(&x);
    draw2(&y);
}

```

上面代码，有几个非常重要的点：

- `draw1` 函数的参数是 `Box<dyn Draw>` 形式的特征对象，该特征对象是通过 `Box::new(x)` 的方式创建的
- `draw2` 函数的参数是 `&dyn Draw` 形式的特征对象，该特征对象是通过 `&x` 的方式创建的
- `dyn` 关键字只用在特征对象的类型声明上，在创建时无需使用 `dyn`

因此，可以使用特征对象来代表泛型或具体的类型。

继续来完善之前的 UI 组件代码，首先来实现 Screen：

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

其中存储了一个动态数组，里面元素的类型是 Draw 特征对象：Box<dyn Draw>，任何实现了 Draw 特征的类型，都可以存放其中。

再来为 Screen 定义 run 方法，用于将列表中的 UI 组件渲染在屏幕上：

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

至此，我们就完成了之前的目标：在列表中存储多种不同类型的实例，然后将它们使用同一个方法逐一渲染在屏幕上！

再来看看，如果通过泛型实现，会如何：

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

上面的 Screen 的列表中，存储了类型为 T 的元素，然后在 screen 中使用特征约束让 T 实现了 Draw 特征，进而可以调用 draw 方法。

但是这种写法限制了 Screen 实例的 Vec<T> 中的每个元素必须是 Button 类型或者全是 SelectBox 类型。如果只需要同质（相同类型）集合，更倾向于采用泛型+特征约束这种写法，因其实现更清晰，且性能更好（特征对象，需要在运行时从 vtable 动态查找需要调用的方法）。

现在来运行渲染下咱们精心设计的 UI 组件列表：

```
fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}
```

上面使用 `Box::new(T)` 的方式来创建了两个 `Box<dyn Draw>` 特征对象，如果以后还需要增加一个 UI 组件，那么让该组件实现 `Draw` 特征，则可以很轻松的将其渲染在屏幕上，甚至用户可以引入我们的库作为三方库，然后在自己的库中为自己的类型实现 `Draw` 特征，然后进行渲染。

在动态类型语言中，有一个很重要的概念：**鸭子类型**(*duck typing*)，简单来说，就是只关心值长啥样，而不关心它实际是什么。当一个东西走起来像鸭子，叫起来像鸭子，那么它就是一只鸭子，就算它实际上是一个奥特曼，也不重要，我们就当它是鸭子。

在上例中，`Screen` 在 `run` 的时候，我们并不需要知道各个组件的具体类型是什么。它也不检查组件到底是 `Button` 还是 `SelectBox` 的实例，只要它实现了 `Draw` 特征，就能通过 `Box::new` 包装成 `Box<dyn Draw>` 特征对象，然后被渲染在屏幕上。

使用特征对象和 Rust 类型系统来进行类似鸭子类型操作的优势是，无需在运行时检查一个值是否实现了特定方法或者担心在调用时因为值没有实现方法而产生错误。如果值没有实现特征对象所需的特征，那么 Rust 根本就不会编译这些代码：

```

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };

    screen.run();
}

```

因为 `String` 类型没有实现 `Draw` 特征，编译器直接就会报错，不会让上述代码运行。如果想要 `String` 类型被渲染在屏幕上，那么只需要为其实现 `Draw` 特征即可，非常容易。

注意 `dyn` 不能单独作为特征对象的定义，例如下面的代码编译器会报错，原因是特征对象可以是任意实现了某个特征的类型，编译器在编译期不知道该类型的大小，不同的类型大小是不同的。

而 `&dyn` 和 `Box<dyn>` 在编译期都是已知大小，所以可以用作特征对象的定义。

```

fn draw2(x: dyn Draw) {
    x.draw();
}

10 | fn draw2(x: dyn Draw) {
|     ^ doesn't have a size known at compile-time
|
|= help: the trait `Sized` is not implemented for `(dyn Draw + 'static)`
help: function arguments must have a statically known size, borrowed types always
have a known size

```

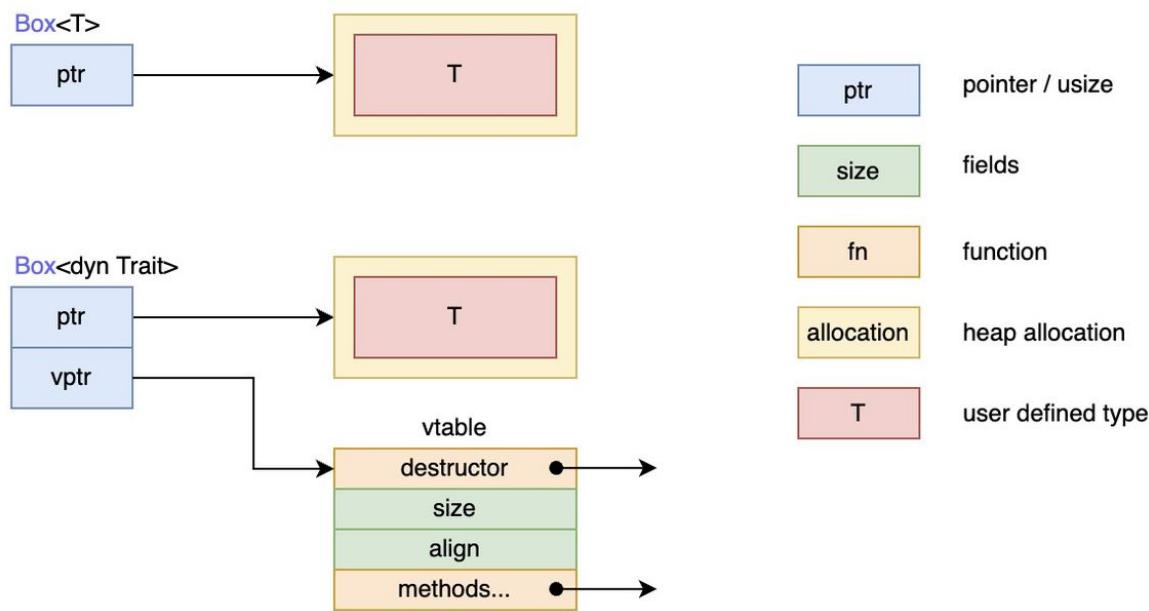
## 特征对象的动态分发

回忆一下泛型章节我们提到过的，泛型是在编译期完成处理的：编译器会为每一个泛型参数对应的具体类型生成一份代码，这种方式是**静态分发(static dispatch)**，因为是在编译期完成的，对于运行期性能完全没有任何影响。

与静态分发相对应的是**动态分发(dynamic dispatch)**，在这种情况下，直到运行时，才能确定需要调用什么方法。之前代码中的关键字 `dyn` 正是在强调这一“动态”的特点。

当使用特征对象时，Rust 必须使用动态分发。编译器无法知晓所有可能用于特征对象代码的类型，所以它也不知道应该调用哪个类型的哪个方法实现。为此，Rust 在运行时使用特征对象中的指针来知晓需要调用哪个方法。动态分发也阻止编译器有选择的内联方法代码，这会相应的禁用一些优化。

下面这张图很好的解释了静态分发 `Box<T>` 和动态分发 `Box<dyn Trait>` 的区别：



<https://doc.rust-lang.org/reference/type-layout.html>

结合上文的内容和这张图可以了解：

- **特征对象大小不固定**：这是因为，对于特征 Draw，类型 Button 可以实现特征 Draw，类型 SelectBox 也可以实现特征 Draw，因此特征没有固定大小
- **几乎总是使用特征对象的引用方式**，如 `&dyn Draw`、`Box<dyn Draw>`
  - 虽然特征对象没有固定大小，但它的引用类型的大小是固定的，它由两个指针组成（ptr 和 vptr），因此占用两个指针大小
  - 一个指针 ptr 指向实现了特征 Draw 的具体类型的实例，也就是当作特征 Draw 来用的类型的实例，比如类型 Button 的实例、类型 SelectBox 的实例
  - 另一个指针 vptr 指向一个虚表 vtable，vtable 中保存了类型 Button 或类型 SelectBox 的实例对于可以调用的实现于特征 Draw 的方法。当调用方法时，直接从 vtable 中找到方法并调用。之所以要使用一个 vtable 来保存各实例的方法，是因为实现了特征 Draw 的类型有多种，这些类型拥有的方法各不相同，当将这些类型的实例都当作特征 Draw 来使用时(此时，它们全都看作是特征 Draw 类型的实例)，有必要区分这些实例各自有哪些方法可调用

简而言之，当类型 Button 实现了特征 Draw 时，类型 Button 的实例对象 btn 可以当作特征 Draw 的特征对象类型来使用，btn 中保存了作为特征对象的数据指针（指向类型 Button 的实例数据）和行为指针（指向 vtable）。

一定要注意，此时的 `btn` 是 `Draw` 的特征对象的实例，而不再是具体类型 `Button` 的实例，而且 `btn` 的 `vtable` 只包含了实现于特征 `Draw` 的那些方法（比如 `draw`），因此 `btn` 只能调用实现于特征 `Draw` 的 `draw` 方法，而不能调用类型 `Button` 本身实现的方法和类型 `Button` 实现于其他特征的方法。**也就是说，`btn` 是哪个特征对象的实例，它的 `vtable` 中就包含了该特征的方法。**

## Self 与 self

在 Rust 中，有两个 `self`，一个指代当前的实例对象，一个指代特征或者方法类型的别名：

```
trait Draw {
    fn draw(&self) -> Self;
}

#[derive(Clone)]
struct Button;
impl Draw for Button {
    fn draw(&self) -> Self {
        return self.clone()
    }
}

fn main() {
    let button = Button;
    let newb = button.draw();
}
```

上述代码中，`self` 指代的就是当前的实例对象，也就是 `button.draw()` 中的 `button` 实例，`Self` 则指代的是 `Button` 类型。

当理解了 `self` 与 `Self` 的区别后，我们再来看看何为对象安全。

## 特征对象的限制

不是所有特征都能拥有特征对象，只有对象安全的特征才行。当一个特征的所有方法都有如下属性时，它的对象才是安全的：

- 方法的返回类型不能是 `Self`
- 方法没有任何泛型参数

对象安全对于特征对象是必须的，因为一旦有了特征对象，就不再需要知道实现该特征的具体类型是什么了。如果特征方法返回了具体的 `Self` 类型，但是特征对象忘记了其真正的类型，那这个 `Self` 就非常

尴尬，因为没人知道它是谁了。但是对于泛型类型参数来说，当使用特征时其会放入具体的类型参数：此具体类型变成了实现该特征的类型的一部分。而当使用特征对象时其具体类型被抹去了，故而无从得知放入泛型参数类型到底是什么。

标准库中的 `Clone` 特征就不符合对象安全的要求：

```
pub trait Clone {  
    fn clone(&self) -> Self;  
}
```

因为它的其中一个方法，返回了 `Self` 类型，因此它是对象不安全的。

`String` 类型实现了 `Clone` 特征，`String` 实例上调用 `clone` 方法时会得到一个 `String` 实例。类似的，当调用 `Vec<T>` 实例的 `clone` 方法会得到一个 `Vec<T>` 实例。`clone` 的签名需要知道什么类型会代替 `Self`，因为这是它的返回值。

如果违反了对象安全的规则，编译器会提示你。例如，如果尝试使用之前的 `Screen` 结构体来存放实现了 `Clone` 特征的类型：

```
pub struct Screen {  
    pub components: Vec<Box<dyn Clone>>,  
}
```

将会得到如下错误：

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object  
--> src/lib.rs:2:5  
|  
2 |     pub components: Vec<Box<dyn Clone>>,  
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone`  
cannot be made into an object  
|= note: the trait cannot require that `Self : Sized`
```

这意味着不能以这种方式使用此特征作为特征对象。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 深入了解特征

特征之于 Rust 更甚于接口之于其他语言，因此特征在 Rust 中很重要也相对较为复杂，我们决定把特征分为两篇进行介绍，[第一篇](#)在之前已经讲过，现在就是第二篇：关于特征的进阶篇，会讲述一些不常用到但是你该了解的特性。

## 关联类型

在方法一章中，我们讲到了[关联函数](#)，但是实际上关联类型和关联函数并没有任何交集，虽然它们的名字有一半的交集。

关联类型是在特征定义的语句块中，申明一个自定义类型，这样就可以在特征的方法签名中使用该类型：

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

以上是标准库中的迭代器特征 `Iterator`，它有一个 `Item` 关联类型，用于替代遍历的值的类型。

同时，`next` 方法也返回了一个 `Item` 类型，不过使用 `Option` 枚举进行了包裹，假如迭代器中的值是 `i32` 类型，那么调用 `next` 方法就将获取一个 `Option<i32>` 的值。

还记得 `Self` 吧？在之前的章节[提到过](#)，`Self` 用来指代当前调用者的具体类型，那么 `Self::Item` 就用来指代该类型实现中定义的 `Item` 类型：

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
    }
}

fn main() {
    let c = Counter{..}
    c.next()
}
```

在上述代码中，我们为 Counter 类型实现了 Iterator 特征，变量 c 是特征 Iterator 的实例，也是 next 方法的调用者。结合之前的黑体内容可以得出：对于 next 方法而言，Self 是调用者 c 的具体类型：Counter，而 Self::Item 是 Counter 中定义的 Item 类型：u32。

聪明的读者之所以聪明，是因为你们喜欢联想和举一反三，同时你们也喜欢提问：为何不用泛型，例如如下代码：

```
pub trait Iterator<Item> {
    fn next(&mut self) -> Option<Item>;
}
```

答案其实很简单，为了代码的可读性，当你使用了泛型后，你需要在所有地方都写 Iterator<Item>，而使用了关联类型，你只需要写 Iterator，当类型定义复杂时，这种写法可以极大的增加可读性：

```
pub trait CacheableItem: Clone + Default + fmt::Debug + Decodable + Encodable {
    type Address: AsRef<[u8]> + Clone + fmt::Debug + Eq + Hash;
    fn is_null(&self) -> bool;
}
```

例如上面的代码，Address 的写法自然远比 AsRef<[u8]> + Clone + fmt::Debug + Eq + Hash 要简单的多，而且含义清晰。

再例如，如果使用泛型，你将得到以下的代码：

```
trait Container<A,B> {
    fn contains(&self,a: A,b: B) -> bool;
}

fn difference<A,B,C>(container: &C) -> i32
where
    C : Container<A,B> {...}
```

可以看到，由于使用了泛型，导致函数头部也必须增加泛型的声明，而使用关联类型，将得到可读性好得多的代码：

```
trait Container{
    type A;
    type B;
    fn contains(&self, a: &Self::A, b: &Self::B) -> bool;
}

fn difference<C: Container>(container: &C) {}
```

## 默认泛型类型参数

当使用泛型类型参数时，可以为其指定一个默认的具体类型，例如标准库中的 `std::ops::Add` 特征：

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

它有一个泛型参数 `RHS`，但是与我们以往的用法不同，这里它给 `RHS` 一个默认值，也就是当用户不指定 `RHS` 时，默认使用两个同样类型的值进行相加，然后返回一个关联类型 `Output`。

可能上面那段不太好理解，下面我们用代码来举例：

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
              Point { x: 3, y: 3 });
}
```

上面的代码主要干了一件事，就是为 `Point` 结构体提供 `+` 的能力，这就是**运算符重载**，不过 Rust 并不支持创建自定义运算符，你也无法为所有运算符进行重载，目前来说，只有定义在 `std::ops` 中的运算符才能进行重载。

跟 `+` 对应的特征是 `std::ops::Add`，我们在之前也看过它的定义 `trait Add<RHS=Self>`，但是上面的例子中并没有为 `Point` 实现 `Add<RHS>` 特征，而是实现了 `Add` 特征（没有默认泛型类型参数），这意味着我们使用了 `RHS` 的默认类型，也就是 `Self`。换句话说，我们这里定义的是两个相同的 `Point` 类型相加，因此无需指定 `RHS`。

与上面的例子相反，下面的例子，我们来创建两个不同类型的相加：

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

这里，是进行 `Millimeters + Meters` 两种数据类型的 `+` 操作，因此此时不能再使用默认的 `RHS`，否则就会变成 `Millimeters + Millimeters` 的形式。使用 `Add<Meters>` 可以将 `RHS` 指定为 `Meters`，那么 `fn add(self, rhs: RHS)` 自然而言的变成了 `Millimeters` 和 `Meters` 的相加。

默认类型参数主要用于两个方面：

1. 减少实现的样板代码
2. 扩展类型但是无需大幅修改现有的代码

之前的例子就是第一点，虽然效果也就那样。在 `+` 左右两边都是同样类型时，只需要 `impl Add` 即可，否则你需要 `impl Add<SOME_TYPE>`，嗯，会多写几个字：)

对于第二点，也很好理解，如果你在一个复杂类型的基础上，新引入一个泛型参数，可能需要修改很多地方，但是如果新引入的泛型参数有了默认类型，情况就会好很多，添加泛型参数后，使用这个类型的代码需要逐个在类型提示部分添加泛型参数，就很麻烦；但是有了默认参数（且默认参数取之前的实现里假设的值的情况下）之后，原有的使用这个类型的代码就不需要做改动了。

归根到底，默认泛型参数，是有用的，但是大多数情况下，咱们确实用不到，当需要用到时，大家再回头来查阅本章即可，**手上有剑，心中不慌**。

## 调用同名的方法

不同特征拥有同名的方法是很正常的事情，你没有任何办法阻止这一点；甚至除了特征上的同名方法外，在你的类型上，也有同名方法：

```

trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}

```

这里，不仅仅两个特征 `Pilot` 和 `Wizard` 有 `fly` 方法，就连实现那两个特征的 `Human` 单元结构体，也拥有一个同名方法 `fly` (这世界怎么了，非要这么卷吗？程序员何苦为程序员，哎)。

既然代码已经不可更改，那下面我们就来讲讲该如何调用这些 `fly` 方法。

## 优先调用类型上的方法

当调用 `Human` 实例的 `fly` 时，编译器默认调用该类型中定义的方法：

```

fn main() {
    let person = Human;
    person.fly();
}

```

这段代码会打印 `*waving arms furiously*`，说明直接调用了类型上定义的方法。

## 调用特征上的方法

为了能够调用两个特征的方法，需要使用显式调用的语法：

```
fn main() {
    let person = Human;
    Pilot::fly(&person); // 调用Pilot特征上的方法
    Wizard::fly(&person); // 调用Wizard特征上的方法
    person.fly(); // 调用Human类型自身的方法
}
```

运行后依次输出：

```
This is your captain speaking.
Up!
*waving arms furiously*
```

因为 `fly` 方法的参数是 `self`，当显式调用时，编译器就可以根据调用的类型(`self` 的类型)决定具体调用哪个方法。

这个时候问题又来了，如果方法没有 `self` 参数呢？稍等，估计有读者会问：还有方法没有 `self` 参数？看到这个疑问，作者的眼泪不禁流了下来，大明湖畔的[关联函数](#)，你还记得嘛？

但是成年人的世界，就算再伤心，事还得做，咱们继续：

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

就像人类妈妈会把自己的宝宝起爱称一样，狗妈妈也会。狗妈妈称呼自己的宝宝为**Spot**，其它动物称呼狗宝宝为**puppy**，这个时候假如有动物不知道该如何称呼狗宝宝，它需要查询一下。

`Dog::baby_name()` 的调用方式显然不行，因为这只是狗妈妈对宝宝的爱称，可能你会想到通过下面的方式查询其他动物对狗狗的称呼：

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

铛铛，无情报错了：

```
error[E0283]: type annotations needed // 需要类型注释
--> src/main.rs:20:43
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                     ^^^^^^^^^^^^^^^^^ cannot infer type //
无法推断类型
|
= note: cannot satisfy `_: Animal`
```

因为单纯从 `Animal::baby_name()` 上，编译器无法得到任何有效的信息：实现 `Animal` 特征的类型可能有很多，你究竟是想获取哪个动物宝宝的名称？狗宝宝？猪宝宝？还是熊宝宝？

此时，就需要使用**完全限定语法**。

### 完全限定语法

完全限定语法是调用函数最为明确的方式：

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

在尖括号中，通过 `as` 关键字，我们向 Rust 编译器提供了类型注解，也就是 `Animal` 就是 `Dog`，而不是其他动物，因此最终会调用 `impl Animal for Dog` 中的方法，获取到其它动物对狗宝宝的称呼：`puppy`。

言归正题，完全限定语法定义为：

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

上面定义中，第一个参数是方法接收器 `receiver`（三种 `self`），只有方法才拥有，例如关联函数就没有 `receiver`。

完全限定语法可以用于任何函数或方法调用，那么我们为何很少用到这个语法？原因是 Rust 编译器能根据上下文自动推导出调用的路径，因此大多数时候，我们都无需使用完全限定语法。只有当存在多个同名

函数或方法，且 Rust 无法区分出你想调用的目标函数时，该用法才能真正有用武之地。

## 特征定义中的特征约束

有时，我们会需要让某个特征 A 能使用另一个特征 B 的功能(另一种形式的特征约束)，这种情况下，不仅仅要为类型实现特征 A，还要为类型实现特征 B 才行，这就是 `supertrait` (实在不知道该如何翻译，有大佬指导下嘛？)

例如有一个特征 `OutlinePrint`，它有一个方法，能够对当前的实现类型进行格式化输出：

```
use std::fmt::Display;

trait OutlinePrint: Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

等等，这里有一个眼熟的语法：`OutlinePrint: Display`，感觉很像之前讲过的**特征约束**，只不过用在了特征定义中而不是函数的参数中，是的，在某种意义上来说，这和特征约束非常类似，都用来说说明一个特征需要实现另一个特征，这里就是：如果你想要实现 `OutlinePrint` 特征，首先你需要实现 `Display` 特征。

想象一下，假如没有这个特征约束，那么 `self.to_string` 还能够调用吗（`to_string` 方法会为实现 `Display` 特征的类型自动实现）？编译器肯定是不愿意的，会报错说当前作用域中找不到用于 `&Self` 类型的方法 `to_string`：

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

impl OutlinePrint for Point {}
```

因为 `Point` 没有实现 `Display` 特征，会得到下面的报错：

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
 --> src/main.rs:20:6
 |
20 |     impl OutlinePrint for Point {}  
|         ^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;  
try using `:?` instead if you are using a format string  
|
= help: the trait `std::fmt::Display` is not implemented for `Point`
```

既然我们有求于编译器，那只能选择满足它咯：

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

上面代码为 `Point` 实现了 `Display` 特征，那么 `to_string` 方法也将自动实现：最终获得字符串是通过这里的 `fmt` 方法获得的。

## 在外部类型上实现外部特征(newtype)

在[特征](#)章节中，有提到孤儿规则，简单来说，就是特征或者类型必需至少有一个是本地的，才能在此类型上定义特征。

这里提供一个办法来绕过孤儿规则，那就是使用**newtype 模式**，简而言之：就是为一个[元组结构体](#)创建新类型。该元组结构体封装有一个字段，该字段就是希望实现特征的具体类型。

该封装类型是本地的，因此我们可以为此类型实现外部的特征。

`newtype` 不仅仅能实现以上的功能，而且它在运行时没有任何性能损耗，因为在编译期，该类型会被自动忽略。

下面来看一个例子，我们有一个动态数组类型：`Vec<T>`，它定义在标准库中，还有一个特征 `Display`，它也定义在标准库中，如果没有 `newtype`，我们是无法为 `Vec<T>` 实现 `Display` 的：

```
error[E0117]: only traits defined in the current crate can be implemented for
arbitrary types
--> src/main.rs:5:1
|
5 | impl<T> std::fmt::Display for Vec<T> {
| ^^^^^^^^^^^^^^^^^^^^^^^^^^-----|
| |
| |                         |
| |                         Vec is not defined in the current crate
| impl doesn't use only types from inside the current crate
|
= note: define and implement a trait or new type instead
```

编译器给了我们提示： `define and implement a trait or new type instead`，重新定义一个特征，或者使用 `new type`，前者当然不可行，那么来试试后者：

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

其中，`struct Wrapper(Vec<String>)` 就是一个元组结构体，它定义了一个新类型 `Wrapper`，代码很简单，相信大家也很容易看懂。

既然 `new type` 有这么多好处，它有没有不好的地方呢？答案是肯定的。注意到我们怎么访问里面的数组吗？`self.0.join(", ")`，是的，很啰嗦，因为需要先从 `Wrapper` 中取出数组：`self.0`，然后才能执行 `join` 方法。

类似的，任何数组上的方法，你都无法直接调用，需要先用 `self.0` 取出数组，然后再进行调用。

当然，解决办法还是有的，要不怎么说 Rust 是极其强大灵活的编程语言！Rust 提供了一个特征叫 `Deref`，实现该特征后，可以自动做一层类似类型转换的操作，可以将 `Wrapper` 变成 `Vec<String>` 来使用。这样就会像直接使用数组那样去使用 `Wrapper`，而无需为每一个操作都添加上 `self.0`。

同时，如果不想 `Wrapper` 暴露底层数组的所有方法，我们还可以为 `Wrapper` 去重载这些方法，实现隐藏的目的。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 集合类型

在 Rust 标准库中有这样一批原住民，它们天生贵族，当你看到的一瞬间，就能爱上它们，上面是我瞎编的，其实主要是离了它们不行，不信等会我介绍后，你放个狠话，偏不用它们试试？

集合在 Rust 中是一类比较特殊的类型，因为 Rust 中大多数数据类型都只能代表一个特定的值，但是集合却可以代表一大堆值。而且与语言级别的数组、字符串类型不同，标准库里的这些家伙是分配在堆上，因此都可以进行动态的增加和减少。

瞧，第一个集合排着整齐的队列登场了，它里面的每个元素都雄赳赳气昂昂跟在另外一个元素后面，大小、宽度、高度竟然全部一致，真是令人惊叹。它就是 `Vector` 类型，允许你创建一个动态数组，它里面的元素是一个紧挨着另一个排列的。

紧接着，第二个集合在全场的嘘声和羡慕眼光中闪亮登场，只见里面的元素排成一对一对的，彼此都手牵着手，非对方莫属，这种情深深雨蒙蒙的样子真是...挺欠扁的。它就是 `HashMap` 类型，该类型允许你在里面存储 `kv` 对，每一个 `k` 都有唯一的 `v` 与之配对。

最后，请用热烈的掌声迎接我们的 `String` 集合，哦，抱歉，`String` 集合天生低调，见不得前两个那样，因此被气走了，你可以去[这里](#)找它。

言归正传，本章所讲的 `Vector`、`HashMap` 再加上之前的 `String` 类型，是标准库中最常用的集合类型，可以说，几乎任何一段代码中都可以找到它们的身影，那么先来看看 `Vector`。

# 动态数组 Vector

动态数组类型用 `Vec<T>` 表示，事实上，在之前的章节，它的身影多次出现，我们一直没有细讲，只是简单的把它当作数组处理。

动态数组允许你存储多个值，这些值在内存中一个紧挨着另一个排列，因此访问其中某个元素的成本非常低。动态数组只能存储相同类型的元素，如果你想存储不同类型的元素，可以使用之前讲过的枚举类型或者特征对象。

总之，当我们想拥有一个列表，里面都是相同类型的数据时，动态数组将会非常有用。

## 创建动态数组

在 Rust 中，有多种方式可以创建动态数组。

### `Vec::new`

使用 `Vec::new` 创建动态数组是最 rusty 的方式，它调用了 `Vec` 中的 `new` 关联函数：

```
let v: Vec<i32> = Vec::new();
```

这里，`v` 被显式地声明了类型 `Vec<i32>`，这是因为 Rust 编译器无法从 `Vec::new()` 中得到任何关于类型的暗示信息，因此也无法推导出 `v` 的具体类型，但是当你向里面增加一个元素后，一切又不同了：

```
let mut v = Vec::new();
v.push(1);
```

此时，`v` 就无需手动声明类型，因为编译器通过 `v.push(1)`，推测出 `v` 中的元素类型是 `i32`，因此推导出 `v` 的类型是 `Vec<i32>`。

---

如果预先知道要存储的元素个数，可以使用 `Vec::with_capacity(capacity)` 创建动态数组，这样可以避免因为插入大量新数据导致频繁的内存分配和拷贝，提升性能

---

## vec![]

还可以使用宏 `vec!` 来创建数组，与 `Vec::new` 有所不同，前者能在创建同时给予初始化值：

```
let v = vec![1, 2, 3];
```

同样，此处的 `v` 也无需标注类型，编译器只需检查它内部的元素即可自动推导出 `v` 的类型是 `Vec<i32>` (Rust 中，整数默认类型是 `i32`，在[数值类型](#)中有详细介绍)。

## 更新 Vector

向数组尾部添加元素，可以使用 `push` 方法：

```
let mut v = Vec::new();
v.push(1);
```

与其它类型一样，必须将 `v` 声明为 `mut` 后，才能进行修改。

## Vector 与其元素共存亡

跟结构体一样，`Vector` 类型在超出作用域范围后，会被自动删除：

```
{
    let v = vec![1, 2, 3];
    // ...
} // <- v超出作用域并在此处被删除
```

当 `Vector` 被删除后，它内部存储的所有内容也会随之被删除。目前来看，这种解决方案简单直白，但是当 `Vector` 中的元素被引用后，事情可能会没那么简单。

## 从 Vector 中读取元素

读取指定位置的元素有两种方式可选：

- 通过下标索引访问。

- 使用 `get` 方法。

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("第三个元素是 {}", third);

match v.get(2) {
    Some(third) => println!("第三个元素是 {}", third),
    None => println!("去你的第三个元素，根本没有！"),
}
```

和其它语言一样，集合类型的索引下标都是从 `0` 开始，`&v[2]` 表示借用 `v` 中的第三个元素，最终会获得该元素的引用。而 `v.get(2)` 也是访问第三个元素，但是有所不同的是，它返回了 `Option<&T>`，因此还需要额外的 `match` 来匹配解构出具体的值。

---

细心的同学会注意到这里使用了两种格式化输出的方式，其中第一种我们在之前已经见过，而第二种是后续新版本中引入的写法，也是更推荐的用法，具体介绍请参见[格式化输出章节](#)。

## 下标索引与 `.get` 的区别

这两种方式都能成功的读取到指定的数组元素，既然如此为什么会有两种方法？何况 `.get` 还会增加使用复杂度，这就涉及到数组越界的问题了，让我们通过示例说明：

```
let v = vec![1, 2, 3, 4, 5];

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

运行以上代码，`&v[100]` 的访问方式会导致程序无情报错退出，因为发生了数组越界访问。但是 `v.get` 就不会，它在内部做了处理，有值的时候返回 `Some(T)`，无值的时候返回 `None`，因此 `v.get` 的使用方式非常安全。

既然如此，为何不统一使用 `v.get` 的形式？因为实在是有些啰嗦，Rust 语言的设计者和使用者在审美这方面还是相当统一的：简洁即正义，何况性能上也会有轻微的损耗。

既然有两个选择，肯定就有如何选择的问题，答案很简单，当你确保索引不会越界的时候，就用索引访问，否则用 `.get`。例如，访问第几个数组元素并不取决于我们，而是取决于用户的输入时，用 `.get` 会非常适合，天知道那些可爱的用户会输入一个什么样的数字进来！

## 同时借用多个数组元素

既然涉及到借用数组元素，那么很可能会遇到同时借用多个数组元素的情况，还记得在[所有权和借用](#)章节咱们讲过的借用规则嘛？如果记得，就来看看下面的代码：）

```
let mut v = vec![1, 2, 3, 4, 5];  
let first = &v[0];  
  
v.push(6);  
  
println!("The first element is: {}", first);
```

先不运行，来推断下结果，首先 `first = &v[0]` 进行了不可变借用，`v.push` 进行了可变借用，如果 `first` 在 `v.push` 之后不再使用，那么该段代码可以成功编译（原因见[引用的作用域](#)）。

可是上面的代码中，`first` 这个不可变借用在可变借用 `v.push` 后被使用了，那么妥妥的，编译器就会报错：

```
$ cargo run  
Compiling collections v0.1.0 (file:///projects/collections)  
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable  
无法对v进行可变借用，因此之前已经进行了不可变借用  
--> src/main.rs:6:5  
|  
4 |     let first = &v[0];  
|             - immutable borrow occurs here // 不可变借用发生在此处  
5 |  
6 |     v.push(6);  
|     ^^^^^^^^^^ mutable borrow occurs here // 可变借用发生在此处  
7 |  
8 |     println!("The first element is: {}", first);  
|                         ----- immutable borrow later used here //  
不可变借用在这里被使用  
  
For more information about this error, try `rustc --explain E0502`.  
error: could not compile `collections` due to previous error
```

其实，按理来说，这两个引用不应该互相影响的：一个是查询元素，一个是在数组尾部插入元素，完全不相干的操作，为何编译器要这么严格呢？

原因在于：数组的大小是可变的，当旧数组的大小不够用时，Rust 会重新分配一块更大的内存空间，然后把旧数组拷贝过来。这种情况下，之前的引用显然会指向一块无效的内存，这非常 rusty —— 对用户进行严格的教育。

其实想想，在长大之后，我们感激人生路上遇到过的严师益友，正是因为他们，我们才在正确的道路上不断前行，虽然在那个时候，并不能理解他们，而 Rust 就如那个良师益友，它不断的在纠正我们不好的编程习惯，直到某一天，你发现自己能写出一次性通过的漂亮代码时，就能明白它的良苦用心。

---

若读者想要更深入的了解 `Vec<T>`，可以看看[Rustonomicon](#)，其中从零手撸一个动态数组，非常适合深入学习。

---

## 迭代遍历 Vector 中的元素

如果想要依次访问数组中的元素，可以使用迭代的方式去遍历数组，这种方式比用下标的方式去遍历数组更安全也更高效（每次下标访问都会触发数组边界检查）：

```
let v = vec![1, 2, 3];
for i in &v {
    println!("{}{}", i);
}
```

也可以在迭代过程中，修改 `Vector` 中的元素：

```
let mut v = vec![1, 2, 3];
for i in &mut v {
    *i += 10
}
```

## 存储不同类型的元素

在本节开头，有讲到数组的元素必须类型相同，但是也提到了解决方案：那就是通过使用枚举类型和特征对象来实现不同类型元素的存储。先来看看通过枚举如何实现：

```
#[derive(Debug)]
enum IpAddr {
    V4(String),
    V6(String)
}
fn main() {
    let v = vec![
        IpAddr::V4("127.0.0.1".to_string()),
        IpAddr::V6(":1".to_string())
    ];

    for ip in v {
        show_addr(ip)
    }
}

fn show_addr(ip: IpAddr) {
    println!("{}: {:?}", ip);
}
```

数组 `v` 中存储了两种不同的 `ip` 地址，但是这两种都属于 `IpAddr` 枚举类型的成员，因此可以存储在数组中。

再来看看特征对象的实现：

```

trait IpAddr {
    fn display(&self);
}

struct V4(String);
impl IpAddr for V4 {
    fn display(&self) {
        println!("ipv4: {:?}", self.0)
    }
}
struct V6(String);
impl IpAddr for V6 {
    fn display(&self) {
        println!("ipv6: {:?}", self.0)
    }
}

fn main() {
    let v: Vec<Box<dyn IpAddr>> = vec![
        Box::new(V4("127.0.0.1".to_string())),
        Box::new(V6(":1".to_string())),
    ];

    for ip in v {
        ip.display();
    }
}

```

比枚举实现要稍微复杂一些，我们为 `v4` 和 `v6` 都实现了特征 `IpAddr`，然后将它俩的实例用 `Box::new` 包裹后，存在了数组 `v` 中，需要注意的是，这里必须手动地指定类型：`Vec<Box<dyn IpAddr>>`，表示数组 `v` 存储的是特征 `IpAddr` 的对象，这样就实现了在数组中存储不同的类型。

在实际使用场景中，**特征对象数组要比枚举数组常见很多**，主要原因在于**特征对象**非常灵活，而编译器对枚举的限制较多，且无法动态增加类型。

## Vector 常用方法

初始化 `vec` 的更多方式：

```

fn main() {
    let v = vec![0; 3];      // 默认值为 0, 初始长度为 3
    let v_from = Vec::from([0, 0, 0]);
    assert_eq!(v, v_from);
}

```

动态数组意味着我们增加元素时，如果容量不足就会导致 **vector 扩容**（目前的策略是重新申请一块 2 倍大小的内存，再将所有元素拷贝到新的内存位置，同时更新指针数据），显然，当频繁扩容或者当元素数量较多且需要扩容时，大量的内存拷贝会降低程序的性能。

可以考虑在初始化时就指定一个实际的预估容量，尽量减少可能的内存拷贝：

```
fn main() {
    let mut v = Vec::with_capacity(10);
    v.extend([1, 2, 3]);      // 附加数据到 v
    println!("Vector 长度是: {}, 容量是: {}", v.len(), v.capacity());

    v.reserve(100);          // 调整 v 的容量，至少要有 100 的容量
    println!("Vector (reserve) 长度是: {}, 容量是: {}", v.len(), v.capacity());

    v.shrink_to_fit();       // 释放剩余的容量，一般情况下，不会主动去释放容量
    println!("Vector (shrink_to_fit) 长度是: {}, 容量是: {}", v.len(), v.capacity());
}
```

Vector 常见的一些方法示例：

```
let mut v = vec![1, 2];
assert!(!v.is_empty());           // 检查 v 是否为空

v.insert(2, 3);                  // 在指定索引插入数据，索引值不能大于 v 的长度，v: [1, 2,
3]
assert_eq!(v.remove(1), 2);       // 移除指定位置的元素并返回，v: [1, 3]
assert_eq!(v.pop(), Some(3));    // 删除并返回 v 尾部的元素，v: [1]
assert_eq!(v.pop(), Some(1));    // v: []
assert_eq!(v.pop(), None);      // 记得 pop 方法返回的是 Option 枚举值
v.clear();                      // 清空 v, v: []

let mut v1 = [11, 22].to_vec(); // append 操作会导致 v1 清空数据，增加可变声明
v.append(&mut v1);            // 将 v1 中的所有元素附加到 v 中，v1: []
v.truncate(1);                // 截断到指定长度，多余的元素被删除，v: [11]
v.retain(|x| *x > 10);        // 保留满足条件的元素，即删除不满足条件的元素

let mut v = vec![11, 22, 33, 44, 55];
// 删除指定范围的元素，同时获取被删除元素的迭代器，v: [11, 55], m: [22, 33, 44]
let mut m: Vec<_> = v.drain(1..=3).collect();

let v2 = m.split_off(1);        // 指定索引处切分成两个 vec, m: [22], v2: [33, 44]
```

当然也可以像[数组切片](#)的方式获取 vec 的部分元素：

```
fn main() {
    let v = vec![11, 22, 33, 44, 55];
    let slice = &v[1..=3];
    assert_eq!(slice, &[22, 33, 44]);
}
```

更多细节，阅读 Vector 的[标准库文档](#)。

## Vector 的排序

在 rust 里，实现了两种排序算法，分别为稳定的排序 `sort` 和 `sort_by`，以及非稳定排序 `sort_unstable` 和 `sort_unstable_by`。

当然，这个所谓的 非稳定 并不是指排序算法本身不稳定，而是指在排序过程中对相等元素的处理方式。在 稳定 排序算法里，对相等的元素，不会对其进行重新排序。而在 不稳定 的算法里则不保证这点。

总体而言，非稳定 排序的算法的速度会优于 稳定 排序算法，同时，稳定 排序还会额外分配原数组一半的空间。

### 整数数组的排序

以下是对整数列进行排序的例子。

```
fn main() {
    let mut vec = vec![1, 5, 10, 2, 15];
    vec.sort_unstable();
    assert_eq!(vec, vec![1, 2, 5, 10, 15]);
}
```

### 浮点数组的排序

我们尝试使用上面的方法来对浮点数进行排序：

```
fn main() {
    let mut vec = vec![1.0, 5.6, 10.3, 2.0, 15f32];
    vec.sort_unstable();
    assert_eq!(vec, vec![1.0, 2.0, 5.6, 10.3, 15f32]);
}
```

结果，居然报错了，

```
error[E0277]: the trait bound `f32: Ord` is not satisfied
--> src/main.rs:29:13
|
29 |     vec.sort_unstable();
|     ^^^^^^^^^^^^^^^^ the trait `Ord` is not implemented for `f32`
|
= help: the following other types implement trait `Ord`:
    i128
    i16
    i32
    i64
    i8
    isize
    u128
    u16
    and 4 others
note: required by a bound in `core::slice::<impl [T]>::sort_unstable`
--> /home/keijack/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/slice/mod.rs:2635:12
|
2635 |     T: Ord,
|     ^^^ required by this bound in `core::slice::<impl [T]>::sort_unstable`

For more information about this error, try `rustc --explain E0277`.
```

原来，在浮点数当中，存在一个 `NAN` 的值，这个值无法与其他的浮点数进行对比，因此，浮点数类型并没有实现全数值可比较 `Ord` 的特性，而是实现了部分可比较的特性 `PartialOrd`。

如此，如果我们确定在我们的浮点数数组当中，不包含 `NAN` 值，那么我们可以使用 `partial_cmp` 来作为大小判断的依据。

```
fn main() {
    let mut vec = vec![1.0, 5.6, 10.3, 2.0, 15f32];
    vec.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());
    assert_eq!(vec, vec![1.0, 2.0, 5.6, 10.3, 15f32]);
}
```

OK，现在可以正确执行了。

## 对结构体数组进行排序

有了上述浮点数排序的经验，我们推而广之，那么对结构体是否也可以使用这种自定义对比函数的方式来进 行呢？马上来试一下：

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u32,
}

impl Person {
    fn new(name: String, age: u32) -> Person {
        Person { name, age }
    }
}

fn main() {
    let mut people = vec![
        Person::new("Zoe".to_string(), 25),
        Person::new("Al".to_string(), 60),
        Person::new("John".to_string(), 1),
    ];
    // 定义一个按照年龄倒序排序的对比函数
    people.sort_unstable_by(|a, b| b.age.cmp(&a.age));

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

执行后输出：

```
[Person { name: "Al", age: 60 }, Person { name: "Zoe", age: 25 }, Person { name: "John", age: 1 }]
```

结果正确。

从上面我们学习过程当中，排序需要我们实现 `Ord` 特性，那么如果我们把我们的结构体实现了该特性，是否就不需要我们自定义对比函数了呢？

是，但不完全是，实现 `Ord` 需要我们实现 `Ord`、`Eq`、`PartialEq`、`PartialOrd` 这些属性。好消息是，你可以 `derive` 这些属性：

```
#[derive(Debug, Ord, Eq, PartialEq, PartialOrd)]
struct Person {
    name: String,
    age: u32,
}

impl Person {
    fn new(name: String, age: u32) -> Person {
        Person { name, age }
    }
}

fn main() {
    let mut people = vec![
        Person::new("Zoe".to_string(), 25),
        Person::new("Al".to_string(), 60),
        Person::new("Al".to_string(), 30),
        Person::new("John".to_string(), 1),
        Person::new("John".to_string(), 25),
    ];

    people.sort_unstable();

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

执行输出

```
[Person { name: "Al", age: 30 }, Person { name: "Al", age: 60 }, Person { name: "John", age: 1 }, Person { name: "John", age: 25 }, Person { name: "Zoe", age: 25 }]
```

需要 `derive Ord` 相关特性，需要确保你的结构体中所有的属性均实现了 `Ord` 相关特性，否则会发生编译错误。`derive` 的默认实现会依据属性的顺序依次进行比较，如上述例子中，当 `Person` 的 `name` 值相同，则会使用 `age` 进行比较。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# KV 存储 HashMap

和动态数组一样，HashMap 也是 Rust 标准库中提供的集合类型，但是又与动态数组不同，HashMap 中存储的是一一映射的 KV 键值对，并提供了平均复杂度为  $O(1)$  的查询方法，当我们希望通过一个 Key 去查询值时，该类型非常有用，以致于 Go 语言将该类型设置成了语言级别的内置特性。

Rust 中哈希类型（哈希映射）为 `HashMap<K,V>`，在其它语言中，也有类似的数据结构，例如 `hash map`，`map`，`object`，`hash table`，`字典` 等等，引用小品演员孙涛的一句台词：大家都是本地狐狸，别搞那装貂 :)。

## 创建 HashMap

跟创建动态数组 `Vec` 的方法类似，可以使用 `new` 方法来创建 `HashMap`，然后通过 `insert` 方法插入键值对。

### 使用 new 方法创建

```
use std::collections::HashMap;

// 创建一个HashMap，用于存储宝石种类和对应的数量
let mut my_gems = HashMap::new();

// 将宝石类型和对应的数量写入表中
my_gems.insert("红宝石", 1);
my_gems.insert("蓝宝石", 2);
my_gems.insert("河边捡的误以为是宝石的破石头", 18);
```

很简单对吧？跟其它语言没有区别，聪明的同学甚至能够猜到该 `HashMap` 的类型：

`HashMap<&str, i32>`。

但是还有一点，你可能没有注意，那就是使用 `HashMap` 需要手动通过 `use ...` 从标准库中引入到我们当前的作用域中来，仔细回忆下，之前使用另外两个集合类型 `String` 和 `Vec` 时，我们是否有手动引用过？答案是 **No**，因为 `HashMap` 并没有包含在 Rust 的 `prelude` 中（Rust 为了简化用户使用，提前将最常用的类型自动引入到作用域中）。

所有的集合类型都是动态的，意味着它们没有固定的内存大小，因此它们底层的数据都存储在内存堆上，然后通过一个存储在栈中的引用类型来访问。同时，跟其它集合类型一致，`HashMap` 也是内聚性的，即所有的 `K` 必须拥有同样的类型，`V` 也是如此。

---

跟 `Vec` 一样，如果预先知道要存储的 `kv` 对个数，可以使用 `HashMap::with_capacity(capacity)` 创建指定大小的 `HashMap`，避免频繁的内存分配和拷贝，提升性能。

---

## 使用迭代器和 `collect` 方法创建

在实际使用中，不是所有的场景都能 `new` 一个哈希表后，然后悠哉悠哉的依次插入对应的键值对，而是可能会从另外一个数据结构中，获取到对应的数据，最终生成 `HashMap`。

例如考虑一个场景，有一张表格中记录了足球联赛中各队伍名称和积分的信息，这张表如果被导入到 Rust 项目中，一个合理的数据结构是 `Vec<(String, u32)>` 类型，该数组中的元素是一个个元组，该数据结构跟表格数据非常契合：表格中的数据都是逐行存储，每一个行都存有一个（队伍名称，积分）的信息。

但是在很多时候，又需要通过队伍名称来查询对应的积分，此时动态数组就不适用了，因此可以用 `HashMap` 来保存相关的队伍名称 -> 积分映射关系。理想很丰满，现实很骨感，如何将 `Vec<(String, u32)>` 中的数据快速写入到 `HashMap<String, u32>` 中？

一个动动脚趾头就能想到的笨方法如下：

```
fn main() {
    use std::collections::HashMap;

    let teams_list = vec![
        ("中国队".to_string(), 100),
        ("美国队".to_string(), 10),
        ("日本队".to_string(), 50),
    ];

    let mut teams_map = HashMap::new();
    for team in &teams_list {
        teams_map.insert(&team.0, team.1);
    }

    println!("{:?}", teams_map)
}
```

遍历列表，将每一个元组作为一对 `kv` 插入到 `HashMap` 中，很简单，但是……也不太聪明的样子，换个词说就是——不够 `rusty`。

好在，Rust 为我们提供了一个非常精妙的解决办法：先将 `Vec` 转为迭代器，接着通过 `collect` 方法，将迭代器中的元素收集后，转成 `HashMap`：

```

fn main() {
    use std::collections::HashMap;

    let teams_list = vec![
        ("中国队".to_string(), 100),
        ("美国队".to_string(), 10),
        ("日本队".to_string(), 50),
    ];

    let teams_map: HashMap<_, _> = teams_list.into_iter().collect();

    println!("{:?}", teams_map)
}

```

代码很简单，`into_iter` 方法将列表转为迭代器，接着通过 `collect` 进行收集，不过需要注意的是，`collect` 方法在内部实际上支持生成多种类型的目标集合，因此我们需要通过类型标注 `HashMap<_, _>` 来告诉编译器：请帮我们收集为 `HashMap` 集合类型，具体的 `kv` 类型，麻烦编译器您老人家帮我们推导。

由此可见，Rust 中的编译器时而小聪明，时而大聪明，不过好在，它大聪明的时候，会自家人知道自己事，总归会通知你一声：

```

error[E0282]: type annotations needed // 需要类型标注
--> src/main.rs:10:9
 |
10 |     let teams_map = teams_list.into_iter().collect();
 |     ^^^^^^^^^^ consider giving `teams_map` a type // 给予 `teams_map` 一个具体的类型

```

## 所有权转移

`HashMap` 的所有权规则与其它 Rust 类型没有区别：

- 若类型实现 `Copy` 特征，该类型会被复制进 `HashMap`，因此无所谓所有权
- 若没实现 `Copy` 特征，所有权将被转移给 `HashMap` 中

例如我参选帅气男孩时的场景再现：

```

fn main() {
    use std::collections::HashMap;

    let name = String::from("Sunface");
    let age = 18;

    let mut handsome_boys = HashMap::new();
    handsome_boys.insert(name, age);

    println!("因为过于无耻，{}已经被从帅气男孩名单中除名", name);
    println!("还有，他的真实年龄远远不止{}岁", age);
}

```

运行代码，报错如下：

```

error[E0382]: borrow of moved value: `name`
--> src/main.rs:10:32
|
4 |     let name = String::from("Sunface");
|         ---- move occurs because `name` has type `String`, which does not
implement the `Copy` trait
...
8 |     handsome_boys.insert(name, age);
|             ---- value moved here
9 |
10 |     println!("因为过于无耻，{}已经被除名", name);
|                     ^^^^^ value borrowed here after move
|

```

提示很清晰，`name` 是 `String` 类型，因此它受到所有权的限制，在 `insert` 时，它的所有权被转移给 `handsome_boys`，所以最后在使用时，会遇到这个无情但是意料之中的报错。

**如果你使用引用类型放入 `HashMap` 中，请确保该引用的生命周期至少跟 `HashMap` 活得一样久：**

```

fn main() {
    use std::collections::HashMap;

    let name = String::from("Sunface");
    let age = 18;

    let mut handsome_boys = HashMap::new();
    handsome_boys.insert(&name, age);

    std::mem::drop(name);
    println!("因为过于无耻，{:?}已经被除名", handsome_boys);
    println!("还有，他的真实年龄远远不止{}岁", age);
}

```

上面代码，我们借用 `name` 获取了它的引用，然后插入到 `handsome_boys` 中，至此一切都很完美。但是紧接着，就通过 `drop` 函数手动将 `name` 字符串从内存中移除，再然后就报错了：

```
handsome_boys.insert(&name, age);
|                                ----- borrow of `name` occurs here // name借用发生在此处
9 |
10 |     std::mem::drop(name);
|           ^^^^ move out of `name` occurs here // name的所有权被转移走
11 |     println!("因为过于无耻, {:?}已经被除名", handsome_boys);
|                                         ----- borrow later used
here // 所有权转移后, 还试图使用name
```

最终，某人因为过于无耻，真正的被除名了 :)

## 查询 HashMap

通过 `get` 方法可以获取元素：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score: Option<&i32> = scores.get(&team_name);
```

上面有几点需要注意：

- `get` 方法返回一个 `Option<&i32>` 类型：当查询不到时，会返回一个 `None`，查询到时返回 `Some(&i32)`
- `&i32` 是对 `HashMap` 中值的借用，如果不使用借用，可能会发生所有权的转移

还可以继续拓展下，上面的代码中，如果我们想直接获得值类型的 `score` 该怎么办，答案简约但不简单：

```
let score: i32 = scores.get(&team_name).copied().unwrap_or(0);
```

这里留给大家一个小作业：去官方文档中查询下 `Option` 的 `copied` 方法和 `unwrap_or` 方法的含义及该如何使用。

还可以通过循环的方式依次遍历 `kv` 对：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

最终输出：

```
Yellow: 50
Blue: 10
```

## 更新 HashMap 中的值

更新值的时候，涉及多种情况，咱们在代码中一一进行说明：

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert("Blue", 10);

    // 覆盖已有的值
    let old = scores.insert("Blue", 20);
    assert_eq!(old, Some(10));

    // 查询新插入的值
    let new = scores.get("Blue");
    assert_eq!(new, Some(&20));

    // 查询Yellow对应的值，若不存在则插入新值
    let v = scores.entry("Yellow").or_insert(5);
    assert_eq!(*v, 5); // 不存在，插入5

    // 查询Yellow对应的值，若不存在则插入新值
    let v = scores.entry("Yellow").or_insert(50);
    assert_eq!(*v, 5); // 已经存在，因此50没有插入
}
```

具体的解释在代码注释中已有，这里不再进行赘述。

## 在已有值的基础上更新

另一个常用场景如下：查询某个 `key` 对应的值，若不存在则插入新值，若存在则对已有的值进行更新，例如在文本中统计词语出现的次数：

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();
// 根据空格来切分字符串(英文单词都是通过空格切分)
for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

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

上面代码中，新建一个 `map` 用于保存词语出现的次数，插入一个词语时会进行判断：若之前没有插入过，则使用该词语作 `Key`，插入次数 0 作为 `Value`，若之前插入过则取出之前统计的该词语出现的次数，对其加一。

有两点值得注意：

- `or_insert` 返回了 `&mut` 引用，因此可以通过该可变引用直接修改 `map` 中对应的值
- 使用 `count` 引用时，需要先进行解引用 `*count`，否则会出现类型不匹配

## 哈希函数

你肯定比较好奇，为何叫哈希表，到底什么是哈希。

先来设想下，如果要实现 `Key` 与 `Value` 的一一对应，是不是意味着我们要能比较两个 `Key` 的相等性？例如 "a" 和 "b"，1 和 2，当这些类型做 `Key` 且能比较时，可以很容易知道 1 对应的值不会错误的映射到 2 上，因为 1 不等于 2。因此，一个类型能否作为 `Key` 的关键就是是否能进行相等比较，或者说该类型是否实现了 `std::cmp::Eq` 特征。

---

f32 和 f64 浮点数，没有实现 `std::cmp::Eq` 特征，因此不可以作为 `HashMap` 的 `Key`。

好了，理解完这个，再来设想一点，若一个复杂点的类型作为 `Key`，那怎么在底层对它进行存储，怎么使用它进行查询和比较？是不是很棘手？好在我们有哈希函数：通过它把 `Key` 计算后映射为哈希值，然后使用该哈希值来进行存储、查询、比较等操作。

但是问题又来了，如何保证不同 Key 通过哈希后的两个值不会相同？如果相同，那意味着我们使用不同的 Key，却查到了同一个结果，这种明显是错误的行为。此时，就涉及到安全性跟性能的取舍了。

若要追求安全，尽可能减少冲突，同时防止拒绝服务（Denial of Service, DoS）攻击，就要使用密码学安全的哈希函数，HashMap 就是使用了这样的哈希函数。反之若要追求性能，就需要使用没有那么安全的算法。

## 高性能三方库

因此若性能测试显示当前标准库默认的哈希函数不能满足你的性能需求，就需要去 [crates.io](#) 上寻找其它的哈希函数实现，使用方法很简单：

```
use std::hash::BuildHasherDefault;
use std::collections::HashMap;
// 引入第三方的哈希函数
use twox_hash::XxHash64;

// 指定HashMap使用第三方的哈希函数XxHash64
let mut hash: HashMap<_, _, BuildHasherDefault<XxHash64>> = Default::default();
hash.insert(42, "the answer");
assert_eq!(hash.get(&42), Some(&"the answer"));
```

---

目前，HashMap 使用的哈希函数是 SipHash，它的性能不是很高，但是安全性很高。SipHash 在中等大小的 Key 上，性能相当不错，但是对于小型的 Key（例如整数）或者大型 Key（例如字符串）来说，性能还是不够好。若你需要极致性能，例如实现算法，可以考虑这个库：[ahash](#)。

---

最后，如果你想要了解 HashMap 更多的用法，请参见本书的标准库解析章节：[HashMap 常用方法](#)

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。

# 认识生命周期

各位读者，之前的集合章节挺简单吧？是不是安逸了挺久了？要不咱们加点料？来试试 Rust 中令人闻风丧胆的生命周期？

生命周期，简而言之就是引用的有效作用域。在大多数时候，我们无需手动的声明生命周期，因为编译器可以自动进行推导，用类型来类比下：

- 就像编译器大部分时候可以自动推导类型 `<->` 一样，编译器大多数时候也可以自动推导生命周期
- 在多种类型存在时，编译器往往要求我们手动标明类型 `<->` 当多个生命周期存在，且编译器无法推导出某个引用的生命周期时，就需要我们手动标明生命周期

Rust 生命周期之所以难，是因为这个概念对于我们来说是全新的，没有其它编程语言的经验可以借鉴。当你觉得难的时候，不用过于担心，这个难对于所有人都是平等的，多点付出就能早点解决此拦路虎，同时本书也会尽力帮助大家减少学习难度(生命周期很可能是 Rust 中最难的部分)。

## 悬垂指针和生命周期

生命周期的主要作用是避免悬垂引用，它会导致程序引用了本不该引用的数据：

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

这段代码有几点值得注意：

- `let r;` 的声明方式貌似存在使用 `null` 的风险，实际上，当我们不初始化它就使用时，编译器会给予报错
- `r` 引用了内部花括号中的 `x` 变量，但是 `x` 会在内部花括号 `}` 处被释放，因此回到外部花括号后，`r` 会引用一个无效的 `x`

此处 `r` 就是一个悬垂指针，它引用了提前被释放的变量 `x`，可以预料到，这段代码会报错：

```
error[E0597]: `x` does not live long enough // `x` 活得不够久
--> src/main.rs:7:17
|
7 |         r = &x;
|             ^^^ borrowed value does not live long enough // 被借用的 `x` 活得不
够久
8 |     }
|     - `x` dropped here while still borrowed // `x` 在这里被丢弃，但是它依然还在被
借用
9 |
10|     println!("r: {}", r);
|             - borrow later used here // 对 `x` 的借用在此处被使用
```

在这里 `r` 拥有更大的作用域，或者说**活得更久**。如果 Rust 不阻止该悬垂引用的发生，那么当 `x` 被释放后，`r` 所引用的值就不再是合法的，会导致我们程序发生异常行为，且该异常行为有时候会很难被发现。

## 借用检查

为了保证 Rust 的所有权和借用的正确性，Rust 使用了一个借用检查器(Borrow checker)，来检查我们程序的借用正确性：

```
{
    let r;                      // -----
                                // |
{
    let x = 5;                // -+-- 'b
    r = &x;                   // |
}                           // -+
                                // |
    println!("r: {}", r); // |
}                           // -----+
```

这段代码和之前的一模一样，唯一的区别在于增加了对变量生命周期的注释。这里，`r` 变量被赋予了生命周期 '`a`'，`x` 被赋予了生命周期 '`b`'，从图示上可以明显看出生命周期 '`b`' 比 '`a`' 小很多。

在编译期，Rust 会比较两个变量的生命周期，结果发现 `r` 明明拥有生命周期 '`a`'，但是却引用了一个小得多的生命周期 '`b`'，在这种情况下，编译器会认为我们的程序存在风险，因此拒绝运行。

如果想要编译通过，也很简单，只要 '`b`' 比 '`a`' 大就好。总之，`x` 变量只要比 `r` 活得久，那么 `r` 就能随意引用 `x` 且不会存在危险：

```

{
    let x = 5;           // -----+-- 'b
    //
    let r = &x;          // --+- 'a |
    //
    println!("r: {}", r); // | |
    //
}
// -----

```

根据之前的结论，我们重新实现了代码，现在 `x` 的生命周期 '`b`' 大于 `r` 的生命周期 '`a`'，因此 `r` 对 `x` 的引用是安全的。

通过之前的内容，我们了解了何为生命周期，也了解了 Rust 如何利用生命周期来确保引用是合法的，下面来看看函数中的生命周期。

## 函数中的生命周期

先来考虑一个例子 - 返回两个字符串切片中较长的那个，该函数的参数是两个字符串切片，返回值也是字符串切片：

```

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

```

这段 `longest` 实现，非常标准优美，就连多余的 `return` 和分号都没有，可是现实总是给我们重重一击：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:9:33
|
9 | fn longest(x: &str, y: &str) -> &str {
|           ----      ^ expected named lifetime parameter // 参数需要一个
生命周期
|
| = help: this function's return type contains a borrowed value, but the signature
does not say whether it is
borrowed from `x` or `y`
= 帮助：该函数的返回值是一个引用类型，但是函数签名无法说明，该引用是借用自 `x` 还是 `y`
help: consider introducing a named lifetime parameter // 考虑引入一个生命周期
|
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
|     ^^^^^^    ^^^^^^^^    ^^^^^^^^    ^^^
```

喔，这真是一个复杂的提示，那感觉就好像是生命周期去非诚勿扰相亲，结果在初印象环节就 23 盏灯全灭。等等，先别急，如果你仔细阅读，就会发现，其实主要是编译器无法知道该函数的返回值到底引用 `x` 还是 `y`，**因为编译器需要知道这些，来确保函数调用后的引用生命周期分析。**

不过说来尴尬，就这个函数而言，我们也不知道返回值到底引用哪个，因为一个分支返回 `x`，另一个分支返回 `y` ...这可咋办？先来分析下。

我们在定义该函数时，首先无法知道传递给函数的具体值，因此到底是 `if` 还是 `else` 被执行，无从得知。其次，传入引用的具体生命周期也无法知道，因此也不能像之前的例子那样通过分析生命周期来确定引用是否有效。同时，编译器的借用检查也无法推导出返回值的生命周期，因为它不知道 `x` 和 `y` 的生命周期跟返回值的生命周期之间的关系是怎样的(说实话，人都搞不清，何况编译器这个大聪明)。

因此，这时就回到了文章开头说的内容：在存在多个引用时，编译器有时会无法自动推导生命周期，此时就需要我们手动去标注，通过为参数标注合适的生命周期来帮助编译器进行借用检查的分析。

## 生命周期标注语法

---

生命周期标注并不会改变任何引用的实际作用域 -- 鲁迅

---

鲁迅说过的话，总是值得重点标注，当你未来更加理解生命周期时，你才会发现这句话的精髓和重要！现在先简单记住，**标记的生命周期只是为了取悦编译器，让编译器不要难为我们**，记住了吗？没记住，再回头看一遍，这对未来你遇到生命周期问题时会有很大的帮助！

在很多时候编译器是很聪明的，但是总有些时候，它会化身大聪明，自以为什么都很懂，然后去拒绝我们代码的执行，此时，就需要我们通过生命周期标注来告诉这个大聪明：别自作聪明了，听我的就好。

例如一个变量，只能活一个花括号，那么就算你给它标注一个活全局的生命周期，它还是会在前面的花括号结束处被释放掉，并不会真的全局存活。

生命周期的语法也颇为与众不同，以 '`'` 开头，名称往往是一个单独的小写字母，大多数人都用 '`'a`' 来作为生命周期的名称。如果是引用类型的参数，那么生命周期会位于引用符号 `&` 之后，并用一个空格来将生命周期和引用参数分隔开：

```
&i32          // 一个引用
&'a i32      // 具有显式生命周期的引用
&'a mut i32 // 具有显式生命周期的可变引用
```

一个生命周期标注，它自身并不具有什么意义，因为生命周期的作用就是告诉编译器多个引用之间的关系。例如，有一个函数，它的第一个参数 `first` 是一个指向 `i32` 类型的引用，具有生命周期 `'a`，该函数还有另一个参数 `second`，它也是指向 `i32` 类型的引用，并且同样具有生命周期 `'a`。此处生命周期标注仅仅说明，**这两个参数 `first` 和 `second` 至少活得和 '`'a` 一样久，至于到底活多久或者哪个活得更久，抱歉我们都无法得知：**

```
fn useless<'a>(first: &'a i32, second: &'a i32) {}
```

## 函数签名中的生命周期标注

继续之前的 `longest` 函数，从两个字符串切片中返回较长的那个：

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

需要注意的点如下：

- 和泛型一样，使用生命周期参数，需要先声明 `<'a>`
- `x`、`y` 和返回值至少活得和 `'a` 一样久(因为返回值要么是 `x`，要么是 `y`)

该函数签名表明对于某些生命周期 `'a`，函数的两个参数都至少跟 `'a` 活得一样久，同时函数的返回引用也至少跟 `'a` 活得一样久。实际上，这意味着返回值的生命周期与参数生命周期中的较小值一致：虽然两个参数的生命周期都是标注了 `'a`，但是实际上这两个参数的真实生命周期可能是不一样的(生命周期 `'a` 不代表生命周期等于 `'a`，而是大于等于 `'a`)。

回忆下“鲁迅”说的话，再参考上面的内容，可以得出：**在通过函数签名指定生命周期参数时，我们并没有改变传入引用或者返回引用的真实生命周期，而是告诉编译器当不满足此约束条件时，就拒绝编译通过。**

因此 `longest` 函数并不知道 `x` 和 `y` 具体会活多久，只要知道它们的作用域至少能持续 '`a`' 这么长就行。

当把具体的引用传给 `longest` 时，那生命周期 '`a`' 的大小就是 `x` 和 `y` 的作用域的重合部分，换句话说，'`a`' 的大小将等于 `x` 和 `y` 中较小的那个。由于返回值的生命周期也被标记为 '`a`'，因此返回值的生命周期也是 `x` 和 `y` 中作用域较小的那个。

说实话，这段文字我写的都快崩溃了，不知道你们读起来如何，实在\*\*\*太绕了。。那就干脆用一个例子来解释吧：

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

在上例中，`string1` 的作用域直到 `main` 函数的结束，而 `string2` 的作用域到内部花括号的结束 }，那么根据之前的理论，'`a`' 是两者中作用域较小的那个，也就是 '`a`' 的生命周期等于 `string2` 的生命周期，同理，由于函数返回的生命周期也是 '`a`'，可以得出函数返回的生命周期也等于 `string2` 的生命周期。

现在来验证下上面的结论：`result` 的生命周期等于参数中生命周期最小的，因此要等于 `string2` 的生命周期，也就是说，`result` 要活得和 `string2` 一样久，观察下代码的实现，可以发现这个结论是正确的！

因此，在这种情况下，通过生命周期标注，编译器得出了和我们肉眼观察一样的结论，而不再是一个蒙圈的大聪明。

再来看一个例子，该例子证明了 `result` 的生命周期必须等于两个参数中生命周期较小的那个：

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Bang，错误冒头了：

```
error[E0597]: `string2` does not live long enough
--> src/main.rs:6:44
|
6 |         result = longest(string1.as_str(), string2.as_str());
|                         ^^^^^^^ borrowed value does not live
long enough
7 |     }
|     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
|                     ----- borrow later used here
```

在上述代码中，`result` 必须要活到 `println!` 处，因为 `result` 的生命周期是 '`a`'，因此 '`a`' 必须持续到 `println!`。

在 `longest` 函数中，`string2` 的生命周期也是 '`a`'，由此说明 `string2` 也必须活到 `println!` 处，可是 `string2` 在代码中实际上只能活到内部语句块的花括号处 }，小于它应该具备的生命周期 '`a`'，因此编译出错。

作为人类，我们可以很清晰的看出 `result` 实际上引用了 `string1`，因为 `string1` 的长度明显要比 `string2` 长，既然如此，编译器不该如此矫情才对，它应该能认识到 `result` 没有引用 `string2`，让我们这段代码通过。只能说，作为尊贵的人类，编译器的发明者，你高估了这个工具的能力，它真的做不到！而且 Rust 编译器在调教上是非常保守的：当可能出错也可能不出错时，它会选择前者，抛出编译错误。

总之，显式的使用生命周期，可以让编译器正确的认识到多个引用之间的关系，最终帮我们提前规避可能存在的代码风险。

小练习：尝试着去更改 `longest` 函数，例如修改参数、生命周期或者返回值，然后推测结果如何，最后再跟编译器的输出进行印证。

## 深入思考生命周期标注

使用生命周期的方式往往取决于函数的功能，例如之前的 `longest` 函数，如果它永远只返回第一个参数 `x`，生命周期的标注该如何修改(该例子就是上面的小练习结果之一)？

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

在此例中，`y` 完全没有被使用，因此 `y` 的生命周期与 `x` 和返回值的生命周期没有任何关系，意味着我们也不必再为 `y` 标注生命周期，只需要标注 `x` 参数和返回值即可。

**函数的返回值如果是一个引用类型，那么它的生命周期只会来源于：**

- 函数参数的生命周期
- 函数体中某个新建引用的生命周期

若是后者情况，就是典型的悬垂引用场景：

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

上面的函数的返回值就和参数 `x`, `y` 没有任何关系，而是引用了函数体内创建的字符串，那么很显然，该函数会报错：

```
error[E0515]: cannot return value referencing local variable `result` // 返回值result
引用了本地的变量
--> src/main.rs:11:5
11 |     result.as_str()
   |     -----^^^^^^^^^
   |
   |     |
   |     returns a value referencing data owned by the current function
   |     `result` is borrowed here
```

主要问题就在于，`result` 在函数结束后就被释放，但是在函数结束后，对 `result` 的引用依然在继续。在这种情况下，没有办法指定合适的生命周期来让编译通过，因此我们也就在 Rust 中避免了悬垂引用。

那遇到这种情况该怎么办？最好的办法就是返回内部字符串的所有权，然后把字符串的所有权转移给调用者：

```
fn longest<'a>(_x: &str, _y: &str) -> String {
    String::from("really long string")
}

fn main() {
    let s = longest("not", "important");
}
```

至此，可以对生命周期进行下总结：生命周期语法用来将函数的多个引用参数和返回值的作用域关联到一起，一旦关联到一起后，Rust 就拥有充分的信息来确保我们的操作是内存安全的。

## 结构体中的生命周期

不仅仅函数具有生命周期，结构体其实也有这个概念，只不过我们之前对结构体的使用都停留在非引用类型字段上。细心的同学应该能回想起来，之前为什么不在结构体中使用字符串字面量或者字符串切片，而是统一使用 `String` 类型？原因很简单，后者在结构体初始化时，只要转移所有权即可，而前者，抱歉，它们是引用，它们不能为所欲为。

既然之前已经理解了生命周期，那么意味着在结构体中使用引用也变得可能：只要为结构体中的每一个引用标注上生命周期即可：

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

`ImportantExcerpt` 结构体中有一个引用类型的字段 `part`，因此需要为它标注上生命周期。结构体的生命周期标注语法跟泛型参数语法很像，需要对生命周期参数进行声明 `<'a>`。该生命周期标注说明，**结构体 `ImportantExcerpt` 所引用的字符串 `str` 必须比该结构体活得更久**。

从 `main` 函数实现来看，`ImportantExcerpt` 的生命周期从第 4 行开始，到 `main` 函数末尾结束，而该结构体引用的字符串从第一行开始，也是到 `main` 函数末尾结束，可以得出结论**结构体引用的字符串活得比结构体久**，这符合了编译器对生命周期的要求，因此编译通过。

与之相反，下面的代码就无法通过编译：

```

#[derive(Debug)]
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let i;
    {
        let novel = String::from("Call me Ishmael. Some years ago...");
        let first_sentence = novel.split('.').next().expect("Could not find a '.'");
        i = ImportantExcerpt {
            part: first_sentence,
        };
    }
    println!("{}:?}", i);
}

```

观察代码，**可以看出结构体比它引用的字符串活得更久**，引用字符串在内部语句块末尾 } 被释放后，`println!` 依然在外面使用了该结构体，因此会导致无效的引用，不出所料，编译报错：

```

error[E0597]: `novel` does not live long enough
--> src/main.rs:10:30
|
10 |         let first_sentence = novel.split('.').next().expect("Could not find a
'.'");
|                         ^^^^^^^^^^^^^^^^^ borrowed value does not live long
enough
...
14 |     }
|     - `novel` dropped here while still borrowed
15 |     println!("{}:?", i);
|             - borrow later used here

```

## 生命周期消除

实际上，对于编译器来说，每一个引用类型都有一个生命周期，那么为什么我们在使用过程中，很多时候无需标注生命周期？例如：

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

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

    &s[..]
}
```

该函数的参数和返回值都是引用类型，尽管我们没有显式的为其标注生命周期，编译依然可以通过。其实原因不复杂，**编译器为了简化用户的使用，运用了生命周期消除大法**。

对于 `first_word` 函数，它的返回值是一个引用类型，那么该引用只有两种情况：

- 从参数获取
- 从函数体内部新创建的变量获取

如果是后者，就会出现悬垂引用，最终被编译器拒绝，因此只剩一种情况：返回值的引用是获取自参数，这就意味着参数和返回值的生命周期是一样的。道理很简单，我们能看出来，编译器自然也能看出来，因此，就算我们不标注生命周期，也不会产生歧义。

实际上，在 Rust 1.0 版本之前，这种代码果断不给通过，因为 Rust 要求必须显式的为所有引用标注生命周期：

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

在写了大量的类似代码后，Rust 社区抱怨声四起，包括开发者自己都忍不了了，最终揭锅而起，这才有了我们今日的幸福。

生命周期消除的规则不是一蹴而就，而是伴随着 `总结-改善` 流程的周而复始，一步一步走到今天，这也意味着，该规则以后可能也会进一步增加，我们需要手动标注生命周期的时候也会越来越少，hooray!

在开始之前有几点需要注意：

- 消除规则不是万能的，若编译器不能确定某件事是正确时，会直接判为不正确，那么你还是需要手动标注生命周期
- **函数或者方法中，参数的生命周期被称为 输入生命周期，返回值的生命周期被称为 输出生命周期**

### 三条消除规则

编译器使用三条消除规则来确定哪些场景不需要显式地去标注生命周期。其中第一条规则应用在输入生命周期上，第二、三条应用在输出生命周期上。若编译器发现三条规则都不适用时，就会报错，提示你需要

手动标注生命周期。

### 1. 每一个引用参数都会获得独自的生命周期

例如一个引用参数的函数就有一个生命周期标注: `fn foo<'a>(x: &'a i32)`, 两个引用参数的有两个生命周期标注: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`, 依此类推。

### 2. 若只有一个输入生命周期(函数参数中只有一个引用类型), 那么该生命周期会被赋给所有的输出生命周期, 也就是所有返回值的生命周期都等于该输入生命周期

例如函数 `fn foo(x: &i32) -> &i32`, `x` 参数的生命周期会被自动赋给返回值 `&i32`, 因此该函数等同于 `fn foo<'a>(x: &'a i32) -> &'a i32`

### 3. 若存在多个输入生命周期, 且其中一个是 `&self` 或 `&mut self`, 则 `&self` 的生命周期被赋给所有的输出生命周期

拥有 `&self` 形式的参数, 说明该函数是一个 方法, 该规则让方法的使用便利度大幅提升。

规则其实很好理解, 但是, 爱思考的读者肯定要发问了, 例如第三条规则, 若一个方法, 它的返回值的生命周期就是跟参数 `&self` 的不一样怎么办? 总不能强迫我返回的值总是和 `&self` 活得一样久吧? ! 问得好, 答案很简单: 手动标注生命周期, 因为这些规则只是编译器发现你没有标注生命周期时默认去使用的, 当你标注生命周期后, 编译器自然会乖乖听你的话。

让我们假装自己是编译器, 然后看下以下的函数该如何应用这些规则:

#### 例子 1

```
fn first_word(s: &str) -> &str { // 实际项目中的手写代码}
```

首先, 我们手写的代码如上所示时, 编译器会先应用第一条规则, 为每个参数标注一个生命周期:

```
fn first_word<'a>(s: &'a str) -> &str { // 编译器自动为参数添加生命周期}
```

此时, 第二条规则就可以进行应用, 因为函数只有一个输入生命周期, 因此该生命周期会被赋予所有的输出生命周期:

```
fn first_word<'a>(s: &'a str) -> &'a str { // 编译器自动为返回值添加生命周期}
```

此时, 编译器为函数签名中的所有引用都自动添加了具体的生命周期, 因此编译通过, 且用户无需手动去标注生命周期, 只要按照 `fn first_word(s: &str) -> &str { }` 的形式写代码即可。

#### 例子 2 再来看一个例子:

```
fn longest(x: &str, y: &str) -> &str { // 实际项目中的手写代码
```

首先，编译器会应用第一条规则，为每个参数都标注生命周期：

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

但是此时，第二条规则却无法被使用，因为输入生命周期有两个，第三条规则也不符合，因为它是函数，不是方法，因此没有 `&self` 参数。在套用所有规则后，编译器依然无法为返回值标注合适的生命周期，因此，编译器就会报错，提示我们需要手动标注生命周期：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:1:47
|
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
|           -----      -----      ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `x` or `y`
note: these named lifetimes are available to use
--> src/main.rs:1:12
|
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
|           ^^  ^^
help: consider using one of the available lifetimes here
|
1 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'lifetime str {
|                           +++++++
```

不得不说，Rust 编译器真的很强大，还贴心的给我们提示了该如何修改，虽然。。。好像。。。它的提示貌似不太准确。这里我们更希望参数和返回值都是 '`a` 生命周期。

## 方法中的生命周期

先来回忆下泛型的语法：

```

struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

```

实际上，为具有生命周期的结构体实现方法时，我们使用的语法跟泛型参数语法很相似：

```

struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}

```

其中有几点需要注意的：

- `impl` 中必须使用结构体的完整名称，包括 `<'a>`，因为生命周期标注也是结构体类型的一部分！
- 方法签名中，往往不需要标注生命周期，得益于生命周期消除的第一和第三规则

下面的例子展示了第三规则应用的场景：

```

impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}

```

首先，编译器应用第一规则，给予每个输入参数一个生命周期：

```

impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&'a self, announcement: &'b str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}

```

需要注意的是，编译器不知道 `announcement` 的生命周期到底多长，因此它无法简单的给予它生命周期 '`a`'，而是重新声明了一个全新的生命周期 '`b`'。

接着，编译器应用第三规则，将 `&self` 的生命周期赋给返回值 `&str`：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&'a self, announcement: &'b str) -> &'a str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

Bingo，最开始的代码，尽管我们没有给方法标注生命周期，但是在第一和第三规则的配合下，编译器依然完美的为我们亮起了绿灯。

在结束这块儿内容之前，再来做一个有趣的修改，将方法返回的生命周期改为 '`b`'：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&'a self, announcement: &'b str) -> &'b str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

此时，编译器会报错，因为编译器无法知道 '`a`' 和 '`b`' 的关系。`&self` 生命周期是 '`a`'，那么 `self.part` 的生命周期也是 '`a`'，但是好巧不巧的是，我们手动为返回值 `self.part` 标注了生命周期 '`b`'，因此编译器需要知道 '`a`' 和 '`b`' 的关系。

有一点很容易推理出来：由于 `&'a self` 是被引用的一方，因此引用它的 `&'b str` 必须要活得比它短，否则会出现悬垂引用。因此说明生命周期 '`b`' 必须要比 '`a`' 小，只要满足了这一点，编译器就不会再报错：

```
impl<'a: 'b, 'b> ImportantExcerpt<'a> {
    fn announce_and_return_part(&'a self, announcement: &'b str) -> &'b str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

Bang，一个复杂的玩意儿被甩到了你面前，就问怕不怕？

就关键点稍微解释下：

- '`'a: 'b`'，是生命周期约束语法，跟泛型约束非常相似，用于说明 '`a`' 必须比 '`b`' 活得久

- 可以把 '`a`' 和 '`b`' 都在同一个地方声明（如上），或者分开声明但通过 `where 'a: 'b'` 约束生命周期关系，如下：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part<'b>(&'a self, announcement: &'b str) -> &'b str
    where
        'a: 'b,
    {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

总之，实现方法比想象中简单：加一个约束，就能暗示编译器，尽管引用吧，反正我想引用的内容比我活得久，爱咋咋地，我怎么都不会引用到无效的内容！

## 静态生命周期

在 Rust 中有一个非常特殊的生命周期，那就是 `'static`，拥有该生命周期的引用可以和整个程序活得一样久。

在之前我们学过字符串字面量，提到过它是被硬编码进 Rust 的二进制文件中，因此这些字符串变量全部具有 `'static` 的生命周期：

```
let s: &'static str = "我没啥优点，就是活得久，嘿嘿";
```

这时候，有些聪明的小脑瓜就开始开动了：当生命周期不知道怎么标时，对类型施加一个静态生命周期的约束 `T: 'static` 是不是很爽？这样我和编译器再也不用操心它到底活多久了。

嗯，只能说，这个想法是对的，在不少情况下，`'static` 约束确实可以解决生命周期编译不通过的问题，但是问题来了：本来该引用没有活那么久，但是你非要说它活那么久，万一引入了潜在的 BUG 怎么办？

因此，遇到因为生命周期导致的编译不通过问题，首先想的应该是：是否是我们试图创建一个悬垂引用，或者是试图匹配不一致的生命周期，而不是简单粗暴的用 `'static` 来解决问题。

但是，话说回来，存在即合理，有时候，`'static` 确实可以帮助我们解决非常复杂的生命周期问题甚至是无法被手动解决的生命周期问题，那么此时就应该放心大胆的用，只要你确定：**你的所有引用的生命周期都是正确的，只是编译器太笨不懂罢了。**

总结下：

- 生命周期 'static 意味着能和程序活得一样久，例如字符串字面量和特征对象
- 实在遇到解决不了的生命周期标注问题，可以尝试 T: 'static，有时候它会给你奇迹

---

事实上，关于 'static，有两种用法: &'static 和 T: 'static，详细内容请参见[此处](#)。

---

## 一个复杂例子：泛型、特征约束

手指已经疲软无力，我好想停止，但是华丽的开场都要有与之匹配的谢幕，那我们就用一个稍微复杂点的例子来结束：

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

依然是熟悉的配方 longest，但是多了一段废话：ann，因为要用格式化 {} 来输出 ann，因此需要它实现 Display 特征。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。（本节暂无习题解答）

---

## 总结

我不知道支撑我一口气写完的勇气是什么，也许是不做完不爽夫斯基，也许是一些读者对本书的期待，不管如何，这章足足写了 17000 字，可惜不是写小说，不然肯定可以获取很多月票 :)

从本章开始，最大的收获就是在结构体中使用引用类型了，说实话，为了引入这个特性，我已经憋了足足 N 个章节.....

但是，还没完，是的，就算是将近两万字，生命周期的旅程依然没有完结，在本书的进阶部分，我们将介绍一些关于生命周期的高级特性，这些特性你在其它中文书中目前还看不到的。

# 返回值和错误处理

飞鸽传书、八百里加急，自古以来，掌权者最需要的就是及时获得对某个事物的信息反馈，在此过程中，也定义了相应的应急处理措施。

社会演变至今，这种思想依然没变，甚至来到计算中的微观世界，也是如此。及时、准确的获知系统在发生什么，是程序设计的重中之重。因此能够准确的分辨函数返回值是正确的还是错误的、以及在发生错误时该怎么快速处理，成了程序设计语言的必备功能。

Go 语言为人诟病的其中一点就是 `if err != nil {}` 的大量使用，缺乏一些程序设计的美感，不过我倒是觉得这种简单的方式也有其好处，就是阅读代码时的流畅感很强，你不需要过多的思考各种语法是什么意思。与 Go 语言不同，Rust 博采众家之长，实现了颇具自身色彩的返回值和错误处理体系，本章我们就高屋建瓴地来学习，更加深入的讲解见[错误处理](#)。

## Rust 的错误哲学

错误对于软件来说是不可避免的，因此一门优秀的编程语言必须有其完整的错误处理哲学。在很多情况下，Rust 需要你承认自己的代码可能会出错，并提前采取行动，来处理这些错误。

Rust 中的错误主要分为两类：

- **可恢复错误**，通常用于从系统全局角度来看可以接受的错误，例如处理用户的访问、操作等错误，这些错误只会影响某个用户自身的操作进程，而不会对系统的全局稳定性产生影响
- **不可恢复错误**，刚好相反，该错误通常是全局性或者系统性的错误，例如数组越界访问，系统启动时发生了影响启动流程的错误等等，这些错误的影响往往对于系统来说是致命的

很多编程语言，并不会区分这些错误，而是直接采用异常的方式去处理。Rust 没有异常，但是 Rust 也有自己的卧龙凤雏：`Result<T, E>` 用于可恢复错误，`panic!` 用于不可恢复错误。

# panic 深入剖析

在正式开始之前，先来思考一个问题：假设我们想要从文件读取数据，如果失败，你有没有好的办法通知调用者为何失败？如果成功，你有没有好的办法把读取的结果返还给调用者？

## panic! 与不可恢复错误

上面的问题在真实场景会经常遇到，其实处理起来挺复杂的，让我们先做一个假设：文件读取操作发生在系统启动阶段。那么可以轻易得出一个结论，一旦文件读取失败，那么系统启动也将失败，这意味着该失败是不可恢复的错误，无论是因为文件不存在还是操作系统硬盘的问题，这些只是错误的原因不同，但是归根到底都是不可恢复的错误(梳理清楚当前场景的错误类型非常重要)。

对于这些严重到影响程序运行的错误，触发 `panic` 是很好的解决方式。在 Rust 中触发 `panic` 有两种方式：被动触发和主动调用，下面依次来看看。

### 被动触发

先来看一段简单又熟悉的代码：

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

心明眼亮的同学立马就能看出这里发生了严重的错误——数组访问越界，在其它编程语言中无一例外，都会报出严重的异常，甚至导致程序直接崩溃关闭。

而 Rust 也不例外，运行后将看到如下报错：

```
$ cargo run
Compiling panic v0.1.0 (file:///projects/panic)
Finished dev [unoptimized + debuginfo] target(s) in 0.27s
Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

上面给出了非常详细的报错信息，包含了具体的异常描述以及发生的位置，甚至你还可以加入额外的命令来看到异常发生时的堆栈信息，这个会在后面详细展开。

总之，类似的 `panic` 还有很多，而被动触发的 `panic` 是我们日常开发中最常遇到的，这也是 Rust 给我们的一种保护，毕竟错误只有抛出来，才有可能被处理，否则只会偷偷隐藏起来，寻觅时机给你致命一击。

## 主动调用

在某些特殊场景中，开发者想要主动抛出一个异常，例如开头提到的在系统启动阶段读取文件失败。

对此，Rust 为我们提供了 `panic!` 宏，当调用执行该宏时，**程序会打印出一个错误信息，展开报错点往前的函数调用堆栈，最后退出程序。**

---

切记，一定是不可恢复的错误，才调用 `panic!` 处理，你总不想系统仅仅因为用户随便传入一个非法参数就崩溃吧？所以，**只有当你不知道该如何处理时，再去调用 `panic!`。**

首先，来调用一下 `panic!`，这里使用了最简单的代码实现，实际上你在程序的任何地方都可以这样调用：

```
fn main() {
    panic!("crash and burn");
}
```

运行后输出：

```
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

以上信息包含了两条重要信息：

- `main` 函数所在的线程崩溃了，发生的代码位置是 `src/main.rs` 中的第 2 行第 5 个字符（包含该行前面的空字符）
- 在使用时加上一个环境变量可以获取更详细的栈展开信息：
  - Linux/macOS 等 UNIX 系统： `RUST_BACKTRACE=1 cargo run`
  - Windows 系统 (PowerShell) : `$env:RUST_BACKTRACE=1 ; cargo run`

下面让我们针对第二点进行详细展开讲解。

## backtrace 栈展开

在真实场景中，错误往往涉及到很长的调用链甚至会深入第三方库，如果没有栈展开技术，错误将难以跟踪处理，下面我们来看一个真实的崩溃例子：

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}
```

上面的代码很简单，数组只有 3 个元素，我们却尝试去访问它的第 100 号元素(数组索引从 0 开始)，那自然会崩溃。

我们的读者里不乏正义之士，此时肯定要质疑，一个简单的数组越界访问，为何要直接让程序崩溃？是不是有些小题大作了？

如果有过 C 语言的经验，即使你越界了，问题不大，我依然尝试去访问，至于这个值是不是你想要的（100 号内存地址也有可能有值，只不过是其它变量或者程序的！），抱歉，不归我管，我只负责取，你要负责管理好自己的索引访问范围。上面这种情况被称为**缓冲区溢出**，并可能会导致安全漏洞，例如攻击者可以通过索引来访问到数组后面不被允许的数据。

说实话，我宁愿程序崩溃，为什么？当你取到了一个不属于你的值，这很多时候会导致程序上的逻辑 BUG！有编程经验的人都知道这种逻辑上的 BUG 是多么难被发现和修复！因此程序直接崩溃，然后告诉我们问题发生的位置，最后我们对此进行修复，这才是最合理的软件开发流程，而不是把问题藏着掖着：

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

好的，现在成功知道问题发生的位置，但是如果想知道该问题之前经过了哪些调用环节，该怎么办？那就按照提示使用 `RUST_BACKTRACE=1 cargo run` 或 `$env:RUST_BACKTRACE=1 ; cargo run` 来再一次运行程序：

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
stack backtrace:
 0: rust_begin_unwind
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/std/src/panicking.rs:517:5
 1: core::panicking::panic_fmt
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/core/src/panicking.rs:101:14
 2: core::panicking::panic_bounds_check
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/core/src/panicking.rs:77:5
 3: <usize as core::slice::index::SliceIndex<[T]>>::index
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/core/src/slice/index.rs:184:1
 0
 4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/core/src/slice/index.rs:15:9
 5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/alloc/src/vec/mod.rs:2465:9
 6: world_hello::main
    at ./src/main.rs:4:5
 7: core::ops::function::FnOnce::call_once
    at
/rustc/59eed8a2aac0230a8b53e89d4e99d55912ba6b35/library/core/src/ops/function.rs:227:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

上面的代码就是一次栈展开(也称栈回溯)，它包含了函数调用的顺序，当然按照逆序排列：最近调用的函数排在列表的最上方。因为咱们的 `main` 函数基本是最先调用的函数了，所以排在了倒数第二位，还有一个关注点，排在最顶部最后一个调用的函数是 `rust_begin_unwind`，该函数的目的就是进行栈展开，呈现这些列表信息给我们。

要获取到栈回溯信息，你还需要开启 `debug` 标志，该标志在使用 `cargo run` 或者 `cargo build` 时自动开启（这两个操作默认是 `Debug` 运行方式）。同时，栈展开信息在不同操作系统或者 Rust 版本上也有所不同。

## panic 时的两种终止方式

当出现 `panic!` 时，程序提供了两种方式来处理终止流程：**栈展开和直接终止**。

其中，默认的方式就是 `栈展开`，这意味着 Rust 会回溯栈上数据和函数调用，因此也意味着更多的善后工作，好处是可以给出充分的报错信息和栈调用信息，便于事后的问题复盘。`直接终止`，顾名思义，不清理

数据就直接退出程序，善后工作交与操作系统来负责。

对于绝大多数用户，使用默认选择是最好的，但是当你关心最终编译出的二进制可执行文件大小时，那么可以尝试去使用直接终止的方式，例如下面的配置修改 `Cargo.toml` 文件，实现在 `release` 模式下遇到 `panic` 直接终止：

```
[profile.release]
panic = 'abort'
```

## 线程 `panic` 后，程序是否会终止？

长话短说，如果是 `main` 线程，则程序会终止，如果是其它子线程，该线程会终止，但是不会影响 `main` 线程。因此，尽量不要在 `main` 线程中做太多任务，将这些任务交由子线程去做，就算子线程 `panic` 也不会导致整个程序的结束。

具体解析见 [panic 原理剖析](#)。

## 何时该使用 `panic!`

下面让我们大概罗列下何时适合使用 `panic`，也许经过之前的学习，你已经能够对 `panic` 的使用有了自己的看法，但是我们还是会罗列一些常见的用法来加深你的理解。

先来一点背景知识，在前面章节我们粗略讲过 `Result<T, E>` 这个枚举类型，它是用来表示函数的返回结果：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

当没有错误发生时，函数返回一个用 `Result` 类型包裹的值 `Ok(T)`，当错误时，返回一个 `Err(E)`。对于 `Result` 返回我们有很多处理方法，最简单粗暴的就是 `unwrap` 和 `expect`，这两个函数非常类似，我们以 `unwrap` 举例：

```
use std::net::IpAddr;
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

上面的 `parse` 方法试图将字符串 "127.0.0.1" 解析为一个 IP 地址类型 `IpAddr`，它返回一个 `Result<IpAddr, E>` 类型，如果解析成功，则把 `Ok(IpAddr)` 中的值赋给 `home`，如果失败，则不处理 `Err(E)`，而是直接 `panic`。

因此 `unwrap` 简而言之：成功则返回值，失败则 `panic`，总之不进行任何错误处理。

## 示例、原型、测试

这几个场景下，需要快速地搭建代码，错误处理会拖慢编码的速度，也不是特别有必要，因此通过 `unwrap`、`expect` 等方法来处理是最快的。

同时，当我们回头准备做错误处理时，可以全局搜索这些方法，不遗漏地进行替换。

### 你确切的知道你的程序是正确时，可以使用 `panic`

因为 `panic` 的触发方式比错误处理要简单，因此可以让代码更清晰，可读性也更加好，当我们的代码注定是正确时，你可以用 `unwrap` 等方法直接进行处理，反正也不可能 `panic`：

```
use std::net::IpAddr;
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

例如上面的例子，"127.0.0.1" 就是 ip 地址，因此我们知道 `parse` 方法一定会成功，那么就可以直接用 `unwrap` 方法进行处理。

当然，如果该字符串是来自于用户输入，那在实际项目中，就必须用错误处理的方式，而不是 `unwrap`，否则你的程序一天要崩溃几十万次吧！

### 可能导致全局有害状态时

有害状态大概分为几类：

- 非预期的错误
- 后续代码的运行会受到显著影响
- 内存安全的问题

当错误预期会出现时，返回一个错误较为合适，例如解析器接收到格式错误的数据，HTTP 请求接收到错误的参数甚至该请求内的任何错误（不会导致整个程序有问题，只影响该次请求）。**因为错误是可预期的，因此也是可以处理的。**

当启动时某个流程发生了错误，对后续代码的运行造成了影响，那么就应该使用 `panic`，而不是处理错误后继续运行，当然你可以通过重试的方式来继续。

上面提到过，数组访问越界，就要 `panic` 的原因，这个就是属于内存安全的范畴，一旦内存访问不安全，那么我们就无法保证自己的程序会怎么运行下去，也无法保证逻辑和数据的正确性。

## panic 原理剖析

本来不想写这块儿内容，因为真的难写，但是转念一想，既然号称圣经，那么本书就得与众不同，避重就轻显然不是该有的态度。

当调用 `panic!` 宏时，它会

1. 格式化 `panic` 信息，然后使用该信息作为参数，调用 `std::panic::panic_any()` 函数
2. `panic_any` 会检查应用是否使用了 `panic hook`，如果使用了，该 `hook` 函数就会被调用（`hook` 是一个钩子函数，是外部代码设置的，用于在 `panic` 触发时，执行外部代码所需的功能）
3. 当 `hook` 函数返回后，当前的线程就开始进行栈展开：从 `panic_any` 开始，如果寄存器或者栈因为某些原因信息错乱了，那很可能该展开会发生异常，最终线程会直接停止，展开也无法继续进行
4. 展开的过程是一帧一帧的去回溯整个栈，每个帧的数据都会随之被丢弃，但是在展开过程中，你可能会遇到被用户标记为 `catching` 的帧（通过 `std::panic::catch_unwind()` 函数标记），此时用户提供的 `catch` 函数会被调用，展开也随之停止：当然，如果 `catch` 选择在内部调用 `std::panic::resume_unwind()` 函数，则展开还会继续。

还有一种情况，在展开过程中，如果展开本身 `panic` 了，那展开线程会终止，展开也随之停止。

一旦线程展开被终止或者完成，最终的输出结果是取决于哪个线程 `panic`：对于 `main` 线程，操作系统提供的终止功能 `core::intrinsics::abort()` 会被调用，最终结束当前的 `panic` 进程；如果是其它子线程，那么子线程就会简单的终止，同时信息会在稍后通过 `std::thread::join()` 进行收集。

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 可恢复的错误 Result

还记得上一节中，提到的关于文件读取的思考题吧？当时我们解决了读取文件时遇到不可恢复错误该怎么处理的问题，现在来看看，读取过程中，正常返回和遇到可以恢复的错误时该如何处理。

假设，我们有一台消息服务器，每个用户都通过 websocket 连接到该服务器来接收和发送消息，该过程就涉及到 socket 文件的读写，那么此时，如果一个用户的读写发生了错误，显然不能直接 panic，否则服务器会直接崩溃，所有用户都会断开连接，因此我们需要一种更温和的错误处理方式：Result<T, E>。

之前章节有提到过，Result<T, E> 是一个枚举类型，定义如下：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

泛型参数 T 代表成功时存入的正确值的类型，存放方式是 Ok(T)，E 代表错误时存入的错误值，存放方式是 Err(E)，枯燥的讲解永远不及代码生动准确，因此先来看下打开文件的例子：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

以上 File::open 返回一个 Result 类型，那么问题来了：

---

## 如何获知变量类型或者函数的返回类型

有几种常用的方式，此处更推荐第二种方法：

- 第一种是查询标准库或者三方库文档，搜索 File，然后找到它的 open 方法
- 在 Rust IDE 章节，我们推荐了 VSCode IDE 和 rust-analyzer 插件，如果你成功安装的话，那么就可以在 vscode 中很方便的通过代码跳转的方式查看代码，同时 rust-analyzer 插件还会对代码中的类型进行标注，非常方便好用！
- 你还可以尝试故意标记一个错误的类型，然后让编译器告诉你：

---

```
let f: u32 = File::open("hello.txt");
```

错误提示如下：

```
error[E0308]: mismatched types
--> src/main.rs:4:18
|
4 |     let f: u32 = File::open("hello.txt");
|           ^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
|
= note: expected type `u32`
         found type `std::result::Result<std::fs::File, std::io::Error>`
```

上面代码，故意将 `f` 类型标记成整形，编译器立刻不乐意了，你是在忽悠我吗？打开文件操作返回一个整形？来，大哥来告诉你返回什么：`std::result::Result<std::fs::File, std::io::Error>`，我的天呐，怎么这么长的类型！

别慌，其实很简单，首先 `Result` 本身是定义在 `std::result` 中的，但是因为 `Result` 很常用，所以就被包含在了 `prelude` 中（将常用的东东提前引入到当前作用域内），因此无需手动引入 `std::result::Result`，那么返回类型可以简化为 `Result<std::fs::File, std::io::Error>`，你看是不是很像标准的 `Result<T, E>` 枚举定义？只不过 `T` 被替换成了具体的类型 `std::fs::File`，是一个文件句柄类型，`E` 被替换成 `std::io::Error`，是一个 IO 错误类型。

这个返回值类型说明 `File::open` 调用如果成功则返回一个可以进行读写的文件句柄，如果失败，则返回一个 IO 错误：文件不存在或者没有访问文件的权限等。总之 `File::open` 需要一个方式告知调用者是成功还是失败，并同时返回具体的文件句柄(成功)或错误信息(失败)，万幸的是，这些信息可以通过 `Result` 枚举提供：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem opening the file: {:?}", error)
        },
    };
}
```

代码很清晰，对打开文件后的 `Result<T, E>` 类型进行匹配取值，如果是成功，则将 `Ok(file)` 中存放的文件句柄 `file` 赋值给 `f`，如果失败，则将 `Err(error)` 中存放的错误信息 `error` 使用 `panic` 抛出来，进而结束程序，这非常符合上文提到过的 `panic` 使用场景。

好吧，也没有那么合理 :)

## 对返回的错误进行处理

直接 `panic` 还是过于粗暴，因为实际上 IO 的错误有很多种，我们需要对部分错误进行特殊处理，而不是所有错误都直接崩溃：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => panic!("Problem opening the file: {:?}", other_error),
        },
    };
}
```

上面代码在匹配出 `error` 后，又对 `error` 进行了详细的匹配解析，最终结果：

- 如果是文件不存在错误 `ErrorKind::NotFound`，就创建文件，这里创建文件 `File::create` 也是返回 `Result`，因此继续用 `match` 对其结果进行处理：创建成功，将新的文件句柄赋值给 `f`，如果失败，则 `panic`
- 剩下的错误，一律 `panic`

虽然很清晰，但是代码还是有些啰嗦，我们会在[简化错误处理](#)一章重点讲述如何写出更优雅的错误。

## 失败就 `panic`: `unwrap` 和 `expect`

上一节中，已经看到过这两兄弟的简单介绍，这里再回来回顾下。

在不需要处理错误的场景，例如写原型、示例时，我们不想使用 `match` 去匹配 `Result<T, E>` 以获取其中的 `T` 值，因为 `match` 的穷尽匹配特性，你总要去处理下 `Err` 分支。那么有没有办法简化这个过程？有，答案就是 `unwrap` 和 `expect`。

它们的作用就是，如果返回成功，就将 `Ok(T)` 中的值取出来，如果失败，就直接 `panic`，真的勇士绝不多 BB，直接崩溃。

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

如果调用这段代码时 `hello.txt` 文件不存在，那么 `unwrap` 就将直接 `panic`：

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 2,
kind: NotFound, message: "No such file or directory" }', src/main.rs:4:37
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

`expect` 跟 `unwrap` 很像，也是遇到错误直接 `panic`，但是会带上自定义的错误提示信息，相当于重载了错误打印的函数：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

报错如下：

```
thread 'main' panicked at 'Failed to open hello.txt: Os { code: 2, kind: NotFound,
message: "No such file or directory" }', src/main.rs:4:37
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

可以看出，`expect` 相比 `unwrap` 能提供更精确的错误信息，在有些场景也会更加实用。

## 传播错误

咱们的程序几乎不太可能只有 A->B 形式的函数调用，一个设计良好的程序，一个功能涉及十几层的函数调用都有可能。而错误处理也往往不是哪里调用出错，就在哪里处理，实际应用中，大概率会把错误层层上传然后交给调用链的上游函数进行处理，错误传播将极为常见。

例如以下函数从文件中读取用户名，然后将结果进行返回：

```

use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    // 打开文件, f是`Result<文件句柄, io::Error>`
    let f = File::open("hello.txt");

    let mut f = match f {
        // 打开文件成功, 将file句柄赋值给f
        Ok(file) => file,
        // 打开文件失败, 将错误返回(向上传播)
        Err(e) => return Err(e),
    };
    // 创建动态字符串s
    let mut s = String::new();
    // 从f文件句柄读取数据并写入s中
    match f.read_to_string(&mut s) {
        // 读取成功, 返回Ok封装的字符串
        Ok(_) => Ok(s),
        // 将错误向上传播
        Err(e) => Err(e),
    }
}

```

有几点值得注意：

- 该函数返回一个 `Result<String, io::Error>` 类型，当读取用户名成功时，返回 `Ok(String)`，失败时，返回 `Err(io::Error)`
- `File::open` 和 `f.read_to_string` 返回的 `Result<T, E>` 中的 `E` 就是 `io::Error`

由此可见，该函数将 `io::Error` 的错误往上进行传播，该函数的调用者最终会对 `Result<String, io::Error>` 进行再处理，至于怎么处理就是调用者的事，如果是错误，它可以选择继续向上传播错误，也可以直接 `panic`，亦或将具体的错误原因包装后写入 socket 中呈现给终端用户。

但是上面的代码也有自己的问题，那就是太长了(优秀的程序员身上的优点极多，其中最大的优点就是懒)，我自认为也有那么一点点优秀，因此见不得这么啰嗦的代码，下面咱们来讲讲如何简化它。

## 传播界的大明星：？

大明星出场，必须得有排面，来看看 ? 的排面：

```

use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}

```

看到没，这就是排面，相比前面的 `match` 处理错误的函数，代码直接减少了一半不止，但是，一山更比一山难，看不懂啊！

其实 `?` 就是一个宏，它的作用跟上面的 `match` 几乎一模一样：

```

let mut f = match f {
    // 打开文件成功，将file句柄赋值给f
    Ok(file) => file,
    // 打开文件失败，将错误返回(向上传播)
    Err(e) => return Err(e),
};

```

如果结果是 `Ok(T)`，则把 `T` 赋值给 `f`，如果结果是 `Err(E)`，则返回该错误，所以 `?` 特别适合用来传播错误。

虽然 `?` 和 `match` 功能一致，但是事实上 `?` 会更胜一筹。何解？

想象一下，一个设计良好的系统中，肯定有自定义的错误特征，错误之间很可能会存在上下级关系，例如标准库中的 `std::io::Error` 和 `std::error::Error`，前者是 IO 相关的错误结构体，后者是一个最最通用的标准错误特征，同时前者实现了后者，因此 `std::io::Error` 可以转换为 `std::error::Error`。

明白了以上的错误转换，`?` 的更胜一筹就很好理解了，它可以自动进行类型提升（转换）：

```

fn open_file() -> Result<File, Box<dyn std::error::Error>> {
    let mut f = File::open("hello.txt")?;
    Ok(f)
}

```

上面代码中 `File::open` 报错时返回的错误是 `std::io::Error` 类型，但是 `open_file` 函数返回的错误类型是 `std::error::Error` 的特征对象，可以看到一个错误类型通过 `?` 返回后，变成了另一个错误类型，这就是 `?` 的神奇之处。

根本原因是在于标准库中定义的 `From` 特征，该特征有一个方法 `from`，用于把一个类型转成另外一个类型，`?` 可以自动调用该方法，然后进行隐式类型转换。因此只要函数返回的错误 `ReturnError` 实现了

`From<OtherError>` 特征，那么`?` 就会自动把 `OtherError` 转换为 `ReturnError`。

这种转换非常好用，意味着你可以用一个大而全的 `ReturnError` 来覆盖所有错误类型，只需要为各种子错误类型实现这种转换即可。

强中自有强中手，一码更比一码短：

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

瞧见没？`?` 还能实现链式调用，`File::open` 遇到错误就返回，没有错误就将 `Ok` 中的值取出来用于下一个方法调用，简直太精妙了，从 Go 语言过来的我，内心狂喜（其实学 Rust 的苦和痛我才不会告诉你们）。

不仅有更强，还要有最强，我不信还有人比我更短(不要误解)：

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

fn read_username_from_file() -> Result<String, io::Error> {
    // read_to_string是定义在std::io中的方法，因此需要在上面进行引用
    fs::read_to_string("hello.txt")
}
```

从文件读取数据到字符串中，是比较常见的操作，因此 Rust 标准库为我们提供了 `fs::read_to_string` 函数，该函数内部会打开一个文件、创建 `String`、读取文件内容最后写入字符串并返回，因为该函数其实与本章讲的内容关系不大，因此放在最后来讲，其实只是我想震你们一下 :)

## ? 用于 Option 的返回

`?` 不仅仅可以用于 `Result` 的传播，还能用于 `Option` 的传播，再回忆下 `option` 的定义：

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

Result 通过 ? 返回错误，那么 Option 就通过 ? 返回 None：

```
fn first(arr: &[i32]) -> Option<&i32> {
    let v = arr.get(0)?;
    Some(v)
}
```

上面的函数中，arr.get 返回一个 Option<&i32> 类型，因为 ? 的使用，如果 get 的结果是 None，则直接返回 None，如果是 Some(&i32)，则把里面的值赋给 v。

其实这个函数有些画蛇添足，我们完全可以写出更简单的版本：

```
fn first(arr: &[i32]) -> Option<&i32> {
    arr.get(0)
}
```

有一句话怎么说？没有需求，制造需求也要上.....大家别跟我学习，这是软件开发大忌。只能用代码洗洗眼了：

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

上面代码展示了在链式调用中使用 ? 提前返回 None 的用法，.next 方法返回的是 Option 类型：如果返回 Some(&str)，那么继续调用 chars 方法，如果返回 None，则直接从整个函数中返回 None，不再继续进行链式调用。

## 新手用 ? 常会犯的错误

初学者在用 ? 时，老是会犯错，例如写出这样的代码：

```
fn first(arr: &[i32]) -> Option<&i32> {
    arr.get(0)?
}
```

这段代码无法通过编译，切记：? 操作符需要一个变量来承载正确的值，这个函数只会返回 Some(&i32) 或者 None，只有错误值能直接返回，正确的值不行，所以如果数组中存在 0 号元素，那么函数第二行使用 ? 后的返回类型为 &i32 而不是 Some(&i32)。因此 ? 只能用于以下形式：

- let v = xxx()?;
- xxx()?.yyy()?;

## 带返回值的 main 函数

在了解了 `?` 的使用限制后，这段代码你很容易看出它无法编译：

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

运行后会报错：

```
$ cargo run
...
the `?` operator can only be used in a function that returns `Result` or `Option`
(or another type that implements `FromResidual`)
--> src/main.rs:4:48
|
3 | fn main() {
| ----- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
|                                     ^ cannot use the `?` operator in a
function that returns `()`
|
= help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`
```

因为 `?` 要求 `Result<T, E>` 形式的返回值，而 `main` 函数的返回是 `()`，因此无法满足，那是不是就无法解了呢？

实际上 Rust 还支持另外一种形式的 `main` 函数：

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box> {
    let f = File::open("hello.txt")?;

    Ok(())
}
```

这样就能使用 `?` 提前返回了，同时我们又一次看到了 `Box<dyn Error>` 特征对象，因为 `std::error::Error` 是 Rust 中抽象层次最高的错误，其它标准库中的错误都实现了该特征，因此我们可以用该特征对象代表一切错误，就算 `main` 函数中调用任何标准库函数发生错误，都可以通过 `Box<dyn Error>` 这个特征对象进行返回。

至于 `main` 函数可以有多种返回值，那是因为实现了 `std::process::Termination` 特征，目前为止该特征还没进入稳定版 Rust 中，也许未来你可以为自己的类型实现该特征！

## try!

在 `?` 横空出世之前( Rust 1.13 )，Rust 开发者还可以使用 `try!` 来处理错误，该宏的大致定义如下：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

简单看一下与 `?` 的对比：

```
// `?`  
let x = function_with_error()?;
// 若返回 Err，则立刻返回；若返回 Ok(255)，则将 x 的值设置为  
255  
  
// `try!()`  
let x = try!(function_with_error());
```

可以看出 `?` 的优势非常明显，何况 `?` 还能做链式调用。

总之，`try!` 作为前浪已经死在了沙滩上，**在当前版本中，我们要尽量避免使用 `try!`。**

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 包和模块

当工程规模变大时，把代码写到一个甚至几个文件中，都是不太聪明的做法，可能存在以下问题：

1. 单个文件过大，导致打开、翻页速度大幅变慢
2. 查询和定位效率大幅降低，类比下，你会把所有知识内容放在一个几十万字的文档中吗？
3. 只有一个代码层次：函数，难以维护和协作，想象一下你的操作系统只有一个根目录，剩下的都是单层子目录会如何：`disaster`
4. 容易滋生 Bug

同时，将大的代码文件拆分成包和模块，还允许我们实现代码抽象和复用：将你的代码封装好后提供给用户，那么用户只需要调用公共接口即可，无需知道内部该如何实现。

因此，跟其它语言一样，Rust 也提供了相应概念用于代码的组织管理：

- 项目(Packages)：一个 `Cargo` 提供的 `feature`，可以用来构建、测试和分享包
- 包(Crate)：一个由多个模块组成的树形结构，可以作为三方库进行分发，也可以生成可执行文件进行运行
- 模块(Module)：可以一个文件多个模块，也可以一个文件一个模块，模块可以被认为是真实项目中的代码组织单元

下面，让我们——来学习这些概念以及如何在实践中运用。

# 包和 Package

当读者按照章节顺序读到本章时，意味着你已经几乎具备了参与真实项目开发的能力。但是真实项目远比我们之前的 `cargo new` 的默认目录结构要复杂，好在，Rust 为我们提供了强大的包管理工具：

- **项目(Package)**: 可以用来构建、测试和分享包
- **工作空间(WorkSpace)**: 对于大型项目，可以进一步将多个包联合在一起，组织成工作空间
- **包(Crate)**: 一个由多个模块组成的树形结构，可以作为三方库进行分发，也可以生成可执行文件进行运行
- **模块(Module)**: 可以一个文件多个模块，也可以一个文件一个模块，模块可以被认为是真实项目中的代码组织单元

## 定义

其实项目 Package 和包 Crate 很容易被搞混，甚至在很多书中，这两者都是不分的，但是由于官方对此做了明确的区分，因此我们会在本章节中试图(挣扎着)理清这个概念。

### 包 Crate

对于 Rust 而言，包是一个独立的可编译单元，它编译后会生成一个可执行文件或者一个库。

一个包会将相关联的功能打包在一起，使得该功能可以很方便的在多个项目中分享。例如标准库中没有提供但是在三方库中提供的 `rand` 包，它提供了随机数生成的功能，我们只需要将该包通过 `use rand;` 引入到当前项目的作用域中，就可以在项目中使用 `rand` 的功能：`rand::XXX`。

同一个包中不能有同名的类型，但是在不同包中就可以。例如，虽然 `rand` 包中，有一个 `Rng` 特征，可是我们依然可以在自己的项目中定义一个 `Rng`，前者通过 `rand::Rng` 访问，后者通过 `Rng` 访问，对于编译器而言，这两者的边界非常清晰，不会存在引用歧义。

### 项目 Package

鉴于 Rust 团队标新立异的起名传统，以及包的名称被 `crate` 占用，库的名称被 `library` 占用，经过斟酌，我们决定将 `Package` 翻译成项目，你也可以理解为工程、软件包。

由于 `Package` 就是一个项目，因此它包含有独立的 `Cargo.toml` 文件，以及因为功能性被组织在一起的一个或多个包。一个 `Package` 只能包含一个库(library)类型的包，但是可以包含多个二进制可执行类型

的包。

## 二进制 Package

让我们来创建一个二进制 Package：

```
$ cargo new my-project
    Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

这里，Cargo 为我们创建了一个名称是 `my-project` 的 Package，同时在其中创建了 `Cargo.toml` 文件，可以看一下该文件，里面并没有提到 `src/main.rs` 作为程序的入口，原因是 Cargo 有一个惯例：  
**`src/main.rs` 是二进制包的根文件，该二进制包的包名跟所属 Package 相同，在这里都是 `my-project`，所有的代码执行都从该文件中的 `fn main()` 函数开始。**

使用 `cargo run` 可以运行该项目，输出：Hello, world!。

## 库 Package

再来创建一个库类型的 Package：

```
$ cargo new my-lib --lib
    Created library `my-lib` package
$ ls my-lib
Cargo.toml
src
$ ls my-lib/src
lib.rs
```

首先，如果你试图运行 `my-lib`，会报错：

```
$ cargo run
error: a bin target must be available for `cargo run`
```

原因是库类型的 Package 只能作为三方库被其它项目引用，而不能独立运行，只有之前的二进制 Package 才可以运行。

与 `src/main.rs` 一样，Cargo 知道，如果一个 Package 包含有 `src/lib.rs`，意味它包含有一个库类型的同名包 `my-lib`，该包的根文件是 `src/lib.rs`。

## 易混淆的 Package 和包

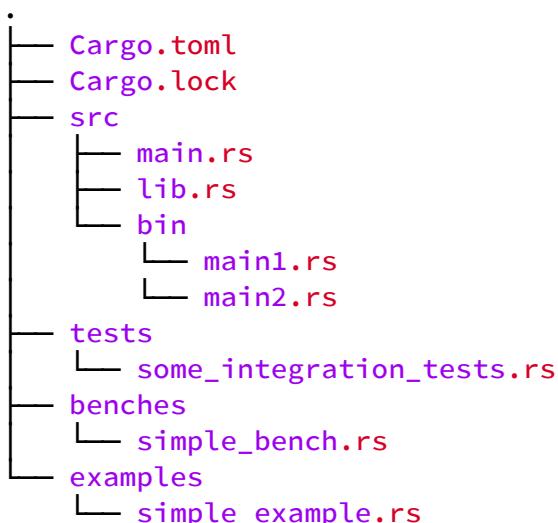
看完上面，相信大家看出来为何 Package 和包容易被混淆了吧？因为你用 `cargo new` 创建的 Package 和它其中包含的包是同名的！

不过，只要你牢记 Package 是一个项目工程，而包只是一个编译单元，基本上也就不会混淆这个两个概念了：`src/main.rs` 和 `src/lib.rs` 都是编译单元，因此它们都是包。

## 典型的 Package 结构

上面创建的 Package 中仅包含 `src/main.rs` 文件，意味着它仅包含一个二进制同名包 `my-project`。如果一个 Package 同时拥有 `src/main.rs` 和 `src/lib.rs`，那就意味着它包含两个包：库包和二进制包，这两个包名也都是 `my-project` —— 都与 Package 同名。

一个真实项目中典型的 Package，会包含多个二进制包，这些包文件被放在 `src/bin` 目录下，每一个文件都是独立的二进制包，同时也会包含一个库包，该包只能存在一个 `src/lib.rs`：



- 唯一库包：`src/lib.rs`
- 默认二进制包：`src/main.rs`，编译后生成的可执行文件与 Package 同名
- 其余二进制包：`src/bin/main1.rs` 和 `src/bin/main2.rs`，它们会分别生成一个文件同名的二进制可执行文件
- 集成测试文件：`tests` 目录下
- 基准性能测试 `benchmark` 文件：`benches` 目录下
- 项目示例：`examples` 目录下

这种目录结构基本上是 Rust 的标准目录结构，在 GitHub 的大多数项目上，你都将看到它的身影。

理解了包的概念，我们再来看看构成包的基本单元：模块。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 模块 Module

在本章节，我们将深入讲讲 Rust 的代码构成单元：模块。使用模块可以将包中的代码按照功能性进行重组，最终实现更好的可读性及易用性。同时，我们还能非常灵活地去控制代码的可见性，进一步强化 Rust 的安全性。

## 创建嵌套模块

小旅馆，sorry，是小餐馆，相信大家都挺熟悉的，学校外的估计也没少去，那么咱就用小餐馆为例，来看看 Rust 的模块该如何使用。

使用 `cargo new --lib restaurant` 创建一个小餐馆，注意，这里创建的是一个库类型的 Package，然后将以下代码放入 `src/lib.rs` 中：

```
// 餐厅前厅，用于吃饭
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

以上的代码创建了三个模块，有几点需要注意的：

- 使用 `mod` 关键字来创建新模块，后面紧跟着模块名称
- 模块可以嵌套，这里嵌套的原因是招待客人和服务都发生在前厅，因此我们的代码模拟了真实场景
- 模块中可以定义各种 Rust 类型，例如函数、结构体、枚举、特征等
- 所有模块均定义在同一个文件中

类似上述代码中所做的，使用模块，我们就能将功能相关的代码组织到一起，然后通过一个模块名称来说明这些代码为何被组织在一起。这样其它程序员在使用你的模块时，就可以更快地理解和上手。

## 模块树

在[上一节](#)中，我们提到过 `src/main.rs` 和 `src/lib.rs` 被称为包根(crate root)，这个奇葩名称的来源(我不想承认是自己翻译水平太烂,-)是由于这两个文件的内容形成了一个模块 `crate`，该模块位于包的树形结构(由模块组成的树形结构)的根部：

```
crate
└── front_of_house
    ├── hosting
    │   └── add_to_waitlist
    │       └── seat_at_table
    └── serving
        ├── take_order
        ├── serve_order
        └── take_payment
```

这颗树展示了模块之间**彼此的嵌套**关系，因此被称为**模块树**。其中 `crate` 包根是 `src/lib.rs` 文件，包根文件中的三个模块分别形成了模块树的剩余部分。

### 父子模块

如果模块 A 包含模块 B，那么 A 是 B 的父模块，B 是 A 的子模块。在上例中，`front_of_house` 是 `hosting` 和 `serving` 的父模块，反之，后两者是前者的子模块。

聪明的读者，应该能联想到，模块树跟计算机上文件系统目录树的相似之处。不仅仅是组织结构上的相似，就连使用方式都很相似：每个文件都有自己的路径，用户可以通过这些路径使用它们，在 Rust 中，我们也通过路径的方式来引用模块。

## 用路径引用模块

想要调用一个函数，就需要知道它的路径，在 Rust 中，这种路径有两种形式：

- **绝对路径**，从包根开始，路径名以包名或者 `crate` 作为开头
- **相对路径**，从当前模块开始，以 `self`，`super` 或当前模块的标识符作为开头

让我们继续经营那个惨淡的小餐馆，这次为它实现一个小功能：文件名：`src/lib.rs`

```

mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 绝对路径
    crate::front_of_house::hosting::add_to_waitlist();

    // 相对路径
    front_of_house::hosting::add_to_waitlist();
}

```

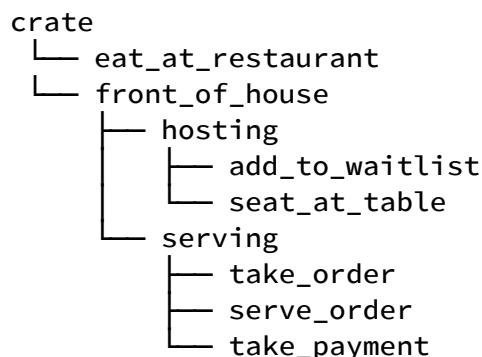
上面的代码为了简化实现，省去了其余模块和函数，这样可以把关注点放在函数调用上。`eat_at_restaurant` 是一个定义在包根中的函数，在该函数中使用了两种方式对 `add_to_waitlist` 进行调用。

## 绝对路径引用

因为 `eat_at_restaurant` 和 `add_to_waitlist` 都定义在一个包中，因此在绝对路径引用时，可以直接以 `crate` 开头，然后逐层引用，每一层之间使用 `::` 分隔：

```
crate::front_of_house::hosting::add_to_waitlist();
```

对比下之前的模块树：



可以看出，绝对路径的调用，完全符合了模块树的层级递进，非常符合直觉，如果类比文件系统，就跟使用绝对路径调用可执行程序差不多：`/front_of_house/hosting/add_to_waitlist`，使用 `crate` 作为开始就和使用 `/` 作为开始一样。

## 相对路径引用

再回到模块树中，因为 `eat_at_restaurant` 和 `front_of_house` 都处于包根 `crate` 中，因此相对路径可以使用 `front_of_house` 作为开头：

```
front_of_house::hosting::add_to_waitlist();
```

如果类比文件系统，那么它类似于调用同一个目录下的程序，你可以这么做：

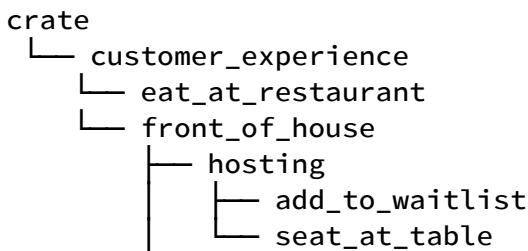
`front_of_house/hosting/add_to_waitlist`，嗯也很符合直觉。

## 绝对还是相对？

如果只是为了引用到指定模块中的对象，那么两种都可以，但是在实际使用时，需要遵循一个原则：**当代码被挪动位置时，尽量减少引用路径的修改**，相信大家都遇到过，修改了某处代码，导致所有路径都要挨个替换，这显然不是好的路径选择。

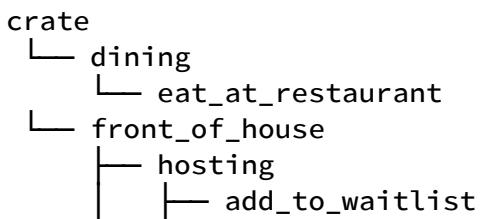
回到之前的例子，如果我们把 `front_of_house` 模块和 `eat_at_restaurant` 移动到一个模块中 `customer_experience`，那么绝对路径的引用方式就必须进行修改：

`crate::customer_experience::front_of_house ...`，但是假设我们使用的相对路径，那么该路径就无需修改，因为它们两个的相对位置其实没有变：



从新的模块树中可以很清晰的看出这一点。

再比如，其它的都不动，把 `eat_at_restaurant` 移动到模块 `dining` 中，如果使用相对路径，你需要修改该路径，但如果使用的是绝对路径，就无需修改：



不过，如果不确定哪个好，你可以考虑优先使用绝对路径，因为调用的地方和定义的地方往往是分离的，而定义的地方较少会变动。

# 代码可见性

让我们运行下面(之前)的代码：

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 绝对路径
    crate::front_of_house::hosting::add_to_waitlist();

    // 相对路径
    front_of_house::hosting::add_to_waitlist();
}
```

意料之外的报错了，毕竟看上去确实很简单且没有任何问题：

```
error[E0603]: module `hosting` is private
--> src/lib.rs:9:28
9 |     crate::front_of_house::hosting::add_to_waitlist();
|          ^^^^^^ private module
```

错误信息很清晰：`hosting` 模块是私有的，无法在包根进行访问，那么为何 `front_of_house` 模块就可以访问？因为它和 `eat_at_restaurant` 同属于一个包根作用域内，同一个模块内的代码自然不存在私有化问题(所以我们之前章节的代码都没有报过这个错误！)。

模块不仅仅对于组织代码很有用，它还能定义代码的私有化边界：在这个边界内，什么内容能让外界看到，什么内容不能，都有很明确的定义。因此，如果希望让函数或者结构体等类型变成私有化的，可以使用模块。

Rust 出于安全的考虑，默认情况下，所有的类型都是私有化的，包括函数、方法、结构体、枚举、常量，是的，就连模块本身也是私有化的。在中国，父亲往往不希望孩子拥有小秘密，但是在 Rust 中，**父模块完全无法访问子模块中的私有项，但是子模块却可以访问父模块、父父..模块的私有项**。

## pub 关键字

类似其它语言的 `public` 或者 Go 语言中的首字母大写，Rust 提供了 `pub` 关键字，通过它你可以控制模块和模块中指定项的可见性。

由于之前的解释，我们知道了只需要将 `hosting` 模块标记为对外可见即可：

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

/*--- snip ----*/
```

但是不幸的是，又报错了：

```
error[E0603]: function `add_to_waitlist` is private
--> src/lib.rs:12:30
|
12 |     front_of_house::hosting::add_to_waitlist();
      ^^^^^^^^^^^^^^^^^^ private function
```

哦？难道模块可见还不够，还需要将函数 `add_to_waitlist` 标记为可见的吗？是的，没错，模块可见性不代表模块内部项的可见性，模块的可见性仅仅是允许其它模块去引用它，但是想要引用它内部的项，还得继续将对应的项标记为 `pub`。

在实际项目中，一个模块需要对外暴露的数据和 API 往往就寥寥数个，如果将模块标记为可见代表着内部项也全部对外可见，那你是不是还得把那些不可见的，一个一个标记为 `private`？反而是更麻烦的多。

既然知道了如何解决，那么我们为函数也标记上 `pub`：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

/*--- snip ----*/
```

Bang，顺利通过编译，感觉自己又变强了。

## 使用 `super` 引用模块

在[用路径引用模块](#)中，我们提到了相对路径有三种方式开始：`self`、`super` 和 `crate` 或者模块名，其中第三种在前面已经讲到过，现在来看看通过 `super` 的方式引用模块项。

`super` 代表的是父模块为开始的引用方式，非常类似于文件系统中的 `..` 语法：`../a/b` 文件名：`src/lib.rs`

```
fn serve_order() {}

// 厨房模块
mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }

    fn cook_order() {}
}
```

嗯，我们的小餐馆又完善了，终于有厨房了！看来第一个客人也快可以有了。。。在厨房模块中，使用 `super::serve_order` 语法，调用了父模块(包根)中的 `serve_order` 函数。

那么你可能会问，为何不使用 `crate::serve_order` 的方式？额，其实也可以，不过如果你确定未来这种层级关系不会改变，那么 `super::serve_order` 的方式会更稳定，未来就算它们都不在包根了，依然无需修改引用路径。所以路径的选用，往往还是取决于场景，以及未来代码的可能走向。

## 使用 `self` 引用模块

`self` 其实就是引用自身模块中的项，也就是说和我们之前章节的代码类似，都调用同一模块中的内容，区别在于之前章节中直接通过名称调用即可，而 `self`，你得多此一举：

```
fn serve_order() {
    self::back_of_house::cook_order()
}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        crate::serve_order();
    }

    pub fn cook_order() {}
}
```

是的，多此一举，因为完全可以直接调用 `back_of_house`，但是 `self` 还有一个大用处，在下一节中我们会讲。

## 结构体和枚举的可见性

为何要把结构体和枚举的可见性单独拎出来讲呢？因为这两个家伙的成员字段拥有完全不同的可见性：

- 将结构体设置为 `pub`，但它的所有字段依然是私有的
- 将枚举设置为 `pub`，它的所有字段也将对外可见

原因在于，枚举和结构体的使用方式不一样。如果枚举的成员对外不可见，那该枚举将一点用都没有，因此枚举成员的可见性自动跟枚举可见性保持一致，这样可以简化用户的使用。

而结构体的应用场景比较复杂，其中的字段也往往部分在 A 处被使用，部分在 B 处被使用，因此无法确定成员的可见性，那索性就设置为全部不可见，将选择权交给程序员。

## 模块与文件分离

在之前的例子中，我们所有的模块都定义在 `src/lib.rs` 中，但是当模块变多或者变大时，需要将模块放入一个单独的文件中，让代码更好维护。

现在，把 `front_of_house` 前厅分离出来，放入一个单独的文件中 `src/front_of_house.rs`：

```
pub mod hosting {  
    pub fn add_to_waitlist() {}  
}
```

然后，将以下代码留在 `src/lib.rs` 中：

```
mod front_of_house;  
  
pub use crate::front_of_house::hosting;  
  
pub fn eat_at_restaurant() {  
    hosting::add_to_waitlist();  
    hosting::add_to_waitlist();  
    hosting::add_to_waitlist();  
}
```

so easy！其实跟之前在同一个文件中也没有太大的不同，但是有几点值得注意：

- `mod front_of_house;` 告诉 Rust 从另一个和模块 `front_of_house` 同名的文件中加载该模块的内容
- 使用绝对路径的方式来引用 `hosting` 模块：`crate::front_of_house::hosting;`

需要注意的是，和之前代码中 `mod front_of_house{..}` 的完整模块不同，现在的代码中，模块的声明和实现是分离的，实现是在单独的 `front_of_house.rs` 文件中，然后通过 `mod front_of_house;` 这条声明语句从该文件中把模块内容加载进来。因此我们可以认为，模块 `front_of_house` 的定义还是在 `src/lib.rs` 中，只不过模块的具体内容被移动到了 `src/front_of_house.rs` 文件中。

在这里出现了一个新的关键字 `use`，联想到其它章节我们见过的标准库引入 `use std::fmt;`，可以大致猜测，该关键字用来将外部模块中的项引入到当前作用域中来，这样无需冗长的父模块前缀即可调用：`hosting::add_to_waitlist();`，在下节中，我们将对 `use` 进行详细的讲解。

当一个模块有许多子模块时，我们也可以通过文件夹的方式来组织这些子模块。

在上述例子中，我们可以创建一个目录 `front_of_house`，然后在文件夹里创建一个 `hosting.rs` 文件，`hosting.rs` 文件现在就剩下：

```
pub fn add_to_waitlist() {}
```

现在，我们尝试编译程序，很遗憾，编译器报错：

```
error[E0583]: file not found for module `front_of_house`
--> src/lib.rs:3:1
  |
1 | mod front_of_house;
  | ^^^^^^^^^^^^^^^^^^
  |
= help: to create the module `front_of_house`, create file "src/front_of_house.rs"
or "src/front_of_house/mod.rs"
```

是的，如果需要将文件夹作为一个模块，我们需要进行显示指定暴露哪些子模块。按照上述的报错信息，我们有两种方法：

- 在 `front_of_house` 目录里创建一个 `mod.rs`，如果你使用的 `rustc` 版本 1.30 之前，这是唯一的方法。
- 在 `front_of_house` 同级目录里创建一个与模块（目录）同名的 `rs` 文件 `front_of_house.rs`，在新版本里，更建议使用这样的命名方式来避免项目中存在大量同名的 `mod.rs` 文件（Python 点了个踩）。

而无论是上述哪个方式创建的文件，其内容都是一样的，你需要定义你的子模块（子模块名与文件名相同）：

```
pub mod hosting;
// pub mod serving;
```

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

# 使用 use 及受限可见性

如果代码中，通篇都是 `crate::front_of_house::hosting::add_to_waitlist` 这样的函数调用形式，我不知道有谁会喜欢，也许靠代码行数赚工资的人会很喜欢，但是强迫症肯定受不了，悲伤的是程序员大多都有强迫症。。。

因此我们需要一个办法来简化这种使用方式，在 Rust 中，可以使用 `use` 关键字把路径提前引入到当前作用域中，随后的调用就可以省略该路径，极大地简化了代码。

## 基本引入方式

在 Rust 中，引入模块中的项有两种方式：[绝对路径和相对路径](#)，这两者在前面章节都有讲过，就不再赘述，先来看看使用绝对路径的引入方式。

### 绝对路径引入模块

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

这里，我们使用 `use` 和绝对路径的方式，将 `hosting` 模块引入到当前作用域中，然后只需通过 `hosting::add_to_waitlist` 的方式，即可调用目标模块中的函数，相比 `crate::front_of_house::hosting::add_to_waitlist()` 的方式要简单的多，那么还能更简单吗？

### 相对路径引入模块中的函数

在下面代码中，我们不仅要使用相对路径进行引入，而且与上面引入 `hosting` 模块不同，直接引入该模块中的 `add_to_waitlist` 函数：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
    add_to_waitlist();
    add_to_waitlist();
}
```

很明显，三兄弟又变得更短了，不过，怎么觉得这句话怪怪的。。

## 引入模块还是函数

从使用简洁性来说，引入函数自然是更甚一筹，但是在某些时候，引入模块会更好：

- 需要引入同一个模块的多个函数
- 作用域中存在同名函数

在以上两种情况中，使用 `use front_of_house::hosting;` 引入模块要比 `use front_of_house::hosting::add_to_waitlist;` 引入函数更好。

例如，如果想使用 `HashMap`，那么直接引入该结构体是比引入模块更好的选择，因为在 `collections` 模块中，我们只需要使用一个 `HashMap` 结构体：

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

其实严格来说，对于引用方式并没有需要遵守的惯例，主要还是取决于你的喜好，不过我们建议：**优先使用最细粒度(引入函数、结构体等)的引用方式，如果引起了某种麻烦(例如前面两种情况)，再使用引入模块的方式。**

# 避免同名引用

根据上一章节的内容，我们只要保证同一个模块中不存在同名项就行，模块之间、包之间的同名，谁管得着谁啊，话虽如此，一起看看，如果遇到同名的情况该如何处理。

## 模块::函数

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

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

上面的例子给出了很好的解决方案，使用模块引入的方式，具体的 `Result` 通过 `模块::Result` 的方式进行调用。

可以看出，避免同名冲突的关键，就是使用**父模块的方式来调用**，除此之外，还可以给予引入的项起一个别名。

## as 别名引用

对于同名冲突问题，还可以使用 `as` 关键字来解决，它可以赋予引入项一个全新的名称：

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

如上所示，首先通过 `use std::io::Result` 将 `Result` 引入到作用域，然后使用 `as` 给予它一个全新的名称 `IoResult`，这样就不会再产生冲突：

- `Result` 代表 `std::fmt::Result`
- `IoResult` 代表 `std::io::Result`

## 引入项再导出

当外部的模块项 A 被引入到当前模块中时，它的可见性自动被设置为私有的，如果你希望允许其它外部代码引用我们的模块项 A，那么可以对它进行再导出：

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

如上，使用 `pub use` 即可实现。这里 `use` 代表引入 `hosting` 模块到当前作用域，`pub` 表示将该引入的内容再度设置为可见。

当你希望将内部的实现细节隐藏起来或者按照某个目的组织代码时，可以使用 `pub use` 再导出，例如统一使用一个模块来提供对外的 API，那该模块就可以引入其它模块中的 API，然后进行再导出，最终对于用户来说，所有的 API 都是由一个模块统一提供的。

## 使用第三方包

之前我们一直在引入标准库模块或者自定义模块，现在来引入下第三方包中的模块，关于如何引入外部依赖，我们在 [Cargo 入门](#) 中就有讲，这里直接给出操作步骤：

1. 修改 `Cargo.toml` 文件，在 `[dependencies]` 区域添加一行： `rand = "0.8.3"`
2. 此时，如果你用的是 `VSCode` 和 `rust-analyzer` 插件，该插件会自动拉取该库，你可能需要等它完成后，再进行下一步（`VSCode` 左下角有提示）

好了，此时，`rand` 包已经被我们添加到依赖中，下一步就是在代码中使用：

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..101);
}
```

这里使用 `use` 引入了第三方包 `rand` 中的 `Rng` 特征，因为我们需要调用的 `gen_range` 方法定义在该特征中。

### crates.io, lib.rs

Rust 社区已经为我们贡献了大量高质量的第三方包，你可以在 `crates.io` 或者 `lib.rs` 中检索和使用，从目前来说查找包更推荐 `lib.rs`，搜索功能更强大，内容展示也更加合理，但是下载依赖包还是得用 `crates.io`。

你可以在网站上搜索 `rand` 包，看看它的文档使用方式是否和我们之前引入方式相一致：在网上找到想要的包，然后将你想要的包和版本信息写入到 `Cargo.toml` 中。

## 使用 {} 简化引入方式

对于以下一行一行的引入方式：

```
use std::collections::HashMap;
use std::collections::BTreeMap;
use std::collections::HashSet;

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

可以使用 `{}` 来一起引入进来，在大型项目中，使用这种方式来引入，可以减少大量 `use` 的使用：

```
use std::collections::{HashMap, BTreeMap, HashSet};
use std::{cmp::Ordering, io};
```

对于下面的同时引入模块和模块中的项：

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

可以使用 `{}` 的方式进行简化：

```
use std::io::{self, Write};
```

## self

上面使用到了模块章节提到的 `self` 关键字，用来替代模块自身，结合上一节中的 `self`，可以得出它在模块中的两个用途：

- `use self::xxx`，表示加载当前模块中的 `xxx`。此时 `self` 可省略
- `use xxx::{self, yyy}`，表示，加载当前路径下模块 `xxx` 本身，以及模块 `xxx` 下的 `yyy`

## 使用 \* 引入模块下的所有项

对于之前一行一行引入 `std::collections` 的方式，我们还可以使用

```
use std::collections::*;


```

以上这种方式来引入 `std::collections` 模块下的所有公共项，这些公共项自然包含了 `HashMap`，`HashSet` 等想手动引入的集合类型。

当使用 `*` 来引入的时候要格外小心，因为你很难知道到底哪些被引入到了当前作用域中，有哪些会和你自己程序中的名称相冲突：

```
use std::collections::*;

struct HashMap;
fn main() {
    let mut v = HashMap::new();
    v.insert("a", 1);
}
```

以上代码中，`std::collection::HashMap` 被 `*` 引入到当前作用域，但是由于存在另一个同名的结构体，因此 `HashMap::new` 根本不存在，因为对于编译器来说，本地同名类型的优先级更高。

在实际项目中，这种引用方式往往用于快速写测试代码，它可以把所有东西一次性引入到 `tests` 模块中。

## 受限的可见性

在上一节中，我们学习了[可见性](#)这个概念，这也是模块体系中最为核心的概念，控制了模块中哪些内容可以被外部看见，但是在实际使用时，光被外面看到还不行，我们还想控制哪些人能看，这就是 Rust 提供的受限可见性。

例如，在 Rust 中，包是一个模块树，我们可以通过 `pub(crate) item;` 这种方式来实现：`item` 虽然是对外可见的，但是只在当前包内可见，外部包无法引用到该 `item`。

所以，如果我们想要让某一项可以在整个包中都可以被使用，那么有两种办法：

- 在包根中定义一个非 `pub` 类型的 `x` (父模块的项对子模块都是可见的，因此包根中的项对模块树上的所有模块都可见)
- 在子模块中定义一个 `pub` 类型的 `y`，同时通过 `use` 将其引入到包根

```
mod a {
    pub mod b {
        pub fn c() {
            println!("{:?}", crate::x);
        }
    }

    #[derive(Debug)]
    pub struct Y;
}

#[derive(Debug)]
struct X;
use a::b::Y;
fn d() {
    println!("{:?}", Y);
}
```

以上代码充分说明了之前两种办法的使用方式，但是有时我们会遇到这两种方法都不太好用的时候。例如希望对于某些特定的模块可见，但是对于其他模块又不可见：

```
// 目标: `a` 导出 `I`、`bar` and `foo`，其他的不导出
pub mod a {
    pub const I: i32 = 3;

    fn semisecret(x: i32) -> i32 {
        use self::b::c::J;
        x + J
    }

    pub fn bar(z: i32) -> i32 {
        semisecret(I) * z
    }
    pub fn foo(y: i32) -> i32 {
        semisecret(I) + y
    }
}

mod b {
    mod c {
        const J: i32 = 4;
    }
}
}
```

这段代码会报错，因为与父模块中的项对子模块可见相反，子模块中的项对父模块是不可见的。这里 `semisecret` 方法中，`a -> b -> c` 形成了父子模块链，那 `c` 中的 `J` 自然对 `a` 模块不可见。

如果使用之前的可见性方式，那么想保持 `J` 私有，同时让 `a` 继续使用 `semisecret` 函数的办法是将该函数移动到 `c` 模块中，然后用 `pub use` 将 `semisecret` 函数进行再导出：

```

pub mod a {
    pub const I: i32 = 3;

    use self::b::semisecret;

    pub fn bar(z: i32) -> i32 {
        semisecret(I) * z
    }
    pub fn foo(y: i32) -> i32 {
        semisecret(I) + y
    }

    mod b {
        pub use self::c::semisecret;
        mod c {
            const J: i32 = 4;
            pub fn semisecret(x: i32) -> i32 {
                x + J
            }
        }
    }
}

```

这段代码说实话问题不大，但是有些破坏了我们之前的逻辑，如果想保持代码逻辑，同时又只让 `J` 在 `a` 内可见该怎么办？

```

pub mod a {
    pub const I: i32 = 3;

    fn semisecret(x: i32) -> i32 {
        use self::b::c::J;
        x + J
    }

    pub fn bar(z: i32) -> i32 {
        semisecret(I) * z
    }
    pub fn foo(y: i32) -> i32 {
        semisecret(I) + y
    }

    mod b {
        pub(in crate::a) mod c {
            pub(in crate::a) const J: i32 = 4;
        }
    }
}

```

通过 `pub(in crate::a)` 的方式，我们指定了模块 `c` 和常量 `J` 的可见范围都只是 `a` 模块中，`a` 之外的模块是完全访问不到它们的。

## 限制可见性语法

`pub(crate)` 或 `pub(in crate:::a)` 就是限制可见性语法，前者是限制在整个包内可见，后者是通过绝对路径，限制在包内的某个模块内可见，总结一下：

- `pub` 意味着可见性无任何限制
- `pub(crate)` 表示在当前包可见
- `pub(self)` 在当前模块可见
- `pub(super)` 在父模块可见
- `pub(in <path>)` 表示在某个路径代表的模块中可见，其中 `path` 必须是父模块或者祖先模块

## 一个综合例子

```
// 一个名为 `my_mod` 的模块
mod my_mod {
    // 模块中的项默认具有私有的可见性
    fn private_function() {
        println!("called `my_mod::private_function()`");
    }

    // 使用 `pub` 修饰语来改变默认可见性。
    pub fn function() {
        println!("called `my_mod::function()`");
    }

    // 在同一模块中，项可以访问其它项，即使它是私有的。
    pub fn indirect_access() {
        print!("called `my_mod::indirect_access()`, that\n> ");
        private_function();
    }

    // 模块也可以嵌套
    pub mod nested {
        pub fn function() {
            println!("called `my_mod::nested::function()`");
        }

        #[allow(dead_code)]
        fn private_function() {
            println!("called `my_mod::nested::private_function()`");
        }
    }

    // 使用 `pub(in path)` 语法定义的函数只在给定的路径中可见。
    // `path` 必须是父模块 (parent module) 或祖先模块 (ancestor module)
    pub(in crate::my_mod) fn public_function_in_my_mod() {
        print!("called `my_mod::nested::public_function_in_my_mod()`, that\n> ");
        public_function_in_nested()
    }

    // 使用 `pub(self)` 语法定义的函数则只在当前模块中可见。
    pub(self) fn public_function_in_nested() {
        println!("called `my_mod::nested::public_function_in_nested()`");
    }

    // 使用 `pub(super)` 语法定义的函数只在父模块中可见。
    pub(super) fn public_function_in_super_mod() {
        println!("called my_mod::nested::public_function_in_super_mod");
    }
}

pub fn call_public_function_in_my_mod() {
```

```
    print!("called `my_mod::call_public_function_in_my_mod()` , that\n> ");
    nested::public_function_in_my_mod();
    print!("> ");
    nested::public_function_in_super_mod();
}

// `pub(crate)` 使得函数只在当前包中可见
pub(crate) fn public_function_in_crate() {
    println!("called `my_mod::public_function_in_crate()`");
}

// 嵌套模块的可见性遵循相同的规则
mod private_nested {
    #[allow(dead_code)]
    pub fn function() {
        println!("called `my_mod::private_nested::function()`");
    }
}

fn function() {
    println!("called `function()`");
}

fn main() {
    // 模块机制消除了相同名字的项之间的歧义。
    function();
    my_mod::function();

    // 公有项，包括嵌套模块内的，都可以在父模块外部访问。
    my_mod::indirect_access();
    my_mod::nested::function();
    my_mod::call_public_function_in_my_mod();

    // pub(crate) 项可以在同一个 crate 中的任何地方访问
    my_mod::public_function_in_crate();

    // pub(in path) 项只能在指定的模块中访问
    // 报错! 函数 `public_function_in_my_mod` 是私有的
    // my_mod::nested::public_function_in_my_mod();
    // 试一试 ^ 取消该行的注释

    // 模块的私有项不能直接访问，即便它是嵌套在公有模块内部的

    // 报错! `private_function` 是私有的
    // my_mod::private_function();
    // 试一试 ^ 取消此行注释

    // 报错! `private_function` 是私有的
    // my_mod::nested::private_function();
    // 试一试 ^ 取消此行的注释
```

```
// 报错! `private_nested` 是私有的
//my_mod::private_nested::function();
// 试一试 ^ 取消此行的注释
}
```

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 注释和文档

好的代码会说话，好的程序员不写注释，这些都是烂大街的“编程界俚语”。但是，如果你真的遇到一个不写注释的项目或程序员，那一定会对它/他“刮目相看”。

在之前的章节我们学习了包和模块如何使用，在此章节将进一步学习如何书写文档注释，以及如何使用 `cargo doc` 生成项目的文档，最后将以一个包、模块和文档的综合性例子，来将这些知识融会贯通。

## 注释的种类

在 Rust 中，注释分为三类：

- 代码注释，用于说明某一块代码的功能，读者往往是同一个项目的协作开发者
- 文档注释，支持 `Markdown`，对项目描述、公共 API 等用户关心的功能进行介绍，同时还能提供示例代码，目标读者往往是想要了解你项目的人
- 包和模块注释，严格来说这也是文档注释中的一种，它主要用于说明当前包和模块的功能，方便用户迅速了解一个项目

通过这些注释，实现了 Rust 极其优秀的文档化支持，甚至你还能在文档注释中写测试用例，省去了单独写测试用例的环节，我直呼好家伙！

## 代码注释

显然之前的刮目相看是打了引号的，想要去掉引号，该写注释的时候，就老老实实的，不过写时需要遵循八字原则：**围绕目标，言简意赅**，记住，洋洋洒洒那是用来形容文章的，不是形容注释！

代码注释方式有两种：

### 行注释 //

```
fn main() {
    // 我是Sun...
    // face
    let name = "sunface";
    let age = 18; // 今年好像是18岁
}
```

如上所示，行注释可以放在某一行代码的上方，也可以放在当前代码行的后方。如果超出一行的长度，需要在新行的开头也加上 `//`。

当注释行数较多时，你还可以使用**块注释**

### 块注释/\* ..... \*/

```
fn main() {
    /*
        我
        是
        S
        u
        n
        ...
        哎，好长！
    */
    let name = "sunface";
    let age = "???"; // 今年其实。。。挺大了
}
```

如上所示，只需要将注释内容使用 `/* */` 进行包裹即可。

你会发现，Rust 的代码注释跟其它语言并没有区别，主要区别其实在于文档注释这一块，也是本章节内容的重点。

## 文档注释

当查看一个 `crates.io` 上的包时，往往需要通过它提供的文档来浏览相关的功能特性、使用方式，这种文档就是通过文档注释实现的。

Rust 提供了 `cargo doc` 的命令，可以用于把这些文档注释转换成 `HTML` 网页文件，最终展示给用户浏览，这样用户就知道这个包是做什么的以及该如何使用。

### 文档行注释///

本书的一大特点就是废话不多，因此我们开门见山：

```
/// `add_one` 将指定值加1
///
/// # Examples
///
/// ``
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ``
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

以上代码有几点需要注意：

- 文档注释需要位于 `lib` 类型的包中，例如 `src/lib.rs` 中
- 文档注释可以使用 `markdown` 语法！例如 `# Examples` 的标题，以及代码块高亮
- 被注释的对象需要使用 `pub` 对外可见，记住：文档注释是给用户看的，**内部实现细节不应该被暴露出去**

咦？文档注释中的例子，为什看上去像是能运行的样子？竟然还是有 `assert_eq!` 这种常用于测试目的的宏。嗯，你的感觉没错，详细内容会在本章后面讲解，容我先卖个关子。

## 文档块注释 `/** ... */`

与代码注释一样，文档也有块注释，当注释内容多时，使用块注释可以减少 `///` 的使用：

```
/** `add_two` 将指定值加2

```
let arg = 5;
let answer = my_crate::add_two(arg);

assert_eq!(7, answer);
```
*/
pub fn add_two(x: i32) -> i32 {
    x + 2
}
```

## 查看文档 `cargo doc`

锦衣不夜行，这是中国人的传统美德。我们写了这么漂亮的文档注释，当然要看看网页中是什么效果咯。

很简单，运行 `cargo doc` 可以直接生成 HTML 文件，放入 `target/doc` 目录下。

当然，为了方便，我们使用 `cargo doc --open` 命令，可以在生成文档后，自动在浏览器中打开网页，最终效果如图所示：



非常棒，而且非常简单，这就是 Rust 工具链的强大之处。

## 常用文档标题

之前我们见到了在文档注释中该如何使用 `markdown`，其中包括 `# Examples` 标题。除了这个标题，还有一些常用的，你可以在项目中酌情使用：

- **Panics**: 函数可能会出现的异常状况，这样调用函数的人就可以提前规避
- **Errors**: 描述可能出现的错误及什么情况会导致错误，有助于调用者针对不同的错误采取不同的处理方式
- **Safety**: 如果函数使用 `unsafe` 代码，那么调用者就需要注意一些使用条件，以确保 `unsafe` 代码块的正常工作

话说回来，这些标题更多的是一种惯例，如果你非要用中文标题也没问题，但是最好在团队中保持同样的风格：)

## 包和模块级别的注释

除了函数、结构体等 Rust 项的注释，你还可以给包和模块添加注释，需要注意的是，**这些注释要添加到包、模块的最上方！**

与之前的任何注释一样，包级别的注释也分为两种：行注释 `//!` 和块注释 `/*! ... */`。

现在，为我们的包增加注释，在 `src/lib.rs` 包根的最上方，添加：

```
/*! lib包是world_hello二进制包的依赖包,  
里面包含了compute等有用模块 */  
  
pub mod compute;
```

然后再为该包根的子模块 `src/compute.rs` 添加注释：

```
///! 计算一些你口算算不出来的复杂算术题
```

```
/// `add_one`将指定值加1  
///
```

运行 `cargo doc --open` 查看下效果：



包模块注释，可以让用户从整体的角度理解包的用途，对于用户来说是非常友好的，就和一篇文章的开头一样，总是要对文章的内容进行大致的介绍，让用户在看的时候心中有数。

至此，关于如何注释的内容，就结束了，那么注释还能用来做什么？可以玩出花来吗？答案是 Yes .

## 文档测试(Doc Test)

相信读者之前都写过单元测试用例，其中一个很蛋疼的问题就是，随着代码的进化，单元测试用例经常会失效，过段时间后(为何是过段时间？应该这么问，有几个开发喜欢写测试用例 =\_=)，你发现需要连续修改不少处代码，才能让测试重新工作起来。然而，在 Rust 中，大可不必。

在之前的 `add_one` 中，我们写的示例代码非常像是一个单元测试的用例，这是偶然吗？并不是。因为 Rust 允许我们在文档注释中写单元测试用例！方法就如同之前做的：

```
/// `add_one` 将指定值加1
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = world_hello::compute::add_one(arg);
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

以上的注释不仅仅是文档，还可以作为单元测试的用例运行，使用 `cargo test` 运行测试：

```
Doc-tests world_hello

running 2 tests
test src/compute.rs - compute::add_one (line 8) ... ok
test src/compute.rs - compute::add_two (line 22) ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 1.00s
```

可以看到，文档中的测试用例被完美运行，而且输出中也明确提示了 `Doc-tests world_hello`，意味着这些测试的名字叫 `Doc test` 文档测试。

---

需要注意的是，你可能需要使用类如 `world_hello::compute::add_one(arg)` 的完整路径来调用函数，因为测试是在另外一个独立的线程中运行的

## 造成 panic 的文档测试

文档测试中的用例还可以造成 `panic`：

```
/// # Panics
///
/// The function panics if the second argument is zero.
///
/// ````rust
/// // panics on division by zero
/// world_hello::compute::div(10, 0);
/// ``
pub fn div(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    }

    a / b
}
```

以上测试运行后会 panic：

```
---- src/compute.rs - compute::div (line 38) stdout ----
Test executable failed (exit code 101).

stderr:
thread 'main' panicked at 'Divide-by-zero error', src/compute.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

如果想要通过这种测试，可以添加 should\_panic：

```
/// # Panics
///
/// The function panics if the second argument is zero.
///
/// ````rust,should_panic
/// // panics on division by zero
/// world_hello::compute::div(10, 0);
/// ````
```

通过 should\_panic，告诉 Rust 我们这个用例会导致 panic，这样测试用例就能顺利通过。

## 保留测试，隐藏文档

在某些时候，我们希望保留文档测试的功能，但是又要将某些测试用例的内容从文档中隐藏起来：

```
/// ``
/// # // 使用#开头的行会在文档中被隐藏起来，但是依然会在文档测试中运行
/// # fn try_main() -> Result<(), String> {
/// let res = world_hello::compute::try_div(10, 0)?;
/// # Ok(()) // returning from try_main
/// #
/// # fn main() {
///     try_main().unwrap();
/// #
/// #
/// #
/// ``
pub fn try_div(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divide-by-zero"))
    } else {
        Ok(a / b)
    }
}
```

以上文档注释中，我们使用 `#` 将不想让用户看到的内容隐藏起来，但是又不影响测试用例的运行，最终用户将只能看到那行没有隐藏的 `let res = world_hello::compute::try_div(10, 0)?;`：

The screenshot shows a dark-themed Rust documentation page. At the top, it says "Function world\_hello::compute::try\_div". Below that is the function signature: "pub fn try\_div(a: i32, b: i32) -> Result<i32, String>". A note below the signature says "[–] Using hidden try\_main in doc tests." followed by the code "let res = world\_hello::compute::try\_div(10, 0)?;".

```
pub fn try_div(a: i32, b: i32) -> Result<i32, String>

[–] Using hidden try_main in doc tests.

let res = world_hello::compute::try_div(10, 0)?;
```

## 文档注释中的代码跳转

Rust 在文档注释中还提供了一个非常强大的功能，那就是可以实现对外部项的链接：

## 跳转到标准库

```
/// `add_one` 返回一个[`Option`]类型
pub fn add_one(x: i32) -> Option<i32> {
    Some(x + 1)
}
```

此处的 [Option] 就是一个链接，指向了标准库中的 Option 枚举类型，有两种方式可以进行跳转：

- 在 IDE 中，使用 Command + 鼠标左键 (macOS)， CTRL + 鼠标左键 (Windows)
- 在文档中直接点击链接

再比如，还可以使用路径的方式跳转：

```
use std::sync::mpsc::Receiver;

/// [`Receiver<T>`]   [`std::future`].
///
/// [`std::future::Future`] [`Self::recv()`].
pub struct AsyncReceiver<T> {
    sender: Receiver<T>,
}

impl<T> AsyncReceiver<T> {
    pub async fn recv() -> T {
        unimplemented!()
    }
}
```

## 使用完整路径跳转到指定项

除了跳转到标准库，你还可以通过指定具体的路径跳转到自己代码或者其它库的指定项，例如在 lib.rs 中添加以下代码：

```
pub mod a {
    /// `add_one` 返回一个[`Option`]类型
    /// 跳转到[`crate::MySpecialFormatter`]
    pub fn add_one(x: i32) -> Option<i32> {
        Some(x + 1)
    }
}

pub struct MySpecialFormatter;
```

使用 crate::MySpecialFormatter 这种路径就可以实现跳转到 lib.rs 中定义的结构体上。

## 同名项的跳转

如果遇到同名项，可以使用标示类型的方式进行跳转：

```
/// 跳转到结构体 [`Foo`](struct@Foo)
pub struct Bar;

/// 跳转到同名函数 [`Foo`](fn@Foo)
pub struct Foo {}

/// 跳转到同名宏 [`foo!`]
pub fn Foo() {}

#[macro_export]
macro_rules! foo {
    () => {}
}
```

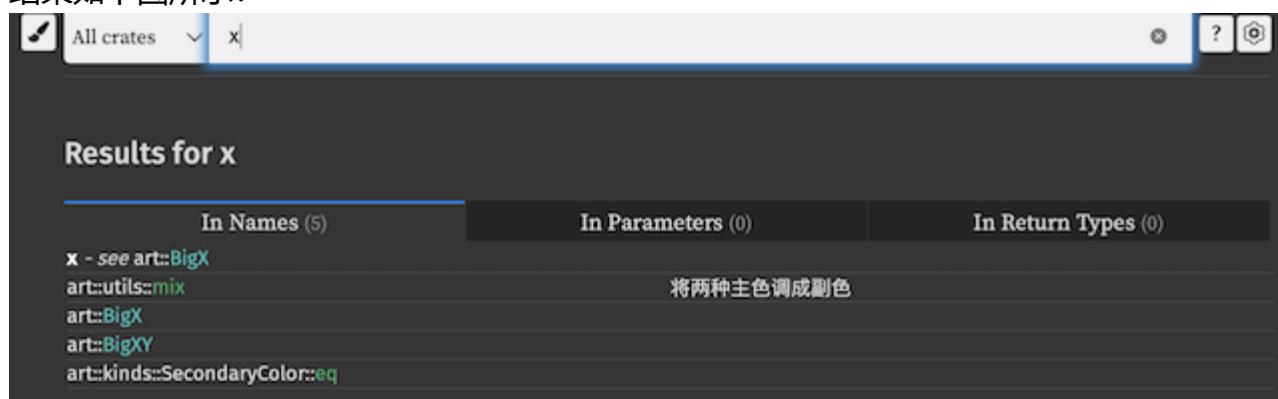
## 文档搜索别名

Rust 文档支持搜索功能，我们可以为自己的类型定义几个别名，以实现更好的搜索展现，当别名命中时，搜索结果会被放在第一位：

```
#[doc(alias = "x")]
#[doc(alias = "big")]
pub struct BigX;

#[doc(alias("y", "big"))]
pub struct BigY;
```

结果如下图所示：



The screenshot shows the Rust documentation search interface. The search bar at the top contains the letter 'x'. Below the search bar, the title 'Results for x' is displayed. There are three tabs: 'In Names (5)', 'In Parameters (0)', and 'In Return Types (0)'. The 'In Names (5)' tab is selected. Under this tab, there is a list of items:

- x - see art::BigX
- art::utils::mix
- art::BigX
- art::BigXY
- art::kinds::SecondaryColor::eq

Next to the list, there is a note: '将两种主色调成副色'.

## 一个综合例子

这个例子我们将重点应用几个知识点：

- 文档注释
- 一个项目可以包含两个包：二进制可执行包和 `lib` 包（库包），它们的包根分别是 `src/main.rs` 和 `src/lib.rs`
- 在二进制包中引用 `lib` 包
- 使用 `pub use` 再导出 API，并观察文档

首先，使用 `cargo new art` 创建一个 Package `art`：

```
Created binary (application) `art` package
```

系统提示我们创建了一个二进制 Package，根据[之前章节](#)学过的知识，可以知道该 Package 包含一个同名的二进制包：包名为 `art`，包根为 `src/main.rs`，该包可以编译成二进制然后运行。

现在，在 `src` 目录下创建一个 `lib.rs` 文件，同样，根据之前学习的知识，创建该文件等于又创建了一个库类型的包，包名也是 `art`，包根为 `src/lib.rs`，该包是库类型的，因此往往作为依赖库被引入。

将以下内容添加到 `src/lib.rs` 中：

```

//! # Art
//!
//! 未来的艺术建模库，现在的调色库

pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;

pub mod kinds {
    //! 定义颜色的类型

    /// 主色
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// 副色
    #[derive(Debug, PartialEq)]
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    //! 实用工具，目前只实现了调色板
    use crate::kinds::*;

    /// 将两种主色调成副色
    /// ````rust
    /// use art::utils::mix;
    /// use art::kinds::{PrimaryColor, SecondaryColor};
    /// assert!(matches!(mix(PrimaryColor::Yellow, PrimaryColor::Blue),
    SecondaryColor::Green));
    /// ````

    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        SecondaryColor::Green
    }
}

```

在库包的包根 `src/lib.rs` 下，我们又定义了几个子模块，同时将子模块中的三个项通过 `pub use` 进行了再导出。

接着，将下面内容添加到 `src/main.rs` 中：

```
use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let blue = PrimaryColor::Blue;
    let yellow = PrimaryColor::Yellow;
    println!("{:?}", mix(blue, yellow));
}
```

在二进制可执行包的包根 `src/main.rs` 下，我们引入了库包 `art` 中的模块项，同时使用 `main` 函数作为程序的入口，该二进制包可以使用 `cargo run` 运行：

Green

至此，库包完美提供了用于调色的 API，二进制包引入这些 API 完美的实现了调色并打印输出。

最后，再来看看文档长啥样：



## 总结

在 Rust 中，注释分为三个主要类型：代码注释、文档注释、包和模块注释，每个注释类型都拥有两种形式：行注释和块注释，熟练掌握包模块和注释的知识，非常有助于我们创建工程性更强的项目。

如果读者看到这里对于包模块还是有些模糊，强烈建议回头看看相关的章节以及本章节的最后一个综合例子。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。（本节暂无习题解答）

# 格式化输出

提到格式化输出，可能很多人立刻就想到 "{}"，但是 Rust 能做到的远比这个多的多，本章节我们将深入讲解格式化输出的各个方面。

## 满分初印象

先来一段代码，看看格式化输出的初印象：

```
println!("Hello");           // => "Hello"
println!("Hello, {}!", "world"); // => "Hello, world!"
println!("The number is {}", 1); // => "The number is 1"
println!("{}:{}",(3, 4));      // => "(3, 4)"
println!("{}{}", value=4);    // => "4"
println!("{} {}", 1, 2);       // => "1 2"
println!("{:04}", 42);         // => "0042" with leading zeros
```

可以看到 `println!` 宏接受的是可变参数，第一个参数是一个字符串常量，它表示最终输出字符串的格式，包含其中形如 {} 的符号是占位符，会被 `println!` 后面的参数依次替换。

## print!, println!, format!

它们是 Rust 中用来格式化输出的三大金刚，用途如下：

- `print!` 将格式化文本输出到标准输出，不带换行符
- `println!` 同上，但是在行的末尾添加换行符
- `format!` 将格式化文本输出到 `String` 字符串

在实际项目中，最常用的是 `println!` 及 `format!`，前者常用来调试输出，后者常用来生成格式化的字符串：

```
fn main() {
    let s = "hello";
    println!("{} world", s);
    let s1 = format!("{} world", s);
    print!("{} ", s1);
    print!("{}\n", "!");
}
```

其中，`s1` 是通过 `format!` 生成的 `String` 字符串，最终输出如下：

```
hello, world  
hello, world!
```

### eprint!, eprintln!

除了三大金刚外，还有两大护法，使用方式跟 `print!`, `println!` 很像，但是它们输出到标准错误输出：

```
eprintln!("Error: Could not complete task")
```

它们仅应该被用于输出错误信息和进度信息，其它场景都应该使用 `print!` 系列。

## {} 与

与其它语言常用的 `%d`, `%s` 不同，Rust 特立独行地选择了 `{}` 作为格式化占位符（说到这个，有点想吐槽下，Rust 中自创的概念其实还挺多的，真不知道该夸奖还是该吐槽-.-），事实证明，这种选择非常正确，它帮助用户减少了很多使用成本，你无需再为特定的类型选择特定的占位符，统一用 `{}` 来替代即可，剩下的类型推导等细节只要交给 Rust 去做。

与 `{}` 类似，`{:?}` 也是占位符：

- `{}` 适用于实现了 `std::fmt::Display` 特征的类型，用来以更优雅、更友好的方式格式化文本，例如展示给用户
- `{:?}` 适用于实现了 `std::fmt::Debug` 特征的类型，用于调试场景

其实两者的选择很简单，当你在写代码需要调试时，使用 `{:?}`，剩下的场景，选择 `{}`。

### Debug 特征

事实上，为了方便我们调试，大多数 Rust 类型都实现了 `Debug` 特征或者支持派生该特征：

```

#[derive(Debug)]
struct Person {
    name: String,
    age: u8
}

fn main() {
    let i = 3.1415926;
    let s = String::from("hello");
    let v = vec![1, 2, 3];
    let p = Person{name: "sunface".to_string(), age: 18};
    println!("{:?}, {:?}, {:?}, {:?}", i, s, v, p);
}

```

对于数值、字符串、数组，可以直接使用 `{:?}",` 进行输出，但是对于结构体，需要[派生 Debug 特征](#)后，才能进行输出，总之很简单。

## Display 特征

与大部分类型实现了 `Debug` 不同，实现了 `Display` 特征的 Rust 类型并没有那么多，往往需要我们自定义想要的格式化方式：

```

let i = 3.1415926;
let s = String::from("hello");
let v = vec![1, 2, 3];
let p = Person {
    name: "sunface".to_string(),
    age: 18,
};
println!("{}, {}, {}, {}", i, s, v, p);

```

运行后可以看到 `v` 和 `p` 都无法通过编译，因为没有实现 `Display` 特征，但是你又不能像派生 `Debug` 一般派生 `Display`，只能另寻他法：

- 使用 `{:?}",` 或 `{:#?}`
- 为自定义类型实现 `Display` 特征
- 使用 `newtype` 为外部类型实现 `Display` 特征

下面来——看看这三种方式。

`{:#?}` 与 `{:?}",` 几乎一样，唯一的区别在于它能更优美地输出内容：

```
// {::?}
[1, 2, 3], Person { name: "sunface", age: 18 }

// {:#?}
[
    1,
    2,
    3,
], Person {
    name: "sunface",
}
```

因此对于 `Display` 不支持的类型，可以考虑使用 `{:#?}` 进行格式化，虽然理论上它更适合进行调试输出。

## 为自定义类型实现 `Display` 特征

如果你的类型是定义在当前作用域中的，那么可以为其实现 `Display` 特征，即可用于格式化输出：

```
struct Person {
    name: String,
    age: u8,
}

use std::fmt;
impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f,
               "大佬在上，请受我一拜，小弟姓名{}, 年芳{}, 家里无田又无车，生活苦哈哈",
               self.name, self.age
        )
    }
}
fn main() {
    let p = Person {
        name: "sunface".to_string(),
        age: 18,
    };
    println!("{}", p);
}
```

如上所示，只要实现 `Display` 特征中的 `fmt` 方法，即可为自定义结构体 `Person` 添加自定义输出：

大佬在上，请受我一拜，小弟姓名sunface，年芳18，家里无田又无车，生活苦哈哈

## 为外部类型实现 Display 特征

在 Rust 中，无法直接为外部类型实现外部特征，但是可以使用 `newtype` 解决此问题：

```
struct Array(Vec<i32>);

use std::fmt;
impl fmt::Display for Array {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "数组是: {:?}", self.0)
    }
}
fn main() {
    let arr = Array(vec![1, 2, 3]);
    println!("{}", arr);
}
```

`Array` 就是我们的 `newtype`，它将想要格式化输出的 `Vec` 包裹在内，最后只要为 `Array` 实现 `Display` 特征，即可进行格式化输出：

```
数组是: [1, 2, 3]
```

至此，关于 `{}` 与 `{:?}` 的内容已介绍完毕，下面让我们正式开始格式化输出的旅程。

## 位置参数

除了按照依次顺序使用值去替换占位符之外，还能让指定位置的参数去替换某个占位符，例如 `{1}`，表示用第二个参数替换该占位符(索引从 0 开始)：

```
fn main() {
    println!("{}{}", 1, 2); // =>"12"
    println!("{}{}", 1, 2); // =>"21"
    // => Alice, this is Bob. Bob, this is Alice
    println!("{}{}, this is {}{}. {}is {}", "Alice", "Bob");
    println!("{}{}{}{}{}", 1, 2); // => 2112
}
```

## 具名参数

除了像上面那样指定位置外，我们还可以为参数指定名称：

```
fn main() {
    println!("{}argument", argument = "test"); // => "test"
    println!("{}name {}", 1, name = 2); // => "2 1"
    println!("{}a {} b", a = "a", b = 'b', c = 3); // => "a 3 b"
}
```

需要注意的是：带名称的参数必须放在不带名称参数的后面，例如下面代码将报错：

```
println!("{} {}", abc = "def", 2);

error: positional arguments cannot follow named arguments
--> src/main.rs:4:36
  |
4 |     println!("{} {}", abc = "def", 2);
  |                         ^ positional arguments must be before named
arguments
  |                         |
  |                         named argument
```

## 格式化参数

格式化输出，意味着对输出格式会有更多的要求，例如只输出浮点数的小数点后两位：

```
fn main() {
    let v = 3.1415926;
    // Display => 3.14
    println!(" {:.2}", v);
    // Debug => 3.14
    println!(" {:.2?}", v);
}
```

上面代码只输出小数点后两位。同时我们还展示了 {} 和 {:?} 的用法，后面如无特殊区别，就只针对 {} 提供格式化参数说明。

接下来，让我们一起来看看 Rust 中有哪些格式化参数。

### 宽度

宽度用来指示输出目标的长度，如果长度不够，则进行填充和对齐：

## 字符串填充

字符串格式化默认使用空格进行填充，并且进行左对齐。

```
fn main() {
    //-----
    // 以下全部输出 "Hello x      !"
    // 为"x"后面填充空格，补齐宽度5
    println!("Hello {:5}!", "x");
    // 使用参数5来指定宽度
    println!("Hello {:1$}!", "x", 5);
    // 使用x作为占位符输出内容，同时使用5作为宽度
    println!("Hello {1:0$}!", 5, "x");
    // 使用有名称的参数作为宽度
    println!("Hello {:width$}!", "x", width = 5);
    //-----

    // 使用参数5为参数x指定宽度，同时在结尾输出参数5 => Hello x      !5
    println!("Hello {:1$}!{}", "x", 5);
}
```

## 数字填充:符号和 0

数字格式化默认也是使用空格进行填充，但与字符串左对齐不同的是，数字是右对齐。

```
fn main() {
    // 宽度是5 => Hello      5!
    println!("Hello {:5}!", 5);
    // 显式的输出正号 => Hello +5!
    println!("Hello {:+}!", 5);
    // 宽度5，使用0进行填充 => Hello 00005!
    println!("Hello {:05}!", 5);
    // 负号也要占用一位宽度 => Hello -0005!
    println!("Hello {:05}!", -5);
}
```

## 对齐

```
fn main() {
    // 以下全部都会补齐5个字符的长度
    // 左对齐 => Hello x !
    println!("Hello {:<5}!", "x");
    // 右对齐 => Hello      x!
    println!("Hello {:>5}!", "x");
    // 居中对齐 => Hello   x !
    println!("Hello {:^5}!", "x");

    // 对齐并使用指定符号填充 => Hello x&&&&!
    // 指定符号填充的前提条件是必须有对齐字符
    println!("Hello {:&<5}!", "x");
}
```

## 精度

精度可以用于控制浮点数的精度或者字符串的长度

```
fn main() {
    let v = 3.1415926;
    // 保留小数点后两位 => 3.14
    println!("{:.2}", v);
    // 带符号保留小数点后两位 => +3.14
    println!("{:+.2}", v);
    // 不带小数 => 3
    println!("{:.0}", v);
    // 通过参数来设定精度 => 3.1416, 相当于{:.4}
    println!("{:.1$}", v, 4);

    let s = "hi我是Sunface孙飞";
    // 保留字符串前三个字符 => hi我
    println!("{:.3}", s);
    // {:.*}接收两个参数, 第一个是精度, 第二个是被格式化的值 => Hello abc!
    println!("Hello {:.*}!", 3, "abcdefg");
}
```

## 进制

可以使用 # 号来控制数字的进制输出:

- #b, 二进制
- #o, 八进制

- `#x` , 小写十六进制
- `#X` , 大写十六进制
- `x` , 不带前缀的小写十六进制

```
fn main() {
    // 二进制 => 0b11011!
    println!("{:#b}!", 27);
    // 八进制 => 0o33!
    println!("{:#o}!", 27);
    // 十进制 => 27!
    println!("{}!", 27);
    // 小写十六进制 => 0x1b!
    println!("{:#x}!", 27);
    // 大写十六进制 => 0x1B!
    println!("{:#X}!", 27);

    // 不带前缀的十六进制 => 1b!
    println!("{:x}!", 27);

    // 使用0填充二进制, 宽度为10 => 0b00011011!
    println!("{:#010b}!", 27);
}
```

## 指数

```
fn main() {
    println!("{:2e}", 1000000000); // => 1e9
    println!("{:2E}", 1000000000); // => 1E9
}
```

## 指针地址

```
let v = vec![1, 2, 3];
println!("{:p}", v.as_ptr()) // => 0x600002324050
```

## 转义

有时需要输出 { 和 } , 但这两个字符是特殊字符, 需要进行转义:

```

fn main() {
    // "{}" 转义为 '{'   "}" 转义为 '}'     "\\" 转义为 ''
    // => Hello "{World}"
    println!(" Hello \"{{World}}\" ");

    // 下面代码会报错，因为占位符{}只有一个右括号}，左括号被转义成字符串的内容
    // println!(" {{ Hello }} ");
    // 也不可使用 '\' 来转义 "{}"
    // println!(" \{ Hello \} ");
}

}

```

## 在格式化字符串时捕获环境中的值 (Rust 1.58 新增)

在以前，想要输出一个函数的返回值，你需要这么做：

```

fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let p = get_person();
    println!("Hello, {}!", p);                      // implicit position
    println!("Hello, {0}!", p);                      // explicit index
    println!("Hello, {person}!", person = p);
}

```

问题倒也不大，但是一旦格式化字符串长了后，就会非常冗余，而在 1.58 后，我们可以这么写：

```

fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let person = get_person();
    println!("Hello, {person}!");
}

```

是不是清晰、简洁了很多？甚至还可以将环境中的值用于格式化参数：

```

let (width, precision) = get_format();
for (name, score) in get_scores() {
    println!("{}: {:.width$p.{precision$}}");
}

```

但也有局限，它只能捕获普通的变量，对于更复杂的类型（例如表达式），可以先将它赋值给一个变量或使用以前的 `name = expression` 形式的格式化参数。目前除了 `panic!` 外，其它接收格式化参数的

宏，都可以使用新的特性。对于 `panic!` 而言，如果还在使用 2015 版本 或 2018 版本，那 `panic!("{}")` 依然会被当成正常的字符串来处理，同时编译器会给予 `warn` 提示。而对于 2021 版本，则可以正常使用：

```
fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let person = get_person();
    panic!("Hello, {person}!");
}
```

输出：

```
thread 'main' panicked at 'Hello, sunface!', src/main.rs:6:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

## 课后练习

---

[Rust By Practice](#)，支持代码在线编辑和运行，并提供详细的习题解答。（本节暂无习题解答）

---

## 总结

把这些格式化都牢记在脑中是不太现实的，也没必要，我们要做的就是知道 Rust 支持相应的格式化输出，在需要之时，读者再来查阅本文即可。

还是那句话，[`<<Rust 语言圣经>>`](#)不仅仅是 Rust 学习书籍，还是一本厚重的工具书！

至此，Rust 的基础内容学习已经全部完成，下面我们将学习 Rust 的高级进阶内容，正式开启你的高手之路。

# 构建一个简单命令行程序

在前往更高的山峰前，我们应该驻足欣赏下身后的风景，虽然是半览众山不咋小，但总比身在此山中无法窥全貌要强一丢丢。

在本章中，我们将一起构建一个命令行程序，目标是尽可能帮大家融会贯通之前的学到的知识。

linux 系统中的 `grep` 命令很强大，可以完成各种文件搜索任务，我们肯定做不了那么强大，但是假冒一个伪劣的版本还是可以的，它将从命令行参数中读取指定的文件名和字符串，然后在相应的文件中找到包含该字符串的内容，最终打印出来。

---

这里推荐一位大神写的知名 Rust 项目 [ripgrep](#)，绝对是 `grep` 真正的高替品，值得学习和使用

---

# 实现基本功能

无论功能设计的再怎么花里胡哨，对于一个文件查找命令而言，首先得指定文件和待查找的字符串，它们需要用户从命令行给予输入，然后我们在程序内进行读取。

## 接收命令行参数

国际惯例，先创建一个新的项目 `minigrep`，该名字充分体现了我们的自信：就是不如 `grep`。

```
cargo new minigrep
    Created binary (application) `minigrep` project
$ cd minigrep
```

首先来思考下，如果要传入文件路径和待搜索的字符串，那这个命令该长啥样，我觉得大概率是这样：

```
cargo run -- searchstring example-filename.txt
```

-- 告诉 cargo 后面的参数是给我们的程序使用的，而不是给 cargo 自己使用，例如 -- 前的 run 就是给它用的。

接下来就是在程序中读取传入的参数，这个很简单，下面代码就可以：

```
// in main.rs
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

首先通过 `use` 引入标准库中的 `env` 包，然后 `env::args` 方法会读取并分析传入的命令行参数，最终通过 `collect` 方法输出一个集合类型 `Vector`。

可能有同学疑惑，为啥不直接引入 `args`，例如 `use std::env::args`，这样就无需 `env::args` 来繁琐调用，直接 `args.collect()` 即可。原因很简单，`args` 方法只会使用一次，啰嗦就啰嗦点吧，把相同的好名字让给 `let args..` 这位大哥不好吗？毕竟人家要出场多次的。

---

## 不可信的输入

所有的用户输入都不可信！不可信！不可信！

重要的话说三遍，我们的命令行程序也是，用户会输入什么你根本都不知道，例如他输入了一个非 Unicode 字符，你能阻止吗？显然不能，但是这种输入会直接让我们的程序崩溃！

原因是当传入的命令行参数包含非 Unicode 字符时，`std::env::args` 会直接崩溃，如果有这种特殊需求，建议大家使用 `std::env::args_os`，该方法产生的数组将包含 `OsString` 类型，而不是之前的 `String` 类型，前者对于非 Unicode 字符会有更好的处理。

至于为啥我们不用，两个理由，你信哪个：1. 用户爱输入啥输入啥，反正崩溃了，他就知道自己错了 2. `args_os` 会引入额外的跨平台复杂性

---

`collect` 方法其实并不是 `std::env` 包提供的，而是迭代器自带的方法(`env::args()` 会返回一个迭代器)，它会将迭代器消费后转换成我们想要的集合类型，关于迭代器和 `collect` 的具体介绍，请参考[这里](#)。

最后，代码中使用 `dbg!` 宏来输出读取到的数组内容，来看看长啥样：

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.61s
    Running `target/debug/minigrep`
[src/main.rs:5] args = [
    "target/debug/minigrep",
]

$ cargo run -- needle haystack
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 1.57s
    Running `target/debug/minigrep needle haystack`
[src/main.rs:5] args = [
    "target/debug/minigrep",
    "needle",
    "haystack",
]
```

上面两个版本分别是无参数和两个参数，其中无参数版本实际上也会读取到一个字符串，仔细看，是不是长得像我们的程序名，Bingo! `env::args` 读取到的参数中第一个就是程序的可执行路径名。

## 存储读取到的参数

在编程中，给予清晰合理的变量名是一项基本功，咱总不能到处都是 `args[1]`、`args[2]` 这样的糟糕代码吧。

因此我们需要两个变量来存储文件路径和待搜索的字符串：

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

很简单的代码，来运行下：

```
$ cargo run -- test sample.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

输出结果很清晰的说明了我们的目标：在文件 `sample.txt` 中搜索包含 `test` 字符串的内容。

事实上，就算作为一个简单的程序，它也太过于简单了，例如用户不提供任何参数怎么办？因此，错误处理显然是不可少的，但是在添加之前，先来看看如何读取文件内容。

## 文件读取

既然读取文件，那么首先我们需要创建一个文件并给予一些内容，来首诗歌如何？"我啥也不是，你呢?"

I'm nobody! Who are you?  
我啥也不是，你呢?  
Are you nobody, too?  
牛逼如你也是无名之辈吗?  
Then there's a pair of us - don't tell!  
那我们就是天生一对，嘘！别说话！  
They'd banish us, you know.  
你知道，我们不属于这里。  
How dreary to be somebody!  
因为这里属于没劲的大人物！  
How public, like a frog  
他们就像青蛙一样呱噪，  
To tell your name the livelong day  
成天将自己的大名  
To an admiring bog!  
传遍整个无聊的沼泽！

在项目根目录创建 `poem.txt` 文件，并写入如上的优美诗歌(可能翻译的很烂，别打我，哈哈，事实上大家写入英文内容就够了)。

接下来修改 `main.rs` 来读取文件内容：

```
use std::env;
use std::fs;

fn main() {
    // --省略之前的内容--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{}", contents);
}
```

首先，通过 `use std::fs` 引入文件操作包，然后通过 `fs::read_to_string` 读取指定的文件内容，最后返回的 `contents` 是 `std::io::Result<String>` 类型。

运行下试试，这里无需输入第二个参数，因为我们还没有实现查询功能：

```
$ cargo run -- the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

完美，虽然代码还有很多瑕疵，例如所有内容都在 `main` 函数，这个不符合软件工程，没有错误处理，功能不完善等。不过没关系，万事开头难，好歹我们成功迈开了第一步。

好了，是时候重构赚波 KPI 了，读者：are you serious? 这就开始重构了？

# 增加模块化和错误处理

但凡稍微没那么糟糕的程序，都应该具有代码模块化和错误处理，不然连玩具都谈不上。

梳理我们的代码和目标后，可以整理出大致四个改进点：

- **单一且庞大的函数。**对于 `minigrep` 程序而言，`main` 函数当前执行两个任务：解析命令行参数和读取文件。但随着代码的增加，`main` 函数承载的功能也将快速增加。从软件工程角度来看，一个函数具有的功能越多，越是难以阅读和维护。因此最好的办法是将大的函数拆分成更小的功能单元。
- **配置变量散乱在各处。**还有一点要考虑的是，当前 `main` 函数中的变量都是独立存在的，这些变量很可能被整个程序所访问，在这个背景下，独立的变量越多，越是难以维护，因此我们还可以将这些用于配置的变量整合到一个结构体中。
- **细化错误提示。**目前的实现中，我们使用 `expect` 方法来输出文件读取失败时的错误信息，这个没问题，但是无论任何情况下，都只输出 `Should have been able to read the file` 这条错误提示信息，显然是有问题的，毕竟文件不存在、无权限等等都是可能的错误，一条大一统的消息无法给予用户更多的提示。
- **使用错误而不是异常。**假如用户不给任何命令行参数，那我们的程序显然会无情崩溃，原因很简单：`index out of bounds`，一个数组访问越界的 `panic`，但问题来了，用户能看懂吗？甚至于未来接收的维护者能看懂吗？因此需要增加合适的错误处理代码，来给予使用者给详细友善的提示。还有就是需要在一个统一的位置来处理所有错误，利人利己！

## 分离 `main` 函数

关于如何处理庞大的 `main` 函数，Rust 社区给出了统一的指导方案：

- 将程序分割为 `main.rs` 和 `lib.rs`，并将程序的逻辑代码移动到后者内
- 命令行解析属于非常基础的功能，严格来说不算是逻辑代码的一部分，因此还可以放在 `main.rs` 中

按照这个方案，将我们的代码重新梳理后，可以得出 `main` 函数应该包含的功能：

- 解析命令行参数
- 初始化其它配置
- 调用 `lib.rs` 中的 `run` 函数，以启动逻辑代码的运行
- 如果 `run` 返回一个错误，需要对该错误进行处理

这个方案有一个很优雅的名字：关注点分离(Separation of Concerns)。简而言之，`main.rs` 负责启动程序，`lib.rs` 负责逻辑代码的运行。从测试的角度而言，这种分离也非常合理：`lib.rs` 中的主体逻辑

代码可以得到简单且充分的测试，至于 `main.rs`？确实没办法针对其编写额外的测试代码，但是它的代码也很少啊，很容易就能保证它的正确性。

---

关于如何在 Rust 中编写测试代码，请参见如下章节：<https://course.rs/test/intro.html>

---

## 分离命令行解析

根据之前的分析，我们需要将命令行解析的代码分离到一个单独的函数，然后将该函数放置在 `main.rs` 中：

```
// in main.rs
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --省略--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];

    (query, file_path)
}
```

经过分离后，之前的设计目标完美达成，即精简了 `main` 函数，又将配置相关的代码放在了 `main.rs` 文件里。

看起来貌似是杀鸡用了牛刀，但是重构就是这样，一步一步，踏踏实实的前行，否则未来代码多一些后，你岂不是还要再重来一次重构？因此打好项目的基础是非常重要的！

## 聚合配置变量

前文提到，配置变量并不适合分散的到处都是，因此使用一个结构体来统一存放是非常好的选择，这样修改后，后续的使用以及未来的代码维护都将更加简单明了。

```

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    // --snip--
}

struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

    Config { query, file_path }
}

```

值得注意的是，`Config` 中存储的并不是 `&str` 这样的引用类型，而是一个 `String` 字符串，也就是 `Config` 并没有去借用外部的字符串，而是拥有内部字符串的所有权。`clone` 方法的使用也可以佐证这一点。大家可以尝试不用 `clone` 方法，看看该如何解决相关的报错 :D

### clone 的得与失

在上面的代码中，除了使用 `clone`，还有其它办法来达成同样的目的，但 `clone` 无疑是最简单的方法：直接完整的复制目标数据，无需被所有权、借用等问题所困扰，但是它也有其缺点，那就是有一定的性能损耗。

因此是否使用 `clone` 更多是一种性能上的权衡，对于上面的使用而言，由于是配置的初始化，因此整个程序只需要执行一次，性能损耗几乎是忽略不计的。

总之，判断是否使用 `clone`：

- 是否严肃的项目，玩具项目直接用 `clone` 就行，简单不好吗？
- 要看所在的代码路径是否是热点路径(hot path)，例如执行次数较多的显然就是热点路径，热点路径就值得去使用性能更好的实现方式

好了，言归正传，从 c 语言过来的同学可能会觉得上面的代码已经很棒了，但是从 OO 语言角度来说，还差了那么一点意思。

下面我们试着来优化下，通过构造函数来初始化一个 `Config` 实例，而不是直接通过函数返回实例，典型的，标准库中的 `String::new` 函数就是一个范例。

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

修改后，类似 `String::new` 的调用，我们可以通过 `Config::new` 来创建一个实例，看起来代码是不是更有那味儿了：）

## 错误处理

回顾一下，如果用户不输入任何命令行参数，我们的程序会怎么样？

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1',
src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

结果喜闻乐见，由于 `args` 数组没有任何元素，因此通过索引访问时，会直接报出数组访问越界的 `panic`。

报错信息对于开发者会很明确，但是对于使用者而言，就相当难理解了，下面一起来解决它。

## 改进报错信息

还记得在错误处理章节，我们提到过 `panic` 的两种用法：被动触发和主动调用嘛？上面代码的出现方式很明显是被动触发，这种报错信息是不可控的，下面我们先改成主动调用的方式：

```
// in main.rs
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
// --snip--
```

目的很明确，一旦传入的参数数组长度小于 3，则报错并让程序崩溃推出，这样后续的数组访问就不会再越界了。

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

不错，用户看到了更为明确的提示，但是还是有一大堆 `debug` 输出，这些我们其实是不想让用户看到的。这么看来，想要输出对用户友好的信息，`panic` 是不太适合的，它更适合告知开发者，哪里出现了问题。

## 返回 `Result` 来替代直接 `panic`

那只能祭出之前学过的错误处理大法了，也就是返回一个 `Result`：成功时包含 `Config` 实例，失败时包含一条错误信息。

有一点需要额外注意下，从代码惯例的角度出发，`new` 往往不会失败，毕竟新建一个实例没道理失败，对不？因此修改为 `build` 会更加合适。

```
impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

这里的 `Result` 可能包含一个 `Config` 实例，也可能包含一条错误信息 `&'static str`，不熟悉这种字符串类型的同学可以回头看看字符串章节，代码中的字符串字面量都是该类型，且拥有 `'static` 生命周期。

## 处理返回的 `Result`

接下来就是在调用 `build` 函数时，对返回的 `Result` 进行处理了，目的就是给出准确且友好的报错提示，为了让大家更好的回顾我们修改过的内容，这里给出整体代码：

```

use std::env;
use std::fs;
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    // 对 build 返回的 `Result` 进行处理
    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{}", contents);
}

struct Config {
    query: String,
    file_path: String,
}

impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}

```

上面代码有几点值得注意:

- 当 `Result` 包含错误时, 我们不再调用 `panic` 让程序崩溃, 而是通过 `process::exit(1)` 来终结进程, 其中 `1` 是一个信号值(事实上非 `0` 值都可以), 通知调用我们程序的进程, 程序是因为错误而退出的。
- `unwrap_or_else` 是定义在 `Result<T,E>` 上的常用方法, 如果 `Result` 是 `Ok`, 那该方法就类似 `unwrap`: 返回 `Ok` 内部的值; 如果是 `Err`, 就调用闭包中的自定义代码对错误进行进一步处理

综上可知，`config` 变量的值是一个 `Config` 实例，而 `unwrap_or_else` 闭包中的 `err` 参数，它的类型是 `'static str`，值是 "not enough arguments" 那个字符串字面量。

运行后，可以看到以下输出：

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.48s
Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

终于，我们得到了自己想要的输出：既告知了用户为何报错，又消除了多余的 debug 信息，非常棒。可能有用户疑惑，`cargo run` 底下还有一大堆 debug 信息呢，实际上，这是 `cargo run` 自带的，大家可以试试编译成二进制可执行文件后再调用，会是什么效果。

## 分离主体逻辑

接下来可以继续精简 `main` 函数，那就是将主体逻辑(例如业务逻辑)从 `main` 中分离出去，这样 `main` 函数就保留主流程调用，非常简洁。

```
// in main.rs
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{}", contents);
}

// --snip--
```

如上所示，`main` 函数仅保留主流程各个环节的调用，一眼看过去非常简洁清晰。

继续之前，先请大家仔细看看 `run` 函数，你们觉得还缺少什么？提示：参考 `build` 函数的改进过程。

## 使用 `? 和特征对象来返回错误`

答案就是 `run` 函数没有错误处理，因为在文章开头我们提到过，错误处理最好统一在一个地方完成，这样极其有利于后续的代码维护。

```
//in main.rs
use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;
    println!("With text:\n{contents}");

    Ok(())
}
```

值得注意的是这里的 `Result<(), Box<dyn Error>>` 返回类型，首先我们的程序无需返回任何值，但是为了满足 `Result<T,E>` 的要求，因此使用了 `ok()` 返回一个单元类型 `()`。

最重要的是 `Box<dyn Error>`，如果按照顺序学到这里，大家应该知道这是一个 `Error` 的特征对象(为了使用 `Error`，我们通过 `use std::error::Error;` 进行了引入)，它表示函数返回一个类型，该类型实现了 `Error` 特征，这样我们就无需指定具体的错误类型，否则你还需要查看 `fs::read_to_string` 返回的错误类型，然后复制到我们的 `run` 函数返回中，这么做一个是麻烦，最主要的是，一旦这么做，意味着我们无法在上层调用时统一处理错误，但是 `Box<dyn Error>` 不同，其它函数也可以返回这个特征对象，然后调用者就可以使用统一的方式来处理不同函数返回的 `Box<dyn Error>`。

明白了 `Box<dyn Error>` 的重要战略地位，接下来大家分析下，`fs::read_to_string` 返回的具体错误类型是怎么被转化为 `Box<dyn Error>` 的？其实原因在之前章节都有讲过，这里就不直接给出答案了，参见 [?-传播界的大明星](#)。

运行代码看看效果：

```

$ cargo run the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
--> src/main.rs:19:5
|
19 |     run(config);
|     ^^^^^^^^^^^^^^
|
= note: `#[warn(unused_must_use)]` on by default
= note: this `Result` may be an `Err` variant, which should be handled

warning: `minigrep` (bin "minigrep") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.71s
    Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!

```

没任何问题，不过 Rust 编译器也给出了善意的提示，那就是 `Result` 并没有被使用，这可能意味着存在错误的潜在可能性。

## 处理返回的错误

```

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

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        println!("Application error: {e}");
        process::exit(1);
    }
}

```

先回忆下在 `build` 函数调用时，我们怎么处理错误的？然后与这里的方式做一下对比，是不是发现了一些区别？

没错 `if let` 的使用让代码变得更简洁，可读性也更加好，原因是，我们并不关注 `run` 返回的 `ok` 值，因此只需要用 `if let` 去匹配是否存在错误即可。

好了，截止目前，代码看起来越来越美好了，距离我们的目标也只差一个：将主体逻辑代码分离到一个独立的文件 `lib.rs` 中。

## 分离逻辑代码到库包中

---

对于 Rust 的代码组织( 包和模块 )还不熟悉的同学，强烈建议回头温习下[这一章](#)。

---

首先，创建一个 `src/lib.rs` 文件，然后将所有的非 `main` 函数都移动到其中。代码大概类似：

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        // --snip--
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}
```

为了内容的简洁性，这里忽略了具体的实现，下一步就是在 `main.rs` 中引入 `lib.rs` 中定义的 `Config` 类型。

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = minigrep::run(config) {
        // --snip--
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

很明显，这里的 `mingrep::run` 的调用，以及 `Config` 的引入，跟使用其它第三方包已经没有任何区别，也意味着我们成功的将逻辑代码放置到一个独立的库包中，其它包只要引入和调用就行。

呼，一顿书写猛如虎，回头一看。。。这么长的篇幅就写了这么点简单的代码？？只能说，我也希望像很多国内的大学教材一样，只要列出定理和解题方法，然后留下足够的习题，就万事大吉了，但是咱们不行。

接下来，到了最喜(令)闻(人)乐(讨)见(厌)的环节：写测试代码，一起来开心吧。

# 测试驱动开发

开始之前，推荐大家先了解下[如何在 Rust 中编写测试代码](#)，这块儿内容不复杂，先了解下有利于本章的继续阅读

在之前的章节中，我们完成了对项目结构的重构，并将进入逻辑代码编程的环节，但在此之前，我们需要先编写一些测试代码，也是最近颇为流行的测试驱动开发模式(TDD, Test Driven Development)：

1. 编写一个注定失败的测试，并且失败的原因和你指定的一样
2. 编写一个成功的测试
3. 编写你的逻辑代码，直到通过测试

这三个步骤将在我们的开发过程中不断循环，直到所有的代码都开发完成并成功通过所有测试。

## 注定失败的测试用例

既然要添加测试，那之前的 `println!` 语句将没有大的用处，毕竟 `println!` 存在的目的就是为了让我们看到结果是否正确，而现在测试用例将取而代之。

接下来，在 `lib.rs` 文件中，添加 `tests` 模块和 `test` 函数：

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}
```

测试用例将在指定的内容中搜索 `duct` 字符串，目测可得：其中有一行内容是包含有目标字符串的。

但目前为止，还无法运行该测试用例，更何况还想幸灾乐祸的看其失败，原因是 `search` 函数还没有实现！毕竟是测试驱动、测试先行。

```
// in lib.rs
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

先添加一个简单的 `search` 函数实现，非常简单粗暴的返回一个空的数组，显而易见测试用例将成功通过，真是一个居心叵测的测试用例！

注意这里生命周期 `'a` 的使用，之前的章节有[详细介绍](#)，不太明白的同学可以回头看看。

喔，这么复杂的代码，都用上生命周期了！嘚瑟两下试试：

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 0.97s
Running unitests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `["safe, fast, productive."]`,
  right: `[]`, src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  tests::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

太棒了！它失败了...

# 务必成功的测试用例

接着就是测试驱动的第二步：编写注定成功的测试。当然，前提条件是实现我们的 `search` 函数。它包含以下步骤：

- 遍历迭代 `contents` 的每一行
- 检查该行内容是否包含我们的目标字符串
- 若包含，则放入返回值列表中，否则忽略
- 返回匹配到的返回值列表

## 遍历迭代每一行

Rust 提供了一个很便利的 `lines` 方法将目标字符串进行按行分割：

```
// in lib.rs
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

这里的 `lines` 返回一个迭代器，关于迭代器在后续章节会详细讲解，现在只要知道 `for` 可以遍历取出迭代器中的值即可。

## 在每一行中查询目标字符串

```
// in lib.rs
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

与之前的 `lines` 函数类似，Rust 的字符串还提供了 `contains` 方法，用于检查 `line` 是否包含待查询的 `query`。

接下来，只要返回合适的值，就可以完成 `search` 函数的编写。

## 存储匹配到的结果

简单，创建一个 `Vec` 动态数组，然后将查询到的每一个 `line` 推进数组中即可：

```
// in lib.rs
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

至此，`search` 函数已经完成了既定目标，为了检查功能是否正确，运行下我们之前编写的测试用例：

```
$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 1.22s
Running unit tests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Running unit tests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

测试通过，意味着我们的代码也完美运行，接下来就是在 `run` 函数中大显身手了。

## 在 run 函数中调用 search 函数

```
// in src/lib.rs
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{}{}", line);
    }

    Ok(())
}
```

好，再运行下看看结果，看起来我们距离成功从未如此之近！

```
$ cargo run -- frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38s
    Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

酷！成功查询到包含 `frog` 的行，再来试试 `body`：

```
$ cargo run -- body poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

完美，三行，一行不少，为了确保万无一失，再来试试查询一个不存在的单词：

```
cargo run -- monomorphization poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
    Running `target/debug/minigrep monomorphization poem.txt`
```

至此，章节开头的目标已经全部完成，接下来思考一个小问题：如果要为程序加上大小写不敏感的控制命令，由用户进行输入，该怎么实现比较好呢？毕竟在实际搜索查询中，同时支持大小写敏感和不敏感还是很重要的。

答案留待下一章节揭晓。

# 使用环境变量来增强程序

在上一章节中，留下了一个悬念，该如何实现用户控制的大小写敏感，其实答案很简单，你在其它程序中肯定也遇到过不少，例如如何控制 `panic` 后的栈展开？Rust 提供的解决方案是通过命令行参数来控制：

```
RUST_BACKTRACE=1 cargo run
```

与之类似，我们也可以使用环境变量来控制大小写敏感，例如：

```
IGNORE_CASE=1 cargo run -- to poem.txt
```

既然有了目标，那么一起来看看该如何实现吧。

## 编写大小写不敏感的测试用例

还是遵循之前的规则：测试驱动，这次是对一个新的大小写不敏感函数进行测试  
`search_case_insensitive`。

还记得 TDD 的测试步骤嘛？首先编写一个注定失败的用例：

```

// in src/lib.rs
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}

```

可以看到，这里新增了一个 `case_insensitive` 测试用例，并对 `search_case_insensitive` 进行了测试，结果显而易见，函数都没有实现，自然会失败。

接着来实现这个大小写不敏感的搜索函数：

```

pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}

```

跟之前一样，但是引入了一个新的方法 `to_lowercase`，它会将 `line` 转换成全小写的字符串，类似的方法在其它语言中也差不多，就不再赘述。

还要注意的是 `query` 现在是 `String` 类型，而不是之前的 `&str`，因为 `to_lowercase` 返回的是 `String`。

修改后，再来跑一次测试，看能否通过。

```

$ cargo test
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished test [unoptimized + debuginfo] target(s) in 1.33s
Running unitests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Running unitests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

```

Ok, TDD的第二步也完成了，测试通过，接下来就是最后一步，在 `run` 中调用新的搜索函数。但是在此之前，要新增一个配置项，用于控制是否开启大小写敏感。

```
// in lib.rs
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
```

接下来就是检查该字段，来判断是否启动大小写敏感：

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{}{}", config.ignore_case, line);
    }

    Ok(())
}
```

现在的问题来了，该如何控制这个配置项呢。这个就要借助于章节开头提到的环境变量，好在 Rust 的 `env` 包提供了相应的方法。

```

use std::env;
// --snip--

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}

```

`env::var` 没啥好说的，倒是 `is_ok` 值得说道下。该方法是 `Result` 提供的，用于检查是否有值，有就返回 `true`，没有则返回 `false`，刚好完美符合我们的使用场景，因为我们并不关心 `Ok<T>` 中具体的值。

运行下试试：

```

$ cargo run -- to poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!

```

看起来没有问题，接下来测试下大小写不敏感：

```

$ IGNORE_CASE=1 cargo run -- to poem.txt

```

```

Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!

```

大小写不敏感后，查询到的内容明显多了很多，也很符合我们的预期。

最后，给大家留一个小作业：同时使用命令行参数和环境变量的方式来控制大小写不敏感，其中环境变量的优先级更高，也就是两个都设置的情况下，优先使用环境变量的设置。

# 重定向错误信息的输出

迄今为止，所有的输出信息，无论 debug 还是 error 类型，都是通过 `println!` 宏输出到终端的标准输出(`stdout`)，但是对于程序来说，错误信息更适合输出到标准错误输出(`stderr`)。

这样修改后，用户就可以选择将普通的日志类信息输出到日志文件 1，然后将错误信息输出到日志文件 2，甚至还可以输出到终端命令行。

## 目前的错误输出位置

我们先来观察下，目前的输出信息包括错误，是否是如上面所说，都写到标准错误输出。

测试方式很简单，将标准错误输出的内容重定向到文件中，看看是否包含故意生成的错误信息即可。

```
$ cargo run > output.txt
```

首先，这里的运行没有带任何参数，因此会报出类如文件不存在的错误，其次，通过 `>` 操作符，标准输出上的内容被重定向到文件 `output.txt` 中，不再打印到控制上。

大家先观察下控制台，然后再看看 `output.txt`，是否发现如下的错误信息已经如期被写入到文件中？

```
Problem parsing arguments: not enough arguments
```

所以，可以得出一个结论，如果错误信息输出到标准输出，那么它们将跟普通的日志信息混在一起，难以分辨，因此我们需要将错误信息进行单独输出。

## 标准错误输出 `stderr`

将错误信息重定向到 `stderr` 很简单，只需在打印错误的地方，将 `println!` 宏替换为 `eprintln!` 即可。

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

接下来，还是同样的运行命令：

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

可以看到，日志信息成功的重定向到 `output.txt` 文件中，而错误信息由于 `eprintln!` 的使用，被写入到标准错误输出中，默认还是输出在控制台中。

再来试试没有错误的情况：

```
$ cargo run -- to poem.txt > output.txt
```

这次运行参数很正确，因此也没有任何错误信息产生，同时由于我们重定向了标准输出，因此相应的输出日志会写入到 `output.txt` 中，打开可以看到如下内容：

```
Are you nobody, too?
How dreary to be somebody!
```

至此，简易搜索程序 `minigrep` 已经基本完成，下一章节将使用迭代器进行部分改进，请大家在看完[迭代器章节](#)后，再回头阅读。

# 使用迭代器来改进我们的程序

本章节是可选内容，请大家在看完[迭代器章节](#)后，再来阅读

在之前的 `minigrep` 中，功能虽然已经 ok，但是一些细节上还值得打磨下，下面一起看看如何使用迭代器来改进 `Config::build` 和 `search` 的实现。

## 移除 `clone` 的使用

虽然之前有讲过为什么这里可以使用 `clone`，但是也许总有同学心有芥蒂，毕竟程序员嘛，都希望代码处处完美，而不是丑陋的处处妥协。

```
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

之前的代码大致长这样，两行 `clone` 着实有点啰嗦，好在，在学习完迭代器后，我们知道 `build` 函数实际上可以**直接拿走迭代器的所有权**，而不是去借用一个数组切片 `&[String]`。

这里先不给出代码，下面统一给出。

## 直接使用返回的迭代器

在之前的实现中，我们的 `args` 是一个动态数组：

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

当时还提到了 `collect` 方法的使用，相信大家学完迭代器后，对这个方法会有更加深入的认识。

现在呢，无需数组了，直接传入迭代器即可：

```
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // --snip--
}
```

如上所示，我们甚至省去了一行代码，原因是 `env::args` 可以直接返回一个迭代器，再作为 `Config::build` 的参数传入，下面再来改写 `build` 方法。

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
    // --snip--
```

为了可读性和更好的通用性，这里的 `args` 类型并没有使用本身的 `std::env::Args`，而是使用了特征约束的方式来描述 `impl Iterator<Item = String>`，这样意味着 `arg` 可以是任何实现了 `String` 迭代器的类型。

还有一点值得注意，由于迭代器的所有权已经转移到 `build` 内，因此可以直接对其进行修改，这里加上了 `mut` 关键字。

## 移除数组索引的使用

数组索引会越界，为了安全性和简洁性，使用 `Iterator` 特征自带的 `next` 方法是一个更好的选择：

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        // 第一个参数是程序名，由于无需使用，因此这里直接空调用一次
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

喔，上面使用了迭代器和模式匹配的代码，看上去是不是很 Rust？我想我们已经走在了正确的道路上。

## 使用迭代器适配器让代码更简洁

为了帮大家更好的回忆和对比，之前的 `search` 长这样：

```
// in lib.rs
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

引入了迭代器后，就连古板的 `search` 函数也可以变得更 rusty 些：

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Rock，让我们的函数编程 Style rock 起来，这种一行到底的写法有时真的让人沉迷。

## 总结

至此，整个大章节全部结束，本章没有试图覆盖已学的方方面面(也许未来会)，而是聚焦于 Rust 的一些核心知识：所有权、生命周期、借用、模式匹配等等。

强烈推荐大家忘记已有的一切，自己重新实现一遍 `minigrep`，甚至可以根据自己的想法和喜好，来完善一些，也欢迎在评论中附上自己的练习项目，供其它人学习参考(提个小建议，项目主页写清楚新增的功能、亮点等)。

从下一章开始，我们将正式开始 Rust 进阶学习，请深呼吸一口，然后问自己：你..准备好了吗？

# Rust 高级进阶

恭喜你，学会 Rust 基础后，金丹大道已在向你招手，大部分 Rust 代码对你来说都是家常便饭，简单得很。可是，对于一门难度传言在外的语言，怎么可能如此简单的就被征服，最难的生命周期，咱还没见过长啥样呢。

从本章开始，我们将进入 Rust 的进阶学习环节，与基础环节不同的是，由于你已经对 Rust 有了一定的认识，因此我们**不会再对很多细节进行翻来覆去的详细讲解，甚至会一带而过**。

总之，欢迎来到高级 Rust 的世界，全新的 Boss，全新的装备，你准备好了吗？

# 生命周期

何为高阶？一个字：难，二个字：很难，七个字：其实也没那么难。至于到底难不难，还是交给各位看官评判吧 :D

大家都知道，生命周期在 Rust 中是最难的部分之一，因此相关内容被分成了两个章节：基础和进阶，其中基础部分已经在之前学习过，下面一起来看看真正的 难 字怎么写。

# 深入生命周期

其实关于生命周期的常用特性，在上一节中，我们已经概括得差不多了，本章主要讲解生命周期的一些高级或者不为人知的特性。对于新手，完全可以跳过本节内容，进行下一章节的学习。

## 不太聪明的生命周期检查

在 Rust 语言学习中，一个很重要的部分就是阅读一些你可能不经常遇到，但是一旦遇到就难以理解的代码，这些代码往往最令人头疼的就是生命周期，这里我们就来看看一些本以为可以编译，但是却因为生命周期系统不够聪明导致编译失败的代码。

### 例子 1

```
#[derive(Debug)]
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self {
        &*self
    }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
    println!("{:?}", loan);
}
```

上面的代码中，`foo.mutate_and_share()` 虽然借用了 `&mut self`，但是它最终返回的是一个 `&self`，然后赋值给 `loan`，因此理论上来说它最终是进行了不可变借用，同时 `foo.share()` 也进行了不可变借用，那么根据 Rust 的借用规则：多个不可变借用可以同时存在，因此该代码应该编译通过。

事实上，运行代码后，你将看到一个错误：

```

error[E0502]: cannot borrow `foo` as immutable because it is also borrowed as mutable
--> src/main.rs:12:5
|
11 |     let loan = foo.mutate_and_share();
|           ----- mutable borrow occurs here
12 |     foo.share();
|     ^^^^^^^^^^^^ immutable borrow occurs here
13 |     println!("{}:?", loan);
|           ---- mutable borrow later used here

```

编译器的提示在这里其实有些难以理解，因为可变借用仅在 `mutate_and_share` 方法内部有效，出了该方法后，就只有返回的不可变借用，因此，按理来说可变借用不应该在 `main` 的作用范围内存在。

对于这个反直觉的事情，让我们用生命周期来解释下，可能你就很好理解了：

```

struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self {
        &'a *self
    }
    fn share<'a>(&'a self) {}
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
            'd: {
                Foo::share::<'d>(&'d foo);
            }
            println!("{}:?", loan);
        }
    }
}

```

以上是模拟了编译器的生命周期标注后的代码，可以注意到 `&mut foo` 和 `loan` 的生命周期都是 `'c`。

还记得生命周期消除规则中的第三条吗？因为该规则，导致了 `mutate_and_share` 方法中，参数 `&mut self` 和返回值 `&self` 的生命周期是相同的，因此，若返回值的生命周期在 `main` 函数有效，那 `&mut self` 的借用也是在 `main` 函数有效。

这就解释了可变借用为啥会在 `main` 函数作用域内有效，最终导致 `foo.share()` 无法再进行不可变借用。

总结下：`&mut self` 借用的生命周期和 `loan` 的生命周期相同，将持续到 `println` 结束。而在此期间 `foo.share()` 又进行了一次不可变 `&foo` 借用，违背了可变借用与不可变借用不能同时存在的规则，最

终导致了编译错误。

上述代码实际上完全是正确的，但是因为生命周期系统的“粗糙实现”，导致了编译错误，目前来说，遇到这种生命周期系统不够聪明导致的编译错误，我们也没有太好的办法，只能修改代码去满足它的需求，并期待以后它会更聪明。

## 例子 2

再来看一个例子：

```
#![allow(unused)]
fn main() {
    use std::collections::HashMap;
    use std::hash::Hash;
    fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut V
    where
        K: Clone + Eq + Hash,
        V: Default,
    {
        match map.get_mut(&key) {
            Some(value) => value,
            None => {
                map.insert(key.clone(), V::default());
                map.get_mut(&key).unwrap()
            }
        }
    }
}
```

这段代码不能通过编译的原因是编译器未能精确地判断出某个可变借用不再需要，反而谨慎的给该借用安排了一个很大的作用域，结果导致后续的借用失败：

```

error[E0499]: cannot borrow `*map` as mutable more than once at a time
--> src/main.rs:13:17
|
5 |         fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut V
|             -- lifetime `'m` defined here
...
10 |             match map.get_mut(&key) {
|                 ----- first mutable borrow occurs here
|                 |
|                 |             Some(value) => value,
|                 |             None => {
|                 |                 map.insert(key.clone(), V::default());
|                 |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ second mutable borrow
occurs here
14 |                 map.get_mut(&key).unwrap()
15 |             }
16 |         }
|         ----- returning this value requires that `*map` is borrowed for `'m`

```

分析代码可知在 `match map.get_mut(&key)` 方法调用完成后，对 `map` 的可变借用就可以结束了。但从报错看来，编译器不太聪明，它认为该借用会持续到整个 `match` 语句块的结束(第 16 行处)，这便造成了后续借用的失败。

类似的例子还有很多，由于篇幅有限，就不在这里一一列举，如果大家想要阅读更多的类似代码，可以看看[《Rust 代码鉴赏》](#)一书。

## 无界生命周期

不安全代码(`unsafe`)经常会凭空产生引用或生命周期，这些生命周期被称为是 **无界(unbound)** 的。

无界生命周期往往是在解引用一个裸指针(裸指针 `raw pointer`)时产生的，换句话说，它是凭空产生的，因为输入参数根本就没有这个生命周期：

```

fn f<'a, T>(x: *const T) -> &'a T {
    unsafe {
        &x
    }
}

```

上述代码中，参数 `x` 是一个裸指针，它并没有任何生命周期，然后通过 `unsafe` 操作后，它被进行了解引用，变成了一个 Rust 的标准引用类型，该类型必须要有生命周期，也就是 `'a`。

可以看出 '`a`' 是凭空产生的，因此它是无界生命周期。这种生命周期由于没有受到任何约束，因此它想要多大就多大，这实际上比 '`static`' 要强大。例如 `&'static &'a T` 是无效类型，但是无界生命周期 `&'unbounded &'a T` 会被视为 `&'a &'a T` 从而通过编译检查，因为它可大可小，就像孙猴子的金箍棒一般。

我们在实际应用中，要尽量避免这种无界生命周期。最简单的避免无界生命周期的方式就是在函数声明中运用生命周期消除规则。**若一个输出生命周期被消除了，那么必定因为有一个输入生命周期与之对应。**

## 生命周期约束 HRTB

生命周期约束跟特征约束类似，都是通过形如 '`a: b`' 的语法，来说明两个生命周期的长短关系。

`'a: 'b`

假设有两个引用 `&'a i32` 和 `&'b i32`，它们的生命周期分别是 '`a`' 和 '`b`'，若 '`a >= b`'，则可以定义 '`'a: 'b`'，表示 '`a`' 至少要活得跟 '`b`' 一样久。

```
struct DoubleRef<'a, 'b: 'a, T> {
    r: &'a T,
    s: &'b T
}
```

例如上述代码定义一个结构体，它拥有两个引用字段，类型都是泛型 `T`，每个引用都拥有自己的生命周期，由于我们使用了生命周期约束 '`b: a`'，因此 '`b`' 必须活得比 '`a`' 久，也就是结构体中的 `s` 字段引用的值必须要比 `r` 字段引用的值活得要久。

`T: 'a`

表示类型 `T` 必须比 '`a`' 活得要久：

```
struct Ref<'a, T: 'a> {
    r: &'a T
}
```

因为结构体字段 `r` 引用了 `T`，因此 `r` 的生命周期 '`a`' 必须要比 `T` 的生命周期更短(被引用者的生命周期必须要比引用长)。

在 Rust 1.30 版本之前，该写法是必须的，但是从 1.31 版本开始，编译器可以自动推导 `T: 'a` 类型的约束，因此我们只需这样写即可：

```
struct Ref<'a, T> {
    r: &'a T
}
```

来看一个使用了生命周期约束的综合例子：

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a: 'b, 'b> ImportantExcerpt<'a> {
    fn announce_and_return_part(&'a self, announcement: &'b str) -> &'b str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

上面的例子中必须添加约束 `'a: 'b` 后，才能成功编译，因为 `self.part` 的生命周期与 `self` 的生命周期一致，将 `&'a` 类型的生命周期强行转换为 `&'b` 类型，会报错，只有在 `'a >= 'b` 的情况下，`'a` 才能转换成 `'b`。

## 闭包函数的消除规则

先来看一段简单的代码：

```
fn fn_elision(x: &i32) -> &i32 { x }
let closure_elision = |x: &i32| -> &i32 { x };
```

乍一看，这段代码比古天乐还平平无奇，能有什么问题呢？来，拄拐走两圈试试：

```
error: lifetime may not live long enough
--> src/main.rs:39:39
|
39 |     let closure = |x: &i32| -> &i32 { x }; // fails
|           -           -           ^ returning this value requires that ``1``
must outlive ``2``
|
|           |           |
|           |           let's call the lifetime of this reference ``2``
|           let's call the lifetime of this reference ``1``
```

咦？竟然报错了，明明两个一模一样功能的函数，一个正常编译，一个却报错，错误原因是编译器无法推断返回的引用和传入的引用谁活得更久！

真的是非常奇怪的错误，学过上一节的读者应该都记得这样一条生命周期消除规则：**如果函数参数中只有一个引用类型，那该引用的生命周期会被自动分配给所有的返回引用**。我们当前的情况完美符合，`function` 函数的顺利编译通过，就充分说明了问题。

先给出一个结论：**这个问题，可能很难被解决，建议大家遇到后，还是老老实实用正常的函数，不要秀闭包了。**

对于函数的生命周期而言，它的消除规则之所以能生效是因为它的生命周期完全体现在签名的引用类型上，在函数体中无需任何体现：

```
fn fn_elision(x: &i32) -> &i32 {..}
```

因此编译器可以做各种编译优化，也很容易根据参数和返回值进行生命周期的分析，最终得出消除规则。

可是闭包，并没有函数那么简单，它的生命周期分散在参数和闭包函数体中(主要是它没有确切的返回值签名)：

```
let closure_slision = |x: &i32| -> &i32 { x };
```

编译器就必须深入到闭包函数体中，去分析和推测生命周期，复杂度因此急剧提升：试想一下，编译器该如何从复杂的上下文中分析出参数引用的生命周期和闭包体中生命周期的关系？

由于上述原因(当然，实际情况复杂的多)，Rust 语言开发者目前其实是有意针对函数和闭包实现了两种不同的生命周期消除规则。

---

## 用 Fn 特征解决闭包生命周期

之前我们提到了很难解决，但是并没有完全堵死(论文字的艺术-，-)这不 @Ykong1337 同学就带了一个解决方法，为他点赞！

```
fn main() {
    let closure_slision = fun(|x: &i32| -> &i32 { x });
    assert_eq!(*closure_slision(&45), 45);
    // Passed !
}

fn fun<T, F: Fn(&T) -> &T>(f: F) -> F {
    f
}
```

---

## NLL (Non-Lexical Lifetime)

之前我们在[引用与借用](#)那一章其实有讲到过这个概念，简单来说就是：**引用的生命周期正常来说应该从借用开始一直持续到作用域结束**，但是这种规则会让多引用共存的情况变得更复杂：

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

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);
    // 新编译器中，r1, r2作用域在这里结束

    let r3 = &mut s;
    println!("{}", r3);
}
```

按照上述规则，这段代码将会报错，因为 `r1` 和 `r2` 的不可变引用将持续到 `main` 函数结束，而在此范围内，我们又借用了 `r3` 的可变引用，这违反了借用的规则：要么多个不可变借用，要么一个可变借用。

好在，该规则从 1.31 版本引入 NLL 后，就变成了：**引用的生命周期从借用处开始，一直持续到最后一次使用的地方。**

按照最新的规则，我们再来分析一下上面的代码。`r1` 和 `r2` 不可变借用在 `println!` 后就不再使用，因此生命周期也随之结束，那么 `r3` 的借用就不再违反借用的规则，皆大欢喜。

再来看一段关于 NLL 的代码解释：

```
let mut u = 0i32;
let mut v = 1i32;
let mut w = 2i32;

// lifetime of `a` = α ∪ β ∪ γ
let mut a = &mut u;          // --+ α. lifetime of `&mut u`  ---+ lexical "lifetime" of
`&mut u`, `&mut u`, `&mut w` and `a`
use(a);                   //   |
*a = 3; // <-----+           |
...                      //           |
a = &mut v;              // --+ β. lifetime of `&mut v`  |
use(a);                  //   |
*a = 4; // <-----+           |
...                      //           |
a = &mut w;              // --+ γ. lifetime of `&mut w`  |
use(a);                  //   |
*a = 5; // <-----+ <-----+           |
```

这段代码一目了然，`a` 有三段生命周期：`α`，`β`，`γ`，每一段生命周期都随着当前值的最后一次使用而结束。

在实际项目中，NLL 规则可以大幅减少引用冲突的情况，极大的便利了用户，因此广受欢迎，最终该规则甚至演化成一个独立的项目，未来可能会进一步简化我们的使用，Polonius：

- 项目地址
- 具体介绍

## Reborrow 再借用

学完 NLL 后，我们就有了一定的基础，可以继续学习关于借用和生命周期的一个高级内容：**再借用**。

先来看一段代码：

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_to(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}

fn main() {
    let mut p = Point { x: 0, y: 0 };
    let r = &mut p;
    let rr: &Point = &*r;

    println!("{:?}", rr);
    r.move_to(10, 10);
    println!("{:?}", r);
}
```

以上代码，大家可能会觉得可变引用 `r` 和不可变引用 `rr` 同时存在会报错吧？但是事实上并不会，原因在于 `rr` 是对 `r` 的再借用。

对于再借用而言，`rr` 再借用时不会破坏借用规则，但是你不能在它的生命周期内再使用原来的借用 `r`，来看看对上段代码的分析：

```

fn main() {
    let mut p = Point { x: 0, y: 0 };
    let r = &mut p;
    // reborrow! 此时对`r`的再借用不会导致跟上面的借用冲突
    let rr: &Point = &*r;

    // 再借用`rr`最后一次使用发生在这里，在它的生命周期中，我们并没有使用原来的借用`r`，因此不会报
错
    println!("{:?}", rr);

    // 再借用结束后，才去使用原来的借用`r`
    r.move_to(10, 10);
    println!("{:?}", r);
}

```

再来看一个例子：

```

use std::vec::Vec;
fn read_length(strings: &mut Vec<String>) -> usize {
    strings.len()
}

```

如上所示，函数体内对参数的二次借用也是典型的 Reborrow 场景。

那么下面让我们来做件坏事，破坏这条规则，使其报错：

```

fn main() {
    let mut p = Point { x: 0, y: 0 };
    let r = &mut p;
    let rr: &Point = &*r;

    r.move_to(10, 10);

    println!("{:?}", rr);

    println!("{:?}", r);
}

```

果然，破坏永远比重建简单 :) 只需要在 `rr` 再借用的生命周期内使用一次原来的借用 `r` 即可！

## 生命周期消除规则补充

在上一节中，我们介绍了三大基础生命周期消除规则，实际上，随着 Rust 的版本进化，该规则也在不断演进，这里再介绍几个常见的消除规则：

## impl 块消除

```
impl<'a> Reader for BufReader<'a> {
    // methods go here
    // impl内部实际上没有用到'a
}
```

如果你以前写的 `impl` 块长上面这样，同时在 `impl` 内部的方法中，根本就没有用到 `'a`，那就可以写成下面的代码形式。

```
impl Reader for BufReader<'_> {
    // methods go here
}
```

`'_` 生命周期表示 `BufReader` 有一个不使用的生命周期，我们可以忽略它，无需为它创建一个名称。

歪个楼，有读者估计会发问：既然用不到 `'a`，为何还要写出来？如果你仔细回忆下上一节的内容，里面有一句专门用粗体标注的文字：**生命周期参数也是类型的一部分**，因此 `BufReader<'a>` 是一个完整的类型，在实现它的时候，你不能把 `'a` 给丢了！

## 生命周期约束消除

```
// Rust 2015
struct Ref<'a, T: 'a> {
    field: &'a T
}

// Rust 2018
struct Ref<'a, T> {
    field: &'a T
}
```

在本节的生命周期约束中，也提到过，新版本 Rust 中，上面情况中的 `T: 'a` 可以被消除掉，当然，你也可以显式的声明，但是会影响代码可读性。关于类似的场景，Rust 团队计划在未来提供更多的消除规则，但是，你懂的，计划未来就等于未知。

## 一个复杂的例子

下面是一个关于生命周期声明过大的例子，会较为复杂，希望大家能细细阅读，它能帮你对生命周期的理解更加深入。

```

struct Interface<'a> {
    manager: &'a mut Manager<'a>
}

impl<'a> Interface<'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>,
}

impl<'a> List<'a> {
    pub fn get_interface(&'a mut self) -> Interface {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // 下面的调用会失败，因为同时有不可变/可变借用
    // 但是Interface在之前调用完成后就应该被释放了
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}" , list.manager.text);
}

```

运行后报错：

```
error[E0502]: cannot borrow `list` as immutable because it is also borrowed as
mutable // `list`无法被借用, 因为已经被可变借用
--> src/main.rs:40:14
|
34 |     list.get_interface().noop();
|     ---- mutable borrow occurs here // 可变借用发生在这里
...
40 |     use_list(&list);
|     ^^^^^^
|     |
|     immutable borrow occurs here // 新的不可变借用发生在这
|     mutable borrow later used here // 可变借用在这里结束
```

这段代码看上去并不复杂，实际上难度挺高的，首先在直觉上，`list.get_interface()` 借用的可变引用，按理来说应该在这行代码结束后，就归还了，但是为什么还能持续到 `use_list(&list)` 后面呢？

这是因为我们在 `get_interface` 方法中声明的 `lifetime` 有问题，该方法的参数的生命周期是 '`a`'，而 `List` 的生命周期也是 '`a`'，说明该方法至少活得跟 `List` 一样久，再回到 `main` 函数中，`list` 可以活到 `main` 函数的结束，因此 `list.get_interface()` 借用的可变引用也会活到 `main` 函数的结束，在此期间，自然无法再进行借用了。

要解决这个问题，我们需要为 `get_interface` 方法的参数给予一个不同于 `List<'a>` 的生命周期 '`b`'，最终代码如下：

```

struct Interface<'b, 'a: 'b> {
    manager: &'b mut Manager<'a>
}

impl<'b, 'a: 'b> Interface<'b, 'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>,
}

impl<'a> List<'a> {
    pub fn get_interface<'b>(&'b mut self) -> Interface<'b, 'a>
    where 'a: 'b {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {

    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // 下面的调用可以通过，因为Interface的生命周期不需要跟list一样长
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}

```

至此，生命周期终于完结，两章超级长的内容，可以满足几乎所有对生命周期的学习目标。学完生命周期，意味着你正式入门了 Rust，只要再掌握几个常用概念，就可以上手写项目了。

# &'static 和 T: 'static

Rust 的难点之一就在于它有不少容易混淆的概念，例如 `&str`、`str` 与 `String`，再比如本文标题那两位。不过与字符串也有不同，这两位对于普通用户来说往往是无需进行区分的，但是当大家想要深入学习或使用 Rust 时，它们就会成为成功路上的拦路虎了。

与生命周期的其它章节不同，本文短小精悍，阅读过程可谓相当轻松愉快，话不多说，let's go。

`'static` 在 Rust 中是相当常见的，例如字符串字面值就具有 `'static` 生命周期：

```
fn main() {
    let mark_twain: &str = "Samuel Clemens";
    print_author(mark_twain);
}
fn print_author(author: &'static str) {
    println!("{}", author);
}
```

除此之外，特征对象的生命周期也是 `'static`，例如[这里](#)所提到的。

除了 `&'static` 的用法外，我们在另外一种场景中也可以见到 `'static` 的使用：

```
use std::fmt::Display;
fn main() {
    let mark_twain = "Samuel Clemens";
    print(&mark_twain);
}

fn print<T: Display + 'static>(message: &T) {
    println!("{}", message);
}
```

在这里，很明显 `'static` 是作为生命周期约束来使用了。**那么问题来了，`&'static` 和 `T: 'static` 的用法到底有何区别？**

## &'static

`&'static` 对于生命周期有着非常强的要求：一个引用必须要活得跟剩下的程序一样久，才能被标注为 `&'static`。

对于字符串字面量来说，它直接被打包到二进制文件中，永远不会被 `drop`，因此它能跟程序活得一样久，自然它的生命周期是 `'static`。

但是，`&'static` 生命周期针对的仅仅是引用，而不是持有该引用的变量，对于变量来说，还是要遵循相应的作用域规则：

```
use std::slice::from_raw_parts, str::from_utf8_unchecked;

fn get_memory_location() -> (usize, usize) {
    // "Hello World" 是字符串字面量，因此它的生命周期是 `&'static`。
    // 但持有它的变量 `string` 的生命周期就不一样了，它完全取决于变量作用域，对于该例子来说，也就是当前的函数范围
    let string = "Hello World!";
    let pointer = string.as_ptr() as usize;
    let length = string.len();
    (pointer, length)
    // `string` 在这里被 drop 释放
    // 虽然变量被释放，无法再被访问，但是数据依然还会继续存活
}

fn get_str_at_location(pointer: usize, length: usize) -> &'static str {
    // 使用裸指针需要 `unsafe{}` 语句块
    unsafe { from_utf8_unchecked(from_raw_parts(pointer as *const u8, length)) }
}

fn main() {
    let (pointer, length) = get_memory_location();
    let message = get_str_at_location(pointer, length);
    println!(
        "The {} bytes at 0x{:X} stored: {}",
        length, pointer, message
    );
    // 如果大家想知道为何处理裸指针需要 `unsafe`，可以试着反注释以下代码
    // let message = get_str_at_location(1000, 10);
}
```

上面代码有两点值得注意：

- `&'static` 的引用确实可以和程序活得一样久，因为我们通过 `get_str_at_location` 函数直接取到了对应的字符串
- 持有 `&'static` 引用的变量，它的生命周期受到作用域的限制，大家务必不要搞混了

## T: `'static`

相比起来，这种形式的约束就有些复杂了。

首先，在以下两种情况下，`T: 'static` 与 `&'static` 有相同的约束：`T` 必须活得和程序一样久。

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

fn print_it<T: Debug + 'static>( input: T) {
    println!( "'static value passed in is: {:?}", input );
}

fn print_it1( input: impl Debug + 'static ) {
    println!( "'static value passed in is: {:?}", input );
}

fn main() {
    let i = 5;

    print_it(&i);
    print_it1(&i);
}
```

以上代码会报错，原因很简单：`&i` 的生命周期无法满足 `'static` 的约束，如果大家将 `i` 修改为常量，那自然一切 OK。

见证奇迹的时候，请不要眨眼，现在我们来稍微修改下 `print_it` 函数：

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

fn print_it<T: Debug + 'static>( input: &T) {
    println!( "'static value passed in is: {:?}", input );
}

fn main() {
    let i = 5;

    print_it(&i);
}
```

这段代码竟然不报错了！原因在于我们约束的是 `T`，但是使用的却是它的引用 `&T`，换而言之，我们根本没有直接使用 `T`，因此编译器就没有去检查 `T` 的生命周期约束！它只要确保 `&T` 的生命周期符合规则即可，在上面代码中，它自然是符合的。

再来看一个例子：

```

use std::fmt::Display;

fn main() {
    let r1;
    let r2;
    {
        static STATIC_EXAMPLE: i32 = 42;
        r1 = &STATIC_EXAMPLE;
        let x = "&'static str";
        r2 = x;
        // r1 和 r2 持有的数据都是 'static 的，因此在花括号结束后，并不会被释放
    }

    println!("&'static i32: {}", r1); // -> 42
    println!("&'static str: {}", r2); // -> &'static str

    let r3: &str;

    {
        let s1 = "String".to_string();

        // s1 虽然没有 'static 生命周期，但是它依然可以满足 T: 'static 的约束
        // 充分说明这个约束是多么的弱。。
        static_bound(&s1);

        // s1 是 String 类型，没有 'static 的生命周期，因此下面代码会报错
        r3 = &s1;

        // s1 在这里被 drop
    }
    println!("{}", r3);
}

fn static_bound<T: Display + 'static>(t: &T) {
    println!("{}", t);
}

```

## static 到底针对谁？

大家有没有想过，到底是 `&'static` 这个引用还是该引用指向的数据活得跟程序一样久呢？

**答案是引用指向的数据，而引用本身是要遵循其作用域范围的，我们来简单验证下：**

```
fn main() {
{
    let static_string = "I'm in read-only memory";
    println!("static_string: {}", static_string);

    // 当 `static_string` 超出作用域时，该引用不能再被使用，但是数据依然会存在于 binary 所
    // 占用的内存中
}

    println!("static_string reference remains alive: {}", static_string);
}
```

以上代码不出所料会报错，原因在于虽然字符串字面量 "I'm in read-only memory" 的生命周期是 `'static`，但是持有它的引用并不是，它的作用域在内部花括号 `}` 处就结束了。

## 课后练习

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。（本节暂无习题解答）

## 总结

总之，`&'static` 和 `T: 'static` 大体上相似，相比起来，后者的使用形式会更加复杂一些。

至此，相信大家对于 `'static` 和 `T: 'static` 也有了清晰的理解，那么我们应该如何使用它们呢？

作为经验之谈，可以这么来：

- 如果你需要添加 `&'static` 来让代码工作，那很可能是设计上出问题了
- 如果你希望满足和取悦编译器，那就使用 `T: 'static`，很多时候它都能解决问题

一个小知识，在 Rust 标准库中，有 48 处用到了 `&'static`，112 处用到了 `T: 'static`，看来取悦编译器不仅仅是菜鸟需要的，高手也经常用到：）

# 函数式编程

罗马不是一天建成的，编程语言亦是如此，每一门编程语言在借鉴前辈的同时，也会提出自己独有的特性，Rust 即是如此。当站在巨人肩膀上时，一个人所能看到的就更高更远，恰好，我们看到了函数式语言的优秀特性，例如：

- 使用函数作为参数进行传递
- 使用函数作为函数返回值
- 将函数赋值给变量

见猎心喜，我们忍不住就借鉴了过来，于是你能看到本章的内容，天下语言一大。。。跑题了。

关于函数式编程到底是什么的争论由来已久，本章节并不会踏足这个泥潭，因此我们在这里主要关注的是函数式特性：

- 闭包 Closure
- 迭代器 Iterator
- 模式匹配
- 枚举

其中后两个在前面章节我们已经深入学习过，因此本章的重点就是闭包和迭代器，**这些函数式特性可以让代码的可读性和易写性大幅提升**。对于 Rust 语言来说，掌握这两者就相当于你同时拥有了倚天剑屠龙刀，威力无穷。

# 闭包 Closure

闭包这个词语由来已久，自上世纪 60 年代就由 Scheme 语言引进之后，被广泛用于函数式编程语言中，进入 21 世纪后，各种现代化的编程语言也都不约而同地把闭包作为核心特性纳入到语言设计中来。那么到底何为闭包？

闭包是一种匿名函数，它可以赋值给变量也可以作为参数传递给其它函数，不同于函数的是，它允许捕获调用者作用域中的值，例如：

```
fn main() {  
    let x = 1;  
    let sum = |y| x + y;  
  
    assert_eq!(3, sum(2));  
}
```

上面的代码展示了非常简单的闭包 `sum`，它拥有一个入参 `y`，同时捕获了作用域中的 `x` 的值，因此调用 `sum(2)` 意味着将 2（参数 `y`）跟 1（`x`）进行相加，最终返回它们的和： 3。

可以看到 `sum` 非常符合闭包的定义：可以赋值给变量，允许捕获调用者作用域中的值。

## 使用闭包来简化代码

### 传统函数实现

想象一下，我们要进行健身，用代码怎么实现（写代码什么鬼，健身难道不应该去健身房嘛？答曰：健身太累了，还是虚拟健身好，点到为止）？这里是我的想法：

```

use std::thread;
use std::time::Duration;

// 开始健身，好累，我得发出声音: muuuu...
fn muuuuu(intensity: u32) -> u32 {
    println!("muuuu.....");
    thread::sleep(Duration::from_secs(2));
    intensity
}

fn workout(intensity: u32, random_number: u32) {
    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑!",
            muuuuu(intensity)
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推!",
            muuuuu(intensity)
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟!",
            muuuuu(intensity)
        );
    }
}

fn main() {
    // 强度
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}

```

可以看到，在健身时我们根据想要的强度来调整具体的动作，然后调用 `muuuuu` 函数来开始健身。这个程序本身很简单，没啥好说的，但是假如未来不用 `muuuuu` 函数了，是不是得把所有 `muuuuu` 都替换成，比如说 `woooo`？如果 `muuuuu` 出现了几十次，那意味着我们要修改几十处地方。

## 函数变量实现

一个可行的办法是，把函数赋值给一个变量，然后通过变量调用：

```

use std::thread;
use std::time::Duration;

// 开始健身，好累，我得发出声音：muuuu...
fn muuuuu(intensity: u32) -> u32 {
    println!("muuuu.....");
    thread::sleep(Duration::from_secs(2));
    intensity
}

fn workout(intensity: u32, random_number: u32) {
    let action = muuuuu;
    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑！",
            action(intensity)
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推！",
            action(intensity)
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟！",
            action(intensity)
        );
    }
}

fn main() {
    // 强度
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}

```

经过上面修改后，所有的调用都通过 `action` 来完成，若未来声(动)音(作)变了，只要修改为 `let action = woooo` 即可。

但是问题又来了，若 `intensity` 也变了怎么办？例如变成 `action(intensity + 1)`，那你又得哐哐哐修改几十处调用。

该怎么办？没太好的办法了，只能祭出大杀器：闭包。

## 闭包实现

上面提到 `intensity` 要是变化怎么办，简单，使用闭包来捕获它，这是我们的拿手好戏：

```
use std::thread;
use std::time::Duration;

fn workout(intensity: u32, random_number: u32) {
    let action = || {
        println!("muuuu.....");
        thread::sleep(Duration::from_secs(2));
        intensity
    };

    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑！",
            action()
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推！",
            action()
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟！",
            action()
        );
    }
}

fn main() {
    // 动作次数
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}
```

在上面代码中，无论你要修改什么，只要修改闭包 `action` 的实现即可，其它地方只负责调用，完美解决了我们的问题！

Rust 闭包在形式上借鉴了 Smalltalk 和 Ruby 语言，与函数最大的不同就是它的参数是通过 `|param1|` 的形式进行声明，如果是多个参数就 `|param1, param2, ...|`，下面给出闭包的形式定义：

```
|param1, param2, ...| {  
    语句1;  
    语句2;  
    返回表达式  
}
```

如果只有一个返回表达式的话，定义可以简化为：

```
|param1| 返回表达式
```

上例中还有两点值得注意：

- **闭包中最后一行表达式返回的值，就是闭包执行后的返回值**，因此 `action()` 调用返回了 `intensity` 的值 `10`
- `let action = ||...|` 只是把闭包赋值给变量 `action`，并不是把闭包执行后的结果赋值给 `action`，因此这里 `action` 就相当于闭包函数，可以跟函数一样进行调用： `action()`

## 闭包的类型推导

Rust 是静态语言，因此所有的变量都具有类型，但是得益于编译器的强大类型推导能力，在很多时候我们并不需要显式地去声明类型，但是显然函数并不在此列，必须手动为函数的所有参数和返回值指定类型，原因在于函数往往会被作为 API 提供给你的用户，因此你的用户必须在使用时知道传入参数的类型和返回值类型。

与函数相反，闭包并不会作为 API 对外提供，因此它可以享受编译器的类型推导能力，无需标注参数和返回值的类型。

为了增加代码可读性，有时候我们会显式地给类型进行标注，出于同样的目的，也可以给闭包标注类型：

```
let sum = |x: i32, y: i32| -> i32 {  
    x + y  
}
```

与之相比，不标注类型的闭包声明会更简洁些：`let sum = |x, y| x + y`，需要注意的是，针对 `sum` 闭包，如果你只进行了声明，但是没有使用，编译器会提示你为 `x, y` 添加类型标注，因为它缺乏必要的上下文：

```
let sum = |x, y| x + y;  
let v = sum(1, 2);
```

这里我们使用了 `sum`，同时把 `1` 传给了 `x`，`2` 传给了 `y`，因此编译器才可以推导出 `x,y` 的类型为 `i32`。

下面展示了同一个功能的函数和闭包实现形式：

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|           { x + 1 };
let add_one_v4 = |x|           x + 1 ;
```

可以看出第一行的函数和后面的闭包其实在形式上是非常接近的，同时三种不同的闭包也展示了三种不同的使用方式：省略参数、返回值类型和花括号对。

虽然类型推导很好用，但是它不是泛型，**当编译器推导出一种类型后，它就会一直使用该类型**：

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

首先，在 `s` 中，编译器为 `x` 推导出类型 `String`，但是紧接着 `n` 试图用 `5` 这个整型去调用闭包，跟编译器之前推导的 `String` 类型不符，因此报错：

```
error[E0308]: mismatched types
--> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |     ^
  |     |
  |     expected struct `String`, found integer // 期待String
类型, 却发现一个整数
  |                                     help: try using a conversion method: `5.to_string()`
```

## 结构体中的闭包

假设我们要实现一个简易缓存，功能是获取一个值，然后将其缓存起来，那么可以这样设计：

- 一个闭包用于获取值
- 一个变量，用于存储该值

可以使用结构体来代表缓存对象，最终设计如下：

```
struct Cacher<T>
where
    T: Fn(u32) -> u32,
{
    query: T,
    value: Option<u32>,
}
```

等等，我都跟着这本教程学完 Rust 基础了，为何还有我不认识的东东？`Fn(u32) -> u32` 是什么鬼？别急，先回答你第一个问题：骚年，too young too naive，你以为 Rust 的语法特性就基础入门那一些吗？太年轻了！如果是长征，你才刚到赤水河。

其实，可以看得出这一长串是 `T` 的特征约束，再结合之前的已知信息：`query` 是一个闭包，大概可以推测出，`Fn(u32) -> u32` 是一个特征，用来表示 `T` 是一个闭包类型？Bingo，恭喜你，答对了！

那为什么不用具体的类型来标注 `query` 呢？原因很简单，每一个闭包实例都有独属于自己的类型，即使两个签名一模一样的闭包，它们的类型也是不同的，因此你无法用一个统一的类型来标注 `query` 闭包。

而标准库提供的 `Fn` 系列特征，再结合特征约束，就能很好的解决了这个问题。`T: Fn(u32) -> u32` 意味着 `query` 的类型是 `T`，该类型必须实现了相应的闭包特征 `Fn(u32) -> u32`。从特征的角度来看它长得非常反直觉，但是如果从闭包的角度来看又极其符合直觉，不得不佩服 Rust 团队的鬼才设计。。。

特征 `Fn(u32) -> u32` 从表面来看，就对闭包形式进行了显而易见的限制：**该闭包拥有一个 `u32` 类型的参数，同时返回一个 `u32` 类型的值。**

---

需要注意的是，其实 `Fn` 特征不仅仅适用于闭包，还适用于函数，因此上面的 `query` 字段除了使用闭包作为值外，还能使用一个具名的函数来作为它的值

---

接着，为缓存实现方法：

```

impl<T> Cacher<T>
where
    T: Fn(u32) -> u32,
{
    fn new(query: T) -> Cacher<T> {
        Cacher {
            query,
            value: None,
        }
    }

    // 先查询缓存值 `self.value`，若不存在，则调用 `query` 加载
    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.query)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}

```

上面的缓存有一个很大的问题：只支持 `u32` 类型的值，若我们想要缓存 `&str` 类型，显然就行不通了，因此需要将 `u32` 替换成泛型 `E`，该练习就留给读者自己完成，具体代码可以参考[这里](#)

## 捕获作用域中的值

在之前代码中，我们一直在用闭包的匿名函数特性（赋值给变量），然而闭包还拥有一项函数所不具备的特性：捕获作用域中的值。

```

fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}

```

上面代码中，`x` 并不是闭包 `equal_to_x` 的参数，但是它依然可以去使用 `x`，因为 `equal_to_x` 在 `x` 的作用域范围内。

对于函数来说，就算你把函数定义在 `main` 函数体中，它也不能访问 `x`：

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool {
        z == x
    }

    let y = 4;

    assert!(equal_to_x(y));
}
```

报错如下：

```
error[E0434]: can't capture dynamic environment in a fn item // 在函数中无法捕获动态的环境
--> src/main.rs:5:14
|
5 |         z == x
|         ^
|
= help: use the `|| { ... }` closure form instead // 使用闭包替代
```

如上所示，编译器准确地告诉了我们错误，甚至同时给出了提示：使用闭包来替代函数，这种聪明令我有些无所适从，总感觉会显得我很笨。

## 闭包对内存的影响

当闭包从环境中捕获一个值时，会分配内存去存储这些值。对于有些场景来说，这种额外的内存分配会成为一种负担。与之相比，函数就不会去捕获这些环境值，因此定义和使用函数不会拥有这种内存负担。

## 三种 Fn 特征

闭包捕获变量有三种途径，恰好对应函数参数的三种传入方式：转移所有权、可变借用、不可变借用，因此相应的 Fn 特征也有三种：

1. `FnOnce`，该类型的闭包会拿走被捕获变量的所有权。`Once` 顾名思义，说明该闭包只能运行一次：

```

fn fn_once<F>(func: F)
where
    F: FnOnce(usize) -> bool,
{
    println!("{}", func(3));
    println!("{}", func(4));
}

fn main() {
    let x = vec![1, 2, 3];
    fn_once(|z|{z == x.len()})
}

```

仅实现 FnOnce 特征的闭包在调用时会转移所有权，所以显然不能对已失去所有权的闭包变量进行二次调用：

```

error[E0382]: use of moved value: `func`
--> src\main.rs:6:20
|
1 | fn fn_once<F>(func: F)
|     ----- move occurs because `func` has type `F`, which does not
implement the `Copy` trait
|         // 因为`func`的类型是没有实现`Copy`特性的 `F`，所以发生了所有权的转移
...
5 |     println!("{}", func(3));
|         ----- `func` moved due to this call // 转移在这
6 |     println!("{}", func(4));
|         ^^^^ value used here after move // 转移后再次用
|

```

这里面有一个很重要的提示，因为 F 没有实现 Copy 特征，所以会报错，那么我们添加一个约束，试试实现了 Copy 的闭包：

```

fn fn_once<F>(func: F)
where
    F: FnOnce(usize) -> bool + Copy, // 改动在这里
{
    println!("{}", func(3));
    println!("{}", func(4));
}

fn main() {
    let x = vec![1, 2, 3];
    fn_once(|z|{z == x.len()})
}

```

上面代码中，func 的类型 F 实现了 copy 特征，调用时使用的将是它的拷贝，所以并没有发生所有权的转移。

```
true  
false
```

如果你想强制闭包取得捕获变量的所有权，可以在参数列表前添加 `move` 关键字，这种用法通常用于闭包的生命周期大于捕获变量的生命周期时，例如将闭包返回或移入其他线程。

```
use std::thread;  
let v = vec![1, 2, 3];  
let handle = thread::spawn(move || {  
    println!("Here's a vector: {:?}", v);  
});  
handle.join().unwrap();
```

2. `FnMut`，它以可变借用的方式捕获了环境中的值，因此可以修改该值：

```
fn main() {  
    let mut s = String::new();  
  
    let update_string = |str| s.push_str(str);  
    update_string("hello");  
  
    println!("{:?}", s);  
}
```

在闭包中，我们调用 `s.push_str` 去改变外部 `s` 的字符串值，因此这里捕获了它的可变借用，运行下试试：

```
error[E0596]: cannot borrow `update_string` as mutable, as it is not declared as  
mutable  
--> src/main.rs:5:5  
|  
4 |     let update_string = |str| s.push_str(str);  
|     -----  
|         - calling `update_string` requires mutable binding  
due to mutable borrow of `s`  
|         |  
|         help: consider changing this to be mutable: `mut update_string`  
5 |     update_string("hello");  
|     ^^^^^^^^^^^^^^ cannot borrow as mutable
```

虽然报错了，但是编译器给出了非常清晰的提示，想要在闭包内部捕获可变借用，需要把该闭包声明为可变类型，也就是 `update_string` 要修改为 `mut update_string`：

```

fn main() {
    let mut s = String::new();

    let mut update_string = |str| s.push_str(str);
    update_string("hello");

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

```

这种写法有点反直觉，相比起来前面的 move 更符合使用和阅读习惯。但是如果你忽略 update\_string 的类型，仅仅把它当成一个普通变量，那么这种声明就比较合理了。

再来看一个复杂点的：

```

fn main() {
    let mut s = String::new();

    let update_string = |str| s.push_str(str);

    exec(update_string);

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

fn exec<'a, F: FnMut(&'a str)>(mut f: F) {
    f("hello")
}

```

这段代码中 update\_string 没有使用 mut 关键字修饰，而上文提到想要在闭包内部捕获可变借用，需要用关键词把该闭包声明为可变类型。我们确实这么做了—— exec(mut f: F) 表明我们的 exec 接收的是一个可变类型的闭包。这段代码中 update\_string 看似被声明为不可变闭包，但是 exec(mut f: F) 函数接收的又是可变参数，为什么可以正常执行呢？

rust 不可能接受类型不匹配的形参和实参通过编译，我们提供的实参又是可变的，这说明 update\_string 一定是一个可变类型的闭包，我们不妨看看 rust-analyzer 自动给出的类型标注：

```

let mut s: String = String::new();

let update_string: impl FnMut(&str) = |str| s.push_str(str);

```

rust-analyzer 给出的类型标注非常清晰的说明了 update\_string 实现了 FnMut 特征。

为什么 update\_string 没有用 mut 修饰却是一个可变类型的闭包？事实上，FnMut 只是 trait 的名字，声明变量为 FnMut 和要不要 mut 没啥关系，FnMut 是推导出的特征类型，mut 是 rust 语言层面的一个修饰符，用于声明一个绑定是可变的。Rust 从特征类型系统和语言修饰符两方面保障了我们的程序正确运行。

我们在使用 FnMut 类型闭包时需要捕获外界的可变借用，因此我们常常搭配 mut 修饰符使用。但我们要始终记住，二者是相互独立的。

因此，让我们再回头分析一下这段代码：在 main 函数中，首先创建了一个可变的字符串 s，然后定义了一个可变类型闭包 update\_string，该闭包接受一个字符串参数并将其追加到 s 中。接下来调用了 exec 函数，并将 update\_string 闭包的所有权移交给它。最后打印出了字符串 s 的内容。

细心的读者可能注意到，我们在上文的分析中提到 update\_string 闭包的所有权被移交给 exec 函数。这说明 update\_string 没有实现 Copy 特征，但并不是所有闭包都没有实现 Copy 特征，闭包自动实现 Copy 特征的规则是，只要闭包捕获的类型都实现了 Copy 特征的话，这个闭包就会默认实现 Copy 特征。

我们来看一个例子：

```
let s = String::new();
let update_string = || println!("{}", s);
```

这里取得的是 s 的不可变引用，所以是能 Copy 的。而如果拿到的是 s 的所有权或可变引用，都是不能 Copy 的。我们刚刚的代码就属于第二类，取得的是 s 的可变引用，没有实现 Copy。

```
// 拿所有权
let s = String::new();
let update_string = move || println!("{}", s);

exec(update_string);
// exec2(update_string); // 不能再用了

// 可变引用
let mut s = String::new();
let mut update_string = || s.push_str("hello");
exec(update_string);
// exec1(update_string); // 不能再用了
```

3. Fn 特征，它以不可变借用的方式捕获环境中的值 让我们把上面的代码中 exec 的 F 泛型参数类型修改为 Fn(&'a str)：

```

fn main() {
    let mut s = String::new();

    let update_string = |str| s.push_str(str);

    exec(update_string);

    println!("{}:{}", s);
}

fn exec<'a, F: Fn(&'a str)>(mut f: F) {
    f("hello")
}

```

然后运行看看结果：

```

error[E0525]: expected a closure that implements the `Fn` trait, but this closure
only implements `FnMut`
--> src/main.rs:4:26 // 期望闭包实现的是`Fn`特征，但是它只实现了`FnMut`特征
|
4 |     let update_string = |str| s.push_str(str);
|     ^^^^^^--^^^^^--^^^^^--^^^^^--^^^^^--^
|             |           |
|             closure is `FnMut` because it mutates the variable
`s` here
|           this closure implements `FnMut`, not `Fn` //闭包实现的是
FnMut, 而不是Fn
5 |
6 |     exec(update_string);
|     ---- the requirement to implement `Fn` derives from here

```

从报错中很清晰的看出，我们的闭包实现的是 `FnMut` 特征，需要的是可变借用，但是在 `exec` 中却给它标注了 `Fn` 特征，因此产生了不匹配，再来看看正确的不可变借用方式：

```

fn main() {
    let s = "hello, ".to_string();

    let update_string = |str| println!("{}:{}", s, str);

    exec(update_string);

    println!("{}:{}", s);
}

fn exec<'a, F: Fn(String) -> ()>(f: F) {
    f("world".to_string())
}

```

在这里，因为无需改变 `s`，因此闭包中只对 `s` 进行了不可变借用，那么在 `exec` 中，将其标记为 `Fn` 特征就完全正确。

## move 和 Fn

在上面，我们讲到了 `move` 关键字对于 `FnOnce` 特征的重要性，但是实际上使用了 `move` 的闭包依然可能实现了 `Fn` 或 `FnMut` 特征。

因为，一个闭包实现了哪种 `Fn` 特征取决于该闭包如何使用被捕获的变量，而不是取决于闭包如何捕获它们。`move` 本身强调的就是后者，闭包如何捕获变量：

```
fn main() {
    let s = String::new();

    let update_string = move || println!("{}", s);

    exec(update_string);
}

fn exec<F: FnOnce()>(f: F) {
    f()
}
```

我们在上面的闭包中使用了 `move` 关键字，所以我们的闭包捕获了它，但是由于闭包对 `s` 的使用仅仅是不可变借用，因此该闭包实际上还实现了 `Fn` 特征。

细心的读者肯定发现我在上段中使用了一个 `还` 字，这是什么意思呢？因为该闭包不仅仅实现了 `FnOnce` 特征，还实现了 `Fn` 特征，将代码修改成下面这样，依然可以编译：

```
fn main() {
    let s = String::new();

    let update_string = move || println!("{}", s);

    exec(update_string);
}

fn exec<F: Fn()>(f: F) {
    f()
}
```

## 三种 Fn 的关系

实际上，一个闭包并不仅仅实现某一种 `Fn` 特征，规则如下：

- 所有的闭包都自动实现了 FnOnce 特征，因此任何一个闭包都至少可以被调用一次
- 没有移出所捕获变量的所有权的闭包自动实现了 FnMut 特征
- 不需要对捕获变量进行改变的闭包自动实现了 Fn 特征

用一段代码来简单诠释上述规则：

```
fn main() {
    let s = String::new();

    let update_string = || println!("{}", s);

    exec(update_string);
    exec1(update_string);
    exec2(update_string);
}

fn exec<F: FnOnce()>(f: F) {
    f()
}

fn exec1<F: FnMut()>(mut f: F) {
    f()
}

fn exec2<F: Fn()>(f: F) {
    f()
}
```

虽然，闭包只是对 s 进行了不可变借用，实际上，它可以适用于任何一种 Fn 特征：三个 exec 函数说明了一切。强烈建议读者亲自动手试试各种情况下使用的 Fn 特征，更有助于加深这方面的理解。

关于第二条规则，有如下示例：

```
fn main() {
    let mut s = String::new();

    let update_string = |str| -> String {s.push_str(str); s};

    exec(update_string);
}

fn exec<'a, F: FnMut(&'a str) -> String>(mut f: F) {
    f("hello");
}
```

```

5 |     let update_string = |str| -> String {s.push_str(str); s };
|           ^^^^^^^^^^
|           because it moves the variable `s` out of its environment
|           - closure is `FnOnce`
|           // 闭包实现了`FnOnce`，因
|           为它从捕获环境中移出了变量`s`
|           |
|           this closure implements `FnOnce`， not `FnMut`
```

此例中，闭包从捕获环境中移出了变量 `s` 的所有权，因此这个闭包仅自动实现了 `FnOnce`，未实现 `FnMut` 和 `Fn`。再次印证之前讲的一个闭包实现了哪种 `Fn` 特征取决于该闭包如何使用被捕获的变量，而不是取决于闭包如何捕获它们，跟是否使用 `move` 没有必然联系。

如果还是有疑惑？没关系，我们来看看这三个特征的简化版源码：

```

pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&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 FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

看到没？从特征约束能看出来 `Fn` 的前提是实现 `FnMut`，`FnMut` 的前提是实现 `FnOnce`，因此要实现 `Fn` 就要同时实现 `FnMut` 和 `FnOnce`，这段源码从侧面印证了之前规则的正确性。

从源码中还能看出一点：`Fn` 获取 `&self`，`FnMut` 获取 `&mut self`，而 `FnOnce` 获取 `self`。在实际项目中，建议先使用 `Fn` 特征，然后编译器会告诉你正误以及该如何选择。

## 闭包作为函数返回值

看到这里，相信大家对于如何使用闭包作为函数参数，已经很熟悉了，但是如果要使用闭包作为函数返回值，该如何做？

先来看一段代码：

```

fn factory() -> Fn(i32) -> i32 {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);

```

上面这段代码看起来还是蛮正常的，用 `Fn(i32) -> i32` 特征来代表 `|x| x + num`，非常合理嘛，肯定可以编译通过，可惜理想总是难以照进现实，编译器给我们报了一大堆错误，先挑几个重点来看看：

```

fn factory<T>() -> Fn(i32) -> i32 {
    |                                     ^^^^^^^^^^^^^^ doesn't have a size known at compile-time // 该
类型在编译器没有固定的大小

```

Rust 要求函数的参数和返回类型，必须有固定的内存大小，例如 `i32` 就是 4 个字节，引用类型是 8 个字节，总之，绝大部分类型都有固定的大小，但是不包括特征，因为特征类似接口，对于编译器来说，无法知道它后面藏的真实类型是什么，因为也无法得知具体的大小。

同样，我们也无法知道闭包的具体类型，该怎么办呢？再看看报错提示：

```

help: use `impl Fn(i32) -> i32` as the return type, as all return paths are of type
`[closure@src/main.rs:11:5: 11:21]`, which implements `Fn(i32) -> i32`
|
8 | fn factory<T>() -> impl Fn(i32) -> i32 {

```

嗯，编译器提示我们加一个 `impl` 关键字，哦，这样一看，读者可能就想起来了，`impl Trait` 可以用来返回一个实现了指定特征的类型，那么这里 `impl Fn(i32) -> i32` 的返回值形式，说明我们要返回一个闭包类型，它实现了 `Fn(i32) -> i32` 特征。

完美解决，但是，在[特征那一章](#)，我们提到过，`impl Trait` 的返回方式有一个非常大的局限，就是你只能返回同样的类型，例如：

```

fn factory(x:i32) -> impl Fn(i32) -> i32 {
    let num = 5;

    if x > 1{
        move |x| x + num
    } else {
        move |x| x - num
    }
}

```

运行后，编译器报错：

```
error[E0308]: `if` and `else` have incompatible types
--> src/main.rs:15:9
12 |     if x > 1{
13 |         move |x| x + num
14 |             ----- expected because of this
15 |     } else {
16 |         move |x| x - num
|             ^^^^^^^^^^^^^^^^^ expected closure, found a different closure
|     }
|----- `if` and `else` have incompatible types
```

嗯，提示很清晰：`if` 和 `else` 分支中返回了不同的闭包类型，这就很奇怪了，明明这两个闭包长的一样的，好在细心的读者应该回想起来，本章节前面咱们有提到：就算签名一样的闭包，类型也是不同的，因此在这种情况下，就无法再使用 `impl Trait` 的方式去返回闭包。

怎么办？再看看编译器提示，里面有这样一行小字：

```
= help: consider boxing your closure and/or using it as a trait object
```

哦，相信你已经恍然大悟，可以用特征对象！只需要用 `Box` 的方式即可实现：

```
fn factory(x:i32) -> Box<dyn Fn(i32) -> i32> {
    let num = 5;

    if x > 1{
        Box::new(move |x| x + num)
    } else {
        Box::new(move |x| x - num)
    }
}
```

至此，闭包作为函数返回值就已完美解决，若以后你再遇到报错时，一定要仔细阅读编译器的提示，很多时候，转角都能遇到爱。

## 闭包的生命周期

这块儿内容在进阶生命周期章节中有讲，这里就不再赘述，读者可移步[此处](#)进行回顾。

## 课后习题

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

---

# 迭代器 Iterator

如果你询问一个 Rust 资深开发：写 Rust 项目最需要掌握什么？相信迭代器往往就是答案之一。无论你是编程新手亦或是高手，实际上大概率都用过迭代器，虽然自己可能并没有意识到这一点。)

迭代器允许我们迭代一个连续的集合，例如数组、动态数组 `Vec`、`HashMap` 等，在此过程中，只需关心集合中的元素如何处理，而无需关心如何开始、如何结束、按照什么样的索引去访问等问题。

## For 循环与迭代器

从用途来看，迭代器跟 `for` 循环颇为相似，都是去遍历一个集合，但是实际上它们存在不小的差别，其中最主要的差别就是：**是否通过索引来访问集合**。

例如以下的 JS 代码就是一个循环：

```
let arr = [1, 2, 3];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

在上面代码中，我们设置索引的开始点和结束点，然后再通过索引去访问元素 `arr[i]`，这就是典型的循环，来对比下 Rust 中的 `for`：

```
let arr = [1, 2, 3];
for v in arr {
    println!("{}" ,v);
}
```

首先，不得不说这两语法还挺像！与 JS 循环不同，Rust 中没有使用索引，它把 `arr` 数组当成一个迭代器，直接去遍历其中的元素，从哪里开始，从哪里结束，都无需操心。因此严格来说，Rust 中的 `for` 循环是编译器提供的语法糖，最终还是对迭代器中的元素进行遍历。

那又有同学要发问了，在 Rust 中数组是迭代器吗？因为在之前的代码中直接对数组 `arr` 进行了迭代，答案是 No。既然数组不是迭代器，为啥咱可以对它的元素进行迭代呢？

简而言之就是数组实现了 `IntoIterator` 特征，Rust 通过 `for` 语法糖，自动把实现了该特征的数组类型转换为迭代器（你也可以为自己的集合类型实现此特征），最终让我们可以直接对一个数组进行迭代，类似的还有：

```
for i in 1..10 {  
    println!("{}", i);  
}
```

直接对数值序列进行迭代，也是很常见的使用方式。

`IntoIterator` 特征拥有一个 `into_iter` 方法，因此我们还可以显式的把数组转换成迭代器：

```
let arr = [1, 2, 3];  
for v in arr.into_iter() {  
    println!("{}", v);  
}
```

迭代器是函数语言的核心特性，它赋予了 Rust 远超于循环的强大表达能力，我们将在本章中一一为大家进行展现。

## 惰性初始化

在 Rust 中，迭代器是惰性的，意味着如果你不使用它，那么它将不会发生任何事：

```
let v1 = vec![1, 2, 3];  
  
let v1_iter = v1.iter();  
  
for val in v1_iter {  
    println!("{}", val);  
}
```

在 `for` 循环之前，我们只是简单的创建了一个迭代器 `v1_iter`，此时不会发生任何迭代行为，只有在 `for` 循环开始后，迭代器才会开始迭代其中的元素，最后打印出来。

这种惰性初始化的方式确保了创建迭代器不会有额外的性能损耗，其中的元素也不会被消耗，只有使用到该迭代器的时候，一切才开始。

## next 方法

对于 `for` 如何遍历迭代器，还有一个问题，它如何取出迭代器中的元素？

先来看一个特征：

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // 省略其余有默认实现的方法  
}
```

呦，该特征竟然和迭代器 `iterator` 同名，难不成。。。没错，它们就是有一腿。**迭代器之所以成为迭代器，就是因为实现了 Iterator 特征**，要实现该特征，最主要的就是实现其中的 `next` 方法，该方法控制如何从集合中取值，最终返回值的类型是[关联类型 Item](#)。

因此，之前问题的答案已经很明显：`for` 循环通过不停调用迭代器上的 `next` 方法，来获取迭代器中的元素。

既然 `for` 可以调用 `next` 方法，是不是意味着我们也可以？来试试：

```
fn main() {  
    let arr = [1, 2, 3];  
    let mut arr_iter = arr.into_iter();  
  
    assert_eq!(arr_iter.next(), Some(1));  
    assert_eq!(arr_iter.next(), Some(2));  
    assert_eq!(arr_iter.next(), Some(3));  
    assert_eq!(arr_iter.next(), None);  
}
```

果不其然，将 `arr` 转换成迭代器后，通过调用其上的 `next` 方法，我们获取了 `arr` 中的元素，有两点需要注意：

- `next` 方法返回的是 `Option` 类型，当有值时返回 `Some(i32)`，无值时返回 `None`
- 遍历是按照迭代器中元素的排列顺序依次进行的，因此我们严格按照数组中元素的顺序取出了 `Some(1)`，`Some(2)`，`Some(3)`
- 手动迭代必须将迭代器声明为 `mut` 可变，因为调用 `next` 会改变迭代器其中的状态数据（当前遍历的位置等），而 `for` 循环去迭代则无需标注 `mut`，因为它会帮我们自动完成

总之，`next` 方法对**迭代器的遍历是消耗性的**，每次消耗它一个元素，最终迭代器中将没有任何元素，只能返回 `None`。

### 例子：模拟实现 for 循环

因为 `for` 循环是迭代器的语法糖，因此我们完全可以通过迭代器来模拟实现它：

```

let values = vec![1, 2, 3];

{
    let result = match IntoIterator::into_iter(values) {
        mut iter => loop {
            match iter.next() {
                Some(x) => { println!("{}", x); },
                None => break,
            }
        },
    };
    result
}

```

`IntoIterator::into_iter` 是使用[完全限定](#)的方式去调用 `into_iter` 方法，这种调用方式跟 `values.into_iter()` 是等价的。

同时我们使用了 `loop` 循环配合 `next` 方法来遍历迭代器中的元素，当迭代器返回 `None` 时，跳出循环。

## IntoIterator 特征

其实有一个细节，由于 `Vec` 动态数组实现了 `IntoIterator` 特征，因此可以通过 `into_iter` 将其转换为迭代器，那如果本身就是一个迭代器，该怎么办？实际上，迭代器自身也实现了 `IntoIterator`，标准库早就帮我们考虑好了：

```

impl<I: Iterator> IntoIterator for I {
    type Item = I::Item;
    type IntoIter = I;

    #[inline]
    fn into_iter(self) -> I {
        self
    }
}

```

最终你完全可以写出这样的奇怪代码：

```
fn main() {
    let values = vec![1, 2, 3];

    for v in values.into_iter().into_iter().into_iter() {
        println!("{}", v)
    }
}
```

### **into\_iter, iter, iter\_mut**

在之前的代码中，我们统一使用了 `into_iter` 的方式将数组转化为迭代器，除此之外，还有 `iter` 和 `iter_mut`，聪明的读者应该大概能猜到这三者的区别：

- `into_iter` 会夺走所有权
- `iter` 是借用
- `iter_mut` 是可变借用

其实如果以后见多识广了，你会发现这种问题一眼就能看穿，`into_` 之类的，都是拿走所有权，`_mut` 之类的都是可变借用，剩下的就是不可变借用。

使用一段代码来解释下：

```

fn main() {
    let values = vec![1, 2, 3];

    for v in values.into_iter() {
        println!("{}", v)
    }

    // 下面的代码将报错，因为 values 的所有权在上面 `for` 循环中已经被转移走
    // println!("{:?}", values);

    let values = vec![1, 2, 3];
    let _values_iter = values.iter();

    // 不会报错，因为 values_iter 只是借用了 values 中的元素
    println!("{:?}", values);

    let mut values = vec![1, 2, 3];
    // 对 values 中的元素进行可变借用
    let mut values_iter_mut = values.iter_mut();

    // 取出第一个元素，并修改为0
    if let Some(v) = values_iter_mut.next() {
        *v = 0;
    }

    // 输出[0, 2, 3]
    println!("{:?}", values);
}

```

具体解释在代码注释中，就不再赘述，不过有两点需要注意的是：

- `.iter()` 方法实现的迭代器，调用 `next` 方法返回的类型是 `Some(&T)`
- `.iter_mut()` 方法实现的迭代器，调用 `next` 方法返回的类型是 `Some(&mut T)`，因此在 `if let Some(v) = values_iter_mut.next()` 中，`v` 的类型是 `&mut i32`，最终我们可以通过 `*v = 0` 的方式修改其值

## Iterator 和 IntoIterator 的区别

这两个其实还蛮容易搞混的，但我们只需要记住，`Iterator` 就是迭代器特征，只有实现了它才能称为迭代器，才能调用 `next`。

而 `IntoIterator` 强调的是某一个类型如果实现了该特征，它可以通过 `into_iter`, `iter` 等方法变成一个迭代器。

# 消费者与适配器

消费者是迭代器上的方法，它会消费掉迭代器中的元素，然后返回其类型的值，这些消费者都有一个共同的特点：在它们的定义中，都依赖 `next` 方法来消费元素，因此这也是为什么迭代器要实现 `Iterator` 特征，而该特征必须要实现 `next` 方法的原因。

## 消费者适配器

只要迭代器上的某个方法 `A` 在其内部调用了 `next` 方法，那么 `A` 就被称为**消费性适配器**：因为 `next` 方法会消耗掉迭代器上的元素，所以方法 `A` 的调用也会消耗掉迭代器上的元素。

其中一个例子是 `sum` 方法，它会拿走迭代器的所有权，然后通过不断调用 `next` 方法对里面的元素进行求和：

```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);

    // v1_iter 是借用了 v1，因此 v1 可以照常使用
    println!("{:?}", v1);

    // 以下代码会报错，因为 `sum` 拿到了迭代器 `v1_iter` 的所有权
    // println!("{:?}", v1_iter);
}
```

如代码注释中所说明的：在使用 `sum` 方法后，我们将无法再使用 `v1_iter`，因为 `sum` 拿走了该迭代器的所有权：

```
fn sum<S>(self) -> S
    where
        Self: Sized,
        S: Sum<Self::Item>,
    {
        Sum::sum(self)
    }
```

从 `sum` 源码中也可以清晰看出，`self` 类型的方法参数拿走了所有权。

## 迭代器适配器

既然消费者适配器是消费掉迭代器，然后返回一个值。那么迭代器适配器，顾名思义，会返回一个新的迭代器，这是实现链式方法调用的关键：`v.iter().map().filter()...`。

与消费者适配器不同，迭代器适配器是惰性的，意味着你需要一个消费者适配器来收尾，最终将迭代器转换成一个具体的值：

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

运行后输出：

```
warning: unused `Map` that must be used
--> src/main.rs:4:5
|
4 |     v1.iter().map(|x| x + 1);
|     ^^^^^^^^^^^^^^^^^^
|
= note: `#[warn(unused_must_use)]` on by default
= note: iterators are lazy and do nothing unless consumed // 迭代器 map 是惰性的，这里
不产生任何效果
```

如上述中文注释所说，这里的 `map` 方法是一个迭代者适配器，它是惰性的，不产生任何行为，因此我们还需要一个消费者适配器进行收尾：

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

## collect

上面代码中，使用了 `collect` 方法，该方法就是一个消费者适配器，使用它可以将一个迭代器中的元素收集到指定类型中，这里我们为 `v2` 标注了 `Vec<_>` 类型，就是为了告诉 `collect`：请把迭代器中的元素消费掉，然后把值收集成 `Vec<_>` 类型，至于为何使用 `_`，因为编译器会帮我们自动推导。

为何 `collect` 在消费时要指定类型？是因为该方法其实很强大，可以收集成多种不同的集合类型，`Vec<T>` 仅仅是其中之一，因此我们必须显式的告诉编译器我们想要收集成的集合类型。

还有一点值得注意，`map` 会对迭代器中的每一个值进行一系列操作，然后把该值转换成另外一个新值，该操作是通过闭包 `|x| x + 1` 来完成：最终迭代器中的每个值都增加了 1，从 `[1, 2, 3]` 变为 `[2,`

3, 4]。

再来看看如何使用 `collect` 收集成 `HashMap` 集合:

```
use std::collections::HashMap;
fn main() {
    let names = ["sunface", "sunfei"];
    let ages = [18, 18];
    let folks: HashMap<_, _> = names.into_iter().zip(ages.into_iter()).collect();

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

`zip` 是一个迭代器适配器，它的作用就是将两个迭代器的内容压缩到一起，形成 `Iterator<Item=(ValueFromA, ValueFromB)>` 这样的新的迭代器，在此处就是形如 `[(name1, age1), (name2, age2)]` 的迭代器。

然后再通过 `collect` 将新迭代器中 `(K, V)` 形式的值收集成 `HashMap<K, V>`，同样的，这里必须显式声明类型，然后 `HashMap` 内部的 `KV` 类型可以交给编译器去推导，最终编译器会推导出 `HashMap<&str, i32>`，完全正确！

## 闭包作为适配器参数

之前的 `map` 方法中，我们使用闭包来作为迭代器适配器的参数，它最大的好处不仅在于可以就地实现迭代器中元素的处理，还在于可以捕获环境值：

```
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

`filter` 是迭代器适配器，用于对迭代器中的每个值进行过滤。它使用闭包作为参数，该闭包的参数 `s` 是来自迭代器中的值，然后使用 `s` 跟外部环境中的 `shoe_size` 进行比较，若相等，则在迭代器中保留 `s` 值，若不相等，则从迭代器中剔除 `s` 值，最终通过 `collect` 收集为 `Vec<Shoe>` 类型。

## 实现 Iterator 特征

之前的内容我们一直基于数组来创建迭代器，实际上，不仅仅是数组，基于其它集合类型一样可以创建迭代器，例如 `HashMap`。你也可以创建自己的迭代器——只要为自定义类型实现 `Iterator` 特征即可。

首先，创建一个计数器：

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

我们为计数器 `Counter` 实现了一个关联函数 `new`，用于创建新的计数器实例。下面我们继续为计数器实现 `Iterator` 特征：

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}
```

首先，将该特征的关联类型设置为 `u32`，由于我们的计数器保存的 `count` 字段就是 `u32` 类型，因此在 `next` 方法中，最后返回的是实际上是 `Option<u32>` 类型。

每次调用 `next` 方法，都会让计数器的值加一，然后返回最新的计数值，一旦计数大于 5，就返回 `None`。

最后，使用我们新建的 `Counter` 进行迭代：

```
let mut counter = Counter::new();

assert_eq!(counter.next(), Some(1));
assert_eq!(counter.next(), Some(2));
assert_eq!(counter.next(), Some(3));
assert_eq!(counter.next(), Some(4));
assert_eq!(counter.next(), Some(5));
assert_eq!(counter.next(), None);
```

## 实现 Iterator 特征的其它方法

可以看出，实现自己的迭代器非常简单，但是 Iterator 特征中，不仅仅是只有 `next` 一个方法，那为什么我们只需要实现它呢？因为其它方法都具有默认实现，所以无需像 `next` 这样手动去实现，而且这些默认实现的方法其实都是基于 `next` 方法实现的。

下面的代码演示了部分方法的使用：

```
let sum: u32 = Counter::new()
    .zip(Counter::new().skip(1))
    .map(|(a, b)| a * b)
    .filter(|x| x % 3 == 0)
    .sum();
assert_eq!(18, sum);
```

其中 `zip`，`map`，`filter` 是迭代器适配器：

- `zip` 把两个迭代器合并成一个迭代器，新迭代器中，每个元素都是一个元组，由之前两个迭代器的元素组成。例如将形如 `[1, 2, 3, 4, 5]` 和 `[2, 3, 4, 5]` 的迭代器合并后，新的迭代器形如 `[(1, 2), (2, 3), (3, 4), (4, 5)]`
- `map` 是将迭代器中的值经过映射后，转换成新的值`[2, 6, 12, 20]`
- `filter` 对迭代器中的元素进行过滤，若闭包返回 `true` 则保留元素`[6, 12]`，反之剔除

而 `sum` 是消费者适配器，对迭代器中的所有元素求和，最终返回一个 `u32` 值 `18`。

## enumerate

在之前的流程控制章节，针对 `for` 循环，我们提供了一种方法可以获取迭代时的索引：

```
let v = vec![1u64, 2, 3, 4, 5, 6];
for (i, v) in v.iter().enumerate() {
    println!("第{}个值是{}", i, v)
}
```

相信当时，很多读者还是很迷茫的，不知道为什么要这么复杂才能获取到索引，学习本章节后，相信你有了全新的理解，首先 `v.iter()` 创建迭代器，其次 调用 `Iterator` 特征上的方法 `enumerate`，该方法产生一个新的迭代器，其中每个元素均是元组（索引，值）。

因为 `enumerate` 是迭代器适配器，因此我们可以对它返回的迭代器调用其它 `Iterator` 特征方法：

```
let v = vec![1u64, 2, 3, 4, 5, 6];
let val = v.iter()
    .enumerate()
    // 每两个元素剔除一个
    // [1, 3, 5]
    .filter(|&(idx, _)| idx % 2 == 0)
    .map(|(_, val)| val)
    // 累加 1+3+5 = 9
    .fold(0u64, |sum, acm| sum + acm);

println!("{}", val);
```

## 迭代器的性能

前面提到，要完成集合遍历，既可以使用 `for` 循环也可以使用迭代器，那么二者之间该怎么选择呢，性能有多大差距呢？

理论分析不会有结果，直接测试最为靠谱：

```

#![feature(test)]

extern crate rand;
extern crate test;

fn sum_for(x: &[f64]) -> f64 {
    let mut result: f64 = 0.0;
    for i in 0..x.len() {
        result += x[i];
    }
    result
}

fn sum_iter(x: &[f64]) -> f64 {
    x.iter().sum::<f64>()
}

#[cfg(test)]
mod bench {
    use test::Bencher;
    use rand::{Rng, thread_rng};
    use super::*;

    const LEN: usize = 1024*1024;

    fn rand_array(cnt: u32) -> Vec<f64> {
        let mut rng = thread_rng();
        (0..cnt).map(|_| rng.gen::<f64>()).collect()
    }

    #[bench]
    fn bench_for(b: &mut Bencher) {
        let samples = rand_array(LEN as u32);
        b.iter(|| {
            sum_for(&samples)
        })
    }

    #[bench]
    fn bench_iter(b: &mut Bencher) {
        let samples = rand_array(LEN as u32);
        b.iter(|| {
            sum_iter(&samples)
        })
    }
}

```

上面的代码对比了 `for` 循环和迭代器 `iterator` 完成同样的求和任务的性能对比，可以看到迭代器还要更快一点。

```
test bench::bench_for ... bench:      998,331 ns/iter (+/- 36,250)
test bench::bench_iter ... bench:     983,858 ns/iter (+/- 44,673)
```

迭代器是 Rust 的 **零成本抽象** (zero-cost abstractions) 之一，意味着抽象并不会引入运行时开销，这与 Bjarne Stroustrup (C++ 的设计和实现者) 在 *Foundations of C++* (2012) 中所定义的 **零开销** (zero-overhead) 如出一辙：

---

In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

一般来说，C++的实现遵循零开销原则：没有使用时，你不必为其买单。更进一步说，需要使用时，你也无法写出更优的代码了。（翻译一下：用就完事了）

总之，迭代器是 Rust 受函数式语言启发而提供的高级语言特性，可以写出更加简洁、逻辑清晰的代码。编译器还可以通过循环展开 (Unrolling) 、向量化、消除边界检查等优化手段，使得迭代器和 `for` 循环都有极为高效的执行效率。

所以请放心大胆的使用迭代器，在获得更高的表达力的同时，也不会导致运行时的损失，何乐而不为呢！

## 学习其它方法

迭代器用的好不好，就在于你是否掌握了它的常用方法，且能活学活用，因此多多看看[标准库](#)是有好处的，只有知道有什么方法，在需要的时候你才能知道该用什么，就和算法学习一样。

同时，本书在后续章节还提供了对迭代器常用方法的[深入讲解](#)，方便大家学习和查阅。

# 深入类型

Rust 是强类型语言，同时也是强安全语言，这些特性导致了 Rust 的类型注定比一般语言要更深入也更困难。

本章将深入讲解一些进阶的 Rust 类型以及类型转换，希望大家喜欢。

# 类型转换

Rust 是类型安全的语言，因此在 Rust 中做类型转换不是一件简单的事，这一章节我们将对 Rust 中的类型转换进行详尽讲解。

---

高能预警：本章节有些难，可以考虑学了进阶后回头再看

---

## as转换

先来看一段代码：

```
fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    if a < b {
        println!("Ten is less than one hundred.");
    }
}
```

能跟着这本书一直学习到这里，说明你对 Rust 已经有了一定的理解，那么一眼就能看出这段代码注定会报错，因为 `a` 和 `b` 拥有不同的类型，Rust 不允许两种不同的类型进行比较。

解决办法很简单，只要把 `b` 转换成 `i32` 类型即可，Rust 中内置了一些基本类型之间的转换，这里使用 `as` 操作符来完成：`if a < (b as i32) {...}`。那么为什么不把 `a` 转换成 `u16` 类型呢？

因为每个类型能表达的数据范围不同，如果把范围较大的类型转换成较小的类型，会造成错误，因此我们需要把范围较小的类型转换成较大的类型，来避免这些问题的发生。

---

使用类型转换需要小心，因为如果执行以下操作 `300_i32 as i8`，你将获得 `44` 这个值，而不是 `300`，因为 `i8` 类型能表达的最大值为  $2^7 - 1$ ，使用以下代码可以查看 `i8` 的最大值：

---

```
let a = i8::MAX;
println!("{}", a);
```

下面列出了常用的转换形式：

```
fn main() {
    let a = 3.1 as i8;
    let b = 100_i8 as i32;
    let c = 'a' as u8; // 将字符'a'转换为整数, 97

    println!("{} , {} , {}", a, b, c)
}
```

## 内存地址转换为指针

```
let mut values: [i32; 2] = [1, 2];
let p1: *mut i32 = values.as_mut_ptr();
let first_address = p1 as usize; // 将p1内存地址转换为一个整数
let second_address = first_address + 4; // 4 == std::mem::size_of::<i32>(), i32类型占用
// 4个字节, 因此将内存地址 + 4
let p2 = second_address as *mut i32; // 访问该地址指向的下一个整数p2
unsafe {
    *p2 += 1;
}
assert_eq!(values[1], 3);
```

## 强制类型转换的边角知识

1. 转换不具有传递性 就算 `e as U1 as U2` 是合法的，也不能说明 `e as U2` 是合法的（`e` 不能直接转换成 `U2`）。

## TryInto 转换

在一些场景中，使用 `as` 关键字会有比较大的限制。如果你想要在类型转换上拥有完全的控制而不依赖内置的转换，例如处理转换错误，那么可以使用 `TryInto`：

```

use std::convert::TryInto;

fn main() {
    let a: u8 = 10;
    let b: u16 = 1500;

    let b_: u8 = b.try_into().unwrap();

    if a < b_ {
        println!("Ten is less than one hundred.");
    }
}

```

上面代码中引入了 `std::convert::TryInto` 特征，但是却没有使用它，可能有些同学会为此困惑，主要原因在于**如果你要使用一个特征的方法，那么你需要引入该特征到当前的作用域中**，我们在上面用到了 `try_into` 方法，因此需要引入对应的特征。但是 Rust 又提供了一个非常便利的办法，把最常用的标准库中的特征通过 `std::prelude` 模块提前引入到当前作用域中，其中包括了 `std::convert::TryInto`，你可以尝试删除第一行的代码 `use ...`，看看是否会报错。

`try_into` 会尝试进行一次转换，并返回一个 `Result`，此时就可以对其进行相应的错误处理。由于我们的例子只是为了快速测试，因此使用了 `unwrap` 方法，该方法在发现错误时，会直接调用 `panic` 导致程序的崩溃退出，在实际项目中，请不要这么使用，具体见[panic](#)部分。

最主要的是 `try_into` 转换会捕获大类型向小类型转换时导致的溢出错误：

```

fn main() {
    let b: i16 = 1500;

    let b_: u8 = match b.try_into() {
        Ok(b1) => b1,
        Err(e) => {
            println!("{:?}", e.to_string());
            0
        }
    };
}

```

运行后输出如下 "out of range integral type conversion attempted"，在这里我们程序捕获了错误，编译器告诉我们类型范围超出的转换是不被允许的，因为我们试图把 `1500_i16` 转换为 `u8` 类型，后者明显不足以承载这么大的值。

## 通用类型转换

虽然 `as` 和 `TryInto` 很强大，但是只能应用在数值类型上，可是 Rust 有如此多的类型，想要为这些类型实现转换，我们需要另谋出路，先来看看在一个笨办法，将一个结构体转换为另外一个结构体：

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

简单粗暴，但是从另外一个角度来看，也挺啰嗦的，好在 Rust 为我们提供了更通用的方式来完成这个目的。

## 强制类型转换

在某些情况下，类型是可以进行隐式强制转换的，虽然这些转换弱化了 Rust 的类型系统，但是它们的存在是为了让 Rust 在大多数场景可以工作(说白了，帮助用户省事)，而不是报各种类型上的编译错误。

首先，在匹配特征时，不会做任何强制转换(除了方法)。一个类型 `T` 可以强制转换为 `U`，不代表 `impl T` 可以强制转换为 `impl U`，例如下面的代码就无法通过编译检查：

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}

fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

报错如下：

```
error[E0277]: the trait bound `&mut i32: Trait` is not satisfied
--> src/main.rs:9:9
|
9 |     foo(t);
|         ^ the trait `Trait` is not implemented for `&mut i32`
|
= help: the following implementations were found:
<&'a i32 as Trait>
= note: `Trait` is implemented for `&i32`, but not for `&mut i32`
```

&i32 实现了特征 Trait，&mut i32 可以转换为 &i32，但是 &mut i32 依然无法作为 Trait 来使用。

## 点操作符

方法调用的点操作符看起来简单，实际上非常不简单，它在调用时，会发生很多魔法般的类型转换，例如：自动引用、自动解引用，强制类型转换直到类型能匹配等。

假设有一个方法 foo，它有一个接收器(接收器就是 self、&self、&mut self 参数)。如果调用 value.foo()，编译器在调用 foo 之前，需要决定到底使用哪个 Self 类型来调用。现在假设 value 拥有类型 T。

再进一步，我们使用[完全限定语法](#)来进行准确的函数调用：

1. 首先，编译器检查它是否可以直接调用 T::foo(value)，称之为**值方法调用**
2. 如果上一步调用无法完成(例如方法类型错误或者特征没有针对 Self 进行实现，上文提到过特征不能进行强制转换)，那么编译器会尝试增加自动引用，例如会尝试以下调用：`<&T>::foo(value)` 和 `<&mut T>::foo(value)`，称之为**引用方法调用**
3. 若上面两个方法依然不工作，编译器会试着解引用 T，然后再进行尝试。这里使用了 Deref 特征——若 `T: Deref<Target = U>` (T 可以被解引用为 U)，那么编译器会使用 U 类型进行尝试，称之为**解引用方法调用**
4. 若 T 不能被解引用，且 T 是一个定长类型(在编译期类型长度是已知的)，那么编译器也会尝试将 T 从定长类型转为不定长类型，例如将 `[i32; 2]` 转为 `[i32]`
5. 若还是不行，那...没有那了，最后编译器大喊一声：汝欺我甚，不干了！

下面我们来用一个例子来解释上面的方法查找算法：

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

array 数组的底层数据隐藏在了重重封锁之后，那么编译器如何使用 array[0] 这种数组原生访问语法通过重重封锁，准确的访问到数组中的第一个元素？

- 首先，`array[0]` 只是 `Index` 特征的语法糖：编译器会将 `array[0]` 转换为 `array.index(0)` 调用，当然在调用之前，编译器会先检查 `array` 是否实现了 `Index` 特征。
- 接着，编译器检查 `Rc<Box<[T; 3]>>` 是否有实现 `Index` 特征，结果是否，不仅如此，`&Rc<Box<[T; 3]>>` 与 `&mut Rc<Box<[T; 3]>>` 也没有实现。
- 上面的都不能工作，编译器开始对 `Rc<Box<[T; 3]>>` 进行解引用，把它转变成 `Box<[T; 3]>`
- 此时继续对 `Box<[T; 3]>` 进行上面的操作：`Box<[T; 3]>`, `&Box<[T; 3]>`, 和 `&mut Box<[T; 3]>` 都没有实现 `Index` 特征，所以编译器开始对 `Box<[T; 3]>` 进行解引用，然后我们得到了 `[T; 3]`
- `[T; 3]` 以及它的各种引用都没有实现 `Index` 索引(是不是很反直觉:D，在直觉中，数组都可以通过索引访问，实际上只有数组切片才可以!)，它也不能再进行解引用，因此编译器只能祭出最后的大杀器：将定长转为不定长，因此 `[T; 3]` 被转换成 `[T]`，也就是数组切片，它实现了 `Index` 特征，因此最终我们可以通过 `index` 方法访问到对应的元素。

过程看起来很复杂，但是也还好，挺好理解，如果你现在不能彻底理解，也不要紧，等以后对 Rust 理解更深了，同时需要深入理解类型转换时，再来细细品读本章。

再来看看以下更复杂的例子：

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

上面例子中 `cloned` 的类型是什么？首先编译器检查能不能进行**值方法调用**，`value` 的类型是 `&T`，同时 `clone` 方法的签名也是 `&T : fn clone(&T) -> T`，因此可以进行值方法调用，再加上编译器知道了 `T` 实现了 `Clone`，因此 `cloned` 的类型是 `T`。

如果 `T: Clone` 的特征约束被移除呢？

```
fn do_stuff<T>(value: &T) {
    let cloned = value.clone();
}
```

首先，从直觉上来说，该方法会报错，因为 `T` 没有实现 `Clone` 特征，但是真实情况是什么呢？

我们先来推导一番。首先通过值方法调用就不再可行，因为 `T` 没有实现 `Clone` 特征，也就无法调用 `T` 的 `clone` 方法。接着编译器尝试**引用方法调用**，此时 `T` 变成 `&T`，在这种情况下，`clone` 方法的签名如下：`fn clone(&&T) -> &T`，接着我们现在对 `value` 进行了引用。编译器发现 `&T` 实现了 `Clone` 类型(所有的引用类型都可以被复制，因为其实质就是复制一份地址)，因此可以推出 `cloned` 也是 `&T` 类型。

最终，我们复制出一份引用指针，这很合理，因为值类型 `T` 没有实现 `Clone`，只能去复制一个指针了。

下面的例子也是自动引用生效的地方：

```

#[derive(Clone)]
struct Container<T>(Arc<T>);

fn clone_containers<T>(foo: &Container<i32>, bar: &Container<T>) {
    let foo_cloned = foo.clone();
    let bar_cloned = bar.clone();
}

```

推断下上面的 `foo_cloned` 和 `bar_cloned` 是什么类型？提示：关键在 `Container` 的泛型参数，一个是 `i32` 的具体类型，一个是泛型类型，其中 `i32` 实现了 `Clone`，但是 `T` 并没有。

首先要复习一下复杂类型派生 `Clone` 的规则：一个复杂类型能否派生 `Clone`，需要它内部的所有子类型都能进行 `Clone`。因此 `Container<T>(Arc<T>)` 是否实现 `Clone` 的关键在于 `T` 类型是否实现了 `Clone` 特征。

上面代码中，`Container<i32>` 实现了 `Clone` 特征，因此编译器可以直接进行值方法调用，此时相当于直接调用 `foo.clone`，其中 `clone` 的函数签名是 `fn clone(&T) -> T`，由此可以看出 `foo_cloned` 的类型是 `Container<i32>`。

然而，`bar_cloned` 的类型却是 `&Container<T>`，这个不合理啊，明明我们为 `Container<T>` 派生了 `Clone` 特征，因此它也应该是 `Container<T>` 类型才对。万事皆有因，我们先来看下 `derive` 宏最终生成的代码大概是啥样的：

```

impl<T> Clone for Container<T> where T: Clone {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}

```

从上面代码可以看出，派生 `Clone` 能实现的根本是 `T` 实现了 `Clone` 特征：`where T: Clone`，因此 `Container<T>` 就没有实现 `Clone` 特征。

编译器接着会去尝试引用方法调用，此时 `&Container<T>` 引用实现了 `Clone`，最终可以得出 `bar_cloned` 的类型是 `&Container<T>`。

当然，也可以为 `Container<T>` 手动实现 `Clone` 特征：

```

impl<T> Clone for Container<T> {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}

```

此时，编译器首次尝试值方法调用即可通过，因此 `bar_cloned` 的类型变成 `Container<T>`。

这一块儿内容真的挺复杂，每一个坚持看完的读者都是真正的勇士，我也是：为了写好这块儿内容，作者足足花了 4 个小时！

## 变形记(Transmutes)

前方危险，敬请绕行！

类型系统，你让开！我要自己转换这些类型，不成功便成仁！虽然本书大多是关于安全的内容，我还是希望你能仔细考虑避免使用本章讲到的内容。这是你在 Rust 中所能做到的真真正正、彻彻底底、最最可怕的非安全行为，在这里，所有的保护机制都形同虚设。

先让你看看深渊长什么样，开开眼，然后你再决定是否深入：`mem::transmute<T, U>` 将类型 `T` 直接转成类型 `U`，唯一的要求就是，这两个类型占用同样大小的字节数！我的天，这也算限制？这简直就是无底线的转换好吧？看看会导致什么问题：

1. 首先也是最重要的，转换后创建一个任意类型的实例会造成无法想象的混乱，而且根本无法预测。  
不要把 `3` 转换成 `bool` 类型，就算你根本不会去使用该 `bool` 类型，也不要这样转换
2. 变形后会有一个重载的返回类型，即使你没有指定返回类型，为了满足类型推导的需求，依然会产生千奇百怪的类型
3. 将 `&` 变形为 `&mut` 是未定义的行为
  - 这种转换永远都是未定义的
  - 不，你不能这么做
  - 不要多想，你没有那种幸运
4. 变形为一个未指定生命周期的引用会导致[无界生命周期](#)
5. 在复合类型之间互相变换时，你需要保证它们的排列布局是一模一样的！一旦不一样，那么字段就会得到不可预期的值，这也是未定义的行为，至于你会不会因此愤怒，**WHO CARES**，你都用了变形了，老兄！

对于第 5 条，你该如何知道内存的排列布局是一样的呢？对于 `repr(C)` 类型和 `repr(transparent)` 类型来说，它们的布局是有着精确定义的。但是对于你自己的“普通却自信”的 Rust 类型 `repr(Rust)` 来说，它可不是有着精确定义的。甚至同一个泛型类型的不同实例都可以有不同的内存布局。`Vec<i32>` 和 `Vec<u32>` 它们的字段可能有着相同的顺序，也可能没有。对于数据排列布局来说，**什么能保证，什么不能保证** 目前还在 Rust 开发组的[工作任务](#)中呢。

你以为你之前凝视的是深渊吗？不，你凝视的只是深渊的大门。`mem::transmute_copy<T, U>` 才是真正的深渊，它比之前的还要更加危险和不安全。它从 `T` 类型中拷贝出 `U` 类型所需的字节数，然后转换成 `U`。`mem::transmute` 尚有大小检查，能保证两个数据的内存大小一致，现在这哥们干脆连这个也丢了，只不过 `U` 的尺寸若是比 `T` 大，会是一个未定义行为。

当然，你也可以通过裸指针转换和 `unions` (todo!) 获得所有的这些功能，但是你将无法获得任何编译提示或者检查。裸指针转换和 `unions` 也不是魔法，无法逃避上面说的规则。

`transmute` 虽然危险，但作为一本工具书，知识当然要全面，下面列举两个有用的 `transmute` 应用场景：）。

- 将裸指针变成函数指针：

```
fn foo() -> i32 {
    0
}

let pointer = foo as *const ();
let function = unsafe {
    // 将裸指针转换为函数指针
    std::mem::transmute<*const (), fn() -> i32>(pointer)
};
assert_eq!(function(), 0);
```

- 延长生命周期，或者缩短一个静态生命周期寿命：

```
struct R<'a>(&'a i32);

// 将 'b 生命周期延长至 'static 生命周期
unsafe fn extend_lifetime<'b>(r: R<'b>) -> R<'static> {
    std::mem::transmute<R<'b>, R<'static>>(r)
}

// 将 'static 生命周期缩短至 'c 生命周期
unsafe fn shorten_invariant_lifetime<'b, 'c>(r: &'b mut R<'static>) -> &'b mut R<'c>
{
    std::mem::transmute<&'b mut R<'static>, &'b mut R<'c>>(r)
}
```

以上例子非常先进！但是是非常不安全的 Rust 行为！

## 课后练习

---

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。

- [as](#)
  - [习题解答](#)
- [From/Into](#)
  - [习题解答](#)
- [其它转换](#)

◦ 习题解答

---

# 深入 Rust 类型

弱弱地、不负责任地说，Rust 的学习难度之恶名，可能有一半来源于 Rust 的类型系统，而其中一半的一半则来自于本章节的内容。在本章，我们将重点学习如何创建自定义类型，以及了解何为动态大小的类型。

## newtype

何为 newtype？简单来说，就是使用[元组结构体](#)的方式将已有的类型包裹起来：`struct Meters(u32);`，那么此处 `Meters` 就是一个 newtype。

为何需要 newtype？Rust 这多如繁星的 Old 类型满足不了我们吗？这是因为：

- 自定义类型可以让我们给出更有意义和可读性的类型名，例如与其使用 `u32` 作为距离的单位类型，我们可以使用 `Meters`，它的可读性要好得多
- 对于某些场景，只有 newtype 可以很好地解决
- 隐藏内部类型的细节

一箩筐的理由～～让我们先从第二点讲起。

### 为外部类型实现外部特征

在之前的章节中，我们有讲过，如果在外部类型上实现外部特征必须使用 newtype 的方式，否则你就得遵循孤儿规则：要为类型 `A` 实现特征 `T`，那么 `A` 或者 `T` 必须至少有一个在当前的作用范围内。

例如，如果想使用 `println!("{}", v)` 的方式去格式化输出一个动态数组 `Vec`，以期给用户提供更加清晰可读的内容，那么就需要为 `Vec` 实现 `Display` 特征，但是这里有一个问题：`Vec` 类型定义在标准库中，`Display` 亦然，这时就可以祭出大杀器 newtype 来解决：

```

use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}

```

如上所示，使用元组结构体语法 `struct Wrapper(Vec<String>)` 创建了一个 newtype `Wrapper`，然后为它实现 `Display` 特征，最终实现了对 `Vec` 动态数组的格式化输出。

## 更好的可读性及类型异化

首先，更好的可读性不等于更少的代码（如果你学过 Scala，相信会深有体会），其次下面的例子只是一个示例，未必能体现出更好的可读性：

```

use std::ops::Add;
use std::fmt;

struct Meters(u32);
impl fmt::Display for Meters {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "目标地点距离你{}米", self.0)
    }
}

impl Add for Meters {
    type Output = Self;

    fn add(self, other: Meters) -> Self {
        Self(self.0 + other.0)
    }
}
fn main() {
    let d = calculate_distance(Meters(10), Meters(20));
    println!("{}", d);
}

fn calculate_distance(d1: Meters, d2: Meters) -> Meters {
    d1 + d2
}

```

上面代码创建了一个 newtype Meters，为其实现 Display 和 Add 特征，接着对两个距离进行求和计算，最终打印出该距离：

```
目标地点距离你30米
```

事实上，除了可读性外，还有一个极大的优点：如果给 calculate\_distance 传一个其它的类型，例如 struct Millimeters(u32)；，该代码将无法编译。尽管 Meters 和 Millimeters 都是对 u32 类型的简单包装，但是它们是不同的类型！

### 隐藏内部类型的细节

众所周知，Rust 的类型有很多自定义的方法，假如我们把某个类型传给了用户，但是又不想用户调用这些方法，就可以使用 newtype：

```
struct Meters(u32);

fn main() {
    let i: u32 = 2;
    assert_eq!(i.pow(2), 4);

    let n = Meters(i);
    // 下面的代码将报错，因为`Meters`类型上没有`pow`方法
    // assert_eq!(n.pow(2), 4);
}
```

不过需要偷偷告诉你的是，这种方式实际上是掩耳盗铃，因为用户依然可以通过 n.0.pow(2) 的方式来调用内部类型的方法：)

## 类型别名(Type Alias)

除了使用 newtype，我们还可以使用一个更传统的方式来创建新类型：类型别名

```
type Meters = u32
```

嗯，不得不说，类型别名的方式看起来比 newtype 顺眼的多，而且跟其它语言的使用方式几乎一致，但是：**类型别名并不是一个独立的全新的类型，而是某一个类型的别名**，因此编译器依然会把 Meters 当 u32 来使用：

```
type Meters = u32;

let x: u32 = 5;
let y: Meters = 5;

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

上面的代码将顺利编译通过，但是如果你使用 `newtype` 模式，该代码将无情报错，简单做个总结：

- 类型别名仅仅是别名，只是为了让可读性更好，并不是全新的类型，`newtype` 才是！
- 类型别名无法实现为外部类型实现外部特征等功能，而 `newtype` 可以

类型别名除了让类型可读性更好，还能**减少模版代码的使用**：

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --snip--
}
```

`f` 是一个令人眼花缭乱的类型 `Box<dyn Fn() + Send + 'static>`，如果仔细看，会发现其实只有一个 `Send` 特征不认识，`Send` 是什么在这里不重要，你只需理解，`f` 就是一个 `Box<dyn T>` 类型的特征对象，实现了 `Fn()` 和 `Send` 特征，同时生命周期为 `'static`。

因为 `f` 的类型贼长，导致了后面我们在使用它时，到处都充斥这些不太优美的类型标注，好在类型别名可解君忧：

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Bang！是不是？！立刻大幅简化了我们的使用。喝着奶茶、哼着歌、我写起代码撩起妹，何其快哉！

在标准库中，类型别名应用最广的就是简化 `Result<T, E>` 枚举。

例如在 `std::io` 库中，它定义了自己的 `Error` 类型：`std::io::Error`，那么如果要使用该 `Result` 就要用这样的语法：`std::result::Result<T, std::io::Error>;`，想象一下代码中充斥着这样的东东是一种什么感受？颤抖吧。。。

由于使用 `std::io` 库时，它的所有错误类型都是 `std::io::Error`，那么我们完全可以把该错误对用户隐藏起来，只在内部使用即可，因此就可以使用类型别名来简化实现：

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

Bingo，这样一来，其它库只需要使用 `std::io::Result<T>` 即可替代冗长的 `std::result::Result<T, std::io::Error>` 类型。

更香的是，由于它只是别名，因此我们可以用它来调用真实类型的所有方法，甚至包括 `?` 符号！

## !永不返回类型

在[函数](#)那章，曾经介绍过 `!` 类型：`!` 用来说明一个函数永不返回任何值，当时可能体会不深，没事，在学习了更多手法后，保证你有全新的体验：

```
fn main() {
    let i = 2;
    let v = match i {
        0..=3 => i,
        _ => println!("不合规定的值:{}" , i)
    };
}
```

上面函数，会报出一个编译错误：

```
error[E0308]: `match` arms have incompatible types // match的分支类型不同
--> src/main.rs:5:13
|
3 |     let v = match i {
|     |
4 |     |         0..3 => i,
|     |         - this is found to be of type '{integer}' // 该分支返回整数类型
5 |     |         _ => println!("不合规定的值:{}" , i)
|     |         ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected integer, found `()` // 该分支
|     |         返回()单元类型
6 |     |     };
|     |     - `match` arms have incompatible types
```

原因很简单：要赋值给 `v`，就必须保证 `match` 的各个分支返回的值是同一个类型，但是上面一个分支返回数值、另一个分支返回元类型 `()`，自然会出错。

既然 `println` 不行，那再试试 `panic`

```
fn main() {
    let i = 2;
    let v = match i {
        0..=3 => i,
        _ => panic!("不合规定的值:{}" , i)
    };
}
```

神奇的事发生了，此处 `panic` 竟然通过了编译。难道这两个宏拥有不同的返回类型？

你猜的没错：`panic` 的返回值是 `!`，代表它决不会返回任何值，既然没有任何返回值，那自然不会有分支类型不匹配的情况。

# Sized 和不定长类型 DST

在 Rust 中类型有多种抽象的分类方式，例如本书之前章节的：基本类型、集合类型、复合类型等。再比如说，如果从编译器何时能获知类型大小的角度出发，可以分成两类：

- 定长类型( sized )，这些类型的大小在编译时是已知的
- 不定长类型( unsized )，与定长类型相反，它的大小只有到了程序运行时才能动态获知，这种类型又被称为 DST

首先，我们来深入看看何为 DST。

## 动态大小类型 DST

读者大大们之前学过的几乎所有类型，都是固定大小的类型，包括集合 `Vec`、`String` 和 `HashMap` 等，而动态大小类型刚好与之相反：**编译器无法在编译期得知该类型值的大小，只有到了程序运行时，才能动态获知**。对于动态类型，我们使用 `DST` (dynamically sized types)或者 `unsized` 类型来称呼它。

上述的这些集合虽然底层数据可动态变化，感觉像是动态大小的类型。但是实际上，**这些底层数据只是保存在堆上，在栈中还存有一个引用类型**，该引用包含了集合的内存地址、元素数目、分配空间信息，通过这些信息，编译器对于该集合的实际大小了若指掌，最最重要的是：**栈上的引用类型是固定大小的**，因此它们依然是固定大小的类型。

**正因为编译器无法在编译期获知类型大小，若你试图在代码中直接使用 DST 类型，将无法通过编译。**

现在给你一个挑战：想出几个 DST 类型。俺厚黑地说一句，估计大部分人都想不出这样的一个类型，就连我，如果不是查询着资料在写，估计一时半会儿也想不到一个。

先来看一个最直白的：

### 试图创建动态大小的数组

```
fn my_function(n: usize) {  
    let array = [123; n];  
}
```

以上代码就会报错(错误输出的内容并不是因为 DST，但根本原因是类似的)，因为 `n` 在编译期无法得知，而数组类型的一个组成部分就是长度，长度变为动态的，自然类型就变成了 `unsized`。

## 切片

切片也是一个典型的 DST 类型，具体详情参见另一篇文章：[易混淆的切片和切片引用](#)。

### str

考虑一下这个类型： `str`，感觉有点眼生？是的，它既不是 `String` 动态字符串，也不是 `&str` 字符串切片，而是一个 `str`。它是一个动态类型，同时还是 `String` 和 `&str` 的底层数据类型。由于 `str` 是动态类型，因此它的大小直到运行期才知道，下面的代码会因此报错：

```
// error
let s1: str = "Hello there!";
let s2: str = "How's it going?";

// ok
let s3: &str = "on?"
```

Rust 需要明确地知道一个特定类型的值占据了多少内存空间，同时该类型的所有值都必须使用相同大小的内存。如果 Rust 允许我们使用这种动态类型，那么这两个 `str` 值就需要占用同样大小的内存，这显然是不现实的：`s1` 占用了 12 字节，`s2` 占用了 15 字节，总不至于为了满足同样的内存大小，用空白字符去填补字符串吧？

所以，我们只有一条路走，那就是给它们一个固定大小的类型：`&str`。那么为何字符串切片 `&str` 就是固定大小呢？因为它的引用存储在栈上，具有固定大小(类似指针)，同时它指向的数据存储在堆中，也是已知的大小，再加上 `&str` 引用中包含有堆上数据内存地址、长度等信息，因此最终可以得出字符串切片是固定大小类型的结论。

与 `&str` 类似，`String` 字符串也是固定大小的类型。

正是因为 `&str` 的引用有了底层堆数据的明确信息，它才是固定大小类型。假设如果它没有这些信息呢？那它也将变成一个动态类型。因此，将动态数据固定化的秘诀就是**使用引用指向这些动态数据，然后在引用中存储相关的内存位置、长度等信息**。

## 特征对象

```
fn foobar_1(thing: &dyn MyThing) {}      // OK
fn foobar_2(thing: Box<dyn MyThing>) {} // OK
fn foobar_3(thing: MyThing) {}           // ERROR!
```

如上所示，只能通过引用或 `Box` 的方式来使用特征对象，直接使用将报错！

## 总结：只能间接使用的 DST

Rust 中常见的 DST 类型有: `str`、`[T]`、`dyn Trait`，它们都无法单独被使用，必须要通过引用或者 `Box` 来间接使用。

我们之前已经见过，使用 `Box` 将一个没有固定大小的特征变成一个有固定大小的特征对象，那能否故技重施，将 `str` 封装成一个固定大小类型？留个悬念先，我们来看看 `Sized` 特征。

## Sized 特征

既然动态类型的问题这么大，那么在使用泛型时，Rust 如何保证我们的泛型参数是固定大小的类型呢？例如以下泛型函数：

```
fn generic<T>(t: T) {  
    // --snip--  
}
```

该函数很简单，就一个泛型参数 `T`，那么如何保证 `T` 是固定大小的类型？仔细回想下，貌似在之前的课程章节中，我们也没有做过任何事情去做相关的限制，那 `T` 怎么就成了固定大小的类型了？奥秘在于编译器自动帮我们加上了 `Sized` 特征约束：

```
fn generic<T: Sized>(t: T) {  
    // --snip--  
}
```

在上面，Rust 自动添加的特征约束 `T: Sized`，表示泛型函数只能用于一切实现了 `Sized` 特征的类型上，而所有在编译时就能知道其大小的类型，都会自动实现 `Sized` 特征，例如。。。也没啥好例如的，你能想到的几乎所有类型都实现了 `Sized` 特征，除了上面那个坑坑的 `str`，哦，还有特征。

**每一个特征都是一个可以通过名称来引用的动态大小类型。**因此如果想把特征作为具体的类型来传递给函数，你必须将其转换成一个特征对象：诸如 `&dyn Trait` 或者 `Box<dyn Trait>` (还有 `Rc<dyn Trait>`) 这些引用类型。

现在还有一个问题：假如想在泛型函数中使用动态数据类型怎么办？可以使用 `?Sized` 特征(不得不说这个命名方式很 Rusty，竟然有点幽默)：

```
fn generic<T: ?Sized>(t: &T) {  
    // --snip--  
}
```

?Sized 特征用于表明类型  $T$  既有可能是固定大小的类型，也可能是动态大小的类型。还有一点要注意的是，函数参数类型从  $T$  变成了  $\&T$ ，因为  $T$  可能是动态大小的，因此需要用一个固定大小的指针(引用)来包裹它。

## Box<str>

在结束前，再来看看之前遗留的问题：使用 `Box` 可以将一个动态大小的特征变成一个具有固定大小的特征对象，能否故技重施，将 `str` 封装成一个固定大小类型？

先回想下，章节前面的内容介绍过该如何把一个动态大小类型转换成固定大小的类型：**使用引用指向这些动态数据，然后在引用中存储相关的内存位置、长度等信息。**

好的，根据这个，我们来一起推测。首先，`Box<str>` 使用了一个引用来指向 `str`，嗯，满足了第一个条件。但是第二个条件呢？`Box` 中有该 `str` 的长度信息吗？显然是 No。那为什么特征就可以变成特征对象？其实这个还蛮复杂的，简单来说，对于特征对象，编译器无需知道它具体是什么类型，只要知道它能调用哪几个方法即可，因此编译器帮我们实现了剩下的一切。

来验证下我们的推测：

```
fn main() {
    let s1: Box<str> = Box::new("Hello there!" as str);
}
```

报错如下：

```
error[E0277]: the size for values of type `str` cannot be known at compilation time
--> src/main.rs:2:24
 |
2 |     let s1: Box<str> = Box::new("Hello there!" as str);
|           ^^^^^^^^^ doesn't have a size known at compile-time
|
= help: the trait `Sized` is not implemented for `str`
= note: all function arguments must have a statically known size
```

提示得很清晰，不知道 `str` 的大小，因此无法使用这种语法进行 `Box` 进装，但是你可以这么做：

```
let s1: Box<str> = "Hello there!".into();
```

主动转换成 `str` 的方式不可行，但是可以让编译器来帮我们完成，只要告诉它我们需要的类型即可。

# 整数转换为枚举

在 Rust 中，从枚举到整数的转换很容易，但是反过来，就没那么容易，甚至部分实现还挺邪恶，例如使用 `transmute`。

## 一个真实场景的需求

在实际场景中，从整数到枚举的转换有时还是非常需要的，例如你有一个枚举类型，然后需要从外面传入一个整数，用于控制后续的流程走向，此时就需要用整数去匹配相应的枚举(你也可以用整数匹配整数，看看会不会被喷)。

既然有了需求，剩下的就是看看该如何实现，这篇文章的水远比你想象的要深，且看八仙过海各显神通。

## C 语言的实现

对于 C 语言来说，万物皆邪恶，因此我们不讨论安全，只看实现，不得不说很简洁：

```
#include <stdio.h>

enum atomic_number {
    HYDROGEN = 1,
    HELIUM = 2,
    // ...
    IRON = 26,
};

int main(void)
{
    enum atomic_number element = 26;

    if (element == IRON) {
        printf("Beware of Rust!\n");
    }

    return 0;
}
```

但是在 Rust 中，以下代码：

```
enum MyEnum {
    A = 1,
    B,
    C,
}

fn main() {
    // 将枚举转换成整数，顺利通过
    let x = MyEnum::C as i32;

    // 将整数转换为枚举，失败
    match x {
        MyEnum::A => {}
        MyEnum::B => {}
        MyEnum::C => {}
        _ => {}
    }
}
```

就会报错: `MyEnum::A => {} mismatched types, expected i32, found enum MyEnum。`

## 使用三方库

首先可以想到的肯定是三方库，毕竟 Rust 的生态目前已经发展的很不错，类似的需求总是有的，这里我们先使用 `num-traits` 和 `num-derive` 来试试。

在 `Cargo.toml` 中引入：

```
[dependencies]
num-traits = "0.2.14"
num-derive = "0.3.3"
```

代码如下：

```

use num_derive::FromPrimitive;
use num_traits::FromPrimitive;

#[derive(FromPrimitive)]
enum MyEnum {
    A = 1,
    B,
    C,
}

fn main() {
    let x = 2;

    match FromPrimitive::from_i32(x) {
        Some(MyEnum::A) => println!("Got A"),
        Some(MyEnum::B) => println!("Got B"),
        Some(MyEnum::C) => println!("Got C"),
        None           => println!("Couldn't convert {}", x),
    }
}

```

除了上面的库，还可以使用一个较新的库：[num\\_enums](#)。

## TryFrom + 宏

在 Rust 1.34 后，可以实现 TryFrom 特征来做转换：

```

use std::convert::TryFrom;

impl TryFrom<i32> for MyEnum {
    type Error = ();

    fn try_from(v: i32) -> Result<Self, Self::Error> {
        match v {
            x if x == MyEnum::A as i32 => Ok(MyEnum::A),
            x if x == MyEnum::B as i32 => Ok(MyEnum::B),
            x if x == MyEnum::C as i32 => Ok(MyEnum::C),
            _ => Err(()),
        }
    }
}

```

以上代码定义了从 `i32` 到 `MyEnum` 的转换，接着就可以使用 `TryInto` 来实现转换：

```

use std::convert::TryInto;

fn main() {
    let x = MyEnum::C as i32;

    match x.try_into() {
        Ok(MyEnum::A) => println!("a"),
        Ok(MyEnum::B) => println!("b"),
        Ok(MyEnum::C) => println!("c"),
        Err(_) => eprintln!("unknown number"),
    }
}

```

但是上面的代码有个问题，你需要为每个枚举成员都实现一个转换分支，非常麻烦。好在可以使用宏来简化，自动根据枚举的定义来实现 TryFrom 特征：

```

#[macro_export]
macro_rules! back_to_enum {
    ($(#[$meta:meta])* $vis:vis enum $name:ident {
        $($(#[$vmeta:meta])* $vname:ident $(= $val:expr)?,)*
    }) => {
        $($[#[$meta]])*
        $vis enum $name {
            $($(#[$vmeta])* $vname $(= $val)?,)*
        }
    }

    impl std::convert::TryFrom<i32> for $name {
        type Error = ();

        fn try_from(v: i32) -> Result<$Self, Self::Error> {
            match v {
                $($x if x == $name::$vname as i32 => Ok($name::$vname),)*
                _ => Err(()),
            }
        }
    }
}

back_to_enum! {
    enum MyEnum {
        A = 1,
        B,
        C,
    }
}

```

## 邪恶之王 std::mem::transmute

这个方法原则上并不推荐，但是有其存在的意义，如果要使用，你需要清晰的知道自己为什么使用。

在之前的类型转换章节，我们提到过非常邪恶的 [transmute 转换](#)，其实，当你知道数值一定不会超过枚举的范围时(例如枚举成员对应 1, 2, 3，传入的整数也在这个范围内)，就可以使用这个方法完成变形。

---

最好使用#[repr(..)]来控制底层类型的大小，免得本来需要 i32，结果传入 i64，最终内存无法对齐，产生奇怪的结果

---

```
#[repr(i32)]
enum MyEnum {
    A = 1, B, C
}

fn main() {
    let x = MyEnum::C;
    let y = x as i32;
    let z: MyEnum = unsafe { std::mem::transmute(y) };

    // match the enum that came from an int
    match z {
        MyEnum::A => { println!("Found A"); }
        MyEnum::B => { println!("Found B"); }
        MyEnum::C => { println!("Found C"); }
    }
}
```

既然是邪恶之王，当然得有真本事，无需标准库、也无需 unstable 的 Rust 版本，我们就完成了转换！awesome!??

## 总结

本文列举了常用(其实差不多也是全部了，还有一个 unstable 特性没提到)的从整数转换为枚举的方式，推荐度按照出现的先后顺序递减。

但是推荐度最低，不代表它就没有出场的机会，只要使用边界清晰，一样可以大放光彩，例如最后的 [transmute 函数](#)。

# 智能指针

在各个编程语言中，指针的概念几乎都是相同的：**指针是一个包含了内存地址的变量，该内存地址引用或者指向了另外的数据。**

在 Rust 中，最常见的指针类型是引用，引用通过 `&` 符号表示。不同于其它语言，引用在 Rust 中被赋予了更深层次的含义，那就是：借用其它变量的值。引用本身很简单，除了指向某个值外并没有其它的功能，也不会造成性能上的额外损耗，因此是 Rust 中使用最多的指针类型。

而智能指针则不然，它虽然也号称指针，但是它是一个复杂的家伙：通过比引用更复杂的数据结构，包含比引用更多的信息，例如元数据，当前长度，最大可用长度等。总之，Rust 的智能指针并不是独创，在 C++ 或者其他语言中也存在相似的概念。

Rust 标准库中定义的那些智能指针，虽重但强，可以提供比引用更多的功能特性，例如本章将讨论的引用计数智能指针。该智能指针允许你同时拥有同一个数据的多个所有权，它会跟踪每一个所有者并进行计数，当所有的所有者都归还后，该智能指针及指向的数据将自动被清理释放。

引用和智能指针的另一个不同在于前者仅仅是借用了数据，而后者往往可以拥有它们指向的数据，然后再为其它人提供服务。

在之前的章节中，实际上我们已经见识过多种智能指针，例如动态字符串 `String` 和动态数组 `Vec`，它们的数据结构中不仅仅包含了指向底层数据的指针，还包含了当前长度、最大长度等信息，其中 `String` 智能指针还提供了一种担保信息：所有的数据都是合法的 `UTF-8` 格式。

智能指针往往是基于结构体实现，它与我们自定义的结构体最大的区别在于它实现了 `Deref` 和 `Drop` 特征：

- `Deref` 可以让智能指针像引用那样工作，这样你就可以写出同时支持智能指针和引用的代码，例如 `*T`
- `Drop` 允许你指定智能指针超出作用域后自动执行的代码，例如做一些数据清除等收尾工作

智能指针在 Rust 中很常见，我们在本章不会全部讲解，而是挑选几个最常用、最有代表性的进行讲解：

- `Box<T>`，可以将值分配到堆上
- `Rc<T>`，引用计数类型，允许多所有权存在
- `Ref<T>` 和 `RefMut<T>`，允许将借用规则检查从编译期移动到运行期进行

# Box<T> 堆对象分配

关于作者帅不帅，估计争议还挺多的，但是如果说 `Box<T>` 是不是 Rust 中最常见的智能指针，那估计没有任何争议。因为 `Box<T>` 允许你将一个值分配到堆上，然后在线上保留一个智能指针指向堆上的数据。

之前我们在[所有权章节](#)简单讲过堆栈的概念，这里再补充一些。

## Rust 中的堆栈

高级语言 Python/Java 等往往会弱化堆栈的概念，但是要用好 C/C++/Rust，就必须对堆栈有深入的了解，原因是两者的内存管理方式不同：前者有 GC 垃圾回收机制，因此无需你去关心内存的细节。

栈内存从高位地址向下增长，且栈内存是连续分配的，一般来说**操作系统对栈内存的大小都有限制**，因此 C 语言中无法创建任意长度的数组。在 Rust 中，`main` 线程的**栈大小是 8MB**，普通线程是 2MB，在函数调用时会在其中创建一个临时栈空间，调用结束后 Rust 会让这个栈空间里的对象自动进入 `Drop` 流程，最后栈顶指针自动移动到上一个调用栈顶，无需程序员手动干预，因而栈内存申请和释放是非常高效的。

与栈相反，堆上内存则是从低位地址向上增长，**堆内存通常只受物理内存限制**，而且通常是不连续的，因此从性能的角度看，栈往往比堆更高。

相比其它语言，Rust 堆上对象还有一个特殊之处，它们都拥有一个所有者，因此受所有权规则的限制：当赋值时，发生的是所有权的转移（只需浅拷贝栈上的引用或智能指针即可），例如以下代码：

```
fn main() {
    let b = foo("world");
    println!("{}", b);
}

fn foo(x: &str) -> String {
    let a = "Hello, ".to_string() + x;
    a
}
```

在 `foo` 函数中，`a` 是 `String` 类型，它其实是一个智能指针结构体，该智能指针存储在函数栈中，指向堆上的字符串数据。当被从 `foo` 函数转移给 `main` 中的 `b` 变量时，栈上的智能指针被复制一份赋予给 `b`，而底层数据无需发生改变，这样就完成了所有权从 `foo` 函数内部到 `b` 的转移。

## 堆栈的性能

很多人可能会觉得栈的性能肯定比堆高，其实未必。由于我们在后面的性能专题会专门讲解堆栈的性能问题，因此这里就大概给出结论：

- 小型数据，在栈上的分配性能和读取性能都要比堆上高
- 中型数据，栈上分配性能高，但是读取性能和堆上并无区别，因为无法利用寄存器或 CPU 高速缓存，最终还是要经过一次内存寻址
- 大型数据，只建议在堆上分配和使用

总之，栈的分配速度肯定比堆上快，但是读取速度往往取决于你的数据能不能放入寄存器或 CPU 高速缓存。因此不要仅仅因为堆上性能不如栈这个印象，就总是优先选择栈，导致代码更复杂的实现。

## Box 的使用场景

由于 Box 是简单的封装，除了将值存储在堆上外，并没有其它性能上的损耗。而性能和功能往往是鱼和熊掌，因此 Box 相比其它智能指针，功能较为单一，可以在以下场景中使用它：

- 特意的将数据分配在堆上
- 数据较大时，又不想在转移所有权时进行数据拷贝
- 类型的大小在编译期无法确定，但是我们又需要固定大小的类型时
- 特征对象，用于说明对象实现了一个特征，而不是某个特定的类型

以上场景，我们在本章将一一讲解，后面车速较快，请系好安全带。

### 使用 Box<T> 将数据存储在堆上

如果一个变量拥有一个数值 `let a = 3`，那变量 a 必然是存储在栈上的，那如果我们想要 a 的值存储在堆上就需要使用 `Box<T>`：

```
fn main() {
    let a = Box::new(3);
    println!("a = {}", a); // a = 3

    // 下面一行代码将报错
    // let b = a + 1; // cannot add `<integer>` to `Box<<integer>>`
}
```

这样就可以创建一个智能指针指向了存储在堆上的 3，并且 a 持有了该指针。在本章的引言中，我们提到了智能指针往往都实现了 `Deref` 和 `Drop` 特征，因此：

- `println!` 可以正常打印出 `a` 的值，是因为它隐式地调用了 `Deref` 对智能指针 `a` 进行了解引用
- 最后一行代码 `let b = a + 1` 报错，是因为在表达式中，我们无法自动隐式地执行 `Deref` 解引用操作，你需要使用 `\*` 操作符 `let b = \*a + 1`，来显式的进行解引用
- `a` 持有的智能指针将在作用域结束（`main` 函数结束）时，被释放掉，这是因为 `Box<T>` 实现了 `Drop` 特征

以上的例子在实际代码中其实很少会存在，因为将一个简单的值分配到堆上并没有太大的意义。将其分配在栈上，由于寄存器、CPU 缓存的原因，它的性能将更好，而且代码可读性也更好。

## 避免栈上数据的拷贝

当栈上数据转移所有权时，实际上是把数据拷贝了一份，最终新旧变量各自拥有不同的数据，因此所有权并未转移。

而堆上则不然，底层数据并不会被拷贝，转移所有权仅仅是复制一份栈中的指针，再将新的指针赋予新的变量，然后让拥有旧指针的变量失效，最终完成了所有权的转移：

```
fn main() {
    // 在栈上创建一个长度为1000的数组
    let arr = [0;1000];
    // 将arr所有权转移arr1，由于 `arr` 分配在栈上，因此这里实际上是直接重新深拷贝了一份数据
    let arr1 = arr;

    // arr 和 arr1 都拥有各自的栈上数组，因此不会报错
    println!("{:?}", arr.len());
    println!("{:?}", arr1.len());

    // 在堆上创建一个长度为1000的数组，然后使用一个智能指针指向它
    let arr = Box::new([0;1000]);
    // 将堆上数组的所有权转移给 arr1，由于数据在堆上，因此仅仅拷贝了智能指针的结构体，底层数据并没有
    被拷贝
    // 所有权顺利转移给 arr1，arr 不再拥有所有权
    let arr1 = arr;
    println!("{:?}", arr1.len());
    // 由于 arr 不再拥有底层数组的所有权，因此下面代码将报错
    // println!("{:?}", arr.len());
}
```

从以上代码，可以清晰看出大块的数据为何应该放入堆中，此时 `Box` 就成为了我们最好的帮手。

## 将动态大小类型变为 Sized 固定大小类型

Rust 需要在编译时知道类型占用多少空间，如果一种类型在编译时无法知道具体的大小，那么被称为动态大小类型 DST。

其中一种无法在编译时知道大小的类型是**递归类型**: 在类型定义中又使用到了自身, 或者说该类型的值的一部分可以是相同类型的其它值, 这种值的嵌套理论上可以无限进行下去, 所以 Rust 不知道递归类型需要多少空间:

```
enum List {
    Cons(i32, List),
    Nil,
}
```

以上就是函数式语言中常见的 `Cons List`, 它的每个节点包含一个 `i32` 值, 还包含了一个新的 `List`, 因此这种嵌套可以无限进行下去, Rust 认为该类型是一个 DST 类型, 并给予报错:

```
error[E0072]: recursive type `List` has infinite size //递归类型 `List` 拥有无限长的大小
--> src/main.rs:3:1
|
3 | enum List {
| ^^^^^^^^^^^ recursive type has infinite size
4 |     Cons(i32, List),
|             ----- recursive without indirection
```

此时若想解决这个问题, 就可以使用我们的 `Box<T>`:

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

只需要将 `List` 存储到堆上, 然后使用一个智能指针指向它, 即可完成从 DST 到 Sized 类型(固定大小类型)的华丽转变。

## 特征对象

在 Rust 中, 想实现不同类型组成的数组只有两个办法: 枚举和特征对象, 前者限制较多, 因此后者往往是最常用的解决办法。

```

trait Draw {
    fn draw(&self);
}

struct Button {
    id: u32,
}
impl Draw for Button {
    fn draw(&self) {
        println!("这是屏幕上第{}号按钮", self.id)
    }
}

struct Select {
    id: u32,
}

impl Draw for Select {
    fn draw(&self) {
        println!("这个选择框贼难用{}", self.id)
    }
}

fn main() {
    let elems: Vec<Box<dyn Draw>> = vec![Box::new(Button { id: 1 }), Box::new(Select { id: 2 })];

    for e in elems {
        e.draw()
    }
}

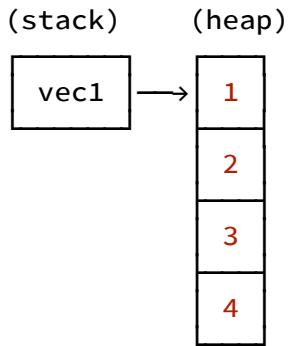
```

以上代码将不同类型的 `Button` 和 `Select` 包装成 `Draw` 特征的特征对象，放入一个数组中，`Box<dyn Draw>` 就是特征对象。

其实，特征也是 DST 类型，而特征对象在做的就是将 DST 类型转换为固定大小类型。

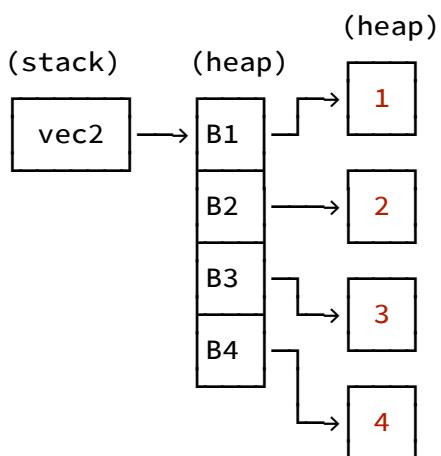
## Box 内存布局

先来看看 `Vec<i32>` 的内存布局：



之前提到过 `Vec` 和 `String` 都是智能指针，从上图可以看出，该智能指针存储在栈中，然后指向堆上的数组数据。

那如果数组中每个元素都是一个 `Box` 对象呢？来看看 `Vec<Box<i32>>` 的内存布局：



上面的 `B1` 代表被 `Box` 分配到堆上的值 `1`。

可以看出智能指针 `vec2` 依然是存储在栈上，然后指针指向一个堆上的数组，该数组中每个元素都是一个 `Box` 智能指针，最终 `Box` 智能指针又指向了存储在堆上的实际值。

因此当我们从数组中取出某个元素时，取到的是对应的智能指针 `Box`，需要对该智能指针进行解引用，才能取出最终的值：

```

fn main() {
    let arr = vec![Box::new(1), Box::new(2)];
    let (first, second) = (&arr[0], &arr[1]);
    let sum = **first + **second;
}

```

以上代码有几个值得注意的点：

- 使用 `&` 借用数组中的元素，否则会报所有权错误

- 表达式不能隐式的解引用，因此必须使用 `**` 做两次解引用，第一次将 `&Box<i32>` 类型转成 `Box<i32>`，第二次将 `Box<i32>` 转成 `i32`

## Box::leak

`Box` 中还提供了一个非常有用的关联函数：`Box::leak`，它可以消费掉 `Box` 并且强制目标值从内存中泄漏，读者可能会觉得，这有啥用啊？

其实还真有点用，例如，你可以把一个 `String` 类型，变成一个 `'static` 生命周期的 `&str` 类型：

```
fn main() {
    let s = gen_static_str();
    println!("{}", s);
}

fn gen_static_str() -> &'static str{
    let mut s = String::new();
    s.push_str("hello, world");

    Box::leak(s.into_boxed_str())
}
```

在之前的代码中，如果 `String` 创建于函数中，那么返回它的唯一方法就是转移所有权给调用者 `fn move_str() -> String`，而通过 `Box::leak` 我们不仅返回了一个 `&str` 字符串切片，它还是 `'static` 生命周期的！

要知道真正具有 `'static` 生命周期的往往都是编译期就创建的值，例如 `let v = "hello, world"`，这里 `v` 是直接打包到二进制可执行文件中的，因此该字符串具有 `'static` 生命周期，再比如 `const` 常量。

又有读者要问了，我还可以手动为变量标注 `'static` 啊。其实你标注的 `'static` 只是用来忽悠编译器的，但是超出作用域，一样被释放回收。而使用 `Box::leak` 就可以将一个运行期的值转为 `'static`。

## 使用场景

光看上面的描述，大家可能还是云里雾里、一头雾水。

那么我说一个简单的场景，**你需要一个在运行期初始化的值，但是可以全局有效，也就是和整个程序活得一样久**，那么就可以使用 `Box::leak`，例如有一个存储配置的结构体实例，它是在运行期动态插入内容，那么就可以将其转为全局有效，虽然 `Rc/Arc` 也可以实现此功能，但是 `Box::leak` 是性能最高的。

## 总结

Box 背后是调用 `jemalloc` 来做内存管理，所以堆上的空间无需我们的手动管理。与此类似，带 GC 的语言中的对象也是借助于 Box 概念来实现的，**一切皆对象 = 一切皆 Box**，只不过我们无需自己去 Box 罢了。

其实很多时候，编译器的鞭答可以助我们更快的成长，例如所有权规则里的借用、move、生命周期就是编译器在教我们做人，哦不是，是教我们深刻理解堆栈、内存布局、作用域等等你在其它 GC 语言无需去关注的东西。刚开始是很痛苦，但是一旦熟悉了这套规则，写代码的效率和代码本身的质量将飞速上升，直到你可以用 Java 开发的效率写出 Java 代码不可企及的性能和安全性，最终 Rust 语言所谓的开发效率低、心智负担高，对你来说终究不是个事。

因此，不要怪 Rust，**它只是在帮我们成为那个更好的程序员，而这些苦难终究成为我们走向优秀的垫脚石。**

# Deref 解引用

在开始之前，我们先来看一段代码：

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8
}

impl Person {
    fn new(name: String, age: u8) -> Self {
        Person { name, age }
    }

    fn display(self: &mut Person, age: u8) {
        let Person{name, age} = &self;
    }
}
```

以上代码有一个很奇怪的地方：在 `display` 方法中，`self` 是 `&mut Person` 的类型，接着我们对其取了一次引用 `&self`，此时 `&self` 的类型是 `&&mut Person`，然后我们又将其和 `Person` 类型进行匹配，取出其中的值。

那么问题来了，Rust 不是号称安全的语言吗？为何允许将 `&&mut Person` 跟 `Person` 进行匹配呢？答案就在本章节中，等大家学完后，再回头自己来解决这个问题 :) 下面正式开始咱们的新章节学习。

何为智能指针？能不让你写出 `****s` 形式的解引用，我认为就是智能:)，智能指针的名称来源，主要就在于它实现了 `Deref` 和 `Drop` 特征，这两个特征可以智能地帮助我们节省使用上的负担：

- `Deref` 可以让智能指针像引用那样工作，这样你就可以写出同时支持智能指针和引用的代码，例如 `*T`
- `Drop` 允许你指定智能指针超出作用域后自动执行的代码，例如做一些数据清除等收尾工作

先来看看 `Deref` 特征是如何工作的。

## 通过 `*` 获取引用背后的值

在正式讲解 `Deref` 之前，我们先来看下常规引用的解引用。

常规引用是一个指针类型，包含了目标数据存储的内存地址。对常规引用使用 `*` 操作符，就可以通过解引用的方式获取到内存地址对应的数据值：

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

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

这里 `y` 就是一个常规引用，包含了值 `5` 所在的内存地址，然后通过解引用 `*y`，我们获取到了值 `5`。如果你试图执行 `assert_eq!(5, y);`，代码就会无情报错，因为你无法将一个引用与一个数值做比较：

```
error[E0277]: can't compare '{integer}' with '&{integer}' //无法将{integer} 与&
{integer}进行比较
--> src/main.rs:6:5
|
6 |     assert_eq!(5, y);
|     ^^^^^^^^^^^^^ no implementation for '{integer} == &{integer}'
|
= help: the trait `PartialEq<&{integer}>` is not implemented for '{integer}' //
// 你需要为{integer}实现用于比较的特征PartialEq<&{integer}>
```

## 智能指针解引用

上面所说的解引用方式和其它大多数语言并无区别，但是 Rust 中将解引用提升到了一个新高度。考虑一下智能指针，它是一个结构体类型，如果你直接对它进行 `*myStruct`，显然编译器不知道该如何办，因此我们可以为智能指针结构体实现 `Deref` 特征。

实现 `Deref` 后的智能指针结构体，就可以像普通引用一样，通过 `*` 进行解引用，例如 `Box<T>` 智能指针：

```
fn main() {
    let x = Box::new(1);
    let sum = *x + 1;
}
```

智能指针 `x` 被 `*` 解引用为 `i32` 类型的值 `1`，然后再进行求和。

### 定义自己的智能指针

现在，让我们一起来实现一个智能指针，功能上类似 `Box<T>`。由于 `Box<T>` 本身很简单，并没有包含类如长度、最大长度等信息，因此用一个元组结构体即可。

```

struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

```

跟 `Box<T>` 一样，我们的智能指针也持有一个 `T` 类型的值，然后使用关联函数 `MyBox::new` 来创建智能指针。由于还未实现 `Deref` 特征，此时使用 `*` 肯定会报错：

```

fn main() {
    let y = MyBox::new(5);

    assert_eq!(5, *y);
}

```

运行后，报错如下：

```

error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:12:19
 |
12 |     assert_eq!(5, *y);
      ^ ^

```

### 为智能指针实现 `Deref` 特征

现在来为 `MyBox` 实现 `Deref` 特征，以支持 `*` 解引用操作符：

```

use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

```

很简单，当解引用 `MyBox` 智能指针时，返回元组结构体中的元素 `&self.0`，有几点要注意的：

- 在 `Deref` 特征中声明了关联类型 `Target`，在之前章节中介绍过，关联类型主要是为了提升代码可读性
- `deref` 返回的是一个常规引用，可以被 `*` 进行解引用

之前报错的代码此时已能顺利编译通过。当然，标准库实现的智能指针要考虑很多边边角角情况，肯定比我们的实现要复杂。

## \* 背后的原理

当我们对智能指针 `Box` 进行解引用时，实际上 Rust 为我们调用了以下方法：

```
*(y.deref())
```

首先调用 `deref` 方法返回值的常规引用，然后通过 `*` 对常规引用进行解引用，最终获取到目标值。

至于 Rust 为什么要使用这个有点啰嗦的方式实现，原因在于所有权系统的存在。如果 `deref` 方法直接返回一个值，而不是引用，那么该值的所有权将被转移给调用者，而我们不希望调用者仅仅只是 `*T` 一下，就拿走了智能指针中包含的值。

需要注意的是，`*` 不会无限递归替换，从 `*y` 到 `*(y.deref())` 只会发生一次，而不会继续进行替换然后产生形如 `*((y.deref()).deref())` 的怪物。

## 函数和方法中的隐式 Deref 转换

对于函数和方法的传参，Rust 提供了一个极具有用的隐式转换：`Deref` 转换。若一个类型实现了 `Deref` 特征，那它的引用在传给函数或方法时，会根据参数签名来决定是否进行隐式的 `Deref` 转换，例如：

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

fn display(s: &str) {
    println!("{}", s);
}
```

以上代码有几点值得注意：

- `String` 实现了 `Deref` 特征，可以在需要时自动被转换为 `&str` 类型
- `&s` 是一个 `&String` 类型，当它被传给 `display` 函数时，自动通过 `Deref` 转换成了 `&str`
- 必须使用 `&s` 的方式来触发 `Deref` (仅引用类型的实参才会触发自动解引用)

## 连续的隐式 Deref 转换

如果你以为 `Deref` 仅仅这点作用，那就大错特错了。`Deref` 可以支持连续的隐式转换，直到找到适合的形式为止：

```
fn main() {
    let s = MyBox::new(String::from("hello world"));
    display(&s)
}

fn display(s: &str) {
    println!("{}", s);
}
```

这里我们使用了之前自定义的智能指针 `MyBox`，并将其通过连续的隐式转换变成 `&str` 类型：首先 `MyBox` 被 `Deref` 成 `String` 类型，结果并不能满足 `display` 函数参数的要求，编译器发现 `String` 还可以继续 `Deref` 成 `&str`，最终成功的匹配了函数参数。

想象一下，假如 Rust 没有提供这种隐式转换，我们该如何调用 `display` 函数？

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    display(&(*m)[..]);
}
```

结果不言而喻，肯定是 `&s` 的方式优秀得多。总之，当参与其中的类型定义了 `Deref` 特征时，Rust 会分析该类型并且连续使用 `Deref` 直到最终获得一个引用来匹配函数或者方法的参数类型，这种行为完全不会造成任何的性能损耗，因为完全是在编译期完成。

但是 `Deref` 并不是没有缺点，缺点就是：如果你不知道某个类型是否实现了 `Deref` 特征，那么在看到某段代码时，并不能在第一时间反应过来该代码发生了隐式的 `Deref` 转换。事实上，不仅仅是 `Deref`，在 Rust 中还有各种 `From/Into` 等等会给阅读代码带来一定负担的特征。还是那句话，一切选择都是权衡，有得必有失，得了代码的简洁性，往往就失去了可读性，Go 语言就是一个刚好相反的例子。

再来看一下在方法、赋值中自动应用 `Deref` 的例子：

```
fn main() {
    let s = MyBox::new(String::from("hello, world"));
    let s1: &str = &s;
    let s2: String = s.to_string();
}
```

对于 `s1`，我们通过两次 `Deref` 将 `&str` 类型的值赋给了它（**赋值操作需要手动解引用**）；而对于 `s2`，我们在其上直接调用方法 `to_string`，实际上 `MyBox` 根本没有实现该方法，能调用

`to_string`，完全是因为编译器对 `MyBox` 应用了 `Deref` 的结果（**方法调用会自动解引用**）。

## Deref 规则总结

在上面，我们零碎的介绍了不少关于 `Deref` 特征的知识，下面来通过较为正式的方式来对其规则进行下总结。

一个类型为 `T` 的对象 `foo`，如果 `T: Deref<Target=U>`，那么，相关 `foo` 的引用 `&foo` 在应用的时候会自动转换为 `&U`。

粗看这条规则，貌似有点类似于 `AsRef`，而跟 解引用 似乎风马牛不相及，实际里面有些玄妙之处。

### 引用归一化

Rust 编译器实际上只能对 `&v` 形式的引用进行解引用操作，那么问题来了，如果是一个智能指针或者 `&&&&v` 类型的呢？该如何对这两个进行解引用？

答案是：Rust 会在解引用时自动把智能指针和 `&&&&v` 做引用归一化操作，转换成 `&v` 形式，最终再对 `&v` 进行解引用：

- 把智能指针（比如在库中定义的，`Box`、`Rc`、`Arc`、`Cow` 等）从结构体脱壳为内部的引用类型，也就是转成结构体内部的 `&v`
- 把多重 `&`，例如 `&&&&&&v`，归一成 `&v`

关于第二种情况，这么干巴巴的说，也许大家会迷迷糊糊的，我们来看一段标准库源码：

```
impl<T: ?Sized> Deref for &T {
    type Target = T;

    fn deref(&self) -> &T {
        *self
    }
}
```

在这段源码中，`&T` 被自动解引用为 `T`，也就是 `&T: Deref<Target=T>`。按照这个代码，`&&&&T` 会被自动解引用为 `&&&T`，然后再自动解引用为 `&&T`，以此类推，直到最终变成 `&T`。

PS: 以下是 LLVM 编译后的部分中间层代码：

```
// Rust 代码
let mut _2: &i32;
let _3: &&&&i32;

bb0: {
    _2 = (*(*(_3)))
}
```

## 几个例子

```
fn foo(s: &str) {}

// 由于 String 实现了 Deref<Target=str>
let owned = "Hello".to_string();

// 因此下面的函数可以正常运行:
foo(&owned);

use std::rc::Rc;

fn foo(s: &str) {}

// String 实现了 Deref<Target=str>
let owned = "Hello".to_string();
// 且 Rc 智能指针可以被自动脱壳为内部的 `owned` 引用: &String，然后 &String 再自动解引用
为 &str
let counted = Rc::new(owned);

// 因此下面的函数可以正常运行:
foo(&counted);

struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&f).foo();
```

## 三种 Deref 转换

在之前，我们讲的都是不可变的 `Deref` 转换，实际上 Rust 还支持将一个可变的引用转换成另一个可变的引用以及将一个可变引用转换成不可变的引用，规则如下：

- 当 `T: Deref<Target=U>`，可以将 `&T` 转换成 `&U`，也就是我们之前看到的例子
- 当 `T: DerefMut<Target=U>`，可以将 `&mut T` 转换成 `&mut U`
- 当 `T: Deref<Target=U>`，可以将 `&mut T` 转换成 `&U`

来看一个关于 `DerefMut` 的例子：

```
struct MyBox<T> {
    v: T,
}

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox { v: x }
    }
}

use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.v
    }
}

use std::ops::DerefMut;

impl<T> DerefMut for MyBox<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.v
    }
}

fn main() {
    let mut s = MyBox::new(String::from("hello, "));
    display(&mut s)
}

fn display(s: &mut String) {
    s.push_str("world");
    println!("{}", s);
}
```

以上代码有几点值得注意：

- 要实现 `DerefMut` 必须要先实现 `Deref` 特征：`pub trait DerefMut: Deref`
- `T: DerefMut<Target=U>` 解读：将 `&mut T` 类型通过 `DerefMut` 特征的方法转换为 `&mut U` 类型，对应上例中，就是将 `&mut MyBox<String>` 转换为 `&mut String`

对于上述三条规则中的第三条，它比另外两条稍微复杂了点：Rust 可以把可变引用隐式的转换成不可变引用，但反之则不行。

如果从 Rust 的所有权和借用规则的角度考虑，当你拥有一个可变的引用，那该引用肯定是对应数据的唯一借用，那么此时将可变引用变成不可变引用并不会破坏借用规则；但是如果你拥有一个不可变引用，那同时可能还存在其它几个不可变的引用，如果此时将其中一个不可变引用转换成可变引用，就变成了可变引用与不可变引用的共存，最终破坏了借用规则。

## 总结

`Deref` 可以说是 Rust 中最常见的隐式类型转换，而且它可以连续的实现如 `Box<String> -> String -> &str` 的隐式转换，只要链条上的类型实现了 `Deref` 特征。

我们也可以为自己的类型实现 `Deref` 特征，但是原则上来说，只应该为自定义的智能指针实现 `Deref`。例如，虽然你可以为自己的自定义数组类型实现 `Deref` 以避免 `myArr.0[0]` 的使用形式，但是 Rust 官方并不推荐这么做，特别是在你开发三方库时。

# Drop 释放资源

在 Rust 中，我们之所以可以一拳打跑 GC 的同时一脚踢翻手动资源回收，主要就归功于 `Drop` 特征，同时它也是智能指针的必备特征之一。

## 学习目标

如何自动和手动释放资源及执行指定的收尾工作

## Rust 中的资源回收

在一些无 GC 语言中，程序员在一个变量无需再被使用时，需要手动释放它占用的内存资源，如果忘记了，那么就会发生内存泄漏，最终臭名昭著的 `oom` 问题可能就会发生。

而在 Rust 中，你可以指定在一个变量超出作用域时，执行一段特定的代码，最终编译器将帮你自动插入这段收尾代码。这样，就无需在每一个使用该变量的地方，都写一段代码来进行收尾工作和资源释放。不禁让人感叹，Rust 的大腿真粗，香！

没错，指定这样一段收尾工作靠的就是咱这章的主角 - `Drop` 特征。

# 一个不那么简单的 Drop 例子

```
struct HasDrop1;
struct HasDrop2;
impl Drop for HasDrop1 {
    fn drop(&mut self) {
        println!("Dropping HasDrop1!");
    }
}
impl Drop for HasDrop2 {
    fn drop(&mut self) {
        println!("Dropping HasDrop2!");
    }
}
struct HasTwoDrops {
    one: HasDrop1,
    two: HasDrop2,
}
impl Drop for HasTwoDrops {
    fn drop(&mut self) {
        println!("Dropping HasTwoDrops!");
    }
}

struct Foo;

impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping Foo!");
    }
}

fn main() {
    let _x = HasTwoDrops {
        two: HasDrop2,
        one: HasDrop1,
    };
    let _foo = Foo;
    println!("Running!");
}
```

上面代码虽然长，但是目的其实很单纯，就是为了观察不同情况下变量级别的、结构体内部字段的 Drop，有几点值得注意：

- Drop 特征中的 drop 方法借用了目标的可变引用，而不是拿走了所有权，这里先设置一个悬念，后边会讲
- 结构体中每个字段都有自己的 Drop

来看看输出：

```
Running!
Dropping Foo!
Dropping HasTwoDrops!
Dropping HasDrop1!
Dropping HasDrop2!
```

嗯，结果符合预期，每个资源都成功的执行了收尾工作，虽然 `println!` 这种收尾工作毫无意义 =\_=

## Drop 的顺序

观察以上输出，我们可以得出以下关于 Drop 顺序的结论

- **变量级别，按照逆序的方式**，`_x` 在 `_foo` 之前创建，因此 `_x` 在 `_foo` 之后被 drop
- **结构体内部，按照顺序的方式**，结构体 `_x` 中的字段按照定义中的顺序依次 drop

## 没有实现 Drop 的结构体

实际上，就算你不为 `_x` 结构体实现 Drop 特征，它内部的两个字段依然会调用 `drop`，移除以下代码，并观察输出：

```
impl Drop for HasTwoDrops {
    fn drop(&mut self) {
        println!("Dropping HasTwoDrops!");
    }
}
```

原因在于，Rust 自动为几乎所有类型都实现了 Drop 特征，因此就算你不手动为结构体实现 Drop，它依然会调用默认实现的 `drop` 函数，同时再调用每个字段的 `drop` 方法，最终打印出：

```
Dropping HasDrop1!
Dropping HasDrop2!
```

## 手动回收

当使用智能指针来管理锁的时候，你可能希望提前释放这个锁，然后让其它代码能及时获得锁，此时就需要提前去手动 `drop`。但是在之前我们提到一个悬念，`Drop::drop` 只是借用了目标值的可变引用，所以，就算你提前调用了 `drop`，后面的代码依然可以使用目标值，但是这就会访问一个并不存在的值，非常不安全，好在 Rust 会阻止你：

```
#[derive(Debug)]
struct Foo;

impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping Foo!")
    }
}

fn main() {
    let foo = Foo;
    foo.drop();
    println!("Running!:{:?}", foo);
}
```

报错如下：

```
error[E0040]: explicit use of destructor method
--> src/main.rs:37:9
37 |     foo.drop();
   |     ^^^^^^
   |     |
   |     explicit destructor calls not allowed
   | help: consider using `drop` function: `drop(foo)`
```

如上所示，编译器直接阻止了我们调用 Drop 特征的 drop 方法，原因是对于 Rust 而言，不允许显式地调用析构函数（这是一个用来清理实例的通用编程概念）。好在在报错的同时，编译器还给出了一个提示：使用 drop 函数。

针对编译器提示的 drop 函数，我们可以大胆推测下：它能够拿走目标值的所有权。现在来看看这个猜测正确与否，以下是 std::mem::drop 函数的签名：

```
pub fn drop<T>(_x: T)
```

如上所示， drop 函数确实拿走了目标值的所有权，来验证下：

```
fn main() {
    let foo = Foo;
    drop(foo);
    // 以下代码会报错：借用了所有权被转移的值
    // println!("Running!:{:?}", foo);
}
```

Bingo，完美拿走了所有权，而且这种实现保证了后续的使用必定会导致编译错误，因此非常安全！

细心的同学可能已经注意到，这里直接调用了 `drop` 函数，并没有引入任何模块信息，原因是该函数在 `std::prelude` 里。

## Drop 使用场景

对于 Drop 而言，主要有两个功能：

- 回收内存资源
- 执行一些收尾工作

对于第二点，在之前我们已经详细介绍过，因此这里主要对第一点进行下简单说明。

在绝大多数情况下，我们都无需手动去 `drop` 以回收内存资源，因为 Rust 会自动帮我们完成这些工作，它甚至会对复杂类型的每个字段都单独的调用 `drop` 进行回收！但是确实有极少数情况，需要你自己来回收资源的，例如文件描述符、网络 socket 等，当这些值超出作用域不再使用时，就需要进行关闭以释放相关的资源，在这些情况下，就需要使用者自己来解决 Drop 的问题。

## 互斥的 Copy 和 Drop

我们无法为一个类型同时实现 `Copy` 和 `Drop` 特征。因为实现了 `Copy` 的特征会被编译器隐式的复制，因此非常难以预测析构函数执行的时间和频率。因此这些实现了 `Copy` 的类型无法拥有析构函数。

```
#[derive(Copy)]
struct Foo;

impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping Foo!")
    }
}
```

以上代码报错如下：

```
error[E0184]: the trait `Copy` may not be implemented for this type; the type has a
destructor
--> src/main.rs:24:10
|
24 | #[derive(Copy)]
|     ^^^^ Copy not allowed on types with destructors
```

## 总结

`Drop` 可以用于许多方面，来使得资源清理及收尾工作变得方便和安全，甚至可以用其创建我们自己的内存分配器！通过 `Drop` 特征和 Rust 所有权系统，你无需担心之后的代码清理，Rust 会自动考虑这些问题。

我们也无需担心意外的清理掉仍在使用的值，这会造成编译器错误：所有权系统确保引用总是有效的，也会确保 `drop` 只会在值不再被使用时被调用一次。

# Rc 与 Arc

Rust 所有权机制要求一个值只能有一个所有者，在大多数情况下，都没有问题，但是考虑以下情况：

- 在图数据结构中，多个边可能会拥有同一个节点，该节点直到没有边指向它时，才应该被释放清理
- 在多线程中，多个线程可能会持有同一个数据，但是你受限于 Rust 的安全机制，无法同时获取该数据的可变引用

以上场景不是很常见，但是一旦遇到，就非常棘手，为了解决此类问题，Rust 在所有权机制之外又引入了额外的措施来简化相应的实现：通过引用计数的方式，允许一个数据资源在同一时刻拥有多个所有者。

这种实现机制就是 `Rc` 和 `Arc`，前者适用于单线程，后者适用于多线程。由于二者大部分情况下都相同，因此本章将以 `Rc` 作为讲解主体，对于 `Arc` 的不同之处，另外进行单独讲解。

## Rc<T>

引用计数(reference counting)，顾名思义，通过记录一个数据被引用的次数来确定该数据是否正在被使用。当引用次数归零时，就代表该数据不再被使用，因此可以被清理释放。

而 `Rc` 正是引用计数的英文缩写。当我们希望在堆上分配一个对象供程序的多个部分使用且无法确定哪个部分最后一个结束时，就可以使用 `Rc` 成为数据值的所有者，例如之前提到的多线程场景就非常适合。

下面是经典的所有权被转移导致报错的例子：

```
fn main() {
    let s = String::from("hello, world");
    // s在这里被转移给a
    let a = Box::new(s);
    // 报错！此处继续尝试将 s 转移给 b
    let b = Box::new(s);
}
```

使用 `Rc` 就可以轻易解决：

```
use std::rc::Rc;
fn main() {
    let a = Rc::new(String::from("hello, world"));
    let b = Rc::clone(&a);

    assert_eq!(2, Rc::strong_count(&a));
    assert_eq!(Rc::strong_count(&a), Rc::strong_count(&b))
}
```

以上代码我们使用 `Rc::new` 创建了一个新的 `Rc<String>` 智能指针并赋给变量 `a`，该指针指向底层的字符串数据。

智能指针 `Rc<T>` 在创建时，还会将引用计数加 1，此时获取引用计数的关联函数 `Rc::strong_count` 返回的值将是 1。

## Rc::clone

接着，我们又使用 `Rc::clone` 克隆了一份智能指针 `Rc<String>`，并将该智能指针的引用计数增加到 2。

由于 `a` 和 `b` 是同一个智能指针的两个副本，因此通过它们两个获取引用计数的结果都是 2。

不要被 `clone` 字样所迷惑，以为所有的 `clone` 都是深拷贝。这里的 `clone` **仅仅复制了智能指针并增加了引用计数，并没有克隆底层数据**，因此 `a` 和 `b` 是共享了底层的字符串 `s`，这种复制效率是非常高的。当然你也可以使用 `a.clone()` 的方式来克隆，但是从可读性角度，我们更加推荐 `Rc::clone` 的方式。

实际上在 Rust 中，还有不少 `clone` 都是浅拷贝，例如迭代器的克隆。

## 观察引用计数的变化

使用关联函数 `Rc::strong_count` 可以获取当前引用计数的值，我们来观察下引用计数如何随着变量声明、释放而变化：

```
use std::rc::Rc;
fn main() {
    let a = Rc::new(String::from("test ref counting"));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Rc::clone(&a);
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Rc::clone(&a);
        println!("count after creating c = {}", Rc::strong_count(&c));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

有几点值得注意：

- 由于变量 `c` 在语句块内部声明，当离开语句块时它会因为超出作用域而被释放，所以引用计数会减少 1，事实上这个得益于 `Rc<T>` 实现了 `Drop` 特征
- `a`、`b`、`c` 三个智能指针引用计数都是同样的，并且共享底层的数据，因此打印计数时用哪个都行
- 无法看到的是：当 `a`、`b` 超出作用域后，引用计数会变成 0，最终智能指针和它指向的底层字符串都会被清理释放

## 不可变引用

事实上，`Rc<T>` 是指向底层数据的不可变的引用，因此你无法通过它来修改数据，这也符合 Rust 的借用规则：要么存在多个不可变借用，要么只能存在一个可变借用。

但是实际开发中我们往往需要对数据进行修改，这时单独使用 `Rc<T>` 无法满足我们的需求，需要配合其它数据类型来一起使用，例如内部可变性的 `RefCell<T>` 类型以及互斥锁 `Mutex<T>`。事实上，在多线程编程中，`Arc` 跟 `Mutex` 锁的组合使用非常常见，它们既可以让我们在不同的线程中共享数据，又允许在各个线程中对其进行修改。

## 一个综合例子

考虑一个场景，有很多小工具，每个工具都有自己的主人，但是存在多个工具属于同一个主人的情况，此时使用 `Rc<T>` 就非常适合：

```

use std::rc::Rc;

struct Owner {
    name: String,
    // ...其它字段
}

struct Gadget {
    id: i32,
    owner: Rc<Owner>,
    // ...其它字段
}

fn main() {
    // 创建一个基于引用计数的 `Owner`。
    let gadget_owner: Rc<Owner> = Rc::new(Owner {
        name: "Gadget Man".to_string(),
    });

    // 创建两个不同的工具，它们属于同一个主人
    let gadget1 = Gadget {
        id: 1,
        owner: Rc::clone(&gadget_owner),
    };
    let gadget2 = Gadget {
        id: 2,
        owner: Rc::clone(&gadget_owner),
    };

    // 释放掉第一个 `Rc<Owner>`
    drop(gadget_owner);

    // 尽管在上面我们释放了 `gadget_owner`，但是依然可以在这里使用 `owner` 的信息
    // 原因是在 `drop` 之前，存在三个指向 `Gadget Man` 的智能指针引用，上面仅仅
    // `drop` 掉其中一个智能指针引用，而不是 `drop` 掉 `owner` 数据，外面还有两个
    // 引用指向底层的 `owner` 数据，引用计数尚未清零
    // 因此 `owner` 数据依然可以被使用
    println!("Gadget {} owned by {}", gadget1.id, gadget1.owner.name);
    println!("Gadget {} owned by {}", gadget2.id, gadget2.owner.name);

    // 在函数最后，`gadget1` 和 `gadget2` 也被释放，最终引用计数归零，随后底层
    // 数据也被清理释放
}

```

以上代码很好的展示了 `Rc<T>` 的用途，当然你也可以用借用的方式，但是实现起来就会复杂得多，而且随着 `Gadget` 在代码的各个地方使用，引用生命周期也将变得更加复杂，毕竟结构体中的引用类型，总是令人不那么愉快，对不？

## Rc 简单总结

- Rc/Arc 是不可变引用，你无法修改它指向的值，只能进行读取，如果要修改，需要配合后面章节的内部可变性 RefCell 或互斥锁 Mutex
- 一旦最后一个拥有者消失，则资源会自动被回收，这个生命周期是在编译期就确定下来的
- Rc 只能用于同一线程内部，想要用于线程之间的对象共享，你需要使用 Arc
- Rc<T> 是一个智能指针，实现了 Deref 特征，因此你无需先解开 Rc 指针，再使用里面的 T，而是可以直接使用 T，例如上例中的 gadget1.owner.name

## 多线程无力的 Rc<T>

来看看在多线程场景使用 Rc<T> 会如何：

```
use std::rc::Rc;
use std::thread;

fn main() {
    let s = Rc::new(String::from("多线程漫游者"));
    for _ in 0..10 {
        let s = Rc::clone(&s);
        let handle = thread::spawn(move || {
            println!("{}", s)
        });
    }
}
```

由于我们还没有学习多线程的章节，上面的例子就特地简化了相关的实现。首先通过 thread::spawn 创建一个线程，然后使用 move 关键字把克隆出的 s 的所有权转移到线程中。

能够实现这一点，完全得益于 Rc 带来的多所有权机制，但是以上代码会报错：

```
error[E0277]: `Rc<String>` cannot be sent between threads safely
```

表面原因是 Rc<T> 不能在线程间安全的传递，实际上是因为它没有实现 Send 特征，而该特征是恰恰是多线程间传递数据的关键，我们会在多线程章节中进行讲解。

当然，还有更深层的原因：由于 Rc<T> 需要管理引用计数，但是该计数器并没有使用任何并发原语，因此无法实现原子化的计数操作，最终会导致计数错误。

好在天无绝人之路，一起来看看 Rust 为我们提供的功能类似但是多线程安全的 Arc。

# Arc

`Arc` 是 `Atomic Rc` 的缩写，顾名思义：原子化的 `Rc<T>` 智能指针。原子化是一种并发原语，我们在后续章节会进行深入讲解，这里你只要知道它能保证我们的数据能够安全的在线程间共享即可。

## Arc 的性能损耗

你可能好奇，为何不直接使用 `Arc`，还要画蛇添足弄一个 `Rc`，还有 Rust 的基本数据类型、标准库数据类型为什么不自动实现原子化操作？这样就不存在线程不安全的问题了。

原因在于原子化或者其它锁虽然可以带来的线程安全，但是都会伴随着性能损耗，而且这种性能损耗还很小。因此 Rust 把这种选择权交给你，毕竟需要线程安全的代码其实占比并不高，大部分时候我们开发的程序都在一个线程内。

`Arc` 和 `Rc` 拥有完全一样的 API，修改起来很简单：

```
use std::sync::Arc;
use std::thread;

fn main() {
    let s = Arc::new(String::from("多线程漫游者"));
    for _ in 0..10 {
        let s = Arc::clone(&s);
        let handle = thread::spawn(move || {
            println!("{}", s)
        });
    }
}
```

对了，两者还有一点区别：`Arc` 和 `Rc` 并没有定义在同一个模块，前者通过 `use std::sync::Arc` 来引入，后者通过 `use std::rc::Rc`。

# 总结

在 Rust 中，所有权机制保证了一个数据只会有一个所有者，但如果你想要在图数据结构、多线程等场景中共享数据，这种机制会成为极大的阻碍。好在 Rust 为我们提供了智能指针 `Rc` 和 `Arc`，使用它们就能实现多个所有者共享一个数据的功能。

`Rc` 和 `Arc` 的区别在于，后者是原子化实现的引用计数，因此是线程安全的，可以用于多线程中共享数据。

这两者都是只读的，如果想要实现内部数据可修改，必须配合内部可变性 RefCell 或者互斥锁 Mutex 来一起使用。

# Cell 和 RefCell

Rust 的编译器之严格，可以说是举世无双。特别是在所有权方面，Rust 通过严格的规则来保证所有权和借用的正确性，最终为程序的安全保驾护航。

但是严格是一把双刃剑，带来安全提升的同时，损失了灵活性，有时甚至会让用户痛苦不堪、怨声载道。因此 Rust 提供了 `Cell` 和 `RefCell` 用于内部可变性，简而言之，可以在拥有不可变引用的同时修改目标数据，对于正常的代码实现来说，这个是不可能做到的（要么一个可变借用，要么多个不可变借用）。

---

内部可变性的实现是因为 Rust 使用了 `unsafe` 来做到这一点，但是对于使用者来说，这些都是透明的，因为这些不安全代码都被封装到了安全的 API 中

---

## Cell

`Cell` 和 `RefCell` 在功能上没有区别，区别在于 `Cell<T>` 适用于 `T` 实现 `Copy` 的情况：

```
use std::cell::Cell;
fn main() {
    let c = Cell::new("asdf");
    let one = c.get();
    c.set("qwer");
    let two = c.get();
    println!("{} , {}", one, two);
}
```

以上代码展示了 `Cell` 的基本用法，有几点值得注意：

- "asdf" 是 `&str` 类型，它实现了 `Copy` 特征
- `c.get` 用来取值，`c.set` 用来设置新值

取到值保存在 `one` 变量后，还能同时进行修改，这个违背了 Rust 的借用规则，但是由于 `Cell` 的存在，我们很优雅地做到了这一点，但是如果你尝试在 `Cell` 中存放 `String`：

```
let c = Cell::new(String::from("asdf"));
```

编译器会立刻报错，因为 `String` 没有实现 `Copy` 特征：

```
| pub struct String {  
|     ----- doesn't satisfy `String: Copy`  
|  
|= note: the following trait bounds were not satisfied:  
`String: Copy`
```

## RefCell

由于 `Cell` 类型针对的是实现了 `Copy` 特征的值类型，因此在实际开发中，`Cell` 使用的并不多，因为我们要解决的往往是可变、不可变引用共存导致的问题，此时就需要借助于 `RefCell` 来达成目的。

我们可以将所有权、借用规则与这些智能指针做一个对比：

Rust 规则	智能指针带来的额外规则
一个数据只有一个所有者	<code>Rc/Arc</code> 让一个数据可以拥有多个所有者
要么多个不可变借用，要么一个可变借用	<code>RefCell</code> 实现编译期可变、不可变引用共存
违背规则导致编译错误	违背规则导致运行时 <code>panic</code>

可以看出，`Rc/Arc` 和 `RefCell` 合在一起，解决了 Rust 中严苛的所有权和借用规则带来的某些场景下难使用的问题。但是它们并不是银弹，例如 `RefCell` 实际上并没有解决可变引用和引用可以共存的问题，只是将报错从编译期推迟到运行时，从编译器错误变成了 `panic` 异常：

```
use std::cell::RefCell;  
  
fn main() {  
    let s = RefCell::new(String::from("hello, world"));  
    let s1 = s.borrow();  
    let s2 = s.borrow_mut();  
  
    println!("{}{},{}", s1, s2);  
}
```

上面代码在编译期不会报任何错误，你可以顺利运行程序：

```
thread 'main' panicked at 'already borrowed: BorrowMutError', src/main.rs:6:16  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

但是依然会因为违背了借用规则导致了运行期 `panic`，这非常像中国的天网，它也许会被罪犯蒙蔽一时，但是并不会被蒙蔽一世，任何导致安全风险的存在都将不能被容忍，法网恢恢，疏而不漏。

## RefCell 为何存在

相信肯定有读者有疑问了，这么做有任何意义吗？还不如在编译期报错，至少能提前发现问题，而且性能还更好。

存在即合理，究其根因，在于 Rust 编译期的**宁可错杀，绝不放过**的原则，当编译器不能确定你的代码是否正确时，就统统会判定为错误，因此难免会导致一些误报。

而 RefCell 正是用于你确信代码是正确的，而编译器却发生了误判时。

对于大型的复杂程序，也可以选择使用 RefCell 来让事情简化。例如在 Rust 编译器的 `ctxt` 结构体中有大量的 RefCell 类型的 `map` 字段，主要的原因是：这些 `map` 会被分散在各个地方的代码片段所广泛使用或修改。由于这种分散在各处的使用方式，导致了管理可变和不可变成为一件非常复杂的任务（甚至不可能），你很容易就碰到编译器抛出来的各种错误。而且 RefCell 的运行时错误在这种情况下也变得非常可爱：一旦有人做了不正确的使用，代码会 `panic`，然后告诉我们哪些借用冲突了。

总之，当你确信编译器误报但不知道该如何解决时，或者你有一个引用类型，需要被四处使用和修改然后导致借用关系难以管理时，都可以优先考虑使用 RefCell。

## RefCell 简单总结

- 与 Cell 用于可 `Copy` 的值不同，RefCell 用于引用
- RefCell 只是将借用规则从编译期推迟到程序运行期，并不能帮你绕过这个规则
- RefCell 适用于编译期误报或者一个引用被在多处代码使用、修改以至于难于管理借用关系时
- 使用 RefCell 时，违背借用规则会导致运行期的 `panic`

## 选择 Cell 还是 RefCell

根据本文的内容，我们可以大概总结下两者的区别：

- Cell 只适用于 `copy` 类型，用于提供值，而 RefCell 用于提供引用
- Cell 不会 `panic`，而 RefCell 会

## 性能比较

Cell 没有额外的性能损耗，例如以下两段代码的性能其实是一致的：

```
// code snippet 1
let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());

// code snippet 2
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

虽然性能一致，但代码 1 拥有代码 2 不具有的优势：它能编译成功：)

与 `Cell` 的 `zero cost` 不同，`RefCell` 其实是有一点运行期开销的，原因是它包含了一个字节大小的“借用状态”指示器，该指示器在每次运行时借用时都会被修改，进而产生一点开销。

总之，当非要使用内部可变性时，首选 `Cell`，只有你的类型没有实现 `Copy` 时，才去选择 `RefCell`。

## 内部可变性

之前我们提到 `RefCell` 具有内部可变性，何为内部可变性？简单来说，对一个不可变的值进行可变借用，但这个并不符合 Rust 的基本借用规则：

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

上面的代码会报错，因为我们不能对一个不可变的值进行可变借用，这会破坏 Rust 的安全性保证，相反，你可以对一个可变值进行不可变借用。原因是：当值不可变时，可能会有多个不可变的引用指向它，此时若将其中一个修改为可变的，会造成可变引用与不可变引用共存的情况；而当值可变时，最多只会有一个可变引用指向它，将其修改为不可变，那么最终依然是只有一个不可变的引用指向它。

虽然基本借用规则是 Rust 的基石，然而在某些场景中，一个值可以在其方法内部被修改，同时对于其它代码不可变，是很有用的：

```

// 定义在外部库中的特征
pub trait Messenger {
    fn send(&self, msg: String);
}

// -----
// 我们的代码中的数据结构和实现
struct MsgQueue {
    msg_cache: Vec<String>,
}

impl Messenger for MsgQueue {
    fn send(&self, msg: String) {
        self.msg_cache.push(msg)
    }
}

```

如上所示，外部库中定义了一个消息发送器特征 `Messenger`，它只有一个发送消息的功能：`fn send(&self, msg: String)`，因为发送消息不需要修改自身，因此原作者在定义时，使用了 `&self` 的不可变借用，这个无可厚非。

我们要在自己的代码中使用该特征实现一个异步消息队列，出于性能的考虑，消息先写到本地缓存(内存)中，然后批量发送出去，因此在 `send` 方法中，需要将消息先行插入到本地缓存 `msg_cache` 中。但是问题来了，该 `send` 方法的签名是 `&self`，因此上述代码会报错：

```

error[E0596]: cannot borrow `self.msg_cache` as mutable, as it is behind a `&` reference
--> src/main.rs:11:9
 |
2 |     fn send(&self, msg: String);
|         ----- help: consider changing that to be a mutable reference: `&mut self`
|
...
11 |         self.msg_cache.push(msg)
|             ^^^^^^^^^^^^^^^^^ `self` is a `&` reference, so the data it refers to
|             cannot be borrowed as mutable

```

在报错的同时，编译器大聪明还善意地给出了提示：将 `&self` 修改为 `&mut self`，但是。。。我们实现的特征是定义在外部库中，因此该签名根本不能修改。值此危急关头，`RefCell` 闪亮登场：

```

use std::cell::RefCell;
pub trait Messenger {
    fn send(&self, msg: String);
}

pub struct MsgQueue {
    msg_cache: RefCell<Vec<String>>,
}

impl Messenger for MsgQueue {
    fn send(&self, msg: String) {
        self.msg_cache.borrow_mut().push(msg)
    }
}

fn main() {
    let mq = MsgQueue {
        msg_cache: RefCell::new(Vec::new()),
    };
    mq.send("hello, world".to_string());
}

```

这个 MQ 功能很弱，但是并不妨碍我们演示内部可变性的核心用法：通过包裹一层 `RefCell`，成功的让 `&self` 中的 `msg_cache` 成为一个可变值，然后实现对其的修改。

## Rc + RefCell 组合使用

在 Rust 中，一个常见的组合就是 `Rc` 和 `RefCell` 在一起使用，前者可以实现一个数据拥有多个所有者，后者可以实现数据的可变性：

```

use std::cell::RefCell;
use std::rc::Rc;
fn main() {
    let s = Rc::new(RefCell::new("我很善变，还拥有多个主人".to_string()));

    let s1 = s.clone();
    let s2 = s.clone();
    // let mut s2 = s.borrow_mut();
    s2.borrow_mut().push_str(", oh yeah!");

    println!(":{}\n:{}\n:{}", s, s1, s2);
}

```

上面代码中，我们使用 `RefCell<String>` 包裹一个字符串，同时通过 `Rc` 创建了它的三个所有者：`s`、`s1` 和 `s2`，并且通过其中一个所有者 `s2` 对字符串内容进行了修改。

由于 `Rc` 的所有者们共享同一个底层的数据，因此当一个所有者修改了数据时，会导致全部所有者持有的数据都发生了变化。

程序的运行结果也在预料之中：

```
RefCell { value: "我很善变, 还拥有多个主人, oh yeah!" }
RefCell { value: "我很善变, 还拥有多个主人, oh yeah!" }
RefCell { value: "我很善变, 还拥有多个主人, oh yeah!" }
```

## 性能损耗

相信这两者组合在一起使用时，很多人会好奇到底性能如何，下面我们来简单分析下。

首先给出一个大概的结论，这两者结合在一起使用的性能其实非常高，大致相当于没有线程安全版本的 C++ `std::shared_ptr` 指针，事实上，C++ 这个指针的主要开销也在于原子性这个并发原语上，毕竟线程安全在哪个语言中开销都不小。

## 内存损耗

两者结合的数据结构与下面类似：

```
struct Wrapper<T> {
    // Rc
    strong_count: usize,
    weak_count: usize,

    // Refcell
    borrow_count: isize,

    // 包裹的数据
    item: T,
}
```

从上面可以看出，从对内存的影响来看，仅仅多分配了三个 `usize/isize`，并没有其它额外的负担。

## CPU 损耗

从 CPU 来看，损耗如下：

- 对 `Rc<T>` 解引用是免费的（编译期），但是 `*` 带来的间接取值并不免费
- 克隆 `Rc<T>` 需要将当前的引用计数跟 `0` 和 `usize::Max` 进行一次比较，然后将计数值加 1
- 释放 (drop) `Rc<T>` 需要将计数值减 1，然后跟 `0` 进行一次比较
- 对 `RefCell` 进行不可变借用，需要将 `isize` 类型的借用计数加 1，然后跟 `0` 进行比较

- 对 `RefCell` 的不可变借用进行释放，需要将 `isize` 减 1
- 对 `RefCell` 的可变借用大致流程跟上面差不多，但是需要先跟 `0` 比较，然后再减 1
- 对 `RefCell` 的可变借用进行释放，需要将 `isize` 加 1

其实这些细节不必过于关注，只要知道 CPU 消耗也非常低，甚至编译器还会对此进行进一步优化！

## CPU 缓存 Miss

唯一需要担心的可能就是这种组合数据结构对于 CPU 缓存是否亲和，这个我们无法证明，只能提出来存在这个可能性，最终的性能影响还需要在实际场景中进行测试。

总之，分析这两者组合的性能还挺复杂的，大概总结下：

- 从表面来看，它们带来的内存和 CPU 损耗都不大
- 但是由于 `Rc` 额外的引入了一次间接取值（`*`），在少数场景下可能会造成性能上的显著损失
- CPU 缓存可能也不够亲和

## 通过 `Cell::from_mut` 解决借用冲突

在 Rust 1.37 版本中新增了两个非常实用的方法：

- `Cell::from_mut`，该方法将 `&mut T` 转为 `&Cell<T>`
- `Cell::as_slice_of_cells`，该方法将 `&Cell<[T]>` 转为 `&[Cell<T>]`

这里我们不做深入的介绍，但是来看看如何使用这两个方法来解决一个常见的借用冲突问题：

```
fn is_even(i: i32) -> bool {
    i % 2 == 0
}

fn retain_even(nums: &mut Vec<i32>) {
    let mut i = 0;
    for num in nums.iter().filter(|&num| is_even(*num)) {
        nums[i] = *num;
        i += 1;
    }
    nums.truncate(i);
}
```

以上代码会报错：

```
error[E0502]: cannot borrow `*nums` as mutable because it is also borrowed as
immutable
--> src/main.rs:8:9
7 |     for num in nums.iter().filter(|&num| is_even(*num)) {
   |     -----|
   |     |
   |     immutable borrow occurs here
   |     immutable borrow later used here
8 |     nums[i] = *num;
   |     ^^^^ mutable borrow occurs here
```

很明显，报错是因为同时借用了不可变与可变引用，你可以通过索引的方式来避免这个问题：

```
fn retain_even(nums: &mut Vec<i32>) {
    let mut i = 0;
    for j in 0..nums.len() {
        if is_even(nums[j]) {
            nums[i] = nums[j];
            i += 1;
        }
    }
    nums.truncate(i);
}
```

但是这样就违背我们的初衷了，毕竟迭代器会让代码更加简洁，那么还有其它的办法吗？

这时就可以使用 `Cell` 新增的这两个方法：

```
use std::cell::Cell;

fn retain_even(nums: &mut Vec<i32>) {
    let slice: &[Cell<i32>] = Cell::from_mut(&mut nums[..])
        .as_slice_of_cells();

    let mut i = 0;
    for num in slice.iter().filter(|num| is_even(num.get())) {
        slice[i].set(num.get());
        i += 1;
    }

    nums.truncate(i);
}
```

此时代码将不会报错，因为 `Cell` 上的 `set` 方法获取的是不可变引用 `pub fn set(&self, val: T)`。

当然，以上代码的本质还是对 `Cell` 的运用，只不过这两个方法可以很方便的帮我们把 `&mut [T]` 类型转换成 `&[Cell<T>]` 类型。

## 总结

`Cell` 和 `RefCell` 都为我们带来了内部可变性这个重要特性，同时还将借用规则的检查从编译期推迟到运行期，但是这个检查并不能被绕过，该来早晚还是会来，`RefCell` 在运行期的报错会造成 `panic`。

`RefCell` 适用于编译器误报或者一个引用被在多个代码中使用、修改以至于难于管理借用关系时，还有就是需要内部可变性时。

从性能上看，`RefCell` 由于是非线程安全的，因此无需保证原子性，性能虽然有一点损耗，但是依然非常好，而 `Cell` 则完全不存在任何额外的性能损耗。

`Rc` 跟 `RefCell` 结合使用可以实现多个所有者共享同一份数据，非常好用，但是潜在的性能损耗也要考虑进去，建议对于热点代码使用时，做好 `benchmark`。

# 循环引用与自引用

实现一个链表是学习各大编程语言的常用技巧，但是在 Rust 中实现链表意味着…Hell，是的，你没看错，Welcome to hell。

链表在 Rust 中之所以这么难，完全是因为循环引用和自引用的问题引起的，这两个问题可以说综合了 Rust 的很多难点，难出了新高度，因此本书专门开辟一章，分为上下两篇，试图彻底解决这两个老大难。

本章难度较高，但是非常值得深入阅读，它会让你对 Rust 的理解上升到一个新的境界。

# Weak 与循环引用

Rust 的安全性是众所周知的，但是不代表它不会内存泄漏。一个典型的例子就是同时使用 `Rc<T>` 和 `RefCell<T>` 创建循环引用，最终这些引用的计数都无法被归零，因此 `Rc<T>` 拥有的值也不会被释放清理。

## 何为循环引用

关于内存泄漏，如果你没有充足的 Rust 经验，可能都无法造出一份代码来再现它：

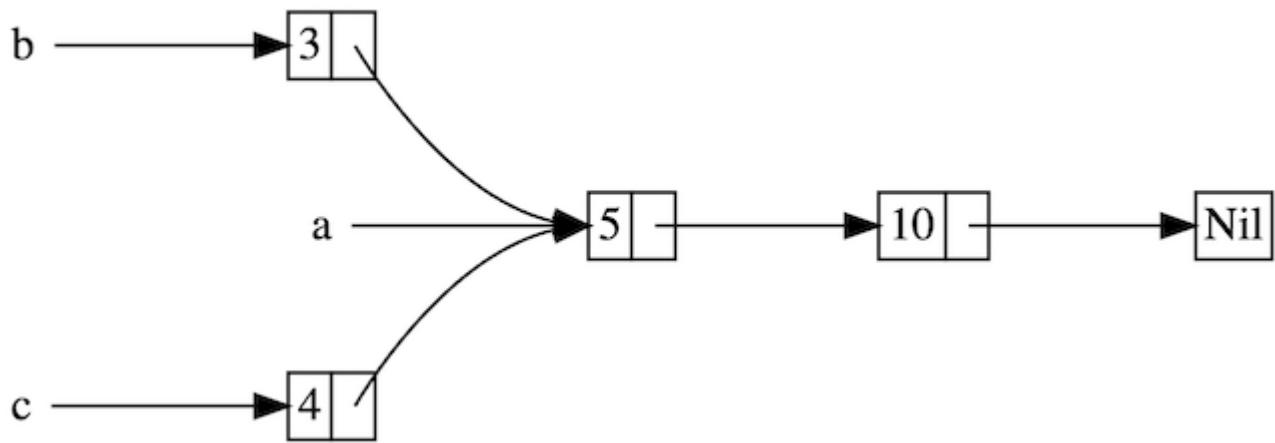
```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

这里我们创建一个有些复杂的枚举类型 `List`，这个类型很有意思，它的每个值都指向了另一个 `List`，此外，得益于 `Rc` 的使用还允许多个值指向一个 `List`：



如上图所示，每个矩形框节点都是一个 `List` 类型，它们或者是拥有值且指向另一个 `List` 的 `Cons`，或者是一个没有值的终结点 `Nil`。同时，由于 `RefCell` 的使用，每个 `List` 所指向的 `List` 还能够被修改。

下面来使用一下这个复杂的 `List` 枚举：

```

fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::newNil())));
    println!("a的初始化rc计数 = {}", Rc::strong_count(&a));
    println!("a指向的节点 = {:?}", a.tail());

    // 创建`b`到`a`的引用
    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("在b创建后, a的rc计数 = {}", Rc::strong_count(&a));
    println!("b的初始化rc计数 = {}", Rc::strong_count(&b));
    println!("b指向的节点 = {:?}", b.tail());

    // 利用RefCell的可变性, 创建了`a`到`b`的引用
    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("在更改a后, b的rc计数 = {}", Rc::strong_count(&b));
    println!("在更改a后, a的rc计数 = {}", Rc::strong_count(&a));

    // 下面一行println!将导致循环引用
    // 我们可怜的8MB大小的main线程栈空间将被它冲垮, 最终造成栈溢出
    // println!("a next item = {:?}", a.tail());
}

```

这个类型定义看着复杂，使用起来更复杂！不过排除这些因素，我们可以清晰看出：

1. 在创建了 a 后，紧接着就使用 a 创建了 b，因此 b 引用了 a
2. 然后我们又利用 Rc 克隆了 b，然后通过 RefCell 的可变性，让 a 引用了 b

至此我们成功创建了循环引用 a -> b -> a -> b ....

先来观察下引用计数：

a的初始化rc计数 = 1

a指向的节点 = Some(RefCell { value: Nil })

在b创建后，a的rc计数 = 2

b的初始化rc计数 = 1

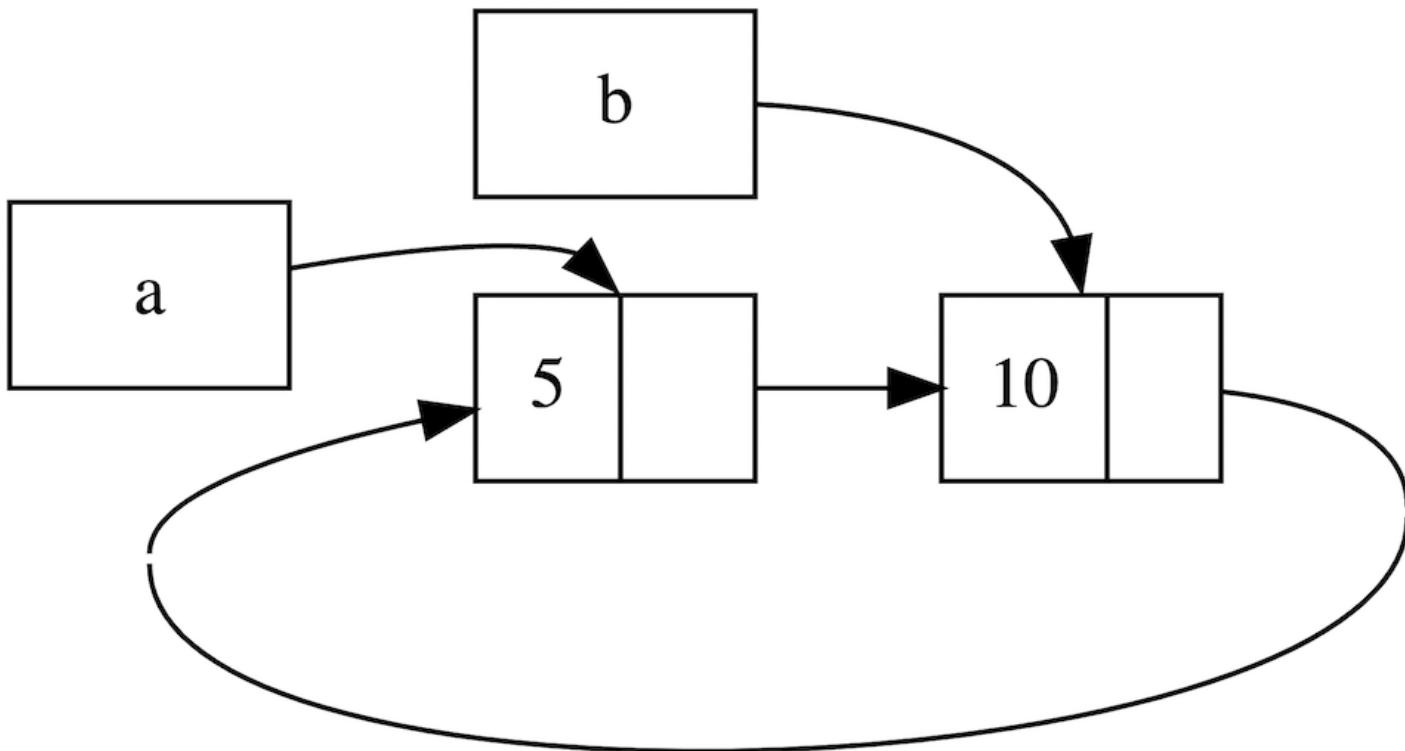
b指向的节点 = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })

在更改a后，b的rc计数 = 2

在更改a后，a的rc计数 = 2

在 main 函数结束前，a 和 b 的引用计数均是 2，随后 b 触发 Drop，此时引用计数会变为 1，并不会归 0，因此 b 所指向内存不会被释放，同理可得 a 指向的内存也不会被释放，最终发生了内存泄漏。

下面一张图很好的展示了这种引用循环关系：



现在我们还需要轻轻的推一下，让塔米诺骨牌轰然倒塌。反注释最后一行代码，试着运行下：

```
RefCell { value: Cons(5, RefCell { value: Cons(10, RefCell { value: Cons(5, RefCell {  
value: Cons(10, RefCell { value: Cons(5, RefCell { value: Cons(10, RefCell {  
...无穷无尽  
thread 'main' has overflowed its stack  
fatal runtime error: stack overflow
```

通过 `a.tail` 的调用，Rust 试图打印出 `a -> b -> a ...` 的所有内容，但是在不懈的努力后，`main` 线程终于不堪重负，发生了[栈溢出](#)。

以上的代码可能并不会造成什么大的问题，但是在一个更加复杂的程序中，类似的问题可能会造成你的程序不断地分配内存、泄漏内存，最终程序会不幸[OOM](#)，当然这其中的 CPU 损耗也不可小觑。

总之，创建循环引用并不简单，但是也并不是完全遇不到，当你使用 `RefCell< Rc< T > >` 或者类似的类型嵌套组合（具备内部可变性和引用计数）时，就要打起万分精神，前面可能是深渊！

那么问题来了？如果我们确实需要实现上面的功能，该怎么办？答案是使用 `Weak`。

## Weak

`Weak` 非常类似于 `Rc`，但是与 `Rc` 持有所有权不同，`Weak` 不持有所有权，它仅仅保存一份指向数据的弱引用：如果你想要访问数据，需要通过 `Weak` 指针的 `upgrade` 方法实现，该方法返回一个类型为 `Option< Rc< T > >` 的值。

看到这个返回，相信大家就懂了：何为弱引用？就是**不保证引用关系依然存在**，如果不存在，就返回一个 `None`！

因为 `Weak` 引用不计入所有权，因此它**无法阻止所引用的内存值被释放掉**，而且 `Weak` 本身不对值的存在性做任何担保，引用的值还存在就返回 `Some`，不存在就返回 `None`。

### Weak 与 Rc 对比

我们来将 `Weak` 与 `Rc` 进行以下简单对比：

Weak	Rc
不计数	引用计数
不拥有所有权	拥有值的所有权
不阻止值被释放(drop)	所有权计数归零，才能 drop
引用的值存在返回 <code>Some</code> ，不存在返回 <code>None</code>	引用的值必定存在

Weak	Rc
通过 <code>upgrade</code> 取到 <code>Option&lt;Rc&lt;T&gt;&gt;</code> ，然后再取值	通过 <code>Deref</code> 自动解引用，取值无需任何操作

通过这个对比，可以非常清晰的看出 `Weak` 为何这么弱，而这种弱恰恰非常适合我们实现以下的场景：

- 持有一个 `Rc` 对象的临时引用，并且不在乎引用的值是否依然存在
- 阻止 `Rc` 导致的循环引用，因为 `Rc` 的所有权机制，会导致多个 `Rc` 都无法计数归零

使用方式简单总结下：**对于父子引用关系，可以让父节点通过 `Rc` 来引用子节点，然后让子节点通过 `Weak` 来引用父节点。**

## Weak 总结

因为 `Weak` 本身并不是很好理解，因此我们再来帮大家梳理总结下，然后再通过一个例子，来彻底掌握。

`Weak` 通过 `use std::rc::Weak` 来引入，它具有以下特点：

- 可访问，但没有所有权，不增加引用计数，因此不会影响被引用值的释放回收
- 可由 `Rc<T>` 调用 `downgrade` 方法转换成 `Weak<T>`
- `Weak<T>` 可使用 `upgrade` 方法转换成 `Option<Rc<T>>`，如果资源已经被释放，则 `Option` 的值是 `None`
- 常用于解决循环引用的问题

一个简单的例子：

```
use std::rc::Rc;
fn main() {
    // 创建Rc, 持有一个值5
    let five = Rc::new(5);

    // 通过Rc, 创建一个Weak指针
    let weak_five = Rc::downgrade(&five);

    // Weak引用的资源依然存在, 取到值5
    let strong_five: Option<Rc<_>> = weak_five.upgrade();
    assert_eq!(*strong_five.unwrap(), 5);

    // 手动释放资源`five`
    drop(five);

    // Weak引用的资源已不存在, 因此返回None
    let strong_five: Option<Rc<_>> = weak_five.upgrade();
    assert_eq!(strong_five, None);
}
```

需要承认的是，使用 `Weak` 让 Rust 本来就堪忧的代码可读性又下降了不少，但是。。。真香，因为可以解决循环引用了。

## 使用 `Weak` 解决循环引用

理论知识已经足够，现在用两个例子来模拟下真实场景下可能会遇到的循环引用。

### 工具间的故事

工具间里，每个工具都有其主人，且多个工具可以拥有一个主人；同时一个主人也可以拥有多个工具，在这种场景下，就很容易形成循环引用，好在我们有 `Weak`：

```
use std::rc::Rc;
use std::rc::Weak;
use std::cell::RefCell;

// 主人
struct Owner {
    name: String,
    gadgets: RefCell<Vec<Weak<Gadget>>>,
}

// 工具
struct Gadget {
    id: i32,
    owner: Rc<Owner>,
}

fn main() {
    // 创建一个 Owner
    // 需要注意，该 Owner 也拥有多个 `gadgets`。
    let gadget_owner : Rc<Owner> = Rc::new(
        Owner {
            name: "Gadget Man".to_string(),
            gadgets: RefCell::new(Vec::new()),
        }
    );

    // 创建工具，同时与主人进行关联：创建两个 gadget，他们分别持有 gadget_owner 的一个引用。
    let gadget1 = Rc::new(Gadget{id: 1, owner: gadget_owner.clone()});
    let gadget2 = Rc::new(Gadget{id: 2, owner: gadget_owner.clone()});

    // 为主任更新它所拥有的工具
    // 因为之前使用了 `Rc`，现在必须要使用 `Weak`，否则就会循环引用
    gadget_owner.gadgets.borrow_mut().push(Rc::downgrade(&gadget1));
    gadget_owner.gadgets.borrow_mut().push(Rc::downgrade(&gadget2));

    // 遍历 gadget_owner 的 gadgets 字段
    for gadget_opt in gadget_owner.gadgets.borrow().iter() {

        // gadget_opt 是一个 Weak<Gadget>。因为 weak 指针不能保证他所引用的对象
        // 仍然存在。所以我们需要显式的调用 upgrade() 来通过其返回值(Option<_>)来判
        // 断其所指向的对象是否存在。
        // 当然，Option 为 None 的时候这个引用原对象就不存在了。
        let gadget = gadget_opt.upgrade().unwrap();
        println!("Gadget {} owned by {}", gadget.id, gadget.owner.name);
    }

    // 在 main 函数的最后，gadget_owner, gadget1 和 gadget2 都被销毁。
    // 具体是，因为这几个结构体之间没有了强引用(`Rc<T>`)，所以，当他们销毁的时候。
    // 首先 gadget2 和 gadget1 被销毁。
    // 然后因为 gadget_owner 的引用数量为 0，所以这个对象可以被销毁了。
}
```

```
// 循环引用问题也就避免了  
}
```

## tree 数据结构

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );
    }

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}

println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
```

```
    );  
}
```

这个例子就留给读者自己解读和分析，我们就不画蛇添足了：）

## unsafe 解决循环引用

除了使用 Rust 标准库提供的这些类型，你还可以使用 `unsafe` 里的裸指针来解决这些棘手的问题，但是由于我们还没有讲解 `unsafe`，因此这里就不进行展开，只附上[源码链接](#)，挺长的，需要耐心 o\_o

虽然 `unsafe` 不安全，但是在各种库的代码中依然很常见用它来实现自引用结构，主要优点如下：

- 性能高，毕竟直接用裸指针操作
- 代码更简单更符合直觉：对比下 `Option<Rc<RefCell<Node>>>`

## 总结

本文深入讲解了何为循环引用以及如何使用 `Weak` 来解决，同时还结合 `Rc`、`RefCell`、`Weak` 等实现了两个有实战价值的例子，让大家对智能指针的使用更加融会贯通。

至此，智能指针一章即将结束（严格来说还有一个 `Mutex` 放在多线程一章讲解），而 Rust 语言本身的学习之旅也即将结束，后面我们将深入多线程、项目工程、应用实践、性能分析等特色专题，来一睹 Rust 在这些领域的风采。

## 结构体自引用

结构体自引用在 Rust 中是一个众所周知的难题，而且众说纷纭，也没有一篇文章能把相关的话题讲透，那本文就王婆卖瓜，来试试看能不能讲透这一块儿内容，让读者大大们舒心。

### 平平无奇的自引用

可能也有不少人第一次听说自引用结构体，那咱们先来看看它们长啥样。

```
struct SelfRef<'a> {
    value: String,
    // 该引用指向上面的value
    pointer_to_value: &'a str,
}
```

以上就是一个很简单的自引用结构体，看上去好像没什么，那来试着运行下：

```
fn main(){
    let s = "aaa".to_string();
    let v = SelfRef {
        value: s,
        pointer_to_value: &s
    };
}
```

运行后报错：

```
let v = SelfRef {
12 |     value: s,
|         - value moved here
13 |     pointer_to_value: &s
|             ^^ value borrowed here after move
```

因为我们试图同时使用值和值的引用，最终所有权转移和借用一起发生了。所以，这个问题貌似并没有那么好解决，不信你可以回想下自己具有的知识，是否可以解决？

# 使用 Option

最简单的方式就是使用 Option 分两步来实现：

```
#[derive(Debug)]
struct WhatAboutThis<'a> {
    name: String,
    nickname: Option<&'a str>,
}

fn main() {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.nickname = Some(&tricky.name[..4]);

    println!("{}: {}", tricky);
}
```

在某种程度上来说，Option 这个方法可以工作，但是这个方法的限制较多，例如从一个函数创建并返回它是不可能的：

```
fn creator<'a>() -> WhatAboutThis<'a> {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.nickname = Some(&tricky.name[..4]);

    tricky
}
```

报错如下：

```
error[E0515]: cannot return value referencing local data `tricky.name`
--> src/main.rs:24:5
|
22 |     tricky.nickname = Some(&tricky.name[..4]);
|                         ----- `tricky.name` is borrowed here
23 |
24 |     tricky
|     ^^^^^^ returns a value referencing data owned by the current function
```

其实从函数签名就能看出来端倪，'a 生命周期是凭空产生的！

如果是通过方法使用，你需要一个无用 `&'a self` 生命周期标识，一旦有了这个标识，代码将变得更加受限，你将很容易就获得借用错误，就连 NLL 规则都没用：

```
#[derive(Debug)]
struct WhatAboutThis<'a> {
    name: String,
    nickname: Option<&'a str>,
}

impl<'a> WhatAboutThis<'a> {
    fn tie_the_knot(&'a mut self) {
        self.nickname = Some(&self.name[..4]);
    }
}

fn main() {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.tie_the_knot();

    // cannot borrow `tricky` as immutable because it is also borrowed as mutable
    // println!("{}:?", tricky);
}
```

## unsafe 实现

既然借用规则妨碍了我们，那就一脚踢开：

```

#[derive(Debug)]
struct SelfRef {
    value: String,
    pointer_to_value: *const String,
}

impl SelfRef {
    fn new(txt: &str) -> Self {
        SelfRef {
            value: String::from(txt),
            pointer_to_value: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.value;
        self.pointer_to_value = self_ref;
    }

    fn value(&self) -> &str {
        &self.value
    }

    fn pointer_to_value(&self) -> &String {
        assert!(!self.pointer_to_value.is_null(),
                "Test::b called without Test::init being called first");
        unsafe { &*(self.pointer_to_value) }
    }
}

fn main() {
    let mut t = SelfRef::new("hello");
    t.init();
    // 打印值和指针地址
    println!("{} , {:p}" , t.value() , t.pointer_to_value());
}

```

在这里，我们在 `pointer_to_value` 中直接存储裸指针，而不是 Rust 的引用，因此不再受到 Rust 借用规则和生命周期的限制，而且实现起来非常清晰、简洁。但是缺点就是，通过指针获取值时需要使用 `unsafe` 代码。

当然，上面的代码你还能通过裸指针来修改 `String`，但是需要将 `*const` 修改为 `*mut`：

```

#[derive(Debug)]
struct SelfRef {
    value: String,
    pointer_to_value: *mut String,
}

impl SelfRef {
    fn new(txt: &str) -> Self {
        SelfRef {
            value: String::from(txt),
            pointer_to_value: std::ptr::null_mut(),
        }
    }

    fn init(&mut self) {
        let self_ref: *mut String = &mut self.value;
        self.pointer_to_value = self_ref;
    }

    fn value(&self) -> &str {
        &self.value
    }

    fn pointer_to_value(&self) -> &String {
        assert!(!self.pointer_to_value.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.pointer_to_value) }
    }
}

fn main() {
    let mut t = SelfRef::new("hello");
    t.init();
    println!("{} , {:p}", t.value(), t.pointer_to_value());

    t.value.push_str(", world");
    unsafe {
        (&mut *t.pointer_to_value).push_str("!");
    }

    println!("{} , {:p}", t.value(), t.pointer_to_value());
}

```

运行后输出：

```

hello, 0x16f3aec70
hello, world!, 0x16f3aec70

```

上面的 `unsafe` 虽然简单好用，但是它不太安全，是否还有其他选择？还真的有，那就是 `Pin`。

## 无法被移动的 Pin

Pin 在后续章节会深入讲解，目前你只需要知道它可以固定住一个值，防止该值在内存中被移动。

通过开头我们知道，自引用最麻烦的就是创建引用的同时，值的所有权会被转移，而通过 Pin 就可以很好的防止这一点：

```

use std::marker::PhantomPinned;
use std::pin::Pin;
use std::ptr::NonNull;

// 下面是一个自引用数据结构体，因为 slice 字段是一个指针，指向了 data 字段
// 我们无法使用普通引用来实现，因为违背了 Rust 的编译规则
// 因此，这里我们使用了一个裸指针，通过 NonNull 来确保它不会为 null
struct Unmovable {
    data: String,
    slice: NonNull<String>,
    _pin: PhantomPinned,
}

impl Unmovable {
    // 为了确保函数返回时数据的所有权不会被转移，我们将它放在堆上，唯一的访问方式就是通过指针
    fn new(data: String) -> Pin<Box<Self>> {
        let res = Unmovable {
            data,
            // 只有在数据到位时，才创建指针，否则数据会在开始之前就被转移所有权
            slice: NonNull::dangling(),
            _pin: PhantomPinned,
        };
        let mut boxed = Box::pin(res);

        let slice = NonNull::from(&boxed.data);
        // 这里其实安全的，因为修改一个字段不会转移整个结构体的所有权
        unsafe {
            let mut_ref: Pin<&mut Self> = Pin::as_mut(&mut boxed);
            Pin::get_unchecked_mut(mut_ref).slice = slice;
        }
        boxed
    }
}

fn main() {
    let unmoved = Unmovable::new("hello".to_string());
    // 只要结构体没有被转移，那指针就应该指向正确的位置，而且我们可以随意移动指针
    let mut still_unmoved = unmoved;
    assert_eq!(still_unmoved.slice, NonNull::from(&still_unmoved.data));

    // 因为我们的类型没有实现 `Unpin` 特征，下面这段代码将无法编译
    // let mut new_unmoved = Unmovable::new("world".to_string());
    // std::mem::swap(&mut *still_unmoved, &mut *new_unmoved);
}

```

上面的代码也非常清晰，虽然使用了 `unsafe`，其实更多的是无奈之举，跟之前的 `unsafe` 实现完全不可同日而语。

其实 `Pin` 在这里并没有魔法，它也并不是实现自引用类型的主要原因，最关键的还是里面的裸指针的使用，而 `Pin` 起到的作用就是确保我们的值不会被移走，否则指针就会指向一个错误的地址！

## 使用 ouroboros

对于自引用结构体，三方库也有支持的，其中一个就是 [ouroboros](#)，当然它也有自己的限制，我们后面会提到，先来看看该如何使用：

```
use ouroboros::self_referencing;

#[self_referencing]
struct SelfRef {
    value: String,
    #[borrows(value)]
    pointer_to_value: &'this str,
}

fn main(){
    let v = SelfRefBuilder {
        value: "aaa".to_string(),
        pointer_to_value_builder: |value: &String| value,
    }.build();

    // 借用value值
    let s = v.borrow_value();
    // 借用指针
    let p = v.borrow_pointer_to_value();
    // value值和指针指向的值相等
    assert_eq!(s, *p);
}
```

可以看到，`ouroboros` 使用起来并不复杂，就是需要你去按照它的方式创建结构体和引用类型：

`SelfRef` 变成 `SelfRefBuilder`，引用字段从 `pointer_to_value` 变成  
`pointer_to_value_builder`，并且连类型都变了。

在使用时，通过 `borrow_value` 来借用 `value` 的值，通过 `borrow_pointer_to_value` 来借用  
`pointer_to_value` 这个指针。

看上去很美好对吧？但是你可以尝试着去修改 `String` 字符串的值试试，`ouroboros` 限制还是较多的，  
但是对于基本类型依然是支持的不错，以下例子来源于官方：

```

use ouroboros::self_referencing;

#[self_referencing]
struct MyStruct {
    int_data: i32,
    float_data: f32,
    #[borrows(int_data)]
    int_reference: &'this i32,
    #[borrows(mut float_data)]
    float_reference: &'this mut f32,
}

fn main() {
    let mut my_value = MyStructBuilder {
        int_data: 42,
        float_data: 3.14,
        int_reference_builder: |int_data: &i32| int_data,
        float_reference_builder: |float_data: &mut f32| float_data,
    }.build();

    // Prints 42
    println!("{:?}", my_value.borrow_int_data());
    // Prints 3.14
    println!("{:?}", my_value.borrow_float_reference());
    // Sets the value of float_data to 84.0
    my_value.with_mut(|fields| {
        **fields.float_reference = (**fields.int_reference as f32) * 2.0;
    });

    // We can hold on to this reference...
    let int_ref = *my_value.borrow_int_reference();
    println!("{:?}", int_ref);
    // As long as the struct is still alive.
    drop(my_value);
    // This will cause an error!
    // println!("{:?}", int_ref);
}

```

总之，使用这个库前，强烈建议看一些官方的例子中支持什么样的类型和 API，如果能满足的你的需求，就果断使用它，如果不能满足，就继续往下看。

只能说，它确实帮助我们解决了问题，但是一个是破坏了原有的结构，另外就是并不是所有数据类型都支持：它需要目标值的内存地址不会改变，因此 `Vec` 动态数组就不适合，因为当内存空间不够时，Rust 会重新分配一块空间来存放该数组，这会导致内存地址的改变。

类似的库还有：

- [rental](#)，这个库其实是最有名的，但是好像不再维护了，用倒是没问题
- [owning-ref](#)，将所有者和它的引用绑定到一个封装类型

这三个库，各有各的特点，也各有各的缺陷，建议大家需要时，一定要仔细调研，并且写 demo 进行测试，不可大意。

---

rental 虽然不怎么维护，但是可能依然是这三个里面最强大的，而且网上的用例也比较多，容易找到参考代码

---

## Rc + RefCell 或 Arc + Mutex

类似于循环引用的解决方式，自引用也可以用这种组合来解决，但是会导致代码的类型标识到处都是，大大的影响了可读性。

## 终极大法

如果两个放在一起会报错，那就分开它们。对，终极大法就这么简单，当然思路上的简单不代表实现上的简单，最终结果就是导致代码复杂度的上升。

## 学习一本书：如何实现链表

最后，推荐一本专门将如何实现链表的书（真是富有 Rust 特色，链表都能复杂到出书了 o\_o），[Learn Rust by writing Entirely Too Many Linked Lists](#)

## 总结

上面讲了这么多方法，但是我们依然无法正确的告诉你在某个场景应该使用哪个方法，这个需要你自己的判断，因为自引用实在是过于复杂。

我们能做的就是告诉你，有这些办法可以解决自引用问题，而这些办法每个都有自己适用的范围，需要你未来去深入的挖掘和发现。

偷偷说一句，就算是我，遇到自引用一样挺头疼，好在这种情况真的不常见，往往是实现特定的算法和数据结构时才需要，应用代码中几乎用不到。

# 多线程并发编程

安全和高效的处理并发是 Rust 语言的主要目标之一。随着现代处理器的核心数不断增加，并发和并行已经成为日常编程不可或缺的一部分，甚至于 Go 语言已经将并发简化到一个 `go` 关键字就可以。

可惜的是，在 Rust 中由于语言设计理念、安全、性能的多方面考虑，并没有采用 Go 语言大道至简的方式，而是选择了多线程与 `async/await` 相结合，优点是可控性更强、性能更高，缺点是复杂度并不低，当然这也是系统级语言的应有选择：**使用复杂度换取可控性和性能**。

不过，大家也不用担心，本书的目标就是降低 Rust 使用门槛，这个门槛自然也包括如何在 Rust 中进行异步并发编程，我们将从多线程以及 `async/await` 两个方面去深入浅出地讲解，首先，从本章的多线程开始。

在本章，我们将深入讲解并发和并行的区别以及如何使用多线程进行 Rust 并发编程，那么先来看看何为并行与并发。

# 并发和并行

并发是同一时间应对多件事情的能力 - Rob Pike

并行和并发其实并不难，但是也给一些用户造成了困扰，因此我们专门开辟一个章节，用于讲清楚这两者的区别。

Erlang 之父 Joe Armstrong (伟大的异步编程先驱，开创一个时代的殿堂级计算机科学家，我还犹记得当年刚学到 Erlang 时的震撼，respect! ) 用一张 5 岁小孩都能看懂的图片解释了并发与并行的区别：

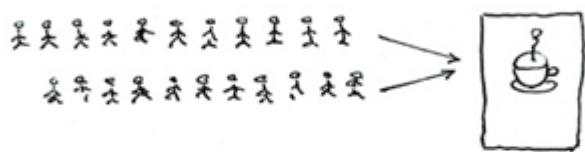
## Concurrent and Parallel Programming

05 Apr 2013

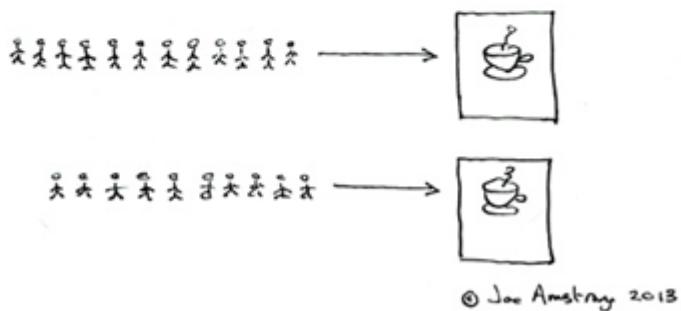
What's the difference between concurrency and parallelism?

Explain it to a five year old.

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2013

Concurrent = Two queues and one coffee machine.

Parallel = Two queues and two coffee machines.

上图很直观的体现了：

- **并发(Concurrent)** 是多个队列使用同一个咖啡机，然后两个队列轮换着使用（未必是 1:1 轮换，也可能是其它轮换规则），最终每个人都能接到咖啡

- **并行(Parallel)** 是每个队列都拥有一个咖啡机，最终也是每个人都能接到咖啡，但是效率更高，因为同时可以有两个人在接咖啡

当然，我们还可以对比下串行：只有一个队列且仅使用一台咖啡机，前面哪个人接咖啡时突然发呆了几分钟，后面的人就只能等他结束才能继续接。可能有读者有疑问了，从图片来看，并发也存在这个问题啊，前面的人发呆了几分钟不接咖啡怎么办？很简单，另外一个队列的人把他推开就行了，自己队友不能在背后开枪，但是其它队的可以：）

在正式开始之前，先给出一个结论：**并发和并行都是对“多任务”处理的描述，其中并发是轮流处理，而并行是同时处理。**

## CPU 多核

现在的个人计算机动辄拥有十来个核心（M1 Max/Intel 12 代），如果使用串行的方式那真是太低效了，因此我们把各种任务简单分成多个队列，每个队列都交给一个 CPU 核心去执行，当某个 CPU 核心没有任务时，它还能去其它核心的队列中偷任务（真·老黄牛），这样就实现了并行化处理。

### 单核心并发

那问题来了，在早期只有一个 CPU 核心时，我们的任务是怎么处理的呢？其实聪明的读者应该已经想到，是的，并发解君愁。当然，这里还得提到操作系统的多线程，正是操作系统多线程 + CPU 核心，才实现了现代化的多任务操作系统。

在 OS 级别，多线程负责管理我们的任务队列，你可以简单认为一个线程管理着一个任务队列，然后线程之间还能根据空闲度进行任务调度。我们的程序只会跟 OS 线程打交道，并不关心 CPU 到底有多少个核心，真正关心的只是 OS，当线程把任务交给 CPU 核心去执行时，如果只有一个 CPU 核心，那么它就只能同时处理一个任务。

相信大家都看出来了：**CPU 核心对应的是上图的咖啡机，而多个线程的任务队列就对应的多个排队的队列**，由于终受限于 CPU 核心数，每个队列每次只会有一个任务被处理。

和排队一样，假如某个任务执行时间过长，就会导致用户界面的假死（相信使用 Windows 的同学或多或少都碰到过假死的问题），那么就需要 CPU 的任务调度了（真实 CPU 的调度很复杂，我们这里做了简化），有一个调度器会按照某些条件从队列中选择任务进行执行，并且当一个任务执行时间过长时，会强行切换该任务到后台中（或者放入任务队列，真实情况很复杂！），去执行新的任务。

不断这样的快速任务切换，对用户而言就实现了表面上的多任务同时处理，但是实际上最终也只有一个 CPU 核心在不停的工作。

因此并发的关键在于：**快速轮换处理不同的任务**，给用户带来所有任务同时在运行的假象。

## 多核心并行

当 CPU 核心增多到  $N$  时，那么同一时间就能有  $N$  个任务被处理，那么我们的并行度就是  $N$ ，相应的处理效率也变成了单核心的  $N$  倍（实际情况并没有这么高）。

## 多核心并发

当核心增多到  $N$  时，操作系统同时在进行的任务肯定远不止  $N$  个，这些任务将被放入  $M$  个线程队列中，接着交给  $N$  个 CPU 核心去执行，最后实现了  $M:N$  的处理模型，在这种情况下，**并发与并行是同时在发生的，所有用户任务从表面来看都在并发的运行，但实际上，同一时刻只有  $N$  个任务能被同时并行的处理。**

看到这里，相信大家已经明白两者的区别，那么我们下面给出一个正式的定义（该定义摘选自《并发的艺术》）。

## 正式的定义

如果某个系统支持两个或者多个动作的**同时存在**，那么这个系统就是一个并发系统。如果某个系统支持两个或者多个动作**同时执行**，那么这个系统就是一个并行系统。并发系统与并行系统这两个定义之间的关键差异在于“**存在**”这个词。

在并发程序中可以同时拥有两个或者多个线程。这意味着，如果程序在单核处理器上运行，那么这两个线程将交替地换入或者换出内存。这些线程是**同时“存在”**的——每个线程都处于执行过程中的某个状态。如果程序能够并行执行，那么就一定是运行在多核处理器上。此时，程序中的每个线程都将分配到一个独立的处理器核上，因此可以同时运行。

相信你已经能够得出结论——“**并行**”概念是“**并发**”概念的一个子集。也就是说，你可以编写一个拥有多个线程或者进程的并发程序，但如果有多核处理器来执行这个程序，那么就不能以并行方式来运行代码。因此，凡是在求解单个问题时涉及多个执行流程的编程模式或者执行行为，都属于并发编程的范畴。

## 编程语言的并发模型

如果大家学过其它语言的多线程，可能就知道不同语言对于线程的实现可能大相径庭：

- 由于操作系统提供了创建线程的 API，因此部分语言会直接调用该 API 来创建线程，因此最终程序内的线程数和该程序占用的操作系统线程数相等，一般称之为**1:1 线程模型**，例如 Rust。
- 还有些语言在内部实现了自己的线程模型（绿色线程、协程），程序内部的  $M$  个线程最后会以某种映射方式使用  $N$  个操作系统线程去运行，因此称之为**M:N 线程模型**，其中  $M$  和  $N$  并没有特定的彼

此限制关系。一个典型的代表就是 Go 语言。

- 还有些语言使用了 Actor 模型，基于消息传递进行并发，例如 Erlang 语言。

总之，每一种模型都有其优缺点及选择上的权衡，而 Rust 在设计时考虑的权衡就是运行时(Runtime)。出于 Rust 的系统级使用场景，且要保证调用 C 时的极致性能，它最终选择了尽量小的运行时实现。

---

运行时是那些会被打包到所有程序可执行文件中的 Rust 代码，根据每个语言的设计权衡，运行时虽然有大有小（例如 Go 语言由于实现了协程和 GC，运行时相对就会更大一些），但是除了汇编之外，每个语言都拥有它。小运行时的其中一个好处在于最终编译出的可执行文件会相对较小，同时也让该语言更容易被其它语言引入使用。

而绿色线程/协程的实现会显著增大运行时的大小，因此 Rust 只在标准库中提供了 1:1 的线程模型，如果你愿意牺牲一些性能来换取更精确的线程控制以及更小的线程上下文切换成本，那么可以选择 Rust 中的 M:N 模型，这些模型由三方库提供了实现，例如大名鼎鼎的 `tokio`。

在了解了并发和并行后，我们可以正式开始 Rust 的多线程之旅。

# 使用线程

放在十年前，多线程编程可能还是一个少数人才掌握的核心概念，但是在今天，随着编程语言的不断发展，多线程、多协程、Actor 等并发编程方式已经深入人心，同时多线程编程的门槛也在不断降低，本章节我们来看看在 Rust 中该如何使用多线程。

## 多线程编程的风险

由于多线程的代码是同时运行的，因此我们无法保证线程间的执行顺序，这会导致一些问题：

- 竞态条件(race conditions)，多个线程以非一致性的顺序同时访问数据资源
- 死锁(deadlocks)，两个线程都想使用某个资源，但是又都在等待对方释放资源后才能使用，结果最终都无法继续执行
- 一些因为多线程导致的很隐晦的 BUG，难以复现和解决

虽然 Rust 已经通过各种机制减少了上述情况的发生，但是依然无法完全避免上述情况，因此我们在编程时需要格外的小心，同时本书也会列出多线程编程时常见的陷阱，让你提前规避可能的风险。

## 创建线程

使用 `thread::spawn` 可以创建线程：

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

有几点值得注意：

- 线程内部的代码使用闭包来执行
- `main` 线程一旦结束，程序就立刻结束，因此需要保持它的存活，直到其它子线程完成自己的任务
- `thread::sleep` 会让当前线程休眠指定的时间，随后其它线程会被调度运行（上一节并发与并行中有简单介绍过），因此就算你的电脑只有一个 CPU 核心，该程序也会表现的如同多 CPU 核心一般，这就是并发！

来看看输出：

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

如果多运行几次，你会发现好像每次输出会不太一样，因为：虽说线程往往是轮流执行的，但是这一点无法被保证！线程调度的方式往往取决于你使用的操作系统。总之，**千万不要依赖线程的执行顺序。**

## 等待子线程的结束

上面的代码你不但可能无法让子线程从 1 顺序打印到 10，而且可能打印的数字会变少，因为主线程会提前结束，导致子线程也随之结束，更过分的是，如果当前系统繁忙，甚至该子线程还没被创建，主线程就已经结束了！

因此我们需要一个方法，让主线程安全、可靠地等所有子线程完成任务后，再 `kill self`：

```

use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}

```

通过调用 `handle.join`，可以让当前线程阻塞，直到它等待的子线程的结束，在上面代码中，由于 `main` 线程会被阻塞，因此它直到子线程结束后才会输出自己的 `1..5`：

```

hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!

```

以上输出清晰的展示了线程阻塞的作用，如果你将 `handle.join` 放置在 `main` 线程中的 `for` 循环后面，那就是另外一个结果：两个线程交替输出。

## 在线程闭包中使用 move

在[闭包](#)章节中，有讲过 `move` 关键字在闭包中的使用可以让该闭包拿走环境中某个值的所有权，同样地，你可以使用 `move` 来将所有权从一个线程转移到另外一个线程。

首先，来看看在一个线程中直接使用另一个线程中的数据会如何：

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

以上代码在子线程的闭包中捕获了环境中的 `v` 变量，来看看结果：

```
error[E0373]: closure may outlive the current function, but it borrows `v`, which is
owned by the current function
--> src/main.rs:6:32
|
6 |     let handle = thread::spawn(|| {
|             ^^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
|                 - `v` is borrowed here
|
note: function requires argument type to outlive ``static``
--> src/main.rs:6:18
|
6 |     let handle = thread::spawn(|| {
|             ^
7 |     |-----^
|     |         println!("Here's a vector: {:?}", v);
|     |         ^
|     |-----^
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
|
6 |     let handle = thread::spawn(move || {
|             +***+
```

其实代码本身并没有什么问题，问题在于 Rust 无法确定新的线程会活多久（多个线程的结束顺序并不是固定的），所以也无法确定新线程所引用的 `v` 是否在使用过程中一直合法：

```

use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}

```

大家要记住，线程的启动时间点和结束时间点是不确定的，因此存在一种可能，当主线程执行完，`v` 被释放掉时，新的线程很可能还没有结束甚至还没有被创建成功，此时新线程对 `v` 的引用立刻就不再合法！

好在报错里进行了提示： `to force the closure to take ownership of v (and any other referenced variables), use the `move` keyword`，让我们使用 `move` 关键字拿走 `v` 的所有权即可：

```

use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();

    // 下面代码会报错borrow of moved value: `v`
    // println!("{:?}", v);
}

```

如上所示，很简单的代码，而且 Rust 的所有权机制保证了数据使用上的安全：`v` 的所有权被转移给新的线程后，`main` 线程将无法继续使用：最后一行代码将报错。

## 线程是如何结束的

之前我们提到 `main` 线程是程序的主线程，一旦结束，则程序随之结束，同时各个子线程也将被强行终止。那么有一个问题，如果父线程不是 `main` 线程，那么父线程的结束会导致什么？自生自灭还是被干

掉？

在系统编程中，操作系统提供了直接杀死线程的接口，简单粗暴，但是 Rust 并没有提供这样的接口，原因在于，粗暴地终止一个线程可能会导致资源没有释放、状态混乱等不可预期的结果，一向以安全自称的 Rust，自然不会砸自己的饭碗。

那么 Rust 中线程是如何结束的呢？答案很简单：线程的代码执行完，线程就会自动结束。但是如果线程中的代码不会执行完呢？那么情况可以分为两种进行讨论：

- 线程的任务是一个循环 IO 读取，任务流程类似：IO 阻塞，等待读取新的数据 -> 读到数据，处理完成 -> 继续阻塞等待 … -> 收到 socket 关闭的信号 -> 结束线程，在此过程中，绝大部分时间线程都处于阻塞的状态，因此虽然看上去是循环，CPU 占用其实很小，也是网络服务中最常见的模型
- 线程的任务是一个循环，里面没有任何阻塞，包括休眠这种操作也没有，此时 CPU 很不幸的会被跑满，而且你如果没有设置终止条件，该线程将持续跑满一个 CPU 核心，并且不会被终止，直到 main 线程的结束

第一情况很常见，我们来模拟看看第二种情况：

```
use std::thread;
use std::time::Duration;
fn main() {
    // 创建一个线程A
    let new_thread = thread::spawn(move || {
        // 再创建一个线程B
        thread::spawn(move || {
            loop {
                println!("I am a new thread.");
            }
        })
    });
    // 等待新创建的线程执行完成
    new_thread.join().unwrap();
    println!("Child thread is finish!");

    // 睡眠一段时间，看子线程创建的子线程是否还在运行
    thread::sleep(Duration::from_millis(100));
}
```

以上代码中，`main` 线程创建了一个新的线程 A，同时该新线程又创建了一个新的线程 B，可以看到 A 线程在创建完 B 线程后就立即结束了，而 B 线程则在不停地循环输出。

从之前的线程结束规则，我们可以猜测程序将这样执行：A 线程结束后，由它创建的 B 线程仍在疯狂输出，直到 `main` 线程在 100 毫秒后结束。如果你把该时间增加到几十秒，就可以看到你的 CPU 核心 100% 的盛况了-,-

# 多线程的性能

下面我们从多个方面来看看多线程的性能大概是怎么样的。

## 创建线程的性能

据不精确估算，创建一个线程大概需要 0.24 毫秒，随着线程的变多，这个值会变得更大，因此线程的创建耗时是不可忽略的，只有当真的需要处理一个值得用线程去处理的任务时，才使用线程，一些鸡毛蒜皮的任务，就无需创建线程了。

## 创建多少线程合适

因为 CPU 的核心数限制，当任务是 CPU 密集型时，就算线程数超过了 CPU 核心数，也并不能帮你获得更好的性能，因为每个线程的任务都可以轻松让 CPU 的某个核心跑满，既然如此，让线程数等于 CPU 核心数是最好的。

但是当你的任务大部分时间都处于阻塞状态时，就可以考虑增多线程数量，这样当某个线程处于阻塞状态时，会被切走，进而运行其它的线程，典型就是网络 IO 操作，我们可以为每一个进来的用户连接创建一个线程去处理，该连接绝大部分时间都是处于 IO 读取阻塞状态，因此有限的 CPU 核心完全可以处理成百上千的用户连接线程，但是事实上，对于这种网络 IO 情况，一般都不再使用多线程的方式了，毕竟操作系统的线程数是有限的，意味着并发数也很容易达到上限，而且过多的线程也会导致线程上下文切换的代价过大，使用 `async/await` 的 M:N 并发模型，就没有这个烦恼。

## 多线程的开销

下面的代码是一个无锁实现(CAS)的 `HashMap` 在多线程下的使用：

```

for i in 0..num_threads {
    let ht = Arc::clone(&ht);

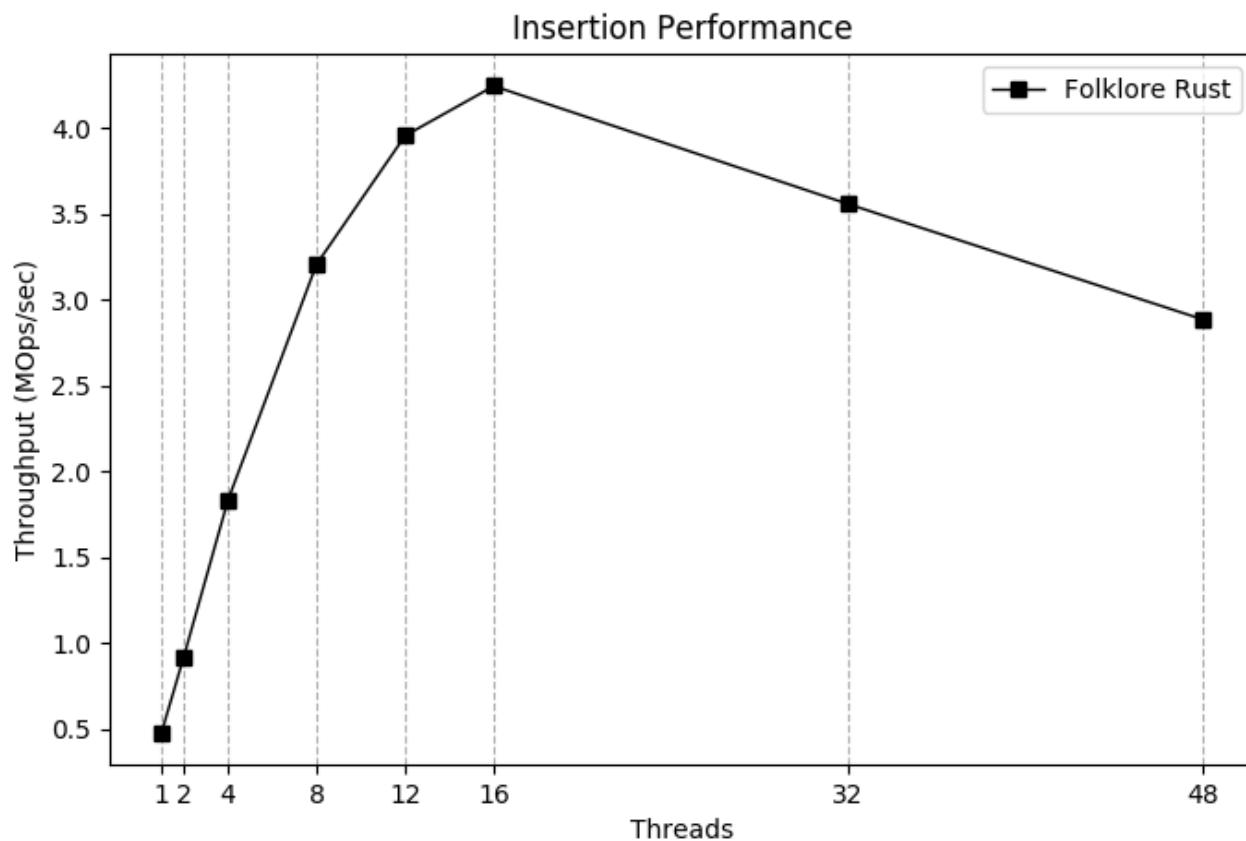
    let handle = thread::spawn(move || {
        for j in 0..adds_per_thread {
            let key = thread_rng().gen::<u32>();
            let value = thread_rng().gen::<u32>();
            ht.set_item(key, value);
        }
    });
    handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}

```

按理来说，既然是无锁实现了，那么锁的开销应该几乎没有，性能会随着线程数的增加接近线性增长，但真的是这样吗？

下图是该代码在 48 核机器上的运行结果：



从图上可以明显的看出：吞吐并不是线性增长，尤其从 16 核开始，甚至开始肉眼可见的下降，这是为什么呢？

限于书本的篇幅有限，我们只能给出大概的原因：

- 虽然是无锁，但是内部是 CAS 实现，大量线程的同时访问，会让 CAS 重试次数大幅增加
- 线程过多时，CPU 缓存的命中率会显著下降，同时多个线程竞争一个 CPU Cache-line 的情况也会经常发生
- 大量读写可能会让内存带宽也成为瓶颈
- 读和写不一样，无锁数据结构的读往往可以很好地线性增长，但是写不行，因为写竞争太大

总之，多线程的开销往往是在锁、数据竞争、缓存失效上，这些限制了现代化软件系统随着 CPU 核心的增多性能也线性增加的野心。

## 线程屏障(Barrier)

在 Rust 中，可以使用 `Barrier` 让多个线程都执行到某个点后，才继续一起往后执行：

```
use std::sync::{Arc, Barrier};
use std::thread;

fn main() {
    let mut handles = Vec::with_capacity(6);
    let barrier = Arc::new(Barrier::new(6));

    for _ in 0..6 {
        let b = barrier.clone();
        handles.push(thread::spawn(move|| {
            println!("before wait");
            b.wait();
            println!("after wait");
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

上面代码，我们在线程打印出 `before wait` 后增加了一个屏障，目的就是等所有的线程都打印出**before wait**后，各个线程再继续执行：

```
before wait
after wait
after wait
after wait
after wait
after wait
after wait
```

## 线程局部变量(Thread Local Variable)

对于多线程编程，线程局部变量在一些场景下非常有用，而 Rust 通过标准库和三方库对此进行了支持。

### 标准库 `thread_local`

使用 `thread_local` 宏可以初始化线程局部变量，然后在线程内部使用该变量的 `with` 方法获取变量值：

```

use std::cell::RefCell;
use std::thread;

thread_local!(static FOO: RefCell<u32> = RefCell::new(1));

FOO.with(|f| {
    assert_eq!(*f.borrow(), 1);
    *f.borrow_mut() = 2;
});

// 每个线程开始时都会拿到线程局部变量的FOO的初始值
let t = thread::spawn(move|| {
    FOO.with(|f| {
        assert_eq!(*f.borrow(), 1);
        *f.borrow_mut() = 3;
    });
});

// 等待线程完成
t.join().unwrap();

// 尽管子线程中修改为了3，我们在这里依然拥有main线程中的局部值：2
FOO.with(|f| {
    assert_eq!(*f.borrow(), 2);
});

```

上面代码中，`FOO` 即是我们创建的**线程局部变量**，每个新的线程访问它时，都会使用它的初始值作为开始，各个线程中的 `FOO` 值彼此互不干扰。注意 `FOO` 使用 `static` 声明为生命周期为 '`static`' 的静态变量。

可以注意到，线程中对 `FOO` 的使用是通过借用的方式，但是若我们需要每个线程独自获取它的拷贝，最后进行汇总，就有些强人所难了。

你还可以在结构体中使用线程局部变量：

```

use std::cell::RefCell;

struct Foo;
impl Foo {
    thread_local! {
        static FOO: RefCell<usize> = RefCell::new(0);
    }
}

fn main() {
    Foo::FOO.with(|x| println!("{:?}", x));
}

```

或者通过引用的方式使用它：

```
use std::cell::RefCell;
use std::thread::LocalKey;

thread_local! {
    static FOO: RefCell<usize> = RefCell::new(0);
}

struct Bar {
    foo: &'static LocalKey<RefCell<usize>>,
}

impl Bar {
    fn constructor() -> Self {
        Self {
            foo: &FOO,
        }
    }
}
```

### 三方库 [thread-local](#)

除了标准库外，一位大神还开发了 [thread-local](#) 库，它允许每个线程持有值的独立拷贝：

```

use thread_local::ThreadLocal;
use std::sync::Arc;
use std::cell::Cell;
use std::thread;

let tls = Arc::new(ThreadLocal::new());
let mut v = vec![];
// 创建多个线程
for _ in 0..5 {
    let tls2 = tls.clone();
    let handle = thread::spawn(move || {
        // 将计数器加1
        // 请注意，由于线程 ID 在线程退出时会被回收，因此一个线程有可能回收另一个线程的对象
        // 这只能在线程退出后发生，因此不会导致任何竞争条件
        let cell = tls2.get_or(|| Cell::new(0));
        cell.set(cell.get() + 1);
    });
    v.push(handle);
}
for handle in v {
    handle.join().unwrap();
}
// 一旦所有子线程结束，收集它们的线程局部变量中的计数器值，然后进行求和
let tls = Arc::try_unwrap(tls).unwrap();
let total = tls.into_iter().fold(0, |x, y| {
    // 打印每个线程局部变量中的计数器值，发现不一定有5个线程，
    // 因为一些线程已退出，并且其他线程会回收退出线程的对象
    println!("x: {}, y: {}", x, y.get());
    x + y.get()
});
// 和为5
assert_eq!(total, 5);

```

该库不仅仅使用了值的拷贝，而且还能自动把多个拷贝汇总到一个迭代器中，最后进行求和，非常好用。

## 用条件控制线程的挂起和执行

条件变量(Condition Variables)经常和 Mutex 一起使用，可以让线程挂起，直到某个条件发生后再继续执行：

```
use std::thread;
use std::sync::{Arc, Mutex, Condvar};

fn main() {
    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = pair.clone();

    thread::spawn(move|| {
        let (lock, cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        println!("changing started");
        *started = true;
        cvar.notify_one();
    });

    let (lock, cvar) = &*pair;
    let mut started = lock.lock().unwrap();
    while !*started {
        started = cvar.wait(started).unwrap();
    }

    println!("started changed");
}
```

上述代码流程如下：

1. main 线程首先进入 while 循环，调用 wait 方法挂起等待子线程的通知，并释放了锁 started
2. 子线程获取到锁，并将其修改为 true，然后调用条件变量的 notify\_one 方法来通知主线程继续执行

## 只被调用一次的函数

有时，我们会需要某个函数在多线程环境下只被调用一次，例如初始化全局变量，无论是哪个线程先调用函数来初始化，都会保证全局变量只会被初始化一次，随后的其它线程调用就会忽略该函数：

```

use std::thread;
use std::sync::Once;

static mut VAL: usize = 0;
static INIT: Once = Once::new();

fn main() {
    let handle1 = thread::spawn(move || {
        INIT.call_once(|| {
            unsafe {
                VAL = 1;
            }
        });
    });

    let handle2 = thread::spawn(move || {
        INIT.call_once(|| {
            unsafe {
                VAL = 2;
            }
        });
    });

    handle1.join().unwrap();
    handle2.join().unwrap();

    println!("{}", unsafe { VAL });
}

```

代码运行的结果取决于哪个线程先调用 `INIT.call_once`（虽然代码具有先后顺序，但是线程的初始化顺序并无法被保证！因为线程初始化是异步的，且耗时较久），若 `handle1` 先，则输出 1，否则输出 2。

## call\_once 方法

执行初始化过程一次，并且只执行一次。

如果当前有另一个初始化过程正在运行，线程将阻止该方法被调用。

当这个函数返回时，保证一些初始化已经运行并完成，它还保证由执行的闭包所执行的任何内存写入都能被其他线程在这时可靠地观察到。

## 总结

Rust 的线程模型是 1:1 模型，因为 Rust 要保持尽量小的运行时。

我们可以使用 `thread::spawn` 来创建线程，创建出的多个线程之间并不存在执行顺序关系，因此代码逻辑千万不要依赖于线程间的执行顺序。

`main` 线程若是结束，则所有子线程都将被终止，如果希望等待子线程结束后，再结束 `main` 线程，你需要使用创建线程时返回的句柄的 `join` 方法。

在线程中无法直接借用外部环境中的变量值，因为新线程的启动时间点和结束时间点是不确定的，所以 Rust 无法保证该线程中借用的变量在使用过程中依然是合法的。你可以使用 `move` 关键字将变量的所有权转移给新的线程，来解决此问题。

父线程结束后，子线程仍在持续运行，直到子线程的代码运行完成或者 `main` 线程的结束。

# 线程间的消息传递

在多线程间有多种方式可以共享、传递数据，最常用的方式就是通过消息传递或者将锁和 Arc 联合使用，而对于前者，在编程界还有一个大名鼎鼎的 Actor 线程模型 为其背书，典型的有 Erlang 语言，还有 Go 语言中很经典的一句话：

---

Do not communicate by sharing memory; instead, share memory by communicating

---

而对于后者，我们将在下一节中进行讲述。

## 消息通道

与 Go 语言内置的 `chan` 不同，Rust 是在标准库里提供了消息通道(`channel`)，你可以将其想象成一场直播，多个主播联合起来在搞一场直播，最终内容通过通道传输给屏幕前的我们，其中主播被称之为**发送者**，观众被称之为**接收者**，显而易见的是：一个通道应该支持多个发送者和接收者。

但是，在实际使用中，我们需要使用不同的库来满足诸如：**多发送者 -> 单接收者**，**多发送者 -> 多接收者** 等场景形式，此时一个标准库显然就不够了，不过别急，让我们先从标准库讲起。

## 多发送者，单接收者

标准库提供了通道 `std::sync::mpsc`，其中 `mpsc` 是 *multiple producer, single consumer* 的缩写，代表了该通道支持多个发送者，但是只支持唯一的接收者。当然，支持多个发送者也意味着支持单个发送者，我们先来看看单发送者、单接收者的简单例子：

```

use std::sync::mpsc;
use std::thread;

fn main() {
    // 创建一个消息通道，返回一个元组：(发送者, 接收者)
    let (tx, rx) = mpsc::channel();

    // 创建线程，并发送消息
    thread::spawn(move || {
        // 发送一个数字1, send方法返回Result<T,E>, 通过unwrap进行快速错误处理
        tx.send(1).unwrap();

        // 下面代码将报错，因为编译器自动推导出通道传递的值是i32类型，那么Option<i32>类型将产生不
        // 匹配错误
        // tx.send(Some(1)).unwrap()
    });

    // 在主线程中接收子线程发送的消息并输出
    println!("receive {}", rx.recv().unwrap());
}

```

以上代码并不复杂，但仍有几点需要注意：

- `tx, rx` 对应发送者和接收者，它们的类型由编译器自动推导：`tx.send(1)` 发送了整数，因此它们分别是 `mpsc::Sender<i32>` 和 `mpsc::Receiver<i32>` 类型，需要注意，由于内部是泛型实现，一旦类型被推导确定，该通道就只能传递对应类型的值，例如此例中非 `i32` 类型的值将导致编译错误
- 接收消息的操作 `rx.recv()` 会阻塞当前线程，直到读取到值，或者通道被关闭
- 需要使用 `move` 将 `tx` 的所有权转移到子线程的闭包中

在注释中提到 `send` 方法返回一个 `Result<T,E>`，说明它有可能返回一个错误，例如接收者被 `drop` 导致了发送的值不会被任何人接收，此时继续发送毫无意义，因此返回一个错误最为合适，在代码中我们仅仅使用 `unwrap` 进行了快速处理，但在实际项目中你需要对错误进行进一步的处理。

同样的，对于 `recv` 方法来说，当发送者关闭时，它也会接收到一个错误，用于说明不会再有任何值被发送过来。

## 不阻塞的 `try_recv` 方法

除了上述 `recv` 方法，还可以使用 `try_recv` 尝试接收一次消息，该方法**并不会阻塞线程**，当通道中没有消息时，它会立刻返回一个错误：

```

use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send(1).unwrap();
    });

    println!("receive {:?}", rx.try_recv());
}

```

由于子线程的创建需要时间，因此 `println!` 和 `try_recv` 方法会先执行，而此时子线程的消息还未被发出。`try_recv` 会尝试立即读取一次消息，因为消息没有发出，此次读取最终会报错，且主线程运行结束（可悲的是，相对于主线程中的代码，子线程的创建速度实在是过慢，直到主线程结束，都无法完成子线程的初始化。。）：

```
receive Err(Empty)
```

如上，`try_recv` 返回了一个错误，错误内容是 `Empty`，代表通道并没有消息。如果你尝试把 `println!` 复制一些行，就会发现一个有趣的输出：

```

...
receive Err(Empty)
receive Ok(1)
receive Err(Disconnected)
...

```

如上，当子线程创建成功且发送消息后，主线程会接收到 `Ok(1)` 的消息内容，紧接着子线程结束，发送者也随着被 `drop`，此时接收者又会报错，但是这次错误原因有所不同：`Disconnected` 代表发送者已经被关闭。

## 传输具有所有权的数据

使用通道来传输数据，一样要遵循 Rust 的所有权规则：

- 若值的类型实现了 `Copy` 特征，则直接复制一份该值，然后传输过去，例如之前的 `i32` 类型
- 若值没有实现 `Copy`，则它的所有权会被转移给接收端，在发送端继续使用该值将报错

一起来看看第二种情况：

```

use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let s = String::from("我，飞走咯!");
        tx.send(s).unwrap();
        println!("val is {}", s);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}

```

以上代码中，`String` 底层的字符串是存储在堆上，并没有实现 `Copy` 特征，当它被发送后，会将所有权从发送端的 `s` 转移给接收端的 `received`，之后 `s` 将无法被使用：

```

error[E0382]: borrow of moved value: `s`
--> src/main.rs:10:31
|
8 |         let s = String::from("我，飞走咯!");
|             - move occurs because `s` has type `String`, which does not
implement the `Copy` trait // 所有权被转移，由于`String`没有实现`Copy`特征
9 |         tx.send(s).unwrap();
|             - value moved here // 所有权被转移走
10 |         println!("val is {}", s);
|                         ^ value borrowed here after move // 所有权被转移后,
依然对s进行了借用

```

各种细节不禁令人感叹：Rust 还是安全！假如没有所有权的保护，`String` 字符串将被两个线程同时持有，任何一个线程对字符串内容的修改都会导致另外一个线程持有的字符串被改变，除非你故意这么设计，否则这就是不安全的隐患。

## 使用 `for` 进行循环接收

下面来看看如何连续接收通道中的值：

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

在上面代码中，主线程和子线程是并发运行的，子线程在不停的**发送消息 -> 休眠 1 秒**，与此同时，主线程使用 `for` 循环**阻塞**的从 `rx` **迭代器**中接收消息，当子线程运行完成时，发送者 `tx` 会随之被 `drop`，此时 `for` 循环将被终止，最终 `main` 线程成功结束。

## 使用多发送者

由于子线程会拿走发送者的所有权，因此我们必须对发送者进行克隆，然后让每个线程拿走它的一份拷贝：

```

use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    let tx1 = tx.clone();
    thread::spawn(move || {
        tx.send(String::from("hi from raw tx")).unwrap();
    });

    thread::spawn(move || {
        tx1.send(String::from("hi from cloned tx")).unwrap();
    });

    for received in rx {
        println!("Got: {}", received);
    }
}

```

代码并无太大区别，就多了一个对发送者的克隆 `let tx1 = tx.clone();`，然后一个子线程拿走 `tx` 的所有权，另一个子线程拿走 `tx1` 的所有权，皆大欢喜。

但是有几点需要注意：

- 需要所有的发送者都被 `drop` 掉后，接收者 `rx` 才会收到错误，进而跳出 `for` 循环，最终结束主线程
- 这里虽然用了 `clone` 但是并不会影响性能，因为它并不在热点代码路径中，仅仅会被执行一次
- 由于两个子线程谁先创建完成是未知的，因此哪条消息先发送也是未知的，最终主线程的输出顺序也不确定

## 消息顺序

上述第三点的消息顺序仅仅是因为线程创建引起的，并不代表通道中的消息是无序的，对于通道而言，消息的发送顺序和接收顺序是一致的，满足 `FIFO` 原则(先进先出)。

由于篇幅有限，具体的代码这里就不再给出，感兴趣的读者可以自己验证下。

## 同步和异步通道

Rust 标准库的 `mpsc` 通道其实分为两种类型：同步和异步。

## 异步通道

之前我们使用的都是异步通道：无论接收者是否正在接收消息，消息发送者在发送消息时都不会阻塞：

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx, rx) = mpsc::channel();

    let handle = thread::spawn(move || {
        println!("发送之前");
        tx.send(1).unwrap();
        println!("发送之后");
    });

    println!("睡眠之前");
    thread::sleep(Duration::from_secs(3));
    println!("睡眠之后");

    println!("receive {}", rx.recv().unwrap());
    handle.join().unwrap();
}
```

运行后输出如下：

```
睡眠之前
发送之前
发送之后
//...睡眠3秒
睡眠之后
receive 1
```

主线程因为睡眠阻塞了 3 秒，因此并没有进行消息接收，而子线程却在此期间轻松完成了消息的发送。等主线程睡眠结束后，才姗姗来迟的从通道中接收了子线程老早之前发送的消息。

从输出还可以看出，`发送之前` 和 `发送之后` 是连续输出的，没有受到接收端主线程的任何影响，因此通过 `mpsc::channel` 创建的通道是异步通道。

## 同步通道

与异步通道相反，同步通道**发送消息是阻塞的，只有在消息被接收后才解除阻塞**，例如：

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx, rx) = mpsc::sync_channel(0);

    let handle = thread::spawn(move || {
        println!("发送之前");
        tx.send(1).unwrap();
        println!("发送之后");
    });

    println!("睡眠之前");
    thread::sleep(Duration::from_secs(3));
    println!("睡眠之后");

    println!("receive {}", rx.recv().unwrap());
    handle.join().unwrap();
}
```

运行后输出如下：

```
睡眠之前
发送之前
//...睡眠3秒
睡眠之后
receive 1
发送之后
```

可以看出，主线程由于睡眠被阻塞导致无法接收消息，因此子线程的发送也一直被阻塞，直到主线程结束睡眠并成功接收消息后，发送才成功：**发送之后的输出是在receive 1之后**，说明**只有接收消息彻底成功后，发送消息才算完成。**

## 消息缓存

细心的读者可能已经发现在创建同步通道时，我们传递了一个参数 `0: mpsc::sync_channel(0);`，这是什么意思呢？

答案不急给出，先将 `0` 改成 `1`，然后再运行试试：

```
睡眠之前
发送之前
发送之后
睡眠之后
receive 1
```

纳尼。。。竟然得到了和异步通道一样的效果：根本没有等待主线程的接收开始，消息发送就立即完成了！难道同步通道变成了异步通道？别急，将子线程中的代码修改下试试：

```
println!("首次发送之前");
tx.send(1).unwrap();
println!("首次发送之后");
tx.send(1).unwrap();
println!("再次发送之后");
```

在子线程中，我们又多发了一条消息，此时输出如下：

```
睡眠之前
首次发送之前
首次发送之后
//...睡眠3秒
睡眠之后
receive 1
再次发送之后
```

Bingo，更奇怪的事出现了，第一条消息瞬间发送完成，没有阻塞，而发送第二条消息时却符合同步通道的特点：阻塞了，直到主线程接收后，才发送完成。

其实，一切的关键就在于 `1` 上，该值可以用来指定同步通道的消息缓存条数，当你设定为 `N` 时，发送者就可以无阻塞的往通道中发送 `N` 条消息，当消息缓冲队列满了后，新的消息发送将被阻塞(如果没有接收者消费缓冲队列中的消息，那么第 `N+1` 条消息就将触发发送阻塞)。

问题又来了，异步通道创建时完全没有这个缓冲值参数 `mpsc::channel()`，它的缓冲值怎么设置呢？额。。。都异步了，都可以无限发送了，都有摩托车了，还要自行车做啥子哦？事实上异步通道的缓冲上限取决于你的内存大小，不要撑爆就行。

因此，使用异步消息虽然能非常高效且不会造成发送线程的阻塞，但是存在消息未及时消费，最终内存过大的问题。在实际项目中，可以考虑使用一个带缓冲值的同步通道来避免这种风险。

## 关闭通道

之前我们数次提到了通道关闭，并且提到了当通道关闭后，发送消息或接收消息将会报错。那么如何关闭通道呢？很简单：**所有发送者被 drop 或者所有接收者被 drop 后，通道会自动关闭**。

神奇的是，这件事是在编译期实现的，完全没有运行期性能损耗！只能说 Rust 的 Drop 特征 YYDS！

## 传输多种类型的数据

之前提到过，一个消息通道只能传输一种类型的数据，如果你想要传输多种类型的数据，可以为每个类型创建一个通道，你也可以使用枚举类型来实现：

```
use std::sync::mpsc::{self, Receiver, Sender};

enum Fruit {
    Apple(u8),
    Orange(String)
}

fn main() {
    let (tx, rx): (Sender<Fruit>, Receiver<Fruit>) = mpsc::channel();

    tx.send(Fruit::Orange("sweet".to_string())).unwrap();
    tx.send(Fruit::Apple(2)).unwrap();

    for _ in 0..2 {
        match rx.recv().unwrap() {
            Fruit::Apple(count) => println!("received {} apples", count),
            Fruit::Orange(flavor) => println!("received {} oranges", flavor),
        }
    }
}
```

如上所示，枚举类型还能让我们带上想要传输的数据，但是有一点需要注意，Rust 会按照枚举中占用内存最大的那个成员进行内存对齐，这意味着就算你传输的是枚举中占用内存最小的成员，它占用的内存依然和最大的成员相同，因此会造成内存上的浪费。

## 新手容易遇到的坑

mpsc 虽然相当简洁明了，但是在使用起来还是可能存在坑：

```

use std::sync::mpsc;
fn main() {

    use std::thread;

    let (send, recv) = mpsc::channel();
    let num_threads = 3;
    for i in 0..num_threads {
        let thread_send = send.clone();
        thread::spawn(move || {
            thread_send.send(i).unwrap();
            println!("thread {:#?} finished", i);
        });
    }

    // 在这里drop send...

    for x in recv {
        println!("Got: {}", x);
    }
    println!("finished iterating");
}

```

以上代码看起来非常正常，但是运行后主线程会一直阻塞，最后一行打印输出也不会被执行，原因在于：子线程拿走的是复制后的 `send` 的所有权，这些拷贝会在子线程结束后被 `drop`，因此无需担心，但是 `send` 本身却直到 `main` 函数的结束才会被 `drop`。

之前提到，通道关闭的两个条件：发送者全部 `drop` 或接收者被 `drop`，要结束 `for` 循环显然是要求发送者全部 `drop`，但是由于 `send` 自身没有被 `drop`，会导致该循环永远无法结束，最终主线程会一直阻塞。

解决办法很简单，`drop` 掉 `send` 即可：在代码中的注释下面添加一行 `drop(send);`。

## mpmc 更好的性能

如果你需要 `mpmc`(多发送者，多接收者)或者需要更高的性能，可以考虑第三方库：

- [crossbeam-channel](#), 老牌强库，功能较全，性能较强，之前是独立的库，但是后面合并到了 `crossbeam` 主仓库中
- [flume](#), 官方给出的性能数据某些场景要比 `crossbeam` 更好些

# 线程同步：锁、Condvar 和信号量

在多线程编程中，同步性极其的重要，当你需要同时访问一个资源、控制不同线程的执行次序时，都需要使用到同步性。

在 Rust 中有多种方式可以实现同步性。在上一节中讲到的消息传递就是同步性的一种实现方式，例如我们可以通过消息传递来控制不同线程间的执行次序。还可以使用共享内存来实现同步性，例如通过锁和原子操作等并发原语来实现多个线程同时且安全地去访问一个资源。

## 该如何选择

共享内存可以说是同步的灵魂，因为消息传递的底层实际上也是通过共享内存来实现，两者的区别如下：

- 共享内存相对消息传递能节省多次内存拷贝的成本
- 共享内存的实现简洁的多
- 共享内存的锁竞争更多

消息传递适用的场景很多，我们下面列出了几个主要的使用场景：

- 需要可靠和简单的(简单不等于简洁)实现时
- 需要模拟现实世界，例如用消息去通知某个目标执行相应的操作时
- 需要一个任务处理流水线(管道)时，等等

而使用共享内存(并发原语)的场景往往就比较简单粗暴：需要简洁的实现以及更高的性能时。

总之，消息传递类似一个单所有权的系统：一个值同时只能有一个所有者，如果另一个线程需要该值的所有权，需要将所有权通过消息传递进行转移。而共享内存类似于一个多所有权的系统：多个线程可以同时访问同一个值。

## 互斥锁 Mutex

既然是共享内存，那并发原语自然是重中之重，先来一起看看皇冠上的明珠：互斥锁 Mutex (mutual exclusion 的缩写)。

Mutex 让多个线程并发的访问同一个值变成了排队访问：同一时间，只允许一个线程 A 访问该值，其它线程需要等待 A 访问完成后才能继续。

## 单线程中使用 Mutex

先来看看单线程中 Mutex 该如何使用:

```
use std::sync::Mutex;

fn main() {
    // 使用`Mutex`结构体的关联函数创建新的互斥锁实例
    let m = Mutex::new(5);

    {
        // 获取锁，然后deref为`m`的引用
        // lock返回的是Result
        let mut num = m.lock().unwrap();
        *num = 6;
        // 锁自动被drop
    }

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

在注释中，已经大致描述了代码的功能，不过有一点需要注意：和 Box 类似，数据被 Mutex 所拥有，要访问内部的数据，需要使用方法 `m.lock()` 向 `m` 申请一个锁，该方法会**阻塞当前线程，直到获取到锁**，因此当多个线程同时访问该数据时，只有一个线程能获取到锁，其它线程只能阻塞着等待，这样就保证了数据能被安全的修改！

`m.lock()` 方法也**有可能报错**，例如当前正在持有锁的线程 `panic` 了。在这种情况下，其它线程不可能再获得锁，因此 `lock` 方法会返回一个错误。

这里你可能奇怪，`m.lock` 明明返回一个锁，怎么就变成我们的 `num` 数值了？聪明的读者可能会想到智能指针，没错，因为 `Mutex<T>` 是一个智能指针，准确的说是 `m.lock()` 返回一个智能指针 `MutexGuard<T>`：

- 它实现了 `Deref` 特征，会被自动解引用后获得一个引用类型，该引用指向 `Mutex` 内部的数据
- 它还实现了 `Drop` 特征，在超出作用域后，自动释放锁，以便其它线程能继续获取锁

正因为智能指针的使用，使得我们无需任何操作就能获取其中的数据。如果释放锁，你需要做的仅仅是做好锁的作用域管理，例如上述代码的内部花括号使用，建议读者尝试下去掉内部的花括号，然后再次尝试获取第二个锁 `num1`，看看会发生什么，友情提示：不会报错，但是主线程会永远阻塞，因为不幸发生了死锁。

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    let mut num = m.lock().unwrap();
    *num = 6;
    // 锁还没有被 drop 就尝试申请下一个锁，导致主线程阻塞
    // drop(num); // 手动 drop num，可以让 num1 申请到下个锁
    let mut num1 = m.lock().unwrap();
    *num1 = 7;
    // drop(num1); // 手动 drop num1，观察打印结果的不同

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

## 多线程中使用 Mutex

单线程中使用锁，说实话纯粹是为了演示功能，毕竟多线程才是锁的舞台。现在，我们再来看看，如何在多线程下使用 `Mutex` 来访问同一个资源。

## 无法运行的Rc<T>

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    // 通过`Rc`实现`Mutex`的多所有权
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        // 创建子线程，并将`Mutex`的所有权拷贝传入到子线程中
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    // 等待所有子线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 输出最终的计数结果
    println!("Result: {}", *counter.lock().unwrap());
}
```

由于子线程需要通过 `move` 拿走锁的所有权，因此我们需要使用多所有权来保证每个线程都拿到数据的独立所有权，恰好智能指针 `Rc<T>` 可以做到(**上面代码会报错！** 具体往下看，别跳过-，-)。

以上代码实现了在多线程中计数的功能，由于多个线程都需要去修改该计数器，因此我们需要使用锁来保证同一时间只有一个线程可以修改计数器，否则会导致脏数据：想象一下 A 线程和 B 线程同时拿到计数器，获取了当前值 1，并且同时对其进行了修改，最后值变成 2，你会不会在风中凌乱？毕竟正确的值是 3，因为两个线程各自加 1。

可能有人会说，有那么巧的事情吗？事实上，对于人类来说，因为干啥啥慢，并没有那么多巧合，所以人总会存在巧合心理。但是对于计算机而言，每秒可以轻松运行上亿次，在这种频次下，一切巧合几乎都将必然发生，因此千万不要有任何侥幸心理。

---

如果事情有变坏的可能，不管这种可能性有多小，它都会发生！ - 在计算机领域歪打正着的墨菲定律

事实上，上面的代码会报错：

```
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
               // `Rc`无法在线程中安全的传输
--> src/main.rs:11:22
13 |         let handle = thread::spawn(move || {
14 |             -----^`Rc<Mutex<i32>>` cannot be sent between threads safely
15 |             let mut num = counter.lock().unwrap();
16 |             *num += 1;
17 |         });
-----^ within this `[closure@src/main.rs:11:36: 15:10]`

= help: within `#[closure@src/main.rs:11:36: 15:10]`, the trait `Send` is not
implemented for `Rc<Mutex<i32>>`
// `Rc`没有实现`Send`特征
= note: required because it appears within the type `[closure@src/main.rs:11:36:
15:10]`
```

错误中提到了一个关键点：`Rc<T>` 无法在线程中传输，因为它没有实现 `Send` 特征(在下一节将详细介绍)，而该特征可以确保数据在线程中安全的传输。

### 多线程安全的 `Arc<T>`

好在，我们有 `Arc<T>`，得益于它的[内部计数器](#)是多线程安全的，因此可以在多线程环境中使用：

```

use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}

```

以上代码可以顺利运行:

```
Result: 10
```

## 内部可变性

在之前章节，我们提到过[内部可变性](#)，其中 `Rc<T>` 和 `RefCell<T>` 的结合，可以实现单线程的内部可变性。

现在我们又有了新的武器，由于 `Mutex<T>` 可以支持修改内部数据，当结合 `Arc<T>` 一起使用时，可以实现多线程的内部可变性。

简单总结下：`Rc<T>/RefCell<T>` 用于单线程内部可变性，`Arc<T>/Mutex<T>` 用于多线程内部可变性。

## 需要小心使用的 Mutex

如果有其它语言的编程经验，就知道互斥锁这家伙不好对付，想要正确使用，你得牢记在心：

- 在使用数据前必须先获取锁
- 在数据使用完成后，必须**及时**的释放锁，比如文章开头的例子，使用内部语句块的目的就是为了及时的释放锁

这两点看起来不起眼，但要正确的使用，其实是相当不简单的，对于其它语言，忘记释放锁是经常发生的，虽然 Rust 通过智能指针的 drop 机制帮助我们避免了这一点，但是由于不及时释放锁导致的性能问题也是常见的。

正因为这种困难性，导致很多用户都热衷于使用消息传递的方式来实现同步，例如 Go 语言直接把 channel 内置在语言特性中，甚至还有无锁的语言，例如 erlang，完全使用 Actor 模型，依赖消息传递来完成共享和同步。幸好 Rust 的类型系统、所有权机制、智能指针等可以很好的帮助我们减轻使用锁时的负担。

另一个值的注意的是在使用 Mutex<T> 时，Rust 无法帮我们避免所有的逻辑错误，例如在之前章节，我们提到过使用 Rc<T> 可能会导致[循环引用的问题](#)。类似的，Mutex<T> 也存在使用上的风险，例如创建死锁（deadlock）：当一个操作试图锁住两个资源，然后两个线程各自获取其中一个锁，并试图获取另一个锁时，就会造成死锁。

## 死锁

在 Rust 中有多种方式可以创建死锁，了解这些方式有助于你提前规避可能的风险，一起来看看。

### 单线程死锁

这种死锁比较容易规避，但是当代码复杂后还是有可能遇到：

```
use std::sync::Mutex;

fn main() {
    let data = Mutex::new(0);
    let d1 = data.lock();
    let d2 = data.lock();
} // d1锁在此处释放
```

非常简单，只要你在另一个锁还未被释放时去申请新的锁，就会触发，当代码复杂后，这种情况可能就没有那么显眼。

### 多线程死锁

当我们拥有两个锁，且两个线程各自使用了其中一个锁，然后试图去访问另一个锁时，就可能发生死锁：

```

use std::sync::{Mutex, MutexGuard}, thread;
use std::thread::sleep;
use std::time::Duration;

use lazy_static::lazy_static;
lazy_static! {
    static ref MUTEX1: Mutex<i64> = Mutex::new(0);
    static ref MUTEX2: Mutex<i64> = Mutex::new(0);
}

fn main() {
    // 存放子线程的句柄
    let mut children = vec![];
    for i_thread in 0..2 {
        children.push(thread::spawn(move || {
            for _ in 0..1 {
                // 线程1
                if i_thread % 2 == 0 {
                    // 锁住MUTEX1
                    let guard: MutexGuard<i64> = MUTEX1.lock().unwrap();

                    println!("线程 {} 锁住了MUTEX1, 接着准备去锁MUTEX2 !", i_thread);

                    // 当前线程睡眠一小会儿, 等待线程2锁住MUTEX2
                    sleep(Duration::from_millis(10));

                    // 去锁MUTEX2
                    let guard = MUTEX2.lock().unwrap();
                } else {
                    // 锁住MUTEX2
                    let _guard = MUTEX2.lock().unwrap();

                    println!("线程 {} 锁住了MUTEX2, 准备去锁MUTEX1", i_thread);

                    let _guard = MUTEX1.lock().unwrap();
                }
            }
        }));
    }

    // 等子线程完成
    for child in children {
        let _ = child.join();
    }

    println!("死锁没有发生");
}

```

在上面的描述中，我们用了“可能”二字，原因在于死锁在这段代码中不是必然发生的，总有一次运行你能看到最后一行打印输出。这是由于子线程的初始化顺序和执行速度并不确定，我们无法确定哪个线程中的

锁先被执行，因此也无法确定两个线程对锁的具体使用顺序。

但是，可以简单的说明下死锁发生的必然条件：线程 1 锁住了 `MUTEX1` 并且线程 2 锁住了 `MUTEX2`，然后线程 1 试图去访问 `MUTEX2`，同时线程 2 试图去访问 `MUTEX1`，就会死锁。因为线程 2 需要等待线程 1 释放 `MUTEX1` 后，才会释放 `MUTEX2`，而与此同时，线程 1 需要等待线程 2 释放 `MUTEX2` 后才能释放 `MUTEX1`，这种情况造成了两个线程都无法释放对方需要的锁，最终死锁。

那么为何某些时候，死锁不会发生？原因很简单，线程 2 在线程 1 锁 `MUTEX1` 之前，就已经全部执行完了，随之线程 2 的 `MUTEX2` 和 `MUTEX1` 被全部释放，线程 1 对锁的获取将不再有竞争者。同理，线程 1 若全部被执行完，那线程 2 也不会被锁，因此我们在线程 1 中间加一个睡眠，增加死锁发生的概率。如果你在线程 2 中同样的位置也增加一个睡眠，那死锁将必然发生！

### `try_lock`

与 `lock` 方法不同，`try_lock` 会尝试去获取一次锁，如果无法获取会返回一个错误，因此**不会发生阻塞**：

```
use std::sync::{Mutex, MutexGuard}, thread;
use std::thread::sleep;
use std::time::Duration;

use lazy_static::lazy_static;
lazy_static! {
    static ref MUTEX1: Mutex<i64> = Mutex::new(0);
    static ref MUTEX2: Mutex<i64> = Mutex::new(0);
}

fn main() {
    // 存放子线程的句柄
    let mut children = vec![];
    for i_thread in 0..2 {
        children.push(thread::spawn(move || {
            for _ in 0..1 {
                // 线程1
                if i_thread % 2 == 0 {
                    // 锁住MUTEX1
                    let guard: MutexGuard<i64> = MUTEX1.lock().unwrap();

                    println!("线程 {} 锁住了MUTEX1, 接着准备去锁MUTEX2 !", i_thread);

                    // 当前线程睡眠一小会儿, 等待线程2锁住MUTEX2
                    sleep(Duration::from_millis(10));

                    // 去锁MUTEX2
                    let guard = MUTEX2.try_lock();
                    println!("线程 {} 获取 MUTEX2 锁的结果: {:?}", i_thread, guard);
                } else {
                    // 锁住MUTEX2
                    let _guard = MUTEX2.lock().unwrap();

                    println!("线程 {} 锁住了MUTEX2, 准备去锁MUTEX1", i_thread);
                    sleep(Duration::from_millis(10));
                    let guard = MUTEX1.try_lock();
                    println!("线程 {} 获取 MUTEX1 锁的结果: {:?}", i_thread, guard);
                }
            }
        }));
    }

    // 等子线程完成
    for child in children {
        let _ = child.join();
    }

    println!("死锁没有发生");
}
```

为了演示 `try_lock` 的作用，我们特定使用了之前必定会死锁的代码，并且将 `lock` 替换成 `try_lock`，与之前的结果不同，这段代码将不会再有死锁发生：

```
线程 0 锁住了MUTEX1, 接着准备去锁MUTEX2 !
线程 1 锁住了MUTEX2, 准备去锁MUTEX1
线程 1 获取 MUTEX1 锁的结果: Err("WouldBlock")
线程 0 获取 MUTEX2 锁的结果: Ok(0)
死锁没有发生
```

如上所示，当 `try_lock` 失败时，会报出一个错误：`Err("WouldBlock")`，接着线程中的剩余代码会继续执行，不会被阻塞。

---

一个有趣的命名规则：在 Rust 标准库中，使用 `try_xxx` 都会尝试进行一次操作，如果无法完成，就立即返回，不会发生阻塞。例如消息传递章节中的 `try_recv` 以及本章节中的 `try_lock`

---

## 读写锁 `RwLock`

`Mutex` 会对每次读写都进行加锁，但某些时候，我们需要大量的并发读，`Mutex` 就无法满足需求了，此时就可以使用 `RwLock`：

```

use std::sync::RwLock;

fn main() {
    let lock = RwLock::new(5);

    // 同一时间允许多个读
    {
        let r1 = lock.read().unwrap();
        let r2 = lock.read().unwrap();
        assert_eq!(r1, 5);
        assert_eq!(r2, 5);
    } // 读锁在此处被drop

    // 同一时间只允许一个写
    {
        let mut w = lock.write().unwrap();
        *w += 1;
        assert_eq!(w, 6);

        // 以下代码会阻塞发生死锁，因为读和写不允许同时存在
        // 写锁w直到该语句块结束才被释放，因此下面的读锁依然处于`w`的作用域中
        // let r1 = lock.read();
        // println!("{}:{}", r1);
    } // 写锁在此处被drop
}

```

`RwLock` 在使用上和 `Mutex` 区别不大，只有在多个读的情况下不阻塞程序，其他如读写、写读、写写情况下均会对后获取锁的操作进行阻塞。

我们也可以使用 `try_write` 和 `try_read` 来尝试进行一次写/读，若失败则返回错误：

```
Err("WouldBlock")
```

简单总结下 `RwLock`：

1. 同时允许多个读，但最多只能有一个写
2. 读和写不能同时存在
3. 读可以使用 `read`、`try_read`，写 `write`、`try_write`，在实际项目中，`try_xxx` 会安全的多

## Mutex 还是 RwLock

首先简单性上 `Mutex` 完胜，因为使用 `RwLock` 你得操心几个问题：

- 读和写不能同时发生，如果使用 `try_xxx` 解决，就必须做大量的错误处理和失败重试机制

- 当读多写少时，写操作可能会因为一直无法获得锁导致连续多次失败([writer starvation](#))
- RwLock 其实是操作系统提供的，实现原理要比 Mutex 复杂的多，因此单就锁的性能而言，比不上原生实现的 Mutex

再来简单总结下两者的使用场景：

- 追求高并发读取时，使用 RwLock，因为 Mutex 一次只允许一个线程去读取
- 如果要保证写操作的成功性，使用 Mutex
- 不知道哪个合适，统一使用 Mutex

需要注意的是，RwLock 虽然看上去貌似提供了高并发读取的能力，但这个不能说明它的性能比 Mutex 高，事实上 Mutex 性能要好不少，后者唯一的问题也仅仅在于不能并发读取。

一个常见的、错误的使用 RwLock 的场景就是使用 HashMap 进行简单读写，因为 HashMap 的读和写都非常快，RwLock 的复杂实现和相对低的性能反而会导致整体性能的降低，因此一般来说更适合使用 Mutex。

总之，如果你要使用 RwLock 要确保满足以下两个条件：**并发读，且需要对读到的资源进行"长时间"的操作**，HashMap 也许满足了并发读的需求，但是往往并不能满足后者：“长时间”的操作。

---

benchmark 永远是你在迷茫时最好的朋友！

---

## 三方库提供的锁实现

标准库在设计时总会存在取舍，因为往往性能并不是最好的，如果你追求性能，可以使用三方库提供的并发原语：

- [parking\\_lot](#), 功能更完善、稳定，社区较为活跃，star 较多，更新较为活跃
- [spin](#), 在多数场景中性能比 parking\_lot 高一点，最近没怎么更新

如果不是追求特别极致的性能，建议选择前者。

## 用条件变量(Condvar)控制线程的同步

Mutex 用于解决资源安全访问的问题，但是我们还需要一个手段来解决资源访问顺序的问题。而 Rust 考虑到了这一点，为我们提供了条件变量(Condition Variables)，它经常和 Mutex 一起使用，可以让线程挂起，直到某个条件发生后再继续执行，其实 Condvar 我们在之前的多线程章节就已经见到过，现在再来看一个不同的例子：

```

use std::sync::{Arc,Mutex,Condvar};
use std::thread::{spawn,sleep};
use std::time::Duration;

fn main() {
    let flag = Arc::new(Mutex::new(false));
    let cond = Arc::new(Condvar::new());
    let cflag = flag.clone();
    let ccond = cond.clone();

    let hdl = spawn(move || {
        let mut lock = cflag.lock().unwrap();
        let mut counter = 0;

        while counter < 3 {
            while !*lock {
                // wait方法会接收一个MutexGuard<'a, T>, 且它会自动地暂时释放这个锁, 使其他线程
                // 可以拿到锁并进行数据更新。
                // 同时当前线程在此处会被阻塞, 直到被其他地方notify后, 它会将原本的
                // MutexGuard<'a, T>还给我们, 即重新获取到了锁, 同时唤醒了此线程。
                lock = ccond.wait(lock).unwrap();
            }

            *lock = false;

            counter += 1;
            println!("inner counter: {}", counter);
        }
    });

    let mut counter = 0;
    loop {
        sleep(Duration::from_millis(1000));
        *flag.lock().unwrap() = true;
        counter += 1;
        if counter > 3 {
            break;
        }
        println!("outside counter: {}", counter);
        cond.notify_one();
    }
    hdl.join().unwrap();
    println!("{:?}", flag);
}

```

例子中通过主线程来触发子线程实现交替打印输出：

```
outside counter: 1
inner counter: 1
outside counter: 2
inner counter: 2
outside counter: 3
inner counter: 3
Mutex { data: true, poisoned: false, .. }
```

## 信号量 Semaphore

在多线程中，另一个重要的概念就是信号量，使用它可以让我们精准的控制当前正在运行的任务最大数量。想象一下，当一个新游戏刚开服时(有些较火的老游戏也会，比如 wow)，往往会影响游戏内玩家的同时在线数，一旦超过某个临界值，就开始进行排队进服。而在实际使用中，也有很多时候，我们需要通过信号量来控制最大并发数，防止服务器资源被撑爆。

本来 Rust 在标准库中有提供一个[信号量实现](#)，但是由于各种原因这个库现在已经不再推荐使用了，因此我们推荐使用 `tokio` 中提供的 `Semaphore` 实现：`tokio::sync::Semaphore`。

```
use std::sync::Arc;
use tokio::sync::Semaphore;

#[tokio::main]
async fn main() {
    let semaphore = Arc::new(Semaphore::new(3));
    let mut join_handles = Vec::new();

    for _ in 0..5 {
        let permit = semaphore.clone().acquire_owned().await.unwrap();
        join_handles.push(tokio::spawn(async move {
            //
            // 在这里执行任务...
            //
            drop(permit);
        }));
    }

    for handle in join_handles {
        handle.await.unwrap();
    }
}
```

上面代码创建了一个容量为 3 的信号量，当正在执行的任务超过 3 时，剩下的任务需要等待正在执行任务完成并减少信号量后到 3 以内时，才能继续执行。

这里的关键其实说白了就在于：信号量的申请和归还，使用前需要申请信号量，如果容量满了，就需要等待；使用后需要释放信号量，以便其它等待者可以继续。

## 总结

在很多时候，消息传递都是非常好用的手段，它可以让我们的数据在任务流水线上不断流转，实现起来非常优雅。

但是它并不能优雅的解决所有问题，因为我们面临的真实世界是非常复杂的，无法用某一种银弹统一解决。当面临消息传递不太适用的场景时，或者需要更好的性能和简洁性时，我们往往需要用锁来解决这些问题，因为锁允许多个线程同时访问同一个资源，简单粗暴。

除了锁之外，其实还有一种并发原语可以帮助我们解决并发访问数据的问题，那就是原子类型 Atomic，在下一章节中，我们会对其进行深入讲解。

# 线程同步：Atomic 原子类型与内存顺序

`Mutex` 用起来简单，但是无法并发读，`RwLock` 可以并发读，但是使用场景较为受限且性能不够，那么有没有一种全能性选手呢？欢迎我们的 `Atomic` 闪亮登场。

从 Rust1.34 版本后，就正式支持原子类型。原子指的是一系列不可被 CPU 上下文交换的机器指令，这些指令组合在一起就形成了原子操作。在多核 CPU 下，当某个 CPU 核心开始运行原子操作时，会先暂停其它 CPU 内核对内存的操作，以保证原子操作不会被其它 CPU 内核所干扰。

由于原子操作是通过指令提供的支持，因此它的性能相比锁和消息传递会好很多。相比较于锁而言，原子类型不需要开发者处理加锁和释放锁的问题，同时支持修改，读取等操作，还具备较高的并发性能，几乎所有的语言都支持原子类型。

可以看出原子类型是无锁类型，但是无锁不代表无需等待，因为原子类型内部使用了 `CAS` 循环，当大量的冲突发生时，该等待还是得等待！但是总归比锁要好。

---

`CAS` 全称是 Compare and swap，它通过一条指令读取指定的内存地址，然后判断其中的值是否等于给定的前置值，如果相等，则将其修改为新的值

---

## 使用 `Atomic` 作为全局变量

原子类型的一个常用场景，就是作为全局变量来使用：

```

use std::ops::Sub;
use std::sync::atomic::{AtomicU64, Ordering};
use std::thread::{self, JoinHandle};
use std::time::Instant;

const N_TIMES: u64 = 10000000;
const N_THREADS: usize = 10;

static R: AtomicU64 = AtomicU64::new(0);

fn add_n_times(n: u64) -> JoinHandle<()> {
    thread::spawn(move || {
        for _ in 0..n {
            R.fetch_add(1, Ordering::Relaxed);
        }
    })
}

fn main() {
    let s = Instant::now();
    let mut threads = Vec::with_capacity(N_THREADS);

    for _ in 0..N_THREADS {
        threads.push(add_n_times(N_TIMES));
    }

    for thread in threads {
        thread.join().unwrap();
    }

    assert_eq!(N_TIMES * N_THREADS as u64, R.load(Ordering::Relaxed));

    println!("{}:{:?}", Instant::now().sub(s));
}

```

以上代码启动了数个线程，每个线程都在疯狂对全局变量进行加 1 操作，最后将它与 线程数 \* 加1次数 进行比较，如果发生了因为多个线程同时修改导致了脏数据，那么这两个必将不相等。好在，它没有让我们失望，不仅快速的完成了任务，而且保证了 100%的并发安全性。

当然以上代码的功能其实也可以通过 Mutex 来实现，但是后者的强大功能是建立在额外的性能损耗基础上的，因此性能会逊色不少：

Atomic实现：673ms  
Mutex实现：1136ms

可以看到 Atomic 实现会比 Mutex 快41%，实际上在复杂场景下还能更快(甚至达到 4 倍的性能差距)！

还有一点值得注意: 和 Mutex 一样， Atomic 的值具有内部可变性，你无需将其声明为 mut：

```
use std::sync::Mutex;
use std::sync::atomic::{Ordering, AtomicU64};

struct Counter {
    count: u64
}

fn main() {
    let n = Mutex::new(Counter {
        count: 0
    });

    n.lock().unwrap().count += 1;

    let n = AtomicU64::new(0);

    n.fetch_add(0, Ordering::Relaxed);
}
```

这里有一个奇怪的枚举成员 `Ordering::Relaxed`, 看上去很像是排序作用, 但是我们并没有做排序操作啊? 实际上它用于控制原子操作使用的**内存顺序**。

## 内存顺序

内存顺序是指 CPU 在访问内存时的顺序, 该顺序可能受以下因素的影响:

- 代码中的先后顺序
- 编译器优化导致在编译阶段发生改变(内存重排序 reordering)
- 运行阶段因 CPU 的缓存机制导致顺序被打乱

### 编译器优化导致内存顺序的改变

对于第二点, 我们举个例子:

```

static mut X: u64 = 0;
static mut Y: u64 = 1;

fn main() {
    ...           // A

    unsafe {
        ...         // B
        X = 1;
        ...         // C
        Y = 3;
        ...         // D
        X = 2;
        ...         // E
    }
}

```

假如在 C 和 D 代码片段中，根本没有用到 `X = 1`，那么编译器很可能会将 `X = 1` 和 `X = 2` 进行合并：

```

...           // A

unsafe {
    ...         // B
    X = 2;
    ...         // C
    Y = 3;
    ...         // D
    ...         // E
}

```

若代码 A 中创建了一个新的线程用于读取全局静态变量 `X`，则该线程将无法读取到 `X = 1` 的结果，因为在编译阶段就已经被优化掉。

## CPU 缓存导致的内存顺序的改变

假设之前的 `X = 1` 没有被优化掉，并且在代码片段 A 中有一个新的线程：

```

initial state: X = 0, Y = 1

THREAD Main      THREAD A
X = 1;           if X == 1 {
Y = 3;           Y *= 2;
X = 2;           }

```

我们来讨论下以上线程状态，`Y` 最终的可能值(可能性依次降低)：

- `Y = 3`：线程 Main 运行完后才运行线程 A，或者线程 A 运行完后再运行线程 Main

- $Y = 6$ : 线程 Main 的  $Y = 3$  运行完, 但  $X = 2$  还没被运行, 此时线程 A 开始运行  $Y *= 2$ , 最后才运行 Main 线程的  $X = 2$
- $Y = 2$ : 线程 Main 正在运行  $Y = 3$  还没结束, 此时线程 A 正在运行  $Y *= 2$ , 因此 Y 取到了值 1, 然后 Main 的线程将 Y 设置为 3, 紧接着就被线程 A 的  $Y = 2$  所覆盖
- $Y = 2$ : 上面的还只是一般的数据竞争, 这里虽然产生了相同的结果 2, 但是背后的原理大相径庭: 线程 Main 运行完  $Y = 3$ , 但是 CPU 缓存中的  $Y = 3$  还没有被同步到其它 CPU 缓存中, 此时线程 A 中的  $Y *= 2$  就开始读取 Y, 结果读到了值 1, 最终计算出结果 2

甚至更改成:

```
initial state: X = 0, Y = 1

THREAD Main      THREAD A
X = 1;           if X == 2 {
Y = 3;           Y *= 2;
X = 2;           }
```

还是可能出现  $Y = 2$ , 因为 Main 线程中的 X 和 Y 被同步到其它 CPU 缓存中的顺序未必一致。

## 限定内存顺序的 5 个规则

在理解了内存顺序可能存在的改变后, 你就可以明白为什么 Rust 提供了 `Ordering::Relaxed` 用于限定内存顺序了, 事实上, 该枚举有 5 个成员:

- **Relaxed**, 这是最宽松的规则, 它对编译器和 CPU 不做任何限制, 可以乱序
- **Release 释放**, 设定内存屏障(Memory barrier), 保证它之前的操作永远在它之前, 但是它后面的操作可能被重排到它前面
- **Acquire 获取**, 设定内存屏障, 保证在它之后的访问永远在它之后, 但是它之前的操作却有可能被重排到它后面, 往往和 Release 在不同线程中联合使用
- **AcqRel**, 是 Acquire 和 Release 的结合, 同时拥有它们俩提供的保证。比如你要对一个 `atomic` 自增 1, 同时希望该操作之前和之后的读取或写入操作不会被重新排序
- **SeqCst 顺序一致性**, SeqCst 就像是 AcqRel 的加强版, 它不管原子操作是属于读取还是写入的操作, 只要某个线程有用到 SeqCst 的原子操作, 线程中该 SeqCst 操作前的数据操作绝对不会被重新排在该 SeqCst 操作之后, 且该 SeqCst 操作后的数据操作也绝对不会被重新排在 SeqCst 操作前。

这些规则由于是系统提供的, 因此其它语言提供的相应规则也大同小异, 大家如果不明白可以看看其它语言的相关解释。

## 内存屏障的例子

下面我们以 `Release` 和 `Acquire` 为例, 使用它们构筑出一对内存屏障, 防止编译器和 CPU 将屏障前(`Release`)和屏障后(`Acquire`)中的数据操作重新排在屏障围成的范围之外:

```

use std::thread::{self, JoinHandle};
use std::sync::atomic::{Ordering, AtomicBool};

static mut DATA: u64 = 0;
static READY: AtomicBool = AtomicBool::new(false);

fn reset() {
    unsafe {
        DATA = 0;
    }
    READY.store(false, Ordering::Relaxed);
}

fn producer() -> JoinHandle<()> {
    thread::spawn(move || {
        unsafe {
            DATA = 100; // A
        }
        READY.store(true, Ordering::Release); // B: 内存屏障 ↑
    })
}

fn consumer() -> JoinHandle<()> {
    thread::spawn(move || {
        while !READY.load(Ordering::Acquire) {} // C: 内存屏障 ↓

        assert_eq!(100, unsafe { DATA }); // D
    })
}

fn main() {
    loop {
        reset();

        let t_producer = producer();
        let t_consumer = consumer();

        t_producer.join().unwrap();
        t_consumer.join().unwrap();
    }
}

```

原则上，Acquire 用于读取，而 Release 用于写入。但是由于有些原子操作同时拥有读取和写入的功能，此时就需要使用 AcqRel 来设置内存顺序了。在内存屏障中被写入的数据，都可以被其它线程读取到，不会有 CPU 缓存的问题。

## 内存顺序的选择

1. 不知道怎么选择时，优先使用 SeqCst，虽然会稍微减慢速度，但是慢一点也比出现错误好

2. 多线程只计数 `fetch_add` 而不使用该值触发其他逻辑分支的简单使用场景，可以使用 `Relaxed`  
参考 [Which std::sync::atomic::Ordering to use?](#)

## 多线程中使用 Atomic

在多线程环境中要使用 `Atomic` 需要配合 `Arc`：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{hint, thread};

fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));

    let spinlock_clone = Arc::clone(&spinlock);
    let thread = thread::spawn(move|| {
        spinlock_clone.store(0, Ordering::SeqCst);
    });

    // 等待其它线程释放锁
    while spinlock.load(Ordering::SeqCst) != 0 {
        hint::spin_loop();
    }

    if let Err(panic) = thread.join() {
        println!("Thread had an error: {:?}", panic);
    }
}
```

## Atomic 能替代锁吗

那么原子类型既然这么全能，它可以替代锁吗？答案是不行：

- 对于复杂的场景下，锁的使用简单粗暴，不容易有坑
- `std::sync::atomic` 包中仅提供了数值类型的原子操作：`AtomicBool`, `AtomicIsize`, `AtomicUsize`, `AtomicI8`, `AtomicU16` 等，而锁可以应用于各种类型
- 在有些情况下，必须使用锁来配合，例如上一章节中使用 `Mutex` 配合 `Condvar`

## Atomic 的应用场景

事实上，`Atomic` 虽然对于用户不太常用，但是对于高性能库的开发者、标准库开发者都非常常用，它是并发原语的基石，除此之外，还有一些场景适用：

- 无锁(lock free)数据结构
- 全局变量，例如全局自增 ID，在后续章节会介绍
- 跨线程计数器，例如可以用于统计指标

以上列出的只是 `Atomic` 适用的部分场景，具体场景需要大家未来根据自己的需求进行权衡选择。

# 基于 Send 和 Sync 的线程安全

为何 Rc、RefCell 和裸指针不可以在多线程间使用？如何让裸指针可以在多线程使用？我们一起来探寻下这些问题的答案。

## 无法用于多线程的Rc

先来看一段多线程使用 Rc 的代码：

```
use std::thread;
use std::rc::Rc;
fn main() {
    let v = Rc::new(5);
    let t = thread::spawn(move || {
        println!("{}", v);
    });

    t.join().unwrap();
}
```

以上代码将 v 的所有权通过 move 转移到子线程中，看似正确实则会报错：

```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
----- 省略部分报错 -----
 = help: within `[closure@src/main.rs:5:27: 7:6]`, the trait `Send` is not
 implemented for `Rc<i32>`
```

表面原因是 Rc 无法在线程间安全的转移，实际是编译器给予我们的那句帮助：the trait `Send` is not implemented for `Rc<i32>`（Rc<i32> 未实现 Send 特征），那么此处的 Send 特征又是何方神圣？

## Rc 和 Arc 源码对比

在介绍 Send 特征之前，再来看看 Arc 为何可以在多线程使用，玄机在于两者的源码实现上：

```

// Rc源码片段
impl<T: ?Sized> !marker::Send for Rc<T> {}
impl<T: ?Sized> !marker::Sync for Rc<T> {}

// Arc源码片段
unsafe impl<T: ?Sized + Sync + Send> Send for Arc<T> {}
unsafe impl<T: ?Sized + Sync + Send> Sync for Arc<T> {}

```

! 代表移除特征的相应实现，上面代码中 `Rc<T>` 的 `Send` 和 `Sync` 特征被特地移除了实现，而 `Arc<T>` 则相反，实现了 `Sync + Send`，再结合之前的编译器报错，大概可以明白了：`Send` 和 `Sync` 是在线程间安全使用一个值的关键。

## Send 和 Sync

`Send` 和 `Sync` 是 Rust 安全并发的重中之重，但是实际上它们只是标记特征(marker trait)，该特征未定义任何行为，因此非常适合用于标记)，来看看它们的作用：

- 实现 `Send` 的类型可以在线程间安全的传递其所有权
- 实现 `Sync` 的类型可以在线程间安全的共享(通过引用)

这里还有一个潜在的依赖：一个类型要在线程间安全的共享的前提是，指向它的引用必须能在线程间传递。因为如果引用都不能被传递，我们就无法在多个线程间使用引用去访问同一个数据了。

由上可知，**若类型 T 的引用 &T 是 Send，则 T 是 Sync。**

没有例子的概念讲解都是耍流氓，来看看 `RwLock` 的实现：

```
unsafe impl<T: ?Sized + Send + Sync> Sync for RwLock<T> {}
```

首先 `RwLock` 可以在线程间安全的共享，那它肯定是实现了 `Sync`，但是我们的关注点不在这里。众所周知，`RwLock` 可以并发的读，说明其中的值 `T` 必定也可以在线程间共享，那 `T` 必定要实现 `Sync`。

果不其然，上述代码中，`T` 的特征约束中就有一个 `Sync` 特征，那问题又来了，`Mutex` 是不是相反？再来看看：

```
unsafe impl<T: ?Sized + Send> Sync for Mutex<T> {}
```

不出所料，`Mutex<T>` 中的 `T` 并没有 `Sync` 特征约束。

武学秘籍再好，不见生死也是花拳绣腿。同样的，我们需要通过实战来彻底掌握 `Send` 和 `Sync`，但在实战之前，先来简单看看有哪些类型实现了它们。

## 实现Send和Sync的类型

在 Rust 中，几乎所有类型都默认实现了 Send 和 Sync，而且由于这两个特征都是可自动派生的特征(通过 derive 派生)，意味着一个复合类型(例如结构体)，只要它内部的所有成员都实现了 Send 或者 Sync，那么它就自动实现了 Send 或 Sync。

正是因为以上规则，Rust 中绝大多数类型都实现了 Send 和 Sync，除了以下几个(事实上不止这几个，只不过它们比较常见)：

- 裸指针两者都没实现，因为它本身就没有安全保证
- UnsafeCell 不是 Sync，因此 Cell 和 RefCell 也不是
- Rc 两者都没实现(因为内部的引用计数器不是线程安全的)

当然，如果是自定义的复合类型，那没实现那哥俩的就较为常见了：**只要复合类型中有一个成员不是 Send 或 Sync，那么该复合类型也就不是 Send 或 Sync。**

**手动实现 Send 和 Sync 是不安全的**，通常并不需要手动实现 Send 和 Sync trait，实现者需要使用 unsafe 小心维护并发安全保证。

至此，相关的概念大家已经掌握，但是我敢肯定，对于这两个滑不溜秋的家伙，大家依然会非常模糊，不知道它们该如何使用。那么我们来一起看看如何让裸指针可以在线程间安全的使用。

## 为裸指针实现Send

上面我们提到裸指针既没实现 Send，意味着下面代码会报错：

```
use std::thread;
fn main() {
    let p = 5 as *mut u8;
    let t = thread::spawn(move || {
        println!("{:?}", p);
    });

    t.join().unwrap();
}
```

报错跟之前无二：`\*mut u8` cannot be sent between threads safely，但是有一个问题，我们无法为其直接实现 Send 特征，好在可以用 newtype 类型：struct MyBox(\*mut u8)；。

还记得之前的规则吗：复合类型中有一个成员没实现 Send，该复合类型就不是 Send，因此我们需要手动为它实现：

```
use std::thread;

#[derive(Debug)]
struct MyBox(*mut u8);
unsafe impl Send for MyBox {}

fn main() {
    let p = MyBox(5 as *mut u8);
    let t = thread::spawn(move || {
        println!("{:?}", p);
    });

    t.join().unwrap();
}
```

此时，我们的指针已经可以欢快的在多线程间撒欢，以上代码很简单，但有一点需要注意：`Send` 和 `Sync` 是 `unsafe` 特征，实现时需要用 `unsafe` 代码块包裹。

## 为裸指针实现`Sync`

由于 `Sync` 是多线程间共享一个值，大家可能会想这么实现：

```
use std::thread;
fn main() {
    let v = 5;
    let t = thread::spawn(|| {
        println!("{:?}", &v);
    });

    t.join().unwrap();
}
```

关于这种用法，在多线程章节也提到过，线程如果直接去借用其它线程的变量，会报错：`closure may outlive the current function`，原因在于编译器无法确定主线程 `main` 和子线程 `t` 谁的生命周期更长，特别是当两个线程都是子线程时，没有任何人知道哪个子线程会先结束，包括编译器！

因此我们得配合 `Arc` 去使用：

```

use std::thread;
use std::sync::Arc;
use std::sync::Mutex;

#[derive(Debug)]
struct MyBox(*const u8);
unsafe impl Send for MyBox {}

fn main() {
    let b = &MyBox(5 as *const u8);
    let v = Arc::new(Mutex::new(b));
    let t = thread::spawn(move || {
        let _v1 = v.lock().unwrap();
    });

    t.join().unwrap();
}

```

上面代码将智能指针 `v` 的所有权转移给新线程，同时 `v` 包含了一个引用类型 `b`，当在新的线程中试图获取内部的引用时，会报错：

```

error[E0277]: `*const u8` cannot be shared between threads safely
--> src/main.rs:25:13
|
25 |     let t = thread::spawn(move || {
|           ^^^^^^^^^^^^^ `*const u8` cannot be shared between threads safely
|
= help: within `MyBox`, the trait `Sync` is not implemented for `*const u8`

```

因为我们访问的引用实际上还是对主线程中的数据的借用，转移进来的仅仅是外层的智能指针引用。要解决很简单，为 `MyBox` 实现 `Sync`：

```
unsafe impl Sync for MyBox {}
```

## 总结

通过上面的两个裸指针的例子，我们了解了如何实现 `Send` 和 `Sync`，以及如何只实现 `Send` 而不实现 `Sync`，简单总结下：

1. 实现 `Send` 的类型可以在线程间安全的传递其所有权，实现 `Sync` 的类型可以在线程间安全的共享(通过引用)
2. 绝大部分类型都实现了 `Send` 和 `Sync`，常见的未实现的有：裸指针、`Cell`、`RefCell`、`Rc` 等
3. 可以为自定义类型实现 `Send` 和 `Sync`，但是需要 `unsafe` 代码块

4. 可以为部分 Rust 中的类型实现 `Send`、`Sync`，但是需要使用 `newtype`，例如文中的裸指针例子

# 全局变量

在一些场景，我们可能需要全局变量来简化状态共享的代码，包括全局 ID，全局数据存储等等，下面一起来看看有哪些创建全局变量的方法。

首先，有一点可以肯定，全局变量的生命周期肯定是 `'static`，但是不代表它需要用 `static` 来声明，例如常量、字符串字面值等无需使用 `static` 进行声明，原因是它们已经被打包到二进制可执行文件中。

下面我们从编译期初始化及运行期初始化两个类别来介绍下全局变量有哪些类型及该如何使用。

## 编译期初始化

我们大多数使用的全局变量都只需要在编译期初始化即可，例如静态配置、计数器、状态值等等。

### 静态常量

全局常量可以在程序任何一部分使用，当然，如果它是定义在某个模块中，你需要引入对应的模块才能使用。常量，顾名思义它是不可变的，很适合用作静态配置：

```
const MAX_ID: usize = usize::MAX / 2;
fn main() {
    println!("用户ID允许的最大值是{}", MAX_ID);
}
```

### 常量与普通变量的区别

- 关键字是 `const` 而不是 `let`
- 定义常量必须指明类型（如 `i32`）不能省略
- 定义常量时变量的命名规则一般是全部大写
- 常量可以在任意作用域进行定义，其生命周期贯穿整个程序的生命周期。编译时编译器会尽可能将其内联到代码中，所以在不同地方对同一常量的引用并不能保证引用到相同的内存地址
- 常量的赋值只能是常量表达式/数学表达式，也就是说必须是在编译期就能计算出的值，如果需要在运行时才能得出结果的值比如函数，则不能赋值给常量表达式
- 对于变量出现重复的定义(绑定)会发生变量遮盖，后面定义的变量会遮住前面定义的变量，常量则不允许出现重复的定义

## 静态变量

静态变量允许声明一个全局的变量，常用于全局数据统计，例如我们希望用一个变量来统计程序当前的总请求数：

```
static mut REQUEST_RECV: usize = 0;
fn main() {
    unsafe {
        REQUEST_RECV += 1;
        assert_eq!(REQUEST_RECV, 1);
    }
}
```

Rust 要求必须使用 `unsafe` 语句块才能访问和修改 `static` 变量，因为这种使用方式往往并不安全，其实编译器是对的，当在多线程中同时去修改时，会不可避免的遇到脏数据。

只有在同一线程内或者不在乎数据的准确性时，才应该使用全局静态变量。

和常量相同，定义静态变量的时候必须赋值为在编译期就可以计算出的值(常量表达式/数学表达式)，不能是运行时才能计算出的值(如函数)

## 静态变量和常量的区别

- 静态变量不会被内联，在整个程序中，静态变量只有一个实例，所有的引用都会指向同一个地址
- 存储在静态变量中的值必须要实现 `Sync` trait

## 原子类型

想要全局计数器、状态控制等功能，又想要线程安全的实现，原子类型是非常好的办法。

```
use std::sync::atomic::{AtomicUsize, Ordering};
static REQUEST_RECV: AtomicUsize = AtomicUsize::new(0);
fn main() {
    for _ in 0..100 {
        REQUEST_RECV.fetch_add(1, Ordering::Relaxed);
    }

    println!("当前用户请求数{:?}", REQUEST_RECV);
}
```

关于原子类型的讲解看[这篇文章](#)

## 示例：全局 ID 生成器

来看看如何使用上面的内容实现一个全局 ID 生成器：

```

use std::sync::atomic::{Ordering, AtomicUsize};

struct Factory{
    factory_id: usize,
}

static GLOBAL_ID_COUNTER: AtomicUsize = AtomicUsize::new(0);
const MAX_ID: usize = usize::MAX / 2;

fn generate_id()->usize{
    // 检查两次溢出，否则直接加一可能导致溢出
    let current_val = GLOBAL_ID_COUNTER.load(Ordering::Relaxed);
    if current_val > MAX_ID{
        panic!("Factory ids overflowed");
    }
    GLOBAL_ID_COUNTER.fetch_add(1, Ordering::Relaxed)
    let next_id = GLOBAL_ID_COUNTER.load(Ordering::Relaxed);
    if next_id > MAX_ID{
        panic!("Factory ids overflowed");
    }
    next_id
}

impl Factory{
    fn new()->Self{
        Self{
            factory_id: generate_id()
        }
    }
}

```

## 运行期初始化

以上的静态初始化有一个致命的问题：无法用函数进行静态初始化，例如你如果想声明一个全局的 Mutex 锁：

```

use std::sync::Mutex;
static NAMES: Mutex<String> = Mutex::new(String::from("Sunface, Jack, Allen"));

fn main() {
    let v = NAMES.lock().unwrap();
    println!("{}" ,v);
}

```

运行后报错如下：

```
error[E0015]: calls in statics are limited to constant functions, tuple structs and
tuple variants
--> src/main.rs:3:42
|
3 |     static NAMES: Mutex<String> = Mutex::new(String::from("sunface"));
```

但你又必须在声明时就对 NAMES 进行初始化，此时就陷入了两难的境地。好在天无绝人之路，我们可以使用 `lazy_static` 包来解决这个问题。

## lazy\_static

`lazy_static` 是社区提供的非常强大的宏，用于懒初始化静态变量，之前的静态变量都是在编译期初始化的，因此无法使用函数调用进行赋值，而 `lazy_static` 允许我们在运行期初始化静态变量！

```
use std::sync::Mutex;
use lazy_static::lazy_static;
lazy_static! {
    static ref NAMES: Mutex<String> = Mutex::new(String::from("Sunface, Jack,
Allen"));
}

fn main() {
    let mut v = NAMES.lock().unwrap();
    v.push_str(", Myth");
    println!("{}" ,v);
}
```

当然，使用 `lazy_static` 在每次访问静态变量时，会有轻微的性能损失，因为其内部实现用了一个底层的并发原语 `std::sync::Once`，在每次访问该变量时，程序都会执行一次原子指令用于确认静态变量的初始化是否完成。

`lazy_static` 宏，匹配的是 `static ref`，所以定义的静态变量都是不可变引用

可能有读者会问，为何需要在运行期初始化一个静态变量，除了上面的全局锁，你会遇到最常见的场景就是：**一个全局的动态配置，它在程序开始后，才加载数据进行初始化，最终可以让各个线程直接访问使用**

再来看一个使用 `lazy_static` 实现全局缓存的例子：

```

use lazy_static::lazy_static;
use std::collections::HashMap;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "foo");
        m.insert(1, "bar");
        m.insert(2, "baz");
        m
    };
}

fn main() {
    // 首次访问`HASHMAP`的同时对其进行初始化
    println!("The entry for `0` is `{}`.`", HASHMAP.get(&0).unwrap());

    // 后续的访问仅仅获取值，再不会进行任何初始化操作
    println!("The entry for `1` is `{}`.`", HASHMAP.get(&1).unwrap());
}

```

需要注意的是，`lazy_static` 直到运行到 `main` 中的第一行代码时，才进行初始化，非常 `lazy static`。

## Box::leak

在 `Box` 智能指针章节中，我们提到了 `Box::leak` 可以用于全局变量，例如用作运行期初始化的全局动态配置，先来看看如果不使用 `lazy_static` 也不使用 `Box::leak`，会发生什么：

```

#[derive(Debug)]
struct Config {
    a: String,
    b: String,
}
static mut CONFIG: Option<&mut Config> = None;

fn main() {
    unsafe {
        CONFIG = Some(&mut Config {
            a: "A".to_string(),
            b: "B".to_string(),
        });

        println!("{:?}", CONFIG)
    }
}

```

以上代码我们声明了一个全局动态配置 CONFIG，并且其值初始化为 None，然后在程序开始运行后，给它赋予相应的值，运行后报错：

```
error[E0716]: temporary value dropped while borrowed
--> src/main.rs:10:28
|
10 |         CONFIG = Some(&mut Config {
|             |-----^
|             |
11 |             ||         a: "A".to_string(),
12 |             ||         b: "B".to_string(),
13 |             ||     });
|             ||-----^-- temporary value is freed at the end of this statement
|             ||-----|| assignment requires that borrow lasts for `'static` creates a temporary which is freed while still in use
```

可以看到，Rust 的借用和生命周期规则限制了我们做到这一点，因为试图将一个局部生命周期的变量赋值给全局生命周期的 CONFIG，这明显是不安全的。

好在 Rust 为我们提供了 Box::leak 方法，它可以将一个变量从内存中泄漏(听上去怪怪的，竟然做主动内存泄漏)，然后将其变为 'static 生命周期，最终该变量将和程序活得一样久，因此可以赋值给全局静态变量 CONFIG。

```
#[derive(Debug)]
struct Config {
    a: String,
    b: String
}
static mut CONFIG: Option<&mut Config> = None;

fn main() {
    let c = Box::new(Config {
        a: "A".to_string(),
        b: "B".to_string(),
    });

    unsafe {
        // 将`c`从内存中泄漏，变成`'static`生命周期
        CONFIG = Some(Box::leak(c));
        println!("{:?}", CONFIG);
    }
}
```

## 从函数中返回全局变量

问题又来了，如果我们需要在运行期，从一个函数返回一个全局变量该如何做？例如：

```
#[derive(Debug)]
struct Config {
    a: String,
    b: String,
}
static mut CONFIG: Option<&mut Config> = None;

fn init() -> Option<&'static mut Config> {
    Some(&mut Config {
        a: "A".to_string(),
        b: "B".to_string(),
    })
}

fn main() {
    unsafe {
        CONFIG = init();

        println!("{:?}", CONFIG)
    }
}
```

报错这里就不展示了，跟之前大同小异，还是生命周期引起的，那么该如何解决呢？依然可以用 `Box::leak`：

```
#[derive(Debug)]
struct Config {
    a: String,
    b: String,
}
static mut CONFIG: Option<&mut Config> = None;

fn init() -> Option<&'static mut Config> {
    let c = Box::new(Config {
        a: "A".to_string(),
        b: "B".to_string(),
    });
    Some(Box::leak(c))
}

fn main() {
    unsafe {
        CONFIG = init();

        println!("{:?}", CONFIG)
    }
}
```

## 标准库中的 OnceCell

在 Rust 标准库中提供了实验性的 `lazy::OnceCell` 和 `lazy::SyncOnceCell` (在 Rust 1.70.0 版本及以上标准库中，替换为稳定的 `cell::OnceCell` 和 `sync::OnceLock`) 两种 `Cell`，前者用于单线程，后者用于多线程，它们用来存储堆上的信息，并且具有最多只能赋值一次的特性。如实现一个多线程的日志组件 `Logger`：

```

// 低于Rust 1.70版本中， OnceCell 和 SyncOnceCell 的API为实验性的 ,
// 需启用特性 `#![feature(once_cell)]`。
#![feature(once_cell)]
use std::lazy::SyncOnceCell, thread;

// Rust 1.70版本以上,
// use std::sync::OnceLock, thread;

fn main() {
    // 子线程中调用
    let handle = thread::spawn(|| {
        let logger = Logger::global();
        logger.log("thread message".to_string());
    });

    // 主线程调用
    let logger = Logger::global();
    logger.log("some message".to_string());

    let logger2 = Logger::global();
    logger2.log("other message".to_string());

    handle.join().unwrap();
}

#[derive(Debug)]
struct Logger;

// 低于Rust 1.70版本
static LOGGER: SyncOnceCell<Logger> = SyncOnceCell::new();

// Rust 1.70版本以上
// static LOGGER: OnceLock<Logger> = OnceLock::new();

impl Logger {
    fn global() -> &'static Logger {
        // 获取或初始化 Logger
        LOGGER.get_or_init(|| {
            println!("Logger is being created..."); // 初始化打印
            Logger
        })
    }

    fn log(&self, message: String) {
        println!("{}", message)
    }
}

```

以上代码我们声明了一个 `global()` 关联函数，并在其内部调用 `get_or_init` 进行初始化 `Logger`，之后在不同线程上多次调用 `Logger::global()` 获取其实例：

```
Logger is being created...
some message
other message
thread message
```

可以看到，`Logger is being created...` 在多个线程中使用也只被打印了一次。

特别注意，目前 `OnceCell` 和 `SyncOnceCell` API 暂未稳定，需启用特性 `#![feature(once_cell)]`。

## 总结

在 Rust 中有很多方式可以创建一个全局变量，本章也只是介绍了其中一部分，更多的还等待大家自己去挖掘学习(当然，未来可能本章节会不断完善，最后变成一个巨无霸-，-)。

简单来说，全局变量可以分为两种：

- 编译期初始化的全局变量，`const` 创建常量，`static` 创建静态变量，`Atomic` 创建原子类型
- 运行期初始化的全局变量，`lazy_static` 用于懒初始化，`Box::leak` 利用内存泄漏将一个变量的生命周期变为`'static`

# 错误处理

在之前的[返回值和错误处理](#)章节中，我们学习了几个重要的概念，例如 `Result` 用于返回结果处理，`?` 用于错误的传播，若大家对此还较为模糊，强烈建议回头温习下。

在本章节中一起来看看如何对 `Result` (`Option`) 做进一步的处理，以及如何定义自己的错误类型。

## 组合器

在设计模式中，有一个组合器模式，相信有 Java 背景的同学对此并不陌生。

---

将对象组合成树形结构以表示“部分整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。-GoF <<设计模式>>

---

与组合器模式有所不同，在 Rust 中，组合器更多的是用于对返回结果的类型进行变换：例如使用 `ok_or` 将一个 `Option` 类型转换成 `Result` 类型。

下面我们来看看一些常见的组合器。

### `or()` 和 `and()`

跟布尔关系的与/或很像，这两个方法会对两个表达式做逻辑组合，最终返回 `Option` / `Result`。

- `or()`，表达式按照顺序求值，若任何一个表达式的结果是 `Some` 或 `Ok`，则该值会立刻返回
- `and()`，若两个表达式的结果都是 `Some` 或 `Ok`，则**第二个表达式中的值被返回**。若任何一个的结果是 `None` 或 `Err`，则立刻返回。

实际上，只要将布尔表达式的 `true` / `false`，替换成 `Some` / `None` 或 `Ok` / `Err` 就很好理解了。

```

fn main() {
    let s1 = Some("some1");
    let s2 = Some("some2");
    let n: Option<&str> = None;

    let o1: Result<&str, &str> = Ok("ok1");
    let o2: Result<&str, &str> = Ok("ok2");
    let e1: Result<&str, &str> = Err("error1");
    let e2: Result<&str, &str> = Err("error2");

    assert_eq!(s1.or(s2), s1); // Some1 or Some2 = Some1
    assert_eq!(s1.or(n), s1); // Some or None = Some
    assert_eq!(n.or(s1), s1); // None or Some = Some
    assert_eq!(n.or(n), n); // None1 or None2 = None2

    assert_eq!(o1.or(o2), o1); // Ok1 or Ok2 = Ok1
    assert_eq!(o1.or(e1), o1); // Ok or Err = Ok
    assert_eq!(e1.or(o1), o1); // Err or Ok = Ok
    assert_eq!(e1.or(e2), e2); // Err1 or Err2 = Err2

    assert_eq!(s1.and(s2), s2); // Some1 and Some2 = Some2
    assert_eq!(s1.and(n), n); // Some and None = None
    assert_eq!(n.and(s1), n); // None and Some = None
    assert_eq!(n.and(n), n); // None1 and None2 = None1

    assert_eq!(o1.and(o2), o2); // Ok1 and Ok2 = Ok2
    assert_eq!(o1.and(e1), e1); // Ok and Err = Err
    assert_eq!(e1.and(o1), e1); // Err and Ok = Err
    assert_eq!(e1.and(e2), e1); // Err1 and Err2 = Err1
}

```

除了 `or` 和 `and` 之外，Rust 还为我们提供了 `xor`，但是它只能应用在 `Option` 上，其实想想也是这个道理，如果能应用在 `Result` 上，那你又该如何对一个值和错误进行异或操作？

### `or_else()` 和 `and_then()`

它们跟 `or()` 和 `and()` 类似，唯一的区别在于，它们的第二个表达式是一个闭包。

```
fn main() {
    // or_else with Option
    let s1 = Some("some1");
    let s2 = Some("some2");
    let fn_some = || Some("some2"); // 类似于: let fn_some = || -> Option<&str> {
        Some("some2") };

    let n: Option<&str> = None;
    let fn_none = || None;

    assert_eq!(s1.or_else(fn_some), s1); // Some1 or_else Some2 = Some1
    assert_eq!(s1.or_else(fn_none), s1); // Some or_else None = Some
    assert_eq!(n.or_else(fn_some), s2); // None or_else Some = Some
    assert_eq!(n.or_else(fn_none), None); // None1 or_else None2 = None2

    // or_else with Result
    let o1: Result<&str, &str> = Ok("ok1");
    let o2: Result<&str, &str> = Ok("ok2");
    let fn_ok = |_| Ok("ok2"); // 类似于: let fn_ok = |_| -> Result<&str, &str> {
        Ok("ok2") };

    let e1: Result<&str, &str> = Err("error1");
    let e2: Result<&str, &str> = Err("error2");
    let fn_err = |_| Err("error2");

    assert_eq!(o1.or_else(fn_ok), o1); // Ok1 or_else Ok2 = Ok1
    assert_eq!(o1.or_else(fn_err), o1); // Ok or_else Err = Ok
    assert_eq!(e1.or_else(fn_ok), o2); // Err or_else Ok = Ok
    assert_eq!(e1.or_else(fn_err), e2); // Err1 or_else Err2 = Err2
}
```

```

fn main() {
    // and_then with Option
    let s1 = Some("some1");
    let s2 = Some("some2");
    let fn_some = |_| Some("some2"); // 类似于: let fn_some = |_| -> Option<&str> {
    Some("some2") };

    let n: Option<&str> = None;
    let fn_none = |_| None;

    assert_eq!(s1.and_then(fn_some), s2); // Some1 and_then Some2 = Some2
    assert_eq!(s1.and_then(fn_none), n); // Some and_then None = None
    assert_eq!(n.and_then(fn_some), n); // None and_then Some = None
    assert_eq!(n.and_then(fn_none), n); // None1 and_then None2 = None1

    // and_then with Result
    let o1: Result<&str, &str> = Ok("ok1");
    let o2: Result<&str, &str> = Ok("ok2");
    let fn_ok = |_| Ok("ok2"); // 类似于: let fn_ok = |_| -> Result<&str, &str> {
    Ok("ok2") };

    let e1: Result<&str, &str> = Err("error1");
    let e2: Result<&str, &str> = Err("error2");
    let fn_err = |_| Err("error2");

    assert_eq!(o1.and_then(fn_ok), o2); // Ok1 and_then Ok2 = Ok2
    assert_eq!(o1.and_then(fn_err), e2); // Ok and_then Err = Err
    assert_eq!(e1.and_then(fn_ok), e1); // Err and_then Ok = Err
    assert_eq!(e1.and_then(fn_err), e1); // Err1 and_then Err2 = Err1
}

```

## filter

filter 用于对 Option 进行过滤:

```

fn main() {
    let s1 = Some(3);
    let s2 = Some(6);
    let n = None;

    let fn_is_even = |x: &i8| x % 2 == 0;

    assert_eq!(s1.filter(fn_is_even), n); // Some(3) -> 3 is not even -> None
    assert_eq!(s2.filter(fn_is_even), s2); // Some(6) -> 6 is even -> Some(6)
    assert_eq!(n.filter(fn_is_even), n); // None -> no value -> None
}

```

## map() 和 map\_err()

map 可以将 Some 或 Ok 中的值映射为另一个：

```
fn main() {
    let s1 = Some("abcde");
    let s2 = Some(5);

    let n1: Option<&str> = None;
    let n2: Option<usize> = None;

    let o1: Result<&str, &str> = Ok("abcde");
    let o2: Result<usize, &str> = Ok(5);

    let e1: Result<&str, &str> = Err("abcde");
    let e2: Result<usize, &str> = Err("abcde");

    let fn_character_count = |s: &str| s.chars().count();

    assert_eq!(s1.map(fn_character_count), s2); // Some1 map = Some2
    assert_eq!(n1.map(fn_character_count), n2); // None1 map = None2

    assert_eq!(o1.map(fn_character_count), o2); // Ok1 map = Ok2
    assert_eq!(e1.map(fn_character_count), e2); // Err1 map = Err2
}
```

但是如果你想要将 Err 中的值进行改变， map 就无能为力了，此时我们需要用 map\_err：

```
fn main() {
    let o1: Result<&str, &str> = Ok("abcde");
    let o2: Result<&str, isize> = Ok("abcde");

    let e1: Result<&str, &str> = Err("404");
    let e2: Result<&str, isize> = Err(404);

    let fn_character_count = |s: &str| -> isize { s.parse().unwrap() }; // 该函数返回一个 isize

    assert_eq!(o1.map_err(fn_character_count), o2); // Ok1 map = Ok2
    assert_eq!(e1.map_err(fn_character_count), e2); // Err1 map = Err2
}
```

通过对 o1 的操作可以看出，与 map 面对 Err 时的短小类似， map\_err 面对 Ok 时也是相当无力的。

## map\_or() 和 map\_or\_else()

map\_or 在 map 的基础上提供了一个默认值：

```

fn main() {
    const V_DEFAULT: u32 = 1;

    let s: Result<u32, ()> = Ok(10);
    let n: Option<u32> = None;
    let fn_closure = |v: u32| v + 2;

    assert_eq!(s.map_or(V_DEFAULT, fn_closure), 12);
    assert_eq!(n.map_or(V_DEFAULT, fn_closure), V_DEFAULT);
}

```

如上所示，当处理 `None` 的时候，`V_DEFAULT` 作为默认值被直接返回。

`map_or_else` 与 `map_or` 类似，但是它是通过一个闭包来提供默认值：

```

fn main() {
    let s = Some(10);
    let n: Option<i8> = None;

    let fn_closure = |v: i8| v + 2;
    let fn_default = || 1;

    assert_eq!(s.map_or_else(fn_default, fn_closure), 12);
    assert_eq!(n.map_or_else(fn_default, fn_closure), 1);

    let o = Ok(10);
    let e = Err(5);
    let fn_default_for_result = |v: i8| v + 1; // 闭包可以对 Err 中的值进行处理，并返回一个
新值

    assert_eq!(o.map_or_else(fn_default_for_result, fn_closure), 12);
    assert_eq!(e.map_or_else(fn_default_for_result, fn_closure), 6);
}

```

### ok\_or() and ok\_or\_else()

这两兄弟可以将 `Option` 类型转换为 `Result` 类型。其中 `ok_or` 接收一个默认的 `Err` 参数：

```

fn main() {
    const ERR_DEFAULT: &str = "error message";

    let s = Some("abcde");
    let n: Option<&str> = None;

    let o: Result<&str, &str> = Ok("abcde");
    let e: Result<&str, &str> = Err(ERR_DEFAULT);

    assert_eq!(s.ok_or(ERR_DEFAULT), o); // Some(T) -> Ok(T)
    assert_eq!(n.ok_or(ERR_DEFAULT), e); // None -> Err(default)
}

```

而 `ok_or_else` 接收一个闭包作为 `Err` 参数:

```

fn main() {
    let s = Some("abcde");
    let n: Option<&str> = None;
    let fn_err_message = || "error message";

    let o: Result<&str, &str> = Ok("abcde");
    let e: Result<&str, &str> = Err("error message");

    assert_eq!(s.ok_or_else(fn_err_message), o); // Some(T) -> Ok(T)
    assert_eq!(n.ok_or_else(fn_err_message), e); // None -> Err(default)
}

```

以上列出的只是常用的一部分，强烈建议大家看看标准库中有哪些可用的 API，在实际项目中，这些 API 将会非常有用: [Option](#) 和 [Result](#)。

## 自定义错误类型

虽然标准库定义了大量的错误类型，但是一个严谨的项目，光使用这些错误类型往往是不够的，例如我们可能会为暴露给用户的错误定义相应的类型。

为了帮助我们更好的定义错误，Rust 在标准库中提供了一些可复用的特征，例如 `std::error::Error` 特征:

```

use std::fmt::{Debug, Display};

pub trait Error: Debug + Display {
    fn source(&self) -> Option<&(Error + 'static)> { ... }
}

```

当自定义类型实现该特征后，该类型就可以作为 `Err` 来使用，下面一起来看看。

---

实际上，自定义错误类型只需要实现 `Debug` 和 `Display` 特征即可，`source` 方法是可选的，而 `Debug` 特征往往也无需手动实现，可以直接通过 `derive` 来派生

---

## 最简单的错误

```
use std::fmt;

// AppError 是自定义错误类型，它可以是当前包中定义的任何类型，在这里为了简化，我们使用了单元结构体作为例子。
// 为 AppError 自动派生 Debug 特征
#[derive(Debug)]
struct AppError;

// 为 AppError 实现 std::fmt::Display 特征
impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "An Error Occurred, Please Try Again!") // user-facing output
    }
}

// 一个示例函数用于产生 AppError 错误
fn produce_error() -> Result<(), AppError> {
    Err(AppError)
}

fn main(){
    match produce_error() {
        Err(e) => eprintln!("{}", e),
        _ => println!("No error"),
    }

    eprintln!("{:?}", produce_error()); // Err({ file: src/main.rs, line: 17 })
}
```

上面的例子很简单，我们定义了一个错误类型，当为它派生了 `Debug` 特征，同时手动实现了 `Display` 特征后，该错误类型就可以作为 `Err` 来使用了。

事实上，实现 `Debug` 和 `Display` 特征并不是作为 `Err` 使用的必要条件，大家可以把这两个特征实现和相应使用去除，然后看看代码会否报错。既然如此，我们为何要为自定义类型实现这两个特征呢？原因有二：

- 错误得打印输出后，才能有实际用处，而打印输出就需要实现这两个特征

- 可以将自定义错误转换成 `Box<dyn std::error::Error>` 特征对象，在后面的**归一化不同错误类型**部分，我们会详细介绍

## 更详尽的错误

上一个例子中定义的错误非常简单，我们无法从错误中得到更多的信息，现在再来定义一个具有错误码和信息的错误：

```
use std::fmt;

struct AppError {
    code: usize,
    message: String,
}

// 根据错误码显示不同的错误信息
impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        let err_msg = match self.code {
            404 => "Sorry, Can not find the Page!",
            _ => "Sorry, something is wrong! Please Try Again!",
        };

        write!(f, "{}", err_msg)
    }
}

impl fmt::Debug for AppError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(
            f,
            "AppError {{ code: {}, message: {} }}",
            self.code, self.message
        )
    }
}

fn produce_error() -> Result<(), AppError> {
    Err(AppError {
        code: 404,
        message: String::from("Page not found"),
    })
}

fn main() {
    match produce_error() {
        Err(e) => eprintln!("{}: {}", e),
        _ => println!("No error"),
    }

    eprintln!("{}: {:?}", produce_error());
    // Err(AppError { code: 404, message: Page not found })

    eprintln!("{}: {:?}", produce_error());
    // Err(
    //     AppError { code: 404, message: Page not found }
    // )
}
}
```

在本例中，我们除了增加了错误码和消息外，还手动实现了 `Debug` 特征，原因在于，我们希望能自定义 `Debug` 的输出内容，而不是使用派生后系统提供的默认输出形式。

## 错误转换 `From` 特征

标准库、三方库、本地库，各有各的精彩，各也有各的错误。那么问题就来了，我们该如何将其它的错误类型转换成自定义的错误类型？总不能神鬼牛魔，同台共舞吧。。

好在 Rust 为我们提供了 `std::convert::From` 特征：

```
pub trait From<T>: Sized {  
    fn from(_: T) -> Self;  
}
```

---

事实上，该特征在之前的 [? 操作符](#)章节中就有所介绍。

大家都使用过 `String::from` 函数吧？它可以通过 `&str` 来创建一个 `String`，其实该函数就是 `From` 特征提供的

---

下面一起来看看如何为自定义类型实现 `From` 特征：

```

use std::fs::File;
use std::io;

#[derive(Debug)]
struct AppError {
    kind: String, // 错误类型
    message: String, // 错误信息
}

// 为 AppError 实现 std::convert::From 特征，由于 From 包含在 std::prelude 中，因此可以直接
// 简化引入。
// 实现 From<io::Error> 意味着我们可以将 io::Error 错误转换成自定义的 AppError 错误
impl From<io::Error> for AppError {
    fn from(error: io::Error) -> Self {
        AppError {
            kind: String::from("io"),
            message: error.to_string(),
        }
    }
}

fn main() -> Result<(), AppError> {
    let _file = File::open("nonexistent_file.txt")?;

    Ok(())
}

// ----- 上述代码运行后输出 -----
Error: AppError { kind: "io", message: "No such file or directory (os error 2)" }

```

上面的代码中除了实现 `From` 外，还有一点特别重要，那就是 `?`  可以将错误进行隐式的强制转换：`File::open` 返回的是 `std::io::Error`，我们并没有进行任何显式的转换，它就能自动变成 `AppError`，这就是 `?`  的强大之处！

上面的例子只有一个标准库错误，再来看看多个不同的错误转换成 `AppError` 的实现：

```
use std::fs::File;
use std::io::{self, Read};
use std::num;

#[derive(Debug)]
struct AppError {
    kind: String,
    message: String,
}

impl From<io::Error> for AppError {
    fn from(error: io::Error) -> Self {
        AppError {
            kind: String::from("io"),
            message: error.to_string(),
        }
    }
}

impl From<num::ParseIntError> for AppError {
    fn from(error: num::ParseIntError) -> Self {
        AppError {
            kind: String::from("parse"),
            message: error.to_string(),
        }
    }
}

fn main() -> Result<(), AppError> {
    let mut file = File::open("hello_world.txt")?;

    let mut content = String::new();
    file.read_to_string(&mut content)?;

    let _number: usize;
    _number = content.parse()?;

    Ok(())
}

// ----- 上述代码运行后的可能输出 -----
// 01. 若 hello_world.txt 文件不存在
Error: AppError { kind: "io", message: "No such file or directory (os error 2)" }

// 02. 若用户没有相关的权限访问 hello_world.txt
Error: AppError { kind: "io", message: "Permission denied (os error 13)" }

// 03. 若 hello_world.txt 包含有非数字的内容, 例如 Hello, world!
Error: AppError { kind: "parse", message: "invalid digit found in string" }
```

## 归一化不同的错误类型

至此，关于 Rust 的错误处理大家已经了若指掌了，下面再来看看一些实战中的问题。

在实际项目中，我们往往会为不同的错误定义不同的类型，这样做非常好，但是如果你要在一个函数中返回不同的错误呢？例如：

```
use std::fs::read_to_string;

fn main() -> Result<(), std::io::Error> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String, std::io::Error> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}
```

上面的代码会报错，原因在于 `render` 函数中的两个 `?`  返回的实际上是不同的错误：`env::var()` 返回的是 `std::env::VarError`，而 `read_to_string` 返回的是 `std::io::Error`。

为了满足 `render` 函数的签名，我们就需要将 `env::VarError` 和 `io::Error` 归一化为同一种错误类型。要实现这个目的有三种方式：

- 使用特征对象 `Box<dyn Error>`
- 自定义错误类型
- 使用 `thiserror`

下面依次来看看相关的解决方式。

### `Box<dyn Error>`

大家还记得我们之前提到的 `std::error::Error` 特征吧，当时有说：自定义类型实现 `Debug + Display` 特征的主要原因就是为了能转换成 `Error` 的特征对象，而特征对象恰恰是在同一个地方使用不同类型的关键：

```
use std::fs::read_to_string;
use std::error::Error;
fn main() -> Result<(), Box<dyn Error>> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String, Box<dyn Error>> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}
```

这个方法很简单，在绝大多数场景中，性能也非常够用，但是有一个问题：`Result` 实际上不会限制错误的类型，也就是一个类型就算不实现 `Error` 特征，它依然可以在 `Result<T, E>` 中作为 `E` 来使用，此时这种特征对象的解决方案就无能为力了。

## 自定义错误类型

与特征对象相比，自定义错误类型麻烦归麻烦，但是它非常灵活，因此也不具有上面的类似限制：

```

use std::fs::read_to_string;

fn main() -> Result<(), MyError> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String, MyError> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}

#[derive(Debug)]
enum MyError {
    EnvironmentVariableNotFound,
    IOError(std::io::Error),
}

impl From<std::env::VarError> for MyError {
    fn from(_: std::env::VarError) -> Self {
        Self::EnvironmentVariableNotFound
    }
}

impl From<std::io::Error> for MyError {
    fn from(value: std::io::Error) -> Self {
        Self::IOError(value)
    }
}

impl std::error::Error for MyError {}

impl std::fmt::Display for MyError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            MyError::EnvironmentVariableNotFound => write!(f, "Environment variable not found"),
            MyError::IOError(err) => write!(f, "IO Error: {}", err.to_string()),
        }
    }
}

```

上面代码中有一行值得注意：`impl std::error::Error for MyError {}`，只有为自定义错误类型实现 `Error` 特征后，才能转换成相应的特征对象。

不得不说，真是啰嗦啊。因此在能用特征对象的时候，建议大家还是使用特征对象，无论如何，代码可读性还是很重要的！

上面的第二种方式灵活归灵活，啰嗦也是真啰嗦，好在 Rust 的社区为我们提供了 `thiserror` 解决方案，下面一起来看看该如何简化 Rust 中的错误处理。

## 简化错误处理

对于开发者而言，错误处理是代码中打交道最多的一部分之一，因此选择一把趁手的武器也很重要，它可以帮助我们节省大量的时间和精力，好钢应该用在代码逻辑而不是冗长的错误处理上。

### thiserror

`thiserror` 可以帮助我们简化上面的第二种解决方案：

```
use std::fs::read_to_string;

fn main() -> Result<(), MyError> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String, MyError> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}

#[derive(thiserror::Error, Debug)]
enum MyError {
    #[error("Environment variable not found")]
    EnvironmentVariableNotFound(#[from] std::env::VarError),
    #[error(transparent)]
    IOError(#[from] std::io::Error),
}
```

如上所示，只要简单写写注释，就可以实现错误处理了，惊不惊喜？

### error-chain

`error-chain` 也是简单好用的库，可惜不再维护了，但是我觉得它依然可以在合适的地方大放光彩，值得大家去了解下。

```
use std::fs::read_to_string;

error_chain::error_chain! {
    foreign_links {
        EnvironmentVariableNotFound(::std::env::VarError);
        IOError(::std::io::Error);
    }
}

fn main() -> Result<()> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}
```

喏，简单吧？使用 `error-chain` 的宏你可以获得：`Error` 结构体，错误类型 `ErrorKind` 枚举以及一个自定义的 `Result` 类型。

## anyhow

`anyhow` 和 `thiserror` 是同一个作者开发的，这里是作者关于 `anyhow` 和 `thiserror` 的原话：

---

如果你想要设计自己的错误类型，同时给调用者提供具体的信息时，就使用 `thiserror`，例如当你在开发一个三方库代码时。如果你只想要简单，就使用 `anyhow`，例如在自己的应用服务中。

---

```
use std::fs::read_to_string;

use anyhow::Result;

fn main() -> Result<()> {
    let html = render()?;
    println!("{}", html);
    Ok(())
}

fn render() -> Result<String> {
    let file = std::env::var("MARKDOWN")?;
    let source = read_to_string(file)?;
    Ok(source)
}
```

关于如何选用 `thiserror` 和 `anyhow` 只需要遵循一个原则即可：**是否关注自定义错误消息**，关注则使用 `thiserror`（常见业务代码），否则使用 `anyhow`（编写第三方库代码）。

## 总结

Rust 一个为人津津乐道的点就是强大、易用的错误处理，对于新手来说，这个机制可能会有些复杂，但是一旦体会到了其中的好处，你将跟我一样沉醉其中不能自拔。

# unsafe 简介

圣人论迹不论心，论心世上无圣人，对于编程语言而言，亦是如此。

虽然在本章之前，我们学到的代码都是在编译期就得到了 Rust 的安全保障，但是在其内心深处也隐藏了一些阴暗面，在这些阴暗面里，内存安全就存在一些变数了：当不娴熟的开发者接触到这些阴暗面，就可能写出不安全的代码，因此我们称这种代码为 `unsafe` 代码块。

## 为何会有 `unsafe`

几乎每个语言都有 `unsafe` 关键字，但 Rust 语言使用 `unsafe` 的原因可能与其它编程语言还有所不同。

### 过强的编译器

说来尴尬，`unsafe` 的存在主要是因为 Rust 的静态检查太强了，但是强就算了，它还很保守，这就会导致当编译器在分析代码时，一些正确代码会因为编译器无法分析出它的所有正确性，结果将这段代码拒绝，导致编译错误。

这种保守的选择确实也没有错，毕竟安全就是要防微杜渐，但是对于使用者来说，就不是那么愉快的事了，特别是当配合 Rust 的所有权系统一起使用时，有个别问题是真的棘手和难以解决。

举个例子，在之前的自引用章节中，我们就提到了相关的编译检查是很难绕过的，如果想要绕过，最常用的方法之一就是使用 [unsafe 和 Pin](#)。

好在，当遇到这些情况时，我们可以使用 `unsafe` 来解决。此时，你需要替代编译器的部分职责对 `unsafe` 代码的正确性负责，例如在正常代码中不可能遇到的空指针解引用问题在 `unsafe` 中就可能会遇到，我们需要自己来处理好这些类似的问题。

### 特定任务的需要

至于 `unsafe` 存在的另一个原因就是：它必须要存在。原因是计算机底层的一些硬件就是不安全的，如果 Rust 只允许你做安全的操作，那一些任务就无法完成，换句话说，我们还怎么跟 C++ 干架？

Rust 的一个主要定位就是系统编程，众所周知，系统编程就是底层编程，往往需要直接跟操作系统打交道，甚至于去实现一个操作系统。而为了实现底层系统编程，`unsafe` 就是必不可少的。

在了解了为何会有 `unsafe` 后，我们再来看看，除了这些必要性，`unsafe` 还能给我们带来哪些超能力。

## unsafe 的超能力

使用 `unsafe` 非常简单，只需要将对应的代码块标记下即可：

```
fn main() {
    let mut num = 5;

    let r1 = &num as *const i32;

    unsafe {
        println!("r1 is: {}", *r1);
    }
}
```

上面代码中，`r1` 是一个裸指针(raw pointer)，由于它具有破坏 Rust 内存安全的潜力，因此只能在 `unsafe` 代码块中使用，如果你去掉 `unsafe {}`，编译器会立刻报错。

言归正传，`unsafe` 能赋予我们 5 种超能力，这些能力在安全的 Rust 代码中是无法获取的：

- 解引用裸指针，就如上例所示
- 调用一个 `unsafe` 或外部的函数
- 访问或修改一个可变的静态变量
- 实现一个 `unsafe` 特征
- 访问 `union` 中的字段

在本章中，我们将着重讲解裸指针和 FFI 的使用。

## unsafe 的安全保证

曾经在 `reddit` 上有一个讨论还挺热闹的，是关于 `unsafe` 的命名是否合适，总之公有公理，婆有婆理，但有一点是不可否认的：虽然名称自带不安全，但是 Rust 依然提供了强大的安全支撑。

首先，`unsafe` 并不能绕过 Rust 的借用检查，也不能关闭任何 Rust 的安全检查规则，例如当你在 `unsafe` 中使用引用时，该有的检查一样都不会少。

因此 `unsafe` 能给大家提供的也仅仅是之前的 5 种超能力，在使用这 5 种能力时，编译器才不会进行内存安全方面的检查，最典型的就是使用裸指针(引用和裸指针有很大的区别)。

## 谈虎色变？

在网上充斥着这样的言论：千万不要使用 `unsafe`，因为它不安全，甚至有些库会以没有 `unsafe` 代码作为噱头来吸引用户。事实上，大可不必，如果按照这个标准，Rust 的标准库也将不复存在！

Rust 中的 `unsafe` 其实没有那么可怕，虽然听上去很不安全，但是实际上 Rust 依然提供了很多机制来帮我们提升了安全性，因此不必像对待 Go 语言的 `unsafe` 那样去畏惧于使用 Rust 中的 `unsafe`。

大致使用原则总结如下：没必要用时，就不要用，当有必要用时，就大胆用，但是尽量控制好边界，让 `unsafe` 的范围尽可能小。

## 控制 `unsafe` 的使用边界

`unsafe` 不安全，但是该用的时候就要用，在一些时候，它能帮助我们大幅降低代码实现的成本。

而作为使用者，你的水平决定了 `unsafe` 到底有多不安全，因此你需要在 `unsafe` 中小心谨慎地去访问内存。

即使做到小心谨慎，依然会有出错的可能性，但是 `unsafe` 语句块决定了：就算内存访问出错了，你也能立刻意识到，错误是在 `unsafe` 代码块中，而不花大量时间像无头苍蝇一样去寻找问题所在。

正因为此，写代码时要尽量控制好 `unsafe` 的边界大小，越小的 `unsafe` 越会让我们在未来感谢自己当初的选择。

除了控制边界大小，另一个很常用的方式就是在 `unsafe` 代码块外包裹一层 `safe` 的 API，例如一个函数声明为 `safe` 的，然后在其内部有一块儿是 `unsafe` 代码。

---

忍不住抱怨一句，内存安全方面的 bug，是真心难查！

---

# 五种兵器

古龙有一部小说，名为“七种兵器”，其中每一种都精妙绝伦，令人闻风丧胆，而 `unsafe` 也有五种兵器，它们可以让你拥有其它代码无法实现的能力，同时它们也像七种兵器一样令人闻风丧胆，下面一起来看看庐山真面目。

## 解引用裸指针

裸指针(raw pointer，又称原生指针)在功能上跟引用类似，同时它也需要显式地注明可变性。但是又和引用有所不同，裸指针长这样：`*const T` 和 `*mut T`，它们分别代表了不可变和可变。

大家在之前学过 `*` 操作符，知道它可以用于解引用，但是在裸指针 `*const T` 中，这里的 `*` 只是类型名称的一部分，并没有解引用的含义。

至此，我们已经学过三种类似指针的概念：引用、智能指针和裸指针。与前两者不同，裸指针：

- 可以绕过 Rust 的借用规则，可以同时拥有一个数据的可变、不可变指针，甚至还能拥有多个可变的指针
- 并不能保证指向合法的内存
- 可以是 `null`
- 没有实现任何自动的回收 (drop)

总之，裸指针跟 C 指针是非常像的，使用它需要以牺牲安全性为前提，但我们获得了更好的性能，也可以跟其它语言或硬件打交道。

### 基于引用创建裸指针

下面的代码**基于值的引用**同时创建了可变和不可变的裸指针：

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

`as` 可以用于强制类型转换，在之前章节中有讲解。在这里，我们将引用 `&num` / `&mut num` 强转为相应的裸指针 `*const i32` / `*mut i32`。

细心的同学可能会发现，在这段代码中并没有 `unsafe` 的身影，原因在于：**创建裸指针是安全的行为，而解引用裸指针才是不安全的行为**：

```
fn main() {
    let mut num = 5;

    let r1 = &num as *const i32;

    unsafe {
        println!("r1 is: {}", *r1);
    }
}
```

## 基于内存地址创建裸指针

在上面例子中，我们基于现有的引用来创建裸指针，这种行为是很安全的。但是接下来的方式就不安全了：

```
let address = 0x012345usize;
let r = address as *const i32;
```

这里基于一个内存地址来创建裸指针，可以想像，这种行为是相当危险的。试图使用任意的内存地址往往是一种未定义的行为(undefined behavior)，因为该内存地址有可能存在值，也有可能没有，就算有值，也大概率不是你需要的值。

同时编译器也有可能会优化这段代码，会造成没有任何内存访问发生，甚至程序还可能发生段错误(segmentation fault)。**总之，你几乎没有好的理由像上面这样实现代码，虽然它是可行的。**

如果真的要使用内存地址，也是类似下面的用法，先取地址，再使用，而不是凭空捏造一个地址：

```

use std::slice::from_raw_parts, str::from_utf8_unchecked;

// 获取字符串的内存地址和长度
fn get_memory_location() -> (usize, usize) {
    let string = "Hello World!";
    let pointer = string.as_ptr() as usize;
    let length = string.len();
    (pointer, length)
}

// 在指定的内存地址读取字符串
fn get_str_at_location(pointer: usize, length: usize) -> &'static str {
    unsafe { from_utf8_unchecked(from_raw_parts(pointer as *const u8, length)) }
}

fn main() {
    let (pointer, length) = get_memory_location();
    let message = get_str_at_location(pointer, length);
    println!(
        "The {} bytes at 0x{:X} stored: {}",
        length, pointer, message
    );
    // 如果大家想知道为何处理裸指针需要 `unsafe`，可以试着反注释以下代码
    // let message = get_str_at_location(1000, 10);
}

```

以上代码同时还演示了访问非法内存地址会发生什么，大家可以试着去反注释这段代码试试。

## 使用 \* 解引用

```

let a = 1;
let b: *const i32 = &a as *const i32;
let c: *const i32 = &a;
unsafe {
    println!("{}", *c);
}

```

使用 `*` 可以对裸指针进行解引用，由于该指针的内存安全性并没有任何保证，因此我们需要使用 `unsafe` 来包裹解引用的逻辑(切记，`unsafe` 语句块的范围一定要尽可能的小，具体原因在上一章节有讲)。

以上代码另一个值得注意的点就是：除了使用 `as` 来显式的转换，我们还使用了隐式的转换方式 `let c: *const i32 = &a;`。在实际使用中，我们建议使用 `as` 来转换，因为这种显式的方式更有助于提醒用户：你在使用的指针是裸指针，需要小心。

## 基于智能指针创建裸指针

还有一种创建裸指针的方式，那就是基于智能指针来创建：

```
let a: Box<i32> = Box::new(10);
// 需要先解引用a
let b: *const i32 = &a;
// 使用 into_raw 来创建
let c: *const i32 = Box::into_raw(a);
```

## 小结

像之前代码演示的那样，使用裸指针可以让我们创建两个可变指针都指向同一个数据，如果使用安全的 Rust，你是无法做到这一点的，违背了借用规则，编译器会对我们进行无情的阻止。因此裸指针可以绕过借用规则，但是由此带来的数据竞争问题，就需要大家自己来处理了，总之，需要小心！

既然这么危险，为何还要使用裸指针？除了之前提到的性能等原因，还有一个重要用途就是跟 c 语言的代码进行交互( FFI )，在讲解 FFI 之前，先来看看如何调用 unsafe 函数或方法。

## 调用 unsafe 函数或方法

unsafe 函数从外表上来看跟普通函数并无区别，唯一的区别就是它需要使用 unsafe fn 来进行定义。这种定义方式是为了告诉调用者：当调用此函数时，你需要注意它的相关需求，因为 Rust 无法担保调用者在使用该函数时能满足它所需的一切需求。

强制调用者加上 unsafe 语句块，就可以让他清晰的认识到，正在调用一个不安全的函数，需要小心看文档，看看函数有哪些特别的要求需要被满足。

```
unsafe fn dangerous() {}
fn main() {
    dangerous();
}
```

如果试图像上面这样调用，编译器就会报错：

```
error[E0133]: call to unsafe function is unsafe and requires unsafe function or block
--> src/main.rs:3:5
|
3 |     dangerous();
|     ^^^^^^^^^^^^ call to unsafe function
```

按照报错提示，加上 `unsafe` 语句块后，就能顺利执行了：

```
unsafe {
    dangerous();
}
```

道理很简单，但一定要牢记在心：使用 `unsafe` 声明的函数时，一定要看看相关的文档，确定自己没有遗漏什么。

还有，`unsafe` 无需俄罗斯套娃，在 `unsafe` 函数体中使用 `unsafe` 语句块是多余的行为。

## 用安全抽象包裹 `unsafe` 代码

一个函数包含了 `unsafe` 代码不代表我们需要将整个函数都定义为 `unsafe fn`。事实上，在标准库中有大量的安全函数，它们内部都包含了 `unsafe` 代码块，下面我们一起来看看一个很好用的标准库函数：`split_at_mut`。

大家可以想象一下这个场景：需要将一个数组分成两个切片，且每一个切片都要求是可变的。类似需求在安全 Rust 中是很难实现的，因为要对同一个数组做两个可变借用：

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid], &mut slice[mid..])
}

fn main() {
    let mut v = vec![1, 2, 3, 4, 5, 6];

    let r = &mut v[..];

    let (a, b) = split_at_mut(r, 3);

    assert_eq!(a, &mut [1, 2, 3]);
    assert_eq!(b, &mut [4, 5, 6]);
}
```

上面代码一眼看过去就知道会报错，因为我们试图在自定义的 `split_at_mut` 函数中，可变借用 `slice` 两次：

```

error[E0499]: cannot borrow `*slice` as mutable more than once at a time
--> src/main.rs:6:30
|
1 | fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
|   - let's call the lifetime of this reference ''1``
...
6 |     (&mut slice[..mid], &mut slice[mid..])
|     ^^^^^^
|     |           |
|     |           second mutable borrow occurs here
|     |           first mutable borrow occurs here
|     returning this value requires that `*slice` is borrowed for ''1``

```

对于 Rust 的借用检查器来说，它无法理解我们是分别借用了同一个切片的两个不同部分，但事实上，这种行为是没有任何问题的，毕竟两个借用没有任何重叠之处。总之，不太聪明的 Rust 编译器阻碍了我们用这种简单且安全的方式去实现，那只能剑走偏锋，试试 `unsafe` 了。

```

use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}

fn main() {
    let mut v = vec![1, 2, 3, 4, 5, 6];

    let r = &mut v[..];

    let (a, b) = split_at_mut(r, 3);

    assert_eq!(a, &mut [1, 2, 3]);
    assert_eq!(b, &mut [4, 5, 6]);
}

```

相比安全实现，这段代码就显得没那么好理解了，甚至于我们还需要像 C 语言那样，通过指针地址的偏移去控制数组的分割。

- `as_mut_ptr` 会返回指向 `slice` 首地址的裸指针 `*mut i32`

- `slice::from_raw_parts_mut` 函数通过指针和长度来创建一个新的切片，简单来说，该切片的初始地址是 `ptr`，长度为 `mid`
- `ptr.add(mid)` 可以获取第二个切片的初始地址，由于切片中的元素是 `i32` 类型，每个元素都占用了 4 个字节的内存大小，因此我们不能简单的用 `ptr + mid` 来作为初始地址，而应该使用 `ptr + 4 * mid`，但是这种使用方式并不安全，因此 `.add` 方法是最佳选择

由于 `slice::from_raw_parts_mut` 使用裸指针作为参数，因此它是一个 `unsafe fn`，我们在使用它时，就必须用 `unsafe` 语句块进行包裹，类似的，`.add` 方法也是如此(还是那句话，不要将无关的代码包含在 `unsafe` 语句块中)。

部分同学可能会有疑问，那这段代码我们怎么保证 `unsafe` 中使用的裸指针 `ptr` 和 `ptr.add(mid)` 是合法的呢？秘诀就在于 `assert!(mid <= len);`，通过这个断言，我们保证了裸指针一定指向了 `slice` 切片中的某个元素，而不是一个莫名其妙的内存地址。

再回到我们的主题：虽然 `split_at_mut` 使用了 `unsafe`，但我们无需将其声明为 `unsafe fn`，这种情况下就是使用安全的抽象包裹 `unsafe` 代码，这里的 `unsafe` 使用是非常安全的，因为我们从合法数据中创建了的合法指针。

与之对比，下面的代码就非常危险了：

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let slice: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
println!("{:?}", slice);
```

这段代码从一个任意的内存地址，创建了一个 10000 长度的 `i32` 切片，我们无法保证切片中的元素都是合法的 `i32` 值，这种访问就是一种未定义行为(UB = undefined behavior)。

```
zsh: segmentation fault
```

不出所料，运行后看到了一个段错误。

## FFI

FFI (Foreign Function Interface) 可以用来与其它语言进行交互，但是并不是所有语言都这么称呼，例如 Java 称之为 JNI (Java Native Interface)。

`FFI`之所以存在是由于现实中很多代码库都是由不同语言编写的，如果我们需要使用某个库，但是它是由其它语言编写的，那么往往只有两个选择：

- 对该库进行重写或者移植
- 使用 `FFI`

前者相当不错，但是在很多时候，并没有那么多时间去重写，因此 `FFI` 就成了最佳选择。回到 Rust 语言上，由于这门语言依然很年轻，一些生态是缺失的，我们在写一些不是那么大众的项目时，可能会同时遇到没有相应的 Rust 库可用的尴尬境况，此时通过 `FFI` 去调用 C 语言的库就成了相当棒的选择。

还有在将 C/C++ 的代码重构为 Rust 时，先将相关代码引入到 Rust 项目中，然后逐步重构，也是不错的（为什么用不错来形容？因为重构一个有一定规模的 C/C++ 项目远没有想象中美好，因此最好的选择还是对于新项目使用 Rust 实现，老项目。。。就让它先运行着吧）。

当然，除了 `FFI` 还有一个办法可以解决跨语言调用的问题，那就是将其作为一个独立的服务，然后使用网络调用的方式去访问，HTTP，gRPC 都可以。

言归正传，之前我们提到 `unsafe` 的另一个重要目的就是对 `FFI` 提供支持，它的全称是 `Foreign Function Interface`，顾名思义，通过 `FFI`，我们的 Rust 代码可以跟其它语言的外部代码进行交互。

下面的例子演示了如何调用 C 标准库中的 `abs` 函数：

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

C 语言的代码定义在了 `extern` 代码块中，而 `extern` 必须使用 `unsafe` 才能进行调用，原因在于其它语言的代码并不会强制执行 Rust 的规则，因此 Rust 无法对这些代码进行检查，最终还是要靠开发者自己来保证代码的正确性和程序的安全性。

## ABI

在 `extern "C"` 代码块中，我们列出了想要调用的外部函数的签名。其中 “C” 定义了外部函数所使用的 **应用二进制接口 ABI** (Application Binary Interface)：ABI 定义了如何在汇编层面来调用该函数。在所有 `ABI` 中，C 语言的是最常见的。

## 在其它语言中调用 Rust 函数

在 Rust 中调用其它语言的函数是让 Rust 利用其他语言的生态，那反过来可以吗？其他语言可以利用 Rust 的生态不？答案是肯定的。

我们可以使用 `extern` 来创建一个接口，其它语言可以通过该接口来调用相关的 Rust 函数。但是此处的语法与之前有所不同，之前用的是语句块，而这里是在函数定义时加上 `extern` 关键字，当然，别忘了指定相应的 ABI：

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

上面的代码可以让 `call_from_c` 函数被 C 语言的代码调用，当然，前提是将其编译成一个共享库，然后链接到 C 语言中。

这里还有一个比较奇怪的注解 `#[no_mangle]`，它用于告诉 Rust 编译器：不要乱改函数的名称。

Mangling 的定义是：当 Rust 因为编译需要去修改函数的名称，例如为了让名称包含更多的信息，这样其它的编译部分就能从该名称获取相应的信息，这种修改会导致函数名变得相当不可读。

因此，为了让 Rust 函数能顺利被其它语言调用，我们必须要禁止掉该功能。

## 访问或修改一个可变的静态变量

这部分我们在之前的[全局变量章节](#)中有过详细介绍，这里就不再赘述，大家可以前往此章节阅读。

## 实现 `unsafe` 特征

说实话，`unsafe` 的特征确实不多见，如果大家还记得的话，我们在之前的[Send 和 Sync 章节](#)中实现过 `unsafe` 特征 `Send`。

之所以会有 `unsafe` 的特征，是因为该特征至少有一个方法包含有编译器无法验证的内容。`unsafe` 特征的声明很简单：

```
unsafe trait Foo {
    // 方法列表
}

unsafe impl Foo for i32 {
    // 实现相应的方法
}

fn main() {}
```

通过 `unsafe impl` 的使用，我们告诉编译器：相应的正确性由我们自己来保证。

再回到刚提到的 `Send` 特征，若我们的类型中的所有字段都实现了 `Send` 特征，那该类型也会自动实现 `Send`。但是如果我们想要为某个类型手动实现 `Send`，例如为裸指针，那么就必须使用 `unsafe`，相关的代码在之前的链接中也有，大家可以移步查看。

总之，`Send` 特征标记为 `unsafe` 是因为 Rust 无法验证我们的类型是否能在线程间安全的传递，因此就需要通过 `unsafe` 来告诉编译器，它无需操心，剩下的交给我们自己来处理。

## 访问 union 中的字段

截止目前，我们还没有介绍过 `union`，原因很简单，它主要用于跟 c 代码进行交互。

访问 `union` 的字段是不安全的，因为 Rust 无法保证当前存储在 `union` 实例中的数据类型。

```
#[repr(C)]
union MyUnion {
    f1: u32,
    f2: f32,
}
```

上从可以看出，`union` 的使用方式跟结构体确实很相似，但是前者的所有字段都共享同一个存储空间，意味着往 `union` 的某个字段写入值，会导致其它字段的值会被覆盖。

关于 `union` 的更多信息，可以在[这里查看](#)。

## 一些实用工具(库)

由于 `unsafe` 和 `FFI` 在 Rust 的使用场景中是相当常见的(例如相对于 Go 的 `unsafe` 来说)，因此社区已经开发出了相当一部分实用的工具，可以改善相应的开发体验。

## rust-bindgen 和 cbindgen

对于 FFI 调用来说，保证接口的正确性是非常重要的，这两个库可以帮我们自动生成相应的接口，其中 [rust-bindgen](#) 用于在 Rust 中访问 C 代码，而 [cbindgen](#) 则反之。

下面以 `rust-bindgen` 为例，来看看如何自动生成调用 C 的代码，首先下面是 C 代码：

```
typedef struct Doggo {
    int many;
    char wow;
} Doggo;

void eleven_out_of_ten_majestic_af(Doggo* pupper);
```

下面是自动生成的可以调用上面代码的 Rust 代码：

```
/* automatically generated by rust-bindgen 0.99.9 */

#[repr(C)]
pub struct Doggo {
    pub many: ::std::os::raw::c_int,
    pub wow: ::std::os::raw::c_char,
}

extern "C" {
    pub fn eleven_out_of_ten_majestic_af(pupper: *mut Doggo);
}
```

## cxx

如果需要跟 C++ 代码交互，非常推荐使用 `cxx`，它提供了双向的调用，最大的优点就是安全：是的，你无需通过 `unsafe` 来使用它！

## Miri

`miri` 可以生成 Rust 的中间层表示 MIR，对于编译器来说，我们的 Rust 代码首先会被编译为 MIR，然后再提交给 LLVM 进行处理。

可以通过 `rustup component add miri` 来安装它，并通过 `cargo miri` 来使用，同时还可以使用 `cargo miri test` 来运行测试代码。

`miri` 可以帮助我们检查常见的未定义行为(UB = Undefined Behavior)，以下列出了一部分：

- 内存越界检查和内存释放后再使用(use-after-free)

- 使用未初始化的数据
- 数据竞争
- 内存对齐问题

但是需要注意的是，它只能帮助识别被执行代码路径的风险，那些未被执行到的代码是没办法被识别的。

## Clippy

官方的 `clippy` 检查器提供了有限的 `unsafe` 支持，虽然不多，但是至少有一定帮助。例如 `missing_safety_docs` 检查可以帮助我们检查哪些 `unsafe` 函数遗漏了文档。

需要注意的是：Rust 编译器并不会默认开启所有检查，大家可以调用 `rustc -W help` 来看看最新的信息。

## Prusti

`prusti` 需要大家自己来构建一个证明，然后通过它证明代码中的不变量是正确被使用的，当你在安全代码中使用不安全的不变量时，就会非常有用。具体的使用文档见[这里](#)。

## 模糊测试(fuzz testing)

在 [Rust Fuzz Book](#) 中列出了一些 Rust 可以使用的模糊测试方法。

同时，我们还可以使用 `rutenspitz` 这个过程宏来测试有状态的代码，例如数据结构。

## 总结

至此，`unsafe` 的五种兵器已介绍完毕，大家是否意犹未尽？我想说的是，就算意犹未尽，也没有其它兵器了。

就像上一章中所提到的，`unsafe` 只应该用于这五种场景，其它场景，你应该坚决的使用安全的代码，否则就会像 `actix-web` 的前作者一样，被很多人议论，甚至被喷。。。

总之，能不使用 `unsafe` 一定不要使用，就算使用也要控制好边界，让范围尽可能的小，就像本章的例子一样，只有真的需要 `unsafe` 的代码，才应该包含其中，而不是将无关代码也纳入进来。

## 进一步学习

1. [Unsafe Rust: How and when \(not\) to use it](#)

# 内联汇编

---

本章内容对于学习 Rust 不是必须的，而且难度很高，大家简单知道有这回事就好，不必非要学学会  
:D

---

Rust 提供了 `asm!` 宏，可以让大家在 Rust 代码中嵌入汇编代码，对于一些极致高性能或者底层的场景还是非常有用的，例如操作系统内核开发。但通常来说，大家并不应该在自己的项目中使用到该项技术，它为极客而生！

本章的例子是基于 `x86/x86-64` 汇编的，但是其它架构也是支持的，目前支持的包括：

- x86 和 x86-64
- ARM
- AArch64
- RISC-V

当使用在不支持的平台上时，编译器会给出报错。

## 基本用法

先从一个简单例子开始：

```
use std::arch::asm;

unsafe {
    asm!("nop");
}
```

注意 `unsafe` 语句块依然是必不可少的，因为可能在里面插入危险的指令，最终破坏代码的安全性。

上面代码将插入一个 `NOP` 指令( 空操作 ) 到编译器生成的汇编代码中，其中指令作为 `asm!` 的第一个参数传入。

## 输入和输出

上面的代码有够无聊的，来点实际的：

```
use std::arch::asm;

let x: u64;
unsafe {
    asm!("mov {}, 5", out(reg) x);
}
assert_eq!(x, 5);
```

这段代码将 5 赋给 u64 类型的变量 x，值得注意的是 `asm!` 的指令参数实际上是一个格式化字符串。至于传给格式化字符串的参数，看起来还是比较陌生的：

- 首先，需要说明目标变量是作为内联汇编的输入还是输出，在本例中，是一个输出 `out`
- 最后，要指定变量将要使用的寄存器，本例中使用通用寄存器 `reg`，编译器会自动选择合适的寄存器

```
use std::arch::asm;

let i: u64 = 3;
let o: u64;
unsafe {
    asm!(
        "mov {0}, {1}",
        "add {0}, 5",
        out(reg) o,
        in(reg) i,
    );
}
assert_eq!(o, 8);
```

上面的代码中进一步使用了输入 `in`，将 5 加到输入的变量 `i` 上，然后将结果写到输出变量 `o`。实际的操作方式是首先将 `i` 的值拷贝到输出，然后再加上 5。

上例还能看出几点：

- `asm!` 允许使用多个格式化字符串，每一个作为单独一行汇编代码存在，看起来跟阅读真实的汇编代码类似
- 输入变量通过 `in` 来声明
- 和以前见过的格式化字符串一样，可以使用多个参数，通过 `{0}, {1}` 来指定，这种方式特别有用，毕竟在代码中，变量是经常复用的，而这种参数的指定方式刚好可以复用

事实上，还可以进一步优化代码，去掉 `mov` 指令：

```
use std::arch::asm;

let mut x: u64 = 3;
unsafe {
    asm!("add {0}, 5", inout(reg) x);
}
assert_eq!(x, 8);
```

又多出一个 `inout` 关键字，但是不难猜，它说明 `x` 即是输入又是输出。与之前的分离方式还有一点很大的区别，这种方式可以保证使用同一个寄存器来完成任务。

当然，你可以在使用 `inout` 的情况下，指定不同的输入和输出：

```
use std::arch::asm;

let x: u64 = 3;
let y: u64;
unsafe {
    asm!("add {0}, 5", inout(reg) x => y);
}
assert_eq!(y, 8);
```

## 延迟输出操作数

Rust 编译器对于操作数分配是较为保守的，它会假设 `out` 可以在任何时间被写入，因此 `out` 不会跟其它参数共享它的位置。然而为了保证最佳性能，使用尽量少的寄存器是有必要的，这样它们不必在内联汇编的代码块内保存和重加载。

为了达成这个目标( 共享位置或者说寄存器，以实现减少寄存器使用的性能优化 )，Rust 提供一个 `lateout` 关键字，可以用于任何只在所有输入被消费后才被写入的输出，与之类似还有一个 `inlateout`。

但是 `inlateout` 在某些场景中是无法使用的，例如下面的例子：

```
use std::arch::asm;

let mut a: u64 = 4;
let b: u64 = 4;
let c: u64 = 4;
unsafe {
    asm!(
        "add {0}, {1}",
        "add {0}, {2}",
        inout(reg) a,
        in(reg) b,
        in(reg) c,
    );
}
assert_eq!(a, 12);
```

一旦用了 `inlateout` 后，上面的代码就只能运行在 `Debug` 模式下，原因是 `Debug` 并没有做任何优化，但是 `release` 发布不同，为了性能是要做很多编译优化的。

在编译优化时，编译器可以很容易的为输入 `b` 和 `c` 分配同样的寄存器，因为它知道它们有同样的值。如果这里使用 `inlateout`，那么 `a` 和 `c` 就可以被分配到相同的寄存器，在这种情况下，第一条指令将覆盖掉 `c` 的值，最终导致汇编代码产生错误的结果。

因此这里使用 `inout`，那么编译器就会为 `a` 分配一个独立的寄存器。

但是下面的代码又不同，它是可以使用 `inlateout` 的：

```
use std::arch::asm;

let mut a: u64 = 4;
let b: u64 = 4;
unsafe {
    asm!("add {0}, {1}", inlateout(reg) a, in(reg) b);
}
assert_eq!(a, 8);
```

原因在于输出只有在所有寄存器都被读取后，才被修改。因此，即使 `a` 和 `b` 被分配了同样的寄存器，代码也会正常工作，不存在之前的覆盖问题。

## 显式指定寄存器

一些指令会要求操作数只能存在特定的寄存器中，因此 Rust 的内联汇编提供了一些限制操作符。

大家应该记得之前出现过的 `reg` 是适用于任何架构的通用寄存器，意味着编译器可以自己选择合适的寄存器，但是当你需要显式地指定寄存器时，很可能会变成平台相关的代码，适用移植性会差很多。例如 `x86` 下的寄存器：`eax`, `ebx`, `ecx`, `ebp`, `esi` 等等。

```
use std::arch::asm;

let cmd = 0xd1;
unsafe {
    asm!("out 0x64, eax", in("eax") cmd);
}
```

上面的例子调用 `out` 指令将 `cmd` 变量的值输出到 `0x64` 内存地址中。由于 `out` 指令只接收 `eax` 和它的子寄存器，因此我们需要使用 `eax` 来指定特定的寄存器。

---

显式寄存器操作数无法用于格式化字符串中，例如我们之前使用的 `{}`，只能直接在字符串中使用 `eax`。同时，该操作数只能出现在最后，也就是在其它操作数后面出现

```
use std::arch::asm;

fn mul(a: u64, b: u64) -> u128 {
    let lo: u64;
    let hi: u64;

    unsafe {
        asm!(
            // The x86 mul instruction takes rax as an implicit input and writes
            // the 128-bit result of the multiplication to rax:rdx.
            "mul {},",
            in(reg) a,
            inlateout("rax") b => lo,
            lateout("rdx") hi
        );
    }

    ((hi as u128) << 64) + lo as u128
}
```

这段代码使用了 `mul` 指令，将两个 64 位的输入相乘，生成一个 128 位的结果。

首先将变量 `a` 的值存到寄存器 `reg` 中，其次显式使用寄存器 `rax`，它的值来源于变量 `b`。结果的低 64 位存储在 `rax` 中，然后赋给变量 `lo`，而结果的高 64 位则存在 `rdx` 中，最后赋给 `hi`。

## Clobbered 寄存器

在很多情况下，无需作为输出的状态都会被内联汇编修改，这个状态被称为 "clobbered"。

我们需要告诉编译器相关的情况，因为编译器需要在内联汇编语句块的附近存储和恢复这种状态。

```
use std::arch::asm;

fn main() {
    // three entries of four bytes each
    let mut name_buf = [0_u8; 12];
    // String is stored as ascii in ebx, edx, ecx in order
    // Because ebx is reserved, the asm needs to preserve the value of it.
    // So we push and pop it around the main asm.
    // (in 64 bit mode for 64 bit processors, 32 bit processors would use ebx)

    unsafe {
        asm!(
            "push rbx",
            "cpuid",
            "mov [rdi], ebx",
            "mov [rdi + 4], edx",
            "mov [rdi + 8], ecx",
            "pop rbx",
            // We use a pointer to an array for storing the values to simplify
            // the Rust code at the cost of a couple more asm instructions
            // This is more explicit with how the asm works however, as opposed
            // to explicit register outputs such as `out("ecx") val`
            // The *pointer itself* is only an input even though it's written behind
            in("rdi") name_buf.as_mut_ptr(),
            // select cpuid 0, also specify eax as clobbered
            inout("eax") 0 => _,
            // cpuid clobbers these registers too
            out("ecx") _,
            out("edx") _
        );
    }

    let name = core::str::from_utf8(&name_buf).unwrap();
    println!("CPU Manufacturer ID: {}", name);
}
```

例子中，我们使用 `cpuid` 指令来读取 CPU ID，该指令会将值写入到 `eax`、`edx` 和 `ecx` 中。

即使 `eax` 从没有被读取，我们依然需要告知编译器这个寄存器被修改过，这样编译器就可以在汇编代码之前存储寄存器中的值。这个需要通过将输出声明为 `_` 而不是一个具体的变量名，代表着该输出值被丢弃。

这段代码也会绕过一个限制：`ebx` 是一个 LLVM 保留寄存器，意味着 LLVM 会假设它拥有寄存器的全部控制权，并在汇编代码块结束时将寄存器的状态恢复到最开始的状态。由于这个限制，该寄存器无法被用于输入或者输出，除非编译器使用该寄存器的满足一个通用寄存器的需求（例如 `in(reg)`）。但这样使用后，`reg` 操作数就在使用保留寄存器时变得危险起来，原因是可能会无意识的破坏输入或者输出，毕竟它们共享同一个寄存器。

为了解决这个问题，我们使用 `rdi` 来存储指向输出数组的指针，通过 `push` 的方式存储 `ebx`：在汇编代码块的内部读取 `ebx` 的值，然后写入到输出数组。后面再可以通过 `pop` 的方式来回复 `ebx` 到初始的状态。

`push` 和 `pop` 使用完成的 64 位 `rbx` 寄存器，来确保整个寄存器的内容都被保存。如果是在 32 位机器上，代码将使用 `ebx` 替代。

还还可以在汇编代码内部使用通用寄存器：

```
use std::arch::asm;

// Multiply x by 6 using shifts and adds
let mut x: u64 = 4;
unsafe {
    asm!(
        "mov {tmp}, {x}",
        "shl {tmp}, 1",
        "shl {x}, 2",
        "add {x}, {tmp}",
        x = inout(reg) x,
        tmp = out(reg) _,
    );
}
assert_eq!(x, 4 * 6);
```

## 总结

由于这块儿内容过于专业，本书毕竟是通用的 Rust 学习书籍，因此关于内联汇编就不再赘述。事实上，如果你要真的写出可用的汇编代码，要学习的还很多…

感兴趣的同学可以看看如下英文资料：[Rust Reference](#) 和 [Rust By Example](#)。

# Macro 宏编程

在编程世界可以说是谈“宏”色变，原因在于 C 语言中的宏是非常危险的东东，但并不是所有语言都像 C 这样，例如对于古老的语言 Lisp 来说，宏就是就是一个非常强大的好帮手。

那话说回来，在 Rust 中宏到底是好是坏呢？本章将带你揭开它的神秘面纱。

事实上，我们虽然没有见过宏，但是已经多次用过它，例如在全书的第一个例子中就用到了：`println!`（“你好，世界”），这里 `println!` 就是一个最常用的宏，可以看到它和函数最大的区别是：它在调用时多了一个`!`，除此之外还有 `vec!`、`assert_eq!` 都是相当常用的，可以说**宏在 Rust 中无处不在**。

细心的读者可能会注意到 `println!` 后面跟着的是`()`，而 `vec!` 后面跟着的是`[]`，这是因为宏的参数可以使用`()`、`[]` 以及`{}`：

```
fn main() {
    println!("aaaa");
    println!["aaaa"];
    println!{"aaaa"}
}
```

虽然三种使用形式皆可，但是 Rust 内置的宏都有自己约定俗成的使用方式，例如 `vec![...]`、`assert_eq!(...)` 等。

在 Rust 中宏分为两大类：**声明式宏(declarative macros)** `macro_rules!` 和三种**过程宏(procedural macros)**：

- `#[derive]`，在之前多次见到的派生宏，可以为目标结构体或枚举派生指定的代码，例如 `Debug` 特征
- 类属性宏(Attribute-like macro)，用于为目标添加自定义的属性
- 类函数宏(Function-like macro)，看上去就像是函数调用

如果感觉难以理解，也不必担心，接下来我们将逐个看看它们的庐山真面目，在此之前，先来看下为何需要宏，特别是 Rust 的函数明明已经很强大了。

## 宏和函数的区别

宏和函数的区别并不少，而且对于宏擅长的领域，函数其实是有些无能为力的。

## 元编程

从根本上来说，宏是通过一种代码来生成另一种代码，如果大家熟悉元编程，就会发现两者的共同点。

在[附录 D](#)中讲到的 `derive` 属性，就会自动为结构体派生出相应特征所需的代码，例如 `#[derive(Debug)]`，还有熟悉的 `println!` 和 `vec!`，所有的这些宏都会展开成相应的代码，且很可能是长得多的代码。

总之，元编程可以帮我们减少所需编写的代码，也可以一定程度上减少维护的成本，虽然函数复用也有类似的作用，但是宏依然拥有自己独特的优势。

## 可变参数

Rust 的函数签名是固定的：定义了两个参数，就必须传入两个参数，多一个少一个都不行，对于从 JS/TS 过来的同学，这一点其实是有些恼人的。

而宏就可以拥有可变数量的参数，例如可以调用一个参数的 `println!("hello")`，也可以调用两个参数的 `println!("hello {}", name)`。

## 宏展开

由于宏会被展开成其它代码，且这个展开过程是发生在编译器对代码进行解释之前。因此，宏可以为指定的类型实现某个特征：先将宏展开成实现特征的代码后，再被编译。

而函数就做不到这一点，因为它直到运行时才能被调用，而特征需要在编译期被实现。

## 宏的缺点

相对函数来说，由于宏是基于代码再展开成代码，因此实现相比函数来说会更加复杂，再加上宏的语法更为复杂，最终导致定义宏的代码相当地难读，也难以理解和维护。

# 声明式宏 `macro_rules!`!

在 Rust 中使用最广的就是声明式宏，它们也有一些其它的称呼，例如示例宏(`macros by example`)、`macro_rules!` 或干脆直接称呼为**宏**。

声明式宏允许我们写出类似 `match` 的代码。`match` 表达式是一个控制结构，其接收一个表达式，然后将表达式的结果与多个模式进行匹配，一旦匹配了某个模式，则该模式相关联的代码将被执行：

```
match target {
    模式1 => 表达式1,
    模式2 => {
        语句1;
        语句2;
        表达式2
    },
    _ => 表达式3
}
```

而宏也是将一个值跟对应的模式进行匹配，且该模式会与特定的代码相关联。但是与 `match` 不同的是，**宏里的值是一段 Rust 源代码(字面量)**，模式用于跟这段源代码的结构相比较，一旦匹配，传入宏的那段源代码将被模式关联的代码所替换，最终实现宏展开。值得注意的是，**所有的这些都在编译期发生，并没有运行期的性能损耗。**

## 简化版的 `vec!`

在[动态数组 Vector 章节](#)中，我们学习了使用 `vec!` 来便捷的初始化一个动态数组：

```
let v: Vec<u32> = vec![1, 2, 3];
```

最重要的是，通过 `vec!` 创建的动态数组支持任何元素类型，也并没有限制数组的长度，如果使用函数，我们是无法做到这一点的。

好在我们有 `macro_rules!`，来看看该如何使用它来实现 `vec!`，以下是一个简化实现：

```
#[macro_export]
macro_rules! vec {
    ( $($x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

简化实现版本？这也太难了吧！！只能说，欢迎来到宏的世界，在这里你能见到优雅 Rust 的另一面:) 标准库中的 `vec!` 还包含了预分配内存空间的代码，如果引入进来，那大家将更难以接受。

`#[macro_export]` 注释将宏进行了导出，这样其它的包就可以将该宏引入到当前作用域中，然后才能使用。可能有同学会提问：我们在使用标准库 `vec!` 时也没有引入宏啊，那是因为 Rust 已经通过 `std::prelude` 的方式为我们自动引入了。

紧接着，就使用 `macro_rules!` 进行了宏定义，需要注意的是宏的名称是 `vec`，而不是 `vec!`，后者的感叹号只在调用时才需要。

`vec` 的定义结构跟 `match` 表达式很像，但这里我们只有一个分支，其中包含一个模式 `( $( $x:expr ),*` )，跟模式相关联的代码就在 `=>` 之后。一旦模式成功匹配，那这段相关联的代码就会替换传入的源代码。

由于 `vec` 宏只有一个模式，因此它只能匹配一种源代码，其它类型的都将导致报错，而更复杂的宏往往会有更多的分支。

虽然宏和 `match` 都称之为模式，但是前者跟[后者的](#)模式规则是不同的。如果大家想要更深入的了解宏的模式，可以查看[这里](#)。

## 模式解析

而现在，我们先来简单讲解下 `( $( $x:expr ),*` ) 的含义。

首先，我们使用圆括号 `()` 将整个宏模式包裹其中。紧随其后的是 `$()`，跟括号中模式相匹配的值(传入的 Rust 源代码)会被捕获，然后用于代码替换。在这里，模式 `$x:expr` 会匹配任何 Rust 表达式并给予该模式一个名称：`$x`。

`$()` 之后的逗号说明在 `$()` 所匹配的代码的后面会有一个可选的逗号分隔符，紧随逗号之后的 `*` 说明 `*` 之前的模式会被匹配零次或任意多次(类似正则表达式)。

当我们使用 `vec![1, 2, 3]` 来调用该宏时，`$x` 模式将被匹配三次，分别是 `1`、`2`、`3`。为了帮助大家巩固，我们再来一起过一下：

1. `$()` 中包含的是模式 `$x:expr`，该模式中的 `expr` 表示会匹配任何 Rust 表达式，并给予该模式一个名称 `$x`
2. 因此 `$x` 模式可以跟整数 `1` 进行匹配，也可以跟字符串 `"hello"` 进行匹配：`vec!["hello", "world"]`
3. `$()` 之后的逗号，意味着 `1` 和 `2` 之间可以使用逗号进行分割，也意味着 `3` 既可以没有逗号，也可以有逗号：`vec![1, 2, 3, ]`
4. `*` 说明之前的模式可以出现零次也可以任意次，这里出现了三次

接下来，我们再来看看与模式相关联、在 `=>` 之后的代码：

```
{  
    {  
        let mut temp_vec = Vec::new();  
        $()  
            temp_vec.push($x);  
        )*  
        temp_vec  
    }  
};
```

这里就比较好理解了，`$()` 中的 `temp_vec.push()` 将根据模式匹配的次数生成对应的代码，当调用 `vec![1, 2, 3]` 时，下面这段生成的代码将替代传入的源代码，也就是替代 `vec![1, 2, 3]`：

```
{  
    let mut temp_vec = Vec::new();  
    temp_vec.push(1);  
    temp_vec.push(2);  
    temp_vec.push(3);  
    temp_vec  
}
```

如果是 `let v = vec![1, 2, 3]`，那生成的代码最后返回的值 `temp_vec` 将被赋予给变量 `v`，等同于：

```
let v = {  
    let mut temp_vec = Vec::new();  
    temp_vec.push(1);  
    temp_vec.push(2);  
    temp_vec.push(3);  
    temp_vec  
}
```

至此，我们定义了一个宏，它可以接受任意类型和数量的参数，并且理解了其语法的含义。

## 未来将被替代的 `macro_rules`

对于 `macro_rules` 来说，它是存在一些问题的，因此，Rust 计划在未来使用新的声明式宏来替换它：工作方式类似，但是解决了目前存在的一些问题，在那之后，`macro_rules` 将变为 `deprecated` 状态。

由于绝大多数 Rust 开发者都是宏的用户而不是编写者，因此在这里我们不会对 `macro_rules` 进行更深入的学习，如果大家感兴趣，可以看看这本书 “The Little Book of Rust Macros”。

## 用过程宏为属性标记生成代码

第二种常用的宏就是[过程宏](#)(*procedural macros*)，从形式上来看，过程宏跟函数较为相像，但过程宏是使用源代码作为输入参数，基于代码进行一系列操作后，再输出一段全新的代码。**注意，过程宏中的 derive 宏输出的代码并不会替换之前的代码，这一点与声明宏有很大的不同！**

至于前文提到的过程宏的三种类型(自定义 `derive`、属性宏、函数宏)，它们的工作方式都是类似的。

当创建过程宏时，它的定义必须要放入一个独立的包中，且包的类型也是特殊的，这么做的原因相当复杂，大家只要知道这种限制在未来可能会有所改变即可。

---

事实上，根据[这个说法](#)，过程宏放入独立包的原因在于它必须先被编译后才能使用，如果过程宏和使用它的代码在一个包，就必须先单独对过程宏的代码进行编译，然后再对我们的代码进行编译，但悲剧的是 Rust 的编译单元是包，因此你无法做到这一点。

---

假设我们要创建一个 `derive` 类型的过程宏：

```
use proc_macro;

#[proc_macro_derive(HelloMacro)]
pub fn some_name(input: TokenStream) -> TokenStream {  
}
```

用于定义过程宏的函数 `some_name` 使用 `TokenStream` 作为输入参数，并且返回的也是同一个类型。`TokenStream` 是在 `proc_macro` 包中定义的，顾名思义，它代表了一个 `Token` 序列。

在理解了过程宏的基本定义后，我们再来看看该如何创建三种类型的过程宏，首先，从大家最熟悉的 `derive` 开始。

## 自定义 `derive` 过程宏

假设我们有一个特征 `HelloMacro`，现在有两种方式让用户使用它：

- 为每个类型手动实现该特征，就像之前[特征章节](#)所做的
- 使用过程宏来统一实现该特征，这样用户只需要对类型进行标记即可：`#[derive(HelloMacro)]`

以上两种方式并没有孰优孰劣，主要在于不同的类型是否可以使用同样的默认特征实现，如果可以，那过程宏的方式可以帮我们减少很多代码实现：

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Sunfei;

#[derive(HelloMacro)]
struct Sunface;

fn main() {
    Sunfei::hello_macro();
    Sunface::hello_macro();
}
```

简单吗？简单！不过为了实现这段代码展示的功能，我们还需要创建相应的过程宏才行。首先，创建一个新的工程用于演示：

```
$ cargo new hello_macro
$ cd hello_macro/
$ touch src/lib.rs
```

此时，`src` 目录下包含两个文件 `lib.rs` 和 `main.rs`，前者是 `lib` 包根，后者是二进制包根，如果大家对包根不熟悉，可以看看[这里](#)。

接下来，先在 `src/lib.rs` 中定义过程宏所需的 `HelloMacro` 特征和其关联函数：

```
pub trait HelloMacro {
    fn hello_macro();
}
```

然后在 `src/main.rs` 中编写主体代码，首先映入大家脑海的可能会是如下实现：

```

use hello_macro::HelloMacro;

struct Sunfei;

impl HelloMacro for Sunfei {
    fn hello_macro() {
        println!("Hello, Macro! My name is Sunfei!");
    }
}

struct Sunface;

impl HelloMacro for Sunface {
    fn hello_macro() {
        println!("Hello, Macro! My name is Sunface!");
    }
}

fn main() {
    Sunfei::hello_macro();
}

```

但是这种方式有个问题，如果想要实现不同的招呼内容，就需要为每一个类型都实现一次相应的特征，Rust 不支持反射，因此我们无法在运行时获得类型名。

使用宏，就不存在这个问题：

```

use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Sunfei;

#[derive(HelloMacro)]
struct Sunface;

fn main() {
    Sunfei::hello_macro();
    Sunface::hello_macro();
}

```

简单明了的代码总是令人愉快，为了让代码运行起来，还需要定义下过程宏。就如前文提到的，目前只能在单独的包中定义过程宏，尽管未来这种限制会被取消，但是现在我们还得遵循这个规则。

宏所在的包名自然也有要求，必须以 `derive` 为后缀，对于 `hello_macro` 宏而言，包名就应该是 `hello_macro_derive`。在之前创建的 `hello_macro` 项目根目录下，运行如下命令，创建一个单独的 `lib` 包：

```
cargo new hello_macro_derive --lib
```

至此，`hello_macro` 项目的目录结构如下：

```
hello_macro
├── Cargo.toml
└── src
    ├── main.rs
    └── lib.rs
└── hello_macro_derive
    ├── Cargo.toml
    └── src
        └── lib.rs
```

由于过程宏所在的包跟我们的项目紧密相连，因此将它放在项目之中。现在，问题又来了，该如何在项目的 `src/main.rs` 中引用 `hello_macro_derive` 包的内容？

方法有两种，第一种是将 `hello_macro_derive` 发布到 `crates.io` 或 GitHub 中，就像我们引用的其它依赖一样；另一种就是使用相对路径引入的本地化方式，修改 `hello_macro/Cargo.toml` 文件添加以下内容：

```
[dependencies]
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
# 也可以使用下面的相对路径
# hello_macro_derive = { path = "./hello_macro_derive" }
```

此时，`hello_macro` 项目就可以成功的引用到 `hello_macro_derive` 本地包了，对于项目依赖引入的详细介绍，可以参见 [Cargo 章节](#)。

另外，学习过程更好的办法是通过展开宏来阅读和调试自己写的宏，这里需要用到一个 `cargo-expand` 的工具，可以通过下面的命令安装

```
cargo install cargo-expand
```

接下来，就到了重头戏环节，一起来看看该如何定义过程宏。

## 定义过程宏

首先，在 `hello_macro_derive/Cargo.toml` 文件中添加以下内容：

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

其中 `syn` 和 `quote` 依赖包都是定义过程宏所必需的，同时，还需要在 `[lib]` 中将过程宏的开关开启：  
`proc-macro = true`。

其次，在 `hello_macro_derive/src/lib.rs` 中添加如下代码：

```
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;
use syn;
use syn::DeriveInput;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // 基于 input 构建 AST 语法树
    let ast: DeriveInput = syn::parse(input).unwrap();

    // 构建特征实现代码
    impl_hello_macro(&ast)
}
```

这个函数的签名我们在之前已经介绍过，总之，这种形式的过程宏定义是相当通用的，下面来分析下这段代码。

首先有一点，对于绝大多数过程宏而言，这段代码往往只在 `impl_hello_macro(&ast)` 中的实现有所区别，对于其它部分基本都是一致的，如包的引入、宏函数的签名、语法树构建等。

`proc_macro` 包是 Rust 自带的，因此无需在 `Cargo.toml` 中引入依赖，它包含了相关的编译器 API，可以用于读取和操作 Rust 源代码。

由于我们为 `hello_macro_derive` 函数标记了 `#[proc_macro_derive(HelloMacro)]`，当用户使用 `#[derive(HelloMacro)]` 标记了他的类型后，`hello_macro_derive` 函数就将被调用。这里的秘诀就是特征名 `HelloMacro`，它就像一座桥梁，将用户的类型和过程宏联系在一起。

`syn` 将字符串形式的 Rust 代码解析为一个 AST 树的数据结构，该数据结构可以在随后的 `impl_hello_macro` 函数中进行操作。最后，操作的结果又会被 `quote` 包转换回 Rust 代码。这些包非常关键，可以帮助我们节省大量的精力，否则你需要自己去编写支持代码解析和还原的解析器，这可不是一件简单的任务！

derive过程宏只能用在struct/enum/union上，多数用在结构体上，我们先来看一下一个结构体由哪些部分组成：

```
// vis, 可视范围          ident, 标识符      generic, 范型      fields: 结构体的字段
pub          struct    User           <'a, T>       {
// vis  ident  type
pub  name:  &'a T,
}
```

其中type还可以细分，具体请阅读syn文档或源码

syn::parse 调用会返回一个 DeriveInput 结构体来代表解析后的 Rust 代码：

```
DeriveInput {
    // --snip--
    vis: Visibility,
    ident: Ident {
        ident: "Sunfei",
        span: #0 bytes(95..103)
    },
    generics: Generics,
    // Data是一个枚举，分别是DataStruct, DataEnum, DataUnion, 这里以 DataStruct 为例
    data: Data(
        DataStruct {
            struct_token: Struct,
            fields: Fields,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

以上就是源代码 `struct Sunfei;` 解析后的结果，里面有几点值得注意：

- `fields: Fields` 是一个枚举类型，`Fields::Named`, `Fields::Unnamed`, `Fields::Unit` 分别表示结构体中的显式命名字段（如例子所示），元组或元组变体中的匿名字段（例如 `Some(T)`），单元类型或单元变体字段（例如 `None`）。
- `ident: "Sunfei"` 说明类型名称为 `Sunfei`，`ident` 是标识符 `identifier` 的简写

如果想要了解更多的信息，可以查看 [syn 文档](#)。

大家可能会注意到在 `hello_macro_derive` 函数中有 `unwrap` 的调用，也许会以为这是为了演示目的，没有做错误处理，实际上并不是的。由于该函数只能返回 `TokenStream` 而不是 `Result`，那么在报错时

直接 `panic` 来抛出错误就成了相当好的选择。当然，这里实际上还是做了简化，在生产项目中，你应该通过 `panic!` 或 `expect` 抛出更具体的报错信息。

至此，这个函数大家应该已经基本理解了，下面来看看如何构建特征实现的代码，也是过程宏的核心目标：

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        };
        gen.into()
    }
}
```

首先，将结构体的名称赋予给 `name`，也就是 `name` 中会包含一个字段，它的值是字符串 "Sunfei"。

其次，使用 `quote!` 可以定义我们想要返回的 Rust 代码。由于编译器需要的内容和 `quote!` 直接返回的不一样，因此还需要使用 `.into` 方法将其转换为 `TokenStream`。

大家注意到 `#name` 的使用了吗？这也是 `quote!` 提供的功能之一，如果想要深入了解 `quote`，可以看看[官方文档](#)。

特征的 `hell_macro()` 函数只有一个功能，就是使用 `println!` 打印一行欢迎语句。

其中 `stringify!` 是 Rust 提供的内置宏，可以将一个表达式(例如 `1 + 2`)在编译期转换成一个字符串字面值(`"1 + 2"`)，该字面量会直接打包进编译出的二进制文件中，具有 `'static` 生命周期。而 `format!` 宏会对表达式进行求值，最终结果是一个 `String` 类型。在这里使用 `stringify!` 有两个好处：

- `#name` 可能是一个表达式，我们需要它的字面值形式
- 可以减少一次 `String` 带来的内存分配

在运行之前，可以先用 `expand` 展开宏，观察是否有错误或符合预期：

```
$ cargo expand --bin hello_macro
```

```

struct Sunfei;
impl HelloMacro for Sunfei {
    fn hello_macro() {
        {
            ::std::io::_print(
                ::core::fmt::Arguments::new_v1(
                    &["Hello, Macro! My name is ", "\n"],
                    &[::core::fmt::ArgumentV1::new_display(&"Sunfei")],
                ),
            );
        };
    }
}
struct Sunface;
impl HelloMacro for Sunface {
    fn hello_macro() {
        {
            ::std::io::_print(
                ::core::fmt::Arguments::new_v1(
                    &["Hello, Macro! My name is ", "\n"],
                    &[::core::fmt::ArgumentV1::new_display(&"Sunface")],
                ),
            );
        };
    }
}
fn main() {
    Sunfei::hello_macro();
    Sunface::hello_macro();
}

```

从展开的代码也能看出derive宏的特性，`struct Sunfei;` 和 `struct Sunface;` 都被保留了，也就是说最后 `impl_hello_macro()` 返回的token被加到结构体后面，这和类属性宏可以修改输入的token是不一样的，input的token并不能被修改。

至此，过程宏的定义、特征定义、主体代码都已经完成，运行下试试：

```

$ cargo run

Running `target/debug/hello_macro`
Hello, Macro! My name is Sunfei!
Hello, Macro! My name is Sunface!

```

Bingo，虽然过程有些复杂，但是结果还是很喜人，我们终于完成了自己的第一个过程宏！

下面来实现一个更实用的例子，实现官方的#[derive(Default)]宏，废话不说直接开干：

```

extern crate proc_macro;
use proc_macro::TokenStream;
use quote::quote;
use syn::{self, Data};
use syn::DeriveInput;

#[proc_macro_derive(MyDefault)]
pub fn my_default(input: TokenStream) -> TokenStream {
    let ast: DeriveInput = syn::parse(input).unwrap();
    let id = ast.ident;

    let Data::Struct(s) = ast.data else{
        panic!("MyDefault derive macro must use in struct");
    };

    // 声明一个新的ast，用于动态构建字段赋值的token
    let mut field_ast = quote!();

    // 这里就是要动态添加token的地方了，需要动态完成Self的字段赋值
    for (idx,f) in s.fields.iter().enumerate() {
        let (field_id, field_ty) = (&f.ident, &f.ty);

        if field_id.is_none(){
            //没有ident表示是匿名字段，对于匿名字段，都需要添加 `#field_idx:#field_type::default(),` 这样的代码
            let field_idx = syn::Index::from(idx);
            field_ast.extend(quote! {
                });
        }else{
            //对于命名字段，都需要添加 `#field_name: #field_type::default(),` 这样的代码
            field_ast.extend(quote! {
                });
        }
    }

    quote! {
        impl Default for #id {
            fn default() -> Self {
                Self {
                }
            }
        },.into()
    }
}

```

然后来写使用代码:

```
#[derive(MyDefault)]
struct SomeData (u32, String);

#[derive(MyDefault)]
struct User {
    name: String,
    data: SomeData,
}

fn main() {
```

然后我们先展开代码看一看

```
struct SomeData(u32, String);
impl Default for SomeData {
    fn default() -> Self {
        Self {
            0: u32::default(),
            1: String::default(),
        }
    }
}
struct User {
    name: String,
    data: SomeData,
}
impl Default for User {
    fn default() -> Self {
        Self {
            name: String::default(),
            data: SomeData::default(),
        }
    }
}
fn main() {}
```

展开的代码符合预期，然后我们修改一下使用代码并测试结果

```
#[derive(MyDefault, Debug)]
struct SomeData (u32, String);

#[derive(MyDefault, Debug)]
struct User {
    name: String,
    data: SomeData,
}

fn main() {
    println!("{:?}", User::default());
}
```

执行

```
$ cargo run

Running `target/debug/aaa`
User { name: "", data: SomeData(0, "") }
```

接下来，再来看看过程宏的另外两种类型跟 `derive` 类型有何区别。

## 类属性宏(Attribute-like macros)

类属性过程宏跟 `derive` 宏类似，但是前者允许我们定义自己的属性。除此之外，`derive` 只能用于结构体和枚举，而类属性宏可以用于其它类型项，例如函数。

假设我们在开发一个 `web` 框架，当用户通过 `HTTP GET` 请求访问 / 根路径时，使用 `index` 函数为其提供服务：

```
#[route(GET, "/")]
fn index() {
```

如上所示，代码功能非常清晰、简洁，这里的 `#[route]` 属性就是一个过程宏，它的定义函数大概如下：

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

与 `derive` 宏不同，类属性宏的定义函数有两个参数：

- 第一个参数时用于说明属性包含的内容：Get， "/" 部分
- 第二个是属性所标注的类型项，在这里是 `fn index() {...}`，注意，函数体也被包含其中

除此之外，类属性宏跟 `derive` 宏的工作方式并无区别：创建一个包，类型是 `proc-macro`，接着实现一个函数用于生成想要的代码。

## 类函数宏(Function-like macros)

类函数宏可以让我们定义像函数那样调用的宏，从这个角度来看，它跟声明宏 `macro_rules` 较为类似。

区别在于，`macro_rules` 的定义形式与 `match` 匹配非常相像，而类函数宏的定义形式则类似于之前讲过的两种过程宏：

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

而使用形式则类似于函数调用：

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

大家可能会好奇，为何我们不使用声明宏 `macro_rules` 来定义呢？原因是这里需要对 SQL 语句进行解析并检查其正确性，这个复杂的过程是 `macro_rules` 难以对付的，**而过程宏相比起来就会灵活的多。**

## 补充学习资料

1. [dtolnay/proc-macro-workshop](#)，学习如何编写过程宏
2. [The Little Book of Rust Macros](#)，学习如何编写声明宏 `macro_rules!`
3. [syn](#) 和 [quote](#)，用于编写过程宏的包，它们的文档有很多值得学习的东西
4. [Structuring, testing and debugging procedural macro crates](#)，从测试、debug、结构化的角度来编写过程宏
5. [blog.turbo.fish](#)，里面的过程宏系列文章值得一读
6. [Rust 宏小册中文版](#)，非常详细的解释了宏各种知识

## 总结

Rust 中的宏主要分为两大类：声明宏和过程宏。

声明宏目前使用 `macro_rules` 进行创建，它的形式类似于 `match` 匹配，对于用户而言，可读性和维护性都较差。由于其存在的问题和限制，在未来，`macro_rules` 会被 `deprecated`，Rust 会使用一个新

的声明宏来替代它。

而过程宏的定义更像是我们平时写函数的方式，因此它更加灵活，它分为三种类型：`derive` 宏、类属性宏、类函数宏，具体在文中都有介绍。

虽然 Rust 中的宏很强大，但是它并不应该成为我们的常规武器，原因是它会影响 Rust 代码的可读性和可维护性，我相信没有几个人愿意去维护别人写的宏：）

因此，大家应该熟悉宏的使用场景，但是不要滥用，当你真的需要时，再回来查看本章了解实现细节，这才是最完美的使用方式。

# 异步编程

在艰难的学完 Rust 入门和进阶所有的 70 个章节后，我们终于来到了这里。假如之前攀登的是珠穆朗玛峰，那么现在攀登的就是乔戈里峰( 比珠峰还难攀爬... )。

如果你想开发 Web 服务器、数据库驱动、消息服务等需要高并发的服务，那么本章的内容将值得认真对待和学习。

接下来，我们将深入了解 `async/await` 的使用方式及背后的原理。

---

本章在内容上大量借鉴和翻译了原版英文书籍[Asynchronous Programming In Rust](#), 特此感谢

---

# Async 编程简介

众所周知，Rust 可以让我们写出性能高且安全的软件，那么异步编程这块儿呢？是否依然在高性能的同时保证了安全？

我们先通过一张 web 框架性能对比图来感受下 Rust 异步编程的性能：

Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext	Weighted score
5	actix	1,563,586	635,091	34,955	653,529	24,301	7,004,195	8,894   83.6%
55	gin	418,675	204,101	14,464	95,845	1,061	716,953	1,667   15.7%

上图并不能说 Rust 写的 `actix` 框架比 Go 的 `gin` 更好、更优秀，但是确实可以一定程度上说明 Rust 的异步性能非常的高！

简单来说，异步编程是一个[并发编程模型](#)，目前主流语言基本都支持了，当然，支持的方式有所不同。异步编程允许我们同时并发运行大量的任务，却仅仅需要几个甚至一个 OS 线程或 CPU 核心，现代化的异步编程在使用体验上跟同步编程也几无区别，例如 Go 语言的 `go` 关键字，也包括我们后面将介绍的 `async/await` 语法，该语法是 JavaScript 和 Rust 的核心特性之一。

## async 简介

`async` 是 Rust 选择的异步编程模型，下面我们来介绍下它的优缺点，以及何时适合使用。

### async vs 其它并发模型

由于并发编程在现代社会非常重要，因此每个主流语言都对自己的并发模型进行过权衡取舍和精心设计，Rust 语言也不例外。下面的列表可以帮助大家理解不同并发模型的取舍：

- **OS 线程**, 它最简单，也无需改变任何编程模型(业务/代码逻辑)，因此非常适合作为语言的原生并发模型，我们在[多线程章节](#)也提到过，Rust 就选择了原生支持线程级的并发编程。但是，这种模型也有缺点，例如线程间的同步将变得更加困难，线程间的上下文切换损耗较大。使用线程池在一定程度上可以提升性能，但是对于 IO 密集的场景来说，线程池还是不够。
- **事件驱动(Event driven)**, 这个名词你可能比较陌生，如果说事件驱动常常跟回调( Callback )一起使用，相信大家就恍然大悟了。这种模型性能相当的好，但最大的问题是存在回调地狱的风险：非线性的控制流和结果处理导致了数据流向和错误传播变得难以掌控，还会导致代码可维护性和可读性的大幅降低，大名鼎鼎的 JavaScript 曾经就存在回调地狱。
- **协程(Coroutines)** 可能是目前最火的并发模型，Go 语言的协程设计就非常优秀，这也是 Go 语言能够迅速火遍全球的杀手锏之一。协程跟线程类似，无需改变编程模型，同时，它也跟 `async` 类

似，可以支持大量的任务并发运行。但协程抽象层次过高，导致用户无法接触到底层的细节，这对于系统编程语言和自定义异步运行时是难以接受的

- **actor 模型**是 erlang 的杀手锏之一，它将所有并发计算分割成一个一个单元，这些单元被称为 `actor`，单元之间通过消息传递的方式进行通信和数据传递，跟分布式系统的设计理念非常相像。由于 `actor` 模型跟现实很贴近，因此它相对来说更容易实现，但是一旦遇到流控制、失败重试等场景时，就会变得不太好用
- **async/await**，该模型性能高，还能支持底层编程，同时又像线程和协程那样无需过多的改变编程模型，但有得必有失，`async` 模型的问题就是内部实现机制过于复杂，对于用户来说，理解和使用起来也没有线程和协程简单，好在前者的复杂性开发者们已经帮我们封装好，而理解和使用起来不够简单，正是本章试图解决的问题。

总之，Rust 经过权衡取舍后，最终选择了同时提供多线程编程和 `async` 编程：

- 前者通过标准库实现，当你无需那么高的并发时，例如需要并行计算时，可以选择它，优点是线程内的代码执行效率更高、实现更直观更简单，这块内容已经在多线程章节进行过深入讲解，不再赘述
- 后者通过语言特性 + 标准库 + 三方库的方式实现，在你需要高并发、异步 I/O 时，选择它就对了

## async: Rust vs 其它语言

目前已经有诸多语言都通过 `async` 的方式提供了异步编程，例如 `JavaScript`，但 Rust 在实现上有所区别：

- **Future 在 Rust 中是惰性的**，只有在被轮询(`poll`)时才会运行，因此丢弃一个 `future` 会阻止它未来再被运行，你可以将 `Future` 理解为一个在未来某个时间点被调度执行的任务。
- **Async 在 Rust 中使用开销是零**，意味着只有你能看到的代码(自己的代码)才有性能损耗，你看不到的(`async` 内部实现)都没有性能损耗，例如，你可以无需分配任何堆内存、也无需任何动态分发来使用 `async`，这对于热点路径的性能有非常大的好处，正是得益于此，Rust 的异步编程性能才会这么高。
- **Rust 没有内置异步调用所必需的运行时**，但是无需担心，Rust 社区生态中已经提供了非常优异的运行时实现，例如大明星 `tokio`
- **运行时同时支持单线程和多线程**，这两者拥有各自的优缺点，稍后会讲

## Rust: async vs 多线程

虽然 `async` 和多线程都可以实现并发编程，后者甚至还能通过线程池来增强并发能力，但是这两个方式并不互通，从一个方式切换成另一个需要大量的代码重构工作，因此提前为自己的项目选择适合的并发模型就变得至关重要。

os 线程非常适合少量任务并发，因为线程的创建和上下文切换是非常昂贵的，甚至于空闲的线程都会消耗系统资源。虽说线程池可以有效的降低性能损耗，但是也无法彻底解决问题。当然，线程模型也有其优

点，例如它不会破坏你的代码逻辑和编程模型，你之前的顺序代码，经过少量修改适配后依然可以在新线程中直接运行，同时在某些操作系统中，你还可以改变线程的优先级，这对于实现驱动程序或延迟敏感的应用(例如硬实时系统)很有帮助。

对于长时间运行的 CPU 密集型任务，例如并行计算，使用线程将更有优势。这种密集任务往往会让所在的线程持续运行，任何不必要的线程切换都会带来性能损耗，因此高并发反而在此时成为了一种多余。同时你所创建的线程数应该等于 CPU 核心数，充分利用 CPU 的并行能力，甚至还可以将线程绑定到 CPU 核心上，进一步减少线程上下文切换。

而高并发更适合 IO 密集型任务，例如 web 服务器、数据库连接等等网络服务，因为这些任务绝大部分时间都处于等待状态，如果使用多线程，那线程大量时间会处于无所事事的状态，再加上线程上下文切换的高昂代价，让多线程做 IO 密集任务变成了一件非常奢侈的事。而使用 `async`，既可以有效的降低 CPU 和内存的负担，又可以让大量的任务并发的运行，一个任务一旦处于 IO 或者其他等待(阻塞)状态，就会被立刻切走并执行另一个任务，而这里的任务切换的性能开销要远远低于使用多线程时的线程上下文切换。

事实上，`async` 底层也是基于线程实现，但是它基于线程封装了一个运行时，可以将多个任务映射到少量线程上，然后将线程切换变成了任务切换，后者仅仅是内存中的访问，因此要高效的多。

不过 `async` 也有其缺点，原因是编译器会为 `async` 函数生成状态机，然后将整个运行时打包进来，这会造成我们编译出的二进制可执行文件体积显著增大。

总之，`async` 编程并没有比多线程更好，最终还是根据你的使用场景作出合适的选择，如果无需高并发，或者也不在意线程切换带来的性能损耗，那么多线程使用起来会简单、方便的多！最后再简单总结下：

---

若大家使用 `tokio`，那 CPU 密集的任务尤其需要用线程的方式去处理，例如使用 `spawn_blocking` 创建一个阻塞的线程去完成相应 CPU 密集任务。

至于具体的原因，不仅是上文说到的那些，还有一个是：`tokio` 是协作式的调度器，如果某个 CPU 密集的异步任务是通过 `tokio` 创建的，那理论上来说，该异步任务需要跟其它的异步任务交错执行，最终大家都得到了执行，皆大欢喜。但实际情况是，CPU 密集的任务很可能会一直霸占着 CPU，此时 `tokio` 的调度方式决定了该任务会一直被执行，这意味着，其它的异步任务无法得到执行的机会，最终这些任务都会因为得不到资源而饿死。

而使用 `spawn_blocking` 后，会创建一个单独的 OS 线程，该线程并不会被 `tokio` 所调度(被 OS 所调度)，因此它所执行的 CPU 密集任务也不会导致 `tokio` 调度的那些异步任务被饿死

---

- 有大量 IO 任务需要并发运行时，选 `async` 模型
- 有部分 IO 任务需要并发运行时，选多线程，如果想要降低线程创建和销毁的开销，可以使用线程池

- 有大量 CPU 密集任务需要并行运行时，例如并行计算，选多线程模型，且让线程数等于或者稍大于 CPU 核心数
- 无所谓时，统一选多线程

## async 和多线程的性能对比

操作	async	线程
创建	0.3 微秒	17 微秒
线程切换	0.2 微秒	1.7 微秒

可以看出， async 在线程切换的开销显著低于多线程，对于 IO 密集的场景，这种性能开销累计下来会非常可怕！

## 一个例子

在大概理解 async 后，我们再来看一个简单的例子。如果想并发的下载文件，你可以使用多线程如下实现：

```
fn get_two_sites() {
    // 创建两个新线程执行任务
    let thread_one = thread::spawn(|| download("https://course.rs"));
    let thread_two = thread::spawn(|| download("https://fancy.rs"));

    // 等待两个线程的完成
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}
```

如果是在一个小项目中简单的去下载文件，这么写没有任何问题，但是一旦下载文件的并发请求多起来，那一个下载任务占用一个线程的模式就太重了，会很容易成为程序的瓶颈。好在，我们可以使用 async 的方式来解决：

```
async fn get_two_sites_async() {
    // 创建两个不同的`future`，你可以把`future`理解为未来某个时刻会被执行的计划任务
    // 当两个`future`被同时执行后，它们将并发的去下载目标页面
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // 同时运行两个`future`，直至完成
    join!(future_one, future_two);
}
```

此时，不再有线程创建和切换的昂贵开销，所有的函数都是通过静态的方式进行分发，同时也没有任何内存分配发生。这段代码的性能简直无懈可击！

事实上，`async` 和多线程并不是二选一，在同一应用中，可以根据情况两者一起使用，当然，我们还可以使用其它的并发模型，例如上面提到事件驱动模型，前提是三方库提供了相应的实现。

## Async Rust 当前的进展

简而言之，Rust 语言的 `async` 目前还没有达到多线程的成熟度，其中一部分内容还在不断进化中，当然，这并不影响我们在生产级项目中使用，因为社区中还有 `tokio` 这种大杀器。

使用 `async` 时，你会遇到好的，也会遇到不好的，例如：

- 收获卓越的性能
- 会经常跟进阶语言特性打交道，例如生命周期等，这些家伙可不好对付
- 一些兼容性问题，例如同步和异步代码、不同的异步运行时(`tokio` 与 `async-std`)
- 更昂贵的维护成本，原因是 `async` 和社区开发的运行时依然在不停的进化

总之，`async` 在 Rust 中并不是一个善茬，你会遇到更多的困难或者说坑，也会带来更高的代码阅读成本及维护成本，但是为了性能，一切都值了，不是吗？

不过好在，这些进化早晚会彻底稳定成熟，而且在实际项目中，我们往往会展开成熟的三方库，例如 `tokio`，因此可以避免一些类似的问题，但是对于本章的学习来说，`async` 的一些难点还是我们必须要去面对和征服的。

### 语言和库的支持

`async` 的底层实现非常复杂，且会导致编译后文件体积显著增加，因此 Rust 没有选择像 Go 语言那样内置了完整的特性和运行时，而是选择了通过 Rust 语言提供了必要的特性支持，再通过社区来提供 `async` 运行时的支持。因此要完整的使用 `async` 异步编程，你需要依赖以下特性和外部库：

- 所必须的特征(例如 `Future`)、类型和函数，由标准库提供实现
- 关键字 `async/await` 由 Rust 语言提供，并进行了编译器层面的支持
- 众多实用的类型、宏和函数由官方开发的 `futures` 包提供(不是标准库)，它们可以用于任何 `async` 应用中。
- `async` 代码的执行、`IO` 操作、任务创建和调度等等复杂功能由社区的 `async` 运行时提供，例如 `tokio` 和 `async-std`

还有，你在同步(`synchronous`)代码中使用的一些语言特性在 `async` 中可能将无法再使用，而且 Rust 也不允许你在特征中声明 `async` 函数(可以通过三方库实现)，总之，你会遇到一些在同步代码中不会遇

到的奇奇怪怪、形形色色的问题，不过不用担心，本章会专门用一个章节罗列这些问题，并给出相应的解决方案。

## 编译和错误

在大多数情况下，`async` 中的编译错误和运行时错误跟之前没啥区别，但是依然有以下几点值得注意：

- 编译错误，由于 `async` 编程时需要经常使用复杂的语言特性，例如生命周期和 `Pin`，因此相关的错误可能会出现的更加频繁
- 运行时错误，编译器会为每一个 `async` 函数生成状态机，这会导致在栈跟踪时会包含这些状态机的细节，同时还包含了运行时对函数的调用，因此，栈跟踪记录(例如 `panic` 时)将变得更加难以解读
- 一些隐蔽的错误也可能发生，例如在一个 `async` 上下文中去调用一个阻塞的函数，或者没有正确的实现 `Future` 特征都有可能导致这种错误。这种错误可能会悄无声息的通过编译检查甚至有时候会通过单元测试。好在一旦你深入学习并掌握了本章的内容和 `async` 原理，可以有效的降低遇到这些错误的概率

## 兼容性考虑

异步代码和同步代码并不总能和睦共处。例如，我们无法在一个同步函数中去调用一个 `async` 异步函数，同步和异步代码也往往使用不同的设计模式，这些都会导致两者融合上的困难。

甚至于有时候，异步代码之间也存在类似的问题，如果一个库依赖于特定的 `async` 运行时来运行，那么这个库非常有必要告诉它的用户，它用了这个运行时。否则一旦用户选了不同的或不兼容的运行时，就会导致不可预知的麻烦。

## 性能特性

`async` 代码的性能主要取决于你使用的 `async` 运行时，好在这些运行时都经过了精心的设计，在你能遇到的绝大多数场景中，它们都能拥有非常棒的性能表现。

但是世事皆有例外。目前主流的 `async` 运行时几乎都使用了多线程实现，相比单线程虽然增加了并发表现，但是对于执行性能会有所损失，因为多线程实现会有同步和切换上的性能开销，若你需要极致的顺序执行性能，那么 `async` 目前并不是一个好的选择。

同样的，对于延迟敏感的任务来说，任务的执行次序需要能被严格掌控，而不是交由运行时去自动调度，后者会导致不可预知的延迟，例如一个 web 服务器总是有 1% 的请求，它们的延迟会远高于其它请求，因为调度过于繁忙导致了部分任务被延迟调度，最终导致了较高的延时。正因为此，这些延迟敏感的任务非常依赖于运行时或操作系统提供调度次序上的支持。

以上的两个需求，目前的 `async` 运行时并不能很好的支持，在未来可能会有更好的支持，但在此之前，我们可以尝试用多线程解决。

# async/.await 简单入门

`async/.await` 是 Rust 内置的语言特性，可以让我们用同步的方式去编写异步的代码。

通过 `async` 标记的语法块会被转换成实现了 `Future` 特征的状态机。与同步调用阻塞当前线程不同，当 `Future` 执行并遇到阻塞时，它会让出当前线程的控制权，这样其它的 `Future` 就可以在该线程中运行，这种方式完全不会导致当前线程的阻塞。

下面我们来通过例子学习 `async/.await` 关键字该如何使用，在开始之前，需要先引入 `futures` 包。编辑 `Cargo.toml` 文件并添加以下内容：

```
[dependencies]
futures = "0.3"
```

## 使用 `async`

首先，使用 `async fn` 语法来创建一个异步函数：

```
async fn do_something() {
    println!("go go go !");
}
```

需要注意，**异步函数的返回值是一个 `Future`**，若直接调用该函数，不会输出任何结果，因为 `Future` 还未被执行：

```
fn main() {
    do_something();
}
```

运行后，`go go go` 并没有打印，同时编译器给予一个提示：`warning: unused implementer of Future that must be used`，告诉我们 `Future` 未被使用，那么到底该如何使用？答案是使用一个执行器(`executor`)：

```
// `block_on`会阻塞当前线程直到指定的`Future`执行完成，这种阻塞当前线程以等待任务完成的方式较为简单、粗暴，  
// 好在其它运行时的执行器(executor)会提供更加复杂的行为，例如将多个`future`调度到同一个线程上执行。  
use futures::executor::block_on;  
  
async fn hello_world() {  
    println!("hello, world!");  
}  
  
fn main() {  
    let future = hello_world(); // 返回一个Future，因此不会打印任何输出  
    block_on(future); // 执行`Future`并等待其运行完成，此时"hello, world!"会被打印输出  
}
```

## 使用.await

在上述代码的 `main` 函数中，我们使用 `block_on` 这个执行器等待 `Future` 的完成，让代码看上去非常像是同步代码，但是如果你要在一个 `async fn` 函数中去调用另一个 `async fn` 并等待其完成后再执行后续的代码，该如何做？例如：

```
use futures::executor::block_on;  
  
async fn hello_world() {  
    hello_cat();  
    println!("hello, world!");  
}  
  
async fn hello_cat() {  
    println!("hello, kitty!");  
}  
fn main() {  
    let future = hello_world();  
    block_on(future);  
}
```

这里，我们在 `hello_world` 异步函数中先调用了另一个异步函数 `hello_cat`，然后再输出 `hello, world!`，看看运行结果：

```
warning: unused implementer of `futures::Future` that must be used  
--> src/main.rs:6:5  
|  
6 |     hello_cat();  
|     ^^^^^^^^^^  
= note: futures do nothing unless you `.await` or poll them  
...  
hello, world!
```

不出所料，`main` 函数中的 `future` 我们通过 `block_on` 函数进行了运行，但是这里的 `hello_cat` 返回的 `Future` 却没有任何人去执行它，不过好在编译器友善的给出了提示：`futures do nothing unless you `await` or poll them`，两种解决方法：使用 `.await` 语法或者对 `Future` 进行轮询(`poll`)。

后者较为复杂，暂且不表，先来使用 `.await` 试试：

```
use futures::executor::block_on;

async fn hello_world() {
    hello_cat().await;
    println!("hello, world!");
}

async fn hello_cat() {
    println!("hello, kitty!");
}
fn main() {
    let future = hello_world();
    block_on(future);
}
```

为 `hello_cat()` 添加上 `.await` 后，结果立刻大为不同：

```
hello, kitty!
hello, world!
```

输出的顺序跟代码定义的顺序完全符合，因此，我们在上面代码中**使用同步的代码顺序实现了异步的执行效果**，非常简单、高效，而且很好理解，未来也绝对不会有关调地狱的发生。

总之，在 `async fn` 函数中使用 `.await` 可以等待另一个异步调用的完成。**但是与 `block_on` 不同，`.await` 并不会阻塞当前的线程**，而是异步的等待 `Future A` 的完成，在等待的过程中，该线程还可以继续执行其它的 `Future B`，最终实现了并发处理的效果。

## 一个例子

考虑一个载歌载舞的例子，如果不用 `.await`，我们可能会有如下实现：

```
use futures::executor::block_on;

struct Song {
    author: String,
    name: String,
}

async fn learn_song() -> Song {
    Song {
        author: "周杰伦".to_string(),
        name: String::from("《菊花台》"),
    }
}

async fn sing_song(song: Song) {
    println!(
        "给大家献上一首{}的{} ~ {}",
        song.author, song.name, "菊花残, 满地伤~ ~"
    );
}

async fn dance() {
    println!("唱到情深处, 身体不由自主的动了起来~ ~");
}

fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}
```

当然，以上代码运行结果无疑是正确的，但。。。它的性能何在？需要通过连续三次阻塞去等待三个任务的完成，一次只能做一件事，实际上我们完全可以载歌载舞啊：

```
use futures::executor::block_on;

struct Song {
    author: String,
    name: String,
}

async fn learn_song() -> Song {
    Song {
        author: "曲婉婷".to_string(),
        name: String::from("《我的歌声里》"),
    }
}

async fn sing_song(song: Song) {
    println!(
        "给大家献上一首{}的{} ~ {}",
        song.author, song.name, "你存在我深深的脑海里~ ~"
    );
}

async fn dance() {
    println!("唱到情深处，身体不由自主的动了起来~ ~");
}

async fn learn_and_sing() {
    // 这里使用`await`来等待学歌的完成，但是并不会阻塞当前线程，该线程在学歌的任务`await`后，完全可以去执行跳舞的任务
    let song = learn_song().await;

    // 唱歌必须要在学歌之后
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!`可以并发的处理和等待多个`Future`，若`learn_and_sing Future`被阻塞，那`dance Future`可以拿过线程的所有权继续执行。若`dance`也变成阻塞状态，那`learn_and_sing`又可以再次拿回线程所有权，继续执行。
    // 若两个都被阻塞，那么`async main`会变成阻塞状态，然后让出线程所有权，并将其交给`main`函数中的`block_on`执行器
    futures::join!(f1, f2);
}

fn main() {
    block_on(async_main());
}
```

上面代码中，学歌和唱歌具有明显的先后顺序，但是这两者都可以跟跳舞一同存在，也就是你可以在跳舞的时候学歌，也可以在跳舞的时候唱歌。如果上面代码不使用 `.await`，而是使用 `block_on(learn_song())`，那在学歌时，当前线程就会阻塞，不再可以做其它任何事，包括跳舞。

因此 `.await` 对于实现异步编程至关重要，它允许我们在同一个线程内并发的运行多个任务，而不是一个一个先后完成。若大家看到这里还是不太明白，强烈建议回头再仔细看一遍，同时亲自上手修改代码试试效果。

至此，读者应该对 Rust 的 `async/.await` 异步编程有了一个清晰的初步印象，下面让我们一起来看看这背后的原理：`Future` 和任务在底层如何被执行。

# 底层探秘: Future 执行器与任务调度

异步编程背后到底藏有什么秘密？究竟是哪只幕后之手在操纵这一切？如果你对这些感兴趣，就继续看下去，否则可以直接跳过，因为本章节的内容对于一个 API 工程师并没有太多帮助。

但是如果你希望能深入理解 Rust 的 `async/.await` 代码是如何工作、理解运行时和性能，甚至未来想要构建自己的 `async` 运行时或相关工具，那么本章节终究不会辜负于你。

## Future 特征

`Future` 特征是 Rust 异步编程的核心，毕竟异步函数是异步编程的核心，而 `Future` 恰恰是异步函数的返回值和被执行的关键。

首先，来给出 `Future` 的定义：它是一个能产出值的异步计算(虽然该值可能为空，例如 `()`)。光看这个定义，可能会觉得很空洞，我们来看看一个简化版的 `Future` 特征：

```
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn() -> Poll<Self::Output>);
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

在上一章中，我们提到过 `Future` 需要被执行器 `poll`(轮询)后才能运行，诺，这里 `poll` 就来了，通过调用该方法，可以推进 `Future` 的进一步执行，直到被切走为止(这里不好理解，但是你只需要知道 `Future` 并不能保证在一次 `poll` 中就被执行完，后面会详解介绍)。

若在当前 `poll` 中，`Future` 可以被完成，则会返回 `Poll::Ready(result)`，反之则返回 `Poll::Pending`，并且安排一个 `wake` 函数：当未来 `Future` 准备好进一步执行时，该函数会被调用，然后管理该 `Future` 的执行器(例如上一章节中的 `block_on` 函数)会再次调用 `poll` 方法，此时 `Future` 就可以继续执行了。

如果没有 `wake` 方法，那执行器无法知道某个 `Future` 是否可以继续被执行，除非执行器定期的轮询每一个 `Future`，确认它是否能被执行，但这种作法效率较低。而有了 `wake`，`Future` 就可以主动通知执行器，然后执行器就可以精确的执行该 `Future`。这种“事件通知->执行”的方式要远比定期对所有 `Future` 进行一次全遍历来的高效。

也许大家还是迷迷糊糊的，没事，我们用一个例子来说明下。考虑一个需要从 `socket` 读取数据的场景：如果有数据，可以直接读取数据并返回 `Poll::Ready(data)`，但如果没数据，`Future` 会被阻塞且不会再继续执行，此时它会注册一个 `wake` 函数，当 `socket` 数据准备好时，该函数将被调用以通知执行器：我们的 `Future` 已经准备好了，可以继续执行。

下面的 `SocketRead` 结构体就是一个 `Future`：

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn() -> Poll<Self::Output>) {
        if self.socket.has_data_to_read() {
            // socket有数据，写入buffer中并返回
            Poll::Ready(self.socket.read_buf())
        } else {
            // socket中还没数据
            //
            // 注册一个`wake`函数，当数据可用时，该函数会被调用，
            // 然后当前Future的执行器会再次调用`poll`方法，此时就可以读取到数据
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```

这种 `Future` 模型允许将多个异步操作组合在一起，同时还无需任何内存分配。不仅如此，如果你需要同时运行多个 `Future` 或链式调用多个 `Future`，也可以通过无内存分配的状态机实现，例如：

```

trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn() -> Poll<Self::Output>);
}

enum Poll<T> {
    Ready(T),
    Pending,
}

/// 一个SimpleFuture，它会并发地运行两个Future直到它们完成
///
/// 之所以可以并发，是因为两个Future的轮询可以交替进行，一个阻塞，另一个就可以立刻执行，反之亦然
pub struct Join<FutureA, FutureB> {
    // 结构体的每个字段都包含一个Future，可以运行直到完成。
    // 等到Future完成后，字段会被设置为 `None`。这样Future完成后，就不会再被轮询
    a: Option<FutureA>,
    b: Option<FutureB>,
}

impl<FutureA, FutureB> SimpleFuture for Join<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn() -> Poll<Self::Output>) {
        // 尝试去完成一个 Future `a`
        if let Some(a) = &mut self.a {
            if let Poll::Ready(_) = a.poll(wake) {
                self.a.take();
            }
        }

        // 尝试去完成一个 Future `b`
        if let Some(b) = &mut self.b {
            if let Poll::Ready(_) = b.poll(wake) {
                self.b.take();
            }
        }

        if self.a.is_none() && self.b.is_none() {
            // 两个 Future都已完成 - 我们可以成功地返回了
            Poll::Ready(())
        } else {
            // 至少还有一个 Future 没有完成任务，因此返回 `Poll::Pending`。
            // 当该 Future 再次准备好时，通过调用`wake()`函数来继续执行
            Poll::Pending
        }
    }
}

```

上面代码展示了如何同时运行多个 Future，且在此过程中没有任何内存分配，让并发编程更加高效。类似的，多个 Future 也可以一个接一个的连续运行：

```
/// 一个SimpleFuture，它使用顺序的方式，一个接一个地运行两个Future
//
// 注意：由于本例子用于演示，因此功能简单，`AndThenFut` 会假设两个 Future 在创建时就可用了。
// 而真实的`Andthen`允许根据第一个`Future`的输出来创建第二个`Future`，因此复杂的多。
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}

impl<FutureA, FutureB> SimpleFuture for AndThenFut<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self, wake: fn() -> Poll<Self::Output>) {
        if let Some(first) = &mut self.first {
            match first.poll(wake) {
                // 我们已经完成了第一个 Future，可以将它移除，然后准备开始运行第二个
                Poll::Ready(()) => self.first.take(),
                // 第一个 Future 还不能完成
                Poll::Pending => return Poll::Pending,
            };
        }

        // 运行到这里，说明第一个Future已经完成，尝试去完成第二个
        self.second.poll(wake)
    }
}
```

这些例子展示了在不需要内存对象分配以及深层嵌套回调的情况下，该如何使用 Future 特征去表达异步控制流。在了解了基础的控制流后，我们再来看看真实的 Future 特征有何不同之处。

```
trait Future {
    type Output;
    fn poll(
        // 首先值得注意的地方是，`self`的类型从`&mut self`变成了`Pin<&mut Self>`：
        self: Pin<&mut Self>,
        // 其次将`wake: fn()` 修改为 `cx: &mut Context<'_>`：
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

首先这里多了一个 Pin，关于它我们会在后面章节详细介绍，现在你只需要知道使用它可以创建一个无法被移动的 Future，因为无法被移动，所以它将具有固定的内存地址，意味着我们可以存储它的指针

(如果内存地址可能会变动，那存储指针地址将毫无意义！），也意味着可以实现一个自引用数据结构：  
`struct MyFut { a: i32, ptr_to_a: *const i32 }`。而对于 `async/await` 来说，`Pin` 是不可或缺的关键特性。

其次，从 `wake: fn()` 变成了 `&mut Context<'_>`。意味着 `wake` 函数可以携带数据了，为何要携带数据？考虑一个真实世界的场景，一个复杂应用例如 web 服务器可能有数千连接同时在线，那么同时就有数千 `Future` 在被同时管理着，如果不能携带数据，当一个 `Future` 调用 `wake` 后，执行器该如何知道是哪个 `Future` 调用了 `wake`，然后进一步去 `poll` 对应的 `Future`？没有办法！那之前的例子为啥就可以使用没有携带数据的 `wake`？因为足够简单，不存在歧义性。

总之，在正式场景要进行 `wake`，就必须携带上数据。而 `Context` 类型通过提供一个 `Waker` 类型的值，就可以用来唤醒特定的任务。

## 使用 Waker 来唤醒任务

对于 `Future` 来说，第一次被 `poll` 时无法完成任务是很正常的。但它需要确保在未来一旦准备好时，可以通知执行器再次对其进行 `poll` 进而继续往下执行，该通知就是通过 `Waker` 类型完成的。

`Waker` 提供了一个 `wake()` 方法可以用于告诉执行器：相关的任务可以被唤醒了，此时执行器就可以对相应的 `Future` 再次进行 `poll` 操作。

### 构建一个定时器

下面一起来实现一个简单的定时器 `Future`。为了让例子尽量简单，当计时器创建时，我们会启动一个线程接着让该线程进入睡眠，等睡眠结束后再通知给 `Future`。

注意本例子还会在后面继续使用，因此我们重新创建一个工程来演示：使用 `cargo new --lib timer_future` 来创建一个新工程，在 `lib` 包的根路径 `src/lib.rs` 中添加以下内容：

```
use std::{
    future::Future,
    pin::Pin,
    sync::{Arc, Mutex},
    task::{Context, Poll, Waker},
    thread,
    time::Duration,
};
```

继续来实现 `Future` 定时器，之前提到：新建线程在睡眠结束后会需要将状态同步给定时器 `Future`，由于是多线程环境，我们需要使用 `Arc<Mutex<T>>` 来作为一个共享状态，用于在新线程和 `Future` 定时器间共享。

```

pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

/// 在Future和等待的线程间共享状态
struct SharedState {
    /// 定时(睡眠)是否结束
    completed: bool,

    /// 当睡眠结束后, 线程可以用`waker`通知`TimerFuture`来唤醒任务
    waker: Option<Waker>,
}

```

下面给出 Future 的具体实现:

```

impl Future for TimerFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        // 通过检查共享状态, 来确定定时器是否已经完成
        let mut shared_state = self.shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            // 设置`waker`, 这样新线程在睡眠(计时)结束后可以唤醒当前的任务, 接着再次对`Future`进行`poll`操作,
            //
            // 下面的`clone`每次被`poll`时都会发生一次, 实际上, 应该是只`clone`一次更加合理。
            // 选择每次都`clone`的原因是: `TimerFuture`可以在执行器的不同任务间移动, 如果只克隆一次,
            // 那么获取到的`waker`可能已经被篡改并指向了其它任务, 最终导致执行器运行了错误的任务
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}

```

代码很简单, 只要新线程设置了 `shared_state.completed = true`, 那任务就能顺利结束。如果没有设置, 会为当前的任务克隆一份 Waker, 这样新线程就可以使用它来唤醒当前的任务。

最后, 再来创建一个 API 用于构建定时器和启动计时线程:

```

impl TimerFuture {
    /// 创建一个新的`TimerFuture`，在指定的时间结束后，该`Future`可以完成
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));
        // 创建新线程
        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            // 睡眠指定时间实现计时功能
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            // 通知执行器定时器已经完成，可以继续`poll`对应的`Future`了
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });
        TimerFuture { shared_state }
    }
}

```

至此，一个简单的定时器 Future 就已创建成功，那么该如何使用它呢？相信部分爱动脑筋的读者已经猜到了：我们需要创建一个执行器，才能让程序动起来。

## 执行器 Executor

Rust 的 Future 是惰性的：只有屁股上拍一拍，它才会努力动一动。其中一个推动它的方式就是在 `async` 函数中使用 `.await` 来调用另一个 `async` 函数，但是这个只能解决 `async` 内部的问题，那么这些最外层的 `async` 函数，谁来推动它们运行呢？答案就是我们之前多次提到的执行器 `executor`。

执行器会管理一批 Future (最外层的 `async` 函数)，然后通过不停地 `poll` 推动它们直到完成。最开始，执行器会先 `poll` 一次 Future，后面就不会主动去 `poll` 了，而是等待 Future 通过调用 `wake` 函数来通知它可以继续，它才会继续去 `poll`。这种 **wake 通知然后 poll** 的方式会不断重复，直到 Future 完成。

### 构建执行器

下面我们将实现一个简单的执行器，它可以同时并发运行多个 Future。例子中，需要用到 `futures` 包的 `ArcWake` 特征，它可以提供一个方便的途径去构建一个 `Waker`。编辑 `Cargo.toml`，添加下面依赖：

```
[dependencies]
futures = "0.3"
```

在之前的内容中，我们在 `src/lib.rs` 中创建了定时器 `Future`，现在在 `src/main.rs` 中来创建程序的主体内容，开始之前，先引入所需的包：

```
use {
    futures::{
        future::{BoxFuture, FutureExt},
        task::{waker_ref, ArcWake},
    },
    std::{
        future::Future,
        sync::mpsc::{sync_channel, Receiver, SyncSender},
        sync::{Arc, Mutex},
        task::{Context, Poll},
        time::Duration,
    },
    // 引入之前实现的定时器模块
    timer_future::TimerFuture,
};
```

执行器需要从一个消息通道(`channel`)中拉取事件，然后运行它们。当一个任务准备好后(可以继续执行)，它会将自己放入消息通道中，然后等待执行器 `poll`。

```

/// 任务执行器，负责从通道中接收任务然后执行
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

/// `Spawner` 负责创建新的`Future`然后将它发送到任务通道中
#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

/// 一个Future，它可以调度自己(将自己放入任务通道中)，然后等待执行器去`poll`、
struct Task {
    /// 进行中的Future，在未来的某个时间点会被完成
    ///
    /// 按理来说`Mutex`在这里是多余的，因为我们只有一个线程来执行任务。但是由于
    /// Rust并不聪明，它无法知道`Future`只会在一个线程内被修改，并不会被跨线程修改。因此
    /// 我们需要使用`Mutex`来满足这个笨笨的编译器对线程安全的执着。
    ///
    /// 如果是生产级的执行器实现，不会使用`Mutex`，因为会带来性能上的开销，取而代之的是使用
    `UnsafeCell`、
    future: Mutex<Option<BoxFuture<'static, ()>>>,
}

/// 可以将该任务自身放回到任务通道中，等待执行器的poll
task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    // 任务通道允许的最大缓冲数(任务队列的最大长度)
    // 当前的实现仅仅是为了简单，在实际的执行中，并不会这么使用
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender })
}

```

下面再来添加一个方法用于生成 Future，然后将它放入任务通道中：

```

impl Spawner {
    fn spawn(&self, future: impl Future<Output = ()> + 'static + Send) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender.send(task).expect("任务队列已满");
    }
}

```

在执行器 poll 一个 Future 之前，首先需要调用 wake 方法进行唤醒，然后再由 Waker 负责调度该任务并将其放入任务通道中。创建 Waker 的最简单的方式就是实现 ArcWake 特征，先来为我们的任务实现 ArcWake 特征，这样它们就能被转变成 Waker 然后被唤醒：

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        // 通过发送任务到任务管道的方式来实现`wake`，这样`wake`后，任务就能被执行器`poll`、
        let cloned = arc_self.clone();
        arc_self
            .task_sender
            .send(cloned)
            .expect("任务队列已满");
    }
}
```

当任务实现了 ArcWake 特征后，它就变成了 Waker，在调用 wake() 对其唤醒后会将任务复制一份所有权( Arc )，然后将其发送到任务通道中。最后我们的执行器将从通道中获取任务，然后进行 poll 执行：

```
impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.recv() {
            // 获取一个future，若它还没有完成(仍然是Some，不是None)，则对它进行一次poll并尝试完
            成它
            let mut future_slot = task.future.lock().unwrap();
            if let Some(mut future) = future_slot.take() {
                // 基于任务自身创建一个 `LocalWaker`、
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                // `BoxFuture<T>`是`Pin<Box<dyn Future<Output = T> + Send +
                'static>>`的类型别名
                // 通过调用`as_mut`方法，可以将上面的类型转换成`Pin<&mut dyn Future + Send
                + 'static>
                if future.as_mut().poll(context).is_pending() {
                    // Future还没执行完，因此将它放回任务中，等待下次被poll
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

恭喜！我们终于拥有了自己的执行器，下面再来写一段代码使用该执行器去运行之前的定时器 Future：  
：

```

fn main() {
    let (executor, spawner) = new_executor_and_spawner();

    // 生成一个任务
    spawner.spawn(async {
        println!("howdy!");
        // 创建定时器Future，并等待它完成
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("done!");
    });

    // drop掉任务，这样执行器就知道任务已经完成，不会再有新的任务进来
    drop(spawner);

    // 运行执行器直到任务队列为空
    // 任务运行后，会先打印`howdy!`，暂停2秒，接着打印 `done!`
    executor.run();
}

```

## 执行器和系统 IO

前面我们一起看过一个使用 Future 从 Socket 中异步读取数据的例子：

```

pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl SimpleFuture for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(&mut self, wake: fn() -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            // socket有数据，写入buffer中并返回
            Poll::Ready(self.socket.read_buf())
        } else {
            // socket中还没数据
            //
            // 注册一个`wake`函数，当数据可用时，该函数会被调用，
            // 然后当前Future的执行器会再次调用`poll`方法，此时就可以读取到数据
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}

```

该例子中，`Future` 将从 `Socket` 读取数据，若当前还没有数据，则会让出当前线程的所有权，允许执行器去执行其它的 `Future`。当数据准备好后，会调用 `wake()` 函数将该 `Future` 的任务放入任务通道中，等待执行器的 `poll`。

关于该流程已经反复讲了很多次，相信大家应该非常清楚了。然而该例子中还有一个疑问没有解决：

- `set_readable_callback` 方法到底是怎么工作的？怎么才能知道 `socket` 中的数据已经可以被读取了？

关于第二点，其中一个简单粗暴的方法就是使用一个新线程不停的检查 `socket` 中是否有了数据，当有了后，就调用 `wake()` 函数。该方法确实可以满足需求，但是性能着实太低了，需要为每个阻塞的 `Future` 都创建一个单独的线程！

在现实世界中，该问题往往是通过操作系统提供的 IO 多路复用机制来完成，例如 Linux 中的 `epoll`，FreeBSD 和 macOS 中的 `kqueue`，Windows 中的 `IOCP`，Fuchsia 中的 `ports` 等(可以通过 Rust 的跨平台包 `mio` 来使用它们)。借助 IO 多路复用机制，可以实现一个线程同时阻塞地去等待多个异步 IO 事件，一旦某个事件完成就立即退出阻塞并返回数据。相关实现类似于以下代码：

```

struct IoBlocker {
    /* ... */
}

struct Event {
    // Event的唯一ID，该事件发生后，就会被监听起来
    id: usize,

    // 一组需要等待或者已发生的信号
    signals: Signals,
}

impl IoBlocker {
    /// 创建需要阻塞等待的异步IO事件的集合
    fn new() -> Self { /* ... */ }

    /// 对指定的IO事件表示兴趣
    fn add_io_event_interest(
        &self,
        /// 事件所绑定的socket
        io_object: &IoObject,
        event: Event,
    ) { /* ... */ }

    /// 进入阻塞，直到某个事件出现
    fn block(&self) -> Event { /* ... */ }
}

let mut io_blocker = IoBlocker::new();
io_blocker.add_io_event_interest(
    &socket_1,
    Event { id: 1, signals: READABLE },
);
io_blocker.add_io_event_interest(
    &socket_2,
    Event { id: 2, signals: READABLE | WRITABLE },
);
let event = io_blocker.block();

// 当socket的数据可以读取时，打印 "Socket 1 is now READABLE"
println!("Socket {:#?} is now {:#?}", event.id, event.signals);

```

这样，我们只需要一个执行器线程，它会接收 IO 事件并将其分发到对应的 Waker 中，接着后者会唤醒相关的任务，最终通过执行器 poll 后，任务可以顺利地继续执行，这种 IO 读取流程可以不停的循环，直到 socket 关闭。

# 定海神针 Pin 和 Unpin

在 Rust 异步编程中，有一个定海神针般的存在，它就是 `Pin`，作用说简单也简单，说复杂也非常复杂，当初刚出来时就连一些 Rust 大佬都一头雾水，何况瑟瑟发抖的我。好在今非昔比，目前网上的资料已经很全，而我就借花献佛，给大家好好讲讲这个 `Pin`。

在 Rust 中，所有的类型可以分为两类：

- **类型的值可以在内存中安全地被移动**，例如数值、字符串、布尔值、结构体、枚举，总之你能想到的几乎所有类型都可以落入到此范畴内
- **自引用类型**，大魔王来了，大家快跑，在之前章节我们已经见识过它的厉害

下面就是一个自引用类型

```
struct SelfRef {  
    value: String,  
    pointer_to_value: *mut String,  
}
```

在上面的结构体中，`pointer_to_value` 是一个裸指针，指向第一个字段 `value` 持有的字符串 `String`。很简单对吧？现在考虑一个情况，若 `String` 被移动了怎么办？

此时一个致命的问题就出现了：新的字符串的内存地址变了，而 `pointer_to_value` 依然指向之前的地址，一个重大 bug 就出现了！

灾难发生，英雄在哪？只见 `Pin` 闪亮登场，它可以防止一个类型在内存中被移动。再回忆下之前在 `Future` 章节中，我们提到过在 `poll` 方法的签名中有一个 `self: Pin<&mut Self>`，那么为何要在这里使用 `Pin` 呢？

## 为何需要 Pin

其实 `Pin` 还有一个小伙伴 `UnPin`，与前者相反，后者表示类型可以在内存中安全地移动。在深入之前，我们先来回忆下 `async/.await` 是如何工作的：

```
let fut_one = /* ... */; // Future 1  
let fut_two = /* ... */; // Future 2  
async move {  
    fut_one.await;  
    fut_two.await;  
}
```

在底层，`async` 会创建一个实现了 `Future` 的匿名类型，并提供了一个 `poll` 方法：

```
// `async { ... }` 语句块创建的 `Future` 类型
struct AsyncFuture {
    fut_one: FutOne,
    fut_two: FutTwo,
    state: State,
}

// `async` 语句块可能处于的状态
enum State {
    AwaitingFutOne,
    AwaitingFutTwo,
    Done,
}

impl Future for AsyncFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        loop {
            match self.state {
                State::AwaitingFutOne => match self.fut_one.poll(..) {
                    Poll::Ready(()) => self.state = State::AwaitingFutTwo,
                    Poll::Pending => return Poll::Pending,
                }
                State::AwaitingFutTwo => match self.fut_two.poll(..) {
                    Poll::Ready(()) => self.state = State::Done,
                    Poll::Pending => return Poll::Pending,
                }
                State::Done => return Poll::Ready(()),
            }
        }
    }
}
```

当 `poll` 第一次被调用时，它会去查询 `fut\_one` 的状态，若 `fut\_one` 无法完成，则 `poll` 方法会返回。未来对 `poll` 的调用将从上一次调用结束的地方开始。该过程会一直持续，直到 `Future` 完成为止。

然而，如果我们的 `async` 语句块中使用了引用类型，会发生什么？例如下面例子：

```
async {
    let mut x = [0; 128];
    let read_into_buf_fut = read_into_buf(&mut x);
    read_into_buf_fut.await;
    println!("{:?}", x);
}
```

这段代码会编译成下面的形式：

```
struct ReadIntoBuf<'a> {
    buf: &'a mut [u8], // 指向下面的`x`字段
}

struct AsyncFuture {
    x: [u8; 128],
    read_into_buf_fut: ReadIntoBuf<'what_lifetime?>,
}
```

这里，`ReadIntoBuf` 拥有一个引用字段，指向了结构体的另一个字段 `x`，一旦 `AsyncFuture` 被移动，那 `x` 的地址也将随之变化，此时对 `x` 的引用就变成了不合法的，也就是 `read_into_buf_fut.buf` 会变为不合法的。

若能将 `Future` 在内存中固定到一个位置，就可以避免这种问题的发生，也就可以安全的创建上面这种引用类型。

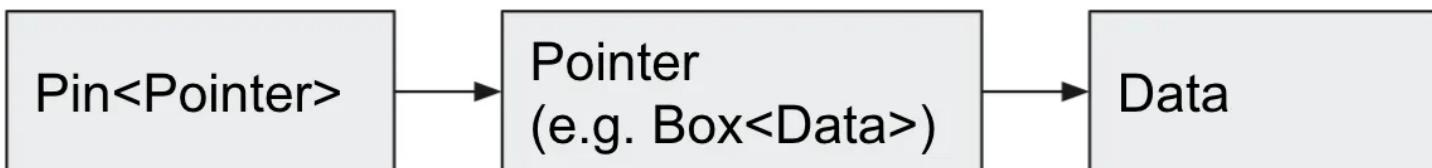
## Unpin

事实上，绝大多数类型都不在意是否被移动(开篇提到的第一种类型)，因此它们都**自动实现了** `Unpin` 特征。

从名字推测，大家可能以为 `Pin` 和 `Unpin` 都是特征吧？实际上，`Pin` 不按套路出牌，它是一个结构体：

```
pub struct Pin<P> {
    pointer: P,
}
```

它包裹一个指针，并且能确保该指针指向的数据不会被移动，例如 `Pin<&mut T>`，`Pin<&T>`，`Pin<Box<T>>`，都能确保 `T` 不会被移动。



而 `Unpin` 才是一个特征，它表明一个类型可以随意被移动，那么问题来了，可以被 `Pin` 住的值，它有没有实现什么特征呢？答案很出乎意料，可以被 `Pin` 住的值实现的特征是 `!Unpin`，大家可能之前没有

见过，但是它其实很简单，`!` 代表没有实现某个特征的意思，`!Unpin` 说明类型没有实现 `Unpin` 特征，那自然就可以被 `Pin` 了。

那是不是意味着类型如果实现了 `Unpin` 特征，就不能被 `Pin` 了？其实，还是可以 `Pin` 的，毕竟它只是一个结构体，你可以随意使用，**但是不再有任何效果而已，该值一样可以被移动！**

例如 `Pin<&mut u8>`，显然 `u8` 实现了 `Unpin` 特征，它可以在内存中被移动，因此 `Pin<&mut u8>` 跟 `&mut u8` 实际上并无区别，一样可以被移动。

因此，一个类型如果不能被移动，它必须实现 `!Unpin` 特征。如果大家对 `Pin`、`Unpin` 还是模模糊糊，建议再重复看一遍之前的内容，理解它们对于我们后面要讲到的内容非常重要！

如果将 `Unpin` 与之前章节学过的 `Send`/`Sync` 进行下对比，会发现它们都很像：

- 都是标记特征( marker trait )，该特征未定义任何行为，非常适用于标记
- 都可以通过`!`语法去除实现
- 绝大多数情况都是自动实现，无需我们的操心

## 深入理解 `Pin`

对于上面的问题，我们可以简单的归结为如何在 Rust 中处理自引用类型(果然，只要是难点，都和自引用脱离不了关系)，下面用一个稍微简单点的例子来理解下 `Pin`：

```

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.a;
        self.b = self_ref;
    }

    fn a(&self) -> &str {
        &self.a
    }

    fn b(&self) -> &String {
        assert!(!self.b.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.b) }
    }
}

```

Test 提供了方法用于获取字段 a 和 b 的值的引用。这里 b 是 a 的一个引用，但是我们并没有使用引用类型而是用了裸指针，原因是：Rust 的借用规则不允许我们这样用，因为不符合生命周期的要求。此时的 Test 就是一个自引用结构体。

如果不移动任何值，那么上面的例子将没有任何问题，例如：

```

fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    println!("a: {}, b: {}", test2.a(), test2.b());
}

```

输出非常正常：

```
a: test1, b: test1
a: test2, b: test2
```

明知山有虎，偏向虎山行，这才是我辈年轻人的风华。既然移动数据会导致指针不合法，那我们就移动下数据试试，将 `test1` 和 `test2` 进行下交换：

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    println!("a: {}, b: {}", test2.a(), test2.b());

}
```

按理来说，这样修改后，输出应该如下：

```
a: test1, b: test1
a: test1, b: test1
```

但是实际运行后，却产生了下面的输出：

```
a: test1, b: test1
a: test1, b: test2
```

原因是 `test2.b` 指针依然指向了旧的地址，而该地址对应的值现在在 `test1` 里，最终会打印出意料之外的值。

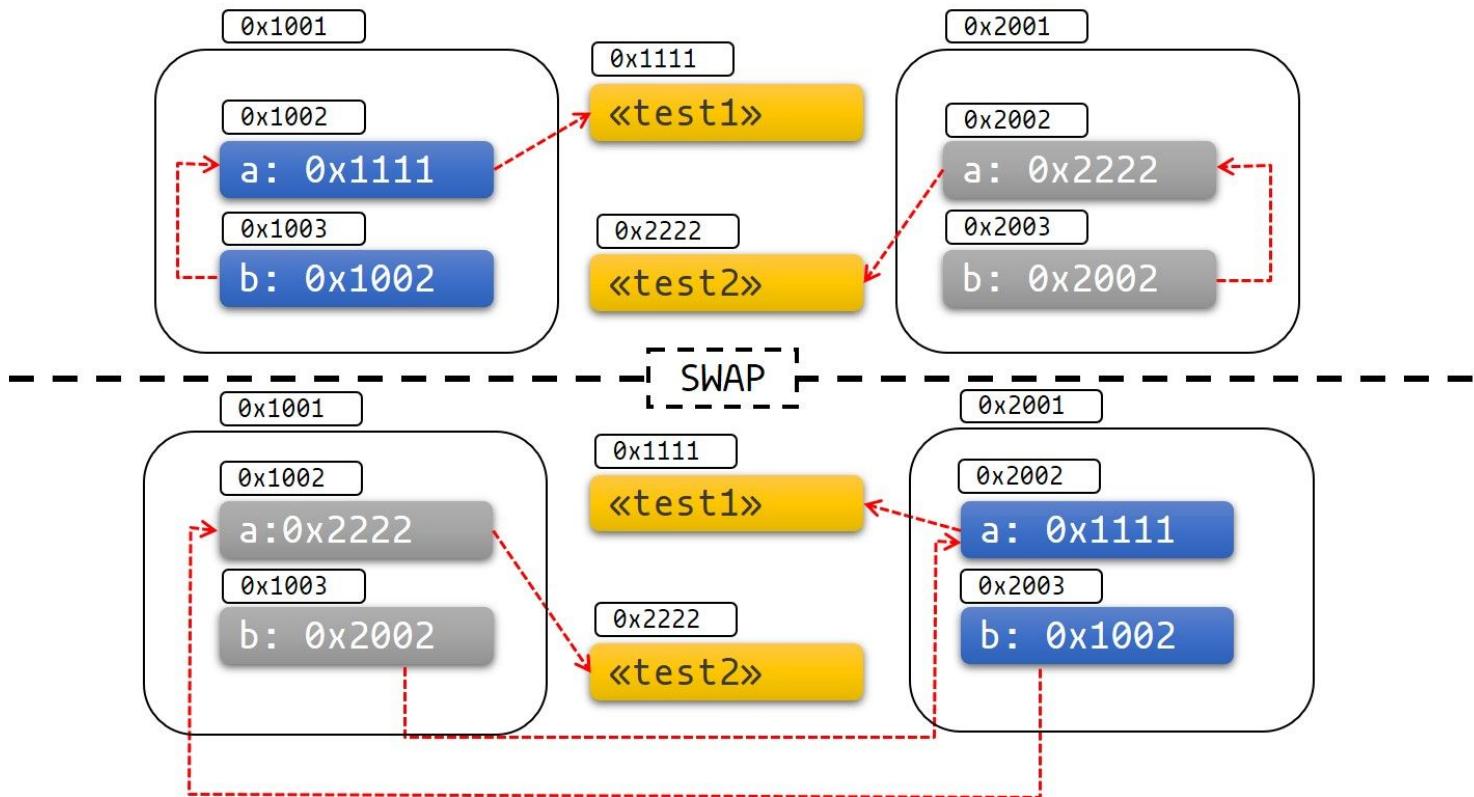
如果大家还是将信将疑，那再看看下面的代码：

```
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    std::mem::swap(&mut test1, &mut test2);
    test1.a = "I've totally changed now!".to_string();
    println!("a: {}, b: {}", test2.a(), test2.b());

}
```

下面的图片也可以帮助更好的理解这个过程：



## Pin 在实践中的运用

在理解了 Pin 的作用后，我们再来看看它怎么帮我们解决问题。

### 将值固定到栈上

回到之前的例子，我们可以用 Pin 来解决指针指向的数据被移动的问题：

```

use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned, // 这个标记可以让我们的类型自动实现特征`!Unpin`
        }
    }

    fn init(self: Pin<&mut Self>) {
        let self_ptr: *const String = &self.a;
        let this = unsafe { self.get_unchecked_mut() };
        this.b = self_ptr;
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        assert!(!self.b.is_null(), "Test::b called without Test::init being called first");
        unsafe { &*(self.b) }
    }
}

```

上面代码中，我们使用了一个标记类型 `PhantomPinned` 将自定义结构体 `Test` 变成了 `!Unpin`（编译器会自动帮我们实现），因此该结构体无法再被移动。

一旦类型实现了 `!Unpin`，那将它的值固定到栈(`stack`)上就是不安全的行为，因此在代码中我们使用了 `unsafe` 语句块来进行处理，你也可以使用 `pin_utils` 来避免 `unsafe` 的使用。

---

BTW, Rust 中的 `unsafe` 其实没有那么可怕，虽然听上去很不安全，但是实际上 Rust 依然提供了很多机制来帮我们提升了安全性，因此不必像对待 Go 语言的 `unsafe` 那样去畏惧于使用 Rust 中的 `unsafe`，大致使用原则总结如下：没必要用时，就不要用，当有必要用时，就大胆用，但是尽量控制好边界，让 `unsafe` 的范围尽可能小

---

此时，再去尝试移动被固定的值，就会导致**编译错误**：

```
pub fn main() {
    // 此时的`test1`可以被安全的移动
    let mut test1 = Test::new("test1");
    // 新的`test1`由于使用了`Pin`，因此无法再被移动，这里的声明会将之前的`test1`遮蔽掉(shadow)
    let mut test1 = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1.as_mut());

    let mut test2 = Test::new("test2");
    let mut test2 = unsafe { Pin::new_unchecked(&mut test2) };
    Test::init(test2.as_mut());

    println!("a: {}, b: {}", Test::a(test1.as_ref()), Test::b(test1.as_ref()));
    std::mem::swap(test1.get_mut(), test2.get_mut());
    println!("a: {}, b: {}", Test::a(test2.as_ref()), Test::b(test2.as_ref()));
}
```

注意到之前的粗体字了吗？是的，Rust 并不是在运行时做这件事，而是在编译期就完成了，因此没有额外的性能开销！来看看报错：

```
error[E0277]: `PhantomPinned` cannot be unpinned
--> src/main.rs:47:43
  |
47 |     std::mem::swap(test1.get_mut(), test2.get_mut());
  |     ^^^^^^^^ within `Test`, the trait
`Unpin` is not implemented for `PhantomPinned`
```

---

需要注意的是固定在栈上非常依赖于你写出的 `unsafe` 代码的正确性。我们知道 `&'a mut T` 可以固定的生命周期是 `'a`，但是我们却不知道当生命周期 `'a` 结束后，该指针指向的数据是否会被移走。如果你的 `unsafe` 代码里这么实现了，那么就会违背 `Pin` 应该具有的作用！

一个常见的错误就是忘记去**遮蔽( shadow )**初始的变量，因为你可以 `drop` 掉 `Pin`，然后在 `&'a mut T` 结束后去移动数据：

```
fn main() {
    let mut test1 = Test::new("test1");
    let mut test1_pin = unsafe { Pin::new_unchecked(&mut test1) };
    Test::init(test1_pin.as_mut());

    drop(test1_pin);
    println!(r#"test1.b points to "test1": {:?}..."#, test1.b);

    let mut test2 = Test::new("test2");
    mem::swap(&mut test1, &mut test2);
    println!("... and now it points nowhere: {:?}", test1.b);
}
```

---

## 固定到堆上

将一个 `!Unpin` 类型的值固定到堆上，会给予该值一个稳定的内存地址，它指向的堆中的值在 `Pin` 后是无法被移动的。而且与固定在栈上不同，我们知道堆上的值在整个生命周期内都会被稳稳地固定住。

```

use std::pin::Pin;
use std::marker::PhantomPinned;

#[derive(Debug)]
struct Test {
    a: String,
    b: *const String,
    _marker: PhantomPinned,
}

impl Test {
    fn new(txt: &str) -> Pin<Box<Self>> {
        let t = Test {
            a: String::from(txt),
            b: std::ptr::null(),
            _marker: PhantomPinned,
        };
        let mut boxed = Box::pin(t);
        let self_ptr: *const String = &boxed.as_ref().a;
        unsafe { boxed.as_mut().get_unchecked_mut().b = self_ptr };

        boxed
    }

    fn a(self: Pin<&Self>) -> &str {
        &self.get_ref().a
    }

    fn b(self: Pin<&Self>) -> &String {
        unsafe { &*(self.b) }
    }
}

pub fn main() {
    let test1 = Test::new("test1");
    let test2 = Test::new("test2");

    println!("a: {}, b: {}", test1.as_ref().a(), test1.as_ref().b());
    println!("a: {}, b: {}", test2.as_ref().a(), test2.as_ref().b());
}

```

## 将固定住的 Future 变为 Unpin

之前的章节我们有提到 `async` 函数返回的 `Future` 默认就是 `!Unpin` 的。

但是，在实际应用中，一些函数会要求它们处理的 `Future` 是 `Unpin` 的，此时，若你使用的 `Future` 是 `!Unpin` 的，必须要使用以下的方法先将 `Future` 进行固定：

- `Box::pin`，创建一个 `Pin<Box<T>>`

- `pin_utils::pin_mut!` , 创建一个 `Pin<&mut T>`

固定后获得的 `Pin<Box<T>>` 和 `Pin<&mut T>` 既可以用于 `Future` , 又会自动实现 `Unpin`。

```
use pin_utils::pin_mut; // `pin_utils` 可以在crates.io中找到

// 函数的参数是一个`Future`，但是要求该`Future`实现`Unpin`
fn execute_unpin_future(x: impl Future<Output = ()> + Unpin) { /* ... */ }

let fut = async { /* ... */ };
// 下面代码报错：默认情况下，`fut` 实现的是`!Unpin`，并没有实现`Unpin`
// execute_unpin_future(fut);

// 使用`Box`进行固定
let fut = async { /* ... */ };
let fut = Box::pin(fut);
execute_unpin_future(fut); // OK

// 使用`pin_mut!`进行固定
let fut = async { /* ... */ };
pin_mut!(fut);
execute_unpin_future(fut); // OK
```

## 总结

相信大家看到这里，脑袋里已经快被 `Pin` 、 `Unpin` 、 `!Unpin` 整爆炸了，没事，我们再来火上浇油下:)

- 若 `T: Unpin` (Rust 类型的默认实现), 那么 `Pin<'a, T>` 跟 `&'a mut T` 完全相同, 也就是 `Pin` 将没有任何效果, 该移动还是照常移动
- 绝大多数标准库类型都实现了 `Unpin` , 事实上, 对于 Rust 中你能遇到的绝大多数类型, 该结论依然成立 , 其中一个例外就是: `async/await` 生成的 `Future` 没有实现 `Unpin`
- 你可以通过以下方法为自己的类型添加 `!Unpin` 约束:
  - 使用文中提到的 `std::marker::PhantomPinned`
  - 使用 `nightly` 版本下的 `feature flag`
- 可以将值固定到栈上, 也可以固定到堆上
  - 将 `!Unpin` 值固定到栈上需要使用 `unsafe`
  - 将 `!Unpin` 值固定到堆上无需 `unsafe` , 可以通过 `Box::pin` 来简单的实现
- 当固定类型 `T: !Unpin` 时, 你需要保证数据从被固定到被 `drop` 这段时期内, 其内存不会变得非法或者被重用

# async/await 和 Stream 流处理

在入门章节中，我们简单学习了该如何使用 `async/.await`，同时在后面也了解了一些底层原理，现在是时候继续深入了。

`async/.await` 是 Rust 语法的一部分，它在遇到阻塞操作时(例如 IO)会让出当前线程的所有权而不是阻塞当前线程，这样就允许当前线程继续去执行其它代码，最终实现并发。

有两种方式可以使用 `async`：`async fn` 用于声明函数，`async { ... }` 用于声明语句块，它们会返回一个实现 `Future` 特征的值：

```
// `foo()` 返回一个`Future<Output = u8>`，  
// 当调用`foo().await`时，该`Future`将被运行，当调用结束后我们将获取到一个`u8`值  
async fn foo() -> u8 { 5 }  
  
fn bar() -> impl Future<Output = u8> {  
    // 下面的`async`语句块返回`Future<Output = u8>`  
    async {  
        let x: u8 = foo().await;  
        x + 5  
    }  
}
```

`async` 是懒惰的，直到被执行器 `poll` 或者 `.await` 后才会开始运行，其中后者是最常用的运行 `Future` 的方法。当 `.await` 被调用时，它会尝试运行 `Future` 直到完成，但是若该 `Future` 进入阻塞，那就会让出当前线程的控制权。当 `Future` 后面准备再一次被运行时(例如从 `socket` 中读取到了数据)，执行器会得到通知，并再次运行该 `Future`，如此循环，直到完成。

以上过程只是一个简述，详细内容在[底层探秘](#)中已经被深入讲解过，因此这里不再赘述。

## async 的生命周期

`async fn` 函数如果拥有引用类型的参数，那它返回的 `Future` 的生命周期就会被这些参数的生命周期所限制：

```
async fn foo(x: &u8) -> u8 { *x }  
  
// 上面的函数跟下面的函数是等价的：  
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {  
    async move { *x }  
}
```

意味着 `async fn` 函数返回的 `Future` 必须满足以下条件: 当 `x` 依然有效时, 该 `Future` 就必须继续等待(`.await`), 也就是说 `x` 必须比 `Future` 活得更久。

在一般情况下, 在函数调用后就立即 `.await` 不会存在任何问题, 例如 `foo(&x).await`。但是, 若 `Future` 被先存起来或发送到另一个任务或者线程, 就可能存在问题是:

```
use std::future::Future;
fn bad() -> impl Future<Output = u8> {
    let x = 5;
    borrow_x(&x) // ERROR: `x` does not live long enough
}

async fn borrow_x(x: &u8) -> u8 { *x }
```

以上代码会报错, 因为 `x` 的生命周期只到 `bad` 函数的结尾。但是 `Future` 显然会活得更久:

```
error[E0597]: `x` does not live long enough
--> src/main.rs:4:14
|
4 |     borrow_x(&x) // ERROR: `x` does not live long enough
|-----^--^
|       |
|       borrowed value does not live long enough
|       argument requires that `x` is borrowed for `static`
5 | }
| - `x` dropped here while still borrowed
```

其中一个常用的解决方法就是将具有引用参数的 `async fn` 函数转变成一个具有 `'static` 生命周期的 `Future`。以上解决方法可以通过将参数和对 `async fn` 的调用放在同一个 `async` 语句块来实现:

```
use std::future::Future;

async fn borrow_x(x: &u8) -> u8 { *x }

fn good() -> impl Future<Output = u8> {
    async {
        let x = 5;
        borrow_x(&x).await
    }
}
```

如上所示, 通过将参数移动到 `async` 语句块内, 我们将它的生命周期扩展到 `'static`, 并跟返回的 `Future` 保持了一致。

## async move

`async` 允许我们使用 `move` 关键字来将环境中变量的所有权转移到语句块内，就像闭包那样，好处是你不再发愁该如何解决借用生命周期的问题，坏处就是无法跟其它代码实现对变量的共享：

```
// 多个不同的 `async` 语句块可以访问同一个本地变量，只要它们在该变量的作用域内执行
async fn blocks() {
    let my_string = "foo".to_string();

    let future_one = async {
        // ...
        println!("{}my_string");
    };

    let future_two = async {
        // ...
        println!("{}my_string");
    };

    // 运行两个 Future 直到完成
    let (((), ()),) = futures::join!(future_one, future_two);
}
```

```
// 由于 `async move` 会捕获环境中的变量，因此只有一个 `async move` 语句块可以访问该变量，
// 但是它也有非常明显的好处：变量可以转移到返回的 Future 中，不再受借用生命周期的限制
fn move_block() -> impl Future<Output = ()> {
    let my_string = "foo".to_string();
    async move {
        // ...
        println!("{}my_string");
    }
}
```

## 当`.await` 遇见多线程执行器

需要注意的是，当使用多线程 `Future` 执行器(`executor`)时，`Future` 可能会在线程间被移动，因此 `async` 语句块中的变量必须要能在线程间传递。至于 `Future` 会在线程间移动的原因是：它内部的任何 `.await` 都可能导致它被切换到一个新线程上去执行。

由于需要在多线程环境使用，意味着 `Rc`、`RefCell`、没有实现 `Send` 的所有权类型、没有实现 `Sync` 的引用类型，它们都是不安全的，因此无法被使用

---

需要注意！实际上它们还是有可能被使用的，只要在 `.await` 调用期间，它们没有在作用域范围内。

类似的原因，在 `.await` 时使用普通的锁也不安全，例如 `Mutex`。原因是，它可能会导致线程池被锁：当一个任务获取锁 A 后，若它将线程的控制权还给执行器，然后执行器又调度运行另一个任务，该任务也去尝试获取了锁 A，结果当前线程会直接卡死，最终陷入死锁中。

因此，为了避免这种情况的发生，我们需要使用 `futures` 包下的锁 `futures::lock` 来替代 `Mutex` 完成任务。

## Stream 流处理

`Stream` 特征类似于 `Future` 特征，但是前者在完成前可以生成多个值，这种行为跟标准库中的 `Iterator` 特征倒是颇为相似。

```
trait Stream {
    // Stream生成的值的类型
    type Item;

    // 尝试去解析Stream中的下一个值,
    // 若无数据, 返回`Poll::Pending`，若有数据, 返回 `Poll::Ready(Some(x))`，`Stream`完成则
    // 返回 `Poll::Ready(None)`
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

关于 `Stream` 的一个常见例子是消息通道（`futures` 包中的）的消费者 `Receiver`。每次有消息从 `Send` 端发送后，它都可以接收到一个 `Some(val)` 值，一旦 `Send` 端关闭(`drop`)，且消息通道中没有消息后，它会接收到一个 `None` 值。

```

async fn send_recv() {
    const BUFFER_SIZE: usize = 10;
    let (mut tx, mut rx) = mpsc::channel::<i32>(BUFFER_SIZE);

    tx.send(1).await.unwrap();
    tx.send(2).await.unwrap();
    drop(tx);

    // `StreamExt::next` 类似于 `Iterator::next`，但是前者返回的不是值，而是一个
    `Future<Output = Option<T>>`,
    // 因此还需要使用`.await`来获取具体的值
    assert_eq!(Some(1), rx.next().await);
    assert_eq!(Some(2), rx.next().await);
    assert_eq!(None, rx.next().await);
}

```

## 迭代和并发

跟迭代器类似，我们也可以迭代一个 `Stream`。例如使用 `map`，`filter`，`fold` 方法，以及它们的遇到错误提前返回的版本：`try_map`，`try_filter`，`try_fold`。

但是跟迭代器又有所不同，`for` 循环无法在这里使用，但是命令式风格的循环 `while let` 是可以用的，同时还可以使用 `next` 和 `try_next` 方法：

```

async fn sum_with_next(mut stream: Pin<&mut dyn Stream<Item = i32>>) -> i32 {
    use futures::stream::StreamExt; // 引入 next
    let mut sum = 0;
    while let Some(item) = stream.next().await {
        sum += item;
    }
    sum
}

async fn sum_with_try_next(
    mut stream: Pin<&mut dyn Stream<Item = Result<i32, io::Error>>,
) -> Result<i32, io::Error> {
    use futures::stream::TryStreamExt; // 引入 try_next
    let mut sum = 0;
    while let Some(item) = stream.try_next().await? {
        sum += item;
    }
    Ok(sum)
}

```

上面代码是一次处理一个值的模式，但是需要注意的是：如果你选择一次处理一个值的模式，可能会造成无法并发，这就失去了异步编程的意义。因此，如果可以的话我们还是要选择从一个 `Stream` 并发处理多个值的方式，通过 `for_each_concurrent` 或 `try_for_each_concurrent` 方法来实现：

```
async fn jump_around(
    mut stream: Pin<&mut dyn Stream<Item = Result<u8, io::Error>>>,
) -> Result<(), io::Error> {
    use futures::stream::TryStreamExt; // 引入 `try_for_each_concurrent`
    const MAX_CONCURRENT_JUMPERS: usize = 100;

    stream.try_for_each_concurrent(MAX_CONCURRENT_JUMPERS, |num| async move {
        jump_n_times(num).await?;
        report_n_jumps(num).await?;
        Ok(())
    }).await?;

    Ok(())
}
```

# 使用 `join!` 和 `select!` 同时运行多个 Future

招数单一，杀伤力惊人，说的就是 `.await`，但是光用它，还真做不到一招鲜吃遍天。比如我们该如何同时运行多个任务，而不是使用 `.await` 慢悠悠地排队完成。

## join!

`futures` 包中提供了很多实用的工具，其中一个就是 `join!` 宏，它允许我们同时等待多个不同 `Future` 的完成，且可以并发地运行这些 `Future`。

先来看一个不是很给力的、使用 `.await` 的版本：

```
async fn enjoy_book_and_music() -> (Book, Music) {
    let book = enjoy_book().await;
    let music = enjoy_music().await;
    (book, music)
}
```

这段代码可以顺利运行，但是有一个很大的问题，就是必须先看完书后，才能听音乐。咱们以前，谁又不是那个摇头晃脑爱读书(耳朵里偷偷塞着耳机，听的正 high)的好学生呢？

要支持同时看书和听歌，有些人可能会凭空生成下面代码：

```
// WRONG -- 别这么做
async fn enjoy_book_and_music() -> (Book, Music) {
    let book_future = enjoy_book();
    let music_future = enjoy_music();
    (book_future.await, music_future.await)
}
```

看上去像模像样，嗯，在某些语言中也许可以，但是 Rust 不行。因为在某些语言中，`Future` 一旦创建就开始运行，等到返回的时候，基本就可以同时结束并返回了。但是 Rust 中的 `Future` 是惰性的，直到调用 `.await` 时，才会开始运行。而那两个 `await` 由于在代码中有先后顺序，因此它们是顺序运行的。

为了正确的并发运行两个 `Future`，我们来试试 `futures::join!` 宏：

```
use futures::join;

async fn enjoy_book_and_music() -> (Book, Music) {
    let book_fut = enjoy_book();
    let music_fut = enjoy_music();
    join!(book_fut, music_fut)
}
```

Duang，目标顺利达成。同时 `join!` 会返回一个元组，里面的值是对应的 `Future` 执行结束后输出的值。

---

如果希望同时运行一个数组里的多个异步任务，可以使用 `futures::future::join_all` 方法

---

## try\_join!

由于 `join!` 必须等待它管理的所有 `Future` 完成后才能完成，如果你希望在某一个 `Future` 报错后就立即停止所有 `Future` 的执行，可以使用 `try_join!`，特别是当 `Future` 返回 `Result` 时：

```
use futures::try_join;

async fn get_book() -> Result<Book, String> { /* ... */ Ok(Book) }
async fn get_music() -> Result<Music, String> { /* ... */ Ok(Music) }

async fn get_book_and_music() -> Result<(Book, Music), String> {
    let book_fut = get_book();
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}
```

有一点需要注意，传给 `try_join!` 的所有 `Future` 都必须拥有相同的错误类型。如果错误类型不同，可以考虑使用来自 `futures::future::TryFutureExt` 模块的 `map_err` 和 `err_info` 方法将错误进行转换：

```

use futures::*;

future::TryFutureExt,
try_join,
};

async fn get_book() -> Result<Book, ()> { /* ... */ Ok(Book) }
async fn get_music() -> Result<Music, String> { /* ... */ Ok(Music) }

async fn get_book_and_music() -> Result<(Book, Music), String> {
    let book_fut = get_book().map_err(|()| "Unable to get book".to_string());
    let music_fut = get_music();
    try_join!(book_fut, music_fut)
}

```

`join!` 很好很强大，但是人无完人，工无完工，它有一个很大的问题。

## select!

`join!` 只有等所有 `Future` 结束后，才能集中处理结果，如果你想同时等待多个 `Future`，且任何一个 `Future` 结束后，都可以立即被处理，可以考虑使用 `futures::select!`：

```

use futures::*;

future::FutureExt, // for `fuse()`
pin_mut,
select,
};

async fn task_one() { /* ... */ }
async fn task_two() { /* ... */ }

async fn race_tasks() {
    let t1 = task_one().fuse();
    let t2 = task_two().fuse();

    pin_mut!(t1, t2);

    select! {
        () = t1 => println!("任务1率先完成"),
        () = t2 => println!("任务2率先完成"),
    }
}

```

上面的代码会同时并发地运行 `t1` 和 `t2`，无论两者哪个先完成，都会调用对应的 `println!` 打印相应的输出，然后函数结束且不会等待另一个任务的完成。

但是，在实际项目中，我们往往需要等待多个任务都完成后，再结束，像上面这种其中一个任务结束就立刻结束的场景着实不多。

## default 和 complete

select! 还支持 default 和 complete 分支：

- complete 分支当所有的 Future 和 Stream 完成后才会被执行，它往往配合 loop 使用，loop 用于循环完成所有的 Future
- default 分支，若没有任何 Future 或 Stream 处于 Ready 状态，则该分支会被立即执行

```
use futures::future;
use futures::select;
pub fn main() {
    let mut a_fut = future::ready(4);
    let mut b_fut = future::ready(6);
    let mut total = 0;

    loop {
        select! {
            a = a_fut => total += a,
            b = b_fut => total += b,
            complete => break,
            default => panic!(), // 该分支永远不会运行，因为 `Future` 会先运行，然后是
`complete`
        };
    }
    assert_eq!(total, 10);
}
```

以上代码 default 分支由于最后一个运行，而在它之前 complete 分支已经通过 break 跳出了循环，因此 default 永远不会被执行。

如果你希望 default 也有机会露下脸，可以将 complete 的 break 修改为其它的，例如 `println!("completed!")`，然后再观察下运行结果。

再回到 select 的第一个例子中，里面有一段代码长这样：

```
let t1 = task_one().fuse();
let t2 = task_two().fuse();

pin_mut!(t1, t2);
```

当时没有展开讲，相信大家也有疑惑，下面我们一起看看。

## 跟 Unpin 和 FusedFuture 进行交互

首先，`.fuse()` 方法可以让 `Future` 实现 `FusedFuture` 特征，而 `pin_mut!` 宏会为 `Future` 实现 `Unpin` 特征，这两个特征恰恰是使用 `select` 所必须的：

- `Unpin`，由于 `select` 不会通过拿走所有权的方式使用 `Future`，而是通过可变引用的方式去使用，这样当 `select` 结束后，该 `Future` 若没有被完成，它的所有权还可以继续被其它代码使用。
- `FusedFuture` 的原因跟上面类似，当 `Future` 一旦完成后，那 `select` 就不能再对其进行轮询使用。Fuse 意味着熔断，相当于 `Future` 一旦完成，再次调用 `poll` 会直接返回 `Poll::Pending`。

只有实现了 `FusedFuture`，`select` 才能配合 `loop` 一起使用。假如没有实现，就算一个 `Future` 已经完成了，它依然会被 `select` 不停的轮询执行。

`Stream` 稍有不同，它们使用的特征是 `FusedStream`。通过 `.fuse()` (也可以手动实现)实现了该特征的 `Stream`，对其调用 `.next()` 或 `.try_next()` 方法可以获取实现了 `FusedFuture` 特征的 `Future`：

```
use futures::{
    stream::{Stream, StreamExt, FusedStream},
    select,
};

async fn add_two_streams(
    mut s1: impl Stream<Item = u8> + FusedStream + Unpin,
    mut s2: impl Stream<Item = u8> + FusedStream + Unpin,
) -> u8 {
    let mut total = 0;

    loop {
        let item = select! {
            x = s1.next() => x,
            x = s2.next() => x,
            complete => break,
        };
        if let Some(next_num) = item {
            total += next_num;
        }
    }

    total
}
```

## 在 select 循环中并发

一个很实用但又鲜为人知的函数是 `Fuse::terminated()`，可以使用它构建一个空的 `Future`，空自然没啥用，但是如果它能在后面再被填充呢？

考虑以下场景：当你要在 `select` 循环中运行一个任务，但是该任务却是在 `select` 循环内部创建时，上面的函数就非常好用了。

```
use futures::{
    future::{Fuse, FusedFuture, FutureExt},
    stream::{FusedStream, Stream, StreamExt},
    pin_mut,
    select,
};

async fn get_new_num() -> u8 { /* ... */ }

async fn run_on_new_num(_: u8) { /* ... */ }

async fn run_loop(
    mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
    starting_num: u8,
) {
    let run_on_new_num_fut = run_on_new_num(starting_num).fuse();
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(run_on_new_num_fut, get_new_num_fut);
    loop {
        select! {
            () = interval_timer.select_next_some() => {
                // 定时器已结束，若`get_new_num_fut`没有在运行，就创建一个新的
                if get_new_num_fut.is_terminated() {
                    get_new_num_fut.set(get_new_num().fuse());
                }
            },
            new_num = get_new_num_fut => {
                // 收到新的数字 -- 创建一个新的`run_on_new_num_fut`并丢弃掉旧的
                run_on_new_num_fut.set(run_on_new_num(new_num).fuse());
            },
            // 运行 `run_on_new_num_fut`
            () = run_on_new_num_fut => {},
            // 若所有任务都完成，直接 `panic`，原因是 `interval_timer` 应该连续不断的产生值，而不是结束
            // 后，执行到 `complete` 分支
            complete => panic!("`interval_timer` completed unexpectedly"),
        }
    }
}
```

当某个 `Future` 有多个拷贝都需要同时运行时，可以使用 `FuturesUnordered` 类型。下面的例子跟上个例子大体相似，但是它会将 `run_on_new_num_fut` 的每一个拷贝都运行到完成，而不是像之前那样一旦创建新的就终止旧的。

```

use futures::*;

future:: {Fuse, FusedFuture, FutureExt},
stream:: {FusedStream, FuturesUnordered, Stream, StreamExt},
pin_mut,
select,
};

async fn get_new_num() -> u8 { /* ... */ 5 }

async fn run_on_new_num(_ : u8) -> u8 { /* ... */ 5 }

// 使用从 `get_new_num` 获取的最新数字 来运行 `run_on_new_num`
//
// 每当计时器结束后，`get_new_num` 就会运行一次，它会立即取消当前正在运行的`run_on_new_num`，  

// 并且使用新返回的值来替换
async fn run_loop(
    mut interval_timer: impl Stream<Item = ()> + FusedStream + Unpin,
    starting_num: u8,
) {
    let mut run_on_new_num_futs = FuturesUnordered::new();
    run_on_new_num_futs.push(run_on_new_num(starting_num));
    let get_new_num_fut = Fuse::terminated();
    pin_mut!(get_new_num_fut);
    loop {
        select! {
            () = interval_timer.select_next_some() => {
                // 定时器已结束，若 `get_new_num_fut` 没有在运行，就创建一个新的
                if get_new_num_fut.is_terminated() {
                    get_new_num_fut.set(get_new_num().fuse());
                }
            },
            new_num = get_new_num_fut => {
                // 收到新的数字 -- 创建一个新的 `run_on_new_num_fut` (并没有像之前的例子那样
                // 丢弃掉旧值)
                run_on_new_num_futs.push(run_on_new_num(new_num));
            },
            // 运行 `run_on_new_num_futs`，并检查是否有已经完成的
            res = run_on_new_num_futs.select_next_some() => {
                println!("run_on_new_num_fut returned {:?}", res);
            },
            // 若所有任务都完成，直接 `panic`，原因是 `interval_timer` 应该连续不断的产生值，  

            // 而不是结束
            // 后，执行到 `complete` 分支
            complete => panic!("`interval_timer` completed unexpectedly"),
        }
    }
}

```

# 一些疑难问题的解决办法

async 在 Rust 依然比较新，疑难杂症少不了，而它们往往还处于活跃开发状态，短时间内无法被解决，因此才有了本文。下面一起来看看这些问题以及相应的临时解决方案。

## 在 async 语句块中使用 ?

async 语句块和 async fn 最大的区别就是前者无法显式的声明返回值，在大多数时候这都不是问题，但是当配合 ? 一起使用时，问题就有所不同：

```
async fn foo() -> Result<u8, String> {
    Ok(1)
}
async fn bar() -> Result<u8, String> {
    Ok(1)
}
pub fn main() {
    let fut = async {
        foo().await?;
        bar().await?;
        Ok(())
    };
}
```

以上代码编译后会报错：

```
error[E0282]: type annotations needed
--> src/main.rs:14:9
|
11 |     let fut = async {
|         --- consider giving `fut` a type
...
14 |         Ok(1)
|             ^^ cannot infer type for type parameter `E` declared on the enum
`Result`
```

原因在于编译器无法推断出 `Result<T, E>` 中的 `E` 的类型，而且编译器的提示 `consider giving `fut` a type` 你也别傻乎乎的相信，然后尝试半天，最后无奈放弃：目前还没有办法为 `async` 语句块指定返回类型。

既然编译器无法推断出类型，那咱就给它更多提示，可以使用 `::< ... >` 的方式来增加类型注释：

```
let fut = async {
    foo().await?;
    bar().await?;
    Ok::<(), String>(())
        // 在这一行进行显式的类型注释
};
```

给予类型注释后此时编译器就知道 `Result<T, E>` 中的 `E` 的类型是 `String`，进而成功通过编译。

## async 函数和 Send 特征

在多线程章节我们深入讲过 `Send` 特征对于多线程间数据传递的重要性，对于 `async fn` 也是如此，它返回的 `Future` 能否在线程间传递的关键在于 `.await` 运行过程中，作用域中的变量类型是否是 `Send`。

学到这里，相信大家已经很清楚 `Rc` 无法在多线程环境使用，原因就在于它并未实现 `Send` 特征，那咱就用它来做例子：

```
use std::rc::Rc;

#[derive(Default)]
struct NotSend(Rc<()>);
```

事实上，未实现 `Send` 特征的变量可以出现在 `async fn` 语句块中：

```
async fn bar() {}
async fn foo() {
    NotSend::default();
    bar().await;
}

fn require_send(_: impl Send) {}

fn main() {
    require_send(foo());
}
```

即使上面的 `foo` 返回的 `Future` 是 `Send`，但是在它内部短暂的使用 `NotSend` 依然是安全的，原因在于它的作用域并没有影响到 `.await`，下面来试试声明一个变量，然后让 `.await` 的调用处于变量的作用域中试试：

```
async fn foo() {
    let x = NotSend::default();
    bar().await;
}
```

不出所料，错误如期而至：

```
error: future cannot be sent between threads safely
--> src/main.rs:17:18
17 |     require_send(foo());
|         ^^^^^^ future returned by `foo` is not `Send`
|
= help: within `impl futures::Future<Output = ()>`, the trait `std::marker::Send` is not implemented for `Rc<()>`
note: future is not `Send` as this value is used across an await
--> src/main.rs:11:5
10 |     let x = NotSend::default();
|         - has type `NotSend` which is not `Send`
11 |     bar().await;
|         ^^^^^^^^^^^^ await occurs here, with `x` maybe used later
12 | }
| - `x` is later dropped here
```

提示很清晰，`.await` 在运行时处于 `x` 的作用域内。在之前章节有提到过，`.await` 有可能被执行器调度到另一个线程上运行，而 `Rc` 并没有实现 `Send`，因此编译器无情拒绝了咱们。

其中一个可能的解决方法是在 `.await` 之前就使用 `std::mem::drop` 释放掉 `Rc`，但是很可惜，截止今天，该方法依然不能解决这种问题。

不知道有多少同学还记得语句块 `{ ... }` 在 Rust 中其实具有非常重要的作用(特别是相比其它大多数语言来说时)：可以将变量声明在语句块内，当语句块结束时，变量会自动被 `Drop`，这个规则可以帮助我们解决很多借用冲突问题，特别是在 `NLL` 出来之前。

```
async fn foo() {
{
    let x = NotSend::default();
}
bar().await;
}
```

是不是很简单？最终我们还是通过 `Drop` 的方式解决了这个问题，当然，还是期待未来 `std::mem::drop` 也能派上用场。

## 递归使用 async fn

在内部实现中，`async fn` 被编译成一个状态机，这会导致递归使用 `async fn` 变得较为复杂，因为编译后的状态机还需要包含自身。

```
// foo函数:  
async fn foo() {  
    step_one().await;  
    step_two().await;  
}  
// 会被编译成类似下面的类型:  
enum Foo {  
    First(StepOne),  
    Second(StepTwo),  
}  
  
// 因此 recursive 函数  
async fn recursive() {  
    recursive().await;  
    recursive().await;  
}  
  
// 会生成类似以下的类型  
enum Recursive {  
    First(Recursive),  
    Second(Recursive),  
}
```

这是典型的[动态大小类型](#)，它的大小会无限增长，因此编译器会直接报错：

```
error[E0733]: recursion in an `async fn` requires boxing  
--> src/lib.rs:1:22  
|  
1 | async fn recursive() {  
|           ^ an `async fn` cannot invoke itself directly  
|  
= note: a recursive `async fn` must be rewritten to return a boxed future.
```

如果认真学习过之前的章节，大家应该知道只要将其使用 `Box` 放到堆上而不是栈上，就可以解决，在这里还是要称赞下 Rust 的编译器，给出的提示总是这么精确 `recursion in an `async fn` requires boxing`。

就算是使用 `Box`，这里也大有讲究。如果我们试图使用 `Box::pin` 这种方式去包裹是不行的，因为编译器自身的限制限制了我们(刚夸过它。。。)。为了解决这种问题，我们只能将 `recursive` 转变成一个正常的函数，该函数返回一个使用 `Box` 包裹的 `async` 语句块：

```
use futures::future::{BoxFuture, FutureExt};

fn recursive() -> BoxFuture<'static, ()> {
    async move {
        recursive().await;
        recursive().await;
    }.boxed()
}
```

## 在特征中使用 `async`

在目前版本中，我们还无法在特征中定义 `async fn` 函数，不过大家也不用担心，目前已经有计划在未来移除这个限制了。

```
trait Test {
    async fn test();
}
```

运行后报错：

```
error[E0706]: functions in traits cannot be declared `async`
--> src/main.rs:5:5
|
5 |     async fn test();  
|-----^~~~~~  
|  
|     `async` because of this
|
= note: `async` trait functions are not currently supported
= note: consider using the `async-trait` crate: https://crates.io/crates/async-trait
```

好在编译器给出了提示，让我们使用 `async-trait` 解决这个问题：

```
use async_trait::async_trait;

#[async_trait]
trait Advertisement {
    async fn run(&self);
}

struct Modal;

#[async_trait]
impl Advertisement for Modal {
    async fn run(&self) {
        self.render_fullscreen().await;
        for _ in 0..4u16 {
            remind_user_to_join_mailing_list().await;
        }
        self.hide_for_now().await;
    }
}

struct AutoplayingVideo {
    media_url: String,
}

#[async_trait]
impl Advertisement for AutoplayingVideo {
    async fn run(&self) {
        let stream = connect(&self.media_url).await;
        stream.play().await;

        // 用视频说服用户加入我们的邮件列表
        Modal.run().await;
    }
}
```

不过使用该包并不是免费的，每一次特征中的 `async` 函数被调用时，都会产生一次堆内存分配。对于大多数场景，这个性能开销都可以接受，但是当函数一秒调用几十万、几百万次时，就得小心这块儿代码的性能了！

# 一个实践项目: Web 服务器

知识学得再多，不实际应用也是纸上谈兵，不是忘掉就是废掉，对于技术学习尤为如此。在之前章节中，我们已经学习了 Async Rust 的方方面面，现在来将这些知识融会贯通，最终实现一个并发 Web 服务器。

## 多线程版本的 Web 服务器

在正式开始前，先来看一个单线程版本的 Web 服务器，该例子来源于 [Rust Book](#) 一书。

src/main.rs :

```
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;

fn main() {
    // 监听本地端口 7878，等待 TCP 连接的建立
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    // 阻塞等待请求的进入
    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    // 从连接中顺序读取 1024 字节数据
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    // 处理HTTP协议头，若不符合则返回404和对应的 `html` 文件
    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };
    let contents = fs::read_to_string(filename).unwrap();

    // 将回复内容写入连接缓存中
    let response = format!("{}{}", status_line, contents);
    stream.write_all(response.as_bytes()).unwrap();
    // 使用 flush 将缓存中的内容发送到客户端
    stream.flush().unwrap();
}

hello.html:
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

404.html：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

运行以上代码，并从浏览器访问 127.0.0.1:7878 你将看到一条来自 Ferris 的问候。

在回忆了单线程版本该如何实现后，我们也将进入正题，一起来实现一个基于 `async` 的异步 Web 服务器。

## 运行异步代码

一个 Web 服务器必须要能并发的处理大量来自用户的请求，也就是我们不能在处理完上一个用户的请求后，再处理下一个用户的请求。上面的单线程版本可以修改为多线程甚至于线程池来实现并发处理，但是线程还是太重了，使用 `async` 实现 Web 服务器才是最适合的。

首先将 `handle_connection` 修改为 `async` 实现：

```
async fn handle_connection(mut stream: TcpStream) {
    //<-- snip -->
}
```

该修改会将函数的返回值从 `()` 变成 `Future<Output=()>`，因此直接运行将不再有任何效果，只用通过 `.await` 或执行器的 `poll` 调用后才能获取 `Future` 的结果。

在之前的代码中，我们使用了自己实现的简单的执行器来进行 `.await` 或 `poll`，实际上这只是为了学习原理，在实际项目中，需要选择一个三方的 `async` 运行时来实现相关功能。具体的选择我们将在下一章节进行讲解，现在先选择 `async-std`，该包的最大优点就是跟标准库的 API 类似，相对来说更简单易用。

## 使用 `async-std` 作为异步运行时

下面的例子将演示如何使用一个异步运行时 `async-std` 来让之前的 `async fn` 函数运行起来，该运行时允许使用属性 `#[async_std::main]` 将我们的 `fn main` 函数变成 `async fn main`，这样就可以在 `main` 函数中直接调用其它 `async` 函数，否则你得用之前章节的 `block_on` 方法来让 `main` 去阻塞等待异步函数的完成，但是这种简单粗暴的阻塞等待方式并不灵活。

修改 `Cargo.toml` 添加 `async-std` 包并开启相应的属性：

```
[dependencies]
futures = "0.3"

[dependencies.async-std]
version = "1.6"
features = ["attributes"]
```

下面将 `main` 函数修改为异步的，并在其中调用前面修改的异步版本 `handle_connection`：

```
##[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        // 警告，这里无法并发
        handle_connection(stream).await;
    }
}
```

上面的代码虽然已经是异步的，实际上它还无法并发，原因我们后面会解释，先来模拟一下慢请求：

```

use std::time::Duration;
use async_std::task;

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        task::sleep(Duration::from_secs(5)).await;
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };
    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{}{}", status_line, contents);
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}

```

上面是全新实现的 `handle_connection`，它会在内部睡眠 5 秒，模拟一次用户慢请求，需要注意的是，我们并没有使用 `std::thread::sleep` 进行睡眠，原因是该函数是阻塞的，它会让当前线程陷入睡眠中，导致其它任务无法继续运行！因此我们需要一个睡眠函数 `async_std::task::sleep`，它仅会让当前的任务陷入睡眠，然后该任务会让出线程的控制权，这样线程就可以继续运行其它任务。

因此，光把函数变成 `async` 往往是不够的，还需要将它内部的代码也都变成异步兼容的，阻塞线程绝对是不可行的。

现在运行服务器，并访问 `127.0.0.1:7878/sleep`，你会发现只有在完成第一个用户请求(5 秒后)，才能开始处理第二个用户请求。现在再来看看该如何解决这个问题，让请求并发起来。

## 并发地处理连接

上面代码最大的问题是 `listener.incoming()` 是阻塞的迭代器。当 `listener` 在等待连接时，执行器是无法执行其它 `Future` 的，而且只有在我们处理完已有的连接后，才能接收新的连接。

解决方法是将 `listener.incoming()` 从一个阻塞的迭代器变成一个非阻塞的 `stream`，后者在前面章节有过专门介绍：

```

use async_std::net::TcpListener;
use async_std::net::TcpStream;
use futures::stream::StreamExt;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").await.unwrap();
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None, |tcpstream| async move {
            let tcpstream = tcpstream.unwrap();
            handle_connection(tcpstream).await;
        })
        .await;
}

```

异步版本的 `TcpListener` 为 `listener.incoming()` 实现了 `Stream` 特征，以上修改有两个好处：

- `listener.incoming()` 不再阻塞
- 使用 `for_each_concurrent` 并发地处理从 `Stream` 获取的元素

现在上面的实现的关键在于 `handle_connection` 不能再阻塞：

```

use async_std::prelude::*;

async fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).await.unwrap();

    //<-- snip -->
    stream.write(response.as_bytes()).await.unwrap();
    stream.flush().await.unwrap();
}

```

在将数据读写改造成异步后，现在该函数也彻底变成了异步的版本，因此一次慢请求不再会阻止其它请求的运行。

## 使用多线程并行处理请求

聪明的读者不知道有没有发现，之前的例子有一个致命的缺陷：只能使用一个线程并发的处理用户请求。是的，这样也可以实现并发，一秒处理几千次请求问题不大，但是这毕竟没有利用上 CPU 的多核并行能力，无法实现性能最大化。

`async` 并发和多线程其实并不冲突，而 `async-std` 包也允许我们使用多个线程去处理，由于 `handle_connection` 实现了 `Send` 特征且不会阻塞，因此使用 `async_std::task::spawn` 是非常安全

的:

```
use async_std::task::spawn;

#[async_std::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").await.unwrap();
    listener
        .incoming()
        .for_each_concurrent(/* limit */ None, |stream| async move {
            let stream = stream.unwrap();
            spawn(handle_connection(stream));
        })
        .await;
}
```

至此，我们实现了同时使用并行(多线程)和并发( `async` )来同时处理多个请求！

## 测试 `handle_connection` 函数

对于测试 Web 服务器，使用集成测试往往是最简单的，但是在本例子中，将使用单元测试来测试连接处理函数的正确性。

为了保证单元测试的隔离性和确定性，我们使用 `MockTcpStream` 来替代 `TcpStream`。首先，修改 `handle_connection` 的函数签名让测试更简单，之所以可以修改签名，原因在于 `async_std::net::TcpStream` 实际上并不是必须的，只要任何结构体实现了 `async_std::io::Read`，`async_std::io::Write` 和 `marker::Unpin` 就可以替代它：

```
use std::marker::Unpin;
use async_std::io::{Read, Write};

async fn handle_connection(mut stream: impl Read + Write + Unpin) {
```

下面，来构建一个 mock 的 `TcpStream` 并实现了上面这些特征，它包含一些数据，这些数据将被拷贝到 `read` 缓存中，然后返回 `Poll::Ready` 说明 `read` 已经结束：

```

use super::*;
use futures::io::Error;
use futures::task::{Context, Poll};

use std::cmp::min;
use std::pin::Pin;

struct MockTcpStream {
    read_data: Vec<u8>,
    write_data: Vec<u8>,
}

impl Read for MockTcpStream {
    fn poll_read(
        self: Pin<&mut Self>,
        _context: &mut Context,
        buf: &mut [u8],
    ) -> Poll<Result<usize, Error>> {
        let size: usize = min(self.read_data.len(), buf.len());
        buf[..size].copy_from_slice(&self.read_data[..size]);
        Poll::Ready(Ok(size))
    }
}

```

Write 的实现也类似，需要实现三个方法：poll\_write，poll\_flush，与 poll\_close。poll\_write 会拷贝输入数据到 mock 的 TcpStream 中，当完成后返回 Poll::Ready。由于 TcpStream 无需 flush 和 close，因此另两个方法直接返回 Poll::Ready 即可。

```

impl Write for MockTcpStream {
    fn poll_write(
        mut self: Pin<&mut Self>,
        _context: &mut Context,
        buf: &[u8],
    ) -> Poll<Result<usize, Error>> {
        self.write_data = Vec::from(buf);

        Poll::Ready(Ok(buf.len()))
    }

    fn poll_flush(self: Pin<&mut Self>, _context: &mut Context) -> Poll<Result<(), Error>> {
        Poll::Ready(Ok(()))
    }

    fn poll_close(self: Pin<&mut Self>, _context: &mut Context) -> Poll<Result<(), Error>> {
        Poll::Ready(Ok(()))
    }
}

```

最后，我们的 mock 需要实现 `Unpin` 特征，表示它可以在内存中安全的移动，具体内容在[前面章节](#)有讲。

```
use std::marker::Unpin;
impl Unpin for MockTcpStream {}
```

现在可以准备开始测试了，在使用初始化数据设置好 `MockTcpStream` 后，我们可以使用 `# [async_std::test]` 来运行 `handle_connection` 函数，该函数跟 `# [async_std::main]` 的作用类似。为了确保 `handle_connection` 函数正确工作，需要根据初始化数据检查正确的数据被写入到 `MockTcpStream` 中。

```
use std::fs;

#[async_std::test]
async fn test_handle_connection() {
    let input_bytes = b"GET / HTTP/1.1\r\n";
    let mut contents = vec![0u8; 1024];
    contents[..input_bytes.len()].clone_from_slice(input_bytes);
    let mut stream = MockTcpStream {
        read_data: contents,
        write_data: Vec::new(),
    };

    handle_connection(&mut stream).await;
    let mut buf = [0u8; 1024];
    stream.read(&mut buf).await.unwrap();

    let expected_contents = fs::read_to_string("hello.html").unwrap();
    let expected_response = format!("HTTP/1.1 200 OK\r\n{}\r\n", expected_contents);
    assert!(stream.write_data.starts_with(expected_response.as_bytes()));
}
```

# 实践应用：多线程Web服务器

一般来说，现代化的 web 服务器往往都基于更加轻量级的协程或 `async/await` 等模式实现，但是基于本章的内容，我们还是采取较为传统的多线程的方式来实现，即：一个请求连接分配一个线程去独立处理，当然还有升级版的线程池。

在本章中你将了解：

1. 学习一点 TCP 和 HTTP
  2. 在套接字 socket 上监听进入的 TCP 连接
  3. 解析 HTTP 请求
  4. 创建合适的 HTTP 应答
  5. 使用线程池来提升 web 服务器的吞吐量
- 

本章的实现方法并不是在 Rust 中实现 Web 服务器的最佳方法，后续章节的 `async/await` 会更加适合！

---

# 构建单线程 Web 服务器

在开始之前先来简单回顾下构建所需的网络协议: HTTP 和 TCP。这两种协议都是请求-应答模式的网络协议，意味着在客户端发起请求后，服务器会监听并处理进入的请求，最后给予应答，至于这个过程怎么进行，取决于具体的协议定义。

与 HTTP 有所不同，TCP 是一个底层协议，它仅描述客户端传递了信息给服务器，至于这个信息长什么样，怎么解析处理，则不在该协议的职责范畴内。

而 HTTP 协议是更高层的通信协议，一般来说都基于 TCP 来构建 (HTTP/3 是基于 UDP 构建的协议)，更高层的协议也意味着它会对传输的信息进行解析处理。

更加深入的学习网络协议并不属于本书的范畴，因此让我们从如何读取 TCP 传输的字节流开始吧。

## 监听 TCP 连接

先来创建一个全新的项目：

```
$ cargo new hello
    Created binary (application) `hello` project
$ cd hello
```

接下来，使用 `std::net` 模块监听进入的请求连接，IP 和端口是 `127.0.0.1:7878`。

```
use std::net::TcpListener;

fn main() {
    // 监听地址: 127.0.0.1:7878
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

选择 7878 端口的原因是，80 和 8080 往往都被 HTTP 服务器霸占，因此我们需要选择一个跟已经监听的端口不冲突的。

`bind` 非常类似 `new`，它返回一个 `TcpListener` 实例，只不过我们一般都说 "绑定到某个端口"，因此 `bind` 这个名称会更合适。

`unwrap` 的使用是因为 `bind` 返回 `Result<T, E>`，毕竟监听是有可能报错的，例如：如果要监听 80 端口往往需要管理员权限；监听了同样的端口，等等。

`incoming` 会返回一个迭代器，它每一次迭代都会返回一个新的连接 `stream`（客户端发起，web 服务器监听接收），因此，接下来要做的就是从 `stream` 中读取数据，然后返回处理后的结果。

细心的同学可能会注意到，代码中对 `stream` 还进行了一次 `unwrap` 处理，原因在于我们并不是在迭代一个一个连接，而是在迭代处理一个一个请求建立连接的尝试，而这种尝试可能会失败！例如，操作系统的最大连接数限制。

现在，启动服务器，然后在你的浏览器中，访问地址 `127.0.0.1:7878`，这时应该会看到一条错误信息，类似："Connection reset"，毕竟我们的服务器目前只是接收了连接，并没有回复任何数据。

```
$ cargo run
  Running `target/debug/hello`
Connection established!
Connection established!
Connection established!
```

无论浏览器怎么摆烂，我们的服务器还是成功打出了信息：TCP 连接已经成功建立。

可能大家会疑问，为啥在浏览器访问一次，可能会在终端打印出多次请求建立的信息，难道不是应该一一对应吗？原因在于当 `stream` 超出作用域时，会触发 `drop` 的扫尾工作，其中包含了关闭连接。但是，浏览器可能会存在自动重试的情况，因此还会重新建立连接，最终打印了多次。

由于 `listener.incoming` 会在当前阻塞式监听，也就是 `main` 线程会被阻塞，我们最后需要通过 `ctrl + c` 来结束程序进程。

## 读取请求

连接建立后，就可以开始读取客户端传来的数据：

```

use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("Request: {:#?}", http_request);
}

```

这段代码有几点值得注意:

- 引入 `std::io::prelude` 和 `std::io::BufReader` 是引入相应的特征和类型，帮助我们读取和写入数据
- `BufReader` 可以实现缓冲区读取，底层其实是基于 `std::io::Read` 实现
- 可以使用 `lines` 方法来获取一个迭代器，可以对传输的内容流进行按行迭代读取，要使用该方法，必须先引入 `std::io::BufRead`
- 最后使用 `collect` 消费掉迭代器，最终客户端发来的请求数据被存到 `http_request` 这个动态数组中

大家可能会比较好奇，该如何判断客户端发来的 HTTP 数据是否读取完成，答案就在于客户端会在请求数据的结尾附上两个换行符，当我们检测到某一行字符串为空时，就意味着请求数据已经传输完毕，可以 `collect` 了。

来运行下试试:

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
    Running `target/debug/hello`
Request: [
    "GET / HTTP/1.1",
    "Host: 127.0.0.1:7878",
    "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:99.0)
Gecko/20100101 Firefox/99.0",
    "Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8",
    "Accept-Language: en-US,en;q=0.5",
    "Accept-Encoding: gzip, deflate, br",
    "DNT: 1",
    "Connection: keep-alive",
    "Upgrade-Insecure-Requests: 1",
    "Sec-Fetch-Dest: document",
    "Sec-Fetch-Mode: navigate",
    "Sec-Fetch-Site: none",
    "Sec-Fetch-User: ?1",
    "Cache-Control: max-age=0",
]
]
```

呦，还挺长的，是不是长得很像我们以前见过的 HTTP 请求 JSON，来简单分析下。

## HTTP 请求长啥样

刚才的文本挺长的，但其实符合以下的格式：

```
Method Request-URI HTTP-Version
headers CRLF
message-body
```

- 第一行 Method 是请求的方法，例如 GET、POST 等，Request-URI 是该请求希望访问的目标资源路径，例如 /、/hello/world 等
- 类似 JSON 格式的数据都是 HTTP 请求报头 headers，例如 "Host: 127.0.0.1:7878"
- 至于 message-body 是消息体，它包含了用户请求携带的具体数据，例如更改用户名的请求，就要提交新的用户名数据，至于刚才的 GET 请求，它是没有 message-body 的

大家可以尝试换一个浏览器再访问一次，看看不同的浏览器请求携带的 headers 是否不同。

## 请求应答

目前为止，都是在服务器端的操作，浏览器的请求依然还会报错，是时候给予相应的请求应答了，HTTP 格式类似：

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

应答的格式与请求相差不大，其中 Status-Code 是最重要的，它用于告诉客户端，当前的请求是否成功，若失败，大概是什么原因，它就是著名的 HTTP 状态码，常用的有 200：请求成功，404 目标不存在，等等。

为了帮助大家更直观的感受下应答格式第一行长什么样，下面给出一个示例：

```
HTTP/1.1 200 OK\r\n\r\n
```

下面将该应答发送回客户端：

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let response = "HTTP/1.1 200 OK\r\n\r\n";

    stream.write_all(response.as_bytes()).unwrap();
}
```

由于 `write_all` 方法接受 `&[u8]` 类型作为参数，这里需要用 `as_bytes` 将字符串转换为字节数组。

重新启动服务器，然后再观察下浏览器中的输出，这次应该不再有报错，而是一个空白页面，因为没有返回任何具体的数据( message-body )，上面只是一条最简单的符合 HTTP 格式的数据。

## 返回 HTML 页面

空白页面显然会让人不知所措，那就返回一个简单的 HTML 页面，给用户打个招呼。

在项目的根目录下创建 `hello.html` 文件并写入如下内容：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

看得出来，这是一个非常简单的 HTML5 网页文档，基本上没人读不懂吧：）

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

// --snip--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{}Content-Length: {}\r\n\r\n{}", status_line, length, contents);

    stream.write_all(response.as_bytes()).unwrap();
}
```

新修改的代码中，读取了新增 HTML 的内容，并按照 HTTP 格式，将内容传回给客户端。

具体的运行验证就不再赘述，我们再来看看如何增加一些验证和选择性回复。

---

用这么奇怪的格式返回应答数据，原因只有一个，我们在模拟实现真正的 http web 服务器框架。事实上，写逻辑代码时，只需使用现成的 web 框架（例如 `rocket`）去启动 web 服务即可，解析请求数据和返回应答数据都已经被封装在 API 中，非常简单易用

---

## 验证请求和选择性应答

用户想要获取他的个人信息，你给他 say hi，用户想要查看他的某篇文章内容，你给他 say hi，好吧用户想要骂你，你还是给它 say hi。

是的，这种服务态度我们很欣赏，但是这种服务质量属实令人堪忧。因此我们要针对用户的不同请求给出相应不同的回复，让场景模拟更加真实。

```
fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

注意迭代器方法 `next` 的使用，原因在于我们只需要读取第一行，判断具体的 HTTP METHOD 是什么。

接着判断了用户是否请求了 / 根路径，如果是，返回之前的 `hello.html` 页面；如果不是...尚未实现。

重新运行服务器，如果你继续访问 `127.0.0.1:7878`，那么看到的依然是 `hello.html` 页面，因为默认访问根路径，但是一旦换一个路径访问，例如 `127.0.0.1:7878/something-else`，那你将继续看到之前看过多次的连接错误。

下面来完善下，当用户访问根路径之外的页面时，给他展示一个友好的 404 页面( 相比直接报错 )。

```

// --snip--
} else {
    let status_line = "HTTP/1.1 404 NOT FOUND";
    let contents = fs::read_to_string("404.html").unwrap();
    let length = contents.len();

    let response = format!(
        "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
    );

    stream.write_all(response.as_bytes()).unwrap();
}

```

哦对了，别忘了在根路径下创建 `404.html` 并填入下面内容：

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>你好!</title>
  </head>
  <body>
    <h1>很抱歉!</h1>
    <p>由于运维删库跑路，我们的数据全部丢失，总监也已经准备跑路，88</p>
  </body>
</html>

```

最后，上面的代码其实有很多重复，可以提取出来进行简单重构：

```

// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{}Content-Length: {}\r\n\r\n{}", status_line, length);
    stream.write_all(response.as_bytes()).unwrap();
}

```

至此，单线程版本的服务器已经完成，但是说实话，没啥用，总不能让你的用户排队等待访问吧，那也太糟糕了…

# 构建多线程 Web 服务器

目前的单线程版本只能依次处理用户的请求：一时间只能处理一个请求连接。随着用户的请求数增多，可以预料的是排在后面的用户可能要等待数十秒甚至超时！

本章我们将解决这个问题，但是首先来模拟一个慢请求场景，看看单线程是否真的如此糟糕。

## 基于单线程模拟慢请求

下面的代码中，使用 `sleep` 的方式让每次请求持续 5 秒，模拟真实的慢请求：

```
// in main.rs
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };
    // --snip--
}
```

由于增加了新的请求路径 `/sleep`，之前的 `if else` 被修改为 `match`，需要注意的是，由于 `match` 不会像方法那样自动做引用或者解引用，因此我们需要显式调用：`match &request_line[..]`，来获取所需的 `&str` 类型。

可以看出，当用户访问 `/sleep` 时，请求会持续 5 秒后才返回，下面来试试，启动服务器后，打开你的浏览器，这次要分别打开两个页面(tab页): `http://127.0.0.1:7878/` 和 `http://127.0.0.1:7878/sleep`。

此时，如果我们连续访问 `/` 路径，那效果跟之前一样：立刻看到请求的页面。但假如先访问 `/sleep`，接着在另一个页面访问 `/`，就会看到 `/` 的页面直到 5 秒后才会刷出来，验证了请求排队这个糟糕的事实。

至于如何解决，其实办法不少，本章我们来看看一个经典解决方案：线程池。

## 使用线程池改善吞吐

线程池包含一组已生成的线程，它们时刻等待着接收并处理新的任务。当程序接收到新任务时，它会将线程池中的一个线程指派给该任务，在该线程忙着处理时，新来的任务会交给池中剩余的线程进行处理。最终，当执行任务的线程处理完后，它会被重新放入到线程池中，准备处理新任务。

假设线程池中包含 N 个线程，那么可以推断出，服务器将拥有并发处理 N 个请求连接的能力，从而增加服务器的吞吐量。

同时，我们将限制线程池中的线程数量，以保护服务器免受拒绝服务攻击（DoS）的影响：如果针对每个请求创建一个新线程，那么一个人向我们的服务器发出 1000 万个请求，会直接耗尽资源，导致后续用户的请求无法被处理，这也是拒绝服务名称的来源。

因此，还需对线程池进行一定的架构设计，首先是设定最大线程数的上限，其次维护一个请求队列。池中的线程去队列中依次弹出请求并处理。这样就可以同时并发处理 N 个请求，其中 N 是线程数。

但聪明的读者可能会想到，假如每个请求依然耗时很长，那请求队列依然会堆积，后续的用户请求还是需要等待较长的时间，毕竟你也就 N 个线程，但总归比单线程要强 N 倍吧 :D

当然，线程池依然是较为传统的提升吞吐方法，比较新的有：单线程异步 IO，例如 redis；多线程异步 IO，例如 Rust 的主流 web 框架。事实上，大家在下一个实战项目中，会看到相关技术的应用。

## 为每个请求生成一个线程

这显然不是我们的最终方案，原因在于它会生成无上限的线程数，最终导致资源耗尽。但它确实是一个好的起点：

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

这种实现下，依次访问 `/sleep` 和 `/` 就无需再等待，不错的开始。

## 限制创建线程的数量

原则上，我们希望在上面代码的基础上，尽量少的去修改，下面是一个假想的线程池 API 实现：

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

代码跟之前的类似，也非常简洁明了，`ThreadPool::new(4)` 创建一个包含 4 个线程的线程池，接着通过 `pool.execute` 去分发执行请求。

显然，上面的代码无法编译，下面来逐步实现。

## 使用编译器驱动的方式开发 ThreadPool

你可能听说过测试驱动开发，但听过编译器驱动开发吗？来见识下 Rust 中的绝招吧。

检查之前的代码，看看报什么错：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
 --> src/main.rs:11:16
  |
11 |     let pool = ThreadPool::new(4);
  |           ^^^^^^^^^^ use of undeclared type `ThreadPool`


For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` due to previous error
```

俗话说，不怕敌人很强，就怕他们不犯错，很好，编译器漏出了破绽。看起来我们需要实现 `ThreadPool` 类型。看起来，还需要添加一个库包，未来线程池的代码都将在这个独立的包中完成，甚至于未来你要实现其它的服务，也可以复用这个多线程库包。

创建 `src/lib.rs` 文件并写入如下代码：

```
pub struct ThreadPool;
```

接着在 `main.rs` 中引入：

```
// main.rs
use hello::ThreadPool;
```

编译后依然报错：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for struct
`ThreadPool` in the current scope
 --> src/main.rs:12:28
  |
12 |     let pool = ThreadPool::new(4);
  |           ^^^ function or associated item not found in
`ThreadPool`


For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

好，继续实现 `new` 函数：

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

继续检查：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for struct `ThreadPool` in the current
scope
--> src/main.rs:17:14
 |
17 |         pool.execute(|| {
|             ^^^^^^^^ method not found in `ThreadPool`


For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

这个方法类似于 `thread::spawn`，用于将闭包中的任务交给某个空闲的线程去执行。

其实这里有一个小难点：`execute` 的参数是一个闭包，回忆下之前学过的内容，闭包作为参数时可以由三个特征进行约束：`Fn`、`FnMut` 和 `FnOnce`，选哪个就成为一个问题。由于 `execute` 在实现上类似 `thread::spawn`，我们可以参考下后者的签名如何声明。

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

可以看出，`spawn` 选择 `FnOnce` 作为 `F` 闭包的特征约束，原因是闭包作为任务只需被线程执行一次即可。

`F` 还有一个特征约束 `Send`，也可以照抄过来，毕竟闭包需要从一个线程传递到另一个线程，至于生命周期约束 `'static`，是因为我们并不知道线程需要多久时间来执行该任务。

```
impl ThreadPool {
    // --snip--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

在理解 `spawn` 后，就可以轻松写出如上的 `execute` 实现，注意这里的 `FnOnce()` 跟 `spawn` 有所不同，原因是 `execute` 传入的闭包没有参数也没有返回值。

```
$ cargo check
Checking hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

成功编译，但在浏览器访问依然会报之前类似的错误，下面来实现 `execute`。

## new 还是 build

关于 `ThreadPool` 的构造函数，存在两个选择 `new` 和 `build`。

`new` 往往用于简单初始化一个实例，而 `build` 往往会完成更加复杂的构建工作，例如入门实战中的 `Config::build`。

在这个项目中，我们并不需要在初始化线程池的同时创建相应的线程，因此 `new` 是更适合的选择：

```
impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }
    // --snip--
}
```

这里有两点值得注意：

- `usize` 类型包含 `0`，但是创建没有任何线程的线程池显然是无意义的，因此做一下 `assert!` 验证
- `ThreadPool` 拥有不错的[文档注释](#)，甚至包含了可能 `panic` 的情况，通过 `cargo doc --open` 可以访问文档注释

## 存储线程

创建 `ThreadPool` 后，下一步就是存储具体的线程，既然要存放线程，一个绕不过去的问题就是：用什么类型来存放，例如假如使用 `Vec<T>` 来存储，那这个 `T` 应该是什么？

估计还得探索下 `thread::spawn` 的签名，毕竟它生成并返回一个线程：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

看起来 `JoinHandle<T>` 是我们需要的，这里的 `T` 是传入的闭包任务所返回的，我们的任务无需任何返回，因此 `T` 直接使用 `()` 即可。

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // create some threads and store them in the vector
        }

        ThreadPool { threads }
    }
    // --snip--
}
```

如上所示，最终我们使用 `Vec<thread::JoinHandle<()>>` 来存储线程，同时设定了容量上限 `with_capacity(size)`，该方法还可以提前分配好内存空间，比 `Vec::new` 的性能要更好一点。

## 将代码从 ThreadPool 发送到线程中

上面的代码留下一个未实现的 `for` 循环，用于创建和存储线程。

学过多线程一章后，大家应该知道 `thread::spawn` 虽然是生成线程最好的方式，但是它会立即执行传入的任务，然而，在我们的使用场景中，创建线程和执行任务明显是要分离的，因此标准库看起来不再适合。

可以考虑创建一个 `Worker` 结构体，作为 `ThreadPool` 和任务线程联系的桥梁，它的任务是获得将要执行的代码，然后在具体的线程中去执行。想象一个场景：一个餐馆，`Worker` 等待顾客的点餐，然后将具体的点餐信息传递给厨房，感觉类似服务员？

引入 `Worker` 后，就无需再存储 `JoinHandle<()>` 实例，直接存储 `Worker` 实例：该实例内部会存储 `JoinHandle<()>`。下面是新的线程池创建流程：

```

use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --snip--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        // 尚未实现..
        let thread = thread::spawn(|| {});
        // 每个 `Worker` 都拥有自己的唯一 id
        Worker { id, thread }
    }
}

```

由于外部调用者无需知道 `Worker` 的存在，因此这里使用了私有的声明。

大家可以编译下代码，如果出错了，请仔细检查下，是否遗漏了什么，截止目前，代码是完全可以通过编译的，但是任务该怎么执行依然还没有实现。

## 将请求发送给线程

在上面的代码中，`thread::spawn(|| {})` 还没有给予实质性的内容，现在一起来完善下。

首先 `Worker` 结构体需要从线程池 `ThreadPool` 的队列中获取待执行的代码，对于这类场景，消息传递非常适合：我们将使用消息通道(`channel`)作为任务队列。

```
use std::{sync::mpsc, thread};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}
```

阅读过之前内容的同学应该知道，消息通道有发送端和接收端，其中线程池 `ThreadPool` 持有发送端，通过 `execute` 方法来发送任务。那么问题来了，谁持有接收端呢？答案是 `Worker`，它的内部线程将接收任务，然后进行处理。

```

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool { workers, sender }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker { id, thread }
    }
}

```

看起来很美好，但是很不幸，它会报错：

```

$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:26:42
|
21 |         let (sender, receiver) = mpsc::channel();
|             ----- move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
...
26 |             workers.push(Worker::new(id, receiver));
|                     ^^^^^^^^^^ value moved here, in previous
iteration of loop

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` due to previous error

```

原因也很简单，`receiver` 并没有实现 `Copy`，因此它的所有权在第一次循环中，就被传入到第一个 `Worker` 实例中，后续自然无法再使用。

报错就解决呗，但 Rust 中的 channel 实现是 mpsc，即多生产者单消费者，因此我们无法通过克隆消费者的方式来修复这个错误。当然，发送多条消息给多个接收者也不在考虑范畴，该怎么办？似乎陷入了绝境。

雪上加霜的是，就算 receiver 可以克隆，但是你得保证同一个时间只有一个 receiver 能接收消息，否则一个任务可能同时被多个 worker 执行，因此多个线程需要安全的共享和使用 receiver，等等，安全的共享？听上去 Arc 这个多所有权结构非常适合，互斥使用？貌似 Mutex 很适合，结合一下，Arc<Mutex<T>>，这不就是我们之前见过多次的线程安全类型吗？

总之，Arc 允许多个 Worker 同时持有 receiver，而 Mutex 可以确保一次只有一个 Worker 能从 receiver 接收消息。

```
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};

// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    // --snip--
}

// --snip--


impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
    }
}
```

修改后，每一个 Worker 都可以安全的持有 receiver，同时不必担心一个任务会被重复执行多次，完美！

## 实现 execute 方法

首先，需要为一个很长的类型创建一个别名，有多长呢？

```
// --snip--  
  
type Job = Box<dyn FnOnce() + Send + 'static>;  
  
impl ThreadPool {  
    // --snip--  
  
    pub fn execute<F>(&self, f: F)  
    where  
        F: FnOnce() + Send + 'static,  
    {  
        let job = Box::new(f);  
  
        self.sender.send(job).unwrap();  
    }  
}  
  
// --snip--
```

创建别名的威力暂时还看不到，敬请期待。总之，这里的工作很简单，将传入的任务包装成 Job 类型后，发送出去。

但是还没完，接收的代码也要完善下：

```
// --snip--  
  
impl Worker {  
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {  
        let thread = thread::spawn(move || loop {  
            let job = receiver.lock().unwrap().recv().unwrap();  
  
            println!("Worker {} got a job; executing.", id);  
  
            job();  
        });  
  
        Worker { id, thread }  
    }  
}
```

修改后，就可以不停地循环去接收任务，最后进行执行。还可以看到因为之前 Job 别名的引入， new 函数的签名才没有过度复杂，否则你将看到的是 fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Box<dyn FnOnce() + Send + 'static>>>>) -> Worker ，感受下类型别名的威力吧 :D

`lock()` 方法可以获得一个 `Mutex` 锁，至于为何使用 `unwrap`，难道获取锁还能失败？没错，假如当前持有锁的线程 `panic` 了，那么这些等待锁的线程就会获取一个错误，因此通过 `unwrap` 来让当前等待的线程 `panic` 是一个不错的解决方案，当然你还可以换成 `expect`。

一旦获取到锁里的内容 `mpsc::Receiver<Job>` 后，就可以调用其上的 `recv` 方法来接收消息，依然是一个 `unwrap`，原因在于持有发送端的线程可能会被关闭，这种情况下直接 `panic` 也是不错的。

`recv` 的调用过程是阻塞的，意味着若没有任何任务，那当前的调用线程将一直等待，直到接收到新的任务。`Mutex<T>` 可以同一个任务只会被一个 Worker 获取，不会被重复执行。

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never read: `workers`
--> src/lib.rs:7:5
|
7 |     workers: Vec<Worker>,
|     ^^^^^^^^^^^^^^^^^^
|
|= note: `#[warn(dead_code)]` on by default

warning: field is never read: `id`
--> src/lib.rs:48:5
|
48 |     id: usize,
|     ^^^^^^
|
warning: field is never read: `thread`
--> src/lib.rs:49:5
|
49 |     thread: thread::JoinHandle<()>,
|     ^^^^^^^^^^^^^^^^^^
|
warning: `hello` (lib) generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 1.40s
    Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

终于，程序如愿运行起来，我们的线程池可以并发处理任务了！从打印的数字可以看到，只有 4 个线程去执行任务，符合我们对线程池的要求，这样再也不用担心系统的线程资源会被消耗殆尽了！

---

注意：处于缓存的考虑，有些浏览器会对多次同样的请求进行顺序的执行，因此你可能还是会遇到访问 `/sleep` 后，就无法访问另一个 `/sleep` 的问题：(

---

## while let 的巨大陷阱

还有一个问题，为啥之前我们不用 `while let` 来循环？例如：

```
// --snip--  
  
impl Worker {  
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {  
        let thread = thread::spawn(move || {  
            while let Ok(job) = receiver.lock().unwrap().recv() {  
                println!("Worker {id} got a job; executing.");  
  
                job();  
            }  
        });  
  
        Worker { id, thread }  
    }  
}
```

这段代码编译起来没问题，但是并不会产生我们预期的结果：后续请求依然需要等待慢请求的处理完成后，才能被处理。奇怪吧，仅仅是从 `let` 改成 `while let` 就会变成这样？大家可以思考下为什么会这样，具体答案会在下一章节末尾给出，这里先给出一个小提示：`Mutex` 获得的锁在作用域结束后才会被释放。

# 优雅关闭和资源清理

之前的程序，如果使用 `ctrl-c` 的方法来关闭，所有的线程都会立即停止，这会造成正在请求的用户感知到一个明显的错误。

因此我们需要添加一些优雅关闭( Graceful Shutdown )，以更好的完成资源清理等收尾工作。

## 为线程池实现 Drop

当线程池被 drop 时，需要等待所有的子线程完成它们的工作，然后再退出，下面是一个初步尝试：

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

这里通过实现 `Drop` 特征来为线程池添加资源收尾工作，代码比较简单，就是依次调用每个线程的 `join` 方法。编译下试试：

```
$ cargo check
   Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a mutable reference
--> src/lib.rs:52:13
|
52 |         worker.thread.join().unwrap();
|         ^^^^^^^^^^^^^^ ----- `worker.thread` moved due to this method call
|         |
|             move occurs because `worker.thread` has type `JoinHandle<()>`, which
does not implement the `Copy` trait
|
note: this function takes ownership of the receiver `self`, which moves
`worker.thread`

For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` due to previous error
```

这里的报错很明显，`worker.thread` 试图拿走所有权，但是 `worker` 仅仅是一个可变借用，显然是不可行的。

目前来看，只能将 `thread` 从 `worker` 中移动出来，一个可行的尝试：

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

对于 `Option` 类型，可以使用 `take` 方法拿走内部值的所有权，同时留下一个 `None` 在风中孤独凌乱。  
继续尝试编译驱动开发模式：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `join` found for enum `Option` in the current scope
--> src/lib.rs:52:27
52 |         worker.thread.join().unwrap();
|             ^^^^^^ method not found in `Option<JoinHandle<()>>`
|
note: the method `join` exists on the type `JoinHandle<()`>
help: consider using `Option::expect` to unwrap the `JoinHandle<()`> value, panicking
if the value is an `Option::None`
|
52 |         worker.thread.expect("REASON").join().unwrap();
|             ++++++++
```

```
error[E0308]: mismatched types
--> src/lib.rs:72:22
72 |     Worker { id, thread }
|           ^^^^^^ expected enum `Option`, found struct `JoinHandle`
|
= note: expected enum `Option<JoinHandle<()>>`
       found struct `JoinHandle<_>`
help: try wrapping the expression in `Some`
|
72 |     Worker { id, thread: Some(thread) }
|           ++++++++ +
```

先来解决第二个类型不匹配的错误：

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}
```

简单搞定，回头看看第一个错误，既然换了 Option，就可以用 take 拿走所有权：

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

注意这种 if let 的写法，若 worker.thread 已经是 None，什么都不会发生，符合我们的预期；若包含一个线程，那就拿走其所有权，然后调用 join。

## 停止工作线程

虽然调用了 join，但是目标线程依然不会停止，原因在于它们在无限的 loop 循环等待，看起来需要借用 channel 的 drop 机制：释放 sender 发送端后，receiver 接收端会收到报错，然后再退出即可。

```

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --snip--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --snip--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

```

上面做了两处改变:

1. 为 `sender` 增加 `Option` 封装, 这样可以用 `take` 拿走所有权, 跟之前的 `thread` 一样
2. 主动调用 `drop` 关闭发送端 `sender`

关闭 `sender` 后, 将关闭对应的 `channel`, 意味着不会再有任何消息被发送。随后, 所有的处于无限 `loop` 的接收端将收到一个错误, 我们根据错误再进行进一步的处理。

```

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || loop {
            let message = receiver.lock().unwrap().recv();

            match message {
                Ok(job) => {
                    println!("Worker {id} got a job; executing.");
                    job();
                }
                Err(_) => {
                    println!("Worker {id} disconnected; shutting down.");
                    break;
                }
            }
        });
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

为了快速验证代码是否正确，修改 `main` 函数，让其只接收前两个请求：

```

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

```

`take` 是迭代器 `Iterator` 上的方法，会限制后续的迭代进行最多两次，然后就结束监听，随后 `ThreadPool` 也将超出作用域并自动触发 `drop`。

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 1.0s
    Running `target/debug/hello`
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

可以看到，代码按照我们的设想如期运行，至此，一个基于线程池的简单 Web 服务器已经完成，下面是完整的代码：

## 完整代码

```
// src/main.rs
use hello::ThreadPool;
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        status_line,
        contents.len(),
        contents
    );
}
```

```
    stream.write_all(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

```
// src/lib.rs
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
```

```

        drop(self.sender.take());

    for worker in &mut self.workers {
        println!("Shutting down worker {}", worker.id);

        if let Some(thread) = worker.thread.take() {
            thread.join().unwrap();
        }
    }
}

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || loop {
            let message = receiver.lock().unwrap().recv();

            match message {
                Ok(job) => {
                    println!("Worker {} got a job; executing.", id);

                    job();
                }
                Err(_) => {
                    println!("Worker {} disconnected; shutting down.", id);
                    break;
                }
            }
        });
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

## 可以做的更多

事实上，我们还可以做更多，但是受制于篇幅，就不再展开，感兴趣的同學可以自行完成。

- 增加更多的文档
- 为线程池增加测试
- 尽可能移除 `unwrap`，替换为错误处理

- 使用线程池完成其它类型的工作，而不仅仅是本章的 Web 服务器
- 在 `crates.io` 上找到一个线程池实现，然后使用该包实现一个类似的 Web 服务器

## 上一章节的遗留问题

在上一章节的末尾，我们提到将 `let` 替换为 `while let` 后，多线程的优势将荡然无存，原因藏的很隐蔽：

1. `Mutex` 结构体没有提供显式的 `unlock`，要依赖作用域结束后的 `drop` 来自动释放
2. `let job = receiver.lock().unwrap().recv().unwrap();` 在这行代码中，由于使用了 `let`，右边的任何临时变量会在 `let` 语句结束后立即被 `drop`，因此锁会自动释放
3. 然而 `while let` (还包括 `if let` 和 `match`) 直到最后一个花括号后，才触发 `drop`

```
impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {id} got a job; executing.");
                job();
            }
        });
        Worker { id, thread }
    }
}
```

根据之前的分析，上面的代码直到 `job()` 任务执行结束后，才会释放锁，去执行另一个请求，最终造成请求排队。

# 进阶实战：实现一个简单 redis

在入门实战，你可以说众览半山不咋小，但是能坚持到这里，甚至当完成后，就真的是一览众山小，余敌皆鱼虾了。

在进阶实战中，我们要来真的了，之前的简单命令行程序，是真的简单，但是这次的简单 redis 是真的不简单，在这里你将被迫使用十八般武艺，特别的，我们还将学会 Rust 异步镇山之宝 `tokio` 包的使用。

---

本章在内容上大量借鉴和翻译了 `tokio` 官方文档[Tokio Tutorial](#)，但是重新组织了内容形式并融入了很多自己的见解和感悟，给大家提供更好的可读性和知识扩展性

---

# tokio 概览

对于 Async Rust，最最重要的莫过于底层的异步运行时，它提供了执行器、任务调度、异步 API 等核心服务。简单来说，使用 Rust 提供的 `async/await` 特性编写的异步代码要运行起来，就必须依赖于异步运行时，否则这些代码将毫无用处。

## 异步运行时

Rust 语言本身只提供了异步编程所需的基本特性，例如 `async/await` 关键字，标准库中的 `Future` 特征，官方提供的 `futures` 实用库，这些特性单独使用没有任何用处，因此我们需要一个运行时来将这些特性实现的代码运行起来。

异步运行时是由 Rust 社区提供的，它们的核心是一个 `reactor` 和一个或多个 `executor` (执行器)：

- `reactor` 用于提供外部事件的订阅机制，例如 I/O、进程间通信、定时器等
- `executor` 在上一章我们有过深入介绍，它用于调度和执行相应的任务( `Future` )

目前最受欢迎的几个运行时有：

- `tokio`，目前最受欢迎的异步运行时，功能强大，还提供了异步所需的各种工具(例如 `tracing` )、网络协议框架(例如 HTTP, gRPC )等等
- `async-std`，最大的优点就是跟标准库兼容性较强
- `smol`，一个小巧的异步运行时

但是，大浪淘沙，留下的才是金子，随着时间的流逝，`tokio` 越来越亮眼，无论是性能、功能还是社区、文档，它在各个方面都异常优秀，时至今日，可以说已成为事实上的标准。

### 异步运行时的兼容性

为何选择异步运行时这么重要？不仅仅是它们在功能、性能上存在区别，更重要的是当你选择了一个，往往就无法切换到另外一个，除非异步代码很少。

使用异步运行时，往往伴随着对它相关的生态系统的深入使用，因此耦合性会越来越强，直至最后你很难切换到另一个运行时，例如 `tokio` 和 `async-std`，就存在这种问题。

如果你实在有这种需求，可以考虑使用 `async-compat`，该包提供了一个中间层，用于兼容 `tokio` 和其它运行时。

## 结论

相信大家看到现在，心中应该有一个结论了。首先，运行时之间的不兼容性，让我们必须提前选择一个运行时，并且在未来坚持用下去，那这个运行时就应该是最优秀、最成熟的那个，`tokio` 几乎成了不二选择，当然 `tokio` 也有自己的问题：更难上手和运行时之间的兼容性。

如果你只用 `tokio`，那兼容性自然不是问题，至于难以上手，Rust 这么难，我们都学到现在了，何况区区一个异步运行时，在本书的帮助下，这些都不再是问题：）

## tokio 简介

`tokio` 是一个纸醉金迷之地，只要有钱就可以为所欲为，哦，抱歉，走错片场了。`tokio` 是 Rust 最优秀的异步运行时框架，它提供了写异步网络服务所需的几乎所有功能，不仅仅适用于大型服务器，还适用于小型嵌入式设备，它主要由以下组件构成：

- 多线程版本的异步运行时，可以运行使用 `async/await` 编写的代码
- 标准库中阻塞 API 的异步版本，例如 `thread::sleep` 会阻塞当前线程，`tokio` 中就提供了相应的异步实现版本
- 构建异步编程所需的生态，甚至还提供了 `tracing` 用于日志和分布式追踪，提供 `console` 用于 Debug 异步编程

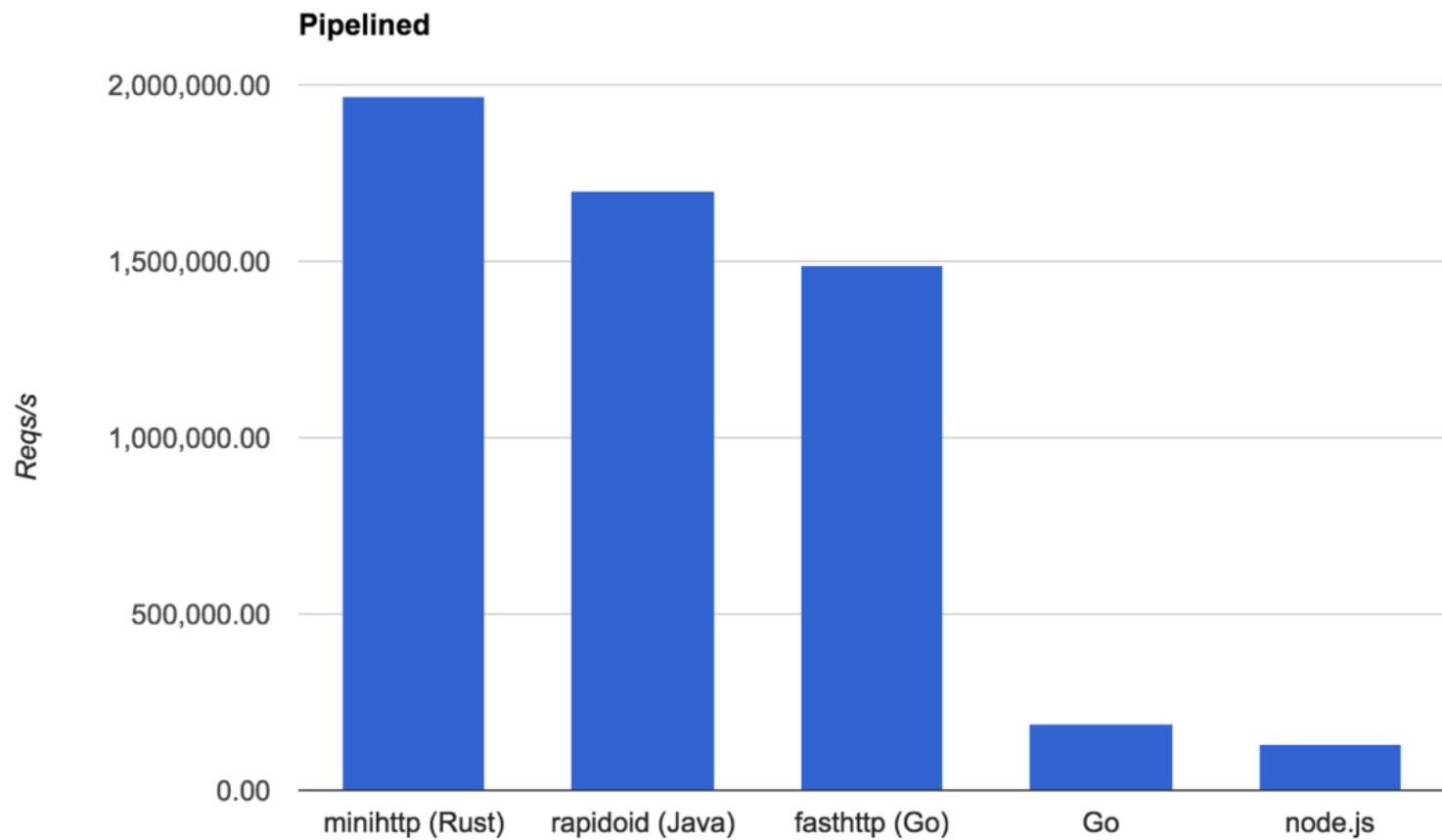
## 优势

下面一起来看看使用 `tokio` 能给你提供哪些优势。

### 高性能

因为快所以快，前者是 Rust 快，后者是 `tokio` 快。`tokio` 在编写时充分利用了 Rust 提供的各种零成本抽象和高性能特性，而且贯彻了 Rust 的牛逼思想：如果你选择手写代码，那么最好的结果就是跟 `tokio` 一样快！

以下是一张官方提供的性能参考图，大致能体现出 `tokio` 的性能之恐怖：



## 高可靠

Rust 语言的安全可靠性顺理成章的影响了 `tokio` 的可靠性，曾经有一个调查给出了令人乍舌的[结论](#)：软件系统 70% 的高危漏洞都是由内存不安全性导致的。

在 Rust 提供的安全性之外，`tokio` 还致力于提供一致性的行为表现：无论你何时运行系统，它的预期表现和性能都是一致的，例如不会出现莫名其妙的请求延迟或响应时间大幅增加。

## 简单易用

通过 Rust 提供的 `async/await` 特性，编写异步程序的复杂性相比当初已经大幅降低，同时 `tokio` 还为我们提供了丰富的生态，进一步大幅降低了其复杂性。

同时 `tokio` 遵循了标准库的命名规则，让熟悉标准库的用户可以很快习惯于 `tokio` 的语法，再借助于 Rust 强大的类型系统，用户可以轻松地编写和交付正确的代码。

## 使用灵活性

`tokio` 支持你灵活的定制自己想要的运行时，例如你可以选择多线程 + 任务窃取模式的复杂运行时，也可以选择单线程的轻量级运行时。总之，几乎你的每一种需求在 `tokio` 中都能寻找到支持(画外音：强大的灵活性需要一定的复杂性来换取，并不是免费的午餐)。

## 劣势

虽然 `tokio` 对于大多数需要并发的项目都是非常适合的，但是确实有一些场景它并不适合使用：

- **并行运行 CPU 密集型的任务。** `tokio` 非常适合于 IO 密集型任务，这些 IO 任务的绝大多数时间都用于阻塞等待 IO 的结果，而不是刷刷刷的单烤 CPU。如果你的应用是 CPU 密集型(例如并行计算)，建议使用 `rayon`，当然，对于其中的 IO 任务部分，你依然可以混用 `tokio`
- **读取大量的文件。** 读取文件的瓶颈主要在于操作系统，因为 OS 没有提供异步文件读取接口，大量的并发并不会提升文件读取的并行性能，反而可能会造成不可忽视的性能损耗，因此建议使用线程(或线程池)的方式
- **发送少量 HTTP 请求。** `tokio` 的优势是给予你并发处理大量任务的能力，对于这种轻量级 HTTP 请求场景，`tokio` 除了增加你的代码复杂性，并无法带来什么额外的优势。因此，对于这种场景，你可以使用 `reqwest` 库，它会更加简单易用。

---

若大家使用 `tokio`，那 CPU 密集的任务尤其需要用线程的方式去处理，例如使用 `spawn_blocking` 创建一个阻塞的线程去完成相应 CPU 密集任务。

原因是：`tokio` 是协作式的调度器，如果某个 CPU 密集的异步任务是通过 `tokio` 创建的，那理论上来说，该异步任务需要跟其它的异步任务交错执行，最终大家都得到了执行，皆大欢喜。但实际情况是，CPU 密集的任务很可能会一直霸着着 CPU，此时 `tokio` 的调度方式决定了该任务会一直被执行，这意味着，其它的异步任务无法得到执行的机会，最终这些任务都会因为得不到资源而饿死。

而使用 `spawn_blocking` 后，会创建一个单独的 OS 线程，该线程并不会被 `tokio` 所调度(被 OS 所调度)，因此它所执行的 CPU 密集任务也不会导致 `tokio` 调度的那些异步任务被饿死

---

## 总结

离开第三方开源社区提供的异步运行时，`async/await` 什么都不是，甚至还不如一堆破铜烂铁，除非你选择根据自己的需求手撸一个。

而 `tokio` 就是那颗皇冠上的夜明珠，也是值得我们投入时间去深入学习的开源库，它的设计原理和代码实现都异常优秀，在之后的章节中，我们将对其进行深入学习和剖析，敬请期待。

# tokio 初印象

又到了喜闻乐见的初印象环节，这个环节决定了你心中的那 24 盏灯最终是全亮还是全灭。

在本文中，我们将看看本专题的学习目标、`tokio` 该怎么引入以及如何实现一个 `Hello Tokio` 项目，最终亮灯还是灭灯的决定权留给各位看官。但我提前说好，如果你全灭了，但却找不到更好的，未来还是得回来真香:P

## 专题目标

通过 API 学项目无疑是无聊的，因此我们采用一个与众不同的方式：边学边练，在本专题的最后你将拥有一个 `redis` 客户端和服务端，当然不会实现一个完整版本的 `redis`，只会提供基本的功能和部分常用的命令。

### mini-redis

`redis` 的项目源码可以[在这里访问](#)，本项目是从[官方地址](#) fork 而来，在未来会提供注释和文档汉化。

再次声明：该项目仅仅用于学习目的，因此它的文档注释非常全，但是它完全无法作为 `redis` 的替代品。

## 环境配置

首先，我们假定你已经安装了 Rust 和相关的工具链，例如 `cargo`。其中 Rust 版本的最低要求是 `1.45.0`，建议使用最新版 `1.58`：

```
sunfei@sunface $ rustc --version
rustc 1.58.0 (02072b482 2022-01-11)
```

接下来，安装 `mini-redis` 的服务器端，它可以用来测试我们后面将要实现的 `redis` 客户端：

```
$ cargo install mini-redis
```

---

如果下载失败，也可以通过[这个地址](#)下载源码，然后在本地通过 `cargo run` 运行。

---

下载成功后，启动服务端：

```
$ mini-redis-server
```

然后，再使用客户端测试下刚启动的服务端：

```
$ mini-redis-cli set foo 1
OK
$ mini-redis-cli get foo
"1"
```

不得不说，还挺好用的，先自我陶醉下 :) 此时，万事俱备，只欠东风，接下来是时候亮"箭"了：实现我们的 Hello Tokio 项目。

## Hello Tokio

与简单无比的 Hello World 有所不同(简单？还记得本书开头时，湖畔边的那个多国语言版本的 你好，世界嘛~~)，Hello Tokio 它承载着"非常艰巨"的任务，那就是向刚启动的 redis 服务器写入一个 key=hello，value=world，然后再读取出来，嗯，使用 mini-redis 客户端 :)

### 分析未到，代码先行

在详细讲解之前，我们先来看看完整的代码，让大家有一个直观的印象。首先，创建一个新的 Rust 项目：

```
$ cargo new my-redis
$ cd my-redis
```

然后在 Cargo.toml 中添加相关的依赖：

```
[dependencies]
tokio = { version = "1", features = ["full"] }
mini-redis = "0.4"
```

接下来，使用以下代码替换 main.rs 中的内容：

```
use mini_redis::{client, Result};

#[tokio::main]
async fn main() -> Result<()> {
    // 建立与mini-redis服务器的连接
    let mut client = client::connect("127.0.0.1:6379").await?;

    // 设置 key: "hello" 和 值: "world"
    client.set("hello", "world".into()).await?;

    // 获取"key=hello"的值
    let result = client.get("hello").await?;

    println!("从服务器端获取到结果={:?}", result);

    Ok(())
}
```

不知道你之前启动的 `mini-redis-server` 关闭没有，如果关了，记得重新启动下，否则我们的代码就是意大利空气炮。

最后，运行这个项目：

```
$ cargo run
从服务器端获取到结果=Some("world")
```

Perfect，代码成功运行，是时候来解释下其中蕴藏的至高奥秘了。

## 原理解释

代码篇幅虽然不长，但是还是有不少值得关注的地方，接下来我们一起来看看。

```
let mut client = client::connect("127.0.0.1:6379").await?;
```

`client::connect` 函数由 `mini-redis` 包提供，它使用异步的方式跟指定的远程 IP 地址建立 TCP 长连接，一旦连接建立成功，那 `client` 的赋值初始化也将完成。

特别值得注意的是：虽然该连接是异步建立的，但是从代码本身来看，完全是**同步的代码编写方式**，唯一能说明异步的点就是 `.await`。

## 什么是异步编程

大部分计算机程序都是按照代码编写的顺序来执行的：先执行第一行，然后第二行，以此类推(当然，还要考虑流程控制，例如循环)。当进行同步编程时，一旦程序遇到一个操作无法被立即完成，它就会进入阻塞状态，直到该操作完成为止。

因此同步编程非常符合我们人类的思维习惯，是一个顺其自然的过程，被几乎每一个程序员所喜欢(本来想说所有，但我不敢打包票，毕竟总有特立独行之士)。例如，当建立 TCP 连接时，当前线程会被阻塞，直到等待该连接建立完成，然后才往下继续进行。

而使用异步编程，无法立即完成的操作会被切到后台去等待，因此当前线程不会被阻塞，它会接着执行其它的操作。一旦之前的操作准备好可以继续执行后，它会通知执行器，然后执行器会调度它并从上次离开的点继续执行。但是大家想象下，如果没有使用 `await`，而是按照这个异步的流程使用通知 -> 回调的方式实现，代码该多么的难写和难读！

好在 Rust 为我们提供了 `async/await` 的异步编程特性，让我们可以像写同步代码那样去写异步的代码，也让这个世界美好依旧。

## 编译时绿色线程

一个函数可以通过 `async fn` 的方式被标记为异步函数：

```
use mini_redis::Result;
use mini_redis::client::Client;
use tokio::net::ToSocketAddrs;

pub async fn connect<T: ToSocketAddrs>(addr: T) -> Result<Client> {
    // ...
}
```

在上例中，`redis` 的连接函数 `connect` 实现如上，它看上去很像是一个同步函数，但是 `async fn` 出卖了它。`async fn` 异步函数并不会直接返回值，而是返回一个 `Future`，顾名思义，该 `Future` 会在未来某个时间点被执行，然后最终获取到真实的返回值 `Result<Client>`。

---

`async/await` 的原理就算大家不理解，也不妨碍使用 `tokio` 写出能用的服务，但是如果想要更深入的用好，强烈建议认真读下本书的 [async/await 异步编程章节](#)，你会对 Rust 的异步编程有一个全新且深刻的认识。

---

由于 `async` 会返回一个 `Future`，因此我们还需要配合使用 `.await` 来让该 `Future` 运行起来，最终获得返回值：

```
async fn say_to_world() -> String {
    String::from("world")
}

#[tokio::main]
async fn main() {
    // 此处的函数调用是惰性的，并不会执行 `say_to_world()` 函数体中的代码
    let op = say_to_world();

    // 首先打印出 "hello"
    println!("hello");

    // 使用 `.await` 让 `say_to_world` 开始运行起来
    println!("{}", op.await);
}
```

上面代码输出如下:

```
hello
world
```

而大家可能很好奇 `async fn` 到底返回什么吧？它实际上返回的是一个实现了 `Future` 特征的匿名类型：

```
impl Future<Output = String> .
```

### async main

在代码中，使用了一个与众不同的 `main` 函数：`async fn main`，而且是用 `#[tokio::main]` 属性进行了标记。异步 `main` 函数有以下意义：

- `.await` 只能在 `async` 函数中使用，如果是以前的 `fn main`，那它内部是无法直接使用 `async` 函数的！这个会极大的限制了我们的使用场景
- 异步运行时本身需要初始化

因此 `#[tokio::main]` 宏在将 `async fn main` 隐式的转换为 `fn main` 的同时还对整个异步运行时进行了初始化。例如以下代码：

```
#[tokio::main]
async fn main() {
    println!("hello");
}
```

将被转换成：

```
fn main() {
    let mut rt = tokio::runtime::Runtime::new().unwrap();
    rt.block_on(async {
        println!("hello");
    })
}
```

最终，Rust 编译器就愉快地执行这段代码了。

## cargo feature

在引入 `tokio` 包时，我们在 `Cargo.toml` 文件中添加了这么一行：

```
tokio = { version = "1", features = ["full"] }
```

里面有个 `features = ["full"]` 可能大家会比较迷惑，当然，关于它的具体解释在本书的 [Cargo 专题](#) 有介绍，这里就简单进行说明。

`Tokio` 有很多功能和特性，例如 `TCP`，`UDP`，`Unix sockets`，同步工具，多调度类型等等，不是每个应用都需要所有的这些特性。为了优化编译时间和最终生成可执行文件大小、内存占用大小，应用可以对这些特性进行可选引入。

而这里为了演示的方便，我们使用 `full`，表示直接引入所有的特性。

## 总结

大家对 `tokio` 的初印象如何？可否 24 灯全亮通过？

总之，`tokio` 做的事情其实是细雨润无声的，在大多数时候，我们并不能感觉到它的存在，但是它确实是异步编程中最重要的一环(或者之一)，深入了解它对我们的未来之路会有莫大的帮助。

接下来，正式开始 `tokio` 的学习之旅。

# 创建异步任务

同志们，抓稳了，我们即将换挡提速，通向 `mini-redis` 服务端的高速之路已经开启。

不过在开始之前，先来做点收尾工作：上一章节中，我们实现了一个简易的 `mini-redis` 客户端并支持了 `SET / GET` 操作，现在将该[代码](#)移动到 `examples` 文件夹下，因为我们这个章节要实现的是服务器，后面可以通过运行 `example` 的方式，用之前客户端示例对我们的服务器端进行测试：

```
$ mkdir -p examples
$ mv src/main.rs examples/hello-redis.rs
```

并在 `Cargo.toml` 里添加 `[[example]]` 说明。关于 `example` 的详细说明，可以在[Cargo使用指南](#)里进一步了解。

```
[[example]]
name = "hello-redis"
path = "examples/hello-redis.rs"
```

然后再重新创建一个空的 `src/main.rs` 文件，至此替换文档已经完成，提速正式开始。

## 接收 sockets

作为服务器端，最基础的工作无疑是接收外部进来的 TCP 连接，可以通过 `tokio::net::TcpListener` 来完成。

---

Tokio 中大多数类型的名称都和标准库中对应的同步类型名称相同，而且，如果没有特殊原因，Tokio 的 API 名称也和标准库保持一致，只不过用 `async fn` 取代 `fn` 来声明函数。

---

`TcpListener` 监听 **6379** 端口，然后通过循环来接收外部进来的连接，每个连接在处理完后会被关闭。对于目前来说，我们的任务很简单：读取命令、打印到标准输出 `stdout`，最后回复给客户端一个错误。

```

use tokio::net::{TcpListener, TcpStream};
use mini_redis::{Connection, Frame};

#[tokio::main]
async fn main() {
    // Bind the listener to the address
    // 监听指定地址，等待 TCP 连接进来
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        // 第二个被忽略的项中包含有新连接的 `IP` 和端口信息
        let (socket, _) = listener.accept().await.unwrap();
        process(socket).await;
    }
}

async fn process(socket: TcpStream) {
    // `Connection` 对于 redis 的读写进行了抽象封装，因此我们读到的是一个一个数据帧frame(数据帧
    // = redis命令 + 数据)，而不是字节流
    // `Connection` 是在 mini-redis 中定义
    let mut connection = Connection::new(socket);

    if let Some(frame) = connection.read_frame().await.unwrap() {
        println!("GOT: {:?}", frame);

        // 回复一个错误
        let response = Frame::Error("unimplemented".to_string());
        connection.write_frame(&response).await.unwrap();
    }
}

```

现在运行我们的简单服务器：

```
cargo run
```

此时服务器会处于循环等待以接收连接的状态，接下来在一个新的终端窗口中启动上一章节中的 redis 客户端，由于相关代码已经放入 examples 文件夹下，因此我们可以使用 --example 来指定运行该客户端示例：

```
$ cargo run --example hello-redis
```

此时，客户端的输出是：Error: "unimplemented"，同时服务器端打印出了客户端发来的由 **redis 命令和数据** 组成的数据帧：GOT: Array([Bulk(b"set"), Bulk(b"hello"), Bulk(b"world")])。

# 生成任务

上面的服务器，如果你仔细看，它其实一次只能接受和处理一条 TCP 连接，只有等当前的处理完并结束后，才能开始接收下一条连接。原因在于 `loop` 循环中的 `await` 会导致当前任务进入阻塞等待，也就是 `loop` 循环会被阻塞。

而这显然不是我们想要的，服务器能并发地处理多条连接的请求，才是正确的打开姿势，下面来看看如何实现真正的并发。

---

关于并发和并行，在[多线程章节](#)中有详细的解释

---

为了并发的处理连接，需要为每一条进来的连接都生成(`spawn`)一个新的任务，然后在该任务中处理连接：

```
use tokio::net::TcpListener;

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        // 为每一条连接都生成一个新的任务,
        // `socket` 的所有权将被移动到新的任务中，并在那里进行处理
        tokio::spawn(async move {
            process(socket).await;
        });
    }
}
```

## 任务

一个 Tokio 任务是一个异步的绿色线程，它们通过 `tokio::spawn` 进行创建，该函数会返回一个 `JoinHandle` 类型的句柄，调用者可以使用该句柄跟创建的任务进行交互。

`spawn` 函数的参数是一个 `async` 语句块，该语句块甚至可以返回一个值，然后调用者可以通过 `JoinHandle` 句柄获取该值：

```
#[tokio::main]
async fn main() {
    let handle = tokio::spawn(async {
        10086
    });

    let out = handle.await.unwrap();
    println!("GOT {}", out);
}
```

以上代码会打印出 GOT 10086。实际上，上面代码中 .await 会返回一个 Result，若 spawn 创建的任务正常运行结束，则返回一个 ok( $T$ ) 的值，否则会返回一个错误 Err：例如任务内部发生了 panic 或任务因为运行时关闭被强制取消时。

任务是调度器管理的执行单元。spawn 生成的任务会首先提交给调度器，然后由它负责调度执行。需要注意的是，执行任务的线程未必是创建任务的线程，任务完全有可能运行在另一个不同的线程上，而且任务在生成后，它还可能会在线程间被移动。

任务在 Tokio 中远比看上去要更轻量，例如创建一个任务仅仅需要一次 64 字节大小的内存分配。因此应用程序在生成任务上，完全不应该有任何心理负担，除非你在一台没那么好的机器上疯狂生成了几百万个任务。。。

## 'static 约束

当使用 Tokio 创建一个任务时，该任务类型的生命周期必须是 'static。意味着，在任务中不能使用外部数据的引用：

```
use tokio::task;

#[tokio::main]
async fn main() {
    let v = vec![1, 2, 3];

    task::spawn(async {
        println!("Here's a vec: {:?}", v);
    });
}
```

上面代码中，spawn 出的任务引用了外部环境中的变量 v，导致以下报错：

```
error[E0373]: async block may outlive the current function, but
    it borrows `v`, which is owned by the current function
--> src/main.rs:7:23
|
7 |     task::spawn(async {
|     |-----^
|     |         println!("Here's a vec: {:?}", v);
|     |             ^ `v` is borrowed here
|     |
|     |     });
|     |-----^ may outlive borrowed value `v`
|
note: function requires argument type to outlive ``static``
--> src/main.rs:7:17
|
7 |     task::spawn(async {
|     |-----^
|     |         println!("Here's a vector: {:?}", v);
|     |     });
|     |-----^
help: to force the async block to take ownership of `v` (and any other
referenced variables), use the `move` keyword
|
7 |     task::spawn(async move {
8 |         println!("Here's a vec: {:?}", v);
9 |     });
|
```

原因在于：默认情况下，变量并不是通过 `move` 的方式转移进 `async` 语句块的，`v` 变量的所有权依然属于 `main` 函数，因为任务内部的 `println!` 是通过借用的方式使用了 `v`，但是这种借用并不能满足 `'static` 生命周期的要求。

在报错的同时，Rust 编译器还给出了相当有帮助的提示：为 `async` 语句块使用 `move` 关键字，这样就能将 `v` 的所有权从 `main` 函数转移到新创建的任务中。

但是 `move` 有一个问题，一个数据只能被一个任务使用，如果想要多个任务使用一个数据，就有些强人所难。不知道还有多少同学记得 `Arc`，它可以轻松解决该问题，还是线程安全的。

在上面的报错中，还有一句很奇怪的信息 `function requires argument type to outlive ``static```，函数要求参数类型的生命周期必须比 `'static` 长，问题是 `'static` 已经活得跟整个程序一样久了，难道函数的参数还能活得更久？大家可能会觉得编译器秀逗了，毕竟其它语言编译器也有秀逗的时候：)

先别急着给它扣帽子，虽然我有时候也想这么做。。原因是它说的是类型必须活得比 `'static` 长，而不是值。当我们说一个值是 `'static` 时，意味着它将永远存活。这个很重要，因为编译器无法知道新创建的任务将存活多久，所以唯一的办法就是让任务永远存活。

如果大家对于 `&'static` 和 `T: 'static` 较为模糊，强烈建议回顾下[该章节](#)。

## Send 约束

`tokio::spawn` 生成的任务必须实现 `Send` 特征，因为当这些任务在 `.await` 执行过程中发生阻塞时，Tokio 调度器会将任务在线程间移动。

**一个任务要实现 `Send` 特征，那它在 `.await` 调用的过程中所持有的全部数据都必须实现 `Send` 特征。**当 `.await` 调用发生阻塞时，任务会让出当前线程所有权给调度器，然后当任务准备好后，调度器会从上一次暂停的位置继续执行该任务。该流程能正确的工作，任务必须将 `.await` 之后使用的所有状态保存起来，这样才能在中断后恢复现场并继续执行。若这些状态实现了 `Send` 特征(可以在线程间安全地移动)，那任务自然也就可以在线程间安全地移动。

例如以下代码可以工作：

```
use tokio::task::yield_now;
use std::rc::Rc;

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        // 语句块的使用强制了 `rc` 会在 `await` 被调用前就被释放,
        // 因此 `rc` 并不会影响 `await` 的安全性
        {
            let rc = Rc::new("hello");
            println!("{}" , rc);
        }

        // `rc` 的作用范围已经失效，因此当任务让出所有权给当前线程时，它无需作为状态被保存起来
        yield_now().await;
    });
}
```

但是下面代码就不行：

```
use tokio::task::yield_now;
use std::rc::Rc;

#[tokio::main]
async fn main() {
    tokio::spawn(async {
        let rc = Rc::new("hello");

        // `rc` 在 `.await` 后还被继续使用，因此它必须被作为任务的状态保存起来
        yield_now().await;

        // 事实上，注释掉下面一行代码，依然会报错
        // 原因是：是否保存，不取决于 `rc` 是否被使用，而是取决于 `.await` 在调用时是否仍然处于
        // `rc` 的作用域中
        println!("{}", rc);

        // rc 作用域在这里结束
    });
}
```

这里有一个很重要的点，代码注释里有讲到，但是我们再重复一次：`rc` 是否会保存到任务状态中，取决于 `.await` 的调用是否处于它的作用域中，上面代码中，就算你注释掉 `println!` 函数，该报错依然会报错，因为 `rc` 的作用域直到 `async` 的末尾才结束！

下面是相应的报错，在下一章节，我们还会继续深入讨论该错误：

```
error: future cannot be sent between threads safely
--> src/main.rs:6:5
6 |     tokio::spawn(async {
|         ^^^^^^^^^^^^ future created by async block is not `Send`
|
|         ::: [..]spawn.rs:127:21
|
127 |             T: Future + Send + 'static,
|                 ----- required by this bound in
|                 `tokio::task::spawn::spawn`
|
| = help: within `impl std::future::Future`, the trait
|       `std::marker::Send` is not implemented for
|       `std::rc::Rc<&str>`
note: future is not `Send` as this value is used across an await
--> src/main.rs:10:9
|
7 |         let rc = Rc::new("hello");
|             -- has type `std::rc::Rc<&str>` which is not `Send`
...
10 |         yield_now().await;
|             ^^^^^^^^^^^^^^^^^^ await occurs here, with `rc` maybe
|                           used later
11 |         println!("{}", rc);
12 |     });
|         - `rc` is later dropped here
```

## 使用 HashMap 存储数据

现在，我们可以继续前进了，下面来实现 `process` 函数，它用于处理进入的命令。相应的值将被存储在 `HashMap` 中：通过 `SET` 命令存值，通过 `GET` 命令来取值。

同时，我们将使用循环的方式在同一个客户端连接中处理多次连续的请求：

```

use tokio::net::TcpStream;
use mini_redis::{Connection, Frame};

async fn process(socket: TcpStream) {
    use mini_redis::Command::{self, Get, Set};
    use std::collections::HashMap;

    // 使用 hashmap 来存储 redis 的数据
    let mut db = HashMap::new();

    // `mini-redis` 提供的便利函数，使用返回的 `connection` 可以用于从 socket 中读取数据并解析
    // 为数据帧
    let mut connection = Connection::new(socket);

    // 使用 `read_frame` 方法从连接获取一个数据帧：一条redis命令 + 相应的数据
    while let Some(frame) = connection.read_frame().await.unwrap() {
        let response = match Command::from_frame(frame).unwrap() {
            Set(cmd) => {
                // 值被存储为 `Vec<u8>` 的形式
                db.insert(cmd.key().to_string(), cmd.value().to_vec());
                Frame::Simple("OK".to_string())
            }
            Get(cmd) => {
                if let Some(value) = db.get(cmd.key()) {
                    // `Frame::Bulk` 期待数据的类型是 `Bytes`，该类型会在后面章节讲解，
                    // 此时，你只要知道 `&Vec<u8>` 可以使用 `into()` 方法转换成 `Bytes` 类
                    // 型
                    Frame::Bulk(value.clone().into())
                } else {
                    Frame::Null
                }
            }
            cmd => panic!("unimplemented {:?}", cmd),
        };
        // 将请求响应返回给客户端
        connection.write_frame(&response).await.unwrap();
    }
}

// main 函数在之前已实现

```

使用 cargo run 运行服务器，然后再打开另一个终端窗口，运行 hello-redis 客户端示例：cargo run --example hello-redis。

Bingo，在看了这么多原理后，我们终于迈出了小小的第一步，并获取到了存在 HashMap 中的值：从服务器端获取到结果=Some(b"world")。

但是问题又来了：这些值无法在 TCP 连接中共享，如果另外一个用户连接上来并试图同时获取 `hello` 这个 `key`，他将一无所获。

# 共享状态

上一章节中，咱们搭建了一个异步的 redis 服务器，并成功的提供了服务，但是其隐藏了一个巨大的问题：状态(数据)无法在多个连接之间共享，下面一起来看看该如何解决。

## 解决方法

好在 Tokio 十分强大，上面问题对应的解决方法也不止一种：

- 使用 `Mutex` 来保护数据的共享访问
- 生成一个异步任务去管理状态，然后各个连接使用消息传递的方式与其进行交互

其中，第一种方法适合比较简单的数据，而第二种方法适用于需要异步工作的，例如 I/O 原语。由于我们使用的数据存储类型是 `HashMap`，使用到的相关操作是 `insert` 和 `get`，又因为这两个操作都不是异步的，因此只要使用 `Mutex` 即可解决问题。

在上面的描述中，说实话第二种方法及其适用的场景并不是很好理解，但没关系，在后面章节会进行详细介绍。

## 添加 bytes 依赖包

在上一节中，我们使用 `Vec<u8>` 来保存目标数据，但是它有一个问题，对它进行克隆时会将底层数据也整个复制一份，效率很低，但是克隆操作对于我们在多连接间共享数据又是必不可少的。

因此这里咱们新引入一个 `bytes` 包，它包含一个 `Bytes` 类型，当对该类型的值进行克隆时，就不再会克隆底层数据。事实上，`Bytes` 是一个引用计数类型，跟 `Arc` 非常类似，或者准确的说，`Bytes` 就是基于 `Arc` 实现的，但相比后者 `Bytes` 提供了一些额外的能力。

在 `Cargo.toml` 的 `[dependencies]` 中引入 `bytes`：

```
bytes = "1"
```

## 初始化 HashMap

由于 `HashMap` 会在多个任务甚至多个线程间共享，再结合之前的选择，最终我们决定使用 `Arc<Mutex<T>>` 的方式对其进行包裹。

但是，大家先来畅想一下使用它进行包裹后的类型长什么样？大概，可能，长这样：

`Arc<Mutex<HashMap<String, Bytes>>>`，天哪噜，一不小心，你就遇到了 Rust 的阴暗面：类型大串烧。可以想象，如果要在代码中到处使用这样的类型，可读性会极速下降，因此我们需要一个**类型别名**(`type alias`)来简化下：

```
use bytes::Bytes;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

type Db = Arc<Mutex<HashMap<String, Bytes>>>;
```

此时，`Db` 就是一个类型别名，使用它就可以替代那一大串的东东，等下你就能看到功效。

接着，我们需要在 `main` 函数中对 `HashMap` 进行初始化，然后使用 `Arc` 克隆一份它的所有权并将其传入到生成的异步任务中。事实上在 Tokio 中，这里的 `Arc` 被称为 **handle**，或者更宽泛的说，`handle` 在 Tokio 中可以用来访问某个共享状态。

```
use tokio::net::TcpListener;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    println!("Listening");

    let db = Arc::new(Mutex::new(HashMap::new()));

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        // 将 handle 克隆一份
        let db = db.clone();

        println!("Accepted");
        tokio::spawn(async move {
            process(socket, db).await;
        });
    }
}
```

## 为何使用 `std::sync::Mutex`

上面代码还有一点非常重要，那就是我们使用了 `std::sync::Mutex` 来保护 `HashMap`，而不是使用 `tokio::sync::Mutex`。

在使用 Tokio 编写异步代码时，一个常见的错误无条件地使用 `tokio::sync::Mutex`，而真相是：Tokio 提供的异步锁只应该在跨多个 `.await` 调用时使用，而且 Tokio 的 `Mutex` 实际上内部使用的也是 `std::sync::Mutex`。

多补充几句，在异步代码中，关于锁的使用有以下经验之谈：

- 锁如果在多个 `.await` 过程中持有，应该使用 Tokio 提供的锁，原因是 `.await` 的过程中锁可能在线程间转移，若使用标准库的同步锁存在死锁的可能性，例如某个任务刚获取完锁，还没使用完就因为 `.await` 让出了当前线程的所有权，结果下个任务又去获取了锁，造成死锁
- 锁竞争不多的情况下，使用 `std::sync::Mutex`
- 锁竞争多，可以考虑使用三方库提供的性能更高的锁，例如 [parking\\_lot::Mutex](#)

## 更新 `process()`

`process()` 函数不再初始化 `HashMap`，取而代之的是它使用了 `HashMap` 的一个 `handle` 作为参数：

```

use tokio::net::TcpStream;
use mini_redis::{Connection, Frame};

async fn process(socket: TcpStream, db: Db) {
    use mini_redis::Command::{self, Get, Set};

    let mut connection = Connection::new(socket);

    while let Some(frame) = connection.read_frame().await.unwrap() {
        let response = match Command::from_frame(frame).unwrap() {
            Set(cmd) => {
                let mut db = db.lock().unwrap();
                db.insert(cmd.key().to_string(), cmd.value().clone());
                Frame::Simple("OK".to_string())
            }
            Get(cmd) => {
                let db = db.lock().unwrap();
                if let Some(value) = db.get(cmd.key()) {
                    Frame::Bulk(value.clone())
                } else {
                    Frame::Null
                }
            }
            cmd => panic!("unimplemented {:?}", cmd),
        };
        connection.write_frame(&response).await.unwrap();
    }
}

```

## 任务、线程和锁竞争

当竞争不多的时候，使用阻塞性的锁去保护共享数据是一个正确的选择。当一个锁竞争触发后，当前正在执行任务(请求锁)的线程会被阻塞，并等待锁被前一个使用者释放。这里的关键就是：**锁竞争不仅仅会导致当前的任务被阻塞，还会导致执行任务的线程被阻塞，因此该线程准备执行的其它任务也会因此被阻塞！**

默认情况下，Tokio 调度器使用了多线程模式，此时如果有大量的任务都需要访问同一个锁，那么锁竞争将变得激烈起来。当然，你也可以使用 **current\_thread** 运行时设置，在该设置下会使用一个单线程的调度器(执行器)，所有的任务都会创建并执行在当前线程上，因此不再会有锁竞争。

---

`current_thread` 是一个轻量级、单线程的运行时，当任务数不多或连接数不多时是一个很好的选择。例如你想在一个异步客户端库的基础上提供给用户同步的 API 访问时，该模式就很适用

---

当同步锁的竞争变成一个问题时，使用 Tokio 提供的异步锁几乎并不能帮你解决问题，此时可以考虑如下选项：

- 创建专门的任务并使用消息传递的方式来管理状态
- 将锁进行分片
- 重构代码以避免锁

在我们的例子中，由于每一个 `key` 都是独立的，因此对锁进行分片将成为一个不错的选择：

```
type ShardedDb = Arc<Vec<Mutex<HashMap<String, Vec<u8>>>>;  
  
fn new_sharded_db(num_shards: usize) -> ShardedDb {  
    let mut db = Vec::with_capacity(num_shards);  
    for _ in 0..num_shards {  
        db.push(Mutex::new(HashMap::new()));  
    }  
    Arc::new(db)  
}
```

在这里，我们创建了 N 个不同的存储实例，每个实例都会存储不同的分片数据，例如我们有 a-i 共 9 个不同的 `key`，可以将存储分成 3 个实例，那么第一个实例可以存储 a-c，第二个 d-f，以此类推。在这种情况下，访问 `b` 时，只需要锁住第一个实例，此时二、三实例依然可以正常访问，因此锁被成功的分片了。

在分片后，使用给定的 `key` 找到对应的值就变成了两个步骤：首先，使用 `key` 通过特定的算法寻找到对应的分片，然后再使用该 `key` 从分片中查询到值：

```
let shard = db[hash(key) % db.len()].lock().unwrap();  
shard.insert(key, value);
```

这里我们使用 `hash` 算法来进行分片，但是该算法有个缺陷：分片的数量不能变，一旦变了后，那之前落入分片 1 的 `key` 很可能将落入到其它分片中，最终全部乱掉。此时你可以考虑[dashmap](#)，它提供了更复杂、更精妙的支持分片的 `hash map`。

## 在 `.await` 期间持有锁

在某些时候，你可能会不经意写下这种代码：

```

use std::sync::{Mutex, MutexGuard};

async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    let mut lock: MutexGuard<i32> = mutex.lock().unwrap();
    *lock += 1;

    do_something_async().await;
} // 锁在这里超出作用域

```

如果你要 `spawn` 一个任务来执行上面的函数的话，会报错：

```

error: future cannot be sent between threads safely
--> src/lib.rs:13:5
|
13 |     tokio::spawn(async move {
|     ^^^^^^^^^^^^^ future created by async block is not `Send`
|
::: /playground/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-
0.2.21/src/task/spawn.rs:127:21
|
127 |         T: Future + Send + 'static,
|             ----- required by this bound in `tokio::task::spawn::spawn`  

|             = help: within `impl std::future::Future`, the trait `std::marker::Send` is not
| implemented for `std::sync::MutexGuard<'_, i32>`
note: future is not `Send` as this value is used across an await
--> src/lib.rs:7:5
|
4 |     let mut lock: MutexGuard<i32> = mutex.lock().unwrap();
|             ----- has type `std::sync::MutexGuard<'_, i32>` which is not `Send`  

...
7 |     do_something_async().await;
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^ await occurs here, with `mut lock` maybe used
later
8 | }
| - `mut lock` is later dropped here

```

错误的原因在于 `std::sync::MutexGuard` 类型并没有实现 `Send` 特征，这意味着你不能将一个 `Mutex` 锁发送到另一个线程，因为 `.await` 可能会让任务转移到另一个线程上执行，这个之前也介绍过。

## 提前释放锁

要解决这个问题，就必须重构代码，让 `Mutex` 锁在 `.await` 被调用前就被释放掉。

```
// 下面的代码可以工作!
async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    {
        let mut lock: MutexGuard<i32> = mutex.lock().unwrap();
        *lock += 1;
    } // lock在这里超出作用域（被释放）

    do_something_async().await;
}
```

---

大家可能已经发现，很多错误都是因为 `.await` 引起的，其实你只要记住，在 `.await` 执行期间，任务可能会在线程间转移，那么这些错误将变得很好理解，不必去死记硬背

但是下面的代码不工作：

```
use std::sync::{Mutex, MutexGuard};

async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    let mut lock: MutexGuard<i32> = mutex.lock().unwrap();
    *lock += 1;
    drop(lock);

    do_something_async().await;
}
```

原因我们之前解释过，编译器在这里不够聪明，目前它只能根据作用域的范围来判断，`drop` 虽然释放了锁，但是锁的作用域依然会持续到函数的结束，未来也许编译器会改进，但是现在至少还是不行的。

聪明的读者此时的小脑袋已经飞速运转起来，既然锁没有实现 `Send`，那我们主动给它实现如何？这样不就可以顺利运行了吗？答案依然是不可以，原因就是我们之前提到过的死锁，如果一个任务获取了锁，然后还没释放就在 `.await` 期间被挂起，接着开始执行另一个任务，这个任务又去获取锁，就会导致死锁。

再来看看其它解决方法：

### 重构代码：在 `.await` 期间不持有锁

之前的代码其实也是为了在 `.await` 期间不持有锁，但是我们还有更好的实现方式，例如，你可以把 `Mutex` 放入一个结构体中，并且只在该结构体的非异步方法中使用该锁：

```
use std::sync::Mutex;

struct CanIncrement {
    mutex: Mutex<i32>,
}

impl CanIncrement {
    // 该方法不是 `async` 
    fn increment(&self) {
        let mut lock = self.mutex.lock().unwrap();
        *lock += 1;
    }
}

async fn increment_and_do_stuff(can_incr: &CanIncrement) {
    can_incr.increment();
    do_something_async().await;
}
```

## 使用异步任务和通过消息传递来管理状态

该方法常常用于共享的资源是 I/O 类型的资源时，我们在下一章节将详细介绍。

## 使用 Tokio 提供的异步锁

Tokio 提供的锁最大的优点就是：它可以在 `.await` 执行期间被持有，而且不会有任何问题。但是代价就是，这种异步锁的性能开销会更高，因此如果可以，使用之前的两种方法来解决会更好。

```
use tokio::sync::Mutex; // 注意，这里使用的是 Tokio 提供的锁

// 下面的代码会编译
// 但是就这个例子而言，之前的方式会更好
async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    let mut lock = mutex.lock().await;
    *lock += 1;

    do_something_async().await;
} // 锁在这里被释放
```

# 消息传递

迄今为止，你已经学了不少关于 Tokio 的并发编程的内容，是时候见识下真正的挑战了，接下来，我们一起来实现下客户端这块儿的功能。

首先，将之前实现的 `src/main.rs` 文件中的[服务器端代码](#)放入到一个 bin 文件中，等下可以直接通过该文件来运行我们的服务器：

```
mkdir src/bin  
mv src/main.rs src/bin/server.rs
```

接着创建一个新的 bin 文件，用于包含我们即将实现的客户端代码：

```
touch src/bin/client.rs
```

由于不再使用 `main.rs` 作为程序入口，我们需要使用以下命令来运行指定的 bin 文件：

```
cargo run --bin server
```

此时，服务器已经成功运行起来。同样的，可以用 `cargo run --bin client` 这种方式运行即将实现的客户端。

万事俱备，只欠代码，一起来看看客户端该如何实现。

## 错误的实现

如果想要同时运行两个 redis 命令，我们可能会为每一个命令生成一个任务，例如：

```

use mini_redis::client;

#[tokio::main]
async fn main() {
    // 创建到服务器的连接
    let mut client = client::connect("127.0.0.1:6379").await.unwrap();

    // 生成两个任务，一个用于获取 key，一个用于设置 key
    let t1 = tokio::spawn(async {
        let res = client.get("hello").await;
    });

    let t2 = tokio::spawn(async {
        client.set("foo", "bar".into()).await;
    });

    t1.await.unwrap();
    t2.await.unwrap();
}

```

这段代码不会编译，因为两个任务都需要去访问 `client`，但是 `client` 并没有实现 `Copy` 特征，再加上我们并没有实现相应的共享代码，因此自然会报错。还有一个问题，方法 `set` 和 `get` 都使用了 `client` 的可变引用 `&mut self`，由此还会造成同时借用两个可变引用的错误。

在上一节中，我们介绍了几个解决方法，但是它们大部分都不太适用于此时的情况，例如：

- `std::sync::Mutex` 无法被使用，这个问题在之前章节有详解介绍过，同步锁无法跨越 `.await` 调用时使用
- 那么你可能会想，是不是可以使用 `tokio::sync::Mutex`，答案是可以用，但是同时就只能运行一个请求。若客户端实现了 redis 的 `pipelining`，那这个异步锁就会导致连接利用率不足

这个不行，那个也不行，是不是没有办法解决了？还记得我们上一章节提到过几次的消息传递，但是一直没有看到它的庐山真面目吗？现在可以来看看。

## 消息传递

之前章节我们提到可以创建一个专门的任务 `c1` (消费者 Consumer) 和通过消息传递来管理共享的资源，这里的共享资源就是 `client`。若任务 `p1` (生产者 Producer) 想要发出 Redis 请求，首先需要发送信息给 `c1`，然后 `c1` 会发出请求给服务器，在获取到结果后，再将结果返回给 `p1`。

在这种模式下，只需要建立一条连接，然后由一个统一的任务来管理 `client` 和该连接，这样之前的 `get` 和 `set` 请求也将不存在资源共享的问题。

同时，`P1` 和 `C1` 进行通信的消息通道是有缓冲的，当大量的消息发送给 `C1` 时，首先会放入消息通道的缓冲区中，当 `C1` 处理完一条消息后，再从该缓冲区中取出下一条消息进行处理，这种方式跟消息队列(Message queue)非常类似，可以实现更高的吞吐。而且这种方式还有利于实现连接池，例如不止一个 `P` 和 `C` 时，多个 `P` 可以往消息通道中发送消息，同时多个 `C`，其中每个 `C` 都维护一条连接，并从消息通道获取消息。

## Tokio 的消息通道( channel )

Tokio 提供了多种消息通道，可以满足不同场景的需求：

- `mpsc`，多生产者，单消费者模式
- `oneshot`，单生产者，单消费者，一次只能发送一条消息
- `broadcast`，多生产者，多消费者，其中每一条发送的消息都可以被所有接收者收到，因此是广播
- `watch`，单生产者，多消费者，只保存一条最新的消息，因此接收者只能看到最近的一条消息，例如，这种模式适用于配置文件变化的监听

细心的同学可能会发现，这里还少了一种类型：多生产者、多消费者，且每一条消息只能被其中一个消费者接收，如果有这种需求，可以使用 `async-channel` 包。

以上这些消息通道都有一个共同点：适用于 `async` 编程，对于其它场景，你可以使用在[多线程章节](#)中提到过的 `std::sync::mpsc` 和 `crossbeam::channel`，这些通道在等待消息时会阻塞当前的线程，因此不适用于 `async` 编程。

在下面的代码中，我们将使用 `mpsc` 和 `oneshot`，本章节完整的代码见[这里](#)。

## 定义消息类型

在大多数场景中使用消息传递时，都是多个发送者向一个任务发送消息，该任务在处理完后，需要将响应内容返回给相应的发送者。例如我们的例子中，任务需要将 `GET` 和 `SET` 命令处理的结果返回。首先，我们需要定一个 `Command` 枚举用于代表命令：

```
use bytes::Bytes;

#[derive(Debug)]
enum Command {
    Get {
        key: String,
    },
    Set {
        key: String,
        val: Bytes,
    }
}
```

## 创建消息通道

在 `src/bin/client.rs` 的 `main` 函数中，创建一个 `mpsc` 消息通道：

```
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    // 创建一个新通道，缓冲队列长度是 32
    let (tx, mut rx) = mpsc::channel(32);

    // ... 其它代码
}
```

一个任务可以通过此通道将命令发送给管理 redis 连接的任务，同时由于通道支持多个生产者，因此多个任务可以同时发送命令。创建该通道会返回一个发送和接收句柄，这两个句柄可以分别被使用，例如它们可以被移动到不同的任务中。

通道的缓冲队列长度是 32，意味着如果消息发送的比接收的快，这些消息将被存储在缓冲队列中，一旦存满了 32 条消息，使用 `send(...).await` 的发送者会**进入睡眠**，直到缓冲队列可以放入新的消息(被接收者消费了)。

```
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let tx2 = tx.clone();

    tokio::spawn(async move {
        tx.send("sending from first handle").await;
    });

    tokio::spawn(async move {
        tx2.send("sending from second handle").await;
    });

    while let Some(message) = rx.recv().await {
        println!("GOT = {}", message);
    }
}
```

你可以使用 `clone` 方法克隆多个发送者，但是接收者无法被克隆，因为我们的通道是 `mpsc` 类型。

当所有的发送者都被 `Drop` 掉后(超出作用域或被 `drop(...)` 函数主动释放)，就不再会有任何消息发送给该通道，此时 `recv` 方法将返回 `None`，也意味着该通道已经被关闭。

在我们的例子中，接收者是在管理 redis 连接的任务中，当该任务发现所有发送者都关闭时，它知道它的使命可以完成了，因此它会关闭 redis 连接。

## 生成管理任务

下面，我们来一起创建一个管理任务，它会管理 redis 的连接，当然，首先需要创建一条到 redis 的连接：

```

use mini_redis::client;
// 将消息通道接收者 rx 的所有权转移到管理任务中
let manager = tokio::spawn(async move {
    // Establish a connection to the server
    // 建立到 redis 服务器的连接
    let mut client = client::connect("127.0.0.1:6379").await.unwrap();

    // 开始接收消息
    while let Some(cmd) = rx.recv().await {
        use Command::*;

        match cmd {
            Get { key } => {
                client.get(&key).await;
            }
            Set { key, val } => {
                client.set(&key, val).await;
            }
        }
    }
});
```

如上所示，当从消息通道接收到一个命令时，该管理任务会将此命令通过 redis 连接发送到服务器。

现在，让两个任务发送命令到消息通道，而不是像最开始报错的那样，直接发送命令到各自的 redis 连接：

```

// 由于有两个任务，因此我们需要两个发送者
let tx2 = tx.clone();

// 生成两个任务，一个用于获取 key，一个用于设置 key
let t1 = tokio::spawn(async move {
    let cmd = Command::Get {
        key: "hello".to_string(),
    };

    tx.send(cmd).await.unwrap();
});

let t2 = tokio::spawn(async move {
    let cmd = Command::Set {
        key: "foo".to_string(),
        val: "bar".into(),
    };

    tx2.send(cmd).await.unwrap();
});
```

在 `main` 函数的末尾，我们让 3 个任务，按照需要的顺序开始运行：

```
t1.await.unwrap();
t2.await.unwrap();
manager.await.unwrap();
```

## 接收响应消息

最后一步，就是让发出命令的任务从管理任务那里获取命令执行的结果。为了完成这个目标，我们将使用 `oneshot` 消息通道，因为它针对一发一收的使用类型做过特别优化，且特别适用于此时的场景：接收一条从管理任务发送的结果消息。

```
use tokio::sync::oneshot;

let (tx, rx) = oneshot::channel();
```

使用方式跟 `mpsc` 很像，但是它并没有缓存长度，因为只能发送一条，接收一条，还有一点不同：你无法对返回的两个句柄进行 `clone`。

为了让管理任务将结果准确的返回到发送者手中，这个管道的发送端必须要随着命令一起发送，然后发出命令的任务保留管道的发送端。一个比较好的实现就是将管道的发送端放入 `Command` 的数据结构中，同时使用一个别名来代表该发送端：

```
use tokio::sync::oneshot;
use bytes::Bytes;

#[derive(Debug)]
enum Command {
    Get {
        key: String,
        resp: Responder<Option<Bytes>>,
    },
    Set {
        key: String,
        val: Bytes,
        resp: Responder<()>,
    },
}

/// 管理任务可以使用该发送端将命令执行的结果传回给发出命令的任务
type Responder<T> = oneshot::Sender<mini_redis::Result<T>>;
```

下面，更新发送命令的代码：

```

let t1 = tokio::spawn(async move {
    let (resp_tx, resp_rx) = oneshot::channel();
    let cmd = Command::Get {
        key: "hello".to_string(),
        resp: resp_tx,
    };

    // 发送 GET 请求
    tx.send(cmd).await.unwrap();

    // 等待回复
    let res = resp_rx.await;
    println!("GOT = {:?}", res);
});

let t2 = tokio::spawn(async move {
    let (resp_tx, resp_rx) = oneshot::channel();
    let cmd = Command::Set {
        key: "foo".to_string(),
        val: "bar".into(),
        resp: resp_tx,
    };

    // 发送 SET 请求
    tx2.send(cmd).await.unwrap();

    // 等待回复
    let res = resp_rx.await;
    println!("GOT = {:?}", res);
});

```

最后，更新管理任务：

```

while let Some(cmd) = rx.recv().await {
    match cmd {
        Command::Get { key, resp } => {
            let res = client.get(&key).await;
            // 忽略错误
            let _ = resp.send(res);
        }
        Command::Set { key, val, resp } => {
            let res = client.set(&key, val).await;
            // 忽略错误
            let _ = resp.send(res);
        }
    }
}

```

有一点值得注意，往 `oneshot` 中发送消息时，并没有使用 `.await`，原因是该发送操作要么直接成功、要么失败，并不需要等待。

当 `oneshot` 的接受端被 `drop` 后，继续发送消息会直接返回 `Err` 错误，它表示接收者已经不感兴趣了。对于我们的场景，接收者不感兴趣是非常合理的操作，并不是一种错误，因此可以直接忽略。

本章的完整代码见[这里](#)。

## 对消息通道进行限制

无论何时使用消息通道，我们都需要对缓存队列的长度进行限制，这样系统才能优雅的处理各种负载状况。如果不限制，假设接收端无法及时处理消息，那消息就会迅速堆积，最终可能会导致内存消耗殆尽，就算内存没有消耗完，也可能会导致整体性能的大幅下降。

Tokio 在设计时就考虑了这种状况，例如 `async` 操作在 Tokio 中是惰性的：

```
loop {
    async_op();
}
```

如果上面代码中，`async_op` 不是惰性的，而是在每次循环时立即执行，那该循环会立即将一个 `async_op` 发送到缓冲队列中，然后开始执行下一个循环，因为无需等待任务执行完成，这种发送速度是非常恐怖的，一秒钟可能会有几十万、上百万的消息发送到消息队列中。在其它语言编程中，相信大家或多或少遇到过这种情况。

然后在 `Async Rust` 和 Tokio 中，上面的代码 `async_op` 根本就不会运行，也就不会往消息队列中写入消息。原因是我们在没有调用 `.await`，就算使用了 `.await` 上面的代码也不会有问题，因为只有等当前循环的任务结束后，才会开始下一次循环。

```
loop {
    // 当前 `async_op` 完成后，才会开始下一次循环
    async_op().await;
}
```

总之，在 Tokio 中我们必须要显式地引入并发和队列：

- `tokio::spawn`
- `select!`
- `join!`
- `mpsc::channel`

当这么做时，我们需要小心的控制并发度来确保系统的安全。例如，当使用一个循环去接收 TCP 连接时，你要确保当前打开的 `socket` 数量在可控范围内，而不是毫无原则的接收连接。再比如，当使用 `mpsc::channel` 时，要设置一个缓冲值。

挑选一个合适的限制值是 Tokio 编程中很重要的一部分，可以帮助我们的系统更加安全、可靠的运行。

# I/O

本章节中我们将深入学习 Tokio 中的 I/O 操作，了解它的原理以及该如何使用。

Tokio 中的 I/O 操作和 std 在使用方式上几无区别，最大的区别就是前者是异步的，例如 Tokio 的读写特征分别是 `AsyncRead` 和 `AsyncWrite`：

- 有部分类型按照自己的所需实现了它们: `TcpStream`, `File`, `Stdout`
- 还有数据结构也实现了它们: `Vec<u8>`、`&[u8]`，这样就可以直接使用这些数据结构作为读写器(`reader / writer`)

## AsyncRead 和 AsyncWrite

这两个特征为字节流的异步读写提供了便利，通常我们会使用 `AsyncReadExt` 和 `AsyncWriteExt` 提供的工具方法，这些方法都使用 `async` 声明，且需要通过 `.await` 进行调用，

### async fn read

`AsyncReadExt::read` 是一个异步方法可以将数据读入缓冲区( `buffer` )中，然后返回读取的字节数。

```
use tokio::fs::File;
use tokio::io::{self, AsyncReadExt};

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
    let mut buffer = [0; 10];

    // 由于 buffer 的长度限制，当次的 `read` 调用最多可以从文件中读取 10 个字节的数据
    let n = f.read(&mut buffer[..]).await?;

    println!("The bytes: {:?}", &buffer[..n]);
    Ok(())
}
```

需要注意的是：当 `read` 返回 `Ok(0)` 时，意味着字节流( `stream` )已经关闭，在这之后继续调用 `read` 会立刻完成，依然获取到返回值 `Ok(0)`。例如，字节流如果是 `TcpStream` 类型，那 `Ok(0)` 说明该**连接的读取端已经被关闭**(写入端关闭，会报其它的错误)。

## async fn read\_to\_end

AsyncReadExt::read\_to\_end 方法会从字节流中读取所有的字节，直到遇到 EOF：

```
use tokio::io::{self, AsyncReadExt};
use tokio::fs::File;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
    let mut buffer = Vec::new();

    // 读取整个文件的内容
    f.read_to_end(&mut buffer).await?;
    Ok(())
}
```

## async fn write

AsyncWriteExt::write 异步方法会尝试将缓冲区的内容写入到写入器( writer )中，同时返回写入的字节数:

```
use tokio::io::{self, AsyncWriteExt};
use tokio::fs::File;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut file = File::create("foo.txt").await?;

    let n = file.write(b"some bytes").await?;

    println!("Wrote the first {} bytes of 'some bytes'.", n);
    Ok(())
}
```

上面代码很清晰，但是大家可能会疑惑 b"some bytes" 是什么意思。这种写法可以将一个 &str 字符串转变成一个字节数组: &[u8;10]，然后 write 方法又会将这个 &[u8;10] 的数组类型隐式强转为数组切片: &[u8]。

## async fn write\_all

AsyncWriteExt::write\_all 将缓冲区的内容全部写入到写入器中:

```
use tokio::io::{self, AsyncWriteExt};
use tokio::fs::File;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut file = File::create("foo.txt").await?;

    file.write_all(b"some bytes").await?;
    Ok(())
}
```

以上只是部分方法，实际上还有一些实用的方法由于篇幅有限无法列出，大家可以通过 [API 文档](#) 查看完整的列表。

## 实用函数

另外，和标准库一样，`tokio::io` 模块包含了多个实用的函数或 API，可以用于处理标准输入/输出/错误等。

例如，`tokio::io::copy` 异步的将读取器(`reader`)中的内容拷贝到写入器(`writer`)中。

```
use tokio::fs::File;
use tokio::io;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut reader: &[u8] = b"hello";
    let mut file = File::create("foo.txt").await?;

    io::copy(&mut reader, &mut file).await?;
    Ok(())
}
```

还记得我们之前提到的字节数组 `&[u8]` 实现了 `AsyncRead` 吗？正因为这个原因，所以这里可以直接将 `&u8` 用作读取器。

## 回声服务( Echo )

就如同写代码必写 `hello, world`，实现 web 服务器，往往会选择实现一个回声服务。该服务会将用户的输入内容直接返回给用户，就像回声壁一样。

具体来说，就是从用户建立的 TCP 连接的 socket 中读取到数据，然后立刻将同样的数据写回到该 socket 中。因此客户端会收到和自己发送的数据一模一样的回复。

下面我们将使用两种稍有不同的方法实现该回声服务。

### 使用 `io::copy()`

先来创建一个新的 bin 文件，用于运行我们的回声服务：

```
touch src/bin/echo-server-copy.rs
```

然后可以通过以下命令运行它(跟上一章节的方式相同)：

```
cargo run --bin echo-server-copy
```

至于客户端，可以简单的使用 `telnet` 的方式来连接，或者也可以使用 `tokio::net::TcpStream`，它的[文档示例](#)非常适合大家进行参考。

先来实现一下基本的服务器框架：通过 loop 循环接收 TCP 连接，然后为每一条连接创建一个单独的任务去处理。

```
use tokio::io;
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6142").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            // 在这里拷贝数据
        });
    }
}
```

下面，来看看重头戏 `io::copy`，它有两个参数：一个读取器，一个写入器，然后将读取器中的数据直接拷贝到写入器中，类似的实现代码如下：

```
io::copy(&mut socket, &mut socket).await
```

这段代码相信大家一眼就能看出问题，由于我们的读取器和写入器都是同一个 socket，因此需要对其进行两次可变借用，这明显违背了 Rust 的借用规则。

## 分离读写器

显然，使用同一个 socket 是不行的，为了实现目标功能，必须将 socket 分离成一个读取器和写入器。

任何一个读写器( reader + writer )都可以使用 `io::split` 方法进行分离，最终返回一个读取器和写入器，这两者可以独自的使用，例如可以放入不同的任务中。

例如，我们的回声客户端可以这样实现，以实现同时并发读写：

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpStream;

#[tokio::main]
async fn main() -> io::Result<()> {
    let socket = TcpStream::connect("127.0.0.1:6142").await?;
    let (mut rd, mut wr) = io::split(socket);

    // 创建异步任务，在后台写入数据
    tokio::spawn(async move {
        wr.write_all(b"hello\r\n").await?;
        wr.write_all(b"world\r\n").await?;

        // 有时，我们需要给予 Rust 一些类型暗示，它才能正确的推导出类型
        Ok::<_, io::Error>(())
    });

    let mut buf = vec![0; 128];

    loop {
        let n = rd.read(&mut buf).await?;

        if n == 0 {
            break;
        }

        println!("GOT {:?}", &buf[..n]);
    }

    Ok(())
}
```

实际上，`io::split` 可以用于任何同时实现了 `AsyncRead` 和 `AsyncWrite` 的值，它的内部使用了 `Arc` 和 `Mutex` 来实现相应的功能。如果大家觉得这种实现有些重，可以使用 Tokio 提供的 `TcpStream`，它提供了两种方式进行分离：

- `TcpStream::split` 会获取字节流的引用，然后将其分离成一个读取器和写入器。但由于使用了引用的方式，它们俩必须和 `split` 在同一个任务中。优点就是，这种实现没有性能开销，因为无需 `Arc` 和 `Mutex`。

- `TcpStream::into_split` 还提供了一种分离实现，分离出来的结果可以在任务间移动，内部是通过 `Arc` 实现

再来分析下我们的使用场景，由于 `io::copy()` 调用时所在的任务和 `split` 所在的任务是同一个，因此可以使用性能最高的 `TcpStream::split`：

```
tokio::spawn(async move {
    let (mut rd, mut wr) = socket.split();

    if io::copy(&mut rd, &mut wr).await.is_err() {
        eprintln!("failed to copy");
    }
});
```

使用 `io::copy` 实现的完整代码见[此处](#)。

## 手动拷贝

程序员往往拥有一颗手动干翻一切的心，因此如果你不想用 `io::copy` 来简单实现，还可以自己手动去拷贝数据：

```

use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:6142").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buf = vec![0; 1024];

            loop {
                match socket.read(&mut buf).await {
                    // 返回值 `Ok(0)` 说明对端已经关闭
                    Ok(0) => return,
                    Ok(n) => {
                        // Copy the data back to socket
                        // 将数据拷贝回 socket 中
                        if socket.write_all(&buf[..n]).await.is_err() {
                            // 非预期错误，由于我们这里无需再做什么，因此直接停止处理
                            return;
                        }
                    }
                    Err(_) => {
                        // 非预期错误，由于我们无需再做什么，因此直接停止处理
                        return;
                    }
                }
            }
        });
    }
}

```

建议这段代码放入一个和之前 `io::copy` 不同的文件中 `src/bin/echo-server.rs`，然后使用 `cargo run --bin echo-server` 运行。

下面一起来看看这段代码有哪些值得注意的地方。首先，由于使用了 `write_all` 和 `read` 方法，需要先将对应的特征引入到当前作用域内：

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
```

### 在堆上分配缓冲区

在上面代码中，我们需要将数据从 `socket` 中读取到一个缓冲区 `buffer` 中：

```
let mut buf = vec![0; 1024];
```

可以看到，此处的缓冲区是一个 `Vec` 动态数组，它的数据是存储在堆上，而不是栈上(若改成 `let mut buf = [0; 1024];`，则存储在栈上)。

在此之前，我们提到过一个数据如果想在 `.await` 调用过程中存在，那它必须存储在当前任务内。在我们的代码中，`buf` 会在 `.await` 调用过程中被使用，因此它必须要存储在任务内。

若该缓冲区数组创建在栈上，那每条连接所对应的任务的内部数据结构看上去可能如下所示：

```
struct Task {  
    task: enum {  
        AwaitingRead {  
            socket: TcpStream,  
            buf: [BufferType],  
        },  
        AwaitingWriteAll {  
            socket: TcpStream,  
            buf: [BufferType],  
        }  
    }  
}
```

可以看到，栈数组要被使用，就必须存储在相应的结构体内，其中两个结构体分别持有了不同的栈数组 `[BufferType]`，这种方式会导致任务结构变得很大。特别地，我们选择缓冲区长度往往会选择分页长度 (page size)，因此使用栈数组会导致任务的内存大小变得很奇怪甚至糟糕：`$page-size + 一些额外的字节`。

当然，编译器会帮助我们做一些优化。例如，会进一步优化 `async` 语句块的布局，而不是像上面一样简单的使用 `enum`。在实践中，变量也不会在枚举成员间移动。

但是再怎么优化，任务的结构体至少也会跟其中的栈数组一样大，因此通常情况下，使用堆上的缓冲区会高效实用的多。

---

当任务因为调度在线程间移动时，存储在栈上的数据需要进行保存和恢复，过大的栈上变量会带来不小的数据拷贝开销

因此，存储大量数据的变量最好放到堆上

---

## 处理 EOF

当 TCP 连接的读取端关闭后，再调用 `read` 方法会返回 `Ok(0)`。此时，再继续下去已经没有意义，因此我们需要退出循环。忘记在 EOF 时退出读取循环，是网络编程中一个常见的 bug：

```
loop {
    match socket.read(&mut buf).await {
        Ok(0) => return,
        // ... 其余错误处理
    }
}
```

大家不妨深入思考下，如果没有退出循环会怎么样？之前我们提到过，一旦读取端关闭后，那后面的 `read` 调用就会立即返回 `Ok(0)`，而不会阻塞等待，因此这种无阻塞循环会最终导致 CPU 立刻跑到 100%，并将一直持续下去，直到程序关闭。

# 解析数据帧

现在，鉴于大家已经掌握了 Tokio 的基本 I/O 用法，我们可以开始实现 `mini-redis` 的帧 `frame`。通过帧可以将字节流转换成帧组成的流。每个帧就是一个数据单元，例如客户端发送的一次请求就是一个帧。

```
use bytes::Bytes;

enum Frame {
    Simple(String),
    Error(String),
    Integer(u64),
    Bulk(Bytes),
    Null,
    Array(Vec<Frame>),
}
```

可以看到帧除了数据之外，并不具备任何语义。命令解析和实现会在更高的层次进行(相比帧解析层)。我们再来通过 HTTP 的帧来帮大家加深下相关的理解：

```
enum HttpFrame {
    RequestHead {
        method: Method,
        uri: Uri,
        version: Version,
        headers: HeaderMap,
    },
    ResponseHead {
        status: StatusCode,
        version: Version,
        headers: HeaderMap,
    },
    BodyChunk {
        chunk: Bytes,
    },
}
```

为了实现 `mini-redis` 的帧，我们需要一个 `Connection` 结构体，里面包含了一个 `TcpStream` 以及对帧进行读写的方法：

```

use tokio::net::TcpStream;
use mini_redis::{Frame, Result};

struct Connection {
    stream: TcpStream,
    // ... 这里定义其它字段
}

impl Connection {
    /// 从连接读取一个帧
    ///
    /// 如果遇到EOF, 则返回 None
    pub async fn read_frame(&mut self)
        -> Result<Option<Frame>>
    {
        // 具体实现
    }

    /// 将帧写入到连接中
    pub async fn write_frame(&mut self, frame: &Frame)
        -> Result<()>
    {
        // 具体实现
    }
}

```

关于 Redis 协议的说明, 可以看看[官方文档](#), Connection 代码的完整实现见[这里](#).

## 缓冲读取(Buffered Reads)

`read_frame` 方法会等到一个完整的帧都读取完毕后才返回, 与之相比, 它底层调用的 `TcpStream::read` 只会返回任意多的数据(填满传入的缓冲区 `buffer`), 它可能返回帧的一部分、一个帧、多个帧, 总之这种读取行为是不确定的。

当 `read_frame` 的底层调用 `TcpStream::read` 读取到部分帧时, 会将数据先缓冲起来, 接着继续等待并读取数据。如果读到多个帧, 那第一个帧会被返回, 然后剩下的数据依然被缓冲起来, 等待下一次 `read_frame` 被调用。

为了实现这种功能, 我们需要为 `Connection` 增加一个读取缓冲区。数据首先从 `socket` 中读取到缓冲区中, 接着这些数据会被解析为帧, 当一个帧被解析后, 该帧对应的数据会从缓冲区被移除。

这里使用 `BytesMut` 作为缓冲区类型, 它是 `Bytes` 的可变版本。

```

use bytes::BytesMut;
use tokio::net::TcpStream;

pub struct Connection {
    stream: TcpStream,
    buffer: BytesMut,
}

impl Connection {
    pub fn new(stream: TcpStream) -> Connection {
        Connection {
            stream,
            // 分配一个缓冲区，具有4kb的缓冲长度
            buffer: BytesMut::with_capacity(4096),
        }
    }
}

```

接下来，实现 `read_frame` 方法：

```

use tokio::io::AsyncReadExt;
use bytes::Buf;
use mini_redis::Result;

pub async fn read_frame(&mut self)
    -> Result<Option<Frame>>
{
    loop {
        // 尝试从缓冲区的数据中解析出一个数据帧,
        // 只有当数据足够被解析时，才返回对应的帧
        if let Some(frame) = self.parse_frame()? {
            return Ok(Some(frame));
        }

        // 如果缓冲区中的数据还不足以被解析为一个数据帧,
        // 那么我们需要从 socket 中读取更多的数据
        //
        // 读取成功时，会返回读取到的字节数，0 代表着读到了数据流的末尾
        if 0 == self.stream.read_buf(&mut self.buffer).await? {
            // 代码能执行到这里，说明了对端关闭了连接,
            // 需要看缓冲区是否还有数据，若没有数据，说明所有数据成功被处理,
            // 若还有数据，说明对端在发送帧的过程中断开了连接，导致只发送了部分数据
            if self.buffer.is_empty() {
                return Ok(None);
            } else {
                return Err("connection reset by peer".into());
            }
        }
    }
}

```

`read_frame` 内部使用循环的方式读取数据，直到一个完整的帧被读取到时，才会返回。当然，当远程的对端关闭了连接后，也会返回。

## Buf 特征

在上面的 `read_frame` 方法中，我们使用了 `read_buf` 来读取 socket 中的数据，该方法的参数是来自 [bytes](#) 包的 `BufMut`。

可以先来考虑下该如何使用 `read()` 和 `Vec<u8>` 来实现同样的功能：

```
use tokio::net::TcpStream;

pub struct Connection {
    stream: TcpStream,
    buffer: Vec<u8>,
    cursor: usize,
}

impl Connection {
    pub fn new(stream: TcpStream) -> Connection {
        Connection {
            stream,
            // 4kb 大小的缓冲区
            buffer: vec![0; 4096],
            cursor: 0,
        }
    }
}
```

下面是相应的 `read_frame` 方法：

```

use mini_redis::{Frame, Result};

pub async fn read_frame(&mut self)
    -> Result<Option<Frame>>
{
    loop {
        if let Some(frame) = self.parse_frame()? {
            return Ok(Some(frame));
        }

        // 确保缓冲区长度足够
        if self.buffer.len() == self.cursor {
            // 若不够，需要增加缓冲区长度
            self.buffer.resize(self.cursor * 2, 0);
        }

        // 从游标位置开始将数据读入缓冲区
        let n = self.stream.read(
            &mut self.buffer[self.cursor..]).await?;

        if 0 == n {
            if self.cursor == 0 {
                return Ok(None);
            } else {
                return Err("connection reset by peer".into());
            }
        } else {
            // 更新游标位置
            self.cursor += n;
        }
    }
}

```

在这段代码中，我们使用了非常重要的技术：通过游标( cursor )跟踪已经读取的数据，并将下次读取的数据写入到游标之后的缓冲区中，只有这样才不会让新读取的数据将之前读取的数据覆盖掉。

一旦缓冲区满了，还需要增加缓冲区的长度，这样才能继续写入数据。还有一点值得注意，在 `parse_frame` 方法的内部实现中，也需要通过游标来解析数据: `self.buffer[..self.cursor]`，通过这种方式，我们可以准确获取到目前已经读取的全部数据。

在网络编程中，通过字节数组和游标的方式读取数据是非常普遍的，因此 `bytes` 包提供了一个 `Buf` 特征，如果一个类型可以被读取数据，那么该类型需要实现 `Buf` 特征。与之对应，当一个类型可以被写入数据时，它需要实现 `BufMut`。

当 `T: BufMut` ( 特征约束，说明类型 `T` 实现了 `BufMut` 特征 ) 被传给 `read_buf()` 方法时，缓冲区 `T` 的内部游标会自动进行更新。正因为如此，在使用了 `BufMut` 版本的 `read_frame` 中，我们并不需要管理自己的游标。

除了游标之外，`Vec<u8>` 的使用也值得关注，该缓冲区在使用时必须要被初始化：`vec![0; 4096]`，该初始化会创建一个 4096 字节长度的数组，然后将数组的每个元素都填充上 0。当缓冲区长度不足时，新创建的缓冲区数组依然会使用 0 被重新填充一遍。事实上，这种初始化过程会存在一定的性能开销。

与 `Vec<u8>` 相反，`BytesMut` 和 `BufMut` 就没有这个问题，它们无需被初始化，而且 `BytesMut` 还会阻止我们读取未初始化的内存。

## 帧解析

在理解了该如何读取数据后，再来看看该如何通过两个部分解析出一个帧：

- 确保有一个完整的帧已经被写入了缓冲区，找到该帧的最后一个字节所在的位置
- 解析帧

```

use mini_redis::{Frame, Result};
use mini_redis::frame::Error::Incomplete;
use bytes::Buf;
use std::io::Cursor;

fn parse_frame(&mut self)
    -> Result<Option<Frame>>
{
    // 创建 `T: Buf` 类型
    let mut buf = Cursor::new(&self.buffer[..]);

    // 检查是否读取了足够解析出一个帧的数据
    match Frame::check(&mut buf) {
        Ok(_) => {
            // 获取组成该帧的字节数
            let len = buf.position() as usize;

            // 在解析开始之前，重置内部的游标位置
            buf.set_position(0);

            // 解析帧
            let frame = Frame::parse(&mut buf)?;

            // 解析完成，将缓冲区该帧的数据移除
            self.buffer.advance(len);

            // 返回解析出的帧
            Ok(Some(frame))
        }
        Err(Incomplete) => Ok(None),
        Err(e) => Err(e.into()),
    }
}

```

完整的 `Frame::check` 函数实现在这里，感兴趣的同学可以看看，在这里我们不会对它进行完整的介绍。

值得一提的是，`Frame::check` 使用了 `Buf` 的字节迭代风格的 API。例如，为了解析一个帧，首先需要检查它的第一个字节，该字节用于说明帧的类型。这种首字节检查是通过 `Buf::get_u8` 函数完成的，该函数会获取游标所在位置的字节，然后将游标位置向右移动一个字节。

## 缓冲写入(Buffered writes)

关于帧操作的另一个 API 是 `write_frame(frame)` 函数，它会将一个完整的帧写入到 socket 中。每一次写入，都会触发一次或数次系统调用，当程序中有大量的连接和写入时，系统调用的开销将变得非常高昂，具体可以看看 SyllaDB 团队写过的一篇[性能调优文章](#)。

为了降低系统调用的次数，我们需要使用一个写入缓冲区，当写入一个帧时，首先会写入该缓冲区，然后等缓冲区数据足够多时，再集中将其中的数据写入到 socket 中，这样就将多次系统调用优化减少到一次。

还有，缓冲区也不总是能提升性能。例如，考虑一个 `bulk` 帧(多个帧放在一起组成一个 bulk，通过批量发送提升效率)，该帧的特点就是：由于由多个帧组合而成，因此帧体数据可能会很大。所以我们不能将其帧体数据写入到缓冲区中，因为数据较大时，先写入缓冲区再写入 socket 会有较大的性能开销(实际上缓冲区就是为了批量写入，既然 bulk 已经是批量了，因此不使用缓冲区也很正常)。

为了实现缓冲写，我们将使用 `BufWriter` 结构体。该结构体实现了 `AsyncWrite` 特征，当 `write` 方法被调用时，不会直接写入到 socket 中，而是先写入到缓冲区中。当缓冲区被填满时，其中的内容会自动刷到(写入到)内部的 socket 中，然后再将缓冲区清空。当然，其中还存在某些优化，通过这些优化可以绕过缓冲区直接访问 socket。

由于篇幅有限，我们不会实现完整的 `write_frame` 函数，想要看完整代码可以访问[这里](#)。

首先，更新下 `Connection` 的结构体：

```
use tokio::io::BufWriter;
use tokio::net::TcpStream;
use bytes::BytesMut;

pub struct Connection {
    stream: BufWriter<TcpStream>,
    buffer: BytesMut,
}

impl Connection {
    pub fn new(stream: TcpStream) -> Connection {
        Connection {
            stream: BufWriter::new(stream),
            buffer: BytesMut::with_capacity(4096),
        }
    }
}
```

接着来实现 `write_frame` 函数：

```

use tokio::io::{self, AsyncWriteExt};
use mini_redis::Frame;

async fn write_frame(&mut self, frame: &Frame)
    -> io::Result<()>
{
    match frame {
        Frame::Simple(val) => {
            self.stream.write_u8(b'+').await?;
            self.stream.write_all(val.as_bytes()).await?;
            self.stream.write_all(b"\r\n").await?;
        }
        Frame::Error(val) => {
            self.stream.write_u8(b'-').await?;
            self.stream.write_all(val.as_bytes()).await?;
            self.stream.write_all(b"\r\n").await?;
        }
        Frame::Integer(val) => {
            self.stream.write_u8(b':').await?;
            self.write_decimal(*val).await?;
        }
        Frame::Null => {
            self.stream.write_all(b"$-1\r\n").await?;
        }
        Frame::Bulk(val) => {
            let len = val.len();

            self.stream.write_u8(b'$').await?;
            self.write_decimal(len as u64).await?;
            self.stream.write_all(val).await?;
            self.stream.write_all(b"\r\n").await?;
        }
        Frame::Array(_val) => unimplemented!(),
    }

    self.stream.flush().await;

    Ok(())
}

```

这里使用的方法由 `AsyncWriteExt` 提供，它们在 `TcpStream` 中也有对应的函数。但是在没有缓冲区的情况下最好避免使用这种逐字节的写入方式！不然，每写入几个字节就会触发一次系统调用，写完整个数据帧可能需要几十次系统调用，可以说是丧心病狂！

- `write_u8` 写入一个字节
- `write_all` 写入所有数据
- `write_decimal` 由 `mini-redis` 提供

在函数结束前，我们还额外的调用了一次 `self.stream.flush().await`，原因是缓冲区可能还存在数据，因此需要手动刷一次数据：`flush` 的调用会将缓冲区中剩余的数据立刻写入到 `socket` 中。

当然，当帧比较小的时候，每写一次帧就 `flush` 一次的模式性能开销会比较大，此时我们可以选择在 `Connection` 中实现 `flush` 函数，然后将等帧积累多个后，再一次性在 `Connection` 中进行 `flush`。当然，对于我们的例子来说，简洁性是非常重要的，因此选了将 `flush` 放入到 `write_frame` 中。

# 深入 Tokio 背后的异步原理

在经过多个章节的深入学习后，Tokio 对我们来说不再是一座隐于云雾中的高山，它其实蛮简单好用的，甚至还有丝丝的可爱!?

但从现在开始，如果想要进一步的深入 Tokio，首先需要深入理解 `async` 的原理，其实我们在之前的章节已经深入学习过，这里结合 Tokio 再来回顾下。

## Future

先来回顾一下 `async fn` 异步函数：

```
use tokio::net::TcpStream;

async fn my_async_fn() {
    println!("hello from async");
    // 通过 .await 创建 socket 连接
    let _socket = TcpStream::connect("127.0.0.1:3000").await.unwrap();
    println!("async TCP operation complete");
    // 关闭socket
}
```

接着对它进行调用获取一个返回值，再在返回值上调用 `.await`：

```
#[tokio::main]
async fn main() {
    let what_is_this = my_async_fn();
    // 上面的调用不会产生任何效果

    // ... 执行一些其它代码

    what_is_this.await;
    // 直到 .await 后，文本才被打印，socket 连接也被创建和关闭
}
```

在上面代码中 `my_async_fn` 函数为何可以惰性执行(直到 `.await` 调用时才执行)? 秘密就在于 `async fn` 声明的函数返回一个 `Future`。

`Future` 是一个实现了 `std::future::Future` 特征的值，该值包含了一系列异步计算过程，而这个过程直到 `.await` 调用时才会被执行。

`std::future::Future` 的定义如下所示:

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context)
        -> Poll<Self::Output>;
}
```

代码中有几个关键点:

- **关联类型** `Output` 是 `Future` 执行完成后返回的值的类型
- `Pin` 类型是在异步函数中进行借用的关键, 在[这里](#)有非常详细的介绍

和其它语言不同, Rust 中的 `Future` 不代表一个发生在后台的计算, 而是 `Future` 就代表了计算本身, 因此 `Future` 的所有者有责任去推进该计算过程的执行, 例如通过 `Future::poll` 函数。听上去好像还挺复杂? 但是大家不必担心, 因为这些都在 Tokio 中帮你自动完成了 :)

## 实现 Future

下面来一起实现个五脏俱全的 `Future`, 它将: 1. 等待某个特定时间点的到来 2. 在标准输出打印文本 3. 生成一个字符串

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
}

// 为我们的 Delay 类型实现 Future 特征
impl Future for Delay {
    type Output = &'static str;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<&'static str>
    {
        if Instant::now() >= self.when {
            // 时间到了, Future 可以结束
            println!("Hello world");
            // Future 执行结束并返回 "done" 字符串
            Poll::Ready("done")
        } else {
            // 目前先忽略下面这行代码
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}

#[tokio::main]
async fn main() {
    let when = Instant::now() + Duration::from_millis(10);
    let future = Delay { when };

    // 运行并等待 Future 的完成
    let out = future.await;

    // 判断 Future 返回的字符串是否是 "done"
    assert_eq!(out, "done");
}

```

以上代码很清晰的解释了如何自定义一个 Future，并指定它如何通过 poll 一步一步执行，直到最终完成返回 "done" 字符串。

### async fn 作为 Future

大家有没有注意到，上面代码我们在 main 函数中初始化一个 Future 并使用 .await 对其进行调用执行，如果你是在 fn main 中这么做，是会报错的。

原因是 `.await` 只能用于 `async fn` 函数中，因此我们将 `main` 函数声明成 `async fn main` 同时使用 `#[tokio::main]` 进行了标注，此时 `async fn main` 生成的代码类似下面：

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};

enum MainFuture {
    // 初始化, 但永远不会被 poll
    State0,
    // 等待 `Delay` 运行, 例如 `future.await` 代码行
    State1(Delay),
    // Future 执行完成
    Terminated,
}

impl Future for MainFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<()>
    {
        use MainFuture::*;

        loop {
            match *self {
                State0 => {
                    let when = Instant::now() +
                        Duration::from_millis(10);
                    let future = Delay { when };
                    *self = State1(future);
                }
                State1(ref mut my_future) => {
                    match Pin::new(my_future).poll(cx) {
                        Poll::Ready(out) => {
                            assert_eq!(out, "done");
                            *self = Terminated;
                            return Poll::Ready(());
                        }
                        Poll::Pending => {
                            return Poll::Pending;
                        }
                    }
                }
                Terminated => {
                    panic!("future polled after completion")
                }
            }
        }
    }
}
```

可以看出，编译器会将 Future 变成状态机，其中 MainFuture 包含了 Future 可能处于的状态：从 State0 状态开始，当 poll 被调用时，Future 会尝试去尽可能的推进内部的状态，若它可以被完成时，就会返回 Poll::Ready，其中还会包含最终的输出结果。

若 Future 无法被完成，例如它所等待的资源还没有准备好，此时就会返回 Poll::Pending，该返回值会通知调用者：Future 会在稍后才能完成。

同时可以看到：当一个 Future 由其它 Future 组成时，调用外层 Future 的 poll 函数会同时调用一次内部 Future 的 poll 函数。

## 执行器( Executor )

async fn 返回 Future，而后者需要通过被不断的 poll 才能往前推进状态，同时该 Future 还能包含其它 Future，那么问题来了谁来负责调用最外层 Future 的 poll 函数？

回一下之前的内容，为了运行一个异步函数，我们必须使用 tokio::spawn 或通过 #[tokio::main] 标注的 async fn main 函数。它们有一个非常重要的作用：将最外层 Future 提交给 Tokio 的执行器。该执行器负责调用 poll 函数，然后推动 Future 的执行，最终直至完成。

### mini tokio

为了更好理解相关的内容，我们一起来实现一个迷你版本的 Tokio，完整的代码见[这里](#)。

先来看一段基础代码：

```
use std::collections::VecDeque;
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};
use futures::task;

fn main() {
    let mut mini_tokio = MiniTokio::new();

    mini_tokio.spawn(async {
        let when = Instant::now() + Duration::from_millis(10);
        let future = Delay { when };

        let out = future.await;
        assert_eq!(out, "done");
    });

    mini_tokio.run();
}

struct MiniTokio {
    tasks: VecDeque<Task>,
}

type Task = Pin<Box<dyn Future<Output = ()> + Send>>;

impl MiniTokio {
    fn new() -> MiniTokio {
        MiniTokio {
            tasks: VecDeque::new(),
        }
    }

    /// 生成一个 Future并放入 mini-tokio 实例的任务队列中
    fn spawn<F>(&mut self, future: F)
    where
        F: Future<Output = ()> + Send + 'static,
    {
        self.tasks.push_back(Box::pin(future));
    }

    fn run(&mut self) {
        let waker = task::noop_waker();
        let mut cx = Context::from_waker(&waker);

        while let Some(mut task) = self.tasks.pop_front() {
            if task.as_mut().poll(&mut cx).is_pending() {
                self.tasks.push_back(task);
            }
        }
    }
}
```

```
    }
}
```

以上代码运行了一个 `async` 语句块 `mini_tokio.spawn(async {...})`，还创建了一个 `Delay` 实例用于等待所需的时间。看上去相当不错，但这个实现有一个 **重大缺陷**：我们的执行器永远也不会休眠。执行器会持续的循环遍历所有的 `Future`，然后不停的 `poll` 它们，但是事实上，大多数 `poll` 都是没有用的，因为此时 `Future` 并没有准备好，因此会继续返回 `Poll::Pending`，最终这个循环遍历会让你的 CPU 疲于奔命，真打工人！

鉴于此，我们的 `mini-tokio` 只应该在 `Future` 准备好可以进一步运行后，才去 `poll` 它，例如该 `Future` 之前阻塞等待的资源已经准备好并可以被使用了，就可以对其进行 `poll`。再比如，如果一个 `Future` 任务在阻塞等待从 TCP socket 中读取数据，那我们只想在 `socket` 中有数据可以读取后才去 `poll` 它，而不是没事就 `poll` 着玩。

回到上面的代码中，`mini-tokio` 只应该当任务的延迟时间到了后，才去 `poll` 它。为了实现这个功能，我们需要 `通知 -> 运行` 机制：当任务可以进一步被推进运行时，它会主动通知执行器，然后执行器再来 `poll`。

## Waker

一切的答案都在 `Waker` 中，资源可以用它来通知正在等待的任务：该资源已经准备好，可以继续运行了。

再来看下 `Future::poll` 的定义：

```
fn poll(self: Pin<&mut Self>, cx: &mut Context)
    -> Poll<Self::Output>;
```

`Context` 参数中包含有 `waker()` 方法。该方法返回一个绑定到当前任务上的 `Waker`，然后 `Waker` 上定义了一个 `wake()` 方法，用于通知执行器相关的任务可以继续执行。

准确来说，当 `Future` 阻塞等待的资源已经准备好时(例如 `socket` 中有了可读取的数据)，该资源可以调用 `wake()` 方法，来通知执行器可以继续调用该 `Future` 的 `poll` 函数来推进任务的执行。

### 发送 wake 通知

现在，为 `Delay` 添加下 `Waker` 支持：

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::{Duration, Instant};
use std::thread;

struct Delay {
    when: Instant,
}

impl Future for Delay {
    type Output = &'static str;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<&'static str>
    {
        if Instant::now() >= self.when {
            println!("Hello world");
            Poll::Ready("done")
        } else {
            // 为当前任务克隆一个 waker 的句柄
            let waker = cx.waker().clone();
            let when = self.when;

            // 生成一个计时器线程
            thread::spawn(move || {
                let now = Instant::now();

                if now < when {
                    thread::sleep(when - now);
                }

                waker.wake();
            });
        }
        Poll::Pending
    }
}

```

此时，计时器用来模拟一个阻塞等待的资源，一旦计时结束(该资源已经准备好)，资源会通过 `waker.wake()` 调用通知执行器我们的任务再次被调度执行了。

当然，现在的实现还较为粗糙，等会我们会来进一步优化，在此之前，先来看看如何监听这个 `wake` 通知。

---

当 `Future` 会返回 `Poll::Pending` 时，一定要确保 `wake` 能被正常调用，否则会导致任务永远被挂起，再也不会被执行器 `poll`。

## 忘记在返回 Poll::Pending 时调用 wake 是很多难以发现 bug 的潜在源头！

---

再回忆下最早实现的 Delay 代码：

```
impl Future for Delay {
    type Output = &'static str;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<&'static str>
    {
        if Instant::now() >= self.when {
            // 时间到了，Future 可以结束
            println!("Hello world");
            // Future 执行结束并返回 "done" 字符串
            Poll::Ready("done")
        } else {
            // 目前先忽略下面这行代码
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}
```

在返回 Poll::Pending 之前，先调用了 `cx.waker().wake_by_ref()`，由于此时我们还没有模拟计时资源，因此这里直接调用了 `wake` 进行通知，这样做会导致当前的 Future 被立即再次调度执行。

由此可见，这种通知的控制权是在你手里的，甚至可以像上面代码这样，还没准备好资源，就直接进行 `wake` 通知，但是总归意义不大，而且浪费了 CPU，因为这种 执行 -> 立即通知再调度 -> 执行 的方式会造成一个非常繁忙的循环。

### 处理 wake 通知

下面，让我们更新 mini-tokio 服务，让它能接收 wake 通知：当 `waker.wake()` 被调用后，相关联的任务会被放入执行器的队列中，然后等待执行器的调用执行。

为了实现这一点，我们将使用消息通道来排队存储这些被唤醒并等待调度的任务。有一点需要注意，从消息通道接收消息的线程(执行器所在的线程)和发送消息的线程 (唤醒任务时所在的线程) 可能是不同的，因此消息(`Waker`)必须要实现 `Send` 和 `Sync`，才能跨线程使用。

---

关于 `Send` 和 `Sync` 的具体讲解见[这里](#)

---

基于以上理由，我们选择使用来自于 `crossbeam` 的消息通道，因为标准库中的消息通道不是 `Sync` 的。在 `Cargo.toml` 中添加以下依赖：

```
crossbeam = "0.8"
```

再来更新下 `MiniTokio` 结构体：

```
use crossbeam::channel;
use std::sync::Arc;

struct MiniTokio {
    scheduled: channel::Receiver<Arc<Task>>,
    sender: channel::Sender<Arc<Task>>,
}

struct Task {
    // 先空着，后面会填充代码
}
```

`Waker` 实现了 `Sync` 特征，同时还可以被克隆，当 `wake` 被调用时，任务就会被调度执行。

为了实现上述的目的，我们引入了消息通道，当 `waker.wake()` 函数被调用时，任务会被发送到该消息通道中：

```
use std::sync::{Arc, Mutex};

struct Task {
    // `Mutex` 是为了让 `Task` 实现 `Sync` 特征，它能保证同一时间只有一个线程可以访问 `Future`。
    // 事实上 `Mutex` 并没有在 Tokio 中被使用，这里我们只是为了简化：Tokio 的真实代码实在太长了 :D
    future: Mutex<Pin<Box<dyn Future<Output = ()> + Send>>,
    executor: channel::Sender<Arc<Task>>,
}

impl Task {
    fn schedule(self: &Arc<Self>) {
        self.executor.send(self.clone());
    }
}
```

接下来，我们需要让 `std::task::Waker` 能准确的找到所需的调度函数 关联起来，对此标准库中提供了一个底层的 API `std::task::RawWakerVTable` 可以用于手动的访问 `vtable`，这种实现提供了最大的灵活性，但是需要大量 `unsafe` 的代码。

因此我们选择更加高级的实现：由 `futures` 包提供的 `ArcWake` 特征，只要简单实现该特征，就可以将我们的 `Task` 转变成一个 `waker`。在 `Cargo.toml` 中添加以下包：

```
futures = "0.3"
```

然后为我们的任务 Task 实现 ArcWake：

```
use futures::task::{self, ArcWake};
use std::sync::Arc;
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        arc_self.schedule();
    }
}
```

当之前的计时器线程调用 `waker.wake()` 时，所在的任务会被推入到消息通道中。因此接下来，我们需要实现接收端的功能，然后 `MiniTokio::run()` 函数中执行该任务：

```

impl MiniTokio {
    // 从消息通道中接收任务，然后通过 poll 来执行
    fn run(&self) {
        while let Ok(task) = self.scheduled.recv() {
            task.poll();
        }
    }

    /// 初始化一个新的 mini-tokio 实例
    fn new() -> MiniTokio {
        let (sender, scheduled) = channel::unbounded();

        MiniTokio { scheduled, sender }
    }

    /// 在下面函数中，通过参数传入的 future 被 `Task` 包裹起来，然后会被推入到调度队列中，当
    `run` 被调用时，该 future 将被执行
    fn spawn<F>(&self, future: F)
    where
        F: Future<Output = ()> + Send + 'static,
    {
        Task::spawn(future, &self.sender);
    }
}

impl Task {
    fn poll(self: Arc<Self>) {
        // 基于 Task 实例创建一个 waker，它使用了之前的 `ArcWake`
        let waker = task::waker(self.clone());
        let mut cx = Context::from_waker(&waker);

        // 没有其他线程在竞争锁时，我们将获取到目标 future
        let mut future = self.future.try_lock().unwrap();

        // 对 future 进行 poll
        let _ = future.as_mut().poll(&mut cx);
    }

    // 使用给定的 future 来生成新的任务
    //
    // 新的任务会被推到 `sender` 中，接着该消息通道的接收端就可以获取该任务，然后执行
    fn spawn<F>(future: F, sender: &channel::Sender<Arc<Task>>)
    where
        F: Future<Output = ()> + Send + 'static,
    {
        let task = Arc::new(Task {
            future: Mutex::new(Box::pin(future)),
            executor: sender.clone(),
        });

        let _ = sender.send(task);
    }
}

```

```
    }  
}  
}
```

首先，我们实现了 `MiniTokio::run()` 函数，它会持续从消息通道中接收被唤醒的任务，然后通过 `poll` 来推动其继续执行。

其次，`MiniTokio::new()` 和 `MiniTokio::spawn()` 使用了消息通道而不是一个 `VecDeque`。当新任务生成后，这些任务中会携带上消息通道的发送端，当任务中的资源准备就绪时，会使用该发送端将该任务放入消息通道的队列中，等待执行器 `poll`。

`Task::poll()` 函数使用 `futures` 包提供的 `ArcWake` 创建了一个 `waker`，后者可以用来创建 `task::Context`，最终该 `Context` 会被传给执行器调用的 `poll` 函数。

---

注意，`Task::poll` 和执行器调用的 `poll` 是完全不同的，大家别搞混了

---

## 一些遗留问题

至此，我们的程序已经差不多完成，还剩几个遗留问题需要解决下。

### 在异步函数中生成异步任务

之前实现 `Delay Future` 时，我们提到有几个问题需要解决。Rust 的异步模型允许一个 `Future` 在执行过程中可以跨任务迁移：

```
use futures::future::poll_fn;
use std::future::Future;
use std::pin::Pin;

#[tokio::main]
async fn main() {
    let when = Instant::now() + Duration::from_millis(10);
    let mut delay = Some(Delay { when });

    poll_fn(move |cx| {
        let mut delay = delay.take().unwrap();
        let res = Pin::new(&mut delay).poll(cx);
        assert!(res.is_pending());
        tokio::spawn(async move {
            delay.await;
        });

        Poll::Ready(())
    }).await;
}
```

首先，`poll_fn` 函数使用闭包创建了一个 `Future`，其次，上面代码还创建一个 `Delay` 实例，然后在闭包中，对其进行了一次 `poll`，接着再将该 `Delay` 实例发送到一个新的任务，在此任务中使用 `.await` 进行了执行。

在例子中，`Delay::poll` 被调用了不止一次，且使用了不同的 `Waker` 实例，在这种场景下，你必须确保调用最近一次 `poll` 函数中的 `Waker` 参数中的 `wake` 方法。也就是调用最内层 `poll` 函数参数(`Waker`)上的 `wake` 方法。

当实现一个 `Future` 时，很关键的一点就是要假设每次 `poll` 调用都会应用到一个不同的 `Waker` 实例上。因此 `poll` 函数必须要使用一个新的 `waker` 去更新替代之前的 `waker`。

我们之前的 `Delay` 实现中，会在每一次 `poll` 调用时都生成一个新的线程。这么做问题不大，但是当 `poll` 调用较多时会出现明显的性能问题！一个解决方法就是记录你是否已经生成了一个线程，然后只有在没有生成时才去创建一个新的线程。但是一旦这么做，就必须确保线程的 `Waker` 在后续 `poll` 调用中被正确更新，否则你无法唤醒最近的 `Waker`！

这一段大家可能会看得云里雾里的，没办法，原文就绕来绕去，好在终于可以看代码了。。我们可以通过代码来解决疑惑：

```

use std::future::Future;
use std::pin::Pin;
use std::sync::{Arc, Mutex};
use std::task::{Context, Poll, Waker};
use std::thread;
use std::time::{Duration, Instant};

struct Delay {
    when: Instant,
    // 用于说明是否已经生成一个线程
    // Some 代表已经生成, None 代表还没有
    waker: Option<Arc<Mutex<Waker>>,
}

impl Future for Delay {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        // 若这是 Future 第一次被调用, 那么需要先生成一个计时器线程。
        // 若不是第一次调用(该线程已在运行), 那要确保已存储的 `Waker` 跟当前任务的 `waker` 匹配
        if let Some(waker) = &self.waker {
            let mut waker = waker.lock().unwrap();

            // 检查之前存储的 `waker` 是否跟当前任务的 `waker` 相匹配。
            // 这是必要的, 原因是 `Delay Future` 的实例可能会在两次 `poll` 之间被转移到另一个
            // 任务中, 然后
            // 存储的 waker 被该任务进行了更新。
            // 这种情况一旦发生, `Context` 包含的 `waker` 将不同于存储的 `waker`。
            // 因此我们必须对存储的 `waker` 进行更新
            if !waker.will_wake(cx.waker()) {
                *waker = cx.waker().clone();
            }
        } else {
            let when = self.when;
            let waker = Arc::new(Mutex::new(cx.waker().clone()));
            self.waker = Some(waker.clone());

            // 第一次调用 `poll`, 生成计时器线程
            thread::spawn(move || {
                let now = Instant::now();

                if now < when {
                    thread::sleep(when - now);
                }

                // 计时结束, 通过调用 `waker` 来通知执行器
                let waker = waker.lock().unwrap();
                waker.wake_by_ref();
            });
        }
    }

    // 一旦 waker 被存储且计时器线程已经开始, 我们就需要检查 `delay` 是否已经完成
}

```

```
// 若计时已完成，则当前 Future 就可以完成并返回 `Poll::Ready`  
if Instant::now() >= self.when {  
    Poll::Ready(())  
} else {  
    // 计时尚未结束，Future 还未完成，因此返回 `Poll::Pending`。  
    //  
    // `Future` 特征要求当 `Pending` 被返回时，那我们要确保当资源准备好时，必须调用  
    // `waker` 以通知执行器。 在我们的例子中，会通过生成的计时线程来保证  
    //  
    // 如果忘记调用 waker， 那等待我们的将是深渊：该任务将被永远的挂起，无法再执行  
    Poll::Pending  
}  
}  
}
```

这着实有些复杂(原文。。。)，但是简单来看就是：在每次 `poll` 调用时，都会检查 `Context` 中提供的 `waker` 和我们之前记录的 `waker` 是否匹配。若匹配，就什么都不用做，若不匹配，那之前存储的就必须进行更新。

## Notify

我们之前证明了如何用手动编写的 `waker` 来实现 `Delay Future`。`Waker` 是 Rust 异步编程的基石，因此绝大多数时候，我们并不需要直接去使用它。例如，在 `Delay` 的例子中，可以使用 `tokio::sync::Notify` 去实现。

该 `Notify` 提供了一个基础的任务通知机制，它会处理这些 `waker` 的细节，包括确保两次 `waker` 的匹配：

```
use tokio::sync::Notify;
use std::sync::Arc;
use std::time::{Duration, Instant};
use std::thread;

async fn delay(dur: Duration) {
    let when = Instant::now() + dur;
    let notify = Arc::new(Notify::new());
    let notify2 = notify.clone();

    thread::spawn(move || {
        let now = Instant::now();

        if now < when {
            thread::sleep(when - now);
        }

        notify2.notify_one();
    });
}

notify.notified().await;
}
```

当使用 `Notify` 后，我们就可以轻松的实现如上的 `delay` 函数。

## 总结

在看完这么长的文章后，我们来总结下，否则大家可能还会遗忘：

- 在 Rust 中，`async` 是惰性的，直到执行器 `poll` 它们时，才会开始执行
- `Waker` 是 `Future` 被执行的关键，它可以链接起 `Future` 任务和执行器
- 当资源没有准备时，会返回一个 `Poll::Pending`
- 当资源准备好时，会通过 `waker.wake` 发出通知
- 执行器会收到通知，然后调度该任务继续执行，此时由于资源已经准备好，因此任务可以顺利往前推进了

# select!

在实际使用时，一个重要的场景就是同时等待多个异步操作的结果，并且对其结果进行进一步处理，在本章节，我们来看看，强大的 `select!` 是如何帮助咱们更好的控制多个异步操作并发执行的。

## tokio::select!

`select!` 允许同时等待多个计算操作，然后当其中一个操作完成时就退出等待：

```
use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::channel();
    let (tx2, rx2) = oneshot::channel();

    tokio::spawn(async {
        let _ = tx1.send("one");
    });

    tokio::spawn(async {
        let _ = tx2.send("two");
    });

    tokio::select! {
        val = rx1 => {
            println!("rx1 completed first with {:?}", val);
        }
        val = rx2 => {
            println!("rx2 completed first with {:?}", val);
        }
    }

    // 任何一个 select 分支结束后，都会继续执行接下来的代码
}
```

这里用到了两个 `oneshot` 消息通道，虽然两个操作的创建在代码上有先后顺序，但在实际执行时却不一样。因此，`select` 在从两个通道阻塞等待接收消息时，`rx1` 和 `rx2` 都有可能被先打印出来。

需要注意，任何一个 `select` 分支完成后，都会继续执行后面的代码，没被执行的分支会被丢弃(`dropped`)。

## 取消

对于 Async Rust 来说，释放( drop )掉一个 Future 就意味着取消任务。从上一章节可以得知，`async` 操作会返回一个 `Future`，而后者是惰性的，直到被 `poll` 调用时，才会被执行。一旦 `Future` 被释放，那操作将无法继续，因为所有相关的关系都被释放。

对于 Tokio 的 `oneshot` 的接收端来说，它在被释放时会发送一个关闭通知到发送端，因此发送端可以通过释放任务的方式来终止正在执行的任务。

```
use tokio::sync::oneshot;

async fn some_operation() -> String {
    // 在这里执行一些操作...
}

#[tokio::main]
async fn main() {
    let (mut tx1, rx1) = oneshot::channel();
    let (tx2, rx2) = oneshot::channel();

    tokio::spawn(async {
        // 等待 `some_operation` 的完成
        // 或者处理 `oneshot` 的关闭通知
        tokio::select! {
            val = some_operation() => {
                let _ = tx1.send(val);
            }
            _ = tx1.closed() => {
                // 收到了发送端发来的关闭信号
                // `select` 即将结束，此时，正在进行的 `some_operation()` 任务会被取消，任务
                自动完成，
                // tx1 被释放
            }
        }
    });

    tokio::spawn(async {
        let _ = tx2.send("two");
    });

    tokio::select! {
        val = rx1 => {
            println!("rx1 completed first with {:?}", val);
        }
        val = rx2 => {
            println!("rx2 completed first with {:?}", val);
        }
    }
}
```

上面代码的重点就在于 `tx1.closed` 所在的分支，一旦发送端被关闭，那该分支就会被执行，然后 `select` 会退出，并清理掉还没执行的第一个分支 `val = some_operation()`，这其中 `some_operation` 返回的 `Future` 也会被清理，根据之前的内容，`Future` 被清理那相应的任务会立即取消，因此 `some_operation` 会被取消，不再执行。

## Future 的实现

为了更好的理解 `select` 的工作原理，我们来看看如果使用 `Future` 该如何实现。当然，这里是一个简化版本，在实际中，`select!` 会包含一些额外的功能，例如一开始会随机选择一个分支进行 `poll`。

```

use tokio::sync::oneshot;
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

struct MySelect {
    rx1: oneshot::Receiver<&'static str>,
    rx2: oneshot::Receiver<&'static str>,
}

impl Future for MySelect {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        if let Poll::Ready(val) = Pin::new(&mut self.rx1).poll(cx) {
            println!("rx1 completed first with {:?}", val);
            return Poll::Ready(());
        }

        if let Poll::Ready(val) = Pin::new(&mut self.rx2).poll(cx) {
            println!("rx2 completed first with {:?}", val);
            return Poll::Ready(());
        }

        Poll::Pending
    }
}

#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::channel();
    let (tx2, rx2) = oneshot::channel();

    // 使用 tx1 和 tx2

    MySelect {
        rx1,
        rx2,
    }.await;
}

```

`MySelect` 包含了两个分支中的 `Future`，当它被 `poll` 时，第一个分支会先执行。如果执行完成，那取出的值会被使用，然后 `MySelect` 也随之结束。而另一个分支对应的 `Future` 会被释放掉，对应的操作也会被取消。

还记得上一章节中很重要的一段话吗？

---

当一个 `Future` 返回 `Poll::Pending` 时，它必须确保会在某一个时刻通过 `Waker` 来唤醒，不然

该 Future 将永远地被挂起

---

但是仔细观察我们之前的代码，里面并没有任何的 `wake` 调用！事实上，这是因为参数 `cx` 被传入了内层的 `poll` 调用。只要内部的 `Future` 实现了唤醒并且返回了 `Poll::Pending`，那 `MySelect` 也等于实现了唤醒！

## 语法

目前来说，`select!` 最多可以支持 64 个分支，每个分支形式如下：

`<模式> = <async 表达式> => <结果处理>,`

当 `select` 宏开始执行后，所有的分支会开始并发的执行。当任何一个表达式完成时，会将结果跟模式进行匹配。若匹配成功，则剩下的表达式会被释放。

最常用的模式就是用变量名去匹配表达式返回的值，然后该变量就可以在结果处理环节使用。

如果当前的模式不能匹配，剩余的 `async` 表达式将继续并发的执行，直到下一个完成。

由于 `select!` 使用的是一个 `async` 表达式，因此我们可以定义一些更复杂的计算。

例如从在分支中进行 TCP 连接：

```
use tokio::net::TcpStream;
use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel();

    // 生成一个任务，用于向 oneshot 发送一条消息
    tokio::spawn(async move {
        tx.send("done").unwrap();
    });

    tokio::select! {
        socket = TcpStream::connect("localhost:3465") => {
            println!("Socket connected {:?}", socket);
        }
        msg = rx => {
            println!("received message first {:?}", msg);
        }
    }
}
```

再比如，在分支中进行 TCP 监听：

```

use tokio::net::TcpListener;
use tokio::sync::oneshot;
use std::io;

#[tokio::main]
async fn main() -> io::Result<()> {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(async move {
        tx.send(()).unwrap();
    });

    let mut listener = TcpListener::bind("localhost:3465").await?;

    tokio::select! {
        _ = async {
            loop {
                let (socket, _) = listener.accept().await?;
                tokio::spawn(async move { process(socket) });
            }
        }
        // 给予 Rust 类型暗示
        Ok::<_, io::Error>(_) => {}
        _ = rx => {
            println!("terminating accept loop");
        }
    }
}

Ok(())
}

```

分支中接收连接的循环会一直运行，直到遇到错误才停止，或者当 `rx` 中有值时，也会停止。`_` 表示我们并不关心这个值，这样使用唯一的目的是为了结束第一分支中的循环。

## 返回值

`select!` 还能返回一个值：

```
async fn computation1() -> String {
    // .. 计算
}

async fn computation2() -> String {
    // .. 计算
}

#[tokio::main]
async fn main() {
    let out = tokio::select! {
        res1 = computation1() => res1,
        res2 = computation2() => res2,
    };

    println!("Got = {}", out);
}
```

需要注意的是，此时 `select!` 的所有分支必须返回一样的类型，否则编译器会报错！

## 错误传播

在 Rust 中使用 `?` 可以对错误进行传播，但是在 `select!` 中，`?` 如何工作取决于它是在分支中的 `async` 表达式使用还是在结果处理的代码中使用：

- 在分支中 `async` 表达式使用会将该表达式的结果变成一个 `Result`
- 在结果处理中使用，会将错误直接传播到 `select!` 之外

```

use tokio::net::TcpListener;
use tokio::sync::oneshot;
use std::io;

#[tokio::main]
async fn main() -> io::Result<()> {
    // [设置 `rx` oneshot 消息通道]

    let listener = TcpListener::bind("localhost:3465").await?;

    tokio::select! {
        res = async {
            loop {
                let (socket, _) = listener.accept().await?;
                tokio::spawn(async move { process(socket) });
            }
        } Ok::<_, io::Error>(())
        _ = rx => {
            println!("terminating accept loop");
        }
    }

    Ok(())
}

```

`listener.accept().await?` 是分支表达式中的 `?`，因此它会将表达式的返回值变成 `Result` 类型，然后赋予给 `res` 变量。

与之不同的是，结果处理中的 `res?;` 会让 `main` 函数直接结束并返回一个 `Result`，可以看出，这里 `?` 的用法跟我们平时的用法并无区别。

## 模式匹配

既然是模式匹配，我们需要再回忆下 `select!` 的分支语法形式：

`<模式> = <async 表达式> => <结果处理>,`

迄今为止，我们只用了变量绑定的模式，事实上，[任何 Rust 模式](#)都可以在此处使用。

```
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (mut tx1, mut rx1) = mpsc::channel(128);
    let (mut tx2, mut rx2) = mpsc::channel(128);

    tokio::spawn(async move {
        // 用 tx1 和 tx2 干一些不为人知的事
    });

    tokio::select! {
        Some(v) = rx1.recv() => {
            println!("Got {:?} from rx1", v);
        }
        Some(v) = rx2.recv() => {
            println!("Got {:?} from rx2", v);
        }
        else => {
            println!("Both channels closed");
        }
    }
}
```

上面代码中，`rx` 通道关闭后，`recv()` 方法会返回一个 `None`，可以看到没有任何模式能够匹配这个 `None`，那为何不会报错？秘密就在于 `else` 上：当使用模式去匹配分支时，若之前的所有分支都无法被匹配，那 `else` 分支将被执行。

## 借用

当在 Tokio 中生成(`spawn`)任务时，其 `async` 语句块必须拥有其中数据的所有权。而 `select!` 并没有这个限制，它的每个分支表达式可以直接借用数据，然后进行并发操作。只要遵循 Rust 的借用规则，多个分支表达式可以不可变的借用同一个数据，或者在一个表达式可变的借用某个数据。

来看个例子，在这里我们同时向两个 TCP 目标发送同样的数据：

```

use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;
use std::io;
use std::net::SocketAddr;

async fn race(
    data: &[u8],
    addr1: SocketAddr,
    addr2: SocketAddr
) -> io::Result<()> {
    tokio::select! {
        Ok(_) = async {
            let mut socket = TcpStream::connect(addr1).await?;
            socket.write_all(data).await?;
            Ok::<_, io::Error>(())
        } => {}
        Ok(_) = async {
            let mut socket = TcpStream::connect(addr2).await?;
            socket.write_all(data).await?;
            Ok::<_, io::Error>(())
        } => {}
        else => {}
    };
    Ok(())
}

```

这里其实有一个很有趣的题外话，由于 TCP 连接过程是在模式中发生的，因此当某一个连接过程失败后，它通过 ? 返回的 Err 类型并无法匹配 Ok，因此另一个分支会继续被执行，继续连接。

如果你把连接过程放在了结果处理中，那连接失败会直接从 race 函数中返回，而不是继续执行另一个分支中的连接！

还有一个非常重要的点，**借用规则在分支表达式和结果处理中存在很大的不同**。例如上面代码中，我们在两个分支表达式中分别对 data 做了不可变借用，这当然 ok，但是若是两次可变借用，那编译器会立即进行报错。但是转折来了：当在结果处理中进行两次可变借用时，却不会报错，大家可以思考下为什么，提示下：思考下分支在执行完成后会发生什么？

```
use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::channel();
    let (tx2, rx2) = oneshot::channel();

    let mut out = String::new();

    tokio::spawn(async move {
});

    tokio::select! {
        _ = rx1 => {
            out.push_str("rx1 completed");
        }
        _ = rx2 => {
            out.push_str("rx2 completed");
        }
    }

    println!("{}", out);
}
```

例如以上代码，就在两个分支的结果处理中分别进行了可变借用，并不会报错。原因就在于：`select!`会保证只有一个分支的结果处理会被运行，然后在运行结束后，另一个分支会被直接丢弃。

## 循环

来看看该如何在循环中使用`select!`，顺便说一句，跟循环一起使用是最常见的使用方式。

```
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (tx1, mut rx1) = mpsc::channel(128);
    let (tx2, mut rx2) = mpsc::channel(128);
    let (tx3, mut rx3) = mpsc::channel(128);

    loop {
        let msg = tokio::select! {
            Some(msg) = rx1.recv() => msg,
            Some(msg) = rx2.recv() => msg,
            Some(msg) = rx3.recv() => msg,
            else => { break }
        };
        println!("Got {}", msg);
    }

    println!("All channels have been closed.");
}
```

在循环中使用 `select!` 最大的不同就是，当某一个分支执行完成后，`select!` 会继续循环等待并执行下一个分支，直到所有分支最终都完成，最终匹配到 `else` 分支，然后通过 `break` 跳出循环。

老生常谈的一句话：`select!` 中哪个分支先被执行是无法确定的，因此不要依赖于分支执行的顺序！想象一下，在异步编程场景，若 `select!` 按照分支的顺序来执行会如何：若 `rx1` 中总是有数据，那每次循环都只会去处理第一个分支，后面两个分支永远不会被执行。

## 恢复之前的异步操作

```
async fn action() {
    // 一些异步逻辑
}

#[tokio::main]
async fn main() {
    let (mut tx, mut rx) = tokio::sync::mpsc::channel(128);

    let operation = action();
    tokio::pin!(operation);

    loop {
        tokio::select! {
            _ = &mut operation => break,
            Some(v) = rx.recv() => {
                if v % 2 == 0 {
                    break;
                }
            }
        }
    }
}
```

在上面代码中，我们没有直接在 `select!` 分支中调用 `action()`，而是在 `loop` 循环外面先将 `action()` 赋值给 `operation`，因此 `operation` 是一个 `Future`。

**重点来了**，在 `select!` 循环中，我们使用了一个奇怪的语法 `&mut operation`，大家想象一下，如果不加 `&mut` 会如何？答案是，每一次循环调用的都是一次全新的 `action()` 调用，但是当加了 `&mut` `operatoroion` 后，每一次循环调用就变成了对同一次 `action()` 的调用。也就是我们实现了在每次循环中恢复了之前的异步操作！

`select!` 的另一个分支从消息通道收取消息，一旦收到值是偶数，就跳出循环，否则就继续循环。

还有一个就是我们使用了 `tokio::pin!`，具体的细节这里先不介绍，值得注意的点是：如果要在一个引用上使用 `.await`，那么引用的值就必须是不能移动的或者实现了 `Unpin`，关于 `Pin` 和 `Unpin` 可以参见[这里](#)。

一旦移除 `tokio::pin!` 所在行的代码，然后试图编译，就会获得以下错误：

```
error[E0599]: no method named `poll` found for struct
  `std::pin::Pin<&mut &mut std::future::Future>`
  in the current scope
--> src/main.rs:16:9
|
16 |     tokio::select! {
17 |         _ = &mut operation => break,
18 |         Some(v) = rx.recv() => {
19 |             if v % 2 == 0 {
...
22 |             }
23 |         }
|         ^ method not found in
|         `std::pin::Pin<&mut &mut std::future::Future>`
|
= note: the method `poll` exists but the following trait bounds
        were not satisfied:
        `impl std::future::Future: std::marker::Unpin`
        which is required by
        `&mut impl std::future::Future: std::future::Future`
```

虽然我们已经学了很多关于 Future 的知识，但是这个错误依然不太好理解。但是它不难解决：当你试图在一个引用上调用 .await 然后遇到了 Future 未实现 这种错误时，往往只需要将对应的 Future 进行固定即可：`tokio::pin!(operation);`。

## 修改一个分支

下面一起来看一个稍微复杂一些的 loop 循环，首先，我们拥有：

- 一个消息通道可以传递 i32 类型的值
- 定义在 i32 值上的一个异步操作

想要实现的逻辑是：

- 在消息通道中等待一个偶数出现
- 使用该偶数作为输入来启动一个异步操作
- 等待异步操作完成，与此同时监听消息通道以获取更多的偶数
- 若在异步操作完成前一个新的偶数到了，终止当前的异步操作，然后接着使用新的偶数开始异步操作

```

async fn action(input: Option<i32>) -> Option<String> {
    // 若 input (输入) 是None, 则返回 None
    // 事实上也可以这么写: `let i = input?;`
    let i = match input {
        Some(input) => input,
        None => return None,
    };

    // 这里定义一些逻辑
}

#[tokio::main]
async fn main() {
    let (mut tx, mut rx) = tokio::sync::mpsc::channel(128);

    let mut done = false;
    let operation = action(None);
    tokio::pin!(operation);

    tokio::spawn(async move {
        let _ = tx.send(1).await;
        let _ = tx.send(3).await;
        let _ = tx.send(2).await;
    });

    loop {
        tokio::select! {
            res = &mut operation, if !done => {
                done = true;

                if let Some(v) = res {
                    println!("GOT = {}", v);
                    return;
                }
            }
            Some(v) = rx.recv() => {
                if v % 2 == 0 {
                    // `.set` 是 `Pin` 上定义的方法
                    operation.set(action(Some(v)));
                    done = false;
                }
            }
        }
    }
}

```

当第一次循环开始时，第一个分支会立即完成，因为 `operation` 的参数是 `None`。当第一个分支执行完成时，`done` 会变成 `true`，此时第一个分支的条件将无法被满足，开始执行第二个分支。

当第二个分支收到一个偶数时，`done` 会被修改为 `false`，且 `operation` 被设置了值。此后再一次循环时，第一个分支会被执行，且 `operation` 返回一个 `Some(2)`，因此会触发 `return`，最终结束循环并返回。

这段代码引入了一个新的语法：`if !done`，在解释之前，先看看去掉后会如何：

```
thread 'main' panicked at ``async fn` resumed after completion', src/main.rs:1:55
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```async fn` resumed after completion'` 错误的含义是：`async fn` 异步函数在完成后，依然被恢复了(继续使用)。

回到例子中来，这个错误是由于 `operation` 在它已经调用完成后依然被使用。通常来说，当使用 `.await` 后，调用 `.await` 的值会被消耗掉，因此并不存在这个问题。但是在这例子中，我们在引用上调用 `.await`，因此之后该引用依然可以被使用。

为了避免这个问题，需要在第一个分支的 `operation` 完成后禁止再使用该分支。这里的 `done` 的引入就很好的解决了问题。对于 `select!` 来说 `if !done` 的语法被称为预条件(**precondition**)，该条件会在分支被 `.await` 执行前进行检查。

那大家肯定有疑问了，既然 `operation` 不能再被调用了，我们该如何在有偶数值时，再回到第一个分支对其进行调用呢？答案就是 `operation.set(action(Some(v)));`，该操作会重新使用新的参数设置 `operation`。

## spawn 和 select! 的一些不同

学到现在，相信大家对于 `tokio::spawn` 和 `select!` 已经非常熟悉，它们的共同点就是都可以并发的运行异步操作。然而它们使用的策略大相径庭。

`tokio::spawn` 函数会启动新的任务来运行一个异步操作，每个任务都是一个独立的对象可以单独被 Tokio 调度运行，因此两个不同的任务的调度都是独立进行的，甚至于它们可能会运行在两个不同的操作系统线程上。鉴于此，生成的任务和生成的线程有一个相同的限制：不允许对外部环境中的值进行借用。

而 `select!` 宏就不一样了，它在同一个任务中并发运行所有的分支。正是因为这样，在同一个任务中，这些分支无法被同时运行。`select!` 宏在单个任务中实现了多路复用的功能。

# Stream

大家有没有想过， Rust 中的迭代器在迭代时能否异步进行？若不可以，是不是有相应的解决方案？

以上的问题其实很重要，因为在实际场景中，迭代一个集合，然后异步的去执行是很常见的需求，好在 Tokio 为我们提供了 `stream`，我们可以在异步函数中对其进行迭代，甚至和迭代器 `Iterator` 一样，`stream` 还能使用适配器，例如 `map`！Tokio 在 `StreamExt` 特征上定义了常用的适配器。

要使用 `stream`，目前还需要手动引入对应的包：

```
tokio-stream = "0.1"
```

---

`stream` 没有放在 `tokio` 包的原因在于标准库中的 `Stream` 特征还没有稳定，一旦稳定后，`stream` 将移动到 `tokio` 中来

---

## 迭代

目前，Rust 语言还不支持异步的 `for` 循环，因此我们需要 `while let` 循环和 `StreamExt::next()` 一起使用来实现迭代的目的：

```
use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    let mut stream = tokio_stream::iter(&[1, 2, 3]);

    while let Some(v) = stream.next().await {
        println!("GOT = {:?}", v);
    }
}
```

和迭代器 `Iterator` 类似，`next()` 方法返回一个 `Option<T>`，其中 `T` 是从 `stream` 中获取的值的类型。若收到 `None` 则意味着 `stream` 迭代已经结束。

## mini-redis 广播

下面我们来实现一个复杂一些的 mini-redis 客户端，完整代码见[这里](#)。

在开始之前，首先启动一下完整的 mini-redis 服务器端：

```
$ mini-redis-server

use tokio_stream::StreamExt;
use mini_redis::client;

async fn publish() -> mini_redis::Result<()> {
    let mut client = client::connect("127.0.0.1:6379").await?;

    // 发布一些数据
    client.publish("numbers", "1".into()).await?;
    client.publish("numbers", "two".into()).await?;
    client.publish("numbers", "3".into()).await?;
    client.publish("numbers", "four".into()).await?;
    client.publish("numbers", "five".into()).await?;
    client.publish("numbers", "6".into()).await?;
    Ok(())
}

async fn subscribe() -> mini_redis::Result<()> {
    let client = client::connect("127.0.0.1:6379").await?;
    let subscriber = client.subscribe(vec!["numbers".to_string()]).await?;
    let messages = subscriber.into_stream();

    tokio::pin!(messages);

    while let Some(msg) = messages.next().await {
        println!("got = {:?}", msg);
    }

    Ok(())
}

#[tokio::main]
async fn main() -> mini_redis::Result<()> {
    tokio::spawn(async {
        publish().await
    });

    subscribe().await?;

    println!("DONE");

    Ok(())
}
```

上面生成了一个异步任务专门用于发布消息到 min-redis 服务器端的 `numbers` 消息通道中。然后，在 `main` 中，我们订阅了 `numbers` 消息通道，并且打印从中接收到的消息。

还有几点值得注意的:

- `into_stream` 会将 `Subscriber` 变成一个 `stream`
- 在 `stream` 上调用 `next` 方法要求该 `stream` 被固定住(`pinned`)，因此需要调用 `tokio::pin!`

---

关于 Pin 的详细解读，可以阅读[这篇文章](#)

大家可以去掉 `pin!` 的调用，然后观察下报错，若以后你遇到这种错误，可以尝试使用下 `pin!`。

此时，可以运行下我们的客户端代码看看效果(别忘了先启动前面提到的 mini-redis 服务端):

```
got = Ok(Message { channel: "numbers", content: b"1" })
got = Ok(Message { channel: "numbers", content: b"two" })
got = Ok(Message { channel: "numbers", content: b"3" })
got = Ok(Message { channel: "numbers", content: b"four" })
got = Ok(Message { channel: "numbers", content: b"five" })
got = Ok(Message { channel: "numbers", content: b"6" })
```

在了解了 `stream` 的基本用法后，我们再来看看如何使用适配器来扩展它。

## 适配器

在前面章节中，我们了解了迭代器有[两种适配器](#):

- 迭代器适配器，会将一个迭代器转变成另一个迭代器，例如 `map`，`filter` 等
- 消费者适配器，会消费掉一个迭代器，最终生成一个值，例如 `collect` 可以将迭代器收集成一个集合

与迭代器类似，`stream` 也有适配器，例如一个 `stream` 适配器可以将一个 `stream` 转变成另一个 `stream`，例如 `map`、`take` 和 `filter`。

在之前的客户端中，`subscribe` 订阅一直持续下去，直到程序被关闭。现在，让我们来升级下，让它在收到三条消息后就停止迭代，最终结束。

```
let messages = subscriber
    .into_stream()
    .take(3);
```

这里关键就在于 `take` 适配器，它会限制 `stream` 只能生成最多 `n` 条消息。运行下看看结果：

```
got = Ok(Message { channel: "numbers", content: b"1" })
got = Ok(Message { channel: "numbers", content: b"two" })
got = Ok(Message { channel: "numbers", content: b"3" })
```

程序终于可以正常结束了。现在，让我们过滤 stream 中的消息，只保留数字类型的值：

```
let messages = subscriber
    .into_stream()
    .filter(|msg| match msg {
        Ok(msg) if msg.content.len() == 1 => true,
        _ => false,
    })
    .take(3);
```

运行后输出：

```
got = Ok(Message { channel: "numbers", content: b"1" })
got = Ok(Message { channel: "numbers", content: b"3" })
got = Ok(Message { channel: "numbers", content: b"6" })
```

需要注意的是，适配器的顺序非常重要，`.filter(...).take(3)` 和 `.take(3).filter(...)` 的结果可能大相径庭，大家可以自己尝试下。

现在，还有一件事要做，咱们的消息被不太好看的 `Ok(...)` 所包裹，现在通过 `map` 适配器来简化下：

```
let messages = subscriber
    .into_stream()
    .filter(|msg| match msg {
        Ok(msg) if msg.content.len() == 1 => true,
        _ => false,
    })
    .map(|msg| msg.unwrap().content)
    .take(3);
```

注意到 `msg.unwrap()` 了吗？大家可能会以为我们是出于示例的目的才这么用，实际上并不是，由于 `filter` 的先执行，`map` 中的 `msg` 只能是 `Ok(...)`，因此 `unwrap` 非常安全。

```
got = b"1"
got = b"3"
got = b"6"
```

还有一点可以改进的地方：当 `filter` 和 `map` 一起使用时，你往往可以用一个统一的方法来实现 `filter_map`。

```
let messages = subscriber
    .into_stream()
    .filter_map(|msg| match msg {
        Ok(msg) if msg.content.len() == 1 => Some(msg.content),
        _ => None,
    })
    .take(3);
```

想要学习更多的适配器，可以看看 [StreamExt](#) 特征。

## 实现 Stream 特征

如果大家还没忘记 Future 特征，那 stream 特征相信你也会很快记住，因为它们非常类似：

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;

    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }
}
```

Stream::poll\_next() 函数跟 Future::poll 很相似，区别就是前者为了从 stream 收到多个值需要重复的进行调用。就像在 [深入async](#) 章节提到的那样，当一个 stream 没有做好返回一个值的准备时，它将返回一个 Poll::Pending，同时将任务的 waker 进行注册。一旦 stream 准备好后，waker 将被调用。

通常来说，如果想要手动实现一个 Stream，需要组合 Future 和其它 Stream。下面，还记得在 [深入async](#) 中构建的 Delay Future 吗？现在让我们来更进一步，将它转换成一个 stream，每 10 毫秒生成一个值，总共生成 3 次：

```

use tokio_stream::Stream;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::Duration;

struct Interval {
    rem: usize,
    delay: Delay,
}

impl Stream for Interval {
    type Item = ();

    fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<()>>
    {
        if self.rem == 0 {
            // 去除计时器实现
            return Poll::Ready(None);
        }

        match Pin::new(&mut self.delay).poll(cx) {
            Poll::Ready(_) => {
                let when = self.delay.when + Duration::from_millis(10);
                self.delay = Delay { when };
                self.rem -= 1;
                Poll::Ready(Some(()))
            }
            Poll::Pending => Poll::Pending,
        }
    }
}

```

## async-stream

手动实现 `Stream` 特征实际上是相当麻烦的事，不幸地是，Rust 语言的 `async/await` 语法目前还不能用于定义 `stream`，虽然相关的工作已经在进行中。

作为替代方案，`async-stream` 包提供了一个 `stream!` 宏，它可以将一个输入转换成 `stream`，使用这个包，上面的代码可以这样实现：

```
use async_stream::stream;
use std::time::{Duration, Instant};

stream! {
    let mut when = Instant::now();
    for _ in 0..3 {
        let delay = Delay { when };
        delay.await;
        yield ();
        when += Duration::from_millis(10);
    }
}
```

嗯，看上去还是相当不错的，代码可读性大幅提升！

是不是发现了一个关键字 `yield`，他是用来配合生成器使用的。详见[原文](#)

# 优雅的关闭

如果你的服务是一个小说阅读网站，那大概率用不到优雅关闭的，简单粗暴的关闭服务器，然后用户再次请求时获取一个错误就是了。但如果是一个 web 服务或数据库服务呢？当前的连接很可能在做着重要的事情，一旦关闭会导致数据的丢失甚至错误，此时，我们就需要优雅的关闭(graceful shutdown)了。

要让一个异步应用优雅的关闭往往需要做到 3 点：

- 找出合适的关闭时机
- 通知程序的每一个子部分开始关闭
- 在主线程等待各个部分的关闭结果

在本文的下面部分，我们一起来看看该如何做到这三点。如果想要进一步了解在真实项目中该如何使用，大家可以看看 mini-redis 的完整代码实现，特别是 `src/server.rs` 和 `src/shutdown.rs`。

## 找出合适的关闭时机

一般来说，何时关闭是取决于应用自身的，但是一个常用的关闭准则就是当应用收到来自于操作系统的关闭信号时。例如通过 `ctrl + c` 来关闭正在运行的命令行程序。

为了检测来自操作系统的关闭信号，Tokio 提供了一个 `tokio::signal::ctrl_c` 函数，它将一直睡眠直到收到对应的信号：

```
use tokio::signal;

#[tokio::main]
async fn main() {
    // ... spawn application as separate task ...
    // 在一个单独的任务中处理应用逻辑

    match signal::ctrl_c().await {
        Ok(_) => {},
        Err(err) => {
            eprintln!("Unable to listen for shutdown signal: {}", err);
        },
    }

    // 发送关闭信号给应用所在的任务，然后等待
}
```

## 通知程序的每一个部分开始关闭

大家看到这个标题，不知道会想到用什么技术来解决问题，反正我首先想到的是，真的很像广播哎。。

事实上也是如此，最常见的通知程序各个部分关闭的方式就是使用一个广播消息通道。关于如何实现，其实也不复杂：应用中的每个任务都持有一个广播消息通道的接收端，当消息被广播到该通道时，每个任务都可以收到该消息，并关闭自己：

```
let next_frame = tokio::select! {
    res = self.connection.read_frame() => res?,
    _ = self.shutdown.recv() => {
        // 当收到关闭信号后，直接从 `select!` 返回，此时 `select!` 中的另一个分支会自动释放，其中的任务也会结束
        return Ok(());
    }
};
```

在 `mini-redis` 中，当收到关闭消息时，任务会立即结束，但在实际项目中，这种方式可能会过于理想，例如当我们向文件或数据库写入数据时，立刻终止任务可能会导致一些无法预料的错误，因此，在结束前做一些收尾工作会是非常好的选择。

除此之外，还有两点值得注意：

- 将广播消息通道作为结构体的一个字段是相当不错的选择，例如[这个例子](#)
- 还可以使用 `watch channel` 实现同样的效果，与之前的方式相比，这两种方法并没有太大的区别

## 等待各个部分的结束

在之前章节，我们讲到过一个 `mpsc` 消息通道有一个重要特性：当所有发送端都 `drop` 时，消息通道会自动关闭，此时继续接收消息就会报错。

大家发现没？这个特性特别适合优雅关闭的场景：主线程持有消息通道的接收端，然后每个代码部分拿走一个发送端，当该部分结束时，就 `drop` 掉发送端，因此所有发送端被 `drop` 也就意味着所有的部分都已关闭，此时主线程的接收端就会收到错误，进而结束。

```
use tokio::sync::mpsc::{channel, Sender};
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let (send, mut recv) = channel(1);

    for i in 0..10 {
        tokio::spawn(some_operation(i, send.clone()));
    }

    // 等待各个任务的完成
    //
    // 我们需要 drop 自己的发送端，因为等下的 `recv()` 调用会阻塞，如果不 `drop`，那发送端就无法被全部关闭
    // `recv` 也将永远无法结束，这将陷入一个类似死锁的困境
    drop(send);

    // 当所有发送端都超出作用域被 `drop` 时（当前的发送端并不是因为超出作用域被 `drop`，而是手动 `drop` 的）
    // `recv` 调用会返回一个错误
    let _ = recv.recv().await;
}

async fn some_operation(i: u64, _sender: Sender<()>) {
    sleep(Duration::from_millis(100 * i)).await;
    println!("Task {} shutting down.", i);

    // 发送端超出作用域，然后被 `drop`
}
```

关于忘记 `drop` 本身持有的发送端进而导致 bug 的问题，大家可以看看这篇文章。[这篇文章](#)。

# 异步跟同步共存

一些异步程序例如 tokio 指南 章节中的绝大多数例子，它们整个程序都是异步的，包括程序入口 `main` 函数：

```
#[tokio::main]
async fn main() {
    println!("Hello world");
}
```

在一些场景中，你可能只想在异步程序中运行一小部分同步代码，这种需求可以考虑下 `spawn_blocking`。

但是在很多场景中，我们只想让程序的某一个部分成为异步的，也许是因为同步代码更好实现，又或许是同步代码可读性、兼容性都更好。例如一个 GUI 应用可能想要让 UI 相关的代码在主线程中，然后通过另一个线程使用 `tokio` 的运行时来处理一些异步任务。

因此本章节的目标很纯粹：如何在同步代码中使用一小部分异步代码。

## #[tokio::main] 的展开

在 Rust 中，`main` 函数不能是异步的，有同学肯定不愿意了，我们在之前章节..不对，就在开头，你还用到了 `async fn main` 的声明方式，怎么就不能异步了呢？

其实，`#[tokio::main]` 该宏仅仅是提供语法糖，目的是让大家可以更简单、更一致的去写异步代码，它会将你写下的 `async fn main` 函数替换为：

```
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            println!("Hello world");
        })
}
```

注意到上面的 `block_on` 方法了吗？在我们自己的同步代码中，可以使用它开启一个 `async/await` 世界。

## mini-redis 的同步接口

在下面，我们将一起构建一个同步的 `mini-redis`，为了实现这一点，需要将 `Runtime` 对象存储起来，然后利用上面提到的 `block_on` 方法。

首先，创建一个文件 `src/blocking_client.rs`，然后使用下面代码将异步的 `Client` 结构体包裹起来：

```
use tokio::net::ToSocketAddrs;
use tokio::runtime::Runtime;

pub use crate::client::Message;

/// 建立到 redis 服务端的连接
pub struct BlockingClient {
    /// 之前实现的异步客户端 `Client`
    inner: crate::client::Client,
    /// 一个 `current_thread` 模式的 `tokio` 运行时,
    /// 使用阻塞的方式来执行异步客户端 `Client` 上的操作
    rt: Runtime,
}

pub fn connect<T: ToSocketAddrs>(addr: T) -> crate::Result<BlockingClient> {
    // 构建一个 tokio 运行时: Runtime
    let rt = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()?;
    // 使用运行时来调用异步的连接方法
    let inner = rt.block_on(crate::client::connect(addr))?;
    Ok(BlockingClient { inner, rt })
}
```

在这里，我们使用了一个构造器函数用于在同步代码中执行异步的方法：使用 `Runtime` 上的 `block_on` 方法来执行一个异步方法并返回结果。

有一个很重要的点，就是我们还使用了 `current_thread` 模式的运行时。这个可不常见，原因是异步程序往往要利用多线程的威力来实现更高的吞吐性能，相对应的模式就是 `multi_thread`，该模式会生成多个运行在后台的线程，它们可以高效的实现多个任务的同时并行处理。

但是对于我们的使用场景来说，在同一时间点只需要做一件事，无需并行处理，多个线程并不能帮助到任何事情，因此 `current_thread` 此时成为了最佳的选择。

在构建 `Runtime` 的过程中还有一个 `enable_all` 方法调用，它可以开启 Tokio 运行时提供的 IO 和定时器服务。

---

由于 `current_thread` 运行时并不生成新的线程，只是运行在已有的主线程上，因此只有当 `block_on` 被调用后，该运行时才能执行相应的操作。一旦 `block_on` 返回，那运行时上所有生成的任务将再次冻结，直到 `block_on` 的再次调用。

如果这种模式不符合使用场景的需求，那大家还是需要用 `multi_thread` 运行时来代替。事实上，在 tokio 之前的章节中，我们默认使用的就是 `multi_thread` 模式。

---

```
use bytes::Bytes;
use std::time::Duration;

impl BlockingClient {
    pub fn get(&mut self, key: &str) -> crate::Result<Option<Bytes>> {
        self.rt.block_on(self.inner.get(key))
    }

    pub fn set(&mut self, key: &str, value: Bytes) -> crate::Result<()> {
        self.rt.block_on(self.inner.set(key, value))
    }

    pub fn set_expires(
        &mut self,
        key: &str,
        value: Bytes,
        expiration: Duration,
    ) -> crate::Result<()> {
        self.rt.block_on(self.inner.set_expires(key, value, expiration))
    }

    pub fn publish(&mut self, channel: &str, message: Bytes) -> crate::Result<u64> {
        self.rt.block_on(self.inner.publish(channel, message))
    }
}
```

这代码看上去挺长，实际上很简单，通过 `block_on` 将异步形式的 `client` 的方法变成同步调用的形式。例如 `BlockingClient` 的 `get` 方法实际上是对内部的异步 `get` 方法的同步调用。

与上面的平平无奇相比，下面的代码将更有趣，因为它将 `Client` 转变成一个 `Subscriber` 对象：

```

/// 下面的客户端可以进入 pub/sub (发布/订阅) 模式
///
/// 一旦客户端订阅了某个消息通道，那就只能执行 pub/sub 相关的命令。
/// 将`BlockingClient` 类型转换成 `BlockingSubscriber` 是为了防止非 `pub/sub` 方法被调用
pub struct BlockingSubscriber {
    /// 异步版本的 `Subscriber`
    inner: crate::client::Subscriber,
    /// 一个 `current_thread` 模式的 `tokio` 运行时,
    /// 使用阻塞的方式来执行异步客户端 `Client` 上的操作
    rt: Runtime,
}

impl BlockingClient {
    pub fn subscribe(self, channels: Vec<String>) -> crate::Result<BlockingSubscriber> {
        let subscriber = self.rt.block_on(self.inner.subscribe(channels))?;
        Ok(BlockingSubscriber {
            inner: subscriber,
            rt: self.rt,
        })
    }
}

impl BlockingSubscriber {
    pub fn get_subscribed(&self) -> &[String] {
        self.inner.get_subscribed()
    }

    pub fn next_message(&mut self) -> crate::Result<Option<Message>> {
        self.rt.block_on(self.inner.next_message())
    }

    pub fn subscribe(&mut self, channels: &[String]) -> crate::Result<()> {
        self.rt.block_on(self.inner.subscribe(channels))
    }

    pub fn unsubscribe(&mut self, channels: &[String]) -> crate::Result<()> {
        self.rt.block_on(self.inner.unsubscribe(channels))
    }
}

```

由上可知，`subscribe` 方法会使用运行时将一个异步的 `Client` 转变成一个异步的 `Subscriber`，此外，`Subscriber` 结构体有一个非异步的方法 `get_subscribed`，对于这种方法，只需直接调用即可，而无需使用运行时。

# 其它方法

上面介绍的是最简单的方法，但是，如果只有这一种，tokio 也不会如此大名鼎鼎。

## runtime.spawn

可以通过 `Runtime` 的 `spawn` 方法来创建一个基于该运行时的后台任务：

```
use tokio::runtime::Builder;
use tokio::time::{sleep, Duration};

fn main() {
    let runtime = Builder::new_multi_thread()
        .worker_threads(1)
        .enable_all()
        .build()
        .unwrap();

    let mut handles = Vec::with_capacity(10);
    for i in 0..10 {
        handles.push(runtime.spawn(my_bg_task(i)));
    }

    // 在后台任务运行的同时做一些耗费时间的事情
    std::thread::sleep(Duration::from_millis(750));
    println!("Finished time-consuming task.");

    // 等待这些后台任务的完成
    for handle in handles {
        // `spawn` 方法返回一个 `JoinHandle`，它是一个 `Future`，因此可以通过 `block_on` 来等待它完成
        runtime.block_on(handle).unwrap();
    }
}

async fn my_bg_task(i: u64) {
    let millis = 1000 - 50 * i;
    println!("Task {} sleeping for {} ms.", i, millis);

    sleep(Duration::from_millis(millis)).await;

    println!("Task {} stopping.", i);
}
```

运行该程序，输出如下：

```
Task 0 sleeping for 1000 ms.  
Task 1 sleeping for 950 ms.  
Task 2 sleeping for 900 ms.  
Task 3 sleeping for 850 ms.  
Task 4 sleeping for 800 ms.  
Task 5 sleeping for 750 ms.  
Task 6 sleeping for 700 ms.  
Task 7 sleeping for 650 ms.  
Task 8 sleeping for 600 ms.  
Task 9 sleeping for 550 ms.  
Task 9 stopping.  
Task 8 stopping.  
Task 7 stopping.  
Task 6 stopping.  
Finished time-consuming task.  
Task 5 stopping.  
Task 4 stopping.  
Task 3 stopping.  
Task 2 stopping.  
Task 1 stopping.  
Task 0 stopping.
```

在此例中，我们生成了 10 个后台任务在运行时中运行，然后等待它们的完成。作为一个例子，想象一下在图形渲染应用( GUI )中，有时候需要通过网络访问远程服务来获取一些数据，那上面的这种模式就非常适合，因为这些网络访问比较耗时，而且不会影响图形的主体渲染，因此可以在主线程中渲染图形，然后使用其它线程来运行 Tokio 的运行时，并通过该运行时使用异步的方式完成网络访问，最后将这些网络访问的结果发送到 GUI 进行数据渲染，例如一个进度条。

还有一点很重要，在本例子中只能使用 `multi_thread` 运行时。如果我们使用了 `current_thread`，你会发现主线程的耗时任务会在后台任务开始之前就完成了。因为在 `current_thread` 模式下，生成的任务只会在 `block_on` 期间才执行。

在 `multi_thread` 模式下，我们并不需要通过 `block_on` 来触发任务的运行，这里仅仅是用来阻塞并等待最终的结果。而除了通过 `block_on` 等待结果外，你还可以：

- 使用消息传递的方式，例如 `tokio::sync::mpsc`，让异步任务将结果发送到主线程，然后主线程通过 `.recv` 方法等待这些结果
- 通过共享变量的方式，例如 `Mutex`，这种方式非常适合实现 GUI 的进度条：GUI 在每个渲染帧读取该变量即可。

## 发送消息

在同步代码中使用异步的另一个方法就是生成一个运行时，然后使用消息传递的方式跟它进行交互。这个方法虽然更啰嗦一些，但是相对于之前的两种方法更加灵活：

```
use tokio::runtime::Builder;
use tokio::sync::mpsc;

pub struct Task {
    name: String,
    // 一些信息用于描述该任务
}

async fn handle_task(task: Task) {
    println!("Got task {}", task.name);
}

#[derive(Clone)]
pub struct TaskSpawner {
    spawn: mpsc::Sender<Task>,
}

impl TaskSpawner {
    pub fn new() -> TaskSpawner {
        // 创建一个消息通道用于通信
        let (send, mut recv) = mpsc::channel(16);

        let rt = Builder::new_current_thread()
            .enable_all()
            .build()
            .unwrap();

        std::thread::spawn(move || {
            rt.block_on(async move {
                while let Some(task) = recv.recv().await {
                    tokio::spawn(handle_task(task));
                }
            })
            // 一旦所有的发送端超出作用域被 drop 后，`recv()` 方法会返回 None，同时
            // 循环会退出，然后线程结束
        });
    }

    TaskSpawner {
        spawn: send,
    }
}

pub fn spawn_task(&self, task: Task) {
    match self.spawn.blocking_send(task) {
        Ok(_) => {},
        Err(_) => panic!("The shared runtime has shut down."),
    }
}
```

为何说这种方法比较灵活呢？以上面代码为例，它可以在很多方面进行配置。例如，可以使用信号量 [Semaphore](#) 来限制当前正在进行的任务数，或者你还可以使用一个消息通道将消息反向发送回任务生成器 `spawner`。

抛开细节，抽象来看，这是不是很像一个 Actor？

# Rust 难点攻关

当大家一路看到这里时，我敢说 90% 的人还是云里雾里的，例如你能说清楚：

- 切片和切片引用的区别吗？
- 各种字符串之间的区别吗？
- 各种指针、引用的区别吗？
- 所有权转移、拷贝、克隆的区别吗？

以及到底该用它们之中哪一个吗？

如果不行，就跟随我一起来看看吧，本章的目标就是帮大家彻底理清这些概念，为后面的进一步学习和实战打好坚实的基础。

# 切片和切片引用

关于 `str / &str , [u8] / &[u8]` 区别，你能清晰的说出来嘛？如果答案是 No，那就跟随我一起来看看切片和切片引用到底有何区别吧。

---

在继续之前，查看[这里](#)了解何为切片

---

切片允许我们引用集合中部分连续的元素序列，而不是引用整个集合。例如，字符串切片就是一个子字符串，数组切片就是一个子数组。

## 无法被直接使用的切片类型

Rust 语言特性内置的 `str` 和 `[u8]` 类型都是切片，前者是字符串切片，后者是数组切片，下面我们来尝试下使用 `str`：

```
let string: str = "banana";
```

上面代码创建一个 `str` 类型的字符串，看起来很正常，但是编译就会报错：

```
error[E0277]: the size for values of type `str` cannot be known at compilation time
--> src/main.rs:4:9
 |
4 |     let string: str = "banana";
|     ^^^^^^ doesn't have a size known at compile-time
```

编译器准确的告诉了我们原因：`str` 字符串切片它是 [DST 动态大小类型](#)，这意味着编译器无法在编译期知道 `str` 类型的大小，只有到了运行期才能动态获知，这对于强类型、强安全的 Rust 语言来说是不可接受的。

也就是说，我们无法直接使用 `str`，而对于 `[u8]` 也是类似的，大家可以自己动手试试。

总之，我们可以总结出一个结论：**在 Rust 中，所有的切片都是动态大小类型，它们都无法直接被使用。**

### 为何切片是动态大小类型

原因在于底层的切片长度是可以动态变化的，而编译器无法在编译期得知它的具体的长度，因此该类型无法被分配在栈上，只能分配在堆上。

## 为何切片只能通过引用来使用

既然切片只能分配到堆上，我们就无法直接使用它，大家可以想想，所有分配在堆上的数据，是不是都是通过一个在栈上的引用来访问的？切片也不例外。

## 为何切片引用可以存储在栈上

切片引用是一个宽指针，存储在栈上，指向了堆上的切片数据，该引用包含了切片的起始位置和长度，而且最重要的是，类似于指针，引用的大小是固定的(起始位置和长度都是整形)，因此它才可以存储在栈上。

## 有没有可以存储在栈上的

有，使用固定长度的数组: `let a: [i8;4] = [1,2,3,4];`，注意看，数组的类型与切片是不同的，前者的类型带有长度: `[i8;4]`，而后者仅仅是 `[i8]`。

# 切片引用

那么问题来了，该如何使用切片呢？

何以解忧，唯有引用。由于引用类型的大小在编译期是已知的，因此在 Rust 中，如果要使用切片，就必须要使用它的引用。

`str` 切片的引用类型是 `&str`，而 `[i32]` 的引用类型是 `&[i32]`，相信聪明的读者已经看出来了，`&str` 和 `&[i32]` 都是我们非常常用的类型，例如：

```
let s1: &str = "banana";
let s2: &str = &String::from("banana");

let arr = [1, 2, 3, 4, 5];

let s3: &[i32] = &arr[1..3];
```

这段代码就可以正常通过，原因在于这些切片引用的大小在编译器都是已知的。

## 总结

我们常常说使用切片，实际上我们在用的是切片的引用，我们也在频繁说使用字符串，实际上我们在使用的也是字符串切片的引用。

总之，切片在 Rust 中是动态大小类型 DST，是无法被我们直接使用的，而我们在使用的都是切片的引用。

切片	切片引用
str 字符串切片	&str 字符串切片的引用
[u8] 数组切片	&[u8] 数组切片的引用

**但是出于方便，我们往往不会说使用切片引用，而是直接说使用字符串切片或数组切片，实际上，这时指代的都是切片的引用！**

# Eq 和 PartialEq

在 Rust 中，想要重载操作符，你就需要实现对应的特征。

例如 <、<=、> 和 >= 需要实现 `PartialOrd` 特征：

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

再比如，+ 号需要实现 `std::ops::Add` 特征，而本文的主角 `Eq` 和 `PartialEq` 正是 `==` 和 `!=` 所需的特征，那么问题来了，这两个特征有何区别？

我相信很多同学都说不太清楚，包括一些老司机，而且就算是翻文档，可能也找不到特别明确的解释。如果大家看过标准库示例，可能会看过这个例子：

```
enum BookFormat { Paperback, Hardback, Ebook }

struct Book {
    isbn: i32,
    format: BookFormat,
}

impl PartialEq for Book {
    fn eq(&self, other: &Self) -> bool {
        self.isbn == other.isbn
    }
}

impl Eq for Book {}
```

这里只实现了 `PartialEq`，并没有实现 `Eq`，而是直接使用了默认实现 `impl Eq for Book {}`，奇了怪了，别急，还有呢：

```
impl PartialEq<IpAddr> for Ipv4Addr {
    #[inline]
    fn eq(&self, other: &IpAddr) -> bool {
        match other {
            IpAddr::V4(v4) => self == v4,
            IpAddr::V6(_) => false,
        }
    }
}

impl Eq for Ipv4Addr {}
```

以上代码来自 Rust 标准库，可以看到，依然是这样使用，类似的情况数不胜数。既然如此，是否说明**如果要为我们的类型增加相等性比较，只要实现 `PartialEq` 即可？**

其实，关键点就在于 `partial` 上，**如果我们的类型只在部分情况下具有相等性，那你就只能实现 `PartialEq`，否则可以实现 `PartialEq` 然后再默认实现 `Eq`。**

好的，问题逐步清晰起来，现在我们只需要搞清楚何为部分相等。

## 部分相等性

首先我们需要找到一个类型，它实现了 `PartialEq` 但是没有实现 `Eq`（你可能会想有没有反过来的情况？当然没有啦，部分相等肯定是全部相等的子集！）

在 `HashMap` 章节提到过 `HashMap` 的 `key` 要求实现 `Eq` 特征，也就是要能完全相等，而浮点数由于没有实现 `Eq`，因此不能用于 `HashMap` 的 `key`。

当时由于一些知识点还没有介绍，因此就没有进一步展开，那么让我们考虑浮点数既然没有实现 `Eq` 为何还能进行比较呢？

```
fn main() {
    let f1 = 3.14;
    let f2 = 3.14;

    if f1 == f2 {
        println!("hello, world!");
    }
}
```

以上代码是可以看到输出内容的，既然浮点数没有实现 `Eq` 那说明它实现了 `PartialEq`，一起写个简单代码验证下：

```
fn main() {
    let f1 = 3.14;
    is_eq(f1);
    is_partial_eq(f1)
}

fn is_eq<T: Eq>(f: T) {}
fn is_partial_eq<T: PartialEq>(f: T) {}
```

上面的代码通过特征约束的方式验证了我们的结论:

```
3 |     is_eq(f1);
|----- ^^^ the trait `Eq` is not implemented for `float`
```

好的，既然我们成功找到了一个类型实现了 `PartialEq` 但没有实现 `Eq`，那就通过它来看看何为部分相等性。

其实答案很简单，浮点数有一个特殊的值 `Nan`，它是无法进行相等性比较的:

```
fn main() {
    let f1 = f32::NAN;
    let f2 = f32::NAN;

    if f1 == f2 {
        println!("Nan 竟然可以比较，这很不数学啊！")
    } else {
        println!("果然，虽然两个都是 Nan，但是它们其实并不相等")
    }
}
```

大家猜猜哪一行会输出 :) 至于 `Nan` 为何不能比较，这个原因就比较复杂了(有读者会说，其实就是你不知道，我只能义正严辞的说：咦？你怎么知道 :P )。

既然浮点数有一个值不可以比较相等性，那它自然只能实现 `PartialEq` 而不能实现 `Eq` 了，以此类推，如果我们的类型也有这种特殊要求，那也应该这么做。

## Ord 和 PartialOrd

事实上，还有一对与 `Eq/PartialEq` 非常类似的特征，它们可以用于 `<`、`<=`、`>` 和 `>=` 比较，至于哪个类型实现了 `PartialOrd` 却没有实现 `Ord` 就交给大家自己来思考了：)

---

小提示：`Ord` 意味着一个类型的所有值都可以进行排序，而 `PartialOrd` 则不然

---

# 疯狂字符串

字符串让人疯狂，这句话用在 Rust 中一点都不夸张，不信？那你能否清晰的说出 `String`、`str`、`&str`、`&String`、`Box<str>` 或 `Box<&str>` 的区别？

Rust 语言的类型可以大致分为两种：基本类型和标准库类型，前者是由语言特性直接提供的，而后者是在标准库中定义。即将登场的 `str` 类型就是唯一定义在语言特性中的字符串。

---

在继续之前，大家需要先了解字符串的[基本知识](#)，本文主要在于概念对比，而不是字符串讲解

---

## str

如上所述，`str` 是唯一定义在 Rust 语言特性中的字符串，但是也是我们几乎不会用到的字符串类型，为何？

原因在于 `str` 字符串它是 [DST 动态大小类型](#)，这意味着编译器无法在编译期知道 `str` 类型的大小，只有到了运行期才能动态获知，这对于强类型、强安全的 Rust 语言来说是不可接受的。

```
let string: str = "banana";
```

上面代码创建一个 `str` 类型的字符串，看起来很正常，但是编译就会报错：

```
error[E0277]: the size for values of type `str` cannot be known at compilation time
--> src/main.rs:4:9
 |
4 |     let string: str = "banana";
|     ^^^^^^ doesn't have a size known at compile-time
```

如果追求更深层的原因，我们可以总结如下：**所有的切片都是动态类型，它们都无法直接被使用，而 `str` 就是字符串切片，`[u8]` 是数组切片。**

同时还是 `String` 和 `&str` 的底层数据类型。由于 `str` 是动态

`str` 类型是硬编码进可执行文件，也无法被修改，但是 `String` 则是一个可增长、可改变且具有所有权的 UTF-8 编码字符串，当 Rust 用户提到字符串时，往往指的就是 `String` 类型和 `&str` 字符串切片类型，这两个类型都是 UTF-8 编码。

除了 `String` 类型的字符串，Rust 的标准库还提供了其他类型的字符串，例如 `OsString`，`OsStr`，`CString` 和 `CStr` 等，注意到这些名字都以 `String` 或者 `Str` 结尾了吗？它们分别对应的是具有所有权和被借用的变量。

## 未完待续

[https://pic1.zhimimg.com/80/v2-177bce575bfaf289ae12d677689a26f4\\_1440w.png](https://pic1.zhimimg.com/80/v2-177bce575bfaf289ae12d677689a26f4_1440w.png)

[https://pic2.zhimimg.com/80/v2-697ad53cb502cc4b2e98c40975344f\\_1440w.png](https://pic2.zhimimg.com/80/v2-697ad53cb502cc4b2e98c40975344f_1440w.png)

<https://medium.com/@alisomay/strings-in-rust-28c08a2d3130>

# 作用域、生命周期和 NLL todo

# **move、Copy和Clone todo**

# **裸指针、引用和智能指针 TODO**

# 测试

---

测试可以有效的发现程序存在的缺陷，但是它却无法证明程序不存在缺陷 - Edsger W. Dijkstra,  
"The Humble Programmer"

---

对于程序开发而言，测试可以说是至关重要的一环，虽然它无法完全消除所有的 Bug，但是依然可以在某种程度上保证程序的正确性。

Rust 语言本身就非常关注安全性，但是语言级别的安全性并不能保证代码逻辑上的正确性，因为后者其实是一个相当复杂的问题。也许 Rust 的类型系统可以提供一些帮助，但是依然远远不够。

例如，假设我们有一个函数 `add_two` 用于将两个整数进行相加并返回一个整数结果。没错，Rust 的类型系统可以通过函数签名确保我们的输入和输出类型都是正确的，譬如你无法传入一个字符串作为输入，但是 Rust 无法保证函数中代码逻辑的正确性：明明目标是相加操作，却给整成了 `x - y`。

好在，写测试可以解决类似的问题。但也不要迷信测试，文章开头的那句话说明一切。

# 编写测试及控制执行

在 Rust 中，测试是通过函数的方式实现的，它可用于验证被测试代码的正确性。测试函数往往依次执行以下三种行为：

1. 设置所需的数据或状态
2. 运行想要测试的代码
3. 判断(`assert`)返回的结果是否符合预期

让我们来看看该如何使用 Rust 提供的特性来按照上述步骤编写测试用例。

## 测试函数

当使用 `Cargo` 创建一个 `lib` 类型的包时，它会为我们自动生成一个测试模块。先来创建一个 `lib` 类型的 `adder` 包：

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

创建成功后，在 `src/lib.rs` 文件中可以发现如下代码：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

其中，`tests` 就是一个测试模块，`it_works` 则是我们的主角：测试函数。

可以看出，测试函数需要使用 `test` 属性进行标注。关于属性(`attribute`)，我们在之前的章节已经见过类似的 `derive`，使用它可以派生自动实现的 `Debug`、`Copy` 等特征，同样的，使用 `test` 属性，我们也可以获取 Rust 提供的测试特性。

经过 `test` 标记的函数就可以被测试执行器发现，并进行运行。当然，在测试模块 `tests` 中，还可以定义非测试函数，这些函数可以用于设置环境或执行一些通用操作：例如为部分测试函数提供某个通用的功能，这种功能就可以抽象为一个非测试函数。

换而言之，正是因为测试模块既可以定义测试函数又可以定义非测试函数，导致了我们必须提供一个特殊的标记 `test`，用于告知哪个函数才是测试函数。

### `assert_eq`

在测试函数中，还使用到了一个内置的断言：`assert_eq`，该宏用于对结果进行断言：`2 + 2` 是否等于 `4`。与之类似，Rust 还内置了其它一些实用的断言，具体参见[后续章节](#)。

## cargo test

下面使用 `cargo test` 命令来运行项目中的所有测试：

```
$ cargo test
Compiling adder v0.1.0 (file:///projects/adder)
Finished test [unoptimized + debuginfo] target(s) in 0.57s
Running unit tests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

上面测试输出中，有几点值得注意：

- 测试用例是分批执行的，`running 1 test` 表示下面的输出 `test result` 来自一个测试用例的运行结果。
- `test tests::it_works` 中包含了测试用例的名称
- `test result: ok` 中的 `ok` 表示测试成功通过
- `1 passed` 代表成功通过一个测试用例(因为只有一个)，`0 failed`：没有测试用例失败，`0 ignored` 说明我们没有将任何测试函数标记为运行时可忽略，`0 filtered` 意味着没有对测试结果做任何过滤，`0 mesasured` 代表[基准测试\(benchmark\)](#)的结果

关于 `filtered` 和 `ignored` 的使用，在本章节的后续内容我们会讲到，这里暂且略过。

还有一个很重要的点，输出中的 `Doc-tests adder` 代表了文档测试，由于我们的代码中没有任何文档测试的内容，因此这里的测试用例数为 0，关于文档测试的详细介绍请参见[这里](#)。

大家还可以尝试修改下测试函数的名称，例如修改为 `exploration`，看看运行结果将如何变化。

## 失败的测试用例

是时候开始写自己的测试函数了，为了演示，这次我们来写一个会运行失败的：

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

新的测试函数 `another` 相当简单粗暴，直接使用 `panic` 来报错，使用 `cargo test` 运行看看结果：

```
running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

从结果看，两个测试函数，一个成功，一个失败，同时在输出中准确的告知了失败的函数名：`failures: tests::another`，同时还给出了具体的失败原因：`tests::another stdout`。这两者虽然看起来存在重复，但是前者用于说明每个失败的具体原因，后者用于给出一眼可得结论的汇总信息。

有同学可能会好奇，这两个测试函数以什么方式运行？它们会运行在同一个线程中吗？答案是否定的，Rust 在默认情况下会为每一个测试函数启动单独的线程去处理，当主线程 `main` 发现有一个测试线程死掉时，`main` 会将相应的测试标记为失败。

事实上，多线程运行测试虽然性能高，但是存在数据竞争的风险，在后文我们会对其进行详细介绍并给出解决方案。

## 自定义失败信息

默认的失败信息有时候并不是我们想要的，来看一个例子：

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Sunface");
        assert!(result.contains("孙飞"));
    }
}
```

使用 `cargo test` 运行后，错误如下：

```
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at 'assertion failed:
result.contains(\"孙飞\")', src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::greeting_contains_name
```

可以看出，这段报错除了告诉我们错误发生的地方，并没有更多的信息，那再来看看该如何提供一些更有用的信息：

```
fn greeting_contains_name() {
    let result = greeting("Sunface");
    let target = "孙飞";
    assert!(
        result.contains(target),
        "你的问候中并没有包含目标姓名 {}， 你的问候是 `{}`",
        target,
        result
    );
}
```

这段代码跟之前并无不同，只是为 `assert!` 新增了几个格式化参数，这种使用方式与 `format!` 并无区别。再次运行后，输出如下：

```
---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at '你的问候中并没有包含目标姓名 孙飞， 你
的问候是 `Hello Sunface!`', src/lib.rs:14:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

这次的报错就清晰太多了，真棒！在测试用例少的时候，也许这种信息还无法体现最大的价值，但是一旦测试多了后，详尽的报错信息将帮助我们更好的进行 Debug。

## 测试 panic

在之前的例子中，我们通过 `panic` 来触发报错，但是如果一个函数本来就会 `panic`，而我们想要检查这种结果呢？

也就是说，我们需要一个办法来测试一个函数是否会 `panic`，对此，Rust 提供了 `should_panic` 属性注解，和 `test` 注解一样，对目标测试函数进行标注即可：

```

pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}

```

上面是一个简单的猜数字游戏，`Guess` 结构体的 `new` 方法在传入的值不在 [1,100] 之间时，会直接 `panic`，而在测试函数 `greater_than_100` 中，我们传入的值 200 显然没有落入该区间，因此 `new` 方法会直接 `panic`，为了测试这个预期的 `panic` 行为，我们使用 `#[should_panic]` 对其进行了标注。

```

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

```

从输出可以看出，`panic` 的结果被准确的进行了测试，那如果测试函数中的代码不再 `panic` 呢？例如：

```

fn greater_than_100() {
    Guess::new(50);
}

```

此时显然会测试失败，因为我们预期一个 `panic`，但是 `new` 函数顺利的返回了一个 `Guess` 实例：

```
running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected // 测试并没有按照预期发生 panic
```

## expected

虽然 `panic` 被成功测试到，但是如果代码发生的 `panic` 和我们预期的 `panic` 不符合呢？因为一段糟糕的代码可能会在不同的代码行生成不同的 `panic`。

鉴于此，我们可以使用可选的参数 `expected` 来说明预期的 `panic` 长啥样：

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",,
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",,
                value
            );
        }
        Guess { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

这段代码会通过测试，因为通过增加了 `expected`，我们成功指定了期望的 `panic` 信息，大家可以顺着代码推测下：把 `200` 带入到 `new` 函数中看看会触发哪个 `panic`。

如果注意看，你会发现 `expected` 的字符串和实际 `panic` 的字符串可以不同，前者只需要是后者的字符串前缀即可，如果改成 `#[should_panic(expected = "Guess value must be less than")]`，一样可以通过测试。

这里由于篇幅有限，我们就不再展示测试失败的报错，大家可以自己修改下 `expected` 的信息，然后看看报错后的输出长啥样。

## 使用 `Result<T, E>`

在之前的例子中，`panic` 扫清一切障碍，但是它也不是万能的，例如你想在测试中使用 `?`  操作符进行链式调用该怎么办？那就得请出 `Result<T, E>` 了：

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

如上所示，测试函数不会再使用 `assert_eq!` 导致 `panic`，而是手动进行了逻辑判断，并返回一个 `Result`。当然，当这么实现时，`#[should_panic]` 将无法再被使用。

至此，关于如何写测试的基本知识，大家已经了解的差不多了，下面来看看该如何控制测试的执行。

## 使用 `-- 分割命令行参数`

大家应该都知道 `cargo build` 可以将代码编译成一个可执行文件，那你知道 `cargo run` 和 `cargo test` 是如何运行的吗？其实道理都一样，这两个也是将代码编译成可执行文件，然后进行运行，唯一的区别就在于这个可执行文件随后会被删除。

正因为如此，`cargo test` 也可以通过命令行参数来控制测试的执行，例如你可以通过参数来让默认的多线程测试变成单线程下的测试。需要注意的是命令行参数有两种，这两种通过 `--` 进行分割：

- 第一种是提供给 `cargo test` 命令本身的，这些参数在 `--` 之前指定
- 第二种是提供给编译后的可执行文件的，在 `--` 之后指定

例如我们可以使用 `cargo test --help` 来查看第一种参数的帮助列表，还可以通过 `cargo test -- --help` 来查看第二种的帮助列表。

先来看看第二种参数中的其中一个，它可以控制测试是并行运行还是顺序运行。

## 测试用例的并行或顺序执行

当运行多个测试函数时，默认情况下是为每个测试都生成一个线程，然后通过主线程来等待它们的完成和结果。这种模式的优点很明显，那就是并行运行会让整体测试时间变短很多，运行过大量测试用例的同学都明白并行测试的重要性：生命苦短，我用并行。

但是有利就有弊，并行测试最大的问题就在于共享状态的修改，因为你难以控制测试的运行顺序，因此如果多个测试共享一个数据，那么对该数据的使用也将变得不可控制。

例如，我们有多个测试，它们每个都会往该文件中写入一些**自己的数据**，最后再从文件中读取这些数据进行对比。由于所有测试都是同时运行的，当测试 A 写入数据准备读取并对比时，很有可能会被测试 B 写入新的数据，导致 A 写入的数据被覆盖，然后 A 再读取到的就是 B 写入的数据。结果 A 测试就会失败，而且这种失败还不是因为测试代码不正确导致的！

解决办法也有，我们可以让每个测试写入自己独立的文件中，当然，也可以让所有测试一个接着一个顺序运行：

```
$ cargo test -- --test-threads=1
```

首先能注意到的是该命令行参数是第二种类型：提供给编译后的可执行文件的，因为它在 `--` 之后进行传递。其次，细心的同学可能会想到，线程数不仅仅可以指定为 `1`，还可以指定为 `4`、`8`，当然，想要顺序运行，就必须是 `1`。

## 测试函数中的 `println!`

默认情况下，如果测试通过，那写入标准输出的内容是不会显示在测试结果中的：

```

fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}

```

上面代码使用 `println!` 输出收到的参数值，来看看测试结果：

```

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
  right: `10``, src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

```

大家注意看，`I got the value 4` 并没有被输出，因为该测试顺利通过了，如果就是想要看所有的输出，该怎么办呢？

```
$ cargo test -- --show-output
```

如上所示，只需要增加一个参数，具体的输出就不再展示，总之这次大家一定可以顺利看到 I got the value 4 的身影。

## 指定运行一部分测试

在 MySQL 中有上百万的单元测试，如果使用类似 `cargo test` 的命令来运行全部的测试，那开发真的工作十分钟，吹牛八小时了。对于 Rust 的中大型项目也一样，每次都运行全部测试是不可接受的，特别是你的工作仅仅是项目中的一部分时。

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

如果直接使用 `cargo test` 运行，那三个测试函数会同时并行的运行：

```
running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

就不说上百万测试，就说几百个，想象一下结果会是怎么样，下面我们来看看该如何解决这个问题。

## 运行单个测试

这个很简单，只需要将指定的测试函数名作为参数即可：

```
$ cargo test one_hundred
running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished
in 0.00s
```

此时，只有测试函数 `one_hundred` 会被运行，其它两个由于名称不匹配，会被直接忽略。同时，在上面的输出中，Rust 也通过 `2 filtered out` 提示我们：有两个测试函数被过滤了。

但是，如果你试图同时指定多个名称，那抱歉：

```
$ cargo test one_hundred,add_two_and_two
$ cargo test one_hundred add_two_and_two
```

这两种方式统统不行，此时就需要使用名称过滤的方式来实现了。

## 通过名称来过滤测试

我们可以通过指定部分名称的方式来过滤运行相应的测试：

```
$ cargo test add
running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished
in 0.00s
```

事实上，你不仅可以使用前缀，还能使用名称中间的一部分：

```
$ cargo test and
running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished
in 0.00s
```

其中还有一点值得注意，那就是测试模块 `tests` 的名称也出现在了最终结果中：

`tests::add_two_and_two`，这是非常贴心的细节，也意味着我们可以通过**模块名称来过滤测试**：

```
$ cargo test tests

running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

## 忽略部分测试

有时候，一些测试会非常耗时间，因此我们希望在 `cargo test` 中对它进行忽略，如果使用之前的方式，我们需要将所有需要运行的名称指定一遍，这非常麻烦，好在 Rust 允许通过 `ignore` 关键字来忽略特定的测试用例：

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // 这里的代码需要几十秒甚至几分钟才能完成
}
```

在这里，我们使用 `#[ignore]` 对 `expensive_test` 函数进行了标注，看看结果：

```
$ cargo test
running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

输出中的 `test expensive_test ... ignored` 意味着该测试函数被忽略了，因此并没有被执行。

当然，也可以通过以下方式运行被忽略的测试函数：

```
$ cargo test -- --ignored
running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished
in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

## 组合过滤

上面的方式虽然很强大，但是单独使用依然存在局限性。好在它们还能组合使用，例如还是之前的代码：

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    #[ignore]
    fn expensive_test() {
        // 这里的代码需要几十秒甚至几分钟才能完成
    }

    #[test]
    #[ignore]
    fn expensive_run() {
        // 这里的代码需要几十秒甚至几分钟才能完成
    }
}
```

然后运行 `tests` 模块中的被忽略的测试函数

```
$ cargo test tests -- --ignored
running 2 tests
test tests::expensive_test ... ok
test tests::expensive_run ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished
in 0.00s
```

运行名称中带 `run` 且被忽略的测试函数：

```
$ cargo test run -- --ignored
running 1 test
test tests::expensive_run ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished
in 0.00s
```

类似的还有很多，大家可以自己摸索研究下，总之，熟练掌握测试的使用是非常重要的，虽然包括我在内的很多开发并不喜欢写测试：)

## [dev-dependencies]

与 `package.json` (Nodejs)文件中的 `devDependencies` 一样，Rust 也能引入只在开发测试场景使用的外部依赖。

其中一个例子就是 `pretty_assertions`，它可以用来扩展标准库中的 `assert_eq!` 和 `assert_ne!`，例如提供彩色字体的结果对比。

在 `Cargo.toml` 文件中添加以下内容来引入 `pretty_assertions`：

```
# standard crate data is left out
[dev-dependencies]
pretty_assertions = "1"
```

然后在 `src/lib.rs` 中添加：

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    use pretty_assertions::assert_eq; // 该包仅能用于测试

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }
}
```

在 `tests` 模块中，我们通过 `use pretty_assertions::assert_eq;` 成功的引入之前添加的包，由于 `tests` 模块明确的用于测试目的，这种引入并不会报错。大家可以试试在正常代码(非测试代码)中引入该包，看看会发生什么。

## 生成测试二进制文件

在有些时候，我们可能希望将测试与别人分享，这种情况下生成一个类似 `cargo build` 的可执行二进制文件是很好的选择。

事实上，在 `cargo test` 运行的时候，系统会自动为我们生成一个可运行测试的二进制可执行文件：

```
$ cargo test
Finished test [unoptimized + debuginfo] target(s) in 0.00s
    Running unit tests (target/debug/deps/study_cargo-0d693f72a0f49166)
```

这里的 `target/debug/deps/study_cargo-0d693f72a0f49166` 就是可执行文件的路径和名称，我们直接运行该文件来执行编译好的测试：

```
$ target/debug/deps/study_cargo-0d693f72a0f49166

running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

如果你只想生成编译生成文件，不想看 `cargo test` 的输出结果，还可以使用 `cargo test --no-run`。

# 单元测试、集成测试

在了解了如何在 Rust 中写测试用例后，本章节我们将学习如何实现单元测试、集成测试，其实它们用到的技术还是[上一章节](#)中的测试技术，只不过对如何组织测试代码提出了新的要求。

## 单元测试

单元测试目标是测试某一个代码单元(一般都是函数)，验证该单元是否能按照预期进行工作，例如测试一个 `add` 函数，验证当给予两个输入时，最终返回的和是否符合预期。

在 Rust 中，单元测试的惯例是将测试代码的模块跟待测试的正常代码放入同一个文件中，例如 `src/lib.rs` 文件中有如下代码：

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(add_two(2), 4);
    }
}
```

`add_two` 是我们的项目代码，为了对它进行测试，我们在同一个文件中编写了测试模块 `tests`，并使用 `#[cfg(test)]` 进行了标注。

### 条件编译 `#[cfg(test)]`

上面代码中的 `#[cfg(test)]` 标注可以告诉 Rust 只有在 `cargo test` 时才编译和运行模块 `tests`，其它时候当这段代码是空即可，例如在 `cargo build` 时。这么做有几个好处：

- 节省构建代码时的编译时间
- 减小编译出的可执行文件的体积

其实集成测试就不需要这个标注，因为它们被放入单独的目录文件中，而单元测试是跟正常的逻辑代码在同一个文件，因此必须对其进行特殊的标注，以便 Rust 可以识别。

在 `#[cfg(test)]` 中，`cfg` 是配置 `configuration` 的缩写，它告诉 Rust：当 `test` 配置项存在时，才运行下面的代码，而 `cargo test` 在运行时，就会将 `test` 这个配置项传入进来，因此后面的 `tests` 模块会被包含进来。

大家看出来了吗？这是典型的条件编译，Cargo 会根据指定的配置来选择是否编译指定的代码，事实上关于条件编译 Rust 能做的不仅仅是这些，在 [Cargo 专题](#) 中我们会进行更为详细的介绍。

## 测试私有函数

关于私有函数能否被直接测试，编程社区里一直争论不休，甚至于部分语言可能都不支持对私有函数进行测试或者难以测试。无论你的立场如何，反正 Rust 是支持对私有函数进行测试的：

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

`internal_adder` 并没有使用 `pub` 进行声明，因此它是一个私有函数。根据我们之前学过的内容，`tests` 作为另一个模块，是绝对无法对它进行调用的，因为它们根本不在同一个模块中！

但是在上述代码中，我们使用 `use super::*;` 将 `tests` 的父模块中的所有内容引入到当前作用域中，这样就可以非常简单的实现对私有函数的测试。

## 集成测试

与单元测试的同吃同住不同，集成测试的代码是在一个单独的目录下的。由于它们使用跟其它模块一样的方式去调用你想要测试的代码，因此只能调用通过 `pub` 定义的 API，这一点与单元测试有很大的不同。

如果说单元测试是对代码单元进行测试，那集成测试则是对某一个功能或者接口进行测试，因此单元测试的通过，并不意味着集成测试就能通过：局部上反映不出的问题，在全局上很可能会暴露出来。

## tests 目录

一个标准的 Rust 项目，在它的根目录下会有一个 `tests` 目录，大名鼎鼎的 `ripgrep` 也不能免俗。

没错，该目录就是用来存放集成测试的，Cargo 会自动来此目录下寻找集成测试文件。我们可以在该目录下创建任何文件，Cargo 会对每个文件都进行自动编译，但友情提示下，最好按照合适的逻辑来组织你的测试代码。

首先来创建一个集成测试文件 `tests/integration_test.rs`，注意，`tests` 目录一般来说需要手动创建，该目录在项目的根目录下，跟 `src` 目录同级。然后在文件中填入如下测试代码：

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

这段测试代码是对之前**私有函数**中的示例进行测试，该示例代码在 `src/lib.rs` 中。

首先与单元测试有所不同，我们并没有创建测试模块。其次，`tests` 目录下的每个文件都是一个单独的包，我们需要将待测试的包引入到当前包的作用域后：`use adder`，才能进行测试。大家应该还记得[包和模块章节](#)中讲过的内容吧？在创建项目后，`src/lib.rs` 自动创建一个与项目同名的 `lib` 类型的包，由于我们的项目名是 `adder`，因此包名也是 `adder`。

因为 `tests` 目录本身就说明了它的特殊用途，因此我们无需再使用 `#[cfg(test)]` 来取悦 Cargo。后者会在运行 `cargo test` 时，对 `tests` 目录中的每个文件都进行编译运行。

```
$ cargo test
    Running unitests (target/debug/deps/adder-8a400aa2b5212836)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

    Running tests/integration_test.rs (target/debug/deps/integration_test-
2d3aeee6f15d1f20)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

运行 `cargo test`，可以看到上述输出。测试内容有三个部分：单元测试，集成测试和文档测试。

首先是单元测试被运行 `Running unitests`，其次就是我们的主角集成测试的运行 `Running tests/integration_test.rs`，可以看出，集成测试的输出内容与单元测试并没有大的区别。最后运行的是文档测试 `Doc-tests adder`。

与单元测试类似，我们可以通过[指定名称的方式来运行特定的集成测试用例](#):

```
$ cargo test --test integration_test
    Running tests/integration_test.rs (target/debug/deps/integration_test-
82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

这次，单元测试、文档测试啥的都没有运行，只有集成测试目录下的 `integration_test` 文件被顺利执行。

大家可以尝试下在同一个测试文件中添加更多的测试用例或者添加更多的测试文件，并观察测试输出会如何变化。

## 共享模块

在集成测试的 `tests` 目录下，每一个文件都是一个独立的包，这种组织方式可以很好的帮助我们理清测试代码的关系，但是如果大家想要在多个文件中共享同一个功能该怎么做？例如函数 `setup` 可以用于状态初始化，然后多个测试包都需要使用该函数进行状态的初始化。

也许你会想要创建一个 `tests/common.rs` 文件，然后将 `setup` 函数放入其中：

```
pub fn setup() {  
    // 初始化一些测试状态  
    // ...  
}
```

但是当我们运行 `cargo test` 后，会发现该函数被当作集成测试函数运行了，即使它并没有包含任何测试功能，也没有被其它测试文件所调用：

```
$ cargo test  
    Running tests/common.rs (target/debug/deps/common-5c21f4f2c87696fb)  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished  
in 0.00s
```

显然，这个结果并不是我们想要的。为了避免这种输出，我们不能创建 `tests/common.rs`，而是要创建 `tests/common/mod.rs`，此时再运行 `cargo test` 就不会再看到相应的输出。原因是通过这种文件组织和命名方式，Rust 不再将 `common` 模块看作是集成测试文件。

总结来说，`tests` 目录下的子目录中的文件不会被当作独立的包，也不会有测试输出。

```
use adder;  
  
mod common;  
  
#[test]  
fn it_adds_two() {  
    common::setup();  
    assert_eq!(4, adder::add_two(2));  
}
```

此时，就可以在测试中调用 `common` 中的共享函数了，不过还有一点值得注意，为了使用 `common`，这里使用了 `mod common` 的方式来声明该模块。

## 二进制包的集成测试

目前来说，Rust 只支持对 `lib` 类型的包进行集成测试，对于二进制包例如 `src/main.rs` 是无能为力的。原因在于，我们无法在其它包中使用 `use` 引入二进制包，而只有 `lib` 类型的包才能被引入，例如 `src/lib.rs`。

这就是为何我们需要将代码逻辑从 `src/main.rs` 剥离出去放入 `lib` 包中，例如很多 Rust 项目中都同时有 `src/main.rs` 和 `src/lib.rs`，前者中只保留代码的主体脉络部分，而具体的实现通通放在类似后者的 `lib` 包中。

这样，我们就可以对 `lib` 包中的具体实现进行集成测试，由于 `main.rs` 中的主体脉络足够简单，当集成测试通过时，意味着 `main.rs` 中相应的调用代码也将正常运行。

## 总结

Rust 提供了单元测试和集成测试两种方式来帮助我们组织测试代码以解决代码正确性问题。

单元测试针对的是具体的代码单元，例如函数，而集成测试往往针对的是一个功能或接口 API，正因为目标上的不同，导致了两者在组织方式上的不同：

- 单元测试的模块和待测试的代码在同一个文件中，且可以很方便地对私有函数进行测试
- 集成测试文件放在项目根目录下的 `tests` 目录中，由于该目录下每个文件都是一个包，我们必须引入待测试的代码到当前包的作用域中，才能进行测试，正因为此，集成测试只能对声明为 `pub` 的 API 进行测试

下个章节，我们再来看看该如何使用 GitHub Actions 对 Rust 项目进行持续集成。

# 断言 assertion

在编写测试函数时，断言决定了我们的测试是通过还是失败，它为结果代言。在前面，大家已经见识过 `assert_eq!` 的使用，下面一起来看看 Rust 为我们提供了哪些好用的断言。

## 断言列表

在正式开始前，来看看常用的断言有哪些：

- `assert!`, `assert_eq!`, `assert_ne!`, 它们会在所有模式下运行
- `debug_assert!`, `debug_assert_eq!`, `debug_assert_ne!`, 它们只会在 Debug 模式下运行

## assert\_eq!

`assert_eq!` 宏可以用于判断两个表达式返回的值是否相等：

```
fn main() {
    let a = 3;
    let b = 1 + 2;
    assert_eq!(a, b);
}
```

当不相等时，当前线程会直接 `panic`：

```
fn main() {
    let a = 3;
    let b = 1 + 3;
    assert_eq!(a, b, "我们在测试两个数之和{} + {}, 这是额外的错误信息", a, b);
}
```

运行后报错如下：

```
$ cargo run
thread 'main' panicked at 'assertion failed: `left == right)`
  left: `3`,
  right: `4`: 我们在测试两个数之和3 + 4, 这是额外的错误信息', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

可以看到，错误不仅按照预期发生了，我们还成功的定制了错误信息！这种格式化输出的方式跟 `println!` 并无区别，具体参见 `std::fmt`。

因为涉及到相等比较(`==`)和错误信息打印，因此两个表达式的值必须实现 `PartialEq` 和 `Debug` 特征，其中所有的原生类型和大多数标准库类型都实现了这些特征，而对于你自己定义的结构体、枚举，如果想要对其进行 `assert_eq!` 断言，则需要实现 `PartialEq` 和 `Debug` 特征：

- 若希望实现个性化相等比较和错误打印，则需手动实现
- 否则可以为自定义的结构体、枚举添加 `#[derive(PartialEq, Debug)]` 注解，来自[自动派生](#)对应的特征

以上特征限制对于下面即将讲解的 `assert_ne!` 一样有效，就不再重复讲述。

## assert\_ne!

`assert_ne!` 在使用和限制上与 `assert_eq!` 并无区别，唯一的区别就在于，前者判断的是两者的不相等性。

我们将之前报错的代码稍作修改：

```
fn main() {
    let a = 3;
    let b = 1 + 3;
    assert_ne!(a, b, "我们在测试两个数之和{} + {}, 这是额外的错误信息", a, b);
}
```

由于 `a` 和 `b` 不相等，因此 `assert_ne!` 会顺利通过，不再报错。

## assert!

`assert!` 用于判断传入的布尔表达式是否为 `true`：

```

// 以下断言的错误信息只包含给定表达式的返回值
assert!(true);

fn some_computation() -> bool { true }

assert!(some_computation());

// 使用自定义报错信息
let x = true;
assert!(x, "x wasn't true!");

// 使用格式化的自定义报错信息
let a = 3; let b = 27;
assert!(a + b == 30, "a = {}, b = {}", a, b);

```

来看看该如何使用 `assert!` 进行单元测试：

```

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

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}

```

## debug\_assert! 系列

debug\_assert! , debug\_assert\_eq! , debug\_assert\_ne! 这三个在功能上与之前讲解的版本并无区别，主要区别在于， debug\_assert! 系列只能在 Debug 模式下输出，例如如下代码：

```
fn main() {
    let a = 3;
    let b = 1 + 3;
    debug_assert_eq!(a, b, "我们在测试两个数之和{} + {}, 这是额外的错误信息", a, b);
}
```

在 Debug 模式下运行输出错误信息：

```
$ cargo run
thread 'main' panicked at 'assertion failed: `left == right)`
  left: `3`,
  right: `4`: 我们在测试两个数之和3 + 4, 这是额外的错误信息', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

但是在 Release 模式下却没有任何输出：

```
$ cargo run --release
```

若一些断言检查会影响发布版本的性能时，大家可以使用 debug\_assert! 来避免这种情况的发生。

# 用 GitHub Actions 进行持续集成

[GitHub Actions](#) 是官方于 2018 年推出的持续集成服务，它非常强大，本文将手把手带领大家学习如何使用 GitHub Actions 对 Rust 项目进行持续集成。

持续集成是软件开发中异常重要的一环，大家应该都听说过 Jenkins，它就是一个拥有悠久历史的持续集成工具。简单来说，持续集成会定期拉取同一个项目中所有成员的相关代码，对其进行自动化构建。

在没有持续集成前，首先开发者需要手动编译代码并运行单元测试、集成测试等基础测试，然后启动项目相关的所有服务，接着测试人员开始介入对整个项目进行回归测试、黑盒测试等系统化的测试，当测试通过后，最后再手动发布到指定的环境中运行，这个过程是非常冗长，且所有成员都需要同时参与的。

在有了持续集成后，只要编写好相应的编译、测试、发布配置文件，那持续集成平台会自动帮助我们完成整个相关的流程，期间无需任何人介入，高效且可靠。

## GitHub Actions

而本文的主角正是这样的持续集成平台，它由 GitHub 官方提供，并且跟 GitHub 进行了深度的整合，其中 actions 代表了代码拉取、测试运行、登陆远程服务器、发布到第三方服务等操作行为。

最妙的是 GitHub 发现这些 actions 其实在很多项目中都是类似的，意味着 actions 完全可以被多个项目共享使用，而不是每个项目都从零开发自己的 actions。

若你需要某个 action，不必自己写复杂的脚本，直接引用他人写好的 action 即可，整个持续集成过程，就变成了多个 action 的组合，这就是 GitHub Actions 最厉害的地方。

### action 的分享与引用

既然 action 这么强大，我们就可以将自己的 action 分享给他人，也可以引用他人分享的 action，有以下几种方式：

1. 将你的 action 放在 GitHub 上的公共仓库中，这样其它开发者就可以引用，例如 [github-profile-summary-cards](#) 就提供了相应的 action，可以生成 GitHub 用户统计信息，然后嵌入到你的个人主页中，具体效果见[这里](#)
2. GitHub 提供了一个[官方市场](#)，里面收集了许多质量不错的 actions，并支持在线搜索
3. [awesome-actions](#)，由三方开发者收集并整理的 actions
4. [starter workflows](#)，由官方提供的工作流(workflow)模版

对于第一点这里再补充下，如果你想要引用某个代码仓库中的 action，可以通过 `userName/repoName` 方式来引用：例如你可以通过 `actions/setup-node` 来引用 `github.com/actions/setup-node` 仓库中的 action，该 action 的作用是安装 Node.js。

由于 action 是代码仓库，因此就有版本的概念，你可以使用 @ 符号来引入同一个仓库中不同版本的 action，例如：

```
actions/setup-node@master # 指向一个分支  
actions/setup-node@v2.5.1   # 指向一个 release  
actions/setup-node@f099707 # 指向一个 commit
```

如果希望深入了解，可以进一步查看官方的[文档](#)。

## Actions 基础

在了解了何为 GitHub Actions 后，再来通过一个基本的例子来学习下它的基本概念，注意，由于篇幅有限，我们只会讲解最常用的部分，如果想要完整的学习，请移步[这里](#)。

### 创建 action demo

首先，为了演示，我们需要创建一个公开的 GitHub 仓库 `rust-action`，然后在仓库主页的导航栏中点击 `Actions`，你会看到如下页面：

The screenshot shows the GitHub Actions page for the repository `sunface/rust-action`. The repository is public. The top navigation bar includes links for Code, Issues, Pull requests, Actions (which is the active tab), Projects, Wiki, Security, Insights, and Settings. There are also buttons for Pin, Unwatch (with 1 watch), and Fork.

## Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and [set up a workflow yourself →](#)

Search workflows

### Suggested for this repository

#### Simple workflow

By GitHub



Start with a file with the minimum necessary structure.

[Configure](#)

### Deployment

---

接着点击 `set up a workflow yourself →`，你将看到系统为你自动创建的一个工作流 workflow，在 `rust-action/.github/workflows/main.yml` 文件中包含以下内容：

```
# 下面是一个基础的工作流，你可以基于它来编写自己的 GitHub Actions
name: CI

# 控制工作流何时运行
on:
  # 当 `push` 或 `pull request` 事件发生时就触发工作流的执行，这里仅仅针对 `main` 分支
  push:
    branches: [main]
  pull_request:
    branches: [main]

# 允许用于在 `Actions` 标签页中手动运行工作流
workflow_dispatch:

# 工作流由一个或多个作业( job )组成，这些作业可以顺序运行也可以并行运行
jobs:
  # 当前的工作流仅包含一个作业，作业 id 是 "build"
  build:
    # 作业名称
    name: build rust action
    # 执行作业所需的运行器 runner
    runs-on: ubuntu-latest

    # steps 代表了作业中包含的一系列可被执行的任务
    steps:
      # 在 $GITHUB_WORKSPACE 下 checks-out 当前仓库，这样当前作业就可以访问该仓库
      - uses: actions/checkout@v2

      # 使用运行器的终端来运行一个命令
      - name: Run a one-line script
        run: echo Hello, world!

      # 使用运行器的终端运行一组命令
      - name: Run a multi-line script
        run: |
          echo Add other actions to build,
          echo test, and deploy your project.
```

## 查看工作流信息

通过内容的注释，大家应该能大概理解这个工作流是怎么回事了，在具体讲解前，我们先完成 Actions 的创建，点击右上角的 Start Commit 绿色按钮提交，然后再回到 Actions 标签页，你可以看到如下界面：

The screenshot shows the GitHub Actions 'All workflows' page for the repository 'surface/rust-action'. A single workflow run for 'Create main.yml' is listed. The run was triggered by a commit ('CI #1: Commit 9c929f3 pushed by surface') and is currently in the 'Queued' state, with the status shown as 'now'. The run is associated with the 'main' branch.

这里包含了我们刚创建的工作流及当前的状态，从右下角可以看出，该工作流的运行时间是 now 也就是现在， queued 代表它已经被安排到了运行队列中，等待运行。

等过几秒(也可能几十秒后)，刷新当前页面，就能看到成功运行的工作流：

The screenshot shows the GitHub Actions 'All workflows' page for the repository 'surface/rust-action'. The same workflow run for 'Create main.yml' is now listed as successful ('7 minutes ago'). The status is shown as '17s'.

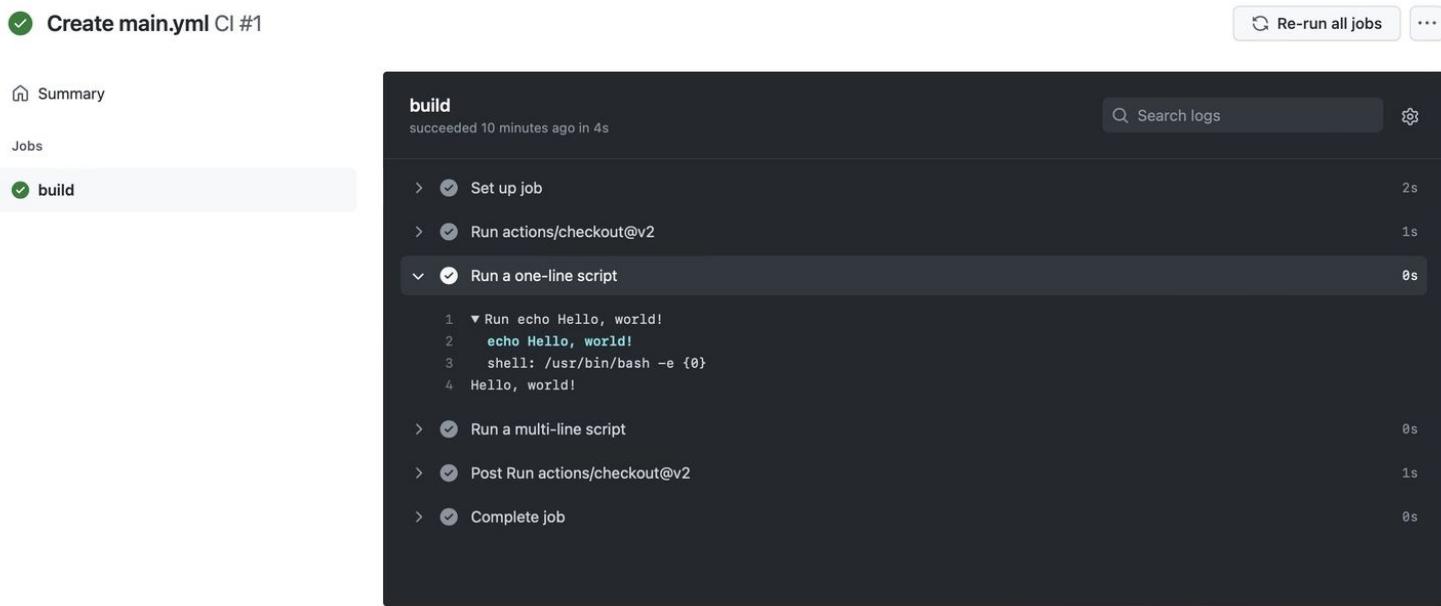
还记得之前配置中的 `workflow_dispatch` 嘛？它允许工作流被手动执行：点击左边的 All workflows -> CI，可以看到如下页面。

The screenshot shows the GitHub Actions 'CI' page for the 'main.yml' workflow. It indicates that the workflow has a `workflow_dispatch` event trigger. A 'Run workflow' button is visible next to the workflow run details.

页面中通过蓝色的醒目高亮提示我们 this workflow has a `workflow_dispatch` event trigger，因此可以点击右边的 Run workflow 手动再次执行该工作流。

注意，目前 Actions 并不会自动渲染最新的结果，因此需要刷新页面来看到最新的结果

点击 Create main.yml 可以查看该工作流的详细信息：



至此，我们已经初步掌握 GitHub Actions 的用法，现在来看看一些基本的概念。

## 基本概念

- **GitHub Actions**，每个项目都拥有一个 Actions，可以包含多个工作流
- **workflow 工作流**，描述了一次持续集成的过程
- **job 作业**，一个工作流可以包含多个作业，因为一次持续集成本身就由多个不同的部分组成
- **step 步骤**，每个作业由多个步骤组成，按照顺序一步一步完成
- **action 动作**，每个步骤可以包含多个动作，例如上例中的 Run a multi-line script 步骤就包含了两个动作

可以看出，每一个概念都是相互包含的关系，前者包含了后者，层层相扣，正因为这些精心设计的对象才有了强大的 GitHub Actions。

## on

on 可以设定事件用于触发工作流的运行：

1. 一个或多个 GitHub 事件，例如 push 一个 commit、创建一个 issue、提交一次 pr 等等，详细的事件列表参见[这里](#)
2. 预定的时间，例如每天零点零分触发，详情见[这里](#)

```
on:  
  schedule: -cron: '0 0 * * *'
```

3. 外部事件触发，例如你可以通过 REST API 向 GitHub 发送请求去触发，具体请查阅[官方文档](#)

## jobs

工作流由一个或多个作业 `job` 组成，这些作业可以顺序运行也可以并行运行，同时我们还能使用 `needs` 来指定作业之间的依赖关系：

```
jobs:  
  job1:  
  
  job2:  
    needs: job1  
  
  job3:  
    needs: [job1, job2]
```

这里的 `job2` 必须等待 `job1` 成功后才能运行，而 `job3` 则需要等待 `job1` 和 `job2`。

## runs-on

指定作业的运行环境，运行器 `runner` 分为两种：`GitHub-hosted runner` 和 `self-hosted runner`，后者是使用自己的机器来运行作业，但是需要 GitHub 能进行访问并给予相应的机器权限，感兴趣的同学可以看看[这里](#)。

而对于前者，GitHub 提供了以下的运行环境：

Virtual environment	YAML workflow label	Notes
Windows Server 2022	windows-2022	The windows-latest label currently uses the Windows Server 2019 runner image.
Windows Server 2019	windows-latest or windows-2019	
Windows Server 2016 <sup>[deprecated]</sup>	windows-2016	Migrate to Windows 2019 or Windows 2022. For more information, see <a href="#">the blog post</a> .
Ubuntu 20.04	ubuntu-latest or ubuntu-20.04	
Ubuntu 18.04	ubuntu-18.04	
macOS Big Sur 11	macos-latest or macos-11	The macos-latest label currently uses the macOS 11 runner image.
macOS Catalina 10.15	macos-10.15	

其中比较常用的就是 runs-on:ubuntu-latest。

## strategy.matrix

有时候我们常常需要对多个操作系统、多个平台、多个编程语言版本进行测试，为此我们可以配置一个 matrix 矩阵：

```
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-18.04, ubuntu-20.04]
    node: [10, 12, 14]
steps:
  - uses: actions/setup-node@v2
    with:
      node-version: ${{ matrix.node }}
```

大家猜猜，这段代码会最终构建多少个作业？答案是  $2 * 3 = 6$ ，通过 `os` 和 `node` 进行组合，就可以得出这个结论，这也是 `matrix` 矩阵名称的来源。

当然，`matrix` 能做的远不止这些，例如，你还可以定义自己想要的 `kv` 键值对，想要深入学习的话可以看看[官方文档](#)。

## strategy

除了 `matrix`，`strategy` 中还能设定以下内容：

- `fail-fast`：默认为 `true`，即一旦某个矩阵任务失败则立即取消所有还在进行中的任务
- `max-parallel`：可同时执行的最大并发数，默认情况下 GitHub 会动态调整

## env

用于设定环境变量，可以用于以下地方：

- `env`
- `jobs.<job_id>.env`
- `jobs.<job_id>.steps.env`

```
env:  
  NODE_ENV: dev  
  
jobs:  
  job1:  
    env:  
      NODE_ENV: test  
  
    steps:  
      - name:  
        env:  
          NODE_ENV: prod
```

如果有多个 `env` 存在，会使用就近那个。

至此，GitHub Actions 的常用内容大家已经基本了解，下面来看一个实用的示例。

# 真实示例：生成 GitHub 统计卡片

相信大家看过不少用户都定制了自己的个性化 GitHub 首页，这个是通过在个人名下创建一个同名的仓库来实现的，该仓库中的 `Readme.md` 的内容会自动展示在你的个人首页中，例如 Sunface 的[个人首页](#)和[内容所在的仓库](#)。

大家可能会好奇上面链接中的 GitHub 统计卡片如何生成，其实有两种办法：

- 使用 [github-readme-stats](#)
- 使用 GitHub Actions 来引用其它人提供的 action 生成对应的卡片，再嵌入进来，Sunface 的个人首页就是这么做的

第一种的优点就是非常简单，缺点是样式不太容易统一，不能对齐对于强迫症来说实在难受 :( 而后者的优点是规规矩矩的卡片，缺点就是使用起来更加复杂，而我们正好借此来看看真实的 GitHub Actions 长什么样。

首先，在你的同名项目下创建 `.github/workflows/profile-summary-cards.yml` 文件，然后填入以下内容：

```
# 工作流名称
name: GitHub-Profile-Summary-Cards

on:
  schedule:
    # 每24小时触发一次
    - cron: "0 * * * *"
  # 开启手动触发
  workflow_dispatch:

jobs:
  # job id
  build:
    runs-on: ubuntu-latest
    name: generate

    steps:
      # 第一步，checkout 当前项目
      - uses: actions/checkout@v2
      # 第二步，引入目标 action: vn7n24fzkq/github-profile-summary-cards仓库中的
      `release` 分支
      - uses: vn7n24fzkq/github-profile-summary-cards@release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          USERNAME: ${{ github.repository_owner }}
```

当提交后，该工作流会自动在当前项目中生成 `profile-summary-card-output` 目录，然后将所有卡片放入其中，当然我们这里使用了定时触发的机制，并没有基于 `pr` 或 `push` 来触发，如果你在编写过程中，希望手动触发来看看结果，请参考前文的手动触发方式。

这里我们引用了 `vn7n24fzkq/github-profile-summary-cards@release` 的 `action`，位于 <https://github.com/vn7n24fzkq/github-profile-summary-cards> 仓库中，并指定使用 `release` 分支。

接下来就可以愉快的[使用这些卡片](#)来定制我们的主页了：）

## 使用 Actions 来构建 Rust 项目

其实 Rust 项目也没有什么特别之处，我们只需要在 `steps` 逐步构建即可，下面给出该如何测试和构建的示例。

## 测试

```
on: [push, pull_request]

name: Continuous integration

jobs:
  check:
    name: Check
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions-rs/toolchain@v1
        with:
          profile: minimal
          toolchain: stable
          override: true
      - run: cargo check

    test:
      name: Test Suite
      runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v2
        - uses: actions-rs/toolchain@v1
          with:
            profile: minimal
            toolchain: stable
            override: true
        - run: cargo test

    fmt:
      name: Rustfmt
      runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v2
        - uses: actions-rs/toolchain@v1
          with:
            profile: minimal
            toolchain: stable
            override: true
        - run: rustup component add rustfmt
        - run: cargo fmt --all -- --check

  clippy:
    name: Clippy
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions-rs/toolchain@v1
        with:
          profile: minimal
```

```
  toolchain: stable
    override: true
- run: rustup component add clippy
- run: cargo clippy -- -D warnings
```

# 构建

```
name: build
on:
  workflow_dispatch: {}
jobs:
  build:
    name: build
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        build: [linux, macos, windows]
        include:
          - build: linux
            os: ubuntu-18.04
            rust: nightly
            target: x86_64-unknown-linux-musl
            archive-name: sgf-render-linux.tar.gz
          - build: macos
            os: macos-latest
            rust: nightly
            target: x86_64-apple-darwin
            archive-name: sgf-render-macos.tar.gz
          - build: windows
            os: windows-2019
            rust: nightly-x86_64-msvc
            target: x86_64-pc-windows-msvc
            archive-name: sgf-render-windows.7z
    fail-fast: false

steps:
  - name: Checkout repository
    uses: actions/checkout@v2

  - name: Install Rust
    uses: actions-rs/toolchain@v1
    with:
      toolchain: ${{ matrix.rust }}
      profile: minimal
      override: true
      target: ${{ matrix.target }}

  - name: Build binary
    run: cargo build --verbose --release --target ${{ matrix.target }}
    env:
      RUST_BACKTRACE: 1

  - name: Strip binary (linux and macos)
    if: matrix.build == 'linux' || matrix.build == 'macos'
    run: strip "target/${{ matrix.target }}/release/sgf-render"
```

```
- name: Build archive
  shell: bash
  run: |
    mkdir archive
    cp LICENSE README.md archive/
    cd archive
    if [ "${{ matrix.build }}" = "windows" ]; then
      cp "../target/{{ matrix.target }}/release/sgf-render.exe" ./
      7z a "{{ matrix.archive-name }}" LICENSE README.md sgf-render.exe
    else
      cp "../target/{{ matrix.target }}/release/sgf-render" ./
      tar -czf "{{ matrix.archive-name }}" LICENSE README.md sgf-render
    fi
- name: Upload archive
  uses: actions/upload-artifact@v1
  with:
    name: "{{ matrix.archive-name }}"
    path: archive/{{ matrix.archive-name }}
```

限于文章篇幅有限，我们就不再多做解释，大家有疑问可以看看文中给出的文档链接，顺便说一句官方文档是支持中文的！

# 基准测试 benchmark

几乎所有开发都知道，如果要测量程序的性能，就需要性能测试。

性能测试包含了两种：压力测试和基准测试。前者是针对接口 API，模拟大量用户去访问接口然后生成接口级别的性能数据；而后者是针对代码，可以用来测试某一段代码的运行速度，例如一个排序算法。

而本文将要介绍的就是基准测试 `benchmark`，在 Rust 中，有两种方式可以实现：

- 官方提供的 `benchmark`
- 社区实现，例如 `criterion.rs`

事实上我们更推荐后者，原因在后文会详细介绍，下面先从官方提供的工具开始。

## 官方 `benchmark`

官方提供的测试工具，目前最大的问题就是只能在非 `stable` 下使用，原因是需要在代码中引入 `test` 特性: `#![feature(test)]`。

### 设置 Rust 版本

因此在开始之前，我们需要先将当前仓库中的 Rust 版本从 `stable` 切换为 `nightly`：

1. 安装 `nightly` 版本: `$ rustup install nightly`
2. 使用以下命令确认版本已经安装成功

```
$ rustup toolchain list
stable-aarch64-apple-darwin (default)
nightly-aarch64-apple-darwin (override)
```

3. 进入 `adder` 项目(之前为了学习测试专门创建的项目)的根目录，然后运行 `rustup override set nightly`，将该项目使用的 `rust` 设置为 `nightly`

很简单吧，其实只要一个命令就可以切换指定项目的 Rust 版本，例如你还能在基准测试后再使用 `rustup override set stable` 切换回 `stable` 版本。

## 使用 benchmark

当完成版本切换后，就可以开始正式编写 benchmark 代码了。首先，将 `src/lib.rs` 中的内容替换成如下代码：

```
#![feature(test)]  
  
extern crate test;  
  
pub fn add_two(a: i32) -> i32 {  
    a + 2  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;

    #[test]  
    fn it_works() {  
        assert_eq!(4, add_two(2));  
    }
}

#[bench]
fn bench_add_two(b: &mut Bencher) {  
    b.iter(|| add_two(2));  
}
```

可以看出，benchmark 跟单元测试区别不大，最大的区别在于它是通过 `#[bench]` 标注，而单元测试是通过 `#[test]` 进行标注，这意味着 `cargo test` 将不会运行 benchmark 代码：

```
$ cargo test  
running 2 tests  
test tests::bench_add_two ... ok  
test tests::it_works ... ok  
  
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished  
in 0.00s
```

`cargo test` 直接把我们的 benchmark 代码当作单元测试处理了，因此没有任何性能测试的结果产生。

对此，需要使用 `cargo bench` 命令：

```
$ cargo bench
running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:           0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured; 0 filtered out; finished
in 0.29s
```

看到没，一个截然不同的结果，除此之外还能看出几点：

- 单元测试 `it_works` 被忽略，并没有执行：`tests::it_works ... ignored`
- benchmark 的结果是 `0 ns/iter`，表示每次迭代(`b.iter`)耗时 `0 ns`，奇怪，怎么是 `0 纳秒` 呢？别急，原因后面会讲

## 一些使用建议

关于 `benchmark`，这里有一些使用建议值得大家关注：

- 将初始化代码移动到 `b.iter` 循环之外，否则每次循环迭代都会初始化一次，这里只应该存放需要精准测试的代码
- 让代码每次都做一样的事情，例如不要去做累加或状态更改的操作
- 最好让 `iter` 之外的代码也具有幂等性，因为它也可能被 `benchmark` 运行多次
- 循环内的代码应该尽量的短小快速，因为这样循环才能被尽可能多的执行，结果也会更加准确

## 谜一般的性能结果

在写 `benchmark` 时，你可能会遇到一些很纳闷的棘手问题，例如以下代码：

```

#![feature(test)]

extern crate test;

fn fibonacci_u64(number: u64) -> u64 {
    let mut last: u64 = 1;
    let mut current: u64 = 0;
    let mut buffer: u64;
    let mut position: u64 = 1;

    return loop {
        if position == number {
            break current;
        }

        buffer = last;
        last = current;
        current = buffer + current;
        position += 1;
    };
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(fibonacci_u64(1), 0);
        assert_eq!(fibonacci_u64(2), 1);
        assert_eq!(fibonacci_u64(12), 89);
        assert_eq!(fibonacci_u64(30), 514229);
    }

    #[bench]
    fn bench_u64(b: &mut Bencher) {
        b.iter(|| {
            for i in 100..200 {
                fibonacci_u64(i);
            }
        });
    }
}

```

通过 cargo bench 运行后，得到一个难以置信的结果： test tests::bench\_u64 ... bench: 0 ns/iter (+/- 0)，难道 Rust 已经到达量子计算机级别了？

其实，原因藏在 LLVM 中： LLVM 认为 fibonacci\_u64 函数调用的结果没有使用，同时也认为该函数没有任何副作用(造成其它的影响，例如修改外部变量、访问网络等)，因此它有理由把这个函数调用优化掉！

解决很简单，使用 Rust 标准库中的 `black_box` 函数：

```
for i in 100..200 {
    test::black_box(fibonacci_u64(test::black_box(i)));
}
```

通过这个函数，我们告诉编译器，让它尽量少做优化，此时 LLVM 就不会再自作主张了：)

```
$ cargo bench
running 2 tests
test tests::it_works ... ignored
test tests::bench_u64 ... bench:      5,626 ns/iter (+/- 267)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured; 0 filtered out; finished
in 0.67s
```

嗯，这次结果就明显正常了。

## criterion.rs

官方 `benchmark` 有两个问题，首先就是不支持 `stable` 版本的 Rust，其次是结果有些简单，缺少更详细的统计分布。

因此社区 `benchmark` 就应运而生，其中最有名的就是 `criterion.rs`，它有几个重要特性：

- 统计分析，例如可以跟上一次运行的结果进行差异比对
- 图表，使用 `gnuplots` 展示详细的结果图表

首先，如果你需要图表，需要先安装 `gnuplots`，其次，我们需要引入相关的包，在 `Cargo.toml` 文件中新增：

```
[dev-dependencies]
criterion = "0.3"

[[bench]]
name = "my_benchmark"
harness = false
```

接着，在项目中创建一个测试文件：`$PROJECT/benches/my_benchmark.rs`，然后加入以下内容：

```

use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n-1) + fibonacci(n-2),
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| b.iter(|| fibonacci(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);

```

最后，使用 `cargo bench` 运行并观察结果：

```

Running target/release/deps/example-423eedc43b2b3a93
Benchmarking fib 20
Benchmarking fib 20: Warming up for 3.0000 s
Benchmarking fib 20: Collecting 100 samples in estimated 5.0658 s (188100 iterations)
Benchmarking fib 20: Analyzing
fib 20           time: [26.029 us 26.251 us 26.505 us]
Found 11 outliers among 99 measurements (11.11%)
  6 (6.06%) high mild
  5 (5.05%) high severe
slope [26.029 us 26.505 us] R^2          [0.8745662 0.8728027]
mean   [26.106 us 26.561 us] std. dev.    [808.98 ns 1.4722 us]
median [25.733 us 25.988 us] med. abs. dev. [234.09 ns 544.07 ns]

```

可以看出，这个结果是明显比官方的更详尽的，如果大家希望更深入的学习它的使用，可以参见[官方文档](#)。

# Cargo 使用指南

Rust 语言的名气之所以这么大，保守估计 Cargo 的贡献就占了三分之一。

Cargo 是包管理工具，可以用于依赖包的下载、编译、更新、分发等，与 Cargo 一样有名的还有 [crates.io](https://crates.io)，它是社区提供的包注册中心：用户可以将自己的包发布到该注册中心，然后其它用户通过注册中心引入该包。

---

本章内容是基于 [Cargo Book](#) 翻译，并做了一些内容优化和目录组织上的调整

---



# 上手使用

Cargo 会在安装 Rust 的时候一并进行安装，无需我们手动的操作执行，安装 Rust 参见[这里](#)。

在开始之前，先来明确一个名词：Package，由于 Crate 被翻译成包，因此 Package 再被翻译成包就很不合适，经过斟酌，我们决定翻译成项目，你也可以理解为工程、软件包，总之，在本书中 Package 意味着项目，而项目也意味着 Package。

安装完成后，接下来使用 Cargo 来创建一个新的[二进制项目](#)，二进制意味着该项目可以作为一个服务运行或被编译成可执行文件运行。

```
$ cargo new hello_world
```

这里我们使用 cargo new 创建一个新的项目，事实上该命令等价于 cargo new hello\_world --bin，bin 是 binary 的简写，代表着二进制程序，由于 --bin 是默认参数，因此可以对其进行省略。

创建成功后，先来看看项目的基本目录结构长啥样：

```
$ cd hello_world
$ tree .
.
└── Cargo.toml
    └── src
        └── main.rs

1 directory, 2 files
```

这里有一个很显眼的文件 Cargo.toml，一看就知道它是 Cargo 使用的配置文件，这个关系类似于：package.json 是 npm 的配置文件。

```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[dependencies]
```

以上就是 Cargo.toml 的全部内容，它被称之为清单( manifest )，包含了 Cargo 编译程序所需的所有元数据。

下面是 src/main.rs 的内容：

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

可以看出 Cargo 还为我们自动生成了一个 `hello world` 程序，或者说[二进制包](#)，对程序进行编译构建：

```
$ cargo build
Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

然后再运行编译出的二进制可执行文件：

```
$ ./target/debug/hello_world
Hello, world!
```

注意到路径中的 `debug` 了吗？它说明我们刚才的编译是 `Debug` 模式，该模式主要用于测试目的，如果想要进行生产编译，我们需要使用 `Release` 模式 `cargo build --release`，然后通过 `./target/release/hello_world` 运行。

除了上面的编译 + 运行方式外，在日常开发中，我们还可以使用一个简单的命令直接运行：

```
$ cargo run
Fresh hello_world v0.1.0 (file:///path/to/package/hello_world)
Running `target/hello_world`
Hello, world!
```

`cargo run` 会帮我们自动完成编译、运行的过程，当然，该命令也支持 `Release` 模式：`cargo run --release`。

---

如果你的程序在跑性能测试 `benchmark`，一定要使用 `Release` 模式，因为该模式下，程序会做大量性能优化

---

在快速了解 `Cargo` 的使用方式后，下面，我们将正式进入 `Cargo` 的学习之旅。

# 使用手册

在本章中，我们将学习 Cargo 的详细使用方式，例如 Package 的创建与管理、依赖拉取、Package 结构描述等。

# 为何会有 Cargo

根据之前学习的知识，Rust 有两种类型的包：库包和二进制包，前者是我们经常使用的依赖包，用于被其它包所引入，而后者是一个应用服务，可以编译成二进制可执行文件进行运行。

包是通过 Rust 编译器 `rustc` 进行编译的：

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

上面我们直接使用 `rustc` 对二进制包 `hello.rs` 进行编译，生成二进制可执行文件 `hello`，并对其进行运行。

该方式虽然简单，但有几个问题：

- 必须要指定文件名编译，当项目复杂后，这种编译方式也随之更加复杂
- 如果要指定编译参数，情况将更加复杂

最关键的是，外部依赖库的引入也将是一个大问题。大部分实际的项目都有不少依赖包，而这些依赖包又间接的依赖了新的依赖包，在这种复杂情况下，如何管理依赖包及其版本也是一个相当棘手的问题。

正是因为这些原因，与其使用 `rustc`，我们可以使用一个强大的包管理工具来解决问题：欢迎 Cargo 闪亮登场。

## Cargo

Cargo 解决了之前描述的所有问题，同时它保证了每次重复的构建都不会改变上一次构建的结果，这背后是通过完善且强大的依赖包版本管理来实现的。

总之，Cargo 为了实现目标，做了四件事：

- 引入两个元数据文件，包含项目的方方面面信息：`Cargo.toml` 和 `Cargo.lock`
- 获取和构建项目的依赖，例如 `Cargo.toml` 中的依赖包版本描述，以及从 `crates.io` 下载包
- 调用 `rustc` (或其它编译器) 并使用的正确的参数来构建项目，例如 `cargo build`
- 引入一些惯例，让项目的使用更加简单

毫不夸张的说，得益于 Cargo 的标准化，只要你使用它构建过一个项目，那构建其它使用 Cargo 的项目，也将不存在任何困难。

# 下载并构建 Package

如果看中 GitHub 上的某个开源 Rust 项目，那下载并构建它将是非常简单的。

```
$ git clone https://github.com/rust-lang/regex.git  
$ cd regex
```

如上所示，直接从 GitHub 上克隆下来想要的项目，然后使用 `cargo build` 进行构建即可：

```
$ cargo build  
Compiling regex v1.5.0 (file:///path/to/package/regex)
```

该命令将下载相关的依赖库，等下载成功后，再对 `package` 和下载的依赖进行一同的编译构建。

这就是包管理工具的强大之处，`cargo build` 搞定一切，而背后隐藏的复杂配置、参数你都无需关心。

# 添加依赖

[crates.io](#) 是 Rust 社区维护的中心化注册服务，用户可以在其中寻找和下载所需的包。对于 cargo 来说，默认就是从这里下载依赖。

下面我们来添加一个 `time` 依赖包，若你的 `Cargo.toml` 文件中没有 `[dependencies]` 部分，就手动添加一个，并添加目标包名和版本号：

```
[dependencies]
time = "0.1.12"
```

可以看到我们指定了 `time` 包的版本号 "0.1.12"，关于版本号，实际上还有其它的指定方式，具体参见[指定依赖项](#)章节。

如果想继续添加 `regex` 包，只需在 `time` 包后面添加即可：

```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

此时，再通过运行 `cargo build` 来重新构建，首先 cargo 会获取新的依赖以及依赖的依赖，接着对它们进行编译并更新 `Cargo.lock`：

```
$ cargo build
  Updating crates.io index
  Downloading memchr v0.1.5
  Downloading libc v0.1.10
  Downloading regex-syntax v0.2.1
  Downloading memchr v0.1.5
  Downloading aho-corasick v0.3.0
  Downloading regex v0.1.41
    Compiling memchr v0.1.5
    Compiling libc v0.1.10
    Compiling regex-syntax v0.2.1
    Compiling memchr v0.1.5
    Compiling aho-corasick v0.3.0
    Compiling regex v0.1.41
  Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
```

在 `Cargo.lock` 中包含了我们项目使用的所有依赖的准确版本信息。这个非常重要，未来就算 `regexp` 的作者升级了该包，我们依然会下载 `Cargo.lock` 中的版本，而不是最新的版本，只有这样，才能保证项目依赖包不会莫名其妙的因为更新升级导致无法编译。当然，你还可以使用 `cargo update` 来手动更新包的版本。

此时，就可以在 `src/main.rs` 中使用新引入的 `regexp` 包：

```
use regex::Regex;

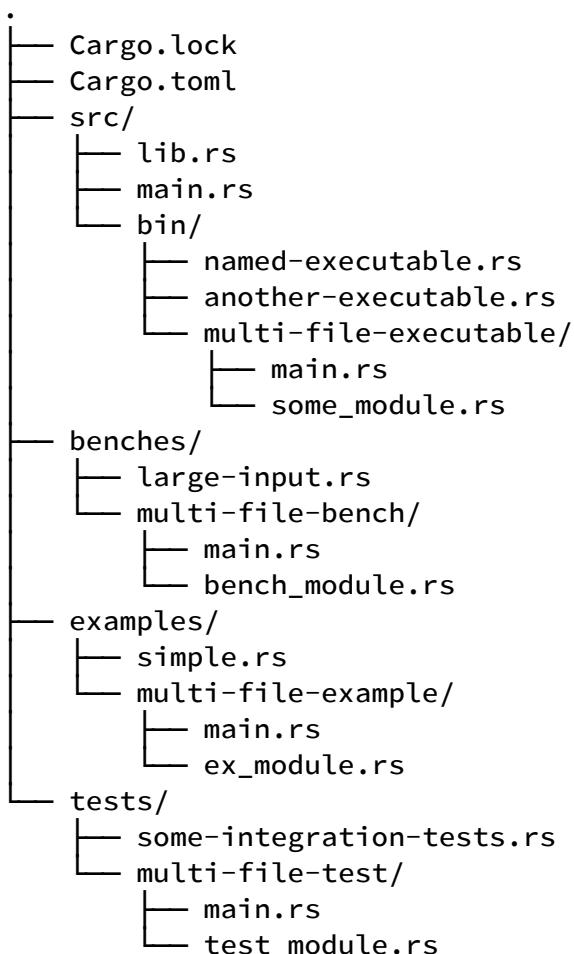
fn main() {
    let re = Regex::new(r"\d{4}-\d{2}-\d{2}").unwrap();
    println!("Did our date match? {}", re.is_match("2014-01-01"));
}
```

运行后输出：

```
$ cargo run
Running `target/hello_world`
Did our date match? true
```

# 标准的 Package 目录结构

一个典型的 Package 目录结构如下：



这也是 Cargo 推荐的目录结构，解释如下：

- `Cargo.toml` 和 `Cargo.lock` 保存在 package 根目录下
- 源代码放在 `src` 目录下
- 默认的 `lib` 包根是 `src/lib.rs`
- 默认的二进制包根是 `src/main.rs`
  - 其它二进制包根放在 `src/bin/` 目录下
- 基准测试 benchmark 放在 `benches` 目录下
- 示例代码放在 `examples` 目录下
- 集成测试代码放在 `tests` 目录下

关于 Rust 中的包和模块，[之前的章节](#)有更详细的解释。

此外，`bin`、`tests`、`examples` 等目录路径都可以通过配置文件进行配置，它们被统一称之为 [Cargo Target](#)。

# Cargo.toml vs Cargo.lock

`Cargo.toml` 和 `Cargo.lock` 是 Cargo 的两个元配置文件，但是它们拥有不同的目的：

- 前者从用户的角度出发来描述项目信息和依赖管理，因此它是由用户来编写
- 后者包含了依赖的精确描述信息，它是由 Cargo 自行维护，因此不要去手动修改

它们的关系跟 `package.json` 和 `package-lock.json` 非常相似，从 JavaScript 过来的同学应该会比较好理解。

## 是否上传本地的 `Cargo.lock`

当本地开发时，`Cargo.lock` 自然是非常重要的，但是当你要把项目上传到 Git 时，例如 GitHub，那是否上传 `Cargo.lock` 就成了一个问题。

关于是否上传，有如下经验准则：

- 从实践角度出发，如果你构建的是三方库类型的服务，请把 `Cargo.lock` 加入到 `.gitignore` 中。
- 若构建的是一个面向用户终端的产品，例如可以像命令行工具、应用程序一样执行，那就把 `Cargo.lock` 上传到源代码目录中。

例如 `axum` 是 web 开发框架，它属于三方库类型的服务，因此源码目录中不应该出现 `Cargo.lock` 的身影，它的归宿是 `.gitignore`。而 `ripgrep` 则恰恰相反，因为它是一个面向终端的产品，可以直接运行提供服务。

### 那么问题来了，为何会有这种选择？

原因是 `Cargo.lock` 会详尽描述上一次成功构建的各种信息：环境状态、依赖、版本等等，Cargo 可以使用它提供确定性的构建环境和流程，无论何时何地。这种特性对于终端服务是非常重要的：能确定、稳定的在用户环境中运行起来是终端服务最重要的特性之一。

而对于三方库来说，情况就有些不同。它不仅仅被库的开发者所使用，还会间接影响依赖链下游的使用者。用户引入了三方库是不会去看它的 `Cargo.lock` 信息的，也不应该受这个库的确定性运行条件所限制。

还有个原因，在项目中，可能会有几个依赖库引用同一个三方库的同一个版本，那如果该三方库使用了 `Cargo.lock` 文件，那可能三方库的多个版本会被引入使用，这时就会造成版本冲突。换句话说，通过指定版本的方式引用一个依赖库是无法看到该依赖库的完整情况的，而只有终端的产品才会看到这些完整的情况。

## 假设没有 Cargo.lock

Cargo.toml 是一个清单文件( manifest )包含了我们 package 的描述元数据。例如，通过以下内容可以说明对另一个 package 的依赖：

```
[package]
name = "hello_world"
version = "0.1.0"

[dependencies]
regex = { git = "https://github.com/rust-lang/regex.git" }
```

可以看到，只有一个依赖，且该依赖的来源是 GitHub 上一个特定的仓库。由于我们没有指定任何版本信息，Cargo 会自动拉取该依赖库的最新版本( master 或 main 分支上的最新 commit )。

这种使用方式，其实就错失了包管理工具的最大的优点：版本管理。例如你在今天构建使用了版本 A，然后过了一段时间后，由于依赖包的升级，新的构建却使用了大更新版本 B，结果因为版本不兼容，导致了构建失败。

可以看出，确保依赖版本的确定性是非常重要的：

```
[dependencies]
regex = { git = "https://github.com/rust-lang/regex.git", rev = "9f9f693" }
```

这次，我们使用了指定 rev ( revision )的方式来构建，那么不管未来何时再次构建，使用的依赖库都会是该 rev，而不是最新的 commit。

但是，这里还有一个问题：rev 需要手动的管理，你需要在每次更新包的时候都思考下 SHA-1，这显然非常麻烦。

## 当有了 Cargo.lock 后

当有了 Cargo.lock 后，我们无需手动追踪依赖库的 rev，Cargo 会自动帮我们完成，还是之前的清单：

```
[package]
name = "hello_world"
version = "0.1.0"

[dependencies]
regex = { git = "https://github.com/rust-lang/regex.git" }
```

第一次构建时，Cargo 依然会拉取最新的 master commit，然后将以下信息写到 Cargo.lock 文件中：

```
[[package]]
name = "hello_world"
version = "0.1.0"
dependencies = [
    "regex 1.5.0 (git+https://github.com/rust-lang/regex.git#9f9f693768c584971a4d53bc3c586c33ed3a6831)",
]

[[package]]
name = "regex"
version = "1.5.0"
source = "git+https://github.com/rust-lang/regex.git#9f9f693768c584971a4d53bc3c586c33ed3a6831"
```

可以看出，其中包含了依赖库的准确 rev 信息。当未来再次构建时，只要项目中还有该 Cargo.lock 文件，那构建依然会拉取同一个版本的依赖库，并且再也无需我们手动去管理 rev 的 SHA 信息！

## 更新依赖

由于 Cargo.lock 会锁住依赖的版本，你需要通过手动的方式将依赖更新到新的版本：

```
$ cargo update          # 更新所有依赖
$ cargo update -p regex # 只更新 “regex”
```

以上命令将使用新的版本信息重新生成 Cargo.lock，需要注意的是 cargo update -p regex 传递的参数实际上是一个 Package ID，regex 只是一个简写形式。

# 测试和 CI

Cargo 可以通过 `cargo test` 命令运行项目中的测试文件：它会在 `src/` 底下的文件寻找单元测试，也会在 `tests/` 目录下寻找集成测试。

```
$ cargo test
Compiling regex v1.5.0 (https://github.com/rust-lang/regex.git#9f9f693)
Compiling hello_world v0.1.0 (file:///path/to/package/hello_world)
    Running target/test/hello_world-9c2b65bbb79eabce

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

从上面结果可以看出，项目中实际上还没有任何测试代码。

事实上，除了单元测试、集成测试，`cargo test` 还会编译 `examples/` 下的示例文件以及[文档中的示例](#)。

如果希望深入学习如何在 Rust 编写及运行测试，请查阅[该章节](#)。

## CI

持续集成是软件开发中异常重要的一环，大家应该都听说过 Jenkins，它就是一个拥有悠久历史的持续集成工具。简单来说，持续集成会定期拉取同一个项目中所有成员的相关代码，对其进行自动化构建。

在没有持续集成前，首先开发者需要手动编译代码并运行单元测试、集成测试等基础测试，然后启动项目相关的所有服务，接着测试人员开始介入对整个项目进行回归测试、黑盒测试等系统化的测试，当测试通过后，最后再手动发布到指定的环境中运行，这个过程是非常冗长，且所有成员都需要同时参与的。

在有了持续集成后，只要编写好相应的编译、测试、发布配置文件，那持续集成平台会自动帮助我们完成整个相关的流程，期间无需任何人介入，高效且可靠。

### GitHub Actions

关于如何使用 GitHub Actions 进行持续集成，在[之前的章节](#)已经有过详细的介绍，这里就不再赘述。

## Travis CI

以下是 Travis CI 需要的一个简单的示例配置文件:

```
language: rust
rust:
  - stable
  - beta
  - nightly
matrix:
  allow_failures:
    - rust: nightly
```

以上配置将测试所有的 Rust 发布版本，但是 nightly 版本的构建失败不会导致全局测试的失败，可以查看 [Travis CI Rust 文档](#) 获取更详细的说明。

## Gitlab CI

以下是一个示例 .gitlab-ci.yml 文件:

```
stages:
  - build

rust-latest:
  stage: build
  image: rust:latest
  script:
    - cargo build --verbose
    - cargo test --verbose

rust-nightly:
  stage: build
  image: rustlang/rust:nightly
  script:
    - cargo build --verbose
    - cargo test --verbose
  allow_failure: true
```

这里将测试 stable 和 nightly 发布版本，同样的， nightly 下的测试失败不会导致全局测试的失败。查看 [Gitlab CI 文档](#) 获取更详细的说明。

# Cargo 缓存

Cargo 使用了缓存的方式提升构建效率，当构建时，Cargo 会将已下载的依赖包放在 `CARGO_HOME` 目录下，下面一起来看看。

## Cargo Home

默认情况下，Cargo Home 所在的目录是 `$HOME/.cargo/`，例如在 `macos`，对应的目录是：

```
$ echo $HOME/.cargo/  
/Users/sunfei/.cargo/
```

我们也可以通过修改 `CARGO_HOME` 环境变量的方式来重新设定该目录的位置。若你需要在项目中通过代码的方式来获取 `CARGO_HOME`，`home` 包提供了相应的 API。

---

注意！Cargo Home 目录的内部结构并没有稳定化，在未来可能会发生变化

---

## 文件

- `config.toml` 是 Cargo 的全局配置文件，具体请查看[这里](#)
- `credentials.toml` 为 `cargo login` 提供私有化登录证书，用于登录 `package` 注册中心，例如 `crates.io`
- `.crates.toml`, `.crates2.json` 这两个是隐藏文件，包含了通过 `cargo install` 安装的包的 `package` 信息，**请不要手动修改！**

## 目录

- `bin` 目录包含了通过 `cargo install` 或 `rustup` 下载的包编译出的可执行文件。你可以将该目录加入到 `$PATH` 环境变量中，以实现对这些可执行文件的直接访问
- `git` 中存储了 `Git` 的资源文件：
  - `git/db`，当一个包依赖某个 `git` 仓库时，Cargo 会将该仓库克隆到 `git/db` 目录下，如果未来需要还会对其进行更新

- `git/checkouts`，若指定了 `git` 源和 `commit`，那相应的仓库就会从 `git/db` 中 `checkout` 到该目录下，因此同一个仓库的不同 `checkout` 共存成为了可能
- `registry` 包含了注册中心(例如 `crates.io`)的元数据和 `packages`
  - `registry/index` 是一个 `git` 仓库，包含了注册中心中所有可用包的元数据(版本、依赖等)
  - `registry/cache` 中保存了已下载的依赖，这些依赖包以 `gzip` 的压缩档案形式保存，后缀名为 `.crate`
  - `registry/src`，若一个已下载的 `.crate` 档案被一个 `package` 所需要，该档案会被解压缩到 `registry/src` 文件夹下，最终 `rustc` 可以在其中找到所需的 `.rs` 文件

## 在 CI 时缓存 Cargo Home

为了避免持续集成时重复下载所有的包依赖，我们可以将 `$CARGO_HOME` 目录进行缓存，但缓存整个目录是效率低下的，原因是源文件可能会被缓存两次。

例如我们依赖一个包 `serde 1.0.92`，如果将整个 `$CACHE_HOME` 目录缓存，那么 `serde` 的源文件就会被缓存两次：在 `registry/cache` 中的 `serde-1.0.92.crate` 以及 `registry/src` 下被解压缩的 `.rs` 文件。

因此，在 CI 构建时，出于效率的考虑，我们仅应该缓存以下目录：

- `bin/`
- `registry/index/`
- `registry/cache/`
- `git/db/`

## 清除缓存

理论上，我们可以手动移除缓存中的任何一部分，当后续有包需要时 `Cargo` 会尽可能去恢复这些资源：

- 解压缩 `registry/cache` 下的 `.crate` 档案
- 从 `.git` 中 `checkout` 缓存的仓库
- 如果以上都没了，会从网络上重新下载

你也可以使用 `cargo-cache` 包来选择性的清除 `cache` 中指定的部分，当然，它还可以用来查看缓存中的组件大小。

## 构建时卡住: Blocking waiting for file lock ..

在开发过程中，或多或少我们都会碰到这种问题，例如你同时打开了 VSCode IDE 和终端，然后在 `Cargo.toml` 中刚添加了一个新的依赖。

此时 IDE 会捕捉到这个修改然后自动去重新下载依赖(这个过程可能还会更新 `crates.io` 使用的索引列表)，在此过程中，Cargo 会将相关信息写入到 `$HOME/.cargo/.package_cache` 下，并将其锁住。

如果你试图在另一个地方(例如终端)对同一个项目进行构建，就会报错: `Blocking waiting for file lock on package cache`。

解决办法很简单：

- 既然下载慢，那就使用[国内的注册服务](#)，不再使用 `crates.io`
- 耐心等待持有锁的用户构建完成
- 强行停止正在构建的进程，例如杀掉 IDE 使用的 `rust-analyzer` 插件进程，然后删除 `$HOME/.cargo/.package_cache` 目录

# 构建( Build )缓存

`cargo build` 的结果会被放入项目根目录下的 `target` 文件夹中，当然，这个位置可以三种方式更改：设置 `CARGO_TARGET_DIR` 环境变量、`build.target-dir` 配置项以及 `--target-dir` 命令行参数。

## target 目录结构

`target` 目录的结构取决于是否使用 `--target` 标志为特定的平台构建。

### 不使用 --target

若 `--target` 标志没有指定，Cargo 会根据宿主机架构进行构建，构建结果会放入项目根目录下的 `target` 目录中，`target` 下每个子目录中包含了相应的 [发布配置profile](#) 的构建结果，例如 `release`、`debug` 是自带的 profile，前者往往用于生产环境，因为会做大量的性能优化，而后者则用于开发环境，此时的编译效率和报错信息是最好的。

除此之外我们还可以定义自己想要的 profile，例如用于测试环境的 `profile : test`，用于预发环境的 `profile : pre-prod` 等。

目录	描述
<code>target/debug/</code>	包含了 <code>dev</code> profile 的构建输出( <code>cargo build</code> 或 <code>cargo build --debug</code> )
<code>target/release/</code>	<code>release</code> profile 的构建输出， <code>cargo build --release</code>
<code>target/foo/</code>	自定义 <code>foo</code> profile 的构建输出， <code>cargo build --profile=foo</code>

出于历史原因：

- `dev` 和 `test` profile 的构建结果都存放在 `debug` 目录下
- `release` 和 `bench` profile 则存放在 `release` 目录下
- 用户定义的 profile 存在同名的目录下

### 使用 --target

当使用 `--target XXX` 为特定的平台编译后，输出会放在 `target/XXX/` 目录下：

目录	示例
target/<triple>/debug/	target/thumbv7em-none-eabihf/debug/
target/<triple>/release/	target/thumbv7em-none-eabihf/release/

**注意：**，当没有使用 `--target` 时，Cargo 会与构建脚本和过程宏一起共享你的依赖包，对于每个 `rustc` 命令调用而言，`RUSTFLAGS` 也将被共享。

而使用 `--target` 后，构建脚本、过程宏会针对宿主机的 CPU 架构进行各自构建，且不会共享 `RUSTFLAGS`。

## target 子目录说明

在 profile 文件夹中(例如 `debug` 或 `release`)，包含编译后的最终成果：

目录	描述
target/debug/	包含编译后的输出，例如二进制可执行文件、库对象( library target )
target/debug/examples /	包含示例对象( example target )

还有一些命令会在 `target` 下生成自己的独立目录：

目录	描述
target/doc/	包含通过 <code>cargo doc</code> 生成的文档
target/package/	包含 <code>cargo package</code> 或 <code>cargo publish</code> 生成的输出

Cargo 还会创建几个用于构建过程的其它类型目录，它们的目录结构只应该被 Cargo 自身使用，因此可能会在未来发生变化：

目录	描述
target/debugdeps	依赖和其它输出成果
target/debug/incremental	<code>rustc</code> 增量编译的输出，该缓存可以用于提升后续的编译速度
target/debug/build/	构建脚本的输出

## 依赖信息文件

在每一个编译成果的旁边，都有一个依赖信息文件，文件后缀是 `.d`。该文件的语法类似于 `Makefile`，用于说明构建编译成果所需的所有依赖包。

该文件往往用于提供给外部的构建系统，这样它们就可以判断 `Cargo` 命令是否需要再次被执行。

文件中的路径默认是绝对路径，你可以通过 `build.dep-info-basedir` 配置项来修改为相对路径。

```
# 关于 `.` 文件的一个示例 : target/debug/foo.d
/path/to/myproj/target/debug/foo: /path/to/myproj/src/lib.rs
/path/to/myproj/src/main.rs
```

## 共享缓存

`sccache` 是一个三方工具，可以用于在不同的工作空间中共享已经构建好的依赖包。

为了设置 `sccache`，首先需要使用 `cargo install sccache` 进行安装，然后在调用 `Cargo` 之前将 `RUSTC_WRAPPER` 环境变量设置为 `sccache`。

- 如果用的 `bash`，可以将 `export RUSTC_WRAPPER=sccache` 添加到 `.bashrc` 中
- 也可以使用 `build.rustc-wrapper` 配置项

# 进阶指南

进阶指南包含了 Cargo 的参考级内容，大家可以先看一遍了解下大概有什么，然后在后面需要时，再回来查询如何使用。

# 指定依赖项

我们的项目可以引用在 [crates.io](#) 或 [GitHub](#) 上的依赖包，也可以引用存放在本地文件系统中的依赖包。

大家可能会想，直接从前两个引用即可，为何还提供了本地方式？可以设想下，如果你要有一个正处于开发中的包，然后需要在本地的另一个项目中引用测试，那是将该包先传到网上，然后再引用简单，还是直接从本地路径的方式引用简单呢？答案显然不言而喻。

本章节，我们一起来看看有哪些方式可以指定和引用三方依赖包。

## 从 crates.io 引入依赖包

默认设置下，Cargo 就从 [crates.io](#) 上下载依赖包，只需要一个包名和版本号即可：

```
[dependencies]
time = "0.1.12"
```

字符串 "0.1.12" 是一个 [semver](#) 格式的版本号，符合 "x.y.z" 的形式，其中 x 被称为主版本 major，y 被称为小版本 minor，而 z 被称为补丁 patch，可以看出从左到右，版本的影响范围逐步降低，补丁的更新是无关痛痒的，并不会造成 API 的兼容性被破坏。

"0.1.12" 中并没有任何额外的符号，在版本语义上，它跟使用了 ^ 的 "^0.1.12" 是相同的，都是指定非常具体的版本进行引入。

但是 ^ 能做的更多。

---

npm 使用的就是 semver 版本号，从 JavaScript 过来的同学应该非常熟悉。

---

### ^ 指定版本

与之前的 "0.1.12" 不同，^ 可以指定一个版本号范围，**然后会使用该范围内的最大版本号来引用对应的包。**

只要新的版本号没有修改最左边的非零数字，那该版本号就在允许的版本号范围内。例如 "^0.1.12" 最左边的非零数字是 1，因此，只要新的版本号是 "0.1.z" 就可以落在范围内，而 0.2.0 显然就没有落在范围内，因此通过 "^0.1.12" 引入的依赖包是无法被升级到 0.2.0 版本的。

同理，若是 "`^1.0`"，则 `1.1` 在范围内，`2.0` 则不在。大家思考下，"`^0.0.1`" 与哪些版本兼容？答案是：无，因为它最左边的数字是 `1`，而该数字已经退无可退，我们又不能修改 `1`，因此没有版本落在范围内。

```
^1.2.3  :=  >=1.2.3, <2.0.0
^1.2     :=  >=1.2.0, <2.0.0
^1       :=  >=1.0.0, <2.0.0
^0.2.3   :=  >=0.2.3, <0.3.0
^0.2     :=  >=0.2.0, <0.3.0
^0.0.3   :=  >=0.0.3, <0.0.4
^0.0     :=  >=0.0.0, <0.1.0
^0       :=  >=0.0.0, <1.0.0
```

以上是更多的例子，**事实上，这个规则跟 SemVer 还有所不同**，因为对于 SemVer 而言，`0.x.y` 的版本是没有其它版本与其兼容的，而对于 Rust，只要版本号 `0.x.y` 满足：`z>=y` 且 `x>0` 的条件，那它就能更新到 `0.x.z` 版本。

## ~ 指定版本

~ 指定了最小化版本：

```
~1.2.3  :=  >=1.2.3, <1.3.0
~1.2     :=  >=1.2.0, <1.3.0
~1       :=  >=1.0.0, <2.0.0
```

## \* 通配符

这种方式允许将 `*` 所在的位置替换成任何数字：

```
*       :=  >=0.0.0
1.*     :=  >=1.0.0, <2.0.0
1.2.*   :=  >=1.2.0, <1.3.0
```

不过 crates.io 并不允许我们只使用孤零零一个 `*` 来指定版本号：`*`。

## 比较符

可以使用比较符的方式来指定一个版本号范围或一个精确的版本号：

```
>= 1.2.0  
> 1  
< 2  
= 1.2.3
```

同时还能使用比较符进行组合，并通过逗号分隔：

```
>= 1.2, < 1.5
```

需要注意，以上的版本号规则仅仅针对 `crate.io` 和基于它搭建的注册服务(例如科大服务源)，其它注册服务(例如 GitHub )有自己相应的规则。

## 从其它注册服务引入依赖包

为了使用 `crates.io` 之外的注册服务，我们需要对 `$HOME/.cargo/config.toml` (`$CARGO_HOME` 下) 文件进行配置，添加新的服务提供商，有两种方式可以实现。

---

由于国内访问国外注册服务的不稳定性，我们可以使用[科大的注册服务](#)来提升下载速度，以下注册服务的链接都是科大的

首先是在 `crates.io` 之外添加新的注册服务，修改 `.cargo/config.toml` 添加以下内容：

```
[registries]  
ustc = { index = "https://mirrors.ustc.edu.cn/crates.io-index/" }
```

对于这种方式，我们的项目的 `Cargo.toml` 中的依赖包引入方式也有所不同：

```
[dependencies]  
time = { registry = "ustc" }
```

在重新配置后，初次构建可能要较久的时间，因为要下载更新 `ustc` 注册服务的索引文件，还挺大的...

注意，这一种使用方式最大的缺点就是在引用依赖包时要指定注册服务: `time = { registry = "ustc" }`。

而第二种方式就不需要，因为它是直接使用新注册服务来替代默认的 `crates.io`。

```
[source.crates-io]
replace-with = 'ustc'

[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

上面配置中的第一个部分，首先将源 `source.crates-io` 替换为 `ustc`，然后在第二部分指定了 `ustc` 源的地址。

---

注意，如果你要发布包到 `crates.io` 上，那该包的依赖也必须在 `crates.io` 上

---

## 引入 git 仓库作为依赖包

若要引入 git 仓库中的库作为依赖包，你至少需要提供一个仓库的地址：

```
[dependencies]
regex = { git = "https://github.com/rust-lang/regex" }
```

由于没有指定版本，Cargo 会假定我们使用 `master` 或 `main` 分支的最新 `commit`。你可以使用 `rev`、`tag` 或 `branch` 来指定想要拉取的版本。例如下面代码拉取了 `next` 分支上的最新 `commit`：

```
[dependencies]
regex = { git = "https://github.com/rust-lang/regex", branch = "next" }
```

任何非 `tag` 和 `branch` 的类型都可以通过 `rev` 来引入，例如通过最近一次 `commit` 的哈希值引入：`rev = "4c59b707"`，再比如远程仓库提供的的具名引用：`rev = "refs/pull/493/head"`。

一旦 `git` 依赖被拉取下来，该版本就会被记录到 `Cargo.lock` 中进行锁定。因此 `git` 仓库中后续新的提交不再会被自动拉取，除非你通过 `cargo update` 来升级。需要注意的是锁定一旦被删除，那 Cargo 依然会按照 `Cargo.toml` 中配置的地址和版本去拉取新的版本，如果你配置的版本不正确，那可能会拉取下来一个不兼容的新版本！

**因此不要依赖锁定来完成版本的控制，而应该老老实实的在 `Cargo.toml` 小心配置你希望使用的版本。**

如果访问的是私有仓库，你可能需要授权来访问该仓库，可以查看[这里](#)了解授权的方式。

## 通过路径引入本地依赖包

Cargo 支持通过路径的方式来引入本地的依赖包：一般来说，本地依赖包都是同一个项目内的内部包，例如假设我们有一个 `hello_world` 项目( package )，现在在其根目录下新建一个包：

```
# 在 hello_world/ 目录下
$ cargo new hello_utils
```

新建的 `hello_utils` 文件夹跟 `src`、`Cargo.toml` 同级，现在修改 `Cargo.toml` 让 `hello_world` 项目引入新建的包：

```
[dependencies]
hello_utils = { path = "hello_utils" }
# 以下路径也可以
# hello_utils = { path = "./hello_utils" }
# hello_utils = { path = "../hello_world/hello_utils" }
```

但是，此时的 `hello_world` 是无法发布到 `crates.io` 上的。想要发布，需要先将 `hello_utils` 先发布到 `crates.io` 上，然后再通过 `crates.io` 的方式来引入：

```
[dependencies]
hello_utils = { path = "hello_utils", version = "0.1.0" }
```

---

注意！使用 `path` 指定依赖的 package 将无法发布到 `crates.io`，除非 `path` 存在于 `[dev-dependencies]` 中。当然，你还可以使用多种引用混合的方式来解决这个问题，下面将进行介绍

---

## 多引用方式混合

实际上，我们可以同时使用多种方式来引入同一个包，例如本地引入和 `crates.io`：

```
[dependencies]
# 本地使用时，通过 path 引入，
# 发布到 `crates.io` 时，通过 `crates.io` 的方式引入： version = "1.0"
bitflags = { path = "my-bitflags", version = "1.0" }

# 本地使用时，通过 git 仓库引入
# 当发布时，通过 `crates.io` 引入： version = "1.0"
smallvec = { git = "https://github.com/servo/rust-smallvec", version = "1.0" }

# N.B. 若 version 无法匹配，Cargo 将无法编译
```

这种方式跟下章节将要讲述的依赖覆盖类似，但是前者只会应用到当前声明的依赖包上。

## 根据平台引入依赖

我们还可以根据特定的平台来引入依赖:

```
[target.'cfg(windows)'.dependencies]
winhttp = "0.4.0"

[target.'cfg(unix)'.dependencies]
openssl = "1.0.1"

[target.'cfg(target_arch = "x86")'.dependencies]
native = { path = "native/i686" }

[target.'cfg(target_arch = "x86_64")'.dependencies]
native = { path = "native/x86_64" }
```

此处的语法跟 Rust 的 `#[cfg]` 语法非常相像，因此我们还能使用逻辑操作符进行控制:

```
[target.'cfg(not(unix))'.dependencies]
openssl = "1.0.1"
```

这里的意思是，当不是 `unix` 操作系统时，才对 `openssl` 进行引入。

如果你想要知道 `cfg` 能够作用的目标，可以在终端中运行 `rustc --print=cfg` 进行查询。当然，你可以指定平台查询: `rustc --print=cfg --target=x86_64-pc-windows-msvc`，该命令将对 64bit 的 Windows 进行查询。

聪明的同学已经发现，这非常类似于条件依赖引入，那我们是不是可以根据自定义的条件来决定是否引入某个依赖呢？具体答案参见后续的 [feature](#) 章节。这里是一个简单的示例:

```
[dependencies]
foo = { version = "1.0", optional = true }
bar = { version = "1.0", optional = true }

[features]
fancy-feature = ["foo", "bar"]
```

但是需要注意的是，你如果妄图通过 `cfg(feature)`、`cfg(debug_assertions)`，`cfg(test)` 和 `cfg(proc_macro)` 的方式来条件引入依赖，那是不可行的。

Cargo 还允许通过下面的方式来引入平台特定的依赖:

```
[target.x86_64-pc-windows-gnu.dependencies]
winhttp = "0.4.0"

[target.i686-unknown-linux-gnu.dependencies]
openssl = "1.0.1"
```

## 自定义 target 引入

如果你在使用自定义的 target : 例如 `--target bar.json` , 那么可以通过下面方式来引入依赖:

```
[target.bar.dependencies]
winhttp = "0.4.0"

[target.my-special-i686-platform.dependencies]
openssl = "1.0.1"
native = { path = "native/i686" }
```

---

需要注意, 这种使用方式在 `stable` 版本的 Rust 中无法被使用, 建议大家如果没有特别的需求, 还是使用之前提到的 feature 方式

---

## [dev-dependencies]

你还可以为项目添加只在测试时需要的依赖库, 类似于 `package.json` ( Nodejs )文件中的 `devDependencies` , 可以在 `Cargo.toml` 中添加 `[dev-dependencies]` 来实现:

```
[dev-dependencies]
tempdir = "0.3"
```

这里的依赖只会在运行测试、示例和 benchmark 时才会被引入。并且, 假设 A 包引用了 B , 而 B 通过 `[dev-dependencies]` 的方式引用了 c 包, 那 A 是不会引用 c 包的。

当然, 我们还可以指定平台特定的测试依赖包:

```
[target.'cfg(unix)'.dev-dependencies]
mio = "0.0.1"
```

---

注意，当发布包到 crates.io 时，`[dev-dependencies]` 中的依赖只有指定了 `version` 的才会被包含在发布包中。况且，再加上测试稳定性的考虑，我们建议为 `[dev-dependencies]` 中的包指定相应的版本号

---

## [build-dependencies]

我们还可以指定某些依赖仅用于构建脚本：

```
[build-dependencies]
cc = "1.0.3"
```

当然，平台特定的依然可以使用：

```
[target.'cfg(unix)'.build-dependencies]
cc = "1.0.3"
```

有一点需要注意：构建脚本(`build.rs`)和项目的正常代码是彼此独立，因此它们的依赖不能互通：构建脚本无法使用 `[dependencies]` 或 `[dev-dependencies]` 中的依赖，而 `[build-dependencies]` 中的依赖也无法被构建脚本之外的代码所使用。

## 选择 features

如果你依赖的包提供了条件性的 `features`，你可以指定使用哪一个：

```
[dependencies.awesome]
version = "1.3.5"
default-features = false # 不要包含默认的 features，而是通过下面的方式来指定
features = ["secure-password", "civet"]
```

更多的信息参见 [Features 章节](#)

## 在 Cargo.toml 中重命名依赖

如果你想要实现以下目标：

- 避免在 Rust 代码中使用 `use foo as bar`
- 依赖某个包的多个版本
- 依赖来自于不同注册服务的同名包

那可以使用 Cargo 提供的 `package key` :

```
[package]
name = "mypackage"
version = "0.0.1"

[dependencies]
foo = "0.1"
bar = { git = "https://github.com/example/project", package = "foo" }
baz = { version = "0.1", registry = "custom", package = "foo" }
```

此时，你的代码中可以使用三个包：

```
extern crate foo; // 来自 crates.io
extern crate bar; // 来自 git repository
extern crate baz; // 来自 registry `custom`
```

有趣的是，由于这三个 `package` 的名称都是 `foo` (在各自的 `Cargo.toml` 中定义)，因此我们显式的通过 `package = "foo"` 的方式告诉 Cargo：我们需要的就是这个 `foo package`，虽然它被重命名为 `bar` 或 `baz`。

有一点需要注意，当使用可选依赖时，如果你将 `foo` 包重命名为 `bar` 包，那引用前者的 `feature` 时的路径名也要做相应的修改：

```
[dependencies]
bar = { version = "0.1", package = 'foo', optional = true }

[features]
log-debug = ['bar/log-debug'] # 若使用 'foo/log-debug' 会导致报错
```

# 依赖覆盖

依赖覆盖对于本地开发来说，是很常见的，大部分原因都是我们希望在某个包发布到 `crates.io` 之前使用它，例如：

- 你正在同时开发一个包和一个项目，而后者依赖于前者，你希望能在该项目中对正在开发的包进行测试
- 你引入的一个依赖包在 `master` 分支发布了新的代码，恰好修复了某个 bug，因此你希望能单独对该分支进行下测试
- 你即将发布一个包的新版本，为了确保新版本正常工作，你需要对其进行集成测试
- 你为项目的某个依赖包提了一个 PR 并解决了一个重要 bug，在等待合并到 `master` 分支，但是时间不等人，因此你决定先使用自己修改的版本，等未来合并后，再继续使用官方版本

下面我们来具体看看类似的问题该如何解决。

---

上一章节中我们讲了如果通过[多种引用方式](#)来引入一个包，其实这也是一种依赖覆盖。

---

## 测试 bugfix 版本

假设我们有一个项目正在使用 `uuid` 依赖包，但是却不幸地发现了一个 bug，由于这个 bug 影响了使用，没办法等到官方提交新版本，因此还是自己修复为好。

我们项目的 `Cargo.toml` 内容如下：

```
[package]
name = "my-library"
version = "0.1.0"

[dependencies]
uuid = "0.8.2"
```

为了修复 bug，首先需要将 `uuid` 的源码克隆到本地，笔者是克隆到和项目同级的目录下：

```
git clone https://github.com/uuid-rs/uuid
```

下面，修改项目的 `Cargo.toml` 添加以下内容以引入本地克隆的版本：

```
[patch.crates-io]
uuid = { path = "../uuid" }
```

这里我们使用自己修改过的 `patch` 来覆盖来自 `crates.io` 的版本，由于克隆下来的 `uuid` 目录和我们的项目同级，因此通过相对路径 `"../uuid"` 即可定位到。

在成功为 `uuid` 打了本地补丁后，现在尝试在项目下运行 `cargo build`，但是却报错了，而且报错内容有一些看不太懂：

```
$ cargo build
  Updating crates.io index
warning: Patch `uuid v1.0.0-alpha.1 (/Users/sunfei/development/rust/demos/uuid)` was
not used in the crate graph.
Check that the patched package version and available features are compatible
with the dependency requirements. If the patch has a different version from
what is locked in the Cargo.lock file, run `cargo update` to use the new
version. This may also occur with an optional dependency that is not enabled.
```

具体原因比较复杂，但是仔细观察，会发现克隆下来的 `uuid` 的版本是 `v1.0.0-alpha.1` (在 `"../uuid/Cargo.toml"` 中可以查看)，然后我们本地引入的 `uuid` 版本是 `0.8.2`，根据之前讲过的 `crates.io` 的版本规则，这两者是不兼容的，`0.8.2` 只能升级到 `0.8.z`，例如 `0.8.3`。

既然如此，我们先将 `"../uuid/Cargo.toml"` 中的 `version = "1.0.0-alpha.1"` 修改为 `version = "0.8.3"`，然后看看结果先：

```
$ cargo build
  Updating crates.io index
  Compiling uuid v0.8.3 (/Users/sunfei/development/rust/demos/uuid)
```

大家注意到最后一行了吗？我们成功使用本地的 `0.8.3` 版本的 `uuid` 作为最新的依赖，因此也侧面证明了，补丁 `patch` 的版本也必须遵循相应的版本兼容规则！

如果修改后还是有问题，大家可以试试以下命令，指定版本进行更新：

```
% cargo update -p uuid --precise 0.8.3
  Updating crates.io index
  Updating uuid v0.8.3 (/Users/sunfei/development/rust/demos/uuid) -> v0.8.3
```

修复 bug 后，我们可以提交 pr 给 `uuid`，一旦 pr 被合并到了 `master` 分支，你可以直接通过以下方式来使用补丁：

```
[patch.crates-io]
uuid = { git = 'https://github.com/uuid-rs/uuid' }
```

等未来新的内容更新到 crates.io 后，大家就可以移除这个补丁，直接更新 [dependencies] 中的 uuid 版本即可！

## 使用未发布的小版本

还是 uuid 包，这次假设我们要为它新增一个特性，同时我们已经修改完毕，在本地测试过，并提交了相应的 pr，下面一起来看看该如何在它发布到 crates.io 之前继续使用。

再做一个假设，对于 uuid 来说，目前 crates.io 上的版本是 1.0.0，在我们提交了 pr 并合并到 master 分支后，master 上的版本变成了 1.0.1，这意味着未来 crates.io 上的版本也将变成 1.0.1。

为了使用新加的特性，同时当该包在未来发布到 crates.io 后，我们可以自动使用 crates.io 上的新版本，而无需再使用 patch 补丁，可以这样修改 Cargo.toml：

```
[package]
name = "my-library"
version = "0.1.0"

[dependencies]
uuid = "1.0.1"

[patch.crates-io]
uuid = { git = 'https://github.com/uuid-rs/uuid' }
```

注意，我们将 [dependencies] 中的 uuid 版本提前修改为 1.0.1，由于该版本在 crates.io 尚未发布，因此 patch 版本会被使用。

现在，我们的项目是基于 patch 版本的 uuid 来构建，也就是从 github 的 master 分支中拉取最新的 commit 来构建。一旦未来 crates.io 上有了 1.0.1 版本，那项目就会继续基于 crates.io 来构建，此时， patch 就可以删除了。

## 间接使用 patch

现在假设项目 A 的依赖是 B 和 uuid，而 B 的依赖也是 uuid，此时我们可以让 A 和 B 都使用来自 GitHub 的 patch 版本，配置如下：

```
[package]
name = "my-binary"
version = "0.1.0"

[dependencies]
my-library = { git = 'https://example.com/git/my-library' }
uuid = "1.0.1"

[patch.crates-io]
uuid = { git = 'https://github.com/uuid-rs/uuid' }
```

如上所示，`patch` 不仅仅对于 `my-binary` 项目有用，对于 `my-binary` 的依赖 `my-library` 来说，一样可以间接生效。

## 非 crates.io 的 patch

若我们想要覆盖的依赖并不是来自 `crates.io`，就需要对 `[patch]` 做一些修改。例如依赖是 `git` 仓库，然后使用本地路径来覆盖它：

```
[patch."https://github.com/your/repository"]
my-library = { path = "../my-library/path" }
```

easy，轻松搞定！

## 使用未发布的大版本

现在假设我们要发布一个大版本 `2.0.0`，与之前类似，可以将 `Cargo.toml` 修改如下：

```
[dependencies]
uuid = "2.0"

[patch.crates-io]
uuid = { git = "https://github.com/uuid-rs/uuid", branch = "2.0.0" }
```

此时 `2.0` 版本在 `crates.io` 上还不存在，因此我们使用了 `patch` 版本且指定了 `branch = "2.0.0"`。

## 间接使用 patch

这里需要注意，与之前的小版本不同，大版本的 `patch` 不会发生间接的传递！，例如：

```
[package]
name = "my-binary"
version = "0.1.0"

[dependencies]
my-library = { git = 'https://example.com/git/my-library' }
uuid = "1.0"

[patch.crates-io]
uuid = { git = 'https://github.com/uuid-rs/uuid', branch = '2.0.0' }
```

以上配置中，`my-binary` 将继续使用 `1.x.y` 系列的版本，而 `my-library` 将使用最新的 `2.0.0` patch。

原因是，大版本更新往往带来破坏性的功能，Rust 为了让我们平稳的升级，采用了滚动的方式：在依赖图中逐步推进更新，而不是一次性全部更新。

## 多版本[patch]

在之前章节，我们介绍过如何使用 `package` key 来重命名依赖包，现在来看看如何使用它同时引入多个 `patch`。

假设，我们对 `serde` 有两个新的 `patch` 需求：

- `serde` 官方解决了一个 bug 但是还没发布到 `crates.io`，我们想直接从 `git` 仓库的最新 `commit` 拉取版本 `1.*`
- 我们自己为 `serde` 添加了新的功能，命名为 `2.0.0` 版本，并将该版本上传到自己的 `git` 仓库中

为了满足这两个 `patch`，可以使用如下内容的 `Cargo.toml`：

```
[patch.crates-io]
serde = { git = 'https://github.com/serde-rs/serde' }
serde2 = { git = 'https://github.com/example/serde', package = 'serde', branch = 'v2'
}
```

第一行说明，第一个 `patch` 从官方仓库 `main` 分支的最新 `commit` 拉取，而第二个则从我们自己的仓库拉取 `v2` 分支，同时将其重命名为 `serde2`。

这样，在代码中就可以分别通过 `serde` 和 `serde2` 引用不同版本的依赖库了。

## 通过[path]来覆盖依赖

有时我们只是临时性地对一个项目进行处理，因此并不想去修改它的 `Cargo.toml`。此时可以使用 `Cargo` 提供的路径覆盖方法: **注意，这个方法限制较多，如果可以，还是要使用 [patch]**。

与 `[patch]` 修改 `Cargo.toml` 不同，路径覆盖修改的是 `Cargo` 自身的[配置文件](#) `$Home/.cargo/config.toml`:

```
paths = ["/path/to/uuid"]
```

`paths` 数组中的元素是一个包含 `Cargo.toml` 的目录(依赖包)，在当前例子中，由于我们只有一个 `uuid`，因此只需要覆盖它即可。目标路径可以是相对的，也是绝对的，需要注意，如果是相对路径，那是相对包含 `.cargo` 的 `$Home` 来说的。

## 不推荐的[replace]

---

`[replace]` 已经被标记为 `deprecated`，并将在未来被移除，请使用 `[patch]` 替代

---

虽然不建议使用，但是如果大家阅读其它项目时依然可能会碰到这种用法:

```
[replace]
"foo:0.1.0" = { git = 'https://github.com/example/foo' }
"bar:1.0.2" = { path = 'my/local/bar' }
```

语法看上去还是很清晰的，`[replace]` 中的每一个 `key` 都是 `Package ID` 格式，通过这种写法可以在依赖图中任意挑选一个节点进行覆盖。

# Cargo.toml 格式讲解

`Cargo.toml` 又被称为清单( `manifest` ), 文件格式是 TOML , 每一个清单文件都由以下部分组成:

- `cargo-features` — 只能用于 `nightly` 版本的 `feature`
- `[package]` — 定义项目( `package` )的元信息
  - `name` — 名称
  - `version` — 版本
  - `authors` — 开发作者
  - `edition` — Rust edition.
  - `rust-version` — 支持的最小化 Rust 版本
  - `description` — 描述
  - `documentation` — 文档 URL
  - `readme` — README 文件的路径
  - `homepage` - 主页 URL
  - `repository` — 源代码仓库的 URL
  - `license` — 开源协议 License.
  - `license-file` — License 文件的路径.
  - `keywords` — 项目的关键词
  - `categories` — 项目分类
  - `workspace` — 工作空间 workspace 的路径
  - `build` — 构建脚本的路径
  - `links` — 本地链接库的名称
  - `exclude` — 发布时排除的文件
  - `include` — 发布时包含的文件
  - `publish` — 用于阻止项目的发布
  - `metadata` — 额外的配置信息, 用于提供给外部工具
  - `default-run` — [ `cargo run` ] 所使用的默认可执行文件( `binary` )
  - `autobins` — 禁止可执行文件的自动发现
  - `autoexamples` — 禁止示例文件的自动发现
  - `autotests` — 禁止测试文件的自动发现
  - `autobenches` — 禁止 bench 文件的自动发现
  - `resolver` — 设置依赖解析器( dependency resolver)
- Cargo Target 列表: (查看 [Target 配置](#) 获取详细设置)
  - `[lib]` — Library target 设置.
  - `[[bin]]` — Binary target 设置.
  - `[[example]]` — Example target 设置.
  - `[[test]]` — Test target 设置.

- `[[bench]]` — Benchmark target 设置.
- Dependency tables:
  - `[dependencies]` — 项目依赖包
  - `[dev-dependencies]` — 用于 examples、tests 和 benchmarks 的依赖包
  - `[build-dependencies]` — 用于构建脚本的依赖包
  - `[target]` — 平台特定的依赖包
- `[badges]` — 用于在注册服务(例如 crates.io ) 上显示项目的一些状态信息，例如当前的维护状态：活跃中、寻找维护者、deprecated
- `[features]` — features 可以用于条件编译
- `[patch]` — 推荐使用的依赖覆盖方式
- `[replace]` — 不推荐使用的依赖覆盖方式 (deprecated).
- `[profile]` — 编译器设置和优化
- `[workspace]` — 工作空间的定义

下面，我们将对其中一些部分进行详细讲解。

## [package]

`Cargo.toml` 中第一个部分就是 `package`，用于设置项目的相关信息：

```
[package]
name = "hello_world" # the name of the package
version = "0.1.0"      # the current version, obeying semver
authors = ["Alice <a@example.com>", "Bob <b@example.com>"]
```

其中，只有 `name` 和 `version` 字段是**必须填写的**。当发布到注册服务时，可能会有额外的字段要求，具体参见[发布到 crates.io](#)。

### name

项目名用于引用一个项目(`package`)，它有几个用途：

- 其它项目引用我们的 `package` 时，会使用该 `name`
- 编译出的可执行文件(bin target)的默认名称

`name` 只能使用 `alphanumeric` 字符、`-` 和 `_`，并且不能为空。

事实上，`name` 的限制不止如此，例如：

- 当使用 `cargo new` 或 `cargo init` 创建时，`name` 还会被施加额外的限制，例如不能使用 Rust 关键字名称作为 `name`
- 如果要发布到 `crates.io`，那还有更多的限制：`name` 使用 ASCII 码，不能使用已经被使用的名称，例如 `uuid` 已经在 `crates.io` 上被使用，因此我们只能使用类如 `uuid_v1` 的名称，才能将项目发布到 `crates.io` 上

## version

Cargo 使用了[语义化版本控制](#)的概念，例如字符串 "0.1.12" 是一个 semver 格式的版本号，符合 "`x.y.z`" 的形式，其中 `x` 被称为主版本 `major`，`y` 被称为小版本 `minor`，而 `z` 被称为补丁 `patch`，可以看出从左到右，版本的影响范围逐步降低，补丁的更新是无关痛痒的，并不会造成 API 的兼容性被破坏。

使用该规则，你还需要遵循一些基本规则：

- 使用标准的 `x.y.z` 形式的版本号，例如 `1.0.0` 而不是 `1.0`
- 在版本到达 `1.0.0` 之前，怎么都行，但是如果有关破环性变更( breaking changes )，需要增加 `minor` 版本号。例如，为结构体新增字段或为枚举新增成员就是一种破环性变更
- 在 `1.0.0` 之后，如果发生破环性变更，需要增加 `major` 版本号
- 在 `1.0.0` 之后不要去破环构建流程
- 在 `1.0.0` 之后，不要在 `patch` 更新中添加新的 `api` ( `pub` 声明)，如果要添加新的 `pub` 结构体、特征、类型、函数、方法等对象时，增加 `minor` 版本号

如果大家想知道 Rust 如何使用版本号来解析依赖，可以查看[这里](#)。同时 [SemVer 兼容性](#) 提供了更为详尽的破环性变更列表。

## authors

```
[package]
authors = ["Sunfei <contact@im.dev>"]
```

该字段仅用于项目的元信息描述和 `build.rs` 用到的 `CARGO_PKG_AUTHORS` 环境变量，它并不会显示在 `crates.io` 界面上。

---

警告：清单中的 `[package]` 部分一旦发布到 `crates.io` 就无法进行更改，因此对于已发布的包来说，`authors` 字段是无法修改的

---

## edition

可选字段，用于指定项目所使用的 Rust Edition。

该配置将影响项目中的所有 Cargo Target 和包，前者包含测试用例、benchmark、可执行文件、示例等。

```
[package]
# ...
edition = '2021'
```

大多数时候，我们都无需手动指定，因为 cargo new 的时候，会自动帮我们添加。若 edition 配置不存在，那 2015 Edition 会被默认使用。

## rust-version

可选字段，用于说明你的项目支持的最低 Rust 版本(编译器能顺利完成编译)。一旦你使用的 Rust 版本比这个字段设置的要低，Cargo 就会报错，然后告诉用户所需的最低版本。

该字段是在 Rust 1.56 引入的，若大家使用的 Rust 版本低于该版本，则该字段会被自动忽略时。

```
[package]
# ...
edition = '2021'
rust-version = "1.56"
```

还有一点，rust-version 必须比第一个引入 edition 的 Rust 版本要新。例如 Rust Edition 2021 是在 Rust 1.56 版本引入的，若你使用了 edition = '2021' 的 [package] 配置，则指定的 rust version 字段必须要大于等于 1.56 版本。

还可以使用 --ignore-rust-version 命令行参数来忽略 rust-version。

该字段将影响项目中的所有 Cargo Target 和包，前者包含测试用例、benchmark、可执行文件、示例等。

## description

该字段是项目的简介，crates.io 会在项目首页使用该字段包含的内容，**不支持 Markdown 格式**。

```
[package]
# ...
description = "A short description of my package"
```

---

注意: 若发布 `crates.io` , 则该字段是必须的

---

## documentation

该字段用于说明项目文档的地址, 若没有设置, `crates.io` 会自动链接到 `docs.rs` 上的相应页面。

```
[package]
# ...
documentation = "https://docs.rs/bitflags"
```

## readme

`readme` 字段指向项目的 `README.md` 文件, 该文件应该存在项目的根目录下(跟 `Cargo.toml` 同级), 用于向用户描述项目的详细信息, 支持 `Markdown` 格式。大家看到的 `crates.io` 上的项目首页就是基于该文件的内容进行渲染的。

```
[package]
# ...
readme = "README.md"
```

若该字段未设置且项目根目录下存在 `README.md`、`README.txt` 或 `README` 文件, 则该文件的名称将被默认使用。

你也可以通过将 `readme` 设置为 `false` 来禁止该功能, 若设置为 `true` , 则默认值 `README.md` 将被使用。

## homepage

该字段用于设置项目主页的 URL:

```
[package]
# ...
homepage = "https://serde.rs/"
```

## repository

设置项目的源代码仓库地址，例如 GitHub 链接:

```
[package]
# ...
repository = "https://github.com/rust-lang/cargo/"
```

## license 和 license-file

license 字段用于描述项目所遵循的开源协议。而 license-file 则用于指定包含开源协议的文件所在的路径(相对于 Cargo.toml)。

如果要发布到 crates.io，则该协议必须是 [SPDX2.1 协议表达式](#)。同时 license 名称必须是来自于 [SPDX 协议列表 3.11](#)。

SPDX 只支持使用 AND 、 OR 来组合多个开源协议:

```
[package]
# ...
license = "MIT OR Apache-2.0"
```

OR 代表用户可以任选一个协议进行遵循，而 AND 表示用户必须要同时遵循两个协议。还可以通过 WITH 来在指定协议之外添加额外的要求:

- MIT OR Apache-2.0
- LGPL-2.1-only AND MIT AND BSD-2-Clause
- GPL-2.0-or-later WITH Bison-exception-2.2

若项目使用了非标准的协议，你可以通过指定 license-file 字段来替代 license 的使用:

```
[package]
# ...
license-file = "LICENSE.txt"
```

---

注意: crates.io 要求必须设置 license 或 license-file

---

## keywords

该字段使用字符串数组的方式来指定项目的关键字列表，当用户在 crates.io 上搜索时，这些关键字可以提供索引的功能。

```
[package]
# ...
keywords = ["gamedev", "graphics"]
```

---

注意: crates.io 最多只支持 5 个关键字, 每个关键字都必须是合法的 ASCII 文本, 且需要使用字母作为开头, 只能包含字母、数字、\_ 和 - , 最多支持 20 个字符长度

---

## categories

categories 用于描述项目所属的类别:

```
categories = ["command-line-utilities", "development-tools::cargo-plugins"]
```

---

注意: crates.io 最多只支持 5 个类别, 目前不支持用户随意自定义类别, 你所使用的类别需要跟 [https://crates.io/category\\_slugs](https://crates.io/category_slugs) 上的类别精准匹配。

---

## workspace

该字段用于配置当前项目所属的工作空间。

若没有设置, 则将沿着文件目录向上寻找, 直至找到第一个设置了 [workspace] 的 Cargo.toml。因此, 当一个成员不在工作空间的子目录时, 设置该字段将非常有用。

```
[package]
# ...
workspace = "path/to/workspace/root"
```

需要注意的是 Cargo.toml 清单还有一个 [workspace] 部分专门用于设置工作空间, 若它被设置了, 则 package 中的 workspace 字段将无法被指定。这是因为一个包无法同时满足两个角色:

- 该包是工作空间的根包(root crate), 通过 [workspace] 指定)
- 该包是另一个工作空间的成员, 通过 package.workspace 指定

若要了解工作空间的更多信息, 请参见[这里](#)。

## build

build 用于指定位于项目根目录中的构建脚本, 关于构建脚本的更多信息, 可以阅读 [构建脚本一章](#)。

```
[package]
# ...
build = "build.rs"
```

还可以使用 `build = false` 来禁止构建脚本的自动检测。

## links

用于指定项目链接的本地库的名称，更多的信息请看构建脚本章节的 [links](#)

```
[package]
# ...
links = "foo"
```

## exclude 和 include

这两个字段可以用于显式地指定想要包含在外或在内的文件列表，往往用于发布到注册服务时。你可以使用 `cargo package --list` 来检查哪些文件被包含在项目中。

```
[package]
# ...
exclude = ["/ci", "images/", ".*"]

[package]
# ...
include = ["/src", "COPYRIGHT", "/examples", "!/examples/big_example"]
```

尽管大家可能没有指定 `include` 或 `exclude`，但是任然会有些规则自动被应用，一起来看看。

若 `include` 没有被指定，则以下文件将被排除在外：

- 项目不是 git 仓库，则所有以 `.` 开头的隐藏文件会被排除
- 项目是 git 仓库，通过 `.gitignore` 配置的文件会被排除

无论 `include` 或 `exclude` 是否被指定，以下文件都会被排除在外：

- 任何包含 `Cargo.toml` 的子目录会被排除
- 根目录下的 `target` 目录会被排除

以下文件会永远被 `include`，你无需显式地指定：

- `Cargo.toml`
- 若项目包含可执行文件或示例代码，则最小化的 `Cargo.lock` 会自动被包含

- `license-file` 指定的协议文件
- 

这两个字段很强大，但是对于生产实践而言，我们还是推荐通过 `.gitignore` 来控制，因为这样协作者更容易看懂。如果大家希望更深入的了解 `include/exclude`，可以参考下官方的 [Cargo 文档](#)

---

## publish

该字段常常用于防止项目因为失误被发布到 `crates.io` 等注册服务上，例如如果希望项目在公司内部私有化，你应该设置：

```
[package]
# ...
publish = false
```

也可以通过字符串数组的方式来指定允许发布到的注册服务名称：

```
[package]
# ...
publish = ["some-registry-name"]
```

若 `publish` 数组中包含了一个注册服务名称，则 `cargo publish` 命令会使用该注册服务，除非你通过 `--registry` 来设定额外的规则。

## metadata

Cargo 默认情况下会对 `Cargo.toml` 中未使用的 `key` 进行警告，以帮助大家提前发现风险。但是 `package.metadata` 并不在其中，因为它是由用户自定义的提供给外部工具的配置文件。例如：

```
[package]
name = "..."
# ...

# 以下配置元数据可以在生成安卓 APK 时使用
[package.metadata.android]
package-name = "my-awesome-android-app"
assets = "path/to/static"
```

与其相似的还有 `[workspace.metadata]`，都可以作为外部工具的配置信息来使用。

## default-run

当大家使用 `cargo run` 来运行项目时，该命令会使用默认的二进制可执行文件作为程序启动入口。

我们可以通过 `default-run` 来修改默认的入口，例如现在有两个二进制文件 `src/bin/a.rs` 和 `src/bin/b.rs`，通过以下配置可以将入口设置为前者：

```
[package]
default-run = "a"
```

## [badges]

该部分用于指定项目当前的状态，该状态会展示在 `crates.io` 的项目主页中，例如以下配置可以设置项目的维护状态：

```
[badges]
# `maintenance` 是项目的当前维护状态，它可能会被其它注册服务所使用，但是目前还没有被 `crates.io` 使用： https://github.com/rust-lang/crates.io/issues/2437
#
# `status` 字段时必须的，以下是可用的选项：
# - `actively-developed`：新特性正在积极添加中，bug 在持续修复中
# - `passively-maintained`：目前没有计划去支持新的特性，但是项目维护者可能会回答你提出的 issue
# - `as-is`：该项目的功能已经完结，维护者不准备继续开发和提供支持了，但是它的功能已经达到了预期
# - `experimental`：作者希望同大家分享，但是还不准备满足任何人的特殊要求
# - `looking-for-maintainer`：当前维护者希望将项目转移给新的维护者
# - `deprecated`：不再推荐使用该项目，需要说明原因以及推荐的替代项目
# - `none`： 不显示任何 badge，因此维护者没有说明他们的状态，用户需要自己去调查发生了什么
maintenance = { status = "..." }
```

## [dependencies]

在[之前章节](#)中，我们已经详细介绍过 `[dependencies]`、`[dev-dependencies]` 和 `[build-dependencies]`，这里就不再赘述。

## [profile.\*]

该部分可以对编译器进行配置，例如 `debug` 和 `optimize`，在后续的[编译器优化](#)章节有详细介绍。

# Cargo Target

**Cargo 项目中包含有一些对象，它们包含的源代码文件可以被编译成相应的包，这些对象被称之为 Cargo Target。** 例如[之前章节](#)提到的库对象 `Library`、二进制对象 `Binary`、示例对象 `Examples`、测试对象 `Tests` 和基准性能对象 `Benches` 都是 Cargo Target。

本章节我们一起来看看该如何在 `Cargo.toml` 清单中配置这些对象，当然，大部分时候都无需手动配置，因为默认的配置通常由项目目录的布局自动推断出来。

## 对象介绍

在开始讲解如何配置对象前，我们先来看看这些对象究竟是什么，估计还有些同学对此有些迷糊 :)

### 库对象(Library)

库对象用于定义一个库，该库可以被其它的库或者可执行文件所链接。该对象包含的默认文件名是 `src/lib.rs`，且默认情况下，库对象的名称[跟项目名是一致的](#)，

一个工程只能有一个库对象，因此也只能有一个 `src/lib.rs` 文件，以下是一种自定义配置：

```
# 一个简单的例子：在 Cargo.toml 中定制化库对象
[lib]
crate-type = ["cdylib"]
bench = false
```

### 二进制对象(Binaries)

二进制对象在被编译后可以生成可执行的文件，默认的文件名是 `src/main.rs`，二进制对象的名称跟项目名也是相同的。

大家应该还记得，一个项目拥有多个二进制文件，因此一个项目可以拥有多个二进制对象。当拥有多个对象时，对象的文件默认会被放在 `src/bin/` 目录下。

二进制对象可以使用库对象提供的公共 API，也可以通过 `[dependencies]` 来引入外部的依赖库。

我们可以使用 `cargo run --bin <bin-name>` 的方式来运行指定的二进制对象，以下是二进制对象的配置示例：

```
# Example of customizing binaries in Cargo.toml.  
[[bin]]  
name = "cool-tool"  
test = false  
bench = false  
  
[[bin]]  
name = "frobnicator"  
required-features = ["frobinate"]
```

## 示例对象(Examples)

示例对象的文件在根目录下的 `examples` 目录中。既然是示例，自然是使用项目中的库对象的功能进行演示。示例对象编译后的文件会存储在 `target/debug/examples` 目录下。

如上所示，示例对象可以使用库对象的公共 API，也可以通过 `[dependencies]` 来引入外部的依赖库。

默认情况下，示例对象都是可执行的二进制文件(带有 `fn main()` 函数入口)，毕竟例子是用来测试和演示我们的库对象，是用来运行的。而你完全可以将示例对象改成库的类型：

```
[[example]]  
name = "foo"  
crate-type = ["staticlib"]
```

如果想要指定运行某个示例对象，可以使用 `cargo run --example <example-name>` 命令。如果是库类型的示例对象，则可以使用 `cargo build --example <example-name>` 进行构建。

与此类似，还可以使用 `cargo install --example <example-name>` 来将示例对象编译出的可执行文件安装到默认的目录中，将该目录添加到 `$PATH` 环境变量中，就可以直接全局运行安装的可执行文件。

最后，`cargo test` 命令默认会对示例对象进行编译，以防止示例代码因为长久没运行，导致严重过期以至于无法运行。

## 测试对象(Tests)

测试对象的文件位于根目录下的 `tests` 目录中，如果大家还有印象的话，就知道该目录是[集成测试](#)所使用的。

当运行 `cargo test` 时，里面的每个文件都会被编译成独立的包，然后被执行。

测试对象可以使用库对象提供的公共 API，也可以通过 `[dependencies]` 来引入外部的依赖库。

## 基准性能对象(Benches)

该对象的文件位于 `benches` 目录下，可以通过 `cargo bench` 命令来运行，关于基准测试，可以通过[这篇文章](#)了解更多。

## 配置一个对象

我们可以通过 `Cargo.toml` 中的 `[lib]`、`[[bin]]`、`[[example]]`、`[[test]]` 和 `[[bench]]` 部分对以上对象进行配置。

大家可能会疑惑 `[lib]` 和 `[[bin]]` 的写法为何不一致，原因是这种语法是 TOML 提供的[数组特性](#)，`[[bin]]` 这种写法意味着我们可以在 `Cargo.toml` 中创建多个 `[[bin]]`，每一个对应一个二进制文件

上文提到过，我们只能指定一个库对象，因此这里只能使用 `[lib]` 形式

由于它们的配置内容都是相似的，因此我们以 `[lib]` 为例来说明相应的配置项：

```
[lib]
name = "foo"          # 对象名称：库对象、`src/main.rs` 二进制对象的名称默认是项目名
path = "src/lib.rs"    # 对象的源文件路径
test = true            # 能否被测试，默认是 true
doctest = true         # 文档测试是否开启，默认是 true
bench = true           # 基准测试是否开启
doc = true             # 文档功能是否开启
plugin = false          # 是否可以用于编译器插件(deprecated).
proc-macro = false      # 是否是过程宏类型的库
harness = true          # 是否使用libtest harness : https://doc.rust-
lang.org/stable/rustc/tests/index.html
edition = "2015"        # 对象使用的 Rust Edition
crate-type = ["lib"]     # 生成的包类型
required-features = []  # 构建对象所需的 Cargo Features (N/A for lib).
```

### name

对于库对象和默认的二进制对象(`src/main.rs`)，默认的名称是项目的名称(`package.name`)。

对于其它类型的对象，默认是目录或文件名。

除了 `[lib]` 外，`name` 字段对于其他对象都是必须的。

## **proc-macro**

该字段的使用方式在[过程宏章节](#)有详细的介绍。

## **edition**

对使用的 Rust Edition 版本进行设置。

如果没有设置，则默认使用 `[package]` 中配置的 `package.edition`，通常来说，这个字段不应该被单独设置，只有在一些特殊场景中才可能用到：例如将一个大型项目逐步升级为新的 edition 版本。

## **crate-type**

该字段定义了对象生成的[包类型](#)。它是一个数组，因此为同一个对象指定多个包类型。

需要注意的是，只有库对象和示例对象可以被指定，因为其他的二进制、测试和基准测试对象只能是 `bin` 这个包类型。

默认的包类型如下：

对象	包类型
正常的库对象	"lib"
过程宏的库对象	"proc-macro"
示例对象	"bin"

可用的选项包括 `bin`、`lib`、`rlib`、`dylib`、`cdylib`、`staticlib` 和 `proc-macro`，如果大家想了解更多，可以看下官方的[参考手册](#)。

## **required-features**

该字段用于指定在构建对象时所需的 `features` 列表。

该字段只对 `[[bin]]`、`[[bench]]`、`[[test]]` 和 `[[example]]` 有效，对于 `[lib]` 没有任何效果。

```
[features]
# ...
postgres = []
sqlite = []
tools = []

[[bin]]
name = "my-pg-tool"
required-features = ["postgres", "tools"]
```

## 对象自动发现

默认情况下，Cargo 会基于项目的[目录文件布局](#)自动发现和确定对象，而之前的配置项则允许我们对其进行手动的配置修改(若项目布局跟标准的不一样时)。

而这种自动发现对象的设定可以通过以下配置来禁用：

```
[package]
# ...
autobins = false
autoexamples = false
autotests = false
autobenches = false
```

只有在特定场景下才应该禁用自动对象发现。例如，你有一个模块想要命名为 `bin`，目录结构如下：

```
└── Cargo.toml
    └── src
        └── lib.rs
            └── bin
                └── mod.rs
```

这在默认情况下会导致问题，因为 Cargo 会使用 `src/bin` 作为存放二进制对象的地方。

为了阻止这一点，可以设置 `autobins = false`：

```
└── Cargo.toml
    └── src
        └── lib.rs
            └── bin
                └── mod.rs
```

# 工作空间 Workspace

一个工作空间是由多个 package 组成的集合，它们共享同一个 Cargo.lock 文件、输出目录和一些设置(例如 profiles : 编译器设置和优化)。组成工作空间的 packages 被称之为工作空间的成员。

## 工作空间的两种类型

工作空间有两种类型： root package 和虚拟清单( virtual manifest )。

### 根 package

若一个 package 的 Cargo.toml 包含了 [package] 的同时又包含了 [workspace] 部分，则该 package 被称为工作空间的根 package 。

换而言之，一个工作空间的根( root )是该工作空间的 Cargo.toml 文件所在的目录。

举个例子，我们现在有多个 package ，它们的目录是嵌套关系，然后我们在最外层的 package ，也就是最外层目录中的 Cargo.toml 中定义一个 [workspace] ，此时这个最外层的 package 就是工作空间的根。

再举个例子，大名鼎鼎的 ripgrep 就在最外层的 package 中定义了 [workspace] :

```
[workspace]
members = [
    "crates/globset",
    "crates/grep",
    "crates/cli",
    "crates/matcher",
    "crates/pcre2",
    "crates/printer",
    "crates/regex",
    "crates/searcher",
    "crates/ignore",
]
```

那么最外层的目录就是 ripgrep 的工作空间的根。

### 虚拟清单

若一个 Cargo.toml 有 [workspace] 但是没有 [package] 部分，则它是虚拟清单类型的工作空间。

对于没有主 package 的场景或你希望将所有的 package 组织在单独的目录中时，这种方式就非常适合。

例如 [rust-analyzer](#) 就是这样的项目，它的根目录中的 `Cargo.toml` 中并没有 `[package]`，说明该根目录不是一个 package，但是却有 `[workspace]`：

```
[workspace]
members = ["xtask/", "lib/*", "crates/*"]
exclude = ["crates/proc_macro_test/imp"]
```

结合 `rust-analyzer` 的目录布局可以看出，该工作空间的所有成员 package 都在单独的目录中，因此这种方式很适合虚拟清单的工作空间。

## 关键特性

工作空间的几个关键点在于：

- 所有的 package 共享同一个 `Cargo.lock` 文件，该文件位于工作空间的根目录中
- 所有的 package 共享同一个[输出目录](#)，该目录默认的名称是 `target`，位于工作空间根目录下
- 只有工作空间根目录的 `Cargo.toml` 才能包含 `[patch]`, `[replace]` 和 `[profile.*]`，而成员的 `Cargo.toml` 中的相应部分将被自动忽略

## [workspace]

`Cargo.toml` 中的 `[workspace]` 部分用于定义哪些 packages 属于工作空间的成员：

```
[workspace]
members = ["member1", "path/to/member2", "crates/*"]
exclude = ["crates/foo", "path/to/other"]
```

若某个本地依赖包是通过 `path` 引入，且该包位于工作空间的目录中，则该包自动成为工作空间的成员。

剩余的成员需要通过 `workspace.members` 来指定，里面包含了各个成员所在的目录(成员目录中包含了 `Cargo.toml` )。

`members` 还支持使用 `glob` 来匹配多个路径，例如上面的例子中使用 `crates/*` 匹配 `crates` 目录下的所有包。

`exclude` 可以将指定的目录排除在工作空间之外，例如还是上面的例子，`crates/*` 在包含了 `crates` 目录下的所有包后，又通过 `exclude` 中 `crates/foo` 将 `crates` 下的 `foo` 目录排除在外。

你也可以将一个空的 `[workspace]` 直接联合 `[package]` 使用，例如：

```
[package]
name = "hello"
version = "0.1.0"

[workspace]
```

此时的工作空间的成员包含：

- 根 `package : "hello"`
- 所有通过 `path` 引入的本地依赖(位于工作空间目录下)

## 选择工作空间

选择工作空间有两种方式：Cargo 自动查找、手动指定 `package.workspace` 字段。

当位于工作空间的子目录中时，Cargo 会自动在该目录的父目录中寻找带有 `[workspace]` 定义的 `Cargo.toml`，然后再决定使用哪个工作空间。

我们还可以使用下面的方法来覆盖 Cargo 自动查找功能：将成员包中的 `package.workspace` 字段修改为工作区间根目录的位置，这样就能显式地让一个成员使用指定的工作空间。

当成员不在工作空间的子目录下时，这种手动选择工作空间的方法就非常适用。毕竟 Cargo 的自动搜索是沿着父目录往上查找，而成员并不在工作空间的子目录下，这意味着顺着成员的父目录往上找是无法找到该工作空间的 `Cargo.toml` 的，此时就只能手动指定了。

## 选择 package

在工作空间中，`package` 相关的 Cargo 命令(例如 `cargo build`)可以使用 `-p`、`--package` 或 `--workspace` 命令行参数来指定想要操作的 `package`。

若没有指定任何参数，则 Cargo 将使用当前工作目录的中的 `package`。若工作目录是虚拟清单类型的工作空间，则该命令将作用在所有成员上(就好像是使用了 `--workspace` 命令行参数)。而 `default-members` 可以在命令行参数没有被提供时，手动指定操作的成员：

```
[workspace]
members = ["path/to/member1", "path/to/member2", "path/to/member3/*"]
default-members = ["path/to/member2", "path/to/member3/foo"]
```

这样一来，`cargo build` 就不会应用到虚拟清单工作空间的所有成员，而是指定的成员上。

## workspace.metadata

与 `package.metadata` 非常类似，`workspace.metadata` 会被 Cargo 自动忽略，就算没有被使用也不会发出警告。

这个部分可以用于让工具在 `Cargo.toml` 中存储一些工作空间的配置元信息。例如：

```
[workspace]
members = ["member1", "member2"]

[workspace.metadata.webcontents]
root = "path/to/webproject"
tool = ["npm", "run", "build"]
# ...
```

# 条件编译 Features

Cargo Feature 是非常强大的机制，可以为大家提供[条件编译](#)和可选依赖的高级特性。

## [features]

Feature 可以通过 `Cargo.toml` 中的 `[features]` 部分来定义：其中每个 `feature` 通过列表的方式指定了它所能启用的其他 `feature` 或可选依赖。

假设我们有一个 2D 图像处理库，然后该库所支持的图片格式可以通过以下方式启用：

```
[features]
# 定义一个 feature : webp，但它并没有启用其它 feature
webp = []
```

当定义了 `webp` 后，我们就可以在代码中通过 [cfg 表达式](#)来进行条件编译。例如项目中的 `lib.rs` 可以使用以下代码对 `webp` 模块进行条件引入：

```
#[cfg(feature = "webp")]
pub mod webp;
```

`#[cfg(feature = "webp")]` 的含义是：只有在 `webp` feature 被定义后，以下的 `webp` 模块才能被引入进来。由于我们之前在 `[features]` 里定义了 `webp`，因此以上代码的 `webp` 模块会被成功引入。

在 `Cargo.toml` 中定义的 `feature` 会被 `Cargo` 通过命令行参数 `--cfg` 传给 `rustc`，最终由后者完成编译：`rustc --cfg ...`。若项目中的代码想要测试 `feature` 是否存在，可以使用 [cfg 属性](#)或 [cfg 宏](#)。

之前我们提到了一个 `feature` 还可以开启其他 `feature`，举个例子，例如 ICO 图片格式包含 BMP 和 PNG 格式，因此当 `ico` 被启用后，它还得确保启用 `bmp` 和 `png`：

```
[features]
bmp = []
png = []
ico = ["bmp", "png"]
webp = []
```

对此，我们可以理解为：`bmp` 和 `png` 是开启 `ico` 的先决条件（注：开启 `ico`，会自动开启 `bmp`，`png`）。

Feature 名称可以包含来自 [Unicode XID standard](#) 定义的字母，允许使用 \_ 或 0-9 的数字作为起始字符，在起始字符后，还可以使用 -、+ 或 .。

但是我们还是推荐按照 [crates.io](#) 的方式来设置 Feature 名称：crate.io 要求名称只能由 ASCII 字母数字、\_、- 或 + 组成。

## default feature

默认情况下，所有的 feature 都会被自动禁用，可以通过 default 来启用它们：

```
[features]
default = ["ico", "webp"]
bmp = []
png = []
ico = ["bmp", "png"]
webp = []
```

使用如上配置的项目被构建时，default feature 首先会被启用，然后它接着启用了 ico 和 webp feature，当然我们还可以关闭 default：

- --no-default-features 命令行参数可以禁用 default feature
- default-features = false 选项可以在依赖声明中指定

---

当你要去改变某个依赖库的 default 启用的 feature 列表时(例如觉得该库引入的 feature 过多，导致最终编译出的文件过大)，需要格外的小心，因为这可能会导致某些功能的缺失

## 可选依赖

当依赖被标记为 "可选 optional" 时，意味着它默认不会被编译。假设我们的 2D 图片处理库需要用到一个外部的包来处理 GIF 图片：

```
[dependencies]
gif = { version = "0.11.1", optional = true }
```

这种可选依赖的写法会自动定义一个与依赖同名的 feature，也就是 gif feature，这样一来，当我们启用 gif feature 时，该依赖库也会被自动引入并启用：例如通过 --feature gif 的方式启用 feature。

---

注意：目前来说，`[feature]` 中定义的 feature 还不能与已引入的依赖库同名。但是在 `nightly` 中已经提供了实验性的功能用于改变这一点：[namespaced features](#)

当然，**我们还可以通过显式定义 feature 的方式来启用这些可选依赖库**，例如为了支持 AVIF 图片格式，我们需要引入两个依赖包，由于 `avif` 是通过 `feature` 引入的可选格式，因此它依赖的两个包也必须声明为可选的：

```
[dependencies]
ravif = { version = "0.6.3", optional = true }
rgb = { version = "0.8.25", optional = true }

[features]
avif = ["ravif", "rgb"]
```

之后，`avif` feature 一旦被启用，那这两个依赖库也将自动被引入。

---

注意：我们之前也讲过条件引入依赖的方法，那就是使用[平台相关的依赖](#)，与基于 `feature` 的可选依赖不同，它们是基于特定平台的可选依赖

## 依赖库自身的 feature

就像我们的项目可以定义 `feature` 一样，依赖库也可以定义它自己的 `feature`，也有需要启用的 `feature` 列表，当引入该依赖库时，我们可以通过以下方式为其启用相关的 `features`：

```
[dependencies]
serde = { version = "1.0.118", features = ["derive"] }
```

以上配置为 `serde` 依赖开启了 `derive` feature，还可以通过 `default-features = false` 来禁用依赖库的 `default` feature：

```
[dependencies]
flate2 = { version = "1.0.3", default-features = false, features = ["zlib"] }
```

这里我们禁用了 `flate2` 的 `default` feature，但又手动为它启用了 `zlib` feature。

---

注意：这种方式未必能成功禁用 `default`，原因是可能会有其它依赖也引入了 `flate2`，并且没有对 `default` 进行禁用，那此时 `default` 依然会被启用。

查看下文的 [feature 同一化](#) 获取更多信息

---

除此之外，还能通过下面的方式来间接开启依赖库的 feature：

```
[dependencies]
jpeg-decoder = { version = "0.1.20", default-features = false }

[features]
# Enables parallel processing support by enabling the "rayon" feature of jpeg-
decoder.
parallel = ["jpeg-decoder/rayon"]
```

如上所示，我们定义了一个 parallel feature，同时为其启用了 jpeg-decoder 依赖的 rayon feature。

---

注意：上面的 "package-name/feature-name" 语法形式不仅会开启指定依赖的指定 feature，若该依赖是可选依赖，那还会自动将其引入

在 nightly 版本中，可以对这种行为进行禁用：[weak dependency features](#)

---

## 通过命令行参数启用 feature

以下的命令行参数可以启用指定的 feature：

- `--features FEATURES`：启用给出的 feature 列表，可以使用逗号或空格进行分隔，若你是在终端中使用，还需要加上双引号，例如 `--features "foo bar"`。若在工作空间中构建多个 package，可以使用 `package-name/feature-name` 为特定的成员启用 features
- `--all-features`：启用命令行上所选择的所有包的所有 features
- `--no-default-features`：对选择的包禁用 default feature

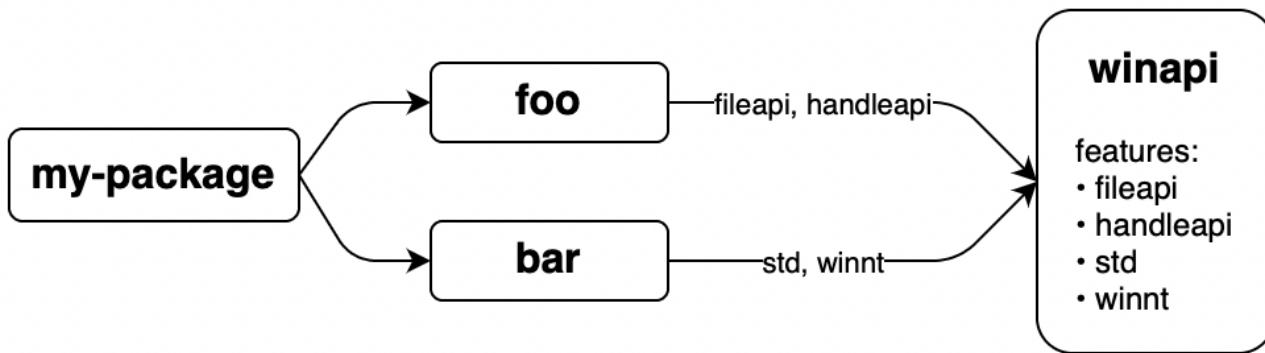
## feature 同一化

feature 只有在定义的包中才是唯一的，不同包之间的 feature 允许同名。因此，在一个包上启用 feature 不会导致另一个包的同名 feature 被误启用。

当一个依赖被多个包所使用时，这些包对该依赖所设置的 feature 将被进行合并，这样才能确保该依赖只有一个拷贝存在，这个过程就称之为同一化。大家可以查看[这里](#)了解下解析器如何对 feature 进行解

析处理。

这里，我们使用 `winapi` 为例来说明这个过程。首先，`winapi` 使用了大量的 `features`；然后我们有两个包 `foo` 和 `bar` 分别使用了它的两个 `features`，那么在合并后，最终 `winapi` 将同时启四个 `features`：



由于这种不可控性，我们需要让 `启用feature = 添加特性` 这个等式成立，换而言之，**启用一个 feature 不应该导致某个功能被禁止**。这样才能的让多个包启用同一个依赖的不同 `features`。

例如，如果我们想可选的支持 `no_std` 环境(不使用标准库)，那么有两种做法：

- 默认代码使用标准库的，当 `no_std` feature 启用时，禁用相关的标准库代码
- 默认代码使用非标准库的，当 `std` feature 启用时，才使用标准库的代码

前者就是功能削减，与之相对，后者是功能添加，根据之前的内容，我们应该选择后者的做法：

```
#![no_std]

#[cfg(feature = "std")]
extern crate std;

#[cfg(feature = "std")]
pub fn function_that_requires_std() {
    // ...
}
```

### 彼此互斥的 feature

某极少数情况下，`features` 之间可能会互相不兼容。我们应该避免这种设计，因为如果一旦这么设计了，那你可能需要修改依赖图的很多地方才能避免两个不兼容 `feature` 的同时启用。

如果实在没有办法，可以考虑增加一个编译错误来让报错更清晰：

```
#[cfg(all(feature = "foo", feature = "bar"))]
compile_error!("feature \"foo\" and feature \"bar\" cannot be enabled at the same
time");
```

当同时启用 `foo` 和 `bar` 时，编译器就会爆出一个更清晰的错误：feature `foo` 和 `bar` 无法同时启用。

总之，我们还是应该在设计上避免这种情况的发生，例如：

- 将某个功能分割到多个包中
- 当冲突时，设置 feature 优先级，`cfg-if` 包可以帮助我们写出更复杂的 `cfg` 表达式

## 检视已解析的 features

在复杂的依赖图中，如果想要了解不同的 features 是如何被多个包多启用的，这是相当困难的。好在 `cargo tree` 命令提供了几个选项可以帮组我们更好的检视哪些 features 被启用了：

`cargo tree -e features`，该命令以依赖图的方式来展示已启用的 features，包含了每个依赖包所启用的特性：

```
$ cargo tree -e features
test_cargo v0.1.0 (/Users/sunfei/development/rust/demos/test_cargo)
└── uuid feature "default"
    ├── uuid v0.8.2
    └── uuid feature "std"
        └── uuid v0.8.2
```

`cargo tree -f "{p} {f}"` 命令会提供一个更加紧凑的视图：

```
$ cargo tree -f "{p} {f}"
test_cargo v0.1.0 (/Users/sunfei/development/rust/demos/test_cargo)
└── uuid v0.8.2 default,std
```

`cargo tree -e features -i foo`，该命令会显示 features 会如何“流入”指定的包 `foo` 中：

```
$ cargo tree -e features -i uuid
uuid v0.8.2
└── uuid feature "default"
    └── test_cargo v0.1.0 (/Users/sunfei/development/rust/demos/test_cargo)
        └── test_cargo feature "default" (command-line)
└── uuid feature "std"
    └── uuid feature "default" (*)
```

该命令在依赖图较为复杂时非常有用，使用它可以让你了解某个依赖包上开启了哪些 `features` 以及其中的原因。

大家可以查看官方的 `cargo tree` 文档获取更加详细的使用信息。

## Feature 解析器 V2 版本

我们还能通过以下配置指定使用 V2 版本的解析器( [resolver](#) ):

```
[package]
name = "my-package"
version = "1.0.0"
resolver = "2"
```

V2 版本的解析器可以在某些情况下避免 feature 同一化的发生，具体的情况在[这里](#)有描述，下面做下简单的总结：

- 为特定平台开启的 `features` 且此时并没有被构建，会被忽略
- `build-dependencies` 和 `proc-macros` 不再跟普通的依赖共享 `features`
- `dev-dependencies` 的 `features` 不会被启用，除非正在构建的对象需要它们(例如测试对象、示例对象等)

对于部分场景而言，feature 同一化确实是需要避免的，例如，一个构建依赖开启了 `std` feature，而同一个依赖又被用于 `no_std` 环境，很明显，开启 `std` 将导致错误的发生。

说完优点，我们再来看看 V2 的缺点，其中增加编译构建时间就是其中之一，原因是同一个依赖会被构建多次(每个都拥有不同的 feature 列表)。

---

由于此部分内容可能只有极少数的用户需要，因此我们并没有对其进行扩展，如果大家希望了解更多关于 V2 的内容，可以查看[官方文档](#)

## 构建脚本

[构建脚本](#)可以通过 `CARGO_FEATURE_<name>` 环境变量获取启用的 `feature` 列表，其中 `<name>` 是 `feature` 的名称，该名称被转换成大全写字母，且 `-` 被转换为 `_`。

## required-features

该字段可以用于禁用特定的 Cargo Target: 当某个 feature 没有被启用时, 查看[这里](#)获取更多信息。

## SemVer 兼容性

启用一个 feature 不应该引入一个不兼容 SemVer 的改变。例如, 启用的 feature 不应该改变现有的 API, 因为这会给用户造成不兼容的破坏性变更。如果大家想知道哪些变化是兼容的, 可以参见[官方文档](#)。

总之, 在新增/移除 feature 或可选依赖时, 你需要小心, 因此这些可能会造成向后不兼容性。更多信息参见[这里](#), 简单总结如下:

- 在发布 `minor` 版本时, 以下通常是安全的:
  - 新增 feature 或可选依赖
  - 修改某个依赖的 features
- 在发布 `minor` 版本时, 以下操作应该避免:
  - 移除 feature 或可选依赖
  - 将现有的公有代码放在某个 feature 之后
  - 从 feature 列表中移除一个 feature

## feature 文档和发现

将你的项目支持的 feature 信息写入到文档中是非常好的选择:

- 我们可以通过在 `lib.rs` 的顶部添加[文档注释](#)的方式来实现。例如 `regex` 就是这么做的。
- 若项目拥有一个用户手册, 那也可以在那里添加说明, 例如 `serde.rs`。
- 若项目是二进制类型(可运行的应用服务, 包含 `fn main` 入口), 可以将说明放在 `README` 文件或其他文档中, 例如 `sccache`。

特别是对于不稳定的或者不该再被使用的 feature 而言, 它们更应该被放在文档中进行清晰的说明。

当构建发布到 `docs.rs` 上的文档时, 会使用 `Cargo.toml` 中的元数据来控制哪些 features 会被启用。查看[docs.rs 文档](#)获取更多信息。

## 如何发现 features

若依赖库的文档中对其使用的 `features` 做了详细描述，那你会更容易知道他们使用了哪些 `features` 以及该如何使用。

当依赖库的文档没有相关信息时，你也可以通过源码仓库的 `Cargo.toml` 文件来获取，但是有些时候，使用这种方式来跟踪并获取全部相关的信息是相当困难的。

# Features 示例

以下我们一起来看看一些来自真实世界的示例。

## 最小化构建时间和文件大小

如果一些包的部分特性不再启用，就可以减少该包占用的大小以及编译时间：

- `syn` 包可以用来解析 Rust 代码，由于它很受欢迎，大量的项目都在引用，因此它给出了[非常清晰的文档](#)关于如何最小化使用它包含的 `features`
- `regex` 也有关于 `features` 的[描述文档](#)，例如移除 Unicode 支持的 `feature` 可以降低最终生成可执行文件的大小
- `winapi` 拥有[众多 features](#)，这些 `feature` 对用了各种 Windows API，你可以只引入代码中用到的 API 所对应的 `feature`.

## 行为扩展

`serde_json` 拥有一个 `preserve_order feature`，可以用于在序列化时保留 JSON 键值对的顺序。同时，该 `feature` 还会启用一个可选依赖 `indexmap`。

当这么做时，一定要小心不要破坏了 SemVer 的版本兼容性，也就是说：启用 `feature` 后，代码依然要能正常工作。

## no\_std 支持

一些包希望能同时支持 `no_std` 和 `std` 环境，例如该包希望支持嵌入式系统或资源紧张的系统，且又希望能支持其它的平台，此时这种做法是非常有用的，因为标准库 `std` 会大幅增加编译出来的文件的大小，对于资源紧张的系统来说，`no_std` 才是最合适的。

`wasm-bindgen` 定义了一个 `std feature`，它是[默认启用的](#)。首先，在库的顶部，它[无条件的启用了 no\\_std 属性](#)，它可以确保 `std` 和 `std prelude` 不会自动引入到作用域中来。其次，在不同的地方([示例 1](#), [示例 2](#))，它通过 `#[cfg(feature = "std")]` 启用 `std feature` 来添加 `std` 标准库支持。

## 对依赖库的 features 进行再导出

从依赖库再导出 features 在有些场景中会相当有用，这样用户就可以通过依赖包的 features 来控制功能而不是自己去手动定义。

例如 `regex` 将 `regex_syntax` 包的 features 进行了再导出，这样 `regex` 的用户无需知道 `regex_syntax` 包，但是依然可以访问后者包含的 features。

## feature 优先级

一些包可能会拥有彼此互斥的 features(无法共存，上一章节中有讲到)，其中一个办法就是为 feature 定义优先级，这样其中一个就会优于另一个被启用。

例如 `log` 包，它有几个 features 可以用于在编译期选择最大的日志级别，这里，它就使用了 `cfg-if` 的方式来设置优先级。一旦多个 features 被启用，那更高优先级的就会优先被启用。

## 过程宏包

一些包拥有过程宏，这些宏必须定义在一个独立的包中。但是不是所有的用户都需要过程宏的，因此也无需引入该包。

在这种情况下，将过程宏所在的包定义为可选依赖，是很不错的选择。这样做还有一个好处：有时过程宏的版本必须要跟父包进行同步，但是我们又不希望所有的用户都进行同步。

其中一个例子就是 `serde`，它有一个 `derive` feature 可以启用 `serde_derive` 过程宏。由于 `serde_derive` 包跟 `serde` 的关系非常紧密，因此它使用了版本相同的需求来保证两者的版本同步性。

## 只能用于 nightly 的 feature

Rust 有些实验性的 API 或语言特性只能在 nightly 版本下使用，但某些使用了这些 API 的包并不想强制他们的用户也使用 nightly 版本，因此他们会通过 feature 的方式来控制。

若用户希望使用这些 API 时，需要启用相应的 feature，而这些 feature 只能在 nightly 下使用。若用户不需要使用这些 API，就无需开启相应的 feature，自然也不需要使用 nightly 版本。

例如 `rand` 包有一个 `simd_support` feature 就只能在 nightly 下使用，若我们不使用该 feature，则在 stable 下依然可以使用 `rand`。

## 实验性 feature

有一些包会提前将一些实验性的 API 放出去，既然是实验性的，自然无法保证其稳定性。在这种情况下，通常会在文档中将相应的 features 标记为实验性，意味着它们在未来可能会发生大的改变(甚至 minor 版本都可能发生)。

其中一个例子是 `async-std` 包，它拥有一个 `unstable feature`，用来标记一些新的 API，表示人们已经可以选择性的使用但是还没有准备好去依赖它。

# 发布配置 Profile

细心的同学可能发现了迄今为止我们已经为 Cargo 引入了不少新的名词，而且这些名词有一个共同的特点，不容易或不适合翻译成中文，因为难以表达的很准确，例如 Cargo Target, Feature 等，这不现在又多了一个 Profile。

## 默认的 profile

Profile 其实是一种发布配置，例如它默认包含四种: `dev`、`release`、`test` 和 `bench`，正常情况下，我们无需去指定，Cargo 会根据我们使用的命令来自动进行选择

- 例如 `cargo build` 自动选择 `dev` profile，而 `cargo test` 则是 `test` profile，出于历史原因，这两个 profile 输出的结果都存放在项目根目录下的 `target/debug` 目录中，结果往往用于开发/测试环境
- 而 `cargo build --release` 自动选择 `release` profile，并将输出结果存放在 `target/release` 目录中，结果往往用于生产环境

可以看出 Profile 跟 Nodejs 的 `dev` 和 `prod` 很像，都是通过不同的配置来为目标环境构建最终编译后的结果：`dev` 编译输出的结果用于开发环境，`prod` 则用于生产环境。

针对不同的 profile，编译器还会提供不同的优化级别，例如 `dev` 用于开发环境，因此构建速度是最重要的：此时，我们可以牺牲运行性能来换取编译性能，那么优化级别就会使用最低的。而 `release` 则相反，优化级别会使用最高，导致的结果就是运行得非常快，但是编译速度大幅降低。

---

初学者一个常见的错误，就是使用非 `release` profile 去测试性能，例如 `cargo run`，这种方式显然无法得到正确的结果，我们应该使用 `cargo run --release` 的方式测试性能

---

profile 可以通过 `Cargo.toml` 中的 `[profile]` 部分进行设置和改变：

```
[profile.dev]
opt-level = 1          # 使用稍高一些的优化级别，最低是0，最高是3
overflow-checks = false # 关闭整数溢出检查
```

需要注意的是，每一种 profile 都可以单独的进行设置，例如上面的 `[profile.dev]`。

如果是工作空间的话，只有根 package 的 `Cargo.toml` 中的 `[profile]` 设置才会被使用，其它成员或依赖包中的设置会被自动忽略。

另外，profile 还能在 Cargo 自身的配置文件中进行覆盖，总之，通过 `.cargo/config.toml` 或环境变量的方式所指定的 `profile` 配置会覆盖项目的 `Cargo.toml` 中相应的配置。

## 自定义 profile

除了默认的四种 profile，我们还可以定义自己的。对于大公司来说，这个可能会非常有用，自定义的 profile 可以帮助我们建立更灵活的工作发布流和构建模型。

当定义 profile 时，你必须指定 `inherits` 用于说明当配置缺失时，该 profile 要从哪个 profile 那里继承配置。

例如，我们想在 release profile 的基础上增加 LTO 优化，那么可以在 `Cargo.toml` 中添加如下内容：

```
[profile.release-lto]
inherits = "release"
lto = true
```

然后在构建时使用 `--profile` 来指定想要选择的自定义 profile：

```
$ cargo build --profile release-lto
```

与默认的 profile 相同，自定义 profile 的编译结果也存放在 `target/` 下的同名目录中，例如 `--profile release-lto` 的输出结果存储在 `target/release-lto` 中。

## 选择 profile

- 默认使用 `dev`：`cargo build`, `cargo rustc`, `cargo check`, 和 `cargo run`
- 默认使用 `test`: `cargo test`
- 默认使用 `bench`: `cargo bench`
- 默认使用 `release`: `cargo install`, `cargo build --release`, `cargo run --release`
- 使用自定义 profile: `cargo build --profile release-lto`

## profile 设置

下面我们来看看 profile 中可以进行哪些优化设置。

## **opt-level**

该字段用于控制 `-C opt-level` 标志的优化级别。更高的优化级别往往意味着运行更快的代码，但是也意味着更慢的编译速度。

同时，更高的编译级别甚至会造成编译代码的改变和再排列，这会为 debug 带来更高的复杂度。

`opt-level` 支持的选项包括：

- 0 : 无优化
- 1 : 基本优化
- 2 : 一些优化
- 3 : 全部优化
- "s": 优化输出的二进制文件的大小
- "z": 优化二进制文件大小，但也会关闭循环向量化

我们非常推荐你根据自己的需求来找到最适合的优化级别(例如，平衡运行和编译速度)。而且有一点值得注意，有的时候优化级别和性能的关系可能会出乎你的意料之外，例如 3 比 2 更慢，再比如 "s" 并没有让你的二进制文件变得更小。

而且随着 `rustc` 版本的更新，你之前的配置也可能要随之变化，总之，为项目的热点路径做好基准性能测试是不错的选择，不然总不能每次都手动重写代码来测试吧：)

如果想要了解更多，可以参考 [rustc 文档](#)，这里有更高级的优化技巧。

## **debug**

`debug` 控制 `-C debuginfo` 标志，而后者用于控制最终二进制文件输出的 `debug` 信息量。

支持的选项包括：

- 0 或 `false` : 不输出任何 `debug` 信息
- 1 : 行信息
- 2 : 完整的 `debug` 信息

## **split-debuginfo**

`split-debuginfo` 控制 `-C split-debuginfo` 标志，用于决定输出的 `debug` 信息是存放在二进制可执行文件里还是邻近的文件中。

## debug-assertions

该字段控制 `-C debug-assertions` 标志，可以开启或关闭其中一个条件编译选项：

```
cfg(debug_assertions) .
```

`debug-assertion` 会提供运行时的检查，该检查只能用于 `debug` 模式，原因是对于 `release` 来说，这种检查的成本较为高昂。

大家熟悉的 `debug_assert!` 宏也是通过该标志开启的。

支持的选项包括：

- `true`: 开启
- `false`: 关闭

## overflow-checks

用于控制 `-C overflow-checks` 标志，该标志可以控制运行时的整数溢出行为。**当开启后，整数溢出会导致 panic。**

支持的选项包括：

- `true`: 开启
- `false`: 关闭

## lto

`lto` 用于控制 `-C lto` 标志，而后者可以控制 LLVM 的链接时优化(`link time optimizations`)。通过对整个程序进行分析，并以增加链接时间为代价，LTO 可以生成更加优化的代码。

支持的选项包括：

- `false`: 只会对代码生成单元中的本地包进行 "thin" LTO 优化，若代码生成单元数为 1 或者 `opt-level` 为 0，则不会进行任何 LTO 优化
- `true` 或 "fat" : 对依赖图中的所有包进行 "fat" LTO 优化
- "thin" : 对依赖图的所有包进行 "thin" LTO，相比 "fat" 来说，它仅牺牲了一点性能，但是换来了链接时间的可观减少
- `off` : 禁用 LTO

如果大家想了解跨语言 LTO，可以看下 `-C linker-plugin-lto` 标志。

## panic

`panic` 控制 [-C panic](#) 标志，它可以控制 `panic` 策略的选择。

支持的选项包括：

- "unwind"：遇到 `panic` 后对栈进行展开( `unwind` )
- "abort"：遇到 `panic` 后直接停止程序

当设置为 "unwind" 时，具体的栈展开信息取决于特定的平台，例如 NVPTX 不支持 `unwind`，因此程序只能 "abort"。

测试、基准性能测试、构建脚本和过程宏会忽略 `panic` 设置，目前来说它们要求是 "unwind"，如果大家希望修改成 "abort"，可以看看 [panic-abort-tests](#)。

另外，当你使用 "abort" 策略且在执行测试时，由于上述的要求，除了测试代码外，所有的依赖库也会忽略该 "abort" 设置而使用 "unwind" 策略。

## incremental

`incremental` 控制 [-C incremental](#) 标志，用于开启或关闭增量编译。开启增量编译时，`rustc` 会将必要的信息存放到硬盘中( `target` 目录中)，当下次编译时，这些信息可以被复用以改善编译时间。

支持的选项包括：

- `true`：启用
- `false`：关闭

**增量编译只能用于工作空间的成员和通过 `path` 引入的本地依赖。**

大家还可以通过环境变量 `CARGO_INCREMENTAL` 或 Cargo 配置 [build.incremental](#) 在全局对 `incremental` 进行覆盖。

## codegen-units

`codegen-units` 控制 [-C codegen-units](#) 标志，可以指定一个包会被分隔为多少个代码生成单元。**更多的代码生成单元会提升代码的并行编译速度，但是可能会降低运行速度。**

对于增量编译，默认值是 256，非增量编译是 16。

## r-path

用于控制 [-C rpath](#) 标志，可以控制 `rpath` 的启用与关闭。

`rpath` 代表硬编码到二进制可执行文件或库文件中的**运行时代码搜索(runtime search path)**, 动态链接库的加载器就通过它来搜索所需的库。

## 默认 profile

### dev

`dev` profile 往往用于开发和 debug, `cargo build` 或 `cargo run` 默认使用的就是 `dev` profile, `cargo build --debug` 也是。

---

注意: `dev` profile 的结果并没有输出到 `target/dev` 同名目录下, 而是 `target/debug` , 这是历史遗留问题

---

默认的 `dev` profile 设置如下:

```
[profile.dev]
opt-level = 0
debug = true
split-debuginfo = '...' # Platform-specific.
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'
incremental = true
codegen-units = 256
rpath = false
```

### release

`release` 往往用于预发/生产环境或性能测试, 以下命令使用的就是 `release` profile:

- `cargo build --release`
- `cargo run --release`
- `cargo install`

默认的 `release` profile 设置如下:

```
[profile.release]
opt-level = 3
debug = false
split-debuginfo = '...' # Platform-specific.
debug-assertions = false
overflow-checks = false
lto = false
panic = 'unwind'
incremental = false
codegen-units = 16
rpath = false
```

## test

该 profile 用于构建测试，它的设置是继承自 dev

## bench

bench profile 用于构建基准测试 benchmark，它的设计默认继承自 release

## 构建本身依赖

默认情况下，所有的 profile 都不会对构建过程本身所需的依赖进行优化，构建过程本身包括构建脚本、过程宏。

默认的设置是：

```
[profile.dev.build-override]
opt-level = 0
codegen-units = 256

[profile.release.build-override]
opt-level = 0
codegen-units = 256
```

如果是自定义 profile，那它会自动从当前正在使用的 profile 继承相应的设置，但不会修改。

## 重写 profile

我们还可以对特定的包使用的 profile 进行重写(override)：

```
# `foo` package 将使用 -Copt-level=3 标志。  
[profile.dev.package.foo]  
opt-level = 3
```

这里的 package 名称实际上是一个 [Package ID](#)，因此我们还可以通过版本号来选择：

```
[profile.dev.package."foo:2.1.0"]。
```

如果要为所有依赖包重写(不包括工作空间的成员)：

```
[profile.dev.package."*"]  
opt-level = 2
```

为构建脚本、过程宏和它们的依赖重写：

```
[profile.dev.build-override]  
opt-level = 3
```

---

注意：如果一个依赖同时被正常代码和构建脚本所使用，当 `--target` 没有指定时，Cargo 只会构建该依赖一次。

但是当使用了 `build-override` 后，该依赖会被构建两次，一次为正常代码，一次为构建脚本，因此会增加一些编译时间

---

重写的优先级按以下顺序执行(第一个匹配获胜)：

- `[profile.dev.package.name]`，指定名称进行重写
- `[profile.dev.package."*"]`，对所有非工作空间成员的 package 进行重写
- `[profile.dev.build-override]`，对构建脚本、过程宏及它们的依赖进行重写
- `[profile.dev]`
- Cargo 内置的默认值

重写无法使用 `panic`、`lto` 或 `rpath` 设置。

# 通过 config.toml 对 Cargo 进行配置

Cargo 相关的配置有两种，第一种是对自身进行配置，第二种是对指定的项目进行配置，关于后者请查看 [Cargo.toml 清单](#)。对于普通用户而言第二种才是我们最常使用的。

本文讲述的是如何对 Cargo 相关的工具进行配置，该配置中的部分内容可能会覆盖掉 `Cargo.toml` 中对应的部分，例如关于 `profile` 的内容。

## 层级结构

在前面我们已经见识过如何为 Cargo 进行全局配置：`$HOME/.cargo/config.toml`，事实上，还支持在一个 `package` 内对它进行配置。

总体原则是：Cargo 会顺着当前目录往上查找，直到找到目标配置文件。例如我们在目录 `/projects/foo/bar/baz` 下调用 Cargo 命令，那查找路径如下所示：

- `/projects/foo/bar/baz/.cargo/config.toml`
- `/projects/foo/bar/.cargo/config.toml`
- `/projects/foo/.cargo/config.toml`
- `/projects/.cargo/config.toml`
- `/.cargo/config.toml`
- `$CARGO_HOME/config.toml` 默认是：
  - Windows: `%USERPROFILE%\.cargo\config.toml`
  - Unix: `$HOME/.cargo/config.toml`

有了这种机制，我们既可以在全局中设置默认的配置，又可以每个包都设定独立的配置，甚至还能做版本控制。

如果一个 `key` 在多个配置中出现，那这些 `key` 只会保留一个：最靠近 Cargo 执行目录的配置文件中的 `key` 的值将被最终使用(因此，`$HOME` 下的都是最低优先级)。需要注意的是，如果 `key` 的值是数组，那相应的值将被合并(`join`)。

对于工作空间而言，Cargo 的搜索策略是从 root 开始，对于内部成员中包含的 `.cargo.toml` 会自动忽略。例如一个工作空间拥有两个成员，每个成员都有配置文件：

`/projects/foo/bar/baz/mylib/.cargo/config.toml` 和  
`/projects/foo/bar/baz/mybin/.cargo/config.toml`，但是 Cargo 并不会读取它们而是从工作空间的根(`/projects/foo/bar/baz/`)开始往上查找。

---

注意: Cargo 还支持没有 `.toml` 后缀的 `.cargo/config` 文件。对于 `.toml` 的支持是从 Rust 1.39 版本开始, 同时也是目前最推荐的方式。**但若同时存在有后缀和无后缀的文件, Cargo 将使用无后缀的!**

---

## 配置文件概览

下面是一个完整的配置文件, 并对**常用的选项**进行了翻译, 大家可以参考下:

```
paths = ["/path/to/override"] # 覆盖 `Cargo.toml` 中通过 path 引入的本地依赖

[alias]      # 命令别名
b = "build"
c = "check"
t = "test"
r = "run"
rr = "run --release"
space_example = ["run", "--release", "--", "\"command list\""]

[build]
jobs = 1                      # 并行构建任务的数量, 默认等于 CPU 的核心数
rustc = "rustc"                # rust 编译器
rustc-wrapper = "..."          # 使用该 wrapper 来替代 rustc
rustc-workspace-wrapper = "..." # 为工作空间的成员使用 该 wrapper 来替代 rustc
rustdoc = "rustdoc"             # 文档生成工具
target = "triple"               # 为 target triple 构建 ( `cargo install` 会忽略该选项)
target-dir = "target"           # 存放编译输出结果的目录
rustflags = ["...", "..."]      # 自定义flags, 会传递给所有的编译器命令调用
rustdocflags = ["...", "..."]    # 自定义flags, 传递给 rustdoc
incremental = true              # 是否开启增量编译
dep-info-basedir = "..."        # path for the base directory for targets in depfiles
pipelining = true               # rustc pipelining

[doc]
browser = "chromium"           # `cargo doc --open` 使用的浏览器,
                                # 可以通过 `BROWSER` 环境变量进行重写

[env]
# Set ENV_VAR_NAME=value for any process run by Cargo
ENV_VAR_NAME = "value"
# Set even if already present in environment
ENV_VAR_NAME_2 = { value = "value", force = true }
# Value is relative to .cargo directory containing `config.toml`, make absolute
ENV_VAR_NAME_3 = { value = "relative/path", relative = true }

[cargo-new]
vcs = "none"                   # 所使用的 VCS ('git', 'hg', 'pijul', 'fossil', 'none')

[http]
debug = false                   # HTTP debugging
proxy = "host:port"             # HTTP 代理, libcurl 格式
ssl-version = "tlsv1.3"          # TLS version to use
ssl-version.max = "tlsv1.3"       # 最高支持的 TLS 版本
ssl-version.min = "tlsv1.1"       # 最小支持的 TLS 版本
timeout = 30                     # HTTP 请求的超时时间, 秒
low-speed-limit = 10             # 网络超时阈值 (bytes/sec)
cainfo = "cert.pem"              # path to Certificate Authority (CA) bundle
check-revoke = true              # check for SSL certificate revocation
multiplexing = true              # HTTP/2 multiplexing
user-agent = "..."                # the user-agent header
```

```
[install]
root = "/some/path"          # `cargo install` 安装到的目标目录

[net]
retry = 2                    # 网络重试次数
git-fetch-with-cli = true    # 是否使用 `git` 命令来执行 git 操作
offline = true                # 不能访问网络

[patch.<registry>]
# Same keys as for [patch] in Cargo.toml

[profile.<name>]           # profile 配置, 详情见"如何在 Cargo.toml 中配置 profile" :
https://course.rs/cargo/reference/profiles.html#profile设置
opt-level = 0
debug = true
split-debuginfo = '...'
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'
incremental = true
codegen-units = 16
rpath = false
[profile.<name>.build-override]
[profile.<name>.package.<name>]

[registries.<name>] # 设置其它的注册服务: https://course.rs/cargo/reference/specify-
deps.html#从其它注册服务引入依赖包
index = "..."          # 注册服务索引列表的 URL
token = "..."          # 连接注册服务所需的鉴权 token

[registry]
default = "..."         # 默认的注册服务名称: crates.io
token = "..."

[source.<name>]        # 注册服务源和替换source definition and replacement
replace-with = "..."     # 使用给定的 source 来替换当前的 source, 例如使用科大源来替换crates.io源
以提升国内的下载速度: [source.crates-io] replace-with = 'ustc'
directory = "..."        # path to a directory source
registry = "..."          # 注册源的 URL , 例如科大源: [source.ustc] registry =
"git://mirrors.ustc.edu.cn/crates.io-index"
local-registry = "..."   # path to a local registry source
git = "..."              # URL of a git repository source
branch = "..."            # branch name for the git repository
tag = "..."               # tag name for the git repository
rev = "..."                # revision for the git repository

[target.<triple>]
linker = "..."            # linker to use
runner = "..."              # wrapper to run executables
rustflags = ["...", "..."] # custom flags for `rustc`
```

```
[target.<cfg>]
runner = "..."           # wrapper to run executables
rustflags = ["...", "..."] # custom flags for `rustc`

[target.<triple>.<links>] # `links` build script override
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = ["-L", "/some/path"]
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["..."]
metadata_key1 = "value"
metadata_key2 = "value"

[term]
verbose = false          # whether cargo provides verbose output
color = 'auto'            # whether cargo colorizes output
progress.when = 'auto'    # whether cargo shows progress bar
progress.width = 80        # width of progress bar
```

## 环境变量

除了 config.toml 配置文件，我们还可以使用环境变量的方式对 Cargo 进行配置。

配置文件中的 key foo.bar 对应的环境变量形式为 CARGO\_FOO\_BAR，其中的 .、 - 被转换成 \_，且字母都变成大写的。例如， target.x86\_64-unknown-linux-gnu.runner key 转换成环境变量后变成 CARGO\_TARGET\_X86\_64\_UNKNOWN\_LINUX\_GNU\_RUNNER。

就优先级而言，环境变量是比配置文件更高的。除了上面的机制，Cargo 还支持一些[预定义的环境变量](#)。

---

官方 Cargo Book 中本文的内容还有很多，但是剩余内容对于绝大多数用户都用不到，因此我们并没有涵盖其中。

---

# 发布到 crates.io

如果你想要把自己的开源项目分享给全世界，那最好的办法自然是 GitHub。但如果是 Rust 的库，那除了发布到 GitHub 外，我们还可以将其发布到 [crates.io](#) 上，然后其它用户就可以很简单的对其进行引用。

---

注意：发布包到 `crates.io` 后，特定的版本无法被覆盖，要发布就必须使用新的版本号，代码也无法被删除！

---

## 首次发布之前

**首先，我们需要一个账号：**访问 crates.io 的[主页](#)，然后在右上角使用 GitHub 账户登陆，接着访问你的[账户设置](#)页面，进入到 API Tokens 标签页下，生成新的 Token，并使用该 Token 在终端中进行登录：

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

该命令将告诉 Cargo 你的 API Token，然后将其存储在本地的 `~/.cargo/credentials.toml` 文件中。

---

注意：你需要妥善保管好 API Token，并且不要告诉任何人，一旦泄漏，请撤销( `Revoke` )并重新生成。

---

## 发布包之前

`crates.io` 上的**包名遵循先到先得**的方式：一旦你想要的包名已经被使用，那么你就得换一个不同的包名。

在发布之前，**确保** `Cargo.toml` 中以下字段已经被设置：

- `license` 或 `license-file`
- `description`
- `homepage`
- `documentation`
- `repository`

- `readme`

你还可以设置[关键字](#)和[类别](#)等元信息，让包更容易被其他人搜索发现，虽然它们不是必须的。

如果你发布的是一个依赖库，那么你可能需要遵循相关的[命名规范](#)和[API Guidelines](#).

## 打包

下一步就是将你的项目进行打包，然后上传到 `crates.io`。为了实现这个目的，我们可以使用 `cargo publish` 命令，该命令执行了以下步骤：

1. 对项目进行一些验证
2. 将源代码压缩到 `.crate` 文件中
3. 将 `.crate` 文件解压并放入到临时的目录中，并验证解压出的代码可以顺利编译
4. 上传 `.crate` 文件到 `crates.io`
5. 注册服务会对上传的包进行一些额外的验证，然后才会添加它到注册服务列表中

在发布之前，我们推荐你先运行 `cargo publish --dry-run` (或 `cargo package`) 命令来确保代码没有 warning 或错误。

```
$ cargo publish --dry-run
```

你可以在 `target/package` 目录下观察生成的 `.crate` 文件。例如，目前 `crates.io` 要求该文件的大小不能超过 10MB，你可以通过手动检查该文件的大小来确保不会无意间打包进一些较大的资源文件，比如测试数据、网站文档或生成的代码等。我们还可以使用以下命令来检查其中包含的文件：

```
$ cargo package --list
```

当打包时，Cargo 会自动根据版本控制系统的配置来忽略指定的文件，例如 `.gitignore`。除此之外，你还可以通过 `exclude` 来排除指定的文件：

```
[package]
# ...
exclude = [
    "public/assets/*",
    "videos/*",
]
```

如果想要显式地将某些文件包含其中，可以使用 `include`，但是需要注意的是，这个 key 一旦设置，那 `exclude` 就将失效：

```
[package]
# ...
include = [
    "**/*.rs",
    "Cargo.toml",
]
```

## 上传包

准备好后，我们就可以正式来上传指定的包了，在根目录中运行：

```
$ cargo publish
```

就是这么简单，恭喜你，完成了第一个包的发布！

## 发布已上传包的新版本

绝大多数时候，我们并不是在发布新包，而是发布已经上传过的包的新版本。

为了实现这一点，只需修改 `Cargo.toml` 中的 `version` 字段，但需要注意：**版本号需要遵循 semver 规则。**

然后再次使用 `cargo publish` 就可以上传新的版本了。

## 管理 crates.io 上的包

目前来说，管理包更多地是通过 `cargo` 命令而不是在线管理，下面是一些你可以使用的命令。

### cargo yank

有的时候你会遇到发布的包版本实际上并不可用(例如语法错误，或者忘记包含一个文件等)，对于这种情况，Cargo 提供了 `yank` 命令：

```
$ cargo yank --vers 1.0.1
$ cargo yank --vers 1.0.1 --undo
```

该命令**并不能删除任何代码**，例如如果你上传了一段隐私内容，你需要的是立刻重置它们，而不是使用 `cargo yank`。

`yank` 能做到的就是让其它人不能再使用这个版本作为依赖，但是现存的依赖依然可以继续工作。

`crates.io` 的一个主要目标就是作为一个不会随着时间变化的永久性包存档，但**删除某个版本显然违背了这个目标**。

## cargo owner

一个包可能会有多个主要开发者，甚至维护者 `maintainer` 都会发生变更。目前来说，只有包的 `owner` 才能发布新的版本，但是一个 `owner` 可以指定其它的用户为 `owner`:

```
$ cargo owner --add github-handle
$ cargo owner --remove github-handle
$ cargo owner --add github:rust-lang:owners
$ cargo owner --remove github:rust-lang:owners
```

命令中使用的 `ownerID` 必须是 GitHub 用户名或 Team 名。

一旦一个用户 `B` 通过 `--add` 被加入到 `owner` 列表中，他将拥有该包相关的所有权利。例如发布新版本、`yank` 一个版本，还能增加和移除 `owner`，包含添加 `B` 为 `owner` 的 `A` 都可以被移除！

因此，我们必须严肃的指出：**不要将你不信任的人添加为 owner！** 免得哪天反目成仇后，他把你移除了 -\_-

但是对于 Team 又有所不同，通过 `-add` 添加的 GitHub Team `owner`，只拥有受限的权利。它们可以发布或 `yank` 某个版本，但是他们**不能添加或移除 owner！** 总之，Team 除了可以很方便的管理所有者分组的同时，还能防止一些未知的恶意。

如果大家在添加 team 时遇到问题，可以看看官方的[相关文档](#)，由于绝大多数人都无需此功能，因此这里不再详细展开。

# 构建脚本( Build Scripts)

一些项目希望编译第三方的非 Rust 代码，例如 C 依赖库；一些希望链接本地或者基于源码构建的 C 依赖库；还有一些项目需要功能性的工具，例如在构建之间执行一些代码生成的工作等。

对于这些目标，社区已经提供了一些工具来很好的解决，Cargo 并不想替代它们，但是为了给用户带来一些便利，Cargo 提供了自定义构建脚本的方式，来帮助用户更好的解决类似的问题。

## build.rs

若要创建构建脚本，我们只需在项目的根目录下添加一个 `build.rs` 文件即可。这样一来，Cargo 就会先编译和执行该构建脚本，然后再去构建整个项目。

以下是一个非常简单的脚本示例：

```
fn main() {
    // 以下代码告诉 Cargo，一旦指定的文件 `src/hello.c` 发生了改变，就重新运行当前的构建脚本
    println!("cargo:rerun-if-changed=src/hello.c");
    // 使用 `cc` 来构建一个 C 文件，然后进行静态链接
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
}
```

关于构建脚本的一些使用场景如下：

- 构建 C 依赖库
- 在操作系统中寻找指定的 C 依赖库
- 根据某个说明描述文件生成一个 Rust 模块
- 执行一些平台相关的配置

下面的部分我们一起来看看构建脚本具体是如何工作的，然后在[下个章节](#)中还提供了一些关于如何编写构建脚本的示例。

---

Note: `package.build` 可以用于改变构建脚本的名称，或者直接禁用该功能

---

## 构建脚本的生命周期

在项目被构建之前，Cargo 会将构建脚本编译成一个可执行文件，然后运行该文件并执行相应的任务。

在运行的过程中，**脚本可以使用之前 `println!` 的方式跟 Cargo 进行通信**：通信内容是以 `cargo:` 开头的格式化字符串。

需要注意的是，Cargo 也不是每次都会重新编译构建脚本，只有当脚本的内容或依赖发生变化时才会。默认情况下，任何文件变化都会触发重新编译，如果你希望对其进行定制，可以使用 `rerun-if` 命令，后文会讲。

在构建脚本成功执行后，我们的项目就会开始进行编译。如果构建脚本的运行过程中发生错误，脚本应该通过返回一个非 0 码来立刻退出，在这种情况下，构建脚本的输出会被打印到终端中。

## 构建脚本的输入

我们可以通过[环境变量](#)的方式给构建脚本提供一些输入值，除此之外，构建脚本所在的当前目录也可以。

## 构建脚本的输出

构建脚本如果会产出文件，那么这些文件需要放在统一的目录中，该目录可以通过 `OUT_DIR` 环境变量来指定，**构建脚本不应该修改该目录之外的任何文件！**

在之前提到过，构建脚本可以通过 `println!` 输出内容跟 Cargo 进行通信：Cargo 会将每一行带有 `cargo:` 前缀的输出解析为一条指令，其它的输出内容会自动被忽略。

通过 `println!` 输出的内容在构建过程中默认是隐藏的，如果大家想要在终端中看到这些内容，你可以使用 `-vv` 来调用，以下 `build.rs`：

```
fn main() {
    println!("hello, build.rs");
}
```

将输出：

```
$ cargo run -vv
[study_cargo 0.1.0] hello, build.rs
```

构建脚本打印到标准输出 `stdout` 的所有内容将保存在文件 `target/debug/build/<pkg>/output` 中(具体的位置可能取决于你的配置), `stderr` 的输出内容也将保存在同一个目录中。

以下是 Cargo 能识别的通信指令以及简介, 如果大家希望深入了解每个命令, 可以点击具体的链接查看官方文档的说明。

- `cargo:rerun-if-changed=PATH` — 当指定路径的文件发生变化时, Cargo 会重新运行脚本
- `cargo:rerun-if-env-changed=VAR` — 当指定的环境变量发生变化时, Cargo 会重新运行脚本
- `cargo:rustc-link-arg=FLAG` - 将自定义的 flags 传给 linker, 用于后续的基准性能测试 benchmark、可执行文件 binary,、`cdylib` 包、示例和测试
- `cargo:rustc-link-arg-bin=BIN=FLAG` - 自定义的 flags 传给 linker, 用于可执行文件 BIN
- `cargo:rustc-link-arg-bins=FLAG` - 自定义的 flags 传给 linker, 用于可执行文件
- `cargo:rustc-link-arg-tests=FLAG` - 自定义的 flags 传给 linker, 用于测试
- `cargo:rustc-link-arg-examples=FLAG` - 自定义的 flags 传给 linker, 用于示例
- `cargo:rustc-link-arg-benches=FLAG` - 自定义的 flags 传给 linker, 用于基准性能测试 benchmark
- `cargo:rustc-cdylib-link-arg=FLAG` — 自定义的 flags 传给 linker, 用于 `cdylib` 包
- `cargo:rustc-link-lib=[KIND=]NAME` — 告知 Cargo 通过 `-l` 去链接一个指定的库, 往往用于链接一个本地库, 通过 FFI
- `cargo:rustc-link-search=[KIND=]PATH` — 告知 Cargo 通过 `-L` 将一个目录添加到依赖库的搜索路径中
- `cargo:rustc-flags=FLAGS` — 将特定的 flags 传给编译器
- `cargo:rustc-cfg=KEY[="VALUE"]` — 开启编译时 `cfg` 设置
- `cargo:rustc-env=VAR=VALUE` — 设置一个环境变量
- `cargo:warning=MESSAGE` — 在终端打印一条 warning 信息
- `cargo:KEY=VALUE` — `links` 脚本使用的元数据

## 构建脚本的依赖

构建脚本也可以引入其它基于 Cargo 的依赖包, 只需要在 `Cargo.toml` 中添加或修改以下内容:

```
[build-dependencies]
cc = "1.0.46"
```

需要这么配置的原因在于构建脚本无法使用通过 `[dependencies]` 或 `[dev-dependencies]` 引入的依赖包, 因为构建脚本的编译运行过程跟项目本身的编译过程是分离的的, 且前者先于后者发生。同样的, 我们项目也无法使用 `[build-dependencies]` 中的依赖包。

**大家在引入依赖的时候，需要仔细考虑它会给编译时间、开源协议和维护性等方面带来什么样的影响。如果**你在 [build-dependencies] 和 [dependencies] 引入了同样的包，这种情况下 Cargo 也许会对依赖进行复用，也许不会，例如在交叉编译时，如果不会，那编译速度自然会受到不小的影响。

## links

在 Cargo.toml 中可以配置 package.links 选项，它的目的是告诉 Cargo 当前项目所链接的本地库，同时提供了一种方式可以在项目构建脚本之间传递元信息。

```
[package]
# ...
links = "foo"
```

以上配置表明项目链接到一个 libfoo 本地库，当使用 links 时，项目必须拥有一个构建脚本，并且该脚本需要使用 rustc-link-lib 指令来链接目标库。

Cargo 要求一个本地库最多只能被一个项目所链接，换而言之，你无法让两个项目链接到同一个本地库，但是有一种方法可以降低这种限制，感兴趣的同学可以看看[官方文档](#)。

假设 A 项目的构建脚本生成任意数量的 kv 形式的元数据，那这些元数据将传递给 A 用作依赖包的项目的构建脚本。例如，如果包 bar 依赖于 foo，当 foo 生成 key=value 形式的构建脚本元数据时，那么 bar 的构建脚本就可以通过环境变量的形式使用该元数据：DEP\_FOO\_KEY=value。

需要注意的是，该元数据只能传给直接相关者，对于间接的，例如依赖的依赖，就无能为力了。

## 覆盖构建脚本

当 Cargo.toml 设置了 links 时，Cargo 就允许我们使用自定义库对现有的构建脚本进行覆盖。在[Cargo 使用的配置文件](#)中添加以下内容：

```
[target.x86_64-unknown-linux-gnu.foo]
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = "-L /some/path"
rustc-cfg = ['key="value"]'
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["..."]
metadata_key1 = "value"
metadata_key2 = "value"
```

增加这个配置后，在未来，一旦我们的某个项目声明了它链接到 `foo`，那项目的构建脚本将不会被编译和运行，替代的是这里的配置将被使用。

`warning`, `rerun-if-changed` 和 `rerun-if-env-changed` 这三个 key 在这里不应该被使用，就算用了也会被忽略。

# 构建脚本示例

下面我们通过一些例子来说明构建脚本该如何使用。社区中也提供了一些构建脚本的[常用功能](#)，例如：

- [bindgen](#), 自动生成 Rust -> C 的 FFI 绑定
- [cc](#), 编译 C/C++/汇编
- [pkg-config](#), 使用 `pkg-config` 工具检测系统库
- [cmake](#), 运行 `cmake` 来构建一个本地库
- [autocfg](#), [rustc\\_version](#), [version\\_check](#), 这些包提供基于 `rustc` 的当前版本来实现条件编译的方法

## 代码生成

一些项目需要在编译开始前先生成一些代码，下面我们来看看如何在构建脚本中生成一个库调用。

先来看看项目的目录结构：

```
.  
├── Cargo.toml  
├── build.rs  
└── src  
    └── main.rs
```

1 directory, 3 files

`Cargo.toml` 内容如下：

```
# Cargo.toml  
  
[package]  
name = "hello-from-generated-code"  
version = "0.1.0"
```

接下来，再来看看构建脚本的内容：

```
// build.rs

use std::env;
use std::fs;
use std::path::Path;

fn main() {
    let out_dir = env::var_os("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("hello.rs");
    fs::write(
        &dest_path,
        "pub fn message() -> &'static str {
            \"Hello, World!\""
    );
    .unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

以上代码中有几点值得注意：

- OUT\_DIR 环境变量说明了构建脚本的输出目录，也就是最终生成的代码文件的存放地址
- 一般来说，构建脚本不应该修改 OUT\_DIR 之外的任何文件
- 这里的代码很简单，但是我们这是为了演示，大家完全可以生成更复杂、更实用的代码
- return-if-changed 指令告诉 Cargo 只有在脚本内容发生变化时，才能重新编译和运行构建脚本。如果没有这一行，项目的任何文件发生变化都会导致 Cargo 重新编译运行该构建脚本

下面，我们来看看 main.rs：

```
// src/main.rs

include!(concat!(env!("OUT_DIR"), "/hello.rs"));

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

这里才是真正技术的地方，我们联合使用 rustc 定义的 include! 以及 concat! 和 env! 宏，将生成的代码文件( hello.rs )纳入到我们项目的编译流程中。

例子虽然很简单，但是它清晰地告诉了我们该如何生成代码文件以及将这些代码文件纳入到编译中来，大家以后有需要只要回头看看即可。

# 构建本地库

有时，我们需要在项目中使用基于 C 或 C++ 的本地库，而这种使用场景恰恰是构建脚本非常擅长的。

例如，下面来看看该如何在 Rust 中调用 C 并打印 Hello, World。首先，来看看项目结构和

Cargo.toml：

```
.  
└── Cargo.toml  
└── build.rs  
└── src  
    └── hello.c  
    └── main.rs
```

1 directory, 4 files

```
# Cargo.toml  
  
[package]  
name = "hello-world-from-c"  
version = "0.1.0"  
edition = "2021"
```

现在，我们还不会使用任何构建依赖，先来看看构建脚本：

```
// build.rs  
  
use std::process::Command;  
use std::env;  
use std::path::Path;  
  
fn main() {  
    let out_dir = env::var("OUT_DIR").unwrap();  
  
    Command::new("gcc").args(&["src/hello.c", "-c", "-fPIC", "-o"])
                    .arg(&format!("{}/hello.o", out_dir))
                    .status().unwrap();  
    Command::new("ar").args(&["crus", "libhello.a", "hello.o"])
                    .current_dir(&Path::new(&out_dir))
                    .status().unwrap();  
  
    println!("cargo:rustc-link-search=native={}", out_dir);
    println!("cargo:rustc-link-lib=static=hello");
    println!("cargo:rerun-if-changed=src/hello.c");  
}
```

首先，构建脚本将我们的 C 文件通过 `gcc` 编译成目标文件，然后使用 `ar` 将该文件转换成一个静态库，最后告诉 Cargo 我们的输出内容在 `out_dir` 中，编译器要在这里搜索相应的静态库，最终通过 `-l static-hello` 标志将我们的项目跟 `libhello.a` 进行静态链接。

但是这种硬编码的解决方式有几个问题：

- `gcc` 命令的跨平台性是受限的，例如 Windows 下就难以使用它，甚至于有些 Unix 系统也没有 `gcc` 命令，同样，`ar` 也有这个问题
- 这些命令往往不会考虑交叉编译的问题，如果我们要为 Android 平台进行交叉编译，那么 `gcc` 很可能无法输出一个 ARM 的可执行文件

但是别怕，构建依赖 `[build-dependencies]` 解君忧：社区中已经有现成的解决方案，可以让这种任务得到更容易的解决。例如文章开头提到的 `cc` 包。首先在 `Cargo.toml` 中为构建脚本引入 `cc` 依赖：

```
[build-dependencies]
cc = "1.0"
```

然后重写构建脚本使用 `cc`：

```
// build.rs

fn main() {
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
    println!("cargo:rerun-if-changed=src/hello.c");
}
```

不得不说，Rust 社区的大腿就是粗，代码立刻简洁了很多，最重要的是：可移植性、稳定性等头疼的问题也得到了一并解决。

简单来说，`cc` 包将构建脚本使用 `c` 的需求进行了抽象：

- `cc` 会针对不同的平台调用合适的编译器：windows 下调用 MSVC, MinGW 下调用 `gcc`, Unix 平台调用 `cc` 等
- 在编译时会考虑到平台因素，例如将合适的标志传给正在使用的编译器
- 其它环境变量，例如 `OPT_LEVEL`、`DEBUG` 等会自动帮我们处理
- 标准输出和 `OUT_DIR` 的位置也会被 `cc` 所处理

如上所示，与其在每个构建脚本中复制粘贴相同的代码，将尽可能多的功能通过构建依赖来完成是好得多的选择。

再回到例子中，我们来看看 `src` 下的项目文件：

```

// src/hello.c

#include <stdio.h>

void hello() {
    printf("Hello, World!\n");
}

// src/main.rs

// 注意，这里没有再使用 `#[link]` 属性。我们把选择使用哪个 link 的责任交给了构建脚本，而不是在这里
进行硬编码
extern { fn hello(); }

fn main() {
    unsafe { hello(); }
}

```

至此，这个简单的例子已经完成，我们学到了该如何使用构建脚本来构建 C 代码，当然又一次被构建脚本和构建依赖的强大所震撼！但控制下情绪，因为构建脚本还能做到更多。

## 链接系统库

当一个 Rust 包想要链接一个本地系统库时，如何实现平台透明化，就成了一个难题。

例如，我们想使用在 Unix 系统中的 `zlib` 库，用于数据压缩的目的。实际上，社区中的 `libz-sys` 包已经这么做了，但是出于演示的目的，我们来看看该如何手动完成，当然，这里只是简化版的，想要看完整代码，见[这里](#)。

为了更简单的定位到目标库的位置，可以使用 `pkg-config` 包，该包使用系统提供的 `pkg-config` 工具来查询库的信息。它会自动告诉 Cargo 该如何链接到目标库。

先修改 `Cargo.toml`：

```

# Cargo.toml

[package]
name = "libz-sys"
version = "0.1.0"
edition = "2021"
links = "z"

[build-dependencies]
pkg-config = "0.3.16"

```

这里的 `links = "z"` 用于告诉 Cargo 我们想要链接到 `libz` 库，在[下文](#)还有更多的示例。

构建脚本也很简单：

```
// build.rs

fn main() {
    pkg_config::Config::new().probe("zlib").unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

下面再在代码中使用：

```
// src/lib.rs

use std::os::raw::{c_uint, c_ulong};

extern "C" {
    pub fn crc32(crc: c_ulong, buf: *const u8, len: c_uint) -> c_ulong;
}

#[test]
fn test_crc32() {
    let s = "hello";
    unsafe {
        assert_eq!(crc32(0, s.as_ptr(), s.len() as c_uint), 0x3610a686);
    }
}
```

代码很清晰，也很简洁，这里就不再过多介绍，运行 `cargo build --vv` 来看看部分结果( 系统中需要已经安装 `libz` 库)：

```
[libz-sys 0.1.0] cargo:rustc-link-search=native=/usr/lib
[libz-sys 0.1.0] cargo:rustc-link-lib=z
[libz-sys 0.1.0] cargo:rerun-if-changed=build.rs
```

非常棒，`pkg-config` 帮助我们找到了目标库，并且还告知了 Cargo 所有需要的信息！

实际使用中，我们需要做的比上面的代码更多，例如 `libz-sys` 包会先检查环境变量 `LIBZ_SYS_STATIC` 或者 `static` feature，然后基于源码去构建 `libz`，而不是直接去使用系统库。

## 使用其它 sys 包

本例中，一起来看看该如何使用 `libz-sys` 包中的 `zlib` 来创建一个 C 依赖库。

若你有一个依赖于 `zlib` 的库，那可以使用 `libz-sys` 来自动发现或构建该库。这个功能对于交叉编译非常有用，例如 Windows 下往往不会安装 `zlib`。

`libz-sys` 通过设置 `include` 元数据来告知其它包去哪里找到 `zlib` 的头文件，然后我们的构建脚本可以通过 `DEP_Z_INCLUDE` 环境变量来读取 `include` 元数据( 关于元数据的传递，见[这里](#) )。

```
# Cargo.toml

[package]
name = "zuser"
version = "0.1.0"
edition = "2021"

[dependencies]
libz-sys = "1.0.25"

[build-dependencies]
cc = "1.0.46"
```

通过包含 `libz-sys`，确保了最终只会使用一个 `libz` 库，并且给了我们在构建脚本中使用的途径：

```
// build.rs

fn main() {
    let mut cfg = cc::Build::new();
    cfg.file("src/zuser.c");
    if let Some(include) = std::env::var_os("DEP_Z_INCLUDE") {
        cfg.include(include);
    }
    cfg.compile("zuser");
    println!("cargo:rerun-if-changed=src/zuser.c");
}
```

由于 `libz-sys` 帮我们完成了繁重的相关任务，C 代码只需要包含 `zlib` 的头文件即可，甚至于它还能在没有安装 `zlib` 的系统上找到头文件：

```
// src/zuser.c

#include "zlib.h"

// ... 在剩余的代码中使用 zlib
```

## 条件编译

构建脚本可以通过发出 `rustc-cfg` 指令来开启编译时的条件检查。在本例中，一起来看看 `openssl` 包是如何支持多版本的 OpenSSL 库的。

`openssl-sys` 包对 OpenSSL 库进行了构建和链接，支持多个不同的实现(例如 LibreSSL)和多个不同的版本。它也使用了 `links` 配置，这样就可以给**其它构建脚本**传递所需的信息。例如 `version_number`，包含了检测到的 OpenSSL 库的版本号信息。`openssl-sys` 自己的**构建脚本**中有类似于如下的代码：

```
println!("cargo:version_number={:x}", openssl_version);
```

该指令将 `version_number` 的信息通过环境变量 `DEP_OPENSSL_VERSION_NUMBER` 的方式传递给直接使用 `openssl-sys` 的项目。例如 `openssl` 包提供了更高级的抽象接口，并且它使用了 `openssl-sys` 作为依赖。`openssl` 的构建脚本会通过环境变量读取 `openssl-sys` 提供的版本号的信息，然后使用该版本号来生成一些 `cfg`：

```
// (portion of build.rs)

if let Ok(version) = env::var("DEP_OPENSSL_VERSION_NUMBER") {
    let version = u64::from_str_radix(&version, 16).unwrap();

    if version >= 0x1_00_01_00_0 {
        println!("cargo:rustc-cfg=openssl101");
    }
    if version >= 0x1_00_02_00_0 {
        println!("cargo:rustc-cfg=openssl102");
    }
    if version >= 0x1_01_00_00_0 {
        println!("cargo:rustc-cfg=openssl110");
    }
    if version >= 0x1_01_00_07_0 {
        println!("cargo:rustc-cfg=openssl110g");
    }
    if version >= 0x1_01_01_00_0 {
        println!("cargo:rustc-cfg=openssl111");
    }
}
```

这些 `cfg` 可以跟 `cfg 属性` 或 `cfg 宏`一起使用以实现条件编译。例如，在 OpenSSL 1.1 中引入了 SHA3 的支持，那么我们就可以指定只有当版本号为 1.1 时，才包含并编译相关的代码：

```
// (portion of openssl crate)

#[cfg(ssl111)]
pub fn sha3_224() -> MessageDigest {
    unsafe { MessageDigest(ffi::EVP_sha3_224()) }
}
```

当然，大家在使用时一定要小心，因为这可能会导致生成的二进制文件进一步依赖当前的构建环境。例如，当二进制可执行文件需要在另一个操作系统中分发运行时，那它依赖的信息对于该操作系统可能是不存在的！

# Rust的使用案例

自从 Rust 基金会成立后，一些优秀的落地案例如雨后春笋般冒了出来，这些充分说明了 Rust 语言在全球范围的流行。

这里我们将通过几个精心挑选的使用案例，来帮助大家看看 Rust 能为企业解决哪些痛点，以及该如何在企业内落地。

# 使用 Rust 来节约企业成本

夸 Rust 语言的方式至少有 3000 种，但是从环保和可持续发展角度去夸的大家见过嘛？这不，AWS 就给我们带来了一篇非常精彩的文章，一起来看看。

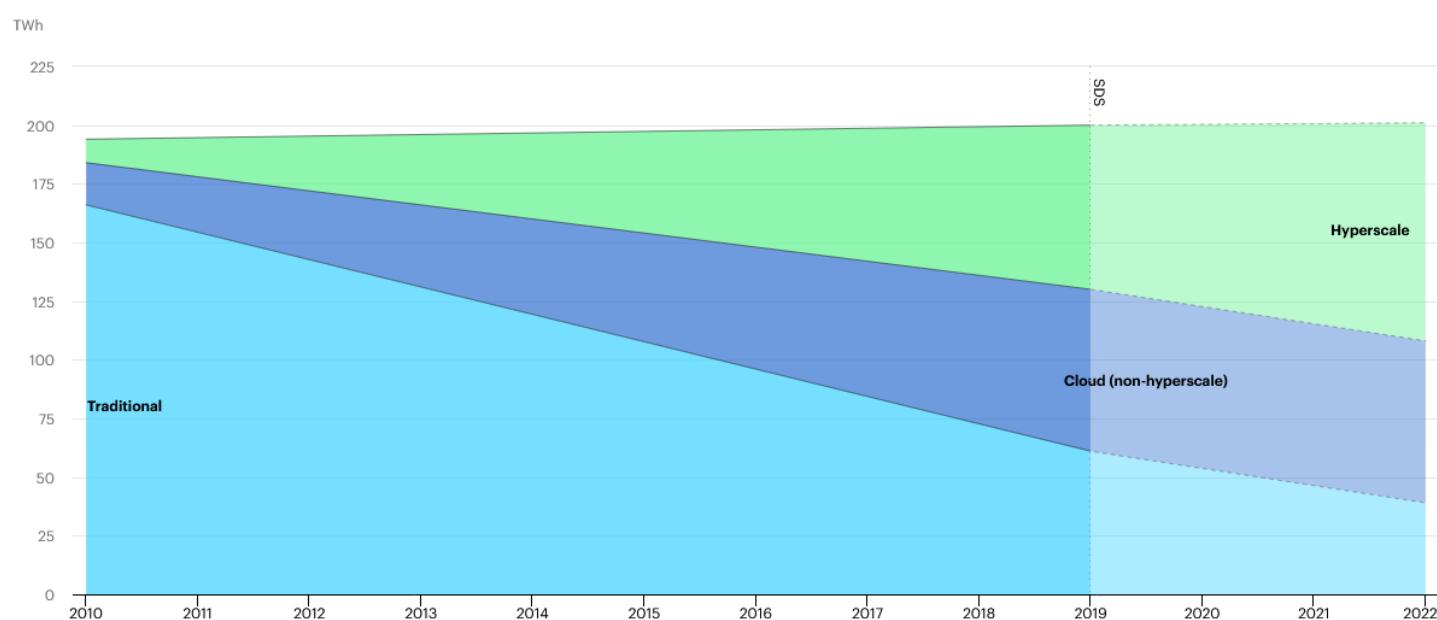
原文链接：<https://aws.amazon.com/cn/blogsopensource/sustainability-with-rust/>，由于原文过长，译文进行了适当的精简(例如夸 AWS 的部分 - , -)

Rust 是一门完全开源的语言，在 2015 年发布了 1.0 版本，但是在 2020 年才迎来了真正的大发展契机，这是由于 Rust 的开源支持由 Mozilla 移交给了 [Rust基金会](#)，后者是由亚马逊云AWS、谷歌、华为、微软和 Mozilla 共同创建的非营利性组织。

在 AWS，Rust 已经成为构建大规模基础设施的关键，其中一部分：

- [Firecraker](#) 是开源的虚拟化技术，用于支持 AWS Lambda 和其它无服务器计算
- 为 Amazon S3、Amazon EC2、Amazon CloudFront 等提供关键性服务
- 在 2020 年发布了 [Bottlerocket](#) 一个基于 Linux 的容器化操作系统

## 云计算的能源效率



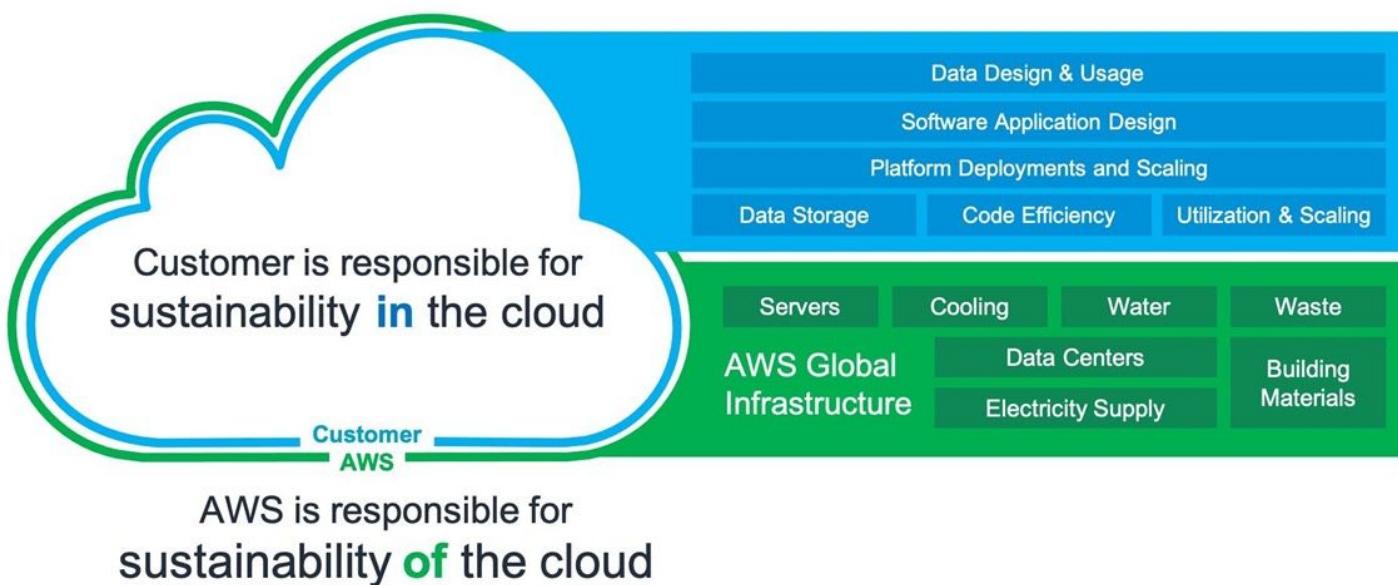
来源: IEA(2021)，全球数据中心能源需求，2010-2022，<https://www.iea.org/data-and-statistics/charts/global-data-centre-energy-demand-by-data-centre-type-2010-2022>

在全世界范围内，数据中心每年大概消耗 200 太瓦时 的能源，大概占全球能源消耗总量的 1%。

图中有几个有趣的点值得关注。首先，数据中心消耗的能源总量在过去 12 年间并没有显著的变化，这个其实蛮反直觉的，因为在这些年间，大数据、机器学习、边缘计算、区块链等等耗电大户发展速度都非常快。其次，虽然总量没有变，但是三种数据中心的能源占比分布发生了巨大的变化：超大规模(hyperscale)、传统服务、云计算。

总量反直觉变化的关键，其实就在于这些年能源效率得到了大幅提升，以及很多传统服务都迁移到了云计算上，而后者通过多租户、智能硬件资源利用、优化驱动和存储、更高效的冷却系统等一系列措施大幅降低了能源的消耗。

尽管能源效率提升巨大，但是，依然存在两个问题。首先，当前的现状已经足够好了吗？例如让能源占比保持在 1% 的水准。其次，能源效率能否像过去一样继续快速提升？考虑到即将爆发的无人驾驶、机器人、机器学习领域，我们对此保持不乐观的态度，因为这个领域需要处理异常巨大的数据集。



当然，还能从能源本身入手来改善，例如 AWS 计划在 2025 年之前实现所有数据中心都使用可再生能源，但可再生能源并不意味着它没有环境影响，它依然需要 50 万英亩的太阳能板来生成 200 太瓦时的数据中心能源需求。总之，可再生能源不是一个设计上的概念，也不能替代设计上的优化，我们还需要其它方式。

这些其它方式包括：为非关键服务放松 SLA 的要求和资源供给优先级，利用虚拟化实现更长的设备升级周期，更多的利用缓存、设置更长的 TTL，对数据进行分类并通过自动化的策略来实现尽可能及时的数据删除，为加密和压缩选择更高效的算法等等，最后但也最重要的是：**我们可以选择使用一门能源效率高的编程语言来实现基础服务和用户端的软件。**

## 编程语言的能源效率

对于开发者来说，估计没几个人能搞清楚自己服务的能源效率，那么该如何对比编程语言之间的能源效率呢？好在国外有专家做了相关的学术研究。

他精心设计了 10 个测试场景，然后衡量了 27 种不同的语言的执行时间、能源消耗、最大内存使用，最终得出了一个结论：C 和 Rust 在能源效率方面无可争议地击败了其它语言——比 Java 高出 98%，是 Python 的近 76 倍。

	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

其实，C 和 Rust 的能效高很正常，但比其它语言高出这么多就相当出乎意料了：根据上图的数据，采用这两门语言后，程序的能耗将降低至少一倍。这还是与榜单中靠前的 Java 语言对比的结果。

那么问题来了，既然这两个都可以，为何不选择历史更悠久的 C 语言呢？它的生态和社区都比 Rust 要更好。好在，linux 创始人 Linus Torvalds 在 2021 年度的开源峰会上给出了答案：他承认，[使用 C 语言就像是拿着一把链锯在玩耍](#)，同时还说道：“C 语言的类型互动并不总是合乎逻辑的，以至于对于绝大多数人来说，这种互动都可能存在陷阱”。

作为侧面的证明，Rust 在去年下半年被纳入了 Linux 的官方开发语言，如果大家想知道其它的官方语言有哪些？我可以很轻松的列出一个列表：C 语言。。。这叫列表？这就结束了？其它的呢？别急，事实上，之前仅有 C 语言是官方支持的，可想而知 Rust 是多么的优秀才能从这么多竞争者中脱颖而出！

总之，Linus Torvalds 亲口说过 Rust 是他见过的第一门可以称之为能很好的解决问题的编程语言，Rust 在比肩 C 的效率的同时，还能避免各种不安全的风险，对于能耗来说，我们就能在节省一半能耗的同时还不用担心安全性。

多个分析报告也指出：七成以上的 C/C++ 的高风险 CVE 可以在 Rust 中得到有效的规避，且使用的是同样的解决方法！事实上，ISRG(非盈利组织，Let's Encrypt 项目发起者) 就有一个目标，希望能将所有对网络安全敏感的基础设施转移到 Rust 上。正在进行的项目包括：Linux 内核对 Rust 的进一步支持，以及将 curl 迁移到基于 TLS 和 HTTP 的 Rust 版本上。

	Energy	Time	Mb
(c) C	1.00	1.00	1.00
(c) Rust	1.03	1.04	1.05
(c) C++	1.34	1.56	1.17
(c) Ada	1.70	1.85	1.24
(v) Java	1.98	1.89	1.34
(c) Pascal	2.14	2.14	1.47
(c) Chapel	2.18	2.83	1.54
(v) Lisp	2.27	3.02	1.92
(c) Ocaml	2.40	3.09	2.45
(c) Fortran	2.52	3.14	2.57
(c) Swift	2.79	3.40	2.71
(c) Haskell	3.10	3.55	2.80
(v) C#	3.14	4.20	2.82
(c) Go	3.23	4.20	2.85
(i) Dart	3.83	6.30	3.34
(v) F#	4.13	6.52	3.52
(i) JavaScript	4.45	6.67	3.97
(v) Racket	7.91	11.27	4.00
(i) TypeScript	21.50	26.99	4.25
(i) Hack	24.02	27.64	4.59
(i) PHP	29.30	36.71	4.69
(v) Erlang	42.23	43.44	4.90
(i) Lua	45.98	46.20	5.01
(i) Jruby	46.54	59.34	6.62
(i) Ruby	69.91	65.79	6.72
(i) Python	75.88	71.90	7.20
(i) Perl	79.58	82.91	8.64

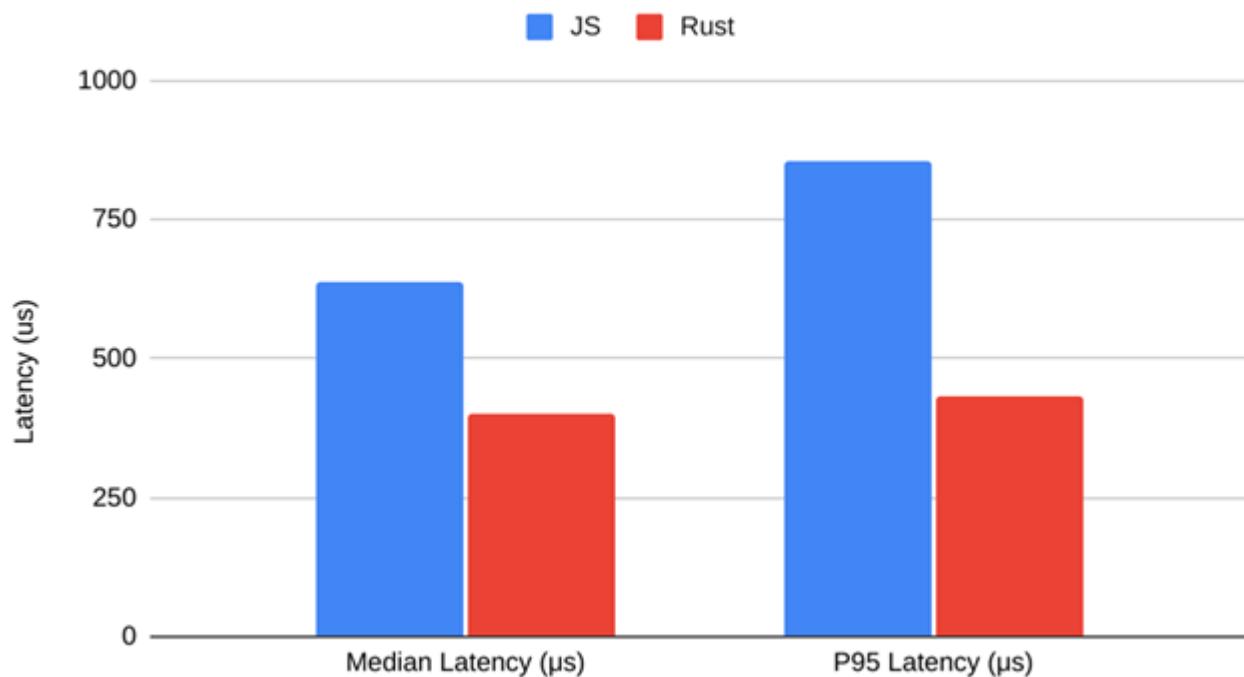
不仅仅是能耗，上图中的中间列还提供了执行时间的测量指标，可以看出 Rust 和 C 在性能上也相当接近，而且两者都比其它语言也更快(其实对于 C++ 的结果，我个人不太理解，如果有大神阅读过之前提到的学术研究报告，欢迎给出答案:D )。这意味着，当为了能效和安全选择了 Rust 后，我们依然可以获得类似 C 语言级别的性能，还是优化过的。

## Rust 成功案例

下面一起来看几个关于使用 Rust 后获得性能和成本双丰收的案例。

## Tenable

Latency per datagram



<https://medium.com/tenable-techblog/optimizing-700-cpus-away-with-rust-dc7a000dbdb2>

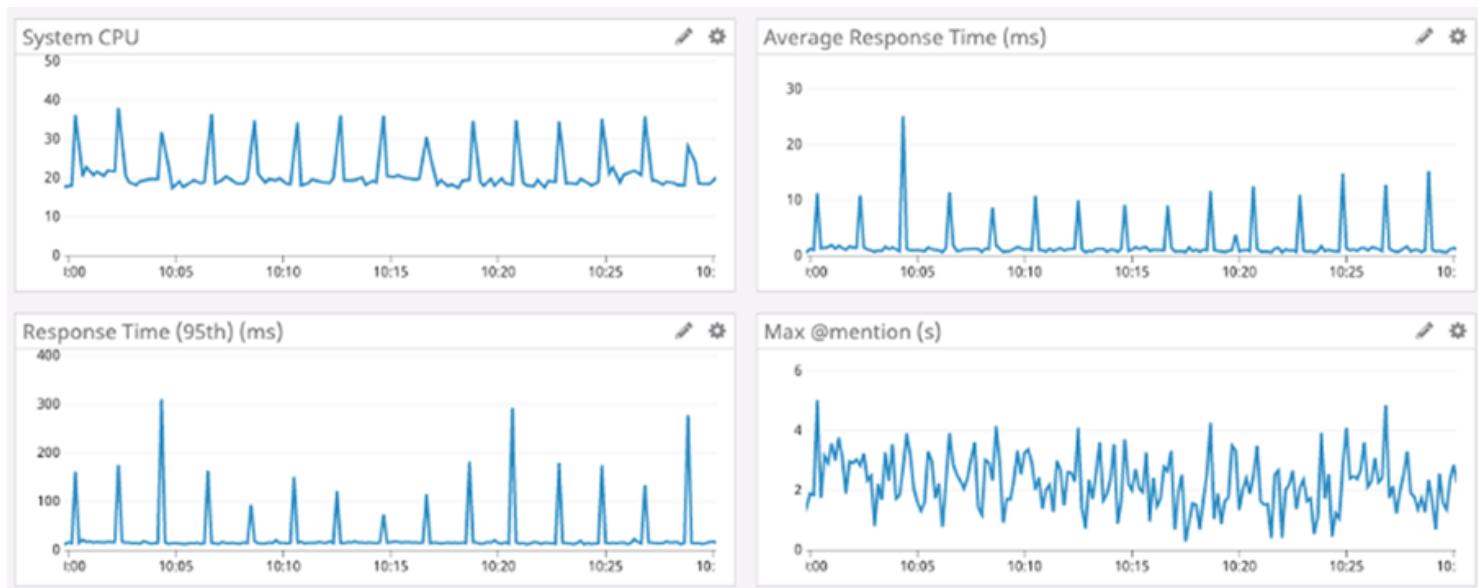
Tenable 是一家网络安全解决方案提供商，它提供了一套可视化工具，并通过一个 `sidecar agent` 来过滤采集到的指标数据。最开始，该公司使用 JavaScript 作为主要语言，当业务开始快速增长时，性能降级的问题就不可避免的发生了。

因此，在经过一系列调研后，Tenable 最终决定使用 Rust 来重写该服务，以获取更好的性能和安全性。最终结果也没有让他们失望，在延迟方面获得了 50% 的提升，如上图所示。

除了用户体验的提升之外，在 CPU 和内存占用方面也提升巨大，这可以帮助他们节省大量的硬件和能耗成本：



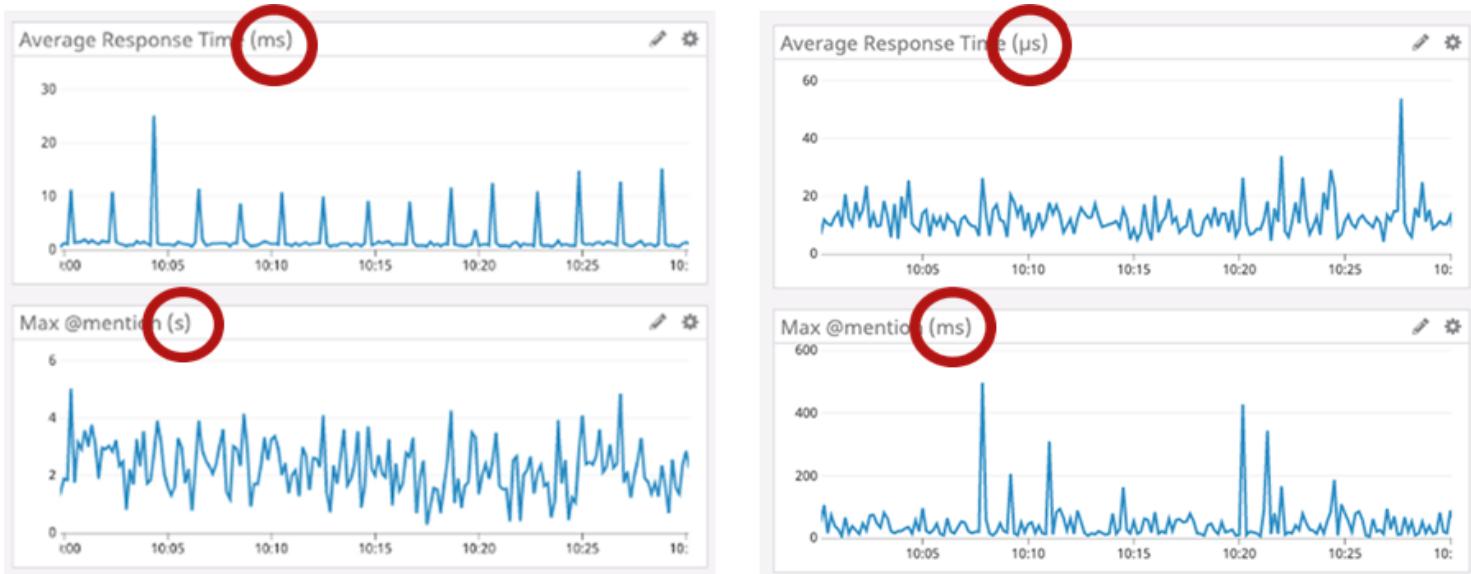
## Discord



<https://discord.com/blog/why-discord-is-switching-from-go-to-rust>

Discord 最初使用 Python、Go、Elixir 来实现，但是随即他们发现其中一个关键的 Go 服务存在一些问题。该服务很简单，但是会出现较慢的尾延迟现象，原因是 Go 语言拥有 GC 垃圾回收，当对象在快速创建和释放时，GC 会较为频繁的运行，然后导致整个程序的暂停，因此当 GC 发生时，程序是无法响应用户的请求的，从图中的不断起伏的峰值表现，你可以看出问题所在。

为了解决问题，Discord 选择使用 Rust 来重写该服务，然后下图是结果对比，Go 语言是左边一列，Rust 是右边一列：



从图中可以看出几点：

- 由于 GC 导致的峰值起伏没有了
- Rust 版本比 Go 的版本响应时间降低了 10 倍以上，注意看，图中的时间单位是不一样的

在重写后，由于性能的大幅提升，还帮助 Discord 降低了服务器资源的需求，变相节省了大笔金钱。

从上面两个例子中，我们看到两个公司都是为了性能才去使用 Rust，但是在性能之外他们还收获了能效上的提升和硬件成本上的降低，这不得不说是一种意外之喜了。

# 日志和监控

这几年 AIOps 特别火，但是你要是逮着一个运维问一下，他估计很难说出个所以然来，毕竟概念和现实往往是脱节的，前者的发展速度肯定远快于后者。

好在我大概了解这块儿领域，可以说智能化运维的核心就在于日志和监控，换而言之？何为智能，不就是基于已有的海量数据分析后进行决策吗？当然，你要说以前的知识库类型的运维决策也是智能，我也没办法杠：D

总之，不仅仅是对于开发者，对于整个技术链条的参与者，甚至包括老板，**日志和监控都是开发实践中最重要的一环。**

# 详解日志

相比起监控，日志好理解的多：在某个时间点向指定的地方输出一条信息，里面记录着重要性、时间、地点和发生的事件，这就是日志。

---

注意，本文和 Rust 无关，我们争取从一个中立的角度去介绍何为日志

---

## 日志级别和输出位置

### 日志级别

日志级别是对基本的“滚动文本”式日志记录的一个重要补充。每条日志消息都会基于其重要性或严重程度分配到一个日志级别。例如，对于某个程序，“你的电脑着火了”是一个非常重要的消息，而“无法找到配置文件”的重要等级可能就低一些；但对于另外一些程序，“无法找到配置文件”可能才是最严重的错误，会直接导致程序无法正常启动，而“电脑着火”？我们可能会记录为一条 Debug 日志(参见下文) :D。

至于到底该如何定义日志级别，这是仁者见仁的事情，并没有一个约定俗成的方式，就连很多大公司，都无法保证自己的开发者严格按照它所制定的规则来输出日志。而下面是我认为的日志级别以及相关定义：

- Fatal: 程序发生致命错误，祝你好运。这种错误往往来自于程序逻辑的严重异常，例如之前提到的“无法找到配置文件”，再比如无法分配足够的硬盘空间、内存不够用等。遇到这种错误，建议立即退出或者重启程序，然后记录下相应的错误信息
- Error: 错误，一般指的是程序级别的错误或者严重的业务错误，但这种错误并不会影响程序的运行。一般的用户错误，例如用户名、密码错误等，不使用 Error 级别
- Warn: 警告，说明这条记录信息需要注意，但是不确定是否发生了错误，因此需要相关的开发来辨别下。或者这条信息既不是错误，但是级别又没有低到 info 级别，就可以用 Warn 来给出警示。例如某条用户连接异常关闭、无法找到相关的配置只能使用默认配置、XX秒后重试等
- Info: 信息，这种类型的日志往往用于记录程序的运行信息，例如用户操作或者状态的变化，再比如之前的用户名、密码错误，用户请求的开始和结束都可以记录为这个级别
- Debug: 调试信息，顾名思义是给开发者用的，用于了解程序当前的详细运行状况，例如用户请求详细信息跟踪、读取到的配置信息、连接握手包(连接的建立和结束往往是 Info 级别)，就可以记录为 Debug 信息

可以看出，日志级别很多，特别是 Debug 日志，如果在生产环境中开启，简直就是一场灾难，每秒几百上千条都很正常。因此我们需要控制日志的最低级别：将最低级别设置为 Info 时，意味着低于 Info 的日志都不会输出，对于上面的分级来说，Debug 日志将不会被输出。

有些开发为了让特定的日志在控制上显示更明显，还会为不同的级别使用不同颜色的文字。

## 输出位置

通常来说，日志可以输出两个地方：终端控制台和文件。对于前者，我们还有一个称呼标准输出，例如使用 `println!` 打印到终端的信息就是输出到标准输出中。

如果没有日志持久化的需求，你只是为了调试程序，建议输出到控制台即可。悄悄的说一句，我们还可以为不同的级别设定不同的输出位置，例如 Debug 日志输出到控制台，既方便开发查看，但又不会占用硬盘，而 Info 和 Warning 日志可以输出到文件 `info.log` 中，至于 Error、Fatal 则可以输出到 `error.log` 中。

但是如果大家以为只有输出到文件才能持久化日志，那你就错了，在后面的日志采集我们会详细介绍，先来看看日志查看。

## 日志查看

关于如何查看日志，相信大家都非常熟悉了，常用的方式有三种(事实上，可能也只有这三种)：

- 在控制台查看，即可以直接查看输出到标准输出的日志，还可以使用 `tail`、`cat`、`grep` 等命令从日志文件中搜索查询或者以实时滚动的方式查看最新的日志
- 最简单的，进入到日志文件中，进行字符串搜索，或者从头到尾、从尾到头进行逐行查看
- 在可视化界面上查看，但是这个往往要配合日志采集工具，将日志采集到 ElasticSearch 或者其它搜索平台、数据中，然后再通过 `kibana`、`grafana` 等图形化服务进行搜索、查看，最重要的是可以进行日志的聚合统计，例如可以很方便的在 `kibana` 中查询满足指定条件的日志在某段时间内出现了多少次。

大家现在知道了，可视化，首先需要将日志集中采集起来，那么该如何采集日志呢？

## 日志采集

之前我们提到，不是只有输出到文件才能持久化日志，事实上，输出到控制台也能持久化日志。

其中的秘诀就在于使用一个日志采集工具去从控制台的标准输出读取日志数据，然后将读取到的数据发送到日志存储平台，例如 ElasticSearch，进行集中存储。当然，在存储前，还需要进行日志格式、数据的

处理，以便只保留我们需要的格式和日志数据。

最典型的就是容器或容器云环境的日志采集，基本都是通过上面的方式进行的：容器中的进程将日志输出到标准输出，然后一个单独的日志采集服务直接读取标准输出中的日志，再通过网络发送到日志处理、存储的平台。大家发现了吗？这个流程完全不会在应用运行的本地或宿主机上存储任何日志，所以特别适合容器环境！

目前常用的日志采集工具有 filebeat、vector( Rust 开发，功能强大，性能非常高 ) 等，它们都是以 agent 的形式运行在你的应用程序旁边( 在同一个 pod 或虚拟机上 )，提供贴心的服务。

## 中心化日志存储

最后，我们再来简单介绍下日志存储。提到存储，首先不得不提的就是日志使用方式。

其实，除了 Debug 的时候，我们使用日志基本都是基于某个关键字进行搜索的，将日志存储在各台主机上的硬盘文件中，然后逐个去查询显然是非常非常低效的，最好的方式就是将日志集中收集上来后，存储在一个搜索平台中，例如 ElasticSearch。

当然，存储的时候肯定也不是简单的一行一行存储，而是需要将一条日志的多个关键词切取出来，然后以关键词索引的方式进行存储( 简化模型 )，这样我们就可以在后续使用时，通过关键词来搜索日志了。

# 日志门面 log

就如同 slf4j 是 Java 的日志门面库，`log` 也是 Rust 的日志门面库( 这不是我自己编的，官方用语: logging facade )，它目前由官方积极维护，因此大家可以放心使用。

使用方式很简单，只要在 `Cargo.toml` 中引入即可：

```
[dependencies]
log = "0.4"
```

---

日志门面不是说排场很大的意思，而是指相应的日志 API 已成为事实上的标准，会被其它日志框架所使用。通过这种统一的门面，开发者就可以不必再拘泥于日志框架的选择，未来大不了再换一个日志框架就是

---

既然是门面，`log` 自然定义了一套统一的日志特征和 API，将日志的操作进行了抽象。

## Log 特征

例如，它定义了一个 `Log` 特征：

```
pub trait Log: Sync + Send {
    fn enabled(&self, metadata: &Metadata<'_>) -> bool;
    fn log(&self, record: &Record<'_>);
    fn flush(&self);
}
```

- `enabled` 用于判断某条带有元数据的日志是否能被记录，它对于 `log_enabled!` 宏特别有用
- `log` 会记录 `record` 所代表的日志
- `flush` 会将缓存中的日志数据刷到输出中，例如标准输出或者文件中

# 日志宏

`log` 还为我们提供了一整套标准的宏，用于方便地记录日志。看到 `trace!`、`debug!`、`info!`、`warn!`、`error!`，大家是否感觉眼熟呢？是的，它们跟上一章节提到的日志级别几乎一模一样，唯一的

区别就是这里乱入了一个 `trace!`，它比 `debug!` 的日志级别还要低，记录的信息还要详细。可以说，你如果想巨细无遗地了解某个流程的所有踪迹，它就是不二之选。

```
use log::{info, trace, warn};

pub fn shave_the_yak(yak: &mut Yak) {
    trace!("Commencing yak shaving");

    loop {
        match find_a_razor() {
            Ok(razor) => {
                info!("Razor located: {}", razor);
                yak.shave(razor);
                break;
            }
            Err(err) => {
                warn!("Unable to locate a razor: {}, retrying", err);
            }
        }
    }
}
```

上面的例子使用 `trace!` 记录了一条可有可无的信息：准备开始剃须，然后开始寻找剃须刀，找到后就用 `info!` 记录一条可能事后也没人看的信息：找到剃须刀；没找到的话，就记录一条 `warn!` 信息，这条信息就有一定价值了，不仅告诉我们没找到的原因，还记录了发生的次数，有助于事后定位问题。

可以看出，这里使用日志级别的方式和我们上一章节所述基本相符。

除了以上常用的，`log` 还提供了 `log!` 和 `log_enabled!` 宏，后者用于确定一条消息在当前模块中，对于给定的日志级别是否能够被记录

```
use log::Level::Debug;
use log::{debug, log_enabled};

// 判断能否记录 Debug 消息
if log_enabled!(Debug) {
    let data = expensive_call();
    // 下面的日志记录较为昂贵，因此我们先在前面判断了是否能够记录，能，才继续这里的逻辑
    debug!("expensive debug data: {} {}", data.x, data.y);
}
if log_enabled!(target: "Global", Debug) {
    let data = expensive_call();
    debug!(target: "Global", "expensive debug data: {} {}", data.x, data.y);
}
```

而 `log!` 宏就简单的多，它是一个通用的日志记录方式，因此需要我们手动指定日志级别：

```
use log::{log, Level};

let data = (42, "Forty-two");
let private_data = "private";

log!(Level::Error, "Received errors: {}, {}", data.0, data.1);
log!(target: "app_events", Level::Warn, "App warning: {}, {}, {}",
    data.0, data.1, private_data);
```

## 日志输出在哪里？

我不知道有没有同学尝试运行过上面的代码，但是我知道，就算你们运行了，也看不到任何输出。

为什么？原因很简单，`log` 仅仅是日志门面库，**它并不具备完整的日志库功能！**，因此你无法在控制台中看到任何日志输出，这种情况下，说实话，远不如一个 `println!` 有用！

但是别急，让我们看看该如何让 `log` 有用起来。

## 使用具体日志库

`log` 包这么设计，其实有很多好处的。

### Rust 库的开发者

最直接的好处就是，如果你是一个 Rust 库开发者，那你自己或库的用户肯定都不希望这个库绑定任何具体日志库，否则用户想使用 `log1` 来记录日志，你的库却使用了 `log2`，这就存在很多问题了！

因此，**作为库的开发者，你只要在库中使用门面库即可**，将具体日志库交给用户去选择和绑定。

```
use log::{info, trace, warn};
pub fn deal_with_something() {
    // 开始处理

    // 记录一些日志
    trace!("a trace log");
    info!("a info long: {}", "abc");
    warn!("a warning log: {}, retrying", err);

    // 结束处理
}
```

## 应用开发者

如果是应用开发者，那你的应用运行起来，却看不到任何日志输出，这种场景想想都捉急。此时就需要去选择一个具体日志库了。

目前来说，已经有了不少日志库实现，官方也[推荐了一些](#)，大家可以根据自己的需求来选择，不过 `env_logger` 是一个相当不错的选择。

`log` 还提供了 `set_logger` 函数用于设置日志库，`set_max_level` 用于设置最大日志级别，但是如果你选了具体日志库，它往往会提供更高级的 API，无需我们手动调用这两个函数，例如下面的 `env_logger` 就是如此。

### `env_logger`

修改 `Cargo.toml`，添加以下内容：

```
# in Cargo.toml

[dependencies]
log = "0.4.0"
env_logger = "0.9"
```

在 `src/main.rs` 中添加如下代码：

```
use log::{debug, error, log_enabled, info, Level};

fn main() {
    // 注意，env_logger 必须尽可能早的初始化
    env_logger::init();

    debug!("this is a debug {}", "message");
    error!("this is printed by default");

    if log_enabled!(Level::Info) {
        let x = 3 * 4; // expensive computation
        info!("the answer was: {}", x);
    }
}
```

在运行程序时，可以通过环境变量来设定日志级别：

```
$ RUST_LOG=error ./main
[2017-11-09T02:12:24Z ERROR main] this is printed by default
```

我们还可以为单独一个模块指定日志级别：

```
$ RUST_LOG=main=info ./main
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12
```

还能为某个模块开启所有日志级别：

```
$ RUST_LOG=main ./main
[2017-11-09T02:12:24Z DEBUG main] this is a debug message
[2017-11-09T02:12:24Z ERROR main] this is printed by default
[2017-11-09T02:12:24Z INFO main] the answer was: 12
```

需要注意的是，如果文件名包含 -，你需要将其替换成下划线来使用，原因是 Rust 的模块和包名不支持使用 -。

```
$ RUST_LOG=my_app ./my-app
[2017-11-09T02:12:24Z DEBUG my_app] this is a debug message
[2017-11-09T02:12:24Z ERROR my_app] this is printed by default
[2017-11-09T02:12:24Z INFO my_app] the answer was: 12
```

默认情况下，`env_logger` 会输出到标准错误 `stderr`，如果你想要输出到标准输出 `stdout`，可以使用 `Builder` 来改变日志对象(`target`)：

```
use std::env;
use env_logger::{Builder, Target};

let mut builder = Builder::from_default_env();
builder.target(Target::Stdout);

builder.init();
```

默认

```
if cfg!(debug_assertions) {
    eprintln!("debug: {:#?} -> {:#?}", record, fields);
}
```

## 日志库开发者

对于这类开发者而言，自然要实现自己的 `Log` 特征咯：

```
use log::{Record, Level, Metadata};
struct SimpleLogger;
impl log::Log for SimpleLogger {
    fn enabled(&self, metadata: &Metadata) -> bool {
        metadata.level() <= Level::Info
    }
    fn log(&self, record: &Record) {
        if self.enabled(record.metadata()) {
            println!("{} - {}", record.level(), record.args());
        }
    }
    fn flush(&self) {}
}
```

除此之外，我们还需要像 `env_logger` 一样包装下 `set_logger` 和 `set_max_level`：

```
use log::{SetLoggerError, LevelFilter};
static LOGGER: SimpleLogger = SimpleLogger;
pub fn init() -> Result<(), SetLoggerError> {
    log::set_logger(&LOGGER)
        .map(|()| log::set_max_level(LevelFilter::Info))
}
```

## 更多示例

关于 `log` 门面库和具体的日志库还有更多的使用方式，详情请参见锈书的[开发者工具](#)一章。

# 使用 tracing 记录日志

严格来说，tracing 并不是一个日志库，而是一个分布式跟踪的 SDK，用于采集监控数据的。

随着微服务的流行，现在一个产品有多个系统组成是非常常见的，这种情况下，一条用户请求可能会横跨几个甚至几十个服务。此时再用传统的日志方式去跟踪这条用户请求就变得较为困难，这就是分布式跟踪在现代化监控系统中这么炙手可热的原因。

关于分布式追踪，在后面的监控章节进行详细介绍，大家只要知道：分布式追踪的核心就是在请求的开始生成一个 `trace_id`，然后将该 `trace_id` 一直往后透穿，请求经过的每个服务都会使用该 `trace_id` 记录相关信息，最终将整个请求形成一个完整的链路予以记录下来。

那么后面当要查询这次请求的相关信息时，只要使用 `trace_id` 就可以获取整个请求链路的所有信息了，非常简单好用。看到这里，相信大家也明白为什么这个库的名称叫 `tracing` 了吧？

至于为何把它归到日志库的范畴呢？因为 `tracing` 支持 `log` 门面库的 API，因此，它既可以作为分布式追踪的 SDK 来使用，也可以作为日志库来使用。

---

在分布式追踪中，`trace_id` 都是由 SDK 自动生成和往后透穿，对于用户的使用来说是完全透明的。如果你要手动用日志的方式来实现请求链路的追踪，那么就必须考虑 `trace_id` 的手动生成、透传，以及不同语言之间的协议规范等问题

---

## 一个简单例子

开始之前，需要先将 `tracing` 添加到项目的 `Cargo.toml` 中：

```
[dependencies]
tracing = "0.1"
```

注意，在写作本文时，0.2 版本已经快要出来了，所以具体使用的版本请大家以阅读时为准。

下面的例子中将同时使用 `log` 和 `tracing`：

```
use log;
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

fn main() {
    // 只有注册 subscriber 后，才能在控制台上看到日志输出
    tracing_subscriber::registry()
        .with(fmt::layer())
        .init();

    // 调用 `log` 包的 `info!`
    log::info!("Hello world");

    let foo = 42;
    // 调用 `tracing` 包的 `info!`
    tracing::info!(foo, "Hello from tracing");
}
```

可以看出，门面库的排场还是有的，`tracing` 在 API 上明显是使用了 `log` 的规范。

运行后，输出如下日志：

```
2022-04-09T14:34:28.965952Z INFO test_tracing: Hello world
2022-04-09T14:34:28.966011Z INFO test_tracing: Hello from tracing foo=42
```

还可以看出，`log` 的日志格式跟 `tracing` 一模一样，结合上一章节的知识，相信聪明的同学已经明白了这是为什么。

那么 `tracing` 跟 `log` 的具体日志实现框架有何区别呢？别急，我们再来接着看。

## 异步编程中的挑战

除了分布式追踪，在异步编程中使用传统的日志也是存在一些问题的，最大的挑战就在于异步任务的执行没有确定的顺序，那么输出的日志也将没有确定的顺序并混在一起，无法按照我们想要的逻辑顺序串联起来。

**归根到底，在于日志只能针对某个时间点进行记录，缺乏上下文信息，而线程间的执行顺序又是不确定的，因此日志就有些无能为力。**而 `tracing` 为了解决这个问题，引入了 `span` 的概念(这个概念也来自于分布式追踪)，一个 `span` 代表了一个时间段，拥有开始和结束时间，在此期间的所有类型数据、结构化数据、文本数据都可以记录其中。

大家发现了吗？`span` 是可以拥有上下文信息的，这样就能帮我们把信息按照所需的逻辑性串联起来了。

# 核心概念

tracing 中最重要的三个概念是 Span、Event 和 Collector，下面我们来一一简单介绍下。

## Span

相比起日志只能记录在某个时间点发生的事件，span 最大的意义就在于它可以记录一个过程，也就是在某一段时间内发生的事件流。既然是记录时间段，那自然有开始和结束：

```
use tracing::{span, Level};
fn main() {
    let span = span!(Level::TRACE, "my_span");

    // `enter` 返回一个 RAII，当其被 drop 时，将自动结束该 span
    let enter = span.enter();
    // 这里开始进入 `my_span` 的上下文
    // 下面执行一些任务，并记录一些信息到 `my_span` 中
    // ...
} // 这里 enter 将被 drop，`my_span` 也随之结束
```

## Event 事件

Event 代表了某个时间点发生的事件，这方面它跟日志类似，但是不同的是，Event 还可以产生在 span 的上下文中。

```
use tracing::{event, span, Level};
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

fn main() {
    tracing_subscriber::registry().with(fmt::layer()).init();
    // 在 span 的上下文之外记录一次 event 事件
    event!(Level::INFO, "something happened");

    let span = span!(Level::INFO, "my_span");
    let _guard = span.enter();

    // 在 "my_span" 的上下文中记录一次 event
    event!(Level::DEBUG, "something happened inside my_span");
}
```

```
2022-04-09T14:51:38.382987Z  INFO test_tracing: something happened
2022-04-09T14:51:38.383111Z DEBUG my_span: test_tracing: something happened inside
my_span
```

虽然 `event` 在哪里都可以使用，**但是最好只在 span 的上下文中使用**：用于代表一个时间点发生的事情，例如记录 HTTP 请求返回的状态码，从队列中获取一个对象，等等。

## Collector 收集器

当 `Span` 或 `Event` 发生时，它们会被实现了 `collect` 特征的收集器所记录或聚合。这个过程是通过通知的方式实现的：当 `Event` 发生或者 `Span` 开始/结束时，会调用 `Collect` 特征的[相应方法](#)通知 `Collector`。

### tracing-subscriber

我们前面提到只有使用了 `tracing-subscriber` 后，日志才能输出到控制台中。

之前大家可能还不理解，现在应该明白了，它是一个 `Collector`，可以将记录的日志收集后，再输出到控制台中。

## 使用方法

### span! 宏

`span!` 宏可以用于创建一个 `Span` 结构体，然后通过调用结构体的 `enter` 方法来开始，再通过超出作用域时的 `drop` 来结束。

```
use tracing::{span, Level};
fn main() {
    let span = span!(Level::TRACE, "my_span");

    // `enter` 返回一个 RAII，当其被 drop 时，将自动结束该 span
    let enter = span.enter();
    // 这里开始进入 `my_span` 的上下文
    // 下面执行一些任务，并记录一些信息到 `my_span` 中
    // ...
} // 这里 enter 将被 drop，`my_span` 也随之结束
```

## #[instrument]

如果想要将某个函数的整个函数体都设置为 span 的范围，最简单的方法就是为函数标记上 `# [instrument]`，此时 tracing 会自动为函数创建一个 span，span 名跟函数名相同，在输出的信息中还会自动带上函数参数。

```
use tracing::{info, instrument};
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

#[instrument]
fn foo(ans: i32) {
    info!("in foo");
}

fn main() {
    tracing_subscriber::registry().with(fmt::layer()).init();
    foo(42);
}
```

```
2022-04-10T02:44:12.885556Z  INFO foo{ans=42}: test_tracing: in foo
```

关于 `# [instrument]` 详细说明，请参见[官方文档](#)。

## in\_scope

对于没有内置 tracing 支持或者无法使用 `#instrument` 的函数，例如外部库的函数，我们可以使用 Span 结构体的 `in_scope` 方法，它可以将同步代码包裹在一个 span 中：

```
use tracing::info_span;

let json = info_span!("json.parse").in_scope(|| serde_json::from_slice(&buf))?;
```

## 在 `async` 中使用 span

需要注意，如果是在异步编程时使用，要避免以下使用方式：

```

async fn my_async_function() {
    let span = info_span!("my_async_function");

    // WARNING: 该 span 直到 drop 后才结束, 因此在 .await 期间, span 依然处于工作中状态
    let _enter = span.enter();

    // 在这里 span 依然在记录, 但是 .await 会让出当前任务的执行权, 然后运行时会去运行其它任务, 此
    // 时这个 span 可能会记录其它任务的执行信息, 最终记录了不正确的 trace 信息
    some_other_async_function().await

    // ...
}

```

我们建议使用以下方式, 简单又有效:

```

use tracing::{info, instrument};
use tokio::{io::AsyncWriteExt, net::TcpStream};
use std::io;

#[instrument]
async fn write(stream: &mut TcpStream) -> io::Result<usize> {
    let result = stream.write(b"hello world\n").await;
    info!("wrote to stream; success={:?}", result.is_ok());
    result
}

```

那有同学可能要问了, 是不是我们无法在异步代码中使用 `span.enter` 了, 答案是: 是也不是。

是, 你无法直接使用 `span.enter` 语法了, 原因上面也说过, 但是可以通过下面的方式来曲线使用:

```

use tracing::Instrument;

let my_future = async {
    // ...
};

my_future
    .instrument(tracing::info_span!("my_future"))
    .await

```

## span 嵌套

`tracing` 的 `span` 不仅仅是上面展示的基本用法, 它们还可以进行嵌套!

```
use tracing::{debug, info, span, Level};
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

fn main() {
    tracing_subscriber::registry().with(fmt::layer()).init();

    let scope = span!(Level::DEBUG, "foo");
    let _enter = scope.enter();
    info!("Hello in foo scope");
    debug!("before entering bar scope");
    {
        let scope = span!(Level::DEBUG, "bar", ans = 42);
        let _enter = scope.enter();
        debug!("enter bar scope");
        info!("In bar scope");
        debug!("end bar scope");
    }
    debug!("end bar scope");
}
```

```
INFO foo: log_test: Hello in foo scope
DEBUG foo: log_test: before entering bar scope
DEBUG foo:bar{ans=42}: log_test: enter bar scope
INFO foo:bar{ans=42}: log_test: In bar scope
DEBUG foo:bar{ans=42}: log_test: end bar scope
DEBUG foo: log_test: end bar scope
```

在上面的日志中，`foo:bar` 不仅包含了 `foo` 和 `bar` span 名，还显示了它们之间的嵌套关系。

## 对宏进行配置

### 日志级别和目标

`span!` 和 `event!` 宏都需要设定相应的日志级别，而且它们支持可选的 `target` 或 `parent` 参数( 只能二者选其一 )，该参数用于描述事件发生的位置，如果父 span 没有设置，`target` 参数也没有提供，那这个位置默认分别是当前的 span 和当前的模块。

```

use tracing::{debug, info, span, Level, event};
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

fn main() {
    tracing_subscriber::registry().with(fmt::layer()).init();

    let s = span!(Level::TRACE, "my span");
    // 没进入 span, 因此输出日志将不会带上 span 的信息
    event!(target: "app_events", Level::INFO, "something has happened 1!");

    // 进入 span (开始)
    let _enter = s.enter();
    // 没有设置 target 和 parent
    // 这里的对象位置分别是当前的 span 名和模块名
    event!(Level::INFO, "something has happened 2!");
    // 设置了 target
    // 这里的对象位置分别是当前的 span 名和 target
    event!(target: "app_events", Level::INFO, "something has happened 3!");

    let span = span!(Level::TRACE, "my span 1");
    // 这里就更为复杂一些, 留给大家作为思考题
    event!(parent: &span, Level::INFO, "something has happened 4!");
}

```

## 记录字段

我们可以通过语法 `field_name = field_value` 来输出结构化的日志

```

// 记录一个事件, 带有两个字段:
// - "answer", 值是 42
// - "question", 值是 "life, the universe and everything"
event!(Level::INFO, answer = 42, question = "life, the universe, and everything");

// 日志输出 -> INFO test_tracing: answer=42 question="life, the universe, and
everything"

```

## 捕获环境变量

还可以捕获环境中的变量:

```

let user = "ferris";

// 下面的简写方式
(Level::TRACE, "login", user);
// 等价于:
(Level::TRACE, "login", user = user);

use tracing::{info, span, Level};
use tracing_subscriber::{fmt, layer::SubscriberExt, util::SubscriberInitExt};

fn main() {
    tracing_subscriber::registry().with(fmt::layer()).init();

    let user = "ferris";
    let s = span!(Level::TRACE, "login", user);
    let _enter = s.enter();

    info!(welcome="hello", user);
    // 下面一行将报错，原因是这种写法是格式化字符串的方式，必须使用 info!("hello {}", user)
    // info!("hello", user);
}

// 日志输出 -> INFO login{user="ferris"}: test_tracing: welcome="hello" user="ferris"

```

## 字段名的多种形式

字段名还可以包含 . :

```

let user = "ferris";
let email = "ferris@rust-lang.org";
event!(Level::TRACE, user, user.email = email);

// 还可以使用结构体
let user = User {
    name: "ferris",
    email: "ferris@rust-lang.org",
};

// 直接访问结构体字段，无需赋值即可使用
(Level::TRACE, "login", user.name, user.email);

// 字段名还可以使用字符串
event!(Level::TRACE, "guid:x-request-id" = "abcdef", "type" = "request");

// 日志输出 ->
// TRACE test_tracing: user="ferris" user.email="ferris@rust-lang.org"
// TRACE test_tracing: user.name="ferris" user.email="ferris@rust-lang.org"
// TRACE test_tracing: guid:x-request-id="abcdef" type="request"

```

?

? 符号用于说明该字段将使用 `fmt::Debug` 来格式化。

```
# [derive(Debug)]
struct MyStruct {
    field: &'static str,
}

let my_struct = MyStruct {
    field: "Hello world!",
};

// `my_struct` 将使用 Debug 的形式输出
event!(Level::TRACE, greeting = ?my_struct);
// 等价于:
event!(Level::TRACE, greeting = tracing::field::debug(&my_struct));

// 下面代码将报错, my_struct 没有实现 Display
// event!(Level::TRACE, greeting = my_struct);

// 日志输出 -> TRACE test_tracing: greeting=MyStruct { field: "Hello world!" }
```

%

% 说明字段将用 `fmt::Display` 来格式化。

```
// `my_struct.field` 将使用 `fmt::Display` 的格式化形式输出
event!(Level::TRACE, greeting = %my_struct.field);
// 等价于:
event!(Level::TRACE, greeting = tracing::field::display(&my_struct.field));

// 作为对比, 大家可以看下 Debug 和正常的字段输出长什么样
event!(Level::TRACE, greeting = ?my_struct.field);
event!(Level::TRACE, greeting = my_struct.field);

// 下面代码将报错, my_struct 没有实现 Display
// event!(Level::TRACE, greeting = %my_struct);
```

```
2022-04-10T03:49:00.834330Z TRACE test_tracing: greeting=Hello world!
2022-04-10T03:49:00.834410Z TRACE test_tracing: greeting=Hello world!
2022-04-10T03:49:00.834422Z TRACE test_tracing: greeting="Hello world!"
2022-04-10T03:49:00.834433Z TRACE test_tracing: greeting="Hello world!"
```

## Empty

字段还能标记为 `Empty`，用于说明该字段目前没有任何值，但是可以在后面进行记录。

```
use tracing::{trace_span, field};

let span = trace_span!("my_span", greeting = "hello world", parting = field::Empty);

// ...

// 现在，为 parting 记录一个值
span.record("parting", &"goodbye world!");
```

## 格式化字符串

除了以字段的方式记录信息，我们还可以使用格式化字符串的方式(同 `println!`、`format!`)。

---

注意，当字段跟格式化的方式混用时，必须把格式化放在最后，如下所示

---

```
let question = "the ultimate question of life, the universe, and everything";
let answer = 42;
event!{
    Level::DEBUG,
    question.answer = answer,
    question.tricky = true,
    "the answer to {} is {}.", question, answer
};

// 日志输出 -> DEBUG test_tracing: the answer to the ultimate question of life, the
universe, and everything is 42. question.answer=42 question.tricky=true
```

## 文件输出

截至目前，我们上面的日志都是输出到控制台中。

针对文件输出，`tracing` 提供了一个专门的库 [tracing-appender](#)，大家可以查看官方文档了解更多。

## 一个综合例子

最后，再来看一个综合的例子，使用了 `color-eyre` 和文件输出，前者用于为输出的日志加上更易读的颜色。

```
use color_eyre::{eyre::eyre, Result};
use tracing::{error, info, instrument};
use tracing_appender::{non_blocking, rolling};
use tracing_error::ErrorLayer;
use tracing_subscriber::{
    filter::EnvFilter, fmt, layer::SubscriberExt, util::SubscriberInitExt, Registry,
};

#[instrument]
fn return_err() -> Result<()> {
    Err(eyre!("Something went wrong"))
}

#[instrument]
fn call_return_err() {
    info!("going to log error");
    if let Err(err) = return_err() {
        // 推荐大家运行下，看看这里的输出效果
        error!(&err, "error");
    }
}

fn main() -> Result<()> {
    let env_filter = EnvFilter::try_from_env().unwrap_or_else(|_| EnvFilter::new("info"));
    // 输出到控制台中
    let formatting_layer = fmt::layer().pretty().with_writer(std::io::stderr);

    // 输出到文件中
    let file_appender = rolling::never("logs", "app.log");
    let (non_blocking_appender, _guard) = non_blocking(file_appender);
    let file_layer = fmt::layer()
        .with_ansi(false)
        .with_writer(non_blocking_appender);

    // 注册
    Registry::default()
        .with(env_filter)
        // ErrorLayer 可以让 color-eyre 获取到 span 的信息
        .with(ErrorLayer::default())
        .with(formatting_layer)
        .with(file_layer)
        .init();

    // 安装 color-eyre 的 panic 处理句柄
    color_eyre::install()?;

    call_return_err();
    Ok(())
}
```

## 总结 & 推荐

至此，`tracing` 的介绍就已结束，相信大家都看得出，它比上个章节的 `log` 及兄弟们要更加复杂一些，一方面是因为它能更好的支持异步编程环境，另一方面就是它还是一个分布式追踪的库，对于后者，我们将在后续的监控章节进行讲解。

如果你让我推荐使用哪个，那我的建议是：

- 对于简单的工程，例如用于 POC（Proof of Concepts）目的，使用 `log` 即可
- 对于需要认真对待，例如生产级或优秀的开源项目，建议使用 `tracing` 的方式，一举解决日志和监控的后顾之忧

# 使用 tracing 输出自定义的 Rust 日志

在 [tracing](#) 包出来前，Rust 的日志也就 `log` 有一战之力，但是 `log` 的功能相对来说还是简单一些。在大名鼎鼎的 tokio 开发团队推出 `tracing` 后，我现在坚定的认为 `tracing` 就是未来！

---

截至目前，rust 编译器团队、GraphQL 都在使用 `tracing`，而且 tokio 在密谋一件大事：基于 `tracing` 开发一套终端交互式 debug 工具：[console](#)！

基于这种坚定的信仰，我们决定将公司之前使用的 `log` 包替换成 `tracing`，但是有一个问题：后者提供的 JSON logger 总感觉不是那个味儿。这意味着，对于程序员来说，最快乐的时光又要到来了：定制自己的开发工具。

好了，闲话少说，下面我们一起来看看该如何构建自己的 logger，以及深入了解 `tracing` 的一些原理，当然你也可以只选择来凑个热闹，总之，开始吧！

## 打地基(1)

首先，使用 `cargo new --bin test-tracing` 创建一个新的二进制类型( binary )的项目。

然后引入以下依赖：

```
# in cargo.toml

[dependencies]
serde_json = "1"
tracing = "0.1"
tracing-subscriber = "0.3"
```

其中 `tracing-subscriber` 用于订阅正在发生的事情、监控事件，然后可以对它们进行进一步的处理。`serde_json` 可以帮我们更好的处理格式化的 JSON，毕竟咱们要解决的问题就来自于 JSON logger。

下面来实现一个基本功能：设置自定义的 logger，并使用 `info!` 来打印一行日志。

```
// in examples/figure_0/main.rs

use tracing::info;
use tracing_subscriber::prelude::*;

mod custom_layer;
use custom_layer::CustomLayer;

fn main() {
    // 设置 `tracing-subscriber` 对 tracing 数据的处理方式
    tracing_subscriber::registry().with(CustomLayer).init();

    // 打印一条简单的日志。用 `tracing` 的行话来说，`info!` 将创建一个事件
    info!(a_bool = true, answer = 42, message = "first example");
}
```

大家会发现，上面引入了一个模块 `custom_layer`，下面从该模块开始，来实现我们的自定义 logger。首先，`tracing-subscriber` 提供了一个特征 `Layer` 专门用于处理 `tracing` 的各种事件(`span`, `event`)。

```
// in examples/figure_0/custom_layer.rs

use tracing_subscriber::Layer;

pub struct CustomLayer;

impl<S> Layer<S> for CustomLayer where S: tracing::Subscriber {}
```

由于还没有填入任何代码，运行该示例比你打的水漂还无力 - 毫无效果。

## 捕获事件

在 `tracing` 中，当 `info!`、`error!` 等日志宏被调用时，就会产生一个相应的事件 `Event`。

而我们首先，就要为之前的 `Layer` 特征实现 `on_event` 方法。

```
// in examples/figure_0/custom_layer.rs

where
    S: tracing::Subscriber,
{
    fn on_event(
        &self,
        event: &tracing::Event<'_>,
        _ctx: tracing_subscriber::layer::Context<'_, S>,
    ) {
        println!("Got event!");
        println!(" level={:?}", event.metadata().level());
        println!(" target={:?}", event.metadata().target());
        println!(" name={:?}", event.metadata().name());
        for field in event.fields() {
            println!(" field={}", field.name());
        }
    }
}
```

从代码中可以看出，我们打印了事件中包含的事件名、日志等级以及事件发生的代码路径。运行后，可以看到以下输出：

```
$ cargo run --example figure_1

Got event!
level=Level(Info)
target="figure_1"
name="event examples/figure_1/main.rs:10"
field=a_bool
field=answer
field=message
```

但是奇怪的是，我们无法通过 API 来获取到具体的 `field` 值。还有就是，上面的输出还不是 JSON 格式。

现在问题来了，要创建自己的 logger，不能获取 `filed` 显然是不靠谱的。

## 访问者模式

在设计上，`tracing` 作出了一个选择：永远不会自动存储产生的事件数据( spans, events )。如果我们要获取这些数据，就必须自己手动存储。

解决办法就是使用访问者模式(Visitor Pattern)：手动实现 `Visit` 特征去获取事件中的值。`Visit` 为每个 `tracing` 可以处理的类型都提供了对应的 `record_X` 方法。

```

// in examples/figure_2/custom_layer.rs

struct PrintlnVisitor;

impl tracing::field::Visit for PrintlnVisitor {
    fn record_f64(&mut self, field: &tracing::field::Field, value: f64) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_i64(&mut self, field: &tracing::field::Field, value: i64) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_u64(&mut self, field: &tracing::field::Field, value: u64) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_bool(&mut self, field: &tracing::field::Field, value: bool) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_str(&mut self, field: &tracing::field::Field, value: &str) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_error(
        &mut self,
        field: &tracing::field::Field,
        value: &(dyn std::error::Error + 'static),
    ) {
        println!(" field={} value={}", field.name(), value)
    }

    fn record_debug(&mut self, field: &tracing::field::Field, value: &dyn
std::fmt::Debug) {
        println!(" field={} value={:?}", field.name(), value)
    }
}

```

然后在之前的 `on_event` 中来使用这个新的访问者：`event.record(&mut visitor)` 可以访问其中的所有值。

```
// in examples/figure_2/custom_layer.rs

fn on_event(
    &self,
    event: &tracing::Event<'_>,
    _ctx: tracing_subscriber::layer::Context<'_, S>,
) {
    println!("Got event!");
    println!("  level={:?}", event.metadata().level());
    println!("  target={:?}", event.metadata().target());
    println!("  name={:?}", event.metadata().name());
    let mut visitor = PrintlnVisitor;
    event.record(&mut visitor);
}
```

这段代码看起来有模有样，来运行下试试：

```
$ cargo run --example figure_2

Got event!
  level=Level(Info)
  target="figure_2"
  name="event examples/figure_2/main.rs:10"
  field=a_bool value=true
  field=answer value=42
  field=message value=first example
```

Bingo！一切完美运行！

## 构建 JSON logger

目前为止，离我们想要的 JSON logger 只差一步了。下面来实现一个 `JsonVisitor` 替代之前的 `PrintlnVisitor` 用于构建一个 JSON 对象。

```
// in examples/figure_3/custom_layer.rs

impl<'a> tracing::field::Visit for JsonVisitor<'a> {
    fn record_f64(&mut self, field: &tracing::field::Field, value: f64) {
        self.0
            .insert(field.name().to_string(), serde_json::json!(value));
    }

    fn record_i64(&mut self, field: &tracing::field::Field, value: i64) {
        self.0
            .insert(field.name().to_string(), serde_json::json!(value));
    }

    fn record_u64(&mut self, field: &tracing::field::Field, value: u64) {
        self.0
            .insert(field.name().to_string(), serde_json::json!(value));
    }

    fn record_bool(&mut self, field: &tracing::field::Field, value: bool) {
        self.0
            .insert(field.name().to_string(), serde_json::json!(value));
    }

    fn record_str(&mut self, field: &tracing::field::Field, value: &str) {
        self.0
            .insert(field.name().to_string(), serde_json::json!(value));
    }

    fn record_error(
        &mut self,
        field: &tracing::field::Field,
        value: &(dyn std::error::Error + 'static),
    ) {
        self.0.insert(
            field.name().to_string(),
            serde_json::json!(value.to_string()),
        );
    }

    fn record_debug(&mut self, field: &tracing::field::Field, value: &dyn
std::fmt::Debug) {
        self.0.insert(
            field.name().to_string(),
            serde_json::json!(format!("{}: {:?}", value)),
        );
    }
}
```

```
// in examples/figure_3/custom_layer.rs

fn on_event(
    &self,
    event: &tracing::Event<'_>,
    _ctx: tracing_subscriber::layer::Context<'_, S>,
) {
    // Convert the values into a JSON object
    let mut fields = BTreeMap::new();
    let mut visitor = JsonVisitor(&mut fields);
    event.record(&mut visitor);

    // Output the event in JSON
    let output = serde_json::json!({
        "target": event.metadata().target(),
        "name": event.metadata().name(),
        "level": format!("{}:{}", event.metadata().level()),
        "fields": fields,
    });
    println!("{}", serde_json::to_string_pretty(&output).unwrap());
}
```

继续运行：

```
$ cargo run --example figure_3

{
  "fields": {
    "a_bool": true,
    "answer": 42,
    "message": "first example"
  },
  "level": "Level(Info)",
  "name": "event examples/figure_3/main.rs:10",
  "target": "figure_3"
}
```

终于，我们实现了自己的 logger，并且成功地输出了一条 JSON 格式的日志。并且新实现的 Layer 就可以添加到 tracing-subscriber 中用于记录日志事件。

下面再来一起看看如何使用 tracing 提供的 period-of-time spans 为日志增加更详细的上下文信息。

## 何为 span

在之前我们多次提到 span 这个词，但是何为 span？

不知道大家知道分布式追踪不？在分布式系统中每一个请求从开始到返回，会经过多个服务，这条请求路径被称为请求跟踪链路( trace )，可以看出，一条链路是由多个部分组成，我们可以简单的把其中一个部分认为是一个 span。

跟 log 是对某个时间点的记录不同，span 记录的是一个时间段。当程序开始执行一系列任务时，span 就会开始，当这一系列任务结束后，span 也随之结束。

由此可见，tracing 其实不仅仅是一个日志库，它还是一个分布式追踪的库，可以帮助我们采集信息，然后上传给 jaeger 等分布式追踪平台，最终实现对指定应用程序的监控。

在理解后，再来看看该如何为自定义的 logger 实现 spans。

## 打地基(2)

先来创建一个外部 span 和一个内部 span，从概念上来说，spans 和 events 创建的东东类似以下嵌套结构：

- 进入外部 span
  - 进入内部 span
    - 事件已创建，内部 span 是它的父 span，外部 span 是它的祖父 span
  - 结束内部 span
- 结束外部 span

---

有些同学可能还是不太理解，你就把 span 理解成为监控埋点，进入 span == 埋点开始，结束 span == 埋点结束

---

在下面的代码中，当使用 `span.enter()` 创建的 span 超出作用域时，将自动退出：根据 Drop 特征触发的顺序，`inner_span` 将先退出，然后才是 `outer_span` 的退出。

```
// in examples/figure_5/main.rs

use tracing::{debug_span, info, info_span};
use tracing_subscriber::prelude::*;

mod custom_layer;
use custom_layer::CustomLayer;

fn main() {
    tracing_subscriber::registry().with(CustomLayer).init();

    let outer_span = info_span!("outer", level = 0);
    let _outer_entered = outer_span.enter();

    let inner_span = debug_span!("inner", level = 1);
    let _inner_entered = inner_span.enter();

    info!(a_bool = true, answer = 42, message = "first example");
}
```

再回到事件处理部分，通过使用 `examples/figure_0/main.rs` 我们能获取到事件的父 span，当然，前提是它存在。但是在实际场景中，直接使用 `ctx.event_scope(event)` 来迭代所有 span 会更加简单好用。

注意，这种迭代顺序类似于栈结构，以上面的代码为例，先被迭代的是 `inner_span`，然后才是 `outer_span`。

当然，如果你不想以类似于出栈的方式访问，还可以使用 `scope.from_root()` 直接反转，此时的访问将从最外层开始：`outer -> inner`。

对了，为了使用 `ctx.event_scope()`，我们的订阅者还需实现 `LookupRef`。提前给出免责声明：这里的实现方式有些诡异，大家可能难以理解，但是..我们其实也无需理解，只要这么用即可。

---

译者注：这里用到了高阶生命周期 HRTB( Higher Rank Trait Bounds ) 的概念，一般的读者无需了解，感兴趣的可以看看[这里](https://doc.rust-lang.org/nomicon/hrtb.html)[<https://doc.rust-lang.org/nomicon/hrtb.html>]

```

// in examples/figure_5/custom_layer.rs

impl<S> Layer<S> for CustomLayer
where
    S: tracing::Subscriber,
    // 好可怕！还好我们不需要理解它，只要使用即可
    S: for<'lookup> tracing_subscriber::registry::LookupSpan<'lookup>,
{
    fn on_event(&self, event: &tracing::Event<'_>, ctx:
tracing_subscriber::layer::Context<'_, S>) {
        // 父 span
        let parent_span = ctx.event_span(event).unwrap();
        println!("parent span");
        println!("  name={}", parent_span.name());
        println!("  target={}", parent_span.metadata().target());

        println!();

        // 迭代范围内的所有的 spans
        let scope = ctx.event_scope(event).unwrap();
        for span in scope.from_root() {
            println!("an ancestor span");
            println!("  name={}", span.name());
            println!("  target={}", span.metadata().target());
        }
    }
}

```

运行下看看效果：

```

$ cargo run --example figure_5

parent span
  name=inner
  target=figure_5

an ancestor span
  name=outer
  target=figure_5
an ancestor span
  name=inner
  target=figure_5

```

细心的同学可能会发现，这里怎么也没有 field 数据？没错，而且恰恰是这些 field 包含的数据才让日志和监控有意义。那我们可以像之前一样，使用访问器 Visitor 来解决吗？

## span 的数据在哪里

答案是：No。因为 `ctx.event_scope` 返回的东东没有任何办法可以访问其中的字段。

不知道大家还记得我们为何之前要使用访问器吗？很简单，因为 `tracing` 默认不会去存储数据，既然如此，那 `span` 这种跨了某个时间段的，就更不可能去存储数据了。

现在只能看看 `Layer` 特征有没有提供其它的方法了，哦呦，发现了一个 `on_new_span`，从名字可以看出，该方法是在 `span` 创建时调用的。

```
// in examples/figure_6/custom_layer.rs

impl<S> Layer<S> for CustomLayer
where
    S: tracing::Subscriber,
    S: for<'lookup> tracing_subscriber::registry::LookupSpan<'lookup>,
{
    fn on_new_span(
        &self,
        attrs: &tracing::span::Attributes<'_>,
        id: &tracing::span::Id,
        ctx: tracing_subscriber::layer::Context<'_, S>,
    ) {
        let span = ctx.span(id).unwrap();
        println!("Got on_new_span!");
        println!("  level={:?}", span.metadata().level());
        println!("  target={:?}", span.metadata().target());
        println!("  name={:?}", span.metadata().name());

        // Our old friend, `println!` exploration.
        let mut visitor = PrintlnVisitor;
        attrs.record(&mut visitor);
    }
}

$ cargo run --example figure_6
Got on_new_span!
  level=Level(Info)
  target="figure_7"
  name="outer"
  field=level value=0
Got on_new_span!
  level=Level(Debug)
  target="figure_7"
  name="inner"
  field=level value=1
```

芜湖! 我们的数据回来了! 但是这里有一个隐患: 只能在创建的时候去访问数据。如果仅仅是为了记录 spans, 那没什么大问题, 但是如果我们随后需要记录事件然后去尝试访问之前的 span 呢? 此时 span 的数据已经不存在了!

如果 `tracing` 不能存储数据, 那我们这些可怜的开发者该怎么办?

## 自己存储 span 数据

何为一个优秀的程序员? 能偷懒的时候绝不多动半跟手指, 但是需要勤快的时候, 也是自己动手丰衣足食的典型。

因此, 既然 `tracing` 不支持, 那就自己实现吧。先确定一个目标: 捕获 span 的数据, 然后存储在某个地方以便后续访问。

好在 `tracing-subscriber` 提供了扩展 `extensions` 的方式, 可以让我们轻松地存储自己的数据, 该扩展甚至可以跟每一个 span 联系在一起!

虽然我们可以把之前见过的 `BTreeMap<String, serde_json::Value>` 存在扩展中, 但是由于扩展数据是被 `registry` 中的所有 `layers` 所共享的, 因此出于私密性的考虑, 还是只保存私有字段比较合适。这里使用一个 `newtype` 模式来创建新的类型:

```
// in examples/figure_8/custom_layer.rs

#[derive(Debug)]
struct CustomFieldStorage(BTreeMap<String, serde_json::Value>);
```

每次发现一个新的 span 时, 都基于它来构建一个 JSON 对象, 然后将其存储在扩展数据中。

```
// in examples/figure_8/custom_layer.rs

fn on_new_span(
    &self,
    attrs: &tracing::span::Attributes<'_>,
    id: &tracing::span::Id,
    ctx: tracing_subscriber::layer::Context<'_, S>,
) {
    // 基于 field 值来构建我们自己的 JSON 对象
    let mut fields = BTreeMap::new();
    let mut visitor = JsonVisitor(&mut fields);
    attrs.record(&mut visitor);

    // 使用之前创建的 newtype 包裹下
    let storage = CustomFieldStorage(fields);

    // 获取内部 span 数据的引用
    let span = ctx.span(id).unwrap();
    // 获取扩展，用于存储我们的 span 数据
    let mut extensions = span.extensions_mut();
    // 存储!
    extensions.insert::<CustomFieldStorage>(storage);
}
```

这样，未来任何时候我们都可以取到该 span 包含的数据( 例如在 `on_event` 方法中 )。

```
// in examples/figure_8/custom_layer.rs

fn on_event(&self, event: &tracing::Event<'_>, ctx: tracing_subscriber::layer::Context<'_, S>) {
    let scope = ctx.event_scope(event).unwrap();
    println!("Got event!");
    for span in scope.from_root() {
        let extensions = span.extensions();
        let storage = extensions.get::<CustomFieldStorage>().unwrap();
        println!(" span");
        println!("   target={:?}", span.metadata().target());
        println!("   name={:?}", span.metadata().name());
        println!("   stored fields={:?}", storage);
    }
}
```

## 功能齐全的 JSON logger

截至目前，我们已经学了不少东西，下面来利用这些知识实现最后的 JSON logger。

```
// in examples/figure_9/custom_layer.rs

fn on_event(&self, event: &tracing::Event<'_>, ctx: tracing_subscriber::layer::Context<'_, S>) {
    // All of the span context
    let scope = ctx.event_scope(event).unwrap();
    let mut spans = vec![];
    for span in scope.from_root() {
        let extensions = span.extensions();
        let storage = extensions.get::<CustomFieldStorage>().unwrap();
        let field_data: &BTreeMap<String, serde_json::Value> = &storage.0;
        spans.push(serde_json::json!({
            "target": span.metadata().target(),
            "name": span.name(),
            "level": format!("{}:?}", span.metadata().level()),
            "fields": field_data,
        }));
    }
}

// The fields of the event
let mut fields = BTreeMap::new();
let mut visitor = JsonVisitor(&mut fields);
event.record(&mut visitor);

// And create our output
let output = serde_json::json!({
    "target": event.metadata().target(),
    "name": event.metadata().name(),
    "level": format!("{}:?", event.metadata().level()),
    "fields": fields,
    "spans": spans,
});
println!("{}", serde_json::to_string_pretty(&output).unwrap());
}
```

```

$ cargo run --example figure_9

{
  "fields": {
    "a_bool": true,
    "answer": 42,
    "message": "first example"
  },
  "level": "Level(Info)",
  "name": "event examples/figure_9/main.rs:16",
  "spans": [
    {
      "fields": {
        "level": 0
      },
      "level": "Level(Info)",
      "name": "outer",
      "target": "figure_9"
    },
    {
      "fields": {
        "level": 1
      },
      "level": "Level(Debug)",
      "name": "inner",
      "target": "figure_9"
    }
  ],
  "target": "figure_9"
}

```

嗯，完美。

## 等等，你说功能齐全？

上面的代码在发布到生产环境后，依然运行地相当不错，但是我发现还缺失了一个功能：span 在创建之后，依然要能记录数据。

```

// in examples/figure_10/main.rs

let outer_span = info_span!("outer", level = 0, other_field = tracing::field::Empty);
let _outer_entered = outer_span.enter();
// Some code...
outer_span.record("other_field", &7);

```

如果基于之前的代码运行上面的代码，我们将不会记录 `other_field`，因为该字段在收到 `on_new_span` 事件时，还不存在。

对此，Layer 提供了 on\_record 方法：

```
// in examples/figure_10/custom_layer.rs

fn on_record(
    &self,
    id: &tracing::span::Id,
    values: &tracing::span::Record<'_>,
    ctx: tracing_subscriber::layer::Context<'_, S>,
) {
    // 获取正在记录数据的 span
    let span = ctx.span(id).unwrap();

    // 获取数据的可变引用，该数据是在 on_new_span 中创建的
    let mut extensions_mut = span.extensions_mut();
    let custom_field_storage: &mut CustomFieldStorage =
        extensions_mut.get_mut::<CustomFieldStorage>().unwrap();
    let json_data: &mut BTreeMap<String, serde_json::Value> = &mut
    custom_field_storage.0;

    // 使用我们的访问器老朋友
    let mut visitor = JsonVisitor(json_data);
    values.record(&mut visitor);
}
```

终于，在最后，我们拥有了一个功能齐全的自定义的 JSON logger，大家快去尝试下吧。当然，你也可以根据自己的需求来定制专属于你的 logger，毕竟方法是一通百通的。

---

在以下 github 仓库，可以找到完整的代码: <https://github.com/bryanburgers/tracing-blog-post>

本文由 Rustt 提供翻译 原文链接:

<https://github.com/studyrs/Rustt/blob/main/Articles/%5B2022-04-07%5D%20在%20Rust%20中使用%20tracing%20自定义日志.md>

---

# 监控

监控是一个很大的领域，大到老板、前端开发、后端开发理解的监控可能都不相同。

- 老板眼中的监控：业务大数据实时展示
- 前端眼中的监控：手机 APP 收集上来的异常、崩溃、用户操作日志等
- 后端眼中的监控：请求链路跟踪、一段时间内的请求错误率、QPS 过高、异常日志等

正是因为这些复杂性，导致很多同学难以准确的说出监控到底是什么。

下面，我们将试图解释清楚监控的概念，并引入一个全新的概念：可观测性。

# 可观测性

在监控章节的[引言](#)中，我们提到了老板、前端、后端眼中的监控是各不相同的，那么有没有办法将监控模型进行抽象、统一呢？

来简单分析一下：

- 业务指标实时展示，这是一个指标型的数据( metric )
- 手机 APP 上传的数据，包含了日志( log )和指标类型( metric )，如果考虑到 APP 作为一次 HTTP 请求的发起端，那还涉及到请求链路的跟踪( trace )
- 后端链路跟踪是 trace，请求错误率、QPS 是 metric，异常日志是 log

喔，好像线索很明显哎，我们貌似可以把监控模型分为三种：指标 metric、日志 log 和链路 trace。

先别急，我们对总结出来的三种类型进行下对比，看看彼此之间是否存在关联性( 良好的模型设计，彼此之间应该是无关联的 )：

- 指标：用于表示在某一段时间内，一个行为出现的次数和分布
- 日志：记录在某一个时间点发生的一次事件
- 链路：记录一次请求所经过的完整的服务链路，可能会横跨线程、进程，也可能会横跨服务( 分布式、微服务 )

按照这个定义来看，三种类型几乎没有关联性，是不是意味着我们的监控模型非常成功？

恭喜你，刚才总结出的监控模型正是这几年非常火热的可观测性监控的三大基础：Metrics / Log / Trace。

## 各自为战的三种模型

但是如果按照这个模型，我们将监控分成三个部分开发，彼此没有关联，并且在使用之时，也带着孤立的观点去看待这些数据和功能，那可观测性就失去了其应有的意义。

例如要看指标趋势变化就使用 metrics，查看详细问题使用 log，要看请求链路、链路各部分的耗时、服务依赖都使用 trace，虽然看起来很美好，但是它们都在各自为战。

例如一个很常见的场景，现在我们通过 metrics 获得了一个告警，发现某个服务的 SLA 降低、错误率上升，此时该如何排查错误原因？查看日志？你如何确保日志跟错误率上升有内在的联系呢？而且一个大型服务，它的各种类型的日志、错误都是非常频繁的，要大海捞针般地找出特定的日志，非常难。

由于缺乏数据模型上的关联，最后只能各自为战：发现了错误率上升，就人工去找日志和链路，运气好，就能很快地查明原因，运气不好？等待老板和用户的咆哮吧

这个过程很不美好，需要工程师们充分理解每一项数据的底层逻辑，而在大型微服务架构中，没有一个工程师可以清晰的知道所有的底层逻辑，此时就需要分工协作去排查，那问题处理的复杂度和挑战性最终会急剧增加。

## 模型纽带

看来，要解决这个问题，我们需要一个纽带，来把三个模型串联起来，目前来看，trace 是最适合的。

因为问题的跟踪和解决其实就是沿着数据的流向来的，我们只要在 trace 流动的过程中，在沿途把相关的 log 收集上来，然后再针对收到的各种 trace，根据其标签去统计相应的指标。

这样，是不是就成功地将三个模型关联在了一起？而且还不是强扭的瓜！

再回到之前假设的场景：当我们对某个 Metric 波动发生兴趣时，可以直接将造成此波动的 Trace 关联检索出来，然后查看这些 Trace 在各个微服务中的所有执行细节，最后发现是底层某个微服务在执行请求过程中发生了 Panic，这个错误不断向上传播导致了服务对外 SLA 下降。

如果可观测平台做得更完善一些，将微服务的变更事件数据也呈现出来，那么一个工程师就可以快速完成整个排障和根因定位的过程，甚至不需要人，通过机器就可以自动完成整个排障和根因定位过程。

看到这里，相信大家都已经明白了 trace 的重要性以及可观测性监控到底优秀在哪里。那么问题来了，该如何落地？

## 数据采集

首先，没有数据，就没有一切，因此我们需要先把监控数据采集上来。

除了跨服务的数据统一规范外，由于现在的微服务往往使用多种语言实现，我们的数据采集还要支持不同的语言，选择一个合适的数据采集 SDK 就成了重中之重。

目前来说，我们最推荐大家采用 [OpenTelemetry](#) 作为可观测性解决方案，它提供了完整的数据协议规范、API 和多语言采集 SDK，我们将在下个章节进行详细介绍。

## 数据处理和存储

虽然在我们之前的模型设计完善后，数据彼此之间存在内在关联性，但是不代表它们就能够按照同样的格式来存储了，甚至都无法保证使用同一个数据库来存储。

就目前而言，对于三种模型的数据处理和存储推荐如下：

- Trace，使用 jaeger 接收采集上来的 trace 数据，经过处理后存储到一个分布式数据库中，例如 cassandra、scyllaDB 等
- Log，如果对日志的关键词索引有较高的要求，还是建议使用 ElasticeSearch，如果可以提前在日志中通过 kv 的形式打上标签，然后未来也只需要通过标签来索引，那可以考虑使用 loki
- Metrics，啥都不用说了，prometheus 走起，当然还可以使用 influxdb，后者正在使用 Rust 重写，期待未来的一飞冲天

## 数据查询和展示

大家知道可观测性现在为什么很多人搞不清楚吗？就是因为你怎么做都可以，比如之前的存储，就有很多解决方案，而且还都不错。

对于数据展示也是，你可以使用上面的 jaeger、prometheus 自带的 UI，也可以使用 grafana 这种统一性的 UI，而从我个人来说，更推荐使用 grafana，毕竟 UI 的统一性和内联性对于监控数据的查询是非常重要的。

再说了，grafana 的 UI 做的好看啊，没人能拒绝美好的事物吧 :D

好了，一篇口水文终于结束了，在后续章节我们将学习如何使用 OpenTelemetry + Jaeger + Prometheus + Grafana 搭建一套可用的监控服务，先来看看如何搭建和使用分布式追踪监控。

---

"tracing 呢？你这个监控服务怎么没有它的身影，日志章节口口声声的爱，现在就忘记了吗？"

"别急，我还记得呢，先卖个关子"

---

# 分布式追踪

# Rust最佳实践

对于生产级项目而言，运行稳定性和可维护性是非常重要的，本章就一起来看看 Rust 项目有哪些最佳实践准则。

# 日常开发三方库精选

对计算机、编程、架构的理解决定一个程序员的上限，而工具则决定了他的下限，三尺森寒利剑在手，问世间谁敢一战。

本文就分门别类的精心挑选了一些非常适合日常开发使用的三方库，同时针对优缺点、社区活跃等进行了评价，同一个类别的库，按照**推荐度优先级降序排列**，希望大家能喜欢。

---

本文节选自[Cook Rust](#)

---

## 目录

- 日常开发常用的Rust库:
  - [Web/HTTP, SQL客户端, NoSql客户端, 网络通信协议, 异步网络编程](#)
  - [服务发现, 消息队列, 搜索引擎](#)
  - [编解码, Email, 常用正则模版](#)
  - [日志监控, 代码Debug, 性能优化](#)
- [精选中文学习资料](#)
- [精选Rust开源项目](#)

## 日常开发常用Rust库

### Web/HTTP

- HTTP客户端
  - [reqwest](#) 一个简单又强大的HTTP客户端，[reqwest](#) 是目前使用最多的HTTP库
- Web框架
  - [axum](#) 基于Tokio和Hyper打造，模块化设计较好，目前口碑很好，值得使用Ergonomic and modular web framework built with Tokio, Tower, and Hyper
  - [Rocket](#) 功能强大，API简单的Web框架，但是主要开发者目前因为个人原因无法进行后续开发，未来存在不确定性

- [actix-web](#) 性能极高的Web框架，就是团队内部有些问题，未来存在一定的不确定性
- 总体来说，上述三个web框架都有很深的用户基础，其实都可以选用，如果让我推荐，顺序如下：`axum > Rocket > actix-web`。不过如果你不需要多么完善的web功能，只需要一个性能极高的http库，那么 [actix-web](#) 是非常好的选择，它的性能非常非常高！

## 日志监控

- 日志 [[crates.io](#)] [[github](#)]
  - [tokio-rs/tracing](#) 强大的日志框架，同时还支持OpenTelemetry格式，无缝打通未来的监控
  - [rust-lang/log](#) 官方日志库，事实上的API标准，但是三方库未必遵循
  - [estk/log4rs](#) 模仿JAVA `logback` 和 `log4j` 实现的日志库，可配置性较强
  - 在其它文章中，也许会推荐 `slog`，但是我们不推荐，一个是因为近半年未更新，一个是 `slog` 自己也推荐使用 `tracing`。
- 监控
  - [OpenTelemetry](#) 是现在非常火的可观测性解决方案，提供了协议、API、SDK等核心工具，用于收集监控数据，最后将这些metrics/logs/traces数据写入到 `prometheus`，`jaeger` 等监控平台中。主要是，它后台很硬，后面有各大公司作为背书，未来非常看好！
  - [vectordotdev/vector](#) 一个性能很高的数据采集agent，采集本地的日志、监控等数据，发送到远程的kafka、jaeger等数据下沉端，它最大的优点就是能从多种数据源(包括 OpenTelemetry)收集数据，然后推送到多个数据处理或者存储等下沉端。

## SQL客户端

- 通用
  - [launchbadge/sqlx](#) 异步实现、高性能、纯Rust代码的SQL库，支持 `PostgreSQL`，`MySQL`，`SQLite`，和 `MSSQL`。
- ORM
  - [rbatis/rbatis](#) 国内团队开发的ORM，异步、性能高、简单易上手
  - [diesel-rs/diesel](#) 安全、扩展性强的Rust ORM库，支持 `MySQL`、`PostgreSQL`、`SQLite`
- MySQL
  - [blackbeam/rust-mysql-simple](#) 纯Rust实现的MySQL驱动，提供连接池
  - [blackbeam/mysql\\_async](#) 基于Tokio实现的异步MySQL驱动
  - 上面两个都是一个团队出品，前者文档更全、star更多，建议使用前者
- PostgreSQL

- [sfackler/rust-postgres](#) 纯Rust实现的PostgreSQL客户端
- SQLite
  - [rusqlite](#) 用于Sqlite3的Rust客户端

## NoSQL客户端

- Redis
  - [mitsuhiko/redis-rs](#) 虽然最近更新不太活跃，但是它依然是最好的Redis客户端，说实话，我期待更好的，可能这也是Rust生态的未来可期之处吧
- Cassandra
  - [krojew/cdrs-tokio](#) [cdrs-tokio] 生产可用的Cassandra客户端，异步、纯Rust实现，就是个人项目 + star较少，未来不确定会不会不维护
  - [scylla-rust-driver](#) ScyllaDB提供的官方库，支持cql协议，由于背靠大山，未来非常可期
- MongoDB
  - [mongodb/mongo-rust-driver](#) 官方MongoDB客户端，闭着眼睛选就对了

## 分布式

### 服务发现

- [luncj/etcd-rs](#) 异步实现的Rust etcd客户端，优点是有一定的文档、作者较为活跃，意味着你提问题他可能会回答，不过，如果你不放心，还是考虑使用HTTP的方式访问etcd

### 消息队列

- Kafka
  - [fede1024/rust-rdkafka](#) Rust Kafka客户端，基于C版本的Kafka库[librdkafka]实现，文档较全、功能较为全面
  - [kafka-rust/kafka-rust](#) 相比上一个库，它算是纯Rust实现，文档还行，支持Kafka0.8.2及以后的版本，但是对于部分0.9版本的特性还不支持。同时有一个问题：最初的作者不维护了，转给了现在的作者，但是感觉好像也不是很活跃
- Nats

- [nats-io/nats.rs](#) Nats官方提供的客户端

## 网络、通信协议

- WebSocket
  - [snapview/tokio-tungstenite](#) 更适合Web应用使用的生产级Websocket库，它是异步非阻塞的，基于下面的 `tungstenite-rs` 库和tokio实现
  - [rust-websocket](#) 老牌Websocket库，提供了客户端和服务器端实现，但是。。。很久没更新了
  - [snapview/tungstenite-rs](#) 轻量级的Websocket流实现，该库更偏底层，例如，你可以用来构建其它网络库
- gRPC
  - [hyperium/tonic](#) 纯Rust实现的gRPC客户端和服务器端，支持async/await异步调用，文档和示例较为清晰
  - [tikv/grpc-rs](#) 国产开源之光TiDB团队出品的gRPC框架，基于C的代码实现，就是最近好像不是很活跃
  - 其实这两个实现都很优秀，把 `tonic` 放在第一位，主要是因为它是纯Rust实现，同时社区也更为活跃，但是并不代表它比 `tikv` 的更好！
- QUIC
  - [cloudflare/quiche](#) 大名鼎鼎 `cloudflare` 提供的QUIC实现，据说在公司内部重度使用，有了大规模生产级别的验证，非常值得信任，同时该库还实现了HTTP/3
  - [quinn-rs/quinn](#) 提供异步API调用，纯Rust实现，同时提供了几个有用的网络库
- MQTT
  - [bytebeamio/rumqtt](#) MQTT3.1.1/5协议库，同时实现了客户端与服务器端broker
  - [ntex-rs/ntex-mqtt](#) 客户端与服务端框架，支持MQTT3.1.1与5协议
  - [eclipse/paho.mqtt.rust](#) 老牌MQTT框架，对MQTT支持较全，其它各语言的实现也有

## 异步网络编程

- [tokio-rs/tokio](#) 最火的异步网络库，除了复杂上手难度高一些外，没有其它大的问题。同时tokio团队提供了多个非常优秀的Rust库，整个生态欣欣向荣，用户认可度很高
- [async-std](#) 跟标准库API很像的异步网络库，相对简单易用，但是貌似开发有些停滞，还有就是功能上不够完善。但是对于普通用户来说，这个库非常值得一试，它在功能和简单易用上取得了很好的平衡

- [actix](#) 基于Actor模型的异步网络库，但这个库的开发貌似已经停滞，他们团队一直在专注于 [actix-web](#) 的开发
- [mio](#) 严格来说，MIO与之前三个不是同一个用途的，MIO = Meta IO，是一个底层IO库，往往用于构建其它网络库，当然如果你对应用网络性能有非常极限的要求，可以考虑它，因为它的层次比较低，所带来的抽象负担小，所以性能损耗小
- 如果你要开发生产级别的项目，我推荐使用 [tokio](#)，稳定可靠，功能丰富，控制粒度细；自己的学习项目或者没有那么严肃的开源项目，我推荐 [async-std](#)，简单好用，值得学习；当你确切知道需要Actor网络模型时，就用 [actix](#)

## 搜索引擎

- ElasticSearch客户端
  - [elastic/elasticsearch](#) 官方es客户端，目前第三方的基本都处于停滞状态，所以不管好坏，用呗
- Rust搜索引擎
  - [Tantivy](#) Tantivy是Rust实现的本地搜索库，功能对标 [lucene](#)，如果你不需要分布式，那么引入tantivy作为自己本地Rust服务的一个搜索，是相当不错的选择，该库作者一直很活跃，而且最近还创立了搜索引擎公司，感觉大有作为。该库的优点在于纯Rust实现，性能高(lucene的2-3倍)，资源占用低(对比java自然不是一个数量级)，社区活跃。
- Rust搜索平台
  - [quickwit](#) 对标ElasticSearch，一个通用目的的分布式搜索平台，目前还在起步阶段(0.2版本)，未来非常可期，目前还不建议使用
  - [MeiliSearch](#) 虽然也是一个搜索平台，但是并不是通用目的的，MeiliSearch 目标是为终端用户提供边输入边提示的即刻搜索功能，因此是一个轻量级搜索平台，不适用于数据量大的搜索目的。总之，如果你需要在网页端或者APP为用户提供一个搜索条，然后支持输入容错、前缀搜索时，就可以使用它。

## 代码Debug

- GDB
  - [gdbgui](#) 提供浏览器支持的gdb debug工具，支持C, C++, Rust和Go.
- LLDB
  - [CodeLLDB](#) 专门为VSCode设计的LLDB Debug扩展

## 性能优化

- [bheisler/criterion.rs](#) 比官方提供的benchmark库更好，目前已经成为事实上标准的性能测试工具
- [Bytehound](#) Linux下的内存分析工具，可以用来分析：内存泄漏、内存分配、调用栈追踪，甚至它还有一个浏览器UI！懂的人都懂，性能测试工具的UI服务是多么稀缺和珍贵！
- [llogiq/flame](#) 专为Rust打造的火焰图分析工具，可以告诉你程序在哪些代码上花费的时间过多，非常适合用于代码性能瓶颈的分析。与 `perf` 不同，`flame` 库允许你自己定义想要测试的代码片段，只需要在代码前后加上相应的指令即可，非常好用
- [sharkdp/hyperfine](#) 一个命令行benchmark工具，支持任意shell命令，支持缓存清除、预热、多次运行统计分析等，尽量保证结果的准确性

## 编解码

- [Serde](#) 一个超高性能的通用序列化/反序列化框架，可以跟多种协议的库联合使用，实现统一编解码格式
- CSV
  - [BurntSushi/rust-csv](#) 高性能CSV读写库，支持[Serde](#)
- JSON
  - [serde-rs/json](#) 快到上天的JSON库，也是Rust事实上的标准JSON库，你也可以使用它的大哥 [Serde](#)，一个更通用的序列化/反序列化库
- MsgPack
  - [3Hren/msgpack-rust](#) 纯Rust实现的MessagePack编解码协议
- Protocol Buffers
  - [tokio-rs/prost](#) tokio出品，基本都属精品，此库也不例外，简单易用，文档详细
  - [stepancheg/rust-protobuf](#) 纯Rust实现
- TOML
  - [alexcrichton/toml-rs](#) TOML编码/解码，可以配合 `serde` 使用
- XML
  - [tafia/quick-xml](#) 高性能XML库，可以配合 `serde` 使用，文档较为详细
- YAML

- [dtolnay/serde-yaml](#) 使用 `serde` 编解码 YAML 格式的数据

## Email

- [lettre/lettre](#) — Rust SMTP库

## 常用正则模版

# 命名规范

基本的 Rust 命名规范在 [RFC 430](#) 中有描述。

通常，对于 **type-level** 的构造 Rust 倾向于使用**驼峰命名法**，而对于 **value-level** 的构造使用**蛇形命名法**。详情如下：

条目	惯例
包 Crates	unclear
模块 Modules	snake_case
类型 Types	UpperCamelCase
特征 Traits	UpperCamelCase
枚举 Enumerations	UpperCamelCase
结构体 Structs	UpperCamelCase
函数 Functions	snake_case
方法 Methods	snake_case
通用构造器 General constructors	new or with_more_details
转换构造器 Conversion constructors	from_some_other_type
宏 Macros	snake_case!
局部变量 Local variables	snake_case
静态类型 Statics	SCREAMING_SNAKE_CASE
常量 Constants	SCREAMING_SNAKE_CASE
类型参数 Type parameters	UpperCamelCase，通常使用一个大写字母: T
生命周期 Lifetimes	通常使用小写字母: 'a, 'de, 'src
Features	unclear but see <a href="#">C-FEATURE</a>

对于**驼峰命名法**，复合词的缩略形式我们认为是一个单独的词语，所以**只对首字母进行大写**：使用 `Uuid` 而不是 `UUID`，`Usize` 而不是 `usize`，`Stdin` 而不是 `stdin`。

对于**蛇形命名法**，缩略词用全小写：`is_xid_start`。

对于**蛇形命名法**（包括全大写的 `SCREAMING_SNAKE_CASE`），除了最后一部分，其它部分的词语都不能由单个字母组成：`btree_map` 而不是 `b_tree_map`，`PI_2` 而不是 `PI2`。

包名**不应该**使用 `-rs` 或者 `-rust` 作为后缀，因为每一个包都是 Rust 写的，因此这种多余的注释其实没有任何意义。

## 特征命名

特征的名称应该使用动词，而不是形容词或者名词，例如 `Print` 和 `Draw` 明显好于 `Printable` 和 `Drawable`。

## 类型转换要遵守 `as_`, `to_`, `into_` 命名惯例(C-CONV)

类型转换应该通过方法调用的方式实现，其中的前缀规则如下：

方法前缀	性能开销	所有权改变
<code>as_</code>	Free	borrowed -> borrowed
<code>to_</code>	Expensive	borrowed -> borrowed borrowed -> owned (non-Copy types) owned -> owned (Copy types)
<code>into_</code>	Variable	owned -> owned (non-Copy types)

例如：

- `str::as_bytes()` 把 `str` 变成 UTF-8 字节数组，性能开销是 0。输入是一个借用的 `&str`，输出也是一个借用的 `&str`
- `Path::to_str` 会执行一次昂贵的 UTF-8 字节数组检查，输入和输出都是借用的。对于这种情况，如果把方法命名为 `as_str` 是不正确的，因为这个方法的开销还挺大
- `str::to_lowercase()` 在调用过程中会遍历字符串的字符，且可能会分配新的内存对象。输入是一个借用的 `str`，输出是一个有独立所有权的 `String`
- `String::into_bytes()` 返回 `String` 底层的 `Vec<u8>` 数组，转换本身是零消耗的。该方法获取 `String` 的所有权，然后返回一个新的有独立所有权的 `Vec<u8>`

当一个单独的值被某个类型所包装时，访问该类型的内部值应通过 `into_inner()` 方法来访问。例如将一个缓冲区值包装为 `BufReader` 类型，还有 `GzDecoder`、`AtomicBool` 等，都是这种类型。

如果 `mut` 限定符在返回类型中出现，那么在命名上也应该体现出来。例如，`Vec::as_mut_slice` 就说明它返回了一个 `mut` 切片，在这种情况下 `as_mut_slice` 比 `as_slice_mut` 更适合。

```
// 返回类型是一个 `mut` 切片
fn as_mut_slice(&mut self) -> &mut [T];
```

## 标准库中的一些例子

- `Result::as_ref`
- `RefCell::as_ptr`
- `slice::to_vec`
- `Option::into_iter`

## 读访问器(Getter)的名称遵循 Rust 的命名规范(C-GETTER)

除了少数例外，在 Rust 代码中 `get` 前缀不用于 Getter。

```
pub struct S {
    first: First,
    second: Second,
}

impl S {
    // 而不是 get_first
    pub fn first(&self) -> &First {
        &self.first
    }

    // 而不是 get_first_mut, get_mut_first, or mut_first
    pub fn first_mut(&mut self) -> &mut First {
        &mut self.first
    }
}
```

至于上文提到的少数例外，如下：当有且仅有一个值能被 Getter 所获取时，才使用 `get` 前缀。例如，`Cell::get` 能直接访问到 `Cell` 中的内容。

有些 Getter 会在过程中执行运行时检查，那么我们就可以考虑添加 `_unchecked` Getter 函数，这个函数虽然不安全，但是往往具有更高的性能。典型的例子如下：

```
fn get(&self, index: K) -> Option<&V>;
fn get_mut(&mut self, index: K) -> Option<&mut V>;
unsafe fn get_unchecked(&self, index: K) -> &V;
unsafe fn get_unchecked_mut(&mut self, index: K) -> &mut V;
```

## 标准库示例

- `std::io::Cursor::get_mut`

- `std::ptr::Unique::get_mut`
- `std::sync::PoisonError::get_mut`
- `std::sync::atomic::AtomicBool::get_mut`
- `std::collections::hash_map::OccupiedEntry::get_mut`
- `<[T]>::get_unchecked`

## 一个集合上的方法，如果返回迭代器，需遵循命名规则：`iter`, `iter_mut`, `into_iter` (C-ITER)

```
fn iter(&self) -> Iter           // Iter implements Iterator<Item = &U>
fn iter_mut(&mut self) -> IterMut // IterMut implements Iterator<Item = &mut U>
fn into_iter(self) -> IntoIter   // IntoIter implements Iterator<Item = U>
```

上面的规则适用于同构性的数据集合。与之相反，`str` 类型是一个 UTF-8 字节数组切片，与同构性集合有一点微妙的差别，它可以认为是字节集合，也可以认为是字符集合，因此它提供了 `str::bytes` 去遍历字节，还有 `str::chars` 去遍历字符，而并没有直接定义 `iter` 等方法。

上述规则只适用于方法，并不适用于函数。例如 `url` 包的 `percent_encode` 函数返回一个迭代器用于遍历百分比编码 (Percent encoding) 的字符串片段。在这种情况下，使用 `iter` / `iter_mut` / `into_iter` 诸如此类的函数命名无法表达任何具体的含义。

## 标准库示例

- `Vec::iter`
- `Vec::iter_mut`
- `Vec::into_iter`
- `BTreeMap::iter`
- `BTreeMap::iter_mut`

## 迭代器的类型应该与产生它的方法名相匹配(C-ITER-TY)

例如形如 `into_iter()` 的方法应该返回一个 `IntoIter` 类型，与之相似，其它任何返回迭代器的方法也应该遵循这种命名惯例。

上述规则主要应用于方法，但是经常对于函数也适用。例如上文提到的 `url` 包中的 `percent_encode` 函数，返回了一个 `PercentEncode` 类型。

特别是，当这些类型跟包名前缀一起使用时，将具备非常清晰的含义，例如 `vec::IntoIter`。

## 标准库示例

- `Vec::iter` returns `Iter`
- `Vec::iter_mut` returns `IterMut`
- `Vec::into_iter` returns `IntoIter`
- `BTreeMap::keys` returns `Keys`
- `BTreeMap::values` returns `Values`

## Cargo Feature 的名称不应该包含占位词(C-FEATURE)

不要在 `Cargo feature` 中包含无法传达任何意义的词，例如 `use-abc` 或 `with-abc`，直接命名为 `abc` 即可。

一个典型的例子就是：一个包对标准库有可选性的依赖。标准的写法如下：

```
# 在 Cargo.toml 中

[features]
default = ["std"]
std = []

// 在我们自定义的 lib.rs 中

#![cfg_attr(not(feature = "std"), no_std)]
```

除了 `std` 之外，不要使用任何 `ust-std` 或者 `with-std` 等自以为很有创造性的名称。

## 命名要使用一致性的词序(C-WORD-ORDER)

这是一些标准库中的错误类型：

- `JoinPathsError`
- `ParseBoolError`
- `ParseCharError`
- `ParseFloatError`

- `ParseIntError`
- `RecvTimeoutError`
- `StripPrefixError`

它们都使用了 `谓语-宾语-错误` 的词序，如果我们想要表达一个网络地址无法分析的错误，由于词序一致性的原则，命名应该如下 `ParseAddrError`，而不是 `AddrParseError`。

词序和个人习惯有很大关系，想要注意的是，你可以选择合适的词序，但是要在包的范畴内保持一致性，就如标准库中的包一样。

# 面试经验 doing

其实这一章节的处境有些尴尬，Rust 虽然在世界范围有点小名气，但是在国内目前还处于开荒阶段，因此至少就公开招聘而言，岗位并不多。

但是既然号称最佳实践，少了面试，总觉得会缺少些什么，由于现在还没有太多的经验可以参考，我们选择先从网上摘选些文章分享给大家，**但仅供参考，具体还要大家自己来辨别。**

---

感谢 [Kasper4649](#) 的章节提议和资源分享

---

- [记一次 Rust 技术面试](#)
- [飞书 Rust 实习](#)
- [字节跳动 Rust/C++ 实习](#)
- [记一次面试](#)
- [字节跳动面试经历](#)

To be continued..

## **最佳实践**

[https://www.reddit.com/r/rust/comments/rgjsbt/whats\\_your\\_top\\_rust\\_tip\\_crate\\_tool\\_other\\_for/](https://www.reddit.com/r/rust/comments/rgjsbt/whats_your_top_rust_tip_crate_tool_other_for/)

[https://www.reddit.com/r/rust/comments/rnmmqz/question\\_how\\_to\\_keep\\_code\\_dry\\_when\\_many\\_similar/](https://www.reddit.com/r/rust/comments/rnmmqz/question_how_to_keep_code_dry_when_many_similar/)

[https://www.reddit.com/r/rust/comments/rrgho1/what\\_is\\_the\\_recommended\\_way\\_to\\_use\\_a\\_library/](https://www.reddit.com/r/rust/comments/rrgho1/what_is_the_recommended_way_to_use_a_library/)

## **最佳开发流程workflow**

cargo watch

[https://www.reddit.com/r/rust/comments/rxrkbo/what\\_is\\_your\\_workflow\\_when\\_working\\_with\\_rust/](https://www.reddit.com/r/rust/comments/rxrkbo/what_is_your_workflow_when_working_with_rust/)

## **测试文件组织结构**

[https://www.reddit.com/r/rust/comments/rsuhnn/need\\_a\\_piece\\_of\\_advice\\_about\\_organising\\_tests/](https://www.reddit.com/r/rust/comments/rsuhnn/need_a_piece_of_advice_about_organising_tests/)

## **git备份**

<https://github.com/tkellogg/dura>

## **code cover**

<https://docs codecov com/docs>

## clippy

[https://www.reddit.com/r/rust/comments/s62xu0/inspect\\_enum\\_variant\\_size\\_differences/](https://www.reddit.com/r/rust/comments/s62xu0/inspect_enum_variant_size_differences/)

## todo

unimplemented!() todo!()

## 如何获知变量类型或者函数的返回类型

有几种常用的方式:

- 第一种是查询标准库或者三方库文档，搜索 `File`，然后找到它的 `open` 方法，但是此处更推荐第二种方法：
- 在[Rust IDE]章节，我们推荐了 `vsCode` IED和 `rust-analyze` 插件，如果你成功安装的话，那么就可以在 `vscode` 中很方便的通过代码跳转的方式查看代码，同时 `rust-analyze` 插件还会对代码中的类型进行标注，非常方便好用！
- 你还可以尝试故意标记一个错误的类型，然后让编译器告诉你：

```
let f: u32 = File::open("hello.txt");
```

错误提示如下：

```
error[E0308]: mismatched types
--> src/main.rs:4:18
|
4 |     let f: u32 = File::open("hello.txt");
|           ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
`std::result::Result`
|
= note: expected type `u32`
         found type `std::result::Result<std::fs::File, std::io::Error>`
```

## 代码风格(todo)

[https://www.reddit.com/r/rust/comments/rlsatb/generically\\_working\\_with\\_futuresinkstream/](https://www.reddit.com/r/rust/comments/rlsatb/generically_working_with_futuresinkstream/)

# 手把手带你实现链表

---

其它语言：兄弟，语言学了吗？来写一个链表证明你基本掌握了语法。

Rust 语言：兄弟，语言精通了吗？来写一个链表证明你已经精通了 Rust！

---

上面的对话非常真实，我们在之前的章节也讲过避免从入门到放弃，其中最重要的就是 - 不要写链表或者类似的数据结构！

而本章，你就将见识到何为真正的深坑，看完后，就知道没有提早跳进去是一个多么幸运的事。总之，在专题中，你将学会如何使用 Rust 来实现链表。

专题内容翻译自英文开源书 [Learning Rust With Entirely Too Many Linked Lists](#)，但是在内容上做了一些调整(原书虽然非常棒，但是在一些内容组织和文字细节上我觉得还是可以优化下的 : D)，希望大家喜欢。

# 我们到底需不需要链表

经常有读者询问该如何实现一个链表，怎么说呢，这个答案主要取决于你的需求，因此并不是很好回答。鉴于此，我决定通过这本书来详尽的介绍该如何实现一个链表，大家应该都能从这本书中找到答案。

书中我们将通过实现 6 种链表来学习基本和进阶 Rust 编程知识，在此过程中，你能学到：

- 指针类型: `&`, `&mut`, `Box`, `Rc`, `Arc`, `*const`, `*mut`, `NonNull`
- 所有权、借用、继承可变性、内部可变性、`Copy`
- 所有的关键字: `struct`、`enum`、`fn`、`pub`、`impl`、`use`, ...
- 模式匹配、泛型、解构
- 测试、安装新的工具链、使用 `miri`
- `Unsafe`: 裸指针、别名、栈借用、`UnsafeCell`、变体 variance

是的，链表就是这么可怕，只有将这些知识融会贯通后，你才能掌握 :(

---

事实上这本书中关于 Rust 语言的绝大部分知识都在 [Rust语言圣经](#) 中有讲，因此除非特殊情况，我们将直接提供链接供大家学习，重点还是放在链表实现上

---

## 创建一个项目

在开始前，先来创建一个项目专门用于链表学习：

```
$ cargo new --lib lists  
$ cd lists
```

之后，我们会将每个一个链表放入单独的文件中，需要注意的是我们会尽量模拟真实的 Rust 开发场景：你写了一段代码，然后编译器开始跳出试图教你做事，只有这样才能真正学会 Rust，温室环境是无法培养出强大的 Rustacean 的。

## 义务告知

首先，本书不是保姆式教程，而且我个人认为编程应该是快乐，这种快乐往往需要你自己发现而不是别人的事无巨细的讲解。

其次，我讨厌链表。链表真的是一种糟糕的数据结构，尽管它在部分场景下确实很有用：

- 对列表进行大量的分割和合并操作

- 无锁并发
- 要实现内核或嵌入式的服务
- 你在使用一个纯函数式语言，由于受限的语法和缺少可变性，因此你需要使用链表来解决这些问题

但是实事求是的说，这些场景对于几乎任何 Rust 开发都是很少遇到的，99% 的场景你可以使用 `Vec` 来替代，然后 1% 中的 99% 可以使用 `VecDeque`。由于它们具有更少的内存分配次数、更低的内存占用、随机访问和缓存亲和特性，因此能够适用于绝大多数工作场景。总之，类似于 `trie` 树，链表也是一种非常小众的数据结构，特别是对于 Rust 开发而言。

---

本书只是为了学习链表该如何实现，如果大家只是为了使用链表，强烈推荐直接使用标准库或者社区提供的现成实现，例如 `std::collections::LinkedList`

---

### 链表有 $O(1)$ 的分割、合并、插入、移除性能

是的，但是你首先要考虑的是，这些代码被调用的频率是怎么样的？是否在热点路径？答案如果是否定的，那么还是强烈建议使用 `Vec` 等传统数据结构，况且整个数组的拷贝也是相当快的！

况且，`Vec` 上的 `push` 和 `pop` 操作是  $O(1)$  的，它们比链表提供的 `push` 和 `pop` 要更快！我们只需要通过一个指针 + 内存偏移就可以访问了。

---

关于是否使用链表这个问题，Bjarne Stroustrup 有过非常深入的[讲解](#)

---

但是如果你的整体项目确实因为某一段分割、合并的代码导致了性能低下，那么就放心大胆的使用链表吧。

### 我无法接受内存重新分配的代价

是的，`Vec` 当 `capacity` 不够时，会重新分配一块内存，然后将之前的 `Vec` 全部拷贝过去，但是对于绝大多数使用场景，要么 `Vec` 不在热点路径中，要么 `Vec` 的容量可以提前预测。

对于前者，那性能如何自然无关紧要。而对于后者，我们只需要使用 `Vec::with_capacity` 提前分配足够的空间即可，同时，Rust 中所有的迭代器还提供了 `size_hint` 也可以解决这种问题。

当然，如果这段代码在热点路径，且你无法提前预测所需的容量，那么链表确实会更提升性能。

## 链表更节省内存空间

首先，这个问题较为复杂。一个标准的数组调整策略是：增加或减少数组的长度使数组最多有一半为空，例如 capacity 增长是翻倍的策略。这确实会导致内存空间的浪费，特别是在 Rust 中，我们不会自动收缩集合类型。

但是上面说的是最坏的情况，如果是最好的情况，那整个数组其实只有 3 个指针大小(指针在 Rust 中占用一个 word 的空间，例如 64 位机器就是 8 个字节的大小)的内存浪费，或者说，没有浪费。

而且链表实际上也有内存浪费，例如链表中的每个元素都会占用额外的内存：单向链表浪费一个指针，双向链表浪费两个指针。当然，如果你的链表中每个元素都很大，那相对来说，这种浪费也微不足道，但是如果链表的元素较小且数量很多呢？那浪费的空间就相当可观了！

当然，这个也和使用的内存分配器有关( allocator )：对链表节点的分配和回收会经常发生，这样就不会浪费内存。

总之，如果链表的元素较大，你也无法预测数组的空间，同时还有一个不错的内存分配器，那链表确实可以节省空间！

## 我在函数语言中一直使用链表

对于函数语言而言，链表确实非常棒，因为你可以解决可变性问题，还能递归地去使用，当然，可能还有一定的图方便的因素，因为链表不用操心长度等问题。

但彼之蜜糖不等于吾之蜜糖，函数语言的一些使用习惯不应该带入到其它语言中，例如 Rust。

- 函数语言往往将链表用于迭代，但是 Rust 中最适合迭代的数据结构是迭代器 Iterator
- 函数式语言的不可变对于 Rust 也不是问题
- Rust 还支持对数组进行切片以获取其中一部分连续的元素，而在函数语言中你可能得通过链表的 head/tail 分割来完成

其实，在函数语言中，我们也应该选择合适的数据结构来解决适合的场景，而不是一根链表挂腰间，潇潇洒洒走天下。

## 链表适合构建并发数据结构

是这样的，如果有这样的需求，那么链表会非常合适！但是只有在你确实需要并发数据结构，且没有其它办法时，再考虑链表！

## **链表非常适合教学目的**

额... 这么说也没错，毕竟所有的编程语言课程都以链表来作为最常见的练手项目，包括本书也是服务于这个目的的。

# 糟糕的单向链表栈

本章，让我们用一个不咋样的单向链表来实现一个栈数据结构，因为不咋样，实现起来倒是很简单。

首先，创建一个文件 `src/first.rs` 用于存放本章节的链表代码，虽然糟糕，也不能用完就扔，大家说是不 :P 然后在 `lib.rs` 中添加这一行代码：

```
// in lib.rs
pub mod first;
```

# 基本数据布局( Layout )

发现一件尴尬的事情，之前介绍了这么多，但是竟然没有介绍链表是什么...亡羊补牢未为晚也，链表就是一系列存储在堆上的连续数据，大家是不是发现这个定义跟动态数据 `Vector` 非常相似，那么区别在于什么呢？

区别就在于链表中的每一个元素都指向下一个元素，最终形成 - 顾名思义的链表：`A1 -> A2 -> A3 -> Null`。而数组中的元素只是连续排列，并不存在前一个元素指向后一个元素的情况，而是每个元素通过下标索引来访问。

既然函数式语言的程序员最常使用链表，那么我们来看看他们给出的定义长什么样：

```
List a = Empty | Elem a (List a)
```

mu...看上去非常像一个数学定义，我们可以这样阅读它，列表 `a` 要么是空，要么是一个元素后面再跟着一个列表。非常递归也不咋好懂的定义，果然，这很函数式语言。

下面我们再来使用 Rust 的方式对链表进行下定义，为了简单性，这先不使用泛型：

```
// in first.rs

pub enum List {
    Empty,
    Elem(i32, List),
}
```

喔，看上去人模狗样，来，运行下看看：

```
$ cargo run
error[E0072]: recursive type `List` has infinite size
--> src/first.rs:1:1
1 | pub enum List {
| ^^^^^^^^^^ recursive type has infinite size
2 |     Empty,
3 |     Elem(i32, List),
|         ----- recursive without indirection
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
|
3 |     Elem(i32, Box<List>),
```

帅不过 3 秒的玩意儿～～ 好在这个问题，我们在[之前的章节](#)中就讲过了，简而言之，当前的类型是[不定长的](#)，对于 Rust 编译器而言，所有栈上的类型都必须在编译期有固定的长度，一个简单的解决方案就是

使用 `Box` 将值封装到堆上，然后使用栈上的定长指针来指向堆上不定长的值。

实际上，如果大家有仔细看编译错误的话，它还给出了我们提示：`Elem(i32, Box<List>)`，和我们之前的结论一致，下面来试试：

```
pub enum List {
    Empty,
    Elem(i32, Box<List>),
}

$ cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.22s
```

万能的编译器再一次拯救了我们，这次的代码成功完成了编译。但是这依然是不咋滴的 `List` 定义，有几个原因。

首先，考虑一个拥有两个元素的 `List`：

```
[] = Stack
() = Heap

[Elem A, ptr] -> (Elem B, ptr) -> (Empty, *junk*)
```

这里有两个问题：

- 最后一个节点分配在了堆上，但是它看上去根本不像一个 `Node`
- 第一个 `Node` 是存储在栈上的，结果一家子不能整整齐齐的待在堆上了

这两点看上去好像有点矛盾：你希望所有节点在堆上，但是又觉得最后一个节点不应该在堆上。那再来考虑另一种布局(`Layout`)方式：

```
[ptr] -> (Elem A, ptr) -> (Elem B, *null*)
```

在这种布局下，我们无条件的在堆上创建所有节点，最大的区别就是这里不再有 `junk`。那么什么是 `junk`？为了理解这个概念，先来看看枚举类型的内存布局(`Layout`)长什么样：

```
enum Foo {
    D1(u8),
    D2(u16),
    D3(u32),
    D4(u64)
}
```

大家觉得 `Foo::D1(99)` 占用多少内存空间？是 `u8` 对应的 1 个字节吗？答案是 8 个字节（为了好理解，这里不考虑 enum tag 所占用的额外空间），因为枚举有一个特点，枚举成员占用的内存空间大小跟最大的成员对齐，在这个例子中，所有的成员都会跟 `u64` 进行对齐。

在理解了这点后，再回到我们的 `List` 定义。这里最大的问题就是尽管 `List::Empty` 是空的节点，但是它依然得消耗和其它节点一样的内存空间。

与其让一个节点不进行内存分配，不如让它一直进行内存分配，无论是否有内容，因为后者会保证节点内存布局的一致性。这种一致性对于 `push` 和 `pop` 节点来说可能没有什么影响，但是对于链表的分割和合并而言，就有意义了。

下面，我们对之前两种不同布局的 `List` 进行下分割：

`layout 1:`

`[Elem A, ptr] -> (Elem B, ptr) -> (Elem C, ptr) -> (Empty *junk*)`

`split off C:`

`[Elem A, ptr] -> (Elem B, ptr) -> (Empty *junk*)  
[Elem C, ptr] -> (Empty *junk*)`

`layout 2:`

`[ptr] -> (Elem A, ptr) -> (Elem B, ptr) -> (Elem C, *null*)`

`split off C:`

`[ptr] -> (Elem A, ptr) -> (Elem B, *null*)  
[ptr] -> (Elem C, *null*)`

可以看出，在布局 1 中，需要将 `C` 节点从堆上拷贝到栈中，而布局 2 则无需此过程。而且从分割后的布局清晰度而言，2 也要优于 1。

现在，我们应该都相信布局 1 更糟糕了，而且不幸的是，我们之前的实现就是布局 1，那么该如何实现新的布局呢？也许，我们可以实现类似如下的 `List`：

```
pub enum List {
    Empty,
    ElemThenEmpty(i32),
    ElemThenNotEmpty(i32, Box<List>),
}
```

但是，你们有没有觉得更糟糕了...有些不忍直视的感觉。这让我们的代码复杂度大幅提升，例如你现在得实现一个完全不合法的状态：`ElemThenNotEmpty(0, Box(Empty))`，而且这种实现依然有之前的不一致性的  
问题。

之前我们提到过枚举成员的内存空间占用和 enum tag 问题，实际上我们可以创建一个特例：

```
enum Foo {
    A,
    B(ContainsANonNullPtr),
}
```

在这里 null 指针的优化就开始介入了，它会消除枚举成员 A 占用的额外空间，原因在于编译器可以直接将 A 优化成 0，而 B 则不行，因为它包含了非 null 指针。这样一来，编译器就无需给 A 打 tag 进行识别了，而是直接通过 0 就能识别出这是 A 成员，非 0 的自然就是 B 成员。

事实上，编译器还会对枚举做一些其他优化，但是 null 指针优化是其中最重要的一条。

所以我们应该怎么避免多余的 junk，保持内存分配的一致性，还能保持 null 指针优化呢？枚举可以让我们声明一个类型用于表达多个不同的值，而结构体可以声明一个类型同时包含多个值，只要将这两个类型结合在一起，就能实现之前的目标：枚举类型用于表示 List，结构体类型用于表示 Node。

```
struct Node {
    elem: i32,
    next: List,
}

pub enum List {
    Empty,
    More(Box<Node>),
}
```

让我们看看新的定义是否符合之前的目标：

- List 的尾部不会再分配多余的 junk 值，通过！
- List 枚举的形式可以享受 null 指针优化，完美！
- 所有的元素都拥有统一的内存分配，Good!

很好，我们准确构建了之前想要的内存布局，并且证明了最初的内存布局问题多多，编译下试试：

```
error[E0446]: private type `Node` in public interface
--> src/first.rs:8:10
|
1 | struct Node {
| ----- `Node` declared as private
...
8 |     More(Box<Node>),
|           ^^^^^^ can't leak private type
```

在英文书中，这里是一个 warning，但是在笔者使用的最新版中(Rust 1.59)，该 warning 已经变成一个错误。主要原因在于 pub enum 会要求它的所有成员必须是 pub，但是由于 Node 没有声明为 pub，因

此产生了冲突。

这里最简单的解决方法就是将 `Node` 结构体和它的所有字段都标记为 `pub`：

```
pub struct Node {
    pub elem: i32,
    pub next: List,
}
```

但是从编程的角度而言，我们还是希望让实现细节只保留在内部，而不是对外公开，因此以下代码相对会更加适合：

```
pub struct List {
    head: Link,
}

enum Link {
    Empty,
    More(Box<Node>),
}

struct Node {
    elem: i32,
    next: Link,
}
```

从代码层面看，貌似多了一层封装，但是实际上 `List` 只有一个字段，因此结构体的大小跟字段大小是相等的，没错，传说中的零开销抽象！

至此，一个令人满意的数据布局就已经设计完成，下面一起来看看该如何使用这些数据。

# 定义基本操作

这个章节我们一起来为新创建的 `List` 定义一些基本操作，首先从创建链表开始。

## New

为了将实际的代码跟类型关联在一起，我们需要使用 `impl` 语句块：

```
impl List {  
    // TODO  
}
```

下一步就是创建一个关联函数，用于构建 `List` 的新实例，该函数的作用类似于其他语言的构造函数。

```
impl List {  
    pub fn new() -> Self {  
        List { head: Link::Empty }  
    }  
}
```

---

学习链接: [impl](#)、[关联函数](#)、[Self](#)

---

## Push

在开始实现之前，你需要先了解 `self`、`&self`、`&mut self` 这几个概念。

在创建链表后，下一步就是往链表中插入新的元素，由于 `push` 会改变链表，因此我们使用 `&mut self` 的方法签名：

```
impl List {  
    pub fn push(&mut self, elem: i32) {  
        // TODO  
    }  
}
```

根据之前的数据定义，首先需要创建一个 `Node` 来存放该元素：

```
pub fn push(&mut self, elem: i32) {
    let new_node = Node {
        elem: elem,
        next: ??????
    };
}
```

下一步需要让该节点指向之前的旧 List：

```
pub fn push(&mut self, elem: i32) {
    let new_node = Node {
        elem: elem,
        next: self.head,
    };
}

error[E0507]: cannot move out of `self.head` which is behind a mutable reference
--> src/first.rs:23:19
|
23 |         next: self.head,
|             ^^^^^^^^^^ move occurs because `self.head` has type `Link`,
|             which does not implement the `Copy` trait
```

但是，如上所示，这段代码会报错，因为试图将借用的值 self 中的 head 字段的所有权转移给 next，在 Rust 中这是不被允许的。那如果我们试图将值再放回去呢？

```
pub fn push(&mut self, elem: i32) {
    let new_node = Box::new(Node {
        elem: elem,
        next: self.head,
    });

    self.head = Link::More(new_node);
}
```

其实在写之前，应该就预料到结果了，显然这也是不行的，虽然从我们的角度来看还挺正常的，但是 Rust 并不会接受(有多种原因，其中主要的是[Exception safety](#))。

我们需要一个办法，让 Rust 不再阻挠我们，其中一个可行的办法是使用 clone：

```
pub struct List {
    head: Link,
}

#[derive(Clone)]
enum Link {
    Empty,
    More(Box<Node>),
}

#[derive(Clone)]
struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: Link::Empty }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Node {
            elem: elem,
            next: self.head.clone(),
        };
        self.push();
    }
}
```

clone 用起来简单，且可解万愁，但是。。。既然是链表，性能那自然是很重要的，特别是要封装成库给其他代码使用时，那性能更是重中之重。

没办法了，我们只能向大名鼎鼎的 Rust 黑客 Indiana Jones 求助了：



经过一番诚心祈愿，Indy 建议我们使用 `mem::replace` 秘技。这个非常有用的函数允许我们从一个借用中偷出一个值的同时再放入一个新值。

```
pub fn push(&mut self, elem: i32) {
    let new_node = Box::new(Node {
        elem: elem,
        next: std::mem::replace(&mut self.head, Link::Empty),
    });

    self.head = Link::More(new_node);
}
```

这里，我们从借用 `self` 中偷出了它的值 `head` 并赋予给 `next` 字段，同时将一个新值 `Link::Empty` 放入到 `head` 中，成功完成偷梁换柱。不得不说，这个做法非常刺激，但是很不幸的是，目前为止，最好的办法可能也只能是它了。

但是不管怎样，我们成功的完成了 `push` 方法，下面再来看看 `pop`。

## Pop

`push` 是插入元素，那 `pop` 自然就是推出一个元素，因此也需要使用 `&mut self`，除此之外，推出的元素需要被返回，这样调用者就可以获取该元素：

```
pub fn pop(&mut self) -> Option<i32> {
    // TODO
}
```

我们还需要一个办法来根据 `Link` 是否有值进行不同的处理，这个可以使用 `match` 来进行模式匹配：

```
pub fn pop(&mut self) -> Option<i32> {
    match self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
}
```

目前的代码显然会报错，因为函数的返回值是 `Option<T>` 枚举，而目前的返回值是 `()`。当然，我们可以返回一个 `Option<T>` 的枚举成员 `None`，但是一个更好的做法是使用 `unimplemented!()`，该宏可以

明确地说明目前的代码还没有实现，一旦代码执行到 `unimplemented!()` 的位置，就会发生一个 `panic`。

```
pub fn pop(&mut self) -> Option<i32> {
    match self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
    unimplemented!()
}
```

`panics` 是一种发散函数，该函数永不返回任何值，因此可以用于需要返回任何类型的地方。这句话很不好理解，但是从上面的代码中可以看出 `unimplemented!()` 是永不返回的函数，但是它却可以用于一个返回 `Option<i32>` 的函数中来替代返回值。

以上代码果不其然又报错了：

```
$ cargo build

error[E0507]: cannot move out of borrowed content
--> src/first.rs:28:15
|
28 |     match self.head {
|     |     ^^^^^^^^^^
|     |
|     |     cannot move out of borrowed content
|     |     help: consider borrowing here: `&self.head`
...
32 |         Link::More(node) => {
|             ----- data moved here
|
note: move occurs because `node` has type `std::boxed::Box<first::Node>`, which does
not implement the `Copy` trait
```

好在编译器偷偷提示了我们使用借用来替代所有权转移：`&self.head`。修改后，如下：

```

pub fn pop(&mut self) -> Option<i32> {
    match &self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
    unimplemented!()
}

```

是时候填写相应的逻辑了：

```

pub fn pop(&mut self) -> Option<i32> {
    let result;
    match &self.head {
        Link::Empty => {
            result = None;
        }
        Link::More(node) => {
            result = Some(node.elem);
            self.head = node.next;
        }
    };
    result
}

```

当链表为 `Empty` 时，返回一个 `None`，表示我们没有 `pop` 到任何元素；若不为空，则返回第一个元素，并将 `head` 指向下一个节点 `node.next`。但是这段代码又报错了：

```

error[E0507]: cannot move out of `node.next` which is behind a shared reference
--> src/first.rs:37:29
|
37 |         self.head = node.next;
|             ^^^^^^^^^^ move occurs because `node.next` has type
|             `Link`, which does not implement the `Copy` trait

```

原因是试图转移 `node` 的所有权，但只有它的引用。回头仔细看看代码，会发现这里的关键是我们希望移除一些东西，这意味着需要通过值的方式获取链表的 `head`。看来只能故技重施了：

```
pub fn pop(&mut self) -> Option<i32> {
    let result;
    match std::mem::replace(&mut self.head, Link::Empty) {
        Link::Empty => {
            result = None;
        }
        Link::More(node) => {
            result = Some(node.elem);
            self.head = node.next;
        }
    };
    result
}
```

我们将 `self.head` 的值偷出来，然后再将 `Link::Empty` 填回到 `self.head` 中。此时用于 `match` 匹配的就是一个拥有所有权的值类型，而不是之前的引用类型。

事实上，上面的代码有些啰嗦，我们可以直接在 `match` 的两个分支中通过表达式进行返回：

```
pub fn pop(&mut self) -> Option<i32> {
    match std::mem::replace(&mut self.head, Link::Empty) {
        Link::Empty => None,
        Link::More(node) => {
            self.head = node.next;
            Some(node.elem)
        }
    }
}
```

这样修改后，代码就更加简洁，可读性也更好了，至此链表的基本操作已经完成，下面让我们写一个测试代码来测试下它的功能和正确性。

# 一些收尾工作以及最终代码

在之前的章节中，我们完成了 Bad 单链表栈的数据定义和基本操作，下面一起来写一些测试代码。

## 单元测试

关于如何编写测试，请参见[自动化测试章节](#)

首先，单元测试代码要放在待测试的目标代码旁边，也就是同一个文件中：

```
// in first.rs
#[cfg(test)]
mod test {
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), None);
    }
}
```

在 `src/first.rs` 中添加以上测试模块，然后使用 `cargo test` 运行相关的测试用例：

```
$ cargo test

error[E0433]: failed to resolve: use of undeclared type or module `List`
 --> src/first.rs:43:24
 |
43 |     let mut list = List::new();
|           ^^^^ use of undeclared type or module `List`
```

Ooops! 报错了，从错误内容来看，是因为我们在一个不同的模块 `test` 中，引入了 `first` 模块中的代码，由于前者是后者的子模块，因此可以使用以下方式引入 `first` 模块中的 `List` 定义：

```
#[cfg(test)]
mod test {
    use super::List;
    // 其它代码保持不变
}
```

大家可以再次尝试使用 `cargo test` 运行测试用例，具体的结果就不再展开，关于结果的解读，请参看文章开头的链接。

## Drop

现在还有一个问题，我们是否需要手动来清理释放我们的链表？答案是 No，因为 Rust 为我们提供了 `Drop` 特征，若变量实现了该特征，则在它离开作用域时将自动调用解构函数以实现资源清理释放工作，最妙的是，这一切都发生在编译期，因此没有多余的性能开销。

---

关于 `Drop` 特征的详细介绍，请参见[智能指针 - Drop](#)

---

事实上，我们无需手动为自定义类型实现 `Drop` 特征，原因是 Rust 自动为几乎所有类型都实现了 `Drop`，例如我们自定义的结构体，只要结构体的所有字段都实现了 `Drop`，那结构体也会自动实现 `Drop`！

但是，有的时候这种自动实现可能不够优秀，例如考虑以下链表：

```
list -> A -> B -> C
```

当 List 被自动 drop 后，接着会去尝试 Drop A，然后是 B，最后是 C。这个时候，其中一部分读者可能会紧张起来，因此这其实是一段递归代码，可能会直接撑爆我们的 stack 栈。

例如以下的测试代码会试图创建一个很长的链表，然后会导致栈溢出错误：

```
#[test]
fn long_list() {
    let mut list = List::new();
    for i in 0..100000 {
        list.push(i);
    }
    drop(list);
}

thread 'first::test::long_list' has overflowed its stack
```

可能另一部分同学会想 "这显然是尾递归，一个靠谱的编程语言是不会让尾递归撑爆我们的 stack"。然后，这个想法并不正确，下面让我们尝试模拟编译器来看看 Drop 会如何实现：

```

impl Drop for List {
    fn drop(&mut self) {
        // NOTE: 在 Rust 代码中, 我们不能显式的调用 `drop` 方法, 只能调用 std::mem::drop 函数
        // 这里只是在模拟编译器!
        self.head.drop(); // 尾递归 - good!
    }
}

impl Drop for Link {
    fn drop(&mut self) {
        match *self {
            Link::Empty => {} // Done!
            Link::More(ref mut boxed_node) => {
                boxed_node.drop(); // 尾递归 - good!
            }
        }
    }
}

impl Drop for Box<Node> {
    fn drop(&mut self) {
        self.ptr.drop(); // 糟糕, 这里不是尾递归!
        deallocate(self.ptr); // 不是尾递归的原因是在 `drop` 后, 还有额外的操作
    }
}

impl Drop for Node {
    fn drop(&mut self) {
        self.next.drop();
    }
}

```

从上面的代码和注释可以看出为 `Box<Node>` 实现的 `drop` 方法中, 在 `self.ptr.drop` 后调用的 `deallocate` 会导致非尾递归的情况发生。

因此我们需要手动为 `List` 实现 `Drop` 特征:

```

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empty);
        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Empty);
            // boxed_node 在这里超出作用域并被 drop,
            // 由于它的 `next` 字段拥有的 `Node` 被设置为 Link::Empty,
            // 因此这里并不会有无边界的递归发生
        }
    }
}

```

测试下上面的实现以及之前的长链表例子：

```
$ cargo test  
Running target/debug/lists-5c71138492ad4b4a  
  
running 2 tests  
test first::test::basics ... ok  
test first::test::long_list ... ok  
  
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

完美！



### 为什么要提前优化？

事实上，我们在这里做了提前优化，否则可以使用 `while let Some(_) = self.pop() { }`，这种实现显然更加简单。那么问题来了：它们的区别是什么，有哪些性能上的好处？特别是在链表不仅仅支持 `i32` 时。

▼ 点击这里展开答案

`self.pop()` 的会返回 `Option<i32>`，而我们之前的实现仅仅对智能指针 `Box<Node>` 进行操作。前者会对值进行拷贝，而后者仅仅使用的是指针类型。

当链表中包含的值是其他较大的类型时，那这个拷贝的开销将变得非常高昂。

## 最终代码

```
use std::mem;

pub struct List {
    head: Link,
}

enum Link {
    Empty,
    More(Box<Node>),
}

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: Link::Empty }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: mem::replace(&mut self.head, Link::Empty),
        });

        self.head = Link::More(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match mem::replace(&mut self.head, Link::Empty) {
            Link::Empty => None,
            Link::More(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empty);

        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Empty);
        }
    }
}
```

```

}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));
    }
}

```

从代码行数也可以看出，我们实现的肯定不是一个精致的链表：总共只有 80 行代码，其中一半还是测试！

但是万事开头难，既然开了一个好头，那接下来我们一鼓作气，继续看看更精致的链表长什么样。

# 还可以的单向链表

在之前我们写了一个最小可用的单向链表，下面一起来完善下，首先创建一个新的文件 `src/second.rs`，然后在 `lib.rs` 中引入：

```
// in lib.rs

pub mod first;
pub mod second;
```

并将 `first.rs` 中的所有内容拷贝到 `second.rs` 中。

# 优化类型定义

首先，我们需要优化下类型的定义，可能一部分同学已经觉得之前的类型定义相当不错了，但是如果大家仔细观察下 `Link`：

```
enum Link {  
    Empty,  
    More(Box<Node>),  
}
```

会发现，它其实跟 `Option<Box<Node>>` 非常类似。

## Option

但是为了代码可读性，我们不能直接使用这个冗长的类型，否则代码中将充斥着 `Option<Box<Node>>` 这种令人难堪的类型，为此可以使用类型别名。首先，将之前的代码使用新的 `Link` 进行修改：

```

use std::mem;

pub struct List {
    head: Link,
}

// 类型别名, type alias
type Link = Option<Box<Node>>;

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: mem::replace(&mut self.head, None),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match mem::replace(&mut self.head, None) {
            None => None,
            Some(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, None);
        while let Some(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, None);
        }
    }
}

```

代码看上去稍微好了一些，但是 Option 的好处远不止这些。

首先，之前咱们用到了 `mem::replace` 这个让人胆战心惊但是又非常有用的函数，而 `option` 直接提供了一个方法 `take` 用于替代它：

```
pub struct List {
    head: Link,
}

type Link = Option<Box<Node>>;

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match self.head.take() {
            None => None,
            Some(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

其次，`match option { None => None, Some(x) => Some(y) }` 这段代码可以直接使用 `map` 方法代替，`map` 会对 `Some(x)` 中的值进行映射，最终返回一个新的 `Some(y)` 值。

---

我们往往将闭包作为参数传递给 map 方法，关于闭包可以参见[此章](#)

---

```
pub fn pop(&mut self) -> Option<i32> {
    self.head.take().map(|node| {
        self.head = node.next;
        node.elem
    })
}
```

不错，看上去简洁了很多，下面运行下测试代码确保链表依然可以正常运行(这就是 TDD 的优点！)：

```
$ cargo test

Running target/debug/lists-5c71138492ad4b4a

running 2 tests
test first::test::basics ... ok
test second::test::basics ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

很棒，接下来让我们来解决目前链表最大的问题：只支持 `i32` 类型的元素值。

## 泛型

为了让链表支持任何类型的元素，泛型就是绕不过去的坎，首先将所有的类型定义修改为泛型实现：

```

pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}

```

大家在修改了 List 的定义后，别忘了将 impl 中的 List 修改为 List<T>，切记泛型参数也是类型定义的一部分。

```
$ cargo test

    Running target/debug/lists-5c71138492ad4b4a

running 2 tests
test first::test::basics ... ok
test second::test::basics ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

如上所示，截至目前，测试用例依然运行良好，尽管我们把代码修改成了更加复杂的泛型。这里有一个点特别值得注意，我们并没有修改关联函数 `new`：

```
pub fn new() -> Self {
    List { head: None }
}
```

原因是 `Self` 承载了我们所有的荣耀，`List` 时，`Self` 就代表 `List`，当变成 `List<T>` 时，`Self` 也随之变化，代表 `List<T>`，可以看出使用它可以让未来的代码重构变得更加简单。

# Peek 函数

在之前章节中，我们定义了 `push`、`pop` 等基础操作，下面一起添加几个进阶操作，让我们的链表有用起来。

首先实现的就是 `peek` 函数，它会返回链表的表头元素的引用：

```
pub fn peek(&self) -> Option<&T> {
    self.head.map(|node| {
        &node.elem
    })
}

$ cargo build

error[E0515]: cannot return reference to local data `node.elem`
 --> src/second.rs:37:13
 |
37 |         &node.elem
 |         ^^^^^^^^^^ returns a reference to data owned by the current function

error[E0507]: cannot move out of borrowed content
 --> src/second.rs:36:9
 |
36 |     self.head.map(|node| {
 |     ^^^^^^^^^^ cannot move out of borrowed content
```

哎，Rust 大爷，您又哪里不满意了。不过问题倒是也很明显：`map` 方法是通过 `self` 获取的值，我们相当于把内部值的引用返回给函数外面的调用者。

一个比较好的解决办法就是让 `map` 作用在引用上，而不是直接作用在 `self.head` 上，为此我们可以使用 `Option` 的 `as_ref` 方法：

```
impl<T> Option<T> {
    pub fn as_ref(&self) -> Option<&T>;
}
```

该方法将一个 `Option<T>` 变成了 `Option<&T>`，然后再调用 `map` 就会对引用进行处理了：

```
pub fn peek(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        &node.elem
    })
}
```

```
$ cargo build  
Finished dev [unoptimized + debuginfo] target(s) in 0.32s
```

当然，我们还可以通过类似的方式获取一个可变引用:

```
pub fn peek_mut(&mut self) -> Option<&mut T> {  
    self.head.as_mut().map(|node| {  
        &mut node.elem  
    })  
}
```

至此 `peek` 已经完成，为了测试它的功能，我们还需要编写一个测试用例:

```
#[test]  
fn peek() {  
    let mut list = List::new();  
    assert_eq!(list.peek(), None);  
    assert_eq!(list.peek_mut(), None);  
    list.push(1); list.push(2); list.push(3);  
  
    assert_eq!(list.peek(), Some(&3));  
    assert_eq!(list.peek_mut(), Some(&mut 3));  
    list.peek_mut().map(|&mut value| {  
        value = 42  
    });  
  
    assert_eq!(list.peek(), Some(&42));  
    assert_eq!(list.pop(), Some(42));  
}  
  
$ cargo test  
  
error[E0384]: cannot assign twice to immutable variable `value`  
--> src/second.rs:100:13  
|  
99 |     list.peek_mut().map(|&mut value| {  
|         -----  
|         |  
|         first assignment to `value`  
|         help: make this binding mutable: `mut value`  
100|         value = 42  
|         ^^^^^^^^^^ cannot assign twice to immutable variable           ^~~~~~
```

天呐，错误源源不断，这次编译器抱怨说 `value` 是不可变的，但是我们明明确使用 `&mut value`，发生了什么？难道说在闭包中通过这种方式声明的可变性实际上没有效果？

实际上 `&mut value` 是一个模式匹配，它用 `&mut value` 模式去匹配一个可变的引用，此时匹配出来的 `value` 显然是一个值，而不是可变引用，因为只有完整的形式才是可变引用！

因此我们没必要画蛇添足，这里直接使用 `|value|` 来匹配可变引用即可，那么此时匹配出来的 `value` 就是一个可变引用。

```
#[test]
fn peek() {
    let mut list = List::new();
    assert_eq!(list.peek(), None);
    assert_eq!(list.peek_mut(), None);
    list.push(1); list.push(2); list.push(3);

    assert_eq!(list.peek(), Some(&3));
    assert_eq!(list.peek_mut(), Some(&mut 3));

    list.peek_mut().map(|value| {
        *value = 42
    });

    assert_eq!(list.peek(), Some(&42));
    assert_eq!(list.pop(), Some(42));
}
```

这次我们直接匹配出来可变引用 `value`，然后对其修改即可。

```
$ cargo test

Running target/debug/lists-5c71138492ad4b4a

running 3 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::peek ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

## 迭代器

集合类型可以通过 `Iterator` 特征进行迭代，该特征看起来比 `Drop` 要复杂点：

```
pub trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

这里的 `Item` 是[关联类型](#)，用来指代迭代器中具体的元素类型，`next` 方法返回的也是该类型。

其实上面的说法有点不够准确，原因是 `next` 方法返回的是 `Option<Self::Item>`，使用 `Option<T>` 枚举的原因是为了方便用户，不然用户需要 `has_next` 和 `get_next` 才能满足使用需求。有值时返回 `Some(T)`，无值时返回 `None`，这种 API 设计工程性更好，也更加安全，完美！

有点悲剧的是，Rust 截至目前还没有 `yield` 语句，因此我们需要自己来实现相关的逻辑。还有点需要注意，每个集合类型应该实现 3 种迭代器类型：

- `IntoIter` - `T`
- `IterMut` - `&mut T`
- `Iter` - `&T`

也许大家不认识它们，但是其实很好理解，`IntoIter` 类型迭代器的 `next` 方法会拿走被迭代值的所有权，`IterMut` 是可变借用，`Iter` 是不可变借用。事实上，类似的[命名规则](#)在 Rust 中随处可见，当熟悉后，以后见到类似的命名大家就可以迅速的理解其对值的运用方式。

## IntoIter

先来看看 `IntoIter` 该怎么实现：

```

pub struct IntoIter<T>(List<T>);

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // access fields of a tuple struct numerically
        self.0.pop()
    }
}

```

这里我们通过[元组结构体](#)的方式定义了 `IntoIter`，下面来测试下：

```

#[test]
fn into_iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), None);
}

$ cargo test

    Running target/debug/lists-5c71138492ad4b4a

running 4 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured

```

## Iter

相对来说，`IntoIter` 是最好实现的，因为它只是简单的拿走值，不涉及到引用，也不涉及到生命周期，而 `Iter` 就有所不同了。

这里的基本逻辑是我们持有一个当前节点的指针，当生成一个值后，该指针将指向下一个节点。

```
pub struct Iter<T> {
    next: Option<&Node<T>>,
}

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter { next: self.head.map(|node| &node) }
    }
}

impl<T> Iterator for Iter<T> {
    type Item = &T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &node);
            &node.elem
        })
    }
}

$ cargo build

error[E0106]: missing lifetime specifier
 --> src/second.rs:72:18
 |
72 |     next: Option<&Node<T>>,
 |           ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
 --> src/second.rs:82:17
 |
82 |     type Item = &T;
 |           ^ expected lifetime parameter
```

许久不见的错误又冒了出来，而且这次直指 Rust 中最难的点之一：生命周期。关于生命周期的讲解，这里就不再展开，如果大家还不熟悉，强烈建议看看[此章节](#)，然后再继续。

首先，先加一个生命周期试试：

```
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}
```

```

$ cargo build

error[E0106]: missing lifetime specifier
 --> src/second.rs:83:22
|
83 |     impl<T> Iterator for Iter<T> {
|             ^^^^^^^^ expected lifetime parameter

error[E0106]: missing lifetime specifier
 --> src/second.rs:84:17
|
84 |         type Item = &T;
|                 ^ expected lifetime parameter

error: aborting due to 2 previous errors

```

好的，现在有了更多的提示，来按照提示修改下代码：

```

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<'a, T> List<T> {
    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &'a node) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&'a mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &'a node);
            &'a node.elem
        })
    }
}

```

```

$ cargo build

error: expected `:`, found `node`
 --> src/second.rs:77:47
77 |         Iter { next: self.head.map(|node| &'a node) }
|         ---- while parsing this struct           ^^^^^^ expected `:`

error: expected `:`, found `node`
 --> src/second.rs:85:50
85 |         self.next = node.next.map(|node| &'a node);
|         ^^^^^^ expected `:`

error[E0063]: missing field `next` in initializer of `second::Iter<'_, _>`
 --> src/second.rs:77:9
77 |         Iter { next: self.head.map(|node| &'a node) }
|         ^^^^^ missing `next`
```

怎么回事。。感觉错误犹如雨后春笋般冒了出来，Rust 是不是被我们搞坏了 :(

现在看来，我们的生命周期是用错了，聪明的同学可能已经看出了端倪，那么再修改下试试；

```

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

// 这里无需生命周期，因为 List 没有使用生命周期的关联项
impl<T> List<T> {
    // 这里我们为 `iter` 声明一个生命周期 `a`，此时 `&self` 需要至少和 `Iter` 活得一样久
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &node) }
    }
}

// 这里声明生命周期是因为下面的关联类型 Item 需要
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    // 这里无需更改，因为上面已经处理了。
    // Self 依然是这么棒
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &node);
            &node.elem
        })
    }
}
```

现在，我们也许可以自信的编译下试试了：

```
$ cargo build

error[E0308]: mismatched types
--> src/second.rs:77:22
|
77 |         Iter { next: self.head.map(|node| &node) }
|                         ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct `second::Node`,
found struct `std::boxed::Box`
|
= note: expected type `std::option::Option<&second::Node<T>>`
        found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`

error[E0308]: mismatched types
--> src/second.rs:85:25
|
85 |         self.next = node.next.map(|node| &node);
|                         ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
|
= note: expected type `std::option::Option<&'a second::Node<T>>`
        found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`
```

(╯°□°)╯︵ ┻━┻

这么看，生命周期的问题解决了，但是又引入了新的错误。原因在于，我们希望存储 `&Node` 但是获取的却是 `&Box<Node>`。嗯，小问题，解引用搞定：

```
impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &*node) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &*node);
            &node.elem
        })
    }
}
```

```
$ cargo build

Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0515]: cannot return reference to local data `*node`
--> src/second.rs:77:43
|
77 |         Iter { next: self.head.map(|node| &*node) }
|                         ^^^^^^ returns a reference to data
owned by the current function

error[E0507]: cannot move out of borrowed content
--> src/second.rs:77:22
|
77 |         Iter { next: self.head.map(|node| &*node) }
|                         ^^^^^^^^^^ cannot move out of borrowed content

error[E0515]: cannot return reference to local data `*node`
--> src/second.rs:85:46
|
85 |         self.next = node.next.map(|node| &*node);
|                         ^^^^^^ returns a reference to data
owned by the current function

error[E0507]: cannot move out of borrowed content
--> src/second.rs:85:25
|
85 |         self.next = node.next.map(|node| &*node);
|                         ^^^^^^^^^^ cannot move out of borrowed content
```

又怎么了! (/得益) /—————

大家还记得之前章节的内容吗? 原因是这里我们忘记了 `as_ref`, 然后值的所有权被转移到了 `map` 中, 结果我们在内部引用了一个局部值, 造成一个悬垂引用:

```

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_ref().map(|node| &*node) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_ref().map(|node| &*node);
            &node.elem
        })
    }
}

```

§ cargo build

```

Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0308]: mismatched types
--> src/second.rs:77:22
|
77 |     Iter { next: self.head.as_ref().map(|node| &*node) }
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
|
= note: expected type `std::option::Option<&second::Node<T>>`
         found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`

error[E0308]: mismatched types
--> src/second.rs:85:25
|
85 |     self.next = node.next.as_ref().map(|node| &*node);
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
|
= note: expected type `std::option::Option<&'a second::Node<T>>`
         found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`

```



错误的原因是，`as_ref` 增加了一层间接引用，需要被移除，这里使用另外一种方式来实现：

```

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_deref() }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

$ cargo build

```



`as_deref` 和 `as_deref_mut` 函数在 Rust 1.40 版本中正式稳定下来。在那之前，你只能在 `stable` 版本中使用 `map(|node| &**node)` 和 `map(|node| &mut**node)` 的方式来替代。

大家可能会觉得 `&**` 的形式看上去有些烂，没错，确实如此。但是就像一瓶好酒一样，Rust 也随着时间的推进变得越来越好，因此现在我们已经无需再这么做了。事实上，Rust 很擅长隐式地做类似的转换，或者可以称之为 [Deref](#)。

但是 `Deref` 在这里并不能很好的完成自己的任务，原因是在闭包中使用 `Option<&T>` 而不是 `&T` 对于它来说有些过于复杂了，因此我们需要显式地去帮助它完成任务。好在根据我的经验来看，这种情况还是相当少见的。

事实上，还可以使用另一种方式来实现：

```
self.next = node.next.as_ref().map::<&Node<T>, _>(|node| &node);
```

这种类型暗示的方式可以使用的原因在于 `map` 是一个泛型函数：

```
pub fn map<U, F>(&self, f: F) -> Option<U>
```

turbofish 形式的符号 `::<>` 可以告诉编译器我们希望用哪个具体的类型来替代泛型类型，在这种情况下，`::<&Node<T>, _>` 意味着：它应该返回一个 `&Node<T>`。这种方式可以让编译器知道它需要对 `&node` 应用 `deref`，这样我们就不用手动的添加 `**` 来进行解引用。

好了，既然编译通过，那就写个测试来看看运行结果：

```
#[test]
fn iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&1));
}

$ cargo test

    Running target/debug/lists-5c71138492ad4b4a

running 5 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::peek ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

最后，还有一点值得注意，之前的代码事实上可以应用[生命周期消除原则](#)：

```
impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_deref() }
    }
}
```

这段代码跟以下代码是等价的：

```
impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter { next: self.head.as_deref() }
    }
}
```

当然，如果你就喜欢生命周期那种自由、飘逸的 feeling，还可以使用 Rust 2018 引入的“显式生命周期消除”语法 '`_`'：

```
impl<T> List<T> {
    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}
```

# IterMut以及完整代码

上一章节中我们讲到了要为 `List` 实现三种类型的迭代器并实现了其中两种: `IntoIter` 和 `Iter`。下面再来看看最后一种 `IterMut`。

再来回顾下 `Iter` 的实现:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> { /* stuff */ }
}
```

这段代码可以进行下脱糖( desugar ):

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next<'b>(&'b mut self) -> Option<&'a T> { /* stuff */ }
}
```

可以看出 `next` 方法的输入和输出之间的生命周期并没有关联，这样我们就可以无条件的一遍又一遍地调用 `next`:

```
let mut list = List::new();
list.push(1); list.push(2); list.push(3);

let mut iter = list.iter();
let x = iter.next().unwrap();
let y = iter.next().unwrap();
let z = iter.next().unwrap();
```

对于不可变借用而言，这种方式没有任何问题，因为不可变借用可以同时存在多个，但是如果可是可变引用呢？因此，大家可能会以为使用安全代码来写 `IterMut` 是一件相当困难的事。但是令人诧异的是，事实上，我们可以使用安全的代码来为很多数据结构实现 `IterMut`。

先将之前的代码修改成可变的:

```

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn iter_mut(&self) -> IterMut<'_, T> {
        IterMut { next: self.head.as_deref_mut() }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref_mut();
            &mut node.elem
        })
    }
}
}

$ cargo build

error[E0596]: cannot borrow `self.head` as mutable, as it is behind a `&` reference
--> src/second.rs:95:25
|
94 |     pub fn iter_mut(&self) -> IterMut<'_, T> {
|             ----- help: consider changing this to be a mutable
reference: `&mut self`
95 |         IterMut { next: self.head.as_deref_mut() }
|                 ^^^^^^^^^ `self` is a `&` reference, so the data it
refers to cannot be borrowed as mutable

error[E0507]: cannot move out of borrowed content
--> src/second.rs:103:9
|
103 |     self.next.map(|node| {
|             ^^^^^^^^^ cannot move out of borrowed content
|

```

果不其然，两个错误发生了。第一错误看上去很清晰，甚至告诉了我们该如何解决：

```

pub fn iter_mut(&mut self) -> IterMut<'_, T> {
    IterMut { next: self.head.as_deref_mut() }
}

```

但是另一个好像就没那么容易了。但是之前的代码就可以工作啊，为何这里就不行了？

原因在于有些类型可以 [Copy](#)，有些不行。而 `Option` 和不可变引用 `&T` 恰恰是可以 `Copy` 的，但尴尬的是，可变引用 `&mut T` 不可以，因此这里报错了。

因此我们需要使用 `take` 方法来处理这种情况：

```
fn next(&mut self) -> Option<Self::Item> {
    self.next.take().map(|node| {
        self.next = node.next.as_deref_mut();
        &mut node.elem
    })
}
```

```
$ cargo build
```

老规矩，来测试下：

```
#[test]
fn iter_mut() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter_mut();
    assert_eq!(iter.next(), Some(&mut 3));
    assert_eq!(iter.next(), Some(&mut 2));
    assert_eq!(iter.next(), Some(&mut 1));
}
```

```
$ cargo test
```

```
Running target/debug/lists-5c71138492ad4b4a

running 6 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::peek ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured
```

最终，我们完成了迭代器的功能，下面是完整的代码。

## 完整代码

```
pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }

    pub fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| {
            &node.elem
        })
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        self.head.as_mut().map(|node| {
            &mut node.elem
        })
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
```

```
        Iter { next: self.head.as_deref() }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        IterMut { next: self.head.as_deref_mut() }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}

pub struct IntoIter<T>(List<T>);

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // access fields of a tuple struct numerically
        self.0.pop()
    }
}

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.take().map(|node| {
            self.next = node.next.as_deref_mut();
            &mut node.elem
        })
    }
}
```

```
        }

    }

#[cfg(test)]
mod test {
    use super::List;

#[test]
fn basics() {
    let mut list = List::new();

    // Check empty list behaves right
    assert_eq!(list.pop(), None);

    // Populate list
    list.push(1);
    list.push(2);
    list.push(3);

    // Check normal removal
    assert_eq!(list.pop(), Some(3));
    assert_eq!(list.pop(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push(4);
    list.push(5);

    // Check normal removal
    assert_eq!(list.pop(), Some(5));
    assert_eq!(list.pop(), Some(4));
}

// Check exhaustion
assert_eq!(list.pop(), Some(1));
assert_eq!(list.pop(), None);
}

#[test]
fn peek() {
    let mut list = List::new();
    assert_eq!(list.peek(), None);
    assert_eq!(list.peek_mut(), None);
    list.push(1); list.push(2); list.push(3);

    assert_eq!(list.peek(), Some(&3));
    assert_eq!(list.peek_mut(), Some(&mut 3));

    list.peek_mut().map(|value| {
        *value = 42
    });

    assert_eq!(list.peek(), Some(&42));
    assert_eq!(list.pop(), Some(42));
}
```

```
#[test]
fn into_iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), None);
}

#[test]
fn iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&1));
}

#[test]
fn iter_mut() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter_mut();
    assert_eq!(iter.next(), Some(&mut 3));
    assert_eq!(iter.next(), Some(&mut 2));
    assert_eq!(iter.next(), Some(&mut 1));
}
}
```

# 持久化单向链表

迄今为止，我们已经掌握了如何实现一个可变的单向链表。但是之前的链表都是单所有权的，在实际使用中，共享所有权才是更实用的方式，下面一起来看看该如何实现一个不可变的、共享所有权的持久化链表（persistent）。

开始之前，还需要创建一个新文件 `third.rs`，并在 `lib.rs` 中添加以下内容：

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
```

与上一个链表有所不同，这次我们无需拷贝之前的代码，而是从零开始构建一个新的链表。

# 数据布局和基本操作

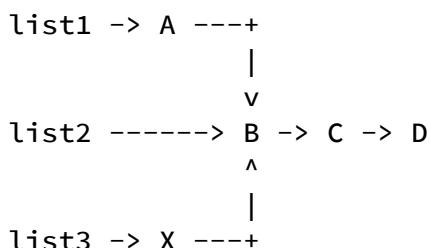
对于新的链表来说，最重要的就是我们可以自由地操控列表的尾部( tail )。

## 数据布局

例如以下是一个不太常见的持久化列表布局：

```
list1 = A -> B -> C -> D
list2 = tail(list1) = B -> C -> D
list3 = push(list2, X) = X -> B -> C -> D
```

如果上面的不够清晰，我们还可以从内存角度来看：



这里大家可能会看出一些端倪：节点 B 被多个链表所共享，这造成了我们无法通过 Box 的方式来实现，因为如果使用 Box，还存在一个问题，谁来负责清理释放？如果 drop list2，那 B 节点会被清理释放吗？

函数式语言或者说其它绝大多数语言，并不存在这个问题，因为 GC 垃圾回收解千愁，但是 Rust 并没有。

好在标准库为我们提供了引用计数的数据结构：Rc / Arc，引用计数可以被认为是一种简单的 GC，对于很多场景来说，引用计数的数据吞吐量要远小于垃圾回收，而且引用计数还存在循环引用的风险！但... 我们有其它选择吗？:(

不过使用 Rc 意味着我们的数据将无法被改变，因为它不具备内部可变性，关于 Rc/Arc 的详细介绍请看[这里](#)。

下面，简单的将我们的数据结构通过 Rc 来实现：

```
// in third.rs
use std::rc::Rc;

pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Rc<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}
```

需要注意的是，`Rc` 在 Rust 中并不是一等公民，它没有被包含在 `std::prelude` 中，因此我们必须手动引入 `use std::rc::Rc` (混得好失败 - , -)

## 基本操作

首先，对于 `List` 的构造器，可以直接复制粘贴：

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }
}
```

而之前的 `push` 和 `pop` 已无任何意义，因为新链表是不可变的，但我们可以使用功能相似的 `prepend` 和 `tail` 来返回新的链表。

```
pub fn prepend(&self, elem: T) -> List<T> {
    List { head: Some(Rc::new(Node {
        elem: elem,
        next: self.head.clone(),
    })) }
}
```

大家可能会大惊失色，什么，你竟然用了 `clone`，不是号称高性能链表实现吗？别急，这里其实只是 `Rc::clone`，对于该方法而言，`clone` 仅仅是增加引用计数，并不是复制底层的数据。虽然 `Rc` 的性能要比 `Box` 的引用方式低一点，但是它依然是多所有权前提下最好的解决方式或者说之一。

还有一点值得注意，`head` 是 `Option<Rc<Node<T>>>` 类型，那么为何不先匹配出内部的 `Rc<Node<T>>`，然后再 `clone` 呢？原因是 `Option` 也提供了相应的 API，它的功能跟我们的需求是一致

的。

运行下试试：

```
$ cargo build

warning: field is never used: `elem`
--> src/third.rs:10:5
10 |     elem: T,
|     ^^^^^^
|
|= note: #[warn(dead_code)] on by default

warning: field is never used: `next`
--> src/third.rs:11:5
11 |     next: Link<T>,
|     ^^^^^^^^^^^^^^
```

胆战心惊的编译通过(胆战心惊? 日常基本操作, 请坐下!)。

继续来实现 `tail`, 该方法会将现有链表的首个元素移除, 并返回剩余的链表:

```
pub fn tail(&self) -> List<T> {
    List { head: self.head.as_ref().map(|node| node.next.clone()) }
}

$ cargo build

error[E0308]: mismatched types
--> src/third.rs:27:22
27 |     List { head: self.head.as_ref().map(|node| node.next.clone()) }
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected
struct `std::rc::Rc`, found enum `std::option::Option`
|
= note: expected type `std::option::Option<std::rc::Rc<_>>`
        found type `std::option::Option<std::option::Option<std::rc::Rc<_>>>`
```

看起来这里的 `map` 多套了一层 `Option`, 可以用 `and_then` 替代:

```
pub fn tail(&self) -> List<T> {
    List { head: self.head.as_ref().and_then(|node| node.next.clone()) }
}
```

顺利通过编译，很棒！最后就是实现 `head` 方法，它返回首个元素的引用，跟之前链表的 `peek` 方法一样：

```
pub fn head(&self) -> Option<&T> {
    self.head.as_ref().map(|node| &node.elem)
}
```

好了，至此，新链表的基本操作都已经实现，最后让我们写几个测试用例来看看它们是骡子还是马：

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let list = List::new();
        assert_eq!(list.head(), None);

        let list = list.prepend(1).prepend(2).prepend(3);
        assert_eq!(list.head(), Some(&3));

        let list = list.tail();
        assert_eq!(list.head(), Some(&2));

        let list = list.tail();
        assert_eq!(list.head(), Some(&1));

        let list = list.tail();
        assert_eq!(list.head(), None);

        // Make sure empty tail works
        let list = list.tail();
        assert_eq!(list.head(), None);
    }
}
```

```
$ cargo test
```

```
Running target/debug/lists-5c71138492ad4b4a

running 5 tests
test first::test::basics ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured
```

哦对了... 我们好像忘了一个重要特性：对链表的迭代。

```
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

#[test]
fn iter() {
    let list = List::new().prepend(1).prepend(2).prepend(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&1));
}

$ cargo test

Running target/debug/lists-5c71138492ad4b4a

running 7 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured
```

细心的同学可能会觉得我在凑字数，这不跟之前的链表迭代实现一样一样的嘛？恭喜你答对了：）

最后，给大家留个作业，你可以尝试下看能不能实现 `IntoIter` 和 `IterMut`，如果实现不了请不要打我，冤有头债有主，都是 `Rc` 惹的祸 :)

# Drop、Arc 及完整代码

## Drop

与之前链表存在的问题相似，新的链表也有递归的问题。下面是之前的解决方法：

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

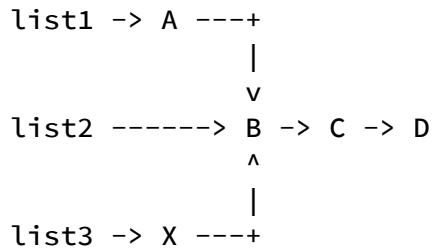
但是 `boxed_node.next.take()` 的方式在新的链表中无法使用，因为我们没办法去修改 Rc 持有的值。

考虑一下相关的逻辑，可以发现，如果当前的节点仅被当前链表所引用(Rc 的引用计数为 1)，那该节点是可以安全 drop 的：

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut head = self.head.take();
        while let Some(node) = head {
            if let Ok(mut node) = Rc::try_unwrap(node) {
                head = node.next.take();
            } else {
                break;
            }
        }
    }
}
```

这里有一个没见过的方法 `Rc::Try_unwrap`，该方法会判断当前的 Rc 是否只有一个强引用，若是，则返回 Rc 持有的值，否则返回一个错误。

可以看出，我们会一直 drop 到第一个被其它链表所引用的节点：



例如如果要 drop List2，那会从头节点开始一直 drop 到 B 节点时停止，剩余的 B -> C -> D 三个节点由于引用计数不为 1 (同时被多个链表引用)，因此不会被 drop。

测试下新的代码：

```

$ cargo test

Compiling lists v0.1.0 (/Users/ABeingessner/dev/too-many-lists/lists)
Finished dev [unoptimized + debuginfo] target(s) in 1.10s
    Running /Users/ABeingessner/dev/too-many-lists/lists/target/debug/deps/lists-
86544f1d97438f1f

running 8 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 8 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

完美通过，下面再来考虑一个问题，如果我们的链表要在多线程环境使用该怎么办？

## Arc

不可变链表的一个很大的好处就在于多线程访问时自带安全性，毕竟共享可变性是多线程危险的源泉，最好也是最简单的解决办法就是直接干掉可变性。

但是 `Rc<T>` 本身并不是线程安全的，原因在之前的章节也有讲：它内部的引用计数器并不是线程安全的，通俗来讲，计数器没有加锁也没有实现原子性。

再结合之前章节学过的内容，绝大部分同学应该都能想到，`Arc<T>` 就是我们的最终答案。

那么还有一个问题，我们怎么知道一个类型是不是类型安全？会不会在多线程误用了非线程安全的类型呢？这就是 Rust 安全性的另一个强大之处：Rust 通过提供 `Send` 和 `Sync` 两个特征来保证线程安全。

---

关于 `Send` 和 `Sync` 的详细介绍，请参见[此章节](#)

---

## 完整代码

又到了喜闻乐见的环节，新链表的代码相比之前反而还更简单了，不可变就是香！

```
use std::rc::Rc;

pub struct List<T> {
    head: Link<T>,
}

type Link<T> = Option<Rc<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn prepend(&self, elem: T) -> List<T> {
        List { head: Some(Rc::new(Node {
            elem: elem,
            next: self.head.clone(),
        })) }
    }

    pub fn tail(&self) -> List<T> {
        List { head: self.head.as_ref().and_then(|node| node.next.clone()) }
    }

    pub fn head(&self) -> Option<&T> {
        self.head.as_ref().map(|node| &node.elem)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut head = self.head.take();
        while let Some(node) = head {
            if let Ok(mut node) = Rc::try_unwrap(node) {
                head = node.next.take();
            } else {
                break;
            }
        }
    }
}

pub struct Iter<'a, T> {
```

```
    next: Option<&'a Node<T>>,  
}  
  
impl<'a, T> Iterator for Iter<'a, T> {  
    type Item = &'a T;  
  
    fn next(&mut self) -> Option<Self::Item> {  
        self.next.map(|node| {  
            self.next = node.next.as_deref();  
            &node.elem  
        })  
    }  
}  
  
#[cfg(test)]  
mod test {  
    use super::List;  
  
    #[test]  
    fn basics() {  
        let list = List::new();  
        assert_eq!(list.head(), None);  
  
        let list = list.prepend(1).prepend(2).prepend(3);  
        assert_eq!(list.head(), Some(&3));  
  
        let list = list.tail();  
        assert_eq!(list.head(), Some(&2));  
  
        let list = list.tail();  
        assert_eq!(list.head(), Some(&1));  
  
        let list = list.tail();  
        assert_eq!(list.head(), None);  
  
        // Make sure empty tail works  
        let list = list.tail();  
        assert_eq!(list.head(), None);  
    }  
  
    #[test]  
    fn iter() {  
        let list = List::new().prepend(1).prepend(2).prepend(3);  
  
        let mut iter = list.iter();  
        assert_eq!(iter.next(), Some(&3));  
        assert_eq!(iter.next(), Some(&2));  
        assert_eq!(iter.next(), Some(&1));  
    }  
}
```

# 不太优秀的双端队列

在实现了之前的队列后，我们不禁浮想联翩，如果 `Rc` 是可变的，那是不是可以实现一个双向链表？

心动不如行动，先来创建新的链表文件 `fourth.rs`，并在 `src/lib.rs` 中添加以下内容：

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
pub mod fourth;
```

依然是熟悉的从零开始，当然，也依然会用到熟悉的 CV 配方。

---

声明：大家看到目录名时，心里就应该在嘀咕了吧？其实你的嘀咕是对的，是的，本章的目的是为了证明之前的想法是糟糕的！

---

# 数据布局和构建

聪明的读者应该已经想到了：让 `Rc` 可变，就需要使用 `RefCell` 的配合。关于 `RefCell` 的一切，在之前的章节都有介绍，还不熟悉的同学请移步[这里](#)。

好了，绝世神兵在手，接下来...我们将见识一个绝世啰嗦的数据结构...如果你来自 GC 语言，那很可能就没有见识过这种阵仗。

## 数据布局

双向链表意味着每一个节点将同时指向前一个和下一个节点，因此我们的数据结构可能会变成这样：

```
use std::rc::Rc;
use std::cell::RefCell;

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>;
```

```
struct Node<T> {
    elem: T,
    next: Link<T>,
    prev: Link<T>,
}
```

耳听忐忑，心怀忐忑，尝试编译下，竟然顺利通过了，thanks god! 接下来再来看看该如何使用它。

## 构建

如果按照之前的构建方式来构建新的数据结构，会有点笨拙，因此我们先尝试将其拆分：

```
impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }
}
```

> cargo build

\*\*一大堆 DEAD CODE 警告，但是好歹可以成功编译\*\*

## Push

很好，再来向链表的头部推入一个元素。由于双向链表的数据结构和操作逻辑明显更加复杂，因此相比单向链表的单行实现，双向链表的 `push` 操作也要复杂的多。

除此之外，我们还需要处理一些关于空链表的边界问题：对于绝大部分操作而言，可能只需要使用 `head` 或 `tail` 指针，但是对于空链表，则需要同时使用它们。

一个验证方法 `methods` 是否有效的办法就是看它是否能保持不变性，每个节点都应该有两个指针指向它：中间的节点被它前后的节点所指向，而头部的节点除了被它后面的节点所指向外，还会被链表本身所指向，尾部的节点亦是如此。

```

pub fn push_front(&mut self, elem: T) {
    let new_head = Node::new(elem);
    match self.head.take() {
        Some(old_head) => {
            // 非空链表，将新的节点跟老的头部相链接
            old_head.prev = Some(new_head.clone());
            new_head.next = Some(old_head);
            self.head = Some(new_head);
        }
        None => {
            // 空链表，需要设置 tail 和 head
            self.tail = Some(new_head.clone());
            self.head = Some(new_head);
        }
    }
}

> cargo build

error[E0609]: no field `prev` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
--> src/fourth.rs:39:26
|
39 |         old_head.prev = Some(new_head.clone()); // +1 new_head
|             ^^^^ unknown field

error[E0609]: no field `next` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
--> src/fourth.rs:40:26
|
40 |         new_head.next = Some(old_head);           // +1 old_head
|             ^^^^ unknown field

```

虽然有报错，但是一切尽在掌握，今天真是万事顺利啊！

从报错来看，我们无法直接去访问 `prev` 和 `next`，回想一下 `RefCell` 的使用方式，修改代码如下：

```
pub fn push_front(&mut self, elem: T) {
    let new_head = Node::new(elem);
    match self.head.take() {
        Some(old_head) => {
            old_head.borrow_mut().prev = Some(new_head.clone());
            new_head.borrow_mut().next = Some(old_head);
            self.head = Some(new_head);
        }
        None => {
            self.tail = Some(new_head.clone());
            self.head = Some(new_head);
        }
    }
}
```

```
$ cargo build
```

```
warning: field is never used: `elem`
--> src/fourth.rs:12:5
12 |     elem: T,
   |     ^^^^^^
   |
= note: #[warn(dead_code)] on by default
```

嘿，我又可以了！既然状态神勇，那就趁热打铁，再来看看 `pop`。

## Pop

如果说 `new` 和 `push` 是在构建链表，那 `pop` 显然就是一个破坏者。

何为完美的破坏？按照构建的过程逆着来一遍就是完美的！

```
pub fn pop_front(&mut self) -> Option<T> {
    self.head.take().map(|old_head| {
        match old_head.borrow_mut().next.take() {
            Some(new_head) => {
                // 非空链表
                new_head.borrow_mut().prev.take();
                self.head = Some(new_head);
            }
            None => {
                // 空链表
                self.tail.take();
            }
        }
        old_head.elem
    })
}
```

```
$ cargo build
```

```
error[E0609]: no field `elem` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
--> src/fourth.rs:64:22
|
64 |         old_head.elem
|             ^^^^^ unknown field
```

哎，怎么就不长记性呢，又是 RefCell 赖的祸：

```
pub fn pop_front(&mut self) -> Option<T> {
    self.head.take().map(|old_head| {
        match old_head.borrow_mut().next.take() {
            Some(new_head) => {
                new_head.borrow_mut().prev.take();
                self.head = Some(new_head);
            }
            None => {
                self.tail.take();
            }
        }
        old_head.borrow_mut().elem
    })
}
```

```
$ cargo build

error[E0507]: cannot move out of borrowed content
--> src/fourth.rs:64:13
|
64 |         old_head.borrow_mut().elem
|         ^^^^^^^^^^^^^^^^^^^^^^^^^ cannot move out of borrowed content
```

额... 我凌乱了，看上去 Box 是罪魁祸首，borrow\_mut 只能返回一个 &mut Node<T>，因此无法拿走其所有权。

我们需要一个方法来拿走 RefCell<T> 的所有权，然后返回给我们一个 T，翻一翻[文档](#)，可以发现下面这段内容：

---

```
fn into_inner(self) -> T
```

---

消费掉 RefCell 并返回内部的值

---

喔，看上去好有安全感的方法：

```
old_head.into_inner().elem

$ cargo build

error[E0507]: cannot move out of an `Rc`
--> src/fourth.rs:64:13
|
64 |         old_head.into_inner().elem
|         ^^^^^^^ cannot move out of an `Rc`
```

...看走眼了，没想到你浓眉大眼也会耍花枪。into\_inner 想要拿走 RefCell 的所有权，但是还有一个 Rc 不愿意，因为 Rc<T> 只能让我们获取内部值的不可变引用。

大家还记得我们之前实现 Drop 时用过的方法吗？在这里一样适用：

```
Rc::try_unwrap(old_head).unwrap().into_inner().elem
```

Rc::try\_unwrap 返回一个 Result，由于我们不关心 Err 的情况(如果代码合理，这里不会是 Err)，直接使用 unwrap 即可。

```
$ cargo build

error[E0599]: no method named `unwrap` found for type
`std::result::Result<std::cell::RefCell<fourth::Node<T>>,
std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>` in the current scope
--> src/fourth.rs:64:38
|
64 |         Rc::try_unwrap(old_head).unwrap().into_inner().elem
|                         ^^^^^^
|
= note: the method `unwrap` exists but the following trait bounds were not
satisfied:
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>> : std::fmt::Debug`
```

额，`unwrap` 要求目标类型是实现了 `Debug` 的，这样才能在报错时提供 `debug` 输出，而 `RefCell<T>` 要实现 `Debug` 需要它内部的 `T` 实现 `Debug`，而我们的 `Node` 并没有实现。

当然，我们可以选择为 `Node` 实现，也可以这么做：

```
Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
```

```
$ cargo build
```

终于成功的运行了，下面依然是惯例 - 写几个测试用例：

```

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop_front(), None);

        // Populate list
        list.push_front(1);
        list.push_front(2);
        list.push_front(3);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(3));
        assert_eq!(list.pop_front(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push_front(4);
        list.push_front(5);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(5));
        assert_eq!(list.pop_front(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop_front(), Some(1));
        assert_eq!(list.pop_front(), None);
    }
}

```

\$ cargo test

```

Running target/debug/lists-5c71138492ad4b4a

running 9 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::basics ... ok
test fifth::test::iter_mut ... ok
test third::test::basics ... ok
test second::test::iter ... ok
test third::test::iter ... ok
test second::test::into_iter ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured

```

## Drop

在[循环引用](#)章节，我们介绍过 `Rc` 最怕的就是引用形成循环，而双向链表恰恰如此。因此，当使用默认的实现来 `drop` 我们的链表时，两个节点会将各自的引用计数减少到 1，然后就不会继续减少，最终造成内存泄漏。

所以，这里最好的实现就是将每个节点 `pop` 出去，直到获得 `None`：

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}
```

细心的读者可能已经注意到，我们还未实现在链表尾部 `push` 和 `pop` 的操作，但由于所需的实现跟之前差别不大，因此我们会在后面直接给出，下面先来看看更有趣的。

# Peek

`push` 和 `pop` 的防不胜防的编译报错着实让人出了些冷汗，下面来看看轻松的，至少在之前的链表中是很轻松的：）

```
pub fn peek_front(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        &node.elem
    })
}
```

额...好像被人发现我是复制黏贴的了，赶紧换一个：

```
pub fn peek_front(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        // BORROW!!!!
        &node.borrow().elem
    })
}

$ cargo build

error[E0515]: cannot return value referencing temporary value
--> src/fourth.rs:66:13
66 |         &node.borrow().elem
   |         ^-----^^^^^
   |         |
   |         temporary value created here
   |
   |         returns a value referencing data owned by the current function
```

从报错可以看出，原因是我们在引用了局部的变量并试图在函数中返回。为了解释这个问题，先来看看 `borrow` 的定义：

```
fn borrow<'a>(&'a self) -> Ref<'a, T>
fn borrow_mut<'a>(&'a self) -> RefMut<'a, T>
```

这里返回的并不是 `&T` 或 `&mut T`，而是一个 `Ref` 和 `RefMut`，那么它们是什么？说白了，它们就是在借用到的引用外包裹了一层。而且 `Ref` 和 `RefMut` 分别实现了 `Deref` 和 `DerefMut`，在绝大多数场景中，我们都可以像使用 `&T` 一样去使用它们。

只能说是成是败都赖萧何，恰恰就因为这一层包裹，导致生命周期改变了，也就是 `Ref` 和内部引用的生命周期不再和 `RefCell` 相同，而 `Ref` 的生命周期是什么，相信大家都能看得出来，因此就造成了局部引用的问题。

事实上，这是必须的，如果内部的引用和外部的 `Ref` 生命周期不一致，那该如何管理？当 `Ref` 因超出作用域被 `drop` 时，内部的引用怎么办？

现在该怎么办？我们只想要一个引用，现在却多了一个 `Ref` 拦路虎。等等，如果我们不返回 `&T` 而是返回 `Ref` 呢？

```
use std::cell::{Ref, RefCell};

pub fn peek_front(&self) -> Option<Ref<T>> {
    self.head.as_ref().map(|node| {
        node.borrow()
    })
}

$ cargo build

error[E0308]: mismatched types
--> src/fourth.rs:64:9
|
64 |     self.head.as_ref().map(|node| {
65 |         node.borrow()
66 |     })
|-----^ expected type parameter, found struct `fourth::Node`
|
= note: expected type `std::option::Option<std::cell::Ref<'_, T>>`
         found type `std::option::Option<std::cell::Ref<'_, fourth::Node<T>>>`
```

嗯，类型不匹配了，要返回的是 `Ref<T>` 但是获取的却是 `Ref<Node<T>>`，那么现在看上去有两个选择：

- 抛弃这条路，换一条重新开始
- 一条路走到死，最终通过更复杂的实现来解决

但是，仔细想想，这两个选择都不是我们想要的，那没办法了，只能继续深挖，看看有没有其它解决办法。啊哦，还真发现了一只野兽：

```
map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
where F: FnOnce(&T) -> &U,
      U: ?Sized
```

就像在 `Result` 和 `Option` 上使用 `map` 一样，我们还能在 `Ref` 上使用 `map`：

```
pub fn peek_front(&self) -> Option<Ref<T>> {
    self.head.as_ref().map(|node| {
        Ref::map(node.borrow(), |node| &node.elem)
    })
}
```

```
$ cargo build
```

Gooood! 本章节的编译错误可以说是多个链表中最难解决的之一，依然被我们成功搞定了！

下面来写下测试用例，需要注意的是 `Ref` 不能被直接比较，因此我们需要先利用 `Deref` 解引用出其中的值，再进行比较。

```
#[test]
fn peek() {
    let mut list = List::new();
    assert!(list.peek_front().is_none());
    list.push_front(1); list.push_front(2); list.push_front(3);

    assert_eq!(&*list.peek_front().unwrap(), &3);
}
```

```
$ cargo test
```

```
Running target/debug/lists-5c71138492ad4b4a

running 10 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test fourth::test::peek ... ok
test second::test::iter_mut ... ok
test second::test::into_iter ... ok
test third::test::basics ... ok
test second::test::peek ... ok
test second::test::iter ... ok
test third::test::iter ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured
```

终于可以把文章开头的冷汗擦拭干净了，忘掉这个章节吧，让我来养你...哦不对，让我们开始一段真正轻松的章节。

# 基本操作的对称镜像

之前我们仅实现了头部的 `push`、`pop`，现在来补全一下，大自然的对称之美咱的双向链表也不能少了。

```
tail <-> head
next <-> prev
front -> back
```

需要注意的是，这里还新增了 `mut` 类型的 `peek`:

```
use std::cell::{Ref, RefCell, RefMut};

// ..

pub fn push_back(&mut self, elem: T) {
    let new_tail = Node::new(elem);
    match self.tail.take() {
        Some(old_tail) => {
            old_tail.borrow_mut().next = Some(new_tail.clone());
            new_tail.borrow_mut().prev = Some(old_tail);
            self.tail = Some(new_tail);
        }
        None => {
            self.head = Some(new_tail.clone());
            self.tail = Some(new_tail);
        }
    }
}

pub fn pop_back(&mut self) -> Option<T> {
    self.tail.take().map(|old_tail| {
        match old_tail.borrow_mut().prev.take() {
            Some(new_tail) => {
                new_tail.borrow_mut().next.take();
                self.tail = Some(new_tail);
            }
            None => {
                self.head.take();
            }
        }
        Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
    })
}

pub fn peek_back(&self) -> Option<Ref<T>> {
    self.tail.as_ref().map(|node| {
        Ref::map(node.borrow(), |node| &node.elem)
    })
}

pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
    self.tail.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}

pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
    self.head.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}
```

再更新测试用例:

```
#[test]
fn basics() {
    let mut list = List::new();

    // Check empty list behaves right
    assert_eq!(list.pop_front(), None);

    // Populate list
    list.push_front(1);
    list.push_front(2);
    list.push_front(3);

    // Check normal removal
    assert_eq!(list.pop_front(), Some(3));
    assert_eq!(list.pop_front(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push_front(4);
    list.push_front(5);

    // Check normal removal
    assert_eq!(list.pop_front(), Some(5));
    assert_eq!(list.pop_front(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop_front(), Some(1));
    assert_eq!(list.pop_front(), None);

    // ---- back ----

    // Check empty list behaves right
    assert_eq!(list.pop_back(), None);

    // Populate list
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);

    // Check normal removal
    assert_eq!(list.pop_back(), Some(3));
    assert_eq!(list.pop_back(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push_back(4);
    list.push_back(5);

    // Check normal removal
    assert_eq!(list.pop_back(), Some(5));
    assert_eq!(list.pop_back(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop_back(), Some(1));
```

```

    assert_eq!(list.pop_back(), None);
}

#[test]
fn peek() {
    let mut list = List::new();
    assert!(list.peek_front().is_none());
    assert!(list.peek_back().is_none());
    assert!(list.peek_front_mut().is_none());
    assert!(list.peek_back_mut().is_none());

    list.push_front(1); list.push_front(2); list.push_front(3);

    assert_eq!(&*list.peek_front().unwrap(), &3);
    assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
    assert_eq!(&*list.peek_back().unwrap(), &1);
    assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
}

```

什么？你问我这里的测试用例全吗？只能说如果测试全部的组合情况，这一章节会被撑爆。至于现在，能不出错就谢天谢地了：（

```

$ cargo test

Running target/debug/lists-5c71138492ad4b4a

running 10 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test fourth::test::peek ... ok
test second::test::iter ... ok
test third::test::iter ... ok
test second::test::into_iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured

```

我想说：Ctrl CV 是最好的编程工具，大家同意吗？

# 迭代器

坏男孩最令人头疼，而链表实现中，迭代器就是这样的坏男孩，所以我们放在最后来处理。

## IntoIter

由于是转移所有权，因此 `IntoIter` 一直都是最好实现的：

```
pub struct IntoIter<T>(List<T>);

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        self.0.pop_front()
    }
}
```

但是关于双向链表，有一个有趣的事，它不仅可以从前向后迭代，还能反过来。前面实现的是传统的从前到后，那问题来了，反过来该如何实现呢？

答案是：`DoubleEndedIterator`，它继承自 `Iterator`（通过 `supertrait`），因此意味着要实现该特征，首先需要实现 `Iterator`。

这样只要为 `DoubleEndedIterator` 实现 `next_back` 方法，就可以支持双向迭代了：`Iterator` 的 `next` 方法从前往后，而 `next_back` 从后向前。

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.0.pop_back()
    }
}
```

测试下：

```

#[test]
fn into_iter() {
    let mut list = List::new();
    list.push_front(1); list.push_front(2); list.push_front(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next_back(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next_back(), None);
    assert_eq!(iter.next(), None);
}

```

\$ cargo test

```

Running target/debug/lists-5c71138492ad4b4a

running 11 tests
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test fourth::test::into_iter ... ok
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test third::test::iter ... ok
test third::test::basics ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok

test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured

```

## Iter

这里又要用到糟糕的 Ref :

```

pub struct Iter<'a, T>(Option<Ref<'a, Node<T>>>);

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter(self.head.as_ref().map(|head| head.borrow()))
    }
}

```

\$ cargo build

迄今为止一切运行正常，接下来的 next 实现起来会有些麻烦：

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = Ref<'a, T>;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node_ref| {
            self.0 = node_ref.next.as_ref().map(|head| head.borrow());
            Ref::map(node_ref, |node| &node.elem)
        })
    }
}

$ cargo build

error[E0521]: borrowed data escapes outside of closure
--> src/fourth.rs:155:13
|
153 |     fn next(&mut self) -> Option<Self::Item> {
|         ----- `self` is declared here, outside of the closure body
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.borrow());
|                 ^^^^^^ ----- borrow is only valid in the closure body
|                 |
|                 reference to `node_ref` escapes the closure body here

error[E0505]: cannot move out of `node_ref` because it is borrowed
--> src/fourth.rs:156:22
|
153 |     fn next(&mut self) -> Option<Self::Item> {
|         ----- lifetime `'1` appears in the type of `self`
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.borrow());
|                 ----- borrow of `node_ref` occurs here
|                 |
|                 assignment requires that `node_ref` is borrowed for `'1`
156 |             Ref::map(node_ref, |node| &node.elem)
|                 ^^^^^^^^^ move out of `node_ref` occurs here
```

果然，膝盖又中了一箭。

node\_ref 活得不够久，跟一般的引用不同，Rust 不允许我们这样分割 Ref，从 head.borrow() 中取出的 Ref 只允许跟 node\_ref 活得一样久。

而我们想要的函数是存在的：

```
pub fn map_split<U, V, F>(orig: Ref<'b, T>, f: F) -> (Ref<'b, U>, Ref<'b, V>) where
    F: FnOnce(&T) -> (&U, &V),
    U: ?Sized,
    V: ?Sized,
```

喔，这个函数定义的泛型直接晃瞎了我的眼睛。。

```
fn next(&mut self) -> Option<Self::Item> {
    self.0.take().map(|node_ref| {
        let (next, elem) = Ref::map_split(node_ref, |node| {
            (&node.next, &node.elem)
        });
        self.0 = next.as_ref().map(|head| head.borrow());
        elem
    })
}

$ cargo build

Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0521]: borrowed data escapes outside of closure
--> src/fourth.rs:159:13
|
153 |     fn next(&mut self) -> Option<Self::Item> {
|         ----- `self` is declared here, outside of the closure body
...
159 |         self.0 = next.as_ref().map(|head| head.borrow());
|         ^^^^^^ ---- borrow is only valid in the closure body
|         |
|         reference to `next` escapes the closure body here
```

额，借用的内容只允许在闭包体中使用，看起来我们还是得用 `Ref::map` 来解决问题：

```
fn next(&mut self) -> Option<Self::Item> {
    self.0.take().map(|node_ref| {
        let (next, elem) = Ref::map_split(node_ref, |node| {
            (&node.next, &node.elem)
        });

        self.0 = if next.is_some() {
            Some(Ref::map(next, |next| &**next.as_ref().unwrap()))
        } else {
            None
        };
        elem
    })
}
```

```
error[E0308]: mismatched types
--> src/fourth.rs:162:22
|
162 |             Some(Ref::map(next, |next| &**next.as_ref().unwrap())))
|                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected
struct `fourth::Node`, found struct `std::cell::RefCell`  

|  
= note: expected type `std::cell::Ref<'_, fourth::Node<_>>`  
      found type `std::cell::Ref<'_, std::cell::RefCell<fourth::Node<_>>>`
```

晕，多了一个 RefCell，随着我们的对链表的逐步深入，RefCell 的代码嵌套变成了不可忽视的问题。

看起来我们已经无能为力了，只能试着去摆脱 RefCell 了。Rc 怎么样？我们完全可以对 Rc 进行完整的克隆：

```
pub struct Iter<T>(Option<Rc<Node<T>>>);

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter(self.head.as_ref().map(|head| head.clone()))
    }
}

impl<T> Iterator for Iter<T> {
    type Item =
```

等等，那现在返回的是什么？&T 还是 Ref<T>？

两者都不是，现在我们的 Iter 已经没有生命周期了：无论是 &T 还是 Ref<T> 都需要我们在 next 之前声明好生命周期。但是我们试图从 Rc 中取出来的值其实是迭代器的引用。

也可以通过对 Rc 进行 map 获取到 Rc<T>？但是标准库并没有给我们提供相应功能，第三方倒是有一个。

但是，即使这么做了，还有一个更大的坑在等着：一个会造成迭代器不合法的可怕幽灵。事实上，之前我们对于迭代器不合法是免疫的，但是一旦迭代器产生 Rc，那它们就不再会借用链表。这意味着人们可以在持有指向链表内部的指针时，还可以进行 push 和 pop 操作。

严格来说，push 问题不大，因为链表两端的增长不会对我们正在关注的某个子链表造成影响。

但是 pop 就是另一个故事了，如果在我们关注的子链表之外 pop，那问题不大。但是如果 pop 一个正在引用的子链表中的节点呢？那一切就完了，特别是，如果大家还试图去 unwrap try\_unwrap 返回的 Result，会直接造成整个程序的 panic。

仔细想一想，好像也不错，程序一切正常，除非去 pop 我们正在引用的节点，最美的是，就算遇到这种情况，程序也会直接崩溃，提示我们错误的发生。

其实我们大部分的努力都是为了实现隐藏的细节和优雅的 API，典型的二八原则，八成时间花在二成的细节上。但是如果都不关心这些细节，可以接受自己的平凡的话，那把节点简单的到处传递就行。

总之，可以看出，内部可变性非常适合写一个安全性的应用程序，但是如果是安全性高的库，那内部可变性就有些捉襟见肘了。

最终，我选择了放弃，不再实现 `Iter` 和 `IterMut`，也许努力下，可以实现，但是。。。不愉快，算了。

# 最终代码

这一章真不好写(也很难翻译...), 最终我们实现了一个 100% 安全但是功能残缺的双向链表。

同时在实现中, 还有大量 `Rc` 和 `RefCell` 引起的运行时检查, 最终会影响链表的性能。整个双向链表实现史就是一部别名和所有权的奋斗史。

总之, 不管爱与不爱, 它就这样了, 特别是如果我们不在意内部的细节暴露给外面用户时。

而从下一章开始, 我们将实现一个真正能够全盘掌控的链表, 当然...通过 `unsafe` 代码实现!

```

#![allow(unused)]
fn main() {
use std::rc::Rc;
use std::cell::{Ref, RefMut, RefCell};

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
    prev: Link<T>,
}

impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push_front(&mut self, elem: T) {
        let new_head = Node::new(elem);
        match self.head.take() {
            Some(old_head) => {
                old_head.borrow_mut().prev = Some(new_head.clone());
                new_head.borrow_mut().next = Some(old_head);
                self.head = Some(new_head);
            }
            None => {
                self.tail = Some(new_head.clone());
                self.head = Some(new_head);
            }
        }
    }

    pub fn push_back(&mut self, elem: T) {
        let new_tail = Node::new(elem);

```

```

        match self.tail.take() {
            Some(old_tail) => {
                old_tail.borrow_mut().next = Some(new_tail.clone());
                new_tail.borrow_mut().prev = Some(old_tail);
                self.tail = Some(new_tail);
            }
            None => {
                self.head = Some(new_tail.clone());
                self.tail = Some(new_tail);
            }
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        self.tail.take().map(|old_tail| {
            match old_tail.borrow_mut().prev.take() {
                Some(new_tail) => {
                    new_tail.borrow_mut().next.take();
                    self.tail = Some(new_tail);
                }
                None => {
                    self.head.take();
                }
            }
            Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
        })
    }

    pub fn pop_front(&mut self) -> Option<T> {
        self.head.take().map(|old_head| {
            match old_head.borrow_mut().next.take() {
                Some(new_head) => {
                    new_head.borrow_mut().prev.take();
                    self.head = Some(new_head);
                }
                None => {
                    self.tail.take();
                }
            }
            Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
        })
    }

    pub fn peek_front(&self) -> Option<Ref<T>> {
        self.head.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }

    pub fn peek_back(&self) -> Option<Ref<T>> {
        self.tail.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }
}

```

```
}

pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
    self.tail.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}

pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
    self.head.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}

pub fn into_iter(self) -> IntoIter<T> {
    IntoIter(self)
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}

pub struct IntoIter<T>(List<T>);

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        self.0.pop_front()
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.0.pop_back()
    }
}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop_front(), None);

        // Populate list
    }
}
```

```
list.push_front(1);
list.push_front(2);
list.push_front(3);

// Check normal removal
assert_eq!(list.pop_front(), Some(3));
assert_eq!(list.pop_front(), Some(2));

// Push some more just to make sure nothing's corrupted
list.push_front(4);
list.push_front(5);

// Check normal removal
assert_eq!(list.pop_front(), Some(5));
assert_eq!(list.pop_front(), Some(4));

// Check exhaustion
assert_eq!(list.pop_front(), Some(1));
assert_eq!(list.pop_front(), None);

// ---- back ----

// Check empty list behaves right
assert_eq!(list.pop_back(), None);

// Populate list
list.push_back(1);
list.push_back(2);
list.push_back(3);

// Check normal removal
assert_eq!(list.pop_back(), Some(3));
assert_eq!(list.pop_back(), Some(2));

// Push some more just to make sure nothing's corrupted
list.push_back(4);
list.push_back(5);

// Check normal removal
assert_eq!(list.pop_back(), Some(5));
assert_eq!(list.pop_back(), Some(4));

// Check exhaustion
assert_eq!(list.pop_back(), Some(1));
assert_eq!(list.pop_back(), None);
}

#[test]
fn peek() {
    let mut list = List::new();
    assert!(list.peek_front().is_none());
    assert!(list.peek_back().is_none());
    assert!(list.peek_front_mut().is_none());
```

```
assert!(list.peek_back_mut().is_none());

list.push_front(1); list.push_front(2); list.push_front(3);

assert_eq!(&*list.peek_front().unwrap(), &3);
assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
assert_eq!(&*list.peek_back().unwrap(), &1);
assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
}

#[test]
fn into_iter() {
    let mut list = List::new();
    list.push_front(1); list.push_front(2); list.push_front(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next_back(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next_back(), None);
    assert_eq!(iter.next(), None);
}
}
```

# 不错的unsafe队列

在之前章节中，基于内部可变性和引用计数的双向链表有些失控了，原因在于 `Rc` 和 `RefCell` 对于简单的任务而言，它们是非常称职的，但是对于复杂的任务，它们可能会变得相当笨拙，特别是当我们试图隐藏一些细节时。

总之，一定有更好的办法！下面来看看该如何使用裸指针和 `unsafe` 代码实现一个单向链表。

---

大家可能想等着看我犯错误，`unsafe` 嘛，不犯错误不可能的，但是呢，俺偏就不犯错误：）

---

国际惯例，添加第五个链表所需的文件 `fifth.rs`：

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
pub mod fourth;
pub mod fifth;
```

虽然我们依然会从零开始撸代码，但是 `fifth.rs` 的代码会跟 `second.rs` 存在一定的重叠，因为对于链表而言，队列其实就是栈的增强。

# 数据布局

那么单向链表的队列长什么样？对于栈来说，我们向一端推入( push )元素，然后再从同一端弹出( pop )。对于栈和队列而言，唯一的区别在于队列从末端弹出。

栈的实现类似于下图：

```
input list:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)  
  
stack push X:  
[Some(ptr)] -> (X, Some(ptr)) -> (A, Some(ptr)) -> (B, None)  
  
stack pop:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)
```

由于队列是首端进，末端出，因此我们需要决定将 push 和 pop 中的那个放到末端去操作，如果将 push 放在末端操作：

```
input list:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)  
  
flipped push X:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, Some(ptr)) -> (X, None)
```

而如果将 pop 放在末端：

```
input list:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, Some(ptr)) -> (X, None)  
  
flipped pop:  
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)
```

但是这样实现有一个很糟糕的地方：两个操作都需要遍历整个链表后才能完成。队列要求 push 和 pop 操作需要高效，但是遍历整个链表才能完成的操作怎么看都谈不上高效！

其中一个解决办法就是保存一个指针指向末端：

```

use std::mem;

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>, // NEW!
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            // 在尾端推入一个新节点时，新节点的下一个节点永远是 None
            next: None,
        });

        // 让 tail 指向新的节点，并返回之前的 old tail
        let old_tail = mem::replace(&mut self.tail, Some(new_tail));

        match old_tail {
            Some(mut old_tail) => {
                // 若 old tail 存在，则让该节点指向新的节点
                old_tail.next = Some(new_tail);
            }
            None => {
                // 否则，将 head 指向新的节点
                self.head = Some(new_tail);
            }
        }
    }
}

```

在之前的各种链表锤炼下，我们对于相关代码应该相当熟悉了，因此可以适当提提速 - 在写的过程中，事实上我碰到了很多错误，这些错误就不再一一列举。

但是如果你担心不再能看到错误，那就纯属多余了：

```
$ cargo build

error[E0382]: use of moved value: `new_tail`
 --> src/fifth.rs:38:38
|
26 |         let new_tail = Box::new(Node {
|             ----- move occurs because `new_tail` has type
`std::boxed::Box<fifth::Node<T>>`, which does not implement the `Copy` trait
...
33 |         let old_tail = mem::replace(&mut self.tail, Some(new_tail));
|                                         ----- value moved
here
...
38 |             old_tail.next = Some(new_tail);
|                     ^^^^^^^^^ value used here after move
```

新鲜出炉的错误，接好！`Box` 并没有实现 `Copy` 特征，因此我们不能在两个地方进行赋值。好在，可以使用没有所有权的引用类型：

```

pub struct List<T> {
    head: Link<T>,
    tail: Option<&mut Node<T>>, // NEW!
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            next: None,
        });

        let new_tail = match self.tail.take() {
            Some(old_tail) => {
                old_tail.next = Some(new_tail);
                old_tail.next.as_deref_mut()
            }
            None => {
                self.head = Some(new_tail);
                self.head.as_deref_mut()
            }
        };
        self.tail = new_tail;
    }
}

$ cargo build

error[E0106]: missing lifetime specifier
--> src/fifth.rs:3:18
|
3 |     tail: Option<&mut Node<T>>, // NEW!
|           ^ expected lifetime parameter

```

好吧，结构体中的引用类型需要显式的标注生命周期，先加一个 'a 吧：

```
pub struct List<'a, T> {
    head: Link<T>,
    tail: Option<&'a mut Node<T>>, // NEW!
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<'a, T> List<'a, T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            next: None,
        });

        let new_tail = match self.tail.take() {
            Some(old_tail) => {
                old_tail.next = Some(new_tail);
                old_tail.next.as_deref_mut()
            }
            None => {
                self.head = Some(new_tail);
                self.head.as_deref_mut()
            }
        };
        self.tail = new_tail;
    }
}
```

```

$ cargo build

error[E0495]: cannot infer an appropriate lifetime for autoref due to conflicting
requirements
--> src/fifth.rs:35:27
|
35 |         self.head.as_deref_mut()
|             ^^^^^^^^^^^^^^

note: first, the lifetime cannot outlive the anonymous lifetime #1 defined on the
method body at 18:5...
--> src/fifth.rs:18:5
|
18 | /     pub fn push(&mut self, elem: T) {
19 | |         let new_tail = Box::new(Node {
20 | |             elem: elem,
21 | |             // When you push onto the tail, your next is always None
...
39 | |         self.tail = new_tail;
40 | |
| |-----^
note: ...so that reference does not outlive borrowed content
--> src/fifth.rs:35:17
|
35 |         self.head.as_deref_mut()
|             ^^^^^^^^^^

note: but, the lifetime must be valid for the lifetime 'a as defined on the impl at
13:6...
--> src/fifth.rs:13:6
|
13 | impl<'a, T> List<'a, T> {
|   ^
= note: ...so that the expression is assignable:
    expected std::option::Option<&'a mut fifth::Node<T>>
        found std::option::Option<&mut fifth::Node<T>>

```

好长... Rust 为啥这么难... 但是，这里有一句重点：

---

the lifetime must be valid for the lifetime 'a as defined on the impl

---

意思是说生命周期至少要和 'a 一样长，是不是因为编译器为 self 推导的生命周期不够长呢？我们试着来手动标注下：

```
pub fn push(&'a mut self, elem: T) {
```

当当当当，成功通过编译：

```
$ cargo build

warning: field is never used: `elem`
--> src/fifth.rs:9:5
9 |     elem: T,
|     ^^^^^^
|
|= note: #[warn(dead_code)] on by default
```

这个错误可以称之为错误之王，但是我们依然成功的解决了它，太棒了！再来实现下 `pop`：

```
pub fn pop(&'a mut self) -> Option<T> {
    self.head.take().map(|head| {
        let head = *head;
        self.head = head.next;

        if self.head.is_none() {
            self.tail = None;
        }

        head.elem
    })
}
```

看起来不错，写几个测试用例溜一溜：

```
mod test {
    use super::List;
#[test]
fn basics() {
    let mut list = List::new();

    // Check empty list behaves right
    assert_eq!(list.pop(), None);

    // Populate list
    list.push(1);
    list.push(2);
    list.push(3);

    // Check normal removal
    assert_eq!(list.pop(), Some(1));
    assert_eq!(list.pop(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push(4);
    list.push(5);

    // Check normal removal
    assert_eq!(list.pop(), Some(3));
    assert_eq!(list.pop(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop(), Some(5));
    assert_eq!(list.pop(), None);
}
}
```

```
$ cargo test

error[E0499]: cannot borrow `list` as mutable more than once at a time
--> src/fifth.rs:68:9
|
65 |         assert_eq!(list.pop(), None);
|             ---- first mutable borrow occurs here
...
68 |         list.push(1);
|             ^^^^
|             |
|             second mutable borrow occurs here
|             first borrow later used here

error[E0499]: cannot borrow `list` as mutable more than once at a time
--> src/fifth.rs:69:9
|
65 |         assert_eq!(list.pop(), None);
|             ---- first mutable borrow occurs here
...
69 |         list.push(2);
|             ^^^^
|             |
|             second mutable borrow occurs here
|             first borrow later used here

error[E0499]: cannot borrow `list` as mutable more than once at a time
--> src/fifth.rs:70:9
|
65 |         assert_eq!(list.pop(), None);
|             ---- first mutable borrow occurs here
...
70 |         list.push(3);
|             ^^^^
|             |
|             second mutable borrow occurs here
|             first borrow later used here

.....

** WAY MORE LINES OF ERRORS **

.....
error: aborting due to 11 previous errors
```

🐱 🐱 🐱，震惊！但编译器真的没错，因为都是我们刚才那个标记惹的祸。

我们为 `self` 标记了 '`a`'，意味着在 '`a`' 结束前，无法再去使用 `self`，大家可以自己推断下 '`a`' 的生命周期是多长。

那么该怎么办？回到老路 `RefCell` 上？显然不可能，那只能祭出大杀器：裸指针。

---

事实上，上文的问题主要是自引用引起的，感兴趣的同學可以查看[这里](#)深入阅读。

---

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>, // DANGER DANGER
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}
```

如上所示，当使用裸指针后，`head` 和 `tail` 就不会形成自引用的问题，也不再违反 Rust 严苛的借用规则。

---

注意！当前的实现依然是有严重问题的，在后面我们会修复

---

果然，编程的最高境界就是回归本质：使用 C 语言的东东。

# 基本操作

本章节的代码中有一个隐藏的 bug，因为它藏身于 unsafe 中，因此不会导致报错，我们会在后续章节解决这个问题，所以，请不要在生产环境使用此处的代码

在开始之前，大家需要先了解 unsafe 的[相关知识](#)。那么，言归正传，该如何构建一个链表？在之前我们是这么做的：

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }
}
```

但是我们不再在 tail 中使用 Option：

```
$ cargo build

error[E0308]: mismatched types
--> src/fifth.rs:15:34
|
15 |     List { head: None, tail: None }
|                         ^^^^ expected *-ptr, found
|                         enum `std::option::Option`
|
= note: expected type `*mut fifth::Node<T>`
         found type `std::option::Option<_>`
```

我们是可以使用 Option 包裹一层，但是 \*mut 裸指针之所以裸，是因为它狂，它可以是 null！因此 Option 就变得没有意义：

```
use std::ptr;

// defns...

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: ptr::null_mut() }
    }
}
```

如上所示，通过 std::ptr::null\_mut 函数可以获取一个 null，当然，还可以使用 0 as \*mut \_，但是...已经这么不安全了，好歹我们要留一点代码可读性上的尊严吧 =, =

好了，现在是时候去重新实现 `push`，之前获取的是 `Option<&mut Node<T>>` 成为我们的拦路虎，这次来看看如果是获取 `*mut Node<T>` 还会不会有类似的问题。

首先，该如何将一个普通的引用变成裸指针？答案是：强制转换 Coercions。

```
let raw_tail: *mut _ = &mut *new_tail;
```

来看看 `push` 的实现：

```
pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(Node {
        elem: elem,
        next: None,
    });

    let raw_tail: *mut _ = &mut *new_tail;

    // .is_null 会检查是否为 null，在功能上等价于 `None` 的检查
    if !self.tail.is_null() {
        // 如果 old tail 存在，那将其指向新的 tail
        self.tail.next = Some(new_tail);
    } else {
        // 否则让 head 指向新的 tail
        self.head = Some(new_tail);
    }

    self.tail = raw_tail;
}

$ cargo build

error[E0609]: no field `next` on type `*mut fifth::Node<T>`
--> src/fifth.rs:31:23
|
31 |         self.tail.next = Some(new_tail);
|             -----^^^^
|
|             help: `self.tail` is a raw pointer;
|             try dereferencing it: `(*self.tail).next`
```

当使用裸指针时，一些 Rust 提供的便利条件也将不复存在，例如由于不安全性的存在，裸指针需要我们手动去解引用(deref)：

```
*self.tail.next = Some(new_tail);
```

```
$ cargo build

error[E0609]: no field `next` on type `*mut fifth::Node<T>`
 --> src/fifth.rs:31:23
31 |         *self.tail.next = Some(new_tail);
|         -----^^^^^
|         |
|         help: `self.tail` is a raw pointer;
|         try dereferencing it: `(*self.tail).next`
```

哦哦，运算符的优先级问题：

```
(*self.tail).next = Some(new_tail);

$ cargo build

error[E0133]: dereference of raw pointer is unsafe and requires
              unsafe function or block
--> src/fifth.rs:31:13
31 |         (*self.tail).next = Some(new_tail);
|         ^^^^^^^^^^^^^^ dereference of raw pointer
|
= note: raw pointers may be NULL, dangling or unaligned;
      they can violate aliasing rules and cause data races:
      all of these are undefined behavior
```

哎...太难了，错误一个连一个，好在编译器给出了提示：由于我们在进行不安全的操作，因此需要使用 `unsafe` 语句块。那么问题来了，是将某几行代码包在 `unsafe` 中还是将整个函数包在 `unsafe` 中呢？如果大家不知道哪个是正确答案的话，证明[之前的章节](#)还是没有仔细学，请回去再看一下，巩固巩固：)

```

pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(Node {
        elem: elem,
        next: None,
    });

    let raw_tail: *mut _ = &mut *new_tail;

    if !self.tail.is_null() {
        // 你好编译器，我知道我在做危险的事情，我向你保证：就算犯错了，也和你没有关系，都是我这个不
优秀的程序员的责任
        unsafe {
            (*self.tail).next = Some(new_tail);
        }
    } else {
        self.head = Some(new_tail);
    }

    self.tail = raw_tail;
}

$ cargo build
warning: field is never used: `elem`
--> src/fifth.rs:11:5
|
11 |     elem: T,
|     ^^^^^^
|
= note: #[warn(dead_code)] on by default

```

细心的同学可能会发现：不是所有的裸指针代码都有 unsafe 的身影。原因在于：**创建原生指针是安全的行为，而解引用原生指针才是不安全的行为**

呼，长出了一口气，终于成功实现了 push，下面来看看 pop：

```

pub fn pop(&mut self) -> Option<T> {
    self.head.take().map(|head| {
        let head = *head;
        self.head = head.next;

        if self.head.is_none() {
            self.tail = ptr::null_mut();
        }

        head.elem
    })
}

```

测试下：

```

#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), None);

        // Check the exhaustion case fixed the pointer right
        list.push(6);
        list.push(7);

        // Check normal removal
        assert_eq!(list.pop(), Some(6));
        assert_eq!(list.pop(), Some(7));
        assert_eq!(list.pop(), None);
    }
}

```

摊牌了，我们偷懒了，这些测试就是从之前的栈链表赋值过来的，但是依然做了些改变，例如在末尾增加了几个步骤以确保在 `pop` 中不会发生尾指针损坏( tail-pointer corruption )的情况。

```
$ cargo test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

# Miri

看到这里，大家是不是暗中松了口气？unsafe 不过如此嘛，不知道为何其它人都谈之色变。

怎么说呢？你以为的编译器已经不是以前的编译器了，它不报错不代表没有错误。包括测试用例也是，正常地运行不能意味着代码没有任何错误。

在周星驰电影功夫中，还有一个奇怪大叔 10 元一本主动上门卖如来神掌，那么有没有 10 元一本的 Rust 秘笈呢？（喂，Rust 语言圣经都免费让你读了，有了摩托车，还要什么拖拉机... 哈哈，开个玩笑）

有的，奇怪大叔正在赶来，他告诉我们先来安装一个命令：

```
rustup +nightly-2022-01-21 component add miri
info: syncing channel updates for 'nightly-2022-01-21-x86_64-pc-windows-msvc'
info: latest update on 2022-01-21, rust version 1.60.0-nightly (777bb86bc 2022-01-20)
info: downloading component 'cargo'
info: downloading component 'clippy'
info: downloading component 'rust-docs'
info: downloading component 'rust-std'
info: downloading component 'rustc'
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
info: installing component 'rust-std'
info: installing component 'rustc'
info: installing component 'rustfmt'
info: downloading component 'miri'
info: installing component 'miri'
```

等等，你在我电脑上装了什么奇怪的东西？！ "好东西"

---

miri 目前只能在 nightly Rust 上安装，+nightly-2022-01-21 告诉 rustup 我们想要安装的 nightly 版本，事实上，你可以直接通过 rustup +nightly component add miri 安装，这里指定版本主要因为 miri 有时候会因为某些版本而出错。

2022-01-21 是我所知的 miri 可以成功运行的版本，你可以检查[这个网址](#)获取更多信息

- 是一种临时性的规则运用，如果你不想每次都使用 +nightly-2022-01-21，可以使用 `rustup override set` 命令对当前项目的 Rust 版本进行覆盖
-

```
$ cargo +nightly-2022-01-21 miri test  
I will run `"/cargo.exe" "install" "xargo"` to install  
a recent enough xargo. Proceed? [Y/n]
```

额， xargo 是什么东东？ "不要担心，选择 y 就行，我像是会坑你的人吗？ "

```
> y  
  
Updating crates.io index  
Installing xargo v0.3.24  
...  
    Finished release [optimized] target(s) in 10.65s  
Installing C:\Users\ninte\.cargo\bin\xargo-check.exe  
Installing C:\Users\ninte\.cargo\bin\xargo.exe  
    Installed package `xargo v0.3.24` (executables `xargo-check.exe`, `xargo.exe`)  
  
I will run `"/rustup" "component" "add" "rust-src"` to install  
the `rust-src` component for the selected toolchain. Proceed? [Y/n]
```

额？ "不要怕，多给你一份 Rust 源代码，不开心嘛？ "

```
> y  
  
info: downloading component 'rust-src'  
info: installing component 'rust-src'
```

"看吧，我就说我不骗你的，不相信我，等着错误砸脸吧！" 真是一个奇怪的大叔...

```
Compiling lists v0.1.0 (C:\Users\ninte\dev\tmp\lists)
Finished test [unoptimized + debuginfo] target(s) in 0.25s
Running unitests (lists-5cc11d9ee5c3e924.exe)

error: Undefined Behavior: trying to reborrow for Unique at alloc84055,
but parent tag <209678> does not have an appropriate item in
the borrow stack

--> \lib\rustlib\src\rust\library\core\src\option.rs:846:18
846 |     Some(x) => Some(f(x)),
|         ^ trying to reborrow for Unique at alloc84055,
|             but parent tag <209678> does not have an
|                 appropriate item in the borrow stack
|
|= help: this indicates a potential bug in the program:
it performed an invalid operation, but the rules it
violated are still experimental
|= help: see https://github.com/rust-lang/unsafe-code-guidelines/blob/master/wip/stacked-borrows.md
for further information

= note: inside `std::option::Option::<std::boxed::Box<fifth::Node<i32>>>::map::
<i32, [closure@src\fifth.rs:31:30: 40:10]>` at
\lib\rustlib\src\rust\library\core\src\option.rs:846:18

note: inside `fifth::List::<i32>::pop` at src\fifth.rs:31:9
--> src\fifth.rs:31:9
31 |     self.head.take().map(|head| {
32 |         let head = *head;
33 |         self.head = head.next;
34 |
...
39 |         head.elem
40 |     })
|-----^
note: inside `fifth::test::basics` at src\fifth.rs:74:20
--> src\fifth.rs:74:20
|
74 |     assert_eq!(list.pop(), Some(1));
|           ^^^^^^^^^^^^
note: inside closure at src\fifth.rs:62:5
--> src\fifth.rs:62:5
|
61 |     #[test]
|----- in this procedural macro expansion
62 |     fn basics() {
63 |         let mut list = List::new();
64 |
65 |         // Check empty list behaves right
...
|
```

```
96 | |         assert_eq!(list.pop(), None);
97 | |     }
| |____^
...
error: aborting due to previous error
```

咦还真有错误，大叔，这是什么错误？大叔？...奇怪的大叔默默离开了，留下我在风中凌乱。

果然不靠谱...还是得靠自己，首先得了解下何为 `miri`。

`miri` 可以生成 Rust 的中间层表示 MIR，对于编译器来说，我们的 Rust 代码首先会被编译为 MIR，然后再提交给 LLVM 进行处理。

可以通过 `rustup component add miri` 来安装它，并通过 `cargo miri` 来使用，同时还可以使用 `cargo miri test` 来运行测试代码。

`miri` 可以帮助我们检查常见的未定义行为(UB = Undefined Behavior)，以下列出了一部分：

- 内存越界检查和内存释放后再使用(use-after-free)
- 使用未初始化的数据
- 数据竞争
- 内存对齐问题

UB 检测是必须的，因为它发生在运行时，因此很难发现，如果 `miri` 能在编译期检测出来，那自然是最好不过的。

总之，`miri` 的使用很简单：

```
$ cargo +nightly-2022-01-21 miri test
```

下面来看看具体的错误：

```
error: Undefined Behavior: trying to reborrow for Unique at alloc84055, but parent
tag <209678> does not have an appropriate item in the borrow stack

--> \lib\rustlib\src\rust\library\core\src\option.rs:846:18
846 |         Some(x) => Some(f(x)),
|             ^ trying to reborrow for Unique at alloc84055,
|             but parent tag <209678> does not have an
|             appropriate item in the borrow stack
|
= help: this indicates a potential bug in the program: it
performed an invalid operation, but the rules it
violated are still experimental

= help: see
https://github.com/rust-lang/unsafe-code-guidelines/blob/master/wip/stacked-borrows.md
for further information
```

嗯，只能看出是一个错误，其它完全看不懂了，例如什么是 borrow stack ?

# 栈借用( Stacked Borrow )

上一章节中我们运行 miri 时遇到了一个栈借用错误，还给了文档链接，但这些文档主要是给编译器开发者和 Rust 研究者看的，因此就不进行讲解了。

而这里，我们将从一个更高层次的角度来看看何为栈借用。

---

目前栈借用在 Rust 语义模型中还是试验阶段，因此破坏这些规则不一定说明你的程序错了。但是除非你在做编译器开发，否则最好还是修复这些错误。事前的麻烦总比事后的不安全要好，特别是当涉及到 UB 未定义行为时

---

## 指针混叠( Pointer Aliasing )

在开始了解我们破坏的规则之前，首先应该了解为何会有这些规则的存在。这里有多个动机，但是我认为最重要的动机是：指针混叠。

当两个指针指向的内存区域存在重叠时，就说这两个指针发生了混叠，这种情况会造成一些问题。例如，编译器使用指针混叠的信息来优化内存的访问，当这些信息出错时，那程序就会被不正确地编译，然后产生一些奇怪的结果。

---

实际上，混叠更多关心的是内存访问而不是指针本身，而且只有在其中一个访问是可变的时，才可能出问题。之所以说指针，是因为指针这个概念更方便跟一些规则进行关联。

---

再比如，编译器需要获取一个值时，是该去缓存中查询还是每次都去内存中加载呢？关于这个选择，编译器需要清晰地知道是否有一个指针在背后修改内存，如果内存值被修改了，那缓存显然就失效了。

## 安全地栈借用

有了之前的铺垫，大家肯定希望编译器能对指针混叠的信息了若指掌，但是可以吗？对于 Rust 正常代码而言，这种情况是可以避免的，因为严格的借用规则是我们的后盾：要么同时存在一个可变引用，要么同时存在多个不可变引用，这种规则简直完美避免了：两个指针指向同一块儿重叠内存区域，而其中一个是可变指针。

然而实际使用中，有一些情况会较为复杂，例如以下代码中发生了可变引用的再借用( reborrow )：

```

let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

*ref2 += 2;
*ref1 += 1;

println!("{}", data);

```

看上去像是违反了借用规则，但是这段代码确实可以正常编译运行，如果交换下引用使用的顺序呢？

```

let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

// ORDER SWAPPED!
*ref1 += 1;
*ref2 += 2;

println!("{}", data);

error[E0503]: cannot use `*ref1` because it was mutably borrowed
--> src/main.rs:6:5
|
4 |     let ref2 = &mut *ref1;
|           ----- borrow of `*ref1` occurs here
5 |
6 |     *ref1 += 1;
|     ^^^^^^^^^^ use of borrowed `*ref1`
7 |     *ref2 += 2;
|           ----- borrow later used here

For more information about this error, try `rustc --explain E0503`.
error: could not compile `playground` due to previous error

```

果不其然，编译器抛出了错误，当我们再借用了一个可变引用时，那原始的引用就不能再被使用，直到借用者完成了任务：借用者的借用有效范围并不是看作用域，而是看最后一次使用的位置，正因为如此，第一段代码可以编译通过，而第二段不行，这是著名的生命周期 [NLL 规则](#)。

以上就是我们拥有再借用但是还拥有混叠信息的原因：所有的再借用都在清晰地进行嵌套，因此每个再借用都不会与其它的冲突。那大家知道什么方法可以很好的展现嵌套的事物吗？答案就是使用栈来存放这些嵌套的借用。

嘿，这不就是栈借用吗？

这个栈的顶部借用就是当前正在使用( live )的借用，而它清晰的知道在它使用的期间不会发生混叠。当对一个指针进行再借用时，新的借用会被插入到栈的顶部，并变成 live 状态。如果要将一个旧的指针变成

live，就需要将借用栈上在它之前的借用全部弹出( pop )。

通过栈借用的方式，我们保证了尽管存在多个再借用，但是在同一个时间，只会有一个可变引用访问目标内存，再也不用担心指针混叠的问题了。只要不去访问一个已经被弹出借用栈的指针，就会非常安全！

从表述方式来说，与其说使用 `ref1` 会让 `ref2` 不合法，不如说 `ref2` 必须要在所有使用情况下合法，`ref1` 恰恰是其中一种情况，会破坏 `ref2` 的合法性。而编译器的报错也是选择了第二种表述方式：无法使用 `*ref1`，原因是它已经被可变借用了，可以看出，第二种表述方式比第一种要更加符合直觉。

**但是，当使用 unsafe 指针时，借用检查器就无法再帮助我们了！**

## 不安全地栈借用

所以，我们现在需要一个方式让 `unsafe` 指针也可以参与到栈借用系统中来，即使编译器无法正确地跟踪它们。同时我们也希望这个系统能宽松一些，不要很容易就产生 UB。

这是一个困难的问题，我也不知道该如何解决，但是目前在编写栈借用系统的开发者显然是有想法的，例如 `miri` 就是其中一个产物。

从一个高抽象层次来看，当我们转换成裸指针时，就是一种再借用。那么随后，裸指针就可以对目标内存进行操作，当再借用结束时，发生的事情跟正常的再借用结束也没有区别。

但是问题是，你还可以将一个裸指针转变成引用，最重要的是，还可以对裸指针进行拷贝！如果发生了以下转换 `&mut -> *mut -> &mut -> *mut`，然后去访问第一个 `*mut`，这种见鬼的情况下，栈借用该如何发挥作用？

反正我不知道，只能求助于 `miri` 了。事实上，正因为这种情况，`miri` 还提供了试验性的模式：`-Zmiri-tag-raw-pointers`。可以通过环境的方式来开启该模式：

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test
```

如果是 Windows，你需要设置全局变量：

```
$env:MIRIFLAGS="-Zmiri-tag-raw-pointers"  
cargo +nightly-2022-01-21 miri test
```

## 管理栈借用

因为之前的问题，使用裸指针，应该遵守一个原则：**一旦开始使用裸指针，就要尝试着只使用它。**

现在，我们依然希望在接口中使用安全的引用去构建一个安全的抽象，例如在函数参数中使用引用而不是裸指针，这样我们的用户就无需操心 unsafe 的问题。

为此，我们需要做以下事情：

1. 在开始时，将输入参数中的引用转换成裸指针
2. 在函数体中只使用裸指针
3. 返回之前，将裸指针转换成安全的指针

但是由于数据结构中的字段都是私有的，无需暴露给用户，因此无需这么麻烦，直接使用裸指针即可。

事实上，一个依然存在的问题就是还在继续使用 Box，它会告诉编译器：hey，这个看上去很像是 `&mut`，因为它唯一的持有那个指针。

但是我们在链表中一直使用的裸指针是指向 Box 的内部，所以无论何时我们通过正常的方式访问 Box，我们都可能让该裸指针的再借用变得不合法。

# 测试栈借用

---

关于上一章节的简单总结:

- Rust 通过借用栈来处理再借用
  - 只有栈顶的元素是处于 `live` 状态的( 被借用 )
  - 当访问栈顶下面的元素时, 该元素会变为 `live`, 而栈顶元素会被弹出( `pop` )
  - 从借用栈中弹出的元素无法再被借用
  - 借用检查器会保证我们的安全代码遵守以上规则
  - Miri 可以在一定程度上保证裸指针在运行时也遵循以上规则
- 

作为作者同时也是读者, 我想说上一章节的内容相当不好理解, 下面来看一些例子, 通过它们可以帮助大家更好的理解栈借用模型。

在实际项目中捕获 UB 是一件相当不容易的事, 毕竟你是在编译器的盲区之外摸索和行动。

如果我们足够幸运的话, 写出来的代码是可以"正常运行的", 但是一旦编译器聪明一点或者你修改了某处代码, 那这些代码可能会立刻化身为一颗安静的定时炸弹。当然, 如果你还是足够幸运, 那程序会发生崩溃, 你也就可以捕获和处理相应的错误。但是如果你不幸运呢?

那代码就算出问题了, 也只是会发生一些奇怪的现象, 面对这些现象你将束手无策, 甚至不知道该如何处理!

Miri 为何可以一定程度上提前发现这些 UB 问题? 因为它会去获取 `rustc` 对我们的程序最原生、且没有任何优化的视角, 然后对看到的内容进行解释和跟踪。只要这个过程能够开始, 那这个解决方法就相当有效, 但是问题来了, 该如何让这个过程开始? 要知道 Miri 和 `rustc` 是不可能去逐行分析代码中的所有行为的, 这样做的结果就是编译时间大大增加!

因此我们需要使用测试用例来让程序中可能包含 UB 的代码路径被真正执行到, 当然, 就算你这么做了, 也不能完全依赖 Miri。既然是分析, 就有可能遗漏, 也可能误杀友军。

## 基本借用

在上一章节中, 借用检查器似乎不喜欢以下代码:

```
let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

*ref1 += 1;
*ref2 += 2;

println!("{}", data);
```

它违背了再借用的原则，大家可以用借用栈的分析方式去验证下上一章节所学的知识。

下面来看看，如果使用裸指针会怎么样：

```
unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;

    *ref1 += 1;
    *ptr2 += 2;

    println!("{}", data);
}

$ cargo run

Compiling miri-sandbox v0.1.0
Finished dev [unoptimized + debuginfo] target(s) in 0.71s
Running `target\debug\miri-sandbox.exe`
```

13

嗯，编译器看起来很满意：不仅获取了预期的结果，还没有任何警告。那么再来征求下 Miri 的意见：

```

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running cargo-miri.exe target\miri

error: Undefined Behavior: no item granting read access
to tag <untagged> at alloc748 found in borrow stack.

--> src\main.rs:9:9
9 |     *ptr2 += 2;
|     ^^^^^^^^^^ no item granting read access to tag <untagged>
|                 at alloc748 found in borrow stack.
|
= help: this indicates a potential bug in the program:
  it performed an invalid operation, but the rules it
  violated are still experimental

```

喔，果然出问题了。下面再来试试更复杂的 `&mut -> *mut -> &mut -> *mut` :

```

unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;
    let ref3 = &mut *ptr2;
    let ptr4 = ref3 as *mut _;

    // 首先访问第一个裸指针
    *ptr2 += 2;

    // 接着按照借用栈的顺序来访问
    *ptr4 += 4;
    *ref3 += 3;
    *ptr2 += 2;
    *ref1 += 1;

    println!("{}", data);
}

```

`$ cargo run`

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
to tag <1621> at alloc748 found in borrow stack.

--> src\main.rs:13:5
|
13 |     *ptr4 += 4;
|     ^^^^^^^^^^ no item granting read access to tag <1621>
|             at alloc748 found in borrow stack.
|
```

不错，可以看出 miri 有能力分辨两个裸指针的使用限制：当使用第二个时，需要先让之前的失效。

再来移除乱入的那一行，让借用栈可以真正顺利的工作：

```
unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;
    let ref3 = &mut *ptr2;
    let ptr4 = ref3 as *mut _;

    // Access things in "borrow stack" order
    *ptr4 += 4;
    *ref3 += 3;
    *ptr2 += 2;
    *ref1 += 1;

    println!("{}", data);
}

$ cargo run
20

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
20
```

我现在可以负责任的说：在座的各位，都是...可以获取编程语言内存模型设计博士学位的存在，编译器？那是什么东东！简单的很。

---

旁白：那个..关于博士的一切，请不要当真，但是我依然为你们骄傲

---

## 测试数组

下面来干一票大的：使用指针偏移来搞乱一个数组。

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // 获取第 1 个元素的引用
    let ptr2_at_0 = ref1_at_0 as *mut i32; // 裸指针 ptr 指向第 1 个元素
    let ptr3_at_1 = ptr2_at_0.add(1);      // 对裸指针进行运算，指向第 2 个元素

    *ptr3_at_1 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [3, 3, 0, ...]
    println!("{}:?", &data[..]);
}
```

```
$ cargo run
[3, 3, 0, 0, 0, 0, 0, 0, 0]
```

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
to tag <1619> at alloc748+0x4 found in borrow stack.
--> src\main.rs:8:5
|
8 |     *ptr3_at_1 += 3;
|     ^^^^^^^^^^^^^^ no item granting read access to tag <1619>
|                     at alloc748+0x4 found in borrow stack.
```

咦？我们命名按照借用栈的方式来完美使用了，为何 miri 还是提示了 UB 风险？难道是因为 `ptr -> ptr` 的过程中发生了什么奇怪的事情？如果我们只是拷贝指针，让它们都指向同一个位置呢？

```

unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];
    let ptr2_at_0 = ref1_at_0 as *mut i32;
    let ptr3_at_0 = ptr2_at_0;

    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [6, 0, 0, ...]
    println!("{:?}", &data[..]);
}

$ cargo run
[6, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[6, 0, 0, 0, 0, 0, 0, 0, 0]

```

果然，顺利通过，下面我们还是让它们指向同一个位置，但是来首名为混乱的 BGM:

```

unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // Reference to 0th element
    let ptr2_at_0 = ref1_at_0 as *mut i32;   // Ptr to 0th element
    let ptr3_at_0 = ptr2_at_0;              // Ptr to 0th element
    let ptr4_at_0 = ptr2_at_0.add(0);        // Ptr to 0th element
    let ptr5_at_0 = ptr3_at_0.add(1).sub(1); // Ptr to 0th element

    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ptr4_at_0 += 4;
    *ptr5_at_0 += 5;
    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [20, 0, 0, ...]
    println!("{:?}", &data[..]);
}

$ cargo run
[20, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[20, 0, 0, 0, 0, 0, 0, 0, 0]

```

可以看出，miri 对于这种裸指针派生是相当纵容的：当它们都共享同一个借用时(borrowing, 也可以用 miri 的称呼：tag)。

---

当代码足够简单时，编译器是有可能介入跟踪所有派生的裸指针，并尽可能去优化它们的。但是这套规则比引用的那套脆弱得多！

---

那么问题来了：真正的问题到底是什么？

对于部分数据结构，Rust 允许对其中的字段进行独立借用，例如一个结构体，它的多个字段可以被分开借用，来试试这里的数组可不可以。

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // Reference to 0th element
    let ref2_at_1 = &mut data[1];           // Reference to 1th element
    let ptr3_at_0 = ref1_at_0 as *mut i32; // Ptr to 0th element
    let ptr4_at_1 = ref2_at_1 as *mut i32; // Ptr to 1th element

    *ptr4_at_1 += 4;
    *ptr3_at_0 += 3;
    *ref2_at_1 += 2;
    *ref1_at_0 += 1;

    // Should be [3, 3, 0, ...]
    println!("{:?}", &data[..]);
}

error[E0499]: cannot borrow `data[_]` as mutable more than once at a time
--> src\main.rs:5:21
|
4 |     let ref1_at_0 = &mut data[0];           // Reference to 0th element
|           ----- first mutable borrow occurs here
5 |     let ref2_at_1 = &mut data[1];           // Reference to 1th element
|           ^^^^^^^^^^^^^ second mutable borrow occurs here
6 |     let ptr3_at_0 = ref1_at_0 as *mut i32; // Ptr to 0th element
|           ----- first borrow later used here
|
= help: consider using `split_at_mut(position)` or similar method
      to obtain two mutable non-overlapping sub-slices
```

显然..不行，Rust 不允许我们对数组的不同元素进行单独的借用，注意到提示了吗？可以使用 `.split_at_mut(position)` 来将一个数组分成多个部分：

```
unsafe {
    let mut data = [0; 10];

    let slice1 = &mut data[..];
    let (slice2_at_0, slice3_at_1) = slice1.split_at_mut(1);

    let ref4_at_0 = &mut slice2_at_0[0];      // Reference to 0th element
    let ref5_at_1 = &mut slice3_at_1[0];      // Reference to 1th element
    let ptr6_at_0 = ref4_at_0 as *mut i32;    // Ptr to 0th element
    let ptr7_at_1 = ref5_at_1 as *mut i32;    // Ptr to 1th element

    *ptr7_at_1 += 7;
    *ptr6_at_0 += 6;
    *ref5_at_1 += 5;
    *ref4_at_0 += 4;

    // Should be [10, 12, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
$ cargo run
[10, 12, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[10, 12, 0, 0, 0, 0, 0, 0, 0, 0]
```

将数组切分成两个部分后，代码就成功了，如果我们将一个切片转换成指针呢？那指针是否还拥有访问整个切片的权限？

```

unsafe {
    let mut data = [0; 10];

    let slice1_all = &mut data[..];           // Slice for the entire array
    let ptr2_all = slice1_all.as_mut_ptr(); // Pointer for the entire array

    let ptr3_at_0 = ptr2_all;                // Pointer to 0th elem (the same)
    let ptr4_at_1 = ptr2_all.add(1);         // Pointer to 1th elem
    let ref5_at_0 = &mut *ptr3_at_0;        // Reference to 0th elem
    let ref6_at_1 = &mut *ptr4_at_1;        // Reference to 1th elem

    *ref6_at_1 += 6;
    *ref5_at_0 += 5;
    *ptr4_at_1 += 4;
    *ptr3_at_0 += 3;

    // 在循环中修改所有元素( 仅仅为了有趣 )
    // (可以使用任何裸指针, 它们共享同一个借用!)
    for idx in 0..10 {
        *ptr2_all.add(idx) += idx;
    }

    // 同样为了有趣, 再实现下安全版本的循环
    for (idx, elem_ref) in slice1_all.iter_mut().enumerate() {
        *elem_ref += idx;
    }

    // Should be [8, 12, 4, 6, 8, 10, 12, 14, 16, 18]
    println!("{:?}", &data[..]);
}

$ cargo run
[8, 12, 4, 6, 8, 10, 12, 14, 16, 18]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[8, 12, 4, 6, 8, 10, 12, 14, 16, 18]

```

## 测试不可变引用

在之前的例子中，我们使用的都是可变引用，而 Rust 中还有不可变引用。那么它将如何工作呢？

我们已经见过裸指针可以被简单的拷贝只要它们共享同一个借用，那不可变引用是不是也可以这么做？

注意，下面的 `println!` 会自动对待打印的目标值进行 `ref/deref` 等操作，因此为了保证测试的正确性，我们将其放入一个函数中。

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let sref2 = &mref1;
    let sref3 = sref2;
    let sref4 = &*sref2;

    // Random hash of shared reference reads
    opaque_read(sref3);
    opaque_read(sref2);
    opaque_read(sref4);
    opaque_read(sref2);
    opaque_read(sref3);

    *mref1 += 1;

    opaque_read(&data);
}
```

```
$ cargo run
```

```
warning: unnecessary `unsafe` block
--> src\main.rs:6:1
|
6 | unsafe {
| ^^^^^^ unnecessary `unsafe` block
|
= note: `#[warn(unused_unsafe)]` on by default

warning: `miri-sandbox` (bin "miri-sandbox") generated 1 warning

10
10
10
10
10
11
```

虽然这里没有使用裸指针，但是可以看到对于不可变引用而言，上面的使用方式不存在任何问题。下面来增加一些裸指针：

```

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*mref1;
    let ptr4 = sref3 as *mut i32;

    *ptr4 += 4;
    opaque_read(sref3);
    *ptr2 += 2;
    *mref1 += 1;

    opaque_read(&data);
}

```

\$ cargo run

```

error[E0606]: casting `&i32` as `*mut i32` is invalid
--> src/main.rs:11:20
|
11 |         let ptr4 = sref3 as *mut i32;
      |         ^^^^^^^^^^
      |         ^^^^^^^^^^

```

可以看出，我们无法将一个不可变的引用转换成可变的裸指针，只能曲线救国了：

```
let ptr4 = sref3 as *const i32 as *mut i32;
```

如上，先将不可变引用转换成不可变的裸指针，然后再转换成可变的裸指针。

```
$ cargo run
```

```
14
17
```

编译器又一次满意了，再来看看 miri：

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting write access to
tag <1621> at alloc742 found in borrow stack.
--> src\main.rs:13:5
|
13 |     *ptr4 += 4;
|     ^^^^^^^^^^^^ no item granting write access to tag <1621>
|                 at alloc742 found in borrow stack.
```

果然，miri 提示了，原因是当我们使用不可变引用时，就相当于承诺不会去修改其中的值，那 miri 发现了这种修改行为，自然会给予相应的提示。

对此，可以用一句话来简单总结：**在借用栈中，一个不可变引用，它上面的所有引用(在它之后被推入借用栈的引用)都只能拥有只读的权限。**

但是我们可以这样做：

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*mref1;
    let ptr4 = sref3 as *const i32 as *mut i32;

    opaque_read(&*ptr4);
    opaque_read(sref3);
    *ptr2 += 2;
    *mref1 += 1;

    opaque_read(&data);
}
```

可以看到，我们其实可以创建一个可变的裸指针，只要不去使用写操作，而是只使用读操作。

```
$ cargo run

10
10
13

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
10
10
13
```

再来检查下不可变的引用是否可以像平时一样正常弹出:

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*mref1;

    *ptr2 += 2;
    opaque_read(sref3); // Read in the wrong order?
    *mref1 += 1;

    opaque_read(&data);
}
```

```
$ cargo run

12
13

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: trying to reborrow for SharedReadonly
at alloc742, but parent tag <1620> does not have an appropriate
item in the borrow stack

--> src\main.rs:13:17
|
13 |     opaque_read(sref3); // Read in the wrong order?
|           ^^^^^^ trying to reborrow for SharedReadonly
|           at alloc742, but parent tag <1620>
|           does not have an appropriate item
|           in the borrow stack
|
```

细心的同学可能会发现，我们这次获得了一个相当具体的 miri 提示，而不是之前的某个 tag。真是令人感动...毕竟这种错误信息会更有帮助。

## 测试内部可变性

还记得之前我们试图用 `RefCell + Rc` 去实现的那个糟糕的链表吗？这两个组合在一起就可以实现内部可变性。与 `RefCell` 类似的还有 `Cell`：

```
use std::cell::Cell;

unsafe {
    let mut data = Cell::new(10);
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut Cell<i32>;
    let sref3 = &*mref1;

    sref3.set(sref3.get() + 3);
    (*ptr2).set((*ptr2).get() + 2);
    mref1.set(mref1.get() + 1);

    println!("{}", data.get());
}
```

地狱一般的代码，就等着 miri 来优化你吧。

```
$ cargo run
16
MIRIFLAGS="-Zmiri-tag=raw-pointers" cargo +nightly-2022-01-21 miri run
16
```

等等，竟然没有任何问题，我们需要深入调查下原因：

```
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}
```

以上是标准库中的 `Cell` 源码，可以看到里面有一个 `UnsafeCell`，通过名字都能猜到，这个数据结构相当的不安全，在[标准库](#)中有以下描述：

---

Rust 中用于内部可变性的核心原语( primitive )。

如果你拥有一个引用 `&T`，那一般情况下，Rust编译器会基于 `&T` 指向不可变的数据这一事实来进行相关的优化。通过别名或者将 `&T` 强制转换成 `&mut T` 是一种 UB 行为。

而 `UnsafeCell<T>` 移除了 `&T` 的不可变保证：一个不可变引用 `&UnsafeCell<T>` 指向一个可以改变的数据。, 这就是内部可变性。

---

感觉像是魔法，那下面就用该魔法让 miri happy 下：

```
use std::cell::UnsafeCell;

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = UnsafeCell::new(10);
    let mref1 = &mut data;                      // Mutable ref to the *outside*
    let ptr2 = mref1.get();                     // Get a raw pointer to the insides
    let sref3 = &*mref1;                        // Get a shared ref to the *outside*

    *ptr2 += 2;                                // Mutate with the raw pointer
    opaque_read(&sref3.get());                 // Read from the shared ref
    *sref3.get() += 3;                          // Write through the shared ref
    *mref1.get() += 1;                          // Mutate with the mutable ref

    println!("{}", *data.get());
}

$ cargo run

12
16

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
12
16
```

这段代码非常成功！但是等等..这里的代码顺序有问题：我们首先获取了内部的裸指针 `ptr2`，然后获取了一个不可变引用 `sref3`，接着我们使用了裸指针，然后是 `sref3`，这不就是标准的借用栈错误典范吗？既然如此，为何 miri 没有给出提示？

现在有两个解释：

- Miri 并不完美，它依然会有所遗漏，也会误判
- 我们的简化模型貌似过于简化了

大家选择哪个？..我不管，反正我选择第二个。不过，虽然我们的借用栈过于简单，但是依然是亲孩子嘛，最后再基于它来实现一个真正正确的版本：

```
use std::cell::UnsafeCell;

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = UnsafeCell::new(10);
    let mref1 = &mut data;
    // These two are swapped so the borrows are *definitely* totally stacked
    let sref2 = &*mref1;
    // Derive the ptr from the shared ref to be super safe!
    let ptr3 = sref2.get();

    *ptr3 += 3;
    opaque_read(&*sref2.get());
    *sref2.get() += 2;
    *mref1.get() += 1;

    println!("{}", *data.get());
}

$ cargo run
13
16

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
13
16
```

## 测试 Box

大家还记得为何我们讲了这么长的两章借用栈吗？原因就在于 Box 和裸指针混合使用时出了问题。

Box 在某种程度上类似 `&mut`，因为对于它指向的内存区域，它拥有唯一的所有权。

```

unsafe {
    let mut data = Box::new(10);
    let ptr1 = (&mut *data) as *mut i32;

    *data += 10;
    *ptr1 += 1;

    // Should be 21
    println!("{}", data);
}

$ cargo run
21

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
      to tag <1707> at alloc763 found in borrow stack.

--> src\main.rs:7:5
|
7 |     *ptr1 += 1;
|     ^^^^^^^^^^ no item granting read access to tag <1707>
|                 at alloc763 found in borrow stack.
|

```

现在到现在为止，大家一眼就能看出来这种代码不符合借用栈的规则。当然，miri 也讨厌这一点，因此我们来改正下。

```

unsafe {
    let mut data = Box::new(10);
    let ptr1 = (&mut *data) as *mut i32;

    *ptr1 += 1;
    *data += 10;

    // Should be 21
    println!("{}", data);
}

$ cargo run
21

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
21

```

在经过这么长的旅程后，我们终于完成了借用栈的学习，兄弟们我已经累趴了，你们呢？

但是，话说回来，该如何使用 Box 来解决栈借用的问题？当然，我们可以像之前的测试例子一样写一些玩具代码，但是在实际链表中中，将 Box 存储在某个地方，然后长时间持有一个裸指针才是经常遇到的。

等等，你说链表？天呐，我都忘记了我们还在学习链表，那接下来，继续实现之前未完成的链表吧。

# 数据布局2：再裸一些吧

TL;DR 在之前部分中，将安全的指针 `&`、`&mut` 和 `Box` 跟不安全的裸指针 `*mut` 和 `*const` 混用是 UB 的根源之一，原因是安全指针会引入额外的约束，但是裸指针并不会遵守这些约束。

一个好消息，一个坏消息。坏消息是我们又要开始写链表了，悲剧 =，= 好消息呢是之前我们已经讨论过该如何设计了，之前做的工作基本都是正确的，除了混用安全指针和不安全指针的部分。

## 布局

在新的布局中我们将只使用裸指针，然后大家就等着好消息吧！

下面是之前的"破代码"：

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>, // 好人一枚
}

type Link<T> = Option<Box<Node<T>>>; // 恶魔一只

struct Node<T> {
    elem: T,
    next: Link<T>,
}
```

现在删除恶魔：

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>,
}

type Link<T> = *mut Node<T>; // 嘿，新的好人卡，请查收

struct Node<T> {
    elem: T,
    next: Link<T>,
}
```

请大家牢记：当使用裸指针时，`Option` 对我们是相当不友好的，所以这里不再使用。在后面还将引入`NonNull`类型，但是现在还无需操心。

## 基本操作

`List::new` 与之前几乎没有区别：

```
use ptr;

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
}
```

`Push` 也几乎没区...

```
pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(
```

等等，我们不再使用 `Box` 了，既然如此，该怎么分配内存呢？

也许我们可以使用 `std::alloc::alloc`，但是大家想象一下拿着武士刀进厨房切菜的场景，所以，还是算了吧。

我们想要 `Box` 又不想要，这里有一个也许很野但是管用的方法：

```
struct Node<T> {
    elem: T,
    real_next: Option<Box<Node<T>>>,
    next: *mut Node<T>,
}
```

先创建一个 `Box`，并使用一个裸指针指向 `Box` 中的 `Node`，然后就一直使用该裸指针直到我们处理完 `Node` 且可以销毁它之时。最后，可以将 `Box` 从 `real_next` 中 `take` 出来，并 `drop` 掉。

从上面来看，这个非常符合我们之前的简化版借用栈模型？借用 `Box`，再借用一个裸指针，然后先弹出该裸指针，再弹出 `Box`，嗯，果然很符合。

但是问题来了，这样做看上去有趣，但是你能保证这个简化版借用栈顺利的工作吗？所以，我们还是使用 `Box::into_raw` 函数吧！

---

```
pub fn into_raw(b: Box<T>) -> *mut T
```

消费掉 Box (拿走所有权), 返回一个裸指针。该指针会被正确的对齐且不为 null

在调用该函数后, 调用者需要对之前被 Box 所管理的内存负责, 特别地, 调用者需要正确的清理 T 并释放相应的内存。最简单的方式是通过 Box::from\_raw 函数将裸指针再转回到 Box, 然后 Box 的析构器就可以自动执行清理了。

注意: 这是一个关联函数, 因此 b.into\_raw() 是不正确的, 我们得使用 Box::into\_raw(b)。因此该函数不会跟内部类型的同名方法冲突。

## 示例

将裸指针转换成 Box 以实现自动的清理:

```
let x = Box::new(String::from("Hello"));
let ptr = Box::into_raw(x);
let x = unsafe { Box::from_raw(ptr) };
```

---

太棒了, 简直为我们量身定制。而且它还很符合我们试图遵循的规则: 从安全的东东开始, 将其转换成裸指针, 最后再将裸指针转回安全的东东以实现安全的 drop。

现在, 我们就可以到处使用裸指针, 也无需再注意 unsafe 的范围, 反正现在都是 unsafe 了, 无所谓。

```
pub fn push(&mut self, elem: T) {
    unsafe {
        // 一开始就将 Box 转换成裸指针
        let new_tail = Box::into_raw(Box::new(Node {
            elem: elem,
            next: ptr::null_mut(),
        }));
        if !self.tail.is_null() {
            (*self.tail).next = new_tail;
        } else {
            self.head = new_tail;
        }
        self.tail = new_tail;
    }
}
```

嘿, 都说 unsafe 不应该使用, 但没想到 unsafe 真的是好! 现在代码整体看起来简洁多了。

继续实现 `pop`，它跟之前区别不大，但是我们不要忘了使用 `Box::from_raw` 来清理内存：

```
pub fn pop(&mut self) -> Option<T> {
    unsafe {
        if self.head.is_null() {
            None
        } else {
            let head = Box::from_raw(self.head);
            self.head = head.next;

            if self.head.is_null() {
                self.tail = ptr::null_mut();
            }

            Some(head.elem)
        }
    }
}
```

纪念下死去的 `take` 和 `map`，现在我们得手动检查和设置 `null` 了。

然后再实现下析构器，直接循环 `pop` 即可，怎么说，简单可爱，谁不爱呢？

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() { }
    }
}
```

现在到了检验正确性的时候：

```
#[cfg(test)]
mod test {
    use super::List;
#[test]
fn basics() {
    let mut list = List::new();

    // Check empty list behaves right
    assert_eq!(list.pop(), None);

    // Populate list
    list.push(1);
    list.push(2);
    list.push(3);

    // Check normal removal
    assert_eq!(list.pop(), Some(1));
    assert_eq!(list.pop(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push(4);
    list.push(5);

    // Check normal removal
    assert_eq!(list.pop(), Some(3));
    assert_eq!(list.pop(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop(), Some(5));
    assert_eq!(list.pop(), None);

    // Check the exhaustion case fixed the pointer right
    list.push(6);
    list.push(7);

    // Check normal removal
    assert_eq!(list.pop(), Some(6));
    assert_eq!(list.pop(), Some(7));
    assert_eq!(list.pop(), None);
}
}
```

```
$ cargo test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

测试没问题，还有一个拦路虎 miri 呢。

```
MIRIFLAGS="-Zmiri-tag=raw-pointers" cargo +nightly-2022-01-21 miri test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

苦尽甘来，苦尽甘来啊！我们这些章节的努力没有白费，它终于成功的工作了。

# 额外的操作

在搞定 `push`、`pop` 后，剩下的基本跟栈链表的实现没有啥区别。只有会改变链表长度的操作才会使用 `tail` 尾指针。

当然，现在一切都是裸指针，因此我们要重写代码来使用它们，在此过程中必须要确保没有遗漏地修改所有地方。

首先，先从栈链表实现中拷贝以下代码：

```
// ...

pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}
```

这里的 `Iter` 和 `IterMut` 并没有实现裸指针，先来修改下：

```

pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: *mut Node<T>,
}

pub struct IterMut<'a, T> {
    next: *mut Node<T>,
}

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        IterMut { next: self.head }
    }
}

```

看起来不错!

```

error[E0392]: parameter ``a`` is never used
--> src\fifth.rs:17:17
|
17 | pub struct Iter<'a, T> {
|         ^^^ unused parameter
|
= help: consider removing ``a``, referring to it in a field,
  or using a marker such as `PhantomData`


error[E0392]: parameter ``a`` is never used
--> src\fifth.rs:21:20
|
21 | pub struct IterMut<'a, T> {
|         ^^^ unused parameter
|
= help: consider removing ``a``, referring to it in a field,
  or using a marker such as `PhantomData`

```

咦? 这里的 [PhantomData](#) 是什么?

---

PhantomData<sup>zero sized</sup> 是零大小的类型

在你的类型中添加一个 `PhantomData<T>` 字段，可以告诉编译器你的类型对 `T` 进行了使用，虽然并没有。说白了，就是让编译器不再给出 `T` 未被使用的警告或者错误。

如果想要更深入的了解，可以看下 [Nomicon](#)

---

大概最适用于 `PhantomData` 的场景就是一个结构体拥有未使用的生命周期，典型的就是在 `unsafe` 中使用。

总之，之前的错误是可以通过 `PhantomData` 来解决的，但是我想将这个秘密武器留到下一章中的双向链表，它才是真正的需求。

那现在只能破坏我们之前的豪言壮语了，灰溜溜的继续使用引用貌似也是不错的选择。能使用引用的原因是：我们可以创建一个迭代器，在其中使用安全引用，然后再丢弃迭代器。一旦迭代器被丢弃后，就可以继续使用 `push` 和 `pop` 了。

事实上，在迭代期间，我们还是需要解引用大量的裸指针，但是可以把引用看作裸指针的再借用。

偷偷的说一句：对于这个方法，我不敢保证一定能成功，先来试试吧..

```
pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        unsafe {
            Iter { next: self.head.as_ref() }
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        unsafe {
            IterMut { next: self.head.as_mut() }
        }
    }
}
```

为了存储引用，这里使用 `Option` 来包裹，并通过 `ptr::as_ref` 和 `ptr::as_mut` 来将裸指针转换成引用。

通常，我会尽量避免使用 `as_ref` 这类方法，因为它们在做一些不可思议的转换！但是上面却是极少数可以使用的场景之一。

这两个方法的使用往往会伴随很多警告，其中最有趣的是：

---

你必须要遵循混叠(Aliasing)的规则，原因是返回的生命周期 '`a`' 只是任意选择的，并不能代表数据真实的生命周期。特别的，在这段生命周期的过程中，指针指向的内存区域绝不能被其它指针所访问。

---

好消息是，我们貌似不存在这个问题，因为混叠是我们一直在讨论和避免的问题。除此之外，还有一个恶魔：

```
pub unsafe fn as_mut<'a>(self) -> Option<&'a mut T>
```

大家注意到这个凭空出现的 '`a`' 吗？这里 `self` 是一个值类型，按照生命周期的规则，'`a`' 无根之木，它就是[无界生命周期](#)。

兄弟们，我很紧张，但是该继续的还是得继续，让我们从栈链表中再复制一些代码过来：

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.pop()
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        unsafe {
            self.next.map(|node| {
                self.next = node.next.as_ref();
                &node.elem
            })
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        unsafe {
            self.next.take().map(|node| {
                self.next = node.next.as_mut();
                &mut node.elem
            })
        }
    }
}
```

验证下测试用例：

```
cargo test

running 15 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test third::test::basics ... ok

test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

还有 miri:

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test

running 15 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

嗯，还有 peek 和 peek\_mut 的实现：

```

pub fn peek(&self) -> Option<&T> {
    unsafe {
        self.head.as_ref()
    }
}

pub fn peek_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.head.as_mut()
    }
}

```

实现这么简单，运行起来肯定没问题：

```

$ cargo build
error[E0308]: mismatched types
--> src\fifth.rs:66:13
|
25 |     impl<T> List<T> {
|         |
|             pub fn peek(&self) -> Option<&T> {
|                             ----- expected `Option<&T>`
|                             because of return type
|             unsafe {
|                 self.head.as_ref()
|                 ^^^^^^^^^^^^^^ expected type parameter `T` ,
|                               found struct `fifth::Node`
|
| = note: expected enum `Option<&T>`
|           found enum `Option<&fifth::Node<T>>`

```

哦，这个简单，map 以下就可以了：

```

pub fn peek(&self) -> Option<&T> {
    unsafe {
        self.head.as_ref().map(|node| &node.elem)
    }
}

pub fn peek_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.head.as_mut().map(|node| &mut node.elem)
    }
}

```

我感觉有很多错误正在赶来的路上，因此大家需要提高警惕，要么先写一个测试吧：把我们的 API 都混合在一起，让 miri 来享用 - miri food!

```
#[test]
fn miri_food() {
    let mut list = List::new();

    list.push(1);
    list.push(2);
    list.push(3);

    assert!(list.pop() == Some(1));
    list.push(4);
    assert!(list.pop() == Some(2));
    list.push(5);

    assert!(list.peek() == Some(&3));
    list.push(6);
    list.peek_mut().map(|x| *x *= 10);
    assert!(list.peek() == Some(&30));
    assert!(list.pop() == Some(30));

    for elem in list.iter_mut() {
        *elem *= 100;
    }

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&400));
    assert_eq!(iter.next(), Some(&500));
    assert_eq!(iter.next(), Some(&600));
    assert_eq!(iter.next(), None);
    assert_eq!(iter.next(), None);

    assert!(list.pop() == Some(400));
    list.peek_mut().map(|x| *x *= 10);
    assert!(list.peek() == Some(&5000));
    list.push(7);

    // Drop it on the ground and let the dtor exercise itself
}
```

```
cargo test

running 16 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test
```

```
running 16 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

完美。

# 最终代码

得益于不安全代码的引入，新的实现可以获得线性的性能提升，同时我们还设法复用了栈链表的很多代码。

当然，这个过程中，我们还引入了新的概念，例如借用栈，相信直到现在有些同学还晕乎乎的。不管如何，我们不用再去写一大堆嵌套来嵌套去的 `Rc` 和 `RefCell`。

下面来看看咱们这个不安全链表的全貌吧。

```
use std::ptr;

pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>,
}

type Link<T> = *mut Node<T>;
```

```
struct Node<T> {
    elem: T,
    next: Link<T>,
}
```

```
pub struct IntoIter<T>(List<T>);
```

```
pub struct Iter<'a, T> {
    next: Option<&'a mut Node<T>>,
}
```

```
pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}
```

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
    pub fn push(&mut self, elem: T) {
        unsafe {
            let new_tail = Box::into_raw(Box::new(Node {
                elem: elem,
                next: ptr::null_mut(),
            }));
            if !self.tail.is_null() {
                (*self.tail).next = new_tail;
            } else {
                self.head = new_tail;
            }
            self.tail = new_tail;
        }
    }
    pub fn pop(&mut self) -> Option<T> {
        unsafe {
            if self.head.is_null() {
                None
            } else {
                let head = Box::from_raw(self.head);
                self.head = head.next;
            }
        }
    }
}
```

```
        if self.head.is_null() {
            self.tail = ptr::null_mut();
        }

        Some(head.elem)
    }
}

pub fn peek(&self) -> Option<&T> {
    unsafe {
        self.head.as_ref().map(|node| &node.elem)
    }
}

pub fn peek_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.head.as_mut().map(|node| &mut node.elem)
    }
}

pub fn into_iter(self) -> IntoIter<T> {
    IntoIter(self)
}

pub fn iter(&self) -> Iter<'_, T> {
    unsafe {
        Iter { next: self.head.as_ref() }
    }
}

pub fn iter_mut(&mut self) -> IterMut<'_, T> {
    unsafe {
        IterMut { next: self.head.as_mut() }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() { }
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.pop()
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
}
```

```
fn next(&mut self) -> Option<Self::Item> {
    unsafe {
        self.next.map(|node| {
            self.next = node.next.as_ref();
            &node.elem
        })
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        unsafe {
            self.next.take().map(|node| {
                self.next = node.next.as_mut();
                &mut node.elem
            })
        }
    }
}

#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
    }
}
```

```
assert_eq!(list.pop(), None);

// Check the exhaustion case fixed the pointer right
list.push(6);
list.push(7);

// Check normal removal
assert_eq!(list.pop(), Some(6));
assert_eq!(list.pop(), Some(7));
assert_eq!(list.pop(), None);
}

#[test]
fn into_iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next(), None);
}

#[test]
fn iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&1));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), None);
}

#[test]
fn iter_mut() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter_mut();
    assert_eq!(iter.next(), Some(&mut 1));
    assert_eq!(iter.next(), Some(&mut 2));
    assert_eq!(iter.next(), Some(&mut 3));
    assert_eq!(iter.next(), None);
}

#[test]
fn miri_food() {
    let mut list = List::new();

    list.push(1);
```

```
list.push(2);
list.push(3);

assert!(list.pop() == Some(1));
list.push(4);
assert!(list.pop() == Some(2));
list.push(5);

assert!(list.peek() == Some(&3));
list.push(6);
list.peek_mut().map(|x| *x *= 10);
assert!(list.peek() == Some(&30));
assert!(list.pop() == Some(30));

for elem in list.iter_mut() {
    *elem *= 100;
}

let mut iter = list.iter();
assert_eq!(iter.next(), Some(&400));
assert_eq!(iter.next(), Some(&500));
assert_eq!(iter.next(), Some(&600));
assert_eq!(iter.next(), None);
assert_eq!(iter.next(), None);

assert!(list.pop() == Some(400));
list.peek_mut().map(|x| *x *= 10);
assert!(list.peek() == Some(&5000));
list.push(7);

// Drop it on the ground and let the dtor exercise itself
}
```

# 使用高级技巧实现链表

说句实话，我们之前实现的链表都达不到生产级可用的程度，而且也没有用到一些比较时髦的技巧。

本章我们一起来看一些更时髦的链表实现：

1. 生产级可用的双向链表
2. 双重单向链表
3. 栈分配的链表
4. 自引用和Arena分配器实现( 原文作者还未实现，所以... Todo )
5. GhostCell 实现( 同上 )

# 生产级可用的双向链表

打开[原文](#)，发现这一篇只有两行，我以为自己看花了眼，揉了揉眼，定睛一看，还是两行。

没错，貌似作者想要偷懒，而且为了掩饰，他还提供了标准库的实现:) 如果大家想要学习，看[标准库吧](#) :D

---

为了能更好的看懂标准库实现，你可能还需要这本书的辅助: [Rustonomicon](#)

---

# 双双向链表

在之前的双向链表章节中，我们一度非常纠结，原因来自同样纠结成一团的所有权依赖。还有一个重要原因就是：先入为主的链表定义。

谁说所有的链接一定要一个方向呢？这里一起来尝试下新的东东：链表的其中一半朝左，另一半朝右。

新规矩(老规矩是创建文件)，创建一个新的模块：

```
// lib.rs
// ...
pub mod silly1;      // NEW!

// silly1.rs
use crate::second::List as Stack;

struct List<T> {
    left: Stack<T>,
    right: Stack<T>,
}
```

这里将之前的 `List` 引入进来，并重命名为 `Stack`，接着，创建一个新的链表。现在既可以向左增长又可以向右增长。

```
pub struct Stack<T> {
    head: Link<T>,
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Self {
        Stack { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            let node = *node;
            self.head = node.next;
            node.elem
        })
    }

    pub fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| {
            &node.elem
        })
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        self.head.as_mut().map(|node| {
            &mut node.elem
        })
    }
}

impl<T> Drop for Stack<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

```
    }
}
```

稍微修改下 push 和 pop :

```
pub fn push(&mut self, elem: T) {
    let new_node = Box::new(Node {
        elem: elem,
        next: None,
    });
    self.push_node(new_node);
}

fn push_node(&mut self, mut node: Box<Node<T>>) {
    node.next = self.head.take();
    self.head = Some(node);
}

pub fn pop(&mut self) -> Option<T> {
    self.pop_node().map(|node| {
        node.elem
    })
}

fn pop_node(&mut self) -> Option<Box<Node<T>>> {
    self.head.take().map(|mut node| {
        self.head = node.next.take();
        node
    })
}
```

现在可以开始构造新的链表:

```
pub struct List<T> {
    left: Stack<T>,
    right: Stack<T>,
}

impl<T> List<T> {
    fn new() -> Self {
        List { left: Stack::new(), right: Stack::new() }
    }
}
```

当然，还有一大堆左左右右类型的操作:

```
pub fn push_left(&mut self, elem: T) { self.left.push(elem) }
pub fn push_right(&mut self, elem: T) { self.right.push(elem) }
pub fn pop_left(&mut self) -> Option<T> { self.left.pop() }
pub fn pop_right(&mut self) -> Option<T> { self.right.pop() }
pub fn peek_left(&self) -> Option<&T> { self.left.peek() }
pub fn peek_right(&self) -> Option<&T> { self.right.peek() }
pub fn peek_left_mut(&mut self) -> Option<&mut T> { self.left.peek_mut() }
pub fn peek_right_mut(&mut self) -> Option<&mut T> { self.right.peek_mut() }
```

其中最有趣的是：还可以来回闲逛了。

```
pub fn go_left(&mut self) -> bool {
    self.left.pop_node().map(|node| {
        self.right.push_node(node);
    }).is_some()
}

pub fn go_right(&mut self) -> bool {
    self.right.pop_node().map(|node| {
        self.left.push_node(node);
    }).is_some()
}
```

这里返回 `bool` 是为了告诉调用者我们是否成功的移动。最后，再来测试下：

```
#[cfg(test)]
mod test {
    use super::List;

#[test]
fn walk_aboot() {
    let mut list = List::new(); // []

    list.push_left(0); // [0, _]
    list.push_right(1); // [0, _, 1]
    assert_eq!(list.peek_left(), Some(&0));
    assert_eq!(list.peek_right(), Some(&1));

    list.push_left(2); // [0, 2, _, 1]
    list.push_left(3); // [0, 2, 3, _, 1]
    list.push_right(4); // [0, 2, 3, _, 4, 1]

    while list.go_left() {} // [_, 0, 2, 3, 4, 1]

    assert_eq!(list.pop_left(), None);
    assert_eq!(list.pop_right(), Some(0)); // [_, 2, 3, 4, 1]
    assert_eq!(list.pop_right(), Some(2)); // [_, 3, 4, 1]

    list.push_left(5); // [5, _, 3, 4, 1]
    assert_eq!(list.pop_right(), Some(3)); // [5, _, 4, 1]
    assert_eq!(list.pop_left(), Some(5)); // [_, 4, 1]
    assert_eq!(list.pop_right(), Some(4)); // [_, 1]
    assert_eq!(list.pop_right(), Some(1)); // []

    assert_eq!(list.pop_right(), None);
    assert_eq!(list.pop_left(), None);

}
}
```

```
$ cargo test

    Running target/debug/lists-5c71138492ad4b4a

running 16 tests
test fifth::test::into_iter ... ok
test fifth::test::basics ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fourth::test::into_iter ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test first::test::basics ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test third::test::basics ... ok
test third::test::iter ... ok
test second::test::peek ... ok
test silly1::test::walk_aboot ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured
```

上上下下，左左右右，BABA，哦耶，这个链表无敌了！

以上是一个非常典型的手指型数据结构，在其中维护一个手指，然后操作所需的时间与手指的距离成正比。

# 栈上的链表

在之前的章节中，无一例外，我们创建的都是数据存储在堆上的链表，这种链表最常见也最实用：堆内存动态分配的场景非常好用。

但是，既然是高级技巧章节，那栈链表也应该拥有一席之地。但与堆内存的简单分配相比，栈内存就没那么友好了，你们猜大名鼎鼎的 C 语言的 `alloca` 是因为什么而出名的：)

限于章节篇幅，这里我们使用一个简单的栈分配方法：调用一个函数，获取一个新的、拥有更多空间的栈帧。说实话，该解决方法要多愚蠢有多愚蠢，但是它确实相当实用，甚至...有用。

任何时候，当我们在做一些递归的任务时，都可以将当前步骤状态的指针传递给下一个步骤。如果指针本身就是状态的一部分，那恭喜你：你在创建一个栈上分配的链表！

新的链表类型本身就是一个 `Node`，并且包含一个引用指向另一个 `Node`:

```
pub struct List<'a, T> {
    pub data: T,
    pub prev: Option<&'a List<'a, T>>,
}
```

该链表只有一个操作 `push`，需要注意的是，跟其它链表不同，这里的 `push` 是通过回调的方式来完成新元素推入，并将回调返回的值直接返回给 `push` 的调用者：

```
impl<'a, T> List<'a, T> {
    pub fn push<U>(
        prev: Option<&'a List<'a, T>>,
        data: T,
        callback: impl FnOnce(&List<'a, T>) -> U,
    ) -> U {
        let list = List { data, prev };
        callback(&list)
    }
}
```

搞定，提前问一句：你见过回调地狱吗？

```
List::push(None, 3, |list| {
    println!("{}", list.data);
    List::push(Some(list), 5, |list| {
        println!("{}", list.data);
        List::push(Some(list), 13, |list| {
            println!("{}", list.data);
        })
    })
})
```

不禁让人感叹，这段回调代码多么的美丽动人。。

用户还可以简单地使用 `while-let` 的方式来编译遍历链表，但是为了增加一些趣味，咱们还是继续使用迭代器：

```
impl<'a, T> List<'a, T> {
    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: Some(self) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.prev;
            &node.data
        })
    }
}
```

测试下：

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn elegance() {
        List::push(None, 3, |list| {
            assert_eq!(list.iter().copied().sum::<i32>(), 3);
            List::push(Some(list), 5, |list| {
                assert_eq!(list.iter().copied().sum::<i32>(), 5 + 3);
                List::push(Some(list), 13, |list| {
                    assert_eq!(list.iter().copied().sum::<i32>(), 13 + 5 + 3);
                })
            })
        })
    }
}
```

```
$ cargo test
```

```
running 18 tests
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::basics ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test second::test::into_iter ... ok
test fourth::test::peek ... ok
test fourth::test::into_iter ... ok
test second::test::iter_mut ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test third::test::basics ... ok
test silly1::test::walk_aboot ... ok
test silly2::test::elegance ... ok
test second::test::peek ... ok
test third::test::iter ... ok
```

```
test result: ok. 18 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

部分读者此时可能会有一些大胆的想法：咦？我能否修改 Node 中的值？大胆但貌似可行，不妨来试试。

```
pub struct List<'a, T> {
    pub data: T,
    pub prev: Option<&'a mut List<'a, T>>,
}

pub struct Iter<'a, T> {
    next: Option<&'a List<'a, T>>,
}

impl<'a, T> List<'a, T> {
    pub fn push<U>(
        prev: Option<&'a mut List<'a, T>>,
        data: T,
        callback: impl FnOnce(&mut List<'a, T>) -> U,
    ) -> U {
        let mut list = List { data, prev };
        callback(&mut list)
    }

    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: Some(self) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.prev.as_ref().map(|prev| &mut prev);
            &node.data
        })
    }
}
```

```

$ cargo test

error[E0521]: borrowed data escapes outside of closure
--> src\silly2.rs:47:32
|
46 |     List::push(Some(list), 13, |list| {
|             |
|             |
|             `list` declared here, outside of the closure body
|             `list` is a reference that is only valid in the closure body
47 |     assert_eq!(list.iter().copied().sum::<i32>(), 13 + 5 + 3);
|                     ^^^^^^^^^^ `list` escapes the closure body here

error[E0521]: borrowed data escapes outside of closure
--> src\silly2.rs:45:28
|
44 |     List::push(Some(list), 5, |list| {
|             |
|             |
|             `list` declared here, outside of the closure body
|             `list` is a reference that is only valid in the closure body
45 |     assert_eq!(list.iter().copied().sum::<i32>(), 5 + 3);
|                     ^^^^^^^^^^ `list` escapes the closure body here

```

<ad infinitum>

嗯，没想到是浓眉大眼的迭代器背叛了我们，为了验证到底是哪里出了问题，我们来修改下测试：

```

#[test]
fn elegance() {
    List::push(None, 3, |list| {
        assert_eq!(list.data, 3);
        List::push(Some(list), 5, |list| {
            assert_eq!(list.data, 5);
            List::push(Some(list), 13, |list| {
                assert_eq!(list.data, 13);
            })
        })
    })
}

```

```

$ cargo test

error[E0521]: borrowed data escapes outside of closure
--> src\silly2.rs:46:17
|
44 |     List::push(Some(list), 5, |list| {
|         |
|             |
|                 `list` declared here, outside of the closure body
|                 `list` is a reference that is only valid in the closure body
45 |         assert_eq!(list.data, 5);
46 |     /     List::push(Some(list), 13, |list| {
47 |     |         assert_eq!(list.data, 13);
48 |     |     })
|     |-----^ `list` escapes the closure body here

error[E0521]: borrowed data escapes outside of closure
--> src\silly2.rs:44:13
|
42 |     List::push(None, 3, |list| {
|         |
|             |
|                 `list` declared here, outside of the closure body
|                 `list` is a reference that is only valid in the closure body
43 |         assert_eq!(list.data, 3);
44 |     /     List::push(Some(list), 5, |list| {
45 |     |         assert_eq!(list.data, 5);
46 |     |     List::push(Some(list), 13, |list| {
47 |     |         assert_eq!(list.data, 13);
48 |     |     })
49 |     | }
|     |-----^ `list` escapes the closure body here

```

原因在于我们的链表不小心依赖了型变。型变是一个相当复杂的概念，下面来简单了解下。

每一个节点( Node )都包含一个引用，该引用指向另一个节点，且这两个节点是同一个类型。如果从最里面的节点角度来看，那所有外部的节点都在使用和它一样的生命周期，但这个显然是不对的：链表中的每一个节点都会比它指向的节点活得更久，因为它们的作用域是嵌套存在的。

那之前的不可变引用版本为何可以正常工作呢？原因是在大多数时候，编译器都能自己判断：虽然某些东东活得太久了，但是这是安全的。当我们把一个 List 塞入另一个时，编译器会迅速将生命周期进行收缩以满足新的 List 的需求，**这种生命周期收缩就是一种型变**。

如果大家还是觉得不太理解，我们来考虑下其它拥有继承特性的编程语言。在该语言中，当你将一个 Cat 传递给需要 Animal 的地方时( Animal 是 Cat 的父类型)，型变就发生了。从字面来说，将一只猫传给需要动物的地方，也是合适的，毕竟猫确实是动物的一种。

总之，可以看出无论是从大的生命周期收缩为小的生命周期，还是从 `Cat` 到 `Animal`，型变的典型特征就是：范围在减小，毕竟子类型的功能肯定是比父类型多的。

既然有型变，为何可变引用的版本会报错呢？其实在于型变不总是安全的，假如之前的代码可以编译，那我们可以写出释放后再使用 的代码：

```
List::push(None, 3, |list| {
    List::push(Some(list), 5, |list| {
        List::push(Some(list), 13, |list| {
            // 哈哈，好爽，由于所有的生命周期都是相同的，因此编译器允许我重写父节点，并让它持有一个
            // 可变指针指向我自己。
            // 我将创建所有的 use-after-free !
            *list.prev.as_mut().unwrap().prev = Some(list);
        })
    })
})
```

一旦引入可变性，型变就会造成这样的隐患：意外修改了不该被修改的代码，但这些代码的调用者还在期待着和往常一样的结果！例如以下例子：

```
let mut my_kitty = Cat;           // Make a Cat (long lifetime)
let animal: &mut Animal = &mut my_kitty; // Forget it's a Cat (shorten lifetime)
*animal = Dog;                   // Write a Dog (short lifetime)
my_kitty.meow();                // Meowing Dog! (Use After Free)
```

我们将长生命周期的猫转换成短生命周期的动物，可变的！然后通过短生命周期的动物将指针重新指向一只狗。此时我们想去撸软萌猫的时候，就听到：旺旺...呜嗷嗷，对，你没听错，不仅没有了猫叫，甚至于狗还没叫完，就可能在某个地方又被修改成狼了。

因此，**虽然你可以修改可变引用的生命周期，但是一旦开始嵌套，它们就将失去型变，变成不变(invariant)**。此时，就再也无法对生命周期进行收缩了。

具体来说：`&mut &'big mut T` 无法被转换成 `&mut &'small mut T`，这里 `'big` 代表比 `'small` 更大的生命周期。或者用更正式的说法：`&'a mut T` 对于 `'a` 来说是协变(covariant)的，但是对于 `T` 是不变的(invariant)。

---

说了这么多高深的理论，那么该如何改变链表的数据呢？答案就是：使用老本行 - 内部可变性。

下面让我们回滚到之前的不可变版本，然后使用 `Cell` 来替代 `&mut`。

```

#[test]
fn cell() {
    use std::cell::Cell;

    List::push(None, Cell::new(3), |list| {
        List::push(Some(list), Cell::new(5), |list| {
            List::push(Some(list), Cell::new(13), |list| {
                // Multiply every value in the list by 10
                for val in list.iter() {
                    val.set(val.get() * 10)
                }

                let mut vals = list.iter();
                assert_eq!(vals.next().unwrap().get(), 130);
                assert_eq!(vals.next().unwrap().get(), 50);
                assert_eq!(vals.next().unwrap().get(), 30);
                assert_eq!(vals.next(), None);
                assert_eq!(vals.next(), None);
            })
        })
    })
}

```

\$ cargo test

```

running 19 tests
test fifth::test::into_iter ... ok
test fifth::test::basics ... ok
test fifth::test::iter_mut ... ok
test fifth::test::iter ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test first::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fifth::test::miri_food ... ok
test silly2::test::cell ... ok
test third::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test silly1::test::walk_aboot ... ok
test silly2::test::elegance ... ok
test third::test::basics ... ok
test second::test::iter ... ok

test result: ok. 19 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;

```

简简单单搞定，虽然之前我们嫌弃内部可变性，但是在这里：真香！

# **征服编译错误**

# **对抗编译检查**

# **生命周期**

本章并不讲太多的概念，主要是用例子来引导大家去思考该如何对抗编译检查。

# 生命周期声明的范围过大

在大多时候，Rust 的生命周期你只要标识了，即可以通过编译，但是总是存在一些情况，会导致编译无法通过，本文就讲述这样一种情况：因为生命周期声明的范围过大，导致了编译无法通过，希望大家喜欢

## 例子 1

```
struct Interface<'a> {
    manager: &'a mut Manager<'a>
}

impl<'a> Interface<'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>,
}

impl<'a> List<'a> {
    pub fn get_interface(&'a mut self) -> Interface {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // this fails because immutable/mutable borrow
    // but Interface should be already dropped here and the borrow released
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}
```

运行后报错：

```
error[E0502]: cannot borrow `list` as immutable because it is also borrowed as
mutable // `list`无法被借用, 因为已经被可变借用
--> src/main.rs:40:14
|
34 |     list.get_interface().noop();
|     ---- mutable borrow occurs here // 可变借用发生在这里
...
40 |     use_list(&list);
|     ^^^^^^
|     |
|     immutable borrow occurs here // 新的不可变借用发生在这
|     mutable borrow later used here // 可变借用在这里结束
```

这段代码看上去并不复杂，实际上难度挺高的，首先在直觉上，`list.get_interface()` 借用的可变引用，按理来说应该在这行代码结束后，就归还了，为何能持续到 `use_list(&list)` 后面呢？

这是因为我们在 `get_interface` 方法中声明的 `lifetime` 有问题，该方法的参数的生命周期是 '`a`'，而 `List` 的生命周期也是 '`a`'，说明该方法至少活得跟 `List` 一样久，再回到 `main` 函数中，`list` 可以活到 `main` 函数的结束，因此 `list.get_interface()` 借用的可变引用也会活到 `main` 函数的结束，在此期间，自然无法再进行借用了。

要解决这个问题，我们需要为 `get_interface` 方法的参数给予一个不同于 `List<'a>` 的生命周期 '`b`'，最终代码如下：

```

struct Interface<'b, 'a: 'b> {
    manager: &'b mut Manager<'a>
}

impl<'b, 'a: 'b> Interface<'b, 'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>,
}

impl<'a> List<'a> {
    pub fn get_interface<'b>(&'b mut self) -> Interface<'b, 'a>
    where 'a: 'b {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // this fails because immutable/mutable borrow
    // but Interface should be already dropped here and the borrow released
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}

```

当然，咱还可以给生命周期给予更有意义的名称：

```
struct Interface<'text, 'manager> {
    manager: &'manager mut Manager<'text>
}

impl<'text, 'manager> Interface<'text, 'manager> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'text> {
    text: &'text str
}

struct List<'text> {
    manager: Manager<'text>,
}

impl<'text> List<'text> {
    pub fn get_interface<'manager>(&'manager mut self) -> Interface<'text, 'manager>
    where 'text: 'manager {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // this fails because immutable/mutable borrow
    // but Interface should be already dropped here and the borrow released
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{} {}", list.manager.text);
}
```

# 生命周期过大-02

继上篇文章后，我们再来看一段可能涉及生命周期过大导致的无法编译问题：

```
fn bar(writer: &mut Writer) {
    baz(writer.indent());
    writer.write("world");
}

fn baz(writer: &mut Writer) {
    writer.write("hello");
}

pub struct Writer<'a> {
    target: &'a mut String,
    indent: usize,
}

impl<'a> Writer<'a> {
    fn indent(&'a mut self) -> &'a mut Self {
        &mut Self {
            target: self.target,
            indent: self.indent + 1,
        }
    }

    fn write(&mut self, s: &str) {
        for _ in 0..self.indent {
            self.target.push(' ');
        }
        self.target.push_str(s);
        self.target.push('\n');
    }
}

fn main() {}
```

报错如下：

```
error[E0623]: lifetime mismatch
--> src/main.rs:2:16
1 | fn bar(writer: &mut Writer) {
|     -----
|     |
|         these two types are declared with different lifetimes...
2 |     baz(writer.indent());
|             ^^^^^^ ...but data from `writer` flows into `writer` here
```

WTF，这什么报错，之前都没有见过，而且很难理解，什么叫 writer 滑入了另一个 writer？

别急，我们先来仔细看下代码，注意这一段：

```
impl<'a> Writer<'a> {
    fn indent(&'a mut self) -> &'a mut Self {
        &mut Self {
            target: self.target,
            indent: self.indent + 1,
        }
    }
}
```

这里的生命周期定义说明 indent 方法使用的。。。等等！你的代码错了，你怎么能在一个函数中返回一个新创建实例的引用？！！最重要的是，编译器不提示这个错误，竟然提示一个莫名其妙看不懂的东东。

行，那我们先解决这个问题，将该方法修改为：

```
fn indent(&'a mut self) -> Writer<'a> {
    Writer {
        target: self.target,
        indent: self.indent + 1,
    }
}
```

怀着惴惴这心，再一次运行程序，果不其然，编译器又朝我们扔了一坨错误：

```
error[E0308]: mismatched types
--> src/main.rs:2:9
|
2 |     baz(writer.indent());
|     ^^^^^^^^^^^^^^^^^^
|     |
|     expected `&mut Writer<'_>`, found struct `Writer`
|     help: consider mutably borrowing here: `&mut writer.indent()`
```

哦，这次错误很明显，因为 baz 需要 &mut Writer，但是咱们 writer.indent 返回了一个 Writer，因此修改下即可：

```
fn bar(writer: &mut Writer) {
    baz(&mut writer.indent());
    writer.write("world");
}
```

这次总该成功了吧？再次心慌慌的运行编译器，哐：

```
error[E0623]: lifetime mismatch
--> src/main.rs:2:21
1 | fn bar(writer: &mut Writer) {
|   -----
|   |
|   |     these two types are declared with different lifetimes...
2 |     baz(&mut writer.indent());
|           ^^^^^^ ...but data from `writer` flows into `writer` here
```

可恶，还是这个看不懂的错误，仔细检查了下代码，这次真的没有其他错误了，只能硬着头皮上。

大概的意思可以分析，生命周期范围不匹配，说明一个大一个小，然后一个 `writer` 中流入到另一个 `writer` 说明，两个 `writer` 的生命周期定义错了，既然这里提到了 `indent` 方法调用，那么我们再去仔细看一眼：

```
impl<'a> Writer<'a> {
    fn indent(&'a mut self) -> Writer<'a> {
        Writer {
            target: self.target,
            indent: self.indent + 1,
        }
    }
    ...
}
```

好像有点问题，`indent` 返回的 `Writer` 的生命周期和外面调用者的 `Writer` 的生命周期一模一样，这很不合理，一眼就能看出前者远小于后者。

这里稍微展开以下，为何 `indent` 方法返回值的生命周期不能与参数中的 `self` 相同。首先，我们假设它们可以相同，也就是上面的代码可以编译通过，由于此时在返回值中借用了 `self` 的可变引用，意味着如果你在返回值被使用后，还继续使用 `self` 会导致重复借用的错误，因为返回值的生命周期将持续到 `self` 结束。

既然不能相同，那我们尝试着修改下 `indent`：

```
fn indent<'b>(&'b mut self) -> Writer<'b> {
    Writer {
        target: self.target,
        indent: self.indent + 1,
    }
}
```

Bang! 编译成功，不过稍等，回想下生命周期消除的规则，我们还可以实现的更优雅：

```
fn bar(writer: &mut Writer) {
    baz(&mut writer.indent());
    writer.write("world");
}

fn baz(writer: &mut Writer) {
    writer.write("hello");
}

pub struct Writer<'a> {
    target: &'a mut String,
    indent: usize,
}

impl<'a> Writer<'a> {
    fn indent(&mut self) -> Writer {
        Writer {
            target: self.target,
            indent: self.indent + 1,
        }
    }

    fn write(&mut self, s: &str) {
        for _ in 0..self.indent {
            self.target.push(' ');
        }
        self.target.push_str(s);
        self.target.push('\n');
    }
}

fn main() {}
```

至此，问题彻底解决，太好了，我感觉我又变强了。可是默默看了眼自己的头发，只能以哎~一声叹息结束本章内容。

# 蠢笨编译器之循环生命周期

当涉及生命周期时，Rust 编译器有时会变得不太聪明，如果再配合循环，蠢笨都不足以形容它，不信？那继续跟着我一起看看。

## 循环中的生命周期错误

Talk is cheap, 一起来看个例子：

```
use rand::{thread_rng, Rng};

#[derive(Debug, PartialEq)]
enum Tile {
    Empty,
}

fn random_empty_tile(arr: &mut [Tile]) -> &mut Tile {
    loop {
        let i = thread_rng().gen_range(0..arr.len());
        let tile = &mut arr[i];
        if Tile::Empty == *tile {
            return tile;
        }
    }
}
```

我们来看看上面的代码中，`loop` 循环有几个引用：

- `arr.len()`, 一个不可变引用，生命周期随着函数调用的结束而结束
- `tile` 是可变引用，生命周期在下次循环开始前会结束

根据以上的分析，可以得出个初步结论：在同一次循环间各个引用生命周期互不影响，在两次循环间，引用也互不影响。

那就简单了，开心运行，开心。。。报错：

```

error[E0502]: cannot borrow `*arr` as immutable because it is also borrowed as
mutable
--> src/main.rs:10:43
|
8 | fn random_empty_tile(arr: &mut [Tile]) -> &mut Tile {
|                         - let's call the lifetime of this reference ''1``
9 |     loop {
10 |         let i = thread_rng().gen_range(0..arr.len());
|                           ^^^ immutable borrow occurs here
11 |         let tile = &mut arr[i];
|                         ----- mutable borrow occurs here
12 |         if Tile::Empty == *tile{
13 |             return tile;
|                 ---- returning this value requires that `arr[_]` is borrowed
for ''1``

error[E0499]: cannot borrow `arr[_]` as mutable more than once at a time
--> src/main.rs:11:20
|
8 | fn random_empty_tile(arr: &mut [Tile]) -> &mut Tile {
|                         - let's call the lifetime of this reference ''1``
...
11 |         let tile = &mut arr[i];
|             ^^^^^^^^^^^^^ `arr[_]` was mutably borrowed here in the
previous iteration of the loop
12 |         if Tile::Empty == *tile{
13 |             return tile;
|                 ---- returning this value requires that `arr[_]` is borrowed
for ''1``

```

不仅是错误，还是史诗级别的错误！无情刷屏了！只能想办法梳理下：

1. `arr.len()` 报错，原因是它借用了不可变引用，但是在紧跟着的 `&mut arr[i]` 中又借用了可变引用
2. `&mut arr[i]` 报错，因为在上一次循环中，已经借用过同样的可变引用 `&mut arr[i]`
3. `tile` 的生命周期跟 `arr` 不一致

奇了怪了，跟我们之前的分析完全背道而驰，按理来说 `arr.len()` 的借用应该在调用后立刻结束，而不是持续到后面的代码行；同时可变借用 `&mut arr[i]` 也应该随着每次循环的结束而结束，为什么前两次循环会因为同一处的引用而报错？

## 尝试去掉中间变量

虽然报错复杂，不过可以看出，所有的错误都跟 `tile` 这个中间变量有关，我们试着移除它看看：

```
use rand::{thread_rng, Rng};

#[derive(Debug, PartialEq)]
enum Tile {
    Empty,
}

fn random_empty_tile(arr: &mut [Tile]) -> &mut Tile {
    loop {
        let i = thread_rng().gen_range(0..arr.len());
        if Tile::Empty == arr[i] {
            return &mut arr[i];
        }
    }
}
```

见证奇迹的时刻，竟然编译通过了！到底发生了什么？仅仅移除了中间变量，就编译通过了？是否可以大胆的猜测，因为中间变量，导致编译器变蠢了，因此无法正确的识别引用的生命周期。

## 循环展开

如果不使用循环呢？会不会也有这样的错误？咱们试着把循环展开：

```

use rand::{thread_rng, Rng};

#[derive(Debug, PartialEq)]
enum Tile {
    Empty,
}

fn random_empty_tile_2<'arr>(arr: &'arr mut [Tile]) -> &'arr mut Tile {
    let len = arr.len();

    // First loop iteration
    {
        let i = thread_rng().gen_range(0..len);
        let tile = &mut arr[i]; // Lifetime: 'arr
        if Tile::Empty == *tile {
            return tile;
        }
    }

    // Second loop iteration
    {
        let i = thread_rng().gen_range(0..len);
        let tile = &mut arr[i]; // Lifetime: 'arr
        if Tile::Empty == *tile {
            return tile;
        }
    }

    unreachable!()
}

```

结果，编译器还是不给通过，报的错误几乎一样

## 深层原因

令人沮丧的是，我找遍了网上，也没有具体的原因，大家都说这是编译器太笨导致的问题，但是关于深层的原因，也没人能说出个所有然。

因此，我无法在本文中给出为什么编译器会这么笨的真实原因，如果以后有结果，会在这里进行更新。

-----2022年1月13日更新-----兄弟们，我带着挖掘出的一些内容回来了，再来看段错误代码先：

```

struct A {
    a: i32
}

impl A {
    fn one(&mut self) -> &i32{
        self.a = 10;
        &self.a
    }
    fn two(&mut self) -> &i32 {
        loop {
            let k = self.one();
            if *k > 10i32 {
                return k;
            }

            // 可能存在的剩余代码
            // ...
        }
    }
}

```

我们来逐步深入分析下：

- 首先为 two 方法增加一下生命周期标识: `fn two<'a>(&'a mut self) -> &'a i32 { .. }`, 这里根据生命周期的消除规则添加的
- 根据生命周期标识可知: two 中返回的 k 的生命周期必须是 'a
- 根据第 2 条, 又可知: `let k = self.one();` 中对 self 的借用生命周期也是 'a
- 因为 k 的借用发生在 loop 循环内, 因此它需要小于等于循环的生命周期, 但是根据之前的推断, 它又要大于等于函数的生命周期 'a , 而函数的生命周期又大于等于循环生命周期,

由上可以推出: `let k = self.one();` 中 k 的生命周期要大于等于循环的生命周期, 又要小于等于循环的生命周期, 唯一满足条件的就是: k 的生命周期等于循环生命周期。

但是我们的 two 方法在循环中对 k 进行了提前返回, 编译器自然会认为存在其它代码, 这会导致 k 的生命周期小于循环的生命周期。

怎么办呢? 很简单:

```

fn two(&mut self) -> &i32 {
    loop {
        let k = self.one();
        return k;
    }
}

```

不要在 if 分支中返回 k , 而是直接返回, 这样就让它们的生命周期相等了, 最终可以顺利编译通过。

---

如果一个引用值从函数的某个路径提前返回了，那么该借用必须要在函数的所有返回路径都合法

---

## 解决方法

虽然不能给出原因，但是我们可以看看解决办法，在上面，**移除中间变量和消除代码分支**都是可行的方法，还有一种方法就是将部分引用移到循环外面。

### 引用外移

```
fn random_empty_tile(arr: &mut [Tile]) -> &mut Tile {
    let len = arr.len();
    let mut the_chosen_i = 0;
    loop {
        let i = rand::thread_rng().gen_range(0..len);
        let tile = &mut arr[i];
        if Tile::Empty == *tile {
            the_chosen_i = i;
            break;
        }
    }
    &mut arr[the_chosen_i]
}
```

在上面代码中，我们只在循环中保留一个可变引用，剩下的 `arr.len` 和返回值引用，都移到循环外面，顺利通过编译。

## 一个更复杂的例子

再来看一个例子，代码会更复杂，但是原因几乎相同：

```

use std::collections::HashMap;

enum Symbol {
    A,
}

pub struct SymbolTable {
    scopes: Vec<Scope>,
    current: usize,
}

struct Scope {
    parent: Option<usize>,
    symbols: HashMap<String, Symbol>,
}

impl SymbolTable {
    pub fn get_mut(&mut self, name: &String) -> &mut Symbol {
        let mut current = Some(self.current);

        while let Some(id) = current {
            let scope = self.scopes.get_mut(id).unwrap();
            if let Some(symbol) = scope.symbols.get_mut(name) {
                return symbol;
            }

            current = scope.parent;
        }

        panic!("Value not found: {}", name);
    }
}

```

运行后报错如下：

```

error[E0499]: cannot borrow `self.scopes` as mutable more than once at a time
--> src/main.rs:22:25
|
18 |     pub fn get_mut(&mut self, name: &String) -> &mut Symbol {
|         - let's call the lifetime of this reference ''1``
...
22 |             let scope = self.scopes.get_mut(id).unwrap();
|                 ^^^^^^^^^^ `self.scopes` was mutably borrowed here in
the previous iteration of the loop
23 |                 if let Some(symbol) = scope.symbols.get_mut(name) {
24 |                     return symbol;
|                         ----- returning this value requires that `self.scopes` is borrowed for ''1``

```

对于上述代码，只需要将返回值修改下，即可通过编译：

```
fn get_mut(&mut self, name: &String) -> &mut Symbol {
    let mut current = Some(self.current);

    while let Some(id) = current {
        let scope = self.scopes.get_mut(id).unwrap();
        if scope.symbols.contains_key(name) {
            return self.scopes.get_mut(id).unwrap().symbols.get_mut(name).unwrap();
        }

        current = scope.parent;
    }

    panic!("Value not found: {}", name);
}
```

其中的关键就在于返回的时候，新建一个引用，而不是使用中间状态的引用。

## 新编译器 Polonius

针对现有编译器存在的各种问题，Rust 团队正在研发一个全新的编译器，名曰 `polonius`，但是目前它仍然处在开发阶段，如果想在自己项目中使用，需要在 `rustc/RUSTFLAGS` 中增加标志 `-Zpolonius`，但是可能会导致编译速度变慢，或者引入一些新的编译错误。

## 总结

编译器不是万能的，它也会迷茫，也会犯错。

因此我们在循环中使用引用类型时要格外小心，特别是涉及可变引用，这种情况下，最好的办法就是避免中间状态，或者在返回时避免使用中间状态。

# 当闭包碰到特征对象 1

特征对象是一个好东西，闭包也是一个好东西，但是如果两者你都想要时，可能就会火星撞地球，boom！至于这两者为何会勾搭到一起？考虑一个常用场景：使用闭包作为回调函数。

## 学习目标

如何使用闭包作为特征对象，并解决以下错误：`the parameter type `impl Fn(&str) -> Res` may not live long enough`

## 报错的代码

在下面代码中，我们通过闭包实现了一个简单的回调函数(错误代码已经标注)：

```

pub struct Res<'a> {
    value: &'a str,
}

impl<'a> Res<'a> {
    pub fn new(value: &str) -> Res {
        Res { value }
    }
}

pub struct Container<'a> {
    name: &'a str,
    callback: Option<Box<dyn Fn(&str) -> Res>>,
}

impl<'a> Container<'a> {
    pub fn new(name: &str) -> Container {
        Container {
            name,
            callback: None,
        }
    }
}

pub fn set(&mut self, cb: impl Fn(&str) -> Res) {
    self.callback = Some(Box::new(cb));
}
}

fn main() {
    let mut inl = Container::new("Inline");

    inl.set(|val| {
        println!("Inline: {}", val);
        Res::new("inline")
    });

    if let Some(cb) = inl.callback {
        cb("hello, world");
    }
}
}

error[E0310]: the parameter type `impl Fn(&str) -> Res` may not live long enough
--> src/main.rs:25:30
|
24 |     pub fn set(&mut self, cb: impl Fn(&str) -> Res) {
|                         ----- help: consider adding an
explicit lifetime bound...: `impl Fn(&str) -> Res + 'static`
25 |         self.callback = Some(Box::new(cb));
|                         ^^^^^^^^^^ ...so that the type `impl Fn(&str) ->
Res` will meet its required lifetime bounds

```

从第一感觉来说，报错属实不应该，因为我们连引用都没有用，生命周期都不涉及，怎么就报错了？在继续深入之前，先来观察下该闭包是如何被使用的：

```
callback: Option<Box<dyn Fn(&str) -> Res>>,
```

众所周知，闭包跟哈姆雷特一样，每一个都有[自己的类型](#)，因此我们无法通过类型标注的方式来声明一个闭包，那么只有一个办法，就是使用特征对象，因此上面代码中，通过 `Box<dyn Trait>` 的方式把闭包特征封装成一个特征对象。

## 深入挖掘报错原因

事出诡异必有妖，那接下来我们一起去会会这只妖。

### 特征对象的生命周期

首先编译器报错提示我们闭包活得不够久，那可以大胆推测，正因为使用了闭包作为特征对象，所以才活得不够久。因此首先需要调查下特征对象的生命周期。

首先给出结论：**特征对象隐式的具有 `'static` 生命周期**。

其实在 Rust 中，`'static` 生命周期很常见，例如一个没有引用字段的结构体它其实也是 `'static`。当 `'static` 用于一个类型时，该类型不能包含任何非 `'static` 引用字段，例如以下结构体：

```
struct Foo<'a> {
    x: &'a [u8]
};
```

除非 `x` 字段借用了 `'static` 的引用，否则 `'a` 肯定比 `'static` 要小，那么该结构体实例的生命周期肯定不是 `'static: 'a: 'static` 的限制不会被满足([HRTB](#))。

对于特征对象来说，它没有包含非 `'static` 的引用，因此它隐式的具有 `'static` 生命周期，`Box<dyn Trait>` 就跟 `Box<dyn Trait + 'static>` 是等价的。

### `'static` 闭包的限制

其实以上代码的错误很好解决，甚至编译器也提示了我们：

```
help: consider adding an explicit lifetime bound...: `impl Fn(&str) -> Res + 'static`
```

但是解决问题不是本文的目标，我们还是要继续深挖一下，如果闭包使用了 'static 会造成什么问题。

### 1. 无本地变量被捕获

```
inl.set(|val| {
    println!("Inline: {}", val);
    Res::new("inline")
});
```

以上代码只使用了闭包中传入的参数，并没有本地变量被捕获，因此 'static 闭包一切 OK。

### 2. 有本地变量被捕获

```
let local = "hello".to_string();

// 编译错误： 闭包不是'static!
inl.set(|val| {
    println!("Inline: {}", val);
    println!("{}", local);
    Res::new("inline")
});
```

这里我们在闭包中捕获了本地环境变量 local，因为 local 不是 'static，那么闭包也不再是 'static。

### 3. 将本地变量 move 进闭包

```
let local = "hello".to_string();

inl.set(move |val| {
    println!("Inline: {}", val);
    println!("{}", local);
    Res::new("inline")
});

// 编译错误： local已经被移动到闭包中，这里无法再被借用
// println!("{}", local);
```

如上所示，你也可以选择将本地变量的所有权 move 进闭包中，此时闭包再次具有 'static 生命周期

### 4. 非要捕获本地变量的引用？

对于第 2 种情况，如果非要这么干，那 'static 肯定是没办法了，我们只能给予闭包一个新的生命周期：

```

pub struct Container<'a, 'b> {
    name: &'a str,
    callback: Option<Box<dyn Fn(&str) -> Res + 'b>>,
}

impl<'a, 'b> Container<'a, 'b> {
    pub fn new(name: &str) -> Container {
        Container {
            name,
            callback: None,
        }
    }

    pub fn set(&mut self, cb: impl Fn(&str) -> Res + 'b) {
        self.callback = Some(Box::new(cb));
    }
}

```

肉眼可见，代码复杂度哐哐提升，不得不说 `'static` 真香！

友情提示：由此修改引发的一系列错误，需要你自行修复：）（再次友情小提示，可以考虑把 `main` 中的 `local` 变量声明位置挪到 `inl` 声明位置之前）

## 姗姗来迟的正确代码

其实，大家应该都知道该如何修改了，不过出于严谨，我们还是继续给出完整的正确代码：

```
pub fn set(&mut self, cb: impl Fn(&str) -> Res + 'static) {
```

可能大家觉得我重新定义了 `完整` 两个字，其实是我不想水篇幅：）

## 总结

闭包和特征对象的相爱相杀主要原因就在于特征对象默认具备 `'static` 的生命周期，同时我们还对什么样的类型具备 `'static` 进行了简单的分析。

同时，如果一个闭包拥有 `'static` 生命周期，那闭包无法通过引用的方式来捕获本地环境中的变量。如果你想要非要捕获，只能使用非 `'static`。

# 重复借用

本章讲述如何解决类似 `cannot borrow *self as mutable because it is also borrowed as immutable` 这种重复借用的错误。

# 同时在函数内外使用引用导致的重复借用错误

本文将彻底解决一个困扰广大 Rust 用户已久的常见错误：因为在函数内外同时借用一个引用，导致了重复借用错误 `cannot borrow *self as mutable because it is also borrowed as immutable.`

---

本文大部分内容节选自[Rust 常见陷阱专题](#)，由于借用是新手绕不过去的坎，因此将其提取出来形成一个新的系列

---

## 正确的代码

```
struct Test {
    a : u32,
    b : u32
}

impl Test {
    fn increase(&mut self) {
        let mut a = &mut self.a;
        let mut b = &mut self.b;
        *b += 1;
        *a += 1;
    }
}
```

这段代码是可以正常编译的，也许有读者会有疑问，`self` 在这里被两个变量以可变的方式借用了，明明违反了 Rust 的所有权规则，为何它不会报错？

答案要从很久很久之前开始(啊哒~~~由于我太啰嗦，被正义群众来了一下，那咱现在开始长话短说，直接进入主题)。

### 正确代码为何不报错？

虽然从表面来看，`a` 和 `b` 都可变引用了 `self`，但是 Rust 的编译器很多时候都足够聪明，它发现我们其实仅仅引用了同一个结构体中的不同字段，因此完全可以将其的借用权分离开来。

因此，虽然我们不能同时对整个结构体进行可变引用，但是我们可以分别对结构体中的不同字段进行可变引用，当然，一个字段至多也只能存在一个可变引用，这个最基本的所有权规则还是不能违反的。变量 `a` 引用结构体字段 `a`，变量 `b` 引用结构体字段 `b`，从底层来说，这种方式也不会造成两个可变引用指向了同一块内存。

至此，正确代码我们已经挖掘完毕，再来看看重构后的错误代码。

## 重构后的错误代码

```
struct Test {
    a : u32,
    b : u32
}

impl Test {

    fn increase_a (&mut self) {
        self.a += 1;
    }

    fn increase(&mut self) {
        let b = &mut self.b;
        self.increase_a();
        *b += 1;
    }
}
```

果然不正义的代码就是不好看，但是邪恶的它更强了吗？

```
error[E0499]: cannot borrow `*self` as mutable more than once at a time
--> src/main.rs:14:9
|
13 |     let b = &mut self.b;
|           ----- first mutable borrow occurs here
14 |     self.increase_a();
|           ^^^^^ second mutable borrow occurs here
15 |     *b += 1;
|           ----- first borrow later used here
```

嗯，最开始提到的错误，它终于出现了。

## 大聪明编译器

为什么？明明之前还是正确的代码，就因为放入函数中就报错了？我们先从一个简单的理解谈起，当然这个理解也是浮于表面的，等会会深入分析真实的原因。

之前讲到 Rust 编译器挺聪明，可以识别到引用到不同的结构体字段，因此不会报错。但是现在这种情况下，编译器又不够聪明了，一旦放入函数中，编译器将无法理解我们对 `self` 的使用：它仅仅用到了一个字段，而不是整个结构体。

因此它会简单的认为，这个结构体作为一个整体被可变借用了，产生两个可变引用，一个引用整个结构体，一个引用了结构体字段 `b`，这两个引用存在重叠的部分，最终导致编译错误。

## 被冤枉的编译器

在工作生活中，我们无法理解甚至错误的理解一件事，有时是因为层次不够导致的。同样，对于本文来说，也是因为我们对编译器的所知不够，才冤枉了它，还给它起了一个屈辱的“大聪明”外号。

### 深入分析

---

如果只改变相关函数的实现而不改变它的签名，那么不会影响编译的结果

---

何为相关函数？当函数 `a` 调用了函数 `b`，那么 `b` 就是 `a` 的相关函数。

上面这句是一条非常重要的编译准则，意思是，对于编译器来说，只要函数签名没有变，那么任何函数实现的修改都不会影响已有的编译结果(前提是函数实现没有错误-，-)。

以前面的代码为例：

```
fn increase_a (&mut self) {
    self.a += 1;
}

fn increase(&mut self) {
    let b = &mut self.b;
    self.increase_a();
    *b += 1;
}
```

虽然 `increase_a` 在函数实现中没有访问 `self.b` 字段，但是它的签名允许它访问 `b`，因此违背了借用规则。事实上，该函数有没有访问 `b` 不重要，**因为编译器在这里只关心签名，签名存在可能性，那么就立刻报出错误。**

为何会有这种编译器行为，主要有两个原因：

1. 一般来说，我们希望编译器有能力独立的编译每个函数，而无需深入到相关函数的内部实现，因为这样做会带来快得多的编译速度。

2. 如果没有这种保证，那么在实际项目开发中，我们会特别容易遇到各种错误。假设我们要求编译器不仅仅关注相关函数的签名，还要深入其内部关注实现，那么由于 Rust 严苛的编译规则，当你修改了某个函数内部实现的代码后，可能会引起使用该函数的其它函数的各种错误！对于大型项目来说，这几乎是不可接受的！

然后，我们的借用类型这么简单，编译器有没有可能针对这种场景，在现有的借用规则之外增加特殊规则？答案是否定的，由于 Rust 语言的设计哲学：特殊规则的加入需要慎之又慎，而我们的这种情况其实还蛮好解决的，因此编译器不会为此新增规则。

## 解决办法

在深入分析中，我们提到一条重要的规则，要影响编译行为，就需要更改相关函数的签名，因此可以修改 `increase_a` 的签名：

```
fn increase_a (a :&mut u32) {
    *a += 1;
}

fn increase(&mut self) {
    let b = &mut self.b;
    Test::increase_a(&mut self.a);
    *b += 1;
}
```

此时，`increase_a` 这个相关函数，不再使用 `&mut self` 作为签名，而是获取了结构体中的字段 `a`，此时编译器又可以清晰的知道：函数 `increase_a` 和变量 `b` 分别引用了结构体中的不同字段，因此可以编译通过。

当然，除了修改相关函数的签名，你还可以修改调用者的实现：

```
fn increase(&mut self) {
    self.increase_a();
    self.b += 1;
}
```

在这里，我们不再单独声明变量 `b`，而是直接调用 `self.b+=1` 进行递增，根据借用生命周期NLL的规则，第一个可变借用 `self.increase_a()` 的生命周期随着方法调用的结束而结束，那么就不会影响 `self.b += 1` 中的借用。

## CPU 模拟例子

我们再来看一个例子：

```
use std::collections::HashMap;

struct Cpu {
    pc: u16,
    cycles: u32,
    opcodes: HashMap<u8, Opcode>,
}

struct Opcode {
    size: u16,
    cycles: u32,
}

impl Cpu {
    fn new() -> Cpu {
        Cpu {
            pc: 0,
            cycles: 0,
            opcodes: HashMap::from([
                (0x00, Opcode::new(1, 7)),
                (0x01, Opcode::new(2, 6))
            ]),
        }
    }

    fn tick(&mut self) {
        let address = self.pc as u8;
        let opcode = &self.opcodes[&address];

        step(&mut self, opcode);
    }
}

fn step(cpu : &mut Cpu, opcode: &Opcode) {

}

impl Opcode {
    fn new(size: u16, cycles: u32) -> Opcode {
        Opcode { size, cycles }
    }
}

fn main() {
    let mut cpu = Cpu::new();
    cpu.tick();
}
```

## **总结**

知其然知其所以然，要彻底解决借用导致的编译错误，我们就必须深入了解其原理，心中有剑则手中无“贱”。

上面的例子就留给读者朋友自己去解决，相信你以后在遇到这种常见问题时，会更加游刃有余。

# 智能指针引起的重复借用错误

本文将彻底解决一个困扰广大 Rust 用户已久的常见错误：当智能指针和结构体一起使用时导致的借用错误：`cannot borrow mut_s as mutable because it is also borrowed as immutable.`

相信看过[《对抗 Rust 编译检查系列》](#)的读者都知道结构体中的不同字段可以独立借用吧？

## 结构体中的字段借用

不知道也没关系，我们这里再简单回顾一下：

```
struct Test {
    a : u32,
    b : u32
}

impl Test {
    fn increase(&mut self) {
        let mut a = &mut self.a;
        let mut b = &mut self.b;
        *b += 1;
        *a += 1;
    }
}
```

这段代码看上去像是重复借用了 `&mut self`，违反了 Rust 的借用规则，实际上在聪明的 Rust 编译器面前，这都不是事。它能发现我们其实借用了目标结构体的不同字段，因此完全可以将其借用权分离开来。

因此，虽然我们不能同时对整个结构体进行多次可变借用，但是我们可以分别对结构体中的不同字段进行可变借用，当然，一个字段至多也只能存在一个可变借用，这个最基本的所有权规则还是不能违反的。变量 `a` 引用结构体字段 `a`，变量 `b` 引用结构体字段 `b`，从底层来说，这种方式也不会造成两个可变引用指向了同一块内存。

## RefCell

如果你还不知道 RefCell，可以看看[这篇文章](#)，当然不看也行，简而言之，RefCell 能够实现：

- 将借用规则从编译期推迟到运行期，但是并不会绕过借用规则，当不符合时，程序直接 panic
- 实现内部可变性：简单来说，对一个不可变的值进行可变借用，然后修改内部的值

## 被 RefCell 包裹的结构体

既然了解了结构体的借用规则和 RefCell，我们来看一段结合了两者的代码：

```
use std::cell::RefCell;
use std::io::Write;

struct Data {
    string: String,
}

struct S {
    data: Data,
    writer: Vec<u8>,
}

fn write(s: RefCell<S>) {
    let mut mut_s = s.borrow_mut();
    let str = &mut_s.data.string;
    mut_s.writer.write(str.as_bytes());
}
```

以上代码从 s 中可变借用出结构体 s，随后又对结构体中的两个字段进行了分别借用，按照之前的规则这段代码应该顺利通过编译：

```
error[E0502]: cannot borrow `mut_s` as mutable because it is also borrowed as
immutable
--> src/main.rs:16:5
|
15 |     let str = &mut_s.data.string;
|         ----- immutable borrow occurs here
16 |     mut_s.writer.write(str.as_bytes());
|     ^^^^^^           --- immutable borrow later used here
|
|     mutable borrow occurs here
```

只能说，还好它报错了，否则本篇文章已经可以结束。。。错误很简单，首先对结构体 s 的 data 字段进行了不可变借用，其次又对 writer 字段进行了可变借用，这个符合之前的规则：对结构体不同字段分开借用，为何报错了？

## 深入分析

第一感觉，问题是出在 borrow\_mut 方法返回的类型上，先来看看：

```
pub fn borrow_mut(&self) -> RefMut<'_, T>
```

可以看出，该方法并没有直接返回我们的结构体，而是一个 `RefMut` 类型，而要使用该类型，需要经过编译器为我们做一次隐式的 `Deref` 转换，编译器展开后的代码大概如下：

```
use std::cell::RefMut;
use std::ops::{Deref, DerefMut};

fn write(s: RefCell<S>) {
    let mut mut_s: RefMut<S> = s.borrow_mut();
    let str = &Deref::deref(&mut_s).data.string;
    DerefMut::deref_mut(&mut mut_s).writer.write(str.as_bytes());
}
```

可以看出，对结构体字段的调用，实际上经过一层函数，一层函数！？我相信你应该想起了什么，是的，在[上一篇文章](#)中讲过类似的问题，大意就是**编译器对于函数往往只会分析签名，并不关心内部到底如何使用结构体。**

而上面的 `&Deref::deref(&mut_s)` 和 `DerefMut::deref_mut(&mut mut_s)` 函数，签名全部使用的是结构体，并不是结构体中的某一个字段，因此对于编译器来说，该结构体明显是被重复借用了！

## 解决方法

因此要解决这个问题，我们得把之前的展开形式中的 `Deref::deref` 消除掉，这样没有了函数签名，编译器也将不再懒政。

既然两次 `Deref::deref` 调用都是对智能指针的自动 `Deref`，那么可以提前手动的把它 `Deref` 了，只做一次！

```
fn write(s: RefCell<S>) {
    let mut mut_s = s.borrow_mut();
    let mut tmp = &mut *mut_s; // Here
    let str = &tmp.data.string;
    tmp.writer.write(str.as_bytes());
}
```

以上代码通过 `*` 对 `mut_s` 进行了解引用，获得结构体，然后又对结构体进行了可变借用 `&mut`，最终赋予 `tmp` 变量，那么该变量就持有我们的结构体的可变引用，而不再是持有一个智能指针。

此后对 `tmp` 的使用就回归到文章开头的那段代码：分别借用结构体的不同字段，成功通过编译！

## 展开代码

我们再来模拟编译器对正确的代码进行一次展开:

```
use std::cell::RefMut;
use std::ops::DerefMut;

fn write(s: RefCell<S>) {
    let mut mut_s: RefMut<S> = s.borrow_mut();
    let tmp: &mut S = DerefMut::deref_mut(&mut mut_s);
    let str = &tmp.data.string;
    tmp.writer.write(str.as_bytes());
}
```

可以看出，此时对结构体的使用不再有 `DerefMut::deref` 的身影，我们成功消除了函数边界对编译器的影响！

## 不仅仅是 RefCell

事实上，除了 RefCell 外，还有不少会导致这种问题的智能指针，当然原理都是互通的，我们这里就不再进行——深入讲解，只简单列举下：

- Box
- MutexGuard (来源于 Mutex)
- PeekMut (来源于 BinaryHeap)
- RwLockWriteGuard (来源于 RwLock)
- String
- Vec
- Pin

## 一个练习

下面再来一个练习巩固一下，强烈建议大家按照文章的思路进行分析和解决：

```
use std::rc::Rc;
use std::cell::RefCell;

pub struct Foo {
    pub foo1: Vec<bool>,
    pub foo2: Vec<i32>,
}

fn main() {
    let foo_cell = Rc::new(RefCell::new(Foo {
        foo1: vec![true, false],
        foo2: vec![1, 2]
    }));

    let borrow = foo_cell.borrow_mut();
    let foo1 = &borrow.foo1;
    // 下面代码会报错,因为`foo1`和`foo2`发生了重复借用
    borrow.foo2.iter_mut().enumerate().for_each(|(idx, foo2)| {
        if foo1[idx] {
            *foo2 *= -1;
        }
    });
}
```

## 总结

当结构体的引用穿越函数边界时，我们要格外小心，因为编译器只会对函数签名进行检查，并不关心内部到底用了结构体的那个字段，当签名都使用了结构体时，会立即报错。

而智能指针由于隐式解引用 `Deref` 的存在，导致了两次 `Deref` 时都让结构体穿越了函数边界 `Deref::deref`，结果造成了重复借用的错误。

解决办法就是提前对智能指针进行手动解引用，然后对内部的值进行借用后，再行使用。

**类型未限制**

**幽灵数据**

# Rust 陷阱系列

本章收录一些 Rust 常见的陷阱，一不小心就会坑你的那种(当然，这不是 Rust 语言的问题，而是一些边边角角的知识点)。

# for 循环中使用外部数组

一般来说，`for` 循环能做到的，`while` 也可以，反之亦然，但是有一种情况，还真不行，先来看代码：

```
let mut v = vec![1, 2, 3];

for i in 0..v.len() {
    v.push(i);
    println!("{}: {}", v);
}
```

我们的目的是创建一个无限增长的数组，往里面插入 `0..` (看不懂该表达式同学请查阅[流程控制](#))的数值序列。

看起来上面代码可以完成，因为随着数组不断增长，`v.len()` 也会不断变大，但是事实上真的如此吗？

```
[1, 2, 3, 0]
[1, 2, 3, 0, 1]
[1, 2, 3, 0, 1, 2]
```

输出很清晰的表明，只新插入了三个元素：`0..=2`，刚好是 `v` 的初始长度。

这是因为：**在 for 循环中，`v.len()` 只会在循环伊始之时进行求值，之后就一直使用该值。**

行，问题算是清楚了，那该如何解决呢，我们可以使用 `while` 循环，该循环与 `for` 相反，每次都会重新求值：

```
let mut v = vec![1, 2, 3];

let mut i = 0;
while i < v.len() {
    v.push(i);
    i+=1;
    println!("{}: {}", v);
}
```

友情提示，在你运行上述代码时，千万要及时停止，否则会 Boom - 炸翻控制台。

# 线程类型导致的栈溢出

在 Rust 中，我们不太容易遇到栈溢出，因为默认栈还挺大的，而且大的数据往往存在堆上(动态增长)，但是一旦遇到该如何处理？先来看段代码：

```
#![feature(test)]
extern crate test;

#[cfg(test)]
mod tests {
    use test::Bencher;

    #[bench]
    fn it_works(b: &mut Bencher) {
        b.iter(|| { let stack = [[0.0; 2]; 512]; 512]; });
    }
}
```

以上代码是一个测试模块，它在堆上生成了一个数组 `stack`，初步看起来数组挺大的，先尝试运行下 `cargo test`：

---

你很可能会遇到 `#![feature(test)]` 错误，因为该特性目前只存在 Rust Nightly 版本上，具体解决方法见[Rust 语言圣经](#)

```
running 1 test

thread 'tests::it_works' has overflowed its stack
fatal runtime error: stack overflow
```

Bang，很不幸，遇到了百年一遇的栈溢出错误，再来试试 `cargo bench`，竟然通过了测试，这是什么原因？为何 `cargo test` 和 `cargo bench` 拥有完全不同的行为？这就要从 Rust 的栈原理讲起。

首先看看 `stack` 数组，它的大小是  $8 \times 2 \times 512 \times 512 = 4 \text{ MiB}$ ，嗯，很大，崩溃也正常(读者说，正常，只是作者你不太正常。。)。

其次，`cargo test` 和 `cargo bench`，前者运行在一个新创建的线程上，而后者运行在**main 线程上**。

最后，`main` 线程由于是老大，所以资源比较多，拥有令其它兄弟艳羡不已的 8MB 栈大小，而其它新线程只有区区 2MB 栈大小(取决于操作系统, linux 是 2MB, 其它的可能更小)，再对比我们的 `stack` 大小，不崩溃就奇怪了。

因此，你现在明白，为何 `cargo test` 不能运行，而 `cargo bench` 却可以欢快运行。

如果实在想要增大栈的默认大小，以通过该测试，你可以这样运行: `RUST_MIN_STACK=8388608 cargo test`,结果如下:

```
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

Bingo, 成功了,最后再补充点测试的背景知识:

---

`cargo test` 为何使用新线程? 因为它需要并行的运行测试用例, 与之相反, `cargo bench` 只需要顺序的执行, 因此 main 线程足矣

---

# 算术溢出导致的 panic

在 Rust 中，溢出后的数值被截断是很正常的：

```
let x: u16 = 65535;
let v = x as u8;
println!("{}", v)
```

最终程序会输出 255，因此大家可能会下意识地就觉得算数操作在 Rust 中只会导致结果的不正确，并不会导致异常。但是实际上，如果是因为算术操作符导致的溢出，就会让整个程序 panic：

```
fn main() {
    let x: u8 = 10;

    let v = x + u8::MAX;
    println!("{}", v)
}
```

输出结果如下：

```
thread 'main' panicked at 'attempt to add with overflow', src/main.rs:5:13
```

那么当我们确实有这种需求时，该如何做呢？可以使用 Rust 提供的 `checked_xxx` 系列方法：

```
fn main() {
    let x: u8 = 10;

    let v = x.checked_add(u8::MAX).unwrap_or(0);
    println!("{}", v)
}
```

也许你会觉得本章内容其实算不上什么陷阱，但是在实际项目快速迭代中，越是不起眼的地方越是容易出错：

```
fn main() {
    let v = production_rate_per_hour(5);
    println!("{}", v);
}

pub fn production_rate_per_hour(speed: u8) -> f64 {
    let cph: u8 = 221;
    match speed {
        1..=4 => (speed * cph) as f64,
        5..=8 => (speed * cph) as f64 * 0.9,
        9..=10 => (speed * cph) as f64 * 0.77,
        _ => 0 as f64,
    }
}

pub fn working_items_per_minute(speed: u8) -> u32 {
    (production_rate_per_hour(speed) / 60 as f64) as u32
}
```

上述代码中，`speed * cph` 就会直接 panic:

```
thread 'main' panicked at 'attempt to multiply with overflow', src/main.rs:10:18
```

是不是还藏的挺隐蔽的？因此大家在 Rust 中做数学运算时，要多留一个心眼，免得上了生产才发现问题所在。或者，你也可以做好单元测试：）

# 闭包上奇怪的生命周期

Rust 一道独特的靓丽风景就是生命周期，也是反复折磨新手的最大黑手，就连老手，可能一不注意就会遇到一些生命周期上的陷阱，例如闭包上使用引用。

## 一段简单的代码

先来看一段简单的代码：

```
fn fn_elision(x: &i32) -> &i32 { x }
let closure_elision = |x: &i32| -> &i32 { x };
```

乍一看，这段代码比古天乐还平平无奇，能有什么问题呢？来，走两圈试试：

```
error: lifetime may not live long enough
--> src/main.rs:39:39
|
39 |     let closure = |x: &i32| -> &i32 { x }; // fails
|           -           -           ^ returning this value requires that ``1``
must outlive ``2``
|           |
|           |           let's call the lifetime of this reference ``2``
|           let's call the lifetime of this reference ``1``
```

咦？竟然报错了，明明两个一模一样功能的函数，一个正常编译，一个却报错，错误原因是编译器无法推断返回的引用和传入的引用谁活得更久！

真的是非常奇怪的错误，学过Rust 生命周期的读者应该都记得这样一条生命周期消除规则：**如果函数参数中只有一个引用类型，那该引用的生命周期会被自动分配给所有的返回引用**。我们当前的情况完美符合，`fn_elision` 函数的顺利编译通过，就充分说明了问题。

那为何闭包就出问题了？

## 一段复杂的代码

为了验证闭包无法应用生命周期消除规则，再来看一个复杂一些的例子：

```

use std::marker::PhantomData;

trait Parser<'a>: Sized + Copy {
    fn parse(&self, tail: &'a str) -> &'a str {
        tail
    }
    fn wrap(self) -> Wrapper<'a, Self> {
        Wrapper {
            parser: self,
            marker: PhantomData,
        }
    }
}

#[derive(Copy, Clone)]
struct T<'x> {
    int: &'x i32,
}

impl<'a, 'x> Parser<'a> for T<'x> {}

struct Wrapper<'a, P>
where
    P: Parser<'a>,
{
    parser: P,
    marker: PhantomData<&'a ()>,
}

fn main() {
    // Error.
    let closure_wrap = |parser: T| parser.wrap();

    // No error.
    fn parser_wrap(parser: T<'_>) -> Wrapper<'_, T<'_>> {
        parser.wrap()
    }
}

```

该例子之所以这么复杂，纯粹是为了证明闭包上生命周期会失效，读者大大轻拍:) 编译后，不出所料的报错了：

```
error: lifetime may not live long enough
--> src/main.rs:32:36
|
32 |     let closure_wrap = |parser: T| parser.wrap();
|           ----- - ^^^^^^^^^^^^^ returning this value requires
that `'1` must outlive `'2`
|           |
|           return type of closure is Wrapper<'_, T<'2>>
|           has type `T<'1>`
```

## 深入调查

一模一样的报错，说明在这种情况下，生命周期的消除规则也没有生效，看来事情确实不简单，我眉头一皱，决定深入调查，最后还真翻到了一些讨论，经过整理后，大概分享给大家。

首先给出一个结论：**这个问题，可能很难被解决，建议大家遇到后，还是老老实实用正常的函数，不要秀闭包了。**

对于函数的生命周期而言，它的消除规则之所以能生效是因为它的生命周期完全体现在签名的引用类型上，在函数体中无需任何体现：

```
fn fn_elision(x: &i32) -> &i32 { .. }
```

因此编译器可以做各种编译优化，也很容易根据参数和返回值进行生命周期的分析，最终得出消除规则。

可是闭包，并没有函数那么简单，它的生命周期分散在参数和闭包函数体中(主要是它没有确切的返回值签名)：

```
let closure_slision = |x: &i32| -> &i32 { x };
```

编译器就必须深入到闭包函数体中，去分析和推测生命周期，复杂度因此极具提升：试想一下，编译器该如何从复杂的上下文中分析出参数引用的生命周期和闭包体中生命周期的关系？

由于上述原因(当然，实际情况复杂的多)，Rust 语言开发者其实目前是有意为之，针对函数和闭包实现了两种不同的生命周期消除规则。

## 总结

虽然我言之凿凿，闭包的生命周期无法解决，但是未来谁又知道呢。最大的可能性就是之前开头那种简单的场景，可以被自动识别和消除。

总之，如果有这种需求，还是像古天乐一样做一个平平无奇的男人，老老实实使用函数吧。

# 失效的可变性

众所周知 Rust 是一门安全性非常强的系统级语言，其中，显式的设置变量可变性，是安全性的重要组成部分。按理来说，变量可变不可变在设置时就已经决定了，但是你遇到过可变变量在某些情况失效，变成不可变吗？

先来看段正确的代码：

```
#[derive(Debug)]
struct A {
    f1: u32,
    f2: u32,
    f3: u32
}

#[derive(Debug)]
struct B<'a> {
    f1: u32,
    a: &'a mut A,
}

fn main() {
    let mut a: A = A{ f1: 0, f2: 1, f3: 2 };
    // b不可变
    let b: B = B{ f1: 3, a: &mut a };
    // 但是b中的字段a可以变
    b.a.f1 += 1;

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

在这里，虽然变量 `b` 被设置为不可变，但是 `b` 的其中一个字段 `a` 被设置为可变的结构体，因此我们可以通过 `b.a.f1 += 1` 来修改 `a` 的值。

也许有人还不知道这种部分可变性的存在，不过没关系，因为马上就不可变了：）

- 结构体可变时，里面的字段都是可变的，例如 `&mut a`
- 结构体不可变时，里面的某个字段可以单独设置为可变，例如 `b.a`

在理解了上面两条简单规则后，来看看下面这段代码：

```

#[derive(Debug)]
struct A {
    f1: u32,
    f2: u32,
    f3: u32
}

#[derive(Debug)]
struct B<'a> {
    f1: u32,
    a: &'a mut A,
}

impl B<'_> {
    // this will not work
    pub fn changeme(&self) {
        self.a.f1 += 1;
    }
}

fn main() {
    let mut a: A = A{ f1: 0, f2: 1, f3: 2 };
    // b is immutable
    let b: B = B{ f1: 3, a: &mut a };
    b.changeme();

    println!("b is {:?}", &b);
}

```

这段代码，仅仅做了一个小改变，不再直接修改 `b.a`，而是通过调用 `b` 上的方法去修改其中的 `a`，按理说不会有任何区别。因此我预言：通过方法调用跟直接调用不应该有任何区别，运行验证下：

```

error[E0594]: cannot assign to `self.a.f1`, which is behind a `&` reference
--> src/main.rs:18:9
|
17 |     pub fn changeme(&self) {
|         ----- help: consider changing this to be a mutable
reference: `&mut self`
18 |         self.a.f1 += 1;
|             ^^^^^^^^^^ `self` is a `&` reference, so the data it refers to
cannot be written

```

啪，又被打脸了。我说我是大意了，没有闪，大家信不？反正马先生应该是信的:D

## 简单分析

观察第一个例子，我们调用的 `b.a` 实际上是用 `b` 的值直接调用的，在这种情况下，由于所有权规则，编译器可以认定，只有一个可变引用指向了 `a`，因此这种使用是非常安全的。

但是，在第二个例子中，`b` 被藏在了 `&` 后面，根据所有权规则，同时可能存在多个 `b` 的借用，那么就意味着可能会存在多个可变引用指向 `a`，因此编译器就拒绝了这段代码。

事实上如果你将第一段代码的调用改成：

```
let b: &B = &B{ f1: 3, a: &mut a };
b.a.f1 += 1;
```

一样会报错！

## 一个练习

结束之前再来一个练习，稍微有点绕，大家品味品味：

```
#[derive(Debug)]
struct A {
    f1: u32,
    f2: u32,
    f3: u32
}

#[derive(Debug)]
struct B<'a> {
    f1: u32,
    a: &'a mut A,
}

fn main() {
    let mut a: A = A{ f1: 0, f2: 1, f3: 2 };
    let b: B = B{ f1: 3, a: &mut a };
    b.a.f1 += 1;
    a.f1 = 10;

    println!("b is {:?}", &b);
}
```

小提示：这里 `b.a.f1 += 1` 和 `a.f1 = 10` 只能有一个存在，否则就会报错。

## 总结

根据之前的观察和上面的小提示，可以得出一个结论：**可变性的真正含义是你对目标对象的独占修改权。**在实际项目中，偶尔会遇到比上述代码更复杂的可变性情况，记住这个结论，有助于我们拨云见日，直达本质。

学习，就是不断接近和认识事物本质的过程，对于 Rust 语言的学习亦是如此。

# 代码重构导致的可变借用错误

相信大家都听说过**重构一时爽，一直重构一直爽**的说法，私以为这种说法是很有道理的，不然技术团队绩效从何而来？但是，在 Rust 中，重构可能就不是那么爽快的事了，不信？咱们来看看。

## 欣赏下报错

很多时候，错误也是一种美，但是当这种错误每天都能见到时(呕)：

```
error[E0499]: cannot borrow `* self` as mutable more than once at a time;
```

虽然这一类错误长得一样，但是我这里的错误可能并不是大家常遇到的那些妖艳错误，废话不多说，一起来看看。

## 重构前的正确代码

```
struct Test {
    a : u32,
    b : u32
}

impl Test {
    fn increase(&mut self) {
        let mut a = &mut self.a;
        let mut b = &mut self.b;
        *b += 1;
        *a += 1;
    }
}
```

这段代码是可以正常编译的，也许有读者会有疑问，`self` 在这里被两个变量以可变的方式借用了，明明违反了 Rust 的所有权规则，为何它不会报错？

答案要从很久很久之前开始(啊哒~~~由于我太啰嗦，被正义群众来了一下，那咱现在开始长话短说，直接进入主题)。

## 正确代码为何不报错？

虽然从表面来看，`a` 和 `b` 都可变引用了 `self`，但是 Rust 的编译器在很多时候都足够聪明，它发现我们其实仅仅引用了同一个结构体中的不同字段，因此完全可以将其的借用权分离开来。

因此，虽然我们不能同时对整个结构体进行可变引用，但是我们可以分别对结构体中的不同字段进行可变引用，当然，一个字段至多也只能存在一个可变引用，这个最基本的所有权规则还是不能违反的。变量 `a` 引用结构体字段 `a`，变量 `b` 引用结构体字段 `b`，从底层来说，这种方式也不会造成两个可变引用指向了同一块内存。

至此，正确代码我们已经挖掘完毕，再来看看重构后的错误代码。

## 重构后的错误代码

由于领导说我们这个函数没办法复用，那就敷衍一下呗：

```
struct Test {
    a : u32,
    b : u32
}

impl Test {

    fn increase_a (&mut self) {
        self.a += 1;
    }

    fn increase(&mut self) {
        let b = &mut self.b;
        self.increase_a();
        *b += 1;
    }
}
```

既然领导说了，咱照做，反正他也没说怎么个复用法，咱就来个简单的，把 `a` 的递增部分复用下。

代码说实话。。。更丑了，但是更强了吗？

```
error[E0499]: cannot borrow `*self` as mutable more than once at a time
--> src/main.rs:14:9
13 |     let b = &mut self.b;
   |             ----- first mutable borrow occurs here
14 |     self.increase_a();
   |         ^^^^^ second mutable borrow occurs here
15 |     *b += 1;
   |             ----- first borrow later used here
```

嗯，最开始提到的错误，它终于出现了。

## 大聪明编译器

为什么？明明之前还是正确的代码，就因为放入函数中就报错了？我们先从一个简单的理解谈起，当然这个理解也是浮于表面的，等会会深入分析真实的原因。

之前讲到 Rust 编译器挺聪明，可以识别到引用到不同的结构体字段，因此不会报错。但是现在这种情况下，编译器又不够聪明了，一旦放入函数中，编译器将无法理解我们对 `self` 的使用：它仅仅用到了一个字段，而不是整个结构体。

因此它会简单的认为，这个结构体作为一个整体被可变借用了，产生两个可变引用，一个引用整个结构体，一个引用了结构体字段 `b`，这两个引用存在重叠的部分，最终导致编译错误。

## 被冤枉的编译器

在工作生活中，我们无法理解甚至错误的理解一件事，有时是因为层次不够导致的。同样，对于本文来说，也是因为我们对编译器的所知不够，才冤枉了它，还给它起了一个屈辱的“大聪明”外号。

### 深入分析

---

如果只改变相关函数的实现而不改变它的签名，那么不会影响编译的结果

---

何为相关函数？当函数 `a` 调用了函数 `b`，那么 `b` 就是 `a` 的相关函数。

上面这句是一条非常重要的编译准则，意思是，对于编译器来说，只要函数签名没有变，那么任何函数实现的修改都不会影响已有的编译结果(前提是函数实现没有错误-，-)。

以前面的代码为例：

```
fn increase_a (&mut self) {
    self.a += 1;
}

fn increase(&mut self) {
    let b = &mut self.b;
    self.increase_a();
    *b += 1;
}
```

虽然 `increase_a` 在函数实现中没有访问 `self.b` 字段，但是它的签名允许它访问 `b`，因此违背了借用规则。事实上，该函数有没有访问 `b` 不重要，**因为编译器在这里只关心签名，签名存在可能性，那么就立刻报出错误。**

为何会有这种编译器行为，主要有两个原因：

1. 一般来说，我们希望编译器有能力独立的编译每个函数，而无需深入到相关函数的内部实现，因为这样做会带来快得多的编译速度。
2. 如果没有这种保证，那么在实际项目开发中，我们会特别容易遇到各种错误。假设我们要求编译器不仅仅关注相关函数的签名，还要深入其内部关注实现，那么由于 Rust 严苛的编译规则，当你修改了某个函数内部实现的代码后，可能会引起使用该函数的其它函数的各种错误！对于大型项目来说，这几乎是不可接受的！

然后，我们的借用类型这么简单，编译器有没有可能针对这种场景，在现有的借用规则之外增加特殊规则？答案是否定的，由于 Rust 语言的设计哲学：特殊规则的加入需要慎之又慎，而我们的这种情况其实还蛮好解决的，因此编译器不会为此新增规则。

## 解决办法

在深入分析中，我们提到一条重要的规则，要影响编译行为，就需要更改相关函数的签名，因此可以修改 `increase_a` 的签名：

```
fn increase_a (a :&mut u32) {
    *a += 1;
}

fn increase(&mut self) {
    let b = &mut self.b;
    Test::increase_a(&mut self.a);
    *b += 1;
}
```

此时，`increase_a` 这个相关函数，不再使用 `&mut self` 作为签名，而是获取了结构体中的字段 `a`，此时编译器又可以清晰的知道：函数 `increase_a` 和变量 `b` 分别引用了结构体中的不同字段，因此可以编译通过。

当然，除了修改相关函数的签名，你还可以修改调用者的实现：

```
fn increase(&mut self) {  
    self.increase_a();  
    self.b += 1;  
}
```

在这里，我们不再单独声明变量 `b`，而是直接调用 `self.b+=1` 进行递增，根据借用生命周期NLL的规则，第一个可变借用 `self.increase_a()` 的生命周期随着方法调用的结束而结束，那么就不会影响 `self.b += 1` 中的借用。

## 闭包中的例子

再来看一个使用了闭包的例子：

```

use tokio::runtime::Runtime;

struct Server {
    number_of_connections : u64
}

impl Server {
    pub fn new() -> Self {
        Server { number_of_connections : 0}
    }

    pub fn increase_connections_count(&mut self) {
        self.number_of_connections += 1;
    }
}

struct ServerRuntime {
    runtime: Runtime,
    server: Server
}

impl ServerRuntime {
    pub fn new(runtime: Runtime, server: Server) -> Self {
        ServerRuntime { runtime, server }
    }

    pub fn increase_connections_count(&mut self) {
        self.runtime.block_on(async {
            self.server.increase_connections_count()
        })
    }
}

```

代码中使用了 `tokio`，在 `increase_connections_count` 函数中启动了一个异步任务，并且等待它的完成。这个函数中分别引用了 `self` 中的不同字段: `runtime` 和 `server`，但是可能因为闭包的原因，编译器没有像本文最开始的例子中那样聪明，并不能识别这两个引用仅仅引用了同一个结构体的不同部分，因此报错了：

```
error[E0501]: cannot borrow `self.runtime` as mutable because previous closure
requires unique access
--> the_little_things\src\main.rs:28:9
28     self.runtime.block_on(async {
^-----|
|           first borrow later used by call
|-----|
29     self.server.increase_connections_count()
----- first borrow occurs due to use of `self` in generator
30   })
----- second borrow occurs here
|-----|
generator construction occurs here
```

## 解决办法

解决办法很粗暴，既然编译器不能理解闭包中的引用是不同的，那么我们就主动告诉它：

```
pub fn increase_connections_count(&mut self) {
    let runtime = &mut self.runtime;
    let server = &mut self.server;
    runtime.block_on(async {
        server.increase_connections_count()
    })
}
```

上面通过变量声明的方式，在闭包外声明了两个变量分别引用结构体 `self` 的不同字段，这样一来，编译器就不会那么笨，编译顺利通过。

你也可以这么写：

```
pub fn increase_connections_count(&mut self) {
    let ServerRuntime { runtime, server } = self;
    runtime.block_on(async {
        server.increase_connections_count()
    })
}
```

当然，如果难以解决，还有一个笨办法，那就是将 `server` 和 `runtime` 分离开来，不要放在一个结构体中。

## **总结**

心中有剑，手中无剑，是武学至高境界。

本文列出的那条编译规则，在未来就将是大家心中的那把剑，当这些心剑招式足够多时，量变产生质变，终将天下无敌。

# 不太勤快的迭代器

迭代器，在 Rust 中是一个非常耀眼的存在，它光鲜亮丽，它让 Rust 大道至简，它备受用户的喜爱。可是，它也是懒惰的，不信？一起来看看。

## for 循环 vs 迭代器

在迭代器学习中，我们提到过迭代器在功能上可以替代循环，性能上略微优于循环(避免边界检查)，安全性上优于循环，因此在 Rust 中，迭代器往往都是更优的选择，前提是迭代器得发挥作用。

在下面代码中，分别是使用 for 循环和迭代器去生成一个 HashMap。

使用循环：

```
use std::collections::HashMap;
#[derive(Debug)]
struct Account {
    id: u32,
}

fn main() {
    let accounts = [Account { id: 1 }, Account { id: 2 }, Account { id: 3 }];

    let mut resolvers = HashMap::new();
    for a in accounts {
        resolvers.entry(a.id).or_insert(Vec::new()).push(a);
    }

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

使用迭代器：

```
let mut resolvers = HashMap::new();
accounts.into_iter().map(|a| {
    resolvers
        .entry(a.id)
        .or_insert(Vec::new())
        .push(a);
});
println!("{:?}", resolvers);
```

## 预料之外的结果

两端代码乍一看(很多时候我们快速浏览代码的时候，不会去细看)都很正常，运行下试试：

- `for` 循环很正常，输出 `{2: [Account { id: 2 }], 1: [Account { id: 1 }], 3: [Account { id: 3 }]}`
- 迭代器很。。。不正常，输出了一个 `{}`，黑人问号？？？

在继续深挖之前，我们先来简单回顾下迭代器。

## 回顾下迭代器

在迭代器章节中，我们曾经提到过，迭代器的[适配器](#)分为两种：消费者适配器和迭代器适配器，前者用来将一个迭代器变为指定的集合类型，往往通过 `collect` 实现；后者用于生成一个新的迭代器，例如上例中的 `map`。

还提到过非常重要的一点：**迭代器适配器都是懒惰的，只有配合消费者适配器使用时，才会进行求值。**

## 懒惰是根因

在我们之前的迭代器示例中，只有一个迭代器适配器 `map`：

```
accounts.into_iter().map(|a| {
    resolvers
        .entry(a.id)
        .or_insert(Vec::new())
        .push(a);
});
```

首先，`accounts` 被拿走所有权后转换成一个迭代器，其次该迭代器通过 `map` 方法生成一个新的迭代器，最后，在此过程中没有以类如 `collect` 的消费者适配器收尾。

因此在上述过程中，`map` 完全是懒惰的，它没有做任何事情，它在等一个消费者适配器告诉它：赶紧起床，任务可以开始了，它才会开始行动。

自然，我们的插值计划也失败了。

---

事实上，IDE 和编译器都会对这种代码给出警告：iterators are lazy and do nothing unless

consumed

---

## 解决办法

原因非常清晰，如果读者还有疑惑，建议深度了解下上面给出的迭代器链接，我们这里就不再赘述。

下面列出三种合理的解决办法：

1. 不再使用迭代器适配器 `map`，改成 `for_each`：

```
let mut resolvers = HashMap::new();
accounts.into_iter().for_each(|a| {
    resolvers
        .entry(a.id)
        .or_insert(Vec::new())
        .push(a);
});
```

但是，相关的文档也友善的提示了我们，除非作为链式调用的收尾，否则更建议使用 `for` 循环来处理这种情况。哎，忙忙碌碌，又回到了原点，不禁让人感叹：天道有轮回。

2. 使用消费者适配器 `collect` 来收尾，将 `map` 产生的迭代器收集成一个集合类型：

```
let resolvers: HashMap<_, _> = accounts
    .into_iter()
    .map(|a| (a.id, a))
    .collect();
```

嗯，还挺简洁，挺 `rusty`。

3. 使用 `fold`，语义表达更强：

```
let resolvers = account.into_iter().fold(HashMap::new(), |mut resolvers, a| {
    resolvers.entry(a.id).or_insert(Vec::new()).push(a);
    resolvers
});
```

## **总结**

在使用迭代器时，要清晰的认识到需要用到的方法是迭代型还是消费型适配器，如果一个调用链中没有以消费型适配器结尾，就需要打起精神了，也许，不远处就是一个陷阱在等你跳:)

# 奇怪的序列x..y

@todo

[https://www.reddit.com/r/rust/comments/rrgxr0/a\\_critique\\_of\\_rusts\\_range\\_types/](https://www.reddit.com/r/rust/comments/rrgxr0/a_critique_of_rusts_range_types/)

# 无处不在的迭代器

Rust 的迭代器无处不在，直至你在它上面栽了跟头，经过深入调查才发现：哦，原来是迭代器的锅。不信的话，看看这个报错你能想到是迭代器的问题吗：borrow of moved value: words.

## 报错的代码

以下的代码非常简单，用来统计文本中字词的数量，并打印出来：

```
fn main() {
    let s = "hello world";
    let mut words = s.split(" ");
    let n = words.count();
    println!("{}: {}", words, n);
}
```

四行代码，行云流水，一气呵成，且看成效：

```
error[E0382]: borrow of moved value: `words`
--> src/main.rs:5:21
|
3 |     let mut words = s.split(" ");
|           ----- move occurs because `words` has type `std::str::Split<'_, &str>`, which does not implement the `Copy` trait
4 |     let n = words.count();
|           ----- `words` moved due to this method call
5 |     println!("{}: {}", words, n);
|           ^^^^^^ value borrowed here after move
```

世事难料，我以为只有的生命周期、闭包才容易背叛革命，没想到一个你浓眉大眼的 count 方法也背叛革命。从报错来看，是因为 count 方法拿走了 words 的所有权，来看看签名：

```
fn count(self) -> usize
```

从签名来看，编译器的报错是正确的，但是为什么？为什么一个简单的标准库 count 方法就敢拿走所有权？

## 迭代器回顾

在[迭代器](#)章节中，我们曾经学习过两个概念：迭代器适配器和消费者适配器，前者用于对迭代器中的元素进行操作，最终生成一个新的迭代器，例如 `map`、`filter` 等方法；而后者用于消费掉迭代器，最终产生一个结果，例如 `collect` 方法，一个典型的示例如下：

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

在其中，我们还提到一个细节，消费者适配器会拿走迭代器的所有权，那么这个是否与我们最开始碰到的问题有关系？

## 深入调查

要解释这个问题，必须要找到 `words` 是消费者适配器的证据，因此我们需要深入源码进行查看。

其实。。。也不需要多深，只要进入 `words` 的源码，就能看出它属于 `Iterator` 特征，那说明 `split` 方法產生了一个迭代器？再来看看：

```
pub fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>
where
    P: Pattern<'a>,
    //An iterator over substrings of this string slice, separated by characters matched
    //by a pattern.
```

还真是，从代码注释来看，`Split` 就是一个迭代器类型，用来迭代被分隔符隔开的子字符串集合。

真相大白了，`split` 产生一个迭代器，而 `count` 方法是一个消费者适配器，用于消耗掉前者产生的迭代器，最终生成字词统计的结果。

本身问题不复杂，但是在[解决方法上](#)，可能还有点在各位客官的意料之外，且看下文。

## 最 `rusty` 的解决方法

你可能会想用 `collect` 来解决这个问题，先收集成一个集合，然后进行统计。当然此方法完全可行，但是很不 `rusty` (很符合 rust 规范、潮流的意思)，以下给出最 `rusty` 的解决方案：

```
let words = s.split(",");
let n = words.clone().count();
```

在继续之前，我得先找一个地方藏好，因为俺有一个感觉，烂西红柿正在铺天盖地的呼啸而来，伴随而来的是读者的正义呵斥：**你管 clone 叫最好、最 rusty 的解决方法？？**

大家且听我慢慢道来，事实上，在 Rust 中 `clone` 不总是性能低下的代名词，因为 `clone` 的行为完全取决于它的具体实现。

## 迭代器的`clone`代价

对于迭代器而言，它其实并不需要持有数据才能进行迭代，事实上它包含一个引用，该引用指向了保存在堆上的数据，而迭代器自身的结构是保存在栈上。

因此对迭代器的 `clone` 仅仅是复制了一份栈上的简单结构，性能非常高效，例如：

```
pub struct Split<'a, T: 'a, P>
where
    P: FnMut(&T) -> bool,
{
    // Used for `SplitWhitespace` and `SplitAsciiWhitespace`'s `as_str` methods
    pub(crate) v: &'a [T],
    pred: P,
    // Used for `SplitAsciiWhitespace`'s `as_str` method
    pub(crate) finished: bool,
}

impl<T, P> Clone for Split<'_, T, P>
where
    P: Clone + FnMut(&T) -> bool,
{
    fn clone(&self) -> Self {
        Split { v: self.v, pred: self.pred.clone(), finished: self.finished }
    }
}
```

以上代码实现了对 `Split` 迭代器的克隆，可以看出，底层的数组 `self.v` 并没有被克隆而是简单的复制了一个引用，依然指向了底层的数组 `&[T]`，因此这个克隆非常高效。

## 总结

看起来是无效借用导致的错误，实际上是迭代器被消费了导致的问题，这说明 Rust 编译器虽然会告诉你错误原因，但是这个原因不总是根本原因。我们需要一双慧眼和勤劳的手，来挖掘出这个宝藏，最后为己

所用。

同时，克隆在 Rust 中也并不总是**bad guy**的代名词，有的时候我们可以大胆去使用，当然前提是了解你的代码场景和具体的 `clone` 实现，这样你也能像文中那样作出非常 *rusty* 的选择。

# 线程间传递消息导致主线程无法结束

本篇陷阱较短，主要解决新手在多线程间传递消息时可能会遇到的一个问题：主线程会一直阻塞，无法结束。

Rust 标准库中提供了一个消息通道，非常好用，也相当简单明了，但是但是在使用起来还是可能存在坑：

```
use std::sync::mpsc;
fn main() {

    use std::thread;

    let (send, recv) = mpsc::channel();
    let num_threads = 3;
    for i in 0..num_threads {
        let thread_send = send.clone();
        thread::spawn(move || {
            thread_send.send(i).unwrap();
            println!("thread {} finished", i);
        });
    }

    for x in recv {
        println!("Got: {}", x);
    }
    println!("finished iterating");
}
```

以上代码看起来非常正常，运行下试试：

```
thread 0 finished
thread 1 finished
Got: 0
Got: 1
thread 2 finished
Got: 2
```

奇怪，主线程竟然卡死了，最后一行 `println!("finished iterating");` 一直没有被输出。

其实，上面的描述有问题，主线程并不是卡死，而是 `for` 循环并没有结束，至于 `for` 循环不结束的原因是消息通道没有被关闭。

回忆一下 Rust 消息通道关闭的两个条件：所有发送者全部被 `drop` 或接收者被 `drop`，由于 `for` 循环还在使用接收者，因为后者条件无法被满足，那么只能发送者全部被 `drop`，才能让例子中的消息通道关闭。

来分析下代码，每一个子线程都从 send 获取了一个拷贝，然后该拷贝在子线程结束时自动被 drop，看上去没问题啊。等等，好像 send 本身并没有被 drop，因为 send 要等到 main 函数结束才会被 drop，那么代码就陷入了一个尴尬的境地：main 函数要结束需要 for 循环结束，for 循环结束需要 send 被 drop，而 send 要被 drop 需要 main 函数结束。。。

破局点只有一个，那就是主动 drop 掉 send，这个简单，使用 std::mem::drop 函数即可，得益于 prelude，我们只需要使用 drop：

```
use std::sync::mpsc;
fn main() {
    use std::thread;

    let (send, recv) = mpsc::channel();
    let num_threads = 3;
    for i in 0..num_threads {
        let thread_send = send.clone();
        thread::spawn(move || {
            thread_send.send(i).unwrap();
            println!("thread {:#?} finished", i);
        });
    }

    drop(send);
    for x in recv {
        println!("Got: {}", x);
    }
    println!("finished iterating");
}
```

此时再运行，主线程将顺利结束。

## 总结

本文总结了一个新手在使用消息通道时常见的错误，那就是忘记处理创建通道时得到的发送者，最后由于该发送者的存活导致通道无法被关闭，最终主线程阻塞，造成程序错误。

# 警惕 UTF-8 引发的性能隐患

大家应该都知道，虽然 Rust 的字符串 `&str`、`String` 在底层是通过 `Vec<u8>` 实现的：字符串数据以字节数组的形式存在堆上，但在使用时，它们都是 UTF-8 编码的，例如：

```
fn main() {
    let s: &str = "中国人";
    for c in s.chars() {
        println!("{}", c) // 依次输出：中、国、人
    }

    let c = &s[0..3]; // 1. "中" 在 UTF-8 中占用 3 个字节 2. Rust 不支持字符串索引，因此只能
    // 通过切片的方式获取 "中"
    assert_eq!(c, "中");
}
```

从上述代码，可以很清晰看出，Rust 的字符串确实是 UTF-8 编码的，这就带来一个隐患：可能在某个转角，你就会遇到来自糟糕性能的示爱。

## 问题描述 & 解决

例如我们尝试写一个词法解析器，里面用到了以下代码

```
self.source.chars().nth(self.index).unwrap(); 去获取下一个需要处理的字符，大家可能会以为
.nth 的访问应该非常快吧？事实上它确实很快，但是并不妨碍这段代码在循环处理 70000 长度的字符串时，需要消耗 5s 才能完成！
```

这么看来，唯一的问题就在于 `.chars()` 上了。

其实原因很简单，简单到我们不需要用代码来说明，只需要文字描述即可传达足够的力量：每一次循环时，`.chars().nth(index)` 都需要对字符串进行一次 UTF-8 解析，这个解析实际上是相当昂贵的，特别是当配合循环时，算法的复杂度就是平方级的。

既然找到原因，那解决方法也很简单：只要将 `self.source.chars()` 的迭代器存储起来就行，这样每次 `.nth` 调用都会复用已经解析好的迭代器，而不是重新去解析一次 UTF-8 字符串。

当然，我们还可以使用三方库来解决这个问题，例如 [str\\_indices](#)。

## 总结

最终的优化结果如下：

- 保存迭代器后: 耗时 5s -> 4ms
- 进一步使用 u8 字节数组来替换 char , 最后使用 `String::from_utf8` 来构建 UTF-8 字符串: 耗时 4ms -> 400us

**肉眼可见的巨大提升，12500 倍！**

总之，我们在热点路径中使用字符串做 UTF-8 的相关操作时，就算不提前优化，也要做到心里有数，这样才能在问题发生时，进退自如。

# Rust性能剖析 todo

# 深入内存

部分内容借鉴了Rust in action和Rust高级编程

<https://www.youtube.com/watch?v=rDoqT-a6UFg>

# **指针和引用(todo)**

# 未初始化内存

<https://lucumr.pocoo.org/2022/1/30/unsafe-rust/>

# 内存分配(todo)

[https://www.reddit.com/r/rust/comments/s4pknf/investigating\\_memory\\_allocations\\_in\\_rust/](https://www.reddit.com/r/rust/comments/s4pknf/investigating_memory_allocations_in_rust/)

# 内存布局(todo)

[https://www.reddit.com/r/rust/comments/rwta4h/why\\_arent\\_rust\\_structs\\_laid\\_out\\_in\\_memory\\_like\\_c/](https://www.reddit.com/r/rust/comments/rwta4h/why_arent_rust_structs_laid_out_in_memory_like_c/)

# 虚拟内存

# **performance**

<https://nnethercote.github.io/perf-book/profiling.html>

## **How do I profile a Rust web application in production?**

[https://www.reddit.com/r/rust/comments/rupcux/how\\_do\\_i\\_profile\\_a\\_rust\\_web\\_application\\_in/](https://www.reddit.com/r/rust/comments/rupcux/how_do_i_profile_a_rust_web_application_in/)

<https://zhuanlan.zhihu.com/p/191655266>

## **内存对齐**

[https://www.reddit.com/r/rust/comments/s793x7/force\\_4byte\\_memory\\_alignment/](https://www.reddit.com/r/rust/comments/s793x7/force_4byte_memory_alignment/)

## **riggrep 为啥这么快**

[https://www.reddit.com/r/rust/comments/sr02aj/what\\_makes\\_ripgrep\\_so\\_fast/](https://www.reddit.com/r/rust/comments/sr02aj/what_makes_ripgrep_so_fast/)

## **测试堆性能**

[https://flakm.github.io/posts/heap\\_allocation/](https://flakm.github.io/posts/heap_allocation/)

[https://www.reddit.com/r/rust/comments/t06hk7/string\\_concatenations\\_benchmarks\\_updated/](https://www.reddit.com/r/rust/comments/t06hk7/string_concatenations_benchmarks_updated/)

# Rust所有权转移时发生了奇怪的深拷贝

深拷贝可以说是Rust性能优化的禁忌之词，但是在最不该发生深拷贝的地方却发生了，本文带领大家来深入分析下原因。

在所有权章节中，我们详细介绍过[所有权转移\(move\)](#)，里面提到过一个重点：当类型实现 Copy 特征时，不会转移所有权，而是直接对值进行拷贝：

```
fn main() {
    let x = 1;
    let y = x;
    // 不会报错
    println!("我(x)的值仅仅是被复制了，我还拥有值的所有权，不信你看: {:?}", x);

    let s = "aaa".to_string();
    let s1 = s;
    // 会报错
    println!("我(s)的值被转移给了s1，我已经失去所有权了: {}", s);
}
```

这里的 `x` 是数值类型，因此实现了 Copy 特征，当赋值给 `y` 时，仅仅是复制了值，并没有把所有权转移给 `y`，但是 `s` 恰好相反，它没有实现 Copy 特征，当赋值后，所有权被转移给 `s1`，最终导致了最后一行代码的报错。

根据之前的所有权学习章节，所有权转移时的仅仅是复制一个引用，并不会复制底层的数据，例如上面代码中，`s` 的所有权转移给 `s1` 时，仅仅是复制了一个引用，该引用继续指向之前的字符串底层数据，因此 **所有权转移的性能是非常高的**。

但是如果一切都这么完美，也不会出现这篇文章了，实际上是怎么样？先来看一段代码。

## move时发生了数据的深拷贝

```
struct LargeArray {
    a: [i128; 10000],
}

impl LargeArray {
    #[inline(always)]
    fn transfer(mut self) -> Self {
        println!("{:?}", &mut self.a[1] as *mut i128);

        // 改变数组中的值
        self.a[1] += 23;
        self.a[4] += 24;

        // 返回所有权
        self
    }
}

fn main() {
    let mut f = LargeArray { a: [10i128; 10000] };

    println!("{:?}", &mut f.a[1] as *mut i128);

    let mut f2 = f.transfer();

    println!("{:?}", &mut f2.a[1] as *mut i128);
}
```

上面的例子很简单，创建了一个结构体 `f` (内部有一个大数组)，接着将它的所有权转移给 `transfer` 方法，最后再通过 `self` 返回，转移给 `f2`，在此过程中，观察结构体中的数组第二个元素的内存地址如何变化。

这里还有几个注意点：

- `LargeArray` 没有实现 `Copy` 特征，因此在所有权转移时，**本应该只是复制一下引用，底层的数组并不会被复制**
- `transfer` 方法的参数 `self` 表示接收所有权，而不是借用，返回类型 `Self` 也表示返回所有权，而不是返回借用，具体内容在[方法](#)章节有介绍

从上可知，我们并不应该去复制底层的数组，那么底层数组的地址也不应该变化，换而言之三次内存地址输出应该是同一个地址。但是真的如此吗？世事难料：

```
0x16f9d6870
0x16fa4bbc0
0x16fa24ac0
```

果然，结果让人大跌眼镜，竟然三次地址都不一样，意味着每次转移所有权都发生了底层数组的深拷贝！什么情况？！！如果这样，我们以后还能信任Rust吗？完全不符合官方的宣传。

在福建有一个武夷山5A景区，不仅美食特别好吃，而且风景非常优美，其中最著名的就是历时1个多小时的九曲十八弯漂流，而我们的结论是否也能像漂游一样来个大转折？大家拭目以待。

## 罪魁祸首println?

首先，通过谷歌搜索，我发现了一些蛛丝马迹，有文章提到如果通过 `println` 输出内存地址，可能会导致编译器优化失效，也就是从本该有的所有权转移变成了深拷贝，不妨来试试。

但是问题又来了，如果不用 `println` 或者类似的方法，我们怎么观察内存地址？好像陷入了绝路。。。只能从Rust之外去想办法了，此时大学学过的汇编发挥了作用：

```

.LCPI0_0:
    .quad 10
    .quad 0
example::xxx:
    mov    eax, 160000
    call   __rust_probestack
    sub    rsp, rax
    mov    rax, rsp
    lea    rcx, [rsp + 160000]
    vbroadcasti128 ymm0, xmmword ptr [rip + .LCPI0_0]

.LBB0_1:
    vmovdqu ymmword ptr [rax], ymm0
    vmovdqu ymmword ptr [rax + 32], ymm0
    vmovdqu ymmword ptr [rax + 64], ymm0
    vmovdqu ymmword ptr [rax + 96], ymm0
    vmovdqu ymmword ptr [rax + 128], ymm0
    vmovdqu ymmword ptr [rax + 160], ymm0
    vmovdqu ymmword ptr [rax + 192], ymm0
    vmovdqu ymmword ptr [rax + 224], ymm0
    vmovdqu ymmword ptr [rax + 256], ymm0
    vmovdqu ymmword ptr [rax + 288], ymm0
    vmovdqu ymmword ptr [rax + 320], ymm0
    vmovdqu ymmword ptr [rax + 352], ymm0
    vmovdqu ymmword ptr [rax + 384], ymm0
    vmovdqu ymmword ptr [rax + 416], ymm0
    vmovdqu ymmword ptr [rax + 448], ymm0
    vmovdqu ymmword ptr [rax + 480], ymm0
    vmovdqu ymmword ptr [rax + 512], ymm0
    vmovdqu ymmword ptr [rax + 544], ymm0
    vmovdqu ymmword ptr [rax + 576], ymm0
    vmovdqu ymmword ptr [rax + 608], ymm0
    add    rax, 640
    cmp    rax, rcx
    jne    .LBB0_1
    mov    rax, qword ptr [rsp + 16]
    mov    rdx, qword ptr [rsp + 24]
    add    rax, 69
    adc    rdx, 0
    add    rsp, 160000
    vzeroupper
    ret

```

去掉所有 `println` 后的汇编生成如上所示(大家可以在godbolt上自己尝试), 以我蹩脚的汇编水平来看, 貌似没有任何数组拷贝的发生, 也就是说: 如同量子的不可观测性, 我们的 `move` 也这么傲娇? 我们用 `println` 观测, 它就傲娇去复制, 不观测时, 就老老实实转移所有权? WTF!

事情感觉进入了僵局, 下一步该如何办?

## 栈和堆的不同move行为

我突然灵光一现，想到一个问题，之前的所有权转移其实可以分为两类：**栈上数据的复制和堆上数据的转移**，这也是非常符合直觉的，例如 `i32` 这种类型实现了 `Copy` 特征，可以存储在栈上，因此它就是复制行为，而 `String` 类型是引用存储在栈上，底层数据存储在堆上，因此转移所有权时只需要复制一下引用即可。

那问题来了，我们的 `LargeArray` 存在哪里？这也许就是一个破局点！

```
struct LargeArray {  
    a: [i128; 10000],  
}
```

结构体是一个复合类型，它内部字段的数据存在哪里，就大致决定了它存在哪里。而该结构体里面的 `a` 字段是一个数组，而不是动态数组 `Vec`，从[数组](#)章节可知：数组是存储在栈上的数据结构！

再想想，栈上的数据在 `move` 的时候，是要从一个栈复制到另外一个栈的，那是不是内存地址就变了？！因此，就能完美解释，为什么使用 `println` 时，数组的地址会变化了，是因为栈上的数组发生了复制。

但是问题还有，为什么不使用 `println`，数组地址就不变？要解释清楚这个问题，先从编译器优化讲起。

## 编译器对move的优化

从根本上来说，`move` 就意味着拷贝复制，只不过看是浅拷贝还是深拷贝，对于堆上的数据来说，浅拷贝只复制引用，而栈上的数据则是整个复制。

但是在实际场景中，由于编译器的复杂实现，它能优化的场景远比我们想象中更多，例如对于 `move` 过程中的复制，编译器有可能帮你优化掉，在没有 `println` 的代码中，该 `move` 过程就被Rust编译器优化了。

但是这种编译器优化非常复杂，而且随着Rust的版本更新在不停变化，因此几乎没有人能说清楚这里面的门道，但是有一点可以知道：**move 确实存在被优化的可能性，最终避免了复制的发生**。

那么 `println` 没有被优化的原因也呼之欲出了：它阻止了编译器对 `move` 的优化。

## println阻止了优化

编译器优化的一个基本准则就是：中间过程没有其它代码在使用，则可以尝试消除这些中间过程。

回头来看看 `println`:

```
println!("{:?}", &mut f.a[1] as *mut i128);
```

它需要打印数组在各个点的内存地址，假如编译器优化了复制，那这些中间状态的内存地址是不是就丢失了？对于这种可能会导致状态丢失的情况，编译器是不可能进行优化的，因此 `move` 时的栈上数组复制就顺理成章的发生了，还是2次。

## 最佳实践

那么，在实践中遇到这种情况怎么办？

### **&mut self**

其实办法也很多，首当其冲的就是使用 `&mut self` 进行可变借用，而不是转移进来所有权，再转移出去。

### **Box分配到堆上**

如果你确实需要依赖所有权的转移来实现某个功能(例如链式方法调用: `x.a().b()...`)，那么就需要使用 `Box` 把该数组分配在堆上，而不是栈上：

```

struct LargeArray {
    a: Box<[i128; 10000]>,
}

impl LargeArray {
    #[inline(always)]
    fn transfer(mut self) -> Self {
        println!("{:?}", &mut self.a[1] as *mut i128);

        //do some stuff to alter it
        self.a[1] += 23;
        self.a[4] += 24;

        //return the same object
        self
    }
}

fn main() {
    let mut f = LargeArray { a: Box::new([10i128; 10000])};

    println!("{:?}", &mut f.a[1] as *mut i128);

    let mut f2 = f.transfer();

    println!("{:?}", &mut f2.a[1] as *mut i128);
}

```

输出如下：

```

0x138008010
0x138008010
0x138008010

```

完美符合了我们对堆上数据的预期，hooray!

## 神龟莫测的编译器优化

当然，你也可以选择相信编译器的优化，虽然很难识它真面目，同时它的行为也神鬼莫测，但是总归是在之前的例子中证明了，它确实可以，不是嘛？=，=

# 糟糕的提前优化

## 函数调用

由于Rust的编译器和LLVM很强大，因此就算你使用了多层函数调用去完成一件事(嵌套函数调用往往出于设计上的考虑)，依然不会有性能上的影响，因为最终生成的机器码会消除这些多余的函数调用。

总之用Rust时，你不必操心多余的函数调用，只要写合理的代码，然后Rust会帮助你运行的更快!

# Clone和Copy

# 减少runtime check

[https://www.reddit.com/r/rust/comments/sx8b7m/how\\_is\\_rust\\_able\\_to\\_elide\\_bounds\\_checks/](https://www.reddit.com/r/rust/comments/sx8b7m/how_is_rust_able_to_elide_bounds_checks/)

## 减少集合访问的边界检查

以下代码，我们实现了两种循环方式：

```
// 第一种
let collection = [1, 2, 3, 4, 5];
for i in 0..collection.len() {
    let item = collection[i];
    // ...
}

// 第二种
for item in collection {
```

第一种方式是循环索引，然后通过索引下标去访问集合，第二种方式是直接循环迭代集合中的元素，优劣如下：

- **性能**: 第一种使用方式中 `collection[index]` 的索引访问，会因为边界检查(bounds checking)导致运行时的性能损耗 - Rust会检查并确认 `index` 是落在集合内也就是合法的，但是第二种直接迭代的方式就不会触发这种检查,因为编译器会在编译时就完成分析并证明这种访问是合法的`

## Box::leak

[https://www.reddit.com/r/rust/comments/rntx7s/why\\_use\\_boxleak/](https://www.reddit.com/r/rust/comments/rntx7s/why_use_boxleak/)

## bounds check

[https://www.reddit.com/r/rust/comments/rnbubh/whats\\_the\\_big\\_deal\\_with\\_bounds\\_checking/](https://www.reddit.com/r/rust/comments/rnbubh/whats_the_big_deal_with_bounds_checking/)

[https://www.reddit.com/r/rust/comments/s6u65e/optimization\\_of\\_bubble\\_sort\\_fails\\_without\\_hinting/](https://www.reddit.com/r/rust/comments/s6u65e/optimization_of_bubble_sort_fails_without_hinting/)

## 使用assert 优化检查性能

[https://www.reddit.com/r/rust/comments/rui1zz/write\\_assertions\\_that\\_clarify\\_code\\_to\\_both\\_the/](https://www.reddit.com/r/rust/comments/rui1zz/write_assertions_that_clarify_code_to_both_the/)

# CPU缓存性能优化

[https://github.com/TC5027/blog/blob/master/false\\_sharing.md](https://github.com/TC5027/blog/blob/master/false_sharing.md)

## On a use of the "repr" attribute in Rust

Consider we work with the following struct representing a counter,

```
struct Counter(u64);
```

and we want to increment it with random `u8` values with the help of a for loop :

```
use rand::Rng;
fn main() {
    let mut counter = Counter(0);
    let mut rng = rand::thread_rng();

    for _ in 0..1_000_000 {
        counter.0 += rng.gen::<u8>() as u64;
    }
}
```

This takes 1.90ms to run on my laptop using `cargo run --release`. Remember this timing as it will be our reference value :) Now suppose we were given this struct, holding not 1 but 2 counters :

```
struct Counters {
    c1 : u64,
    c2 : u64
}
```

Using the same approach, performing the increments for the 2 counters in a single-threaded fashion, we would expect to be twice slower (in fact it takes 3.71ms to execute). Can we do better ? Well, as our 2 counters are independent, we could spawn 2 threads, assign them one counter and increment concurrently ! Given I have 4 CPUs on my laptop, I would expect to be just as fast as the first scenario. Let's see !

First thing, we could create a local variable in each thread which would be incremented and then we would set the counter value to this incremented one (spoiler : good idea). But we could also save these 2 variables and share the `Counter` between the 2 threads with an `Arc` (spoiler : definitely not worth). Let's do this second option ! ^^

Doing the following code,

```
fn main() {
    let counters = Arc::new(Counters{c1:0, c2:0});
    let counters_clone = counters.clone();

    let handler1 = thread::spawn(move || {
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            counters.c1 += rng.gen::<u8>() as u64;
        }
    });
    let handler2 = thread::spawn(move || {
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            counters_clone.c2 += rng.gen::<u8>() as u64;
        }
    });
    handler1.join(); handler2.join();
}
```

we end up with an error :

**cannot assign to data in an Arc cannot assign help: trait DerefMut is required to modify through a dereference, but it is not implemented for std::sync::Arc<Counters> rustc(E0594)**

Unlucky. Maybe we could use **atomic types**. These types provide operations that synchronize updates between threads. In fact, as an equivalent of `+=` we could use the `fetch_add` method which has the following signature : `pub fn fetch_add(&self, val: u64, order: Ordering) -> u64`. What should be highlighted is the `&self`. We could expect a `&mut self` given the modification we want to perform using it but thanks to the property that an atomic operation is performed without interruptions we don't need exclusive access to the variable to safely update it. We can solve the error replacing the counter's type by `AtomicU64` as like that we only require `Arc` to implement the `Deref` trait (given the signature of `fetch_add`) and it is the case !

We so have to change a bit our struct to :

```
struct Counters {
    c1 : AtomicU64,
    c2 : AtomicU64,
}
```

and our code to :

```

fn main() {
    let counters = Arc::new(Counters{
        c1 : AtomicU64::new(0),
        c2 : AtomicU64::new(0)
    });
    let counters_clone = counters.clone();
    let handler1 = thread::spawn(move || {
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            counters.c1.fetch_add(rng.gen::<u8>() as u64, Relaxed);
        }
    });
    let handler2 = thread::spawn(move || {
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            counters_clone.c2.fetch_add(rng.gen::<u8>() as u64, Relaxed);
        }
    });
    handler1.join(); handler2.join();
}

```

We could naturally expect the operation on Atomics to be a bit slower than the ones on `u64` but let's see ! 30.22ms .. ok... that's terrible ^^ Do Atomics operations explain all this ? I ran a benchmark to compare `+=` and `fetch_add( ,Relaxed)` to figure it out :

```

let mut sum = 0;
let start = Instant::now();
for _ in 0..10_000_000 {
    sum += rng.gen::<u8>() as u64;
}
println!("time spent u64 sum : {:?}", start.elapsed());
let atomic_sum = AtomicU64::new(0);
let start = Instant::now();
for _ in 0..10_000_000 {
    atomic_sum.fetch_add(rng.gen::<u8>() as u64, Relaxed);
}
println!("time spent AtomicU64 sum : {:?}", start.elapsed());

```

The `u64` sums takes 20.07ms while the `AtomicU64` one takes 70.28ms. So we should only be 3 times slower than 2ms but we are 15 times slower how can it be ???

Hint : CPU cache... but why should we care ? CPU cache is a data storage, located close to CPU, offering a fast access to data. In a computer, when the CPU needs to read or write a value, it checks if it is present inside the cache or not. If it is the case then the CPU directly uses the cached data. Otherwise, the cache allocates a new entry and copies data from main memory, an entry being of fixed size and called *cache line*. CPU cache is relatively small compared to RAM but much faster,

and that's why a program should be designed to use as much as possible data lying in cache, based on a locality principle, to avoid expensive access to RAM.

If we represent our current situation it looks like this : 

The red square corresponds to the first counter and the green one to the second. They can potentially lie in the same cache line !

If data is modified through CPU 0 in its L1 cache we expect our computer to reflect the changes both in memory and in the other L1 cache. To ensure this coherency, there exists coherence protocols which can force the **whole cache line** impacted by the change to be propagated through the whole system, in order to update the copies of the value changed.

With that in mind, what is happening in our code comes from that : we suffer from coherency protocol due to our 2 counters lying on the same cache line. Updating first counter through CPU 0 involves an update in the system of the data stored in the cache line where the second counter (unchanged) potentially lies. During this update, CPU 1 cannot access the second counter whereas it is clearly independent from the change made by CPU 0, and that's why we are slow. How can we solve then ? well by making sure that the counters lie on different cache lines and that's where we can use the `repr` attribute.

In Rust, we can specify the alignment we want for our type with the `repr(align)` attribute. We use it like this :

```
#[repr(align(64))]
struct CachePadded(AtomicU64);
```

A data of alignment X is stored in memory at address multiple of X. Knowing this, giving to our counters an alignment equal to the size of a cache line, we ensure that the 2 counters won't be stored in the same cache line !

We can get the size of cache lines with command `getconf LEVEL1_DCACHE_LINESIZE`. On my laptop the output value is 64.

With those changes we have now a timing of 7.16ms which seems decent given we work with Atomics. Mission succeeded !

Finally given my remark at the beginning, I wanted to share a potentially better solution, using local variables in the threads, and channels to communicate these local variables back to the main thread :

```

use std::sync::mpsc::channel;
fn main() {
    let (s1,t1) = channel();
    let (s2,t2) = channel();
    let h1 = thread::spawn(move || {
        let mut local_counter = 0;
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            local_counter += rng.gen::<u8>() as u64;
        }
        s1.send(local_counter)
    });
    let h2 = thread::spawn(move || {
        let mut local_counter = 0;
        let mut rng = rand::thread_rng();
        for _ in 0..1_000_000 {
            local_counter += rng.gen::<u8>() as u64;
        }
        s2.send(local_counter)
    });

    h1.join();
    h2.join();
    let counter = Counters{c1: t1.recv().unwrap(),c2: t2.recv().unwrap()};
}

```

It takes 2.03 ms to execute :)

## 动态和静态分发

[https://www.reddit.com/r/rust/comments/ruavjm/is\\_there\\_a\\_difference\\_in\\_performance\\_between/](https://www.reddit.com/r/rust/comments/ruavjm/is_there_a_difference_in_performance_between/)

# 计算性能优化

[https://www.reddit.com/r/rust/comments/rn7ozz/find\\_perfect\\_number\\_comparison\\_go\\_java\\_rust/](https://www.reddit.com/r/rust/comments/rn7ozz/find_perfect_number_comparison_go_java_rust/)

```

package main

import (
    "fmt"
    "math"
    "time"
)

func main() {
    n := 320000
    nums := make(map[int][]int)
    start := time.Now()
    calPerfs(n, nums)
    fmt.Printf("runtime: %s\n", time.Since(start))
}

func calPerfs(n int, nums map[int][]int) {
    for i := 1; i <= n; i++ {
        d := divs(i)
        if sum(d) == i {
            nums[i] = all(d)
        }
    }
}

func divs(num int) map[int]struct{} {
    r := make(map[int]struct{})
    r[1] = struct{}{}
    mid := int(math.Sqrt(float64(num)))
    for i := 2; i <= mid; i++ {
        if num%i == 0 {
            r[i] = struct{}{}
            r[num/i] = struct{}{}
        }
    }
    return r
}

func sum(ds map[int]struct{}) int {
    var n int
    for k := range ds {
        n += k
    }
    return n
}

func all(ds map[int]struct{}) []int {
    var a []int
    for k := range ds {
        a = append(a, k)
    }
}

```

```
        return a
    }
```

## 120ms

```
use std::time::Instant;

const N: usize = 320_000           ;

fn is_perfect(n: usize) -> bool {
    //println!("{}:{:?}", n);
    let mut sum = 1;
    let end = (n as f64).sqrt() as usize;
    for i in 2..end + 1{
        if n % i == 0 {
            if i * i == n {
                sum += i;
            }
            else {
                sum += i + n / i;
            }
        }
    }
    sum == n
}

fn find_perfs(n: usize) -> Vec<usize> {
    let mut perfs:Vec<usize> = vec![];
    for i in 2..n + 1 {
        if is_perfect(i) {
            perfs.push(i)
        }
    }
    perfs
}

fn main() {
    let start = Instant::now();
    let perfects = find_perfs(N);
    println!("{}:{:?}", start.elapsed());
    println!("{}:{?}, in {}:{?}, perfects, N");
}
```

## 90ms

```
use {
    std::time::Instant,
};

const N: usize = 320000;

// Optimized, takes about 320ms on an Core i7 6700 @ 3.4GHz
fn cal_perfs2(n: usize) -> Vec<usize> {
    (1..=n)
        .into_iter()
        .filter(|i| cal2(*i) == *i)
        .collect::<Vec<_>>()
}

fn cal2(n: usize) -> usize {
    (2..=(n as f64).sqrt()) as usize
        .into_iter()
        .filter_map(|i| if n % i == 0 { Some([i, n / i]) } else { None })
        .map(|a| a[0] + a[1])
        .sum::<usize>()
        + 1
}

fn main() {
    let start = Instant::now();
    let perf2 = cal_perfs2(N);
    println!("{}: {:?}", perf2);
    println!("Optimized: {:?}", start.elapsed());
}
```

# **堆和栈**

[https://www.reddit.com/r/rust/comments/rkddg3/stackheap\\_question\\_regarding\\_performance/](https://www.reddit.com/r/rust/comments/rkddg3/stackheap_question_regarding_performance/)

# **内存allocator todo**

[https://www.reddit.com/r/rust/comments/s28g4x/allocating\\_many\\_boxes\\_at\\_once/](https://www.reddit.com/r/rust/comments/s28g4x/allocating_many_boxes_at_once/)

[https://www.reddit.com/r/rust/comments/szza43/memory\\_freed\\_but\\_not\\_immediately/](https://www.reddit.com/r/rust/comments/szza43/memory_freed_but_not_immediately/)

# **常用性能测试工具**

<https://era.co/blog/unbuffered-io-slows-rust-programs>

## **profiling**

[https://www.reddit.com/r/rust/comments/rxj81f/rust\\_profiling/](https://www.reddit.com/r/rust/comments/rxj81f/rust_profiling/)

# Enum内存优化 todo

<https://blog.zhuangty.com/rust-enum-layout/>

# **对抗编译检查**

# **LLVM todo**

<https://ttalk.im/2021/12/llvm-infrastructure-and-rust.html>

# 常见属性标记

## 强制内存对齐

```
#[repr(align(64))]  
struct CachePadded(AtomicU64);
```

A data of alignment X is stored in memory at address multiple of X

<https://doc.rust-lang.org/reference/attributes.html>

# 优化编译速度

# 编译器优化

在Rust中，一段很不起眼的代码中可能也隐藏着玄机，编译器在细无声的为我们做着各种优化，本章将记录这些优化，帮助大家更好的理解程序的性能。

# Option枚举

[https://www.reddit.com/r/learnrust/comments/rz34ht/where\\_does\\_the\\_data\\_go\\_if\\_you\\_replace\\_so\\_me\\_with/](https://www.reddit.com/r/learnrust/comments/rz34ht/where_does_the_data_go_if_you_replace_so_me_with/)

## 附录 A：关键字

下面的列表包含 Rust 中正在使用或者以后会用到的关键字。因此，这些关键字不能被用作标识符（除了 [原生标识符](#)），包括函数、变量、参数、结构体字段、模块、包、常量、宏、静态值、属性、类型、特征或生命周期。

### 目前正在使用的关键字

如下关键字目前有对应其描述的功能。

- `as` - 强制类型转换，或 `use` 和 `extern crate` 包和模块引入语句中的重命名
- `break` - 立刻退出循环
- `const` - 定义常量或原生常量指针 (constant raw pointer)
- `continue` - 继续进入下一次循环迭代
- `crate` - 链接外部包
- `dyn` - 动态分发特征对象
- `else` - 作为 `if` 和 `if let` 控制流结构的 fallback
- `enum` - 定义一个枚举类型
- `extern` - 链接一个外部包，或者一个宏变量(该变量定义在另外一个包中)
- `false` - 布尔值 `false`
- `fn` - 定义一个函数或 [函数指针类型](#) (*function pointer type*)
- `for` - 遍历一个迭代器或实现一个 trait 或者指定一个更高级的生命周期
- `if` - 基于条件表达式的结果来执行相应的分支
- `impl` - 为结构体或者特征实现具体功能
- `in` - `for` 循环语法的一部分
- `let` - 绑定一个变量
- `loop` - 无条件循环
- `match` - 模式匹配
- `mod` - 定义一个模块
- `move` - 使闭包获取其所捕获项的所有权
- `mut` - 在引用、裸指针或模式绑定中使用，表明变量是可变的
- `pub` - 表示结构体字段、`impl` 块或模块的公共可见性
- `ref` - 通过引用绑定
- `return` - 从函数中返回
- `Self` - 实现特征类型的类型别名
- `self` - 表示方法本身或当前模块

- `static` - 表示全局变量或在整个程序执行期间保持其生命周期
- `struct` - 定义一个结构体
- `super` - 表示当前模块的父模块
- `trait` - 定义一个特征
- `true` - 布尔值 `true`
- `type` - 定义一个类型别名或关联类型
- `unsafe` - 表示不安全的代码、函数、特征或实现
- `use` - 在当前代码范围内(模块或者花括号对)引入外部的包、模块等
- `where` - 表示一个约束类型的从句
- `while` - 基于一个表达式的结果判断是否继续循环

## 保留做将来使用的关键字

如下关键字没有任何功能，不过由 Rust 保留以备将来的应用。

- `abstract`
- `async`
- `await`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`
- `virtual`
- `yield`

## 原生标识符

原生标识符 (Raw identifiers) 允许你使用通常不能使用的关键字，其带有 `r#` 前缀。

例如，`match` 是关键字。如果尝试编译如下使用 `match` 作为名字的函数：

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

会得到这个错误：

```
error: expected identifier, found keyword `match`
--> src/main.rs:4:4
|
4 | fn match(needle: &str, haystack: &str) -> bool {
|     ^^^^^^ expected identifier, found keyword
```

该错误表示你不能将关键字 `match` 用作函数标识符。你可以使用原生标识符将 `match` 作为函数名称使用：

文件名: src/main.rs

```
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

此代码编译没有任何错误。注意 `r#` 前缀需同时用于函数名定义和 `main` 函数中的调用。

原生标识符允许使用你选择的任何单词作为标识符，即使该单词恰好是保留关键字。此外，原生标识符允许你使用其它 Rust 版本编写的库。比如，`try` 在 Rust 2015 edition 中不是关键字，却在 Rust 2018 edition 是关键字。所以如果用 2015 edition 编写的库中带有 `try` 函数，在 2018 edition 中调用时就需要使用原始标识符语法，在这里是 `r#try`。

## 附录 B：运算符与符号

该附录包含了 Rust 目前出现过的各种符号，这些符号之前都分散在各个章节中。

### 运算符

表 B-1 包含了 Rust 中的运算符、上下文中的示例、简短解释以及该运算符是否可重载。如果一个运算符是可重载的，则该运算符上用于重载的特征也会列出。

下表中，`expr` 是表达式，`ident` 是标识符，`type` 是类型，`var` 是变量，`trait` 是特征，`pat` 是匹配分支(pattern)。

表 B-1：运算符

运算符	示例	解释	是否可重载
!	<code>ident!(...), ident!{...}, ident![...]</code>	宏展开	
!	<code>!expr</code>	按位非或逻辑非	<code>Not</code>
<code>!=</code>	<code>var != expr</code>	不等比较	<code>PartialEq</code>
<code>%</code>	<code>expr % expr</code>	算术求余	<code>Rem</code>
<code>%=</code>	<code>var %= expr</code>	算术求余与赋值	<code>RemAssign</code>
<code>&amp;</code>	<code>&amp;expr, &amp;mut expr</code>	借用	
<code>&amp;</code>	<code>&amp;type, &amp;mut type, &amp;'a type, &amp;'a mut type</code>	借用指针类型	
<code>&amp;</code>	<code>expr &amp; expr</code>	按位与	<code>BitAnd</code>
<code>&amp;=</code>	<code>var &amp;= expr</code>	按位与及赋值	<code>BitAndAssign</code>
<code>&amp;&amp;</code>	<code>expr &amp;&amp; expr</code>	逻辑与	
<code>*</code>	<code>expr * expr</code>	算术乘法	<code>Mul</code>
<code>*=</code>	<code>var *= expr</code>	算术乘法与赋值	<code>MulAssign</code>
<code>*</code>	<code>*expr</code>	解引用	
<code>*</code>	<code>*const type, *mut type</code>	裸指针	
<code>+</code>	<code>trait + trait, 'a + trait</code>	复合类型限制	

运算符	示例	解释	是否可重载
+	expr + expr	算术加法	Add
+=	var += expr	算术加法与赋值	AddAssign
,	expr, expr	参数以及元素分隔符	
-	- expr	算术取负	Neg
-	expr - expr	算术减法	Sub
-=	var -= expr	算术减法与赋值	SubAssign
->	fn(...) -> type,  ...  -> type	函数与闭包, 返回类型	
.	expr.ident	成员访问	
..	.., expr..., ..expr, expr..expr	右半开区间	PartialOrd
..=	..=expr, expr..=expr	闭合区间	PartialOrd
..	..expr	结构体更新语法	
..	variant(x, ..), struct_type { x, .. }	“代表剩余部分”的模式绑定	
...	expr...expr	(不推荐使用, 用 ..= 替代) 闭合区间	
/	expr / expr	算术除法	Div
/=	var /= expr	算术除法与赋值	DivAssign
:	pat: type, ident: type	约束	
:	ident: expr	结构体字段初始化	
:	'a: loop {...}	循环标志	
;	expr;	语句和语句结束符	
;	[...; len]	固定大小数组语法的部分	
<<	expr << expr	左移	Shl
<<=	var <<= expr	左移与赋值	ShlAssign
<	expr < expr	小于比较	PartialOrd
<=	expr <= expr	小于等于比较	PartialOrd
=	var = expr, ident = type	赋值/等值	
==	expr == expr	等于比较	PartialEq
=>	pat => expr	匹配分支语法的部分	
>	expr > expr	大于比较	PartialOrd

运算符	示例	解释	是否可重载
<code>&gt;=</code>	<code>expr &gt;= expr</code>	大于等于比较	<code>PartialOrd</code>
<code>&gt;&gt;</code>	<code>expr &gt;&gt; expr</code>	右移	<code>Shr</code>
<code>&gt;&gt;=</code>	<code>var &gt;&gt;= expr</code>	右移与赋值	<code>ShrAssign</code>
<code>@</code>	<code>ident @ pat</code>	模式绑定	
<code>^</code>	<code>expr ^ expr</code>	按位异或	<code>BitXor</code>
<code>^=</code>	<code>var ^= expr</code>	按位异或与赋值	<code>BitXorAssign</code>
<code> </code>	<code>pat   pat</code>	模式匹配中的多个可选条件	
<code> </code>	<code>expr   expr</code>	按位或	<code>BitOr</code>
<code> =</code>	<code>var  = expr</code>	按位或与赋值	<code>BitOrAssign</code>
<code>  </code>	<code>expr    expr</code>	逻辑或	
<code>?</code>	<code>expr?</code>	错误传播	

## 非运算符符号

表 B-2：独立语法

符号	解释
<code>'ident</code>	生命周期名称或循环标签
<code>...u8, ...i32, ...f64, ...usize, 等</code>	指定类型的数值常量
<code>"..."</code>	字符串常量
<code>r"...", r#"..."#, r##"..."##, etc.</code>	原生字符串, 未转义字符
<code>b"..."</code>	将 <code>&amp;str</code> 转换成 <code>&amp;[u8; N]</code> 类型的数组
<code>br"...", br#"..."#, br##"..."##, 等</code>	原生字节字符串, 原生和字节字符串字面值的结合
<code>'...</code>	Char 字符
<code>b'...'</code>	ASCII 字节
<code>  ...   expr</code>	闭包
<code>!</code>	代表总是空的类型, 用于发散函数(无返回值函数)

符号	解释
-	模式绑定中表示忽略的意思；也用于增强整型字面值的可读性

表 B-3 展示了模块和对象调用路径的语法。

表 B-3：路径相关语法

符号	解释
<code>ident::ident</code>	命名空间路径
<code>::path</code>	从当前的包的根路径开始的相对路径
<code>self::path</code>	与当前模块相对的路径（如一个显式相对路径）
<code>super::path</code>	与父模块相对的路径
<code>type::ident, &lt;type as trait&gt;::ident</code>	关联常量、关联函数、关联类型
<code>&lt;type&gt;::...</code>	不可以被直接命名的关联项类型（如 <code>&lt;&amp;T&gt;::...</code> , <code>&lt;[T]&gt;::...</code> , 等）
<code>trait::method(...)</code>	使用特征名进行方法调用，以消除方法调用的二义性
<code>type::method(...)</code>	使用类型名进行方法调用，以消除方法调用的二义性
<code>&lt;type as trait&gt;::method(...)</code>	将类型转换为特征，再进行方法调用，以消除方法调用的二义性

表 B-4 展示了使用泛型参数时用到的符号。

表 B-4：泛型

符号	解释
<code>path&lt;...&gt;</code>	为一个类型中的泛型指定具体参数（如 <code>Vec&lt;u8&gt;</code> ）
<code>path::&lt;...&gt;, method::&lt;...&gt;</code>	为一个泛型、函数或表达式中的方法指定具体参数，通常指双冒号（turbofish）（如 <code>"42".parse::&lt;i32&gt;()</code> ）
<code>fn ident&lt;...&gt; ...</code>	泛型函数定义
<code>struct ident&lt;...&gt; ...</code>	泛型结构体定义
<code>enum ident&lt;...&gt; ...</code>	泛型枚举定义
<code>impl&lt;...&gt; ...</code>	实现泛型
<code>for&lt;...&gt; type</code>	高阶生命周期限制

符号	解释
<code>type&lt;ident=type&gt;</code>	泛型，其一个或多个相关类型必须被指定为特定类型（如 <code>Iterator&lt;Item=T&gt;</code> ）

表 B-5 展示了使用特征约束来限制泛型参数的符号。

表 B-5：特征约束

符号	解释
<code>T: U</code>	泛型参数 <code>T</code> 需实现 <code>U</code> 类型
<code>T: 'a</code>	泛型 <code>T</code> 的生命周期必须长于 <code>'a</code> （意味着该类型不能传递包含生命周期短于 <code>'a</code> 的任何引用）
<code>T : 'static</code>	泛型 <code>T</code> 只能使用声明周期为 <code>'static</code> 的引用
<code>'b: 'a</code>	生命周期 <code>'b</code> 必须长于生命周期 <code>'a</code>
<code>T: ?Sized</code>	使用一个不定大小的泛型类型
<code>'a + trait, trait + trait</code>	多个类型组成的复合类型限制

表 B-6 展示了宏以及在一个对象上定义属性的符号。

表 B-6：宏与属性

符号	解释
<code>#[meta]</code>	外部属性
<code>#! [meta]</code>	内部属性
<code>\$ident</code>	宏替换
<code>\$ident:kind</code>	宏捕获
<code>\$(...)...</code>	宏重复
<code>ident!(...), ident!{...}, ident![...]</code>	宏调用

表 B-7 展示了写注释的符号。

表 B-7：注释

符号	注释
<code>//</code>	行注释
<code>//!</code>	内部行(hang)文档注释
<code>///</code>	外部行文档注释

符号	注释
<code>/*...*/</code>	块注释
<code>/*!...*/</code>	内部块文档注释
<code>/**...*/</code>	外部块文档注释

表 B-8 展示了出现在使用元组时的符号。

表 B-8：元组

符号	解释
<code>()</code>	空元组（亦称单元），即是字面值也是类型
<code>(expr)</code>	括号表达式
<code>(expr,)</code>	单一元素元组表达式
<code>(type,)</code>	单一元素元组类型
<code>(expr, ...)</code>	元组表达式
<code>(type, ...)</code>	元组类型
<code>expr(expr, ...)</code>	函数调用表达式；也用于初始化元组结构体 <code>struct</code> 以及元组枚举 <code>enum</code> 变体
<code>expr.0, expr.1, etc.</code>	元组索引

表 B-9 展示了使用大括号的上下文。

表 B-9：大括号

符号	解释
<code>{...}</code>	代码块表达式
<code>Type {...}</code>	结构体字面值

表 B-10 展示了使用方括号的上下文。

表 B-10：方括号

符号	解释
<code>[...]</code>	数组
<code>[expr; len]</code>	数组里包含 <code>len</code> 个 <code>expr</code>
<code>[type; len]</code>	数组里包含了 <code>len</code> 个 <code>type</code> 类型的对象
<code>expr[expr]</code>	集合索引。重载 ( <code>Index</code> , <code>IndexMut</code> )

符号	解释
<code>expr[..]</code> , <code>expr[a..]</code> , <code>expr[..b]</code> , <code>expr[a..b]</code>	集合索引, 也称为集合切片, 索引要实现以下特征中的其中一个: <code>Range</code> , <code>RangeFrom</code> , <code>RangeTo</code> 或 <code>RangeFull</code>

# 附录 C：表达式

在[语句与表达式](#)章节中，我们对表达式有过介绍，下面对这些常用表达式进行一一说明。

## 基本表达式

```
let n = 3;
let s = "test";
```

## if 表达式

```
fn main() {
    let var1 = 10;

    let var2 = if var1 >= 10 {
        var1
    } else {
        var1 + 10
    };

    println!("{}", var2);
}
```

通过 `if` 表达式将值赋予 `var2`。

你还可以在循环中结合 `continue`、`break` 来使用：

```
let mut v = 0;
for i in 1..10 {
    v = if i == 9 {
        continue
    } else {
        i
    }
}
println!("{}", v);
```

## if let 表达式

```
let o = Some(3);
let v = if let Some(x) = o {
    x
} else {
    0
};
```

## match 表达式

```
let o = Some(3);
let v = match o {
    Some(x) => x,
    _ => 0
};
```

## loop 表达式

```
let mut n = 0;
let v = loop {
    if n == 10 {
        break n
    }
    n += 1;
};
```

## 语句块

```
let mut n = 0;
let v = {
    println!("before: {}", n);
    n += 1;
    println!("after: {}", n);
    n
};
println!("{}", v);
```

## 附录 D：派生特征 trait

在本书的各个部分中，我们讨论了可应用于结构体和枚举定义的 `derive` 属性。被 `derive` 标记的对象会自动实现对应的默认特征代码，继承相应的功能。

在本附录中，我们列举了所有标准库存在的 `derive` 特征，每个特征覆盖了以下内容

- 该特征将会派生什么样的操作符和方法
- 由 `derive` 提供什么样的特征实现
- 实现特征对于类型意味着什么
- 你需要什么条件来实现该特征
- 特征示例

如果你希望不同于 `derive` 属性所提供的行为，请查阅 [标准库文档](#) 中每个特征的细节以了解如何手动实现它们。

除了本文列出的特征之外，标准库中定义的其它特征不能通过 `derive` 在类型上实现。这些特征不存在有意义的默认行为，所以由你负责以合理的方式实现它们。

一个无法被派生的特征例子是为终端用户处理格式化的 `Display`。你应该时常考虑使用合适的方法来为终端用户显示一个类型。终端用户应该看到类型的什么部分？他们会找出相关部分吗？对他们来说最关心的数据格式是什么样的？Rust 编译器没有这样的洞察力，因此无法为你提供合适的默认行为。

本附录所提供的可派生特征列表其实并不全面：库可以为其内部的特征实现 `derive`，因此除了本文列出的标准库 `derive` 之外，还有很多很多其它库的 `derive`。实现 `derive` 涉及到过程宏的应用，这在[宏章节](#)中有介绍。

## 用于开发者输出的 Debug

`Debug` 特征可以让指定对象输出调试格式的字符串，通过在 {} 占位符中增加 `:?`` 表明，例如 `println!("show you some debug info: {:?})", MyObject);`

`Debug` 特征允许以调试为目的来打印一个类型的实例，所以程序员可以在执行过程中看到该实例的具体信息。

例如，在使用 `assert_eq!` 宏时，`Debug` 特征是必须的。如果断言失败，这个宏就把给定实例的值打印出来，这样程序员就能看到两个实例为什么不相等。

## 等值比较的 PartialEq 和 Eq

PartialEq 特征可以比较一个类型的实例以检查是否相等，并开启了 `==` 和 `!=` 运算符的功能。

派生的 PartialEq 实现了 `eq` 方法。当 PartialEq 在结构体上派生时，只有所有的字段都相等时两个实例才相等，同时只要有任何字段不相等则两个实例就不相等。当在枚举上派生时，每一个成员都和其自身相等，且和其他成员都不相等。

例如，当使用 `assert_eq!` 宏时，需要比较一个类型的两个实例是否相等，则 PartialEq 特征是必须的。

Eq 特征没有方法，其作用是表明每一个被标记类型的值都等于其自身。Eq 特征只能应用于那些实现了 PartialEq 的类型，但并非所有实现了 PartialEq 的类型都可以实现 Eq。浮点类型就是一个例子：浮点数的实现表明两个非数字（`NaN`，not-a-number）值是互不相等的。

例如，对于一个 `HashMap<K, V>` 中的 key 来说，Eq 是必须的，这样 `HashMap<K, V>` 就可以知道两个 key 是否一样。

## 次序比较的 PartialOrd 和 Ord

PartialOrd 特征可以让一个类型的多个实例实现排序功能。实现了 PartialOrd 的类型可以使用 `<`、`>`、`<=` 和 `>=` 操作符。一个类型想要实现 PartialOrd 的前提是该类型已经实现了 PartialEq。

派生 PartialOrd 实现了 `partial_cmp` 方法，一般情况下其返回一个 `Option<Ordering>`，但是当给定的值无法进行排序时将返回 `None`。尽管大多数类型的值都可以比较，但一个无法产生顺序的例子是：浮点类型的非数字值。当在浮点数上调用 `partial_cmp` 时，`NaN` 的浮点数将返回 `None`。

当在结构体上派生时，PartialOrd 以在结构体定义中字段出现的顺序比较每个字段的值来比较两个实例。当在枚举上派生时，认为在枚举定义中声明较早的枚举项小于其后的枚举项。

例如，对于来自于 `rand` 包的 `gen_range` 方法来说，当在一个大值和小值指定的范围内生成一个随机值时，PartialOrd trait 是必须的。

对于派生了 Ord 特征的类型，任何两个该类型的值都能进行排序。Ord 特征实现了 `cmp` 方法，它返回一个 `Ordering` 而不是 `Option<Ordering>`，因为总存在一个合法的顺序。一个类型要想使用 Ord 特征，它必须要先实现 PartialOrd 和 Eq。当在结构体或枚举上派生时，`cmp` 方法和 PartialOrd 的 `partial_cmp` 方法表现是一致的。

例如，当在 `BTreeSet<T>`（一种基于有序值存储数据的数据结构）上存值时，Ord 是必须的。

## 复制值的 Clone 和 Copy

`Clone` 特征用于创建一个值的深拷贝 (deep copy) , 复制过程可能包含代码的执行以及堆上数据的复制。查阅 [通过 Clone 进行深拷贝](#) 获取有关 `Clone` 的更多信息。

派生 `Clone` 实现了 `clone` 方法, 当为整个的类型实现 `Clone` 时, 在该类型的每一部分上都会调用 `clone` 方法。这意味着类型中所有字段或值也必须实现了 `Clone` , 这样才能够派生 `Clone` 。

例如, 当在一个切片 (slice) 上调用 `to_vec` 方法时, `Clone` 是必须的。切片只是一个引用, 并不拥有其所包含的实例数据, 但是从 `to_vec` 中返回的 `Vector` 需要拥有实例数据, 因此, `to_vec` 需要在每个元素上调用 `clone` 来逐个复制。因此, 存储在切片中的类型必须实现 `Clone` 。

`Copy` 特征允许你通过只拷贝存储在栈上的数据来复制值(浅拷贝), 而无需复制存储在堆上的底层数据。查阅 [通过 Copy 复制栈数据](#) 的部分来获取有关 `Copy` 的更多信息。

实际上 `Copy` 特征并不阻止你在实现时使用了深拷贝, 只是, 我们不应该这么做, 毕竟遵循一个语言的惯例是很重要的。当用户看到 `Copy` 时, 潜意识就应该知道这是浅拷贝, 复制一个值会非常快。

当一个类型的内部字段全部实现了 `Copy` 时, 你就可以在该类型上派上 `Copy` 特征。一个类型如果要实现 `Copy` 它必须先实现 `Clone` , 因为一个类型实现 `Clone` 后, 就等于顺便实现了 `Copy` 。

总之, `Copy` 拥有更好的性能, 当浅拷贝足够的时候, 就不要使用 `Clone` , 不然会导致你的代码运行更慢, 对于[性能优化](#)来说, 一个很大的方面就是减少热点路径深拷贝的发生。

## 固定大小的值映射的 Hash

`Hash` 特征允许你使用 `hash` 函数把一个任意大小的实例映射到一个固定大小的值上。派生 `Hash` 实现了 `hash` 方法, 对某个类型进行 `hash` 调用, 其实就是对该类型下每个字段单独进行 `hash` 调用, 然后把结果进行汇总, 这意味着该类型下的所有的字段也必须实现了 `Hash` , 这样才能够派生 `Hash` 。

例如, 在 `HashMap<K, V>` 上存储数据, 存放 key 的时候, `Hash` 是必须的。

## 默认值的 Default

`Default` 特征会帮你创建一个类型的默认值。派生 `Default` 意味着自动实现了 `default` 函数。`default` 函数的派生实现调用了类型每部分的 `default` 函数, 这意味着类型中所有的字段也必须实现了 `Default` , 这样才能够派生 `Default` 。

`Default::default` 函数通常结合结构体更新语法一起使用, 这在第五章的 [结构体更新语法](#) 部分有讨论。可以自定义一个结构体的一小部分字段而剩余字段则使用 `..Default::default()` 设置为默认值。

例如，当你在 `option<T>` 实例上使用 `unwrap_or_default` 方法时，`Default` 特征是必须的。如果 `option<T>` 是 `None` 的话，`unwrap_or_default` 方法将返回 `T` 类型的 `Default::default` 的结果。

## 附录 E: prelude 模块

# 附录 F：Rust 版本发布

## Rust 版本说明

早在第一章，我们见过 `cargo new` 在 `Cargo.toml` 中增加了一些有关 `edition` 的元数据。本附录将解释其意义！

与其它语言相比，Rust 的更新迭代较为频繁（得益于精心设计过的发布流程以及 Rust 语言开发者团队管理）：

- 每 6 周发布一个迭代版本
- 2 - 3 年发布一个新的大版本：每一个版本会结合已经落地的功能，并提供一个清晰的带有完整更新文档和工具的功能包。新版本会作为常规的 6 周发布过程的一部分发布。

好处在于，可以满足不同的用户群体的需求：

- 对于活跃的 Rust 用户，他们总是能很快获取到新的语言内容，毕竟，尝鲜是技术爱好者的共同特点：)
- 对于一般的用户，`edition` 的发布会告诉这些用户：Rust 语言相比上次大版本发布，有了重大的改进，值得一看
- 对于 Rust 语言开发者，可以让他们的工作成果更快的被世人所知，不必锦衣夜行

在本文档编写时，Rust 已经有三个版本：Rust 2015、2018、2021。本书基于 `Rust 2021 edition` 编写。

`Cargo.toml` 中的 `edition` 字段表明代码应该使用哪个版本编译。如果该字段不存在，其默认为 `2021` 以提供后向兼容性。

每个项目都可以选择不同于默认的 `Rust 2021 edition` 的版本。这样，版本可能会包含不兼容的修改，比如新版本中新增的关键字可能会与老代码中的标识符冲突并导致错误。不过，除非你选择应用这些修改，否则旧代码依然能够被编译，即便你升级了编译器版本。

所有 Rust 编译器都支持任何之前存在的编译器版本，并可以链接任何支持版本的包。编译器修改只影响最初的解析代码的过程。因此，如果你使用 `Rust 2021` 而某个依赖使用 `Rust 2018`，你的项目仍旧能够编译并使用该依赖。反之，若项目使用 `Rust 2018` 而依赖使用 `Rust 2021` 亦可工作。

有一点需要明确：大部分功能在所有版本中都能使用。开发者使用任何 Rust 版本将能继续接收最新稳定版的改进。然而在一些情况，主要是增加了新关键字的时候，则可能出现了只能用于新版本的功能。只需切换版本即可利用新版本的功能。

请查看 [Edition Guide](#) 了解更多细节，这是一个完全介绍版本的书籍，包括如何通过 `cargo fix` 自动将代码迁移到新版本。

## Rust 自身开发流程

本附录介绍 Rust 语言自身是如何开发的以及这如何影响作为 Rust 开发者的你。

### 无停滞稳定

作为一个语言，Rust **十分** 注重代码的稳定性。我们希望 Rust 成为你代码坚实的基础，假如持续地有东西在变，这个希望就实现不了。但与此同时，如果不能实验新功能的话，在发布之前我们又无法发现其中重大的缺陷，而一旦发布便再也没有修改的机会了。

对于这个问题我们的解决方案被称为“无停滞稳定”（“stability without stagnation”），其指导性原则是：无需担心升级到最新的稳定版 Rust。每次升级应该是无痛的，并应带来新功能，更少的 Bug 和更快的编译速度。

### Choo, Choo! ~ 小火车发布流程启动

开发 Rust 语言是基于一个**火车时刻表**来进行的：所有的开发工作在 Master 分支上完成，但是发布就像火车时刻表一样，拥有不同的时间，发布采用的软件发布列车模型，被用于思科 IOS 等其它软件项目。

Rust 有三个**发布通道** (*release channel*)：

- Nightly
- Beta
- Stable (稳定版)

大部分 Rust 开发者主要采用稳定版通道，不过希望实验新功能的开发者可能会使用 nightly 或 beta 版。

如下是一个开发和发布过程如何运转的例子：假设 Rust 团队正在进行 Rust 1.5 的发布工作。该版本发布于 2015 年 12 月，这个版本和时间显然比较老了，不过这里只是为了提供一个真实的版本。Rust 新增了一项功能：一个 `master` 分支的新提交。每天晚上，会产生一个新的 nightly 版本。每天都是发布版本的日子，而这些发布由发布基础设施自动完成。所以随着时间推移，发布轨迹看起来像这样，版本一天一发：

```
nightly: * - - * - - *
```

每 6 周时间，是准备发布新版本的时候了！Rust 仓库的 `beta` 分支会从用于 `nightly` 的 `master` 分支产生。现在，有了两个发布版本：

```
nightly: * - - * - - *
          |
beta:      *
```

大部分 Rust 用户不会主要使用 `beta` 版本，不过在 CI 系统中对 `beta` 版本进行测试能够帮助 Rust 发现可能的回归缺陷（regression）。同时，每天仍产生 `nightly` 发布：

```
nightly: * - - * - - * - - * - - *
          |
beta:      *
```

比如我们发现了一个回归缺陷。好消息是在这些缺陷流入稳定发布之前还有一些时间来测试 `beta` 版本！`fix` 被合并到 `master`，为此 `nightly` 版本得到了修复，接着这些 `fix` 将 `backport` 到 `beta` 分支，一个新的 `beta` 发布就产生了：

```
nightly: * - - * - - * - - * - - * - - *
          |
beta:      * - - - - - - - - *
```

第一个 `beta` 版的 6 周后，是发布稳定版的时候了！`stable` 分支从 `beta` 分支生成：

```
nightly: * - - * - - * - - * - - * - - * - *
          |
beta:      * - - - - - - - - *
          |
stable:   *
```

好的！Rust 1.5 发布了！然而，我们忘了些东西：因为又过了 6 周，我们还需发布 **新版** Rust 的 `beta` 版，Rust 1.6。所以从 `beta` 分支生成 `stable` 分支后，新版的 `beta` 分支也再次从 `nightly` 生成：

```
nightly: * - - * - - * - - * - - * - - * - *
          |
beta:      * - - - - - - - - *
          |
stable:   *
```

这被称为“train model”，因为每 6 周，一个版本“离开车站”（“leaves the station”），不过从 `beta` 通道到达稳定通道还有一段旅程。

Rust 每 6 周发布一个版本，如时钟般准确。如果你知道了某个 Rust 版本的发布时间，就可以知道下个版本的时间：6 周后。每 6 周发布版本的一个好的方面是下一班车会来得更快。如果特定版本碰巧缺失某个

功能也无需担心：另一个版本很快就会到来！这有助于减少因临近发版时间而偷偷释出未经完善的功能的压力。

多亏了这个过程，你总是可以切换到下一版本的 Rust 并验证是否可以轻易的升级：如果 beta 版不能如期工作，你可以向 Rust 团队报告并在发布稳定版之前得到修复！beta 版造成的破坏是非常少见的，不过 `rustc` 也不过是一个软件，可能会存在 Bug。

## 不稳定功能

这个发布模型中另一个值得注意的地方：不稳定功能（unstable features）。Rust 使用一个被称为“功能标记”（“feature flags”）的技术来确定给定版本的某个功能是否启用。如果新功能正在积极地开发中，其提交到了 `master`，因此会出现在 nightly 版中，不过会位于一个 **功能标记** 之后。作为用户，如果你希望尝试这个正在开发的功能，则可以在源码中使用合适的标记来开启，不过必须使用 nightly 版。

如果使用的是 beta 或稳定版 Rust，则不能使用任何功能标记。这是在新功能被宣布为永久稳定之前获得实用价值的关键。这既满足了希望使用最尖端技术的同学，那些坚持稳定版的同学也知道其代码不会被破坏。这就是无停滞稳定。

本书只包含稳定的功能，因为还在开发中的功能仍可能改变，当其进入稳定版时肯定会与编写本书的时候有所不同。你可以在网上获取 nightly 版的文档。

## Rustup 和 Rust Nightly 的职责

### 安装 Rust Nightly 版本

Rustup 使得改变不同发布通道的 Rust 更为简单，其在全局或分项目的层次工作。其默认会安装稳定版 Rust。例如为了安装 nightly：

```
$ rustup install nightly
```

你会发现 `rustup` 也安装了所有的 **工具链**（*toolchains*，Rust 和其相关组件）。如下是一位作者的 Windows 计算机上的例子：

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

## 在指定目录使用 Rust Nightly

如你所见，默认是稳定版。大部分 Rust 用户在大部分时间使用稳定版。你可能也会这么做，不过如果你关心最新的功能，可以为特定项目使用 nightly 版。为此，可以在项目目录使用 `rustup override` 来设置当前目录 `rustup` 使用 nightly 工具链：

```
$ cd ~/projects/needs-nightly  
$ rustup override set nightly
```

现在，每次在 \*~/需要 nightly 的项目/\*下(在项目的根目录下，也就是 `Cargo.toml` 所在的目录) 调用 `rustc` 或 `cargo`，`rustup` 会确保使用 nightly 版 Rust。在你有很多 Rust 项目时大有裨益！

## RFC 过程和团队

那么你如何了解这些新功能呢？Rust 开发模式遵循一个 **Request For Comments (RFC) 过程**。如果你希望改进 Rust，可以编写一个提议，也就是 RFC。

任何人都可以编写 RFC 来改进 Rust，同时这些 RFC 会被 Rust 团队评审和讨论，他们由很多不同分工的子团队组成。这里是 [Rust 官网](#) 上所有团队的总列表，其包含了项目中每个领域的团队：语言设计、编译器实现、基础设施、文档等。各个团队会阅读相应的提议和评论，编写回复，并最终达成接受或回绝功能的一致。

如果功能被接受了，在 Rust 仓库会打开一个 issue，人们就可以实现它。实现功能的人可能不是最初提议功能的人！当实现完成后，其会合并到 `master` 分支并位于一个特性开关 (feature gate) 之后，正如 [不稳定功能](#) 部分所讨论的。

在稍后的某个时间，一旦使用 nightly 版的 Rust 团队能够尝试这个功能了，团队成员会讨论这个功能在 nightly 中运行的情况，并决定是否应该进入稳定版。如果决定继续推进，特性开关会移除，然后这个功能就被认为是稳定的了！乘着“发布的列车”，最终在新的稳定版 Rust 中出现。

# 附录 G: Rust 更新版本列表

本目录包含了 Rust 历次版本更新的重要内容解读，需要注意，每个版本实际更新的内容要比这里记录的更多，全部内容请访问每节开头的官方链接查看。

# Rust 新版解读 | 1.58 | 重点: 格式化字符串捕获环境中的值

众所周知, Rust 小版本发布非常频繁, 6 周就发布一次, 因此通常不会有特别值得普通用户关注的内容, 但是这次 1.58 版本不同, 新增了(stable 化了)一个非常好用的功能: **在格式化字符串时捕获环境中的值。**

---

Rust 1.58 官方 release doc: [Announcing Rust 1.58.0 | Rust Blog](#)

---

## 在格式化字符串时捕获环境中的值

在以前, 想要输出一个函数的返回值, 你需要这么做:

```
fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let p = get_person();
    println!("Hello, {}!", p);                      // implicit position
    println!("Hello, {}!", p);                      // explicit index
    println!("Hello, {person}!", person = p);
}
```

问题倒也不大, 但是一旦格式化字符串长了后, 就会非常冗余, 而在 1.58 后, 我们可以这么写:

```
fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let person = get_person();
    println!("Hello, {person}!");
}
```

是不是清晰、简洁了很多? 甚至还可以将环境中的值用于格式化参数:

```
let (width, precision) = get_format();
for (name, score) in get_scores() {
    println!("{}: {:.width$.precision$}", name, score);
}
```

但也有局限，它只能捕获普通的变量，对于更复杂的类型（例如表达式），可以先将它赋值给一个变量或使用以前的 `name = expression` 形式的格式化参数。

目前除了 `panic!` 外，其它接收格式化参数的宏，都可以使用新的特性。对于 `panic!` 而言，如果还在使用 2015 版本 或 2018 版本 版本，那 `panic!("{}")` 依然会被当成正常的字符串来处理，同时编译器会给予 `warn` 提示。而对于 2021 版本，则可以正常使用：

```
fn get_person() -> String {
    String::from("sunface")
}
fn main() {
    let person = get_person();
    panic!("Hello, {person}!");
}
```

输出：

```
thread 'main' panicked at 'Hello, sunface!', src/main.rs:6:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

## 比 `unwrap` 更危险的 `unwrap_unchecked`

在 1.58 中为 `Option` 和 `Result` 新增了 `unwrap_unchecked` 方法，与 `unwrap` 遇到错误或者空值直接 `panic` 不同，`unwrap_unchecked` 遇到错误时处理方式糟糕的多：

```
fn get_num() -> Option<i32> {
    None
}
fn main() {
    unsafe {
        let n = get_num().unwrap_unchecked();
    }
}
```

输出如下：

```
zsh: segmentation fault cargo run
```

嗯，段错误了，对比下 `panic`，有一种泪流满面的冲动：我要这不安全的方法何用？

其实，还真有些用：

- 想要较小的可执行文件时（嵌入式，WASM 等），该方法就可以大显身手。因为 `panic` 会导致二进制可执行文件变大不少
- 它还可以提高一点性能，因为编译器可能无法优化掉 `unwrap` 的指令分支，虽然它只会增加区区几条分支预测指令

# Rust 新版解读 | 1.59 | 重点: 内联汇编、解构式赋值

Rust 团队于今天凌晨( 2022-02-25 )发布了最新的 1.59 版本，其中最引人瞩目的特性应该就是支持在代码中内联汇编了，一起来看看。

## 内联汇编( inline assembly )

该特性对于需要底层控制的应用非常有用，例如想要控制底层执行、访问特定的机器指令等。

例如，如果目标平台是 x86-64 时，你可以这么写：

```
use std::arch::asm;

// 使用 shifts 和 adds 实现 x 乘以 6
let mut x: u64 = 4;
unsafe {
    asm!(
        "mov {tmp}, {x}",
        "shl {tmp}, 1",
        "shl {x}, 2",
        "add {x}, {tmp}",
        x = inout(reg) x,
        tmp = out(reg) _,
    );
}
assert_eq!(x, 4 * 6);
```

大家发现没，这里的格式化字符串的使用方式跟我们平时的 `println!`、`format!` 并无区别，除了 `asm!` 之外，`global_asm!` 宏也可以这么使用。

内联汇编中使用的汇编语言和指令取决于相应的机器平台，截至目前，Rust 支持以下平台的内联汇编：

- x86 和 x86-64
- ARM
- AArch64
- RISC-V

如果大家希望深入了解，可以看官方的 [Reference](#) 文档，同时在 [Rust Exercise](#) 中提供了更多的示例(目前正在翻译中..)。

## 解构式赋值( Destructuring assignments)

现在你可以在赋值语句的左式中使用元组、切片和结构体模式了。

```
let (a, b, c, d, e);  
  
(a, b) = (1, 2);  
[c, .., d, _) = [1, 2, 3, 4, 5];  
Struct { e, .. } = Struct { e: 5, f: 3 };  
  
assert_eq!([1, 2, 1, 4, 5], [a, b, c, d, e]);
```

这种使用方式跟 `let` 保持了一致性，但是需要注意，使用 `+=` 的赋值语句还不支持解构式赋值。

## const 泛型

### 为参数设置默认值

现在我们可以为 `const` 泛型参数设置默认值：

```
struct ArrayStorage<T, const N: usize = 2> {  
    arr: [T; N],  
}  
  
impl<T> ArrayStorage<T> {  
    fn new(a: T, b: T) -> ArrayStorage<T> {  
        ArrayStorage {  
            arr: [a, b],  
        }  
    }  
}
```

### 取消参数顺序的限制

在之前版本中，类型参数必须要在所有的 `const` 泛型参数之前，现在，这个限制被放宽了，例如你可以这样交替排列它们：

```
fn cartesian_product<
    T, const N: usize,
    U, const M: usize,
    V, F
>(a: [T; N], b: [U; M], f: F) -> [[V; N]; M]
where
    F: FnMut(&T, &U) -> V
{
    // ...
}
```

## 缩小二进制文件体积：删除 debug 信息

对于受限的环境来说，缩小编译出的二进制文件体积是非常重要的。

以往我们可以在二进制文件被创建后，手动的来完成。现在 cargo 和 rustc 支持在链接( linked )后就删除 debug 信息，在 `Cargo.toml` 中新增以下配置：

```
[profile.release]
strip = "debuginfo"
```

以上配置会将 `release` 二进制文件中的 debug 信息移除。你也可以使用 `"symbols"` 或 `true` 选项来移除所有支持的 `symbol` 信息。

根据 reddit 网友的测试，如果使用了 `strip = true`，那编译后的体积将大幅减少(50% 左右)：

- 先使用 `lto = true` : 4,397,320 bytes
- 再使用 `strip = true` : 2,657,304 bytes
- 最后 `opt-level = "z"` : 1,857,680 bytes

如果是 WASM，还可以使用以下配置进一步减少体积：

```
[package.metadata.wasm-pack.profile.release]
wasm-opt = ['-Os']
```

[github 上一个开源仓库](#)也证明了这一点，总体来看，这个配置的效果是非常显著的！

## 默认关闭增量编译

1.59.0 版本默认关闭了增量编译的功能（你可以通过环境变量显式地启用：`RUSTC_FORCE_INCREMENTAL=1`），这会降低已知 Bug #94124 的影响，该 Bug 会导致增量编译过程中的反序列化错误和 `panic`。

不过大家也不用担心，这个 Bug 会在 1.60.0 版本修复，也就是 6 周后，增量编译会重新设置为默认开启，如果没有意外的话：）

## 稳定化的 API 列表

一些方法和特征实现现在已经可以 stable 中使用，具体见[官方发布说明](#)

# Rust 新版解读 | 1.60 | 重点: 查看 Cargo 构建耗时详情、Cargo Feature 增加新语法

原文地址: <https://blog.rust-lang.org/2022/04/07/Rust-1.60.0.html>

通过 `rustup` 安装的同学可以使用以下命令升级到 1.60 版本:

```
$ rustup update stable
```

## 基于源码的代码覆盖

`rustc` 新增了基于 LLVM 的代码覆盖率测量，想要测试的同学可以通过以下方式重新构建你的项目:

```
$ RUSTFLAGS="-C instrument-coverage" cargo build
```

运行新生成的可执行文件将在当前目录下产生一个 `default.profraw` 文件(路径和文件名可以通过环境变量进行[覆盖](#))。

`llvm-tools-preview` 组件包含了 `llvm-propdata`，可以用于处理和合并原生的测量结果输出(测量区域执行数)。

`llvm-cov` 用于报告生成，它将 `llvm-propdata` 处理后的输出跟二进制可执行文件自身相结合，对于前者大家可能好理解，但是为何要跟后者可执行文件相结合呢？原因在于可执行文件中嵌入了一个从计数器到实际源代码单元的映射。

```
rustup component add llvm-tools-preview
$(rustc --print sysroot)/lib/rustlib/x86_64-unknown-linux-gnu/bin/llvm-propdata merge
-sparse default.profraw -o default.propdata
$(rustc --print sysroot)/lib/rustlib/x86_64-unknown-linux-gnu/bin/llvm-cov show -
Xdemangler=rustfilt target/debug/coverage-testing \
-instr-profile=default.propdata \
-show-line-counts-or-regions \
-show-instantiations
```

基于一个简单的 `hello world` 可执行文件，执行以上命令就可以获得如下带有标记的结果：

```
1|     fn main() {  
2|         println!("Hello, world!");  
3|     }
```

从结果中可以看出：每一行代码都已经被成功覆盖。

如果大家还想要了解更多，可以看下[官方的 rustc 文档](#)。目前来说，基准功能已经稳定了，并将以某种形式存在于未来所有的 Rust 发布版本中。但输出格式和产生这些输出的 LLVM 工具可能依然会发生变化，基于此，大家在使用时需要确保 `llvm-tools-preview` 和 `rustc` ( 用于编译代码的 ) 使用了相同的版本。

## 查看 Cargo 构建耗时

新版本中，以下命令已经可以正常使用了：

```
$ cargo build --timings  
Compiling hello-world v0.1.0 (hello-world)  
    Timing report saved to target/cargo-timings/cargo-timing-20220318T174818Z.html  
Finished dev [unoptimized + debuginfo] target(s) in 0.98s
```

此命令会生成一个 `cargo build` 的耗时详情报告，除了上面提到的路径外，报告还会被拷贝到 `target/cargo-timings/cargo-timing.html`。这里是一个[在线示例](#)。该报告在你需要提升构建速度时会非常有用，更多的信息请[查看文档](#)。

## Cargo Feature 的新语法

---

关于 Cargo Features，强烈推荐大家看看 [Cargo 使用指南](#)，可能是目前最好的中文翻译版本。

---

新版本为 Cargo Features 引入了两个新的语法：命名空间 (Namespaced) 和弱依赖，它们可以让 features 跟可选依赖进行更好的交互。

Cargo 支持[可选依赖](#)已经很久了，例如以下代码所示：

```
[dependencies]  
jpeg-decoder = { version = "0.1.20", default-features = false, optional = true }  
  
[features]  
# 通过开启 jpeg-decoder 依赖的 "rayon` fature, 来启用并行化处理  
parallel = ["jpeg-decoder/rayon"]
```

这个例子有两点值得注意：

- 可选依赖 jpeg-decoder 隐式地定义了一个同名的 feature，当启用 jpeg-decoder feature 时将同时启用 jpeg-decoder
- "jpeg-decoder/rayon" 语法会启用 jpeg-decoder 依赖，并且还会启用 jpeg-decoder 依赖的 rayon feature

而命名空间正是为了处理第一个问题而出现的。新版本中，我们可以在 [features] 中使用 dep: 前缀来显式地引用一个可选的依赖。再无需像第一点一样：先隐式的将可选依赖暴露为一个 feature，再通过 feature 来启用它。

这样一来，我们将能更好的定义可选依赖所对应的 feature，包括将可选依赖隐藏在一个更具描述性的 feature 名称后面。

弱依赖用于处理第二点：根据第二点，optional-dependency/feature-name 必定会启用 optional-dependency 这个可选依赖。然而在一些场景中，我们只希望在其它 features 已经启用了可选依赖 optional-dependency 时才去启用 feature-name 这个 feature。

从 1.60 开始，我们可以使用 "package-name?/feature-name" 这种带有 ? 形式的语法：只有当其它项已经启用了可选依赖 package-name 的情况下才去开启给定的 feature feature-name。

---

译者注：简单来说，要启用 feature 必须需要别人先启用了其前置的可选依赖，再也无法像之前的第二点一样，既能开启可选依赖，又能启用 feature。

---

例如，我们希望为自己的库增加一些序列化功能，它需要开启某个可选依赖中的指定 feature，可以这么做：

```
[dependencies]
serde = { version = "1.0.133", optional = true }
rgb = { version = "0.8.25", optional = true }

[features]
serde = ["dep:serde", "rgb?/serde"]
```

这里定义了以下关系：

1. 启用 serde feature 将启用可选的 serde 依赖
2. 只有当 rgb 依赖在其它地方已经被启用后，此处才能启用 rgb 的 serde feature

## 增量编译重启开启

在 [1.59 更新说明中](#)，我们有提到因为某些问题，增量编译被默认关闭了，现在官方修复了其中一些，并且确认目前的状态不会再影响用户的使用，因此在 1.60 版本中，增量编译又重新默认开启了。

## Instant 单调性保证

译者注：Instant 可以获取当前的时间，因此保证其单调增长是非常重要的，例如 uuid 的生成往往依赖于时间戳的单调增长，一旦时间回退，就可能出现 uuid 重复的情况。

在目前所有的平台上，Instant 会去尝试使用系统提供的 API 来保证单调性行为( 目前主要针对 tier 1 的平台 )。然而在实际场景中，这种单调性偶尔会因为硬件、虚拟化或操作系统bug 等原因而失效。

为了解决这些失效或是平台没有提供 API 的情况，`Instant::duration_since`, `Instant::elapsed` 和 `Instant::sub` 现在饱和为零( 这里不太好翻译，原文是 now saturate to zero，大概意思是非负？ )。而在老版本中，这种时间回退的情况会导致 panic。

`Instant::checked_duration_since` 也可以用于检测和处理单调性失败或 `Instants` 的减法顺序不正确的情况。

但是目前的解决方法会遮掩一些错误的发生，因此在未来版本中，Rust 可能会重新就某些场景引入 panic 机制。

在 1.60 版本前，单调性主要通过标准库的互斥锁 Mutex 或原子性 atomic 来保证，但是在 `Instant::now()` 调用频繁时，可能会导致明显的性能问题。

# Rust 新版解读 | 1.61 | 重点: 自定义 main 函数 ExitCode、const fn 增强、为锁定的 stdio 提供静态句柄

---

原文链接: <https://blog.rust-lang.org/2022/05/19/Rust-1.61.0.html> 翻译 by : AllanDowney

---

通过 `rustup` 安装的同学可以使用以下命令升级到 1.61 版本:

```
$ rustup update stable
```

## 支持自定义 main 函数 ExitCode

一开始, Rust `main` 函数只能返回单元类型 `()` (隐式或显式), 总是指示成功的退出状态, 如果您要您想要其它的, 必须调用 `process::exit(code)`。从 Rust 1.26 开始, `main` 允许返回一个 `Result`, 其中 `Ok` 转换为 C `EXIT_SUCCESS`, `Err` 转换为 `EXIT_FAILURE` (也调试打印错误)。在底层, 这些返回类型统一使用不稳定的 `Termination` 特征。

在此版本中, 最终稳定了 `Termination` 特征, 以及一个更通用的 `ExitCode` 类型, 它封装了特定于平台的返回类型。它具有 `SUCCESS` 和 `FAILURE` 常量, 并为更多任意值实现 `From<u8>`。也可以为您自己的类型实现 `Termination` 特征, 允许您在转换为 `ExitCode` 之前定制任何类型的报告。

例如, 下面是一种类型安全的方式来编写 `git bisect` 运行脚本的退出代码:

```

use std::process::{ExitCode, Termination};

#[repr(u8)]
pub enum GitBisectResult {
    Good = 0,
    Bad = 1,
    Skip = 125,
    Abort = 255,
}

impl Termination for GitBisectResult {
    fn report(self) -> ExitCode {
        // Maybe print a message here
        ExitCode::from(self as u8)
    }
}

fn main() -> GitBisectResult {
    std::panic::catch_unwind(|| {
        todo!("test the commit")
    }).unwrap_or(GitBisectResult::Abort)
}

```

## const fn 增强

这个版本稳定了几个增量特性，以支持 const 函数的更多功能：

- fn 指针的基本处理：现在可以在 const fn 中创建、传递和强制转换函数指针。例如，在为解释器构建编译时函数表时，这可能很有用。但是，仍然不允许调用 fn 指针。
- 特征约束：现在可以将特征约束写在 const fn 的泛型参数上，如 T: Copy，以前只允许 Sized。
- dyn Trait 类型：类似地，const fn 现在可以处理特征对象 dyn Trait。
- impl Trait 类型：const fn 的参数和返回值现在可以是不透明的 impl Trait 类型。

注意，特征特性还不支持在 const fn 中调用这些特征的方法。

## 为锁定的 stdio 提供静态句柄

三种标准 I/O 流—— Stdin 、 Stdout 和 Stderr ——都有一个 锁(&self)，允许对同步读写进行更多控制。但是，它们返回的锁守卫具有从 &self 借来的生命周期，因此它们被限制在原始句柄的范围

内。这被认为是一个不必要的限制，因为底层锁实际上是在静态存储中，所以现在守卫返回一个'static 生命期，与句柄断开连接。

例如，一个常见的错误来自于试图获取一个句柄并将其锁定在一个语句中：

```
// error[E0716]: temporary value dropped while borrowed
let out = std::io::stdout().lock();
//          ^^^^^^^^^^^^^^ - temporary value is freed at the end of this
statement
//          |
//          creates a temporary which is freed while still in use
```

现在锁守卫是'static，而不是借用那个临时的，所以这个可以正常工作！

# Rust 新版解读 | 1.62 | 重点: Cargo add, #[default] 枚举变量, Linux 上更薄更快的 Mutex, 裸机 x86\_64 构架

---

原文地址: <https://blog.rust-lang.org/2022/06/30/Rust-1.62.0.html> 翻译 by : AllanDowney

---

通过 `rustup` 安装的同学可以使用以下命令升级到 1.62 版本:

```
$ rustup update stable
```

## Cargo add

现在可以使用 `cargo add` 直接从命令行添加新的依赖项。此命令支持指定功能和版本。它还可以用来修改现有的依赖关系。

例如:

```
$ cargo add log
$ cargo add serde --features derive
$ cargo add nom@5
```

有关更多信息, 请参阅 [cargo 文档](#)。

## # [default] 枚举变量

如果指定枚举默认变量, 现在可以使用 `#[derive(Default)]`。例如, 到目前为止, 您必须手动为此枚举写入 `Default`:

```
#[derive(Default)]
enum Maybe<T> {
    #[default]
    Nothing,
    Something(T),
}
```

到目前为止，只允许将“单元”变量（没有字段的变量）标记为#[default]。RFC 中提供了有关此功能的更多信息。

## Linux 上更薄更快的 Mutex

以前，Linux 上的 `pthreads` 库支持 `Mutex`、`Condvar` 和 `RwLock`。`pthreads` 锁 支持比 Rust API 本身更多的功能，包括运行时配置，并且设计用于比 Rust 提供的静态保证更少的语言中。

例如，`Mutex` 实现是 40 个字节，不能被移动(move)。这迫使标准库在后台为使用 `pthreads` 的平台的每个新 `Mutex` 分配一个 `Box`。

现在 Rust 的标准库在 Linux 上提供了这些锁的原始 `futex` 实现，它非常轻量级，不需要额外分配。在 1.62.0 中，`Mutex` 在 Linux 上的内部状态只需要 5 个字节，尽管在未来的版本中可能会发生变化。

这是提高 Rust 的锁类型效率的长期努力的一部分，包括以前在 Windows 上的改进，如取消绑定其原语。您可以在[跟踪问题](#)中了解更多有关这方面的信息。

## 裸机 x86\_64 构架

现在更容易为 `x86_64` 构建无操作系统的二进制文件，例如在编写内核时。`x86_64-unknown-none` 构架已升级到第 2 层，可以用 `rustup` 安装。

```
$ rustup target add x86_64-unknown-none
$ rustc --target x86_64-unknown-none my_no_std_program.rs
```

您可以在[Embedded Rust book](#) 中阅读更多关于使用 `no_std` 进行开发的信息。

# Rust 新版解读 | 1.63 | 重点: Scoped threads

Rust 1.63 官方 release doc: [Announcing Rust 1.63.0](#) | [Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.63 版本:

```
$ rustup update stable
```

## 区域线程 Scoped threads

Rust 从 1.0 版本起，就可以使用 `std::thread::spawn` 来创建一个线程，但是这个函数要求了其生成的线程必须拥有任何传递进去的参数的所有权，也就是说你不能把引用数据传递进去。在一些线程会在方法末尾退出的情况下（通常使用 `join()` 方法），这个严格的约束显得不必要，在此之前也通常使用 `Arc` 包裹数据的方法来妥协。

随着 1.63 版本的推出，标准库新增了**区域线程**，允许在区域 `scope` 内创建使用当前调用栈内引用数据的线程。`std::thread::scope` 的 API 保证其中创建的线程会在自身返回前推出，也就允许安全的借用数据。看下面的例子，在 `scope` 内创建两个线程来，分别借用了数据：

```
let mut a = vec![1, 2, 3];
let mut x = 0;

std::thread::scope(|s| {
    s.spawn(|| {
        println!("hello from the first scoped thread");
        // 可以借用变量 `a`
        dbg!(&a);
    });
    s.spawn(|| {
        println!("hello from the second scoped thread");
        // 没有其它线程在使用，所以也可以可变借用 `x`
        x += a[0] + a[2];
    });
    println!("hello from the main thread");
});

// Scope 退出后，可以继续修改、访问变量。
a.push(4);
assert_eq!(x, a.len());
```

## Rust 对原始文件描述符/句柄的所有权

之前 Rust 代码在使用平台相关 API，涉及到文件描述符 (file descriptor on unix) 或句柄 (handles on windows) 的时候，都是直接使用对应的描述符（比如，`c_int alias RawFd`）。因此类型系统无法判断 API 是会获取文件描述符的所有权，还是仅仅借用它。

现在，Rust 提供了封装类型诸如 `BorrowedFd` 和 `OwnedFd`。这些封装类型都标记为了 `# [repr(transparent)]`，意味着 `extern "C"` 绑定下也可以直接使用这些类型来编码所有权语义。完整的封装类型参见原文下的 [stabilized apis in 1.63](#)

## Mutex, RwLock, Condvar 作为静态变量

`Condvar::new`, `Mutex::new` 和 `RwLock::new` 可以在 `const` 上下文里被调用了，不必再使用 `lazy_static` 库来写全局静态的 `Mutex`, `RwLock`, `Condvar` 了。

## Turbofish 可用于含有 `impl Trait` 的泛型函数上

诸如 `fn foo<T>(value: T, f: impl Copy)` 的函数签名，使用 Turbofish `foo::<u32>(3,3)` 来指定 `T` 的具体类型会出现编译错误：

```
error[E0632]: cannot provide explicit generic arguments when `impl Trait` is used in
argument position
--> src/lib.rs:4:11
 |
4 |     foo::<u32>(3, 3);
|         ^^^ explicit generic argument not allowed
|
= note: see issue #83701 <https://github.com/rust-lang/rust/issues/83701> for more
information
```

1.63里这个限制被放松了，显式泛型类型可以用 Turbofish 来指定了。不过 `impl Trait` 参数，尽管已经脱糖(desugared)成了泛型，因为还是不透明的所以无法通过 Turbofish 指定。

## 完成了 Non-lexical-lifetime 的生命周期检查器的迁移

1.63 的rustc，完全删除了之前的词法借用检查，完全启用了新的 NLL 借用检查器。这不会对编译结果有任何变化，但对编译器的借用错误检查有优化效果。

如果对NLL不了解，在本书[引用与借用](#)一章里有介绍。

或者看官方博客的介绍[NLL](#)

更详细内容可以看原博客[blog](#)

# Rust 新版解读 | 1.64 | 重点: IntoFuture , Cargo 优化

Rust 1.64 官方 release doc: Announcing Rust 1.64.0 | Rust Blog

通过 [rustup](#) 安装的同学可以使用以下命令升级到 1.64 版本：

```
$ rustup update stable
```

## 使用 IntoFuture 增强 .await

1.64 稳定了 `IntoFuture` trait，不同于用在 `for ... in ...` 的 `IntoIterator` trait，`IntoFuture` 增强了 `.await` 关键字。现在 `.await` 可以 `await` 除了 `futures` 外，还可以 `await` 任何实现了 `IntoFuture` trait 并经此转换成 `Future` 的对象。这可以让你的 api 对用户更加优化。

举一个用在网络存储供应端的例子：

```
pub struct Error { ... }

pub struct StorageResponse { ... }:
pub struct StorageRequest(bool);

impl StorageRequest {
    /// 实例化一个 `StorageRequest` 
    pub fn new() -> Self { ... }

    /// 是否开启 debug 模式
    pub fn set_debug(self, b: bool) -> Self { ... }

    /// 发送请求并接受回复
    pub async fn send(self) -> Result<StorageResponse, Error> { ... }
}
```

通常地使用方法可能类似如下代码：

```
let response = StorageRequest::new() // 1. 实例化
    .set_debug(true) // 2. 设置一些选项
    .send() // 3. 构造 future
    .await?; // 4. 执行 future，传递 error
```

这个代码已经不错了，不过 1.64 后可以做的更好。使用 `IntoFuture`，把第三步的“构造 future”和第四步的“执行 future”合并到一个步骤里：

```
let response = StorageRequest::new() // 1. 实例化
    .set_debug(true) // 2. 设置一些选项
    .await?; // 3. 构造并执行 future，传递 error
```

想要实现上面的效果，我们需要给 `StorageRequest` 实现 `IntoFuture` trait。`IntoFuture` 需要确定好要返回的 future，可以用下面的代码来实现：

```
// 首先需要引入一些必须的类型
use std::pin::Pin;
use std::future::{Future, IntoFuture};

pub struct Error { ... }
pub struct StorageResponse { ... }
pub struct StorageRequest(bool);

impl StorageRequest {
    /// 实例化一个 `StorageRequest`
    pub fn new() -> Self { ... }
    /// 是否开启 debug 模式
    pub fn set_debug(self, b: bool) -> Self { ... }
    /// 发送请求并接受回复
    pub async fn send(self) -> Result<StorageResponse, Error> { ... }
}

// 新的实现内容
// 1. 定义好返回的 future 类型
pub type StorageRequestFuture = Pin<Box<dyn Future<Output = Result<StorageResponse, Error>> + Send + 'static>>
// 2. 给 `StorageRequest` 实现 `IntoFuture`
impl IntoFuture for StorageRequest {
    type IntoFuture = StorageRequestFuture;
    type Output = <StorageRequestFuture as Future>::Output;
    fn into_future(self) -> Self::IntoFuture {
        Box::pin(self.send())
    }
}
```

这确实需要多写一点实现代码，不过可以给用户提供一个更简单的 api。

未来，Rust 异步团队 希望能够通过给类型别名提供 `impl Trait Type Alias Impl Trait`，来简化定义 futures 实现 `IntoFuture` 的代码；再想办法移除 `Box` 来提升性能。

## core 和 alloc 中和 C 语言兼容的 FFI 类型

当调用 C-ABI 或者调用 C-ABI 的时候，Rust 代码通常会使用诸如 `c_uint` 或者 `c_ulong` 的类型别名来匹配目标语言里的对应类型。

在此之前，这些类型别名仅在 `std` 里可用，而在嵌入式或者其它仅能使用 `core` 或者 `alloc` 的场景下无法使用。

1.64 里在 `core::ffi` 里提供了所有 `c_*` 的类型别名，还有 `core::ffi::CStr` 对应 C 的字符串，还有仅用 `alloc` 库情况下可以用 `alloc::ffi::CString` 来对应 C 的字符串。

## 可以通过 rustup 来使用 rust-analyzer

`rust-analyzer` 现在被加进 Rust 工具集里了。这让在各平台上下载使用 `rust-analyzer` 更加方便。通过 [rustup component](#) 来安装：

```
rustup component add rust-analyzer
```

目前，使用 `rustup` 安装的版本，需要这样启用：

```
rustup run stable rust-analyzer
```

下一次 `rustup` 的发布本把会提供一个内置的代理，来运行对应版本的 `rust-analyzer`。

## Cargo 优化，workspace 继承和多目标构建

当在一个 Cargo workspace 里管理多个相关的库/产品时，现在可以避免在多个库里使用相同的字段值了，比如相同的版本号，仓库链接，`rust-version`。在更新的时候也可以更容易地保持这些信息的一致性。更多细节可以参考：

- [workspace.package](#)
- [workspace.dependencies](#)
- ["Inheriting a dependency from a workspace"](#)

另外在构建多个目标地时候，现在可以直接传递多个 `--target` 选项给 `cargo build` 来一次性编译所有目标。也可以在 `.cargo/config.toml` 里设置一个 `build.target` 的 array 来改变默认构建时的对象。

## 稳定API && Others

更多稳定API列表和其它更新内容, 请参考原文最后 [stabilized-apis](#)

# Rust 新版解读 | 1.65 | 重点: 泛型关联类型, 新绑定语法!

Rust 1.65 官方 release doc: [Announcing Rust 1.65.0](#) | [Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.65 版本:

```
$ rustup update stable
```

## 泛型关联类型 Generic associated types (GATs)

关联类型 (associated types) 里现在可以加上生命周期、类型、`const` 泛型了, 类似于:

```
trait Foo {
    type Bar<'x>;
}
```

三言两语说不清这个变化的好处, 看几个例子来感受一下:

```
/// 一个类似于 `Iterator` 的 trait, 可以借用 `Self`。
trait LendingIterator {
    type Item<'a> where Self: 'a;

    fn next<'a>(&'a mut self) -> Option<Self::Item<'a>>;
}

/// 可以给智能指针类型, 比如 `Rc` 和 `Arc` 实现的 trait, 来实现指针类型的泛用性
trait PointerFamily {
    type Pointer<T>: Deref<Target = T>;

    fn new<T>(value: T) -> Self::Pointer<T>;
}

/// 允许借用数组对象, 对不需要连续存储数据的固定长度数组类型很有用
trait BorrowArray<T> {
    type Array<'x, const N: usize> where Self: 'x;

    fn borrow_array<'a, const N: usize>(&'a self) -> Self::Array<'a, N>;
}
```

泛型关联类型十分通用，能够写出许多之前无法实现的模式。更多的信息可以参考下面的链接：

- [2021/08/03/GAT稳定版本推进](#)
- [2022/10/28/GAT稳定版本发布公告](#)

第一个对上面的例子进行了更深入的讨论，第二个讨论了一些已知的局限性。

更深入的阅读可以在关联类型的 [nightly reference](#) 和 [原始 RFC](#) (已经过去6.5年了！) 里找到。

## let - else 语法

新的 `let` 语法，尝试模式匹配，找不到匹配的情况下执行发散的 `else` 块。

```
let PATTERN: TYPE = EXPRESSION else {
    DIVERGING_CODE;
};
```

常规的 `let` 语法仅能使用 `irrefutable patterns`，直译为不可反驳的模式，也就是一定要匹配上。一般情况下都是单个变量绑定，也用在解开结构体，元组，数组等复合类型上。原先并不适用条件匹配，比如从枚举里确定枚举值。直到现在我们有了 `let - else`。这是 `refutable pattern`，直译为可反驳的模式，能够像常规 `let` 一样匹配并绑定变量到周围范围内，在模式不匹配的时候执行发送的 `else`（可以是 `break`, `return`, `panic!`）。

```
fn get_count_item(s: &str) -> (u64, &str) {
    let mut it = s.split(' ');
    let (Some(count_str), Some(item)) = (it.next(), it.next()) else {
        panic!("Can't segment count item pair: '{s}'");
    };
    let Ok(count) = u64::from_str(count_str) else {
        panic!("Can't parse integer: '{count_str}'");
    };
    (count, item)
}
assert_eq!(get_count_item("3 chairs"), (3, "chairs"));
```

`if - else` 和 `match` 或者 `if let` 最大不一样的地方是变量绑定的范围，在此之前你需要多写一点重复的代码和一次外层的 `let` 绑定来完成：

```
let (count_str, item) = match (it.next(), it.next()) {
    (Some(count_str), Some(item)) => (count_str, item),
    _ => panic!("Can't segment count item pair: '{s}'"),
};

let count = if let Ok(count) = u64::from_str(count_str) {
    count
} else {
    panic!("Can't parse integer: '{count_str}'");
};
```

## break 跳出标记过的代码块

块表达式现在可以标记为 `break` 的目标，来达到提前终止块的目的。这听起来有点像 `goto` 语法，不过这并不是随意的跳转，只能从块里跳转到块末尾。这在之前已经可以用 `loop` 块来实现了，你可能大概率见过这种总是只执行一次的 `loop`。

1.65 可以直接给块语句添加标记来提前退出了，还可以携带返回值：

```
let result = 'block: {
    do_thing();
    if condition_not_met() {
        break 'block 1;
    }
    do_next_thing();
    if condition_not_met() {
        break 'block 2;
    }
    do_last_thing();
    3
};
```

## Others

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.66 | 重点: 有字段枚举的显示判别

Rust 1.66 官方 release doc: [Announcing Rust 1.66.0 | Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.66 版本:

```
$ rustup update stable
```

## 对有字段枚举的显示判别

枚举的显示判别在跨语言传递值时很关键，需要两个语言里每个枚举值的判别是一致的，比如：

```
#[repr(u8)]
enum Bar {
    A,
    B,
    C = 42,
    D,
}
```

这个例子里，枚举 `Bar` 使用了 `u8` 作为原语表形(representation)，并且 `Bar::C` 使用 42 来判别，其它没有显示判别的枚举值会按照源码里的顺序自动地递增赋值，这里的 `Bar::A` 是 0，`Bar::B` 是 1，`Bar::D` 是 43。如果没有显示判别，那就只能在 `Bar::B` 和 `Bar::C` 之间加上 40 个无意义的枚举值了。

在 1.66 之前，枚举的显示判别只能用在无字段枚举上。现在对有字段枚举的显示判别也稳定了：

```
#[repr(u8)]
enum Foo {
    A(u8),
    B(i8),
    C(bool) = 42,
}
```

注意：可以通过 `as` 转换（比如 `Bar::C as u8`）来判断一个无字段枚举的判别值，但是 Rust 还没有给有字段枚举提供语言层面上的获取原始判别值的方法，只能通过 `unsafe` 的代码来检查有字段枚举的判别

值。考虑到这个使用场景往往出现在必须使用 unsafe 代码的跨语言的 FFI 里，希望这没有造成太大的负担。如果你的确需要的话，参考 `std::mem::discriminant`。

## 黑盒方法 `core::hint::black_box`

当对编译器产生的代码做基准测试时，常常需要阻止一些优化，比如下面的代码里，`push_cap` 在一个循环里执行了4次 `Vec::push`：

```
fn push_cap(v: &mut Vec<i32>) {
    for i in 0..4 {
        v.push(i);
    }
}

pub fn bench_push() -> Duration {
    let mut v = Vec::with_capacity(4);
    let now = Instant::now();
    push_cap(&mut v);
    now.elapsed()
}
```

如果你检查一下在 x86\_64 机器上编译的优化输出结果，你会注意到整个 `push_cap` 方法都被优化掉了...

```
example::bench_push:
    sub rsp, 24
    call qword ptr [rip + std::time::Instant::now@GOTPCREL]
    lea rdi, [rsp + 8]
    mov qword ptr [rsp + 8], rax
    mov dword ptr [rsp + 16], edx
    call qword ptr [rip + std::time::elapsed@GOTPCREL]
    add rsp, 24
    ret
```

现在可以通过调用 `black_box` 来避免类似情况的发送。虽然实际上 `black_box` 内部只会取走值并直接返回，但是编译器会认为这个方法可能做任何事情。

```
use std::hint::black_box;

fn push_cap(v: &mut Vec<i32>) {
    for i in 0..4 {
        v.push(i);
        black_box(v.as_ptr());
    }
}
```

这样就可以得到展开循环的结果：

```
mov dword ptr [rbx], 0
mov qword ptr [rsp + 8], rbx
mov dword ptr [rbx + 4], 1
mov qword ptr [rsp + 8], rbx
mov dword ptr [rbx + 8], 2
mov qword ptr [rsp + 8], rbx
mov dword ptr [rbx + 12], 3
mov qword ptr [rsp + 8], rbx
```

你还能发现结果里有 `black_box` 带来的副作用，无意义的 `mov qword ptr [rsp + 8], rbx` 指令在每一次循环后出现，用来获取 `v.as_ptr()` 作为参数传递给并未真正使用的方法。

注意到上面的例子里，`push` 指令都不用考虑内存分配的问题，这是因为编译器运行在 `Vec::with_capacity(4)` 的条件下。你可以尝试改动一下 `black_box` 的位置或者在多处使用，来看看其对编译的优化输出的影响。

## cargo remove

1.62里我们引入了 `cargo add` 来通过命令行给你的项目增加依赖项。现在可以使用 `cargo remove` 来移除依赖了。

## Others

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.67 | #[must\_use] in async fn

---

Rust 1.67 官方 release doc: [Announcing Rust 1.67.0 | Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.67 版本:

```
$ rustup update stable
```

2023新年好！大年初五更新的新版本，来看看有什么新变化~

## #[must\_use] 作用于 async fn 上

注明了 `#[must_use]` 的 `async` 函数会把该属性应用在返回的 `impl Future` 结果上。`Future trait` 已经注明了 `#[must_use]`，所以所有实现了 `Future` 的类型都会自动加上 `#[must_use]`。

所以在 1.67 版本，编译器会警告返回值没有被使用：

```
#[must_use]
async fn bar() -> u32 { 0 }

async fn caller() {
    bar().await;
}

warning: unused output of future returned by `bar` that must be used
--> src/lib.rs:5:5
|
5 |     bar().await;
|     ^^^^^^^^^^^^
|
= note: `#[warn(unused_must_use)]` on by default
```

## **std::sync::mpsc 实现更新**

标准库里的 mpsc (多生产者单消费者) 通道自从 1.0 版本就有了，这次版本更新将其实现修改成了基于 [crossbeam-channel](#)。不涉及到API的变更，但是修改了一些已有的bug，提升了性能和代码可维护性。用户应该不太会感知到明显的变化。

## **Others**

其它更新细节，和稳定的API列表，参考[原Blog](#)

# Rust 新版解读 | 1.68 | crates index 优化

Rust 1.68 官方 release doc: [Announcing Rust 1.68.0](#) | [Rust Blog](#)

通过 [rustup](#) 安装的同学可以使用以下命令升级到 1.68 版本:

```
$ rustup update stable
```

## Cargo 稀疏注册协议 (sparse protocol)

Cargo的“稀疏”注册协议已经稳定，它是用来读取注册在 crates.io 上的 crates 的索引的基础设施。以前的 git 协议（目前仍然是默认协议）会克隆一个包括所有 crates 的索引的仓库，但这已经开始遇到扩展限制问题，在更新该仓库时会出现明显的延迟。新协议应在访问 crates.io 时提供显着的性能提升，因为它只会下载有关实际用到的 crates 的索引。

要使用新的协议，需要设置环境变量 `CARGO_REGISTRIES_CRATES_IO_PROTOCOL=sparse`，或者编辑 `.cargo/config.toml` 文件添加：

```
[registries.crates-io]
protocol = "sparse"
```

稀疏注册协议目前计划于 1.70.0 版本成为默认的协议。更多细节可以看官方博客的 [announcement](#), [RFC 2789](#), 当前 Cargo Book 的 [文档](#)

## 局部 Pin 构造

新增的 `pin!` 宏能够用 `T` 构造一个 `Pin<&mut T>`，从而匿名捕获在局部状态内。这通常叫做 堆栈固定 (stack-pinning)，同时这个堆栈也可以被 `async fn` 或者 代码块 来捕获住。这个宏和一些 crates 里提供的（比如 `tokio::pin!`）很像，但是标准库可以利用 `Pin` 的内部结构和 临时生命周期拓展 (Temporary lifetime extension) 来实现出更像表达式的宏。

```
// Runs a future to completion.
fn block_on<F: Future>(future: F) -> F::Output {
    let waker_that_unparks_thread = todo!();
    let mut cx = Context::from_waker(&waker_that_unparks_thread);
    // Pin the future so it can be polled.
    let mut pinned_future = pin!(future);
    loop {
        match pinned_future.as_mut().poll(&mut cx) {
            Poll::Pending => thread::park(),
            Poll::Ready(result) => return result,
        }
    }
}
```

在这个例子中，原来的 `future` 将被移动到一个临时的局部区域，由新的 `pinned_future` 引用，类型为 `Pin<&mut F>`，并且该 pin 受制于正常的借用检查器以确保它不会超过局部作用域。

## alloc 默认错误处理

当 Rust 内存分配失败时，类似于 `Box::new` 和 `Vec::push` 的 API 无法反映出这个错误，从而采取了一些不同的措施。当使用 `std` 时，程序会打印 `stderr` 然后中止。从 Rust 1.68.0 开始，包含 `std` 的二进制程序仍然会继续这样，而不保护 `std` 只包含 `alloc` 的二进制程序会对内存分配错误调用 `panic!`，如果需要可以再进一步通过 `#[panic_handler]` 来调整其行为。

未来，`std` 也可能改成这样。

## Others

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.69 | cargo fix

Rust 1.69 官方 release doc: [Announcing Rust 1.69.0](#) | [Rust Blog](#)

通过 [rustup](#) 安装的同学可以使用以下命令升级到 1.69 版本:

```
$ rustup update stable
```

## Cargo 提供自动修复建议

在 Rust 1.29.0 版本添加的 `cargo fix` 子命令，能够自动修复一些简单的编译错误。从那以后，能够自动修复的错误/警告原因的数量一直在稳步增加。此外，还增加了对自动修复一些简单的 Clippy 警告的支持。

为了让更多人注意到这些能力，现在当检测到可自动修复的错误时，Cargo 会建议运行 `cargo fix` 或 `cargo clippy --fix` 命令：

```
warning: unused import: `std::hash::Hash`
--> src/main.rs:1:5
  |
1 | use std::hash::Hash;
  | ^^^^^^^^^^^^^^^^
  |
= note: #[warn(unused_imports)]` on by default

warning: `foo` (bin "foo") generated 1 warning (run `cargo fix --bin "foo"` to apply
1 suggestion)
```

注意上面的完整命令（即包含 `--bin foo`）仅在你想要精确修复一个单独的 crate 时需要附上。默认执行 workspace 下所有 fixes 只需要 `cargo fix`。

## 构建脚本默认不再包含调试信息

为了提高编译速度，Cargo 现在默认避免在构建脚本中发出调试信息。构建脚本成功执行时不会有可见的效果，但构建脚本中的回溯（backtraces）将包含更少的信息。

所以如果想要 debug 构建脚本，需要额外开启调试信息，在 `Cargo.toml` 文件里添加

```
[profile.dev.build-override]
debug = true
[profile.release.build-override]
debug = true
```

## Others

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.70 | OnceCell && IsTerminal

---

Rust 1.70 官方 release doc: [Announcing Rust 1.70.0](#) | [Rust Blog](#)

---

通过 `rustup` 安装的同学可以使用以下命令升级到 1.70 版本:

```
$ rustup update stable
```

## crates.io 默认使用稀疏注册协议 (sparse protocol)

在 Rust 1.68.0 版本里稳定但需要手动配置来启用的特性，在这个版本中作为默认值了。如今在拉取 crates.io 索引信息时，应该能够观察到大幅性能提升。

注意当用户处在有防火墙限制的网络环境下时，需要确保能够访问 `https://index.crates.io` 来使用该协议。如果因为某些原因需要继续使用原先由 Github 托管的 git 索引，可以配置 `registries.crates-io.protocol` 来实现。

需要注意，两种访问方式依赖的本地缓存路径是不同的，所以更换访问方式会导致依赖被重新下载。当完全切换到稀疏注册协议后，或许你想要清理存储在 `$CARGO_HOME/registry/*/.github.com-*` 的旧依赖项

## OnceCell 和 OnceLock

稳定了 OnceCell 和多线程安全版本 OnceLock 两种共享数据类型，它们都能让数据不需要立刻初始化的同时保证仅初始化一次。

```

use std::sync::OnceLock;

static WINNER: OnceLock<&str> = OnceLock::new();

fn main() {
    let winner = std::thread::scope(|s| {
        s.spawn(|| WINNER.set("thread"));

        std::thread::yield_now(); // give them a chance...

        WINNER.get_or_init(|| "main")
    });

    println!("{} wins!", winner);
}

```

在之前需要使用 `lazy_static` 或者 `once_cell` 来实现这种效果，如今标准库里从 `once_cell` 的 `unsync` 和 `sync` 模块吸收了这些基础组件。未来可能还会有更多方法被稳定下来，比如存储初始化函数的 `LazyCell` 和 `LazyLock` 类型。不过当前这第一步应该能够覆盖许多使用场景了。

## IsTerminal

新稳定的 Trait，包含 `is_terminal` 一个方法，判断给定的文件描述符或者句柄是否代表一个终端/TTY。标准化了之前第三方 crates 如 `atty` `is-terminal` 实现的功能。一个常见的使用场景是，判断程序是通过脚本执行的还是交互模式执行的，以此在交互模式下实现一些诸如彩色输出、完整 TUI 的功能。

```

use std::io::{stdout, IsTerminal};

fn main() {
    let use_color = stdout().is_terminal();
    // if so, add color codes to program output...
}

```

## 调试信息级别的文本化

之前 `-Cdebuginfo` 编译选项仅支持数字 0,1,2 来表示逐渐增多的 debug 调试信息。（Cargo 默认在 dev 和 test 配置里是 2，在 release 和 bench 配置里是 0）

如今这些级别可以被文本代表：“none”(0), “limited”(1), and “full”(2), 还有两个新级别：“line-directives-only” and “line-tables-only”

之前 Cargo 和 rustc 的文档都把级别 1 叫做 "line-tables-only"，但是级别 1 实际上比这种场景包含的调试信息更多。新的 "line-tables-only" 仅保留的文件名和行号信息的最少调试信息量，或许会在未来成为 `-Cdebuginfo=1`。另外一个 "line-directives-only" 是为 NVPTX 场景准备的，不推荐使用。

注意这些文本化的选项还无法在 `Cargo.toml` 里使用，预计会在下一个 1.71 版本里实现。

## Others

其它更新细节，和稳定的API列表，参考[原Blog](#)

注：可以看到常用的 `Option` `Result` 新增了一些方法：

- `Option::is_some_and`
  - `pub fn is_some_and(self, f: impl FnOnce(T) -> bool) -> bool`
- `Result::is_ok_and`
  - `pub fn is_ok_and(self, f: impl FnOnce(T) -> bool) -> bool`
- `Result::is_err_and`
  - `pub fn is_err_and(self, f: impl FnOnce(E) -> bool) -> bool`

平时使用时可以试试。

# Rust 新版解读 | 1.71 | C-unwind API

Rust 1.71 官方 release doc: [Announcing Rust 1.71.0](#) | [Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.71 版本:

```
$ rustup update stable
```

## C-unwind API

1.71.0 稳定了 `c-unwind` 和其他 `-unwind` 后缀的ABI，具体见[列表](#)

非强制unwinding的结果在这个RFC的[表格](#)里。

带有 `-unwind` 后缀的ABI在由于 `panic` 或者 C++ 的异常(exception)而执行 unwinding 时，穿过 ABI 边界会更安全，除此以外和没有 `-unwind` 后缀的 ABI 基本等效。使用 `panic=unwind` 可以有效地让异常从一种语言堆栈展开(stack unwind)到另一种语言而不需要中止进程（只要这个异常的产生和捕获都是在同一语言内进行的）。而 `panic=abort` 通常会立刻中止进程。

这次稳定不会影响已有的ABI（比如 `c`），通过这些ABI的 unwinding 仍然是 UB 未定义行为。未来的 Rust 版本会按照这个RFC来修复这些ABI（通常会在边界处`abort`）。我们鼓励需要unwind穿过ABI边界的用户使用新的 ABI 来确保未来的兼容性。

译者注：或许以下一些概念对理解上面这个更新内容有一些帮助：

- [FFI](#)
- [what-is-stack-unwinding](#)
- [Rust Unwinding](#)

## 调试器可视化属性

1.71.0 稳定了新的属性: `#[debug_visualizer(natvis_file = "...")]` 和 `# [debug_visualizer(gdb_script_file = "...")]`。它们允许植入 Nativis 描述和 GDB 脚本到 Rust 库里来改善通过调试器查看这些库数据结构时的输出结果。Rust本身有给标准库打包类似的脚本，如今这个特性让库作者也可以给其用户提供类似的体验了。

具体细节查看: [the-debugger\\_visualizer-attribute](#)

## raw-dylib 动态库链接

在 Windows 平台上, 通过在 `#[link]` 里使用新的选项 `kind="raw-dylib"`, Rust 现在支持使用动态库且编译期不需要依赖这个动态库。

这避免了要求用户安装这些库 (这在跨平台交叉编译的时候尤为困难), 也避免了在 crates 指明需要链接的库的具体版本。

使用新的属性 `#[link_ordinal]`, Rust 也支持通过动态库的符号顺序而不是符号名称来进行符号绑定。

## 线程局部常量初始化

其实是 1.59 稳定进标准库的功能 (没有在更新说明和文档里提过)。[文档](#)

```
use std::cell::RefCell;
use std::thread;

thread_local!(static FOO: RefCell<u32> = RefCell::new(1));

FOO.with(|f| {
    assert_eq!(*f.borrow(), 1);
    *f.borrow_mut() = 2;
});

// each thread starts out with the initial value of 1
let t = thread::spawn(move|| {
    FOO.with(|f| {
        assert_eq!(*f.borrow(), 1);
        *f.borrow_mut() = 3;
    });
});

// wait for the thread to complete and bail out on panic
t.join().unwrap();

// we retain our original value of 2 despite the child thread
FOO.with(|f| {
    assert_eq!(*f.borrow(), 2);
});
```

## **Others**

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.72 | feature启用提示

Rust 1.72 官方 release doc: [Announcing Rust 1.72.0](#) | [Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.72 版本:

```
$ rustup update stable
```

## 警告可能有用的 `cfg` 禁用项

一直以来都支持的通过 `cfg` 条件编译部分代码，例如在开启特定 feature 时的函数，或者针对特定平台的逻辑。之前编译器会直接无视掉这些代码，现在会记录这些符号名称和对应的 `cfg` 条件，因此可以警告你正在调用一个特定 feature 下的函数，需要启用 feature:

```
Compiling my-project v0.1.0 (/tmp/my-project)
error[E0432]: unresolved import `rustix::io_uring`
--> src/main.rs:1:5
  |
1  | use rustix::io_uring;
  | ^^^^^^^^^^^^^^^^^ no `io_uring` in the root
  |
note: found an item that was configured out
--> /home/username/.cargo/registry/src/index.crates.io-6f17d22bba15001f/rustix-
0.38.8/src/lib.rs:213:9
  |
213 | pub mod io_uring;
  | ^^^^^^^^^^
= note: the item is gated behind the `io_uring` feature

For more information about this error, try `rustc --explain E0432`.
error: could not compile `my-project` (bin "my-project") due to previous error
```

## 不受限制的常量计算时间

之前为了避免用户自定义的常量在编译期间进行估算时，陷入死循环或占用无限制的时间，Rust 之前会限制用作常量计算的语句数目。然而一些特殊的有创造性的 Rust 代码还是会超过这个限制，进而产生编译错误；更糟糕的情况是，是否达到限制的是会随着用户调用库的不同而变化的。

现在，可以在编译时执行无限制的常量计算。而为了避免长时间编译没有反馈，编译器会在编译时代码运行一段时间后发出一条消息，并在每次翻倍的一段时间后重复该消息。默认情况下，编译器还将在捕获无限循环的大量步骤后报错提示 `const_eval_long_run`，但可以用 `allow(const_eval_long_run)` 允许特别长的常量计算。

## Clippy lints 上升到 rustc

几个原本由 Clippy 提供的 lints，提升到 rustc 里：

- `clippy::undropped_manually_drops => undropped-manually-drops (deny)`
  - 无作用的 `ManullyDrop`
- `clippy::invalid_utf8_in_unchecked => invalid_from_utf8_unchecked (deny) 或 invalid_from_utf8 (warn)`
  - 检查调用 `std::str::from_utf8_unchecked` 和 `std::str::from_utf8_unchecked_mut` 转换不合法的 UTF-8 字面量，这会违反了 safety 前提，导致未定义行为
  - 检查 `std::str::from_utf8` 和 `std::str::from_utf8_mut` 转换不合法的 UTF-8 字面量，这会永远返回错误。
- `clippy::cmp_nan => invalid_nan_comparisons (warn)`
  - 检查使用 `f32::NAN` 或者 `f64::NAN` 参与比较，`NAN` 在比较时无任何意义，即使与自己比较也是无意义行为，建议使用 `is_nan()` 方法
- `clippy::cast_ref_to_mut => invalid_reference_casting (allow)`
  - 检查不使用内部可变性的从 `&T` 到 `&mut T` 的转换，这会导致未定义行为。当前这个lint本身还有些问题，所以是 `allow` 级别，预计会在 1.73 版本修正后变为默认 `deny`

## 未来对 Windows 的支持

未来的 release 版本里会放弃对 win10 以前的系统的官方支持，Rust 1.75 将成为最后一个支持 windows 7,8,8.1的版本，2024 年 2 月起的 rust 1.76 将仅支持 win10 及后续版本（`target : tier-1`）。详情见提案 [MCP 651](#)

## Others

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.73 | panic 报错展示优化

Rust 1.73 官方 release doc: [Announcing Rust 1.73.0 | Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.73 版本:

```
$ rustup update stable
```

## 更简洁的 panic 报错信息

默认的 panic handler 会把报错信息单独列出一行，当报错信息很长、包含多行或者嵌套结构的时候可读性会更强。

```
fn main() {
    let file = "ferris.txt";
    panic!("oh no! {file:?} not found!");
}
```

Rust 1.73 之前的:

```
thread 'main' panicked at 'oh no! "ferris.txt" not found!', src/main.rs:3:5
```

Rust 1.73 之后的:

```
thread 'main' panicked at src/main.rs:3:5:
oh no! "ferris.txt" not found!
```

另外，由 `assert_eq` 和 `assert_ne` 产生的 panic 消息也把自定义信息部分（第三个参数）的展示位置改动了一下：

```
fn main() {
    assert_eq!("🦀", "🦀", "ferris is not a fish");
}
```

Rust 1.73 之前的:

```
thread 'main' panicked at 'assertion failed: `left == right`'
  left: `"\u2699"`,
right: `"\u269a"`: ferris is not a fish', src/main.rs:2:5
```

Rust 1.73 之后的：

```
thread 'main' panicked at src/main.rs:2:5:
assertion `left == right` failed: ferris is not a fish
  left: "\u2699"
right: "\u269a"
```

## 线程局部初始化

如 [RFC 3184 提案](#), `LocalKey<Cell<T>>` 和 `LocalKey<RefCell<T>>` 现在可以直接用 `get()`, `set()`, `take()` 和 `replace()` 方法来操作, 不再需要写 `with(|inner|...)` 的闭包形式。声明线程静态局部变量的宏 `thread_local!` 内部就是使用的就是 `LocalKey<T>`。

新的方法让代码更简洁, 也避免了默认值在新线程运行额外的初始化代码。

```
thread_local! {
    static THINGS: Cell<Vec<i32>> = Cell::new(Vec::new());
}

fn f() {
    // before:
    THINGS.with(|i| i.set(vec![1, 2, 3]));
    // now:
    THINGS.set(vec![1, 2, 3]);
    // ...

    // before:
    let v = THINGS.with(|i| i.take());
    // now:
    let v: Vec<i32> = THINGS.take();
}
```

## Others

其它更新细节, 和稳定的API列表, 参考原Blog

# Rust 新版解读 | 1.74 | 通过 Cargo 配置 Lint

Rust 1.74 官方 release doc: [Announcing Rust 1.74.0](#) | [Rust Blog](#)

通过 [rustup](#) 安装的同学可以使用以下命令升级到 1.74 版本:

```
$ rustup update stable
```

## 通过 Cargo 配置 Lint

如 [RFC 3389](#) 提案, 如今 `Cargo.toml` 支持通过 `[lints]` 表格来配置来自编译器或其它检查工具的 lints 的警告等级 (forbid,deny,warn,allow)。等效于之前 crate 级别的属性:

```
#![forbid(unsafe_code)]
#![deny(clippy::enum_glob_use)]
```

如今可以通过配置 `Cargo.toml` 达到同样的效果:

```
[lints.rust]
unsafe_code = "forbid"

[lints.clippy]
enum_glob_use = "deny"
```

同时这些配置也可以在 `[workspace.lints]` 里配置, 然后通过 `[lints] workspace = true` 的形式在 workspace 里的每个项目里继承这些配置。Cargo 会动态跟踪这里的配置变化, 以在需要的时候重新编译检查。

更多信息, 可以参考 Cargo 参考手册里的 [lints](#) 和 [workspace.lints](#) 章节。

## Cargo 注册服务认证

两个新的特性: 自定义凭据提供商 (credential providers) 和私有注册服务认证 (authenticated private registries)

除了内置的和系统相关的安全凭据认证外，现在支持通过指定自定义的凭据提供商来生成和保存 tokens，减少了注册服务token泄露的可能性。

注册服务如今对于所有操作都可以要求认证，（之前仅仅是发布操作），这对私有注册服务提供了更好的安全性。

更多相关信息可以参考 [Cargo docs: registry-authentication](#)

## 模糊返回类型推断

或许你也曾被编译器警告过：“return type cannot contain a projection or `Self` that references lifetimes from a parent scope”。如今这点得到了改善，编译器如今允许在返回类型里使用 `Self` 和关联类型（`async fn` 和 `impl Trait`）

这个更新让 Rust 编译器变得更符合你期望它应该正常工作的样子，即使你都不知道这个 “projection” 的术语的含义是什么：）

之前做不到这一点的原因是无法正常处理一些使用到的声明周期标注，更多技术细节可以看这个 [stabilization pull request](#)，其中提到了如今可以写出这样的代码：

```
struct Wrapper<'a, T>(&'a T);

// Opaque return types that mention `Self`:
impl Wrapper<'_, ()> {
    async fn async_fn() -> Self { /* ... */ }
    fn impl_trait() -> impl Iterator<Item = Self> { /* ... */ }
}

trait Trait<'a> {
    type Assoc;
    fn new() -> Self::Assoc;
}
impl Trait<'_> for () {
    type Assoc = ();
    fn new() {}
}

// Opaque return types that mention an associated type:
impl<'a, T: Trait<'a>> Wrapper<'a, T> {
    async fn mk_assoc() -> T::Assoc { /* ... */ }
    fn a_few_assocs() -> impl Iterator<Item = T::Assoc> { /* ... */ }
}
```

## **Others**

其它更新细节，和稳定的API列表，参考原Blog

# Rust 新版解读 | 1.75 | async trait 和 RPITIT

Rust 1.75 官方 release doc: [Announcing Rust 1.75.0](#) | [Rust Blog](#)

通过 `rustup` 安装的同学可以使用以下命令升级到 1.75 版本:

```
$ rustup update stable
```

## 在 traits 里使用 async fn 和 impl Trait 形式的返回值

之前的[文章](#)里也提到了，Rust 1.75 将支持在 traits 里使用 `async fn` 和 `impl Trait` 形式的返回值。不过如同前文所说这个版本还是有一些限制。

译者注：实践上来说，即使不考虑兼容低版本的`rustc`，目前也还是需要使用主流的 `async-trait` 三方库来写出简洁灵活的 `async traits`。不过相信官方团队最终肯定会都跟进吸纳这些实用的功能的。

## 指针字节偏移API

裸指针 (`*const T` 和 `*mut T`) 以前主要用作操作某个具体的类型为T的对象。例如，`<*> const T>::add(1)` 会将大小是 `size_of::<T>()` 的字节数添加到指针指向的地址上。在某些情况下，使用字节偏移更方便，这些新API避免了调用者需要先将指针转换为 `*const u8 / *mut u8` 的情况。

- `pointer::byte_add`
- `pointer::byte_offset`
- `pointer::byte_offset_from`
- `pointer::byte_sub`
- `pointer::wrapping_byte_add`
- `pointer::wrapping_byte_offset`
- `pointer::wrapping_byte_sub`

## **rustc 的代码布局优化**

Rust编译器的速度一直在优化，当前新版本应用了[LLVM - BOLT](#)到二进制发布包，使得我们的基准测试用时时间改善了2%。这个工具优化了librustc\_driver.so库的布局，能更好地利用缓存。

我们现在也用 `-Ccodegen-units=1` 来构建rustc，这为LLVM提供了更多的优化机会。这个优化使得我们的基准测试平均用时又提高了1.5%。

这些优化暂时仅限于x86\_64-unknown-linux-gnu编译器，但我们预计后续会扩展到更多平台。

## **Others**

其它更新细节，和稳定的API列表，参考[原Blog](#)