

00 前置篇

系统级语言

00 | 让Rust成为你的下一门主力语言

开篇词 | 让Rust成为你的下一门主力语言

你好，我是陈天，目前是北美最大的免费流媒体服务TubiTV的研发副总裁，也是公众号程序人生和知乎专栏迷思的作者。

十八年以来，我一直从事高性能系统的研发工作，涵盖网络协议、网络安全、服务端架构、区块链以及云服务等方向。

因为喜欢使用合适的工具解决合适的问题，在职业生涯的不同阶段，我深度使用过形态和机理都非常不同的开发语言。

我用 C 和汇编打造过各种网络协议，维护过在网络安全领域非常知名的嵌入式操作系统 ScreenOS；用 Python /JavaScript 撰写过我曾经的创业项目途客圈；用 Node.js/Elixir 打造过 TubiTV 高并发的后端核心；用 Elixir 打造过区块链框架 Forge，也研究过 Haskell/F#、Clojure/Racket、Swift、Golang 和 C# 等其他语言。

2018年起，我开始关注Rust。当时我正在开发 Forge，深感 Elixir 处理计算密集型功能的无力，在汉东，也是《Rust编程之道》作者的介绍下，我开始学习和使用 Rust。

也正是因为之前深度使用了很多开发语言，当我一接触到 Rust，就明白它绝对是面向未来的利器。

首先，你使用起来就会感受到，Rust是一门非常重视开发者用户体验的语言。如果做一个互联网时代的编程语言用户体验的排行，Rust 绝对是傲视群雄的独一档。

你无法想象一门语言的编译器在告知你的代码错误的同时，还会极尽可能，给你推荐正确的代码。这就好比在你开发的时候，旁边坐着一个无所不知还和蔼可亲的大牛，在孜孜不倦地为你审阅代码，帮你找出问题所在。

比如下面的代码，我启动了一个新的线程引用当前线程的变量（[代码](#)）：

```
let name = "Tyr".to_string();
std::thread::spawn(|| {
    println!("hello {}", name);
});
```

这段代码极其简单，但它隐含着线程不安全的访问。当前线程持有的变量 name 可能在新启动的线程使用之前就被释放，发生 use after free 错误。

Rust 编译器，不仅能够通过类型安全在编译期检测出这一错误，告诉你这个错误产生的原因：“may outlive borrowed value”（我们暂且不管它是什么意思），并且，它还进一步推荐你加入“move”解决这个错误。为了方便你进一步了解错误详情，它还贴心地给出一个命令行“rustc --explain E0373”，让你可以从知识库中获取更多的信息：

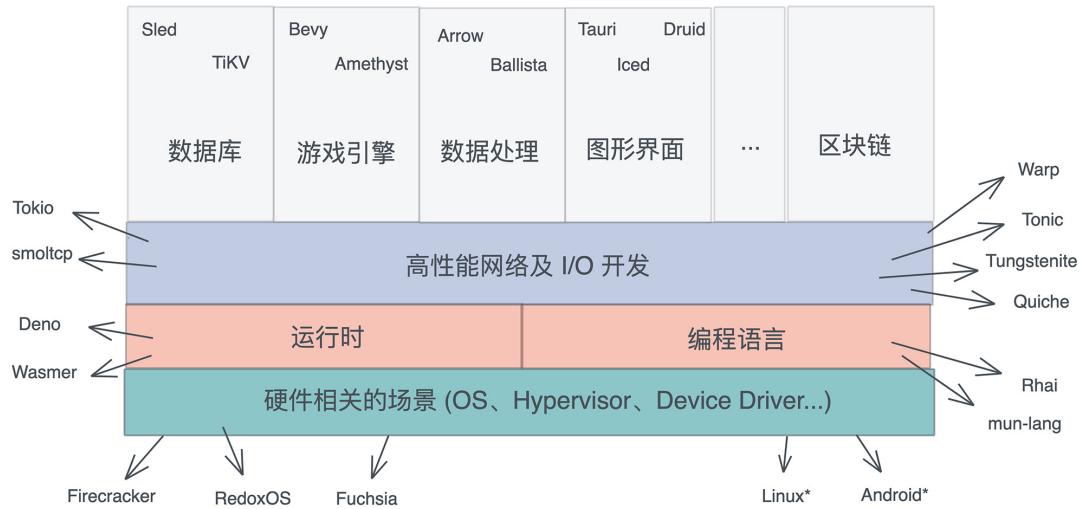
```
error[E0373]: closure may outlive the current function, but it borrows `name`, which is owned by the current function
--> basic/src/lib.rs:5:19
5 |     thread::spawn(|| {
  |         ^^^ may outlive borrowed value `name`
6 |         println!("hello {}", name);
  |                     ---- `name` is borrowed here
note: function requires argument type to outlive ``static``
--> basic/src/lib.rs:5:5
5 | /     thread::spawn(|| {
6 | |         println!("hello {}", name);
7 | |     });
  | |----- help: to force the closure to take ownership of `name` (and any other referenced variables), use the `move` keyword
5 |     thread::spawn(move || {
  |             ^^^^^^
error: aborting due to previous error

For more information about this error, try `rustc --explain E0373`.
```

这种程度的体验，一旦你适应了 Rust，就很难离得开。Rust 语言的这种极致用户体验不仅仅反映在编译器上，整个语言的工具链包括 rustup、cargo 等，都是如此简单易用、善解人意。

其次，众所周知的优异性能和强大的表现力，让Rust在很多场合都能够施展拳脚。

截止 2021 年，主流的互联网公司都把 Rust 纳入主力语言，比如开发操作系统 Redox/Fuchsia、高性能网络 Tokio、应用的高并发后端 TiKV，甚至客户端软件本身（飞书）。我们欣喜地看到，Rust 除了在其传统的系统开发领域，如操作系统、设备驱动、嵌入式等方向高歌猛进之外，还在服务端高性能、高并发场景遍地开花。



最近两年，几乎每隔一段时间我们就能听到很多知名互联网企业用 Rust 重构其技术栈的消息。比如 Dropbox 用 Rust 重写文件同步引擎、Discord 用 Rust 重写其状态服务。其实，**这些公司都是业务层面驱动自然使用到Rust的。**

比如 Discord 原先使用 Golang 的状态服务，一来会消耗大量的内存，二来在高峰期时不时会因为垃圾回收导致巨大的延迟，痛定思痛后，他们选用 Rust 重写。按照 Discord 的官方说法，Rust 除了带来性能上的提升外，还让随着产品迭代进行的代码重构变得举重若轻。

Along with performance, Rust has many advantages for an engineering team. For example, its type safety and borrow checker make it very easy to refactor code as product requirements change or new learnings about the language are discovered. Also, the ecosystem and tooling are excellent and have a significant amount of momentum behind them.

最后，是我自己的使用感觉，Rust会越用越享受。以我个人的开发经验看，很多语言你越深入使用或者越广泛使用，就越会有“怒其不争”的感觉，因为要么掣肘很多，无法施展；要么繁文缛节太多，在性能和简洁之间很难二选一。

而我在使用 Rust 的时候，这样的情况很少见。操作简单的 bit、处理大容量的 parquet、直面 CPU 乱序指令的 atomics，乃至像 Golang 一样高级封装的 channel，Rust 及其生态都应有尽有，让你想做什么的时候不至于“拔剑四顾心茫然”。

学习 Rust 的难点

在体验了 Rust 的强大和美妙后，2019 年，我开办了一系列讲座向我当时的团队普及 Rust，以便于处理 Elixir 难以处理的计算密集型的任务。但在这个过程中，我也深深地感受到把 Rust 的核心思想教给有经验开发者的艰辛。

Rust 被公认是很难学的语言，学习曲线很陡峭。

作为一门有着自己独特思想的语言，Rust 采百家之长，从 C++ 学习并强化了 move 语义和 RAII，从 Cyclone 借鉴和发展了生命周期，从 Haskell 吸收了函数式编程和类型系统等。

所以如果你想从其他语言迁移到 Rust，必须要经过一段时期的思维转换（Paradigm Shift）。

从命令式（imperative）编程语言转换到函数式（functional）编程语言、从变量的可变性（mutable）迁移到不可变性（immutable）、从弱类型语言迁移到强类型语言，以及从手工或者自动内存管理到通过生命周期来管理内存，难度是多重叠加。

而 Rust 中最大的思维转换就是**变量的所有权和生命周期**，这是几乎所有编程语言都未曾涉及的领域。

但是你一旦拿下这个难点，其他的知识点就是所有权和生命周期概念在不同领域的具体使用，比如，所有权和生命周期如何跟类型系统结合起来保证并发安全、生命周期标注如何参与到泛型编程中等等。

学习过程中，在所有权和生命周期之外，语言背景不同的工程师也会有不同难点，你可以重点学习：

- C 开发者，难点是类型系统和泛型编程；
- C++ 开发者，难点主要在类型系统；
- Python/Ruby/JavaScript 开发者，难点在并发处理，类型系统及泛型编程；
- Java 开发者，难点在异步处理和并发安全的理解上；
- Swift 开发者，几乎没有额外的难点，深入理解 Rust 异步处理即可。

只要迈过这段艰难的思维转换期，你就会明白，Rust 确实是一门从内到外透着迷人光芒的语言。

从语言的内核来看，它重塑了我们对一些基本概念的理解。比如 Rust 清晰地定义了变量在一个作用域下的生命周期，让开发者在摒弃垃圾回收（GC）这样的内存和性能杀手的前提下，还能够无需关心手动内存管理，**让内存安全和高性能二者兼得**。

从语言的外观来看，它使用起来感觉很像 Python/TypeScript 这样的高级语言，表达能力一流，但性能丝毫不输于 C/C++，**从而让表达力和高性能二者兼得**。

这种集表达力、高性能、内存安全于一身的体验，让 Rust 在 1.0 发布后不久就一路高飞猛进，从 16 年起，连续六年成为 [Stack Overflow 用户评选出来的最受喜爱的语言](#)。

如何学好 Rust？

Rust 如此受人喜爱，有如此广泛的用途，且当前各大互联网厂商都在纷纷接纳 Rust，那么我们怎样尽可能顺利地度过艰难的思维转换期呢？

在多年编程语言的学习和给团队传授经验的过程中，我总结了一套从入门到进阶的有效学习编程语言的方法，对 Rust 也非常适用。

我认为，**任何语言的学习离不开精准学习+刻意练习。**

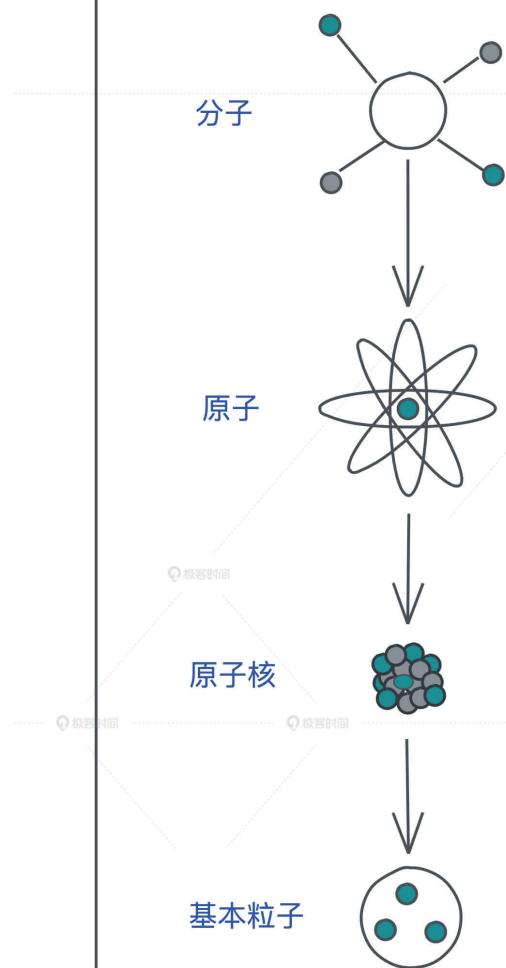
所谓**精准学习**，就是深挖一个个高大上的表层知识点，回归底层基础知识的本原，再使用类比、联想等方法，打通涉及的基础知识；然后从底层设计往表层实现，一层层构建知识体系，这样“撒一层土，夯实，再撒一层”，让你对知识点理解得更透彻、掌握得牢固。

比如 Rust 中的所有权和生命周期，很多同学说自己看书或者看其他资料，这部分都学得云里雾里的，即便深入逐一理解了几条基本规则，也依旧似懂非懂。

但我们进一步思考“值在内存中的访问规则”，最后回归到堆和栈这些最基础的软件开发的概念，重新认识堆栈上的值的存储方式和生命周期之后，再一层层往上，我们就会越学越明白。

这就是回归本原的重要性，也就是常说的第一性原理：回归事物最基础的条件，将其拆分成基本要素解构分析，来探索要解决的问题。

第一性原理



所有权和生命周期

基本规则

1. 值被唯一的 scope 拥有，它们共存亡；
2. 值可以移动到另一个 scope，新的 scope 拥有这个值；
3. 一个值可以有多个只读引用与单个可变引用，它们之间是互斥关系（RwLock）；
4. 引用不能超越值的存活期。

值在内存中的访问规则

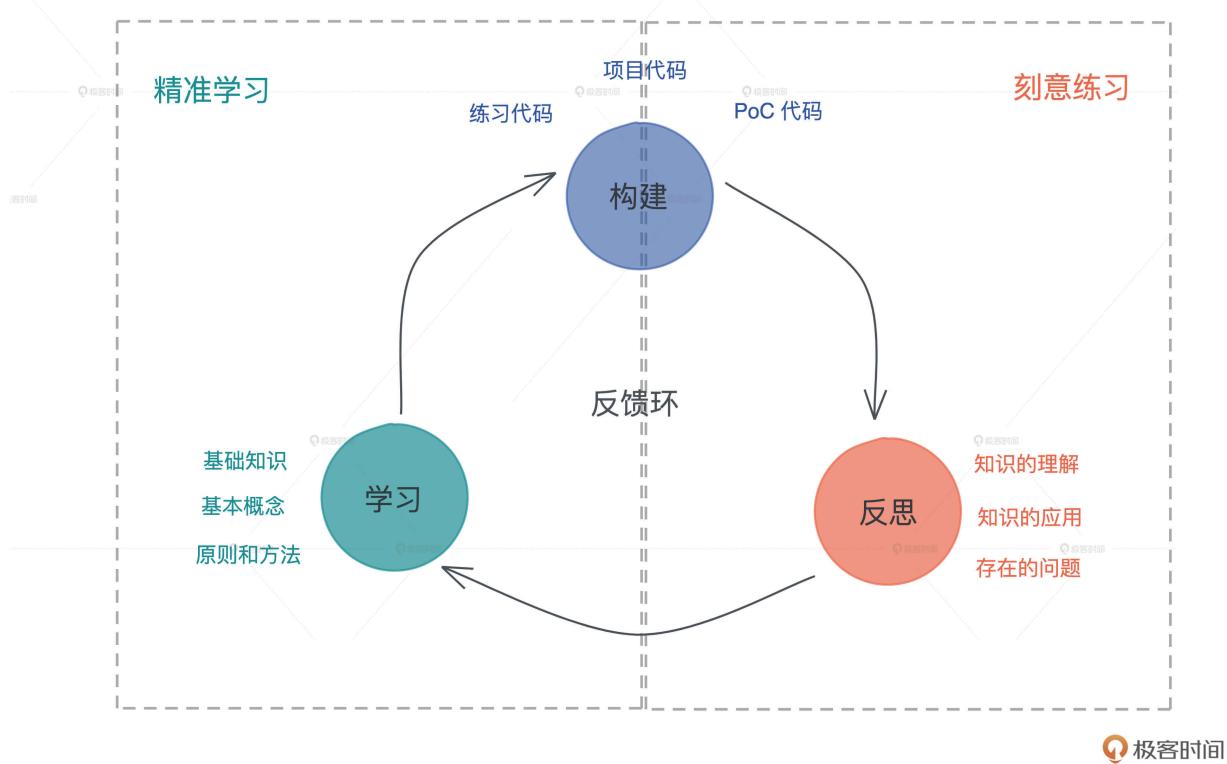
堆或栈中的值的存储方式

回归事物最基础的条件，将其拆分成基本要素解构分析，来探索要解决的问题



精准学习之后，我们就需要刻意练习了。刻意练习，就是用精巧设计的例子，通过练习进一步巩固学到的知识，并且在这个过程中尝试发现学习过程中的不自知问题，让自己从“我不知道我不知道”走向“我知道我不知道”，最终能够在下一个循环中弥补知识的漏洞。

这个过程就像子思在《中庸》里谈治学的方法：博学之，审问之，慎思之，明辨之，笃行之。我们学习就要这样，学了没有学会绝不罢休，不断在学习 - 构建 - 反思这个循环中提升自己。Rust 的学习，也是如此。



极客时间

根据这种学习思路，在这个专栏里，我会带着你循序渐进地探索 Rust 的基本概念和知识、开发的原则和方法，力求掌握 Rust 开发的精髓；同时，每一部分内容，都用一个或多个实操项目帮你巩固知识、查漏补缺。

具体来看，整个专栏会分成五个模块：

1. 前置知识篇

在正式学习 Rust 之前，先来回顾一下软件开发的基础概念：堆、栈、函数、闭包、虚表、泛型、同步和异步等。你要知道，想要学好任意一门编程语言，首先要吃透涉及的概念，**因为编程语言，不过是这些概念的具体表述和载体。**

2. 基础知识篇

我们会先来一个get hands dirty周，从写代码中直观感受Rust到底魅力在哪里，能怎么用，体会编程的快乐。

然后回归理性，深入浅出地探讨 Rust 变量的**所有权和生命周期**，并对比几种主流的内存管理方式，包括，Rust 的内存管理方式、C 的手工管理、Java 的 GC、Swift 的 ARC 。之后围绕着所有权和生命周期，来讨论 Rust 的几大语言特性：函数式编程特性、类型系统、泛型编程以及错误处理。

3. 进阶篇

Pascal 之父，图灵奖得主尼古拉斯·沃斯（Niklaus Wirth）有一个著名的公式：算法+数据结构=程序。想随心所欲地使用Rust 为你的系统构建数据结构，深度掌握类型系统必不可少。

在 Rust 里，你可以使用 Trait 做接口设计、使用泛型做编译期多态、使用 Trait Object 做运行时多态。在你的代码里用好 Trait 和泛型，可以非常高效地解决复杂的问题。

随后我们会介绍 unsafe rust，不要被这个名字吓到。所谓 unsafe，不过是把 Rust 编译器在编译器做的严格检查退步成为 C++ 的样子，由开发者自己为其所撰写的代码的正确性做担保。

最后我们还会讲到 FFI，这是 Rust 和其它语言互通操作的桥梁。掌握好 FFI，你就可以用 Rust 为你的 Python /JavaScript/Elixir/Swift 等主力语言在关键路径上提供更高的性能，也能很方便地引入 Rust 生态中特定的库。

4. 并发篇

从没有一门语言像 Rust 这样，在提供如此广博的并发原语支持的前提下，还能保证并发安全，所以 Rust 敢自称**无畏并发**（Fearless Concurrency）。在并发篇，我带你从 atomics 一路向上，历经 Mutex、Semaphore、Channel，直至 actor model。其他语言中被标榜为实践典范的并发手段，在 Rust 这里，只不过是一种并发工具。

Rust 还有目前最优秀的异步处理模型，我相信假以时日，这种用状态机巧妙实现零成本抽象的异步处理机制，必然会在更多新涌现出来的语言中被采用。

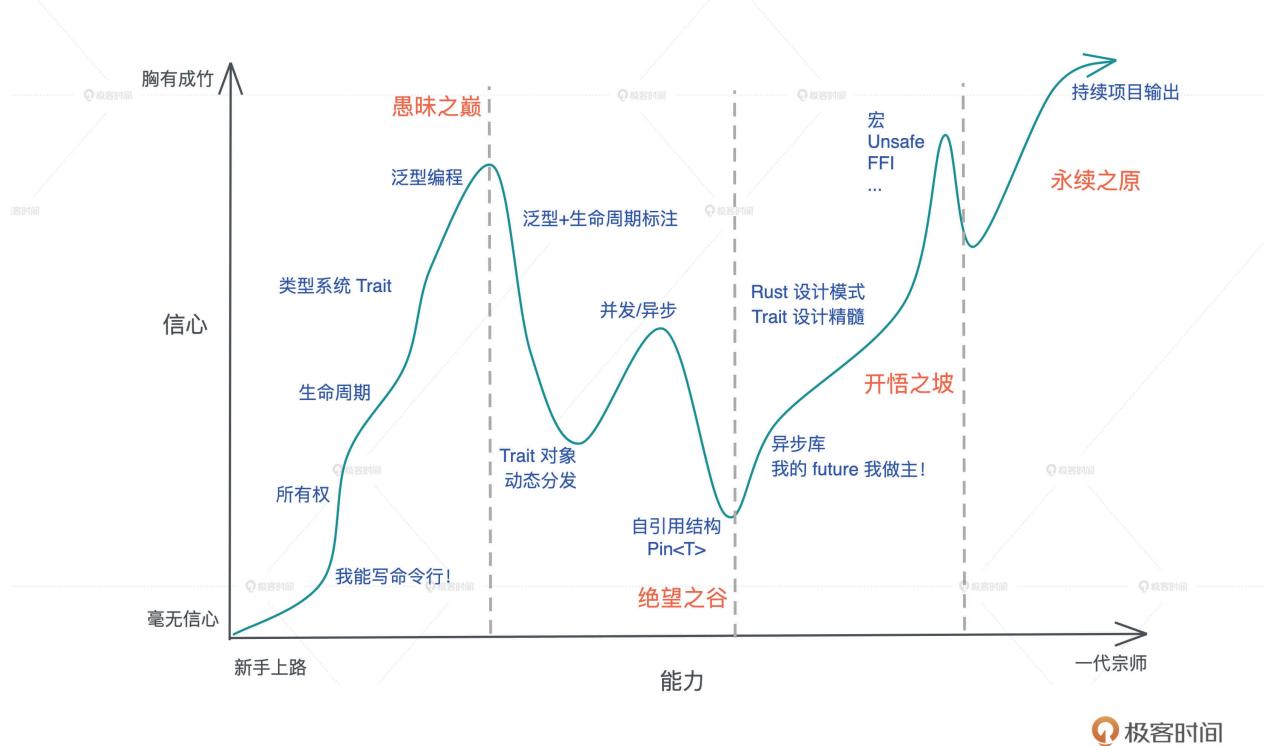
在并发处理这个领域，Rust 就像天秤座圣衣，刀枪剑戟斧钺钩叉，十八般兵器都提供给你，让你用最合适的工具解决最合适的问题。

5. 实战篇

掌握一门语言的特性，能应用这些特性，写出解决一些小问题的代码，算是初窥门径，就像在游泳池里练习冲浪，想真正把语言融会贯通，还要靠大风大浪中的磨炼。在这篇中，我们会学习如何把 Rust 应用在生产环境中、如何使用 Rust 的编程思想解决实际问题，最后谈谈如何用 Rust 构建复杂的软件系统。

整个专栏，我会把内容尽量写得通俗易懂，并把各个知识点类比到不同的语言中，力求让你理解 Rust 繁多概念背后的设计逻辑。每一讲我都会画出重点，理清知识脉络，再通过一个个循序渐进的实操项目，让你把各个知识点融会贯通。

我衷心希望，通过这个专栏的学习，你可以从基本概念出发，一步步跨过下图的愚昧之巅，越过绝望之谷，向着永续之原进发！通过一定的努力，最终自己也可以用 Rust 构建各种各样的系统，让自己职业生涯中多一门面向未来的利器。



我非常希望你能坚持学下去，和我一直走到最后一讲。这中间，你如果有想不明白的地方，可以先多思考多琢磨，如果还有困惑，欢迎你在留言区问我。

在具体写代码的时候，你可以多举一反三，不必局限于我给的例子，可以想想工作生活中的产品场景，思考如何用 Rust 来实现。

每讲的思考题，也希望你尽量完成，记录分享你的分析步骤和思路。有需要进一步总结提炼的知识点，你也可以记录下来，与我与其他学友分享。毕竟，大物理学家费曼总结过他的学习方法，评价和分享/教授给别人是非常重要的步骤，能让你进一步巩固自己学到的知识和技能。

最后，你可以自己立个 Flag，哪怕只是在留言区打卡你的学习天数或者Rust代码行数，我相信都是会有效果的。3个月后，我们再来一起验收。

总之，让我们携手，为自己交付“Rust 开发”这个大技能，让 Rust 成为你的下一门主力语言！

订阅后，[戳这里加入“Rust语言入门交流群”](#)，一起来学习Rust。

01 | 内存：值放堆上还是放栈上，这是一个问题

代码中最基本的概念是变量和值，而存放它们的地方是内存，所以我们就从内存开始。

内存

我们的程序无时无刻都在跟内存打交道。在下面这个把“hello world!”赋值给 s 的简单语句中，就跟只读数据段 (RODATA)、堆、栈分别有深度交互：

```
let s = "hello world".to_string();
```

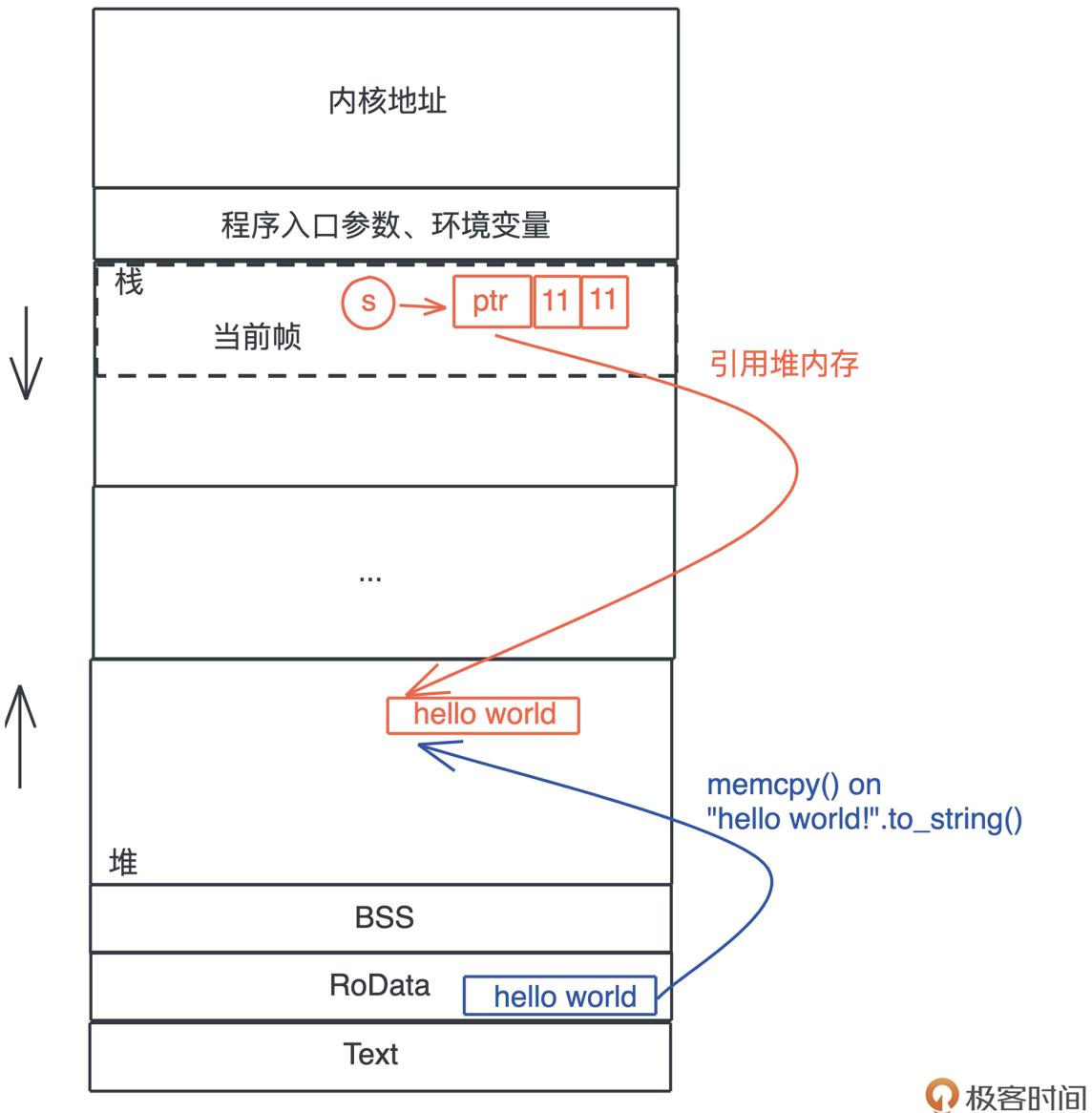
首先，“hello world”作为一个字符串常量 (string literal)，在编译时被存入可执行文件的 .RODATA 段 (GCC) 或者 .RDATA 段 (VC++)，然后在程序加载时，获得一个固定的内存地址。

当执行 “hello world”.to_string() 时，在堆上，一块新的内存被分配出来，并把“hello world”逐个字节拷贝过去。

当我们把堆上的数据赋值给 s 时，s 作为分配在栈上的一个变量，它需要知道堆上内存的地址，另外由于堆上的数据大小不确定且可以增长，我们还需要知道它的长度以及它现在有多大。

最终，为了表述这个字符串，我们使用了三个 word：第一个表示指针、第二个表示字符串的当前长度 (11)、第三个表示这片内存的总容量 (11)。在 64 位系统下，三个 word 是 24 个字节。

你也可以看下图，更直观一些：



数据什么时候可以放在栈上，什么时候需要放在堆上呢？

这个问题，很多使用自动内存管理语言比如 Java/Python 的开发者，可能有一些模糊的印象或者规则：

- 基本类型 (primitive type) 存储在栈上，对象存储在堆上；
- 少量数据存储在栈上，大量的数据存储在堆上。

这些虽然对，但并没有抓到实质。如果你在工作中只背规则套公式，一遇到特殊情况就容易懵，但是如果明白公式背后的推导逻辑，即使忘了，也很快能通过简单思考找到答案，所以接下来我们深挖堆和栈的设计原理，看看它们到底是如何工作的。

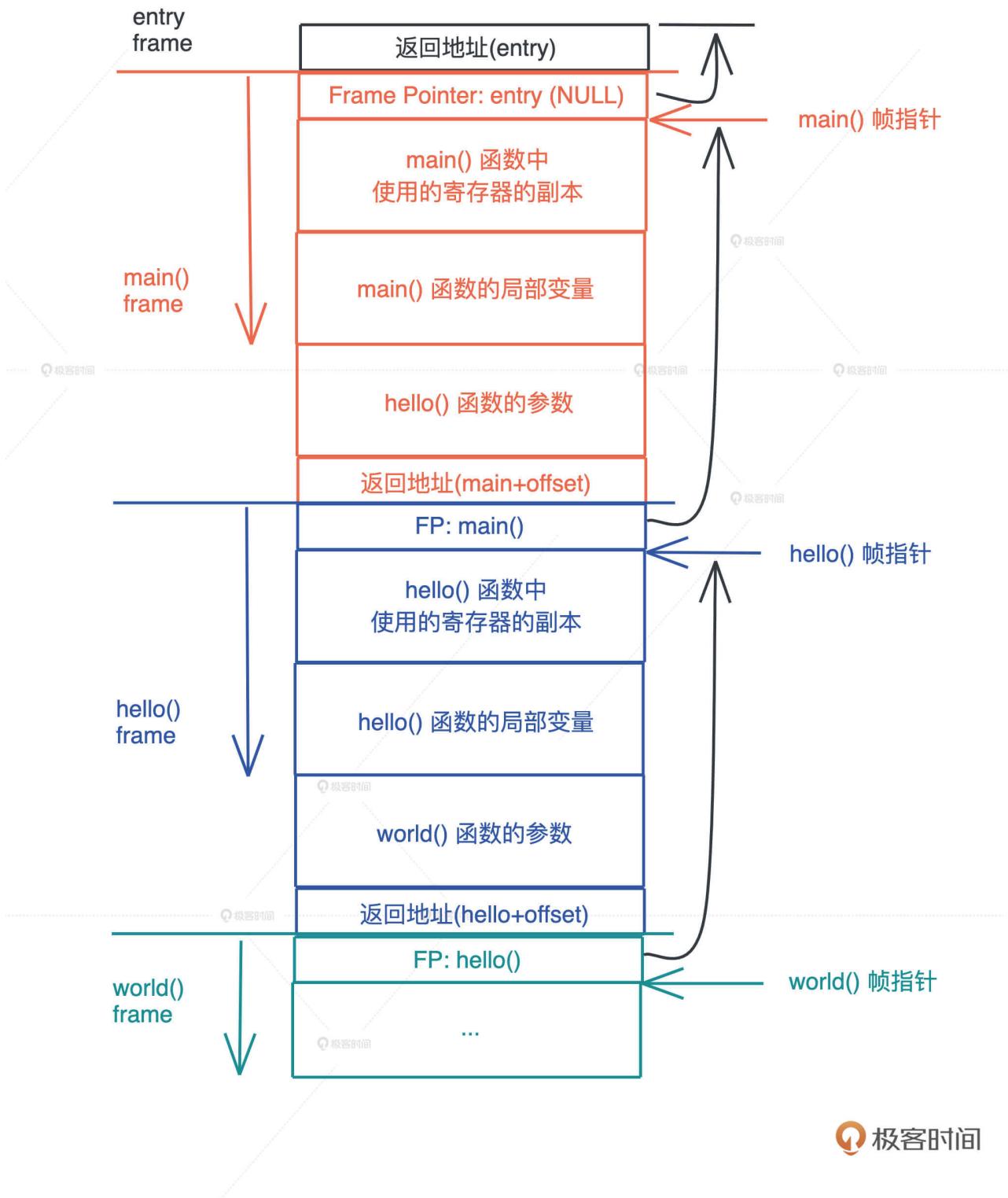
栈

栈是程序运行的基础。每当一个函数被调用时，一块连续的内存就会在栈顶被分配出来，这块内存被称为帧（frame）。

我们知道，栈是自顶向下增长的，一个程序的调用栈最底部，除去入口帧（entry frame），就是 main() 函数对应的帧，而随着 main() 函数一层层调用，栈会一层层扩展；调用结束，栈又会一层层回溯，把内存释放回去。

在调用的过程中，**一个新的帧会分配足够的空间存储寄存器的上下文**。在函数里使用到的通用寄存器会在栈保存一个副本，当这个函数调用结束，通过副本，可以恢复出原本的寄存器的上下文，就像什么都没有经历一样。此外，函数所需要使用到的局部变量，也都会在帧分配的时候被预留出来。

整个过程你可以再看看这张图辅助理解：



那一个函数运行时，怎么确定究竟需要多大的帧呢？

这要归功于编译器。在编译并优化代码的时候，一个函数就是一个最小的编译单元。

在这个函数里，编译器得知道要用到哪些寄存器、栈上要放哪些局部变量，而这些都要在编译时确定。所以编译器就需要明确每个局部变量的大小，以便于预留空间。

这下我们就明白了：**在编译时，一切无法确定大小或者大小可以改变的数据，都无法安全地放在栈上，最好放在堆上。**

放栈上的问题

从刚才的图中你也可以直观看到，栈上的内存分配是非常高效的。只需要改动栈指针（stack pointer），就可以预留相应空间；把栈指针改动回来，预留的空间又会被释放掉。预留和释放只是动动寄存器，不涉及额外计算、不涉及系统调用，因而效率很高。

所以理论上说，只要可能，我们应该把变量分配到栈上，这样可以达到更好的运行速度。

那为什么在实际工作中，我们又要避免把大量的数据分配在栈上呢？

这主要是考虑到调用栈的大小，避免栈溢出（stack overflow）。一旦当前程序的调用栈超出了系统允许的最大栈空间，无法创建新的帧，来运行下一个要执行的函数，就会发生栈溢出，这时程序会被系统终止，产生崩溃信息。

过大的栈内存分配是导致栈溢出的原因之一，更广为人知的原因是递归函数没有妥善终止。一个递归函数会不断调用自己，每次调用都会形成一个新的帧，如果递归函数无法终止，最终就会导致栈溢出。

堆

栈虽然使用起来很高效，但它的局限也显而易见。

1. **当我们需要动态大小的内存时，只能使用堆，比如可变长度的数组、列表、哈希表、字典，它们都分配在堆上。**堆上分配内存时，一般都会预留一些空间，这是最佳实践。
2. 除了动态大小的内存需要被分配到堆上外，**动态生命周期的内存也需要分配到堆上。**栈上的内存存在函数调用结束之后，所使用的帧被回收，相关变量对应的内存也都被回收待用。所以栈上内存的生命周期是不受开发者控制的，并且局限在当前调用栈。而堆上分配出来的每一块内存需要显式地释放，**这就使堆上内存有更加灵活的生命周期，可以在不同的调用栈之间共享数据。**

放堆上的问题

如果手工管理堆内存的话，堆上内存分配后忘记释放，就会造成**内存泄漏**。一旦有内存泄漏，程序运行得越久，就越吃内存，最终会因为占满内存而被操作系统终止运行。

如果堆上内存被多个线程的调用栈引用，该内存的改动要特别小心，需要加锁以独占访问，来避免潜在的问题。比如说，一个线程在遍历列表，而另一个线程在释放列表中的某一项，就可能访问野指针，导致**堆越界** (heap out of bounds)。而堆越界是第一大内存安全问题。

如果堆上内存被释放，但栈上指向堆上内存的相应指针没有被清空，就有可能发生**使用已释放内存 (use after free)** 的情况，程序轻则崩溃，重则隐含安全隐患。根据[微软安全反应中心 \(MSRC\) 的研究](#)，这是第二大内存安全问题。

GC、ARC如何解决

为了避免堆内存手动管理造成的这些问题，以 Java 为首的一系列编程语言，采用了追踪式垃圾回收 (Tracing GC) 的方法，来自动管理堆内存。这种方式通过定期标记 (mark) 找出不再被引用的对象，然后将其清理 (sweep) 掉，来自动管理内存，减轻开发者的负担。

而 ObjC 和 Swift 则走了另一条路：自动引用计数 (Automatic Reference Counting)。在编译时，它为每个函数插入 retain/release 语句来自动维护堆上对象的引用计数，当引用计数为零的时候，release 语句就释放对象。

我们来对比一下这两个方案。

从效率上来说，GC 在内存分配和释放上无需额外操作，而 ARC 添加了大量的额外代码处理引用计数，所以 GC 效率更高，吞吐量 (throughput) 更大。

但是，GC 释放内存的时机是不确定的，释放时引发的 STW (Stop The World)，也会导致代码执行的延迟 (latency) 不确定。所以一般携带 GC 的编程语言，不适于做嵌入式系统或者实时系统。当然，[Erlang VM](#)是个例外，它把 GC 的粒度下放到每个 process，最大程度解决了 STW 的问题。

我们使用 Android 手机偶尔感觉卡顿，而 iOS 手机却运行丝滑，大多是这个原因。而且做后端服务时，API 或者服务响应时间的 p99 (99th percentile) 也会受到 GC STW 的影响而表现不佳。

说句题外话，上面说的GC性能和我们常说的性能，涵义不太一样。常说的性能是吞吐量和延迟的总体感知，和实际性能是有差异的，GC 和 ARC 就是典型例子。GC 分配和释放内存的效率和吞吐量要比 ARC 高，但因为偶尔的高延迟，导致被感知的性能比较差，所以会给人一种 GC 不如 ARC 性能好的感觉。

小结

今天我们重新回顾基础概念，分析了栈和堆的特点。

对于存入栈上的值，它的大小在编译期就需要确定。栈上存储的变量生命周期在当前调用栈的作用域内，无法跨调用栈引用。

堆可以存入大小未知或者动态伸缩的数据类型。堆上存储的变量，其生命周期从分配后开始，一直到释放时才结束，因此堆上的变量允许在多个调用栈之间引用。但也导致堆变量的管理非常复杂，手工管理会引发很多内存安全性问题，而自动管理，无论是 GC 还是 ARC，都有性能损耗和其它问题。

一句话对比总结就是：栈上存放的数据是静态的，固定大小，固定生命周期；堆上存放的数据是动态的，不固定大小，不固定生命周期。

下一讲我们会讨论基础概念，比如值和类型、指针和引用、函数、方法和闭包、接口和虚表、并发与并行、同步和异步，以及 Promise/async/await，这些我们学习 Rust 或者任何语言都会接触到。

拓展阅读

1. [微软安全反应中心（MSRC）的研究](#)

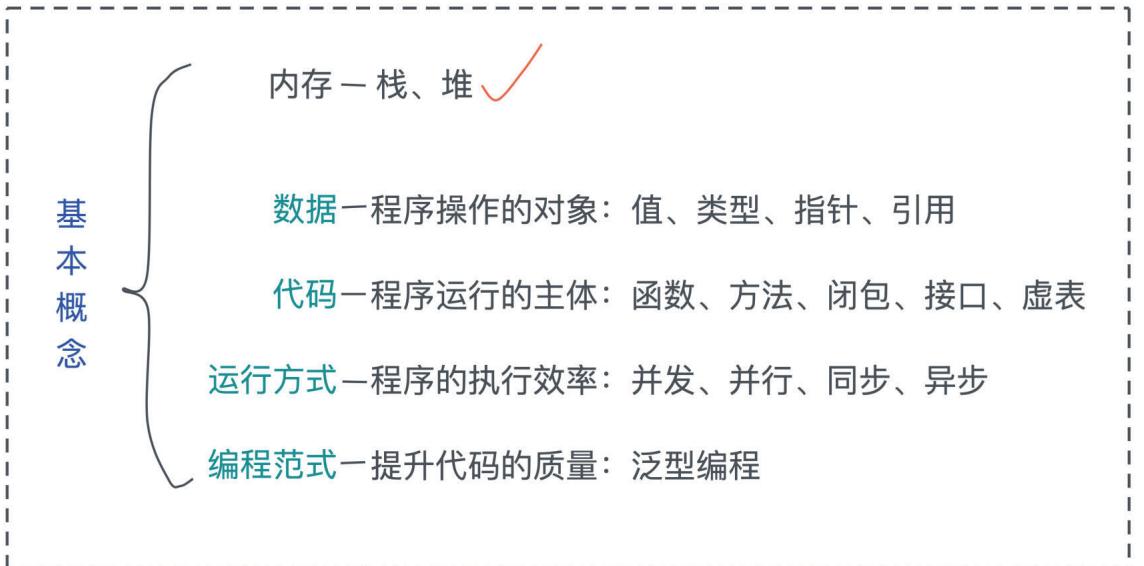
2. [追踪式垃圾回收Tracing GC](#)

3. [自动引用计数Automatic Reference Counting](#)

4. [Erlang VM 把 GC 的粒度下放到每个 process，最大程度解决了 STW 的问题](#)

02 | 串讲：编程开发中，那些你需要掌握的基本概念

编程开发中经常接触到的其它基本概念。需要掌握的小概念点比较多，为了方便你学习，我把它们分为四大类来讲解：数据（值和类型、指针和引用）、代码（函数、方法、闭包、接口和虚表）、运行方式（并发并行、同步异步和 Promise / async / await），以及编程范式（泛型编程）。



数据

数据是程序操作的对象，不进行数据处理的程序是没有意义的，我们先来重温和数据有关的概念，包括值和类型、指针和引用。

值和类型

严谨地说，类型是对值的区分，它包含了值在内存中的长度、对齐以及值可以进行的操作等信息。一个值是符合一个特定类型的数据的某个实体。比如 `64u8`，它是 `u8` 类型，对应一个字节大小、取值范围在 `0~255` 的某个整数实体，这个实体是 `64`。

值以类型规定的表达方式（representation）被存储成一组字节流进行访问。比如 `64`，存储在内存中的表现形式是 `0x40`，或者 `0b 0100 0000`。

这里你要注意，**值是无法脱离具体的类型讨论的**。同样是内存中的一个字节 `0x40`，如果其类型是 ASCII char，那么其含义就不是 `64`，而是 @ 符号。

不管是强类型的语言还是弱类型的语言，语言内部都有其类型的具体表述。一般而言，编程语言的类型可以分为原生类型和组合类型两大类。

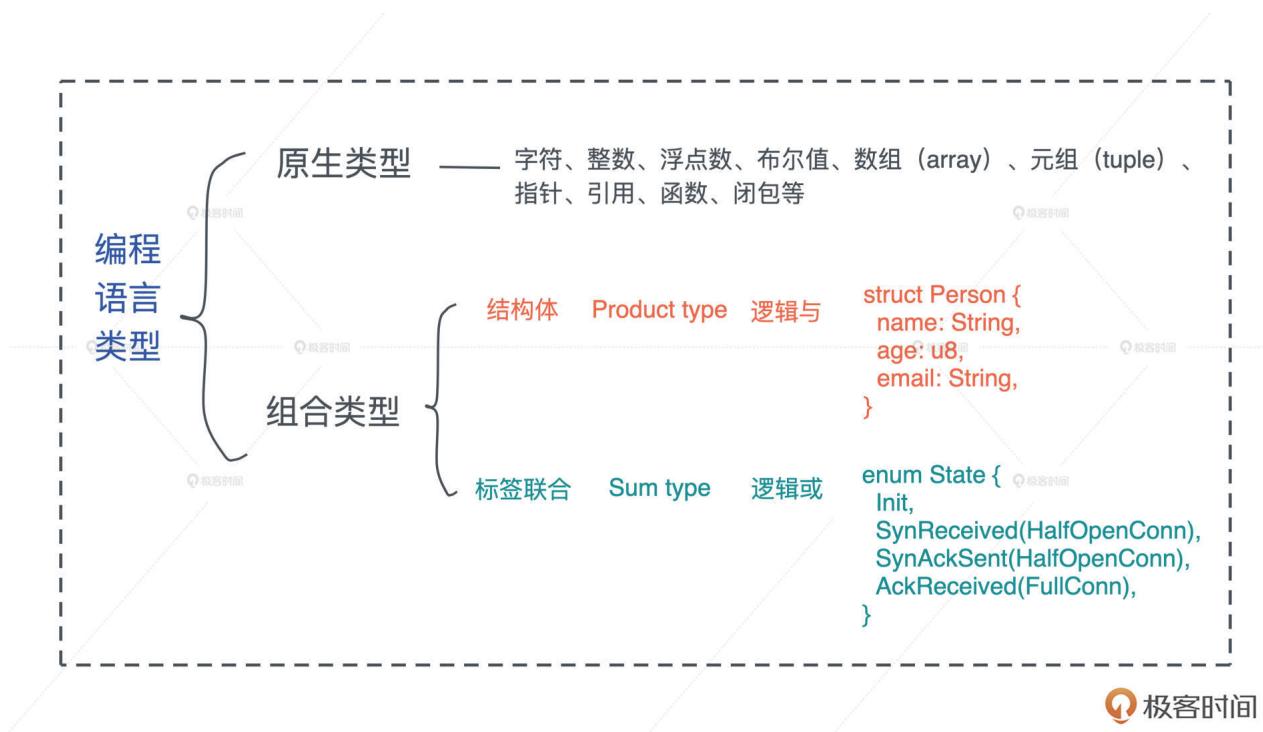
原生类型（primitive type）是编程语言提供的最基础的数据类型。比如字符、整数、浮点数、布尔值、数组（array）、元组（tuple）、指针、引用、函数、闭包等。**所有原生类型的大小都是固定的，因此它们可以被分配到栈上。**

组合类型 (composite type) 或者说复合类型，是指由一组原生类型和其它类型组合而成的类型。组合类型也可以细分为两类：

- **结构体** (structure type)：多个类型组合在一起共同表达一个值的复杂数据结构。比如 Person 结构体，内部包含 name、age、email 等信息。用代数数据类型 (algebraic data type) 的说法，结构体是 product type。
- **标签联合** (tagged union)：也叫不相交并集 (disjoint union)，可以存储一组不同但固定的类型的对象，具体是哪个类型由其标签决定。比如 Haskell 里的 Maybe 类型，或者 Swift 中的 Optional 就是标签联合。用代数数据类型的的说法，标签联合是 sum type。

另外不少语言不支持标签联合，只取其标签部分，提供了枚举类型 (enumerate)。枚举是标签联合的子类型，但功能比较弱，无法表达复杂的结构。

看定义可能不是太好理解，你可以看这张图：



指针和引用

在内存中，一个值被存储到内存中的某个位置，这个位置对应一个内存地址。而指针是一个持有内存地址的值，可以通过解引用 (dereference) 来访问它指向的内存地址，理论上可以解引用到任意数据类型。

引用 (reference) 和指针非常类似，不同的是，引用的解引用访问是受限的，它只能解引用到它引用数据的类型，不能用作它用。比如，指向 42u8 这个值的一个引用，它解引用的时候只能使用 u8 数据类型。

所以，指针的使用限制更少，但也会带来更多的危害。如果没有用正确的类型解引用一个指针，那么会引发各种各样的内存问题，造成系统崩溃或者潜在的安全漏洞。

指针和引用是原生类型，它们可以分配在栈上。

根据指向数据的不同，某些引用除了需要一个指针指向内存地址之外，还需要内存地址的长度和其它信息。

如上一讲提到的指向“hello world”字符串的指针，还包含字符串长度和字符串的容量，一共使用了 3 个 word，在 64 位 CPU 下占用 24 个字节，这样**比正常指针携带更多信息的指针**，我们称之为**胖指针 (fat pointer)**。很多数据结构的引用，内部都是由胖指针实现的。

代码

数据是程序操作的对象，而代码是程序运行的主体，也是我们开发者把物理世界中的需求转换成数字世界中逻辑的载体。我们会讨论函数和闭包、接口和虚表。

函数、方法和闭包

函数是编程语言的基本要素，它是对完成某个功能的一组相关语句和表达式的封装。**函数也是对代码中重复行为的抽象。**在现代编程语言中，函数往往是一等公民，这意味着函数可以作为参数传递，或者作为返回值返回，也可以作为复合类型中的一个组成部分。

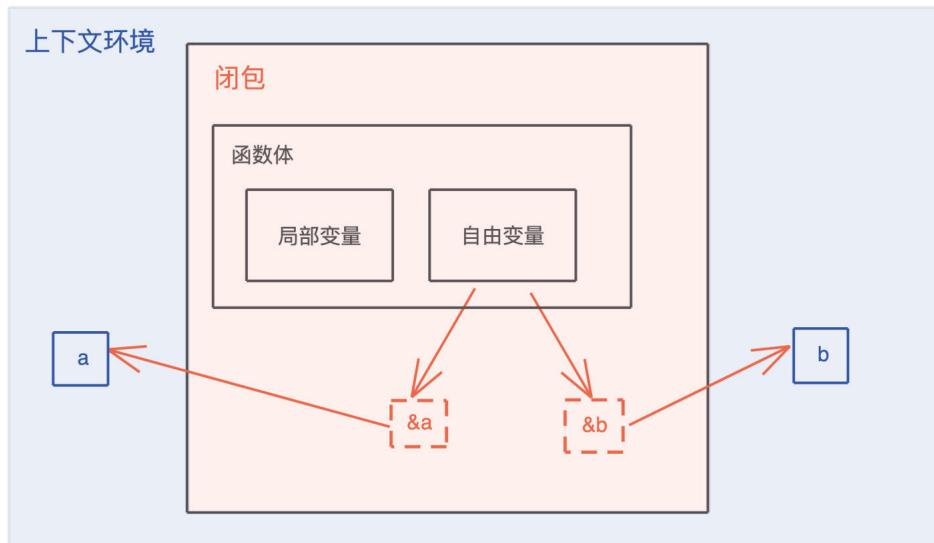
在面向对象的编程语言中，在类或者对象中定义的函数，被称为方法 (method)。方法往往和对象的指针发生关系，比如 Python 对象的 self 引用，或者 Java 对象的 this 引用。

而闭包是将函数，或者说代码和其环境一起存储的一种数据结构。**闭包引用的上下文中的自由变量，会被捕获到闭包的结构中，成为闭包类型的一部分。**

一般来说，如果一门编程语言，其函数是一等公民，那么它必然会支持闭包 (closure)，因为函数作为返回值往往需要返回一个闭包。

你可以看这张图辅助理解，图中展示了一个闭包对上下文环境的捕获。可以 [在这里](#) 运行这段代码：

上下文环境



```
fn main() {  
    let a = "Hello";  
    let b = "Tyr";  
  
    let c = |msg: &str| {  
        println!("{} {}:{} {}",  
            a, b, msg);  
    };  
  
    c("How are you?");  
}
```



接口和虚表

接口是一个软件系统开发的核心部分，它反映了系统的设计者对系统的抽象理解。作为一个抽象层，接口将使用方和实现方隔离开来，使两者不直接有依赖关系，大大提高了复用性和扩展性。

很多编程语言都有接口的概念，允许开发者面向接口设计，比如 Java 的 interface、Elixir 的 behaviour、Swift 的 protocol 和 Rust 的 trait。

比如说，在 HTTP 中，Request/Response 的服务处理模型其实就是一个典型的接口，我们只需要按照服务接口定义出不同输入下，从 Request 到 Response 具体该如何映射，通过这个接口，系统就可以在合适的场景下，把符合要求的 Request 分派给我们的服务。

面向接口的设计是软件开发中的重要能力，而 Rust 尤其重视接口的能力。在后续讲到 Trait 的章节，我们会详细介绍如何用 Trait 来进行接口设计。

当我们在运行期使用接口来引用具体类型的时候，代码就具备了运行时多态的能力。但是，在运行时，一旦使用了关于接口的引用，变量原本的类型被抹去，我们无法单纯从一个指针分析出这个引用具备什么样的能力。

因此，在生成这个引用的时候，我们需要构建胖指针，除了指向数据本身外，还需要指向一张涵盖了这个接口所支持方法的列表。这个列表，就是我们熟知的虚表（virtual table）。

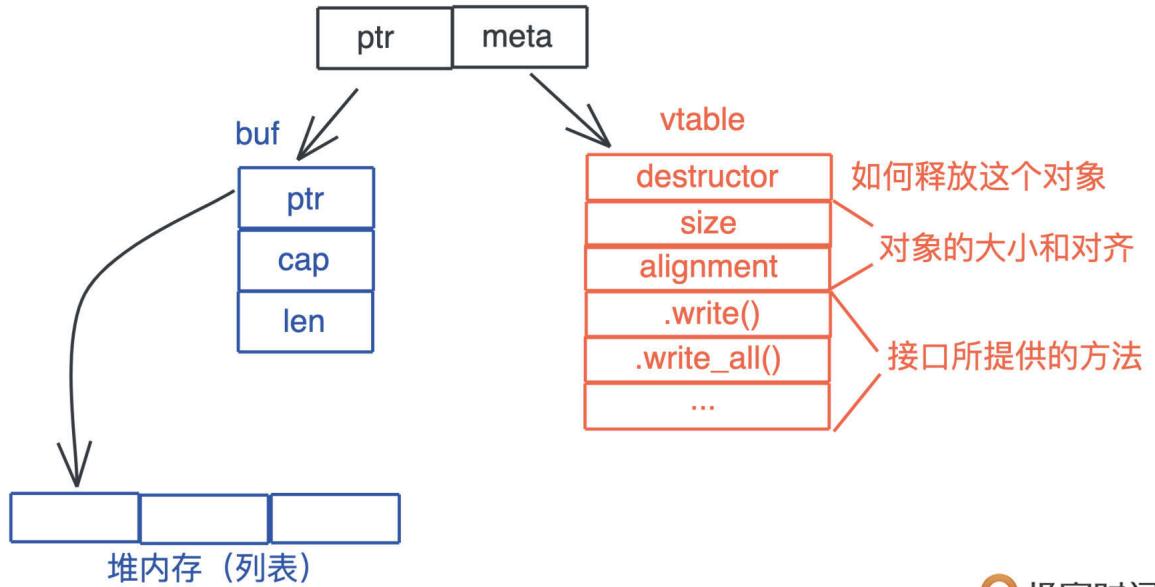
下图展示了一个 Vec 数据在运行期被抹去类型，生成一个指向 Write 接口引用的过程：

```

use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: &mut Write = &mut buf; // 构建一个带有虚表的 trait object

```



由于虚表记录了数据能够执行的接口，所以在运行期，我们想对一个接口有不同实现，可以根据上下文动态分派。

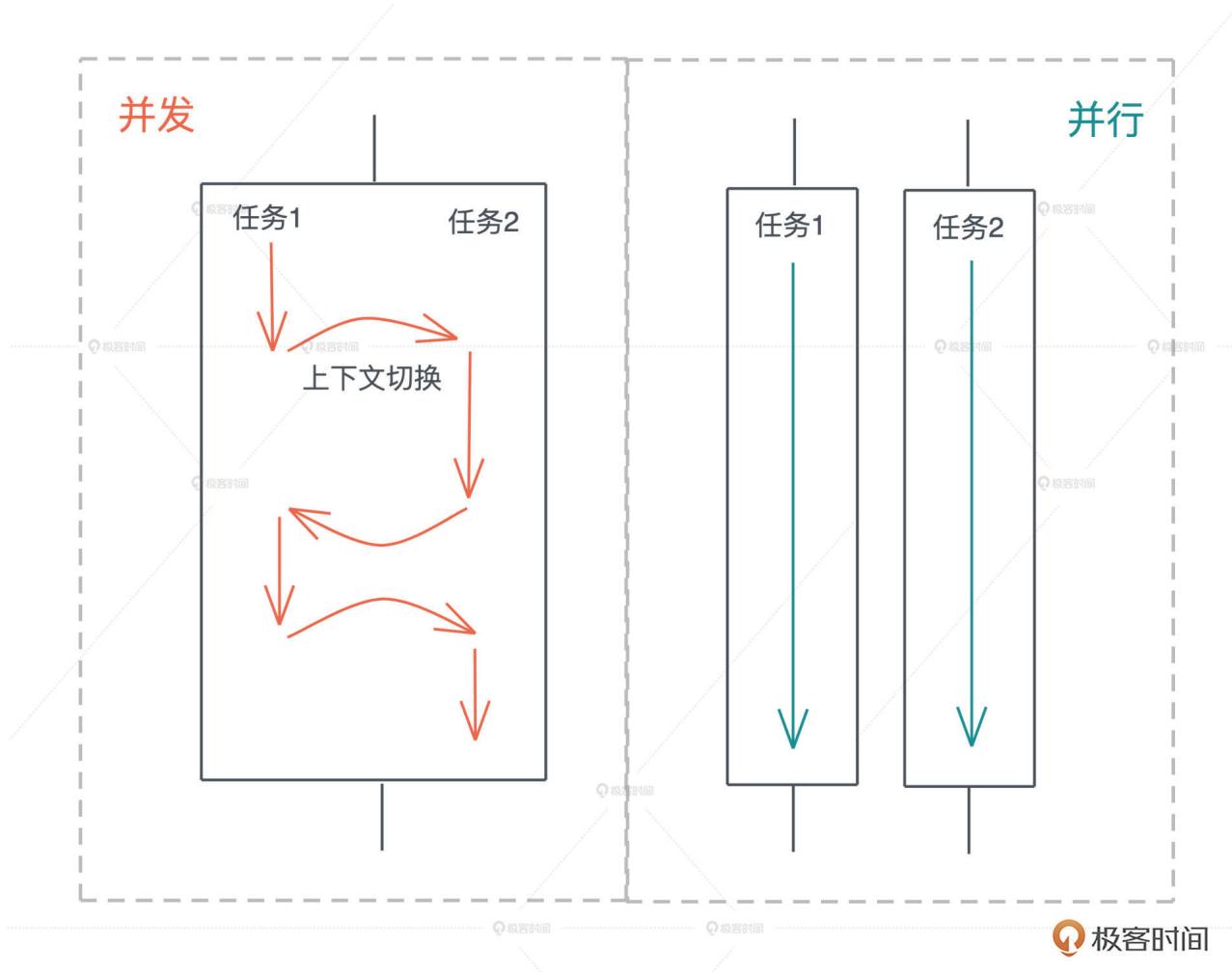
比如我想为一个编辑器的 Formatter 接口实现不同语言的格式化工具。我们可以在编辑器加载时，把所有支持的语言和其格式化工具放入一个哈希表中，哈希表的 key 为语言类型，value 为每种格式化工具 Formatter 接口的引用。这样，当用户在编辑器打开某个文件的时候，我们可以根据文件类型，找到对应 Formatter 的引用，来进行格式化操作。

运行方式

程序在加载后，代码以何种方式运行，往往决定着程序的执行效率。所以我们接下来讨论并发、并行、同步、异步以及异步中的几个重要概念 Promise/async/await。

并发 (concurrency) 与并行 (parallel)

并发和并行是软件开发中经常遇到的概念。



并发是一种能力，而并行是一种手段。当我们的系统拥有了并发的能力后，代码如果跑在多个 CPU core 上，就可以并行运行。所以我们平时都谈论高并发处理，而不会说高并行处理。

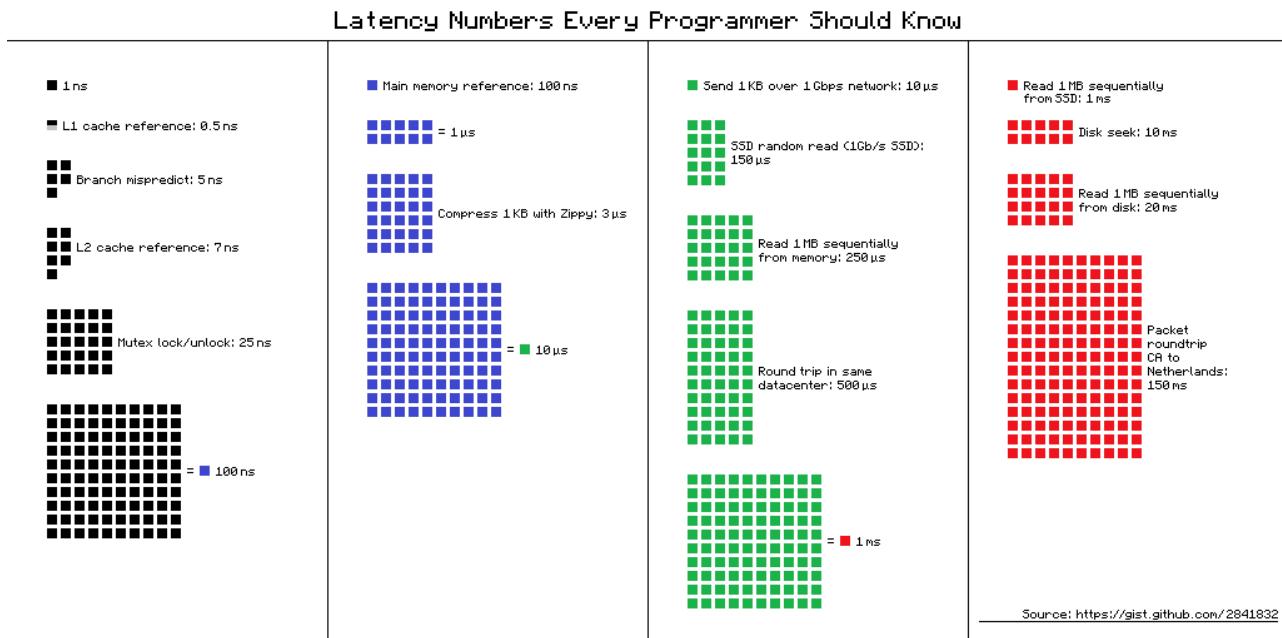
很多拥有高并发处理能力的编程语言，会在用户程序中嵌入一个 M:N 的调度器，把 M 个并发任务，合理地分配在 N 个 CPU core 上并行运行，让程序的吞吐量达到最大。

同步和异步

同步是指一个任务开始执行后，后续的操作会阻塞，直到这个任务结束。在软件中，我们大部分的代码都是同步操作，比如 CPU，只有流水线中的前一条指令执行完成，才会执行下一条指令。一个函数 A 先后调用函数 B 和 C，也会执行完 B 之后才执行 C。

同步执行保证了代码的因果关系 (causality)，是程序正确性的保证。

然而在遭遇 I/O 处理时，高效 CPU 指令和低效 I/O 之间的巨大鸿沟，成为了软件的性能杀手。下图对比了 CPU、内存、I/O 设备、和网络的延迟：



我们可以看到和内存访问相比，I/O 操作的访问速度低了两个数量级，一旦遇到 I/O 操作，CPU 就只能闲置来等待 I/O 设备运行完毕。因此，操作系统为应用程序提供了异步 I/O，让应用可以在当前 I/O 处理完毕之前，将 CPU 时间用作其它任务的处理。

所以，**异步是指一个任务开始执行后，与它没有因果关系的其它任务可以正常执行，不必等待前一个任务结束。**

在异步操作里，异步处理完成后的结果，一般用 Promise 来保存，它是一个对象，用来描述在未来的某个时刻才能获得的结果的值，一般存在三个状态：

1. 初始状态，Promise 还未运行；
2. 等待 (pending) 状态，Promise 已运行，但还未结束；
3. 结束状态，Promise 成功解析出一个值，或者执行失败。

如果你对 Promise 这个词不太熟悉，在很多支持异步的语言中，Promise 也叫 Future / Delay / Deferred 等。除了这个词以外，我们也经常看到 `async/await` 这对关键字。

一般而言，`async` 定义了一个可以并发执行的任务，而 `await` 则触发这个任务并发执行。大多数语言中，`async/await` 是一个语法糖 (syntactic sugar)，它使用状态机将 Promise 包装起来，让异步调用的使用感觉和同步调用非常类似，也让代码更容易阅读。

编程范式

为了在不断迭代时，更好地维护代码，我们还会引入各种各样的编程范式，来提升代码的质量。所以最后来谈谈泛型编程。

如果你来自于弱类型语言，如 C / Python / JavaScript，那泛型编程是你需要重点掌握的概念和技能。泛型编程包含两个层面，数据结构的泛型和使用泛型结构代码的泛型化。

数据结构的泛型化

首先是数据结构的泛型，它也往往被称为参数化类型或者参数多态，比如下面这个数据结构：

```
struct Connection<S> {  
    io: S,  
    state: State,  
}
```

它有一个参数 S，其内部的域 io 的类型是 S，S 具体的类型只有在使用 Connection 的上下文中才得到绑定。

你可以把参数化数据结构理解成一个产生类型的函数，在“调用”时，它接受若干个使用了具体类型的参数，返回携带这些类型的类型。比如我们为 S 提供 TcpStream 这个类型，那么就产生 Connection 这个类型，其中 io 的类型是 TcpStream。

这里你可能会疑惑，如果 S 可以是任意类型，那我们怎么知道 S 有什么行为？如果我们要调用 `io.send()` 发送数据，编译器怎么知道 S 包含这个方法？

这是个好问题，**我们需要用接口对 S 进行约束**。所以我们经常看到，支持泛型编程的语言，会提供强大的接口编程能力，在后续的课程中在讲 Rust 的 trait 时，我会再详细探讨这个问题。

数据结构的泛型是一种高级抽象，就像我们人类用数字抽象具体事物的数量，又发明了代数来进一步抽象具体的数字一样。它带来的好处是我们可以延迟绑定，让数据结构的通用性更强，适用场合更广阔；也大大减少了代码的重复，提高了可维护性。

代码的泛型化

泛型编程的另一个层面是使用泛型结构后代码的泛型化。当我们使用泛型结构编写代码时，相关的代码也需要额外的抽象。

这里用我们熟悉的二分查找的例子解释会比较清楚：

```
int binary_search(int x[], int n, int v) {  
    int l = 0;  
    int u = n;
```

```
    while (true) {  
        if (l > u) return -1;
```

```
        int m = (l + u) / 2;  
  
        if (x[m] < v) l = m + 1;  
        else if (x[m] == v) return m;  
        else u = m - 1;  
    }
```

与具体类型有关 int[]

泛型 \Rightarrow

ForwardIterator \quad value_type(l), comparable

```
template <class I, class T>  
I lower_bound(I first, I last, const T& v) {  
    while (first != last) {  
        auto m = next(first, distance(first, last) / 2);  
  
        if (*m < v) first = next(m);  
        else last = m;  
    }  
    return first;  
}
```

更加通用，与类型无关



左边用 C 撰写的二分查找，标记的几处操作隐含着和 int[] 有关，所以如果对不同的数据类型做二分查找，实现也要跟着改变。右边 C++ 的实现，对这些地方做了抽象，让我们可以用同一套代码二分查找迭代器（iterator）的数据类型。

同样的，这样的代码可以在更广阔的情况下使用，更简洁容易维护。

小结

今天我们讨论了四大类基础概念：数据、代码、运行方式和编程范式。

基本概念

- 数据 { 值 + 类型(原生类型、组合类型)
指针、引用，指向值的内存地址
- 代码 { 函数、方法、闭包
虚表记录数据能够执行的接口
- 运行方式 { 并发是能力、并行是手段
同步阻塞后续操作
异步 Promise/async/await
- 编程范式 { 泛型编程，数据结构+代码



值无法离开类型单独讨论，**类型**一般分为原生类型和组合类型。**指针**和**引用**都指向值的内存地址，只不过二者在解引用时的行为不一样。引用只能解引用到原来的数据类型，而指针没有这个限制，然而，不受约束的指针解引用，会带来内存安全方面的问题。

函数是代码中重复行为的抽象，**方法**是对象内部定义的函数，而**闭包**是一种特殊的函数，它会捕获函数体内使用到的上下文中的自由变量，作为闭包成员的一部分。

而**接口**将调用者和实现者隔离开，大大促进了代码的复用和扩展。面向接口编程可以让系统变得灵活，当使用接口去引用具体的类型时，我们就需要**虚表**来辅助运行时代码的执行。有了虚表，我们可以很方便地进行动态分派，它是运行时多态的基础。

在代码的运行方式中，**并发**是**并行**的基础，是同时与多个任务打交道的能力；并行是并发的体现，是同时处理多个任务的手段。**同步阻塞**后续操作，**异步**允许后续操作。被广泛用于异步操作的 Promise 代表未来某个时刻会得到的结果，async/await 是 Promise 的封装，一般用状态机来实现。

泛型编程通过参数化让数据结构像函数一样延迟绑定，提升其通用性，类型的参数可以用接口约束，使类型满足一定的行为，同时，在使用泛型结构时，我们的代码也需要更高的抽象度。

这些基础概念，这对于后续理解 Rust 的很多概念至关重要。如果你对某些概念还是有些模糊，务必留言，我们可以进一步讨论。

s参考资料

Latency numbers every programmer should know, 对比了 CPU、内存、I/O 设备、和网络的延迟

如何入门

03 | 初窥门径：从你的第一个Rust程序开始！

Rust 安装起来非常方便，你可以用 [rustup.rs](#) 中给出的方法，根据你的操作系统进行安装。比如在 unix 系统下，可以直接运行：

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

这会在你的系统上安装 Rust 工具链，之后，你就可以在本地用 `cargo new` 新建 Rust 项目、尝试 Rust 功能。动起手来，试试用Rust写你的第一个 [hello world](#) 程序吧！

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

你可以使用任何编辑器来撰写 Rust 代码，我个人偏爱 VSCode，因为它免费，功能强大且速度很快。在 VSCode 下我为 Rust 安装了一些插件，下面是我的安装顺序，你可以参考：

1. rust-analyzer：它会实时编译和分析你的 Rust 代码，提示代码中的错误，并对类型进行标注。你也可以使用官方的 rust 插件取代。
2. rust syntax：为代码提供语法高亮。
3. crates：帮助你分析当前项目的依赖是否是最新的版本。
4. better toml：Rust 使用 toml 做项目的配置管理。better toml 可以帮你语法高亮，并展示 toml 文件中的错误。
5. rust test lens：可以帮助你快速运行某个 Rust 测试。
6. Tabnine：基于 AI 的自动补全，可以帮助你更快地撰写代码。

第一个实用的 Rust 程序

首先，我们用 `cargo new scrape_url` 生成一个新项目。默认情况下，这条命令会生成一个可执行项目 `scrape_url`，入口在 `src/main.rs`。我们在 `Cargo.toml` 文件里，加入如下的依赖：

```
[dependencies]
reqwest = { version = "0.11", features = ["blocking"] }
html2md = "0.2"
```

Cargo.toml 是 Rust 项目的配置管理文件，它符合 [toml](#) 的语法。我们为这个项目添加了两个依赖：reqwest 和 html2md。[reqwest](#) 是一个 HTTP 客户端，它的使用方式和 Python 下的 [request](#) 类似；html2md 顾名思义，把 HTML 文本转换成Markdown。

接下来，在 src/main.rs 里，我们为 main() 函数加入以下代码：

```
use std::fs;

fn main() {
    let url = "https://www.rust-lang.org/";
    let output = "rust.md";

    println!("Fetching url: {}", url);
    let body = reqwest::blocking::get(url).unwrap().text().unwrap();

    println!("Converting html to markdown...");
    let md = html2md::parse_html(&body);

    fs::write(output, md.as_bytes()).unwrap();
    println!("Converted markdown has been saved in {}.", output);
}
```

保存后，在命令行下，进入这个项目的目录，运行 `cargo run`，在一段略微漫长的编译后，程序开始运行，在命令行下，你会看到如下的输出：

```
Fetching url: https://www.rust-lang.org/
Converting html to markdown...
Converted markdown has been saved in rust.md.
```

从这段并不长的代码中，我们可以感受到 Rust 的一些基本特点：

首先，Rust 使用名为 `cargo` 的工具来管理项目，它类似 Node.js 的 `npm`、Golang 的 `go`，用来做依赖管理以及开发过程中的任务管理，比如编译、运行、测试、代码格式化等等。

其次，Rust 的整体语法偏 C/C++ 风格。函数体用花括号 `{}` 包裹，表达式之间用分号 `;` 分隔，访问结构体的成员函数或者变量使用点 `.` 运算符，而访问命名空间（namespace）或者对象的静态函数使用双冒号 `::` 运算符。如果要简化对命名空间内部的函数或者数据类型的引用，可以使用 `use` 关键字，比如 `use std::fs`。此外，可执行体的入口函数是 `main()`。

另外，你也很容易看到，Rust 虽然是一门强类型语言，但编译器支持类型推导，这使得写代码时的直观感受和写脚本语言差不多。

很多不习惯类型推导的开发者，觉得这会降低代码的可读性，因为可能需要根据上下文才知道当前变量是什么类型。不过没关系，如果你在编辑器中使用了 rust-analyzer 插件，变量的类型会自动提示出来：

```
use std::fs;
Debug | ► Run | Debug
fn main() {
    let url: &str = "https://www.rust-lang.org/";
    let output: &str = "rust.md";

    println!("Fetching url: {}", url);
    let body: String = reqwest::blocking::get(url).unwrap().text().unwrap();

    println!("Converting html to markdown...");
    let md: String = html2md::parse_html(&body);

    fs::write(path: output, contents: md.as_bytes()).unwrap();
    println!("Converted markdown has been saved in {}.", output);
}
```

最后，Rust 支持宏编程，很多基础的功能比如 `println!()` 都被封装成一个宏，便于开发者写出简洁的代码。

这里例子没有展现出来，但 Rust 还具备的其它特点有：

- Rust 的变量默认是不可变的，如果要修改变量的值，需要显式地使用 `mut` 关键字。
- 除了 `let` / `static` / `const` / `fn` 等少数语句外，Rust 绝大多数代码都是表达式（expression）。所以 `if` / `while` / `for` / `loop` 都会返回一个值，函数最后一个表达式就是函数的返回值，这和函数式编程语言一致。
- Rust 支持面向接口编程和泛型编程。
- Rust 有非常丰富的数据类型和强大的标准库。
- Rust 有非常丰富的控制流程，包括模式匹配（pattern match）。

第一个实用的 Rust 程序就运行成功了，不知道你现在是不是有点迟疑，这些我现在都不太懂怎么办，是不是得先去把这些都掌握了才能继续学？不要迟疑，跟着继续学，后面都会讲到。

接下来，为了快速入门 Rust，我们一起梳理 Rust 开发的基本内容。

这部分涉及的知识在各个编程语言中都大同小异，略微枯燥，但是这一讲是我们后续学习的基础，建议你每段示例代码都写一下，运行一下，并且和自己熟悉的语言对比来加深印象。



基本语法和基础数据类型

首先我们看在 Rust 下，我们如何定义变量、函数和数据结构。

变量和函数

前面说到，Rust 支持类型推导，在编译器能够推导类型的情况下，变量类型一般可以省略，但常量（const）和静态变量（static）必须声明类型。

定义变量的时候，根据需要，你可以添加 `mut` 关键字让变量具备可变性。默认变量不可变是一个很重要的特性，它符合最小权限原则（Principle of Least Privilege），有助于我们写出健壮且正确的代码。当你使用 `mut` 却没有修改变量，Rust 编译期会友好地报警，提示你移除不必要的 `mut`。

在Rust下，函数是一等公民，可以作为参数或者返回值。我们来看一个函数作为参数的例子（[代码](#)）：

```

fn apply(value: i32, f: fn(i32) -> i32) -> i32 {
    &nbsp; &nbsp; f(value)
}

fn square(value: i32) -> i32 {
    &nbsp; &nbsp; value * value
}

fn cube(value: i32) -> i32 {
    &nbsp; &nbsp; value * value * value
}

fn main() {
    &nbsp; &nbsp; println!("apply square: {}", apply(2, square));
    &nbsp; &nbsp; println!("apply cube: {}", apply(2, cube));
}

```

这里 `fn(i32) -> i32` 是 `apply` 函数第二个参数的类型，它表明接受一个函数作为参数，这个传入的函数必须是：参数只有一个，且类型为 `i32`，返回值类型也是 `i32`。

Rust 函数参数的类型和返回值的类型都必须显式定义，如果没有返回值可以省略，返回 `unit`。函数内部如果提前返回，需要用 `return` 关键字，否则最后一个表达式就是其返回值。如果最后一个表达式后添加了 `;` 分号，隐含其返回值为 `unit`。你可以看这个例子（[代码](#)）：

```

fn pi() -> f64 {
    3.1415926
}

fn not_pi() {
    3.1415926;
}

fn main() {
    let is_pi = pi();
    let is_unit1 = not_pi();
    let is_unit2 = {
        pi();
    };

    println!("is_pi: {:?}", is_pi, is_unit1, is_unit2);
}

```

数据结构

了解了函数如何定义后，我们来看看 Rust 下如何定义数据结构。

数据结构是程序的核心组成部分，在对复杂的问题进行建模时，我们就要自定义数据结构。Rust 非常强大，可以用 struct 定义结构体，用 enum 定义标签联合体（tagged union），还可以像 Python 一样随手定义元组（tuple）类型。

比如我们可以这样定义一个聊天服务的数据结构（[代码](#)）：

```
#[derive(Debug, Copy, Clone)]
enum Gender {
    #[allow(dead_code)]
    Unspecified = 0,
    Female = 1,
    Male = 2,
}

#[derive(Debug, Copy, Clone)]
struct UserId(u64);

#[derive(Debug, Copy, Clone)]
struct TopicId(u64);

#[derive(Debug, Clone)]
struct User {
    id: UserId,
    name: String,
    gender: Gender,
}

#[derive(Debug, Clone)]
struct Topic {
    id: TopicId,
    name: String,
    owner: UserId,
}

#[derive(Debug, Clone)]
#[allow(dead_code)]
enum Event {
    Join((UserId, TopicId)),
    Leave((UserId, TopicId)),
    Message((UserId, TopicId, String)),
}

fn main() {
    let alice = User { id: UserId(1), name: "Alice".into(), gender: Gender::Female };
    let bob = User { id: UserId(2), name: "Bob".into(), gender: Gender::Male };

    let topic = Topic { id: TopicId(1), name: "rust".into(), owner: UserId(1) };
    let event1 = Event::Join((alice.id, topic.id));
    let event2 = Event::Join((bob.id, topic.id));
    let event3 = Event::Message((alice.id, topic.id, "Hello world!".into()));

    println!("event1: {:?}", event1, event2, event3);
}
```

简单解释一下：

1. Gender：一个枚举类型，在 Rust 下，使用 enum 可以定义类似 C 的枚举类型
2. UserId/TopicId：struct 的特殊形式，称为元组结构体。它的域都是匿名的，可以用索引访问，适用于简单的结构体。
3. User/Topic：标准的结构体，可以把任何类型组合在结构体里使用。
4. Event：标准的标签联合体，它定义了三种事件：Join、Leave、Message。每种事件都有自己的数据结构。

在定义数据结构的时候，我们一般会加入修饰，为数据结构引入一些额外的行为。在 Rust 里，数据的行为通过 trait 来定义，后续我们会详细介绍 trait，你现在可以暂时认为 trait 定义了数据结构可以实现的接口，类似 Java 中的 interface。

一般我们用 impl 关键字为数据结构实现 trait，但 Rust 贴心地提供了派生宏（derive macro），可以大大简化一些标准接口的定义，比如 `#[derive(Debug)]` 为数据结构实现了 Debug trait，提供了 debug 能力，这样可以通过 `{:?}`，用 `println!` 打印出来。

在定义 UserId / TopicId 时我们还用到了 Copy / Clone 两个派生宏，Clone 让数据结构可以被复制，而 Copy 则让数据结构可以在参数传递的时候自动按字节拷贝。在下一讲所有权中，我会具体讲什么时候需要 Copy。

简单总结一下 Rust 定义变量、函数和数据结构：

	定义	说明	示例	
变量	不可变: let x: T; 可变: let mut x: T;	在栈上分配一个类型为 T 的变量, 变量名为 x。 当声明为可变变量时, x 的内容可以被修改, 且允许可变引用。	let name = "Tyr"; let pi = 3.1415926; let mut v: Vec<u8> = Vec::new();	
常量	const X: T = <value>;	常量是一个右值 (rvalue), 它不能被修改。常量编译后被放入可执行文件的数据段, 全局可访问。	const PI: f64 = 3.1415926;	
静态变量	static X: T = T::new(); static mut X: T = T::new();	静态变量和常量一样全局可访问, 它也被编入可执行文件的数据段中。静态变量可以被声明为可变。 在使用静态变量时, 由于一些限制, lazy_static是一个很好的工具。	// 可以编译通过 static V: Vec<u8> = Vec::new(); // 无法编译通过 (需要使用 lazy_static) static MAP: HashMap<String, String> = HashMap::new();	
函数	fn x(a1: T1, ...) -> T {}	在 Rust 中, 函数如果没有返回值, 那么其返回值为 unit, 符号是 ()。	fn valid_email(input: &str) -> bool { ... }	
结构体	struct S {...}	结构体有三种形式: 1. 空结构体, 不占任何内存空间; 2. 元组结构体, struct 的每个域都是匿名的, 可以通过索引访问; 3. 普通结构体, struct 的每个域都有名字, 可以通过名称访问。	struct Marker; struct Color(u8, u8, u8);	struct Person { name: String, age: u8, }
enum	enum E { ... }	enum 有两种形式: 1. 标签联合。enum 可以承载多个不同的数据结构中的一种。 2. 枚举	enum Option<T> { Some(T), None, }	enum Status { Ok = 0, BadName = 1, NotFound = 2, ... }

极客时间

控制流程

程序的基本控制流程分为以下几种, 我们应该都很熟悉了, 重点看如何在 Rust 中运行。

顺序执行就是一行行代码往下执行。在执行的过程中, 遇到函数, 会发生函数调用。**函数调用**是代码在执行过程中, 调用另一个函数, 跳入其上下文执行, 直到返回。

Rust 的**循环**和大部分语言都一致, 支持死循环 loop、条件循环 while, 以及对迭代器的循环 for。循环可以通过 break 提前终止, 或者 continue 来跳到下一轮循环。

满足某个条件时会跳转，Rust 支持分支跳转、模式匹配、错误跳转和异步跳转。

- 分支跳转就是我们熟悉的 if/else；
- Rust 的模式匹配可以通过匹配表达式或者值的某部分的内容，来进行分支跳转；
- 在错误跳转中，当调用的函数返回错误时，Rust 会提前终止当前函数的执行，向上一层返回错误。
- 在 Rust 的异步跳转中，当 async 函数执行 await 时，程序当前上下文可能被阻塞，执行流程会跳转到另一个异步任务执行，直至 await 不再阻塞。

我们通过斐波那契数列，使用 if 和 loop / while / for 这几种循环，来实现程序的基本控制流程（[代码](#)）：

```
fn fib_loop(n: u8) {  
    let mut a = 1;  
    let mut b = 1;  
    let mut i = 2u8;  
  
    println!("\\nfib_loop:");
```

```
    loop {  
        let c = a + b;  
        a = b;  
        b = c;  
        i += 1;  
  
        println!("next val is {}", b);
```

```
        if i >= n {  
            break;  
        }  
    }  
}
```

```
fn fib_while(n: u8) {  
    let (mut a, mut b, mut i) = (1, 1, 2);
```

```
    println!("\\nfib_while:");
```

```
    while i < n {  
        let c = a + b;  
        a = b;  
        b = c;  
        i += 1;  
  
        println!("next val is {}", b);
```

```
    }
```

```
}
```

```
fn fib_for(n: u8) {  
    let (mut a, mut b) = (1, 1);  
  
    println!("\\nfib_for:");
```

```
for _i in 2..n {
    let c = a + b;
    a = b;
    b = c;
    println!("next val is {}", b);
}
}

fn main() {
    let n = 10;
    fib_loop(n);
    fib_while(n);
    fib_for(n);
}
```

这里需要指出的是，Rust 的 for 循环可以用于任何实现了 `Intolterator` trait 的数据结构。

在执行过程中，`Intolterator` 会生成一个迭代器，for 循环不断从迭代器中取值，直到迭代器返回 `None` 为止。因而，for 循环实际上只是一个语法糖，编译器会将其展开使用 loop 循环对迭代器进行循环访问，直至返回 `None`。

在 `fib_for` 函数中，我们还看到 `2...n` 这样的语法，想必 Python 开发者一眼就能明白这是 Range 操作，`2...n` 包含 $2 \leq x < n$ 的所有值。和 Python 一样，在 Rust 中，你也可以省略 Range 的下标或者上标，比如：

```
let arr = [1, 2, 3];
assert_eq!(arr[..], [1, 2, 3]);
assert_eq!(arr[0..=1], [1, 2]);
```

和 Python 不同的是，Range 不支持负数，所以你不能使用 `arr[1..=1]` 这样的代码。这是因为，Range 的下标上标都是 `usize` 类型，不能为负数。

下表是 Rust 主要控制流程的一个总结：

功能	说明	示例
死循环 loop {}	无限循环，除非内部调用 break 强制退出。	<pre>loop { println!("hello forever!"); }</pre> <pre>let mut i = 1; loop { println!("hello for {} times", i); if i >= 10 { break; } i += 1; }</pre>
条件循环 while expr {}	如果表达式 expr 为真，就一直循环，直到表达式 expr 为假。	<pre>let mut i = 1; while i < 10 { println!("hello for {} times", i); i += 1; }</pre>
迭代器循环 for x in iter {}	如果迭代器一直能取出下一个值，就一直循环，直到迭代器返回 None。	<pre>for i in 0..10 { println!("hello for {} times", i); }</pre>
分支跳转 if expr {} else {}	如果表达式为真，执行随后的代码块，否则执行 else 后的代码块。	<pre>if age < 13 { println!("kids!"); } else { println!("not kids!"); }</pre>
其它跳转	退出循环: break 跳转到下一次循环: continue 退出函数: return expr	<pre>let mut i = 1; while i < 10 { if i == 5 { continue; } println!("hello for {} times", i); i += 1; }</pre>
模式匹配 match expr {} if let pat = expr {}	根据表达式所有可能的值进行匹配，进行相应的处理。 if let 是 match 的简写，用于只关心某种模式匹配的情况。	<pre>match result { Ok(value) => Ok(value), Err(err) => return Err(err.into()), }</pre> <pre>if let Some(name) = Some("Tyr") { println!("Hello {}", name); } else { println!("You don't have name"); }</pre>
错误跳转 expr?	当错误发生时退出函数，并返回错误	<pre>fs::write("/tmp/test", b"hello")?;</pre>
异步处理 expr.await	执行 async 函数，直至其完成，在这个过程中当前上下文可能被阻塞零到多次。	<pre>socket.write(data).await?;</pre>

模式匹配

Rust 的模式匹配吸取了函数式编程语言的优点，强大优雅且效率很高。它可以用于 struct / enum 中匹配部分或者全部内容，比如上文中我们设计的数据结构 Event，可以这样匹配（[代码](#)）：

```
fn process_event(event: &Event) {
    match event {
```

```

Event::Join((uid, _tid)) => println!("user {:?} joined", uid),
Event::Leave((uid, tid)) => println!("user {:?} left {:?}", uid, tid),
Event::Message(_, _, msg)) => println!("broadcast: {}", msg),
}
}

```

从代码中我们可以看到，可以直接对 enum 内层的数据进行匹配并赋值，这比很多只支持简单模式匹配的语言，例如 JavaScript、Python，可以省出好几行代码。

除了使用 match 关键字做模式匹配外，我们还可以用 if let / while let 做简单的匹配，如果上面的代码我们只关心 Event::Message，可以这么写 ([代码](#))：

```

fn process_message(event: &Event) {
    if let Event::Message(_, _, msg) = event {
        println!("broadcast: {}", msg);
    }
}

```

Rust 的模式匹配是一个很重要的语言特性，被广泛应用在状态机处理、消息处理和错误处理中，如果你之前使用的语言是 C / Java / Python / JavaScript，没有强大的模式匹配支持，要好好练习这一块。

错误处理

Rust 没有沿用 C++/Java 等诸多前辈使用的异常处理方式，而是借鉴 Haskell，**把错误封装在 Result<T, E>类型中，同时提供了 ? 操作符来传播错误**，方便开发。Result<T, E> 类型是一个泛型数据结构，T 代表成功执行返回的结果类型，E 代表错误类型。

```

use std::fs;
fn main() {
    let url = "https://www.rust-lang.org/";
    let output = "rust.md";

    println!("Fetching url: {}", url);
    let body = reqwest::blocking::get(url).unwrap().text().unwrap();

    println!("Converting html to markdown... ");
    let md = html2md::parse_html(&body);

    fs::write(output, md.as_bytes()).unwrap();
    println!("Converted markdown has been saved in {}.", output);
}

```

如果想让错误传播，可以把所有的 unwrap() 换成 ? 操作符，并让 main() 函数返回一个 Result<T, E>，如下所示：

```
use std::fs;
// main 函数现在返回一个 Result
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let url = "https://www.rust-lang.org/";
    let output = "rust.md";

    println!("Fetching url: {}", url);
    let body = reqwest::blocking::get(url)?.text()?;

    println!("Converting html to markdown...");
    let md = html2md::parse_html(&body);

    fs::write(output, md.as_bytes())?;
    println!("Converted markdown has been saved in {}.", output);

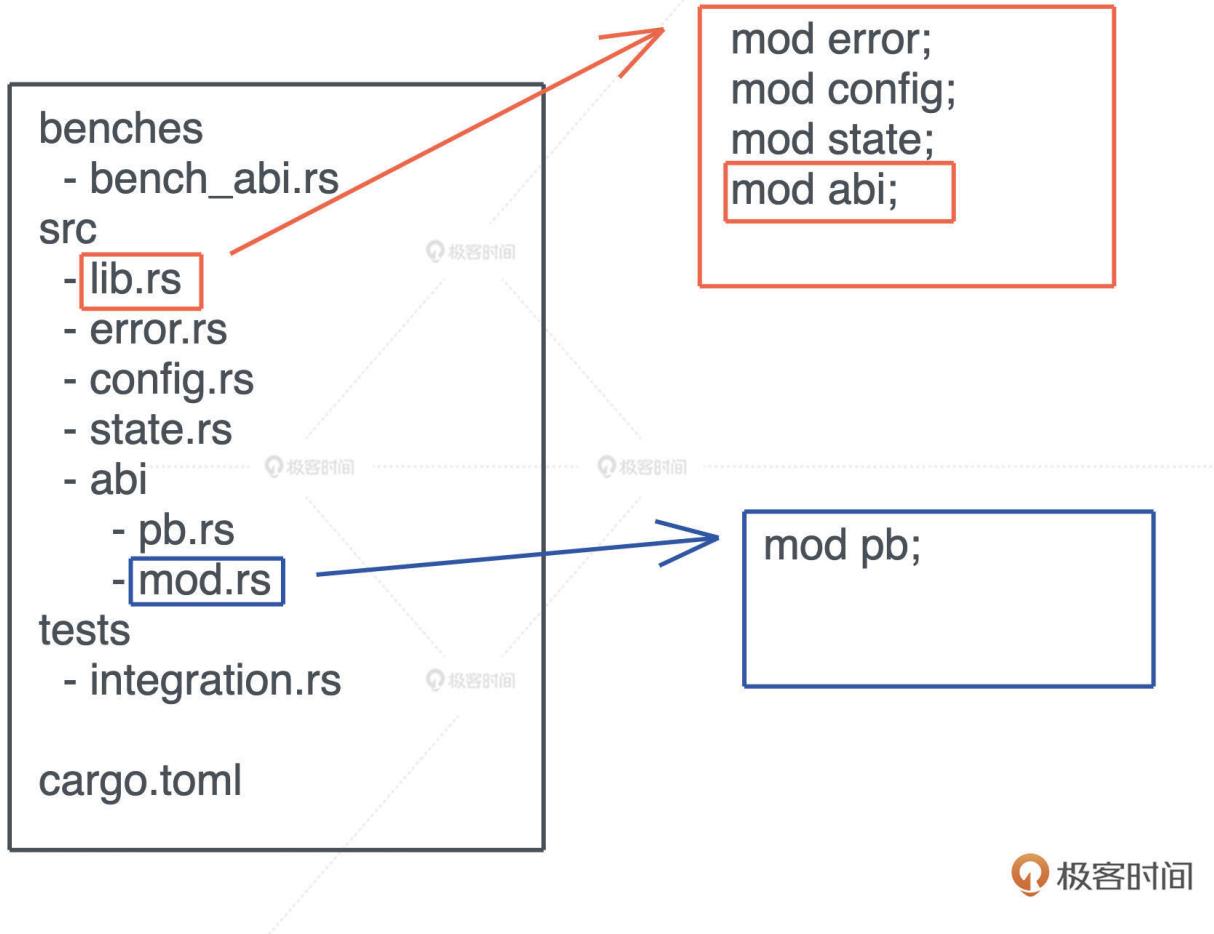
    Ok(())
}
```

关于错误处理我们先讲这么多，之后我们会单开一讲，对比其他语言，来详细学习 Rust 的错误处理。

Rust 项目的组织

当 Rust 代码规模越来越大时，我们就无法用单一文件承载代码了，需要多个文件甚至多个目录协同工作，这时我们可以用 **mod** 来组织代码。

具体做法是：在项目的入口文件 lib.rs / main.rs 里，用 mod 来声明要加载的其它代码文件。如果模块内容比较多，可以放在一个目录下，在该目录下放一个 mod.rs 引入该模块的其它文件。这个文件，和 Python 的 `__init__.py` 有异曲同工之妙。这样处理之后，就可以用 mod + 目录名引入这个模块了，如下图所示：



在 Rust 里，一个项目也被称为一个 **crate**。crate 可以是可执行项目，也可以是一个库，我们可以用 `cargo new <name> -- lib` 来创建一个库。当 crate 里的代码改变时，这个 crate 需要被重新编译。

在一个 crate 下，除了项目的源代码，单元测试和集成测试的代码也会放在 crate 里。

Rust 的单元测试一般放在和被测代码相同的文件中，使用条件编译 `#[cfg(test)]` 来确保测试代码只在测试环境下编译。以下是一个[单元测试](#)的例子：

```

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}

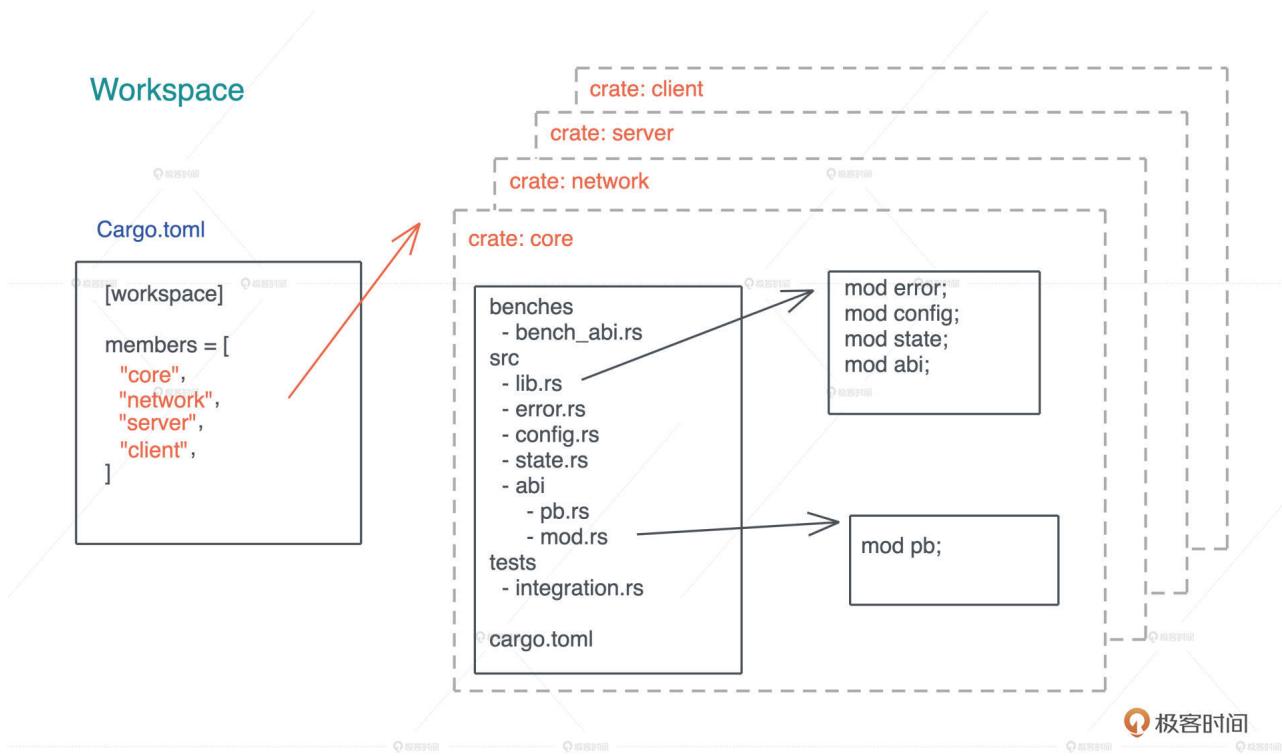
```

集成测试一般放在 tests 目录下，和 src 平行。和单元测试不同，集成测试只能测试 crate 下的公开接口，编译时编译成单独的可执行文件。

在 crate 下，如果要运行测试用例，可以使用 `cargo test`。

当代码规模继续增长，把所有代码放在一个 crate 里就不是一个好主意了，因为任何代码的修改都会导致这个 crate 重新编译，这样效率不高。我们可以使用 **workspace**。

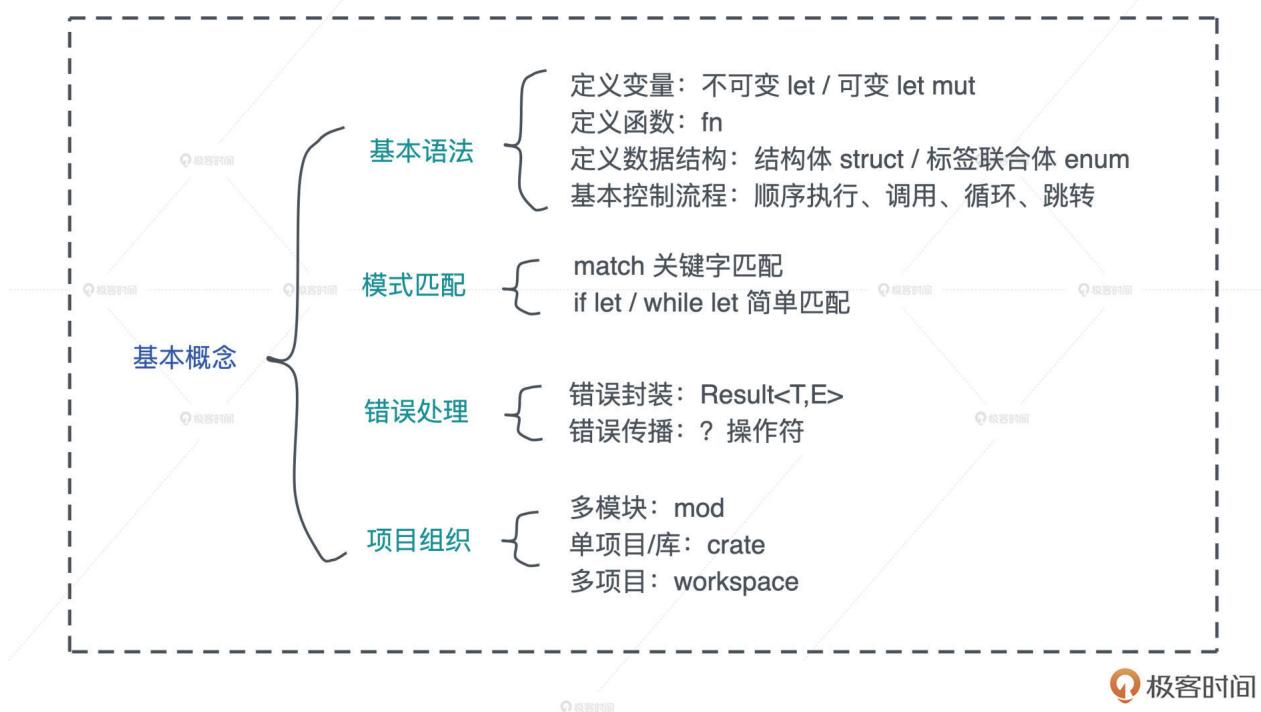
一个 workspace 可以包含一到多个 crates，当代码发生改变时，只有涉及的 crates 才需要重新编译。当我们构建一个 workspace 时，需要先在某个目录下生成一个如图所示的 Cargo.toml，包含 workspace 里所有的 crates，然后可以 `cargo new` 生成对应的 crates：



小结

我们简单梳理了 Rust 的基本概念。通过 let/let mut 定义变量、用 fn 定义函数、用 struct / enum 定义复杂的数据结构，也学习了 Rust 的基本的控制流程，了解了模式匹配如何运作，知道如何处理错误。

最后考虑到代码规模问题，介绍了如何使用 mod、crate 和 workspace 来组织 Rust 代码。我总结到图中你可以看看。



参考资料

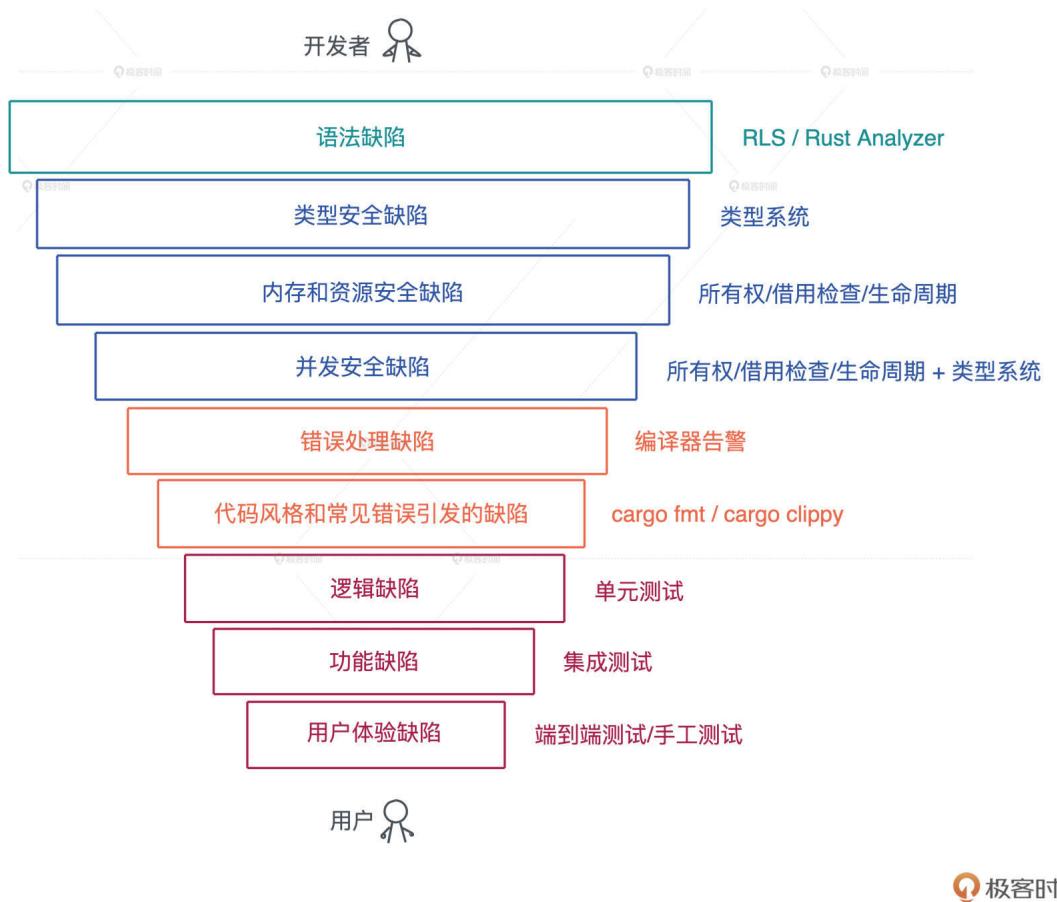
1. [TOML](#)
2. [static 关键字](#)
3. [lazy_static](#)
4. [unit 类型](#)
5. [How to write tests](#)
6. [More about cargo and crates.io](#)
7. Rust 支持声明宏 (declarative macro) 和过程宏 (procedure macro) , 其中过程宏又包含三种方式：函数宏 (function macro) , 派生宏 (derive macro) 和属性宏 (attribute macro) 。println! 是函数宏，是因为 Rust 是强类型语言，函数的类型需要在编译期敲定，而 println! 接受任意个数的参数，所以只能用宏来表达。

04 | rust怎么学,是否值得学?

代码缺陷

从软件开发的角度来看，一个软件系统想要提供具有良好用户体验的功能，**最基本的要求就是控制缺陷**。为了控制缺陷，在软件工程中，我们定义了各种各样的流程，从代码的格式，到 linting，到 code review，再到单元测试、集成测试、手工测试。

所有这些手段就像一个个漏斗，不断筛查代码，把缺陷一层层过滤掉，让软件在交付到用户时尽善尽美。我画了一张图，将在开发过程中可能出现的缺陷分了类，从上往下看：



(课程里的图片都是用 `excalidraw` 绘制的)

语法缺陷

首先在我们开始写代码的时候，在语法层面可能会出现小问题，比如说初学者会对某些语法点不太熟悉，资深工程师在用一些不常用的语法时也会出现语法缺陷。

对于这个缺陷，目前大部分的编程语言都会在你写代码的时候，给到详尽的提示，告诉你语法错误出现在哪里。

对 Rust 来说，它提供了 Rust Language Server / Rust Analyzer 第一时间报告语法错误，如果你用第三方 IDE 如 VSCode，会有这些工具的集成。

类型安全缺陷

然后就是类型方面的缺陷，这类缺陷需要语言本身的类型系统，帮助你把缺陷找出来，所以大部分非类型安全的语言，对这类错误就束手无策了。

以Python/Elixir为例，如果你期望函数的参数使用类型A，但是实际用了类型B，这种错误只有你的代码在真正运行的时候才能被检查出来，相当于把错误发现的时机大大延后了。

所以现在很多脚本语言也倾向尽可能让开发者多写一些类型标注，但因为它不是语言原生的部分，所以也很难强制，在实际写脚本语言的代码时，你需要特别注意类型安全。

内存和资源安全缺陷

几乎所有的语言中都会有内存安全问题。

对于内存自动管理的语言来说，自动管理机制可以帮你解决大部分内存问题，不会出现内存使用了没有释放、使用了已释放内存、使用了悬停指针等等情况。

我们之前也讲到了，大部分语言，如 Java / Python / Golang / Elixir 等，他们通过语言的运行时解决了内存安全问题。

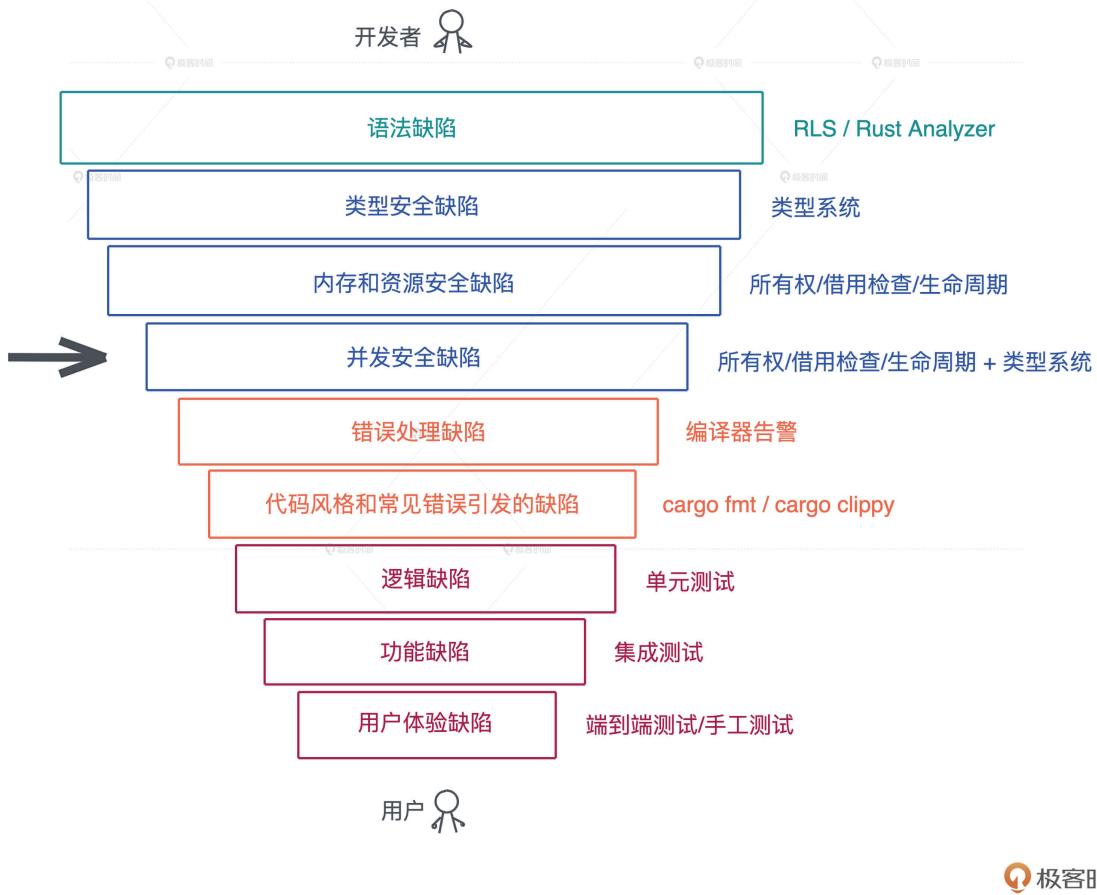
但是这只是大部分被解决了，还有比如逻辑上存在的内存泄漏的问题，比如一个带 TTL 的缓存，如果没设计好，表中的内容超时后并没有被删除，就会导致内存使用一直增长。这种因为设计缺陷导致的内存泄漏，现在所有语言都没有能够解决这个问题，只能说尽可能地解决。

资源安全缺陷也是大部分语言都会有的问题，诸如文件/socket 这样的资源，如果分配出来但没有很好释放，就会带来资源的泄漏，支持 GC 的语言对此也无能为力，很多时候只能靠程序员手工释放。

然而资源的释放并不简单，尤其是在做异常处理或者非正常流程的时候，很容易忘记要释放已经分配的资源。

Rust 可以说基本上解决了主要的内存和资源的安全问题，通过所有权、借用检查和生命周期检查，来保证内存和资源一旦被分配，在其生命周期结束时，会被释放掉。

并发安全缺陷



Q 极客时间

这个问题

发生在支持多线程的语言中，比如说两个线程间访问同一个变量，如果没有做合适的临界区保护，就很容易发生并发安全问题。

Rust 通过所有权规则和类型系统，主要是两个 trait: Send/Sync 来解决这个问题。

很多高级语言会把线程概念屏蔽掉，只允许开发者使用语言提供的运行时来保证并发安全，比如Golang 要使用 channel 和 Goroutine 、Erlang 只能用 Erlang process，只要你在它这个框架下，并发处理就是安全的。

这样可以处理绝大多数并发场景，但遇到某些情况就容易导致效率不高，甚至阻塞其它并发任务。比如当有一个长时间运行的 CPU 密集型任务，使用单独的线程来处理要好得多。

处理并发有很多手段，但是大部分语言为了并发安全，把不少手段都屏蔽了，开发者无法接触到，但是Rust都提供给你，同时还提供了很好的并发安全保障，让你可以在合适的场景，安全地使用合适的工具。

错误处理缺陷

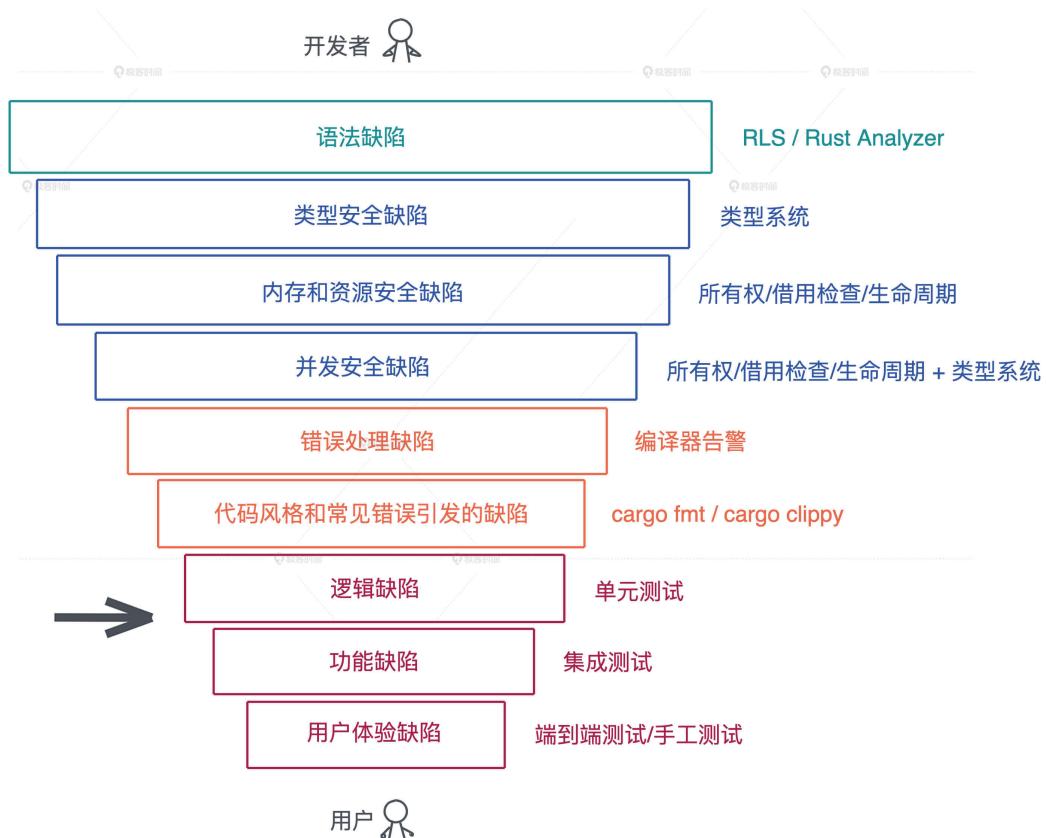
错误处理作为代码的一个分支，会占到代码量的30%甚至更多。在实际工程中，函数频繁嵌套的时候，整个过程会变得非常复杂，一旦处理不好就会引入缺陷。常见的问题是系统出错了，但抛出的错误并没有得到处理，导致程序在后续的运行中崩溃。

很多语言并没有强制开发者一定要处理错误，Rust 使用 `Result<T, E>` 类型来保证错误的类型安全，还强制你必须处理这个类型返回的值，避免开发者丢弃错误。

代码风格和常见错误引发的缺陷

很多语言都会提供代码格式化工具和 linter 来消灭这类缺陷。Rust 有内置的 `cargo fmt` 和 `cargo clippy` 来帮助开发者统一代码风格，来避免常见的开发错误。

再往下的三类缺陷是语言和编译器无法帮助解决的。



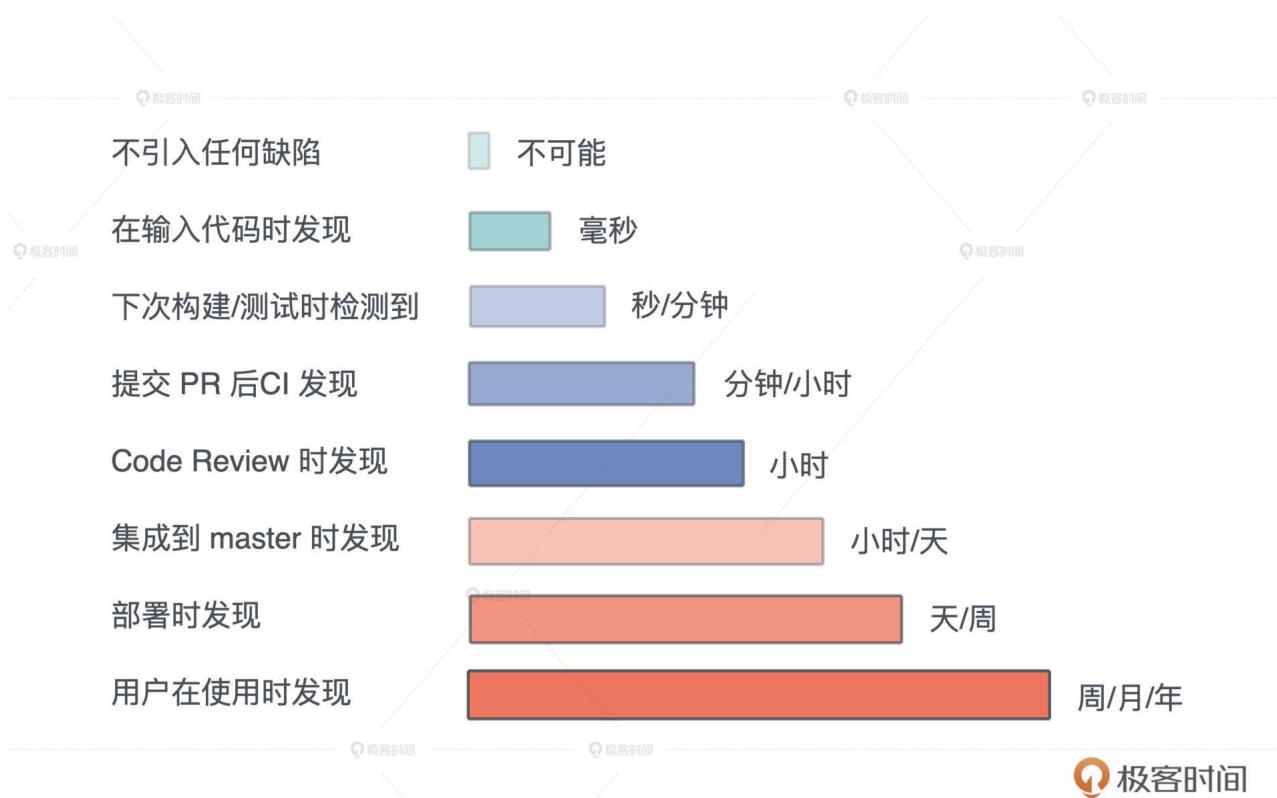
- 对于逻辑缺陷，我们需要有不错的单元测试覆盖率；
- 对于功能缺陷，需要通过足够好的集成测试，把用户主要使用的功能测试一遍；
- 对于用户体验缺陷，需要端到端的测试，甚至手工测试，才能发现。

从上述介绍中你可以看到，Rust 帮我们把尽可能多的缺陷扼杀在摇篮中。Rust 在编译时解决掉的很多缺陷，如资源释放安全、并发安全和错误处理方面的缺陷，在其他大多数语言中并没有完整的解决方案。

所以 Rust 这门语言，让开发者的时间和精力都尽可能的放在对逻辑、功能、用户体验缺陷的优化上。

引入缺陷的代价

我们再来从引入缺陷的代价这个角度来看，Rust 这样的处理方式到底有什么好处。



首先，任何系统不引入缺陷是不可能的。

如果在写代码的时候就发现缺陷，纠正的时间是毫秒到秒级；如果在测试的时候检测出来，那可能是秒到分钟级。以此类推，如果缺陷在从code review 到集成到master才被发现，那时间就非常长。

如果一直到用户使用的时候才发现，那可能是以周、月，甚至以年为单位。我之前做防火墙系统时，一个新功能的 bug 往往在一年甚至两年之后，才在用户的生产环境中被暴露出来，这个时候再去解决缺陷的代价就非常大。

所以Rust在设计之初，尽可能把大量缺陷在编译期，在秒和分钟级就替你检测出来，让你修改，不至于把缺陷带到后续环境，最大程度的保证代码质量。

这也是为什么虽然 Rust 初学者前期需要和编译器做艰难斗争，但这是非常值得的，只要你跨过了这道坎，能够让代码编译通过，基本上你代码的安全性没有太大问题。

语言发展前景判断

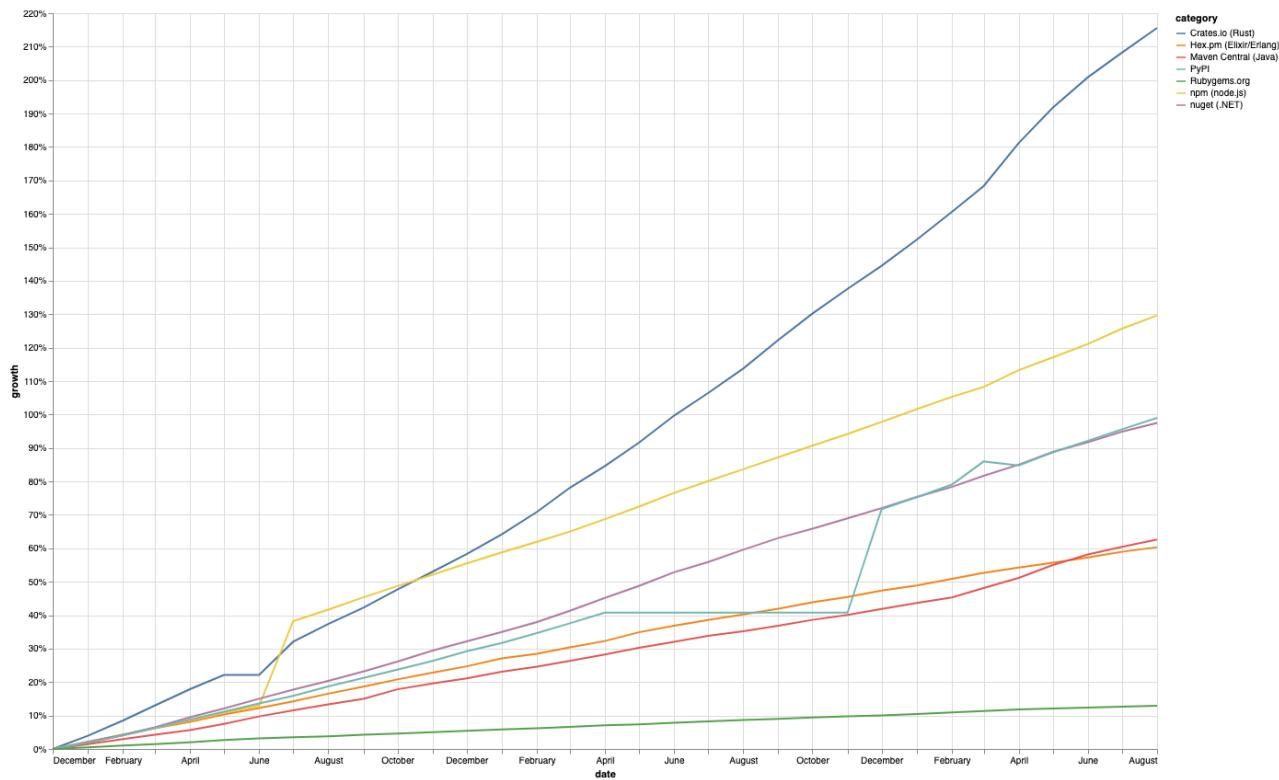
有很多同学比较关心 Rust 的发展前景，留言问 Rust 和其他语言的对比，经常会聊现在或者未来什么语言会被Rust替代、Rust会不会一统前后端天下等等。我觉得不会。

每种语言都有它们各自的优劣和适用场景，谈不上谁一定取代谁。 社区的形成、兴盛和衰亡是一个长久的过程，就像“世界上最好的语言 PHP”也还在顽强地生长着。

那么如何判断一门新的语言的发展前景呢？下图是我用 pandas 处理过的 [modulecounts](#) 的数据，这个数据统计了主流语言的库的数量。可以看到 2019 年初 Rust crates 的起点并不高，只有两万出头，两年后就有六万多。

	Crates.io (Rust)	Hex.pm (Elixir/Erlang)	Maven Central (Java)	npm (node.js)	nuget (.NET)	PyPI	Rubygems.org
date							
2018/12/01	20601.0	7360.0	258811.0	734982.0	135480.0	160265.0	148270.0
2019/01/01	21419.0	7498.0	262534.0	750665.0	138333.0	163292.0	148999.0
2019/02/01	22359.0	7661.0	266448.0	766704.0	141283.0	166760.0	149790.0
2019/03/01	23277.0	7815.0	269822.0	781105.0	144212.0	170256.0	150454.0
2019/04/01	24283.0	7959.0	273407.0	798153.0	148326.0	174331.0	151276.0
2019/05/01	25161.0	8119.0	278170.0	814491.0	151856.0	178139.0	152257.0
2019/06/01	25161.0	8261.0	283985.0	830442.0	155829.0	182054.0	152983.0
2019/07/01	27205.0	8406.0	288759.0	1015666.0	159480.0	185707.0	153514.0
2019/08/01	28304.0	8577.0	293333.0	1041301.0	163054.0	190209.0	153923.0
2019/09/01	29299.0	8733.0	297552.0	1068275.0	166885.0	194286.0	154550.0
2019/10/01	30421.0	8891.0	304997.0	1093050.0	170890.0	198236.0	155098.0
2019/11/01	31521.0	9041.0	309513.0	1117881.0	175334.0	202493.0	155738.0
2019/12/01	32607.0	9179.0	313441.0	1142725.0	179039.0	207107.0	156366.0
2020/01/01	33826.0	9355.0	318565.0	1166944.0	182861.0	211036.0	156929.0
2020/02/01	35202.0	9455.0	322513.0	1190347.0	186915.0	215760.0	157481.0
2020/03/01	36688.0	9595.0	326888.0	1212805.0	191483.0	220474.0	158037.0
2020/04/01	38030.0	9736.0	331899.0	1240034.0	196663.0	225607.0	158732.0
2020/05/01	39465.0	9928.0	337090.0	1267538.0	201507.0	225607.0	159205.0
2020/06/01	41134.0	10074.0	341740.0	1297522.0	207063.0	225607.0	159855.0
2020/07/01	42525.0	10197.0	346377.0	1323647.0	211175.0	225607.0	160477.0
2020/08/01	44035.0	10318.0	349931.0	1349657.0	216203.0	225607.0	161068.0
2020/09/01	45802.0	10445.0	354139.0	1376082.0	220938.0	225607.0	161622.0
2020/10/01	47419.0	10587.0	358717.0	1401507.0	224704.0	225607.0	162265.0
2020/11/01	48939.0	10706.0	362494.0	1427107.0	228906.0	225607.0	162732.0
2020/12/01	50362.0	10847.0	367189.0	1453690.0	233037.0	275178.0	163201.0
2021/01/01	51971.0	10959.0	371748.0	1481737.0	237554.0	280926.0	163715.0
2021/02/01	53683.0	11103.0	376075.0	1508867.0	241627.0	286978.0	164527.0
2021/03/01	55276.0	11235.0	383305.0	1530451.0	246108.0	298037.0	165094.0
2021/04/01	57930.0	11351.0	391200.0	1567491.0	250664.0	296139.0	165757.0
2021/05/01	60101.0	11457.0	401171.0	1595562.0	255836.0	302403.0	166221.0
2021/06/01	61978.0	11576.0	409289.0	1624932.0	259718.0	307966.0	166595.0
2021/07/01	63491.0	11699.0	415277.0	1658603.0	264050.0	313476.0	167011.0
2021/08/01	65019.0	11800.0	420964.0	1687759.0	267522.0	318927.0	167394.0

作为一门新的语言，Rust 生态虽然绝对数量不高，但增长率一直遥遥领先，过去两年多的增长速度差不多是第二名 NPM 的两倍。很遗憾，Golang 的库没有一个比较好的统计渠道，所以这里没法比较 Golang 的数据。但和 JavaScript / Java / Python 等语言的对比足以说明 Rust 的潜力。



Rust 和 Golang

很多同学关心 Rust 和 Golang 的对比，其实网上有很多详尽的分析，[这一篇](#)比较不错可以看看。我这里也简单说一下。

Rust 和 Golang 重叠的领域主要在服务开发领域。

Golang 的优点是简单、上手快，语言已经给你安排好了并发模型，直接用即可。对于日程紧迫、有很多服务要写，且不在乎极致性能的开发团队，Golang 是不错的选择。

Golang 因为设计之初要考虑如何能适应新时代的并发需求，所以使用了运行时、使用调度器调度 Goroutine，在Golang 中内存是不需要开发者手动释放的，所以运行时中还有GC来帮助开发者管理内存。

另外，**为了语法简便，在语言诞生之初便不支持泛型**，这也是目前 Golang 最被诟病的一点，因为一旦系统复杂到一定程度，你的每个类型都需要做一遍实现。

Golang 可能在 2022 年的 1.18 版本添加对泛型的支持，但泛型对 Golang 来说是一把达摩克利斯之剑，它带来很多好处，但同时会大大破坏 Golang 的简洁和极速的编译体验，到时候可能会带给开发者这样一种困惑：既然 Golang 已经变得不简单，不那么容易上手，我为何不学 Rust 呢？

Rust 的很多设计思路和 Golang 相反。

Go 相对小巧，类型系统很简单；而 Rust 借鉴了 Haskell，有完整的类型系统，支持泛型。为了性能的考虑，Rust 在处理泛型函数的时候会做**单态化（Monomorphization）**，泛型函数里每个用到的类型会编译出一份代码，这也是为什么在编译的时候 Rust 编译速度如此缓慢。

Rust 面向系统级的开发，Go 虽然想做新时代的 C，但是它并不适合面向系统级开发，使用场景更多是应用程序、服务等的开发，因为它的庞大的运行时，决定了它不适合做直接和机器打交道的底层开发。

Rust 的诞生目标就是取代 C/C++，想要做出更好的系统层面的开发工具，所以在语言设计之初就要求不能有运行时。所以你看到的类似 Golang 运行时的库比如 Tokio，都是第三方库，不在语言核心中，这样可以把是否需要引入运行时的自由度给到开发者。

Rust 社区里有句话说得好：

Go for the code that has to ship tomorrow, Rust for the code that has to keep running for the next five years.

所以，我对 Rust 的前途持非常乐观的态度。它在系统开发层面可以取代一部分 C/C++ 的场景、在服务开发层面可以和 Java/Golang 竞争、在高性能前端应用通过编译成 WebAssembly，可以部分取代 JavaScript，同时，它又可以方便地通过 FFI 为各种流行的脚本语言提供安全的、高性能的底层库。

我觉得在整个编程语言的生态里，未来 Rust 会像水一样，无处不在且善利万物。

最后给你分享一下我在学习 Rust 的过程中觉得不错的一些资料，也顺带会说明怎么配合这门课程使用。

官方学习资料

Rust 社区里就有大量的学习资料供我们使用。

首先是官方的 [Rust book](#)，它涵盖了语言的方方面面，是入门 Rust 最权威的免费资料。不过这本书比较细碎，有些需要重点解释的内容又一笔带过，让人读完还是云里雾里的。

我记得当时学习 Deref trait 时，官方文档这段文字直接把我看懵了：

Rust does deref coercion when it finds types and trait implementations in three cases:

- From &T to &U when T: Deref<Target=U>
- From &mut T to &mut U when T: DerefMut<Target=U>
- From &mut T to &U when T: Deref<Target=U>

所以我觉得这本书适合学习语言的概貌，对于一时理解不了的内容，需要自己花时间另找资料，或者自己通过练习来掌握。在学习课程的过程中，如果你想巩固所学的内容，可以翻阅这本书。

另外一本官方的 [Rust 死灵书 \(The Rustonomicon\)](#)，讲述 Rust 的高级特性，主要是如何撰写和使用 unsafe Rust，内容不适合初学者。建议在学习完课程之后，或者起码学完进阶内容之后，再阅读这本书。

Rust 代码的文档系统 [docs.rs](#) 是所有编程语言中使用起来最舒服，也是体验最一致的。无论是标准库的文档，还是第三方库的文档，都是用相同的工具生成的，非常便于阅读，你自己撰写的 crate，发布后也会放在 [docs.rs](#) 里。在平时学习和撰写代码的时候，用好这些文档会对你的学习效率和开发效率大有裨益。

[标准库的文档](#) 建议你在学到某个数据类型或者概念时再去阅读，在每一讲中涉及的内容，我都会放上标准库的链接，你可以延伸阅读。

为了帮助 Rust 初学者进一步巩固 Rust 学习的效果，Rust 官方还出品了 [rustlings](#)，它涵盖了大量的小练习，可以用来夯实对知识和概念的理解。有兴趣、有余力的同学可以尝试一下。

其他学习资料

说完了官方的资料，我们看看其它关于 Rust 的内容包括书籍、博客、视频。

首先讲几本书。第一本是汉东的《[Rust 编程之道](#)》，详尽深入，是不可多得的 Rust 中文书。汉东在极客时间有一门 Rust 视频课程，如果你感兴趣，也可以订阅。英文书有 [Programming Rust](#)，目前出了第二版，我读过第一版，写得不错，面面俱到，适合从头读到尾，也适合查漏补缺。

除了书籍相关的资料，我还订阅了一些不错的博客和公众号，也分享给你。博客我主要会看 [This week in Rust](#)，你可以订阅其邮件列表，每期扫一下感兴趣的主题再深度阅读。

公众号主要用于获取信息，可以了解社区的一些动态，有Rust 语言中文社区、Rust 碎碎念，这两个公众号有时会推 This week in Rust 里的内容，甚至会有翻译。

还有一个非常棒的内容来源是 [Rust 语言开源杂志](#)，每月一期，囊括了大量优秀的 Rust 文章。不过这个杂志的主要受众，我感觉还是对 Rust 有一定掌握的开发者，建议你在学完了进阶篇后再读里面的文章效果更好。

在 Rust 社区里，也有很多不错的视频资源。社区里不少人推荐 [Beginner's Series to: Rust](#)，这是微软推出的一系列 Rust 培训，比较新。我简单看了一下还不错，讲得有些慢，可以 1.5 倍速播放节省时间。我自己主要订阅了 [Jon Gjengset](#) 的 YouTube 频道，他的视频面向中高级 Rust 用户，适合学习完本课程后再去观看。

国内视频的话，在 bilibili 上，也有大量的 Rust 培训资料，但需要自己先甄别。我做了几期“程序君的 Rust 培训”感兴趣也可以看看，可以作为课程的补充资料。

说这么多，希望你能够坚定对学习 Rust 的信心。相信我，**不管你未来是否使用 Rust，单单是学习 Rust 的过程，就能让你成为一个更好的程序员。**

欢迎你在留言区分享你的想法，我们一起讨论。

参考资料

- 1.配合课程使用：官方的 [Rust book](#)、微软推出的一系列 Rust 培训 [Beginner's Series to: Rust](#)、英文书 [Programming Rust](#) 查漏补缺
- 2.学完课程后进阶学习：官方的 [Rust 死灵书 \(The Rustonomicon\)](#)、每月一期的 [Rust 语言开源杂志](#)、[Jon Gjengset](#) 的 YouTube 频道、张汉东的《[Rust 编程之道](#)》、我的B站上的“程序君的 Rust 培训”系列。
- 3.学有余力的练习：[Rust 代码的文档系统 docs.rs](#)、小练习 [rustlings](#)
- 4.社区动态：博客 [This week in Rust](#)、公众号 Rust 语言中文社区、公众号 Rust 碎碎念
- 5.如果你对这个专栏怎么学还有疑惑，欢迎围观几个同学的学习方法和经历，在课程目录最后的“学习锦囊”系列，听听课代表们怎么说，相互借鉴，共同进步。直达链接也贴在这里：[学习锦囊（一）](#)、[学习锦囊（二）](#)、[学习锦囊（三）](#)

01 基础篇

所有权

总结

三原则

1. 每个值(资源)都有一个所有者 (变量)， 并非所有变量都拥有资源 (例如引用)
2. 值在任意时刻只有一个所有者
3. 所有者离开作用域， 值将被丢弃 (除非值被转移给另一个变量)

Move语义与Copy语义

1. 赋值或传参会导致值Move， 所有权转移
2. 如果值实现了Copy trait， 那么赋值或传参会使用Copy语义， 按位拷贝
3. 原生类型默认实现了copy trait， 如函数、不可变引用和裸指针， 数组和元组， 换句话说， 存在栈上的数据

借用 (引用)

1. 值可以多个只读引用， 只读引用默认实现了Copy语义
2. 任意时刻只能有一个活跃的可变引用， 可变引用和只读引用是互斥的关系
3. 引用的生命周期不能超出值的生命周期

共享所有权

访问方式		数据	不可变借用	可变借用
单一所有权 (外部可见性)		T	&T	&mut T
共享所有权 (内部可见性)	单线程	Rc<T>	&Rc<T>	无法得到
	单线程	Rc<RefCell<T>>	v.borrow()	v.borrow_mut()
共享所有权 (内部可见性)	多线程	Arc<T>	&Arc<T>	无法得到
	多线程	Arc<Mutex<T>>	v.lock()	v.lock ()
	多线程	Arc<RwLock<T>>	v.read()	v.borrow()

Rc 比 Arc性能效率高， 外部可见性应对编译期检查；内部可见性应对运行时检查， 检查不过会panic

聊聊rust所有权系统

所有权 (ownership) 系统是 Rust 最与众不同的特性，让Rust实现了既要保障内存安全又要无GC，运行时高性能的目标。

所有的编程语言都需要考虑内存管理机制。

一些语言提供垃圾回收机制（garbage collection,GC），例如：Java、Python、Golang、Lisp、Haskell、JavaScript等等。另一些语言需要编程人员手工管理内存，进行分配和释放内存。例如：C、C++。

两种机制各有优势和缺点：

带GC的编程语言，自动管理内存，消除人工管理带来的内存管理安全性问题，降低编程语言学习复杂度和使用复杂度，但是带来了额外的运行时性能开销，无法保证高性能和高实时性。

手工管理内存，运行时高性能，但是增加编程人员的使用心智负担，容易造成内存管理安全性上的bug。

Rust选择了第三种方式，通过（自己创立的）所有权系统进行内存管理。这也是为什么最近Rust编程语言从一出现就备受瞩目的原因之一。

关于所有权系统，编译器会在**编译期**依据所有权规则对代码进行检查。不会给运行期带来额外的开销。（缺点是编译期阶段的时间变的很长，编译后的代码体积会膨胀（就所有权而言，实际上跟范型化的生命周期标记的实现相关））。

让我们来看看所有权规则：

Rust 所有权规则 (Rust ownership rules)

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

规则1：Rust 中的每一个值都有一个被称为其 所有者（owner）的变量。

规则2：值在任一时刻有且只有一个所有者。

规则3：当所有者（变量）离开作用域，这个值将被丢弃。

内存管理

Rust所有权规则1中的值，要么分配在栈上，要么分配在堆上。栈上的数据会随着入栈出栈操作被自动清理，编程语言主要是对堆进行内存管理。

规则1与规则2

规则1 理解起来比较简单直观，在 Rust 中通过 `let` 关键字将一个值绑定到一个变量上。

```
fn main (){  
    let a = 12;  
    let b = "hello";  
  
    let (c,d) = ("str1","str2");  
  
    println!("{};{};{};{}",a ,b ,c ,d)  
}
```

move语义

如果将一个变量赋值给另外一个变量会如何？考虑下面这种情况：

```
fn main (){  
    let a = String::from("hello world");  
    let b = a;  
  
    println!("{}",a);  
    println!("{}",b);  
}
```

编译会报错：

```
error[E0382]: borrow of moved value: `a`
--> src/main.rs:5:18
|
2 |     let a = String::from("hello world");
|         - move occurs because `a` has type `String`, which does not implement the `Copy` trait
3 |     let b = a;
|         - value moved here
4 |
5 |     println!("{}", a);
|         ^ value borrowed here after move
```

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.

直接错误提示是： error[E0382]: borrow of moved value: `a` 借用已经move的值a。

同时在 let b = a; 那行编译器提示 value moved here , 在此处值被move了。

我们对 let 关键字的理解更加深入：对于形如 let x = y; 的语句，如果 y 是变量， let 会引发变量的所有权转移。

也即是：let操作默认是move语义。从而保障了规则2的约束。

Copy trait

新的问题出现，下面这段代码没有遵循 move 语义，但编译成功：

```
fn main (){
    let a = 128;
    let b = a;// move ?
    println!("{}", a);//128
    println!("{}", b);//128
}
```

那是因为考虑到使用上的便利性，在有些场景下并不希望应用默认的 move 语义。Rust的基础数据类型都实现了 std::marker::Copy trait ，所以在进行 let 操作时，实际上发生了 copy 。

哪些类型是满足 `Copy` 特性的？类似整型这样的存储在栈上的类型、不需要分配内存或某种形式资源的类型。常见的 `Copy` 类型如下：

- 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。

Clone trait

保留原来的变量，并且将值内容拷贝给新的变量，对简单的数字类型很容易理解接受，那如果是复杂类型或者说是自定义类型呢？如果不希望发生所有权转移，可以使用 `clone`。如下：

```
fn main (){
    let a = String::from("hello world");
    let b = a.clone();

    println!("{}", a);
    println!("{}", b);
}
```

`String` 实现了 `std::clone::Clone trait`。使用 `clone` 操作后两个变量持有不同的值（这两个值是不同的内存空间）。能不能像使用基础数据类型那样方便地应用 `copy` 语义呢？可以的，如下：

```
#[derive(Debug,Copy,Clone)] // here!
struct Foo {
    x: i32,
    y: i64,
}

pub fn main() {
    let f1 = Foo { x: 1, y: 2 };
    let f2: Foo = f1;
    println!("p1 = {:?}", f1);
    println!("p1 = {:?}", f2);
}
```

当然这里的例子有一点点特殊，`Foo` 结构体的成员都是基础类型，所以我们标记注解 `#[derive(Debug,Copy,Clone)]` 即可满足。如果其成员是复杂的类型，就需要实现 `Copy` 和 `Clone` 的 `trait`。

注意：`Clone trait` 是 `Copy trait` 的 `supertrait`，所以任何一个实现 `Copy` 的类型必须实现 `Clone`。

所有权与函数

调用函数的时候，会将实际参数传递给形式参数，这个操作在语义上与给变量赋值相似。可能会移动或者复制，遵循与赋值语句一样的规则。例如：

```
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
```

这里没有特殊情况，很好。

引用与借用

再看如下的代码：

```
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)
}
```

在每一次函数调用的时候，都获取所有权，然后在函数调用完成的时候，再返回所有权，这种方式略显繁琐。而这种调用在实际场景中极其常见，Rust提供了引用（references）功能来解决这个问题。

利用引用，上面的代码可以改写成：

```
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()
}
```

传给 `calculate_length()` 的是 `&s1` 而不是 `s1`。函数的声明是 `calculate_length(s: &String)` 参数是 `s: &String`。

&就是引用。它允许你使用值但不获取其所有权。毕竟我们需要遵循规则2。

我们把采用引用作为函数参数称为借用（borrowing）

这里并不想展开讨论不可变引用和可变引用的相关规则，那是并发编程场景下需要关注的点。

关于引用还需要特别注意的一个点是作用域。一个引用的作用域从声明的地方开始一直持续到最后一次使用为止。如下代码是可以编译的：

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

```
let r1 = &mut s; // 没问题
println!("{}", r1); // r1最后使用的地方

let r2 = &mut s; // 没问题
println!("{}", r2);
}
```

编译器负责推断引用最后一次使用的代码位置。

slice

对集合一段连续元素的引用就是[slice类型]。 例如：

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

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

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

实际上，字符串字面量就是 slice 。

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

这里的 s 的类型是 &str ， 它是一个指向二进制程序特定位置的 slice 。 &str 是一个不可变引用。

规则3

当所有者（变量）离开作用域，这个值将被丢弃。

如何在合适的时机清理回收内存？当持有堆内存数据的变量离开作用域的时候，堆内存的数据将被 drop 掉。

编译器会自动插入 drop 相关的代码，在运行时需要 drop 的时候调用。

这条规则的重点就是作用域了。

作用域

作用域是一个项 (item) 在程序中有效的范围。最常见的就是**函数作用域**，比如

```
fn main(){// s无效  
    let s = "hello world";// s进入作用域  
    println!("{}",s);  
}// s离开作用域
```

还有块作用域，例如：

```
fn main(){// s无效  
    let s = "hello world";// s进入作用域  
    println!("{}",s);  
    {  
        let a = 1; //a 进入作用域  
        println!("{}",a);  
    }// a离开作用域  
  
    //println!("{}",a); 这里会出错  
}// s离开作用域
```

实际上，`let` 关键字会隐式地开启一个作用域，对于以下代码：

```
fn main(){  
    let s1 = "hello";  
    let s2 ="world";  
}
```

利用 <https://play.rust-lang.org/> 的 SHOW MIR 功能可以看到：

```

fn main() -> () {
    let mut _0: (); // return place in scope 0 at src/main.rs:1:10: 1:10
    let _1: &str; // in scope 0 at src/main.rs:3:9: 3:11
    scope 1 {
        debug s1 => _1; // in scope 1 at src/main.rs:3:9: 3:11
        let _2: &str; // in scope 1 at src/main.rs:4:9: 4:11
        scope 2 {
            debug s2 => _2; // in scope 2 at src/main.rs:4:9: 4:11
        }
    }
}

```

...

main 函数的作用域 scope 0，在其中，let s1 = "hello"; 创建一个作用域 scope 1，然后 let s2 ="world"; 又在 scope 1 里面创建了 scope 2。

关于引用的特殊作用域问题，上一小节已经说明，这里不再重复。

闭包带来的问题

闭包 (closures) 可以从环境捕获变量，并在闭包体中使用。有三种使用变量的方式：获取所有权、可变引用、不可变引用。Rust 提供了3种 Fn trait 以便编译器能够更清晰直接地处理不同场景下闭包对变量所有权的操作问题。

- FnOnce 获取变量所有权，Once表明闭包不能多次获取同一个变量的所有权，所以只能调用一次。
- FnMut 获取变量可变的借用值
- Fn 获取变量的不可变的借用值

注意：FnOnce 是 FnMut Fn 的 supertrait；FnMut 是 Fn 的 supertrait。FnOnce 的例子：

```

fn do_action<F>(func:F)where F:FnOnce()->usize{
    println!("disappeared variable length : {}", func());
}

```

```

pub fn main(){
    let a = String::from("hello world");
    let useless_warpper = move ||->usize{a.len()} ;
    do_action(useless_warpper);

    //println!("{}",a); //这里再使用a会报错
}

```

使用了一个关键字 `move` 将闭包需要捕获的变量的所有权 `move` 到闭包内。`FnMut` 的例子：

```
fn do_action<F>(mut func:F)where F:FnMut()=>usize{
    println!("mutable result {}", func());
}

pub fn main(){
    let mut a = 12;
    let mut mutable_warpper = ||=>usize{ a+=2;a } ;
    do_action(mutable_warpper);

    println!("now {}",a);
}
```

`Fn` 的例子（将闭包形式简化）：

```
fn do_action<F>(func:F)where F:Fn()=>usize{
    println!("result {}", func());
}

pub fn main(){
    let a = 12;
    let square = || a*a ;
    do_action(square);

    println!("now {}",a);
}
```

引用的生命周期(lifetime)标记

大部分场景下，我们使用变量的时候，编译器能够自动推断变量的作用域并正常工作。但有时候不那么明显，特别是使用引用的很多场景下，需要手工标记变量的生命周期，帮助编译器检查引用的生命周期不会超过对象的生命周期。例如：

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

编译失败：

```
error[E0597]: `x` does not live long enough
--> src/main.rs:5:13
|
5|     r = &x;
|     ^^^ borrowed value does not live long enough
6| }
| - `x` dropped here while still borrowed
7| println!("r: {}", r);
|           - borrow later used here
```

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.

这里提示说 `'x'` does not live long enough。`r` 引用了一个存活不够久的 `x`。`r` 的生命周期比 `x` 的生命周期长。

Rust 通过借用检查器 (borrow checker) 比较作用域来确保所有的引用都是有效的。

继续看下面的代码：

```
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
    }
}
```

编译错误：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:9:33
|
9| fn longest(x: &str, y: &str) -> &str {
|         ---- ---- ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from
```

```
'x` or `y`
help: consider introducing a named lifetime parameter
|
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
|     ^^^^  ^^^^^^  ^^^^^^  ^^^
error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
```

错误提示说，函数返回值包含借用值，但是签名未说明是借用了 `x` 还是 `y`。同时给出了建议将函数签名改写成：

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

实际上我们无法在编译期确定该函数到底返回 `x` 还是 `y`，也无法知道传入参数（引用）的生命周期，这就导致编译器无法判定返回的引用是否有效。这就需要按照提示改写函数签名。

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

现在函数签名表明对于生命周期 `'a`，函数会获取两个参数，他们都是与生命周期 `'a` 存在的一样长的字符串 slice。函数会返回一个同样也与生命周期 `'a` 存在的一样长的字符串 slice。这样Rust 的借用检查器就可以在编译期检查传给该函数的参数是否合法。

当具体的引用被传递给 `longest` 时，被 `'a` 所替代的具体生命周期是 `x` 的作用域与 `y` 的作用域相重叠的那一部分。换一种说法就是泛型生命周期 `'a` 的具体生命周期等同于 `x` 和 `y` 的生命周期中较小的那个。因为我们用相同的生命周期参数 `'a` 标注了返回的引用值，所以返回的引用值就能保证在 `x` 和 `y` 中较短的那个生命周期结束之前保持有效。

如果结构体中使用引用，同样也需要手工标注生命周期。如下代码：

```
#[derive(Debug)]
struct Foo{
    str: &str,
    a : i32,
}

pub fn main(){
    let x = Foo{str: "hello",a: 1};
```

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

同样会编译错误：

```
error[E0106]: missing lifetime specifier
--> src/main.rs:3:10
|
3 |     str: &str,
|         ^ expected named lifetime parameter
|
help: consider introducing a named lifetime parameter
|
2 | struct Foo<'a>{
3 |     str: &'a str,
|
error: aborting due to previous error
```

For more information about this error, try `rustc --explain E0106`.

改写成带标记生命周期的形式：

```
#[derive(Debug)]
struct Foo<'a>{
    str: &'a str,
    a : i32,
}

pub fn main(){
    let x = Foo{str: "hello",a: 1};
    println!("{:?}",x);
}
```

保证编译器可以检查对比 `x` 结构体的生命周期与其成员 `str` 的生命周期，前者比后者小的时候，编译通过。

静态生命周期 `'static`，其生命周期能够存活于整个程序期间。所有的字符串字面量值都是 `'static` 的。

好了就这些。

参考：<https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html>

01 | 所有权：值的生杀大权到底在谁手上？

其实所有权和生命周期之所以这么难学明白，除了其与众不同的解决内存安全问题的角度外，另一个很大的原因是，目前的资料对初学者都不友好，上来就讲 Copy / Move 语义怎么用，而没有讲明白为什么要这样用。

所以这一讲我们换个思路，从一个变量使用堆栈的行为开始，探究 Rust 设计所有权和生命周期的用意，帮你从根上解决这些编译问题。

变量在函数调用时发生了什么

首先，我们来看一看，在我们熟悉的大多数编程语言中，变量在函数调用时究竟会发生什么、存在什么问题。

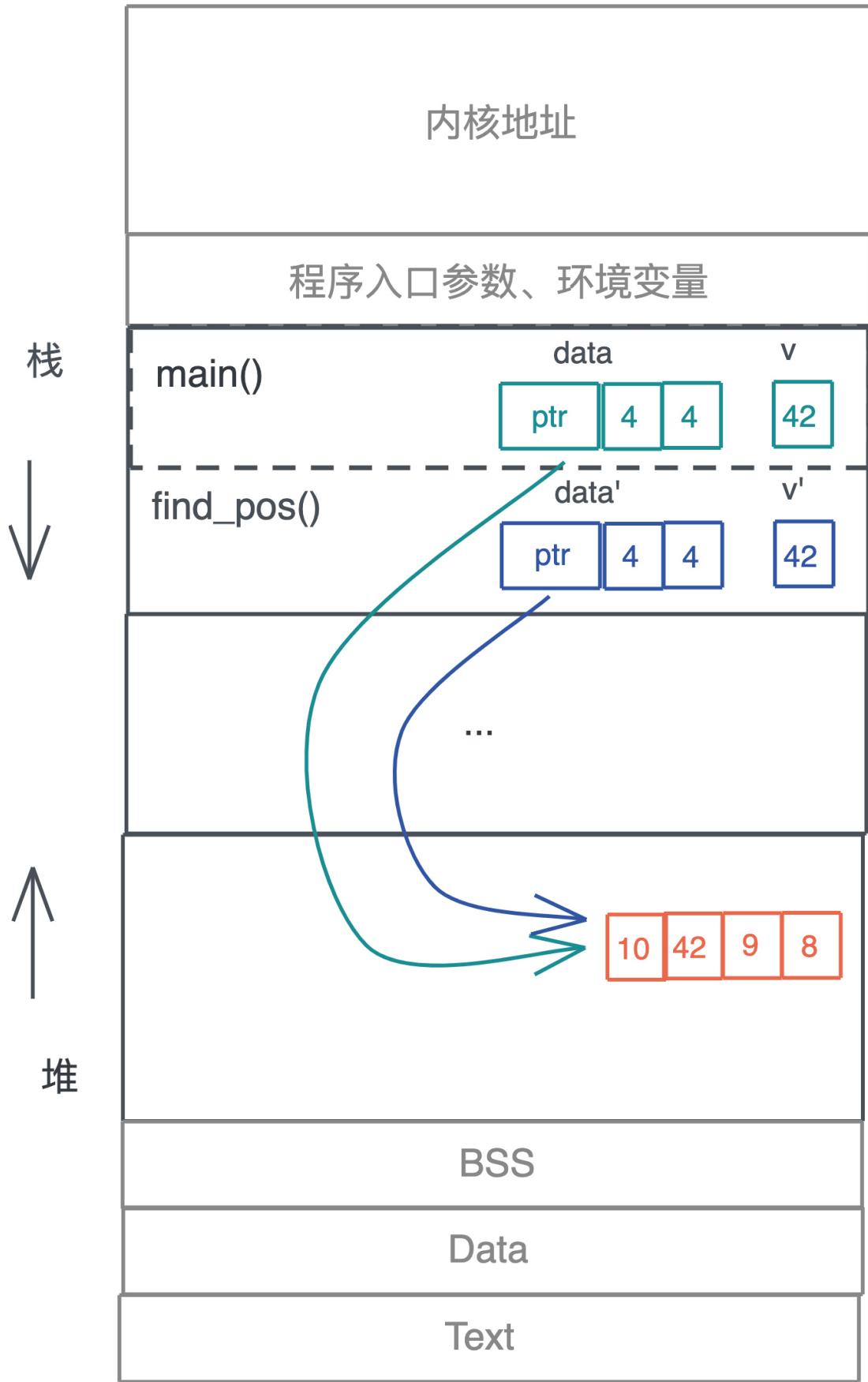
看这段代码，main() 函数中定义了一个动态数组 data 和一个值 v，然后将其传递给函数 find_pos，在 data 中查找 v 是否存在，存在则返回 v 在 data 中的下标，不存在返回 None ([代码1](#))：

```
fn main() {
    let data = vec![10, 42, 9, 8];
    let v = 42;
    if let Some(pos) = find_pos(data, v) {
        println!("Found {} at {}", v, pos);
    }
}

fn find_pos(data: Vec<u32>, v: u32) -> Option<usize> {
    for (pos, item) in data.iter().enumerate() {
        if *item == v {
            return Some(pos);
        }
    }
    None
}
```

这段代码不难理解，要再强调一下的是，**动态数组因为大小在编译期无法确定，所以放在堆上，并且在栈上有一个包含了长度和容量的胖指针指向堆上的内存。**

在调用 find_pos() 时，main() 函数中的局部变量 data 和 v 作为参数传递给了 find_pos()，所以它们会被放在 find_pos() 的参数区。



按照大多数编程语言的做法，现在堆上的内存就有了两个引用。不光如此，我们每把 `data` 作为参数传递一次，堆上的内存就会多一次引用。

但是，这些引用究竟会做什么操作，我们不得而知，也无从限制；而且堆上的内存究竟什么时候能释放，尤其在多个调用栈引用时，很难厘清，取决于最后一个引用什么时候结束。所以，这样一个看似简单的函数调用，给内存管理带来了极大麻烦。

对于堆内存多次引用的问题，我们先来看大多数语言的方案：

- **C/C++ 要求开发者手工处理**，非常不便。这需要我们在写代码时高度自律，按照前人总结的最佳实践来操作。但人必然会犯错，一个不慎就会导致内存安全问题，要么内存泄露，要么使用已释放内存，导致程序崩溃。
- **Java 等语言使用追踪式 GC**，通过定期扫描堆上数据还有没有人引用，来替开发者管理堆内存，不失为一种解决之道，但 GC 带来的 STW 问题让语言的使用场景受限，性能损耗也不小。
- **ObjC/Swift 使用自动引用计数（ARC）**，在编译时自动添加维护引用计数的代码，减轻开发者维护堆内存的负担。但同样地，它也会有不小的运行时性能损耗。

现存方案都是从管理引用的角度思考的，有各自的弊端。我们回顾刚才梳理的函数调用过程，从源头上看，本质问题是堆上内存会被随意引用，那么换个角度，我们是不是可以限制引用行为本身呢？

Rust 的解决思路

这个想法打开了新的大门，Rust就是这样另辟蹊径的。

在 Rust 以前，引用是一种随意的、可以隐式产生的、对权限没有界定的行为，比如 C 里到处乱飞的指针、Java 中随处可见的按引用传参，它们可读可写，权限极大。而 Rust 决定限制开发者随意引用的行为。

其实作为开发者，我们在工作中常常能体会到：**恰到好处的限制，反而会释放无穷的创意和生产力**。最典型的就是各种开发框架，比如 React、Ruby on Rails 等，他们限制了开发者使用语言的行为，却极大地提升了生产力。

好，思路我们已经有了，具体怎么实现来限制数据的引用行为呢？

要回答这个问题，我们需要先来回答：谁真正拥有数据或者说值的生杀大权，这种权利可以共享还是需要独占？

所有权和 Move 语义

照旧我们先尝试回答一下，对于值的生杀大权可以共享还是需要独占这一问题，我们大概都会觉得，一个值最好只有一个拥有者，因为所有权共享，势必会带来使用和释放上的不明确，走回追踪式 GC 或者 ARC 的老路。

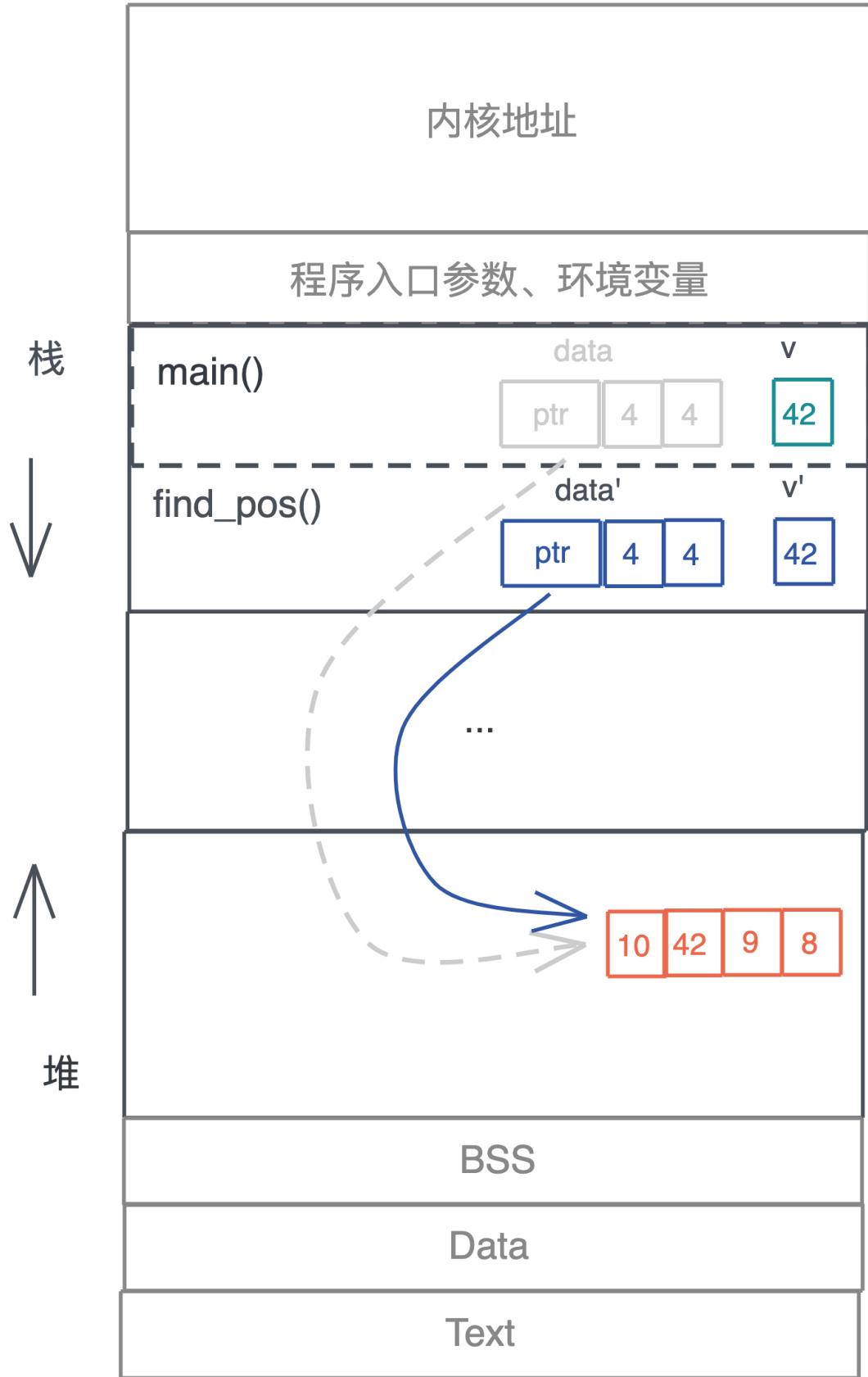
那么如何保证独占呢？具体实现其实是有些困难的，因为太多情况需要考虑。比如说一个变量被赋给另一个变量、作为参数传给另一个函数，或者作为返回值从函数返回，都可能造成这个变量的拥有者不唯一。怎么办？

对此，Rust 给出了如下规则：

- **一个值只能被一个变量所拥有，这个变量被称为所有者** (Each value in Rust has a variable that's called its owner) 。
- **一个值同一时刻只能有一个所有者** (There can only be one owner at a time) ，也就是说不能有两个变量拥有相同的值。所以对应刚才说的变量赋值、参数传递、函数返回等行为，旧的所有者会把值的所有权转移给新的所有者，以便保证单一所有者的约束。
- **当所有者离开作用域，其拥有的值被丢弃** (When the owner goes out of scope, the value will be dropped) ，内存得到释放。

这三条规则很好理解，核心就是保证单一所有权。其中第二条规则讲的所有权转移是 Move 语义，Rust 从 C++ 那里学习和借鉴了这个概念。

在这三条所有权规则的约束下，我们看开头的引用问题是如何解决的：



原先 main() 函数中的 data，被移动到 find_pos() 后，就失效了，编译器会保证 main() 函数随后的代码无法访问这个变量，这样，就确保了堆上的内存依旧只有唯一的引用。

看这个图，你可能会有一个小小的疑问：main() 函数传递给 find_pos() 函数的另一个参数 v，也会被移动吧？为什么图上并没有标灰？咱们暂且将这个疑问放到一边，等这一讲学完，相信你会有答案的。

现在，我们来写段代码加深一下对所有权的理解。

在这段代码里，先创建了一个不可变数据 data，然后将 data 赋值给 data1。按照所有权的规则，赋值之后，data 指向的值被移动给了 data1，它自己便不可访问了。而随后，data1 作为参数被传给函数 sum()，在 main() 函数下，data1 也不可访问了。

但是后续的代码依旧试图访问 data1 和 data，所以，这段代码应该会有两处错误（[代码2](#)）：

```
fn main() {
    let data = vec![1, 2, 3, 4];
    let data1 = data;
    println!("sum of data1: {}", sum(data1));
    println!("data1: {:?}", data1); // error1
    println!("sum of data: {}", sum(data)); // error2
}

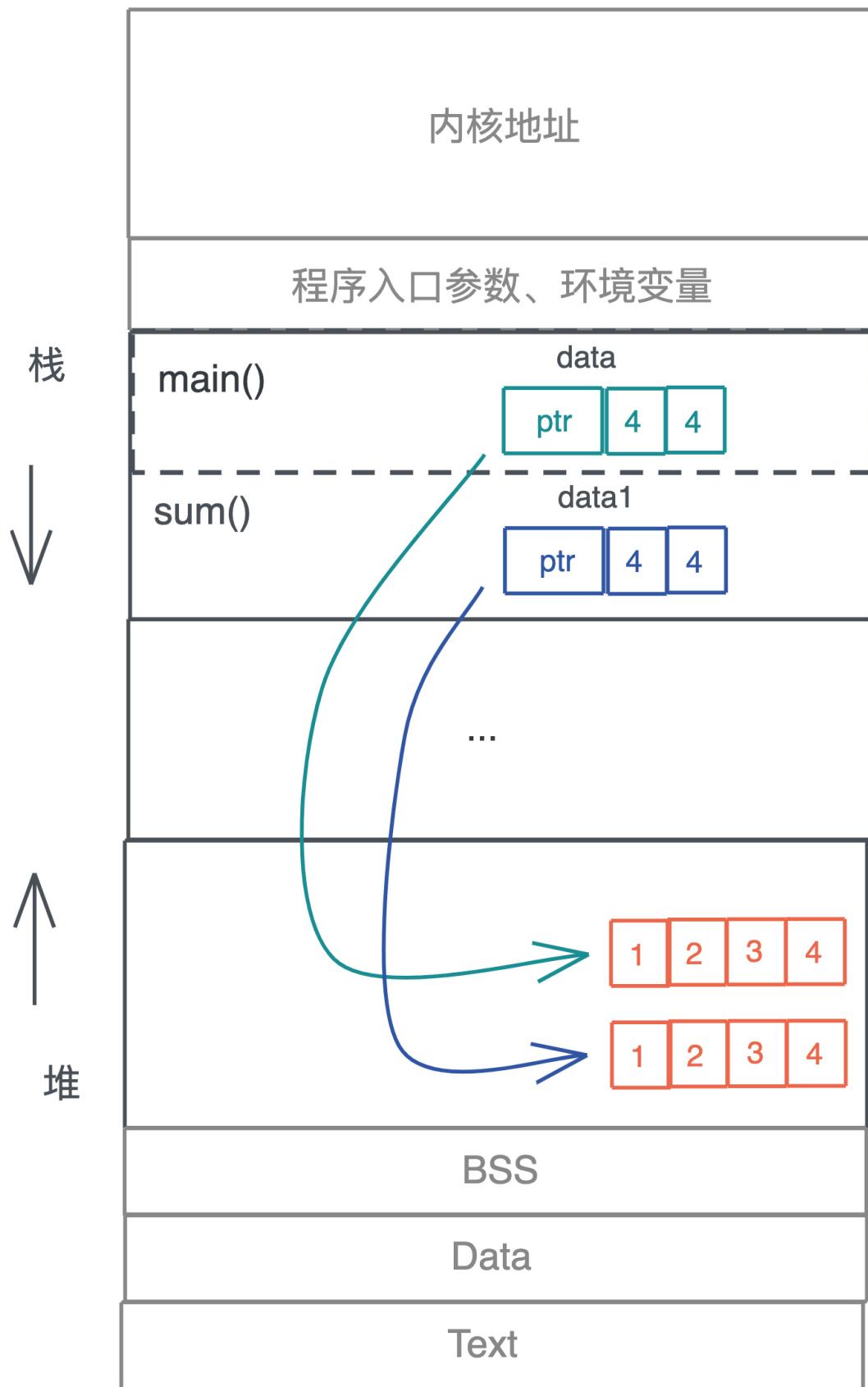
fn sum(data: Vec<u32>) -> u32 {
    data.iter().fold(0, |acc, x| acc + x)
}
```

运行时，编译器也确实捕获到了这两个错误，并清楚地告诉我们不能使用已经移动过的变量：

```
> cargo run --bin ownership_error
Compiling ownership v0.1.0 (/Users/tchen/projects/mycode/rust/geek-time-rust-resources/04/ownership)
error[E0382]: borrow of moved value: `data1`
--> src/ownership_error.rs:5:29
3 |     let data1 = data;
|         ---- move occurs because `data1` has type `Vec<u32>`, which does not implement the `Copy` trait
4 |     println!("sum of data1: {}", sum(data1));
|                     ---- value moved here
5 |     println!("data1: {:?}", data1);
|             ^^^^^ value borrowed here after move
error[E0382]: use of moved value: `data`
--> src/ownership_error.rs:6:37
2 |     let data = vec![1, 2, 3, 4];
|         ---- move occurs because `data` has type `Vec<u32>`, which does not implement the `Copy` trait
3 |     let data1 = data;
|             ---- value moved here
...
6 |     println!("sum of data: {}", sum(data));
|             ^^^ value used here after move
```

如果我们要在把 data1 传给 sum(), 同时, 还想让 main() 能够访问 data, 该怎么办?

我们可以调用 `data.clone()` 把 data 复制一份出来给 data1, 这样, 在堆上就有 `vec![1,2,3,4]` 两个互不影响且可以独立释放的副本, 如下图所示:



可以看到，所有权规则，解决了谁真正拥有数据的生杀大权问题，让堆上数据的多重引用不复存在，这是它最大的优势。

但是，这也会让代码变复杂，尤其是一些只存储在栈上的简单数据，如果要避免所有权转移之后不能访问的情况，我们就需要手动复制，会非常麻烦，效率也不高。

Rust 考虑到了这一点，提供了两种方案：

1. 如果你不希望值的所有权被转移，在 Move 语义外，Rust 提供了 **Copy 语义**。如果一个数据结构实现了 [Copy trait](#)，那么它就会使用 Copy 语义。这样，在你赋值或者传参时，值会自动按位拷贝（浅拷贝）。
2. 如果你不希望值的所有权被转移，又无法使用 Copy 语义，那你可以“借用”数据，我们下一讲会详细讨论“借用”。

我们先看今天要讲的第一种方案：Copy 语义。

Copy 语义和 Copy trait

符合 Copy 语义的类型，在你赋值或者传参时，值会自动按位拷贝。这句话不难理解，那在Rust中是具体怎么实现的呢？

我们再仔细看看刚才代码编译器给出的错误，你会发现，它抱怨 data 的类型 `Vec<u32>` 没有实现 Copy trait，在赋值或者函数调用的时候无法 Copy，于是就按默认使用 Move 语义。而 Move 之后，原先的变量 data 无法访问，所以出错。

```
error[E0382]: use of moved value: `data`
--> src/ownership_error.rs:6:37
2 |     let data = vec![1, 2, 3, 4];
|         ---- move occurs because `data` has type `Vec<u32>`, which does not implement the `Copy` trait
3 |     let data1 = data;
|             ---- value moved here
...
6 |     println!("sum of data: {}", sum(data));
|             ^^^^^ value used here after move
```

换句话说，当你要移动一个值，如果值的类型实现了 **Copy trait**，就会自动使用 Copy 语义进行拷贝，否则使用 Move 语义进行移动。

讲到这里，我插一句，在学习 Rust 的时候，你可以根据编译器详细的错误说明来尝试修改代码，使编译通过，在这个过程中，你可以用 Stack Overflow 搜索错误信息，进一步学习自己不了解的知识点。我也非常建议你根据上图中的错误代码 E0382 使用 `rustc --explain E0382` 探索更详细的信息。

好，回归正文，那在 Rust 中，什么数据结构实现了 Copy trait 呢？你可以通过下面的代码快速验证一个数据结构是否实现了 Copy trait ([验证代码](#))：

```
fn is_copy<T: Copy>() {}

fn types_impl_copy_trait() {
    is_copy::<bool>();
    is_copy::<char>();

    // all iXX and uXX, usize/isize, fXX implement Copy trait
    is_copy::<i8>();
    is_copy::<u64>();
    is_copy::<i64>();
    is_copy::<usize>();

    // function (actually a pointer) is Copy
    is_copy::<fn()>();

    // raw pointer is Copy
    is_copy::<*>const String();
    is_copy::<*>mut String();

    // immutable reference is Copy
    is_copy::<&[Vec<u8>]>();
    is_copy::<&String>();

    // array/tuple with values which is Copy is Copy
    is_copy::<[u8; 4]>();
    is_copy::<(&str, &str)>();
}

fn types_not_impl_copy_trait() {
    // unsized or dynamic sized type is not Copy
    is_copy::<str>();
    is_copy::<[u8]>();
    is_copy::<Vec<u8>>();
    is_copy::<String>();

    // mutable reference is not Copy
    is_copy::<&mut String>();

    // array / tuple with values that not Copy is not Copy
    is_copy::<[Vec<u8>; 4]>();
    is_copy::<(String, u32)>();
}

fn main() {
    types_impl_copy_trait();
    types_not_impl_copy_trait();
}
```

推荐你动手运行这段代码，并仔细阅读编译器错误，加深印象。我也总结一下：

- 原生类型，包括函数、不可变引用和裸指针实现了 Copy；
- 数组和元组，如果其内部的数据结构实现了 Copy，那么它们也实现了 Copy；
- 可变引用没有实现 Copy；
- 非固定大小的数据结构，没有实现 Copy。

另外，[官方文档介绍 Copy trait 的页面](#)包含了 Rust 标准库中实现 Copy trait 的所有数据结构。你也可以在访问某个数据结构的时候，查看其文档的 Trait implementation 部分，看看它是否实现了 Copy trait。

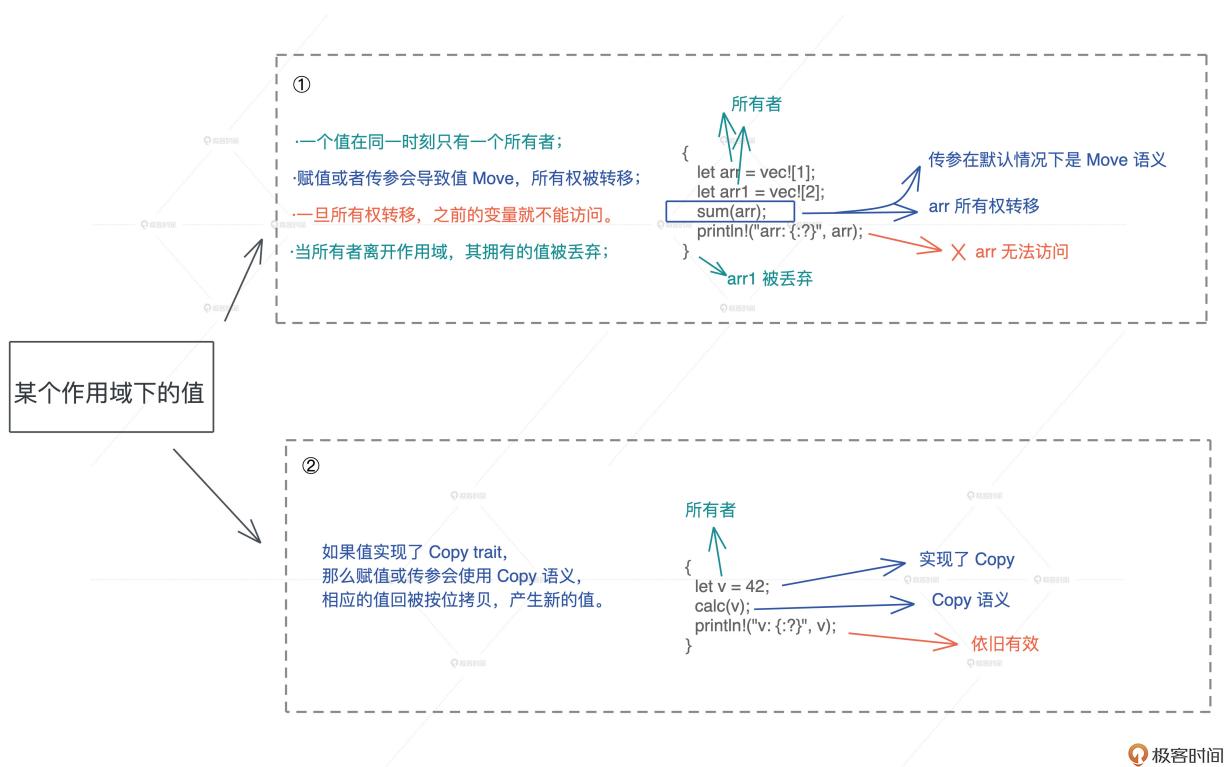
The screenshot shows the Rust documentation for the primitive type `i64`. On the left, a sidebar lists various traits implemented by `i64`, with `Copy` highlighted. The main content area shows the `Primitive Type i64` documentation, which includes sections for `Implementations`, `Examples`, and `Examples` for both `MIN` and `MAX` constants. A "Run" button is present next to each example code snippet.

小结

今天我们学习了 Rust 的单一所有权模式、Move 语义、Copy 语义，我整理一下关键信息，方便你再回顾一遍。

- 所有权：一个值只能被一个变量所拥有，且同一时刻只能有一个所有者，当所有者离开作用域，其拥有的值被丢弃，内存得到释放。
- Move 语义：赋值或者传参会导致值 Move，所有权被转移，一旦所有权转移，之前的变量就不能访问。

- Copy 语义：如果值实现了 Copy trait，那么赋值或传参会使用 Copy 语义，相应的值会被按位拷贝（浅拷贝），产生新的值。



通过单一所有权模式，Rust 解决了堆内存过于灵活、不容易安全高效地释放的问题，不过所有权模型也引入了很多新的概念，比如今天讲的 Move / Copy 语义。

由于是全新的概念，我们学习起来有一定的难度，但是你只要抓住了核心点：Rust 通过单一所有权来限制任意引用的行为，就不难理解这些新概念背后的设计意义。

下一讲我们会继续学习Rust的所有权和生命周期，在不希望值的所有权被转移，又无法使用 Copy 语义的情况下，如何“借用”数据……

参考资料

trait 是 Rust 用于定义数据结构行为的接口。如果一个数据结构实现了 Copy trait，那么它在赋值、函数调用以及函数返回时会执行 Copy 语义，值会被按位拷贝一份（浅拷贝），而非移动。你可以看关于 [Copy trait](#) 的资料。

02 | 所有权：值的借用是如何工作的？

虽然，单一所有权解决了其它语言中值被任意共享带来的问题，但也引发了一些不便。我们上一讲提到：**当你不希望值的所有权被转移，又因为没有实现 Copy trait 而无法使用 Copy 语义，怎么办？** 你可以“借用”数据，也就是这一讲我们要继续介绍的 [Borrow 语义](#)。

Borrow 语义

顾名思义，Borrow 语义允许一个值的所有权，在不发生转移的情况下，被其它上下文使用。就好像住酒店或者租房那样，旅客/租客只有房间的临时使用权，但没有它的所有权。另外，Borrow 语义通过引用语法（& 或者 &mut）来实现。

看到这里，你是不是有点迷惑了，怎么引入了一个“借用”的新概念，但是又写“引用”语法呢？

其实，在 Rust 中，“借用”和“引用”是一个概念，只不过在其他语言中引用的意义和 Rust 不同，所以 Rust 提出了新概念“借用”，便于区分。

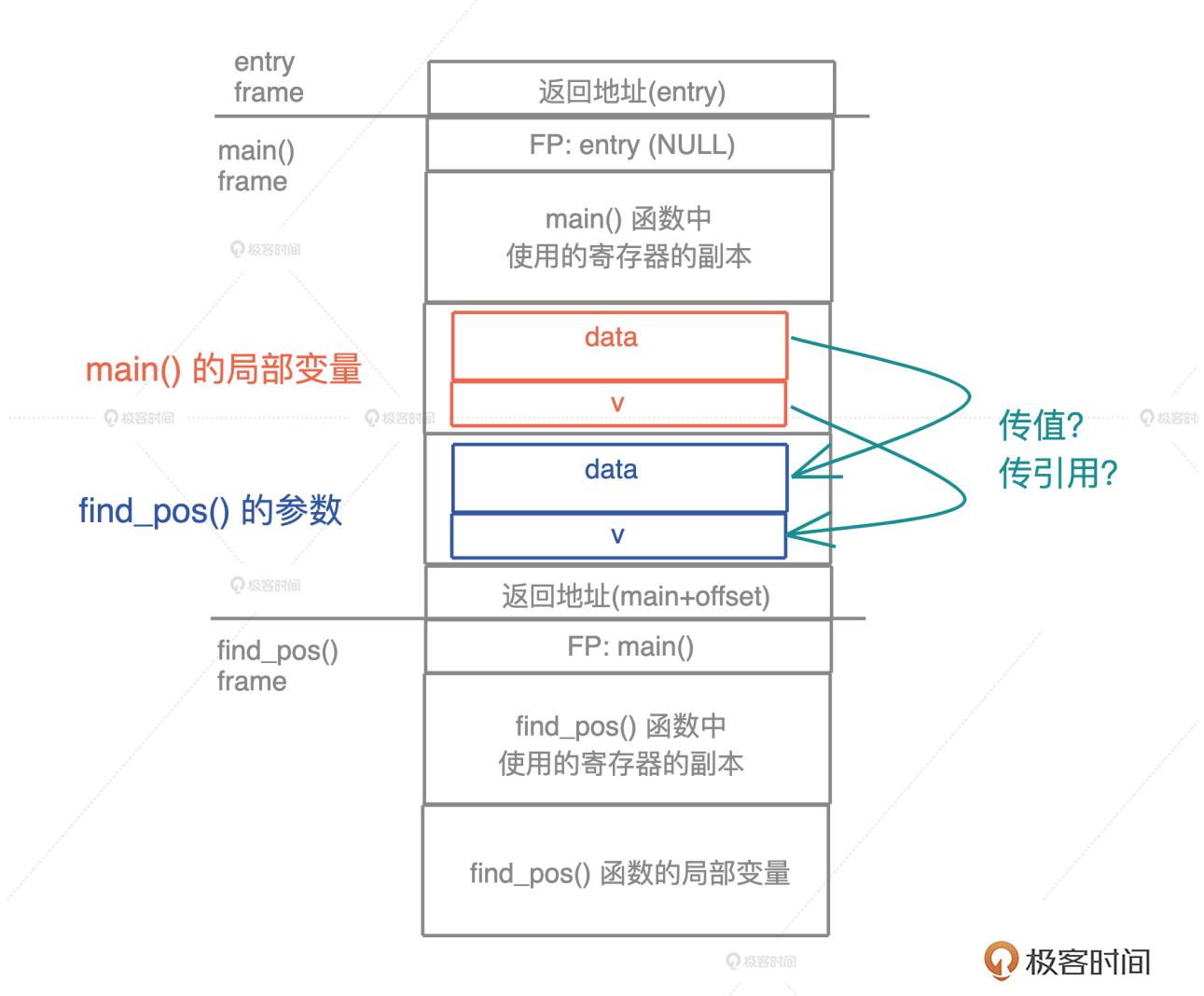
在其他语言中，引用是一种别名，你可以简单理解成鲁迅之于周树人，多个引用拥有对值的无差别的访问权限，本质上是共享了所有权；而在 Rust 下，所有的引用都只是借用了“临时使用权”，它并不破坏值的单一所有权约束。

因此默认情况下，Rust 的借用都是只读的，就好像住酒店，退房时要完好无损。但有些情况下，我们也需要可变的借用，就像租房，可以对房屋进行必要的装饰，这一点待会详细讲。

所以，如果我们想避免 Copy 或者 Move，可以使用借用，或者说引用。

只读借用/引用

本质上，引用是一个受控的指针，指向某个特定的类型。在学习其他语言的时候，你会注意到函数传参有两种方式：传值（pass-by-value）和传引用（pass-by-reference）。



极客时间

以 Java 为例，给函数传一个整数，这是传值，和 Rust 里的 Copy 语义一致；而给函数传一个对象，或者任何堆上的数据结构，Java 都会自动隐式地传引用。刚才说过，Java 的引用是对象的别名，这也导致随着程序的执行，同一块内存的引用到处都是，不得不依赖 GC 进行内存回收。

但 Rust 没有传引用的概念，**Rust 所有的参数传递都是传值**，不管是 Copy 还是 Move。所以在Rust中，你必须显式地把某个数据的引用，传给另一个函数。

Rust 的引用实现了 Copy trait，所以按照 Copy 语义，这个引用会被复制一份交给要调用的函数。对这个函数来说，它并不拥有数据本身，数据只是临时借给它使用，所有权还在原来的拥有者那里。

在 Rust里，引用是一等公民，和其他数据类型地位相等。

还是用上一讲有两处错误的 [代码2](#) 来演示。

```

fn main() {
    let data = vec![1, 2, 3, 4];
    let data1 = data;
    println!("sum of data1: {}", sum(data1));
    println!("data1: {:?}", data1); // error1
    println!("sum of data: {}", sum(data)); // error2
}

fn sum(data: Vec<u32>) -> u32 {
    data.iter().fold(0, |acc, x| acc + x)
}

```

我们把 [代码2](#) 稍微改变一下，通过添加引用，让编译通过，并查看值和引用的地址（[代码3](#)）：

```

fn main() {
    let data = vec![1, 2, 3, 4];
    let data1 = &data;
    // 值的地址是什么？引用的地址又是什么？
    println!(
        "addr of value: {:p}({{:p}}), addr of data {:p}, data1: {:p}",
        &data, data1, &&data, &data1
    );
    println!("sum of data1: {}", sum(data1));

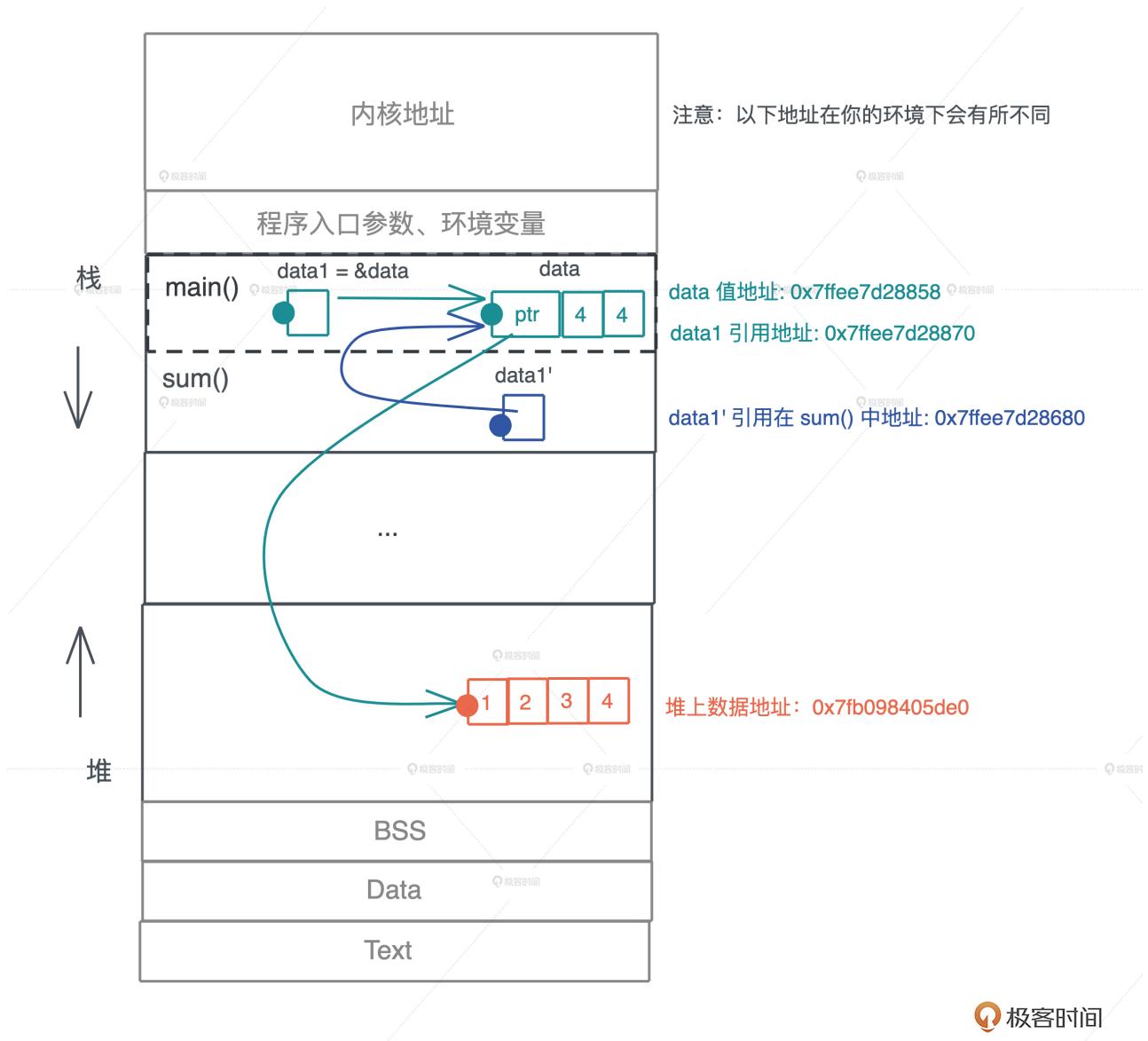
    // 堆上数据的地址是什么？
    println!(
        "addr of items: [{:p}, {:p}, {:p}, {:p}]",
        &data[0], &data[1], &data[2], &data[3]
    );
}

fn sum(data: &Vec<u32>) -> u32 {
    // 值的地址会改变么？引用的地址会改变么？
    println!("addr of value: {:p}, addr of ref: {:p}", data, &data);
    data.iter().fold(0, |acc, x| acc + x)
}

```

在运行这段代码之前，你可以先思考一下，`data` 对应值的地址是否保持不变，而 `data1` 引用的地址，在传给 `sum()` 函数后，是否还指向同一个地址。

好，如果你有想法了，可以再运行代码验证一下你是否正确，我们再看下图分析：



data1、&data 和传到 sum() 里的 data1' 都指向 data 本身，这个值的地址是固定的。但是它们引用的地址都是不同的，这印证了我们讲 Copy trait 的时候，介绍过只读引用实现了 Copy trait，也就意味着引用的赋值、传参都会产生新的浅拷贝。

虽然 data 有很多只读引用指向它，但堆上的数据依旧只有 data 一个所有者，所以值的任意多个引用并不会影响所有权的唯一性。

但我们马上就发现了新问题：一旦 data 离开了作用域被释放，如果还有引用指向 data，岂不是造成我们想极力避免的使用已释放内存（use after free）这样的内存安全问题？怎么办呢？

借用的生命周期及其约束

所以，我们对值的引用也要有约束，这个约束是：借用不能超过（outlive）值的生存期。

这个约束很直观，也很好理解。在上面的代码中，`sum()` 函数处在 `main()` 函数下一层调用栈中，它结束之后 `main()` 函数还会继续执行，所以在 `main()` 函数中定义的 `data` 生命周期要比 `sum()` 中对 `data` 的引用要长，这样不会有任何问题。

但如果是这样的代码呢（[情况1](#)）？

```
fn main() {
    let r = local_ref();
    println!("r: {:?}", r);
}

fn local_ref<'a>() -> &'a i32 {
    let a = 42;
    &a
}
```

显然，生命周期更长的 `main()` 函数变量 `r`，引用了生命周期更短的 `local_ref()` 函数里的局部变量，这违背了有关引用的约束，所以 Rust 不允许这样的代码编译通过。

那么，如果我们在堆内存中，使用栈内存的引用，可以么？

根据过去的开发经验，你也许会脱口而出：不行！因为堆内存的生命周期显然比栈内存要更长更灵活，这样做内存不安全。

我们写段代码试试看，把一个本地变量的引用存入一个可变数组中。从基础知识的学习中我们知道，可变数组存放在堆上，栈上只有一个胖指针指向它，所以这是一个典型的把栈上变量的引用存在堆上的例子（[情况2](#)）：

```
fn main() {
    let mut data: Vec<&u32> = Vec::new();
    let v = 42;
    data.push(&v);
    println!("data: {:?}", data);
}
```

竟然编译通过，怎么回事？我们变换一下，看看还能编译不（[情况3](#)），又无法通过了！

```
fn main() {
    let mut data: Vec<&u32> = Vec::new();
    push_local_ref(&mut data);
```

```

    println!("data: {:?}", data);
}

fn push_local_ref(data: &mut Vec<&u32>) {
    let v = 42;
    data.push(&v);
}

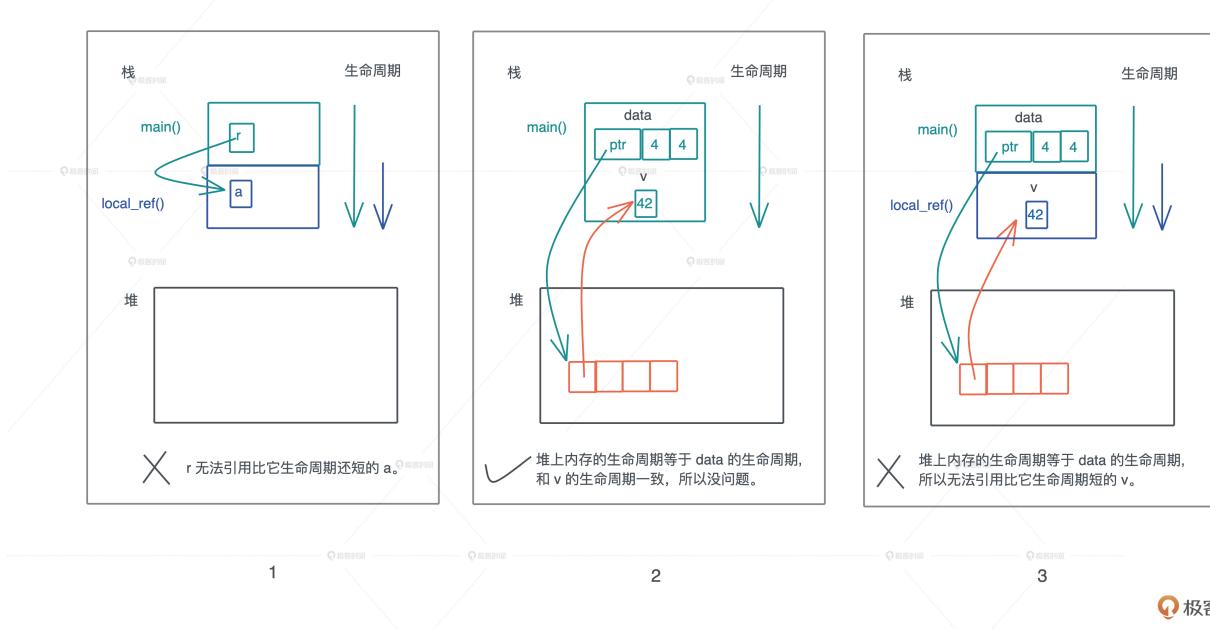
```

到这里，你是不是有点迷糊了，这三种情况，为什么同样是对栈内存的引用，怎么编译结果都不一样？

这三段代码看似错综复杂，但如果抓住了一个核心要素“在一个作用域下，同一时刻，一个值只能有一个所有者”，你会发现，其实很简单。

堆变量的生命周期不具备任意长短的灵活性，因为堆上内存的生死存亡，跟栈上的所有者牢牢绑定。而栈上内存的生命周期，又跟栈的生命周期相关，所以我们核心只需要关心调用栈的生命周期。

现在你是不是可以轻易判断出，为什么情况 1 和情况 3 的代码无法编译通过了，因为它们引用了生命周期更短的值，而情况 2 的代码虽然在堆内存里引用栈内存，但生命周期是相同的，所以没有问题。



好，到这里，默认情况下，Rust 的只读借用就讲完了，借用者不能修改被借用的值，简单类比就像住酒店，只有使用权。

但之前也提到，有些情况下，我们也需要可变借用，想在借用的过程中修改值的内容，就像租房，需要对房屋进行必要的装饰。

可变借用/引用

在没有引入可变借用之前，因为一个值同一时刻只有一个所有者，所以如果要修改这个值，只能通过唯一的拥有者进行。但是，如果允许借用改变值本身，会带来新的问题。

我们先看第一种情况，**多个可变引用共存**：

```
fn main() {
    let mut data = vec![1, 2, 3];

    for item in data.iter_mut() {
        data.push(*item + 1);
    }
}
```

这段代码在遍历可变数组 `data` 的过程中，还往 `data` 里添加新的数据，这是很危险的动作，因为它破坏了循环的不变性（loop invariant），容易导致死循环甚至系统崩溃。所以，在同一个作用域下有多个可变引用，是不安全的。

由于 Rust 编译器阻止了这种情况，上述代码会编译出错。我们可以用 Python 来体验一下多个可变引用可能带来的死循环：

```
if __name__ == "__main__":
    data = [1, 2]
    for item in data:
        data.append(item + 1)
        print(item)
    # unreachable code
    print(data)
```

同一个上下文中多个可变引用是不安全的，那如果同时有一个可变引用和若干个只读引用，会有问题吗？我们再看一段代码：

```
fn main() {
    let mut data = vec![1, 2, 3];
    let data1 = vec![&data[0]];
    println!("data[0]: {:p}", &data[0]);

    for i in 0..100 {
        data.push(i);
    }

    println!("data[0]: {:p}", &data[0]);
```

```
    println!("boxed: {:p}", &data1);
}
```

在这段代码里，不可变数组 `data1` 引用了可变数组 `data` 中的一个元素，这是个只读引用。后续我们往 `data` 中添加了 100 个元素，在调用 `data.push()` 时，我们访问了 `data` 的可变引用。

这段代码中，`data` 的只读引用和可变引用共存，似乎没有什么影响，因为 `data1` 引用的元素并没有任何改动。

如果你仔细推敲，就会发现这里有内存不安全的潜在操作：如果继续添加元素，堆上的数据预留的空间不够了，就会重新分配一片足够大的内存，把之前的值拷过来，然后释放旧的内存。这样就会让 `data1` 中保存的 `&data[0]` 引用失效，导致内存安全问题。

Rust的限制

多个可变引用共存、可变引用和只读引用共存这两种问题，通过 GC 等自动内存管理方案可以避免第二种，但是第一个问题 GC 也无济于事。

所以为了保证内存安全，Rust 对可变引用的使用也做了严格的约束：

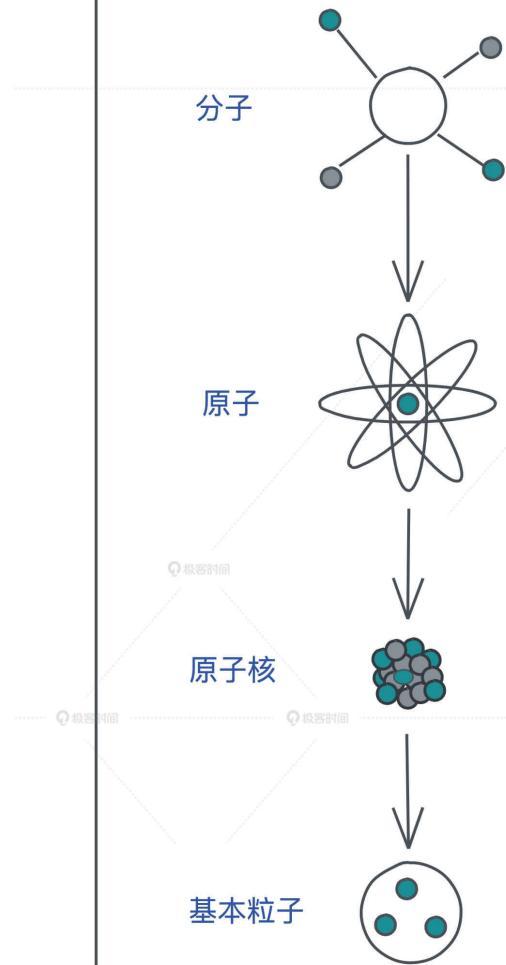
- 在一个作用域内，仅允许一个活跃的可变引用。所谓活跃，就是真正被使用来修改数据的可变引用，如果只是定义了，却没有使用或者当作只读引用使用，不算活跃。
- 在一个作用域内，活跃的可变引用（写）和只读引用（读）是互斥的，不能同时存在。

这个约束你是不是觉得看上去似曾相识？对，它和数据在并发下的读写访问（比如 `RwLock`）规则非常类似，你可以类比学习。

从可变引用的约束我们也可以看到，Rust 不光解决了 GC 可以解决的内存安全问题，还解决了 GC 无法解决的问题。在编写代码的时候，Rust 编译器就像你的良师益友，不断敦促你采用最佳实践来撰写安全的代码。

学完今天的内容，我们再回看[开篇词](#)展示的第一性原理图，你的理解是不是更透彻了？

第一性原理



所有权和生命周期

基本规则

1. 值被唯一的 scope 拥有，它们共存亡；
2. 值可以移动到另一个 scope，新的 scope 拥有这个值；
3. 一个值可以有多个只读引用与单个可变引用，它们之间是互斥关系（RwLock）；
4. 引用不能超越值的存活期。

值在内存中的访问规则

堆或栈中的值的存储方式

回归事物最基础的条件，将其拆分成基本要素解构分析，来探索要解决的问题



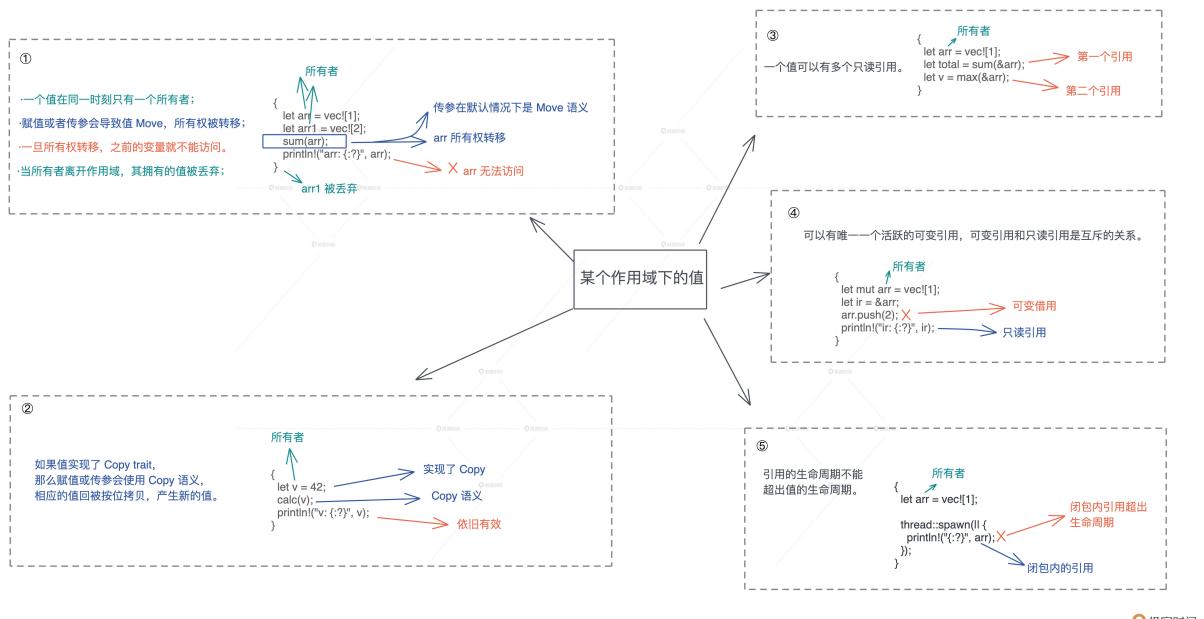
其实，我们拨开表层的众多所有权规则，一层层深究下去，触及最基础的概念，搞清楚堆或栈中值到底是如何存放的、在内存中值是如何访问的，然后从这些概念出发，或者扩展其外延，或者限制其使用，从根本上寻找解决之道，这才是我们处理复杂问题的最佳手段，也是Rust的设计思路。

小结

今天我们学习了 Borrow 语义，搞清楚了只读引用和可变引用的原理，结合上一讲学习的 Move / Copy 语义，Rust 编译器会通过检查，来确保代码没有违背这一系列的规则：

1. 一个值在同一时刻只有一个所有者。当所有者离开作用域，其拥有的值会被丢弃。赋值或者传参会导致值 Move，所有权被转移，一旦所有权转移，之前的变量就不能访问。
2. 如果值实现了 Copy trait，那么赋值或传参会使用 Copy 语义，相应的值会被按位拷贝，产生新的值。
3. 一个值可以有多个只读引用。
4. 一个值可以有唯一一个活跃的可变引用。可变引用（写）和只读引用（读）是互斥的关系，就像并发下数据的读写互斥那样。
5. 引用的生命周期不能超出值的生命周期。

你也可以看这张图快速回顾：



但总有一些特殊情况，比如DAG，我们想绕过“一个值只有一个所有者”的限制，怎么办？下一讲我们继续学习……

思考题

1. 上一讲我们在讲 Copy trait 时说到，可变引用没有实现 Copy trait。结合这一讲的内容，想想为什么？
2. 下面这段代码，如何修改才能使其编译通过，避免同时有只读引用和可变引用？

```
fn main() {
    let mut arr = vec![1, 2, 3];
    // cache the last item
    let last = arr.last();
    arr.push(4);
    // consume previously stored last item
    println!("last: {:?}", last);
}
```

欢迎在留言区分享你的思考。今天你完成了 Rust 学习的第八次打卡！如果你觉得有收获，也欢迎你分享给身边的朋友，邀TA一起讨论。

参考资料

有同学评论,好奇可变引用是如何导致堆内存重新分配的, 我们看一个例子。我先分配一个 capacity 为 1 的 Vec, 然后放入 32 个元素, 此时它会重新分配, 然后打印重新分配前后 &v[0] 的堆地址时, 会看到发生了变化。

所以, 如果我们有指向旧的 &v[0] 的地址, 就会读到已释放内存, 这就是我在文中说为什么在同一个作用域下, 可变引用和只读引用不能共存 ([代码](#)) 。

```
use std::mem;

fn main() {
    // capacity 是 1, len 是 0
    let mut v = vec![1];
    // capacity 是 8, len 是 0
    let v1: Vec<i32> = Vec::with_capacity(8);

    print_vec("v1", v1);

    // 我们先打印 heap 地址, 然后看看添加内容是否会导致堆重分配
    println!("heap start: {:p}", &v[0] as *const i32);

    extend_vec(&mut v);

    // heap 地址改变了! 这就是为什么可变引用和不可变引用不能共存的原因
    println!("new heap start: {:p}", &v[0] as *const i32);

    print_vec("v", v);
}

fn extend_vec(v: &mut Vec<i32>) {
    // Vec<T> 堆内存里 T 的个数是指数增长的, 我们让它恰好 push 33 个元素
    // capacity 会变成 64
    (2..34).into_iter().for_each(|i| v.push(i));
}

fn print_vec<T>(name: &str, data: Vec<T>) {
    let p: [usize; 3] = unsafe { mem::transmute(data) };
}
```

```
// 打印 Vec<T> 的堆地址, capacity, len
println!("{}: 0x{:x}, {}, {}", name, p[0], p[1], p[2]);
}
```

打印结果（地址在你机器上会不一样）：

```
v1: 0x7f8a2f405e00, 8, 0
heap start: 0x7f8a2f405df0
new heap start: 0x7f8a2f405e20
v: 0x7f8a2f405e20, 64, 33
```

如果你运行了这段代码，你可能会注意到一个很有意思的细节：我在 playground 代码链接中给出的代码和文中的代码稍微有些不同。

在文中我的环境是 OS X，很少量的数据就会让堆内存重新分配，而 playground 是 Linux 环境，我一直试到 > 128KB 内存才让 Vec 的堆内存重分配。

03 | 所有权：一个值可以有多个所有者么？

之前介绍的单一所有权规则，能满足我们大部分场景中分配和使用内存的需求，而且在编译时，通过 Rust 借用检查器就能完成静态检查，不会影响运行时效率。

但是，规则总会有例外，在日常工作中有些特殊情况该怎么处理呢？

- 一个有向无环图（DAG）中，某个节点可能有两个以上的节点指向它，这个按照所有权模型怎么表述？
- 多个线程要访问同一块共享内存，怎么办？

我们知道，这些问题在程序运行过程中才会遇到，在编译期，所有权的静态检查无法处理它们，所以为了更好的灵活性，Rust 提供了运行时的动态检查，来满足特殊场景下的需求。

这也是 Rust 处理很多问题的思路：编译时，处理大部分使用场景，保证安全性和效率；运行时，处理无法在编译时处理的场景，会牺牲一部分效率，提高灵活性。后续讲到静态分发和动态分发也会有体现，这个思路很值得我们借鉴。

那具体如何在运行时做动态检查呢？运行时的动态检查又如何与编译时的静态检查结合呢？

Rust 的答案是使用引用计数的智能指针：[Rc \(Reference counter\)](#) 和 [Arc \(Atomic reference counter\)](#)。这里要特别说明一下，Arc 和 ObjC/Swift 里的 ARC (Automatic Reference Counting) 不是一个意思，不过它们解决问题的手段类似，都是通过引用计数完成的。

Rc

我们先看 Rc。对某个数据结构 T，我们可以创建引用计数 Rc，使其有多个所有者。Rc 会把对应的数据结构创建在堆上，我们在第二讲谈到过，堆是唯一可以让动态创建的数据被到处使用的内存。

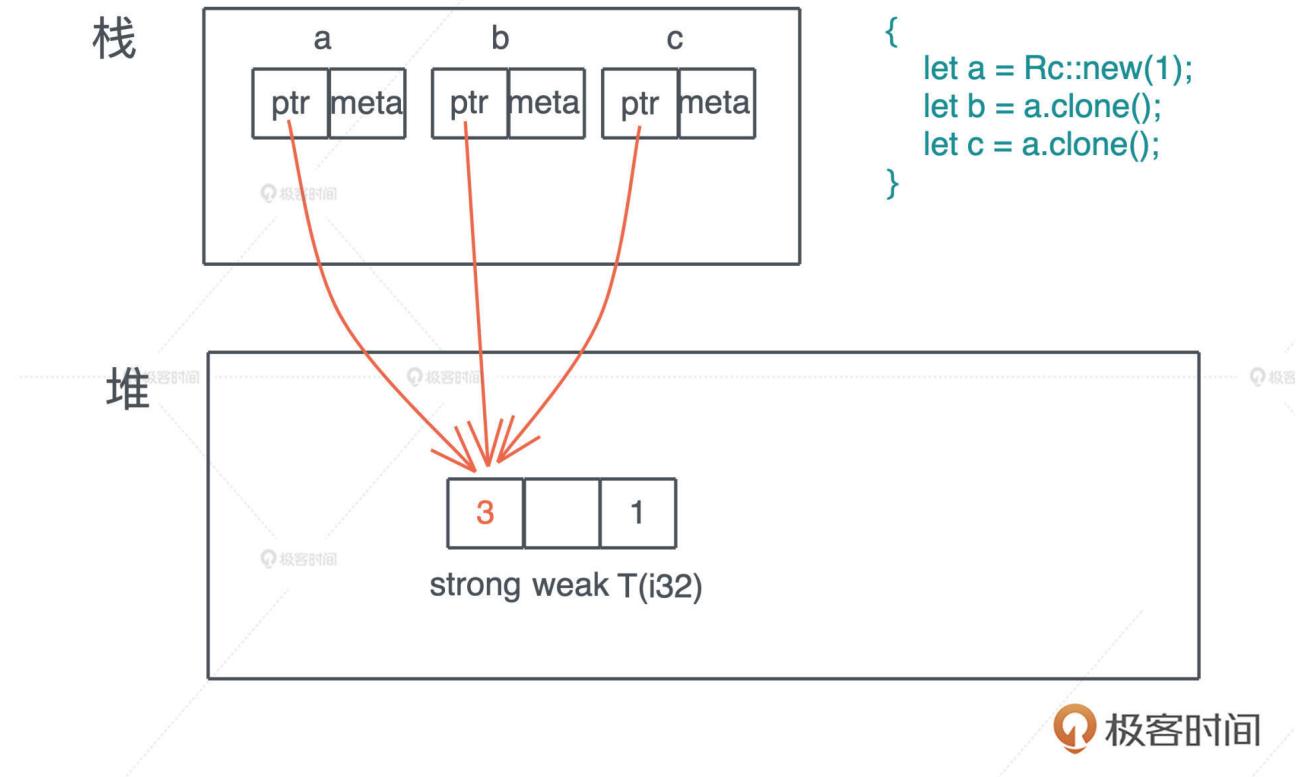
```
use std::rc::Rc;
fn main() {
    let a = Rc::new(1);
}
```

之后，如果想对数据创建更多的所有者，我们可以通过 clone() 来完成。

对一个 Rc 结构进行 clone()，不会将其内部的数据复制，只会增加引用计数。而当一个 Rc 结构离开作用域被 drop() 时，也只会减少其引用计数，直到引用计数为零，才会真正清除对应的内存。

```
use std::rc::Rc;
fn main() {
    let a = Rc::new(1);
    let b = a.clone();
    let c = a.clone();
}
```

上面的代码我们创建了三个 Rc，分别是 a、b 和 c。它们共同指向堆上相同的数据，也就是说，堆上的数据有了三个共享的所有者。在这段代码结束时，c 先 drop，引用计数变成 2，然后 b drop、a drop，引用计数归零，堆上内存被释放。



放。

你也许会有疑问：为什么我们生成了对同一块内存的多个所有者，但是，编译器不抱怨所有权冲突呢？

仔细看这段代码：首先 a 是 Rc::new(1) 的所有者，这毋庸置疑；然后 b 和 c 都调用了 a.clone()，分别得到了一个新的 Rc，所以从编译器的角度，abc 都各自拥有一个 Rc。如果文字你觉得稍微有点绕，看看 Rc 的 clone() 函数的实现，就很清楚了（[源代码](#)）：

```
fn clone(&self) -> Rc<T> {  
    // 增加引用计数  
    self.inner().inc_strong();  
    // 通过 self.ptr 生成一个新的 Rc 结构  
    Self::from_inner(self.ptr)  
}
```

所以，Rc 的 clone() 正如我们刚才说的，不复制实际的数据，只是一个引用计数的增加。

你可能继续会疑惑：Rc 是怎么产生在堆上的？并且为什么这段堆内存不受栈内存生命周期的控制呢？

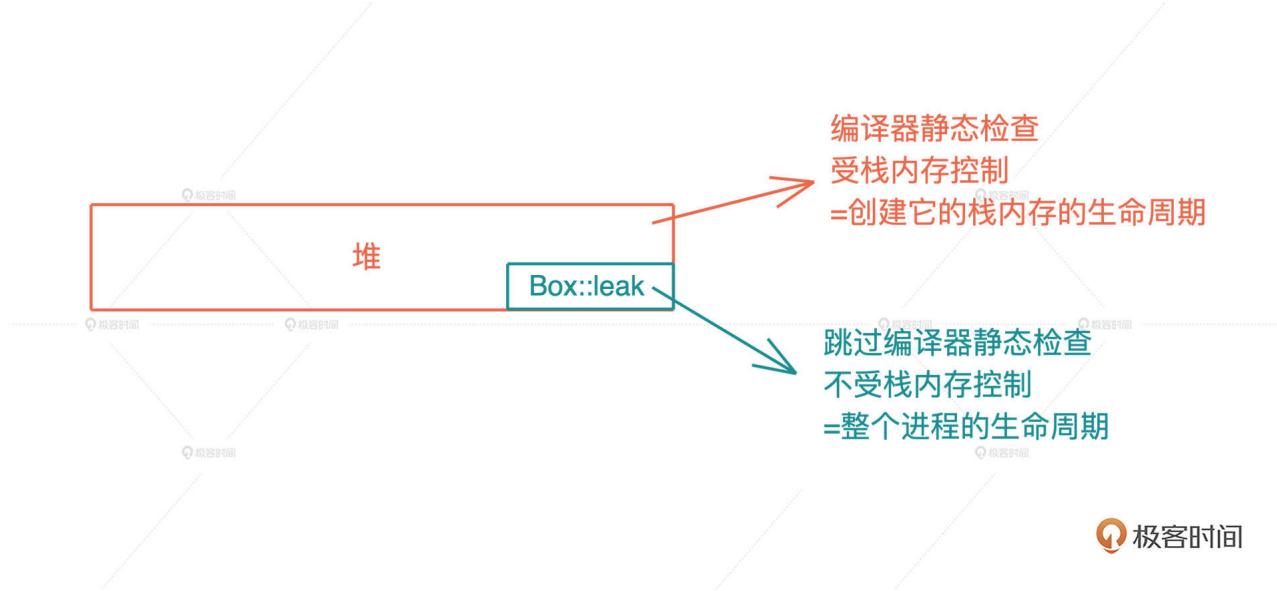
Box::leak()机制

上一讲我们讲到，在所有权模型下，堆内存的生命周期，和创建它的栈内存的生命周期保持一致。所以 Rc 的实现似乎与此格格不入。的确，如果完全按照上一讲的单一所有权模型，Rust 是无法处理 Rc 这样的引用计数的。

Rust 必须提供一种机制，让代码可以像 C/C++ 那样，**创建不受栈内存控制的堆内存**，从而绕过编译时的所有权规则。Rust 提供的方式是 Box::leak()。

Box 是 Rust 下的智能指针，它可以强制把任何数据结构创建在堆上，然后在栈上放一个指针指向这个数据结构，但此时堆内存的生命周期仍然是受控的，跟栈上的指针一致。我们后续讲到智能指针时会详细介绍 Box。

Box::leak()，顾名思义，它创建的对象，从堆内存上泄漏出去，不受栈内存控制，是一个自由的、生命周期可以大到和整个进程的生命周期一致的对象。



所以我们相当于主动撕开了一个口子，允许内存泄漏。注意，在 C/C++ 下，其实你通过 malloc 分配的每一片堆内存，都类似 Rust 下的 Box::leak()。我很喜欢 Rust 这样的设计，它符合最小权限原则（Principle of least privilege），最大程度帮助开发者撰写安全的代码。

有了 Box::leak()，我们就可以跳出 Rust 编译器的静态检查，保证 Rc 指向的堆内存，有最大的生命周期，然后我们再通过引用计数，在合适的时机，结束这段内存的生命周期。如果你对此感兴趣，可以看 [Rc::new\(\) 的源码](#)。

插一句，在学习语言的过程中，不要因为觉得自己是个初学者，就不敢翻阅标准库的源码，相反，遇到不懂的地方，如果你去看对应的源码，得到的是第一手的知识，一旦搞明白，就会学得非常扎实，受益无穷。

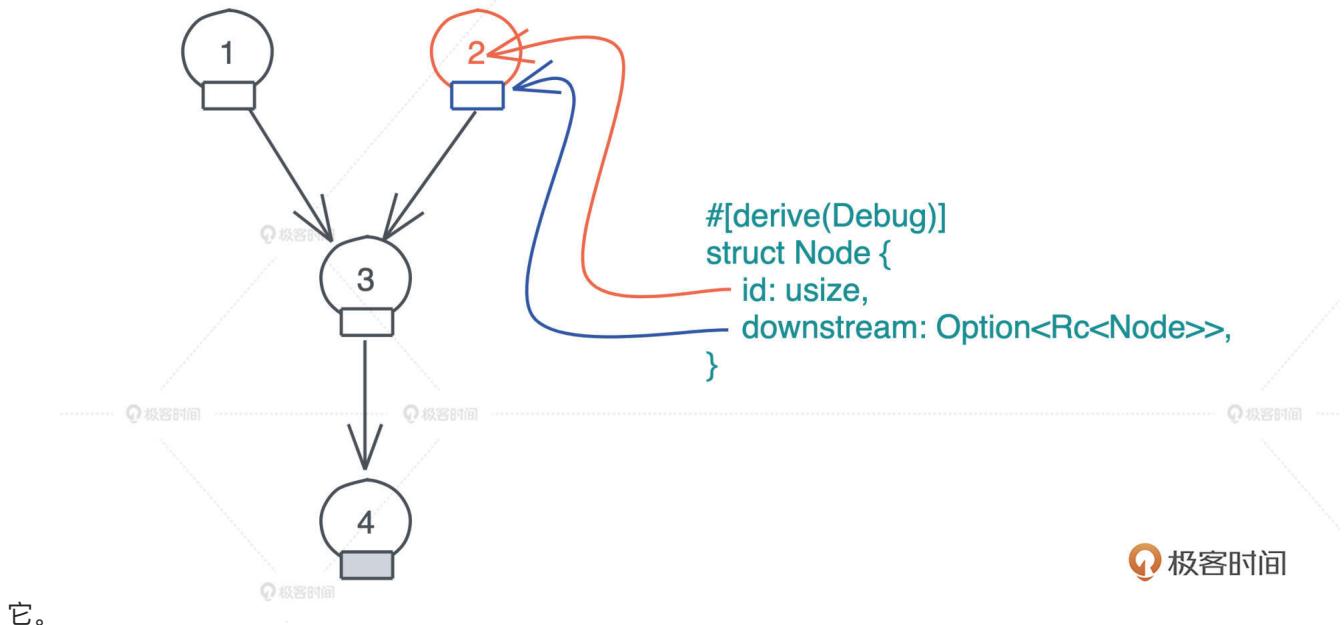
搞明白了 Rc，我们就进一步理解 Rust 是如何进行所有权的静态检查和动态检查了：

- 静态检查，靠编译器保证代码符合所有权规则；
- 动态检查，通过 Box::leak 让堆内存拥有不受限的生命周期，然后在运行过程中，通过对引用计数的检查，保证这样的堆内存最终会得到释放。

实现 DAG

现在我们用 Rc 来实现之前无法实现的 DAG。

假设 Node 就只包含 id 和指向下游（downstream）的指针，因为 DAG 中的一个节点可能被多个其它节点指向，所以我们使用 `Rc<Node>` 来表述它；一个节点可能没有下游节点，所以我们用 `Option<Rc<Node>>` 来表述



它。

要建立这样一个 DAG，我们需要为 Node 提供以下方法：

- new(): 建立一个新的 Node。
- update_downstream(): 设置 Node 的 downstream。
- get_downstream(): clone 一份 Node 里的 downstream。

有了这些方法，我们就可以创建出拥有上图关系的 DAG 了（[代码1](#)）：

```
use std::rc::Rc;
```

```

#[derive(Debug)]
struct Node {
    id: usize,
    downstream: Option<Rc<Node>>,
}

```

```

    downstream: Option<Rc<Node>>,
}

impl Node {
    pub fn new(id: usize) -> Self {
        Self {
            id,
            downstream: None,
        }
    }

    pub fn update_downstream(&mut self, downstream: Rc<Node>) {
        self.downstream = Some(downstream);
    }

    pub fn get_downstream(&self) -> Option<Rc<Node>> {
        self.downstream.as_ref().map(|v| v.clone())
    }
}

fn main() {
    let mut node1 = Node::new(1);
    let mut node2 = Node::new(2);
    let mut node3 = Node::new(3);
    let node4 = Node::new(4);
    node3.update_downstream(Rc::new(node4));

    node1.update_downstream(Rc::new(node3));
    node2.update_downstream(node1.get_downstream().unwrap());
    println!("node1: {:?}", node1, node2);
}

```

RefCell

在运行上述代码时，细心的你也许会疑惑：整个 DAG 在创建完成后还能修改么？

按最简单的写法，我们可以在上面的代码1的 `main()` 函数后，加入这段代码（[代码2](#)），来修改 `Node3` 使其指向一个新的节点 `Node5`：

```

let node5 = Node::new(5);
let node3 = node1.get_downstream().unwrap();
node3.update_downstream(Rc::new(node5));

println!("node1: {:?}", node1, node2);

```

然而，它无法编译通过，编译器会告诉你“`node3 cannot borrow as mutable`”。

这是因为Rc 是一个只读的引用计数器，你无法拿到 Rc 结构内部数据的可变引用，来修改这个数据。这可怎么办？

这里，我们需要使用 RefCell。

和 Rc 类似，RefCell 也绕过了 Rust 编译器的静态检查，允许我们在运行时，对某个只读数据进行可变借用。这就涉及 Rust 另一个比较独特且有点难懂的概念：[内部可变性 \(interior mutability\)](#)。

内部可变性

有内部可变性，自然能联想到外部可变性，所以我们先看这个更简单的定义，对比着学。

当我们用 `let mut` 显式地声明一个可变的值，或者，用 `&mut` 声明一个可变引用时，编译器可以在编译时进行严格地检查，保证只有可变的值或者可变的引用，才能修改值内部的数据，这被称作外部可变性 (exterior mutability)，[外部可变性通过 `mut` 关键字声明](#)。

然而，这样不够灵活，有时候我们希望能够绕开这个编译时的检查，对并未声明成 `mut` 的值或者引用，也想进行修改。也就是说，在编译器的眼里，值是只读的，但是在运行时，这个值可以得到可变借用，从而修改内部的数据，这就是 `RefCell` 的用武之地。

我们看一个简单的例子 ([代码2](#))：

```
use std::cell::RefCell;

fn main() {
    let data = RefCell::new(1);
    {
        // 获得 RefCell 内部数据的可变借用
        let mut v = data.borrow_mut();
        *v += 1;
    }
    println!("data: {:?}", data.borrow());
}
```

在这个例子里，`data` 是一个 `RefCell`，其初始值为 1。可以看到，我们并未将 `data` 声明为可变变量。之后我们可以通过使用 `RefCell` 的 `borrow_mut()` 方法，来获得一个可变的内部引用，然后对它做加 1 的操作。最后，我们可以通过 `RefCell` 的 `borrow()` 方法，获得一个不可变的内部引用，因为加了 1，此时它的值为 2。

你也许奇怪，这里为什么要把获取和操作可变借用的两句代码，用花括号分装到一个作用域下？

因为根据所有权规则，在同一个作用域下，我们不能同时有活跃的可变借用和不可变借用。通过这对花括号，我们明确地缩小了可变借用的生命周期，不至于和后续的不可变借用冲突。

这里再想一步，如果没有这对花括号，这段代码是无法编译通过？还是运行时会出错（[代码3](#)）？

```
use std::cell::RefCell;

fn main() {
    let data = RefCell::new(1);
    let mut v = data.borrow_mut();
    *v += 1;
    println!("data: {:?}", data.borrow());
}
```

如果你运行代码3，编译没有任何问题，但在运行到第9行时，会得到：“already mutably borrowed: BorrowError”这样的错误。可以看到，所有权的借用规则在此依旧有效，只不过它在运行时检测。

这就是外部可变性和内部可变性的重要区别，我们用下表来总结一下：

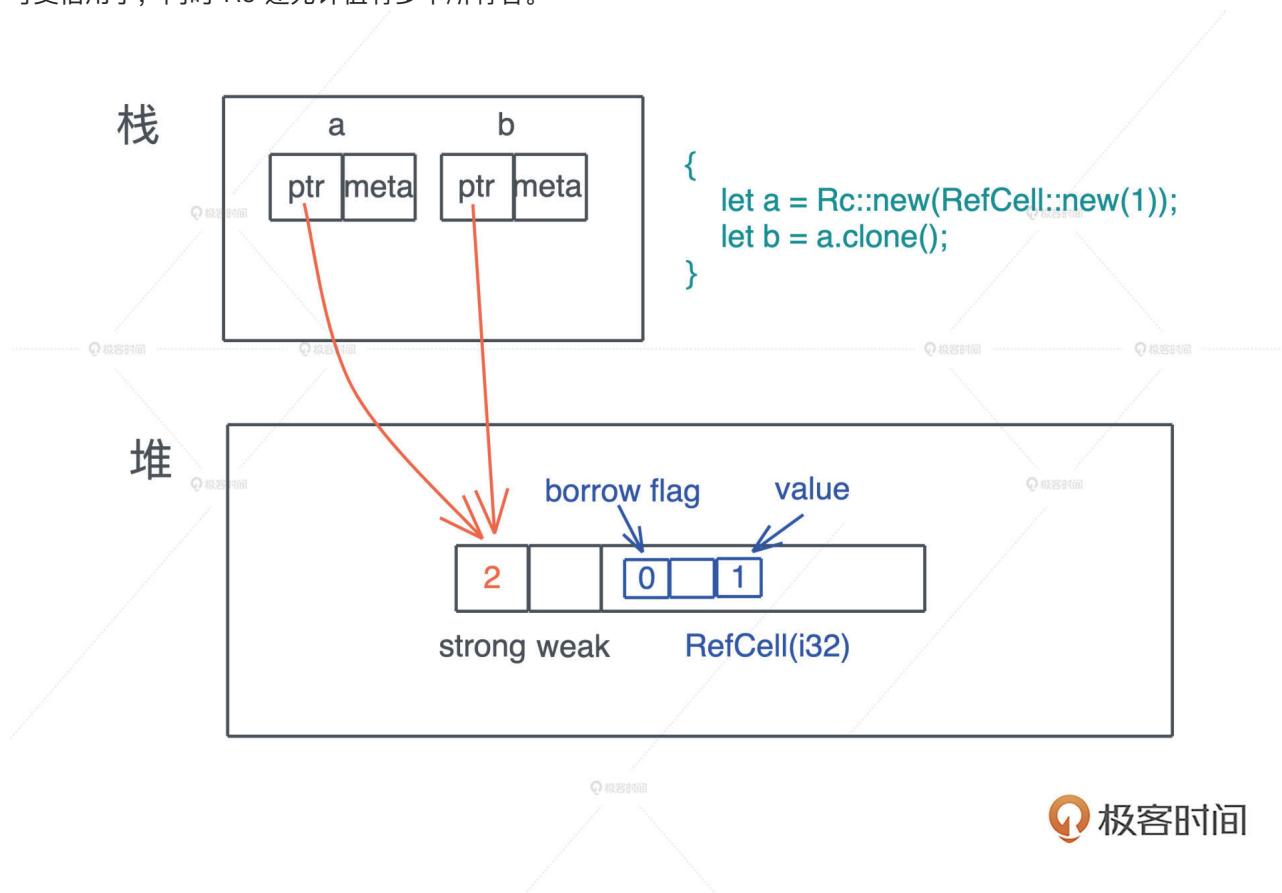
使用方法	所有权检查
外部可变性 let mut 或者 &mut	编译时，如果不符合规则，产生编译错误
内部可变性 使用 Cell / RefCell	运行时，如果不符合规则，产生 panic



实现可修改DAG

好，现在我们对 RefCell 有一个直观的印象，看看如何使用它和 Rc 来让之前的 DAG 变得可修改。

首先数据结构的 downstream 需要 Rc 内部嵌套一个 RefCell，这样，就可以利用 RefCell 的内部可变性，来获得数据的可变借用了，同时 Rc 还允许值有多个所有者。



完整的代码我放到这里了（[代码4](#)）：

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    id: usize,
    // 使用 Rc<RefCell<T>> 让节点可以被修改
    downstream: Option<Rc<RefCell<Node>>,
}

impl Node {
    pub fn new(id: usize) -> Self {
        Self {
            id,
            downstream: None,
        }
    }

    pub fn update_downstream(&mut self, downstream: Rc<RefCell<Node>>) {
        self.downstream = Some(downstream);
    }
}
```



```

pub fn get_downstream(&self) -> Option<Rc<RefCell<Node>>> {
    self.downstream.as_ref().map(|v| v.clone())
}

fn main() {
    let mut node1 = Node::new(1);
    let mut node2 = Node::new(2);
    let mut node3 = Node::new(3);
    let node4 = Node::new(4);

    node3.update_downstream(Rc::new(RefCell::new(node4)));
    node1.update_downstream(Rc::new(RefCell::new(node3)));
    node2.update_downstream(node1.get_downstream().unwrap());
    println!("node1: {:?}", node1, node2);

    let node5 = Node::new(5);
    let node3 = node1.get_downstream().unwrap();
    // 获得可变引用，来修改 downstream
    node3.borrow_mut().downstream = Some(Rc::new(RefCell::new(node5)));

    println!("node1: {:?}", node1, node2);
}

```

可以看到，通过使用 `Rc<RefCell<T>>` 这样的嵌套结构，我们的 DAG 也可以正常修改了。

Arc 和 Mutex/RwLock

我们用 `Rc` 和 `RefCell` 解决了 DAG 的问题，那么，开头提到的多个线程访问同一块内存的问题，是否也可以使用 `Rc` 来处理呢？

不行。因为 `Rc` 为了性能，使用的不是线程安全的引用计数器。因此，我们需要另一个引用计数的智能指针：`Arc`，它实现了线程安全的引用计数器。

`Arc` 内部的引用计数使用了 [Atomic Usize](#)，而非普通的 `usize`。从名称上也可以感觉出来，`Atomic Usize` 是 `usize` 的原子类型，它使用了 CPU 的特殊指令，来保证多线程下的安全。如果你对原子类型感兴趣，可以看 [std::sync::atomic](#) 的文档。

Rust 实现两套不同的引用计数数据结构，完全是为了性能考虑，从这里我们也可以感受到 Rust 对性能的极致渴求。如果不用跨线程访问，可以用效率非常高的 `Rc`；如果要跨线程访问，那么必须用 `Arc`。

同样的，`RefCell` 也不是线程安全的，如果我们要在多线程中，使用内部可变性，Rust 提供了 `Mutex` 和 `RwLock`。

这两个数据结构你应该都不陌生，Mutex是互斥量，获得互斥量的线程对数据独占访问，RwLock是读写锁，获得写锁的线程对数据独占访问，但当没有写锁的时候，允许有多个读锁。读写锁的规则和 Rust 的借用规则非常类似，我们可以类比着学。

Mutex 和 RwLock 都用在多线程环境下，对共享数据访问的保护上。刚才中我们构建的 DAG 如果要用在多线程环境下，需要把 `Rc<RefCell<T>>` 替换为 `Arc<Mutex<T>>` 或者 `Arc<RwLock<T>>`。更多有关 Arc/Mutex/RwLock 的知识，我们会在并发篇详细介绍。

小结

我们对所有权有了更深入的了解，掌握了 Rc / Arc、RefCell / Mutex / RwLock 这些数据结构的用法。

如果想绕过“一个值只有一个所有者”的限制，我们可以使用 Rc / Arc 这样带引用计数的智能指针。其中，Rc 效率很高，但只能使用在单线程环境下；Arc 使用了原子结构，效率略低，但可以安全使用在多线程环境下。

然而，Rc / Arc 是不可变的，如果想要修改内部的数据，**需要引入内部可变性**，在单线程环境下，可以在 Rc 内部使用 RefCell；在多线程环境下，可以使用 Arc 嵌套 Mutex 或者 RwLock 的方法。

你可以看这张表快速回顾：

访问方式		数据	不可变借用	可变借用
单一所有权		T	&T	&mut T
单线程	Rc<T>		&Rc<T>	无法得到可变借用
	Rc<RefCell<T>>		v.borrow()	v.borrow_mut()
共享所有权	Arc<T>		&Arc<T>	无法得到可变借用
	Arc<Mutex<T>>		v.lock()	v.lock()
	Arc<RwLock<T>>		v.read()	v.write()

1. 运行下面的代码，查看错误，并阅读 `std::thread::spawn` 的文档，找到问题的原因后，修改代码使其编译通过。

```
fn main() {
    let arr = vec![1];

    std::thread::spawn(|| {
        println!("{:?}", arr);
    });
}
```

2. 你可以写一段代码，在 `main()` 函数里生成一个字符串，然后通过 `std::thread::spawn` 创建一个线程，让 `main()` 函数所在的主线程和新的线程共享这个字符串么？提示：使用 `std::sync::Arc`。

3. 我们看到了 `Rc` 的 `clone()` 方法的实现：

```
fn clone(&self) -> Rc<T> {
    // 增加引用计数
    self.inner().inc_strong();
    // 通过 self.ptr 生成一个新的 Rc 结构
    Self::from_inner(self.ptr)
}
```

你有没有注意到，这个方法传入的参数是 `&self`，是个不可变引用，然而它调用了 `self.inner().inc_strong()`，光看函数名字，它用来增加 `self` 的引用计数，可是，为什么这里对 `self` 的不可变引用可以改变 `self` 的内部数据呢？

欢迎在留言区分享你的思考。恭喜你完成了 Rust 学习的第九次打卡，如果你觉得有收获，也欢迎分享给你身边的朋友，邀TA一起讨论。

参考资料

1. `clone()` 函数的[实现源码](#)
2. [最小权限原则](#)
3. `Rc::new()` 的[源码](#)
4. `Arc` 内部的引用计数使用了 [Atomic Usiz](#)e
5. `Atomic Usiz`e 是 `usize` 的原子类型：[std::sync::atomic](#) 的文档
6. 内部可变性：除了 `RefCell` 之外，Rust 还提供了 `Cell`。如果你想对 `RefCell` 和 `Cell` 进一步了解，可以看 Rust 标准库里[cell](#) 的文档。

生命周期

目标与意义

帮助编译器执行一个简单的规则：引用不应该活得比所指对象长(*no reference should outlive its referent*)。

因此，需要记住的第一件事就是，它们全都是关于引用 (references) 的，与其他东西无关。

生命周期这个词被宽泛地用来指代三种不同的东西——变量真实的生命周期、生命周期约束和生命周期标注。所以令人困惑！

变量的生命周期

变量的生命周期，就是其作用域。全局、函数、语句块等。变量在生命周期结束会被自动调用Drop释放。

一般来说，堆内存的生命周期，会默认和其栈内存的生命周期绑定在一起。

生命周期约束

变量在代码中的交互方式对它们的生命周期有一定的约束。例如，在下面的代码中，`x=&y;`这一行添加了一个约束，`x`的生命周期应该封闭在`y`的生命周期之内。

约束并不改变真实的生命周期。

生命周期标注

所有变量都有生命周期标注，只不过编译器自动标注了绝大多数场景。

要创建一个生命周期标注，必须先声明一个**生命周期参数**。例如，`<'a>`是一个生命周期声明。生命周期参数是一种泛型参数，你可以把`<'a>`读作”对于某生命周期'a...”。一旦一个生命周期参数被声明，它就可以在**引用**中使用以创建一个生命周期约束。

一般而言，当相同的生命周期参数标注了一个函数中两个及以上的参数，返回的引用必须不能活得比参数生命周期中最小的那个长。

生命周期省略

被省略的不是生命周期，而是生命周期标注以及对应的生命周期约束。

下列2、3情况中可以省略生命周期标注：

- 所有引用类型的参数都有独立的生命周期 '`a`、'`b` 等。
- 只有一个输入引用时。
- 当有多个输入引用，但是第一个参数是`&self`或者`&mut self`时。

01 | 生命周期：你创建的值究竟能活多久？

之前提到过，在任何语言里，栈上的值都有自己的生命周期，它和帧的生命周期一致，而 Rust，进一步明确这个概念，并且为堆上的内存也引入了生命周期。

我们知道，在其它语言中，堆内存的生命周期是不确定的，或者是未定义的。因此，要么开发者手工维护，要么语言在运行时做额外的检查。而在 Rust 中，除非显式地做 `Box::leak()` / `Box::into_raw()` / `ManualDrop` 等动作，**一般来说，堆内存的生命周期，会默认和其栈内存的生命周期绑定在一起。**

所以在这种默认情况下，在每个函数的作用域中，编译器就可以对比值和其引用的生命周期，来确保“引用的生命周期不超出值的生命周期”。

那你有没有想过，Rust 编译器是如何做到这一点的呢？

值的生命周期

在进一步讨论之前，我们先给值可能的生命周期下个定义。

如果一个值的生命周期贯穿整个进程的生命周期，那么我们就称这种生命周期为**静态生命周期**。

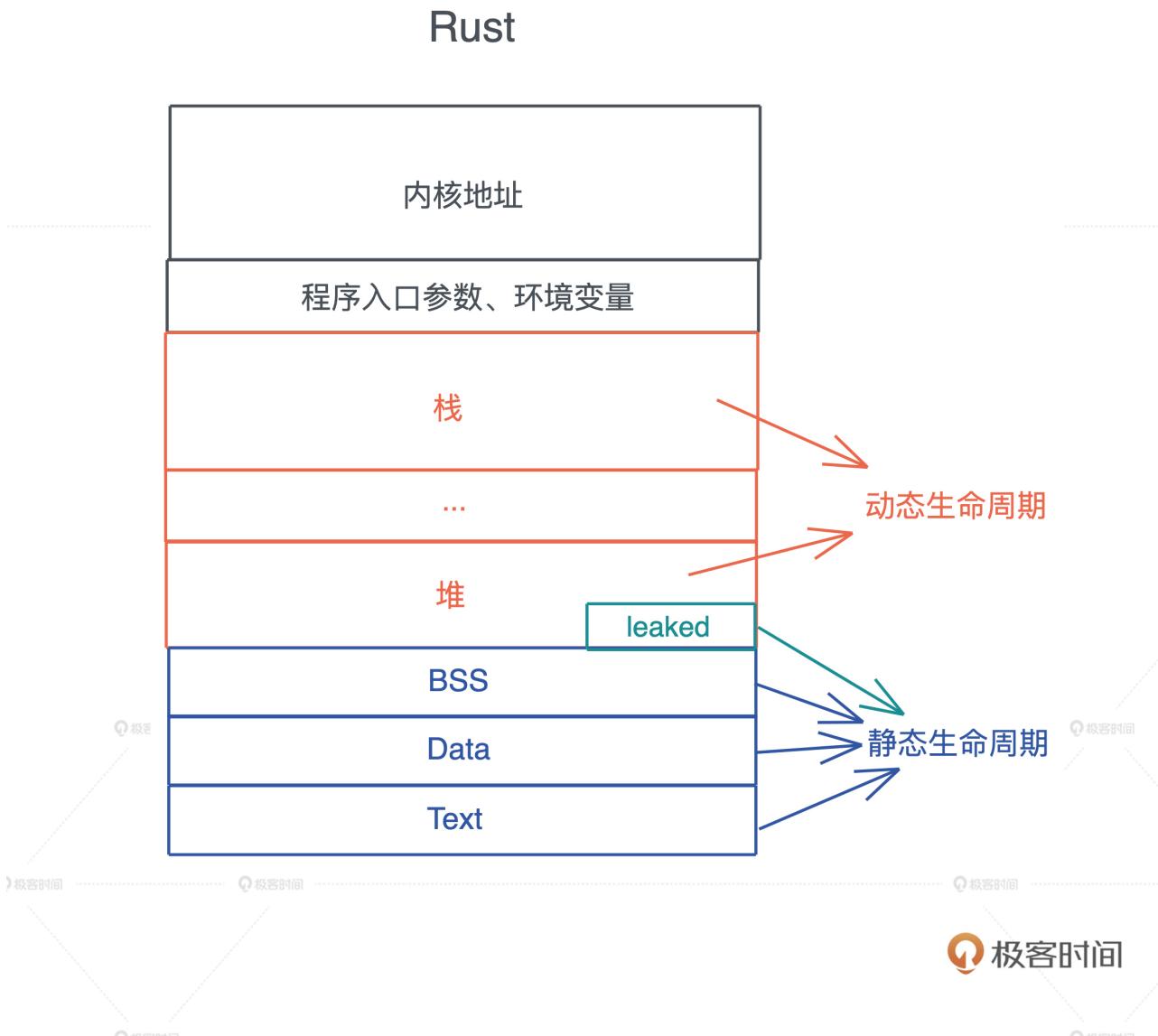
当值拥有静态生命周期，其引用也具有静态生命周期。我们在表述这种引用的时候，可以用 `'static` 来表示。比如：`&static str` 代表这是一个具有静态生命周期的字符串引用。

一般来说，全局变量、静态变量、字符串字面量（string literal）等，都拥有静态生命周期。我们上文中提到的堆内存，如果使用了 `Box::leak` 后，也具有静态生命周期。

如果一个值是在某个作用域中定义的，也就是说它被创建在栈上或者堆上，那么其生命周期是动态的。

当这个值的作用域结束时，值的生命周期也随之结束。对于动态生命周期，我们约定用 `'a`、`'b` 或者 `'hello` 这样的小写字符或者字符串来表达。`'` 后面具体是什么名字不重要，它代表某一段动态的生命周期，其中，`&'a str` 和 `&'b str` 表示这两个字符串引用的生命周期可能不一致。

我们通过图总结一下：



- 分配在堆和栈上的内存有其各自的作用域，它们的生命周期是动态的。
- 全局变量、静态变量、字符串字面量、代码等内容，在编译时，会被编译到可执行文件中的 BSS/Data/RoData /Text 段，然后在加载时，装入内存。因而，它们的生命周期和进程的生命周期一致，所以是静态的。
- 所以，函数指针的生命周期也是静态的，因为函数在 Text 段中，只要进程活着，其内存一直存在。

明白了这些基本概念后，我们来看对于值和引用，编译器是如何识别其生命周期的。

编译器如何识别生命周期

我们先从两个最基本最简单的例子开始。

左图的**例1**，`x` 引用了在内层作用域中创建出来的变量 `y`。由于，变量从开始定义到其作用域结束的这段时间，是它的生命周期，所以 `x` 的生命周期 '`a`' 大于 `y` 的生命周期 '`b`'，当 `x` 引用 `y` 时，编译器报错。

右图**例2** 中，`y` 和 `x` 处在同一个作用域下，`x` 引用了 `y`，我们可以看到 `x` 的生命周期 '`a`' 和 `y` 的生命周期 '`b`' 几乎同时结束，或者说 '`a`' 小于等于 '`b`'，所以，`x` 引用 `y` 是可行的。

```
fn main() {  
    let x; // x 生命周期 'a'  
    {  
        let y = 42; // y 生命周期 'b'  
        x = &y;  
    }  
    println!("x: {}", x);  
}
```

例 1: '`a`>'`b`, `x` (`y` 的引用) 生命周期大于 `y`

```
fn main() {  
    let y = 42; // y 生命周期 'b'  
    let x = &y; // x 生命周期 'a'  
    println!("x: {}", x);  
}
```

例 2: '`a`<='`b`, `x` (`y` 的引用) 生命周期等于 `y`

极客时间

这两个小例子很好理解，我们再看个稍微复杂一些的。

示例代码在 `main()` 函数里创建了两个 `String`，然后将其传入 `max()` 函数比较大小。`max()` 函数接受两个字符串引用，返回其中较大的那个字符串的引用（[示例代码](#)）：

```
fn main() {  
    let s1 = String::from("Lindsey");  
    let s2 = String::from("Rosie");  
  
    let result = max(&s1, &s2);  
  
    println!("bigger one: {}", result);  
}
```

```
fn max(s1: &str, s2: &str) -> &str {  
    if s1 > s2 {  
        s1  
    } else {  
        s2  
    }  
}
```

这段代码是无法编译通过的，它会报错“missing lifetime specifier”，也就是说，**编译器在编译 `max()` 函数时，无法判断 `s1`、`s2` 和返回值的生命周期**。

你是不是很疑惑，站在我们开发者的角度，这个代码理解起来非常直观，在 main() 函数里 s1 和 s2 两个值生命周期一致，它们的引用传给 max() 函数之后，无论谁的被返回，生命周期都不会超过 s1 或 s2。所以这应该是一段正确的代码啊？

为什么编译器报错了，不允许它编译通过呢？我们把这段代码稍微扩展一下，你就能明白编译器的困惑了。

在刚才的示例代码中，我们创建一个新的函数 get_max()，它接受一个字符串引用，然后和“Cynthia”这个字符串字面量比较大小。之前我们提到，**字符串字面量的生命周期是静态的，而 s1 是动态的，它们的生命周期显然不一致**（[代码](#)）：

```
fn main() {
    let s1 = String::from("Lindsey");
    let s2 = String::from("Rosie");

    let result = max(&s1, &s2);

    println!("bigger one: {}", result);

    let result = get_max(&s1);
    println!("bigger one: {}", result);
}

fn get_max(s1: &str) -> &str {
    max(s1, "Cynthia")
}

fn max(s1: &str, s2: &str) -> &str {
    if s1 > s2 {
        s1
    } else {
        s2
    }
}
```

当出现了多个参数，它们的生命周期可能不一致时，返回值的生命周期就不好确定了。编译器在编译某个函数时，并不知道这个函数将来有谁调用、怎么调用，所以，**函数本身携带的信息，就是编译器在编译时使用的全部信息**。

根据这一点，我们再看示例代码，在编译 max() 函数时，参数 s1 和 s2 的生命周期是什么关系、返回值和参数的生命周期又有什么关系，编译器是无法确定的。

此时，就需要我们在函数签名中提供生命周期的信息，也就是生命周期标注（lifetime specifier）。在生命周期标注时，使用的参数叫生命周期参数（lifetime parameter）。通过生命周期标注，我们告诉编译器这些引用间生命周期的约束。

生命周期参数的描述方式和泛型参数一致，不过只使用小写字母。这里，两个入参 s1、s2，以及返回值都用 `'a` 来约束。**生命周期参数，描述的是参数和参数之间、参数和返回值之间的关系，并不改变原有的生命周期。**

在我们添加了生命周期参数后，s1 和 s2 的生命周期只要大于等于 `(outlive) 'a`，就符合参数的约束，而返回值的生命周期同理，也需要大于等于 `'a`。

在你运行上述示例代码的时候，编译器已经提示你，可以这么修改 `max()` 函数：

```
fn max<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1 > s2 {
        s1
    } else {
        s2
    }
}
```

当 `main()` 函数调用 `max()` 函数时，s1 和 s2 有相同的生命周期 `'a`，所以它满足 `(s1: &'a str, s2: &'a str)` 的约束。当 `get_max()` 函数调用 `max()` 时，“Cynthia”是静态生命周期，它大于 s1 的生命周期 `'a`，所以它也可以满足 `max()` 的约束需求。

你的引用需要额外标注吗

学到这里，你可能会有困惑了：为什么我之前写的代码，很多函数的参数或者返回值都使用了引用，编译器却没有提示我要额外标注生命周期呢？

这是因为编译器希望尽可能减轻开发者的负担，其实所有使用了引用的函数，都需要生命周期的标注，只不过编译器会自动做这件事，省却了开发者的麻烦。

比如这个例子，`first()` 函数接受一个字符串引用，找到其中的第一个单词并返回（[代码](#)）：

```
fn main() {
    let s1 = "Hello world";
    println!("first word of s1: {}", first(&s1));
}
```

```
fn first(s: &str) -> &str {
    let trimmed = s.trim();
    match trimmed.find(' ') {
        None => "",
```

```
    Some(pos) => &trimmed[..pos],  
}  
}
```

虽然我们没有做任何生命周期的标注，但编译器会通过一些简单的规则为函数自动添加标注：

- 所有引用类型的参数都有独立的生命周期 `'a` 、 `'b` 等。
- 如果只有一个引用型输入，它的生命周期会赋给所有输出。
- 如果有多个引用类型的参数，其中一个是 `self`，那么它的生命周期会赋给所有输出。

规则 3 适用于 trait 或者自定义数据类型，我们先放在一边，以后遇到会再详细讲的。例子中的 `first()` 函数通过规则 1 和 2，可以得到一个带生命周期的版本（[代码](#)）：

```
fn first<'a>(s: &'a str) -> &'a str {  
    let trimmed = s.trim();  
    match trimmed.find(' ') {  
        None => "",  
        Some(pos) => &trimmed[..pos],  
    }  
}
```

你可以看到，所有引用都能正常标注，没有冲突。那么对比之前返回较大字符串的示例代码（[示例代码](#)），`max()` 函数为什么编译器无法处理呢？

按照规则 1，我们可以对 `max()` 函数的参数 `s1` 和 `s2` 分别标注 `'a` 和 `'b`，但是返回值如何标注？是 `'a` 还是 `'b` 呢？这里的冲突，编译器无能为力。

```
fn max<'a, 'b>(s1: &'a str, s2: &'b str) -> &'??? str
```

所以，只有我们明白了代码逻辑，才能正确标注参数和返回值的约束关系，顺利编译通过。

引用标注小练习

好，Rust 的生命周期这个知识点我们就讲完了，接下来我们来尝试写一个字符串分割函数 `strtok()`，即时练习一下，如何加引用标注。

相信有过 C/C++ 经验的开发者都接触过这个 `strtok()` 函数，它会把字符串按照分隔符（delimiter）切出一个 token 并返回，然后将传入的字符串引用指向后续的 token。

用 Rust 实现并不困难，由于传入的 `s` 需要可变的引用，所以它是一个指向字符串引用的可变引用 `&mut &str`。
[\(练习代码\)](#)：

```
pub fn strtok(s: &mut &str, delimiter: char) -> &str {
    if let Some(i) = s.find(delimiter) {
        let prefix = &s[..i];
        // 由于 delimiter 可以是 utf8，所以我们需要获得其 utf8 长度，
        // 直接使用 len 返回的是字节长度，会有问题
        let suffix = &s[(i + delimiter.len_utf8())..];
        *s = suffix;
        prefix
    } else { // 如果没找到，返回整个字符串，把原字符串指针 s 指向空串
        let prefix = *s;
        *s = "";
        prefix
    }
}

fn main() {
    let s = "hello world".to_owned();
    let mut s1 = s.as_str();
    let hello = strtok(&mut s1, ' ');
    println!("hello is: {}, s1: {}, s: {}", hello, s1, s);
}
```

当我们尝试运行这段代码时，会遇到生命周期相关的编译错误。类似刚才讲的示例代码，是因为按照编译器的规则，`&mut &str` 添加生命周期后变成 `&'b mut &'a str`，这将导致返回的 `'&str` 无法选择一个合适的生命周期。

要解决这个问题，我们首先要思考一下：返回值和谁的生命周期有关？是指向字符串引用的可变引用 `&mut`，还是字符串引用 `&str` 本身？

显然是后者。所以，我们可以为 `strtok` 添加生命周期标注：

```
pub fn strtok<'b, 'a>(s: &'b mut &'a str, delimiter: char) -> &'a str {...}
```

因为返回值的生命周期跟字符串引用有关，我们只为这部分的约束添加标注就可以了，剩下的标注交给编译器自动添加，所以代码也可以简化成如下这样，让编译器将其扩展成上面的形式：

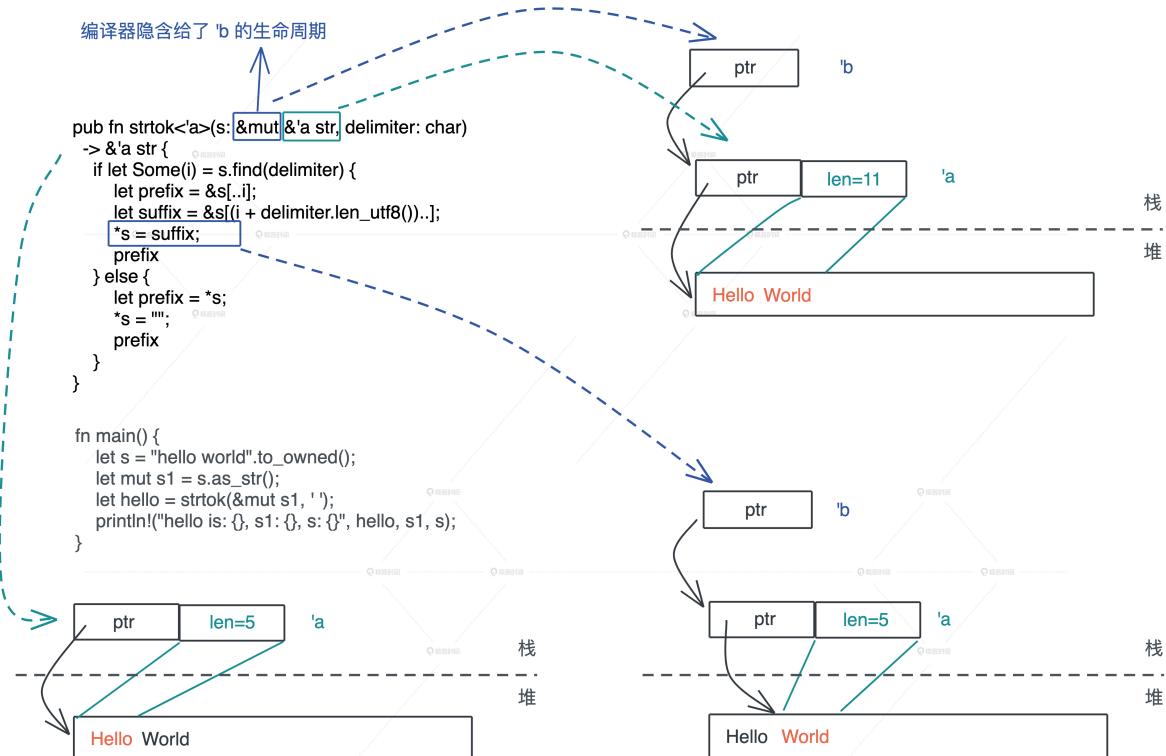
```
pub fn strtok<'a>(s: &mut &'a str, delimiter: char) -> &'a str {...}
```

最终，正常工作的代码如下（[练习代码_改](#)），可以通过编译：

```
pub fn strtok<'a>(s: &mut &'a str, delimiter: char) -> &'a str {
    if let Some(i) = s.find(delimiter) {
        let prefix = &s[..i];
        let suffix = &s[(i + delimiter.len_utf8())..];
        *s = suffix;
        prefix
    } else {
        let prefix = *s;
        *s = "";
        prefix
    }
}

fn main() {
    let s = "hello world".to_owned();
    let mut s1 = s.as_str();
    let hello = strtok(&mut s1, ' ');
    println!("hello is: {}, s1: {}, s: {}", hello, s1, s);
}
```

为了帮助你更好地理解这个函数的生命周期关系，我将每个堆上和栈上变量的关系画了个图供你参考。



这里跟你分享一个小技巧：如果你觉得某段代码理解或者分析起来很困难，也可以画类似的图，从最基础的数据在堆和栈上的关系开始想，就很容易厘清脉络。

在处理生命周期时，编译器会根据一定规则自动添加生命周期的标注。然而，当自动标注产生冲突时，需要我们手工标注。

生命周期标注的目的是，在参数和返回值之间建立联系或者约束。调用函数时，传入的参数的生命周期需要大于等于 (outlive) 标注的生命周期。

当每个函数都添加好生命周期标注后，编译器，就可以从函数调用的上下文中分析出，在传参时，引用的生命周期，是否和函数签名中要求的生命周期匹配。如果不匹配，就违背了“引用的生命周期不能超出值的生命周期”，编译器就会报错。

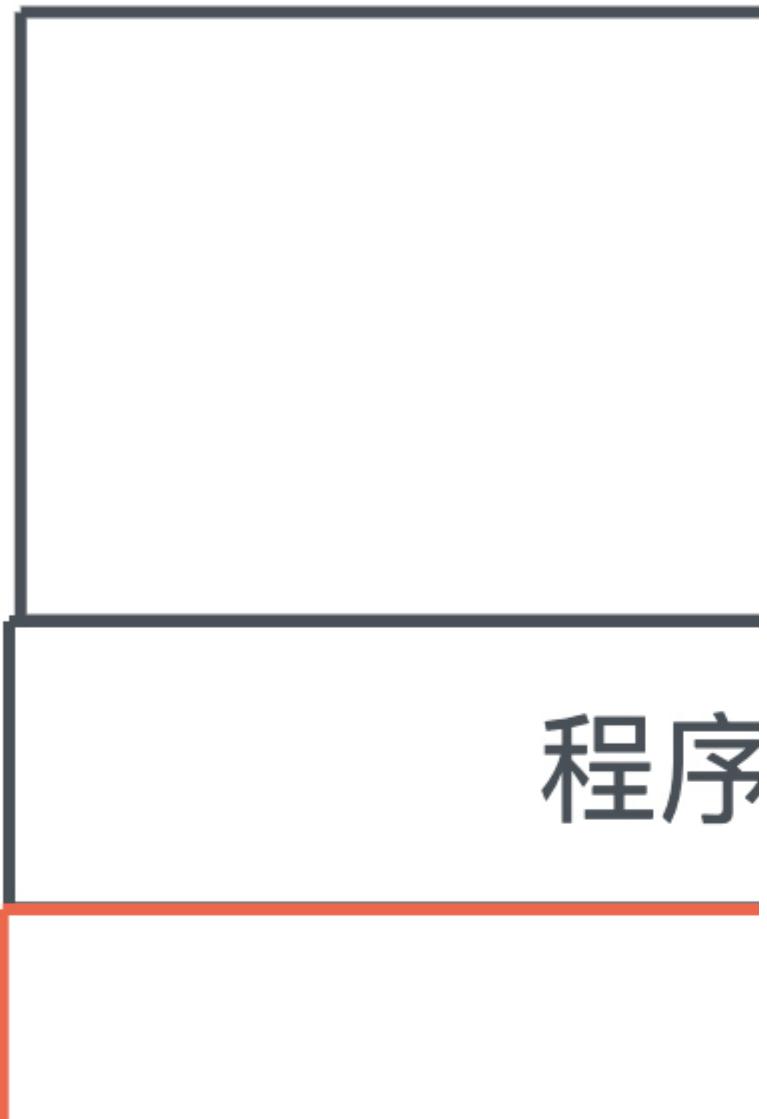
如果你搞懂了函数的生命周期标注，那么数据结构的生命周期标注也是类似。比如下面的例子，Employee 的 name 和 title 是两个字符串引用，Employee 的生命周期不能大于它们，否则会访问失效的内存，因而我们需要妥善标注：

```
struct Employee<'a, 'b> {
    name: &'a str,
    title: &'b str,
    age: u8,
}
```

使用数据结构时，数据结构自身的生命周期，需要小于等于其内部字段的所有引用的生命周期。

小结

今天我们介绍了静态生命周期和动态生命周期的概念，以及编译器如何识别值和引用的生命周期。



程序



极客



根据所有权规则，值的生命周期可以确认，它可以一直存活到所有者离开作用域；而引用的生命周期不能超过值的生命周期。在同一个作用域下，这是显而易见的。然而，当发生函数调用时，编译器需要通过函数的签名来确定，参数和返回值之间生命周期的约束。

大多数情况下，编译器可以通过上下文中的规则，自动添加生命周期的约束。如果无法自动添加，则需要开发者手工来添加约束。一般，我们只需要确定好返回值和哪个参数的生命周期相关就可以了。而对于数据结构，当内部有引用时，我们需要为引用标注生命周期。

思考题

1. 如果我们把 `strtok()` 函数的签名写成这样，会发生什么问题？为什么它会发生这个问题？你可以试着编译一下看看。

```
pub fn strtok<'a>(s: &'a mut &str, delimiter: char) -> &'a str {...}
```

2. 回顾[第 6 讲 SQL 查询工具](#)的代码，现在，看看你是不是对代码中的生命周期标注有了更深理解？

感谢你的收听，你已经打卡 Rust 学习 10 次啦！

如果你觉得有收获，也欢迎你分享给你身边的朋友，邀他一起讨论。坚持学习，我们下节课见。

参考资料

1. 栈上的内存不必特意释放，顶多是编译时编译器不再允许该变量被访问。因为栈上的内存会随着栈帧的结束而结束。如果你有点模糊，可以再看看[前置知识](#)，温习一下栈和堆。
2. Rust 的 I/O 安全性目前是“almost safety”，为什么不是完全安全，感兴趣的同学可以看这个 [RFC](#)。
3. 更多[关于 Box::leak 的信息](#)。
4. [ArcInner 的结构](#)。
5. Rust 的生命周期管理一直在进化，进化方向是在常见的场景下，尽量避免因为生命周期的处理，代码不得不换成不那么容易阅读的写作方式。比如下面的代码：

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert("hello", "world");
    let key = "hello1";

    // 按照之前的说法，这段代码无法编译通过，因为同一个 scope 下不能有两个可变引用
    // 但因为 RFC2094 non-lexical lifetimes，Rust 编译器可以处理这个场景，
    // 因为当 None 时，map.get_mut() 的引用实际已经结束
    match map.get_mut(key) /* <---- 可变引用的生命周期一直持续到 match 结果 */ {
        Some(v) => do_something(v),
        None => {
            map.insert(key, "tyr"); // <--- 这里又获得了一个可变引用
        }
    }
}

fn do_something(_v: &mut &str) {
    todo!()
}
```

如果你对此感兴趣，想了解更多，可以参看：[RFC2094 – Non-lexical lifetimes](#)。我们在平时写代码时，可以就像这段代码这样先按照正常的方式去写，如果编译器抱怨，再分析引用的生命周期，换个写法。此外，随时保持你的 Rust 版本是最新的，也有助于让你的代码总是可以使用最简单的方式撰写。

02 | 内存管理：从创建到消亡，值都经历了什么？

初探 Rust 以来，我们一直在学习有关所有权和生命周期的内容，想必现在，你对 Rust 内存管理的核心思想已经有足够理解了。

通过单一所有权模式，Rust 解决了堆内存过于灵活、不容易安全高效地释放的问题，既避免了手工释放内存带来的巨大心智负担和潜在的错误；又避免了全局引入追踪式 GC 或者 ARC 这样的额外机制带来的效率问题。

不过所有权模型也引入了很多新概念，从 Move / Copy / Borrow 语义到生命周期管理，所以学起来有些难度。

但是，你发现了吗，其实大部分新引入的概念，包括 Copy 语义和值的生命周期，在其它语言中都是隐式存在的，只不过 Rust 把它们定义得更清晰，更明确地界定了使用的范围而已。

今天我们沿着之前的思路，先梳理和总结 Rust 内存管理的基本内容，然后从一个值的奇幻之旅讲起，看看在内存中，一个值，从创建到消亡都经历了什么，把之前讲的融会贯通。

到这里你可能有点不耐烦了吧，怎么今天又要讲内存的知识。其实是因为，**内存管理是任何编程语言的核心**，重要性就像武学中的内功。只有当我们把数据在内存中如何创建、如何存放、如何销毁弄明白，之后阅读代码、分析问题才会有一种游刃有余的感觉。

内存管理

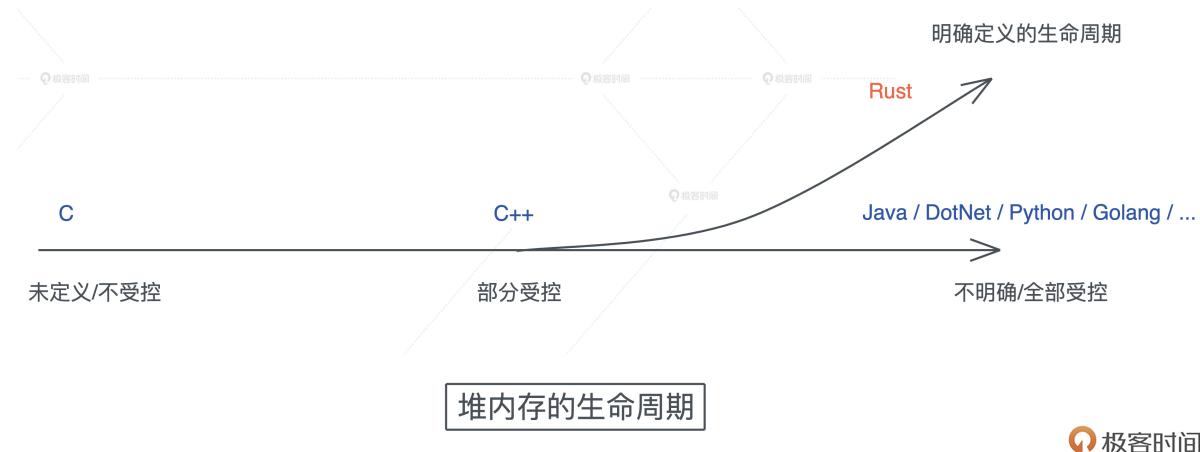
我们在[第一讲](#)说过堆和栈，它们是代码中使用内存的主要场合。

栈内存“分配”和“释放”都很高效，在编译期就确定好了，因而它无法安全承载动态大小或者生命周期超出帧存活范围外的值。所以，我们需要运行时可以自由操控的内存，也就是堆内存，来弥补栈的缺点。

堆内存足够灵活，然而堆上数据的生命周期该如何管理，成为了各门语言的心头大患。

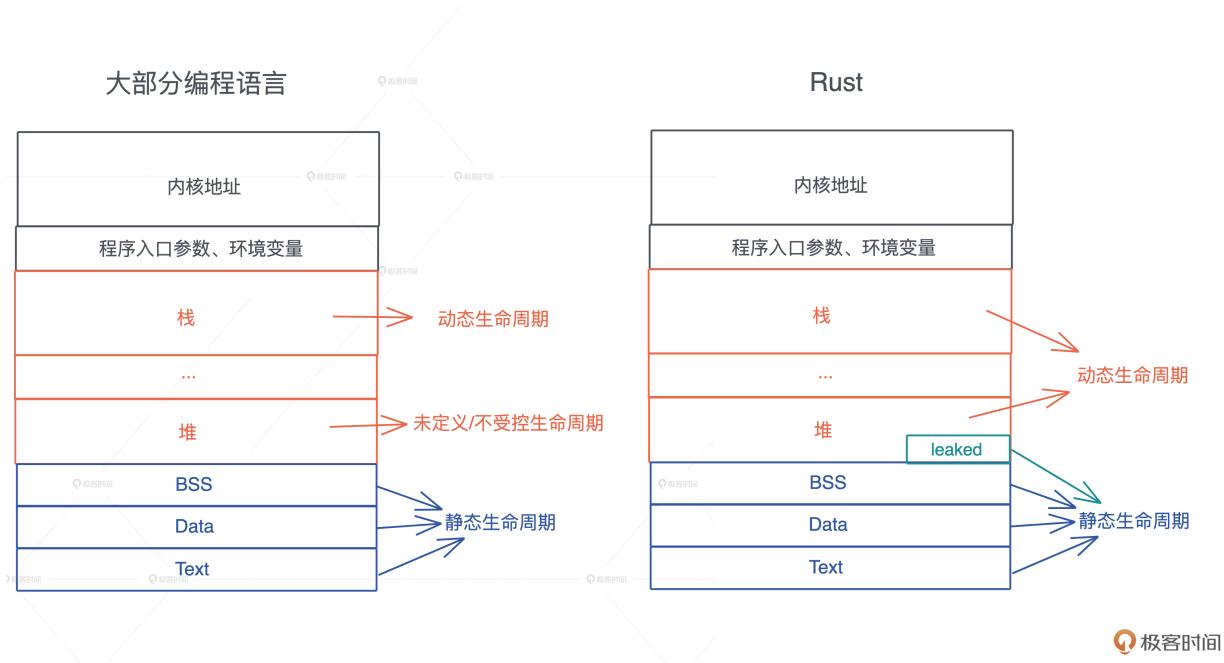
C 采用了未定义的方式，由开发者手工控制；C++ 在 C 的基础上改进，引入智能指针，半手工半自动。Java 和 DotNet 使用 GC 对堆内存全面接管，堆内存进入了受控（managed）时代。所谓受控代码（managed code），就是代码在一个“运行时”下工作，由运行时来保证堆内存的安全访问。

整个堆内存生命周期管理的发展史如下图所示：



而Rust的创造者们，重新审视了堆内存的生命周期，发现大部分堆内存的需求在于动态大小，小部分需求是更长的生命周期。所以它默认将堆内存的生命周期和使用它的栈内存的生命周期绑在一起，并留了个小口子 leaked机制，让堆内存 在需要的时候，可以有超出帧存活期的生命周期。

我们看下图的对比总结：



有了这些基本的认知，我们再看看在值的创建、使用和销毁的过程中，Rust是如何管理内存的。

希望学完今天的内容之后，看到一个 Rust 的数据结构，你就可以在脑海中大致浮现出，这个数据结构在内存中的布局：哪些字段在栈上、哪些在堆上，以及它大致的大小。

值的创建

当我们为数据结构创建一个值，并将其赋给一个变量时，根据值的性质，它有可能被创建在栈上，也有可能被创建在堆上。

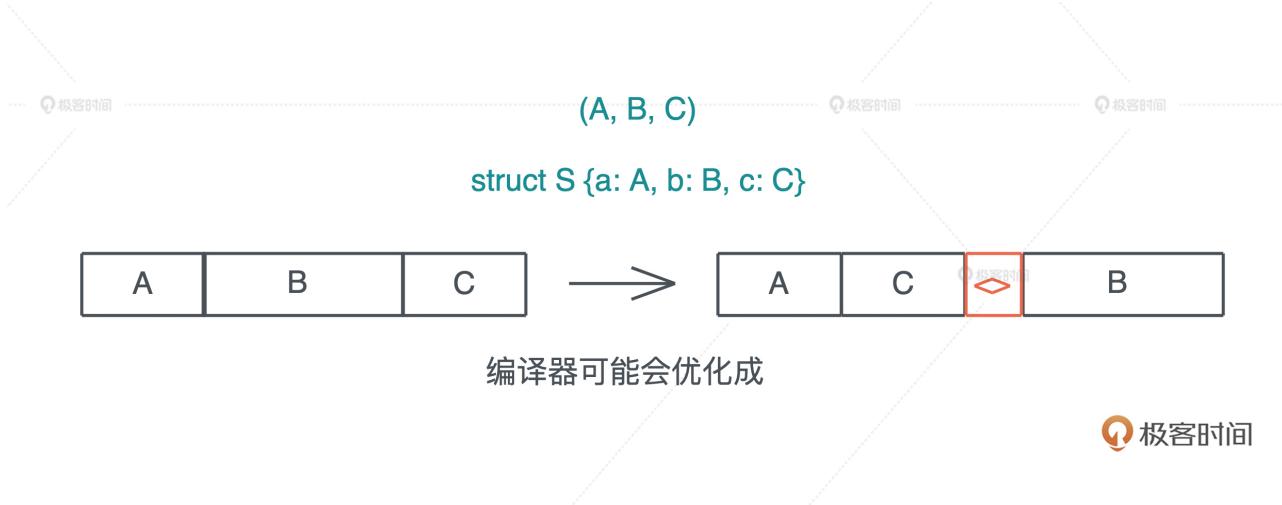
简单回顾一下，我们在第一、[第二讲](#)说过，理论上，编译时可以确定大小的值都会放在栈上，包括 Rust 提供的原生类型比如字符、数组、元组 (tuple) 等，以及开发者自定义的固定大小的结构体 (struct)、枚举 (enum) 等。

如果数据结构的大小无法确定，或者它的大小确定但是在使用时需要更长的生命周期，就最好放在堆上。

接下来我们来看 struct / enum / vec / String 这几种重要的数据结构在创建时的内存布局。

struct

Rust 在内存中排布数据时，会根据每个域的对齐（alignment）对数据进行重排，使其内存大小和访问效率最好。比如，一个包含 A、B、C 三个域的 struct，它在内存中的布局可能是 A、C、B：



极客时间

为什么 Rust 编译器会这么做呢？

我们先看看 C 语言在内存中表述一个结构体时遇到的问题。来写一段代码，其中两个数据结构 S1 和 S2 都有三个域 a、b、c，其中 a 和 c 是 u8，占用一个字节，b 是 u16，占用两个字节。S1 在定义时顺序是 a、b、c，而 S2 在定义时顺序是 a、c、b：

猜猜看 S1 和 S2 的大小是多少？

```
#include <stdio.h>

struct S1 {
    u_int8_t a;
    u_int16_t b;
    u_int8_t c;
};

struct S2 {
    u_int8_t a;
    u_int8_t c;
    u_int16_t b;
};

void main() {
    printf("size of S1: %d, S2: %d", sizeof(struct S1), sizeof(struct S2));
}
```

正确答案是：6 和 4。

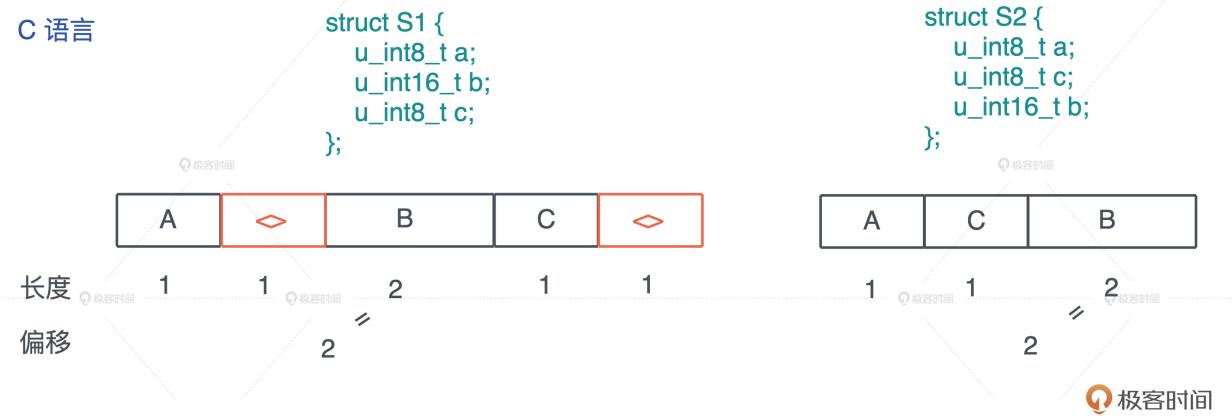
为什么明明只用了 4 个字节，S1 的大小却是 6 呢？这是因为 CPU 在加载不对齐的内存时，性能会急剧下降，所以**要避免用户定义不对齐的数据结构时，造成的性能影响**。

对于这个问题，C 语言会对结构体会做这样的处理：

- 首先确定每个域的长度和对齐长度，原始类型的对齐长度和类型的长度一致。
- 每个域的起始位置要和其对齐长度对齐，如果无法对齐，则添加 padding 直至对齐。
- 结构体的对齐大小和其最大域的对齐大小相同，而结构体的长度则四舍五入到其对齐的倍数。

字面上看这三条规则，你是不是觉得像绕口令，别担心，我们结合刚才的代码再来看，其实很容易理解。

对于 S1，字段 a 是 u8 类型，所以其长度和对齐都是 1，b 是 u16，其长度和对齐是 2。然而因为 a 只占了一个字节，b 的偏移是 1，根据第二条规则，起始位置和 b 的长度无法对齐，所以编译器会添加一个字节的 padding，让 b 的偏移为 2，这样 b 就对齐了。



随后 c 长度和对齐都是 1，不需要 padding。这样算下来，S1 的大小是 5，但根据上面的第三条规则，S1 的对齐是 2，和 5 最接近的“2 的倍数”是 6，所以 S1 最终的长度是 6。其实，这最后一条规则是为了让 S1 放在数组中，可以有效对齐。

所以，**如果结构体的定义考虑地不够周全，会为了对齐浪费很多空间**。我们看到，保存同样的数据，S1 和 S2 的大小相差了 50%。

使用 C 语言时，定义结构体的最佳实践是，充分考虑每一个域的对齐，合理地排列它们，使其内存使用最高效。这个工作由开发者做会很费劲，尤其是嵌套的结构体，需要仔细地计算才能得到最优解。

而 Rust 编译器替我们自动完成了这个优化，这就是为什么 Rust 会自动重排你定义的结构体，来达到最高效率。我们看同样的代码，在 Rust 下，S1 和 S2 大小都是 4（[代码](#)）：

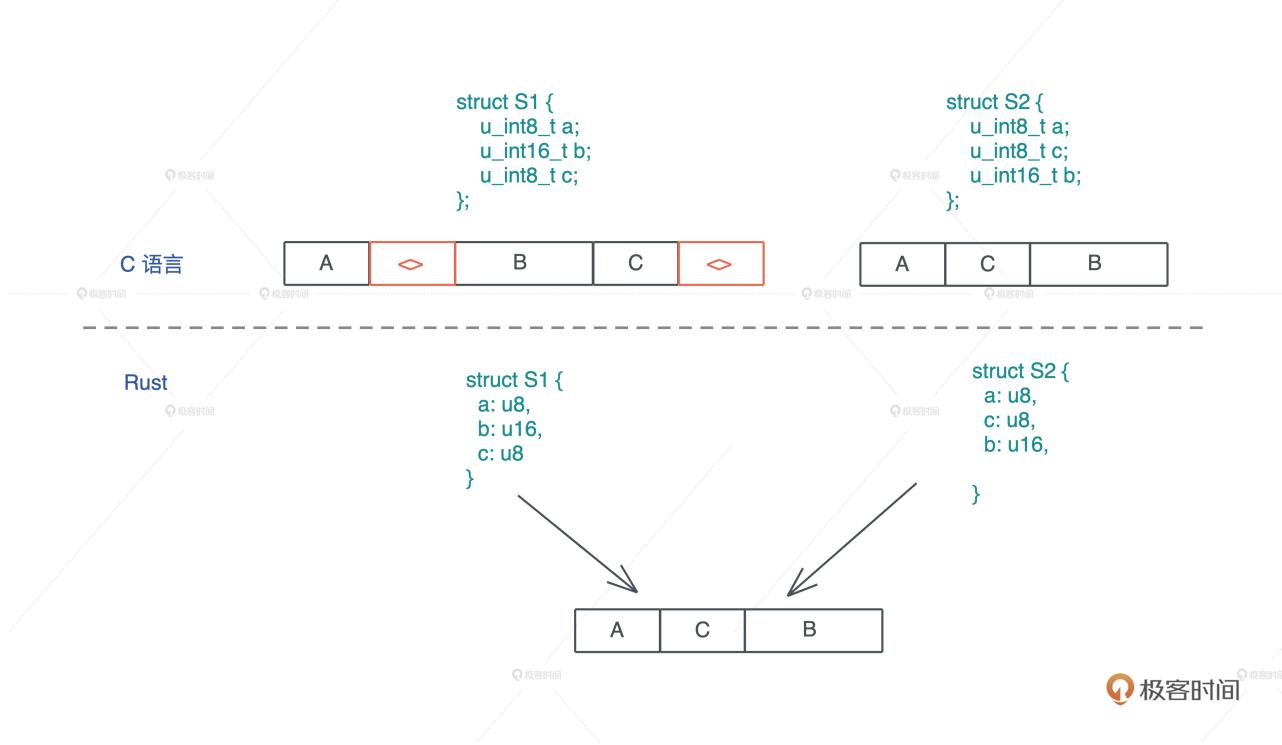
```
use std::mem::{align_of, size_of};

struct S1 {
    a: u8,
    b: u16,
    c: u8,
}

struct S2 {
    a: u8,
    c: u8,
    b: u16,
}

fn main() {
    println!("sizeof S1: {}", size_of::<S1>(), size_of::<S2>());
    println!("alignof S1: {}", align_of::<S1>(), align_of::<S2>());
}
```

你也可以看这张图来直观对比，C 和 Rust 的行为：



虽然，Rust 编译器默认认为开发者优化结构体的排列，但你也可以使用 `#[repr]` 宏，强制让 Rust 编译器不做优化，和 C 的行为一致，这样，Rust 代码可以方便地和 C 代码无缝交互。

在明白了 Rust 下 struct 的布局后（tuple 类似），我们看看 enum。

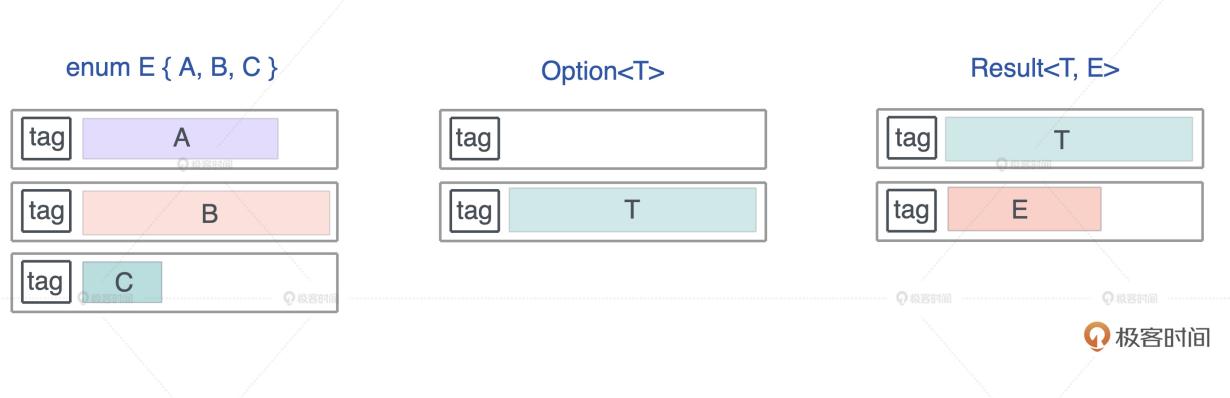
enum

enum 我们之前讲过，在 Rust 下它是一个标签联合体（tagged union），它的大小是标签的大小，加上最大类型的长度。

[第三讲](#)基础语法中，我们定义 enum 数据结构时，简单提到有 Option 和 Result<T, E> 两种设计举例，Option 是有值/无值这种最简单的枚举类型，Result 包括成功返回数据和错误返回数据的枚举类型，后面会详细讲到。这里我们理解其内存设计就可以了。

根据刚才说的三条对齐规则，tag 后的内存，会根据其对齐大小进行对齐，所以对于 Option，其长度是 $1 + 1 = 2$ 字节，而 Option，长度是 $8 + 8 = 16$ 字节。一般而言，64 位 CPU 下，enum 的最大长度是：最大类型的长度 + 8，因为 64 位 CPU 的最大对齐是 64bit，也就是 8 个字节。

下图展示了 enum、Option 以及 Result<T, E> 的布局：



值得注意的是，Rust 编译器会对 enum 做一些额外的优化，让某些常用结构的内存布局更紧凑。我们先来写一段代码，帮你更好地了解不同数据结构占用的大小（[代码](#)）：

```
use std::collections::HashMap;
use std::mem::size_of;

enum E {
    A(f64),
    B(HashMap<String, String>),
    C(Result<Vec<u8>, String>),
}

// 这是一个声明宏，它会打印各种数据结构本身的大小，在 Option 中的大小，以及在 Result 中的大小
macro_rules! show_size {
    (header) => {
        println!(

```

```

    ":{<24} {:>4}  {}  {}",
    "Type", "T", "Option<T>", "Result<T, io::Error>"
);
    println!("{}","-".repeat(64));
};

($t:ty) => {
    println!(
        ":{<24} {:4} {:8} {:12}",
        stringify!($t),
        size_of::<$t>(),
        size_of::<Option<$t>>(),
        size_of::<Result<$t, std::io::Error>>(),
    )
};

fn main() {
    show_size!(header);
    show_size!(u8);
    show_size!(f64);
    show_size!(&u8);
    show_size!(Box<u8>);
    show_size!(&[u8]);

    show_size!(String);
    show_size!(Vec<u8>);
    show_size!(HashMap<String, String>);
    show_size!(E);
}
}

```

这段代码包含了一个声明宏 (declarative macro) `show_size`, 我们先不必管它。运行这段代码时, 你会发现, `Option` 配合带有引用类型的数据结构, 比如 `&u8`、`Box`、`Vec`、`HashMap`, **没有额外占用空间**, 这就很有意思了。

Type
 Option<T>
 Result<T, io::Error>

u8
 f64
 &u8
 Box<u8>
 &[u8]
 String
 Vec<u8>
 HashMap<String, String>

对于 Option 结构而言，它的 tag 只有两种情况：0 或 1， tag 为 0 时，表示 None， tag 为 1 时，表示 Some。

正常来说，当我们把它和一个引用放在一起时，虽然 tag 只占 1 个 bit，但 64 位 CPU 下，引用结构的对齐是 8，所以它自己加上额外的 padding，会占据 8 个字节，一共16字节，这非常浪费内存。怎么办呢？

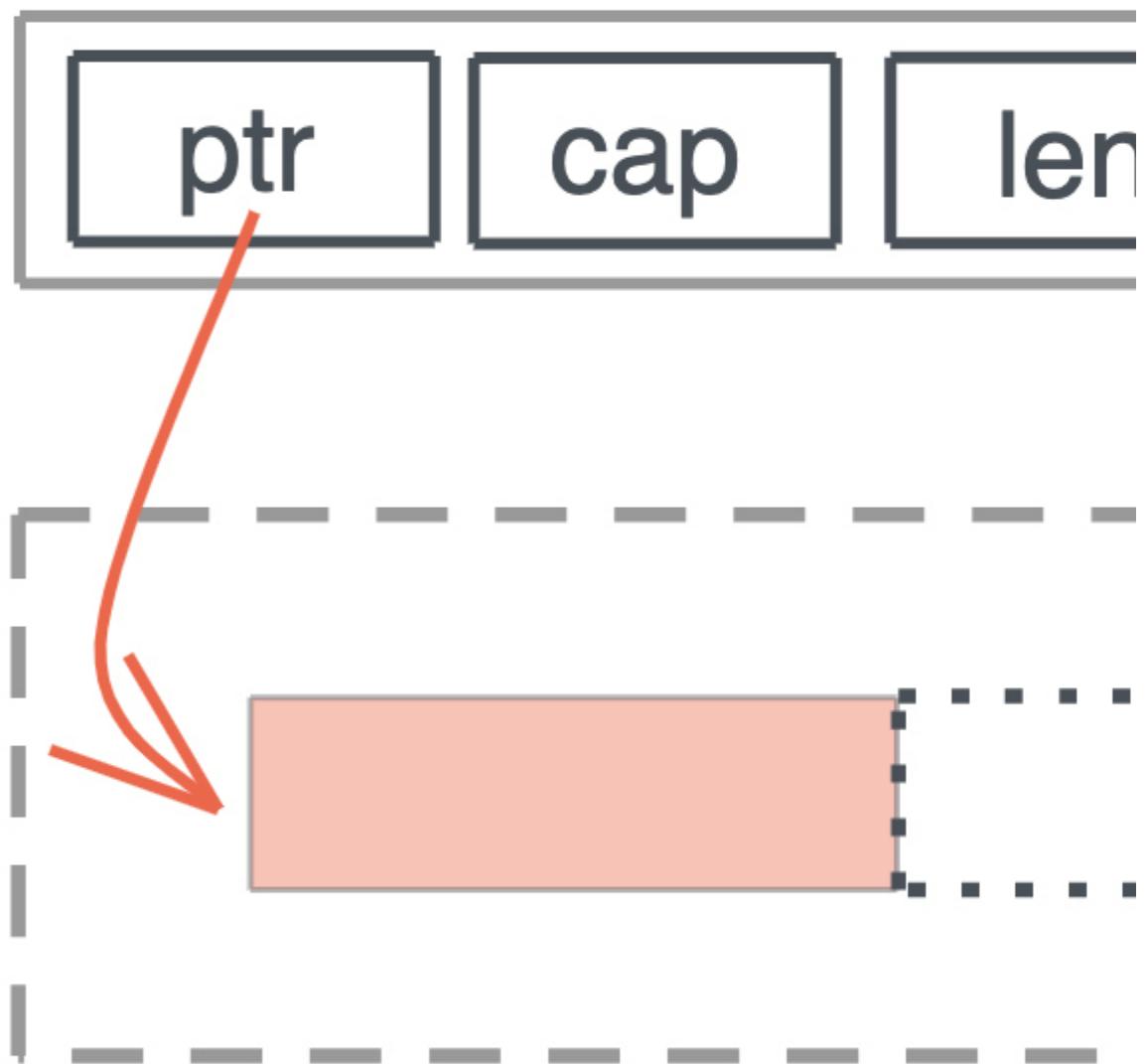
Rust 是这么处理的，我们知道，引用类型的第一个域是个指针，而指针是不可能等于 0 的，但是我们可以复用这个指针：当其为 0 时，表示 None，否则是 Some，减少了内存占用，这是个非常巧妙的优化，我们可以学习。

vec 和 String

从刚才代码的结果中，我们也看到 String 和 Vec 占用相同的大小，都是 24 个字节。其实，如果你打开 String 结构的[源码](#)，可以看到，它内部就是一个 Vec。

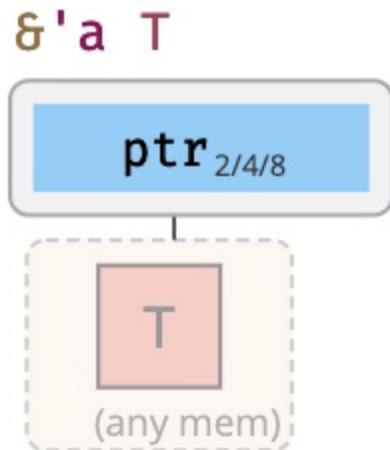
而 [Vec结构](#)是 3 个 word 的胖指针，包含：一个指向堆内存的指针pointer、分配的堆内存的容量capacity，以及数据在堆内存的长度length，如下图所示：

String(Vec<u8>)

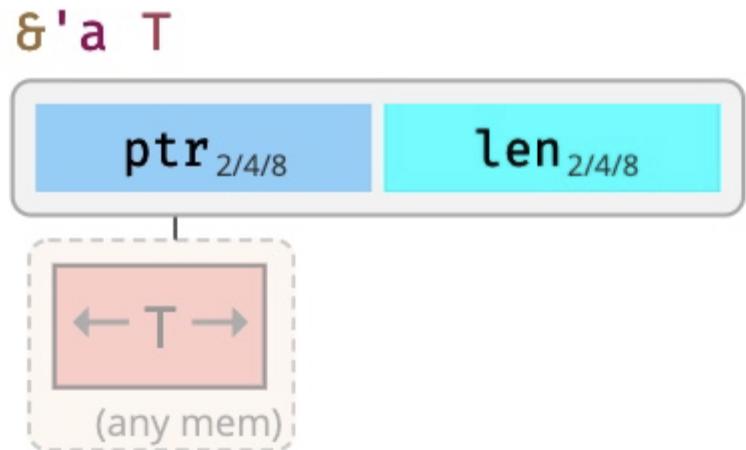


很多动态大小的数据结构，在创建时都有类似的内存布局：栈内存放的胖指针，指向堆内存分配出来的数据，我们之前介绍的 Rc 也是如此。

关于值在创建时的内存布局，今天就先讲这么多。如果你对其他数据结构的内存布局感兴趣，可以访问 [cheats.rs](#)，它是 Rust 语言的备忘清单，非常适合随时翻阅。比如，引用类型的内存布局：



No meta for
sized target.
(pointer is thin).



If `T` is a DST `struct` such as
`S { x: [u8] }` meta field `len` is
length of dyn. sized content.

现在，值已经创建成功了，我们对它的内存布局有了足够的认识。那在使用期间，它的内存会发生什么样的变化呢，我们接着看。

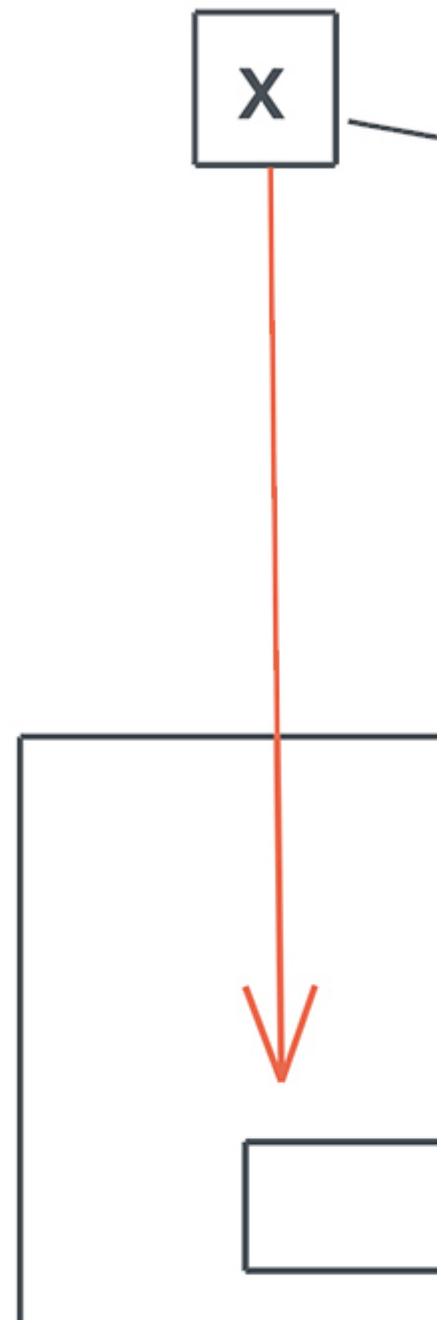
值的使用

在讲所有权的时候，我们知道，对 Rust 而言，一个值如果没有实现 Copy，在赋值、传参以及函数返回时会被 Move。

其实 Copy 和 Move 在内部实现上，都是浅层的按位做内存复制，只不过 Copy 允许你访问之前的变量，而 Move 不允许。我们看图：

代码

栈内存



堆内存

在我们的认知中，内存复制是个很重的操作，效率很低。确实是这样，如果你的关键路径中的每次调用，都要复制几百 k 的数据，比如一个大数组，是很低效的。

但是，如果你要复制的只是原生类型（Copy）或者栈上的胖指针（Move），不涉及堆内存的复制也就是深拷贝（deep copy），那这个效率是非常高的，我们不必担心每次赋值或者每次传参带来的性能损失。

所以，无论是 Copy 还是 Move，它的效率都是非常高的。

不过也有一个例外，要说明：对栈上的大数组传参，由于需要复制整个数组，会影响效率。所以，一般我们**建议在栈上不要放大数据**，如果实在需要，那么传递这个数组时，最好用传引用而不是传值。

在使用值的过程中，除了 Move，你还需要注意值的动态增长。因为Rust 下，集合类型的数据结构，都会在使用过程中自动扩展。

以一个 Vec 为例，当你使用完堆内存目前的容量后，还继续添加新的内容，就会触发堆内存的自动增长。有时候，集合类型里的数据不断进进出出，导致集合一直增长，但只使用了很小部分的容量，内存的使用效率很低，所以你要考虑使用，比如 `shrink_to_fit` 方法，来节约对内存的使用。

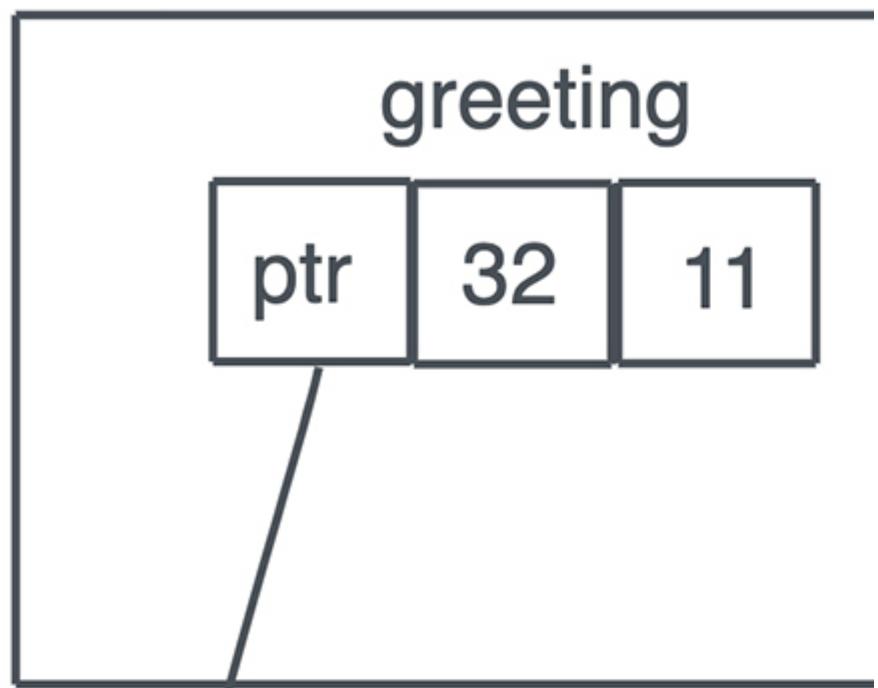
值的销毁

好，这个值的旅程已经过半，创建和使用都已经讲完了，最后我们谈谈值的销毁。

之前笼统地谈到，当所有者离开作用域，它拥有的值会被丢弃。那从代码层面讲，Rust 到底是如何丢弃的呢？

这里用到了 Drop trait。Drop trait 类似面向对象编程中的析构函数，当一个值要被释放，它的 Drop trait 会被调用。比如下面的代码，变量 greeting 是一个字符串，在退出作用域时，其 drop() 函数被自动调用，释放堆上包含“hello world”的内存，然后再释放栈上的内存：

栈



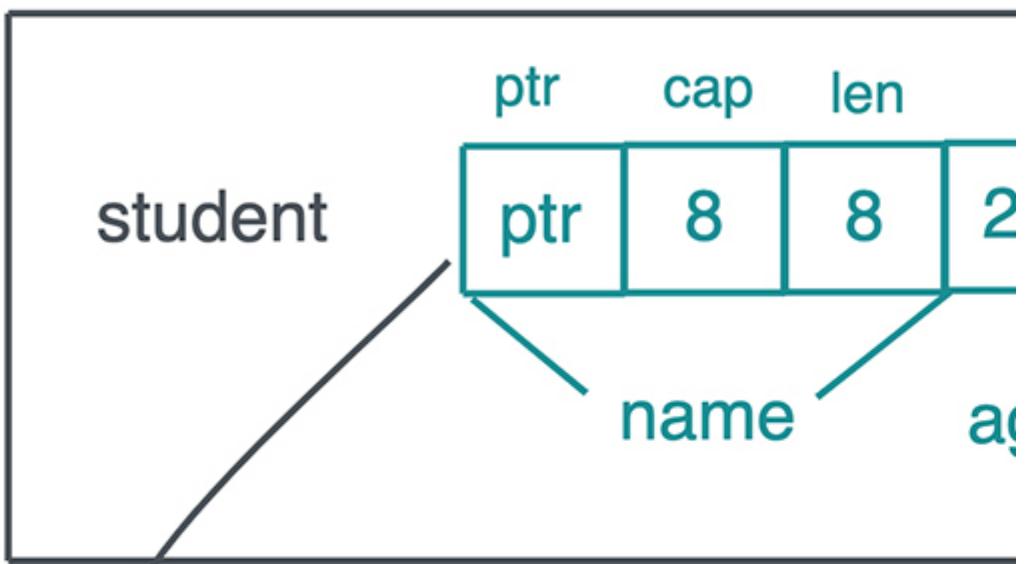
堆



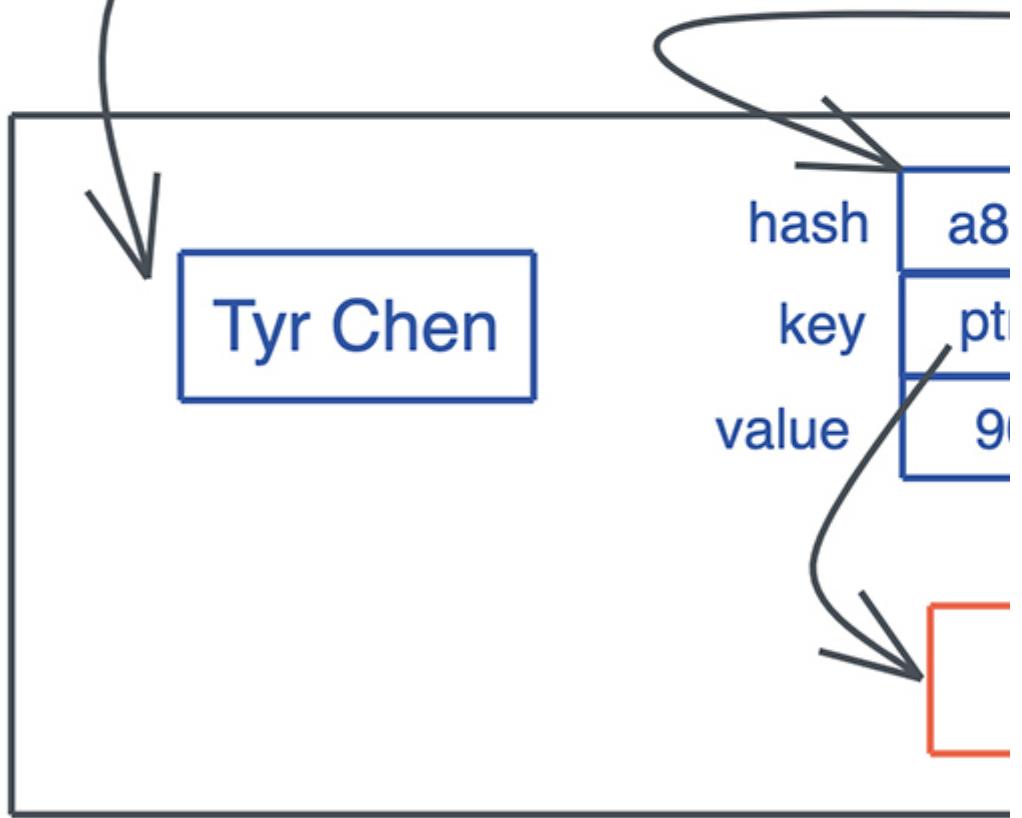
如果要释放的值是一个复杂的数据结构，比如一个结构体，那么这个结构体在调用 drop() 时，会依次调用每一个域的 drop() 函数，如果域又是一个复杂的结构或者集合类型，就会递归下去，直到每一个域都释放干净。

我们可以看这个例子：

栈



堆



释放顺序：①

代码中的 student 变量是一个结构体，有 name、age、scores。其中 name 是 String，scores 是 HashMap，它们本身需要额外 drop()。又因为 HashMap 的 key 是 String，所以还需要进一步调用这些 key 的 drop()。整个释放顺序从内到外是：先释放 HashMap 下的 key，然后释放 HashMap 堆上的表结构，最后释放栈上的内存。

堆内存释放

所有权机制规定了，一个值只能有一个所有者，所以在释放堆内存的时候，整个过程简单清晰，就是单纯调用 Drop trait，不需要有其他顾虑。这种对值安全，也没有额外负担的释放能力，是 Rust 独有的。

我觉得 Rust 在内存管理方面的设计特别像蚁群。在蚁群中，每个个体的行为都遵循着非常简单死板的规范，最终，**大量简单的个体能构造出一个高效且不出错的系统**。

反观其它语言，每个个体或者说值，都非常灵活，引用传来传去，最终却构造出来一个很难分析的复杂系统。单靠编译器无法决定，每个值在各个作用域中究竟能不能安全地释放，导致系统，要么像 C/C++ 一样将这个重担部分或者全部地交给开发者，要么像 Java 那样构建另一个系统来专门应对内存安全释放的问题。

在Rust里，你自定义的数据结构，绝大多数情况下，不需要实现自己的 Drop trait，编译器缺省的行为就足够了。但是，如果你想自己控制 drop 行为，你也可以为这些数据结构实现它。

如果你定义的 drop() 函数和系统自定义的 drop() 函数都 drop() 某个域，Rust 编译器会确保，这个域只会被 drop 一次。至于 Drop trait 怎么实现、有什么注意事项、什么场合下需要自定义，我们在后续的课程中会再详细展开。

释放其他资源

我们刚才讲 Rust 的 Drop trait 主要是为了应对堆内存释放的问题，其实，它还可以释放任何资源，比如 socket、文件、锁等等。Rust 对所有的资源都有很好的 [RAII 支持](#)。

比如我们创建一个文件 file，往里面写入“hello world”，当 file 离开作用域时，不但它的内存会被释放，它占用的资源、操作系统打开的文件描述符，也会被释放，也就是文件会自动被关闭。[\(代码\)](#)

```
use std::fs::File;
use std::io::prelude::*;
fn main() -> std::io::Result<()> {
    let mut file = File::create("foo.txt")?;
    file.write_all(b"hello world")?;
    Ok(())
}
```

在其他语言中，无论 Java、Python 还是 Golang，你都需要显式地关闭文件，避免资源的泄露。这是因为，即便 GC 能够帮助开发者最终释放不再引用的内存，它并不能释放除内存外的其它资源。

而 Rust，再一次地，因为其清晰的所有权界定，使编译器清楚地知道：当一个值离开作用域的时候，这个值不会有任何人引用，它占用的任何资源，包括内存资源，都可以立即释放，而不会导致问题（也有例外，感兴趣可以看这个 [RFC](#)）。

说到这，你也许觉得不用显式地关闭文件、关闭 socket、释放锁，不过是省了一句“close()”而已，有什么大不了的？

然而，不要忘了，在庞大的业务代码中，还有很大一部分要用来处理错误。当错误处理搅和进来，我们面对的代码，逻辑更复杂，需要添加 close() 调用的上下文更多。虽然 Python 的 with、Golang 的 defer，可以一定程度上解决资源释放的问题，但还不够完美。

一旦，多个变量和多种异常或者错误叠加，我们忘记释放资源的风险敞口会成倍增加，很多死锁或者资源泄露就是这么产生的。

从 Drop trait 中我们再一次看到，从事物的本原出发解决问题，会极其优雅地解决掉很多其他关联问题。好比，所有权，几个简单规则，就让我们顺带处理掉了资源释放的大难题。

小结

我们进一步探讨了 Rust 的内存管理，在所有权和生命周期管理的基础上，介绍了一个值在内存中创建、使用和销毁的过程，学习了数据结构在创建时，是如何在内存中布局的，大小和对齐之间的关系；数据在使用过程中，是如何 Move 和自动增长的；以及数据是如何销毁的。

检查时间

检查效果

高
效

检查位置

检查机制

bor
der

数据结构在内存中的布局，尤其是哪些部分放在栈上，哪些部分放在堆上，非常有助于我们理解代码的结构和效率。

你不必强行记忆这些内容，只要有个思路，在需要的时候，翻阅本文或者 [cheats.rs](#) 即可。当我们掌握了数据结构如何创建、在使用过程中如何 Move 或者 Copy、最后如何销毁，我们在阅读别人的代码或者自己撰写代码时就会更加游刃有余。

思考题

`Result<String, ()>` 占用多少内存？为什么？

感谢你的收听，如果你觉得有收获，也欢迎你分享给你身边的朋友，邀他一起讨论。你的Rust学习第11次打卡完成，我们下节课见。

参考资料

1. Rust 语言的备忘清单 [cheats.rs](#)
2. 代码受这个[Stack Overflow 帖子](#)启发，有删改
3. String 结构的[源码](#)
4. Vec 结构[源码](#)
5. [RAII](#) 是一个拗口的名词，中文意思是“资源获取即初始化”。

类型系统

类型系统

01 | 类型系统：Rust的类型系统有什么特点？

类型系统完全是一种工具，编译器在编译时对数据做静态检查，或者语言在运行时对数据做动态检查的时候，来保证某个操作处理的数据是开发者期望的数据类型。

类型系统基本概念与分类

类型系统其实就是，对类型进行定义、检查和处理的系统。按定义后类型是否可以隐式转换，可以分为强类型和弱类型。Rust 不同类型间不能自动转换，所以是强类型语言，而 C / C++ / JavaScript 会自动转换，是弱类型语言。按类型检查的时机，在编译时检查还是运行时检查，可以分为静态类型系统和动态类型系统。对于静态类型系统，还可以进一步分为显式静态和隐式静态，Rust / Java / Swift 等语言都是显式静态语言，而 Haskell 是隐式静态语言。

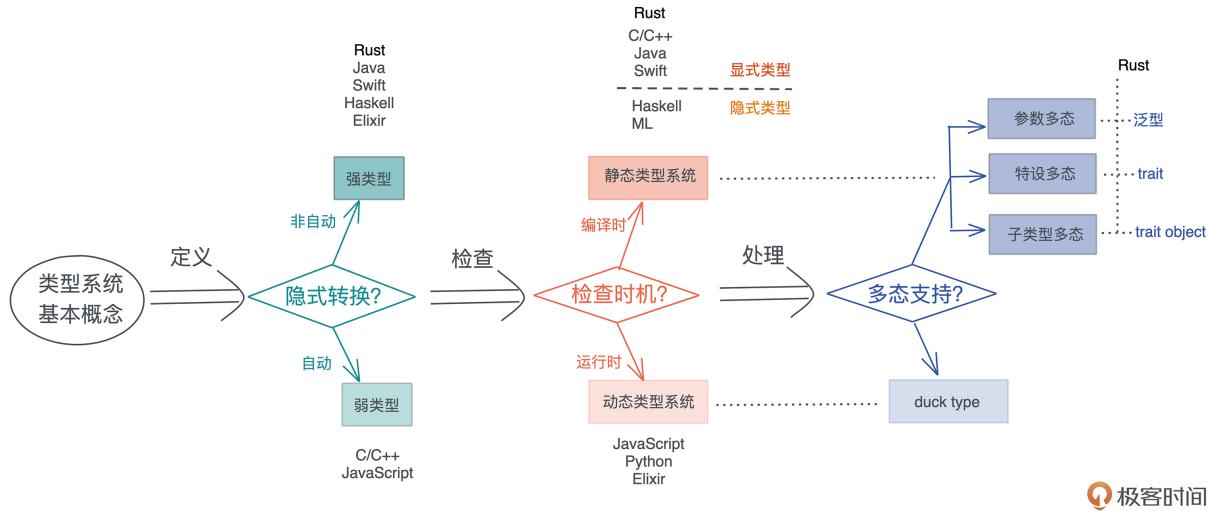
在类型系统中，多态是一个非常重要的思想，它是指在使用相同的接口时，不同类型的对象，会采用不同的实现。

对于动态类型系统，多态通过**鸭子类型**（duck typing）实现；而对于静态类型系统，多态可以通过**参数多态**（parametric polymorphism）、**特设多态**（adhoc polymorphism）和**子类型多态**（subtype polymorphism）实现。

- 参数多态是指，代码操作的类型是一个满足某些约束的参数，而非具体的类型。
- 特设多态是指同一种行为有多个不同实现的多态。比如加法，可以 $1+1$ ，也可以是“abc” + “cde”、matrix1 + matrix2、甚至 matrix1 + vector1。在面向对象编程语言中，特设多态一般指函数的重载。
- 子类型多态是指，在运行时，子类型可以被当成父类型使用。

在 Rust 中，参数多态通过泛型来支持、特设多态通过 trait 来支持、子类型多态可以用 trait object 来支持，我们待会讲参数多态，下节课再详细讲另外两个。

你可以看下图来更好地厘清这些概念之间的关系：



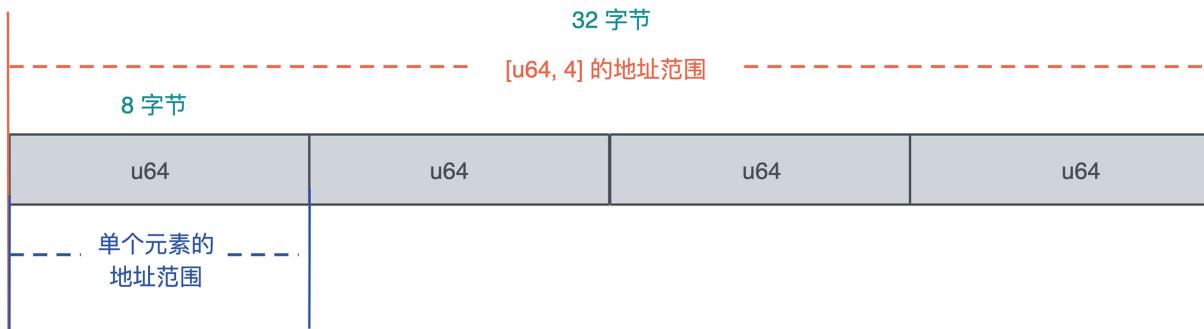
Rust 类型系统

好，掌握了类型系统的基本概念和分类，再看 Rust 的类型系统。

从内存的角度看，**类型安全**是指代码，只能按照被允许的方法，访问它被授权访问的内存。

以一个长度为 4，存放 u64 数据的数组为例，访问这个数组的代码，只能在这个数组的起始地址到数组的结束地址之间这片 32 个字节的内存中访问，而且访问是按照 8 字节来对齐的，另外，数组中的每个元素，只能做 u64 类型允许的操作。

作。对此，编译器会对代码进行严格检查来保证这个行为。我们看下图：



所以 C/C++ 这样，定义后数据可以隐式转换类型的弱类型语言，不是内存安全的，而 Rust 这样的强类型语言，是类型安全的，不会出现开发者不小心引入了一个隐式转换，导致读取不正确的数据，甚至内存访问越界的问题。

在此基础上，Rust 还进一步对内存的访问进行了读/写分开的授权。所以，**Rust 下的内存安全更严格：代码只能按照被允许的方法和被允许的权限，访问它被授权访问的内存。**

为了做到这么严格的类型安全，Rust 中除了 `let` / `fn` / `static` / `const` 这些定义性语句外，都是表达式，而一切表达式都有类型，所以说在 Rust 中，类型无处不在。

你也许会有疑问，那类似这样的代码，它的类型是什么？

```
if has_work {  
    do_something();  
}
```

在Rust中，对于一个作用域，无论是 `if` / `else` / `for` 循环，还是函数，最后一个表达式的返回值就是作用域的返回值，如果表达式或者函数不返回任何值，那么它返回一个 `unit ()`。`unit` 是只有一个值的类型，它的值和类型都是 `()`。

像上面这个 `if` 块，它的类型和返回值是 `()`，所以当它被放在一个没有返回值的函数中，如下所示：

```
fn work(has_work: bool) {  
    if has_work {
```

```
    do_something();  
}  
}
```

Rust 类型无处不在这个逻辑还是自洽的。

到这里简单总结一下，我们了解到 Rust 是强类型/静态类型语言，并且在代码中，类型无处不在。

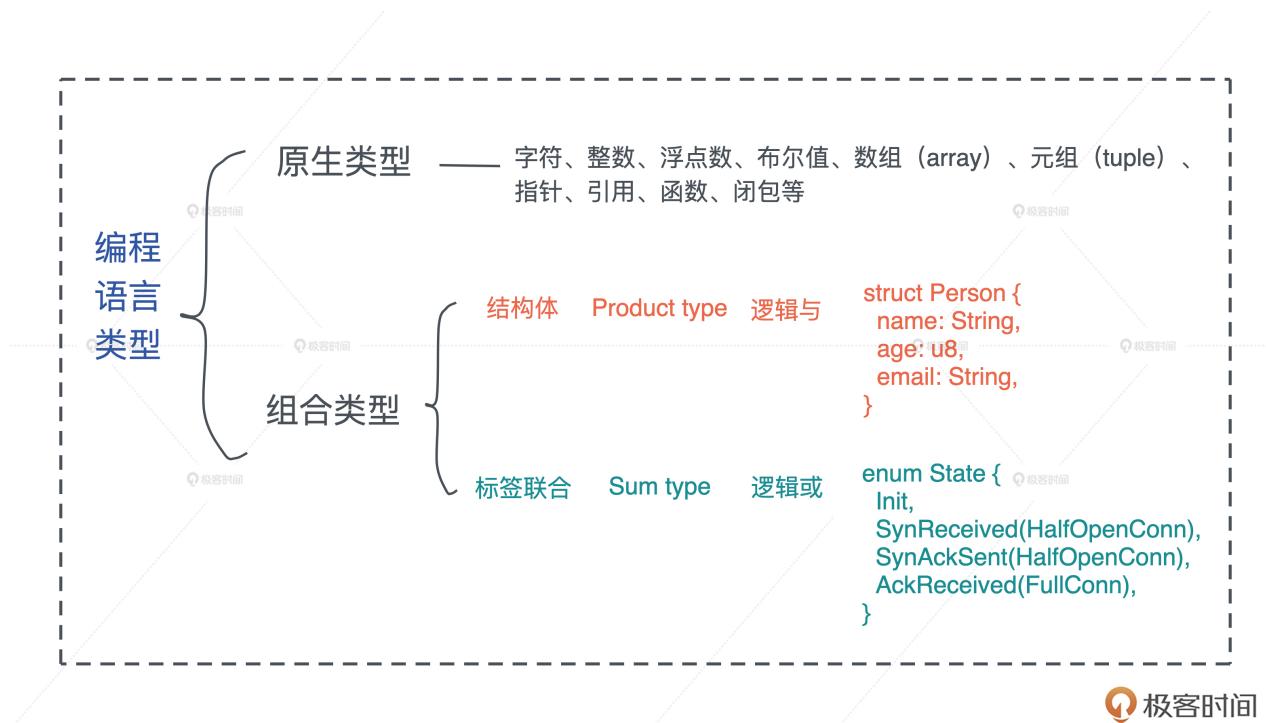
作为静态类型语言，Rust 提供了大量的数据类型，但是在使用的过程中，进行类型标注是很费劲的，所以Rust 类型系统贴心地提供了**类型推导**。

而对比动态类型系统，静态类型系统还比较麻烦的是，同一个算法，对应输入的数据结构不同，需要有不同的实现，哪怕这些实现没有什么逻辑上的差异。对此，Rust 给出的答案是**泛型（参数多态）**。

所以接下来，我们先看 Rust 有哪些基本的数据类型，然后了解一下类型推导是如何完成的，最后看 Rust 是如何支持泛型的。

数据类型

在第二讲中介绍了原生类型和组合类型的定义，今天就详细介绍一下这两种类型在 Rust 中的设计。



类型推导

作为静态类型系统的语言，虽然能够在编译期保证类型的安全，但Rust 的数据类型相当多，所以，为了减轻开发者的负担，Rust 支持局部的类型推导。

在一个作用域之内，Rust 可以根据变量使用的上下文，推导出变量的类型，这样我们就不需要显式地进行类型标注了。比如这段[代码](#)，创建一个 BTreeMap 后，往这个 map 里添加了 key 为 “hello”、value 为 “world” 的值：

```
use std::collections::BTreeMap;

fn main() {
    let mut map = BTreeMap::new();
    map.insert("hello", "world");
    println!("map: {:?}", map);
}
```

此时，Rust 编译器可以从上下文中推导出，`BTreeMap<K, V>` 的类型 K 和 V 都是字符串引用 `&str`，所以这段代码可以编译通过，然而，如果你把第 5 行这个作用域内的 `insert` 语句注释掉，Rust 编译器就会报错：“cannot infer type for type parameter `K`”。

很明显，Rust 编译器需要足够的上下文来进行类型推导，所以有些情况下，编译器无法推导出合适的类型，比如下面的代码尝试把一个列表中的偶数过滤出来，生成一个新的列表（[代码](#)）：

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    let even_numbers = numbers
        .into_iter()
        .filter(|n| n % 2 == 0)
        .collect();

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

`collect` 是[Iterator trait 的方法](#)，它把一个 iterator 转换成一个集合。因为很多集合类型，如 `Vec`、`HashMap<K, V>` 等都实现了 `Iterator`，所以这里的 `collect` 究竟要返回什么类型，编译器是无法从上下文中推断的。

所以这段代码无法编译，它会给出如下错误：“consider giving `even_numbers` a type”。

这种情况，就无法依赖类型推导来简化代码了，必须让 `even_numbers` 有一个明确的类型。所以，我们可以使用类型声明（[代码](#)）：

```

fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    let even_numbers: Vec<_> = numbers
        .into_iter()
        .filter(|n| n % 2 == 0)
        .collect();

    println!("{:?}", even_numbers);
}

```

注意这里编译器只是无法推断出集合类型，但集合类型内部元素的类型，还是可以根据上下文得出，所以我们可以简写成 `Vec<_>`。

除了给变量一个显式的类型外，我们也可以让 `collect` 返回一个明确的类型（[代码](#)）：

```

fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    let even_numbers = numbers
        .into_iter()
        .filter(|n| n % 2 == 0)
        .collect::<Vec<_>>();

    println!("{:?}", even_numbers);
}

```

你可以看到，在泛型函数后使用 `::<T>` 来强制使用类型 `T`，这种写法被称为 `turbofish`。我们再看一个对 IP 地址和端口转换的例子（[代码](#)）：

```

use std::net::SocketAddr;

fn main() {
    let addr = "127.0.0.1:8080".parse::<SocketAddr>().unwrap();
    println!("addr: {:?}, port: {:?}", addr.ip(), addr.port());
}

```

`turbofish` 的写法在很多场景都有优势，因为在某些上下文中，你想直接把一个表达式传递给一个函数或者当成一个作用域的返回值，比如：

```

match data {
    Some(s) => v.parse::<User>()?,
    _ => return Err(...),
}

```

如果 User 类型在上下文无法被推导出来，又没有 turbofish 的写法，我们就不得不先给一个局部变量赋值时声明类型，然后再返回，这样代码就变得冗余了。

有些情况下，即使上下文中含有类型的信息，也需要开发者为变量提供类型，比如常量和静态变量的定义。看一个例子（[代码](#)）：

```
const PI: f64 = 3.1415926;
static E: f32 = 2.71828;

fn main() {
    const V: u32 = 10;
    static V1: &str = "hello";
    println!("PI: {}, E: {}, V {}, V1: {}", PI, E, V, V1);
}
```

这可能是因为 const / static 主要用于定义全局变量，它们可以在不同的上下文中使用，所以为了代码的可读性，需要明确的类型声明。

用泛型实现参数多态

类型的定义和使用就讲到这里，刚才说过 Rust 通过泛型，来避免开发者为不同的类型提供不同的算法。一门静态类型语言不支持泛型，用起来是很痛苦的，比如我们熟悉的 Vec，你能想像不支持泛型时，每一个类型 T，都要实现一遍 Vec 么？太麻烦了。

所以我们现在来看看 Rust 对泛型的支持如何。今天先讲参数多态，它包括泛型数据结构和泛型函数，下一讲介绍特设多态和子类型多态。

泛型数据结构

Rust 对数据结构的泛型，或者说参数化类型，有着完整的支持。

在过去的学习中，其实你已经接触到了很多带有参数的数据类型，这些参数化类型可以极大地增强代码的复用性，减少代码的冗余。几乎所有支持静态类型系统的现代编程语言，都支持参数化类型，不过 [Golang 目前是个例外](#)。

我们从一个最简单的泛型例子 Option开始回顾：

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

这个数据结构你应该很熟悉了，`T` 代表任意类型，当 `Option` 有值时是 `Some(T)`，否则是 `None`。

在定义刚才这个泛型数据结构的时候，你有没有这样的感觉，有点像在定义函数：

- 函数，**是把重复代码中的参数抽取出来**，使其更加通用，调用函数的时候，根据参数的不同，我们得到不同的结果；
- 而泛型，**是把重复数据结构中的参数抽取出来**，在使用泛型类型时，根据不同的参数，我们会得到不同的具体类型。

再来看一个复杂一点的泛型结构 `Vec` 的例子，验证一下这个想法：

```
pub struct Vec<T, A: Allocator = Global> {
    buf: RawVec<T, A>,
    len: usize,
}

pub struct RawVec<T, A: Allocator = Global> {
    ptr: Unique<T>,
    cap: usize,
    alloc: A,
}
```

`Vec` 有两个参数，一个是 `T`，是列表里的每个数据的类型，另一个是 `A`，它有进一步的限制 `A: Allocator`，也就是说 `A` 需要满足 `Allocator trait`。

`A` 这个参数有默认值 `Global`，它是 [Rust 默认的全局分配器](#)，这也是为什么 `Vec` 虽然有两个参数，使用时都只需要用 `T`。

在讲生命周期标注的时候，我们讲过，数据类型内部如果有借用的数据，需要显式地标注生命周期。其实在 Rust 里，**生命周期标注也是泛型的一部分**，一个生命周期 `'a` 代表任意的生命周期，和 `T` 代表任意类型是一样的。

来看一个枚举类型 `Cow` 的例子：

```
pub enum Cow<'a, B: ?Sized + 'a> where B: ToOwned,  
{  
    // 借用的数据  
    Borrowed(&'a B),  
    // 拥有的数据  
    Owned(<B as ToOwned>::Owned),  
}
```

Cow (Clone-on-Write) 是Rust中一个很有意思且很重要的数据结构。它就像 Option 一样，在返回数据的时候，提供了一种可能：要么返回一个借用的数据（只读），要么返回一个拥有所有权的数据（可写）。

这里你搞清楚泛型参数的约束就可以了，未来还会遇到 Cow，届时再详细讲它的用法。

对于拥有所有权的数据 B，第一个是生命周期约束。这里 B 的生命周期是 ''a'，所以 B 需要满足 ''a'，这里和泛型约束一样，也是用 `B: 'a` 来表示。当 Cow 内部的类型 B 生命周期为 ''a' 时，Cow 自己的生命周期也是 ''a'。

B 还有两个约束：`?Sized` 和“where B: `ToOwned`”。

在表述泛型参数的约束时，Rust 允许两种方式，一种类似函数参数的类型声明，用 “:” 来表明约束，多个约束之间用 + 来表示；另一种是使用 where 子句，在定义的结尾来表明参数的约束。两种方法都可以，且可以共存。

`?Sized` 是一种特殊的约束写法，? 代表可以放松问号之后的约束。由于 Rust 默认的泛型参数都需要是 `Sized`，也就是固定大小的类型，所以这里 `?Sized` 代表用可变大小的类型。

`ToOwned` 是一个 trait，它可以把借用的数据克隆出一个拥有所有权的数据。

所以这里对 B 的三个约束分别是：

- 生命周期 ''a'
- 长度可变 `?Sized`
- 符合 `ToOwned` trait

最后我解释一下 Cow 这个 enum 里 `<B as ToOwned>::Owned` 的含义：它对 B 做了一个强制类型转换，转成 `ToOwned` trait，然后访问 `ToOwned` trait 内部的 `Owned` 类型。

因为在 Rust 里，子类型可以强制转换成父类型，B 可以用 `ToOwned` 约束，所以它是 `ToOwned` trait 的子类型，因而 B 可以安全地强制转换成 `ToOwned`。这里 B as `ToOwned` 是成立的。

上面 Vec 和 Cow 的例子中，泛型参数的约束都发生在开头 struct 或者 enum 的定义中，其实，很多时候，我们也可以在不同的实现下逐步添加约束，比如下面这个例子（[代码](#)）：

```
use std::fs::File;
use std::io::{BufReader, Read, Result};

// 定义一个带有泛型参数 R 的 reader，此处我们不限制 R
struct MyReader<R> {
    reader: R,
    buf: String,
}

// 实现 new 函数时，我们不需要限制 R
impl<R> MyReader<R> {
    pub fn new(reader: R) -> Self {
        Self {
            reader,
            buf: String::with_capacity(1024),
        }
    }
}

// 定义 process 时，我们需要用到 R 的方法，此时我们限制 R 必须实现 Read trait
impl<R> MyReader<R>
where
    R: Read,
{
    pub fn process(&mut self) -> Result<usize> {
        self.reader.read_to_string(&mut self.buf)
    }
}

fn main() {
    // 在 windows 下，你需要换个文件读取，否则会出错
    let f = File::open("/etc/hosts").unwrap();
    let mut reader = MyReader::new(BufReader::new(f));

    let size = reader.process().unwrap();
    println!("total size read: {}", size);
}
```

逐步添加约束，可以让约束只出现在它不得不出现的地方，这样代码的灵活性最大。

泛型函数

了解了泛型数据结构是如何定义和使用的，再来看泛型函数，它们的思想类似。在声明一个函数的时候，我们还可以不指定具体的参数或返回值的类型，而是由泛型参数来代替。对函数而言，这是更高阶的抽象。

一个简单的例子 ([代码](#)) :

```
fn id<T>(x: T) -> T {
    return x;
}

fn main() {
    let int = id(10);
    let string = id("Tyr");
    println!("{} , {}", int, string);
}
```

这里，`id()` 是一个泛型函数，它接受一个带有泛型类型的参数，返回一个泛型类型。

对于泛型函数，Rust 会进行单态化 (Monomorphization) 处理，也就是在编译时，把所有用到的泛型函数的泛型参数展开，生成若干个函数。所以，刚才的 `id()` 编译后会得到一个处理后的多个版本 ([代码](#)) :

```
fn id_i32(x: i32) -> i32 {
    return x;
}
fn id_str(x: &str) -> &str {
    return x;
}
fn main() {
    let int = id_i32(42);
    let string = id_str("Tyr");
    println!("{} , {}", int, string);
}
```

单态化的好处是，泛型函数的调用是静态分派 (static dispatch)，在编译时就一一对应，既保有多态的灵活性，又没有任何效率的损失，和普通函数调用一样高效。

但是对比刚才编译会展开的代码也能很清楚看出来，单态化有很明显的坏处，就是编译速度很慢，**一个泛型函数，编译器需要找到所有用到的不同类型，一个个编译**，所以 Rust 编译代码的速度总被人吐槽，这和单态化脱不开干系（另一个重要因素是宏）。

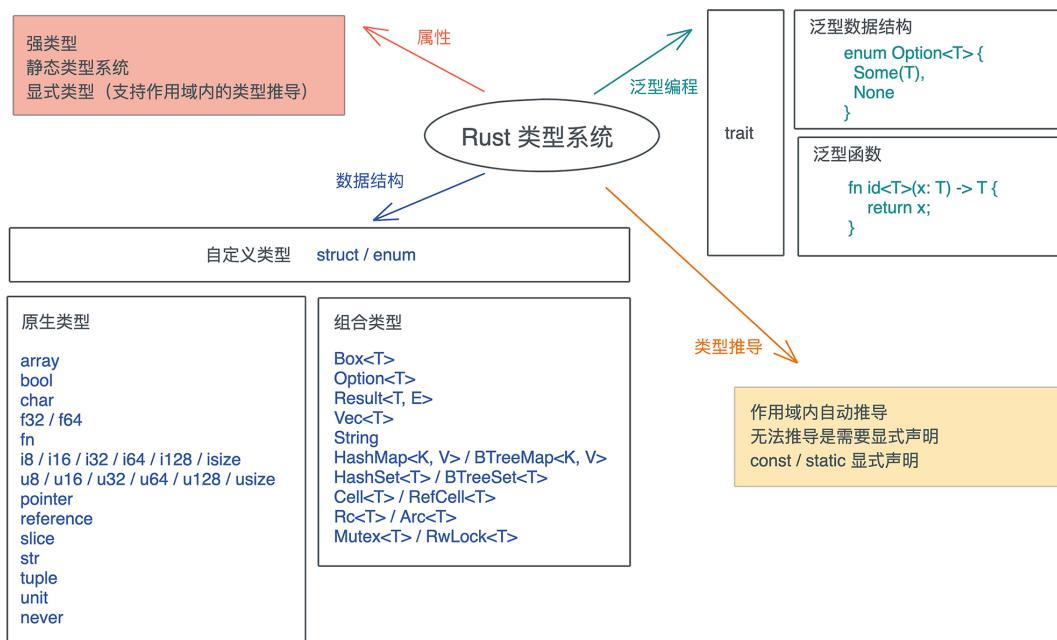
同时，这样编出来的二进制会比较大，因为泛型函数的二进制代码实际存在 N 份。

还有一个可能你不怎么注意的问题：**因为单态化，代码以二进制分发会损失泛型的信息**。如果我写了一个库，提供了如上的 `id()` 函数，使用这个库的开发者如果拿到的是二进制，那么这个二进制中必须带有原始的泛型函数，才能正确调用。但单态化之后，原本的泛型信息就被丢弃了。

小结

今天我们介绍了类型系统的一些基本概念以及 Rust 的类型系统。

用一张图描述了 Rust 类型系统的主要特征，包括其属性、数据结构、类型推导和泛型编程：



极客时间

按类型定义、检查以及检查时能否被推导出来，Rust 是**强类型+静态类型+显式类型**。

因为是静态类型，那么在写代码时常用的类型你需要牢牢掌握。为了避免静态类型要到处做类型标注的繁琐，Rust 提供了类型推导。

在少数情况下，Rust 无法通过上下文进行类型推导，我们需要为变量显式地标注类型，或者通过 turbofish 语法，为泛型函数提供一个确定的类型。有个例外是在 Rust 代码中定义常量或者静态变量时，即使上下文中类型信息非常明确，也需要显式地进行类型标注。

在参数多态上，Rust 提供有完善支持的泛型。你可以使用和定义**泛型数据结构**，在声明一个函数的时候，也可以不指定具体的参数或返回值的类型，而是由泛型参数来代替，也就是**泛型函数**。它们的思想其实差不多，因为当数据结构可以泛型时，函数自然也就需要支持泛型。

另外，生命周期标注其实也是泛型的一部分，而对于泛型函数，在编译时会被单态化，导致编译速度慢。

下一讲我们接着介绍特设多态和子类型多态……

参考资料

1.绝大多数支持静态类型系统的语言同时也会支持动态类型系统，因为单纯靠静态类型无法支持运行时的类型转换，比如[里氏替换原则](#)。

里氏替换原则简单说就是子类型对象可以在程序中代替父类型对象。它是运行时多态的基础。所以如果要支持运行时多态，以及动态分派、后期绑定、反射等功能，编程语言需要支持动态类型系统。

2.动态类型系统的缺点是没有编译期的类型检查，程序不够安全，只能通过大量的单元测试来保证代码的健壮性。但使用动态类型系统的程序容易撰写，不用花费大量的时间来抠数据结构或者函数的类型。

所以一般用在脚本语言中，如 JavaScript / Python / Elixir。不过因为这些脚本语言越来越被用在大型项目中，所以它们也都有各自的类型标注的方法，来提供编译时的额外检查。

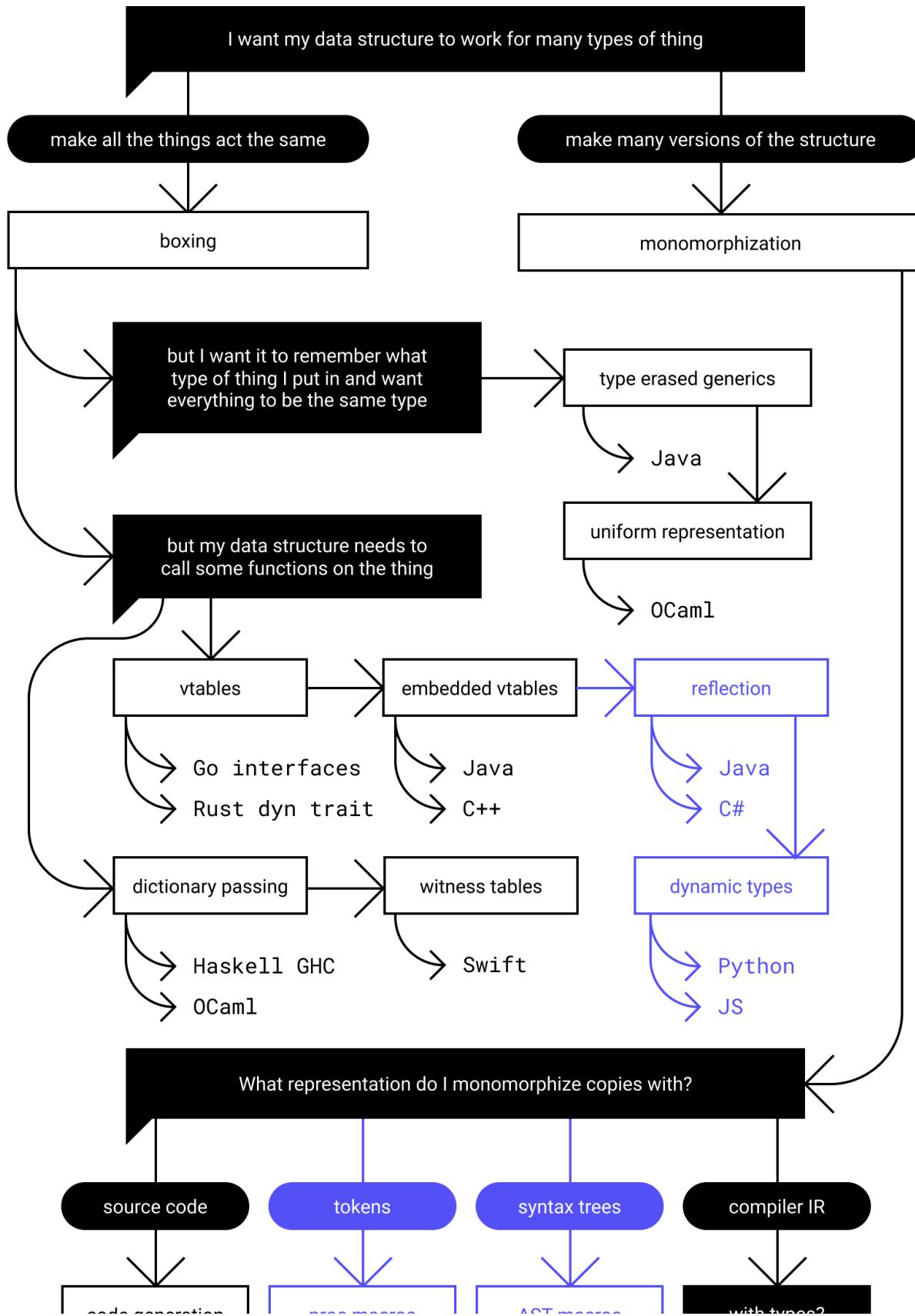
3.为了语言的简单易懂，编译高效，Golang 在设计之初没有支持泛型，但未来在[Golang 2 中也许会添加泛型](#)。

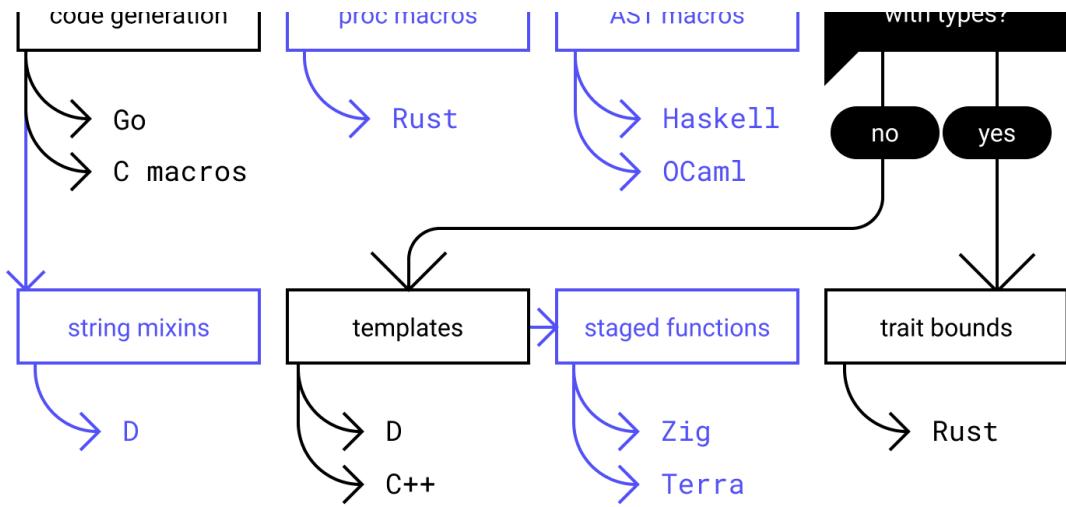
4.当我们在堆上分配内存的时候，我们通过分配器来进行内存的分配，以及管理已分配的内存，包括增大（grow）、缩小（shrink）等。在处理某些情况下，默认的分配器也许不够高效，我们可以使用[jemalloc 来分配内存](#)。

5.如果你对各个语言是如何实现和处理泛型比较感兴趣的话，可以参考下图（[来源](#)）：

How Languages Implement Generics and Extensions to Metaprogramming

<http://thume.ca/>





02 | 类型系统：如何使用 trait 来定义接口？

什么是 trait？

trait 是 Rust 中的接口，它定义了类型使用这个接口的行为。你可以类比到自己熟悉的语言中理解，trait 对于 Rust 而言，相当于 interface 之于 Java、protocol 之于 Swift、type class 之于 Haskell。

在开发复杂系统的时候，我们常常会强调接口和实现要分离。因为这是一种良好的设计习惯，它把调用者和实现者隔离开，双方只要按照接口开发，彼此就可以不受对方内部改动的影响。

trait 就是这样。它可以把数据结构中的行为单独抽取出来，使其可以在多个类型之间共享；也可以作为约束，在泛型编程中，限制参数化类型必须符合它规定的行为。

基本 trait

我们来看看基本 trait 如何定义。这里，以标准库中 `std::io::Write` 为例，可以看到这个 trait 中定义了一系列方法的接口：

```

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize> { ... }
    fn is_write_vectored(&self) -> bool { ... }
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) -> Result<()> { ... }
    fn write_fmt(&mut self, fmt: Arguments<'_>) -> Result<()> { ... }
    fn by_ref(&mut self) -> &mut Self where Self: Sized { ... }
}

```

这些方法也被称作关联函数（associate function）。在 trait 中，方法可以有缺省的实现，对于这个 Write trait，你只需要实现 write 和 flush 两个方法，其他都有缺省实现。

如果你把 trait 类比为父类，实现 trait 的类型类比为子类，那么缺省实现的方法就相当于子类中可以重载但不是必须重载的方法。

在刚才定义方法的时候，我们频繁看到两个特殊的关键字：Self 和 self。

- Self 代表当前的类型，比如 File 类型实现了 Write，那么实现过程中使用到的 Self 就指代 File。
- self 在用作方法的第一个参数时，实际上是 self: Self 的简写，所以 &self 是 self: &Self，而 &mut self 是 self: &mut Self。

光讲定义，理解不太深刻，我们构建一个 BufBuilder 结构实现 Write trait，结合代码来说明。（[Write trait 代码](#)）：

```
use std::fmt;
use std::io::Write;

struct BufBuilder {
    buf: Vec<u8>,
}

impl BufBuilder {
    pub fn new() -> Self {
        Self {
            buf: Vec::with_capacity(1024),
        }
    }
}

// 实现 Debug trait，打印字符串
impl fmt::Debug for BufBuilder {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}", String::from_utf8_lossy(&self.buf))
    }
}

impl Write for BufBuilder {
    fn write(&mut self, buf: &[u8]) -> std::io::Result<usize> {
        // 把 buf 添加到 BufBuilder 的尾部
        self.buf.extend_from_slice(buf);
        Ok(buf.len())
    }
}

fn flush(&mut self) -> std::io::Result<()> {
    // 由于是在内存中操作，所以不需要 flush
    Ok(())
}
```

```
fn main() {
    let mut buf = BufBuilder::new();
    buf.write_all(b"Hello world!".unwrap());
    println!("{}", buf);
}
```

从代码中可以看到，我们实现了 write 和 flush 方法，其它的方法都用缺省实现，这样 BufBuilder 对 Write trait 的实现是完整的。如果没有实现 write 或者 flush，Rust 编译器会报错，你可以自己尝试一下。

数据结构一旦实现了某个 trait，那么这个 trait 内部的方法都可以被使用，比如这里我们调用了 `buf.write_all()`。

那么 `write_all()` 是如何被调用的呢？我们回去看 `write_all` 的签名：

```
fn write_all(&mut self, buf: &[u8]) -> Result<()>
```

它接受两个参数：`&mut self` 和 `&[u8]`，第一个参数传递的是 `buf` 这个变量的可变引用，第二个参数传递的是 `b"Hello world!"`。

基本 trait 练习

好，搞明白 trait 基本的定义和使用后，我们来尝试定义一个 trait 巩固下。

假设我们要做一个字符串解析器，可以把字符串的某部分解析成某个类型，那么可以这么定义这个 trait：它有一个方法是 `parse`，这个方法接受一个字符串引用，返回 `Self`。

```
pub trait Parse {
    fn parse(s: &str) -> Self;
}
```

这个 `parse` 方法是 trait 的静态方法，因为它的第一个参数和 `self` 无关，所以在调用时需要使用 `T::parse(str)`。

我们来尝试为 `u8` 这个数据结构来实现 `parse`，比如说：“123abc”会被解析出整数 123，而“abcd”会被解析出 0。

要达到这样的目的，需要引入一个新的库 [Regex](#) 使用正则表达式提取需要的内容，除此之外，还需要使用 [str::parse](#) 函数 把一个包含数字的字符串转换成数字。

整个代码如下 ([Parse trait 练习代码](#)) :

```
use regex::Regex;
pub trait Parse {
    fn parse(s: &str) -> Self;
}

impl Parse for u8 {
    fn parse(s: &str) -> Self {
        let re: Regex = Regex::new(r"^[0-9]+").unwrap();
        if let Some(captures) = re.captures(s) {
            // 取第一个 match, 将其捕获的 digits 换成 u8
            captures.map_or(0, |s| s.as_str().parse().unwrap_or(0))
        } else {
            0
        }
    }

#[test]
fn parse_should_work() {
    assert_eq!(u8::parse("123abcd"), 123);
    assert_eq!(u8::parse("1234abcd"), 0);
    assert_eq!(u8::parse("abcd"), 0);
}

fn main() {
    println!("result: {}", u8::parse("255 hello world"));
}
```

这个实现并不难，如果你感兴趣的话，可以再尝试为 f64 实现这个 Parse trait，比如 “123.45abcd” 需要被解析成 123.45。

在实现 f64 的过程中，你是不是感觉除了类型和用于捕获的 regex 略有变化外，整个代码基本和上面的代码是重复的？作为开发者，我们希望 Don't Repeat Yourself (DRY)，所以这样的代码写起来很别扭，让人不舒服。有没有更好的方法？

有！上一讲介绍了泛型编程，所以在实现 trait 的时候，也可以用泛型参数来实现 trait，需要注意的是，要对泛型参数做一定的限制。

- 第一，不是任何类型都可以通过字符串解析出来，在例子中，我们只能处理数字类型，并且这个类型还要能够被 `str::parse` 处理。

具体看文档，`str::parse` 是一个泛型函数，它返回任何实现了 `FromStr` trait 的类型，所以这里对泛型参数的第一个限制是，它必须实现了 `FromStr` trait。

- 第二，上面代码当无法正确解析字符串的时候，会直接返回 0，表示无法处理，但我们使用泛型参数后，无法返回 0，因为 0 不一定是某个符合泛型参数的类型中的一个值。怎么办？

其实返回 0 的目的是为处理不了的情况，返回一个缺省值，在 Rust 标准库中有 Default trait，绝大多数类型都实现了这个 trait，来为数据结构提供缺省值，所以泛型参数的另一个限制是 Default。

好，基本的思路有了，来看看代码吧（[Parse trait DRY代码](#)）：

```
use std::str::FromStr;

use regex::Regex;
pub trait Parse {
    fn parse(s: &str) -> Self;
}

// 我们约束 T 必须同时实现了 FromStr 和 Default
// 这样在使用的时候我们就可以用这两个 trait 的方法了
impl<T> Parse for T
where
    T: FromStr + Default,
{
    fn parse(s: &str) -> Self {
        let re: Regex = Regex::new(r"^[0-9]+(\.[0-9]+)?").unwrap();
        // 生成一个创建缺省值的闭包，这里主要是为了简化后续代码
        // Default::default() 返回的类型根据上下文能推导出来，是 Self
        // 而我们约定了 Self，也就是 T 需要实现 Default trait
        let d = || Default::default();
        if let Some(captures) = re.captures(s) {
            captures
                .get(0)
                .map_or(d(), |s| s.as_str().parse().unwrap_or(d())))
        } else {
            d()
        }
    }
}

#[test]
fn parse_should_work() {
    assert_eq!(u32::parse("123abcd"), 123);
    assert_eq!(u32::parse("123.45abcd"), 0);
    assert_eq!(f64::parse("123.45abcd"), 123.45);
    assert_eq!(f64::parse("abcd"), 0f64);
}

fn main() {
    println!("result: {}", u8::parse("255 hello world"));
}
```

通过对带有约束的泛型参数实现 trait，一份代码就实现了 u32 / f64 等类型的 Parse trait，非常精简。不过，看这段代码你有没有感觉还是有些问题？当无法正确解析字符串时，我们返回了缺省值，难道不是应该返回一个错误么？

是的。这里返回缺省值的话，会跟解析“0abcd”这样的情况混淆，不知道解析出的 0，究竟是出错了，还是本该解析出 0。

所以更好的方式是 parse 函数返回一个 Result<T, E>：

```
pub trait Parse {  
    fn parse(s: &str) -> Result<Self, E>;  
}
```

但这里 Result 的 E 让人犯难了：要返回的错误信息，在 trait 定义时并不确定，不同的实现者可以使用不同的错误类型，这里 trait 的定义者最好能够把这种灵活性留给 trait 的实现者。怎么办？

想想既然 trait 允许内部包含方法，也就是关联函数，可不可以进一步包含关联类型呢？答案是肯定的。

带关联类型的 trait

Rust 允许 trait 内部包含关联类型，实现时跟关联函数一样，它也需要实现关联类型。我们看怎么为 Parse trait 添加关联类型：

```
pub trait Parse {  
    type Error;  
    fn parse(s: &str) -> Result<Self, Self::Error>;  
}
```

有了关联类型 Error，Parse trait 就可以在出错时返回合理的错误了，看修改后的代码（[Parse trait DRY.2代码](#)）：

```
use std::str::FromStr;  
  
use regex::Regex;  
pub trait Parse {  
    type Error;  
    fn parse(s: &str) -> Result<Self, Self::Error>  
    where  
        Self: Sized;  
}  
  
impl<T> Parse for T
```

```

where
T: FromStr + Default,
{
    // 定义关联类型 Error 为 String
    type Error = String;
    fn parse(s: &str) -> Result<Self, Self::Error> {
        let re: Regex = Regex::new(r"^[0-9]+(\.[0-9]+)?").unwrap();
        if let Some(captures) = re.captures(s) {
            // 当出错时我们返回 Err(String)
            captures
                .get(0)
                .map_or(Err("failed to capture".to_string()), |s| {
                    s.as_str()
                        .parse()
                        .map_err(|_err| "failed to parse captured string".to_string())
                })
        } else {
            Err("failed to parse string".to_string())
        }
    }
}

#[test]
fn parse_should_work() {
    assert_eq!(u32::parse("123abcd"), Ok(123));
    assert_eq!(
        u32::parse("123.45abcd"),
        Err("failed to parse captured string".into())
    );
    assert_eq!(f64::parse("123.45abcd"), Ok(123.45));
    assert!(f64::parse("abcd").is_err());
}

fn main() {
    println!("result: {:?}", u8::parse("255 hello world"));
}

```

上面的代码中，我们允许用户把错误类型延迟到 trait 实现时才决定，这种带有关联类型的 trait 比普通 trait，更加灵活，抽象度更高。

trait 方法里的参数或者返回值，都可以用关联类型来表述，而在实现有关联类型的 trait 时，只需要额外提供关联类型的具体类型即可。

支持泛型的 trait

到目前为止，我们一步步了解了基础 trait 的定义、使用，以及更为复杂灵活的带关联类型的 trait。所以结合上一讲介绍的泛型，你有没有想到这个问题：trait 的定义是不是也可以支持泛型呢？

比如要定义一个 Concat trait 允许数据结构拼接起来，那么自然而然地，我们希望 String 可以和 String 拼接、和 &str 拼接，甚至和任何能转换成 String 的数据结构拼接。这个时候，就需要 Trait 也支持泛型了。

来看看标准库里的操作符是如何重载的，以 [std::ops::Add](#) 这个用于提供加法运算的 trait 为例：

```
pub trait Add<Rhs = Self> {
    type Output;
    #[must_use]
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

这个 trait 有一个泛型参数 Rhs，代表加号右边的值，它被用在 add 方法的第二个参数位。这里 Rhs 默认是 Self，也就是说你用 Add trait，如果不提供泛型参数，那么加号右值和左值都要是相同的类型。

我们来定义一个复数类型，尝试使用下这个 trait ([Add trait 练习代码1](#))：

```
use std::ops::Add;

#[derive(Debug)]
struct Complex {
    real: f64,
    imagine: f64,
}

impl Complex {
    pub fn new(real: f64, imagine: f64) -> Self {
        Self { real, imagine }
    }
}

// 对 Complex 类型的实现
impl Add for Complex {
    type Output = Self;

    // 注意 add 第一个参数是 self，会移动所有权
    fn add(self, rhs: Self) -> Self::Output {
        let real = self.real + rhs.real;
        let imagine = self.imagine + rhs.imagine;
        Self::new(real, imagine)
    }
}

fn main() {
    let c1 = Complex::new(1.0, 1f64);
    let c2 = Complex::new(2 as f64, 3.0);
    println!("{} + {} = {}", c1, c2, c1 + c2);
    // c1、c2 已经被移动，所以下面这句无法编译
    // println!("{} + {} = {}", c1, c2, c1 + c2);
}
```

复数类型有实部和虚部，两个复数的实部相加，虚部相加，得到一个新的复数。注意 add 的第一个参数是 self，它会移动所有权，所以调用完两个复数 $c1 + c2$ 后，根据所有权规则，它们就无法使用了。

所以，Add trait 对于实现了 Copy trait 的类型如 u32、f64 等结构来说，用起来很方便，但对于我们定义的 Complex 类型，执行一次加法，原有的值就无法使用，很不方便，怎么办？能不能对 Complex 的引用实现 Add trait 呢？

可以的。我们为 &Complex 也实现 Add ([Add trait 练习代码2](#))：

```
// ...

// 如果不想移动所有权，可以为 &Complex 实现 add，这样可以做 &c1 + &c2
impl Add for &Complex {
    // 注意返回值不应该是 Self 了，因为此时 Self 是 &Complex
    type Output = Complex;

    fn add(self, rhs: Self) -> Self::Output {
        let real = self.real + rhs.real;
        let imagine = self.imagine + rhs.imagine;
        Complex::new(real, imagine)
    }
}

fn main() {
    let c1 = Complex::new(1.0, 1f64);
    let c2 = Complex::new(2 as f64, 3.0);
    println!("{} + {}i", c1, c2);
    println!("{} + {}i", c1.add(c2));
}
```

可以做 $\&c1 + \&c2$ ，这样所有权就不会移动了。

讲了这么多，你可能有疑问了，这里都只使用了缺省的泛型参数，那定义泛型有什么用？

我们用加法的实际例子，来回答这个问题。之前都是两个复数的相加，现在设计一个复数和一个实数直接相加，相加的结果是实部和实数相加，虚部不变。好，来看看这个需求怎么实现 ([Add trait 练习代码3](#))：

```
// ...

// 因为 Add<Rhs = Self> 是个泛型 trait，我们可以为 Complex 实现 Add<f64>
impl Add<f64> for &Complex {
    type Output = Complex;

    // rhs 现在是 f64 了
```

```

fn add(self, rhs: f64) -> Self::Output {
    let real = self.real + rhs;
    Complex::new(real, self.imagine)
}

fn main() {
    let c1 = Complex::new(1.0, 1f64);
    let c2 = Complex::new(2 as f64, 3.0);
    println!("{} + {} = {}", c1, c2, c1 + c2);
    println!("{} + 5.0 = {}", c1, c1 + 5.0);
    println!("{} + c2 = {}", c1, c1 + c2);
}

```

通过使用 Add，为 Complex 实现了和 f64 相加的方法。所以泛型 trait 可以让我们在需要的时候，对同一种类型的同一个 trait，有多个实现。

这个小例子实用性不太够，再来看一个实际工作中可能会使用到的泛型 trait，你就知道这个支持有多强大了。

[tower::Service](#) 是一个第三方库，它定义了一个精巧的用于处理请求，返回响应的经典 trait，在不少著名的第三方网络库中都有使用，比如处理 gRPC 的 [tonic](#)。

看 Service 的定义：

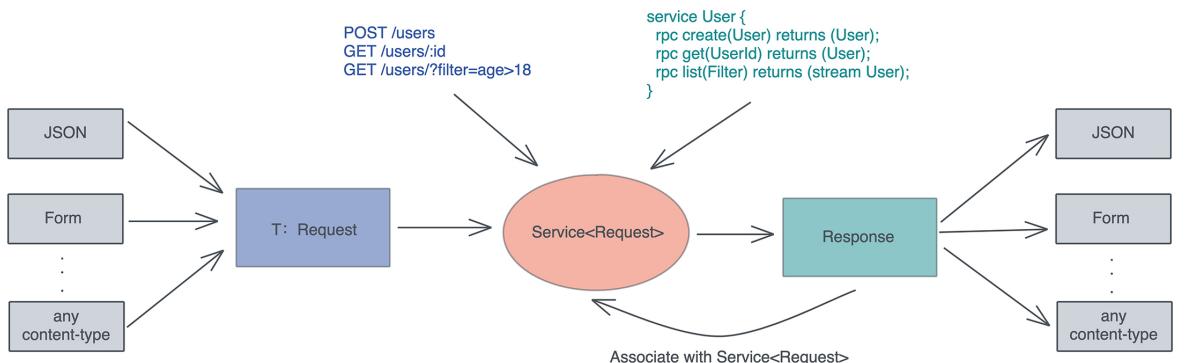
```

// Service trait 允许某个 service 的实现能处理多个不同的 Request
pub trait Service<Request> {
    type Response;
    type Error;
    // Future 类型受 Future trait 约束
    type Future: Future;
    fn poll_ready(
        &mut self,
        cx: &mut Context<'_>
    ) -> Poll<Result<(), Self::Error>>;
    fn call(&mut self, req: Request) -> Self::Future;
}

```

这个 trait 允许某个 Service 能处理多个不同的 Request。我们在 Web 开发中使用该 trait 的话，每个 Method+URL 可以定义为一个 Service，其 Request 是输入类型。

注意对于某个确定的 Request 类型，只会返回一种 Response，所以这里 Response 使用关联类型，而非泛型。如果有可能返回多个 Response，那么应该使用泛型 Service<Request, Response>。



极客时间

未来讲网络开发的时候再详细讲这个 trait，现在你只要能理解泛型 trait 的广泛应用场景就可以了。

trait 的“继承”

在 Rust 中，一个 trait 可以“继承”另一个 trait 的关联类型和关联函数。比如 trait B: A，是说任何类型 T，如果实现了 trait B，它也必须实现 trait A，换句话说，**trait B 在定义时可以使用 trait A 中的关联类型和方法**。

可“继承”对扩展 trait 的能力很有帮助，很多常见的 trait 都会使用 trait 继承来提供更多的能力，比如 tokio 库中的 `AsyncWriteExt`、futures 库中的 `StreamExt`。

以 StreamExt 为例，由于 StreamExt 中的方法都有缺省的实现，且所有实现了 Stream trait 的类型都实现了 StreamExt：

```
impl<T: ?Sized> StreamExt for T where T: Stream {}
```

所以如果你实现了 Stream trait，就可以直接使用 StreamExt 里的方法了，非常方便。

好，到这里 trait 就基本讲完了，简单总结一下，trait 作为对不同数据结构中相同行为的一种抽象。除了基本 trait 之外，

- 当行为和具体的数据关联时，比如字符串解析时定义的 Parse trait，我们引入了带有关联类型的 trait，把和行为有关的数据类型的定义，进一步延迟到 trait 实现的时候。

- 对于同一个类型的同一个 trait 行为，可以有不同的实现，比如我们之前大量使用的 From，此时可以用泛型 trait。

可以说 Rust 的 trait 就像一把瑞士军刀，把需要定义接口的各种场景都考虑进去了。

而特设多态是同一种行为的不同实现。所以其实，通过定义 trait 以及为不同的类型实现这个 trait，我们就已经实现了特设多态。

刚刚讲过的 Add trait 就是一个典型的特设多态，同样是加法操作，根据操作数据的不同进行不同的处理。Service trait 是一个不那么明显的特设多态，同样是 Web 请求，对于不同的 URL，我们使用不同的代码去处理。

如何做子类型多态？

从严格意义上说，子类型多态是面向对象语言的专利。如果一个对象 A 是对象 B 的子类，那么 A 的实例可以出现在任何期望 B 的实例的上下文中，比如猫和狗都是动物，如果一个函数的接口要求传入一个动物，那么传入猫和狗都是允许的。

Rust 虽然没有父类和子类，但 trait 和实现 trait 的类型之间也是类似的关系，所以，Rust 也可以做子类型多态。看一个例子（[代码](#)）：

```
struct Cat;
struct Dog;

trait Animal {
    fn name(&self) -> &'static str;
}

impl Animal for Cat {
    fn name(&self) -> &'static str {
        "Cat"
    }
}

impl Animal for Dog {
    fn name(&self) -> &'static str {
        "Dog"
    }
}

fn name(animal: impl Animal) -> &'static str {
    animal.name()
}

fn main() {
    let cat = Cat;
    println!("cat: {}", name(cat));
}
```

这里 `impl Animal` 是 `T: Animal` 的简写，所以 `name` 函数的定义和以下定义等价：

```
fn name<T: Animal>(animal: T) -> &'static str;
```

上一讲提到过，这种泛型函数会根据具体使用的类型被单态化，编译成多个实例，是静态分派。

静态分派固然很好，效率很高，但很多时候，类型可能很难在编译时决定。比如要撰写一个格式化工具，这个在 IDE 里很常见，我们可以定义一个 `Formatter` 接口，然后创建一系列实现：

```
pub trait Formatter {
    fn format(&self, input: &mut String) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with Markdown formatter");
        true
    }
}

struct RustFormatter;
impl Formatter for RustFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with Rust formatter");
        true
    }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with HTML formatter");
        true
    }
}
```

首先，使用什么格式化方法，只有当打开文件，分析出文件内容之后才能确定，我们无法在编译期给定一个具体类型。其次，一个文件可能有一到多个格式化工具，比如一个 Markdown 文件里有 Rust 代码，同时需要 `MarkdownFormatter` 和 `RustFormatter` 来格式化。

这里如果使用一个 `Vec` 来提供所有需要的格式化工具，那么，下面这个函数其 `formatters` 参数该如何确定类型呢？

```
pub fn format(input: &mut String, formatters: Vec<???>) {
    for formatter in formatters {
        formatter.format(input);
    }
}
```

正常情况下，`Vec<&dyn Formatter>` 容器里的类型需要是一致的，但此处无法给定一个一致的类型。

所以我们要有一种手段，告诉编译器，此处需要并且仅需要任何实现了 `Formatter` 接口的数据类型。在 Rust 里，这种类型叫**Trait Object**，表现为 `&dyn Trait` 或者 `Box<dyn Trait>`。

这里，`dyn` 关键字只是用来帮助我们更好地区分普通类型和 Trait 类型，阅读代码时，看到 `dyn` 就知道后面跟的是一个 trait 了。

于是，上述代码可以写成：

```
pub fn format(input: &mut String, formatters: Vec<&dyn Formatter>) {
    for formatter in formatters {
        formatter.format(input);
    }
}
```

这样可以在运行时，构造一个 `Formatter` 的列表，传递给 `format` 函数进行文件的格式化，这就是**动态分派**（dynamic dispatching）。

看最终调用的**格式化工具代码**：

```
pub trait Formatter {
    fn format(&self, input: &mut String) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with Markdown formatter");
        true
    }
}

struct RustFormatter;
impl Formatter for RustFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with Rust formatter");
    }
}
```

```
        true
    }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
    fn format(&self, input: &mut String) -> bool {
        input.push_str("\nformatted with HTML formatter");
        true
    }
}

pub fn format(input: &mut String, formatters: Vec<&dyn Formatter>) {
    for formatter in formatters {
        formatter.format(input);
    }
}

fn main() {
    let mut text = "Hello world!".to_string();
    let html: &dyn Formatter = &HtmlFormatter;
    let rust: &dyn Formatter = &RustFormatter;
    let formatters = vec![html, rust];
    format(&mut text, formatters);

    println!("text: {}", text);
}
```

这个实现是不是很简单？学到这里你在兴奋之余，不知道会不会感觉有点负担，又一个Rust新名词出现了。别担心，虽然 Trait Object 是 Rust 独有的概念，但是这个概念并不新鲜。为什么这么说呢，来看它的实现机理。

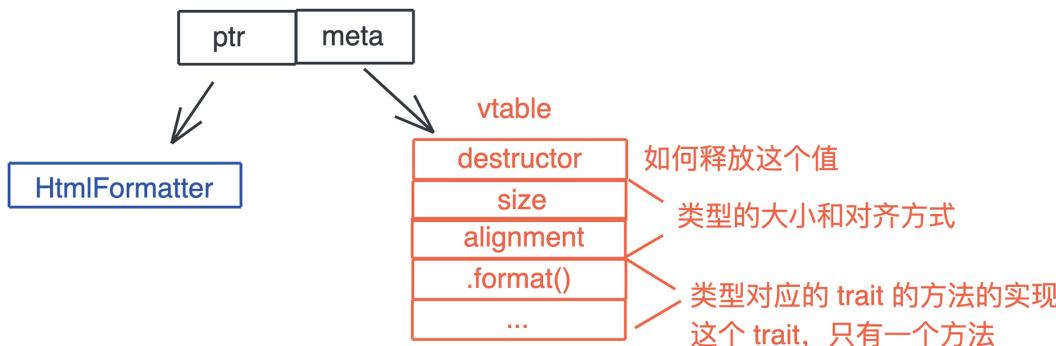
Trait Object 的实现机理

当需要使用 Formatter trait 做动态分派时，可以像如下例子一样，将一个具体类型的引用，赋给 `&Formatter`：

```
let mut text = "Hello world!".to_string();

let formatter: &dyn Formatter = &HtmlFormatter; // 使用 &HtmlFormatter 赋值给
                                                // &Formatter 创建一个 trait object

formatter.format(&mut text); // 调用 trait 的方法
```



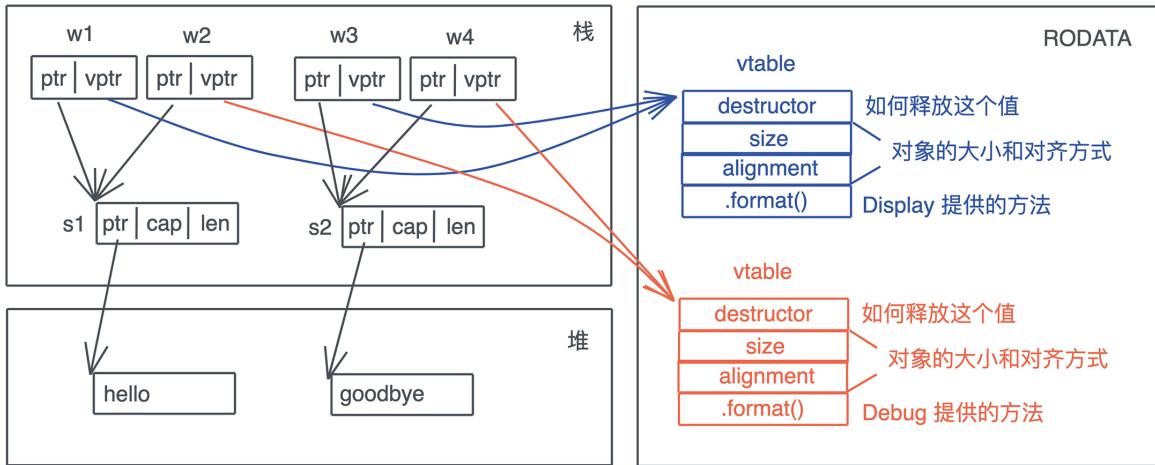
HtmlFormatter 的引用赋值给 Formatter 后，会生成一个 Trait Object，在上图中可以看到，Trait Object 的底层逻辑就是 **胖指针**。其中，一个指针指向数据本身，另一个则指向虚函数表（vtable）。

vtable 是一张静态的表，Rust 在编译时会为使用了 trait object 的类型的 trait 实现生成一张表，放在可执行文件中（一般在 TEXT 或 RODATA 段）。看下图，可以帮助你理解：

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

// Display / Debug trait object for s1
let w1: &dyn Display = &s1;
let w2: &dyn Debug = &s1;

// Display / Debug trait object for s2
let w3: &dyn Display = &s2;
let w4: &dyn Debug = &s2;
```



极客时间

在这张表里，包含具体类型的一些信息，如 `size`、`aligment` 以及一系列函数指针：

- 这个接口支持的所有方法，比如 `format()`；
- 具体类型的 drop trait，当 Trait object 被释放，它用来释放其使用的所有资源。

这样，当在运行时执行 `formatter.format()` 时，formatter 就可以从 vtable 里找到对应的函数指针，执行具体的操作。

所以，Rust 里的 Trait Object 没什么神秘的，它不过是我们熟知的 C++ / Java 中 vtable 的一个变体而已。

这里说句题外话，C++ / Java 指向 vtable 的指针，在编译时放在类结构里，而 Rust 放在 Trait object 中。这也是为什么 Rust 很容易对原生类型做动态分派，而 C++/Java 不行。

事实上，Rust 也并不区分原生类型和组合类型，对 Rust 来说，所有类型的地位都是一致的。

不过，你使用 trait object 的时候，要注意对象安全（object safety）。只有满足对象安全的 trait 才能使用 trait object，在[官方文档](#)中有详细讨论。

那什么样的 trait 不是对象安全的呢？

如果 trait 所有的方法，返回值是 Self 或者携带泛型参数，那么这个 trait 就不能产生 trait object。

不允许返回 Self，是因为 trait object 在产生时，原来的类型会被抹去，所以 Self 究竟是谁不知道。比如 Clone trait 只有一个方法 clone()，返回 Self，所以它就不能产生 trait object。

不允许携带泛型参数，是因为 Rust 里带泛型的类型在编译时会做单态化，而 trait object 是运行时的产物，两者不能兼容。

比如 From trait，因为整个 trait 带了泛型，每个方法也自然包含泛型，就不能产生 trait object。如果一个 trait 只有部分方法返回 Self 或者使用了泛型参数，那么这部分方法在 trait object 中不能调用。

小结

今天完整地介绍了 trait 是如何定义和使用的，包括最基本的 trait、带关联类型的 trait，以及泛型 trait。我们还回顾了通过 trait 做静态分发以及使用 trait object 做动态分发。

今天的内容比较多，不太明白的地方建议你多看几遍，你也可以通过下图来回顾这一讲的主要内容：

<p>基本 trait</p> <p>可以为数据结构定义抽象的接口 使用 self 可以访问实现 trait 的数据结构</p> <pre>pub trait Write { fn write(&mut self, buf: &[u8]) -> Result<usize>; ... }</pre>	<p>泛型 trait</p> <p>trait 可以有一个甚至多个泛型参数</p> <pre>pub trait From<T> { fn from(T) -> Self; } pub trait Service <Request> { type Response; type Error; // Future 类型受 Future trait 约束 type Future: Future; fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>; fn call(&mut self, req: Request) -> Self::Future; }</pre>
<p>带关联类型的 trait</p> <p>trait 内部可以定义和这个 trait 关联的类型</p> <pre>pub trait Iterator { type Item; fn next(&mut self) -> Option <Self::Item>; ... }</pre>	
<p>动态分派</p> <p>使用 trait object</p> <pre>pub trait Formatter { fn format(&self, input: &mut String) -> bool; } pub fn format(input: &mut String, formatters: Vec< &dyn Formatter >) { for formatter in formatters { formatter.format(input); } }</pre>	
<p>静态分派</p> <p>使用泛型函数</p> <pre>trait Animal { fn name(&self) -> &'static str; } fn name(animal: impl Animal) -> &'static str { animal.name() }</pre>	



trait 作为对不同数据结构中相同行为的一种抽象，它可以让我们在开发时，通过用户需求，先敲定系统的行为，把这些行为抽象成 trait，之后再慢慢确定要使用的数据结构，以及如何为数据结构实现这些 trait。

所以，trait 是你做 Rust 开发的核心元素。什么时候使用什么 trait，需要根据需求来确定。

但是需求往往不是那么明确的，尤其是因为我们要把用户需求翻译成系统设计上的需求。这种翻译能力，得靠足够多源码的阅读和思考，以及足够丰富的历练，一点点累积成的。因为 Rust 的 trait 再强大，也只是一把瑞士军刀，能让它充分发挥作用的是持有它的那个人。

以在 get hands dirty 系列中写的代码为例，我们使用了 trait 对系统进行解耦，并增强其扩展性，你可以简单回顾一下。比如第 5 讲的 Engine trait 和 SpecTransform trait，使用了普通 trait：

```

// Engine trait: 未来可以添加更多的 engine, 主流程只需要替换 engine
pub trait Engine {
    // 对 engine 按照 specs 进行一系列有序的处理
    fn apply(&mut self, specs: &[Spec]);
    // 从 engine 中生成目标图片, 注意这里用的是 self, 而非 self 的引用
    fn generate(self, format: ImageOutputFormat) -> Vec<u8>;
}

// SpecTransform: 未来如果添加更多的 spec, 只需要实现它即可
pub trait SpecTransform<T> {
    // 对图片使用 op 做 transform
    fn transform(&mut self, op: T);
}

```

第 6 讲的 Fetch/Load trait, 使用了带关联类型的 trait:

```

// Rust 的 async trait 还没有稳定, 可以用 async_trait 宏
#[async_trait]
pub trait Fetch {
    type Error;
    async fn fetch(&self) -> Result<String, Self::Error>;
}

pub trait Load {
    type Error;
    fn load(self) -> Result<DataSet, Self::Error>;
}

```

思考题

1.对于 Add trait, 如果我们不用泛型, 把 Rhs 作为 Add trait 的关联类型, 可以么? 为什么?

2.如下代码能编译通过么, 为什么?

```

use std::{fs::File, io::Write};
fn main() {
    let mut f = File::create("/tmp/test_write_trait").unwrap();
    let w: &mut dyn Write = &mut f;
    w.write_all(b"hello ").unwrap();
    let w1 = w.by_ref();
    w1.write_all(b"world").unwrap();
}

```

3.在 Complex 的例子中, $c1 + c2$ 会导致所有权移动, 所以我们使用了 $\&c1 + \&c2$ 来避免这种行为。除此之外, 你还有什么方法能够让 $c1 + c2$ 执行完之后还能继续使用么? 如何修改 Complex 的代码来实现这个功能呢?

```
// c1、c2 已经被移动，所以下面这句无法编译
// println!("{}", c1 + c2);
```

4. 学有余力的同学可以挑战一下，[Iterator](#) 是 Rust 下的迭代器的 trait，你可以阅读 Iterator 的文档获得更多的信息。它有一个关联类型 Item 和一个方法 next() 需要实现，每次调用 next，如果迭代器中还能得到一个值，则返回 Some (Item)，否则返回 None。请阅读[如下代码](#)，想想看如何实现 Sentencelter 这个结构的迭代器？

```
struct Sentencelter<'a> {
    s: &'a mut &'a str,
    delimiter: char,
}

impl<'a> Sentencelter<'a> {
    pub fn new(s: &'a mut &'a str, delimiter: char) -> Self {
        Self { s, delimiter }
    }
}

impl<'a> Iterator for Sentencelter<'a> {
    type Item; // 想想 Item 应该是什么类型?

    fn next(&mut self) -> Option<Self::Item> {
        // 如何实现 next 方法让下面的测试通过?
        todo!()
    }
}

#[test]
fn it_works() {
    let mut s = "This is the 1st sentence. This is the 2nd sentence.";
    let mut iter = Sentencelter::new(&mut s, '.');
    assert_eq!(iter.next(), Some("This is the 1st sentence."));
    assert_eq!(iter.next(), Some("This is the 2nd sentence."));
    assert_eq!(iter.next(), None);
}

fn main() {
    let mut s = "a。 b。 c";
    let sentences: Vec<_> = Sentencelter::new(&mut s, '。').collect();
    println!("sentences: {:?}", sentences);
}
```

今天你已经完成了Rust学习的第13次打卡。我们下节课见～

延伸阅读

使用 trait 有两个注意事项：

- 第一，在定义和使用 trait 时，我们需要遵循孤儿规则（Orphan Rule）。

trait 和实现 trait 的数据类型，至少有一个是在当前 crate 中定义的，也就是说，你不能为第三方的类型实现第三方的 trait，当你尝试这么做时，Rust 编译器会报错。我们在第6讲的 SQL查询工具query中，定义了很多简单的直接包裹已有数据结构的类型，就是为了应对孤儿规则。

- 第二，Rust 对含有 async fn 的 trait，还没有一个很好的被标准库接受的实现，如果你感兴趣可以看[这篇文章](#)了解它背后的原因。

在第5讲Thumbar图片服务器我们使用了 async_trait 这个库，为 trait 的实现添加了一个标记宏 #[async_trait]。这是目前最推荐的无缝使用 async trait 的方法。未来 async trait 如果有了标准实现，我们不需要对现有代码做任何改动。

使用 async_trait 的代价是每次调用会发生额外的堆内存分配，但绝大多数应用场景下，这并不会有性能上的问题。

还记得当时写get hands dirty系列时，说我们在后面讲到具体知识点会再回顾么。你可以再回去看看（第5讲）在 Thumbar图片服务器中定义的 Engine / SpecTransform，以及（第6讲）在SQL查询工具query中定义的 Fetch / Load，想想它们的作用以及给架构带来的好处。

另外，有同学可能好奇为什么我说“ vtable 会为每个类型的每个 trait 实现生成一张表”。这个并没有在任何公开的文档中提及，不过既然它是一个数据结构，我们就可以通过打印它的地址来追踪它的行为。我写了一段代码，你可以自行运行来进一步加深对 vtable 的理解（[代码](#)）：

```
use std::fmt::{Debug, Display};
use std::mem::transmute;

fn main() {
    let s1 = String::from("hello world!");
    let s2 = String::from("goodbye world!");
    // Display / Debug trait object for s
    let w1: &dyn Display = &s1;
    let w2: &dyn Debug = &s1;

    // Display / Debug trait object for s1
    let w3: &dyn Display = &s2;
    let w4: &dyn Debug = &s2;

    // 强行把 trait object 转换成两个地址 (usize, usize)
    // 这是不安全的，所以是 unsafe
    let (addr1, vtable1): (usize, usize) = unsafe { transmute(w1) };
    let (addr2, vtable2): (usize, usize) = unsafe { transmute(w2) };
    let (addr3, vtable3): (usize, usize) = unsafe { transmute(w3) };
    let (addr4, vtable4): (usize, usize) = unsafe { transmute(w4) };
```

```
&nbsp; &nbsp; // s 和 s1 在栈上的地址, 以及 main 在 TEXT 段的地址
&nbsp; &nbsp; println!(
&nbsp; &nbsp; &nbsp; &nbsp; "s1: {:p}, s2: {:p}, main(): {:p}",
&nbsp; &nbsp; &nbsp; &nbsp; &s1, &s2, main as *const ()
&nbsp; &nbsp; );
&nbsp; &nbsp; // trait object(s / Display) 的 ptr 地址和 vtable 地址
&nbsp; &nbsp; println!("addr1: 0x{:x}, vtable1: 0x{:x}", addr1, vtable1);
&nbsp; &nbsp; // trait object(s / Debug) 的 ptr 地址和 vtable 地址
&nbsp; &nbsp; println!("addr2: 0x{:x}, vtable2: 0x{:x}", addr2, vtable2);

&nbsp; &nbsp; // trait object(s1 / Display) 的 ptr 地址和 vtable 地址
&nbsp; &nbsp; println!("addr3: 0x{:x}, vtable3: 0x{:x}", addr3, vtable3);

&nbsp; &nbsp; // trait object(s1 / Display) 的 ptr 地址和 vtable 地址
&nbsp; &nbsp; println!("addr4: 0x{:x}, vtable4: 0x{:x}", addr4, vtable4);

&nbsp; &nbsp; // 指向同一个数据的 trait object 其 ptr 地址相同
&nbsp; &nbsp; assert_eq!(addr1, addr2);
&nbsp; &nbsp; assert_eq!(addr3, addr4);

&nbsp; &nbsp; // 指向同一种类型的同一个 trait 的 vtable 地址相同
&nbsp; &nbsp; // 这里都是 String + Display
&nbsp; &nbsp; assert_eq!(vtable1, vtable3);
&nbsp; &nbsp; // 这里都是 String + Debug
&nbsp; &nbsp; assert_eq!(vtable2, vtable4);
}
```

(如果你觉得有收获, 也欢迎你分享给身边的朋友, 邀他一起讨论~)

03 | 类型系统：有哪些必须掌握的Trait?

开发软件系统时, 我们弄清楚需求, 要对需求进行架构上的分析和设计。在这个过程中, 合理地定义和使用 trait, 会让代码结构具有很好的扩展性, 让系统变得非常灵活。

之前在 get hands dirty 系列中就粗略见识到了 trait 的巨大威力, 使用了 From / TryFrom trait 进行类型的转换 ([第 5 讲](#)), 还使用了 Deref trait ([第 6 讲](#)) 让类型在不暴露其内部结构代码的同时, 让内部结构的方法可以对外使用。

经过上两讲的学习, 相信你现在对 trait 的理解就深入了。在实际解决问题的过程中, **用好这些 trait, 会让你的代码结构更加清晰, 阅读和使用都更加符合 Rust 生态的习惯**。比如数据结构实现了 Debug trait, 那么当你想打印数据结构时, 就可以用 `{:?}` 来打印; 如果你的数据结构实现了 From, 那么, 可以直接使用 `into()` 方法做数据转换。

trait

Rust 语言的标准库定义了大量的标准 trait, 来先来数已经学过的, 看看攒了哪些:

- Clone / Copy trait, 约定了数据被深拷贝和浅拷贝的行为;

- Read / Write trait, 约定了对 I/O 读写的行为；
- Iterator, 约定了迭代器的行为；
- Debug, 约定了数据如何被以 debug 的方式显示出来的行为；
- Default, 约定数据类型的缺省值如何产生的行为；
- From / TryFrom, 约定了数据间如何转换的行为。

我们会再学习几类重要的 trait，包括和内存分配释放相关的 trait、用于区别不同类型协助编译器做类型安全检查的标记 trait、进行类型转换的 trait、操作符相关的 trait，以及 Debug/Display/Default。

在学习这些 trait 的过程中，你也可以结合之前讲的内容，有意识地思考一下 Rust 为什么这么设计，在增进对语言理解的同时，也能写出更加优雅的 Rust 代码。

内存相关：Clone / Copy / Drop

首先来看内存相关的 Clone/Copy/Drop。这三个 trait 在介绍所有权的时候已经学习过，这里我们再深入研究一下它们的定义和使用场景。

Clone trait

首先看 Clone：

```
pub trait Clone {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Clone trait 有两个方法，`clone()` 和 `clone_from()`，后者有缺省实现，所以平时我们只需要实现 `clone()` 方法即可。你也许会疑惑，这个 `clone_from()` 有什么作用呢？因为看起来 `a.clone_from(&b)`，和 `a = b.clone()` 是等价的。

其实不是，如果 `a` 已经存在，在 `clone` 过程中会分配内存，那么用 `a.clone_from(&b)` 可以避免内存分配，提高效率。

Clone trait 可以通过派生宏直接实现，这样能简化不少代码。如果在你的数据结构里，每一个字段都已经实现了Clone trait，你可以用 `#[derive(Clone)]`，看下面的[代码](#)，定义了 Developer 结构和 Language 枚举：

```
#[derive(Clone, Debug)]
struct Developer {
    name: String,
    age: u8,
    lang: Language
}

#[allow(dead_code)]
#[derive(Clone, Debug)]
enum Language {
    Rust,
    TypeScript,
    Elixir,
    Haskell
}

fn main() {
    let dev = Developer {
        name: "Tyr".to_string(),
        age: 18,
        lang: Language::Rust
    };
    let dev1 = dev.clone();
    println!("dev: {:?}", dev, addr of dev name: {:p}", dev, dev.name.as_str());
    println!("dev1: {:?}", dev1, addr of dev1 name: {:p}", dev1, dev1.name.as_str())
}
```

如果没有为 Language 实现 Clone 的话，Developer 的派生宏 Clone 将会编译出错。运行这段代码可以看到，对于 name，也就是 String 类型的 Clone，其堆上的内存也被 Clone 了一份，所以 Clone 是深度拷贝，栈内存和堆内存一起拷贝。

值得注意的是，clone 方法的接口是 `&self`，这在绝大多数场合下都是适用的，我们在 clone 一个数据时只需要有已有数据的只读引用。但对 Rc 这样在 `clone()` 时维护引用计数的数据结构，`clone()` 过程中会改变自己，所以要用 Cell 这样提供内部可变性的结构来进行改变，如果你也有类似的需求，可以参考。

Copy trait

和 Clone trait 不同的是，Copy trait 没有任何额外的方法，它只是一个标记 trait (marker trait)。它的 trait 定义：

```
pub trait Copy: Clone {}
```

所以看这个定义，如果要实现 Copy trait 的话，必须实现 Clone trait，然后实现一个空的 Copy trait。你是不是有点疑惑：这样不包含任何行为的 trait 有什么用呢？

这样的 trait 虽然没有任何行为，但它可以用作 trait bound 来进行类型安全检查，所以我们管它叫标记 trait。

和 Clone 一样，如果数据结构的所有字段都实现了 Copy，也可以用 `#[derive(Copy)]` 宏来为数据结构实现 Copy。试着为 Developer 和 Language 加上 Copy：

```
#[derive(Clone, Copy, Debug)]
struct Developer {
    name: String,
    age: u8,
    lang: Language
}

#[derive(Clone, Copy, Debug)]
enum Language {
    Rust,
    TypeScript,
    Elixir,
    Haskell
}
```

这个代码会出错。因为 String 类型没有实现 Copy。因此，Developer 数据结构只能 clone，无法 copy。我们知道，如果类型实现了 Copy，那么在赋值、函数调用的时候，值会被拷贝，否则所有权会被移动。

所以上面的代码 Developer 类型在做参数传递时，会执行 Move 语义，而 Language 会执行 Copy 语义。

在讲所有权可变/不可变引用的时候提到，不可变引用实现了 Copy，而可变引用 `&mut T` 没有实现 Copy。为什么是这样？

因为如果可变引用实现了 Copy trait，那么生成一个可变引用然后把它赋值给另一个变量时，就会违背所有权规则：同一个作用域下只能有一个可变引用。可见，Rust 标准库在哪些结构可以 Copy、哪些不可以 Copy 上，有着仔细的考量。

Drop trait

在内存管理中已经详细探讨过 Drop trait。这里我们再看一下它的定义：

```
pub trait Drop {
    fn drop(&mut self);
}
```

大部分场景无需为数据结构提供 Drop trait，系统默认会依次对数据结构的每个域做 drop。但有两种情况你可能需要手工实现 Drop。

第一种是希望在数据结束生命周期的时候做一些事情，比如记日志。

第二种是需要对资源回收的场景。编译器并不知道你额外使用了哪些资源，也就无法帮助你 drop 它们。比如说锁资源的释放，在 MutexGuard 中实现了 Drop 来释放锁资源：

```
impl<T: ?Sized> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        unsafe {
            self.lock.poison.done(&self.poison);
            self.lock.inner.raw_unlock();
        }
    }
}
```

需要注意的是，Copy trait 和 Drop trait 是互斥的，两者不能共存，当你尝试为同一种数据类型实现 Copy 时，也实现 Drop，编译器就会报错。这其实很好理解：**Copy是按位做浅拷贝，那么它会默认拷贝的数据没有需要释放的资源；而Drop恰恰是为了释放额外的资源而生的。**

我们还是写一段代码来辅助理解，在代码中，强行用 Box::into_raw 获得堆内存的指针，放入 RawBuffer 结构中，这样就接管了这块堆内存的释放。

虽然 RawBuffer 可以实现 Copy trait，但这样一来就无法实现 Drop trait。如果程序非要这么写，会导致内存泄漏，因为该释放的堆内存没有释放。

但是这个操作不会破坏 Rust 的正确性保证：即便你 Copy 了 N 份 RawBuffer，由于无法实现 Drop trait，RawBuffer 指向的那同一块堆内存不会释放，所以不会出现 use after free 的内存安全问题。（[代码](#)）

```
use std::{fmt, slice};

// 注意这里，我们实现了 Copy，这是因为 *mut u8/usize 都支持 Copy
#[derive(Clone, Copy)]
struct RawBuffer {
    // 裸指针用 *const / *mut 来表述，这和引用的 & 不同
    ptr: *mut u8,
```

```

    len: usize,
}

impl From<Vec<u8>> for RawBuffer {
    fn from(vec: Vec<u8>) -> Self {
        let slice = vec.into_boxed_slice();
        Self {
            len: slice.len(),
            // into_raw 之后, Box 就不管这块内存的释放了, RawBuffer 需要处理释放
            ptr: Box::into_raw(slice) as *mut u8,
        }
    }
}

// 如果 RawBuffer 实现了 Drop trait, 就可以在所有者退出时释放堆内存
// 然后, Drop trait 会跟 Copy trait 冲突, 要么不实现 Copy, 要么不实现 Drop
// 如果不实现 Drop, 那么就会导致内存泄漏, 但它不会对正确性有任何破坏
// 比如不会出现 use after free 这样的问题。
// 你可以试着把下面注释去掉, 看看会出什么问题
// impl Drop for RawBuffer {
//     #[inline]
//     fn drop(&mut self) {
//         let data = unsafe { Box::from_raw(slice::from_raw_parts_mut(self.ptr, self.len)) };
//         drop(data)
//     }
// }

impl fmt::Debug for RawBuffer {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let data = self.as_ref();
        write!(f, "{:p}: {:?}", self.ptr, data)
    }
}

impl AsRef<[u8]> for RawBuffer {
    fn as_ref(&self) -> &[u8] {
        unsafe { slice::from_raw_parts(self.ptr, self.len) }
    }
}

fn main() {
    let data = vec![1, 2, 3, 4];

    let buf: RawBuffer = data.into();

    // 因为 buf 允许 Copy, 所以这里 Copy 了一份
    use_buffer(buf);

    // buf 还能用
    println!("buf: {:?}", buf);
}

fn use_buffer(buf: RawBuffer) {
    println!("buf to die: {:?}", buf);

    // 这里不用特意 drop, 写出来只是为了说明 Copy 出来的 buf 被 Drop 了
    drop(buf)
}

```

对于代码安全来说，内存泄漏危害大？还是 use after free 危害大呢？肯定是后者。Rust 的底线是内存安全，所以两害相权取其轻。

实际上，任何编程语言都无法保证不发生人为的内存泄漏，比如程序在运行时，开发者疏忽了，对哈希表只添加不删除，就会造成内存泄漏。但 Rust 会保证即使开发者疏忽了，也不会出现内存安全问题。

建议你仔细阅读这段代码中的注释，试着把注释掉的 Drop trait 恢复，然后再把代码改得可以编译通过，认真思考一下 Rust 这样做的良苦用心。

标记 trait: Sized / Send / Sync / Unpin

好，讲完内存相关的主要 trait，来看标记 trait。

刚才我们已经看到了一个标记 trait: Copy。Rust 还支持其它几种标记 trait: [Sized](#) / [Send](#) / [Sync](#) / [Unpin](#)。

Sized trait 用于标记有具体大小的类型。在使用泛型参数时，Rust 编译器会自动为泛型参数加上 Sized 约束，比如下面的 Data 和处理 Data 的函数 process_data：

```
struct Data<T> {
    inner: T,
}

fn process_data<T>(data: Data<T>) {
    todo!();
}
```

它等价于：

```
struct Data<T: Sized> {
    inner: T,
}

fn process_data<T: Sized>(data: Data<T>) {
    todo!();
}
```

大部分时候，我们都希望能自动添加这样的约束，因为这样定义出的泛型结构，在编译期，大小是固定的，可以作为参数传递给函数。如果没有这个约束，T 是大小不固定的类型，process_data 函数会无法编译。

但是这个自动添加的约束有时候不太适用，在少数情况下，需要 T 是可变类型的，怎么办？Rust 提供了 ?Sized 来摆脱这个约束。

如果开发者显式定义了 T: ?Sized，那么 T 就可以是任意大小。如果你对（第12讲）之前说的 Cow 还有印象，可能会记得 Cow 中泛型参数 B 的约束是 ?Sized：

```
pub enum Cow<'a, B: ?Sized + 'a> where B: ToOwned,  
{  
    // 借用的数据  
    Borrowed(&'a B),  
    // 拥有的数据  
    Owned(<B as ToOwned>::Owned),  
}
```

这样 B 就可以是 [T] 或者 str 类型，大小都是不固定的。要注意 Borrowed(&'a B) 大小是固定的，因为它内部是对 B 的一个引用，而引用的大小是固定的。

Send / Sync

说完了 Sized，我们再来看 Send / Sync，定义是：

```
pub unsafe auto trait Send {}  
pub unsafe auto trait Sync {}
```

这两个 trait 都是 unsafe auto trait，auto 意味着编译器会在合适的场合，自动为数据结构添加它们的实现，而 unsafe 代表实现的这个 trait 可能会违背 Rust 的内存安全准则，如果开发者手工实现这两个 trait，要自己为它们的安全性负责。

Send/Sync 是 Rust 并发安全的基础：

- 如果一个类型 T 实现了 Send trait，意味着 T 可以安全地从一个线程移动到另一个线程，也就是说所有权可以在线程间移动。
- 如果一个类型 T 实现了 Sync trait，则意味着 &T 可以安全地在多个线程中共享。一个类型 T 满足 Sync trait，当且仅当 &T 满足 Send trait。

对于 Send / Sync 在线程安全中的作用，可以这么看，如果一个类型 T: Send，那么 T 在某个线程中的独占访问是线程安全的；如果一个类型 T: Sync，那么 T 在线程间的只读共享是安全的。

对于我们自己定义的数据结构，如果其内部的所有域都实现了 Send / Sync，那么这个数据结构会被自动添加 Send / Sync。基本上原生数据结构都支持 Send / Sync，也就是说，绝大多数自定义的数据结构都是满足 Send / Sync 的。标准库中，不支持 Send / Sync 的数据结构主要有：

- 裸指针 *const T / *mut T。它们是不安全的，所以既不是 Send 也不是 Sync。
- UnsafeCell 不支持 Sync。也就是说，任何使用了 Cell 或者 RefCell 的数据结构不支持 Sync。
- 引用计数 Rc 不支持 Send 也不支持 Sync。所以 Rc 无法跨线程。

之前介绍过 Rc / RefCell ([第9讲](#))，我们来看看，如果尝试跨线程使用 Rc / RefCell，会发生什么。在 Rust 下，如果想创建一个新的线程，需要使用 `std::thread::spawn`：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

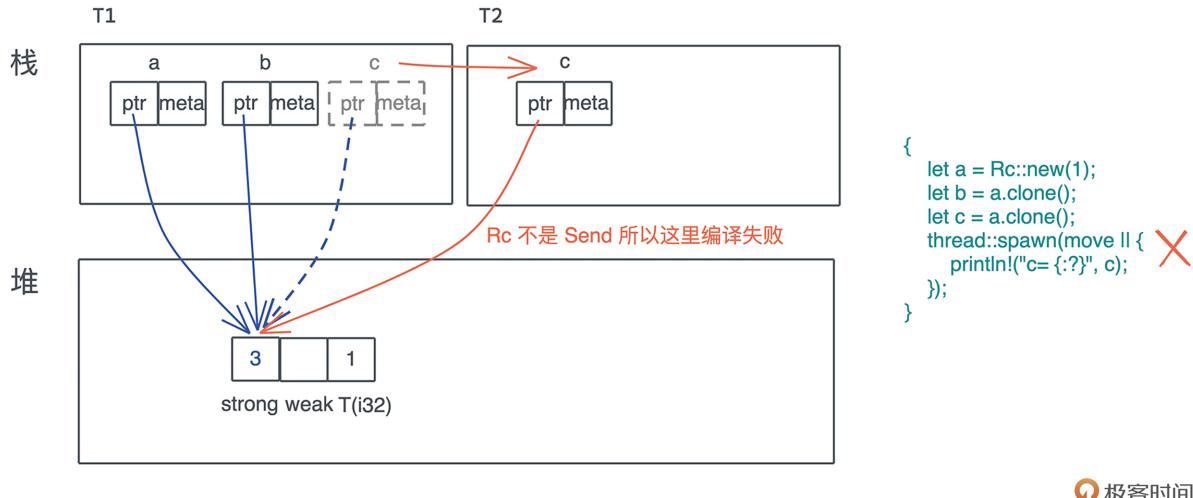
它的参数是一个闭包（后面会讲），这个闭包需要 Send + 'static：

- 'static 意思是闭包捕获的自由变量必须是一个拥有所有权的类型，或者是一个拥有静态生命周期的引用；
- Send 意思是，这些被捕获自由变量的所有权可以从一个线程移动到另一个线程。

从这个接口上，可以得出结论：如果在线程间传递 Rc，是无法编译通过的，因为 [Rc 的实现不支持 Send 和 Sync](#)。写段代码验证一下 ([代码](#))：

```
// Rc 既不是 Send，也不是 Sync
fn rc_is_not_send_and_sync() {
    let a = Rc::new(1);
    let b = a.clone();
    let c = a.clone();
    thread::spawn(move || {
        println!("c= {:?}", c);
    });
}
```

果然，这段代码不通过。



极客时间

那么，RefCell 可以在线程间转移所有权么？RefCell 实现了 Send，但没有实现 Sync，所以，看起来是可以工作的（[代码](#)）：

```
fn refcell_is_send() {
    let a = RefCell::new(1);
    thread::spawn(move || {
        println!("a= {:?}", a);
    });
}
```

验证一下发现，这是 OK 的。

既然 Rc 不能 Send，我们无法跨线程使用 Rc<RefCell> 这样的数据，那么使用[支持 Send/Sync 的 Arc](#)呢，使用 Arc<RefCell> 来获得，一个可以在多线程间共享，且可以修改的类型，可以么（[代码](#)）？

```
// RefCell 现在有多个 Arc 持有它，虽然 Arc 是 Send/Sync，但 RefCell 不是 Sync
fn refcell_is_not_sync() {
    let a = Arc::new(RefCell::new(1));
    let b = a.clone();
    let c = a.clone();
    thread::spawn(move || {
        println!("c= {:?}", c);
    });
}
```

不可以。

因为 Arc 内部的数据是共享的，需要支持 Sync 的数据结构，但是 RefCell 不是 Sync，编译失败。所以在多线程情况下，我们只能使用支持 Send/Sync 的 Arc，和 Mutex 一起，构造一个可以在多线程间共享且可以修改的类型（[代码](#)）：

```
use std::{
    sync::{Arc, Mutex},
    thread,
};

// Arc<Mutex<T>> 可以多线程共享且修改数据
fn arc_mutext_is_send_sync() {
    let a = Arc::new(Mutex::new(1));
    let b = a.clone();
    let c = a.clone();
    let handle = thread::spawn(move || {
        let mut g = c.lock().unwrap();
        *g += 1;
    });

    {
        let mut g = b.lock().unwrap();
        *g += 1;
    }

    handle.join().unwrap();
    println!("a= {:?}", a);
}

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

这几段代码建议你都好好阅读和运行一下，对于编译出错的情况，仔细看看编译器给出的错误，会帮助你理解好 Send /Sync trait 以及它们如何保证并发安全。

最后一个标记 trait Unpin，是用于自引用类型的，在后面讲到 Future trait 时，再详细讲这个 trait。

类型转换相关：From / Into/AsRef / AsMut

好，学完了标记 trait，来看看和类型转换相关的 trait。在软件开发的过程中，我们经常需要在某个上下文中，把一种数据结构转换成另一种数据结构。

不过转换有很多方式，看下面的代码，你觉得哪种方式更好呢？

```
// 第一种方法，为每一种转换提供一个方法
// 把字符串 s 转换成 Path
let v = s.to_path();
// 把字符串 s 转换成 u64
let v = s.to_u64();

// 第二种方法，为 s 和要转换的类型之间实现一个 Into<T> trait
// v 的类型根据上下文得出
let v = s.into();
// 或者也可以显式地标注 v 的类型
let v: u64 = s.into();
```

第一种方式，在类型 T 的实现里，要为每一种可能的转换提供一个方法；第二种，我们为类型 T 和类型 U 之间的转换实现一个数据转换 trait，这样可以用同一个方法来实现不同的转换。

显然，第二种方法要更好，因为它符合软件开发的开闭原则（Open–Close Principle），“**软件中的对象（类、模块、函数等等）对扩展是开放的，但是对修改是封闭的**”。

在第一种方式下，未来每次要添加对新类型的转换，都要重新修改类型 T 的实现，而第二种方式，我们只需要添加一个对于数据转换 trait 的新实现即可。

基于这个思路，对值类型的转换和对引用类型的转换，Rust 提供了两套不同的 trait：

- 值类型到值类型的转换：From / Into / TryFrom / TryInto
- 引用类型到引用类型的转换：AsRef / AsMut

From / Into

先看 From 和 Into。这两个 trait 的定义如下：

```
pub trait From<T> {
    fn from(T) -> Self;
}

pub trait Into<T> {
    fn into(self) -> T;
}
```

在实现 From 的时候会自动实现 Into。这是因为：

```
// 实现 From 会自动实现 Into
impl<T, U> Into<U> for T where U: From<T> {
    fn into(self) -> U {
        U::from(self)
    }
}
```

所以大部分情况下，只用实现 From，然后这两种方式都能做数据转换，比如：

```
let s = String::from("Hello world!");
let s: String = "Hello world!".into();
```

这两种方式是等价的，怎么选呢？From 可以根据上下文做类型推导，使用场景更多；而且因为实现了 From 会自动实现 Into，反之不会。所以需要的时候，不要去实现 Into，只要实现 From 就好了。

此外，From 和 Into 还是自反的：把类型 T 的值转换成类型 T，会直接返回。这是因为标准库有如下的实现：

```
// From (以及 Into) 是自反的
impl<T> From<T> for T {
    fn from(t: T) -> T {
        t
    }
}
```

有了 From 和 Into，很多函数的接口就可以变得灵活，比如函数如果接受一个 `IpAddr` 为参数，我们可以使用 Into 让更多的类型可以被这个函数使用，看下面的[代码](#)：

```
use std::net::{IpAddr, Ipv4Addr, Ipv6Addr};

fn print(v: impl Into<IpAddr>) {
    println!("{}: {:?}", v.into());
}

fn main() {
    let v4: Ipv4Addr = "2.2.2.2".parse().unwrap();
    let v6: Ipv6Addr = "::1".parse().unwrap();

    // IPAddr 实现了 From<[u8; 4]>, 转换 IPv4 地址
    print([1, 1, 1, 1]);
    // IPAddr 实现了 From<[u16; 8]>, 转换 IPv6 地址
    print([0xfe80, 0, 0, 0, 0xaede, 0x48ff, 0xfe00, 0x1122]);
    // IPAddr 实现了 From<Ipv4Addr>
    print(v4);
    // IPAddr 实现了 From<Ipv6Addr>
```

```
    print(v6);
}
```

所以，合理地使用 From / Into，可以让代码变得简洁，符合 Rust 可读性强的风格，更符合开闭原则。

注意，如果你的数据类型在转换过程中有可能出现错误，可以使用 TryFrom 和 TryInto，它们的用法和 From / Into 一样，只是 trait 内多了一个关联类型 Error，且返回的结果是 Result<T, Self::Error>。

AsRef / AsMut

搞明白了 From / Into 后，AsRef 和 AsMut 就很好理解了，用于从引用到引用的转换。还是先看它们的定义：

```
pub trait AsRef<T> where T: ?Sized {
    fn as_ref(&self) -> &T;
}
```

```
pub trait AsMut<T> where T: ?Sized {
    fn as_mut(&mut self) -> &mut T;
}
```

在 trait 的定义上，都允许 T 使用大小可变的类型，如 str、[u8] 等。AsMut 除了使用可变引用生成可变引用外，其它都和 AsRef 一样，所以我们重点看 AsRef。

看标准库中打开文件的接口 [std::fs::File::open](#)：

```
pub fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

它的参数 path 是符合 AsRef 的类型，所以，你可以为这个参数传入 String、&str、PathBuf、Path 等类型。而且，当你使用 path.as_ref() 时，会得到一个 &Path。

来写一段代码体验一下 AsRef 的使用和实现（[代码](#)）：

```
#![allow(dead_code)]
enum Language {
    Rust,
    TypeScript,
    Elixir,
    Haskell,
```

```
}

impl AsRef<str> for Language {
    fn as_ref(&self) -> &str {
        match self {
            Language::Rust => "Rust",
            Language::TypeScript => "TypeScript",
            Language::Elixir => "Elixir",
            Language::Haskell => "Haskell",
        }
    }
}

fn print_ref(v: impl AsRef<str>) {
    println!("{}: {}", v.as_ref());
}

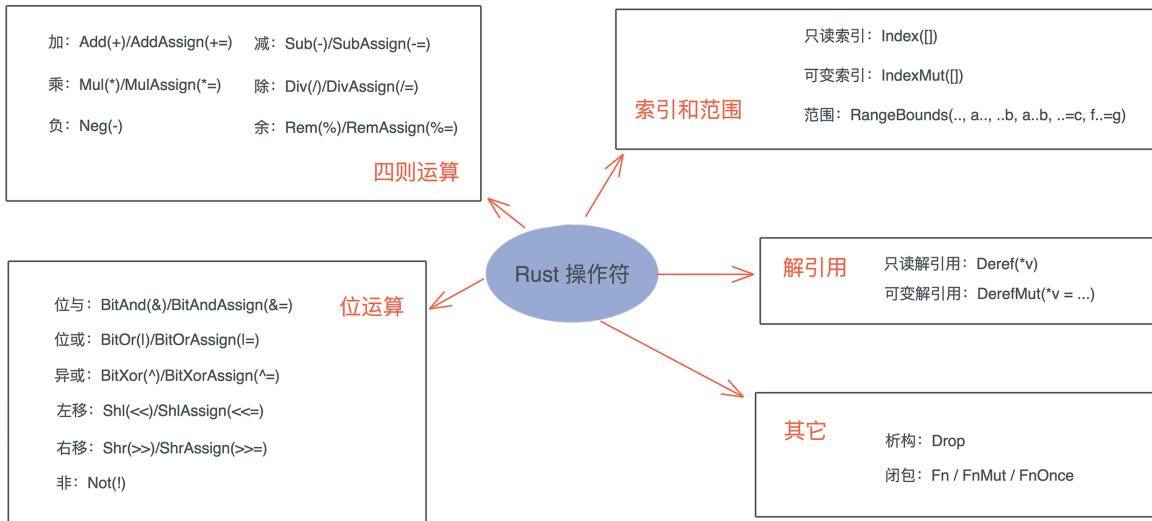
fn main() {
    let lang = Language::Rust;
    // &str 实现了 AsRef<str>
    print_ref("Hello world!");
    // String 实现了 AsRef<str>
    print_ref("Hello world!".to_string());
    // 我们自己定义的 enum 也实现了 AsRef<str>
    print_ref(lang);
}
```

现在对在 Rust 下，如何使用 From / Into / AsRef / AsMut 进行类型间转换，有了深入了解，未来我们还会在实战中使用到这些 trait。

刚才的小例子中要额外说明一下的是，如果你的代码出现 `v.as_ref().clone()` 这样的语句，也就是说你要对 `v` 进行引用转换，然后又得到了拥有所有权的值，那么你应该实现 From，然后做 `v.into()`。

操作符相关：Deref / DerefMut

操作符相关的 trait，上一讲我们已经看到了 Add trait，它允许你重载加法运算符。Rust 为所有的运算符都提供了 trait，你可以为自己的类型重载某些操作符。这里用下图简单概括一下，更详细的信息你可以阅读[官方文档](#)。



极客时间

今天重点要介绍的操作符是 [Deref](#) 和 [DerefMut](#)。来看它们的定义：

```
pub trait Deref {
    // 解引用出来的结果类型
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

可以看到，`DerefMut` “继承”了 `Deref`，只是它额外提供了一个 `deref_mut` 方法，用来获取可变的解引用。所以这里重点学习 `Deref`。

对于普通的引用，解引用很直观，因为它只有一个指向值的地址，从这个地址可以获取到所需要的值，比如下面的例子：

```
let mut x = 42;
let y = &mut x;
// 解引用，内部调用 DerefMut (其实现就是 *self)
*y += 1;
```

但对智能指针来说，拿什么域来解引用就不那么直观了，我们来看之前学过的 `Rc` 是怎么实现 `Deref` 的：

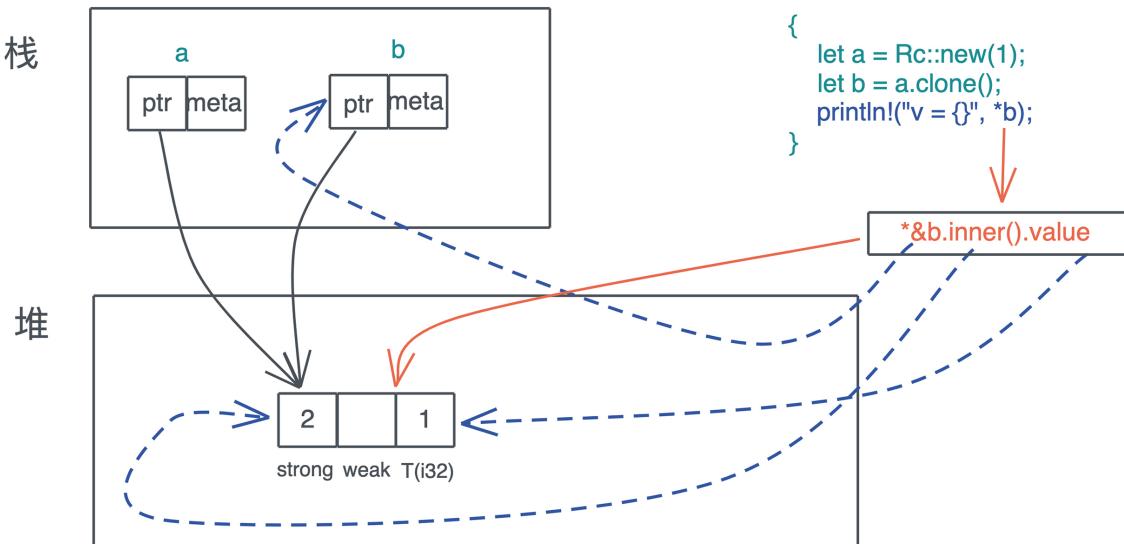
```

impl<T: ?Sized> Deref for Rc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.inner().value
    }
}

```

可以看到，它最终指向了堆上的 RcBox 内部的 value 的地址，然后如果对其解引用的话，得到了 value 对应的值。以下图为例，最终打印出 $v = 1$ 。



极客时间

从图中还可以看到，`Deref` 和 `DerefMut` 是自动调用的，`*b` 会被展开为 `*(b.deref())`。

在 Rust 里，绝大多数智能指针都实现了 `Deref`，我们也可以为自己的数据结构实现 `Deref`。看一个例子（[代码](#)）：

```

use std::ops::{Deref, DerefMut};

#[derive(Debug)]
struct Buffer<T>(Vec<T>);

impl<T> Buffer<T> {
    pub fn new(v: impl Into<Vec<T>>) -> Self {
        Self(v.into())
    }
}

```

```

impl<T> Deref for Buffer<T> {
    type Target = [T];

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

impl<T> DerefMut for Buffer<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

fn main() {
    let mut buf = Buffer::new([1, 3, 2, 4]);
    // 因为实现了 Deref 和 DerefMut，这里 buf 可以直接访问 Vec<T> 的方法
    // 下面这句相当于：(&mut buf).deref_mut().sort()，也就是(&mut buf.0).sort()
    buf.sort();
    println!("buf: {:?}", buf);
}

```

但是在这个例子里，数据结构 Buffer 包裹住了 Vec，但这样一来，原本 Vec 实现了的很多方法，现在使用起来就很不方便，需要用 buf.0 来访问。怎么办？

可以实现 Deref 和 DerefMut，这样在解引用的时候，直接访问到 buf.0，省去了代码的啰嗦和数据结构内部字段的隐藏。

在这段代码里，还有一个值得注意的地方：写 buf.sort() 的时候，并没有做解引用的操作，为什么会相当于访问了 buf.0.sort() 呢？这是因为 sort() 方法第一个参数是 &mut self，此时 Rust 编译器会强制做 Deref/DerefMut 的解引用，所以这相当于 (*(&mut buf)).sort()。

其它：Debug / Display / Default

现在我们对运算符相关的 trait 有了足够的了解，最后来看看其它一些常用的 trait：[Debug](#) / [Display](#) / [Default](#)。

先看 Debug / Display，它们的定义如下：

```

pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

```

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

可以看到，Debug 和 Display 两个 trait 的签名一样，都接受一个 `&self` 和一个 `&mut Formatter`。那为什么要有两个一样的 trait 呢？

这是因为 `Debug` 是为开发者调试打印数据结构所设计的，而 `Display` 是给用户显示数据结构所设计的。这也是为什么 `Debug` trait 的实现可以通过派生宏直接生成，而 `Display` 必须手工实现。在使用的时候，`Debug` 用 `{:?}` 来打印，`Display` 用 `{}` 打印。

最后看 `Default` trait。它的定义如下：

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

`Default` trait 用于为类型提供缺省值。它也可以通过 `derive` 宏 `##[derive(Default)]` 来生成实现，前提是类型中的每个字段都实现了 `Default` trait。在初始化一个数据结构时，我们可以部分初始化，然后剩余的部分使用 `Default::default()`。

`Debug/Display/Default` 如何使用，统一看个例子（[代码](#)）：

```
use std::fmt;
// struct 可以 derive Default，但我们需要所有字段都实现了 Default
#[derive(Clone, Debug, Default)]
struct Developer {
    name: String,
    age: u8,
    lang: Language,
}

// enum 不能 derive Default
#[allow(dead_code)]
#[derive(Clone, Debug)]
enum Language {
    Rust,
    TypeScript,
    Elixir,
    Haskell,
}

// 手工实现 Default
impl Default for Language {
    fn default() -> Self {
        Language::Rust
    }
}
```

```

    }

impl Developer {
    pub fn new(name: &str) -> Self {
        // 用 ..Default::default() 为剩余字段使用缺省值
        Self {
            name: name.to_owned(),
            ..Default::default()
        }
    }
}

impl fmt::Display for Developer {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        writeln!(
            f,
            "{} years old: {} developer",
            self.name, self.age, self.lang
        )
    }
}

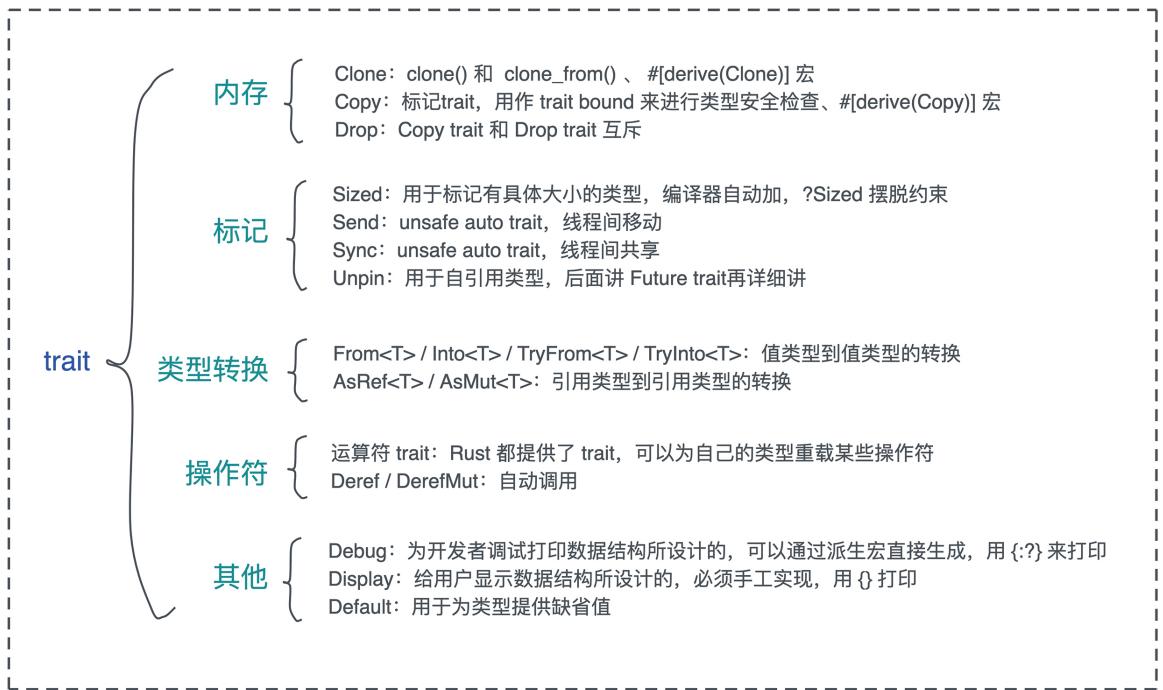
fn main() {
    // 使用 T::default()
    let dev1 = Developer::default();
    // 使用 Default::default(), 但此时类型无法通过上下文推断, 需要提供类型
    let dev2: Developer = Default::default();
    // 使用 T::new
    let dev3 = Developer::new("Tyr");
    println!("dev1: {}\ndev2: {}\ndev3: {}", dev1, dev2, dev3);
}

```

它们实现起来非常简单, 你可以看文中的代码。

小结

今天介绍了内存管理、类型转换、操作符、数据显示等相关的 basic trait, 还介绍了标记 trait, 它是一种特殊的 trait, 主要是用于协助编译器检查类型安全。



在我们使用 Rust 开发时，trait 占据了非常核心的地位。一个设计良好的 trait 可以大大提升整个系统的可用性和扩展性。

很多优秀的第三方库，都围绕着 trait 展开它们的能力，比如上一讲提到的 tower-service 中的 [Service trait](#)，再比如你日后可能会经常使用到的 parser combinator 库 [nom](#) 的 [Parser trait](#)。

因为 trait 实现了延迟绑定。不知道你是否还记得，之前串讲编程基础概念的时候，就谈到了延迟绑定。在软件开发中，延迟绑定会带来极大的灵活性。

从数据的角度看，数据结构是具体数据的延迟绑定，泛型结构是具体数据结构的延迟绑定；从代码的角度看，函数是一组实现某个功能的表达式的延迟绑定，泛型函数是函数的延迟绑定。那么 trait 是什么的延迟绑定呢？

trait 是行为的延迟绑定。我们可以在不知道具体要处理什么数据结构的前提下，先通过 trait 把系统的很多行为约定好。这也是为什么开头解释标准 trait 时，频繁用到了“约定……行为”。

相信通过今天的学习，你能对 trait 有更深刻的认识，在撰写自己的数据类型时，就能根据需要实现这些 trait。

思考题

1. Vec 可以实现 Copy trait 吗？为什么？

2. 在使用 Arc<Mutex> 时，为什么下面这段代码可以直接使用 shared.lock()？

```
use std::sync::{Arc, Mutex};
let shared = Arc::new(Mutex::new(1));
let mut g = shared.lock().unwrap();
*g += 1;
```

3. 有余力的同学可以尝试一下，为下面的 List 类型实现 Index，使得所有的测试都能通过。这段代码使用了 std::collections::LinkedList，你可以参考[官方文档](#)阅读它支持的方法（[代码](#)）：

```
use std::{
    collections::LinkedList,
    ops::{Deref, DerefMut, Index},
};

struct List<T>(LinkedList<T>);

impl<T> Deref for List<T> {
    type Target = LinkedList<T>;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

impl<T> DerefMut for List<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

impl<T> Default for List<T> {
    fn default() -> Self {
        Self(Default::default())
    }
}

impl<T> Index<isize> for List<T> {
    type Output = T;
    fn index(&self, index: isize) -> &Self::Output {
        todo!();
    }
}

#[test]
fn it_works() {
    let mut list: List<u32> = List::default();
    for i in 0..16 {
```

```
    list.push_back(i);

}

assert_eq!(list[0], 0);
assert_eq!(list[5], 5);
assert_eq!(list[15], 15);
assert_eq!(list[16], 0);
assert_eq!(list[-1], 15);
assert_eq!(list[128], 0);
assert_eq!(list[-128], 0);
}
```

今天你已经完成了Rust学习的第14次打卡，坚持学习，如果你觉得有收获，也欢迎分享给身边的朋友，邀TA一起讨论。我们下节课见～

04 | 类型系统：如何在实战中使用泛型编程？

在《架构整洁之道》里 Uncle Bob 说：**架构师的工作不是作出决策，而是尽可能久地推迟决策，在现在不作出重大决策的情况下构建程序，以便以后有足够的信息时再作出决策。**所以，如果我们能通过泛型来推迟决策，系统的架构就可以足够灵活，可以更好地面对未来的变更。

今天，我们就来讲讲如何在实战中使用泛型编程，来延迟决策。如果你对 Rust 的泛型编程掌握得还不够牢靠，建议再温习一下第 12 和 13 讲，也可以阅读 The Rust Programming Language 第 10 章作为辅助。

泛型数据结构的逐步约束

在进入正题之前，我们以标准库的 `BufReader` 结构为例，先简单回顾一下，在定义数据结构和实现数据结构时，如果使用了泛型参数，到底有什么样的好处。

看这个定义的小例子：

```
pub struct BufReader<R> {
    inner: R,
    buf: Box<[u8]>,
    pos: usize,
    cap: usize,
}
```

`BufReader` 对要读取的 `R` 做了一个泛型的抽象。也就是说，`R` 此刻是个 `File`，还是一个 `Cursor`，或者直接是 `Vec`，都不重要。在定义 `struct` 的时候，我们并未对 `R` 做进一步的限制，这是最常用的使用泛型的方式。

到了实现阶段，根据不同的需求，我们可以为 `R` 做不同的限制。这个限制需要细致到什么程度呢？只需要添加刚好满足实现需要的限制即可。

比如在提供 capacity()、buffer() 这些不需要使用 R 的任何特殊能力的时候，可以[不做任何限制](#)：

```
impl<R> BufReader<R> {
    pub fn capacity(&self) -> usize { ... }
    pub fn buffer(&self) -> &[u8] { ... }
}
```

但在实现 new() 的时候，因为使用了 Read trait 里的方法，所以这时需要明确传进来的 R 满足 Read 约束：

```
impl<R: Read> BufReader<R> {
    pub fn new(inner: R) -> BufReader<R> { ... }
    pub fn with_capacity(capacity: usize, inner: R) -> BufReader<R> { ... }
}
```

同样，在实现 Debug 时，也可以要求 R 满足 Debug trait 的约束：

```
impl<R> fmt::Debug for BufReader<R>
where
    R: fmt::Debug
{
    fn fmt(&self, fmt: &mut fmt::Formatter<'_>) -> fmt::Result { ... }
}
```

如果你多花一些时间，把 [bufreader.rs](#) 对接口的所有实现都过一遍，还会发现 BufReader 在实现过程中使用了 Seek trait。

整体而言，impl BufReader 的代码根据不同的约束，分成了不同的代码块。这是一种非常典型的实现泛型代码的方式，我们可以学习起来，在自己的代码中也应用这种方法。

通过使用泛型参数，BufReader 把决策交给使用者。我们在上一讲期中考试的 rgrep 实现中也看到了，在测试和 rgrep 的实现代码中，是如何为 BufReader 提供不同的类型来满足不同的使用场景的。

泛型参数的三种使用场景

泛型参数的使用和逐步约束就简单复习到这里，相信你已经掌握得比较好，我们开始今天的重头戏，来学习实战中如何使用泛型编程。

先看泛型参数，它有三种常见的使用场景：

- 使用泛型参数延迟数据结构的绑定；
- 使用泛型参数和 PhantomData，声明数据结构中不直接使用，但在实现过程中需要用到的类型；
- 使用泛型参数让同一个数据结构对同一个 trait 可以拥有不同的实现。

用泛型参数做延迟绑定

先来看我们已经比较熟悉的，用泛型参数做延迟绑定。在 KV server 的[上篇](#)中，我构建了一个 Service 数据结构：

```
// Service 数据结构
pub struct Service<Store = MemTable> {
    inner: Arc<ServiceInnner<Store>>,
}
```

它使用了一个泛型参数 Store，并且这个泛型参数有一个缺省值 MemTable。指定了泛型参数缺省值的好处是，在使用时，可以不必提供泛型参数，直接使用缺省值。这个泛型参数在随后的实现中可以被逐渐约束：

```
impl<Store> Service<Store> {
    pub fn new(store: Store) -> Self { ... }
}

impl<Store: Storage> Service<Store> {
    pub fn execute(&self, cmd: CommandRequest) -> CommandResponse { ... }
}
```

同样的，在泛型函数中，可以使用 impl Storage 或者 <Store: Storage> 的方式去约束：

```
pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse { ... }
// 等价于
pub fn dispatch<Store: Storage>(cmd: CommandRequest, store: &Store) -> CommandResponse { ... }
```

这种用法，想必你现在已经非常熟悉了，可以在开发中使用泛型参数来对类型进行延迟绑定。

使用泛型参数和幽灵数据（PhantomData）提供额外类型

在熟悉了泛型参数的基本用法后，我来考考你：现在要设计一个 User 和 Product 数据结构，它们都有一个 u64 类型的 id。然而我希望每个数据结构的 id 只能和同种类型的 id 比较，也就是说如果 user.id 和 product.id 比较，编译器就能直接报错，拒绝这种行为。该怎么做呢？

你可以停下来先想一想。

很可能会立刻想到这个办法。先用一个自定义的数据结构 Identifier 来表示 id：

```
pub struct Identifier<T> {
    inner: u64,
}
```

然后，在 User 和 Product 中，各自用 Identifier 来让 Identifier 和自己的类型绑定，达到让不同类型的 id 无法比较的目的。有了这个构想，你可以很快写出这样的代码（[代码](#)）：

```
#[derive(Debug, Default, PartialEq, Eq)]
pub struct Identifier<T> {
    inner: u64,
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct User {
    id: Identifier<Self>,
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct Product {
    id: Identifier<Self>,
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn id_should_not_be_the_same() {
        let user = User::default();
        let product = Product::default();

        // 两个 id 不能比较，因为他们属于不同的类型
        // assert_ne!(user.id, product.id);

        assert_eq!(user.id.inner, product.id.inner);
    }
}
```

然而它无法编译通过。为什么呢？

因为 Identifier 在定义时，并没有使用泛型参数 T，编译器认为 T 是多余的，所以只能把 T 删除掉才能编译通过。但是，删除掉 T，User 和 Product 的 id 就可以比较了，我们就无法实现想要的功能了，怎么办？唉，刚刚还踌躇满志觉得可以用泛型来指点江山，现在面对这么个小问题却万念俱灭？

别急。如果你使用过任何其他支持泛型的语言，无论是 Java、Swift 还是 TypeScript，可能都接触过**Phantom Type（幽灵类型）**的概念。像刚才的写法，Swift / TypeScript 会让其通过，因为它们的编译器会自动把多余的泛型参数当成 Phantom type 来用，比如下面 TypeScript 的例子，可以编译：

```
// NotUsed is allowed
class MyNumber<T, NotUsed> {
    inner: T;
    add: (x: T, y: T) => T;
}
```

但 Rust 对此有洁癖。Rust 并不希望在定义类型时，出现目前还没使用，但未来会被使用的泛型参数，所以 Rust 编译器对此无情拒绝，把门关得严严实实。

不过，别担心，作为过来人，Rust 知道 Phantom Type 的必要性，所以开了一扇叫 [PhantomData](#) 的窗户：让我们可以用 PhantomData 来持有 Phantom Type。PhantomData 中文一般翻译成幽灵数据，这名字透着一股让人不敢亲近的邪魅，但它被广泛用在处理，数据结构定义过程中不需要，但是在实现过程中需要的泛型参数。

我们来试一下：

```
use std::marker::PhantomData;

#[derive(Debug, Default, PartialEq, Eq)]
pub struct Identifier<T> {
    inner: u64,
    _tag: PhantomData<T>,
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct User {
    id: Identifier<Self>,
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct Product {
    id: Identifier<Self>,
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_id_equality() {
        let user1 = User { id: Identifier { inner: 1 } };
        let user2 = User { id: Identifier { inner: 1 } };

        assert_eq!(user1.id, user2.id);
    }
}
```

```

#[test]
fn id_should_not_be_the_same() {
    let user = User::default();
    let product = Product::default();

    // 两个 id 不能比较，因为他们属于不同的类型
    // assert_ne!(user.id, product.id);

    assert_eq!(user.id.inner, product.id.inner);
}
}

```

Bingo! 编译通过！在使用了 PhantomData 后，编译器允许泛型参数 T 的存在。

现在我们确认了：在定义数据结构时，对于额外的、暂时不需要的泛型参数，用 PhantomData 来“拥有”它们，这样可以规避编译器的报错。PhantomData 正如其名，它实际上长度为零，是个 ZST (Zero-Sized Type)，就像不存在一样，唯一作用就是类型的标记。

再来写一个例子，加深对 PhantomData 的理解 ([代码](#))：

```

use std::{
    marker::PhantomData,
    sync::atomic::{AtomicU64, Ordering},
};

static NEXT_ID: AtomicU64 = AtomicU64::new(1);

pub struct Customer<T> {
    id: u64,
    name: String,
    _type: PhantomData<T>,
}

pub trait Free {
    fn feature1(&self);
    fn feature2(&self);
}

pub trait Personal: Free {
    fn advance_feature(&self);
}

impl<T> Free for Customer<T> {
    fn feature1(&self) {
        println!("feature 1 for {}", self.name);
    }

    fn feature2(&self) {
        println!("feature 2 for {}", self.name);
    }
}

```

```

}

impl Personal for Customer<PersonalPlan> {
    fn advance_feature(&self) {
        println!(
            "Dear {}(as our valuable customer {}), enjoy this advanced feature!",
            self.name, self.id
        );
    }
}

pub struct FreePlan;
pub struct PersonalPlan(f32);

impl<T> Customer<T> {
    pub fn new(name: String) -> Self {
        Self {
            id: NEXT_ID.fetch_add(1, Ordering::Relaxed),
            name,
            _type: PhantomData::default(),
        }
    }
}

impl From<Customer<FreePlan>> for Customer<PersonalPlan> {
    fn from(c: Customer<FreePlan>) -> Self {
        Self::new(c.name)
    }
}

/// 订阅成为付费用户
pub fn subscribe(customer: Customer<FreePlan>, payment: f32) -> Customer<PersonalPlan> {
    let _plan = PersonalPlan(payment);
    // 存储 plan 到 DB
    // ...
    customer.into()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_customer() {
        // 一开始是个免费用户
        let customer = Customer::<FreePlan>::new("Tyr".into());
        // 使用免费 feature
        customer.feature1();
        customer.feature2();
        // 用着用着觉得产品不错愿意付费
        let customer = subscribe(customer, 6.99);
        customer.feature1();
        customer.feature1();
        // 付费用户解锁了新技能
        customer.advance_feature();
    }
}

```

在这个例子里，Customer 有个额外的类型 T。

通过类型 T，我们可以将用户分成不同的等级，比如免费用户是 Customer、付费用户是 Customer，免费用户可以转化成付费用户，解锁更多权益。使用 PhantomData 处理这样的状态，可以在编译期做状态的检测，避免运行期检测的负担和潜在的错误。

使用泛型参数来提供多个实现

用泛型参数做延迟绑定、结合PhantomData来提供额外类型，是我们经常能看到的泛型参数的用法。

有时候，对于同一个 trait，我们想要有不同的实现，该怎么办？比如一个方程，它可以是线性方程，也可以是二次方程，我们希望为不同的类型实现不同 Iterator。可以这样做（[代码](#)）：

```
use std::marker::PhantomData;

#[derive(Debug, Default)]
pub struct Equation<IterMethod> {
    current: u32,
    _method: PhantomData<IterMethod>,
}

// 线性增长
#[derive(Debug, Default)]
pub struct Linear;

// 二次增长
#[derive(Debug, Default)]
pub struct Quadratic;

impl Iterator for Equation<Linear> {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.current += 1;
        if self.current >= u16::MAX as u32 {
            return None;
        }

        Some(self.current)
    }
}

impl Iterator for Equation<Quadratic> {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.current += 1;
    }
}
```

```

        if self.current >= u32::MAX {
            return None;
        }

        Some(self.current * self.current)
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_linear() {
        let mut equation = Equation::<Linear>::default();
        assert_eq!(Some(1), equation.next());
        assert_eq!(Some(2), equation.next());
        assert_eq!(Some(3), equation.next());
    }

    #[test]
    fn test_quadratic() {
        let mut equation = Equation::<Quadratic>::default();
        assert_eq!(Some(1), equation.next());
        assert_eq!(Some(4), equation.next());
        assert_eq!(Some(9), equation.next());
    }
}

```

这个代码很好理解，但你可能会有疑问：这样做有什么好处么？为什么不构建两个数据结构 LinearEquation 和 QuadraticEquation，分别实现 Iterator 呢？

的确，对于这个例子，使用泛型的意义并不大，因为 Equation 自身没有很多共享的代码。但如果 Equation，只除了实现 Iterator 的逻辑不一样，其它大量的代码都是相同的，并且未来除了一次方程和二次方程，还会支持三次、四次……，那么，**用泛型数据结构来统一相同的逻辑，用泛型参数的具体类型来处理变化的逻辑**，就非常有必要了。

来看一个真实存在的例子[AsyncProstReader](#)，它来自之前我们在 KV server 里用过的 `async-prost` 库。`async-prost` 库，可以把 TCP 或者其他协议中的 stream 里传输的数据，分成一个个 frame 处理。其中的 `AsyncProstReader` 为 `AsyncDestination` 和 `AsyncFrameDestination` 提供了不同的实现，你可以不用关心它具体做了些什么，只要学习它的接口的设计：

```

/// A marker that indicates that the wrapping type is compatible with `AsyncProstReader` with Prost support.
#[derive(Debug)]
pub struct AsyncDestination;

/// a marker that indicates that the wrapper type is compatible with `AsyncProstReader` with Framed support.
#[derive(Debug)]
pub struct AsyncFrameDestination;

/// A wrapper around an async reader that produces an asynchronous stream of prost-decoded values
#[derive(Debug)]

```

```
pub struct AsyncProstReader<R, T, D> {
    reader: R,
    pub(crate) buffer: BytesMut,
    into: PhantomData<T>,
    dest: PhantomData<D>,
}
```

这个数据结构虽然使用了三个泛型参数，其实数据结构中真正用到的只有一个 R，它可以是一个实现了 AsyncRead 的数据结构（稍后会看到）。另外两个泛型参数 T 和 D，在数据结构定义的时候其实并不需要，只是在数据结构的实现过程中，才需要用到它们的约束。其中，

- T 是从 R 中读取出的数据反序列化出来的类型，在实现时用 prost::Message 约束。
- D 是一个类型占位符，它会根据需要被具体化为 AsyncDestination 或者 AsyncFrameDestination。

类型参数 D 如何使用，我们可以先想像一下。实现 AsyncProstReader 的时候，我们希望在使用 AsyncDestination 时，提供一种实现，而在使用 AsyncFrameDestination 时，提供另一种实现。也就是说，这里的类型参数 D，在 impl 的时候，会被具体化成某个类型。

拿着这个想法，来看 AsyncProstReader 在实现 Stream 时，D 是如何具体化的。这里你不用关心 Stream 具体是什么以及如何实现。实现的代码不重要，重要的是接口（[代码](#)）：

```
impl<R, T> Stream for AsyncProstReader<R, T, AsyncDestination>
where
    T: Message + Default,
    R: AsyncRead + Unpin,
{
    type Item = Result<T, io::Error>;
    fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
        ...
    }
}
```

再看对另外一个对 D 的具体实现：

```
impl<R, T> Stream for AsyncProstReader<R, T, AsyncFrameDestination>
where
    R: AsyncRead + Unpin,
    T: Framed + Default,
{
    type Item = Result<T, io::Error>;
    fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
        ...
    }
}
```

在这个例子里，除了 Stream 的实现不同外，AsyncProstReader 的其它实现都是共享的。所以我们有必要为其增加一个泛型参数 D，使其可以根据不同的 D 的类型，来提供不同的 Stream 实现。

AsyncProstReader 综合使用了泛型的三种用法，感兴趣的话你可以看源代码。如果你无法一下子领悟它的代码，也不必担心。很多时候，这样的高级技巧在阅读代码时用途会更大一些，起码你能搞明白别人的代码为什么这么写。至于自己写的时候是否要这么用，你可以根据自己掌握的程度来决定。

毕竟，我们写代码的首要目标是正确地实现所需要的功能，在正确性的前提下，优雅简洁的表达才有意义。

泛型函数的高级技巧

如果你掌握了泛型数据结构的基本使用方法，那么泛型函数并不复杂，因为在使用泛型参数和对泛型参数进行约束方面是一致的。

之前的课程中，我们已经在函数参数中多次使用泛型参数了，想必你已经有足够的掌握。关于泛型函数，我们讲两点，一是返回值如果想返回泛型参数，该怎么处理？二是对于复杂的泛型参数，该如何声明？

返回值携带泛型参数怎么办？

在 KV server 中，构建 Storage trait 的 get_iter 接口时，我们已经见到了这样的用法：

```
pub trait Storage {  
    ...  
    /// 遍历 HashTable，返回 kv pair 的 Iterator  
    fn get_iter(&self, table: &str) ->  
        Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;  
}
```

对于 get_iter() 方法，并不关心返回值是一个什么样的 Iterator，只要它能够允许我们不断调用 next() 方法，获得一个 Kvpair 的结构，就可以了。在实现里，使用了 trait object。

你也许会有疑惑，为什么不能直接使用 impl Iterator 呢？

```
// 目前 trait 还不支持  
fn get_iter(&self, table: &str) -> Result<impl Iterator<Item = Kvpair>, KvError>;
```

原因是 Rust 目前还不支持在 trait 里使用 impl trait 做返回值：

```
pub trait ImplTrait {
    // 允许
    fn impl_in_args(s: impl Into<String>) -> String {
        s.into()
    }

    // 不允许
    fn impl_as_return(s: String) -> impl Into<String> {
        s
    }
}
```

那么使用泛型参数做返回值呢？可以，但是在实现的时候会很麻烦，你很难在函数中正确构造一个返回泛型参数的语句：

```
// 可以正确编译
pub fn generics_as_return_working(i: u32) -> impl Iterator<Item = u32> {
    std::iter::once(i)
}

// 期待泛型类型，却返回一个具体类型
pub fn generics_as_return_not_working<T: Iterator<Item = u32>>(i: u32) -> T {
    std::iter::once(i)
}
```

那怎么办？很简单，我们可以返回 trait object，它消除了类型的差异，把所有不同的实现 Iterator 的类型都统一到一个相同的 trait object 下：

```
// 返回 trait object
pub fn trait_object_as_return_working(i: u32) -> Box<dyn Iterator<Item = u32>> {
    Box::new(std::iter::once(i))
}
```

明白了这一点，回到刚才 KV server 的 Storage trait：

```
pub trait Storage {
    ...
    /// 遍历 HashTable，返回 kv pair 的 Iterator
    fn get_iter(&self, table: &str) ->
        Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
}
```

现在你是不是更好地理解了，在这个 trait 里，为何我们需要使用 Box<dyn Iterator<Item = Kvpair>> ?

不过使用 trait object 是有额外的代价的，首先这里有一次额外的堆分配，其次动态分派会带来一定的性能损失。

复杂的泛型参数该如何处理？

在泛型函数中，有时候泛型参数可以非常复杂。比如泛型参数是一个闭包，闭包返回一个 Iterator，Iterator 中的 Item 又有某个约束。看下面的示例代码：

```
pub fn consume_iterator<F, Iter, T>(mut f: F)
where
    F: FnMut(i32) -> Iter, // F 是一个闭包，接受 i32，返回 Iter 类型
    Iter: Iterator<Item = T>, // Iter 是一个 Iterator，Item 是 T 类型
    T: std::fmt::Debug, // T 实现了 Debug trait
{
    // 根据 F 的类型，f(10) 返回 iterator，所以可以用 for 循环
    for item in f(10) {
        println!("{}: {:?}", item); // item 实现了 Debug trait，所以可以用 {:?} 打印
    }
}
```

这个代码的泛型参数虽然非常复杂，不过一步步分解，其实并不难理解其实质：

1. 参数 F 是一个闭包，接受 i32，返回 Iter 类型；
2. 参数 Iter 是一个 Iterator，Item 是 T 类型；
3. 参数 T 是一个实现了 Debug trait 的类型。

这么分解下来，我们就可以看到，为何这段代码能够编译通过，同时也可写出合适的测试示例，来测试它：

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_consume_iterator() {
        // 不会 panic 或者出错
        consume_iterator(|i| (0..i).into_iter())
    }
}
```

小结

泛型编程在 Rust 开发中占据着举足轻重的地位，几乎你写的每一段代码都或多或少会使用到泛型有关的结构，比如标准库的 `Vec`、`HashMap<K, V>` 等。当我们自己构建数据结构和函数时要思考，是否使用泛型参数，让代码更加灵活、可扩展性更强。

当然，泛型编程也是一把双刃剑。任何时候，当我们引入抽象，即便能做到零成本抽象，要记得抽象本身也是一种成本。

当我们把代码抽象成函数、把数据结构抽象成泛型结构，即便运行时几乎并无添加额外成本，它还是会带来设计时的成本，如果抽象得不好，还会带来更大的维护上的成本。**做系统设计，我们考虑 ROI (Return On Investment) 时，要把 TCO (Total Cost of Ownership) 也考虑进去。**这也是为什么过度设计的系统和不做设计的系统，它们长期的 TCO 都非常糟糕。

建议你在自己的代码中使用复杂的泛型结构前，最好先做一些准备。

首先，自然是了解使用泛型的场景，以及主要的模式，就像本文介绍的那样；之后，可以多读别人的代码，多看优秀的系统，都是如何使用泛型来解决实际问题的。同时，不要着急把复杂的泛型引入到你自己的系统中，可以先多写一些小的、测试性质的代码，就像文中的那些示例代码一样，从小处着手，去更深入地理解泛型；

有了这些准备打底，最后在你的大型项目中，需要的时候引入自己的泛型数据结构或者函数，去解决实际问题。

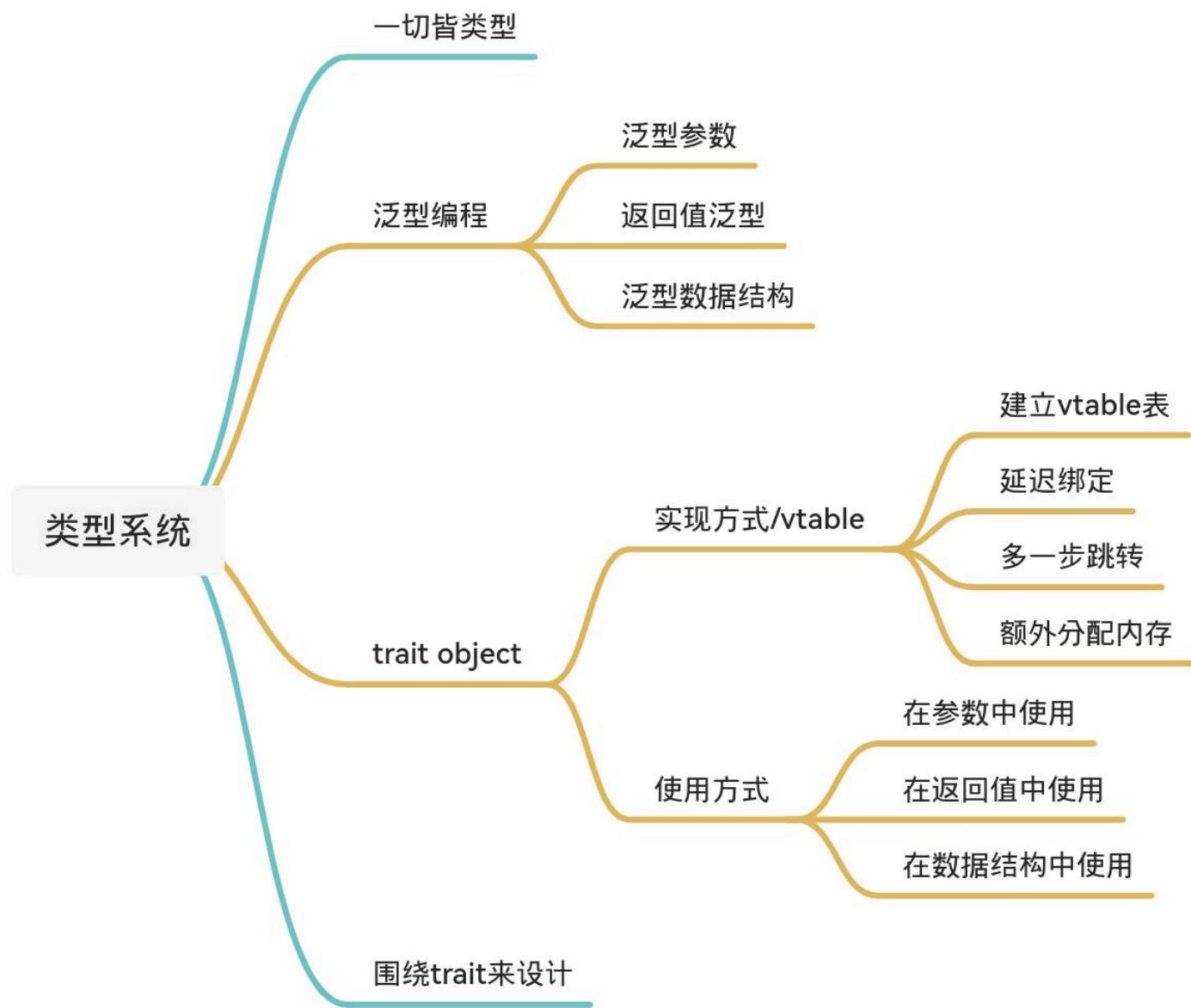
思考题

如果你理解了今天讲的泛型的用法，那么阅读 `futures` 库时，遇到类似的复杂泛型声明，比如说 `StreamExt trait` 的 `for_each_concurrent`，你能搞明白它的参数 `f` 代表什么吗？你该怎么使用这个方法呢？

```
fn for_each_concurrent<Fut, F>(
    self,
    limit: impl Into<Option<usize>>,
    f: F,
) -> ForEachConcurrent<Self, Fut, F>
where
    F: FnMut(Self::Item) -> Fut,
    Fut: Future<Output = ()>,
    Self: Sized,
{
    { ... }
```

今天你已经完成了Rust学习的第23次打卡。如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见。

05 | 类型系统：如何在实战中使用 Trait Object?



今天我们来看看 trait object 是如何在实战中使用的。

照例先来回顾一下 trait object。当我们在运行时想让某个具体类型，只表现出某个 trait 的行为，可以通过将其赋值给一个 dyn T，无论是 &dyn T，还是 Box，还是 Arc，都可以，这里，T 是当前数据类型实现的某个 trait。此时，原有的类型被抹去，Rust 会创建一个 trait object，并为其分配满足该 trait 的 vtable。

你可以再阅读一下[第 13 讲](#)的这个图，来回顾 trait object 是怎么回事：

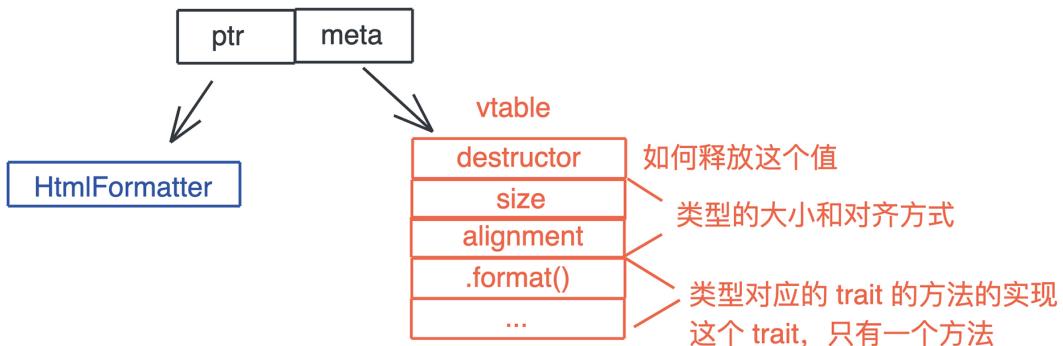
```

let mut text = "Hello world!".to_string();

let formatter: &dyn Formatter = &HtmlFormatter; // 使用 &HtmlFormatter 赋值给
                                                // &Formatter 创建一个 trait object

formatter.format(&mut text); // 调用 trait 的方法

```



在编译 dyn T 时，Rust 会为使用了 trait object 类型的 trait 实现，生成相应的 vtable，放在可执行文件中（一般在 TEXT 或 RODATA 段）：

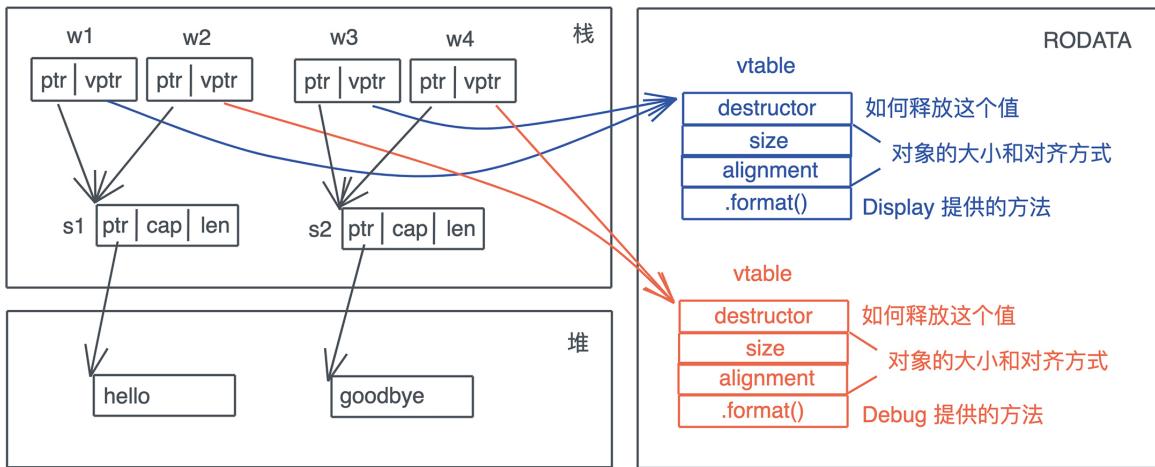
```

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

// Display / Debug trait object for s1
let w1: &dyn Display = &s1;
let w2: &dyn Debug = &s1;

// Display / Debug trait object for s2
let w3: &dyn Display = &s2;
let w4: &dyn Debug = &s2;

```



这样，当 trait object 调用 trait 的方法时，它会先从 vptr 中找到对应的 vtable，进而找到对应的方法来执行。

使用 trait object 的好处是，**当在某个上下文中需要满足某个 trait 的类型，且这样的类型可能有很多，当前上下文无法确定会得到哪一个类型时，我们可以用 trait object 来统一处理行为**。和泛型参数一样，trait object 也是一种延迟绑定，它让决策可以延迟到运行时，从而得到最大的灵活性。

当然，有得必有失。trait object 把决策延迟到运行时，带来的后果是执行效率的打折。在 Rust 里，函数或者方法的执行就是一次跳转指令，而 trait object 方法的执行还多一步，它涉及额外的内存访问，才能得到要跳转的位置再进行跳转，执行的效率要低一些。

此外，如果要把 trait object 作为返回值返回，或者要在线程间传递 trait object，都免不了使用 Box 或者 Arc，会带来额外的堆分配的开销。

好，对 trait object 的回顾就到这里，如果你对它还一知半解，请复习 [13 讲](#)，并且阅读 Rust book 里的：[Using Trait Objects that allow for values of different types](#)。接下来我们讲讲实战中 trait object 的主要使用场景。

在函数中使用 trait object

我们可以在函数的参数或者返回值中使用 trait object。

先来看在参数中使用 trait object。下面的代码构建了一个 Executor trait，并对比做静态分发的 impl Executor、做动态分发的 &dyn Executor 和 Box 这几种不同的参数的使用：

```
use std::{error::Error, process::Command};

pub type BoxedError = Box<dyn Error + Send + Sync>;

pub trait Executor {
    fn run(&self) -> Result<Option<i32>, BoxedError>;
}

pub struct Shell<'a, 'b> {
    cmd: &'a str,
    args: &'b [&'a str],
}

impl<'a, 'b> Shell<'a, 'b> {
    pub fn new(cmd: &'a str, args: &'b [&'a str]) -> Self {
        Self { cmd, args }
    }
}

impl<'a, 'b> Executor for Shell<'a, 'b> {
    fn run(&self) -> Result<Option<i32>, BoxedError> {
        let output = Command::new(self.cmd).args(self.args).output()?;
        Ok(output.status.code())
    }
}

/// 使用泛型参数
pub fn execute_generics(cmd: &impl Executor) -> Result<Option<i32>, BoxedError> {
    cmd.run()
}

/// 使用 trait object: &dyn T
pub fn execute_trait_object(cmd: &dyn Executor) -> Result<Option<i32>, BoxedError> {
    cmd.run()
}

/// 使用 trait object: Box<dyn T>
pub fn execute_boxed_trait_object(cmd: Box<dyn Executor>) -> Result<Option<i32>, BoxedError> {
    cmd.run()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn shell_shall_work() {
        let cmd = Shell::new("ls", &[]);
        let result = cmd.run().unwrap();
    }
}
```

```

        assert_eq!(result, Some(0));
    }

#[test]
fn execute_shall_work() {
    let cmd = Shell::new("ls", &[]);

    let result = execute_generics(&cmd).unwrap();
    assert_eq!(result, Some(0));
    let result = execute_trait_object(&cmd).unwrap();
    assert_eq!(result, Some(0));
    let boxed = Box::new(cmd);
    let result = execute_boxed_trait_object(boxed).unwrap();
    assert_eq!(result, Some(0));
}
}

```

其中，`impl Executor` 使用的是泛型参数的简化版本，而 `&dyn Executor` 和 `Box` 是 trait object，前者在栈上，后者分配在堆上。值得注意的是，分配在堆上的 trait object 也可以作为返回值返回，比如示例中的 `Result<Option, BoxedError>` 里使用了 trait object。

这里为了简化代码，我使用了 `type` 关键字创建了一个 `BoxedError` 类型，是 `Box<dyn Error + Send + Sync + 'static>` 的别名，它是 `Error trait` 的 trait object，除了要求类型实现了 `Error trait` 外，它还有额外的约束：类型必须满足 `Send / Sync` 这两个 trait。

在参数中使用 trait object 比较简单，再来看一个实战中的[例子](#)巩固一下：

```

pub trait CookieStore: Send + Sync {
    fn set_cookies(
        &self,
        cookie_headers: &mut dyn Iterator<Item = &HeaderValue>,
        url: &Url
    );
    fn cookies(&self, url: &Url) -> Option<HeaderValue>;
}

```

这是我们之前使用过的 `reqwest` 库中的一个处理 `CookieStore` 的 trait。在 `set_cookies` 方法中使用了 `&mut dyn Iterator` 这样一个 trait object。

在函数返回值中使用

好，相信你对在参数中如何使用 trait object 已经没有什么问题了，我们再看返回值中使用 trait object，这是 trait object 使用频率比较高的场景。

之前已经出现过很多次了。比如上一讲已经详细介绍的，为何 KV server 里的 Storage trait 不能使用泛型参数来处理返回的 iterator，只能用 Box：

```
pub trait Storage: Send + Sync + 'static {  
    ...  
    /// 遍历 HashTable, 返回 kv pair 的 Iterator  
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;  
}
```

再来看一些实战中会遇到的例子。

首先是 `async_trait`。它是一种特殊的 trait，方法中包含 async fn。目前 Rust 并不支持 trait 中使用 async fn，一个变通的方法是使用 `async_trait` 宏。

在 get hands dirty 系列中，我们就使用过 `async trait`。下面是第 6 讲 SQL 查询工具数据源的获取中定义的 Fetch trait：

```
// Rust 的 async trait 还没有稳定，可以用 async_trait 宏  
#[async_trait]  
pub trait Fetch {  
    type Error;  
    async fn fetch(&self) -> Result<String, Self::Error>;  
}
```

这里宏展开后，类似于：

```
pub trait Fetch {  
    type Error;  
    fn fetch<'a>(&'a self) ->  
        Result<Pin<Box<dyn Future<Output = String> + Send + 'a>>, Self::Error>;  
}
```

它使用了 trait object 作为返回值。这样，不管 `fetch()` 的实现，返回什么样的 Future 类型，都可以被 trait object 统一起来，调用者只需要按照正常 Future 的接口使用即可。

我们再看一个 `snow` 下的 `CryptoResolver` 的例子：

```
/// An object that resolves the providers of Noise crypto choices  
pub trait CryptoResolver {  
    /// Provide an implementation of the Random trait or None if none available.  
}
```

```

fn resolve_rng(&self) -> Option<Box<dyn Random>>;
/// Provide an implementation of the Dh trait for the given DHChoice or None if unavailable.
fn resolve_dh(&self, choice: &DHChoice) -> Option<Box<dyn Dh>>;
/// Provide an implementation of the Hash trait for the given HashChoice or None if unavailable.
fn resolve_hash(&self, choice: &HashChoice) -> Option<Box<dyn Hash>>;
/// Provide an implementation of the Cipher trait for the given CipherChoice or None if unavailable.
fn resolve_cipher(&self, choice: &CipherChoice) -> Option<Box<dyn Cipher>>;
/// Provide an implementation of the Kem trait for the given KemChoice or None if unavailable
#[cfg(feature = "hfs")]
fn resolve_kem(&self, _choice: &KemChoice) -> Option<Box<dyn Kem>> {
    None
}
}

```

这是一个处理 [Noise Protocol](#) 使用何种加密算法的一个 trait。这个 trait 的每个方法，都返回一个 trait object，每个 trait object 都提供加密算法中所需要的不同的能力，比如随机数生成算法（Random）、DH 算法（Dh）、哈希算法（Hash）、对称加密算法（Cipher）和密钥封装算法（Kem）。

所有这些，都有一系列的具体的算法实现，通过 CryptoResolver trait，可以得到当前使用的某个具体算法的 trait object，**这样，在处理业务逻辑时，我们不用关心当前究竟使用了什么算法，就能根据这些 trait object 构筑相应的实现**，比如下面的 generate_keypair：

```

pub fn generate_keypair(&self) -> Result<Keypair, Error> {
    // 拿到当前的随机数生成算法
    let mut rng = self.resolver.resolve_rng().ok_or(InitStage::GetRngImpl)?;
    // 拿到当前的 DH 算法
    let mut dh = self.resolver.resolve_dh(&self.params.dh).ok_or(InitStage::GetDhlImpl)?;
    let mut private = vec![0u8; dh.priv_len()];
    let mut public = vec![0u8; dh.pub_len()];
    // 使用随机数生成器 和 DH 生成密钥对
    dh.generate(&mut *rng);

    private.copy_from_slice(dh.privkey());
    public.copy_from_slice(dh.pubkey());

    Ok(Keypair { private, public })
}

```

说句题外话，如果你想更好地学习 trait 和 trait object 的使用，snow 是一个很好的学习资料。你可以顺着 CryptoResolver 梳理它用到的这几个主要的加密算法相关的 trait，看看别人是怎么定义 trait、如何把各个 trait 关联起来，以及最终如何把 trait 和核心数据结构联系起来的（小提示：Builder 以及 HandshakeState）。

在数据结构中使用 trait object

了解了在函数中是如何使用 trait object 的，接下来我们再看看在数据结构中，如何使用 trait object。

继续以 snow 的代码为例，看 HandshakeState这个用于处理 Noise Protocol 握手协议的数据结构，用到了哪些 trait object ([代码](#))：

```
pub struct HandshakeState {  
    pub(crate) rng: Box<dyn Random>,  
    pub(crate) symmetricstate: SymmetricState,  
    pub(crate) cipherstates: CipherStates,  
    pub(crate) s: Toggle<Box<dyn Dh>>,  
    pub(crate) e: Toggle<Box<dyn Dh>>,  
    pub(crate) fixed_ephemeral: bool,  
    pub(crate) rs: Toggle<[u8; MAXDHLEN]>,  
    pub(crate) re: Toggle<[u8; MAXDHLEN]>,  
    pub(crate) initiator: bool,  
    pub(crate) params: NoiseParams,  
    pub(crate) psks: [Option<[u8; PSKLEN]>; 10],  
#[cfg(feature = "hfs")]  
    pub(crate) kem: Option<Box<dyn Kem>>,  
#[cfg(feature = "hfs")]  
    pub(crate) kem_re: Option<[u8; MAXKEMPUBLEN]>,  
    pub(crate) my_turn: bool,  
    pub(crate) message_patterns: MessagePatterns,  
    pub(crate) pattern_position: usize,  
}
```

你不需要了解 Noise protocol，也能够大概可以明白这里 Random、Dh 以及 Kem 三个 trait object 的作用：它们为握手期间使用的加密协议提供最大的灵活性。

想想看，如果这里不用 trait object，这个数据结构该怎么处理？

可以用泛型参数，也就是说：

```
pub struct HandshakeState<R, D, K>  
where  
    R: Random,  
    D: Dh,  
    K: Kem  
{  
    ...  
}
```

这是我们大部分时候处理这样的数据结构的选择。但是，过多的泛型参数会带来两个问题：首先，代码实现过程中，所有涉及的接口都变得非常臃肿，你在使用 `HandshakeState<R, D, K>` 的任何地方，都必须带着这几个泛型参数以及它们的约束。其次，这些参数所有被使用到的情况，组合起来，会生成大量的代码。

而使用 trait object，我们在牺牲一点性能的前提下，消除了这些泛型参数，实现的代码更干净清爽，且代码只会有一份实现。

在数据结构中使用 trait object 还有一种很典型的场景是，闭包。

因为在 Rust 中，闭包都是以匿名类型的方式出现，我们无法直接在数据结构中使用其类型，只能用泛型参数。而对闭包使用泛型参数后，如果捕获的数据太大，可能造成数据结构本身太大；但有时，我们并不在意一点点性能损失，更愿意让代码处理起来更方便。

比如用于做 RBAC 的库 [oso](#) 里的 `AttributeGetter`，它包含了一个 Fn：

```
#[derive(Clone)]
pub struct AttributeGetter(
    Arc<dyn Fn(&Instance, &mut Host) -> crate::Result<PolarValue> + Send + Sync>,
);
```

如果你对在 Rust 中如何实现 Python 的 `getattr` 感兴趣，可以看看 [oso](#) 的代码。

再比如做交互式 CLI 的 [dialoguer](#) 的 `Input`，它的 `validator` 就是一个 FnMut：

```
pub struct Input<'a, T> {
    prompt: String,
    default: Option<T>,
    show_default: bool,
    initial_text: Option<String>,
    theme: &'a dyn Theme,
    permit_empty: bool,
    validator: Option<Box<dyn FnMut(&T) -> Option<String> + 'a>>,
    #[cfg(feature = "history")]
    history: Option<&'a mut dyn History<T>>,
}
```

用 trait object 处理 KV server 的 Service 结构

好，到这里用 trait object 做动态分发的几个场景我们就介绍完啦，来写段代码练习一下。

就用之前写的 KV server 的 Service 结构来趁热打铁，我们尝试对它做个处理，使其内部使用 trait object。

其实对于 KV server 而言，使用泛型是更好的选择，因为此处泛型并不会造成太多的复杂性，我们也不希望丢掉哪怕一点点性能。然而，出于学习的目的，我们可以看看如果 store 使用 trait object，代码会变成什么样子。你自己可以先尝试一下，再来看下面的示例（[代码](#)）：

```
use std::{error::Error, sync::Arc};

// 定义类型，让 KV server 里的 trait 可以被编译通过
pub type KvError = Box;
pub struct Value(i32);
pub struct Kvpair(i32, i32);

/// 对存储的抽象，我们不关心数据存在哪儿，但需要定义外界如何和存储打交道
pub trait Storage: Send + Sync + 'static {
    fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
    fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError>;
    fn contains(&self, table: &str, key: &str) -> Result<bool, KvError>;
    fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
    fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError>;
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
}

// 使用 trait object，不需要泛型参数，也不需要 ServiceInnner 了
pub struct Service {
    pub store: Arc<dyn Storage>,
}

// impl 的代码略微简单一些
impl Service {
    pub fn new<S: Storage>(store: S) -> Self {
        Self {
            store: Arc::new(store),
        }
    }
}

// 实现 trait 时也不需要带着泛型参数
impl Clone for Service {
    fn clone(&self) -> Self {
        Self {
            store: Arc::clone(&self.store),
        }
    }
}
```

从这段代码中可以看到，通过牺牲一点性能，我们让代码整体撰写和使用起来方便了不少。

小结

无论是上一讲的泛型参数，还是今天的 trait object，都是 Rust 处理多态的手段。当系统需要使用多态来解决复杂多变的需求，让同一个接口可以展现不同的行为时，我们要决定究竟是编译时的静态分发更好，还是运行时的动态分发更好。

一般情况下，作为 Rust 开发者，我们不介意泛型参数带来的稍微复杂的代码结构，愿意用开发时的额外付出，换取运行时的高效；但有时候，当泛型参数过多，导致代码出现了可读性问题，或者运行效率并不是主要矛盾的时候，我们可以通过使用 trait object 做动态分发，来降低代码的复杂度。

具体看，在有些情况，我们不太容易使用泛型参数，比如希望函数返回某个 trait 的实现，或者数据结构中某些参数在运行时的组合过于复杂，比如上文提到的 HandshakeState，此时，使用 trait object 是更好的选择。

思考题

期中测试中我给出的 [rgrep 的代码](#)，如果把 StrategyFn 的接口改成使用 trait object：

```
/// 定义类型，这样，在使用时可以简化复杂类型的书写
pub type StrategyFn = fn(&Path, &mut dyn BufRead, &Regex, &mut dyn Write) -> Result<(), GrepError>;
```

你能把实现部分修改，使测试通过么？对比修改前后的代码，你觉得对 rgrep，哪种实现更好？为什么？

今天你完成了Rust学习的第24次打卡。如果你觉得有收获，也欢迎分享给你身边的朋友，邀他一起讨论。我们下节课见。

延伸阅读

我们总说 trait object 性能会差一些，因为需要从 vtable 中额外加载对应的方法的地址，才能跳转执行。那么这个性能差异究竟有多大呢？网上有人说调用 trait object 的方法，性能会比直接调用类型的方法差一个数量级，真的么？

我用 criterion 做了一个简单的测试，测试的 trait 使用的就是我们这一讲使用的 Executor trait。测试代码如下（你可以访问 [GitHub repo](#) 中这一讲的代码）：

```
use advanced_trait_objects::{
    execute_boxed_trait_object, execute_generics, execute_trait_object, Shell,
};
use criterion::{black_box, criterion_group, criterion_main, Criterion};
```

```

pub fn generics_benchmark(c: &mut Criterion) {
    c.bench_function("generics", |b| {
        b.iter(|| {
            let cmd = Shell::new("ls", &[]);
            execute_generics(black_box(&cmd)).unwrap();
        })
    });
}

pub fn trait_object_benchmark(c: &mut Criterion) {
    c.bench_function("trait object", |b| {
        b.iter(|| {
            let cmd = Shell::new("ls", &[]);
            execute_trait_object(black_box(&cmd)).unwrap();
        })
    });
}

pub fn boxed_object_benchmark(c: &mut Criterion) {
    c.bench_function("boxed object", |b| {
        b.iter(|| {
            let cmd = Box::new(Shell::new("ls", &[]));
            execute_boxed_trait_object(black_box(cmd)).unwrap();
        })
    });
}

criterion_group!(
    benches,
    generics_benchmark,
    trait_object_benchmark,
    boxed_object_benchmark
);
criterion_main!(benches);

```

为了不让实现本身干扰接口调用的速度，我们在 trait 的方法中什么也不做，直接返回：

```

impl<'a, 'b> Executor for Shell<'a, 'b> {
    fn run(&self) -> Result<Option<i32>, BoxedError> {
        // let output = Command::new(self.cmd).args(self.args).output()?;
        // Ok(output.status.code())
        Ok(Some(0))
    }
}

```

测试结果如下：

```

generics          time: [3.0995 ns 3.1549 ns 3.2172 ns]
                  change: [-96.890% -96.810% -96.732%] (p = 0.00 < 0.05)
                  Performance has improved.

```

```
Found 5 outliers among 100 measurements (5.00%)
```

```
4 (4.00%) high mild
```

```
1 (1.00%) high severe
```

```
trait object      time: [4.0348 ns 4.0934 ns 4.1552 ns]
```

```
change: [-96.024% -95.893% -95.753%] (p = 0.00 < 0.05)
```

```
Performance has improved.
```

```
Found 8 outliers among 100 measurements (8.00%)
```

```
3 (3.00%) high mild
```

```
5 (5.00%) high severe
```

```
boxed object      time: [65.240 ns 66.473 ns 67.777 ns]
```

```
change: [-67.403% -66.462% -65.530%] (p = 0.00 < 0.05)
```

```
Performance has improved.
```

```
Found 2 outliers among 100 measurements (2.00%)
```

可以看到，使用泛型做静态分发最快，平均 3.15ns；使用 &dyn Executor 平均速度 4.09ns，要慢 30%；而使用 Box 平均速度 66.47ns，慢了足足 20 倍。可见，额外的内存访问并不是 trait object 的效率杀手，有些场景下为了使用 trait object 不得不做的额外的堆内存分配，才是主要的效率杀手。

那么，这个性能差异重要么？

在回答这个问题之前，我们把 run() 方法改回来：

```
impl<'a, 'b> Executor for Shell<'a, 'b> {
    fn run(&self) -> Result<Option<i32>, BoxedError> {
        let output = Command::new(self.cmd).args(self.args).output()?;
        Ok(output.status.code())
    }
}
```

我们知道 Command 的执行速度比较慢，但是想再看看，对于执行效率低的方法，这个性能差异是否重要。

新的测试结果不出所料：

```
generics      time: [4.6901 ms 4.7267 ms 4.7678 ms]
```

```
change: [+145694872% +148496855% +151187366%] (p = 0.00 < 0.05)
```

```
Performance has regressed.
```

```
Found 7 outliers among 100 measurements (7.00%)
```

```
3 (3.00%) high mild
```

```
4 (4.00%) high severe
```

```
trait object      time: [4.7452 ms 4.7912 ms 4.8438 ms]
```

```
change: [+109643581% +113478268% +116908330%] (p = 0.00 < 0.05)
```

```
Performance has regressed.
```

```
Found 7 outliers among 100 measurements (7.00%)
```

```
4 (4.00%) high mild
```

```
3 (3.00%) high severe
```

```
boxed object      time: [4.7867 ms 4.8336 ms 4.8874 ms]
```

```
change: [+6935303% +7085465% +7238819%] (p = 0.00 < 0.05)
```

```
Performance has regressed.
```

```
Found 8 outliers among 100 measurements (8.00%)
```

```
4 (4.00%) high mild
```

```
4 (4.00%) high severe
```

因为执行一个 Shell 命令的效率实在太低，到毫秒的量级，虽然 generics 依然最快，但使用 &dyn Executor 和 Box 也不过只比它慢了 1% 和 2%。

所以，如果是那种执行效率在数百纳秒以内的函数，是否使用 trait object，尤其是 boxed trait object，性能差别会比较明显；但当函数本身的执行需要数微妙到数百微妙时，性能差别就很小了；到了毫秒的量级，性能的差别几乎无关紧要。

总的来说，大部分情况，我们在撰写代码的时候，不必太在意 trait object 的性能问题。如果你实在在意关键路径上 trait object 的性能，那么先尝试看能不能不要做额外的堆内存分配。

06 | 类型系统：如何围绕 Trait 来设计和架构系统？

Trait, trait, trait, 怎么又是 trait? how old are you?

希望你还没有厌倦我们没完没了地聊关于 trait 的话题。因为 trait 在 Rust 开发中的地位，怎么吹都不为过。

其实不光是 Rust 中的 trait，任何一门语言，和接口处理相关的概念，都是那门语言在使用过程中最重要的概念。**软件开发的整个行为，基本上可以说是不断创建和迭代接口，然后在这些接口上进行实现的过程。**

在这个过程中，有些接口是标准化的，雷打不动，就像钢筋、砖瓦、螺丝、钉子、插座等这些材料一样，无论要构筑的房子是什么样子的，这些标准组件的接口在确定下来后，都不会改变，它们就像 Rust 语言标准库中的标准 trait 一样。

而有些接口是跟构造的房子息息相关的，比如门窗、电器、家具等，它们就像你要设计的系统中的 trait 一样，可以把系统的各个部分联结起来，最终呈现给用户一个完整的使用体验。

之前讲了 trait 的基础知识，也介绍了如何在实战中使用 trait 和 trait object。今天，我们再花一讲的时间，来看看如何围绕着 trait 来设计和架构系统。

由于在讲架构和设计时，不免要引入需求，然后我需要解释这需求的来龙去脉，再提供设计思路，再介绍 trait 在其中的作用，但这样下来，一堂课的内容能讲好一个系统设计就不错了。所以我们换个方式，把之前设计过的系统捋一下，重温它们的 trait 设计，看看其中的思路以及取舍。

用 trait 让代码自然舒服好用

在第 5 讲, thumbor 的项目里, 我设计了一个 SpecTransform trait, 通过它可以统一处理任意类型的、描述我们希望如何处理图片的 spec:

```
// 一个 spec 可以包含上述的处理方式之一 (这是 protobuf 定义)
message Spec {
    oneof data {
        Resize resize = 1;
        Crop crop = 2;
        Flipv flipv = 3;
        Fliph fliph = 4;
        Contrast contrast = 5;
        Filter filter = 6;
        Watermark watermark = 7;
    }
}
```

SpecTransform trait 的定义如下 ([代码](#)) :

```
// SpecTransform: 未来如果添加更多的 spec, 只需要实现它即可
pub trait SpecTransform<T> {
    // 对图片使用 op 做 transform
    fn transform(&mut self, op: T);
}
```

它可以用来对图片使用某个 spec 进行处理。

但如果你阅读 GitHub 上的源码, 你可能会发现一个没用到的文件 [imageproc.rs](#) 中类似的 trait ([代码](#)) :

```
pub trait ImageTransform {
    fn transform(&self, image: &mut PhotonImage);
}
```

这个 trait 是第一版的 trait。我依旧保留着它, 就是想在此展示一下 trait 设计上的取舍。

当你审视这段代码的时候会不会觉得, 这个 trait 的设计有些草率? 因为如果传入的 image 来自不同的图片处理引擎, 而某个图片引擎提供的 image 类型不是 PhotonImage, 那这个接口不就无法使用了么?

hmm，这是个设计上的大问题啊。想想看，以目前所学的知识，怎么解决这个问题呢？什么可以帮助我们延迟 image 是否必须是 PhotonImage 的决策呢？

对，泛型。我们可以使用泛型 trait 修改一下刚才那段代码：

```
// 使用 trait 可以统一处理的接口，以后无论增加多少功能，只需要加新的 Spec，然后实现 ImageTransform 接口
pub trait ImageTransform<Image> {
    fn transform(&self, image: &mut Image);
}
```

把传入的 image 类型抽象成泛型类型之后，延迟了图片类型判断和支持的决策，可用性更高。

但如果你继续对比现在的 ImageTransform 和之前写的 SpecTransform，会发现，它们实现 trait 的数据结构和用在 trait 上的泛型参数，正好掉了个个。

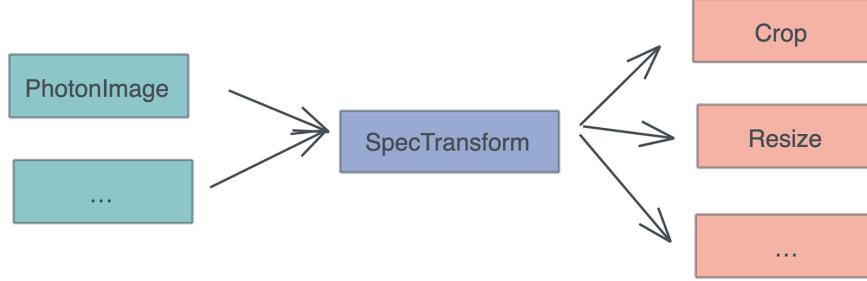
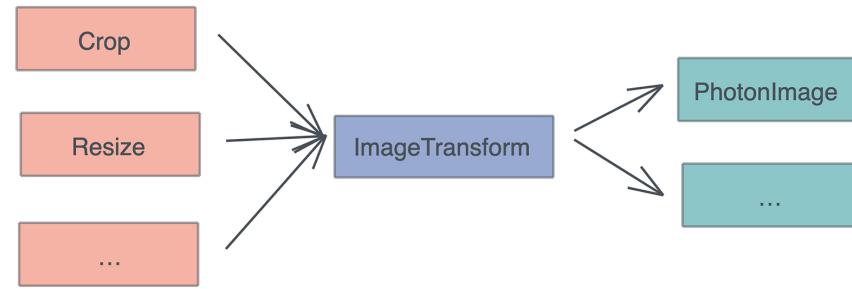
你看，PhotonImage 下对于 Contrast 的 ImageTransform 的实现：

```
impl ImageTransform<PhotonImage> for Contrast {
    fn transform(&self, image: &mut Image) {
        effects::adjust_contrast(image, self.contrast);
    }
}
```

而同样的，PhotonImage 下对 Contract 的 SpecTransform 的实现：

```
impl SpecTransform<&Contrast> for Photon {
    fn transform(&mut self, op: &Contrast) {
        effects::adjust_contrast(&mut self.0, op.contrast);
    }
}
```

这两种方式基本上等价，但一个围绕着 Spec 展开，一个围绕着 Image 展开：



极客时间

那么，哪种设计更好呢？

其实二者并没有功能上或者性能上的优劣。

那为什么我选择了 `SpecTransform` 的设计呢？在第一版的设计我还没有考虑 `Engine` 的时候，是以 `Spec` 为中心的；但在把 `Engine` 考虑进去后，我以 `Engine` 为中心重新做了设计，这样做的好处是，开发新的 `Engine` 的时候，`SpecTransform` trait 用起来更顺手，更自然一些。

嗯，顺手，自然。接口的设计一定要关注使用者的体验，一个使用起来感觉自然顺手舒服的接口，就是更好的接口。**因为这意味着使用的时候，代码可以自然而然写出来，而无需看文档。**

比如同样是 Python 代码：

```
df[df["age"] > 10]
```

就要比：

```
df.filter(df.col("age").gt(10))
```

要更加自然舒服。前面的代码，你看一眼别人怎么用，自己就很快能写出来，而后者，你需要先搞清楚 filter 函数是怎么回事，以及 col()、gt() 这两个方法如何使用。

我们再来看两段 Rust 代码。这行使用了 From/Into trait 的代码：

```
let url = generate_url_with_spec(image_spec.into());
```

就要比：

```
let data = image_spec.encode_to_vec();
let s = encode_config(data, URL_SAFE_NO_PAD);
let url = generate_url_with_spec(s);
```

要简洁、自然得多。它把实现细节都屏蔽了起来，只让用户关心他们需要关心的逻辑。

所以，我们在设计 trait 的时候，除了关注功能，还要注意是否好用、易用。这也是为什么我们在介绍 KV server 的时候，不断强调，trait 在设计结束之后，不要先着急撰写实现 trait 的代码，而是最好先写一些对于 trait 使用的测试代码。

你在写这些测试代码的使用体验，就是别人在使用你的 trait 构建系统时的真实体验，如果它用起来别扭、啰嗦，不看文档就不容易用对，那这个 trait 本身还有待进一步迭代。

用 trait 做桥接

在软件开发的绝大多数时候，我们都不会从零到一完完全全设计和构建系统的所有部分。就像盖房子，不可能从一杯土、一块瓦片开始打造。我们需要依赖生态系统中已有的组件。

作为架构师，你的职责是在生态系统中找到合适的组件，连同你自己打造的部分，一起粘合起来，形成一个产品。所以，你会遇到那些接口与你预期不符的组件，可是自己又无法改变那些组件来让接口满足你的预期，怎么办？

此刻，我们需要桥接。

就像要用的电器是二相插口，而附近墙上的插座只有三相插口，我们总不能修改电器或者墙上的插座，使其满足对方吧？正确的做法是购置一个多项插座来桥接二者。

在 Rust 里，桥接的工作可以通过函数来完成，但最好通过 trait 来桥接。继续看[第 5 讲](#) thumbor 里的另一个 trait Engine ([代码](#))：

```
// Engine trait: 未来可以添加更多的 engine, 主流程只需要替换 engine
pub trait Engine {
    // 对 engine 按照 specs 进行一系列有序的处理
    fn apply(&mut self, specs: &[Spec]);
    // 从 engine 中生成目标图片，注意这里用的是 self, 而非 self 的引用
    fn generate(self, format: ImageOutputFormat) -> Vec<u8>;
}
```

通过 Engine 这个 trait，我们把第三方的库 photon 和自己设计的 Image Spec 连接起来，使得我们不用关心 Engine 背后究竟是什么，只需要调用 apply 和 generate 方法即可：

```
// 使用 image engine 处理
let mut engine: Photon = data
    .try_into()
    .map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
engine.apply(&spec.specs);
let image = engine.generate(ImageOutputFormat::Jpeg(85));
```

这段代码中，由于之前为 Photon 实现了 TryFrom，所以可以直接调用 try_into() 来得到一个 photon engine：

```
// 从 Bytes 转换成 Photon 结构
impl TryFrom<Bytes> for Photon {
    type Error = anyhow::Error;

    fn try_from(data: Bytes) -> Result<Self, Self::Error> {
        Ok(Self(open_image_from_bytes(&data)?))
    }
}
```

就桥接 thumbor 代码和 photon crate 而言，Engine 表现良好，它让我们不但很容易使用 photon crate，还可以很方便在未来需要的时候替换掉 photon crate。

不过，Engine 在构造时，所做的桥接还是不够直观和自然，如果不仔细看代码或者文档，使用者可能并不清楚，第3行代码，如何通过 TryFrom/TryInto 得到一个实现了 Engine 的结构。从这个使用体验来看，我们会希望通过使用 Engine trait，任何一个图片引擎都可以统一地创建 Engine 结构。怎么办？

可以为这个 trait 添加一个缺省的 create 方法：

```
// Engine trait: 未来可以添加更多的 engine, 主流程只需要替换 engine
pub trait Engine {
    // 生成一个新的 engine
    fn create<T>(data: T) -> Result<Self>
    where
        Self: Sized,
        T: TryInto<Self>,
    {
        data.try_into()
            .map_err(|_| anyhow!("failed to create engine"))
    }
    // 对 engine 按照 specs 进行一系列有序的处理
    fn apply(&mut self, specs: &[Spec]);
    // 从 engine 中生成目标图片, 注意这里用的是 self, 而非 self 的引用
    fn generate(self, format: ImageOutputFormat) -> Vec<u8>;
}
```

注意看新 create 方法的约束：任何 T，只要实现了相应的 TryFrom/TryInto，就可以用这个缺省的 create() 方法来构造 Engine。

有了这个接口后，上面使用 engine 的代码可以更加直观，省掉了第3行的try_into()处理：

```
// 使用 image engine 处理
let mut engine = Photon::create(data)
    .map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
engine.apply(&spec.specs);
let image = engine.generate(ImageOutputFormat::Jpeg(85));
```

桥接是架构中一个非常重要的思想，我们一定要掌握这个思想的精髓。

再举个例子。比如现在想要系统可以通过访问某个 REST API，得到用户自己发布的、按时间顺序倒排的朋友圈。怎么写这段代码呢？最简单粗暴的方式是：

```
let secret_api = api_with_user_token(&user, params);
let data: Vec<Status> = reqwest::get(secret_api)?.json()??;
```

更好的方式是使用 trait 桥接来屏蔽实现细节：

```
pub trait FriendCircle {
    fn get_published(&self, user: &User) -> Result<Vec<Status>, FriendCircleError>;
    ...
}
```

这样，我们的业务逻辑代码可以围绕着这个接口展开，而无需关心它具体的实现是来自 REST API，还是其它什么地方；也不用关心实现做没做 cache、有没有重传机制、具体都会返回什么样的错误（FriendCircleError 就已经提供了所有的出错可能）等等。

使用 trait 提供控制反转

继续看刚才的 Engine 代码，在 Engine 和 T 之间通过 TryInto trait 进行了解耦，使得调用者可以灵活处理他们的 T：

```
pub trait Engine {
    // 生成一个新的 engine
    fn create<T>(data: T) -> Result<Self>
    where
        Self: Sized,
        T: TryInto<Self>,
    {
        data.try_into()
            .map_err(|_| anyhow!("failed to create engine"))
    }
    ...
}
```

这里还体现了 trait 在设计中，另一个很重要的作用，控制反转。

通过使用 trait，我们可以在设计底层库的时候告诉上层：**我需要某个满足 trait X 的数据，因为我依赖这个数据实现的 trait X 方法来完成某些功能，但这个数据具体怎么实现，我不知道，也不关心。**

刚才为 Engine 新构建的 create 方法。T 是实现 Engine 所需要的依赖，我们不知道属于类型 T 的 data 是如何在上下文中产生的，也不关心 T 具体是什么，只要 T 实现了 TryInto 即可。这就是典型的控制反转。

使用 trait 做控制反转另一个例子是[第 6 讲](#)中的 Dialect trait (代码)：

```
pub trait Dialect: Debug + Any {
    /// Determine if a character starts a quoted identifier. The default
    /// implementation, accepting "double quoted" ids is both ANSI-compliant
    /// and appropriate for most dialects (with the notable exception of
    /// MySQL, MS SQL, and sqlite). You can accept one of characters listed
```

```

/// in `Word::matching_end_quote` here
fn is_delimited_identifier_start(&self, ch: char) -> bool {
    ch == '"'
}
/// Determine if a character is a valid start character for an unquoted identifier
fn is_identifier_start(&self, ch: char) -> bool;
/// Determine if a character is a valid unquoted identifier character
fn is_identifier_part(&self, ch: char) -> bool;
}

```

我们只需要为自己的 SQL 方言实现 Dialect trait:

```

// 创建自己的 sql 方言。TyrDialect 支持 identifier 可以是简单的 url
impl Dialect for TyrDialect {
    fn is_identifier_start(&self, ch: char) -> bool {
        ('a'..='z').contains(&ch) || ('A'..='Z').contains(&ch) || ch == '_'
    }

    // identifier 可以有 ':', '/', '?', '&', '='
    fn is_identifier_part(&self, ch: char) -> bool {
        ('a'..='z').contains(&ch)
        || ('A'..='Z').contains(&ch)
        || ('0'..='9').contains(&ch)
        || [':', '/', '?', '&', '=', '—', '_', '.'].contains(&ch)
    }
}

```

就可以让 sql parser 解析我们的 SQL 方言:

```
let ast = Parser::parse_sql(&TyrDialect::default(), sql.as_ref())?;
```

这就是 Dialect 这个看似简单的 trait 的强大用途。

对于我们这些使用者来说，通过 Dialect trait，可以很方便地注入自己的解析函数，来提供我们的 SQL 方言的额外信息；对于 sqlparser 这个库的作者来说，通过 Dialect trait，他不必关心未来会有多少方言、每个方言长什么样子，只需要方言的作者告诉他如何 tokenize 一个标识符即可。

控制反转是架构中经常使用到的功能，它能够让调用者和被调用者之间的关系在某个时刻调转过来，被调用者反过来调用调用者提供的能力，二者协同完成一些事情。

比如 MapReduce 的架构：用于 map 的方法和用于 reduce 的方法是啥，MapReduce 的架构设计者并不清楚，但调用者可以把这些方法提供给 MapReduce 架构，由 MapReduce 架构在合适的时候进行调用。

当然，控制反转并非只能由 trait 来完成，但使用 trait 做控制反转会非常灵活，调用者和被调用者只需要关心它们之间的接口，而非具体的数据结构。

用 trait 实现 SOLID 原则

其实刚才介绍的用 trait 做控制反转，核心体现的就是面向对象设计时SOLID原则中的，依赖反转原则DIP，这是一个很重要的构建灵活系统的思想。

在做面向对象设计时，我们经常会探讨 [SOLID 原则](#)：

- SRP：单一职责原则，是指每个模块应该只负责单一的功能，不应该让多个功能耦合在一起，而是应该将其组合在一起。
- OCP：开闭原则，是指软件系统应该对修改关闭，而对扩展开放。
- LSP：里氏替换原则，是指如果组件可替换，那么这些可替换的组件应该遵守相同的约束，或者说接口。
- ISP：接口隔离原则，是指使用者只需要知道他们感兴趣的方法，而不该被迫了解和使用对他们来说无用的方法或者功能。
- DIP：依赖反转原则，是指某些场合下底层代码应该依赖高层代码，而非高层代码去依赖底层代码。

虽然 Rust 不是一门面向对象语言，但这些思想都是通用的。

在过去的课程中，我一直强调 SRP 和 OCP。你看[第 6 讲](#)的 Fetch / Load trait，它们都只负责一个很简单的动作：

```
#[async_trait]
pub trait Fetch {
    type Error;
    async fn fetch(&self) -> Result<String, Self::Error>;
}

pub trait Load {
    type Error;
    fn load(self) -> Result<DataSet, Self::Error>;
}
```

以 Fetch 为例，我们先实现了 UrlFetcher，后来又根据需要，实现了 FileFetcher。

FileFetcher 的实现并不会对 UrlFetcher 的实现代码有任何影响，也就是说，在实现 FileFetcher 的时候，已有的所有实现了 Fetch 接口的代码都是稳定的，它们对修改是关闭的；同时，在实现 FileFetcher 的时候，我们扩展了系统的能力，使系统可以根据不同的前缀（`from file://` 或者 `from <http://>`）进行不同的处理，这是对扩展开放。

前面提到的 SpecTransform / Engine trait，包括 21 讲中 KV server 里涉及的 CommandService trait：

```
/// 对 Command 的处理的抽象
pub trait CommandService {
    /// 处理 Command, 返回 Response
    fn execute(self, store: &impl Storage) -> CommandResponse;
}
```

也是 SRP 和 OCP 原则的践行者。

LSP 里氏替换原则自不必说，我们本文中所有的内容都在践行通过使用接口，来使组件可替换。比如上文提到的 Engine trait，在 KV server 中我们使用的 Storage trait，都允许我们在不改变代码核心逻辑的前提下，替换其中的主要组件。

至于 ISP 接口隔离原则，我们目前撰写的 trait 都很简单，天然满足接口隔离原则。其实，大部分时候，当你的 trait 满足 SRP 单一职责原则时，它也满足接口隔离原则。

但在 Rust 中，有些 trait 的接口可能会比较庞杂，此时，如果我们想减轻调用者的负担，让它们能够在需要的时候才引入某些接口，可以使用 trait 的继承。比如 AsyncRead / AsyncWrite / Stream 和它们对应的 AsyncReadExt / AsyncWriteExt / StreamExt 等。这样，复杂的接口被不同的 trait 分担了并隔离开。

小结

接口设计是架构设计中最核心的环节。好的接口容易使用，很难误用，会让使用接口的人产生共鸣。当我们说一段代码读起来/写起来感觉很舒服，或者很不舒服、很冗长、很难看，这种感觉往往就来自于接口给人的感觉，我们可以妥善使用 trait 来降低甚至消除这种不舒服的感觉。

当我们的代码和其他人的代码共存时，接口在不同的组件之间就起到了桥接的作用。通过桥接，甚至可以把原本设计不好的代码，先用接口封装成我们希望的样子，然后实现这个简单的包装，之后再慢慢改动原先不好的设计。

这样，由于系统的其它部分使用新的接口处理，未来改动的影响被控制在很小的范围。在第 5 讲设计 thumbor 的时候我也提到，photon 库并不是一个设计良好的库，然而，通过 Engine trait 的桥接，未来即使我们 fork 一下 photon 库，对其大改，并不会影响 thumbor 的代码。

最后，在软件设计时，我们还要注意 SOLID 原则。基本上，好的 trait，设计一定是符合 SOLID 原则的，从 Rust 标准库的设计，以及之前讲到的 trait，结合今天的解读，想必你对此有了一定的认识。未来在使用 trait 构建你自己的接口时，你也可以将 SOLID 原则作为一个备忘清单，随时检查。

思考题

Rust 下有一个处理 Web 前端的库叫 [yew](#)。请 clone 到你本地，然后使用 ag 或者 rgrep (eat our own dogfood) 查找一下所有的 trait 定义，看看这些 trait 被设计的目的和意义，并且着重阅读一下它最核心的 Component trait，思考几个问题：

1. Component trait 可以做 trait object 吗？
2. 关联类型 Message 和 Properties 的作用是什么？
3. 作为使用者，该如何用 Component trait？它的 lifecycle 是什么样子的？
4. 如果你之前有前端开发的经验，比较一下 React / Vue / Elm component 和 yew component 的区别？

```
yew on master via v1.55.0
rgrep "pub trait" "**/*.rs"
examples/router/src/generator.rs
    155:1 pub trait Generated: Sized {
packages/yew/src/html/component/mod.rs
    42:1 pub trait Component: Sized + 'static {
packages/yew/src/html/component/properties.rs
    6:1 pub trait Properties: PartialEq {
examples/boids/src/math.rs
    128:1 pub trait WeightedMean<T = Self>: Sized {
    152:1 pub trait Mean<T = Self>: Sized {
packages/yew/src/functional/mod.rs
    69:1 pub trait FunctionProvider {
packages/yew/src/html/conversion.rs
    5:1 pub trait ImplicitClone: Clone {}
    18:1 pub trait IntoPropValue<T> {
packages/yew/src/html/listener/mod.rs
    27:1 pub trait TargetCast
    136:1 pub trait IntoEventCallback<EVENT> {
packages/yew/src/scheduler.rs
    11:1 pub trait Runnable {
packages/yew-router/src/routable.rs
    16:1 pub trait Routable: Sized + Clone {
packages/yew-agent/src/pool.rs
    60:1 pub trait Dispatched: Agent + Sized + 'static {
    78:1 pub trait Dispatchable {}
packages/yew/src/html/component/scope.rs
    508:1 pub trait SendAsMessage<COMP: Component> {
packages/yew-macro/src/stringify.rs
    16:1 pub trait Stringify {
packages/yew-agent/src/lib.rs
    22:1 pub trait Agent: Sized + 'static {
    82:1 pub trait Discoverer {
    92:1 pub trait Bridge<AGN: Agent> {
    98:1 pub trait Bridged: Agent + Sized + 'static {
packages/yew-agent/src/utils/store.rs
    20:1 pub trait Store: Sized + 'static {
    138:1 pub trait Bridgeable: Sized + 'static {
packages/yew-macro/src/html_tree/mod.rs
    178:1 pub trait ToNodeIterator {
```

```
packages/yew/src/virtual_dom/listeners.rs
42:1 pub trait Listener {
packages/yew-agent/src/worker/mod.rs
17:1 pub trait Threaded {
24:1 pub trait Packed {
```

感谢你的阅读，如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。今天你已经完成Rust学习的第25次打卡啦，我们下节课见！

数据结构

数据结构

01 | 数据结构：这些浓眉大眼的结构竟然都是智能指针？

到现在为止我们学了Rust的所有权与生命周期、内存管理以及类型系统，基础知识里还剩一块版图没有涉及：数据结构，数据结构里最容易让人困惑的就是智能指针，所以今天我们就来解决这个难点。

我们之前简单介绍过指针，这里还是先回顾一下：指针是一个持有内存地址的值，可以通过解引用访问它指向的内存地址，理论上可以解引用到任意数据类型；引用是一个特殊的指针，它的解引用访问是受限的，只能解引用到它引用数据的类型，不能用作它用。

那什么是智能指针呢？

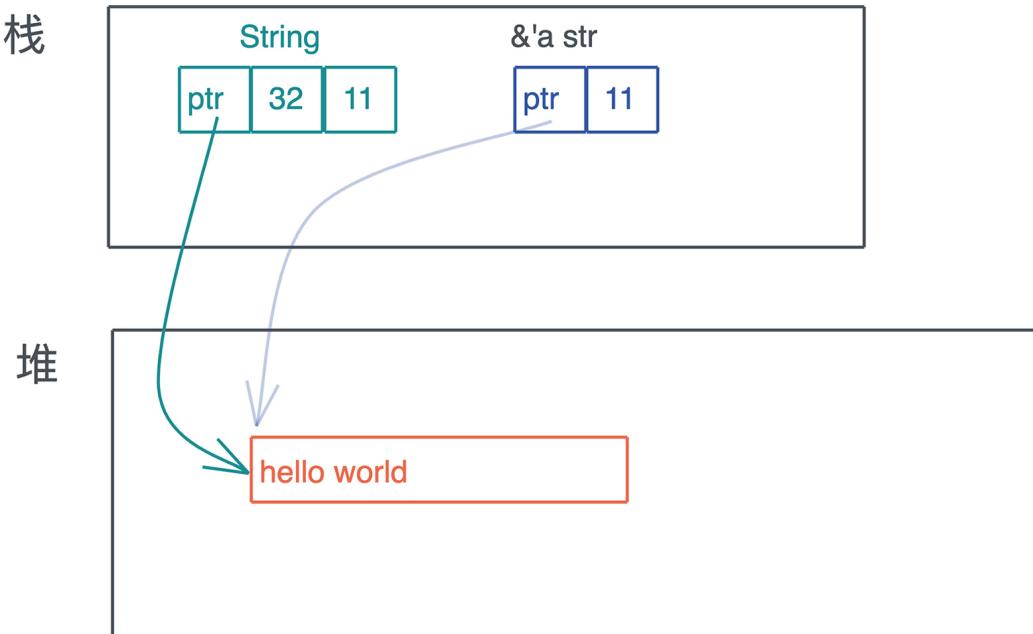
智能指针

在指针和引用的基础上，Rust 偷师 C++，提供了智能指针。智能指针是一个表现行为很像指针的数据结构，但除了指向数据的指针外，它还有元数据以提供额外的处理能力。

这个定义有点模糊，我们对比其他的数据结构来明确一下。

你有没有觉得很像之前讲的胖指针。智能指针一定是一个胖指针，但胖指针不一定是一个智能指针。比如 `&str` 就只是一个胖指针，它有指向堆内存字符串的指针，同时还有关于字符串长度的元数据。

我们看智能指针 String 和 &str 的区别：



从图上可以看到，String 除了多一个 capacity 字段，似乎也没有什么特殊。但 String 对堆上的值有所有权，而 &str 是没有所有权的，这是 Rust 中智能指针和普通胖指针的区别。

那么又有一个问题了，智能指针和结构体有什么区别呢？因为我们知道，String 是用结构体定义的：

```
pub struct String {  
    vec: Vec<u8>,  
}
```

和普通的结构体不同的是，String 实现了 Deref 和 DerefMut，这使得它在解引用的时候，会得到 &str，看下面的[标准库的实现](#)：

```
impl ops::Deref for String {  
    type Target = str;  
  
    fn deref(&self) -> &str {  
        unsafe { str::from_utf8_unchecked(&self.vec) }  
    }  
}  
  
impl ops::DerefMut for String {  
    fn deref_mut(&mut self) -> &mut str {  
        // Implementation  
    }  
}
```

```
        unsafe { str::from_utf8_unchecked_mut(&mut *self.vec) }
    }
}
```

另外，由于在堆上分配了数据，String 还需要为其分配的资源做相应的回收。而 String 内部使用了 Vec，所以它可以依赖 Vec 的能力来释放堆内存。下面是标准库中 Vec 的 [Drop trait 的实现](#)：

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for Vec<T, A> {
    fn drop(&mut self) {
        unsafe {
            // use drop for [T]
            // use a raw slice to refer to the elements of the vector as weakest necessary type;
            // could avoid questions of validity in certain cases
            ptr::drop_in_place(ptr::slice_from_raw_parts_mut(self.as_mut_ptr(), self.len))
        }
        // RawVec handles deallocation
    }
}
```

所以再清晰一下定义，在 Rust 中，凡是需要做资源回收的数据结构，且实现了 Deref/DerefMut/Drop，都是智能指针。

按照这个定义，除了 String，在之前的课程中我们遇到了很多智能指针，比如用于在堆上分配内存的 Box 和 Vec、用于引用计数的 Rc 和 Arc。很多其他数据结构，如 PathBuf、Cow<'a, B>、MutexGuard、RwLockReadGuard 和 RwLockWriteGuard 等也是智能指针。

今天我们就深入分析三个使用智能指针的数据结构：在堆上创建内存的 Box、提供写时克隆的 Cow<'a, B>，以及用于数据加锁的 MutexGuard。

而且最后我们会尝试实现自己的智能指针。希望学完后你不但能更好地理解智能指针，还能在需要的时候，构建自己的智能指针来解决问题。

Box

我们先看 Box，它是 Rust 中最基本的在堆上分配内存的方式，绝大多数其它包含堆内存分配的数据类型，内部都是通过 Box 完成的，比如 Vec。

为什么有Box的设计，我们得先回忆一下在 C 语言中，堆内存是怎么分配的。

C 需要使用 malloc/calloc/realloc/free 来处理内存的分配，很多时候，被分配出来的内存在函数调用中来来回回使用，导致谁应该负责释放这件事情很难确定，给开发者造成了极大的心智负担。

C++ 在此基础上改进了一下，提供了一个智能指针 [unique_ptr](#)，可以在指针退出作用域的时候释放堆内存，这样保证了堆内存的单一所有权。这个 [unique_ptr](#) 就是 Rust 的 Box 的前身。

你看 Box 的定义里，内部就是一个 [Unique](#) 用于致敬 C++，Unique 是一个私有的数据结构，我们不能直接使用，它包裹了一个 *mut T 指针，并唯一拥有这个指针。

```
pub struct Unique<T: ?Sized> {
    pointer: *const T,
    // NOTE: this marker has no consequences for variance, but is necessary
    // for dropck to understand that we logically own a `T`.
    //
    // For details, see:
    // https://github.com/rust-lang/rfcs/blob/master/text/0769-sound-generic-drop.md#phantom-data
    _marker: PhantomData<T>,
}
```

我们知道，在堆上分配内存，需要使用内存分配器（Allocator）。如果你上过操作系统课程，应该还记得一个简单的 [buddy system](#) 是如何分配和管理堆内存的。

设计内存分配器的目的除了保证正确性之外，就是为了有效地利用剩余内存，并控制内存在分配和释放过程中产生的碎片的数量。在多核环境下，它还要能够高效地处理并发请求。（如果你对通用内存分配器感兴趣，可以看参考资料）

堆上分配内存的 Box 其实有一个缺省的泛型参数 A，就需要满足 [Allocator trait](#)，并且默认是 Global：

```
pub struct Box<T: ?Sized, A: Allocator = Global>(Unique<T>, A)
```

Allocator trait 提供很多方法：

- `allocate` 是主要方法，用于分配内存，对应 C 的 `malloc/calloc`；
- `deallocate`，用于释放内存，对应 C 的 `free`；
- 还有 `grow / shrink`，用来扩大或缩小堆上已分配的内存，对应 C 的 `realloc`。

这里对 Allocator trait 我们就不详细介绍了，如果你想替换默认的内存分配器，可以使用 `#[global_allocator]` 标记宏，定义你自己的全局分配器。下面的代码展示了如何在 Rust 下使用 [jemalloc](#)：

```
use jemallocator::Jemalloc;
```

```
#[global_allocator]
static GLOBAL: Jemalloc = Jemalloc;

fn main() {}
```

这样设置之后，你使用 `Box::new()` 分配的内存就是 `jemalloc` 分配出来的了。另外，如果你想撰写自己的全局分配器，可以实现 [GlobalAlloc trait](#)，它和 `Allocator trait` 的区别，主要在于是否允许分配长度为零的内存。

使用场景

下面我们来实现一个自己的内存分配器。别担心，这里就是想 debug 一下，看看内存如何分配和释放，并不会实际实现某个分配算法。

首先看内存的分配。这里 `MyAllocator` 就用 `System allocator`，然后加 `eprintln!()`，和我们常用的 `println!()` 不同的是，`eprintln!()` 将数据打印到 `stderr` ([代码](#))：

```
use std::alloc::{GlobalAlloc, Layout, System};

struct MyAllocator;

unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let data = System.alloc(layout);
        eprintln!("ALLOC: {:p}, size {}", data, layout.size());
        data
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        System.dealloc(ptr, layout);
        eprintln!("FREE: {:p}, size {}", ptr, layout.size());
    }
}

#[global_allocator]
static GLOBAL: MyAllocator = MyAllocator;

#[allow(dead_code)]
struct Matrix {
    // 使用不规则的数字如 505 可以让 dbg! 的打印很容易分辨出来
    data: [u8; 505],
}

impl Default for Matrix {
    fn default() -> Self {
        Self { data: [0; 505] }
    }
}

fn main() {
```

```

// 在这句执行之前已经有好多内存分配
let data = Box::new(Matrix::default());

// 输出中有一个 1024 大小的内存分配，是 println! 导致的
println!(
    "!!! allocated memory: {:p}, len: {}", 
    &*data,
    std::mem::size_of::<Matrix>()
);

// data 在这里 drop，可以在打印中看到 FREE
// 之后还有很多其它内存被释放
}

```

注意这里不能使用 `println!()`。因为 `stdout` 会打印到一个由 `Mutex` 互斥锁保护的共享全局 `buffer` 中，这个过程中会涉及内存的分配，分配的内存又会触发 `println!()`，最终造成程序崩溃。而 `eprintln!` 直接打印到 `stderr`，不会 `buffer`。

运行这段代码，你可以看到类似如下输出，其中 505 大小的内存是我们 `Box::new()` 出来的：

```

cargo run --bin allocator --quiet
ALLOC: 0x7fbe0dc05c20, size 4
ALLOC: 0x7fbe0dc05c30, size 5
FREE: 0x7fbe0dc05c20, size 4
ALLOC: 0x7fbe0dc05c40, size 64
ALLOC: 0x7fbe0dc05c80, size 48
ALLOC: 0x7fbe0dc05cb0, size 80
ALLOC: 0x7fbe0dc05da0, size 24
ALLOC: 0x7fbe0dc05dc0, size 64
ALLOC: 0x7fbe0dc05e00, size 505
ALLOC: 0x7fbe0e008800, size 1024
!!! allocated memory: 0x7fbe0dc05e00, len: 505
FREE: 0x7fbe0dc05e00, size 505
FREE: 0x7fbe0e008800, size 1024
FREE: 0x7fbe0dc05c30, size 5
FREE: 0x7fbe0dc05c40, size 64
FREE: 0x7fbe0dc05c80, size 48
FREE: 0x7fbe0dc05cb0, size 80
FREE: 0x7fbe0dc05dc0, size 64
FREE: 0x7fbe0dc05da0, size 24

```

在使用 `Box` 分配堆内存的时候要注意，`Box::new()` 是一个函数，所以传入它的数据会出现在栈上，再移动到堆上。所以，如果我们的 `Matrix` 结构不是 505 个字节，是一个非常大的结构，就有可能出问题。

比如下面的代码想在堆上分配 16M 内存，如果你在 playground 里运行，直接栈溢出 stack overflow ([代码](#))：

```

fn main() {
    // 在堆上分配 16M 内存，但它会现在栈上出现，再移动到堆上
}

```

```
let boxed = Box::new([0u8; 1 << 24]);
println!("len: {}", boxed.len());
}
```

但如果你在本地使用“cargo run —release”编译成 release 代码运行，会正常执行！

这是因为“cargo run”或者在 playground 下运行，默认是 debug build，它不会做任何 inline 的优化，而 Box::new() 的实现就一行代码，并注明了要 inline，在 release 模式下，这个函数调用会被优化掉：

```
#[cfg(not(no_global_oom_handling))]
#[inline(always)]
#[doc(alias = "alloc")]
#[doc(alias = "malloc")]
#[stable(feature = "rust1", since = "1.0.0")]
pub fn new(x: T) -> Self {
    box x
}
```

如果不 inline，整个 16M 的大数组会通过栈内存传递给 Box::new，导致栈溢出。这里我们惊喜地发现了一个新的关键字 box。然而 box 是 Rust 内部的关键字，用户代码无法调用，它只出现在 Rust 代码中，用于分配堆内存，box 关键字在编译时，会使用内存分配器分配内存。

搞明白 Box 的内存分配，我们还很关心内存是如何释放的，来看它实现的 Drop trait：

```
#[stable(feature = "rust1", since = "1.0.0")]
unsafe impl<#[may_dangle] T: ?Sized, A: Allocator> Drop for Box<T, A> {
    fn drop(&mut self) {
        // FIXME: Do nothing, drop is currently performed by compiler.
    }
}
```

哈，目前 drop trait 什么都没有做，编译器会自动插入 deallocate 的代码。这是 Rust 语言的一种策略：**在具体实现还没有稳定下来之前，我先把接口稳定，实现随着之后的迭代慢慢稳定。**

这样可以极大地避免语言在发展的过程中，引入对开发者而言的破坏性更新（breaking change）。破坏性更新会使得开发者在升级语言的版本时，不得不大幅更改原有代码。

Python 是个前车之鉴，由于引入了大量的破坏性更新，Python 2 到 3 的升级花了十多年才慢慢完成。所以 Rust 在设计接口时非常谨慎，很多重要的接口都先以库的形式存在了很久，最终才成为标准库的一部分，比如 Future trait。一旦接口稳定后，内部的实现可以慢慢稳定。

Cow<'a, B>

了解了 Box 的工作原理后，再来看 Cow<'a, B>的原理和使用场景，（[第12讲](#)）讲泛型数据结构的时候，我们简单讲过参数B的三个约束。

Cow 是 Rust 下用于提供写时克隆（Clone-on-Write）的一个智能指针，它跟虚拟内存管理的写时复制（Copy-on-write）有异曲同工之妙：包裹一个只读借用，但如果调用者需要所有权或者需要修改内容，那么它会 clone 借用的数据。

我们看Cow的定义：

```
pub enum Cow<'a, B> where B: 'a + ToOwned + ?Sized {
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

它是一个 enum，可以包含一个对类型 B 的只读引用，或者包含对类型 B 的拥有所有权的数据。

这里又引入了两个 trait，首先是 ToOwned，在 ToOwned trait 定义的时候，又引入了 Borrow trait，它们都是 [std::borrow](#) 下的 trait：

```
pub trait ToOwned {
    type Owned: Borrow<Self>;
    #[must_use = "cloning is often expensive and is not expected to have side effects"]
    fn to_owned(&self) -> Self::Owned;

    fn clone_into(&self, target: &mut Self::Owned) { ... }
}

pub trait Borrow<Borrowed> where Borrowed: ?Sized {
    fn borrow(&self) -> &Borrowed;
}
```

如果你看不懂这段代码，不要着急，想要理解 Cow trait，ToOwned trait 是一道坎，因为 type Owned: Borrow 不好理解，耐下心来我们拆开一点点解读。

首先，type Owned: Borrow 是一个带有关联类型的 trait，如果你对这个知识点有些遗忘，可以再复习一下[第 13 讲](#)。这里 Owned 是关联类型，需要使用者定义，和我们之前介绍的关联类型不同的是，这里 Owned 不能是任意类型，它必须满足 Borrow trait。例如我们看 [str 对 ToOwned trait 的实现](#)：

```

impl ToOwned for str {
    type Owned = String;
    #[inline]
    fn to_owned(&self) -> String {
        unsafe { String::from_utf8_unchecked(self.as_bytes().to_owned()) }
    }

    fn clone_into(&self, target: &mut String) {
        let mut b = mem::take(target).into_bytes();
        self.as_bytes().clone_into(&mut b);
        *target = unsafe { String::from_utf8_unchecked(b) }
    }
}

```

可以看到关联类型 Owned 被定义为 String，而根据要求，String 必须定义 Borrow，那这里 Borrow 里的泛型变量 T 是谁呢？

ToOwned 要求是 Borrow，而此刻实现 ToOwned 的主体是 str，所以 Borrow 是 Borrow，也就是说 String 要实现 Borrow，我们看[文档](#)，它的确实现了这个 trait：

```

impl Borrow<str> for String {
    #[inline]
    fn borrow(&self) -> &str {
        &self[..]
    }
}

```

你是不是有点晕了，我用一张图梳理了这几个 trait 之间的关系：

Cow源码定义：Cow<



通过这张图，我们可以更好地搞清楚 Cow 和 ToOwned / Borrow 之间的关系。

这里，你可能会疑惑，为何 Borrow 要定义成一个泛型 trait 呢？搞这么复杂，难道一个类型还可以被借用成不同的引用么？

是的。我们看一个例子（[代码](#)）：

```
use std::borrow::Borrow;

fn main() {
    let s = "hello world!".to_owned();

    // 这里必须声明类型，因为 String 有多个 Borrow<T> 实现
    // 借用为 &String
    let r1: &String = s.borrow();
    // 借用为 &str
    let r2: &str = s.borrow();

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

在这里例子里，String 可以被借用为 &String，也可以被借用为 &str。

好，再来继续看 Cow。我们说它是智能指针，那它自然需要[实现 Deref trait](#)：

```
impl<B: ?Sized + ToOwned> Deref for Cow<'_, B> {
    type Target = B;

    fn deref(&self) -> &B {
        match *self {
            Borrowed(borrowed) => borrowed,
            Owned(ref owned) => owned.borrow(),
        }
    }
}
```

实现的原理很简单，根据 self 是 Borrowed 还是 Owned，我们分别取其内容，生成引用：

- 对于 Borrowed，直接就是引用；
- 对于 Owned，调用其 borrow() 方法，获得引用。

这就很厉害了。虽然 Cow 是一个 enum，但是通过 Deref 的实现，我们可以获得统一的体验，比如 Cow，使用的感觉和 &str / String 是基本一致的。注意，这种根据 enum 的不同状态来进行统一分发的方法是第三种分发手段，之前讲过可以使用泛型参数做静态分发和使用 trait object 做动态分发。

使用场景

那么 Cow 有什么用呢？显然，它可以在需要的时候才进行内存的分配和拷贝，在很多应用场合，它可以大大提升系统的效率。如果 Cow<'a, B> 中的 Owned 数据类型是一个需要在堆上分配内存的类型，如 String、Vec 等，还能减少堆内存分配的次数。

我们说过，相对于栈内存的分配释放来说，堆内存的分配和释放效率要低很多，其内部还涉及系统调用和锁，**减少不必要的堆内存分配是提升系统效率的关键手段**。而 Rust 的 Cow<'a, B>，在帮助你达成这个效果的同时，使用体验还非常简单舒服。

光这么说没有代码佐证，我们看一个使用 Cow 的实际例子。

在解析 URL 的时候，我们经常需要将 querystring 中的参数，提取成 KV pair 来进一步使用。绝大多数语言中，提取出来的 KV 都是新的字符串，在每秒钟处理几十 k 甚至上百 k 请求的系统中，你可以想象这会带来多少次堆内存的分配。

但在 Rust 中，我们可以用 Cow 类型轻松高效处理它，在读取 URL 的过程中：

- 每解析出一个 key 或者 value，我们可以用一个 &str 指向 URL 中相应的位置，然后用 Cow 封装它；
- 而当解析出来的内容不能直接使用，需要 decode 时，比如 “hello%20world”，我们可以生成一个解析后的 String，同样用 Cow 封装它。

看下面的例子（[代码](#)）：

```
use std::borrow::Cow;

use url::Url;
fn main() {
    let url = Url::parse("<https://tyr.com/rust?page=1024&sort=desc&extra=hello%20world>").unwrap();
```

```

let mut pairs = url.query_pairs();

assert_eq!(pairs.count(), 3);

let (mut k, v) = pairs.next().unwrap();
// 因为 k, v 都是 Cow<str> 他们用起来感觉和 &str 或者 String 一样
// 此刻, 他们都是 Borrowed
println!("key: {}, v: {}", k, v);
// 当修改发生时, k 变成 Owned
k.to_mut().push_str("_lala");

print_pairs((k, v));

print_pairs(pairs.next().unwrap());
// 在处理 extra=hello%20world 时, value 被处理成 "hello world"
// 所以这里 value 是 Owned
print_pairs(pairs.next().unwrap());
}

fn print_pairs(pair: (Cow<str>, Cow<str>)) {
    println!("key: {}, value: {}", show_cow(pair.0), show_cow(pair.1));
}

fn show_cow(cow: Cow<str>) -> String {
    match cow {
        Cow::Borrowed(v) => format!("Borrowed {}", v),
        Cow::Owned(v) => format!("Owned {}", v),
    }
}

```

是不是很简洁。

类似 URL parse 这样的处理方式, 在 Rust 标准库和第三方库中非常常见。比如 Rust 下著名的 [serde 库](#), 可以非常高效地对 Rust 数据结构, 进行序列化/反序列化操作, 它对 Cow 就有很好的支持。

我们可以通过如下代码将一个 JSON 数据反序列化成 User 类型, 同时让 User 中的 name 使用 Cow 来引用 JSON 文本中的内容 ([代码](#)) :

```

use serde::Deserialize;
use std::borrow::Cow;

#[derive(Debug, Deserialize)]
struct User<'input> {
    #[serde(borrow)]
    name: Cow<'input, str>,
    age: u8,
}

fn main() {
    let input = r#"{"name": "Tyr", "age": 18}"#;
    let user: User = serde_json::from_str(input).unwrap();
}

```

```

match user.name {
    Cow::Borrowed(x) => println!("borrowed {}", x),
    Cow::Owned(x) => println!("owned {}", x),
}
}

```

未来在你用 Rust 构造系统时，也可以充分考虑在数据类型中使用 Cow。

MutexGuard

如果说，上面介绍的 String、Box、Cow<'a, B> 等智能指针，都是通过 Deref 来提供良好的用户体验，那么 MutexGuard 是另外一类很有意思的智能指针：它不但通过 Deref 提供良好的用户体验，还通过 Drop trait 来确保，使用到的内存以外的资源在退出时进行释放。

MutexGuard 这个结构是在调用 [Mutex::lock](#) 时生成的：

```

pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
    unsafe {
        self.inner.raw_lock();
        MutexGuard::new(self)
    }
}

```

首先，它会取得锁资源，如果拿不到，会在这里等待；如果拿到了，会把 Mutex 结构的引用传递给 MutexGuard。

我们看 MutexGuard 的[定义](#)以及它的 Deref 和 Drop 的[实现](#)，很简单：

```

// 这里用 must_use，当你得到了却不使用 MutexGuard 时会报警
#[must_use = "if unused the Mutex will immediately unlock"]
pub struct MutexGuard<'a, T: ?Sized + 'a> {
    lock: &'a Mutex<T>,
    poison: poison::Guard,
}

impl<T: ?Sized> Deref for MutexGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}

```

```

impl<T: ?Sized> DerefMut for MutexGuard<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.lock.data.get() }
    }
}

impl<T: ?Sized> Drop for MutexGuard<'_, T> {
    #[inline]
    fn drop(&mut self) {
        unsafe {
            self.lock.poison.done(&self.poison);
            self.lock.inner.raw_unlock();
        }
    }
}

```

从代码中可以看到，当 MutexGuard 结束时，Mutex 会做 unlock，这样用户在使用 Mutex 时，可以不必关心何时释放这个互斥锁。因为无论你在调用栈上怎样传递 MutexGuard，哪怕在错误处理流程上提前退出，Rust 有所有权机制，可以确保只要 MutexGuard 离开作用域，锁就会被释放。

使用场景

我们来看一个使用 Mutex 和 MutexGuard 的例子（[代码](#)），代码很简单，我写了详尽的注释帮助你理解。

```

use lazy_static::lazy_static;
use std::borrow::Cow;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

// lazy_static 宏可以生成复杂的 static 对象
lazy_static! {
    // 一般情况下 Mutex 和 Arc 一起在多线程环境下提供对共享内存的使用
    // 如果你把 Mutex 声明成 static，其生命周期是静态的，不需要 Arc
    static ref METRICS: Mutex<HashMap<Cow<'static, str>, usize>> =
        Mutex::new(HashMap::new());
}

fn main() {
    // 用 Arc 来提供并发环境下的共享所有权（使用引用计数）
    let metrics: Arc<Mutex<HashMap<Cow<'static, str>, usize>>> =
        Arc::new(Mutex::new(HashMap::new()));
    for _ in 0..32 {
        let m = metrics.clone();
        thread::spawn(move || {
            let mut g = m.lock().unwrap();
            // 此时只有拿到 MutexGuard 的线程可以访问 HashMap
            let data = &mut *g;
            // Cow 实现了很多数据结构的 From trait,
        })
    }
}

```

```

        // 所以我们可以用 "hello".into() 生成 Cow
let entry = data.entry("hello".into()).or_insert(0);
*entry += 1;                                // MutexGuard 被 Drop, 锁被释放
    });
}

thread::sleep(Duration::from_millis(100));

println!("metrics: {:?}", metrics.lock().unwrap());
}

```

如果你有疑问，这样如何保证锁的线程安全呢？如果我在线程 1 拿到了锁，然后把 MutexGuard 移动给线程 2 使用，加锁和解锁在完全不同的线程下，会有很大的死锁风险。怎么办？

不要担心，MutexGuard 不允许 Send，只允许 Sync，也就是说，你可以把 MutexGuard 的引用传给另一个线程使用，但你无法把 MutexGuard 整个移动到另一个线程：

```

impl<T: ?Sized> !Send for MutexGuard<'_, T> {}
unsafe impl<T: ?Sized + Sync> Sync for MutexGuard<'_, T> {}

```

类似 MutexGuard 的智能指针有很多用途。比如要创建一个连接池，你可以在 Drop trait 中，回收 checkout 出来的连接，将其再放回连接池。如果你对此感兴趣，可以看看 [r2d2 的实现](#)，它是 Rust 下一个数据库连接池的实现。

实现自己的智能指针

到目前为止，三个经典的智能指针，在堆上创建内存的 Box、提供写时克隆的 Cow<'a, B>，以及用于数据加锁的 MutexGuard，它们的实现和使用方法就讲完了。

那么，如果我们想实现自己的智能指针，该怎么做？或者咱们换个问题：有什么数据结构适合实现成为智能指针？

因为很多时候，**我们需要实现一些自动优化的数据结构**，在某些情况下是一种优化的数据结构和相应的算法，在其他情况下使用通用的结构和通用的算法。

比如当一个 HashSet 的内容比较少的时候，可以用数组实现，但内容逐渐增多，再转换成用哈希表实现。如果我们想让使用者不用关心这些实现的细节，使用同样的接口就能享受到更好的性能，那么，就可以考虑用智能指针来统一它的行为。

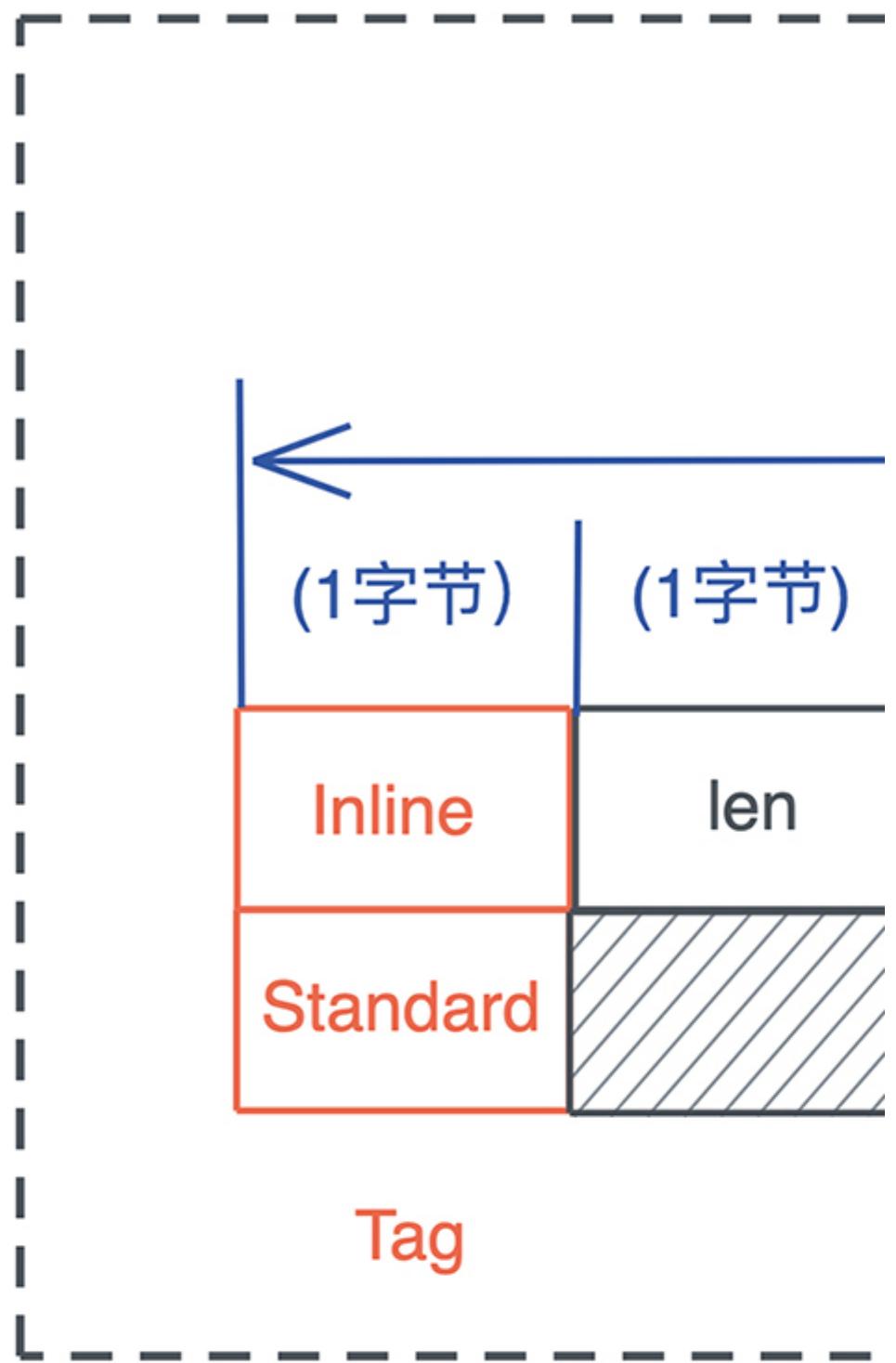
使用小练习

我们来看一个实际的例子。之前讲过，Rust 下 String 在栈上占了 24 个字节，然后在堆上存放字符串实际的内容，对于一些比较短的字符串，这很浪费内存。有没有办法在字符串长到一定程度后，才使用标准的字符串呢？

参考 Cow，我们可以用一个 enum 来处理：当字符串小于 N 字节时，我们直接用栈上的数组，否则，使用 String。但是这个 N 不宜太大，否则当使用 String 时，会比目前的版本浪费内存。

怎么设计呢？之前在内存管理的部分讲过，当使用 enum 时，额外的 tag + 为了对齐而使用的 padding 会占用一些内存。因为 String 结构是 8 字节对齐的，我们的 enum 最小 $8 + 24 = 32$ 个字节。

所以，可以设计一个数据结构，内部用一个字节表示字符串的长度，用 30 个字节表示字符串内容，再加上 1 个字节的 tag，正好也是 32 字节，可以和 String 放在一个 enum 里使用。我们暂且称这个 enum 叫 MyString，它的结构如下图所示：



为了让 MyString 表现行为和 &str 一致，我们可以通过实现 Deref trait 让 MyString 可以被解引用成 &str。除此之外，还可以实现 Debug/Display 和 From trait，让 MyString 使用起来更方便。

整个实现的代码如下 ([代码](#))，代码本身不难理解，你可以试着自己实现一下，或者一行行抄下来运行，感受一下。

```

use std::{fmt, ops::Deref, str};

const MINI_STRING_MAX_LEN: usize = 30;

// MyString 里, String 有 3 个 word, 供 24 字节, 所以它以 8 字节对齐
// 所以 enum 的 tag + padding 最少 8 字节, 整个结构占 32 字节。
// MiniString 可以最多有 30 字节 (再加上 1 字节长度和 1字节 tag) , 就是 32 字节.
struct MiniString {
    len: u8,
    data: [u8; MINI_STRING_MAX_LEN],
}

impl MiniString {
    // 这里 new 接口不暴露出去, 保证传入的 v 的字节长度小于等于 30
    fn new(v: impl AsRef<str>) -> Self {
        let bytes = v.as_ref().as_bytes();
        // 我们在拷贝内容时一定要使用字符串的字节长度
        let len = bytes.len();
        let mut data = [0u8; MINI_STRING_MAX_LEN];
        data[..len].copy_from_slice(bytes);
        Self {
            len: len as u8,
            data,
        }
    }
}

impl Deref for MiniString {
    type Target = str;

    fn deref(&self) -> &Self::Target {
        // 由于生成 MiniString 的接口是隐藏的, 它只能来自字符串, 所以下面这行是安全的
        str::from_utf8(&self.data[..self.len as usize]).unwrap()
        // 也可以直接用 unsafe 版本
        // unsafe { str::from_utf8_unchecked(&self.data[..self.len as usize]) }
    }
}

impl fmt::Debug for MiniString {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        // 这里由于实现了 Deref trait, 可以直接得到一个 &str 输出
        write!(f, "{}", self.deref())
    }
}

#[derive(Debug)]
enum MyString {
    Inline(MiniString),
    Standard(String),
}

// 实现 Deref 接口对两种不同的场景统一得到 &str
impl Deref for MyString {
    type Target = str;

    fn deref(&self) -> &Self::Target {
        match *self {

```

```

        MyString::Inline(ref v) => v.deref(),
        MyString::Standard(ref v) => v.deref(),
    }
}
}

impl From<&str> for MyString {
    fn from(s: &str) -> Self {
        match s.len() > MINI_STRING_MAX_LEN {
            true => Self::Standard(s.to_owned()),
            _ => Self::Inline(MiniString::new(s)),
        }
    }
}

impl fmt::Display for MyString {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.deref())
    }
}

fn main() {
    let len1 = std::mem::size_of::<MyString>();
    let len2 = std::mem::size_of::<MiniString>();
    println!("Len: MyString {}, MiniString {}", len1, len2);

    let s1: MyString = "hello world".into();
    let s2: MyString = "这是一个超过了三十个字节的很长很长的字符串".into();

    // debug 输出
    println!("s1: {:?}, s2: {:?}", s1, s2);
    // display 输出
    println!(
        "s1: {}({} bytes, {} chars), s2: {}({} bytes, {} chars)",
        s1,
        s1.len(),
        s1.chars().count(),
        s2,
        s2.len(),
        s2.chars().count()
    );
}

// MyString 可以使用一切 &str 接口, 感谢 Rust 的自动 Deref
assert!(s1.ends_with("world"));
assert!(s2.starts_with("这"));
}

```

这个简单实现的 MyString，不管它内部的数据是纯栈上的 MiniString 版本，还是包含堆上内存的 String 版本，使用的体验和 &str 都一致，仅仅牺牲了一点点效率和内存，就可以让小容量的字符串，可以高效地存储在栈上并且自如地使用。

事实上，Rust 有个叫 smartstring 的第三方库就实现了这个功能。我们的版本在内存上不算经济，对于 String 来说，额外多用了 8 个字节，smartstring 通过优化，只用了和 String 结构一样大小的 24 个字节，就达到了我们想要的结果。你如果感兴趣的话，欢迎去看看它的[源代码](#)。

小结

今天我们介绍了三个重要的智能指针，它们有各自独特的实现方式和使用场景。

Box 可以在堆上创建内存，是很多其他数据结构的基础。

Cow 实现了 Clone-on-write 的数据结构，让你可以在需要的时候再获得数据的所有权。Cow 结构是一种使用 enum 根据当前的状态进行分发的经典方案。甚至，你可以用类似的方案取代 trait object 做动态分发，[其效率是动态分发的数十倍](#)。

如果你想合理地处理资源相关的管理，MutexGuard 是一个很好的参考，它把从 Mutex 中获得的锁包装起来，实现只要 MutexGuard 退出作用域，锁就一定会释放。如果你要做资源池，可以使用类似 MutexGuard 的方式。

思考题

1. 目前 MyString 只能从 &str 生成。如果要支持从 String 中生成一个 MyString，该怎么做？
2. 目前 MyString 只能读取，不能修改，能不能给它加上类似 String 的 `push_str` 接口？
3. 你知道 Cow<[u8]> 和 Cow 的大小么？试着打印一下看看。想想，为什么它的大小是这样呢？

欢迎在留言区分享你的思考。今天你已经完成Rust学习第15次打卡了，继续加油，我们下节课见～

参考资料

常见的通用内存分配器有 glibc 的 [pthread malloc](#)、Google 开发的 [tcmalloc](#)、FreeBSD 上默认使用的 [jemalloc](#) 等。除了通用内存分配器，对于特定类型内存的分配，我们还可以用 [slab](#)，slab 相当于一个预分配好的对象池，可以扩展和收缩。

02 | 数据结构：Vec_<T>、&[T]、Box_<T>，你真的了解集合容器么？

现在我们接触到了越来越多的数据结构，我把 Rust 中主要的数据结构从原生类型、容器类型和系统相关类型几个维度整理一下，你可以数数自己掌握了哪些。



可以看到，容器占据了数据结构的半壁江山。

提到容器，很可能你首先会想到的就是数组、列表这些可以遍历的容器，但其实只要把某种特定的数据封装在某个数据结构中，这个数据结构就是一个容器。比如 Option，它是一个包裹了 T 存在或不存在的容器，而Cow 是一个封装了内部数据 B 或被借用或拥有所有权的容器。

对于容器的两小类，到目前为止，像 Cow 这样，为特定目的而产生的容器我们已经介绍了不少，包括 Box、Rc、Arc、RefCell、还没讲到的 Option 和 Result 等。

今天我们来详细讲讲另一类，集合容器。

集合容器

集合容器，顾名思义，就是把一系列拥有相同类型的数据放在一起，统一处理，比如：

- 我们熟悉的字符串 String、数组 [T; n]、列表 Vec 和哈希表 HashMap<K, V> 等；
- 虽然到处在使用，但还并不熟悉的切片 slice；
- 在其他语言中使用过，但在 Rust 中还没有用过的循环缓冲区 VecDeque、双向列表 LinkedList 等。

这些集合容器有很多共性，比如可以被遍历、可以进行 map-reduce 操作、可以从一种类型转换成另一种类型等等。

我们会选取两类典型的集合容器：切片和哈希表，深入解读，理解了这两类容器，其它的集合容器设计思路都差不多，并不难学习。今天先介绍切片以及和切片相关的容器，下一讲我们学习哈希表。

切片究竟是什么？

在 Rust 里，切片是描述一组属于同一类型、长度不确定的、在内存中连续存放的数据结构，用 [T] 来表述。因为长度不确定，所以切片是个 DST (Dynamically Sized Type)。

切片一般只出现在数据结构的定义中，不能直接访问，在使用中主要用以下形式：

- &[T]：表示一个只读的切片引用。
- &mut [T]：表示一个可写的切片引用。
- Box<[T]>：一个在堆上分配的切片。

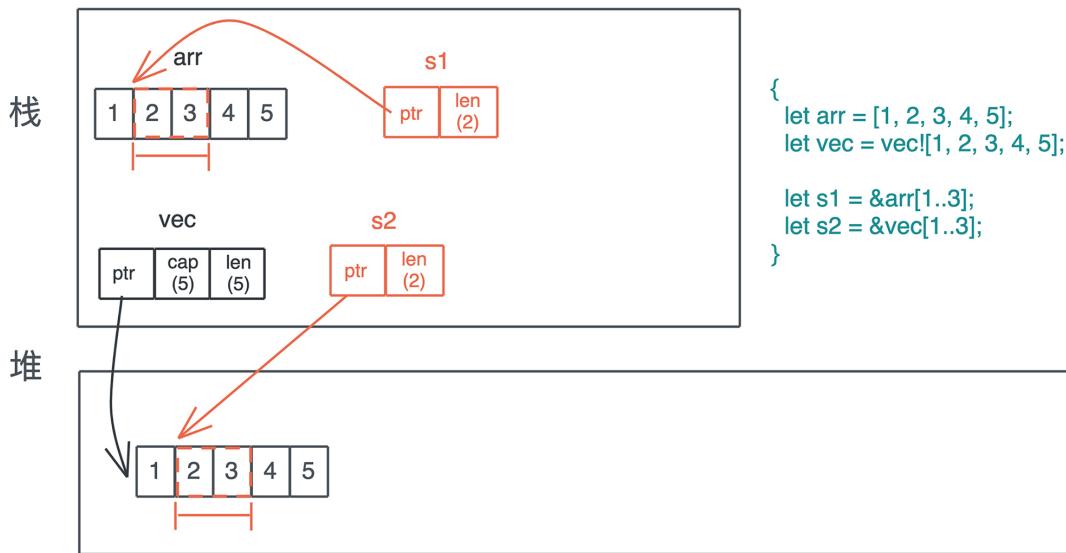
怎么理解切片呢？我打个比方，切片之于具体的数据结构，就像数据库中的视图之于表。你可以把它看成一种工具，让我们可以统一访问行为相同、结构类似但有些许差异的类型。

来看下面的[代码](#)，辅助理解：

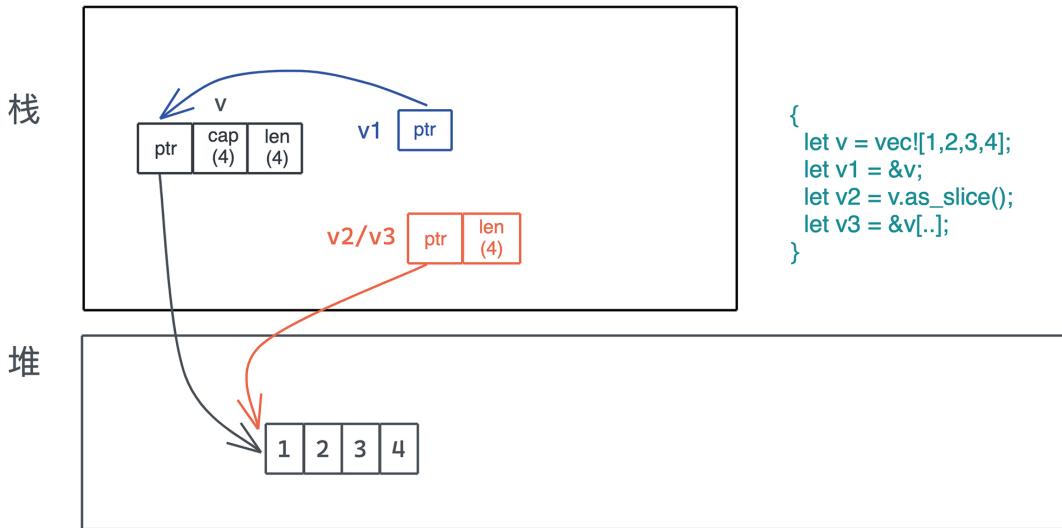
```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let vec = vec![1, 2, 3, 4, 5];
    let s1 = &arr[..2];
    let s2 = &vec[..2];
    println!("s1: {:?}", s1, s2);
    // &[T] 和 &[T] 是否相等取决于长度和内容是否相等
    assert_eq!(s1, s2);
    // &[T] 可以和 Vec<T>/[T;n] 比较，也会看长度和内容
    assert_eq!(&arr[..], vec);
    assert_eq!(&vec[..], arr);
}
```

对于 array 和 vector，虽然是不同的数据结构，一个放在栈上，一个放在堆上，但它们的切片是类似的；而且对于相同内容数据的相同切片，比如 `&arr[1..3]` 和 `&vec[1..3]`，这两者是等价的。除此之外，切片和对应的数据结构也可以直接比较，这是因为它们之间实现了 `PartialEq` trait（[源码参考资料](#)）。

下图比较清晰地呈现了切片和数据之间的关系：



另外在 Rust 下，切片日常中都是使用引用 `&[T]`，所以很多同学容易搞不清楚 `&[T]` 和 `&Vec` 的区别。我画了张图，帮助你更好地理解它们的关系：



在使用的时候，支持切片的具体数据类型，你可以根据需要，解引用转换成切片类型。比如 `Vec` 和 `[T; n]` 会转化成为 `&[T]`，这是因为 `Vec` 实现了 `Deref` trait，而 `array` 内建了到 `&[T]` 的解引用。我们可以写一段代码验证这一行为（[代码](#)）：

```
use std::fmt;  
fn main() {  
    let v = vec![1, 2, 3, 4];  
  
    // Vec 实现了 Deref, &Vec<T> 会被自动解引用为 &[T], 符合接口定义  
    print_slice(&v);  
    // 直接是 &[T], 符合接口定义  
    print_slice(&v[..]);  
  
    // &Vec<T> 支持 AsRef<[T]>  
    print_slice1(&v);  
    // &[T] 支持 AsRef<[T]>  
    print_slice1(&v[..]);  
    // Vec<T> 也支持 AsRef<[T]>  
    print_slice1(v);  
  
    let arr = [1, 2, 3, 4];  
    // 数组虽没有实现 Deref, 但它的解引用就是 &[T]  
    print_slice(&arr);  
    print_slice(&arr[..]);  
    print_slice1(&arr);  
    print_slice1(&arr[..]);  
    print_slice1(arr);
```

```

}

// 注意下面的泛型函数的使用
fn print_slice<T: fmt::Debug>(s: &[T]) {
    println!("{}:?", s);
}

fn print_slice1<T, U>(s: T)
where
    T: AsRef<[U]>,
    U: fmt::Debug,
{
    println!("{}:?", s.as_ref());
}

```

这也就意味着，通过解引用，这几个和切片有关的数据结构都会获得切片的所有能力，包括：binary_search、chunks、concat、contains、start_with、end_with、group_by、iter、join、sort、split、swap 等一系列丰富的功能，感兴趣的同学可以看[切片的文档](#)。

切片和迭代器 Iterator

迭代器可以说是切片的孪生兄弟。切片是集合数据的视图，而迭代器定义了对集合数据的各种各样的访问操作。

通过切片的 [iter\(\) 方法](#)，我们可以生成一个迭代器，对切片进行迭代。

在[第12讲](#)Rust类型推导已经见过了 iterator trait（用 `collect` 方法把过滤出来的数据形成新列表）。iterator trait 有大量的方法，但绝大多数情况下，我们只需要定义它的关联类型 Item 和 next() 方法。

- Item 定义了每次我们从迭代器中取出的数据类型；
- next() 是从迭代器里取下一个值的方法。当一个迭代器的 next() 方法返回 None 时，表明迭代器中没有数据了。

```

#[must_use = "iterators are lazy and do nothing unless consumed"]
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // 大量缺省的方法，包括 size_hint, count, chain, zip, map,
    // filter, for_each, skip, take_while, flat_map, flatten
    // collect, partition 等
    ...
}

```

看一个例子，对 Vec 使用 iter() 方法，并进行各种 map / filter / take 操作。在函数式编程语言中，这样的写法很常见，代码的可读性很强。Rust 也支持这种写法（[代码](#)）：

```
fn main() {
    // 这里 Vec<T> 在调用 iter() 时被解引用成 &[T]，所以可以访问 iter()
    let result = vec![1, 2, 3, 4]
        .iter()
        .map(|v| v * v)
        .filter(|v| *v < 16)
        .take(1)
        .collect::<Vec<_>>();

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

需要注意的是 Rust 下的迭代器是个懒接口（lazy interface），也就是说这段代码直到运行到 collect 时才真正开始执行，之前的部分不过是在不断地生成新的结构，来累积处理逻辑而已。你可能好奇，这是怎么做到的呢？

在 VS Code 里，如果你使用了 rust-analyzer 插件，就可以发现这一奥秘：

```
fn main() {  
    let result: Vec<{unknown}> =  
        [1, 2, 3, 4, 5]  
            .iter()  
            .map(|v: &i32| v * v)  
            .filter(|v: &{unknown}| v % 2 == 0)  
            .take(1)  
            .collect::<Vec<_>>();  
  
    println!("{:?}", result);  
}
```

原来，Iterator 大部分方法都返回一个实现了 Iterator 的数据结构，所以可以这样一路链式下去，在 Rust 标准库中，这些数据结构被称为 [Iterator Adapter](#)。比如上面的 map 方法，它返回 Map 结构，而 Map 结构实现了 Iterator ([源码](#))。

整个过程是这样的（链接均为源码资料）：

- 在 collect() 执行的时候，它实际[试图使用 FromIterator 从迭代器中构建一个集合类型](#)，这会不断调用 next() 获取下一个数据；
- 此时的 Iterator 是 Take，Take 调自己的 next()，也就是它会[调用 Filter 的 next\(\)](#)；
- Filter 的 next() 实际上[调用自己内部的 iter 的 find\(\)](#)，此时内部的 iter 是 Map，find() 会使用 try_fold()，它会[继续调用 next\(\)](#)，也就是 Map 的 next()；
- Map 的 next() 会[调用其内部的 iter 取 next\(\) 然后执行 map 函数](#)。而此时内部的 iter 来自 Vec。

所以，只有在 `collect()` 时，才触发代码一层层调用下去，并且调用会根据需要随时结束。这段代码中我们使用了 `take(1)`，整个调用链循环一次，就能满足 `take(1)` 以及所有中间过程的要求，所以它只会循环一次。

你可能会有疑惑：这种函数式编程的写法，代码是漂亮了，然而这么多无谓的函数调用，性能肯定很差吧？毕竟，函数式编程语言的一大恶名就是性能差。

这个你完全不用担心，Rust 大量使用了 `inline` 等优化技巧，这样非常清晰友好的表达方式，性能和 C 语言的 `for` 循环差别不大。如果你对性能对比感兴趣，可以去最后的参考资料区看看。

介绍完是什么，按惯例我们就要上代码实际使用一下了。不过迭代器是非常重要的一个功能，基本上每种语言都有对迭代器的完整支持，所以只要你之前用过，对此应该并不陌生，大部分的方法，你一看就能明白是在做什么。所以这里就不再额外展示，等你遇到具体需求时，可以翻 [Iterator 的文档查阅](#)。

如果标准库中的功能还不能满足你的需求，你可以看看 [itertools](#)，它是和 Python 下 `itertools` 同名且功能类似的工具，提供了大量额外的 adapter。可以看一个简单的例子（[代码](#)）：

```
use itertools::itertools;

fn main() {
    let err_str = "bad happened";
    let input = vec![Ok(21), Err(err_str), Ok(7)];
    let it = input
        .into_iter()
        .filter_map_ok(|i| if i > 10 { Some(i * 2) } else { None });
    // 结果应该是: vec![Ok(42), Err(err_str)]
    println!("{:?}", it.collect::<Vec<_>>());
}
```

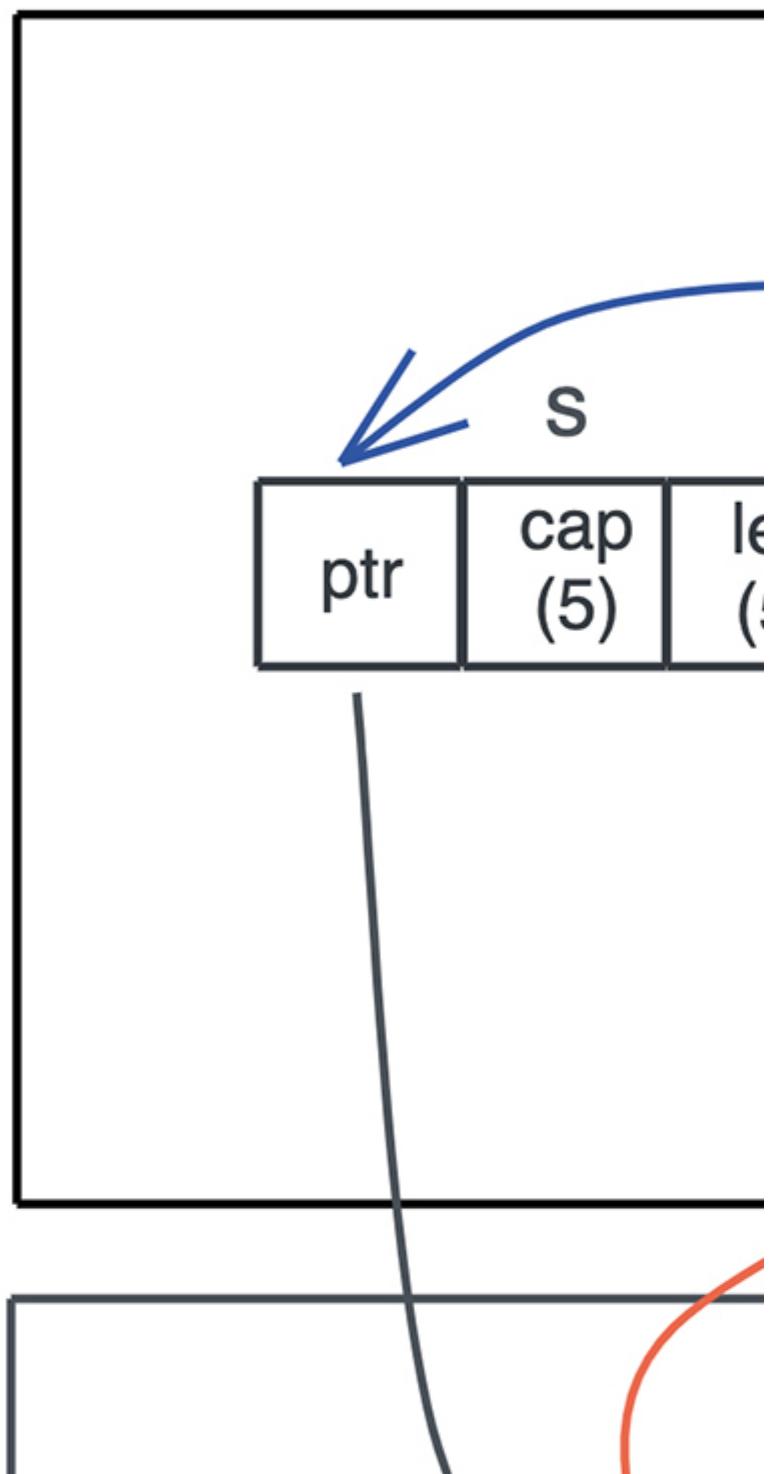
在实际开发中，我们可能从一组 `Future` 中汇聚出一组结果，里面有成功执行的结果，也有失败的错误信息。如果想对成功的结果进一步做 `filter/map`，那么标准库就无法帮忙了，就需要用 `itertools` 里的 `filter_map_ok()`。

特殊的切片：`&str`

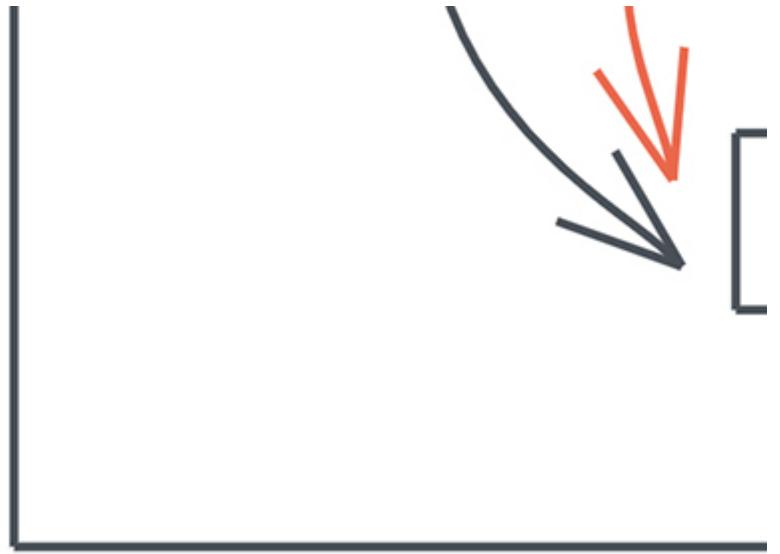
好，学完了普通的切片 `&[T]`，我们来看一种特殊的切片：`&str`。之前讲过，`String` 是一个特殊的 `Vec`，所以在 `String` 上做切片，也是一个特殊的结构 `&str`。

对于 String、&String、&str，很多人也经常分不清它们的区别，我们在之前的一篇加餐中简单聊了这个问题，在上一讲智能指针中，也对比过String和&str。对于&String 和 &str，如果你理解了上文中 &Vec 和 &[T] 的区别，那么它们也是一样的：

栈



堆



String 在解引用时，会转换成 &str。可以用下面的代码验证（[代码](#)）：

```
use std::fmt;
fn main() {
    let s = String::from("hello");
    // &String 会被解引用成 &str
    print_slice(&s);
    // &s[..] 和 s.as_str() 一样，都会得到 &str
    print_slice(&s[..]);
    print_slice1(&s);
    print_slice1(&s[..]);
    print_slice1(s.clone());

    // String 支持 AsRef<str>
    print_slice2(&s);
    print_slice2(&s[..]);
    print_slice2(s);
}
```

fn print_slice(s: &str) {

```

    println!("{:?}", s);
}

fn print_slice1<T: AsRef<str>>(s: T) {
    println!("{:?}", s.as_ref());
}

fn print_slice2<T, U>(s: T)
where
    T: AsRef<[U]>,
    U: fmt::Debug,
{
    println!("{:?}", s.as_ref());
}

```

有同学会有疑问：那么字符的列表和字符串有什么关系和区别？我们直接写一段代码来看看：

```

use std::iter::FromIterator;

fn main() {
    let arr = ['h', 'e', 'l', 'l', 'o'];
    let vec = vec!['h', 'e', 'l', 'l', 'o'];
    let s = String::from("hello");
    let s1 = &arr[1..3];
    let s2 = &vec[1..3];
    // &str 本身就是一个特殊的 slice
    let s3 = &s[1..3];
    println!("s1: {:?}, s2: {:?}, s3: {:?}", s1, s2, s3);

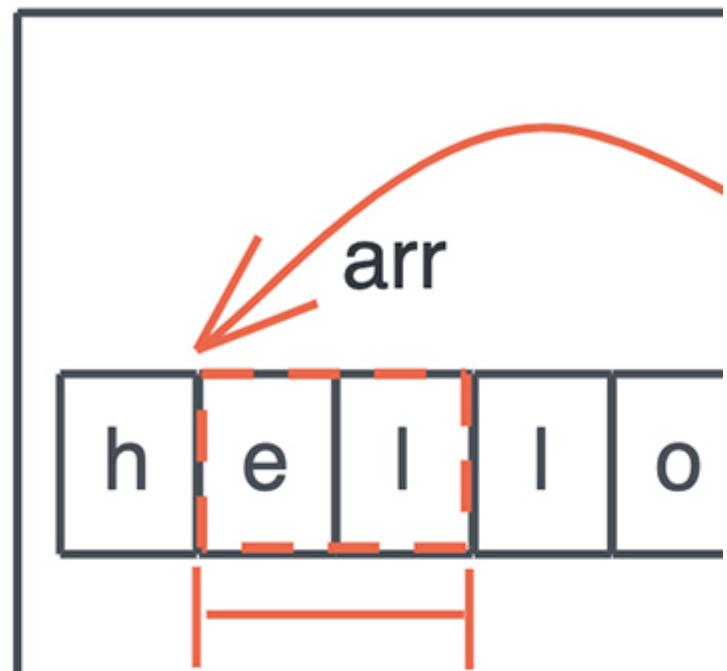
    // &[char] 和 &[char] 是否相等取决于长度和内容是否相等
    assert_eq!(s1, s2);
    // &[char] 和 &str 不能直接对比，我们把 s3 变成 Vec<char>
    assert_eq!(s2, s3.chars().collect::<Vec<_>>());
    // &[char] 可以通过迭代器转换成 String, String 和 &str 可以直接对比
    assert_eq!(String::from_iter(s2), s3);
}

```

可以看到，字符列表可以通过迭代器转换成 String, String 也可以通过 chars() 函数转换成字符列表，如果不转换，二者不能比较。

下图我把数组、列表、字符串以及它们的切片放在一起比较，可以帮你更好地理解它们的区别：

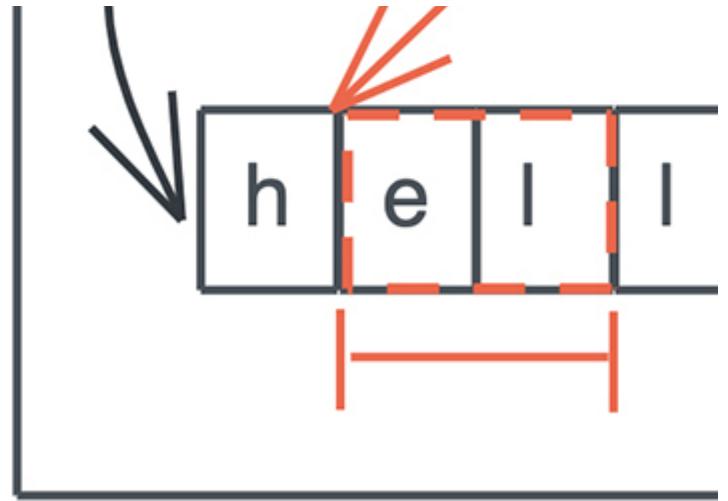
栈



vec



堆



切片的引用和堆上的切片，它们是一回事么？

开头我们讲过，切片主要有三种使用方式：切片的只读引用 `&[T]`、切片的可变引用 `&mut [T]` 以及 `Box<[T]>`。刚才已经详细学习了只读切片 `&[T]`，也和其他各种数据结构进行了对比帮助理解，可变切片 `&mut [T]` 和它类似，不必介绍。

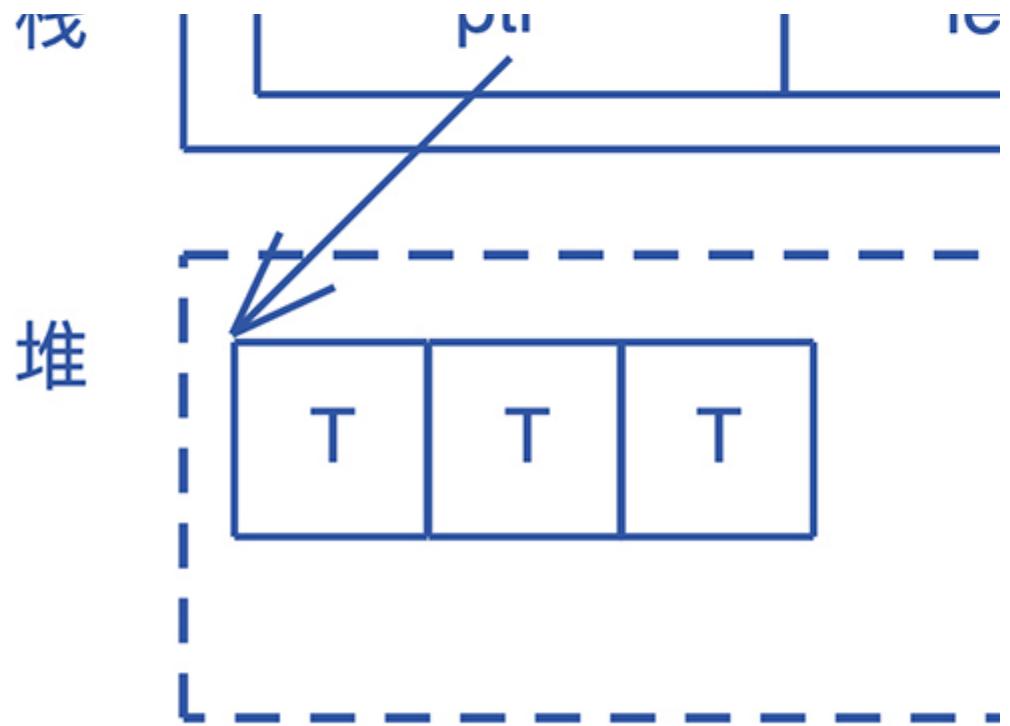
现在我们来看看 `Box<[T]>`。

`Box<[T]>` 是一个比较有意思的存在，它和 `Vec` 有一点点差别：`Vec` 有额外的 `capacity`，可以增长；而 `Box<[T]>` 一旦生成就固定下来，没有 `capacity`，也无法增长。

`Box<[T]>`和切片的引用`&[T]`也很类似：它们都是在栈上有一个包含长度的胖指针，指向存储数据的内存位置。区别是：`Box<[T]>`只会指向堆，`&[T]`指向的位置可以是栈也可以是堆；此外，`Box<[T]>`对数据具有所有权，而`&[T]`只是一个借用。

Box<[T]>





那么如何产生 `Box<[T]>` 呢？目前可用的接口就只有一个：从已有的 `Vec` 中转换。我们看代码：

```
use std::ops::Deref;

fn main() {
    let mut v1 = vec![1, 2, 3, 4];
    v1.push(5);
    println!("cap should be 8: {}", v1.capacity());

    // 从 Vec<T> 转换成 Box<[T]>, 此时会丢弃多余的 capacity
    let b1 = v1.into_boxed_slice();
    let mut b2 = b1.clone();

    let v2 = b1.into_vec();
    println!("cap should be exactly 5: {}", v2.capacity());

    assert!(b2.deref() == v2);
}
```

```

// Box<[T]> 可以更改其内部数据，但无法 push
b2[0] = 2;
// b2.push(6);
println!("b2: {:?}", b2);

// 注意 Box<[T]> 和 Box<[T; n]> 并不相同
let b3 = Box::new([2, 2, 3, 4, 5]);
println!("b3: {:?}", b3);

// b2 和 b3 相等，但 b3.deref() 和 v2 无法比较
assert!(b2 == b3);
// assert!(b3.deref() == v2);
}

```

运行代码可以看到，Vec 可以通过 `into_boxed_slice()` 转换成 Box<[T]>，Box<[T]> 也可以通过 `into_vec()` 转换回 Vec。

这两个转换都是很轻量的转换，只是变换一下结构，不涉及数据的拷贝。区别是，当 Vec 转换成 Box<[T]> 时，没有使用到的容量就会被丢弃，所以整体占用的内存可能会降低。而且 Box<[T]> 有一个很好的特性是，不像 Box<[T;n]> 那样在编译时就要确定大小，它可以在运行期生成，以后大小不会再改变。

所以，当我们需要在堆上创建固定大小的集合数据，且不希望自动增长，那么，可以先创建 Vec，再转换成 Box<[T]>。tokio 在提供 broadcast channel 时，就使用了 Box<[T]> 这个特性，你感兴趣的话，可以自己看看[源码](#)。

小结

我们讨论了切片以及和切片相关的主要数据类型。切片是一个很重要的数据类型，你可以着重理解它存在的意义，以及使用方式。

今天学完相信你也看到了，围绕着切片有很多数据结构，而切片将它们抽象成相同的访问方式，实现了在不同数据结构之上的同一抽象，这种方法很值得我们学习。此外，当我们构建自己的数据结构时，如果它内部也有连续排列的等长的数据结构，可以考虑 AsRef 或者 Deref 到切片。

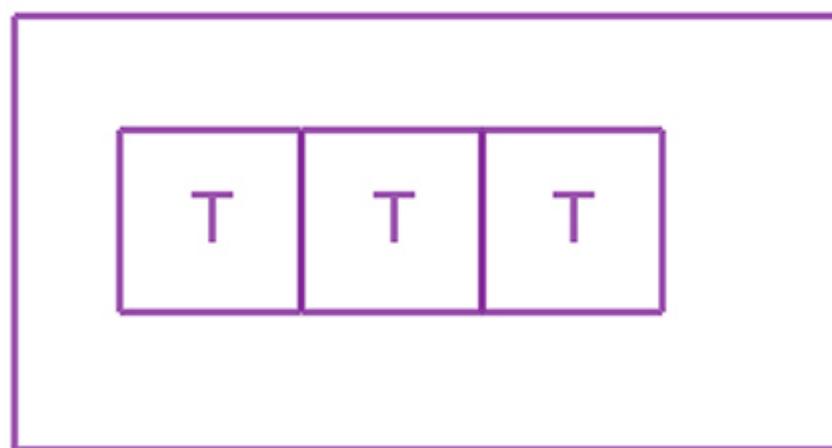
下图描述了切片和数组 [T;n]、列表 Vec、切片引用 &[T] /&mut [T]，以及在堆上分配的切片 Box<[T]> 之间的关系。建议你花些时间理解这张图，也可以用相同的方式去总结学到的其他有关联的数据结构。

切片：[T]

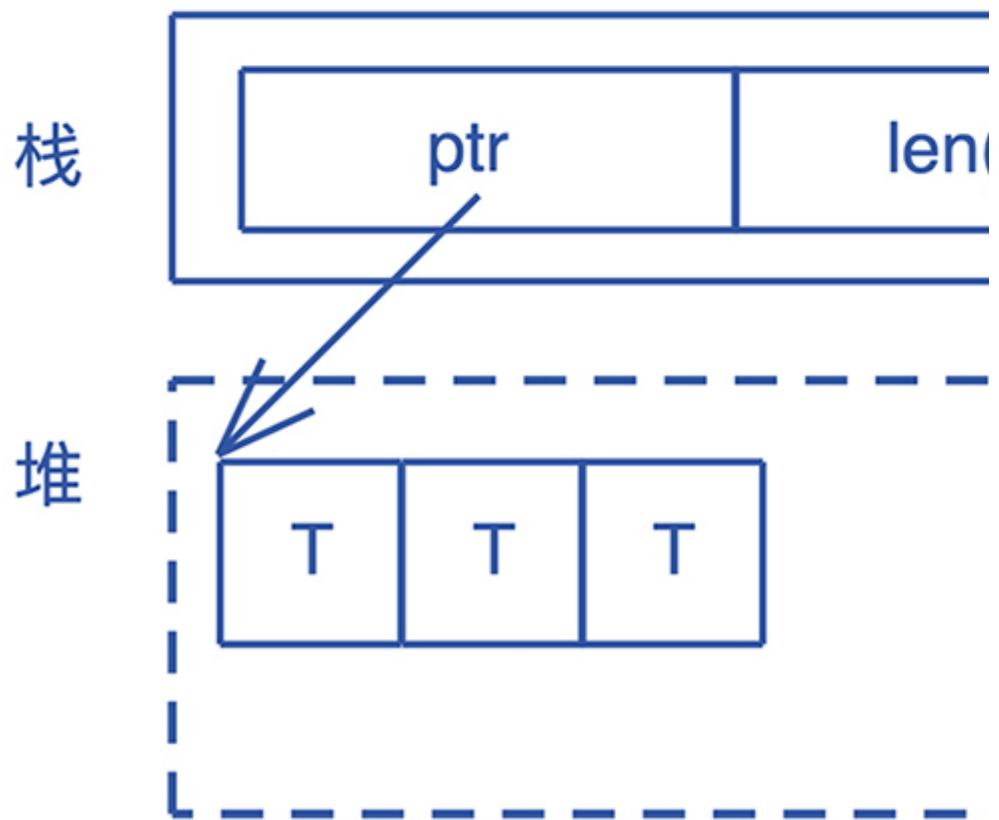


数组：[T;n]

栈



Box<[T]>



下一讲我们继续学习哈希表.....

思考题

1.在讲 `&str` 时，里面的 `print_slice1` 函数，如果写成这样可不可以？你可以尝试一下，然后说明理由。

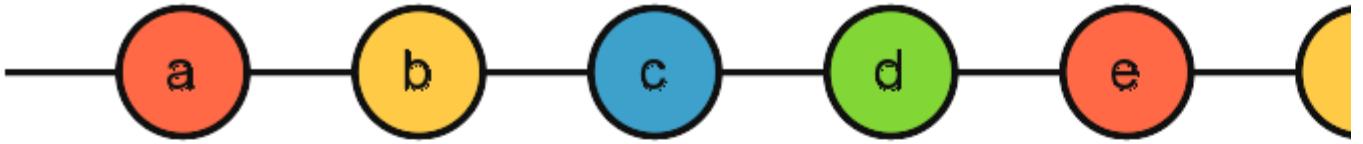
```
// fn print_slice1<T: AsRef<str>>(s: T) {  
//     println!("{:?}", s.as_ref());  
// }
```

```
fn print_slice1<T, U>(s: T)
```

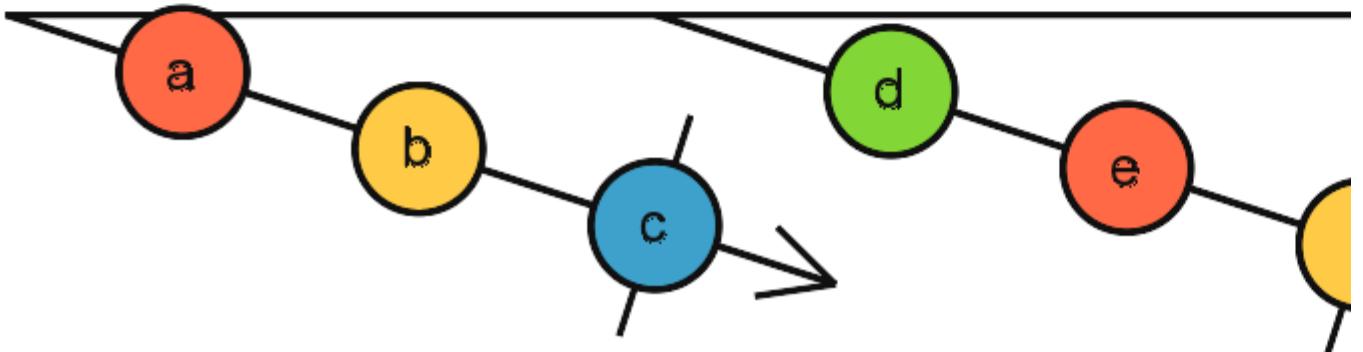
```
where
```

```
    T: AsRef<U>,  
    U: fmt::Debug,  
{  
    println!("{:?}", s.as_ref());  
}
```

2.类似 `itertools`，你可以试着开发一个新的 `Iterator` trait `IteratorExt`，为其提供 `window_count` 函数，使其可以做下图中的动作（[来源](#)）：



windowCo



感谢你的阅读，如果你觉得有收获，也欢迎你分享给你身边的朋友，邀他一起讨论。你已经完成了Rust学习的第16次打卡啦，我们下节课见。

参考资料：Rust 的 Iterator 究竟有多快？

当使用 Iterator 提供的这种函数式编程风格的时候，我们往往会担心性能。虽然我告诉你 Rust 大量使用 inline 来优化，但你可能还心存疑惑。

下面的代码和截图来自一个 Youtube 视频：[Sharing code between iOS & Android with Rust](#)，演讲者通过在使用 Iterator 处理一个很大的图片，比较 Rust / Swift / Kotlin native / C 这几种语言的性能。你也可以看到在处理迭代器时，Rust 代码和 Kotlin 或者 Swift 代码非常类似。

Chunks

```
fn resize(img: &[usize], width: usize) {
    img.chunks(width)
        .map(|slice| slice.chunks(some_size))
        .map(|chunk| { chunk
            .flatten()
            .collect()
        })
}
```

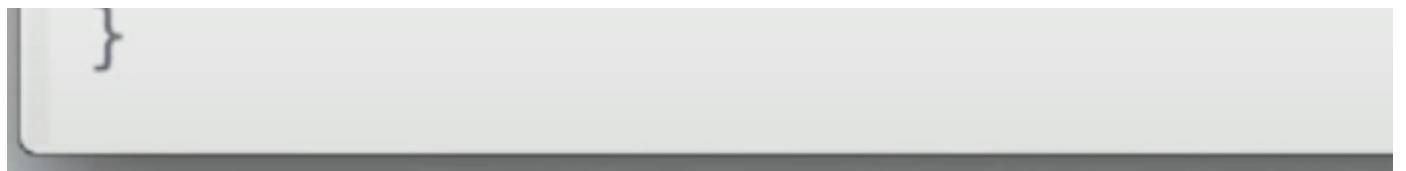
```
fun resize(image: List<Int>, width: Int) {
    return image.chunked(width)
        .map({ chunk -> chunk.chunks(some_size))
        .map({ innerChunk
            .flatten()
        })
}
```

}

Chunks

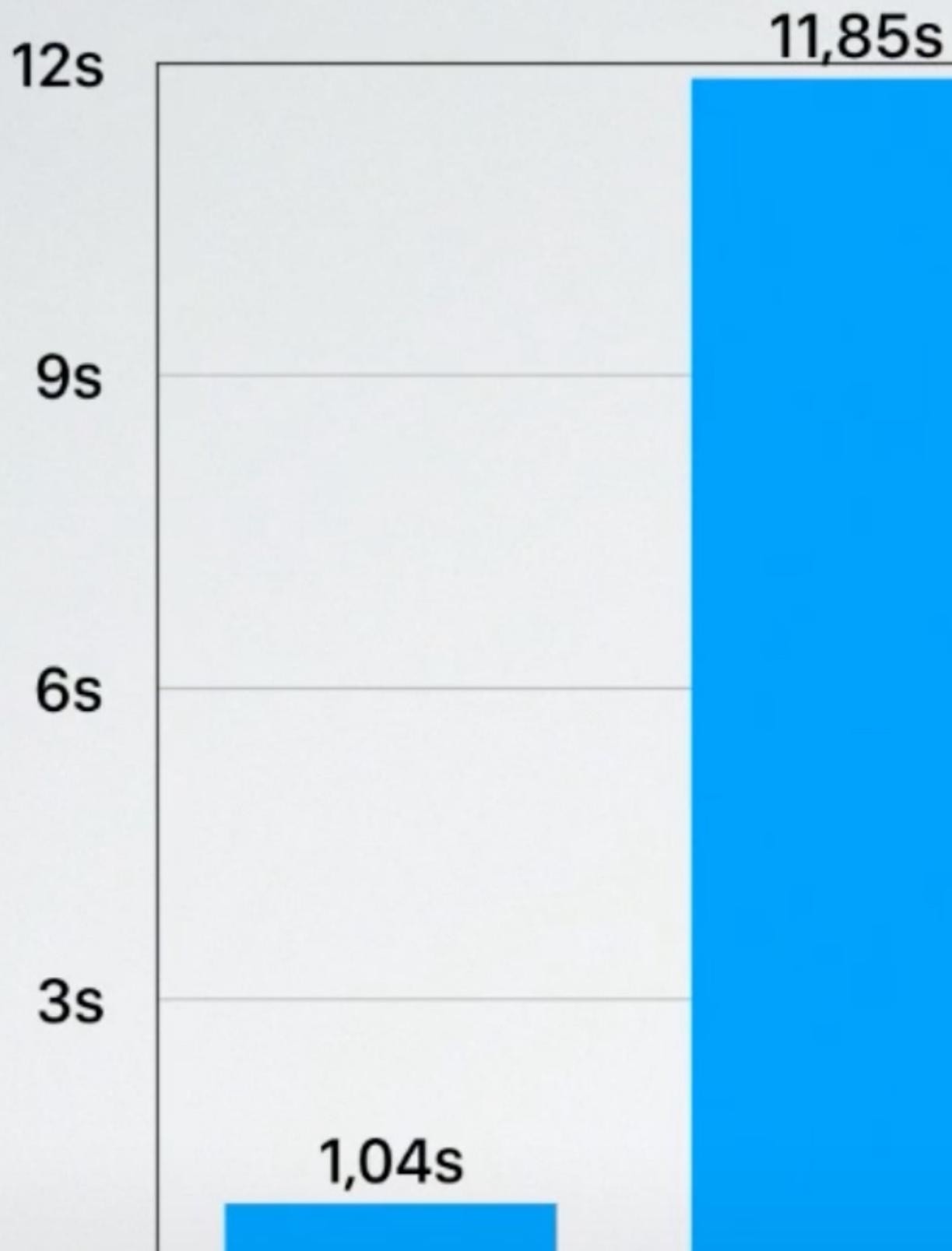
```
func resize_chunk(slice: ArraySlice<[Int]>,
                  size: Int) -> [Int] {
    return slice.chunks(size: size)
        .map { $0.reduce(into: [Int](), { result.append($0) }) }
        .reduce(into: [Int](), { result.append($0) })
}

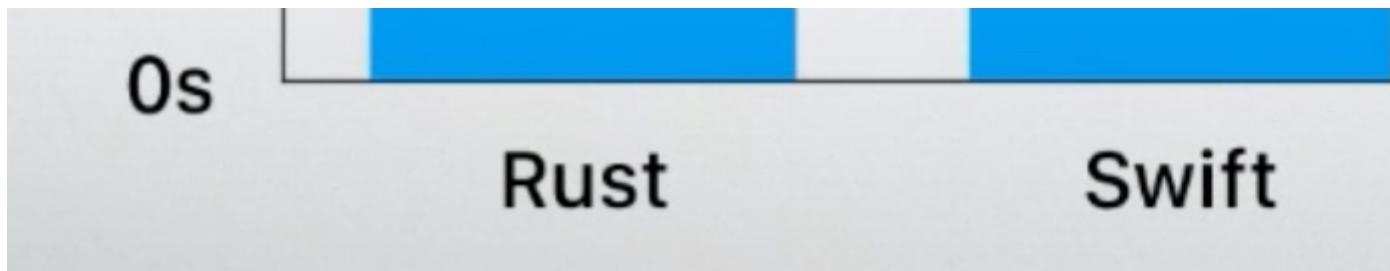
func resize(image: [Int], width: Int) -> [Int] {
    return image.chunks(size: width)
        .map { resize_chunk(slice: $0, size: width) }
        .reduce(into: [Int](), { result.append($0) })
}
```



运行结果，在函数式编程方式下（C 没有函数式编程支持，所以直接使用了 for 循环），Rust 和 C 几乎相当在1s 左右，C 比 Rust 快 20%，Swift 花了 11.8s，而 Kotlin native 直接超时：

Run Chun





所以 Rust 在对函数式编程，尤其是 Iterator 上的优化，还是非常不错的。这里面除了 inline 外，Rust 闭包的优异性能也提供了很多支持（未来我们会讲为什么）。在使用时，你完全不用担心性能。

03 | 数据结构：软件系统核心部件哈希表，内存如何布局？

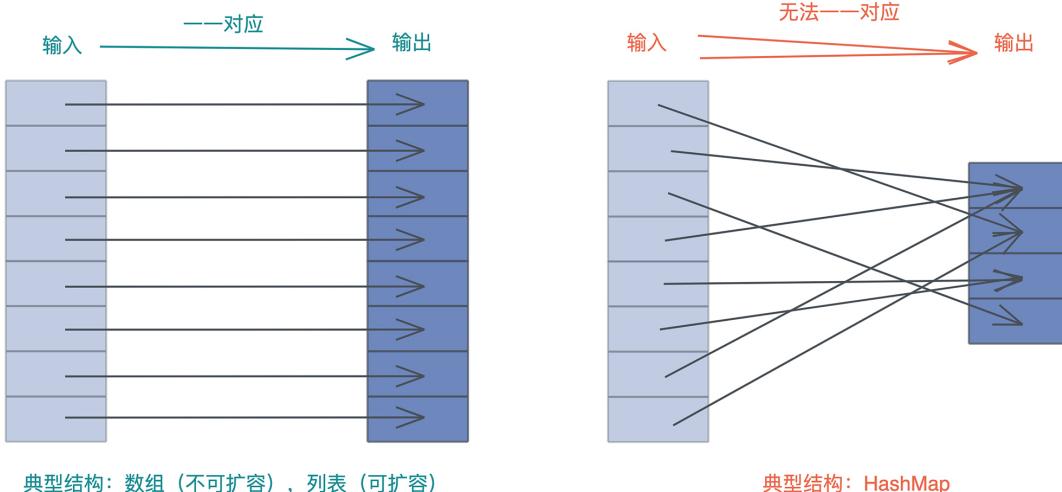
你好，我是陈天。

上一讲我们深入学习了切片，对比了数组、列表、字符串和它们的切片以及切片引用的关系。今天就继续讲 Rust 里另一个非常重要的集合容器：HashMap，也就是哈希表。

如果谈论软件开发中最重要、出镜率最高的数据结构，那哈希表一定位列其中。很多编程语言甚至将哈希表作为一种内置的数据结构，做进了语言的核心。比如 PHP 的关联数组（associate array）、Python 的字典（dict）、JavaScript 的对象（object）和 Map。

Google 的工程师 Matt Kulukundis 在 [cppCon 2017](#) 做的一个演讲，说：全世界 Google 的服务器上 1% 的 CPU 时间用来自做哈希表的计算，超过 4% 的内存用来存储哈希表。足以证明哈希表的重要性。

我们知道，哈希表和列表类似，都用于处理需要随机访问的数据结构。如果数据结构的输入和输出能一一对应，那么可以使用列表，如果无法一一对应，那么就需要使用哈希表。



极客时间

Rust 的哈希表

那 Rust 为我们提供了什么样的哈希表呢？它长什么样？性能如何？我们从官方文档学起。

如果你打开 [HashMap](#) 的文档，会看到这样一句话：

A hash map implemented with `quadratic probing` and `SIMD lookup`.

这一看就有点肾上腺素上升了，出现了两个高端词汇：二次探查（`quadratic probing`）和 SIMD 查表（`SIMD lookup`），都是什么意思？它们是Rust哈希表算法的设计核心，我们今天的学习也会围绕着这两个词展开，所以别着急，等学完相信你会理解这句话的。

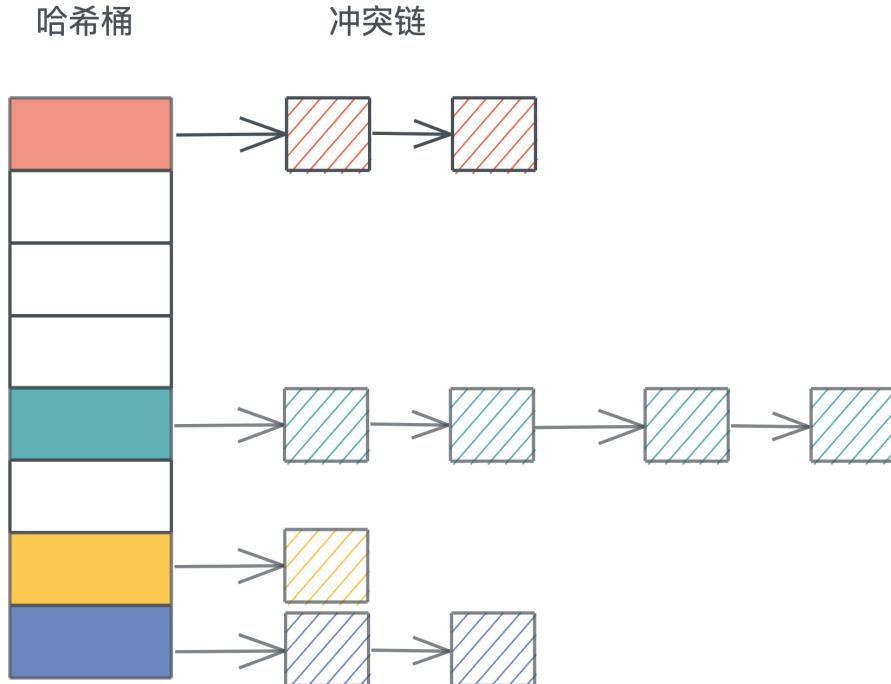
先把基础理论扫一遍。哈希表最核心的特点就是：**巨量的可能输入和有限的哈希表容量**。这就会引发哈希冲突，也就是两个或者多个输入的哈希被映射到了同一个位置，所以我们要能够处理哈希冲突。

要解决冲突，首先可以通过更好的、分布更均匀的哈希函数，以及使用更大的哈希表来缓解冲突，但无法完全解决，所以我们还需要使用冲突解决机制。

如何解决冲突？

理论上，主要的冲突解决机制有链地址法（chaining）和开放寻址法（open addressing）。

链地址法，我们比较熟悉，就是把落在同一个哈希上的数据用单链表或者双链表连接起来。这样在查找的时候，先找到对应的哈希桶（hash bucket），然后再在冲突链上挨个比较，直到找到匹配的项：

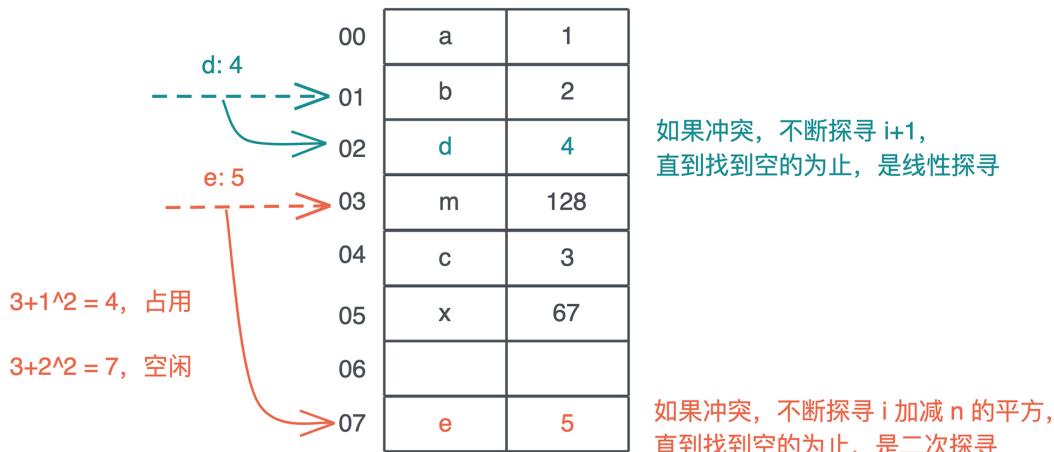


极客时间

冲突链处理哈希冲突非常直观，很容易理解和撰写代码，但缺点是哈希表和冲突链使用了不同的内存，对缓存不友好。

开放寻址法把整个哈希表看做一个大数组，不引入额外的内存，当冲突产生时，按照一定的规则把数据插入到其它空闲的位置。比如线性探寻（linear probing）在出现哈希冲突时，不断往后探寻，直到找到空闲的位置插入。

而二次探查，理论上是在冲突发生时，不断探寻哈希位置加减 n 的二次方，找到空闲的位置插入，我们看图，更容易理解：



极客时间

(图中)

示意是理论上的处理方法，实际为了性能会有很多不同的处理。)

开放寻址还有其它方案，比如二次哈希什么的，今天就不详细介绍了。

好，搞明白哈希表的二次探查的理论知识，我们可以推测，Rust 哈希表不是用冲突链来解决哈希冲突，而是用开放寻址法的二次探查来解决的。当然，后面会讲到 Rust 的二次探查和理论的处理方式有些差别。

而另一个关键词，使用 SIMD 做单指令多数据的查表，也和一会要讲到 Rust 哈希表巧妙的内存布局息息相关。

HashMap 的数据结构

进入正题，我们来看看 Rust 哈希表的数据结构是什么样子的，打开标准库的 [源代码](#)：

```
use hashbrown::hash_map as base;

#[derive(Clone)]
pub struct RandomState {
    k0: u64,
    k1: u64,
}

pub struct HashMap<K, V, S = RandomState> {
    base: base::HashMap<K, V, S>,
}
```

可以看到，HashMap 有三个泛型参数，K 和 V 代表 key / value 的类型，S 是哈希算法的状态，它默认是 RandomState，占两个 u64。RandomState 使用 SipHash 作为缺省的哈希算法，它是一个加密安全的哈希函数（cryptographically secure hashing）。

从定义中还能看到，Rust 的 HashMap 复用了 hashbrown 的 HashMap。hashbrown 是 Rust 下对 [Google Swiss Table](#) 的一个改进版实现，我们打开 hashbrown 的代码，看它的[结构](#)：

```
pub struct HashMap<K, V, S = DefaultHashBuilder, A: Allocator + Clone = Global> {  
    pub(crate) hash_builder: S,  
    pub(crate) table: RawTable<(K, V), A>,  
}
```

可以看到，HashMap 里有两个域，一个是 hash_builder，类型是刚才我们提到的标准库使用的 RandomState，还有一个是具体的 RawTable：

```
pub struct RawTable<T, A: Allocator + Clone = Global> {  
    table: RawTableInnner<A>,  
    // Tell dropck that we own instances of T.  
    marker: PhantomData<T>,  
}  
  
struct RawTableInnner<A> {  
    // Mask to get an index from a hash value. The value is one less than the  
    // number of buckets in the table.  
    bucket_mask: usize,  
  
    // [Padding], T1, T2, ..., Tlast, C1, C2, ...  
    // ^ points here  
    ctrl: NonNull<u8>,  
  
    // Number of elements that can be inserted before we need to grow the table  
    growth_left: usize,  
  
    // Number of elements in the table, only really used by len()  
    items: usize,  
  
    alloc: A,  
}
```

RawTable 中，实际上有意义的数据结构是 RawTableInnner，前四个字段很重要，我们一会讲HashMap的内存布局会再提到：

- usize 的 bucket_mask，是哈希表中哈希桶的数量减一；
- 名字叫 ctrl 的指针，它指向哈希表堆内存末端的 ctrl 区；

- `usize` 的字段 `growth_left`, 指哈希表在下次自动增长前还能存储多少数据;
- `usize` 的 `items`, 表明哈希表现在有多少数据。

这里最后的 `alloc` 字段, 和 `RawTable` 的 `marker` 一样, 只是一个用来占位的类型, 我们现在只需知道, 它用来分配在堆上的内存。

HashMap 的基本使用方法

数据结构搞清楚, 我们再看具体使用方法。Rust 哈希表的使用很简单, 它提供了一系列很方便的方法, 使用起来和其它语言非常类似, 你只要看看[文档](#), 就很容易理解。我们来写段代码, 尝试一下([代码](#)):

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    explain("empty", &map);

    map.insert('a', 1);
    explain("added 1", &map);

    map.insert('b', 2);
    map.insert('c', 3);
    explain("added 3", &map);

    map.insert('d', 4);
    explain("added 4", &map);

    // get 时需要使用引用, 并且也返回引用
    assert_eq!(map.get(&'a'), Some(&1));
    assert_eq!(map.get_key_value(&'b'), Some((&'b', &2)));

    map.remove(&'a');
    // 删除后就找不到了
    assert_eq!(map.contains_key(&'a'), false);
    assert_eq!(map.get(&'a'), None);
    explain("removed", &map);
    // shrink 后哈希表变小
    map.shrink_to_fit();
    explain("shrinked", &map);
}

fn explain<K, V>(name: &str, map: &HashMap<K, V>) {
    println!("{}: len: {}, cap: {}", name, map.len(), map.capacity());
}
```

运行这段代码, 我们可以看到这样的输出:

```
empty: len: 0, cap: 0
added 1: len: 1, cap: 3
added 3: len: 3, cap: 3
added 4: len: 4, cap: 7
removed: len: 3, cap: 7
shrinked: len: 3, cap: 3
```

可以看到，当 `HashMap::new()` 时，它并没有分配空间，容量为零，**随着哈希表不断插入数据，它会以 2的幂减一的方式增长，最小是 3**。当删除表中的数据时，原有的表大小不变，只有显式地调用 `shrink_to_fit`，才会让哈希表变小。

HashMap 的内存布局

但是通过 `HashMap` 的公开接口，我们无法看到 `HashMap` 在内存中是如何布局的，还是需要借助之前使用过的 `std::mem::transmute` 方法，来把数据结构打出来。我们把刚才的代码改一改（[代码](#)）：

```
use std::collections::HashMap;

fn main() {
    let map = HashMap::new();
    let mut map = explain("empty", map);

    map.insert('a', 1);
    let mut map = explain("added 1", map);
    map.insert('b', 2);
    map.insert('c', 3);

    let mut map = explain("added 3", map);

    map.insert('d', 4);

    let mut map = explain("added 4", map);

    map.remove(&'a');

    explain("final", map);
}

// HashMap 结构有两个 u64 的 RandomState，然后是四个 usize,
// 分别是 bucket_mask, ctrl, growth_left 和 items
// 我们 transmute 打印之后，再 transmute 回去
fn explain<K, V>(name: &str, map: HashMap<K, V>) -> HashMap<K, V> {
    let arr: [usize; 6] = unsafe { std::mem::transmute(map) };
    println!(
        "{}: bucket_mask 0x{:x}, ctrl 0x{:x}, growth_left: {}, items: {}",
        name, arr[2], arr[3], arr[4], arr[5]
    );
    unsafe { std::mem::transmute(arr) }
}
```

运行之后，可以看到：

```
empty: bucket_mask 0x0, ctrl 0x1056df820, growth_left: 0, items: 0
added 1: bucket_mask 0x3, ctrl 0x7fa0d1405e30, growth_left: 2, items: 1
added 3: bucket_mask 0x3, ctrl 0x7fa0d1405e30, growth_left: 0, items: 3
added 4: bucket_mask 0x7, ctrl 0x7fa0d1405e90, growth_left: 3, items: 4
final: bucket_mask 0x7, ctrl 0x7fa0d1405e90, growth_left: 4, items: 3
```

有意思，我们发现在运行的过程中，ctrl 对应的堆地址发生了改变。

在我的 OS X 下，一开始哈希表为空，ctrl 地址看上去是一个 TEXT/RODATA 段的地址，应该是指向了一个默认的空表地址；插入第一个数据后，哈希表分配了 4 个 bucket，ctrl 地址发生改变；在插入三个数据后，growth_left 为零，再插入时，哈希表重新分配，ctrl 地址继续改变。

刚才在探索 HashMap 数据结构时，说过 ctrl 是一个指向哈希表堆地址末端 ctrl 区的地址，所以我们可以通过这个地址，计算出哈希表堆地址的起始地址。

因为哈希表有 8 个 bucket ($0x7 + 1$)，每个 bucket 大小是 key (char) + value (i32) 的大小，也就是 8 个字节，所以一共是 64 个字节。对于这个例子，[通过 ctrl 地址减去 64，就可以得到哈希表的堆内存起始地址](#)。然后，我们可以用 rust-gdb / rust-lldb 来打印这个内存（如果你对 rust-gdb / rust-lldb 感兴趣，可以看文末的参考阅读）。

这里我用 Linux 下的 rust-gdb 设置断点，依次查看哈希表有一个、三个、四个值，以及删除一个值的状态：

```
rust-gdb ~/.target/debug/hashmap2
GNU gdb (Ubuntu 9.2-0ubuntu2) 9.2
...
(gdb) b hashmap2.rs:32
Breakpoint 1 at 0xa43e: file src/hashmap2.rs, line 32.
(gdb) r
Starting program: /home/tchen/.target/debug/hashmap2
...
# 最初的状态，哈希表为空
empty: bucket_mask 0x0, ctrl 0x5555555597be0, growth_left: 0, items: 0

Breakpoint 1, hashmap2::explain (name=..., map=...) at src/hashmap2.rs:32
32      unsafe { std::mem::transmute(arr) }
(gdb) c
Continuing.
# 插入了一个元素后，bucket 有 4 个 ( $0x3+1$ )，堆地址起始位置  $0x55555555a7af0 - 4*8(0x20)$ 
added 1: bucket_mask 0x3, ctrl 0x55555555a7af0, growth_left: 2, items: 1

Breakpoint 1, hashmap2::explain (name=..., map=...) at src/hashmap2.rs:32
32      unsafe { std::mem::transmute(arr) }
(gdb) x /12x 0x55555555a7ad0
0x55555555a7ad0: 0x00000061 0x00000001 0x00000000 0x00000000
```

```

0x5555555a7ae0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555a7af0: 0x0affffff 0xfffffff 0xfffffff 0xfffffff
(gdb) c
Continuing.
# 插入了三个元素后，哈希表没有剩余空间，堆地址起始位置不变 0x5555555a7af0 - 4*8(0x20)
added 3: bucket_mask 0x3, ctrl 0x5555555a7af0, growth_left: 0, items: 3

Breakpoint 1, hashmap2::explain (name=..., map=...) at src/hashmap2.rs:32
32      unsafe { std::mem::transmute(arr) }
(gdb) x /12x 0x5555555a7ad0
0x5555555a7ad0: 0x00000061 0x00000001 0x00000062 0x00000002
0x5555555a7ae0: 0x00000000 0x00000000 0x00000063 0x00000003
0x5555555a7af0: 0x0a72ff02 0xfffffff 0xfffffff 0xfffffff
(gdb) c
Continuing.
# 插入第四个元素后，哈希表扩容，堆地址起始位置变为 0x5555555a7b50 - 8*8(0x40)
added 4: bucket_mask 0x7, ctrl 0x5555555a7b50, growth_left: 3, items: 4

Breakpoint 1, hashmap2::explain (name=..., map=...) at src/hashmap2.rs:32
32      unsafe { std::mem::transmute(arr) }
(gdb) x /20x 0x5555555a7b10
0x5555555a7b10: 0x00000061 0x00000001 0x00000000 0x00000000
0x5555555a7b20: 0x00000064 0x00000004 0x00000063 0x00000003
0x5555555a7b30: 0x00000000 0x00000000 0x00000062 0x00000002
0x5555555a7b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555a7b50: 0xff72ffff 0x0aff6502 0xfffffff 0xfffffff
(gdb) c
Continuing.
# 删除 a 后，剩余 4 个位置。注意 ctrl bit 的变化，以及 0x61 0x1 并没有被清除
final: bucket_mask 0x7, ctrl 0x5555555a7b50, growth_left: 4, items: 3

Breakpoint 1, hashmap2::explain (name=..., map=...) at src/hashmap2.rs:32
32      unsafe { std::mem::transmute(arr) }
(gdb) x /20x 0x5555555a7b10
0x5555555a7b10: 0x00000061 0x00000001 0x00000000 0x00000000
0x5555555a7b20: 0x00000064 0x00000004 0x00000063 0x00000003
0x5555555a7b30: 0x00000000 0x00000000 0x00000062 0x00000002
0x5555555a7b40: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555a7b50: 0xff72ffff 0xfffff6502 0xfffffff 0xfffffff

```

这段输出蕴藏了很多信息，我们结合示意图来仔细梳理。

首先，插入第一个元素 ‘a’: 1 后，哈希表的内存布局如下：

栈

F

u64



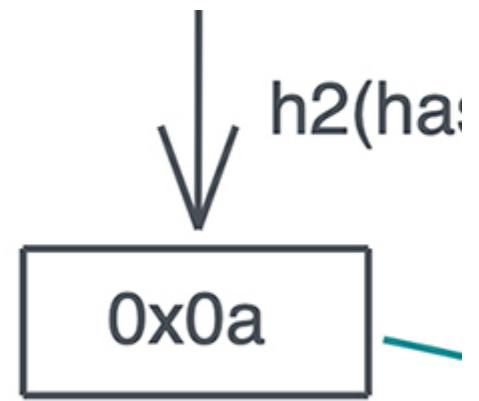
map.insert('a', 1);

SipHash



hash

-



key 'a' 的 hash 和 bucket_mask 0x3 运算后得到第 0 个位置插入。同时，这个 hash 的头 7 位取出来，在 ctrl 表中对应的位置，也就是第 0 个字节，把这个值写入。

要理解这个步骤，关键就是要搞清楚这个 ctrl 表是什么。

ctrl 表

ctrl 表的主要目的是快速查找。它的设计非常优雅，值得我们学习。

一张 ctrl 表里，有若干个 128bit 或者说 16 个字节的分组（group），group 里的每个字节叫 ctrl byte，对应一个 bucket，那么一个 group 对应 16 个 bucket。如果一个 bucket 对应的 ctrl byte 首位不为 1，就表示这个 ctrl byte 被使用；如果所有位都是 1，或者说这个字节是 0xff，那么它是空闲的。

一组 control byte 的整个 128 bit 的数据，可以通过一条指令被加载进来，然后和某个值进行 mask，找到它所在的位置。这就是一开始提到的**SIMD 查表**。

我们知道，现代 CPU 都支持单指令多数据集的操作，而Rust 充分利用了 CPU 这种能力，一条指令可以让多个相关的数据载入到缓存中处理，大大加快查表的速度。所以，Rust 的哈希表查询的效率非常高。

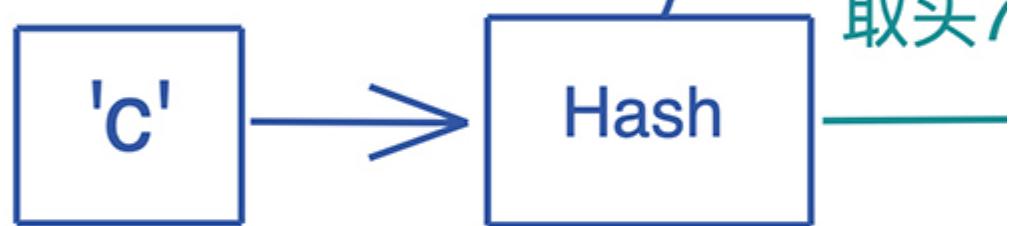
具体怎么操作，我们来看 HashMap 是如何通过 ctrl 表来进行数据查询的。假设这张表里已经添加了一些数据，我们现在要查找 key 为 ‘c’ 的数据：

1. 首先对 ‘c’ 做哈希，得到一个哈希值 h；
2. 把 h 跟 bucket_mask 做与，得到一个值，图中是 139；
3. 拿着这个 139，找到对应的 ctrl group 的起始位置，因为 ctrl group 以 16 为一组，所以这里找到 128；
4. 用 SIMD 指令加载从 128 对应地址开始的 16 个字节；
5. 对 hash 取头 7 个 bit，然后和刚刚取出的 16 个字节一起做与，找到对应的匹配，如果找到了，它（们）很概率是要找的值；
6. 如果不是，那么以二次探查（以 16 的倍数不断累积）的方式往后查找，直到找到为止。

你可以结合下图理解这个算法：

找到
用：

& bucket_mask



SIM

所以，当 HashMap 插入和删除数据，以及因此导致重新分配的时候，主要工作就是在维护这张 ctrl 表和数据的对应。

因为 ctrl 表是所有操作最先触及的内存，所以，在 HashMap 的结构中，**堆内存的指针直接指向 ctrl 表**，而不是指向堆内存的起始位置，这样可以减少一次内存的访问。

哈希表重新分配与增长

好，回到刚才讲的内存布局继续说。在插入第一条数据后，我们的哈希表只有 4 个 bucket，所以只有头 4 个字节的 ctrl 表有用。随着哈希表的增长，bucket 不够，就会导致重新分配。由于 bucket_mask 永远比 bucket 数量少 1，所以插入三个元素后就会重新分配。

根据 rust-gdb 中得到的信息，我们看插入三个元素后没有剩余空间的哈希表，在加入 ‘d’: 4 时，是如何增长的。

首先，**哈希表会按幂扩容**，从 4 个 bucket 扩展到 8 个 bucket。

这会导致分配新的堆内存，然后原来的 ctrl table 和对应的kv数据会被移动到新的内存中。这个例子里因为 char 和 i32 实现了 Copy trait，所以是拷贝；如果 key 的类型是 String，那么只有 String 的 24 个字节 (ptr|cap|len) 的结构被移动，String 的实际内存不需要变动。

在移动的过程中，会涉及**哈希的重分配**。从下图可以看到，‘a’ / ‘c’ 的相对位置和它们的 ctrl byte 没有变化，但重新做 hash 后，‘b’ 的 ctrl byte 和位置都发生了变化：

栈

RandomState

u64





删除一个值

明白了哈希表是如何增长的，我们再来看删除的时候会发生什么。

当要在哈希表中删除一个值时，整个过程和查找类似，先要找到要被删除的 key 所在的位置。在找到具体位置后，**并不需要实际清除内存，只需要将它的 ctrl byte 设回 0xff**（或者标记成删除状态）。这样，这个 bucket 就可以被再次使用了：

栈

u64

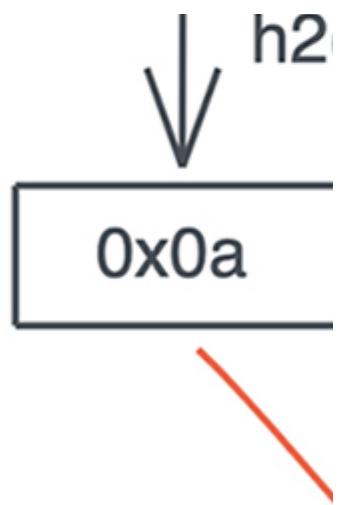
[

map.remove

Si|

↓

hash



change (

这里有一个问题，当 key/value 有额外的内存时，比如 String，它的内存不会立即回收，只有在下一次对应的 bucket 被使用时，让 HashMap 不再拥有这个 String 的所有权之后，这个 String 的内存才被回收。我们看下面的示意图：

栈

RandomState

u64



堆

V8

一般来说，这并不会带来什么问题，顶多是内存占用率稍高一些。但某些极端情况下，比如在哈希表中添加大量内容，又删除大量内容后运行，这时你可以通过 `shrink_to_fit` / `shrink_to` 释放掉不需要的内存。

让自定义的数据结构做 Hash key

有时候，我们需要让自定义的数据结构成为 `HashMap` 的 key。此时，要使用到三个 trait: `Hash`、`PartialEq`、`Eq`，不过这三个 trait 都可以通过派生宏自动生成。其中：

- 实现了 `Hash`，可以让数据结构计算哈希；
- 实现了 `PartialEq`/`Eq`，可以让数据结构进行相等和不相等的比较。`Eq` 实现了比较的自反性 ($a == a$)、对称性 ($a == b$ 则 $b == a$) 以及传递性 ($a == b$, $b == c$, 则 $a == c$)，`PartialEq` 没有实现自反性。

我们可以写个例子，看看自定义数据结构如何支持 `HashMap`:

```
use std::{
    collections::{hash_map::DefaultHasher, HashMap},
    hash::{Hash, Hasher},
};

// 如果要支持 Hash，可以用 #[derive(Hash)]，前提是每个字段都实现了 Hash
// 如果要能作为 HashMap 的 key，还需要 PartialEq 和 Eq
#[derive(Debug, Hash, PartialEq, Eq)]
struct Student<'a> {
    name: &'a str,
```

```

    age: u8,
}

impl<'a> Student<'a> {
    pub fn new(name: &'a str, age: u8) -> Self {
        Self { name, age }
    }
}
fn main() {
    let mut hasher = DefaultHasher::new();
    let student = Student::new("Tyr", 18);
    // 实现了 Hash 的数据结构可以直接调用 hash 方法
    student.hash(&mut hasher);
    let mut map = HashMap::new();
    // 实现了 Hash / PartialEq / Eq 的数据结构可以作为 HashMap 的 key
    map.insert(student, vec!["Math", "Writing"]);
    println!("hash: 0x{:x}, map: {:?}", hasher.finish(), map);
}

```

HashSet / BTreeMap / BTreeSet

最后我们简单讲讲和 HashMap 相关的其它几个数据结构。

有时我们只需要简单确认元素是否在集合中，如果用 HashMap 就有些浪费空间了。这时可以用 HashSet，它就是简化的 HashMap，可以用来存放无序的集合，定义直接是 HashMap<K, ()>：

```

use hashbrown::hash_set as base;

pub struct HashSet<T, S = RandomState> {
    base: base::HashSet<T, S>,
}

pub struct HashSet<T, S = DefaultHashBuilder, A: Allocator + Clone = Global> {
    pub(crate) map: HashMap<T, (), S, A>,
}

```

使用 HashSet 查看一个元素是否属于集合的效率非常高。

另一个和 HashMap 一样常用的数据结构就是BTreeMap了。BTreeMap 是内部使用 [B-tree](#) 来组织哈希表的数据结构。另外 BTreeSet 和 HashSet 类似，是 BTreeMap 的简化版，可以用来存放有序集合。

我们这里重点看下BTreeMap，它的数据结构如下：

```

pub struct BTreeMap<K, V> {
    root: Option<Root<K, V>>,
    length: usize,
}

pub type Root<K, V> = NodeRef<marker::Owned, K, V, marker::LeafOrInternal>;

pub struct NodeRef<BorrowType, K, V, Type> {
    height: usize,
    node: NonNull<LeafNode<K, V>>,
    _marker: PhantomData<(BorrowType, Type)>,
}

struct LeafNode<K, V> {
    parent: Option<NonNull<InternalNode<K, V>>>,
    parent_idx: MaybeUninit<u16>,
    len: u16,
    keys: [MaybeUninit<K>; CAPACITY],
    vals: [MaybeUninit<V>; CAPACITY],
}

struct InternalNode<K, V> {
    data: LeafNode<K, V>,
    edges: [MaybeUninit<BoxedNode<K, V>>; 2 * B],
}

```

和 HashMap 不同的是， BTreeMap 是有序的。我们看个例子 ([代码](#)) :

```

use std::collections::BTreeMap;

fn main() {
    let map = BTreeMap::new();
    let mut map = explain("empty", map);

    for i in 0..16usize {
        map.insert(format!("Tyr {}", i), i);
    }

    let mut map = explain("added", map);

    map.remove("Tyr 1");

    let map = explain("remove 1", map);

    for item in map.iter() {
        println!("{:?}", item);
    }
}

// BTreeMap 结构有 height, node 和 length
// 我们 transmute 打印之后，再 transmute 回去
fn explain<K, V>(name: &str, map: BTreeMap<K, V>) -> BTreeMap<K, V> {
    let arr: [usize; 3] = unsafe { std::mem::transmute(map) };
    println!(
        "BTreeMap<{}, {}> with height {}, length {}, and root node at index {}",
        name, std::any::type_name::(), arr[0], arr[1], arr[2]
    );
}

```

```
    " {}: height: {}, root node: 0x{:x}, len: 0x{:x}",
    name, arr[0], arr[1], arr[2]
);
unsafe { std::mem::transmute(arr) }
}
```

它的输出如下：

```
empty: height: 0, root node: 0x0, len: 0x0
added: height: 1, root node: 0x7f8286406190, len: 0x10
remove 1: height: 1, root node: 0x7f8286406190, len: 0xf
("Tyr 0", 0)
("Tyr 10", 10)
("Tyr 11", 11)
("Tyr 12", 12)
("Tyr 13", 13)
("Tyr 14", 14)
("Tyr 15", 15)
("Tyr 2", 2)
("Tyr 3", 3)
("Tyr 4", 4)
("Tyr 5", 5)
("Tyr 6", 6)
("Tyr 7", 7)
("Tyr 8", 8)
("Tyr 9", 9)
```

可以看到，在遍历时，BTreeMap 会按照 key 的顺序把值打印出来。如果你想让自定义的数据结构可以作为 BTreeMap 的 key，那么需要实现 [PartialOrd](#) 和 [Ord](#)，这两者的关系和 [PartialEq / Eq](#) 类似，[PartialOrd](#) 也没有实现自反性。同样的，[PartialOrd](#) 和 [Ord](#) 也可以通过派生宏来实现。

小结

在学习数据结构的时候，常用数据结构的内存布局和基本算法你一定要理解清楚，对它在不同情况下如何增长，也要尽量做到心里有数。

这一讲我们花大精力详细学习了 HashMap 的数据结构以及算法的基本思路，算是抛砖引玉。这门课无论多深入讲解，也只能触及 Rust 整个生态圈的九牛一毛，不可能面面俱到。

我的原则是“授人以鱼不如授人以渔”，在你掌握这样的分析方法后，以后遇到标准库或者第三方库的其它的数据结构，也可以用类似的方法深入探索学习。

此外，我们程序员学东西，会用是第一层，知道它是如何设计的是第二层，能够自己写出来才是第三层。Rust借鉴的 Google Swiss table 算法简单精巧，虽然 hashbrown 在实现时，为了最大化性能和利用 SSE 指令集，使用了很多 unsafe 代码，但我们撰写一个性能不那么好的 safe 版本，并不是复杂的事情，非常推荐你实现一下。

集合类型我们就暂时讲解到这里，未来实战要使用到某些数据结构时，比如 VecDeque，我们再深入探索。其他的集合类型，你也可以在要用的时候自行阅读[文档](#)。

如果你想了解这两讲中集合类型的时间复杂度，可以看下表（[来源](#)）：

HashMap

BTreeMap

Vec

VecDeque

LinkedList

C

思考题

1.修改下面代码的错误，使其编译通过 ([代码](#)) 。

```
use std::collections::BTreeMap;

#[derive(Debug)]
struct Name {
    pub name: String,
    pub flags: u32,
}

impl Name {
    pub fn new(name: impl AsRef<str>, flags: u32) -> Self {
        Self {
            name: name.as_ref().to_string(),
            flags,
        }
    }
}

fn main() {
    let mut map = BTreeMap::new();
    map.insert(Name::new("/etc/password", 0x1), 12);
    map.insert(Name::new("/etc/hosts", 0x1), 4);
    map.insert(Name::new("/home/tchen", 0x0), 28);
    for item in map.iter() {
        println!("{:?}", item);
    }
}
```

```
&nbsp; &nbsp; }  
}
```

2.思考一下，如果一个 session 表的 key 是 (Source IP、Source Port、Dst IP、Dst Port、Proto) 这样的长度 15 个字节的五元组，value 是 200 字节的 Session 结构，要容纳 1200000 个 Session，整个哈希表要占多大的堆内存？内存的利用率如何？

3.使用文中同样的方式，结合 rust-gdb / rust-lldb 探索 BTreeMap。你能画出来在插入以 26 个字母为 key，1~26 为 value 后的 BTreeMap 的内存布局么？

今天你完成了Rust学习的第17次打卡，我们下节课见。

参考资料

1.为什么 Rust 的 HashMap 要缺省采用加密安全的哈希算法？

我们知道哈希表在软件系统中的重要地位，但哈希表在最坏情况下，如果绝大多数 key 的 hash 都碰撞在一起，性能会到 $O(n)$ ，这会极大拖累系统的效率。

比如 1M 大小的 session 表，正常情况下查表速度是 $O(1)$ ，但极端情况下，需要比较 1M 个数据后才能找到，这样的系统就容易被 DoS 攻击。所以如果不是加密安全的哈希函数，只要黑客知道哈希算法，就可以构造出大量的 key 产生足够的哈希碰撞，造成目标系统 DoS。

[SipHash](#) 就是为了回应 DoS 攻击而创建的哈希算法，虽然和 sha2 这样的加密哈希不同（不要将 SipHash 用于加密！），但它可以提供类似等级的安全性。把 SipHash 作为 HashMap 的缺省的哈希算法，Rust 可以避免开发者在不知情的情况下被 DoS，就像曾经在 [Web 世界](#)发生的那样。

当然，这一切的代价是性能损耗，虽然 SipHash 非常快，但它比 hashbrown 缺省使用的 [Ahash](#) 慢了不少。如果你确定使用的 HashMap 不需要 DoS 防护（比如一个完全内部使用的 HashMap），那么可以用 Ahash 来替换。你只需要使用 Ahash 提供的 RandomState 即可：

```
use ahash::{AHasher, RandomState};  
use std::collections::HashMap;  
let mut map: HashMap<char, i32, RandomState> = HashMap::default();  
map.insert('a', 1);
```

2.如何使用 rust-gdb / rust-lldb？

之前的愚昧之巅[加餐](#)提过 `gdb` / `lldb`，今天就是使用示例。没有使用过的朋友，可以看看它们的文档了解一下。

`gdb` 适合在 Linux 下，`lldb` 可以在 OS X 下调试 Rust 程序。[rust-gdb](#) / [rust-lldb](#) 提供了一些对 Rust 更友好的 pretty-print 功能，在安装 Rust 时，它们也会被安装。使用过 `gdb` 的同学，可以看 [gdb 速查手册](#)，也可以看看 [gdb/lldb 命令对应手册](#)。

我一般不用它们调试程序。**不管任何语言，如果开发时，你发现自己总在设置断点调试程序，说明你撰写代码的方式有问题。**要么，没有把接口和算法设计清楚，想到哪写到哪；要么，是你的函数写得过于复杂，太多状态纠缠，没有遵循 SRP (Single Responsibility Principle) 。

好的代码是写出来的，不是调出来的。与其把时间花在调试上，不如把时间花在设计、日志，以及单元测试上。所以，`gdb/lldb` 对我来说，是一个理解数据结构在内存中布局以及探索算法如何运行的工具。你可以仔细阅读文中展示的 `gdb` session 和与之相关的代码，看看如何构造代码来结合 `gdb` 探索 HashMap 在不同状态下的行为。

如果你觉得有收获，也欢迎分享给你身边的朋友，邀TA一起讨论~

错误处理

错误处理：为什么Rust的错误处理与众不同？

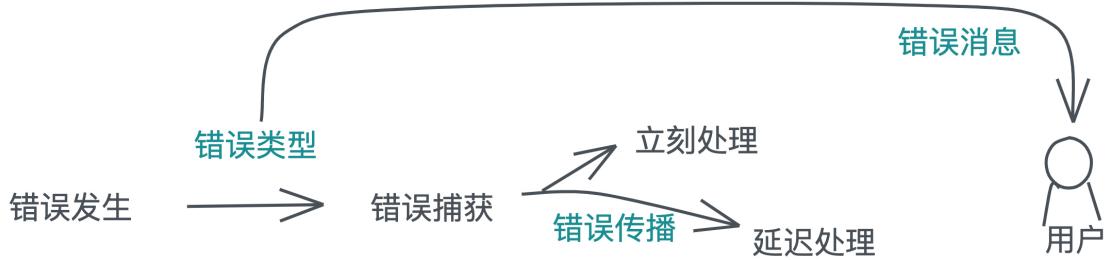
你好，我是陈天。

作为被线上业务毒打过的开发者，我们都对墨菲定律刻骨铭心。任何一个系统，只要运行的时间足够久，或者用户的规模足够大，极小概率的错误就一定会发生。比如，主机的磁盘可能被写满、数据库系统可能会脑裂、上游的服务比如 CDN 可能会宕机，甚至承载服务的硬件本身可能损坏等等。

因为我们平时写练习代码，一般只会关注正常路径，可以对小概率发生的错误路径置之不理；**但在实际生产环境中，任何错误只要没有得到妥善处理，就会给系统埋下隐患**，轻则影响开发者用户体验，重则会给系统带来安全上的问题，马虎不得。

在一门编程语言中，控制流程是语言的核心流程，而错误处理又是控制流程的重要组成部分。

语言优秀的错误处理能力，会大大减少错误处理对整体流程的破坏，让我们写代码更行云流水，读起来心智负担也更小。



对我们开发者来说，错误处理包含这么几部分：

1. 当错误发生时，用合适的错误类型捕获这个错误。
2. 错误捕获后，可以立刻处理，也可以延迟到不得不处理的地方再处理，这就涉及到错误的传播（propagate）。
3. 最后，根据不同的错误类型，给用户返回合适的、帮助他们理解问题所在的错误消息。

作为一门极其注重用户体验的编程语言，Rust 从其它优秀的语言中，尤其是 Haskell，吸收了错误处理的精髓，并以自己独到的方式展现出来。

错误处理的主流方法

在详细介绍 Rust 的错误处理方式之前，让我们稍稍放慢脚步，看看错误处理的三种主流方法以及其他语言是如何应用这些方法的。

使用返回值（错误码）

使用返回值来表征错误，是最古老也是最实用的一种方式，它的使用范围很广，从函数返回值，到操作系统的系统调用的错误码 errno、进程退出的错误码 retval，甚至 HTTP API 的状态码，都能看到这种方法的身影。

举个例子，在 C 语言中，如果 `fopen(filename)` 无法打开文件，会返回 `NULL`，调用者通过判断返回值是否为 `NULL`，来进行相应的错误处理。

我们再看个例子：

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

单看这个接口，我们很难直观了解，当读文件出错时，错误是如何返回的。从文档中，我们知道，如果返回的 size_t 和传入的 size_t 不一致，那么要么发生了错误，要么是读到文件尾（EOF），调用者要进一步通过 ferror 才能得到更详细的错误。

像 C 这样，通过返回值携带错误信息，有很多局限。返回值有它原本的语义，强行把错误类型嵌入到返回值原本的语义中，需要全面且实时更新的文档，来确保开发者能正确区别对待，正常返回和错误返回。

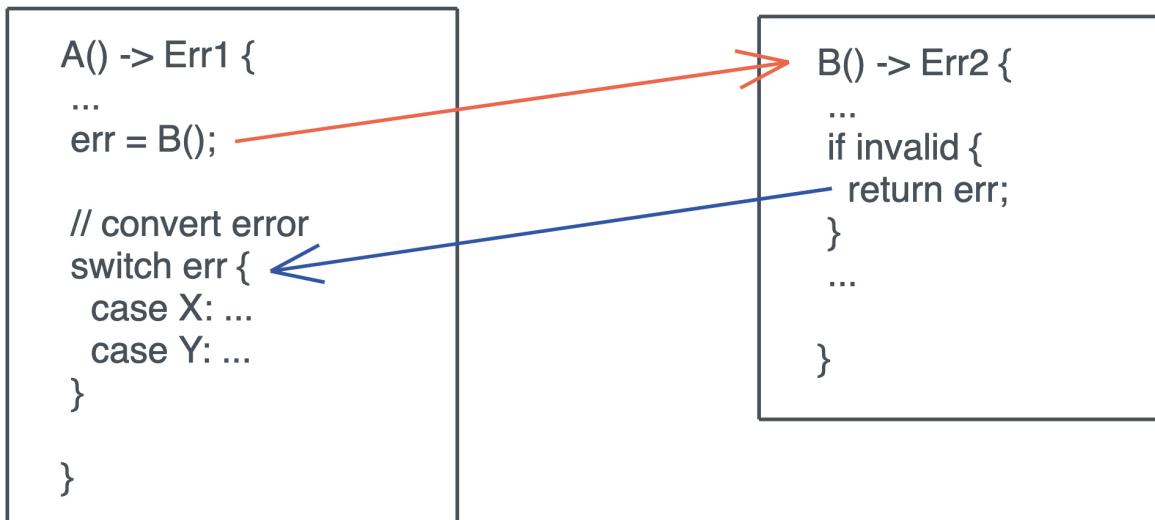
所以 Golang 对其做了扩展，在函数返回的时候，可以专门携带一个错误对象。比如上文的 fread，在 Golang 下可以这么定义：

```
func Fread(file *File, b []byte) (n int, err error)
```

Golang这样，区分开错误返回和正常返回，相对 C 来说进了一大步。

但是使用返回值的方式，始终有个致命的问题：**在调用者调用时，错误就必须得到处理或者显式的传播。**

如果函数 A 调用了函数 B，在 A 返回错误的时候，就要把 B 的错误转换成 A 的错误，显示出来。如下图所示：



这样写出来的代码会非常冗长，对我们开发者的用户体验不太好。如果不处理，又会丢掉这个错误信息，造成隐患。

另外，**大部分生产环境下的错误是嵌套的**。一个 SQL 执行过程中抛出的错误，可能是服务器出错，而更深层次的错误可能是，连接数据库服务器的 TLS session 状态异常。

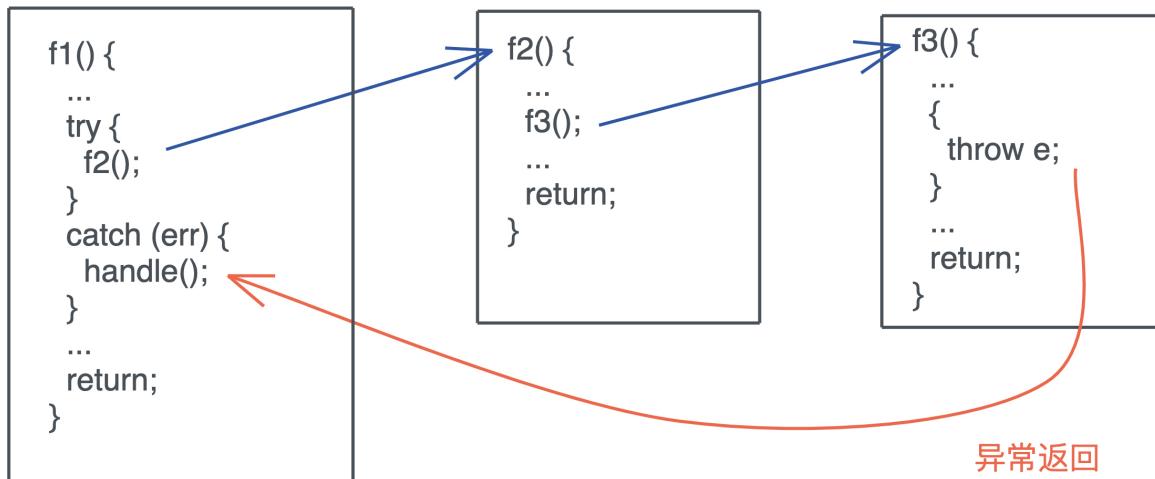
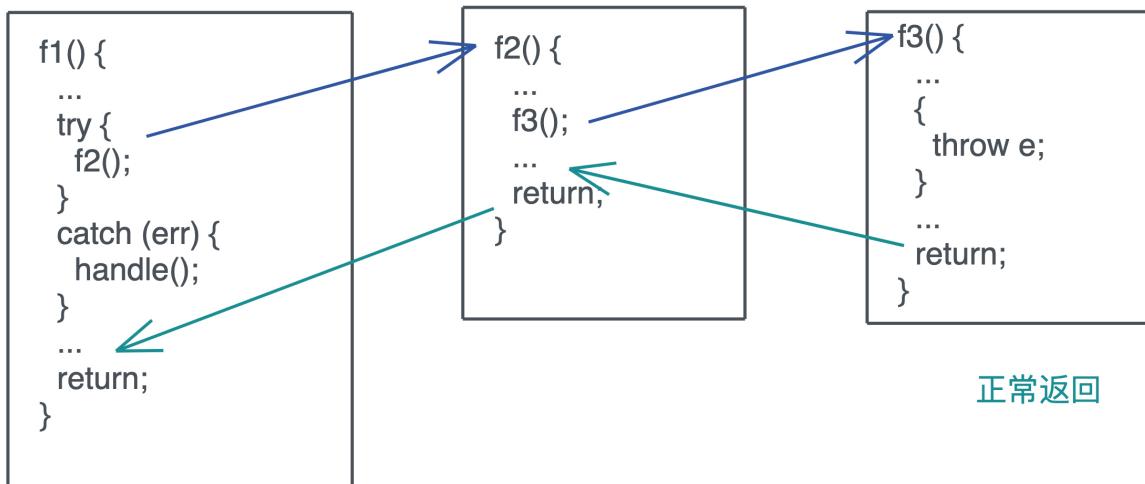
其实知道服务器出错之外，我们更需要清楚服务器出错的内在原因。因为服务器出错这个表层错误会提供给最终用户，而出错的深层原因要提供给我们自己，服务的维护者。但是这样的嵌套错误在 C / Golang 都是很难完美表述的。

使用异常

因为返回值不利于错误的传播，有诸多限制，Java 等很多语言使用异常来处理错误。

你可以把异常看成一种**关注点分离**（Separation of Concerns）：**错误的产生和错误的处理完全被分隔开，调用者不必关心错误，而被调者也不强求调用者关心错误。**

程序中任何可能出错的地方，都可以抛出异常；而异常可以通过栈回溯（stack unwind）被一层层自动传递，直到遇到捕获异常的地方，如果回溯到 main 函数还无人捕获，程序就会崩溃。如下图所示：



使用异常来返回错误可以极大地简化错误处理的流程，它解决了返回值的传播问题。

然而，上图中异常返回的过程看上去很直观，就像数据库中的事务（transaction）在出错时会被整体撤销（rollback）一样。但实际上，这个过程远比你想象的复杂，而且需要额外操心[异常安全（exception safety）](#)。

我们看下面用来切换背景图片的（伪）代码：

```

void transition(...) {
    lock(&mutex);
    delete background;
    ++changed;
    background = new Background(...);
}

```

```
unlock(&mutex);
}
```

试想，如果在创建新的背景时失败，抛出异常，会跳过后续的处理流程，一路栈回溯到 try catch 的代码，那么，这里锁住的 mutex 无法得到释放，而已有的背景被清空，新的背景没有创建，程序进入到一个奇怪的状态。

确实在大多数情况下，用异常更容易写代码，但当异常安全无法保证时，程序的正确性会受到很大的挑战。因此，你在使用异常处理时，需要特别注意异常安全，尤其是在并发环境下。

而比较讽刺的是，保证异常安全的第一个原则就是：[避免抛出异常](#)。这也是 Golang 在语言设计时避开了常规的异常，[走向返回值的老路](#)的原因。

异常处理另外一个比较严重的问题是：开发者会滥用异常。只要有错误，不论是否严重、是否可恢复，都一股脑抛个异常。到了需要的地方，捕获一下了之。殊不知，异常处理的开销要比处理返回值大得多，滥用会有很多额外的开销。

使用类型系统

第三种错误处理的方法就是使用类型系统。其实，在使用返回值处理错误的时候，我们已经看到了类型系统的雏形。

错误信息既然可以通过已有的类型携带，或者通过多返回值的方式提供，那么[通过类型来表征错误，使用一个内部包含正常返回类型和错误返回类型的复合类型](#)，通过类型系统来强制错误的处理和传递，是不是可以达到更好的效果呢？

的确如此。这种方式被大量使用在有强大类型系统支持的函数式编程语言中，如 Haskell/Scala/Swift。其中最典型的包含了错误类型的复合类型是 Haskell 的 Maybe 和 Either 类型。

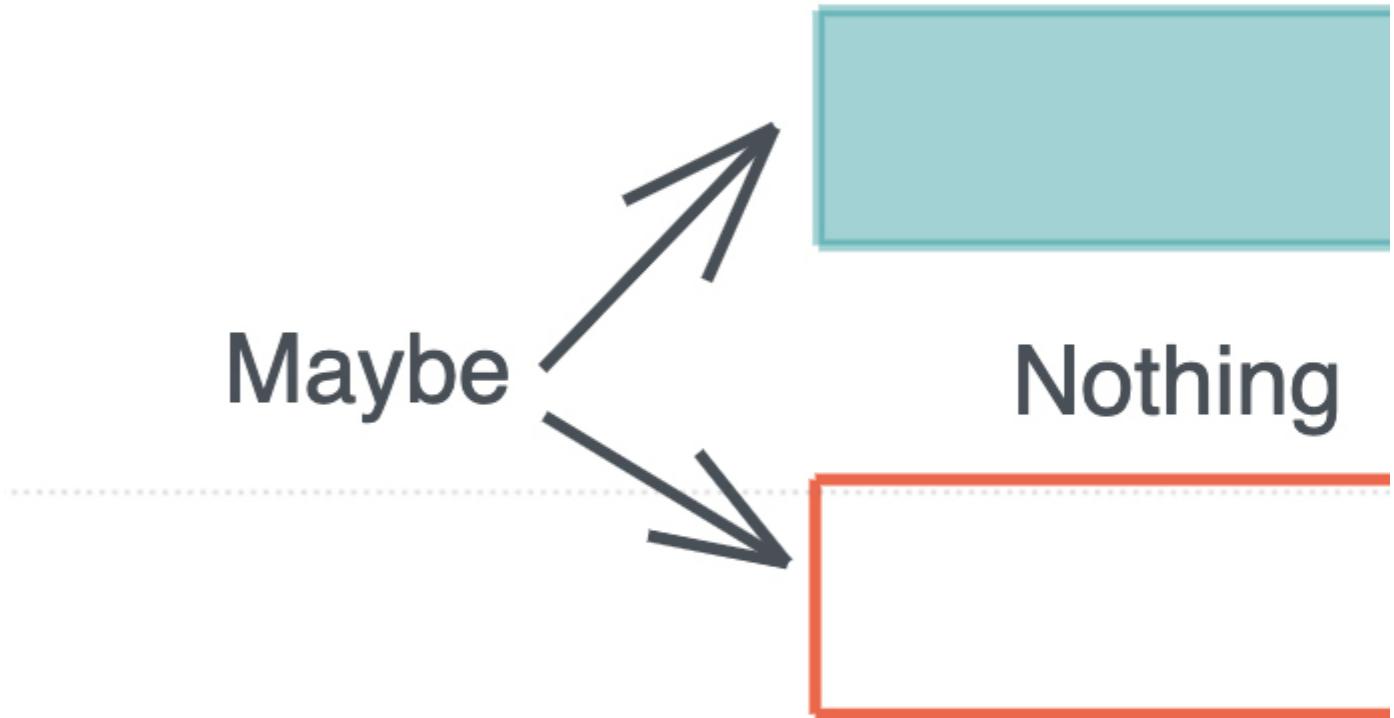
Maybe 类型允许数据包含一个值（Just）或者没有值（Nothing），这对简单的不需要类型的错误很有用。还是以打开文件为例，如果我们只关心成功打开文件的句柄，那么 Maybe 就足够了。

当我们需要更为复杂的错误处理时，我们可以使用 Either 类型。它允许数据是 Left a 或者 Right b。其中，a 是运行出错的数据类型，b 可以是成功的数据类型。

Just(a)

Maybe

Nothing



我们可以看到，这种方法依旧是通过返回值返回错误，但是错误被包裹在一个完整的、必须处理的类型中，比 Golang 的方法更安全。

我们前面提到，使用返回值返回错误的一大缺点是，错误需要被调用者立即处理或者显式传递。但是使用 Maybe / Either 这样的类型来处理错误的好处是，我们可以用函数式编程的方法简化错误的处理，比如 map、fold 等函数，让代码相对不那么冗余。

需要注意的是，很多不可恢复的错误，如“磁盘写满，无法写入”的错误，使用异常处理可以避免一层层传递错误，让代码简洁高效，所以大多数使用类型系统来处理错误的语言，会同时使用异常处理作为补充。

Rust 的错误处理

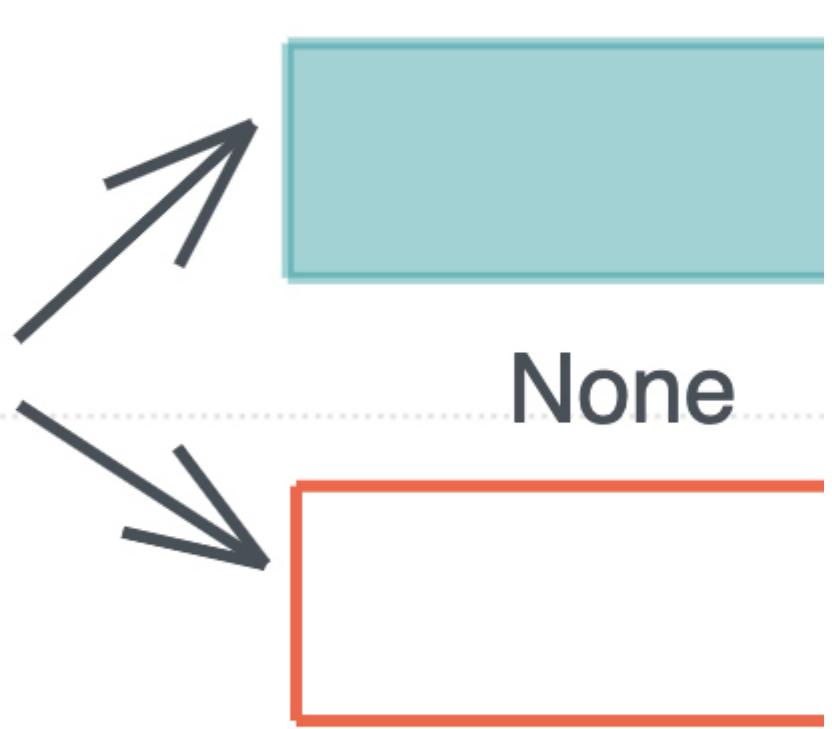
由于诞生的年代比较晚，Rust 有机会从已有的语言中学习到各种错误处理的优劣。对于 Rust 来说，目前的几种方式相比而言，最佳的方法是，使用类型系统来构建主要的错误处理流程。

Rust 偷师 Haskell，构建了对标 Maybe 的 Option 类型和对标 Either 的 Result 类型。

Some(T)

Option

None



Option 和 Result

Option 是一个 enum，其定义如下：

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

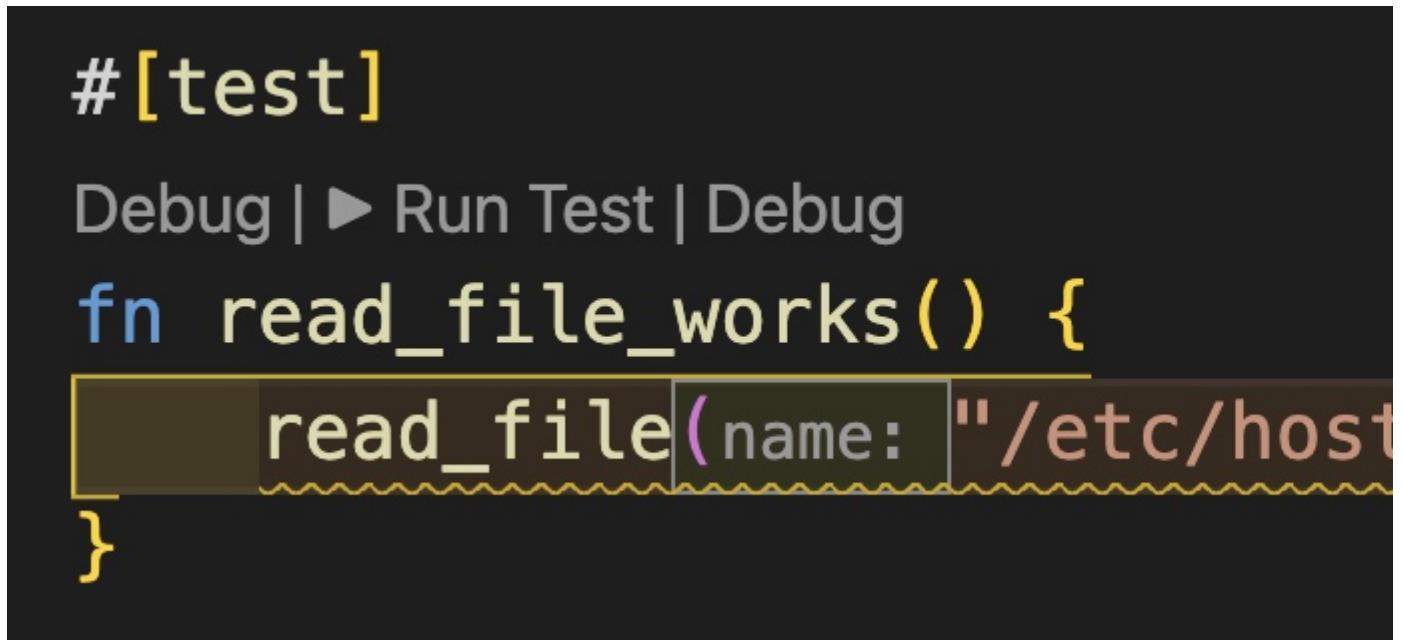
它可以承载有值/无值这种最简单的错误类型。

Result 是一个更加复杂的 enum，其定义如下：

```
#[must_use = "this `Result` may be an `Err` variant, which should be handled"]
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

当函数出错时，可以返回 Err(E)，否则 Ok(T)。

我们看到，Result 类型声明时还有个 must_use 的标注，编译器会对有 must_use 标注的所有类型做特殊处理：如果该类型对应的值没有被显式使用，则会告警。这样，保证错误被妥善处理。如下图所示：



The screenshot shows a code editor with a dark theme. At the top, there's a yellow "#[test]" annotation. Below it is a toolbar with "Debug" and "Run Test" buttons. The main code area contains a function definition:

```
fn read_file_works() {  
    read_file(name: "/etc/host")  
}
```

The line "read_file(name: "/etc/host")" is highlighted with a yellow rectangular background. A wavy yellow underline is placed under the word "read_file". This visual cue indicates that the function's return value is being discarded and is flagged by the compiler as unused.

这里，如果我们调用 read_file 函数时，直接丢弃返回值，由于 #[must_use] 的标注，Rust 编译器报警，要求我们使用其返回值。

这虽然可以极大避免遗忘错误的显示处理，但如果我们并不关心错误，只需要传递错误，还是会写出像 C 或者 Golang 一样比较冗余的代码。怎么办？

? 操作符

好在 Rust 除了有强大的类型系统外，还具备元编程的能力。早期 Rust 提供了 `try!` 宏来简化错误的显式处理，后来为了进一步提升用户体验，`try!` 被进化成 `?` 操作符。

所以在 Rust 代码中，如果你只想传播错误，不想就地处理，可以用 `?` 操作符，比如（[代码](#)）：

```
use std::fs::File;
use std::io::Read;

fn read_file(name: &str) -> Result<String, std::io::Error> {
    let mut f = File::open(name)?;
    let mut contents = String::new();
    f.read_to_string(&mut contents)?;
    Ok(contents)
}
```

通过 `?` 操作符，Rust 让错误传播的代价和异常处理不相上下，同时又避免了异常处理的诸多问题。

`?` 操作符内部被展开成类似这样的代码：

```
match result {
    Ok(v) => v,
    Err(e) => return Err(e.into())
}
```

所以，我们可以方便地写出类似这样的代码，简洁易懂，可读性很强：

```
fut
.await?
.process()?
.next()
.await?;
```

整个代码的执行流程如下：

fut



await

?

虽然 ? 操作符使用起来非常方便，但你要注意在不同的错误类型之间是无法直接使用的，需要实现 From trait 在二者之间建立起转换的桥梁，这会带来额外的麻烦。我们暂且把这个问题放下，稍后我们会谈到解决方案。

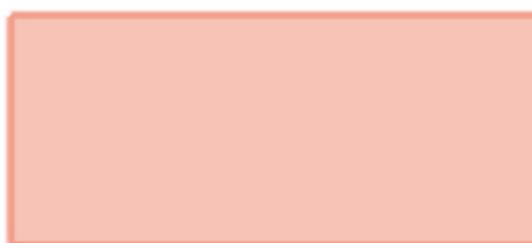
函数式错误处理

Rust 还为 Option 和 Result 提供了大量的辅助函数，如 map / map_err / and_then，你可以很方便地处理数据结构中部分情况。如下图所示：

Ok(T)/Some



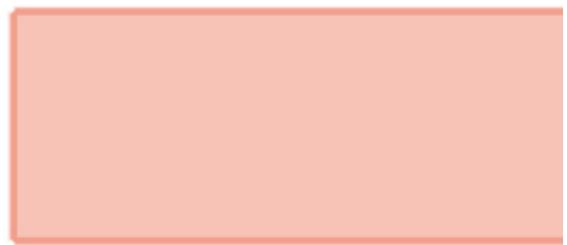
Err(E)/None



Ok(T)



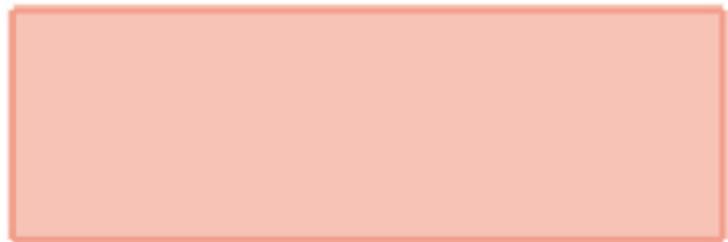
Err(E)



Ok(T)/Some(T)



Err(E)/None



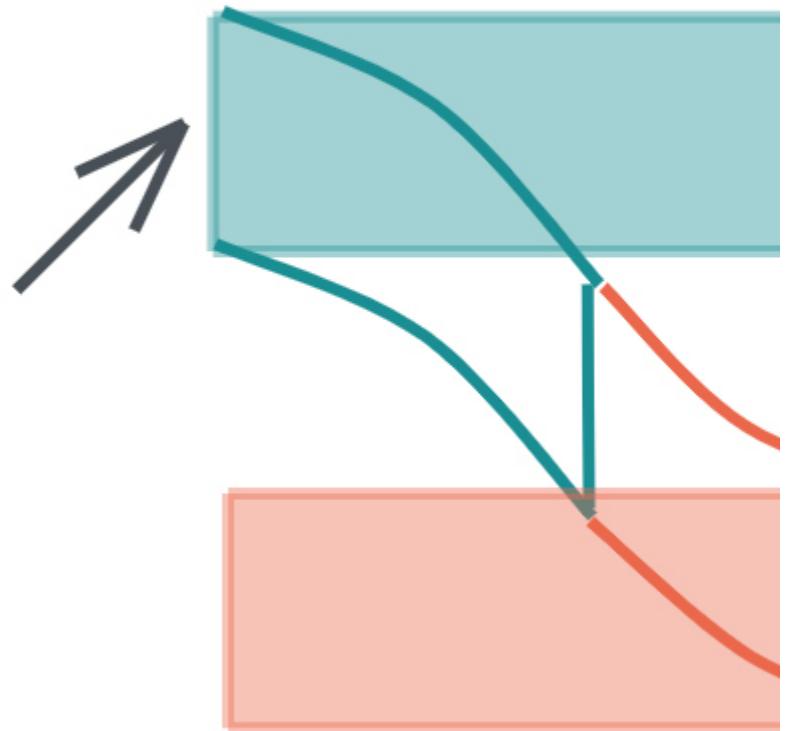
通过这些函数，你可以很方便地对错误处理引入 [Railroad oriented programming](#) 范式。比如用户注册的流程，你需要校验用户输入，对数据进行处理，转换，然后存入数据库中。你可以这么撰写这个流程：

```
Ok(data)
.and_then(validate)
.and_then(process)
.map(transform)
.and_then(store)
.map_error(...)
```

执行流程如下图所示：

validate()

输入
Ok(data)



此外，Option 和 Result的互相转换也很方便，这也得益于 Rust 构建的强大的函数式编程的能力。

我们可以看到，无论是通过 ? 操作符，还是函数式编程进行错误处理，Rust 都力求让错误处理灵活高效，让开发者使用起来简单直观。

panic! 和 catch_unwind

使用 Option 和 Result 是 Rust 中处理错误的首选，绝大多数时候我们也应该使用，但 Rust 也提供了特殊的异常处理能力。

在 Rust 看来，一旦你需要抛出异常，那抛出的一定是严重的错误。所以，Rust 跟 Golang 一样，使用了诸如 panic! 这样的字眼警示开发者：想清楚了再使用我。在使用 Option 和 Result 类型时，开发者也可以对其 unwrap() 或者 expect()，强制把 Option 和 Result<T, E> 转换成 T，如果无法完成这种转换，也会 panic! 出来。

一般而言，panic! 是不可恢复或者不想恢复的错误，我们希望在此刻，程序终止运行并得到崩溃信息。比如下面的代码，它解析 [noise protocol](#) 的协议变量：

```
let params: NoiseParams = "Noise_XX_25519_AESGCM_SHA256".parse().unwrap();
```

如果开发者不小心把协议变量写错了，最佳的方式是立刻 panic! 出来，让错误立刻暴露，以便解决这个问题。

有些场景下，我们也希望能够像异常处理那样能够栈回溯，把环境恢复到捕获异常的上下文。Rust 标准库下提供了 catch_unwind()，把调用栈回溯到 catch_unwind 这一刻，作用和其它语言的 try {...} catch {...} 一样。见如下[代码](#)：

```
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| {
        println!("hello!");
    });
    assert!(result.is_ok());
    let result = panic::catch_unwind(|| {
        panic!("oh no!");
    });
    assert!(result.is_err());
    println!("panic captured: {:?}", result);
}
```

当然，和异常处理一样，并不意味着你可以滥用这一特性，我想，这也是 Rust 把抛出异常称作 panic!，而捕获异常称作 catch_unwind 的原因，让初学者望而生畏，不敢轻易使用。这也是一个不错的用户体验。

catch_unwind 在某些场景下非常有用，比如你在使用 Rust 为 erlang VM 撰写 NIF，你不希望 Rust 代码中的任何 panic! 导致 erlang VM 崩溃。因为崩溃是一个非常不好的体验，它违背了 erlang 的设计原则：process 可以 let it crash，但错误代码不该导致 VM 崩溃。

此刻，你就可以把 Rust 代码整个封装在 catch_unwind() 函数所需要传入的闭包中。这样，一旦任何代码中，包括第三方 crates 的代码，含有能够导致 panic! 的代码，都会被捕获，并被转换为一个 Result。

Error trait 和错误类型的转换

上文中，我们讲到 Result<T, E> 里 E 是一个代表错误的数据类型。为了规范这个代表错误的数据类型的行为，Rust 定义了 Error trait：

```
pub trait Error: Debug + Display {  
    fn source(&self) -> Option<&(dyn Error + 'static)> { ... }  
    fn backtrace(&self) -> Option<&Backtrace> { ... }  
    fn description(&self) -> &str { ... }  
    fn cause(&self) -> Option<&dyn Error> { ... }  
}
```

我们可以定义我们自己的数据类型，然后为其实现 Error trait。

不过，这样的工作已经有人替我们简化了：我们可以使用 thiserror 和 anyhow 来简化这个步骤。thiserror 提供了一个派生宏（derive macro）来简化错误类型的定义，比如：

```
use thiserror::Error;  
#[derive(Error, Debug)]  
#[non_exhaustive]  
pub enum DataStoreError {  
    #[error("data store disconnected")]  
    Disconnect(#[from] io::Error),  
    #[error("the data for key '{0}' is not available")]  
    Redaction(String),  
    #[error("invalid header (expected {expected:?:}, found {found:?:})")]  
    InvalidHeader {  
        expected: String,  
        found: String,  
    },  
    #[error("unknown data store error")]  
    Unknown,  
}
```

如果你在撰写一个 Rust 库，那么 thiserror 可以很好地协助你对这个库里所有可能发生的错误进行建模。

而 anyhow 实现了 anyhow::Error 和任意符合 Error trait 的错误类型之间的转换，让你可以使用 ? 操作符，不必再手工转换错误类型。anyhow 还可以让你很容易地抛出一些临时的错误，而不必费力定义错误类型，当然，我们不提倡滥用这个能力。

作为一名严肃的开发者，我非常建议你在开发前，先用类似 thiserror 的库定义好你项目中主要的错误类型，并随着项目的深入，不断增加新的错误类型，让系统中所有的潜在错误都无所遁形。

小结

这一讲我们讨论了错误处理的三种方式：使用返回值、异常处理和类型系统。而Rust 站在巨人的肩膀上，采各家之长，形成了我们目前看到的方案：**主要用类型系统来处理错误，辅以异常来应对不可恢复的错误。**

- 相比 C/Golang 直接用返回值的错误处理方式，Rust 在类型上更完备，构建了逻辑更为严谨的 Option 类型和 Result 类型，既避免了错误被不慎忽略，也避免了用啰嗦的表达方式传递错误；
- 相对于 C++ / Java 使用异常的方式，Rust 区分了可恢复错误和不可恢复错误，分别使用 Option / Result，以及 panic! / catch_unwind 来应对，更安全高效，避免了异常安全带来的诸多问题；
- 而对比它的老师 Haskell，Rust 的错误处理更加实用简洁，这得益于它强大的元编程功能，使用 ? 操作符来简化错误的传递。

总结一下：Rust 的错误处理很实用、足够强大、处理起来又不会过于冗长，充分使用 Rust 语言本身的能力，大大简化了错误传递的代码，简洁明了，几乎接近于异常处理的方式。

当然，Rust 错误处理还有很多提升空间，尤其标准库没有给出足够的工具，导致社区里有大量的互不兼容的辅助库。不过这些都瑕不掩瑜，对 Rust 语言来说，错误处理还处于一个不断进化的阶段，相信未来标准库会给出更好更方便的答案。

思考题

如果你要开发一个类似Redis 的缓存服务器，你都会定义哪些错误？为什么？

欢迎在留言区分享你的思考。你已经打卡Rust学习第18次啦，如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。

拓展阅读

1. [Exception handling considered harmful](#)

2.[Exception safety](#)

3.[Why does go not have exceptions](#)

4.对 Railroad oriented programming 范式感兴趣的同學可以看看这个 [slides](#)

5.[Noise protocol](#)

6.[Erlang NIF](#)

7.[thiserror](#)

8.[anyhow](#)

02 进阶篇

02 进阶篇

高级特性

[链接](#)

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。

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，那自然不会存在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。。。也没啥好举例的，你能想到的几乎类型都实现了 `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

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，从枚举到整数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。

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。

1 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：

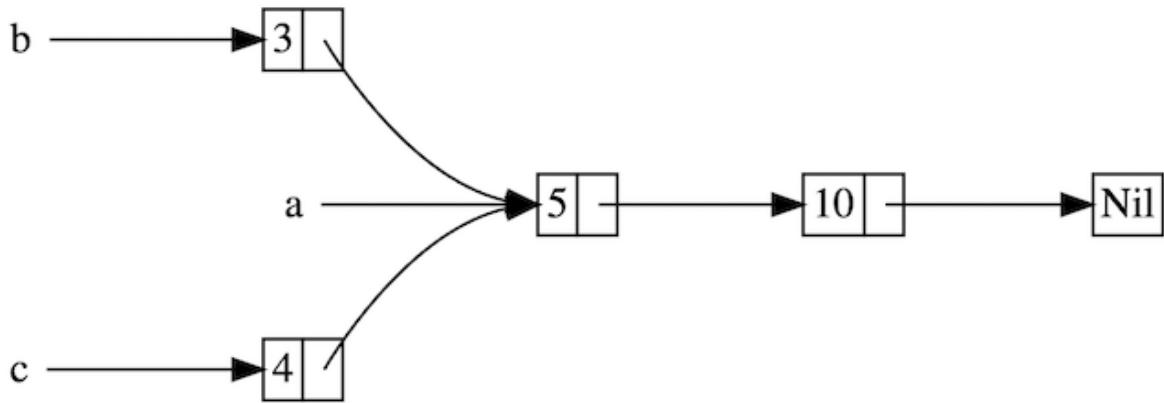
```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[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::new(Nil))));

    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：

- 在创建了 `a` 后，紧接着就使用 `a` 创建了 `b`，因此 `b` 引用了 `a`
- 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：

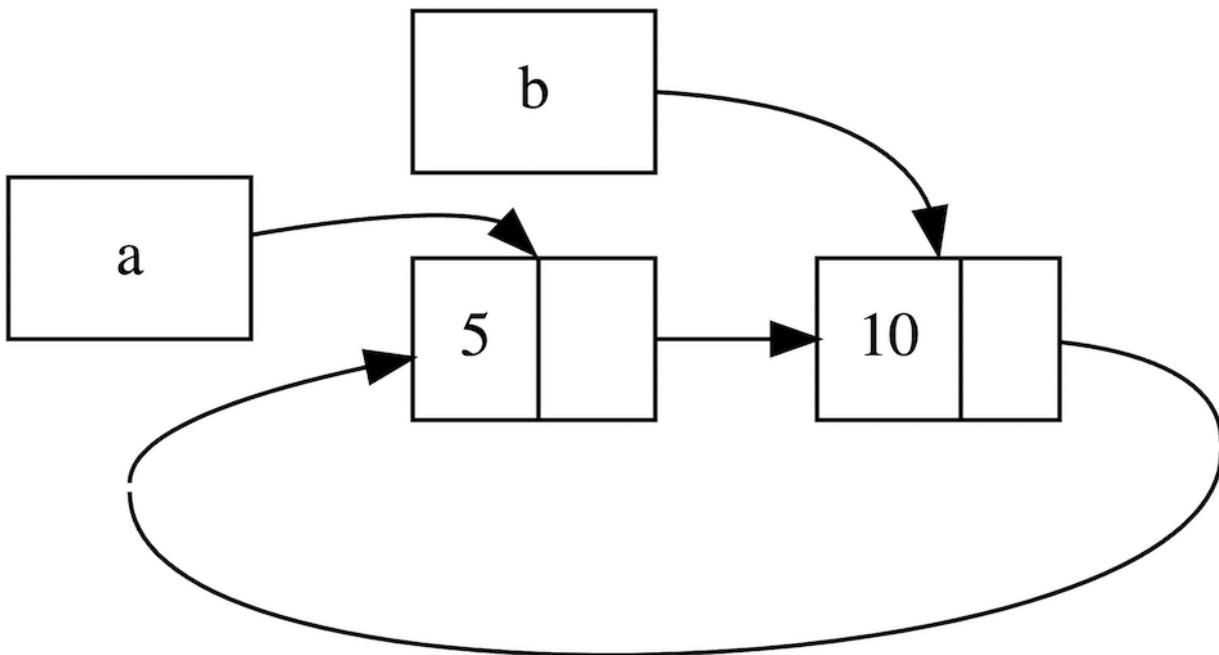
```

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
通过 <code>upgrade</code> 取到 <code>Option<Rc<T>></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<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。

2 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。

模式和匹配

闭包：FnOnce、FnMut 和 Fn，为什么有这么多类型？

你好，我是陈天。

在现代编程语言中，闭包是一个很重要的工具，可以让我们很方便地以函数式编程的方式来撰写代码。因为闭包可以作为参数传递给函数，可以作为返回值被函数返回，也可以为它实现某个 trait，使其能表现出其他行为，而不仅仅是作为函数被调用。

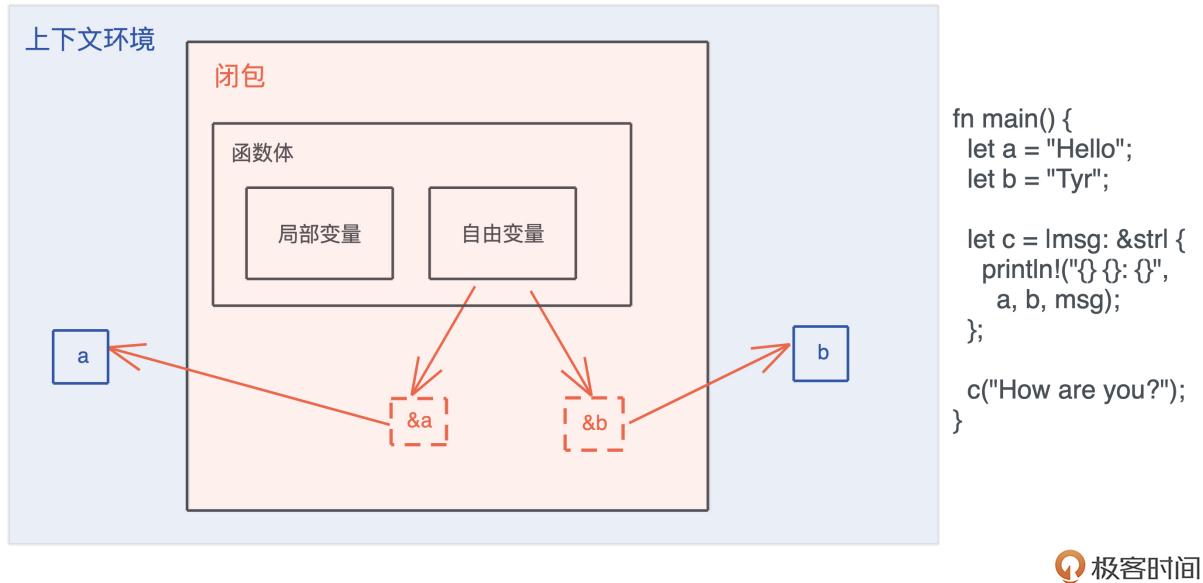
这些都是怎么做到的？这就和 Rust 里闭包的本质有关了，我们今天就来学习基础篇的最后一个知识点：闭包。

闭包的定义

之前介绍了闭包的基本概念和一个非常简单的例子：

闭包是将函数，或者说代码和其环境一起存储的一种数据结构。闭包引用的上下文中的自由变量，会被捕获到闭包的结构中，成为闭包类型的一部分（[第二讲](#)）。

闭包会根据内部的使用情况，捕获环境中的自由变量。在 Rust 里，闭包可以用 `|args| {code}` 来表述，图中闭包 c 捕获了上下文中的 a 和 b，并通过引用来使用这两个自由变量：



除了用引用来捕获自由变量之外，还有另外一个方法使用 move 关键字 `move |args| {code}`。

之前的课程中，多次见到了创建新线程的 `thread::spawn`，它的参数就是一个闭包：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
```

```
F: FnOnce() -> T,  
F: Send + 'static,  
T: Send + 'static,
```

仔细看这个接口：

1. F: FnOnce() → T, 表明 F 是一个接受 0 个参数、返回 T 的闭包。FnOnce 我们稍后再说。
2. F: Send + 'static, 说明闭包 F 这个数据结构，需要静态生命周期或者拥有所有权，并且它还能被发送给另一个线程。
3. T: Send + 'static, 说明闭包 F 返回的数据结构 T，需要静态生命周期或者拥有所有权，并且它还能被发送给另一个线程。

1 和 3 都很好理解，2 就有些费解了。一个闭包，它不就是一段代码 + 被捕获的变量么？需要静态生命周期或者拥有所有权是什么意思？

拆开看。代码自然是静态生命周期了，那么是不是意味着被捕获的变量，需要静态生命周期或者拥有所有权？

的确如此。在使用 `thread::spawn` 时，我们需要使用 `move` 关键字，把变量的所有权从当前作用域移动到闭包的作用域，让 `thread::spawn` 可以正常编译通过：

```
use std::thread;  
  
fn main() {  
    let s = String::from("hello world");  
  
    let handle = thread::spawn(move || {  
        println!("moved: {:?}", s);  
    });  
  
    handle.join().unwrap();  
}
```

但你有没有好奇过，加 `move` 和不加 `move`，这两种闭包有什么本质上的不同？闭包究竟是一种什么样的数据类型，让编译器可以判断它是否满足 `Send + 'static` 呢？我们从闭包的本质下手来尝试回答这两个问题。

闭包本质上是什么？

在官方的 Rust reference 中，有这样的[定义](#)：

A closure expression produces a closure value with a unique, anonymous type that cannot be written out. A closure type is approximately equivalent to a struct which contains the captured variables.

闭包是一种匿名类型，一旦声明，就会产生一个新的类型，但这个类型无法被其它地方使用。这个类型就像一个结构体，会包含所有捕获的变量。

所以闭包类似一个特殊的结构体？

为了搞明白这一点，我们得写段代码探索一下，建议你跟着敲一遍认真思考（[代码](#)）：

```
use std::collections::HashMap, mem::size_of_val;
fn main() {
    // 长度为 0
    let c1 = || println!("hello world!");
    // 和参数无关，长度也为 0
    let c2 = |i: i32| println!("hello: {}", i);
    let name = String::from("tyr");
    let name1 = name.clone();
    let mut table = HashMap::new();
    table.insert("hello", "world");
    // 如果捕获一个引用，长度为 8
    let c3 = || println!("hello: {}", name);
    // 捕获移动的数据 name1(长度 24) + table(长度 48)，closure 长度 72
    let c4 = move || println!("hello: {}, {:?}", name1, table);
    let name2 = name.clone();
    // 和局部变量无关，捕获了一个 String name2，closure 长度 24
    let c5 = move || {
        let x = 1;
        let name3 = String::from("lindsey");
        println!("hello: {}, {:?}, {:?}", x, name2, name3);
    };
    println!(
        "c1: {}, c2: {}, c3: {}, c4: {}, c5: {}, main: {}",
        size_of_val(&c1),
        size_of_val(&c2),
        size_of_val(&c3),
        size_of_val(&c4),
        size_of_val(&c5),
        size_of_val(&main),
    )
}
```

分别生成了 5 个闭包：

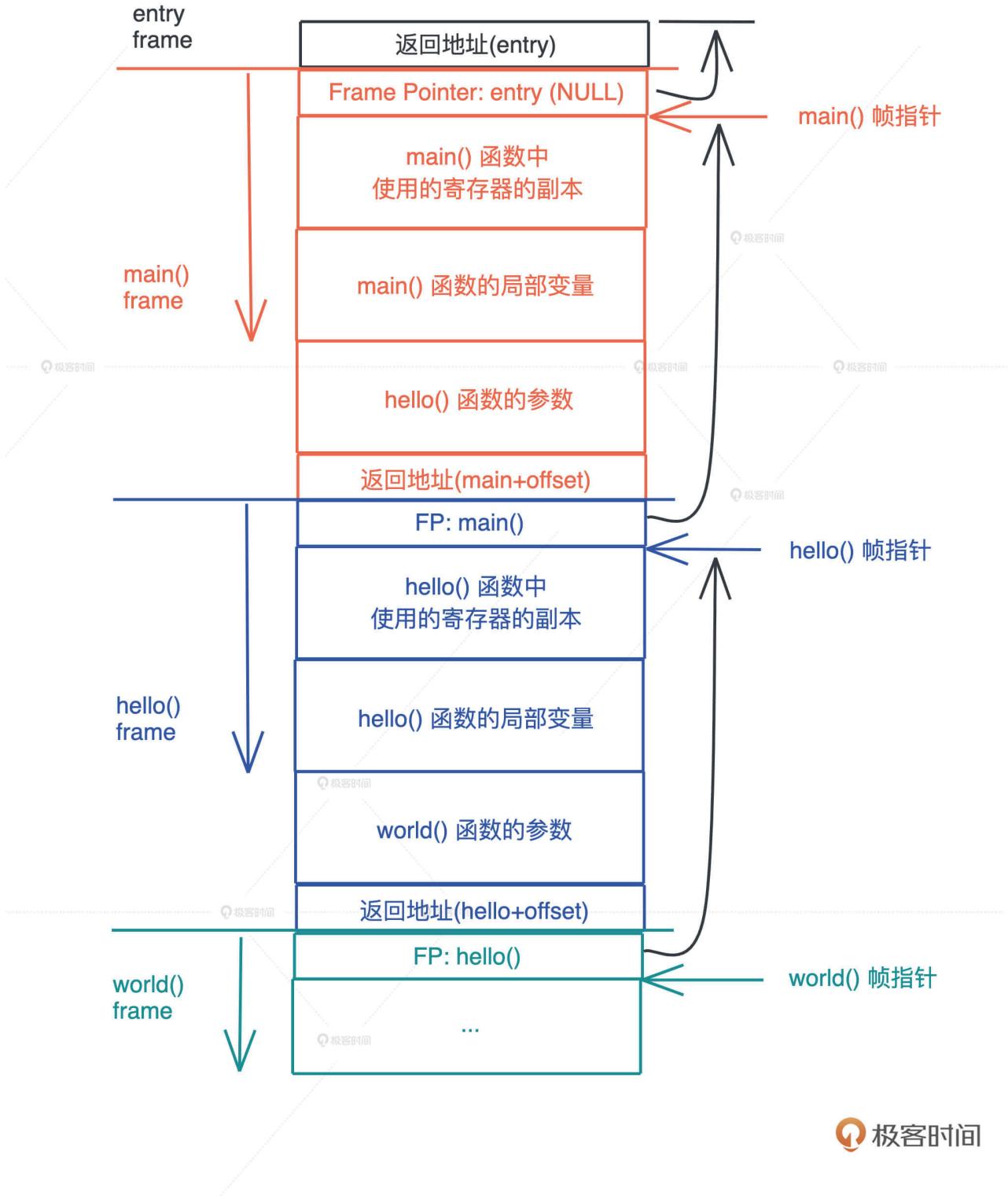
- c1 没有参数，也没捕获任何变量，从代码输出可以看到，c1 长度为 0；
- c2 有一个 i32 作为参数，没有捕获任何变量，长度也为 0，可以看出参数跟闭包的大小无关；

- c3 捕获了一个对变量 name 的引用，这个引用是 &String，长度为 8。而 c3 的长度也是 8；
- c4 捕获了变量 name1 和 table，由于用了 move，它们的所有权移动到了 c4 中。c4 长度是 72，恰好等于 String 的 24 字节，加上 HashMap 的 48 字节。
- c5 捕获了 name2，name2 的所有权移动到了 c5，虽然 c5 有局部变量，但它的大小和局部变量也无关，c5 的大小等于 String 的 24 字节。

学到这里，前面的第一个问题就解决了，可以看到，不带 move 时，闭包捕获的是对应自由变量的引用；带 move 时，对应自由变量的所有权会被移动到闭包结构中。

继续分析这段代码的运行结果。

还知道了，**闭包的大小跟参数、局部变量都无关，只跟捕获的变量有关**。如果你回顾[第一讲](#)函数调用，参数和局部变量在栈中如何存放的图，就很清楚了：因为它们是在调用的时刻才在栈上产生的内存分配，说到底和闭包类型本身是无关的，所以闭包的大小跟它们自然无关。



那一个闭包类型在内存中究竟是如何排布的，和结构体有什么区别？我们要再次结合 rust-gdb 探索，看看上面的代码在运行结束前，几个长度不为 0 闭包内存里都放了什么：

```
(gdb) info locals
c5 = closure_size::main::closure-4 ("tyr")
name2 = "tyr"
c4 = closure_size::main::closure-3 ("tyr", HashMap(size=1) = {"hello" = "world"})
c3 = closure_size::main::closure-2 (0x7fffffff998)
table = HashMap(size=1) = {"hello" = "world"}
name1 = "tyr"
name = "tyr"
c2 = closure_size::main::closure-1
c1 = closure_size::main::closure-0
```

局部变量

```
(gdb) x/gx &c3
0x7fffffff9f8: 0x00007fffffff998
(gdb) x/3gx 0x00007fffffff998
0x7fffffff998: 0x00005555555a9ba0 0x00000000000000000003
0x7fffffff9a8: 0x000000000000000003
(gdb) x/3c 0x00005555555a9ba0
0x5555555a9ba0: 116 't' 121 'y' 114 'r'
```

c3 捕获了一个字符串引用
长度为 8
字符串本身长度为 3, 内容 "tyr"

```
(gdb) x/9gx &c4
0x7fffffffda0: 0x00005555555a9bc0 0x000000000000000003
0x7fffffffda10: 0x000000000000000003 0xb421d4064b548fd0
0x7fffffffda20: 0xca71308aa8535935 0x000000000000000003
0x7fffffffda30: 0x00005555555a9ac0 0x000000000000000002
0x7fffffffda40: 0x000000000000000001
(gdb) x/3c 0x00005555555a9bc0
0x5555555a9bc0: 116 't' 121 'y' 114 'r'
```

c4 捕获了一个移动过来的
字符串 (长24) 和哈希表 (长48)
所以 c4 长度 72
字符串也是 "tyr"

```
(gdb) x/3gx &c5
0x7fffffffda60: 0x00005555555a9c10 0x000000000000000003
0x7fffffffda70: 0x000000000000000003
(gdb) x/3c 0x00005555555a9c10
0x5555555a9c10: 116 't' 121 'y' 114 'r'
```

c5 有局部变量, 也捕获了
一个移动过来的字符串,
所以 c5 长度 24
局部变量和 closure 长度无关

可以看到, c3 的确是一个引用, 把它指向的内存地址的 24 个字节打出来, 是 (ptr | cap | len) 的标准结构。如果打印 ptr 对应的堆内存的 3 个字节, 是 't' 'y' 'r'。

而 c4 捕获的 name 和 table, 内存结构和下面的结构体一模一样:

```
struct Closure4 {
    name: String, // (ptr|cap|len)=24字节
    table: HashMap<&str, &str> // (RandomState(16)|mask|ctrl||left||len)=48字节
}
```

不过, 对于 closure 类型来说, 编译器知道像函数一样调用闭包 c4() 是合法的, 并且知道执行 c4() 时, 代码应该跳转到什么地址来执行。在执行过程中, 如果遇到 name、table, 可以从自己的数据结构中获取。

那么多想一步, 闭包捕获变量的顺序, 和其内存结构的顺序是一致的么? 的确如此, 如果我们调整闭包里使用 name1 和 table 的顺序:

```
let c4 = move || println!("hello: {:?}", {}, table, name1);
```

其数据的位置是相反的，类似于：

```
struct Closure4 {  
    table: HashMap<&str, &str> // (RandomState(16)|mask|ctrl|left|len)=48字节  
    name: String, // (ptr|cap|len)=24字节  
}
```

从 gdb 中也可以看到同样的结果：

(gdb) info locals

c4 = closure_size::main:

c3 = closure_size::main:

table = HashMap(size=1

name1 = "tyr"

name = "tyr"

c2 = closure_size::main:

c1 = closure_size::main:

(gdb) x/9gx &c4

0x7fffffffda60: 0x105fdc0

0x7fffffffda70: 0x00000000

0x7fffffffda80: 0x00000000

0x7fffffd90: 0x00005555
0x7fffffd90: 0x00000000

不过这只是逻辑上的位置，如果你还记得[第 11 讲](#) struct 在内存的排布，Rust 编译器会重排内存，让数据能够以最小的代价对齐，所以有些情况下，内存中数据的顺序可能和 struct 定义不一致。

所以回到刚才闭包和结构体的比较。在 Rust 里，闭包产生的匿名数据类型，格式和 struct 是一样的。看图中 gdb 的输出，**闭包是存储在栈上，并且除了捕获的数据外，闭包本身不包含任何额外函数指针指向闭包的代码**。如果你理解了 c3 / c4 这两个闭包，c5 是如何构造的就很好理解了。

现在，你是不是可以回答为什么 `thread::spawn` 对传入的闭包约束是 `Send + 'static` 了？究竟什么样的闭包满足它呢？很明显，使用了 `move` 且 `move` 到闭包内的数据结构满足 `Send`，因为此时，闭包的数据结构拥有所有数据的所有权，它的生命周期是 `'static`。

看完Rust闭包的内存结构，你是不是想说“就这”，没啥独特之处吧？但是对比其他语言，结合接下来我的解释，你再仔细想想就会有一种“这怎么可能”的惊讶。

不同语言的闭包设计

闭包最大的问题是变量的多重引用导致生命周期不明确，所以你先想，其它支持闭包的语言（lambda 也是闭包），它们的闭包会放在哪里？

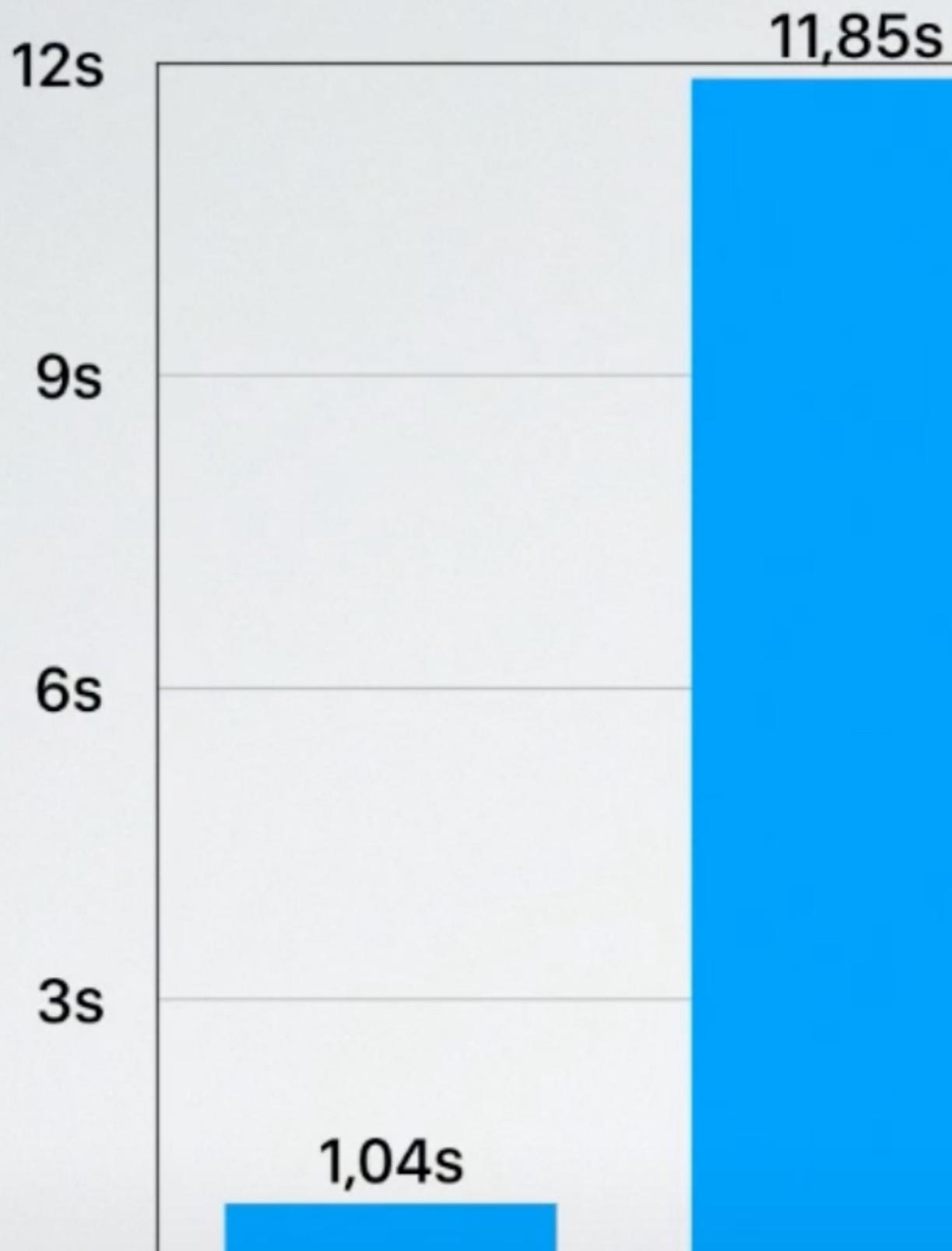
栈上么？是，又好像不是。

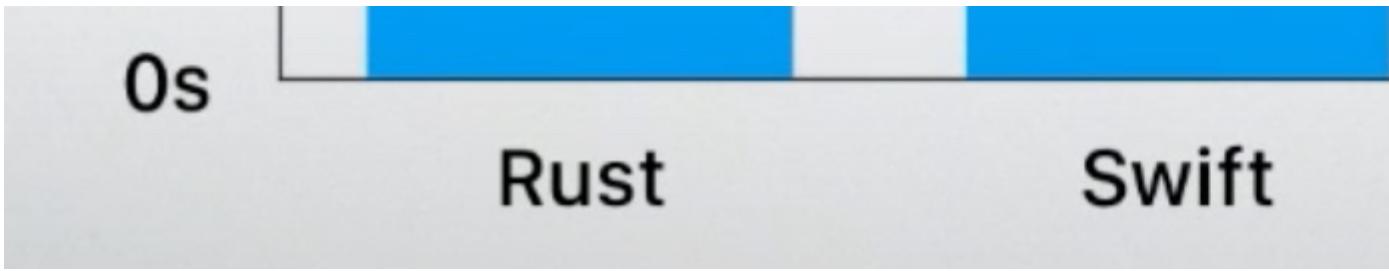
因为闭包这玩意，从当前上下文中捕获了些变量，变得有点不伦不类，不像函数那样清楚，尤其是这些被捕获的变量，它们的归属和生命周期处理起来很麻烦。所以，大部分编程语言的闭包很多时候无法放在栈上，需要额外的堆分配。你可以看这个 [Golang 的例子](#)。

不光 Golang，Java / Swift / Python / JavaScript 等语言都是如此，这也是为什么大多数编程语言闭包的性能要远低于函数调用。因为使用闭包就意味着：额外的堆内存分配、潜在的动态分派（很多语言会把闭包处理成函数指针）、额外的内存回收。

在性能上，唯有 C++ 的 lambda 和 Rust 闭包类似，不过 C++ 的闭包还有一些场景会触发堆内存分配。如果你还记得 16 讲的 Rust / Swift / Kotlin iterator 函数式编程的性能测试：

Run Chun





Kotlin 运行超时，Swift 很慢，Rust 的性能却和使用命令式编程的 C 几乎一样，除了编译器优化的效果，也因为 Rust 闭包的性能和函数差不多。

为什么 Rust 可以做到这样呢？这又跟 Rust 从根本上使用所有权和借用，解决了内存归属问题有关。

在其他语言中，闭包变量因为多重引用导致生命周期不明确，但 Rust 从一开始就消灭了这个问题：

- 如果不使用 move 转移所有权，闭包会引用上下文中的变量，**这个引用受借用规则的约束**，所以只要编译通过，那么闭包对变量的引用就不会超过变量的生命周期，没有内存安全问题。
- 如果使用 move 转移所有权，上下文中的变量在转移后就无法访问，**闭包完全接管这些变量**，它们的生命周期和闭包一致，所以也不会有内存安全问题。

而 Rust 为每个闭包生成一个新的类型，又使得调用闭包时可以直接和代码对应，省去了使用函数指针再转一道手的额外消耗。

所以还是那句话，当回归到最初的本原，你解决的不是单个问题，而是由此引发的所有问题。我们不必为堆内存管理设计 GC、不必为其它资源的回收提供 defer 关键字、不必为并发安全进行诸多限制、也不必为闭包挖空心思搞优化。

Rust的闭包类型

现在我们搞明白了闭包究竟是个什么东西，在内存中怎么表示，接下来我们看看 FnOnce / FnMut / Fn 这三种闭包类型有什么区别。

在声明闭包的时候，我们并不需要指定闭包要满足的约束，但是当闭包作为函数的参数或者数据结构的一个域时，我们需要告诉调用者，对闭包的约束。还以 thread::spawn 为例，它要求传入的闭包满足 FnOnce trait。

FnOnce

先来看 FnOnce。它的[定义](#)如下：

```
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

FnOnce 有一个关联类型 Output，显然，它是闭包返回值的类型；还有一个方法 call_once，要注意的是 call_once 第一个参数是 self，它会转移 self 的所有权到 call_once 函数中。

这也是为什么 FnOnce 被称作 Once：它只能被调用一次。再次调用，编译器就会报变量已经被 move 这样的常见所有权错误了。

至于 FnOnce 的参数，是一个叫 Args 的泛型参数，它并没有任何约束。如果你对这个感兴趣可以看文末的参考资料。

看一个隐式的 FnOnce 的例子：

```
fn main() {
    let name = String::from("Tyr");
    // 这个闭包啥也不干，只是把捕获的参数返回去
    let c = move |greeting: String| (greeting, name);

    let result = c("hello".to_string());

    println!("result: {:?}", result);

    // 无法再次调用
    let result = c("hi".to_string());
}
```

这个闭包 c，啥也没做，只是把捕获的参数返回。就像一个结构体里，某个字段被转移走之后，就不能再访问一样，闭包内部的数据一旦被转移，这个闭包就不完整了，也就无法再次使用，所以它是一个 FnOnce 的闭包。

如果一个闭包并不转移自己的内部数据，那么它就不是 FnOnce，然而，一旦它被当做 FnOnce 调用，自己会被转移到 call_once 函数的作用域中，之后就无法再次调用了，我们看个例子（[代码](#)）：

```
fn main() {
    let name = String::from("Tyr");

    // 这个闭包会 clone 内部的数据返回，所以它不是 FnOnce
    let c = move |greeting: String| (greeting, name.clone());

    // 所以 c1 可以被调用多次
}
```

```

println!("c1 call once: {:?}", c("qiao".into()));
println!("c1 call twice: {:?}", c("bonjour".into()));

// 然而一旦它被当成 FnOnce 被调用，就无法被再次调用
println!("result: {:?}", call_once("hi".into(), c));

// 无法再次调用
// let result = c("hi".to_string());

// fn 也可以被当成 fn 调用，只要接口一致就可以
println!("result: {:?}", call_once("hola".into(), not_closure));
}

fn call_once(arg: String, c: impl FnOnce(String) -> (String, String)) -> (String, String) {
    c(arg)
}

fn not_closure(arg: String) -> (String, String) {
    (arg, "Rosie".into())
}

```

FnMut

理解了 FnOnce，我们再来看 FnMut，它的[定义](#)如下：

```

pub trait FnMut<Args>: FnOnce<Args> {
    extern "rust-call" fn call_mut(
        &mut self,
        args: Args
    ) -> Self::Output;
}

```

首先，FnMut “继承”了 FnOnce，或者说 FnOnce 是 FnMut 的 super trait。所以 FnMut 也拥有 Output 这个关联类型和 call_once 这个方法。此外，它还有一个 call_mut() 方法。**注意 call_mut() 传入 &mut self，它不移动 self，所以 FnMut 可以被多次调用。**

因为 FnOnce 是 FnMut 的 super trait，所以，一个 FnMut 闭包，可以被传给一个需要 FnOnce 的上下文，此时调用闭包相当于调用了 call_once()。

如果你理解了前面讲的闭包的内存组织结构，那么 FnMut 就不难理解，就像结构体如果想改变数据需要用 let mut 声明一样，如果你想改变闭包捕获的数据结构，那么就需要 FnMut。我们看个例子 ([代码](#))：

```

fn main() {
    let mut name = String::from("hello");
    let mut name1 = String::from("hola");
}

```

```

// 捕获 &mut name
let mut c = || {
    name.push_str(" Tyr");
    println!("c: {}", name);
};

// 捕获 mut name1, 注意 name1 需要声明成 mut
let mut c1 = move || {
    name1.push_str("!");
    println!("c1: {}", name1);
};

c();
c1();

call_mut(&mut c);
call_mut(&mut c1);

call_once(c);
call_once(c1);
}

// 在作为参数时, FnMut 也要显式地使用 mut, 或者 &mut
fn call_mut(c: &mut impl FnMut()) {
    c();
}

// 想想看, 为啥 call_once 不需要 mut?
fn call_once(c: impl FnOnce()) {
    c();
}

```

在声明的闭包 c 和 c1 里, 我们修改了捕获的 name 和 name1。不同的是 name 使用了引用, 而 name1 移动了所有权, 这两种情况和其它代码一样, 也需要遵循所有权和借用有关的规则。所以, 如果在闭包 c 里借用了 name, 你就不能把 name 移动给另一个闭包 c1。

这里也展示了, c 和 c1 这两个符合 FnMut 的闭包, 能作为 FnOnce 来调用。我们在代码中也确认了, FnMut 可以被多次调用, 这是因为 call_mut() 使用的是 &mut self, 不移动所有权。

Fn

最后我们来看看 Fn trait。它的[定义](#)如下:

```

pub trait Fn<Args>: FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

```

可以看到，它“继承”了 FnMut，或者说 FnMut 是 Fn 的 super trait。这也就意味着任何需要 FnOnce 或者 FnMut 的场合，都可以传入满足 Fn 的闭包。我们继续看例子（[代码](#)）：

```
fn main() {
    let v = vec![0u8; 1024];
    let v1 = vec![0u8; 1023];

    // Fn, 不移动所有权
    let mut c = |x: u64| v.len() as u64 * x;
    // Fn, 移动所有权
    let mut c1 = move |x: u64| v1.len() as u64 * x;

    println!("direct call: {}", c(2));
    println!("direct call: {}", c1(2));

    println!("call: {}", call(3, &c));
    println!("call: {}", call(3, &c1));

    println!("call_mut: {}", call_mut(4, &mut c));
    println!("call_mut: {}", call_mut(4, &mut c1));

    println!("call_once: {}", call_once(5, c));
    println!("call_once: {}", call_once(5, c1));
}

fn call(arg: u64, c: &impl Fn(u64) -> u64) -> u64 {
    c(arg)
}

fn call_mut(arg: u64, c: &mut impl FnMut(u64) -> u64) -> u64 {
    c(arg)
}

fn call_once(arg: u64, c: impl FnOnce(u64) -> u64) -> u64 {
    c(arg)
}
```

闭包的使用场景

在讲完Rust的三个闭包类型之后，最后来看看闭包的使用场景。虽然今天才开始讲闭包，但其实之前隐晦地使用了很多闭包。thread::spawn 自不必说，我们熟悉的 Iterator trait 里面大部分函数都接受一个闭包，比如 [map](#)：

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    Self: Sized,
    F: FnMut(Self::Item) -> B,
{
    Map::new(self, f)
}
```

可以看到，Iterator 的 map() 方法接受一个 FnMut，它的参数是 Self::Item，返回值是没有约束的泛型参数 B。Self::Item 是 Iterator::next() 方法吐出来的数据，被 map 之后，可以得到另一个结果。

所以在函数的参数中使用闭包，是闭包一种非常典型的用法。另外闭包也可以作为函数的返回值，举个简单的例子（[代码](#)）：

```
use std::ops::Mul;

fn main() {
    let c1 = curry(5);
    println!("5 multiply 2 is: {}", c1(2));

    let adder2 = curry(3.14);
    println!("pi multiply 4^2 is: {}", adder2(4. * 4.));
}

fn curry<T>(x: T) -> impl Fn(T) -> T
where
    T: Mul<Output = T> + Copy,
{
    move |y| x * y
}
```

最后，闭包还有一种并不少见，但可能不太容易理解的用法：**为它实现某个 trait**，使其也能表现出其他行为，而不仅仅是作为函数被调用。比如说有些接口既可以传入一个结构体，又可以传入一个函数或者闭包。

我们看一个 [tonic](#) (Rust 下的 gRPC 库) 的[例子](#)：

```
pub trait Interceptor {
    /// Intercept a request before it is sent, optionally cancelling it.
    fn call(&mut self, request: crate::Request<()>) -> Result<crate::Request<()>, Status>;
}

impl<F> Interceptor for F
where
    F: FnMut(crate::Request<()>) -> Result<crate::Request<()>, Status>,
{
    fn call(&mut self, request: crate::Request<()>) -> Result<crate::Request<()>, Status> {
        self(request)
    }
}
```

在这个例子里，Interceptor 有一个 call 方法，它可以让 gRPC Request 被发送出去之前被修改，一般是添加各种头，比如 Authorization 头。

我们可以创建一个结构体，为它实现 `Interceptor`，不过大部分时候 `Interceptor` 可以直接通过一个闭包函数完成。为了让传入的闭包也能通过 `Interceptor::call()` 来统一调用，可以为符合某个接口的闭包实现 `Interceptor trait`。掌握了这种用法，我们就可以通过某些 `trait` 把特定的结构体和闭包统一起来调用，是不是很神奇。

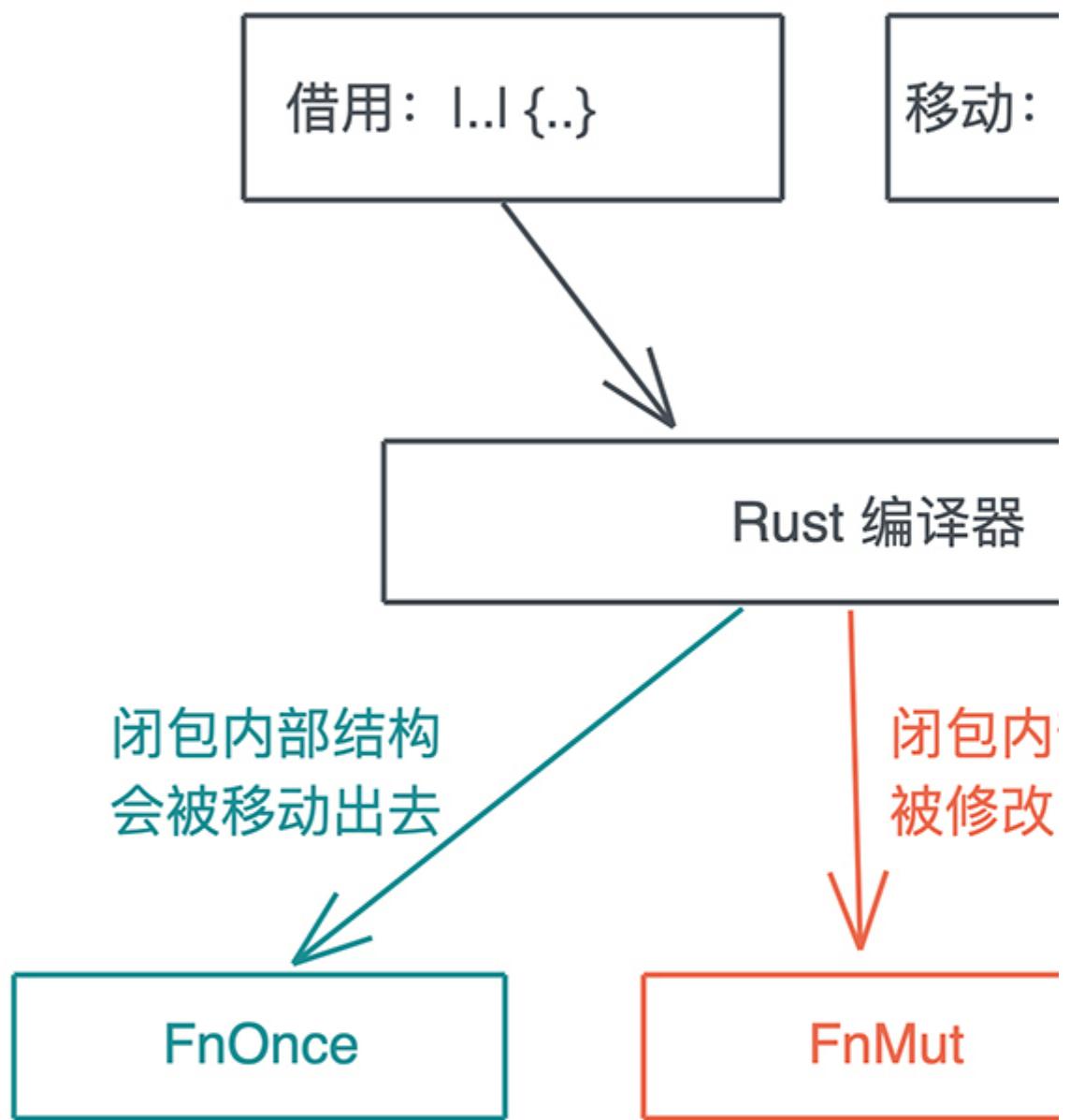
小结

Rust 闭包的效率非常高。首先闭包捕获的变量，都储存在栈上，没有堆内存分配。其次因为闭包在创建时会隐式地创建自己的类型，每个闭包都是一个新的类型。通过闭包自己唯一的类型，Rust 不需要额外的函数指针来运行闭包，所以闭包的调用效率和函数调用几乎一致。

Rust 支持三种不同的闭包 `trait`: `FnOnce`、`FnMut` 和 `Fn`。`FnOnce` 是 `FnMut` 的 super `trait`，而 `FnMut` 又是 `Fn` 的 super `trait`。从这些 `trait` 的接口可以看出，

- `FnOnce` 只能调用一次；
- `FnMut` 允许在执行时修改闭包的内部数据，可以执行多次；
- `Fn` 不允许修改闭包的内部数据，也可以执行多次。

总结一下三种闭包使用的情况以及它们之间的关系：



思考题

1. 下面的代码，闭包 c 相当于一个什么样的结构体？它的长度多大？代码的最后，main() 函数还能访问变量 name 吗？为什么？

```
fn main() {
    let name = String::from("Tyr");
    let vec = vec!["Rust", "Elixir", "Javascript"];
    let v = &vec[..];
    let data = (1, 2, 3, 4);
    let c = move || {
        println!("data: {:?}", data);
        println!("v: {:?}, name: {:?}", v, name.clone());
    };
    c();
    // 请问在这里，还能访问 name 吗？为什么？
}
```

2. 在讲到 FnMut 时，我们放了一段代码，在那段代码里，我问了一个问题：为啥 call_once 不需要 c 是 mut 呢？就像下面这样：

```
// 想想看，为啥 call_once 不需要 mut?
fn call_once(mut c: impl FnOnce()) {
    c();
}
```

3. 为下面的代码添加实现，使其能够正常工作（[代码](#)）：

```
pub trait Executor {
    fn execute(&self, cmd: &str) -> Result<String, &'static str>;
}

struct BashExecutor {
    env: String,
}

impl Executor for BashExecutor {
    fn execute(&self, cmd: &str) -> Result<String, &'static str> {
        Ok(format!(
            "fake bash execute: env: {}, cmd: {}",
            self.env, cmd
        ))
    }
}

// 看看我给的 tonic 的例子，想想怎么实现让 27 行可以正常执行

fn main() {
    let env = "PATH=/usr/bin".to_string();

    let cmd = "cat /etc/passwd";
    let r1 = execute(cmd, BashExecutor { env: env.clone() });
    println!("{}:?", r1);

    let r2 = execute(cmd, |cmd: &str| {
        Ok(format!("fake fish execute: env: {}, cmd: {}", env, cmd))
    });
    println!("{}:?", r2);
}

fn execute(cmd: &str, exec: impl Executor) -> Result<String, &'static str> {
    exec.execute(cmd)
}
```

你已经完成Rust学习的第19次打卡。如果你觉得有收获，也欢迎你分享给身边的朋友，邀TA一起讨论。我们下节课见～

参考资料

怎么理解 FnOnce 的 Args 泛型参数呢？Args 又是怎么和 FnOnce 的约束，比如 FnOnce(String) 这样的参数匹配呢？感兴趣的同学可以看下面的例子，它（不完全）模拟了 FnOnce 中闭包的使用（[代码](#)）：

```
struct ClosureOnce<Captured, Args, Output> {
    // 捕获的数据
    captured: Captured,
    // closure 的执行代码
    func: fn(Args, Captured) -> Output,
```

```

}

impl<Captured, Args, Output> ClosureOnce<Captured, Args, Output> {
    // 模拟 FnOnce 的 call_once, 直接消耗 self
    fn call_once(self, greeting: Args) -> Output {
        (self.func)(greeting, self.captured)
    }
}

// 类似 greeting 闭包的函数体
fn greeting_code1(args: (String,), captured: (String,)) -> (String, String) {
    (args.0, captured.0)
}

fn greeting_code2(args: (String, String), captured: (String, u8)) -> (String, String, String, u8) {
    (args.0, args.1, captured.0, captured.1)
}

fn main() {
    let name = "Tyr".into();
    // 模拟变量捕捉
    let c = ClosureOnce {
        captured: (name,),
        func: greeting_code1,
    };

    // 模拟闭包调用, 这里和 FnOnce 不完全一样, 传入的是一个 tuple 来匹配 Args 参数
    println!(" {:?}", c.call_once(("hola".into(),)));
    // 调用一次后无法继续调用
    // println!(" {:?}", clo.call_once("hola".into()));

    // 更复杂一些的复杂的闭包
    let c1 = ClosureOnce {
        captured: ("Tyr".into(), 18),
        func: greeting_code2,
    };

    println!(" {:?}", c1.call_once(("hola".into(), "hallo".into())));
}

```

unsafe rust

unsafe rust

高级宏

The Little Book of Rust Macros (Rust 宏小册)

本书是续写版本, 续写的版本由 Veykril 撰稿, 续作对原作有补充和删改。

- 原作: [repo](#) | [渲染版](#)
- 原作中文翻译: [repo](#) | [渲染版](#)

原作及其翻译渲染版本没有使用 mdbook 构建，而是使用 py 来生成 HTML。在发布文档和运行样板代码方面诸多不便。

而且由于原作在 2016 年没再更新，其内容基于 Rust 2015 版本，续写的版本也只是把过时的细节更新至 2018 版本。

今年即将发布 2021 版本，**或许** 本书的内容会继续修正吧。毕竟这本书在阐述 **声明宏** 方面搭建了一个很小巧精美的骨架。

续作及本翻译渲染版本使用 mdbook 构建：

- 续作：[repo](#) | [渲染版](#)
- 续作中文翻译：[repo](#) | [渲染版](#) | [加速查看站点](#)

主要补充的部分在于：

- *macro_rules!* : [元变量 \(metavariables\)](#)
- [片段分类符 \(fragment-specifiers\)](#)
- 调试：[其他调试工具](#)
- [作用域](#)
- [导入 / 导出宏](#)
- 计数：[bit twiddling](#)
- 译者补充：[算盘游戏](#)
- 构件：[解析](#)

另外，此翻译版本提供的阅读功能：

1. 行间代码块大部分可以点击右上角按钮运行，有些可以 [编辑](#) 和运行（目的是快速而方便地验证读者思考的代码能否编译通过）。只用于展示说明、或者不适合运行的代码只有复制按钮。

区分能编辑代码块的方法：光标能够在代码块中停留和闪动；有同级竖线；右上角有 undo 图标；选中代码时背景色较浅；看高亮颜色。

2. 每个页面右侧都有本章节的 [大纲目录](#)，可以点击跳转。如果大纲目录显示不完整，可以缩小浏览器页面；或者收起左侧的章节目录。大纲目录仅在电脑网页版生效，移动端网页不会显示。

3. 所有 `code` 蓝色样式、光标移上去有下划线的内容（普通正文或者行内代码）都是链接，可以跳转。无链接的行内代码样式是这样的：`code`。

4. 翻译专有名词时，给出原英文，因为我认为那些词语是初次阅读英文时的障碍，所以当读者查阅其他英文资料时，就不会感到陌生了。

http://129.28.186.100/tlborm/translation_statement.html

编程范式

面向对象编程特性

设计模式

函数式语言功能

闭包

迭代器

安全并发

安全并发

网络开发

03 实战篇

03 实战篇

源码及管理

源码及管理

20 | 4 Steps : 如何更好地阅读Rust源码?

20 | 4 Steps : 如何更好地阅读Rust源码?

你好，我是陈天。

到目前为止，Rust 的基础知识我们就学得差不多了。这倒不是说已经像用筛子一样，把基础知识仔细筛了一遍，毕竟我只能给你提供学习Rust的思路，扫清入门障碍。老话说得好，师傅领进门修行靠个人，在 Rust 世界里打怪升级，还要靠你自己去探索、去努力。

虽然不能帮你打怪，但是打怪的基本技巧可以聊一聊。所以在开始阶段实操引入大量新第三方库之前，我们非常有必要先聊一下这个很重要的技巧：**如何更好地阅读源码**。

其实会读源码是个终生受益的开发技能，却往往被忽略。在我读过的绝大多数的编程书籍里，很少有讲如何阅读代码的，就像世间的书籍千千万万，“如何阅读一本书”这样的题材却凤毛麟角。

当然，在解决“如何”之前，我们要先搞明白“为什么”。

为什么要阅读源码？

如果课程的每一讲你都认真看过，会发现时刻都在引用标准库的源码，让我们在阅读的时候，不光学基础知识，还能围绕它的第一手资料也就是源代码展开讨论。

如果说他人总结的知识是果实，那源代码就是结出这果实的种子。只摘果子吃，就是等他人赏饭，非常被动，也不容易分清果子的好坏；如果靠朴素的源码种子结出了自己的果实，确实前期要耐得住寂寞施肥浇水，但收割的时刻，一切尽在自己的掌控之中。

作为开发者，我们每天都和代码打交道。经过数年的基础教育和职业培训，我们都会“写”代码，或者至少会抄代码和改代码。但是，会读代码的其实并不多，会读代码又真正能读懂一些大项目源码的，少之又少。

这种怪状，真要追究起来，就是因为前期我们所有的教育和培训都在强调怎么写代码，并没有教怎么读代码，而走入工作后，大多数场景也都是一个萝卜一个坑，我们只需要了解系统的一个局部就能开展工作，读和作品内容不相干的代码，似乎没什么用。

那没有读过大量代码究竟有什么问题，毕竟工作好像还是能正常开展？就拿跟写代码有很多相通之处的写作来对比。

小时候我们都经历过读课文、背课文、写作文的过程。除了学习语法和文法知识外，从小学开始，经年累月，阅读各种名家作品，经过各种写作训练，才累积出自己的写作能力。所以可以说，写作建立在大量阅读基础上。

而我们写代码的过程就很不同了，在学会基础的语法和试验了若干 example 后，跳过了大量阅读名家作品的阶段，直接坐火箭般蹿到自己开始写业务代码。

这样跳过了大量的代码阅读有三个问题：

首先没有足够积累，我们很容易养成 StackOverflow driven 的写代码习惯。

遇到不知如何写的代码，从网上找现成的答案，找个高票的复制粘贴，改一改凑活着用，先完成功能再说。写代码的过程中遇到问题，就开启调试模式，要么设置无数断点一步步跟踪，要么到处打印信息试图为满是窟窿的代码打上补丁，导致写代码的整个过程就是一部调代码的血泪史。

其次，**因为平时基础不牢靠，我们靠边写边学的进步是最慢的**。道理很简单，前辈们踩过坑总结的经验教训，都不得不亲自用最慢的法子一点点试着踩一遍。

最后还有一个非常容易被忽略的天花板问题，**周围能触达的那个最强工程师开发水平的上限，就是我们的上限**。

但是如果重视读源码平时积累，并且具备一定阅读技巧，这三个问题就能迎刃而解。就像写作文形容美女时，你立即能想到“肌肤胜雪、明眸善睐、齿如含贝、气若幽兰……”，而不是憋了半天就三字“哇美女”。为了让我们在写代码的时候，摆脱只会“哇美女”这样的初级阶段，多读源码非常关键。

三大功用

读源码的第一个好处是，**知识的源头在你这里，你可以根据事实来分辨是非，而不是迷信权威**。比如说之前讲 Rc 时（[第9讲](#)），我们通过源码引出 Box::leak，回答了为啥 Rc 可以突破 Rust 单一所有权的桎梏；谈到 FnOnce 时（[第19讲](#)），通过源码一眼看透为啥 FnOnce 只能调用一次。

未来你在跟别人分享的时候，可以很自信地回答这些问题，而不必说因为《陈天的 Rust 第一课》里是这么说的，这也解决了刚才的第一个问题。

通过源码我们还学到了很多技巧。比如 Rc::clone() 如何使用内部可变性来保持 Clone trait 的不可变约束（[第9讲](#)）；Iterator 里的方法如何通过不断构造新的 Iterator 数据结构，来支持 lazy evaluation（[第16讲](#)）。

未来你在写代码时，这些技巧都可以使用，从“哇美女”的初级水平到可以试着使用“一笑倾城，再笑倾国”的地步。这是读源码的第二个好处，**看别人的代码，积累了素材，开拓了思路，自己写代码时可以“文思如泉涌，下笔如有神”**。

最后一个能解决的问题就是打破天花板了。累积素材是基础，被启发出来的思路将这些素材串成线，才形成了自己的知识。

优秀的代码读得越多，越能引发思考，从而引发更多的阅读，形成一个飞轮效应，让自己的知识变得越来越丰富。而知识的融会贯通，最终形成读代码的第三大功用：**通过了解、吸收别人的思想，去芜存菁，最终形成自己的思想或者说智慧**。

当然从素材、到知识、再到智慧，要长期积累，并非一朝一夕之功。搞明白“为什么”给到我们的三个学习方向，所以现在来进一步解决“如何”，分享一下我的方法论，为你的积累助力。

如何阅读源码呢？

我们以第三方库 [Bytes](#) 为例，来看看如何阅读源码。希望你跟着今天的节奏走，不管是否关心 bytes 的实现，都先以它为蓝本，把基本方法熟悉一遍，再扩展到更多代码的阅读，比如 [hyper](#)、[nom](#)、[tokio](#)、[tonic](#) 等。

Bytes 是 tokio 下一个高效处理网络数据的库，代码本身 3.5k LoC（不包括 2.1k LoC 注释），加上测试 5.3k。代码结构非常简单：

```
tree src
src
  buf
    buf_impl.rs
    buf_mut.rs
    chain.rs
    iter.rs
    limit.rs
    mod.rs
    reader.rs
    take.rs
    uninit_slice.rs
    vec_deque.rs
    writer.rs
  bytes.rs
  bytes_mut.rs
  fmt
    debug.rs
    hex.rs
    mod.rs
  lib.rs
  loom.rs
  serde.rs
```

能看到，脉络很清晰，是很容易阅读的代码。

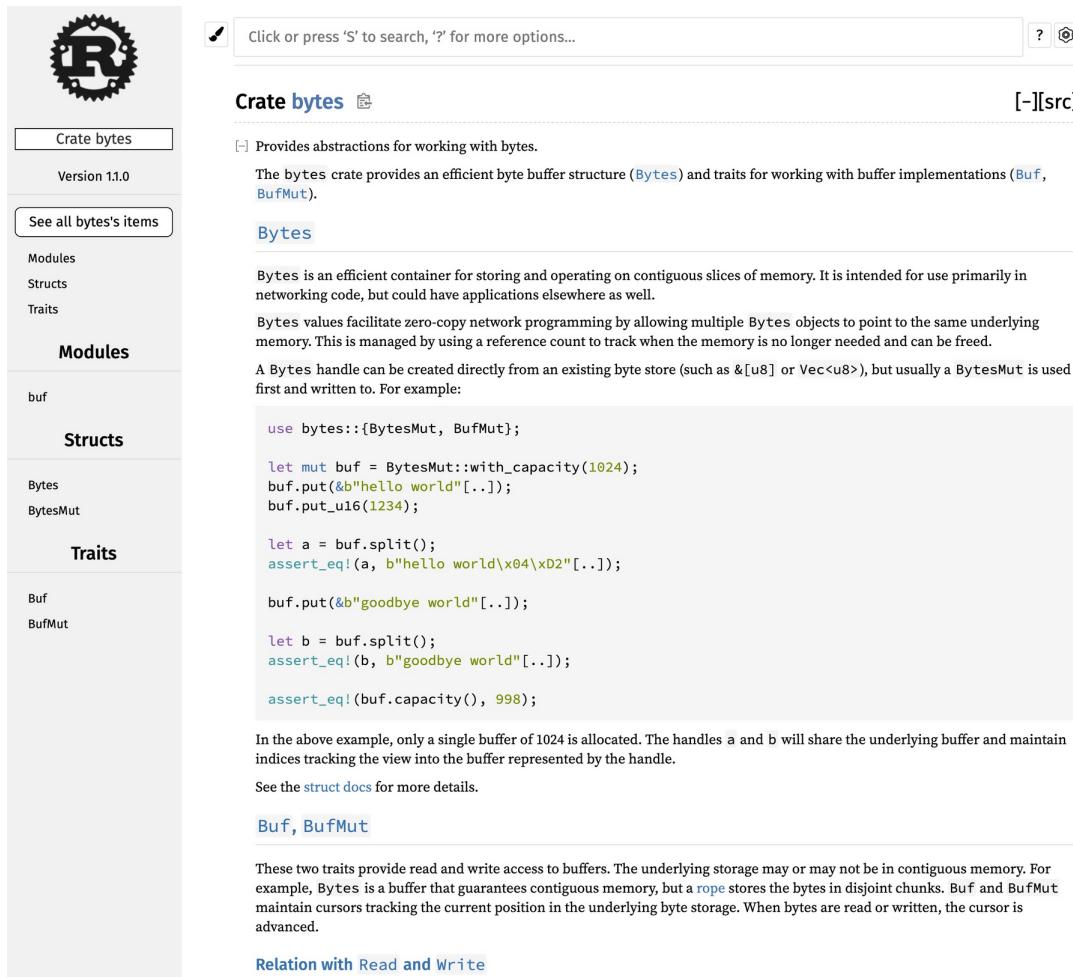
先简单讲一下读 Rust 代码的顺序：从 crate 的大纲开始，先了解目标代码能干什么、怎么用；然后学习核心 trait，看看它支持哪些功能；之后再掌握主要的数据结构，开始写一些示例代码；最后围绕自己感兴趣的情景深入阅读。

至于为什么这么读，我们边读边具体说明。

step1：从大纲开始

我们先从文档的大纲入手。Rust 的文档系统是所有编程语言中处在第一梯队的，即便不是最好的，也是最好之一。它的文档和代码结合地很紧密，可以来回跳转。

Rust 几乎所有库的文档都在 [docs.rs](#) 下，比如 Bytes 的文档可以通过 [docs.rs/bytes](#) 访问：



The screenshot shows the Bytes crate documentation page on docs.rs. The left sidebar lists modules, structs, and traits for Bytes, Buf, and BufMut. The main content area is titled "Bytes" and describes it as an efficient container for memory slices. It includes code examples for creating buffers and splitting them into multiple handles that share the same underlying memory. A note states that handles will maintain their own indices. Below the code, there's a link to struct docs for more details. At the bottom, there's a section about Buf and BufMut traits.

Click or press 'S' to search, '?' for more options...

Crate bytes

Provides abstractions for working with bytes.

The bytes crate provides an efficient byte buffer structure ([Bytes](#)) and traits for working with buffer implementations ([Buf](#), [BufMut](#)).

Bytes

Bytes is an efficient container for storing and operating on contiguous slices of memory. It is intended for use primarily in networking code, but could have applications elsewhere as well.

Bytes values facilitate zero-copy network programming by allowing multiple Bytes objects to point to the same underlying memory. This is managed by using a reference count to track when the memory is no longer needed and can be freed.

A Bytes handle can be created directly from an existing byte store (such as `&[u8]` or `Vec<u8>`), but usually a BytesMut is used first and written to. For example:

```
use bytes::{BytesMut, BufMut};  
  
let mut buf = BytesMut::with_capacity(1024);  
buf.put(b"hello world"[..]);  
buf.put_u16(1234);  
  
let a = buf.split();  
assert_eq!(a, b"hello world\x04\xD2"[..]);  
  
buf.put(&b"goodbye world"[..]);  
  
let b = buf.split();  
assert_eq!(b, b"goodbye world"[..]);  
  
assert_eq!(buf.capacity(), 998);
```

In the above example, only a single buffer of 1024 is allocated. The handles a and b will share the underlying buffer and maintain indices tracking the view into the buffer represented by the handle.

See the [struct docs](#) for more details.

Buf, BufMut

These two traits provide read and write access to buffers. The underlying storage may or may not be in contiguous memory. For example, Bytes is a buffer that guarantees contiguous memory, but a `rope` stores the bytes in disjoint chunks. Buf and BufMut maintain cursors tracking the current position in the underlying byte storage. When bytes are read or written, the cursor is advanced.

[Relation with Read and Write](#)

首先阅读 crate 的文档，这样可以快速了解这个 crate 是做什么的，就像阅读一本书的时候，可以从书的序和前言入手了解梗概。除此之外，我们还可以看一下源码根目录下的 [README.md](#)，作为补充资料。

有了大致了解后，你就可以深入了解自己感兴趣的内容。我们就按照初学的顺序来看。

对于 Bytes，我们看到它有两个 trait Buf / BufMut 以及两个数据结构 Bytes/BytesMut，没有 crate 级别的函数。接下来就是深入阅读代码了。

我看的顺序一般是：trait → struct → 函数/方法。因为这和我们写代码的思考方式非常类似：

- 先从需求的流程中敲定系统的行为，需要定义什么接口 trait；
- 再考虑系统有什么状态，定义了哪些数据结构struct；

- 最后到实现细节，包括如何为数据结构实现 trait、数据结构自身有什么算法、如何把整个流程串起来等等。

step2：熟悉核心 trait 的行为

所以先看trait，我们以 Buf trait 为例。点进去看文档，主页面给了这个 trait 的定义和一个使用示例。

注意左侧导航栏的“required Methods”和“Provided Methods”，前者是实现这个 trait 需要实现的方法，后者是缺省方法。也就是说数据结构只要实现了这个 trait 的三个方法：advance()、chunk() 和 remaining()，就可以自动实现所有的缺省方法。当然，你也可以重载某个缺省方法。

导航栏继续往下拉，可以看到 bytes 为哪些“foreign types”实现了 Buf trait，以及当前模块有哪些 implementors。这些信息很重要，说明了这个 trait 的生态：

get_int_le	fn get_i32_le(&mut self) -> i32	[src]
get_u128	fn get_u64(&mut self) -> u64	[src]
get_u128_le	fn get_u64_le(&mut self) -> u64	[src]
get_u16	fn get_i64(&mut self) -> i64	[src]
get_u16_le	fn get_i64_le(&mut self) -> i64	[src]
get_u32	fn get_uint(&mut self, nbytes: usize) -> u64	[src]
get_u32_le	fn get_uint_le(&mut self, nbytes: usize) -> u64	[src]
get_u64	fn get_int(&mut self, nbytes: usize) -> i64	[src]
get_u64_le	fn get_int_le(&mut self, nbytes: usize) -> i64	[src]
get_u8	fn copy_to_bytes(&mut self, len: usize) -> Bytes	[src]
get_uint		
get_uint_le		
has_remaining		
reader		
take		
Implementations on Foreign Types		
&[u8]		
&mut T		
Box<T>		
Cursor<T>		
VecDeque<u8>		
Implementors		
Other items in bytes		
Modules		
buf		
Structs		
Bytes		
BytesMut		
Traits		
Buf		
Implementors		
impl Buf for Bytes		[src]
impl Buf for BytesMut		[src]
impl<T, U> Buf for Chain<T, U>		[src]
where		
T: Buf,		
U: Buf,		
impl<T: Buf> Buf for Take<T>		[src]

对于其它数据类型 (foreign type) :

- 切片 &[u8]、VecDeque 都实现了 Buf trait;
- 如果 T 满足 Buf trait, 那么 &mut T、Box 也实现了 Buf trait;
- 如果 T 实现了 AsRef<[u8]>, 那 Cursor 也实现了 Buf trait。

所以回过头来, 上一幅图文档给到的示例, 一个 &[u8] 可以使用 Buf trait 里的方法就顺理成章了:

```
use bytes::Buf;

let mut buf = &b"hello world"[..];

assert_eq!(b'h', buf.get_u8());
assert_eq!(b'e', buf.get_u8());
assert_eq!(b'l', buf.get_u8());

let mut rest = [0; 8];
buf.copy_to_slice(&mut rest);
```

```
assert_eq!(&rest[..], &b"lo world"[..]);
```

而且也知道了，如果未来为自己的数据结构 T 实现 Buf trait，那么我们无需为 Box, &mut T 实现 Buf trait，这省去了在各种场景下使用 T 的诸多麻烦。

看到这里，我们目前还没有深入源码，但已经可以学习到高手定义 trait 的一些思路：

- 定义好 trait 后，**可以考虑一下标准库的数据结构**，哪些可以实现这个 trait。
- 如果未来别人的某个类型 T，实现了你的 trait，**那他的 &T、&mut T、Box 等衍生类型，是否能够自动实现这个 trait。**

好，接着看左侧导航栏中的“implementors”，Bytes、BytesMut、Chain、Take 都实现了 Buf trait，这样我们就知道了在这个 crate 里，哪些数据结构实现了这个 trait，之后遇到它们就知道都能用来做什么了。

现在，对 Buf trait 以及围绕着它的生态，我们已经有了一个基本的认识，后面你可以从几个方向深入学习：

- Buf trait 某个缺省方法是如何实现的，比如 [get_u8\(\)](#)。
- 其它类型是如何实现 Buf trait 的，比如 [&\[u8\]](#)。

你甚至不用 clone bytes 的源码，在 [docs.rs](#) 里就可以直接完成这些代码的阅读，非常方便。

step3：掌握主要的struct

扫完 trait 的基本功能后，我们再来看数据结构。以 Bytes 这个结构为例：

Struct Bytes

Methods

clear
copy_from_slice
from_static
is_empty
len
new
slice
slice_ref
split_off
split_to
truncate

Trait Implementations

AsRef<[u8]>
Borrow<[u8]>
Buf
Clone
Debug
Default
Deref
Drop
Eq
From<&static [u8]>
From<&static str>
From<Box<[u8], Global>>
From<BytesMut>
From<String>
From<Vec<u8, Global>>
FromIterator<u8>
Hash
IntoIterator
LowerHex
Ord
PartialEq<&'a T>
PartialEq<[u8]>
PartialEq<Bytes>

Click or press 'S' to search, '?' for more options...

Struct bytes::Bytes [-][src]

```
pub struct Bytes { /* fields omitted */ }
```

A cheaply cloneable and sliceable chunk of contiguous memory.
Bytes is an efficient container for storing and operating on contiguous slices of memory. It is intended for use primarily in networking code, but could have applications elsewhere as well.
Bytes values facilitate zero-copy network programming by allowing multiple Bytes objects to point to the same underlying memory.
Bytes does not have a single implementation. It is an interface, whose exact behavior is implemented through dynamic dispatch in several underlying implementations of Bytes.
All Bytes implementations must fulfill the following requirements:

- They are cheaply cloneable and thereby shareable between an unlimited amount of components, for example by modifying a reference count.
- Instances can be sliced to refer to a subset of the original buffer.

```
use bytes::Bytes;

let mut mem = Bytes::from("Hello world");
let a = mem.slice(0..5);

assert_eq!(a, "Hello");

let b = mem.split_to(6);

assert_eq!(mem, "world");
assert_eq!(b, "Hello");
```

Memory layout

The Bytes struct itself is fairly small, limited to 4 `usize` fields used to track information about which segment of the underlying memory the Bytes handle has access to.
Bytes keeps both a pointer to the shared state containing the full memory slice and a pointer to the start of the region visible by the handle. Bytes also tracks the length of its view into the memory.

Sharing

Bytes contains a vtable, which allows implementations of Bytes to define how sharing/cloning is implemented in detail. When `Bytes::clone()` is called, Bytes will call the vtable function for cloning the backing storage in order to share it behind between multiple Bytes instances.
For Bytes implementations which refer to constant memory (e.g. created via `Bytes::from_static()`) the cloning

一般来说，好的文档会给出数据结构的介绍、用法、使用时的注意事项，以及一些代码示例。了解了数据结构的基本介绍后，继续看看它的内部结构：

```
/// ````text
///
///     Arc ptrs          +-----+
///     _____ / | Bytes 2 |
///     /           +-----+
///     / | Bytes 1 |   |
///     | +-----+   |
///     | |       |   | / data   | tail
///     | data |   tail | /
///     v       v       v           v
/// +-----+-----+-----+
/// | Arc |       |           |   |
/// +-----+-----+-----+
/// ````

pub struct Bytes {
    ptr: *const u8,
    len: usize,
    // inlined "trait object"
    data: AtomicPtr<()>,
    vtable: &'static Vtable,
```

```
}

pub(crate) struct Vtable {
    /// fn(data, ptr, len)
    pub clone: unsafe fn(&AtomicPtr<()>, *const u8, usize) -> Bytes,
    /// fn(data, ptr, len)
    pub drop: unsafe fn(&mut AtomicPtr<()>, *const u8, usize),
}
```

数据结构的代码往往会有一些注释，帮助你理解它的设计。对于 Bytes 来说，顺着代码往下看：

- 它内部使用了裸指针和长度，模拟一个切片，指向内存中的一片连续地址；
- 同时，还使用了 AtomicPtr 和手工打造的 Vtable 来模拟了 trait object 的行为。
- 看 Vtable 的样子，大概可以推断出 Bytes 的 clone() 和 drop() 的行为是动态的，这是个很有意思的发现。

不过先不忙继续探索它如何实现这个行为的，继续看文档。

和 trait 类似的，在左侧的导航栏，有一些值得关注的信息（上图+下图）：这个数据结构有哪些方法（Methods）、实现了哪些 trait（Trait implementations），以及 Auto trait / Blanket trait 的实现。

`impl Send for Bytes` [src]

`impl Sync for Bytes` [src]

Auto Trait Implementations

- `impl RefUnwindSafe for Bytes` [src]
- `impl Unpin for Bytes` [src]
- `impl UnwindSafe for Bytes` [src]

Blanket Implementations

- `impl<T> Any for T` [src]
where
T: 'static + ?Sized,
- `pub fn type_id(&self) -> TypeId` [src]
Gets the TypeId of self. [Read more](#)
- `impl<T> Borrow<T> for T` [src]
where
T: ?Sized,
- `pub fn borrow(&self) -> &T` [src]
Immutably borrows from an owned value. [Read more](#)
- `impl<T> BorrowMut<T> for T` [src]
where
T: ?Sized,
- `pub fn borrow_mut(&mut self) -> &mut T` [src]
Mutably borrows from an owned value. [Read more](#)
- `impl<T> From<T> for T` [src]
- `pub fn from(t: T) -> T` [src]
Performs the conversion.
- `impl<T, U> Into<U> for T` [src]
where
U: From<T>,
- `pub fn into(self) -> U` [src]
Performs the conversion.
- `impl<T> ToOwned for T` [src]
where
T: Clone,
- `type Owned = T` [src]
The resulting type after obtaining ownership.

可以看到，Bytes 除了实现了刚才讲过的 Buf trait 外，还实现了很多标准 trait。

这也带给我们新的启发：**我们自己的数据结构，也应该尽可能实现需要的标准 trait**，包括但不限于：AsRef、Borrow、Clone、Debug、Default、Deref、Drop、PartialEq/Eq、From、Hash、Intolterator（如果是个集合类型）、PartialOrd/Ord 等。

注意，除了这些 trait 外，Bytes 还实现了 Send / Sync。如果看很多我们接触过的数据结构，比如 Vec，Send / Sync 是自动实现的，但 Bytes 需要手工实现：

```
unsafe impl Send for Bytes {}
unsafe impl Sync for Bytes {}
```

这是因为之前讲过，如果你的数据结构里使用了不支持 Send / Sync 的类型，编译器默认这个数据结构不能跨线程安全使用，不会自动添加 Send / Sync trait 的实现。但如果你能确保跨线程的安全性，可以手工通过 unsafe impl 实现它们。

了解一个数据结构实现了哪些 trait，非常有助于理解它如何使用。所以，**标准库里的主要 trait 我们一定要好好学习，多多使用，最好能形成肌肉记忆**。这样，学习别人的代码时，效率会很高。比如我看 Bytes 这个数据结构，扫一下它实现了哪些 trait，就基本能知道：

- 什么数据结构可以转化成 Bytes，也就是如何生成 Bytes 结构；
- Bytes 可以跟谁比较；
- Bytes 是否可以跨线程使用；
- 在使用中，Bytes 的行为和谁比较像（看 Deref trait）。

这就是肌肉记忆的好处。你可以去 [crates.io](#) 的 Data structures 类别下多翻翻不同的库，比如 [IndexMap](#)，看看它实现了哪些标准 trait，不了解的就看看那些 trait 的文档，也可以回顾[第 14 讲](#)（有哪些必须掌握的 trait）。

当你了解了数据结构的基本文档，知道它实现了哪些方法和哪些 trait 后，基本上，这个数据结构的使用就不在话下了。你也可以看源代码里的 examples 目录或者 tests 目录，看看数据结构对外是如何使用的，作为参考。

对于 bytes 库，它没有额外的 examples 目录，所以我们可以看 [tests/test_bytes.rs](#) 来理解 Bytes 类型可以如何使用。现在，你应该能比较从容地使用这个 Bytes 库了，不妨尝试写一些自己的示例代码，感受它的能力。

step4：深入研究实现逻辑

当 trait 和数据结构都掌握好，我们已经可以从它的接口上学到很多开发上的思想和技巧，一些关键接口，也了解了足够多的实现细节。获得的知识对使用这个库来做一些事情已经绰绰有余。

大部分对源代码的学习，可以就此止步。因为对我们来说，没有太富余的时间把每个遇到的库都从头到尾研究一番，只要搞明白如何使用好 Rust 生态中可用的库来构建想构建的系统，就足够了。

但有些时候，我们希望能够更深入一步。

比如说想更好地使用这个库，希望进一步了解 Bytes 是如何做到在多线程中可以共享数据的，它跟 Arc<Vec> 有什么区别，Arc<Vec> 是不是可以完成 Bytes 的工作？又或者说，在实现某个系统时，我们也想像 Bytes 这样，实现数据结构自己的 vtable，让数据结构更灵活。

这时就要去深入按主题阅读代码了。这里我推荐“**主题阅读**”或者说“**情境阅读**”，就是围绕着一个特定的使用场景，以这个场景的主流为脉络，搞明白实现原理。

这时，光靠 [docs.rs](#) 上的代码已经满足不了我们的需求，我们要把代码 clone 下来，用 VS Code 打开仔细研究。下图展示了本地 ~/projects/opensource/rust 目录下的代码，它们都是我在不同时期，为了不同的目的，在某些场景下阅读过的源代码：

```
> ls
Fuzzr/
NuProto/
PyOxidizer/
actix-net/
actix-web/
anyhow/
application-services/
arrow-bincode/
async-raft/
automerge-persistent/
automerge-rs/
bastion/
bevy/
bytes/
casbin-rs/
commitlog/
crossbeam/
darp/
deno/
diamond-types/
eventually-rs/
flume/
fluvio/
fst/
glean/
gluesql/
hashbrown/
hyper/
ipfs-embed/
json-rules-engine-rs/
jsonrpsee/
lemmy/
libipld/
libp2p-quic/
lighthouse/
linkerd2-proxy/
liveview-rust/
localnative/
loit/
lust/
matchi/
mdbook-scientific/
merk/
multistream-batch/
nannou/
nushell/
pallet/
paperclip/
parking_lot/
polkadot/
prost/
pyo3/
rust-memcache/
rust-sqlite/
rustler/
serde-diff/
rhat/
roapi/
routefinder/
rune/
rust/
rust-headless-chrome/
rust-lbtp2p/
rust-memcache/
rust_skiia/
rust_sqlite/
rustler/
serde-diff/
seshat/
slideo/
smartstring/
snow/
solana/
substrate/
tarp/
taur/
tendermint-rs/
tiny-skiia/
tls/
tokio/
tokio-tower/
tonic/
tower/
trie/
trillium/
usher/
vector/
vega_lite_4.rs/
warp/
yamux/
yew/
```

我们就继续以 Bytes 如何实现自己的 vtable 为例，深入看 Bytes 是如何 clone 的？看 clone 的实现：

```
impl Clone for Bytes {
    #[inline]
    fn clone(&self) -> Bytes {
        unsafe { (self.vtable.clone)(&self.data, self.ptr, self.len) }
    }
}
```

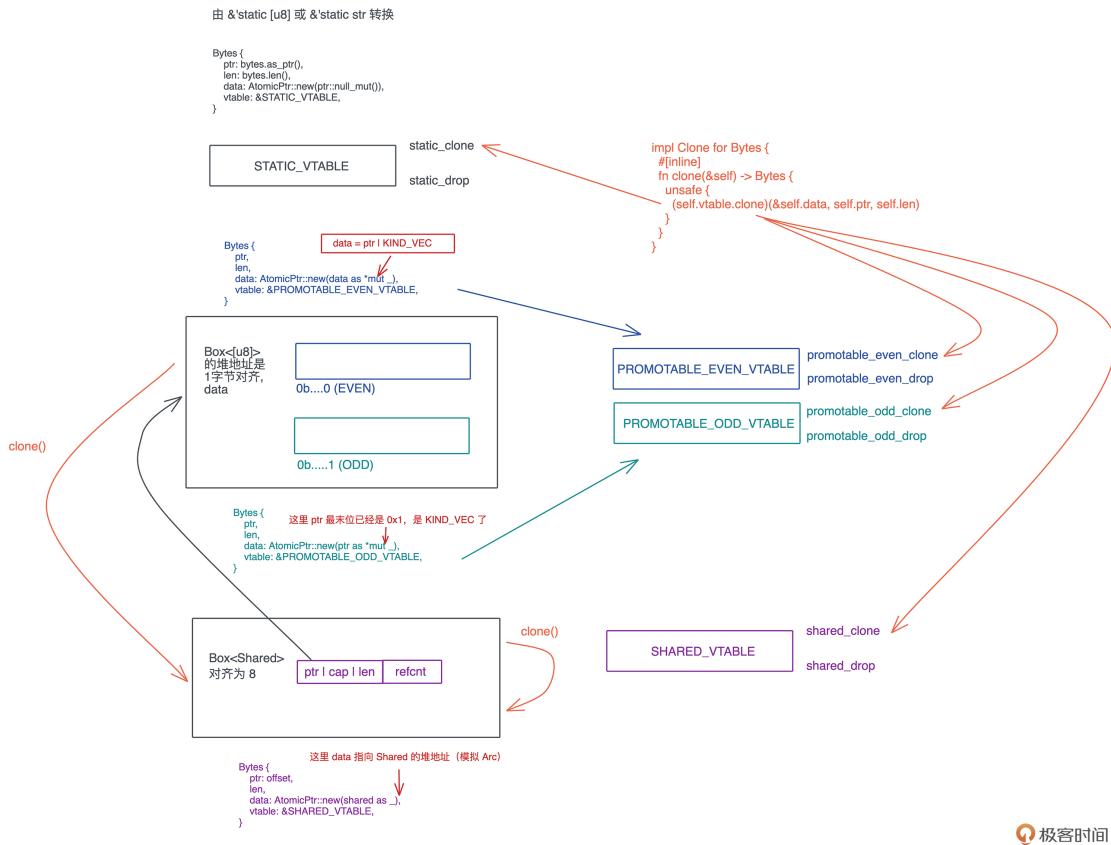
它用了 vtable 的 clone 方法，传入了 data，指向数据的指针以及长度。根据这个信息，我们如果能找到 Bytes 定义的所有 vtable，以及每个 vtable 的 clone() 做了什么事，就足以了解 Bytes 是如何实现 vtable 的了。

因为这一讲并非讲解 Bytes 是如何实现的，就不详细一步步带读代码了。相信你很快从代码中能够找到 STATIC_VTABLE、PROMOTABLE_EVEN_VTABLE、PROMOTABLE_ODD_VTABLE 和 SHARED_VTABLE 这四张表。

后三张表是处理动态数据的，在使用时如果 Bytes 的来源是 Vec、Box<[u8]> 或者 String，它们统统被转换成 Box<[u8]>，并在第一次 clone() 时，生成类似 Arc 的 Shared 结构，维护引用计数。

由于 Bytes 的 ptr 指向这个 Bytes 的起始地址，而 data 指向引用计数的地址，所以，你可以在这段内存上，生成任意多的、大小不同、起始位置不一样的 Bytes 结构，它们都

用同一个引用计数。这要比 Arc<Vec> 要灵活得多。具体流程，你可以看下图：



极客时间

在围绕着情景读代码时，建议你使用绘图工具，边读边记录（我用的excalidraw），非常有助于你理解代码脉络，不至于在无穷无尽的跳转中迷失了方向。

同时，善用 gdb 等工具来辅助阅读，就像第 17 讲我们剖析 HashMap 结构那样。一个场景理解完毕，这张脉络图也出来了，你可以对它稍作整理，使其成为自己知识库的一部分。

你也可以在团队内部的分享会上，对着图来分享代码，帮助团队更好地理解某些复杂的逻辑。所谓 learning by teaching，在分享的过程中，相当于又学了一遍，也许之前迷茫的地方会茅塞顿开，也许别人一个不经意的问题会让你思考之前没有想到的点。

小结

阅读别人的代码，尤其是优秀的代码，能帮助你快速地成长。

Rust 为了让代码和文档可读性更强，在工具链上做了巨大的努力，让我们在读源码或者别人代码的时候，很容易厘清代码的主要流程和使用方式。今天讲的阅读代码尤其是阅读 Rust 代码的很多技巧，少有人分享但又很重要，掌握好它，你就掌握了通向大牛之路的钥匙。

注意阅读的顺序：从大纲开始，先了解目标代码能干什么，怎么用；然后学习它的主要 trait；之后是数据结构，搞明白后再看看示例代码（examples）或者集成测试（tests），自己写一些示例代码；最后，围绕着自己感兴趣的情景深入阅读。并不是所有的代码都需要走到最后一步，你要根据自己的需要和精力量力而行。

思考题

1. 我们一起大致分析了 Bytes 的 clone() 的使用的场景，你能用类似的方式研究一下 drop() 是怎么工作的么？
2. 仔细看 Buf trait 里的方法，想想为什么它为 &mut T 实现了 Buf trait，但没有为 &T 实现 Buf trait 呢？如果你认为你找到了答案，再想想为什么它可以为 &[u8] 实现 Buf trait 呢？
3. 花点时间看看 BufMut trait 的文档。Vec 可以使用 BufMut 么？如果可以，试着写写代码在 Vec 上调用 BufMut 的各种接口，感受一下。
4. 如果有余力，可以研究一下 BytesMut。重点看一下 split_off() 方法是如何实现的。

欢迎你在留言区分享自己读源码的一些故事，欢迎抢答思考题。感谢你的一路坚持，今天你完成了Rust学习的第20次打卡，我们下节课开始第一个阶段的实操，下节课见～

参考资料

如果在阅读 Bytes 的 clone() 场景时，对于 PROMOTABLE_EVEN_VTABLE、PROMOTABLE_ODD_VTABLE 这两张表比较迷惑，且不明白为什么会根据 ptr & 0x1 是否等于 0 来提供不同的 vtable：

```
impl From<Box<[u8]>> for Bytes {
    fn from(slice: Box<[u8]>) -> Bytes {
        // Box<[u8]> doesn't contain a heap allocation for empty slices,
        // so the pointer isn't aligned enough for the KIND_VEC stashing to
        // work.
        if slice.is_empty() {
            return Bytes::new();
        }

        let len = slice.len();
        let ptr = Box::into_raw(slice) as *mut u8;
```

```
if ptr as usize & 0x1 == 0 {
    let data = ptr as usize | KIND_VEC;
    Bytes {
        ptr,
        len,
        data: AtomicPtr::new(data as *mut _),
        vtable: &PROMOTABLE_EVEN_VTABLE,
    }
} else {
    Bytes {
        ptr,
        len,
        data: AtomicPtr::new(ptr as *mut _),
        vtable: &PROMOTABLE_ODD_VTABLE,
    }
}
}
```

这是因为，`Box<[u8]>` 是 1 字节对齐，所以 `Box<[u8]>` 指向的堆地址可能末尾是 0 或者 1。而 `data` 这个 `AtomicPtr` 指针，在指向 `Shared` 结构时，这个结构的对齐是 2/4/8 字节（16/32/64 位 CPU 下），末尾一定为 0：

```
struct Shared {
    // holds vec for drop, but otherwise doesn't access it
    _vec: Vec<u8>,
    ref_cnt: AtomicUsize,
}
```

所以这里用了一个小技巧，以 `data` 指针末尾是否为 `0x1` 来区别，当前的 Bytes 是升级成共享，类似于 Arc 的结构 (`KIND_ARC`)，还是依旧停留在非共享的，类似 `Vec` 的结构 (`KIND_VEC`)。

这个复用指针最后几个 bit 记录一些 flag 的小技巧，在很多系统中都会使用。比如 Erlang VM，在存储 list 时，因为地址的对齐，最后两个 bit 不会被用到，所以当最后一个 bit 是 1 时，代表这是个指向 list 元素的地址。这种技巧，如果你不知道的话，看代码会很懵，一旦了解就没那么神秘了。

如果你觉得有收获，欢迎分享～

文档及测试

自动化测试

持续集成

持续集成

项目实践

04 | get hands dirty：来写个实用的CLI小工具

04 | get hands dirty：来写个实用的CLI小工具

你好，我是陈天。

在上一讲里，我们已经接触了 Rust 的基本语法。你是不是已经按捺不住自己的洪荒之力，想马上用 Rust 写点什么练练手，但是又发现自己好像有点“拔剑四顾心茫然”呢？

那这周我们就来玩个新花样，做一周“**learning by example**”的挑战，来尝试用 Rust 写三个非常有实际价值的小应用，感受下 Rust 的魅力在哪里，解决真实问题的能力到底如何。

你是不是有点担心，我才刚学了最基本语法，还啥都不知道呢，这就能开始写小应用了？那我碰到不理解的知识怎么办？

不要担心，因为你肯定会碰到不太懂的语法，但是，先不要强求自己理解，当成文言文抄写就可以了，哪怕这会不明白，只要你跟着课程节奏，通过撰写、编译和运行，你也能直观感受到 Rust 的魅力，就像小时候背唐诗一样。

好，我们开始今天的挑战。

HTTPie

为了覆盖绝大多数同学的需求，这次挑选的例子是工作中普遍会遇到的：写一个 CLI 工具，辅助我们处理各种任务。

我们就以实现 [HTTPie](#) 为例，看看用 Rust 怎么做 CLI。HTTPie 是用 Python 开发的，一个类似 cURL 但对用户更加友善的命令行工具，它可以帮助我们更好地诊断 HTTP 服务。

下图是用 HTTPie 发送了一个 post 请求的界面，你可以看到，相比 cURL，它在可用性上做了很多工作，包括对不同信息的语法高亮显示：

```
> http post httpbin.org/post greeting=hola name=Tyr
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 541
Content-Type: application/json
Date: Tue, 24 Aug 2021 16:56:38 GMT
Server: gunicorn/19.9.0

{
  "args": {},
  "data": "{\"greeting\": \"hola\", \"name\": \"Tyr\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "application/json, */*;q=0.5",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "35",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "HTTTPie/2.4.0",
    "X-Amzn-Trace-Id": "Root=1-612524c6-78b7832c3b9716d01da1854e"
  },
  "json": {
    "greeting": "hola",
    "name": "Tyr"
  },
  "origin": "174.21.120.54",
  "url": "http://httpbin.org/post"
}
```

你可以先想一想，如果用你最熟悉的语言实现 HTTTPie，要怎么设计、需要用到些什么库、大概用多少行代码？如果用 Rust 的话，又大概会要多少行代码？

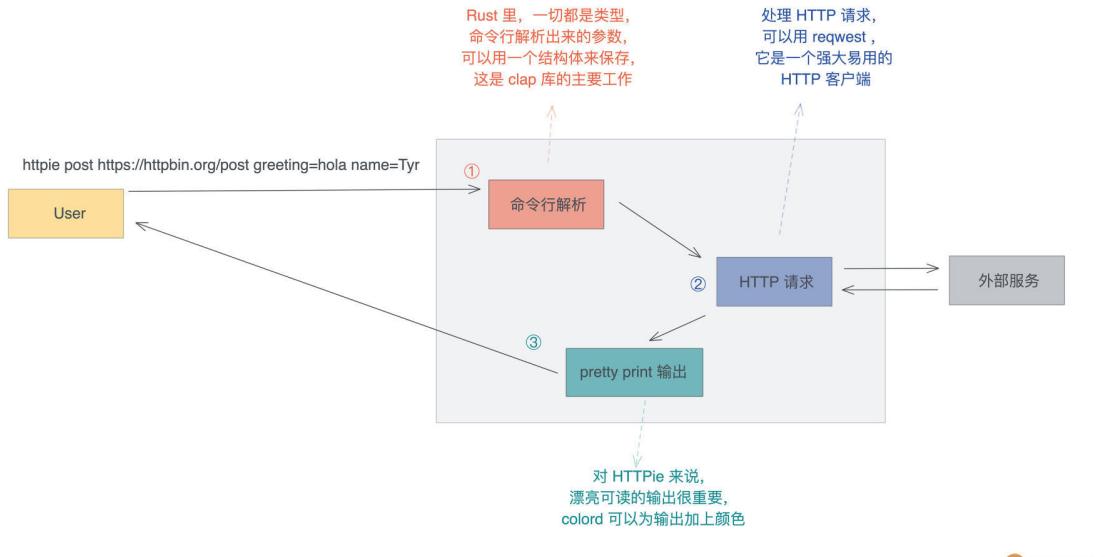
带着你自己的这些想法，开始动手用 Rust 构建这个工具吧！我们的目标是，**用大约 200 行代码实现这个需求。**

功能分析

要做一个 HTTTPie 这样的工具，我们先梳理一下要实现哪些主要功能：

- 首先是做命令行解析，处理子命令和各种参数，验证用户的输入，并且将这些输入转换成我们内部能理解的参数；
- 之后根据解析好的参数，发送一个 HTTP 请求，获得响应；
- 最后用对用户友好的方式输出响应。

这个流程你可以再看下图：



极客时间

我们来看要实现这些功能对应需要用到的库：

- 对于命令行解析，Rust 有很多库可以满足这个需求，我们今天使用官方比较推荐的 [clap](#)。
- 对于 HTTP 客户端，在上一讲我们已经接触过 [reqwest](#)，我们就继续使用它，只不过我们这次尝个鲜，使用它的异步接口。
- 对于格式化输出，为了让输出像 Python 版本的 HTTPie 那样显得生动可读，我们可以引入一个命令终端多彩显示的库，这里我们选择比较简单的 [colored](#)。
- 除此之外，我们还需要一些额外的库：用 [anyhow](#) 做错误处理、用 [jsonxf](#) 格式化 JSON 响应、用 [mime](#) 处理 mime 类型，以及引入 [tokio](#) 做异步处理。

CLI 处理

好，有了基本的思路，我们来创建一个项目，名字就叫 `httpie`：

```
cargo new httpie
cd httpie
```

然后，用 VSCode 打开项目所在的目录，编辑 `Cargo.toml` 文件，添加所需要的依赖：

```

[package]
name = "httpie"
version = "0.1.0"
edition = "2018"

[dependencies]
anyhow = "1" #
clap = "3.0.0-beta.4" #
colored = "2" #
jsonxf = "1.1" # JSON pretty print
mime = "0.3" # mime
reqwest = { version = "0.11", features = ["json"] } # HTTP
tokio = { version = "1", features = ["full"] } #

```

我们先在 main.rs 添加处理 CLI 相关的代码：

```

use clap::AppSettings, Clap;

// HTTPie CLI
// /// clap CLI

/// A naive httpie implementation with Rust, can you imagine how easy it is?
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
struct Opts {
    #[clap(subcommand)]
    subcmd: SubCommand,
}

// HTTP get / post
#[derive(Clap, Debug)]
enum SubCommand {
    Get(Get),
    Post(Post),
    // HTTP
}

// get

/// feed get with an url and we will retrieve the response for you
#[derive(Clap, Debug)]
struct Get {
    /// HTTP URL
    url: String,
}

// post URL key=value json body

/// feed post with an url and optional key=value pairs. We will post the data
/// as JSON, and retrieve the response for you
#[derive(Clap, Debug)]
struct Post {

```

```
    /// HTTP URL
    url: String,
    /// HTTP body
    body: Vec<String>,
}

fn main() {
    let opts: Opts = Opts::parse();
    println!("{:?}", opts);
}
```

代码中用到了 clap 提供的宏来让 CLI 的定义变得简单，这个宏能够生成一些额外的代码帮我们处理 CLI 的解析。通过 clap，我们只需要先用一个数据结构 T 描述 CLI 都会捕获什么数据，之后通过 T::parse() 就可以解析出各种命令行参数了。parse() 函数我们并没有定义，它是 #[derive(Clap)] 自动生成的。

目前我们定义了两个子命令，在 Rust 中子命令可以通过 enum 定义，每个子命令的参数又由它们各自的数据结构 Get 和 Post 来定义。

我们运行一下：

```
cargo build --quiet && target/debug/httpie post httpbin.org/post a=1 b=2
Opts { subcmd: Post(Post { url: "httpbin.org/post", body: ["a=1", "b=2"] }) }
```

默认情况下，cargo build 编译出来的二进制，在项目根目录的 target/debug 下。可以看到，命令行解析成功，达到了我们想要的功能。

加入验证

然而，现在我们还没对用户输入做任何检验，如果有这样的输入，URL 就完全解析错误了：

```
cargo build --quiet && target/debug/httpie post a=1 b=2
Opts { subcmd: Post(Post { url: "a=1", body: ["b=2"] }) }
```

所以，我们需要加入验证。输入有两项，**就要做两个验证，一是验证 URL，另一个是验证body。**

首先来验证 URL 是合法的：

```

use anyhow::Result;
use reqwest::Url;

#[derive(Clap, Debug)]
struct Get {
    /// HTTP URL
    #[clap(parse(try_from_str = parse_url))]
    url: String,
}

fn parse_url(s: &str) -> Result<String> {
    // URL
    let _url: Url = s.parse()?;
    Ok(s.into())
}

```

clap 允许你为每个解析出来的值添加自定义的解析函数，我们这里定义了个 `parse_url` 检查一下。

然后，我们要确保 `body` 里每一项都是 `key=value` 的格式。可以定义一个数据结构 `KvPair` 来存储这个信息，并且也自定义一个解析函数把解析的结果放入 `KvPair`：

```

use std::str::FromStr;
use anyhow::{anyhow, Result};

#[derive(Clap, Debug)]
struct Post {
    /// HTTP URL
    #[clap(parse(try_from_str = parse_url))]
    url: String,
    /// HTTP body
    #[clap(parse(try_from_str=parse_kv_pair))]
    body: Vec<KvPair>,
}

/// key=value parse_kv_pair KvPair
#[derive(Debug)]
struct KvPair {
    k: String,
    v: String,
}

/// FromStr trait str.parse() KvPair
impl FromStr for KvPair {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        // = split
        let mut split = s.split("=");
        let err = || anyhow!(format!("Failed to parse {}", s));
        Ok(Self {
            // key Some(T)/None

```

```

        // Ok(T)/Err(E) ?
        k: (split.next().ok_or_else(err)?).to_string(),
        // value
        v: (split.next().ok_or_else(err)?).to_string(),
    })
}
}

/// KvPair FromStr s.parse() KvPair
fn parse_kv_pair(s: &str) -> Result<KvPair> {
    Ok(s.parse()?)
}

```

这里我们实现了一个 [FromStr trait](#)，可以把满足条件的字符串转换成 KvPair。FromStr 是 Rust 标准库定义的 trait，实现它之后，就可以调用字符串的 parse() 泛型函数，很方便地处理字符串到某个类型的转换了。

这样修改完成后，我们的 CLI 就比较健壮了，可以再测试一下：

```

cargo build --quiet
target/debug/httpie post https://httpbin.org/post a=1 b
error: Invalid value for '<BODY>...': Failed to parse b

For more information try --help
target/debug/httpie post abc a=1
error: Invalid value for '<URL>': relative URL without a base

For more information try --help

target/debug/httpie post https://httpbin.org/post a=1 b=2
Opts { subcmd: Post(Post { url: "https://httpbin.org/post", body: [KvPair { k: "a", v: "1" },
KvPair { k: "b", v: "2" }] }) }

```

Cool，我们完成了基本的验证，不过很明显可以看到，我们并没有把各种验证代码一股脑塞在主流程中，而是通过实现额外的验证函数和 trait 来完成的，这些新添加的代码，高度可复用且彼此独立，并不用修改主流程。

这非常符合软件开发的开闭原则 ([Open-Closed Principle](#))：Rust 可以通过宏、trait、泛型函数、trait object 等工具，帮助我们更容易写出结构良好、容易维护的代码。

目前你也许还不太明白这些代码的细节，但是不要担心，继续写，今天先把代码跑起来就行了，不需要你搞懂每个知识点，之后我们都会慢慢讲到的。

HTTP 请求

好，接下来我们就继续进行 HTTPPie 的核心功能：HTTP 的请求处理了。我们在 main() 函数里添加处理子命令的流程：

```
use reqwest::{header, Client, Response, Url};

#[tokio::main]
async fn main() -> Result<()> {
    let opts: Opts = Opts::parse();
    // HTTP
    let client = Client::new();
    let result = match opts.subcmd {
        SubCommand::Get(ref args) => get(client, args).await?,
        SubCommand::Post(ref args) => post(client, args).await?,
    };
    Ok(result)
}
```

注意看我们把 main 函数变成了 async fn，它代表异步函数。对于 async main，我们需要使用 #[tokio::main] 宏来自动添加处理异步的运行时。

然后在 main 函数内部，我们根据子命令的类型，我们分别调用 get 和 post 函数做具体处理，这两个函数实现如下：

```
use std::collections::HashMap, str::FromStr;

async fn get(client: Client, args: &Get) -> Result<()> {
    let resp = client.get(&args.url).send().await?;
    println!("{}: {}", args.method, resp.text().await?);
    Ok(())
}

async fn post(client: Client, args: &Post) -> Result<()> {
    let mut body = HashMap::new();
    for pair in args.body.iter() {
        body.insert(&pair.k, &pair.v);
    }
    let resp = client.post(&args.url).json(&body).send().await?;
    println!("{}: {}", args.method, resp.text().await?);
    Ok(())
}
```

其中，我们解析出来的 KvPair 列表，需要装入一个 HashMap，然后传给 HTTP client 的 JSON 方法。这样，我们的 HTTPPie 的基本功能就完成了。

不过现在打印出来的数据对用户非常不友好，我们需要进一步用不同的颜色打印 HTTP header 和 HTTP body，就像 Python 版本的 HTTPPie 那样，这部分代码比较简单，我们就不详细介绍。

最后，来看完整的代码：

```
use anyhow::anyhow, Result;
use clap::{AppSettings, Clap};
use colored::*;
use mime::Mime;
use reqwest::{header, Client, Response, Url};
use std::{collections::HashMap, str::FromStr};

// CLI

// HTTPie CLI
// /// clap CLI

/// A naive httpie implementation with Rust, can you imagine how easy it is?
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
struct Opts {
    #[clap(subcommand)]
    subcmd: SubCommand,
}

// HTTP get / post
#[derive(Clap, Debug)]
enum SubCommand {
    Get(Get),
    Post(Post),
    // HTTP
}
}

// get

/// feed get with an url and we will retrieve the response for you
#[derive(Clap, Debug)]
struct Get {
    /// HTTP URL
    #[clap(parse(try_from_str = parse_url))]
    url: String,
}

// post URL key=value json body

/// feed post with an url and optional key=value pairs. We will post the data
/// as JSON, and retrieve the response for you
#[derive(Clap, Debug)]
struct Post {
    /// HTTP URL
    #[clap(parse(try_from_str = parse_url))]
    url: String,
    /// HTTP body
    #[clap(parse(try_from_str=parse_kv_pair))]
    body: Vec<KvPair>,
}

/// key=value parse_kv_pair KvPair
```

```

#[derive(Debug, PartialEq)]
struct KvPair {
    k: String,
    v: String,
}

/// FromStr trait str.parse() KvPair
impl FromStr for KvPair {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        // = split
        let mut split = s.split "=";
        let err = || anyhow!(format!("Failed to parse {}", s));
        Ok(Self {
            // key Some(T)/None
            // Ok(T)/Err(E) ?
            k: (split.next().ok_or_else(err)?).to_string(),
            // value
            v: (split.next().ok_or_else(err)?).to_string(),
        })
    }
}

/// KvPair FromStr s.parse() KvPair
fn parse_kv_pair(s: &str) -> Result<KvPair> {
    Ok(s.parse()?)
}

fn parse_url(s: &str) -> Result<String> {
    // URL
    let _url: Url = s.parse()?;
    Ok(s.into())
}

/// get
async fn get(client: Client, args: &Get) -> Result<()> {
    let resp = client.get(&args.url).send().await?;
    Ok(print_resp(resp).await?)
}

/// post
async fn post(client: Client, args: &Post) -> Result<()> {
    let mut body = HashMap::new();
    for pair in args.body.iter() {
        body.insert(&pair.k, &pair.v);
    }
    let resp = client.post(&args.url).json(&body).send().await?;
    Ok(print_resp(resp).await?)
}

// +
fn print_status(resp: &Response) {
    let status = format!("{}:{} {}", resp.version(), resp.status()).blue();
    println!("{}\n", status);
}

```

```

// HTTP header
fn print_headers(resp: &Response) {
    for (name, value) in resp.headers() {
        println!("{}: {:?}", name.to_string().green(), value);
    }

    print!("\n");
}

/// HTTP body
fn print_body(m: Option<Mime>, body: &String) {
    match m {
        // "application/json" pretty print
        Some(v) if v == mime::APPLICATION_JSON => {
            println!("{}: {}", jsonxf::pretty_print(body).unwrap().cyan())
        }
        // mime type
        _ => println!("{}: {}", body),
    }
}

/// 
async fn print_resp(resp: Response) -> Result<()> {
    print_status(&resp);
    print_headers(&resp);
    let mime = get_content_type(&resp);
    let body = resp.text().await?;
    print_body(mime, &body);
    Ok(())
}

/// content-type Mime
fn get_content_type(resp: &Response) -> Option<Mime> {
    resp.headers()
        .get(header::CONTENT_TYPE)
        .map(|v| v.to_str().unwrap().parse().unwrap())
}

/// HTTP tokio
#[tokio::main]
async fn main() -> Result<()> {
    let opts: Opts = Opts::parse();
    let mut headers = header::HeaderMap::new();
    // HTTP HTTP
    headers.insert("X-POWERED-BY", "Rust".parse()?);
    headers.insert(header::USER_AGENT, "Rust Httpie".parse()?);
    let client = reqwest::Client::builder()
        .default_headers(headers)
        .build()?;
    let result = match opts.subcmd {
        SubCommand::Get(ref args) => get(client, args).await?,
        SubCommand::Post(ref args) => post(client, args).await?,
    };

    Ok(result)
}

// cargo test

```

```

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parse_url_works() {
        assert!(parse_url("abc").is_err());
        assert!(parse_url("http://abc.xyz").is_ok());
        assert!(parse_url("https://httpbin.org/post").is_ok());
    }

    #[test]
    fn parse_kv_pair_works() {
        assert!(parse_kv_pair("a").is_err());
        assert_eq!(
            parse_kv_pair("a=1").unwrap(),
            KvPair {
                k: "a".into(),
                v: "1".into()
            }
        );

        assert_eq!(
            parse_kv_pair("b=").unwrap(),
            KvPair {
                k: "b".into(),
                v: "".into()
            }
        );
    }
}

```

在这个完整代码的最后，我还撰写了几个单元测试，你可以用 cargo test 运行。Rust 支持条件编译，这里 #[cfg(test)] 表明整个 mod tests 都只在 cargo test 时才编译。

使用[代码行数统计工具 tokei](#) 可以看到，我们总共使用了 139 行代码，就实现了这个功能，其中还包含了约 30 行的单元测试代码：

```

tokei src/main.rs
-----
Language      Files      Lines      Code      Comments      Blanks
-----
Rust          1          200        139        33          28
-----
Total         1          200        139        33          28
-----
```

你可以使用 cargo build --release，编译出 release 版本，并将其拷贝到某个在 \$PATH 下的目录，然后体验一下：

```
> httpie
httpie 1.0

Tyr Chen <tvr@chen.com>

A naive httpie implementation with Rust, can you imagine how easy it is?

USAGE:
  httpie <SUBCOMMAND>

FLAGS:
  -h, --help      Print help information
  -V, --version   Print version information

SUBCOMMANDS:
  get    feed get with an url and we will retrieve the response for you
  help   Print this message or the help of the given subcommand(s)
  post   feed post with an url and optional key=value pairs. We will post the data as JSON,
         and retrieve the response for you
```

到这里一个带有完整帮助的 HTTPPie 就可以投入使用了。

我们测试一下效果：

```
> httpie post https://httpbin.org/post greeting=hola name=Tyr
HTTP/1.1 200 OK

date: "Tue, 24 Aug 2021 21:14:36 GMT"
content-type: "application/json"
content-length: "502"
connection: "keep-alive"
server: "gunicorn/19.9.0"
access-control-allow-origin: "*"
access-control-allow-credentials: "true"

{
  "args": {},
  "data": "{\"name\":\"Tyr\",\"greeting\":\"hola\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Content-Length": "32",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Rust Httppie",
    "X-Amzn-Trace-Id": "Root=1-6125613c-39e8b6523b9a89532ea21dbe",
    "X-Powered-By": "Rust"
  },
  "json": {
    "greeting": "hola",
    "name": "Tyr"
  },
  "origin": "174.21.120.54",
  "url": "https://httpbin.org/post"
}
```

这和官方的 HTTPPie 效果几乎一样。今天的源代码可以在[这里](#)找到。

哈，这个例子我们大获成功。我们只用了 100 行代码出头，就实现了 HTTPie 的核心功能，远低于预期的 200 行。不知道你能否从中隐约感受到 Rust 解决实际问题的能力，以今天实现的 HTTPie 为例，

- 要把命令行解析成数据结构，我们只需要在数据结构上，添加一些简单的标注就能搞定。
- 数据的验证，又可以由单独的、和主流程没有任何耦合关系的函数完成。
- 作为 CLI 解析库，clap 的整体体验和 Python 的 [click](#) 非常类似，但比 Golang 的 [cobra](#) 要更简单。

这就是 Rust 语言的能力体现，明明是面向系统级开发，却能够做出类似 Python 的抽象和体验，所以一旦你适应了 Rust，用起来就会感觉非常美妙。

小结

现在你应该有点明白，为什么我会在开篇词中会说，Rust 拥有强大的表现力。

或许你还是有点疑惑，这么学，我也太懵了，跟盲人摸象似的。其实初学者都会以为，必须要先搞明白所有的语法知识，才能动手写代码，不是的。

我们这周写三个实用例子的挑战，就是让你，在懵懂地撰写代码的过程中，直观感受 Rust 处理问题、解决问题的方式，同时可以跟你熟悉的语言去类比，无论是 Golang / Java，还是 Python / JavaScript，如果我用自己熟悉的语言怎么解决、Rust 给了我什么样的支持、我感觉它还缺什么。

在这个过程中，你脑子里会产生各种深度的思考，这些思考又必然会引发越来越多的问号，这是好事，带着这些问号，在未来的课程中才能更有目的地学习，也一定会学得深刻而有效。

今天的小挑战并不太难，你可能还意犹未尽。别急，下一讲我们会再写个难度大一点的、工作中都会用到的 Web 服务，继续体验 Rust 的魅力。

思考题

我们只是实现了 HTTP header 和 body 的高亮区分，但是 HTTP body 还是有些不太美观，可以进一步做语法高亮，如果你完成了今天的代码，觉得自己学有余力可以再挑战一下，你不妨试一试用 [syntect](#) 继续完善我们的 HTTPie。[syntect](#) 是 Rust 的一个语法高亮库，非常强大。

欢迎在留言区分享你的思考。你的 Rust 学习第四次打卡成功，我们下一讲见！

05 | get hands dirty：做一个图片服务器有多难？

05 | get hands dirty：做一个图片服务器有多难？

你好，我是陈天。

上一讲我们只用了百来行代码就写出了 HTTPie 这个小工具，你是不是有点意犹未尽，今天我们就来再写一个实用的小例子，看看Rust还能怎么玩。

再说明一下，代码看不懂完全没关系，先不要强求理解，跟着我的节奏一行行写就好，先让自己的代码跑起来，感受 Rust 和自己常用语言的区别，看看代码风格是什么样的，就可以了。

今天的例子是我们在工作中都会遇到的需求：构建一个 Web Server，对外提供某种服务。类似上一讲的 HTTPie，我们继续找一个已有的开源工具用 Rust 来重写，但是今天来挑战一个稍大一点的项目：构建一个类似 [Thumbor](#) 的图片服务器。

Thumbor

Thumbor 是 Python 下的一个非常著名的图片服务器，被广泛应用在各种需要动态调整图片尺寸的场合里。

它可以通过一个很简单的 HTTP 接口，实现图片的动态剪切和大小调整，另外还支持文件存储、替换处理引擎等其他辅助功能。我在之前的创业项目中还用过它，非常实用，性能也还不错。

我们看它的例子：

```
http://<thumbor-server>/300x200/smart/thumbor.readthedocs.io/en/latest/_images/logo-thumbor.png
```

在这个例子里，Thumbor 可以对这个图片最后的 URL 使用 smart crop 剪切，并调整大小为 300x200 的尺寸输出，用户访问这个 URL 会得到一个 300x200 大小的缩略图。

我们今天就来实现它最核心的功能，对图片进行动态转换。你可以想一想，如果用你最熟悉的语言，要实现这个服务，怎么设计，需要用到些什么库，大概用多少行代码？如果用 Rust 的话，又大概会多少行代码？

带着你自己的一些想法，开始用 Rust 构建这个工具吧！目标依旧是，用大约 200 行代码实现我们的需求。

设计分析

既然是图片转换，最基本的肯定是要支持各种各样的转换功能，比如调整大小、剪切、加水印，甚至包括图片的滤镜但是，**图片转换服务的难点其实在接口设计上**，如何设计一套易用、简洁的接口，让图片服务器未来可以很轻松地扩展。

为什么这么说，你想如果有一天，产品经理来找你，突然想让原本只用来做缩略图的图片服务，支持老照片的滤镜效果，你准备怎么办？

Thumbor 给出的答案是，把要使用的处理方法的接口，按照一定的格式、一定的顺序放在 URL 路径中，不使用的图片处理方法就不放：

```
/hmac/trim/AxB:CxD/(adaptative-)(full-)fit-in/-Ex-F/HALIGN/VALIGN/smart/filters:FILTERNAME  
(ARGUMENT):FILTERNAME(ARGUMENT)/* IMAGE-URI*
```

但这样不容易扩展，解析起来不方便，也很难满足对图片做多个有序操作的要求，比如对某个图片我想先加滤镜再加水印，对另一个图片我想先加水印再加滤镜。

另外，如果未来要加更多的参数，一个不小心，还很可能和已有的参数冲突，或者造成 API 的破坏性更新（breaking change）。作为开发者，我们永远不要低估产品经理那颗什么奇葩想法都有的躁动的心。

所以，在构思这个项目的时候，**我们需要找一种更简洁且可扩展的方式，来描述对图片进行的一系列有序操作**，比如说：先做 resize，之后对 resize 的结果添加一个水印，最后统一使用一个滤镜。

这样的有序操作，对应到代码中，可以用列表来表述，列表中每个操作可以是一个 enum，像这样：

```
//  
struct ImageSpec {  
    specs: Vec<Spec>  
}  
  
//  
enum Spec {  
    Resize(Resize),  
    Crop(Crop),  
    ...  
}  
  
//  resize  
struct Resize {  
    width: u32,  
    height: u32  
}
```

现在需要的数据结构有了，刚才分析了 thumbor 使用的方式拓展性不好，那我们如何设计一个任何客户端可以使用的、体现在 URL 上的接口，使其能够解析成我们设计的数据结构呢？

使用 querystring 么？虽然可行，但它在图片处理步骤比较复杂的时候，容易无序增长，比如我们要对某个图片做七八次转换，这个 querystring 就会非常长。

我这里的思路是使用 protobuf。protobuf 可以描述数据结构，几乎所有语言都有对 protobuf 的支持。当用 protobuf 生成一个 image spec 后，我们可以将其序列化成字节流。但字节流无法放在 URL 中，怎么办？我们可以用 base64 转码！

顺着这个思路，来试着写一下描述 image spec 的 protobuf 消息的定义：

```
message ImageSpec { repeated Spec specs = 1; }

message Spec {
    oneof data {
        Resize resize = 1;
        Crop crop = 2;
        ...
    }
}

...
```

这样我们就可以在 URL 中，嵌入通过 protobuf 生成的 base64 字符串，来提供可扩展的图片处理参数。处理过的 URL 长这个样子：

```
http://localhost:3000/image/CgoKCAjYBBCgBiADCgY6BAgUEBQKBDICCAM/<encoded origin url>
```

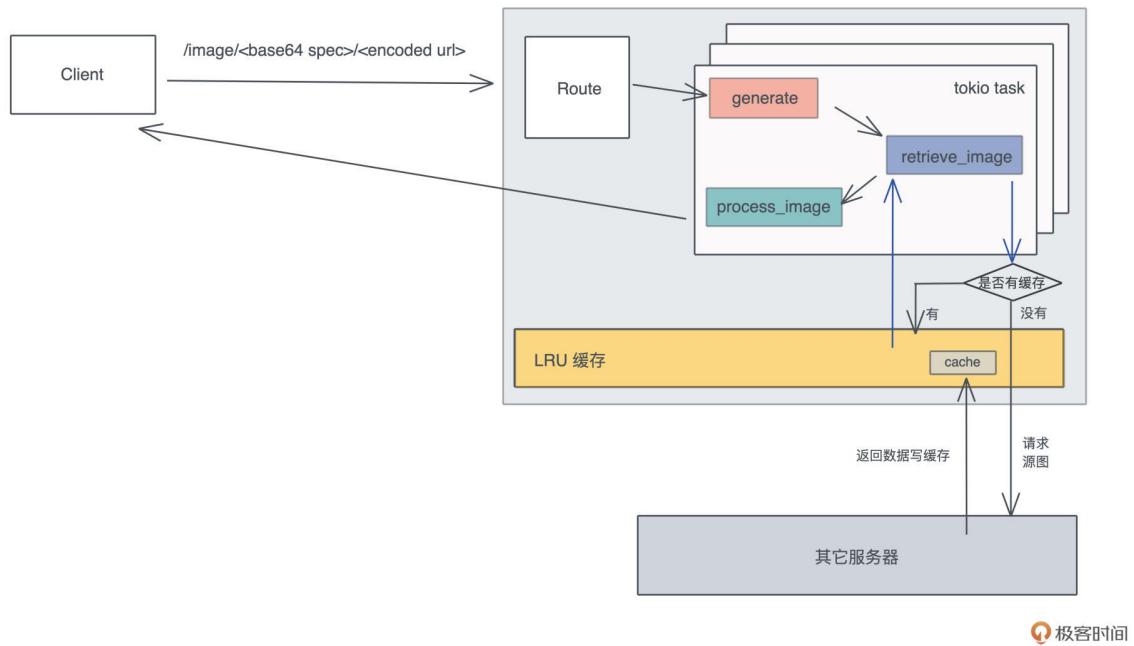
CgoKCAjYBBCgBiADCgY6BAgUEBQKBDICCAM 描述了我们上面说的图片的处理流程：先做 resize，之后对 resize 的结果添加一个水印，最后统一使用一个滤镜。它可以用下面的代码实现：

```
fn print_test_url(url: &str) {
    use std::borrow::Borrow;
    let spec1 = Spec::new_resize(600, 800, resize::SampleFilter::CatmullRom);
    let spec2 = Spec::new_watermark(20, 20);
    let spec3 = Spec::new_filter(filter::Filter::Marine);
    let image_spec = ImageSpec::new(vec![spec1, spec2, spec3]);
    let s: String = image_spec.borrow().into();
    let test_image = percent_encode(url.as_bytes(), NON_ALPHANUMERIC).to_string();
    println!("test url: http://localhost:3000/image/{}/{}", s, test_image);
}
```

使用 protobuf 的好处是，序列化后的结果比较小巧，而且任何支持 protobuf 的语言都可以生成或者解析这个接口。

好，接口我们敲定好，接下来就是做一个 HTTP 服务器提供这个接口。在 HTTP 服务器对 /image 路由的处理流程里，我们需要从 URL 中获取原始的图片，然后按照 image spec 依次处理，最后把处理完的字节流返回给用户。

在这个流程中，显而易见能够想到的优化是，**为原始图片的获取过程，提供一个 LRU (Least Recently Used) 缓存**，因为访问外部网络是整个路径中最缓慢也最不可控的环节。



极客时间

分析完后，是不是感觉 thumbor 也没有什么复杂的？不过你一定会有疑问：200 行代码真的可以完成这么多工作么？我们先写着，完成之后再来统计一下。

protobuf 的定义和编译

这个项目我们需要很多依赖，就不一一介绍了，未来在你的学习、工作中，大部分依赖你都会渐渐遇到和使用到。

我们照样先“cargo new thumbor”生成项目，然后在项目的 Cargo.toml 中添加这些依赖：

```
[dependencies]
axum = "0.2" # web
anyhow = "1" #
```

```

base64 = "0.13" # base64 /
bytes = "1" #
image = "0.23" #
lazy_static = "1" #
lru = "0.6" # LRU
percent-encoding = "2" # url /
photon-rs = "0.3" #
prost = "0.8" # protobuf
reqwest = "0.11" # HTTP client
serde = { version = "1", features = ["derive"] } # /
tokio = { version = "1", features = ["full"] } #
tower = { version = "0.4", features = ["util", "timeout", "load-shed", "limit"] } #
tower-http = { version = "0.1", features = ["add-extension", "compression-full", "trace"] } #
http
tracing = "0.1" #
tracing-subscriber = "0.2" #

[build-dependencies]
prost-build = "0.8" # protobuf

```

在项目根目录下，生成一个 abi.proto 文件，写入我们支持的图片处理服务用到的数据结构：

```

syntax = "proto3";

package abi; // prost abi.rs

// ImageSpec spec
message ImageSpec { repeated Spec specs = 1; }

// 
message Resize {
    uint32 width = 1;
    uint32 height = 2;

    enum ResizeType {
        NORMAL = 0;
        SEAM_CARVE = 1;
    }
}

ResizeType rtype = 3;

enum SampleFilter {
    UNDEFINED = 0;
    NEAREST = 1;
    TRIANGLE = 2;
    CATMULL_ROM = 3;
    GAUSSIAN = 4;
    LANCZOS3 = 5;
}

SampleFilter filter = 4;
}
// 
```

```

message Crop {
    uint32 x1 = 1;
    uint32 y1 = 2;
    uint32 x2 = 3;
    uint32 y2 = 4;
}

// 
message Fliph {}
// 
message Flipv {}
// 
message Contrast { float contrast = 1; }
// 
message Filter {
    enum Filter {
        UNSPECIFIED = 0;
        OCEANIC = 1;
        ISLANDS = 2;
        MARINE = 3;
        // more: https://docs.rs/photon-rs/0.3.1/photon\_rs/filters/fn.filter.html
    }
    Filter filter = 1;
}

// 
message Watermark {
    uint32 x = 1;
    uint32 y = 2;
}

// spec
message Spec {
    oneof data {
        Resize resize = 1;
        Crop crop = 2;
        Flipv flipv = 3;
        Fliph fliph = 4;
        Contrast contrast = 5;
        Filter filter = 6;
        Watermark watermark = 7;
    }
}

```

这包含了我们支持的图片处理服务，以后可以轻松扩展它来支持更多的操作。

protobuf 是一个向下兼容的工具，所以在服务器不断支持更多功能时，还可以和旧版本的客户端兼容。在 Rust 下，我们可以用 [prost](#) 来使用和编译 protobuf。同样，在项目根目录下，创建一个 build.rs，写入以下代码：

```

fn main() {
    prost_build::Config::new()
        .out_dir("src/pb")

```

```

    .compile_protos(&["abi.proto"], &["."])
    .unwrap();
}

```

build.rs 可以在编译 cargo 项目时，做额外的编译处理。这里我们使用 prost_build 把 abi.proto 编译到 src/pb 目录下。

这个目录现在还不存在，你需要 mkdir src/pb 创建它。运行 cargo build，你会发现在 src/pb 下，有一个 abi.rs 文件被生成出来，这个文件包含了从 protobuf 消息转换出来的 Rust 数据结构。我们先不用管 prost 额外添加的各种标记宏，就把它们当成普通的数据结构使用即可。

接下来，我们创建 src/pb/mod.rs，第三讲说过，一个目录下的所有代码，可以通过 mod.rs 声明。在这个文件中，我们引入 abi.rs，并且撰写一些辅助函数。这些辅助函数主要是为了，让 ImageSpec 可以被方便地转换成字符串，或者从字符串中恢复。

另外，我们还写了一个测试确保功能的正确性，你可以 cargo test 测试一下。记得在 main.rs 里添加 mod pb; 引入这个模块。

```

use base64::{decode_config, encode_config, URL_SAFE_NO_PAD};
use photon_rs::transform ::SamplingFilter;
use prost::Message;
use std::convert::TryFrom;

mod abi; // abi.rs
pub use abi::*;

impl ImageSpec {
    pub fn new(specs: Vec<Spec>) -> Self {
        Self { specs }
    }
}

// ImageSpec
impl From<&ImageSpec> for String {
    fn from(image_spec: &ImageSpec) -> Self {
        let data = image_spec.encode_to_vec();
        encode_config(data, URL_SAFE_NO_PAD)
    }
}

// ImageSpec s.parse().unwrap()
impl TryFrom<&str> for ImageSpec {
    type Error = anyhow::Error;

    fn try_from(value: &str) -> Result<Self, Self::Error> {
        let data = decode_config(value, URL_SAFE_NO_PAD)?;
        Ok(ImageSpec::decode(&data[...])?)
    }
}

```

```

// photon_rs
impl filter::Filter {
    pub fn to_str(&self) -> Option<&'static str> {
        match self {
            filter::Filter::Unspecified => None,
            filter::Filter::Oceanic => Some("oceanic"),
            filter::Filter::Islands => Some("islands"),
            filter::Filter::Marine => Some("marine"),
        }
    }
}

// SampleFilter photon_rs SamplingFilter
impl From<resize::SampleFilter> for SamplingFilter {
    fn from(v: resize::SampleFilter) -> Self {
        match v {
            resize::SampleFilter::Undefined => SamplingFilter::Nearest,
            resize::SampleFilter::Nearest => SamplingFilter::Nearest,
            resize::SampleFilter::Triangle => SamplingFilter::Triangle,
            resize::SampleFilter::CatmullRom => SamplingFilter::CatmullRom,
            resize::SampleFilter::Gaussian => SamplingFilter::Gaussian,
            resize::SampleFilter::Lanczos3 => SamplingFilter::Lanczos3,
        }
    }
}

// spec
impl Spec {
    pub fn new_resize_seam_carve(width: u32, height: u32) -> Self {
        Self {
            data: Some(spec::Data::Resize(Resize {
                width,
                height,
                rtype: resize::ResizeType::SeamCarve as i32,
                filter: resize::SampleFilter::Undefined as i32,
            })),
        }
    }

    pub fn new_resize(width: u32, height: u32, filter: resize::SampleFilter) -> Self {
        Self {
            data: Some(spec::Data::Resize(Resize {
                width,
                height,
                rtype: resize::ResizeType::Normal as i32,
                filter: filter as i32,
            })),
        }
    }

    pub fn new_filter(filter: filter::Filter) -> Self {
        Self {
            data: Some(spec::Data::Filter(Filter {
                filter: filter as i32,
            })),
        }
    }
}

```

```

pub fn new_watermark(x: u32, y: u32) -> Self {
    Self {
        data: Some(spec::Data::Watermark(Watermark { x, y })),
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    use std::borrow::Borrow;
    use std::convert::TryInto;

    #[test]
    fn encoded_spec_could_be_decoded() {
        let spec1 = Spec::new_resize(600, 600, resize::SampleFilter::CatmullRom);
        let spec2 = Spec::new_filter(filter::Filter::Marine);
        let image_spec = ImageSpec::new(vec![spec1, spec2]);
        let s: String = image_spec.borrow().into();
        assert_eq!(image_spec, s.as_str().try_into().unwrap());
    }
}

```

引入 HTTP 服务器

处理完和 protobuf 相关的内容，我们来处理 HTTP 服务的流程。Rust 社区有很多高性能的 Web 服务器，比如[actix-web](#)、[rocket](#)、[warp](#)，以及最近新出的 [axum](#)。我们就来用新鲜出炉的 axum 做这个服务器。

根据 axum 的文档，我们可以构建出下面的代码：

```

use axum::{extract::Path, handler::get, http::StatusCode, Router};
use percent_encoding::percent_decode_str;
use serde::Deserialize;
use std::convert::TryInto;

// protobuf
mod pb;

use pb::*;

// serde Deserializeaxum
#[derive(Deserialize)]
struct Params {
    spec: String,
    url: String,
}

#[tokio::main]
async fn main() {

```

```

// tracing
tracing_subscriber::fmt::init();

// 
let app = Router::new()
    // `GET /image` generate spec url
    .route("/image/:spec/:url", get(generate));

// web
let addr = "127.0.0.1:3000".parse().unwrap();
tracing::debug!("listening on {}", addr);
axum::Server::bind(&addr)
    .serve(app.into_make_service())
    .await
    .unwrap();
}

// 
async fn generate(Path(Params { spec, url }): Path<Params>) -> Result<String, StatusCode> {
    let url = percent_decode_str(&url).decode_utf8_lossy();
    let spec: ImageSpec = spec
        .as_str()
        .try_into()
        .map_err(|_| StatusCode::BAD_REQUEST)?;
    Ok(format!("url: {}\n spec: {:?}", url, spec))
}

```

把它们添加到 main.rs 后，使用 cargo run 运行服务器。然后我们就可以用上一讲做的 HTTPie 测试（eat your own dog food）：

```

httpie get "http://localhost:3000/image/CgoKCAjYBBCgBiADCgY6BAgUEBQKBDICCAM/https%3A%2F%
2Fimages%2Epexels%2Ecom%2Fphotos%2F2470905%2Fpexels%2Dphoto%2D2470905%2Ejpeg%3Fauto%3Dcompress%
26cs%3Dtinysrgb%26dpr%3D2%26h%3D750%26w%3D1260"
HTTP/1.1 200 OK

content-type: "text/plain"
content-length: "901"
date: "Wed, 25 Aug 2021 18:03:50 GMT"

url: https://images.pexels.com/photos/2470905/pexels-photo-2470905.jpeg?
auto=compress&cs=tinysrgb&dpr=2&h=750&w=1260
spec: ImageSpec {
    specs: [
        Spec {
            data: Some(
                Resize(
                    Resize {
                        width: 600,
                        height: 800,
                        rtype: Normal,
                        filter: CatmullRom,
                    },
                ),
            ),
        },
    ],
}

```

```
        } ,
    Spec {
        data: Some(
            Watermark(
                Watermark {
                    x: 20,
                    y: 20,
                } ,
            ) ,
        ) ,
    } ,
Spec {
    data: Some(
        Filter(
            Filter {
                filter: Marine,
            } ,
        ) ,
    ) ,
},
],

```

Wow, Web 服务器的接口部分我们已经能够正确处理了。

写到这里, 如果出现的语法让你觉得迷茫, 不要担心。因为我们还没有讲所有权、类型系统、泛型等内容, 所以很多细节你会看不懂。今天这个例子, 你只要跟我的思路走, 了解整个处理流程就可以了。

获取源图并缓存

好, 当接口已经可以工作之后, 我们再来处理获取源图的逻辑。

根据之前的设计, 需要引入 **LRU cache** 来缓存源图。一般 Web 框架都会有中间件来处理全局的状态, axum 也不例外, 可以使用 AddExtensionLayer 添加一个全局的状态, 这个状态目前就是 LRU cache, 在内存中缓存网络请求获得的源图。

我们把 main.rs 的代码, 改成下面的代码:

```
use anyhow::Result;
use axum::{
    extract::{Extension, Path},
    handler::get,
    http::{HeaderMap, HeaderValue, StatusCode},
    AddExtensionLayer, Router,
};
use bytes::Bytes;
```

```

use lru::LruCache;
use percent_encoding::{percent_decode_str, percent_encode, NON_ALPHANUMERIC};
use serde::Deserialize;
use std::{
    collections::hash_map::DefaultHasher,
    convert::TryInto,
    hash::{Hash, Hasher},
    sync::Arc,
};
use tokio::sync::Mutex;
use tower::ServiceBuilder;
use tracing::{info, instrument};

mod pb;

use pb::*;

#[derive(Deserialize)]
struct Params {
    spec: String,
    url: String,
}
type Cache = Arc<Mutex<LruCache<u64, Bytes>>>;

#[tokio::main]
async fn main() {
    // tracing
    tracing_subscriber::fmt::init();
    let cache: Cache = Arc::new(Mutex::new(LruCache::new(1024)));
    //
    let app = Router::new()
        // `GET /` 
        .route("/image/:spec/:url", get(generate))
        .layer(
            ServiceBuilder::new()
                .layer(AddExtensionLayer::new(cache))
                .into_inner(),
        );
    //

    // web
    let addr = "127.0.0.1:3000".parse().unwrap();

    print_test_url("https://images.pexels.com/photos/1562477/pexels-photo-1562477.jpeg?
auto=compress&cs=tinysrgb&dpr=3&h=750&w=1260");

    info!("Listening on {}", addr);

    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}

async fn generate(
    Path(Params { spec, url }): Path<Params>,
    Extension(cache): Extension<Cache>,
) -> Result<(HeaderMap, Vec<u8>), StatusCode> {
    let spec: ImageSpec = spec

```

```

    .as_str()
    .try_into()
    .map_err(|_| StatusCode::BAD_REQUEST)?;

let url: &str = &percent_decode_str(&url).decode_utf8_lossy();
let data = retrieve_image(&url, cache)
    .await
    .map_err(|_| StatusCode::BAD_REQUEST)?;

// TODO:

let mut headers = HeaderMap::new();

headers.insert("content-type", HeaderValue::from_static("image/jpeg"));
Ok((headers, data.to_vec()))
}

#[instrument(level = "info", skip(cache))]
async fn retrieve_image(url: &str, cache: Cache) -> Result<Bytes> {
    let mut hasher = DefaultHasher::new();
    url.hash(&mut hasher);
    let key = hasher.finish();

    let g = &mut cache.lock().await;
    let data = match g.get(&key) {
        Some(v) => {
            info!("Match cache {}", key);
            v.to_owned()
        }
        None => {
            info!("Retrieve url");
            let resp = reqwest::get(url).await?;
            let data = resp.bytes().await?;
            g.put(key, data.clone());
            data
        }
    };
    Ok(data)
}

// 
fn print_test_url(url: &str) {
    use std::borrow::Borrow;
    let spec1 = Spec::new_resize(500, 800, resize::SampleFilter::CatmullRom);
    let spec2 = Spec::new_watermark(20, 20);
    let spec3 = Spec::new_filter(filter::Filter::Marine);
    let image_spec = ImageSpec::new(vec![spec1, spec2, spec3]);
    let s: String = image_spec.borrow().into();
    let test_image = percent_encode(url.as_bytes(), NON_ALPHANUMERIC).to_string();
    println!("test url: http://localhost:3000/image/{}/{}", s, test_image);
}

```

这段代码看起来多，其实主要就是添加了 `retrieve_image` 这个函数。对于图片的网络请求，我们先把 URL 做个哈希，在 LRU 缓存中查找，找不到才用 `reqwest` 发送请求。

你可以 cargo run 运行一下现在的代码：

```
RUST_LOG=info cargo run --quiet

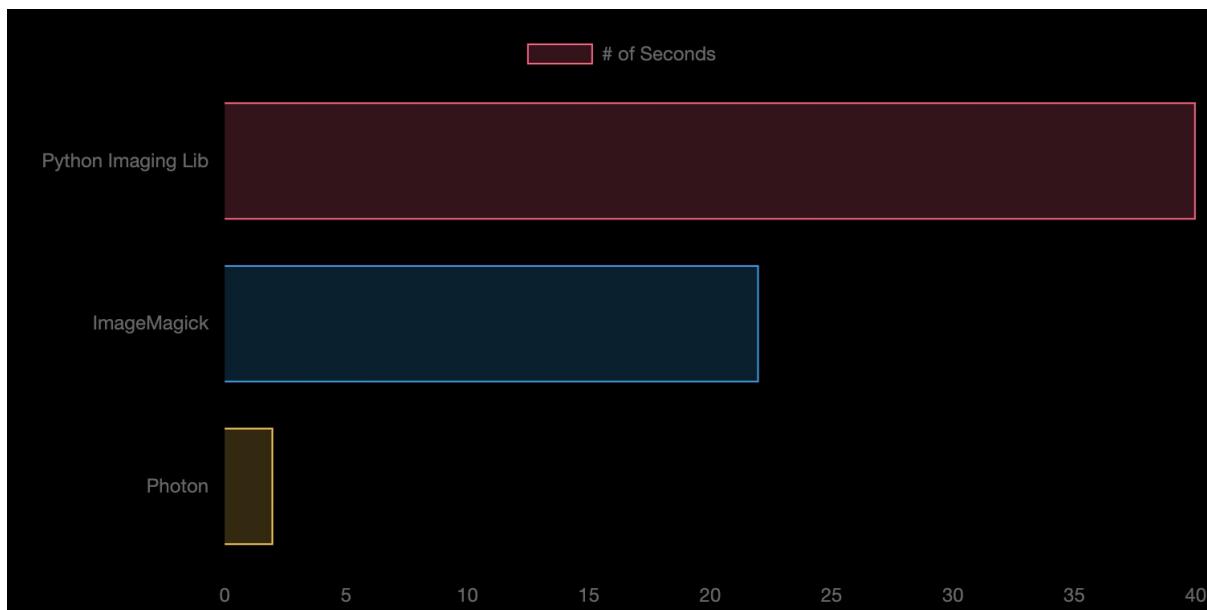
test url: http://localhost:3000/image/CgoKCAj0AxCgBiADCgY6BAgUEBQKBDICCAM/https%3A%2F%2Fimages%
2Epexels%2Ecom%2Fphotos%2F1562477%2Fpexels%2Dphoto%2D1562477%2Ejpeg%3Fauto%3Dcompress%26cs%
3Dtinysrgb%26dpr%3D3%26h%3D750%26w%3D1260
Aug 26 16:43:45.747  INFO server2: Listening on 127.0.0.1:3000
```

为了测试方便，我放了个辅助函数可以生成一个测试 URL，在浏览器中打开后会得到一个和源图一模一样的图片。这就说明，网络处理的部分，我们就搞定了。

图片处理

接下来，我们就可以处理图片了。Rust 下有一个不错的、偏底层的 [image](#) 库，围绕它有很多上层的库，包括我们今天要使用 [photon_rs](#)。

我扫了一下它的源代码，感觉它不算一个特别优秀的库，内部有太多无谓的内存拷贝，所以性能还有不少提升空间。就算如此，从 [photon_rs](#) 自己的 [benchmark](#) 看，也比 PIL / ImageMagick 性能好太多，这也算是 Rust 性能强大的一个小佐证吧。



因为 [photo_rs](#) 使用简单，这里我们也不太关心更高的性能，就暂且用它。然而，作为一个有追求的开发者，我们知道，有朝一日可能要用不同的 [image](#) 引擎替换它，所以我们设计一个 Engine trait：

```

// Engine trait engine engine
pub trait Engine {
    // engine specs
    fn apply(&mut self, specs: &[Spec]);
    // engine self self
    fn generate(self, format: ImageOutputFormat) -> Vec<u8>;
}

```

它提供两个方法，`apply` 方法对 `engine` 按照 `specs` 进行一系列有序的处理，`generate` 方法从 `engine` 中生成目标图片。

那么 `apply` 方法怎么实现呢？我们可以再设计一个 trait，这样可以为每个 `Spec` 生成对应处理：

```

// SpecTransform spec
pub trait SpecTransform<T> {
    // op transform
    fn transform(&mut self, op: T);
}

```

好，有了这个思路，我们创建 `src/engine` 目录，并添加 `src/engine/mod.rs`，在这个文件里添加对 trait 的定义：

```

use crate::pb::Spec;
use image::ImageOutputFormat;

mod photon;
pub use photon::Photon;

// Engine trait engine engine
pub trait Engine {
    // engine specs
    fn apply(&mut self, specs: &[Spec]);
    // engine self self
    fn generate(self, format: ImageOutputFormat) -> Vec<u8>;
}

// SpecTransform spec
pub trait SpecTransform<T> {
    // op transform
    fn transform(&mut self, op: T);
}

```

接下来我们再生成一个文件 `src/engine/photon.rs`，对 `photon` 实现 `Engine trait`，这个文件主要是一些功能的实现细节，就不详述了，你可以看注释。

```

use super::{Engine, SpecTransform};
use crate::pb::*;
use anyhow::Result;
use bytes::Bytes;
use image::{DynamicImage, ImageBuffer, ImageOutputFormat};
use lazy_static::lazy_static;
use photon_rs::{
    effects, filters, multiple, native::open_image_from_bytes, transform, PhotonImage,
};

use std::convert::TryFrom;

lazy_static! {
    // 
    static ref WATERMARK: PhotonImage = {
        // github
        // include_bytes!
        let data = include_bytes!("../../../../rust-logo.png");
        let watermark = open_image_from_bytes(data).unwrap();
        transform::resize(&watermark, 64, 64, transform::SamplingFilter::Nearest)
    };
}

// Photon engine
pub struct Photon(PhotonImage);

// Bytes Photon
impl TryFrom<Bytes> for Photon {
    type Error = anyhow::Error;

    fn try_from(data: Bytes) -> Result<Self, Self::Error> {
        Ok(Self(open_image_from_bytes(&data)?))
    }
}

impl Engine for Photon {
    fn apply(&mut self, specs: &[Spec]) {
        for spec in specs.iter() {
            match spec.data {
                Some(spec::Data::Crop(ref v)) => self.transform(v),
                Some(spec::Data::Contrast(ref v)) => self.transform(v),
                Some(spec::Data::Filter(ref v)) => self.transform(v),
                Some(spec::Data::Fliph(ref v)) => self.transform(v),
                Some(spec::Data::Flipv(ref v)) => self.transform(v),
                Some(spec::Data::Resize(ref v)) => self.transform(v),
                Some(spec::Data::Watermark(ref v)) => self.transform(v),
                // spec
                _ => {}
            }
        }
    }

    fn generate(self, format: ImageOutputFormat) -> Vec<u8> {
        image_to_buf(self.0, format)
    }
}

impl SpecTransform<&Crop> for Photon {
    fn transform(&mut self, op: &Crop) {

```

```

        let img = transform::crop(&mut self.0, op.x1, op.y1, op.x2, op.y2);
        self.0 = img;
    }
}

impl SpecTransform<&&Contrast> for Photon {
    fn transform(&mut self, op: &Contrast) {
        effects::adjust_contrast(&mut self.0, op.contrast);
    }
}

impl SpecTransform<&&Flipv> for Photon {
    fn transform(&mut self, _op: &Flipv) {
        transform::flipv(&mut self.0)
    }
}

impl SpecTransform<&&Fliph> for Photon {
    fn transform(&mut self, _op: &Fliph) {
        transform::fliph(&mut self.0)
    }
}

impl SpecTransform<&&Filter> for Photon {
    fn transform(&mut self, op: &Filter) {
        match filter::Filter::from_i32(op.filter) {
            Some(filter::Filter::Unspecified) => {}
            Some(f) => filters::filter(&mut self.0, f.to_str().unwrap()),
            _ => {}
        }
    }
}

impl SpecTransform<&&Resize> for Photon {
    fn transform(&mut self, op: &Resize) {
        let img = match resize::ResizeType::from_i32(op.rtype).unwrap() {
            resize::ResizeType::Normal => transform::resize(
                &mut self.0,
                op.width,
                op.height,
                resize::SampleFilter::from_i32(op.filter).unwrap().into(),
            ),
            resize::ResizeType::SeamCarve => {
                transform::seam_carve(&mut self.0, op.width, op.height)
            }
        };
        self.0 = img;
    }
}

impl SpecTransform<&&Watermark> for Photon {
    fn transform(&mut self, op: &Watermark) {
        multiple::watermark(&mut self.0, &WATERMARK, op.x, op.y);
    }
}

// photon
fn image_to_buf(img: PhotonImage, format: ImageOutputFormat) -> Vec<u8> {

```

```

let raw_pixels = img.get_raw_pixels();
let width = img.get_width();
let height = img.get_height();

let img_buffer = ImageBuffer::from_vec(width, height, raw_pixels).unwrap();
let dynimage = DynamicImage::ImageRgba8(img_buffer);

let mut buffer = Vec::with_capacity(32768);
dynimage.write_to(&mut buffer, format).unwrap();
buffer
}

```

好，图片处理引擎就搞定了。这里用了一个水印图片，你可以去 [GitHub repo](#) 下载，然后放在项目根目录下。我们同样把 engine 模块加入 main.rs，并引入 Photon：

```

mod engine;
use engine::{Engine, Photon};
use image::ImageOutputFormat;

```

还记得 src/main.rs 的代码中，我们留了一个 TODO 么？

```

// TODO:

let mut headers = HeaderMap::new();

headers.insert("content-type", HeaderValue::from_static("image/jpeg"));
Ok((headers, data.to_vec()))

```

我们把这段替换掉，使用刚才写好的 Photon 引擎处理：

```

// image engine
let mut engine: Photon = data
    .try_into()
    .map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
engine.apply(&spec.specs);

let image = engine.generate(ImageOutputFormat::Jpeg(85));

info!("Finished processing: image size {}", image.len());
let mut headers = HeaderMap::new();

headers.insert("content-type", HeaderValue::from_static("image/jpeg"));
Ok((headers, image))

```

这样整个服务器的全部流程就完成了，完整的代码可以在 [GitHub repo](#) 访问。

我在网上随手找了一张图片来测试下效果。用 cargo build --release 编译 thumbor 项目，然后打开日志运行：

```
RUST_LOG=info target/release/thumbor
```

打开测试链接，在浏览器中可以看到左下角的处理后图片。（原图片来自 [pexels](#)，发布者 [Min An](#)）



500*800

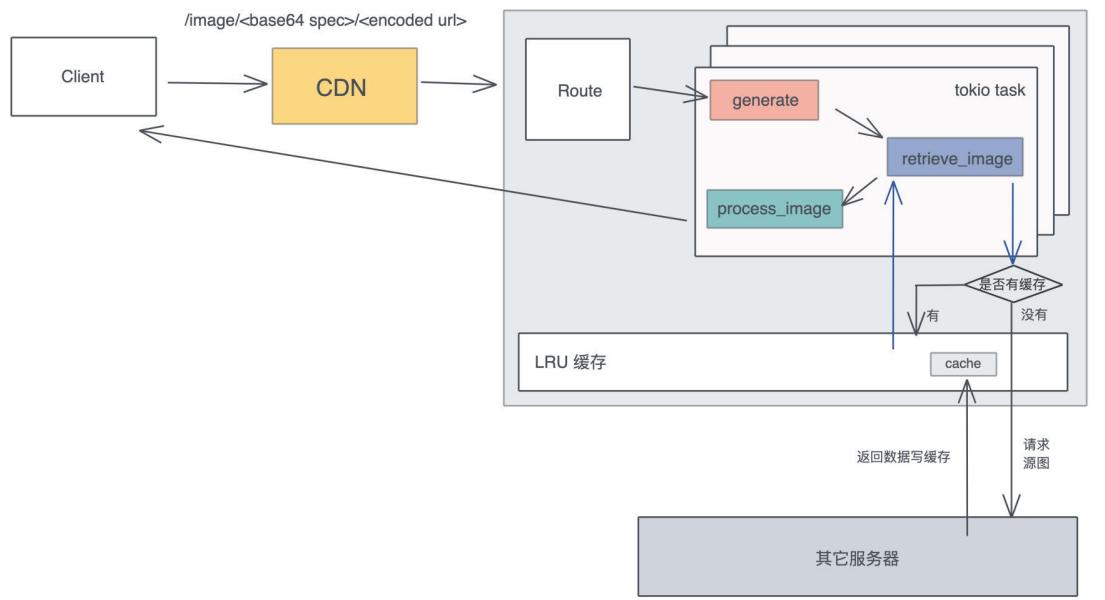
1533*2250

成功了！这就是我们的 Thumbor 服务根据用户的请求缩小到 500x800、加了水印和 Marine 滤镜后的效果。

从日志看，第一次请求时因为没有缓存，需要请求源图，所以总共花了 400ms；如果你再刷新一下，后续对同一图片的请求，会命中缓存，花了大概 200ms。

```
Aug 25 15:09:28.035 INFO thumbor: Listening on 127.0.0.1:3000
Aug 25 15:09:30.523 INFO retrieve_image{url="<https://images.pexels.com/photos/1562477/pexels-photo-1562477.jpeg?auto=compress&cs=tinysrgb&dpr=3&h=750&w=1260>"}: thumbor: Retrieve url
Aug 25 15:09:30.950 INFO thumbor: Finished processing: image size 52674
Aug 25 15:09:35.037 INFO retrieve_image{url="<https://images.pexels.com/photos/1562477/pexels-photo-1562477.jpeg?auto=compress&cs=tinysrgb&dpr=3&h=750&w=1260>"}: thumbor: Match cache
13782279907884137652
Aug 25 15:09:35.254 INFO thumbor: Finished processing: image size 52674
```

这个版本目前是一个没有详细优化过的版本，性能已经足够好。而且，像 Thumbror 这样的图片服务，前面还有 CDN (Content Distribution Network) 扛压力，只有 CDN 需要回源时，才会访问到，所以也可以不用太优化。



最后来看看目标完成得如何。如果不算 protobuf 生成的代码，Thumbor 这个项目，到目前为止我们写了 324 行代码：

```
&nbsp; &nbsp; &nbsp; 394&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; 324&nbsp; &nbsp; &nbsp; &nbsp;  
-----  
&nbsp;Total&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; 4&nbsp; &nbsp;  
&nbsp; &nbsp; &nbsp; 394&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; 324&nbsp; &nbsp; &nbsp; &nbsp;  
-----  
-----
```

三百多行代码就把一个图片服务器的核心部分搞定了，不仅如此，还充分考虑到了架构的可扩展性，用 trait 实现了主要的图片处理流程，并且引入了缓存来避免不必要的网络请求。虽然比我们预期的 200 行代码多了 50% 的代码量，但我相信它进一步佐证了 Rust 强大的表达能力。

而且，通过合理使用 protobuf 定义接口和使用 trait 做图片引擎，未来添加新的功能非常简单，可以像搭积木一样垒上去，不会影响已有的功能，完全符合开闭原则（Open-Closed Principle）。

作为一门系统级语言，Rust 使用独特的内存管理方案，零成本地帮我们管理内存；作为一门高级语言，Rust 提供了足够强大的类型系统和足够完善的标准库，帮我们很容易写出低耦合、高内聚的代码。

小结

今天讲的 Thumbar 要比上一讲的 HTTPPie 难度高一个数量级（完整代码在 [GitHub repo](#)），所以细节理解不了不打紧，但我相信你会进一步被 Rust 强大的表现力、抽象能力和解决实际问题的能力折服。

比如说，我们通过 Engine trait 分离了具体的图片处理引擎和主流程，让主流程变得干净清爽；同时在处理 protobuf 生成的数据结构时，大量使用了 From / TryFrom trait 做数据类型的转换，也是一种解耦（关注点分离）的思路。

听我讲得这么流畅，你是不是觉得我写的时候肯定不会犯错。其实并没有，我在用 axum 写源图获取的流程时，就因为使用 Mutex 的错误而被编译器毒打，花了些时间才解决。

但这种毒打是非常让人心悦诚服且快乐的，因为我知道，这样的并发问题一旦泄露到生产环境，解决起来大概率会毫无头绪，只能一点点试错可能有问题的代码，那个时候代价就远非和编译器搏斗的这十来分钟可比了。

所以只要你入了门，写 Rust 代码的过程绝对是一种享受，绝大多数错误在编译时就被揪出来了，你的代码只要编译能通过，基本上不需要担心它运行时的正确性。

也正是因为这样，在前期学习 Rust 的时候编译很难通过，导致我们直观感觉它是一门难学的语言，但其实它又很容易上手。这听起来矛盾，但确实是我自己的感受：它之所以学起来有些费力，有点像讲拉丁语系的人学习中文一样，要打破很多自己原有的认知，去拥抱新的思想和概念。但是只要多写多思考，时间长了，理解起来就是水到渠成的事。

思考题

之前提到通过合理使用 protobuf 定义接口和使用 trait 做图片引擎，未来添加新的功能非常简单。如果你学有余力，可以自己尝试一下。

我们看如何添加新功能：

- 首先添加新的 proto，定义新的 spec
- 然后为 spec 实现 SpecTransform trait 和一些辅助函数
- 最后在 Engine 中使用 spec

如果要换图片引擎呢？也很简单：

- 添加新的图片引擎，像 Photon 那样，实现 Engine trait 以及为每种 spec 实现 SpecTransform Trait。
- 在 main.rs 里使用新的引擎。

欢迎在留言区分享你的思考，如果你觉得有收获，也欢迎你分享给你身边的朋友，邀他一起挑战。你的 Rust 学习第五次打卡成功，我们下一讲见！

06 | get hands dirty：SQL查询工具怎么一鱼多吃？

06 | get hands dirty：SQL查询工具怎么一鱼多吃？

你好，我是陈天。

通过 HTTPie 和 Thumbror 的例子，相信你对 Rust 的能力和代码风格有了比较直观的了解。之前我们说过Rust的应用范围非常广，但是这两个例子体现得还不是太明显。

有同学想看看，在实际工作中有大量生命周期标注的代码的样子；有同学对 Rust 的宏好奇；有同学对 Rust 和其它语言的互操作感兴趣；还有同学想知道 Rust 做客户端的感觉。所以，我们今天就来用一个很硬核的例子把这些内容都涵盖进来。

话不多说，我们直接开始。

SQL

我们工作的时候经常会跟各种数据源打交道，数据源包括数据库、Parquet、CSV、JSON 等，而打交道的过程无非是：数据的获取（fetch）、过滤（filter）、投影（projection）和排序（sort）。

做大数据的同学可以用类似 Spark SQL 的工具来完成各种异质数据的查询，但是我们平时用 SQL 并没有这么强大。因为虽然用 SQL 对数据库做查询，任何 DBMS 都支持，如果想用 SQL 查询 CSV 或者 JSON，就需要很多额外的处理。

所以如果能有一个简单的工具，**不需要引入 Spark，就能支持对任何数据源使用 SQL 查询**，是不是很有意义？

比如，如果你的 shell 支持这样使用是不是爽爆了？

> select * from ps where status = 'Running' order by mem DESC limit 4;						
#	pid	name	status	cpu	mem	virtual
0	32828	mdworker_shared	Running	0.0000	23.0 MB	4.4 GB
1	32841	mdworker_shared	Running	0.0000	19.3 MB	4.4 GB
2	32829	CoreServicesUIAg	Running	0.0000	16.1 MB	4.5 GB
3	33155	nu_plugin_core_p	Running	3.8496	11.8 MB	4.4 GB

再比如，我们的客户端会从服务器 API 获取数据的子集，如果这个子集可以在前端通过 SQL 直接做一些额外查询，那将非常灵活，并且用户可以得到即时的响应。

软件领域有个著名的[格林斯潘第十定律](#)：

任何 C 或 Fortran 程序复杂到一定程度之后，都会包含一个临时开发的、不合规范的、充满程序错误的、运行速度很慢的、只有一半功能的 Common Lisp 实现。

我们仿照它来一个程序君第四十二定律：

任何 API 接口复杂到一定程度后，都会包含一个临时开发的、不合规范的、充满程序错误的、运行速度很慢的、只有一半功能的 SQL 实现。

所以，我们今天就来设计一个可以对任何数据源使用 SQL 查询，并获得结果的库如何？当然，作为一个 MVP（Mimimum Viable Product），我们就暂且只支持对 CSV 的 SQL 查询。不单如此，我们还希望这个库可以给 Python3 和 Node.js 使用。

猜一猜这个库要花多少行代码？今天难度比较大，怎么着要 500 行吧？我们暂且以 500 行代码为基准来挑战。

设计分析

我们首先需要一个 SQL 解析器。在 Rust 下，写一个解析器并不困难，可以用 [serde](#)、用任何 [parser combinator](#) 或者 [PEG parser](#) 来实现，比如 [nom](#) 或者 [pest](#)。不过 SQL 解析，这种足够常见的需求，Rust 社区已经有方案，我们用 [sqlparser-rs](#)。

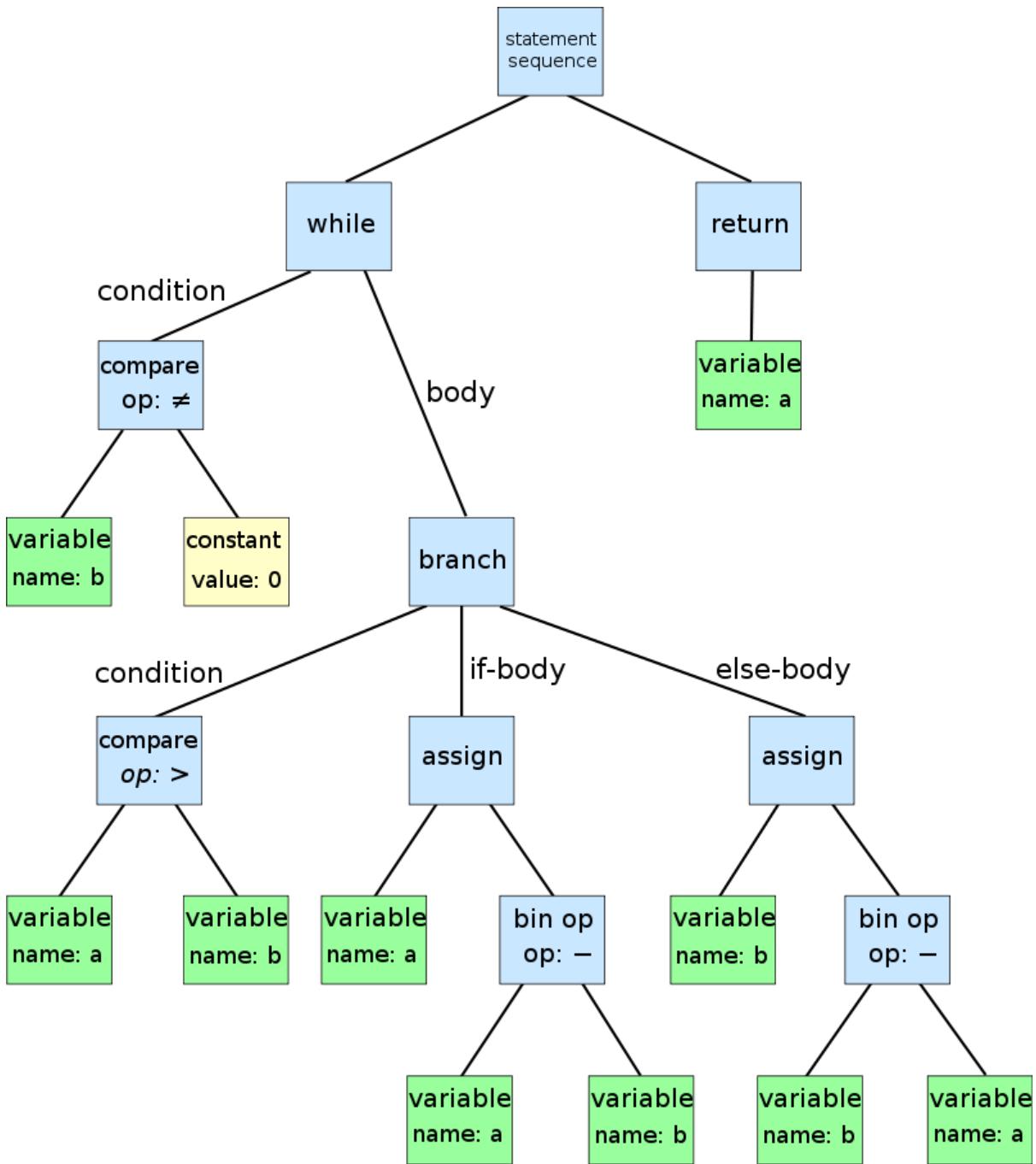
接下来就是如何把 CSV 或者其它数据源加载为 DataFrame。

做过数据处理或者使用过 [pandas](#) 的同学，应该对 DataFrame 并不陌生，它是一个矩阵数据结构，其中每一列可能包含不同的类型，可以在 DataFrame 上做过滤、投影和排序等操作。

在 Rust 下，我们可以用 [polars](#)，来完成数据从 CSV 到 DataFrame 的加载和各种后续操作。

确定了这两个库之后，后续的工作就是：如何把 sqlparser 解析出来的抽象语法树 [AST](#)（Abstract Syntax Tree），映射到 polars 的 DataFrame 的操作上。

抽象语法树是用来描述复杂语法规则的工具，小到 SQL 或者某个 DSL，大到一门编程语言，其语言结构都可以通过 AST 来描述，如下图所示（来源：[wikipedia](#)）：

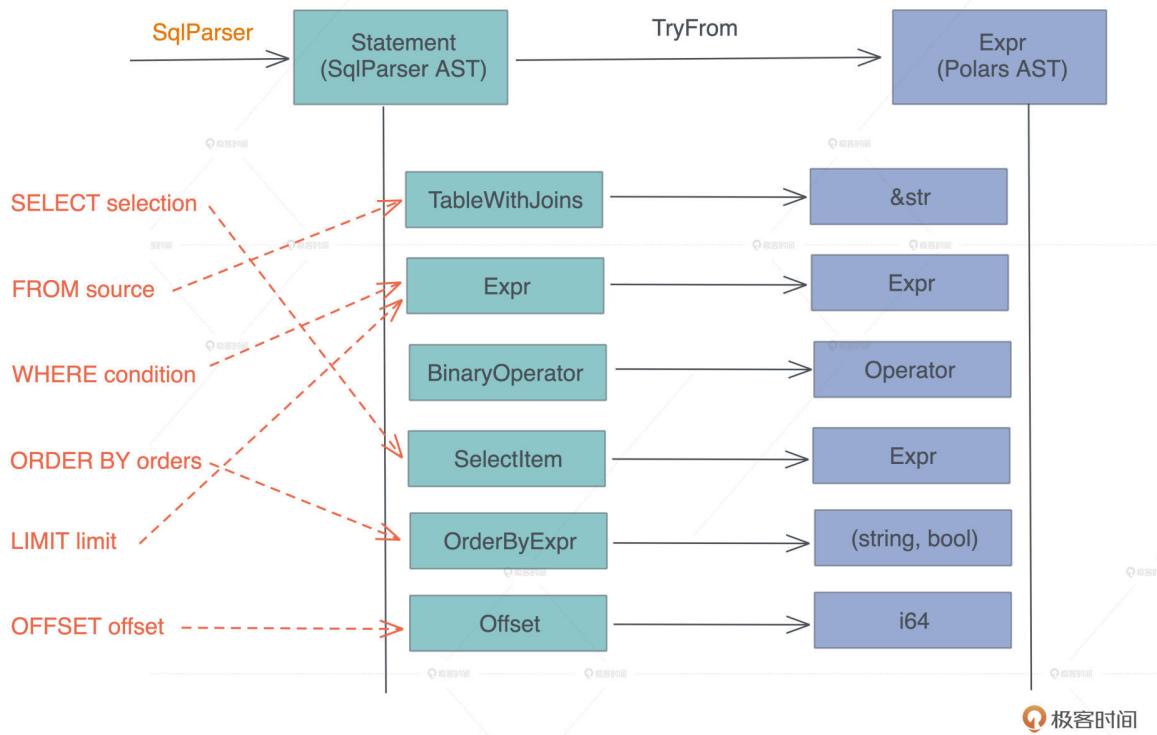


如何在 SQL 语法和 DataFrame 的操作间进行映射呢？比如我们要从数据中选出三列显示，那这个“select a, b, c”就要能映射到 DataFrame 选取 a、b、c 三列输出。

polars 内部有自己的 AST 可以把各种操作聚合起来，最后一并执行。比如对于“where a > 10 and b < 5”，Polars 的表达式是：`col("a").gt(lit(10)).and(col("b").lt(lit(5)))`。`col` 代表列，`gt/lt` 是大于/小于，`lit` 是字面量的意思。

有了这个认知，“对 CSV 等源进行 SQL 查询”核心要解决的问题变成了，**如何把一个 AST (SQL AST) 转换成另一个 AST (DataFrame AST)**。

等等，这不就是宏编程（对于 Rust 来说，是过程宏）做的事情么？因为进一步分析二者的数据结构，我们可以得到这样的对应关系：



你看，我们要做的主要事情其实就是，在两个数据结构之间进行转换。所以，写完今天的代码，你肯定会对宏有足够的信心。

宏编程并没有什么大不了的，抛开 quote/unquote，它主要的工作就是把一棵语法树转换成另一棵语法树，而这个转换的过程深入下去，不过就是数据结构到数据结构的转换而已。所以一句话总结：**宏编程的主要流程就是实现若干 From 和 TryFrom

**，是不是很简单。

当然，这个转换的过程非常琐碎，如果语言本身没有很好的模式匹配能力，进行宏编程绝对是对自己非人道的折磨。

好在 Rust 有很棒的模式匹配支持，它虽然没有 Erlang/Elixir 的模式匹配那么强大，但足以秒杀绝大多数的编程语言。待会你在写的时候，能直观感受到。

创建一个 SQL 方言

好，分析完要做的事情，接下来就是按部就班写代码了。

我们用 cargo new queryer -lib 生成一个库。用 VSCode 打开生成的目录，创建和 src 平级的 examples，并在 Cargo.toml 中添加代码：

```
[ [example]]
name = "dialect"

[dependencies]
anyhow = "1" # thiserror
async-trait = "0.1" # trait async fn
sqlparser = "0.10" # SQL
polars = { version = "0.15", features = ["json", "lazy"] } # DataFrame
reqwest = { version = "0.11", default-features = false, features = ["rustls-tls"] } # HTTP
tokio = { version = "1", features = ["fs"] } #
tracing = "0.1" #

[dev-dependencies]
tracing-subscriber = "0.2" #
tokio = { version = "1", features = ["full"] } # example tokio feature
```

依赖搞定。因为对 sqlparser 的功能不太熟悉，这里写个 example 尝试一下，它会在 examples 目录下寻找 [dialect.rs](#) 文件。

所以，我们创建 examples/dialect.rs 文件，并写一些测试 sqlparser 的代码：

```
use sqlparser::{dialect::GenericDialect, parser::Parser};

fn main() {
    tracing_subscriber::fmt::init();

    let sql = "SELECT a a1, b, 123, myfunc(b), * \
    FROM data_source \
    WHERE a > b AND b < 100 AND c BETWEEN 10 AND 20 \
    ORDER BY a DESC, b \
    LIMIT 50 OFFSET 10";

    let ast = Parser::parse_sql(&GenericDialect::default(), sql);
    println!("{:?}", ast);
}
```

这段代码用一个 SQL 语句来测试 Parser::parse_sql 会输出什么样的结构。当你写库代码时，如果遇到不明白的第三方库，可以用撰写 example 这种方式先试一下。

我们运行 cargo run -example dialect 查看结果：

```
Ok([Query(  
    Query {  
        with: None,  
        body: Select(  
            Select {  
                distinct: false,  
                top: None,  
                projection: [ ... ],  
                from: [ TableWithJoins { ... } ]  
            ),  
            selection: Some(BinaryOp { ... })  
        ),  
        ...  
    },  
    order_by: [ OrderByExpr { ... } ],  
    limit: Some(Value( ... )),  
    offset: Some(Offset { ... })  
}  
])
```

我把这个结构简化了一下，你在命令行里看到的，会远比这个复杂。

写到第9行这里，你有没有突发奇想，如果 SQL 中的 FROM 子句后面可以接一个 URL 或者文件名该多好？这样，我们可以从这个 URL 或文件中读取数据。就像开头那个“select * from ps”的例子，把 ps 命令作为数据源，从它的输出中很方便地取数据。

但是普通的 SQL 语句是不支持这种写法的，不过 sqlparser 允许你创建自己的 SQL 方言，那我们就来尝试一下。

创建 src/dialect.rs 文件，添入下面的代码：

```
use sqlparser::dialect::Dialect;  
  
#[derive(Debug, Default)]  
pub struct TyrDialect;  
  
// sql TyrDialect identifier url  
impl Dialect for TyrDialect {  
    fn is_identifier_start(&self, ch: char) -> bool {  
        ('a'..='z').contains(&ch) || ('A'..='Z').contains(&ch) || ch == '_'  
    }  
  
    // identifier ':', '/', '?', '&', '='  
    fn is_identifier_part(&self, ch: char) -> bool {  
        ('a'..='z').contains(&ch)
```

```

    || ('A'..'Z').contains(&ch)
    || ('0'..'9').contains(&ch)
    || [':', '/', '?', '&', '=', '-', '_', '.'].contains(&ch)
}
}

/// 
pub fn example_sql() -> String {
    let url = "https://raw.githubusercontent.com/owid/covid-19-data/master/public/data/latest
/owid-covid-latest.csv";

    let sql = format!(
        "SELECT location, name, total_cases, new_cases, total_deaths, new_deaths \
        FROM {} WHERE new_deaths >= 500 ORDER BY new_cases DESC LIMIT 6 OFFSET 5",
        url
    );
    sql
}

#[cfg(test)]
mod tests {
    use super::*;

    use sqlparser::parser::Parser;

    #[test]
    fn it_works() {
        assert!(Parser::parse_sql(&TyrDialect::default(), &example_sql()).is_ok());
    }
}

```

这个代码主要实现了 `sqlparser` 的 `Dialect` trait，可以重载 SQL 解析器判断标识符的方法。之后我们需要在 `src/lib.rs` 中添加

```
mod dialect;
```

引入这个文件，最后也写了一个测试，你可以运行 `cargo test` 测试一下看看。

测试通过！现在我们可以正常解析出这样的 SQL 了：

```
SELECT * from https://abc.xyz/covid-cases.csv WHERE new_deaths >= 500
```

Cool! 你看，大约用了 10 行代码（第 7 行到第 19 行），通过添加可以让 URL 合法的字符，就实现了一个自己的支持 URL 的 SQL 方言解析。

为什么这么厉害？因为通过 trait，你可以很方便地做[控制反转（Inversion of Control）](#)，在 Rust 开发中，这是很常见的一件事情。

实现 AST 的转换

刚刚完成了SQL解析，接着就是用polars做AST转换了。

由于我们不太了解 polars 库，接下来还是先测试一下怎么用。创建 examples/covid.rs（记得在 Cargo.toml 中添加它哦），手工实现一个 DataFrame 的加载和查询：

```
use anyhow::Result;
use polars::prelude::*;
use std::io::Cursor;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();

    let url = "https://raw.githubusercontent.com/owid/covid-19-data/master/public/data/latest
/owid-covid-latest.csv";
    let data = reqwest::get(url).await?.text().await?;

    // polars
    let df = CsvReader::new(Cursor::new(data))
        .infer_schema(Some(16))
        .finish()?;
}

let filtered = df.filter(&df["new_deaths"].gt(500))?;
println!(
    "{:?}", 
    filtered.select((
        "location",
        "total_cases",
        "new_cases",
        "total_deaths",
        "new_deaths"
    )));
Ok(())
}
```

如果我们运行这个 example，可以得到一个打印得非常漂亮的表格，它从 GitHub 上的 [owid-covid-latest.csv](#) 文件中，读取并查询 new_deaths 大于 500 的国家和区域：

```

> cargo run --example covid --quiet
Ok(shape: (13, 5)
+-----+-----+-----+-----+-----+
| location | total_cases | new_cases | total_deaths | new_deaths |
| ---      | ---          | ---          | ---          | ---          |
| str      | f64          | f64          | f64          | f64          |
+-----+-----+-----+-----+-----+
| "Africa" | 7.695475e6  | 3.3957e4   | 1.93394e5   | 764         |
+-----+-----+-----+-----+-----+
| "Asia"   | 6.9135616e7 | 2.71675e5  | 1.023341e6  | 4001        |
+-----+-----+-----+-----+-----+
| "Brazil" | 2.0703906e7 | 2.7345e4   | 5.78326e5   | 761         |
+-----+-----+-----+-----+-----+
| "Europe" | 5.5248221e7 | 1.16521e5  | 1.170825e6  | 1440        |
+-----+-----+-----+-----+-----+
| ...     | ...          | ...          | ...          | ...          |
+-----+-----+-----+-----+-----+
| "North America" | 4.6482777e7 | 3.63748e5  | 9.70161e5   | 4254        |
+-----+-----+-----+-----+-----+
| "Russia"    | 6.747681e6   | 1.8982e4   | 1.76904e5   | 777         |
+-----+-----+-----+-----+-----+
| "South America" | 3.6768062e7 | 3.3853e4   | 1.126593e6  | 1019        |
+-----+-----+-----+-----+-----+
| "United States" | 3.8707294e7 | 3.22934e5  | 6.3672e5    | 3156        |
+-----+-----+-----+-----+-----+
| "World"     | 2.15448179e8 | 8.21174e5  | 4.48602e6   | 1.1491e4   |
+-----+-----+-----+-----+-----+
)

```

我们最终要实现的就是这个效果，通过解析一条做类似查询的 SQL，来进行相同的数据查询。怎么做呢？

今天一开始已经分析过了，主要的工作就是把 `sqlparser` 解析出来的 AST 转换成 `polars` 定义的 AST。再回顾一下 SQL AST 的输出：

```

Ok([Query(
  Query {
    with: None,
    body: Select(
      Select {
        distinct: false,
        top: None,
        projection: [ ... ],
        from: [ TableWithJoins { ... } ]
      },
      selection: Some(BinaryOp { ... })
    ),
    ...
  }
),
  order_by: [ OrderByExpr { ... } ],
  limit: Some(Value( ... )),
```

```

        offset: Some(Offset { ... })
    }
])

```

这里的 Query 是 Statement enum 其中一个结构。SQL 语句除了查询外，还有插入数据、删除数据、创建表等其他语句，我们今天不关心这些，只关心 Query。

所以，可以创建一个文件 src/convert.rs，**先定义一个数据结构 Sql 来描述两者的对应关系，然后再实现 Sql 的 TryFrom trait

**:

```

/// SQL
pub struct Sql<'a> {
    pub(crate) selection: Vec<Expr>,
    pub(crate) condition: Option<Expr>,
    pub(crate) source: &'a str,
    pub(crate) order_by: Vec<(String, bool)>,
    pub(crate) offset: Option<i64>,
    pub(crate) limit: Option<usize>,
}

impl<'a> TryFrom<&'a Statement> for Sql<'a> {
    type Error = anyhow::Error;
    fn try_from(sql: &'a Statement) -> Result<Self, Self::Error> {
        match sql {
            // query (select ... from ... where ...)
            Statement::Query(q) => {
                ...
            }
        }
    }
}

```

框有了，继续写转换。我们看 Query 的结构：它有一个 body，是 Select 类型，其中包含 projection、from、select。在 Rust 里，我们可以用一个赋值语句，同时使用模式匹配加上数据的解构，将它们都取出来：

```

let Select {
    from: table_with_joins,
    selection: where_clause,
    projection,
    group_by: _,
    ..
} = match &q.body {

```

```
SetExpr::Select(statement) => statement.as_ref(),
_ => return Err(anyhow!("We only support Select Query at the moment")),
};


```

一句话，从匹配到取引用，再到将引用内部几个字段赋值给几个变量，都完成了，真是太舒服了！这样能够极大提高生产力的语言，你怎能不爱它？

我们再看一个处理 Offset 的例子，需要把 sqlparser 的 Offset 转换成 i64，同样，可以实现一个 TryFrom trait。这次是在 match 的一个分支上，做了数据结构的解构。

```
use sqlparser::ast::Offset as SqlOffset;

// Rust trait trait
//

pub struct Offset<'a>(pub(crate) &'a SqlOffset);

/// SqlParser offset expr i64
impl<'a> From<Offset<'a>> for i64 {
    fn from(offset: Offset) -> Self {
        match offset.0 {
            SqlOffset {
                value: SqlExpr::Value(SqlValue::Number(v, _b)),
                ..
            } => v.parse().unwrap_or(0),
            _ => 0,
        }
    }
}
```

是的，数据的解构也可以在分支上进行，如果你还记得第三讲中谈到的 if let / while let，也是这个用法。这样对模式匹配的全方位支持，你用得越多，就会越感激 Rust 的作者，尤其在开发过程宏的时候。

从这段代码中还可以看到，定义的数据结构 Offset 使用了生命周期标注 `<'a>`，这是因为内部使用了 `SqlOffset` 的引用。有关生命周期的知识，我们很快就会讲到，这里你暂且不需要理解为什么要这么做。

整个 `src/convert.rs` 主要都是通过模式匹配，进行不同子类型之间的转换，代码比较无趣，而且和上面的代码类似，我就不贴了，你可以在这门课程的 [GitHub repo](#) 下的 `06_queryer/queryer/src/convert.rs` 中获取。

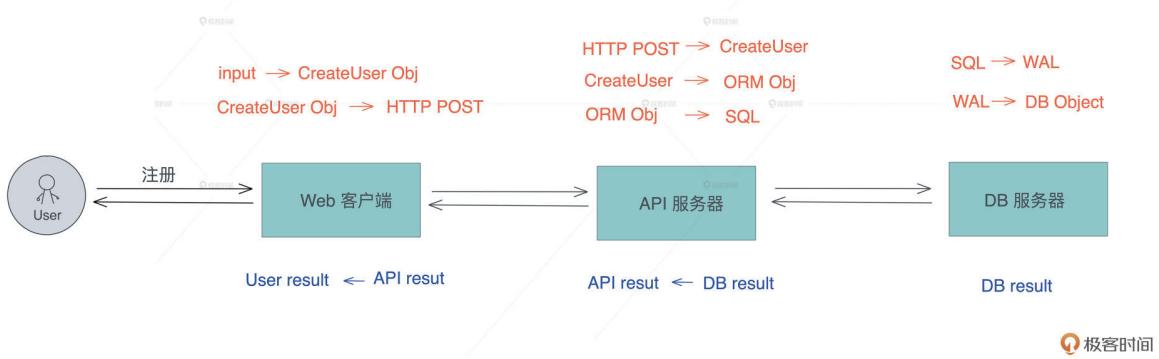
未来你在 Rust 下写过程宏（procedure macro），干的基本就是这个工作，只不过，最后你需要把转换后的 AST 使用 `quote` 输出成代码。在这个例子里，我们不需要这么做，polars 的 `lazy` 接口直接能处理 AST。

说句题外话，我之所以不厌其烦地讲述数据转换的这个过程，是因为它是我们编程活动中非常重要的部分。你想想，我们写代码，主要都在处理什么？**绝大多数处理逻辑都是把数据从一个接口转换成另一个接口。**

以我们熟悉的用户注册流程为例：

1. 用户的输入被前端校验后，转换成 CreateUser 对象，然后再转换成一个 HTTP POST 请求。
2. 当这个请求到达服务器后，服务器将其读取，再转换成服务器的 CreateUser 对象，这个对象在校验和正规化（normalization）后被转成一个 ORM 对象（如果使用 ORM 的话），然后 ORM 对象再被转换成 SQL，发送给数据库服务器。
3. 数据库服务器将 SQL 请求包装成一个 WAL（Write-Ahead Logging），这个 WAL 再被更新到数据库文件中。

整个数据转换过程如下图所示：



这样的处理流程，由于它和业务高度绑定，往往容易被写得很耦合，久而久之就变成了难以维护的意大利面条。**好的代码，应该是每个主流程都清晰简约，代码恰到好处地出现在那里，让人不需要注释也能明白作者在写什么。**

这就意味着，我们要把那些并不重要的细节封装在单独的地方，封装的粒度以一次写完、基本不需要再变动为最佳，或者即使变动，它的影响也非常局部。

这样的代码，方便阅读、容易测试、维护简单，处理起来更是一种享受。Rust 标准库的 From / TryFrom trait，就是出于这个目的设计的，非常值得我们好好使用。

从源中取数据

完成了 AST 的转换，接下来就是从源中获取数据。

我们通过对 Sql 结构的处理和填充，可以得到 SQL FROM 子句里的数据源，这个源，我们规定它必须是以 http(s):// 或者 file:// 开头的字符串。因为，以 http 开头我们可以通过 URL 获取内容，file 开头我们可以通过文件名，打开本地文件获取内容。

所以拿到了这个描述了数据源的字符串后，很容易能写出这样的代码：

```
/// http
async fn retrieve_data(source: impl AsRef<str>) -> Result<String> {
    let name = source.as_ref();
    match &name[..4] {
        // http / https
        "http" => Ok(reqwest::get(name).await?.text().await?),
        // file://<filename>
        "file" => Ok(fs::read_to_string(&name[7..]).await?),
        _ => Err(anyhow!("We only support http/https/file at the moment")),
    }
}
```

代码看起来很简单，但未来并不容易维护。因为一旦你的 HTTP 请求获得的结果需要做一些后续的处理，这个函数很快就会变得很复杂。那该怎么办呢？

如果你回顾前两讲我们写的代码，相信你心里马上有了答案：**可以用 trait 抽取 fetch 的逻辑，定义好接口，然后改变 retrieve_data 的实现。**

所以下面是 src/fetcher.rs 的完整代码：

```
use anyhow::{anyhow, Result};
use async_trait::async_trait;
use tokio::fs;

// Rust async trait async_trait
#[async_trait]
pub trait Fetch {
    type Error;
    async fn fetch(&self) -> Result<String, Self::Error>;
}

/// http data frame
pub async fn retrieve_data(source: impl AsRef<str>) -> Result<String> {
    let name = source.as_ref();
    match &name[..4] {
        // http / https
        "http" => UrlFetcher(name).fetch().await,
        // file://<filename>
        "file" => FileFetcher(name).fetch().await,
        _ => return Err(anyhow!("We only support http/https/file at the moment")),
    }
}

struct UrlFetcher<'a>(pub(crate) &'a str);
struct FileFetcher<'a>(pub(crate) &'a str);

#[async_trait]
```

```

impl<'a> Fetch for UrlFetcher<'a> {
    type Error = anyhow::Error;

    async fn fetch(&self) -> Result<String, Self::Error> {
        Ok(request::get(self.0).await?.text().await?)
    }
}

#[async_trait]
impl<'a> Fetch for FileFetcher<'a> {
    type Error = anyhow::Error;

    async fn fetch(&self) -> Result<String, Self::Error> {
        Ok(fs::read_to_string(&self.0[7..]).await?)
    }
}

```

这看上去似乎没有收益，还让代码变得更多。但它把 `retrieve_data` 和具体每一种类型的处理分离了，还是我们之前讲的思想，通过开闭原则，构建低耦合、高内聚的代码。这样未来我们修改 `UrlFetcher` 或者 `FileFetcher`，或者添加新的 `Fetcher`，对 `retrieve_data` 的变动都是最小的。

现在我们完成了SQL的解析、实现了从SQL到DataFrame的AST的转换，以及数据源的获取。挑战已经完成一大半了，就剩主流程逻辑了。

主流程

一般我们在做一个库的时候，不会把内部使用的数据结构暴露出去，而是会用自己的数据结构包裹它。

但这样代码有一个问题：**原有数据结构的方法，如果我们想暴露出去，每个接口都需要实现一遍**，虽然里面的代码就是一句简单的 proxy，但还是很麻烦。这是我自己的使用很多语言的一个痛点。

正好在 `queryer` 库里也会有这个问题：SQL 查询后的结果，会放在一个 polars 的 DataFrame 中，但我们不想直接暴露这个 DataFrame 出去。因为一旦这么做，未来我们想加额外的 metadata，就无能为力了。

所以我定义了一个 `DataSet`，包裹住 DataFrame。可是，我还想暴露 `DataSet` 的接口，它有好多函数，总不能挨个 proxy 吧？

不用。Rust 提供了 `Deref` 和 `DerefMut` trait 做这个事情，它允许类型在解引用时，可以解引用到其它类型。我们后面在介绍 Rust 常用 trait 时，会详细介绍这两个 trait，现在先来看的 `DataSet` 怎么处理：

```

#[derive(Debug)]
pub struct DataSet(DataFrame);

```

```

///  DataSet  DataFrame
impl Deref for DataSet {
    type Target = DataFrame;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

///  DataSet  DataFrame
impl DerefMut for DataSet {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

// DataSet
impl DataSet {
    ///  DataSet  csv
    pub fn to_csv(&self) -> Result<String> {
        ...
    }
}

```

可以看到，DataSet 在解引用时，它的 Target 是 DataFrame，这样 DataSet 在用户使用时，就和 DataFrame 一致了；我们还为 DataSet 实现了 to_csv 方法，可以把查询结果生成出 CSV。

好，定义好 DataSet，核心函数 query 实现起来其实很简单：先解析出我们要的 Sql 结构，然后从 source 中读入一个 DataSet，做 filter / order_by / offset / limit / select 等操作，最后返回 DataSet。

DataSet 的定义和 query 函数都在 src/lib.rs，它的完整代码如下：

```

use anyhow::{anyhow, Result};
use polars::prelude::*;
use sqlparser::parser::Parser;
use std::convert::TryInto;
use std::ops::{Deref, DerefMut};
use tracing::info;

mod convert;
mod dialect;
mod loader;
mod fetcher;
use convert::Sql;
use loader::detect_content;
use fetcher::retrieve_data;

pub use dialect::example_sql;
pub use dialect::TyrDialect;

```

```

#[derive(Debug)]
pub struct DataSet(DataFrame);

///  DataSet  DataFrame
impl Deref for DataSet {
    type Target = DataFrame;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

///  DataSet  DataFrame
impl DerefMut for DataSet {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

impl DataSet {
    ///  DataSet  csv
    pub fn to_csv(&self) -> Result<String> {
        let mut buf = Vec::new();
        let writer = CsvWriter::new(&mut buf);
        writer.finish(self)?;
        Ok(String::from_utf8(buf)?)
    }
}

///  from  where
pub async fn query<T: AsRef<str>>(sql: T) -> Result<DataSet> {
    let ast = Parser::parse_sql(&TyrDialect::default(), sql.as_ref())?;

    if ast.len() != 1 {
        return Err(anyhow!("Only support single sql at the moment"));
    }

    let sql = &ast[0];

    //  SQL AST  Sql  try_into()
    //
    //
    let Sql {
        source,
        condition,
        selection,
        offset,
        limit,
        order_by,
    } = sql.try_into()?;
    info!("retrieving data from source: {}", source);

    //  source  DataSet
    //  detect_content  detect  DataSet
    let ds = detect_content(retrieve_data(source).await?).load()?;
    let mut filtered = match condition {

```

```

        Some(expr) => ds.0.lazy().filter(expr),
        None => ds.0.lazy(),
    } ;

    filtered = order_by
        .into_iter()
        .fold(filtered, |acc, (col, desc)| acc.sort(&col, desc));

    if offset.is_some() || limit.is_some() {
        filtered = filtered.slice(offset.unwrap_or(0), limit.unwrap_or(usize::MAX));
    }

    Ok(DataSet(filtered.select(selection).collect()?))
}

```

在 query 函数的主流程中，整个 SQL AST 转换成了我们定义的 Sql 结构，细节都埋藏在 try_into() 中，我们只需关注数据结构 Sql 的使用，怎么转换之后需要的时候再关注。

这就是[关注点分离（Separation of Concerns）](#)，是我们控制软件复杂度的法宝。Rust 标准库中那些经过千锤百炼的 trait，就是用来帮助我们写出更好的、复杂度更低的代码。

主流程里有个 detect_content 函数，它可以识别文本内容，选择相应的加载器把文本加载为 DataSet，因为目前只支持 CSV，但未来可以支持 JSON 等其他格式。这个函数定义在 src/loader.rs 里，我们创建这个文件，并添入下面的代码：

```

use crate::DataSet;
use anyhow::Result;
use polars::prelude::*;
use std::io::Cursor;

pub trait Load {
    type Error;
    fn load(self) -> Result<DataSet, Self::Error>;
}

#[derive(Debug)]
#[non_exhaustive]
pub enum Loader {
    Csv(CsvLoader),
}

#[derive(Default, Debug)]
pub struct CsvLoader(pub(crate) String);

impl Loader {
    pub fn load(self) -> Result<DataSet> {
        match self {
            Loader::Csv(csv) => csv.load(),
        }
    }
}

```

```

pub fn detect_content(data: String) -> Loader {
    // TODO:
    Loader::Csv(CsvLoader(data))
}

impl Load for CsvLoader {
    type Error = anyhow::Error;

    fn load(self) -> Result<DataSet, Self::Error> {
        let df = CsvReader::new(Cursor::new(self.0))
            .infer_schema(Some(16))
            .finish()?;
        Ok(DataSet(df))
    }
}

```

同样，通过 trait，我们虽然目前只支持 CsvLoader，但保留了为未来添加更多 Loader 的接口。

好，现在这个库就全部写完了，尝试编译一下。如果遇到了问题，不要着急，可以在这门课的 [GitHub repo](#) 里获取完整的代码，然后对应修改你本地的错误。

如果代码编译通过了，你可以修改之前的 examples/covid.rs，使用 SQL 来查询测试一下：

```

use anyhow::Result;
use queryer::query;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();

    let url = "https://raw.githubusercontent.com/owid/covid-19-data/master/public/data/latest
/owid-covid-latest.csv";

    // sql URL
    let sql = format!(
        "SELECT location_name, total_cases, new_cases, total_deaths, new_deaths \
        FROM {} WHERE new_deaths >= 500 ORDER BY new_cases DESC",
        url
    );
    let df1 = query(sql).await?;
    println!("{:?}", df1);

    Ok(())
}

```

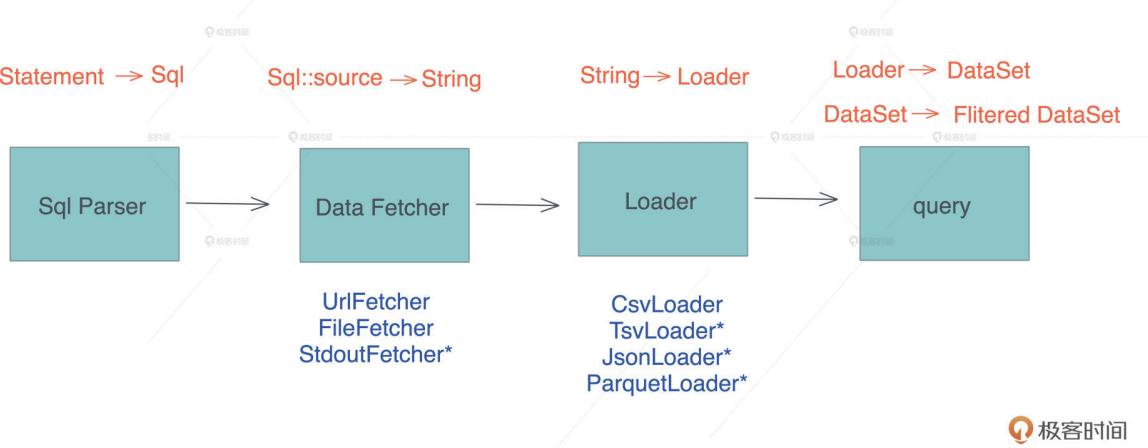
Bingo! 一切正常，我们完成了，用 SQL 语句请求网络上的某个 CSV，并对 CSV 做查询和排序，返回结果的正确无误！

```
> cargo run --example covid --quiet
DataSet(shape: (13, 5)
+-----+-----+-----+-----+-----+
| name      | total_cases | new_cases | total_deaths | new_deaths |
| ---       | ---         | ---        | ---          | ---        |
| str       | f64         | f64        | f64          | f64        |
+=====+=====+=====+=====+=====+
| "World"   | 2.15448179e8 | 8.21174e5 | 4.48602e6  | 1.1491e4  |
+-----+-----+-----+-----+-----+
| "North America" | 4.6482777e7 | 3.63748e5 | 9.70161e5 | 4254      |
+-----+-----+-----+-----+-----+
| "United States" | 3.8707294e7 | 3.22934e5 | 6.3672e5  | 3156      |
+-----+-----+-----+-----+-----+
| "Asia"     | 6.9135616e7 | 2.71675e5 | 1.023341e6 | 4001      |
+-----+-----+-----+-----+-----+
| ...        | ...         | ...        | ...         | ...        |
+-----+-----+-----+-----+-----+
| "South America" | 3.6768062e7 | 3.3853e4  | 1.126593e6 | 1019      |
+-----+-----+-----+-----+-----+
| "Brazil"   | 2.0703906e7 | 2.7345e4  | 5.78326e5  | 761       |
+-----+-----+-----+-----+-----+
| "Mexico"   | 3.311317e6  | 1.9556e4  | 2.5715e5  | 863       |
+-----+-----+-----+-----+-----+
| "Russia"   | 6.747681e6  | 1.8982e4  | 1.76904e5 | 777       |
+-----+-----+-----+-----+-----+
| "Indonesia" | 4.056354e6 | 1.2618e4  | 1.30781e5 | 599       |
+-----+-----+-----+-----+-----+
)
```

用 tokei 查看代码行数，可以看到，用了 375 行，远低于 500 行的目标！

tokei src/					
Language	Files	Lines	Code	Comments	Blanks
Rust	5	466	375	22	69
Total	5	466	375	22	69

在这么小的代码量下，我们在架构上做了很多为解耦考虑的工作：整个架构被拆成了 Sql Parser、Fetcher、Loader 和 query 四个部分。



极客时间

其中未来可能存在变化的 Fetcher 和 Loader 可以轻松扩展，比如我们一开始提到的那个 “select * from ps”，可以用一个 StdoutFetcher 和 TsvLoader 来处理。

支持其它语言

现在我们的核心代码写完了，有没有感觉自己成就感爆棚，实现的queryer工具可以在 Rust 下作为一个库，提供给其它 Rust 程序用，这很美妙。

但我们的故事还远不止如此。这么牛的功能，只能 Rust 程序员享用，太暴殄天物了。毕竟独乐乐不如众乐乐。所以，我们来试着将它集成到其它语言，比如常用的 Node.js/Python。

Node.js/Python 中有很多高性能的代码，都是 C/C++ 写的，但跨语言调用往往涉及繁杂的接口转换代码，所以用 C/C++，写这些接口转换的时候非常痛苦。

我们看看如果用 Rust 的话，能否避免这些繁文缛节？毕竟，我们对使用 Rust，为其它语言提供高性能代码，有很高的期望，如果这个过程也很复杂，那怎么用得起来？

对于 queryer 库，我们想暴露出来的主要接口是：query，用户传入一个 SQL 字符串和一个输出类型的字符串，返回一个按照 SQL 查询处理过的、符合输出类型的字符串。比如对 Python 来说，就是下面的接口：

```
def query(sql, output = 'csv')
```

好，我们来试试看。

先创建一个新的目录 queryer 作为 workspace，把现有的 queryer 移进去，成为它的子目录。然后，我们创建一个 Cargo.toml，包含以下代码：

```
[workspace]

members = [
    "queryer",
    "queryer-py"
]
```

Python

我们在 workspace 的根目录下，`cargo new queryer-py --lib`，生成一个新的 crate。在 queryer-py 下，编辑 Cargo.toml：

```
[package]
name = "queryer_py" # Python
version = "0.1.0"
edition = "2018"

[lib]
crate-type = ["cdylib"] # cdylib

[dependencies]
queryer = { path = "../queryer" } # queryer
tokio = { version = "1", features = ["full"] }

[dependencies.pyo3] # pyo3
version = "0.14"
features = ["extension-module"]

[build-dependencies]
pyo3-build-config = "0.14"
```

Rust 和 Python 交互的库是 [pyo3](#)，感兴趣你可以课后看它的文档。在 src/lib.rs 下，添入如下代码：

```
use pyo3::exceptions, prelude::*;

#[pyfunction]
pub fn example_sql() -> PyResult<String> {
    Ok(queryer::example_sql())
}

#[pyfunction]
pub fn query(sql: &str, output: Option<&str>) -> PyResult<String> {
    let rt = tokio::runtime::Runtime::new().unwrap();
```

```

let data = rt.block_on(async { queryer::query(sql).await.unwrap() });
match output {
    Some("csv") | None => Ok(data.to_csv().unwrap()),
    Some(v) => Err(exceptions::PyTypeError::new_err(format!(
        "Output type {} not supported",
        v
    )));
}
}

#[pymodule]
fn queryer_py(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(query, m)?);
    m.add_function(wrap_pyfunction!(example_sql, m)?);
    Ok(())
}

```

即使我不解释这些代码，你也基本能明白它在干嘛。我们为 Python 模块提供了两个接口 example_sql 和 query。

接下来在 queryer-py 目录下，创建 virtual env，然后用 maturin develop 构建 python 模块：

```

python3 -m venv .env
source .env/bin/activate
pip install maturin ipython
maturin develop

```

构建完成后，可以用 ipython 测试：

```

In [1]: import queryer_py

In [2]: sql = queryer_py.example_sql()

In [3]: print(queryer_py.query(sql, 'csv'))
name,total_cases,new_cases,total_deaths,new_deaths
India,32649947.0,46759.0,437370.0,509.0
Iran,4869414.0,36279.0,105287.0,571.0
Africa,7695475.0,33957.0,193394.0,764.0
South America,36768062.0,33853.0,1126593.0,1019.0
Brazil,20703906.0,27345.0,578326.0,761.0
Mexico,3311317.0,19556.0,257150.0,863.0

In [4]: print(queryer_py.query(sql, 'json'))
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-4-7082f1ffe46a> in <module>
----> 1 print(queryer_py.query(sql, 'json'))

TypeError: Output type json not supported

```

Cool! 仅仅写了 20 行代码，就让我们的模块可以被 Python 调用，错误处理也很正常。你看，在用 Rust 库的基础上，我们稍微写一些辅助代码，就能够让它和不同的语言集成起来。我觉得这是 Rust 非常有潜力的使用方向。

毕竟，对很多公司来说，原有的代码库想要完整迁移到 Rust 成本很大，但是通过 Rust 和各个语言轻便地集成，可以把部分需要高性能的代码迁移到 Rust，尝到甜头，再一点点推广。这样，Rust 就能应用起来了。

小结

回顾这周的 Rust 代码之旅，我们先做了个 HTTPie，流程简单，青铜级难度，你学完所有权，理解了基本的 trait 后就能写。

之后的 Thumbr，引入了异步、泛型和更多的 trait，白银级难度，在你学完类型系统，对异步稍有了解后，应该可以搞定。

今天的 Queryer，使用了大量的 trait，来让代码结构足够符合开闭原则和关注点分离，用了不少生命周期标注，来减少不必要的内存拷贝，还做了不少复杂的模式匹配来获取数据，是黄金级难度，在学完本课程的进阶篇后，你应该可以理解这些代码。

很多人觉得 Rust 代码很难写，尤其是泛型数据结构和生命周期搅在一起的时候。但在前两个例子中，生命周期的标注只出现过了一次。所以，**其实大部分时候，你的代码并不需要复杂的生命周期标注**。

只要对所有权和生命周期的理解没有问题，如果你陷入了无休止的生命周期标注，和编译器痛苦地搏斗，那你也许要停下来先想一想：

编译器如此不喜欢我的写法，会不会我的设计本身就有问题呢？我是不是该使用更好的数据结构？我是不是该重新设计一下？我的代码是不是过度耦合了？

就像茴香豆的茴字有四种写法一样，同一个需求，用相同的语言，不同的人也会有不同的写法。但是，**优秀的设计一定是产生简单易读的代码，而不是相反**。

好，这周的代码之旅就告一段落了，接下来我们就要展开一段壮丽的探险，你将会像比尔博·巴金斯那样，在通往孤山的冒险之旅中，一点点探索迷人的中土世界。等到我们学完了所有权、类型系统、trait、智能指针等内容之后，再来看这三个实例，相信你会有不一样的感悟。我也会在后续的课程中，根据已学内容，回顾今天写的代码，继续优化和完善它们。

思考题

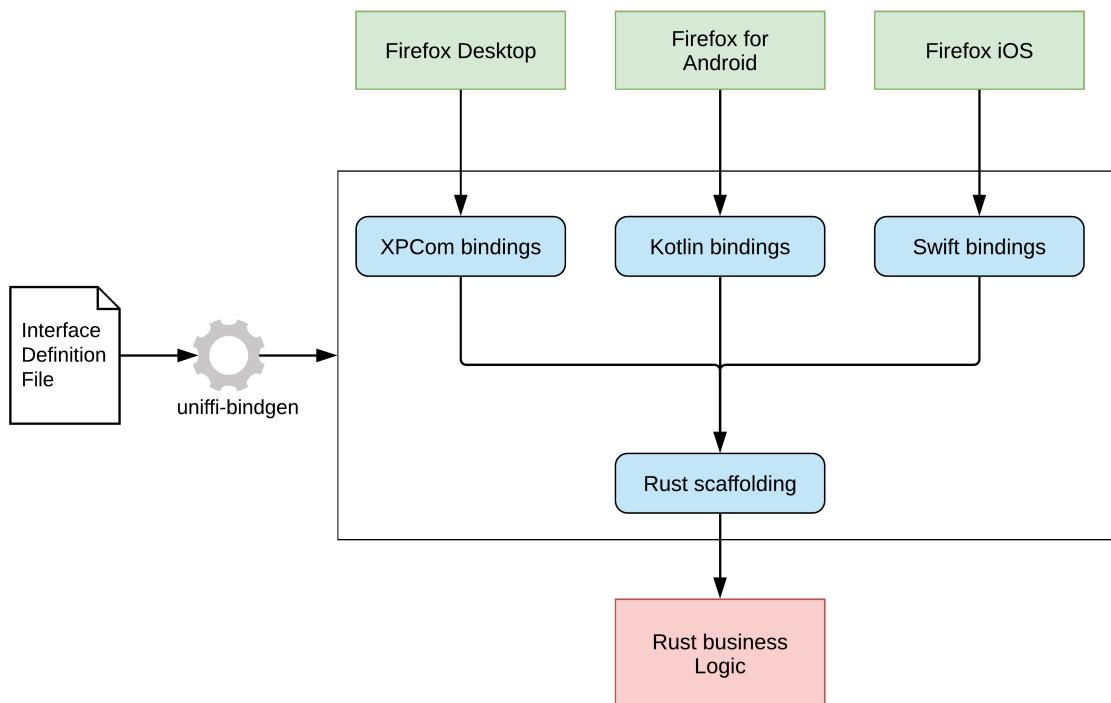
Node.js 的处理和 Python 非常类似，但接口不太一样，就作为今天的思考题让你尝试一下。小提示：Rust 和 nodejs 间交互可以使用 [neon](#)。

欢迎在留言区分享你的思考。你的 Rust 学习第六次打卡成功，我们下一讲见！

参考资料

我们的 queryer 库目前使用到了操作系统的功能，比如文件系统，所以它无法被编译成 WebAssembly。未来如果能移除对操作系统的依赖，这个代码还能被编译成 WASM，供 Web 前端使用。

如果想在 iOS/Android 下使用这个库，可以用类似 Python/Node.js 的方法做接口封装，Mozilla 提供了一个 [uniffi](#) 的库，它自己的 Firefox 各个端也是这么处理的：



对于桌面开发，Rust 下有一个很有潜力的客户端开发工具 [tauri](#)，它很有可能取代很多使用 Electron 的场合。

我写了一个简单的 tauri App 叫 data-viewer，如果你感兴趣的话，可以在 [github repo](#) 下的 data-viewer 目录下看 tauri 使用 queryer 的代码，下面是运行后的效果。为了让代码最简单，前端没有用任何框架，如果你是一名前端开发者，可以用 Vue 或者 React 加上一个合适的 CSS 库让整个界面变得更加友好。

The screenshot shows a window titled "Data Viewer" with a title bar "data-viewer". In the main area, there is a code editor-like interface with a SQL query and its results. The query is:

```
SELECT location_name, total_cases, new_cases, total_deaths, new_deaths FROM https://raw.githubusercontent.com/owid/covid-19-data/master/public/data/latest/owid-covid-latest.csv WHERE new_deaths >= 120 ORDER BY new_cases DESC
```

The results table has columns: name, total_cases, new_cases, total_deaths, new_deaths. The data includes:

name	total_cases	new_cases	total_deaths	new_deaths
World	215448179.0	821174.0	4486020.0	11491.0
North America	46482777.0	363748.0	970161.0	4254.0
United States	38707294.0	322934.0	636720.0	3156.0
Asia	69135616.0	271675.0	1023341.0	4001.0
Europe	55248221.0	116521.0	1170825.0	1440.0
India	32649947.0	46759.0	437370.0	509.0
European Union	36185159.0	42803.0	752892.0	372.0
Iran	4869414.0	36279.0	105287.0	571.0
Africa	7695475.0	33957.0	193394.0	764.0
South America	36768062.0	33853.0	1126593.0	1019.0
Brazil	20703906.0	27345.0	578326.0	761.0
Malaysia	1662913.0	22070.0	15550.0	339.0
Mexico	3311317.0	19556.0	257150.0	863.0
Russia	6747681.0	18982.0	176904.0	777.0
Thailand	1139571.0	18702.0	10587.0	273.0
Turkey	6311607.0	18340.0	55713.0	244.0
Vietnam	410366.0	17428.0	10053.0	386.0
Indonesia	4056354.0	12618.0	130781.0	599.0
South Africa	2747018.0	12045.0	81187.0	361.0
Spain	4831809.0	9489.0	84000.0	139.0
Argentina	5167733.0	5807.0	111270.0	153.0
Sri Lanka	416961.0	4591.0	8371.0	214.0
Pakistan	1148572.0	4231.0	25535.0	120.0

21 | 阶段实操：构建一个简单的 KV server (1) – 基本流程（上）

21 | 阶段实操：构建一个简单的 KV server (1) – 基本流程（上）

你好，我是陈天。

从第七讲开始，我们一路过关斩将，和所有权、生命周期死磕，跟类型系统和 trait 反复拉锯，为的是啥？就是为了能够读懂别人写的代码，进而让自己也能写出越来越复杂且优雅的代码。

今天就到检验自身实力的时候了，毕竟 talk is cheap，知识点掌握得再多，自己写不出来也白搭，所以我们把之前学的知识都运用起来，一起写个简单的 KV server。

不过这次和 get hands dirty 重感性体验的代码不同，我会带你一步步真实打磨，讲得比较细致，所以内容也会比较多，我分成了上下两篇文章，希望你能耐心看完，认真感受 Rust best practice 在架构设计以及代码实现路上的体现。

为什么选 KV server 来实操呢？因为它是一个**足够简单又足够复杂**的服务。参考工作中用到的 Redis / Memcached 等服务，来梳理它的需求。

- 最核心的功能是根据不同的命令进行诸如数据存贮、读取、监听等操作；
- 而客户端要能通过网络访问 KV server，发送包含命令的请求，得到结果；
- 数据要能根据需要，存储在内存中或者持久化到磁盘上。

先来一个短平糙的实现

如果是为了完成任务构建 KV server，其实最初的版本两三百行代码就可以搞定，但是这样的代码以后维护起来就是灾难。

我们看一个省却了不少细节的意大利面条式的版本，你可以随着我的注释重点看流程：

```

use anyhow::Result;
use async_prost::AsyncProstStream;
use dashmap::DashMap;
use futures::prelude::*;
use kv::*;

    command_request::RequestData, CommandRequest, CommandResponse, Hset, KvError, Kvpair, Value,
};

use std::sync::Arc;
use tokio::net::TcpListener;
use tracing::info;

#[tokio::main]
async fn main() -> Result<()> {
    //
    tracing_subscriber::fmt::init();

    let addr = "127.0.0.1:9527";
    let listener = TcpListener::bind(addr).await?;
    info!("Start listening on {}", addr);

    // DashMap kv store
    let table: Arc<DashMap<String, Value>> = Arc::new(DashMap::new());

    loop {
        //
        let (stream, addr) = listener.accept().await?;
        info!("Client {} connected", addr);

        // db tokio
        let db = table.clone();

        // tokio
        tokio::spawn(async move {
            // AsyncProstStream TCP Frame
            // Frame: frame protobuf
            let mut stream =
                AsyncProstStream::from(stream);
            for_async();
            //
            stream.decode
        });
    }
}

```

```

        while let Some(Ok(msg)) = stream.next().await {
            info!("Got a new command: {:?}", msg);
            let resp: CommandResponse = match msg.request_data {
                // HSET
                Some(RequestData::Hset(cmd)) => hset(cmd, &db),
                //
                _ => unimplemented!(),
            };

            info!("Got response: {:?}", resp);
            // CommandResponse
            stream.send(resp).await.unwrap();
        }
    });
}
}

// hset
fn hset(cmd: Hset, db: &DashMap<String, Value>) -> CommandResponse {
    match cmd.pair {
        Some(Kvpair {
            key,
            value: Some(v),
        }) => {
            // db
            let old = db.insert(key, v).unwrap_or_default();
            // value CommandResponse
            old.into()
        }
        v => KvError::InvalidCommand(format!("hset: {:?}", v)).into(),
    }
}
}

```

这段代码非常地平铺直叙，从输入到输出，一蹴而就，如果这样写，任务确实能很快完成，但是它有种“完成之后，哪管洪水滔天”的感觉。

你复制代码后，打开两个窗口，分别运行“cargo run --example naive_server”和“cargo run --example client”，就可以看到运行 server 的窗口有如下打印：

```

Sep 19 22:25:34.016 INFO naive_server: Start listening on 127.0.0.1:9527
Sep 19 22:25:38.401 INFO naive_server: Client 127.0.0.1:51650 connected
Sep 19 22:25:38.401 INFO naive_server: Got a new command: CommandRequest { request_data: Some(Hset(Hset { table: "table1", pair: Some(Kvpair { key: "hello", value: Some(Value { value: Some(String("world")) }) }) })) }
Sep 19 22:25:38.401 INFO naive_server: Got response: CommandResponse { status: 200, message: "", values: [Value { value: None }], pairs: [] }

```

虽然整体功能算是搞定了，不过以后想继续为这个 KV server 增加新的功能，就需要来来回改这段代码。

此外，也不好做单元测试，因为所有的逻辑都被压缩在一起了，没有“单元”可言。虽然未来可以逐步把不同的逻辑分离到不同的函数，使主流程尽可能简单一些。但是，它们依旧是耦合在一起的，如果不做大的重构，还是解决不了实质的问题。

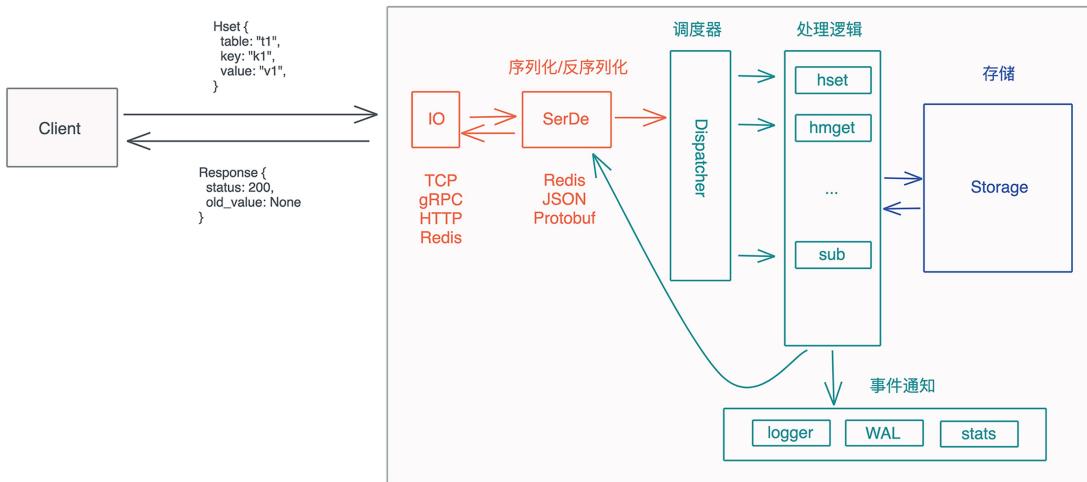
所以不管用什么语言开发，这样的代码都是我们要极力避免的，不光自己不要这么写，code review 遇到别人这么写也要严格地揪出来。

架构和设计

那么，怎样才算是好的实现呢？

好的实现应该是在分析完需求后，首先从系统的主流程开始，搞清楚从客户端的请求到最终客户端收到响应，都会经过哪些主要的步骤；然后根据这些步骤，思考哪些东西需要延迟绑定，构建主要的接口和 trait；等这些东西深思熟虑之后，最后再考虑实现。也就是所谓的“谋定而后动”。

开头已经分析 KV server 这个需求，现在我们来梳理主流程。你可以先自己想想，再参考示意图看看有没有缺漏：



这个流程中有一些关键问题需要进一步探索：

1. 客户端和服务器用什么协议通信？TCP？gRPC？HTTP？支持一种还是多种？
2. 客户端和服务器之间交互的应用层协议如何定义？怎么做序列化/反序列化？是用 Protobuf、JSON 还是 Redis RESP？或者也可以支持多种？
3. 服务器都支持哪些命令？第一版优先支持哪些？
4. 具体的处理逻辑中，需不需要加 hook，在处理过程中发布一些事件，让其他流程可以得到通知，进行额外的处理？这些 hook 可不可以提前终止整个流程的处理？
5. 对于存储，要支持不同的存储引擎么？比如 MemDb（内存）、RocksDb（磁盘）、SledDb（磁盘）等。对于 MemDb，我们考虑支持 WAL（Write-Ahead Log）和 snapshot 么？
6. 整个系统可以配置么？比如服务使用哪个端口、哪个存储引擎？
7. ...

如果你想做好架构，那么，问出这些问题，并且找到这些问题的答案就很重要。值得注意的是，这里面很多问题产品经理并不能帮你回答，或者TA的回答会将你带入歧路。作为一个架构师，我们需要对系统未来如何应对变化负责。

下面是我的思考，你可以参考：

1. 像 KV Server 这样需要高性能的场景，通信应该优先考虑 TCP 协议。所以我们暂时只支持 TCP，未来可以根据需要支持更多的协议，如 HTTP2/gRPC。还有，未来可能对安全性有额外的要求，所以我们要保证 TLS 这样的安全协议可以即插即用。总之，**网络层需要灵活**。

2. 应用层协议我们可以用 protobuf 定义。protobuf 直接解决了协议的定义以及如何序列化和反序列化。Redis 的 RESP 固然不错，但它的短板也显而易见，命令需要额外的解析，而且大量的 \r\n 来分隔命令或者数据，也有些浪费带宽。使用 JSON 的话更加浪费带宽，且 JSON 的解析效率不高，尤其是数据量很大的时候。

protobuf 就很适合 KV server 这样的场景，灵活、可向后兼容式升级、解析效率很高、生成的二进制非常省带宽，唯一的缺点是需要额外的工具 protoc 来编译成不同的语言。虽然 protobuf 是首选，但也许未来为了和 Redis 客户端互通，还是要支持 RESP。

3. 服务器支持的命令我们可以参考[Redis 的命令集](#)。第一版先来支持 HXXX 命令，比如 HSET、HMSET、HGET、HMGET 等。从命令到命令的响应，可以做个 trait 来抽象。

4. 处理流程中计划加这些 hook：收到客户端的命令后 OnRequestReceived、处理完客户端的命令后 OnRequestExecuted、发送响应之前 BeforeResponseSend、发送响应之后 AfterResponseSend。这样，**处理过程中的主要步骤都有事件暴露出去，让我们的 KV server 可以非常灵活，方便调用者在初始化服务的时候注入额外的处理逻辑**。

5. 存储必然需要足够灵活。可以对存储做个 trait 来抽象其基本的行为，一开始可以就只做 MemDb，未来肯定需要有支持持久化的存储。

6.需要支持配置，但优先级不高。等基本流程搞定，使用过程中发现足够的痛点，就可以考虑配置文件如何处理了。

当这些问题都敲定下来，系统的基本思路就有了。我们可以先把几个重要的接口定义出来，然后仔细审视这些接口。

最重要的几个接口就是三个主体交互的接口：客户端和服务器的接口或者说协议、服务器和命令处理流程的接口、服务器和存储的接口。

客户端和服务器间的协议

首先是客户端和服务器之间的协议。来试着用 protobuf 定义一下我们第一版支持的客户端命令：

```
syntax = "proto3";

package abi;

// 
message CommandRequest {
    oneof request_data {
        Hget hget = 1;
        Hgetall hgetall = 2;
        Hmget hmget = 3;
        Hset hset = 4;
        Hmset hmset = 5;
        Hdel hdel = 6;
        Hmdel hmdel = 7;
        Hexist hexist = 8;
        Hmexist hmexist = 9;
    }
}

// 
message CommandResponse {
    // HTTP 2xx/4xx/5xx
    uint32 status = 1;
    // 2xx message
    string message = 2;
    // values
    repeated Value values = 3;
    // kv pairs
    repeated Kvpair pairs = 4;
}

// table key value
message Hget {
    string table = 1;
    string key = 2;
}
```

```
// table Kvpair
message Hgetall { string table = 1; }

// table key value
message Hmget {
    string table = 1;
    repeated string keys = 2;
}

// 
message Value {
    oneof value {
        string string = 1;
        bytes binary = 2;
        int64 integer = 3;
        double float = 4;
        bool bool = 5;
    }
}

// kvpair
message Kvpair {
    string key = 1;
    Value value = 2;
}

// table kvpair
// table table
message Hset {
    string table = 1;
    Kvpair pair = 2;
}

// table kvpair
// table table
message Hmset {
    string table = 1;
    repeated Kvpair pairs = 2;
}

// table key
message Hdel {
    string table = 1;
    string key = 2;
}

// table key
message Hmdel {
    string table = 1;
    repeated string keys = 2;
}

// key
message Hexist {
    string table = 1;
    string key = 2;
}
```

```
// key
message Hmexist {
    string table = 1;
    repeated string keys = 2;
}
```

通过 [prost](#), 这个 protobuf 文件可以被编译成 Rust 代码（主要是 struct 和 enum）, 供我们使用。你应该还记得, 之前在[第 5 讲](#)谈到 thumbor 的开发时, 已经见识到了 prost 处理 protobuf 的方式了。

CommandService trait

客户端和服务器间的协议敲定之后, 就要思考如何处理请求的命令, 返回响应。

我们目前打算支持 9 种命令, 未来可能支持更多命令。所以最好定义一个 trait 来统一处理所有的命令, 返回处理结果。在处理命令的时候, 需要和存储发生关系, 这样才能根据请求中携带的参数读取数据, 或者把请求中的数据存入存储系统中。所以, 这个 trait 可以这么定义:

```
/// Command
pub trait CommandService {
    /// Command Response
    fn execute(self, store: &impl Storage) -> CommandResponse;
}
```

有了这个 trait, 并且每一个命令都实现了这个 trait 后, dispatch 方法就可以是类似这样的代码:

```
// Request Response HGET/HGETALL/HSET
pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse {
    match cmd.request_data {
        Some(RequestData::Hget(param)) => param.execute(store),
        Some(RequestData::Hgetall(param)) => param.execute(store),
        Some(RequestData::Hset(param)) => param.execute(store),
        None => KvError::InvalidCommand("Request has no data".into()).into(),
        _ => KvError::Internal("Not implemented".into()).into(),
    }
}
```

这样, 未来我们支持新命令时, 只需要做两件事: 为命令实现 CommandService、在 dispatch 方法中添加新命令的支持。

Storage trait

再来看为不同的存储而设计的 Storage trait，它提供 KV store 的主要接口：

```
///  
pub trait Storage {  
    /// HashTable key value  
    fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;  
    /// HashTable key value value  
    fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError>;  
    /// HashTable key  
    fn contains(&self, table: &str, key: &str) -> Result<bool, KvError>;  
    /// HashTable key  
    fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;  
    /// HashTable kv pair  
    fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError>;  
    /// HashTable kv pair Iterator  
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;  
}
```

在 CommandService trait 中已经看到，在处理客户端请求的时候，与之打交道的是 Storage trait，而非具体的某个 store。这样做的好处是，未来根据业务的需要，在不同的场景下添加不同的 store，只需要为其实现 Storage trait 即可，不必修改 CommandService 有关的代码。

比如在 HGET 命令的实现时，我们使用 Storage::get 方法，从 table 中获取数据，它跟某个具体的存储方案无关：

```
impl CommandService for Hget {  
    fn execute(self, store: &impl Storage) -> CommandResponse {  
        match store.get(&self.table, &self.key) {  
            Ok(Some(v)) => v.into(),  
            Ok(None) => KvError::NotFound(self.table, self.key).into(),  
            Err(e) => e.into(),  
        }  
    }  
}
```

Storage trait 里面的绝大多数方法相信你可以定义出来，但 get_iter() 这个接口可能你会比较困惑，因为它返回了一个 Box，为什么？

之前 ([第 13 讲](#)) 讲过这是 trait object。

这里我们想返回一个 iterator，调用者不关心它具体是什么类型，只要可以不停地调用 next() 方法取到下一个值就可以了。不同的实现，可能返回不同的 iterator，如果要用同一个接口承载，我们需要使用 trait object。在使用 trait object 时，因为 Iterator 是个带有关联类型的 trait，所以这里需要指明关联类型 Item 是什么类型，这样调用者才好拿到这个类型进行处理。

你也许会有疑问，`set` / `del` 明显是个会导致 `self` 修改的方法，为什么它的接口依旧使用的是 `&self` 呢？

我们思考一下它的用法。对于 `Storage trait`，最简单的实现是 `in-memory` 的 `HashMap`。由于我们支持的是 `HSET` / `HGET` 这样的命令，它们可以从不同的表中读取数据，所以需要嵌套的 `HashMap`，类似 `HashMap<String, HashMap<String, Value>>`。

另外，由于要在多线程/异步环境下读取和更新内存中的 `HashMap`，所以我们需要类似 `Arc<RwLock<HashMap<String, Arc<RwLock<HashMap<String, Value>>>>>` 的结构。这个结构是一个多线程环境下具有内部可变性的数据结构，所以 `get` / `set` 的接口是 `&self` 就足够了。

小结

到现在，我们梳理了 `KV server` 的主要需求和主流程，思考了流程中可能出现的问题，也敲定了三个重要的接口：客户端和服务器的协议、`CommandService trait`、`Storage trait`。下一讲继续实现 `KV server`，在看讲解之前，你可以先想一想自己平时是怎么开发的。

思考题

想一想，对于 `Storage trait`，为什么返回值都用了 `Result<T, E>`？在实现 `MemTable` 的时候，似乎所有返回都是 `Ok(T)` 啊？

欢迎在留言区分享你的思考。我们下篇见～

22 | 阶段实操：构建一个简单的 KV server (1) – 基本流程（下）

22 | 阶段实操：构建一个简单的 KV server (1) – 基本流程（下）

你好，我是陈天。

上篇我们的 `KV store` 刚开了个头，写好了基本的接口。你是不是摩拳擦掌准备开始写具体实现的代码了？别着急，当定义好接口后，先不忙实现，在撰写更多代码前，我们可以从一个使用者的角度来体验接口如何使用、是否好用，反观设计有哪些地方有待完善。

还是按照上一讲定义接口的顺序来一个一个测试：首先我们来构建协议层。

实现并验证协议层

先创建一个项目: cargo new kv --lib。进入到项目目录, 在 Cargo.toml 中添加依赖:

```
[package]
name = "kv"
version = "0.1.0"
edition = "2018"

[dependencies]
bytes = "1" # buffer
prost = "0.8" # protobuf
tracing = "0.1" #

[dev-dependencies]
anyhow = "1" #
async-prost = "0.2.1" # protobuf TCP frame
futures = "0.3" # Stream trait
tokio = { version = "1", features = ["rt", "rt-multi-thread", "io-util", "macros", "net"] } #
tracing-subscriber = "0.2" #

[build-dependencies]
prost-build = "0.8" # protobuf
```

然后在项目根目录下创建 abi.proto, 把上文中 protobuf 的代码放进去。在根目录下, 再创建 build.rs:

```
fn main() {
    let mut config = prost_build::Config::new();
    config.bytes(&["."]);
    config.type_attribute(".", "#[derive(PartialOrd)]");
    config
        .out_dir("src/pb")
        .compile_protos(&["abi.proto"], &["."])
        .unwrap();
}
```

这个代码在第 5 讲已经见过了, build.rs 在编译期运行来进行额外的处理。

这里我们为编译出来的代码额外添加了一些属性。比如为 protobuf 的 bytes 类型生成 Bytes 而非缺省的 Vec, 为所有类型加入 PartialOrd 派生宏。关于 prost-build 的扩展, 你可以看[文档](#)。

记得创建 src/pb 目录, 否则编不过。现在, 在项目根目录下做 cargo build 会生成 src/pb/abi.rs 文件, 里面包含所有 protobuf 定义的消息的 Rust 数据结构。我们创建 src/pb/mod.rs, 引入 abi.rs, 并做一些基本的类型转换:

```

pub mod abi;

use abi::command_request::RequestData, *;

impl CommandRequest {
    /// HSET
    pub fn new_hset(table: impl Into<String>, key: impl Into<String>, value: Value) -> Self {
        Self {
            request_data: Some(RequestData::Hset(Hset {
                table: table.into(),
                pair: Some(Kvpair::new(key, value)),
            })),
        }
    }
}

impl Kvpair {
    /// kv pair
    pub fn new(key: impl Into<String>, value: Value) -> Self {
        Self {
            key: key.into(),
            value: Some(value),
        }
    }
}

/// String Value
impl From<String> for Value {
    fn from(s: String) -> Self {
        Self {
            value: Some(value::Value::String(s)),
        }
    }
}

/// &str Value
impl From<&str> for Value {
    fn from(s: &str) -> Self {
        Self {
            value: Some(value::Value::String(s.into())),
        }
    }
}

```

最后，在 src/lib.rs 中，引入 pb 模块：

```

mod pb;
pub use pb::abi::*;

```

这样，我们就有了能把 KV server 最基本的 protobuf 接口运转起来的代码。

在根目录下创建 examples，这样可以写一些代码测试客户端和服务器之间的协议。我们可以先创建一个 examples /client.rs 文件，写入如下代码：

```
use anyhow::Result;
use async_prost::AsyncProstStream;
use futures::prelude::*;
use kv::{CommandRequest, CommandResponse};
use tokio::net::TcpStream;
use tracing::info;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();

    let addr = "127.0.0.1:9527";
    //
    let stream = TcpStream::connect(addr).await?;

    // AsyncProstStream TCP Frame
    let mut client =
        AsyncProstStream::<_, CommandResponse, CommandRequest, _>::from(stream).for_async();

    // HSET
    let cmd = CommandRequest::new_hset("table1", "hello", "world".into());

    client.send(cmd).await?;
    if let Some(Ok(data)) = client.next().await {
        info!("Got response {:?}", data);
    }
    Ok(())
}
```

这段代码连接服务器的 9527 端口，发送一个 HSET 命令出去，然后等待服务器的响应。

同样的，我们创建一个 examples/dummy_server.rs 文件，写入代码：

```
use anyhow::Result;
use async_prost::AsyncProstStream;
use futures::prelude::*;
use kv::{CommandRequest, CommandResponse};
use tokio::net::TcpListener;
use tracing::info;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();
    let addr = "127.0.0.1:9527";
```

```

let listener = TcpListener::bind(addr).await?;
info!("Start listening on {}", addr);
loop {
    let (stream, addr) = listener.accept().await?;
    info!("Client {} connected", addr);
    tokio::spawn(async move {
        let mut stream =
            AsyncProstStream::<_, CommandRequest, CommandResponse, _>::from(stream);
        for _ in async () {
            while let Some(Ok(msg)) = stream.next().await {
                info!("Got a new command: {:?}", msg);
                // 404 response
                let mut resp = CommandResponse::default();
                resp.status = 404;
                resp.message = "Not found".to_string();
                stream.send(resp).await.unwrap();
            }
            info!("Client {} disconnected", addr);
        });
    });
}
}

```

在这段代码里，服务器监听 9527 端口，对任何客户端的请求，一律返回 status = 404， message 是“Not found”的响应。

如果你对这两段代码中的异步和网络处理半懂不懂，没关系，你先把代码抄下来运行。今天的内容跟网络无关，你重点看处理流程就行。未来会讲到网络和异步处理的。

我们可以打开一个命令行窗口，运行：RUST_LOG=info cargo run --example dummy_server --quiet。然后在另一个命令行窗口，运行：RUST_LOG=info cargo run --example client --quiet。

此时，服务器和客户端都收到了彼此的请求和响应，协议层看上去运作良好。一旦验证通过，就你可以进入下一步，因为协议层的其它代码都只是工作量而已，在之后需要的时候可以慢慢实现。

实现并验证 Storage trait

接下来构建 Storage trait。

我们上一讲谈到了如何使用嵌套的支持并发的 im-memory HashMap 来实现 storage trait。由于 Arc<RwLock<HashMap<K, V>>> 这样的支持并发的 HashMap 是一个刚需，Rust 生态有很多相关的 crate 支持，这里我们可以使用 dashmap 创建一个 MemTable 结构，来实现 Storage trait。

先创建 src/storage 目录，然后创建 src/storage/mod.rs，把刚才讨论的 trait 代码放进去后，在 src/lib.rs 中引入“mod storage”。此时会发现一个错误：并未定义 KvError。

所以来定义 KvError。第 18 讲讨论错误处理时简单演示了，如何使用 `thiserror` 的派生宏来定义错误类型，今天就用它来定义 KvError。创建 `src/error.rs`，然后填入：

```
use crate::Value;
use thiserror::Error;

#[derive(Error, Debug, PartialEq)]
pub enum KvError {
    #[error("Not found for table: {0}, key: {1}")]
    NotFound(String, String),

    #[error("Cannot parse command: `'{0}`")]
    InvalidCommand(String),
    #[error("Cannot convert value {:0} to {1}")]
    ConvertError(Value, &'static str),
    #[error("Cannot process command {0} with table: {1}, key: {2}. Error: {}")]
    StorageError(&'static str, String, String, String),

    #[error("Failed to encode protobuf message")]
    EncodeError(#[from] prost::EncodeError),
    #[error("Failed to decode protobuf message")]
    DecodeError(#[from] prost::DecodeError),

    #[error("Internal error: {0}")]
    Internal(String),
}
```

这些 `error` 的定义其实是在实现过程中逐步添加的，但为了讲解方便，先一次性添加。对于 `Storage` 的实现，我们只关心 `StorageError`，其它的 `error` 定义未来会用到。

同样，在 `src/lib.rs` 下引入 mod `error`，现在 `src/lib.rs` 是这个样子的：

```
mod error;
mod pb;
mod storage;

pub use error::KvError;
pub use pb::abi::*;
pub use storage::*;


```

`src/storage/mod.rs` 是这个样子的：

```
use crate::{KvError, Kvpair, Value};
```

```

/// 
pub trait Storage {
    /// HashTable key value
    fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
    /// HashTable key value value
    fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError>;
    /// HashTable key
    fn contains(&self, table: &str, key: &str) -> Result<bool, KvError>;
    /// HashTable key
    fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError>;
    /// HashTable kv pair
    fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError>;
    /// HashTable kv pair Iterator
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError>;
}

```

代码目前没有编译错误，可以在这个文件末尾添加测试代码，尝试使用这些接口了，当然，我们还没有构建 MemTable，但通过 Storage trait 已经大概知道 MemTable 怎么用，所以可以先写段测试体验一下：

```

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn memtable_basic_interface_should_work() {
        let store = MemTable::new();
        test_basi_interface(store);
    }

    #[test]
    fn memtable_get_all_should_work() {
        let store = MemTable::new();
        test_get_all(store);
    }

    fn test_basi_interface(store: impl Storage) {
        // set table key None
        let v = store.set("t1", "hello".into(), "world".into());
        assert!(v.unwrap().is_none());
        // set key
        let v1 = store.set("t1", "hello".into(), "world1".into());
        assert_eq!(v1, Ok(Some("world".into())));

        // get key
        let v = store.get("t1", "hello");
        assert_eq!(v, Ok(Some("world1".into())));

        // get key table None
        assert_eq!(Ok(None), store.get("t1", "hello1"));
        assert!(store.get("t2", "hello1").unwrap().is_none());

        // contains key true false
        assert_eq!(store.contains("t1", "hello"), Ok(true));
        assert_eq!(store.contains("t1", "hello1"), Ok(false));
    }
}

```

```

assert_eq!(store.contains("t2", "hello"), Ok(false));

// del key
let v = store.del("t1", "hello");
assert_eq!(v, Ok(Some("world1".into())));

// del key table None
assert_eq!(Ok(None), store.del("t1", "hello1"));
assert_eq!(Ok(None), store.del("t2", "hello"));
}

fn test_get_all(store: impl Storage) {
    store.set("t2", "k1".into(), "v1".into().unwrap());
    store.set("t2", "k2".into(), "v2".into().unwrap());
    let mut data = store.get_all("t2").unwrap();
    data.sort_by(|a, b| a.partial_cmp(b).unwrap());
    assert_eq!(
        data,
        vec![
            Kvpair::new("k1", "v1".into()),
            Kvpair::new("k2", "v2".into())
        ]
    )
}

fn test_get_iter(store: impl Storage) {
    store.set("t2", "k1".into(), "v1".into().unwrap());
    store.set("t2", "k2".into(), "v2".into().unwrap());
    let mut data: Vec<_> = store.get_iter("t2").unwrap().collect();
    data.sort_by(|a, b| a.partial_cmp(b).unwrap());
    assert_eq!(
        data,
        vec![
            Kvpair::new("k1", "v1".into()),
            Kvpair::new("k2", "v2".into())
        ]
    )
}
}

```

这种在写实现之前写单元测试，是标准的 TDD (Test-Driven Development) 方式。

我个人不是 TDD 的狂热粉丝，但会在构建完 trait 后，为这个 trait 撰写测试代码，因为写测试代码是个很好的验证接口是否好用的时机。毕竟我们不希望实现 trait 之后，才发现 trait 的定义有瑕疵，需要修改，这个时候改动的代价就比较大了。

所以，当 trait 推敲完毕，就可以开始写使用 trait 的测试代码了。在使用过程中仔细感受，如果写测试用例时用得不舒服，或者为了使用它需要做很多繁琐的操作，那么可以重新审视 trait 的设计。

你如果仔细看单元测试的代码，就会发现我始终秉持测试 trait 接口的思想。尽管在测试中需要一个实际的数据结构进行 trait 方法的测试，但核心的测试代码都用的泛型函数，让这些代码只跟 trait 相关。

这样，一来可以避免某个具体 trait 实现的干扰，二来在之后想加入更多 trait 实现时，可以共享测试代码。比如未来想支持 DiskTable，那么只消加几个测试例，调用已有的泛型函数即可。

好，搞定测试，确认trait设计没有什么问题之后，我们来写具体实现。可以创建 src/storage/memory.rs 来构建 MemTable：

```
use crate::{KvError, Kvpair, Storage, Value};
use dashmap::{mapref::one::Ref, DashMap};

/// DashMap MemTable Storage trait
#[derive(Clone, Debug, Default)]
pub struct MemTable {
    tables: DashMap<String, DashMap<String, Value>>,
}

impl MemTable {
    /// MemTable
    pub fn new() -> Self {
        Self::default()
    }

    /// name hash table
    fn get_or_create_table(&self, name: &str) -> Ref<String, DashMap<String, Value>> {
        match self.tables.get(name) {
            Some(table) => table,
            None => {
                let entry = self.tables.entry(name.into()).or_default();
                entry.downgrade()
            }
        }
    }
}

impl Storage for MemTable {
    fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
        let table = self.get_or_create_table(table);
        Ok(table.get(key).map(|v| v.value().clone()))
    }

    fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError> {
        let table = self.get_or_create_table(table);
        Ok(table.insert(key, value))
    }

    fn contains(&self, table: &str, key: &str) -> Result<bool, KvError> {
        let table = self.get_or_create_table(table);
        Ok(table.contains_key(key))
    }

    fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
        let table = self.get_or_create_table(table);
        Ok(table.remove(key).map(|(_k, v)| v))
    }
}
```

```

    }

    fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError> {
        let table = self.get_or_create_table(table);
        Ok(table
            .iter()
            .map(|v| Kvpair::new(v.key(), v.value().clone())))
            .collect())
    }

    fn get_iter(&self, _table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
        todo!()
    }
}

```

除了 `get_iter()` 外，这个实现代码非常简单，相信你看一下 `dashmap` 的文档，也能很快写出来。`get_iter()` 写起来稍微有些难度，我们先放下不表，会在下一篇 KV server 讲。如果你对此感兴趣，想挑战一下，欢迎尝试。

实现完成之后，我们可以测试它是否符合预期。注意现在 `src/storage/memory.rs` 还没有被添加，所以 `cargo` 并不会编译它。要在 `src/storage/mod.rs` 开头添加代码：

```
mod memory;
pub use memory::MemTable;
```

这样代码就可以编译通过了。因为还没有实现 `get_iter` 方法，所以这个测试需要被注释掉：

```
// #[test]
// fn memtable_iter_should_work() {
//     let store = MemTable::new();
//     test_get_iter(store);
// }
```

如果你运行 `cargo test`，可以看到测试都通过了：

```
> cargo test
Compiling kv v0.1.0 (/Users/tchen/projects/mycode/rust/geek-time-rust-resources/21/kv)
Finished test [unoptimized + debuginfo] target(s) in 1.95s
Running unitests (/Users/tchen/.target/debug/deps/kv-8d746b0f387a5271)

running 2 tests
test storage::tests::memtable_basic_interface_should_work ... ok
test storage::tests::memtable_get_all_should_work ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

```
Doc-tests kv

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

实现并验证 CommandService trait

Storage trait 我们就算基本验证通过了，现在再来验证 CommandService。

我们创建 src/service 目录，以及 src/service/mod.rs 和 src/service/command_service.rs 文件，并在 src/service/mod.rs 写入：

```
use crate::*;

mod command_service;

/// Command
pub trait CommandService {
    /// Command Response
    fn execute(self, store: &impl Storage) -> CommandResponse;
}
```

不要忘记在 src/lib.rs 中加入 service：

```
mod error;
mod pb;
mod service;
mod storage;

pub use error::KvError;
pub use pb::abi::*;
pub use service::*;
pub use storage::*;


```

然后，在 src/service/command_service.rs 中，我们可以先写一些测试。为了简单起见，就列 HSET、HGET、HGETALL 三个命令：

```
use crate::*;

#[cfg(test)]
```

```

mod tests {
    use super::*;

    use crate::command_request::RequestData;

#[test]
fn hset_should_work() {
    let store = MemTable::new();
    let cmd = CommandRequest::new_hset("t1", "hello", "world".into());
    let res = dispatch(cmd.clone(), &store);
    assert_res_ok(res, &[Value::default()], &[]);

    let res = dispatch(cmd, &store);
    assert_res_ok(res, &["world".into()], &[]);
}

#[test]
fn hget_should_work() {
    let store = MemTable::new();
    let cmd = CommandRequest::new_hset("score", "u1", 10.into());
    dispatch(cmd, &store);
    let cmd = CommandRequest::new_hget("score", "u1");
    let res = dispatch(cmd, &store);
    assert_res_ok(res, &[10.into()], &[]);
}

#[test]
fn hget_with_non_exist_key_should_return_404() {
    let store = MemTable::new();
    let cmd = CommandRequest::new_hget("score", "u1");
    let res = dispatch(cmd, &store);
    assert_res_error(res, 404, "Not found");
}

#[test]
fn hgetall_should_work() {
    let store = MemTable::new();
    let cmds = vec![
        CommandRequest::new_hset("score", "u1", 10.into()),
        CommandRequest::new_hset("score", "u2", 8.into()),
        CommandRequest::new_hset("score", "u3", 11.into()),
        CommandRequest::new_hset("score", "u1", 6.into()),
    ];
    for cmd in cmds {
        dispatch(cmd, &store);
    }

    let cmd = CommandRequest::new_hgetall("score");
    let res = dispatch(cmd, &store);
    let pairs = &[
        Kvpair::new("u1", 6.into()),
        Kvpair::new("u2", 8.into()),
        Kvpair::new("u3", 11.into()),
    ];
    assert_res_ok(res, &[], pairs);
}

// Request Response HGET/HGETALL/HSET
fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse {

```

```

match cmd.request_data.unwrap() {
    RequestData::Hget(v) => v.execute(store),
    RequestData::Hgetall(v) => v.execute(store),
    RequestData::Hset(v) => v.execute(store),
    _ => todo!(),
}
}

// 
fn assert_res_ok(mut res: CommandResponse, values: &[Value], pairs: &[Kvpair]) {
    res.pairs.sort_by(|a, b| a.partial_cmp(b).unwrap());
    assert_eq!(res.status, 200);
    assert_eq!(res.message, "");
    assert_eq!(res.values, values);
    assert_eq!(res.pairs, pairs);
}

// 
fn assert_res_error(res: CommandResponse, code: u32, msg: &str) {
    assert_eq!(res.status, code);
    assert!(res.message.contains(msg));
    assert_eq!(res.values, &[]);
    assert_eq!(res.pairs, &[]);
}
}

```

这些测试的作用就是验证产品需求，比如：

- HSET 成功返回上一次的值（这和 Redis 略有不同，Redis 返回表示多少 key 受影响的一个整数）
- HGET 返回 Value
- HGETALL 返回一组无序的 Kvpair

目前这些测试是无法编译通过的，因为里面使用了一些未定义的方法，比如 10.into(): 想把整数 10 转换成一个 Value、CommandRequest::new_hgetall("score"): 想生成一个 HGETALL 命令。

为什么要这么写？因为如果是 CommandService 接口的使用者，自然希望使用这个接口的时候，调用的整体感觉非常简单明了。

如果接口期待一个 Value，但在上下文中拿到的是 10、“hello”这样的值，那我们作为设计者就要考虑为 Value 实现 From，这样调用的时候最方便。同样的，对于生成 CommandRequest 这个数据结构，也可以添加一些辅助函数，来让调用更清晰。

到现在为止我们写了两轮测试了，相信你对测试代码的作用有大概理解。我们来总结一下：

1. 验证并帮助接口迭代

2. 验证产品需求

3. 通过使用核心逻辑，帮助我们更好地思考外围逻辑并反推其实现

前两点是最基本的，也是很多人对TDD的理解，其实还有更重要的也就是第三点。除了前面的辅助函数外，我们在测试代码中还看到了 dispatch 函数，它目前用来辅助测试。**但紧接着你会发现，这样的辅助函数，可以合并到核心代码中。这才是“测试驱动开发”的实质。**

好，根据测试，我们需要在 src/pb/mod.rs 中添加相关的外围逻辑，首先是 CommandRequest 的一些方法，之前写了 new_hset，现在再加入 new_hget 和 new_hgetall：

```
impl CommandRequest {
    /// HGET
    pub fn new_hget(table: impl Into<String>, key: impl Into<String>) -> Self {
        Self {
            request_data: Some(RequestData::Hget(Hget {
                table: table.into(),
                key: key.into(),
            })),
        }
    }

    /// HGETALL
    pub fn new_hgetall(table: impl Into<String>) -> Self {
        Self {
            request_data: Some(RequestData::Hgetall(Hgetall {
                table: table.into(),
            })),
        }
    }

    /// HSET
    pub fn new_hset(table: impl Into<String>, key: impl Into<String>, value: Value) -> Self {
        Self {
            request_data: Some(RequestData::Hset(Hset {
                table: table.into(),
                pair: Some(Kvpair::new(key, value)),
            })),
        }
    }
}
```

然后写对 Value 的 From 的实现：

```
/// i64 Value
impl From<i64> for Value {
    fn from(i: i64) -> Self {
        Self {
            value: Some(value::Value::Integer(i)),
        }
    }
}
```

```
    }
}
```

测试代码目前就可以编译通过了，然而测试显然会失败，因为还没有做具体的实现。我们在 src/service/command_service.rs 下添加 trait 的实现代码：

```
impl CommandService for Hget {
    fn execute(self, store: &impl Storage) -> CommandResponse {
        match store.get(&self.table, &self.key) {
            Ok(Some(v)) => v.into(),
            Ok(None) => KvError::NotFound(self.table, self.key).into(),
            Err(e) => e.into(),
        }
    }
}

impl CommandService for Hgetall {
    fn execute(self, store: &impl Storage) -> CommandResponse {
        match store.get_all(&self.table) {
            Ok(v) => v.into(),
            Err(e) => e.into(),
        }
    }
}

impl CommandService for Hset {
    fn execute(self, store: &impl Storage) -> CommandResponse {
        match self.pair {
            Some(v) => match store.set(&self.table, v.key, v.value.unwrap_or_default()) {
                Ok(Some(v)) => v.into(),
                Ok(None) => Value::default().into(),
                Err(e) => e.into(),
            },
            None => Value::default().into(),
        }
    }
}
```

这自然会引发更多的编译错误，因为我们很多地方都是用了 into() 方法，却没有实现相应的转换，比如，Value 到 CommandResponse 的转换、KvError 到 CommandResponse 的转换、Vec 到 CommandResponse 的转换等等。

所以在 src/pb/mod.rs 里继续补上相应的外围逻辑：

```
/// Value CommandResponse
impl From<Value> for CommandResponse {
    fn from(v: Value) -> Self {
        Self {
            status: StatusCode::OK.as_u16() as _,
            values: vec![v],
        }
    }
}
```

```

        ..Default::default()
    }
}
}

/// Vec<Kvpair> CommandResponse
impl From<Vec<Kvpair>> for CommandResponse {
    fn from(v: Vec<Kvpair>) -> Self {
        Self {
            status: StatusCode::OK.as_u16() as _,
            pairs: v,
            ..Default::default()
        }
    }
}

/// KvError CommandResponse
impl From<KvError> for CommandResponse {
    fn from(e: KvError) -> Self {
        let mut result = Self {
            status: StatusCode::INTERNAL_SERVER_ERROR.as_u16() as _,
            message: e.to_string(),
            values: vec![],
            pairs: vec![],
        };

        match e {
            KvError::NotFound(_, _) => result.status = StatusCode::NOT_FOUND.as_u16() as _,
            KvError::InvalidCommand(_) => result.status = StatusCode::BAD_REQUEST.as_u16() as _,
            _ => {}
        }

        result
    }
}

```

从前面写接口到这里具体实现，不知道你是否感受到了这样一种模式：在 Rust 下，**但凡出现两个数据结构 v1 到 v2 的转换，你都可以先以 v1.into() 来表示这个逻辑，继续往下写代码，之后再去补 From 的实现。**如果 v1 和 v2 都不是你定义的数据结构，那么你需要把其中之一用 struct 包装一下，来绕过（第 14 讲）之前提到的孤儿规则。

你学完这节课可以再去回顾一下[第 6 讲](#)，仔细思考一下当时说的“绝大多数处理逻辑都是把数据从一个接口转换成另一个接口”。

现在代码应该可以编译通过并测试通过了，你可以 cargo test 测试一下。

最后的拼图：Service 结构的实现

好，所有的接口，包括客户端/服务器的协议接口、Storage trait 和 CommandService trait 都验证好了，接下来就是考虑如何用一个数据结构把所有这些东西串联起来。

依旧从使用者的角度来看如何调用它。为此，我们在 src/service/mod.rs 里添加如下的测试代码：

```
#[cfg(test)]
mod tests {
    use super::*;

    use crate::{MemTable, Value};

    #[test]
    fn service_should_works() {
        // service Storage
        let service = Service::new(MemTable::default());

        // service clone
        let cloned = service.clone();

        // table t1 k1, v1
        let handle = thread::spawn(move || {
            let res = cloned.execute(CommandRequest::new_hset("t1", "k1", "v1".into()));
            assert_res_ok(res, &[Value::default()], &[]);
        });
        handle.join().unwrap();

        // table t1 k1 v1
        let res = service.execute(CommandRequest::new_hget("t1", "k1"));
        assert_res_ok(res, &["v1".into()], &[]);
    }
}

#[cfg(test)]
use crate::{Kvpair, Value};

// 
#[cfg(test)]
pub fn assert_res_ok(mut res: CommandResponse, values: &[Value], pairs: &[Kvpair]) {
    res.pairs.sort_by(|a, b| a.partial_cmp(b).unwrap());
    assert_eq!(res.status, 200);
    assert_eq!(res.message, "");
    assert_eq!(res.values, values);
    assert_eq!(res.pairs, pairs);
}

// 
#[cfg(test)]
pub fn assert_res_error(res: CommandResponse, code: u32, msg: &str) {
    assert_eq!(res.status, code);
    assert!(res.message.contains(msg));
    assert_eq!(res.values, &[]);
    assert_eq!(res.pairs, &[]);
}
```

注意，这里的 assert_res_ok() 和 assert_res_error() 是从 src/service/command_service.rs 中挪过来的。在开发的过程中，不光产品代码需要不断重构，测试代码也需要重构来贯彻 DRY 思想。

我见过很多生产环境的代码，产品功能部分还说得过去，但测试代码像是个粪坑，经年累月地 copy/paste 使其臭气熏天，每个开发者在添加新功能的时候，都掩着鼻子往里扔一坨走人，使得维护难度越来越高，每次需求变动，都涉及一大坨测试代码的变动，这样非常不好。

测试代码的质量也要和产品代码的质量同等要求。好的开发者写的测试代码的可读性也是非常强的。你可以对比上面写的三段测试代码多多感受。

在撰写测试的时候，我们要特别注意：测试代码要围绕着系统稳定的部分，也就是接口，来测试，而尽可能少地测试实现。这是我对我这么多年工作中血淋淋的教训的深刻总结。

因为产品代码和测试代码，两者总需要一个是相对稳定的，既然产品代码会不断地根据需求变动，测试代码就必然需要稳定一些。

那什么样的测试代码是稳定的？测试接口的代码是稳定的。只要接口不变，无论具体实现如何变化，哪怕今天引入一个新的算法，明天重写实现，测试代码依旧能够凛然不动，做好产品质量的看门狗。

好，我们回来写代码。在这段测试中，已经敲定了 Service 这个数据结构的使用蓝图，它可以跨线程，可以调用 execute 来执行某个 CommandRequest 命令，返回 CommandResponse。

根据这些想法，在 src/service/mod.rs 里添加 Service 的声明和实现：

```
/// Service
pub struct Service<Store = MemTable> {
    inner: Arc<ServiceInner<Store>>,
}

impl<Store> Clone for Service<Store> {
    fn clone(&self) -> Self {
        Self {
            inner: Arc::clone(&self.inner),
        }
    }
}

/// Service
pub struct ServiceInner<Store> {
    store: Store,
}

impl<Store: Storage> Service<Store> {
    pub fn new(store: Store) -> Self {
        Self {
```

```

        inner: Arc::new(ServiceInner { store }),
    }
}

pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {
    debug!("Got request: {:?}", cmd);
    // TODO: on_received
    let res = dispatch(cmd, &self.inner.store);
    debug!("Executed response: {:?}", res);
    // TODO: on_executed

    res
}
}

// Request Response HGET/HGETALL/HSET
pub fn dispatch(cmd: CommandRequest, store: &impl Storage) -> CommandResponse {
    match cmd.request_data {
        Some(RequestData::Hget(param)) => param.execute(store),
        Some(RequestData::Hgetall(param)) => param.execute(store),
        Some(RequestData::Hset(param)) => param.execute(store),
        None => KvError::InvalidCommand("Request has no data".into()).into(),
        _ => KvError::Internal("Not implemented".into()).into(),
    }
}
}

```

这段代码有几个地方值得注意：

- 首先 Service 结构内部有一个 ServiceInner 存放实际的数据结构，Service 只是用 Arc 包裹了 ServiceInner。这也是 Rust 的一个惯例，把需要在多线程下 clone 的主体和其内部结构分开，这样代码逻辑更加清晰。
- execute() 方法目前就是调用了 dispatch，但它未来潜在可以做一些事件分发。这样处理体现了 SRP (Single Responsibility Principle) 原则。
- dispatch 其实就是把测试代码的 dispatch 逻辑移动过来改动了一下。

再一次，我们重构了测试代码，把它的辅助函数变成了产品代码的一部分。现在，你可以运行 cargo test 测试一下，如果代码无法编译，可能是缺一些 use 代码，比如：

```

use crate::{
    command_request::requestData, CommandRequest, CommandResponse, KvError, MemTable, Storage,
};
use std::sync::Arc;
use tracing::debug;

```

新的 server

现在处理逻辑已经都完成了，可以写个新的 example 测试服务器代码。

把之前的 examples/dummy_server.rs 复制一份，成为 examples/server.rs，然后引入 Service，主要的改动就三句：

```
// main service
let service: Service = Service::new(MemTable::new());
// tokio::spawn service
let svc = service.clone();
// while loop svc cmd
let res = svc.execute(cmd);
```

你可以试着自己修改。完整的代码如下：

```
use anyhow::Result;
use async_prost::AsyncProstStream;
use futures::prelude::*;
use kv::{CommandRequest, CommandResponse, MemTable, Service};
use tokio::net::TcpListener;
use tracing::info;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();
    let service: Service = Service::new(MemTable::new());
    let addr = "127.0.0.1:9527";
    let listener = TcpListener::bind(addr).await?;
    info!("Start listening on {}", addr);
    loop {
        let (stream, addr) = listener.accept().await?;
        info!("Client {} connected", addr);
        let svc = service.clone();
        tokio::spawn(async move {
            let mut stream =
                AsyncProstStream::<_, CommandRequest, CommandResponse, _>::from(stream);
            for_async();
            while let Some(Ok(cmd)) = stream.next().await {
                let res = svc.execute(cmd);
                stream.send(res).await.unwrap();
            }
            info!("Client {} disconnected", addr);
        });
    }
}
```

完成之后，打开一个命令行窗口，运行：RUST_LOG=info cargo run --example server --quiet，然后在另一个命令行窗口，运行：RUST_LOG=info cargo run --example client --quiet。此时，服务器和客户端都收到了彼此的请求和响应，并且处理正常。

我们的 KV server 第一版的基本功能就完工了！当然，目前还只处理了 3 个命令，剩下 6 个需要你自己完成。

小结

KV server 并不是一个很难的项目，但想要把它写好，并不简单。如果你跟着讲解一步步走下来，可以感受到一个有潜在生产环境质量的 Rust 项目应该如何开发。在这上下两讲内容中，有两点我们一定要认真领会。

第一点，你要对需求有一个清晰的把握，找出其中不稳定的部分（variant）和比较稳定的部分（invariant）。在 KV server 中，不稳定的部分是，对各种新的命令的支持，以及对不同的 storage 的支持。**所以需要构建接口来消弭不稳定的因素，让不稳定的部分可以用一种稳定的方式来管理。**

第二点，代码和测试可以围绕着接口螺旋前进，使用 TDD 可以帮助我们进行这种螺旋式的迭代。**在一个设计良好的系统中：接口是稳定的，测试接口的代码是稳定的，实现可以是不稳定的。**在迭代开发的过程中，我们要不断地重构，让测试代码和产品代码都往最优的方向发展。

纵观我们写的 KV server，包括测试在内，你很难发现有函数或者方法超过 50 行，代码可读性非常强，几乎不需要注释，就可以理解。另外因为都是用接口做的交互，未来维护和添加新的功能，也基本上满足 OCP 原则，除了 dispatch 函数需要很小的修改外，其它新的代码都是在实现一些接口而已。

相信你能初步感受到在 Rust 下撰写代码的最佳实践。如果你之前用其他语言，已经采用了类似的最佳实践，那么可以感受一下同样的实践在 Rust 下使用的那种优雅；如果你之前由于种种原因，写的是类似之前意大利面条似的代码，那在开发 Rust 程序时，你可以试着接纳这种更优雅的开发方式。

毕竟，现在我们手中有了更先进的武器，就可以用更先进的打法。

思考题

1. 为剩下 6 个命令 HMGET、HMSET、HDEL、HMDEL、HEXIST、HMEEXIST 构建测试，并实现它们。在测试和实现过程中，你也许需要添加更多的 From 的实现。
2. 如果有余力，可以试着实现 MemTable 的 get_iter() 方法（后续的 KV Store 实现会讲）。

延伸思考

虽然我们的 KV server 使用了 concurrent hashmap 来处理并发，但这并不一定是最好的选择。

我们也可以创建一个线程池，每个线程有自己的 HashMap。当 HGET/HSET 等命令来临时，可以对 key 做个哈希，然后分派到“拥有”那个 key 的线程，这样，可以避免在处理的时候加锁，提高系统的吞吐。你可以想想如果用这种方式处理，该怎么做。

恭喜你完成了学习的第22次打卡。如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下一讲期中测试见～

26 | 阶段实操：构建一个简单的 KV server (2) – 高级 trait 技巧

26 | 阶段实操：构建一个简单的 KV server (2) – 高级 trait 技巧

你好，我是陈天。

到现在，泛型的基础知识、具体如何使用以及设计理念，我们已经学得差不多了，也和函数作了类比帮助你理解，泛型就是数据结构的函数。

如果你觉得泛型难学，是因为它的抽象层级比较高，需要足够多的代码阅读和撰写的历练。所以，通过学习，现阶段你能够看懂包含泛型的代码就够了，至于使用，只能靠你自己在后续练习中不断体会总结。如果实在觉得不好懂，**某种程度上说，你缺乏的不是泛型的能力，而是设计和架构的能力。**

今天我们就用之前1.0版简易的 KV store 来历练一把，看看怎么把之前学到的知识融入代码中。

在 [21 讲](#)、[22讲](#)中，我们已经完成了 KV store 的基本功能，但留了两个小尾巴：

1. Storage trait 的 get_iter() 方法没有实现；
2. Service 的 execute() 方法里面还有一些 TODO，需要处理事件的通知。

我们一个个来解决。先看 get_iter() 方法。

处理 Iterator

在开始撰写代码之前，先把之前在 src/storage/mod.rs 里注掉的测试，加回来：

```
#[test]
fn memtable_iter_should_work() {
```

```
    let store = MemTable::new();
    test_get_iter(store);
}
```

然后在 src/storge/memory.rs 里尝试实现它。

```
impl Storage for MemTable {
    ...
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
        // clone() table snapshot
        let table = self.get_or_create_table(table).clone();
        let iter = table
            .iter()
            .map(|v| Kvpair::new(v.key(), v.value().clone()));
        Ok(Box::new(iter)) // <-- here
    }
}
```

很不幸的，编译器提示我们 Box::new(iter) 不行，“cannot return value referencing local variable table”。这让人很不爽，究其原因，table.iter() 使用了 table 的引用，我们返回 iter，但 iter 引用了作为局部变量的 table，所以无法编译通过。

此刻，我们需要有一个能够完全占有 table 的迭代器。 Rust 标准库里提供了一个 trait IntoIterator，它可以把数据结构的所有权转移到 Iterator 中，看它的声明 ([代码](#))：

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

绝大多数的集合类数据结构都[实现了它](#)。DashMap 也实现了它，所以我们可以用 table.into_iter() 把 table 的所有权转移给 iter：

```
impl Storage for MemTable {
    ...
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
        // clone() table snapshot
        let table = self.get_or_create_table(table).clone();
        let iter = table.into_iter().map(|data| data.into());
        Ok(Box::new(iter))
    }
}
```

这里又遇到了数据转换，从 DashMap 中 iterate 出来的值 (String, Value) 需要转换成 Kvpair，我们依旧用 into() 来完成这件事。为此，需要为 Kvpair 实现这个简单的 From trait：

```
impl From<(String, Value)> for Kvpair {
    fn from(data: (String, Value)) -> Self {
        Kvpair::new(data.0, data.1)
    }
}
```

这两段代码都放在 src/storage/memory.rs 下。

Bingo! 这个代码可以编译通过。现在如果运行 cargo test 进行测试的话，对 get_iter() 接口的测试也能通过。

虽然这个代码可以通过测试，并且本身也非常精简，我们还是有必要思考一下，如果以后想为更多的 data store 实现 Storage trait，都会怎样处理 get_iter() 方法？

我们会：

1. 拿到一个关于某个 table 下的拥有所有权的 Iterator
2. 对 Iterator 做 map
3. 将 map 出来的每个 item 转换成 Kvpair

这里的第 2 步对于每个 Storage trait 的 get_iter() 方法的实现来说，都是相同的。有没有可能把它封装起来呢？使得 Storage trait 的实现者只需要提供它们自己的拥有所有权的 Iterator，并对 Iterator 里的 Item 类型提供 Into ?

来尝试一下，在 src/storage/mod.rs 中，构建一个 StorageIter，并实现 Iterator trait：

```
/// Storage iterator trait
/// iterator StorageIter
/// next() Into<Kvpair>
pub struct StorageIter<T> {
    data: T,
}

impl<T> StorageIter<T> {
    pub fn new(data: T) -> Self {
        Self { data }
    }
}

impl<T> Iterator for StorageIter<T>
```

```

where
    T: Iterator,
    T::Item: Into<Kvpair>,
{
    type Item = Kvpair;

    fn next(&mut self) -> Option<Self::Item> {
        self.data.next().map(|v| v.into())
    }
}

```

这样，我们在 `src/storage/memory.rs` 里对 `get_iter()` 的实现，就可以直接使用 `StorageIter` 了。不过，还要为 `DashMap` 的 `Iterator` 每次调用 `next()` 得到的值 (`String, Value`)，做个到 `Kvpair` 的转换：

```

impl Storage for MemTable {
    ...
    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {
        // clone() table snapshot
        let table = self.get_or_create_table(table).clone();
        let iter = StorageIter::new(table.into_iter()); // Ok(Box::new(iter))
    }
}

```

我们可以再次使用 `cargo test` 测试，同样通过！

如果回顾刚才撰写的代码，你可能会哑然一笑：我辛辛苦苦又写了 20 行代码，创建了一个新的数据结构，就是为了 `get_iter()` 方法里的一行代码改得更漂亮？何苦呢？

的确，在这个 KV server 的例子里，这样的抽象收益不大。但是，如果刚才那个步骤不是 3 步，而是 5 步/10 步，其中大量的步骤都是相同的，也就是说，我们每实现一个新的 store，就要撰写相同的代码逻辑，那么，这个抽象就非常有必要了。

支持事件通知

好，我们再来看事件通知。在 `src/service/mod.rs` 中（以下代码，如无特殊声明，都是在 `src/service/mod.rs` 中），目前的 `execute()` 方法还有很多 TODO 需要解决：

```

pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {
    debug!("Got request: {:?}", cmd);
    // TODO: on_received
    let res = dispatch(cmd, &self.inner.store);
    debug!("Executed response: {:?}", res);
}

```

```
// TODO: on_executed

    res
}
```

为了解决这些 TODO，我们需要提供事件通知的机制：

1. 在创建 Service 时，注册相应的事件处理函数；
2. 在 execute() 方法执行时，做相应的事件通知，使得注册的事件处理函数可以得到执行。

先看事件处理函数如何注册。

如果想要能够注册，那么倒推也就是，Service/Servicelnner 数据结构就需要有地方能够承载事件注册函数。可以尝试着把它加在 Servicelnner 结构里：

```
/// Service
pub struct ServiceInner<Store> {
    store: Store,
    on_received: Vec<fn(&CommandRequest)>,
    on_executed: Vec<fn(&CommandResponse)>,
    on_before_send: Vec<fn(&mut CommandResponse)>,
    on_after_send: Vec<fn()>,
}
```

按照 21 讲的设计，我们提供了四个事件：

1. on_received：当服务器收到 CommandRequest 时触发；
2. on_executed：当服务器处理完 CommandRequest 得到 CommandResponse 时触发；
3. on_before_send：在服务器发送 CommandResponse 之前触发。注意这个接口提供的是 &mut CommandResponse，这样事件的处理器可以根据需要，在发送前，修改 CommandResponse。
4. on_after_send：在服务器发送完 CommandResponse 后触发。

在撰写事件注册的代码之前，还是先写个测试，从使用者的角度，考虑如何进行注册：

```
#[test]
fn event_registration_should_work() {
    fn b(cmd: &CommandRequest) {
        info!("Got {:?}", cmd);
    }
    fn c(res: &CommandResponse) {
```

```

        info!("{}:{}", res);
    }
    fn d(res: &mut CommandResponse) {
        res.status = StatusCode::CREATED.as_u16() as _;
    }
    fn e() {
        info!("Data is sent");
    }

    let service: Service = ServiceInner::new(MemTable::default())
        .fn_received(|_| &CommandRequest| {})
        .fn_received(b)
        .fn_executed(c)
        .fn_before_send(d)
        .fn_after_send(e)
        .into();
}

let res = service.execute(CommandRequest::new_hset("t1", "k1", "v1".into()));
assert_eq!(res.status, StatusCode::CREATED.as_u16() as _);
assert_eq!(res.message, "");
assert_eq!(res.values, vec![Value::default()]);
}

```

从测试代码中可以看到，我们希望通过 ServiceInner 结构，不断调用 fn_xxx 方法，为 ServiceInner 注册相应的事件处理函数；添加完毕后，通过 into() 方法，我们再把 ServiceInner 转换成 Service。这是一个经典的**构造者模式（Builder Pattern）**，在很多 Rust 代码中，都能看到它的身影。

那么，诸如 fn_received() 这样的方法有什么魔力呢？它为什么可以一路做链式调用呢？答案很简单，它把 self 的所有权拿过来，处理完之后，再返回 self。所以，我们继续添加如下代码：

```

impl<Store: Storage> ServiceInner<Store> {
    pub fn new(store: Store) -> Self {
        Self {
            store,
            on_received: Vec::new(),
            on_executed: Vec::new(),
            on_before_send: Vec::new(),
            on_after_send: Vec::new(),
        }
    }

    pub fn fn_received(mut self, f: fn(&CommandRequest)) -> Self {
        self.on_received.push(f);
        self
    }

    pub fn fn_executed(mut self, f: fn(&CommandResponse)) -> Self {
        self.on_executed.push(f);
        self
    }

    pub fn fn_before_send(mut self, f: fn(&mut CommandResponse)) -> Self {
        self.on_before_send.push(f);
        self
    }
}

```

```

        self
    }

    pub fn fn_after_send(mut self, f: fn()) -> Self {
        self.on_after_send.push(f);
        self
    }
}

```

这样处理之后呢，Service 之前的 new() 方法就没有必要存在了，可以把它删除。同时，我们需要为 Service 类型提供一个 From 的实现：

```

impl<Store: Storage> From<ServiceInner<Store>> for Service<Store> {
    fn from(inner: ServiceInner<Store>) -> Self {
        Self {
            inner: Arc::new(inner),
        }
    }
}

```

目前，代码中几处使用了 Service::new() 的地方需要改成使用 ServiceInner::new()，比如：

```

// service Storage
// let service = Service::new(MemTable::default());
let service: Service = ServiceInner::new(MemTable::default()).into();

```

全部改动完成后，代码可以编译通过。

然而，如果运行 cargo test，新加的测试会失败：

```
test service::tests::event_registration_should_work ... FAILED
```

这是因为，我们虽然完成了事件处理函数的注册，但现在还没有发事件通知。

另外因为我们的事件包括不可变事件（比如 on_received）和可变事件（比如 on_before_send），所以事件通知需要把二者分开。来定义两个 trait：Notify 和 NotifyMut：

```

/// 
pub trait Notify<Arg> {

```

```

    fn notify(&self, arg: &Arg);
}

/// 
pub trait NotifyMut<Arg> {
    fn notify(&self, arg: &mut Arg);
}

```

这两个 trait 是泛型 trait，其中的 Arg 参数，对应事件注册函数里的 arg，比如：

```
fn(&CommandRequest);
```

由此，我们可以特地为 `Vec<fn(&Arg)>` 和 `Vec<fn(&mut Arg)>` 实现事件处理，它们涵盖了目前支持的几种事件：

```

impl<Arg> Notify<Arg> for Vec<fn(&Arg)> {
    #[inline]
    fn notify(&self, arg: &Arg) {
        for f in self {
            f(arg)
        }
    }
}

impl<Arg> NotifyMut<Arg> for Vec<fn(&mut Arg)> {
    #[inline]
    fn notify(&self, arg: &mut Arg) {
        for f in self {
            f(arg)
        }
    }
}

```

Notify / NotifyMut trait 实现好之后，我们就可以修改 `execute()` 方法了：

```

impl<Store: Storage> Service<Store> {
    pub fn execute(&self, cmd: CommandRequest) -> CommandResponse {
        debug!("Got request: {:?}", cmd);
        self.inner.on_received.notify(&cmd);
        let mut res = dispatch(cmd, &self.inner.store);
        debug!("Executed response: {:?}", res);
        self.inner.on_executed.notify(&res);
        self.inner.on_before_send.notify(&mut res);
        if !self.inner.on_before_send.is_empty() {
            debug!("Modified response: {:?}", res);
        }
    }
}

```

```
        res
    }
}
```

现在，相应的事件就可以被通知到相应的处理函数中了。这个通知机制目前还是同步的函数调用，未来如果需要，我们可以将其改成消息传递，进行异步处理。

好，现在测试应该可以工作了，cargo test 所有的测试都通过。

为持久化数据库实现 Storage trait

到目前为止，我们的 KV store 还都是一个在内存中的 KV store。一旦终止应用程序，用户存储的所有 key / value 都会消失。我们希望存储能够持久化。

一个方案是为 MemTable 添加 WAL 和 disk snapshot 支持，让用户发送的所有涉及更新的命令都按顺序存储在磁盘上，同时定期做 snapshot，便于数据的快速恢复；另一个方案是使用已有的 KV store，比如 RocksDB，或者 sled。

RocksDB 是 Facebook 在 Google 的 leveldb 基础上开发的嵌入式 KV store，用 C++ 编写，而 sled 是 Rust 社区里涌现的优秀的 KV store，对标 RocksDB。二者功能很类似，从演示的角度，sled 使用起来更简单，更加适合今天的内容，如果在生产环境中使用，RocksDB 更加合适，因为它在各种复杂的生产环境中经历了千锤百炼。

所以，我们今天就尝试为 sled 实现 Storage trait，让它能够适配我们的 KV server。

首先在 Cargo.toml 里引入 sled：

```
sled = "0.34" # sled db
```

然后创建 src/storage/sleddb.rs，并添加如下代码：

```
use sled::Db;
use std::convert::TryInto;
use std::path::Path;
use std::str;

use crate::{KvError, Kvpair, Storage, StorageIter, Value};

#[derive(Debug)]
pub struct SledDb(Db);

impl SledDb {
```

```

pub fn new(path: impl AsRef<Path>) -> Self {
    Self(sled::open(path).unwrap())
}

// sled db scan_prefix prefix
// table
fn get_full_key(table: &str, key: &str) -> String {
    format!("{}:{}", table, key)
}

// table key prefix: table
fn get_table_prefix(table: &str) -> String {
    format!("{}:", table)
}

/// Option<Result<T, E>> flip Result<Option<T>, E>
///
fn flip<T, E>(x: Option<Result<T, E>>) -> Result<Option<T>, E> {
    x.map_or(Ok(None), |v| v.map(Some))
}

impl Storage for SledDb {
    fn get(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
        let name = SledDb::get_full_key(table, key);
        let result = self.0.get(name.as_bytes())?.map(|v| v.as_ref().try_into());
        flip(result)
    }

    fn set(&self, table: &str, key: String, value: Value) -> Result<Option<Value>, KvError> {
        let name = SledDb::get_full_key(table, &key);
        let data: Vec<u8> = value.try_into()?;

        let result = self.0.insert(name, data)?.map(|v| v.as_ref().try_into());
        flip(result)
    }

    fn contains(&self, table: &str, key: &str) -> Result<bool, KvError> {
        let name = SledDb::get_full_key(table, &key);

        Ok(self.0.contains_key(name)?)
    }

    fn del(&self, table: &str, key: &str) -> Result<Option<Value>, KvError> {
        let name = SledDb::get_full_key(table, &key);

        let result = self.0.remove(name)?.map(|v| v.as_ref().try_into());
        flip(result)
    }

    fn get_all(&self, table: &str) -> Result<Vec<Kvpair>, KvError> {
        let prefix = SledDb::get_table_prefix(table);
        let result = self.0.scan_prefix(prefix).map(|v| v.into()).collect();

        Ok(result)
    }

    fn get_iter(&self, table: &str) -> Result<Box<dyn Iterator<Item = Kvpair>>, KvError> {

```

```

        let prefix = SledDb::get_table_prefix(table);
        let iter = StorageIter::new(self.0.scan_prefix(prefix));
        Ok(Box::new(iter))
    }
}

impl From<Result<(IVec, IVec), sled::Error>> for Kvpair {
    fn from(v: Result<(IVec, IVec), sled::Error>) -> Self {
        match v {
            Ok((k, v)) => match v.as_ref().try_into() {
                Ok(v) => Kvpair::new(ivec_to_key(k.as_ref()), v),
                Err(_) => Kvpair::default(),
            },
            _ => Kvpair::default(),
        }
    }
}

fn ivec_to_key(ivec: &[u8]) -> &str {
    let s = str::from_utf8(ivec).unwrap();
    let mut iter = s.split(":");
    iter.next();
    iter.next().unwrap()
}

```

这段代码主要就是在实现 Storage trait。每个方法都很简单，就是在 sled 提供的功能上增加了一次封装。如果你对代码中某个调用有疑虑，可以参考 sled 的文档。

在 src/storage/mod.rs 里引入 sleddb，我们就可以加上相关的测试，测试新的 Storage 实现啦：

```

mod sleddb;

pub use sleddb::SledDb;

#[cfg(test)]
mod tests {
    use tempfile::tempdir;

    use super::*;

    ...

    #[test]
    fn sleddb_basic_interface_should_work() {
        let dir = tempdir().unwrap();
        let store = SledDb::new(dir);
        test_basi_interface(store);
    }

    #[test]
    fn sleddb_get_all_should_work() {
        let dir = tempdir().unwrap();

```

```

    let store = SledDb::new(dir);
    test_get_all(store);
}

#[test]
fn sleddb_iter_should_work() {
    let dir = tempdir().unwrap();
    let store = SledDb::new(dir);
    test_get_iter(store);
}
}

```

因为 SledDb 创建时需要指定一个目录，所以在测试中使用 `tempfile` 库，它能让文件资源在测试结束时被回收。我们在 `Cargo.toml` 中引入它：

```
[dev-dependencies]
...
 tempfile = "3" #
...
```

代码目前就可以编译通过了。如果你运行 `cargo test` 测试，会发现所有测试都正常通过！

构建新的 KV server

现在完成了 SledDb 和事件通知相关的实现，我们可以尝试构建支持事件通知，并且使用 SledDb 的 KV server 了。把 `examples/server.rs` 拷贝出 `examples/server_with_sled.rs`，然后修改 `let service` 那一行：

```
// let service: Service = ServiceInner::new(MemTable::new()).into();
let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserver"))
    .fn_before_send(|res| match res.message.as_ref() {
        "" => res.message = "altered. Original message is empty.".into(),
        s => res.message = format!("altered: {}", s),
    })
    .into();
```

当然，需要引入 SledDb 让编译通过。你看，只需要在创建 KV server 时使用 SledDb，就可以实现 data store 的切换，未来还可以进一步通过配置文件，来选择使用什么样的 store。非常方便。

新的 `examples/server_with_sled.rs` 的完整的代码：

```

use anyhow::Result;
use async_prost::AsyncProstStream;
use futures::prelude::*;
use kv1::{CommandRequest, CommandResponse, Service, ServiceInner, SledDb};
use tokio::net::TcpListener;
use tracing::info;

#[tokio::main]
async fn main() -> Result<()> {
    tracing_subscriber::fmt::init();
    let service: Service<SledDb> = ServiceInner::new(SledDb::new("/tmp/kvserver"))
        .fn_before_send(|res| match res.message.as_ref() {
            "" => res.message = "altered. Original message is empty.".into(),
            s => res.message = format!("altered: {}", s),
        })
        .into();
    let addr = "127.0.0.1:9527";
    let listener = TcpListener::bind(addr).await?;
    info!("Start listening on {}", addr);
    loop {
        let (stream, addr) = listener.accept().await?;
        info!("Client {} connected", addr);
        let svc = service.clone();
        tokio::spawn(async move {
            let mut stream =
                AsyncProstStream::<_, CommandRequest, CommandResponse, _>::from(stream).
for_async();
            while let Some(Ok(cmd)) = stream.next().await {
                info!("Got a new command: {:?}", cmd);
                let res = svc.execute(cmd);
                stream.send(res).await.unwrap();
            }
            info!("Client {} disconnected", addr);
        });
    }
}

```

它和之前的 server 几乎一样，只有 11 行生成 service 的代码应用了新的 storage，并且引入了事件通知。

完成之后，我们可以打开一个命令行窗口，运行：RUST_LOG=info cargo run --example server_with_sled --quiet。然后在另一个命令行窗口，运行：RUST_LOG=info cargo run --example client --quiet。

此时，服务器和客户端都收到了彼此的请求和响应，并且处理正常。如果你停掉服务器，再次运行，然后再运行客户端，会发现，客户端在尝试 HSET 时得到了服务器旧的值，我们的新版 KV server 可以对数据进行持久化了。

此外，如果你注意看 client 的日志，会发现原本应该是空字符串的 message 包含了“altered. Original message is empty.”：

```
RUST_LOG=info cargo run --example client --quiet
Sep 23 22:09:12.215  INFO client: Got response CommandResponse { status: 200, message:
"altered. Original message is empty.", values: [Value { value: Some(String("world")) }], pairs:
[ ] }
```

这是因为，我们的服务器注册了 `fn_before_send` 的事件通知，对返回的数据做了修改。未来我们可以用这些事件做很多事情，比如监控数据的发送，甚至写 WAL。

小结

今天的课程我们进一步认识到了 trait 的威力。当为系统设计了合理的 trait，整个系统的可扩展性就大大增强，之后在添加新的功能的时候，并不需要改动多少已有的代码。

在使用 trait 做抽象时，我们要衡量，这么做的好处是什么，它未来可以为实现者带来什么帮助。就像我们撰写的 `StorageIter`，它实现了 `Iterator` trait，并封装了 `map` 的处理逻辑，让这个公共的步骤可以在 `Storage` trait 中复用。

除此之外，也进一步熟悉了如何为带泛型参数的数据结构实现 trait。我们不仅可以为具体的数据结构实现 trait，也可以为更笼统的泛型参数实现 trait。除了文中这个例子：

```
impl<Arg> Notify<Arg> for Vec<fn(&Arg)> {
    #[inline]
    fn notify(&self, arg: &Arg) {
        for f in self {
            f(arg)
        }
    }
}
```

其实之前还见到过：

```
impl<T, U> Into<U> for T where U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

也是一样的道理。

如果结合这一讲和第 21、22 讲，你会发现，我们目前完成了一个功能比较完整的 KV server 的核心逻辑，但是，整体的代码似乎没有太多复杂的生命周期标注，或者太过抽象的泛型结构。

是的，别看我们在介绍 Rust 的基础知识时，扎的比较深，但是大多数写代码的时候，并不会用到那么深的知识。Rust 编译器会尽最大的努力，让你的代码简单。如果你用 clippy 这样的 linter 的话，它还会进一步给你提一些建议，让你的代码更加简单。

那么，为什么我们还要讲那么深入呢？

这是因为我们在写代码的时候不可避免地要引入第三方库，你也看到了，在写这个项目的时候用了不少依赖，当你使用这些库的时候，又不可避免地要阅读一些它们的源码，而这些源码，可能有各种各样复杂的写法。这也是为什么在开头我会说，现阶段能看懂包含泛型的代码就可以了。

深入地了解 Rust 的基础知识，可以帮助我们更快更清晰地阅读源码，而更快更清晰地读懂别人的源码，又可以更快地帮助我们用好别人的库，从而写好我们的代码。

思考题

1. 如果你在 21 讲已经完成了 KV server 其它的 6 个命令，可以对照着我在 [GitHub repo](#) 里的代码和测试，看看你写的结果。
2. 我们的 Notify 和 NotifyMut trait 目前只能做到通知，无法告诉 execute 提前结束处理并直接给客户端返回错误。试着修改一下这两个 trait，让它具备提前结束整个 pipeline 的能力。
3. [RocksDB](#) 是一个非常优秀的 KV DB，它有对应的 [rust 库](#)。尝试着为 RocksDB 实现 Storage trait，然后写个 example server 应用它。

感谢你的收听，你已经完成了 Rust 学习的第 26 次打卡，如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见~

加餐 | 期中测试：参考实现讲解

加餐 | 期中测试：参考实现讲解

你好，我是陈天。

上一讲给你布置了一份简单的期中考试习题，不知道你完成的怎么样。今天我们来简单讲一讲实现，供你参考。

支持 grep 并不是一件复杂的事情，相信你在使用了 clap、glob、rayon 和 regex 后，都能写出类似的代码（伪代码）：

```
/// Yet another simplified grep built with Rust.  
#[derive(Clap, Debug)]  
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]  
#[clap(setting = AppSettings::ColoredHelp)]  
pub struct GrepConfig {  
    /// regex pattern to match against file contents  
    pattern: String,  
    /// Glob of file pattern  
    glob: String,  
}  
  
impl GrepConfig {  
    pub fn matches(&self) -> Result<()> {  
        let regex = Regex::new(&self.pattern)?;  
        let files: Vec<_> = glob::glob(&self.glob)?.collect();  
        files.into_par_iter().for_each(|v| {  
            if let Ok(filename) = v {  
                if let Ok(file) = File::open(&filename) {  
                    let reader = BufReader::new(file);  
                    for (lineno, line) in reader.lines().enumerate() {  
                        if let Ok(line) = line {  
                            if let Some(_) = pattern.find(&line) {  
                                println!("{}: {}", lineno + 1, &line);  
                            }  
                        }  
                    }  
                }  
            }  
        });  
        Ok(())  
    }  
}
```

这个代码撰写的感觉和 Python 差不多，除了阅读几个依赖花些时间外，几乎没有难度。

不过，这个代码不具备可测试性，会给以后的维护和扩展带来麻烦。我们来看看如何优化，使这段代码更加容易测试。

如何写出好实现

首先，我们要剥离主要逻辑。

主要逻辑是什么？自然是对于单个文件的 grep，也就是代码中标记的部分。我们可以将它抽离成一个函数：

```
fn process(reader: BufReader<File>)
```

当然，从接口的角度来说，这个 `process` 函数定义得太死，如果不是从 `File` 中取数据，改天需求变了，也需要支持从 `stdio` 中取数据呢？就需要改动这个接口了。

所以可以使用泛型：

```
fn process<R: Read>(reader: BufReader<R>)
```

泛型参数 `R` 只需要满足 `std::io::Read` trait 就可以。

这个接口虽然抽取出来了，但它依旧不可测，因为它内部直接 `println!`，把找到的数据直接打印出来了。我们当然可以把要打印的行放入一个 `Vec` 返回，这样就可以测试了。

不过，这是为了测试而测试，**更好的方式是把输出的对象从 `Stdout` 抽象成 `Write`**。现在 `process` 的接口变为：

```
fn process<R: Read, W: Write>(reader: BufReader<R>, writer: &mut Writer)
```

这样，我们就可以使用实现了 `Read` trait 的 `&[u8]` 作为输入，以及使用实现了 `Write` trait 的 `Vec` 作为输出，进行测试了。而在 `rgrep` 的实现时，我们用 `File` 作为输入，`Stdout` 作为输出。这样既满足了需求，让核心逻辑可测，还让接口足够灵活，可以适配任何实现了 `Read` 的输入以及实现了 `Write` 的输出。

好，有了这个思路，来看看我是怎么写这个 `rgrep` 的，供你参考。

首先 `cargo new rgrep` 创建一个新的项目。在 `Cargo.toml` 中，添加如下依赖：

```
[dependencies]
anyhow = "1"
clap = "3.0.0-beta.4" # 3.0.0-beta.4
colored = "2"
glob = "0.3"
itertools = "0.10"
rayon = "1"
regex = "1"
thiserror = "1"
```

对于处理命令行的 clap，我们需要 3.0 的版本。不要在意 VS Code 插件提示你最新版本是 2.33，那是因为 beta 不算正式版本。

然后创建 src/lib.rs 和 src/error.rs，在 [error.rs](#) 中添加一些错误定义：

```
use thiserror::Error;

#[derive(Error, Debug)]
pub enum GrepError {
    #[error("Glob pattern error")]
    GlobPatternError(#[from] glob::PatternError),
    #[error("Regex pattern error")]
    RegexPatternError(#[from] regex::Error),
    #[error("I/O error")]
    IoError(#[from] std::io::Error),
}
```

它们都是需要进行转换的错误。thiserror 能够通过宏帮我们完成错误类型的转换。

在 src/lib.rs 中，添入如下代码：

```
use clap::AppSettings;
use colored::*;
use itertools::Itertools;
use rayon::iter::{IntoParallelIterator, ParallelIterator};
use regex::Regex;
use std::{
    fs::File,
    io::{self, BufRead, BufferedReader, Read, Stdout, Write},
    ops::Range,
    path::Path,
};

mod error;
pub use error::GrepError;

/// 
pub type StrategyFn<W, R> = fn(&Path, BufferedReader<R>, &Regex, &mut W) -> Result<(), GrepError>;

/// grep
#[derive(Clap, Debug)]
#[clap(version = "1.0", author = "Tyr Chen <tyr@chen.com>")]
#[clap(setting = AppSettings::ColoredHelp)]
pub struct GrepConfig {
    ///
    pattern: String,
    ///
    glob: String,
```

```

}

impl GrepConfig {
    /**
     pub fn match_with_default_strategy(&self) -> Result<(), GrepError> {
        self.match_with(default_strategy)
    }

    /**
     pub fn match_with(&self, strategy: StrategyFn<Stdout, File>) -> Result<(), GrepError> {
        let regex = Regex::new(&self.pattern)?;
        //
        let files: Vec<_> = glob::glob(&self.glob)?.collect();
        //
        files.into_par_iter().for_each(|v| {
            if let Ok(filename) = v {
                if let Ok(file) = File::open(&filename) {
                    let reader = BufReader::new(file);
                    let mut stdout = io::stdout();

                    if let Err(e) = strategy(filename.as_path(), reader, &regex, &mut stdout) {
                        println!("Internal error: {:?}", e);
                    }
                }
            }
        });
        Ok(())
    }

    /**
     pub fn default_strategy<W: Write, R: Read>(
        path: &Path,
        reader: BufReader<R>,
        pattern: &Regex,
        writer: &mut W,
    ) -> Result<(), GrepError> {
        let matches: String = reader
            .lines()
            .enumerate()
            .map(|(lineno, line)| {
                line.ok()
                    .map(|line| {
                        pattern
                            .find(&line)
                            .map(|m| format_line(&line, lineno + 1, m.range()))
                    })
                    .flatten()
            })
            .filter_map(|v| v.ok_or(()).ok())
            .join("\n");

        if !matches.is_empty() {
            writer.write(path.display().to_string().green().as_bytes())?;
            writer.write(b"\n")?;
            writer.write(matches.as_bytes())?;
            writer.write(b"\n")?;
        }
    }
}

```

```

        Ok(())
    }

/// 
pub fn format_line(line: &str, lineno: usize, range: Range<usize>) -> String {
    let Range { start, end } = range;
    let prefix = &line[..start];
    format!(
        "{0: >6}:{1: <3} {2}{3}{4}",
        lineno.to_string().blue(),
        // ascii  prefix.len()
        // O(n)
        (prefix.chars().count() + 1).to_string().cyan(),
        prefix,
        &line[start..end].red(),
        &line[end..]
    )
}

```

和刚才的思路稍有不同的是，process 函数叫 default_strategy()。另外我们为 GrepConfig 提供了两个方法，一个 is match_with_default_strategy()，另一个是 match_with()，调用者可以自己传入一个函数或者闭包，对给定的 BufReader 进行处理。这是一种常用的解耦的处理方法。

在 src/lib.rs 里，继续撰写单元测试：

```

#[cfg(test)]
mod tests {

    use super::*;

    #[test]
    fn format_line_should_work() {
        let result = format_line("Hello, Tyr~", 1000, 7..10);
        let expected = format!{
            "{0: >6}:{1: <3} Hello, {2}~",
            "1000".blue(),
            "7".cyan(),
            "Tyr".red()
        };
        assert_eq!(result, expected);
    }

    #[test]
    fn default_strategy_should_work() {
        let path = Path::new("src/main.rs");
        let input = b"hello world!\nhey Tyr!";
        let reader = BufReader::new(&input[..]);
        let pattern = Regex::new(r"he\w+").unwrap();
        let mut writer = Vec::new();
        default_strategy(path, reader, &pattern, &mut writer).unwrap();
        let result = String::from_utf8(writer).unwrap();

```

```

    let expected = [
        String::from("src/main.rs"),
        format_line("hello world!", 1, 0..5),
        format_line("hey Tyr!\n", 2, 0..3),
    ];

    assert_eq!(result, expected.join("\n"));
}
}

```

你可以重点关注测试是如何使用 `default_strategy()` 函数，而 `match_with()` 方法又是如何使用它的。运行 `cargo test`，两个测试都能通过。

最后，在 `src/main.rs` 中添加命令行处理逻辑：

```

use anyhow::Result;
use clap::Clap;
use rgrep::*;

fn main() -> Result<()> {
    let config: GrepConfig = GrepConfig::parse();
    config.match_with_default_strategy()?;
    Ok(())
}

```

在命令行下运行：`cargo run --quiet -- "Re[^\\s]+" "src/*.rs"`，会得到类似如下输出。注意，文件输出的顺序可能不完全一样，因为 rayon 是多个线程并行执行的。

```

> cargo run --quiet -- "Re[^\\s]+" "src/*.rs"
src/main.rs
  1:13  use anyhow::Result;
  5:14  fn main() -> Result<()> {
src/error.rs
  7:14      #[error("Regex pattern error")]
  8:5      RegexPatternError(#[from] regex:Error),
src/lib.rs
  5:12  use regex::Regex;
  8:19      io::{self, BufRead, BufferedReader, Read, Stdout, Write},
17:42  pub type StrategyFn<W, R> = fn(&Path, BufReader<R>, &Regex, &mut W) -> Result<(), GrepError>;
32:50  pub fn match_with_default_strategy(&self) -> Result<(), GrepError> {
37:69      pub fn match_with(&self, strategy: StrategyFn<Stdout, File>) -> Result<(), GrepError> {
38:21          let regex = Regex::new(&self.pattern)?;
45:37          let reader = BufReader::new(file);
59:38  pub fn default_strategy<W: Write, R: Read>(
61:16      reader: BufReader<R>,
62:15      pattern: &Regex,
64:6  ) -> Result<(), GrepError> {
127:25      let reader = BufReader::new(&input[..]);
128:23      let pattern = Regex::new(r"he\w+").unwrap();

```

小结

rgrep 是一个简单的命令行工具，仅仅写了上百行代码，就完成了一个性能相当不错的简化版 grep。在不做复杂的接口设计时，我们可以不用生命周期，不用泛型，甚至不用太关心所有权，就可以写出非常类似脚本语言的代码。

从这个意义上讲，Rust 用来自做一次性的、即用即抛型的代码，或者说，写个快速原型，也有用武之地；当我们需要更好的代码质量、更高的抽象度、更灵活的设计时，Rust 提供了足够多的工具，让我们将原型进化成更成熟的代码。

相信在做 rgrep 的过程中，你能感受到用 Rust 开发软件的愉悦。

今天我们就不再布置思考题了，你可以多多体会KV server和rgrep工具的实现。恭喜你完成了Rust基础篇的学习，进度条过半，我们下节课进阶篇见。

欢迎你分享给身边的朋友，邀他一起讨论。

延伸阅读

在 YouTube 上，有一个新鲜出炉的视频：[Visualizing memory layout of Rust's data types](#)，用 40 分钟的时间，总结了我们前面基础篇二十讲里提到的主要数据结构的内存布局。我个人非常喜欢这个视频，因为它和我一直倡导的“厘清数据是如何在堆和栈上存储”的思路不谋而合，在这里也推荐给你。如果你想快速复习一下，查漏补缺，那么非常建议你花上一个小时时间仔细看一下这个视频。

加餐 | 期中测试：来写一个简单的 grep 命令行

加餐 | 期中测试：来写一个简单的 grep 命令行

你好，我是陈天。

现在 Rust 基础篇已经学完了，相信你已经有足够的信心去应对一些简单的开发任务。今天我们就来个期中测试，实际考察一下你对 Rust 语言的理解以及对所学知识的应用情况。

我们要做的小工具是 rgrep，它是一个类似 grep 的工具。如果你是一个 *nix 用户，那大概率使用过 grep 或者 ag 这样的文本查找工具。

grep 命令用于查找文件里符合条件的字符串。如果发现某个文件的内容符合所指定的字符串，grep 命令会把含有字符串的那一行显示出；若不指定任何文件名称，或是所给予的文件名为 -，grep 命令会从标准输入设备读取数据。

我们的 rgrep 要稍微简单一些，它可以支持以下三种使用场景：

首先是最简单的，给定一个字符串以及一个文件，打印出文件中所有包含该字符串的行：

```
$ rgrep Hello a.txt
55: Hello world. This is an exmaple text
```

然后放宽限制，允许用户提供一个正则表达式，来查找文件中所有包含该字符串的行：

```
$ rgrep Hel[^\s]+ a.txt
55: Hello world. This is an exmaple text
89: Help me! I need assistant!
```

如果这个也可以实现，那进一步放宽限制，允许用户提供一个正则表达式，来查找满足文件通配符的所有文件（你可以使用 [globset](#) 或者 [glob](#) 来处理通配符），比如：

```
$ rgrep Hel[^\s]+ a*.txt
a.txt
  55:1 Hello world. This is an exmaple text
  89:1 Help me! I need assistant!
  5:6  Use `Help` to get help.
abc.txt:
  100:1 Hello Tyr!
```

其中，冒号前面的数字是行号，后面的数字是字符在这一行的位置。

给你一点小提示。

- 对于命令行的部分，你可以使用 [clap3](#) 或者 [structopt](#)，也可以就用 `env.args()`。
- 对于正则表达式的支持，可以使用 [regex](#)。
- 至于文件的读取，可以使用 [std::fs](#) 或者 [tokio::fs](#)。你可以顺序对所有满足通配符的文件进行处理，也可以用 [rayon](#) 或者 [tokio](#) 来并行处理。
- 对于输出的结果，最好能把匹配的文字用不同颜色展示。

```

> rgrep "君子" "/*.md"
06.md
  5:66 子华使于齐，冉子为其母请粟。子曰：“与之釜。”请益。曰：“与之庾。”冉子与之粟五秉。子曰：“赤之适齐也，乘肥马，衣轻裘。吾闻之也，君子周急不继富。”
  14:10 子谓子夏曰：“女为君子儒，无为小人儒。”
  19:24 子曰：“质胜文则野，文胜质则史。文质彬彬，然后君子。”
  27:38 宰我问曰：“仁者，虽告之曰：‘井有仁焉。’其从之也？”子曰：“何为其然也？君子可逝也，不可陷也；可欺也，不可罔也。”
  28:5 子曰：“君子博学于文，约之以礼，亦可以弗畔矣夫！””

10.md
  7:1 君子不以绀緞饰。红紫不以为亵服。当暑，袗絺绤，必表而出之。缁衣羔裘，素衣麑裘，黄衣狐裘。裘袭长。短右袂。必有寝衣，长一身有半。狐貉之厚以居。去丧，无所不佩。非帷裳，必杀之。羔裘玄冠不以吊。吉月，必朝服而朝。
01.md
  2:37 子曰：“学而时习之，不亦说乎？有朋自远方来，不亦乐乎？人不知而不愠，不亦君子乎？”
  3:38 有子曰：“其为人也孝弟，而好犯上者，鲜矣；不好犯上，而好作乱者，未之有也。君子务本，本立而道生。孝弟也者，其为仁之本与！”
  9:5 子曰：“君子不重则威，学则不固。主忠信，无友不如己者，过则勿惮改。”
  15:5 子曰：“君子食无求饱，居无求安，敏于事而慎于言，就有道而正焉，可谓好学也已。”
19.md02.md
  13:5 子曰：“君子不器。”
  14:4 子贡问君子。子曰：“先行其言，而后从之。”
  15:5 子曰：“君子周而不比，小人比而不周。”
08.md
  3:33 子曰：“恭而无礼则劳，慎而无礼则葸，勇而无礼则乱，直而无礼则绞。君子笃于亲，则民兴于仁；故旧不遗，则民不偷。”
  5:38 曾子有疾，孟敬子问之。曾子言曰：“鸟之将死，其鸣也哀；人之将死，其言也善。君子所贵乎道者三：动容貌，斯远暴慢矣；正颜色，斯近信矣；出辞气，斯远鄙倍矣。笾豆之事，则有司存。”
  7:31 曾子曰：“可以托六尺之孤，可以寄百里之命，临大节而不可夺也。君子人与？君子人也。”
13.md
  4:59 子路曰：“卫君待子而为政，子将奚先？”子曰：“必也正名乎！”子路曰：“有是哉，子之迂也！奚其正？”子曰：“野哉由也！君子于其所不知，盖阙如也。名不正，则言不顺；言不顺，则事不成；事不成，则礼乐不兴；礼乐不兴，则刑罚不中；刑罚不中，则民无所措手足。故君子名之必可言也，言之必可行也。君子于其言，无所苟而已矣。”
  24:5 子曰：“君子和而不同，小人同而不和。”
  26:5 子曰：“君子易事而难说也：说之不以道，不说也；及其使人也，器之。小人难事而易说也：说之虽不以道，说也；及其使人也，求备焉。”
  27:5 子曰：“君子泰而不骄，小人骄而不泰。”
12.md
  4:57 子夏之门人问交于子张。子张曰：“子夏云何？”对曰：“子夏曰：‘可者与之，其不可者拒之。’”子张曰：“异乎吾所闻：君子尊贤而容众，嘉善而矜不能。我之大贤与，天下何所不容？我之不贤与，人将拒我，如之何其拒人也？””

```

如果你有余力，可以看看 grep 的文档，尝试实现更多的功能。

祝你好运！

加油，我们下节课作业讲解见。

04 生态篇

有哪些常有的 Rust 库可以为我所用？

你好，我是陈天。

一门编程语言的能力，语言本身的设计占了四成，围绕着语言打造的生态系统占了六成。

[之前](#)我们对比过 Golang 和 Rust，在我看来，Golang 是一门优点和缺点同样突出的语言，Golang 的某些缺点甚至是很严重的，然而，在 Google 的大力加持下，借助微服务和云原生的春风，Golang 构建了一个非常宏大的生态系统。基本上，如果你要做微服务，Golang 完善的第三方库能够满足你几乎所有的需求。

所以，生态可以弥补语言的劣势，[编程语言对外展现出来的能力是语言+生态的一个合集](#)。

举个例子，由于不支持宏编程，Golang 在开发很多项目时不得不引入大量的脚手架代码，这些脚手架代码如果自己写，费时费力，但是社区里会有一大票优秀的框架，帮助你生成这些脚手架代码。

典型的比如 [kubebuilder](#)，它直接把开发 Kubernetes 下 operator 的门槛降了一大截，如果没有类似的工具，用 Golang 开发 Kubernetes 并不比 Python 来得容易。反之，承蒙在 data science 和 machine learning 上无比优秀且简洁实用的生态系统，Python 才得以在这两个领域笑傲江湖，独孤求败。

那么，Rust 的生态是什么样子呢？我们可以用 Rust 做些什么事情呢？为什么我说 Rust 生态系统已经不错，且潜力无穷、后劲很足呢？我们就聊聊这个话题。

今天的内容主要是丰富你对Rust生态系统的了解，方便你在做不同的项目时，可以快速找到适合的库和工具。当然，我无法把所有重要的 crate 都罗列出来，如果本文中的内容无法涵盖到你的需求，也可以去 [crates.io](#) 自行查找。

基础库

首先我们来介绍一些在各类应用中可能都会用到的库。

先按照重要程度依次简单说一下，方便你根据需要自行跳转：序列化和反序列化工具 `serde`、网络和高性能 I/O 库 `tokio`、用于错误处理的 `thiserror` 和 `anyhow`、用于命令行处理的 `clap` 以及其他、用于处理异步的 `futures` 和 `async-trait`、用于提供并发相关的数据结构和算法的 `crossbeam`，以及用于撰写解析器的 `nom` 及其他。



serde

每一个从其他语言转移到 Rust 的开发者，都会惊叹于 [serde](#) 及其周边库的强大能力。只消在数据结构上使用 `#[derive(Serialize, Deserialize)]` 宏，你的数据结构就能够被序列化和反序列化成绝大多数格式：[JSON](#) / [YAML](#) / [TOML](#) / [MsgPack](#) / [CSV](#) / [Bincode](#) 等等。

你还可以为自己的格式撰写对 serde 的支持，比如使用 DynamoDB，你可以用 [serde_dynamo](#)：

```
#[derive(Serialize, Deserialize)]
pub struct User {
    id: String,
    name: String,
    age: u8,
}

// Get documents from DynamoDB
let input = ScanInput {
    table_name: "users".to_string(),
    ..ScanInput::default()
};
let result = client.scan(input).await?;

if let Some(items) = result.items {
    // 直接一句话，就拿到 User 列表
    let users: Vec<User> = serde_dynamo::from_items(items)?;
    println!("Got {} users", users.len());
}
```

如果你用过其它语言的 ORM，那么，你可以把 serde 理解成增强版的、普适性的 ORM，它可以把任意可序列化的数据结构，序列化成任意格式，或者从任意格式中反序列化。

那么什么不是“可序列化的数据结构”呢？很简单，**任何状态无法简单重建的数据结构**，比如一个 `TcpStream`、一个文件描述符、一个 `Mutex`，**是不可序列化的**，而一个 `HashMap<String, Vec>` 是可序列化的。

tokio

如果你要用 Rust 处理高性能网络，那么 [tokio](#) 以及 tokio 的周边库，不能不了解。

tokio 在 Rust 中的地位，相当于 Golang 处理并发的运行时，只不过 Golang 的开发者没得选用不用运行时，而 Rust 开发者可以不用任何运行时，或者在需要的时候有选择地引入 [tokio](#) / [async-std](#) / [smol](#) 等。

在所有这些运行时中，最通用使用最广的是 tokio，围绕着它有：[tonic](#) / [axum](#) / [tokio-uring](#) / [tokio-rustls](#) / [tokio-stream](#) / [tokio-util](#) 等网络和异步 IO 库，以及 [bytes](#) / [tracing](#) / [prost](#) / [mio](#) / [slab](#) 等。我们在介绍[如何阅读 Rust 代码](#)时，简单读了 bytes，在 KV server 的撰写过程中，也遇到了这里提到的很多库。

thiserror / anyhow

错误处理的两个库 thiserror / anyhow 建议掌握，目前 Rust 生态里它们是最主流的错误处理工具。

如果你对它们的使用还不太了解，可以再回顾一下[错误处理](#)那堂课，并且看看在 KV server 中，我们是如何使用 thiserror 和 anyhow 的。

clap / structopt / dialoguer / indicatif

clap 和 structopt 依旧是 Rust 命令行处理的主要选择，其中 clap 3 已经整合了 structopt，所以，一旦它发布正式版本，structopt 的用户可以放心切换过去。

如果你要做交互式的命令行，dialoguer 是一个不错的选择。如果你希望在命令行中还能提供友好的进度条，试试 indicatif。

futures/async-trait

虽然我们还没有正式学习 future，但已经在很多场合使用过 [futures](#) 库和 [async-trait](#) 库。

标准库中已经采纳了 futures 库的 Future trait，并通过 async/await 关键字，使异步处理成为语言的一部分。然而，futures 库中还有很多其它重要的 trait 和数据结构，比如我们之前使用过的 Stream / Sink。futures 库还自带一个简单的 executor，可以在测试时取代 tokio。

async-trait 库顾名思义，就是为了解决 Rust 目前还不支持在 trait 中带有 async fn 的问题。

crossbeam

[crossbeam](#) 是 Rust 下一个非常优秀的处理并发，以及和并发相关的数据结构的库。当你需要撰写自己的调度器时，可以考虑使用 `deque`，当你需要性能更好的 MPMC channel 时，可以使用 `channel`，当你需要一个 epoch-based GC 时，可以使用 `epoch`。

nom/pest/combine

这三者都是非常优秀的 parser 库，可以用来撰写高效的解析器。

在 Rust 下，当你需要处理某些文件格式时，首先可以考虑 `serde`，其次可以考虑这几个库；如果你要处理语法，那么它们是最好的选择。我个人偏爱 `nom`，其次是 `combine`，它们是 parser combinator 库，`pest` 是 PEG 库，你可以用类似 EBNF 的结构定义语法，然后访问生成的代码。

Web 和 Web 服务开发

虽然 Rust 相对很多语言要年轻很多，但 Rust 下 Web 开发工具厮杀的惨烈程度一点也不亚于 Golang / Python 等更成熟的语言。

从 Web 协议支持的角度看，Rust 有 `hyper` 处理 http1/http2，`quinn` / `quiche` 处理 QUIC/http3，`tonic` 处理 gRPC，以及 `tungstenite` / `tokio-tungstenite` 处理 websocket。

从协议序列化/反序列化的角度看，Rust 有 `avro-rs` 处理 apache avro，`capnp` 处理 Cap'n Proto，`prost` 处理 protobuf，`flatbuffers` 处理 google flatbuffers，`thrift` 处理 apache thrift，以及 `serde_json` 处理我们最熟悉的 JSON。

一般来说，如果你提供 REST / GraphQL API，JSON 是首选的序列化工具，如果你提供二进制协议，没有特殊情况（比如做游戏，倾向于 flatbuffers），建议使用 protobuf。

从 Web 框架的角度，有号称性能宇宙第一的 `actix-web`；有简单好用且即将支持异步，性能会大幅提升的 `rocket`；还有 tokio 社区刚刚发布没多久的后起之秀 `axum`。

在 get hands dirty 用 Rust 实现 thumbor 的过程中，我们使用了 axum。如果你喜欢 Django 这样的大而全的 Web 框架，可以尝试 rocket 0.5 及以上版本。如果你特别在意 Web 性能，可以考虑 actix-web。

从数据库的支持角度看，Rust 支持几乎所有主流的数据库，包括但不限于 MySQL、Postgres、Redis、 RocksDB、Cassandra、MongoDB、ScyllaDB、CouchDB 等等。如果你喜欢使用 ORM，可以用 `diesel`，或者 `sea-orm`。如果你享受直接但安全的 SQL 查询，可以使用 `sqlx`。

从模板引擎的角度，Rust 有支持 jinja 语法的 [askama](#)，有类似 jinja2 的 [tera](#)，还有处理 markdown 的 [comrak](#)。

从 Web 前端的角度，Rust 有纯前端的 [yew](#) 和 [seed](#)，以及更偏重全栈的 [MoonZoon](#)。其中，yew 更加成熟一些，熟悉 react/elm 的同学更容易用得起来。

从 Web 测试的角度看，Rust 有对标 puppeteer 的 [headless_chrome](#)，以及对标 selenium 的 [thirtyfour](#) 和 [fantoccini](#)。

从云平台部署的角度看，Rust 有支持 aws 的 [rusoto](#) 和 [aws-sdk-rust](#)、azure 的 [azure-sdk-for-rust](#)。目前 Google Cloud、阿里云、腾讯云还没有官方的 SDK 支持。

在静态网站生成领域，Rust 有对标 hugo 的 [zola](#) 和对标 gitbook 的 [mdbook](#)。它们都是非常成熟的产品，可以放心使用。

客户端开发

这里的客户端，我特指带 GUI 的客户端开发。CLI 在[之前](#)已经提及，就不多介绍了。

在 [areweguiyet.com](#) 页面中，我们可以看到大量的 GUI 库。我个人觉得比较有前景的跨平台解决方案是 [tauri](#)、[druid](#)、[iced](#) 和 [sixtyfps](#)。

其中，tauri 是 electron 的替代品，如果你厌倦了 electron 庞大的身躯和贪婪的内存占用，但又喜欢使用 Web 技术栈构建客户端 GUI，那么可以试试 tauri，它使用了系统自身的 webview，再加上 Rust 本身极其克制的内存使用，性能和内存使用能甩 electron 好几个身位。

剩下三个都是提供原生 GUI，其中 sixtyfps 是一个非常不错的对嵌入式系统有很好支持的原生 GUI 库，不过要注意它的授权是 GPLv3，在商业产品上要谨慎使用（它有商业授权）。

如果你希望能够创建更加丰富，更加出众的 GUI，你可以使用 [skia-safe](#) 和 [tiny-skia](#)。前者是 Google 的 skia 图形引擎的 rust binding，后者是兼容 skia 的一个子集。skia 是目前在跨平台 GUI 领域炙手可热的 Flutter 的底层图形引擎，通过它你可以做任何复杂的对图层的处理。

当然，你也可以用 Flutter 绘制 UI，用 Rust 构建逻辑层。Rust 可以输出 C FFI，dart 可以生成 C FFI 的包装，供 Flutter 使用。

云原生开发

云原生一直是 Golang 的天下，如果你统计用到的 Kubernetes 生态中的 operator，几乎清一色是使用 Golang 撰写的。

然而，Rust 在这个领域渐渐有冒头的趋势。这要感谢之前提到的 serde，以及处理 Kubernetes API 的 [kube-rs](#) 项目做出的巨大努力，还有 Rust 强大的宏编程能力，它使得我们跟 Kubernetes 打交道无比轻松。

举个例子，比如要构建一个 CRD：

```
use kube::{CustomResource, CustomResourceExt};
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};

// Book 作为一个新的 Custom resource
#[derive(CustomResource, Debug, Clone, Serialize, Deserialize, JsonSchema)]
#[kube(group = "k8s.tyr.app", version = "v1", kind = "Book", namespaced)]
pub struct BookSpec {
    pub title: String,
    pub authors: Option<Vec<String>>,
}

fn main() {
    let book = Book::new(
        "rust-programming",
        BookSpec {
            title: "Rust programming".into(),
            authors: Some(vec![("Tyr Chen".into(),)),
        },
    );
    println!("{}", serde_yaml::to_string(&Book::crd().unwrap()));
    println!("{}", serde_yaml::to_string(&book.unwrap()));
}
```

短短 20 行代码就创建了一个 crd，是不是干净利落，写起来一气呵成？

```
cargo run | kubectl apply -f -
Finished dev [unoptimized + debuginfo] target(s) in 0.14s
  Running `'/Users/tchen/.target/debug/k8s-controller`'
customresourcedefinition.apiextensions.k8s.io/books.k8s.tyr.app configured
book.k8s.tyr.app/rust-programming created
```

```
kubectl get crds
NAME      CREATED AT
books.k8s.tyr.app  2021-10-20T01:44:57Z
```

```
kubectl get book
```

```
NAME      AGE
rust-programming  5m22s
```

如果你用 Golang 的 kubebuilder 做过类似的事情，是不是发现 Golang 那些生成大量脚手架代码和大量 YAML 文件的过程，顿时就不香了？

虽然在云原生方面，Rust 还是个小弟，但这个小弟有着强大的降维打击能力。同样的功能，Rust 可以只用 Golang 大概 1/4-1/10 的代码完成功能，这得益于 Rust 宏编程的强大能力。

除了 kube 这样的基础库，Rust 还有刚刚崭露头角的 [krator](#) 和 [Krustlet](#)。krator 可以帮助你更好地构建 kubernetes operator。虽然 operator 并不太强调效率，但用更少的代码，完成更多的功能，还有更低的内存占用，我还是非常看好未来会有更多的 kubernetes operator 用 Rust 开发。

Krustlet 顾名思义，是用来替换 kubelet 的。Krustlet 使用了 [wasmtime](#) 作为数据平台（dataplane）的运行时，而非传统的 containerd。这也就意味着，你可以用更高效、更精简的 WebAssembly 来处理原本只能使用 container 处理的工作。

目前，WebAssembly 在云原生领域的使用还处在早期，生态还不够完善，但是它相对于厚重的 container 来说，绝对是一个降维打击。

云原生另一个主要的方向是 serverless。在这个领域，由于 amazon 开源了用 Rust 开发的高性能 micro VM [firecracker](#)，使得 Rust 在 serverless/FAAS 方面处于领先地位。

WebAssembly 开发

如果说 Web 开发，云原生是 Rust 擅长的领域，那么 WebAssembly 可以说是 Rust 主战场之一。

Rust 内置了 wasm32-unknown-unknown 作为编译目标，如果你没添加，可以用 rustup 添加，然后在编译的时候指明目标，就可以得到 wasm：

```
$ rustup target add wasm32-unknown-unknown
$ cargo build --target wasm32-unknown-unknown --release
```

你可以用 [wasm-pack](#) 和 [wasm-bindgen](#)，不但生成 wasm，同时还生成 ts/js 调用 wasm 的代码。你可以在 [rustwasm](#) 下找到更多相关的项目。

WebAssembly 社区一个很重要的组织是 [Bytecode Alliance](#)。前文提到的 [wasmtime](#) 就是他们的主要开源产品。[wasmtime](#) 可以让 WebAssembly 代码以沙箱的形式运行在服务器。

另外一个 WebAssembly 的运行时 [wasmer](#)，是 [wasmtime](#) 的主要竞争者。目前，WebAssembly 在服务器领域，尤其是 serverless / FaaS 领域，有着很大的发展空间。

嵌入式开发

如果你要用 Rust 做嵌入式开发，那么 [embedded WG](#) 不可不关注。

你也可以在 [Awesome embedded rust](#) 里找感兴趣的嵌入式开发工具。现在很多嵌入式开发其实不是纯粹的嵌入式设备开发，所以云原生、边缘计算、WebAssembly 也在这个领域有很多应用。比如被接纳为 CNCF sandbox 项目不久的 [akri](#)，它就是一个管理嵌入式设备的云原生项目。

机器学习开发

机器学习/深度学习是 Rust 很有潜力，但目前生态还很匮乏的领域。

Rust 有 [tensorflow](#) 的绑定，也有 [tch-rs](#) 这个 libtorch (PyTorch) 的绑定。除了这些著名的 ML 库的 Rust 绑定外，Rust 下还有对标 scikit-learn 的 [linfa](#)。

我觉得 Rust 在机器学习领域未来会有很大突破的地方能是 ML infra，因为最终 ML 构建出来的模型，还是需要一个高性能的 API 系统对外提供服务，而 Rust 将是目前这个领域的玩家们的主要挑战者。

小结：Rust 生态的未来

今天我们讲了 Rust 主要的几个方向上的生态。在我撰写这篇内容时，[crates.io](#) 上有差不多七万个 rust crate，足以涵盖我们工作中遇到的方方面面的需求。



目前 Rust 在 WebAssembly 开发领域处于领先，在 Web 和 Web 服务开发领域已经有非常扎实的基础，而在云原生领域正在奋起直追，后劲十足。这三个领域，加上机器学习领域，是未来几年主流的后端开发方向。

作为一门依旧非常年轻的语言，Rust 的生态还在蓬勃发展中。要知道 Rust 的异步开发是2019年底才进入到稳定版本，在这不到两年的时间里，就出现了大量优秀的、基于异步开发的库被创造出来。

如果给 Rust 更长的时间，我们会看到更多的高性能优秀库会用 Rust 创造，或者用 Rust 改写。

思考题

在今天提到的某个领域下，找一个你感兴趣的库，阅读它的文档，将其 clone 到本地，运行它的 examples，大致浏览一下它的代码。欢迎结合之前讲的[阅读源码的技巧](#)，分享自己的收获。

感谢你的收听，你已经完成Rust学习的第27次打卡。坚持学习，我们下节课见~

工具篇

vscode常用配置

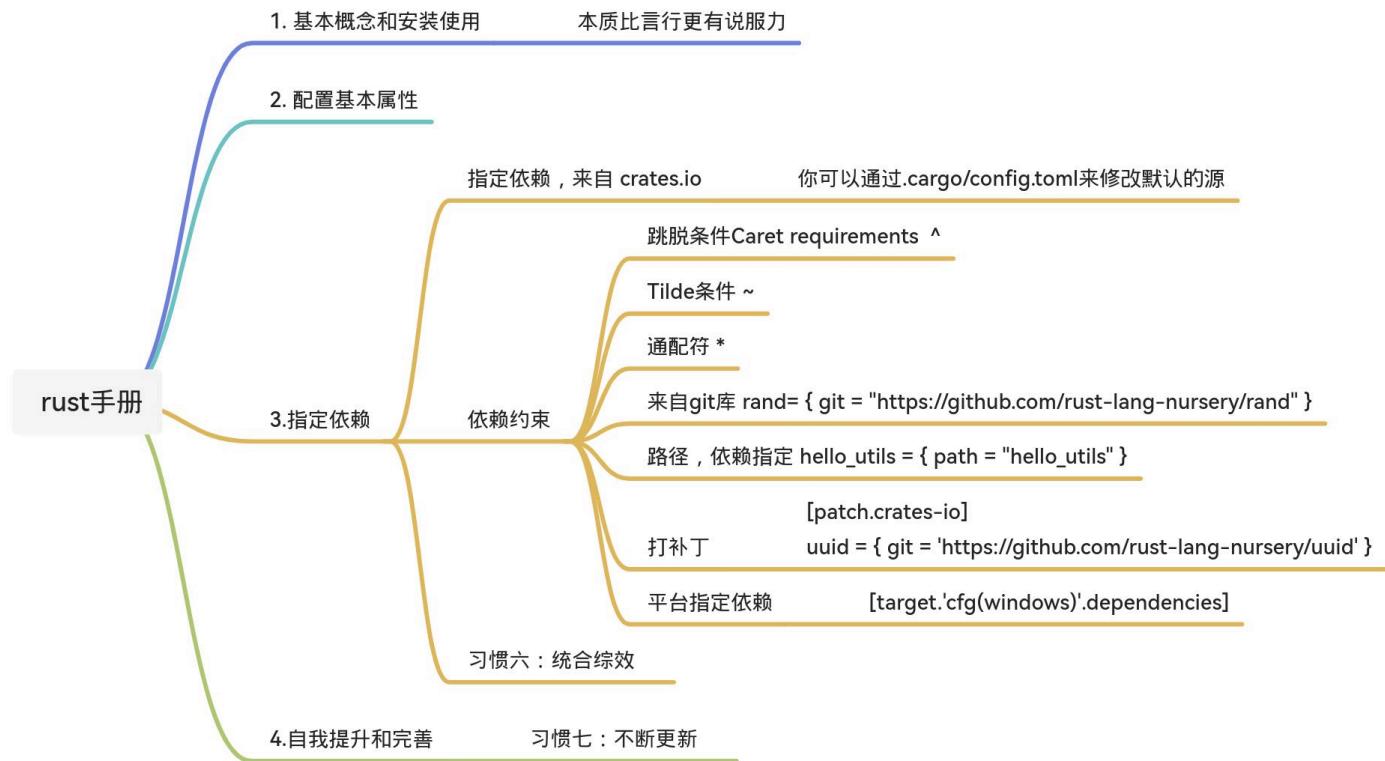
cargo手册

Cargo 是Rust的 包管理。Cargo 会下载您 Rust 的包依赖项，处理一些编译的参数，可重复的编译您的包，生成可分发的包，并将它们上传到crates.io .

书名	cargo手册
作者	Rust-lang
状态	待开始 阅读中 已读完
简介	https://llever.com/cargo-book-zh/index.zh.html https://rustwiki.org/zh-CN/cargo/index.html https://doc.rust-lang.org/stable/cargo/

思维导图

用思维导图，结构化记录本书的核心观点。



读后感

观点1

读完该书后，受益的核心观点与说明...

观点2

| 读完该书后，受益的核心观点与说明...

观点3

| 读完该书后，受益的核心观点与说明...

书摘

- 该书的金句摘录...
- 该书的金句摘录...
- 该书的金句摘录...

相关资料

| 可通过“+K”插入引用链接，或使用“本地文件”引入源文件。

<https://book.douban.com/subject/5325618/>

规范篇

基础库

<https://rustBuilderFactory.org/std/index.html>

标准库

标准库

标准库trait指南

转载自：<https://github.com/pretzelhammer/rust-blog/blob/master/posts/translations/zh-hans/tour-of-rusts-standard-library-traits.md>

Rust 标准库特性指南

2021年3月31日 · #rust · #traits

目录

- [引入 Intro](#引入-intro)

- [特性的基础知识 Trait Basics](#特性的基础知识–trait–basics)
- [特性的记号 Trait Items](#特性的记号–trait–items)
- [Self](#self)
- [函数 Functions](#函数–functions)
- [方法 Methods](#方法–methods)
- [关联类型 Associated Types](#关联类型–associated–types)
- [泛型参数 Generic Parameters](#泛型参数–generic–parameters)
- [泛型类型与关联类型 Generic Types vs Associated Types](#泛型类型与关联类型–generic–types–vs–associated–types)
- [作用域 Scope](#作用域–scope)
- [衍生宏 Derive Macros](#衍生宏–derive–macros)
- [默认实现 Default Impls](#默认实现–default–impls)
- [一揽子泛型实现 Generic Blanket Impls](#一揽子泛型实现–generic–blanket–impls)
- [子特性与超特性 Subtraits & Supertraits](#子特性与超特性–subtraits–supertraits)
- [特性对象 Trait Objects](#特性对象–trait–objects)
- [仅用于标记的特性 Marker Traits](#仅用于标记的特性–marker–traits)
- [可自动实现的特性 Auto Traits](#可自动实现的特性–auto–traits)
- [不安全的特性 Unsafe Traits](#不安全的特性–unsafe–traits)
- [可自动实现的特性 Auto Traits](#可自动实现的特性–auto–traits–1)
- [Send & Sync](#send–sync)
- [Sized](#sized)
- [常用特性 General Traits](#常用特性–general–traits)
- [Default](#default)
- [Clone](#clone)
- [Copy](#copy)
- [Any](#any)
- [文本格式化特性 Formatting Traits](#文本格式化特性–formatting–traits)
- [Display & ToString](#display–tostring)
- [Debug](#debug)
- [算符重载特性 Operator Traits](#算符重载特性–operator–traits)
- [比较特性 Comparison Traits](#比较特性–comparison–traits)

- [PartialEq & Eq](#partialeq--eq)
- [Hash](#hash)
- [PartialOrd & Ord](#partialord--ord)
- [算术特性 Arithmetic Traits](#算术特性-arithmetic-traits)
- [Add & AddAssign](#add--addassign)
- [闭包特性 Closure Traits](#闭包特性-closure-traits)
- [FnOnce, FnMut, & Fn](#fnonce--fnnut--fn)
- [其它特性 Other Traits](#其它特性-other-traits)
- [Deref & DerefMut](#deref--derefmut)
- [Index & IndexMut](#index--indexmut)
- [Drop](#drop)
- [转换特性 Conversion Traits](#转换特性-conversion-traits)
- [From & Into](#from--into)
- [错误处理 Error Handling](#错误处理-error-handling)
- [Error](#error)
- [转换特性深入 Conversion Traits Continued](#转换特性深入-conversion-traits-continued)
- [TryFrom & TryInto](#tryfrom--tryinto)
- [FromStr](#fromstr)
- [AsRef & AsMut](#asref--asmut)
- [Borrow & BorrowMut](#borrow--borrowmut)
- [ToOwned](#toowned)
- [迭代特性 Iteration Traits](#迭代特性-iteration-traits)
- [Iterator](#iterator)
- [IntoIterator](#intoiterator)
- [FromIterator](#fromiterator)
- [输入输出特性 I/O Traits](#输入输出特性-io-traits)
- [Read & Write](#read--write)
- [结语 Conclusion](#结语-conclusion)
- [讨论 Discuss](#讨论-discuss)
- [通告 Notifications](#通告-notifications)

- [更多资料 Further Reading](#更多资料–further–reading)
- [翻译 Translation](#翻译–translation)

引入 Intro

> Have you ever wondered what's the difference between:

- > - `Deref<Target = T>` , `AsRef<T>` , and `Borrow<T>`?
- > - `Clone` , `Copy` , and `ToOwned`?
- > - `From<T>` and `Into<T>`?
- > - `TryFrom<&str>` and `FromStr`?
- > - `FnOnce` , `FnMut` , `Fn` , and `fn`?
- >

你是否曾对以下特性的区别感到困惑:

- `Deref<Target = T>` , `AsRef<T>` 和 `Borrow<T>`?
- `Clone` , `Copy` 和 `ToOwned`?
- `From<T>` 和 `Into<T>`?
- `TryFrom<&str>` 和 `FromStr`?
- `FnOnce` , `FnMut` , `Fn` 和 `fn`?

> Or ever asked yourself the questions:

- > - _"When do I use associated types vs generic types in my trait?"_
- > - _"What are generic blanket impls?"_
- > - _"How do subtraits and supertraits work?"_
- > - _"Why does this trait not have any methods?"_
- >

或者有这样的疑问:

- “我应该在特性中使用关联类型还是泛型类型？”
- “什么是一揽子泛型实现？”
- “子特性与超特性是如何工作的？”
- “为什么某个特性没有实现任何方法？”

> Well then this is the article for you! It answers all of the above questions and much much more. Together we'll do a quick flyby tour of all of the most popular and commonly used traits from the Rust standard library!

>

本文正是为你解答以上困惑而撰写！而且本文绝不仅仅只回答了以上问题。下面，我们将一起对 Rust 标准库中所有最流行、最常用的特性做一个走马观花般的概览！

> You can read this article in order section by section or jump around to whichever traits interest you the most because each trait section begins with a list of links to **Prerequisite** sections that you should read to have adequate context to understand the current section's explanations.

>

你可以按顺序阅读本文，也可以直接跳读至你最感兴趣的特性。每节都会提供**预备知识**列表，它会帮助你获得相应的背景知识，不必担心跳读带来的理解困难。

特性的基础知识 Trait Basics

> We'll cover just enough of the basics so that the rest of the article can be streamlined without having to repeat the same explanations of the same concepts over and over as they reappear in different traits.

>

本章覆盖了特性的基础知识，相应内容在以后的章节中不再赘述。

特性的记号 Trait Items

> Trait items are any items that are part of a trait declaration.

特性的记号指的是，在特性的声明中可使用的记号。

Self

> `Self` always refers to the implementing type.

`Self` 永远引用正被实现的类型。

```rust

```
trait Trait {
 // always returns i32
 // 总是返回 i32
 fn returns_num() -> i32;

 // returns implementing type
 // 总是返回正被实现的类型
 fn returns_self() -> Self;
}
```

```
struct SomeType;
```

```
struct OtherType;
```

```
impl Trait for SomeType {
 fn returns_num() -> i32 {
```

```
5
}

// Self == SomeType

fn returns_self() -> Self {
 SomeType
}
```

```
impl Trait for OtherType {
 fn returns_num() -> i32 {
 6
 }
}
```

```
// Self == OtherType

fn returns_self() -> Self {
 OtherType
}

```

```

函数 Functions

> A trait function is any function whose first parameter does not use the `self` keyword.

特性的函数指的是，任何不以 `self` 关键字作为首参数的函数。

```
```rust
```

```
trait Default {
 // function
 // 函数
 fn default() -> Self;
}
...
...
```

> Trait functions can be called namespaced by the trait or implementing type:

特性的函数同时声明在特性本身以及具体实现类型的命名空间中。

```
```rust  
fn main() {  
    let zero: i32 = Default::default();  
    let zero = i32::default();  
}  
...  
...
```

方法 Methods

> A trait method is any function whose first parameter uses the `self` keyword and is of type `Self`, `&Self`, `&mut Self`. The former types can also be wrapped with a `Box`, `Rc`, `Arc`, or `Pin`.

特性的方法指的是，任何以 `self` 关键字作为首参数的函数，其类型是 `Self`，`&Self` 或 `&mut Self`。前者的类型也可以包裹在 `Box`，`Rc`，`Arc` 或 `Pin` 中。

```
```rust  
trait Trait {
 // methods
 // 方法
```

```
fn takes_self(self);
fn takes_immut_self(&self);
fn takes_mut_self(&mut self);

// above methods desugared
// 以上代码等价于
fn takes_self(self: Self);
fn takes_immut_self(self: &Self);
fn takes_mut_self(self: &mut Self);
}
```

```
// example from standard library
// 来自于标准库的示例
trait ToString {
 fn to_string(&self) -> String;
}
...
```

> Methods can be called using the dot operator on the implementing type:

可以使用点算符在具体实现类型上调用方法：

```
```rust  
fn main() {  
    let five = 5.to_string();  
}  
...
```

> However, similarly to functions, they can also be called namespaced by the trait or implementing type:

并且，与函数相似地，方法也声明在特性本身以及具体实现类型的命名空间中。

```
```rust
fn main() {
 let five = ToString::to_string(&5);
 let five = i32::to_string(&5);
}
```

```

关联类型 Associated Types

> A trait can have associated types. This is useful when we need to use some type other than `Self` within function signatures but would still like the type to be chosen by the implementer rather than being hardcoded in the trait declaration:

特性内部可以声明关联类型。当我们希望在特性函数的签名中使用某种 `Self` 以外的类型，又不希望硬编码这种类型，而是希望后来的实现该特性的程序员来选择该类型具体是什么的时候，关联类型会很有用。

```
```rust
trait Trait {
 type AssociatedType;
 fn func(arg: Self::AssociatedType);
}

struct SomeType;
struct OtherType;

// any type implementing Trait can
// choose the type of AssociatedType
// 我们可以在实现 Trait 特性的时候

```

```

// 再决定 AssociatedType 的具体类型
// 而不必是在声明 Trait 特性的时候

impl Trait for SomeType {

 type AssociatedType = i8; // chooses i8

 fn func(arg: Self::AssociatedType) {}

}

impl Trait for OtherType {

 type AssociatedType = u8; // chooses u8

 fn func(arg: Self::AssociatedType) {}

}

fn main() {

 SomeType::func(-1_i8); // can only call func with i8 on SomeType
 OtherType::func(1_u8); // can only call func with u8 on OtherType
 // 同一特性实现在不同类型上时，可以具有不同的函数签名
}
```

```

泛型参数 Generic Parameters

> _"Generic parameters"_ broadly refers to generic type parameters, generic lifetime parameters, and generic const parameters. Since all of those are a mouthful to say people commonly abbreviate them to _"generic types"_ , _"lifetimes"_ , and _"generic consts"_ . Since generic consts are not used in any of the standard library traits we'll be covering they're outside the scope of this article.

>

“泛型参数”是泛型类型参数、泛型寿命参数以及泛型常量参数的统称。由于这些术语过于佶屈聱牙，我们通常将他们缩略为“泛型类型”，“泛型寿命”和“泛型常量”。鉴于标准库中的特性无一采用泛型常量，本文也略过不讲。

> We can generalize a trait declaration using parameters:

我们可以使用以下参数来声明特性：

```rust

// trait declaration generalized with lifetime & type parameters

// 使用泛型寿命与泛型类型声明特性

trait Trait<'a, T> {

// signature uses generic type

// 在签名中使用泛型类型

fn func1(arg: T);

// signature uses lifetime

// 在签名中使用泛型寿命

fn func2(arg: &'a i32);

// signature uses generic type & lifetime

// 在签名中同时使用泛型类型与泛型寿命

fn func3(arg: &'a T);

}

struct SomeType;

impl<'a> Trait<'a, i8> for SomeType {

fn func1(arg: i8) {}

fn func2(arg: &'a i32) {}

fn func3(arg: &'a i8) {}

}

impl<'b> Trait<'b, u8> for SomeType {

```
fn func1(arg: u8) {}

fn func2(arg: &'b i32) {}

fn func3(arg: &'b u8) {}

}

```

```

> It's possible to provide default values for generic types. The most commonly used default value is `Self` but any type works:

可以为泛型类型指定默认值，最常用的默认值是 `Self`，此外任何其它类型都是可以的。

```rust

```
// make T = Self by default

// T 的默认值是 Self

trait Trait<T = Self> {

fn func(t: T) {}

}
```

```
// any type can be used as the default

// 任何其它类型都可用作默认值

trait Trait2<T = i32> {

fn func2(t: T) {}

}
```

```
struct SomeType;
```

```
// omitting the generic type will

// cause the impl to use the default

// value, which is Self here

// 省略泛型类型时，impl 块使用默认值，在这里是 Self

impl Trait for SomeType {
```

```

fn func(t: SomeType) {}

// default value here is i32
// 这里的默认值是 i32

impl Trait2 for SomeType {
 fn func2(t: i32) {}
}

// the default is overridable as we'd expect
// 默认值可以被重写，正如我们希望的那样

impl Trait<String> for SomeType {
 fn func(t: String) {}
}

// overridable here too
// 这里也可以重写

impl Trait2<String> for SomeType {
 fn func2(t: String) {}
}

...

```

> Aside from parameterizing the trait it's also possible to parameterize individual functions and methods:

不仅可以为特性提供泛型，也可以独立地为函数或方法提供泛型。

```

```rust
trait Trait {
    fn func<'a, T>(t: &'a T);
}

```

...

泛型类型与关联类型 Generic Types vs Associated Types

> Both generic types and associated types defer the decision to the implementer on which concrete types should be used in the trait's functions and methods, so this section seeks to explain when to use one over the other.

通过使用泛型类型与关联类型，我们都可以将具体类型的选择问题抛给后来实现该特性的程序员来决定，这一节将解释我们如何在相似的两者之间做出选择。

> The general rule-of-thumb is:

- > – Use associated types when there should only be a single impl of the trait per type.
- > – Use generic types when there can be many possible impls of the trait per type.

>

按照惯常的经验：

- 对于某一特性，每个类型仅应当有单一实现时，使用关联类型。
- 对于某一特性，每个类型可以有多个实现时，使用泛型类型。

> Let's say we want to define a trait called `Add` which allows us to add values together. Here's an initial design and impl that only uses associated types:

例如，我们声明一个 `Add` 特性，它允许将各值加总在一起。这是仅使用关联类型的初始设计：

```rust

```
trait Add {
 type Rhs;
 type Output;
 fn add(self, rhs: Self::Rhs) -> Self::Output;
}
```

```

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

impl Add for Point {
 type Rhs = Point;
 type Output = Point;
 fn add(self, rhs: Point) -> Point {
 Point {
 x: self.x + rhs.x,
 y: self.y + rhs.y,
 }
 }
}

fn main() {
 let p1 = Point { x: 1, y: 1 };
 let p2 = Point { x: 2, y: 2 };
 let p3 = p1.add(p2);
 assert_eq!(p3.x, 3);
 assert_eq!(p3.y, 3);
}
```

```

> Let's say we wanted to add the ability to add `i32`s to `Point`s where the `i32` would be added to both the `x` and `y` members:

例如，我们希望程序允许将 i32 类型的值与 Point 类型的值相加，其规则是该 i32 类型的值分别加到成员 `x` 与成员 `y`。

```
```rust
```

```
trait Add {
 type Rhs;
 type Output;
 fn add(self, rhs: Self::Rhs) -> Self::Output;
}
```

```
struct Point {
```

```
 x: i32,
 y: i32,
}
```

```
impl Add for Point {
```

```
 type Rhs = Point;
 type Output = Point;
 fn add(self, rhs: Point) -> Point {
 Point {
 x: self.x + rhs.x,
 y: self.y + rhs.y,
 }
 }
}
```

```
impl Add for Point { //
```

```
 type Rhs = i32;
 type Output = Point;
 fn add(self, rhs: i32) -> Point {
 Point {
 x: self.x + rhs,
 y: self.y + rhs,
 }
 }
}
```

```
}

}

}
```

```
fn main() {

let p1 = Point { x: 1, y: 1 };

let p2 = Point { x: 2, y: 2 };

let p3 = p1.add(p2);

assert_eq!(p3.x, 3);

assert_eq!(p3.y, 3);
```

```
let p1 = Point { x: 1, y: 1 };

let int2 = 2;

let p3 = p1.add(int2); //

assert_eq!(p3.x, 3);

assert_eq!(p3.y, 3);

}
```

```
...
```

> Throws:

编译出错:

```
```none
error[E0119]: conflicting implementations of trait `Add` for type `Point`:
--> src/main.rs:23:1
|
12 | impl Add for Point {
| ----- first implementation here
|
...```

```

```
23 | impl Add for Point {  
| ^^^^^^^^^^^^^^^^^^ conflicting implementation for `Point`  
| ...
```

> Since the `Add` trait is not parameterized by any generic types we can only impl it once per type, which means we can only pick the types for both `Rhs` and `Output` once! To allow adding both `Points`s and `i32`s to `Point` we have to refactor `Rhs` from an associated type to a generic type, which would allow us to impl the trait multiple times for `Point` with different type arguments for `Rhs`:

由于 `Add` 特性未提供泛型类型，因而每个类型只能具有该特性的单一实现，这即是说一旦我们指定了 `Rhs` 和 `Output` 的类型后就不可再更改了！为了 Point 类型的值能同时接受 i32 类型和 Point 类型的值作为被加数，我们应当重构之以将 `Rhs` 从关联类型改为泛型类型，这将允许我们为 `Rhs` 指定不同的类型并为同一类型多次实现某一特性。

```rust

```
trait Add<Rhs> {
 type Output;
 fn add(self, rhs: Rhs) -> Self::Output;
}
```

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

```
impl Add<Point> for Point {
 type Output = Self;
 fn add(self, rhs: Point) -> Self::Output {
 Point {
 x: self.x + rhs.x,
 y: self.y + rhs.y,
 }
 }
}
```

```

impl Add<i32> for Point { //
 type Output = Self;
 fn add(self, rhs: i32) -> Self::Output {
 Point {
 x: self.x + rhs,
 y: self.y + rhs,
 }
 }
}

fn main() {
 let p1 = Point { x: 1, y: 1 };
 let p2 = Point { x: 2, y: 2 };
 let p3 = p1.add(p2);
 assert_eq!(p3.x, 3);
 assert_eq!(p3.y, 3);

 let p1 = Point { x: 1, y: 1 };
 let int2 = 2;
 let p3 = p1.add(int2); //
 assert_eq!(p3.x, 3);
 assert_eq!(p3.y, 3);
}
```

```

> Let's say we add a new type called `Line` which contains two `Point`s, and now there are contexts within our program where adding two `Point`s should produce a `Line` instead of a `Point`. This is not possible given the current design of the `Add` trait where `Output` is an associated type but we can satisfy these new requirements by refactoring `Output` from an associated type into a generic type:

例如，我们现在声明一个包含两个 `Point` 类型的新类型 `Line`，要求当两个 `Point` 类型相加时返回 `Line` 而不是 `Point`。在当前 `Add` 特性的设计中 `Output` 是关联类型，不能满足这一要求，重构之以将关联类型改为泛型类型：

```rust

```
trait Add<Rhs, Output> {
 fn add(self, rhs: Rhs) -> Output;
}
```

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

```
impl Add<Point, Point> for Point {
 fn add(self, rhs: Point) -> Point {
 Point {
 x: self.x + rhs.x,
 y: self.y + rhs.y,
 }
 }
}
```

```
impl Add<i32, Point> for Point {
 fn add(self, rhs: i32) -> Point {
 Point {
 x: self.x + rhs,
 y: self.y + rhs,
 }
 }
}
```

```
struct Line {
 start: Point,
 end: Point,
}

impl Add<Point, Line> for Point { //
fn add(self, rhs: Point) -> Line {
 Line {
 start: self,
 end: rhs,
 }
}
}

fn main() {
let p1 = Point { x: 1, y: 1 };
let p2 = Point { x: 2, y: 2 };
let p3: Point = p1.add(p2);
assert!(p3.x == 3 && p3.y == 3);

let p1 = Point { x: 1, y: 1 };
let int2 = 2;
let p3 = p1.add(int2);
assert!(p3.x == 3 && p3.y == 3);

let p1 = Point { x: 1, y: 1 };
let p2 = Point { x: 2, y: 2 };
let l: Line = p1.add(p2); //
assert!(l.start.x == 1 && l.start.y == 1 && l.end.x == 2 && l.end.y == 2)
}
```

```

> So which `Add` trait above is the best? It really depends on the requirements of your program! They're all good in the right situations.

所以说，哪一种 `Add` 特性最好？答案是具体问题具体分析！不管白猫黑猫，会捉老鼠就是好猫。

作用域 Scope

> Trait items cannot be used unless the trait is in scope. Most Rustaceans learn this the hard way the first time they try to write a program that does anything with I/O because the `Read` and `Write` traits are not in the standard library prelude:

特性仅当被引入当前作用域时才可以使用。绝大多数的初学者要在编写 I/O 程序时经历一番痛苦挣扎后，才能领悟到这一点，原因是 `Read` 和 `Write` 两个特性并未包含在标准库的 prelude 模块中。

```rust

```
use std::fs::File;
use std::io;

fn main() -> Result<(), io::Error> {
 let mut file = File::open("Cargo.toml")?;
 let mut buffer = String::new();
 file.read_to_string(&mut buffer)?; // read_to_string not found in File
 // 当前文件中找不到 read_to_string
 Ok(())
}
```

> `read\_to\_string(buf: &mut String)` is declared by the `std::io::Read` trait and implemented by the `std::fs::File` struct but in order to call it `std::io::Read` must be in scope:

`read\_to\_string(buf: &mut String)` 声明于 `std::io::Read` 特性，并实现于 `std::fs::File` 类型，若要调用该函数还须得 `std::io::Read` 特性处于当前作用域中：

```
```rust
use std::fs::File;
use std::io;
use std::io::Read; //


fn main() -> Result<(), io::Error> {
    let mut file = File::open("Cargo.toml")?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?; // Ok(())
}
```
```

```

> The standard library prelude is a module in the standard library, i.e. `std::prelude::v1` , that gets auto imported at the top of every other module, i.e. `use std::prelude::v1::*` . Thus the following traits are always in scope and we never have to explicitly import them ourselves because they're part of the prelude:

诸如 `std::prelude::v1` , prelude 是标准库的一类模块，其特点是该模块命名空间下的成员将被自动导入到任何其它模块的顶部，其作用等效于 `use std::prelude::v1::*` 。因此，以下 prelude 模块中的特性无需我们显式导入，它们永远存在于当前作用域：

- [AsMut](#asref--asmut)
- [AsRef](#asref--asmut)
- [Clone](#clone)
- [Copy](#copy)
- [Default](#default)
- [Drop](#drop)
- [Eq](#partialeq--eq)
- [Fn](#fnonce-fnmut--fn)
- [FnMut](#fnonce-fnmut--fn)

- [FnOnce](#fnonce-fnmut--fn)
- [From](#from--into)
- [Into](#from--into)
- [ToOwned](#toowned)
- [Intoliterator](#intoiterator)
- [Iterator](#iterator)
- [PartialEq](#partialeq--eq)
- [PartialOrd](#partialord--ord)
- [Send](#send--sync)
- [Sized](#sized)
- [Sync](#send--sync)
- [ToString](#display--tostring)
- [Ord](#partialord--ord)

衍生宏 Derive Macros

> The standard library exports a handful of derive macros which we can use to quickly and conveniently impl a trait on a type if all of its members also impl the trait. The derive macros are named after the traits they impl:

标准库导出了一系列实用的衍生宏，我们可以利用它们方便快捷地为特定类型实现某种特性，前提是该类型的成员亦实现了相应的特性。衍生宏与它们各自所实现的特性同名：

- [Clone](#clone)
- [Copy](#copy)
- [Debug](#debug)
- [Default](#default)
- [Eq](#partialeq--eq)
- [Hash](#hash)
- [Ord](#partialord--ord)

- [PartialEq](#partialeq--eq)
- [PartialOrd](#partialord--ord)

> Example usage:

>

用例：

```
```rust
// macro derives Copy & Clone impl for SomeType
// 利用宏的方式为特定类型衍生出 Copy 与 Clone 特性的具体实现
#[derive(Copy, Clone)]
struct SomeType;
```
```

```

> Note: derive macros are just procedural macros and can do anything, there's no hard rule that they must impl a trait or that they can only work if all the members of the type impl a trait, these are just the conventions followed by the derive macros in the standard library.

>

注意：衍生宏仅是一种机械的过程，宏展开之后发生的事情并无一定之规。并没有绝对的规定要求衍生宏展开之后必须要为类型实现某种特性，又或者它们必须要求该类型的所有成员都必须实现某种特性才能为当前类型实现该特性，这仅仅是在标准库衍生宏的编纂过程中逐渐约定俗成的规则。

### 默认实现 Default Impls

> Traits can provide default impls for their functions and methods.

特性可为函数与方法提供默认的实现。

```
```rust
```

```

```
trait Trait {
 fn method(&self) {
 println!("default impl");
 }
}

struct SomeType;
struct OtherType;

// use default impl for Trait::method
// 省略时使用默认实现
impl Trait for SomeType {}

impl Trait for OtherType {
 // use our own impl for Trait::method
 // 重写时覆盖默认实现
 fn method(&self) {
 println!("OtherType impl");
 }
}

fn main() {
 SomeType.method(); // prints "default impl"
 OtherType.method(); // prints "OtherType impl"
}
...
```

> This is especially handy if some of the trait methods can be implemented solely using other trait methods.

这对于实现特性中某些仅依赖于其它方法的方法来说极其方便。

```
```rust
```

```
trait Greet {  
    fn greet(&self, name: &str) -> String;  
    fn greet_loudly(&self, name: &str) -> String {  
        self.greet(name) + "!"  
    }  
}
```

```
struct Hello;
```

```
struct Hola;
```

```
impl Greet for Hello {  
    fn greet(&self, name: &str) -> String {  
        format!("Hello {}", name)  
    }  
    // use default impl for greet_loudly  
    // 省略时使用 greet_loudly 的默认实现  
}
```

```
impl Greet for Hola {  
    fn greet(&self, name: &str) -> String {  
        format!("Hola {}", name)  
    }  
    // override default impl  
    // 重写时覆盖 greet_loudly 的默认实现  
    fn greet_loudly(&self, name: &str) -> String {  
        let mut greeting = self.greet(name);  
        greeting.insert_str(0, "¡");  
        greeting + "!"  
    }  
}
```

```
}

fn main() {
    println!("{}", Hello.greet("John")); // prints "Hello John"
    println!("{}", Hello.greet_loudly("John")); // prints "Hello John!"
    println!("{}", Hola.greet("John")); // prints "Hola John"
    println!("{}", Hola.greet_loudly("John")); // prints "¡Hola John!"
}

```

```

> Many traits in the standard library provide default impls for many of their methods.

标准库中的许多特性都为它们的方法提供默认实现。

### ### 一揽子泛型实现 Generic Blanket Impls

> A generic blanket impl is an impl on a generic type instead of a concrete type. To explain why and how we'd use one let's start by writing an `is\_even` method for number types:

一揽子泛型实现是对泛型类型的实现，与之对应的是对特定类型的实现。我们将以 `is_even` 方法为例说明如何对数字类型实现一揽子泛型实现。

```rust

```
trait Even {
    fn is_even(self) -> bool;
}
```

```
impl Even for i8 {
    fn is_even(self) -> bool {
```

```

self % 2_i8 == 0_i8
}

}

impl Even for u8 {
fn is_even(self) -> bool {
self % 2_u8 == 0_u8
}
}

impl Even for i16 {
fn is_even(self) -> bool {
self % 2_i16 == 0_i16
}
}

// etc

```

```

#[test] //

fn test_is_even() {
assert!(2_i8.is_even());
assert!(4_u8.is_even());
assert!(6_i16.is_even());
// etc
}
```

```

> Obviously, this is very verbose. Also, all of our impls are almost identical. Furthermore, in the unlikely but still possible event that Rust decides to add more number types in the future we have to remember to come back to this code and update it with the new number types. We can solve all these problems using a generic blanket impl:

显而易见地，我们重复实现了近乎相同的逻辑，这非常的繁琐。进一步来讲，如果 Rust 在将来决定增加更多的数字类型（小概率事件并非绝不可能），那么我们将不得不重新回到这里对新增的数字类型编写代码。一揽子泛型实现恰可以解决这些问题：

```
```rust
```

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

```
use std::convert::TryInto;
```

```
use std::ops::Rem;
```

```
trait Even {
```

```
    fn is_even(self) -> bool;
```

```
}
```

```
// generic blanket impl
```

```
// 一揽子泛型实现
```

```
impl<T> Even for T
```

```
where
```

```
T: Rem<Output = T> + PartialEq<T> + Sized,
```

```
u8: TryInto<T>,
```

```
<u8 as TryInto<T>>::Error: Debug,
```

```
{
```

```
    fn is_even(self) -> bool {
```

```
        // these unwraps will never panic
```

```
        // 以下 unwrap 永远不会 panic
```

```
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
```

```
}
```

```
}
```

```
#[test] //
```

```
fn test_is_even() {
```

```
    assert!(2_i8.is_even());
```

```
assert!(4_u8.is_even());
assert!(6_i16.is_even());
// etc
}
```

```

> Unlike default impls, which provide \_an\_ impl, generic blanket impls provide \_the\_ impl, so they are not overridable.

默认实现可以重写，而一揽子泛型实现不可重写。

```rust

```
use std::fmt::Debug;
use std::convert::TryInto;
use std::ops::Rem;
```

```
trait Even {
    fn is_even(self) -> bool;
}
```

impl<T> Even for T

where

```
T: Rem<Output = T> + PartialEq<T> + Sized,
u8: TryInto<T>,
<u8 as TryInto<T>>::Error: Debug,
{
    fn is_even(self) -> bool {
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
    }
}
```

```
impl Even for u8 { //  
fn is_even(self) -> bool {  
    self % 2_u8 == 0_u8  
}  
}  
}  
...
```

> Throws:

编译出错:

```
```none  
error[E0119]: conflicting implementations of trait `Even` for type `u8`:
--> src/lib.rs:22:1
|
10 | / impl<T> Even for T
11 || where
12 || T: Rem<Output = T> + PartialEq<T> + Sized,
13 || u8: TryInto<T>,
... |
19 || }
20 || }
|_- first implementation here
21 |
22 | impl Even for u8 {
| ^^^^^^^^^^^^^^ conflicting implementation for `u8`
...
```

> These impls overlap, hence they conflict, hence Rust rejects the code to ensure trait coherence. Trait coherence is the property that there exists at most one impl of a trait for any given type. The rules Rust uses to enforce trait coherence, the implications of those rules, and workarounds for the implications are outside the scope of this article.

重叠的实现产生了冲突，于是 Rust 拒绝了该代码以确保特性一致性。特性一致性指的是，对任意给定类型，仅能对某一特性具有单一实现。Rust 强制实现特性一致性，而这一规则的潜在影响与变通方法超出了本文的讨论范围。

## ### 子特性与超特性 Subtraits & Supertraits

> The "sub" in "subtrait" refers to subset and the "super" in "supertrait" refers to superset. If we have this trait declaration:

子特性的“子”即为子集，超特性的“超”即为超集。若有下列特性声明：

```rust

trait Subtrait: Supertrait {}

11

> All of the types which impl `Subtrait` are a subset of all the types which impl `Supertrait`, or to put it in opposite but equivalent terms: all the types which impl `Supertrait` are a superset of all the types which impl `Subtrait`.

所有实现了子特性的类型都是实现了超特性的类型的子集，也可以说，所有实现了超特性的类型都是实现了子特性的类型的超集。

> Also, the above is just syntax sugar for:

以上代码等价于：

```rust

```
trait Subtrait where Self: Supertrait {}
```

11

> It's a subtle yet important distinction to understand that the bound is on `Self`, i.e. the type impling `Subtrait`, and not on `Subtrait` itself. The latter would not make any sense, since trait bounds can only be applied to concrete types which can impl traits. Traits cannot impl other traits:

这是一种易于忽略但又至关重要的区别 —— 约束是 `Self` 的约束，而不是 `Subtrait` 的约束。后者没有任何意义，因为特性约束只能应用于具体类型。不能用一种特性去实现其它特性：

```rust

```
trait Supertrait {  
    fn method(&self) {  
        println!("in supertrait");  
    }  
}
```

```
trait Subtrait: Supertrait {  
    // this looks like it might impl or  
    // override Supertrait::method but it  
    // does not  
  
    // 这可能会令你产生超特性的方法被覆盖的错觉（实际不会）  
    fn method(&self) {  
        println!("in subtrait")  
    }  
}
```

```
struct SomeType;
```

```
// adds Supertrait::method to SomeType  
impl Supertrait for SomeType {}  
  
// adds Subtrait::method to SomeType  
impl Subtrait for SomeType {}
```

```
// both methods exist on SomeType simultaneously  
// neither overriding or shadowing the other  
// 两个同名方法同时存在于同一类型时，既不重写也不影射  
  
fn main() {  
    SomeType.method(); // ambiguous method call  
    // 不允许语义模糊的函数调用  
    // must disambiguate using fully-qualified syntax  
    // 必须使用完全限定的记号来明确你要使用的函数  
    <SomeType as Supertrait>::method(&st); // prints "in supertrait"  
    <SomeType as Subtrait>::method(&st); // prints "in subtrait"  
}  
...  
  
}
```

> Furthermore, there are no rules for how a type must impl both a subtrait and a supertrait. It can use the methods from either in the impl of the other.

此外，对于特定类型如何同时实现子特性与超特性并没有规定。子、超特性之间的方法也可以相互调用。

```
```rust  
trait Supertrait {
 fn super_method(&mut self);
}

trait Subtrait: Supertrait {
 fn sub_method(&mut self);
}

struct CallSuperFromSub;
```

```
impl Supertrait for CallSuperFromSub {
 fn super_method(&mut self) {
 println!("in super");
 }
}
```

```
impl Subtrait for CallSuperFromSub {
 fn sub_method(&mut self) {
 println!("in sub");
 self.super_method();
 }
}
```

```
struct CallSubFromSuper;
```

```
impl Supertrait for CallSubFromSuper {
 fn super_method(&mut self) {
 println!("in super");
 self.sub_method();
 }
}
```

```
impl Subtrait for CallSubFromSuper {
 fn sub_method(&mut self) {
 println!("in sub");
 }
}
```

```
struct CallEachOther(bool);
```

```
impl Supertrait for CallEachOther {
fn super_method(&mut self) {
 println!("in super");

 if self.0 {
 self.0 = false;
 self.sub_method();
 }
}
}

impl Subtrait for CallEachOther {
fn sub_method(&mut self) {
 println!("in sub");

 if self.0 {
 self.0 = false;
 self.super_method();
 }
}
}

fn main() {
 CallSuperFromSub.super_method(); // prints "in super"
 CallSuperFromSub.sub_method(); // prints "in sub", "in super"

 CallSubFromSuper.super_method(); // prints "in super", "in sub"
 CallSubFromSuper.sub_method(); // prints "in sub"

 CallEachOther(true).super_method(); // prints "in super", "in sub"
 CallEachOther(true).sub_method(); // prints "in sub", "in super"
}
```

```

> Hopefully the examples above show that the relationship between subtraits and supertraits can be complex. Before introducing a mental model that neatly encapsulates all of that complexity let's quickly review and establish the mental model we use for understanding trait bounds on generic types:

通过以上示例，希望读者能够领会到，子特性与超特性之间的关系并未被一刀切的限制住。接下来我们将学习一种将所有这些复杂性巧妙地封装在一起的心智模型，在这之前我们先来回顾一下我们用来理解泛型类型与特性约束的关系的心智模型。

```rust

```
fn function<T: Clone>(t: T) {
 // impl
}

```
```

> Without knowing anything about the impl of this function we could reasonably guess that `t.clone()` gets called at some point because when a generic type is bounded by a trait that strongly implies it has a dependency on the trait. The mental model for understanding the relationship between generic types and their trait bounds is a simple and intuitive one: generic types depend on their trait bounds.

即便我们不知道这个函数的具体实现，我们仍旧可以有理有据地猜测 `t.clone()` 将在函数的某处被调用，因为当泛型类型被特性所约束的时候，会给人一种它依赖于该特性的强烈暗示。这就是一种理解泛型类型与特性约束的关系的心智模型，它简单且可凭直觉——泛型类型依赖于它们的特性约束。

> Now let's look the trait declaration for `Copy`:

现在，让我们看看 `Copy` 特性的声明：

```rust

```
trait Copy: Clone {}

```
```

> The syntax above looks very similar to the syntax for applying a trait bound on a generic type and yet `Copy` doesn't depend on `Clone` at all. The mental model we developed earlier doesn't help us here. In my opinion, the most simple and elegant mental model for understanding the relationship between subtraits and supertraits is: subtraits refine their supertraits.

以上的记号和之前我们为泛型添加特性约束的记号非常相似，但是 `Copy` 却完全不依赖 `Clone`。早前建立的心智模型现在不适用了。在我看来，理解子特性与超特性的关系的最简单和最优雅的心智模型莫过于——子特性 *改良* 了超特性。

> "Refinement" is intentionally kept somewhat vague because it can mean different things in different contexts:
> – a subtrait might make its supertrait's methods' impls more specialized, faster, use less memory, e.g. `Copy: Clone`
> – a subtrait might make additional guarantees about the supertrait's methods' impls, e.g. `Eq: PartialEq`, `Ord: PartialOrd`, `ExactSizeliterator: Iterator`
> – a subtrait might make the supertrait's methods more flexible or easier to call, e.g. `FnMut: FnOnce`, `Fn: FnMut`
> – a subtrait might extend a supertrait and add new methods, e.g. `DoubleEndedIterator: Iterator`, `ExactSizeliterator: Iterator`
>

“改良”一词故意地预留了一些模糊的空间，它的具体含义在不同的上下文中有所不同：

- 子特性可能比超特性的方法更加特异化、运行更快或使用更少内存等等，例如 `Copy: Clone`
- 子特性可能比超特性的方法具有额外的功能，例如 `Eq: PartialEq`， `Ord: PartialOrd` 和 `ExactSizeliterator: Iterator`
- 子特性可能比超特性的方法更灵活和更易于调用，例如 `FnMut: FnOnce` 和 `Fn: FnMut`
- 子特性可能扩展了超特性并添加了新的方法，例如 `DoubleEndedIterator: Iterator` 和 `ExactSizeliterator: Iterator`

特性对象 Trait Objects

> Generics give us compile-time polymorphism where trait objects give us run-time polymorphism. We can use trait objects to allow functions to dynamically return different types at run-time:

如果说泛型给了我们编译时的多态性，那么特性对象就给了我们运行时的多态性。通过特性对象，我们可以允许函数在运行时动态地返回不同的类型。

```
```rust
```

```
fn example(condition: bool, vec: Vec<i32>) -> Box<dyn Iterator<Item = i32>> {
 let iter = vec.into_iter();
 if condition {
```

```

// Has type:
// Box<Map<Intoler<i32>, Fn(i32) -> i32>>
// But is cast to:
// Box<dyn Iterator<Item = i32>>
Box::new(iter.map(|n| n * 2))

} else {
 // Has type:
 // Box<Filter<Intoler<i32>, Fn(&i32) -> bool>>
 // But is cast to:
 // Box<dyn Iterator<Item = i32>>
 Box::new(iter.filter(|&n| n >= 2))
}

}

// 以上代码中，两种不同的指针类型转换成相同的指针类型
```

```

> Trait objects also allow us to store heterogeneous types in collections:

特性对象也允许我们在集合中存储不同类型的值：

```

```rust
use std::f64::consts::PI;

struct Circle {
 radius: f64,
}

struct Square {
 side: f64
}

```

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

impl Shape for Circle {
 fn area(&self) -> f64 {
 PI * self.radius * self.radius
 }
}

impl Shape for Square {
 fn area(&self) -> f64 {
 self.side * self.side
 }
}

fn get_total_area(shapes: Vec<Box<dyn Shape>>) -> f64 {
 shapes.into_iter().map(|s| s.area()).sum()
}

fn example() {
 let shapes: Vec<Box<dyn Shape>> = vec![
 Box::new(Circle { radius: 1.0 }), // Box<Circle> cast to Box<dyn Shape>
 Box::new(Square { side: 1.0 }), // Box<Square> cast to Box<dyn Shape>
];
 assert_eq!(PI + 1.0, get_total_area(shapes)); //
}
```

```

> Trait objects are unsized so they must always be behind a pointer. We can tell the difference between a concrete type and a trait object at the type level based on the presence of the `dyn` keyword within the type:

特性对象的结构体大小是未知的，所以必须要通过指针来引用它们。具体类型与特性对象在字面上的区别在于，特性对象必须要用 `dyn` 关键字来修饰前缀，了解了这一点我们可以轻松辨别二者。

```rust

```
struct Struct;
```

```
trait Trait {}
```

```
// regular struct
```

```
// 这是一般的结构
```

```
&Struct
```

```
Box<Struct>
```

```
Rc<Struct>
```

```
Arc<Struct>
```

```
// trait objects
```

```
// 这是特性对象
```

```
&dyn Trait
```

```
Box<dyn Trait>
```

```
Rc<dyn Trait>
```

```
Arc<dyn Trait>
```

```
...
```

> Not all traits can be converted into trait objects. A trait is object-safe if it meets these requirements:

> – trait doesn't require `Self: Sized`

> – all of the trait's methods are object-safe

>

并非全部的特性都可以转换为特性对象，一个“对象安全”的特性必须满足：

- 该特性不要求 `Self: Sized`
- 该特性的所有方法都是“对象安全”的

> A trait method is object-safe if it meets these requirements:

- >
- > – method requires `Self: Sized` or
- > – method only uses a `Self` type in receiver position
- >

一个特性的方法若要是“对象安全”的，必须满足：

- 该方法要求 `Self: Sized`
- 该方法仅在接收参数中使用 `Self` 类型

Understanding why the requirements are what they are is not relevant to the rest of this article, but if you're still curious it's covered in [Sizedness in Rust](../../sizedness-in-rust.md).

关于具有这些限制条件的原因超出了本文的讨论范围且与下文无关，如果你对此深感兴趣不妨阅读 [Sizedness in Rust](../../sizedness-in-rust.md) 以了解详情。

### ### 仅用于标记的特性 Marker Traits

> Marker traits are traits that have no trait items. Their job is to "mark" the implementing type as having some property which is otherwise not possible to represent using the type system.

仅用于标记的特性，即是某种声明体为空的特性。它们存在的意义在于“标记”所实现的类型，且该类型具有某种类型系统所无法表达的属性。

```rust

```
// Imploring PartialEq for a type promises
```

```
// that equality for the type has these properties:  
// - symmetry: a == b implies b == a, and  
// - transitivity: a == b && b == c implies a == c  
// But DOES NOT promise this property:  
// - reflexivity: a == a  
  
// 为特定类型实现 PartialEq 特性确保了该类型的相等算符具有以下性质：  
// - 对称性：若有 a == b， 则必有 b == a  
// - 传递性：若有 a == b 和 b == c， 则必有 a == c  
// 但是不能确保具有以下性质：  
// - 自反性： a == a  
  
trait PartialEq {  
    fn eq(&self, other: &Self) -> bool;  
}  
  
// Eq has no trait items! The eq method is already  
// declared by PartialEq, but "impling" Eq  
// for a type promises this additional equality property:  
// - reflexivity: a == a  
  
// Eq 特性的声明体是空的！ 而 eq 方法已经被 PartialEq 所声明，  
// 但是对特定类型“实现”Eq 特性确保了额外的相等性质：  
// - 自反性： a == a  
  
trait Eq: PartialEq {}  
  
// f64 implements PartialEq but not Eq because NaN != NaN  
// i32 implements PartialEq & Eq because there's no NaNs :)  
// f64 实现了 PartialEq 特性但是没有实现 Eq 特性， 因为 NaN != NaN  
// i32 同时实现了 PartialEq 特性与 Eq 特性， 因为没有 NaN 来捣乱 :)  
...
```

可自动实现的特性 Auto Traits

> Auto traits are traits that get automatically implemented for a type if all of its members also impl the trait. What "members" means depends on the type, for example: fields of a struct, variants of an enum, elements of an array, items of a tuple, and so on.

可自动实现的特性指的是，存在这样一种特性，若给定类型的成员都实现了该特性，那么该类型就隐式地自动实现该特性。这里所说的“成员”依据上下文而具有不同的含义，包括而又不限于结构体的字段、枚举的变量、数组的元素和元组的内容等等。

> All auto traits are marker traits but not all marker traits are auto traits. Auto traits must be marker traits so the compiler can provide an automatic default impl for them, which would not be possible if they had any trait items.

所有可自动实现的特性都是仅用于标记的特性，反之则不是。正是由于可自动实现的特性必须是仅用于标记的特性，所以编译器才能够自动为其提供一个默认实现，反之编译器就无能为力了。

> Examples of auto traits:

可自动实现的特性的示例：

```
```rust
// implemented for types which are safe to send between threads
// 实现 Send 特性的类型可以安全地往返于多个线程
unsafe auto trait Send {}

// implemented for types whose references are safe to send between threads
// 实现 Sync 特性的类型，其引用可以安全地往返于多个线程
unsafe auto trait Sync {}

```

```

不安全的特性 Unsafe Traits

> Traits can be marked unsafe to indicate that impling the trait might require unsafe code. Both `Send` and `Sync` are marked `unsafe` because if they aren't automatically implemented for a type that means it must contains some non-`Send` or non-`Sync` member and we have to take extra care as the implementers to make sure there are no data races if we want to manually mark the type as `Send` and `Sync`.

以 `unsafe` 修饰前缀的特性，意味着该特性的实现可能需要不安全的代码。`Send` 特性与 `Sync` 特性以 `unsafe` 修饰前缀意味着，如果特定类型没有自动实现该特性，那么说明该类型的成员并非都实现了该特性，这提示着我们手动实现该特性一定要谨慎小心，以确保没有发生数据竞争。

```rust

```
// SomeType is not Send or Sync

// SomeType 没有实现 Send 和 Sync

struct SomeType {

 not_send_or_sync: *const (),
}
```

```
// but if we're confident that our impl doesn't have any data

// races we can explicitly mark it as Send and Sync using unsafe

// 倘若我们得以社会主义伟大成就的庇佑自信地写出没有数据竞争的代码

// 可以使用 unsafe 来修饰前缀，以显式地实现 Send 特性与 Sync 特性

unsafe impl Send for SomeType {}

unsafe impl Sync for SomeType {}

```

```

可自动实现的特性 Auto Traits

Send & Sync

预备知识

- [Marker Traits](#marker-traits)
- [Auto Traits](#auto-traits)
- [Unsafe Traits](#unsafe-traits)

```rust

```
unsafe auto trait Send {}
```

```
unsafe auto trait Sync {}
```

...

> If a type is `Send` that means it's safe to send between threads. If a type is `Sync` that means it's safe to share references of it between threads. In more precise terms some type `T` is `Sync` if and only if `&T` is `Send`.

实现 `Send` 特性的类型可以安全地往返于多线程。实现 `sync` 特性的类型，其引用可以安全地往返于多线程。用更加准确的术语来讲，当且仅当 `&T` 实现 `Send` 特性时，`T` 才能实现 `Sync` 特性。

> Almost all types are `Send` and `Sync`. The only notable `Send` exception is `Rc` and the only notable `Sync` exceptions are `Rc`, `Cell`, `RefCell`. If we need a `Send` version of `Rc` we can use `Arc`. If we need a `Sync` version of `Cell` or `RefCell` we can `Mutex` or `RwLock`. Although if we're using the `Mutex` or `RwLock` to just wrap a primitive type it's often better to use the atomic primitive types provided by the standard library such as `AtomicBool`, `AtomicI32`, `AtomicUsize`, and so on.

几乎所有类型都实现了 `Send` 特性和 `Sync` 特性。对于 `Send` 唯一需要注意的例外是 `Rc`，对于 `Sync` 唯三需要注意的例外是 `Rc`，`Cell` 和 `RefCell`。如果我们需要 `Send` 版的 `Rc`，可以使用 `Arc`。如果我们需要 `Sync` 版的 `Cell` 或 `RefCell`，可以使用 `Mutex` 或 `RwLock`。尽管我们可以使用 `Mutex` 或 `RwLock` 来包裹住原语类型，但通常使用标准库提供的原子原语类型会更好，诸如 `AtomicBool`，`AtomicI32` 和 `AtomicUsize` 等等。

> That almost all types are `Sync` might be a surprise to some people, but yup, it's true even for types without any internal synchronization. This is possible thanks to Rust's strict borrowing rules.

多亏了 Rust 严格的借用规则，几乎所有的类型都是 `Sync` 的。这对于一些人来讲可能会很惊讶，但事实胜于雄辩，甚至对于那些没有内部同步机制的类型来说也是如此。

> We can pass many immutable references to the same data to many threads and we're guaranteed there are no data races because as long as any immutable references exist Rust statically guarantees the underlying data cannot be mutated:

对于同一数据，我们可以放心地将该数据的多个不可变引用传递给多个线程，因为只要当前存在一个该数据的不可变引用，那么 Rust 就会静态地确保该数据不会被改变：

```
```rust
use crossbeam::thread;

fn main() {
    let mut greeting = String::from("Hello");
    let greeting_ref = &greeting;

    thread::scope(|scoped_thread| {
        // spawn 3 threads
        // 产生三个线程
        for n in 1..=3 {
            // greeting_ref copied into every thread
            // greeting_ref 被拷贝到每个线程
            scoped_thread.spawn(move || {
                println!("{} {}", greeting_ref, n); // prints "Hello {n}"
            });
        }
    });

    // line below could cause UB or data races but compiler rejects it
    // 下面这行代码可能导致数据竞争，于是编译器拒绝了它
    greeting += " world";
    // cannot mutate greeting while immutable refs exist
    // 当不可变引用存在时，不可以修改引用的数据
});

// can mutate greeting after every thread has joined
// 当所有的线程结束之后，可以修改数据
greeting += " world"; //
```

```
    println!("{}", greeting); // prints "Hello world"
}
...

```

> Likewise we can pass a single mutable reference to some data to a single thread and we're guaranteed there will be no data races because Rust statically guarantees aliased mutable references cannot exist and the underlying data cannot be mutated through anything other than the single existing mutable reference:

同样地，我们可以将某个数据的单个可变引用传递给单个线程，在此过程中不必担心出现数据竞争，因为 Rust 静态地确保了不存在其它可变引用。以下数据即仅可通过已经存在的单个可变引用而改变：

```
```rust
use crossbeam::thread;

fn main() {
 let mut greeting = String::from("Hello");
 let greeting_ref = &mut greeting;

 thread::scope(|scoped_thread| {
 // greeting_ref moved into thread
 // greeting_ref 移动到当前线程
 scoped_thread.spawn(move || {
 *greeting_ref += " world";
 println!("{}", greeting_ref); // prints "Hello world"
 });
 });

 // line below could cause UB or data races but compiler rejects it
 // 下面这行代码可能导致数据竞争，于是编译器拒绝了它
 greeting += "!!!";
 // cannot mutate greeting while mutable refs exist
 // 可变引用存在时不可改变数据
});
```

```
// can mutate greeting after the thread has joined

// 当所有的线程结束之后，可以修改数据

greeting += "!!!"; //

println!("{}", greeting); // prints "Hello world!!!"

}

...
```

> This is why most types are `Sync` without requiring any explicit synchronization. In the event we need to simultaneously mutate some data `T` across multiple threads the compiler won't let us until we wrap the data in a `Arc<Mutex<T>>` or `Arc<RwLock<T>>` so the compiler enforces that explicit synchronization is used when it's needed.

>

这就是为什么绝大多数的类型都是 Sync 的而不需要实现任何显式的同步机制。对于数据 T，如果我们试图从多个线程同时修改的话，编译器会对我们作出警告，除非我们将数据包裹在 `Arc<Mutex<T>>` 或 `Arc<RwLock<T>>` 中。所以说，当我们真的需要显式的同步机制时，编译器会强制要求我们这样做的。

### Sized

## 预备知识

- [Marker Traits](#marker-traits)
- [Auto Traits](#auto-traits)

> If a type is `Sized` that means its size in bytes is known at compile-time and it's possible to put instances of the type on the stack.

如果一个类型实现了 `Sized`，那么说明该类型具体大小的字节数在编译时可以确定，并且也就说明该类型的实例可以存放在栈上。

> Sizedness of types and its implications is a subtle yet huge topic that affects a lot of different aspects of the language. It's so important that I wrote an entire article on it called [Sizedness in Rust](../../sizedness-in-rust.md) which I highly recommend reading for anyone who would like to understand sizedness in-depth. I'll summarize a few key things which are relevant to this article.

类型的大小以及其所带来的潜在影响，是一个易于忽略但是又十分宏大的话题，它深刻地影响着本门语言的诸多方面。鉴于它的的重要性，我已经写了一整篇文章 ([\[Sizedness in Rust\]\(../../sizedness-in-rust.md\)](#)) 来具体阐述其内容，我高度推荐对于希望深入 sizedness 的人阅读此篇文章。下面是此篇文章的要点：

> 1. All generic types get an implicit `Sized` bound.

1. 所有的泛型类型都具有隐式的 `Sized` 约束。

```rust

```
fn func<T>(t: &T) {}
```

```
// example above desugared
```

// 以上代码等价于

```
fn func<T: Sized>(t: &T) {}
```

...

> 2. Since there's an implicit `Sized` bound on all generic types, if we want to opt-out of this implicit bound we need to use the special _"relaxed bound"_ syntax `?Sized` which currently only exists for the `Sized` trait:

2. 由于所有的泛型类型都具有隐式的 `Sized` 约束，如果我们希望摆脱这样的隐式约束，那么我们需要使用特殊的 *“宽松约束”* 记号 `?Sized`，目前这样的记号仅适用于 `Sized` 特性：

```rust

```
// now T can be unsized
```

// 现在 T 的大小可以是未知的

```
fn func<T: ?Sized>(t: &T) {}
```

...

> 3. There's an implicit `?Sized` bound on all traits.

3. 所有的特性都具有隐式的 `?Sized` 约束。

```
```rust
trait Trait {}

// example above desugared
// 以上代码等价于

trait Trait: ?Sized {}

```

```

> This is so that trait objects can impl the trait. Again, all of the nitty gritty details are in [Sizedness in Rust](../../sizedness-in-rust.md).

这就是为什么特性对象可以实现具体特性。再次，向您推荐关于一切真相的[Sizedness in Rust](../../sizedness-in-rust.md)。

## ## 常用特性 General traits

### ### Default

#### 预备知识

- [Self](#self)
- [Functions](#functions)
- [Derive Macros](#derive-macros)

```
```rust
```

```
trait Default {

fn default() -> Self;

}
```

...

> It's possible to construct default values of `Default` types.

为特定类型实现 `Default` 特性时，即为该类型赋予了可选的默认值。

```rust

```
struct Color {
```

```
 r: u8,
```

```
 g: u8,
```

```
 b: u8,
```

```
}
```

```
impl Default for Color {
```

```
// default color is black
```

```
// 默认颜色是黑色
```

```
fn default() -> Self {
```

```
 Color {
```

```
 r: 0,
```

```
 g: 0,
```

```
 b: 0,
```

```
 }
```

```
}
```

```
}
```

...

> This is useful for quick prototyping but also in any instance where we just need an instance of a type and aren't picky about what it is:

这不仅利于快速原型设计，另外，在有时我们仅仅只是需要该类型的一个值，却完全不在意该值是什么的时候，这也非常方便。

```
```rust
fn main() {
    // just give me some color!
    let color = Color::default();
}

```

```

> This is also an option we may want to explicitly expose to the users of our functions:

如此，我们可以明确地向该函数的用户传达出该函数某个参数的可选择性：

```
```rust
struct Canvas;

enum Shape {
    Circle,
    Rectangle,
}

impl Canvas {
    // let user optionally pass a color
    // 用户可选地传入一个 color
    fn paint(&mut self, shape: Shape, color: Option<Color>) {
        // if no color is passed use the default color
        // 若用户没有传入 color，即使用默认的 color
        let color = color.unwrap_or_default();
        // etc
    }
}
```

```

> `Default` is also useful in generic contexts where we need to construct generic types:

在泛型编程的语境中，`Default` 特性也可显其威力。

```rust

```
fn guarantee_length<T: Default>(mut vec: Vec<T>, min_len: usize) -> Vec<T> {
    for _ in 0..min_len.saturating_sub(vec.len()) {
        vec.push(T::default());
    }
    vec
}
```

> Another way we can take advantage of `Default` types is for partial initialization of structs using Rust's struct update syntax. We may have a `new` constructor for `Color` which takes every member as an argument:

另外，我们在使用 update 记号构造结构体时也可享受到 `Default` 特性带来的便利。我们以 `Color` 结构的 `new` 构造器函数为例，它接受该结构的全部成员作为参数：

```rust

```
impl Color {
 fn new(r: u8, g: u8, b: u8) -> Self {
 Color {
 r,
 g,
 b,
 }
 }
}
```

> However we can also have convenience constructors that only accept a particular struct member each and fall back to the default values for the other struct members:

考虑以下更加便捷的构造器函数 —— 它仅接受该结构的部分成员作为参数，其它未指定的成员则回落到默认值：

```
```rust
```

```
impl Color {  
    fn red(r: u8) -> Self {  
        Color {  
            r,  
            ..Color::default()  
        }  
    }  
  
    fn green(g: u8) -> Self {  
        Color {  
            g,  
            ..Color::default()  
        }  
    }  
  
    fn blue(b: u8) -> Self {  
        Color {  
            b,  
            ..Color::default()  
        }  
    }  
}
```

> There's also a `Default` derive macro for so we can write `Color` like this:

`Default` 特性也可以用衍生宏的方式来实现：

```
```rust
// default color is still black
// because u8::default() == 0
// 默认颜色仍旧是黑色
// 因为 u8::default() == 0
#[derive(Default)]
struct Color {
 r: u8,
 g: u8,
 b: u8
}
```
``
```

```
#### Clone
```

预备知识

- [Self](#self)
- [Methods](#methods)
- [Default Impls](#default-impls)
- [Derive Macros](#derive-macros)

```
```rust
trait Clone {
 fn clone(&self) -> Self;
}

// provided default impls
// 提供默认实现
```

```
fn clone_from(&mut self, source: &Self);
}
...
```

> We can convert immutable references of `Clone` types into owned values, i.e. `&T` -> `T` . `Clone` makes no promises about the efficiency of this conversion so it can be slow and expensive. To quickly impl `Clone` on a type we can use the derive macro:

对于实现了 `Clone` 特性的类型，我们可以将一个不可变的引用转换为自有的类型，比如 `&T` -> `T` 。`Clone` 特性对于这种转换的效率不做出保证，所以这样的转换速度可能很慢，代价可能很昂贵。

```rust

```
#[derive(Clone)]  
  
struct SomeType {  
  
    cloneable_member1: CloneableType1,  
  
    cloneable_member2: CloneableType2,  
  
    // etc  
}
```

// macro generates impl below

// 宏展开后为

```
impl Clone for SomeType {  
  
    fn clone(&self) -> Self {  
  
        SomeType {  
  
            cloneable_member1: self.cloneable_member1.clone(),  
  
            cloneable_member2: self.cloneable_member2.clone(),  
  
            // etc  
        }  
    }  
}
```

> `Clone` can also be useful in constructing instances of a type within a generic context. Here's an example from the previous section except using `Clone` instead of `Default`:

`Clone` 特性也有利于在泛型编程的语境中构造类型。请看下例：

```
```rust
```

```
fn guarantee_length<T: Clone>(mut vec: Vec<T>, min_len: usize, fill_with: &T) -> Vec<T> {
 for _ in 0..min_len.saturating_sub(vec.len()) {
 vec.push(fill_with.clone());
 }
 vec
}
```

> People also commonly use cloning as an escape hatch to avoid dealing with the borrow checker. Managing structs with references can be challenging, but we can turn the references into owned values by cloning them.

克隆确是一个可以逃避借用检查器的好方法。倘若我们编写的代码无法通过借用检查，那么不妨通过克隆将这些引用转换为自有类型。

```
```rust
```

```
// oof, we gotta worry about lifetimes

// 糟糕！我们真的有自信处理好 lifetime 吗？

struct SomeStruct<'a> {

    data: &'a Vec<u8>,
}
```

```
// now we're on easy street

// 好耶！人生苦短，我用 Clone！

struct SomeStruct {

    data: Vec<u8>,
}
```

...

> If we're working on a program where performance is not the utmost concern then we don't need to sweat cloning data. Rust is a low-level language that exposes a lot of low-level details so it's easy to get caught up in premature optimizations instead of actually solving the problem at hand. For many programs the best order of priorities is usually to build for correctness first, elegance second, and performance third, and only focus on performance after the program has been profiled and the performance bottlenecks have been identified. This is good general advice to follow, and if it doesn't apply to your particular program then you would know.

如果性能因素微不足道，我们不必羞于使用克隆。Rust 是一门底层语言，人们可以自由地控制程序行为的方方面面，这就很容易令人陷入盲目追求优化的陷阱，而不是专注于着手解决问题。对此我给出的建议是：正确第一，优雅第二，性能第三。只有程序初具雏形后，性能瓶颈的问题才可能凸显，这时我们再解决性能问题也不迟。与其说这是一条编程建议，更不如说这是一条人生建议，万事万物大抵如此，如果你现在不信，总有一天你会的。

Copy

预备知识

- [Marker Traits](#marker-traits)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Derive Macros](#derive-macros)

```rust

```
trait Copy: Clone {}
```

...

> We copy `Copy` types, e.g. `T` -> `T`. `Copy` promises the copy operation will be a simple bitwise copy so it will be very fast and efficient. We cannot impl `Copy` ourselves, only the compiler can provide an impl, but we can tell it to do so by using the `Copy` derive macro, together with the `Clone` derive macro since `Copy` is a subtrait of `Clone`:

对于实现了 `Copy` 特性的类型，我们可以拷贝它，即 `T` -> `T`。`Copy` 特性确保了拷贝操作是按位的拷贝，所以它更快更高效。`Copy` 特性不可手动实现，必须由编译器提供其实现。注意：当使用衍生宏为类型实现 `Copy` 特性时，必须同时使用 `Clone` 衍生宏，因为 `Copy` 是 `Clone` 的子特性：

```rust

```
#[derive(Copy, Clone)]
```

```
struct SomeType;
```

```
...
```

> `Copy` refines `Clone`. A clone may be slow and expensive but a copy is guaranteed to be fast and cheap, so a copy is just a fast clone. If a type impls `Copy` that makes the `Clone` impl trivial:

`Copy` 改良了 `Clone`。克隆操作可能速度缓慢且代价昂贵，但是拷贝操作一定是高效低耗的，可以说拷贝就是一种物美价廉的克隆。`Copy` 特性的实现会令 `Clone` 特性的实现变得微不足道：

```
```rust
```

```
// this is what the derive macro generates
```

```
// 衍生宏展开如下
```

```
impl<T: Copy> Clone for T {
```

```
// the clone method becomes just a copy
```

```
// 克隆实际上变成了一种拷贝
```

```
fn clone(&self) -> Self {
```

```
*self
```

```
}
```

```
}
```

```
...
```

> Impling `Copy` for a type changes its behavior when it gets moved. By default all types have \_move semantics\_ but once a type impls `Copy` it gets \_copy semantics\_. To explain the difference between the two let's examine these simple scenarios:

实现了 `Copy` 特性的类型，其在移动时的行为会发生变化。默认情况下，所有的类型都具有 \_移动语义\_，但是一旦该类型实现了 `Copy` 特性，则会变为 \_拷贝语义\_。请考虑下例中语义的不同：

```
```rust
```

```
// a "move", src: !Copy
```

```
// 移动语义，src 没有实现 Copy 特性
```

```
let dest = src;
```

```
// a "copy", src: Copy  
// 拷贝语义，src 实现 Copy 特性  
  
let dest = src;  
...  
...
```

> In both cases, `dest = src` performs a simple bitwise copy of `src`'s contents and moves the result into `dest`, the only difference is that in the case of _"a move"_ the borrow checker invalidates the `src` variable and makes sure it's not used anywhere else later and in the case of _"a copy"_ `src` remains valid and usable.

事实上，这两种语义背后执行的操作是完全相同的，都是将 `src` 按位复制到 `dest`。其不同在于，在移动语义下，借用检查器从此吊销了 `src` 的可用性，而在拷贝语义下，`src` 保持可用。

> In a nutshell: Copies _are_ moves. Moves _are_ copies. The only difference is how they're treated by the borrow checker.

言而总之，拷贝就是移动，移动就是拷贝。它们在底层毫无二致，仅仅是借用检查器对待它们的方式不同。

> For a more concrete example of a move, imagine `src` was a `Vec<i32>` and its contents looked something like this:

对于移动行为来讲更具体的例子——你可以将 `src` 想象为一个 `Vec<i32>`，它的结构体大致如下：

```
```rust  
{ data: *mut [i32], length: usize, capacity: usize }
...
...
```

> When we write `dest = src` we end up with:

执行 `dest = src` 的结果如下：

```
```rust  
src = { data: *mut [i32], length: usize, capacity: usize }  
dest = { data: *mut [i32], length: usize, capacity: usize }  
...  
...
```

> At this point both `src` and `dest` have aliased mutable references to the same data, which is a big no-no, so the borrow checker invalidates the `src` variable so it can't be used again without throwing a compile error.

此时 `src` 和 `dest` 就都是同一数据的可变引用了，这可就糟tm的大糕了，所以借用检查器就吊销了 `src` 的可用性，一旦再次使用 `src` 就会引发编译错误。

> For a more concrete example of a copy, imagine `src` was an `Option<i32>` and its contents looked something like this:

对于拷贝行为来讲更具体的例子 —— 你可以将 `src` 想象为一个 `Option<i32>`，它的结构体大致如下：

```
```rust
{ is_valid: bool, data: i32 }
```
```

```

> Now when we write `dest = src` we end up with:

执行 `dest = src` 的结果如下：

```
```rust
src = { is_valid: bool, data: i32 }
dest = { is_valid: bool, data: i32 }
```
```

```

> These are both usable simultaneously! Hence `Option<i32>` is `Copy`.

此时两者同时可用！因为 `Option<i32>` 实现了 `Copy`。

> Although `Copy` could be an auto trait the Rust language designers decided it's simpler and safer for types to explicitly opt into copy semantics rather than silently inheriting copy semantics whenever the type is eligible, as the latter can cause surprising confusing behavior which often leads to bugs.

或许你已经注意到，令 `Copy` 特性成为可自动实现的特性在理论上是可行的。但是 Rust 语言的设计者认为，比之于在恰当时隐式地继承拷贝语义，显示地声明为拷贝语义更加的简单和安全。前者可能会导致 Rust 语言产生十分反人类的行为，也更容易出现 bug。

Any

预备知识

- [Self](#self)
- [Generic Blanket Impls](#generic-blanket-impls)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Trait Objects](#trait-objects)

```rust

```
trait Any: 'static {
 fn type_id(&self) -> Typeld;
}
````
```

> Rust's style of polymorphism is parametric, but if we're looking to use a more ad-hoc style of polymorphism similar to dynamically-typed languages then we can emulate that using the `Any` trait. We don't have to manually impl this trait for our types because that's already covered by this generic blanket impl:

Rust 的多态性风格本身是参数化的，但如果希望临时使用一种更贴近于动态语言的多态性风格，可以借用 `Any` 特性来模拟。我们不需要手动实现 `Any` 特性，因为该特性通常由一揽子泛型实现所实现。

```rust

```
impl<T: 'static + ?Sized> Any for T {
 fn type_id(&self) -> Typeld {
 Typeld::of::<T>()
 }
}
```

```

> The way we get a `T` out of a `dyn Any` is by using the `downcast_ref::<T>()` and `downcast_mut::<T>()` methods:

对于 `dyn Any` 的特性对象，我们可以使用 `downcast_ref::<T>()` 或 `downcast_mut::<T>()` 来尝试解析出 `T`。

```rust

```
use std::any::Any;
```

```
#[derive(Default)]
```

```
struct Point {
```

```
 x: i32,
```

```
 y: i32,
```

```
}
```

```
impl Point {
```

```
 fn inc(&mut self) {
```

```
 self.x += 1;
```

```
 self.y += 1;
```

```
}
```

```
}
```

```
fn map_any(mut any: Box<dyn Any>) -> Box<dyn Any> {
```

```
 if let Some(num) = any.downcast_mut::<i32>() {
```

```
 *num += 1;
```

```
 } else if let Some(string) = any.downcast_mut::<String>() {
```

```
 *string += "!";
```

```
 } else if let Some(point) = any.downcast_mut::<Point>() {
```

```
 point.inc();
```

```
}
```

```

any
}

fn main() {
 let mut vec: Vec<Box<dyn Any>> = vec![

 Box::new(0),

 Box::new(String::from("a")),

 Box::new(Point::default()),

];
 // vec = [0, "a", Point { x: 0, y: 0 }]

 vec = vec.into_iter().map(map_any).collect();
 // vec = [1, "a!", Point { x: 1, y: 1 }]
}
```

```

> This trait rarely `_needs_` to be used because on top of parametric polymorphism being superior to ad-hoc polymorphism in most scenarios the latter can also be emulated using enums which are more type-safe and require less indirection. For example, we could have written the above example like this:

这个特性鲜少被使用，因为参数化的多态性时常常要优于这样变通使用的多态性，且后者也可以使用更加类型安全和更加直接的枚举来模拟。如下例：

```

```rust
#[derive(Default)]
struct Point {
 x: i32,
 y: i32,
}

impl Point {
 fn inc(&mut self) {
 self.x += 1;
 }
}
```

```
self.y += 1;
}

}

enum Stuff {
 Integer(i32),
 String(String),
 Point(Point),
}

fn map_stuff(mut stuff: Stuff) -> Stuff {
 match &mut stuff {
 Stuff::Integer(num) => *num += 1,
 Stuff::String(string) => *string += "!",
 Stuff::Point(point) => point.inc(),
 }
 stuff
}

fn main() {
 let mut vec = vec![
 Stuff::Integer(0),
 Stuff::String(String::from("a")),
 Stuff::Point(Point::default()),
];
 // vec = [0, "a", Point { x: 0, y: 0 }]
 vec = vec.into_iter().map(map_stuff).collect();
 // vec = [1, "a!", Point { x: 1, y: 1 }]
}
```
    
```

> Despite `Any` rarely being _needed_ it can still be convenient to use sometimes, as we'll later see in the **Error Handling** section.

尽管 `Any` 特性鲜少是必须要被使用的，但有时它又是一种非常便捷的用法，我们将在 **错误处理** 一章中领会这一点。

文本格式化特性 Formatting Traits

> We can serialize types into strings using the formatting macros in `std::fmt`, the most well-known of the bunch being `println!`. We can pass formatting parameters to the `{} placeholders used within format `str`s which are then used to select which trait impl to use to serialize the placeholder's argument.

我们可以使用 `std::fmt` 中提供的文本格式化宏来序列化结构体，例如我们最熟悉的 `println!`。我们可以将文本格式化的参数传入 `{} 占位符`，以选择具体用哪个特性来序列化该结构。

| 特性 | 占位符 | 描述 |

|-----|-----|-----|

| `Display` | `{}` | 常规序列化 |

| `Debug` | `{:?}` | 调试序列化 |

| `Octal` | `{:o}` | 八进制序列化 |

| `LowerHex` | `{:x}` | 小写十六进制序列化 |

| `UpperHex` | `{:X}` | 大写十六进制序列化 |

| `Pointer` | `{:p}` | 内存地址 |

| `Binary` | `{:b}` | 二进制序列化 |

| `LowerExp` | `{:e}` | 小写指数序列化 |

| `UpperExp` | `{:E}` | 大写十六进制序列化 |

Display & ToString

预备知识

- [Self](#self)
- [Methods](#methods)
- [Generic Blanket Impls](#generic-blanket-impls)

```rust

```
trait Display {
 fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
...
```

> `Display` types can be serialized into `String`s which are friendly to the end users of the program. Example impl for `Point`:

实现 `Display` 特性的类型可以被序列化为 `String`。这对于程序的用户来说非常的友好。例如：

```rust

```
use std::fmt;  
  
#[derive(Default)]  
  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl fmt::Display for Point {  
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {  
        write!(f, "({}, {})", self.x, self.y)  
    }  
}
```

```
fn main() {  
    println!("origin: {}", Point::default());  
    // prints "origin: (0, 0)"  
  
    // get Point's Display representation as a String  
    // Point 表达为可显示的 String  
    let stringified_point = format!("{}", Point::default());  
    assert_eq!("(0, 0)", stringified_point); //  
}  
///
```

> Aside from using the `format!` macro to get a type's display representation as a `String` we can use the `ToString` trait:

除了使用 `format!` 宏来序列化结构体，我们也可以使用 `ToString` 特性：

```
```rust  
trait ToString {
 fn to_string(&self) -> String;
}
///
```

> There's no need for us to impl this ourselves. In fact we can't, because of this generic blanket impl that automatically implements `ToString` for any type which implements `Display`:

我们不需要自己手动实现，事实上，我们也不能，因为对于实现了 `Display` 的类型来说，`ToString` 是由一揽子泛型实现所自动实现的。

```
```rust  
impl<T: Display + ?Sized> ToString for T;  
///
```

> Using `ToString` with `Point`:

对 `Point` 使用 `ToString` 特性：

```rust

```
#![test] //

fn display_point() {

 let origin = Point::default();

 assert_eq!(format!("{}", origin), "(0, 0)");

}
```

```
#![test] //
```

```
fn point_to_string() {

 let origin = Point::default();

 assert_eq!(origin.to_string(), "(0, 0)");

}
```

```
#![test] //
```

```
fn display_equals_to_string() {

 let origin = Point::default();

 assert_eq!(format!("{}", origin), origin.to_string());

}
```

```

Debug

预备知识

- [Self](#self)

- [Methods](#methods)
- [Derive Macros](#derive-macros)
- [Display & ToString](#display--tostring)

```rust

```
trait Debug {
 fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```

```

> `Debug` has an identical signature to `Display`. The only difference is that the `Debug` impl is called when we use the `{:?}` formatting specifier. `Debug` can be derived:

`Debug` 与 `Display` 具有相同的签名。唯一的区别在于我们使用 `{:?}` 文本格式化指令来调用 `Debug` 特性。`Debug` 特性可以使用如下方法衍生：

```rust

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

```
#[derive(Debug)]
```

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

```
// derive macro generates impl below
```

// 衍生宏展开如下

```
impl fmt::Debug for Point {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 f.debug_struct("Point")
 .field("x", &self.x)
 .field("y", &self.y)
 }
}
```

```
.finish()
}
}
}
...
```

> Impling `Debug` for a type also allows it to be used within the `dbg!` macro which is superior to `println!` for quick and dirty print logging. Some of its advantages:

- > 1. `dbg!` prints to stderr instead of stdout so the debug logs are easy to separate from the actual stdout output of our program.
- > 2. `dbg!` prints the expression passed to it as well as the value the expression evaluated to.
- > 3. `dbg!` takes ownership of its arguments and returns them so you can use it within expressions:

为特定类型实现 `Debug` 特性的同时，这也使得我们可以使用 `dbg!` 宏来快速地调试程序，这种方式要优于 `println!`。其优点在于：

1. `dbg!` 输出到标准错误流而不是标准输出流，所以我们能够很容易地将调试信息提取出来。
2. `dbg!` 同时输出值和值的求值表达式。
3. `dbg!` 接管参数的属权，但不会吞掉参数，而是再抛出来，所以可以将它用在表达式中：

```rust

```
fn some_condition() -> bool {  
    true  
}
```

```
// no logging  
// 没有日志  
fn example() {  
    if some_condition() {  
        // some code  
    }  
}
```

```
// println! logging
```

```
// 使用 println! 打印日志
fn example_printh() {
    //
    let result = some_condition();
    println!("{}", result); // just prints "true"
    // 仅仅打印 "true"
    if result {
        // some code
    }
}

// dbg! logging

// 使用 dbg! 打印日志
fn example_dbg() {
    //
    if dbg!(some_condition()) { // prints "[src/main.rs:22] some_condition() = true"
        // 太棒了！打印出丰富的调试信息
        // some code
    }
}
```

```

> The only downside is that `dbg!` isn't automatically stripped in release builds so we have to manually remove it from our code if we don't want to ship it in the final executable.

`dbg!` 宏唯一的缺点是，它不能在构建最终发布的二进制文件时自动删除，我们不得不手动删除相关代码。

## 算符重载特性 Operator Traits

> All operators in Rust are associated with traits. If we'd like to impl operators for our types we have to impl the associated traits.

在 Rust 中，所有的算符都与相应的特性相关联。为特定类型实现相应特性，即为该类型实现了相应算符。

| 特性 | 类别 | 算符 | 描述 |

| -----                                                    | ----- | ----- | ----- |
|----------------------------------------------------------|-------|-------|-------|
| `Eq` , `PartialEq`   比较   `==`   相等                      |       |       |       |
| `Ord` , `PartialOrd`   比较   `<` , `>` , `<=` , `>=`   比较 |       |       |       |
| `Add`   算数   `+`   加                                     |       |       |       |
| `AddAssign`   算数   `+=`   加等于                            |       |       |       |
| `BitAnd`   算数   `&`   按位与                                |       |       |       |
| `BitAndAssign`   算数   `&=`   按位与等于                       |       |       |       |
| `BitXor`   算数   `^`   按位异或                               |       |       |       |
| `BitXorAssign`   算数   `^=`   按位异或等于                      |       |       |       |
| `Div`   算数   `/`   除                                     |       |       |       |
| `DivAssign`   算数   `/=`   除等于                            |       |       |       |
| `Mul`   算数   `*`   乘                                     |       |       |       |
| `MulAssign`   算数   `*=`   乘等于                            |       |       |       |
| `Neg`   算数   `-`   一元负                                   |       |       |       |
| `Not`   算数   `!`   一元逻辑非                                 |       |       |       |
| `Rem`   算数   `%`   求余                                    |       |       |       |
| `RemAssign`   算数   `%=`   求余等于                           |       |       |       |
| `Shl`   算数   `<<`   左移                                   |       |       |       |
| `ShlAssign`   算数   `<<=`   左移等于                          |       |       |       |
| `Shr`   算数   `>>`   右移                                   |       |       |       |
| `ShrAssign`   算数   `>>=`   右移等于                          |       |       |       |
| `Sub`   算数   `-`   减                                     |       |       |       |
| `SubAssign`   算数   `-=`   减等于                            |       |       |       |
| `Fn`   闭包   `(...args)`   不可变闭包调用                        |       |       |       |
| `FnMut`   闭包   `(...args)`   可变闭包调用                      |       |       |       |

|                                       |
|---------------------------------------|
| `FnOnce`   闭包   `(...args)`   一次性闭包调用 |
| `Deref`   其它   `*`   不可变解引用           |
| `DerefMut`   其它   `*`   可变解引用         |
| `Drop`   其它   -   类型析构                |
| `Index`   其它   `[]`   不可变索引           |
| `IndexMut`   其它   `[]`   可变索引         |
| `RangeBounds`   其它   `..`   范围迭代      |

### ### 比较特性 Comparison Traits

| 特性                                                   | 类别 | 算符 | 描述 |
|------------------------------------------------------|----|----|----|
| `Eq`, `PartialEq`   比较   `==`   相等                   |    |    |    |
| `Ord`, `PartialOrd`   比较   `<`, `>`, `<=`, `>=`   比较 |    |    |    |

### #### PartialEq & Eq

#### 预备知识

- [Self](#self)
- [Methods](#methods)
- [Generic Parameters](#generic-parameters)
- [Default Impl](#default-impls)
- [Generic Blanket Impl](#generic-blanket-impls)
- [Marker Traits](#marker-traits)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Sized](#sized)

```
```rust
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;

    // provided default impls
    // 提供默认实现
    fn ne(&self, other: &Rhs) -> bool;
}

```

```

> `PartialEq<Rhs>` types can be checked for equality to `Rhs` types using the `==` operator.

实现了 `PartialEq<Rhs>` 特性的类型可以使用 `==` 算符来检查与 `Rhs` 的相等性。

> All `PartialEq<Rhs>` impls must ensure that equality is symmetric and transitive. That means > for all `a`, `b`, and `c`:

> – `a == b` implies `b == a` (symmetry)  
> – `a == b && b == c` implies `a == c` (transitivity)

对 `PartialEq<Rhs>` 的实现须确保实现对称性与传递性。这意味着对于任意 `a` , `b` 和 `c` 有：

- 若 `a == b` 则 `b == a` (对称性)
- 若 `a == b && b == c` 则 `a == c` (传递性)

> By default `Rhs = Self` because we almost always want to compare instances of a type to each other, and not to instances of different types. This also automatically guarantees our impl is symmetric and transitive.

>

默认情况下 `Rhs = Self` 是因为我们几乎总是在相同类型之间进行比较。这也自动地确保了我们的实现是对称的、可传递的。

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

// Rhs == Self == Point
impl PartialEq for Point {
    // impl automatically symmetric & transitive
    // 该实现自动确保了对称性和传递性
    fn eq(&self, other: &Point) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```
```

```

> If all the members of a type impl `PartialEq` then it can be derived:

如果特定类型的成员都实现了 `PartialEq` 特性，那么该类型也可衍生该特性：

```
```rust
#[derive(PartialEq)]
struct Point {
 x: i32,
 y: i32
}

#[derive(PartialEq)]
enum Suit {
 Spade,
```

```
Heart,
```

```
Club,
```

```
Diamond,
```

```
}
```

```
...
```

> Once we impl `PartialEq` for our type we also get equality comparisons between references of our type for free thanks to these generic blanket impls:

多亏了一揽子泛型实现，一旦我们为特定类型实现了 `PartialEq` 特性，那么直接使用该类型的引用互相比较也是可以的：

```
```rust
```

```
// this impl only gives us: Point == Point  
  
// 该衍生宏本身只允许我们在结构体之间进行比较  
#[derive(PartialEq)]  
  
struct Point {  
  
    x: i32,  
  
    y: i32  
}
```

```
// all of the generic blanket impls below  
  
// are provided by the standard library  
  
// 以下的一揽子泛型实现由标准库提供
```

```
// this impl gives us: &Point == &Point  
  
// 这个一揽子泛型实现允许我们通过不可变引用之间进行比较  
  
impl<A, B> PartialEq<&'_ B> for &'_ A  
  
where A: PartialEq<B> + ?Sized, B: ?Sized;  
  
// this impl gives us: &mut Point == &Point
```

```
// 这个一揽子泛型实现允许我们通过可变引用与不可变引用进行比较
impl<A, B> PartialEq<&'_ B> for &'_ mut A
```

```
where A: PartialEq<B> + ?Sized, B: ?Sized;
```

```
// this impl gives us: &Point == &mut Point
```

```
// 这个一揽子泛型实现允许我们通过不可变引用与可变引用进行比较
```

```
impl<A, B> PartialEq<&'_ mut B> for &'_ A
```

```
where A: PartialEq<B> + ?Sized, B: ?Sized;
```

```
// this impl gives us: &mut Point == &mut Point
```

```
// 这个一揽子泛型实现允许我们通过可变引用之间进行比较
```

```
impl<A, B> PartialEq<&'_ mut B> for &'_ mut A
```

```
where A: PartialEq<B> + ?Sized, B: ?Sized;
```

```
```
```

> Since this trait is generic we can define equality between different types. The standard library leverages this to allow checking equality between the many string-like types such as `String`, `&str`, `PathBuf`, `&Path`, `OsString`, `&OsStr`, and so on.

由于该特性提供泛型，我们可以定义不同类型之间的可相等性。标准库正是利用这一点提供了不同类型字符串之间的比较功能，例如`String`、`&str`、`PathBuf`、`&Path`、`OsString` 和 `&OsStr`等等。

> Generally, we should only impl equality between different types if they contain the same kind of data and the only difference between the types is how they represent the data or how they allow interacting with the data.

通常来说我们仅会实现相同类型之间的可相等性，除非两种类型虽然包含同一类数据，但又有表达形式或交互形式的差异，这时我们才会考虑实现不同类型之间的可相等性。

> Here's a cute but bad example of how someone might be tempted to impl `PartialEq` to check equality between different types that don't meet the above criteria:

以下是一个有趣但糟糕的例子，它尝试为不同类型实现 `PartialEq` 但又违背了上述要求：

```
```rust
```

```
#[derive(PartialEq)]
```

```
enum Suit {
```

```
    Spade,
```

```
    Club,
```

```
    Heart,
```

```
    Diamond,
```

```
}
```

```
#[derive(PartialEq)]
```

```
enum Rank {
```

```
    Ace,
```

```
    Two,
```

```
    Three,
```

```
    Four,
```

```
    Five,
```

```
    Six,
```

```
    Seven,
```

```
    Eight,
```

```
    Nine,
```

```
    Ten,
```

```
    Jack,
```

```
    Queen,
```

```
    King,
```

```
}
```

```
#[derive(PartialEq)]
```

```
struct Card {
```

```
    suit: Suit,
```

```
    rank: Rank,
```

```
}
```

```

// check equality of Card's suit
// 检查花色的相等性

impl PartialEq<Suit> for Card {
fn eq(&self, other: &Suit) -> bool {
    self.suit == *other
}
}

// check equality of Card's rank
// 检查牌序的相等性

impl PartialEq<Rank> for Card {
fn eq(&self, other: &Rank) -> bool {
    self.rank == *other
}
}

fn main() {
let AceOfSpades = Card {
    suit: Suit::Spade,
    rank: Rank::Ace,
};

assert!(AceOfSpades == Suit::Spade); //
assert!(AceOfSpades == Rank::Ace); //
}

```

```

> It works and kinda makes sense. A card which is an Ace of Spades is both an Ace and a Spade, and if we're writing a library to handle playing cards it's reasonable that we'd want to make it easy and convenient to individually check the suit and rank of a card. However, something's missing: symmetry! We can `Card == Suit` and `Card == Rank` but we cannot `Suit == Card` or `Rank == Card` so let's fix that:

上述代码有效且其逻辑有几分道理，黑桃 A 既是黑桃也是 A。但如果我们真的去写一个处理扑克牌的库的话，最简单也最方便的方法莫过于独立地检查牌面的花色和牌序。而且，上述代码并不满足对称性！我们可以使用 `Card == Suit` 和 `Card == Rank`，但却不能使用 `Suit == Card` 和 `Rank == Card`，让我们来修复这一点：

```rust

// check equality of Card's suit

// 检查花色的相等性

impl PartialEq<Suit> for Card {

fn eq(&self, other: &Suit) -> bool {

self.suit == *other

}

}

// added for symmetry

// 增加对称性

impl PartialEq<Card> for Suit {

fn eq(&self, other: &Card) -> bool {

*self == other.suit

}

}

// check equality of Card's rank

// 检查牌序的相等性

impl PartialEq<Rank> for Card {

fn eq(&self, other: &Rank) -> bool {

self.rank == *other

}

}

// added for symmetry

// 增加对称性

```
impl PartialEq<Card> for Rank {
fn eq(&self, other: &Card) -> bool {
*self == other.rank
}
}
```
```

```

> We have symmetry! Great. Adding symmetry just broke transitivity! Oops. This is now possible:

我们实现了对称性！棒！但是实现对称性却破坏了传递性！糟tm大糕！考虑以下代码：

```
```rust
fn main() {
// Ace of Spades
// A
let a = Card {
suit: Suit::Spade,
rank: Rank::Ace,
};

let b = Suit::Spade;
// King of Spades
// K
let c = Card {
suit: Suit::Spade,
rank: Rank::King,
};

assert!(a == b && b == c); //
assert!(a == c); //
}
```
```

```

> A good example of impling `PartialEq` to check equality between different types would be a program that works with distances and uses different types to represent different units of measurement.

关于对不同类型实现 `PartialEq` 特性的绝佳示例如下，本程序的功能在于处理空间上的距离，它使用不同的类型以表示不同的测量单位：

```
```rust
```

```
#[derive(PartialEq)]
```

```
struct Foot(u32);
```

```
#[derive(PartialEq)]
```

```
struct Yard(u32);
```

```
#[derive(PartialEq)]
```

```
struct Mile(u32);
```

```
impl PartialEq<Mile> for Foot {
```

```
fn eq(&self, other: &Mile) -> bool {
```

```
    self.0 == other.0 * 5280
```

```
}
```

```
}
```

```
impl PartialEq<Foot> for Mile {
```

```
fn eq(&self, other: &Foot) -> bool {
```

```
    self.0 * 5280 == other.0
```

```
}
```

```
}
```

```
impl PartialEq<Mile> for Yard {
```

```
fn eq(&self, other: &Mile) -> bool {
```

```
    self.0 == other.0 * 1760
```

```
}

}

impl PartialEq<Yard> for Mile {

fn eq(&self, other: &Yard) -> bool {
    self.0 * 1760 == other.0
}
}
```

```
impl PartialEq<Foot> for Yard {

fn eq(&self, other: &Foot) -> bool {
    self.0 * 3 == other.0
}
}
```

```
impl PartialEq<Yard> for Foot {

fn eq(&self, other: &Yard) -> bool {
    self.0 == other.0 * 3
}
}
```

```
fn main() {

let a = Foot(5280);

let b = Yard(1760);

let c = Mile(1);
```

```
// symmetry

// 对称性

assert!(a == b && b == a); //

assert!(b == c && c == b); //
```

```
assert!(a == c && c == a); //  
  
// transitivity  
  
// 传递性  
  
assert!(a == b && b == c && a == c); //  
  
assert!(c == b && b == a && c == a); //  
  
}  
  
...  
  
}
```

> `Eq` is a marker trait and a subtrait of `PartialEq<Self>`.

`Eq` 是仅用于标记的特性，也是 `PartialEq<Self>` 的子特性。

```rust

```
trait Eq: PartialEq<Self> {}

...

}
```

> If we impl `Eq` for a type, on top of the symmetry & transitivity properties required by `PartialEq`, we're also guaranteeing reflexivity, i.e. `a == a` for all `a`. In this sense `Eq` refines `PartialEq` because it represents a stricter version of equality. If all members of a type impl `Eq` then the `Eq` impl can be derived for the type.

鉴于 `PartialEq` 特性提供的对称性与传递性，一旦我们实现 `Eq` 特性，我们也就确保了该类型具有自反性，即对任意 `a` 有 `a == a`。可以说，`Eq` 改良了 `PartialEq`，因为它实现了一个比后者更加严格的可相等性。如果一个类型的全部成员都实现了 `Eq` 特性，那么该类型本身也可以衍生出该特性。

> Floats are `PartialEq` but not `Eq` because `NaN != NaN`. Almost all other `PartialEq` types are trivially `Eq`, unless of course if they contain floats.

所有的浮点类型都实现了 `PartialEq` 但是没有实现 `Eq`，因为 `NaN != NaN`。几乎所有其它实现 `PartialEq` 的类型也都自然地实现了 `Eq`，除非它们包含了浮点数。

> Once a type implements `PartialEq` and `Debug` we can use it in the `assert\_eq!` macro. We can also compare collections of `PartialEq` types.

对于实现了 `PartialEq` 和 `Debug` 的类型，我们也可以将它用于 `assert\_eq!` 宏。并且，我们可以对实现 `PartialEq` 特性的类型组成的集合进行比较。

```rust

```
#[derive(PartialEq, Debug)]  
  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn example_assert(p1: Point, p2: Point) {  
    assert_eq!(p1, p2);  
}  
  
fn example_compare_collections<T: PartialEq>(vec1: Vec<T>, vec2: Vec<T>) {  
    // if T: PartialEq this now works!  
  
    if vec1 == vec2 {  
        // some code  
    } else {  
        // other code  
    }  
}  
...  
  
#### Hash
```

预备知识

- [Self](#self)
- [Methods](#methods)

- [Generic Parameters](#generic-parameters)
- [Default Impls](#default-impls)
- [Derive Macros](#derive-macros)
- [PartialEq & Eq](#partialeq--eq)

```rust

```
trait Hash {
 fn hash<H: Hasher>(&self, state: &mut H);

 // provided default impls
 // 提供默认实现
 fn hash_slice<H: Hasher>(data: &[Self], state: &mut H);
}

```

```

> This trait is not associated with any operator, but the best time to talk about it is right after `PartialEq` & `Eq` so here it is. `Hash` types can be hashed using a `Hasher`.

本特性并未关联到任何算符，之所以在这里提及，是因为它与 `PartialEq` 与 `Eq` 密切的关系。实现 `Hash` 特性的类型可以通过 `Hasher` 作哈希运算。

```rust

```
use std::hash::Hasher;
```

```
use std::hash::Hash;
```

```
struct Point {
```

```
 x: i32,
```

```
 y: i32,
```

```
}
```

```
impl Hash for Point {
```

```
 fn hash<H: Hasher>(&self, hasher: &mut H) {
```

```
 hasher.write_i32(self.x);

 hasher.write_i32(self.y);

}

}

```

```

> There's a derive macro which generates the same impl as above:

以下衍生宏展开与以上代码中相同的实现：

```
```rust
```

```
#[derive(Hash)]

struct Point {

 x: i32,

 y: i32,
}
```

> If a type implements both `Hash` and `Eq` those implementations must agree with each other such that for all `a` and `b` if `a == b` then `a.hash() == b.hash()`. So we should always use the derive macro to implement both or manually implement both, but not mix the two, otherwise we risk breaking the above invariant.

如果一个类型同时实现了 `Hash` 和 `Eq`，那么二者必须要实现步调一致，即对任意 `a` 与 `b`，若有 `a == b`，则必有 `a.hash() == b.hash()`。所以，对于同时实现二者，要么都用衍生宏，要么都手动实现，不要一个用衍生宏，而另一个手动实现，否则我们将冒着步调不一致的极大风险。

> The main benefit of impling `Eq` and `Hash` for a type is that it allows us to store that type as keys in `HashMap`s and `HashSet`s.

实现 `Eq` 和 `Hash` 特性的主要好处在于，这允许我们将该类型作为一个键存储于 `HashMap` 和 `HashSet` 中。

```
```rust
```

```
use std::collections::HashSet;
```

```

// now our type can be stored
// in HashSets and HashMaps!
// 现在我们的类型可以存储于 HashSet 和 HashMap 中了!
#[derive(PartialEq, Eq, Hash)]
struct Point {
    x: i32,
    y: i32,
}

fn example_hashset() {
    let mut points = HashSet::new();
    points.insert(Point { x: 0, y: 0 });
}

```
PartialOrd & Ord

```

## 预备知识

- [Self](#self)
- [Methods](#methods)
- [Generic Parameters](#generic-parameters)
- [Default Impls](#default-impls)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Derive Macros](#derive-macros)
- [Sized](#sized)
- [PartialEq & Eq](#partialeq--eq)

```
```rust
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    // provided default impls
    // 提供默认实现
    fn lt(&self, other: &Rhs) -> bool;
    fn le(&self, other: &Rhs) -> bool;
    fn gt(&self, other: &Rhs) -> bool;
    fn ge(&self, other: &Rhs) -> bool;
}
```

> `PartialOrd<Rhs>` types can be compared to `Rhs` types using the `<`, `<=`, `>`, and `>=` operators.

实现 `PartialOrd<Rhs>` 的类型可以和 `Rhs` 的类型之间使用 `<`, `<=`, `>`, 和 `>=` 算符。

```
> All `PartialOrd` impls must ensure that comparisons are asymmetric and transitive. That means for all `a`, `b`, and `c`:
> - `a < b` implies `!(a > b)` (asymmetry)
> - `a < b && b < c` implies `a < c` (transitivity)
>
```

实现 `PartialOrd` 时须确保比较的非对称性和传递性。这意味着对任意 `a`，`b`，`c` 有：

- 若 `a < b` 则 `!(a > b)` (非对称性)
- 若 `a < b && b < c` 则 `a < c` (传递性)

> `PartialOrd` is a subtrait of `PartialEq` and their impls must always agree with each other.

`PartialOrd` 是 `PartialEq` 的子特性，二者必须要实现步调一致。

```rust

```
fn must_always_agree<T: PartialOrd + PartialEq>(t1: T, t2: T) {
 assert_eq!(t1.partial_cmp(&t2) == Some(Ordering::Equal), t1 == t2);
}
````
```

> `PartialOrd` refines `PartialEq` in the sense that when comparing `PartialEq` types we can check if they are equal or not equal, but when comparing `PartialOrd` types we can check if they are equal or not equal, and if they are not equal we can check if they are unequal because the first item is less than or greater than the second item.

`PartialOrd` 改良了 `PartialEq`，后者仅能比较是否相等，而前者除了能比较是否相等，还能比较孰大孰小。

> By default `Rhs = Self` because we almost always want to compare instances of a type to each other, and not to instances of different types. This also automatically guarantees our impl is symmetric and transitive.

默认情况下 `Rhs = Self`，因为我们几乎总是在相同类型的实例之间相比较，而不是不同类型之间。这一点自动保证了我们的实现的对称性和传递性。

```rust

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

#[derive(PartialEq, PartialOrd)]
```

```

struct Point {
 x: i32,
 y: i32
}

// Rhs == Self == Point

impl PartialOrd for Point {
 // impl automatically symmetric & transitive
 // 该实现自动确保了对称性与传递性

 fn partial_cmp(&self, other: &Point) -> Option<Ordering> {
 Some(match self.x.cmp(&other.x) {
 Ordering::Equal => self.y.cmp(&other.y),
 ordering => ordering,
 })
 }
}
}

```

```

> If all the members of a type impl `PartialOrd` then it can be derived:

如果特定类型的全部成员都实现了 `PartialOrd` 特性，那么该类型也可以衍生出该特性：

```

```rust
#[derive(PartialEq, PartialOrd)]
struct Point {
 x: i32,
 y: i32,
}

#[derive(PartialEq, PartialOrd)]

```

```
enum Stoplight {
 Red,
 Yellow,
 Green,
}
```

> The `PartialOrd` derive macro orders types based on the lexicographical order of their members:

`PartialOrd` 衍生宏依据 \*\*类型成员的定义顺序\*\* 对类型进行排序：

```rust

```
// generates PartialOrd impl which orders

// Points based on x member first and

// y member second because that's the order

// they appear in the source code

// 宏展开的 PartialOrd 实现排序时

// 首先考虑 x 再考虑 y

// 因为这是它们在源代码中出现的顺序

#[derive(PartialOrd, PartialEq)]

struct Point {

    x: i32,

    y: i32,
}
```

```
// generates DIFFERENT PartialOrd impl

// which orders Points based on y member

// first and x member second

// 这里宏展开的 PartialOrd 实现排序时

// 首先考虑 y 再考虑 x
```

```
##[derive(PartialOrd, PartialEq)]  
struct Point {  
    y: i32,  
    x: i32,  
}  
...  
}
```

> `Ord` is a subtrait of `Eq` and `PartialOrd<Self>`:

`Ord` 是 `Eq` 和 `PartialOrd<Self>` 的子特性:

```rust

```
trait Ord: Eq + PartialOrd<Self> {
 fn cmp(&self, other: &Self) -> Ordering;

 // provided default impls
 // 提供默认实现
 fn max(self, other: Self) -> Self;
 fn min(self, other: Self) -> Self;
 fn clamp(self, min: Self, max: Self) -> Self;
}
...
}
```

> If we impl `Ord` for a type, on top of the asymmetry & transitivity properties required by `PartialOrd`, we're also guaranteeing that the asymmetry is total, i.e. exactly one of `a < b`, `a == b` or `a > b` is true for any given `a` and `b`. In this sense `Ord` refines `Eq` and `PartialOrd` because it represents a stricter version of comparisons. If a type impls `Ord` we can use that impl to trivially impl `PartialOrd`, `PartialEq`, and `Eq`:

鉴于 `PartialOrd` 提供的非对称性和传递性，对特定类型实现 `Ord` 特性的同时也就保证了其非对称性，即对于任意 `a` 与 `b` 有 `a < b` , `a == b` , `a > b`。可以说，`Ord` 改良了 `Eq` 和 `PartialOrd`，因为它提供了一种更加严格的比较。如果一个类型实现了 `Ord`，那么 `PartialOrd`，`PartialEq` 和 `Eq` 的实现也就微不足道了。

```rust

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

// of course we can use the derive macros here
// 可以使用衍生宏
#[derive(Ord, PartialOrd, Eq, PartialEq)]

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

// note: as with PartialOrd, the Ord derive macro
// orders a type based on the lexicographical order
// of its members
// 注意：与 PartialOrd 相同，Ord 衍生宏衍生宏依据
// 类型的成员的定义顺序 对类型进行排序

// but here's the impls if we wrote them out by hand
// 以下是我们手动的实现

impl Ord for Point {
    fn cmp(&self, other: &Self) -> Ordering {
        match self.x.cmp(&other.x) {
            Ordering::Equal => self.y.cmp(&other.y),
            ordering => ordering,
        }
    }
}

impl PartialOrd for Point {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
```

```

}

impl PartialEq for Point {
    fn eq(&self, other: &Self) -> bool {
        self.cmp(other) == Ordering::Equal
    }
}

impl Eq for Point {}

...

```

> Floats impl `PartialOrd` but not `Ord` because both `NaN < 0 == false` and `NaN >= 0 == false` are simultaneously true. Almost all other `PartialOrd` types are trivially `Ord`, unless of course if they contain floats.

浮点数类型实现了 `PartialOrd` 但是没有实现 `Ord`，因为 `NaN < 0 == false` 与 `NaN >= 0 == false` 同时为真。几乎所有其它实现 `PartialOrd` 的类型都实现了 `Ord`，除非该类型包含浮点数。

> Once a type implements `Ord` we can store it in `BTreeMap`s and `BTreeSet`s as well as easily sort it using the `sort()` method on slices and any types which deref to slices such as arrays, `Vec`s, and `VecDeque`s.

对于实现了 `Ord` 特性的类型，我们可以将它存储于 `BTreeMap` 和 `BTreeSet`，并且可以通过 `sort()` 方法对切片，或者任何可以自动解引用为切片的类型进行排序，例如 `Vec` 和 `VecDeque`。

```

```rust
use std::collections::BTreeSet;

// now our type can be stored
// in BTreeSets and BTreeMaps!
// 现在我们的类型可以存储于 BTreeSet 和 BTreeMap 中了!

#[derive(Ord, PartialOrd, PartialEq, Eq)]
struct Point {
 x: i32,
 y: i32,
}

```

```

fn example_btreeset() {
 let mut points = BTreeSet::new();
 points.insert(Point { x: 0, y: 0 });
}

// we can also .sort() Ord types in collections!
// 对于实现了 Ord 特性的类型，我们可以使用 .sort() 方法来对集合进行排序！

fn example_sort<T: Ord>(mut sortable: Vec<T>) -> Vec<T> {
 sortable.sort();
 sortable
}
...

```

### ### 算术特性 Arithmetic Traits

特性	类别	算符	描述
`Add`	算数	`+`	加
`AddAssign`	算数	`+=`	加等于
`BitAnd`	算数	`&`	按位与
`BitAndAssign`	算数	`&=`	按位与等于
`BitXor`	算数	`^`	按位异或
`BitXorAssign`	算数	`^=`	按位异或等于
`Div`	算数	`/`	除
`DivAssign`	算数	`/=`	除等于
`Mul`	算数	`*`	乘
`MulAssign`	算数	`*=`	乘等于
`Neg`	算数	`-`	一元负

```
| `Not` | 算数 | `!` | 一元逻辑非 |
| `Rem` | 算数 | `%` | 求余 |
| `RemAssign` | 算数 | `%=` | 求余等于 |
| `Shl` | 算数 | `<<` | 左移 |
| `ShlAssign` | 算数 | `<<=` | 左移等于 |
| `Shr` | 算数 | `>>` | 右移 |
| `ShrAssign` | 算数 | `>>=` | 右移等于 |
| `Sub` | 算数 | `-` | 减 |
| `SubAssign` | 算数 | `-=` | 减等于 |
```

> Going over all of these would be very redundant. Most of these only apply to number types anyway. We'll only go over `Add` and `AddAssign` since the `+` operator is commonly overloaded to do other stuff like adding items to collections or concatenating things together, that way we cover the most interesting ground and don't repeat ourselves.

>

详解以上所有算术特性未免显得多余，且其大多仅用于操作数字类型。本文仅就最常见被重载的 `Add` 和 `AddAssign` 特性，亦即 `+` 和 `+=` 算符，进行说明，其重载广泛用于为集合增加内容或对不同事物的连接。这样，我们多侧重于最有趣的地方，而不是无趣枯燥地重复。

#### #### Add & AddAssign

##### 预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Generic Parameters](#generic-parameters)
- [Generic Types vs Associated Types](#generic-types-vs-associated-types)
- [Derive Macros](#derive-macros)

```rust

```
trait Add<Rhs = Self> {
```

```
type Output;  
fn add(self, rhs: Rhs) -> Self::Output;  
}  
...  
...
```

> `Add<Rhs, Output = T>` types can be added to `Rhs` types and will produce `T` as output.

实现 `Add<Rhs, Output = T>` 特性的类型，与 `Rhs` 类型相加得到 `T` 类型的值。

> Example `Add<Point, Output = Point>` impl for `Point`:

下例对 `Point` 类型实现了 `Add<Rhs, Output = T>` :

```rust

```
#[derive(Clone, Copy)]
```

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

```
impl Add for Point {
 type Output = Point;
 fn add(self, rhs: Point) -> Point {
 Point {
 x: self.x + rhs.x,
 y: self.y + rhs.y,
 }
 }
}
```

```
fn main() {
 let p1 = Point { x: 1, y: 2 };

 let p2 = Point { x: 3, y: 4 };

 let p3 = p1 + p2;

 assert_eq!(p3.x, p1.x + p2.x); //
 assert_eq!(p3.y, p1.y + p2.y); //
}

```

```

> But what if we only had references to `Point`s? Can we still add them then? Let's try:

如果我们将对 `Point` 的引用进行如上操作还能将他们加在一起吗？我们试试：

```
```rust
```

```
fn main() {
 let p1 = Point { x: 1, y: 2 };

 let p2 = Point { x: 3, y: 4 };

 let p3 = &p1 + &p2; //

}
```

> Unfortunately not. The compiler throws:

遗憾的是，并不可以。编译器出错了：

```
```none
```

```
error[E0369]: cannot add `&Point` to `&Point`

--> src/main.rs:50:25
|
50 | let p3: Point = &p1 + &p2;
```

```
| --- ^ --- &Point
|| 
| &Point
|
= note: an implementation of `std::ops::Add` might be missing for `&Point`
```

```

> Within Rust's type system, for some type `T`, the types `T`, `&T`, and `&mut T` are all treated as unique distinct types which means we have to provide trait impls for each of them separately. Let's define an `Add` impl for `&Point`:

>

在 Rust 的类型系统中，对于特定类型 `T` 来讲，`T`，`&T`，`&mut T` 三者本身是具有不同类型的，这意味着我们需要对它们分别实现相应特性。下面我们将对 `&Point` 实现 `Add` 特性：

```rust

```
impl Add for &Point {
    type Output = Point;
    fn add(self, rhs: &Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}
```

```
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = &p1 + &p2; // assert_eq!(p3.x, p1.x + p2.x); //
    assert_eq!(p3.y, p1.y + p2.y); //
}
```

```
}
```

```
...
```

> However, something still doesn't feel quite right. We have two separate impls of `Add` for `Point` and `&Point` and they happen_ to do the same thing currently but there's no guarantee that they will in the future! For example, let's say we decide that when we add two `Point`s together we want to create a `Line` containing those two `Point`s instead of creating a new `Point`, we'd update our `Add` impl like this:

这是可行的，但是不觉得哪里怪怪的吗？我们对 `Point` 和 `&Point` 分别实现了 `Add` 特性，现在来看这两种实现能够保持步调一致，但是未来也能保证吗？例如，我们现在决定对两个 `Point` 相加要产生一个 `Line` 而不是 `Point`，可以对 `Add` 特性的实现做出如下改动：

```
```rust
```

```
use std::ops::Add;
```

```
#[derive(Copy, Clone)]
```

```
struct Point {
```

```
 x: i32,
```

```
 y: i32,
```

```
}
```

```
#[derive(Copy, Clone)]
```

```
struct Line {
```

```
 start: Point,
```

```
 end: Point,
```

```
}
```

```
// we updated this impl
```

```
// 我们更新了这个实现
```

```
impl Add for Point {
```

```
 type Output = Line;
```

```
 fn add(self, rhs: Point) -> Line {
```

```
 Line {
```

```
start: self,
end: rhs,
}

}

}

}

// but forgot to update this impl, uh oh!
// 但是忘记了更新这个实现， 糟tm大糕！

impl Add for &Point {
type Output = Point;
fn add(self, rhs: &Point) -> Point {
Point {
x: self.x + rhs.x,
y: self.y + rhs.y,
}
}
}

fn main() {
let p1 = Point { x: 1, y: 2 };
let p2 = Point { x: 3, y: 4 };
let line: Line = p1 + p2; //

let p1 = Point { x: 1, y: 2 };
let p2 = Point { x: 3, y: 4 };
let line: Line = &p1 + &p2; // expected Line, found Point
// 期待得到 Line , 但是得到 Point
}

```
```
```

> Our current impl of `Add` for `&Point` creates an unnecessary maintenance burden, we want the impl to match `Point`'s impl without having to manually update it every time we change `Point`'s impl. We'd like to keep our code as DRY (Don't Repeat Yourself) as possible. Luckily this is achievable:

我们对 `&Point` 不可变引用类型的 `Add` 实现，给我们带来了不必要的维护困难。是否能够使得，当我们更改 `Point` 类型的实现时，`&Point` 类型的实现也能够自动发生匹配，而不需要我们手动维护呢？我们的愿望是尽可能写出 `DRY` (Don't Repeat Yourself) 的不重复的代码。幸运的是，我们可以如此实现这一点：

```rust

```
// updated, DRY impl

// 使用一种更“干”的实现

impl Add for &Point {
    type Output = <Point as Add>::Output;

    fn add(self, rhs: &Point) -> Self::Output {
        Point::add(*self, *rhs)
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };

    let p2 = Point { x: 3, y: 4 };

    let line: Line = p1 + p2; //  
  

    let p1 = Point { x: 1, y: 2 };

    let p2 = Point { x: 3, y: 4 };

    let line: Line = &p1 + &p2; //  

}
```

> `AddAssign<Rhs>` types allow us to add + assign `Rhs` types to them. The trait declaration:

实现 `AddAssign<Rhs>` 的类型，允许我们对 `Rhs` 的类型相加之并赋值到自身。该特性的声明为：

```
```rust
trait AddAssign<Rhs = Self> {
 fn add_assign(&mut self, rhs: Rhs);
}

```

```

> Example impls for `Point` and `&Point`:

对 `Point` 和 `&Point` 类型的实现示例：

```
```rust
use std::ops::AddAssign;

#[derive(Copy, Clone)]
struct Point {
 x: i32,
 y: i32
}

impl AddAssign for Point {
 fn add_assign(&mut self, rhs: Point) {
 self.x += rhs.x;
 self.y += rhs.y;
 }
}

impl AddAssign<&Point> for Point {
 fn add_assign(&mut self, rhs: &Point) {
 Point::add_assign(self, *rhs);
 }
}
```

```
}
```

```
fn main() {
 let mut p1 = Point { x: 1, y: 2 };
 let p2 = Point { x: 3, y: 4 };
 p1 += &p2;
 p1 += p2;
 assert!(p1.x == 7 && p1.y == 10);
}
```

```
```
```

闭包特性 Closure Traits

| 特性 | 类别 | 算符 | 描述 |
|----------|----|-------------|---------|
| `Fn` | 闭包 | `(...args)` | 不可变闭包调用 |
| `FnMut` | 闭包 | `(...args)` | 可变闭包调用 |
| `FnOnce` | 闭包 | `(...args)` | 一次性闭包调用 |

FnOnce, FnMut, & Fn

预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Generic Parameters](#generic-parameters)

- [Generic Types vs Associated Types](#generic-types-vs-associated-types)
- [Subtraits & Supertraits](#subtraits--supertraits)

```rust

```
trait FnOnce<Args> {
 type Output;
 fn call_once(self, args: Args) -> Self::Output;
}

trait FnMut<Args>: FnOnce<Args> {
 fn call_mut(&mut self, args: Args) -> Self::Output;
}

trait Fn<Args>: FnMut<Args> {
 fn call(&self, args: Args) -> Self::Output;
}
```

```

> Although these traits exist it's not possible to impl them for our own types in stable Rust. The only types we can create which impl these traits are closures. Depending on what the closure captures from its environment determines whether it impls `FnOnce`, `FnMut`, or `Fn`.

事实上，在 stable Rust 中我们并不能对我们自己的类型实现上述特性，唯一的例外是闭包。对于闭包从环境中捕获的值的不同，该闭包会实现不同的特性：`FnOnce`，`FnMut`，`Fn`。

> An `FnOnce` closure can only be called once because it consumes some value as part of its execution:

对于实现 `FnOnce` 的闭包，仅可调用一次，因为它消耗掉了其执行中必须的值：

```rust

```
fn main() {
 let range = 0..10;
}
```

```
let get_range_count = || range.count();
assert_eq!(get_range_count(), 10); //
get_range_count(); //
}

```

```

> The ` `.count()` method on iterators consumes the iterator so it can only be called once. Hence our closure can only be called once. Which is why when we try to call it a second time we get this error:

迭代器上的 ` `.count()` 方法会消耗掉整个迭代器，所以该方法仅能调用一次。所以我们的闭包也就是能调用一次了，这就是为什么当第二次调用该闭包时会出错：

```
```none
error[E0382]: use of moved value: `get_range_count`
--> src/main.rs:5:5
|
```

```
4 | assert_eq!(get_range_count(), 10);
| ----- `get_range_count` moved due to this call
5 | get_range_count();
| ^^^^^^^^^^^^^^^^ value used here after move
|
```

note: closure cannot be invoked more than once because it moves the variable `range` out of its environment

```
--> src/main.rs:3:30
```

```
|
```

```
3 | let get_range_count = || range.count();
| ^^^^
```

note: this value implements `FnOnce` , which causes it to be moved when called

```
--> src/main.rs:4:16
```

```
|
```

```
4 | assert_eq!(get_range_count(), 10);
| ^^^^^^^^^^^^^^^^
```

```

> An `FnMut` closure can be called multiple times and can also mutate variables it has captured from its environment. We might say `FnMut` closures perform side-effects or are stateful. Here's an example of a closure that filters out all non-ascending values from an iterator by keeping track of the smallest value it has seen so far:

对于实现 `FnMut` 特性的闭包，我们可以多次调用，且其可以改变其从环境捕获的值。我们可以说实现 `FnMut` 的闭包的执行具有副作用，或者说它是具有状态的。下例展示了一个闭包，它通过跟踪最小值，来找到一个迭代器中所有非升序的值：

```
```rust
fn main() {
 let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];
 let mut min = i32::MIN;
 let ascending = nums.into_iter().filter(|&n| {
 if n <= min {
 false
 } else {
 min = n;
 true
 }
 }).collect::<Vec<_>>();
 assert_eq!(vec![0, 4, 8, 10, 15, 18], ascending); //
}
```

```

> `FnMut` refines `FnOnce` in the sense that `FnOnce` requires taking ownership of its arguments and can only be called once, but `FnMut` requires only taking mutable references and can be called multiple times. `FnMut` can be used anywhere `FnOnce` can be used.

`FnMut` 改良了 `FnOnce`，`FnOnce` 需要接管参数的属权因此只能调用一次，而 `FnMut` 只需要参数的可变引用即可并可调用多次。`FnMut` 可以在所有 `FnOnce` 可用的地方使用。

> An `Fn` closure can be called multiple times and does not mutate any variables it has captured from its environment. We might say `Fn` closures have no side-effects or are stateless. Here's an example closure that filters out all values less than some stack variable it captures from its environment from an iterator:

对于实现 `Fn` 特性的闭包，我们可以调用多次，且其不改变任何从环境中捕获的变量。我们可以说实现 `Fn` 的闭包的执行不具有副作用，或者说它是不具有状态的。下例展示了一个闭包，它通过与栈上的值进行比较，过滤掉一个迭代器中所有比它小的值：

```
```rust
fn main() {
 let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];
 let min = 9;
 let greater_than_9 = nums.into_iter().filter(|&n| n > min).collect::<Vec<_>>();
 assert_eq!(vec![10, 15, 18, 13], greater_than_9); //
}
```
``
```

> `Fn` refines `FnMut` in the sense that `FnMut` requires mutable references and can be called multiple times, but `Fn` only requires immutable references and can be called multiple times. `Fn` can be used anywhere `FnMut` can be used, which includes anywhere `FnOnce` can be used.

`Fn` 改良了 `FnMut`，尽管它们都可以多次调用，但是 `FnMut` 需要参数的可变引用，而 `Fn` 仅需要参数的不可变引用。`Fn` 可以在所有 `FnMut` 和 `FnOnce` 可用的地方使用。

> If a closure doesn't capture anything from its environment it's technically not a closure, but just an anonymously declared inline function, and can be casted to, used, and passed around as a regular function pointer, i.e. `fn`. Function pointers can be used anywhere `Fn` can be used, which includes anywhere `FnMut` and `FnOnce` can be used.

如果一个闭包不从环境中捕获任何的值，那么从技术上讲它就不是闭包，而仅仅只是一个内联的匿名函数。并且它可以被转换为、用于或传递为一个常规函数指针，即 `fn`。函数指针可以用于任何 `Fn`， `FnMut`， `FnOnce` 可用的地方。

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

```
fn main() {  
  
    let mut fn_ptr: fn(i32) -> i32 = add_one;  
  
    assert_eq!(fn_ptr(1), 2); //  
  
    // capture-less closure cast to fn pointer  
    // 不捕获环境的闭包可转换为普通函数指针  
    fn_ptr = |x| x + 1; // same as add_one  
  
    assert_eq!(fn_ptr(1), 2); //  
  
}  
  
```
```

> Example of passing a regular function pointer in place of a closure:

以下示例中，将常规函数作为闭包而传入：

```
```rust  
fn main() {  
  
    let nums = vec![-1, 1, -2, 2, -3, 3];  
  
    let absolutes: Vec<i32> = nums.into_iter().map(i32::abs).collect();  
  
    assert_eq!(vec![1, 1, 2, 2, 3, 3], absolutes); //  
  
}
```

其它特性 Other Traits

特性	类别	算符	描述
`Deref`	其它	`*`	不可变解引用

```
| `DerefMut` | 其它 | `*` | 可变解引用 |
| `Drop` | 其它 | - | 类型析构 |
| `Index` | 其它 | `[]` | 不可变索引 |
| `IndexMut` | 其它 | `[]` | 可变索引 |
| `RangeBounds` | 其它 | `..` | 范围迭代 |
```

Deref & DerefMut

预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Sized](#sized)

```rust

```
trait Deref {
 type Target: ?Sized;
 fn deref(&self) -> &Self::Target;
}
```

```
trait DerefMut: Deref {
 fn deref_mut(&mut self) -> &mut Self::Target;
}
```

```

> `Deref<Target = T>` types can be dereferenced to `T` types using the dereference operator `*`. This has obvious use-cases for smart pointer types like `Box` and `Rc`. However, we rarely see the dereference operator explicitly used in Rust code, and that's because of a Rust feature called `_deref coercion_`.

实现 `Deref<Target = T>` 的类型，可以通过 `*` 解引用算符，解引用到 `T` 类型。智能指针是该特性的著名实现者，例如 `Box` 和 `Rc`。不过，我们很少在 Rust 编程中看到解引用算符，这是由于 Rust 的强制解引用的特性所导致的。

> Rust automatically dereferences types when they're being passed as function arguments, returned from a function, or used as part of a method call. This is the reason why we can pass `&String` and `&Vec<T>` to functions expecting `&str` and `&[T]` because `String` implements `Deref<Target = str>` and `Vec<T>` implements `Deref<Target = [T]>`.

当作为函数的参数、函数的返回值、方法的调用参数时，Rust 会自动地解引用。这就是为什么我们可以将 `&String` 或 `&Vec<T>` 类型的值作为参数传递给接受 `str` 或 `&[T]` 类型的参数的函数，因为 `String` 实现了 `Deref<Target = str>`，`Vec<T>` 实现了 `Deref<Target = [T]>`。

> `Deref` and `DerefMut` should only be implemented for smart pointer types. The most common way people attempt to misuse and abuse these traits is to try to shoehorn some kind of OOP-style data inheritance into Rust. This does not work. Rust is not OOP. Let's examine a few different situations where, how, and why it does not work. Let's start with this example:

`Deref` 和 `DerefMut` 仅应实现于智能指针类型。最常见的误用或滥用就是，人们经常希望强行把某种面向对象编程风格的数据继承塞到 Rust 编程中。这是不可能的，因为 Rust 不是面向对象的。让我们用一个例子来领会到底为什么这是不可以的：

```
```rust
```

```
use std::ops::Deref;
```

```
struct Human {
```

```
 health_points: u32,
```

```
}
```

```
enum Weapon {
```

```
 Spear,
```

```
 Axe,
```

```
 Sword,
```

```
}
```

```
// a Soldier is just a Human with a Weapon
```

```
// 士兵是手持武器的人类
```

```
struct Soldier {
```

```
 human: Human,
```

```
 weapon: Weapon,
```

```
}
```

```
impl Deref for Soldier {
```

```
 type Target = Human;
```

```
 fn deref(&self) -> &Human {
```

```
 &self.human
```

```
}
```

```
}
```

```
enum Mount {
```

```
 Horse,
```

```
 Donkey,
```

```
 Cow,
```

```
}
```

```
// a Knight is just a Soldier with a Mount
```

```
// 骑士是跨骑坐骑的士兵
```

```
struct Knight {
```

```
 soldier: Soldier,
```

```
 mount: Mount,
```

```
}
```

```
impl Deref for Knight {
```

```
 type Target = Soldier;
```

```
 fn deref(&self) -> &Soldier {
```

```
 &self.soldier
```

```
}

}

enum Spell {
 MagicMissile,
 FireBolt,
 ThornWhip,
}

// a Mage is just a Human who can cast Spells
// 法师是口诵咒语的人类
struct Mage {
 human: Human,
 spells: Vec<Spell>,
}

impl Deref for Mage {
 type Target = Human;
 fn deref(&self) -> &Human {
 &self.human
 }
}

enum Staff {
 Wooden,
 Metallic,
 Plastic,
}

// a Wizard is just a Mage with a Staff
```

```
// 巫师是腰别法宝的法师
struct Wizard {
 mage: Mage,
 staff: Staff,
}

impl Deref for Wizard {
 type Target = Mage;
 fn deref(&self) -> &Mage {
 &self.mage
 }
}

fn borrows_human(human: &Human) {}
fn borrows_soldier(soldier: &Soldier) {}
fn borrows_knight(knight: &Knight) {}
fn borrows_mage(mage: &Mage) {}
fn borrows_wizard(wizard: &Wizard) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
 // all types can be used as Humans
 borrows_human(&human);
 borrows_human(&soldier);
 borrows_human(&knight);
 borrows_human(&mage);
 borrows_human(&wizard);

 // Knights can be used as Soldiers
 borrows_soldier(&soldier);
 borrows_soldier(&knight);

 // Wizards can be used as Mages
}
```

```
borrows_mage(&mage);
borrows_mage(&wizard);

// Knights & Wizards passed as themselves

borrows_knight(&knight);

borrows_wizard(&wizard);

}

```

```

> So at first glance the above looks pretty good! However it quickly breaks down to scrutiny. First of all, deref coercion only works on references, so it doesn't work when we actually want to pass ownership:

事实上，并不可以这么做。首先，强制解引用仅用于引用，所以我们不能移交属权：

```rust

```
fn takes_human(human: Human) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
 // all types CANNOT be used as Humans

 takes_human(human);

 takes_human(soldier); //

 takes_human(knight); //

 takes_human(mage); //

 takes_human(wizard); //

}
```

```

> Furthermore, deref coercion doesn't work in generic contexts. Let's say we impl some trait only on humans:

其次，强制解引用不可用于泛型编程。例如某特性仅对人类实现：

```rust

```

trait Rest {
fn rest(&self);
}

impl Rest for Human {
fn rest(&self) {}

}

fn take_rest<T: Rest>(rester: &T) {
rester.rest()
}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
// all types CANNOT be used as Rest types, only Human
take_rest(&human);
take_rest(&soldier); //
take_rest(&knight); //
take_rest(&mage); //
take_rest(&wizard); //
}
```

```

> Also, although deref coercion works in a lot of places it doesn't work everywhere. It doesn't work on operands, even though operators are just syntax sugar for method calls. Let's say, to be cute, we wanted `Mage`'s to learn `Spell`'s using the `+=` operator:

强制解引用可以用于许多情况，但绝不是所有情况。例如对于算符的操作数而言就不行，即便算符仅是一种方法调用的语法糖。比如，我们希望使用 `+=` 算符来表达法师学习咒语。

```

```rust
impl DerefMut for Wizard {
fn deref_mut(&mut self) -> &mut Mage {

```

```

&mut self.mage
}

}

impl AddAssign<Spell> for Mage {
fn add_assign(&mut self, spell: Spell) {
self.spells.push(spell);
}
}

fn example(mut mage: Mage, mut wizard: Wizard, spell: Spell) {
mage += spell;
wizard += spell; // wizard not coerced to mage here
// 在这里，巫师不能强制转换为法师
wizard.add_assign(spell); // oof, we have to call it like this
// 所以，我们必须这样做
}
...

```

> In languages with OOP-style data inheritance the value of `self` within a method is always equal to the type which called the method but in the case of Rust the value of `self` is always equal to the type which implemented the method:

在带有面向对象风格的数据继承的语言中，方法中的 `self` 值的类型总是等同于调用该方法的类型。但是在 Rust 语言中，`self` 值的类型总是等同于实现该方法时的类型。

```

```rust
struct Human {
profession: &'static str,
health_points: u32,
}

```

```
impl Human {  
    // self will always be a Human here, even if we call it on a Soldier  
    // 该方法中的 self 的类型永远是 Human , 即便我们在 Soldier 类型上调用  
    fn state_profession(&self) {  
        println!("I'm a {}!", self.profession);  
    }  
}
```

```
struct Soldier {  
    profession: &'static str,  
    human: Human,  
    weapon: Weapon,  
}
```

```
fn example(soldier: &Soldier) {  
    assert_eq!("servant", soldier.human.profession);  
    assert_eq!("spearman", soldier.profession);  
    soldier.human.state_profession(); // prints "I'm a servant!"  
    soldier.state_profession(); // still prints "I'm a servant!"  
}  
...
```

> The above gotcha is especially damning when impling `Deref` or `DerefMut` on a newtype. Let's say we want to create a `SortedVec` type which is just a `Vec` but it's always in sorted order. Here's how we might do that:

上述特性常令人感到困惑，特别是在对新类型实现 `Deref` 和 `DerefMut` 的时候。例如我们想要设计一个 `SortedVec` 类型，相比于 `Vec` 类型，它总是处于已排序的状态。我们可能会这样做：

```
```rust
```

```
struct SortedVec<T: Ord>(Vec<T>);
```

```
impl<T: Ord> SortedVec<T> {
```

```

fn new(mut vec: Vec<T>) -> Self {
 vec.sort();
 SortedVec(vec)
}

fn push(&mut self, t: T) {
 self.0.push(t);
 self.0.sort();
}
}

```

```

> Obviously we cannot impl `DerefMut<Target = Vec<T>>` here or anyone using `SortedVec` would be able to trivially break the sorted order. However, impling `Deref<Target = Vec<T>>` surely must be safe, right? Try to spot the bug in the program below:

显然我们不能为其实现 `DerefMut<Target = Vec<T>>`，因为这可能会破坏排序状态。实现 `Deref<Target = Vec<T>>` 必须要保证功能的正确性。尝试指出下列代码中的 bug：

```

```rust
use std::ops::Deref;

struct SortedVec<T: Ord>(Vec<T>);

impl<T: Ord> SortedVec<T> {
 fn new(mut vec: Vec<T>) -> Self {
 vec.sort();
 SortedVec(vec)
 }

 fn push(&mut self, t: T) {
 self.0.push(t);
 self.0.sort();
 }
}

```

```
}
```

```
impl<T: Ord> Deref for SortedVec<T> {
```

```
 type Target = Vec<T>;
```

```
 fn deref(&self) -> &Vec<T> {
```

```
 &self.0
```

```
 }
```

```
}
```

```
fn main() {
```

```
 let sorted = SortedVec::new(vec![2, 8, 6, 3]);
```

```
 sorted.push(1);
```

```
 let sortedClone = sorted.clone();
```

```
 sortedClone.push(4);
```

```
}
```

```
```
```

> We never implemented `Clone` for `SortedVec` so when we call the `clone()` method the compiler is using deref coercion to resolve that method call on `Vec` and so it returns a `Vec` and not a `SortedVec`!

鉴于我们从未对 `SortedVec` 实现 `Clone` 特性，所以当我们调用 `clone()` 方法的时候，编译器会使用强制解引用将该方法调用解析为 `Vec` 的方法调用，所以该方法返回的是 `Vec` 而不是 `SortedVec`！

```
```rust
```

```
fn main() {
```

```
 let sorted: SortedVec<i32> = SortedVec::new(vec![2, 8, 6, 3]);
```

```
 sorted.push(1); // still sorted
```

```
// calling clone on SortedVec actually returns a Vec
```

```
 let sortedClone: Vec<i32> = sorted.clone();
```

```
 sortedClone.push(4); // sortedClone no longer sorted
```

```
}
```

> Anyway, none of the above limitations, constraints, or gotchas are faults of Rust because Rust was never designed to be an OO language or to support any OOP patterns in the first place.

切记，Rust 并非设计为面向对象的语言，也并不将面向对象编程的模式作为一等公民，所以以上的限制、约束和令人困惑的特性并不被认为是在语言中是错误的。

> The main takeaway from this section is do not try to be cute or clever with `Deref` and `DerefMut` impls. They're really only appropriate for smart pointer types, which can only be implemented within the standard library for now as smart pointer types currently require unstable features and compiler magic to work. If we want functionality and behavior similar to `Deref` and `DerefMut` then what we're actually probably looking for is `AsRef` and `AsMut` which we'll get to later.

本节的主旨即是使读者领会为什么不要自作聪明地实现 `Deref` 和 `DerefMut` 特性。这类特性确仅适合于智能指针类型的类型，目前来讲标准库中的智能指针的实现，确需要这样的不稳定特性以及一些编译器魔法才能工作。如果我们确需要一些类似于 `Deref` 和 `DerefMut` 的特性，不妨使用 `AsRef` 和 `AsMut` 特性。我们将在后面的章节中对这类特性做出说明。

#### #### Index & IndexMut

##### 预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Generic Parameters](#generic-parameters)
- [Generic Types vs Associated Types](#generic-types-vs-associated-types)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Sized](#sized)

```rust

```
trait Index<Idx: ?Sized> {
```

```

type Output: ?Sized;
fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> where Idx: ?Sized {
fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
...

```

> We can index `[]` into `Index<T, Output = U>` types with `T` values and the index operation will return `&U` values. For syntax sugar, the compiler auto inserts a deref operator `*` in front of any value returned from an index operation:

对于实现 `Index<T, Output = U>` 的类型，我们可以使用 `[]` 索引算符对 `T` 类型的值索引 `&U` 类型的值。作为语法糖，编译器也会为索引操作返回的值自动添加一个 `*` 解引用算符。

```

```rust
fn main() {
 // Vec<i32> impls Index<usize, Output = i32> so
 // indexing Vec<i32> should produce &i32s and yet...
 // 鉴于 Vec<i32> 实现了 Index<usize, Output = i32>
 // 所以对 Vec<i32> 的索引应当返回 &i32 类型的值，但是。。
 let vec = vec![1, 2, 3, 4, 5];
 let num_ref: &i32 = vec[0]; // expected &i32 found i32
 // above line actually desugars to
 // 以上代码等价于
 let num_ref: &i32 = *vec[0]; // expected &i32 found i32
 // both of these alternatives work
 // 以下是建议使用的一对形式
 let num: i32 = vec[0];
}
```

```
let num_ref: &i32 = &vec[0]; //
}
...
...
```

> It's kinda confusing at first, because it seems like the `Index` trait does not follow its own method signature, but really it's just questionable syntax sugar.

令人困惑的是，似乎 `Index` 特性没有遵循它自己的方法签名，但其实真正有问题的是语法糖。

> Since `Idx` is a generic type the `Index` trait can be implemented many times for a given type, and in the case of `Vec<T>` not only can we index into it using `usize` but we can also index into its using `Range<usize>`'s to get slices.

鉴于 `Idx` 是泛型类型，`Index` 特性对多个给定类型可以多次实现。并且对于 `Vec<T>`，我们不仅可以对 `usize` 索引，还可以对 `Range<usize>` 索引得到切片。

```rust

```
fn main() {  
  
    let vec = vec![1, 2, 3, 4, 5];  
  
    assert_eq!(&vec[..], &[1, 2, 3, 4, 5]); //  
    assert_eq!(&vec[1..], &[2, 3, 4, 5]); //  
    assert_eq!(&vec[..4], &[1, 2, 3, 4]); //  
    assert_eq!(&vec[1..4], &[2, 3, 4]); //  
}
```

> To show off how we might impl `Index` ourselves here's a fun example which shows how we can use a newtype and the `Index` trait to impl wrapping indexes and negative indexes on a `Vec`:

为了展示如何自己实现 `Index` 特性，以下是一个有趣的例子，它设计了一个 `Vec` 的包装结构，其使得循环索引和负数索引成为可能：

```rust

```
use std::ops::Index;
```

```
struct WrappingIndex<T>(Vec<T>);

impl<T> Index<usize> for WrappingIndex<T> {
 type Output = T;

 fn index(&self, index: usize) -> &T {
 &self.0[index % self.0.len()]
 }
}

impl<T> Index<i128> for WrappingIndex<T> {
 type Output = T;

 fn index(&self, index: i128) -> &T {
 let self_len = self.0.len() as i128;
 let idx = (((index % self_len) + self_len) % self_len) as usize;
 &self.0[idx]
 }
}

#[test] //
fn indexes() {
 let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
 assert_eq!(1, wrapping_vec[0_usize]);
 assert_eq!(2, wrapping_vec[1_usize]);
 assert_eq!(3, wrapping_vec[2_usize]);
}

#[test] //
fn wrapping_indexes() {
 let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
```

```

assert_eq!(1, wrapping_vec[3_usize]);
assert_eq!(2, wrapping_vec[4_usize]);
assert_eq!(3, wrapping_vec[5_usize]);
}

#[test] //

fn neg_indexes() {

let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
assert_eq!(1, wrapping_vec[-3_i128]);
assert_eq!(2, wrapping_vec[-2_i128]);
assert_eq!(3, wrapping_vec[-1_i128]);
}

#[test] //

fn wrapping_neg_indexes() {

let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
assert_eq!(1, wrapping_vec[-6_i128]);
assert_eq!(2, wrapping_vec[-5_i128]);
assert_eq!(3, wrapping_vec[-4_i128]);
}

```

```

> There's no requirement that the `Idx` type has to be a number type or a `Range`, it could be an enum! Here's an example using basketball positions to index into a basketball team to retrieve players on the team:

`Idx` 的类型并不非得是数字类型或 `Range` 类型，甚至还可以是枚举！例如我们可以在一支篮球队中，对打什么位置索引从而得到队伍里打这个位置的队员：

```rust

```
use std::ops::Index;
```

```
enum BasketballPosition {
```

```
PointGuard,
ShootingGuard,
Center,
PowerForward,
SmallForward,
}
```

```
struct BasketballPlayer {
 name: &'static str,
 position: BasketballPosition,
}
```

```
struct BasketballTeam {
 point_guard: BasketballPlayer,
 shooting_guard: BasketballPlayer,
 center: BasketballPlayer,
 power_forward: BasketballPlayer,
 small_forward: BasketballPlayer,
}
```

```
impl Index<BasketballPosition> for BasketballTeam {
 type Output = BasketballPlayer;
 fn index(&self, position: BasketballPosition) -> &BasketballPlayer {
 match position {
 BasketballPosition::PointGuard => &self.point_guard,
 BasketballPosition::ShootingGuard => &self.shooting_guard,
 BasketballPosition::Center => &self.center,
 BasketballPosition::PowerForward => &self.power_forward,
 BasketballPosition::SmallForward => &self.small_forward,
 }
 }
}
```

```
}
```

```
}
```

```
...
```

```
Drop
```

## 预备知识

- [Self](#self)
- [Methods](#methods)

```
```rust
```

```
trait Drop {  
    fn drop(&mut self);  
}
```

```
...
```

> If a type implements `Drop` then `drop` will be called on the type when it goes out of scope but before it's destroyed. We will rarely need to implement this for our types but a good example of where it's useful is if a type holds on to some external resources which needs to be cleaned up when the type is destroyed.

对于实现 `Drop` 特性的类型，在该类型脱离作用域并销毁前，其 `drop` 方法会被调用。通常，不必为我们的类型实现这一特性，除非该类型持有某种外部的资源，且该资源需要显式释放。

> There's a `BufWriter` type in the standard library that allows us to buffer writes to `Write` types. However, what if the `BufWriter` gets destroyed before the content in its buffer has been flushed to the underlying `Write` type? Thankfully that's not possible! The `BufWriter` implements the `Drop` trait so that `flush` is always called on it whenever it goes out of scope!

标准库中的 `BufWriter` 类型允许我们对向 `Write` 类型写入的时候进行缓存。显然，当 `BufWriter` 销毁前应当把缓存的内容写入 `Writer` 实例，这就是 `Drop` 所允许我们做到的！对于实现了 `Drop` 的 `BufWriter` 来说，其实例在销毁前会总会调用 `flush` 方法。

```
```rust
```

```
impl<W: Write> Drop for BufWriter<W> {
fn drop(&mut self) {
self.flush_buf();
}
}
}
```
```

> Also, `Mutex`'s in Rust don't have `unlock()` methods because they don't need them! Calling `lock()` on a `Mutex` returns a `MutexGuard` which automatically unlocks the `Mutex` when it goes out of scope thanks to its `Drop` impl:

并且，在 Rust 中 `Mutex` 类型之所以没有 `unlock()` 方法，就是因为它完全不需要！鉴于 `Drop` 特性的实现，调用 `Mutex` 的 `lock()` 方法返回的 `MutexGuard` 类型，在脱离作用域时会自动地释放 `Mutex`。

```rust

```
impl<T: ?Sized> Drop for MutexGuard<'_, T> {
fn drop(&mut self) {
unsafe {
self.lock.inner.raw_unlock();
}
}
}
}
```
```

> In general, if you're impling an abstraction over some resource that needs to be cleaned up after use then that's a great reason to make use of the `Drop` trait.

简而言之，如果你正在设计某种需要显示释放的资源的抽象包装，那么这正是 `Drop` 特性大显神威的地方。

转换特性 Conversion Traits

From & Into

预备知识

- [Self](#self)
- [Functions](#functions)
- [Methods](#methods)
- [Generic Parameters](#generic-parameters)
- [Generic Blanket Impls](#generic-blanket-impls)

```rust

```
trait From<T> {
 fn from(T) -> Self;
}
...
```

> `From<T>` types allow us to convert `T` into `Self`.

实现 `From<T>` 特性的类型允许我们从 `T` 类型转换到自身的类型 `Self`。

```rust

```
trait Into<T> {  
    fn into(self) -> T;  
}  
...
```

> `Into<T>` types allow us to convert `Self` into `T`.

实现 `Into<T>` 特性的类型允许我们从自身的类型 `Self` 转换到 `T` 类型。

> These traits are two different sides of the same coin. We can only impl `From<T>` for our types because the `Into<T>` impl is automatically provided by this generic blanket impl:

这是一对恰好相反的特性，如同一枚硬币的两面。注意，我们只能手动实现 `From<T>` 特性，而不能手动实现 `Into<T>` 特性，因为 `Into<T>` 特性已经被一揽子泛型实现所自动实现。

```rust

```
impl<T, U> Into<U> for T
```

where

```
U: From<T>,
```

```
{
```

```
fn into(self) -> U {
```

```
 U::from(self)
```

```
}
```

```
}
```

```
```
```

> The reason both traits exist is because it allows us to write trait bounds on generic types slightly differently:

这两个特性同时存在的一个好处在于，我们可以在为泛型类型添加约束的时候，使用两种稍有不同的记号：

```rust

```
fn function<T>(t: T)
```

where

```
// these bounds are equivalent
```

```
// 以下两种记号等价
```

```
T: From<i32>,
```

```
i32: Into<T>
```

```
{
```

```
// these examples are equivalent
```

```
// 以下两种记号等价
let example: T = T::from(0);
let example: T = 0.into();
}

```

```

> There are no hard rules about when to use one or the other, so go with whatever makes the most sense for each situation. Now let's look at some example impls on `Point`:

对于具体使用哪种记号并无一定之规，请根据实际情况做出最恰当的选择。接下来我们看看 `Point` 类型的例子：

```rust

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

```
impl From<(i32, i32)> for Point {
 fn from((x, y): (i32, i32)) -> Self {
 Point { x, y }
 }
}
```

```
impl From<[i32; 2]> for Point {
 fn from([x, y]: [i32; 2]) -> Self {
 Point { x, y }
 }
}
```

```
fn example() {
 // using From
}
```

```
let origin = Point::from((0, 0));
let origin = Point::from([0, 0]);

// using Into

let origin: Point = (0, 0).into();
let origin: Point = [0, 0].into();
}
```

> The impl is not symmetric, so if we'd like to convert `Point`s into tuples and arrays we have to explicitly add those as well:

这样的转换并不是对称的，如果我们想将 `Point` 转换为元组或数组，那么我们需要显式地编写相应的代码：

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

impl From<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Self {
        Point { x, y }
    }
}

impl From<Point> for (i32, i32) {
    fn from(Point { x, y }: Point) -> Self {
        (x, y)
    }
}
```

```
impl From<[i32; 2]> for Point {  
    fn from([x, y]: [i32; 2]) -> Self {  
        Point { x, y }  
    }  
}
```

```
impl From<Point> for [i32; 2] {  
    fn from(Point { x, y }: Point) -> Self {  
        [x, y]  
    }  
}
```

```
fn example() {  
    // from (i32, i32) into Point  
    let point = Point::from((0, 0));  
    let point: Point = (0, 0).into();
```

```
// from Point into (i32, i32)  
let tuple = <(i32, i32)>::from(point);  
let tuple: (i32, i32) = point.into();
```

```
// from [i32; 2] into Point  
let point = Point::from([0, 0]);  
let point: Point = [0, 0].into();
```

```
// from Point into [i32; 2]  
let array = <[i32; 2]>::from(point);  
let array: [i32; 2] = point.into();  
}
```

```

> A popular use of `From<T>` is to trim down boilerplate code. Let's say we add a `Triangle` type to our program which contains three `Point`s, here's some of the many ways we can construct it:

借由 `From<T>` 特性，我们可以省却大量编写模板代码的麻烦。例如，我们现在具有一个包含三个 `Point` 的类型 `Triangle` 类型，以下是构造该类型的几种办法：

```rust

```
struct Point {
```

```
    x: i32,
```

```
    y: i32,
```

```
}
```

```
impl Point {
```

```
    fn new(x: i32, y: i32) -> Point {
```

```
        Point { x, y }
```

```
    }
```

```
}
```

```
impl From<(i32, i32)> for Point {
```

```
    fn from((x, y): (i32, i32)) -> Point {
```

```
        Point { x, y }
```

```
    }
```

```
}
```

```
struct Triangle {
```

```
    p1: Point,
```

```
    p2: Point,
```

```
    p3: Point,
```

```
}
```

```
impl Triangle {  
fn new(p1: Point, p2: Point, p3: Point) -> Triangle {  
Triangle { p1, p2, p3 }  
}  
}  
}
```

impl<P> From<[P; 3]> for Triangle

where

```
P: Into<Point>  
{  
fn from([p1, p2, p3]: [P; 3]) -> Triangle {  
Triangle {  
p1: p1.into(),  
p2: p2.into(),  
p3: p3.into(),  
}  
}  
}  
}
```

```
fn example() {  
// manual construction  
let triangle = Triangle {  
p1: Point {  
x: 0,  
y: 0,  
},  
p2: Point {  
x: 1,  
y: 1,  
},
```

```
p3: Point {  
    x: 2,  
    y: 2,  
},  
};  
  
// using Point::new  
let triangle = Triangle {  
    p1: Point::new(0, 0),  
    p2: Point::new(1, 1),  
    p3: Point::new(2, 2),  
};  
  
// using From<(i32, i32)> for Point  
let triangle = Triangle {  
    p1: (0, 0).into(),  
    p2: (1, 1).into(),  
    p3: (2, 2).into(),  
};  
  
// using Triangle::new + From<(i32, i32)> for Point  
let triangle = Triangle::new(  
    (0, 0).into(),  
    (1, 1).into(),  
    (2, 2).into(),  
);  
  
// using From<[Into<Point>; 3]> for Triangle  
let triangle: Triangle = [  
    (0, 0),
```

```
(1, 1),  
(2, 2),  
.into();  
}  
...  
  
``
```

> There are no rules for when, how, or why we should impl `From<T>` for our types so it's up to us to use our best judgement for every situation.

对于 `From<T>` 特性的使用并无一定之规，运用你的智慧明智地使用它吧！

> One popular use of `Into<T>` is to make functions which need owned values generic over whether they take owned or borrowed values:

使用 `Into<T>` 特性的一个神奇之处在于，对于那些本来只能接受特定类型参数的函数，现在你可以有更多不同的选择：

```rust

```
struct Person {
 name: String,
}
```

```
impl Person {
 // accepts:
 // - String
 fn new1(name: String) -> Person {
 Person { name }
 }
```

```
// accepts:
// - String
// - &String
// - &str
```

```

// - Box<str>
// - Cow<'_, str>
// - char
// since all of the above types can be converted into String

fn new2<N: Into<String>>(name: N) -> Person {
 Person { name: name.into() }
}

}
```

```

错误处理 Error Handling

> The best time to talk about error handling and the `Error` trait is after going over `Display`, `Debug`, `Any`, and `From` but before getting to `TryFrom` hence why the **Error Handling** section awkwardly bisects the **Conversion Traits** section.

讲解错误处理与 `Error` 特性的最佳时机，莫过于在 `Display`， `Debug`， `Any` 和 `From` 之后， `TryFrom` 之前，这就是为什么我要将 **错误处理** 这一节硬塞在 **转换特性** 这一章里。

Error

预备知识

- [Self](#self)
- [Methods](#methods)
- [Default Impl](#default-impls)
- [Generic Blanket Impl](#generic-blanket-impls)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Trait Objects](#trait-objects)
- [Display & ToString](#display--tostring)
- [Debug](#debug)

- [Any](#any)
- [From & Into](#from--into)

```rust

```
trait Error: Debug + Display {
 // provided default impls
 // 提供默认实现
 fn source(&self) -> Option<&(dyn Error + 'static)>;
 fn backtrace(&self) -> Option<&Backtrace>;
 fn description(&self) -> &str;
 fn cause(&self) -> Option<&dyn Error>;
}
````
```

> In Rust errors are returned, not thrown. Let's look at some examples.

在 Rust 中，错误是被返回的，而不是被抛出的。让我们看看下面的例子：

> Since dividing integer types by zero panics if we wanted to make our program safer and more explicit we could impl a `safe_div` function which returns a `Result` instead like this:

由于整数的除零操作会导致 panic，为了程序的健壮性，我们显式地实现了安全的 `safe_div` 除法函数，它的返回值是 `Result`：

```rust

```
use std::fmt;

use std::error;

#[derive(Debug, PartialEq)]

struct DivByZero;
```

```

impl fmt::Display for DivByZero {
fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
write!(f, "division by zero error")
}

}

impl error::Error for DivByZero {}

fn safe_div(numerator: i32, denominator: i32) -> Result<i32, DivByZero> {
if denominator == 0 {
return Err(DivByZero);
}
Ok(numerator / denominator)
}

#[test] //
fn test_safe_div() {
assert_eq!(safe_div(8, 2), Ok(4));
assert_eq!(safe_div(5, 0), Err(DivByZero));
}
```

```

> Since errors are returned and not thrown they must be explicitly handled, and if the current function cannot handle an error it should propagate it up to the caller. The most idiomatic way to propagate errors is to use the `?` operator, which is just syntax sugar for the now deprecated `try!` macro which simply does this:

由于错误是被返回的，而不是被抛出的，它们必须被显式地处理。如果当前函数没有处理该错误的能力，那么该错误应当原路返回到上一级调用函数。最理想的返回错误的方法是使用 `?` 算符，它是现在已经过时的 `try!` 宏的语法糖：

```

```rust
macro_rules! try {
($expr:expr) => {

```

```

match $expr {
// if Ok just unwrap the value

// 正常情况下直接解除 Result 的包装

Ok(val) => val,

// if Err map the err value using From and return

// 否则将该错误进行适当转换后，返回到上级调用函数

Err(err) => {

return Err(From::from(err));

}

}

};

}

}

```

```

> If we wanted to write a function which reads a file into a `String` we could write it like this, propagating the `io::Error`'s using `?` everywhere they can appear:

例如，如果我们的函数其功能是将文件读为一个 `String`，那么使用 `?` 算符来将可能的错误 `io::Error` 返回给上级调用函数就很方便：

```

```rust

use std::io::Read;

use std::path::Path;

use std::io;

use std::fs::File;

fn read_file_to_string(path: &Path) -> Result<String, io::Error> {

let mut file = File::open(path)?; // io::Error

let mut contents = String::new();

file.read_to_string(&mut contents)?; // io::Error

Ok(contents)

}

```

...

> But let's say the file we're reading is actually a list of numbers and we want to sum them together, we'd update our function like this:

又例如，如果我们的文件是一系列数字，我们想将它们加在一起，可以这样编写代码：

```rust

```
use std::io::Read;
use std::path::Path;
use std::io;
use std::fs::File;

fn sum_file(path: &Path) -> Result<i32, /* What to put here? */ {
    // 这里填写什么类型好呢?

    let mut file = File::open(path)?; // io::Error
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // io::Error

    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?; // ParseIntError
    }
    Ok(sum)
}
```

> But what's the error type of our `Result` now? It can return either an `io::Error` or a `ParseIntError`. We're going to look at three approaches for solving this problem, starting with the most quick & dirty way and finishing with the most robust way.

现在 `Result` 的类型又如何？该函数内部可能产生 `io::Error` 或 `ParseIntError` 两种错误。我们将介绍三种解决此类问题的方法，从最简单但不优雅的方法，到最健壮的方法：

> The first approach is recognizing that all types which impl `Error` also impl `Display` so we can map all the errors to `String`s and use `String` as our error type:

方法一，我们注意到，所有实现了 `Error` 的类型同时也实现了 `Display`，因此我们可以将错误映射到 `String` 并以此为错误类型：

```rust

```
use std::fs::File;
use std::io;
use std::io::Read;
use std::path::Path;

fn sum_file(path: &Path) -> Result<i32, String> {
 let mut file = File::open(path)
 .map_err(|e| e.to_string())?; // io::Error -> String

 let mut contents = String::new();
 file.read_to_string(&mut contents)
 .map_err(|e| e.to_string())?; // io::Error -> String

 let mut sum = 0;
 for line in contents.lines() {
 sum += line.parse::<i32>()
 .map_err(|e| e.to_string())?; // ParseIntError -> String
 }
 Ok(sum)
}
```

> The obvious downside of stringifying every error is that we throw away type information which makes it harder for the caller to handle the errors.

此方法的明显缺点在于，由于我们将所有的错误都序列化了，以至于丢弃了该错误的类型信息，这对于上级调用函数错误处理来讲，就不是那么方便了。

> One nonobvious upside to the above approach is we can customize the strings to provide more context-specific information. For example, `ParseIntError` usually stringifies to “invalid digit found in string” which is very vague and doesn’t mention what the invalid string is or what integer type it was trying to parse into. If we were debugging this problem that error message would almost be useless. However we can make it significantly better by providing all the context relevant information ourselves:

但此方法也有一个不明显的优点，那就是我们可以使用自定义的字符串，来提供丰富的上下文错误信息。例如，`ParseIntError` 通常序列化为 “invalid digit found in string” 这样模棱两可的文本，既没有提及无效的字符串是什么，也没有提及它要转换到什么样的数字类型。这样的信息对于我们调试程序来讲几乎没有帮助。不过我们可以提供更有意义的，且上下文相关的信息来明显改善这一点：

```
```rust
sum += line.parse::<i32>()

.map_err(|_| format!("failed to parse {} into i32", line))?;

```

```

> The second approach takes advantage of this generic blanket impl from the standard library:

方法二，利用标准库的一揽子泛型实现：

```
```rust
impl<E: error::Error> From<E> for Box<dyn error::Error>;
```

```

> Which means that any `Error` type can be implicitly converted into a `Box<dyn error::Error>` by the `?` operator, so we can set the error type to `Box<dyn error::Error>` in the `Result` return type of any function which produces errors and the `?` operator will do the rest of the work for us:

所有实现了 `Error` 特性的类型都可以隐式地使用 `?` 转换为 `Box<dyn error::Error>` 类型。所以我们可以将 `Result` 的错误类型设为 `Box<dyn error::Error>` 类型，然后 `?` 算符会帮我们实现这一隐式转换。

```
```rust
use std::fs::File;
use std::io::Read;
use std::path::Path;
```

```

```

use std::error;

fn sum_file(path: &Path) -> Result<i32, Box<dyn error::Error>> {
 let mut file = File::open(path)?; // io::Error -> Box<dyn error::Error>
 let mut contents = String::new();
 file.read_to_string(&mut contents)?; // io::Error -> Box<dyn error::Error>
 let mut sum = 0;
 for line in contents.lines() {
 sum += line.parse::<i32>()?;
 }
 Ok(sum)
}
```

```

> While being more concise, this seems to suffer from the same downside of the previous approach by throwing away type information. This is mostly true, but if the caller is aware of the impl details of our function they can still handle the different errors types using the `downcast_ref()` method on `error::Error` which works the same as it does on `dyn Any` types:

这看起来似乎有与第一种方法一样的缺点，丢弃了错误的类型信息。有时确实如此，但倘若上级调用函数知悉该函数的实现细节，那么它仍然可以通过 `error::Error` 特性的 `downcast_ref()` 方法来分辨错误的具体类型，这与实现了 `dyn Any` 特性的类型是一样的：

```

```rust
fn handle_sum_file_errors(path: &Path) {
 match sum_file(path) {
 Ok(sum) => println!("the sum is {}", sum),
 Err(err) => {
 if let Some(e) = err.downcast_ref::<io::Error>() {
 // handle io::Error
 } else if let Some(e) = err.downcast_ref::<ParseIntError>() {
 // handle ParseIntError
 } else {

```

```
// we know sum_file can only return one of the
// above errors so this branch is unreachable
// 由于我们知道该函数只能返回以上两种错误,
// 所以这一选择肢一般是不可能执行的
unreachable!();
}
}
}
}
```

```

> The third approach, which is the most robust and type-safe way to aggregate these different errors would be to build our own custom error type using an enum:

方法三，处理错误的最健壮和类型安全的方法，是通过枚举来构建我们自己的错误类型：

```
```rust
use std::num::ParseIntError;
use std::fs::File;
use std::io;
use std::io::Read;
use std::path::Path;
use std::error;
use std::fmt;

#[derive(Debug)]
enum SumFileError {
 Io(io::Error),
 Parse(ParseIntError),
}
```

```
impl From<io::Error> for SumFileError {
 fn from(err: io::Error) -> Self {
 SumFileError::Io(err)
 }
}

impl From<ParseIntError> for SumFileError {
 fn from(err: ParseIntError) -> Self {
 SumFileError::Parse(err)
 }
}

impl fmt::Display for SumFileError {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 match self {
 SumFileError::Io(err) => write!(f, "sum file error: {}", err),
 SumFileError::Parse(err) => write!(f, "sum file error: {}", err),
 }
 }
}

impl error::Error for SumFileError {
 // the default impl for this method always returns None
 // but we can now override it to make it way more useful!
 // 在默认实现中，该方法总是返回 None，现在重写它！
 fn source(&self) -> Option<&(dyn error::Error + 'static)> {
 Some(match self {
 SumFileError::Io(err) => err,
 SumFileError::Parse(err) => err,
 })
 }
}
```

```
}

fn sum_file(path: &Path) -> Result<i32, SumFileError> {

let mut file = File::open(path)?; // io::Error -> SumFileError

let mut contents = String::new();

file.read_to_string(&mut contents)?; // io::Error -> SumFileError

let mut sum = 0;

for line in contents.lines() {

sum += line.parse::<i32>()?; // ParseIntError -> SumFileError

}

Ok(sum)

}

fn handle_sum_file_errors(path: &Path) {

match sum_file(path) {

Ok(sum) => println!("the sum is {}", sum),

Err(SumFileError::Io(err)) => {

// handle io::Error

},

Err(SumFileError::Parse(err)) => {

// handle ParseIntError

},

}

}

```
    ...

```

转换特性深入 Conversion Traits Continued

TryFrom & TryInto

预备知识

- [Self](#self)
- [Functions](#functions)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Generic Parameters](#generic-parameters)
- [Generic Types vs Associated Types](#generic-types-vs-associated-types)
- [Generic Blanket Impls](#generic-blanket-impls)
- [From & Into](#from--into)
- [Error](#error)

> `TryFrom` and `TryInto` are the fallible versions of `From` and `Into`.

`TryFrom` 和 `TryInto` 是可能失败版本的 `From` 和 `Into`。

```rust

```
trait TryFrom<T> {
 type Error;
 fn try_from(value: T) -> Result<Self, Self::Error>;
}
```

```
trait TryInto<T> {
 type Error;
 fn try_into(self) -> Result<T, Self::Error>;
}
```
```

> Similarly to `Into` we cannot impl `TryInto` because its impl is provided by this generic blanket impl:

与 `Into` 相似地，我们不能手动实现 `TryInto`，因为它已经为一揽子泛型实现所提供。

```rust

```
impl<T, U> TryInto<U> for T
```

where

```
U: TryFrom<T>,
```

```
{
```

```
 type Error = U::Error;
```

```
 fn try_into(self) -> Result<U, U::Error> {
```

```
 U::try_from(self)
```

```
 }
```

```
}
```

```
...
```

> Let's say that in the context of our program it doesn't make sense for `Point`s to have `x` and `y` values that are less than `-1000` or greater than `1000`. This is how we'd rewrite our earlier `From` impls using `TryFrom` to signal to the users of our type that this conversion can now fail:

例如，我们的程序要求 `Point` 的 `x` 和 `y` 的值必须要处于 `-1000` 到 `1000` 之间，相较于 `From`，使用 `TryFrom` 可以告知上级调用者，某些转换可能失败了。

```rust

```
use std::convert::TryFrom;
```

```
use std::error;
```

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

```
struct Point {
```

```
    x: i32,
```

```
y: i32,  
}  
  
#[derive(Debug)]  
struct OutOfBounds;  
  
impl fmt::Display for OutOfBounds {  
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {  
        write!(f, "out of bounds")  
    }  
}  
  
impl error::Error for OutOfBounds {}  
  
// now fallible  
// TryFrom 的转换允许失败  
impl TryFrom<(i32, i32)> for Point {  
    type Error = OutOfBounds;  
  
    fn try_from((x, y): (i32, i32)) -> Result<Point, OutOfBounds> {  
        if x.abs() > 1000 || y.abs() > 1000 {  
            return Err(OutOfBounds);  
        }  
        Ok(Point { x, y })  
    }  
}  
  
// still infallible  
// From 的转换不允许失败  
impl From<Point> for (i32, i32) {  
    fn from(Point { x, y }: Point) -> Self {
```

```
(x, y)
}
}
```

```

> And here's the refactored `TryFrom<[TryInto<Point>; 3]>` impl for `Triangle`:

现在，我们对 `Triangle` 使用 `TryFrom<[TryInto<Point>; 3]>` 进行重构：

```
```rust
```

```
use std::convert::{TryFrom, TryInto};
```

```
use std::error;
```

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

```
struct Point {
```

```
    x: i32,
```

```
    y: i32,
```

```
}
```

```
#[derive(Debug)]
```

```
struct OutOfBounds;
```

```
impl fmt::Display for OutOfBounds {
```

```
fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
```

```
    write!(f, "out of bounds")
```

```
}
```

```
}
```

```
impl error::Error for OutOfBounds {}
```

```
impl TryFrom<(i32, i32)> for Point {
    type Error = OutOfBounds;
    fn try_from((x, y): (i32, i32)) -> Result<Self, Self::Error> {
        if x.abs() > 1000 || y.abs() > 1000 {
            return Err(OutOfBounds);
        }
        Ok(Point { x, y })
    }
}
```

```
struct Triangle {
    p1: Point,
    p2: Point,
    p3: Point,
}
```

```
impl<P> TryFrom<[P; 3]> for Triangle
where
    P: TryInto<Point>,
{
    type Error = P::Error;
    fn try_from([p1, p2, p3]: [P; 3]) -> Result<Self, Self::Error> {
        Ok(Triangle {
            p1: p1.try_into()?,
            p2: p2.try_into()?,
            p3: p3.try_into()?,
        })
    }
}
```

```
fn example() -> Result<Triangle, OutOfBounds> {
    let t: Triangle = [(0, 0), (1, 1), (2, 2)].try_into()?;
    Ok(t)
}
```

```

### ### FromStr

#### 预备知识

- [Self](#self)
- [Functions](#functions)
- [Associated Types](#associated-types)
- [Error](#error)
- [TryFrom & TryInto](#tryfrom--tryinto)

```
```rust
```

```
trait FromStr {  
    type Err;  
    fn from_str(s: &str) -> Result<Self, Self::Err>;  
}  

```

```

> `FromStr` types allow performing a fallible conversion from `&str` into `Self`. The idiomatic way to use `FromStr` is to call the `parse()` method on `&str`s:

实现 `FromStr` 特性的类型允许可失败地从 `&str` 转换至 `Self`。使用这一特性的理想方式是，调用 `&str` 实例的 `parse()` 方法：

```
```rust
```

```
use std::str::FromStr;
```

```
fn example<T: FromStr>(s: &'static str) {  
    // these are all equivalent  
  
    // 以下方法互相等价  
  
    let t: Result<T, _> = FromStr::from_str(s);  
  
    let t = T::from_str(s);  
  
    let t: Result<T, _> = s.parse();  
  
    let t = s.parse::<T>(); // most idiomatic  
  
    // 最理想的使用方式  
  
}
```

> Example impl for `Point`:

下例为 `Point` 实现了 `FromStr` 特性：

```
```rust  
use std::error;
use std::fmt;
use std::iter::Enumerate;
use std::num::ParseIntError;
use std::str::{Chars, FromStr};

#[derive(Debug, Eq, PartialEq)]
struct Point {
 x: i32,
 y: i32,
}

impl Point {
```

```
fn new(x: i32, y: i32) -> Self {
 Point { x, y }
}

#[derive(Debug, PartialEq)]
struct ParsePointError;

impl fmt::Display for ParsePointError {
 fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
 write!(f, "failed to parse point")
 }
}

impl From<ParseIntError> for ParsePointError {
 fn from(_e: ParseIntError) -> Self {
 ParsePointError
 }
}

impl error::Error for ParsePointError {}

impl FromStr for Point {
 type Err = ParsePointError;

 fn from_str(s: &str) -> Result<Self, Self::Err> {
 let is_num = |(i, c): (&(usize, char)| matches!(c, '0'..='9' | '-'));
 let isn't_num = |t: &(i, _)| !is_num(t);

 let get_num =

```

```
|char_idxs: &mut Enumerate<Chars<'_>>| -> Result<(usize, usize), ParsePointError> {
let (start, _) = char_idxs
.skip_while(isnt_num)
.next()
.ok_or(ParsePointError)?;
let (end, _) = char_idxs
.skip_while(is_num)
.next()
.ok_or(ParsePointError)?;
Ok((start, end))
};
```

```
let mut char_idxs = s.chars().enumerate();
let (x_start, x_end) = get_num(&mut char_idxs)?;
let (y_start, y_end) = get_num(&mut char_idxs)?;
```

```
let x = s[x_start..x_end].parse::<i32>()?;
let y = s[y_start..y_end].parse::<i32>()?;

```

```
Ok(Point { x, y })
}
}
```

```
#[test] //
fn pos_x_y() {
let p = "(4, 5)".parse::<Point>();
assert_eq!(p, Ok(Point::new(4, 5)));
}
```

```
#[test] //
```

```

fn neg_x_y() {
let p = "(-6, -2)".parse::<Point>();
assert_eq!(p, Ok(Point::new(-6, -2)));
}

#[test] //

fn not_a_point() {

let p = "not a point".parse::<Point>();
assert_eq!(p, Err(ParsePointError));
}

```

```

> `FromStr` 与 `TryFrom<&str>` 具有相同的函数签名。先实现哪个特性无关紧要，因为我们可以利用先实现的特性实现后实现的特性。例如，我们假定 `Point` 类型已经实现了 `FromStr` 特性，再来实现 `TryFrom<&str>` 特性：

```

```rust
impl TryFrom<&str> for Point {
 type Error = <Point as FromStr>::Err;

 fn try_from(s: &str) -> Result<Point, Self::Error> {
 <Point as FromStr>::from_str(s)
 }
}

```
#### AsRef & AsMut
```

预备知识

- [Self](#self)
- [Methods](#methods)
- [Sized](#sized)
- [Generic Parameters](#generic-parameters)
- [Sized](#sized)
- [Deref & DerefMut](#deref--derefmut)

```rust

```
trait AsRef<T: ?Sized> {
 fn as_ref(&self) -> &T;
}
```

```
trait AsMut<T: ?Sized> {
 fn as_mut(&mut self) -> &mut T;
}
```

```

> `AsRef` is for cheap reference to reference conversions. However, one of the most common ways it's used is to make functions generic over whether they take ownership or not:

`AsRef` 特性的存在很大程度上便捷了引用转换，其最常见的使用是为函数的引用类型的参数的传入提供方便：

```rust

```
// accepts:
// - &str
// - &String

fn takes_str(s: &str) {
 // use &str
}
```

```
// accepts:
```

```

// - &str
// - &String
// - String

fn takes_asref_str<S: AsRef<str>>(s: S) {
 let s: &str = s.as_ref();
 // use &str
}

fn example(slice: &str, borrow: &String, owned: String) {
 takes_str(slice);
 takes_str(borrow);
 takes_str(owned); //
 takes_asref_str(slice);
 takes_asref_str(borrow);
 takes_asref_str(owned); //
}
```

```

> The other most common use-case is returning a reference to inner private data wrapped by a type which protects some invariant. A good example from the standard library is `String` which is just a wrapper around `Vec<u8>`:

另外一个常见的使用是，返回一个包装类型的内部私有数据的引用（该类型用于保证内部私有数据的不变性）。标准库中的 `String` 就是对 `Vec<u8>` 的这样一种包装：

```
```rust
```

```
struct String {
 vec: Vec<u8>,
}
```

```
```
```

> This inner `Vec` cannot be made public because if it was people could mutate any byte and break the `String`'s valid UTF-8 encoding. However, it's safe to expose an immutable read-only reference to the inner byte array, hence this impl:

之所以不公开内部的 `Vec` 数据，是因为一旦允许用户随意修改内部数据，就有可能破坏 `String` 有效的 UTF-8 编码。但是，对外开放一个只读的字节数组的引用是安全的，所以有如下实现：

```
```rust
```

```
impl AsRef<[u8]> for String;
```

```
...
```

> Generally, it often only makes sense to impl `AsRef` for a type if it wraps some other type to either provide additional functionality around the inner type or protect some invariant on the inner type.

通常来讲我们不对类型实现 `AsRef` 特性，除非该类型包装了其它类型以提供额外的功能，或是对内部类型提供了不变性的保护。

> Let's examine a example of bad `AsRef` impls:

以下是实现 `AsRef` 特性的一个反例：

```
```rust
```

```
struct User {  
    name: String,  
    age: u32,  
}
```

```
impl AsRef<String> for User {  
    fn as_ref(&self) -> &String {  
        &self.name  
    }  
}
```

```
impl AsRef<u32> for User {
fn as_ref(&self) -> &u32 {
    &self.age
}
}
```
}
```

> This works and kinda makes sense at first, but quickly falls apart if we add more members to `User`:

乍看起来这似乎有几分道理，但是当我们对 `User` 类型添加新的成员时，缺点就暴露出来了：

```
```rust
```

```
struct User {
    name: String,
    email: String,
    age: u32,
    height: u32,
}
```

```
impl AsRef<String> for User {
fn as_ref(&self) -> &String {
    // uh, do we return name or email here?
    // 既然我们要返回一个字符串引用，那具体应该返回什么呢？
    // name 和 email 都是字符串，如何选择呢？
    // 出于返回类型的限制，似乎我们也难以返回一个混合的字符串。
}
}
```

```
impl AsRef<u32> for User {
fn as_ref(&self) -> &u32 {
```

```
// uh, do we return age or height here?  
// 如上同理  
}  
}  
...  
  
...
```

> A `User` is composed of `String`s and `u32`s but it's not really the same thing as a `String` or a `u32`. Even if we had much more specific types:

`User` 类型由多个 `String` 和 `u32` 类型的成员所组成，但我们也不能说 `User` 是 `String` 或 `u32` 吧？即便由更加具体的类型来构造也不行：

```rust

```
struct User {
 name: Name,
 email: Email,
 age: Age,
 height: Height,
}
```

> It wouldn't make much sense to impl `AsRef` for any of those because `AsRef` is for cheap reference to reference conversions between semantically equivalent things, and `Name`, `Email`, `Age`, and `Height` by themselves are not the same thing as a `User`.

对于 `User` 这样的类型来讲，实现 `AsRef` 特性并没有什么太多意义。因为 `AsRef` 的存在仅是为了做一种最简单的引用转换，这种转换最好存在于语义上相类似的事物之间。`Name`、`Email`、`Age` 和 `Height` 其本身和 `User` 就不是一回事，在逻辑上谈不上转换。

> A good example where we would impl `AsRef` would be if we introduced a new type `Moderator` that just wrapped a `User` and added some moderation specific privileges:

下例展示了 `AsRef` 特性的正确用法，我们实现了一个新的类型 `Moderator`，它仅仅是包装了 `User` 类型，并添加了对其权限的一些控制：

```
```rust
struct User {
    name: String,
    age: u32,
}

// unfortunately the standard library cannot provide
// a generic blanket impl to save us from this boilerplate
// 不幸的是，标准库并没有提供相应的一揽子泛型实现，我们不得不手动实现
impl AsRef<User> for User {
    fn as_ref(&self) -> &User {
        self
    }
}

enum Privilege {
    BanUsers,
    EditPosts,
    DeletePosts,
}

// although Moderators have some special
// privileges they are still regular Users
// and should be able to do all the same stuff
// 尽管主持人类具有一些特殊的权限，
// 但其仍然是普通的用户
// 所有用户类能做到的主持人类也应能做到

struct Moderator {
    user: User,
    privileges: Vec<Privilege>
}
```

```
}
```

```
impl AsRef<Moderator> for Moderator {
```

```
fn as_ref(&self) -> &Moderator {
```

```
    self
```

```
}
```

```
}
```

```
impl AsRef<User> for Moderator {
```

```
fn as_ref(&self) -> &User {
```

```
    &self.user
```

```
}
```

```
}
```

```
// this should be callable with Users
```

```
// and Moderators (who are also Users)
```

```
// 这个函数的参数可以是 User 也可以是 Moderator
```

```
// ( Moderator 也是 User )
```

```
fn create_post<U: AsRef<User>>(u: U) {
```

```
let user = u.as_ref();
```

```
// etc
```

```
}
```

```
fn example(user: User, moderator: Moderator) {
```

```
    create_post(&user);
```

```
    create_post(&moderator); //
```

```
}
```

```
...
```

> This works because `Moderator`'s are just `User`'s. Here's the example from the `Deref` section except using `AsRef` instead:

之所以可以这样做，是因为 `Moderator` 就是 `User`。下例是将 `Deref` 一节中的例子使用 `AsRef` 做出替代：

```
```rust
```

```
use std::convert::AsRef;
```

```
struct Human {
```

```
 health_points: u32,
```

```
}
```

```
impl AsRef<Human> for Human {
```

```
 fn as_ref(&self) -> &Human {
```

```
 self
```

```
 }
```

```
}
```

```
enum Weapon {
```

```
 Spear,
```

```
 Axe,
```

```
 Sword,
```

```
}
```

```
// a Soldier is just a Human with a Weapon
```

```
// 士兵是手持武器的人类
```

```
struct Soldier {
```

```
 human: Human,
```

```
 weapon: Weapon,
```

```
}
```

```
impl AsRef<Soldier> for Soldier {
```

```
fn as_ref(&self) -> &Soldier {
 self
}

impl AsRef<Human> for Soldier {
 fn as_ref(&self) -> &Human {
 &self.human
 }
}

enum Mount {
 Horse,
 Donkey,
 Cow,
}

// a Knight is just a Soldier with a Mount
// 骑士是跨骑坐骑的士兵
struct Knight {
 soldier: Soldier,
 mount: Mount,
}

impl AsRef<Knight> for Knight {
 fn as_ref(&self) -> &Knight {
 self
 }
}
```

```
impl AsRef<Soldier> for Knight {
fn as_ref(&self) -> &Soldier {
&self.soldier
}
}
```

```
impl AsRef<Human> for Knight {
fn as_ref(&self) -> &Human {
&self.soldier.human
}
}
```

```
enum Spell {
MagicMissile,
FireBolt,
ThornWhip,
}
```

```
// a Mage is just a Human who can cast Spells
```

```
// 法师是口诵咒语的人类
struct Mage {
human: Human,
spells: Vec<Spell>,
}
```

```
impl AsRef<Mage> for Mage {
fn as_ref(&self) -> &Mage {
self
}
}
```

```
impl AsRef<Human> for Mage {
fn as_ref(&self) -> &Human {
 &self.human
}
}
```

```
enum Staff {
 Wooden,
 Metallic,
 Plastic,
}
```

```
// a Wizard is just a Mage with a Staff
// 巫师是腰别法杖的法师
struct Wizard {
 mage: Mage,
 staff: Staff,
}
```

```
impl AsRef<Wizard> for Wizard {
fn as_ref(&self) -> &Wizard {
 self
}
}
```

```
impl AsRef<Mage> for Wizard {
fn as_ref(&self) -> &Mage {
 &self.mage
}
}
```

```
impl AsRef<Human> for Wizard {
 fn as_ref(&self) -> &Human {
 &self.mage.human
 }
}

fn borrows_human<H: AsRef<Human>>(human: H) {}
fn borrows_soldier<S: AsRef<Soldier>>(soldier: S) {}
fn borrows_knight<K: AsRef<Knight>>(knight: K) {}
fn borrows_mage<M: AsRef<Mage>>(mage: M) {}
fn borrows_wizard<W: AsRef<Wizard>>(wizard: W) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
 // all types can be used as Humans
 borrows_human(&human);
 borrows_human(&soldier);
 borrows_human(&knight);
 borrows_human(&mage);
 borrows_human(&wizard);
 // Knights can be used as Soldiers
 borrows_soldier(&soldier);
 borrows_soldier(&knight);
 // Wizards can be used as Mages
 borrows_mage(&mage);
 borrows_mage(&wizard);
 // Knights & Wizards passed as themselves
 borrows_knight(&knight);
 borrows_wizard(&wizard);
}
```

```

> `Deref` didn't work in the prior version of the example above because deref coercion is an implicit conversion between types which leaves room for people to mistakenly formulate the wrong ideas and expectations for how it will behave. `AsRef` works above because it makes the conversion between types explicit and there's no room leftover to develop any wrong ideas or expectations.

之所以 `Deref` 在上例之前的版本中不可使用，是因为自动解引用是一种隐式的转换，这就为程序员错误地使用留下了巨大的空间。

而 `AsRef` 在上例中可以使用，是因为其实现的转换是显式的，这样很大程度上就消除了犯错误的空间。

Borrow & BorrowMut

预备知识

- [Self](#self)
- [Methods](#methods)
- [Generic Parameters](#generic-parameters)
- [Subtraits & Supertraits](#subtraits--supertraits)
- [Sized](#sized)
- [AsRef & AsMut](#asref--asmut)
- [PartialEq & Eq](#partialeq--eq)
- [Hash](#hash)
- [PartialOrd & Ord](#partialord--ord)

```rust

```
trait Borrow<Borrowed>
```

where

```
Borrowed: ?Sized,
```

```
{
```

```
fn borrow(&self) -> &Borrowed;
```

```
}
```

```
trait BorrowMut<Borrowed>: Borrow<Borrowed>
```

where

```
Borrowed: ?Sized,
```

```
{
```

```
fn borrow_mut(&mut self) -> &mut Borrowed;
```

```
}
```

```
...
```

> These traits were invented to solve the very specific problem of looking up `String` keys in `HashSet`s, `HashMap`s, `BTreeSet`s, and `BTreeMap`s using `&str` values.

这类特性存在的意义旨在于解决特定领域的问题，例如在 `Hashset`，`HashMap`，`BTreeSet`，`BtreeMap` 中使用 `&str` 查询 `String` 类型的键。

> We can view `Borrow<T>` and `BorrowMut<T>` as stricter versions of `AsRef<T>` and `AsMut<T>`, where the returned reference `&T` has equivalent `Eq`, `Hash`, and `Ord` impls to `Self`. This is more easily explained with a commented example:

我们可以将 `Borrow<T>` 和 `BorrowMut<T>` 视作 `AsRef<T>` 和 `AsMut<T>` 的严格版本，其返回的引用 `&T` 具有与 `Self` 相同的 `Eq`，`Hash` 和 `Ord` 的实现。这一点在下例的注释中得到很好的解释：

```
```rust
```

```
use std::borrow::Borrow;
use std::hash::Hasher;
use std::collections::hash_map::DefaultHasher;
use std::hash::Hash;

fn get_hash<T: Hash>(t: T) -> u64 {
    let mut hasher = DefaultHasher::new();
    t.hash(&mut hasher);
    hasher.finish()
```

```
}
```

```
fn asref_example<Owned, Ref>(owned1: Owned, owned2: Owned)
```

```
where
```

```
Owned: Eq + Ord + Hash + AsRef<Ref>,
```

```
Ref: Eq + Ord + Hash
```

```
{
```

```
let ref1: &Ref = owned1.as_ref();
```

```
let ref2: &Ref = owned2.as_ref();
```

```
// refs aren't required to be equal if owned types are equal
```

```
// 值相等，不意味着其引用一定相等
```

```
assert_eq!(owned1 == owned2, ref1 == ref2); //
```

```
let owned1_hash = get_hash(&owned1);
```

```
let owned2_hash = get_hash(&owned2);
```

```
let ref1_hash = get_hash(&ref1);
```

```
let ref2_hash = get_hash(&ref2);
```

```
// ref hashes aren't required to be equal if owned type hashes are equal
```

```
// 值的哈希值相等，其引用不一定相等
```

```
assert_eq!(owned1_hash == owned2_hash, ref1_hash == ref2_hash); //
```

```
// ref comparisons aren't required to match owned type comparisons
```

```
// 值的比较，与其应用的比较没有必然联系
```

```
assert_eq!(owned1.cmp(&owned2), ref1.cmp(&ref2)); //
```

```
}
```

```
fn borrow_example<Owned, Borrowed>(owned1: Owned, owned2: Owned)
```

```
where
```

```

Owned: Eq + Ord + Hash + Borrow<Owned>,
Borrowed: Eq + Ord + Hash

{

let borrow1: &Owned = owned1.borrow();

let borrow2: &Owned = owned2.borrow();

// borrows are required to be equal if owned types are equal
// 值相等，借用值也必须相等

assert_eq!(owned1 == owned2, borrow1 == borrow2); //


let owned1_hash = get_hash(&owned1);

let owned2_hash = get_hash(&owned2);

let borrow1_hash = get_hash(&borrow1);

let borrow2_hash = get_hash(&borrow2);

// borrow hashes are required to be equal if owned type hashes are equal
// 值的哈希值相等，借用值的哈希值也必须相等

assert_eq!(owned1_hash == owned2_hash, borrow1_hash == borrow2_hash); //


// borrow comparisons are required to match owned type comparisons
// 值的比较，与借用值的比较必须步调一致

assert_eq!(owned1.cmp(&owned2), borrow1.cmp(&borrow2)); //

}

```

```

> It's good to be aware of these traits and understand why they exist since it helps demystify some of the methods on `HashSet`, `HashMap`, `BTreeSet`, and `BTreeMap` but it's very rare that we would ever need to impl these traits for any of our types because it's very rare that we would ever need create a pair of types where one is the "borrowed" version of the other in the first place. If we have some `T` then `&T` will get the job done 99.99% of the time, and `T: Borrow<T>` is already implemented for all `T` because of a generic blanket impl, so we don't need to manually impl it and we don't need to create some `U` such that `T: Borrow<U>`.

理解这类特性存在的意义，有助于我们揭开 `HashSet`，`HashMap`，`BTreeSet` 和 `BTreeMap` 中某些方法的实现的神秘面纱。但是在实际应用中，几乎没有什么地方需要我们去实现这样的特性，因为再难找到一个需要我们对一个值再创造一个“借用”版本的类型的场景了。对于某种类型 `T`，`&T` 就能解决 99.9% 的问题了，且 `T: Borrow<T>` 已经被一揽子泛型实现对 `T` 实现了，所以我们无需手动实现它，也无需去实现某种的对 `U` 有 `T: Borrow<U>` 了。

### ### ToOwned

#### 预备知识

- [Self](#self)
- [Methods](#methods)
- [Default Impls](#default-impls)
- [Clone](#clone)
- [Borrow & BorrowMut](#borrow--borrowmut)

```rust

```
trait ToOwned {  
    type Owned: Borrow<Self>;  
    fn to_owned(&self) -> Self::Owned;  
  
    // provided default impls  
    // 提供默认实现  
    fn clone_into(&self, target: &mut Self::Owned);  
}
```

> `ToOwned` is a more generic version of `Clone`. `Clone` allows us to take a `&T` and turn it into an `T` but `ToOwned` allows us to take a `&Borrowed` and turn it into a `Owned` where `Owned: Borrow<Borrowed>`.

`ToOwned` 特性是 `Clone` 特性的泛型版本。`Clone` 特性允许我们由 `&T` 类型得到 `T` 类型，而 `ToOwned` 特性允许我们由 `&Borrow` 类型得到 `Owned` 类型，其中 `Owned: Borrow<Borrowed>`。

> In other words, we can't "clone" a `&str` into a `String`, or a `&Path` into a `PathBuf`, or an `&OsStr` into an `OsString`, since the `clone` method signature doesn't support this kind of cross-type cloning, and that's what `ToOwned` was made for.

换句话讲，我们不能将 `&str` 克隆为 `String`，将 `&Path` 克隆为 `PathBuf` 或将 `&OsStr` 克隆为 `OsString`。鉴于 `clone` 方法的签名不支持这样跨类型的克隆，这就是 `ToOwned` 特性存在的意义。

> For similar reasons as `Borrow` and `BorrowMut`, it's good to be aware of this trait and understand why it exists but it's very rare we'll ever need to impl it for any of our types.

与 `Borrow` 和 `BorrowMut` 相同地，理解此类特性存在的意义对我们或有帮助，但是鲜少需要我们手动为自己的类实现该特性。

迭代特性 Iteration Traits

Iterator

预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Default Impls](#default-impls)

```rust

```
trait Iterator {
 type Item;

 fn next(&mut self) -> Option<Self::Item>;
```

```
// provided default impls
```

```
// 提供默认实现
fn size_hint(&self) -> (usize, Option<usize>);

fn count(self) -> usize;

fn last(self) -> Option<Self::Item>;

fn advance_by(&mut self, n: usize) -> Result<(), usize>;

fn nth(&mut self, n: usize) -> Option<Self::Item>;

fn step_by(self, step: usize) -> StepBy<Self>;

fn chain<U>(
 self,
 other: U
) -> Chain<Self, <U as Intolterator>::Intolter>

where
 U: Intolterator<Item = Self::Item>;
```

fn zip<U>(self, other: U) -> Zip<Self, <U as Intolterator>::Intolter>

where

```
U: Intolterator;
```

fn map<B, F>(self, f: F) -> Map<Self, F>

where

```
F: FnMut(Self::Item) -> B;
```

fn for\_each<F>(self, f: F)

where

```
F: FnMut(Self::Item);
```

fn filter<P>(self, predicate: P) -> Filter<Self, P>

where

```
P: FnMut(&Self::Item) -> bool;
```

fn filter\_map<B, F>(self, f: F) -> FilterMap<Self, F>

where

```
F: FnMut(Self::Item) -> Option;
```

fn enumerate(self) -> Enumerate<Self>;

fn peekable(self) -> Peekable<Self>;

```
fn skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>
```

where

```
P: FnMut(&Self::Item) -> bool;
```

```
fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
```

where

```
P: FnMut(&Self::Item) -> bool;
```

```
fn map_while<B, P>(self, predicate: P) -> MapWhile<Self, P>
```

where

```
P: FnMut(Self::Item) -> Option;
```

```
fn skip(self, n: usize) -> Skip<Self>;
```

```
fn take(self, n: usize) -> Take<Self>;
```

```
fn scan<St, B, F>(self, initial_state: St, f: F) -> Scan<Self, St, F>
```

where

```
F: FnMut(&mut St, Self::Item) -> Option;
```

```
fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>
```

where

```
F: FnMut(Self::Item) -> U,
```

```
U: Intolterator;
```

```
fn flatten(self) -> Flatten<Self>
```

where

```
Self::Item: Intolterator;
```

```
fn fuse(self) -> Fuse<Self>;
```

```
fn inspect<F>(self, f: F) -> Inspect<Self, F>
```

where

```
F: FnMut(&Self::Item);
```

```
fn by_ref(&mut self) -> &mut Self;
```

```
fn collect(self) -> B
```

where

```
B: Fromlterator<Self::Item>;
```

```
fn partition<B, F>(self, f: F) -> (B, B)
```

where

F: FnMut(&Self::Item) -> bool,

B: Default + Extend<Self::Item>;

fn partition\_in\_place<'a, T, P>(self, predicate: P) -> usize

where

Self: DoubleEndedIterator<Item = &'a mut T>,

T: 'a,

P: FnMut(&T) -> bool;

fn is\_partitioned<P>(self, predicate: P) -> bool

where

P: FnMut(Self::Item) -> bool;

fn try\_fold<B, F, R>(&mut self, init: B, f: F) -> R

where

F: FnMut(B, Self::Item) -> R,

R: Try<Ok = B>;

fn try\_for\_each<F, R>(&mut self, f: F) -> R

where

F: FnMut(Self::Item) -> R,

R: Try<Ok = ()>;

fn fold<B, F>(self, init: B, f: F) -> B

where

F: FnMut(B, Self::Item) -> B;

fn fold\_first<F>(self, f: F) -> Option<Self::Item>

where

F: FnMut(Self::Item, Self::Item) -> Self::Item;

fn all<F>(&mut self, f: F) -> bool

where

F: FnMut(Self::Item) -> bool;

fn any<F>(&mut self, f: F) -> bool

where

F: FnMut(Self::Item) -> bool;  
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>

where

P: FnMut(&Self::Item) -> bool;  
fn find\_map<B, F>(&mut self, f: F) -> Option<B>

where

F: FnMut(Self::Item) -> Option<B>;  
fn try\_find<F, R>(  
&mut self,  
f: F  
) -> Result<Option<Self::Item>, <R as Try>::Error>

where

F: FnMut(&Self::Item) -> R,  
R: Try<Ok = bool>;  
fn position<P>(&mut self, predicate: P) -> Option<usize>

where

P: FnMut(Self::Item) -> bool;  
fn rposition<P>(&mut self, predicate: P) -> Option<usize>

where

Self: ExactSizedIterator + DoubleEndedIterator,

P: FnMut(Self::Item) -> bool;  
fn max(self) -> Option<Self::Item>

where

Self::Item: Ord;  
fn min(self) -> Option<Self::Item>

where

Self::Item: Ord;  
fn max\_by\_key<B, F>(self, f: F) -> Option<Self::Item>

where

F: FnMut(&Self::Item) -> B,

B: Ord;  
fn max\_by<F>(self, compare: F) -> Option<Self::Item>

where

F: FnMut(&Self::Item, &Self::Item) -> Ordering;  
fn min\_by\_key<B, F>(self, f: F) -> Option<Self::Item>

where

F: FnMut(&Self::Item) -> B,

B: Ord;

fn min\_by<F>(self, compare: F) -> Option<Self::Item>  
where

F: FnMut(&Self::Item, &Self::Item) -> Ordering;

fn rev(self) -> Rev<Self>

where

Self: DoubleEndedIterator;

fn unzip<A, B, FromA, FromB>(self) -> (FromA, FromB)

where

Self: Iterator<Item = (A, B)>,

FromA: Default + Extend<A>,

FromB: Default + Extend<B>;

fn copied<'a, T>(self) -> Copied<Self>

where

Self: Iterator<Item = &'a T>,

T: 'a + Copy;

fn cloned<'a, T>(self) -> Cloned<Self>

where

Self: Iterator<Item = &'a T>,

T: 'a + Clone;

fn cycle(self) -> Cycle<Self>

where

Self: Clone;

```
fn sum<S>(self) -> S
```

where

```
S: Sum<Self::Item>;
```

```
fn product<P>(self) -> P
```

where

```
P: Product<Self::Item>;
```

```
fn cmp<I>(self, other: I) -> Ordering
```

where

```
I: Intolterator<Item = Self::Item>,
```

```
Self::Item: Ord;
```

```
fn cmp_by<I, F>(self, other: I, cmp: F) -> Ordering
```

where

```
F: FnMut(Self::Item, <I as Intolterator>::Item) -> Ordering,
```

```
I: Intolterator;
```

```
fn partial_cmp<I>(self, other: I) -> Option<Ordering>
```

where

```
I: Intolterator,
```

```
Self::Item: PartialOrd<<I as Intolterator>::Item>;
```

```
fn partial_cmp_by<I, F>(
```

```
self,
```

```
other: I,
```

```
partial_cmp: F
```

```
) -> Option<Ordering>
```

where

```
F: FnMut(Self::Item, <I as Intolterator>::Item) -> Option<Ordering>,
```

```
I: Intolterator;
```

```
fn eq<I>(self, other: I) -> bool
```

where

```
I: Intolterator,
```

```
Self::Item: PartialEq<<I as Intolterator>::Item>;
```

```
fn eq_by<I, F>(self, other: I, eq: F) -> bool
where
F: FnMut(Self::Item, <I as Intolterator>::Item) -> bool,
I: Intolterator;

fn ne<I>(self, other: I) -> bool
where
I: Intolterator,
Self::Item: PartialEq<<I as Intolterator>::Item>;

fn lt<I>(self, other: I) -> bool
where
I: Intolterator,
Self::Item: PartialOrd<<I as Intolterator>::Item>;

fn le<I>(self, other: I) -> bool
where
I: Intolterator,
Self::Item: PartialOrd<<I as Intolterator>::Item>;

fn gt<I>(self, other: I) -> bool
where
I: Intolterator,
Self::Item: PartialOrd<<I as Intolterator>::Item>;

fn ge<I>(self, other: I) -> bool
where
I: Intolterator,
Self::Item: PartialOrd<<I as Intolterator>::Item>;

fn is_sorted(self) -> bool
where
Self::Item: PartialOrd<Self::Item>;

fn is_sorted_by<F>(self, compare: F) -> bool
where
F: FnMut(&Self::Item, &Self::Item) -> Option<Ordering>;
```

```
fn is_sorted_by_key<F, K>(self, f: F) -> bool
where
 F: FnMut(Self::Item) -> K,
 K: PartialOrd<K>;
}
```

```

> `Iterator<Item = T>` types can be iterated and will produce `T` types. There's no `IteratorMut` trait. Each `Iterator` impl can specify whether it returns immutable references, mutable references, or owned values via the `Item` associated type.

实现 `Iterator<Item = T>` 的类型可以迭代产生 `T` 类型。注意：并不存在 `IteratorMut` 类型，因为可以通过在实现 `Iterator` 特性时指定 `Item` 关联类型，来选择其返回的是不可变引用、可变引用还是自有值。

| | |
|--|------|
| `Vec<T>` 方法 | 返回类型 |
| ----- ----- | |
| `.`iter()` `Iterator<Item = &T>` | |
| `.`iter_mut()` `Iterator<Item = &mut T>` | |
| `.`into_iter()` `Iterator<Item = T>` | |

> Something that is not immediately obvious to beginner Rustaceans but that intermediate Rustaceans take for granted is that most types are not their own iterators. If a type is iterable we almost always impl some custom iterator type which iterates over it rather than trying to make it iterate over itself:

对于 Rust 的初学者而言可能有些费解，但是对于中级学习者而言则是顺理成章的一件事是——绝大多数类型并不是自己的迭代器。这意味着，如果某种类型是可迭代的，那么应当实现某种额外的迭代器类型去迭代它，而不是让它自己迭代自己。

```
```rust
struct MyType {
 items: Vec<String>
}
```

```
impl MyType {
```

```
fn iter(&self) -> impl Iterator<Item = &String> {
 MyTypeIterator {
 index: 0,
 items: &self.items
 }
}
```

```
struct MyTypeIterator<'a> {
 index: usize,
 items: &'a Vec<String>
}
```

```
impl<'a> Iterator for MyTypeIterator<'a> {
 type Item = &'a String;

 fn next(&mut self) -> Option<Self::Item> {
 if self.index >= self.items.len() {
 None
 } else {
 let item = &self.items[self.index];
 self.index += 1;
 Some(item)
 }
 }
}

```
    ...

```

> For the sake of teaching the above example shows how to impl an `Iterator` from scratch but the idiomatic solution in this situation would be to just defer to `Vec`'s `iter` method:

出于教学的原因，我们在上例中从头手动实现了一个迭代器。而在这种情况下，最理想的做法是直接调用 `Vec` 的 `iter` 方法。

```
```rust
struct MyType {
 items: Vec<String>
}

impl MyType {
 fn iter(&self) -> impl Iterator<Item = &String> {
 self.items.iter()
 }
}
```

```

> Also this is a good generic blanket impl to be aware of:

另外，最好了解这个一揽子泛型实现：

```
```rust
impl<I: Iterator + ?Sized> Iterator for &mut I;
```

```

> It says that any mutable reference to an iterator is also an iterator. This is useful to know because it allows us to use iterator methods with `self` receivers as if they had `&mut self` receivers.

任何迭代器的可变引用也是一个迭代器。了解这样的性质有助于我们理解，为什么可以将迭代器的某些参数为 `self` 的方法当作具有 `&mut self` 参数的方法来使用。

> As an example, imagine we have a function which processes an iterator of more than three items, but the first step of the function is to take out the first three items of the iterator and process them separately before iterating over the remaining items, here's how a beginner may attempt to write this function:

举个例子，想象我们有这样一个函数，它处理一个具有三个以上值的迭代器，这个函数首先要取得该迭代器的前三个值并分别地处理他们，然后再依次迭代剩余的值。初学者可能会这样实现该函数：

```
```rust
fn example<I: Iterator<Item = i32>>(mut iter: I) {
 let first3: Vec<i32> = iter.take(3).collect();

 for item in iter { // iter consumed in line above
 // iter 在上一行就已经被消耗掉了

 // process remaining items

 // 处理剩余的值
 }
}

```
```

```

> Well that's annoying. The `take` method has a `self` receiver so it seems like we cannot call it without consuming the whole iterator! Here's what a naive refactor of the above code might look like:

糟糕，`take` 方法具有 `self` 参数，这意味着我们不能在不消耗掉整个迭代器的前提下调用该方法。以下可能是一个初学者的改进：

```
```rust
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = vec![
        iter.next().unwrap(),
        iter.next().unwrap(),
        iter.next().unwrap(),
    ];

    for item in iter { //
        // process remaining items

        // 处理剩余的值
    }
}

```
```

```

> Which is okay. However, the idiomatic refactor is actually:

这是可行的，但是理想的改进方式莫过于：

```
```rust
fn example<I: Iterator<Item = i32>>(mut iter: I) {
 let first3: Vec<i32> = iter.by_ref().take(3).collect();

 for item in iter { //
 // process remaining items
 // 处理剩余的值
 }
}

```
```

```

> Not very easy to discover. But anyway, now we know.

这真是一个很隐蔽的方法，但是被我们抓到了。

> Also, there are no rules or conventions on what can or cannot be an iterator. If the type implements `Iterator` then it's an iterator. Some creative examples from the standard library:

同样，对于什么可以是迭代器，什么不可以是，并无一定之规。实现了 `Iterator` 特性的就是迭代器。而在标准库中，确有一些具有创造性的用例：

```
```rust
use std::sync::mpsc::channel;

use std::thread;

fn paths_can_be_iterated(path: &Path) {
    for part in path {
        // iterate over parts of a path
    }
}
```
```

```

```

// 迭代 path 的不同部分
}

}

fn receivers_can_be_iterated() {
let (send, recv) = channel();

thread::spawn(move || {
    send.send(1).unwrap();
    send.send(2).unwrap();
    send.send(3).unwrap();
});

for received in recv {
    // iterate over received values
    // 迭代接收到的值
}
}

```
 ...
}

Intolterator
```

## 预备知识

- [Self](#self)
- [Methods](#methods)
- [Associated Types](#associated-types)
- [Iterator](#iterator)

```
```rust
trait Intolterator

where

<Self::Intolter as Iterator>::Item == Self::Item,

{
    type Item;

    type Intolter: Iterator;

    fn into_iter(self) -> Self::Intolter;
}
```

> `Intolterator` types can be converted into iterators, hence the name. The `into_iter` method is called on a type when it's used within a `for-in` loop:

闻弦歌而知雅意，实现 `Intolterator` 特性的类型可以被转换为迭代器。当用于 `for-in` 循环时，将自动调用该类型的 `into_iter` 方法。

```
```rust
// vec = Vec<T>

for v in vec {} // v = T

// above line desugared

// 以上代码等价于

for v in vec.into_iter() {}

```

```

> Not only does `Vec` impl `Intolterator` but so does `&Vec` and `&mut Vec` if we'd like to iterate over immutable or mutable references instead of owned values, respectively.

不仅 `Vec` 实现了 `Intolterator` 特性，`&Vec` 与 `&mut Vec` 同样如此。因此我们可以相应的对可变与不可变的引用，以及自有值进行迭代。

```
```rust
```

```
// vec = Vec<T>
for v in &vec {} // v = &T

// above example desugared
// 以上代码等价于

for v in (&vec).into_iter() {}
```

```
// vec = Vec<T>
for v in &mut vec {} // v = &mut T

// above example desugared
// 以上代码等价于

for v in (&mut vec).into_iter() {}
```

### ```rust

#### 预备知识

- [Self](#self)
- [Functions](#functions)
- [Generic Parameters](#generic-parameters)
- [Iterator](#iterator)
- [IntoIterator](#intoiterator)

```rust

```
trait FromIterator<A> {
    fn from_iter<T>(iter: T) -> Self
}
```

where

```
T: IntoIterator<Item = A>;  
}  
...  
...
```

> `FromIterator` types can be created from an iterator, hence the name. `FromIterator` is most commonly and idiomatically used by calling the `collect` method on `Iterator`:

顾叶落而晓秋至，实现 `FromIterator` 特性的类型可以由迭代器而构造。`FromIterator` 特性最常见和最理想的使用方法是调用 `Iterator` 的 `collect` 方法：

```
```rust  
fn collect(self) -> B
where
 B: FromIterator<Self::Item>;
...
...
```

> Example of collecting an `Iterator<Item = char>` into a `String`:

下例展示了如何将 `Iterator<Item = char>` 迭代器的值收集为 `String`：

```
```rust  
fn filter_letters(string: &str) -> String {  
    string.chars().filter(|c| c.is_alphanumeric()).collect()  
}  
...  
...
```

> All the collections in the standard library impl `IntoIterator` and `FromIterator` so that makes it easier to convert between them:

标准库中的全部集合类型都实现了 `IntoIterator` 和 `FromIterator` 特性，所以在它们之间进行转换是很方便的：

```
```rust
```

```
use std::collections::{BTreeSet, HashMap, HashSet, LinkedList};

// String -> HashSet<char>
fn unique_chars(string: &str) -> HashSet<char> {
 string.chars().collect()
}

// Vec<T> -> BTreeSet<T>
fn ordered_unique_items<T: Ord>(vec: Vec<T>) -> BTreeSet<T> {
 vec.into_iter().collect()
}

// HashMap<K, V> -> LinkedList<(K, V)>
fn entry_list<K, V>(map: HashMap<K, V>) -> LinkedList<(K, V)> {
 map.into_iter().collect()
}

// and countless more possible examples
// 还有数不胜数的例子
```
## 输入输出特性 I/O Traits
```

```
### Read & Write
```

预备知识

- [Self](#self)
- [Methods](#methods)
- [Scope](#scope)
- [Generic Blanket Impls](#generic-blanket-impls)

```rust

```
trait Read {
 fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
 // provided default impls
 // 提供默认实现
 fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) -> Result<usize>;
 fn is_read_vectored(&self) -> bool;
 unsafe fn initializer(&self) -> Initializer;
 fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize>;
 fn read_to_string(&mut self, buf: &mut String) -> Result<usize>;
 fn read_exact(&mut self, buf: &mut [u8]) -> Result<()>;
 fn by_ref(&mut self) -> &mut Self
}

where
Self: Sized;

fn bytes(self) -> Bytes<Self>
where
Self: Sized;

fn chain<R: Read>(self, next: R) -> Chain<Self, R>
where
Self: Sized;

fn take(self, limit: u64) -> Take<Self>
where
Self: Sized;
}
```

```

trait Write {

 fn write(&mut self, buf: &[u8]) -> Result<usize>;
 fn flush(&mut self) -> Result<()>;

 // provided default impls
 // 提供默认实现

 fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize>;
 fn is_write_vectored(&self) -> bool;
 fn write_all(&mut self, buf: &[u8]) -> Result<()>;
 fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) -> Result<()>;
 fn write_fmt(&mut self, fmt: Arguments<'_>) -> Result<()>;
 fn by_ref(&mut self) -> &mut Self
}

where
Self: Sized;
}
```

```

> Generic blanket impls worth knowing:

值得关注的一揽子泛型实现:

```

```rust
impl<R: Read + ?Sized> Read for &mut R;
impl<W: Write + ?Sized> Write for &mut W;
```

```

> These say that any mutable reference to a `Read` type is also `Read`, and same with `Write`. This is useful to know because it allows us to use any method with a `self` receiver as if it had a `&mut self` receiver. We already went over how to do this and why it's useful in the `Iterator` trait section so I'm not going to repeat it again here.

对于任何实现了 `Read` 特性的类型，其可变的引用类型也实现了 `Read` 特性。`Write` 也是如此。知晓这一点有助于我们理解为什么，对于具有 `self` 参数的函数可以如同那些具有 `&mut self` 参数的函数一般使用。鉴于我们已经在 `Iterator` 特性一节中做出了相近的说明，对此我不再赘述。

> I'd like to point out that `&[u8]` implements `Read` and that `Vec<u8>` implements `Write` so we can easily unit test our file handling functions using `String`s which are trivial to convert to `&[u8]` and from `Vec<u8>`:

我特别指出的是，在 `&[u8]` 实现 `Read` 的同时，`Vec<u8>` 实现了 `Write`，因此我们可以很方便地使用 `String` 来对我们的文件处理函数进行单元测试，因为它可以轻易地转换到 `&[u8]` 和转换自 `Vec<u8>`。

```
```rust
```

```
use std::path::Path;
use std::fs::File;
use std::io::Read;
use std::io::Write;
use std::io;

// function we want to test
// 欲要测试此函数
fn uppercase<R: Read, W: Write>(mut read: R, mut write: W) -> Result<(), io::Error> {
 let mut buffer = String::new();
 read.read_to_string(&mut buffer)?;
 let uppercase = buffer.to_uppercase();
 write.write_all(uppercase.as_bytes())?;
 write.flush()?;
 Ok(())
}

// in actual program we'd pass Files
// 实际使用中我们传入文件
fn example(in_path: &Path, out_path: &Path) -> Result<(), io::Error> {
 let in_file = File::open(in_path)?;
 let out_file = File::open(out_path)?;
```

```
uppercase(in_file, out_file)

}

// however in unit tests we can use Strings!
// 但是在单元测试中我们使用 String !
#[test] //

fn example_test() {

let in_file: String = "i am screaming".into();

let mut out_file: Vec<u8> = Vec::new();

uppercase(in_file.as_bytes(), &mut out_file).unwrap();

let out_result = String::from_utf8(out_file).unwrap();

assert_eq!(out_result, "I AM SCREAMING");

}

```

```

结语 Conclusion

> We learned a lot together! Too much in fact. This is us now:

我们真是学习了太多！太多了！可能这就是我们现在的样子：

![rust standard library traits](../../assets/jason-jarvis-stdlib-traits.png)

该漫画的创作者: [The Jenkins Comic](<https://thejenkinscomic.wordpress.com/2020/05/06/memory/>)

讨论 Discuss

> Discuss this article on

可以在如下地点讨论本文

- [Github](<https://github.com/pretzelhammer/rust-blog/discussions>)
- [learnrust subreddit](https://www.reddit.com/r/learnrust/comments/ml9shl/tour_of_rusts_standard_library_traits/)
- [official Rust users forum](<https://users.rust-lang.org/t/blog-post-tour-of-rusts-standard-library-traits/57974>)
- [Twitter](<https://twitter.com/pretzelhammer/status/1379561720176336902>)
- [lobste.rs](https://lobste.rs/s/g27ezp/tour_rust_s_standard_library_traits)
- [rust subreddit](https://www.reddit.com/r/rust/comments/mmrao0/tour_of_rusts_standard_library_traits/)

通告 Notifications

> Get notified when the next blog post get published by

在如下处得知我下一篇博文的详情

- [订阅我的推特 pretzelhammer](<https://twitter.com/pretzelhammer>) 或者
- 订阅这个 repo (点击 `Watch` -> 点击 `Custom` -> 选择 `Releases` -> 点击 `Apply`)

更多资料 Further Reading

- [Sizedness in Rust]([../../sizedness-in-rust.md](#))
- [Common Rust Lifetime Misconceptions]([./common-rust-lifetime-misconceptions.md](#))
- [Learning Rust in 2020]([../../learning-rust-in-2020.md](#))
- [Learn Assembly with Entirely Too Many Brainfuck Compilers]([../../too-many-brainfuck-compilers.md](#))

翻译 Translation

鉴于水平所限，

难免出现翻译错误，

如发现错误还请告知！

skanfd 译

2021年4月21日

引入 Intro

Have you ever wondered what's the difference between:

- `Deref<Target = T>` , `AsRef<T>` , and `Borrow<T>` ?
- `Clone` , `Copy` , and `ToOwned` ?
- `From<T>` and `Into<T>` ?
- `TryFrom<&str>` and `FromStr` ?
- `FnOnce` , `FnMut` , `Fn` , and `fn` ?

你是否曾对以下特性的区别感到困惑：

- `Deref<Target = T>` , `AsRef<T>` 和 `Borrow<T>` ?
- `Clone` , `Copy` 和 `ToOwned` ?
- `From<T>` 和 `Into<T>` ?
- `TryFrom<&str>` 和 `FromStr` ?
- `FnOnce` , `FnMut` , `Fn` 和 `fn` ?

Or ever asked yourself the questions:

- *"When do I use associated types vs generic types in my trait?"*
- *"What are generic blanket impls?"*
- *"How do subtraits and supertraits work?"*
- *"Why does this trait not have any methods?"*

或者有这样的疑问：

- “我应该在特性中使用关联类型还是泛型类型？”
- “什么是一揽子泛型实现？”
- “子特性与超特性是如何工作的？”
- “为什么某个特性没有实现任何方法？”

Well then this is the article for you! It answers all of the above questions and much much more. Together we'll do a quick flyby tour of all of the most popular and commonly used traits from the Rust standard library!

本文正是为你解答以上困惑而撰写！而且本文绝不仅仅只回答了以上问题。下面，我们将一起对 Rust 标准库中所有最流行、最常用的特性做一个走马观花般的概览！

You can read this article in order section by section or jump around to whichever traits interest you the most because each trait section begins with a list of links to **Prerequisite** sections that you should read to have adequate context to understand the current section's explanations.

你可以按顺序阅读本文，也可以直接跳读至你最感兴趣的特性。每节都会提供**预备知识**列表，它会帮助你获得相应的背景知识，不必担心跳读带来的理解困难。

特性的基础知识 Trait Basics

We'll cover just enough of the basics so that the rest of the article can be streamlined without having to repeat the same explanations of the same concepts over and over as they reappear in different traits.

本章覆盖了特性的基础知识，相应内容在以后的章节中不再赘述。

特性的记号 Trait Items

Trait items are any items that are part of a trait declaration.

特性的记号指的是，在特性的声明中可使用的记号。

Self

`Self` always refers to the implementing type.

`Self` 永远引用正被实现的类型。

```
trait Trait {  
    // always returns i32  
    // 总是返回 i32  
    fn returns_num() -> i32;  
  
    // returns implementing type  
    // 总是返回正被实现的类型  
    fn returns_self() -> Self;  
}  
  
struct SomeType;  
struct OtherType;  
  
impl Trait for SomeType {  
    fn returns_num() -> i32 {  
        5  
    }  
  
    // Self == SomeType  
    fn returns_self() -> Self {  
        Self  
    }  
}
```

```
        SomeType
    }
}

impl Trait for OtherType {
    fn returns_num() -> i32 {
        6
    }

    // Self == OtherType
    fn returns_self() -> Self {
        OtherType
    }
}
```

函数 Functions

A trait function is any function whose first parameter does not use the `self` keyword.

特性的函数指的是，任何不以 `self` 关键字作为首参数的函数。

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

Trait functions can be called namespaced by the trait or implementing type:

特性的函数同时声明在特性本身以及具体实现类型的命名空间中。

```
fn main() {
    let zero: i32 = Default::default();
    let zero = i32::default();
}
```

方法 Methods

A trait method is any function whose first parameter uses the `self` keyword and is of type `Self`, `&Self`, `&mut Self`. The former types can also be wrapped with a `Box`, `Rc`, `Arc`, or `Pin`.

特性的方法指的是，任何以 `self` 关键字作为首参数的函数，其类型是 `Self`，`&Self` 或 `&mut Self`。前者的类型也可以包裹在 `Box`，`Rc`，`Arc` 或 `Pin` 中。

```
trait Trait {  
    // methods  
    // 方法  
    fn takes_self(self);  
    fn takes_immut_self(&self);  
    fn takes_mut_self(&mut self);  
  
    // above methods desugared  
    // 以上代码等价于  
    fn takes_self(self: Self);  
    fn takes_immut_self(self: &Self);  
    fn takes_mut_self(self: &mut Self);  
}  
  
// example from standard library  
// 来自于标准库的示例  
trait ToString {  
    fn to_string(&self) -> String;  
}
```

Methods can be called using the dot operator on the implementing type:

可以使用点算符在具体实现类型上调用方法：

```
fn main() {  
    let five = 5.to_string();  
}
```

However, similarly to functions, they can also be called namespaced by the trait or implementing type:

并且，与函数相似地，方法也声明在特性本身以及具体实现类型的命名空间中。

```
fn main() {  
    let five = ToString::to_string(&5);  
    let five = i32::to_string(&5);  
}
```

关联类型 Associated Types

A trait can have associated types. This is useful when we need to use some type other than `Self` within function signatures but would still like the type to be chosen by the implementer rather than being hardcoded in the trait declaration:

特性内部可以声明关联类型。当我们希望在特性函数的签名中使用某种 `Self` 以外的类型，又不希望硬编码这种类型，而是希望后来的实现该特性的程序员来选择该类型具体是什么的时候，关联类型会很有用。

```
trait Trait {  
    type AssociatedType;  
    fn func(arg: Self::AssociatedType);  
}  
  
struct SomeType;  
struct OtherType;  
  
// any type implementing Trait can  
// choose the type of AssociatedType  
// 我们可以在实现 Trait 特性的时候  
// 再决定 AssociatedType 的具体类型  
// 而不必是在声明 Trait 特性的时候  
  
impl Trait for SomeType {  
    type AssociatedType = i8; // chooses i8  
    fn func(arg: Self::AssociatedType) {}  
}  
  
impl Trait for OtherType {  
    type AssociatedType = u8; // chooses u8  
    fn func(arg: Self::AssociatedType) {}  
}  
  
fn main() {  
    SomeType::func(-1_i8); // can only call func with i8 on SomeType  
    OtherType::func(1_u8); // can only call func with u8 on OtherType  
    // 同一特性实现在不同类型上时，可以具有不同的函数签名  
}
```

泛型参数 Generic Parameters

"Generic parameters" broadly refers to generic type parameters, generic lifetime parameters, and generic const parameters. Since all of those are a mouthful to say people commonly abbreviate them to *"generic types"*, *"lifetimes"*, and *"generic consts"*. Since generic consts are not used in any of the standard library traits we'll be covering they're outside the scope of this article.

“泛型参数”是泛型类型参数、泛型寿命参数以及泛型常量参数的统称。由于这些术语过于佶屈聱牙，我们通常将他们缩略为“泛型类型”，“泛型寿命”和“泛型常量”。鉴于标准库中的特性无一采用泛型常量，本文也略过不讲。

We can generalize a trait declaration using parameters:

我们可以使用以下参数来声明特性：

```
// trait declaration generalized with lifetime & type parameters
// 使用泛型寿命与泛型类型声明特性
trait Trait<'a, T> {
    // signature uses generic type
    // 在签名中使用泛型类型
    fn func1(arg: T);

    // signature uses lifetime
    // 在签名中使用泛型寿命
    fn func2(arg: &'a i32);

    // signature uses generic type & lifetime
    // 在签名中同时使用泛型类型与泛型寿命
    fn func3(arg: &'a T);
}

struct SomeType;

impl<'a> Trait<'a, i8> for SomeType {
    fn func1(arg: i8) {}
    fn func2(arg: &'a i32) {}
    fn func3(arg: &'a i8) {}
}

impl<'b> Trait<'b, u8> for SomeType {
    fn func1(arg: u8) {}
    fn func2(arg: &'b i32) {}
    fn func3(arg: &'b u8) {}
}
```

It's possible to provide default values for generic types. The most commonly used default value is `Self` but any type works:

可以为泛型类型指定默认值，最常用的默认值是 `Self`，此外任何其它类型都是可以的。

```
// make T = Self by default
// T 的默认值是 Self
trait Trait<T = Self> {
    fn func(t: T) {}
```

```

}

// any type can be used as the default
// 任何其它类型都可用作默认值
trait Trait2<T = i32> {
    fn func2(t: T) {}
}

struct SomeType;

// omitting the generic type will
// cause the impl to use the default
// value, which is Self here
// 省略泛型类型时，impl 块使用默认值，在这里是 Self
impl Trait for SomeType {
    fn func(t: SomeType) {}
}

// default value here is i32
// 这里的默认值是 i32
impl Trait2 for SomeType {
    fn func2(t: i32) {}
}

// the default is overridable as we'd expect
// 默认值可以被重写，正如我们希望的那样
impl Trait<String> for SomeType {
    fn func(t: String) {}
}

// overridable here too
// 这里也可以重写
impl Trait2<String> for SomeType {
    fn func2(t: String) {}
}

```

Aside from parameterizing the trait it's also possible to parameterize individual functions and methods:

不仅可以为特性提供泛型，也可以独立地为函数或方法提供泛型。

```

trait Trait {
    fn func<'a, T>(t: &'a T);
}

```

泛型类型与关联类型 Generic Types vs Associated Types

Both generic types and associated types defer the decision to the implementer on which concrete types should be used in the trait's functions and methods, so this section seeks to explain when to use one over the other.

通过使用泛型类型与关联类型，我们都可以将具体类型的选择问题抛给后来实现该特性的程序员来决定，这一节将解释我们如何在相似的两者之间做出选择。

The general rule-of-thumb is:

- Use associated types when there should only be a single impl of the trait per type.
- Use generic types when there can be many possible impls of the trait per type.

按照惯常的经验：

- 对于某一特性，每个类型仅应当有单一实现时，使用关联类型。
- 对于某一特性，每个类型可以有多个实现时，使用泛型类型。

Let's say we want to define a trait called `Add` which allows us to add values together. Here's an initial design and impl that only uses associated types:

例如，我们声明一个 `Add` 特性，它允许将各值加总在一起。这是仅使用关联类型的初始设计：

```
trait Add {  
    type Rhs;  
    type Output;  
    fn add(self, rhs: Self::Rhs) -> Self::Output;  
}
```

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

```
impl Add for Point {  
    type Rhs = Point;  
    type Output = Point;  
    fn add(self, rhs: Point) -> Point {  
        Point {
```

```

        x: self.x + rhs.x,
        y: self.y + rhs.y,
    }
}
}

fn main() {
    let p1 = Point { x: 1, y: 1 };
    let p2 = Point { x: 2, y: 2 };
    let p3 = p1.add(p2);
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);
}

```

Let's say we wanted to add the ability to add `i32`s to `Point`s where the `i32` would be added to both the `x` and `y` members:

例如，我们希望程序允许将 `i32` 类型的值与 `Point` 类型的值相加，其规则是该 `i32` 类型的值分别加到成员 `x` 与成员 `y`。

```

trait Add {
    type Rhs;
    type Output;
    fn add(self, rhs: Self::Rhs) -> Self::Output;
}

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

impl Add for Point {
    type Rhs = Point;
    type Output = Point;
    fn add(self, rhs: Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

impl Add for Point { //
    type Rhs = i32;
    type Output = Point;
    fn add(self, rhs: i32) -> Point {
        Point {
            x: self.x + rhs,
            y: self.y + rhs,
        }
    }
}

```

```

fn main() {
    let p1 = Point { x: 1, y: 1 };
    let p2 = Point { x: 2, y: 2 };
    let p3 = p1.add(p2);
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);

    let p1 = Point { x: 1, y: 1 };
    let int2 = 2;
    let p3 = p1.add(int2); //
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);
}

```

Throws:

编译出错:

```

error[E0119]: conflicting implementations of trait `Add` for type `Point`:
--> src/main.rs:23:1
|
12 | impl Add for Point {
| ----- first implementation here
...
23 | impl Add for Point {
| ^^^^^^^^^^^^^^^^^^ conflicting implementation for `Point`

```

Since the `Add` trait is not parameterized by any generic types we can only impl it once per type, which means we can only pick the types for both `Rhs` and `Output` once! To allow adding both `Point`s and `i32`s to `Point` we have to refactor `Rhs` from an associated type to a generic type, which would allow us to impl the trait multiple times for `Point` with different type arguments for `Rhs`:

由于 `Add` 特性未提供泛型类型，因而每个类型只能具有该特性的单一实现，这即是说一旦我们指定了 `Rhs` 和 `Output` 的类型后就不可再更改了！为了 `Point` 类型的值能同时接受 `i32` 类型和 `Point` 类型的值作为被加数，我们应当重构之以将 `Rhs` 从关联类型改为泛型类型，这将允许我们为 `Rhs` 指定不同的类型并为同一类型多次实现某一特性。

```

trait Add<Rhs> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}

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

```

```

impl Add<Point> for Point {
    type Output = Self;
    fn add(self, rhs: Point) -> Self::Output {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

impl Add<i32> for Point { //
    type Output = Self;
    fn add(self, rhs: i32) -> Self::Output {
        Point {
            x: self.x + rhs,
            y: self.y + rhs,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 1 };
    let p2 = Point { x: 2, y: 2 };
    let p3 = p1.add(p2);
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);

    let p1 = Point { x: 1, y: 1 };
    let int2 = 2;
    let p3 = p1.add(int2); //
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);
}

```

Let's say we add a new type called `Line` which contains two `Point`s, and now there are contexts within our program where adding two `Point`s should produce a `Line` instead of a `Point`. This is not possible given the current design of the `Add` trait where `Output` is an associated type but we can satisfy these new requirements by refactoring `Output` from an associated type into a generic type:

例如，我们现在声明一个包含两个 `Point` 类型的新类型 `Line`，要求当两个 `Point` 类型相加时返回 `Line` 而不是 `Point`。在当前 `Add` 特性的设计中 `Output` 是关联类型，不能满足这一要求，重构之以将关联类型改为泛型类型：

```

trait Add<Rhs, Output> {
    fn add(self, rhs: Rhs) -> Output;
}

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

```

```

impl Add<Point, Point> for Point {
    fn add(self, rhs: Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

impl Add<i32, Point> for Point {
    fn add(self, rhs: i32) -> Point {
        Point {
            x: self.x + rhs,
            y: self.y + rhs,
        }
    }
}

struct Line {
    start: Point,
    end: Point,
}

impl Add<Point, Line> for Point { //
    fn add(self, rhs: Point) -> Line {
        Line {
            start: self,
            end: rhs,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 1 };
    let p2 = Point { x: 2, y: 2 };
    let p3: Point = p1.add(p2);
    assert!(p3.x == 3 && p3.y == 3);

    let p1 = Point { x: 1, y: 1 };
    let int2 = 2;
    let p3 = p1.add(int2);
    assert!(p3.x == 3 && p3.y == 3);

    let p1 = Point { x: 1, y: 1 };
    let p2 = Point { x: 2, y: 2 };
    let l: Line = p1.add(p2); //
    assert!(l.start.x == 1 && l.start.y == 1 && l.end.x == 2 && l.end.y == 2)
}

```

So which `Add` trait above is the best? It really depends on the requirements of your program! They're all good in the right situations.

所以说，哪一种 `Add` 特性最好？答案是具体问题具体分析！不管白猫黑猫，会捉老鼠就是好猫。

作用域 Scope

Trait items cannot be used unless the trait is in scope. Most Rustaceans learn this the hard way the first time they try to write a program that does anything with I/O because the `Read` and `Write` traits are not in the standard library prelude:

特性仅当被引入当前作用域时才可以使用。绝大多数的初学者要在编写 I/O 程序时经历一番痛苦挣扎后，才能领悟到这一点，原因是 `Read` 和 `Write` 两个特性并未包含在标准库的 prelude 模块中。

```
use std::fs::File;
use std::io;

fn main() -> Result<(), io::Error> {
    let mut file = File::open("Cargo.toml")?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?; // read_to_string not found in File
                                    // 当前文件中找不到 read_to_string
    Ok(())
}
```

`read_to_string(buf: &mut String)` is declared by the `std::io::Read` trait and implemented by the `std::fs::File` struct but in order to call it `std::io::Read` must be in scope:

`read_to_string(buf: &mut String)` 声明于 `std::io::Read` 特性，并实现于 `std::fs::File` 类型，若要调用该函数还须得 `std::io::Read` 特性处于当前作用域中：

```
use std::fs::File;
use std::io;
use std::io::Read; //

fn main() -> Result<(), io::Error> {
    let mut file = File::open("Cargo.toml")?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?; //
    Ok(())
}
```

The standard library prelude is a module in the standard library, i.e. `std::prelude::v1` , that gets auto imported at the top of every other module, i.e. `use std::prelude::v1::*;`. Thus the following traits are always in scope and we never have to explicitly import them ourselves because they're part of the prelude:

诸如 `std::prelude::v1`，`prelude` 是标准库的一类模块，其特点是该模块命名空间下的成员将被自动导入到任何其它模块的顶部，其作用等效于 `use std::prelude::v1::*;`。因此，以下 `prelude` 模块中的特性无需我们显式导入，它们永远存在于当前作用域：

- [AsMut](#)
- [AsRef](#)
- [Clone](#)
- [Copy](#)
- [Default](#)
- [Drop](#)
- [Eq](#)
- [Fn](#)
- [FnMut](#)
- [FnOnce](#)
- [From](#)
- [Into](#)
- [ToOwned](#)
- [Intolterator](#)
- [Iterator](#)
- [PartialEq](#)
- [PartialOrd](#)
- [Send](#)
- [Sized](#)
- [Sync](#)
- [ToString](#)
- [Ord](#)

衍生宏 Derive Macros

The standard library exports a handful of derive macros which we can use to quickly and conveniently impl a trait on a type if all of its members also impl the trait. The derive macros are named after the traits they impl:

标准库导出了一系列实用的衍生宏，我们可以利用它们方便快捷地为特定类型实现某种特性，前提是该类型的成员亦实现了相应的特性。衍生宏与它们各自所实现的特性同名：

- [Clone](#)
- [Copy](#)
- [Debug](#)
- [Default](#)
- [Eq](#)
- [Hash](#)
- [Ord](#)
- [PartialEq](#)
- [PartialOrd](#)

| Example usage:

用例：

```
// macro derives Copy & Clone impl for SomeType
// 利用宏的方式为特定类型衍生出 Copy 与 Clone 特性的具体实现
#[derive(Copy, Clone)]
struct SomeType;
```

Note: derive macros are just procedural macros and can do anything, there's no hard rule that they must impl a trait or that they can only work if all the members of the type impl a trait, these are just the conventions followed by the derive macros in the standard library.

注意：衍生宏仅是一种机械的过程，宏展开之后发生的事情并无一定之规。并没有绝对的规定要求衍生宏展开之后必须要为类型实现某种特性，又或者它们必须要求该类型的所有成员都必须实现某种特性才能为当前类型实现该特性，这仅仅是在标准库衍生宏的编纂过程中逐渐约定俗成的规则。

默认实现 Default Impls

| Traits can provide default impls for their functions and methods.

特性可为函数与方法提供默认的实现。

```
trait Trait {
    fn method(&self) {
```

```

        println!("default impl");
    }
}

struct SomeType;
struct OtherType;

// use default impl for Trait::method
// 省略时使用默认实现
impl Trait for SomeType {}

impl Trait for OtherType {
    // use our own impl for Trait::method
    // 重写时覆盖默认实现
    fn method(&self) {
        println!("OtherType impl");
    }
}

fn main() {
    SomeType.method(); // prints "default impl"
    OtherType.method(); // prints "OtherType impl"
}

```

| This is especially handy if some of the trait methods can be implemented solely using other trait methods.

这对于实现特性中某些仅依赖于其它方法的方法来说极其方便。

```

trait Greet {
    fn greet(&self, name: &str) -> String;
    fn greet_loudly(&self, name: &str) -> String {
        self.greet(name) + "!"
    }
}

struct Hello;
struct Hola;

impl Greet for Hello {
    fn greet(&self, name: &str) -> String {
        format!("Hello {}", name)
    }
    // use default impl for greet_loudly
    // 省略时使用 greet_loudly 的默认实现
}

impl Greet for Hola {
    fn greet(&self, name: &str) -> String {
        format!("Hola {}", name)
    }
    // override default impl
    // 重写时覆盖 greet_loudly 的默认实现
}

```

```

fn greet_loudly(&self, name: &str) -> String {
    let mut greeting = self.greet(name);
    greeting.insert_str(0, "¡");
    greeting + "!"
}

fn main() {
    println!("{}", Hello.greet("John")); // prints "Hello John"
    println!("{}", Hello.greet_loudly("John")); // prints "Hello John!"
    println!("{}", Hola.greet("John")); // prints "Hola John"
    println!("{}", Hola.greet_loudly("John")); // prints "¡Hola John!"
}

```

| Many traits in the standard library provide default impls for many of their methods.

标准库中的许多特性都为它们的方法提供默认实现。

一揽子泛型实现 Generic Blanket Impls

| A generic blanket impl is an impl on a generic type instead of a concrete type. To explain why and how we'd use one let's start by writing an `is_even` method for number types:

一揽子泛型实现是对泛型类型的实现，与之对应的是对特定类型的实现。我们将以 `is_even` 方法为例说明如何对数字类型实现一揽子泛型实现。

```

trait Even {
    fn is_even(self) -> bool;
}

```

```

impl Even for i8 {
    fn is_even(self) -> bool {
        self % 2_i8 == 0_i8
    }
}

```

```

impl Even for u8 {
    fn is_even(self) -> bool {
        self % 2_u8 == 0_u8
    }
}

```

```

impl Even for i16 {
    fn is_even(self) -> bool {
        self % 2_i16 == 0_i16
    }
}

```

```

}

// etc

#[test] //
fn test_is_even() {
    assert!(2_i8.is_even());
    assert!(4_u8.is_even());
    assert!(6_i16.is_even());
    // etc
}

```

Obviously, this is very verbose. Also, all of our impls are almost identical. Furthermore, in the unlikely but still possible event that Rust decides to add more number types in the future we have to remember to come back to this code and update it with the new number types. We can solve all these problems using a generic blanket impl:

显而易见地，我们重复实现了近乎相同的逻辑，这非常的繁琐。进一步来讲，如果 Rust 在将来决定增加更多的数字类型（小概率事件并非绝不可能），那么我们将不得不重新回到这里对新增的数字类型编写代码。一揽子泛型实现恰可以解决这些问题：

```

use std::fmt::Debug;
use std::convert::TryInto;
use std::ops::Rem;

trait Even {
    fn is_even(self) -> bool;
}

// generic blanket impl
// 一揽子泛型实现
impl<T> Even for T
where
    T: Rem<Output = T> + PartialEq<T> + Sized,
    u8: TryInto<T>,
    <u8 as TryInto<T>>::Error: Debug,
{
    fn is_even(self) -> bool {
        // these unwraps will never panic
        // 以下 unwrap 永远不会 panic
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
    }
}

#[test] //
fn test_is_even() {
    assert!(2_i8.is_even());
    assert!(4_u8.is_even());
    assert!(6_i16.is_even());
    // etc
}

```

Unlike default impls, which provide *an* impl, generic blanket impls provide *the* impl, so they are not overridable.

默认实现可以重写，而一揽子泛型实现不可重写。

```
use std::fmt::Debug;
use std::convert::TryInto;
use std::ops::Rem;

trait Even {
    fn is_even(self) -> bool;
}

impl<T> Even for T
where
    T: Rem<Output = T> + PartialEq<T> + Sized,
    u8: TryInto<T>,
    <u8 as TryInto<T>>::Error: Debug,
{
    fn is_even(self) -> bool {
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
    }
}

impl Even for u8 { //
    fn is_even(self) -> bool {
        self % 2_u8 == 0_u8
    }
}
```

Throws:

编译出错：

```
error[E0119]: conflicting implementations of trait `Even` for type `u8`:
--> src/lib.rs:22:1
 |
10 | / impl<T> Even for T
11 || where
12 ||     T: Rem<Output = T> + PartialEq<T> + Sized,
13 ||     u8: TryInto<T>,
... |
19 || }
20 || }
   ||_- first implementation here
```

```
21 |
22 | impl Even for u8 {
|     ^^^^^^^^^^^^^^ conflicting implementation for `u8`
```

These impls overlap, hence they conflict, hence Rust rejects the code to ensure trait coherence. Trait coherence is the property that there exists at most one impl of a trait for any given type. The rules Rust uses to enforce trait coherence, the implications of those rules, and workarounds for the implications are outside the scope of this article.

重叠的实现产生了冲突，于是 Rust 拒绝了该代码以确保特性一致性。特性一致性指的是，对任意给定类型，仅能对某一特性具有单一实现。Rust 强制实现特性一致性，而这一规则的潜在影响与变通方法超出了本文的讨论范围。

子特性与超特性 Subtraits & Supertraits

The "sub" in "subtrait" refers to subset and the "super" in "supertrait" refers to superset. If we have this trait declaration:

子特性的“子”即为子集，超特性的“超”即为超集。若有下列特性声明：

```
trait Subtrait: Supertrait {}
```

All of the types which impl `Subtrait` are a subset of all the types which impl `Supertrait`, or to put it in opposite but equivalent terms: all the types which impl `Supertrait` are a superset of all the types which impl `Subtrait`.

所有实现了子特性的类型都是实现了超特性的类型的子集，也可以说，所有实现了超特性的类型都是实现了子特性的类型的超集。

Also, the above is just syntax sugar for:

以上代码等价于：

```
trait Subtrait where Self: Supertrait {}
```

It's a subtle yet important distinction to understand that the bound is on `Self`, i.e. the type impling `Subtrait`, and not on `Subtrait` itself. The latter would not make any sense, since trait bounds can only be applied to concrete types which can impl traits. Traits cannot impl other traits:

这是一种易于忽略但又至关重要的区别——约束是 `Self` 的约束，而不是 `Subtrait` 的约束。后者没有任何意义，因为特性约束只能应用于具体类型。不能用一种特性去实现其它特性：

```
trait Supertrait {  
    fn method(&self) {  
        println!("in supertrait");  
    }  
}  
  
trait Subtrait: Supertrait {  
    // this looks like it might impl or  
    // override Supertrait::method but it  
    // does not  
    // 这可能会令你产生超特性的方法被覆盖的错觉（实际不会）  
    fn method(&self) {  
        println!("in substrait")  
    }  
}  
  
struct SomeType;  
  
// adds Supertrait::method to SomeType  
impl Supertrait for SomeType {}  
  
// adds Subtrait::method to SomeType  
impl Subtrait for SomeType {}  
  
// both methods exist on SomeType simultaneously  
// neither overriding or shadowing the other  
// 两个同名方法同时存在于同一类型时，既不重写也不影射  
  
fn main() {  
    SomeType.method(); // ambiguous method call  
    // 不允许语义模糊的函数调用  
    // must disambiguate using fully-qualified syntax  
    // 必须使用完全限定的记号来明确你要使用的函数  
    <SomeType as Supertrait>::method(&st); // prints "in supertrait"  
    <SomeType as Subtrait>::method(&st); // prints "in substrait"  
}
```

Furthermore, there are no rules for how a type must impl both a subtrait and a supertrait. It can use the methods from either in the impl of the other.

此外，对于特定类型如何同时实现子特性与超特性并没有规定。子、超特性之间的方法也可以相互调用。

```
trait Supertrait {
    fn super_method(&mut self);
}

trait Subtrait: Supertrait {
    fn sub_method(&mut self);
}

struct CallSuperFromSub;

impl Supertrait for CallSuperFromSub {
    fn super_method(&mut self) {
        println!("in super");
    }
}

impl Subtrait for CallSuperFromSub {
    fn sub_method(&mut self) {
        println!("in sub");
        self.super_method();
    }
}

struct CallSubFromSuper;

impl Supertrait for CallSubFromSuper {
    fn super_method(&mut self) {
        println!("in super");
        self.sub_method();
    }
}

impl Subtrait for CallSubFromSuper {
    fn sub_method(&mut self) {
        println!("in sub");
    }
}

struct CallEachOther(bool);

impl Supertrait for CallEachOther {
    fn super_method(&mut self) {
        println!("in super");
        if self.0 {
            self.0 = false;
            self.sub_method();
        }
    }
}

impl Subtrait for CallEachOther {
    fn sub_method(&mut self) {
        println!("in sub");
        if self.0 {
            self.0 = false;
            self.super_method();
        }
    }
}
```

```

    }
}

fn main() {
    CallSuperFromSub.super_method(); // prints "in super"
    CallSuperFromSub.sub_method(); // prints "in sub", "in super"

    CallSubFromSuper.super_method(); // prints "in super", "in sub"
    CallSubFromSuper.sub_method(); // prints "in sub"

    CallEachOther(true).super_method(); // prints "in super", "in sub"
    CallEachOther(true).sub_method(); // prints "in sub", "in super"
}

```

Hopefully the examples above show that the relationship between subtraits and supertraits can be complex. Before introducing a mental model that neatly encapsulates all of that complexity let's quickly review and establish the mental model we use for understanding trait bounds on generic types:

通过以上示例，希望读者能够领会到，子特性与超特性之间的关系并未被一刀切的限制住。接下来我们将学习一种将所有这些复杂性巧妙地封装在一起的心智模型，在这之前我们先来回顾一下我们用来理解泛型类型与特性约束的关系的心智模型。

```

fn function<T: Clone>(t: T) {
    // impl
}

```

Without knowing anything about the impl of this function we could reasonably guess that `t.clone()` gets called at some point because when a generic type is bounded by a trait that strongly implies it has a dependency on the trait. The mental model for understanding the relationship between generic types and their trait bounds is a simple and intuitive one: generic types *depend on* their trait bounds.

即便我们不知道这个函数的具体实现，我们仍旧可以有理有据地猜测 `t.clone()` 将在函数的某处被调用，因为当泛型类型被特性所约束的时候，会给人一种它依赖于该特性的强烈暗示。这就是一种理解泛型类型与特性约束的关系的心智模型，它简单且可凭直觉——泛型类型依赖于它们的特性约束。

Now let's look the trait declaration for `Copy` :

现在，让我们看看 `Copy` 特性的声明：

```
trait Copy: Clone {}
```

The syntax above looks very similar to the syntax for applying a trait bound on a generic type and yet `Copy` doesn't depend on `Clone` at all. The mental model we developed earlier doesn't help us here. In my opinion, the most simple and elegant mental model for understanding the relationship between subtraits and supertraits is: subtraits *refine* their supertraits.

以上的记号和之前我们为泛型添加特性约束的记号非常相似，但是 `Copy` 却完全不依赖 `Clone`。早前建立的心智模型现在不适用了。在我看来，理解子特性与超特性的关系的最简单和最优雅的心智模型莫过于——子特性 改良 了超特性。

"Refinement" is intentionally kept somewhat vague because it can mean different things in different contexts:

- a subtrait might make its supertrait's methods' impls more specialized, faster, use less memory, e.g. `Copy: Clone`
- a subtrait might make additional guarantees about the supertrait's methods' impls, e.g. `Eq: PartialEq`, `Ord: PartialOrd`, `ExactSizeliterator: Iterator`
- a subtrait might make the supertrait's methods more flexible or easier to call, e.g. `FnMut: FnOnce`, `Fn: FnMut`
- a subtrait might extend a supertrait and add new methods, e.g. `DoubleEndedIterator: Iterator`, `ExactSizeliterator: Iterator`

“改良”一词故意地预留了一些模糊的空间，它的具体含义在不同的上下文中有所不同：

- 子特性可能比超特性的方法更加特异化、运行更快或使用更少内存等等，例如 `Copy: Clone`
- 子特性可能比超特性的方法具有额外的功能，例如 `Eq: PartialEq`, `Ord: PartialOrd` 和 `ExactSizeliterator: Iterator`
- 子特性可能比超特性的方法更灵活和更易于调用，例如 `FnMut: FnOnce` 和 `Fn: FnMut`
- 子特性可能扩展了超特性并添加了新的方法，例如 `DoubleEndedIterator: Iterator` 和 `ExactSizeliterator: Iterator`

特性对象 Trait Objects

Generics give us compile-time polymorphism where trait objects give us run-time polymorphism. We can use trait objects to allow functions to dynamically return different types at run-time:

如果说泛型给了我们编译时的多态性，那么特性对象就给了我们运行时的多态性。通过特性对象，我们可以允许函数在运行时动态地返回不同的类型。

```
fn example(condition: bool, vec: Vec<i32>) -> Box<dyn Iterator<Item = i32>> {
    let iter = vec.into_iter();
    if condition {
        // Has type:
        // Box<Map<Intolter<i32>, Fn(i32) -> i32>>
        // But is cast to:
        // Box<dyn Iterator<Item = i32>>
        Box::new(iter.map(|n| n * 2))
    } else {
        // Has type:
        // Box<Filter<Intolter<i32>, Fn(&i32) -> bool>>
        // But is cast to:
        // Box<dyn Iterator<Item = i32>>
        Box::new(iter.filter(|&n| n >= 2))
    }
}
// 以上代码中，两种不同的指针类型转换成相同的指针类型
```

Trait objects also allow us to store heterogeneous types in collections:

特性对象也允许我们在集合中存储不同类型的值：

```
use std::f64::consts::PI;

struct Circle {
    radius: f64,
}

struct Square {
    side: f64
}

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

impl Shape for Circle {
    fn area(&self) -> f64 {
        PI * self.radius * self.radius
    }
}

impl Shape for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}
```

```

fn get_total_area(shapes: Vec<Box<dyn Shape>>) -> f64 {
    shapes.into_iter().map(|s| s.area()).sum()
}

fn example() {
    let shapes: Vec<Box<dyn Shape>> = vec![
        Box::new(Circle { radius: 1.0 }), // Box<Circle> cast to Box<dyn Shape>
        Box::new(Square { side: 1.0 }), // Box<Square> cast to Box<dyn Shape>
    ];
    assert_eq!(Pi + 1.0, get_total_area(shapes)); //
}

```

Trait objects are unsized so they must always be behind a pointer. We can tell the difference between a concrete type and a trait object at the type level based on the presence of the `dyn` keyword within the type:

特性对象的结构体大小是未知的，所以必须要通过指针来引用它们。具体类型与特性对象在字面上的区别在于，特性对象必须用 `dyn` 关键字来修饰前缀，了解了这一点我们可以轻松辨别二者。

```

struct Struct;
trait Trait {}

// regular struct
// 这是一般的结构
&Struct
Box<Struct>
Rc<Struct>
Arc<Struct>

// trait objects
// 这是特性对象
&dyn Trait
Box<dyn Trait>
Rc<dyn Trait>
Arc<dyn Trait>

```

Not all traits can be converted into trait objects. A trait is object-safe if it meets these requirements:

- trait doesn't require `Self: Sized`
- all of the trait's methods are object-safe

并非全部的特性都可以转换为特性对象，一个“对象安全”的特性必须满足：

- 该特性不要求 `Self: Sized`
- 该特性的所有方法都是“对象安全”的

A trait method is object-safe if it meets these requirements:

- method requires `Self: Sized` or
- method only uses a `Self` type in receiver position

一个特性的方法若要是“对象安全”的，必须满足：

- 该方法要求 `Self: Sized`
- 该方法仅在接收参数中使用 `Self` 类型

Understanding why the requirements are what they are is not relevant to the rest of this article, but if you're still curious it's covered in [Sizedness in Rust](#).

关于具有这些限制条件的原因超出了本文的讨论范围且与下文无关，如果你对此深感兴趣不妨阅读 [Sizedness in Rust](#) 以了解详情。

仅用于标记的特性 Marker Traits

Marker traits are traits that have no trait items. Their job is to "mark" the implementing type as having some property which is otherwise not possible to represent using the type system.

仅用于标记的特性，即是某种声明体为空的特性。它们存在的意义在于“标记”所实现的类型，且该类型具有某种类型系统所无法表达的属性。

```
// Imploring PartialEq for a type promises
// that equality for the type has these properties:
// - symmetry: a == b implies b == a, and
// - transitivity: a == b && b == c implies a == c
// But DOES NOT promise this property:
// - reflexivity: a == a
// 为特定类型实现 PartialEq 特性确保了该类型的相等算符具有以下性质:
```

```

// - 对称性：若有 a == b，则必有 b == a
// - 传递性：若有 a == b 和 b == c，则必有 a == c
// 但是不能确保具有以下性质：
// - 自反性：a == a
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
}

// Eq has no trait items! The eq method is already
// declared by PartialEq, but "impling" Eq
// for a type promises this additional equality property:
// - reflexivity: a == a
// Eq 特性的声明体是空的！而 eq 方法已经被 PartialEq 所声明，
// 但是对特定类型“实现”Eq 特性确保了额外的相等性质：
// - 自反性：a == a
trait Eq: PartialEq {}

// f64 implements PartialEq but not Eq because NaN != NaN
// i32 implements PartialEq & Eq because there's no NaNs :)
// f64 实现了 PartialEq 特性但是没有实现 Eq 特性，因为 NaN != NaN
// i32 同时实现了 PartialEq 特性与 Eq 特性，因为没有 NaN 来捣乱 :)

```

可自动实现的特性 Auto Traits

Auto traits are traits that get automatically implemented for a type if all of its members also impl the trait. What “members” means depends on the type, for example: fields of a struct, variants of an enum, elements of an array, items of a tuple, and so on.

可自动实现的特性指的是，存在这样一种特性，若给定类型的成员都实现了该特性，那么该类型就隐式地自动实现该特性。这里所说的“成员”依据上下文而具有不同的含义，包括而又不限于结构体的字段、枚举的变量、数组的元素和元组的内容等等。

All auto traits are marker traits but not all marker traits are auto traits. Auto traits must be marker traits so the compiler can provide an automatic default impl for them, which would not be possible if they had any trait items.

所有可自动实现的特性都是仅用于标记的特性，反之则不是。正是由于可自动实现的特性必须是仅用于标记的特性，所以编译器才能够自动为其提供一个默认实现，反之编译器就无能为力了。

Examples of auto traits:

可自动实现的特性的示例：

```
// implemented for types which are safe to send between threads
// 实现 Send 特性的类型可以安全地往返于多个线程
unsafe auto trait Send {}

// implemented for types whose references are safe to send between threads
// 实现 Sync 特性的类型，其引用可以安全地往返于多个线程
unsafe auto trait Sync {}
```

不安全的特性 Unsafe Traits

Traits can be marked unsafe to indicate that impling the trait might require unsafe code. Both `Send` and `Sync` are marked `unsafe` because if they aren't automatically implemented for a type that means it must contain some non-`Send` or non-`Sync` member and we have to take extra care as the implementers to make sure there are no data races if we want to manually mark the type as `Send` and `Sync`.

以 `unsafe` 修饰前缀的特性，意味着该特性的实现可能需要不安全的代码。`Send` 特性与 `Sync` 特性以 `unsafe` 修饰前缀意味着，如果特定类型没有自动实现该特性，那么说明该类型的成员并非都实现了该特性，这提示着我们手动实现该特性一定要谨慎小心，以确保没有发生数据竞争。

```
// SomeType is not Send or Sync
// SomeType 没有实现 Send 和 Sync
struct SomeType {
    not_send_or_sync: *const (),
}

// but if we're confident that our impl doesn't have any data
// races we can explicitly mark it as Send and Sync using unsafe
// 倘若我们得以社会主义伟大成就的庇佑自信地写出没有数据竞争的代码
// 可以使用 unsafe 来修饰前缀，以显式地实现 Send 特性与 Sync 特性
unsafe impl Send for SomeType {}
unsafe impl Sync for SomeType {}
```

可自动实现的特性 Auto Traits

Send & Sync

预备知识

- [Marker Traits](#)
- [Auto Traits](#)
- [Unsafe Traits](#)

```
unsafe auto trait Send {}  
unsafe auto trait Sync {}
```

If a type is `Send` that means it's safe to send between threads. If a type is `Sync` that means it's safe to share references of it between threads. In more precise terms some type `T` is `Sync` if and only if `&T` is `Send`.

实现 `Send` 特性的类型可以安全地往返于多线程。实现 `Sync` 特性的类型，其引用可以安全地往返于多线程。用更加准确的术语来讲，当且仅当 `&T` 实现 `Send` 特性时，`T` 才能实现 `Sync` 特性。

Almost all types are `Send` and `Sync`. The only notable `Send` exception is `Rc` and the only notable `Sync` exceptions are `Rc`, `Cell`, `RefCell`. If we need a `Send` version of `Rc` we can use `Arc`. If we need a `Sync` version of `Cell` or `RefCell` we can `Mutex` or `RwLock`. Although if we're using the `Mutex` or `RwLock` to just wrap a primitive type it's often better to use the atomic primitive types provided by the standard library such as `AtomicBool`, `AtomicI32`, `AtomicUsize`, and so on.

几乎所有类型都实现了 `Send` 特性和 `Sync` 特性。对于 `Send` 唯一需要注意的例外是 `Rc`，对于 `Sync` 唯三需要注意的例外是 `Rc`，`Cell` 和 `RefCell`。如果我们需要 `Send` 版的 `Rc`，可以使用 `Arc`。如果我们需要 `Sync` 版的 `Cell` 或 `RefCell`，可以使用 `Mutex` 或 `RwLock`。尽管我们可以使用 `Mutex` 或 `RwLock` 来包裹住原语类型，但通常使用标准库提供的原子原语类型会更好，诸如 `AtomicBool`，`AtomicI32` 和 `AtomicUsize` 等等。

That almost all types are `Sync` might be a surprise to some people, but yup, it's true even for types without any internal synchronization. This is possible thanks to Rust's strict borrowing rules.

多亏了 Rust 严格的借用规则，几乎所有的类型都是 `Sync` 的。这对于一些人来讲可能会很惊讶，但事实胜于雄辩，甚至对于那些没有内部同步机制的类型来说也是如此。

We can pass many immutable references to the same data to many threads and we're guaranteed there are no data races because as long as any immutable references exist Rust statically guarantees the underlying data cannot be mutated:

对于同一数据，我们可以放心地将该数据的多个不可变引用传递给多个线程，因为只要当前存在一个该数据的不可变引用，那么 Rust 就会静态地确保该数据不会被改变：

```
use crossbeam::thread;  
  
fn main() {
```

```

let mut greeting = String::from("Hello");
let greeting_ref = &greeting;

thread::scope(|scoped_thread| {
    // spawn 3 threads
    // 产生三个线程
    for n in 1..=3 {
        // greeting_ref copied into every thread
        // greeting_ref 被拷贝到每个线程
        scoped_thread.spawn(move |_| {
            println!("{} {}", greeting_ref, n); // prints "Hello {n}"
        });
    }

    // line below could cause UB or data races but compiler rejects it
    // 下面这行代码可能导致数据竞争，于是编译器拒绝了它
    greeting += " world";
    // cannot mutate greeting while immutable refs exist
    // 当不可变引用存在时，不可以修改引用的数据
});

// can mutate greeting after every thread has joined
// 当所有的线程结束之后，可以修改数据
greeting += " world"; //
println!("{}", greeting); // prints "Hello world"
})

```

Likewise we can pass a single mutable reference to some data to a single thread and we're guaranteed there will be no data races because Rust statically guarantees aliased mutable references cannot exist and the underlying data cannot be mutated through anything other than the single existing mutable reference:

同样地，我们可以将某个数据的单个可变引用传递给单个线程，在此过程中不必担心出现数据竞争，因为 Rust 静态地确保了不存在其它可变引用。以下数据即仅可通过已经存在的单个可变引用而改变：

```

use crossbeam::thread;

fn main() {
    let mut greeting = String::from("Hello");
    let greeting_ref = &mut greeting;

    thread::scope(|scoped_thread| {
        // greeting_ref moved into thread
        // greeting_ref 移动到当前线程
        scoped_thread.spawn(move |_| {
            *greeting_ref += " world";
            println!("{}", greeting_ref); // prints "Hello world"
        });

        // line below could cause UB or data races but compiler rejects it
        // 下面这行代码可能导致数据竞争，于是编译器拒绝了它
        greeting += "!!!";
        // cannot mutate greeting while mutable refs exist
    })
}

```

```
// 可变引用存在时不可改变数据
});

// can mutate greeting after the thread has joined
// 当所有的线程结束之后，可以修改数据
greeting += "!!!"; //
println!("{}", greeting); // prints "Hello world!!!"
}
```

This is why most types are `Sync` without requiring any explicit synchronization. In the event we need to simultaneously mutate some data `T` across multiple threads the compiler won't let us until we wrap the data in a `Arc<Mutex<T>>` or `Arc<RwLock<T>>` so the compiler enforces that explicit synchronization is used when it's needed.

这就是为什么绝大多数的类型都是 `Sync` 的而不需要实现任何显式的同步机制。对于数据 `T`，如果我们试图从多个线程同时修改的话，编译器会对我们作出警告，除非我们将数据包裹在 `Arc<Mutex<T>>` 或 `Arc<RwLock<T>>` 中。所以说，当我们真的需要显式的同步机制时，编译器会强制要求我们这样做的。

Sized

预备知识

- [Marker Traits](#)
- [Auto Traits](#)

If a type is `Sized` that means its size in bytes is known at compile-time and it's possible to put instances of the type on the stack.

如果一个类型实现了 `Sized`，那么说明该类型具体大小的字节数在编译时可以确定，并且也就说明该类型的实例可以存放在栈上。

Sizedness of types and its implications is a subtle yet huge topic that affects a lot of different aspects of the language. It's so important that I wrote an entire article on it called [Sizedness in Rust](#) which I highly recommend reading for anyone who would like to understand sizedness in-depth. I'll summarize a few key things which are relevant to this article.

类型的大小以及其所带来的潜在影响，是一个易于忽略但是又十分宏大的话题，它深刻地影响着本门语言的诸多方面。鉴于它的重要性，我已经写了一整篇文章 ([Sizedness in Rust](#)) 来具体阐述其内容，我高度推荐对于希望深入 sizedness 的人阅读此篇文章。下面是此篇文章的要点：

1. All generic types get an implicit `Sized` bound.

1. 所有的泛型类型都具有隐式的 `Sized` 约束。

```
fn func<T>(t: &T) {}  
  
// example above desugared  
// 以上代码等价于  
fn func<T: Sized>(t: &T) {}
```

2. Since there's an implicit `Sized` bound on all generic types, if we want to opt-out of this implicit bound we need to use the special "*relaxed bound*" syntax `?Sized` which currently only exists for the `Sized` trait:

2. 由于所有的泛型类型都具有隐式的 `Sized` 约束，如果我们希望摆脱这样的隐式约束，那么我们需要使用特殊的“宽松约束”记号 `?Sized`，目前这样的记号仅适用于 `Sized` 特性：

```
// now T can be unsized  
// 现在 T 的大小可以是未知的  
fn func<T: ?Sized>(t: &T) {}
```

3. There's an implicit `?Sized` bound on all traits.

3. 所有的特性都具有隐式的 `?Sized` 约束。

```
trait Trait {}  
  
// example above desugared  
// 以上代码等价于  
trait Trait: ?Sized {}
```

| This is so that trait objects can impl the trait. Again, all of the nitty gritty details are in [Sizedness in Rust](#).

这就是为什么特性对象可以实现具体特性。再次，向您推荐关于一切真相的[Sizedness in Rust](#)。

常用特性 General traits

Default

预备知识

- [Self](#)
- [Functions](#)
- [Derive Macros](#)

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

| It's possible to construct default values of `Default` types.

为特定类型实现 `Default` 特性时，即为该类型赋予了可选的默认值。

```
struct Color {  
    r: u8,  
    g: u8,  
    b: u8,  
}  
  
impl Default for Color {  
    // default color is black
```

```
// 默认颜色是黑色
fn default() -> Self {
    Color {
        r: 0,
        g: 0,
        b: 0,
    }
}
```

| This is useful for quick prototyping but also in any instance where we just need an instance of a type and aren't picky about what it is:

这不仅利于快速原型设计，另外，在有时我们仅仅只是需要该类型的一个值，却完全不在意该值是什么的时候，这也非常方便。

```
fn main() {
    // just give me some color!
    let color = Color::default();
}
```

| This is also an option we may want to explicitly expose to the users of our functions:

如此，我们可以明确地向该函数的用户传达出该函数某个参数的可选择性：

```
struct Canvas;
enum Shape {
    Circle,
    Rectangle,
}

impl Canvas {
    // let user optionally pass a color
    // 用户可选地传入一个 color
    fn paint(&mut self, shape: Shape, color: Option<Color>) {
        // if no color is passed use the default color
        // 若用户没有传入 color，即使用默认的 color
        let color = color.unwrap_or_default();
        // etc
    }
}
```

| **Default** is also useful in generic contexts where we need to construct generic types:

在泛型编程的语境中，`Default` 特性也可显其威力。

```
fn guarantee_length<T: Default>(mut vec: Vec<T>, min_len: usize) -> Vec<T> {
    for _ in 0..min_len.saturating_sub(vec.len()) {
        vec.push(T::default());
    }
    vec
}
```

Another way we can take advantage of `Default` types is for partial initialization of structs using Rust's struct update syntax. We may have a `new` constructor for `Color` which takes every member as an argument:

另外，我们在使用 `update` 记号构造结构体时也可享受到 `Default` 特性带来的便利。我们以 `Color` 结构的 `new` 构造器函数为例，它接受该结构的全部成员作为参数：

```
impl Color {
    fn new(r: u8, g: u8, b: u8) -> Self {
        Color {
            r,
            g,
            b,
        }
    }
}
```

However we can also have convenience constructors that only accept a particular struct member each and fall back to the default values for the other struct members:

考虑以下更加便捷的构造器函数——它仅接受该结构的部分成员作为参数，其它未指定的成员则回落到默认值：

```
impl Color {
    fn red(r: u8) -> Self {
        Color {
            r,
            ..Color::default()
        }
    }
    fn green(g: u8) -> Self {
        Color {
            g,
            ..Color::default()
        }
    }
}
```

```
fn blue(b: u8) -> Self {  
    Color {  
        b,  
        ..Color::default()  
    }  
}
```

| There's also a `Default` derive macro for so we can write `Color` like this:

`Default` 特性也可以用衍生宏的方式来实现：

```
// default color is still black  
// because u8::default() == 0  
// 默认颜色仍旧是黑色  
// 因为 u8::default() == 0  
#[derive(Default)]  
struct Color {  
    r: u8,  
    g: u8,  
    b: u8  
}
```

Clone

预备知识

- [Self](#)
- [Methods](#)
- [Default Impls](#)
- [Derive Macros](#)

```
trait Clone {  
    fn clone(&self) -> Self;  
  
    // provided default impls  
    // 提供默认实现  
    fn clone_from(&mut self, source: &Self);  
}
```

We can convert immutable references of `Clone` types into owned values, i.e. `&T -> T`. `Clone` makes no promises about the efficiency of this conversion so it can be slow and expensive. To quickly impl `Clone` on a type we can use the derive macro:

对于实现了 `Clone` 特性的类型，我们可以将一个不可变的引用转换为自有的类型，比如 `&T -> T`。`Clone` 特性对于这种转换的效率不做出保证，所以这样的转换速度可能很慢，代价可能很昂贵。

```
#[derive(Clone)]
struct SomeType {
    cloneable_member1: CloneableType1,
    cloneable_member2: CloneableType2,
    // etc
}

// macro generates impl below
// 宏展开后为
impl Clone for SomeType {
    fn clone(&self) -> Self {
        SomeType {
            cloneable_member1: self.cloneable_member1.clone(),
            cloneable_member2: self.cloneable_member2.clone(),
            // etc
        }
    }
}
```

`Clone` can also be useful in constructing instances of a type within a generic context. Here's an example from the previous section except using `Clone` instead of `Default`:

`Clone` 特性也有利于在泛型编程的语境中构造类型。请看下例：

```
fn guarantee_length<T: Clone>(mut vec: Vec<T>, min_len: usize, fill_with: &T) -> Vec<T> {
    for _ in 0..min_len.saturating_sub(vec.len()) {
        vec.push(fill_with.clone());
    }
    vec
}
```

People also commonly use cloning as an escape hatch to avoid dealing with the borrow checker. Managing structs with references can be challenging, but we can turn the references into owned values by cloning them.

克隆确是一个可以逃避借用检查器的好方法。倘若我们编写的代码无法通过借用检查，那么不妨通过克隆将这些引用转换为自有类型。

```

// oof, we gotta worry about lifetimes
// 糟糕！我们真的有自信处理好 lifetime 吗？
struct SomeStruct<'a> {
    data: &'a Vec<u8>,
}

// now we're on easy street
// 好耶！人生苦短，我用 Clone !
struct SomeStruct {
    data: Vec<u8>,
}

```

If we're working on a program where performance is not the utmost concern then we don't need to sweat cloning data. Rust is a low-level language that exposes a lot of low-level details so it's easy to get caught up in premature optimizations instead of actually solving the problem at hand. For many programs the best order of priorities is usually to build for correctness first, elegance second, and performance third, and only focus on performance after the program has been profiled and the performance bottlenecks have been identified. This is good general advice to follow, and if it doesn't apply to your particular program then you would know.

如果性能因素微不足道，我们不必羞于使用克隆。Rust 是一门底层语言，人们可以自由地控制程序行为的方方面面，这就很容易令人陷入盲目追求优化的陷阱，而不是专注于着手解决问题。对此我给出的建议是：正确第一，优雅第二，性能第三。只有程序初具雏形后，性能瓶颈的问题才可能凸显，这时我们再解决性能问题也不迟。与其说这是一条编程建议，更不如说这是一条人生建议，万事万物大抵如此，如果你现在不信，总有一天你会的。

Copy

预备知识

- [Marker Traits](#)
- [Subtraits & Supertraits](#)
- [Derive Macros](#)

```
trait Copy: Clone {
```

We copy `Copy` types, e.g. `T -> T`. `Copy` promises the copy operation will be a simple bitwise copy so it will be very fast and efficient. We cannot impl `Copy` ourselves, only the compiler can provide an impl, but we can tell it to do so by using the `Copy` derive macro, together with the `Clone` derive macro since `Copy` is a subtrait of `Clone` :

对于实现了 `Copy` 特性的类型，我们可以拷贝它，即 `T -> T`。`Copy` 特性确保了拷贝操作是按位的拷贝，所以它更快更高效。`Copy` 特性不可手动实现，必须由编译器提供其实现。注意：当使用衍生宏为类型实现 `Copy` 特性时，必须同时使用 `Clone` 衍生宏，因为 `Copy` 是 `Clone` 的子特性：

```
##[derive(Copy, Clone)]
struct SomeType;
```

`Copy` refines `Clone`。一个克隆可能速度缓慢且代价昂贵，但是拷贝操作一定是高效低耗的，所以说拷贝就是一种物美价廉的克隆。`Copy` 特性的实现会令 `Clone` 特性的实现变得微不足道：

```
// this is what the derive macro generates
// 衍生宏展开如下
impl<T: Copy> Clone for T {
    // the clone method becomes just a copy
    // 克隆实际上变成了一种拷贝
    fn clone(&self) -> Self {
        *self
    }
}
```

Impling `Copy` for a type changes its behavior when it gets moved. By default all types have *move semantics* but once a type implements `Copy` it gets *copy semantics*. To explain the difference between the two let's examine these simple scenarios:

实现了 `Copy` 特性的类型，其在移动时的行为会发生变化。默认情况下，所有的类型都具有 *移动语义*，但是一旦该类型实现了 `Copy` 特性，则会变为 *拷贝语义*。请考虑下例中语义的不同：

```
// a "move", src: !Copy
// 移动语义，src 没有实现 Copy 特性
let dest = src;

// a "copy", src: Copy
// 拷贝语义，src 实现 Copy 特性
let dest = src;
```

In both cases, `dest = src` performs a simple bitwise copy of `src`'s contents and moves the result into `dest`，the only difference is that in the case of "a move" the borrow checker invalidates the `src` variable and makes sure it's not used anywhere else later and in the case of "a copy" `src` remains valid and usable.

事实上，这两种语义背后执行的操作是完全相同的，都是将 `src` 按位复制到 `dest`。其不同在于，在移动语义下，借用检查器从此吊销了 `src` 的可用性，而在拷贝语义下，`src` 保持可用。

In a nutshell: Copies are moves. Moves are copies. The only difference is how they're treated by the borrow checker.

言而总之，拷贝就是移动，移动就是拷贝。它们在底层毫无二致，仅仅是借用检查器对待它们的方式不同。

For a more concrete example of a move, imagine `src` was a `Vec<i32>` and its contents looked something like this:

对于移动行为来讲更具体的例子——你可以将 `src` 想象为一个 `Vec<i32>`，它的结构体大致如下：

```
{ data: *mut [i32], length: usize, capacity: usize }
```

When we write `dest = src` we end up with:

执行 `desc = src` 的结果如下：

```
src = { data: *mut [i32], length: usize, capacity: usize }
dest = { data: *mut [i32], length: usize, capacity: usize }
```

At this point both `src` and `dest` have aliased mutable references to the same data, which is a big no-no, so the borrow checker invalidates the `src` variable so it can't be used again without throwing a compile error.

此时 `src` 和 `dest` 就都是同一数据的可变引用了，这可就糟tm的大糕了，所以借用检查器就吊销了 `src` 的可用性，一旦再次使用 `src` 就会引发编译错误。

For a more concrete example of a copy, imagine `src` was an `Option<i32>` and its contents looked something like this:

对于拷贝行为来讲更具体的例子——你可以将 `src` 想象为一个 `Option<i32>`，它的结构体大致如下：

```
{ is_valid: bool, data: i32 }
```

| Now when we write `dest = src` we end up with:

执行 `desc = src` 的结果如下：

```
src = { is_valid: bool, data: i32 }
dest = { is_valid: bool, data: i32 }
```

| These are both usable simultaneously! Hence `Option<i32>` is `Copy`.

此时两者同时可用！因为 `Option<i32>` 实现了 `Copy`。

| Although `Copy` could be an auto trait the Rust language designers decided it's simpler and safer for types to explicitly opt into copy semantics rather than silently inheriting copy semantics whenever the type is eligible, as the latter can cause surprising confusing behavior which often leads to bugs.

或许你已经注意到，令 `Copy` 特性成为可自动实现的特性在理论上是可行的。但是 Rust 语言的设计者认为，比之于在恰当时隐式地继承拷贝语义，显示地声明为拷贝语义更加的简单和安全。前者可能会导致 Rust 语言产生十分反人类的行为，也更容易出现 bug。

Any

预备知识

- [Self](#)
- [Generic Blanket Impls](#)
- [Subtraits & Supertraits](#)
- [Trait Objects](#)

```
trait Any: 'static {
    fn type_id(&self) -> TypeId;
}
```

Rust's style of polymorphism is parametric, but if we're looking to use a more ad-hoc style of polymorphism similar to dynamically-typed languages then we can emulate that using the `Any` trait. We don't have to manually impl this trait for our types because that's already covered by this generic blanket impl:

Rust 的多态性风格本身是参数化的，但如果希望临时使用一种更贴近于动态语言的多态性风格，可以借用 `Any` 特性来模拟。我们不需要手动实现 `Any` 特性，因为该特性通常由一揽子泛型实现所实现。

```
impl<T: 'static + ?Sized> Any for T {  
    fn type_id(&self) -> TypeId {  
        TypeId::of::<T>()  
    }  
}
```

The way we get a `T` out of a `dyn Any` is by using the `downcast_ref::<T>()` and `downcast_mut::<T>()` methods:

对于 `dyn Any` 的特性对象，我们可以使用 `downcast_ref::<T>()` 或 `downcast_mut::<T>()` 来尝试解析出 `T`。

```
use std::any::Any;  
  
#[derive(Default)]  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl Point {  
    fn inc(&mut self) {  
        self.x += 1;  
        self.y += 1;  
    }  
}  
  
fn map_any(mut any: Box<dyn Any>) -> Box<dyn Any> {  
    if let Some(num) = any.downcast_mut::<i32>() {  
        *num += 1;  
    } else if let Some(string) = any.downcast_mut::<String>() {  
        *string += "!";  
    } else if let Some(point) = any.downcast_mut::<Point>() {  
        point.inc();  
    }  
    any  
}  
  
fn main() {
```

```

let mut vec: Vec<Box<dyn Any>> = vec![
    Box::new(0),
    Box::new(String::from("a")),
    Box::new(Point::default()),
];
// vec = [0, "a", Point { x: 0, y: 0 }]
vec = vec.into_iter().map(map_any).collect();
// vec = [1, "a!", Point { x: 1, y: 1 }]
}

```

This trait rarely *needs* to be used because on top of parametric polymorphism being superior to ad-hoc polymorphism in most scenarios the latter can also be emulated using enums which are more type-safe and require less indirection. For example, we could have written the above example like this:

这个特性鲜少被使用，因为参数化的多态性时常要优于这样变通使用的多态性，且后者也可以使用更加类型安全和更加直接的枚举来模拟。如下例：

```

#[derive(Default)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn inc(&mut self) {
        self.x += 1;
        self.y += 1;
    }
}

enum Stuff {
    Integer(i32),
    String(String),
    Point(Point),
}

fn map_stuff(mut stuff: Stuff) -> Stuff {
    match &mut stuff {
        Stuff::Integer(num) => *num += 1,
        Stuff::String(string) => *string += "!",
        Stuff::Point(point) => point.inc(),
    }
    stuff
}

fn main() {
    let mut vec = vec![
        Stuff::Integer(0),
        Stuff::String(String::from("a")),
        Stuff::Point(Point::default()),
    ];
    // vec = [0, "a", Point { x: 0, y: 0 }]
}

```

```
vec = vec.into_iter().map(map_stuff).collect();
// vec = [1, "a!", Point { x: 1, y: 1 }]
}
```

Despite `Any` rarely being *needed* it can still be convenient to use sometimes, as we'll later see in the [Error Handling](#) section.

尽管 `Any` 特性鲜少是必须要被使用的，但有时它又是一种非常便捷的用法，我们将在 [错误处理](#) 一章中领会这一点。

文本格式化特性 Formatting Traits

We can serialize types into strings using the formatting macros in `std::fmt`, the most well-known of the bunch being `println!`. We can pass formatting parameters to the `{}` placeholders used within `format` `str`'s which are then used to select which trait impl to use to serialize the placeholder's argument.

我们可以使用 `std::fmt` 中提供的文本格式化宏来序列化结构体，例如我们最熟悉的 `println!`。我们可以将文本格式化的参数传入 `{}` 占位符，以选择具体用哪个特性来序列化该结构。

| 特性 | 占位符 | 描述 |
|-----------------------|-------------------|-----------|
| <code>Display</code> | <code>{}</code> | 常规序列化 |
| <code>Debug</code> | <code>{:?}</code> | 调试序列化 |
| <code>Octal</code> | <code>{:o}</code> | 八进制序列化 |
| <code>LowerHex</code> | <code>{:x}</code> | 小写十六进制序列化 |
| <code>UpperHex</code> | <code>{:X}</code> | 大写十六进制序列化 |
| <code>Pointer</code> | <code>{:p}</code> | 内存地址 |
| <code>Binary</code> | <code>{:b}</code> | 二进制序列化 |
| <code>LowerExp</code> | <code>{:e}</code> | 小写指数序列化 |
| <code>UpperExp</code> | <code>{:E}</code> | 大写十六进制序列化 |

Display & ToString

预备知识

- [Self](#)
- [Methods](#)
- [Generic Blanket Impls](#)

```
trait Display {  
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;  
}
```

`Display` types can be serialized into `String`s which are friendly to the end users of the program. Example impl for `Point`:

实现 `Display` 特性的类型可以被序列化为 `String`。这对于程序的用户来说非常的友好。例如：

```
use std::fmt;  
  
#[derive(Default)]  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl fmt::Display for Point {  
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {  
        write!(f, "({}, {})", self.x, self.y)  
    }  
}  
  
fn main() {  
    println!("origin: {}", Point::default());  
    // prints "origin: (0, 0)"  
  
    // get Point's Display representation as a String  
    // Point 表达为可显示的 String  
    let stringified_point = format!("{}", Point::default());  
    assert_eq!("(0, 0)", stringified_point); //  
}
```

Aside from using the `format!` macro to get a type's display representation as a `String` we can use the `ToString` trait:

除了使用 `format!` 宏来序列化结构体，我们也可以使用 `ToString` 特性：

```
trait ToString {
    fn to_string(&self) -> String;
}
```

| There's no need for us to impl this ourselves. In fact we can't, because of this generic blanket impl that automatically implements `ToString` for any type which implements `Display` :

我们不需要自己手动实现，事实上，我们也不能，因为对于实现了 `Display` 的类型来说，`ToString` 是由一揽子泛型实现所自动实现的。

```
impl<T: Display + ?Sized> ToString for T;
```

| Using `ToString` with `Point` :

对 `Point` 使用 `ToString` 特性：

```
#[test] //
fn display_point() {
    let origin = Point::default();
    assert_eq!(format!("{}", origin), "(0, 0)");
}

#[test] //
fn point_to_string() {
    let origin = Point::default();
    assert_eq!(origin.to_string(), "(0, 0)");
}

#[test] //
fn display_equals_to_string() {
    let origin = Point::default();
    assert_eq!(format!("{}", origin), origin.to_string());
}
```

Debug

预备知识

- [Self](#)
- [Methods](#)
- [Derive Macros](#)
- [Display & ToString](#)

```
trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```

`Debug` has an identical signature to `Display`. The only difference is that the `Debug` impl is called when we use the `{:?}` formatting specifier. `Debug` can be derived:

`Debug` 与 `Display` 具有相同的签名。唯一的区别在于我们使用 `{:?}` 文本格式化指令来调用 `Debug` 特性。`Debug` 特性可以使用如下方法衍生：

```
use std::fmt;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

// derive macro generates impl below
// 衍生宏展开如下
impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Point")
            .field("x", &self.x)
            .field("y", &self.y)
            .finish()
    }
}
```

Impling `Debug` for a type also allows it to be used within the `dbg!` macro which is superior to `println!` for quick and dirty print logging. Some of its advantages:

1. `dbg!` prints to stderr instead of stdout so the debug logs are easy to separate from the actual stdout output of our program.
2. `dbg!` prints the expression passed to it as well as the value the expression evaluated to.
3. `dbg!` takes ownership of its arguments and returns them so you can use it within expressions:

为特定类型实现 `Debug` 特性的同时，这也使得我们可以使用 `dbg!` 宏来快速地调试程序，这种方式要优于 `println!`。其优点在于：

1. `dbg!` 输出到标准错误流而不是标准输出流，所以我们能够很容易地将调试信息提取出来。
2. `dbg!` 同时输出值和值的求值表达式。
3. `dbg!` 接管参数的属权，但不会吞掉参数，而是再抛出来，所以可以将它用在表达式中：

```
fn some_condition() -> bool {
    true
}

// no logging
// 没有日志
fn example() {
    if some_condition() {
        // some code
    }
}

// println! logging
// 使用 println! 打印日志
fn example_println() {
    //
    let result = some_condition();
    println!("{}: result", result); // just prints "true"
                                // 仅仅打印 "true"
    if result {
        // some code
    }
}

// dbg! logging
// 使用 dbg! 打印日志
fn example_dbg() {
    //
    if dbg!(some_condition()) { // prints "[src/main.rs:22] some_condition() = true"
        // 太棒了！打印出丰富的调试信息
        // some code
    }
}
```

The only downside is that `dbg!` isn't automatically stripped in release builds so we have to manually remove it from our code if we don't want to ship it in the final executable.

`dbg!` 宏唯一的缺点是，它不能在构建最终发布的二进制文件时自动删除，我们不得不手动删除相关代码。

算符重载特性 Operator Traits

All operators in Rust are associated with traits. If we'd like to impl operators for our types we have to impl the associated traits.

在 Rust 中，所有的算符都与相应的特性相关联。为特定类型实现相应特性，即为该类型实现了相应算符。

| 特性 | 类别 | 算符 | 描述 |
|---|----|--|--------|
| <code>Eq</code>
, <code>PartialEq</code> | 比较 | <code>==</code> | 相等 |
| <code>Ord</code>
, <code>PartialOrd</code> | 比较 | <code><</code>
, <code>></code>
, <code><=</code>
, <code>>=</code> | 比较 |
| <code>Add</code> | 算数 | <code>+</code> | 加 |
| <code>AddAssign</code> | 算数 | <code>+=</code> | 加等于 |
| <code>BitAnd</code> | 算数 | <code>&</code> | 按位与 |
| <code>BitAndAssign</code> | 算数 | <code>&=</code> | 按位与等于 |
| <code>BitXor</code> | 算数 | <code>^</code> | 按位异或 |
| <code>BitXorAssign</code> | 算数 | <code>^=</code> | 按位异或等于 |
| <code>Div</code> | 算数 | <code>/</code> | 除 |
| <code>DivAssign</code> | 算数 | <code>/=</code> | 除等于 |
| <code>Mul</code> | 算数 | <code>*</code> | 乘 |
| <code>MulAssign</code> | 算数 | <code>*=</code> | 乘等于 |
| <code>Neg</code> | 算数 | <code>-</code> | 一元负 |
| <code>Not</code> | 算数 | <code>!</code> | 一元逻辑非 |

| | | | |
|-------------|----|-----------|---------|
| Rem | 算数 | % | 求余 |
| RemAssign | 算数 | %= | 求余等于 |
| Shl | 算数 | << | 左移 |
| ShlAssign | 算数 | <<= | 左移等于 |
| Shr | 算数 | >> | 右移 |
| ShrAssign | 算数 | >>= | 右移等于 |
| Sub | 算数 | - | 减 |
| SubAssign | 算数 | -= | 减等于 |
| Fn | 闭包 | (...args) | 不可变闭包调用 |
| FnMut | 闭包 | (...args) | 可变闭包调用 |
| FnOnce | 闭包 | (...args) | 一次性闭包调用 |
| Deref | 其它 | * | 不可变解引用 |
| DerefMut | 其它 | * | 可变解引用 |
| Drop | 其它 | - | 类型析构 |
| Index | 其它 | [] | 不可变索引 |
| IndexMut | 其它 | [] | 可变索引 |
| RangeBounds | 其它 | .. | 范围迭代 |

比较特性 Comparison Traits

| 特性 | 类别 | 算符 | 描述 |
|---------------------|----|------------------|----|
| Eq
, PartialEq | 比较 | == | 相等 |
| Ord
, PartialOrd | 比较 | <
, >
, <= | 比较 |

PartialEq & Eq

预备知识

- [Self](#)
- [Methods](#)
- [Generic Parameters](#)
- [Default Impls](#)
- [Generic Blanket Impls](#)
- [Marker Traits](#)
- [Subtraits & Supertraits](#)
- [Sized](#)

```
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;

    // provided default impls
    // 提供默认实现
    fn ne(&self, other: &Rhs) -> bool;
}
```

`PartialEq<Rhs>` 类型可以使用 `==` 算符来检查与 `Rhs` 的相等性。

实现了 `PartialEq<Rhs>` 特性的类型可以使用 `==` 算符来检查与 `Rhs` 的相等性。

All `PartialEq<Rhs>` impls must ensure that equality is symmetric and transitive. That means `>` for all `a`, `b`, and `c`:

- `a == b` implies `b == a` (symmetry)
- `a == b && b == c` implies `a == c` (transitivity)

对 `PartialEq<Rhs>` 的实现须确保实现对称性与传递性。这意味着对于任意 `a` , `b` 和 `c` 有:

- 若 `a == b` 则 `b == a` (对称性)
- 若 `a == b && b == c` 则 `a == c` (传递性)

By default `Rhs = Self` because we almost always want to compare instances of a type to each other, and not to instances of different types. This also automatically guarantees our impl is symmetric and transitive.

默认情况下 `Rhs = Self` 是因为我们几乎总是在相同类型之间进行比较。这也自动地确保了我们的实现是对称的、可传递的。

```
struct Point {  
    x: i32,  
    y: i32  
}  
  
// Rhs == Self == Point  
impl PartialEq for Point {  
    // impl automatically symmetric & transitive  
    // 该实现自动确保了对称性和传递性  
    fn eq(&self, other: &Point) -> bool {  
        self.x == other.x && self.y == other.y  
    }  
}
```

If all the members of a type impl `PartialEq` then it can be derived:

如果特定类型的成员都实现了 `PartialEq` 特性，那么该类型也可衍生该特性:

```
##[derive(PartialEq)]  
struct Point {  
    x: i32,  
    y: i32  
}  
  
##[derive(PartialEq)]  
enum Suit {  
    Spade,  
    Heart,  
    Club,
```

```
Diamond,  
}
```

Once we impl `PartialEq` for our type we also get equality comparisons between references of our type for free thanks to these generic blanket impls:

多亏了一揽子泛型实现，一旦我们为特定类型实现了 `PartialEq` 特性，那么直接使用该类型的引用互相比较也是可以的：

```
// this impl only gives us: Point == Point  
// 该衍生宏本身只允许我们在结构体之间进行比较  
#[derive(PartialEq)]  
struct Point {  
    x: i32,  
    y: i32  
}  
  
// all of the generic blanket impls below  
// are provided by the standard library  
// 以下的一揽子泛型实现由标准库提供  
  
// this impl gives us: &Point == &Point  
// 这个一揽子泛型实现允许我们通过不可变引用之间进行比较  
impl<A, B> PartialEq<&'_ B> for &'_ A  
where A: PartialEq<B> + ?Sized, B: ?Sized;  
  
// this impl gives us: &mut Point == &Point  
// 这个一揽子泛型实现允许我们通过可变引用与不可变引用进行比较  
impl<A, B> PartialEq<&'_ B> for &'_ mut A  
where A: PartialEq<B> + ?Sized, B: ?Sized;  
  
// this impl gives us: &Point == &mut Point  
// 这个一揽子泛型实现允许我们通过不可变引用与可变引用进行比较  
impl<A, B> PartialEq<&'_ mut B> for &'_ A  
where A: PartialEq<B> + ?Sized, B: ?Sized;  
  
// this impl gives us: &mut Point == &mut Point  
// 这个一揽子泛型实现允许我们通过可变引用之间进行比较  
impl<A, B> PartialEq<&'_ mut B> for &'_ mut A  
where A: PartialEq<B> + ?Sized, B: ?Sized;
```

Since this trait is generic we can define equality between different types. The standard library leverages this to allow checking equality between the many string-like types such as `String`, `&str`, `PathBuf`, `&Path`, `OsString`, `&OsStr`, and so on.

由于该特性提供泛型，我们可以定义不同类型之间的可相等性。标准库正是利用这一点提供了不同类型字符串之间的比较功能，例如 `String`, `&str`, `PathBuf`, `&Path`, `OsString` 和 `&OsStr` 等等。

Generally, we should only impl equality between different types *if they contain the same kind of data* and the only difference between the types is how they represent the data or how they allow interacting with the data.

通常来说我们仅会实现相同类型之间的可相等性，除非两种类型虽然包含同一类数据，但又有表达形式或交互形式的差异，这时我们才会考虑实现不同类型之间的可相等性。

Here's a cute but bad example of how someone might be tempted to impl `PartialEq` to check equality between different types that don't meet the above criteria:

以下是一个有趣但糟糕的例子，它尝试为不同类型实现 `PartialEq` 但又违背了上述要求：

```
#[derive(PartialEq)]
enum Suit {
    Spade,
    Club,
    Heart,
    Diamond,
}

#[derive(PartialEq)]
enum Rank {
    Ace,
    Two,
    Three,
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King,
}

#[derive(PartialEq)]
struct Card {
    suit: Suit,
    rank: Rank,
}

// check equality of Card's suit
// 检查花色的相等性
impl PartialEq<Suit> for Card {
    fn eq(&self, other: &Suit) -> bool {
        self.suit == *other
    }
}
```

```

// check equality of Card's rank
// 检查牌序的相等性
impl PartialEq<Rank> for Card {
    fn eq(&self, other: &Rank) -> bool {
        self.rank == *other
    }
}

fn main() {
    let AceOfSpades = Card {
        suit: Suit::Spade,
        rank: Rank::Ace,
    };
    assert!(AceOfSpades == Suit::Spade); //
    assert!(AceOfSpades == Rank::Ace); //
}

```

It works and kinda makes sense. A card which is an Ace of Spades is both an Ace and a Spade, and if we're writing a library to handle playing cards it's reasonable that we'd want to make it easy and convenient to individually check the suit and rank of a card. However, something's missing: symmetry! We can `Card == Suit` and `Card == Rank` but we cannot `Suit == Card` or `Rank == Card` so let's fix that:

上述代码有效且其逻辑有几分道理，黑桃 A 既是黑桃也是 A。但如果我们要去写一个处理扑克牌的库的话，最简单也最方便的方法莫过于独立地检查牌面的花色和牌序。而且，上述代码并不满足对称性！我们可以使用 `Card == Suit` 和 `Card == Rank`，但却不能使用 `Suit == Card` 和 `Rank == Card`，让我们来修复这一点：

```

// check equality of Card's suit
// 检查花色的相等性
impl PartialEq<Suit> for Card {
    fn eq(&self, other: &Suit) -> bool {
        self.suit == *other
    }
}

// added for symmetry
// 增加对称性
impl PartialEq<Card> for Suit {
    fn eq(&self, other: &Card) -> bool {
        *self == other.suit
    }
}

// check equality of Card's rank
// 检查牌序的相等性
impl PartialEq<Rank> for Card {
    fn eq(&self, other: &Rank) -> bool {
        self.rank == *other
    }
}

// added for symmetry

```

```
// 增加对称性
impl PartialEq<Card> for Rank {
    fn eq(&self, other: &Card) -> bool {
        *self == other.rank
    }
}
```

We have symmetry! Great. Adding symmetry just broke transitivity! Oops. This is now possible:

我们实现了对称性！棒！但是实现对称性却破坏了传递性！糟tm大糕！考虑以下代码：

```
fn main() {
    // Ace of Spades
    // A
    let a = Card {
        suit: Suit::Spade,
        rank: Rank::Ace,
    };
    let b = Suit::Spade;
    // King of Spades
    // K
    let c = Card {
        suit: Suit::Spade,
        rank: Rank::King,
    };
    assert!(a == b && b == c); //
    assert!(a == c); //
}
```

A good example of impling `PartialEq` to check equality between different types would be a program that works with distances and uses different types to represent different units of measurement.

关于对不同类型实现 `PartialEq` 特性的绝佳示例如下，本程序的功能在于处理空间上的距离，它使用不同的类型以表示不同的测量单位：

```
#[derive(PartialEq)]
struct Foot(u32);

#[derive(PartialEq)]
struct Yard(u32);

#[derive(PartialEq)]
struct Mile(u32);

impl PartialEq<Mile> for Foot {
    fn eq(&self, other: &Mile) -> bool {
```

```

        self.0 == other.0 * 5280
    }
}

impl PartialEq<Foot> for Mile {
    fn eq(&self, other: &Foot) -> bool {
        self.0 * 5280 == other.0
    }
}

impl PartialEq<Mile> for Yard {
    fn eq(&self, other: &Mile) -> bool {
        self.0 == other.0 * 1760
    }
}

impl PartialEq<Yard> for Mile {
    fn eq(&self, other: &Yard) -> bool {
        self.0 * 1760 == other.0
    }
}

impl PartialEq<Foot> for Yard {
    fn eq(&self, other: &Foot) -> bool {
        self.0 * 3 == other.0
    }
}

impl PartialEq<Yard> for Foot {
    fn eq(&self, other: &Yard) -> bool {
        self.0 == other.0 * 3
    }
}

fn main() {
    let a = Foot(5280);
    let b = Yard(1760);
    let c = Mile(1);

    // symmetry
    // 对称性
    assert!(a == b && b == a); //
    assert!(b == c && c == b); //
    assert!(a == c && c == a); //

    // transitivity
    // 传递性
    assert!(a == b && b == c && a == c); //
    assert!(c == b && b == a && c == a); //
}

```

`Eq` 是一个标记 trait，也是 `PartialEq<Self>` 的子特性。

`Eq` 是仅用于标记的特性，也是 `PartialEq<Self>` 的子特性。

```
trait Eq: PartialEq<Self> {}
```

If we impl `Eq` for a type, on top of the symmetry & transitivity properties required by `PartialEq`, we're also guaranteeing reflexivity, i.e. `a == a` for all `a`. In this sense `Eq` refines `PartialEq` because it represents a stricter version of equality. If all members of a type impl `Eq` then the `Eq` impl can be derived for the type.

鉴于 `PartialEq` 特性提供的对称性与传递性，一旦我们实现 `Eq` 特性，我们也就确保了该类型具有自反性，即对任意 `a` 有 `a == a`。可以说，`Eq` 改良了 `PartialEq`，因为它实现了一个比后者更加严格的可相等性。如果一个类型的全部成员都实现了 `Eq` 特性，那么该类型本身也可以衍生出该特性。

Floats are `PartialEq` but not `Eq` because `Nan != Nan`. Almost all other `PartialEq` types are trivially `Eq`, unless of course if they contain floats.

所有的浮点类型都实现了 `PartialEq` 但是没有实现 `Eq`，因为 `Nan != Nan`。几乎所有其它实现 `PartialEq` 的类型也都自然地实现了 `Eq`，除非它们包含了浮点数。

Once a type implements `PartialEq` and `Debug` we can use it in the `assert_eq!` macro. We can also compare collections of `PartialEq` types.

对于实现了 `PartialEq` 和 `Debug` 的类型，我们也可以将它用于 `assert_eq!` 宏。并且，我们可以对实现 `PartialEq` 特性的类型组成的集合进行比较。

```
#[derive(PartialEq, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn example_assert(p1: Point, p2: Point) {
    assert_eq!(p1, p2);
}

fn example_compare_collections<T: PartialEq>(vec1: Vec<T>, vec2: Vec<T>) {
    // if T: PartialEq this now works!
    if vec1 == vec2 {
        // some code
    } else {
        // other code
    }
}
```

Hash

预备知识

- [Self](#)
- [Methods](#)
- [Generic Parameters](#)
- [Default Impls](#)
- [Derive Macros](#)
- [PartialEq & Eq](#)

```
trait Hash {  
    fn hash<H: Hasher>(&self, state: &mut H);  
  
    // provided default impls  
    // 提供默认实现  
    fn hash_slice<H: Hasher>(data: &[Self], state: &mut H);  
}
```

This trait is not associated with any operator, but the best time to talk about it is right after `PartialEq` & `Eq` so here it is. `Hash` types can be hashed using a `Hasher`.

本特性并未关联到任何算符，之所以在这里提及，是因为它与 `PartialEq` 与 `Eq` 密切的关系。实现 `Hash` 特性的类型可以通过 `Hasher` 作哈希运算。

```
use std::hash::Hasher;  
use std::hash::Hash;  
  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl Hash for Point {  
    fn hash<H: Hasher>(&self, hasher: &mut H) {  
        hasher.write_i32(self.x);  
        hasher.write_i32(self.y);  
    }  
}
```

| There's a derive macro which generates the same impl as above:

以下衍生宏展开与以上代码中相同的实现：

```
##[derive(Hash)]
struct Point {
    x: i32,
    y: i32,
}
```

| If a type implements both `Hash` and `Eq`, those implementations must agree with each other such that for all `a` and `b` if `a == b` then `a.hash() == b.hash()`. So we should always use the derive macro to implement both or manually implement both, but not mix the two, otherwise we risk breaking the above invariant.

如果一个类型同时实现了 `Hash` 和 `Eq`，那么二者必须要实现步调一致，即对任意 `a` 与 `b`，若有 `a == b`，则必有 `a.hash() == b.hash()`。所以，对于同时实现二者，要么都用衍生宏，要么都手动实现，不要一个用衍生宏，而另一个手动实现，否则我们将冒着步调不一致的极大风险。

| The main benefit of implementing `Eq` and `Hash` for a type is that it allows us to store that type as keys in `HashMap`s and `HashSet`s.

实现 `Eq` 和 `Hash` 特性的主要好处在于，这允许我们将该类型作为一个键存储于 `HashMap` 和 `HashSet` 中。

```
use std::collections::HashSet;

// now our type can be stored
// in HashSets and HashMaps!
// 现在我们的类型可以存储于 HashSet 和 HashMap 中了!
#[derive(PartialEq, Eq, Hash)]
struct Point {
    x: i32,
    y: i32,
}

fn example_hashset() {
    let mut points = HashSet::new();
    points.insert(Point { x: 0, y: 0 });
}
```

PartialOrd & Ord

预备知识

- [Self](#)
- [Methods](#)
- [Generic Parameters](#)
- [Default Impls](#)
- [Subtraits & Supertraits](#)
- [Derive Macros](#)
- [Sized](#)
- [PartialEq & Eq](#)

```
enum Ordering {
    Less,
    Equal,
    Greater,
}

trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    // provided default impls
    // 提供默认实现
    fn lt(&self, other: &Rhs) -> bool;
    fn le(&self, other: &Rhs) -> bool;
    fn gt(&self, other: &Rhs) -> bool;
    fn ge(&self, other: &Rhs) -> bool;
}
```

PartialOrd<Rhs> types can be compared to Rhs types using the <, <= , > , and >= operators.

实现 PartialOrd<Rhs> 的类型可以和 Rhs 的类型之间使用 < , <= , > , 和 >= 算符。

All PartialOrd impls must ensure that comparisons are asymmetric and transitive. That means for all a , b , and c :

- a < b implies !(a > b) (asymmetry)
- a < b && b < c implies a < c (transitivity)

实现 `PartialOrd` 时须确保比较的非对称性和传递性。这意味着对任意 `a`, `b`, `c` 有:

- 若 `a < b` 则 `!(a > b)` (非对称性)
- 若 `a < b && b < c` 则 `a < c` (传递性)

`PartialOrd` is a subtrait of `PartialEq` and their impls must always agree with each other.

`PartialOrd` 是 `PartialEq` 的子特性，二者必须要实现步调一致。

```
fn must_always_agree<T: PartialOrd + PartialEq>(t1: T, t2: T) {  
    assert_eq!(t1.partial_cmp(&t2) == Some(Ordering::Equal), t1 == t2);  
}
```

`PartialOrd` refines `PartialEq` in the sense that when comparing `PartialEq` types we can check if they are equal or not equal, but when comparing `PartialOrd` types we can check if they are equal or not equal, and if they are not equal we can check if they are unequal because the first item is less than or greater than the second item.

`PartialOrd` 改良了 `PartialEq`，后者仅能比较是否相等，而前者除了能比较是否相等，还能比较孰大孰小。

By default `Rhs = Self` because we almost always want to compare instances of a type to each other, and not to instances of different types. This also automatically guarantees our impl is symmetric and transitive.

默认情况下 `Rhs = Self`，因为我们几乎总是在相同类型的实例之间相比较，而不是不同类型之间。这一点自动保证了我们的实现的对称性和传递性。

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

```
#[derive(PartialEq, PartialOrd)]  
struct Point {  
    x: i32,  
    y: i32  
}
```

```
// Rhs == Self == Point  
impl PartialOrd for Point {
```

```
// impl automatically symmetric & transitive
// 该实现自动确保了对称性与传递性
fn partial_cmp(&self, other: &Point) -> Option<Ordering> {
    Some(match self.x.cmp(&other.x) {
        Ordering::Equal => self.y.cmp(&other.y),
        ordering => ordering,
    })
}
```

If all the members of a type impl `PartialOrd` then it can be derived:

如果特定类型的全部成员都实现了 `PartialOrd` 特性，那么该类型也可以衍生出该特性：

```
##[derive(PartialEq, PartialOrd)]
struct Point {
    x: i32,
    y: i32,
}

##[derive(PartialEq, PartialOrd)]
enum Stoplight {
    Red,
    Yellow,
    Green,
}
```

The `PartialOrd` derive macro orders types based on the lexicographical order of their members:

`PartialOrd` 衍生宏依据 **类型成员的定义顺序** 对类型进行排序：

```
// generates PartialOrd impl which orders
// Points based on x member first and
// y member second because that's the order
// they appear in the source code
// 宏展开的 PartialOrd 实现排序时
// 首先考虑 x 再考虑 y
// 因为这是它们在源代码中出现的顺序
##[derive(PartialOrd, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

// generates DIFFERENT PartialOrd impl
// which orders Points based on y member
```

```
// first and x member second
// 这里宏展开的 PartialOrd 实现排序时
// 首先考虑 y 再考虑 x
#[derive(PartialOrd, PartialEq)]
struct Point {
    y: i32,
    x: i32,
}
```

`Ord` is a subtrait of `Eq` and `PartialOrd<Self>` :

`Ord` 是 `Eq` 和 `PartialOrd<Self>` 的子特性:

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;

    // provided default impls
    // 提供默认实现
    fn max(self, other: Self) -> Self;
    fn min(self, other: Self) -> Self;
    fn clamp(self, min: Self, max: Self) -> Self;
}
```

If we impl `Ord` for a type, on top of the asymmetry & transitivity properties required by `PartialOrd`, we're also guaranteeing that the asymmetry is total, i.e. exactly one of `a < b`, `a == b` or `a > b` is true for any given `a` and `b`. In this sense `Ord` refines `Eq` and `PartialOrd` because it represents a stricter version of comparisons. If a type impls `Ord` we can use that impl to trivially impl `PartialOrd`, `PartialEq`, and `Eq`:

鉴于 `PartialOrd` 提供的非对称性和传递性, 对特定类型实现 `Ord` 特性的同时也就保证了其非对称性, 即对于任意 `a` 与 `b` 有 `a < b`, `a == b`, `a > b`。可以说, `Ord` 改良了 `Eq` 和 `PartialOrd`, 因为它提供了一种更加严格的比较。如果一个类型实现了 `Ord`, 那么 `PartialOrd`, `PartialEq` 和 `Eq` 的实现也就微不足道了。

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

// of course we can use the derive macros here
// 可以使用衍生宏
#[derive(Ord, PartialOrd, Eq, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

// note: as with PartialOrd, the Ord derive macro
// orders a type based on the lexicographical order
// of its members
// 注意: 与 PartialOrd 相同, Ord 衍生宏衍生宏依据
```

```

// 类型的成员的定义顺序 对类型进行排序

// but here's the impls if we wrote them out by hand
// 以下是我们手动的实现
impl Ord for Point {
    fn cmp(&self, other: &Self) -> Ordering {
        match self.x.cmp(&other.x) {
            Ordering::Equal => self.y.cmp(&other.y),
            ordering => ordering,
        }
    }
}

impl PartialOrd for Point {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl PartialEq for Point {
    fn eq(&self, other: &Self) -> bool {
        self.cmp(other) == Ordering::Equal
    }
}

impl Eq for Point {}

```

Floating-point numbers implement `PartialOrd` but not `Ord` because both `NaN < 0 == false` and `NaN >= 0 == false` are simultaneously true. Almost all other `PartialOrd` types are trivially `Ord`, unless of course if they contain floats.

浮点数类型实现了 `PartialOrd` 但是没有实现 `Ord`，因为 `NaN < 0 == false` 与 `NaN >= 0 == false` 同时为真。几乎所有其它实现 `PartialOrd` 的类型都实现了 `Ord`，除非该类型包含浮点数。

Once a type implements `Ord` we can store it in `BTreeMap`s and `BTreeSet`s as well as easily sort it using the `sort()` method on slices and any types which deref to slices such as arrays, `Vec`s, and `VecDeque`s.

对于实现了 `Ord` 特性的类型，我们可以将它存储于 `BTreeMap` 和 `BTreeSet`，并且可以通过 `sort()` 方法对切片，或者任何可以自动解引用为切片的类型进行排序，例如 `Vec` 和 `VecDeque`。

```

use std::collections::BTreeSet;

// now our type can be stored
// in BTreeSets and BTreeMaps!
// 现在我们的类型可以存储于 BTreeSet 和 BTreeMap 中了!
#[derive(Ord, PartialOrd, PartialEq, Eq)]
struct Point {
    x: i32,
    y: i32,
}

```

```

fn example_btreeset() {
    let mut points = BTreeSet::new();
    points.insert(Point { x: 0, y: 0 });
}

// we can also .sort() Ord types in collections!
// 对于实现了 Ord 特性的类型，我们可以使用 .sort() 方法来对集合进行排序！
fn example_sort<T: Ord>(mut sortable: Vec<T>) -> Vec<T> {
    sortable.sort();
    sortable
}

```

算术特性 Arithmetic Traits

| 特性 | 类别 | 算符 | 描述 |
|--------------|----|-----|--------|
| Add | 算数 | + | 加 |
| AddAssign | 算数 | += | 加等于 |
| BitAnd | 算数 | & | 按位与 |
| BitAndAssign | 算数 | &= | 按位与等于 |
| BitXor | 算数 | ^ | 按位异或 |
| BitXorAssign | 算数 | ^= | 按位异或等于 |
| Div | 算数 | / | 除 |
| DivAssign | 算数 | /= | 除等于 |
| Mul | 算数 | * | 乘 |
| MulAssign | 算数 | *= | 乘等于 |
| Neg | 算数 | - | 一元负 |
| Not | 算数 | ! | 一元逻辑非 |
| Rem | 算数 | % | 求余 |
| RemAssign | 算数 | %= | 求余等于 |
| Shl | 算数 | << | 左移 |
| ShlAssign | 算数 | <<= | 左移等于 |

| | | | |
|-----------|----|-----|------|
| Shr | 算数 | >> | 右移 |
| ShrAssign | 算数 | >>= | 右移等于 |
| Sub | 算数 | - | 减 |
| SubAssign | 算数 | -= | 减等于 |

Going over all of these would be very redundant. Most of these only apply to number types anyway. We'll only go over `Add` and `AddAssign` since the `+` operator is commonly overloaded to do other stuff like adding items to collections or concatenating things together, that way we cover the most interesting ground and don't repeat ourselves.

详解以上所有算术特性未免显得多余，且其大多仅用于操作数字类型。本文仅就最常见被重载的 `Add` 和 `AddAssign` 特性，亦即 `+` 和 `+=` 算符，进行说明，其重载广泛用于为集合增加内容或对不同事物的连接。这样，我们多侧重于最有趣的地方，而不是无趣枯燥地重复。

Add & AddAssign

预备知识

- [Self](#)
- [Methods](#)
- [Associated Types](#)
- [Generic Parameters](#)
- [Generic Types vs Associated Types](#)
- [Derive Macros](#)

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

`Add<Rhs, Output = T>` types can be added to `Rhs` types and will produce `T` as output.

实现 `Add<Rhs, Output = T>` 特性的类型，与 `Rhs` 类型相加得到 `T` 类型的值。

Example `Add<Point, Output = Point>` impl for `Point` :

下例对 `Point` 类型实现了 `Add<Rhs, Output = T>` :

```
#[derive(Clone, Copy)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;
    fn add(self, rhs: Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = p1 + p2;
    assert_eq!(p3.x, p1.x + p2.x); //
    assert_eq!(p3.y, p1.y + p2.y); //
}
```

But what if we only had references to `Point`'s? Can we still add them then? Let's try:

如果我们对 `Point` 的引用进行如上操作还能将他们加在一起吗？我们试试：

```
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = &p1 + &p2; //
}
```

Unfortunately not. The compiler throws:

遗憾的是，并不可以。编译器出错了：

```
error[E0369]: cannot add `&Point` to `&Point`
--> src/main.rs:50:25
|
50 |     let p3: Point = &p1 + &p2;
|           ^---- &Point
|           |
|           &Point
|
= note: an implementation of `std::ops::Add` might be missing for `&Point`
```

Within Rust's type system, for some type `T`, the types `T`, `&T`, and `&mut T` are all treated as unique distinct types which means we have to provide trait impls for each of them separately. Let's define an `Add` impl for `&Point`:

在 Rust 的类型系统中，对于特定类型 `T` 来讲，`T`，`&T`，`&mut T` 三者本身是具有不同类型的，这意味着我们需要对它们分别实现相应特性。下面我们对 `&Point` 实现 `Add` 特性：

```
impl Add for &Point {
    type Output = Point;
    fn add(self, rhs: &Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = &p1 + &p2; // assert_eq!(p3.x, p1.x + p2.x); //
    assert_eq!(p3.y, p1.y + p2.y); //
}
```

However, something still doesn't feel quite right. We have two separate impls of `Add` for `Point` and `&Point` and they *happen* to do the same thing currently but there's no guarantee that they will in the future! For example, let's say we decide that when we add two `Point`s together we want to create a `Line` containing those two `Point`s instead of creating a new `Point`, we'd update our `Add` impl like this:

这是可行的，但是不觉得哪里怪怪的吗？我们对 `Point` 和 `&Point` 分别实现了 `Add` 特性，现在来看这两种实现能够保持步调一致，但是未来也能保证吗？例如，我们现在决定对两个 `Point` 相加要产生一个 `Line` 而不是 `Point`，可以对 `Add` 特性的实现做出如下改动：

```

use std::ops::Add;

#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

#[derive(Copy, Clone)]
struct Line {
    start: Point,
    end: Point,
}

// we updated this impl
// 我们更新了这个实现
impl Add for Point {
    type Output = Line;
    fn add(self, rhs: Point) -> Line {
        Line {
            start: self,
            end: rhs,
        }
    }
}

// but forgot to update this impl, uh oh!
// 但是忘记了更新这个实现，糟tm大糕！
impl Add for &Point {
    type Output = Point;
    fn add(self, rhs: &Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = p1 + p2; // 

    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = &p1 + &p2; // expected Line, found Point
                                // 期待得到 Line，但是得到 Point
}

```

Our current impl of `Add` for `&Point` creates an unnecessary maintenance burden, we want the impl to match `Point`'s impl without having to manually update it every time we change `Point`'s impl. We'd like to keep our code as DRY (Don't Repeat Yourself) as possible. Luckily this is achievable:

我们对 `&Point` 不可变引用类型的 `Add` 实现，给我们带来了不必要的维护困难。是否能够使得，当我们更改 `Point` 类型的实现时，`&Point` 类型的实现也能够自动发生匹配，而不需要我们手动维护呢？我们的愿望是尽可能写出 DRY (Don't Repeat Yourself) 的不重复的代码。幸运的是，我们可以如此实现这一点：

```
// updated, DRY impl
// 使用一种更“干”的实现
impl Add for &Point {
    type Output = <Point as Add>::Output;
    fn add(self, rhs: &Point) -> Self::Output {
        Point::add(*self, *rhs)
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = p1 + p2; //

    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = &p1 + &p2; //
}
```

`AddAssign<Rhs>` types allow us to add + assign `Rhs` types to them. The trait declaration:

实现 `AddAssign<Rhs>` 的类型，允许我们对 `Rhs` 的类型相加之并赋值到自身。该特性的声明为：

```
trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}
```

Example impls for `Point` and `&Point` :

对 `Point` 和 `&Point` 类型的实现示例：

```
use std::ops::AddAssign;
```

```
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32
}
```

```
impl AddAssign for Point {
```

```

fn add_assign(&mut self, rhs: Point) {
    self.x += rhs.x;
    self.y += rhs.y;
}

impl AddAssign<&Point> for Point {
    fn add_assign(&mut self, rhs: &Point) {
        Point::add_assign(self, *rhs);
    }
}

fn main() {
    let mut p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    p1 += &p2;
    p1 += p2;
    assert!(p1.x == 7 && p1.y == 10);
}

```

闭包特性 Closure Traits

| 特性 | 类别 | 算符 | 描述 |
|--------|----|-----------|---------|
| Fn | 闭包 | (...args) | 不可变闭包调用 |
| FnMut | 闭包 | (...args) | 可变闭包调用 |
| FnOnce | 闭包 | (...args) | 一次性闭包调用 |

FnOnce, FnMut, & Fn

预备知识

- [Self](#)
- [Methods](#)
- [Associated Types](#)
- [Generic Parameters](#)
- [Generic Types vs Associated Types](#)
- [Subtraits & Supertraits](#)

```

trait FnOnce<Args> {
    type Output;
}

```

```
fn call_once(self, args: Args) -> Self::Output;  
}  
  
trait FnMut<Args>: FnOnce<Args> {  
    fn call_mut(&mut self, args: Args) -> Self::Output;  
}  
  
trait Fn<Args>: FnMut<Args> {  
    fn call(&self, args: Args) -> Self::Output;  
}
```

Although these traits exist it's not possible to impl them for our own types in stable Rust. The only types we can create which impl these traits are closures. Depending on what the closure captures from its environment determines whether it impls `FnOnce` , `FnMut` , or `Fn` .

事实上，在 stable Rust 中我们并不能对我们自己的类型实现上述特性，唯一的例外是闭包。对于闭包从环境中捕获的不同，该闭包会实现不同的特性：`FnOnce` , `FnMut` , `Fn` 。

An `FnOnce` closure can only be called once because it consumes some value as part of its execution:

对于实现 `FnOnce` 的闭包，仅可调用一次，因为它消耗掉了其执行中必须的值：

```
fn main() {  
    let range = 0..10;  
    let get_range_count = || range.count();  
    assert_eq!(get_range_count(), 10); //  
    get_range_count(); //  
}
```

The `.count()` method on iterators consumes the iterator so it can only be called once. Hence our closure can only be called once. Which is why when we try to call it a second time we get this error:

迭代器上的 `.count()` 方法会消耗掉整个迭代器，所以该方法仅能调用一次。所以我们的闭包也就是能调用一次了，这就是为什么当第二次调用该闭包时会出错：

```
error[E0382]: use of moved value: `get_range_count`  
--> src/main.rs:5:5  
|  
4 |     assert_eq!(get_range_count(), 10);  
|     ----- `get_range_count` moved due to this call  
5 |     get_range_count();  
|     ^^^^^^^^^^^^^^ value used here after move  
|
```

```
note: closure cannot be invoked more than once because it moves the variable `range` out of its environment
--> src/main.rs:3:30
|
3 |     let get_range_count = || range.count();
|           ^^^^^^
note: this value implements `FnOnce`, which causes it to be moved when called
--> src/main.rs:4:16
|
4 |     assert_eq!(get_range_count(), 10);
|           ^^^^^^^^^^^^^^^^^^
```

An `FnMut` closure can be called multiple times and can also mutate variables it has captured from its environment. We might say `FnMut` closures perform side-effects or are stateful. Here's an example of a closure that filters out all non-ascending values from an iterator by keeping track of the smallest value it has seen so far:

对于实现 `FnMut` 特性的闭包，我们可以多次调用，且其可以改变其从环境捕获的值。我们可以说实现 `FnMut` 的闭包的执行具有副作用，或者说它是具有状态的。下例展示了一个闭包，它通过跟踪最小值，来找到一个迭代器中所有非升序的值：

```
fn main() {
    let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];
    let mut min = i32::MIN;
    let ascending = nums.into_iter().filter(|&n| {
        if n <= min {
            false
        } else {
            min = n;
            true
        }
    }).collect::<Vec<_>>();
    assert_eq!(vec![0, 4, 8, 10, 15, 18], ascending); //
}
```

`FnMut` refines `FnOnce` in the sense that `FnOnce` requires taking ownership of its arguments and can only be called once, but `FnMut` requires only taking mutable references and can be called multiple times. `FnMut` can be used anywhere `FnOnce` can be used.

`FnMut` 改良了 `FnOnce`，`FnOnce` 需要接管参数的属权因此只能调用一次，而 `FnMut` 只需要参数的可变引用即可并可调用多次。`FnMut` 可以在所有 `FnOnce` 可用的地方使用。

An `Fn` closure can be called multiple times and does not mutate any variables it has captured from its environment. We might say `Fn` closures have no side-effects or are stateless. Here's an example closure that filters out all values less than some stack variable it captures from its environment from an iterator:

对于实现 `Fn` 特性的闭包，我们可以调用多次，且其不改变任何从环境中捕获的变量。我们可以说实现 `Fn` 的闭包的执行不具有副作用，或者说它是不具有状态的。下例展示了一个闭包，它通过与栈上的值进行比较，过滤掉一个迭代器中所有比它小的值：

```
fn main() {  
    let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];  
    let min = 9;  
    let greater_than_9 = nums.into_iter().filter(|&n| n > min).collect::<Vec<_>>();  
    assert_eq!(vec![10, 15, 18, 13], greater_than_9); //  
}
```

`Fn` refines `FnMut` in the sense that `FnMut` requires mutable references and can be called multiple times, but `Fn` only requires immutable references and can be called multiple times. `Fn` can be used anywhere `FnMut` can be used, which includes anywhere `FnOnce` can be used.

`Fn` 改良了 `FnMut`，尽管它们都可以多次调用，但是 `FnMut` 需要参数的可变引用，而 `Fn` 仅需要参数的不可变引用。`Fn` 可以在所有 `FnMut` 和 `FnOnce` 可用的地方使用。

If a closure doesn't capture anything from its environment it's technically not a closure, but just an anonymously declared inline function, and can be casted to, used, and passed around as a regular function pointer, i.e. `fn`. Function pointers can be used anywhere `Fn` can be used, which includes anywhere `FnMut` and `FnOnce` can be used.

如果一个闭包不从环境中捕获任何的值，那么从技术上讲它就不是闭包，而仅仅只是一个内联的匿名函数。并且它可以被转换为、用于或传递为一个常规函数指针，即 `fn`。函数指针可以用于任何 `Fn`，`FnMut`，`FnOnce` 可用的地方。

```
fn add_one(x: i32) -> i32 {  
    x + 1  
}  
  
fn main() {  
    let mut fn_ptr: fn(i32) -> i32 = add_one;  
    assert_eq!(fn_ptr(1), 2); //  
  
    // capture-less closure cast to fn pointer  
    // 不捕获环境的闭包可转换为普通函数指针  
    fn_ptr = |x| x + 1; // same as add_one  
    assert_eq!(fn_ptr(1), 2); //  
}
```

Example of passing a regular function pointer in place of a closure:

以下示例中，将常规函数作为闭包而传入：

```
fn main() {
    let nums = vec![-1, 1, -2, 2, -3, 3];
    let absolutes: Vec<i32> = nums.into_iter().map(i32::abs).collect();
    assert_eq!(vec![1, 1, 2, 2, 3, 3], absolutes); // 
}
```

其它特性 Other Traits

| 特性 | 类别 | 算符 | 描述 |
|-------------|----|----|--------|
| Deref | 其它 | * | 不可变解引用 |
| DerefMut | 其它 | * | 可变解引用 |
| Drop | 其它 | - | 类型析构 |
| Index | 其它 | [] | 不可变索引 |
| IndexMut | 其它 | [] | 可变索引 |
| RangeBounds | 其它 | .. | 范围迭代 |

Deref & DerefMut

预备知识

- [Self](#)
- [Methods](#)
- [Associated Types](#)
- [Subtraits & Supertraits](#)
- [Sized](#)

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

```
trait DerefMut: Deref {
```

```
fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

Deref<Target = T> types can dereferenced to T types using the dereference operator * . This has obvious use-cases for smart pointer types like Box and Rc . However, we rarely see the dereference operator explicitly used in Rust code, and that's because of a Rust feature called *deref coercion*.

实现 Deref<Target = T> 的类型，可以通过 * 解引用算符，解引用到 T 类型。智能指针是该特性的著名实现者，例如 Box 和 Rc 。不过，我们很少在 Rust 编程中看到解引用算符，这是由于 Rust 的强制解引用的特性所导致的。

Rust automatically dereferences types when they're being passed as function arguments, returned from a function, or used as part of a method call. This is the reason why we can pass &String and &Vec<T> to functions expecting &str and &[T] because String implements Deref<Target = str> and Vec<T> implements Deref<Target = [T]> .

当作为函数的参数、函数的返回值、方法的调用参数时，Rust 会自动地解引用。这就是为什么我们可以将 &String 或 &Vec<T> 类型的值作为参数传递给接受 str 或 &[T] 类型的参数的函数，因为 String 实现了 Deref<Target = str> ， Vec<T> 实现了 Deref<Target = [T]> 。

Deref and DerefMut should only be implemented for smart pointer types. The most common way people attempt to misuse and abuse these traits is to try to shoehorn some kind of OOP-style data inheritance into Rust. This does not work. Rust is not OOP. Let's examine a few different situations where, how, and why it does not work. Let's start with this example:

Deref 和 DerefMut 仅应实现于智能指针类型。最常见的误用或滥用就是，人们经常希望强行把某种面向对象编程风格的数据继承塞到 Rust 编程中。这是不可能的，因为 Rust 不是面向对象的。让我们用一个例子来领会到底为什么这是不可以的：

```
use std::ops::Deref;  
  
struct Human {  
    health_points: u32,  
}  
  
enum Weapon {  
    Spear,  
    Axe,  
    Sword,  
}  
  
// a Soldier is just a Human with a Weapon  
// 士兵是手持武器的人类  
struct Soldier {  
    human: Human,
```

```
    weapon: Weapon,
}

impl Deref for Soldier {
    type Target = Human;
    fn deref(&self) -> &Human {
        &self.human
    }
}

enum Mount {
    Horse,
    Donkey,
    Cow,
}
// a Knight is just a Soldier with a Mount
// 骑士是跨骑坐骑的士兵
struct Knight {
    soldier: Soldier,
    mount: Mount,
}

impl Deref for Knight {
    type Target = Soldier;
    fn deref(&self) -> &Soldier {
        &self.soldier
    }
}

enum Spell {
    MagicMissile,
    FireBolt,
    ThornWhip,
}
// a Mage is just a Human who can cast Spells
// 法师是口诵咒语的人类
struct Mage {
    human: Human,
    spells: Vec<Spell>,
}

impl Deref for Mage {
    type Target = Human;
    fn deref(&self) -> &Human {
        &self.human
    }
}

enum Staff {
    Wooden,
    Metallic,
    Plastic,
}
// a Wizard is just a Mage with a Staff
// 巫师是腰别法宝的法师
```

```

struct Wizard {
    mage: Mage,
    staff: Staff,
}

impl Deref for Wizard {
    type Target = Mage;
    fn deref(&self) -> &Mage {
        &self.mage
    }
}

fn borrows_human(human: &Human) {}
fn borrows_soldier(soldier: &Soldier) {}
fn borrows_knight(knight: &Knight) {}
fn borrows_mage(mage: &Mage) {}
fn borrows_wizard(wizard: &Wizard) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types can be used as Humans
    borrows_human(&human);
    borrows_human(&soldier);
    borrows_human(&knight);
    borrows_human(&mage);
    borrows_human(&wizard);
    // Knights can be used as Soldiers
    borrows_soldier(&soldier);
    borrows_soldier(&knight);
    // Wizards can be used as Mages
    borrows_mage(&mage);
    borrows_mage(&wizard);
    // Knights & Wizards passed as themselves
    borrows_knight(&knight);
    borrows_wizard(&wizard);
}

```

So at first glance the above looks pretty good! However it quickly breaks down to scrutiny. First of all, deref coercion only works on references, so it doesn't work when we actually want to pass ownership:

事实上，并不可以这么做。首先，强制解引用仅用于引用，所以我们不能移交属权：

```

fn takes_human(human: Human) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types CANNOT be used as Humans
    takes_human(human);
    takes_human(soldier); //
    takes_human(knight); //
    takes_human(mage); //
    takes_human(wizard); //
}

```

Furthermore, deref coercion doesn't work in generic contexts. Let's say we impl some trait only on humans:

其次，强制解引用不可用于泛型编程。例如某特性仅对人类实现：

```
trait Rest {  
    fn rest(&self);  
}  
  
impl Rest for Human {  
    fn rest(&self) {}  
}  
  
fn take_rest<T: Rest>(rester: &T) {  
    rester.rest()  
}  
  
fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {  
    // all types CANNOT be used as Rest types, only Human  
    take_rest(&human);  
    take_rest(&soldier); //  
    take_rest(&knight); //  
    take_rest(&mage); //  
    take_rest(&wizard); //  
}
```

Also, although deref coercion works in a lot of places it doesn't work everywhere. It doesn't work on operands, even though operators are just syntax sugar for method calls. Let's say, to be cute, we wanted `Mage`s to learn `Spell`s using the `+=` operator:

强制解引用可以用于许多情况，但绝不是所有情况。例如对于算符的操作数而言就不行，即便算符仅是一种方法调用的语法糖。比如，我们希望使用 `+=` 算符来表达法师学习咒语。

```
impl DerefMut for Wizard {  
    fn deref_mut(&mut self) -> &mut Mage {  
        &mut self.mage  
    }  
}  
  
impl AddAssign<Spell> for Mage {  
    fn add_assign(&mut self, spell: Spell) {  
        self.spells.push(spell);  
    }  
}  
  
fn example(mut mage: Mage, mut wizard: Wizard, spell: Spell) {  
    mage += spell;
```

```

wizard += spell; // wizard not coerced to mage here
    // 在这里，巫师不能强制转换为法师
wizard.add_assign(spell); // oof, we have to call it like this
    // 所以，我们必须要这样做
}

```

In languages with OOP-style data inheritance the value of `self` within a method is always equal to the type which called the method but in the case of Rust the value of `self` is always equal to the type which implemented the method:

在带有面向对象风格的数据继承的语言中，方法中的 `self` 值的类型总是等同于调用该方法的类型。但是在 Rust 语言中，`self` 值的类型总是等同于实现该方法时的类型。

```

struct Human {
    profession: &'static str,
    health_points: u32,
}

impl Human {
    // self will always be a Human here, even if we call it on a Soldier
    // 该方法中的 self 的类型永远是 Human，即便我们在 Soldier 类型上调用
    fn state_profession(&self) {
        println!("I'm a {}!", self.profession);
    }
}

struct Soldier {
    profession: &'static str,
    human: Human,
    weapon: Weapon,
}

fn example(soldier: &Soldier) {
    assert_eq!("servant", soldier.human.profession);
    assert_eq!("spearman", soldier.profession);
    soldier.human.state_profession(); // prints "I'm a servant!"
    soldier.state_profession(); // still prints "I'm a servant!"
}

```

The above gotcha is especially damning when impling `Deref` or `DerefMut` on a newtype. Let's say we want to create a `SortedVec` type which is just a `Vec` but it's always in sorted order. Here's how we might do that:

上述特性常令人感到困惑，特别是在对新类型实现 `Deref` 和 `DerefMut` 的时候。例如我们想要设计一个 `SortedVec` 类型，相比于 `Vec` 类型，它总是处于已排序的状态。我们可能会这样做：

```

struct SortedVec<T: Ord>(Vec<T>);

impl<T: Ord> SortedVec<T> {
    fn new(mut vec: Vec<T>) -> Self {
        vec.sort();
        SortedVec(vec)
    }
    fn push(&mut self, t: T) {
        self.0.push(t);
        self.0.sort();
    }
}

```

Obviously we cannot impl `DerefMut<Target = Vec<T>>` here or anyone using `SortedVec` would be able to trivially break the sorted order. However, impling `Deref<Target = Vec<T>>` surely must be safe, right? Try to spot the bug in the program below:

显然我们不能为其实现 `DerefMut<Target = Vec<T>>`，因为这可能会破坏排序状态。实现 `Deref<Target = Vec<T>>` 必须要保证功能的正确性。尝试指出下列代码中的 bug：

```

use std::ops::Deref;

struct SortedVec<T: Ord>(Vec<T>);

impl<T: Ord> SortedVec<T> {
    fn new(mut vec: Vec<T>) -> Self {
        vec.sort();
        SortedVec(vec)
    }
    fn push(&mut self, t: T) {
        self.0.push(t);
        self.0.sort();
    }
}

impl<T: Ord> Deref for SortedVec<T> {
    type Target = Vec<T>;
    fn deref(&self) -> &Vec<T> {
        &self.0
    }
}

fn main() {
    let sorted = SortedVec::new(vec![2, 8, 6, 3]);
    sorted.push(1);
    let sortedClone = sorted.clone();
    sortedClone.push(4);
}

```

We never implemented `Clone` for `SortedVec` so when we call the `.clone()` method the compiler is using deref coercion to resolve that method call on `Vec` and so it returns a `Vec` and not a `SortedVec`!

鉴于我们从未对 `SortedVec` 实现 `Clone` 特性，所以当我们调用 `.clone()` 方法的时候，编译器会使用强制解引用将该方法调用解析为 `Vec` 的方法调用，所以该方法返回的是 `Vec` 而不是 `SortedVec`！

```
fn main() {  
    let sorted: SortedVec<i32> = SortedVec::new(vec![2, 8, 6, 3]);  
    sorted.push(1); // still sorted  
  
    // calling clone on SortedVec actually returns a Vec  
    let sortedClone: Vec<i32> = sorted.clone();  
    sortedClone.push(4); // sortedClone no longer sorted  
}
```

Anyway, none of the above limitations, constraints, or gotchas are faults of Rust because Rust was never designed to be an OO language or to support any OOP patterns in the first place.

切记，Rust 并非设计为面向对象的语言，也并不将面向对象编程的模式作为一等公民，所以以上的限制、约束和令人困惑的特性并不被认为是在语言中是错误的。

The main takeaway from this section is do not try to be cute or clever with `Deref` and `DerefMut` impls. They're really only appropriate for smart pointer types, which can only be implemented within the standard library for now as smart pointer types currently require unstable features and compiler magic to work. If we want functionality and behavior similar to `Deref` and `DerefMut` then what we're actually probably looking for is `AsRef` and `AsMut` which we'll get to later.

本节的主旨即是使读者领会为什么不要自作聪明地实现 `Deref` 和 `DerefMut` 特性。这类特性确仅适合于智能指针类的类型，目前来讲标准库中的智能指针的实现，确需要这样的不稳定特性以及一些编译器魔法才能工作。如果我们确需要一些类似于 `Deref` 和 `DerefMut` 的特性，不妨使用 `AsRef` 和 `AsMut` 特性。我们将在后面的章节中对这类特性做出说明。

Index & IndexMut

预备知识

- [Self](#)
- [Methods](#)
- [Associated Types](#)

- Generic Parameters
- Generic Types vs Associated Types
- Subtraits & Supertraits
- Sized

```
trait Index<Idx: ?Sized> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> where Idx: ?Sized {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

We can index `[]` into `Index<T, Output = U>` types with `T` values and the index operation will return `&U` values. For syntax sugar, the compiler auto inserts a deref operator `*` in front of any value returned from an index operation:

对于实现 `Index<T, Output = U>` 的类型，我们可以使用 `[]` 索引算符对 `T` 类型的值索引 `&U` 类型的值。作为语法糖，编译器也会为索引操作返回的值自动添加一个 `*` 解引用算符。

```
fn main() {
    // Vec<i32> implements Index<usize, Output = i32> so
    // indexing Vec<i32> should produce &i32s and yet...
    // 鉴于 Vec<i32> 实现了 Index<usize, Output = i32>
    // 所以对 Vec<i32> 的索引应当返回 &i32 类型的值，但是。。
    let vec = vec![1, 2, 3, 4, 5];
    let num_ref: &i32 = vec[0]; // expected &i32 found i32

    // above line actually desugars to
    // 以上代码等价于
    let num_ref: &i32 = *vec[0]; // expected &i32 found i32

    // both of these alternatives work
    // 以下是建议使用的一对形式
    let num: i32 = vec[0]; //
    let num_ref: &i32 = &vec[0]; //
}
```

It's kinda confusing at first, because it seems like the `Index` trait does not follow its own method signature, but really it's just questionable syntax sugar.

令人困惑的是，似乎 `Index` 特性没有遵循它自己的方法签名，但其实真正有问题的是语法糖。

Since `Idx` is a generic type the `Index` trait can be implemented many times for a given type, and in the case of `Vec<T>` not only can we index into it using `usize` but we can also index into its using `Range<usize>`s to get slices.

鉴于 `Idx` 是泛型类型，`Index` 特性对多个给定类型可以多次实现。并且对于 `Vec<T>`，我们不仅可以对 `usize` 索引，还可以对 `Range<usize>` 索引得到切片。

```
fn main() {
    let vec = vec![1, 2, 3, 4, 5];
    assert_eq!(&vec[..], &[1, 2, 3, 4, 5]); //
    assert_eq!(&vec[1..], &[2, 3, 4, 5]); //
    assert_eq!(&vec[..4], &[1, 2, 3, 4]); //
    assert_eq!(&vec[1..4], &[2, 3, 4]); //
}
```

To show off how we might impl `Index` ourselves here's a fun example which shows how we can use a newtype and the `Index` trait to impl wrapping indexes and negative indexes on a `Vec` :

为了展示如何自己实现 `Index` 特性，以下是一个有趣的例子，它设计了一个 `Vec` 的包装结构，其使得循环索引和负数索引成为可能：

```
use std::ops::Index;

struct WrappingIndex<T>(Vec<T>);

impl<T> Index<usize> for WrappingIndex<T> {
    type Output = T;
    fn index(&self, index: usize) -> &T {
        &self.0[index % self.0.len()]
    }
}

impl<T> Index<i128> for WrappingIndex<T> {
    type Output = T;
    fn index(&self, index: i128) -> &T {
        let self_len = self.0.len() as i128;
        let idx = (((index % self_len) + self_len) % self_len) as usize;
        &self.0[idx]
    }
}

#[test] //
fn indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[0_usize]);
    assert_eq!(2, wrapping_vec[1_usize]);
```

```

    assert_eq!(3, wrapping_vec[2_usize]);
}

#[test] //
fn wrapping_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[3_usize]);
    assert_eq!(2, wrapping_vec[4_usize]);
    assert_eq!(3, wrapping_vec[5_usize]);
}

#[test] //
fn neg_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[-3_i128]);
    assert_eq!(2, wrapping_vec[-2_i128]);
    assert_eq!(3, wrapping_vec[-1_i128]);
}

#[test] //
fn wrapping_neg_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[-6_i128]);
    assert_eq!(2, wrapping_vec[-5_i128]);
    assert_eq!(3, wrapping_vec[-4_i128]);
}

```

There's no requirement that the `Idx` type has to be a number type or a `Range`, it could be an enum! Here's an example using basketball positions to index into a basketball team to retrieve players on the team:

`Idx` 的类型并不非得是数字类型或 `Range` 类型，甚至还可以是枚举！例如我们可以在一支篮球队中，对打什么位置索引从而得到队伍里打这个位置的队员：

```

use std::ops::Index;

enum BasketballPosition {
    PointGuard,
    ShootingGuard,
    Center,
    PowerForward,
    SmallForward,
}

struct BasketballPlayer {
    name: &'static str,
    position: BasketballPosition,
}

struct BasketballTeam {
    point_guard: BasketballPlayer,
    shooting_guard: BasketballPlayer,
    center: BasketballPlayer,
}

```

```

    power_forward: BasketballPlayer,
    small_forward: BasketballPlayer,
}

impl Index<BasketballPosition> for BasketballTeam {
    type Output = BasketballPlayer;
    fn index(&self, position: BasketballPosition) -> &BasketballPlayer {
        match position {
            BasketballPosition::PointGuard => &self.point_guard,
            BasketballPosition::ShootingGuard => &self.shooting_guard,
            BasketballPosition::Center => &self.center,
            BasketballPosition::PowerForward => &self.power_forward,
            BasketballPosition::SmallForward => &self.small_forward,
        }
    }
}

```

Drop

预备知识

- [Self](#)
- [Methods](#)

```

trait Drop {
    fn drop(&mut self);
}

```

If a type implements `Drop`, then `drop` will be called on the type when it goes out of scope but before it's destroyed. We will rarely need to implement this for our types but a good example of where it's useful is if a type holds on to some external resources which needs to be cleaned up when the type is destroyed.

对于实现 `Drop` 特性的类型，在该类型脱离作用域并销毁前，其 `drop` 方法会被调用。通常，不必为我们的类型实现这一特性，除非该类型持有某种外部的资源，且该资源需要显式释放。

There's a `BufWriter` type in the standard library that allows us to buffer writes to `Write` types. However, what if the `BufWriter` gets destroyed before the content in its buffer has been flushed to the underlying `Write` type? Thankfully that's not possible! The `BufWriter` implements the `Drop` trait so that `flush` is always called on it whenever it goes out of scope!

标准库中的 `BufWriter` 类型允许我们对向 `Write` 类型写入的时候进行缓存。显然，当 `BufWriter` 销毁前应当把缓存的内容写入 `Writer` 实例，这就是 `Drop` 所允许我们做到的！对于实现了 `Drop` 的 `BufWriter` 来说，其实例在销毁前会总会调用 `flush` 方法。

```
impl<W: Write> Drop for BufWriter<W> {
    fn drop(&mut self) {
        self.flush_buf();
    }
}
```

Also, `Mutex`'s in Rust don't have `unlock()` methods because they don't need them! Calling `lock()` on a `Mutex` returns a `MutexGuard` which automatically unlocks the `Mutex` when it goes out of scope thanks to its `Drop` impl:

并且，在 Rust 中 `Mutex` 类型之所以没有 `unlock()` 方法，就是因为它完全不需要！鉴于 `Drop` 特性的实现，调用 `Mutex` 的 `lock()` 方法返回的 `MutexGuard` 类型，在脱离作用域时会自动地释放 `Mutex`。

```
impl<T: ?Sized> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        unsafe {
            self.lock.inner.raw_unlock();
        }
    }
}
```

In general, if you're impling an abstraction over some resource that needs to be cleaned up after use then that's a great reason to make use of the `Drop` trait.

简而言之，如果你正在设计某种需要显示释放的资源的抽象包装，那么这正是 `Drop` 特性大显神威的地方。

转换特性 Conversion Traits

From & Into

预备知识

- [Self](#)
- [Functions](#)

- [Methods](#)
- [Generic Parameters](#)
- [Generic Blanket Impls](#)

```
trait From<T> {
    fn from(T) -> Self;
}
```

| `From<T>` types allow us to convert `T` into `Self`.

实现 `From<T>` 特性的类型允许我们从 `T` 类型转换到自身的类型 `Self`。

```
trait Into<T> {
    fn into(self) -> T;
}
```

| `Into<T>` types allow us to convert `Self` into `T`.

实现 `Into<T>` 特性的类型允许我们从自身的类型 `Self` 转换到 `T` 类型。

| These traits are two different sides of the same coin. We can only impl `From<T>` for our types because the `Into<T>` impl is automatically provided by this generic blanket impl:

这是一对恰好相反的特性，如同一枚硬币的两面。注意，我们只能手动实现 `From<T>` 特性，而不能手动实现 `Into<T>` 特性，因为 `Into<T>` 特性已经被一揽子泛型实现所自动实现。

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

| The reason both traits exist is because it allows us to write trait bounds on generic types slightly differently:

这两个特性同时存在的一个好处在于，我们可以在为泛型类型添加约束的时候，使用两种稍有不同的记号：

```
fn function<T>(t: T)
where
    // these bounds are equivalent
    // 以下两种记号等价
    T: From<i32>,
    i32: Into<T>
{
    // these examples are equivalent
    // 以下两种记号等价
    let example: T = T::from(0);
    let example: T = 0.into();
}
```

There are no hard rules about when to use one or the other, so go with whatever makes the most sense for each situation. Now let's look at some example impls on `Point` :

对于具体使用哪种记号并无一定之规，请根据实际情况做出最恰当的选择。接下来我们看看 `Point` 类型的例子：

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

impl From<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Self {
        Point { x, y }
    }
}

impl From<[i32; 2]> for Point {
    fn from([x, y]: [i32; 2]) -> Self {
        Point { x, y }
    }
}

fn example() {
    // using From
    let origin = Point::from((0, 0));
    let origin = Point::from([0, 0]);

    // using Into
    let origin: Point = (0, 0).into();
    let origin: Point = [0, 0].into();
}
```

The impl is not symmetric, so if we'd like to convert `Point`'s into tuples and arrays we have to explicitly add those as well:

这样的转换并不是对称的，如果我们想将 `Point` 转换为元组或数组，那么我们需要显式地编写相应的代码：

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl From<(i32, i32)> for Point {  
    fn from((x, y): (i32, i32)) -> Self {  
        Point { x, y }  
    }  
}  
  
impl From<Point> for (i32, i32) {  
    fn from(Point { x, y }: Point) -> Self {  
        (x, y)  
    }  
}  
  
impl From<[i32; 2]> for Point {  
    fn from([x, y]: [i32; 2]) -> Self {  
        Point { x, y }  
    }  
}  
  
impl From<Point> for [i32; 2] {  
    fn from(Point { x, y }: Point) -> Self {  
        [x, y]  
    }  
}  
  
fn example() {  
    // from (i32, i32) into Point  
    let point = Point::from((0, 0));  
    let point: Point = (0, 0).into();  
  
    // from Point into (i32, i32)  
    let tuple = <(i32, i32)>::from(point);  
    let tuple: (i32, i32) = point.into();  
  
    // from [i32; 2] into Point  
    let point = Point::from([0, 0]);  
    let point: Point = [0, 0].into();  
  
    // from Point into [i32; 2]  
    let array = <[i32; 2]>::from(point);  
    let array: [i32; 2] = point.into();  
}
```

A popular use of `From<T>` is to trim down boilerplate code. Let's say we add a `Triangle` type to our program which contains three `Point`s, here's some of the many ways we can construct it:

借由 `From<T>` 特性，我们可以省却大量编写模板代码的麻烦。例如，我们现在具有一个包含三个 `Point` 的类型 `Triangle` 类型，以下是构造该类型的几种办法：

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl Point {  
    fn new(x: i32, y: i32) -> Point {  
        Point { x, y }  
    }  
}  
  
impl From<(i32, i32)> for Point {  
    fn from((x, y): (i32, i32)) -> Point {  
        Point { x, y }  
    }  
}  
  
struct Triangle {  
    p1: Point,  
    p2: Point,  
    p3: Point,  
}  
  
impl Triangle {  
    fn new(p1: Point, p2: Point, p3: Point) -> Triangle {  
        Triangle { p1, p2, p3 }  
    }  
}  
  
impl<P> From<[P; 3]> for Triangle  
where  
    P: Into<Point>  
{  
    fn from([p1, p2, p3]: [P; 3]) -> Triangle {  
        Triangle {  
            p1: p1.into(),  
            p2: p2.into(),  
            p3: p3.into(),  
        }  
    }  
}  
  
fn example() {  
    // manual construction  
    let triangle = Triangle {  
        p1: Point {
```

```

        x: 0,
        y: 0,
    },
    p2: Point {
        x: 1,
        y: 1,
    },
    p3: Point {
        x: 2,
        y: 2,
    },
};

// using Point::new
let triangle = Triangle {
    p1: Point::new(0, 0),
    p2: Point::new(1, 1),
    p3: Point::new(2, 2),
};

// using From<(i32, i32)> for Point
let triangle = Triangle {
    p1: (0, 0).into(),
    p2: (1, 1).into(),
    p3: (2, 2).into(),
};

// using Triangle::new + From<(i32, i32)> for Point
let triangle = Triangle::new(
    (0, 0).into(),
    (1, 1).into(),
    (2, 2).into(),
);

// using From<[Into<Point>; 3]> for Triangle
let triangle: Triangle = [
    (0, 0),
    (1, 1),
    (2, 2),
].into();
}

```

There are no rules for when, how, or why we should impl `From<T>` for our types so it's up to us to use our best judgement for every situation.

对于 `From<T>` 特性的使用并无一定之规，运用你的智慧明智地使用它吧！

One popular use of `Into<T>` is to make functions which need owned values generic over whether they take owned or borrowed values:

使用 `Into<T>` 特性的一个神奇之处在于，对于那些本来只能接受特定类型参数的函数，现在你可以有更多不同的选择：

```
struct Person {  
    name: String,  
}  
  
impl Person {  
    // accepts:  
    // - String  
    fn new1(name: String) -> Person {  
        Person { name }  
    }  
  
    // accepts:  
    // - String  
    // - &String  
    // - &str  
    // - Box<str>  
    // - Cow<'_, str>  
    // - char  
    // since all of the above types can be converted into String  
    fn new2<N: Into<String>>(name: N) -> Person {  
        Person { name: name.into() }  
    }  
}
```

错误处理 Error Handling

The best time to talk about error handling and the `Error` trait is after going over `Display`, `Debug`, `Any`, and `From` but before getting to `TryFrom` hence why the **Error Handling** section awkwardly bisects the **Conversion Traits** section.

讲解错误处理与 `Error` 特性的最佳时机，莫过于在 `Display`, `Debug`, `Any` 和 `From` 之后, `TryFrom` 之前，这就是为什么我要将 **错误处理** 这一节硬塞在 **转换特性** 这一章里。

Error

预备知识

- [Self](#)
- [Methods](#)
- [Default Impls](#)

- [Generic Blanket Impls](#)
- [Subtraits & Supertraits](#)
- [Trait Objects](#)
- [Display & ToString](#)
- [Debug](#)
- [Any](#)
- [From & Into](#)

```
trait Error: Debug + Display {
    // provided default impls
    // 提供默认实现
    fn source(&self) -> Option<&(dyn Error + 'static)>;
    fn backtrace(&self) -> Option<&Backtrace>;
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&dyn Error>;
}
```

In Rust errors are returned, not thrown. Let's look at some examples.

在 Rust 中，错误是被返回的，而不是被抛出的。让我们看看下面的例子：

Since dividing integer types by zero panics if we wanted to make our program safer and more explicit we could impl a `safe_div` function which returns a `Result` instead like this:

由于整数的除零操作会导致 panic，为了程序的健壮性，我们显式地实现了安全的 `safe_div` 除法函数，它的返回值是 `Result`：

```
use std::fmt;
use std::error;

#[derive(Debug, PartialEq)]
struct DivByZero;

impl fmt::Display for DivByZero {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "division by zero error")
    }
}

impl error::Error for DivByZero {}

fn safe_div(numerator: i32, denominator: i32) -> Result<i32, DivByZero> {
```

```

if denominator == 0 {
    return Err(DivByZero);
}
Ok(numerator / denominator)
}

#[test] //
fn test_safe_div() {
    assert_eq!(safe_div(8, 2), Ok(4));
    assert_eq!(safe_div(5, 0), Err(DivByZero));
}

```

Since errors are returned and not thrown they must be explicitly handled, and if the current function cannot handle an error it should propagate it up to the caller. The most idiomatic way to propagate errors is to use the `?` operator, which is just syntax sugar for the now deprecated `try!` macro which simply does this:

由于错误是被返回的，而不是被抛出的，它们必须被显式地处理。如果当前函数没有处理该错误的能力，那么该错误应当原路返回到上一级调用函数。最理想的返回错误的方法是使用 `?` 算符，它是现在已经过时的 `try!` 宏的语法糖：

```

macro_rules! try {
    ($expr:expr) => {
        match $expr {
            // if Ok just unwrap the value
            // 正常情况下直接解除 Result 的包装
            Ok(val) => val,
            // if Err map the err value using From and return
            // 否则将该错误进行适当转换后，返回到上级调用函数
            Err(err) => {
                return Err(From::from(err));
            }
        }
    };
}

```

If we wanted to write a function which reads a file into a `String` we could write it like this, propagating the `io::Error`s using `?` everywhere they can appear:

例如，如果我们的函数其功能是将文件读为一个 `String`，那么使用 `?` 算符来将可能的错误 `io::Error` 返回给上级调用函数就很方便：

```

use std::io::Read;
use std::path::Path;
use std::io;
use std::fs::File;

fn read_file_to_string(path: &Path) -> Result<String, io::Error> {

```

```
let mut file = File::open(path)?; // io::Error
let mut contents = String::new();
file.read_to_string(&mut contents)?; // io::Error
Ok(contents)
}
```

But let's say the file we're reading is actually a list of numbers and we want to sum them together, we'd update our function like this:

又例如，如果我们的文件是一系列数字，我们想将它们加在一起，可以这样编写代码：

```
use std::io::Read;
use std::path::Path;
use std::io;
use std::fs::File;

fn sum_file(path: &Path) -> Result<i32, /* What to put here? */ {
    // 这里填写什么类型好呢?
    let mut file = File::open(path)?; // io::Error
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // io::Error
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?; // ParseIntError
    }
    Ok(sum)
}
```

But what's the error type of our `Result` now? It can return either an `io::Error` or a `ParseIntError`. We're going to look at three approaches for solving this problem, starting with the most quick & dirty way and finishing with the most robust way.

现在 `Result` 的类型又如何？该函数内部可能产生 `io::Error` 或 `ParseIntError` 两种错误。我们将介绍三种解决此类问题的方法，从最简单但不优雅的方法，到最健壮的方法：

The first approach is recognizing that all types which impl `Error` also impl `Display` so we can map all the errors to `String`s and use `String` as our error type:

方法一，我们注意到，所有实现了 `Error` 的类型同时也实现了 `Display`，因此我们可以将错误映射到 `String` 并以此为错误类型：

```
use std::fs::File;
use std::io;
```

```

use std::io::Read;
use std::path::Path;

fn sum_file(path: &Path) -> Result<i32, String> {
    let mut file = File::open(path)
        .map_err(|e| e.to_string())?; // io::Error -> String
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .map_err(|e| e.to_string())?; // io::Error -> String
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()
            .map_err(|e| e.to_string())?; // ParseIntError -> String
    }
    Ok(sum)
}

```

The obvious downside of stringifying every error is that we throw away type information which makes it harder for the caller to handle the errors.

此方法的明显缺点在于，由于我们将所有的错误都序列化了，以至于丢弃了该错误的类型信息，这对于上级调用函数错误处理来讲，就不是那么方便了。

One nonobvious upside to the above approach is we can customize the strings to provide more context-specific information. For example, `ParseIntError` usually stringifies to `"invalid digit found in string"` which is very vague and doesn't mention what the invalid string is or what integer type it was trying to parse into. If we were debugging this problem that error message would almost be useless. However we can make it significantly better by providing all the context relevant information ourselves:

但此方法也有一个不明显的优点，那就是我们可以使用自定义的字符串，来提供丰富的上下文错误信息。例如，`ParseIntError` 通常序列化为 `"invalid digit found in string"` 这样模棱两可的文本，既没有提及无效的字符串是什么，也没有提及它要转换到什么样的数字类型。这样的信息对于我们调试程序来讲几乎没有什么帮助。不过我们可以提供更有意义的，且上下文相关的信息来明显改善这一点：

```

sum += line.parse::<i32>()
    .map_err(|_| format!("failed to parse {} into i32", line))?;

```

The second approach takes advantage of this generic blanket impl from the standard library:

方法二，利用标准库的一揽子泛型实现：

```
impl<E: error::Error> From<E> for Box<dyn error::Error>;
```

Which means that any `Error` type can be implicitly converted into a `Box<dyn error::Error>` by the `?` operator, so we can set to error type to `Box<dyn error::Error>` in the `Result` return type of any function which produces errors and the `?` operator will do the rest of the work for us:

所有实现了 `Error` 特性的类型都可以隐式地使用 `?` 转换为 `Box<dyn error::Error>` 类型。所以我们可以将 `Result` 的错误类型设为 `Box<dyn error::Error>` 类型，然后 `?` 算符会帮我们实现这一隐式转换。

```
use std::fs::File;
use std::io::Read;
use std::path::Path;
use std::error;

fn sum_file(path: &Path) -> Result<i32, Box<dyn error::Error>> {
    let mut file = File::open(path)?; // io::Error -> Box<dyn error::Error>
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // io::Error -> Box<dyn error::Error>
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?;
    }
    Ok(sum)
}
```

While being more concise, this seems to suffer from the same downside of the previous approach by throwing away type information. This is mostly true, but if the caller is aware of the impl details of our function they can still handle the different errors types using the `downcast_ref()` method on `error::Error` which works the same as it does on `dyn Any` types:

这看起来似乎有与第一种方法一样的缺点，丢弃了错误的类型信息。有时确实如此，但倘若上级调用函数知悉该函数的实现细节，那么它仍然可以通过 `error::Error` 特性的 `downcast_ref()` 方法来分辨错误的具体类型，这与实现了 `dyn Any` 特性的类型是一样的：

```
fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("the sum is {}", sum),
        Err(err) => {
            if let Some(e) = err.downcast_ref::<io::Error>() {
                // handle io::Error
            } else if let Some(e) = err.downcast_ref::<ParseIntError>() {
                // handle ParseIntError
            } else {
                // we know sum_file can only return one of the
                // above errors so this branch is unreachable
                // 由于我们知道该函数只能返回以上两种错误,
                // 所以这一选择肢一般是不可能执行的
                unreachable!();
            }
        }
    }
}
```

```
        }
    }
}
}
```

The third approach, which is the most robust and type-safe way to aggregate these different errors would be to build our own custom error type using an enum:

方法三，处理错误的最健壮和类型安全的方法，是通过枚举来构建我们自己的错误类型：

```
use std::num::ParseIntError;
use std::fs::File;
use std::io;
use std::io::Read;
use std::path::Path;
use std::error;
use std::fmt;

#[derive(Debug)]
enum SumFileError {
    Io(io::Error),
    Parse(ParseIntError),
}

impl From<io::Error> for SumFileError {
    fn from(err: io::Error) -> Self {
        SumFileError::Io(err)
    }
}

impl From<ParseIntError> for SumFileError {
    fn from(err: ParseIntError) -> Self {
        SumFileError::Parse(err)
    }
}

impl fmt::Display for SumFileError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            SumFileError::Io(err) => write!(f, "sum file error: {}", err),
            SumFileError::Parse(err) => write!(f, "sum file error: {}", err),
        }
    }
}

impl error::Error for SumFileError {
    // the default impl for this method always returns None
    // but we can now override it to make it way more useful!
    // 在默认实现中，该方法总是返回 None，现在重写它！
    fn source(&self) -> Option<&(dyn error::Error + 'static)> {
        Some(match self {
            SumFileError::Io(err) => err,

```

```

        SumFileError::Parse(err) => err,
    })
}
}

fn sum_file(path: &Path) -> Result<i32, SumFileError> {
    let mut file = File::open(path)?; // io::Error -> SumFileError
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // io::Error -> SumFileError
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?;
    }
    Ok(sum)
}

fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("the sum is {}", sum),
        Err(SumFileError::Io(err)) => {
            // handle io::Error
        },
        Err(SumFileError::Parse(err)) => {
            // handle ParseIntError
        },
    }
}

```

转换特性深入 Conversion Traits Continued

TryFrom & TryInto

预备知识

- [Self](#)
- [Functions](#)
- [Methods](#)
- [Associated Types](#)
- [Generic Parameters](#)
- [Generic Types vs Associated Types](#)
- [Generic Blanket Impls](#)
- [From & Into](#)
- [Error](#)

`TryFrom` 和 `TryInto` 是可失败版本的 `From` 和 `Into`。

`TryFrom` 和 `TryInto` 是可能失败版本的 `From` 和 `Into`。

```
trait TryFrom<T> {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}
```

```
trait TryInto<T> {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

Similarly to `Into` we cannot impl `TryInto` because its impl is provided by this generic blanket impl:

与 `Into` 相似地，我们不能手动实现 `TryInto`，因为它已经为一揽子泛型实现所提供。

```
impl<T, U> TryInto<U> for T
where
    U: TryFrom<T>,
{
    type Error = U::Error;

    fn try_into(self) -> Result<U, U::Error> {
        U::try_from(self)
    }
}
```

Let's say that in the context of our program it doesn't make sense for `Point`'s to have `x` and `y` values that are less than `-1000` or greater than `1000`. This is how we'd rewrite our earlier `From` impls using `TryFrom` to signal to the users of our type that this conversion can now fail:

例如，我们的程序要求 `Point` 的 `x` 和 `y` 的值必须要处于 `-1000` 到 `1000` 之间，相较于 `From`，使用 `TryFrom` 可以告知上级调用者，某些转换可能失败了。

```
use std::convert::TryFrom;
use std::error;
use std::fmt;

struct Point {
```

```

x: i32,
y: i32,
}

#[derive(Debug)]
struct OutOfBounds;

impl fmt::Display for OutOfBounds {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "out of bounds")
    }
}

impl error::Error for OutOfBounds {}

// now fallible
// TryFrom 的转换不允许失败
impl TryFrom<(i32, i32)> for Point {
    type Error = OutOfBounds;
    fn try_from((x, y): (i32, i32)) -> Result<Point, OutOfBounds> {
        if x.abs() > 1000 || y.abs() > 1000 {
            return Err(OutOfBounds);
        }
        Ok(Point { x, y })
    }
}

// still infallible
// From 的转换不允许失败
impl From<Point> for (i32, i32) {
    fn from(Point { x, y }: Point) -> Self {
        (x, y)
    }
}

```

| And here's the refactored `TryFrom<[TryInto<Point>; 3]>` impl for `Triangle` :

| 现在，我们对 `Triangle` 使用 `TryFrom<[TryInto<Point>; 3]>` 进行重构：

```

use std::convert::{TryFrom, TryInto};
use std::error;
use std::fmt;

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

#[derive(Debug)]
struct OutOfBounds;

impl fmt::Display for OutOfBounds {

```

```

fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "out of bounds")
}
}

impl error::Error for OutOfBounds {}

impl TryFrom<(i32, i32)> for Point {
    type Error = OutOfBounds;
    fn try_from((x, y): (i32, i32)) -> Result<Self, Self::Error> {
        if x.abs() > 1000 || y.abs() > 1000 {
            return Err(OutOfBounds);
        }
        Ok(Point { x, y })
    }
}

struct Triangle {
    p1: Point,
    p2: Point,
    p3: Point,
}
}

impl<P> TryFrom<[P; 3]> for Triangle
where
    P: TryInto<Point>,
{
    type Error = P::Error;
    fn try_from([p1, p2, p3]: [P; 3]) -> Result<Self, Self::Error> {
        Ok(Triangle {
            p1: p1.try_into()?,
            p2: p2.try_into()?,
            p3: p3.try_into()?,
        })
    }
}

fn example() -> Result<Triangle, OutOfBounds> {
    let t: Triangle = [(0, 0), (1, 1), (2, 2)].try_into()?;
    Ok(t)
}

```

FromStr

预备知识

- [Self](#)
- [Functions](#)
- [Associated Types](#)
- [Error](#)

- TryFrom & TryInto

```
trait FromStr {  
    type Err;  
    fn from_str(s: &str) -> Result<Self, Self::Err>;  
}
```

`FromStr` types allow performing a fallible conversion from `&str` into `Self`. The idiomatic way to use `FromStr` is to call the `.parse()` method on `&str` `s`:

实现 `FromStr` 特性的类型允许可失败地从 `&str` 转换至 `Self`。使用这一特性的理想方式是，调用 `&str` 实例的 `.parse()` 方法：

```
use std::str::FromStr;  
  
fn example<T: FromStr>(s: &'static str) {  
    // these are all equivalent  
    // 以下方法互相等价  
    let t: Result<T, _> = FromStr::from_str(s);  
    let t = T::from_str(s);  
    let t: Result<T, _> = s.parse();  
    let t = s.parse::<T>(); // most idiomatic  
                           // 最理想的使用方式  
}
```

Example impl for `Point` :

下例为 `Point` 实现了 `FromStr` 特性：

```
use std::error;  
use std::fmt;  
use std::iter::Enumerate;  
use std::num::ParseIntError;  
use std::str::{Chars, FromStr};  
  
#[derive(Debug, Eq, PartialEq)]  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl Point {  
    fn new(x: i32, y: i32) -> Self {  
        Point { x, y }  
    }  
}
```

```

    }

}

#[derive(Debug, PartialEq)]
struct ParsePointError;

impl fmt::Display for ParsePointError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "failed to parse point")
    }
}

impl From<ParseIntError> for ParsePointError {
    fn from(_e: ParseIntError) -> Self {
        ParsePointError
    }
}

impl error::Error for ParsePointError {}

impl FromStr for Point {
    type Err = ParsePointError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let is_num = |(c, _)| matches!(c, '0'..='9' | '-');
        let isnt_num = |t: &(_, _)| !is_num(t);

        let get_num =
            |char_idxs: &mut Enumerate<Chars<'_>>| -> Result<(usize, usize), ParsePointError> {
                let (start, _) = char_idxs
                    .skip_while(isnt_num)
                    .next()
                    .ok_or(ParsePointError)?;
                let (end, _) = char_idxs
                    .skip_while(is_num)
                    .next()
                    .ok_or(ParsePointError)?;
                Ok((start, end))
            };

        let mut char_idxs = s.chars().enumerate();
        let (x_start, x_end) = get_num(&mut char_idxs)?;
        let (y_start, y_end) = get_num(&mut char_idxs)?;

        let x = s[x_start..x_end].parse::<i32>()?;
        let y = s[y_start..y_end].parse::<i32>()?;

        Ok(Point { x, y })
    }
}

#[test] //
fn pos_x_y() {
    let p = "(4, 5)".parse::<Point>();
    assert_eq!(p, Ok(Point::new(4, 5)));
}

#[test] //

```

```

fn neg_x_y() {
    let p = "(-6, -2)".parse::<Point>();
    assert_eq!(p, Ok(Point::new(-6, -2)));
}

#[test] //
fn not_a_point() {
    let p = "not a point".parse::<Point>();
    assert_eq!(p, Err(ParsePointError));
}

```

`FromStr` has the same signature as `TryFrom<&str>`. It doesn't matter which one we impl for a type first as long as we forward the impl to the other one. Here's a `TryFrom<&str>` impl for `Point` assuming it already has a `FromStr` impl:

`FromStr` 与 `TryFrom<&str>` 具有相同的函数签名。先实现哪个特性无关紧要，因为我们可以利用先实现的特性实现后实现的特性。例如，我们假定 `Point` 类型已经实现了 `FromStr` 特性，再来实现 `TryFrom<&str>` 特性：

```

impl TryFrom<&str> for Point {
    type Error = <Point as FromStr>::Err;
    fn try_from(s: &str) -> Result<Point, Self::Error> {
        <Point as FromStr>::from_str(s)
    }
}

```

AsRef & AsMut

预备知识

- [Self](#)
- [Methods](#)
- [Sized](#)
- [Generic Parameters](#)
- [Sized](#)
- [Deref & DerefMut](#)

```

trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}

trait AsMut<T: ?Sized> {

```

```
fn as_mut(&mut self) -> &mut T;  
}
```

AsRef 是 for cheap reference to reference conversions. However, one of the most common ways it's used is to make functions generic over whether they take ownership or not:

AsRef 特性的存在很大程度上便捷了引用转换，其最常见的使用是为函数的引用类型的参数的传入提供方便：

```
// accepts:  
// - &str  
// - &String  
fn takes_str(s: &str) {  
    // use &str  
}  
  
// accepts:  
// - &str  
// - &String  
// - String  
fn takes_asref_str<S: AsRef<str>>(s: S) {  
    let s: &str = s.as_ref();  
    // use &str  
}  
  
fn example(slice: &str, borrow: &String, owned: String) {  
    takes_str(slice);  
    takes_str(borrow);  
    takes_str(owned); //  
    takes_asref_str(slice);  
    takes_asref_str(borrow);  
    takes_asref_str(owned); //  
}
```

The other most common use-case is returning a reference to inner private data wrapped by a type which protects some invariant. A good example from the standard library is String which is just a wrapper around Vec<u8> :

另外一个常见的使用是，返回一个包装类型的内部私有数据的引用（该类型用于保证内部私有数据的不变性）。标准库中的 String 就是对 Vec<u8> 的这样一种包装：

```
struct String {  
    vec: Vec<u8>,  
}
```

This inner `Vec` cannot be made public because if it was people could mutate any byte and break the `String`'s valid UTF-8 encoding. However, it's safe to expose an immutable read-only reference to the inner byte array, hence this impl:

之所以不公开内部的 `Vec` 数据，是因为一旦允许用户随意修改内部数据，就有可能破坏 `String` 有效的 UTF-8 编码。但是，对外开放一个只读的字节数组的引用是安全的，所以有如下实现：

```
impl AsRef<[u8]> for String;
```

Generally, it often only makes sense to impl `AsRef` for a type if it wraps some other type to either provide additional functionality around the inner type or protect some invariant on the inner type.

通常来讲我们不对类型实现 `AsRef` 特性，除非该类型包装了其它类型以提供额外的功能，或是对内部类型提供了不变性的保护。

Let's examine a example of bad `AsRef` impls:

以下是实现 `AsRef` 特性的一个反例：

```
struct User {
    name: String,
    age: u32,
}

impl AsRef<String> for User {
    fn as_ref(&self) -> &String {
        &self.name
    }
}

impl AsRef<u32> for User {
    fn as_ref(&self) -> &u32 {
        &self.age
    }
}
```

This works and kinda makes sense at first, but quickly falls apart if we add more members to `User` :

乍看起来这似乎有几分道理，但是当我们对 `User` 类型添加新的成员时，缺点就暴露出来了：

```

struct User {
    name: String,
    email: String,
    age: u32,
    height: u32,
}

impl AsRef<String> for User {
    fn as_ref(&self) -> &String {
        // uh, do we return name or email here?
        // 既然我们要返回一个字符串引用，那具体应该返回什么呢？
        // name 和 email 都是字符串，如何选择呢？
        // 出于返回类型的限制，似乎我们也难以返回一个混合的字符串。
    }
}

impl AsRef<u32> for User {
    fn as_ref(&self) -> &u32 {
        // uh, do we return age or height here?
        // 如上同理
    }
}

```

A `User` is composed of `String`s and `u32`s but it's not really the same thing as a `String` or a `u32`. Even if we had much more specific types:

`User` 类型由多个 `String` 和 `u32` 类型的成员所组成，但我们也不能说 `User` 是 `String` 或 `u32` 吧？即便由更加具体的类型来构造也不行：

```

struct User {
    name: Name,
    email: Email,
    age: Age,
    height: Height,
}

```

It wouldn't make much sense to impl `AsRef` for any of those because `AsRef` is for cheap reference to reference conversions between semantically equivalent things, and `Name`, `Email`, `Age`, and `Height` by themselves are not the same thing as a `User`.

对于 `User` 这样的类型来讲，实现 `AsRef` 特性并没有什么太多意义。因为 `AsRef` 的存在仅是为了做一种最简单的引用转换，这种转换最好存在于语义上相类似的事物之间。`Name`, `Email`, `Age` 和 `Height` 其本身和 `User` 就不是一回事，在逻辑上谈不上转换。

A good example where we would impl `AsRef` would be if we introduced a new type `Moderator` that just wrapped a `User` and added some moderation specific privileges:

下例展示了 `AsRef` 特性的正确用法，我们实现了一个新的类型 `Moderator`，它仅仅是包装了 `User` 类型，并添加了对其权限的一些控制：

```
struct User {
    name: String,
    age: u32,
}

// unfortunately the standard library cannot provide
// a generic blanket impl to save us from this boilerplate
// 不幸的是，标准库并没有提供相应的一揽子泛型实现，我们不得不手动实现
impl AsRef<User> for User {
    fn as_ref(&self) -> &User {
        self
    }
}

enum Privilege {
    BanUsers,
    EditPosts,
    DeletePosts,
}

// although Moderators have some special
// privileges they are still regular Users
// and should be able to do all the same stuff
// 尽管主持人类具有一些特殊的权限，
// 但其仍然是普通的用户
// 所有用户类能做到的主持人类也应能做到
struct Moderator {
    user: User,
    privileges: Vec<Privilege>
}

impl AsRef<Moderator> for Moderator {
    fn as_ref(&self) -> &Moderator {
        self
    }
}

impl AsRef<User> for Moderator {
    fn as_ref(&self) -> &User {
        &self.user
    }
}

// this should be callable with Users
// and Moderators (who are also Users)
// 这个函数的参数可以是 User 也可以是 Moderator
// ( Moderator 也是 User )
```

```

fn create_post<U: AsRef<User>>(u: U) {
    let user = u.as_ref();
    // etc
}

fn example(user: User, moderator: Moderator) {
    create_post(&user);
    create_post(&moderator); //
}

```

This works because `Moderator` s are just `User` s. Here's the example from the `Deref` section except using `AsRef` instead:

之所以可以这样做，是因为 `Moderator` 就是 `User`。下例是将 `Deref` 一节中的例子使用 `AsRef` 做出替代：

```

use std::convert::AsRef;

struct Human {
    health_points: u32,
}

impl AsRef<Human> for Human {
    fn as_ref(&self) -> &Human {
        self
    }
}

enum Weapon {
    Spear,
    Axe,
    Sword,
}

// a Soldier is just a Human with a Weapon
// 士兵是手持武器的人类
struct Soldier {
    human: Human,
    weapon: Weapon,
}

impl AsRef<Soldier> for Soldier {
    fn as_ref(&self) -> &Soldier {
        self
    }
}

impl AsRef<Human> for Soldier {
    fn as_ref(&self) -> &Human {
        &self.human
    }
}

```

```

enum Mount {
    Horse,
    Donkey,
    Cow,
}

// a Knight is just a Soldier with a Mount
// 骑士是跨骑坐骑的士兵
struct Knight {
    soldier: Soldier,
    mount: Mount,
}

impl AsRef<Knight> for Knight {
    fn as_ref(&self) -> &Knight {
        self
    }
}

impl AsRef<Soldier> for Knight {
    fn as_ref(&self) -> &Soldier {
        &self.soldier
    }
}

impl AsRef<Human> for Knight {
    fn as_ref(&self) -> &Human {
        &self.soldier.human
    }
}

enum Spell {
    MagicMissile,
    FireBolt,
    ThornWhip,
}

// a Mage is just a Human who can cast Spells
// 法师是口诵咒语的人类
struct Mage {
    human: Human,
    spells: Vec<Spell>,
}

impl AsRef<Mage> for Mage {
    fn as_ref(&self) -> &Mage {
        self
    }
}

impl AsRef<Human> for Mage {
    fn as_ref(&self) -> &Human {
        &self.human
    }
}

enum Staff {
    Wooden,
}

```

```

Metallic,
Plastic,
}

// a Wizard is just a Mage with a Staff
// 巫师是腰别法宝的法师
struct Wizard {
    mage: Mage,
    staff: Staff,
}

impl AsRef<Wizard> for Wizard {
    fn as_ref(&self) -> &Wizard {
        self
    }
}

impl AsRef<Mage> for Wizard {
    fn as_ref(&self) -> &Mage {
        &self.mage
    }
}

impl AsRef<Human> for Wizard {
    fn as_ref(&self) -> &Human {
        &self.mage.human
    }
}

fn borrows_human<H: AsRef<Human>>(human: H) {}
fn borrows_soldier<S: AsRef<Soldier>>(soldier: S) {}
fn borrows_knight<K: AsRef<Knight>>(knight: K) {}
fn borrows_mage<M: AsRef<Mage>>(mage: M) {}
fn borrows_wizard<W: AsRef<Wizard>>(wizard: W) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types can be used as Humans
    borrows_human(&human);
    borrows_human(&soldier);
    borrows_human(&knight);
    borrows_human(&mage);
    borrows_human(&wizard);
    // Knights can be used as Soldiers
    borrows_soldier(&soldier);
    borrows_soldier(&knight);
    // Wizards can be used as Mages
    borrows_mage(&mage);
    borrows_mage(&wizard);
    // Knights & Wizards passed as themselves
    borrows_knight(&knight);
    borrows_wizard(&wizard);
}

```

Deref didn't work in the prior version of the example above because deref coercion is an implicit conversion between types which leaves room for people to mistakenly formulate the wrong ideas and expectations for

how it will behave. `AsRef` works above because it makes the conversion between types explicit and there's no room leftover to develop any wrong ideas or expectations.

之所以 `Deref` 在上例之前的版本中不可使用，是因为自动解引用是一种隐式的转换，这就为程序员错误地使用留下了巨大的空间。

而 `AsRef` 在上例中可以使用，是因为其实现的转换是显式的，这样很大程度上就消除了犯错误的空间。

Borrow & BorrowMut

预备知识

- [Self](#)
- [Methods](#)
- [Generic Parameters](#)
- [Subtraits & Supertraits](#)
- [Sized](#)
- [AsRef & AsMut](#)
- [PartialEq & Eq](#)
- [Hash](#)
- [PartialOrd & Ord](#)

```
trait Borrow<Borrowed>
where
    Borrowed: ?Sized,
{
    fn borrow(&self) -> &Borrowed;
}

trait BorrowMut<Borrowed>: Borrow<Borrowed>
where
    Borrowed: ?Sized,
{
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

These traits were invented to solve the very specific problem of looking up `String` keys in `HashSet`s, `HashMap`s, `BTreeSet`s, and `BTreeMap`s using `&str` values.

这类特性存在的意义旨在于解决特定领域的问题，例如在 `HashSet`，`HashMap`，`BTreeSet`，`BtreeMap` 中使用 `&str` 查询 `String` 类型的键。

We can view `Borrow<T>` and `BorrowMut<T>` as stricter versions of `AsRef<T>` and `AsMut<T>` , where the returned reference `&T` has equivalent `Eq` , `Hash` , and `Ord` impls to `Self` . This is more easily explained with a commented example:

我们可以将 `Borrow<T>` 和 `BorrowMut<T>` 视作 `AsRef<T>` 和 `AsMut<T>` 的严格版本，其返回的引用 `&T` 具有与 `Self` 相同的 `Eq`，`Hash` 和 `Ord` 的实现。这一点在下例的注释中得到很好的解释：

```
use std::borrow::Borrow;
use std::hash::Hasher;
use std::collections::hash_map::DefaultHasher;
use std::hash::Hash;

fn get_hash<T: Hash>(t: T) -> u64 {
    let mut hasher = DefaultHasher::new();
    t.hash(&mut hasher);
    hasher.finish()
}

fn asref_example<Owned, Ref>(owned1: Owned, owned2: Owned)
where
    Owned: Eq + Ord + Hash + AsRef<Ref>,
    Ref: Eq + Ord + Hash
{
    let ref1: &Ref = owned1.as_ref();
    let ref2: &Ref = owned2.as_ref();

    // refs aren't required to be equal if owned types are equal
    // 值相等，不意味着其引用一定相等
    assert_eq!(owned1 == owned2, ref1 == ref2); //

    let owned1_hash = get_hash(&owned1);
    let owned2_hash = get_hash(&owned2);
    let ref1_hash = get_hash(&ref1);
    let ref2_hash = get_hash(&ref2);

    // ref hashes aren't required to be equal if owned type hashes are equal
    // 值的哈希值相等，其引用不一定相等
    assert_eq!(owned1_hash == owned2_hash, ref1_hash == ref2_hash); //

    // ref comparisons aren't required to match owned type comparisons
    // 值的比较，与其应用的比较没有必然联系
    assert_eq!(owned1.cmp(&owned2), ref1.cmp(&ref2)); //
}

fn borrow_example<Owned, Borrowed>(owned1: Owned, owned2: Owned)
where
    Owned: Eq + Ord + Hash + Borrow<Borrowed>,
```

```

Borrowed: Eq + Ord + Hash
{
    let borrow1: &Borrowed = owned1.borrow();
    let borrow2: &Borrowed = owned2.borrow();

    // borrows are required to be equal if owned types are equal
    // 值相等，借用值也必须相等
    assert_eq!(owned1 == owned2, borrow1 == borrow2); //

    let owned1_hash = get_hash(&owned1);
    let owned2_hash = get_hash(&owned2);
    let borrow1_hash = get_hash(&borrow1);
    let borrow2_hash = get_hash(&borrow2);

    // borrow hashes are required to be equal if owned type hashes are equal
    // 值的哈希值相等，借用值的哈希值也必须相等
    assert_eq!(owned1_hash == owned2_hash, borrow1_hash == borrow2_hash); //

    // borrow comparisons are required to match owned type comparisons
    // 值的比较，与借用值的比较必须步调一致
    assert_eq!(owned1.cmp(&owned2), borrow1.cmp(&borrow2)); //
}

```

It's good to be aware of these traits and understand why they exist since it helps demystify some of the methods on `HashSet`, `HashMap`, `BTreeSet`, and `BTreeMap` but it's very rare that we would ever need to impl these traits for any of our types because it's very rare that we would ever need create a pair of types where one is the "borrowed" version of the other in the first place. If we have some `T` then `&T` will get the job done 99.99% of the time, and `T: Borrow<T>` is already implemented for all `T` because of a generic blanket impl, so we don't need to manually impl it and we don't need to create some `U` such that `T: Borrow<U>`.

理解这类特性存在的意义，有助于我们揭开 `HashSet`, `HashMap`, `BTreeSet` 和 `BTreeMap` 中某些方法的实现的神秘面纱。但是在实际应用中，几乎没有什么地方需要我们去实现这样的特性，因为再难找到一个需要我们对一个值再创造一个“借用”版本的类型的场景了。对于某种类型 `T`, `&T` 就能解决 99.9% 的问题了，且 `T: Borrow<T>` 已经被一揽子泛型实现对 `T` 实现了，所以我们无需手动实现它，也无需去实现某种的对 `U` 有 `T: Borrow<U>` 了。

ToOwned

预备知识

- [Self](#)
- [Methods](#)
- [Default Impls](#)
- [Clone](#)
- [Borrow & BorrowMut](#)

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;

    // provided default impls
    // 提供默认实现
    fn clone_into(&self, target: &mut Self::Owned);
}
```

`ToOwned` 是一个更通用的 `Clone` 版本。`Clone` 允许我们从 `&T` 类型得到 `T` 类型，而 `ToOwned` 允许我们从 `&Borrowed` 类型得到 `Owned` 类型，其中 `Owned: Borrow<Owned>`。

`ToOwned` 特性是 `Clone` 特性的泛型版本。`Clone` 特性允许我们由 `&T` 类型得到 `T` 类型，而 `ToOwned` 特性允许我们由 `&Borrow` 类型得到 `Owned` 类型，其中 `Owned: Borrow<Owned>`。

In other words, we can't "clone" a `&str` into a `String`, or a `&Path` into a `PathBuf`, or an `&OsStr` into an `OsString`, since the `clone` method signature doesn't support this kind of cross-type cloning, and that's what `ToOwned` was made for.

换句话讲，我们不能将 `&str` 克隆为 `String`，将 `&Path` 克隆为 `PathBuf` 或将 `&OsStr` 克隆为 `OsString`。鉴于 `clone` 方法的签名不支持这样跨类型的克隆，这就是 `ToOwned` 特性存在的意义。

For similar reasons as `Borrow` and `BorrowMut`, it's good to be aware of this trait and understand why it exists but it's very rare we'll ever need to impl it for any of our types.

与 `Borrow` 和 `BorrowMut` 相同地，理解此类特性存在的意义对我们或有帮助，但是鲜少需要我们手动为自己的类实现该特性。

迭代特性 Iteration Traits

Iterator

预备知识

- [Self](#)
- [Methods](#)

- [Associated Types](#)
- [Default Impls](#)

```

trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

// provided default impls
// 提供默认实现
fn size_hint(&self) -> (usize, Option<usize>);
fn count(self) -> usize;
fn last(self) -> Option<Self::Item>;
fn advance_by(&mut self, n: usize) -> Result<(), usize>;
fn nth(&mut self, n: usize) -> Option<Self::Item>;
fn step_by(self, step: usize) -> StepBy<Self>;
fn chain<U>(
    self,
    other: U
) -> Chain<Self, <U as Intolterator>::Intolter>
where
    U: Intolterator<Item = Self::Item>;
fn zip<U>(self, other: U) -> Zip<Self, <U as Intolterator>::Intolter>
where
    U: Intolterator;
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    F: FnMut(Self::Item) -> B;
fn for_each<F>(self, f: F)
where
    F: FnMut(Self::Item);
fn filter<P>(self, predicate: P) -> Filter<Self, P>
where
    P: FnMut(&Self::Item) -> bool;
fn filter_map<B, F>(self, f: F) -> FilterMap<Self, F>
where
    F: FnMut(Self::Item) -> Option<B>;
fn enumerate(self) -> Enumerate<Self>;
fn peekable(self) -> Peekable<Self>;
fn skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>
where
    P: FnMut(&Self::Item) -> bool;
fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
where
    P: FnMut(&Self::Item) -> bool;
fn map_while<B, P>(self, predicate: P) -> MapWhile<Self, P>
where
    P: FnMut(Self::Item) -> Option<B>;
fn skip(self, n: usize) -> Skip<Self>;
fn take(self, n: usize) -> Take<Self>;
fn scan<St, B, F>(self, initial_state: St, f: F) -> Scan<Self, St, F>
where
    F: FnMut(&mut St, Self::Item) -> Option<B>;
fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>
where

```

```
F: FnMut(Self::Item) -> U,
U: Intolterator;
fn flatten(self) -> Flatten<Self>
where
    Self::Item: Intolterator;
fn fuse(self) -> Fuse<Self>;
fn inspect<F>(self, f: F) -> Inspect<Self, F>
where
    F: FnMut(&Self::Item);
fn by_ref(&mut self) -> &mut Self;
fn collect<B>(self) -> B
where
    B: Fromlterator<Self::Item>;
fn partition<B, F>(self, f: F) -> (B, B)
where
    F: FnMut(&Self::Item) -> bool,
    B: Default + Extend<Self::Item>;
fn partition_in_place<'a, T, P>(self, predicate: P) -> usize
where
    Self: DoubleEndedIterator<Item = &'a mut T>,
    T: 'a,
    P: FnMut(&T) -> bool;
fn is_partitioned<P>(self, predicate: P) -> bool
where
    P: FnMut(Self::Item) -> bool;
fn try_fold<B, F, R>(&mut self, init: B, f: F) -> R
where
    F: FnMut(B, Self::Item) -> R,
    R: Try<Ok = B>;
fn try_for_each<F, R>(&mut self, f: F) -> R
where
    F: FnMut(Self::Item) -> R,
    R: Try<Ok = ()>;
fn fold<B, F>(self, init: B, f: F) -> B
where
    F: FnMut(B, Self::Item) -> B;
fn fold_first<F>(self, f: F) -> Option<Self::Item>
where
    F: FnMut(Self::Item, Self::Item) -> Self::Item;
fn all<F>(&mut self, f: F) -> bool
where
    F: FnMut(Self::Item) -> bool;
fn any<F>(&mut self, f: F) -> bool
where
    F: FnMut(Self::Item) -> bool;
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
where
    P: FnMut(&Self::Item) -> bool;
fn find_map<B, F>(&mut self, f: F) -> Option<B>
where
    F: FnMut(Self::Item) -> Option<B>;
fn try_find<F, R>(
    &mut self,
    f: F
) -> Result<Option<Self::Item>, <R as Try>::Error>
where
    F: FnMut(&Self::Item) -> R,
    R: Try<Ok = bool>;
```

```
fn position<P>(&mut self, predicate: P) -> Option<usize>
where
    P: FnMut(Self::Item) -> bool;
fn rposition<P>(&mut self, predicate: P) -> Option<usize>
where
    Self: ExactSizedIterator + DoubleEndedIterator,
    P: FnMut(Self::Item) -> bool;
fn max(self) -> Option<Self::Item>
where
    Self::Item: Ord;
fn min(self) -> Option<Self::Item>
where
    Self::Item: Ord;
fn max_by_key<B, F>(self, f: F) -> Option<Self::Item>
where
    F: FnMut(&Self::Item) -> B,
    B: Ord;
fn max_by<F>(self, compare: F) -> Option<Self::Item>
where
    F: FnMut(&Self::Item, &Self::Item) -> Ordering;
fn min_by_key<B, F>(self, f: F) -> Option<Self::Item>
where
    F: FnMut(&Self::Item) -> B,
    B: Ord;
fn min_by<F>(self, compare: F) -> Option<Self::Item>
where
    F: FnMut(&Self::Item, &Self::Item) -> Ordering;
fn rev(self) -> Rev<Self>
where
    Self: DoubleEndedIterator;
fn unzip<A, B, FromA, FromB>(self) -> (FromA, FromB)
where
    Self: Iterator<Item = (A, B)>,
    FromA: Default + Extend<A>,
    FromB: Default + Extend<B>;
fn copied<'a, T>(self) -> Copied<Self>
where
    Self: Iterator<Item = &'a T>,
    T: 'a + Copy;
fn cloned<'a, T>(self) -> Cloned<Self>
where
    Self: Iterator<Item = &'a T>,
    T: 'a + Clone;
fn cycle(self) -> Cycle<Self>
where
    Self: Clone;
fn sum<S>(self) -> S
where
    S: Sum<Self::Item>;
fn product<P>(self) -> P
where
    P: Product<Self::Item>;
fn cmp<l>(self, other: l) -> Ordering
where
    l: IntIterator<Item = Self::Item>,
    Self::Item: Ord;
fn cmp_by<l, F>(self, other: l, cmp: F) -> Ordering
where
```

```
F: FnMut(Self::Item, <I as Intolterator>::Item) -> Ordering,
I: Intolterator;
fn partial_cmp<I>(self, other: I) -> Option<Ordering>
where
    I: Intolterator,
    Self::Item: PartialOrd<<I as Intolterator>::Item>;
fn partial_cmp_by<I, F>(
    self,
    other: I,
    partial_cmp: F
) -> Option<Ordering>
where
    F: FnMut(Self::Item, <I as Intolterator>::Item) -> Option<Ordering>,
    I: Intolterator;
fn eq<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialEq<<I as Intolterator>::Item>;
fn eq_by<I, F>(self, other: I, eq: F) -> bool
where
    F: FnMut(Self::Item, <I as Intolterator>::Item) -> bool,
    I: Intolterator;
fn ne<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialEq<<I as Intolterator>::Item>;
fn lt<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialOrd<<I as Intolterator>::Item>;
fn le<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialOrd<<I as Intolterator>::Item>;
fn gt<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialOrd<<I as Intolterator>::Item>;
fn ge<I>(self, other: I) -> bool
where
    I: Intolterator,
    Self::Item: PartialOrd<<I as Intolterator>::Item>;
fn is_sorted(self) -> bool
where
    Self::Item: PartialOrd<Self::Item>;
fn is_sorted_by<F>(self, compare: F) -> bool
where
    F: FnMut(&Self::Item, &Self::Item) -> Option<Ordering>;
fn is_sorted_by_key<F, K>(self, f: F) -> bool
where
    F: FnMut(Self::Item) -> K,
    K: PartialOrd<K>;
}
```

`Iterator<Item = T>` types can be iterated and will produce `T` types. There's no `IteratorMut` trait. Each `Iterator` impl can specify whether it returns immutable references, mutable references, or owned values via the `Item` associated type.

实现 `Iterator<Item = T>` 的类型可以迭代产生 `T` 类型。注意：并不存在 `IteratorMut` 类型，因为可以通过在实现 `Iterator` 特性时指定 `Item` 关联类型，来选择其返回的是不可变引用、可变引用还是自有值。

| <code>Vec<T></code> | 方法 | 返回类型 |
|---------------------------|---------------------------|--|
| | <code>.iter()</code> | <code>Iterator<Item = &T></code> |
| | <code>.iter_mut()</code> | <code>Iterator<Item = &mut T></code> |
| | <code>.into_iter()</code> | <code>Iterator<Item = T></code> |

Something that is not immediately obvious to beginner Rustaceans but that intermediate Rustaceans take for granted is that most types are not their own iterators. If a type is iterable we almost always impl some custom iterator type which iterates over it rather than trying to make it iterate over itself:

对于 Rust 的初学者而言可能有些费解，但是对于中级学习者而言则是顺理成章的一件事是——绝大多数类型并不是自己的迭代器。这意味着，如果某种类型是可迭代的，那么应当实现某种额外的迭代器类型去迭代它，而不是让它自己迭代自己。

```
struct MyType {
    items: Vec<String>
}

impl MyType {
    fn iter(&self) -> impl Iterator<Item = &String> {
        MyTypeIterator {
            index: 0,
            items: &self.items
        }
    }
}

struct MyTypeIterator<'a> {
    index: usize,
    items: &'a Vec<String>
}

impl<'a> Iterator for MyTypeIterator<'a> {
    type Item = &'a String;
    fn next(&mut self) -> Option<Self::Item> {
        if self.index >= self.items.len() {
            None
        } else {
            let item = &self.items[self.index];
            self.index += 1;
            Some(item)
        }
    }
}
```

```
    } else {
        let item = &self.items[self.index];
        self.index += 1;
        Some(item)
    }
}
```

For the sake of teaching the above example shows how to impl an `Iterator` from scratch but the idiomatic solution in this situation would be to just defer to `Vec`'s `iter` method:

出于教学的原因，我们在上例中从头手动实现了一个迭代器。而在这种情况下，最理想的做法是直接调用 `Vec` 的 `iter` 方法。

```
struct MyType {
    items: Vec<String>
}

impl MyType {
    fn iter(&self) -> impl Iterator<Item = &String> {
        self.items.iter()
    }
}
```

Also this is a good generic blanket impl to be aware of:

另外，最好了解这个一揽子泛型实现：

```
impl<l: Iterator + ?Sized> Iterator for &mut l;
```

It says that any mutable reference to an iterator is also an iterator. This is useful to know because it allows us to use iterator methods with `self` receivers as if they had `&mut self` receivers.

任何迭代器的可变引用也是一个迭代器。了解这样的性质有助于我们理解，为什么可以将迭代器的某些参数为 `self` 的方法当作具有 `&mut self` 参数的方法来使用。

As an example, imagine we have a function which processes an iterator of more than three items, but the first step of the function is to take out the first three items of the iterator and process them separately before iterating over the remaining items, here's how a beginner may attempt to write this function:

举个例子，想象我们有这样一个函数，它处理一个具有三个以上值的迭代器，这个函数首先要取得该迭代器的前三个值并分别地处理他们，然后再依次迭代剩余的值。初学者可能会这样实现该函数：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = iter.take(3).collect();
    for item in iter { // iter consumed in line above
        // iter 在上一行就已经被消耗掉了
        // process remaining items
        // 处理剩余的值
    }
}
```

Well that's annoying. The `take` method has a `self` receiver so it seems like we cannot call it without consuming the whole iterator! Here's what a naive refactor of the above code might look like:

糟糕，`take` 方法具有 `self` 参数，这意味着我们不能在不消耗掉整个迭代器的前提下调用该方法。以下可能是一个初学者的改进：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = vec![
        iter.next().unwrap(),
        iter.next().unwrap(),
        iter.next().unwrap(),
    ];
    for item in iter { //
        // process remaining items
        // 处理剩余的值
    }
}
```

Which is okay. However, the idiomatic refactor is actually:

这是可行的，但是理想的改进方式莫过于：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = iter.by_ref().take(3).collect();
    for item in iter { //
        // process remaining items
        // 处理剩余的值
    }
}
```

| Not very easy to discover. But anyway, now we know.

这真是一个很隐蔽的方法，但是被我们抓到了。

| Also, there are no rules or conventions on what can or cannot be an iterator. If the type implements `Iterator` then it's an iterator. Some creative examples from the standard library:

同样，对于什么可以是迭代器，什么不可以是，并无一定之规。实现了 `Iterator` 特性的就是迭代器。而在标准库中，确有一些具有创造性的用例：

```
use std::sync::mpsc::channel;
use std::thread;

fn paths_can_be_iterated(path: &Path) {
    for part in path {
        // iterate over parts of a path
        // 迭代 path 的不同部分
    }
}

fn receivers_can_be_iterated() {
    let (send, recv) = channel();

    thread::spawn(move || {
        send.send(1).unwrap();
        send.send(2).unwrap();
        send.send(3).unwrap();
    });

    for received in recv {
        // iterate over received values
        // 迭代接收到的值
    }
}
```

Intoliterator

预备知识

- [Self](#)
- [Methods](#)
- [Associated Types](#)
- [Iterator](#)

```
trait Intolterator
where
    <Self::Intolter as Iterator>::Item == Self::Item,
{
    type Item;
    type Intolter: Iterator;
    fn into_iter(self) -> Self::Intolter;
}
```

| Intolterator types can be converted into iterators, hence the name. The into_iter method is called on a type when it's used within a for-in loop:

闻弦歌而知雅意，实现 Intolterator 特性的类型可以被转换为迭代器。当用于 for-in 循环时，将自动调用该类型的 into_iter 方法。

```
// vec = Vec<T>
for v in vec {} // v = T

// above line desugared
// 以上代码等价于
for v in vec.into_iter() {}
```

| Not only does Vec impl Intolterator but so does &Vec and &mut Vec if we'd like to iterate over immutable or mutable references instead of owned values, respectively.

不仅 Vec 实现了 Intolterator 特性，&Vec 与 &mut Vec 同样如此。因此我们可以相应的对可变与不可变的引用，以及自有值进行迭代。

```
// vec = Vec<T>
for v in &vec {} // v = &T

// above example desugared
// 以上代码等价于
for v in (&vec).into_iter() {}

// vec = Vec<T>
for v in &mut vec {} // v = &mut T

// above example desugared
// 以上代码等价于
for v in (&mut vec).into_iter() {}
```

FromIterator

预备知识

- [Self](#)
- [Functions](#)
- [Generic Parameters](#)
- [Iterator](#)
- [Intolterator](#)

```
trait FromIterator<A> {  
    fn from_iter<T>(iter: T) -> Self  
    where  
        T: Intolterator<Item = A>;  
}
```

`FromIterator` types can be created from an iterator, hence the name. `FromIterator` is most commonly and idiomatically used by calling the `collect` method on `Iterator`:

顾叶落而晓秋至，实现 `FromIterator` 特性的类型可以由迭代器而构造。`FromIterator` 特性最常见和最理想的使用方法是调用 `Iterator` 的 `collect` 方法：

```
fn collect<B>(self) -> B  
where  
    B: FromIterator<Self::Item>;
```

Example of collecting an `Iterator<Item = char>` into a `String`:

下例展示了如何将 `Iterator<Item = char>` 迭代器的值收集为 `String`：

```
fn filter_letters(string: &str) -> String {  
    string.chars().filter(|c| c.is_alphabetic()).collect()  
}
```

All the collections in the standard library impl `Intolterator` and `FromIterator` so that makes it easier to convert between them:

标准库中的全部集合类型都实现了 `IntoIterator` 和 `FromIterator` 特性，所以在它们之间进行转换是很方便的：

```
use std::collections::{BTreeSet, HashMap, HashSet, LinkedList};

// String -> HashSet<char>
fn unique_chars(string: &str) -> HashSet<char> {
    string.chars().collect()
}

// Vec<T> -> BTreeSet<T>
fn ordered_unique_items<T: Ord>(vec: Vec<T>) -> BTreeSet<T> {
    vec.into_iter().collect()
}

// HashMap<K, V> -> LinkedList<(K, V)>
fn entry_list<K, V>(map: HashMap<K, V>) -> LinkedList<(K, V)> {
    map.into_iter().collect()
}

// and countless more possible examples
// 还有数不胜数的例子
```

输入输出特性 I/O Traits

Read & Write

预备知识

- [Self](#)
- [Methods](#)
- [Scope](#)
- [Generic Blanket Impls](#)

```
trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
    // provided default impls
    // 提供默认实现
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut'<'_>]) -> Result<usize>;
    fn is_read_vectored(&self) -> bool;
    unsafe fn initializer(&self) -> Initializer;
```

```

fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize>;
fn read_to_string(&mut self, buf: &mut String) -> Result<usize>;
fn read_exact(&mut self, buf: &mut [u8]) -> Result<()>;
fn by_ref(&mut self) -> &mut Self
where
    Self: Sized;
fn bytes(self) -> Bytes<Self>
where
    Self: Sized;
fn chain<R: Read>(self, next: R) -> Chain<Self, R>
where
    Self: Sized;
fn take(self, limit: u64) -> Take<Self>
where
    Self: Sized;
}
}

trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    // provided default impls
    // 提供默认实现
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize>;
    fn is_write_vectored(&self) -> bool;
    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments<'_>) -> Result<()>;
    fn by_ref(&mut self) -> &mut Self
    where
        Self: Sized;
}

```

| Generic blanket impls worth knowing:

| 值得关注的一揽子泛型实现：

```

impl<R: Read + ?Sized> Read for &mut R;
impl<W: Write + ?Sized> Write for &mut W;

```

| These say that any mutable reference to a `Read` type is also `Read`, and same with `Write`. This is useful to know because it allows us to use any method with a `self` receiver as if it had a `&mut self` receiver. We already went over how to do this and why it's useful in the `Iterator` trait section so I'm not going to repeat it again here.

对于任何实现了 `Read` 特性的类型，其可变的引用类型也实现了 `Read` 特性。`Write` 也是如此。知晓这一点有助于我们理解为什么，对于具有 `self` 参数的函数可以如同那些具有 `&mut self` 参数的函数一般使用。鉴于我们已经在 `Iterator` 特性一节中做出了相近的说明，对此我不再赘述。

I'd like to point out that `&[u8]` implements `Read` and that `Vec<u8>` implements `Write` so we can easily unit test our file handling functions using `String`'s which are trivial to convert to `&[u8]` and from `Vec<u8>`:

我特别指出的是，在 `&[u8]` 实现 `Read` 的同时，`Vec<u8>` 实现了 `Write`，因此我们可以很方便地使用 `String` 来对我们的文件处理函数进行单元测试，因为它可以轻易地转换到 `&[u8]` 和转换自 `Vec<u8>`。

```
use std::path::Path;
use std::fs::File;
use std::io::Read;
use std::io::Write;
use std::io;

// function we want to test
// 欲要测试此函数
fn uppercase<R: Read, W: Write>(mut read: R, mut write: W) -> Result<(), io::Error> {
    let mut buffer = String::new();
    read.read_to_string(&mut buffer)?;
    let uppercase = buffer.to_uppercase();
    write.write_all(uppercase.as_bytes())?;
    write.flush()?;
    Ok(())
}

// in actual program we'd pass Files
// 实际使用中我们传入文件
fn example(in_path: &Path, out_path: &Path) -> Result<(), io::Error> {
    let in_file = File::open(in_path)?;
    let out_file = File::open(out_path)?;
    uppercase(in_file, out_file)
}

// however in unit tests we can use Strings!
// 但是在单元测试中我们使用 String !
#[test] // 
fn example_test() {
    let in_file: String = "i am screaming".into();
    let mut out_file: Vec<u8> = Vec::new();
    uppercase(in_file.as_bytes(), &mut out_file).unwrap();
    let out_result = String::from_utf8(out_file).unwrap();
    assert_eq!(out_result, "I AM SCREAMING");
}
```

结语 Conclusion

| We learned a lot together! Too much in fact. This is us now:

我们真是学习了太多！太多了！可能这就是我们现在的样子：

该漫画的创作者: [The Jenkins Comic](#)

讨论 Discuss

| Discuss this article on

可以在如下地点讨论本文

- [Github](#)
- [learnrust subreddit](#)
- [official Rust users forum](#)
- [Twitter](#)
- [lobste.rs](#)
- [rust subreddit](#)

通告 Notifications

| Get notified when the next blog post get published by

在如下处得知我下一篇博文的详情

- [订阅我的推特 pretzelhammer 或者](#)
- [订阅这个 repo \(点击 Watch -> 点击 Custom -> 选择 Releases -> 点击 Apply \)](#)

更多资料 Further Reading

- [Sizedness in Rust](#)
- [Common Rust Lifetime Misconceptions](#)
- [Learning Rust in 2020](#)
- [Learn Assembly with Entirely Too Many Brainfuck Compilers](#)

翻译 Translation

鉴于水平所限，

难免出现翻译错误，

如发现错误还请告知！

skanfd 译

2021年4月21日

核心库

核心库

优秀项目

extism插件框架

根据 [遗忘曲线](#)：如果没有记录和回顾，6天后便会忘记75%的内容

读书笔记正是帮助你记录和回顾的工具，不必拘泥于形式，其核心是：记录、翻看、思考

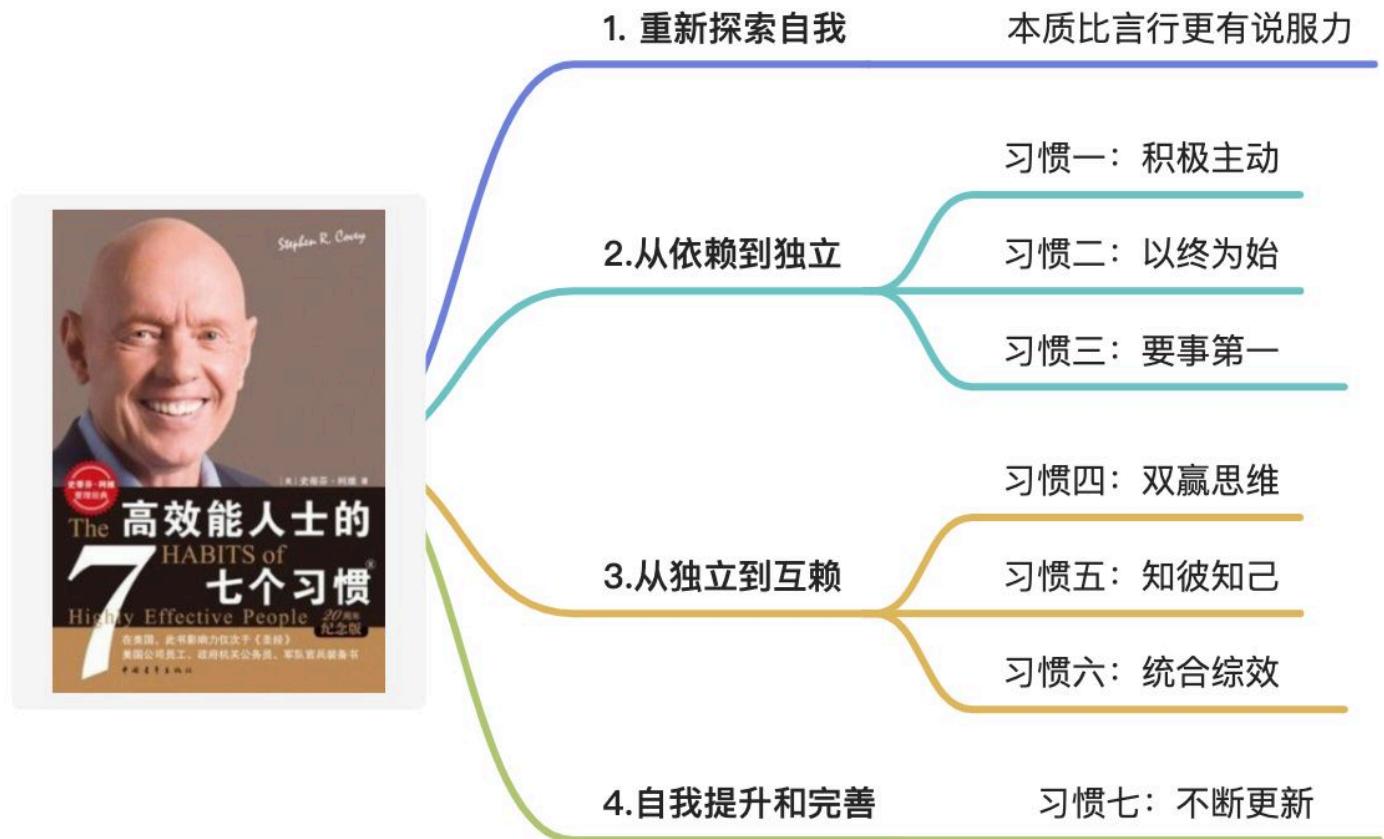
书名

高效能人士的七个习惯

| | |
|----|--|
| 作者 | 史蒂芬·柯维 |
| 状态 | 待开始 阅读中 已读完 |
| 简介 | 本书精选柯维博士“七个习惯”的最核心思想和方法，为忙碌人士带来超价值的自我提升体验。用最少的时间，参透高效能人士的持续成功之路。 |

思维导图

用思维导图，结构化记录本书的核心观点。



读后感

观点1

读完该书后，受益的核心观点与说明...

观点2

读完该书后，受益的核心观点与说明...

观点3

读完该书后，受益的核心观点与说明...

书摘

- 该书的金句摘录...

- 该书的金句摘录...
- 该书的金句摘录...

相关资料

可通过“+K”插入引用链接，或使用“本地文件”引入源文件。

<https://book.douban.com/subject/5325618/>

Tokio 源码分析 字节流 Bytes

网络应用的核心是处理字节流，本篇关注 Tokio 处理字节流的基础库 [bytes](#)，阅读的代码版本为 [v0.6.0](#)。

Tokio 架构图 from [tokio.rs](#)

1. 概览

Bytes 封装了对字节流的常用操作，核心特性是使用引用计数实现内存安全的 `string_view`。`src` 目录下的文件结构为：

```
src
├── buf          # buffer 实现
│   ├── buf_impl.rs
│   ├── buf_mut.rs
│   ├── chain.rs
│   ├── iter.rs
│   ├── limit.rs
│   ├── mod.rs
│   ├── reader.rs
│   ├── take.rs
│   ├── uninit_slice.rs
│   ├── vec_deque.rs
│   └── writer.rs
├── bytes.rs      # bytes 实现
├── bytes_mut.rs # mutable bytes
├── fmt           # 输出格式
│   ├── debug.rs
│   ├── hex.rs
│   └── mod.rs
└── lib.rs
└── loom.rs       # 引用计数
└── serde.rs      # serde 序列化支持
```

来看一个单元测试：

```
#[test]
fn slice() {
    let a = Bytes::from(&b"hello world"[..]);

    let b = a.slice(3..5);
    assert_eq!(b, b"lo"[..]);

    let b = a.slice(0..0);
    assert_eq!(b, b""[..]);

    let b = a.slice(3..3);
    assert_eq!(b, b""[..]);

    let b = a.slice(a.len()..a.len());
    assert_eq!(b, b""[..]);

    let b = a.slice(..5);
    assert_eq!(b, b"hello"[..]);

    let b = a.slice(3..);
    assert_eq!(b, b"lo world"[..]);
}
```

2. Buf

`Buf` 类似 LevelDB 中的 `Slice`，其 trait 定义在 [src/buf/buf_impl.rs](#) 中，依赖 `reader.rs` / `take.rs` / `chain.rs`，依次看依赖的文件：

```
#[derive(Debug)]
pub struct Reader<B> {
    buf: B,
}

pub fn new<B>(buf: B) -> Reader<B> {
    Reader { buf }
}

impl<B: Buf + Sized> io::Read for Reader<B> {
    fn read(&mut self, dst: &mut [u8]) -> io::Result<usize> {
        let len = cmp::min(self.buf.remaining(), dst.len());

        Buf::copy_to_slice(&mut self.buf, &mut dst[0..len]);
        Ok(len)
    }
}

pub trait Buf {
```

```

#[cfg(feature = "std")]
fn reader(self) -> Reader<Self>
    where
        Self: Sized,
    {
        reader::new(self)
    }

fn copy_to_slice(&mut self, dst: &mut [u8]) {
    let mut off = 0;

    assert!(self.remaining() >= dst.len());

    while off < dst.len() {
        let cnt;

        unsafe {
            let src = self.bytes();
            cnt = cmp::min(src.len(), dst.len() - off);

            ptr::copy_nonoverlapping(src.as_ptr(), dst[off..].as_mut_ptr(), cnt);

            off += cnt;
        }

        self.advance(cnt);
    }
}

```

`Reader` 内部包含一个 `buf: B` 对象，执行 `io::Read::read` 时可以直接复制 `buf` 的内存到 `dst`。
`Buf::copy_to_slice` 的过程中会将指针移动到复制结束的位置。`Buf::copy_to_slice` 可以换成 `self.buf.copy_to_slice`。复制过程中调用的 `ptr::copy_nonoverlapping` 是 `unsafe` 的，语义上和 `memcpy` 等价，要求 `src` 和 `dst` 指向的两段内存不存在重叠。

```

#[derive(Debug)]
pub struct Take<T> {
    inner: T,
    limit: usize,
}

pub fn new<T>(inner: T, limit: usize) -> Take<T> {
    Take { inner, limit }
}

impl<T: Buf> Buf for Take<T> {
    fn remaining(&self) -> usize {
        cmp::min(self.inner.remaining(), self.limit)
    }

    fn bytes(&self) -> &[u8] {
        let bytes = self.inner.bytes();
        &bytes[..cmp::min(bytes.len(), self.limit)]
    }
}

```

```

fn advance(&mut self, cnt: usize) {
    assert!(cnt <= self.limit);
    self.inner.advance(cnt);
    self.limit -= cnt;
}
}

pub trait Buf {

    fn take(self, limit: usize) -> Take<Self>
        where
            Self: Sized,
        {
            take::new(self, limit)
        }
}

```

`Take` 内部包含一个 `inner: T` 对象以及限制长度的 `limit`。比较有意思的是 `Take<Buf>` 对象也是一种 `Buf`，也就是说你可以递归的调用 `buf.take(limit)`。

```

#[derive(Debug)]
pub struct Chain<T, U> {
    a: T,
    b: U,
}

impl<T, U> Buf for Chain<T, U>
    where
        T: Buf,
        U: Buf,
    {
        fn remaining(&self) -> usize {
            self.a.remaining() + self.b.remaining()
        }

        fn bytes(&self) -> &[u8] {
            if self.a.has_remaining() {
                self.a.bytes()
            } else {
                self.b.bytes()
            }
        }

        fn advance(&mut self, mut cnt: usize) {
            let a_rem = self.a.remaining();

            if a_rem != 0 {
                if a_rem >= cnt {
                    self.a.advance(cnt);
                    return;
                }
            }
        }
    }
}

```

```

        self.a.advance(a_rem);

        cnt -= a_rem;
    }

    self.b.advance(cnt);
}

#[cfg(feature = "std")]
fn bytes_vectored<'a>(&'a self, dst: &mut [IoSlice<'a>]) -> usize {
    let mut n = self.a.bytes_vectored(dst);
    n += self.b.bytes_vectored(&mut dst[n..]);
    n
}
}

```

`Chain` 对象可以链接两个 `Buf` 对象，并提供两个缓冲区之间的连续视图。换句话说，可以无限的链接 `Buf` 对象，比如 `a.chain(b).chain(c)`，最终得到的类型是 `Chain<Chain<Buf, Buf>, Buf>`。

看完了依赖项，可以继续看 `buf_implementation.rs`。`Buf` trait 中的“纯虚函数”只有下面三个：

```

pub trait Buf {

    fn remaining(&self) -> usize;

    fn bytes(&self) -> &[u8];

    fn advance(&mut self, cnt: usize);
}

```

实现这三个纯虚函数就可以实现 `Buf` trait，比如：

```

impl Buf for &[u8] {
    #[inline]
    fn remaining(&self) -> usize {
        self.len()
    }

    #[inline]
    fn bytes(&self) -> &[u8] {
        self
    }
}

```

```

#[inline]
fn advance(&mut self, cnt: usize) {
    *self = &self[cnt..];
}

```

通过这几个接口，可以实现 `Buf` 的迭代器：

```

#[derive(Debug)]
pub struct Intolter<T> {
    inner: T,
}

impl<T: Buf> Iterator for Intolter<T> {
    type Item = u8;

    fn next(&mut self) -> Option<u8> {
        if !self.inner.has_remaining() {
            return None;
        }

        let b = self.inner.bytes()[0];
        self.inner.advance(1);

        Some(b)
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let rem = self.inner.remaining();
        (rem, Some(rem))
    }
}

```

`Buf` 也有可写的版本 `BufMut`，实现上大同小异，不再赘述。

3. Bytes

`Bytes`，一种低开销的可复制、可切片的连续内存块。通过引用计数保证原始内存块的生命周期，通过 `<ptr, len>` 声明当前指向的位置和大小。一个 `Bytes` 对象的内存布局如下：

```

pub struct Bytes {
    ptr: *const u8,
    len: usize,
    data: AtomicPtr<()>,
}

```

```

    vtable: &'static Vtable,
}

pub(crate) struct Vtable {

    pub clone: unsafe fn(&AtomicPtr<()>, *const u8, usize) -> Bytes,
    pub drop: unsafe fn(&mut AtomicPtr<()>, *const u8, usize),
}

```

`Bytes` 自然会实现 `Buf` trait:

```

impl Buf for Bytes {
    #[inline]
    fn remaining(&self) -> usize {
        self.len()
    }

    #[inline]
    fn bytes(&self) -> &[u8] {
        self.as_slice()
    }

    #[inline]
    fn advance(&mut self, cnt: usize) {
        assert!(
            cnt <= self.len(),
            "cannot advance past `remaining`: {:?} <= {:?}", cnt,
            self.len(),
        );
        unsafe {
            self.inc_start(cnt);
        }
    }

    fn copy_to_bytes(&mut self, len: usize) -> crate::Bytes {
        if len == self.remaining() {
            core::mem::replace(self, Bytes::new())
        } else {
            let ret = self.slice(..len);
            self.advance(len);
            ret
        }
    }
}

```

`Bytes` 的切片操作实现也很简单，通过 `clone` 获得一个新的对象，再修改该对象的指向和长度：

```

impl Bytes {
    pub fn slice(&self, range: impl RangeBounds<usize>) -> Bytes {
        use core::ops::Bound;

        let len = self.len();

        let begin = match range.start_bound() {
            Bound::Included(&n) => n,
            Bound::Excluded(&n) => n + 1,
            Bound::Unbounded => 0,
        };

        let end = match range.end_bound() {
            Bound::Included(&n) => n.checked_add(1).expect("out of range"),
            Bound::Excluded(&n) => n,
            Bound::Unbounded => len,
        };

        assert!(
            begin <= end,
            "range start must not be greater than end: {:?} <= {:?}", begin, end,
        );
        assert!(
            end <= len,
            "range end out of bounds: {:?} <= {:?}", end, len,
        );
    }

    if end == begin {
        return Bytes::new();
    }

    let mut ret = self.clone();

    ret.len = end - begin;
    ret.ptr = unsafe { ret.ptr.offset(begin as isize) };

    ret
}
}

```

为了保证低开销，`clone` 操作有不同的实现方式。对于编译器确定的静态内存块，直接复制 `ptr` 和 `len` 就可以，不需要额外管理析构：

```

const STATIC_VTABLE: Vtable = Vtable {
    clone: static_clone,
    drop: static_drop,
};

unsafe fn static_clone(_: &AtomicPtr<()>, ptr: *const u8, len: usize) -> Bytes {
    let slice = slice::from_raw_parts(ptr, len);

```

```

    Bytes::from_static(slice)
}

unsafe fn static_drop(_: &mut AtomicPtr<()>, _: *const u8, _: usize) {

}

impl Bytes {
    pub fn from_static(bytes: &'static [u8]) -> Bytes {
        Bytes {
            ptr: bytes.as_ptr(),
            len: bytes.len(),
            data: AtomicPtr::new(ptr::null_mut()),
            vtable: &STATIC_VTABLE,
        }
    }
}

```

对于动态申请的内存块，则需要实现引用计数：

```

struct Shared {

    _vec: Vec<u8>,
    ref_cnt: AtomicUsize,
}

const _: [(); 0 - mem::align_of::<Shared>() % 2] = [];

static SHARED_VTABLE: Vtable = Vtable {
    clone: shared_clone,
    drop: shared_drop,
};

unsafe fn shared_clone(data: &AtomicPtr<()>, ptr: *const u8, len: usize) -> Bytes {
    let shared = data.load(Ordering::Relaxed);
    shallow_clone_arc(shared as _, ptr, len)
}

unsafe fn shared_drop(data: &mut AtomicPtr<()>, _ptr: *const u8, _len: usize) {
    data.with_mut(|shared| {
        release_shared(*shared as *mut Shared);
    });
}

unsafe fn shallow_clone_arc(shared: *mut Shared, ptr: *const u8, len: usize) -> Bytes {
    let old_size = (*shared).ref_cnt.fetch_add(1, Ordering::Relaxed);

    if old_size > usize::MAX >> 1 {
        crate::abort();
    }

    Bytes {
        ptr,
        len,
    }
}

```

```

    data: AtomicPtr::new(shared as _),
    vtable: &SHARED_VTABLE,
}
}

unsafe fn release_shared(ptr: *mut Shared) {
    if (*ptr).ref_cnt.fetch_sub(1, Ordering::Release) != 1 {
        return;
    }

    atomic::fence(Ordering::Acquire);
    Box::from_raw(ptr);
}

```

这里实现地比较复杂的是从 `Vec<u8>` 转到 `Bytes`。从 `Vec<u8>` 转到 `Bytes` 时，会将原子指针 `data` 指向该 `Vec<u8>` 的堆内存，并且将最低位设为 1 用以区分指向 `Shared` 对象的原子指针。当发生 `clone` 时，将会使用 `data` 指向的堆内存构建 `Shared` 对象，并将原子指针 `data` 使用 CAS 指向新的 `Shared` 对象，以保证该提升操作只会发生一次。可以学习下这里 `from_raw` / `into_raw` / `mem::forget` 等关于内存的去糖操作。

```

impl From<Vec<u8>> for Bytes {
    fn from(vec: Vec<u8>) -> Bytes {
        if vec.is_empty() {
            return Bytes::new();
        }

        let slice = vec.into_boxed_slice();
        let len = slice.len();
        let ptr = slice.as_ptr();
        drop(Box::into_raw(slice));

        if ptr as usize & 0x1 == 0 {
            let data = ptr as usize | KIND_VEC;
            Bytes {
                ptr,
                len,
                data: AtomicPtr::new(data as *mut _),
                vtable: &PROMOTABLE_EVEN_VTABLE,
            }
        } else {
            Bytes {
                ptr,
                len,
                data: AtomicPtr::new(ptr as *mut _),
                vtable: &PROMOTABLE_ODD_VTABLE,
            }
        }
    }
}

static PROMOTABLE_EVEN_VTABLE: Vtable = Vtable {

```

```

clone: promotable_even_clone,
drop: promotable_even_drop,
};

unsafe fn promotable_even_clone(data: &AtomicPtr<()>, ptr: *const u8, len: usize) -> Bytes {
    let shared = data.load(Ordering::Acquire);
    let kind = shared as usize & KIND_MASK;

    if kind == KIND_ARC {
        shallow_clone_arc(shared as _, ptr, len)
    } else {
        debug_assert_eq!(kind, KIND_VEC);
        let buf = (shared as usize & !KIND_MASK) as *mut u8;
        shallow_clone_vec(data, shared, buf, ptr, len)
    }
}

unsafe fn rebuild_boxed_slice(buf: *mut u8, offset: *const u8, len: usize) -> Box<[u8]> {
    let cap = (offset as usize - buf as usize) + len;
    Box::from_raw(slice::from_raw_parts_mut(buf, cap))
}

#[cold]
unsafe fn shallow_clone_vec(
    atom: &AtomicPtr<()>,
    ptr: *const (),
    buf: *mut u8,
    offset: *const u8,
    len: usize,
) -> Bytes {

let vec = rebuild_boxed_slice(buf, offset, len).into_vec();
    let shared = Box::new(Shared {
        _vec: vec,
        ref_cnt: AtomicUsize::new(2),
    });

    let shared = Box::into_raw(shared);

    debug_assert!(
        0 == (shared as usize & KIND_MASK),
        "internal: Box<Shared> should have an aligned pointer",
    );
}

```

```
let actual = atom.compare_and_swap(ptr as _, shared as _, Ordering::AcqRel);

if actual as usize == ptr as usize {

    return Bytes {
        ptr: offset,
        len,
        data: AtomicPtr::new(shared as _),
        vtable: &SHARED_VTABLE,
    };
}

let shared = Box::from_raw(shared);
mem::forget(*shared);

shallow_clone_arc(actual as _, offset, len)
}
```

Rust 序列化反序列框架 Serde

serder version: 1.0 rust version 1.41 (1.31+)

参考

- [Github](#)
- [官方文档](#)

创建测试项目 `cargo new serde-learn --lib`

依赖 `Cargo.toml`

```
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

一、总览

1、简介

Serde 是 Rust 生态中最主流的 序列化、反序列化框架

设计上，基于 Rust 的静态类型系统和元编程（宏）的能力，使 Serde 序列化的执行速度与手写序列化器的速度相同。

使用上及其简单

- 用户 为自己的类型 `Serialize` 和 `Deserialize` 特质即可（大多数情况下使用过 `derive` 宏即可）
- 序列化提供商，提供 `Serializer` 和 `Deserializer` 特质的实现即可。

Serde 及社区提供主流的序列化协议的支持，具体如下

- [JSON](#), the ubiquitous JavaScript Object Notation used by many HTTP APIs.
- [Bincode](#), a compact binary format used for IPC within the Servo rendering engine.
- [CBOR](#), a Concise Binary Object Representation designed for small message size without the need for version negotiation.
- [YAML](#), a popular human-friendly configuration language that ain't markup language.
- [MessagePack](#), an efficient binary format that resembles a compact JSON.
- [TOML](#), a minimal configuration format used by Cargo.
- [Pickle](#), a format common in the Python world.
- [RON](#), a Rusty Object Notation.
- [BSON](#), the data storage and network transfer format used by MongoDB.
- [Avro](#), a binary format used within Apache Hadoop, with support for schema definition.
- [JSON5](#), A superset of JSON including some productions from ES5.

- [Postcard](#), a no_std and embedded-systems friendly compact binary format.
- [URL](#), the x-www-form-urlencoded format.
- [Envy](#), a way to deserialize environment variables into Rust structs. (deserialization only)
- [Envy Store](#), a way to deserialize AWS Parameter Store parameters into Rust structs. (deserialization only)
- [S-expressions](#), the textual representation of code and data used by the Lisp language family.

2、数据结构

Serde 对常见 Rust 标准库数据结构 提供了开箱即用的实现。例如: `String` , `&str` , `usize` , `Vec<T>` , `HashMap<K,V>` 。

对于自定义类型可以通过 派生宏 提供支持

`src/ch01_overview.rs`

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let point = Point { x: 1, y: 2 };

        let serialized = serde_json::to_string(&point).unwrap();

        println!("serialized = {}", serialized);

        let deserialized: Point = serde_json::from_str(&serialized).unwrap();
    }
}
```

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

二、Serde 数据模型

Serde数据模型是与 Rust 数据结构和数据格式进行交互的API。您可以将其视为Serde的类型系统，Serde 将 Rust 类型分为 29 种

- 针对需要序列化的类型，用户需要实现 `Serialize`：根据 Rust 类型调用参数 `Serializer` 上的方法，而 `Serializer` 的实现由序列化提供商提供
- 针对需要反序列化的类型，用户需要实现 `Deserialize`：根据 Rust 类型调用参数 `Serializer` 上的方法，传递一个实现了 `Visitor` 的类型

1、29 种类型

- 14 基础类型
 - `bool`
 - `i8, i16, i32, i64, i128`
 - `u8, u16, u32, u64, u128`
 - `f32, f64`
 - `char`
- `string`
 - 有长度标记的UTF-8 字节数据（不是 `\0` 结尾的形式），可能长度为0
 - 在序列化时，所有类型的字符串被同等处理。在反序列化时，有三种方案：`transient`, 拥有所有权, 和借用。参见《理解反序列化生命周期》，（Serde使用零拷贝技术）
- `byte array – [u8]`
 - 与字符串相似，在反序列化期间，字节数组可以是 `transient`, 拥有所有权, 和借用
- `option`
 - `None` 或者 `Value`
- `unit`
 - Rust 中 `()` 的类型，它表示不包含数据的匿名值

- unit_struct
 - 例如 `struct Unit` 或 `PhantomData<T>`，它表示不包含数据的命名值
- unit_variant
 - 例如 在 `enum E { A, B }` 中的 `E::A` 和 `E::B`
- newtype_struct
 - 例如 `struct Millimeters(u8)`
- newtype_variant
 - 例如 在 `enum E { N(u8) }` 中的 `E::N`
- seq
 - 可变大小的异质序列
 - 例如 `Vec<T>` 或者 `HashSet<T>`
 - 序列化时，长度在遍历之前可能是未知的。在反序列化时，通过 查看数据 可以得知长度
 - 注意，像 `vec![Value::Bool(true), Value::Char('c')]` 之类的同质Rust集合可以序列化为异构Serde seq，在这种情况下，包含Serde bool和Serde char。
- tuple
 - 大小静态可知的异质序列
 - 例如 `(u8,)` 或 `(String, u64, Vec<T>)` 或 `[u64; 10]`
 - 其长度在反序列化时就已知道，无需查看数据
- tuple_struct
 - 命名元组，例如 `struct Rgb(u8, u8, u8)`
- tuple_variant
 - 例如 在 `enum E { T(u8, u8) }` 中的 `E::T`
- map
 - 大小可变的异类键值对，例如 `BTreeMap <K, V>`。进行序列化时，在遍历所有条目之前，长度可能未知，也可能未知。反序列化时，通过 查看数据 可以得知长度
- struct
 - 静态大小的异构键值对，其中的键是编译时常量字符串，并且在反序列化时无需查看序列化数据即可知道
 - 例如 `struct S { r: u8, g: u8, b: u8 }`
- struct_variant
 - 例如 在 `enum E { S { r: u8, g: u8, b: u8 } }` 中的 `E::S`

2、映射到数据模型

对于大多数Rust类型，将它们映射到Serde数据模型中非常简单。例如，Rust `bool`类型对应于Serde的`bool`类型。Rust元组结构 `Rgb(u8, u8, u8)` 对应于Serde的元组结构类型。

但是不一定就是这种简单一一对应的映射，例如：

`std::ffi::OsString` 类型，在 Windows 和 Unix 下表现不一致。直觉上应该映射为 `string`，但是跨平台无法使用。因此更好的方案是：

```
enum OsString {  
    Unix(Vec<u8>),  
    Windows(Vec<u16>),  
}
```

三、派生宏

实例细节参见：https://github.com/serde-rs/serde/blob/master/test_suite/tests/test_gen.rs

1、实例

参见：一、总览 2、数据结构

2、属性

属性用于使用派生宏的一些配置，主要分为3类：

- 容器属性 Container attributes — 应用在 枚举 和 结构体
- Variant属性 Variant attributes — 应用在 枚举的 Variant 上
- 字段属性 Field attributes — 应用在 结构体 和 枚举 variant 的 字段上

```

#[derive(Serialize, Deserialize)]
#[serde(deny_unknown_fields)]
struct S {
    #[serde(default)]
    f: i32,
}

#[derive(Serialize, Deserialize)]
#[serde(rename = "e")]
enum E {
    #[serde(rename = "a")]
    A(String),
}

```

(1) 容器属性

- `#[serde(rename = "name")]`
 - 使用给定的名字而不是其Rust名
 - 同时允许如下写法
 - `#[serde(rename(serialize = "ser_name"))]`
 - `#[serde(rename(deserialize = "de_name"))]`
 - `#[serde(rename(serialize = "ser_name", deserialize = "de_name"))]`
- `#[serde(rename_all = "...")]`
 - 根据给定的大小写约定重命名所有字段（结构）或 variants（枚举）。`"..."` 的可选值为 `"lowercase"` , `"UPPERCASE"` , `"PascalCase"` , `"camelCase"` , `"snake_case"` , `"SCREAMING_SNAKE_CASE"` , `"kebab-case"` , `"SCREAMING-KEBAB-CASE"`
 - 同时允许如下写法
 - `#[serde(rename_all(serialize = "..."))]`
 - `#[serde(rename_all(deserialize = "..."))]`
 - `#[serde(rename_all(serialize = "...", deserialize = "..."))]`
- `#[serde(deny_unknown_fields)]`
 - 指定遇到未知字段时，在反序列化期间始终出错。
 - 默认情况下，对于诸如JSON之类的自描述格式，未知字段将被忽略
- `#[serde(tag = "type")]`
 - ? ?

- `#[serde(tag = "t", content = "c")]`
 - ??
- `#[serde(untagged)]`
 - ??
- `#[serde(bound = "T: MyTrait")]`
 - 序列化和反序列化的where子句表示。这将替换Serde推断的任何特征范围。
 - 同时允许如下写法
 - `#[serde(bound(serialize = "T: MySerTrait"))]`
 - `#[serde(bound(deserialize = "T: MyDeTrait"))]`
 - `#[serde(bound(serialize = "T: MySerTrait", deserialize = "T: MyDeTrait"))]`
- `#[serde(default)]`
 - 反序列化时，从结构的 Default 实现中 填写所有缺少的字段
 - 仅允许在 struct 中使用
- `#[serde(default = "path")]`
 - 反序列化时，使用给定函数或方法返回的对象中填写所有缺少的字段。该函数声明为 `fn() -> T`。
 - 例如，`default = "my_default"` 将调用 `my_default()`，而 `default = "SomeTrait::some_default"` 将调用 `SomeTrait::some_default()`
 - 仅允许在 struct 中使用
- `#[serde(remote = "...")]`
 - ??
- `#[serde(transparent)]`
 - 只允许应用在只有一个字段枚举或者结构体上
 - 表示只序列化内部对象，不要外部支撑，比如

```
#[derive(Serialize, Deserialize, Debug)]
#[serde(transparent)]
struct Axis {
    x: i32
}
```
- `#[serde(from = "FromType")]`
 - 反序列化为 `FromType` 后，转换为 该类型

- 使用条件
 - 此类型必须实现 `From<FromType>`
 - 且 `FromType` 必须实现 `Deserialize`
- `#[serde(try_from = "FromType")]`
 - 反序列化为 `FromType` 后，转换为 该类型
 - 使用条件
 - 此类型必须实现 `TryFrom<FromType>`， 其中错误类型必须实现 `Display`
 - 且 `FromType` 必须实现 `Deserialize`
- `#[serde(into = "IntoType")]`
 - 序列化之前，现将该类型转换为 `IntoType` 类型，然后再序列化
 - 使用条件
 - 此类型必须实现 `Clone` 和 `Into<IntoType>`
 - 且 `IntoType` 必须实现 `Serialize`
- `#[serde(crate = "...")]`
 - 指定从生成的代码中，引用 Serde API 时，要使用的 Serde crate实例的路径。
 - 这通常仅适用于从不同板条箱中的公共宏调用重新导出的Serde

(2) Variant属性

- `#[serde(rename = "name")]`
 - 使用给定的名字而不是其Rust名
 - 同时允许如下写法
 - `#[serde(rename(serialize = "ser_name"))]`
 - `#[serde(rename(deserialize = "de_name"))]`
 - `#[serde(rename(serialize = "ser_name", deserialize = "de_name"))]`
- `#[serde(alias = "name")]`
 - 反序列化时，对应的别名
 - 允许配置多个
- `#[serde(rename_all = "...")]`

- 根据给定的大小写约定重命名 struct variant。`"..."` 的可选值为 `lowercase` , `UPPERCASE` , `PascalCase` , `camelCase` , `snake_case` , `SCREAMING_SNAKE_CASE` , `kebab-case` , `SCREAMING-KEBAB-CASE`
- 同时允许如下写法
 - `#[serde(rename_all(serialize = "..."))]`
 - `#[serde(rename_all(deserialize = "..."))]`
 - `#[serde(rename_all(serialize = "...", deserialize = "..."))]`
- `#[serde(skip)]`
 - 跳过序列化或反序列化此 variant
 - 尝试序列化时将报错
 - 尝试反序列化时将报错
- `#[serde(skip_serializing)]`
 - 尝试序列化时将报错
- `#[serde(skip_deserializing)]`
 - 尝试反序列化时将报错
- `#[serde(serialize_with = "path")]`
 - 使用 `path` 函数 进行序列化，该序列化函数声明为：`fn<S>(&FIELD0, &FIELD1, ..., S) -> Result<S::Ok, S::Error>` where `S: Serializer`
- `#[serde(deserialize_with = "path")]`
 - 使用 `path` 函数 进行反序列化，该序列化函数声明为：`fn<'de, D>(D) -> Result<FIELDS, D::Error>` where `D: Deserializer<'de>`
- `#[serde(with = "module")]`
 - 指定 `#[serde(serialize_with = "path")]` 和 `#[serde(deserialize_with = "path")]` path 所在 module
- `#[serde(bound = "T: MyTrait")]`
 - 序列化和反序列化的where子句表示。这将替换Serde推断的任何特征范围。
 - 同时允许如下写法
 - `#[serde(bound(serialize = "T: MySerTrait"))]`
 - `#[serde(bound(deserialize = "T: MyDeTrait"))]`
 - `#[serde(bound(serialize = "T: MySerTrait", deserialize = "T: MyDeTrait"))]`
- `#[serde(borrow)]` 和 `#[serde(borrow = "'a + 'b + ...')]`
 - ? ?

- `#[serde(other)]`
 - `? ?` 为匹配其他类型

(3) 字段属性

- `#[serde(rename = "name")]`
 - 使用给定的名字而不是其Rust名
 - 同时允许如下写法
 - `#[serde(rename(serialize = "ser_name"))]`
 - `#[serde(rename(deserialize = "de_name"))]`
 - `#[serde(rename(serialize = "ser_name", deserialize = "de_name"))]`
- `#[serde(alias = "name")]`
 - 反序列化时，对应的别名
 - 允许配置多个
- `#[serde(default)]`
 - 如果反序列化时不存在该值，则使用 `Default::default()`
- `#[serde(default = "path")]`
 - 反序列化时，使用给定函数或方法返回的对象中填写所有缺少的字段。该函数声明为 `fn() -> T`。
 - 例如，`default = "my_default"` 将调用 `my_default()`，而 `default = "SomeTrait::some_default"` 将调用 `SomeTrait::some_default()`
- `#[serde(flatten)]`
 - 展平该字段，也就是将该字段内部抽到当前结构
- `#[serde(skip)]`
 - 跳过此字段：不序列化或反序列化
 - 反序列化时，Serde将使用 `Default::default()` 或 `default = "..."` 生成该值
- `#[serde(skip_serializing)]`
 - 跳过序列化该字段
- `#[serde(skip_deserializing)]`
 - 反序列化时跳过该字段

- 反序列化时，Serde将使用 `Default::default()` 或 `default = "..."` 生成该值
- `#[serde(skip_serializing_if = "path")]`
 - 调用一个函数以确定是否跳过序列化此字段。
 - 该函数声明为 `fn(&T) -> bool`。例如 `skip_serializing_if = "Option::is_none"`
- `#[serde(serialize_with = "path")]`
 - 使用 `path` 函数进行序列化，函数声明为： `fn<S>(&T, S) -> Result<S::Ok, S::Error>` where `S: Serializer`
 - 这样 `T` 就不需要事先 `Serialize`
- `#[serde(deserialize_with = "path")]`
 - 使用 `path` 函数进行反序列化，该序列化函数声明为： `fn<'de, D>(D) -> Result<T, D::Error>` where `D: Deserializer<'de>`
 - 这样 `T` 就不需要事先 `Serialize`
- `#[serde(with = "module")]`
 - 指定 `#[serde(serialize_with = "path")]` 和 `#[serde(deserialize_with = "path")]` `path` 所在 module
- `#[serde(borrow)]` 和 `#[serde(borrow = "'a + 'b + ...'")]`
 - ？？？
- `#[serde(bound = "T: MyTrait")]`
 - 序列化和反序列化的where子句表示。这将替换Serde推断的任何特征范围。
 - 同时允许如下写法
 - `#[serde(bound(serialize = "T: MySerTrait"))]`
 - `#[serde(bound(deserialize = "T: MyDeTrait"))]`
 - `#[serde(bound(serialize = "T: MySerTrait", deserialize = "T: MyDeTrait"))]`
- `#[serde(getter = "...")]`
 - ???

四、自定义序列化/反序列化

Serde的通过 `#[derive(Serialize, Deserialize)]` 派生宏为结构体和枚举提供了合理的默认序列化行为，并且可以使用属性在某种程度上进行自定义。

但是对于特殊需求，Serde可以通过为您的类型手动实现 `Serialize` 和 `Deserialize` 特质来完全自定义序列化行为。这两个特质都有只有一个方法，声明如下

```
pub trait Serialize {  
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
    where  
        S: Serializer;  
}  
  
pub trait Deserialize<'de>: Sized {  
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>  
    where  
        D: Deserializer<'de>;  
}
```

而序列化协议只需要提供 `Serializer` 和 `Deserializer` 的实现即可。

创建测试代码 `src/ch04_custom_serde.rs`

1、实现 `Serialize`

本节参考源码：[serde/src/ser/impls.rs](#)

特质声明如下

```
pub trait Serialize {  
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>  
    where  
        S: Serializer;  
}
```

该方法的工作是：

- 通过调用给定 `Serializer` 参数上的一种方法并传递 `&self`，将 `self` 映射到 Serde数据模型中

(1) 序列化基础数据类型

```

impl Serialize for i32 {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        serializer.serialize_i32(*self)
    }
}

```

此处仅为示例，Serde 已为所有基础数据类型实现了类似的实现（以上代码已经在用户 crate 无法编译，因为不满足孤儿规则）

(2) 序列化序列或者map

复杂类型序列化一般需要三步：初始化，放置元素，结束。

```
use serde::ser::{Serialize, Serializer, SerializeSeq, SerializeMap};
```

```

impl<T> Serialize for Vec<T>
where
    T: Serialize,
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut seq = serializer.serialize_seq(Some(self.len()))?;
        for e in self {
            seq.serialize_element(e)?;
        }
        seq.end()
    }
}

```

```

impl<K, V> Serialize for MyMap<K, V>
where
    K: Serialize,
    V: Serialize,
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut map = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            map.serialize_entry(k, v)?;
        }
        map.end()
    }
}

```

(以上代码已经在 用户 crate 编译)

(3) 序列化元组

通过阅读源码可知，Rust 为 长度为 0~32 的数组 和 长度为 0~16 的元组提供了实现 Serialize 实现

参见源码 [serde/src/ser/impls.rs](#)

(4) 序列化结构体

```
use serde::{Serialize, Serializer};
use serde::ser::{SerializeStruct, SerializeTupleStruct, SerializeStructVariant, SerializeTupleVariant};

struct Color {
    r: u8,
    g: u8,
    b: u8,
}

impl Serialize for Color {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut state = serializer.serialize_struct("Color", 3)?;
        state.serialize_field("r", &self.r)?;
        state.serialize_field("g", &self.g)?;
        state.serialize_field("b", &self.b)?;
        state.end()
    }
}

struct Point2D(f64, f64);

impl Serialize for Point2D {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut state = serializer.serialize_tuple_struct("Point2D", 2)?;
        state.serialize_field(&self.0)?;
        state.serialize_field(&self.1)?;
        state.end()
    }
}

struct Inches(u64);
```

```

impl Serialize for Inches {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        serializer.serialize_newtype_struct("Inches", &self.0)
    }
}

struct Instance;

impl Serialize for Instance {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        serializer.serialize_unit_struct("Instance")
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ch04() {
        println!("Color = {}", serde_json::to_string(&Color { r: 1, g: 2, b: 3 }).unwrap());
        println!("Point2D = {}", serde_json::to_string(&Point2D(1.0, 2.0)).unwrap());
        println!("Inches = {}", serde_json::to_string(&Inches(1)).unwrap());
        println!("Instance = {}", serde_json::to_string(&Instance).unwrap());
    }
}

```

(5) 序列化枚举

```

use serde::{Serialize, Serializer};
use serde::ser::{SerializeStruct, SerializeTupleStruct, SerializeStructVariant, SerializeTupleVariant};

#[allow(unused, dead_code)]
enum E {
    Color { r: u8, g: u8, b: u8 },
    Point2D(f64, f64),
}

```

```

Inches(u64),

    Instance,
}

impl Serialize for E {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        match self {
            E::Color {r,g,b} => {
                let mut state = serializer.serialize_struct_variant("E", 0, "Color", 3)?;
                state.serialize_field("r", r)?;
                state.serialize_field("g", g)?;
                state.serialize_field("b", b)?;
                state.end()
            }
            E::Point2D(x, y) => {
                let mut state = serializer.serialize_tuple_variant("E", 1, "Point2D", 2)?;
                state.serialize_field(x)?;
                state.serialize_field(y)?;
                state.end()
            }
            E::Inches(i) => serializer.serialize_newtype_variant("E", 2, "Inches", i),
            E::Instance => serializer.serialize_unit_variant("E", 3, "Instance"),
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ch04() {
        println!("E::Color = {}", serde_json::to_string(&E::Color{r:1, g:2, b:3}).unwrap());
        println!("E::Point2D = {}", serde_json::to_string(&E::Point2D(1.0, 2.0)).unwrap());
        println!("E::Inches = {}", serde_json::to_string(&E::Inches(1)).unwrap());
        println!("E::Instance = {}", serde_json::to_string(&E::Instance).unwrap());
    }
}

```

(6) 序列化字节数组

由于 [RFC 1210 specialization](#) 尚未稳定。

因此 `[u8]` 和 `Vec<u8>` 分别与 `[T]` 和 `Vec<T>` 重复产生定义。

因此在 `serde` 库中，没有对 `serializer.serialize_bytes(self)` 的使用

因此 Serde 创建了一个专为字节数组实现 `Serialize` 类型的库 `serde_bytes`，以提高字节流的序列化/反序列化效率。

使用方式如下

`Cargo.toml`

`src/ch04_custom_serde.rs`

```
#[derive(Serialize)]
struct Efficient<'a> {
    #[serde(with = "serde_bytes")]
    bytes: &'a [u8],
    byte_buf: Vec<u8>,
}

struct Efficient2<'a> {
    bytes: &'a [u8],
    byte_buf: Vec<u8>,
}

impl <'a> Serialize for Efficient2<'a> {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut state = serializer.serialize_struct("Efficient2", 2)?;
        state.serialize_field("bytes", {

            struct SerializeWith<'__a, 'a: '__a> {
                values: (&'__a &'a [u8],),
                phantom: serde::export::PhantomData<Efficient2<'a>>,
            }
            impl<'__a, 'a: '__a> serde::Serialize for SerializeWith<'__a, 'a> {
                fn serialize<__S>(
                    &self,
                    __S: __S,
                ) -> serde::export::Result<__S::Ok, __S::Error>
                where
                    __S: serde::Serializer,
                {
                    serde_bytes::serialize(self.values.0, __S)
                }
            }
            &SerializeWith {

```

```
    values: (&self.bytes,),
    phantom: serde::export::PhantomData::<Efficient2<'a>>,
}
})?;
state.serialize_field("byte_buf", &self.byte_buf)?;
state.end()
}
}
```

(7) 序列化Option

针对 `Option` 枚举 和 `#[derive(Serialize)]` 普通枚举的代码实现逻辑不同：

- 针对 `Some(value)` 使用 `serializer.serialize_some(value)` 进行序列化，在JSON中将返回 value 的序列化
- 针对 `None` 使用 `serializer.serialize_none()` 进行序列化，在JSON中将返回null

2、实现 Deserialize

本节参考源码：

- [serde/src/de/mod.rs](#)

特质声明如下

```
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>;
}
```

该方法的工作是：

- 通过调用给定 `Deserializer` 参数上的一种方法，并传递一个实现了 `Visitor` 的实例

(1) 反序列化基础数据类型

```
use std::fmt;

use serde::de::{self, Visitor};

struct I32Visitor;

impl<'de> Visitor<'de> for I32Visitor {
    type Value = i32;

    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        formatter.write_str("an integer between -2^31 and 2^31")
    }

    fn visit_i8<E>(self, value: i8) -> Result<Self::Value, E>
    where
        E: de::Error,
    {
        Ok(i32::from(value))
    }

    fn visit_i32<E>(self, value: i32) -> Result<Self::Value, E>
    where
        E: de::Error,
    {
        Ok(value)
    }

    fn visit_i64<E>(self, value: i64) -> Result<Self::Value, E>
    where
        E: de::Error,
    {
        use std::i32;
        if value >= i64::from(i32::MIN) && value <= i64::from(i32::MAX) {
            Ok(value as i32)
        } else {
            Err(E::custom(format!("i32 out of range: {}", value)))
        }
    }
}

impl<'de> Deserialize<'de> for i32 {
    fn deserialize<D>(deserializer: D) -> Result<i32, D::Error>
    where
        D: Deserializer<'de>,
    {
```

```

    deserializer.deserialize_i32(I32Visitor)
}
}

```

此处仅为示例，Serde 已为所有基础数据类型实现了类似的实现（以上代码已经在 用户 crate 无法编译，因为不满足孤儿规则）

需要注意的是

- Visitor特质还有许多方法未实现的方法，如果反序列化协议提供商调用这些方法，将返回类型错误。
- 例如，I32Visitor没有实现 `Visitor::visit_map`，因此在输入包含map时尝试反序列化i32是类型错误。
- 反序列化器不一定会跟随类型提示，因此对`deserialize_i32`的调用不一定意味着反序列化器将调用`I32Visitor :: visit_i32`。（因为不一定序列化协议都支持如此之多的数据类型）例如，JSON将所有带符号的整数类型都视为相同。JSON反序列化器将为任何带符号的整数调用`visit_i64`，为任何无符号的整数调用`visit_u64`，即使提示使用其他类型。

(2) 反序列化Map

```
struct MyMap<K, V>(PhantomData<K>, PhantomData<V>);
```

```

impl<K, V> MyMap<K, V> {
    fn with_capacity(c: usize) -> Self {
        println!("build MyMap size = {}", c);
        MyMap(PhantomData, PhantomData)
    }

    fn insert(&mut self, _: K, _: V) {
        println!("call MyMap insert")
    }
}

```

```
struct MyMapViewVisitor<K, V> {
    marker: PhantomData<fn() -> MyMap<K, V>>
}
```

```

impl<K, V> MyMapViewVisitor<K, V> {
    fn new() -> Self {
        MyMapViewVisitor {
            marker: PhantomData
        }
    }
}

```

```
impl<'de, K, V> Visitor<'de> for MyMapViewVisitor<K, V>
where
```

K: Deserialize<'de>,

```

V: Deserialize<'de>,
{
    type Value = MyMap<K, V>;

    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        formatter.write_str("a very special map")
    }

    fn visit_map<M>(self, mut access: M) -> Result<Self::Value, M::Error>
    where
        M: MapAccess<'de>,
    {
        let mut map = MyMap::with_capacity(access.size_hint().unwrap_or(0));

        while let Some((key, value)) = access.next_entry()? {
            map.insert(key, value);
        }

        Ok(map)
    }
}

impl<'de, K, V> Deserialize<'de> for MyMap<K, V>
where
    K: Deserialize<'de>,
    V: Deserialize<'de>,
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>,
    {
        deserializer.deserialize_map(MyMapViewVisitor::new())
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ch04() {

        let json1 = r#"
        {
            "k1": "v1",
            "k2": "v2"
        }#
        let _r1: MyMap<String, String> = serde_json::from_str(json1).unwrap();
    }
}

```

(3) 反序列化结构

```
use std::fmt;

use serde::de::{self, Deserialize, Deserializer, Visitor, SeqAccess, MapAccess};

#[derive(Debug)]
struct Duration {
    secs: u64,
    nanos: u32,
}

impl Duration {
    fn new(secs: u64, nanos: u32) -> Duration {
        Duration {
            secs,
            nanos
        }
    }
}

impl<'de> Deserialize<'de> for Duration {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>,
    {
        enum Field { Secs, Nanos };
}

impl<'de> Deserialize<'de> for Field {
    fn deserialize<D>(deserializer: D) -> Result<Field, D::Error>
    where
        D: Deserializer<'de>,
    {
        struct FieldVisitor;

        impl<'de> Visitor<'de> for FieldVisitor {
            type Value = Field;

            fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
                formatter.write_str("`secs` or `nanos`")
            }

            fn visit_str<E>(self, value: &str) -> Result<Field, E>
            where
                E: de::Error,
            {
                match value {
                    "secs" => Ok(Field::Secs),
                    "nanos" => Ok(Field::Nanos),
                    _ => Err(de::Error::unknown_field(value, FIELDS)),
                }
            }
        }
    }
}
```

```

        }
    }
}

deserializer.deserialize_identifier(FieldVisitor)
}
}

struct DurationVisitor;

impl<'de> Visitor<'de> for DurationVisitor {
    type Value = Duration;

    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        formatter.write_str("struct Duration")
    }

    fn visit_seq<V>(self, mut seq: V) -> Result<Duration, V::Error>
    where
        V: SeqAccess<'de>,
    {
        let secs = seq.next_element()?;
        .ok_or_else(|| de::Error::invalid_length(0, &self))?;
        let nanos = seq.next_element()?;
        .ok_or_else(|| de::Error::invalid_length(1, &self))?;
        Ok(Duration::new(secs, nanos))
    }

    fn visit_map<V>(self, mut map: V) -> Result<Duration, V::Error>
    where
        V: MapAccess<'de>,
    {
        let mut secs = None;
        let mut nanos = None;

        while let Some(key) = map.next_key::<Field>()? {
            match key {
                Field::Secs => {
                    if secs.is_some() {
                        return Err(de::Error::duplicate_field("secs"));
                    }
                    secs = Some(map.next_value()?);
                }
                Field::Nanos => {
                    if nanos.is_some() {
                        return Err(de::Error::duplicate_field("nanos"));
                    }
                    nanos = Some(map.next_value()?);
                }
            }
        }

        let secs = secs.ok_or_else(|| de::Error::missing_field("secs"))?;
        let nanos = nanos.ok_or_else(|| de::Error::missing_field("nanos"))?;
        Ok(Duration::new(secs, nanos))
    }
}

```

```
const FIELDS: &'static [&'static str] = &["secs", "nanos"];
deserializer.deserialize_struct("Duration", FIELDS, DurationVisitor)
}

}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ch04() {

        let json2 = r#"
{
    "secs": 1,
    "nanos": 2
}
"#;
        let r2: Duration = serde_json::from_str(json2).unwrap();
        println!("Duration = {:?}", r2);
    }
}
```

3、单元测试

参见 <https://serde.rs/unit-testing.html>

五、实现序列化器/反序列化器（Serializer/Deserializer）

参见： <https://serde.rs/data-format.html>

六、反序列化器生命周期

参见： <https://serde.rs/lifetimes.html>

七、no-std 支持

参见： <https://serde.rs/no-std.html>

八、feature 列表

参见：<https://serde.rs/feature-flags.html>

优秀项目解析--serde

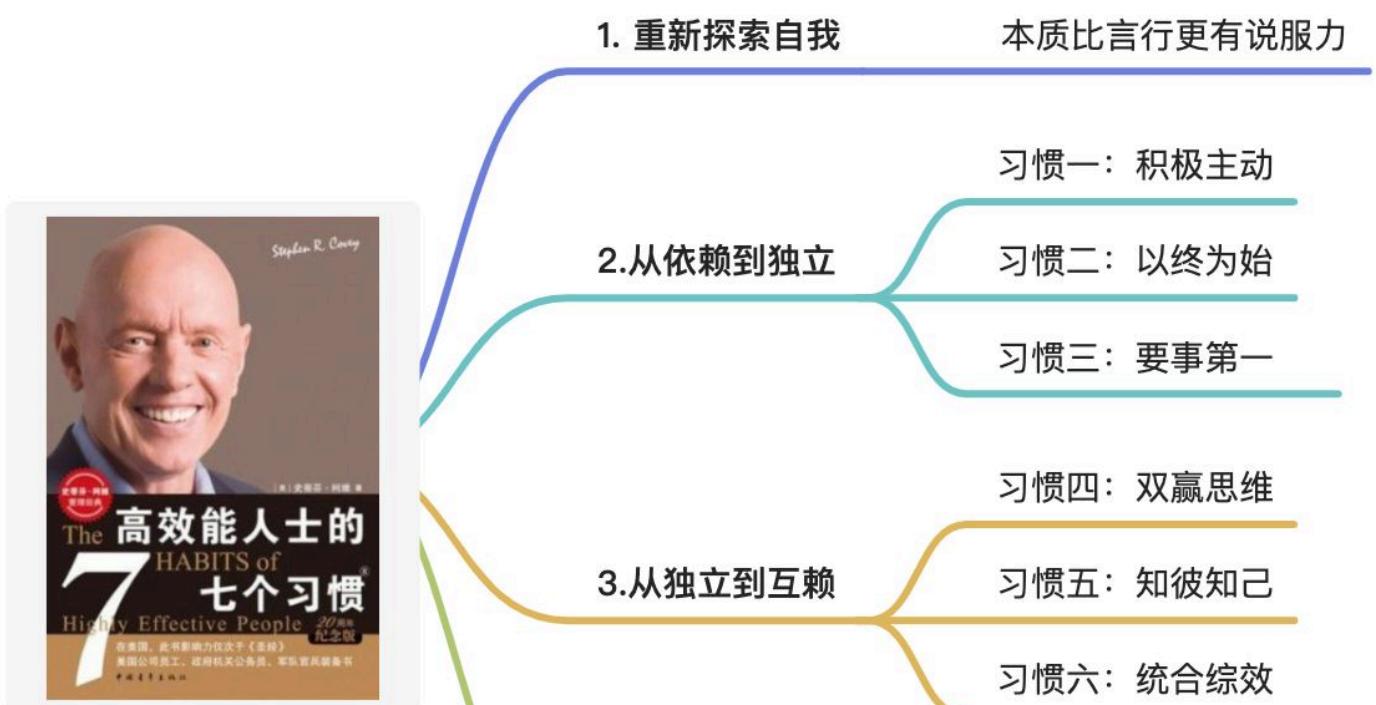
根据 [遗忘曲线](#)：如果没有记录和回顾，6天后便会忘记75%的内容

读书笔记正是帮助你记录和回顾的工具，不必拘泥于形式，其核心是：记录、翻看、思考

| | |
|----|--|
| 书名 | 高效能人士的七个习惯 |
| 作者 | 史蒂芬·柯维 |
| 状态 | 待开始 阅读中 已读完 |
| 简介 | 本书精选柯维博士“七个习惯”的最核心思想和方法，为忙碌人士带来超价值的自我提升体验。用最少的时间，参透高效能人士的持续成功之路。 |

思维导图

用思维导图，结构化记录本书的核心观点。



读后感

观点1

读完该书后，受益的核心观点与说明...

观点2

读完该书后，受益的核心观点与说明...

观点3

读完该书后，受益的核心观点与说明...

书摘

- 该书的金句摘录...
- 该书的金句摘录...
- 该书的金句摘录...

相关资料

可通过“+K”插入引用链接，或使用“本地文件”引入源文件。

<https://book.douban.com/subject/5325618/>

Rust mio库源码情景分析

mio 是 Metal IO，Rust 语言生态比较底层的 I/O 库，官网的介绍：

Mio is a lightweight I/O library for Rust with a focus on adding as little overhead as possible over the OS abstractions.

mio 目前已经发布了 v0.6.19 版本，这次分析代码版本选择 master 分支，commit id
14f37f283576040c8763f45de6c2b2bbcb82436d

我们从官方自带的 example 进行源码跟踪分析。

```
use std::collections::HashMap;
use std::io::{self, Read, Write};
use std::str::from_utf8;

use mio::event::Event;
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interests, Poll, Token};

// Setup some tokens to allow us to identify which event is for which socket.
const SERVER: Token = Token(0);
```

```

// Some data we'll send over the connection.
const DATA: &[u8] = b"Hello world!\n";

fn main() -> io::Result<()> {
    env_logger::init();

    // Create a poll instance.
    let mut poll = Poll::new()?;
    // Create storage for events.
    let mut events = Events::with_capacity(128);

    // Setup the TCP server socket.
    let addr = "127.0.0.1:13265".parse().unwrap();
    let server = TcpListener::bind(addr)?;

    // Register the server with poll we can receive events for it.
    poll.registry()
        .register(&server, SERVER, Interests::READABLE)?;

    // Map of `Token` -> `TcpStream`.
    let mut connections = HashMap::new();
    // Unique token for each incoming connection.
    let mut unique_token = Token(SERVER.0 + 1);

    println!("You can connect to the server using `nc`:");
    println!(" $ nc 127.0.0.1 13265");
    println!("You'll see our welcome message and anything you type we'll be printed here.");

    loop {
        poll.poll(&mut events, None)?;

        for event in events.iter() {
            match event.token() {
                SERVER => {
                    // Received an event for the TCP server socket.
                    // Accept an connection.
                    let (connection, address) = server.accept()?;
                    println!("Accepted connection from: {}", address);

                    let token = next(&mut unique_token);
                    poll.registry().register(
                        &connection,
                        token,
                        Interests::READABLE.add(Interests::WRITABLE),
                    )?;

                    connections.insert(token, connection);
                }
                token => {
                    // (maybe) received an event for a TCP connection.
                    let done = if let Some(connection) = connections.get_mut(&token) {
                        handle_connection_event(&mut poll, connection, event)?
                    } else {
                        // Sporadic events happen.
                        false
                    };
                    if done {
                
```

```

            connections. remove (&token);
        }
    }
}
}

fn next (current: & mut Token) -> Token {
    let next = current. 0 ;
    current. 0 += 1;
    Token (next)
}

/// Returns `true` if the connection is done.
fn handle_connection_event (
    poll: & mut Poll,
    connection: & mut TcpStream,
    event: &Event,
) -> io:: Result < bool > {
    if event. is_writable () {
        // We can (maybe) write to the connection.
        match connection. write (DATA) {
            // We want to write the entire `DATA` buffer in a single go. If we
            // write less we'll return a short write error (same as
            // `io::Write::write_all` does).
            Ok (n) if n < DATA. len () => return Err (io::ErrorKind::WriteZero. into ()),
            Ok (_) => {
                // After we've written something we'll reregister the connection
                // to only respond to readable events.
                poll. registry ()
                    . reregister (&connection, event. token (), Interests::READABLE)?
            }
            // Would block "errors" are the OS's way of saying that the
            // connection is not actually ready to perform this I/O operation.
            Err (ref err) if would_block (err) => {}
            // Got interrupted (how rude!), we'll try again.
            Err (ref err) if interrupted (err) => {
                return handle_connection_event (poll, connection, event)
            }
            // Other errors we'll consider fatal.
            Err (err) => return Err (err),
        }
    }
}

if event. is_readable () {
    let mut connection_closed = false ;
    let mut received_data = Vec :: with_capacity ( 4096 );
    // We can (maybe) read from the connection.
    loop {
        let mut buf = [ 0 ; 256 ];
        match connection. read (& mut buf) {
            Ok ( 0 ) => {
                // Reading 0 bytes means the other side has closed the
                // connection or is done writing, then so are we.
                connection_closed = true ;
                break ;
            }
        }
    }
}

```

```

        Ok (n) => received_data. extend_from_slice (&buf[..n]),
        // Would block "errors" are the OS's way of saying that the
        // connection is not actually ready to perform this I/O operation.
        Err (ref err) if would_block (err) => break ,
        Err (ref err) if interrupted (err) => continue ,
        // Other errors we'll consider fatal.
        Err (err) => return Err (err),
    }
}

if let Ok (str_buf) = from_utf8 (&received_data) {
    println! ("Received data: {}", str_buf. trim_end ());
} else {
    println! ("Received (none UTF-8) data: {:?}", &received_data);
}

if connection_closed {
    println! ("Connection closed");
    return Ok (true);
}
}

Ok (false)
}

fn would_block (err: &io::Error) -> bool {
    err. kind () == io::ErrorKind::WouldBlock
}

fn interrupted (err: &io::Error) -> bool {
    err. kind () == io::ErrorKind::Interrupted
}

```

从main()方法体开始看：

Poll::new()

```

pub fn new () -> io:: Result <Poll> {
    sys::Selector:: new (). map (|selector| Poll {
        registry: Registry { selector },
    })
}

```

看一下 Poll 的结构体

```

pub struct Poll {
    registry: Registry,
}

/// Registers I/O resources.
pub struct Registry {
    selector: sys::Selector,
}

```

```
}
```

sys::Selector 在不同的操作系统上有不同的实现：

```
/// `Poll` is backed by the selector provided by the operating system.  
///  
/// | OS      | Selector |  
/// |-----|-----|  
/// | Android | [epoll] |  
/// | DragonFly BSD | [kqueue] |  
/// | FreeBSD   | [kqueue] |  
/// | Linux     | [epoll] |  
/// | NetBSD    | [kqueue] |  
/// | OpenBSD   | [kqueue] |  
/// | Solaris   | [epoll] |  
/// | Windows   | [IOCP]  |  
/// | iOS       | [kqueue] |  
/// | macOS     | [kqueue] |  
///
```

我们挑选 Linux 的 epoll 跟踪。源文件在 /src/sys/unix/epoll.rs

```
impl Selector {  
    pub fn new() -> io::Result<Selector> {  
        // According to libuv `EPOLL_CLOEXEC` is not defined on Android API <  
        // 21. But `EPOLL_CLOEXEC` is an alias for `O_CLOEXEC` on all platforms,  
        // so we use that instead.  
        syscall!(epoll_create1(libc::O_CLOEXEC)).map(|ep| Selector {  
            #[cfg(debug_assertions)]  
            id: NEXT_ID.fetch_add(1, Ordering::Relaxed),  
            ep,  
        })  
    }  
  
    ...  
}
```

这段代码调用了 linux 的 api epoll_create1() 该接口返回一个 int 值，表示指向 epoll 实例的文件描述符。就是代码中的ep。还涉及到id 自增。NEXT_ID 的定义：

```
/// Unique id for use as `Selectorld`.  
#[cfg(debug_assertions)]  
static NEXT_ID: AtomicUsize = AtomicUsize::new(1);
```

看一下syscall!宏。定义的文件在 /src/sys/unix/mod.rs

```
// Macro must be defined before any modules that uses them.  
macro_rules! syscall {  
    ($fn : ident ($($arg: expr),* $(),*) ) => {  
        let res = unsafe { libc::$fn($($arg,)* ) };  
        if res == -1 {  
            Err(io::Error::last_os_error())  
        } else {
```

```
        } } ;  
    } } ;
```

就是指定函数名和实际参数，调用libc下的函数。

TcpListener

接着是 `let server = TcpListener::bind(addr)?;` 看一看这个bind方法

```
    /// 1. Create a new TCP socket.  
    /// 2. Set the `SO_REUSEADDR` option on the socket on Unix.  
    /// 3. Bind the socket to the specified address.  
    /// 4. Calls `listen` on the socket to prepare it to receive new connections.  
  
    pub fn bind(addr: SocketAddr) -> io::Result<TcpListener> {  
        sys::TcpListener::bind(addr).map(|sys| TcpListener {  
            sys,  
            #[cfg(debug_assertions)]  
            selector_id: SelectorId::new(),  
        })  
    }
```

注释说明该方法完成了 socket 编程的 4 个步骤。由于是调用 `sys::TcpListener::bind()`, 跟进去看

```
pub fn bind(addr: SocketAddr) -> io::Result<TcpListener> {
    new_ip_socket(addr, libc::SOCK_STREAM).and_then(|socket| {
        // Set SO_REUSEADDR (mirrors what libstd does).
        syscall!(setsockopt(
            socket,
            libc::SOL_SOCKET,
            libc::SO_REUSEADDR,
            &1 as *const libc::c_int as *const libc::c_void,
            size_of::<libc::c_int>() as libc::socklen_t,
        ))
        .and_then(|_| {
            let (raw_addr, raw_addr_length) = socket_addr(&addr);
            syscall!(bind(socket, raw_addr, raw_addr_length))
        })
        .and_then(|_| syscall!(listen(socket, 1024)))
        .map_err(|err| {
            // Close the socket if we hit an error, ignoring the error
            // from closing since we can't pass back two errors.
            let _ = unsafe { libc::close(socket) };
            err
        })
        .map(|_| TcpListener {
            inner: unsafe { net::TcpListener::from_raw_fd(socket) },
        })
    })
}
```

回到主方法继续看

poll.registry().register()

```
pub fn registry(& self) -> &Registry {
    & self.registry
}
```

真正完成 register 动作的是`&self.registry`

```
pub fn register <S>(& self, source: &S, token: Token, interests: Interests) -> io::Result <()>
where
    S: event::Source + ?Sized,
{
    trace!(
        "registering event source with poller: token={:?}", interests,
        token,
        interests
    );
    source.register(& self, token, interests)
}
```

又调用了`source.register()`, 看一看`source trait` 的定义:

```
pub trait Source {
    /// Register `self` with the given `Registry` instance.
    ///
    /// This function should not be called directly. Use [`Registry::register`]
    /// instead. Implementors should handle registration by delegating the call
    /// to another `Source` type.
    ///
    /// [`Registry::register`]: crate::Registry::register
    fn register(& self, registry: &Registry, token: Token, interests: Interests) -> io::Result <()>;
    /// Re-register `self` with the given `Registry` instance.
    ///
    /// This function should not be called directly. Use
    /// [`Registry::reregister`] instead. Implementors should handle
    /// re-registration by either delegating the call to another `Source` type.
    ///
    /// [`Registry::reregister`]: crate::Registry::reregister
    fn reregister(& self, registry: &Registry, token: Token, interests: Interests) -> io::Result <()>;
    /// Deregister `self` from the given `Registry` instance.
    ///
    /// This function should not be called directly. Use
    /// [`Registry::deregister`] instead. Implementors should handle
    /// deregistration by delegating the call to another `Source` type.
    ///
    /// [`Registry::deregister`]: crate::Registry::deregister
    fn deregister(& self, registry: &Registry) -> io::Result <()>;
}
```

三个方法，注册、再次注册、反注册。

example 里传入的 source 是 &server: TcpListener。

```
impl event ::Source for TcpListener {
    fn register (& self , registry: &Registry, token: Token, interests: Interests) -> io::Result <()> {
        #[cfg(debug_assertions)]
        self .selector_id. associate_selector (registry)?;
        self .sys. register (registry, token, interests)
    }

    fn reregister (
        & self ,
        registry: &Registry,
        token: Token,
        interests: Interests,
    ) -> io::Result <()> {
        self .sys. reregister (registry, token, interests)
    }

    fn deregister (& self , registry: &Registry) -> io::Result <()> {
        self .sys. deregister (registry)
    }
}
```

其中register()主要涉及到：

```
self .selector_id. associate_selector (registry)?;

self .sys. register (registry, token, interests)
```

先看第一个。

```
impl SelectorId {
    pub fn new () -> SelectorId {
        SelectorId {
            id: AtomicUsize:: new ( 0 ),
        }
    }

    pub fn associate_selector (& self , registry: &Registry) -> io::Result <()> {
        let selector_id = self .id. load (Ordering::SeqCst);

        if selector_id != 0 && selector_id != registry.selector. id () {
            Err (io::Error:: new (
                io::ErrorKind::Other,
                "socket already registered",
            ))
        } else {
            self .id. store (registry.selector. id (), Ordering::SeqCst);
            Ok(())
        }
    }
}
```

```
}
```

这里就是把`registry.selector.id()`赋值给`SelectorId.id`。注意后者是`AtomicUsize`。

看第二个。看看`sys`的类型：

```
pub struct TcpListener {
    sys: sys::TcpListener,
    #[cfg(debug_assertions)]
    selector_id: SelectorId,
}
```

`sys::TcpListener`的`register`方法：

```
fn register (& self, registry: &Registry, token: Token, interests: Interests) -> io::Result<()> {
    SourceFd(& self . as_raw_fd ()). register (registry, token, interests)
}
```

SourceFd()

先看`SourceFd()`的第一个入参`&self.as_raw_fd()`

```
impl AsRawFd for TcpListener {
    fn as_raw_fd (& self ) -> RawFd {
        self . inner. as_raw_fd ()
    }
}
```

这里的`inner`是`net::TcpListener`类型。跟踪到`std::net::TcpListner`的实现：

```
#[stable(feature = "rust1", since = "1.0.0")]
impl AsRawFd for net :: TcpListener {
    fn as_raw_fd (& self ) -> RawFd { * self . as_inner () . socket () . as_inner () }
}

impl AsInner <net_ imp :: TcpListener> for TcpListener {
    fn as_inner (& self ) -> &net_ imp :: TcpListener { & self . 0 }
}
```

注意到`use crate::sys_common::net as net_ imp;`

这里稍微梳理一下`TcpListener`的依赖关系。`mio`提供的`TcpListener --> sys::TcpListener --> net::TcpListener --> sys_common::net::TcpListener`

`sys_common::net::TcpListener`的`socket()`方法

```
pub fn socket (& self ) -> &Socket { & self . inner }
```

std 库的 /src/libstd/sys/unix/net.rs 找到 socket 的定义：

```
pub struct Socket (FileDesc);

impl AsInner<c_int> for Socket {
    fn as_inner(&self) -> &c_int { self.0.as_inner() }
}

pub struct FileDesc {
    fd: c_int,
}

impl AsInner<c_int> for FileDesc {
    fn as_inner(&self) -> &c_int { &self.fd }
}
```

回到 SourceFd(&self.as_raw_fd()).register(registry, token, interests)

```
/// Adapter for [`RawFd`] providing an [`event::Source`] implementation.
///
/// `SourceFd` enables registering any type with an FD with [`Poll`].
///
/// While only implementations for TCP and UDP are provided, Mio supports
/// registering any FD that can be registered with the underlying OS selector.
/// `SourceFd` provides the necessary bridge.
/// ...
```

```
pub struct SourceFd <'a> (pub &'a RawFd);

impl <'a> event::Source for SourceFd <'a> {
    fn register(&self, registry: &Registry, token: Token, interests: Interests) -> io::Result<()> {
        poll::selector(registry).register(*self.0, token, interests)
    }

    fn reregister(
        &self,
        registry: &Registry,
        token: Token,
        interests: Interests,
    ) -> io::Result<()> {
        poll::selector(registry).reregister(*self.0, token, interests)
    }

    fn deregister(&self, registry: &Registry) -> io::Result<()> {
        poll::selector(registry).deregister(*self.0)
    }
}
```

SourceFd 也实现了 event::Source trait。注释中说明很详细，SourceFd 主要是做一个桥接，使得可以注册一个文件描述符到系统的 selector。

为什么要这么绕？因为 sys::selector 需要一个文件描述符参数。

poll::selector()方法的定义：

```
// ===== Accessors for internal usage =====

pub fn selector (registry: &Registry) -> &sys::Selector {
    &registry.selector
}
```

通过前面的代码我们知道`®istry.selector`是`sys::Selector`。跟进去看看

```
pub fn register (& self , fd: RawFd, token: Token, interests: Interests) -> io::Result
<()> {
    let mut event = libc::epoll_event {
        events: interests_to_epoll (interests),
        u64 : usize :: from (token) as u64 ,
    };

    syscall!( epoll_ctl ( self .ep, libc::EPOLL_CTL_ADD, fd, & mut event)). map (|_| ())
}
```

最终是调用 linux 系统的

```
int epoll_ctl (int epfd, int op, int fd, struct epoll_event *event) ;
```

其中的参数

`self.ep`是`sys::Selector::new()`方法里调用`int epoll_create1(int flags);`返回的`int`值，是表示`epoll`实例的文件描述符。

`op`参数传入的是`libc::EPOLL_CTL_ADD`表示本次操作是add一个`fd`到`interest list`

`fd`就是刚才分析的`SourceFd`持有的`RawFd`

`epoll_event *`参数传入了`&mut event`是`libc::epoll_event`结构体。

```
pub struct epoll_event {
    pub events: uint32_t,
    pub u64 : uint64_t,
}
```

Linux 结构体相关定义：

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t ;

struct epoll_event {
    uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
```

```
};
```

uint32_t 的 bit 位表示事件类型，和 epoll_data_t 存放用户的自定义参数，这里是 Token。

看一下向系统注册了什么 interests

```
fn interests_to_epoll (interests: Interests) -> u32 {
    let mut kind = EPOLLET;

    if interests.is_readable () {
        kind = kind | EPOLLIN | EPOLLRDHUP;
    }

    if interests.is_writable () {
        kind |= EPOLLOUT;
    }

    kind as u32
}
```

EPOLLET 表示设置 Edge Triggered 边缘触发模式。

epoll 默认采用 Level Triggered 水平触发模式。关于两种模式更详细的介绍，可以参考 <http://man7.org/linux/man-pages/man7/epoll.7.html>

最开始的 example 里， interests 的实际参数是 Interests::READABLE 所以会走到：

```
if interests.is_readable () {
    kind = kind | EPOLLIN | EPOLLRDHUP;
}
```

这里还添加了 EPOLLIN | EPOLLRDHUP 事件。

EPOLLIN 就是文件处于可读性状态， EPOLLRDHUP 表示 socket 节点关闭了连接，或者关闭了写连接（TCP 是双工模式，任何一方都可以同时读写，所以任何一方也可以只关闭读或者写的连接状态）。

从这里我们知道，除了文件可读，文件关闭也会触发就绪事件。

跟踪到这里，我们总算看到了将 TcpListener 关联的 socket 注册到了 epoll 的 interest list。

poll.poll()

总算进入了核心方法

```
poll.poll (& mut events, None )?;

    /// Wait for readiness events
    ///
    /// Blocks the current thread and waits for readiness events for any of the
    /// [`event::Source`]s that have been registered with this `Poll` instance.
    /// The function will block until either at least one readiness event has
    /// been received or `timeout` has elapsed. A `timeout` of `None` means that
    /// `poll` will block until a readiness event has been received.
    ///
    pub fn poll (& mut self, events: & mut Events, timeout: Option
```

```

<Duration>) -> io:: Result <()> {
    self .registry.selector. select (events. sys (), timeout)
}

```

注释里说明，如果没有就绪事件的话，该方法阻塞当前线程。timeout的实参是None，表示该方法会一直阻塞直到有事件到达。所以外面用一个loop{}循环包裹该方法。

跟进去真正干活的的select()方法

```

pub fn select (& self , events: & mut Events, timeout: Option <Duration>) -> io::
Result <()> {
    let timeout = timeout
        . map (|to| cmp:: min (to. as_millis (), libc::c_int:: max_value () as u128 ) as
libc::c_int)
        . unwrap_or (- 1 );

    events. clear ();
    syscall! ( epoll_wait (
        self .ep,
        events. as_mut_ptr (),
        events. capacity () as i32 ,
        timeout,
    ))
    . map (|n_events| {
        // This is safe because `epoll_wait` ensures that `n_events` are
        // assigned.
        unsafe { events. set_len (n_events as usize ) };
    })
}

```

events.clear()然后调用系统的epoll_wait。如果有就绪事件，返回事件个数，并且将事件设置在events里。系统的epoll_wait

```

int epoll_wait (int epfd, struct epoll_event *events,
                int maxevents, int timeout) ;

```

event consume

回到 example 的代码，如果有客户端连接到server，代码中的 events 被赋值，

```

for event in events. iter () {
    match event. token () {
        SERVER => {
            ...
        },
        token =>{
            ...
        }
    }
}

```

通过匹配event.token()进入SERVER 的代码块（一开始注册的 token 就是SERVER）。看里面的逻辑：

```

        let (connection, address) = server.accept()?;
        println!( "Accepted connection from: {}", address);

        let token = next(& mut unique_token);
        poll.registry().register(
            &connection,
            token,
            Interests::READABLE.add(Interests::WRITABLE),
        )?;

        connections.insert(token, connection);
    }
}

```

1. 通过server.accept()获取connection实例和客户端address
2. 将这个新的connection注册到poll
3. 将connection插入HashMap方便后面使用

一个一个看。先看TcpListener的accept():

```

pub fn accept(& self) -> io:: Result<(TcpStream, SocketAddr)> {
    self.inner.accept().and_then(|(inner, addr)| {
        inner
            .set_nonblocking(true)
            .map(|()| (TcpStream::new(inner), addr))
    })
}

```

跟进去看它的inner(net::TcpListener) 的accept() 方法:

```

#[stable(feature = "rust1", since = "1.0.0")]
pub fn accept(& self) -> io:: Result<(TcpStream, SocketAddr)> {
    // On WASM, `TcpStream` is uninhabited (as it's unsupported) and so
    // the `a` variable here is technically unused.
    #[cfg_attr(target_arch = "wasm32", allow(unused_variables))]
    self.0.accept().map(|(a, b)| (TcpStream(a), b))
}

```

又会像之前一个进入sys_common::net::TcpListener 的方法，不再重复跟踪。

创建一个新的token，然后注册到Poll。注意register() 方法的参数，event::Source是&connection: TcpStream，

Interests是Interests::READABLE.add(Interests::WRITABLE)可读就绪与可写就绪。看一看TcpStream对event::Source的实现：

```

impl event ::Source for TcpStream {
    fn register(& self, registry: & Registry, token: Token, interests: Interests) -> io:: Result<()> {
        #[cfg(debug_assertions)]
        self.selector_id.associate_selector(registry)?;
        self.sys.register(registry, token, interests)
    }

    fn reregister(& self,

```

```

        registry: &Registry,
        token: Token,
        interests: Interests,
    ) -> io:: Result <()> {
    self .sys. reregister (registry, token, interests)
}

fn deregister (& self , registry: &Registry) -> io:: Result <()> {
    self .sys. deregister (registry)
}
}

```

在register()方法里，设置了self.selector_id。通过self.sys.register()注册到selector。

```

pub struct TcpStream {
    sys: sys::TcpStream,
    #[cfg(debug_assertions)]
    selector_id: SelectorId,
}

```

跟进去sys::TcpStream::register()

```

impl event ::Source for TcpStream {
    fn register (& self , registry: &Registry, token: Token, interests: Interests) -> io:: Result <()> {
        SourceFd (& self . as_raw_fd ()). register (registry, token, interests)
    }

    fn reregister (
        & self ,
        registry: &Registry,
        token: Token,
        interests: Interests,
    ) -> io:: Result <()> {
        SourceFd (& self . as_raw_fd ()). reregister (registry, token, interests)
    }

    fn deregister (& self , registry: &Registry) -> io:: Result <()> {
        SourceFd (& self . as_raw_fd ()). deregister (registry)
    }
}

```

进入了SourceFd()，上面已经分析过，不再重复。

将connection插入HashMap方便后面使用，不需要解释。

通过将TcpStream注册到Poll，一旦连接建立稳定，将触发读就绪和写就绪事件。通过event.token()分流到下面的代码：

```

token => {
    // (maybe) received an event for a TCP connection.
    let done = if let Some (connection) = connections. get_mut (&token) {
        handle_connection_event (& mut poll, connection, event)?
    } else {
        // Sporadic events happen.
    }
}

```

```

        false
    };
    if done {
        connections.remove(&token);
    }
}

```

if let Some(connection) = connections.get_mut(&token) 通过&token从HashMap获取保存的 connection然后进入handle_connection_event(), 该方法判断event.is_writable() 和event.is_readable()然后分别处理。对一个TcpStream哪一个事件会先到达? 不确定。代码中的两个if没有先后的依赖关系。

```

/// Returns `true` if the connection is done.
fn handle_connection_event (
    poll: & mut Poll,
    connection: & mut TcpStream,
    event: &Event,
) -> io:: Result < bool > {
    if event.is_writable () {
        ...
    }

    if event.is_readable () {
        ...
    }

    Ok ( false )
}

```

我们从if event.is_writable() 看

```

if event.is_writable () {
    // We can (maybe) write to the connection.
    match connection.write (DATA) {
        // We want to write the entire `DATA` buffer in a single go. If we
        // write less we'll return a short write error (same as
        // `io::Write::write_all` does).
        Ok (n) if n < DATA.len () => return Err (io::ErrorKind::WriteZero. into ()),
        Ok (_) => {
            // After we've written something we'll reregister the connection
            // to only respond to readable events.
            poll.registry ()
                .reregister (&connection, event.token (), Interests::READABLE)?
        }
        // Would block "errors" are the OS's way of saying that the
        // connection is not actually ready to perform this I/O operation.
        Err (ref err) if would_block (err) => {}
        // Got interrupted (how rude!), we'll try again.
        Err (ref err) if interrupted (err) => {
            return handle_connection_event (poll, connection, event)
        }
        // Other errors we'll consider fatal.
        Err (err) => return Err (err),
    }
}

```

如果可写就绪，就立即connection.write(DATA)。客户端会收到"Hello world!\n"字符，该方法的签名

```
fn write (& mut self , buf: &[ u8 ]) -> io:: Result < usize >;
```

如果写入字节长度小于DATA.len()就返回 Err。

如果写入成功，重新注册该connection，只关心Interests::READABLE可读就绪事件。

```
// After we've written something we'll reregister the connection
// to only respond to readable events.
poll. registry ()
    . reregister (&connection, event. token (), Interests::READABLE)?
```

如果返回的错误是阻塞则什么也不做。

如果返回的是被打断，则递归调用handle_connection_event()

接着看if event.is_readable()

```
if event. is_readable () {
    let mut connection_closed = false ;
    let mut received_data = Vec :: with_capacity ( 4096 );
    // We can (maybe) read from the connection.
    loop {
        let mut buf = [ 0 ; 256 ];
        match connection. read (& mut buf) {
            Ok ( 0 ) => {
                // Reading 0 bytes means the other side has closed the
                // connection or is done writing, then so are we.
                connection_closed = true ;
                break ;
            }
            Ok ( n ) => received_data. extend_from_slice (&buf[ .. n ]),
                // Would block "errors" are the OS's way of saying that the
                // connection is not actually ready to perform this I/O operation.
                Err ( ref err ) if would_block (err) => break ,
                Err ( ref err ) if interrupted (err) => continue ,
                // Other errors we'll consider fatal.
                Err (err) => return Err (err),
            }
        }

        if let Ok (str_buf) = from_utf8 (&received_data) {
            println! ( "Received data: {}" , str_buf. trim_end ());
        } else {
            println! ( "Received (none UTF-8) data: {}" , &received_data);
        }

        if connection_closed {
            println! ( "Connection closed" );
            return Ok ( true );
        }
    }
}
```

有一个loop {}, 其中 connection.read(&mut buf) 方法的签名:

```
fn read (& mut self , buf: & mut [ u8 ]) -> io:: Result < usize > {  
    (& self . sys). read (buf)  
}
```

如果返回结果为 0, 表示全部读取完毕。connection_closed = true; 对方已经关闭连接。

如果返回 n, 说明读取到客户端的数据, 将数据追加到received_data。

如果是阻塞, 则跳出loop循环。

如果是被打断, 则继续循环。

将读取到的数据打印出来, 如果connection_closed = true; 则返回Ok(true), 否则后面会默认返回 Ok(false)。

大部分情况下, 可读就绪, 但是客户端没有任何数据输入会走阻塞, 跳出loop那个分支。

回到外层的方法:

```
let done = ... {  
    handle_connection_event (& mut poll, connection, event)?  
} else {  
    // Sporadic events happen.  
    false  
};  
if done {  
    connections. remove (&token);  
}
```

如果刚才返回了Ok(true), 会connections.remove(&token);

Waker

上面的分析没涉及到的, 有一个/src/waker.rs需要提一下。Waker允许跨线程唤醒Poll。

```
#[derive(Debug)]  
pub struct Waker {  
    inner: sys::Waker,  
}  
  
impl Waker {  
    /// Create a new `Waker`.  
    pub fn new (registry: &Registry, token: Token) -> io:: Result <Waker> {  
        sys::Waker:: new (poll:: selector (&registry), token). map (|inner| Waker { inner })  
    }  
  
    /// Wake up the [`Poll`] associated with this `Waker`.  
    ///  
    /// [`Poll`]: crate::Poll  
    pub fn wake (& self ) -> io:: Result < () > {  
        self .inner. wake ()  
    }  
}
```

```
}
```

定义比较简单，也就两个方法。我们看看 Linux 下的实现：

```
##[cfg(any(target_os = "linux", target_os = "android"))]
mod eventfd {
    use crate::sys::Selector;
    use crate::{Interests, Token};

    use std::fs::File;
    use std::io::{self, Read, Write};
    use std::os::unix::io::FromRawFd;

    /// Waker backed by `eventfd`.
    ///
    /// `eventfd` is effectively an 64 bit counter. All writes must be of 8
    /// bytes (64 bits) and are converted (native endian) into an 64 bit
    /// unsigned integer and added to the count. Reads must also be 8 bytes and
    /// reset the count to 0, returning the count.
    #[derive(Debug)]
    pub struct Waker {
        fd: File,
    }

    impl Waker {
        pub fn new(selector: &Selector, token: Token) -> io::Result<Waker> {
            syscall!(eventfd(0, libc::EFD_CLOEXEC | libc::EFD_NONBLOCK)).and_then(|fd| {
                // Turn the file descriptor into a file first so we're ensured
                // it's closed when dropped, e.g. when register below fails.
                let file = unsafe { File::from_raw_fd(fd) };
                selector
                    .register(fd, token, Interests::READABLE)
                    .map(|()| Waker { fd: file })
            })
        }
    }

    pub fn wake(&self) -> io::Result<()> {
        let buf: [u8; 8] = 1u64.to_ne_bytes();
        match (&self.fd).write(&buf) {
            Ok(_) => Ok(()),
            Err(ref err) if err.kind() == io::ErrorKind::WouldBlock => {
                // Writing only blocks if the counter is going to overflow.
                // So we'll reset the counter to 0 and wake it again.
                self.reset()?;
                self.wake()
            }
            Err(err) => Err(err),
        }
    }

    /// Reset the eventfd object, only need to call this if `wake` fails.
    fn reset(&self) -> io::Result<()> {
        let mut buf: [u8; 8] = 0u64.to_ne_bytes();
        match (&self.fd).read(&mut buf) {
            Ok(_) => Ok(()),
            // If the `Waker` hasn't been awoken yet this will return a
            // `WouldBlock` error which we can safely ignore.
        }
    }
}
```

```

        Err ( ref err ) if err . kind () == io :: ErrorKind :: WouldBlock => Ok ( () ),
        Err ( err ) => Err ( err ),
    }
}
}

#[cfg(any(target_os = "linux", target_os = "android"))]
pub use self :: eventfd :: Waker;

```

可以看到，Waker的new()是调用了系统的eventfd()方法。

```
int eventfd (unsigned int initval, int flags) ;
```

eventfd()方法创建一个文件描述符用于事件(event)通知。既可以用于用户空间的应用间通知，还可以用于内核空间事件通知用户空间的应用。实际上是内核空间维护的一个64位的int counter，表示“eventfd object”。

initval参数是初始化的值，flags包括：

EFD_CLOEXEC表示fork子进程时不继承。

EFD_NONBLOCK通常会设置成O_NONBLOCK，如果不设置，read可能会阻塞。

EFD_SEMAPHORE支持semaphore语义的read。

这个“eventfd object”的相关操作很简单，write()会修改counter的值(累加)，read()读取counter的值，然后清零(如果是semaphore模式，就减1)。

上面的源码中，通过eventfd()创建了fd，然后将该fd注册到了sys::selector(实际上就是epoll)，Interests是可读就绪。

```

selector
    . register ( fd, token, Interests :: READABLE )
    . map ( | () | Waker { fd: file } )

```

接着看它的wake()

```

pub fn wake (& self ) -> io :: Result < () > {
    let buf : [ u8 ; 8 ] = 1u64 . to_ne_bytes ();
    match (& self . fd) . write (& buf) {
        Ok ( _) => Ok ( () ),
        Err ( ref err ) if err . kind () == io :: ErrorKind :: WouldBlock => {
            // Writing only blocks if the counter is going to overflow.
            // So we'll reset the counter to 0 and wake it again.
            self . reset ();
            self . wake ()
        }
        Err ( err ) => Err ( err ),
    }
}

```

就是很简单地写入一个64位的1。会被加到当前的counter上。

`write(&buf)`的结果正常都会 OK。但是当超过了counter的最大值`0xfffffffffffffe`会导致int_64溢出，本次写操作会阻塞直到有读操作，如果文件设置了非阻塞（上面源码设置的`libc::EFD_NONBLOCK`），会返回EAGAIN的错误。

这里只处理了`io::ErrorKind::WouldBlock`的错误，是不是存在问题？

并不是。按照 POSIX 的标准，Linux 里面对错误的定义是：

All the error names specified by POSIX.1 must have distinct values, with the exception of EAGAIN and EWOULDBLOCK, which may be the same. On Linux, these two have the same value on all architectures.

是同一个错误值，而 Rust 目前只定义了`io::ErrorKind::WouldBlock`来统一表示`EWOULDBLOCK` 和`EAGAIN`，在有些场景下会引发混乱。

在freeBSD、macOS系统上，通过`kqueue()`, `kevent()`实现Waker。

openbsd, solaris系统上通过`pipe`实现Waker。

windows 环境下通过 IOCP (I/O Completion Port) 实现。

最后

本文对 mio 的核心流程做了一个概要性分析，由于只关注核心流程，很多细节并没有展开。

mio 的代码整体上小巧精悍，大都是对底层操作系统的很薄的一层包装。这也符合自身的定位。

with a focus on adding as little overhead as possible

核心逻辑围绕`sys::selector`、`event::source`展开。`Poll`是一个Facade Pattern。

本次分析没有涉及到 udp，可自行分析，代码在 `/src/net/udp.rs`

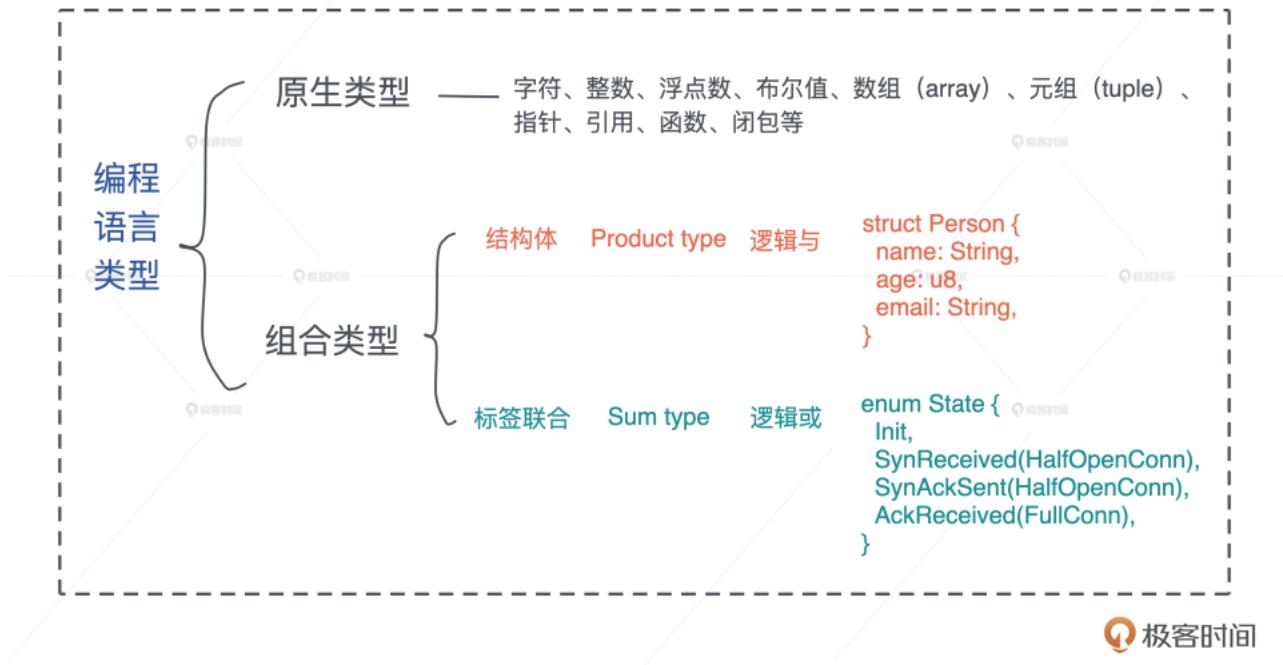
window 平台相关代码在 `/src/sys/windows`。

分析完 mio，也就为分析 tokio 代码打好了基础。

<https://blog.zongwu233.com/rust-mio-source-scenario-analysis/>

附录

- rust基本类型里面的字符串和指针



极客时间

新分组

文档中心

Rust 官方文档翻译项目组提供了全面的学习 Rust 语言的官方文档和其他教程的中文知识库，从入门到深入学习 Rust 编程语言的各类资源，由 Rust 翻译项目组和其他参与 Rust 开源项目的爱好者共同提供。

<https://course.rs/about-book.html>

Rust 程序设计语言

通过例子学 Rust

Rust 参考手册官方的 Rust 语言规范，涵盖 Rust 的最全面的规范内容，目前文档还未稳定，属于预览版。

Rust 标准库官方的 Rust 标准库，是可移植 Rust 软件的基础，标准库文档是开发过程中的必备内容。

Rust CookbookRust 官方图书，由一系列简单程序示例构成，展示了 Rust 实际开发场景的良好实践。

Cargo 手册Cargo 是 Rust 的包管理器，通过本手册全面了解如何构建 Rust 程序和大型项目。

Rust 版本指南了解 Rust 版本有关内容，比如初版（Rust 2015）、Rust 2018 版和 2021 版，掌握如何迁移代码。

rustdoc 手册学习 rustdoc 命令和使用 cargo doc 来生成漂亮的 Rust crate 的帮助文档，更好地管理和上传项目 API 文档。

rustc 手册深入学习 rustc 命令行，熟悉 Rust 编译器的可用选项，构建强大的 Rust 程序。

Rust 编译错误索引表和 Rust 打交道避免不了编译错误，通过编译错误索引表可以快速找到错误的详细说明和介绍，深入理解 Rust 的报错机理。

Rust 规范文档Rust 中文资源和配套措施，相关翻译指引的范，以及中文翻译项目组的社区运作说明等。

Rust 语言术语中英文对照表Rust 中文翻译项目组提供，致力于实现 Rust 的文档和书籍中文的术语都保持一致性。

脑图

learning

learning