Rust 秘典

⚠️ 警告。 这本书是不完整的。 记录所有内容和重写过时的部分需要一段时间。 请参见 issue tracker以检查哪些内容缺失/过时，如果有任何错误或想法仍未被报告，欢迎随时提一个新 Issue。

译者的话：首先，限于译者自身姿势水平，翻译有可能无法做到完全信达雅，并且有一些专业术语不知道如何翻译到中文，在这里先向大家道歉，请多包涵。

不过，译者保证所有翻译的内容都是译者阅读并调整过多次的，并且译者会努力将内容调整到满足**能看懂**的要求，并且做到不遗漏原文内容。

如果大家对于翻译有更好的建议或者想法，欢迎直接 PR~

目前翻译基于 commit：e3f3af69dce71cd37a785bccb7e58449197d940c，基于时间：2023/9/12

Q：为什么不基于之前已有的中文版进行改进?

A：因为翻译成中文版后，很难再回过头去看和现在的英文版原文到底差了啥，所以还不如完全重新翻译一遍。

Q：那会不会有一天你的这个版本也过期了?

A：希望没有那一天。我 watch 了英文原版的所有 pr，如果有变更（希望）能及时更新。当然，也欢迎大家一起贡献 PR。

不安全 Rust 的黑魔法

这些知识是“按原样”提供的，没有任何形式的明示或暗示的保证，包括但不限于释放难以描述的恐怖，粉碎你的心灵，让你的思想漂流在不可知的无限宇宙中。

Rust 秘典挖掘了你在编写不安全 Rust 程序时需要了解的所有可怕的细节。

如果你希望在编写 Rust 程序的过程中获得长久而快乐的职业生涯，你应该现在回头，忘记你曾经看过这本书。 它没有必要。 然而，如果你打算编写不安全代码——或者只是想深入了解语言的内涵——这本书包含了很多有用的信息。

与 *The Rust Programming Language* 不同的是，我们将假设你有相当多的前期知识。 特别是，你应该对基本的系统编程和 Rust 非常熟悉。 如果你对这些主题感到困惑，你应该考虑先阅读The

Book。 也就是说，我们不会假定你已经读过了，而且我们会注意偶尔在适当的时候对基础知识进行复习。 如果你想的话，你可以直接跳过The Book来看这本书：但你需要知道我们不会从头到尾地详细解释一切。

本书主要是作为The Reference的高级配套读物而存在。 《The Reference》的存在是为了详细说明语言的每一部分的语法和语义，而《Rust 秘典》的存在是为了描述如何将这些部分结合起来使用，以及你在这样做时将会遇到的问题。

《The Reference》会告诉你引用、析构器和 unwind 的语法和语义，但它不会告诉你如何将它们结合起来导致异常安全问题，或如何处理这些问题。

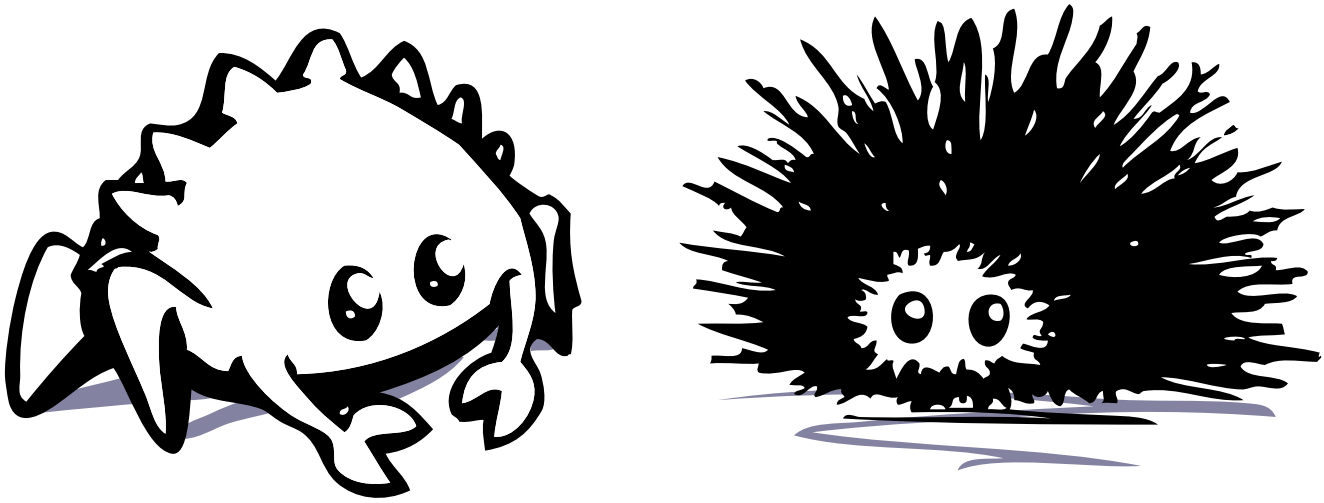
需要注意的是，我们没有很好地同步 The Rustnomicon 和 The Reference，所以它们可能有重复的内容。 一般来说，如果这两个文档有分歧，应该认为《The Reference》是正确的（它还没有被认为是规范性的，只是维护得更好）。

本书范围内的主题包括：（不）安全的含义、语言和标准库提供的不安全基础、用这些不安全基础创建安全抽象的技术、子类型和可变性（variance）、异常安全（恐慌/unwind 安全性）、与未初始化的内存相关的工作、类型转换、并发、与其他语言的互操作（FFI）、优化技巧、如何构建低级到编译器/操作系统/硬件的基元（primitives）、如何**不使**内存模型程序员生气、如何**使**内存模型程序员生气、以及更多。

Rust 秘典不是一个详尽描述标准库中每一个 API 的语义和保证的地方，也不是一个详尽描述 Rust 的每一个特性的地方。

除非另有说明，本书中的 Rust 代码使用 Rust 2021 版。

认识 Safe 与 Unsafe



我们都不想关心底层的实现细节。谁会关心空元组占用了多少空间呢？可悲的是，它有时很重要，我们需要担心这个问题。开发人员开始关心实现细节的最常见的原因是性能，但更重要的是，当与硬件、操作系统或其他语言直接交互时，这些细节就是关乎对错的问题。

当实现细节在安全的编程语言中开始变得重要时，程序员通常有三种选择。

- 调整代码以鼓励编译器/运行时进行优化
- 采用一种更不规范或更繁琐的设计来获得所需的实现
- 用一种能让你处理这些细节的语言重写实现

对于最后一种选择，程序员往往使用的语言是*C*。这对于对接那些只声明 C 语言接口的系统来说往往是必要的。

不幸的是，C 语言使用起来非常不安全（尽管有时有很好的理由），当试图与另一种语言交互时，这种危险会被放大。我们必须小心翼翼地确保 C 语言和其他语言的一致性，以使它们不会越俎代庖。

那么，这与 Rust 有什么关系呢?

嗯，与 C 不同，Rust 是一种安全的编程语言。

但是，和 C 语言一样，Rust 也是一种不安全的编程语言。

更准确地说，Rust 同时*包含*了一种安全和一种不安全的编程语言。

Rust 可以被认为是两种编程语言的结合。*Safe Rust* 和*Unsafe Rust*。顾名思义，Safe Rust 是安全的。Unsafe Rust 是，嗯，不安全的。事实上，Unsafe Rust 让我们做一些*真正*不安全的事情。Rust 的作者会恳求你不要做这些事情，但我们还是要做。

Safe Rust 是*真正*的 Rust 编程语言。如果你只写 Safe Rust，你将永远不必担心类型安全或内存安全的问题。你永远不会遇见悬空的指针，释放后使用（use-after-free），或任何其他类型的未定

义行为。

标准库也为你提供了足够多的开箱即用的工具，你将能够用纯粹的Safe Rust 编写高性能的应用程序和库。

但是，也许你想调用另一种语言，也许你正在写一个标准库没有暴露的低级抽象，也许你正在*写*标准库（它完全是用 Rust 写的），也许你需要做点类型系统看不懂的*底层数据操作*。也许你需要 Unsafe Rust。

Unsafe Rust 与Safe Rust 完全一样，具有所有相同的规则和语义。它只是允许你做一些*额外的*、绝对不安全的事情（我们将在下一节中定义）。

这种分离的价值在于，我们获得了使用像 C 这样的不安全语言的好处——获得对底层实现细节的控制——而与其他安全语言交互时却没有那么多问题了。

仍然有一些问题——最明显的是，我们必须意识到类型系统有一些程序必须遵守的假设的规则，且认真审核任何与 Unsafe Rust 交互的代码以遵守规则。这就是本书的目的：教你了解这些规则以及如何遵守它们。

Safe 和 Unsafe 如何交互

Safe Rust 和 Unsafe Rust 之间有什么关系？它们又是如何交互的?

Safe Rust 和 Unsafe Rust 之间的边界由`unsafe`关键字控制，`unsafe`是承接了它们之间交互的桥梁。这就是为什么我们可以说 Safe Rust 是一种安全的语言：所有不安全的部分都被限制在“unsafe”边界之内。如果你愿意，你甚至可以把`#![forbid(unsafe_code)]`扔进你的代码库，以静态地保证你只写安全的 Rust。

`unsafe`关键字有两个用途：声明编译器不会保证这些代码的安全性，以及声明程序员已经确保这些代码是安全的。

你可以用`unsafe`来表示在 *函数* 和 *trait* *声明* 这些行为不一定安全。对于函数，`unsafe`意味着函数的用户必须仔细阅读该函数的文档，以确保他们在使用该函数时能满足函数能安全运行的条件。对于 trait 声明，`unsafe`意味着 trait 的实现者必须仔细阅读 trait 文档，以确保他们的实现遵循了 trait 所要求条件。

你可以在一个块上使用`unsafe`来声明在其中执行的所有不安全操作都经过了验证以保证操作的安全性。例如，当传递给`slice::get_unchecked`的索引在边界内，这一行为就是安全的。

你可以在 trait 的实现上使用`unsafe`来声明该实现满足了 trait 的条件。例如，实现`Send`说明这个类型移动到另一个线程是真正安全的。

标准库中有许多 unsafe 的函数，包括：

- `slice::get_unchecked`，它不会检查传入索引的有效性，允许违反内存安全的规则。
- `mem::transmute`将一些数据重新解释为给定的类型，绕过类型安全的规则（详见 conversions）。
- 每一个指向一个 Sized 类型的原始指针都有一个`offset`方法，如果传递的偏移量不在“界内”，则该调用是未定义行为。
- 所有 FFI（外部函数接口 Foreign Function Interface）函数的调用都是`unsafe`的，因为 Rust 编译器无法检查其他语言的操作。

从 Rust 1.29.2 开始，标准库定义了以下 unsafe trait（还有其他 trait，但还没有稳定下来，有些可能永远不会稳定下来）：

- `Send`是一个标记 trait（一个没有 API 的 trait），承诺实现了`Send`的类型可以安全地发送（移动）到另一个线程。
- `Sync`是一个标记 trait，承诺线程可以通过共享引用安全地共享实现了`Sync`的类型。
- `GlobalAlloc`允许自定义整个程序的内存分配器。

Rust 标准库的大部分内容也在内部使用了 Unsafe Rust。这些实现一般都经过严格的人工检查，所以建立在这些实现之上的 Safe Rust 接口可以被认为是安全的。

我们需要将它们分离，是因为 Safe Rust 的一个基本属性，即*健全性属性*。

无论怎样，Safe Rust 都不会导致未定义行为。

Safe 与 Unsafe 分离的设计意味着 Safe Rust 和 Unsafe Rust 之间存在着不对等的信任关系。一方面， Safe Rust 本质上必须相信它所接触的任何 Unsafe Rust 都是正确编写的。另一方面，Unsafe Rust 在信任 Safe Rust 时必须非常小心。

例如，Rust 有 `PartialOrd` 和 `Ord` trait 来区分“部分序”比较的类型和“全序”比较的类型（这意味着比较行为必须是合理的）。

`BTreeMap` 对于 `PartialOrd` 的类型来说并没有实际意义，因此它要求其 key 实现 `Ord`。然而，`BTreeMap` 在其实现中包含了 Unsafe 的代码，所以无法接受马虎的（可以用Safe编写的） `Ord` 实现，因为这会导致未定义行为。因此，BTreeMap 中的 Unsafe 代码必须被编写成对实际上不完全的 `Ord` 实现具有鲁棒性——尽管我们要求 `Ord` 是正确实现的。

Unsafe Rust 代码不能相信 Safe Rust 代码会被正确编写。也就是说，如果你输入了没有正确实现全序排序的值， `BTreeMap` 仍然会表现得完全不正常。它只是不会导致未定义行为。

有人可能会问，如果 `BTreeMap` 不能信任 `Ord`，因为它是安全的，那么它为什么能信任*任何*安全的代码呢？例如， `BTreeMap` 依赖于整数和 slice 的正确实现。这些也是安全的，对吗？

区别在于范围的不同。当 `BTreeMap` 依赖于整数和分片时，它依赖于一个非常具体的实现。这是一个可以衡量的风险，可以与收益相权衡。在这种情况下，风险基本上为零；如果整数和 slice 被破坏，那么*所有*人都会被破坏。而且，它们是由维护 `BTreeMap` 的人维护的，所以很容易对它们进行监控。

另一方面， `BTreeMap` 的 key 类型是泛型的。信任它的 `Ord` 实现意味着信任过去、现在和未来的每一个 `Ord` 实现。这里的风险很高：有人会犯错误，把他们的 `Ord` 实现搞得一团糟，甚至直接撒谎说提供了一个完整的排序，因为“它看起来很有效”。 `BTreeMap` 需要做好准备应对这种情况的发生。

同样的逻辑也适用于信任一个传递给你的闭包会有正确的行为。

`unsafe` trait 就是用来解决泛型的信任问题。理论上， `BTreeMap` 类型可以要求 key 实现一个新的 trait，称为 `UnsafeOrd`，而不是 `Ord`，它可能看起来像这样：

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

unsafe trait UnsafeOrd {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

然后，一个类型将使用 `unsafe` 来实现 `UnsafeOrd`，表明他们已经确保他们的实现满足了该 trait 所期望的任何条件。在这种情况下， `BTreeMap` 内部的 Unsafe Rust 有理由相信 key 类型的 `UnsafeOrd` 实现是正确的。如果不是这样，那就是 unsafe trait 实现的错，这与 Rust 的安全保证是一致的。

是否将一个 trait 标记为 `unsafe` 是一个 API 设计。一个 safe trait 更容易实现，但任何依赖它的 Unsafe 代码都必须抵御不正确的行为。将 trait 标记为 `unsafe` 会将这个责任转移到实现者身上。

Rust 习惯于避免将 trait 标记为`unsafe`，因为它使 Unsafe Rust 普遍存在，这并不可取。

`Send`和`Sync`被标记为 unsafe，是因为线程安全是一个*基本*的属性，unsafe 代码不可能像抵御一个有缺陷的`Ord`实现那样去抵御它。同样地，`GlobalAllocator`是对程序中所有的内存进行记录，其他的东西如`Box`或`Vec`都建立在它的基础上。如果它做了一些奇怪的事情（当它还在使用的时候，把同一块内存给了另一个请求），就没有机会检测到并采取任何措施了。

决定是否将你自己的 trait 标记为“unsafe”，也是出于同样的考虑。如果“unsafe”的代码不能抵御 trait 的错误实现，那么将 trait 标记为“unsafe”就是一个合理的选择。

顺便说一下，虽然`Send`和`Sync`是`unsafe`的特性，但它们*也是*自动实现的类型，当这种派生可以证明是安全的。`Send`是自动派生的，只适用于一个类型下所有类型都实现了`Send`。`Sync`是自动派生的，只适用于一个类型下所有类型都实现了`Sync`。这最大限度地减少了使这两个 trait “unsafe” 的危险。而且，没有多少人会去*实现*内存分配器（或者针对这个问题而言，直接使用它们）。

这就是 Safe Rust 和 Unsafe Rust 之间的平衡。这种分离是为了使 Safe Rust 的使用尽可能符合人体工程学，但在编写 Unsafe Rust 时需要额外的努力和小心。本书的其余部分主要是讨论必须采取的谨慎，以及Unsafe Rust 必须坚持的契约。

Unsafe Rust 能做什么

在 Unsafe Rust 中唯一不同的是，你可以：

- 对原始指针进行解引用
- 调用 “Unsafe” 的函数（包括 C 函数、编译器的内建指令和原始分配器。
- 实现 “Unsafe” trait
- 改变静态数据
- 访问 “union” 的字段

这就是全部了。这些操作被归入 unsafe 的原因是，误用其中的任何一项都会引起可怕的未定义行为。调用“未定义行为”使编译器有充分的权利对你的程序做任何坏事。你绝对_不能_调用“未定义行为”。

与 C 语言不同，Rust 中的“未定义行为”的范围相当有限。核心语言中，你只需要关心防止以下事情：

- 解除引用（使用 `*` 运算符）悬空或不对齐的指针（见下文）
- 破坏指名针别规则
- 调用一个 ABI 错误的函数，或者从一个 unwind ABI 错误的函数中 unwinding
- 引起数据竞争
- 执行用当前执行线程不支持的目标特性编译的代码
- 产生无效的值（无论是单独的还是作为一个复合类型的字段，如 `enum` / `struct` / `array` / `tuple` ）
  - 一个不是 0 或 1 的 `bool`
  - 一个具有无效判别符的 `enum`
  - 一个空的 `fn` 指针
  - 一个超出[0x0, 0xD7FF]和[0xE000, 0x10FFFF]范围的 `char`
  - 一个 `!` （所有的值对这个类型都是无效的）
  - 一个从未初始化的内存读出的整数(`i*` / `u*`)、浮点值(`f*`)或原始指针，或 `str` 中的未初始化的内存
  - 一个悬空的、不对齐的、或指向无效值的引用/ `Box`
  - 一个胖指针、 `Box` 或原始指针，具有无效的元数据：
    - 如果一个 `dyn Trait` 指针 / 引用指向的 vtable 和对应 Trait 的 vtable 不匹配，那么 `dyn Trait` 的元数据是无效的
    - 如果 Slice 的长度不是有效的 usize（比如，从未初始化的内存中读取的 usize），那么 Slice 的元数据是无效的
  - 一个由类型自定义的无效值，比如在标准库中的 `NonNull` 和 `NonZero*` (自定义无效值是一个不稳定的特性，但一些稳定的 libstd 类型，如 `NonNull` 使用了这个特性)。

赋值、传递给一个函数/原始操作、从一个函数/原始操作返回的时候，都会“产生”一个值。

如果一个引用/指针是空的，或者它所指向的地址并非都是合法的地址（合法地址都应该是已分配内存的），那么它就是 `悬垂` 的。它所指向的范围是由指针值和被指向类型的大小决定的（使用

`size_of_val`）。因此，如果指向的范围是空的，`悬垂`与`空`是一样的。要注意，切片和字符串指向它们的整个范围，所以它们元数据中的长度不能太大。内存分配的长度、切片和字符串的长度不能大于`isize::MAX`字节。如果因为某些原因，这太麻烦了，可以考虑使用原始指针。

这就是所有 Rust 中可能会导致未定义行为的原因。当然，unsafe 的函数和 trait 可以自由地声明任意的其他约束，程序必须保持这些约束以避免未定义行为。例如，分配器 API 声明，释放未分配的内存是未定义行为。

然而，对这些约束的违反通常只会导致上述问题中的一个，一些额外的约束也可能来自于编译器，编译器为优化代码做出了特殊的假设。例如，Vec 和 Box 使用了内建指令，要求他们的指针在任何时候都是非空的。

Rust 在其他方面对其他可疑的操作是相当宽容的。Rust 认为以下情况是“安全的”：

- 死锁
- 有一个数据竞争
- 内存泄漏
- 整数溢出（使用内置的运算符，比如“+”）
- 中止程序
- 删除生产数据库

然而任何真正可能做这种事情的程序都是*可能*不正确的，Rust 提供了很多工具来尽可能检查出这些问题，但要这些问题完全被预防是不现实的。

使用 Unsafe

Rust 通常让我们以作用域的方式来限制 unsafe 代码块。不幸的是，现实要比这复杂得多。例如，考虑下面这个玩具函数：

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

这个函数是安全和正确的。我们先检查索引是否在界内，如果是，就以不检查的方式索引到数组中。我们说，这样一个正确实现的 unsafe 函数是*健全*的，这意味着安全代码不能通过它引起未定义行为（记住，这是安全 Rust 的唯一基本属性）。

但即使在这样一个微不足道的函数中，不安全的代码块也是值得怀疑的，比如将 `<` 改为 `<=`：

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx <= arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

这个程序现在是*不健全*的，Safe Rust 会导致未定义行为，尽管*我们只修改了安全代码*。这就是安全的基本问题：它是并非只是局部的问题。我们的 unsafe 操作的健壮性必然取决于由其他 “safe” 操作建立的状态。

Safe 是模块化的，你不需要考虑任何其它的 Unsafe 块带来的潜在问题。例如，对一个切片使用一个未经检查的索引并不意味着你突然需要担心这个分片是空的或者包含未初始化的内存。没有任何根本性的变化。然而，Safe 又*不是*模块化的，因为程序本身是有状态的，你的 unsafe 操作可能依赖于任意状态。

当我们加入实际的持久化状态时，这种非局部性会变得更糟糕。例如，让我们看一下 `Vec` 的一个简单实现：

```
use std::ptr;

// 注意：这个定义十分简单。参考实现 Vec 的章节
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

// 注意：这个实现未考虑大小为 `0` 的类型。参考实现 Vec 的章节
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // 这里并不重要
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.add(self.len), elem);
            self.len += 1;
        }
    }
}
```

这段代码很简单，可以很简单地确认和验证，但是现在我们添加以下方法：

```
fn make_room(&mut self) {
    // 增加容量
    self.cap += 1;
}
```

这段代码是 100% 安全的 Rust，但它也是完全不健全的。改变容量违反了 Vec 的不变性（即 `cap` 反映了 Vec 中分配的空间）。这不是 Vec 的其他部分所能防范的。它*不得不*相信容量字段，因为没有办法验证它。

因为它依赖于一个结构字段的不变性，这段 “unsafe” 的代码不仅仅污染了整个函数：它污染了整个*模块*。一般来说，限制不安全代码的范围的唯一方法是在模块边界上设置权限。

然而，其实这个改动是可以*完美地*工作的。`make_room` 的存在对于 Vec 的健全性来说*不是*个问题，因为我们没有把它标记为公共的。只有定义了这个函数的模块可以调用它。另外，`make_room` 直接访问了 Vec 的私有字段，所以它只能写在与 Vec 相同的模块中。

因此，我们有可能基于复杂的不变性，编写一个完全安全的抽象。这对 Safe Rust 和 Unsafe Rust 之间的关系是*非常重要*的。

我们已经看到， unsafe 代码必须一*部分*信任 safe 代码，但不应该完全信任 safe 代码。出于类似的原因，访问控制对不安全代码也很重要：它可以防止我们不得不信任宇宙中所有的 safe 代码，防止它们扰乱我们的信任状态。

安全万岁！

Rust 中的数据布局

低层编程非常关心数据布局，这是个大问题。它也无孔不入地影响着语言的其他部分，所以我们将从挖掘数据在 Rust 中的布局方式开始。

本章最好与《The Reference》中的类型布局部分保持一致，并使之成为仅仅是多渲染了一份。本书刚写的时候，《The Reference》已经完全失修，而 Rust 秘典试图作为《The Reference》的部分替代。现在的情况不再是这样了，所以这一整章最好可以删除。

我们会把这一章再保留一段时间，但理想的情况是，你应该把任何新的事实或改进贡献给《The Reference》。

repr(Rust)

首先，所有类型都有一个以字节为单位的对齐方式，一个类型的对齐方式指定了哪些地址可以用来存储该值。一个具有对齐方式`n`的值只能存储在`n`的倍数的地址上。所以对齐方式 2 意味着你必须存储在一个偶数地址，而 1 意味着你可以存储在任何地方。对齐至少是 1，而且总是 2 的幂。

基础类型通常按照其大小对齐，尽管这是特定平台的行为。例如，在 x86 上`u64`和`f64`通常被对齐到 4 字节（32 位）。

一个类型的大小必须始终是其对齐方式的倍数（零是任何对齐方式的有效大小），这就保证了该类型的数组总是可以通过偏移其大小的倍数来进行索引。注意，在动态大小的类型的情况下，一个类型的大小和对齐方式可能不是静态的。

Rust 给你提供了以下方式来布置复合数据。

- structs （命名复合类型 named product types）
- tuples （匿名复合类型 anonymous product types）
- arrays （同质复合类型 homogeneous product types）
- enums （命名总和类型 —— 有标签的联合体 named sum types -- tagged unions）
- unions （无标签的联合体 untagged unions）

如果一个枚举的所有变体都没有相关联的数据，那么它就被称为*无字段(field-less)*。

默认情况下，复合结构的对齐方式等于其字段对齐方式的最大值。因此，Rust 会在必要时插入填充，以确保所有字段都正确对齐，并且整个类型的大小是其对齐的倍数。比如说：

```
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

将在目标上以 32 位对齐，将这些基本类型对齐到它们各自的大小。因此，整个结构的大小将是 32 位的倍数。它可能变成：

```
struct A {
    a: u8,
    _pad1: [u8; 3], // 需要和 `b` 内存对齐
    b: u32,
    c: u16,
    _pad2: [u8; 2], // 让总体的大小是 4 的倍数
}
```

或者：

```
struct A {
    b: u32,
    c: u16,
    a: u8,
    _pad: u8,
}
```

所有数据都如同C语言中的一样，直接存储在结构里。然而，除了数组（密集排列且有序）之外，数据的布局在默认情况下都不是确定的。给出以下两个结构的定义：

```
struct A {
    a: i32,
    b: u64,
}

struct B {
    a: i32,
    b: u64,
}
```

Rust *确实*保证 A 的两个实例的数据布局完全相同。然而，Rust 目前*并不*保证 A 的实例与 B 的实例具有相同的字段排序或填充。

对于我们编写的 A 和 B 来说，这一点似乎是没有必要的，但是 Rust 的其他几个特性使得该语言有必要以复杂的方式来处理数据布局。

例如，考虑这个结构：

```
struct Foo<T, U> {
    count: u16,
    data1: T,
    data2: U,
}
```

现在考虑一下 `Foo<u32, u16>` 和 `Foo<u16, u32>` 的单态化的结果。如果 Rust 按照指定的顺序排列字段，我们希望它能对结构中的值进行填充以满足其对齐要求。因此，如果 Rust 不对字段重新排序，我们希望它能产生以下结果：

```
struct Foo<u16, u32> {
    count: u16,
    data1: u16,
    data2: u32,
}

struct Foo<u32, u16> {
    count: u16,
    _pad1: u16,
    data1: u32,
    data2: u16,
    _pad2: u16,
}
```

后一种情况很显然浪费了空间，更高效地利用空间要求不同的单体有*不同的字段排序*。

枚举使情况变得更加复杂，直观地说，一个枚举如下：

```
enum Foo {
    A(u32),
    B(u64),
    C(u8),
}
```

可能会被布局成：

```
struct FooRepr {
    data: u64, // 根据 tag 的不同，这一项可以为 u64，u32，或者 u8
    tag: u8,   // 0 = A, 1 = B,  2 = C
}
```

事实上，这正是它的布局方式（根据`tag`的大小和位置来调整）。

然而，在一些情况下，这样的表述是低效的。这方面的典型案例是 Rust 的“空指针优化”：一个由单个外部单元变量（例如`None`）和一个（可能嵌套的）非空指针变量（例如`Some(&T)`）组成的枚举，使得标签没有必要。空指针可以安全地被解释为单位（`None`）的变体。这导致的结果是，`size_of::<Option<&T>>() == size_of::<&T>()`。

在 Rust 中，有许多类型会包含不可为空的指针，如`Box<T>`、`Vec<T>`、`String`、`&T`和`&mut T`。同理，我们可以想象嵌套的枚举将它们的标记集中到一个单一的字段中，因为根据定义，它们的有效值范围有限。原则上，枚举可以使用相当复杂的算法，在整个嵌套类型中用禁止使用的值来存储枚举类型。因此，我们不指定枚举布局是*特别*值得的。

非正常大小的类型

大多数的时候，我们期望类型在编译时能够有一个静态已知的非零大小，但这并不总是 Rust 的常态。

Dynamically Sized Types (DSTs)

Rust 支持动态大小的类型（DST）：这些类型没有静态（编译时）已知的大小或者布局。从表面上看这有点离谱：Rust *必须*知道一个东西的大小和布局，才能正确地进行处理。从这个角度上看，DST 不是一个普通的类型，因为它们没有编译时静态可知的大小，它们只能存在于一个指针之后。任何指向 DST 的指针都会变成一个包含了完善 DST 类型信息的胖指针（详情见下方）。

Rust 暴露了两种主要的 DST 类型：

- trait objects：`dyn MyTrait`
- slices：`[T]`、`str` 及其他

Trait 对象代表某种类型，实现了它所指定的 Trait。确切的原始类型被*删除*，以利于运行时的反射，其中包含使用该类型的所有必要信息的 vtable。补全 Trait 对象指针所需的信息是 vtable 指针，被指向的对象的运行时的大小可以从 vtable 中动态地获取。

一个 slice 只是一些只读的连续存储——通常是一个数组或 `Vec`。补全一个 slice 指针所需的信息只是它所指向的元素的数量，指针的运行时大小只是静态已知元素的大小乘以元素的数量。

结构实际上可以直接存储一个 DST 作为其最后一个字段，但这也会使它们自身成为一个 DST：

```
// 不能直接存储在栈上
struct MySuperSlice {
    info: u32,
    data: [u8],
}
```

如果这样的类型没有方法来构造它，那么它在很大程度上来看是没啥用的。目前，唯一支持的创建自定义 DST 的方法是使你的类型成为泛型，并执行*非固定大小转换（unsizing coercion）*：

```rust
struct MySuperSliceable<T: ?Sized> {
    info: u32,
    data: T,
}

fn main() {
    let sized: MySuperSliceable<[u8; 8]> = MySuperSliceable {
        info: 17,
        data: [0; 8],
    };

    let dynamic: &MySuperSliceable<[u8]> = &sized;

    // 输出: "17 [0, 0, 0, 0, 0, 0, 0, 0]"
    println!("{} {:?}", dynamic.info, &dynamic.data);
}
```

（是的，自定义 DST 目前仅仅是一个基本半成品的功能。）

零大小类型 (ZSTs)

Rust 也允许类型指定他们不占空间：

```rust
struct Nothing; // 无字段意味着没有大小

// 所有字段都无大小意味着整个结构体无大小
struct LotsOfNothing {
    foo: Nothing,
    qux: (),      // 空元组无大小
    baz: [u8; 0], // 空数组无大小
}
```

就其本身而言，零尺寸类型（ZSTs）由于显而易见的原因是相当无用的。然而，就像 Rust 中许多奇怪的布局选择一样，它们的潜力在通用语境中得以实现。在 Rust 中，任何产生或存储 ZST 的操作都可以被简化为无操作（no-op）。首先，存储它甚至没有意义——它不占用任何空间。另外，这种类型的值只有一个，所以任何加载它的操作都可以直接凭空产生它——这也是一个无操作（no-op），因为它不占用任何空间。

这方面最极端的例子之一是 Set 和 Map。给定一个 `Map<Key, Value>`，通常可以实现一个 `Set<Key>`，作为 `Map<Key, UselessJunk>` 的一个薄封装。在许多语言中，这将需要为无用的封装分配空间，并进行存储和加载无用封装的工作，然后将其丢弃。对于编译器来说，证明这一点是不必要的，是一个困难的分析。

然而在 Rust 中，我们可以直接说 `Set<Key> = Map<Key, ()>`。现在 Rust 静态地知道每个加载和存储都是无用的，而且没有分配有任何大小。其结果是，单例化的代码基本上是 HashSet 的自定义实现，而没有 HashMap 要支持值所带来的开销。

安全的代码不需要担心 ZST，但是*不安全的*代码必须小心没有大小的类型的后果。特别是，指针偏移是无操作的，而分配器通常需要一个非零的大小。

请注意，对 ZST 的引用（包括空片），就像所有其他的引用一样，必须是非空的，并且适当地对齐。解引用 ZST 的空指针或未对齐指针是未定义的行为，就像其他类型的引用一样。

空类型

Rust 还允许声明*不能被实例化*的类型。这些类型只能在类型层讨论，而不能在值层讨论。空类型可以通过指定一个没有变体的枚举来声明：

```
enum Void {} // 没有变体的类型 = 空类型
```

空类型甚至比 ZST 更加边缘化。空类型的主要作用是为了让某个类型不可达。例如，假设一个 API 需要在一般情况下返回一个结果，但一个特定的情况实际上是不可能的。实际上可以通过返回一个 `Result<T, Void>` 来在类型级别上传达这个信息。API 的消费者可以放心地 unwrap 这样一个结果，因为他们知道这个值在本质上不可能是 `Err`，因为这需要提供一个 `Void` 类型的值。

原则上，Rust 可以基于这个事实做一些有趣的分析和优化，例如，`Result<T, Void>` 只表示为 `T`，因为 `Err` 的情况实际上并不存在（严格来说，这只是一种优化，并不保证，所以例如将一个转化为另一个仍然是 UB）。

下面的例子*本来应该*可以编译的：

```
enum Void {}

let res: Result<u32, Void> = Ok(0);

// 不存在 Err 的情况，所以 Ok 实际上永远都能匹配成功
let Ok(num) = res;
```

但现在还不让这么玩儿。

关于空类型的最后一个微妙的细节是，构造一个指向它们的原始指针实际上是有效的，但对它们的解引用是未定义行为，因为那是没有意义的。

我们建议不要用 `*const Void` 来模拟 C 的 `void*` 类型。很多人之前这样做，但很快就遇到了麻烦，因为 Rust 没有任何安全防护措施来防止用不安全的代码来实例化空类型，如果你这样做了，就是未定义行为。因为开发者有将原始指针转换为引用的习惯，而构造一个 `&Void` *也*是未定义行为，所以这尤其成问题。

`*const ()`（或等价物）对 `void*` 来说效果相当好，可以做成引用而没有任何安全问题。它仍然不能阻止你试图读取或写入数值，但至少它可以编译成一个 no-op 而不是 UB。

外部类型

有一个已被接受的 RFC 来增加具有未知大小的适当类型，称为 *extern 类型*，这将让 Rust 开发人员更准确地模拟像 C 的 `void*` 和其他“声明但从未定义”的类型。然而，截至 Rust 2018，该功能在 `size_of_val::<MyExternType>()` 应该如何表现方面遇到了一些问题。

可选的数据布局

Rust 允许你指定不同于默认的数据布局策略，并为你提供了不安全代码指南。

repr(C)

这是最重要的“repr”。它的意图相当简单：做 C 所做的事。字段的顺序、大小和对齐方式与你在 C 或 C++ 中期望的完全一样。任何你期望通过 FFI 边界的类型都应该有 `repr(C)`，因为 C 是编程世界的语言框架。这对于合理地使用数据布局做更多的技巧也是必要的，比如将值重新解释为不同的类型。

我们强烈建议使用rust-bindgen和/或cbindgen来为你管理 FFI 的边界。Rust 团队与这些项目紧密合作，以确保它们能够稳健地工作，并与当前和未来关于类型布局和 `repr`s 的保证兼容。

必须记住 `repr(C)` 与 Rust 更奇特的数据布局功能的互动。由于它具有“用于 FFI”和“用于布局控制”的双重目的，`repr(C)` 可以应用于那些如果通过 FFI 边界就会变得无意义或有问题的类型：

- ZST 仍然是零大小，尽管这不是 C 语言的标准行为，而且明确违背了 C++ 中空类型的行为，即它们仍然应该消耗一个字节的空间
- DST 指针（宽指针）和 tuple 在 C 语言中没有对应的概念，因此从来不是 FFI 安全的
- 带有字段的枚举在 C 或 C++ 中也没有对应的概念，但是类型的有效桥接是被定义的
- 如果 `T` 是一个FFI 安全的非空指针类型，`Option<T>` 被保证具有与 `T` 相同的布局和 ABI，因此也是 FFI 安全的。截至目前，这包括 `&`、`&mut` 和函数指针，所有这些都不能为空。
- 就 `repr(C)` 而言，元组结构和结构一样，因为与结构的唯一区别是字段没有命名。
- `repr(C)` 相当于无字段枚举的 `repr(u*)` 之一（见下一节）。选择的大小是目标平台的 C 应用二进制接口（ABI）的默认枚举大小。请注意，C 语言中的枚举表示法是实现定义的，所以这实际上是一个“最佳猜测”。特别是，当对应的 C 代码在编译时带有某些标志时，这可能是不正确的。
- 带有 `repr(C)` 或 `repr(u*)` 的无字段枚举仍然不能在没有相应变量的情况下设置为整数值，尽管这在 C 或 C++ 中是允许的行为。如果（不安全地）构造一个枚举的实例，但不与它的一个变体相匹配，这是未定义的行为(这使得详尽的匹配可以继续被编写和编译为正常行为)。

repr(transparent)

`#[repr(transparent)]` 只能用于只有单个非零大小字段（可能还有其他零大小字段）的结构或者单变体 enum 中。其效果是，整个结构的布局和 ABI 被保证与该字段相同。

注意：有一个叫做 `transparent_unions` 的 nightly 的特性，可以让你对 union 指定 `repr(transparent)`。不过由于设计上的一些顾虑，这个特性目前还未稳定，参考issue-

60405。

我们的目标是使单一字段和结构/枚举之间的转换成为可能。一个例子是`UnsafeCell`，它可以被转换为它所包装的类型。（`UnsafeCell`也用了一个不稳定的特性no_niche，所以当它嵌套其它类型的时候，它的 ABI 也并没有一个稳定的保证。）

另外，当我们通过 FFI 传递结构/枚举，并且其中内部字段类型是另一端所需的类型时，我们能保证这是正确的。特别是，这对于`struct Foo(f32)`或者`enum Foo { Bar(f32) }`总是具有与`f32`相同的 ABI 是必要的。

只有在唯一的字段为`pub`或其内存布局在文档中所承诺的情况下，该 repr 才被视为一个类型的公共 ABI 的一部分。否则，该内存布局不应被其他 crate 所依赖。

更多细节可以参考RFC 1758和RFC 2645。

repr(u*), repr(i*)

这些指定了使无字段枚举的大小。如果判别符超过了它可以容纳的整数，就会产生一个编译时错误。你可以通过将溢出的元素明确设置为 0 来手动要求 Rust 允许这样做。

术语“无字段枚举”仅意味着该枚举在其任何变体中都没有数据。没有`repr(u*)`或`repr(C)`的无字段枚举仍然是一个 Rust 本地类型，没有稳定的 ABI 表示。添加`repr`会使它在 ABI 上被视为与指定的整数大小完全相同。

如果枚举有字段，其效果类似于`repr(C)`的效果，因为该类型有一个定义的布局。这使得将枚举传递给 C 代码或者访问该类型的原始表示并直接操作其标记和字段成为可能，详见RFC。

这些“repr”对结构（struct）没有作用。

在含有字段的枚举中加入明确的`repr(u*)`、`repr(i*)`或`repr(C)`可以抑制空指针优化，比如：

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

#[repr(u8)]
enum MyReprOption<T> {
    Some(T),
    None,
}

assert_eq!(8, size_of::<MyOption<&u16>>());
assert_eq!(16, size_of::<MyReprOption<&u16>>());
```

空指针优化针对无字段且拥有`repr(u*)`、`repr(i*)`或`repr(C)`的枚举仍然生效。

repr(packed)

`repr(packed)` 强制 Rust 去掉任何填充，只将类型对齐到一个字节。这可能会改善内存占用，但可能会有其他负面的副作用。

特别是，大多数架构*强烈地*希望数值被对齐。这可能意味着不对齐的加载会受到惩罚（x86），甚至会出现故障（一些 ARM 芯片）。对于简单的情况，如直接加载或存储一个已打包的字段，编译器可能能够用移位和掩码来解决对齐问题。然而，如果你对一个已打包的字段进行引用，编译器就不太可能发出代码来避免无对齐的加载。

由于这可能导致未定义的行为，我们在 Lint 中已经实现了对应的检查，并且该行为会被认为是错误。

`repr(packed)` 是不能轻易使用的，除非你有极端的要求，否则不应该使用这个。

这个 repr 是对 `repr(C)` 和 `repr(Rust)` 的修改。

repr(align(n))

`repr(align(n))` (其中 `n` 是 2 的幂)强制类型*至少*按照 n 对齐。

这可以实现一些技巧，比如确保数组中的相邻元素不会彼此共享同一个缓存行（这可能会加快某些类型的并发代码）。

这是 `repr(C)` 和 `repr(Rust)` 的一个修改版本，它与 `repr(packed)` 不兼容。

所有权和生命周期

所有权是 Rust 的突破性功能。它使 Rust 能够做到完全的内存安全和高效，同时避免了垃圾回收。在详细介绍所有权系统之前，我们将考虑这一设计的动机。

我们将假设你同意垃圾收集（GC）并不总是一个最佳解决方案，而且在某些情况下手动管理内存更为适合。如果你不接受这一点，我是否可以让你对另一种语言感兴趣?

不管你对 GC 的看法如何，它显然是一个使代码更安全的*好办法*，你永远不必担心你的对象会在引用失效前就被释放。这是一个 C 和 C++ 程序需要处理的普遍存在的问题。比如下面这个简单的错误，我们所有使用过非 GC 语言的人都曾经犯过：

```
fn as_str(data: &u32) -> &str {
    // 计算出字符串
    let s = format!("{}", data);

    // 不好！我们返回了一个仅仅在函数中存在的变量的引用！
    // 悬挂指针！释放后使用！哎呀！
    // （这在 Rust 中无法编译通过）
    &s
}
```

这正是 Rust 的所有权系统所要解决的问题。Rust 知道`&s`所在的范围，因此可以防止它逃逸。然而，这是一个简单的案例，即使是 C 语言的编译器也能合理地抓住。随着代码越来越大，指针被送入各种函数，事情变得越来越复杂。最终，C 语言编译器会倒下，无法进行足够的转义分析来证明你的代码不健全。因此，它将被迫接受你的程序，假设它是正确的。

这种情况永远不会发生在 Rust 上，Rust 要求程序员来向编译器证明一切是正确的。

当然，Rust 围绕所有权的故事要比仅仅验证引用不脱离其所有者的范围要复杂得多，这是因为确保指针始终有效要比这复杂得多。例如，在这段代码中：

```
let mut data = vec![1, 2, 3];
// 获取内部元素的引用
let x = &data[0];

// 不好！ `push` 操作导致 `data` 的存储空间重新分配了
// 悬挂指针！释放后使用！哎呀！
// （这在 Rust 中无法编译通过）
data.push(4);

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

简单的作用域分析不足以防止这个 bug，因为`data`事实上确实存活得足够久，满足我们的需求。然而，当我们对它有一个引用时，它被*改变*了。这就是为什么 Rust 要求任何引用都要冻结引用者和其所有者。

引用

有两种类型的引用：

- 共享的引用：`&`
- 可变引用：`&mut`

它们遵守以下规则：

- 一个引用的生命周期不能超过它所引用对象的生命周期
- 一个可变的引用不能有别名

这就是引用所遵循的整个模型。

当然，我们也许应该定义*别名*的含义：

```
error[E0425]: cannot find value `aliased` in this scope
 --> <rust.rs>:2:20
  |
2 |     println!("{}", aliased);
  |                    ^^^^^^^ not found in this scope

error: aborting due to previous error
```

不幸的是，Rust 还没有真正定义其别名模型。🙀

在我们等待 Rust 的设计者明确他们语言的语义时，让我们用下一节来讨论下在一般场景下别名到底是什么，以及它为什么重要。

别名

首先，让我们先说一些重要的注意事项：

- 为了便于讨论，我们将使用最广泛的别名定义。Rust 的定义可能会有更多限制，以考虑到可变性和有效性。
- 我们将假设一个单线程的、无中断的执行，我们还将忽略像内存映射硬件这样的东西。Rust 假定这些事情不会发生，除非你明确告诉它会发生。更多细节，请参阅并发性章节。

所以，我们现行的定义是：如果变量和指针指向内存的重叠区域，那么它们就是*别名*。

为什么别名很重要

为什么我们需要关注别名呢?

让我们看下这个例子：

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
    // 记住一点: 如果 `input>10`, 那么 `output` 永远为 `2`
}
```

我们*希望*能够把它优化成下面这样的函数：

```
fn compute(input: &u32, output: &mut u32) {
    let cached_input = *input; // 将 `*input` 中的内容保存在寄存器中
    if cached_input > 10 {
        // 如果输入比 10 大，优化之前的代码会将 output 设置为 1，然后乘以 2，
        // 结果一定返回 `2` （因为 `>10` 包括了 `>5` 的情况），
        // 因此这里可以进行优化，
        // 不对 output 重复赋值，直接将其设置为 2
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

在 Rust 中，这种优化应该是可行的。但对于几乎任何其他语言来说，它都不是这样的（除非是全局分析）。这是因为这个优化依赖于知道别名不会发生，而大多数语言在这方面是相当宽松的。具体来说，我们需要担心那些使“输入”和“输出”重叠的函数参数，如 `compute(&x, &mut x)`。

如果按照这样的输入，我们实际上执行的代码如下：

```
                    //  input ==  output == 0xabad1dea
                    // *input == *output == 20
if *input > 10 {    // true  (*input == 20)
    *output = 1;    // 同时覆盖了 input 引用的内容，因为它们实际上引用了同一块内存
}
if *input > 5 {     // false (*input == 1)
    *output *= 2;
}
                    // *input == *output == 1
```

我们的优化函数对于这个输入会产生`*output == 2`，所以在这种情况下，我们的优化就无法实现了。

在 Rust 中，我们知道这个输入是不可能的，因为`&mut`不允许被别名。所以我们可以安全地认为这种情况不会发生，并执行这个优化。在大多数其他语言中，这种输入是完全可能的，因此必须加以考虑。

这就是为什么别名分析很重要的原因：它可以让编译器进行有用的优化! 比如：

- 通过证明没有指针访问该值的内存来保持寄存器中的值
- 通过证明某些内存在我们上次读取后没有被写入，来消除读取
- 通过证明某些内存在下一次写入之前从未被读过，来消除写入
- 通过证明读和写之间不相互依赖来对指令进行移动或重排序

这些优化也用于证明更大的优化的合理性，如循环矢量化、常数传播和死代码消除。

在前面的例子中，我们利用`&mut u32`不能被别名的事实来证明对`*output`的写入不可能影响`*input`。这让我们把`*input`缓存在一个寄存器中，省去了读的过程。

通过缓存这个读，我们知道在`> 10`分支中的写不能影响我们是否采取`> 5`分支，使我们在`*input > 10`时也能消除一个读-修改-写（加倍`*output`）。

关于别名分析，需要记住的关键一点是，写是优化的主要危险。也就是说，阻止我们将读移到程序的任何其他部分的唯一原因是我们有可能将其与写到同一位置重新排序。

例如，在下面这个修改后的函数中，我们不需要担心别名问题，因为我们已经将唯一一个写到`*output`的地方移到了函数的最后。这使得我们可以自由地重新排序在它之前发生的对`*input`的读取：

```
fn compute(input: &u32, output: &mut u32) {
    let mut temp = *output;
    if *input > 10 {
        temp = 1;
    }
    if *input > 5 {
        temp *= 2;
    }
    *output = temp;
}
```

我们仍然依靠别名分析来假设`temp`没有别名`input`，但是证明要简单得多：局部变量的值不能被在它被声明之前就存在的东西所别名。这是每一种语言都可以自由做出的假设，因此这个版本的函数可以在任何语言中按照我们想要的方式进行优化。

这就是为什么 Rust 将使用的“别名”的定义可能涉及到一些有效性和可变性的概念：如果没有任何实际写入内存的情况发生，我们实际上并不关心别名是否发生。

当然，Rust 的完整别名模型还必须考虑到函数调用（可能会改变我们看不到的东西）、原始指针（它本身没有别名要求）和 UnsafeCell（它让`&`的引用被改变）等东西。

生命周期

Rust 通过*生命周期*来执行相关的规则。生命周期是指一个引用必须有效的代码区域，这些区域可能相当复杂，因为它们对应着程序中的执行路径。这些执行路径中甚至可能存在空洞(译者注: 空洞是指一个引用的生命周期可能不是一个连续的代码区域，中间可能有跳跃)，因为我们可能会先使一个引用失效，之后再重新初始化并使用它。包含引用（或假装包含）的类型也可以用生命周期来标记，这样 Rust 就可以防止它们也被失效。

在我们大多数例子中，生命周期将与作用域重合，这是因为我们的例子很简单。下面将介绍它们不重合的更复杂的情况。

在一个函数体中，Rust 通常不需要你明确地命名所涉及的生命周期。这是因为一般来说，在本地环境中谈论生命周期是没有必要的；Rust 拥有所有的信息，并且可以尽可能地以最佳方式解决所有问题。Rust 还会引入许多匿名作用域和临时变量, 你不必显式写出它们, 代码也可以跑通。

然而，一旦你跨越了函数的边界，你就需要开始考虑生命周期了。生命周期是用撇号表示的：`'a`、`'static`。为了尝试使用生命周期，我们将假装我们被允许用生命周期来标记作用域，并尝试手动解一下本章开头例子的语法糖。

我们之前的例子使用了一种*激进*的语法糖——甚至是高果糖玉米糖浆——因为明确地写出所有东西是*非常繁琐*的。所有的 Rust 代码都依赖于积极的推理和对“显而易见”的东西的删除。

一个特别有趣的语法糖是，每个`let`语句都隐含地引入了一个作用域。在大多数情况下，这其实并不重要。然而，这对那些相互引用的变量来说确实很重要。作为一个简单的例子，让我们对这段简单的 Rust 代码进行完全解糖：

```
let x = 0;
let y = &x;
let z = &y;
```

借用检查器总是试图最小化生命周期的范围，所以它很可能会脱糖为以下内容：

```
// NOTE: `'a: {` 和 `&'b x` 不是有效的语法，这里只是为了说明 lifetime 的概念
'a: {
    let x: i32 = 0;
    'b: {
        // y 的生命周期为 'b，因为这已经足够好
        let y: &'b i32 = &'b x;
        'c: {
            // 'c 同上所示
            let z: &'c &'b i32 = &'c y; // "a reference to a reference to an
i32" (with lifetimes annotated)
        }
    }
}
```

哇，这真是……太可怕了！让我们花点时间感谢 Rust 让这一切变得简单。

实际上，传递一个引用到外部作用域将导致 Rust 推断出一个更大的生命周期。

```
let x = 0;
let z;
let y = &x;
z = y;
```

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // y 的生命周期一定为 'b，因为对 x 的引用被传递到了 'b 这个作用域
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

例子：超出所有者生命周期的引用

让我们看看之前的那些例子：

```
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s
}
```

解语法糖后：

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s;
    }
}
```

`as_str` 的这个签名接收了一个具有*某个*生命周期的 u32 的引用，并返回一个可以*存活同样长*的 str 的引用。我们已经大致能猜到为什么这个函数签名可能是个麻烦了，这意味着我们要找的那个 str 要在 u32 的引用所处的作用域上，或者*甚至在更大的作用域上*。这要求有点高。

然后我们继续计算字符串 `s`，并返回它的一个引用。由于我们的函数的契约规定这个引用必须超过 `'a`，这就是我们推断出的引用的生命周期。不幸的是，`s` 被定义在作用域 `'b` 中，所以唯一合理的方法是 `'b` 包含 `'a`，这显然是错误的，因为 `'a` 必须包含函数调用本身。因此，我们创建了一个引用，它的生命周期超过了它的引用者，这正是我们所说的引用不能做的第一件事。编译器理所当然地直接报错。

为了更清楚地说明这一点，我们可以扩展这个例子：

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s
    }
}

fn main() {
    'c: {
        let x: u32 = 0;
        'd: {
            // 这里引入了一个匿名作用域，因为借用不需要在整个 x 的作用域内生效，
            // 这个函数必须返回一个在函数调用之前就存在的某个字符串的引用，事实显然不是这样
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

当然，这个函数的正确写法是这样的：

```
fn to_string(data: &u32) -> String {
    format!("{}", data)
}
```

我们必须在函数里面产生一个拥有所有权的值才能返回! 我们唯一可以返回一个`&'a str`的方法是，它在`&'a u32`的一个字段中，但显然不是这样的。

（实际上我们也可以直接返回一个字符串字面量，作为一个全局的字面量可以被认为是在堆栈的底部；尽管这对我们的实现*有一点*限制）。

示例：别名一个可变引用

来看另一个例子：

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b 这个生命周期范围如我们所愿地小（刚好够 println!）
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // 这里有一个临时作用域，我们不需要更长时间的 &mut 借用
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

这里的问题更微妙、更有趣。我们希望 Rust 拒绝这个程序，理由如下：我们有一个存活的共享引用 `x` 到 `data` 的一个子集，当我们试图把 `data` 的可变引用传给 `push` 时。这将创建一个可变引用的别名，而这将违反引用的*第二条*规则。

然而，这根本不是 Rust 认为这个程序有问题的原因。Rust 不理解 `x` 是对 `data` 的一个子集的引用。它根本就不理解 `Vec` 。它看到的是， `x` 必须在 `'b` 范围内保持存活才能被打印；接下来，`Index::index` 的签名要求我们对 `data` 的引用必须在 `'b` 范围内存活。当我们试图调用 `push` 时，它看到我们试图构造一个 `&'c mut data` 。Rust 知道 `'c` 包含在 `'b` 中，并拒绝了我们的程序，因为 `&'b data` 必然还存活着！

在这里我们看到，和我们真正想要保证的引用规则语义相比，生命周期系统要粗略得多。在大多数情况下，*这完全没问题*，因为它使我们不用花整天的时间向编译器解释我们的程序。然而，这确实意味着有部分程序对于 Rust 的*真正的*语义来说是完全正确的，但却被拒绝了，因为 lifetime 太傻了。

生命周期所覆盖的区域

一个引用（有时称为*borrow*）从它被创建到最后一次使用都是*存活*的。被 borrow 的值的生命周期只需要超过引用的生命周期就行。这看起来很简单，但有一些微妙之处。

下面的代码可以成功编译，因为在打印完 `x` 之后，它就不再需要了，所以它是悬空的还是别名的都无所谓（尽管变量 `x` *技术上*一直存活到作用域的最后）：

```
let mut data = vec![1, 2, 3];
let x = &data[0];
println!("{}", x);
// 这是可行的，因为不再使用 x，编译器也就缩短了 x 的生命周期
data.push(4);
```

然而，如果该值有一个析构器，析构器就会在作用域的末端运行。而运行析构器被认为是一种使用——显然是最后一次使用。所以，这将会编译报错：

```rust
#[derive(Debug)]
struct X<'a>(&'a i32);

impl Drop for X<'_> {
    fn drop(&mut self) {}
}

let mut data = vec![1, 2, 3];
let x = X(&data[0]);
println!("{:?}", x);
data.push(4);
// 编译器会在这里自动插入 drop 函数，也就意味着我们会访问 x 中引用的变量，因此编译失败
```

让编译器相信 `x` 不再有效的一个方法是在 `data.push(4)` 之前使用 `drop(x)` 。

此外，可能会有多种最后一次的引用使用，例如在一个条件的每个分支中：

```rust
let mut data = vec![1, 2, 3];
let x = &data[0];

if some_condition() {
    println!("{}", x); // 这是该分支中最后一次使用 x 这个引用
    data.push(4);      // 因此在这里 push 操作是可行的
} else {
    // 这里不存在对 x 的使用，对于这个分支来说，
    // x 创建即销毁
    data.push(5);
}
```

生命周期中可以有暂停，或者你可以把它看成是两个不同的借用，只是被绑在同一个局部变量上。这种情况经常发生在循环周围（在循环结束时写入一个变量的新值，并在下一次迭代的顶部最后一次使用它）。

```rust
let mut data = vec![1, 2, 3];
// x 是可变的（通过 mut 声明），因此我们可以修改 x 指向的内容
let mut x = &data[0];

println!("{}", x); // 最后一次使用这个引用
data.push(4);
x = &data[3]; // x 在这里借用了新的变量
println!("{}", x);
```

Rust 曾经一直保持着借用的生命，直到作用域结束，所以这些例子在旧的编译器中可能无法编译。此外，还有一些边界条件，Rust 不能正确地缩短借用的有效部分，即使看起来应该这样做，也不能编译。这些问题将随着时间的推移得到解决。

生命周期的局限

让我们来看以下代码：

```
#[derive(Debug)]
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self { &*self }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
    println!("{:?}", loan);
}
```

人们可能期望它能被编译成功，我们调用`mutate_and_share`，它可以暂时可变借用`foo`，但随后只返回一个共享引用。因此我们期望`foo.share()`能够成功，因为`foo`不应该被可变借用。

然而，当我们试图编译它时：

```
error[E0502]: cannot borrow `foo` as immutable because it is also borrowed as
mutable
  --> src/main.rs:12:5
   |
11 |     let loan = foo.mutate_and_share();
   |                --- mutable borrow occurs here
12 |     foo.share();
   |     ^^^ immutable borrow occurs here
13 |     println!("{:?}", loan);
```

这是为啥？好吧，我们得到的推理和上一节例 2完全一样。我们对程序进行解语法糖后，可以得到如下结果：

```
struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self { &'a *self }
    fn share<'a>(&'a self) {}
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
            'd: {
                Foo::share::<'d>(&'d foo);
            }
            println!("{:?}", loan);
        }
    }
}
```

由于`loan`的生命周期和`mutate_and_share`的签名，生命周期系统被迫将`&mut foo`扩展为`'c`的生命周期。然后当我们试图调用`share`时，它看到我们试图别名`&'c mut foo`，然后就炸了！

根据我们真正关心的引用语义，这个程序显然是正确的，但是生命周期系统太蠢了(原话是粗糙)，无法处理这个问题。

不正确地缩减借用

下面的代码无法编译成功，因为 Rust 发现`map`变量被借用了两次，并且不能推断出在第二次借用之前，第一次借用已经不需要了，所以保守地退回到使用整个作用域作为第一次借用的生命周期。不过不用担心，这个问题最终会得到解决：

```
fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut V
where
    K: Clone + Eq + Hash,
    V: Default,
{
    match map.get_mut(&key) {
        Some(value) => value,
        None => {
            map.insert(key.clone(), V::default());
            map.get_mut(&key).unwrap()
        }
    }
}
```

由于所施加的生命周期限制，`&mut map`的生命周期与其他可变的借用重叠，导致编译错误：

```
error[E0499]: cannot borrow `*map` as mutable more than once at a time
  --> src/main.rs:12:13
   |
4  |   fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut
V
   |                  -- lifetime `'m` defined here
...
9  |       match map.get_mut(&key) {
   |       -     --- first mutable borrow occurs here
   |  _____|
   | |
10 | |         Some(value) => value,
11 | |         None => {
12 | |             map.insert(key.clone(), V::default());
   | |             ^^^ second mutable borrow occurs here
13 | |             map.get_mut(&key).unwrap()
14 | |         }
15 | |     }
   | |_____- returning this value requires that `*map` is borrowed for `'m`
```

生命周期省略

为了使常见的模式更符合人体工程学，Rust 允许在函数签名中*省略*生命周期。

*生命周期位置*是指在一个类型中可以写入生命周期的任何地方。

```
&'a T
&'a mut T
T<'a>
```

生命周期位置可以作为“输入”或“输出”出现：

- 对于 `fn` 定义、`fn` 类型以及 Trait `Fn`、`FnMut` 和 `FnOnce`，输入是指形式参数的类型，而输出是指结果类型。所以 `fn foo(s: &str) -> (&str, &str)` 在输入位置有一个生命周期，在输出位置有两个生命周期。请注意，`fn` 方法定义的输入位置不包括方法的 `impl` 头中出现的生命周期（对于默认方法，也不包括 trait 头中出现的生命周期）
- 对于 `impl` 头，所有类型都是输入。所以 `impl Trait<&T> for Struct<&T>` 在输入位置上省略了两个生命周期，而 `impl Struct<&T>` 则省略了一个

省略规则如下：

- 在输入位置的每一个被省略的生命周期都成为一个独立的生命周期参数
- 如果正好有一个输入生命周期的位置（无论是否被省略），该生命周期将被分配给*所有*被省略的输出生命周期
- 如果有多个输入生命周期位置，但其中一个是 `&self` 或 `&mut self`，那么 `self` 的生命周期将被分配给*所有*被省略的输出生命周期
- 否则，省略一个输出生命周期是一个错误

示例：

```
fn print(s: &str);                                    // elided
fn print<'a>(s: &'a str);                             // expanded

fn debug(lvl: usize, s: &str);                        // elided
fn debug<'a>(lvl: usize, s: &'a str);                 // expanded

fn substr(s: &str, until: usize) -> &str;             // elided
fn substr<'a>(s: &'a str, until: usize) -> &'a str;   // expanded

fn get_str() -> &str;                                 // ILLEGAL

fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL

fn get_mut(&mut self) -> &mut T;                      // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // expanded

fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command                  //
elided
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command //
expanded

fn new(buf: &mut [u8]) -> BufWriter;                  // elided
fn new(buf: &mut [u8]) -> BufWriter<'_>;              // elided (with
`rust_2018_idioms`)
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // expanded
```

不受约束的生命周期

不安全的代码经常会凭空产生引用或生命周期，这种生命周期是以*无约束*的形式出现在世界中的。最常见的原因是对原始指针的解引用，这产生了一个具有无约束生命周期的引用。这样的生命周期会随着上下文的要求而变大。这实际上比简单地标记为`'static`更强大，因为例如`&'static &'a T`将无法通过类型检查，但无约束的生命周期将根据需要完美地塑造为`&'a &'a T`。然而，对于大多数意图和目的来说，这样的无约束生命周期可以被看作是`'static`。

几乎没有引用是`'static`的，所以这可能是错误的。`transmute`和`transmute_copy`是另外两个主要的违规者。我们应该尽可能快地约束一个无约束的生命周期，特别是当跨越函数边界的时候。

给定一个函数，任何不来自输入的输出生命周期都是无约束的，比如说：

```
fn get_str<'a>(s: *const String) -> &'a str {
    unsafe { &*s }
}
fn main() {
    let soon_dropped = String::from("hello");
    let dangling = get_str(&soon_dropped);
    drop(soon_dropped);
    println!("Invalid str: {}", dangling); // Invalid str: gə¨`
}
```

避免无约束生命周期的最简单方法是在函数边界使用生命周期省略。如果一个输出的生命周期被省略了，那么它*必须*被一个输入的生命周期所约束。当然，它也可能被错误的生命周期所约束，但这通常只会引起编译错误，而不是让内存安全被简单地违反。

在一个函数中，对生命周期的约束更容易出错。约束生命周期的最安全和最简单的方法是从一个具有约束的生命周期的函数中返回它。然而，如果这样做是不可接受的，可以将引用放在一个有特定生命周期的位置。不幸的是，我们不可能命名一个函数中涉及的所有生命周期。

Higher-Rank Trait Bounds (HRTBs)

Rust 的`Fn` trait 有一些黑魔法，例如，我们可以写出下面的代码：

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where F: Fn(&(u8, u16)) -> &u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

如果我们试图天真地用与生命周期部分相同的方式来对这段代码进行解语法糖，我们会遇到一些麻烦：

```
// NOTE: `&'b data.0` and `'x: {` is not valid syntax!
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    // where F: Fn(&'??? (u8, u16)) -> &'??? u8,
{
    fn call<'a>(&'a self) -> &'a u8 {
        (self.func)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
        let clo = Closure { data: (0, 1), func: do_it };
        println!("{}", clo.call());
    }
}
```

我们究竟应该如何表达`F`的 trait 约束上的生命周期？我们需要在那里提供一些生命周期，但是我们关心的生命周期在进入`call`的主体之前是不能被命名的! 而且，这并不是什么固定的生命周

期；`call`可以与`&self`在这一时刻上的*任一*生命周期一起使用。

要完成这个事情，需要使用到高阶 Trait 约束（HRTB）的魔力。我们的解语法糖方式如下：

```
where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
```

或者：

```
where F: for<'a> Fn(&'a (u8, u16)) -> &'a u8,
```

（其中`Fn(a, b, c) -> d`本身只是不稳定的*真正的*`*Fn`特性的语法糖）

`for<'a>`可以理解为“对于所有`'a`的可能”，并且基本上产生一个*无限的* F *必须*满足的 trait 约束的列表。不过不用紧张，在`Fn` trait 之外，我们遇到 HRTB 的地方不多，即使是那些地方，我们也有一个很好的魔法糖来处理普通的情况。

最终，我们可以把原本的代码重写成更加显式的样子：

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}
impl<F> Closure<F>
    where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}
fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }
fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

子类型和协变

Rust 使用生命周期来追踪借用和所有权之间的关系。但是，原生的生命周期实现可能过于严格，或者会允许未定义行为。

为了实现对生命周期的灵活使用并防止滥用，Rust 使用 **子类型** 和 **协变**。

让我们从一个例子开始。

```
// 注意：debug 需要两个具有相同生命周期的参数
fn debug<'a>(a: &'a str, b: &'a str) {
    println!("a = {a:?} b = {b:?}");
}

fn main() {
    let hello: &'static str = "hello";
    {
        let world = String::from("world");
        let world = &world; // 'world 的生命周期比 'static 短
        debug(hello, world);
    }
}
```

在一个保守的生命周期实现中，由于 `hello` 和 `world` 有不同的生命周期，我们可能会看到以下错误：

```
error[E0308]: mismatched types
 --> src/main.rs:10:16
   |
10 |         debug(hello, world);
   |                      ^
   |                      |
   |                      expected `&'static str`, found struct `&'world str`
```

这是相当不幸的。在这种情况下，我们希望接受的类型至少要和 `'world` 一样长。让我们尝试使用生命周期进行子类型化。

子类型化

子类型化是指一种类型可以替代另一种类型的概念。

我们定义 `Sub` 是 `Super` 的子类型（在本章中我们将使用表示法 `Sub <: Super`）。

这表示 `Super` 定义的 *要求* 集合被 `Sub` 完全满足。然后，`Sub` 可能有更多的要求。

现在，为了使用生命周期进行子类型化，我们需要定义一个生命周期的要求：

> `'a` 定义了一段代码区域。

既然我们为生命周期定义了一组要求，我们就可以定义它们之间的关系：

> 当且仅当 `'long` 定义一个 **完全包含** `'short` 的代码区域时， `'long <: 'short` 。

`'long` 可能定义了一个比 `'short` 更大的区域，但这仍符合我们的定义。

> 正如我们将在本章后面看到的，子类型化比这要复杂得多，但这个简单的规则在大多数情况下是非常好的直觉。 除非您编写不安全的代码，否则编译器将为您自动处理所有的特殊情况。

> 但这是 Rustonomicon。我们正在编写不安全的代码，所以我们需要了解这些东西是如何真正工作的，以及我们如何搞乱它。

回到我们上面的例子，我们可以说 `'static <: 'world` 。现在，让我们也接受子类型生命周期可以通过引用传递的想法（更多内容请参见 协变），例如 `&'static str` 是 `&'world str` 的子类型，然后我们可以将 `&'static str` 降级为 `&'world str` 。有了这个，上面的示例可以编译：

```
fn debug<'a>(a: &'a str, b: &'a str) {
    println!("a = {a:?} b = {b:?}");
}

fn main() {
    let hello: &'static str = "hello";
    {
        let world = String::from("world");
        let world = &world; // 'world 的生命周期比 'static 短
        debug(hello, world); // hello 从 `&'static str` 静默降级为 `&'world str`
    }
}
```

协变

在上面，我们简单地说明了 `'static <: 'b` 静默地暗示了 `&'static T <: &'b T` 。这使用了一个名为 *协变* 的性质。然而，这并不总是像这个例子那样简单。为了理解这一点，让我们尝试稍微扩展这个例子：

```
fn assign<T>(input: &mut T, val: T) {
    *input = val;
}

fn main() {
    let mut hello: &'static str = "hello";
    {
        let world = String::from("world");
        assign(&mut hello, &world);
    }
    println!("{hello}"); // 使用在被释放后的值 😿
}
```

在 `assign` 中，我们将 `hello` 引用设置为指向 `world`。 但是 `world` 在 `println` 使用 `hello` 之前就已经超出了作用域！

这是一个典型的在释放后使用错误！

我们第一反应可能是怪 `assign` 的实现，但实际上这里并没有什么错误。一个值想要赋值到一个具有相同类型的 `T` 也不奇怪。

问题是我们不能假设 `&mut &'static str` 和 `&mut &'b str` 是兼容的。 这意味着，即使 `'static` 是 `&'b` 的子类型， `&mut &'static str` 也 **不能** 是 `&mut &'b str` 的子类型。

协变是 Rust 借用的概念，用于定义泛型参数通过子类型之间的关系。

> 注意：为了方便起见，我们将定义一个泛型类型 `F<T>`，以便我们可以方便地讨论 `T`。希望这在上下文中是清楚的。

类型 `F` 的 *协变性* 是其输入子类型化如何影响其输出子类型化。在 Rust 中有三种协变。设两种类型 `Sub` 和 `Super`，其中 `Sub` 是 `Super` 的子类型：

- `F` 是 **协变的**，如果 `F<Sub>` 是 `F<Super>` 的子类型（子类型属性被传递）
- `F` 是 **逆变的**，如果 `F<Super>` 是 `F<Sub>` 的子类型（子类型属性被 "反转"）
- 否则， `F` 是 **不变的** （不存在子类型关系）

如果我们回想上面的例子， `&'a T` 在 `'a` 上是协变的，因此我们可以对其进行子类型化。我们可以这样说。

此外，我们注意到不能将 `&mut &'a U` 视为 `&mut &'b U` 的子类型，因此我们可以说 `&mut T` 在 `T` 上是 *不变的*

以下是一些其他泛型类型及其协变性的表格：

	'a	T	U
`&'a T`	协变	协变	
`&'a mut T`	协变	不变	

	'a	T	U
`Box<T>`		协变	
`Vec<T>`		协变	
`UnsafeCell<T>`		不变	
`Cell<T>`		不变	
`fn(T) -> U`		**逆**变	协变
`*const T`		协变	
`*mut T`		不变	

这些可以简单地解释为其他类型的关系：

- `Vec<T>` 以及所有其他拥有指针和集合遵循与 `Box<T>` 相同的逻辑
- `Cell<T>` 以及所有其他内部可变性类型遵循与 `UnsafeCell<T>` 相同的逻辑
- 具有内部可变性的 `UnsafeCell<T>` 使其具有与 `&mut T` 相同的协变属性
- `*const T` 遵循 `&T` 的逻辑
- `*mut T` 遵循 `&mut T`（或 `UnsafeCell<T>`）的逻辑

有关其他类型，请参见参考手册的 "协变" 部分。

> 注意：语言中唯一的逆变来源是函数参数，这就是为什么它实际上在实践中很少出现。调用逆变涉及到函数指针的高阶编程，这些函数指针需要具有特定生命周期（而不是通常的 "任意生命周期"）的引用，而这将涉及更高级别的生命周期，它们可以独立于子类型化工作。

现在我们对协变有了更正式的理解，让我们更详细地讨论一些例子。

```
fn assign<T>(input: &mut T, val: T) {
    *input = val;
}

fn main() {
    let mut hello: &'static str = "hello";
    {
        let world = String::from("world");
        assign(&mut hello, &world);
    }
    println!("{hello}");
}
```

运行这个例子会得到什么？

```
error[E0597]: `world` does not live long enough
  --> src/main.rs:9:28
   |
6  |     let mut hello: &'static str = "hello";
   |                    ------------ type annotation requires that `world` is
borrowed for `'static`
...
9  |         assign(&mut hello, &world);
   |                            ^^^^^^ borrowed value does not live long enough
10 |     }
   |     - `world` dropped here while still borrowed
```

很好，它不能编译！让我们详细了解这里发生了什么。

首先让我们看下 `assign` 函数：

```
fn assign<T>(input: &mut T, val: T) {
    *input = val;
}
```

它只是接收一个可变引用和一个值，然后将该值覆盖。这个函数的关键是它创建了一个类型相等约束。它在签名中清楚地说，被引用和值必须是 *完全相同* 的类型。

与此同时，在调用者中，我们传入 `&mut &'static str` 和 `&'world str`。

由于 `&mut T` 在 `T` 上是不变的，所以编译器得出结论，它不能对第一个参数应用任何子类型化，因此 `T` 必须是 `&'static str`。

这与 `&T` 情况相反：

```
fn debug<T: std::fmt::Debug>(a: T, b: T) {
    println!("a = {a:?} b = {b:?}");
}
```

尽管 `a` 和 `b` 必须具有相同的类型 `T`，但由于 `&'a T` 在 `'a` 上是协变的，我们可以执行子类型化。因此，编译器决定 `&'static str` 可以变为 `&'b str` 当且仅当 `&'static str` 是 `&'b str` 的子类型，这将在 `'static <: 'b` 的情况下成立。这是正确的，因此编译器愿意继续编译此代码。

事实证明，Box（以及 Vec，HashMap 等）协变的原因与生命周期协变的原因相似：只要你尝试将它们放入诸如可变引用之类的东西中，就会继承不变性，从而阻止你做任何坏事。

然而，Box 更容易关注重引用的按值方面，我们之前部分忽略了这一点。

与许多允许值在任何时候被自由别名的语言不同，Rust 有一个非常严格的规则：如果您可以修改或移动一个值，那么您必须确保是唯一一个可以访问该值的人。

考虑以下代码：

```
let hello: Box<&'static str> = Box::new("hello");

let mut world: Box<&'b str>;
world = hello;
```

我们已经忘记了 `hello` 的 `'static` 存活时间也没有任何问题，因为当我们将 `hello` 移动到只知道它活跃的变量时，**我们销毁了唯一记住它存活时间更长的东西！**

现在还剩一件事要解释：函数指针。

要了解为什么 `fn(T) -> U` 应该在 `U` 上是协变的，请考虑以下签名：

```
fn get_str() -> &'a str;
```

该函数声明可以生成一个由某个生命周期 `'a` 绑定的 `str`。因此，使用以下签名的函数也是完全有效的：

```
fn get_static() -> &'static str;
```

所以当函数被调用时，它只期望一个至少活着 `&str` 生命周期的值，实际生活的是否更长并不重要。

然而，相同的逻辑不能应用于*参数*。考虑尝试满足：

```
fn store_ref(&'a str);
```

使用：

```
fn store_static(&'static str);
```

第一个函数可以接受任何字符串引用，只要它至少活到 `'a`，但第二个函数不能接受一个生命周期小于 `'static` 的字符串引用，这将导致冲突。协变不适用于此。但是，如果我们将其反过来，实际上*确实*可以工作！如果我们需要一个可以处理 `&'static str` 的函数，一个可以处理*任意*引用生命周期的函数肯定可以很好地工作。

让我们看看实践中的例子

```
thread_local! {
    pub static StaticVecs: RefCell<Vec<&'static str>> =
RefCell::new(Vec::new());
}

/// 将给定的输入保存到一个线程局部的 `Vec<&'static str>`
fn store(input: &'static str) {
    StaticVecs.with(|v| {
        v.borrow_mut().push(input);
    })
}

/// 使用相同生命周期的输入调用函数！
fn demo<'a>(input: &'a str, f: fn(&'a str)) {
    f(input);
}

fn main() {
    demo("hello", store); // "hello" 是 'static。可以正常调用 `store`

    {
        let smuggle = String::from("smuggle");

        // `&smuggle` 不是静态的。如果我们用 `&smuggle` 调用 `store`，
        // 我们将把一个无效的生命周期推入 `StaticVecs`。
        // 因此，`fn(&'static str)` 不能是 `fn(&'a str)` 的子类型
        demo(&smuggle, store);
    }

    StaticVecs.with(|v| {
        println!("{:?}", v.borrow()); // 使用在被释放后的值 😿
    });
}
```

这就是为什么函数类型，与语言中的其他内容不同，是*逆*变量。

现在，这对于标准库提供的类型来说是很好而已，但是如何确定*您*定义的类型的协变呢？结构体，非正式地说，继承了其字段的协变性。如果一个结构体 `MyType` 有一个泛型参数 `A`，并且在字段 `a` 中使用了 `A`，那么 MyType 对 `A` 的协变程度与 `a` 对 `A` 的协变程度完全相同。

然而，如果 `A` 被多个字段使用：

- 如果 `A` 的所有用途都是协变的，则 MyType 在 `A` 上是协变的
- 如果 `A` 的所有用途都是逆变的，则 MyType 在 `A` 上是逆变的
- 否则，MyType 在 `A` 上是不变的

```rust
use std::cell::Cell;

struct MyType<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H, In, Out, Mixed> {
    a: &'a A,      // 对 'a 和 A 是协变的
    b: &'b mut B, // 对 'b 是协变的，对 B 是不变的

    c: *const C,  // 对 C 是协变的
    d: *mut D,    // 对 D 是不变的

    e: E,         // 对 E 是协变的
    f: Vec<F>,    // 对 F 是协变的
    g: Cell<G>,   // 对 G 是不变的

    h1: H,        // 本来也会对 H 是协变的，但...
    h2: Cell<H>,  // 对 H 是不变的，因为不变性在所有冲突中都是胜利者

    i: fn(In) -> Out,       // 对 In 是逆变的，对 Out 是协变的

    k1: fn(Mixed) -> usize, // 本来会对 Mixed 是逆变的，但...
    k2: Mixed,              // 对 Mixed 是不变的，因为不变性在所有冲突中都是胜利者
}
```

现在你对 Rust 中的子类型和协变概念应该有了更深入的理解。尽管本章涵盖了许多概念，但通过编译器和类型系统所提供的严密检查来确保这些规则得到遵循和安全操作。当编写泛型代码时，要确保您正确理解子类型化和协变性，以避免出现意外错误和潜在安全问题。

丢弃检查

我们已经看到了生命周期如何为我们提供了一些相当简单的规则来确保我们永远不会读到悬空的引用。但是到目前为止， *outlives* 是一种包容的关系。也就是说，当我们谈论`'a: 'b`时，`'a`可以和`'b`的寿命一样长。乍一看，这似乎是一个无意义的特点。没有什么东西会和另一个东西同时被丢弃，对吗？这就是为什么我们对以下`let`语句解语法糖：

```
let x;
let y;
```

解语法糖：

```
{
    let x;
    {
        let y;
    }
}
```

有一些更复杂的情况不可能用作用域来解语法糖，但顺序是被定义好的——变量按其定义的相反顺序丢弃，结构体和元组的字段按其定义的顺序丢弃。在 RFC 1857 中有一些关于丢弃顺序的更多细节。

让我们来试试：

```
let tuple = (vec![], vec![]);
```

左边的 Vec 先被丢弃。但这是否意味着在借用检查器的眼中，右边 Vec 一定活得更长？这个问题的答案是 *No*。借用检查器可以分别跟踪元组的字段，但它仍然无法知道 哪个Vec 元素活得更久，因为 Vec 元素是通过借用检查器不理解的纯库代码手动丢弃的。

那么，我们为什么要关心呢？是因为如果类型系统不小心，它可能会意外地产生悬空指针。比如下面这个简单的程序：

```
struct Inspector<'a>(&'a u8);

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
}
```

这个程序看起来很合理，而且可以编译。事实上，`days`的寿命并没有严格地超过`inspector`的寿命，这并不重要。只要`inspector`还活着，`days`也会活着。

然而，如果我们添加一个析构器，程序就不会再编译了！

```
struct Inspector<'a>(&'a u8);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("I was only {} days from retirement!", self.0);
    }
}

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
    // 如果 `days` 碰巧在这里被析构了，然后 Inspector 才被析构，就会造成`内存释放后读取`
的问题！
}
```

```
error[E0597]: `world.days` does not live long enough
  --> src/main.rs:19:38
   |
19 |     world.inspector = Some(Inspector(&world.days));
   |                                      ^^^^^^^^^^^ borrowed value does not
live long enough
...
22 | }
   | -
   | |
   | `world.days` dropped here while still borrowed
   | borrow might be used here, when `world` is dropped and runs the destructor
for type `World<'_>`
```

你可以尝试改变字段的顺序，或者用一个元组来代替struct，但还是不能编译。

实现`Drop`可以让`Inspector`在被丢弃时执行一些代码。使得它有可能观察到那些本该和它生命周期一样长的类型实际上是先被销毁的。

有趣的是，只有泛型需要担心这个问题。如果它们不是泛型的，那么它们唯一能承载的寿命就是`'static`，它将真正地一直活着。这就是为什么这个问题被称为 *sound generic drop*。健壮的泛型丢弃是由 *drop checker* 强制执行的。截止到本文写作时，关于丢弃检查器（也被称为`dropck`）如何验证类型的一些更细微的细节还完全是未知数。然而，“大规则”是我们这一节所关注的微妙之处：

对于一个泛型类型来说，要健壮地实现 drop，其泛型参数必须严格超过它的寿命。

遵守这一规则（通常）是满足借用检查器的必要条件；遵守这一规则是健壮地泛型丢弃的充分不必要条件。即如果你的类型遵守了这个规则，那么它的 drop 肯定是健壮的。

不一定要满足上述规则的原因是，有些 Drop 实现不会访问借用的数据，即使他们的类型给了他们这种访问的能力，或者因为我们知道具体的 Drop 顺序，且借用的数据依旧完好，即使借用检查器不知道。

例如，上述`Inspector`例子的这个变体永远不会访问借来的数据：

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
    // 假设 `days` 刚好在这里析构了，
    // 并且假设析构函数可以确保：该函数确保不会访问对 `days` 的引用
}
```

同样地，下面这个变体也不会访问借来的数据：

```
struct Inspector<T>(T, &'static str);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<T> {
    inspector: Option<Inspector<T>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
    // 假设 `days` 刚好在这里析构了，
    // 并且假设析构函数可以确保：该函数确保不会访问对 `days` 的引用
}
```

然而，上述两种变体在分析 `fn main` 时都被借用检查器拒绝了，说 `days` 的生命周期不够长。

原因是对 `main` 的借用检查分析时，借用检查器并不了解每个 `Inspector` 的 `Drop` 实现的内部情况。就借用检查器在分析 `main` 时知道的情况来看，检查器的析构器主体可能会访问这些借用的数据。

因此，丢弃检查器强迫一个值中的所有借用数据的生命周期严格地超过该值的生命周期。

一种逃逸方法

丢弃检查的精确规则在未来可能会减少限制。

目前的分析是故意保守和琐碎的；它强制一个值中的所有借来的数据的生命周期超过该值的生命周期，这当然是合理的。

未来版本的语言可能会使分析更加精确，以减少正确代码被拒绝为不安全的情况。这将有助于解决诸如上述两个`Inspector`知道在销毁时不访问借来的数据的情况。

但与此同时，有一个不稳定的属性，可以用来断言（不安全的）泛型的析构器 *保证* 不访问任何失效数据，即使它的类型赋予它这样的能力。

这个属性被称为`may_dangle`，是在RFC1327中引入的。要在上面的`Inspector`上用上它，我们可以这么写：

```
#![feature(dropck_eyepatch)]

struct Inspector<'a>(&'a u8, &'static str);

unsafe impl<#[may_dangle] 'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<'a> {
    days: Box<u8>,
    inspector: Option<Inspector<'a>>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
}
```

使用这个属性需要将`Drop`标记为`unsafe`，因为编译器没有检查隐含的断言，即没有访问潜在的失效数据（例如上面的`self.0`）。

该属性可以应用于任何数量的生命周期和类型参数。在下面的例子中，我们断言我们没有访问寿命为`'b`的引用后面的数据，并且`T`的唯一用途是 move 或 drop，但是从`'a`和`U`中省略了该属性，因为我们确实访问具有该生命周期和该类型的数据。

```
#![feature(dropck_eyepatch)]
use std::fmt::Display;

struct Inspector<'a, 'b, T, U: Display>(&'a u8, &'b u8, T, U);

unsafe impl<'a, #[may_dangle] 'b, #[may_dangle] T, U: Display> Drop for
Inspector<'a, 'b, T, U> {
    fn drop(&mut self) {
        println!("Inspector({}, _, _, {})", self.0, self.3);
    }
}
```

有时很明显，不可能发生这样的访问，比如上面的情况。然而，当处理一个通用类型的参数时，这种访问可能会间接地发生，这种间接访问的例子是：

- 调用一个回调
- 通过 trait 方法调用

（未来对语言的修改，如 impl 的特化，可能会增加这种间接访问的其他途径。）

下面是一个回调的例子：

```
struct Inspector<T>(T, &'static str, Box<for <'r> fn(&'r T) -> String>);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        // 如果 `T` 是 `&'a _` 这种类型，那么 self.2 有可能访问了被引用的变量
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 (self.2)(&self.0), self.1);
    }
}
```

下面是一个通过 trait 方法调用的例子：

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        // 这里可能隐藏了一个对于 `<T as Display>::fmt` 的调用，
        // 如果 `T` 是 `&'a _` 这种类型，就可能访问了借用的变量
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 self.0, self.1);
    }
}
```

当然，所有这些访问都可以进一步隐藏在由析构器调用的一些其他方法中，而不是直接写在析构器中。

在上述所有在析构器中访问`&'a u8`的情况下，添加`#[may_dangle]`属性使得该类型容易被误用，而借用检查器不会发现，从而导致问题。所以最好不要添加这个属性。

关于丢弃顺序的附带说明

虽然结构内部字段的删除顺序是被定义的，但对它的依赖是脆弱而微妙的。当顺序很重要时，最好使用`ManuallyDrop`包装器。

这就是关于丢弃检查器的全部内容吗?

事实证明，在编写不安全的代码时，我们通常根本不需要担心为丢弃检查器做正确的事情。然而，有一种特殊情况是需要担心的，我们将在下一节看一下。

幽灵数据

在处理不安全代码时，我们经常会遇到这样的情况：类型或生命周期在逻辑上与结构相关，但实际上并不是字段的一部分。这种情况最常发生在生命周期上。例如，`&'a [T]`的`Iter`（大约）定义如下：

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

但是由于`'a`在结构体中是未使用的，所以它是*无约束*的。由于这在历史上造成的麻烦，在结构定义中，不受约束的生命周期和类型是*禁止*的，因此我们必须在主体中以某种方式引用这些类型，正确地做到这一点对于正确的变异性和丢弃检查是必要的。

我们使用`PhantomData`来做这个，它是一个特殊的标记类型。`PhantomData`不消耗空间，但为了静态分析的目的，模拟了一个给定类型的字段。这被认为比明确告诉类型系统你想要的变量类型更不容易出错，同时也提供了其他有用的东西，例如 auto traits 和 drop check 需要的信息。

Iter 逻辑上包含一堆`&'a T`，所以这正是我们告诉`PhantomData`要模拟的。

```
use std::marker;

struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

就是这样，生命周期将被限定，而你的迭代器将在`'a`和`T`上进行协变。所有的东西都是有效的。

泛型参数和 drop 检查

在过去，曾经有另一个事情是需要仔细思考的，这篇文档曾经这么说：

> 另一个重要的例子是 Vec，它（大约）定义如下：
>
> ```
> struct Vec<T> {
>     data: *const T, // `*const`是可变异的!
>     len: usize,
>     cap: usize,
> }
> ```
>
> 与前面的例子不同的是，*看起来*一切都和我们想的一样。Vec 的每个通用参数至少在一个字段中出现。很好，可以开始了!

不对，不是这样。

丢弃检查器将慷慨地确定 `Vec<T>` 不拥有任何 T 类型的值。这将反过来使它得出结论，它不需要担心 Vec 在其析构器中丢弃任何 T 来确定丢弃检查的合理性。这将反过来允许人们使用 Vec 的析构器来制造不健壮性。

为了告诉 dropck 我们确实拥有 T 类型的值，因此在*我们*丢弃时可能会丢弃一些 T，我们必须添加一个额外的 `PhantomData`，正如这样：

```
use std::marker;

struct Vec<T> {
    data: *const T, // `*const`是可变异的!
    len: usize,
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

但自从RFC 1238之后，这就不正确也并不需要了。

如果你这么写：

```
struct Vec<T> {
    data: *const T, // `*const`是可变异的!
    len: usize,
    cap: usize,
}
impl<T> Drop for Vec<T> { /* … */ }
```

那么 `impl<T> Drop for Vec<T>` 这条语句会让 Rust 知道 `Vec<T>` *拥有* `T` 类型的值（更准确地说：可能会在 `Drop` 实现中使用 `T` 类型的值），那么当 `Vec<T>` 被 drop 的时候，Rust 就不会允许它们*悬垂*。

添加一个额外的 `_owns_T: PhantomData<T>` 字段因此是*多余的*并且什么都不做。

但是这在某些场景下，会导致过于严格，这也是为啥标准库使用了一个不稳定并且 `unsafe` 的属性来切换回旧的 `unchecked` 的 drop 检查行为，也是接下来这个文档所警告的：`#[may_dangle]` 属性。

一个例外：标准库的特殊情况及不稳定的`#[may_dangle]`

如果你只是写自己的库代码，那你可以跳过这章；但是如果你想知道标准库中真正的 `Vec` 是怎么实现的，你会发现它仍然需要 `_owns_T: PhantomData<T>` 字段来保证可靠性。

▶ 点这里查看原因

拥有内存分配的原始指针是如此普遍的模式，以至于标准库为自己整了一个名为`Unique<T>`的类型：

- 包装一个`*const T`，用于变异
- 包括一个`PhantomData<T>`
- 根据包含的 T 自动派生`Send`/`Sync`
- 空指针的优化，将指针标记为`NonZero`

`PhantomData`模式表

下面是一个关于所有可以使用`PhantomData`的神奇方式的表格：（covariant:协变，invariant:不变，contravariant:逆变)

Phantom type	`'a`	`T`	`Send`	`Sync`
`PhantomData<T>`	-	covariant (with drop check)	`T: Send`	`T: Sync`
`PhantomData<&'a T>`	covariant	covariant	`T: Sync`	`T: Sync`
`PhantomData<&'a mut T>`	covariant	invariant	`T: Send`	`T: Sync`
`PhantomData<*const T>`	-	covariant	-	-
`PhantomData<*mut T>`	-	invariant	-	-
`PhantomData<fn(T)>`	-	contravariant	`Send`	`Sync`
`PhantomData<fn() -> T>`	-	covariant	`Send`	`Sync`
`PhantomData<fn(T) -> T>`	-	invariant	`Send`	`Sync`
`PhantomData<Cell<&'a ()>>`	invariant	-	`Send`	-

拆分 Borrows

在处理复合结构时，可变引用的互斥属性会有很大的限制。借用检查器理解一些基本的东西，但是很容易就会出现问题。它对结构有足够的了解，知道有可能同时借用一个结构中不相干的字段。所以现在这个方法是可行的：

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

let mut x = Foo {a: 0, b: 0, c: 0};
let a = &mut x.a;
let b = &mut x.b;
let c = &x.c;
*b += 1;
let c2 = &x.c;
*a += 10;
println!("{} {} {} {}", a, b, c, c2);
```

然而 borrowck 完全不理解数组或 slice，所以这会挂：

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

```
error[E0499]: cannot borrow `x[..]` as mutable more than once at a time
 --> src/lib.rs:4:18
  |
3 |     let a = &mut x[0];
  |                 ---- first mutable borrow occurs here
4 |     let b = &mut x[1];
  |                 ^^^^ second mutable borrow occurs here
5 |     println!("{} {}", a, b);
6 | }
  | - first borrow ends here

error: aborting due to previous error
```

虽然 borrowck 能理解这个简单的案例是合理的，但对于 borrowck 来说，要理解像树这样的一般容器类型的不连通性显然是没有希望的，尤其是当不同的键*确实*映射到相同的值时。

为了“教导” borrowck 我们正在做的事情是正确的，我们需要使用到不安全的代码。例如，可变 slice 暴露了一个 `split_at_mut` 函数，它消耗这个 slice 并返回两个可变 slice。一个用于索引左边的所有内容，一个用于右边的所有内容。直观地讲，我们知道这是安全的，因为这些分片不会重叠，因此可以进行别名操作。然而，这个实现需要一些不安全代码：

```
pub fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();

    unsafe {
        assert!(mid <= len);

        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.add(mid), len - mid))
    }
}
```

这实际上是有点微妙的。为了避免对同一个值进行两次 `&mut`，我们明确地通过原始指针构造全新的切片。

然而，更微妙的是产生可变引用的迭代器如何工作。迭代器 trait 定义如下：

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

考虑到这个定义，Self::Item 与 `self` *没有*联系。这意味着我们可以连续多次调用 `next`，并将所有的结果*并发地*保留下来。这对逐值迭代器来说是非常好的，因为它有这样的语义。这对共享引用来说也很好，因为它们允许对同一事物有任意多的引用（尽管迭代器需要和被共享的事物是一个独立的对象）。

但是可变的引用让这变得一团糟。乍一看，它们似乎与这个 API 完全不兼容，因为它将产生对同一个对象的多个可变引用！

然而它实际上*是*有效的，正是因为迭代器是一次性的对象。IterMut 产生的所有东西最多只能产生一次，所以我们实际上不会产生对同一块数据的多个可变引用。

也许令人惊讶的是，对于许多类型，可变迭代器不需要实现不安全的代码。

例如，这里有一个单向链表：

```
type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

pub struct LinkedList<T> {
    head: Link<T>,
}

pub struct IterMut<'a, T: 'a>(Option<&'a mut Node<T>>);

impl<T> LinkedList<T> {
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut(self.head.as_mut().map(|node| &mut **node))
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node| {
            self.0 = node.next.as_mut().map(|node| &mut **node);
            &mut node.elem
        })
    }
}
```

下面是一个可变的 slice：

```
use std::mem;

pub struct IterMut<'a, T: 'a>(&'a mut[T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        let slice = mem::take(&mut self.0);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        let slice = mem::take(&mut self.0);
        if slice.is_empty() { return None; }

        let new_len = slice.len() - 1;
        let (l, r) = slice.split_at_mut(new_len);
        self.0 = l;
        r.get_mut(0)
    }
}
```

接着是一个二叉树：

```rust
use std::collections::VecDeque;

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    left: Link<T>,
    right: Link<T>,
}

pub struct Tree<T> {
    root: Link<T>,
}

struct NodeIterMut<'a, T: 'a> {
    elem: Option<&'a mut T>,
    left: Option<&'a mut Node<T>>,
    right: Option<&'a mut Node<T>>,
}

enum State<'a, T: 'a> {
    Elem(&'a mut T),
    Node(&'a mut Node<T>),
}

pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);

impl<T> Tree<T> {
    pub fn iter_mut(&mut self) -> IterMut<T> {
        let mut deque = VecDeque::new();
        self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
        IterMut(deque)
    }
}

impl<T> Node<T> {
    pub fn iter_mut(&mut self) -> NodeIterMut<T> {
        NodeIterMut {
            elem: Some(&mut self.elem),
            left: self.left.as_mut().map(|node| &mut **node),
            right: self.right.as_mut().map(|node| &mut **node),
        }
    }
}


impl<'a, T> Iterator for NodeIterMut<'a, T> {
    type Item = State<'a, T>;

    fn next(&mut self) -> Option<Self::Item> {
        match self.left.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.right.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
```

```
                }
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        match self.right.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.left.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.front_mut().and_then(|node_it| node_it.next()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
                None => if let None = self.0.pop_front() { return None },
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
                None => if let None = self.0.pop_back() { return None },
            }
        }
    }
}
```

所有这些都是完全安全的，并且可以在稳定的 Rust 上运行！这最终落在了我们之前看到的简单结构案例中。Rust 知道你可以安全地将一个可变的引用分割成子字段。然后我们可以通过 Options（或者在分片的情况下，用空分片替换）来消耗掉这个引用并进行编码。

类型转换

说到底，一切都只是某处的一堆比特，而类型系统只是为了帮助我们正确使用这些比特。类型系统中有两个常见的问题：需要将这些确切的位重新解释为不同的类型，以及需要改变位以对不同的类型具有同等的意义。因为 Rust 鼓励在类型系统中对重要的属性进行编码，所以这些问题是非常普遍的。因此，Rust 给了你几种方法来解决它们。

首先，我们将看看 Safe Rust 给你提供的重新解释值的方法。最简单的方法是把一个值分解成它的组成部分，然后从它们中建立一个新的类型：

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

但这最好也不过是一种烦人的做法。对于常见的转换，Rust 提供了更符合人体工程学的替代方法。

强转

在某些情况下，类型可以隐式地被强转。这些变化通常只是*削弱*类型，主要集中在指针和生命周期方面。它们的存在主要是为了让 Rust 在更多的情况下“正常工作”，而且基本上是无害的。

关于所有强转类型的详尽列表，请参见《The Reference》中的Coercion types部分。

请注意，在匹配 Trait 时，我们不进行强制转换（除了接收者，见下一页）。如果某个类型`U`有一个`impl`，而`T`可以强转到`U`，这并不构成`T`的实现。例如，下面的内容不会通过类型检查，尽管将`t`强转到`&T`是可以的，并且有针对`&T`的`impl`。

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}

fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

这样编译失败：

```
error[E0277]: the trait bound `&mut i32: Trait` is not satisfied
 --> src/main.rs:9:9
  |
3 | fn foo<X: Trait>(t: X) {}
  |           ----- required by this bound in `foo`
...
9 |     foo(t);
  |         ^ the trait `Trait` is not implemented for `&mut i32`
  |
  = help: the following implementations were found:
            <&'a i32 as Trait>
  = note: `Trait` is implemented for `&i32`, but not for `&mut i32`
```

点运算符

点运算符将执行很多类型转换的魔法：它将执行自动引用、自动去引用和强制转换，直到类型匹配。方法查找的详细机制定义在这里，简要的概述如下：

假设我们有一个函数 `foo`，它有一个接收器（一个 `self`、`&self` 或 `&mut self` 参数）。如果我们调用 `value.foo()`，编译器需要确定 `Self` 是什么类型，然后才能调用该函数的正确实现。在这个例子中，我们将说 `value` 具有 `T` 类型。

我们将使用full-qualified syntax来更清楚地说明我们到底是在哪个类型上调用一个函数。

- 首先，编译器会检查是否可以直接调用 `T::foo(value)`。这被称为“按值”方法调用。
- 如果它不能调用这个函数（例如，如果这个函数的类型不对，或者一个 trait 没有为 `Self` 实现），那么编译器就会尝试添加一个自动引用。这意味着编译器会尝试 `<&T>::foo(value)` 和 `<&mut T>::foo(value)`。这被称为“autoref”方法调用。
- 如果这些候选方法都不奏效，它就对 `T` 解引用并再次尝试。这使用了 `Deref` 特性——如果 `T: Deref<Target = U>`，那么它就用 `U` 而不是 `T` 类型再试。如果它不能解除对 `T` 的引用，它也可以尝试 *unsizing* `T`。这只是意味着，如果 `T` 在编译时有一个已知的大小参数，那么在解析方法时它就会“忘记”它。例如，这个 unsizing 步骤可以通过“忘记”数组的大小将 `[i32; 2]` 转换成 `[i32]`。

下面是一个方法查找算法的例子：

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

当数组在这么多的间接点后面时，编译器是如何实际计算 `array[0]` 的呢？首先，`array[0]` 实际上只是 `Index` 特性的语法糖——编译器会将 `array[0]` 转换成 `array.index(0)`。现在，编译器检查 `array` 是否实现了 `Index`，这样它就可以调用这个函数。

然后，编译器检查 `Rc<Box<[T; 3]>>` 是否实现了 `Index`，但它没有，`&Rc<Box<[T; 3]>>` 和 `&mut Rc<Box<[T; 3]>>` 也没有。由于这些方法都不起作用，编译器将 `Rc<Box<[T; 3]>` 解引用到 `Box<[T; 3]>` 中，并再次尝试。`Box<[T; 3]>`、`&Box<[T; 3]>` 和 `&mut Box<[T; 3]>` 没有实现 `Index`，所以它再次解引用。`[T; 3]` 和它的自动引用也没有实现 `Index`。它不能再继续解引用 `[T; 3]`，所以编译器取消了它的大小，得到了 `[T]`。最后，`[T]` 实现了 `Index`，所以它现在可以调用实际的 `index` 函数。

考虑一下下面这个更复杂的点运算符工作的例子：

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

实现了 `Clone` 的是什么类型？首先，编译器检查是否可以按值调用。`value` 的类型是 `&T`，所以 `clone` 函数的签名是 `fn clone(&T) -> T`。它知道 `T: Clone`，所以编译器发现 `cloned: T`。

如果取消 `T: Clone` 的限制，会发生什么？它将不能按值调用，因为 `T` 没有实现 `Clone`。所以编译器会尝试通过自动搜索来调用。在这种情况下，该函数的签名是 `fn clone(&&T) -> &T`，因为 `Self = &T`。编译器看到 `&T: Clone`，然后推断出 `cloned: &T`。

下面是另一个例子，自动搜索行为被用来创造一些微妙的效果：

```
#[derive(Clone)]
struct Container<T>(Arc<T>);

fn clone_containers<T>(foo: &Container<i32>, bar: &Container<T>) {
    let foo_cloned = foo.clone();
    let bar_cloned = bar.clone();
}
```

`foo_cloned` 和 `bar_cloned` 是什么类型？我们知道，`Container<i32>: Clone`，所以编译器按值调用 `clone`，得到 `foo_cloned: Container<i32>`。然而，`bar_cloned` 实际上有 `&Container<T>` 类型。这肯定是不合理的——我们给 `Container` 添加了 `#[derive(Clone)]`，所以它必须实现 `Clone`！仔细看看，由 `derive` 宏产生的代码是（大致）：

```
impl<T> Clone for Container<T> where T: Clone {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}
```

派生的 `Clone` 实现是只在 `T: Clone` 的地方定义，所以没有 `Container<T>` 的实现。`Clone` 在一般的 `T` 上没有实现。编译器接着查看 `&Container<T>` 是否实现了 `Clone`，最终发现它实现了。因此，它推断出 `clone` 是由 autoref 调用的，所以 `bar_cloned` 的类型是 `&Container<T>`。

我们可以通过手动实现 `Clone` 而不需要 `T: Clone` 来解决这个问题：

```
impl<T> Clone for Container<T> {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}
```

现在，类型检查器推断出，`bar_cloned: Container<T>`。

Casts

Casts（译者注：实在没有找到合适的中文表述）是强转的超集：每个强转都可以通过 cast 来明确调用。然而，有些转换需要 cast。虽然强转是普遍存在的，而且基本上是无害的，但是这些“真正的 cast”是罕见的，而且有潜在的危险。因此，必须使用`as`关键字来明确调用 cast：`expr as Type`。

你可以在《The Reference》中找到一个所有真正的 cast 和 cast 语义的详尽列表。

Casting 的安全性

真正的 cast 通常围绕着原始指针和原始数字类型。尽管它们很危险，但这些转换在运行时是不会出错的。如果一个 cast 触发了一些微妙的边界条件，也不会有任何迹象表明发生了这种情况，cast 会成功。也就是说，cast 必须在类型的级别上有效，否则会在编译时被静态地阻止。例如，`7u8 as bool`编译会出错。

也就是说， cast 并不是`unsafe`的，因为它们*本身*通常不会违反内存安全。例如，将一个整数转换为一个原始指针很容易导致可怕的事情，然而，创建指针的行为本身是安全的，因为实际使用一个原始指针已经被标记为`unsafe`。

一些关于 cast 的说明

cast raw slice 时的长度问题

请注意，在 cast raw slice 时，长度不会被调整：`*const [u16] as *const [u8]`创建的 slice 只包括原始内存的一半。

传递性

Casting 不是传递的，也就是说，即使`e as U1 as U2`是一个有效的表达式，`e as U2`也不一定是。

Transmutes

类型系统，别挡着我们的路！我们要重新解释这些比特，否则就会死掉！尽管这本书是关于做不安全的事情的，但我真的必须强调，你应该深入思考找到本节中所涉及的操作以外的另一种方法。这真的是你在 Rust 中所能做的最可怕的不安全的事情，而这基本不设防。

`mem::transmute<T, U>` 接收一个 `T` 类型的值并将其重新解释为 `U` 类型。唯一的限制是 `T` 和 `U` 被验证为具有相同的大小。导致未定义行为的方法是令人难以置信的。

- 首先，创建一个具有无效状态的*任何*类型的实例都会导致无法真正预测的任意混乱。即使你从未对 `bool` 做过任何事情，也不要把 `3` 转化为 `bool`。就是不要。
- Transmute 有一个重载的返回类型。如果你不指定返回类型，它可能会为了满足类型推导而返回一个令人惊讶的类型。
- 将一个 `&` 转为 `&mut` 是未定义行为，尽管某些用法*可能*是安全的，但是需要注意，Rust 优化器可以自由地假设一个共享引用在它的生命周期内是不变的，而这种转换会违反这个假设。因此：
    - 将一个 `&` 转为 `&mut` *总是*未定义行为
    - 不，你不能这样做
    - 不，你并不特别
- Transmute 到一个没有明确提供生命周期的引用会产生一个无限制的寿命
- 当在不同的复合类型之间转换时，你必须确保它们的布局是一样的！如果布局不同，错误的字段就会被填入错误的数据，这也许仅仅让你 Debug 一阵，也可能会造成 UB（见上文）

  那么你怎么知道布局是否相同呢？对于 `repr(C)` 类型和 `repr(transparent)` 类型，布局是精确定义的。但是对于普通的 `repr(Rust)` 来说，它不是。即使是同一个通用类型的不同实例也可以有截然不同的布局。`Vec<i32>` 和 `Vec<u32>` *可能*有相同的字段顺序，也可能没有。数据布局保证了什么，或者没保证什么的细节可以参考 UCG 工作组。

`mem::transmute_copy<T, U>` 比这个更不安全。它把 `size_of<U>` 字节从 `T` 中复制出来，并把它们解释为 `U`。`mem::transmute` 的大小检查没有了（因为复制出一个前缀可能是有效的），尽管 `U` 比 `T` 大是未定义行为。

当然，你也可以使用原始指针转换或 `union` 来获得这些函数的所有功能，并关闭 Lint 或其他基本的合理性检查。原始指针转换和 `union` 并不能神奇地避免上述规则。

使用未初始化的内存

Rust 程序中所有运行时分配的内存在开始时都是*未初始化*的。在这种状态下，内存的值是一堆不确定的比特，什么都有可能。试图将这个内存解释为*任何*类型的值都将导致未定义行为。请不要这样做。

Rust 提供了一些机制，以检查（安全）和不检查（不安全）的方式处理未初始化的内存。

经检查的未初始化的内存

和 C 语言一样，Rust 中的所有堆栈变量都是未初始化的，直到为它们明确赋值。与 C 不同的是，Rust 静态地阻止你读取它们，直到你为它们赋值。

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

```
  |
3 |     println!("{}", x);
  |                     ^ 使用了没有初始化的 `x`
```

这基于一个基本的分支分析：每个分支都必须在第一次使用`x`之前给它赋值，方便起见，我们会说“x 被初始化了”或者“x 未初始化”。有趣的是，如果每个分支恰好赋值一次，Rust 不要求变量是可变的，以执行延迟初始化。然而这个分析并没有利用常量分析或类似的东西。所以下述的代码是可以编译的：

```
fn main() {
    let x: i32;

    if true {
        x = 1;
    } else {
        x = 2;
    }

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

但这个不行：

```
fn main() {
    let x: i32;
    if true {
        x = 1;
    }
    println!("{}", x);
}
```

```
  |
6 |     println!("{}", x);
  |                     ^ 使用了可能没有初始化的 `x`
```

这个又可以了：

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!("{}", x);
    }
    // 不需要担心还有没有初始化 x 的分支，
    // 因为我们实际上并没有在别的分支使用 x
}
```

当然，虽然分析不考虑实际值，但它对依赖关系和控制流有相对复杂的理解。例如，这样是可以编译通过的：

```
let x: i32;

loop {
    // Rust 不知道这个分支会被无条件执行，
    // 因为它依赖于实际值
    if true {
        // 但是它确实知道循环只会有一次，
        // 因为我们会无条件 break，
        // 所以 x 不需要是可变的
        x = 0;
        break;
    }
}
// Rust 知道如果没有执行 break 的话，代码不会运行到这里
// 所以一旦运行到这里，x 一定已经初始化了
println!("{}", x);
```

如果一个值从一个变量中移出，并且该值的类型不是 Copy，该变量在逻辑上就会变成未初始化。也就是说：

```
fn main() {
    let x = 0;
    let y = Box::new(0);
    let z1 = x; // x 仍然是有效的，因为 i32 可以 Copy
    let z2 = y; // 现在 y 逻辑上未初始化，因为 Box 不能 Copy
}
```

然而，在这个例子中重新给`y`赋值需要将`y`标记为可变，这样一个安全的 Rust 程序就可以观察到`y`的值发生了变化：

```
fn main() {
    let mut y = Box::new(0);
    let z = y; // 现在 y 逻辑上未初始化，因为 Box 不能 Copy
    y = Box::new(1); // 重新初始化 y
}
```

否则`y`就像是一个全新的变量。

丢弃标志

上一节的例子为 Rust 引入了一个有趣的问题。我们已经看到，可以完全安全地对内存位置进行有条件的初始化、非初始化和重新初始化。对于实现了`Copy`的类型来说，这并不特别值得注意，因为它们只是一堆随机的比特。然而，带有析构器的类型是一个不同的故事。Rust 需要知道每当一个变量被赋值，或者一个变量超出范围时，是否要调用一个析构器。它怎么能用条件初始化来做到这一点呢?

请注意，这不是所有赋值都需要担心的问题。特别是，通过解引用的赋值会无条件地被丢弃，而相对的，在`let`中的赋值无论如何都不会被丢弃：

```
let mut x = Box::new(0); // let 创建了一个全新的变量，所以一定(也没有必要)调用 drop
let y = &mut x;
*y = Box::new(1); // 解引用假设原先的变量已经初始化了，因此一定会 drop
```

仅当覆盖先前初始化的变量或其子字段之一时，这才是个问题。

这种情况下，Rust 实际上是在*运行时*跟踪一个类型是否应该被丢弃。当一个变量被初始化和未初始化时，该变量的*丢弃标志*被切换。当一个变量可能需要被丢弃时，这个标志会被读取，以确定它是否应该被丢弃。

当然，通常的情况是，一个值的初始化状态在程序的每一个点上都是静态已知的。如果是这种情况，那么编译器理论上可以生成更有效的代码。例如，直线型代码就有这样的*静态丢弃语义（static drop semantics）*：

```
let mut x = Box::new(0); // x 未初始化；仅覆盖值
let mut y = x;           // y 未初始化；仅覆盖值，并设置 x 为未初始化
x = Box::new(0);         // x 未初始化；仅覆盖值
y = x;                   // y 已初始化；销毁 y，覆盖它的值，设置 x 为未初始化
                         // y 离开作用域；y 已初始化；销毁 y
                         // x 离开作用域；x 未初始化；什么都不用做
```

类似地，所有分支都在初始化方面具有相同行为的代码具有静态丢弃语义：

```
let mut x = Box::new(0); // x 未初始化；仅覆盖值
if condition {
    drop(x);             // x 失去值；设置 x 为未初始化
} else {
    println!("{}", x);
    drop(x);             // x 失去值；设置 x 为未初始化
}
x = Box::new(0);         // x 未初始化；仅覆盖值
                         // x 离开作用域；x 已初始化；销毁 x
```

然而像这样的代码*需要*运行时的信息来正确地 Drop：

```
let x;
if condition {
    x = Box::new(0);        // x 未初始化；仅覆盖值
    println!("{}", x);
}
                            // x 离开了作用域，可能未初始化
                            // 检查 drop 标志位！
```

当然，在这种情况下，获得静态丢弃语义是很简单的：

```
if condition {
    let x = Box::new(0);
    println!("{}", x);
}
```

丢弃标志在栈中被跟踪。 在旧的 Rust 版本中，丢弃标志曾经是隐藏在实现`Drop`的类型中。

未经检查的未初始化的内存

这个规则的一个有趣的例外是与数组一起工作。Safe Rust 不允许你部分初始化一个数组。当你初始化一个数组时，你可以用`let x = [val; N]`将每个值设置为相同的东西，或者你可以用`let x = [val1, val2, val3]`单独指定每个成员。不幸的是，这是很死板的，特别是当你需要以更多的增量或动态方式初始化你的数组时。

不安全的 Rust 给了我们一个强大的工具来处理这个问题：`MaybeUninit`。这个类型可以用来处理还没有完全初始化的内存。

使用`MaybeUninit`，我们可以对一个数组进行逐个元素的初始化，如下所示：

```
use std::mem::{self, MaybeUninit};

// 数组的大小是硬编码的，可以很方便地修改（改变几个硬编码的常数非常容易）
// 这表示我们不能用 [a, b, c] 这种方式初始化数组，因为我们必须要和硬编码中的 `SIZE` 保持同步！
const SIZE: usize = 10;

let x = {
    // 创建一个未初始化，类型为 `MaybeUninit` 的数组，
    // 因为这里声明的是一堆 `MaybeUninit`，不要求初始化，所以 `assume_init` 操作是安全的
    let mut x: [MaybeUninit<Box<u32>>; SIZE] = unsafe {
        MaybeUninit::uninit().assume_init()
    };

    // 因为 drop 一个 `MaybeUninit` 什么都不做，
    // 所以使用直接的裸指针赋值（而非 ptr::write）不会导致原先未初始化的变量被 drop
    // 不需要在这里考虑异常安全，因为 Box 永远不会 panic
    for i in 0..SIZE {
        x[i] = MaybeUninit::new(Box::new(i as u32));
    }

    // 一切都初始化完毕，将未初始化的类型强制转换为初始化的类型
    unsafe { mem::transmute::<_, [Box<u32>; SIZE]>(x) }
};

dbg!(x);
```

这段代码分三步进行：

1. 创建一个`MaybeUninit<T>`的数组。在当前稳定版的 Rust 中，我们必须使用不安全的代码来实现：我们取一些未初始化的内存（`MaybeUninit::uninit()`），并声称我们已经完全初始化了它（`assume_init()`）。这似乎很荒谬，因为我们没有！这是正确的，因为数组本身完全由`MaybeUninit`组成，实际上不需要初始化。对于大多数其他类型，`MaybeUninit::uninit().assume_init()`会产生一个无效的类型实例，所以你荣获了一些未定义行为。

2. 初始化数组。这个问题的微妙之处在于，通常情况下，当我们使用 `=` 赋值给一个 Rust 类型检查器认为已经初始化的值时（比如 `x[i]`），存储在左边的旧值会被丢掉。这将是一场灾难。然而，在这种情况下，左边的类型是 `MaybeUninit<Box<u32>>`，丢弃这个类型什么都不会发生，关于这个 `drop` 问题的更多讨论，见下文。

3. 最后，我们必须改变我们数组的类型，以去除 `MaybeUninit`。在当前稳定的 Rust 中，这需要一个 `transmute`。这种转换是合法的，因为在内存中，`MaybeUninit<T>` 看起来和 `T` 一样。

   然而，请注意，在一般情况下，`Container<MaybeUninit<T>>` 与 `Container<T>` 看起来*并不*一样! 假如 `Container` 是 `Option`，而 `T` 是 `bool`，那么 `Option<bool>` 就利用了 `bool` 只有两个有效值，但 `Option<MaybeUninit<bool>>` 不能这样做，因为 `bool` 不需要被初始化。

   所以，这取决于 `Container` 是否允许将 `MaybeUninit` 转化掉。对于数组来说，它是允许的（最终标准库会通过提供适当的方法来达到这一点）。

让我们在中间的循环上多花一点时间，特别是赋值运算符和它与 `drop` 的交互。比如这样的代码：

```
*x[i].as_mut_ptr() = Box::new(i as u32); // 错误!
```

我们实际上会覆盖一个 `Box<u32>`，导致在未初始化数据上调用 `drop`，这将给你带来很多乐子。

如果由于某种原因我们不能使用 `MaybeUninit::new`，正确的选择是使用 `ptr` 模块。特别是，它提供了三个函数，允许我们将字节分配到内存中的某个位置而不丢弃旧值。`write`、`copy` 和 `copy_nonoverlapping`。

- `ptr::write(ptr, val)` 接收一个 `val` 并将其移动到 `ptr` 所指向的地址
- `ptr::copy(src, dest, count)` 将 `count` 个 T 所占用的位从 src 复制到 dest(这等同于 C 的 memmove —— 注意参数顺序是相反的！)
- `ptr::copy_nonoverlapping(src, dest, count)` 做的是 `copy` 的工作，但是在假设两个内存范围不重叠的情况下，速度更快(这等同于 C 的 memcpy —— 注意参数顺序是相反的！)

自然不用说，这些函数如果被误用，会造成严重的破坏，或者直接导致未定义行为。这些函数*本身*需要的唯一东西是，你想读和写的位置已经被分配并正确对齐。然而，向内存的任意位置写入任意位的方式所带来的问题是无穷无尽的。

值得注意的是，你不需要担心在未实现 `Drop` 或者不包含 `Drop` 类型的类型上使用 `ptr::write` 带来的问题，因为 Rust 知道这个信息，并且不会调用 `drop`。这也是我们在上面的例子中所依赖的。

然而，当你处理未初始化的内存时，你需要时刻警惕 Rust 试图在它们完全初始化之前丢弃你创建的这些值。如果它有一个析构器的话，该变量作用域内的每个控制路径必须在结束前初始化该值。*这包括 panic*。`MaybeUninit` 在这方面有一点用，因为它不会隐式地丢弃它的内容——但在 panic 的情况下，这实际上意味着不是对尚未初始化的部分进行双重释放，而是对已经初始化的部分导致了内存泄漏。

注意，为了使用`ptr`方法，你需要首先获得一个你想初始化的数据的*raw pointer*。对未初始化的数据构建一个*引用*是非法的，这意味着你在获得上述原始指针时必须小心：

- 对于一个`T`的数组，你可以使用`base_ptr.add(idx)`，其中`base_ptr: *mut T`来计算数组索引`idx`的地址。这依赖于数组在内存中的布局方式
- 然而，对于一个结构体，一般来说，我们不知道它是如何布局的，而且我们也不能使用`&mut base_ptr.field`，因为这将创建一个引用。因此，当你使用`addr_of_mut`宏的时候，你必须非常小心，这将跳过中间层直接创建一个指向该字段的裸指针：

```
use std::{ptr, mem::MaybeUninit};
struct Demo {
    field: bool,
}
let mut uninit = MaybeUninit::<Demo>::uninit();
// `&uninit.as_mut().field`将会创建一个指向未初始化的`bool`的指针，而这是 UB 行为。
let f1_ptr = unsafe { ptr::addr_of_mut!((*uninit.as_mut_ptr()).field) };
unsafe { f1_ptr.write(true); }
let init = unsafe { uninit.assume_init() };
```

最后一句话：在阅读旧的 Rust 代码时，你可能会无意中发现被废弃的`mem::uninitialized`函数。这个函数曾经是处理栈上未初始化内存的唯一方法，但它被证明不能与语言的其他部分很好地结合在一起。在新的代码中你总是应该使用`MaybeUninit`来代替，并且当你有机会的时候，可以把旧的代码移植过来。

这就是与未初始化内存打交道的方法。基本上没有任何地方希望得到未初始化的内存，所以如果你要传递它，一定要*非常*小心。

基于所有权的资源管理（OBRM）的危险性

OBRM（又称 RAII：资源获取即初始化）是你在 Rust 中经常会用到的技巧，特别是当你使用标准库的时候。

粗略的说，其模式如下：要获得一个资源，你要创建一个对象来管理它。要释放资源，你只需销毁这个对象，它就会为你清理资源。这种模式管理的最常见的“资源”就是*内存*。`Box`、`Rc`以及`std::collection`中的所有东西都是一种便利，可以正确管理内存。这在 Rust 中特别重要，因为我们没有 GC 来管理内存。重点来了：Rust 是关于控制的。然而，我们并不仅仅局限于内存。几乎所有其他的系统资源，如线程、文件或套接字，都可以通过这种 API 暴露。

构造

构造一个用户定义类型的实例只有一种方法：为其命名，并一次性初始化其所有字段:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}

struct Unit;

let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

就这样。其他所有构造类型实例的方法都是在调用一个完全虚无的函数，这个函数做了一些事情，最后变成了唯一的真实构造函数。

与 C++ 不同，Rust 没有内置的各种构造函数。没有 Copy、Default、Assignment、Move 或其他构造函数。其原因是多方面的，但主要归结为 Rust 的*显式*哲学。

移动构造函数在 Rust 中是没有意义的，因为我们不允许类型“关心”它们在内存中的位置。每个类型都必须准备好被盲目地移动到内存中的其他地方。这意味着纯粹的栈上但仍可移动的侵入性链表在 Rust 中根本无法（安全地）实现。

赋值和复制构造函数也同样不存在，因为移动语义是 Rust 中唯一的语义。`x = y` 最多只是把 y 的位移到 x 变量中。Rust 确实提供了两种方法来提供 C++ 的面向拷贝的语义：`Copy` 和 `Clone`。Clone 类似我们所说的复制构造函数，但它从未被隐式调用。你必须在你想要克隆的元素上明确地调用 `clone`。Copy 是 Clone 的一个特例，它的实现只是“复制比特”。Copy 类型*是*隐式克隆的，只要它们被移动；但由于 Copy 的定义，这只是意味着不把旧的变量当作未初始化的 —— 也就是说，啥都没干（no-op）。

虽然 Rust 提供了一个 `Default` 特性来指定了一个类似默认构造函数的东西，但这个特性很少被使用。这是因为变量不是隐式初始化的。 Default 基本上只对泛型编程有用。在具体环境中，一个类型将为任何类型的“默认”构造函数提供一个静态的 `new` 方法。这与其他语言中的 `new` 没有关系，也没有特殊含义。它只是一个命名惯例。

TODO: talk about "placement new"?

析构

Rust 通过`Drop` trait 提供了完整的自动析构器，它提供了以下这个方法：

```
fn drop(&mut self);
```

这个方法给了类型一些时间来完成它正在做的事情。

在`drop`运行后，Rust 将递归地尝试删除`self`的所有字段。

这是一个方便的功能，这样你就不必写“析构器模板”来丢弃子字段。如果一个结构除了丢弃其子字段之外没有特殊的丢弃逻辑，那么就意味着根本不需要实现`Drop`！

在 Rust 1.0 中没有稳定的方法来阻止这种行为。

请注意，这里使用的是`&mut self`，意味着即使你想要阻止递归的 Drop（例如将字段移出 self），Rust 也会阻止你。对于大多数类型来说，这完全没有问题。

一个自定义的`Box`的实现可以这样写`Drop`：

```
#![feature(ptr_internals, allocator_api)]

use std::alloc::{Allocator, Global, GlobalAlloc, Layout};
use std::mem;
use std::ptr::{drop_in_place, NonNull, Unique};

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>())
        }
    }
}
```

这样做是可行的，因为当 Rust 去丢弃`ptr`字段时，它只是看到一个Unique，没有实际的`Drop`实现。同样的，没有任何东西可以在释放后使用`ptr`，因为当 drop 退出时，它就变得不可访问了。

然而下面这段代码就不可行了：

```
#![feature(allocator_api, ptr_internals)]

use std::alloc::{Allocator, Global, GlobalAlloc, Layout};
use std::ptr::{drop_in_place, Unique, NonNull};
use std::mem;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Box<T> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // 释放 box 的内容，而不是 drop box 的内容
            let c: NonNull<T> = self.my_box.ptr.into();
            Global.deallocate(c.cast::<u8>(), Layout::new::<T>());
        }
    }
}
```

当我们在 SuperBox 的析构器中释放完 `box` 的 ptr 后，Rust 会很高兴地告诉 box 去 Drop 自己，然后，你就能开开心心去 debug use-after-free 和 double-free 的问题了。

请注意，递归 drop 行为适用于所有结构和枚举，无论它们是否实现了 Drop。因此，像这样的代码：

```
struct Boxy<T> {
    data1: Box<T>,
    data2: Box<T>,
    info: u32,
}
```

在它将被丢弃时，它的 data1 和 data2 的字段就会被析构，尽管它本身并没有实现 Drop。我们说这样的类型*需要 Drop*，尽管它本身不是 Drop。

类似地：

```
enum Link {
    Next(Box<Link>),
    None,
}
```

当且仅当一个实例存储了 Next 变量时，它的内部 Box 字段将被丢弃。

一般来说，这种设计非常好，因为当你重构数据布局时，你不需要担心添加/删除`Drop`的问题。当然，也有很多需要用析构器做更棘手的事情的例子。

经典的覆盖递归 drop 行为并允许在`drop`过程中移出 Self 的安全的解决方案是，使用一个 Option：

```
#![feature(allocator_api, ptr_internals)]

use std::alloc::{Allocator, GlobalAlloc, Global, Layout};
use std::ptr::{drop_in_place, Unique, NonNull};
use std::mem;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Option<Box<T>> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // 释放 box 的内容，而不是 drop box 的内容，
            // 需要将 box 字段设置为 None，防止 Rust 对 box 成员可能存在的drop操作
            let my_box = self.my_box.take().unwrap();
            let c: NonNull<T> = my_box.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
            mem::forget(my_box);
        }
    }
}
```

然而这有相当奇怪的语义：你是说一个*应该*总是 Some 的字段*可能*是 None，只是因为这发生在析构器中。当然，这也有一定的意义：你可以在析构器中调用 self 上的任意方法，这应该可以防止你在释放字段后这样做；而并不是说它能阻止你产生无效的状态。

总的来说，这是个可以接受的选择。当然，你应该在默认情况下达到这样的效果。然而，在未来，我们希望有一种更好的方式来指明一个字段不应该被自动 drop 掉。

泄漏

基于所有权的资源管理是为了简化组合：你在创建对象时获得资源，在对象被销毁时释放资源。由于销毁是自动为你处理的，这意味着你不能忘记释放资源，而且会尽快地释放！当然这很完美，我们所有的问题都解决了.....么？

一切都很糟糕，我们有新的、奇特的问题需要去解决。

很多人相信 Rust 能防止资源泄漏。在实践中，这基本上是对的。如果你看到一个安全的 Rust 程序以不受控制的方式泄漏资源，你会感到惊讶。

然而从理论的角度来看，无论你怎么看，都绝对不是这样的。在最严格的意义上，“泄漏”是如此抽象，以至于无法预防。在程序开始时初始化一个集合，用大量带有析构器的对象填充它，然后进入一个从未引用过它的无限事件循环，这是非常容易的。这个集合将毫无用处地坐着，守着它宝贵的资源，直到程序终止（无论如何，这时所有这些资源都会被操作系统回收）。

我们可以考虑一种更有限的泄漏形式：未能丢弃一个无法到达的值。Rust 也没有防止这种情况。事实上，Rust *有一个函数可以做到这一点*。`mem::forget`。这个函数消耗它所传递的值，*然后不运行它的析构器*。

在过去，`mem::forget`被标记为不安全，作为对使用它的一种提示，因为不调用一个析构器通常不是一件好的事情（尽管对一些特殊的不安全代码很有用）。然而，这通常被认为是一种站不住脚的立场：在安全代码中，有很多方法可以不调用析构函数。最著名的例子是使用内部可变性创建一个引用计数指针的循环引用。

对于安全代码来说，假设析构器的泄漏不会发生是合理的，因为任何泄漏析构器的程序都可能是错误的。然而，*不安全的*代码不能依赖析构器的运行来保证安全。对于大多数类型来说，这并不重要：如果你泄露了析构函数，那么根据定义，该类型是不可访问的，所以这并不重要，对吗？例如，如果你泄露了一个`Box<u8>`，那么你会浪费一些内存，但这几乎不会违反内存安全。

然而，我们必须注意的是*代理*类型的解构器泄露。这些类型管理对一个独立对象的访问，但实际上并不拥有它。代理对象是相当罕见的，你需要关注的代理对象就更少了。我们将专注于标准库中三个有趣的例子：

- `vec::Drain`
- `Rc`
- `thread::scoped::JoinGuard`

Drain

`drain`是一个 collections API，它将数据从容器中移出而不消耗容器。这使我们能够在对一个`Vec`的所有内容都获得所有权后重新使用其底层的内存分配。它产生了一个迭代器（Drain），并按值返回 Vec 的内容。

现在，考虑一下迭代中的 Drain：一些值已经被移出，而另一些还没有。这意味着 Vec 的一部分现在充满了逻辑上未初始化的数据! 我们可以在每次移出一个值的时候对 Vec 中的所有元素进行后移，但这将会产生非常灾难性的性能后果。

相反，我们希望 Drain 能在 Vec 被删除时修复它底层需要的内存分配（译者注：也就是 Vec 的内存分配）。它应该自己运行直到完成，并回移任何没有被移除的元素（drain 支持子范围），然后修复 Vec 的`len`。它甚至是 unwind 安全的。很简单!

现在考虑下面的情况：

```
let mut vec = vec![Box::new(0); 4];

{
    // 开始 drain，vec 无法被再次访问
    let mut drainer = vec.drain(..);

    // 从 drain 中取出两个元素，然后立刻销毁它们
    drainer.next();
    drainer.next();

    // 销毁 drainer，但是不调用它的 drop 函数
    mem::forget(drainer);
}

// Oops，vec[0] 已经被 drop 了，我们正在读一块已经释放的内存
println!("{}", vec[0]);
```

这很明显不是好事。不幸的是，我们正处于两难境地：在每一步保持一致的状态有巨大的成本（并且会抵消 API 带来的任何好处）。如果不能保持一致的状态，我们就会在安全代码中出现未定义的行为（使 API 不健全）。

那么我们能做什么呢？好吧，我们可以选择一个微弱的一致性状态：当我们开始迭代时，将 Vec 的 len 设置为 0，并在必要时在析构器中修复它。这样一来，如果一切执行正常，我们就能以最小的开销获得所需的行为。但是如果有人*胆敢*在迭代过程中 forget 了我们，那大不了就是*泄露更多*（并且可能让 Vec 处于一个虽然意外的但其他方面保持一致的状态）。既然我们已经接受了 mem::forget 是安全的，那么这就必须绝对是安全的。我们把一个泄漏导致更多的泄漏称为*泄漏放大*。

Rc

Rc 是一个有趣的例子，因为乍一看，它似乎根本就不是一个代理值。毕竟，它管理着它所指向的数据，丢掉一个值的所有 Rcs 就会丢掉这个值。泄露一个 Rc 似乎并不特别危险。它将使 refcount 永久增加，并阻止数据被释放或丢弃，但这似乎就像 Box，对吗？

并不是这样。

让我们考虑一下 Rc 的一个简化实现：

```
struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    data: T,
    ref_count: usize,
}

impl<T> Rc<T> {
    fn new(data: T) -> Self {
        unsafe {
            // 如果 heap::allocate 像这样不是很好吗?
            let ptr = heap::allocate::<RcBox<T>>();
            ptr::write(ptr, RcBox {
                data: data,
                ref_count: 1,
            });
            Rc { ptr: ptr }
        }
    }

    fn clone(&self) -> Self {
        unsafe {
            (*self.ptr).ref_count += 1;
        }
        Rc { ptr: self.ptr }
    }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // drop 数据并且释放所占据的内存
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

这段代码包含了一个隐含的、微妙的假设：`ref_count` 可以装入 `usize`，因为内存中的 Rcs 不能超过 `usize::MAX`。然而这本身就假设 `ref_count` 准确反映了内存中的 Rcs 数量，我们知道用 `mem::forget` 是错误的。使用 `mem::forget` 我们可以溢出 `ref_count`，然后用大量的 Rcs 将其降至 0。然后我们就可以愉快地对内部数据进行 use-after-free 了。负负得正?

这个问题可以通过检查 `ref_count` 并做一些防御来解决。标准库的立场是直接 abort，因为你的程序肯定是摊上事儿了，摊上大事儿了。卧槽，这真是一个可笑的边界情况。

thread::scoped::JoinGuard

> 实际上这个 API 很早就从标准库中删除了，具体原因可以参考 https://github.com/rust-lang/rust/issues/24292。
>
> 原文也有人提过 issue 询问是否可以删除，得到了答复说，这个例子仍然是非常重要的，所以保留了下来：https://github.com/rust-lang/nomicon/issues/57。

thread::scoped API 旨在允许引用其父线程栈上的数据的线程被创建出来，而不需要对这些数据进行任何同步。它确保父线程在任何共享数据失效之前 join 子线程。

```
pub fn scoped<'a, F>(f: F) -> JoinGuard<'a>
    where F: FnOnce() + Send + 'a
```

这里`f`是一些闭包，供其他线程执行。这里我们定义`F: Send +'a`意思是它捕获了生命周期为`'a`的数据，而且它要么拥有该数据，要么该数据是`Sync`的（暗示`&data`是`Send`）。

因为 JoinGuard 有一个生命周期，它通过借用捕获了所有它需要的父线程中的数据。这意味着 JoinGuard 不能超过其他线程正在处理的数据的生命周期。当*JoinGuard*被丢弃时，它会 block 父线程，确保子线程中捕获的数据在父线程中 drop 之前失效。

使用方法看起来像这样：

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let mut guards = vec![];
    for x in &mut data {
        // 将可变引用移入闭包，并且在另外一个线程执行闭包，
        // 闭包有一个生命周期，由其保存的引用的生命周期决定，
        // 返回的句柄和闭包也有相同的生命周期，
        // 所以它也和闭包一样可变引用了 x，
        // 也就意味着在句柄（线程）销毁之前，我们不能访问 x
        let guard = thread::scoped(move || {
            *x *= 2;
        });
        // 将线程句柄保存起来之后使用
        guards.push(guard);
    }
    // 所有的句柄在这里被 drop，强制线程 Join（主线程在此阻塞），
    // 等到所有的线程 join 之后，其借用的数据就过期了，
    // 因此又可以在主线程中访问了
}
// 在这里数据绝对已经改变了
```

原则上，这完全是可行的！Rust 的所有权系统完美地保证了这一点！……只是它必须依赖于一个保证被调用到的析构器才是安全的。

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // 好一点的情况是存在数据竞争，更坏的是释放内存后使用的问题
        *data += 1;
    });
    //  因为 guard 被主动 forget 了，不会调用 drop 方法，主线程不会阻塞等待 guard 结束
    mem::forget(guard);
}
// Box在这里被销毁，而不确定子线程是否会在这里尝试访问它
```

在这里，一个会运行的析构器对 API 来说是非常基本的。因此它不得不被废弃，而采用完全不同的设计。

Unwinding

> 译者注：unwind 可以翻译为展开、解卷等等，但是没有找到合适的信达雅的翻译，所以暂且保留英文原文，作为固定词语处理。

Rust 有一个*分等级*的错误处理方案：

- 如果某些东西可能不存在，则使用 Option
- 如果出了问题并且可以合理地处理，则使用 Result
- 如果有什么东西出错了，而且不能合理地处理，线程就会 panic
- 如果发生了灾难性的事情，程序就会直接中止（abort）

在大多数情况下，Option 和 Result 是绝大多数人的首选，特别是因为它们可以提供了 API，可以根据用户的决定被提升为 panic 或中止。panic 会导致线程停止正常的执行，并 unwind 它的堆栈，调用析构器，就像每个函数瞬间返回一样。

从 1.0 开始，Rust 在涉及到 panic 时有两种想法。在很久以前，Rust 很像 Erlang，有轻量级的任务，而任务的目的是在达到无法维持的状态时用 panic 来杀死自己。与 Java 或 C++ 中的异常不同，panic 不能在任何时候被捕获。panic 只能被任务的所有者捕捉到，这时必须对其进行处理，否则*该*任务本身就会出现 panic。

unwind 对这个故事很重要，因为如果一个任务的析构器没有被调用，就会导致内存和其他系统资源的泄漏。由于任务会在正常执行过程中死亡，这将使 Rust 在长期运行的系统中变得非常糟糕。

随着我们今天所知道的 Rust 的出现，这种编程风格在对越来越少的抽象的推动下逐渐失去了时尚。轻量级的任务在重量级的操作系统线程中被杀死。尽管如此，在 1.0 版本的稳定版 Rust 上，panic 只能由父线程捕捉。这意味着捕捉 panic 需要使用一整个操作系统线程。不幸的是，这与 Rust 的零成本抽象理念有冲突。

有一个叫做`catch_unwind`的 API，可以在不产生线程的情况下捕捉到一个 panic。不过，我们还是鼓励你少用这个方法。特别是，Rust 目前的 unwind 实现为“不 unwind”的情况做了大量的优化。如果一个程序没有 unwind，那么这个程序在仅仅预备好 unwind 时就不应该有运行时成本。因此，实际 unwind 的成本会比 Java 中的成本高。在正常情况下，不要让你的程序来 unwind。理想情况下，你应该只为编程错误或*极端*的问题而 panic。

Rust 的 unwind 策略没有被指定为与任何其他语言的 unwind 在本质上兼容。因此，从其他语言 unwind 到 Rust，或者从 Rust unwind 到其他语言，都是未定义行为。你必须在 FFI 的边界上*绝对地*捕捉任何 panic！你在这时候（FFI 边界上捕捉到 panic 后）做什么完全由你自己决定，但你*必须*做一些事情。如果你没有做到这一点，最好的情况是你的应用程序会崩溃，在最坏的情况下，你的应用程序*不会*崩溃，但至于会发生什么？祝你好运。

异常安全

尽管程序应该很少使用 unwind，但是有很多代码是*可以* panic 的。如果你 unwrap 一个 None，索引出界，或者除以 0，你的程序就会 panic。在 debug build 中，每一个算术运算如果溢出，都会引起 panic。除非你非常小心并严格控制代码的运行，否则几乎所有的东西都可能 unwind，你需要做好准备。

在更广泛的编程世界中，为 unwind 做好准备通常被称为*异常安全*。在 Rust 中，有两个级别的异常安全需要关注：

- 在不安全的代码中，我们*必须*保证异常安全到不违反内存安全的程度。我们把这称为*最小*的异常安全。
- 在安全代码中，保证异常安全到你的程序能做正确的事情的程度（也就是说，啥都不影响，都恢复了）。我们称其为*最大限度的*异常安全。

正如 Rust 中许多地方的情况一样，不安全的代码必须准备好处理有问题的安全代码，当它涉及到 unwind 时。有可能在某一时刻创建不健壮状态的代码必须注意，panic 不会导致该状态被使用。也就是说，这意味着当这些状态存在时，只有非 panicking 的代码才会被运行；或者你需要做一个防护，在 panic 的情况下清理该状态。这并不一定意味着 panic 所见证的状态是一个完全一致的状态。我们只需要保证它是一个*安全*的状态。

大多数不安全代码都是属于叶子代码（也就是不会再调用其它函数/逻辑），因此相当容易使异常安全化。它控制着所有运行的代码，而且大多数代码都不会发生 panic。然而，不安全代码在重复调用调用者提供的代码时，与未初始化的数组打交道是很常见的。这样的代码需要小心谨慎，并考虑异常安全。

Vec::push_all

`Vec::push_all` 是一个临时性的 hack，可以在没有特例化的情况下，通过一个 slice 来高效地扩展一个 Vec。下面是一个简单的实现：

```
impl<T: Clone> Vec<T> {
    fn push_all(&mut self, to_push: &[T]) {
        self.reserve(to_push.len());
        unsafe {
            // 因为我们刚刚预留了空间，所以这里不会溢出
            self.set_len(self.len() + to_push.len());

            for (i, x) in to_push.iter().enumerate() {
                self.ptr().add(i).write(x.clone());
            }
        }
    }
}
```

我们绕过了`push`，以避免对我们明确知道有容量的 Vec 进行多余的容量和`len`检查。这个逻辑是完全正确的，只是我们的代码有一个微妙的问题：它不是异常安全的！`set_len`、`add`和`write`都没问题；但`clone`是我们忽略的 panic 炸弹。

Clone 完全不受我们的控制，而且完全可以自由地 panic。如果它这样做，我们的函数将提前退出；而因为 Vec 的长度被设置得太大了，如果 Vec 被读取或丢弃，未初始化的内存将被读取！

这种情况下的修复方法相当简单，如果我们想保证我们*已经*复制的值被丢弃，我们可以在每个循环迭代中设置`len`。如果我们只是想保证未初始化的内存不能被观察到，我们可以在循环之后设置`len`。

BinaryHeap::sift_up

把一个元素扔到堆中，比扩展一个 Vec 要复杂一些。伪代码如下：

```
bubble_up(heap, index):
    while index != 0 && heap[index] < heap[parent(index)]:
        heap.swap(index, parent(index))
        index = parent(index)
```

将这段代码按字面意思翻译成 Rust 是完全没有问题的，但是有一个坑爹的性能问题：`self`元素被无用地反复交换。因此，我们可以这么做：

```
bubble_up(heap, index):
    let elem = heap[index]
    while index != 0 && elem < heap[parent(index)]:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

这段代码确保每个元素尽可能少地被复制（事实上，在一般情况下，elem 有必要被复制两次）。但是它现在暴露了一些异常安全问题! 在任何时候，一个值都存在两个副本。如果我们在这个函数中 panic，就会有东西被重复 drop。不幸的是，我们对执行的代码并没有完全的掌控力——因为比较方法是用户定义的！

与 Vec 不同，这里的修复并不容易。一个可选的方案是将用户定义的代码和不安全的代码分成两个独立的阶段：

```
bubble_up(heap, index):
    let end_index = index;
    while end_index != 0 && heap[end_index] < heap[parent(end_index)]:
        end_index = parent(end_index)

    let elem = heap[index]
    while index != end_index:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

如果用户定义的代码炸了，那就没有问题了，因为我们还没有真正接触到堆的状态。一旦我们开始接触堆，我们就只与我们信任的数据和函数打交道，所以不存在 panic 的问题。

也许你对这种设计并不满意，不过我不得不说，这确实是在作弊! 而且我们还得做复杂的堆遍历*两次*! 好吧，让我们咬咬牙，把不可信任的和不安全的代码混在一起。

如果 Rust 像 Java 一样有`try`和`finally`，我们就可以做以下事情：

```
bubble_up(heap, index):
    let elem = heap[index]
    try:
        while index != 0 && elem < heap[parent(index)]:
            heap[index] = heap[parent(index)]
            index = parent(index)
    finally:
        heap[index] = elem
```

基本的想法很简单：如果比较出现问题，我们就把松散的元素扔到逻辑上未初始化的索引中，然后就直接返回。任何观察堆的人都会看到一个潜在的*不一致*的堆，但至少它不会导致任何双重释放问题。而如果算法正常终止，那么这个操作恰好与我们的结束方式不谋而合。

遗憾的是，Rust 没有这样的结构，所以我们需要推出我们自己的结构。这样做的方法是将算法的状态存储在一个单独的结构中，并为“最终”逻辑设置一个析构函数。无论我们是否 panic，这个析构函数都会在我们之后运行和清理：

```rust
struct Hole<'a, T: 'a> {
    data: &'a mut [T],
    /// `elt` 从始至终都是 Some
    elt: Option<T>,
    pos: usize,
}

impl<'a, T> Hole<'a, T> {
    fn new(data: &'a mut [T], pos: usize) -> Self {
        unsafe {
            let elt = ptr::read(&data[pos]);
            Hole {
                data: data,
                elt: Some(elt),
                pos: pos,
            }
        }
    }

    fn pos(&self) -> usize { self.pos }

    fn removed(&self) -> &T { self.elt.as_ref().unwrap() }

    unsafe fn get(&self, index: usize) -> &T { &self.data[index] }

    unsafe fn move_to(&mut self, index: usize) {
        let index_ptr: *const _ = &self.data[index];
        let hole_ptr = &mut self.data[self.pos];
        ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
        self.pos = index;
    }
}

impl<'a, T> Drop for Hole<'a, T> {
    fn drop(&mut self) {
        // fill the hole again
        unsafe {
            let pos = self.pos;
            ptr::write(&mut self.data[pos], self.elt.take().unwrap());
        }
    }
}

impl<T: Ord> BinaryHeap<T> {
    fn sift_up(&mut self, pos: usize) {
        unsafe {
            // 取出 `pos` 的值，然后创建一个 hole
            let mut hole = Hole::new(&mut self.data, pos);

            while hole.pos() != 0 {
                let parent = parent(hole.pos());
                if hole.removed() <= hole.get(parent) { break }
                hole.move_to(parent);
            }
            // 无论是否 panic，这里的 hole 都会被无条件填充
        }
```

```
    }
}
```

Poisoning

尽管所有不安全的代码都*必须*确保其具有最小的异常安全，但并非所有类型都能确保*最大*的异常安全；而即使类型保证了，你的代码也可能导致额外的问题。例如，一个整数当然是异常安全的，但它本身没有语义。panic 的代码有可能无法正确地更新整数，从而产生不一致的程序状态。

这*通常*是没问题的，因为任何见证异常的东西都会被销毁。例如，如果你发送一个 Vec 给另一个线程，而那个线程 panic 了，那么这个 Vec 是否处于一个奇怪的状态并不重要。它将被丢弃并永远消失。然而，有些类型特别擅长跨越 panic 边界获取值。

这些类型可以选择明确地 *毒害（Poison）* 自己，如果他们遇到了一个 panic。Poisoning 并不意味着什么特别的事情。一般来说，它只是意味着阻止正常的使用继续进行。这方面最明显的例子是标准库的 Mutex 类型。如果 Mutex 的一个 MutexGuards（当获得锁时返回的东西）在 panic 中被丢弃，Mutex 将自我中毒。今后任何试图锁定 Mutex 的行为都会返回 `Err` 或 panic。

Mutex 中毒不是为了 Rust 通常关心的真正的安全。它是作为一种安全防护措施，防止盲目地使用在锁定时发生了 panic 的 Mutex 中的数据。这样的 Mutex 中的数据可能正在被修改中，因此可能处于不一致或不完整的状态。需要注意的是，如果正确地编写了这样一个类型，就不会违反内存安全。毕竟，它必须是最低限度的异常安全的。

然而，如果 Mutex 包含，比如说，一个实际上不具备堆属性的 BinaryHeap，那么任何使用它的代码都不可能按照作者的意图运行。因此，程序不应该正常进行。不过，如果你确信你可以对这个值做*一些*事情，Mutex 还是暴露了一个方法来获得锁。毕竟，它*是*安全的，只是可能是无稽之谈。

并发和并行

Rust 作为一种语言，对如何进行并发或并行并没有什么意见。标准库暴露了操作系统线程和阻塞系统调用，因为每个人都有这些东西，而且它们足够统一，你可以以一种相对没有争议的方式提供对它们的抽象。消息传递、绿色线程和异步 API 都是多种多样的，任何对它们的抽象都会涉及到我们不愿意在 1.0 中承诺的 trade-off。

然而，Rust 建立的并发模型，使得将你自己的并发范式设计成一个库变得相对容易，并且让其他人的代码可以与你的代码一起工作。只要要求正确的生命周期、`Sync` 和 `Send`，你就可以不用担心数据竞争了。

数据竞争和竞态条件

安全的 Rust 保证没有数据竞争，数据竞争的定义是：

- 两个或多个线程同时访问一个内存位置
- 其中一个或多个线程是写的
- 其中一个或多个是非同步的

数据竞争具有未定义行为，因此在 Safe Rust 中不可能执行。数据竞争主要是通过 Rust 的所有权系统来防止的：不可能别名一个可变引用，所以不可能进行数据竞争。但内部可变性使其更加复杂，这也是我们有 Send 和 Sync Trait 的主要原因（见下文）。

然而，Rust 并没有（也无法）阻止更广泛的竞态条件。

这从根本上说是不可能的，而且说实话也是不可取的。你的硬件很糟糕，你的操作系统很糟糕，你电脑上的其他程序也很糟糕，而这一切运行的世界也很糟糕。任何能够真正声称防止*所有*竞态条件的系统，如果不是不正确的话，使用起来也是非常糟糕的。

因此，对于一个安全的 Rust 程序来说，在不正确的同步下出现死锁或做一些无意义的事情是完全“正常”的。很明显，这样的程序有问题，但 Rust 只能帮你到这里。不过，Rust 程序中的竞态条件本身并不能违反内存安全；只有与其他不安全的代码结合在一起，竞态条件才能真正违反内存安全。比如说：

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
// 使用 Arc，这样即使程序执行完毕，存储 AtomicUsize 的内存依然存在，
// 否则由于 thread::spawn 的生命周期限制，Rust 不会为我们编译这段代码
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` 捕获了 other_idx 的值，将它移入这个线程
thread::spawn(move || {
    // 因为这是一个原子变量，不存在数据竞争问题，所以可以修改 other_idx 的值
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// 因为我们只读取了一次原子的内存，因此用原子中的值做索引是安全的，
// 然后将读出的值的拷贝传递给 Vec 做为索引，
// 索引过程可以做正确的边界检查，并且在执行索引期间这个值也不会发生改变。
// 但是，如果上面的线程在执行这句代码之前增加了这个值，这段代码会 panic。
// 因为程序的正确执行（panic 几乎不可能是正确的），所以这就是一个 *竞态*，
// 其执行结果依赖于线程的执行顺序
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```rust
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];

let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` 捕获了 other_idx 值，将它移入这个线程
thread::spawn(move || {
    // 因为这是一个原子变量，不存在数据竞争问题，所以可以修改 other_idx 的值
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // 所以在边界检查之后读取 idx 的值可能是不正确的
        // 因为我们这里会 `get_unchecked`，而这个操作是 `unsafe` 的，
        // 所以这里就存在着竞态，并且 *非常危险*！
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Send 和 Sync

并不是所有的东西都服从于继承的可变性。有些类型允许你在内存中对一个位置有多个别名，并且同时修改它。除非这些类型使用同步手段来管理这种访问，否则它们绝对不是线程安全的。Rust 通过 `Send` 和 `Sync` Trait 来解决这个问题：

- 如果将一个类型发送到另一个线程是安全的，那么它就是 `Send`
- 如果一个类型可以安全地在线程间共享，那么它就是 `Sync` 的（当且仅当 `&T` 是 `Send` 时， `T` 是 `Sync` 的）

Send 和 Sync 是 Rust 的并发故事的基础。因此，存在大量的特殊工具来使它们正常工作。首先，它们是不安全的 Trait,这意味着它们的实现是不安全的，而其他不安全的代码可以认为它们是正确实现的。由于它们是*标记特性*（它们没有像方法那样的相关项目），正确实现仅仅意味着它们具有实现者应该具有的内在属性。不正确地实现 Send 或 Sync 会导致未定义行为。

Send 和 Sync 也是自动派生的 Trait。这意味着，与其它 Trait 不同，如果一个类型完全由 Send 或 Sync 类型组成，那么它就是 Send 或 Sync。几乎所有的基本数据类型都是 `Send` 和 `Sync`，因此，几乎所有你将与之交互的类型都是 `Send` 和 `Sync`。

主要的例外情况包括：

- 原始指针既不是 Send 也不是 Sync（因为它们没有安全防护）
- `UnsafeCell` 不是 Sync 的（因此 `Cell` 和 `RefCell` 也不是）
- `Rc` 不是 Send 或 Sync 的（因为 Refcount 是共享的、不同步的）

`Rc` 和 `UnsafeCell` 从根本上说不是线程安全的：它们共享了非同步的可变状态。然而，严格来说，原始指针被标记为线程不安全，更像是一个*提示*。用原始指针做任何有用的事情都需要对其进行解引用，这已经是不安全的了；当然，从这个角度上说，人们也可以认为将它们标记为线程安全的做法也没啥问题。

然而，更重要的是，它们不是线程安全的，是为了防止包含它们的类型被自动标记为线程安全的。这些类型的所有权并不明确，它们的作者也不太可能认真考虑线程安全问题。在 `Rc` 的例子中，我们有一个很好的例子，它包含一个绝对不是线程安全的 `*mut` 类型。

如果需要的话，那些没有自动派生的类型可以很简单地实现它们：

```
struct MyBox(*mut u8);

unsafe impl Send for MyBox {}
unsafe impl Sync for MyBox {}
```

在*难以置信*的罕见情况下，一个类型被不恰当地自动派生为 Send 或 Sync，那么我们也可以不实现 Send 和 Sync：

```
#![feature(negative_impls)]

// 假设我这里存在一些魔法，对于同步原语有着非常神奇的语义
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

请注意，*正常情况下*是不可能错误地派生出 Send 和 Sync 的。只有那些被其他不安全代码赋予特殊意义的类型才有可能因为不正确的 Send 或 Sync 而造成麻烦。

大多数对原始指针的使用应该被封装在一个足够的抽象后面，以便 Send 和 Sync 可以被派生。例如，所有 Rust 的标准集合都是 Send 和 Sync（当它们包含 Send 和 Sync 类型时），尽管它们普遍使用原始指针来管理内存分配和复杂的所有权。同样的，大多数这些集合的迭代器都是 Send 和 Sync 的，因为它们在很大程度上表现得像集合的 `&` 或 `&mut` 。

示例

`Box` 由于各种原因，编译器将其作为自己的特殊内建类型来实现，但是我们可以自己实现一些具有类似行为的东西，来看看什么时候实现 Send 和 Sync 是合理的。让我们把它叫做 `Carton` 。

我们先写代码，把分配在栈上的一个值，转移到堆上：

```
use std::{
    mem::{align_of, size_of},
    ptr,
    cmp::max,
};

struct Carton<T>(ptr::NonNull<T>);

impl<T> Carton<T> {
    pub fn new(value: T) -> Self {
        // 在堆上分配足够的可以存储一个类型 T 大小的空间
        assert_ne!(size_of::<T>(), 0, "Zero-sized types are out of the scope of
this example");
        let mut memptr: *mut T = ptr::null_mut();
        unsafe {
            let ret = libc::posix_memalign(
                (&mut memptr as *mut *mut T).cast(),
                max(align_of::<T>(), size_of::<usize>()),
                size_of::<T>()
            );
            assert_eq!(ret, 0, "Failed to allocate or invalid alignment");
        };

        // NonNull 仅仅是对于指针的一层封装，强制要求指针是非空的
        let ptr = {
            // 安全保证：因为我们从一个引用创建了 memptr，并且独占了所有权，所以可以解引用
            ptr::NonNull::new(memptr.cast::<T>())
                .expect("Guaranteed non-null if posix_memalign returns 0")
        };

        // 将数据从栈上复制到堆上
        unsafe {
            // 安全保证：如果 ptr 是非空的，posix_memalign 会返回一个已经内存对齐的有效
的可写指针
            ptr.as_ptr().write(value);
        }

        Self(ptr)
    }
}
```

这不是很有用，因为一旦我们的用户给了我们一个值，他们就没有办法访问它。`Box`实现了`Deref`和`DerefMut`，这样你就可以访问内部的值。让我们来做这件事：

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Carton<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        unsafe {
            // 安全保证：self 指针已经内存对齐，并且初始化了，在 `Self::new` 方法中已经
解引用，
            // 我们要求 readers 引用 Carton，而这里返回值的生命周期和输入的 self 的生命
周期对齐，
            // 因此 borrow checker 会强制保证这一点：
            // 直到这个引用被 drop，不能修改 Carton 中的内容
            self.0.as_ref()
        }
    }
}

impl<T> DerefMut for Carton<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        unsafe {
            // 安全保证：self 指针已经内存对齐，并且初始化了，在 `Self::new` 方法中已经
解引用，
            // 我们要求 writer 可写引用 Carton，而这里返回值的生命周期和输入的 self 的生
命周期对齐，
            // 因此 borrow checker 会强制保证这一点：
            // 直到这个引用被 drop，不能访问 Carton 中的内容
            self.0.as_mut()
        }
    }
}
```

最后，让我们考虑一下我们的 `Carton` 是否是 Send 和 Sync。一些东西可以安全地成为 Send，除非它与其他东西共享可变的状态，而不对其实施排他性访问。每个 `Carton` 都有一个唯一的指针，所以我们可以标记为 Send：

```
// 安全保证：除了我们没有人拥有Carton中的裸指针，因此，只需要T可以Send，Carton就可以Send
unsafe impl<T> Send for Carton<T> where T: Send {}
```

那么 Sync 呢？为了使 `Carton` 能够 Sync，我们必须强制规定，你不能对存储在一个 `Carton` 中的东西进行写入，而这个东西可以从另一个 `Carton` 中读出或写入。因为你需要一个 `&mut Carton` 来写指针，并且借用检查器强制要求可变引用必须是排他的，所以把 `Carton` 标记为 `Sync` 也没啥问题：

```
// 安全保证：存在将 `&Carton<T>` 转变为 `&T` 的公开 API，
// 而这些 API 是 unsynchronized 的（比如 `Deref`），
// 因此只有在T是 `Sync` 的情况下，`Carton<T>` 才可以是 `Sync` 的，
// 反过来说，`Carton` 本身没有使用到任何 `内部可变性`，
// 所有可变引用都只能通过独占的方式获取（`&mut`），
// 这也就意味着 `T` 的 `Sync` 特性可以传递给 `Carton<T>`
unsafe impl<T> Sync for Carton<T> where T: Sync  {}
```

当我们断言我们的类型是 Send 和 Sync 时，我们通常需要强制要求每个包含的类型都是 Send 和 Sync。当编写行为像标准库类型的自定义类型时，我们可以断言我们有相同的要求。例如，下面的代码断言，如果同类的 Box 是 Send，那么 Carton 就是 Send —— 在这种情况下，这就等于说 T 是 Send：

```
unsafe impl<T> Send for Carton<T> where Box<T>: Send {}
```

现在`Carton<T>`有一个内存泄漏，因为它从未释放它分配的内存。一旦我们解决了这个问题，我们就必须确保满足 Send 的新要求：我们需要确认`free`释放由另一个线程的分配产生的指针。我们可以在`libc::free`的文档中来确认这么做是可行的。

```
impl<T> Drop for Carton<T> {
    fn drop(&mut self) {
        unsafe {
            libc::free(self.0.as_ptr().cast());
        }
    }
}
```

一个不会发生这种情况的好例子是 MutexGuard：注意它不是 Send。MutexGuard 的实现使用的库要求你确保不会释放你在不同线程中获得的锁。如果你能够将 MutexGuard 发送到另一个线程，那么析构器就会在新的线程中运行，这就违反了该要求。但 MutexGuard 仍然可以是 Sync，因为你能发送给另一个线程的只是一个`&MutexGuard`，丢弃一个引用并没有什么作用。

TODO: 更好地解释什么可以或不可以是 Send 或 Sync。仅仅针对数据竞争就足够了？

Atomics

Rust 非常明目张胆地从 C++20 继承了原子的内存模型。这并不是因为这个模型特别优秀或容易理解。事实上，这个模型相当复杂，而且已知有几个缺陷。但不论怎么说，这是一个务实的让步，因为*每个人*在原子建模方面都相当糟糕。至少，我们可以从现有的工具和围绕 C/C++ 内存模型的研究中获益（你会经常看到这个模型被称为"C/C++11"或只是"C11"。C 只是复制了 C++ 的内存模型；而 C++11 是该模型的第一个版本，但从那时起它已经得到了一些错误的修正）。

试图在这本书中完全解释这个模型是相当无望的。它被定义为疯狂的因果关系图，需要一整本书来正确理解。如果你想了解所有琐碎的细节，你应该看看 C++ 规范。不过，我们还是会试着介绍一下基础知识和 Rust 开发者面临的一些问题。

C++ 内存模型从根本上说是为了弥补我们想要的语义、编译器想要的优化和我们的硬件想要的之间不一致的混乱之间的差距。*我们*想只写程序，让它们完全按照我们说的做，但是，你知道，一定要快。那不是很好吗?

编译器重排序

编译器从根本上希望能够进行各种复杂的转换，以减少数据的依赖性，消除死代码。特别是，他们可能会从根本上改变事件的实际顺序，或者使事件永远不会发生！比如这样的代码：

```
x = 1;
y = 3;
x = 2;
```

编译器可能会得出结论，如果你的程序这样做，那会更好：

```
x = 2;
y = 3;
```

这颠倒了事件的顺序，并且完全删除了一个事件。从单线程的角度来看，这是完全无法观察到的：在所有语句执行完毕后，我们处于完全相同的状态。但是如果我们的程序是多线程的，我们可能一直依赖`x`在`y`被分配之前实际被分配为 1。我们希望编译器能够进行这类优化，因为它们可以大量地提高性能；而另一方面，我们也希望能够相信我们的程序*做我们所说的事情*。

硬件重排序

另一方面，即使编译器完全理解我们的意图并尊重我们的意愿，我们的硬件可能反而会给我们带来麻烦。麻烦来自于 CPU 的内存层次结构。在你的硬件中确实有一个全局共享的内存空间，但从每

个 CPU 核心的角度来看，它是*非常遥远的*，而且*非常慢*。每个 CPU 宁可使用其本地的数据缓存，而只在其缓存中没有该内存的时候才去和共享内存对话，这是很痛苦的。

毕竟，这就是缓存的全部意义所在，对吗？如果每次从缓存中读出的数据都要跑回共享内存中去仔细检查是否有变化，那还有什么意义呢？最终的结果是，硬件并不能保证在*一个*线程上以某种顺序发生的事件，在*另一个*线程上以同样的顺序发生。为了保证这一点，我们必须向 CPU 发出特殊指令，让它变得不那么聪明。

例如，假设我们说服编译器发出这样的逻辑：

```
initial state: x = 0, y = 1

线程 1            线程 2
y = 3;           if x == 1 {
x = 1;               y *= 2;
                 }
```

理想情况下，这个程序有两种可能的最终状态：

- `y = 3`：线程 2 在线程 1 完成之前做了检查
- `y = 6`：线程 2 在线程 1 完成后做了检查

然而，还有第三种潜在的状态是硬件可以实现的：

- `y = 2`：线程 2 看到了 `x = 1`，但没有看到 `y = 3`，然后改写了 `y = 3`

值得注意的是，不同种类的 CPU 提供不同的保证。通常将硬件分为两类：强有序和弱有序。最值得注意的是 x86/64 提供强有序保证，而 ARM 提供弱有序保证。这对并发编程有两个后果：

- 在强有序的硬件上要求更强的保证可能很便宜，甚至是无开销的，因为它们已经无条件地提供了强保证；较弱的保证可能只在弱有序的硬件上产生性能优势
- 在强有序硬件上要求太弱的保证，更有可能*恰巧*发生作用，即使你的程序严格来说是不正确的；如果可能的话，并发算法应该在弱有序的硬件上进行测试

数据访问

C++ 内存模型试图通过允许我们谈论我们程序的*因果性*来弥补这一差距。一般来说，这是通过在程序的各个部分和运行它们的线程之间建立一种*happen-before*的关系。这给了硬件和编译器一定的自由度，在没有建立严格的 happen-before 关系的地方更积极地优化程序，但也迫使他们在建立了关系的地方更加小心。我们沟通这些关系的方式是通过*数据访问（data accesses）和原子访问（atomic accesses）*。

数据访问是编程世界的主体，它们从根本上说是不同步的，编译器可以自由地对它们进行积极的优化。特别是，数据访问可以自由地被编译器重新排序，前提是程序是单线程的。硬件也可以自由地将数据访问中的变化传播给其他线程，只要它想，就可以懒散地、不一致地传播。最关键的是，数

据访问是数据竞争发生的方式。数据访问对硬件和编译器非常友好，但正如我们所看到的，如果试图用它来编写同步代码，它提供的语义太弱了。

仅仅使用数据访问是不可能写出正确的同步代码的。

原子访问是我们告诉硬件和编译器我们的程序是多线程的方式。每个原子访问都可以用一个*顺序*来标记，指定它与其他访问的关系。在实践中，这可以归结为告诉编译器和硬件它们*不能*做的某些事情。对于编译器来说，这主要是围绕着指令的重新排序展开的。对于硬件来说，这主要是围绕着如何将写操作传播给其他线程。Rust 所提供的顺序集合是：

- 顺序一致（Squentially Consistent，SeqCst）
- Release
- Acquire
- Relaxed

（注意：我们明确地不暴露 C++ 的 *consume* 排序）

TODO：消极推理与积极推理？TODO：“不能忘记同步”

顺序一致性

顺序一致是所有顺序中最强大的，它意味着包含所有其他顺序的限制。直观地说，一个顺序一致的操作不能被重新排序：一个线程上所有发生在 SeqCst 访问之前和之后的访问都保持在它之前和之后。一个只使用顺序一致的原子和数据访问的无数据竞争程序有一个非常好的特性，即有一个所有线程都同意的程序指令的单一全局执行的顺序。这种执行方式也特别好推理：它只是每个线程的单独执行的交错。如果你开始使用较弱的原子顺序，这就不成立了（译者注：也就是说，同一时刻，针对同一个别名/内存位置，仅能有一条指令在执行，不能出现并发）。

顺序一致性对开发者的相对友好并不是免费的。即使在强排序的平台上，顺序一致性也会涉及到内存屏障。

在实践中，顺序一致性对于程序的正确性很少有必要。然而，如果你对其他的内存顺序没有信心的话，顺序一致性绝对是正确的选择。让你的程序运行得比它需要的慢一点，肯定比它运行得不正确要好！从机制上来说，降低原子操作的等级，以便在以后拥有较弱的一致性也是很容易的。只要把`SeqCst`改成`Relaxed`就可以了！当然，证明这种转换是*正确的*是一个完全不同的问题。

Acquire-Release

Acquire 和 Release 在很大程度上是用来配对使用的。它们的名字暗示了它们的使用情况：它们非常适合于获取和释放锁，并确保关键部分不会重叠。

直观地说，一个 Acquire 的访问可以确保它之后的每一个访问都保持在它之后。然而，在 Acquire 之前发生的操作可以自由地被重新排序到它之后发生。同样地，一个 Release 访问确保它之前的

每一个访问都保持在它之前。然而，在 Release 之后发生的操作可以自由地被重新排序到它之前发生。

当线程 A Release 了内存中的一个位置，然后线程 B 随后 Acquire 了内存中*相同*的位置，因果关系就建立了。在 A Release 之前发生的每一个写（包括非原子写和 Relaxed 的原子写）都会在 B Acquire 之后被观察到。然而，与任何其他线程的因果关系都没有建立。同样地，如果 A 和 B 访问内存中*不同*的位置，也不会建立因果关系。

因此，Release-Acquire 的基本用法很简单：你 Acquire 一个内存位置来开始关键部分，然后 Release 这个位置来结束它。例如，一个简单的自旋锁可能看起来像这样：

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // 我上锁了吗

    // ... 用某种方式将锁分发到各个线程(thread::spawn) ...

    // 尝试将原子变量设置为 true，以此来获得锁
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // 从循环中跳出，说明此时已经获取了锁

    // ... 恐怖的数据访问 ...

    // 工作完成了，释放锁
    lock.store(false, Ordering::Release);
}
```

在强有序平台上，大多数访问都有 Release 或 Acquire 语义，使得 Release 和 Acquire 往往是完全免费的。而在弱有序平台上则不是这样。

Relaxed

Relaxed 的访问是绝对最弱的。它们可以被自由地重新排序，并且不提供任何 happen-before 的关系。不过，Relaxed 的操作仍然是原子性的。也就是说，它们不算是数据访问，对它们进行的任何读-改-写操作都是原子性的。Relaxed 操作适用于那些你肯定希望发生，但并不特别在意的事情。例如，如果你不使用计数器来同步任何其他访问，那么多个线程可以安全地使用 Relaxed 的 `fetch_add` 来增加一个计数器。

在强有序平台上，Relaxed 操作很少有好处，因为它们通常提供 Release-Acquire 的语义。然而，在弱有序平台上，Relaxed 的操作会更便宜。

示例：实现 Vec

为了将所有的东西整合起来，我们将从头开始编写 `std::Vec`。我们将限制自己使用稳定的 Rust。特别是我们不会使用任何可以让我们的代码变得更漂亮或更高效的内建指令，因为内建指令是永远不稳定的。尽管许多内建指令*确实*在其他地方变得稳定了（`std::ptr` 和 `std::mem` 由许多内建指令组成）。

最终，这意味着我们的实现可能不会利用所有可能的优化，但它也绝不是*简陋*的。我们肯定会在细枝末节的细节上钻牛角尖，即使问题并不*真的*值得这样做。

你想要高级的。我们要的就是高级。

布局

首先，我们需要想出结构布局。一个 Vec 有三个部分：一个指向分配的指针，分配的大小，以及已经初始化的元素数量。

直观来说，这意味着我们只需要这样的设计：

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

这确实可以编译成功。但是不幸的是，这有些过于严格了。编译器会给我们太严格的可变性（variance）。比如一个 `&Vec<&'static str>` 不能用在预期 `&Vec<&'a str>` 的地方。参见所有权和生命周期一章中关于可变和 drop checker 的所有细节。

正如我们在所有权一章中看到的，当标准库拥有一个分配对象的原始指针时，它使用 `Unique<T>` 来代替 `*mut T`。Unique 是不稳定的，所以如果可能的话，我们希望不要使用它。

简而言之，Unique 是一个原始指针的包装，并声明以下内容：

- 我们对 `T` 是协变的
- 我们可以拥有一个 `T` 类型的值（这和我们在这的例子无关，但是可以参考 `PhantonData` 那章来看看为什么真正的 `std::vec::Vec<T>` 需要这个）
- 如果 `T` 是 `Send/Sync`，我们就是 `Send/Sync`。
- 我们的指针从不为空（所以 `Option<Vec<T>>` 是空指针优化的）

我们可以在稳定的 Rust 中实现上述所有的要求。为此，我们不使用 `Unique<T>`，而是使用 `NonNull<T>`，这是对原始指针的另一种包装，它为我们提供了上述的两个属性，即它在 `T` 上是协变的，并且被声明为永不为空。通过在 `T` 是 `Send/Sync` 的情况下实现 `Send/Sync`，我们得到与使用 `Unique<T>` 相同的结果：

```
use std::ptr::NonNull;
use std::marker::PhantomData;

pub struct Vec<T> {
    ptr: NonNull<T>,
    cap: usize,
    len: usize,
}

unsafe impl<T: Send> Send for Vec<T> {}
unsafe impl<T: Sync> Sync for Vec<T> {}
```

分配内存

使用`NonNull`会给 Vec（甚至是所有的 std collections）的一个重要特性带来麻烦：创建一个空的 Vec 实际上根本就没有分配。这与分配一个零大小的内存块不同，因为全局分配器不允许这样做（会导致未定义行为！）。所以，如果我们不能分配，但也不能在`ptr`里放一个空指针，我们在`Vec::new`里做什么？好吧，我们只是在里面放一些其他的垃圾值。

这并不会有问题，因为我们已经有了`cap == 0`作为尚未分配的哨兵。我们甚至不需要在任何代码中特别处理它，因为我们通常需要检查`cap > len`或`len > 0`。在这里，Rust 推荐使用的值是`mem::align_of::<T>()`。`NonNull`为此提供了一个便利。`NonNull::dangling()`。有相当多的地方我们会想使用`dangling`，因为没有真正的分配可言，但`null`会让编译器做坏事。

所以，代码如下：

```
use std::mem;

impl<T> Vec<T> {
    pub fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "We're not ready to handle ZSTs");
        Vec {
            ptr: NonNull::dangling(),
            len: 0,
            cap: 0,
        }
    }
}
```

我在这里使用了断言，是因为零大小的类型需要在我们的代码中进行一些特殊的处理，我想把这个问题暂时延后。如果没有这个断言，我们早期的一些实现会导致一些非常糟糕的事情。

接下来，我们需要弄清楚，当我们*确实*想要分配内存时，究竟该怎么做。为此，我们使用全局分配函数`alloc`、`realloc`和`dealloc`，这些函数在稳定的 Rust 中可以使用`std::alloc`。在`std::alloc::Global`类型稳定后，这些函数将被废弃。

我们还需要一种方法来处理内存不足（OOM）的情况。标准库提供了一个函数`alloc::handle_alloc_error`，它将以特定平台的方式中止程序。我们选择中止而不是 panic 的原因是，unwinding 会导致分配的发生，而当你的分配器刚刚回来说“嘿，我没有更多的内存了”时，这似乎是一件坏事。

当然，这看起来有点蠢，因为大多数平台实际上不会以传统方式耗尽内存。如果你顺理成章地用完了所有的内存，你的操作系统可能会通过其他方式杀死这个应用程序。我们最有可能触发 OOM 的方式是一次性要求大量的内存（例如，理论地址空间的一半）。因此，panic *可能*是没问题的，不会发生什么坏事。不过，我们还是想尽可能地像标准库一样，所以我们就把整个程序杀掉。

好了，现在我们可以写 grow 的代码了，简单来说，逻辑应该是这样的：

```
if cap == 0:
    allocate()
    cap = 1
else:
    reallocate()
    cap *= 2
```

但是 Rust 唯一支持的分配器 API 太低级了，我们需要做相当多的额外工作。我们还需要防范一些特殊情况，这些情况可能发生在真正的大分配或空分配中。

特别是，`ptr::offset`会给我们带来很多麻烦，因为它有 LLVM 的 GEP（译者注：GetElementPtr） inbounds 指令的语义。如果你有幸没有处理过这个指令，这里是 GEP 的大致故事：别名分析、别名分析、别名分析！对于一个优化的编译器来说，能够推理出数据的依赖性和别名是超级重要的。

作为一个简单的例子，考虑下面的代码片段：

```
*x *= 7;
*y *= 3;
```

如果编译器能够证明`x`和`y`指向内存中的不同位置，理论上这两个操作可以并行执行（例如将它们加载到不同的寄存器中，并对它们独立工作）。然而，编译器在一般情况下不能这样做，因为如果 x 和 y 指向内存中的同一位置，操作需要对相同的值进行，而且它们不能在事后被合并。

当你使用 GEP inbounds 时，你就是在明确地告诉 LLVM，你要做的偏移是在一个“已分配”对象的范围内（within the bounds of a single "allocated" entity.）。这达到的效果是，LLVM 可以假设，如果两个指针已知指向两个不相干的对象，那么这些指针的所有偏移量*也*被认为不会导致别名（因为你不会在内存中的某个随机地方结束）。LLVM 对 GEP 的偏移量进行了大量的优化，而界内偏移量（inbounds offsets）是所有偏移量中最好的，所以我们尽可能地使用它们是很重要的。

这是 GEP 的作用，它怎么会给我们带来麻烦呢?

第一个问题是，我们用无符号的整数来索引数组，但是 GEP（以及由此产生的`ptr::offset`）需要一个有符号的整数。这意味着一半的看似有效的数组索引会溢出 GEP，并且在实际上是走错了方向！因此，我们必须将所有的分配限制在`isize::MAX`元素。这实际上意味着我们只需要担心字节大小的对象，因为例如`> isize::MAX``u16`s 将真正耗尽系统的所有内存。然而，为了避免出现微妙的边界情况，即有人将一些`< isize::MAX`对象的数组重新解释为字节，std 将所有分配限制为`isize::MAX`字节。

在 Rust 目前支持的所有 64 位平台上，我们被人为地限制在明显少于所有 64 位的地址空间（现代 x64 平台只暴露了 48 位寻址），所以我们可以依靠首先耗尽内存。然而，在 32 位目标上，特别是那些有扩展使用更多地址空间的目标（PAE x86 或 x32），理论上是可以成功分配超过`isize::MAX`字节的内存的。

然而，由于这是一个教程，我们在这里不会特别优化，只是无条件地检查，而不是使用聪明的平台特定的`cfg`s。

我们需要担心的另一个情况是空分配。我们需要担心两种空分配的情况。对于任意 T：`cap = 0`；和对于零大小的类型（zero-sized types）`cap > 0`。

这些情况很棘手，因为它们归结于 LLVM 对“分配”的理解。LLVM 的分配概念要比我们通常使用的方式抽象得多。因为 LLVM 需要与不同语言的语义和自定义分配器一起工作，所以它不能真正深入地理解分配。相反，分配背后的主要想法是“不与其他东西重叠”。也就是说，堆分配、栈分配和 globals 不会随机地重叠在一起。没错，这就是别名分析。因此，Rust 在技术上可以对分配的概念做一些快速和松散的处理，只要它是*一致的*。

回到空分配的情况，有几个地方我们想用 0 来抵消，这是通用代码的结果。那么问题来了：这样做是否一致？对于零大小的类型，我们的结论是，用任意数量的元素进行 GEP 界内偏移确实是一致的。这是一个运行时的无用功，因为每个元素都不占用空间，假装在 `0x01` 处有无限的零尺寸类型分配也是可以的。没有分配器会分配这个地址，因为他们不会分配 `0x00`，而且他们一般会分配到高于一个字节的最小对齐。另外，一般来说，整个第一页的内存是被保护的，不会被分配（在许多平台上，是整个 4k）。

然而，对于正值大小的类型怎么办呢？这个问题就有点棘手了。原则上，你可以说 0 的偏移量没有给 LLVM 带来任何信息：要么地址之前有一个元素，要么在它之后，但它不能知道是哪个。然而，我们选择了保守的假设，即它可能会做坏事。因此，我们将明确地防止这种情况。

呼。

好了，说了这么多废话，让我们实际分配一些内存吧：

```rust
use std::alloc::{self, Layout};

impl<T> Vec<T> {
    fn grow(&mut self) {
        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // 因为 self.cap <= isize::MAX，所以不会溢出
            let new_cap = 2 * self.cap;

            // `Layout::array` 会检查申请的空间是否小于等于 usize::MAX，
            // 但是因为 old_layout.size() <= isize::MAX，
            // 所以这里的 unwrap 永远不可能失败
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // 保证新申请的内存没有超出 `isize::MAX` 字节的大小
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too
large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // 如果分配失败，`new_ptr` 就会成为空指针，我们需要对应 abort 的操作
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}
```

Push 和 Pop

好了，我们现在可以初始化，也可以分配了。让我们实际实现一些功能吧! 让我们从`push`开始。它所需要做的就是检查我们是否已经满了并 grow，然后无条件地写到下一个索引，接着增加我们的长度。

在写入时，我们必须注意不要对我们想要写入的内存做解引用。最坏的情况是，它是来自分配器的真正未初始化的内存（里面是垃圾值）。最好的情况是，它是我们 pop 出的一些旧值的地址。无论是哪种情况，我们都不能索引到那个地址并解引用，因为这将把该内存认为是一个 T 类型的存活的实例；更糟的是，`foo[idx] = x`会试图在`foo[idx]`的旧值上调用`drop`!

正确的方法是使用`ptr::write`，它只是盲目地用我们提供的值的位来覆盖目标地址，而不会对该地址做解引用。

对于`push`，如果旧的 len（在 push 被调用之前）是 0，那么我们正好想写到第 0 个索引，所以我们应该用旧的 len 来作为写入的索引。

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.as_ptr().add(self.len), elem);
    }

    // 不可能出错，因为出错之前一定会 OOM(out of memory)
    self.len += 1;
}
```

是不是很简单! 那么`pop`呢? 虽然这次我们要访问的索引被初始化了，但 Rust 不会让我们直接解构内存的位置来把实例移动（move）出去，因为这将使内存未被初始化（译者注：和 push 一样，如果 pop 出的是在 Vec 的内存中的值，那么当这个值被丢弃后，Vec 的这块内存会被 drop，这就出大事了）! 为此我们需要`ptr::read`，它只是从目标地址复制出 bit，并将其解释为 T 类型的值。这将使这个地址的内存在逻辑上未被初始化，尽管事实上那里有一个完美的 T 的实例。

对于`pop`，举个例子，如果旧的 len 是 1，那我们正好想从第 0 个索引中读出，所以我们应该用新的 len 来作为读出的索引。

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.as_ptr().add(self.len)))
        }
    }
}
```

释放内存

接下来我们应该实现 Drop，这样我们就不会大规模地泄漏大量的资源。最简单的方法是直接调用 `pop`，直到它产生 None，然后再释放我们的 buffer。注意，如果 `T: !Drop` 的话，调用 `pop` 是不需要的。理论上，我们可以询问 Rust 是否 `T` `need_drop` 并省略对 `pop` 的调用。然而在实践中，LLVM 在做类似这样的简单的无副作用的删除代码方面*非常*好，所以我就省得麻烦了，除非你注意到它没有被优化掉（在这种情况下它被优化了）。

要注意的是，当 `self.cap == 0` 时，我们不能调用 `alloc::dealloc`，因为在这种情况下我们实际上没有分配任何内存。

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.ptr.as_ptr() as *mut u8, layout);
            }
        }
    }
}
```

Deref

好了！我们已经实现了一个基本像样的栈。我们可以 push 和 pop，我们还可以自己 drop。然而，我们还需要一大堆的功能。特别是， 尽管我们有了一个合适的数组，但还没有切片的功能。这其实很容易解决：我们可以实现`Deref<Target=[T]>`。这将神奇地使我们的 Vec 在各种条件下强制成为一个切片，并且表现得像一个切片。

我们只需要`slice::from_raw_parts`。它将为我们正确处理空切片。之后当我们设置了零大小的类型支持，它也会对这些类型进行正确的处理。

```
use std::ops::Deref;

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            std::slice::from_raw_parts(self.ptr.as_ptr(), self.len)
        }
    }
}
```

还有 DerefMut：

```
use std::ops::DerefMut;

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            std::slice::from_raw_parts_mut(self.ptr.as_ptr(), self.len)
        }
    }
}
```

现在我们有了`len`、`first`、`last`、索引、切片、排序、`iter`、`iter_mut`，以及 slice 提供的其他各种功能啦！

插入和删除

slice 不提供的东西是`insert`和`remove`，所以我们接下来做这些。

insert 需要将目标索引的所有元素向右移动一个。要做到这一点，我们需要使用`ptr::copy`，它是 C 语言`memmove`的 Rust 版本。它将一些内存块从一个位置复制到另一个位置，正确处理源和目标重叠的情况（这在这里肯定会发生）。

如果我们在索引`i`处插入，我们要使用旧的 len 将`[i ... len]`转移到`[i+1 ... len+1]`。

```
pub fn insert(&mut self, index: usize, elem: T) {
    // 注意：`<=` 是因为我们可以把值插入到任何索引范围 ([0,length-1]) 内的位置之后
    // 这种情况等同于 push
    assert!(index <= self.len, "index out of bounds");
    if self.cap == self.len { self.grow(); }

    unsafe {
        // ptr::copy(src, dest, len) 的含义： "从 src 复制连续的 len 个元素到 dst "
        ptr::copy(
            self.ptr.as_ptr().add(index),
            self.ptr.as_ptr().add(index + 1),
            self.len - index,
        );
        ptr::write(self.ptr.as_ptr().add(index), elem);
        self.len += 1;
    }
}
```

remove 的行为方式正好相反。我们需要将所有的元素从`[i+1 ... len + 1]`转移到`[i ... len]`，使用*新的* len。

```
pub fn remove(&mut self, index: usize) -> T {
    // 注意：使用 `<` 是因为 index 不能删除超出元素下标的范围
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.as_ptr().add(index));
        ptr::copy(
            self.ptr.as_ptr().add(index + 1),
            self.ptr.as_ptr().add(index),
            self.len - index,
        );
        result
    }
}
```

IntoIter

让我们继续，接下来写迭代器。`iter`和`iter_mut`已经为我们写好了，感谢 Deref 的魔法。然而，有两个有趣的迭代器是 Vec 提供的，而 slice 不能提供：`into_iter`和`drain`。

IntoIter 通过消耗掉 Vec 的值（获取 Vec 的所有权），并因此可以产生其元素的值（所有权）。为了实现这个目的，IntoIter 需要控制 Vec 的分配。

IntoIter 也需要是 DoubleEnded，以便能够从两端读取。从后面读取可以通过调用`pop`来实现，但是从前面读取就比较困难了。我们可以调用`remove(0)`，但这将是非常昂贵的。相反，我们将使用`ptr::read`来复制 Vec 两端的值，而不改变缓冲区。

为了做到这一点，我们将使用一个非常常见的 C 语言的数组迭代习惯。我们将建立两个指针；一个指向数组的开始，另一个指向数组结束后的一个元素。当我们想从一端获得一个元素时，我们将读出指向那一端的值，并将指针移到另一端。当这两个指针相等时，我们就知道我们已经完成了。

注意，对于`next`和`next_back`来说，读取和偏移的顺序是相反的。对于`next_back`来说，指针总是在它想读取的元素之后，而对于`next`来说，指针正好在它想读取的元素上。要想知道为什么会这样，请考虑除一个元素外的所有元素都已经产生的情况。

这个数组看起来像这样：

```
          S  E
[X, X, X, O, X, X, X]
```

如果 E 直接指向它想产生的下一个元素，它将与没有更多元素可以产生的情况没有区别。

虽然我们在迭代过程中实际上并不关心它，但我们也需要保留 Vec 的分配信息，以便在 IntoIter 被丢弃后释放它。

所以我们将使用下面的结构。

```
pub struct IntoIter<T> {
    buf: NonNull<T>,
    cap: usize,
    start: *const T,
    end: *const T,
}
```

而这就是我们最终的初始化结果：

```rust
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        // 确保 Vec 不会被 drop，因为那样会释放内存
        let vec = ManuallyDrop::new(self);

        // 因为 Vec 实现了 Drop，所以我们不能销毁它
        let ptr = vec.ptr;
        let cap = vec.cap;
        let len = vec.len;

        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: ptr.as_ptr(),
                end: if cap == 0 {
                    // 不能通过这个指针获取偏移，因为没有分配内存
                    ptr.as_ptr()
                } else {
                    ptr.as_ptr().add(len)
                },
            }
        }
    }
}
```

向前迭代：

```rust
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = (self.end as usize - self.start as usize)
                  / mem::size_of::<T>();
        (len, Some(len))
    }
}
```

向后迭代：

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}
```

因为 IntoIter 拥有其分配的所有权，它需要实现 Drop 来释放它；并且，它也需要在 Drop 里丢弃它所包含的任何没有被迭代到的元素。

```
impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            // 将剩下的元素 drop
            for _ in &mut *self {}
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.buf.as_ptr() as *mut u8, layout);
            }
        }
    }
}
```

RawVec

我们实际上在这里达到了一个有趣的状态：我们在 Vec 和 IntoIter 中重复了指定缓冲区和释放其内存的逻辑。现在我们已经实现了它，并且确定了*实际的*逻辑重复，这是一个进行一些逻辑压缩的好时机。

我们将抽象出`(ptr, cap)`对，并为它们编写分配、增长和释放的逻辑：

```rust
struct RawVec<T> {
    ptr: NonNull<T>,
    cap: usize,
}

unsafe impl<T: Send> Send for RawVec<T> {}
unsafe impl<T: Sync> Sync for RawVec<T> {}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO: implement ZST support");
        RawVec {
            ptr: NonNull::dangling(),
            cap: 0,
        }
    }

    fn grow(&mut self) {
        // 保证新申请的内存没有超出 `isize::MAX` 字节
        let new_cap = if self.cap == 0 { 1 } else { 2 * self.cap };

        // `Layout::array` 会检查申请的空间是否小于等于 usize::MAX,
        // 但是因为 old_layout.size() <= isize::MAX,
        // 所以这里的 unwrap 永远不可能失败
        let new_layout = Layout::array::<T>(new_cap).unwrap();

        // 保证新申请的内存没有超出 `isize::MAX` 字节
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too
large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // 如果分配失败，`new_ptr` 就会成为空指针，我们需要对应 abort 的操作
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.ptr.as_ptr() as *mut u8, layout);
            }
        }
```

```
    }
}
```

随后，把 Vec 改成这样：

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T {
        self.buf.ptr.as_ptr()
    }

    fn cap(&self) -> usize {
        self.buf.cap
    }

    pub fn new() -> Self {
        Vec {
            buf: RawVec::new(),
            len: 0,
        }
    }

    // push/pop/insert/remove 这些操作做了小小的改变，如下所示：
    // * `self.ptr.as_ptr() -> self.ptr()`
    // * `self.cap -> self.cap()`
    // * `self.grow() -> self.buf.grow()`
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // RawVec 来负责释放内存
    }
}
```

最后，我们可以把 IntoIter 改得相当简单：

```rust
pub struct IntoIter<T> {
    _buf: RawVec<T>, // 我们实际上并不关心这个，只需要他们保证分配的空间不被释放
    start: *const T,
    end: *const T,
}

// next 和 next_back 保持不变，因为它们并没有用到 buf

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        // 我们只需要确保 Vec 中所有元素都被读取了，
        // 在这之后这些元素会被自动清理
        for _ in &mut *self {}
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // 需要使用 ptr::read 非安全地把 buf 移出，因为它没有实现 Copy，
            // 而且 Vec 实现了 Drop Trait（因此我们不能销毁它）
            let buf = ptr::read(&self.buf);
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: buf.ptr.as_ptr(),
                end: if buf.cap == 0 {
                    // 不能通过这个指针获取偏移，除非已经分配了内存
                    buf.ptr.as_ptr()
                } else {
                    buf.ptr.as_ptr().add(len)
                },
                _buf: buf,
            }
        }
    }
}
```

是不是好多了！

Drain

接下来，让我们来实现 Drain。 Drain 与 IntoIter 大体上相同，只是它不是消耗 Vec，而是借用 Vec，并且不会修改到其分配。现在我们只实现“基本”的全范围版本。

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // 这里需要限制生命周期，因此我们使用了 `&'a mut Vec<T>`，
    // 也就是我们语义上包含的内容，
    // 我们只会调用 `pop()` 和 `remove(0)` 两个方法
    vec: PhantomData<&'a mut Vec<T>>,
    start: *const T,
    end: *const T,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
```

——等等，这看着好像很眼熟？IntoIter 和 Drain 有完全相同的结构，让我们再做一些抽象：

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // 构建 RawValIter 是不安全的，因为它没有关联的生命周期，
    // 将 RawValIter 存储在与它实际分配相同的结构体中是非常有必要的，
    // 但这里是具体的实现细节，不用对外公开
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // 如果 `len = 0`，说明没有分配内存，需要避免使用 offset，
                // 因为那样会给 LLVM 的 GEP 传递错误的信息
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
            }
        }
    }
}

// Iterator 和 DoubleEndedIterator 和 IntoIter 实现起来很类似
```

IntoIter 我们可以改成这样：

```
pub struct IntoIter<T> {
    _buf: RawVec<T>,
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}
```

请注意，我在这个设计中留下了一些奇怪之处，以使升级 Drain 来处理任意的子范围更容易一些。特别是我们*可以*让 RawVallter 在 drop 时 drain 自身，但这对更复杂的 Drain 来说是不合适的。我们还使用了一个 slice 来简化 Drain 的初始化。

好了，现在实现 Drain 真的很容易了：

```
use std::marker::PhantomData;

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // 这里事关 mem::forget 的安全。
            // 如果 Drain 被 forget，我们就会泄露整个 Vec 的内存，
            // 既然我们始终要做这一步，为何不在这里完成呢？
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}
```

关于`mem::forget`问题的更多细节，参见关于泄漏的章节。

处理零大小的类型

是时候了！我们将与 ZST（零大小类型）这个幽灵作斗争。安全的 Rust 从来不需要关心这个问题，但是 Vec 在原始指针和原始分配上非常密集，这正是需要关心零尺寸类型的两种情况。我们需要注意两件事：

- 如果你在分配大小上传入 0，原始分配器 API 有未定义的行为。
- 原始指针偏移量对于零大小的类型来说是无效的（no-ops），这将破坏我们的 C 风格指针迭代器。

幸好我们之前把指针迭代器和分配处理分别抽象为 `RawValIter` 和 `RawVec`。现在回过头来看，多么的方便啊。

分配零大小的类型

那么，如果分配器 API 不支持零大小的分配，我们到底要把什么作为我们的分配来存储呢？当然是 `NonNull::dangling()`！几乎所有使用 ZST 的操作都是 no-op，因为 ZST 正好有且仅有一个值，因此在存储或加载它们时不需要考虑状态。这实际上延伸到了 `ptr::read` 和 `ptr::write`：它们实际上根本不会去用指针。因此，我们从来不需要改变指针。

然而，请注意，我们之前对在溢出前耗尽内存的防御，在零大小的类型中不再有效了。我们必须明确地防止零大小类型的容量溢出。

由于我们目前的架构，这意味着要写 3 个边界处理，在 `RawVec` 的每个方法中都有一个：

```
impl<T> RawVec<T> {
    fn new() -> Self {
        // 这一段分支代码在编译期间就可以计算出结果返回的结果，返回给 cap
        let cap = if mem::size_of::<T>() == 0 { usize::MAX } else { 0 };

        // `NonNull::dangling()` 有双重含义:
        // `未分配内存 (unallocated)`, `零尺寸 (zero-sized allocation)`
        RawVec {
            ptr: NonNull::dangling(),
            cap: cap,
        }
    }

    fn grow(&mut self) {
        // 因为当 T 的尺寸为 0 时我们设置了 cap 为 usize::MAX
        // 这一步成立意味着 Vec 溢出了
        assert!(mem::size_of::<T>() != 0, "capacity overflow");

        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // 保证新申请的内存没有超出 `isize::MAX` 字节
            let new_cap = 2 * self.cap;

            // `Layout::array` 会检查申请的空间是否小于等于 usize::MAX,
            // 但是因为 old_layout.size() <= isize::MAX,
            // 所以这里的 unwrap 永远不可能失败
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // 保证新申请的内存没有超出 `isize::MAX` 字节
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too
large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // 如果分配失败，`new_ptr` 就会成为空指针，我们需要处理这个意外情况
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();
```

```
        if self.cap != 0 && elem_size != 0 {
            unsafe {
                alloc::dealloc(
                    self.ptr.as_ptr() as *mut u8,
                    Layout::array::<T>(self.cap).unwrap(),
                );
            }
        }
    }
}
```

搞定！我们现在支持 push 和 pop 零大小类型。不过，我们的迭代器（不是由 slice Deref 提供的）仍然是一团浆糊。

迭代 ZST

零大小的偏移量是 no-op。这意味着我们目前的设计总是将 `start` 和 `end` 初始化为相同的值，而我们的迭代器将一无所获。目前的解决方案是将指针转为整数，增加，然后再转回。

```
impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
            },
        }
    }
}
```

现在，我们有了另一个 bug：我们的迭代器不再是完全不运行，而是现在的迭代器*永远*都在运行。我们需要在我们的迭代器 impls 中做同样的技巧。另外，我们的 size_hint 计算代码将对 ZST 除以 0。既然我们会把这两个指针当作是指向字节的，所以我们就把大小 0 映射到除以 1，这样的话 `next` 的代码如下：

```
fn next(&mut self) -> Option<T> {
    if self.start == self.end {
        None
    } else {
        unsafe {
            let result = ptr::read(self.start);
            self.start = if mem::size_of::<T>() == 0 {
                (self.start as usize + 1) as *const _
            } else {
                self.start.offset(1)
            };
            Some(result)
        }
    }
}
```

你找到 bug 了嘛？没人看到！连最初的作者也是几年之后闲逛这个页面的时候，觉得这段代码比较可疑，因为这里直接滥用了迭代器的指针当作了*计数器*，而这就使得了指针不对齐！在使用 ZST 的时候，我们*唯一的工作*就是必须保证指针对齐！*啊这*!

原始指针在任何时候都不需要对齐，所以使用指针作为计数器的基本技巧是*没问题的*，但是当它们被传递给 `ptr::read` 时，它们*应该*是对齐的! 这*可能*是不必要的迂腐操作，因为 `ptr::read` 在处理 ZST 时其实是个 noop，但让我们*稍微*负责一点，当遇到 ZST 时从 `NonNull::dangling` 读取。

（或者你也可以在 ZST 路径上调用 `read_unaligned` 。两者都可以。因为无论哪种方式，我们都是在无中生有，而且都最终编译成 noop。）

```
impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.start = (self.start as usize + 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    let old_ptr = self.start;
                    self.start = self.start.offset(1);
                    Some(ptr::read(old_ptr))
                }
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.end = (self.end as usize - 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    self.end = self.end.offset(-1);
                    Some(ptr::read(self.end))
                }
            }
        }
    }
}
```

好啦，迭代器也搞定啦！

最终的代码

```
use std::alloc::{self, Layout};
use std::marker::PhantomData;
use std::mem;
use std::ops::{Deref, DerefMut};
use std::ptr::{self, NonNull};

struct RawVec<T> {
    ptr: NonNull<T>,
    cap: usize,
}

unsafe impl<T: Send> Send for RawVec<T> {}
unsafe impl<T: Sync> Sync for RawVec<T> {}

impl<T> RawVec<T> {
    fn new() -> Self {
        // !0 等价于 usize::MAX,  这一段分支代码在编译期间就可以计算出结果返回的结果，返回给 cap
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // `NonNull::dangling()` 有双重含义:
        // `未分配内存 (unallocated)`, `零尺寸 (zero-sized allocation)`
        RawVec {
            ptr: NonNull::dangling(),
            cap: cap,
        }
    }

    fn grow(&mut self) {
        // 因为当 T 的尺寸为 0 时，我们设置了 cap 为 usize::MAX,
        // 这一步成立便意味着 Vec 溢出了.
        assert!(mem::size_of::<T>() != 0, "capacity overflow");

        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // 保证新申请的内存没有超出 `isize::MAX` 字节
            let new_cap = 2 * self.cap;

            // `Layout::array` 会检查申请的空间是否小于等于 usize::MAX,
            // 但是因为 old_layout.size() <= isize::MAX,
            // 所以这里的 unwrap 永远不可能失败
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // 保证新申请的内存没有超出 `isize::MAX` 字节
        assert!(
            new_layout.size() <= isize::MAX as usize,
            "Allocation too large"
        );
```

```
        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // 如果分配失败，`new_ptr` 就会成为空指针，我们需要处理该意外情况
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();

        if self.cap != 0 && elem_size != 0 {
            unsafe {
                alloc::dealloc(
                    self.ptr.as_ptr() as *mut u8,
                    Layout::array::<T>(self.cap).unwrap(),
                );
            }
        }
    }
}

pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T {
        self.buf.ptr.as_ptr()
    }

    fn cap(&self) -> usize {
        self.buf.cap
    }

    pub fn new() -> Self {
        Vec {
            buf: RawVec::new(),
            len: 0,
        }
    }
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap() {
            self.buf.grow();
        }

        unsafe {
```

```
            ptr::write(self.ptr().add(self.len), elem);
        }

        // 不会溢出，会先 OOM
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe { Some(ptr::read(self.ptr().add(self.len))) }
        }
    }

    pub fn insert(&mut self, index: usize, elem: T) {
        assert!(index <= self.len, "index out of bounds");
        if self.cap() == self.len {
            self.buf.grow();
        }

        unsafe {
            ptr::copy(
                self.ptr().add(index),
                self.ptr().add(index + 1),
                self.len - index,
            );
            ptr::write(self.ptr().add(index), elem);
            self.len += 1;
        }
    }

    pub fn remove(&mut self, index: usize) -> T {
        assert!(index < self.len, "index out of bounds");
        unsafe {
            self.len -= 1;
            let result = ptr::read(self.ptr().add(index));
            ptr::copy(
                self.ptr().add(index + 1),
                self.ptr().add(index),
                self.len - index,
            );
            result
        }
    }

    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // 这里事关 mem::forget 的安全。
            // 如果 Drain 被 forget，我们就会泄露整个 Vec 的内存
            // 既然我们始终要做这一步，为何不在这里完成呢？
            self.len = 0;

            Drain {
```

```
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // RawVec 来负责释放内存
    }
}

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe { std::slice::from_raw_parts(self.ptr(), self.len) }
    }
}

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe { std::slice::from_raw_parts_mut(self.ptr(), self.len) }
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);
            let buf = ptr::read(&self.buf);
            mem::forget(self);
            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}

struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
```

```
            },
        }
    }
}

impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.start = (self.start as usize + 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    let old_ptr = self.start;
                    self.start = self.start.offset(1);
                    Some(ptr::read(old_ptr))
                }
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.end = (self.end as usize - 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    self.end = self.end.offset(-1);
                    Some(ptr::read(self.end))
                }
            }
        }
    }
}

pub struct IntoIter<T> {
    _buf: RawVec<T>,    // 我们实际上并不关心这个，只需要他们保证分配的空间不被释放
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
```

```
        self.iter.next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        self.iter.size_hint()
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.iter.next_back()
    }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        self.iter.next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        self.iter.size_hint()
    }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> {
        self.iter.next_back()
    }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        // 消耗drain
        for _ in &mut *self {}
    }
}
```

实现 Arc 和 Mutex

了解理论是很好的，但是理解某件事**最好**的方法是使用它。为了更好地理解原子和内部可变性，我们将实现标准库中的`Arc`和`Mutex`类型。

TODO：编写`Mutex`章节。

实现 Arc

在本节中，我们将实现一个更简单的`std::sync::Arc`。与我们之前做的`Vec`的实现类似，我们不会像标准库那样利用许多优化、内建指令或不稳定的代码。

这个实现大致上基于标准库的实现（技术上可以认为是取自 1.49 中的`alloc::sync`，因为它实际上是在那里实现的），但它目前不支持弱引用，因为它们使实现稍微复杂一些。

请注意，这一部分目前还处于 WIP 阶段。

布局

让我们开始为我们的`Arc`的实现做布局。

一个`Arc<T>`为`T`类型的值提供了线程安全的共享所有权，并在堆中分配。在 Rust 中，共享意味着不变性，所以我们不需要设计任何东西来管理对该值的访问，对吧？虽然像 Mutex 这样的内部可变性类型允许 Arc 的用户创建共享可变性，但 Arc 本身并不需要关注这些问题。

然而，有一个地方 Arc 需要关注可变：销毁。当 Arc 的所有所有者都销毁时，我们需要能够`drop`其内容并释放其分配。所以我们需要一种方法让所有者知道它是否是最后一个所有者，而最简单的方法就是对所有者进行计数——引用计数。

不幸的是，这种引用计数本质上是共享的可变状态，所以 Arc *需要* 考虑同步问题。我们可以为此使用 Mutex，但那太过于杀鸡用牛刀了。相反，我们将使用 atomics。既然每个人都需要一个指向 T 的分配的指针，我们也可以把引用计数放在同一个分配中。

直观地说，它看起来就像这样：

```
use std::sync::atomic;

pub struct Arc<T> {
    ptr: *mut ArcInner<T>,
}

pub struct ArcInner<T> {
    rc: atomic::AtomicUsize,
    data: T,
}
```

这可以编译通过，然而它是不正确的。首先，编译器会给我们太严格的可变性。例如，在期望使用`Arc<&'a str>`的地方不能使用`Arc<&'static str>`。更重要的是，它将给 drop checker 提供不正确的所有权信息，因为它将假定我们不拥有任何`T`类型的值。由于这是一个提供值的共享所有权的结构，在某些时候会有一个完全拥有其数据的结构实例。参见关于所有权和生命周期的章节，了解关于变异和 drop checker 的所有细节。

为了解决第一个问题，我们可以使用`NonNull<T>`。请注意，`NonNull<T>`是一个围绕原始指针的包装，并声明以下内容：

- 我们是`T`的协变
- 我们的指针从不为空

为了解决第二个问题，我们可以包含一个包含`ArcInner<T>`的`PhantomData`标记。这将告诉 drop checker，我们对`ArcInner<T>`（它本身包含`T`）的值有一些所有权的概念。

通过这些改变，我们得到了最终的结构：

```rust
use std::marker::PhantomData;
use std::ptr::NonNull;
use std::sync::atomic::AtomicUsize;

pub struct Arc<T> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}

pub struct ArcInner<T> {
    rc: AtomicUsize,
    data: T,
}
```

基本代码

现在我们已经确定了实现 `Arc` 的布局，让我们开始写一些基本代码。

构建 Arc

我们首先需要一种方法来构造一个 `Arc<T>` 。

这很简单，因为我们只需要把 `ArcInner<T>` 扔到一个 Box 里并得到一个 `NonNull<T>` 的指针。

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // 当前的指针就是第一个引用，因此初始时设置 count 为 1
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // 我们从 Box::into_raw 得到该指针，因此使用 `.unwrap()` 是完全可行的
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}
```

Send 和 Sync

由于我们正在构建并发原语，因此我们需要能够跨线程发送它。因此，我们可以实现 `Send` 和 `Sync` 标记特性。有关这些的更多信息，请参阅有关 `Send` 和 `Sync` 的部分。

这是没问题的，因为：

- 当且仅当你拥有唯一的 Arc 引用时，你才能获得其引用数据的可变引用（这仅发生在 `Drop` 中）
- 我们使用原子操作进行共享可变引用计数

```
unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}
```

我们需要约束 `T: Sync + Send` ，因为如果我们不提供这些约束，就有可能通过 `Arc` 跨越线程边界共享不安全的值，这有可能导致数据竞争或不可靠。

例如，如果没有这些约束，`Arc<Rc<u32>>`将是`Sync + Send`，这意味着你可以从`Arc`中克隆出`Rc`来跨线程发送（不需要创建一个全新的`Rc`），这将产生数据竞争，因为`Rc`不是线程安全的.

获取ArcInner

为了将`NonNull<T>`指针解引用为`T`，我们可以调用`NonNull::as_ref`。这是不安全的，与普通的`as_ref`函数不同，所以我们必须这样调用它。

```
unsafe { self.ptr.as_ref() }
```

在这段代码中，我们将多次使用这个片段（通常与相关的`let`绑定）。

这种不安全是没问题的，因为当这个`Arc`存活的时候，我们可以保证内部指针是有效的。

Deref

好了。现在我们可以制作`Arc`了（很快就能正确地克隆和销毁它们），但是我们怎样才能获得里面的数据呢?

我们现在需要的是一个`Deref`的实现。

我们需要导入该 Trait:

```
use std::ops::Deref;
```

这里是实现:

```
impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}
```

看着很简单，对不? 这只是解除了对`ArcInner<T>`的`NonNull`指针的引用，然后得到了对里面数据的引用。

代码

下面是本节的所有代码。

```
use std::ops::Deref;

impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // 当前的指针就是第一个引用，因此初始时设置 count 为 1
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // 我们从 Box::into_raw 得到该指针，因此使用 `.unwrap()` 是完全可行的
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}

unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}


impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}
```

克隆

现在我们已经有了一些基本的代码，我们需要一种方法来克隆`Arc`。

我们大致需要：

1. 递增原子引用计数
2. 从内部指针构建一个新的`Arc`实例

首先，我们需要获得对`ArcInner`的访问。

```
let inner = unsafe { self.ptr.as_ref() };
```

我们可以通过以下方式更新原子引用计数：

```
let old_rc = inner.rc.fetch_add(1, Ordering::???);
```

但是我们在这里应该使用什么顺序？我们实际上没有任何代码在克隆时需要原子同步，因为我们在克隆时不修改内部值。因此，我们可以在这里使用 Relaxed 顺序，这意味着没有 happen-before 的关系，但却是原子性的。然而，当`Drop` Arc 时，我们需要在递减引用计数时进行原子同步。这在关于`Arc`的`Drop`实现部分中有更多描述。关于原子关系和 Relaxed ordering 的更多信息，请参见atomics 部分。

因此，代码变成了这样：

```
let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);
```

我们需要增加一个导入来使用`Ordering`。

```
use std::sync::atomic::Ordering;
```

然而，我们现在的这个实现有一个问题：如果有人决定`mem::forget`一堆 Arc 怎么办？到目前为止，我们所写的代码（以及将要写的代码）假设引用计数准确地描绘了内存中的 Arc 的数量，但在`mem::forget`的情况下，这是错误的。因此，当越来越多的 Arc 从这个 Arc 中克隆出来，而它们又没有被`Drop`和参考计数被递减时，我们就会溢出！这将导致释放后使用（use-after-free）。这是**非常糟糕的事情！**

为了处理这个问题，我们需要检查引用计数是否超过某个任意值（低于`usize::MAX`，因为我们把引用计数存储为`AtomicUsize`），*并做一些防御*。

标准库的实现决定，如果任何线程上的引用计数达到`isize::MAX`（大约是`usize::MAX`的一半），就直接中止程序（因为在正常代码中这是非常不可能的情况，如果它发生，程序可能是非常有问题的）。基于的假设是，不应该有大约 20 亿个线程（或者在一些 64 位机器上大约**9万亿**个）在同时增加引用计数。这就是我们要做的。

实现这种行为是非常简单的。

```
if old_rc >= isize::MAX as usize {
    std::process::abort();
}
```

然后，我们需要返回一个新的`Arc`的实例。

```
Self {
    ptr: self.ptr,
    phantom: PhantomData
}
```

现在，让我们把这一切包在`Clone`的实现中。

```
use std::sync::atomic::Ordering;

impl<T> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        let inner = unsafe { self.ptr.as_ref() };
        // 我们没有修改 Arc 中的数据，因此在这里不需要任何原子的同步操作，
        // 使用 relax 这种排序方式也就完全可行了
        let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);

        if old_rc >= isize::MAX as usize {
            std::process::abort();
        }

        Self {
            ptr: self.ptr,
            phantom: PhantomData,
        }
    }
}
```

丢弃

我们现在需要一种方法来减少引用计数，并在计数足够低时丢弃数据，否则数据将永远存在于堆中。

为了做到这一点，我们可以实现 `Drop`。

我们大致需要：

1. 递减引用计数
2. 如果数据只剩下一个引用，那么：
3. 原子化地对数据进行屏障，以防止对数据的使用和删除进行重新排序
4. 丢弃内部数据

首先，我们需要获得对 `ArcInner` 的访问：

```
let inner = unsafe { self.ptr.as_ref() };
```

现在，我们需要递减引用计数。为了简化我们的代码，如果从 `fetch_sub` 返回的值（递减引用计数之前的值）不等于 `1`，我们可以直接返回（我们不是数据的最后一个引用）。

```
if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
    return;
}
```

然后我们需要创建一个原子屏障来防止重新排序使用数据和删除数据。正如标准库对 `Arc` 的实现中所述。

> 需要这个内存屏障来防止数据使用的重新排序和数据的删除。因为它被标记为 `Release`，引用计数的减少与 `Acquire` 屏障同步。这意味着数据的使用发生在减少引用计数之前，而减少引用计数发生在这个屏障之前，而屏障发生在数据的删除之前。（译者注：use < decrease < 屏障 < delete）
>
> 正如Boost 文档中所解释的那样。
>
> ---
>
> > 强制要求一个线程中对该对象的任何可能的访问（通过现有的引用）*发生在不同线程中删除该对象之前*是很重要的。这可以通过在丢弃一个引用后的“Release”操作来实现（任何通过该引用对对象的访问显然必须在之前发生），以及在删除对象前的“Acquire”操作。
>
> ---
>
> 特别是，虽然 Arc 的内容通常是不可改变的，但有可能对类似 Mutex 的东西进行内部可变。由于 Mutex 在被删除时不会被获取，我们不能依靠它的同步逻辑来使线程 A 的写操作对线程 B 的析构器可见。

> 还要注意的是，这里的 Acquire fence 可能可以用 Acquire load 来代替，这可以在高度竞争的情况下提高性能。 参见2。

为了做到这一点，我们可以这么做：

```
use std::sync::atomic;
atomic::fence(Ordering::Acquire);
```

最后，我们可以 drop 数据本身。我们使用 `Box::from_raw` 来丢弃 Box 中的 `ArcInner<T>` 和它的数据。这需要一个 `*mut T` 而不是 `NonNull<T>`，所以我们必须使用 `NonNull::as_ptr` 进行转换。

```
unsafe { Box::from_raw(self.ptr.as_ptr()); }
```

这是安全的，因为我们知道我们拥有的是最后一个指向 `ArcInner` 的指针，而且其指针是有效的。

现在，让我们在 `Drop` 的实现中把这一切整合起来。

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        let inner = unsafe { self.ptr.as_ref() };
        if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
            return;
        }
        // 我们需要防止针对 inner 的使用和删除的重排序，
        // 因此使用 fence 来进行保护是非常有必要的
        atomic::fence(Ordering::Acquire);
        // 安全保证：我们知道这是最后一个对 ArcInner 的引用，并且这个指针是有效的
        unsafe { Box::from_raw(self.ptr.as_ptr()); }
    }
}
```

最终代码

这就是我们的最终代码，我在这里加了一些额外的注释并排序了一下 imports：

```rust
use std::marker::PhantomData;
use std::ops::Deref;
use std::ptr::NonNull;
use std::sync::atomic::{self, AtomicUsize, Ordering};

pub struct Arc<T> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}

pub struct ArcInner<T> {
    rc: AtomicUsize,
    data: T,
}

impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // 当前的指针就是第一个引用，因此初始时设置 count 为 1
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // 我们从 Box::into_raw 得到该指针，因此使用 `.unwrap()` 是完全可行的
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}

unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}

impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}

impl<T> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        let inner = unsafe { self.ptr.as_ref() };
        // 我们没有修改 Arc 中的数据，因此在这里不需要任何原子的同步操作，
        // 使用 relax 这种排序方式也就完全可行
        let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);

        if old_rc >= isize::MAX as usize {
            std::process::abort();
        }

        Self {
            ptr: self.ptr,
            phantom: PhantomData,
        }
```

```
    }
}

impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        let inner = unsafe { self.ptr.as_ref() };
        if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
            return;
        }
        // 我们需要防止针对 inner 的使用和删除的重排序
        // 因此使用 fence 来进行保护是非常有必要
        atomic::fence(Ordering::Acquire);

        // 安全保证：我们知道这是最后一个对 ArcInner 的引用，并且这个指针是有效的
        unsafe { Box::from_raw(self.ptr.as_ptr()); }
    }
}
```

外部函数接口（FFI）

简介

本指南将使用snappy压缩/解压缩库作为为外部代码编写绑定的示例。Rust 目前无法直接调用 C++ 库，但 snappy 包括一个 C 接口（在`snappy-c.h`）。

关于 libc 的说明

这些例子中有许多使用了the `libc` crate，它为 C 类型提供了各种类型定义，以及其他东西。如果你要自己尝试这些例子，你需要在你的`Cargo.toml`中加入`libc`：

```
[dependencies]
libc = "0.2.0"
```

调用外部函数

下面是一个调用外部函数的最小例子，如果你安装了 snappy，它就可以被编译：

```
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

`extern`块是一个外部库中的函数签名列表，在本例中是平台的 C ABI。`#[link(...)]`属性用来指示链接器与 snappy 库进行链接，以便解析这些符号。

外部函数被认为是不安全的，所以对它们的调用需要用`unsafe {}`来包装，作为对编译器的承诺，其中包含的所有内容都是安全的。C 库经常暴露出不是线程安全的接口，而且几乎所有接受指针参数的函数都对一些输入是无效的，因为指针可能是悬空的，而原始指针不在 Rust 的安全内存模型之内。

当声明一个外部函数的参数类型时，Rust 编译器不能检查声明是否正确，所以正确指定它是在运行时保持绑定正确的一部分。

`extern`块可以被扩展到覆盖整个 snappy API：

```
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
```

创建一个安全的接口

原始的 C 语言 API 需要被包装起来，以提供内存安全，并使用更高级别的概念，如向量。一个库可以选择只公开安全的高级接口而隐藏不安全的内部细节。

封装一个需要内存 buffer 参数的函数需要使用`slice::raw`模块来操作 Rust Vec 作为内存的指针。Rust 的 Vec 被保证为一个连续的内存块，长度是当前包含的元素数，容量是分配的内存的总大小（元素），其中长度必定小于或等于容量：

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) ==
0
    }
}
```

上面的`validate_compressed_buffer`包装器使用了一个`unsafe`块，但它通过在函数签名中去掉`unsafe`来保证调用它对所有输入都是安全的。

`snappy_compress`和`snappy_uncompress`函数更复杂，因为还需要分配一个缓冲区来容纳输出。

`snappy_max_compressed_length`函数可以用来分配一个最大容量的 Vec，以容纳压缩后的输出，然后该向量可以作为输出参数传递给`snappy_compress`函数。还会传递一个输出参数来检索

压缩后的真实长度，以便设置长度：

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

解压缩也是类似的，因为 snappy 将未压缩的大小作为压缩格式的一部分来存储，`snappy_uncompressed_length` 将检索出所需的确切缓冲区大小：

```
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

然后，我们可以添加一些测试来展示如何使用它们：

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn valid() {
        let d = vec![0xde, 0xad, 0xd0, 0x0d];
        let c: &[u8] = &compress(&d);
        assert!(validate_compressed_buffer(c));
        assert!(uncompress(c) == Some(d));
    }

    #[test]
    fn invalid() {
        let d = vec![0, 0, 0, 0];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
    }

    #[test]
    fn empty() {
        let d = vec![];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
        let c = compress(&d);
        assert!(validate_compressed_buffer(&c));
        assert!(uncompress(&c) == Some(d));
    }
}
```

析构器

外部的库经常把资源的所有权交给调用代码，当这种情况发生时，我们必须使用 Rust 的析构器来提供安全并保证这些资源的释放（尤其是在 panic 的情况下）。

关于析构器的更多信息，请参见 Drop trait。

从 C 调用 Rust 代码

你可能想要把 Rust 代码编译成某种形式，以便在 C 中调用。这个并不难，不过需要一些额外的步骤。

Rust 代码侧

首先，我们假设你有一个 lib 库名字叫 `rust_from_c`，其中的 `lib.rs` 应该包含类似这样的代码：

```
#[no_mangle]
pub extern "C" fn hello_from_rust() {
    println!("Hello from Rust!");
}
```

`extern "C"` 使得这个函数使用 C 的调用规约，正如下文外部调用规约一章所述。 `no_mangle` 属性关闭了 Rust 的 name mangling 特性，这使得我们在链接时有个明确定义的符号名。

接下来，为了把我们的 Rust 代码编译成一个可以直接从 C 调用的共享库，我们需要加这些到 `Cargo.toml` 中：

```
[lib]
crate-type = ["cdylib"]
```

（注意：我们也可以用 `staticlib` 类型，不过这会需要我们修改一些链接的参数。）

接下来，执行 `cargo build`，Rust 侧就搞定啦！

C 代码侧

我们将写一段 C 代码来调用 `hello_from_rust` 并用 `gcc` 来编译。

C 代码大致是这样：

```
extern void hello_from_rust();

int main(void) {
    hello_from_rust();
    return 0;
}
```

我们把这个文件命名为 `call_rust.c`，并且把它放到我们 crate 的根目录下，然后编译：

```
gcc call_rust.c -o call_rust -lrust_from_c -L./target/debug
```

`-l` 和 `-L` 告诉 gcc 去找我们的 Rust 库。

最后，我们可以通过指定 `LD_LIBRARY_PATH` 来从 C 调用 Rust：

```
$ LD_LIBRARY_PATH=./target/debug ./call_rust
Hello from Rust!
```

搞定！ 如果需要更多实际的例子，可以参考 `cbindgen`。

从 C 代码到 Rust 函数的回调

一些外部库需要使用回调来向调用者报告其当前状态或中间数据，我们可以将 Rust 中定义的函数传递给外部库。这方面的要求是，回调函数被标记为“extern”，并有正确的调用约定，使其可以从 C 代码中调用。

然后，回调函数可以通过注册调用发送到 C 库中，之后再从那里调用。

一个基本的例子是：

Rust 代码：

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // 触发回调
    }
}
```

C 代码：

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // 在 Rust 中会调用回调函数 callback(7)
}
```

在这个例子中，Rust 的`main()`将调用 C 语言中的`trigger_callback()`，而这又会回调 Rust 中的`callback()`。

针对 Rust 对象的回调

前面的例子展示了如何从 C 代码中调用一个全局函数，然而，人们通常希望回调是针对一个特殊的 Rust 对象，这可能是代表相应的 C 对象的封装器的对象。

这可以通过向 C 库传递一个指向该对象的原始指针来实现，然后，C 库可以在通知中包含指向 Rust 对象的指针，这将使回调能够不安全地访问引用的 Rust 对象。

Rust 代码：

```
struct RustObject {
    a: i32,
    // 其余的成员...
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // 在回调函数中更新 RustObject 的内容
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // 创建一个会被在回调函数中引用的 RustObject
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C 代码：

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // 这会调用 Rust 代码中的 callback(&rustObject, 7)
}
```

异步回调

在之前给出的例子中，回调是作为对外部 C 库的函数调用的同步调用的。为了执行回调，对当前线程的控制从 Rust 切换到 C，再切换到 Rust，但最终回调是在调用触发回调的函数的同一线程上执行。

当外部库生成自己的线程并从那里调用回调时，事情会变得更加复杂。在这种情况下，对回调中的 Rust 数据结构的访问特别不安全，必须使用适当的同步机制。除了像 mutex 这样的经典同步机制，Rust 中的一种可能性是使用通道（在`std::sync::mpsc`中），将数据从调用回调的 C 线程转发到 Rust 线程。

如果一个异步回调的目标是 Rust 地址空间中的一个特殊对象，那么在相应的 Rust 对象被销毁后，C 库也绝对不能再进行回调。这可以通过在对象的析构器中取消对回调的注册来实现，并以保证在取消注册后不执行回调的方式设计库。

链接

`extern`块上的`link`属性提供了基本的构建模块，用于指示 rustc 如何链接到本地库。现在有两种可接受的 link 属性的形式：

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

在这两种情况下，`foo`是我们要链接的本地库的名称，在第二种情况下，`bar`是编译器要链接的本地库的类型。目前已知有三种类型的本地库：

- 动态 - `#[link(name = "readline")]`
- 静态 - `#[link(name = "my_build_dependency", kind = "static")]`
- 框架 - `#[link(name = "CoreFoundation", kind = "framework")]`

注意，框架只在 macOS 上可用。

不同的 `kind` 值是为了区分本地库如何参与链接。从链接的角度来看，Rust 编译器创建了两种类型的工件：部分（rlib/staticlib）和最终（dylib/binary）。原生的动态库和框架依赖被传播到最终的可执行文件中，而静态库的依赖则完全不被传播，因为静态库被直接集成到后续的可执行文件中的。

来看几个这个模型如何使用的例子：

- 一个本地构建依赖。有时在编写一些 Rust 代码时需要一些 C/C++ 胶水，但以库的形式分发 C/C++ 代码是一种负担。在这种情况下，代码将被归档到 `libfoo.a`，然后 Rust crate 将通过 `#[link(name = "foo", kind = "static")]` 声明一个依赖关系。

  无论 crate 的输出是什么，本地静态库都会被包含在输出中，这意味着本地静态库的分发是没有必要的。

- 一个正常的动态依赖。常见的系统库（如 `readline`）在大量的系统上可用，而这些库的静态副本往往找不到。当这种依赖被包含在 Rust crate 中时，部分目标（如 rlibs）将不会链接到该库，但当 rlib 被包含在最终目标（如二进制）中时，本地库将被链接进来。

在 macOS 上，框架的行为与动态库的语义相同。

不安全块

一些操作，如取消引用原始指针或调用被标记为不安全的函数，只允许在不安全块中进行。不安全块隔离了不安全因素，并向编译器承诺不安全因素不会从块中泄露出去。

另一方面，不安全的函数则向世界公布了它。一个不安全的函数是这样写的：

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

这个函数只能从一个“不安全”块或另一个“不安全”函数中调用。

访问外部的全局变量

外部的 API 经常输出一个全局变量，它可以做一些类似于跟踪全局状态的事情。为了访问这些变量，你可以在 `extern` 块中用 `static` 关键字来声明它们：

```
#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             unsafe { rl_readline_version as i32 });
}
```

另外，你可能需要改变由外部接口提供的全局状态。要做到这一点，可以用 `mut` 声明全局变量，这样我们就可以改变它们：

```
use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

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

        rl_prompt = ptr::null();
    }
}
```

注意，所有“可变全局变量”的交互都是不安全的，包括读和写。处理全局可变状态需要非常小心。

外部调用规约

大多数外部代码都暴露了一个 C ABI，Rust 在调用外部函数时默认使用平台的 C 调用约定。一些外部函数，最明显的是 Windows API，使用了其他的调用约定。Rust 提供了一种方法来告诉编译器应该使用哪种约定：

```
#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

这适用于整个 `extern` 块。支持的 ABI 约束列表如下：

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `thiscall`
- `vectorcall` 这是目前隐藏在 `abi_vectorcall` 特性开关后面的，可能会有变化
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`
- `sysv64`

这个列表中的大多数 ABI 是不言自明的，但是 `system` ABI 可能看起来有点奇怪。这个约束条件选择了任何合适的 ABI 来与目标库进行交互操作。例如，在 x86 架构的 win32 上，这意味着使用的 ABI 是 `stdcall`。然而，在 x86_64 上，windows 使用 `C` 调用惯例，所以将使用 `C`。这意味着在我们之前的例子中，我们可以使用 `extern "system" { ... }` 来为所有的 windows 系统定义一个块，而不仅仅是 x86 系统。

与外部代码的互操作性

只有当 `#[repr(C)]` 属性应用于一个 `struct` 时，Rust 才能保证该结构的布局与平台的 C 语言表示兼容。`#[repr(C, packed)]` 可以用来布局结构成员而不需要填充。`#[repr(C)]` 也可以应用于枚举。

Rust 的 Box 类型（`Box<T>`）使用不可为空的指针作为句柄，指向所包含的对象。然而，它们不应该被手动创建，因为它们是由内部分配器管理的。引用可以安全地被认为是直接指向该类型的不可归零的指针。然而，打破借用检查或可变性规则是不安全的，所以如果需要的话，最好使用原始指针（`*`），因为编译器不能对它们做出那么多假设。

向量和字符串共享相同的基本内存布局，并且在 `vec` 和 `str` 模块中提供了与 C API 工作的实用程序。然而，字符串不是以 `\0` 结束的。如果你需要一个以 NUL 结尾的字符串与 C 语言互通，你应该使用 `std::ffi` 模块中的 `CString` 类型。

crates.io 上的 `libc` crate包括 `libc` 模块中的 C 标准库的类型别名和函数定义，Rust 默认与 `libc` 和 `libm` 链接。

Variadic 函数

在 C 语言中，函数可以是“variadic”，这意味着它们接受可变数量的参数。这在 Rust 中可以通过在外部函数声明的参数列表中指定“...”来实现：

```
extern {
    fn foo(x: i32, ...);
}

fn main() {
    unsafe {
        foo(10, 20, 30, 40, 50);
    }
}
```

正常的 Rust 函数*不能*是可变参数的：

```
// 这不会编译通过

fn foo(x: i32, ...) {}
```

"空指针优化"

某些 Rust 类型被定义为永不为"空"。这包括引用（`&T`，`&mut T`），Box（`Box<T>`），和函数指针（`extern "abi" fn()`）。当与 C 语言对接时，经常使用可能为"空"的指针，这似乎需要一些混乱的`transmute`和/或不安全的代码来处理与 Rust 类型的转换。然而，尝试构造或者使用这些无效的值**是 undefined behavior**，所以你应当使用如下的变通方法。

作为一种特殊情况，如果一个`enum`正好包含两个变体，其中一个不包含数据，另一个包含上面列出的非空类型的字段，那么它就有资格获得"空指针优化"。这意味着不需要额外的空间来进行判别；相反，空的变体是通过将一个`null`的值放入不可空的字段来表示。这被称为"优化"，但与其他优化不同，它保证适用于符合条件的类型。

最常见的利用空指针优化的类型是`Option<T>`，其中`None`对应于`null`。所以`Option<extern "C" fn(c_int) -> c_int>`是使用 C ABI（对应于 C 类型`int (*)(int)`）来表示可空函数指针的一种正确方式。

这里有一个臆造的例子：假设某个 C 库有一个用于注册回调的工具，在某些情况下会被调用。回调被传递给一个函数指针和一个整数，它应该以整数为参数运行该函数。所以我们有函数指针在 FFI 边界上双向飞行。

```
use libc::c_int;

extern "C" {
    /// 注册回调函数
    fn register(cb: Option<extern "C" fn(Option<extern "C" fn(c_int) -> c_int>,
c_int) -> c_int>);
}

// 这个函数其实没什么实际的用处，
// 它从C代码接受一个函数指针和一个整数，
// 用整数做参数，调用指针指向的函数，并返回函数的返回值，
// 如果没有指定函数，那默认就返回整数的平方
extern "C" fn apply(process: Option<extern "C" fn(c_int) -> c_int>, int: c_int)
-> c_int {
    match process {
        Some(f) => f(int),
        None    => int * int
    }
}

fn main() {
    unsafe {
        register(Some(apply));
    }
}
```

而 C 语言方面的代码看起来是这样的：

```
void register(int (*f)(int (*)(int), int)) {
    ...
}
```

实际上，不需要 `transmute`！

FFI 和 unwinding

在使用 FFI 时，必须注意 unwinding。大多数 ABI 的名称有两种变体，一种带有 `-unwind` 后缀而另一种不带。`Rust` 的 ABI 总是允许 unwinding，所以不存在 `Rust-unwind` ABI。

如果你希望 Rust `panic`s 或是外部（例如：C++）的异常能够穿越 FFI 的边界，则必须使用正确的 `-unwind` ABI。相反，如果你不希望 unwinding 可以穿越 FFI 边界，使用非 `unwind` 的 ABI。

> 注意：编译时指定 `panic=abort` 会导致 `panic!` 立即终止进程，无论发生 `panic` 的函数指定了何种 ABI。

如果一个 unwinding 操作遇到了不允许 unwind 的 ABI 边界，具体行为会由 unwinding 的源头决定（Rust `panic` 或是外部异常）：

- `panic` 会导致进程安全终止。
- 外部异常会导致未定义行为。

注意 `catch_unwind` 和外部异常的交互行为**是未定义的**，同样， `panic` 和外部异常处理机制的交互也是一样（尤其是 C++ 的 `try` / `catch` ）。

Rust `panic` 与 `"C-unwind"`

```
#[no_mangle]
extern "C-unwind" fn example() {
    panic!("Uh oh");
}
```

该函数（当编译时指定 `panic=unwind` 时）可以 unwind C++ 的栈帧。

```
[通过 `catch_unwind` 停止 unwinding 的 Rust 函数 ]
      |
     ...
      |
   [C++ 栈]
      |                            ^
      | (调用)                      | (向上 unwinding)
      v                            |
[Rust 函数 `example`]               |
      |                            |
      +----- rust 函数 panics -----+
```

如果 C++ 的栈上包含对象，它们将会被析构。

C++ `throw` 与 `"C-unwind"`

```
#[link(...)]
extern "C-unwind" {
    // 一个可能会抛出异常的 C++ 函数
    fn may_throw();
}
#[no_mangle]
extern "C-unwind" fn rust_passthrough() {
    let b = Box::new(5);
    unsafe { may_throw(); }
    println!("{:?}", &b);
}
```

一个有 `try` 语句块的 C++ 函数可以通过调用 `rust_passthrough` 捕获被 `may_throw` 抛出的异常。

```
[在 `try` 语句块中调用 `rust_passthrough`的 C++ 函数]
      |
     ...
      |
[Rust 函数 `rust_passthrough`]
      |                                ^
      | (调用)                          | (向上 unwinding)
      v                                |
[C++ 函数 `may_throw`]                 |
      |                                |
      +------ C++ 函数抛出异常 ------+
```

如果 `may_throw` 抛出了一个异常， `b` 会被正常丢弃。否则将会打印 `5` 。

`panic` 可以在 ABI 边界处停止

```rust
#[no_mangle]
extern "C" fn assert_nonzero(input: u32) {
    assert!(input != 0)
}
```

如果以 `0` 为参数调用了 `assert_nonzero` ，运行时可以保证（安全地）终止进程, 无论编译时是否指定了 `panic=abort` 。

提前捕获 `panic`

在写可能会 panic 的 Rust 代码时，如果不希望进程在其 panic 时被终止，必须使用 `catch_unwind` ：

```rust
use std::panic::catch_unwind;

#[no_mangle]
pub extern "C" fn oh_no() -> i32 {
    let result = catch_unwind(|| {
        panic!("Oops!");
    });
    match result {
        Ok(_) => 0,
        Err(_) => 1,
    }
}

fn main() {}
```

请注意， `catch_unwind` 只捕捉 unwind 的 panic，而不是那些中止进程的 panic。更多信息请参见 `catch_unwind` 的文档。

表示不透明（opaque）的结构

有时，一个 C 语言库想提供一个指向某东西的指针，但又不想让你知道它想要的东西的内部细节。一个稳定而简单的方法是使用一个 `void *` 参数。

```
void foo(void *arg);
void bar(void *arg);
```

我们可以在 Rust 中用 `c_void` 类型来表示。

```
extern "C" {
    pub fn foo(arg: *mut libc::c_void);
    pub fn bar(arg: *mut libc::c_void);
}
```

这是一种完全有效的处理方式。然而，我们可以做得更好一点。为了解决这个问题，一些 C 库会创建一个 `struct`，其中结构的细节和内存布局是私有的，这提供了某种程度的类型安全。这些结构被称为“不透明的”。下面是一个例子，在 C 语言中：

```
struct Foo; /* Foo 是一个接口，但它的内容不属于公共接口 */
struct Bar;
void foo(struct Foo *arg);
void bar(struct Bar *arg);
```

为了在 Rust 中做到这一点，让我们创建我们自己的不透明类型：

```
#[repr(C)]
pub struct Foo {
    _data: [u8; 0],
    _marker:
        core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
}
#[repr(C)]
pub struct Bar {
    _data: [u8; 0],
    _marker:
        core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
}

extern "C" {
    pub fn foo(arg: *mut Foo);
    pub fn bar(arg: *mut Bar);
}
```

通过包括至少一个私有字段和没有构造函数，我们创建了一个不透明的类型，我们不能在这个模块之外实例化（否则，一个没有字段的结构可以被任何人实例化）。我们也想在 FFI 中使用这个类型，所以我们必须添加 `#[repr(C)]`。该标记确保编译器不会将该结构标记为 `Send`、`Sync`，并且 `Unpin` 也不会应用于该结构（`*mut u8` 不是 `Send` 或者 `Sync`，`PhantomPinned` 也不是 `Unpin`）。

但是因为我们的`Foo`和`Bar`类型不同，我们将在它们两个之间获得类型安全，所以我们不能意外地将`Foo`的指针传递给`bar()`。

注意，使用空枚举作为 FFI 类型是一个非常糟糕的主意。编译器假设空枚举是无法使用的，所以处理`&Empty`类型的值会是意料之外的，并可能导致错误的程序行为（通过触发未定义行为）。

> **注意：** 最简单的方法还是使用“extern 类型”。但它目前（截至 2021 年 10 月）还不稳定，而且还有一些未解决的问题，更多细节请参见RFC 页面和跟踪 Issue。

标准库之下

本节记录了（或将记录）`#![no_std]`开发人员必须处理（即提供）由标准库提供的功能，来构建`#![no_std]`的 Rust 二进制程序。下面是一个（可能是不完整的）此类功能的列表：

- `#[lang = "eh_personality"]`
- `#[lang = "start"]`
- `#[lang = "termination"]`
- `#[panic_implementation]`

#[panic_handler]

`#[panic_handler]` 用于定义 `panic!` 在 `#![no_std]` 程序中的行为。`#[panic_handler]` 必须应用于签名为 `fn(&PanicInfo) -> !` 的函数，并且这样的函数仅能在一个二进制程序/动态链接库的整个依赖图中仅出现一次。`PanicInfo` 的 API 可以在 API docs 中找到。

鉴于 `#![no_std]` 应用程序没有*标准*的输出，并且一些 `#![no_std]` 应用程序，例如嵌入式应用程序，在开发和发布时需要不同的 panic 行为，因此拥有专门的 panic crate，即只包含 `#[panic_handler]` 的 crate 是有帮助的。这样，应用程序可以通过简单地链接到一个不同的 panic crate 来轻松地选择 panic 行为。

下面是一个例子，根据使用开发配置文件（`cargo build`）或使用发布配置文件（`cargo build --release`）编译的应用程序具有不同的恐慌行为：

`panic-semihosting` crate —— 使用 semihosting 将 panic 信息记录到主机 stderr：

```
#![no_std]

use core::fmt::{Write, self};
use core::panic::PanicInfo;

struct HStderr {
    // ..
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    let mut host_stderr = HStderr::new();

    // 输出日志: "panicked at '$reason', src/main.rs:27:4"
    writeln!(host_stderr, "{}", info).ok();

    loop {}
}
```

`panic-halt` crate —— panic 时停止线程；消息被丢弃：

```
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

`app` crate：

```rust
#![no_std]

// dev profile
#[cfg(debug_assertions)]
extern crate panic_semihosting;

// release profile
#[cfg(not(debug_assertions))]
extern crate panic_halt;

fn main() {
    // ..
}
```