Rust死灵书 - Rust高级与非安全程序设计

书栈(BookStack.CN)

目　录

致谢

当前文档 《Rust死灵书 - Rust高级与非安全程序设计》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-06-07。

内容来源: tjxing https://github.com/tjxing/rustonomicon_zh-CN

文档地址: http://www.bookstack.cn/books/rustonomicon_zh-CN

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

原文链接：https://doc.rust-lang.org/nomicon/

Rust死灵书

黑魔法 之 Rust高级与非安全程序设计

注意：本文档讨论了诸多Rust尚未稳定的特性，可能包含一些错误或者过时的信息。

> 我一直期待的程序代码并未出现，取而代之的竟是这令人战栗的黑暗与不可名状的孤独。我看见了！那个让所有人都噤声不语的恐怖事实，那个不可言说的秘密中的秘密—这个精心构建的Rust语言，其实并不像它最初看起来那般坚固不朽。事实上，它竟然是非安全的，它的身躯散发着古怪的气味，滋生着诡异的寄生生物。而我，对这一切束手无策，因为它们都是在编译期发生的。

本书将深入挖掘Rust非安全（unsafe）编程中的一些必要但是又可怕的细节。由于此类问题天然的恐怖，本书散发出的不可描述的恐惧之力，极可能将你的神经彻底撕成千万个绝望的碎片。

如果你仍然期待着拥有一个长期且快乐的Rust编程生涯，那么现在就转身离开，彻底忘掉你曾经见到过这本书—你并不会感到生活有什么缺憾。但是，如果你计划编写非安全代码—或者仅仅是想探究一下这门语言的内在秘密—本书将给你许多有用的信息。

与《Rust程序设计》那本书不同，本书假设你具备一定的基础知识。特别是你应该已经熟练掌握了基本的系统编程和Rust语言。要是还没有的话，请考虑先读这本书。我们并不假设你一定去读了，也会在适当的时候复习一下相关的基础知识。你可以跳过上面那本书直接阅读本书，但要了解我们并不会把每一个知识点都从头讲起。

我们将涉及到异常安全(exception-safety)，指针别名(pointer aliasing)，内存模型(memory model)，编译器和硬件实现的细节，甚至还有一些类型理论(type-theory)。还会大费周章地处理一些原本不该有人关注的边界场景，因为当我们敲出unsafe几个字的时候，这些场景一下子就变得特别重要了。

我们还将花费大量时间讨论程序关注的各种不同的安全保证机制。

原文链接：https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

初识安全与非安全代码

大家都希望可以彻底屏蔽代码底层实现的细节。又有谁愿意关心“一个空的元组占用多少内存”这种破事？可惜的是，有时候这些破事却很重要，我们不得不去关注它。开发人员关注实现细节，大部分情况是为了性能优化。但更主要的是，当我们与硬件、操作系统或者其他语言直接打交道的时候，这些细节往往是正确与否的关键。

当使用某种安全编程语言的过程中遇到了处理底层实现的需求时，程序员通常有三种选择：

- 修改代码让编译器或者运行时环境做相关优化
- 采取某些古怪、繁琐的奇技淫巧以实现功能需求
- 使用另一种可以处理底层细节的语言重写代码

对于最后一个选项，程序员通常会选择C语言。某些系统也只对外暴漏了C的接口。

然而，C在使用中往往过于不安全（虽然有时是出于合理的原因）。尤其是在与其他语言交互的过程中，这种不安全性还会被放大。C和与其交互的语言必须时刻小心地确认对方的行为，以防踩到舞伴的脚趾头。

那么这和Rust有什么关系呢？

嗯……不同于C，Rust是一种安全的编程语言。

但是，和C相同的是，Rust是一种非安全的编程语言。

更准确地说，Rust是一种同时包含安全和非安全特性的编程语言。

Rust可以被看作两种编程语言的结合体：安全Rust和非安全Rust。顾名思义，安全Rust是安全的，而不安全Rust……嗯……是不安全的。不安全Rust允许我们做一些非常不安全的事情——就是那些Rust的创造者们求我们别去做可我们偏要做的事情。

安全Rust是一种真正的安全编程语言。如果你所有的代码都是用安全Rust写的，你永远也无需担心类型安全和内存安全，无需费神处理悬垂指针、释放后引用(use-after-free)，或者其他各种未定义的行为。

标准库也提供了相当多的工具，帮助你用符合安全Rust语言规范的方式创建高性能的应用和库。

不过，也许是时候谈论一下另一种语言了。也许你正在写一种标准库没有覆盖到的底层抽象；也许你正在开发标准库（存粹使用Rust语言）；也许你要做一些类型系统不能理解的事情，还要胡乱摆弄各种字节码。也许，你需要非安全Rust了。

非安全Rust和安全Rust的语法规则完全相同，只不过它允许你做一些另外的不安全的行为（下一节再

告诉你都包括什么）。

分离安全与非安全Rust的价值在于，我们既可以享受像C那样的非安全语言的好处——也就是对底层实现细节的控制，又不用处理C与其他安全语言集成时遇到的种种问题。

不过还是会遇到一些问题。最明显的。我们必须非常了解类型系统的全部默认要求，并在每次与非安全代码交互的时候检查它们。这也是本书的目的：教给你这些要求以及如何处理它们。

原文链接：https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

安全与非安全代码的交互方式

安全与非安全代码之间的关系是什么？它们又如何交互呢？

安全与非安全代码是靠 `unsafe` 关键字分离的，它扮演着两种语言之间接口的角色。这也是我们理直气壮地声称安全Rust是安全的原因：所有的非安全代码都被 `unsafe` 隔离在外。只要你愿意，你甚至可以在代码根部添加 `#![forbid(unsafe_code)]` 以保证你只会写安全的代码。

`unsafe` 关键字有两层含义：声明代码中存在编译器无法检查的安全规范，同时声明开发者会自觉遵守相关规范而不会主动破坏它。

你可以使用关键字 `unsafe` 表明函数和trait的声明中存在不受编译器检查的规范。对于函数， `unsafe` 意味着调用函数的开发者必须查阅函数的文档以确保他们的用法符合函数的安全要求。而对于trait的声明， `unsafe` 意味着实现trait的开发者必须查阅trait的文档以确保trait的实现符合其安全要求。

你可以给一个代码块添加 `unsafe` 关键字，声明块中的所有代码都已经人工检查过符合相关规范。比如，传递给 `slice::get_unchecked` 的索引值都没有越界。

你也可以在实现一个trait时使用 `unsafe` 关键字，声明实现符合trait的安全规范。比如，实现 `Send` 的类型可以绝对安全地转移(move)进另一个线程中。

标准库也有一些非安全函数，包括：

- `slice::get_unchecked` ，可接受不受检查的索引值，也就是存在内存安全机制被破坏的可能
- `mem::transmute` ，将值重新解析成另一种类型，即允许随意绕过类型安全机制的限制（详情参考类型转换）
- 所有指向确定大小类型(sized type)的裸指针都有 `offset` 方法，当传入的偏移量越界时将导致未定义行为(Undefined Behavior)。
- 所有FFI（Foreign Function Interface）函数都是 `unsafe` 的，因为其他的语言可以做各种的操作而Rust编译器无法检查它。

在Rust 1.0中，有两个非安全的trait：

- `Send` 是一个标志trait（即没有任何方法的trait），承诺所有的实现都可以安全地发送（move）到另一个线程。
- `Sync` 也是一个标志trait，承诺线程可以通过共享的引用共享它的实现。

许多Rust标准库其实内部也使用了非安全Rust。这些库的实现方法都经过了严苛的人工检查，所以这些基于非安全Rust实现的安全Rust接口依然可以认为是安全的。

这种代码 隔离的存在说明了安全Rust的一个基本特征:

无论如何，安全**Rust**代码都不能导致未定义行为

可以看出，安全和非安全的Rust之间存在一种不对称的信任关系。安全Rust必须无条件信任非安全Rust，假定所有与之打交道的非安全代码都是正确的。反过来，非安全Rust却要谨慎对待安全Rust的代码。

举个例子，Rust有 `PartialOrd` 和 `Ord` 两个trait，区别在于前者仅仅表示可以被比较的类型，而后者则表示实现了完整顺序(total ordering)的类型（也就是比较的机制更符合直觉）。

`BTreeMap` 只有在键是完整顺序时才能正常工作，所以它要求它的键必须实现 `Ord` 。但是， `BTreeMap` 的内部实现却依赖于非安全Rust代码。因为如果 `Ord` 的实现本身是错误的（尽管代码是安全的）将导致未定义行为，所以 `BTreeMap` 内部的非安全代码必须对那些实际上没有做到完整顺序的 `Ord` 保持足够的鲁棒性——虽然完整顺序本身是我们选择 `Ord` 的唯一理由。

非安全Rust不能简单地信任安全Rust都是正确的。也就是说，如果你传入到 `BTreeMap` 的值不具备完整顺序， `BTreeMap` 的行为将会完全混乱。它仅仅能保证不会产生未定义行为罢了。

有人或许会问，如果 `BTreeMap` 不能因为 `Ord` 是安全的就信任它，那为什么 `BTreeMap` 可以信任其他的安全代码？比如， `BTreeMap` 依赖integer和slice的正确实现。那些不也是安全的代码吗？

区别之一是范围。当 `BTreeMap` 依赖于integer和slice时，它是依赖于某种特定的实现，其收益和风险是可以评估的。依赖integer和slice的风险其实几乎为0，因为如果连它们都是错误的话，那么所有的代码都不可能正确了。而且，它们和 `BTreeMap` 是由相同的开发者维护的，也比较容易配合。

而反过来， `BTreeMap` 的键类型是一个范型。信任它意味着要信任过去、现在和未来的所有的 `Ord` 的实现。这种风险就很高了：来自世界某个角落的路人甲可能在实现 `Ord` 时不小心犯了一个错误，或者他可能觉得代码“差不多没什么问题”就贸然声称它实现了完整排序。 `BTreeMap` 必须时刻准备着面对这些情况。

上述逻辑同样适用于是否应该信任外部传递的闭包。

非安全trait的出现就是为了解决这一类不受限的信任问题。 `BTreeMap` 理论上可以要求键实现一个新的叫做 `UnsafeOrd` 的trait，而不是现在的 `Ord` 。代码可能像这样

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

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

接下来，一个类型要使用 `unsafe` 关键字实现 `UnsafeOrd` ，表明其实现符合trait要求的各种安全规范。这时， `BTreeMap` 的内部就可以合理地信任键的类型对于 `UnsafeOrd` 的实现是正确的。如

果真的出错了，这个锅将由实现非安全trait的开发者来背，与Rust自身的安全机制并不冲突。

一个trait是否应该标志为 `unsafe` 是API设计上的选择。Rust通常会尽量避免这么做，因为它会导致非安全Rust的滥用，这并不是设计者们希望看到的。 `Send` 和 `Sync` 被标识为非安全是因为线程安全性是一个底层特性，非安全代码不太可能有效地检查它，并不像检查 `Ord` 的错误实现那样容易。你也可以根据类似的标准判断是否要把你自己的trait标为 `unsafe` 。如果让安全代码去检查trait实现的正确性不太现实，那么把trait标为 `unsafe` 就是合理的。

顺便说一下， `Send` 和 `Sync` 是会被各种类型自动实现的，只要这种实现可以被证明是安全的。如果一种类型其所有的值的类型都实现了 `Send` ，它本身就会自动实现 `Send` ；如果一种类型其所有的值的类型都实现了 `Sync` ，它本身就会自动实现 `Sync` 。将它们设为 `unsafe` 实际减少了非安全代码的滥用。

安全Rust和非安全Rust各有所长。安全Rust被设计成尽可能地方便易用，而使用非安全Rust不仅要投入更多的精力，还要格外地小心。本书接下来的内容主要讨论那些需要小心的点，以及非安全Rust必须满足的规范。

原文链接：https://doc.rust-lang.org/nomicon/what-unsafe-does.html

非安全Rust能做什么

非安全Rust比安全Rust可以多做的事情只有以下几个：

- 解引用裸指针
- 调用非安全函数（包括C语言函数，编译器内联函数，还有直接内存分配等）
- 实现非安全trait
- 访问或修改可变静态变量

就这些。这些操作被归为非安全的，是因为使用得不正确就会导致可怕的未定义行为。一旦触发了未定义行为，编译器就可以放飞自我，肆意破坏你的程序。切记，一定不能给未定义行为任何的机会。

与C不同，Rust充分限制了可能出现的未定义行为的种类。语言核心只需要防止这几种行为：

- 解引用null指针，悬垂指针，或者未赋值的指针
- 读取未初始化的内存
- 破坏指针混淆规则
- 创建非法的基本类型：
    - 悬垂引用与null引用
    - 空的 `fn` 指针
    - 0和1以外的 `bool` 类型值
    - 未定义的枚举类型的项
    - 在[0x0,0xD&FF]和[0xE000, 0x10FFFF]以外的 `char` 类型值
    - 非utf-8编码的 `str`
- 不谨慎地调用其他语言
- 数据竞争

只有这些。Rust语言自身可以导致未定义行为的操作就只有这些。当然，非安全函数和trait可以声明自己专有的安全规范，要求开发者必须遵守以避免未定义行为。比如，allocator API声明回收一段未分配的内存是未定义行为。

但是，违背这些专有的规范通常也只是间接地触发上面列出的行为。另外，编译器内联函数也可能引入一些规则，一般是针对代码优化的假设条件。比如，Vec和Box使用的内联函数要求传入的指针永远不能为null。

Rust对于一些模糊的操作则通常比较宽容。Rust会认为下列操作是安全的：

- 死锁
- 竞争条件
- 内存泄漏

- 调用析构函数失败
- 整型值溢出
- 终止程序
- 删除产品数据库

当然，有以上行为的程序极有可能就是错误的。Rust提供了一系列的工具减少这种事情的发生，但是完全地杜绝它们其实是不现实的。

原文链接：https://doc.rust-lang.org/nomicon/working-with-unsafe.html

编写非安全代码

Rust通常要求我们明确限制非安全Rust代码的作用域。可是，现实情况其实要更复杂一些。举个例子，看一下下面的代码：

```
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。我们检查了索引值有没有越界。如果没有，就从数组中用不安全的方式取出对应的值。然而，哪怕是这么简单的一个函数，unsafe代码块的范围也不是绝对明确的。想象一下，如果把 `<` 改成 `<=` ：

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

这段程序就有潜在的问题了，但我们其实只修改了安全代码的部分。这是安全机制的一个根本性问题：非本地性。意思是，非安全代码的稳定性其实依赖于另一些“安全”代码的状态。

是否进入非安全代码块，并不受其他部分代码正确性的影响，从这个角度看安全机制是模块化的。比如，是否对一个slice进行不安全索引，不受slice是不是null或者是不是包含未初始化的内存这些事情的影响。但是，由于程序本身是有状态的，非安全操作的结果实际依赖于其他部分的状态，从这个角度看安全机制又是非模块化的。

在处理持久化状态时，非本地性带来的问题就更加明显了。看一下 `Vec` 的一个简单实现：

```rust
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 {
            // 与例子本身无关
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.offset(self.len as isize), elem);
            self.len += 1;
        }
    }
}
```

这段代码很简单，便于审查和修改。现在考虑给它添加一个新的方法：

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

这段代码是100%的安全Rust但是彻底的不稳定。改变容量违反了Vec的不变性（ `cap` 表示分配给Vec的空间大小）。Vec的其他部分并不会保护它，我们只能信任它的值是正确的，因为本来没有修改它的方法。

因为代码逻辑依赖于struct的某个成员的不变性，那段 `unsafe` 的代码不仅仅污染了它所在的函数，它还污染了整个module。一般来说，只有在一个私有的module里非安全代码才可能是真正安全的。

上面的改动其实是可以正常工作的。 `make_room` 方法并不会导致Vec的问题，因为我们没有设置它为public。只有定义这个方法的module可以调用它。同时， `make_room` 直接访问了Vec的私有成员，所以它也只能在Vec所在的module内使用。

这允许我们基于一些复杂的不变性写一些绝对安全的抽象。在考虑安全Rust和非安全Rust的关系时，这一点非常重要。

我们已经了解了非安全代码必须信任一部分安全代码，但是不应该信任所有的安全代码。出于相似的原因，私有成员的限制对于非安全代码很重要：我们不需要无条件信任世界上所有的安全代码并且任由他们搞乱我们的可信任状态。

安全机制万岁！

原文链接：https://doc.rust-lang.org/nomicon/data.html

Rust中的数据表示

底层编程经常需要关注数据布局。它非常重要，而且会影响这门语言的方方面面。所以我们将从Rust中数据的表示方式开始讨论。

原文链接：https://doc.rust-lang.org/nomicon/repr-rust.html

repr(Rust)

首先，每种类型都有一个数据对齐属性(alignment)。一种类型的对齐属性决定了哪些内存地址可以合法地存储该类型的值。如果对齐属性是n，那么它的值的存储地址必须是n的倍数。所以，对齐属性2表示值只能存储在偶数地址里，1表示值可以存储在任何的地方。对齐属性最小为1，并且永远是2的整数次幂。虽然不同平台的行为可能会不同，但大部分情况下基础类型都是按照它的类型大小对齐的。特别的是，在x86平台上 `u64` 和 `f64` 都是按照32位对齐的。

一种类型的大小都是它对齐属性的整数倍，这保证了这种类型的值在数组中的偏移量都是其类型尺寸的整数倍，可以按照偏移量进行索引。需要注意的是，动态尺寸类型的大小和对齐可能无法静态获取。

Rust有如下几种复合类型：

- 结构体（带命名的复合类型 named product types）
- 元组（匿名的复合类型 anonymous product types）
- 数组（同类型数据集合 homogeneous product types）
- 枚举（带命名的标签联合体 named sum types – tagged unions）

如果枚举类型的变量没有关联数据，它就被称之为无成员枚举。

结构体的对齐属性等于它所有成员的对齐属性中最大的那个。Rust会在必要的位置填充空白数据，以保证每一个成员都正确地对齐，同时整个类型的尺寸是对齐属性的整数倍。例如：

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

在对齐属性与类型尺寸相同的平台上，这个结构体会按照32位对齐。整个结构体的类型尺寸是32位的整数倍。它实际会转变成这样：

```
struct A {
    a: u8,
    _pad1: [u8; 3], // 为了对齐b
    b: u32,
    c: u16,
    _pad2: [u8; 2], // 保证整体类型尺寸是4的倍数
                    // （译注：原文就是“4的倍数”，但似乎“32的倍数”才对）
}
```

这里所有的类型都是直接存储在结构体中的，成员类型和结构体之间没有其他的中介。这一点和C是一样的。但是除了数组以外（数组的子类型总是按顺序紧密排列），其他的复合类型的数据分布规则并不一定是固定不变的。对于下面两个结构体定义：

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

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

Rust可以保证A的两个实例的数据布局是完全相同的。但是Rust目前不保证A的实例和B的实例有着一样的数据填充和成员顺序，虽然看起来他们似乎就应该是一样的才对。

对于上面的A和B来说，这一点大概显得莫名其妙。可是当Rust要处理更复杂的数据布局问题时，它就变得很有必要了。

例如，对于这个结构体：

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

现在考虑范型 `Foo<u32, u16>` 和 `Foo<u16, u32>` 。如果Rust按照代码中指定的顺序布局结构体成员，那么它就必须填充数据以符合对齐规则。所以，如果Rust不改变成员顺序的话，他们实际上会变成这样：

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

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

```
    data2: u16,
    _pad2: u16,
}
```

后者显然太浪费内存了。所以，内存优化原则要求不同的范型可以有不同的成员顺序。

枚举把这件事搞得更复杂了。举一个简单的枚举类型为例：

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

它的布局会是这样：

```
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
}
```

这也确实就是一般情况下枚举的布局方式。

但是，在很多情况下这种表达方式并不是效率最高的。一个典型场景就是Rust的“null指针优化”：如果一个枚举类型只包含一个单值变量（比如 `None` ）和一个（级联的）非null指针变量（比如 `&T` ），那么tag其实是不需要的，因为那个单值变量完全可以用null指针来表示。所以， `size_of::<Option<&T>>() == size_of::<&T>()` ，这个比较的结果是正确的。

Rust中的许多类型都包含或者本身就是非null指针，比如 `Box<T>` ， `Vec<T>` ， `String` ， `&T` 以及 `&mut T` 。同样的，你或许也能想到，对于级联的枚举类型，Rust会把多个tag变量合并为一个，因为它们本来就只有几个有限的可能取值。大体说来，枚举类型会运用复杂的算法确定各种级联类型的二进制表达方法。因为这件事很重要，我们把枚举的问题留到后面讨论。

原文链接：https://doc.rust-lang.org/nomicon/exotic-sizes.html

类型中的奇行种

大部分情况下，我们考虑的都是拥有固定的正数尺寸的类型。但是，并非所有类型都是这样。

动态尺寸类型(DST, Dynamically Sized Type)

Rust支持动态尺寸类型，即不能静态获取尺寸或对齐属性的类型。乍一看，这事有点荒谬—Rust必须知道一种类型的大小和对齐方式才能正确地使用它啊！从这一点来看，DST不是一个普通的类型。由于类型大小是未知的，只能通过某种指针来访问它。所以，一个指向DST的指针是一个“胖”指针，它包含指针本身和一些额外的信息（具体请往下看）。

语言提供了两种主要的DST：trait对象和slice。

trait对象表示实现了某种指定trait的类型。具体的类型被擦除了，取而代之的是运行期的一个虚函数表，表中包含了使用这种类型所有必要的信息。这就是trai对象的额外信息：一个指向虚函数表的指针。

slice简单来说是一个连续存储结构的视图—最典型的连续存储结构是数组或 `Vec` 。slice对应的额外信息就是它所指向元素的数量。

结构体可以在最后的位置上保存一个DST，但是这样结构体本身也就变成了一个DST。

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

零尺寸类型(ZST, Zero Sized Type)

Rust实际允许一种类型不占用内存空间：

```
struct Foo; // 没有成员 = 没有尺寸

// 所有成员都没有尺寸 = 没有尺寸
struct Baz {
    foo: Foo,
    qux: (),      // 空元组没有尺寸
    baz: [u8; 0], // 空数组没有尺寸
```

```
}
```

对于其自身来说，ZST显然没有任何用处。但是，和Rust中许多奇怪的布局选项一样，它的作用只在特定的上下文中才能体现：Rust认为所有产生或存储ZST的操作都可以被视为无操作(no-op)。首先，存储它没有什么意义—它又不占用空间。而且这种类型实际上只有一个值，所以加载它的操作可以凭空变一个值出来—而这种操作依然是no-op，因为产生的值不占用空间。

ZST的一个最极端的例子是Set和Map。已经有了类型 `Map<Key, Value>` ，那么要实现 `Set<Key, Value>` 的通常做法是简单封装一个 `Map<Key, UselessJunk>` 。很多语言不得不给UselessJunk分配空间，还要存储、加载它，然后再什么都不做直接丢弃它。编译器很难判断出这些行为实际是不必要的。

但是在Rust里，我们可以直接认为 `Set<Key> = Map<Key, ()>` 。Rust静态地知道所有加载和存储操作都毫无用处，也不会真的分配空间。结果就是，这段范型代码直接就是HashSet的一种实现，不需要HashMap对值做什么多余的处理。

安全代码不用关注ZST，但是非安全代码必须考虑零尺寸类型带来的影响。特别注意，计算指针的偏移量是no-op，标准的内存分配器（Rust默认使用jemalloc）在需要分配空间大小为0时可能返回 `nullptr` ，很难区分究竟是这种情况还是内存不足。

空类型

Rust甚至也支持不能被实例化的类型。这种类型只有类型，而没有对应的值。空类型可以通过指定没有变量的枚举来声明它：

```
enum Void {} // 没有变量 = 空类型
```

空类型比ZST更加少见。一个主要的应用场景是在类型层面声明不可到达性(unreachability)。比如，假设一个API一般需要返回一个Result，但是在某个特殊场景下它是绝对不会出错的。这种情况在类型层面的处理方法是将返回值设为 `Result<T, Void>` 。因为不可能产生一个 `Void` 类型的值，所以返回值不可能是一个 `Err` 。知道了这一点，API的调用者就可以信心十足地使用 `unwrap` 。

原则上来说，Rust可以基于这一点做一些很有意思的分析和优化。比如， `Result<T, Void>` 可以表示成 `T` ，因为实际上不存在返回 `Err` 的情况。下面的代码曾经也可以成功编译：

```
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;
```

但是现在这些把戏已经不让玩了。所以Void唯一的用处就是明确地告诉你某些情况永远不会发生。

关于空类型的最后一个坑，创建指向空类型的裸指针实际上是合法的，但是对它解引用是一个未定义行为，因为这么做没有任何意义。也就是说，你可以使用 `*const Void` 模拟C语言的 `void *` 类型，但是使用 `*const ()` 却不会得到任何东西，因为这个函数对于随机解引用是安全的。

原文链接：https://doc.rust-lang.org/nomicon/other-reprs.html

可选的数据表达方式

Rust允许你选择其他的数据布局策略。

repr(C)

这是最重要的一种 `repr` 。它的目的很简单，就是和C保持一致。数据的顺序、大小、对齐方式都和你在C或C++中见到的一摸一样。所有你需要通过FFI交互的类型都应该有 `repr(C)` ，因为C是程序设计领域的世界语。而且如果我们要在数据布局方面玩一些花活的话，比如把数据重新解析成另一种类型， `repr(C)` 也是很有必要的。

但是，一定不要忘了Rust的那几个奇行种。 `repr(C)` 的存在有双重作用，既为了FFI同时也为了常规的布局控制，所以它可以被应用于那些在FFI中没有意义甚至会产生错误的类型。

- 尽管标准的C语言不支持大小为0的类型，但ZST的尺寸仍然是0。而且它也与C++中的空类型有着明显的不同，C++的空类型还是要占用一个字节的空间的。
- DST的指针（胖指针），元组，和带有成员变量的枚举都是C中没有的，因此也不是FFI安全的。
- 如果 `T` 是一个FFI安全的非空指针，那么 `Option<T>` 可以保证和 `T` 拥有相同的布局和ABI，当然它也会是FFI安全的。这一规则适用于 `&` ， `&mut` 和函数指针等所有非空的指针。
- 在 `repr(C)` 中元组结构体与结构体基本相同，唯一的不同是其成员都是未命名的。
- 对于枚举的处理和 `repr(u*)` 是相同的（见下一节）。选择的类型尺寸等于目标平台上C的应用二进制接口(ABI)的默认枚举尺寸。注意C中枚举的数据布局是确定的，所以这确实是一种“最合理的假设”。不过，当目标C代码编译时加了一些特殊的编译器参数时，这一点可能就不正确了。
- `repr(C)` 和 `repr(u*)` 中无成员的枚举不能被赋值为一个没有对应变量的整数，尽管在C\C++中这是一种合法的行为。构建一个没有对应变量的枚举类型实例属于未定义行为。（对于存在准确匹配的值是允许正常编写和编译的）

repr(u), repr(i)

这两个可以指定无成员枚举的大小。如果枚举变量对应的整数值对于设定的大小越界了，将产生一个编译期错误。你可以手工设置越界的元素为0以避免编译错误，不过要注意Rust是不允许一个枚举中的两个变量拥有相同的值的。

“无成员枚举”的意思是枚举的每一个变量里都不关联数据。不指定 `repr(u*)` 或 `repr(i*)` 的无成员枚举依然是一个Rust的合法原生类型，它们都没有固定的ABI表示方法。给它们指定 `repr` 使其有了固定的类型大小，方便在ABI中使用。

Rust中所有有成员的枚举都没有确定的ABI表示方式（即使关联的数据只是 `PhantomData` 或者零尺

寸类型的数据）。

为枚举显式指定 `repr` 后空指针优化将不再起作用。

这些 `repr` 对于结构体无效。

repr(packed)

`repr(packed)` 强制Rust不填充空数据，各个类型的数据紧密排列。这样有助于提升内存的使用效率，但很可能会导致其他的副作用。

尤其是大部分平台都强烈建议数据对齐。这意味着加载未对齐的数据会很低效（x86），甚至是错误的（一些ARM芯片）。像直接加载或存储打包的(packed)成员变量这种简单的场景，编译器可能可以用shift和mask等方式隐藏对齐问题。但是如果是使用一个打包的变量的引用，编译器很可能没办法避免未对齐加载问题。

在Rust 1.0中这会导致未定义行为

`repr(packed)` 不应该随便使用。只有在你有一些极端的需求的情况下才该用它。

这个repr是 `repr(C)` 和 `repr(Rust)` 的修饰器。

原文链接：https://doc.rust-lang.org/nomicon/ownership.html

所有权和生命周期

所有权是Rust的一个突破性功能。它让Rust可以彻底告别垃圾回收，同时做到内存安全和高效率。在涉及到所有权系统的细节之前，我们先看一下这种设计的目的。

我们假设你认同垃圾回收器（GC）不总是内存管理的最佳方案，在一些场景中需要手工地管理内存。如果你并不这么认为，那么请出门右转使用其他的语言吧。

但是，无论你怎么看待GC，它确实是保证代码安全的大杀器。你永远不需要担心有什么内容会被过早释放（尽管有的时候你已经不想再使用它们了……）。这是C和C++会普遍遇到的问题。看一下这个曾纠缠过每一个使用过非GC语言的人的简单错误：

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

    // 哎呀！我们返回了一个只在函数内部存在的东西的引用
    // 悬垂指针！释放后引用！指针别名！
    // （当然这段代码在Rust中不能编译）
    &s
}
```

这正是Rust的所有权系统要解决的问题。Rust知道 `&s` 生效的作用域，所以可以避免出现逃逸。不过这个例子太简单了，哪怕是C的编译器也可能捕捉到其中的错误。但是当代码量越来越大，指针来自四面八方的不同的函数时，事情就变得复杂了。C编译器最终会败下阵来，无法作出充分的逃逸分析来判断你的代码是否足够健壮。它能做的只有假设你的程序是正确的从而接受它。

这种事情永远不会发生在Rust的世界。Rust需要程序员向编译器保证自己代码的健壮性。

当然，Rust所有权系统要做的事有很多，不是仅仅验证引用不会超出被引用内容作用域这么简单。这是因为保证指针有效的条件比这个要复杂得多。以下面的代码为例。

```
let mut data = vec![1, 2, 3];
// 获得内部引用
let x = &data[0];

// 哎呀！push方法导致data的内部存储位置重新分配了
// 悬垂指针！释放后引用！指针别名！
// （当然这段代码在Rust中不能编译）
data.push(4);
```

```
 9.
10. println!("{}", x);
```

简单地分析作用域不足以防止这个bug，因为 `data` 在我们使用它的范围内确实是一直存在的。但是它在我们引用它的同时发生了变化。这就是为什么Rust要求引用的存在要锁定被引用内容和它的owner。

原文链接：https://doc.rust-lang.org/nomicon/references.html

引用

有两种引用的类型：

- 共享指针：`&`
- 可变指针：`&mut`

它们遵守以下的规则：

- 引用的生命周期不能超过被引用内容
- 可变引用不能存在别名(alias)

就这些。这就是全部的引用模型。

当然，我们可能需要定义一下别名(alias)是什么意思。

```
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
```

很不幸，Rust实际上没有定义别名模型。:scream_cat:

在Rust的开发者从语义层面确定别名的意义之前，我们先在下一章讨论一般意义上的别名指什么，还有它为什么很重要。

原文地址：https://doc.rust-lang.org/nomicon/aliasing.html

别名

首先，有几点重要声明：

- 以下的讨论将采用最广泛意义上的别名的定义。而Rust的定义可能会更加严格，需要考虑到可变性和生命周期。
- 我们假设程序都是单线程且不会中断的，同时也不会去考虑存储器映射之类的问题。除非特别指定，否则Rust默认这些事情不存在。更多的细节请见并发章节。

基于这些，我们给出定义：当变量和指针表示的内存区域有重叠时，它们互为对方的别名。

为什么别名很重要

为什么我们要关注别名？

看下面这个简单的函数。

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

我们可能会这样优化它：

```
fn compute(input: &u32, output: &mut u32) {
    let cached_input = *input; // 将*input放入缓存
    if cached_input > 10 {
        *output = 2; // x > 5 则必然 x > 5, 所以直接加倍并立即退出
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

在Rust中，这种优化是正确的。但对于其他几乎所有的语言，都是有错误的（除非编译器进行全局分析）。这是因为优化方案成立的前提是不存在别名，而绝大多数语言并不会限制这一点。例子中我们需

要特别担心的是传递给 `input` 和 `output` 的参数可能会重合，比如 `comput(&x, &mut x)` 。

对于上面的参数，程序流程会是这样：

```
                    //  input ==  output == 0xabad1dea
                    // *input == *output == 20
if *input > 10 {    // true (*input == 20)
    *output = 1;    // 同时覆盖了 *input，以为他们是一样的
}
*input > 5 {        // false (*input == 1)
    *output *= 2;
}
                    // *input == *output == 1
```

我们优化过的函数的结果是 `*output == 2` ，所以对于这样的输入参数，优化函数是不正确的。

在Rust中我们知道不会出现上面那样的输入参数，因为 `&mut` 不允许存在别名。所以我们可以安全的忽略这种可能性而使用优化方案。对于大多数其他语言，这种输入的可能性是存在的，必须特别的考虑到。

这就是别名分析的重要性：它允许编译器做出一些有用的优化。举几个例子：

- 将值放入缓存变量中，因为可以确定没有指针可以访问变量的内存。
- 省略一些读操作，因为可以确定在上一次读内存之后，内存没有发生变化
- 省略一些写操作，因为可以确定下一次写内存之前，内存不会被读取
- 移动或重排读写操作的顺序，因为可以确定它们并不互相依赖

这些优化也可以进一步证明更大程度的优化的可行性，比如循环向量化、常量替换和不可达代码消除等。

在前面的例子中，我们根据 `&mut u32` 不存在别名的原则证明了 `*output` 不可能影响 `*input` 。这使得我们缓存了 `*input` ，并且省略了一次读操作。

通过缓存读操作的结果，我们知道在 `>10` 的分支中的写操作不会影响执行 `>5` 分支的判断条件，这样我们在 `*input > 10` 的情况下省略了一次读-改-写操作(`*output` 加倍)。

关于别名分析需要记住的一个关键点是，写操作是优化的主要障碍。我们不能随意移动读操作的唯一原因，就是可能存在向相同位置写数据的操作，这种移动会破坏他们之间的顺序关系。

比如，下面这个版本的函数中，我们不需要担心别名问题，因为我们把唯一的一次写 `*output` 的操作放到了函数的最后。这让我们可以随意地改变之前的读 `*input` 操作的顺序：

```
fn compute(input: &u32, output: &mut u32) {
    let mut temp = *output;
```

```
3.      if *input > 10 {
4.          temp = 1;
5.      }
6.      if *input > 5 {
7.          temp *= 2;
8.      }
9.      *output = temp;
10. }
```

我们仍然需要别名分析来证明 `temp` 不是 `input` 的别名，但是这时的证明过程要简单得多：一个本地别量不可能是在它的声明之前就存在的变量的别名。这是所有编程语言共有的一个前提，所以这一版本的函数可以按照与其他语言相同的方式去优化它。

这也就是Rust可能采用的“别名”定义与生命周期和可变性有关的原因：在没有写内存操作存在的情况下，我们实际上不需要关注是否存在别名。

当然，一个完整的别名模型也要考虑到诸如函数调用（可能改变我们不可见的内容）、裸指针（不存在限制别名的需求），以及UnsafeCell（允许被 `&` 引用的内容可变）。

原文链接：https://doc.rust-lang.org/nomicon/lifetimes.html

生命周期

Rust在整个生命周期里强制执行生命周期的规则。生命周期说白了就是作用域的名字。每一个引用以及包含引用的数据结构，都要有一个生命周期来指定它保持有效的作用域。

在函数体内，Rust通常不需要你显式地给生命周期起名字。这是因为在本地上下文里，一般没有必要关注生命周期。Rust知道程序的全部信息，从而可以完美地执行各种操作。它可能会引入许多匿名或者临时的作用域让程序顺利执行。

但是如果你要跨出函数的边界，就需要关心生命周期了。生命周期用这样的符号表示：`'a`，`'static`。为了更清晰地了解生命周期，我们假设我们可以为生命周期打标签，去掉本章所有例子的语法糖。

最开始，我们的示例代码对作用域和生命周期使用了很激进的语法糖特性—甜得像玉米糖浆一样，因为把所有的东西都显式地写出来实在很讨厌。所有的Rust代码都采用比较激进的理论以省略“显而易见”的东西。

一个特别有意思的语法糖是，每一个 `let` 表达式都隐式引入了一个作用域。大多数情况下，这一点并不重要。但是当变量之间互相引用的时候，这就很重要了。举个简单的例子，我们彻底去掉下面这段代码的语法糖：

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

借用检查器通常会尽可能减少生命周期的范围，所以去掉语法糖后的代码大概像这样：

```
// 注意：'a: { 和 &'b x 不是合法的语法
'a: {
    let x: i32 = 0;
    'b: {
        // 生命周期是'b，因为这就足够了
        let y: &'b i32 = &'b x;
        'c: {
            // 'c也一样
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

哇！这样的写法……太可怕了。我们先停下来感谢Rust把这一切都简化掉了。

将引用传递到作用域以外会导致生命周期扩大：

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

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // 必须使用'b，因为引用被传递到了'b的作用域
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

示例：引用超出被引用内容生命周期

好了，让我们再看一遍曾经举过的一个例子：

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

去掉语法糖：

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

函数 `as_str` 的签名里接受了一个带有生命周期的u32类型的引用，并且保证会返回一个生命周期一

样长的str类型的引用。从这个签名我们就已经可以看出问题了。它表示我们必须到那个u32引用的作用域，或者比它还要早的作用域里去找一个str。这就有点不合理了。

接下来我们生成一个字符串 `s` ，然后返回它的引用。我们的函数要求这个引用的有效期不能小于 `'a` ，那是我们给引用指定的生命周期。不幸的是， `s` 是在作用域'b里面定义的。除非'b包含'a这个函数才可能是正确的—而这显然不可能，因为'a必须包含它所调用的函数。这样我们创建了一个生命周期超出被引用内容的引用，这明显违背了之前提到的引用的第一条规则。编译器十分感动然后拒绝了我们。

我们扩展一下这个例子，一边看得更清楚：

```
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
            // as_str的返回值必须引用一个在函数调用前就存在的str
            // 显然事实不是这样的。
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

完蛋了！

当然，这个函数的正确写法应该是这样的。

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

我们必须创建一个值然后连同它的所有权一起返回。除非一个字符串是 `&'a u32` 的成员，我们才能返回 `&'a str` ，显然事情并不是这样的。

（其实我们也可以返回一个字符串的字面量，它是一个全局的变量，可以认为是处于栈的底部。尽管这

样极大限制了函数的使用场合。）

示例：存在可变引用的别名

在看另一个老的例子：

```
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: {
        // 对于这个借用来说，'b已经足够大了
        // （借用只需要在println!中生效即可）
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // 引入一个临时作用域，因为&mut不需要存在更长时间
            Vec::push(&'c mut data, e);
        }
        println!("{}", x);
    }
}
```

这里的问题更加微妙也更有趣。我们希望Rust出于如下的原因拒绝编译这段代码：我们有一个有效的指向 `data` 的内部数据的引用 `x` ，而同时又创建了一个 `data` 的可变引用用于执行 `push` 。也就是说出现了可变引用的别名，这违背了引用的第二条规则。

但是Rust其实并非因为这个原因判断这段代码有问题。Rust不知道 `x` 是 `data` 的子内容的引用，它其实完全不知道Vec的内部是什么样子的。它只知道 `x` 必须在 `'b` 范围内有效，这样才能打印其中的内容。函数 `Index::index` 的签名因此要求传递的 `data` 的引用也必须在 `'b` 的范围内有效。当我们调用 `push` 的时候，Rust发现我们要创建一个 `&'c mut data` 。它知道 `'c` 是包含在 `'b` 以内的，因为 `&'b data` 还存活着，所以它拒绝了这段程序。

我们看到了生命周期系统要比引用的保护措施更加简单粗暴。大多数情况下这也没什么，它让我们不用没完没了地向编译器解释我们的程序。但是这也意味着许多语义上正确的程序会被编译器拒绝，因为生命周期的规则太死板了。

原文链接：https://doc.rust-lang.org/nomicon/lifetime-mismatch.html

生命周期的局限

考虑下面的代码：

```
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();
}
```

你可能觉得它能成功编译。我们调用 `mutate_and_share` ，临时可变地借用 `foo` ，但接下来返回一个共享引用。因为调用 `foo.share()` 时没有可变的引用了，所以我们认为可以正常调用。

但是当我们尝试编译它：

```
<anon>:11:5: 11:8 error: cannot borrow `foo` as immutable because it is also
borrowed as mutable
<anon>:11     foo.share();
              ^~~
<anon>:10:16: 10:19 note: previous borrow of `foo` occurs here; the mutable
borrow prevents subsequent moves, borrows, or modification of `foo` until the
borrow ends
<anon>:10     let loan = foo.mutate_and_share();
                         ^~~
<anon>:12:2: 12:2 note: previous borrow ends here
<anon>:8 fn main() {
<anon>:9     let mut foo = Foo;
<anon>:10     let loan = foo.mutate_and_share();
<anon>:11     foo.share();
<anon>:12 }
          ^
```

发生了什么呢？嗯……我们遇到了和上一章的示例2相同的错误。我们去掉语法糖，会得到这样的代码：

```
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);
            }
        }
    }
}
```

生命周期系统强行把 `&mut foo` 的生命周期扩展到'c，以和 `loan` 的生命周期以及 `mutate_and_share` 的签名匹配。接下来我们调用 `share` ，Rust认为我们在给 `&'c mut foo` 创建别名，于是拒绝了我们。

这段程序显然完全符合引用的语义，但是我们的生命周期系统过于粗糙，无法对它进行正确的分析。

接下来，还有什么普遍的问题吗？关于SEME区域的，大概吧？

原文链接：https://doc.rust-lang.org/nomicon/lifetime-elision.html

省略生命周期

为了让语言的表达方式更人性化，Rust允许函数的签名中省略生命周期。

“生命周期位置”指的是你在类型中可以写生命周期的地方。

```
1. &'a T
2. &'a mut T
3. T<'a>
```

生命周期的位置可以在“输入”也可以在“输出”：

- 对于 `fn` 定义的函数，“输入”指的是函数签名中的参数的类型，而“输出”是结果的类型。所以 `fn foo(s: &str) -> (&str, &str)` 省略了一个在输入位置处的生命周期和两个结果位置的生命周期。注意， `fn` 方法定义中的输入位置不包括 `impl` 头处的生命周期（自然地，对于trait的默认方法，也不包括trait的头的位置）。
- 在未来，应该也可能会省略 `impl` 头位置处的生命周期。

省略的规则如下：

- 每一个在输入位置省略的生命周期都对应一个唯一的生命周期参数。
- 如果只有一个输入的生命周期位置（无论省略还是没省略），那个生命周期会赋给所有省略了的输出生命周期。
- 如果有多个输入生命周期位置，而其中一个是 `&self` 或者 `&mut self` ，那么 `self` 的生命周期会赋给所有省略了的输出生命周期。
- 除了上述两种情况，其他省略生命周期的情况都是错误的。

几个例子：

```
1. fn print(s: &str);                                 // 省略的
2. fn print<'a>(s: &'a str);                          // 完整的
3.
4. fn debug(lvl: usize, s: &str);                     // 省略的
5. fn debug<'a>(lvl: usize, s: &'a str);              // 完整的
6.
7. fn substr(s: &str, until: usize) -> &str;          // 省略的
8. fn substr<'a>(s: &'a str, until: usize) -> &'a str;     // 完整的
9.
10. fn get_str() -> &str;                             // 错误
11.
```

```
12. fn frob(s: &str, t: &str) -> &str;                    // 错误
13.
14. fn get_mut(&mut self) -> &mut T;                      // 省略的
15. fn get_mut<'a>(&'a mut self) -> &'a mut T;            // 完整的
16.
    fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command                  //
17. 省略的
    fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command //
18. 完整的
19.
20. fn new(buf: &mut [u8]) -> BufWriter;                  // 省略的
21. fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // 完整的
```

原文链接：https://doc.rust-lang.org/nomicon/unbounded-lifetimes.html

无界生命周期

非安全代码经常会凭空变出来一些引用和生命周期。这些生命周期都是无界的。最常见的场景是解引用一个裸指针，然后产生一个拥有无界生命周期的引用。这些生命周期根据上下文的要求，想要多大就可以有多大。这其实比简单的设为 `'static` 更加强大。比如 `&'static &'a T` 是无法通过类型检查的，但是无界生命周期可以完美适配 `&'a &'a T` 。不过大多数情况下，这种的无界生命周期会被视为`’static’。

几乎没有哪个引用是 `'static` ，所以这样很可能是错误的。 `transmute` 和 `transmute_copy` 是两种很主要的例外情况。我们应该尽量早的确定无界生命周期的边界，特别是在涉及到函数调用的情况下。

对于一个函数，任何不是从输入那里来的输出生命周期都是无界的。比如：

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

这个函数会产生一个拥有无界生命周期的 `&str` 。最简单的避免无界生命周期的方式就是在函数声明中运用生命周期省略。如果一个输出生命周期被省略了，它必须受限于一个输入生命周期。当然它有可能被赋予了一个错误的生命周期，但是这样通常只会产生一个编译错误，总比允许它破坏内存安全要好。

在函数的内部，限制生命周期范围是极容易出错的。最安全且简单的限制生命周期的方法是将它作为一个有有界生命周期的函数的返回值。但是，如果这个不被接受，引用可以被设置成一个特别的生命周期。不幸的是，我们不可能为函数所有的生命周期命名。（译注：我真的没看懂这一段在说什么……强烈建议看原文，不要看我）

原文链接：https://doc.rust-lang.org/nomicon/hrtb.html

高阶trait边界(HRTB)

Rust的 `Fn` trait是个神奇的存在。比如，我们可以写出这样的代码：

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

impl<F> Clousure<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());
}
```

如果我们像在生命周期那一章里一样地去掉这段代码的语法糖，我们会发现一些问题：

```
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());
    }
}
```

我们究竟应该怎么表示 `F` 的trait边界里的生命周期呢？这里需要一个生命周期，但是在我们进入 `call` 函数之前我们都不知道生命周期的名字！而且，那里的生命周期也是不固定的， `&self` 在那一时间点上是什么生命周期， `call` 就也要是什么生命周期。

这里我们需要借助高阶trait边界（HRTB, Higher-Rank Trait Bounds）的神奇力量了。我们去掉语法糖之后的代码应该是这样的：

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

（其中 `Fn(a, b, c) -> d` 本身就是不确定的 `Fn` trait的语法糖）

`for<'a>` 可以读作“对于 `'a` 的所有可能选择”，基本上表示一个无限的列表，包含所有 `F` 需要满足的trait边界。不过别紧张，除了 `Fn` trait之外我们很少会遇到需要HRTB的场景，而且即使遇到了我们还有一个神奇的语法糖相助。

原文链接：https://doc.rust-lang.org/nomicon/subtyping.html

子类型和变性

子类型是类型之间的一种关系，可以让静态类型语言更加地灵活自由。

理解这一概念最简单的方法就是参考一些支持继承特性的语言。比如说一个Animal类型有一个 `eat()` 方法，Cat类型继承了Animal并且添加了一个 `meow()` 方法。如果没有子类型机制，那么要写一个 `feed(Animal)` 函数，我们就不能给它传递Cat类型的参数，因为Cat并不是一个Animal。但是把Cat传递给需要Animal类型的地方似乎非常的合理。毕竟，Cat就是一个Animal外加一些自己的特性。这些特性完全可以被忽略，不应该妨碍我们在这里使用它！

这就是子类型机制允许我们做的事情。因为Cat是一个Animal外加一些特性，我们就可以说Cat是Animal的子类型。任何需要某种类型的地方，我们都可以传递一个那种类型的子类型。很好！虽然实际情况会稍微复杂和微妙一点，但这种基本的理解足够你应对99%的应用场景了。我们在本章的后面会说明剩下的1%是什么。

尽管Rust没有结构体继承的概念，它却有子类型机制。在Rust中，子类型是针对生命周期存在的。生命周期是代码的作用域，所以我们可以根据它们相互包含的关系判断他们的继承关系。

生命周期的子类型指的是：如果 `'big: 'small` （big包含small，或者big比small长寿），那么 `'big` 就是 `'small` 的子类型。这一点很容易弄错，因为它和我们的直觉是相反的：大的范围是小的范围的子类型。不过如果你对比一下我们举的Animal的例子就清楚了：Cat是一个Animal外加一些独有的东西，而 `'big` 是 `'small` 外加一些独有的东西。

换一个角度想，如果需要一个在 `'small` 内有效的引用，实际指的是至少在 `'small` 中有效的引用。我们并不在乎生命周期是不是完全的一致。从这点上来说，永久生命周期 `'static` 是所有生命周期的子类型。

高阶生命周期也是所有具体生命周期的子类型。这是因为一个随意变化的生命周期比特定的一个生命周期更通用。

（将生命周期类型化是一个过于自由的设计，以至于一些人并不赞同它。但是，把生命周期看做一种类型，这确实简化了我们的分析。）

当然你不能写一个接收 `'a` 类型的值的函数！生命周期只是别的类型的一部分，所以我们需要一些办法来处理它。这里，就要涉及到变性。

变性

变性显得有一点复杂。

变性是类型构造函数与它的参数相关的一个属性。Rust中的类型构造函数是一个带有无界参数的通用类型。比如，`Vec`是一个构造函数，它的参数是`T`，返回值是`vec<T>`。`&`和`&mut`也是构造函数，它们有两个类型：一个生命周期，和一个引用指向的类型。

构造函数F的变性表示了它的输入的子类型如何影响它输出的子类型。Rust中有三种变性：

- 如果当`T`是`U`的子类型时，`F<T>`也是`F<U>`的子类型，则F对于`T`是协变的
- 如果当`T`是`U`的子类型时，`F<U>`是`F<T>`的子类型，则F对于`T`是逆变的
- 其他情况（即子类型之间没有关系），则F对于`T`是不变的

注意，在Rust中协变性远比逆变性要普遍和重要。逆变性的存在几乎可以忽略。

一些重要的变性（下文会详细描述）：

- `&'a T`对于`'a`和`T`是协变的
- `&'a mut T`对于`'a`是协变的，对于`T`是不变的
- `fn(T) -> U`对于`T`是逆变的，对于`U`是协变的
- `Box`，`Vec`以及所有的集合类对于它们保存的类型都是协变的
- `UnsafeCell<T>`，`Cell<T>`，`RefCell<T>`，`Mutex<T>`和其他的内部可变类型对于`T`都是不变的

我们举几个例子说明这些变性为什么是正确且必要的。

在介绍子类型的时候，其实已经包括了为什么`&'a T`对`'a`是协变的。当需要一个较短的生命周期时，我们需要能够传递一个更长的生命周期。

类似的理由也可以解释为什么它对于`T`是协变的：给一个要求`&&'a str`的地方传递`&&'static T`是很合理的。这种间接的引用并不影响对生命周期长度的要求。

但是同样的逻辑并不适用于`&mut`。下面的代码演示了为什么`&mut`对于T是不变的：

```
fn overwrite<T: Copy>(intput: &mut T, new: &mut T) {
    *input = *new;
}

fn main() {
    let mut forever_str: &'static str = "hello";
    {
        let string = String::from("world");
        overwrite(&mut forever_str, &mut &*string);
    }
    // 不好！在打印被释放的内存数据
    println!("{}", forever_str);
}
```

`overwrite` 的签名显然是合法的，它接受两个相同类型的可变引用，然后用一个覆盖另外一个。

但是，如果 `&mut T` 对于T是协变的， `&mut &'static str` 将会是 `&mut &'a str` 的子类型，这是因为 `&'static str` 是 `&'a str` 的子类型。这时 `forever_str` 的生命周期就缩减到和 `string` 一样短， `overwrite` 也可以被正常调用。接下来 `string` 被释放，等到打印的时候 `forever_str` 实际指向了一块释放后的内存空间！所以 `&mut` 必须是不变的。

这是变性的一个基本原则：如果生命周期较短的内容有可能存储在生命周期更长的变量里，这时必须要求变性是不变的。

更一般的解释是，子类型和变性可用的前提是我们可以安全地忘掉类型的细节。但对于可变引用，总有一些地方（被引用的原始值）记着类型的信息并且假设它们不会改变。如果我们改变了这些信息，原始值的位置就可能出现异常。

但是， `&'a mut T` 对于 `'a` 却是协变的。 `'a` 和 `T` 最关键的区别是 `'a` 是引用自身的属性，而T则是引用借用的。如果改变了T的类型，T的原始值依然记着它的类型。可如果改变的是生命周期的类型，只有引用自己知道这一变化，因此这是安全的。换句话说， `&'a mut T` 拥有 `'a` ，但是仅仅借用T。

`Box` 和 `Vex` 的情况就很有趣了，他们是协变的，可是你可以在里面存储值。Rust的类型系统允许它们比其他的类型更聪明。为了理解为什么拥有数据所有权的容器类型对于它们的内容是协变的，我们需要考虑两种可能发生子类型变化的方式：通过值和通过引用。

如果子类型通过值发生变化，原有的记录类型信息的位置会被移除，也意味着容器再也不能使用原有的值了。所以我们也就不用担心有其他的地方记录着类型的信息。换言之，通过值使用子类型的特性会彻底销毁原有类型的信息。例如，这段代码可以编译并正常运行：

```
1.  fn get_box<'a>(str: &'a str) -> Box<&'a str> {
2.      // 字符串字面量是&'static str类型，但是我们完全可以“忘掉”这一点，
3.      // 就让调用者认为这个字符串的生命周期只有这么短
4.      Box::new("hello")
5.  }
```

如果子类型通过引用发生变化，那么容器类会以 `&mut Vec<T>` 类型传递。可是 `&mut` 对于它引用的值是不变的，所以 `&mut Vec<T>` 对于 `T` 实际也是不变的。那么 `Vec<T>` 对于 `T` 协变这件事在引用的情况下就完全不重要了。

不过， `Box` 和 `Vec` 的协变性在不可变引用的情况下依然有用。所以你可以将 `&Vec<&'static str>` 传递给需要 `&Vec<&'a str>` 的地方。

cell类型的不变性可以这样理解：对于cell来说 `&` 就是 `&mut` ，因为你可以通过 `&` 储存值。所以cell必须是不变的，以避免生命周期缩短的问题。

`fn` 是最怪异的，因为它具有混合变性，而且它也是唯一用到了逆变性的地方。下面的函数签名展示了为什么 `fn(T) -> U` 对于T是逆变的：

```
1.  // 'a来自父作用域
2.  fn foo(&'a str) -> usize;
```

这个签名表明函数可以接受任何生命周期不小于 `'a` 的 `&str` 。如果函数对于 `&'a str` 是协变的，那么这个函数

```
1.  fn foo(&'static str) -> usize;
```

就是它的子类型并且可以使用。但是，这个函数的要求其实更严格，它只能接受 `&'static str` ，不能接受其他类型。给它传递一个 `&'a str` 是错误的，因为我们不能假设传递给它的值会永远存在。所以，函数对于它的参数类型肯定不能使协变的。

如果我们反过来应用逆变性，就万事大吉了！需要一个函数来处理永远存在的字符串，而我们提供了一个处理有限生命周期字符串的函数，这也是完全合理的。所以，

```
1.  fn foo(&'a str) -> usize;
```

可以被传递给需要

```
1.  fn foo(&'static str) -> usize;
```

的地方。

那 `fn(T) -> U` 对于U怎么又是协变的了呢？看看下面这个函数签名：

```
1.  // 'a来自父作用域
2.  fn foo(usize) -> &'a str;
```

这个函数声明它将返回一个生命周期长于 `'a` 的引用。那么下面这个函数

```
1.  fn foot(usize) -> &'static str;
```

用在这里是完全可以的，因为它的的确确返回了一个生命周期长于 `'a` 的引用。所以函数对于它的返回值是协变的。

`*const` 和 `&` 有着完全一样的语义，所以变性也是一样的。 `*mut` 正相反，它可以解引用出一个 `&mut` ，所以和cell一样，它也是不变的。

以上规则都是针对标准库提供的类型，那么自己定义的类型又如何确定变性呢？简单点说，结构体会继承它的成员的变性。如果结构体 `Foo` 有一个成员 `a` ，它使用了结构体的泛型参数 `A` ，那么 `Foo` 对于 `A` 的变性就等于 `a` 对于 `A` 的变性。可如果 `A` 被用在了多个成员中：

- 如果所有用到A的成员都是协变的，那么Foo对于A就是协变的
- 如果所有用到A的成员都是逆变的，那么Foo对于A也是逆变的
- 其他的情况，Foo对于A是不变的

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

struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H, In, Out, Mixed> {
    a: &'a A,      // 对于'a和A协变
    b: &'b mut B, // 对于'b协变，对于B不变

    c: *const C,  // 对于C协变
    d: *mut D,    // 对于D不变

    e: E,         // 对于E协变
    f: Vec<F>,    // 对于F协变
    g: Cell<G>,   // 对于G不变

    h1: H,        // 对于H本该是可变的，但是……
    h2: Cell<H>,  // 其实对H是不变的，发生变性冲突的都是不变的

    i: fn(In) -> Out,       // 对于In逆变，对于Out协变

    k1: fn(Mixed) -> usize, // 对于Mix本该是逆变的，但是……
    k2: Mixed,              // 其实对Mixed是不变的，发生变性冲突的都是不变的
}
```

原文链接：https://doc.rust-lang.org/nomicon/dropck.html

Drop检查

我们已经知道生命周期给我们提供了一些很简单的规则，以保证我们永远不会读取悬垂引用。但是，到目前为止我们提到生命周期的长短时，指的都是非严格的关系。也就是说，当我们写 `'a: 'b` 的时候， `'a` 其实也可以和 `'b` 一样长。乍一看，这一点没什么意义。本来也不会有两个东西被同时销毁的，不是吗？我们去掉下面的 `let` 表达式的语法糖看看：

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

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

每一个都创建了自己的作用域，可以很清楚地看出来一个在另一个之前被销毁。但是，如果是下面这样的呢？

```
let (x, y) = (vec![], vec![]);
```

有哪一个比另一个存活更长吗？答案是，没有，没有哪个严格地比另一个长。当然，x和y中肯定有一个比另一个先销毁，但是销毁的顺序是不确定的。并非只有元组是这样，复合结构体从Rust 1.0开始就不会保证它们的销毁顺序。

我们已经清楚了元组和结构体这种内置复合类型的行为了。那么Vec这样的类型又是什么样的呢？Vec必须通过标准库代码手动销毁它的元素。通常来说，所有实现了Drop的类型在临死前都有一次回光返照的机会。所以，对于实现了Drop的类型，编译器没有充分的理由判断它们的内容的实际销毁顺序。

可是我们为什么要关心这个？因为如果系统不够小心，就可能搞出来悬垂指针。考虑下面这个简单的程序：

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

fn main() {
    let (inspector, days);
    days = Box::new(1);
```

```
    inspector = Inspector(&days);
}
```

这段程序是正确且可以正常编译的。 `days` 并不严格地比 `inspector` 存活得更长，但这没什么关系。只要 `inspector` 还存活着， `days` 就一定也活着。

可如果我们添加一个析构函数，程序就不能编译了！

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

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("再过{}天我就退休了！", self.0);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
    // 如果days碰巧先被销毁了
    // 那么当销毁Inspector的时候，它会读取被释放的内存
}
```

```
error: `days` does not live long enough
  --> <anon>:15:1
   |
12 |     inspector = Inspector(&days);
   |                           ---- borrow occurs here
...
15 | }
   | ^ `days` dropped here while still borrowed
   |
   = note: values in a scope are dropped in the opposite order they are created

error: aborting due to previous error
```

实现 `Drop` 使得 `Inspector` 可以在销毁前执行任意的代码。一些通常认为和它生命周期一样长的类型可能实际上比它先销毁，而这会有潜在的问题。

有意思的是，只有泛型需要考虑这个问题。如果不是泛型的话，那么唯一可用的生命周期就是 `'static` ，而它确确实实会永远存在。这也就是这一问题被称之为“安全泛型销毁”的原因。安全

泛型销毁是通过drop检查器执行的。我们还未涉及到drop检查器判断类型是否可用的细节，但其实我们之前已经讨论了这个问题的最主要规则：

一个安全地实现**Drop**的类型，它的泛型参数生命周期必须严格地长于它本身

遵守这一规则（大部分情况下）是满足借用检查器要求的必要条件，同时是满足安全要求的充分非必要条件。也就是说，如果类型遵守上述规则，它就一定可以安全地drop。

之所以并不总是满足借用检查器要求的必要条件，是因为有时类型借用了数据但是在Drop的实现里没有访问这些数据。

例如，上面的 `Inspector` 的这一变体就不会访问借用的数据：

```
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);
    }
}

fn main() {
    let (inspector, days);
    days = Box::nex(1);
    inspector = Inspector(&days, "gadget);
    // 假设days碰巧先被销毁。
    // 可当Inspector被销毁时，它的析构函数也不会访问借用的days。
}
```

同样，这个变体也不会访问借用的数据：

```
use std::fmt;

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

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

fn main() {
    let (inspector, days): (Inspector<&u8>, Box<u8>);
```

```
    days = Box::new(1);
    inspector = Inspector(&days, "gadget");
    // 假设days碰巧先被销毁。
    // 可当Inspector被销毁时，它的析构函数也不会访问借用的days。
}
```

但是，借用检查器在分析 `main` 函数的时候会拒绝上面两段代码，并指出 `days` 存活得不够长。

这是因为，当借用检查分析 `main` 函数的时候，它并不知道每个 `Inspector` 的 `Drop` 实现的内部细节。它只知道inspector的析构函数有访问借用数据的可能。

因此，drop检查器强制要求一个值借用的所有数据的生命周期必须严格长于值本身。

留一个后门

上面的类型检查的规则在未来有可能会松动。

当前的分析方法是很保守甚至苛刻的，它强制要求一个值借用的数据必须比值本身长寿，以保证绝对的安全。

未来的版本中，分析过程会更加精细，以减少安全的代码被拒绝的情况。比如上面的两个 `Inspector` ，它们知道在销毁过程中不应该被检查。

同时，有一个还未稳定的属性可以用来（非安全地）声明类型的析构函数保证不会访问过期的数据，即使类型的签名显示有这种可能存在。

这个属性是 `my_dangle` ，在RFC 1327中被引入。我们可以这样将其放在上面的 `Inspector` 例子里：

```
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);
    }
}
```

使用这个属性要求 `Drop` 的实现被标为 `unsafe` ，因为编译器将不会检查有没有过期的数据（比如 `self.0` ）被访问。

这个属性可以赋给任意数量的生命周期和类型参数。下面这个例子里，我们声明我们不会访问有生命周期 `'b` 的引用背后的数据，而类型 `T` 也只会被用来转移或销毁。但是我们没有为 `'a` 和 `U` 添加属性，因为我们确实会用到这个生命周期和类型：

```
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);
    }
}
```

上面的例子中，哪些数据不会被用到是一目了然的。但是，有时候这些泛型参数会被间接地访问。间接访问的形式包括：

- 使用回调函数
- 通过调用trait方法

（在日后的版本里可能增加其他间接访问的途径。）

以下是使用回调的例子：

```
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的类型是&'a _, self.2的调用可能访问借用的数据
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 (self.2)(&self.0), self.1);
    }
}
```

这是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 drop) {
        // 下面有一个对<T as Display>::fmt的隐藏调用,
        // 当T的类型是&'a _时, 可能访问借用数据
        println!("Inspector({}, {}) unwittingly inspects expired data.",
```

```
10.                     self.0, self.1);
11.     }
12. }
```

当然，这些访问可以进一步地被隐藏在其他的析构函数调用的方法里，而不仅是直接写在函数中。

上面的几个例子里， `&'a u8` 都在析构函数里被访问了。如果给它添加 `#[may_dangle]` 属性，这些类型很可能会产生借用检查器无法捕捉的错误，引发不可预料的灾难。所以最好能避免使用这个属性。

drop检查的故事讲完了吗？

我们发现，在写非安全代码时，其实并不用关心是否满足drop检查器的要求。不过有一个特殊的场景是例外的，我们将在下一章讲到它。

原文地址：https://doc.rust-lang.org/nomicon/phantom-data.html

PhantomData（幽灵数据）

在编写非安全代码时，我们常常遇见这种情况：类型或生命周期逻辑上与一个结构体关联起来了，但是却不属于结构体的任何一个成员。这种情况对于生命周期尤为常见。比如，`&'a [T]`的`Iter`大概是这么定义的：

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

但是，因为`'a`没有在结构体内被使用，它是无界的。由于一些历史原因，无界生命周期和类型禁止出现在结构体定义中。所以我们必须想办法在结构体内用到这些类型，这也是正确的变性检查和drop检查的必要条件。

我们使用一个特殊的标志类型`PhantomData`做到这一点。`PhantomData`不消耗存储空间，它只是模拟了某种类型的数据，以方便静态分析。这么做比显式地告诉类型系统你需要的变性更不容易出错，而且还能提供drop检查需要的信息。

`Iter`逻辑上包含一系列`&'a T`，所以我们用`PhantomData`这样去模拟它：

```
use std::marker;

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

就是这样，生命周期变得有界了，你的迭代器对于`'a`和`T`也可变了。一切尽如人意。

另一个重要的例子是`Vec`，它差不多是这么定义的：

```
struct Vec<T> {
    data: *const T, // *const是可变的！
    len: usize,
    cap: usize,
}
```

和之前的例子不同，这个定义已经满足我们的各种要求了。 `Vec` 的每一个泛型参数都被至少一个成员使用过了。非常完美！

你高兴的太早了。

Drop检查器会判断 `Vec<T>` 并不拥有T类型的值，然后它认为无需担心Vec在析构函数里能不能安全地销毁T，再然后它会允许人们创建不安全的Vec析构函数。

为了让drop检查器知道我们确实拥有T类型的值，也就是需要在销毁Vec的时候同时销毁T，我们需要添加一个额外的PhantomData：

```
use std::marker:

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

让裸指针拥有数据是一个很普遍的设计，以至于标准库为它自己创造了一个叫 `Unique<T>` 的组件，它可以：

- 封装一个 `*const T` 处理变性
- 包含一个PhantomData
- 自动实现 `Send` / `Sync` ，模拟和包含T时一样的行为
- 将指针标记为 `NonZero` 以便空指针优化

`PhantomData` 模式表

下表展示了各种牛X闪闪的 `PhantomData` 用法：

Phantom 类型	`'a`	`'T`
`PhantomData<T>`	-	协变（可触发drop检查）
`PhantomData<&'a T>`	协变	协变
`PhantomData<&'a mut T>`	协变	不变
`PhantomData<*const T>`	-	协变
`PhantomData<*mut T>`	-	不变
`PhantomData<fn(T)>`	-	逆变(*)
`PhantomData<fn() -> T`	-	协变
`PhantomData<fn(T) -> T>`	-	不变

`PhantomData<Cell<&'a ()>>`	不变	-

(*)如果发生变性的冲突，这个是不变的

原文链接：https://doc.rust-lang.org/nomicon/borrow-splitting.html

分解借用

可变引用的Mutex属性在处理复合类型时能力非常有限。借用检查器只能理解一些简单的东西，而且极易失败。他对结构体还算是充分了解，知道结构体的成员可能被分别借用。所以这段代码现在可以正常工作：

```
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);
```

但是，借用检查器对于数组和slice的理解却是一团浆糊，所以这段代码无法通过检查：

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

```
<anon>:4:14: 4:18 error: cannot borrow `x[..]` as mutable more than once at a
time
<anon>:4 let b = &mut x[1];
                      ^~~~
<anon>:3:14: 3:18 note: previous borrow of `x[..]` occurs here; the mutable
borrow prevents subsequent moves, borrows, or modification of `x[..]` until the
borrow ends
<anon>:3 let a = &mut x[0];
                      ^~~~
<anon>:6:2: 6:2 note: previous borrow ends here
<anon>:1 fn main() {
```

```
 9.  <anon>:2 let mut x = [1, 2, 3];
10.  <anon>:3 let a = &mut x[0];
11.  <anon>:4 let b = &mut x[1];
12.  <anon>:5 println!("{} {}", a, b);
13.  <anon>:6 }
14.           ^
15.  error: aborting due to 2 previous errors
```

借用检查器连这个简单的场景都理解不了，那它更不可能理解一些通用容器类型了，比如说树，尤其是出现不同的键对应相同的值的时候。

为了能“教育”借用检查器我们的所作所为是正确的，我们还是要使用非安全代码。比如，可变slice暴露了一个 `split_at_mut` 的方法，它接收一个slice然后返回两个可变slice。一个包括索引值左边所有的值，另一个包含右边所有的值。我们知道这个方法是安全的，因为两个slice没有重叠部分，也就不会出现别名问题。但是它的实现还是要涉及到非安全的内容：

```
1.  fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
2.      let len = self.len();
3.      let ptr = self.as_mut_ptr();
4.      assert!(mid <= len);
5.      unsafe {
6.          (from_raw_parts_mut(ptr, mid)),
7.           from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
8.      }
9.  }
```

这有一点难懂。为了避免两个 `&mut` 指向相同的值，我们通过裸指针显式创建了两个全新的slice。

不过迭代器产生可变引用的方法更加难懂。迭代器trait的定义如下：

```
1.  trait Iterator {
2.      typr Item;
3.
4.      fn next(&mut self) -> Option<Self::Item>;
5.  }
```

这份定义里， `Self::Item` 与 `slef` 没有直接关系。也就是说我们可以连续调用 `next` 很多次，并且同时保存着所有的结果。对于值的迭代器这么做完全可以，完全符合语义。对于共享引用这么做也没什么问题，因为允许任意过个共享引用指向同一个值（当然迭代器本身需要是独立于被共享内容的对象）。

但是可变引用就麻烦了。乍一看，可变引用完全不适用这个API，因为那会产生多个指向相同对象的可

变引用。

可实际上它能够正常工作，这是因为迭代器是一个一次性对象。 `IterMut` 生成的东西最多只会生成一次，所以实际上我们没有生成多个指向相同数据的可变指针。

更不可思议的是，可变迭代器对于许多类型的实现甚至不需要非安全代码！

例如，下面是单向列表的代码：

```
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
        })
    }
}
```

这是可变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::replace(&mut self.0, &mut []);
        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::replace(&mut self.0, &mut []);
        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)
    }
}
```

还有二叉树：

```
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>,
}
```

```
14.
15. struct NodeIterMut<'a, T: 'a> {
16.     elem: Option<&'a mut T>,
17.     left: Option<&'a mut Node<T>>,
18.     right: Option<&'a mut Node<T>>,
19. }
20.
21. enum State<'a, T: 'a> {
22.     Elem(&'a mut T),
23.     Node(&'a mut Node<T>),
24. }
25.
26. pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);
27.
28. impl<T> Tree<T> {
29.     pub fn iter_mut(&mut self) -> IterMut<T> {
30.         let mut deque = VecDeque::new();
31.         self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
32.         IterMut(deque)
33.     }
34. }
35.
36. impl<T> Node<T> {
37.     pub fn iter_mut(&mut self) -> NodeIterMut<T> {
38.         NodeIterMut {
39.             elem: Some(&mut self.elem),
40.             left: self.left.as_mut().map(|node| &mut **node),
41.             right: self.right.as_mut().map(|node| &mut **node),
42.         }
43.     }
44. }
45.
46.
47. impl<'a, T> Iterator for NodeIterMut<'a, T> {
48.     type Item = State<'a, T>;
49.
50.     fn next(&mut self) -> Option<Self::Item> {
51.         match self.left.take() {
52.             Some(node) => Some(State::Node(node)),
53.             None => match self.elem.take() {
54.                 Some(elem) => Some(State::Elem(elem)),
55.                 None => match self.right.take() {
```

```
56.                     Some(node) => Some(State::Node(node)),
57.                     None => None,
58.                 }
59.             }
60.         }
61.     }
62. }
63.
64. impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
65.     fn next_back(&mut self) -> Option<Self::Item> {
66.         match self.right.take() {
67.             Some(node) => Some(State::Node(node)),
68.             None => match self.elem.take() {
69.                 Some(elem) => Some(State::Elem(elem)),
70.                 None => match self.left.take() {
71.                     Some(node) => Some(State::Node(node)),
72.                     None => None,
73.                 }
74.             }
75.         }
76.     }
77. }
78.
79. impl<'a, T> Iterator for IterMut<'a, T> {
80.     type Item = &'a mut T;
81.     fn next(&mut self) -> Option<Self::Item> {
82.         loop {
83.             match self.0.front_mut().and_then(|node_it| node_it.next()) {
84.                 Some(State::Elem(elem)) => return Some(elem),
85.                 Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
86.                 None => if let None = self.0.pop_front() { return None },
87.             }
88.         }
89.     }
90. }
91.
92. impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
93.     fn next_back(&mut self) -> Option<Self::Item> {
94.         loop {
95.             match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
96.                 Some(State::Elem(elem)) => return Some(elem),
97.                 Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
```

```
98.                 None => if let None = self.0.pop_back() { return None },
99.             }
100.         }
101.     }
102. }
```

所有这些都是完全安全而且能稳定运行的！这已经超出了我们之前看过的简单结构体的例子：Rust能够理解你把一个可变引用安全地分解为多个部分。接下来我们可以通过Option永久地访问这个引用（或者像对于slice那样，替换为一个空的slice）。

原文链接：https://doc.rust-lang.org/nomicon/conversions.html

类型转换

到目前为止，我们的程序还是一堆散乱的字节，而类型系统拍马赶到教给我们如何正确使用这些字节。将字节翻译成有意义的类型，这件事有两个主要问题：将完全相同的一组字节解析成不同的类型，以及让不同的字节在不同的类型中有相同的含义。因为Rust喜欢将一些重要的属性附加在类型上，这些问题就变得尤其普遍。所以Rust也提供了许多方法解决这一问题。

我们先来看看安全Rust如何重新解析值。最普通的方法是将值的各个组成部分拆分出来，再用它们重新构建一个新的类型的值。例如

```
1.  struct Foo {
2.      x: u32,
3.      y: u16,
4.  }
5.
6.  struct Bar {
7.      a: u32,
8.      b: u16,
9.  }
10.
11. fn reinterpret(foo: Foo) -> Bar {
12.     let Foo { x, y } = foo;
13.     Bar { a: x, b: y }
14. }
```

可是这种方法显然很烦人。对于一般的类型转换场景，Rust提供了更加人性化的选择。

原文链接：https://doc.rust-lang.org/nomicon/coercions.html

强制类型转换

在一些特定场景中，类型会被隐式地强制转换。这种转换通常导致类型被“弱化”，主要针对指针和生命周期。主要目的是让Rust适用于更多的场景，并且基本上是无害的。

强制转换包括下面几种：

如下几种类型之间允许进行强制转换：

- 传递性：当 `T_1` 可以强制转换为 `T_2` 且 `T_2` 可以强制转换为 `T_3` 时， `T_1` 就可以强制转换为 `T_3`
- 指针弱化：
  - `&mut T` 转换为 `&T`
  - `*mut T` 转换为 `*const T`
  - `&T` 转换为 `*const T`
  - `&mut T` 转换为 `*mut T`
- Unsize：如果 `T` 实现了 `CoerceUnsized<U>` ，那么 `T` 可以强制转换为 `U`
- 强制解引用：如果 `T` 可以解引用为 `U` （比如 `T: Deref<Target=U>` ），那么 `&T` 类型的表达式 `&x` 可以强制转换为 `&U` 类型的 `&*x`

所有的指针类型（包括Box和Rc这些智能指针）都实现了 `CoerceUnsized<Pointer<U>> for Pointer<T> where T: Unsize<U>` 。Unsize只能被自动实现，并且实现如下转换方式：

- `[T; n]` => `[T]`
- `T` => `Trait` ，其中 `T: Trait`
- `Foo<..., T, ...>` => Foo<…, U, …>`，其中
  - `T: Unsize<U>`
  - `Foo` 是一个结构体
  - 只有 `Foo` 的最后一个成员是和 `T` 有关的类型
  - 其他成员的类型与 `T` 无关
  - 如果最后一个成员的类型是 `Bar<T>` ，那么必须有 `Bar<T>: Unsize<Bar<U>>`

强制转换会在“强制转换位置”处发生。每一个显式声明了类型的位置都会引起到该类型的强制转换。但如果必须进行类型推断，则不会发生类型转换。表达式 `e` 到类型 `U` 的强制转换位置包括：

- let表达式，静态变量或者常量： `let x: U = e`
- 函数的参数： `takes_a_U(e)`
- 函数返回值： `fn foo() -> U {e}`
- 结构体初始化： `Foo { some_u: e }`
- 数组初始化： `let x: [U; 10] = [e, ...]`

- 元组初始化：`let x: (U, ..) = (e, ..)`
- 代码块中的最后一个表达式：`let x: U = { ..; e }`

注意，在匹配trait的时候不会发生强制类型转换（receiver除外，具体见下）。也就是说，如果为 `U` 实现了一个trait， `T` 可以强制转换为 `U` ，并不能认为 `T` 也实现了这个trait。例如，下面的代码无法通过类型检查，虽然 `t` 可以强制转换为 `&T` ，而且有一个 `&T` 的trait实现。

```
trait Trait {}

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

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

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

```
<anon>:10:5: 10:8 error: the trait bound `&mut i32 : Trait` is not satisfied
[E0277]
<anon>:10     foo(t);
              ^~~
```

原文链接：https://doc.rust-lang.org/nomicon/dot-operator.html

点操作符

点操作符可以做到许多神奇的类型转换任务，比如自动引用，自动解引用，还有级联类型匹配后的强制类型转换。

TODO：从这里偷一些信息 http://stackoverflow.com/questions/28519997/what-are-rusts-exact-auto-dereferencing-rules/28552082#28552082

原文链接：https://doc.rust-lang.org/nomicon/casts.html

显式类型转换

显式类型转换是强制类型转换的超集：所有的强制类型转换都可以通过显式转换的方式主动触发。但有一些场景只适用于显式转换。强制类型转换很普遍而且通常无害，但是显式类型转换是一种“真正的转换“，它的应用就很稀少了，而且有潜在的危险。因此，显式转换必须通过关键字 `as` 主动地触发： `expr as Type` 。

真正的转换一般是针对裸指针和基本数字类型的。虽然说过它们存在风险，但是在运行期却是很稳定的。如果类型转换操作触发了一些奇怪的边界场景，Rust并不会给出任何提示。转换仍然会被认为是成功的。这就要求显式类型转换必须在类型层面是合法的，否则会在编译期被拒绝。比如， `7u8 as bool` 不会编译成功。

也就是说，显式类型转换不属于非安全(unsafe)行为，因为仅凭转换操作是不会违背内存安全性的。比如，将整型转换为裸指针很容易导致可怕的后果。但是，创建一个指针这个行为本身是安全的，而真正使用裸指针的操作则必须被标为 `unsafe` 。

以下是所有显式类型转换的情况。简单起见，我们用 `*` 表示 `*const` 或者 `*mut` ，用 `integer` 表示任意整数基本类型：

- `*T as *U` ，其中 `T, U: Sized`
- `*T as *U` ，TODO: 明确unsize的情况
- `*T as integer`
- `integer as *T`
- `number as number`
- 无成员枚举 `as integer`
- `bool as integer`
- `char as integer`
- `u8 as char`
- `&[T; n] as *const T`
- `fn as *T` ，其中 `T: Sized`
- `fn as integer`

注意，裸slice转换后长度会改变，比如 `*const [u16] as *const [u8]` 创建的slice只包含原本一半的内存。

显示类型转换不是可传递的，也就是说，即使 `e as U1 as U2` 是合法的表达式，也不能认为 `e as U2` 就一定是合法的。

对于数字类型的转换，如下几点需要注意：

- 相同大小的整型互相转换（比如i32->u32）是一个no-op
- 大尺寸的整型转换为小尺寸的整型（比如u32->u8）会被截断
- 小尺寸的整型转换为大尺寸的整型（比如u8->u32）
    - 如果源类型是无符号的，将会补零
    - 如果源类型是有符号的，将会有符号补零
- 浮点类型转换为整型会舍去浮点部分
    - 注意：如果目标整数类型不能表示舍入的结果，在目前这是一个未定义行为。包括Inf和NaN。这是一个bug，会在后续版本中修复。
- 整型转换为浮点类型会产生这个整型的浮点型表示，
- f32转换为f64可以无损失地完美转换，必要的时候做舍入（舍入到最近的可能取值，距离相同的取偶数）
- f64转换为f32会生成最近可能值（舍入到最近的可能取值，距离相同的取偶数）

原文链接：https://doc.rust-lang.org/nomicon/transmutes.html#transmutes

变形(Transmutes)

类型系统你给我滚开！我要自己解析这些字节，不成功便成仁！虽然本书都是关于非安全的内容，我还是希望你能仔细考虑避免使用本章讲到的内容。这是你在Rust中所能做到的真真正正、彻彻底底、最最可怕的非安全行为。所有的保护机制都形同虚设。

`mem::transmute<T, U>` 接受一个 `T` 类型的值，然后将它重新解析为类型 `U` 。唯一的限制是 `T` 和 `U` 必须有同样的大小。可能产生未定义行为的情况让人看着头疼。

- 最重要的，创建任一类型的处于不合法状态的示例，都将产生不可预知的混乱
- transmute有一个重载的返回类型。如果没有明确指定返回类型，它会返回一个满足类型推断的奇怪类型
- 使用不合法的值构建基本类型是未定义行为
- 非repr(C)的类型之间相互变形是未定义行为
- `&` 变形为 `&mut` 是未定义行为
    - `&` 变形为 `&mut` 永远都是未定义行为
    - 不要多想，你绝对不能这么做
    - 不要多想，你没有什么特殊的
- 变形为一个未指定生命周期的引用会产生无界生命周期

`mem::transmute_copy<T, U>` 很神奇地比这更加不安全。它从 `&T` 拷贝 `size_of<U>` 个字节并将它们解析为 `U` 。 `mem::transmute` 仅有的类型大小的检查都不见了（因为拷贝类型前缀有可能是合法的），只不过 `U` 的尺寸比 `T` 大会被视为一个未定义行为。

当然，本章大部分的功能都可以通过指针的显式转换实现。

原文链接：https://doc.rust-lang.org/nomicon/uninitialized.html

未初始化内存

所有运行期分配的内存开始时都是”未初始化”状态。这种状态下内存的值是一组不确定的字节，甚至有可能不是使用这块内存的类型的合法值。将这段内存的值解析为任何类型都是未定义行为。千万不要这么做。

Rust提供了处理未初始化内存的方式，既有安全的方式也有非安全的方式。

原文链接：https://doc.rust-lang.org/nomicon/checked-uninit.html

安全方式

和C一样，所有栈上的变量在显式赋值之前都是未初始化的。而和C不同的是，Rust禁止你在赋值之前读取它们：

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

```
src/main.rs:3:20: 3:21 error: use of possibly uninitialized variable: `x`
src/main.rs:3     println!("{}", x);
                                  ^
```

这个错误基于分支分析：任何一个分支在第一次使用 `x` 之前都必须对它赋值。有意思的是，如果每一个分支都只赋值一次的话，Rust并不要求变量是可变的。但是，这个分析过程没有配合常量分析。所以下面这段代码可以编译：

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

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

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

但是这段却不能编译：

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

```
}
```

```
src/main.rs:6:17: 6:18 error: use of possibly uninitialized variable: `x`
src/main.rs:6   println!("{}", x);
```

而这一段又可以编译：

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!("{}", x);
    }
    // 不关心其他的未初始化变量的分支
    // 因为我们并不使用那些分支
}
```

当然，虽然分析过程不知道变量的实际值，它对依赖和控制流程的理解还是比较深入的。比如，这段代码是正确的：

```
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;
    }
}
// 它也知道如果没有执行break的话，代码不会运行到这里
// 所以在这里x一定已经被初始化了
println!("{}", x);
```

如果值从变量中移出且变量类型不是Copy，那么变量逻辑上处于未初始化状态。就是说：

```
fn main() {
    let x = 0;
    let y = Box::new(0);
```

```
    let z1 = x; // x仍然是合法的，因为i32是Copy
    let z2 = y; // y现在逻辑上未初始化，因为Box不是Copy
}
```

但是，这个例子中对 `y` 重新赋值要求 `y` 是可变的，因为安全Rust能够观察到 `y` 的值发生了变化：

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

否则 `y` 会被视为一个全新的变量。

原文链接：https://doc.rust-lang.org/nomicon/drop-flags.html

Drop标志

前一章的例子涉及到Rust的一个有趣的问题。我们看到我们可以安全地为一段内存初始化、反初始化、再初始化。对于Copy类型，这一点不是很重要，因为数据不过是一堆字节而已。但是对于有析构函数的类型就是另外一回事了：变量每次被赋值或者离开作用域的时候，Rust都需要判断是否要调用析构函数。在有条件地初始化的情况下，Rust是如何做到这一点的呢？

注意，不是所有的赋值操作都需要考虑这一点。通过解引用赋值是一定会触发析构函数，而使用 `let` 赋值则一定不会触发：

```
let mut x = Box::new(0); // let传建一个全新的变量，所以一定不会调用drop
let y = &mut x;
*y = Box::new(1);        // 解引用假设被引用变量是初始化过的，所以一定会调用drop
```

只有当覆盖一个已经初始化的变量或者变量的一个子成员时，才需要考虑这个问题。

Rust实际上是在运行期判断是否销毁变量。当一个变量被初始化和反初始化时，变量会更新它的”drop标志“的状态。通过解析这个标志的值，判断变量是否真的需要执行drop。

当然，大多数情况下，在编译期就可以知道一个值在每一点的初始化状态。符合这一点的话，编译器理论上可以生成更有效率的代码！比如，无分支的程序有着如下的静态drop语义：

```
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
```

类似的，有分支的代码当所有分支中的初始化行为一致的时候，也可以有静态的drop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
```

但是，下面的代码则需要运行时信息以正确执行drop：

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

当然，修改为下面的代码就又可以得到静态drop语义：

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

drop标志储存在栈中，并不在实现Drop的类型里。

原文链接：https://doc.rust-lang.org/nomicon/unchecked-uninit.html

非安全方式

一个特殊情况是数组。安全Rust不允许部分地初始化数组。初始化一个数组时，你可以通过 `let x = [val; N]` 为每一个位置赋予相同的值，或者是单独指定每一个成员的值 `let x = [val1, val2, val3]` 。不幸的是，这个要求太苛刻了。很多时候我们需要用增量或者动态的方式初始化数组。

非安全Rust给我们提供了一个很有力的工具以处理这一问题： `mem::uninitialized` 。这个函数假装返回一个值，但其实它什么也没有做。我们用它来欺骗Rust我们已经初始化了一个变量了，从而可以做一些很神奇的事情，比如有条件还有增量地初始化。

不过，它也给我们打开了各种问题的大门。在Rust中，对于已初始化和未初始化的变量赋值，是有不同的含义的。如果Rust认为变量未初始化，它会将字节拷贝到未初始化的内存区域，别的就什么都不做了。可如果Rust判断变量已初始化，它会销毁原有的值！因为我们欺骗Rust值已经初始化，我们再也不能安全地赋值了。

系统分配器返回一个指向未初始化内存的指针，与它配合时同样会造成问题。

接下来，我们还必须使用 `ptr` 模块。特别是它提供的三个函数，允许我们将字节码写入一块内存而不会销毁原有的变量。这些函数为： `write` ， `copy` 和 `copy_nonoverlapping` 。

- `ptr::write(ptr, val)` 函数接受 `val` 然后将它的值移入 `ptr` 指向的地址
- `ptr::copy(src, dest, count)` 函数从 `src` 处将 `count` 个T占用的字节拷贝到 `dest` 。（这个函数和 `memmove` 相同，不过要注意参数顺序是反的！）
- `ptr::copy_nonoverlapping(src, dest, count)` 和 `copy` 的功能是一样的，不过它假设两段内存不会有重合部分，因此速度会略快一点。（这个函数和 `memcpy` 相同，不过要注意参数顺序是反的！）

很显然，如果这些函数被滥用的话，很可能导致错误或者未定义行为。它们唯一的要求就是被读写的位置必须已经分配了内存。但是，向任意位置写入任意字节很可能造成不可预测的错误。

下面的代码集中展示了它们的用法：

```
use std::mem;
use std::ptr;

// 数组的大小是硬编码的但是可以很方便地修改
// 不过这表示我们不能用[a, b, c]这种方式初始化数组
const SIZE: usize = 10;

let mut x: [Box<u32>; SIZE];
```

```
9.
10. unsafe {
11.     // 欺骗Rust说x已经被初始化
12.     x = mem::uninitialized();
13.     for i in 0..SIZE {
14.         // 十分小心地覆盖每一个索引值而不读取它
15.         // 注意：异常安全性不需要考虑；Box不会panic
16.         ptr::write(&mut x[i], Box::new(i as u32));
17.     }
18. }
19.
20. println!("{:?}", x);
```

需要注意，你不用担心 `ptr::write` 和实现了 `Drop` 的或者包含 `Drop` 子类型的类型之间无法和谐共处，因为Rust知道这时不会调用 `drop` 。类似的，你可以给一个只有局部初始化的结构体的成员赋值，只要那个成员不包含 `Drop` 子类型。

但是，在使用未初始化内存的时候你需要时刻小心，Rust可能会在值未完全初始化的时候就尝试销毁它们。如果一个变量有析构函数，那么变量作用域的每一个代码分支都应该在结束之前完成变量的初始化。否则会导致崩溃。

这就是未初始化内存的全部内容！其他地方基本上不会再涉及到未初始化内存了，所以如果你想跳过本章，请千万小心。

原文链接：https://doc.rust-lang.org/nomicon/obrm.html#the-perils-of-ownership-based-resource-management-obrm

基于所有权的资源管理(OBRM)的风险

OBRM（又被成为RAII：Resource Acquisition is Initialization，资源获取即初始化），在Rust中你会有很多和它打交道的机会，特别是在使用标准库的时候。

这个模式简单来说是这样的：如果要获取资源，你只要创建一个管理它的对象。如果要释放资源，你只要销毁这个对象，由对象负责为你回收资源。而所谓资源通常指的就是内存。 `Box` ， `Rc` ，以及 `std::collections` 中几乎所有的东西都是为了方便且正确地管理内存而存在的。这对于Rust尤为重要，因为我们并没有垃圾回收器帮我们管理内存。关键点就在这：Rust要掌控一切。不过我们并不是只能管理内存。差不多所有的系统资源，比如线程、文件、还有socket，都可以用到这些API。

原文链接：https://doc.rust-lang.org/nomicon/constructors.html

构造函数

创建一个自定义类型的实例的方法只有一种：先命名，然后一次性初始化它的所有成员：

```
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;
```

就是这样。其他的所谓创建类型实例的方式，不过是调用一些函数，而函数的底层还是要依赖于这个真正的构造函数。

和C++不同，Rust没有很多不同种类的构造函数，比如拷贝、默认、赋值、移动、还有其他各种构造函数。之所以这样的原因有很多，不过归根结底还是因为Rust显式化的设计哲学。

移动构造函数对于Rust没什么用，因为我们并不需要让类型关心它们在内存上的位置。没一个类型都有可能随时被memcopy到内存中其他的位置上。这也意味和那种存储于栈上却依然可以移动的侵入式链表在Rust中是不可能（安全地）存在的。

复制和拷贝构造函数也是不存在的，因为Rust中的类型有且仅有移动语义。 `x = y` 只是将 `y` 的字节移动到 `x` 的变量中。Rust倒是提供了两种和C++中的copy语义相似的功能： `Copy` 和 `Clone` 。 `Clone` 很像是拷贝构造函数，但是它不会被隐式调用。你必须在需要复制的元素上显式调用 `clone` 方法、 `Copy` 是 `Clone` 的一个特例，它的实现只会拷贝字节码。Copy类型在移动的时候会隐式地复制，但是因为Copy的定义，这个方法只是不把旧的值设置为未初始化而已—其实是一个no-op。

虽然Rust确实有一个 `Default` trait，它与默认构造函数很相似，但是这个trait极少被用到。这是因为变量不会被隐式初始化。 `Default` 一般只有在泛型编程中才有用。而具体的类型会提供一

个 `new` 静态方法来实现默认构造函数的功能。这个和其他语言中的 `new` 关键字没什么关系，也没有什么特殊的含义。它仅仅是一个明明习惯而已。

TODO: 介绍“placement new”?

原文链接：https://doc.rust-lang.org/nomicon/destructors.html

析构函数

Rust通过 `Drop` trait提供了一个成熟的自动析构函数，包含了这个方法：

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

这个方法给了类型一个彻底完成工作的机会。

`drop` 执行之后，**Rust会d递归地销毁** `self` 的所有成员

这个功能很方便，你不需要每次都写一堆重复的代码来销毁子类型。如果一个结构体在销毁的时候，除了销毁子成员之外不需要做什么特殊的操作，那么它其实可以不用实现 `Drop` 。

在**Rust 1.0**中，没有什么合适的方法可以打断这个过程。

注意，参数是 `&mut self` 意味着即使你可以阻止递归销毁，Rust也不允许你将子成员的所有权移出。对于大多数类型来说，这一点完全没问题。

比如，一个自定义的 `Box` 的实现，它的 `Drop` 可能长这样：

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

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

struct Box<T>{ ptf: 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.dealloc(c.cast(), Layout::new::<T>())
        }
    }
}
```

这段代码是正确的，因为当Rust要销毁 `ptr` 的时候，它见到的是一个Unique，没有 `Drop` 的实现。类似的，也没有人能在销毁后再使用 `ptr` ，因为drop函数退出之后，他就不可见了。

可是这段代码是错误的：

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

use std::alloc::{Alloc, 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.dealloc(c.cast(), LayOut::new::<T>());
        }
    }
}

struct SuperBox<T> ( my_box: Box<T> )

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

当我们在 `SuperBox` 的析构函数里回收了 `box` 的 `ptr` 之后，Rust会继续让 `box` 销毁它自己，这时销毁后使用(use-after-free)和两次释放(double-free)的问题立刻接踵而至，摧毁一切。

注意，递归销毁适用于所有的结构体和枚举类型，不管它有没有实现 `Drop` 。所以，这段代码

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

在销毁的时候也会调用 `data1` 和 `data2` 的析构函数，尽管这个结构体本身并没有实现 `Drop` 。

这样的类型“需要Drop却不是Drop”。

类似的

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

当（且仅当）一个实例储存着 `Next` 变量时，它就会销毁内部的 `Box` 成员。

一般来说这其实是一个很好的设计，它让你在重构数据布局的时候无需费心添加/删除 `drop` 函数。但也有很多的场景要求我们必须在析构函数中玩一些花招。

如果想阻止递归销毁并且在 `drop` 过程中将 `self` 的所有权移出，通常的安全的做法是使用 `Option` ：

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

use std::alloc::{Alloc, 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.dealloc(c.cast(), LayOut::new::<T>());
        }
    }
}

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

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // 回收box的内容，而不是drop它的内容
            // 需要将box设置为None，以阻止Rust销毁它
            let my_box = self.my_box.take().unwrap();
```

```
27.             let c: NonNull<T> = my_box.ptr.into();
28.             Global.dealloc(c.cast(), LayOut::new::<T>());
29.             mem::feorget(my_box);
30.         }
31.     }
32. }
```

但是这段代码显得很奇怪：我们认为一个永远都是 `Some` 的成员有可能是 `None` ，仅仅因为析构函数中用到了一次。但反过来说这种设计又很合理：你可以在析构函数中调用 `self` 的任意方法。在成员被反初始化之后就完全不能这么做了，而不是禁止你搞出一些随意的非法状态。（斜体部分没看懂，建议看原文）

权衡之后，这是一个可以接受的方案。你可以将它作为你的默认选项。但是，我们希望以后能有一个方法明确声明哪一个成员不会自动销毁。

原文链接：https://doc.rust-lang.org/nomicon/leaking.html

泄露

(译注：本章较长，而且译者在翻译过程中喝多了，信达雅全都有如浮云了……求凑合看，有空会回来校对的……)

基于所有权的资源管理是为了简化复合类型而存在的。你在创建对象的时候获取资源，在销毁对象的时候释放资源。由于析构过程做了处理，你不可能忘记释放资源，而且是尽可能早地释放资源！这简直是一个完美的方案，解决了我们所有的问题。

可实际上可怕的事情遍地都是，我们还有新的奇怪的问题需要解决。

许多人觉得Rust已经消除了资源泄露的可能性。实际应用中也差不多是这样。你不太可能看到安全Rust出现不可控制的资源泄露。

但是，从理论的角度来说，情况却完全不同。在科学家看来，“泄露”太过于抽象，根本无法避免。很可能就会有人在程序的开头初始化一个集合，塞进去一大堆带析构函数的对象，接下来就进入一个死循环，再也不理开始的那个集合。那个集合就只能坐在那里无所事事，死死地抱着宝贵的资源等着程序结束（这时操作系统会强制回收资源）。

我们可能要给泄露一个更严格的定义：无法销毁不可达(unreachable)的值。Rust也不能避免这种泄露。事实上Rust还有一个制造泄露的函数：`mem::forget`。这个函数获取传给它的值，但是不调用它的析构函数。

`mem::forget`曾经被标为unsafe，作为不要滥用它的一种警告。毕竟不调用析构函数一般来说不是一个好习惯（尽管在某些特殊情况下很有用）。但其实这个判断比较不靠谱，因为在安全代码中不调用析构函数的情况很多。最经典的例子是一个循环引用的计数引用。

安全代码可以合理假设析构函数泄露是不存在的，因为任何有这一问题的程序都可能是错误的。但是，非安全代码不能依赖于运行析构函数来保证程序安全。对于大多数类型而言，这一点不成问题：如果不能调用析构函数，那其实类型本身也是不可访问的，所以这就不是个问题了，对吧？比如，你没有释放`Box<u8>`，那么你会浪费一点内存，但是这并不会违反内存安全性。

但是对于代理类型，我们就要十分小心它的析构函数了。有几个类型可以访问一个对象，却不拥有对象的所有权。代理类型很少见，而需要你特别小心的类型就更稀少了。但是，我们要仔细研究一下标准库中的三个有意思的例子

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

Drain

`drain` 是一个集合API，它将容器内的数据所有权移出，却不占有容器本身。我们可以声明一个 `Vec` 所有内容的所有权，然后复用分配给它的空间。它产生一个迭代器（Drain），以返回Vec的所有值。

现在，假设Drain正迭代到一半：有一些值被移出，还有一些没移出。这表明Vec里有一堆逻辑上未初始化的数据！我们可以在删除值的时候在Vec里再备份一份，但这种方法的性能是不可忍受的。

实际上，我们希望Drain在销毁的时候能够修复Vec的后台存储。他要备份那些没有被移除的元素（drain支持子范围），然后修改Vec的 `len` 。这种方法甚至还是unwinding安全的！完美！

看看下面这段代码

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

{
    // 开始drain，vec无法再被访问
    let mut drainer = vec.drain(..);

    // 移除两个元素，然后立刻销毁他们
    drainer.next();
    drainer.next();

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

// 不好，vec[0]已经被销毁了，我们在读一块释放后的内存
println!("{}", vec[0]);
```

这个显然很不好。我们现在陷入了两难的境地：保证每一步产生一致的状态，需要付出巨大的性能代价（抵消掉了API带来的所有好处）；而不保证一致状态则会在安全代码中产生未定义行为（使API失去稳定性）。

那我们能做什么呢？我们采用一种简单粗暴的方式保证状态一致性：开始迭代的时候就设置Vec的长度为0，然后在析构函数里根据需要再恢复。这样做，在一切正常的情况下，我们可以用最小的代价获得正确的行为。但是，如果有人就是不管不顾地在迭代中间 `mem::forget` ，那么结果就是泄露或者更坏（还可能让Vec处于一种虽然一致但实际上不正确的状态）。由于我们认为 `mem::forget` 是安全地，那么这种行为也是安全地。我们把造成更多泄露的泄露叫做泄露扩大化(leak amplification)。

Rc

`Rc` 的情况很有意思，第一眼看上去它根本不像是一个代理类型。毕竟，它自己管理着它指向的数据，并且在销毁 `Rc` 的时候也会同时销毁数据的值。泄露 `Rc` 的数据好像并不怎么危险。那会让引用计数持续增长，而数据不会被释放或销毁。这和 `Box`的行为是一项的，对吧？

并不是。

我们看一下这个 `Rc` 的简单实现：

```
1.  struct Rc<T> {
2.      ptr: *mut RcBox<T>,
3.  }
4.
5.  struct RcBox<T> {
6.      data: T,
7.      ref_count: usize,
8.  }
9.
10. impl<T> Rc<T> {
11.     fn new(data: T) -> Self {
12.         unsafe {
13.             // 如果heap::allocate是这样的不是很好嘛？
14.             let ptr = heap::allocate::<RcBox<T>>();
15.             ptr::write(ptr, RcBox {
16.                 data: data,
17.                 ref_count: 1,
18.             });
19.             Rc { ptr: ptr }
20.         }
21.     }
22.
23.     fn clone(&self) -> Self {
24.         unsafe {
25.             (*self.ptr).ref_count += 1;
26.             Rc { ptr: self.ptr }
27.         }
28.     }
29. }
30.
31. impl<T> Drop for Rc<T> {
32.     fn drop(&mut self) {
```

```
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // 销毁数据然后释放空间
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

要解决这个问题，我们可以检查 `ref_count` 并根据情况做一些处理。标准库的做法是直接废弃对象，因为这种情况下你的程序进入了一种非常危险的状态。当然，这是一个十分诡异的边界场景。

thread::scoped::JoinGuard

`thread::scoped` 可以保证父线程在共享数据离开作用域之前join子线程，通过这种方式子线程可以引用父线程栈中的数据而不需要做什么同步操作。

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

这里 `f` 是供其他线程执行的闭包。 `F: Send + 'a` 表示闭包引用数据的生命周期是 `'a` ，而且它可能拥有这个数据或者数据是一个 `Sync` （说明 `&data` 是 `Send` ）。

因为 `JoinGuard` 有生命周期，它所用到的数据都是从父线程里借用的。这意味着 `JoinGuard` 不能比线程使用的数据存活更长。当 `JoinGuard` 被销毁的时候它会阻塞父线程，保在父线程中被引用的数据离开作用域之前子线程都已经终止了。

用法是这样的：

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let guards = vec![];
    for x in &mut data {
        // 将可变引用移入闭包，然后再另外一个线程里执行它
        // 闭包有生命周期，其界限由可变引用x的生命周期决定
        // 返回的guard也和闭包有相同的生命周期，所以它也和x一样可变借用了data
        // 这意味着在guard销毁前我们不能访问data
        let guard = thread::scoped(move || {
            *x *= 2;
```

```
        });
        // 储存线程的guard供后面使用
        guards.push(guard);
    }
    // 所有的guard在这里被销毁，强制线程join（主线程阻塞在这里等待其他线程终止）。
    // 等到线程join后，数据的借用就过期了，数据又可以在主线程中被访问了
}
// 数据在这里已经完全改变了。
```

这个似乎完全能够正常工作！Rust的所有权系统完美地保证了这一点！……不过这一切的前提是析构函数必须被调用。

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // 好一点的情况是这里会有数据竞争
        // 最坏的情况是这里会有释放后应用(use-after-free)
        *data += 1;
    });
    // 因为guard被forget了，线程不会阻塞
    mem::forget(guard);
}
// Box在这里被销毁，而子线程可能会也可能不会在这里访问数据。
```

Duang！保证析构函数能运行是这个api的基础，上面这段代码需要一个全新的设计才行。

原文链接：https://doc.rust-lang.org/nomicon/unwinding.html

展开(Unwinding)

Rust有一个分层的错误处理体系：

- 如果有些值可以为空，就用`Option`
- 如果发生了错误，而错误可以被正常处理，就用`Result`
- 如果发生了错误，但是没办法正常处理，就让线程panic
- 如果发生了更严重的问题，中止(abort)程序

`Option`和`Result`在大多数情况下都是默认的优先选择，因为API的用户可以根据自己的考虑将它们变为panic或中止。panic会导致线程停止正常的执行流程、展开栈（unwind stack）、调用析构函数，整个流程和函数返回时一样。

从1.0开始，Rust对Panic的处理显得有些混乱。在很早很早以前，Rust的设计非常接近Erlang。和Erlang一样，Rust由许多轻量级的任务(task)组成，当任务进入错误状态的时候，它们使用Panic停止自己。Panic和Java或者C++中的异常不同，它不能在任意时间点被捕获。Panic只能被任务的所有者捕获，而捕获后必须立即对它进行相应处理，否则任务会自己停止。

展开(unwinding)在这种场景下十分重要，因为如果任务的析构函数没有被调用的话，会导致内存和其他系统资源的泄露。由于任务有可能在正常运行过程中就挂掉，它对于需要长期运行的系统很不友好。

而在后来Rust的发展过程中，我们推崇尽可能少的抽象，所以上文的编程风格也就显得过时了。轻量级的任务被重量级的操作系统线程所取代。不过在1.0的稳定版本中，panic还是只能被父线程捕获。这意味着捕获一个panic需要唤醒一个系统线程！这和Rust的零开销抽象的设计哲学是完全相悖的。

有一个不稳定的API叫做`catch_panic`，它可以在不启动一个线程的情况下捕获panic。不过我们还是希望你谨慎地使用它。特别是现在Rust对展开的实现已经针对“不展开”的情况做了很多的优化。即使一个程序支持展开，只要它没有做展开的动作，在运行期就没有额外的开销。但同时，真的展开操作是比Java等其他语言的开销更大的。不要在正常运行的情况下让你的程序栈展开。只有当程序出错或遇到极端的问题时，你才应该使用Panic。

Rust的展开方式没有试图和其他任何一种语言的展开方式相兼容。所以，从其他语言展开Rust的栈，或者从Rust展开其他语言的栈，全都属于未定义行为。你必须在进入FFI调用之前捕获所有的Panic！你可以决定具体的实现方法，但不能什么都不做。否则的话，最好的情况是你的应用程序会崩溃。而最坏的情况是，你的程序不会崩溃，但会在彻底混乱的状态下持续运行。

原文链接：https://doc.rust-lang.org/nomicon/exception-safety.html

异常(exception)安全性

虽然前面说过我们应该慎用展开，但是还是有许多的地方会Panic。如果你对 `None` 调用 `unwrap` 、使用超出范围的索引值、或者用0做除数，你的程序就要panic。在debug模式下，所有的计算操作在溢出的时候也都会panic。除非你十分小心并且严格控制着每一条代码的行为，否则所有的东西都有展开的可能，你需要时刻准备迎接它。

在更广大的程序设计世界里，应对展开这件事通常被称之为“异常安全“。在Rust中，我们需要考虑两个层次的异常安全性：

- 在非安全代码中，异常安全的下限是要保证不能违背内存安全性。我们称之为最小异常安全性。
- 在安全代码中，异常安全性要保证程序时刻在做正确的事情。我们称之为最大异常安全性。

在许多情况下，非安全代码在处理展开的时候需要考虑到那些写得很糟糕的安全代码。一些只是暂时导致不稳定状态的程序需要小心，一旦触发了Panic会导致这种状态无法使用。这表示在不稳定状态依然存在的情况下，我们需要保证值运行不触发Panic的代码；或者在触发Panic的时候即使处理，清除这种状态。这也表明Panic看到的状态并不一定非得是连续的状态，我们只需要保证它是安全地状态就可以。

大多数非安全代码都比较容易实现异常安全。因为它控制着程序运行的每个细节，而且大部分代码不会Panic。但是非安全代码也经常要做诸如在未初始化数据的数组上反复运行外部代码这样的操作。这种代码就需要小心考虑异常安全性了。

Vec::push_all

`Vec::push_all` 使用一个 `slice` 扩充 `Vec` ，由于它没有具体化类型，所以能获得较高的效率。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 {
            // 因为我们调用了reserve，所以不会出现溢出
            self.set_len(self.len() + to_push.len());

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

```
    }
}
```

我们不去使用 `push` ，因为它会对Vec的容量和 `len` 做额外的检查，而有些情况下我们能够明确知道容量是充足的。这段代码的逻辑是完全正确的，但是却有一个问题：它不是异常安全的！ `set_len` 、 `offset` 和 `write` 都没问题，但是 `clone` 是一颗引发Panic的炸弹。

`Clone` 的实现是我们无法控制的，它很可能会panic。如果它真的panic了，这个方法会提前退出，但我们之前给Vec设置的更大的长度会一致保持下去。当Vec被访问或者销毁的时候，它会读取未初始化内存！

解决方法很简单。如果我们要保证我们clone的值都被销毁了，我们可以在每一次循环里设置 `len` 。如果我们只是想保证不会出现读取未初始化内存的情况，我们可以在循环之后设置 `len` 。

BinaryHeap::sift_up

对二叉堆做冒泡比扩充一个Vec要更复杂一点。伪代码是这样的：

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

将它翻译成Rust很容易，但是性能不会让人满意： `self` 元素要一遍一遍做无意义的交换。我们更喜欢下面的版本：

```
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
```

这段代码保证各个元素被尽量少的复制(通常每个元素需要被复制两次)。但是这样它会引发异常安全问题！任何时刻都存在着一个值的两份拷贝。如果这个方法中出现panic，有一些东西可能会被二次释放。不幸的是，我们同样不能完全掌控这段代码，因为比较操作是用户定义的。

这个解决方案比Vec的要困难。一个选项是把用户定义代码和非安全代码拆分成两个阶段：

```
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爆炸了，也不会伤及无辜，因为我们还没有实际改变堆的状态。等我们开始在堆上搞事情的时候，我们只会使用我们信任的数据和函数，不用担心panic。

你可能对这个设计感到很不爽。这个属于作弊！而且我们必须对堆完整遍历两次！好吧，让我们直面困难，把不信任代码和不安全代码混合在一起。

如果Rust像Java一样有 `try` 和 `finally` ，我们可以这么做：

```
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
```

基本思想很简单：如果比较操作panic了，我们就把取出的元素塞回到逻辑上未初始化的位置然后退出。访问这个堆的人可能会发现堆的状态是不连续的，但是至少这个方案不会引发二次释放！如果算法正常结束的话，这个设计就和我们最开始不做任何处理的方案一模一样了。

可惜，Rust并没有这些东西，所以我们只能自己早轮子了！我们把算法的状态储存在一个独立的结构体中，结构体的析构函数起到了”finally“的功能。不管有没有panic，析构函数都会被调用并且清除我们留下状态。

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

impl<'a, T> Hole<'a, T> {
```

```
 9.      fn new(data: &'a mut [T], pos: usize) -> Self {
10.          unsafe {
11.              let elt = ptr::read(&data[pos]);
12.              Hole {
13.                  data: data,
14.                  elt: Some(elt),
15.                  pos: pos,
16.              }
17.          }
18.      }
19.
20.      fn pos(&self) -> usize { self.pos }
21.
22.      fn removed(&self) -> &T { self.elt.as_ref().unwrap() }
23.
24.      unsafe fn get(&self, index: usize) -> &T { &self.data[index] }
25.
26.      unsafe fn move_to(&mut self, index: usize) {
27.          let index_ptr: *const _ = &self.data[index];
28.          let hole_ptr = &mut self.data[self.pos];
29.          ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
30.          self.pos = index;
31.      }
32.  }
33.
34.  impl<'a, T> Drop for Hole<'a, T> {
35.      fn drop(&mut self) {
36.          // 再次填充hole
37.          unsafe {
38.              let pos = self.pos;
39.              ptr::write(&mut self.data[pos], self.elt.take().unwrap());
40.          }
41.      }
42.  }
43.
44.  impl<T: Ord> BinaryHeap<T> {
45.      fn sift_up(&mut self, pos: usize) {
46.          unsafe {
47.              // 取出pos处的值，然后创建一个hole
48.              let mut hole = Hole::new(&mut self.data, pos);
49.
50.              while hole.pos() != 0 {
```

```
51.                 let parent = parent(hole.pos());
52.                 if hole.removed() <= hole.get(parent) { break }
53.                 hole.move_to(parent);
54.             }
55.             // 无论有没有panic, hold在此处都会无条件地被填充
56.         }
57.     }
58. }
```

原文链接：https://doc.rust-lang.org/nomicon/poisoning.html

污染

所有的非安全代码都必须保证最小异常安全性，但是并不是所有的类型都能保证最大异常安全性。即使一个类型保证了这一点，我们的代码也可能把它搞乱。比如，一个整数类型肯定是异常安全的，但是它自己没有语义。而一段代码可能在panic的时候没有正确更新整数的值，因此导致了不连续的状态。

这种情况通常没什么大不了的，因为异常发生时所有的东西都应该被销毁。例如，你给一个线程传递了一个Vec而线程panic了，这时Vec处于奇怪的状态其实也无所谓。反正它会被销毁掉并且永远消失。但是，一些类型会在发生panic的时候偷偷隐藏数据的值。

这些类型在遇到panic的时候可能会污染（poison）自己。污染没有什么特殊的含义，它通常只是指禁止其他人正常地使用它。最明显的例子是标准库中的Mutex类型。Mutex会在它的一个MutexGuards（Mutex在获取锁的时候返回的对象）因为panic而销毁的时候污染自己，这之后所有尝试给Mutex上锁的操作都会返回 `Err` 或者Panic。

从Rust惯常的角度看，Mutex的污染不算真正地保障安全性。污染是一种守护机制，在Mutex上锁期间遇到Panic后，禁止访问里面的数据。这种数据可能正被修改了一半，处于一种不连续或者不完整的状态。需要注意，只要数据正常写入了，即使使用这种类型也不会违反内存安全性。毕竟，这是最小异常安全的要求。

但是，如果Mutex包含一个没有设置任何属性的BinaryHeap，那么使用它的代码不太可能执行作者期望的行为。当然，程序也不可能正常运行下去。不过如果你能完全、绝对、百分之一百地肯定你可以用这些数据做点事情，Mutex还是提供了一个让你继续获得锁的方法。毕竟这是安全地，只不过可能没什么意义。

原文链接：https://doc.rust-lang.org/nomicon/concurrency.html

并发和并行

Rust作为一种语言，它其实并不知道怎么做并发或者并行。是标准库提供了操作系统线程和阻塞系统调用的支持。所有的平台都支持这些功能，基于这些一致的功能构建的抽象更容易被广泛接受。而消息传递、绿色线程、异步API这些则没有这么广的支持度，在它们之上构建的抽象就要引入一些权衡取舍，所以我们没有将它们纳入1.0。

但是，Rust构建并发模型的方式也让你可以比较容易地设计自己的并发范式，并作为一个库与其他人的代码一起工作。只要保证生命周期是正确的、Send和Sync设置得合理，以及处理好数据竞争。或者更准确的说，是不……要……竞……争。

原文链接：https://doc.rust-lang.org/nomicon/races.html

数据竞争与竞争条件

安全Rust保证了不存在数据竞争。数据竞争指的是：

- 两个或两个以上的线程并发地访问同一块内存
- 其中一个线程做写操作
- 其中一个线程是非同步(unsynchronized)的

数据竞争导致未定义行为，所以不可能在安全Rust中存在。大多数情况下，Rust的所有权系统就可以避免数据竞争：不可能有可变引用的别名，因此也就不可能有数据竞争。但是内部可变性把这件事弄得复杂了，这也是为什么我们要有Send和Sync（见下）。

但是**Rust**并不会避免一般竞争条件。

因为要做到这一点其实是不可能的，而且好像也是不必要的。你的硬件是竞争的，操作系统是竞争的，计算机上其他的程序是竞争的，整个世界都是竞争的。任何一个声称可以避免所有竞争条件的系统，即使没有错误，也一定及其难用。

所以，安全Rust出现死锁，或者因为不正确的同步而做出一些奇怪的行为，这些都是可以接受的。显然这样的程序并不是最理想的程序，但Rust也只能帮你到这了。而且，竞争条件自己不能违反Rust的内存安全性。只有配合上其他的非安全代码，竞争条件才有可能破坏内存安全。比如：

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

let data = vec![1, 2, 3, 4];
// 使用Arc，这样即使程序已经执行完毕了，存储AtomicUsize的内存依然存在，
// 其他的线程可以增加它的值。否则Rust不能编译这段代码，因为thread:spawn
// 对生命周期有限制。
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// move获得other_idx的所有权，将它移入线程
thread::spawn(move || {
    // 可以改变idx，因为它的值是一个原子，不会引起数据竞争
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// 用原子中的值做索引。这么做是安全的，因为我们只读取了一次原子的内存，
```

```
// 然后将读出的值的拷贝传递给Vec做索引。索引过程可以做正确的边界检查，
// 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。
// 这符合竞争条件，因为程序执行得正确与否（panic几乎不可能是正确的）
// 依赖于线程的执行顺序
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```
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获得other_idx的所有权，将它移入线程
thread::spawn(move || {
    // 可以改变idx，因为它的值是一个原子，不会引起数据竞争
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // 在边界检查之后读取idx的值是不正确的，因为它有可能已经改变了。
        // 这是一个竞争条件，而且十分危险，因为我们要使用的get_unchecked是非安全的。
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

原文链接：https://doc.rust-lang.org/nomicon/send-and-sync.html

Send和Sync

不是所有人都遵守可变性的原则。有一些类型允许你拥有一块内存的多个别名，同时还改变内存的值。除非这些类型使用同步来控制访问，否则它们就不是线程安全的。Rust根据 `Send` 和 `Sync` 这两个trait获取相关信息。

- 如果一个类型可以安全地传递给另一个线程，这个类型是 `Send`
- 如果一个类型可以安全地被多个线程共享(也就是 `&T` 是 `Send`)，这个类型是 `Sync`

`Send` 和 `Sync` 是Rust并发机制的基础。因此，Rust赋予它们许多的特性，以保证它们能正确工作。首当其冲的，它们都是非安全trait。这表明它们的实现也是非安全的，而其他的非安全代码则可以假设这些实现是正确的。由于它们是标志trait（它们没有任何关联的方法），“正确地实现”仅仅意味着实现满足它所需要的内部特征。不正确地实现 `Send` 和 `Sync` 会导致未定义行为。

`Send` 和 `Sync` 还是自动推导的trait。和其他的trait不同，如果一个类型完全由 `Send` 或 `Sync` 组成，那么这个类型本身也是 `Send` 或 `Sync` 。几乎所有的基本类型都是 `Send` 和 `Sync` ，因此你能见到的很多类型也就都是 `Send` 和 `Sync` 。

主要的例外情况有：

- 裸指针不是 `Send` 也不是 `Sync` （因为它们没有安全性保证）
- `UnsafeCell` 不是 `Sync` （所以 `Cell` 和 `RefCell` 也不是）
- `Rc` 不是 `Send` 或 `Sync` （因为引用计数是共享且非同步的）

`Rc` 和 `UnsafeCell` 是典型的非线程安全的：它们允许非同步地共享可变状态。可是，裸指针严格来说并不一定非得是非线程安全不可。通过裸指针做任何有意义的事情都需要先对它解引用，这一步就已经是非安全的了。从这个角度来说，有人可能会认为把它标为线程安全的也未尝不可。

可是，它们被标为非线程安全的主要目的是避免包含它们的类型自动成为线程安全的。这些类型都有着重要的不可追踪的所有权，保证它们线程安全需要花费大量的精力，而他们的作者不太可能做到这一点。 `Rc` 就是一个很好的例子，一个包含 `*mut` 的类型绝对不能是线程安全的。

不是自动推导的类型也可以很容易地实现 `Send` 和 `Sync` ：

```
1.  struct MyBox(*mut u8);
2.
3.  unsafe impl Send for MyBox {}
4.  unsafe impl Sync for MyBox {}
```

还有一个很少见的场景，一个类型被自动推导为 `Send` 或 `Sync` ，但是它其实不满足二者的要求。这是我们可以去掉 `Send` 和 `Sync` 的实现：

```
#![feature(option_builtin_traits)]

// 我对于同步的基础类型有着神奇的语义
struct SpecialThreadToken(u8);

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

注意，一个类型自己不可能被不正确地推导为 `Send` 和 `Sync` 。只有当类型和其他的非安全代码一起实现了一些特殊行为时，它才可能成为一个不正确的 `Send` 或 `Sync` 。

大部分使用裸指针的类型都应该把裸指针用一种抽象隐藏起来，以保证类型可以被推导为 `Send` 和 `Sync` 。比如，所有Rust的标准集合类型都是 `Send` 和 `Sync` （在他们包含 `Send` 和 `Sync` 类型的情况下），虽然它们都大量使用了裸指针处理内存分配和复杂的所有权。类似的，大部分这些集合的迭代器也是 `Send` 和 `Sync` ，因为它们的行为很像这些集合的 `&` 或者 `&mut` 。

TODO：更好地解释什么类型可以是 `Send` 和 `Sync` ，什么类型不可以。只考虑数据竞争是不是就足够了呢？

原文链接：https://doc.rust-lang.org/nomicon/atomics.html

原子操作

Rust臭不要脸地抄袭了C11关于原子操作的内存模型。这么做并不是因为这个模型多么的优秀或者易于理解。事实上，这个模型非常的复杂，而且有一些已知的缺陷。不过，所有的原子操作模型其实都不怎么样，我们不得不因此做出一些妥协。至少，这么做可以让我们借鉴当前关于C的研究成果。

在本书中完整地介绍这个模型是不现实的。模型是基于一个让人神经错乱的因果关系图构建的，需要一整本书去实际地理解它。如果你想知道其中的细节，请看C's specification(Section 7.17)。不过，我们还是会尽量涵盖它的基本内容，以及Rust开发者会面对的问题。

C11的内存模型试图同时满足开发者对语义的要求、编译器对优化的要求、还有硬件对千奇百怪混乱状态的要求。而我们只希望能写一段程序做我们想让它做的事情，并且要做得快。是不是很不错？

编译器重排

编译器努力地通过各种复杂的变换，尽可能减少数据依赖和消除死代码。特别是，它可能会彻底改变事件的顺序，或者干脆让某些事件永远不会发生！如果我们写了这样的代码

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

编译器会发现这段程序最好能变成

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

事件的顺序变了，还有一个事件完全消失了。在单线程的情况下，我们不会察觉有什么区别：毕竟代码执行后可以得到和我们期望的完全相同的状态。但如果程序是多线程的，我们可能确实需要在 `y` 被赋值前将 `x` 赋值为1。我们希望编译器能做出这一类优化，因为这可以提升程序的性能。可另一方面，我们还希望我们写的程序能完全按照我们的指令行事。

硬件重排

即使编译器完全明白了我们的意图并且按照我们的期望去工作，硬件还是有可能来找麻烦的。麻烦来自于在内存分层模式下的CPU。你的硬件系统里确实有一些全局共享的内存空间，但是在各个CPU核心看来，这些内存都离得太远，速度也太慢。CPU希望能在它的本地cache里操作数据，只有在cache里没有需要的内存时才委屈地和共享内存打交道。

毕竟，这不就是cache存在的全部意义吗？如果每一次读取cache都要再去检查共享内存看看数据有没有变化，那么cache还有什么价值呢？最终的结果就是，硬件不能保证相同的事件在两个不同的线程里一定有相同的执行顺序。如果要保证这点，我们必须有一些特殊的方法告诉CPU稍微变笨一点。

比如，我们已经成功地让编译器保证下面的逻辑：

```
1.  初始状态: x = 0, y = 1
2.
3.  线程1          线程2
4.  y = 3;        if x == 1 {
5.  x = 1;            y *= 2;
6.                }
```

这段程序实际上有两种可能的结果：

- `y = 3`：线程2在线程1完成之前检查了x的值
- `y = 6`：线程2在线程1完成之后检查了x的值

但是硬件还会创造出第三种状态：

- `y = 2`：线程2看到了 `x = 1`，但是没看到 `y = 3`，接下来用计算结果覆盖了 `y = 3`

不同的CPU提供了不同的保证机制，但是详细区分它们没什么意义。一般来说只需要把硬件分为两类：强顺序的和弱顺序的。最明显的，x86/64平台提供了强顺序保证，而ARM提供弱顺序保证。对于并发编程来说，它们也会导致不同的结果：

- 在强顺序硬件上要求强顺序保证的开销很小，甚至可能为零，因为硬件本身已经无条件提供了强保证。而弱保证可能只能在弱顺序硬件上获得性能优势。
- 在强顺序硬件上要求过于弱的顺序保证有可能也会碰巧成功，即使你的程序是错误的。如果可能的话，在弱保证硬件上测试并发算法。

数据访问

C11的内存模型允许我们接触到程序的因果关系，希望以此满足多个方面的要求。一般来说，就是要确定程序的各个部分以及运行它们的多个线程之前的时间先后关系。在严格的先后关系没有确定的时候，硬件和编译器有足够的空间做一些激进的优化。而关系确定之后，它们的优化就必须很小心了。我们通过“数据访问”和“原子访问”来控制这种关系。

数据访问是程序设计世界的基础。它们都是非同步的，而且编译器可以做出一些激进的优化。尤其是，编译器认定数据访问都是单线程的，所以可以对它随意地重排。硬件也可以把数据访问的重排的结果移植到其他的线程上去，无论结果多么的滞后和不一致都可以。数据访问最严重的问题是，它会导致数据竞争。数据访问对硬件和编译器很友好，但是我们已经看到了编写和它相关的同步程序是十分可怕的。

事实上，它的同步语义太弱了。

只依靠数据访问是不可能写出正确的同步代码的。

原子访问可以告诉硬件和编译器，我们的程序是多线程的。每一个原子访问都关联一种“排序方式”，以确定它和其他访问之间的关系。归根结底，就是告诉编译器和硬件什么是它们不能做的。对于编译器，主要指的是命令的重排。而对于硬件，指的是写操作的结果如何同步到其他的线程。Rust暴露的排序方式包括：

- 顺序一致性(SeqCst)
- 释放(Release)
- 获取(Acquire)
- Relaxed

（注意：我们没有暴露C11的consume顺序）

TODO: negative reasoning vs positive reasoning? TODO: “can’t forget to synchronize”

（译注：不知道*TODO*的都是什么，需要等到*DO*过了之后才能明白）

顺序一致性

顺序一致性是所有排序方式中最强大的，包含了其他所有排序方式的约束条件。直观上看，顺序一致性操作不能被重排：在同一个线程中，SeqCst之前的访问永远在它之前，之后的访问永远在它之后。只使用顺序一致性原子操作和数据访问就可以构建一个无数据竞争的程序，这种程序的好处是它的命令在所有线程上都有着唯一的执行流程。而且这个执行流程又很容易推导：它就是每个线程各自执行流程的交叉。如果你使用更弱的原子排序方式的话，这一点并不一定继续有效。

顺序一致性给开发者的便利并不是免费的。即使是在强顺序平台上，顺序一致性也会产生内存屏障(memory fence)。

事实上，顺序一致性很少是程序正确性的必要条件。但是，如果你对其他内存排序方式模棱两可的话，顺序一致性绝对是你正确的选择。程序执行得稍微慢一点总比执行出错要好！将它变为具有更弱一致性的原子操作也很容易，只要把 `SeqCst` 变成 `Relaxed` 就完工了！当然，证明这种变化的正确性就是另外一个问题了。

获取-释放

获取和释放经常成对出现。它们的名字就提示了它们的应用场景：它们适用于获取和释放锁，确保临界区不会重叠。

直观看起来，acquire保证在它之后的访问永远在它之后。可在它之前的操作却有可能被重排到它后面、类似的，release保证它之前的操作永远在它之前。但是它后面的操作可能被重排到它前面。

当线程A释放了一块内存空间，紧接着线程B获取了同一块内存，这时因果关系就确定了。在A释放之前的所有写操作的结果，B在获取之后都能看到。但是，它们和其他线程之间没有确定因果关系。同理，如果A和B访问的是不同的内存，它们也没有因果关系。

所以，释放-获取的基本用法很简单：你获取一块内存并进入临界区，然后释放内存并离开临界区。比如，一个简单的自旋锁可能是这样的：

```
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？

    // ...用某种方式把锁分发到各个线程...

    // 设置值为true，以尝试获取锁
    while lock.compare_and_swap(false, true, Ordering::Acquire) {}
    // 跳出循环，表明我们获取到了锁！

    // ...恐怖的数据访问...

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

在强顺序平台上，大多数的访问都有释放和获取的语义，释放和获取通常是无开销的。不过在弱顺序平台上不是这样。

Relaxed

Relaxed访问是最弱的。它们可以被随意重排，也没有先后关系。但是Relaxed操作依然是原子的。也就是说，它并不算是数据访问，所有对它的读-修改-写操作都是原子的。Relaxed操作适用于那些你希望发生但又并不特别在意的事情。比如，多线程可以使用Relaxed的fetch_add来增加计数器，如果你不使用计数器的值去同步其他的访问，这个操作就是安全的。

在强顺序平台上使用Relaxed没什么好处，因为它们通常都有释放-获取语义。不过，在弱顺序平台上，Relaxed可以获取更小的开销。

原文链接：https://doc.rust-lang.org/nomicon/vec.html

实战：实现Vec

我们要把所有的内容汇总起来，从头开始写一个 `std::Vec` 。因为所有编写非安全代码的工具都是不稳定的，这个项目只保证短期有效（从Rust 1.9.0开始）。除了分配器API，我们要用到的大部分不稳定代码都尽量保证和最新的形式一致。

不过，如果可能的话，我们会尽量避免使用不稳定代码。特别是，我们不会使用内在函数(intrinsics)，虽然它可以使代码更好更高效，但它是永久不稳定的功能。尽管许多的内在函数已经在一些地方稳固使用了（std::ptr和str::mem使用了很多内在函数）。

也就是说我们的实现不会借助所有可能的优化手段，即使部分手段其实已经比较成熟了。我们还会深入探究种种内在的细节，哪怕实际问题并不需要这样做。

你想要高级的，我们就给你高级的。

原文链接：https://doc.rust-lang.org/nomicon/vec-layout.html

布局

我们先来看看结构体的布局。Vec由三部分组成：一个指向分配空间的指针、空间的大小、以及已经初始化的元素的数量。

简单来说，我们的设计只要这样：

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

这段代码可以通过编译。可不幸的是，它是不正确的。首先，编译器产生的变性过于严格。所以 `&Vev<&'static str>` 不能当做 `&Vev<&'a str>` 使用。更主要的是，它会给drop检查器传递错误的所有权信息,因为编译器会保守地假设我们不拥有任何的值。关于变性和drop检查的细节，请见所有权和生命周期。

.正如我们在所有权一章见到的，当裸指针指向一块我们拥有所有权的位置，我们应该使用 `Unique<T>` 代替 `*mut T` 。尽管Unique是不稳定的，我们尽可能不去使用它。

复习一下，Unique封装了一个裸指针，并且声明它自己：

- 对 `T` 可变
- 拥有类型T的值（用于drop检查）
- 如果 `T` 是Send/Sync，那就也是Send/Sync
- 指针永远不为null（所以`Option>可以做空指针优化）

除了最后一点，其余的我们都可以用稳定的Rust实现：

```
use std::marker::PhantomData;
use std::ops::Deref;
use std::mem;

struct Unique<T> {
    ptr: *const T,            // 使用*const保证变性
    _marker: PhantomData<T>,  // 用于drop检查
}

// 设置Send和Sync是安全地，因为我们是Unique中的数据的所有者
```

```
// Unique<t>好像就是T一样
unsafe impl<T: Send> Send for Unique<T> {}
unsafe impl<T: Sync> Sync for Unique<T> {}

impl<T> Unique<T> {
    pub fn new(ptr: *mut T) -> Self {
        Unique { ptr: ptr, _marker: PhantomData }
    }

    pub fn as_ptr(&self) -> *mut T {
        self.ptr as *mut T
    }
}
```

可是，声明数据不为0的方法是不稳定的，而且短期内都不太可能会稳定下来。s欧意我们还是接受现实，使用比标准库的Unique:

```
#![feature(ptr_internals)]

use std::ptr::{Unique, self};

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

如果你不太在意空指针优化，那么你可以使用稳定代码。但是我们之后的代码会依赖于这个优化去设计。还要注意，调用 `Unique::new` 是非安全的，因为给它传递null属于未定义行为。我们的稳定Unique就不需要让 `new` 是非安全的，因为它没有对于它的内容做其他的保证。

原文链接：https://doc.rust-lang.org/nomicon/vec-alloc.html

内存分配

使用Unique给Vec（以及所有的标准库集合）造成了一个问题：空的Vec不会分配内存。如果既不能分配内存，又不能给 `ptr` 传递一个空指针，那我们在 `Vec::new` 中能做什么呢？好吧，我们就胡乱往Vec里塞点东西。

这么做没什么问题，因为我们用 `cap == 0` 来表示没有分配空间。我们也不用做什么特殊的处理，因为我们通常都会去检查 `cap > len` 或者 `len > 0` 。Rust推荐的放进去的值是 `mem::align_of::<T>()` 。Unique则提供了一个更方便的方式 `Unique::empty()` 。我们会在很多的地方用到 `empty` ，因为有时候我们没有实际分配的内存，而 `null` 会降低编译器的效率。

所以：

```
#![feature(alloc, heap_api)]

use std::mem;

impl<T> Vec<T> {
    fn new -> Self {
        assert!(mem::size_of::<T>() != 0, "还没准备好处理零尺寸类型");
        Vec { ptr: Unique::empty(), len: 0, cap: 0 }
    }
}
```

我们插入了一个assert语句，因为零尺寸类型需要做很多特殊的处理，我们希望以后再讨论这个问题。如果没有assert的话，我们之前的代码会出现很多严重的问题。

接下来我们要讨论在需要内存空间的时候，我们要做些什么。这里我们需要使用其他的heap API。这些API允许我们直接和Rust的分配器（默认是jemalloc）打交道。

我们还需要能够处理内存不足（OOM）的方法。标准库会调用 `std::alloc::oom()` ，而这个函数会调用 `oom` langitem。默认情况下，它就是执行一个非法的CPU指令来中止程序。之所以要终止程序而不是panic，是因为栈展开的过程也可能需要分配内存，而你的分配器早就告诉过你“嘿，我这没有更多的内存了”。

当然，这么做显得有一点傻乎乎，因为大多数平台正常情况下都不会真的没有内存。如果你的程序正常地耗尽了内存，操作系统可能会用其他的方式kill掉它。真的遇到OOM，最有可能的原因是我们一次性的请求严重过量的内存（比如，理论地址空间的一半）。这种情况下其实可以panic而不用担心有什么问题。不过，我们希望尽量模仿标准库的行为，所以我们还是中止整个程序。

好了，现在我们可以编写扩容的代码了。简单粗暴一点，我们需要这样的逻辑：

```
1.  if cap == 0:
2.      allocate()
3.      cap = 1
4.  else:
5.      reallocate()
6.      cap *= 2
```

但是Rust支持的分配器API过于底层了，我们不得不做一些其他的工作。我们还需要应对过大的或者空的内存分配等特殊的场景。

特别是 `ptr::offset` 会给我们造成很多麻烦。因为它的语义是LLVM的GEP inbounds指令。如果你很幸运，以前没有处理过这个语义，这里就简单介绍一下GEP的作用：别名分析，别名分析，别名分析。推导数据依赖和别名对于一个成熟的编译器来说至关重要。

一个简单的例子，看一下下面这段代码：

```
1.  *x *= 7;
2.  *y *= 3;
```

如果编译器可以证明 `x` 和 `y` 指向内存的不同区域，那么这两个操作理论上可以并行执行(比如，把它们加载到不同的寄存器并各自独立地处理)。但一般编译器不能这么做，因为如果x和y指向相同的区域，两个操作是在同一个值上做的，最后的结果不能合并到一起。

如果你使用了GEP inbounds，你其实是在告诉LLVM你的offset操作是在一个分配实体里面做的。LLVM可以认为，当已知两个指针指向不同的对象时，他们所有的offset也都不是重名的（因为它们只能指向某个确定范围内的位置）。LLVM针对GEP offset做了很多的优化，而inbounds offset是效果最好的，所以我们也要尽可能地利用它。

这就是GEP做的事情，那么它怎么会给我们制造麻烦呢？

第一个问题，我们索引数组时使用的是无符号整数，但GEP（其实也就是 `ptr::offset` ）接受的是有符号整数。这表明有一半合法的索引值是超出了GEP的范围的，会指向错误的方向。所以我们必须限制所有的分配空间最多有 `isize::Max` 个元素。这实际意味着我们只需要关心一个字节大小的对象，因为数量 `> isize::MAX` 个 `u16` 会耗尽系统的内存。不过，为了避免一些奇怪的边界场景，比如有人将少于 `isize::MAX` 个对象的数组重解析为字节数组，标准库还限制了分配空间最大为 `isize::MAX` 个字节。

Rust目前支持的各种64位目标平台，都被人为限制了内存地址空间明显小于64位（现代x86平台只暴露了48位的寻址空间），所以我们可以依赖于OOM实现上面的要求。但是对于32位目标平台，特别是那些借助扩展可以使用多于寻址空间的内存的平台（PAE x86或x32），理论上可能成功分配到多

于 `isize::MAX` 字节的内存。

不过因为本书只是一个教程，我们也不必做得绝对完美。这里就使用无条件检查，而不用更智能的平台相关的 `cfg` 。

另一个需要关注的边界场景是空分配。而空分配又分为两种： `cap = 0` ，以及 `cap > 0` 但是类型大小为0。

这些场景的特殊性在于，它们都做了特殊的处理以适配LLVM的“已分配”的概念。LLVM的分配的概念比我们通常的理解要更加抽象。因为LLVM要适配多种语言的语义以及分配器，它其实并不知道什么叫做分配。它所谓的分配的实际含义是“不要和其他的东西重叠”。也就是说，堆分配、栈分配已经全局变量都不能有重合的区域。是的，这就是别名分析。如果Rust和这一概念保持一致的话，理论上可以做到更快更灵活。

回到空分配的场景，代码中许多的地方都可能需要offset 0。现在的问题是：这么做会导致冲突吗？对于零尺寸类型，我们知道它可以做到任意数量的GEP inbounds offset而不会引起任何问题。这实际上是一个运行期的no-op，因为所有的元素都不占用空间，可以假设有无数个零尺寸类型位于 `0x01` 。当然，没有哪个分配器真的会分配那个地址，因为它们不会分配 `0x00` ，而最小的对齐(alignment)通常要大于一个字节。同时，内存的第一页通常处于受保护状态，不会在上面分配空间（对于大多数平台，一页是4k的空间）。

如果是尺寸大于0的类型呢？这种情况就更复杂一些。原则上，你可以认为offset 0不会给LLVM提供任何的信息：地址的前面或后面可能存在一些元素，可不需要知道它们确切是什么。但是，我们还是谨慎一些，假设这么做有可能导致不好的情况。所以我们会显式地避免这种场景。

终于要结束了。

不要再说这些废话了，我们实际写一段内存分配的代码：

```
use std::alloc::oom;

fn grow(&mut self) {
    // 整段代码都很脆弱，所以我们把它整体设为unsafe
    unsafe {
        // 现在的API允许我们手工指定对齐和尺寸
        let align = mem::align_of::<T>();
        let elem_size = mem::size_of::<T>();

        let (new_cap, ptr) = if self.cap == 0 {
            let ptr = heap::allocate(elem_size, align);
            (1, ptr)
        } else {
            // 简单起见，我们假设self.cap < isize::MAX，所以这里不需要做检查
```

```
15.             let new_cap = self.cap * 2;
16.             // 因为之前已经成功分配过了，所以这块不会溢出
17.             let old_num_bytes = self.cap * elem_size;
18.
19.             // 检查新分配的空间不超过isize::MAX，而不管实际的系统容量大小。
20.             // 这里包含了对new_vap<=isize::MAX和new_num_bytes<=usize::MAX的检查
21.             // 我们不能充分利用所有的地址空间。比如，一个i16的Vec在32位平台上，
22.             // 有2/3的地址空间分配不到。这些空间永远地离开了我们。
                // Alas, poor Yorick -- I knew him, Horatio.（译注：《哈姆雷特》中悼念
23. 逝去生命的经典台词）
24.             assert!(old_num_bytes <= (::std::isize::MAX as usize) / 2,
25.                     "capacity overflow");
26.
27.             let new_num_bytes = old_num_bytes * 2;
28.             let ptr = heap::reallocate(self.ptr.as_ptr() as *mut _,
29.                                        old_num_bytes,
30.                                        new_num_bytes,
31.                                        align);
32.             (new_cap, ptr)
33.         };
34.
35.         // 如果分配或者再分配失败，我们会得到null
36.         if ptr.is_null() { oom(); }
37.
38.         self.ptr = Unique::new(ptr as *mut _);
39.         self.cap = new_cap;
40.     }
41. }
```

没有什么特别奇怪的操作。只是计算类型大小和对其，然后小心地做一些乘法检查。

原文链接：https://doc.rust-lang.org/nomicon/vec-push-pop.html

Push和Pop

很好。我们可以初始化，我们也可以分配内存。现在我们开始实现一些真正的功能！我们就从 `push` 开始吧。它要做的事情就是检查空间是否已满，满了就扩容，然后写数据到下一个索引位置，最后增加长度。

写数据时，我们一定要小心，不要计算我们要写入的内存位置的值。最坏的情况，那块内存是一块未初始化的内存。最好的情况是那里存着我们已经pop出去的值。不管哪种情况，我们都不能直接索引这块内存然后解引用它，因为这样其实是把内存中的值当做了一个合法的T的实例。更糟糕的是， `foo[idx] = x` 会调用 `foo[idx]` 处旧有值的 `drop` 方法！

正确的方法是使用 `ptr::write` ，它直接用值的二进制覆盖目标地址，不会计算任何的值。

对于 `push` ，如果原有的长度（调用push之前的长度）为0，那么我们就要写到第0个索引位置。所以我们应该用原有的长度做offset。

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.offset(self.len as isize), elem);
    }

    // 这一句不会失败，而会首先OOM
    self.len += 1;
}
```

小菜一碟！那么 `pop` 是什么样的呢？尽管现在我们要访问的索引位置已经初始化了，Rust不允许我们用解引用的方式将值移出，因为那样的话整个内存都会回到未初始化状态！这时我们需要用 `ptr:read` ，它从目标位置拷贝出二进制值，然后解析成类型T的值。这时原有位置处的内存逻辑上是未初始化的，可实际上那里还是存在这一个正常的T的实例。

对于 `pop` ，如果原有长度是1，我们要读的是第0个索引位置。所以我们应该是按新的长度做offset。

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
```

```
6.          unsafe {
7.              Some(ptr::read(self.ptr.offset(self.len as isize)))
8.          }
9.      }
10. }
```

原文链接：https://doc.rust-lang.org/nomicon/vec-dealloc.html

回收资源

接下来我们应该实现Drop，否则就要造成大量的资源泄露了。最简单的方法是循环调用 `pop` 直到产生None为止，然后再回收我们的缓存。注意，当 `T: !Drop` 的时候，调用 `pop` 不是必须的。理论上我们可以问一问Rust `T` 是不是 `need_drop` 然后再省略一些 `pop` 调用。可实际上LLVM很擅长移除像这样的无副作用的代码，所以我们不需要再做多余的事，除非你发现LLVM不能成功移除（在这里它能）。

在 `self.cap == 0` 的时候，我们一定不要调用 `heap::deallocate` ，因为这时我们还没有实际分配过任何内存。

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(self.ptr.as_ptr() as *mut _, num_bytes,
align);
            }
        }
    }
}
```

原文链接：https://doc.rust-lang.org/nomicon/vec-deref.html

DeRef

不错！我们实现了一个成熟的小的栈。我们可以push、可以pop、也可以自动清理。但是还是有一堆的功能是我们需要的。特别是，我们已经有了一个很好的数组，但是还没有slice相关的功能。这非常容易解决：我们可以实现 `Deref<Target=[T]>` 。这样我们的Vec就神奇地变成了slice。

我们只需要使用 `slice::from_raw_parts` 。它能够为我们正确处理空slice。等到后面我们完成了零尺寸类型的支持，它们依然可以完美配合。

```
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)
        }
    }
}
```

我们把DefMut也实现了吧：

```
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、分片、排序、 `iter` 、 `iter_mut` ，以及其他所有的slice提供的功能。完美！

原文链接：https://doc.rust-lang.org/nomicon/vec-insert-remove.html

插入和删除

slice并没有提供插入和删除功能，接下来我们就实现它们。

插入需要把目标位置后的所有元素都向右移动1。这里我们需要用到 `ptr::copy` ，它就是C中的 `memmove` 的Rust版。它把一块内存从一个地方拷贝到另一个地方，而且可以正确处理源和目标内存区域有重叠的情况（也正是我们这里遇到的情况）。

如果我们在 `i` 的位置插入，我们需要把`[i .. len]`移动到`[i+1 .. len+1]`，len指的是插入前的值。

```
pub fn insert(&mut self, index: usize, elem: T) {
    // 注意：<=是因为我们可以把值插到所有元素的后面
    // 这种情况等同于push
    assert!(index <= self.len, "index out of bounds");
    if self.cap == self.len { self.grow(); }

    unsafe {
        if index < self.len {
            // ptr::copy(src, dest, len): "从src拷贝len个元素到dest"
            ptr::copy(self.ptr.offset(index as isize),
                      self.ptr.offset(index as isize + 1),
                      self.len - index);
        }
        ptr::write(self.ptr.offset(index as isize), elem);
        self.len += 1;
    }
}
```

删除则是完全相反的行为。我们要把元素`[i+1 .. len + 1]`移动到`[i .. len]`，len是删除后的值。

```
pub fn remove(&mut self, index: usize) -> T {
    // 注意：<是因为我们不能删除所有元素之后的位置
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.offset(index as isize));
        ptr::copy(self.ptr.offset(index as isize + 1),
                  self.ptr.offset(index as isize),
```

```
                    self.len - index);
        result
    }
}
```

原文链接：https://doc.rust-lang.org/nomicon/vec-into-iter.html

IntoIter

我们继续编写迭代器。 `iter` 和 `iter_mut` 其实已经写过了，感谢神奇的DeRef。但是还有两个有意思的迭代器是Vec提供的而slice没有的： `into_iter` 和 `drain` 。

IntoIter以值而不是引用的形式访问Vec，同时也是以值的形式返回元素。为了实现这一点，IntoIter需要获取Vec的分配空间的所有权。

IntoIter也需要DoubleEnd，即从两个方向读数据。从尾部读数据可以通过调用 `pop` 实现，但是从头读数据就困难了。我们可以调用 `remove(0)` ，但是它的开销太大了。我们选择直接使用 `ptr::read` 从Vec的两端拷贝数据，而完全不去改变缓存。

我们要用一个典型的C访问数组的方式来实现这一点。我们先创建两个指针，一个指向数组的开头，另一个指向结尾后面的那个元素。如果我们需要一端的元素，我们就从那一端指针指向的位置处读出值，然后把指针移动一位。当两个指针相等时，就说明迭代完成了。

注意， `next` 和 `next_back` 中的读和offset的顺序是相反的。对于 `next_back` ，指针总是指向它下一次要读的元素的后面，而 `next` 的指针总是指向它下一次要读的元素。为什么要这样呢？考虑一下只剩一个元素还未被读取的情况。

这时的数组像这样：

```
1.           S  E
2. [X, X, X, O, X, X, X]
```

如果E直接指向它下一次要读的元素，我们就无法把上面的情况和所有元素都读过了的情况区分开了。

我们还需要保存Vec的分配空间的信息，虽然在迭代过程中我们并不关心它，但我们在IntoIter被drop的时候需要这些信息来释放空间。

所以我们要用下面这个结构体：

```
1. struct IntoIter<T> {
2.     buf: Unique<T>,
3.     cap: usize,
4.     start: *const T,
5.     end: *const T,
6. }
```

这是初始化的代码：

```
impl<T> Vec<T> {
    fn into_iter(self) -> IntoIter<T> {
        // 因为Vec是Drop，不能销毁它
        let ptr = self.ptr;
        let cap = self.cap;
        let len = self.len;

        // 确保Vec不会被drop，因为那样会释放内存空间
        mem::forget(self);

        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: *ptr,
                end: if cap == 0 {
                    // 没有分配空间，不能计算指针偏移量
                    *ptr
                } else {
                    ptr.offset(len as isize)
                }
            }
        }
    }
}
```

这是前向迭代的代码：

```
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)
            }
        }
    }

```

```
15.     fn size_hint(&self) -> (usize, Option<usize>) {
16.         let len = (self.end as usize - self.start as usize)
17.                   / mem::size_of::<T>();
18.         (len, Some(len))
19.     }
20. }
```

这是逆向迭代的代码：

```
1.  impl<T> DoubleEndedIterator for IntoIter<T> {
2.      fn next_back(&mut self) -> Option<T> {
3.          if self.start == self.end {
4.              None
5.          } else {
6.              unsafe {
7.                  self.end = self.end.offset(-1);
8.                  Some(ptr::read(self.end))
9.              }
10.         }
11.     }
12. }
```

因为IntoIter获得了分配空间的所有权，它需要实现Drop来释放空间。同时Drop也要销毁所有它拥有但是没有读取到的元素。

```
1.  impl<T> Drop for IntoIter<T> {
2.      fn drop(&mut self) {
3.          if self.cap != 0 {
4.              // drop剩下的元素
5.              for _ in &mut *self {}
6.
7.              let align = mem::align_of::<T>();
8.              let elem_size = mem::size_of::<T>();
9.              let num_bytes = elem_size * self.cap;
10.             unsafe {
                    heap::deallocate(self.buf.as_ptr() as *mut _, num_bytes,
11. align);
12.             }
13.         }
14.     }
15. }
```

原文链接：https://doc.rust-lang.org/nomicon/vec-raw.html

RawVec

我们遇到了一个很有意思的情况：我们把初始化缓存和释放内存的逻辑在Vec和IntoIter里面一模一样地写了两次。现在我们已经实现了功能，而且发现了逻辑的重复，是时候对代码做一些压缩了。

我们要抽象出 `(ptr, cap)` ，并赋予它们分配、扩容和释放的逻辑：

```
struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO:实现零尺寸类型的支持");
        RawVec { ptr: Unique::empty(), cap: 0 }
    }

    // 与Vec一样
    fn grow(&mut self) {
        unsafe {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(self.ptr.as_ptr() as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // 如果分配或再分配失败，我们会得到null
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
```

```
            self.cap = new_cap;
        }
    }
}


impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(self.ptr.as_mut() as *mut _, num_bytes,
align);
            }
        }
    }
}
```

然后像下面这样改写Vec：

```
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基本没变，只改变了：
    // self.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() {}
        // 释放空间由RawVec负责
    }
}
```

最后我们可以简化IntoIter:

```
struct IntoIter<T> {
    _buf: RawVec<T>, // 我们并不关心这个，只是需要它们保持分配空间不被销毁
    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) {
        // 只需要保证所有的元素都被读到了
        // 缓存会在随后自己清理自己
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // 需要使用ptr::read非安全地把buf移出，因为它不是Copy，
            // 而且Vec实现了Drop（所以我们不能销毁它）
            let buf = ptr::read(&self.buf);
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: *buf.ptr,
                end: buf.ptr.offset(len as isize),
                _buf: buf,
            }
        }
    }
}
```

现在看起来好多了。

原文链接：https://doc.rust-lang.org/nomicon/vec-drain.html

Drain

我们接着看看Drain。Drain和IntoIter基本相同，只不过它并不获取Vec的值，而是借用Vec并且不改变它的分配空间。现在我们只是先最“基本”的全范围(full-range)的版本。

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // 这里需要限制生命周期。我们使用&'a mut Vec<T>, 因为这就是语义上我们包含的东西。
    // 我们只调用pop()和remove(0)
    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
```

—等一下，这个看着有点眼熟。我们需要做进一步的压缩。IntoIter和Drain有着完全一样的结构，我们把它提取出来。

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // 构建它是非安全的，因为它没有关联的生命周期。
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // 如果len == 0, 说明没有真的分配内存。这时需要避免offset,
                // 因为那会给LLVM的GEP提供错误的信息
                slice.as_ptr()
            } else {
```

```
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

// Iterator和DoubleEndedIterator的实现与IntoIter完全一样。
```

IntoIter变成了这样：

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // 我们并不关心这个，只是需要它们保持分配空间不被销毁
    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.iter {}
    }
}

impl<T> Vec<T> {
    pub 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,
```

```
            }
        }
    }
}
```

注意，我在设计中留下了一些小后门，以便更简单地将Drain升级为可访问任意子范围的版本。特别是，我们可以在drop中让RawValIter遍历它自己。但是这种设计不适用于更复杂的Drain。我们还使用一个slice简化Drain的初始化。

好了，现在Drain变得很简单：

```
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.iter {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // 这一步是为了mem::forget的安全。如果Drain被forget，我们会泄露整个Vec的内
容
            // 同时，既然我们无论如何都会做这一步，为什么不现在做呢？
```

```
31.             self.len = 0;
32.
33.             Drain {
34.                 iter: iter,
35.                 vec: PhantomData,
36.             }
37.         }
38.     }
39. }
```

关于更多的 `mem::forget` 的问题，请见关于泄露的章节。

原文链接：https://doc.rust-lang.org/nomicon/vec-zsts.html

处理零尺寸类型

是时候和零尺寸类型开战了。安全Rust并不需要关心这个，但是Vec大量的依赖裸指针和内存分配，这些都需要零尺寸类型。我们要小心两件事情：

- 当给分配器API传递分配尺寸为0时，会导致未定义行为
- 对零尺寸类型的裸指针做offset是一个no-op，这会破坏我们的C-style指针迭代器。

幸好我们把指针迭代器和内存分配逻辑抽象出来放在RawValIter和RawVec中了。真是太方便了。

为零尺寸类型分配空间

如果分配器API不支持分配大小为0的空间，那么我们究竟储存了些什么呢？当然是 `Unique::empty()` 了！基本上所有关于ZST的操作都是no-op，因为ZST只有一个值，不需要储存或加载任何的状态。这也同样适用于 `ptr::read` 和 `ptr::write` ：它们根本不会看那个指针一眼。所以我们并不需要修改指针。

注意，我们之前的分配代码依赖于OOM会先于数值溢出出现的假设，对于零尺寸类型不再有效了。我们必须显式地保证cap的值在ZST的情况下不会溢出。

基于现在的架构，我们需要写3处保护代码，RawVec的三个方法每个都有一处。

```
impl<T> RawVec<T> {
    fn new() -> Self {
        // !0就是usize::MAX。这段分支代码在编译期就可以计算出结果。
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // Unique::empty()有着“未分配”和“零尺寸分配”的双重含义
        RawVec { ptr: Unique::empty(), cap: cap }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // 因为当elem_size为0时我们设置了cap为usize::MAX,
            // 这一步成立意味着Vec的容量溢出了
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();
```

```
19.
20.             let (new_cap, ptr) = if self.cap == 0 {
21.                 let ptr = heap::allocate(elem_size, align);
22.                 (1, ptr)
23.             } else {
24.                 let new_cap = 2 * self.cap;
25.                 let ptr = heap::reallocate(self.ptr.as_ptr() as *mut _,
26.                                            self.cap * elem_size,
27.                                            new_cap * elem_size,
28.                                            align);
29.                 (new_cap, ptr)
30.             };
31.
32.             // 如果分配或再分配失败，我们会得到null
33.             if ptr.is_null() { oom() }
34.
35.             self.ptr = Unique::new(ptr as *mut _);
36.             self.cap = new_cap;
37.         }
38.     }
39. }
40.
41. impl<T> Drop for RawVec<T> {
42.     fn drop(&mut self) {
43.         let elem_size = mem::size_of::<T>();
44.
45.         // 不要释放零尺寸空间，因为它根本就没有分配过
46.         if self.cap != 0 && elem_size != 0 {
47.             let align = mem::align_of::<T>();
48.
49.             let num_bytes = elem_size * self.cap;
50.             unsafe {
                heap::deallocate(self.ptr.as_ptr() as *mut _, num_bytes,
51. align);
52.             }
53.         }
54.     }
55. }
```

就是这样。我们现在已经支持push和pop零尺寸类型了。但是迭代器（slice未提供的）还不能工作。

迭代零尺寸类型

offset 0是一个no-op。这意味着我们的 `start` 和 `end` 总是会被初始化为相同的值，我们的迭代器也无法产生任何的东西。当前的解决方案是把指针转换为整数，增加他们的值，然后再转换回来：

```
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().offset(slice.len() as isize)
            }
        }
    }
}
```

现在我们有了一个新的bug。我们成功地让迭代器从完全不运行，变成了永远不停地运行。我们需要在迭代器的实现中玩同样的把戏。同时， `size_hint` 在ZST的情况下会出现除数为0的问题。因为我们假设这两个指针都指向某个字节，我们在除数为0的情况下直接将除数变为1。

```
impl<T> Iterator for RawValIter<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 = if mem::size_of::<T>() == 0 {
                    (self.start as usize + 1) as *const _
                } else {
                    self.start.offset(1)
                };
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
```

```
20.         let elem_size = mem::size_of::<T>();
21.         let len = (self.end as usize - self.start as usize)
22.                   / if elem_size == 0 { 1 } else { elem_size };
23.         (len, Some(len))
24.     }
25. }
26.
27. impl<T> DoubleEndedIterator for RawValIter<T> {
28.     fn next_back(&mut self) -> Option<T> {
29.         if self.start == self.end {
30.             None
31.         } else {
32.             unsafe {
33.                 self.end = if mem::size_of::<T>() == 0 {
34.                     (self.end as usize - 1) as *const _
35.                 } else {
36.                     self.end.offset(-1)
37.                 };
38.                 Some(ptr::read(self.end))
39.             }
40.         }
41.     }
42. }
```

很好，迭代器也可以工作了。

原文链接：https://doc.rust-lang.org/nomicon/vec-zsts.html

最终代码

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

use std::ptr::{Unique, NonNull, self};
use std::mem;
use std::ops::{Deref, DerefMut};
use std::marker::PhantomData;
use std::alloc::{Alloc, GlobalAlloc, Layout, Global, handle_alloc_error};

struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        // !0就是usize::MAX。这段分支代码在编译期就可以计算出结果。
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // Unique::empty()有着“未分配”和“零尺寸分配”的双重含义
        RawVec { ptr: Unique::empty(), cap: cap }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // 因为当elem_size为0时我们设置了cap为usize::MAX,
            // 这一步成立意味着Vec的容量溢出了
            assert!(elem_size != 0, "capacity overflow");

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = Global.alloc(Layout::array::<T>(1).unwrap());
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let c: NonNull<T> = self.ptr.into();
```

```
38.                 let ptr = Global.realloc(c.cast(),
39.                                          Layout::array::<T>(self.cap).unwrap(),
                                             Layout::array::<T>
40. (new_cap).unwrap().size());
41.                 (new_cap, ptr)
42.             };
43. 
44.             // 如果分配或再分配失败，oom
45.             if ptr.is_err() {
46.                 handle_alloc_error(Layout::from_size_align_unchecked(
47.                     new_cap * elem_size,
48.                     mem::align_of::<T>(),
49.                 ))
50.             }
51.             let ptr = ptr.unwrap();
52. 
53.             self.ptr = Unique::new_unchecked(ptr.as_ptr() as *mut _);
54.             self.cap = new_cap;
55.         }
56.     }
57. }
58. 
59. impl<T> Drop for RawVec<T> {
60.     fn drop(&mut self) {
61.         let elem_size = mem::size_of::<T>();
62.         if self.cap != 0 && elem_size != 0 {
63.             unsafe {
64.                 let c: NonNull<T> = self.ptr.into();
65.                 Global.dealloc(c.cast(),
66.                                Layout::array::<T>(self.cap).unwrap());
67.             }
68.         }
69.     }
70. }
71. 
72. pub struct Vec<T> {
73.     buf: RawVec<T>,
74.     len: usize,
75. }
76. 
77. impl<T> Vec<T> {
78.     fn ptr(&self) -> *mut T { self.buf.ptr.as_ptr() }
```

```
79.
80.     fn cap(&self) -> usize { self.buf.cap }
81.
82.     pub fn new() -> Self {
83.         Vec { buf: RawVec::new(), len: 0 }
84.     }
85.     pub fn push(&mut self, elem: T) {
86.         if self.len == self.cap() { self.buf.grow(); }
87.
88.         unsafe {
89.             ptr::write(self.ptr().offset(self.len as isize), elem);
90.         }
91.
92.         // 不会溢出，会先OOM
93.         self.len += 1;
94.     }
95.
96.     pub fn pop(&mut self) -> Option<T> {
97.         if self.len == 0 {
98.             None
99.         } else {
100.            self.len -= 1;
101.            unsafe {
102.                Some(ptr::read(self.ptr().offset(self.len as isize)))
103.            }
104.        }
105.    }
106.
107.    pub fn insert(&mut self, index: usize, elem: T) {
108.        assert!(index <= self.len, "index out of bounds");
109.        if self.cap() == self.len { self.buf.grow(); }
110.
111.        unsafe {
112.            if index < self.len {
113.                ptr::copy(self.ptr().offset(index as isize),
114.                          self.ptr().offset(index as isize + 1),
115.                          self.len - index);
116.            }
117.            ptr::write(self.ptr().offset(index as isize), elem);
118.            self.len += 1;
119.        }
120.    }
```

```
121.
122.     pub fn remove(&mut self, index: usize) -> T {
123.         assert!(index < self.len, "index out of bounds");
124.         unsafe {
125.             self.len -= 1;
126.             let result = ptr::read(self.ptr().offset(index as isize));
127.             ptr::copy(self.ptr().offset(index as isize + 1),
128.                       self.ptr().offset(index as isize),
129.                       self.len - index);
130.             result
131.         }
132.     }
133.
134.     pub fn into_iter(self) -> IntoIter<T> {
135.         unsafe {
136.             let iter = RawValIter::new(&self);
137.             let buf = ptr::read(&self.buf);
138.             mem::forget(self);
139.
140.             IntoIter {
141.                 iter: iter,
142.                 _buf: buf,
143.             }
144.         }
145.     }
146.
147.     pub fn drain(&mut self) -> Drain<T> {
148.         unsafe {
149.             let iter = RawValIter::new(&self);
150.
                 // 这一步是为了mem::forget的安全。如果Drain被forget，我们会泄露整个Vec的内
151. 容
152.             // 同时，既然我们无论如何都会做这一步，为什么不现在做呢？
153.             self.len = 0;
154.
155.             Drain {
156.                 iter: iter,
157.                 vec: PhantomData,
158.             }
159.         }
160.     }
161. }
```

```
162.
163. impl<T> Drop for Vec<T> {
164.     fn drop(&mut self) {
165.         while let Some(_) = self.pop() {}
166.         // 分配由RawVec负责
167.     }
168. }
169.
170. impl<T> Deref for Vec<T> {
171.     type Target = [T];
172.     fn deref(&self) -> &[T] {
173.         unsafe {
174.             ::std::slice::from_raw_parts(self.ptr(), self.len)
175.         }
176.     }
177. }
178.
179. impl<T> DerefMut for Vec<T> {
180.     fn deref_mut(&mut self) -> &mut [T] {
181.         unsafe {
182.             ::std::slice::from_raw_parts_mut(self.ptr(), self.len)
183.         }
184.     }
185. }
186.
187.
188.
189.
190.
191. struct RawValIter<T> {
192.     start: *const T,
193.     end: *const T,
194. }
195.
196. impl<T> RawValIter<T> {
197.     unsafe fn new(slice: &[T]) -> Self {
198.         RawValIter {
199.             start: slice.as_ptr(),
200.             end: if mem::size_of::<T>() == 0 {
201.                 ((slice.as_ptr() as usize) + slice.len()) as *const _
202.             } else if slice.len() == 0 {
203.                 slice.as_ptr()
```

```
204.             } else {
205.                 slice.as_ptr().offset(slice.len() as isize)
206.             }
207.         }
208.     }
209. }
210.
211. impl<T> Iterator for RawValIter<T> {
212.     type Item = T;
213.     fn next(&mut self) -> Option<T> {
214.         if self.start == self.end {
215.             None
216.         } else {
217.             unsafe {
218.                 let result = ptr::read(self.start);
219.                 self.start = if mem::size_of::<T>() == 0 {
220.                     (self.start as usize + 1) as *const _
221.                 } else {
222.                     self.start.offset(1)
223.                 };
224.                 Some(result)
225.             }
226.         }
227.     }
228.
229.     fn size_hint(&self) -> (usize, Option<usize>) {
230.         let elem_size = mem::size_of::<T>();
231.         let len = (self.end as usize - self.start as usize)
232.                   / if elem_size == 0 { 1 } else { elem_size };
233.         (len, Some(len))
234.     }
235. }
236.
237. impl<T> DoubleEndedIterator for RawValIter<T> {
238.     fn next_back(&mut self) -> Option<T> {
239.         if self.start == self.end {
240.             None
241.         } else {
242.             unsafe {
243.                 self.end = if mem::size_of::<T>() == 0 {
244.                     (self.end as usize - 1) as *const _
245.                 } else {
```

```
                    self.end.offset(-1)
                };
                Some(ptr::read(self.end))
            }
        }
    }
}



pub struct IntoIter<T> {
    _buf: RawVec<T>, // 我们并不关心这个，只是需要它们保持分配空间不被销毁
    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;
```

```
288.     fn next(&mut self) -> Option<T> { self.iter.next() }
289.     fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
290. }
291.
292. impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
293.     fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
294. }
295.
296. impl<'a, T> Drop for Drain<'a, T> {
297.     fn drop(&mut self) {
298.         // pre-drain the iter
299.         for _ in &mut self.iter {}
300.     }
301. }
302.
303. # fn main() {
304. #     tests::create_push_pop();
305. #     tests::iter_test();
306. #     tests::test_drain();
307. #     tests::test_zst();
308. #     println!("All tests finished OK");
309. # }
310.
311. # mod tests {
312. #     use super::*;
313. #     pub fn create_push_pop() {
314. #         let mut v = Vec::new();
315. #         v.push(1);
316. #         assert_eq!(1, v.len());
317. #         assert_eq!(1, v[0]);
318. #         for i in v.iter_mut() {
319. #             *i += 1;
320. #         }
321. #         v.insert(0, 5);
322. #         let x = v.pop();
323. #         assert_eq!(Some(2), x);
324. #         assert_eq!(1, v.len());
325. #         v.push(10);
326. #         let x = v.remove(0);
327. #         assert_eq!(5, x);
328. #         assert_eq!(1, v.len());
329. #     }
```

```
#
#     pub fn iter_test() {
#         let mut v = Vec::new();
#         for i in 0..10 {
#             v.push(Box::new(i))
#         }
#         let mut iter = v.into_iter();
#         let first = iter.next().unwrap();
#         let last = iter.next_back().unwrap();
#         drop(iter);
#         assert_eq!(0, *first);
#         assert_eq!(9, *last);
#     }
#
#     pub fn test_drain() {
#         let mut v = Vec::new();
#         for i in 0..10 {
#             v.push(Box::new(i))
#         }
#         {
#             let mut drain = v.drain();
#             let first = drain.next().unwrap();
#             let last = drain.next_back().unwrap();
#             assert_eq!(0, *first);
#             assert_eq!(9, *last);
#         }
#         assert_eq!(0, v.len());
#         v.push(Box::new(1));
#         assert_eq!(1, *v.pop().unwrap());
#     }
#
#     pub fn test_zst() {
#         let mut v = Vec::new();
#         for _i in 0..10 {
#             v.push(())
#         }
#
#         let mut count = 0;
#
#         for _ in v.into_iter() {
#             count += 1
#         }
```

```
372.  #
373.  #         assert_eq!(10, count);
374.  #     }
375.  # }
```

原文链接：https://doc.rust-lang.org/nomicon/arc-and-mutex.html

实现Arc和Mutex

知道理论很不错，但是理解一个东西最好的方法是使用它。为了更好地理解原子操作和内部可变性，我们要实现标准库的Arc和Mutex类型。

TODO：所有的内容，我的天呐……

原文链接：https://doc.rust-lang.org/nomicon/ffi.html

外部函数接口(FFI)

介绍

这个教程会使用snappy压缩/解压缩库来介绍外部代码绑定的编写方法。Rust目前还不能直接调用C++的库，但是snappy有C的接口（文档在 `snappy-c.h` 中）。

关于libc的说明

接下来很多的例子会使用 `libc` `crate`，它为我们提供了很多C类型的定义。如果你要亲自尝试一下这些例子的话，你需要把 `libc` 添加到你的 `Cargo.toml` ：

```
[dependencies]
libc = "0.2.0"
```

然后在你的crate的根文件插入一句 `extern crate libc;`

调用外部函数

下面是一个调用外部函数的小例子，安装了snappy才能编译成功。

```
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_mx_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` 代码块中是外部库的函数签名的列表，这个例子中使用的是平台相关的C的ABI。 `#[link(...)]` 属性用来构建一个链接snappy库的链接器，以便解析库中的符号(symbol)。

外部函数都被认为是不安全的，所以对它们的调用必须包装在 `unsafe {}` 中，也就是向编译器承诺块中的代码都是安全的。C的库经常暴露非线程安全的接口，而且几乎所有的接受指针参数的函数都是

不合法的，因为指针可能是悬垂指针，而裸指针不符合Rust的内存安全模型。

在声明外部函数的参数类型时，Rust编译器不能检查声明的正确性，所以我们需要自己保证它是正确的，这也是运行期正确绑定的条件之一。

`extern` 块还可以继续扩展，包含所有的snappy API：

```
1.  extern crate libc;
2.  use libc::{c_int, size_t};
3.
4.  #[link(name = "snappy")]
5.  extern {
6.      fn snappy_compress(input: *const u8,
7.                         input_length: size_t,
8.                         compressed: *mut u8,
9.                         compressed_length: *mut size_t) -> c_int;
10.     fn snappy_uncompress(compressed: *const u8,
11.                          compressed_length: size_t,
12.                          uncompressed: *mut u8,
13.                          uncompressed_length: *mut size_t) -> c_int;
14.     fn snappy_max_compressed_length(source_length: size_t) -> size_t;
15.     fn snappy_uncompressed_length(compressed: *const u8,
16.                                   compressed_length: size_t,
17.                                   result: *mut size_t) -> c_int;
18.     fn snappy_validate_compressed_buffer(compressed: *const u8,
19.                                          compressed_length: size_t) -> c_int;
20. }
```

创建安全接口

原生的C API进行封装，以保证内存安全，还有使用vector等高级概念。库可以选择只暴露安全的、高级的接口，并隐藏非安全的内部细节。

我们使用 `slice::raw` 模块封装接受内存块的函数，这个模块会把Rust的vector转换为内存的指针。Rust的vector是一块连续的内存。它的长度是当前包含的元素的数量，容量是分配内存可存储的元素的总数。长度是小于等于容量的。

```
1.  pub fn validate_compressed_buffer(src: &[u8]) -> bool {
2.      unsafe {
            snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) ==
3.  0
4.      }
```

```
}
```

上方的 `validate_compressed_buffer` 包装器用到了 `unsafe` 代码块，但是函数签名里没有 `unsafe` 关键字，这说明它保证函数调用对所有的输入都是安全的。

`snappy_compress` 和 `snappy_uncompress` 函数更复杂一些，因为它们需要分配一块空间储存输出的结果。

`snappy_max_compressed_length` 函数可以用来分配一段最大容积内的vector，以保存输出的结果。这个vector可以传递给 `snappy_compress` 函数作为输出参数。还会传递一个输出参数获取压缩后的真实长度，以便设置返回值的长度。

```
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的压缩格式中保存了未压缩时的大小，函数 `snappy_uncompressed_length` 可以获取需要的缓存区的尺寸。

```
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
        }
    }
}
```

接下来，我们添加一些测试用例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));
    }
}
```

析构函数

外部库经常把资源的所有权返还给调用代码。如果是这样，我们必须用Rust的析构函数保证所有的资源都被释放了（特别是在panic的情况下）。

更多关于析构函数的内容，请见Drop trait。

C代码到Rust函数的回调

一些外部库需要用到回调向调用者报告当前状态或者中间数据。我们是可以把Rust写的函数传递给外部库的。要求是回调函数必须标为 `extern` 并遵守正确的调用规范，以保证C代码可以调用它。

然后回调函数会通过注册调用传递给C的库，并在外部库中被触发。

下面是一个简单的例子。

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;
}
```

```
8.
9.  void trigger_callback() {
10.   cb(7); // Will call callback(7) in Rust.
11. }
```

这个例子中，Rust的 `main()` 要调用C的 `trigger_callback()` ，而这个函数会反过来调用Rust中的 `callback()` 。

将Rust对象作为回调

之前的例子演示了C代码如何调用全局函数。但是很多情况下回调也可能是一个Rust对象，比如说封装了某个C的结构体的Rust对象。

要实现这一点，我们可以传递一个指向这个对象的裸指针给C的库。C的库接下来可以将指针转换为Rust的对象。这样回调函数就可以非安全地访问相应的Rust对象了。

```
1.  #[repr(C)]
2.  struct RustObject {
3.      a: i32,
4.      // 其他成员……
5.  }
6.
7.  extern "C" fn callback(target: *mut RustObject, a: i32) {
8.      println!("I'm called from C with value {0}", a);
9.      unsafe {
10.         // 用回调函数接收的值更新RustObject的值：
11.         (*target).a = a;
12.     }
13. }
14.
15. #[link(name = "extlib")]
16. extern {
17.    fn register_callback(target: *mut RustObject,
18.                         cb: extern fn(*mut RustObject, i32)) -> i32;
19.    fn trigger_callback();
20. }
21.
22. fn main() {
23.     // 创建回调用到的对象：
24.     let mut rust_object = Box::new(RustObject { a: 5 });
25.
26.     unsafe {
```

```
27.         register_callback(&mut *rust_object, callback);
28.         trigger_callback();
29.     }
30. }
```

C代码:

```
1.  typedef void (*rust_callback)(void*, int32_t);
2.  void* cb_target;
3.  rust_callback cb;
4.
5.  int32_t register_callback(void* callback_target, rust_callback callback) {
6.      cb_target = callback_target;
7.      cb = callback;
8.      return 1;
9.  }
10.
11. void trigger_callback() {
12.   cb(cb_target, 7); // 调用Rust的callback(&rustObject, 7)
13. }
```

异步回调

上面给出的例子里，回调都是外部C库的直接的函数调用。当前线程的控制权从Rust转移到C再转移回Rust，不过最终回调都是在调用触发回调的函数的线程里执行的。

如果外部库启动了自己的线程，并在那个线程里调用回调函数，情况就变得复杂了。这时再访问回调中的Rust数据结构是非常不安全的，必须使用正常地同步机制。除了Mutex等传统的同步机制，还有另一个选项就是使用channel（在 `std::sync::mpsc` 中）将数据从触发回调的C线程传送给一个Rust线程。

如果一个异步回调使用了一个Rust地址空间里的对象，一定要注意，在这个对象销毁之后C的库不能再调用任何的回调。我们可以在对象的析构函数里注销回调，并且重新设计库确保毁掉注销后就不会被调用了。

链接

`extern` 代码块上的 `link` 属性用于指导rustc如何链接到一个本地的库。现在 `link` 属性有两种可用的形式:

- `#[link(name = "foo")]`

- `#[link(name = "foo", kind = "bar")]`

两种形式中，`foo` 都是我们要链接的本地库的名字。而第二种形式中的 `bar` 是要链接的本地库的类型。目前有三种已知的本地库类型：

- 动态 - `#[link(name = "readline")]`
- 静态 - `#[link(name = "my_build_dependency", kind = "static")]`
- 框架 - `#[link(name = "CoreFundation", kind = "framework")]`

注意，框架只适用于MacOS平台。

不同的 `kind` 表明本地库以不同的方式参与链接。从链接器的角度看，Rust编译器产生两种输出结果：部分结果(rlib/staticlib)和最终结果(dylib/binary)。本地动态库和框架依赖可以被最终结果使用，而静态库则不会，因为静态库是直接集成在接下来的输出里的。

举几个这个模型用法的例子：

- 本地构建依赖。有时候编写Rust代码需要一些C/C++作为补充，但是把C/C++代码以一个库的形式发布却不容易。这种情况下，代码应该包装在 `libfoo.a` 中，然后Rust的crate会声明一个依赖 `#[link(name = "foo", kind = "static")]` 。不管crate最终以哪种形式输出，本地静态库都会被包含在输出中，这表明发布静态库并不必要。

- 普通动态库。通用的系统库（比如 `readline` ）在许多系统中都支持，而我们经常遇到找不到库的本地备份的的情况。如果这样的依赖被包含在Rust的crate中，部分结果（比如rlib）不会链接到这个库中。但是如果rlib被最终结果包含了，本地库也会被链接。

在MacOS中，框架和动态库具有相同的语义。

非安全代码块

有一些操作，比如解引用裸指针、或者调用被标为unsafe的函数，它们只能存在于非安全代码块中。非安全代码块隔离了非安全性，并向编译器承诺非安全性不会影响到块以外的代码。

非安全函数则不同，它们声明非安全性一定会影响到函数之外。一个非安全函数写法如下：

```
1.  unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

这个函数只能在 `unsafe` 代码块或者另外一个 `unsafe` 函数里被调用。

访问外部全局变量

外部API经常暴露一些全局变量，用于记录全局状态等。为了访问这些变量，你需要在 `extern` 块中用 `static` 关键字声明它们：

```
extern crate libc;

#[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 });
}
```

有时也可能需要通过外部的接口修改全局状态。如果要这么做，静态变量还要添加 `mut` ，让我们可以修改它们。

```
extern crate libc;

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，所有和 `static mut` 的操作都是非安全的，不管是读还是写。处理全局可变状态的时候一定要格外的小心。

外部调用规范

大多数外部代码都暴露C的ABI，而Rust默认根据平台相关的C的调用规范调用外部函数。还有一些外部函数使用其他的规范，最典型的就是WindowsAPI。Rust也有方法告诉编译器使用哪种规范：

```
extern crate libc;

#[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;
}
```

这段代码作用于整个 `extern` 代码块。支持的ABI包括：

- `stdcall`
- `appcs`
- `cdecl`
- `fastcall`
- `vectorcall` 这个目前被 `abi_vectorcall` 隐藏着，不允许修改。
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`
- `sysv64`

列表中所有的abi都是自解释的，但是 `system` 可能会显得有些奇怪。它的意思是选择一个合适的与目标库通信的ABI。比如，在win32的x86架构上，它实际使用的是 `stdcall` 。而在x86_64上，Windows使用 `C` 调用规范，所以它实际使用的是 `C` 。这意味着在我们之前的例子中，我们可以使用 `extern "system" { ... }` 为所有的Windows系统定义块，而不仅仅是x86的平台。

与外部代码互用性

只有给一个结构体指定了 `#[repr(C)]` ，Rust才保证结构体的布局与平台的C的表示方法相兼容。 `#[repr(C, packed)]` 可以让结构体成员之间无填充。 `#[repr(C)]` 也可以作用于枚举类型。

Rust的 `Box<T>` 用一个非空的指针指向它包含的对象。但是，这些指针不能手工创建，而是要由内部分配器去管理。引用可以安全地等同于非空指针。不过，违背借用检查和可变性规则就不能保证是安全的了，所以在需要使用指针的地方我们尽量使用裸指针，因为编译器不会对它做过多的限制。

Vector和String拥有相同的内存布局，而且 `vec` 和 `str` 模块里也有一些与C API相关的工具。

但是，字符串不是以 `\0` 结尾的。如果你想要一个与C兼容的Null结尾的字符串，你应该使用 `std::ffi` 模块中的 `CString` 类型。

[crate.io的 `libc` crate] `(https://crates.io/crates/libc)在` libc `模块中包含了C标准库的类型别名和函数定义，而Rust默认链接` libc `和` libm`。

可变函数

在C中，函数可以是“可变的”，也就是说可以接收可变数量的参数。在Rust中可以在外部函数声明的参数类表中插入 `...` 实现这一点：

```
1.  extern {
2.      fn foo(x: i32, ...);
3.  }
4.
5.  fn main() {
6.      unsafe {
7.          foo(10, 20, 30, 40, 50);
8.      }
9.  }
```

普通的Rust函数不能是可变的：

```
1.  // 这段不能通过编译
2.  fn foo(x: i32, ...) { }
```

空指针优化

一些Rust类型被定义为永不为 `null` ，包括引用（ `&T` 、 `&mut T` ）、 `Box<T>` 、以及函数指针（ `extern "abi" fn()` ）。可是在使用C的接口时，指针是经常可能为 `null` 的。看起来似乎需要用到 `transmute` 或者非安全代码来处理各种混乱的类型转换。但是，Rust其实提供了另外的方法。

一些特殊情况中， `enum` 很适合做空指针优化，只要它包含两个变量，其中一个不包含数据，而另外一个包含一个非空类型的成员。这样就不需要额外的空间做判断了：给那个包含非空成员的变量传递一个 `null` ，用它来表示另外那个空的变量。这种行为虽然被叫做“优化”，但是和其他的优化不同，它只适用于合适的类型。

最常见的受益于空指针优化的类型是 `Option<T>` ，其中 `None` 可以用 `null` 表示。所以 `Option<extern "C" fn(c_int) - > c_int>` 就很适合表示一个使用C ABI的可为空的函数指针

（对应于C的 `int (*)(int)` ）。

下面是一个刻意造出来的例子。假设一些C的库提供了注册回调的方法，然后在特定的条件下调用回调。回调接受一个函数指针和一个整数，然后用这个整数作为参数调用指针指向的函数。所以我们会向FFI边界的两侧都传递函数指针。

```
extern crate libc;
use libc::c_int;

extern "C" {
    // 注册回调。
    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));
    }
}
```

C的代码是像这样的：

```
void register(void (*f)(void (*)(int), int)) {
    ...
}
```

看，并不需要 `transmute` ！

C调用Rust

你可能想要用某种方式编译Rust，让C可以直接调用它。这件事很简单，只需要做少数的处理：

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
```

`extern` 让它对应的函数符合C的调用规范，在上面的外部调用规范一节有详细讨论。 `no_mangle` 属性关闭Rust的name mangling，让它更方便被链接。

FFI和panic

使用FFI的时候要格外注意 `panic!` 。跨越FFI边界的 `panic!` 属于未定义行为。如果你写的代码可能会panic，你应该使用 `catch_unwind` 在一个闭包里执行它：

```
use std::panic::catch_unwind;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let result = catch_unwind(|| {
        panic!("Oops!");
    });
    match result {
        Ok(_) => 0,
        Err(_) => 1,
    }
}

fn main() {}
```

请注意， `catch_unwind` 只能捕获可展开的panic，不能捕获abort。更多的信息请参考 `catch_unwind` 的文档。

表示不透明结构体

有时候，C的库要提供一个指针指向某个东西，但又不想让你知道那个东西的内部细节。最简单的方式是使用 `void *` ：

```
void foo(void *arg);
void bar(void *arg);
```

在Rust中我们可以用 `c_void` 类型表示它：

```
extern crate libc;

extern "C" {
    pub fn foo(arg: *mut libc::c_void);
    pub fn bar(arg: *mut libc::c_void);
}
```

这是一个完全合法的方法。不过，我们其实还可以做得更好。要解决这个问题，一些C库可能会创建一个结构体，可结构体的细节和内存布局是私有的。这样提高了类型的安全性。这种结构体被称为”不透明“的。下面是一个C的例子：

```
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);
```

在Rust中，我们可以使用枚举来创建我们自己的不透明类型：

```
#[repr(C)] pub struct Foo { _private: [u8; 0] }
#[repr(C)] pub struct Bar { _private: [u8; 0] }

extern "C" {
    pub fn foo(arg: *mut Foo);
    pub fn bar(arg: *mut Bar);
}
# fn main() {}
```

给结构体一个私有成员而不给它构造函数，这样我们就创建了一个不透明的类型，而且我们不能在模块之外实例化它。（没有成员的结构体可以在任何地方实例化）因为我们希望在FFI中使用这个类型，我们必须加上 `#[repr(C)]` 。还为了避免在FFI中使用 `()` 的时候出现警告，我们用了一个空数组。空数组和空类型的行为一致，同时它还是FFI兼容的。

但因为 `Foo` 和 `Bar` 是不同的类型，我们需要保证两者之间的类型安全性，所以我们不能把 `Foo` 的指针传递给 `bar()` 。

注意，用空枚举作为FFI类型是一个很不好的设计。编译器将空枚举视为不可达的空类型，所以使用 `&Empty` 类型的值是很危险的，这可能导致很多程序中的问题（触发未定义行为）。