目录

痛点

可以说 Rust 是一门比大多数”主流”语言更难学的语言。 有特别的人不觉得这么难，但要注意’特别’的真正含义 - 他们是 *例外的*。 许多人先要挣扎一番，后才成功。 最初的艰难是不能预测你的未来!

我们来自世界各地，处于各种编程语言的情况下，这意味着，存在以前主流语言的遗留思维，如 Python 之类的”动态”语言 或 C ++之类的”静态”语言，或其他的。 但无论你过去的思维方式是怎么样的， Rust 都有很大的不同，需要转变思路。有经验的聪明人加入 Rust 学习，觉得说，以他们的聪明才智，却不能立即获得回报，他们会感到失望; 自我认识较低的人则认为自己不够”聪明”。

对于那些具有动态语言经验的人 (包括 Java，我想)，所有的一切都是一个`引用&`，并且所有引用默认都是可变的。还有垃圾收集功能 *确实* 让编写内存安全的程序更容易。 而以内存成本和可预测性为代价，JVM 进展非常迅速。 通常这种成本被认为是值得的 - 传统的新想法认为程序员的生产力比 计算机的性能更重要。

但，世界上大多数电脑 - 如处理汽车阀门控制等之类的真正重要事情 - 并不具备大量资源，甚至连一个便宜笔记本电脑都比不上，和他们需要的是 *实时* 响应。同样，基础软件基础架构需要正确，稳健和快速 (旧工程的三体)。 而这大部分，都是本质上不安全的 C 和 C ++ 完成的，这个不安全的 *总成本* 应该是我们所要正视的。也许你组合项目起来，飞快，但 *在这之后* 真正的开发才刚刚开始。

系统语言无法承担垃圾回收，因为它们是其他所有东西依赖的基础。只要你认为合适，他们就让你自由地浪费资源。

但如果没有垃圾回收，内存就必须以其他方式进行管理。 *手动内存管理* - 我抓住内存，使用它，并明确地将其退回 - 很难做对。 您可以在几周内学会够用的 C 语言，以提高工作效率，而危险性也随之而来 - 要成为一名安全的 C 语言程序员需要花费数年时间，并检查每种可能的错误情况。

Rust 像现代 C ++ 一样管理内存 - 随着对象被破坏，其内存被回收。 你可以在堆上分配内存`Box`，但只要在函数结束，Box'超出范围'时，弃内存就会被回收。所以 Rust 有像`new`这样的事情，但没有`删除{delete}`。 你可以创建一个`File`和在最后，文件 (一个宝贵的资源)会自动被关闭。 在 Rust 中，这被称为 *扔掉{dropping}*。

你需要共享资源 - 复制一切都是非常低效的 - 这就是事情变得有趣的地方。 C ++也有引用，尽管 Rust 引用更像 C 指针 - 你需要使用`*r`才能用引用指向的值 {value}，你需要加上`&`，一个值才作为引用类型传递。

Rust 的 *借用检查器* 确保在原始值被销毁后，引用不可能存在。

类型推断

"静态"和"动态"之间的区别不是一切。 与大多数事情一样，还有很多可以发挥的区域。 C 是静态类型的 (每个变量在编译时，都有一个类型)，但弱类型 (例如，`void*`可以指向 *任何{anything}*); Python 是动态类型的 (类型与值相关，而不是变量)，但却是强类型的。 Java 是静态/非常强类型的 (有反射「reflection」功能，这就像方便，但危险的阀门)， Rust 是静态/强类型的，运行时没有反射。

Java 因需要在麻木细节中， *键入* 所有的类型而出名, Rust 更喜欢 *推断* 类型。 这通常是一个好主意，但这也确实意味着，有时你需要计算出实际类型。 当会看见`let n = 100`，并想知道 - 这是什么样的整数? 默认情况下，它会是`i32` - 一个四字节有符号整数。 现在大家都同意 C 的未指定整数类型 (比如`int`和`long`) 是一个坏主意; 最好明确类型。 你可以随时拼写出类型，如`let n: u32 = 100`，或者强制类型，如`let n = 100u32`。 但是类型推断比这要强! 如果你声明`let n = 100`，然后`rustc`全部知道，`n`一定是 *一些* 整数类型。 如果之后，你想把`n`传递到一个函数，其期望一个`u64`类型，那么这一定是这种类型的`n`!

之后，你尝试给`n`到期望`u32`函数，`rustc`不会让你这样做，因为`n`已被束缚到`u64`，和它 *将不会* 采取简单，且隐式的方法为您转换该整数。这是来自类型的强力攻势 - 没有任何一点转换，和'促销活动'让你的生活更流畅，这样做的同时，整数不会溢出突然咬住你的屁股。你必须明确地表明`n`转换，如`n as u32` - 一个 Rust 类型。 幸好，`rustc`善于以"可行"的方式打败坏家伙 - 也就是说，您可以按照编译器的意见来解决问题.

所以， Rust 代码可以非常明确的类型:

```
let mut v = Vec::new();
// v 被推断为  Vec<i32>类型
v.push(10);
v.push(20);
v.push("hello") <--- 不能这样做，盆友!
```

不能将字符串放入一个整数 Vec 是一个功能，而不是一个错误。 动态类型的灵活性也是一个诅咒。

(如果你将需要 *把* 整数和字符串放入同一个 Vec，那么 Rust `枚举` 类型是安全地使用它的方法。)

有时，你需要至少给一个类型 *暗示*. `collect` 是一个梦幻般的迭代器方法，但它需要一个提示。比如说，我有一个迭代器，它返回 `char` ，然后 `collect` 可有两种方式:

```
// 一个  char vec ['h','e','l','l','o']
let v: Vec<_> = "hello".chars().collect();
// 一个 "doy" 字符串
let m: String = "dolly".chars().filter(|&c| c != 'l').collect();
```

当对某个变量的类型感到不确定时，总会有能，强行让 `rustc` 在错误消息中显示实际类型名称的技巧:

```
let x: () = var;
```

`rustc` 可能会选择十分特定类型。这里我们想把不同的引用放入一个 Vec，但需要使用 `&Debug` 明确声明类型。

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

let answer = 42;
let message = "hello";
let float = 2.7212;

let display: Vec<&Debug> = vec![&message, &answer, &float];

for d in display {
    println!("got {:?}", d);
}
```

可变引用

规则是: 一次只有一个可变引用。 原因在于，当 *到处都是* 都是可变性引用，那跟踪他们就很难。在笨蛋小程序中不明显，但在大型代码库中可能会变得糟糕。

进一步的限制是，当已有一个可变引用时，你不能再拥有不可变引用， 否则，任何有这些引用的人都不能保证他们不会改变。 C ++也有不可变的引用 (例如 `const string&`) ，但是 *不能* 给你这个保证，因为有人可能在你背后，保留一个 `string&` 引用并修改它。

如果您习惯每个引用都是可变的语言，那这会是一个挑战! 不安全的"放松"语言，取决于人们了解他们自己的计划，并秉直地决定不做坏事。 但是大型项目是由不止一个人编写的，并且超出了个人，详细理解的能力。

更气人 事情是，借用检查器并不像它所描述的那样聪明。

```
let mut m = HashMap::new();
m.insert("one", 1);
m.insert("two", 2);

if let Some(r) = m.get_mut("one") { // <-- m 的可变引用
    *r = 10;
} else {
    m.insert("one", 1); // 不能再次可变借用!
}
```

显然这不是 *真的* 违反规则，除非如果我们得到了`None`，而实际上并没有从 map 上借用任何东西。

有各种丑陋的解决方法:

```
let mut found = false;
if let Some(r) = m.get_mut("one") {
    *r = 10;
    found = true;
}
if ! found {
    m.insert("one", 1);
}
```

这很糟糕，但它起作用，因为烦人的借用保留在第一个 if 语句中。

这里更好的方法是使用`HashMap`的entry API.

```
use std::collections::hash_map::Entry;

match m.entry("one") {
    Entry::Occupied(e) => {
        *e.into_mut() = 10;
    },
    Entry::Vacant(e) => {
        e.insert(1);
    }
};
```

当 *非词汇生命周期* 在今年(2018)某个时候到达, 借用检查器获得更少的挫败感。

借用检查器 *还是* 了解一些重要的案例，然而。 如果你有一个结构，字段可以独立借用。所以组合构造是你的朋友; 一个大结构体应该包含更小的结构体，它们有自己的方法。 定义大结构体上的所有可变方法，会导致你不能修改内容的情况，即使这些方法可能只涉及一个字段。

对于可变数据，有一些独立处理部分数据的特别方法。例如，如果你有一个可变切片，那么`split_at_mut`将它分成两个可变切片。 这是完全安全的，因为 Rust 知道切片不重叠。

引用和生命周期

Rust 不能允许一个引用长命过值的情况。 否则，我们会有一个"悬挂引用"，它指的是一个已死亡的值 - 一个错误是不可避免的。

`rustc`往往可以对函数的生命周期做出合理的假设:

```
fn pair(s: &str, ch: char) -> (&str, &str) {
    if let Some(idx) = s.find(ch) {
        (&s[0..idx], &s[idx+1..])
    } else {
        (s, "")
    }
}
fn main() {
    let p = pair("hello:dolly", ':');
    println!("{:?}", p);
}
// ("hello", "dolly")
```

这是非常安全的，因为我们处理好了未找到分隔符的情况。 `rustc`在这里假定元组中的两个字符串都是，从作为一个传递给函数参数的字符串中借用的。

明确地说，函数定义如下所示:

```
fn pair<'a>(s: &'a str, ch: char) -> (&'a str, &'a str) {...}
```

`'a`符号表示输出字符串活得 *至少与输入字符串一样长* 。这并不是说一样的生命周期，我们可以在任何时候放弃它们(引用)，只是它们无法离开`s`。

所以， `rustc`用 *生命周期免写*，使常见案例更漂亮。

现在，如果该函数收到 *两个* 字符串，那么您需要明确地进行生命周期注释，来告诉 Rust 哪个输出字符串是从哪个输入字符串中借用的。

当一个结构借用一个引用时，你总是需要一个明确的生命周期:

```
struct Container<'a> {
    s: &'a str
}
```

这再次坚称，结构不能长命过引用。 对于结构和函数，生命周期都需要在`<>`中声明，当作一个类型参数。

闭包是非常方便和强大的功能 - Rust 迭代器的很多强大之处都来自它们。但是如果你存储它们，你必须指定一个生命周期。 这是因为闭包基本上是一个可以调用，已生成的结构，并且默认情况下，是借用它的环境。这里的`linear`闭包有不可变的引用`m`和`c`。

```
let m = 2.0;
let c = 0.5;

let linear = |x| m*x + c; // 借用/引用
let sc = |x| m*x.cos()
...
```

`linear`和`sc`都为`Fn(x: f64)->f64`类型，但他们是 *不是* 同类 - 他们有不同的类型和大小! 所以要存储它们，你必须做出一个`Box<Fn(x: f64)->f64 + 'a>`。

非常烦人，如果你习惯了 JavaScript 或 Lua 的流畅闭包，但 C ++与 Rust 类似，同样需要`std::function`存储不同的闭包，给虚拟调用一点点惩罚。

字符串

在开始时，经常会对 Rust 字符串感到恼火。 有不同的方式来创建它们。但感觉它们都冗长:

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

“hello” 不是 *已经是* 一个字符串? 好吧，在某种程度上。 `String`是一个具有 *所有权* 字符串，分配在堆上; 字符串字面量”hello”是`&str`类型的 (”字符串切片”) ，并可能被烘焙到可执行文件 (”静态”) 或从一个`String`借用而来的。 系统语言需要这种区别 - 考虑一个微型微控制器，它有 一点 RAM 和更多的 ROM 。 字面字符串将被存储在 ROM (”只读”) 中 ，这既便宜又更少功耗。

但是 (你可能会说) 在 C ++中，它非常简单啊:

```
std::string s = "hello";
```

是短，但字符串对象真正的创建，被隐藏起来了。 因此, Rust 喜欢`to_string`明确分配内存。 另一方面，借用(引用)一个 C ++字符串需要`c_str`，而 C 字符串很蠢。

幸运的是, Rust 的情况更好 - *一旦* 你接受`String`和`&str`两者其实都是必要的。 `String`的方法主要是为了改变字符串，就像`push`添加一个字符 (在引擎盖下它非常像一个`Vec<u8>`)。 但是所有`&str`的方法也可用。得益于`Deref`同一机制，一个`String`可以作为`&str`类型传递给一个函数 - 这就是为什么你很少看到，在函数中定义`&String`。

对应各种 trait，有很多方法可以把`&str`转换为`String`。 Rust 需要这些 trait 来处理泛型类型。 作为一个经验法则，任何实现`Display`，也知道`to_string`，像`42.to_string()`。

一些操作，可能不会按照直觉行事:

```
    let s1 = "hello".to_string();
    let s2 = s1.clone();
    assert!(s1 == s2);  // cool
    assert!(s1 == "hello"); // fine
    assert!(s1 == &s2); // WTF?
```

记得，`String`和`&String`是不同的类型，和没有为该组合定义`==`。这可能会让 C ++ 开发者，感到迷糊，因习惯于引用与数值几乎可以互换。 此外，`&s2`不会 *神奇* 成为一个`&str`， 一个 *deref 强制* 只在分配到 一个`&str`变量或参数时，才会发生。 (明确的`s2.as_str()`能工作。)

但是，这有更值得注意的一个 WTF:

```
let s3 = s1 + s2;  // <--- 不行
```

你不能连接两个`String`值，但可以使用`&str`连接一个`String`。 此外，您不能使用`String`连接一个`&str`。所以大多数人不会使用`+`，而是使用`format!`宏，这很方便，但效率不高。

有些字符串操作可用，但工作方式不同。 例如，编程语言通常有一个`split`方法，能将字符串分解为字符串数组的。Rust 字符串的这个方法，返回一个 *迭代器* ，你可以 *之后* 去 `collect` 成一个 Vec 。

```
let parts: Vec<_> = s.split(',').collect();
```

如果你急着获取 Vec，这有点笨拙。 但是你可以对这部分进行操作， *不用* 分配一个 Vec! 例如，split 过程中，最大的字符串的长度?

```
let max = s.split(',').map(|s| s.len()).max().unwrap();
```

(使用`unwrap`是因为空迭代器没有最大值，我们必须覆盖这种错误情况。)

该`collect`方法返回一个`Vec<&str>`，其中`&str`部分是从原始字符串中借用的 - 我们只需要为引用分配内存空间。(这意味着小且固有大小。) 在 C ++中没有像这样的方法，但直到最近才需要单独分配每个子字符串。 (C ++ 17 有`std::string_view`，其行为像一个 Rust 字符串切片。)

关于分号的说明

分号是 *不是* 可选项，但通常，与 C 相同的地方都被省略，例如，在`{}`代码块之后，他们也不需要`enum`要么`struct` (这是一个 C 特性。)但是，如果该代码块必须有一个 *值* ，那么分号将被丢弃:

```
    let msg = if ok {"ok"} else {"error"};
```

请注意，在`let`声明之后，必须有一个分号在!

如果在`x * *`字符串之后，加上分号，那么返回的值就是`()`(像`Nothing`要么`void`)。定义函数时常见错误:

```
fn sqr(x: f64) -> f64 {
    x * x; //  多了个 分号
}
```

`rustc`在这种情况下，会给你一个明确的错误。

专怼 C ++问题

Rust 值语义是不同的

在 C ++中，可以定义，类似原始的类型，并复制它们自己。 另外，可以定义移动构造函数，来指示如何将值移出临时上下文。

在 Rust 里，原始类型的行为和预期一样，但是`Copy`trait 只能在集合类型 (结构{struct}，元组{tuple}或枚举{enum}) 本身，只包含可复制类型的情况下定义。 任意类型可能有`Clone`，但你必须使用`clone`方法。 Rust 要求任何分配都是明确的，不要隐藏在复制构造函数或赋值运算符中。

所以，复制和移动总是被定义为只是 移动位比特，而不是被覆盖。

如果`s1`不是`Copy`值类型，像`s2 = s1;`导致移动发生，而这 *消耗* `s1`! 所以，当你真的想要一个副本，使用`clone`。

借用通常比复制要好，但是你必须遵循借用规则。 幸运的是，借用 *是* 一个可覆盖的行为。 例如，`String`可以借用成`&str`，并共享`&str`的所有的不可变方法。 *字符串切片* 比 类似的 C ++"借用{borrowing}"操作，更加强大，C ++ 要运用`c_str`提取一个`const char *`。 `&str`由一个指针，指向一些具有所有权的字节 (或一个字符串字面量) 和 一个 *尺寸「size」* 组成。 这造就了一些非常有效的内存模式。 你可以有一个`Vec<&str>`，其中所有的字符串都是从一些底层字符串中借用的 - 你只需要分配该 Vec 内存空间:

例如，按空格拆分:

```
fn split_whitespace(s: &str) -> Vec<&str> {
    s.split_whitespace().collect()
}
```

同样，一个 C ++ 的`s.substr(0,2)`调用将始终复制字符串，但切片只会借用:`&s[0..2]`.

`Vec<T>`和`&[T]`之间是一个雷同关系，就像字符串与字符串切片。

共享引用

Rust 有 *智能指针*，这像 C ++ - 举例，`std::unique_ptr` 相当于是 `Box`。但没必要 `delete(删除)`，因为任何内存或其他资源，在盒子超出作用域时都会被回收 (Rust 非常赞同 RAII)。

```
let mut answer = Box::new("hello".to_string());
*answer = "world".to_string();
answer.push('!');
println!("{} {}", answer, answer.len());
```

起初，人们发现 `to_string` 老烦啦，但，确实 *明朗* 许多。

注意显式的 `*` 取值符号，但智能指针有所不同，它的方法不需要任何特别符号 (我们不用这样写 `(*answer).push('!')`)

显然，只有在原始内容的所有权(者)，被明确定义的情况下，借用才有效。在许多设计中是不可能的。

C ++中，`std::shared_ptr` 用处是; 仅复制，那修改公用数据的引用计数。然而，这并不是没有成本的:

- 即使数据是只读的，不断修改引用计数也会导致缓存失效。
- `std::shared_ptr` 被设计成线程安全的，也带来了锁定开销。

在 Rust 中的 `std::rc::Rc`，也像共享智能指针一样，它使用了引用计数。 但是，它仅适用于不可变引用! 如果你想要一个线程安全的变体，请使用 `std::sync::Arc` ('原子(Atomic) Rc')。 所以， Rust 在提供两种变体方面略显笨拙，但你可以避免非线程操作的锁定开销。

正如上所说，应用的都必须是不可变的引用，因为这是 Rust 内存模型的基础。 但是，有一张权限卡: `std::cell::RefCell`。 如果您有个共享引用定义为 `Rc<RefCell<T>>`，那么你能用它的 `borrow_mut` 方法，获得可变借用。 这使 Rust 借用规则变得 *动态* 起来 - 除此之外像，已有了借用，任何尝试使用 `borrow_mut` 的操作，都会引起恐慌。

权限卡仍然是 *安全*。 恐慌会在任何内存被不当地触动 *之前* 发生! 异常情况下，他们展开调用栈。所以对这样一个结构化的回溯过程来说，恐慌是个不好的词 - 因其实这是一种有序的回溯，而不是恐慌，无序的撤退。

完整的 `Rc<RefCell<T>>` 类型，看着不舒服，但应用程序代码并不会不爽。 Rust (再次) 更喜欢明确表示。

如果你想线程安全地访问共享状态，那么 `Arc<T>` 是唯一的 *安全* 道路。如果你需要可变权限，那么 `Arc<Mutex<T>>` 会帮到你，相当于 `Rc<RefCell<T>>`。而 `Mutex` 与通常定义的方式有点不同: 它是一个值的容器。 在 值 上你得到一个 *锁{lock}*，然后可以修改它。

```
let answer = Arc::new(Mutex::new(10));

// 在其他线程
..
{
  let mut answer_ref = answer.lock().unwrap();
  *answer_ref = 42;
}
```

为什么有个`unwrap`？因为如果前一个线程恐慌了，那么这个`锁`就失败。(这是在文档中，`unwrap`被认为是合理的一个地方，因为显然，事情会走向严重错误，而这时，总要在线程中抓到(unwrap)恐慌。)

重要的是 (像往常一样使用 Mutex 锁) 这个互斥锁尽可能少地持有。 所以，它们出没在一个有限，短的作用域内是很常见的 - 然后，当可变引用超出作用域时锁定结束。

与 C ++中，显然更简单的情况相比 (`use shared*ptr dude`)，这 Rust 很不好看，但是，现在任何共享状态的 *修改* 都变得明显， 还有，“互斥锁{Mutex}”锁定模式会强制线程变得安全。

像所有内容一样，会有使用共享引用的警告。

迭代器

C ++中的迭代器，定义的很不正式;他们涉及到智能指针，通用`c.begin()`，从头开始并以`c.end()`结束。迭代器上的操作，稍后实现为独立的模板函数，如`std::find_if`。

Rust 的迭代器由`Iterator` trait 定义; `next`返回一个`Option`，和当`Option`是`None`时，迭代结束了。

最常见的操作正如下所示的方法，这是`find_if`的等价方法。 它返回一个`Option` (若是没有发现，就是一个`None`) 和这里`if let`语句可以方便地提取非`None`状态:

```
let arr = [10, 2, 30, 5];
if let Some(res) = arr.find(|x| x == 2) {
    // res 是 2
}
```

不安全和链接列表

Rust stdlib 的某些部分实现使用了`unsafe`，这不是什么秘密。 这并不妨碍 借用检查员 的传统做法。 要记住的是，”unsafe”具有特别含义 - 表明 Rust 在编译时，无法完全验证的操作。 从 Rust 的角度来看，C ++始终处于不安全的模式! 所以如果一个大的应用程序需要几十行不安全的代码，那很好，因为这些行代码可以仔细检查(明确是 unsafe)。人类可不善于检查 100Kloc +的代码。

我提到这一点的原因，是因为似乎有一种行为模式: 一个有经验的 C ++人 试图实现 链表或树结构，不过沮丧收场。 那么，一个双链表 *是* 可能符合安全 Rust 的，秘密在于`Rc`引用前进，和`Weak`引用回退。 但是标准库仍能获得了更多的性能，若是不用...指针（But the standard library gets more performance out of using... pointers.）。