# **Rust“与众不同”特点的汇总**

[](https://www.zhihu.com/people/Tmacy)

**[Tmacy](https://www.zhihu.com/people/Tmacy" \t "https://zhuanlan.zhihu.com/p/_blank)**

知乎深潜者

​关注他

14 人赞同了该文章

参考《rust程序设计语言》，一点学习笔记分享

## **所有权**

Rust提供所有权系统，可以让Rust程序无需垃圾回收也可以保障内存安全。所有权系统时理解Rust程序最重要的部分。所有权让写代码变得更谨慎，更复杂，但会让程序运行在不降低运行速度的前提下，更加安全，同时能降低Bug概率。

所有权规则：

1. Rust中的值都有一个被称为所有者变量。
2. 值只能有一个所有者变量。
3. 所有者变量离开作用域，此值将被丢弃。

这里所有权与其他语言的作用域概念时类似的。对于程序里的一个值，它的存储方式有三种，一个是直接编译到程序中，一个是在运行时分配到栈上，最后一个是在运行时分配到堆上。直接编译到程序的值是不可改变的，例如常量。编译后就可以确定存储空间的可变量，会被分配到内存的栈空间中。在运行时才能确定存储空间的值，会被分配到堆空间中。

栈空间的值，如果退出作用域将会被释放，而堆空间的值，对于不同的语言处理方式不一样。C/C++需要在代码中指定释放空间，Java，Go一类有垃圾回收的语言则依靠垃圾回收器来释放不需要的值。

Rust采取的方式是：一旦离开作用域，就会释放空间。这种类似半自动垃圾回收的策略，可以让Rust满足C/C++的运行效率，同时能有效减少内存泄漏的问题。

变量离开作用域后，Rust会调用一个特殊的函数：drop，帮助释放内存。你可以为特定类型定义drop的行为。

### **所有权的转移**

1. 数据的移动

// 将变量x赋值给y

let x = 1;

let y = x;

​

将值1绑定到变量x上，接着生成一个值x的拷贝绑定到变量y上。此时x与y所对应的值都是1。数值的赋值过程其实就是值拷贝，x与y所对应的值其实是两个拷贝。

let s1 = String::from("Hello");

let s2 = s1;

字符串s1的值是对应分配在堆空间的，s1是对于此空间的一个引用(绑定)。s1本身包括一个指向堆空间的指针，一个代表字符串长度的值以及一个代表容量的值。将s1赋值给s2，堆空间的值并没有被复制，s2引用的还是同样的内存空间。

但此时s1将失去对此内存空间的所有权！因此在let s2 = s1之后，s1将无法使用，其值的所有权被转移到s2了。同时，当s1离开作用域时，无需清理任何空间，因为s1已经不再有效。

当s2离开作用域后，其所绑定的内存空间将会被释放。

1. 数据的克隆

如果想同时使用s1与s2，上述情况需要使用深度拷贝:

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

let s2 = s1.clone();

clone会将堆上的数据复制一份绑定给s2，此时s1与s2引用了同样的值("hello")，但并不是同一个值。

Rust中提供一个Copy trait，如果一个类型有Copy的trait，则可以直接通过赋值来传递值。类似整型变量的传递方式。本质就是复制变量所绑定的值。但是如果一个类型的任何部分实现了Drop的trait，则无法使用Copy trait。能使用Copy的类型例如：整数，浮点数，布尔类型，字符类型，包括以上类型的元组。

1. 所有权与函数

函数的传参过程可能会导致所有权的移动，与变量赋值类似。

* 当具有Copy trait类型的变量通过函数传参，就是传值。原变量在传递到函数中后依然可以使用
* 当一个引用传递给函数，其引用的值的所有权也相应转移到函数中了，该引用变量将无法在后续代码中使用。

返回值也会发生所有权的转移。

变量的所有权总是遵循相同的模式：将值赋给另一个变量时移动它。当持有堆中数据值 的变量离开作用域时，其值将通过 drop 被清理掉,除非数据被移动为另一个变量所 有。

**如何在传参与返回过程中取消所有权的转移**

* 传递一个引用

**fn** **main**() { **let** s1 **=** String::from("hello"); **let** len **=** calculate\_length(**&**s1); println**!**("The length of '{}' is {}.", s1, len);}**fn** **calculate\_length**(s: **&**String) -> **usize** { *//s是对String的引用* s.len()}*//s离开作用域，但它不拥有引用值的所有权，所以不会有所有权转移*

使用&符号就表示一个数据的引用。它们允许使用值但不获取所有权。

这种获取引用作为函数参数叫做：借用(borrowing)，例如一个人借用了某样东西，从哪里借来还需要还回去。

**但是借来的东西无法修改**

如果想尝试修改借用变量，会无法编译。如果需要修改一个引用的值，需要传递一个可变引用：&mut s

可变引用有个很大的限制，就是在特定作用域内只允许有一个可变引用。这个限制好处就是Rust可以在编译时就避免数据竞争（data race）。数据竞争的发生会导致未定义的行为，很难在运行时被调试追踪。

数据竞争的产生有三个条件：

1. 两个或更多的指针同时访问一个数据
2. 至少有一个指针写入数据
3. 没有同步数据访问的机制

### **没有所有权的类型**

第一种没有所有权的类型就是引用。其实所有权的概念相对于引用的值，而非引用本身。当然引用本身的值应该就是所引用值的地址。

引用的特点：

1. 在任意时间点，要么只能有一个可变引用，要么只能有多个不可变引用。
2. 引用不能失效。（其实就是引用的值必须存在）

第二种没有所有权类型的值是slice，应该翻译为切片。

slice允许引用集合中一段连续的元素序列，而不用引用整个集合。

**let** s **=** String::from("hello world");**let** hello **=** **&**s[0..5]; *//包括0,但不包括5***let** world **=** **&**s[6..11];*//同上//////////////////////////////////////***let** hello **=** **&**s[0..**=**4]; *//使用=表示包括4***let** world **=** **&**s[6..**=**10];**let** slice **=** **&**s[..2]; *//默认是从0开始***let** slice **=** **&**s[3..]; *//也可以省略结束索引值，表示索引到字符串的最后*

“字符串 slice” 的类型声明写作 &str ，其他类型的例如整型slice，写作&[i32]

### **HashMap与所有权**

对于像 i32 这样的实现了 Copy trait 的类型,其值可以拷贝进哈希 map。对于像 String 这样拥有所有权的值，其值将被移动而哈希 map 会成为这些值的所有者，一旦键值被插入到HashMap中，就被HashMap所有了。

## **生命周期**

Rust中的每个引用都有其生命周期(lifetime)，也就是引用保持有效的作用域。大部分的生命周期是隐含并可以推断的。但也会出现一些引用的生命周期存在一些不同方式的关联，因此Rust也需要我们使用生命周期的标注来表明他们的关系。

### **生命周期的注解语法**

生命周期注解并不改变任何引用的声明周期的长短。生命周期注解描述了多个引用生命周期相互关系，并不影响其生命周期。语法是使用(')开头，名称通常是小写。'a 时大多数人默认使用的名称。生命周期参数注解位于引用符号&之后，并有一个空格来将引用类型与生命周期注解分隔开。

**&i32** *// 引用***&**'a **i32** *//带有显式生命周期的引用***&**'a **mut** **i32** *//带有显式生命周期的可变引用*

单个生命周期注解本身没有多少意义，因为生命周期是表示多个引用的生命周期参数的相互关系。具有相同生命周期注解的引用意味着他们存在一样久。

### **函数签名中的生命周期注解**

一个例子：

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

函数签名表示，对于某些生命周期'a，函数会获得两个参数，他们都是生命周期‘a存在一样长的字符串slice。函数也会返回一个与生命周期'a存在一样长的slice。这就是告诉Rust需要保证这个约定。**注意在函数签名中指定生命周期参数时，我们没有改变任何传入后返回的值的生命周期，而是指任何不遵循这个约定的传入值将被拒绝**。

这些生命周期的注解，只会在函数签名中，不存在函数体中的任何代码中。当函数体引用外部代码的引用时，Rust分析函数的参数或返回值的生命周期是几乎不可能的，这些生命周期在每次调用时都可能不同，因此我们需要手动标记声明周期。

当具体引用被传递给函数时，'a所代表的具体生命周期是x的作用域与y作用域相重叠的部分。返回的引用值，能保证在最短生命周期结束前有效。

### **结构体中的生命周期注解**

定义一个包括引用的结构体：

**struct** **Important<**'a**>** { part :**&**'a **str**,}

part部分存放了一个字符串slice，这是一个引用，必须在结构体名称后声明生命周期参数，以便在结构体中使用生命周期。

### **生命周期注解的省略**

三条规则可以省略：

1. 每一个引用都有自己的生命周期参数。例如一个引用参数有一个生命周期：fn foo<'a>(x: &'a i32),两个引用参数有两个生命周期：fn foo<'a,'b>(x:&'a i32,y: &'b i32)。
2. 如果只有一个输入生命周期，它被用于所有的输出生命周期参数：fn foo <'a>(x: &'a i32) -> &'a i32
3. 如果方法有多个输入生命周期参数，如果其中有&self 或者&mut self，并且self的生命周期被赋予所以输出生命周期参数。

### **方法中的生命周期**

当为带有生命周期的结构体实现方法时，结构体字段的生命周期必须在impl关键字之后声明，并在结构体名称之后被使用，这些生命周期时结构体类型的一部分。

**impl<**'a**>** Important**<**'a**>** { **fn** **level**(**&**self) -> **i32** { 3 }}

这里唯一的参数是self，返回值只有一个i32类型，impl之后何类型名之后的生命周期参数是必要的，但是因为省略规则，self可以不必标注。

**impl<**'a**>** Important**<**'a**>** { **fn** **announce\_and\_return\_part**(**&**self, announcement: **&str**) -> **&str** { println**!**("Attention please: {}", announcement); self.part }}

这里有两个输入生命周期，因为self与announcement有各自的生命周期，因此可以省略。而返回值的生命周期被赋予了self的生命周期，因此所有生命周期都可以被计算出来，可以省略。

### **静态生命周期**

'static是静态生命周期，其存活于整个程序期间。所有的字符串字面量都拥有'static生命周期。因为它们被储存在程序二进制文件中。

### **生命周期注解本质**

生命周期注解的本质也是泛型。其代表了任意一个作用域，这个作用域是需要推断与计算的。

### **几个生命周期高级用法**

1. 明确的指明某一个生命周期不小于另一个生命周期:'a : 'b
2. 为泛型生命周期增加边界:struct Ref<'a,T:'a> (&'a T)
3. 匿名生命周期'\_：在一个需要省略注解的函数中，用在参数与返回值中具有生命周期的struct类型标识：fn foo(string: &str) -> StrWrap<'\_>。

## **枚举**

枚举是一个很多语言都有的功能，不过不同语言中其功能各不相同。Rust 的枚举与 F#、OCaml 和 Haskell 这样的函数式编程语言中的 代数数据类型(algebraic data types)最为相似。这里汇总一下相关特点。

枚举主要是数据类型的列举，每个类型都被看作枚举的成员。参考《Rust程序设计语言》的例子：

**enum** **IpAddrKind** { V4, V6,}

创建两个不同成员的实例：

**let** four **=** IpAddrKind::V4;**let** six **=** IpAddrKind::V6;

可以定义一个函数来获取任意一个IpAddrKind：

fn route(ip\_type: IpAddrKind){}

这样就可以接收任一一个成员了：

route(IpAddrKind::V4);route(IpAddrKind::V6);

可以直接将数据附加在每个成员上：

**enum** **IpAddr** { V4(String), V6(String),}**let** home **=** IpAddr::V4(String::from("127.0.0.1"));**let** loopback **=** IpAddr::V6(String::from("::1"));

用枚举替代结构体还有一个优势：每个成员可以处理不同类型何数量的数据：

**enum** **IpAddr** { V4(**u8**, **u8**, **u8**, **u8**), V6(String),}**let** home **=** IpAddr::V4(127, 0, 0, 1);**let** loopback **=** IpAddr::V6(String::from("::1"));

一个枚举可以嵌入多种多样的类型：

**enum** **Message** { Quit,*//没有关联任何数据* Move { x: **i32**, y: **i32** },*// 一个结构体* Write(String), *//一个字符串* ChangeColor(**i32**, **i32**, **i32**),*//三个i32*}

总体来说还是一个类型，可以直接实现这个Message类型的方法来处理这些不同的类型。

**impl** Message { **fn** **call**(**&**self) { *// 在这里定义方法体* }}**let** m **=** Message::Write(String::from("hello"));m.call();*//可以用一个方法处理各种数据*

### **Option**

Rust中没有空值，利用Option枚举来表示空值。Option的定义如下：

**enum** Option**<**T**>** { Some(T), None,}

Option非常常用，被包括在prelude中，可以直接使用其成员变量Some(T)和None。

为什么Option比空值要好？简单来说Option<T>包括了类型信息，编译器不允许默认就保证Option<T>是有效的。

**let** x: **i8** **=** 5;**let** y: Option**<i8>** **=** Some(5);​**let** sum **=** x **+** y;

编译器会报错，表示Option<i8>与i8不是一种类型。如果要相加，必须处理Option中的None值，这就在一定程度上保证避免最常见的错误：假设某一个值不为空，但实际为空。