

# 06 引用、借用及生命周期

经过上一课对于所有权的学习之后，心情是不是有点儿 Down? 这个 Rust 中的所有权机制竟然如此的严格，写代码的时候一不小心就发生所有权转移了，和 JavaScript 中可以自由传递变量完全没法比嘛！先别急，Rust 的发明者不可能没有意识到这个问题，也不可能不提供对应的解决方案。

## 引用

在 Rust 中，可以通过 `&` 操作符获取一个变量的引用（Reference），本质上是指向某个值的引用。简而言之，引用可以让你在不影响所有权的前提下访问值。

Rust 中的引用有两种：

1. 共享引用（Shared Reference），也可以叫做只读引用，可以使用这些引用读取对应的值。在同一时刻，一个值可以有多个共享引用。指向某个类型 `T` 的共享引用 `&T` 读作：`ref T`
2. 独享引用（Mutable Reference），也叫做可写引用，可以使用这个引用来读取和修改对应的值。在同一时刻，一个值最多只能有一个可写引用。指向某个类型 `T` 的可写引用 `&mut T` 读作：`ref mut T`

Rust 中的引用和 C++ 中的引用非常相似，但是在 JavaScript 中没有类似的概念。

在 Rust 中，创建引用的过程叫做“借用”（Borrowing）。

## 标量类型的引用

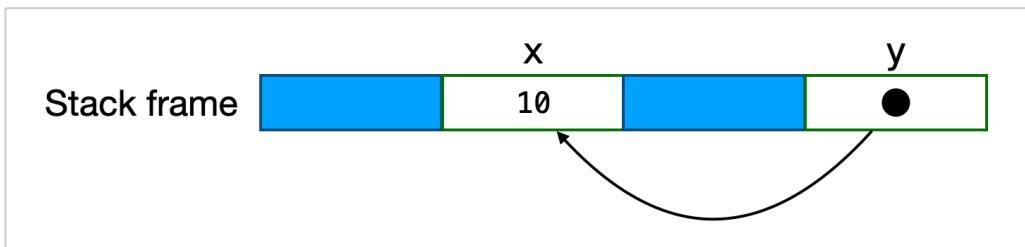
### 只读引用

创建指向标量值的引用比较简单。下面这个例子创建一个只读引用：

```

1 fn main() {
2     let x = 10;
3     let y = &x;
4     assert_eq!(*y, 10);
5 }
```

行 3 的代码使用 `&` 操作符创建了一个指向栈上的值 `10` 的只读引用，栈上空间图下图所示：



此时，`y` 的类型为 `&i32`，读作：“*ref i32*”。行 4 的代码中，我们使用了 `*` 操作符来获取 `y` 的值，这个过程叫做解引用。

## 可变引用

下面这段代码创建了一个指向值 `10` 的可变引用：

```

1 fn main() {
2     let mut x = 10;
3     let y = &mut x;
4     *y = 20;
5     assert_eq!(*y, 20);
6 }
```

细心的同学可能已经发现规律了：要想创建可变引用，那么这个值本身必须得是可变的，所以我们在行 2 代码使用 `let mut` 语句来定义变量。

## 引用的可变性

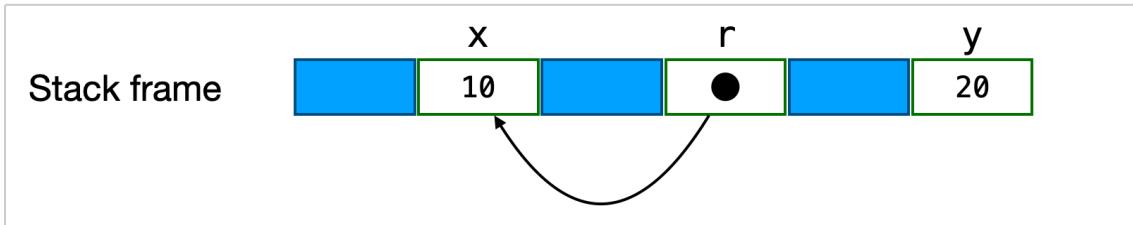
“可变引用”、“引用的可变性”，听起来非常拗口。可变引用是指引用指向的值是否可变，所以下面我们称它为“可写引用”，引用的可变性是指引用的指向是否可变。比如，上面的示例代码中，引用 `y` 在一开始指向了 `x` 的值 `10`，后续 `y` 就不可以指向其他值（地址）了。

如何能让一个引用具备指向其他值的能力？这就需要我们定义引用的时候，让它是可变的。

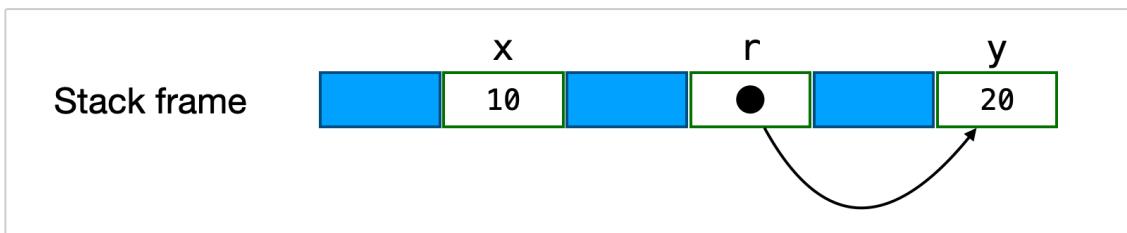
```

1 fn main() {
2     let x = 10;
3     let y = 20;
4     let mut r = &x;
5
6     let condition = true;
7     if condition {
8         r = &y;
9     }
10 }
```

当行 4 运行之后，栈上布局示意如下：



如果行 8 的代码被执行了（示例中是肯定执行的），栈上布局示意如下：



从图上可以直观的看出来，引用 `r` 所指向的地址是可变的，这就是引用的可变性。

## 堆上值的引用

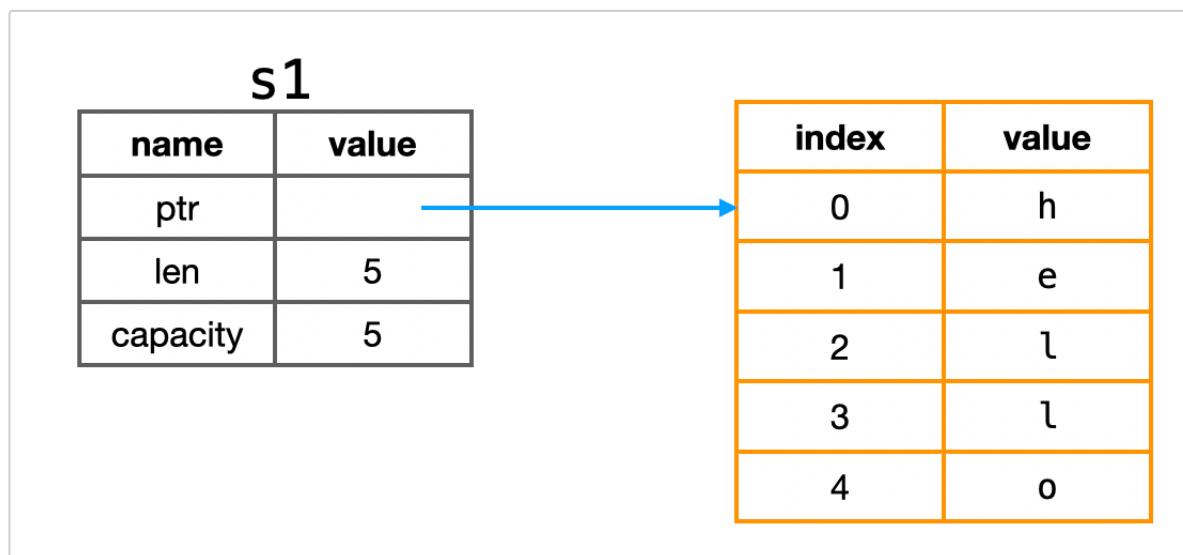
指向分配在栈上的标量类型值的引用相对容易理解。接下来我们看看指向堆上值的引用是什么样子的。

```

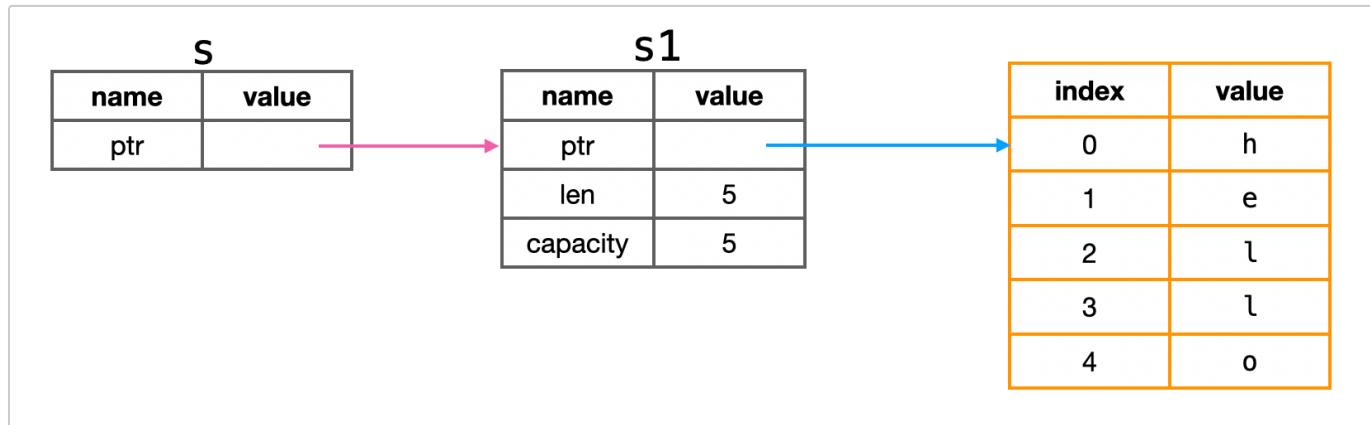
1 fn main() {
2     let s1 = String::from("Hello");
3     let len = calculate_length(&s1);
4     println!("The length of '{}' is {}", s1, len);
5 }
6
7 fn calculate_length(s: &String) -> usize {
8     s.len()
9 }
```

这段代码中，我们在堆上创建了一个 `String` 类型的值 `Hello`，并把它绑定到变量 `s1`。函数 `calculate_length` 的参数为 `&String` 类型的（ref `String`）。当我们从 `main` 函数调用 `calculate_length` 函数的时候，使用 `&` 操作符创建了一个匿名的、指向 `s1` 的只读引用。

行2 执行之后，栈（图左侧深灰色部分）和堆（图右侧橙色部分）的存储示意如下：



行 3 执行的时候，栈和堆的存储示意如下：



可以看出，进入 `calculate_length()` 函数的时候，一个名为 `s` 的引用指向了 `s1`（值 `Hello` 的所有者）。然后行 8 调用 `s.len()` 函数得到字符串的长度。前面几行都没有太多特殊的，和标量类型的引用一样。

行 8 有点儿奇怪了，为什么我们没有对 `&String` 类型的变量 `s` 使用解引用操作符 `*` 呢？这是因为我们调用了 `s` 的方法。当在一个引用上使用 `.` 来调用方法、读取属性的时候，Rust 会自动帮我们解引用，无需显式的使用 `*` 操作符。甚至，多层引用都可以自动解引用。例如，下面这行代码也是可以运行的：

```

1 fn main() {
2     let s = String::from("Hello"); // String 类型
3     let sr = &s;      // &String 类型
4     let srr = &sr;    // &&String 类型
5     let srrr = &srr; // &&&String 类型
6     println!("The length of '{}' is {}", s, srrr.len());
7 }
```

## 可写引用的排他性

前面讲到：“在同一时刻，只能有一个可写引用”。下面我们来通过一段代码来理解一下这个规则：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let mut x = 10;
4     let y = &mut x;
5     let r = &x; // 因为已经有一个可写引用了，所以这里无法成功创建只读引用
6
7     *y = 20;
8     assert_eq!(*y, 20);
9 }
```

因为可写应用具有强烈的排他性，所以也被称作“排他访问”（Exclusive Access），对应的，只读引用也被称作“共享访问”（Shared Access）。

通过上面的代码，我们可以总结出一个结论：Rust 中的引用不允许为空（不像 C/C++ 中的指针可以初始化为 `NULL`），它必须指向某个地址。这种行为可以大大减少空指针问题。

## 生命周期

来看一下下面这段代码：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let r;
4     {
5         let x = 5;
6         r = &x;
7     }
8     println!("r = {}", r);
9 }
```

编译上面的代码会报错：

```

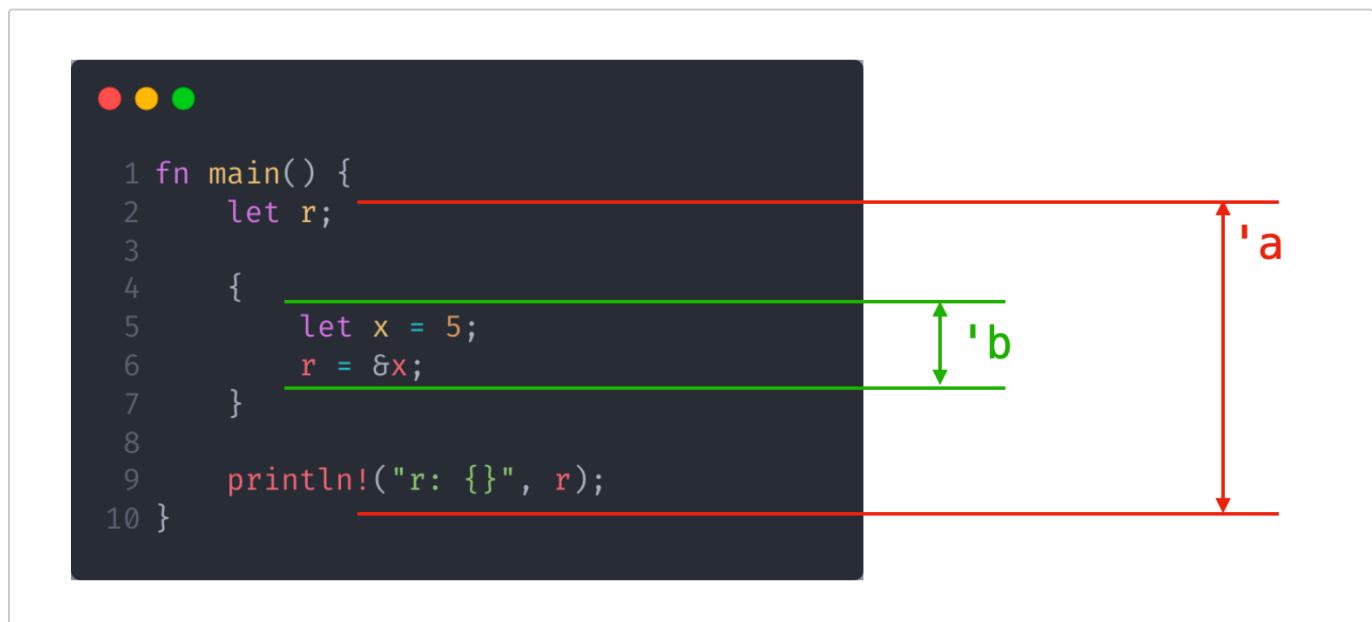
1 Compiling temp v0.1.0 (/Users/yuanyq/work/zhufeng/rust-
course/temp)
2 error[E0597]: `x` does not live long enough
3 --> temp/src/main.rs:5:13
```

```

4   |
5 4 |         let x = 5;
6   |             - binding `x` declared here
7 5 |         r = &x;
8   |             ^` borrowed value does not live long enough
9 6 |     }
10 |     - `x` dropped here while still borrowed
11 7 |     println!("r = {}", r);
12 |             - borrow later used here
13
14 For more information about this error, try `rustc --explain E0597`.
15 error: could not compile `temp` (bin "temp") due to 1 previous error

```

回顾一下变量作用域（Variable Scope）一节的内容，很显然，变量 `x` 的作用域到行 7 就结束了，而变量 `r` 的作用域会持续到行 8，而且，`r` 是 `x` 的引用，换句话说，`r` 指向了 `x`，但是 `x` 的存活时间小于 `r`，这样就有可能导致悬吊指针（Dangling Pointer），Rust 不允许发生这种情况。`r` 和 `x` 的生命周期可以用下图表示：



上图可以更加直观的看出来，`x` 的生命周期 '`b`' 比 `r` 的生命周期 '`a`' 小。

通过上面的代码和内容，不难总结出来：生命周期（lifetiem）是一种用于描述引用（references）有效性的持续时间的机制。

## 函数中的生命周期

函数中的生命周期涉及到参数的生命周期和返回值的生命周期。假定有一个函数，其目的是找出两个字符串中较长的那个并返回：

```

1 // 此代码无法通过编译
2 fn longest(x: &str, y: &str) -> &str {
3     if x.len() > y.len() {
4         x
5     } else {
6         y
7     }
8 }
```

函数 `longest` 的 2 个参数都是 `&str` 类型，并且返回值也是 `&str` 类型。由于无法判断返回的是 `x` 还是 `y`，所以 Rust 不允许这段代码通过编译。

给这段代码加上生命周期参数：

```

1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2     if x.len() > y.len() {
3         x
4     } else {
5         y
6     }
7 }
```

这里采用 `'a` 来表示生命周期，其中入参和返回值的生命周期是一致的。

有时候，在定义函数的时候，可以省略生命周期的声明，例如，下面这段代码，虽然入参和返回值都是引用类型，但是无需显式的声明生命周期：

```

1 fn first_word(s: &str) -> &str {
2     &s[0..1]
3 }
```

函数参数的生命周期称之为“输入生命周期”，函数返回值的生命周期称之为“输出生命周期”。Rust 编译器会根据以下 3 个规则依次检查，如果 3 个规则依次应用了，但是依然无法确定生命周期，那么编译器会报错。

- 规则 1：编译器给每个参数设定一个生命周期。例如：有一个参数则为 `'a`，两个参数则分别为 `'a` 和 `'b`
- 规则 2：如果只有一个输入生命周期，那么这个生命周期也会应用到返回值
- 规则 3：如果参数中有 `&self` 或者 `&mut self`，那么 `self` 的生命周期会应用到返回值上

## 静态生命周期

使用 `'static` 表明静态生命周期。静态生命周期的变量可以在整个程序运行时使用。

```

1 const HELLO: &'static str = "Hello";
2
3 fn main() {
4     let word: &'static str = "World";
5     println!("{} {}", HELLO, word);
6 }
```

## 通过例子深入理解借用规则

Rust 的编译器中有一种机制叫做：借用检查器（Borrow Checker），它是用来确保我们的代码遵守 Rust 的借用和声明周期规则的。前面说到，创建引用的过程叫做借用，既然是“借”的，就说明这个值不属于它，所以就不能一直持有这个值，早晚要还回去。虽然借用的规则很简单，但是实际使用中，总是会写出违背了借用规则的代码。所以，本章我们通过一列的示例来加深借用规则的理解。

先来复习一下：

1. 值的所有者可以创建值、读取值、写入值以及丢弃值
2. 可写引用可以读取值，也可以写入值
3. 只读引用只能读取值

4. 在同一时间，可以有多个只读引用
5. 在同一时间，最多可以有一个可写引用
6. 在同一时间，可写引用和只读引用不能同时存在

## 示例 1

先来看下面一段代码：

```

1 fn main() {
2     let mut s = String::from("Hello World");
3     let rs = &s;
4     println!("The string is: {}", rs);
5
6     let mrs = &mut s;
7     mrs.push_str(" I'm rust");
8 }
```

从代码上看，我们有一个指向 `s` 的只读引用 `rs`，还有一个可写引用 `mrs`。但是这段代码是可以通过编译的。为什么呢？Rust 的借用检查器非常聪明，它可以分析出来只读引用 `rs` 虽然拥有整个函数的生命周期（在整个函数中可用），但是到行 5 之后，实际上就再也没有再使用了，也就是说，`rs` 的作用范围到行 5 就结束了。而 `mrs` 是从行 6 开始的，所以，它可以判断出来这段代码中，只读引用和可写引用的实际作用范围并没有重叠，所以就允许通过编译。

那么，我们修改一下代码：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let mut s = String::from("Hello World");
4     let rs = &s;
5
6     let mrs = &mut s;
7     mrs.push_str(" I'm rust");
8     println!("The string is: {}", rs);
9 }
```

我们来看一下编译器给我们的错误提示：

```

1 error[E0502]: cannot borrow `s` as mutable because it is also
2 borrowed as immutable
3   |
4 4 |     let rs = &s;
5 |             -- immutable borrow occurs here
6 5 |
7 6 |     let mrs = &mut s;
8 |             ^^^^^^ mutable borrow occurs here
9 7 |     mrs.push_str(" I'm rust");
10 8 |     println!("The string is: {}", rs);
11 |             -- immutable borrow later
12 |             used here
13 For more information about this error, try `rustc --explain
E0502`.

```

仔细阅读编译器给出的错误提示，在行 4 创建了一个只读引用，在行 6 创建了可写引用，但是在行 8 还在使用只读引用，他们的作用范围出现了重叠，所以上面的代码编译不通过。

## 示例 2

这个例子和上面的例子很相似，只不过我们不再创建可写引用，而是直接使用所有者 `s` 操作字符串：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let mut s = String::from("Hello World");
4     let rs = &s;
5
6     s.push_str(" I'm rust");
7     println!("The string is: {}", rs);
8 }

```

依然是无法通过借用检查器。但是上面并没有创建可写引用啊？为什么不通过呢？我们来看一下编译器给出的错误信息：

```

1 error[E0502]: cannot borrow `s` as mutable because it is also
2 borrowed as immutable
3 |
4 4 |     let rs = &s;
5 |             -- immutable borrow occurs here
6 5 |
7 6 |     s.push_str(" I'm rust");
8 |             ^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
9 7 |     println!("The string is: {}", rs);
10 |                         -- immutable borrow later
   | used here

```

编译器说，行 6 有一个可写引用！为什么呢？我们来看一下 `push_str` 方法的定义：

```

1 pub fn push_str(&mut self, string: &str)

```

方法定义的第一个参数是 `&mut self`（我们还没有讲解到结构体，这里的 `self` 大致等同于 JavaScript 中的 `this`）。所以，行 6 的代码 `s.push_str(" I'm rust");` 实际上对应的是：`(&mut self).push_str(" I'm rust")`。所以，这段代码和示例 1 中的代码类似，都是同时存在只读引用和可写引用，所以违反了 Rust 的借用规则。

## 示例 3

来看下面一个例子，假设编写一个判断传入的两个字符串是否都为空的函数，本来期望是传入两个不同的字符串，但是我们传入的是同一个 `String` 的两个只读引用：

```

1 fn main() {
2     let s = String::from("Hello world!");
3     let ret = is_both_empty(&s, &s);
4     println!("two string are empty: {}", ret);
5 }
6
7 fn is_both_empty(s1: &str, s2: &str) -> bool {
8     s1.len() == 0 && s2.len() == 0
9 }
```

所以，这段代码是可以正常编译并执行的。因为 Rust 允许多个只读引用同时存在。接下来，我们增加一个需求，就是编写一个函数，把两个字符串都清空：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let mut s = String::from("Hello world!");
4     let ret = is_both_empty(&s, &s);
5     println!("two string are empty: {}", ret);
6
7     if !ret {
8         println!("empty both strings");
9         empty_both(&mut s, &mut s);
10    }
11 }
12
13 fn is_both_empty(s1: &str, s2: &str) -> bool {
14     s1.len() == 0 && s2.len() == 0
15 }
16
17 fn empty_both(s1: &mut String, s2: &mut String) {
18     *s1 = String::from("");
19     *s2 = String::from("");
20 }
```

上面的代码无法通过编译。Rust 的编译器错误信息如下：

```

1 error[E0499]: cannot borrow `s` as mutable more than once at a
time
2 --> ref_1/src/main.rs:9:28
3 |
4 9 |         empty_both(&mut s, &mut s);
5 |         ----- ----- ^^^^^^ second mutable borrow
6 |         |           occurs here
7 |         |           first mutable borrow occurs here
8 |         |           first borrow later used by call

```

提示非常明确：行 9 的第二个 `&mut s` 违反了 Rust 的借用规则，所以编译出错了。

## 示例 4

再来看这么一段代码：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let s1 = String::from("Hello world!");
4     let rs1 = &s1;
5     let s2 = s1;
6     println!("string is: {}", rs1);
7 }

```

编译器给出的错误信息如下：

```

1 error[E0505]: cannot move out of `s1` because it is borrowed
2   --> ref_1/src/main.rs:5:14
3   |
4 3   |     let s1 = String::from("Hello world!");
5   |           -- binding `s1` declared here
6 4   |     let rs1 = &s1;
7   |           --- borrow of `s1` occurs here
8 5   |     let s2 = s1;
9   |           ^^ move out of `s1` occurs here
10 6  |     println!("string is: {}", rs1);
11   |           --- borrow later used here

```

行 5 的代码是要把值的所有权从 `s1` 转交给 `s2`，但是由于此时 `s1` 还被 `rs1` 引用着呢，当然不可以转交了。想象一下，你有一支笔，你把它借给小红用了，在小红还回来之前，你无法把这支笔赠送给小明。

## 示例 5

下面这段代码很容易发现错误：

```

1 // 以下代码无法通过编译
2 fn gen_number() -> &u32 {
3     let x = 42u32;
4     &x
5 }

```

变量 `x` 的生命周期仅在 `gen_number()` 函数中，如果返回的是它的引用，很明显会导致悬吊指针的问题。

## 示例 6

上个示例中的代码问题很明显，那么我们再来看一个不这么明显的。代码如下：

```

1 // 以下代码无法通过编译
2 fn main() {
3     let bytes = gen_string().as_bytes();
4
5     println!("bytes are: {:?}", bytes);
6 }
7
8 fn gen_string() -> String {
9     String::from("Hello world!")
10}

```

编译器给出的错误信息为：

```

1 error[E0716]: temporary value dropped while borrowed
2 --> ref_1/src/main.rs:3:17
3 |
4 3 |     let bytes = gen_string().as_bytes();
5 |             ^^^^^^^^^^^^^^ - temporary value is
6 freed at the end of this statement
7 |
8 4 |     |
9 5 |     println!("bytes are: {:?}", bytes);
10 |             ----- borrow later used here
11 |
12 help: consider using a `let` binding to create a longer lived
13 value
14 3 ~     let binding = gen_string();
15 4 ~     let bytes = binding.as_bytes();
16 |

```

也就是说，在 `main()` 函数调用 `gen_string()` 函数之后，返回了一个 `String` 类型的值，但是这个值没有对应的所有者。没有所有者并不是什么大事儿，问题出在 `as_bytes()`。我们来看一下这个方法的定义：

```
1 pub fn as_bytes(&self) -> &[u8]
```

`as_bytes()` 方法返回的是一个切片，它是所指向值的一段连续空间的引用。在行 3 代码中，Rust 帮我们创建了一个临时的值来持有 `gen_string()` 函数返回的 `String` 值，但是这个值在行 3 运行完毕之后就释放了，所以 `as_bytes()` 返回的切片就成了悬吊指针了，所以无法通过编译。

## 示例 7

通过前面的示例，大概理解了。但是别急，看下面这个例子：

```
1 fn main() {
2     let s = &gen_string();
3     let bytes = s.as_bytes();
4     println!("bytes are: {:?}", bytes);
5 }
6
7 fn gen_string() -> String {
8     String::from("Hello world!")
9 }
```

理论上，行 2 运行完毕之后，Rust 帮我们创建的临时值也释放了啊？为什么这个代码可以通过编译呢？其实你仔细看前面的示例 1 和示例 2，从代码字面来看，都是违反了 Rust 的借用规则的，但是最终可以通过编译。实际上，本示例的代码在早期的 Rust 版本中确实是无法通过编译的，但是在后来的逐步迭代中，Rust 的借用检查器越来越聪明，它会尽可能的更加聪明的理解我们的代码。在本例中，编译器使用了一种叫做“引入的临时值的借用延长”的机制，它发现 `s` 引用了一个临时的值，它自动的把临时值的生命周期延长到 `s` 的周期了，所以可以通过编译。

前面的示例 6 不通过是因为 `gen_string()` 是在调用 `as_bytes` 的时候，创建了一个临时的只读引用 `&self`，但是执行完毕这一行，`&self` 也超范围了，它实在是没办法帮我们自动延长了，所以就失败了。

可能这种行为一定程度上方便了开发者，另外一个角度来说，我认为自动延长机制并不是好事儿。强烈建议：只要是值，那就一定用变量作为所有者来接住，确保代码具有很好的可读性以及不受编译器版本的影响。那么本例中的代码应该写成：

```

1 fn main() {
2     let s = gen_string();
3     let bytes = s.as_bytes();
4     println!("bytes are: {:?}", bytes);
5 }
6
7 fn gen_string() -> String {
8     String::from("Hello world!")
9 }
```

这里的 `s` 是 `gen_string()` 函数创建的 `String` 类型的值转交所有权之后的新所有者，非常清晰的表达。

## 智能指针

### `Rc<T>` 和 `Arc<T>`

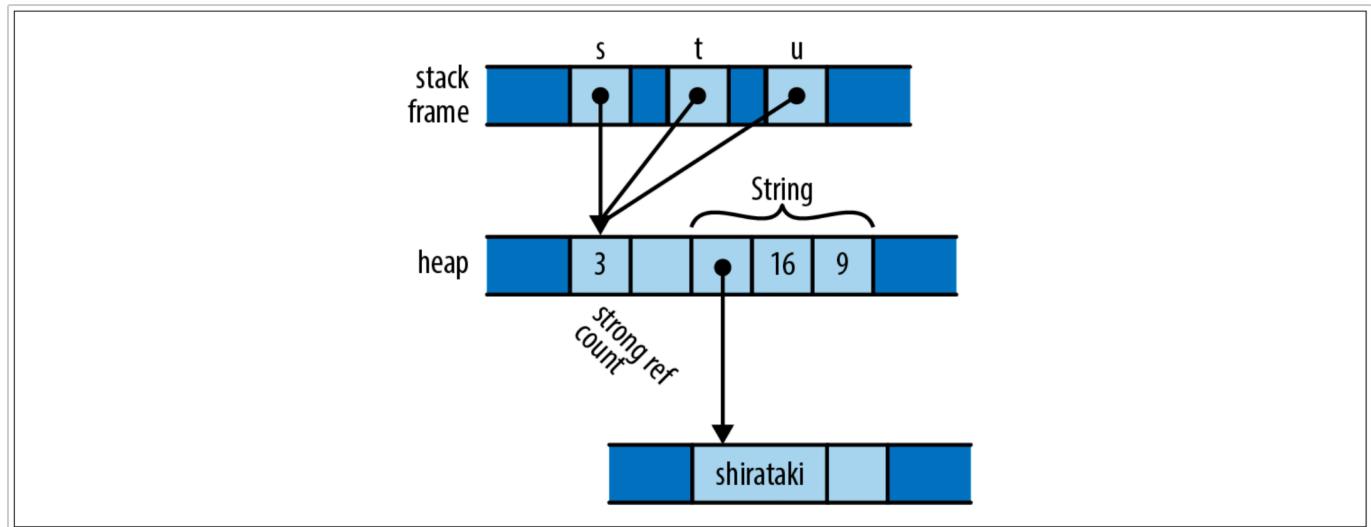
理想情况下，一个值只有一个明确的所有者，但是有些时候我们很难给一个值唯一的拥有者，可以适配所需的生命周期。所以，Rust 中还提供了引用计数指针（reference-counted pointer）的方案来共享所有权：`Rc`（Reference Count）和`Arc`（Atomic Reference Count）。`Rc` 和 `Arc` 的唯一区别就是 `Arc` 可以在多个线程之间传递，而 `Rc` 不能。

- 对于任何类型 `T`，`Rc<T>` 是一个指针，指向堆上的值 `T`。
- 克隆 `Rc<T>` 不会复制 `T` 的值，而是创建一个新的 `Rc<T>`，并增加引用计数
- 每个 `Rc<T>` 超出作用范围的时候，引用计数将会被减 1。直到引用计数为 0，堆上的值将被释放
- `Rc<T>` 指向的值是不可变的（immutable）

```

1 use std::rc::Rc;
2
3 fn main() {
4     let s: Rc<String> = Rc::new(String::from("shirataki"));
5     let t: Rc<String> = s.clone();
6     let u: Rc<String> = s.clone();
7 }
```

上面这段代码在内存中的示意图如下：



可以看到，每次克隆 `Rc<T>`，并不会真正复制堆上的值，而是增加这个值的引用计数。还记得我们前面内容中提到的，自动内存管理的方案：GC 和 ARC。虽然 Rust 有自己独特的所有权来实现自动内存管理，但是最终还是不得不提供了一些其他的手段来满足不同场景的需求。

我们可以修改一下上面的代码，看看是不是如图所示：

```

1 use std::rc::Rc;
2
3 fn main() {
4     let s: Rc<String> = Rc::new(String::from("shirataki"));
5     let t: Rc<String> = s.clone();
6     let u: Rc<String> = s.clone();
7
8     println!("ref count: {}", Rc::strong_count(&s));
9     println!("ref count: {}", Rc::strong_count(&t));
10    println!("ref count: {}", Rc::strong_count(&u));
11 }

```

然后，我们再修改一下，让 `t` 先超出作用范围，看看是否导致了引用计数器的变化：

```

1 use std::rc::Rc;
2
3 fn main() {
4     let s: Rc<String> = Rc::new(String::from("shirataki"));
5     println!("ref count: {}", Rc::strong_count(&s)); // 1
6
7     {
8         let t: Rc<String> = s.clone();
9         println!("ref count: {}", Rc::strong_count(&t)); // 2
10    }
11
12    let u: Rc<String> = s.clone();
13
14    println!("ref count: {}", Rc::strong_count(&s)); // 2
15    println!("ref count: {}", Rc::strong_count(&u)); // 2
16 }

```

Box<T>

前面的内容提到过，对于编译时已知大小的值会在栈上存储，比如整数、布尔值等标量类型的值。但是栈有一个限制就是，只能由当前运行的函数访问其上的数据。针对标量类型的值，如果我们也希望多个线程可以访问呢，就需要把标量值放到堆上。如何把标量值存放到堆上？这是我们就需要 `Box<T>` 类型了。

```

1 fn main() {
2     let mut b = Box::new(5);
3     *b = 6;
4     println!("b = {}", b);
5 }
```

实际上，`Box<T>` 类型就是一个指向堆上 `T` 类型值的一个指针，可以通过 `*` 或者 `.` 解引用。当然了，这里只是把标量值存储到堆上，只是实现了共享访问的第一步，真正的多线程访问还需要更多的工作，后续我们会讲解到。

## `Cell<T>` 和 `RefCell<T>`

根据 Rust 的内存管理规则，在同一时刻，类型 `T` 的值可以：

1. 有一个或多个只读引用 `&T`
2. 最多有一个可写引用 `&mut T`

但是这种限制虽然提升了内存使用的安全性，但是同样导致了灵活性大打折扣。甚至某些场景下，确实是需要多个可写引用的。所以 Rust 提供了 `Cell<T>` 和 `RefCell<T>`，允许我们在可控的前提下实现共享可写引用，也就是“内部可变性”（interior mutability）。但是需要注意的是，这两种类型都是单线程的，不适用于多线程场景。

`Cell<T>` 通过将类型为 `T` 的值移入 `cell`、从 `cell` 中移出来实现内部可变性。也就是说，我们无法获取，也无需获取 `&mut T`，并且只能通过使用一个新的值去替换 `cell` 内部的值，我们才可以直接访问其内部值。

- `Cell::<T>::new(value)` 创建一个 `cell`，并将 `value` 的所有权转交给这个 `cell`
- `cell.get()` 获取 `cell` 内部包裹的值的一个拷贝。仅当 `T` 实现了 `Copy` trait 有此方法

- `cell.replace()` 使用一个新值替换 cell 中包裹的旧值，并返回被替换的旧值（旧值的所有权发生转移）
- `cell.set()` 设置一个新值。原来的旧值将被丢弃（释放所对应的内存空间）
- `cell.take()` 将包裹的值的所有权从 cell 中转移出来，然后使用类型 T 的默认值填充 cell。仅当 T 实现了 `Default` trait 有此方法

在实际开发中，`Cell<T>` 通常被用来处理标量类型的、且占用内存较少的数据，比如数值类型的数据。如果需要针对堆上存储的、尺寸较大的数据，推荐使用 `RefCell<T>`。

`RefCell<T>` 通过临时的、独享的可写引用在不违反 Rust 生命周期的约束前提下实现“动态借用”。

- `ref_cell.borrow()` 获得对内部包裹值的只读引用
- `ref_cell.borrow_mut()` 获得对内部包裹值的可写引用

`Cell<T>` 有一定的运行时开销，因为它需要在运行时检查借用规则（类似于对象的锁），如果违反借用规则会导致执行异常。换句话说，`Cell<T>` 和 `RefCell<T>` 是把 Rust 借用规则的检查从编译器推迟到了运行期。

示例代码：

```

1 use std::cell::RefCell;
2
3 fn main() {
4     let s = RefCell::new(String::from("Hello world!"));
5     append_string(&s);
6     println!("s is: {}", s.borrow());
7 }
8
9 fn append_string(s: &RefCell<String>) {
10     let mut ms = s.borrow_mut();
11     ms.push_str(" I'm Rust");
12 }
```

上面的代码中，`s` 并没有声明为 `mut`，但是我们可以通过 `RefCell` 来实现内部可变的目的。既然说 `Cell<T>` 和 `RefCell<T>` 是把 Rust 借用规则的检查从编译器推迟到了运行期，那么我们来试试能不能破坏这个规则：

```

1 use std::cell::RefCell;
2
3 // 以下代码会发生运行期错误
4 fn main() {
5     let s = RefCell::new(String::from("Hello world!"));
6     append_string(&s);
7     println!("s is: {}", s.borrow());
8 }
9
10 fn append_string(s: &RefCell<String>) {
11     let rs = s.borrow();
12     let mut ms = s.borrow_mut();
13     ms.push_str(" I'm Rust");
14     println!("rs: {}", rs);
15 }
```

上述代码行 11 我们用 `borrow()` 方法创建了一个只读引用存储到 `rs` 中，到行 14 使用这个只读引用。行 12 和行 13 用 `borrow_mut()` 方法获得了可写引用存储到 `ms` 中。很明显，这个不符合 Rust 的借用规则，但是这段代码可以成功编译。然后我们运行一下：

```

1 $ cargo run
2 Compiling ref_1 v0.1.0 (/zhufeng/rust-course/ref_1)
3   Finished `dev` profile [unoptimized + debuginfo] target(s) in
4   0.13s
5       Running `target/debug/ref_1`
6 thread 'main' panicked at ref_1/src/main.rs:11:20:
7 already borrowed: BorrowMutError
8 note: run with `RUST_BACKTRACE=1` environment variable to display
9 a backtrace
```

可见，借用检查在运行期发现违反了 Rust 的借用规则，会出现 `BorrowMutError`，然后程序中止。所以，即使是使用 `Cell<T>` 或 `RefCell<T>` 也要遵守 Rust 的借用规则。

## 回顾

1. 只读引用可以有多个，可写引用最多只能有一个
2. 创建引用的过程称为“借用”
3. 生命周期
4. 智能指针