链滴

Day 02 - 所有权

作者：KuMa

原文链接：https://ld246.com/article/1680083443302

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

- 所有权入门
  - Stack Heap
    - Stack和Heap的存储数据
    - Stack和Heap的访问数据
  - 所有权的职能
- 所有权规则
  - 变量作用域
  - 移动
  - 克隆
  - 复制
  - 函数与所有权
- 引用
- 切片

所有权入门

Rust的核心特性就是所有权，所有计算机语言都必须管理他们使用计算机内存的方式：

- 类似Java的 他会在运行时中 不断地寻找不再使用的内存进行CG

- 类似C/C++的 它会显示的让程序员进行内存的释放与分配（C++只是部分版本）

Stack Heap

栈内存和堆内存都是我们常用的内存但是他们的结构麻~不同

Stack和Heap的存储数据

Stack 顾名思义LIFO的数据结构 添加数据叫压栈 弹出数据叫弹栈。所有存放在栈上的数据必须是已大小的并且拥有固定的大小的，所以我们在编译大小未知的数据的时候会考虑将他放在Heap中。

当你把数据放在Heap中你会请求一定数量的空间，OS会在Heap中找到一块足够大小的空间标记为使用，并返回指向这个空间的指针（内存地址），这种过程呢我们叫做内存空间分配。

压栈存储比我们的分配存储时间要快很多，因为OS不需要再压栈的时候去找给变量放置的空间，而直接找到顶针，然后进行压栈就可以了。

- 因为指针是已知固定大小的，可以把指针存放在 stack 上。
  - 但如果想要实际数据，你必须使用指针来定位。

- 把数据压到 stack 上要比在 heap 上分配快得多：
  - 因为操作系统不需要寻找用来存储新数据的空间，那个位置永远都在 stack 的顶端
- 在 heap 上分配空间需要做更多的工作：
  - 操作系统首先需要找到一个足够大的空间来存放数据，然后要做好记录方便下次分配

Stack和Heap的访问数据

Stack vs Heap
访问数据

- 访问 heap 中的数据要比访问 stack 中的数据慢，因为需要通过指针才能找到 heap 中的数据
  - 对于现代的处理器来说，由于缓存的缘故，如果指令在内存中跳转的次数越少，那么速度就越快

- 如果数据存放的距离比较近，那么处理器的处理速度就会更快一些（stack 上）
- 如果数据之间的距离比较远，那么处理速度就会慢一些（heap 上）

所有权的职能

跟踪代码的哪些部分再使用Heap的哪些数据

最小化Heap的数据量并清理Heap上未使用的数据避免空间不足。

所有权规则

变量作用域

```
fn main() {
    // s不可用
    let s = 2; // s可用 入栈
    // s 可用
} // 退出了s的作用域 s不可用 弹栈
```

接下来我们先了解下String类型

一般的基础数据类型都是放在栈上面的而不是堆上面的，我们的String可以放在堆上面

创建一个字符串

```
fn main() {
    let mut s = String::from("hello");
    s.push_str(",World");
    println!("{s}");
}
```

String为了支持可变性（区别于上述代码的"hello"之类的字面值）需要在Heap中保存编译时未知的本内容

用完String之后需要使用某种方式将内存返回给操作系统：

如果返回了多次和没有返回还有提前返回都会造成Bug、浪费内存和无法使用变量的问题，Rust中某变量走出自己的作用域的时候会默认调用一个drop方法将变量的内存交给操作系统。

移动

普通变量的移动据我们所知 是进行压栈加入的内存

我们这里讨论的主要是类似String的变量

```
fn main() {
    let mut s = String::from("hello");
    let mut s2 = s;
}
```

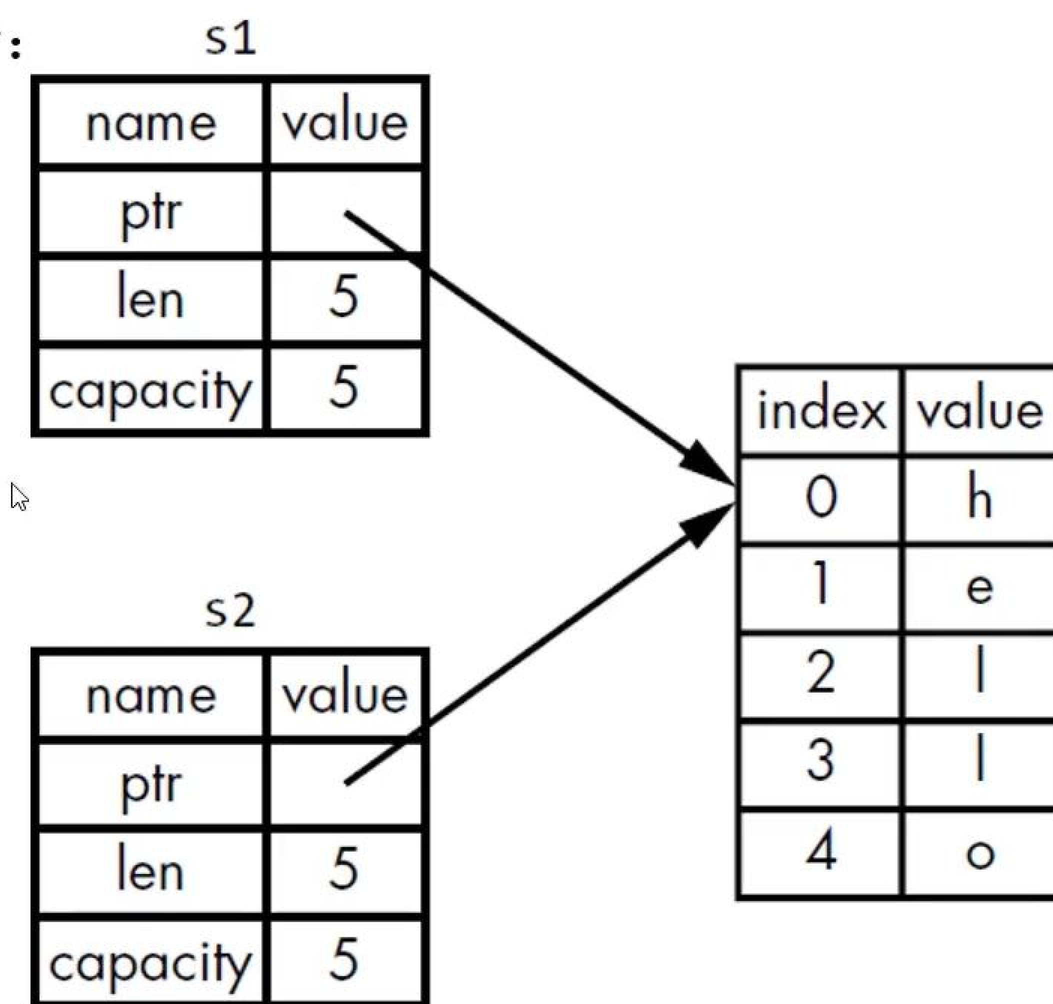
一个String类型有三部分：

- 指向字符串内容的内存指针 ptr
- 长度 len 存放字符串内容所需的字节数
- 容量 capacity 操作系统上总共获得内存的总字节数

上面这些东西都是Stack的 而字符串的内容是放在heap中的

当s的值赋值给s2的时候，s2只是再栈内存中压栈了一个新的元素里面有 ptr len 和 capacity ptr同样指向我们s的堆内存地址中。

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



当我们的s1 和 s2都离开作用域的时候，他们都会进行释放，但是因为堆内存的数据只有一个空间，s2释放了两次导致出现Bug:□ **二次释放**，Rust为了防止这样的事情发生索性 S 给S2赋值之后 S自动失，

```
fn main() {
    let mut s = String::from("hello");
    let mut s2 = s;
    print!("{}", s);
}
```

报错 因为s再把值给s2之后 会失效, 我们将之称之为 s移动到了s2, 移动和我们的浅拷贝类似 不过浅拷会使得所有进行拷贝的变量不失效.

克隆

```
fn main() {
    let mut s = String::from("hello");
    let mut s2 = s.clone();
    print!("{}", s);
}
```

克隆会把堆上面的数据复制一份交给我们的新数据

复制

Rust有这么两个特性： copy trait 和 drop trait（trait这东西他类似与interface）

当一个数据实现了copy trait 那么他再把值交给一个新的变量的时候 是不会自动失效的(也就是旧的任然可以使用) 一个类型实现了DropTrait后就不能实现CopyTrait了。

一些拥有 Copy trait 的类型

- 任何简单标量的组合类型都可以是 Copy 的
- 任何需要分配内存或某种资源的都不是 Copy 的
- 一些拥有 Copy trait 的类型：
  - 所有的整数类型，例如 u32
  - bool
  - char
  - 所有的浮点类型，例如 f64
  - Tuple（元组），如果其所有的字段都是 Copy 的
    - (i32, i32) 是
    - (i32, String) 不是

这就是所有权的三个规则：复制 克隆 移动

函数与所有权

当把一个变量传给函数的参数时，可以看作这个变量移动给了函数，所以下列的代码时不能执行的：

```
fn main() {
    let mut s = String::from("hello");
    do_thing(s);
    print!("public -> {s}"); // 不存在s了 因为再执行参数的使用发生了移动 ->报错

    let mut num = 2;
    do_thing_int(num);
    print!("{num}"); // 可以 因为number时放在栈里面的
}

fn do_thing_int(int:i32) -> i32{
    print!("{int}");
    int
}

fn do_thing(str: String) -> String {
    print!("{}", str);
    str
}
```

总结：

- 以下情况会发生转移
  - 把一个值赋值给其他变量时就会发生移动
  - 当包含heap数据的变量离开作用域的时候会被drop除非他移动到了另一个变量上

为了让我们变量不再因为传参的时候移动到方法中导致变量被销毁，我们可以再执行方法的时候把变返回回去

```
fn main() {
    let a = String::from("你好,我是刘");
    let a = do_thing(a);
    print!("{a}");
}


fn do_thing(str: String) -> String {
    print!("{}", str);
    str
}
```

像极了乱跑的小孩子跑到别人家里捣乱，然后大人无赖的将他送了回来...

▯

引用

引用允许你引用某些值而不是取得其所有权

```
fn main() {
    let a = String::from("你好,我是刘");
    do_thing(&a);
    print!("{a}");
}


fn do_thing(str: &String)  {
    print!("{}", str);
}
```

把引用传给参数就是借用

我们可以再函数里面使用变量名来获取引用的变量这就是我们的自动解引用，和变量一样默认的借用是不可变的。

```
fn main() {
    let mut a = String::from("你好,我是刘"); // 变量是可变的
    do_thing(&mut a); // 传入的参数时可变的
    print!("{a}");
}


fn do_thing(str: &mut String)  { // 传入的参数类型时可变的引用
    str.push_str("你好"); // 自动解引用
    print!("{}", str);
}
```

限制：我们的引用有一个限制就是在同一个作用域下某个变量的引用只能存在一个，因为这样会避免据的竞争。

```
fn main() {
    let mut a = String::from("你好,我是刘");
    let mut b = String::from("你好,我是章");
    let mut ay = &mut a;
    let mut by = &mut a;
    do_thing(ay); // 报错 因为 a的可变引用只能存在一个
}


fn do_thing(str: &mut String)  {
    str.push_str("你好");
    print!("{}", str);
}
```

还有就是一个作用域下不能存在同一个变量的一个可变的引用和一个不可变的引用，因为可变引用一改变了引用的值，不可变引用的效果就失去了...

可以同时读 但是不可以同时写 也不可以一个写一个读

&str 是切片的意思 你可以看看下一章的切片 温馨提示 push_str有用到a的可写引用哦

```
fn main() {
    let mut a = String::from("Hell o World");
    let b = get_char(&mut a); // 有个可写的 返回一个可读的 所以报错
    a.push_str("123");
    println!("{b}");
}

fn get_char(s: &String) -> &str {
    let chars = s.as_bytes();

    for (index, &value) in chars.iter().enumerate() {
        if value == b' ' {
            return &s[..index];
        }
    }
    return &s[..];
}
```

慢慢想为什么这个代码无法被执行

```
fn main() {
    let mut a = String::from("Hell o World");
    let b = get_char(&mut a);
    a.push_str("123");
    println!("{b}");
}

fn get_char(s: &mut String) -> &mut str {
    let chars = s.as_bytes();

    for (index, &value) in chars.iter().enumerate() {
        if value == b' ' {
            return &mut s[..index];
        }
    }
    return &mut s[..];

}
```

切片

```
fn main() {
    let a = String::from("HelloWorld");
    let b = &a[0..3];
    println!("{b}");

    let b = &a[3..];
    println!("{b}");

    let b = &a[..3];
    println!("{b}");
}
```

字符串字面值其实也是一个切片

```
main() {
  let a : &str = "HelloWorld";
```

&str表示的就是一个切片所以他是不可变的

我们之前的代码是

```
fn get_char(s: &String) -> &str
```

这样只能传入字符串到函数中，我们的字面值似乎就无法传进来（因为字符串的字面值不是字符串而切片 -字符串不是字符串~-）

但是我们可以这么理解 切片是字符串

```
fn get_char(s: &str) -> &str
```

所以我们之前的代码可以这么写

```
fn main() {
    let mut a = String::from("Hell o World");
    let ret = {
        let b = get_char(&mut a); // 我们在这里拿到了一个 可写切片的引用
        println!("{}", b);
        b // 我们这里单纯的将切片交给外部
    };
    a.push_str("123"); // 这里我们进行另外一个可写变量的使用
}

fn get_char(s: &mut str) -> &mut str {
    let first_word_len = s.find(' ').unwrap_or_else(|| s.len());
    let first_word = &mut s[..first_word_len];
    first_word
}
```