今天我要介绍一下第四章的内容，第四章的内容是所有权，所有权是rust的一个核心概念。在这本书里，所有权这一章是通过三个部分来介绍的：基本的所有权概念，包括堆栈空间，还介绍了一些字符串的性质，浅拷贝和深拷贝，然后第二部分是引用，第三部分是slice，字符串切片，就是一种特殊引用。

关于这一部分，我的想法是通过四个部分来介绍：

第一部分，变量的赋值、引用、克隆，介绍一下它们之间的区别和适用场景；

第二部分，介绍一下rust的引用特性；

//第三部分，探讨在引用过程中各种权限的转移；

然后最后一部分就是稍微提及一下，用字符串切片（？）

在开始之前，先来说一下rust中所有权最基本的三条规则：

* 每个值都有一个所有者。
* 一次只能有一个所有者。
* 当所有者超出作用域时，值将被丢弃。

从这三条规则开始，以字符串为例，介绍第一部分：赋值、引用赋值、克隆。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 赋值 | 引用 | 克隆 |
| 语法 | a = b | a = (mut) & b | a = b.clone() |
|  |  |  |  |
|  |  |  |  |

我们一行行来看。

首先第一行是三种写法，赋值，引用就是前面加个&，可变引用就再加个mut，克隆是string类中的函数。

然后从所有权的角度来介绍这三者的区别。

这三张图介绍了三种情况（4-4 4-5 4-3）

先来看第一种情况。这一部分是s1的属性，指向这部分值。在执行 s2 = s1之后，s2的ptr也指向了这个位置。但是在这种情况下，这部分值有了两个所有者，这一点rust是不允许的，所以s1的所有值被取消了，这一部分没有意义了。

再来看第二种情况。这里s不是一个完整的字符串，而是指向s1的这个位置，从而指向s1的值。这种情况下，所有权依然在s1的手里，不违背rust的所有权规则。在s的生命周期结束后，仅仅是这一部分结束了，也不影响s1的消亡。

最后看clone的情况，也就是所谓的深拷贝。在执行这个函数后，s2创建了一套和s1一模一样的值，两者所有权和各种操作互不干涉。但是考虑到这个值的部分可能很庞大，所以这种操作的成本比较高，而且在这个操作之后，两个字符串就相互独立了，如果再想建立它们之间的关系什么的，显然是不如引用合适。

这就要进入下一部分，rust的引用特性。

Rust的引用特性是rust非常独特的一个部分。

引用指，在上一个部分已经提到了，将b的引用赋值给a，a可以使用（读写）b的值，但不会拿走b的所有权，就像这个图一样。A超出作用域时，引用被丢弃了，但是原所有者仍是a，这部分值没有被丢弃，具体的权限规则在第三部分探讨。

然后看一下rust引用最重要的两条规则：

在任何给定的时刻，你可以有一个可变引用或者任意数量的不可变引用。

引用必须始终是有效的。

解释一下这两条规则：第一条就是字面意思，如果某个变量有一个可以更改它的引用，就只能有这一个；如果有多个引用，必须都是只读的。这条规则避免了以下这种问题，即

同时有两个或更多的指针访问相同的数据。

至少有一个指针被用来写入数据。

没有使用任何机制来同步对数据的访问。

需要注意的是，这个引用的生命周期和一般理解的可能不一样，引用的生命周期不是指花括号中间，而是从它被引入的地方开始，并持续到它最后一次被使用的时候。这个性质一会会用到。

第二条，引用必须是有效的，就是说rust不允许悬垂引用，不允许引用指向已经被释放的内存。

然后来看书上的两部分代码，分别说明了第一和第二条规则。

第一段代码：这里创建了两个引用，然后编译器的报错是“不能把s多次借用为可变的”

不允许多个可变引用。下面这个虽然有两个可变引用，但是这里花括号结束的时候，第一个引用的生命周期已经结束了，所以是没问题的。

第二段代码：这里前两个引用都是不可变的，第三个是可变的，就会触发报错。这里提到了，“不能将s借用为可变的，因为s已经被借用为不可变的”。

然后再来看第三段代码，说明了引用的生命周期：这里首先创建了两个不可变引用，然后创建了一个可变引用，但是不会报错。因为不可变引用最后使用是在这个位置，是在可变引用创建之前，所以编译器认为到这里前两个引用的生命周期已经结束了，可以看到它们这里的生命周期不一定要是花括号结束的部分。

上面这三段代码是关于第一条规则的，再来看一下第二条规则，更简单一点：

先看这个函数

这个函数先创造了一个”hello”字符串，然后返回值是这个字符串的引用。但是到返回值这一步时，函数的生命周期结束了，字符串的值被丢弃了。这个返回值引用就什么都不指。主函数里调用了这个函数，现在（如果代码能运行的话）reference\_to\_nothing就是一个悬垂引用。

这个函数是无法通过编译的。编译器会报错“这个返回值的类型中包含了一个借用值，但是没有值可以供它借用”。

/\*现在介绍完了引用的基本规则，进入第三部分。仍然是介绍引用，介绍不同引用过程中所有权、读写权限的转移。从这一部分可以更直观的体会到所有权的特性。

先解释一下这章图里的概念，R代表读权限，W代表写权限，O代表所有权。这里说的所有权可能和上面的概念不太一样，这里指当前这个量是否可以被move或者drop。

这里提到了，rust的所有权机制是一个量默认拥有read 和own的权限，如果添加了mut关键字，添加写权限。核心观念是引用可以暂时移除某些权限。

以这段代码为例。

创建字符串时，因为有mut关键字，strs获得了R W O三个权限。这个函数的功能是返回对str第一位的引用，这个引用是不可变的，所以创建时将strs的W O权限移除，引用，first获得R和O权限。在最后if里，执行这个操作需要W权限，所以将first drop，first的生命周期结束之后W O权限回到strs身上。

这解释了rust引用的机制，大概就是进行引用时，会对原来的量和引用的量权限进行相应操作。也可以和某些规则对应上：

如果同时包含不可变引用和可变引用，就可能出现权限的冲突。创建不可变引用移除了字符串的原来W权限，而写入又需要这个W权限，就会导致报错。

而第二条规则，引用的生命周期结束要将权限归还，如果引用没有指向任何对象，就没有地方可以归还权限。

\*/

最后一部分这个就是简单介绍一下切片，切片作为引用的一种可以体现出引用的优势。

就用一下书上的实例：

编写一个函数，该函数接收一个由空格分隔的单词字符串，并返回它在该字符串中找到的第一个单词。如果函数在字符串中没有找到空格，则整个字符串必须是一个单词，因此应返回整个字符串。

先不考虑切片，很容易就能得到下面这段代码：（listing 4-7）

定义了一个函数，参数是一个字符串引用，返回值是一个整数

然后遍历字符串中的每一位，如果这位是空格就返回当前位置

否则返回整个字符串

就上面的功能来说是没什么问题的。但是这可能引发一个问题，是listing4-8里提到的：

我们获取了“hello world”字符串第一个空格出现的位置是5，这一步是没问题的。但是下一步，执行了s.clear()，这个字符串已经不存在了。但是这个word的值仍然是5，那这个时候这个5还是字符串第一个空格的位置吗，所以说现在这个word就没有意义。为了避免这种情况，这本书提供的方案是使用字符串切片。

让我们看一下改进过的函数。这个函数的返回值不是整数，是一个字符串切片，就是一个对字符串的引用，引用从字符串开始到最近一个空格之前的位置，也可以实现同样的功能。现在，再执行s.clear()函数，可以看到编译器会报错。

编译器报错是因为如果要clear，需要一个mutable引用，但是word是一个immutable引用，不允许两种引用同时出现，这个clear无法执行，就不会出现上面的问题。这个例子通过字符串切片体现了引用的优势。

然后我介绍的大致就是这些。