Rust 编程语言入门

杨旭，微软MVP
Rust、Go、C#开发者

4 所有权

所有权是 Rust 最独特的特性，它让 Rust 无需 GC 就可以保证内存安全。

4.1 什么是所有权

- Rust 的核心特性就是所有权
- 所有程序在运行时都必须管理它们使用计算机内存的方式
  - 有些语言有垃圾收集机制，在程序运行时，它们会不断地寻找不再使用的内存
  - 在其他语言中，程序员必须显式地分配和释放内存
- Rust 采用了第三种方式：
  - 内存是通过一个所有权系统来管理的，其中包含一组编译器在编译时检查的规则。
  - 当程序运行时，所有权特性不会减慢程序的运行速度。

Stack vs Heap
栈内存 vs 堆内存

- 在像 Rust 这样的系统级编程语言里，一个值是在 stack 上还是在 heap 上对语言的行为和你为什么要做某些决定是有更大的影响的
- 在你的代码运行的时候，Stack 和 Heap 都是你可用的内存，但他们的结构很不相同。

Stack vs Heap
存储数据

- Stack 按值的接收顺序来存储，按相反的顺序将它们移除（后进先出，LIFO）
  - 添加数据叫做压入栈
  - 移除数据叫做弹出栈
- 所有存储在 Stack 上的数据必须拥有已知的固定的大小
  - 编译时大小未知的数据或运行时大小可能发生变化的数据必须存放在 heap 上
- Heap 内存组织性差一些：
  - 当你把数据放入 heap 时，你会请求一定数量的空间
  - 操作系统在 heap 里找到一块足够大的空间，把它标记为在用，并返回一个指针，也就是这个空间的地址
  - 这个过程叫做在 heap 上进行分配，有时仅仅称为“分配”

Stack vs Heap
存储数据

- 把值压到 stack 上不叫分配

- 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 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）
  - 在 heap 上分配大量的空间也是需要时间的

Stack vs Heap
函数调用

- 当你的代码调用函数时，值被传入到函数（也包括指向 heap 的指针）。函数本地的变量被压到 stack 上。当函数结束后，这些值会从 stack 上弹出

Stack vs Heap
所有权存在的原因

- 所有权解决的问题：
  - 跟踪代码的哪些部分正在使用 heap 的哪些数据
  - 最小化 heap 上的重复数据量
  - 清理 heap 上未使用的数据以避免空间不足。
- 一旦你懂的了所有权，那么就不需要经常去想 stack 或 heap 了。
- 但是知道管理 heap 数据是所有权存在的原因，这有助于解释它为什么会这样工作。

所有权规则

- 每个值都有一个变量，这个变量是该值的所有者
- 每个值同时只能有一个所有者
- 当所有者超出作用域（scope）时，该值将被删除。

变量作用域

- Scope 就是程序中一个项目的有效范围
- （例子）

String 类型

- String 比那些基础标量数据类型更复杂
- 字符串字面值：程序里手写的那些字符串值。它们是不可变的
- Rust 还有第二种字符串类型：String。
  - 在 heap 上分配。能够存储在编译时未知数量的文本。

创建 String 类型的值

- 可以使用 from 函数从字符串字面值创建出 String 类型
- let s = String::from("hello");
  - “::”表示 from 是 String 类型下的函数
- 这类字符串是可以被修改的（例子）
- 为什么 String 类型的值可以修改，而字符串字面值却不能修改？
  - 因为它们处理内存的方式不同

内存和分配

- 字符串字面值，在编译时就知道它的内容了，其文本内容直接被硬编码到最终的可执行文件里
  - 速度快、高效。是因为其不可变性。
- String 类型，为了支持可变性，需要在 heap 上分配内存来保存编译时未知的文本内容：
  - 操作系统必须在运行时来请求内存
    - 这步通过调用 String::from 来实现
  - 当用完 String 之后，需要使用某种方式将内存返回给操作系统
    - 这步，在拥有 GC 的语言中，GC 会跟踪并清理不再使用的内存
    - 没有 GC，就需要我们去识别内存何时不再使用，并调用代码将它返回。
      - 如果忘了，那就浪费内存。
      - 如果提前做了，变量就会非法
      - 如果做了两次，也是 Bug。必须一次分配对应一次释放

内存和分配

- Rust 采用了不同的方式：对于某个值来说，当拥有它的变量走出作用范围时，内存会立即自动的交还给操作系统。（例子）
- drop 函数

变量和数据交互的方式：移动（Move）

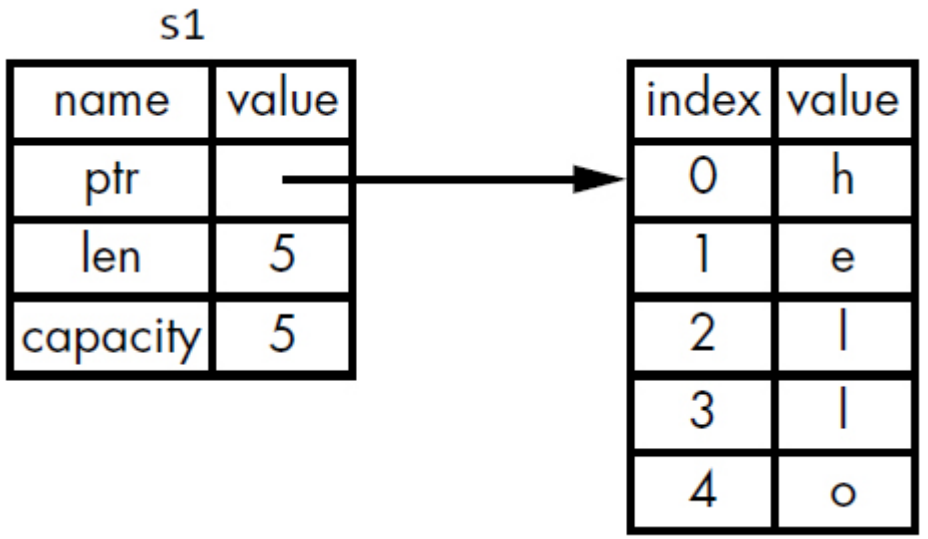
- 多个变量可以与同一个数据使用一种独特的方式来交互
- let x = 5;
  let y = x;
- 整数是已知且固定大小的简单的值，这两个 5 被压到了 stack 中

变量和数据交互的方式：移动（Move）
String 版本

- let s1 = String::from("hello");
  let s2 = s1;
- 情况和前面的例子不同

变量和数据交互的方式：移动（Move）String 版本

- 一个 String 由 3 部分组成：
  - 一个指向存放字符串内容的内存的指针
  - 一个长度
  - 一个容量
- 上面这些东西放在 stack 上。
- 存放字符串内容的部分在 heap 上
- 长度 len，就是存放字符串内容所需的字节数
- 容量 capacity 是指 String 从操作系统总共获得内存的总字节数



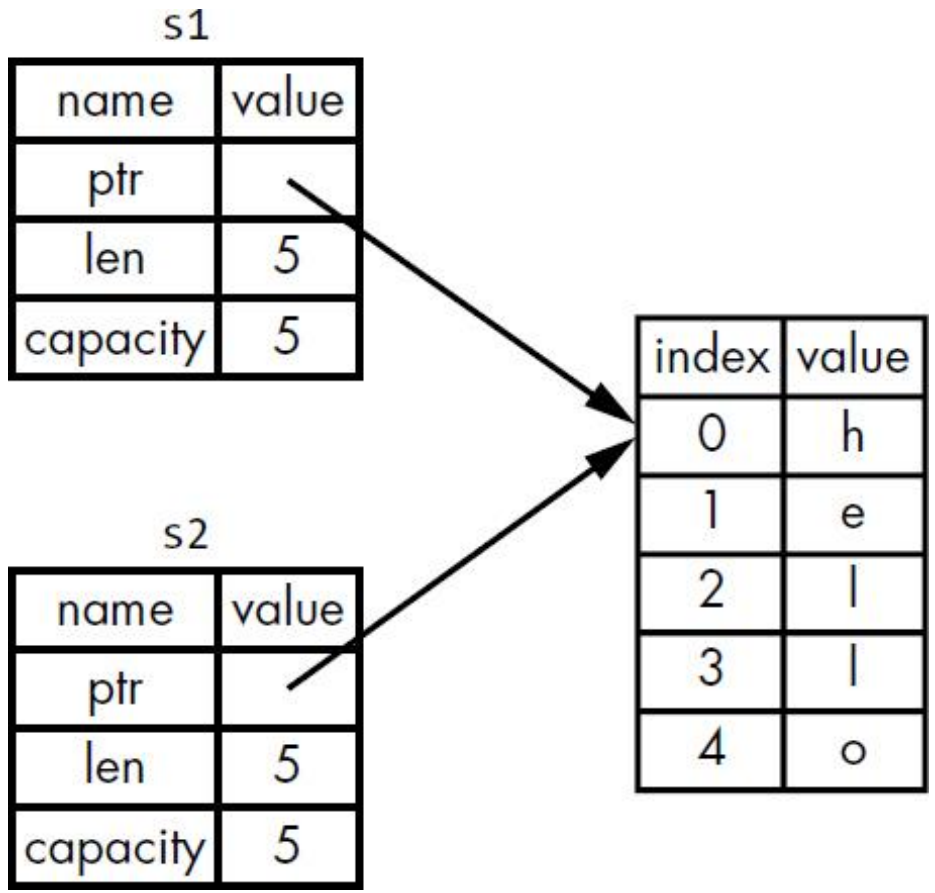
变量和数据交互的方式：移动（Move）String 版本

- 当把 s1 赋给 s2，String 的数据被复制了一份：
  - 在 stack 上复制了一份指针、长度、容量
  - 并没有复制指针所指向的 heap 上的数据

- 当变量离开作用域时，Rust 会自动调用 drop 函数，并将变量使用的 heap 内存释放。

- 当 s1、s2 离开作用域时，它们都会尝试释放相同的内存：
  - 二次释放（double free）bug

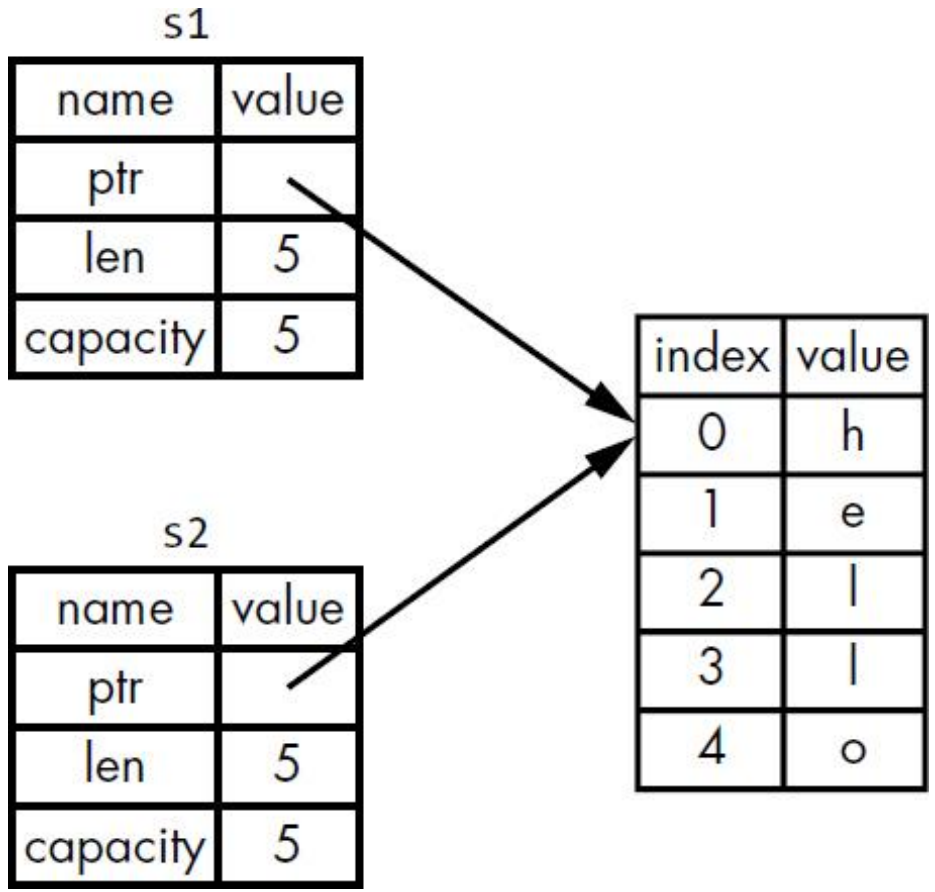
```
let s1 = String::from("hello");
let s2 = s1;
```



变量和数据交互的方式：移动（Move）String 版本

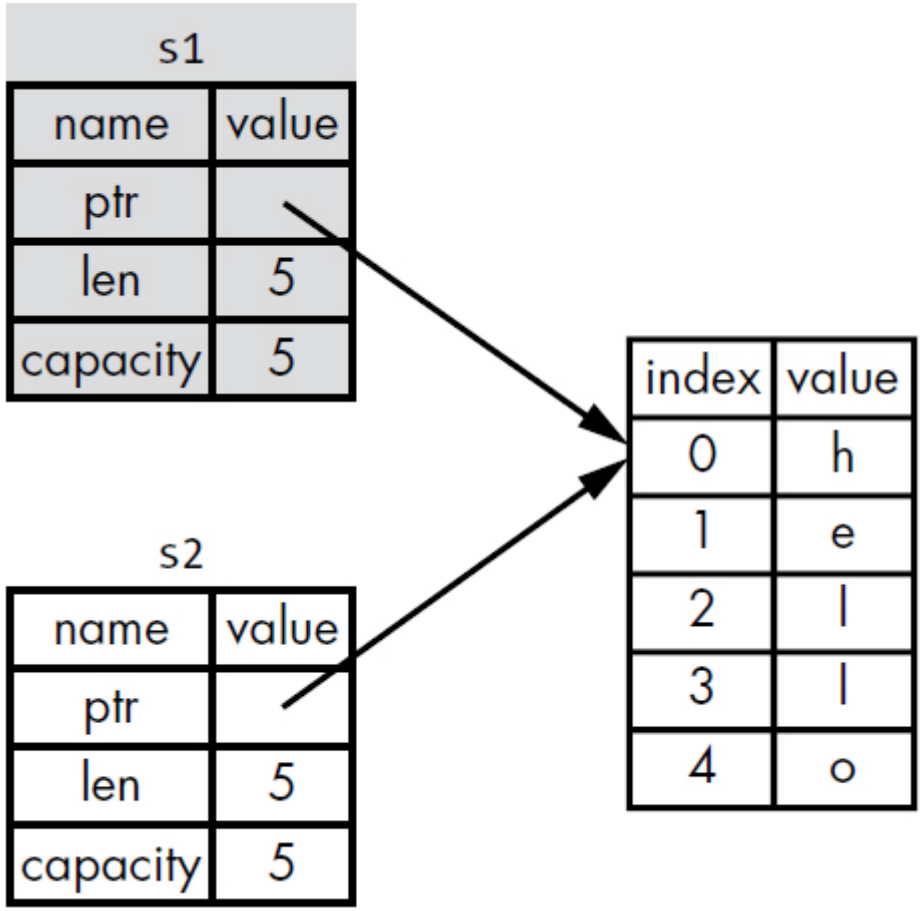
- 为了保证内存安全：
  - Rust 没有尝试复制被分配的内存
  - Rust 让 s1 失效。
    - 当 s1 离开作用域的时候，Rust 不需要释放任何东西
- 试试看当 s2 创建以后 再使用 s1 是什么效果（例子）

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



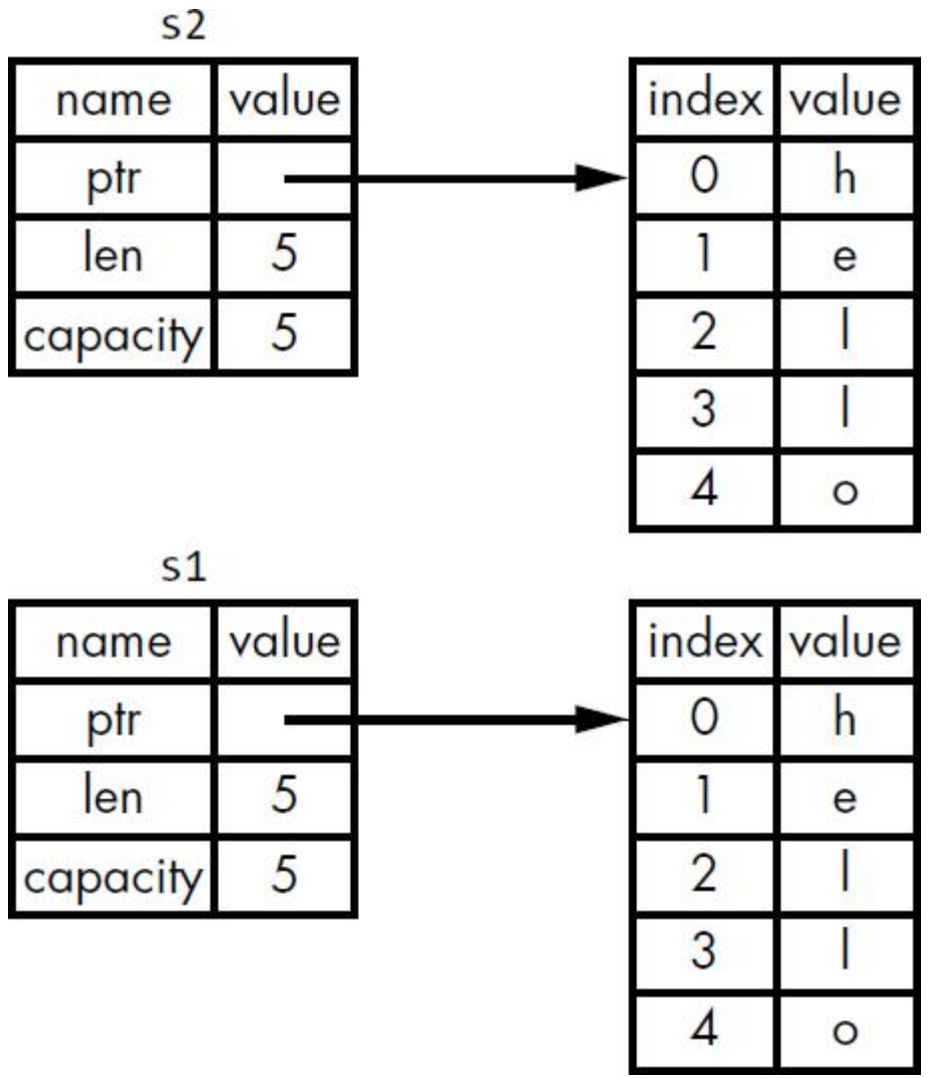
变量和数据交互的方式：移动（Move）

- 浅拷贝（shallow copy）
- 深拷贝（deep copy）
- 你也许会将复制指针、长度、容量视为浅拷贝，但由于 Rust 让 s1 失效了，所以我们用一个新的术语：移动（Move）
- 隐含的一个设计原则：Rust 不会自动创建数据的深拷贝
  - 就运行时性能而言，任何自动赋值的操作都是廉价的



变量和数据交互的方式：克隆（Clone）

- 如果真想对 heap 上面的 String 数据进行深度拷贝，而不仅仅是 stack 上的数据，可以使用 clone 方法（以后再细说，先看个例子）



Stack 上的数据：复制

- （例子）
- Copy trait，可以用于像整数这样完全存放在 stack 上面的类型
- 如果一个类型实现了 Copy 这个 trait，那么旧的变量在赋值后仍然可用
- 如果一个类型或者该类型的一部分实现了 Drop trait，那么 Rust 不允许让它再去实现 Copy trait 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

所有权与函数

- 在语义上，将值传递给函数和把值赋给变量是类似的：
  - 将值传递给函数将发生**移动**或**复制**
- （例子）

返回值与作用域

- 函数在返回值的过程中同样也会发生所有权的转移
- （例子）
- 一个变量的所有权总是遵循同样的模式：
  - 把一个值赋给其它变量时就会发生移动
  - 当一个包含 heap 数据的变量离开作用域时，它的值就会被 drop 函数清理，除非数据的所有权移动到另一个变量上了

如何让函数使用某个值，但不获得其所有权？

- （例子）
- Rust 有一个特性叫做“引用（Reference）”

再见