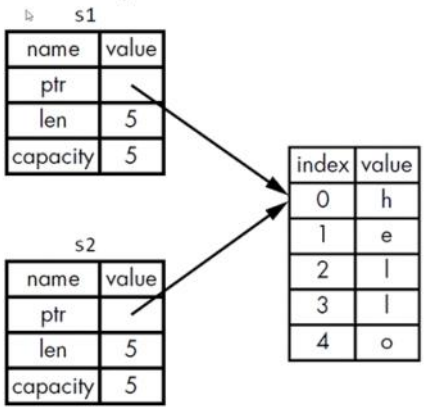
03

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;
```



S1所有权被放出来了，就失效了

- 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

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

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

```rust
fn main() {
    let s = String::from("Hello World");

    take_ownership(s);

    let x = 5;

    makes_copy(x);

    println!("x: {}", x);
}

fn take_ownership(some_string: String) {
    println!("{}", some_string)
}

fn makes_copy(some_number: i32) {
    println!("{}", some_number);
}
```

s被移动到函数中，s不再有效

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

```rust
fn main() {
    let s1 = gives_ownership();

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

    let s3 = takes_and_gives_back(s2);
}

fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string
}

fn takes_and_gives_back(a_string: String) -> String {
    a_string
}
```

S1来自函数体中的变量的所有权转移，s3来自s2的所有权转移。

- 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

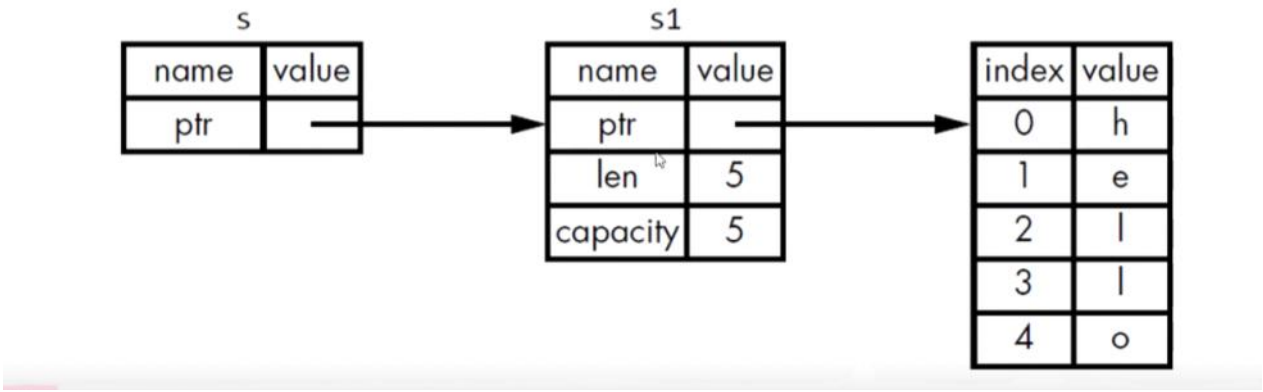
如何让函数使用某个值，但不获得其所有权？

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();

    (s, length)
}
```

引用与借用

4.2 引用和借用

- （例子）
- 参数的类型是 &String 而不是 String
- & 符号就表示引用：允许你引用某些值而不取得其所有权



相当于指针

```
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.len()
}
```

借用

- 我们把引用作为函数参数这个行为叫做借用
  - 是否可以修改借用的东西？（例子）
    - 不行
  - 和变量一样，引用默认也是不可变的

```rust
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.push_str(", world");
    s.len()
}
```

此处无法增加，word，借用不可修改

可变引用（在该变量前加mut）：

```rust
fn main() {
    let mut s1 = String::from("Hello");
    let len = calculate_length(&mut s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &mut String) -> usize {
    s.push_str(", world");
    s.len()
}
```

可变引用

- （例子）
- 可变引用有一个重要的限制：在特定作用域内，对某一块数据，只能有一个可变的引用。
  - 这样做的好处是可在编译时防止数据竞争。
- 以下三种行为下会发生数据竞争：
  - 两个或多个指针同时访问同一个数据
  - 至少有一个指针用于写入数据
  - 没有使用任何机制来同步对数据的访问
- 可以通过创建新的作用域，来允许非同时的创建多个可变引用（例子）

```rust
fn main() {
    let mut s = String::from("Hello");
    let s1 = &mut s;
    let s2 = &mut s;

    println!("The length of '{}' is {}.", s1, s2);
}
```

此处s2引用报错

```rust
fn main() {
    let mut s = String::from("Hello");
    {
        let s1 = &mut s;
    }

    let s2 = &mut s;
}
```

不同作用域，所以存在多个可变引用

另外一个限制

- 不可以同时拥有一个可变引用和一个不变的引用
- 多个不变的引用是可以的

```rust
fn main() {
    let mut s = String::from("Hello");
    let r1 = &s;
    let r2 = &s;
    let s1 = &mut s;

    println!("{} {} {}", r1, r2, s1);
}
```

悬空引用 Dangling References

- 悬空指针（Dangling Pointer）：一个指针引用了内存中的某个地址，而这块内存可能已经释放并分配给其它人使用了。
- 在 Rust 里，编译器可保证引用永远都不是悬空引用：
  - 如果你引用了某些数据，编译器将保证在引用离开作用域之前数据不会离开作用域

```rust
fn main() {
    let r = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

编译时就会报错，因为s出作用域，内存被释放，但函数想返回他的引用，这会指向一个释放的内存空间，显然矛盾

引用的规则

- 在任何给定的时刻，只能满足下列条件之一：
  - 一个可变的引用
  - 任意数量不可变的引用
- 引用必须一直有效