The Slice Type

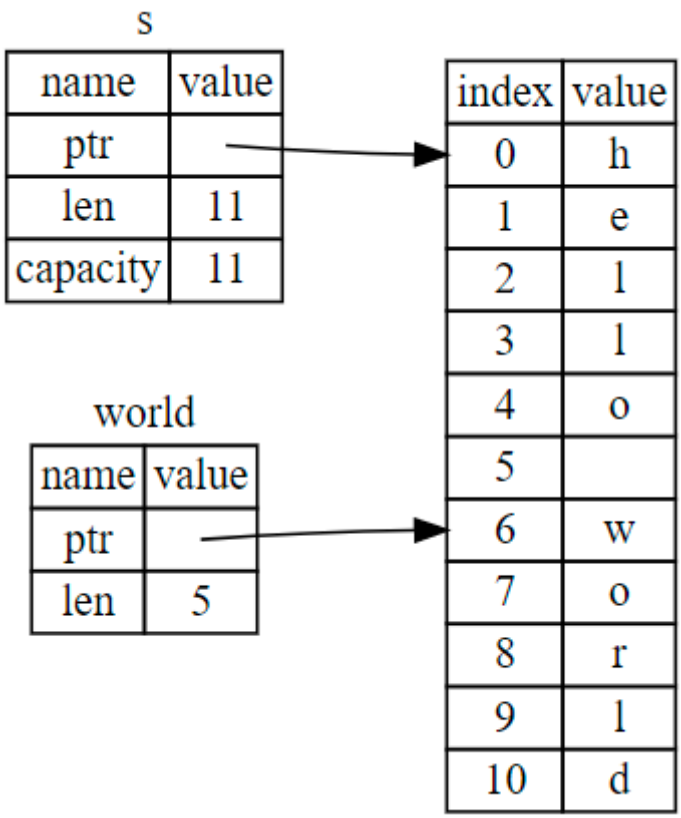
- Slices顾名思义就是像python中的string slice一样，截取一段值。注意的是，rust中的Slices是reference, 这意味着它并不会转移ownership。
- 假设我们现在想写一个程序，这个程序是找出string中第一个word. 我们的第一种实现方法可以是：

```
fn first_word(s: &String) -> usize{
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate(){
        if item == b' '{
            return i;
        }
    }

    s.len()
}
```

  - 这个方法的核心是检查出string中的第一个空格
  - `let bytes = s.as_bytes()`是将string 转化成了array of bytes.
  - `bytes.iter()`是创造了一个iterator，返回的是array中的每一个元素
  - `.enumerate()`则是打包了`.iter()`返回的结果，然后返回一个个tuple：tuple的结构是：(`index`,`reference to the element`).
  - 然后我们用for (i, &item),取出tuple中的元素（一个destructure的过程）
- 上面的方法只是找到第一个word结束的index，接下来，我们像直接用string slices来解决问题：

```
fn main() {
    let s = String::from("Hello, world!");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```

  - 这样的话，world在这还是相当于是一个reference，意味着并不会在out of scope的时候drop掉值：



- 我们可以把`first_word()`函数改进一下

```
fn main() {
    let s = String::from("Hello, world!");

    let word = first_word(&s);

    println!("the first word is {}", word);
}

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

    for (i, &item) in bytes.iter().enumerate(){
        if item == b' '{
            return &s[0..i];
        }
    }

    &s[..]
}
```

- slice都是immutable borrow的，这意味着在上一章中规定的不能同时又mutable reference和immutable reference出现也是适用的。比如下面的代码会报错：

```
fn main() {
    let s = String::from("Hello, world!");

    let word = first_word(&s);

    s.clear();
```

```
    println!("the first word is {}", word);
}

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

    for (i, &item) in bytes.iter().enumerate(){
        if item == b' '{
            return &s[0..i];
        }
    }

    &s[..]
}
```

- `s.clear()`实际上是用了mutable reference，这就会word的immutable borrow冲突了

在这里，我们可以更好的理解String Literals了

- `let s = "Hello, world!";`实际上s就是个 slice, 这也是为什么 string slices是immutable的。string literals本质上是个`&str`
- 但对于指向string的pointer，它本质上是pointer， 它理论上是可以mutate的。
- 看下面这个对比例子：

```
fn main() {
    let mut s1 = String::from("Hello, world!");

    s1.clear();

    let mut s1 = "hi world!";

    s2.clear();
}
```

报错：

```
error[E0599]: no method named `clear` found for reference `&str` in the
current scope
 --> src/main.rs:8:8
  |
8 |     s2.clear();
  |        ^^^^^ method not found in `&str`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `slice` due to previous error
```

- `s1.clear()`是可以完美进行的，但`s2.clear()`就会报错。根本原因是一个是pointer，一个是slice

- 非常要注意的是`&String`和`&str`不是同一个东西，所以下面的程序会报错：

```
fn main() {
    let mut s2 = "hi world!";
```

```
    let word = first_word(&s2);
}

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

    for (i, &item) in bytes.iter().enumerate(){
        if item == b' '{
            return &s[0..i];
        }
    }
    &s[..]
}
```

报错：

```
error[E0308]: mismatched types
 --> src/main.rs:8:27
  |
8 |     let word = first_word(&s2);
  |                           ^^^ expected struct `String`, found `&str`
  |
  = note: expected reference `&String`
             found reference `&&str`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `slice` due to previous error
```

- 但是！下面这样是可以的：

```
fn main() {
    let mut s1 = String::from("Hello, world!");

    let mut s2 = "hi world!";

    let word = first_word(&s1);
}

fn first_word(s: &str) -> &str{
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate(){
        if item == b' '{
            return &s[0..i];
        }
    }
    &s[..]
}
```

  - 原因是first_word(&s1) 会直接把&String识别成一个whole slice。所以，&s1在这里就相当于是一个slice。
- 所以，我们在规定一个函数的时候，更robust的写法是用 `fn first_word(s: &str) -> &str`,因为这样可以容忍输入&String, 或者&str.

---

array slice

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);

println!("the slice is {}", slice[0]);
```

output:

```
the slice is 2
```