rust系统内存模型

rust 系统内存模型

- 应用的内存模型:
  - Stack
  - Heap
  - Static
  - Code

```
static mut X: i32 = 0; //通过static修饰被分配在static区中
unsafe { print!("{:p}", &X); }//0x10019e62c
let str = "hello"; //特殊的&str内存会在被分配在static区

let string = String::from("hello");
let x1 = string.deref();//但是通过解引用的方式则会指向堆上
println!("x1 is {:p}",x1);//0x60000296c050

let x2= Box::new(5); //智能指针会在堆上分配内存
println!("{:p}",Box::into_raw(x2));//0x60000296c070

let x3 = 5;//基础类型数据在栈空间上创建
println!("{:p}", &x3);//0x16b42eca4
```

- 基础类型（在栈空间分配）
  - 所有的整型
  - bool
  - char
  - 所有的浮点
  - 元组（本身在栈上分配，也可以保存指针）
  - 数组

  额外在栈空间分配的还有切片和函数指针，结构体，枚举

  关于copy trait 拥有了 Copy trait，它的变量就可以在赋值给其他变量之后保持可用性

  基础类型中整型和浮点型bool，bool，char，元组中所有字段都实现copy trait 则元组也是copy trait， 固定的数组，函数指针，都默认实现copy trait，

```
        let mut x1 = [String::from("a")];
        let x2 = x1;//发生了move 而不是copy，在这后面将无法在使用x1
        let mut x3 = ["b"];//数组存储指向在常量区的字符串字面量的指针
        let x4 = x3;//发生了copy
        println!("{:?}", x3);
```

一个vector v 和一个数组 a在内存和切片 sa sv在内存上的分布

```
let v = vec![0.0,0.707,1.0,0.707];
let a = [0.0,0.707,1.0,0.707];
let sv = &v;
let sa = &a;
```

string和&str和str在内存上分配

```
let noodle= "noodle".to_string();
let oodles = &noodle[1..];
let poodle = "ಠ_ಠ";
```

- 所有权和move

  在c++string在栈空间指向他堆上分配的buffer

```
fn print_padovan() {
let mut padovan = vec![1,1,1];// padovan 在这里被声明
 for i in 3..10 {
let next = padovan[i-3] + padovan[i-2];
padovan.push(next);
}
println!("p(1..10) = {:?}",padovan);
}// padovan 离开作用域, 它的内存被释放
```

```
let point =Box::new((0.625,0.5));
let label = format!("{:?}",point);
```

```
struct Person {
name: String, //指针指向堆上分配
birth: i32,
}
let mut composers = Vec::new();
composers.push(Person {
name: "Palestrina".to_string(),
birth: 1525,
});
composers.push(Person {
name: "Dowland".to_string(),
birth: 1563,
});
composers.push(Person{
name: "Lully".to_string(),
birth:1632
});
for composer in &composers {
println!("{}, born {}", composer.name, composer.birth);
}

```

moves

```
s=["udon","ramen","soba"];
t=s;
u=s;
```

在python 中字符串数组在内存中的分配

```
vector<string> s ={"udon","ramen","soba"};
vector<string> t =s;
vector<string> u =s;
```

```
let s= vec!["udon","ramen","soba"];
let t =s;
let u =s;//error s 已经被移动，所用权转移到了t中
//如果要像c++的内存分配
let u=s.clone()
let t=s.clone()
```

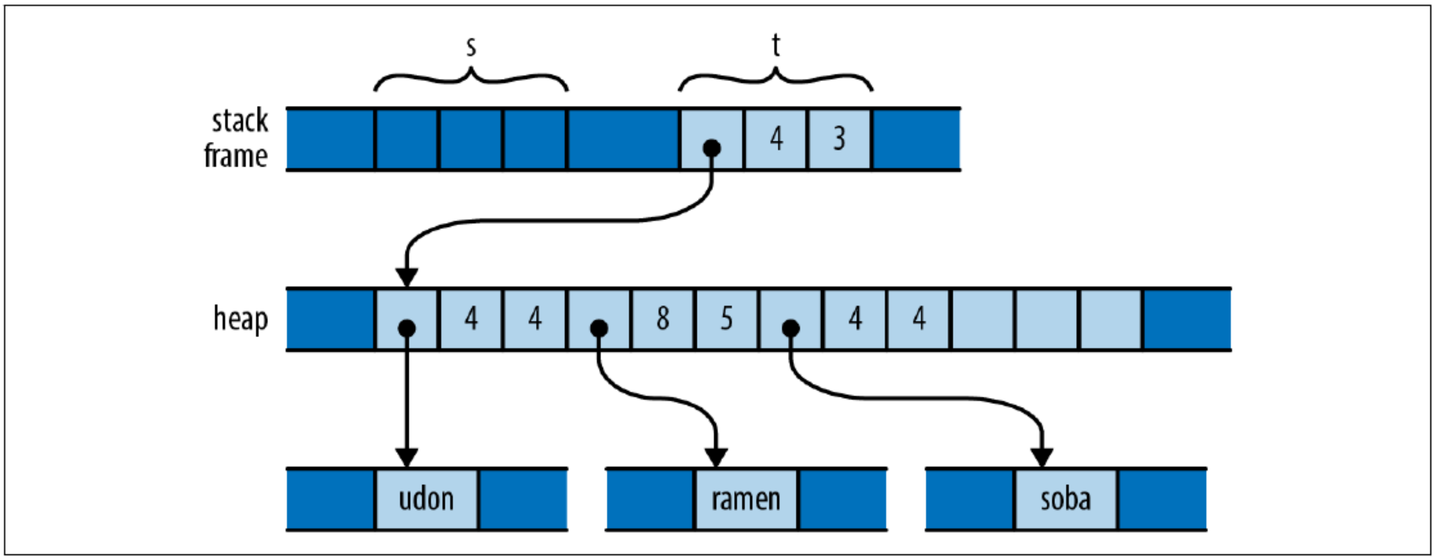

Figure 4-10. The result of assigning `s` to `t` in Rust

```
let mut noodle= vec!["udon"]
let soba = "soba".to_string()
//vec在堆上动态分配内存，拥有空的容量
```

```
noodle.push(soba)
```

```
let last = nooles.pop().unwrap
```

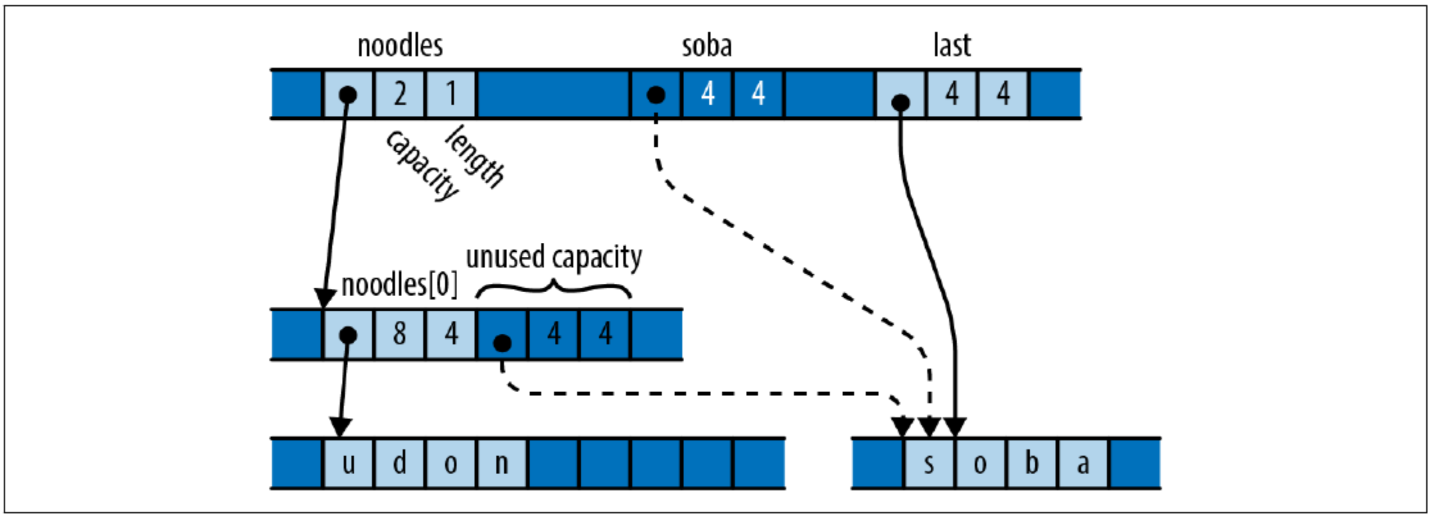

Figure 22-5. After popping an element from the vector into `last`

copy

```
let string1="somnambulance".to_string();
let string2=string1;//发生移动
let num1 =36;
let num2 =num1; //发生拷贝
```

Shared OwnerShip

```
//rc 共享所有权 类似于python
let s = Rc::new("shirataka".to_string());
let t =s.clone();
let u =s.clone();
```

references

```
let x = 10;
let y = 20;
let mut r = &x;
r=&y;
assert!(*r==10||*r==20);
```

```
#[derive(Debug)]
struct Point {
x: i32,
y: i32,
}
let point = Point { x: 1000, y: 729 };
let r = &point;
let rr = &r;
let rrr = &rr;
println!("{:?}", rrr)
```

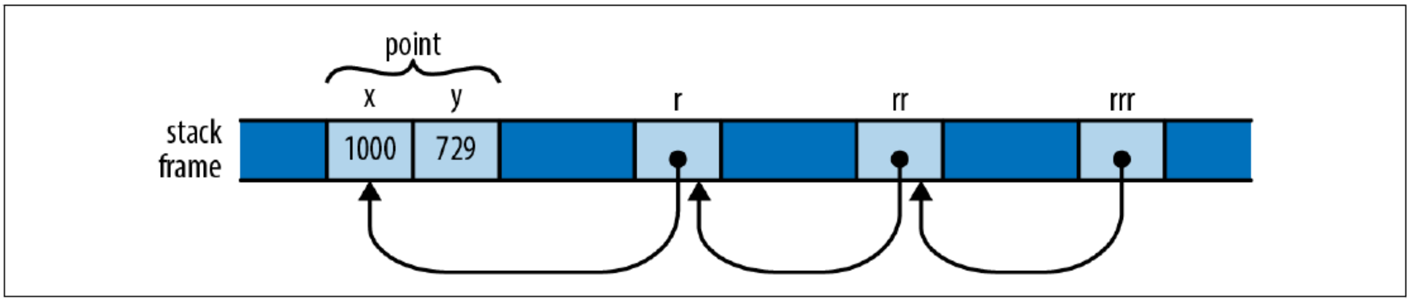

Figure 5-2. A chain of references to references

```
    x的生命周期不能超过他的范围不然会产生悬垂指针，需要延长x的生命周期包裹r![未知.png]
(/d/1Rzn1T0SolAi?f=0)
```

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;//  不可变借用
let aside = v;
r[0];//  所有权转移到了aside，所以这里会报错，无法再使用r
```

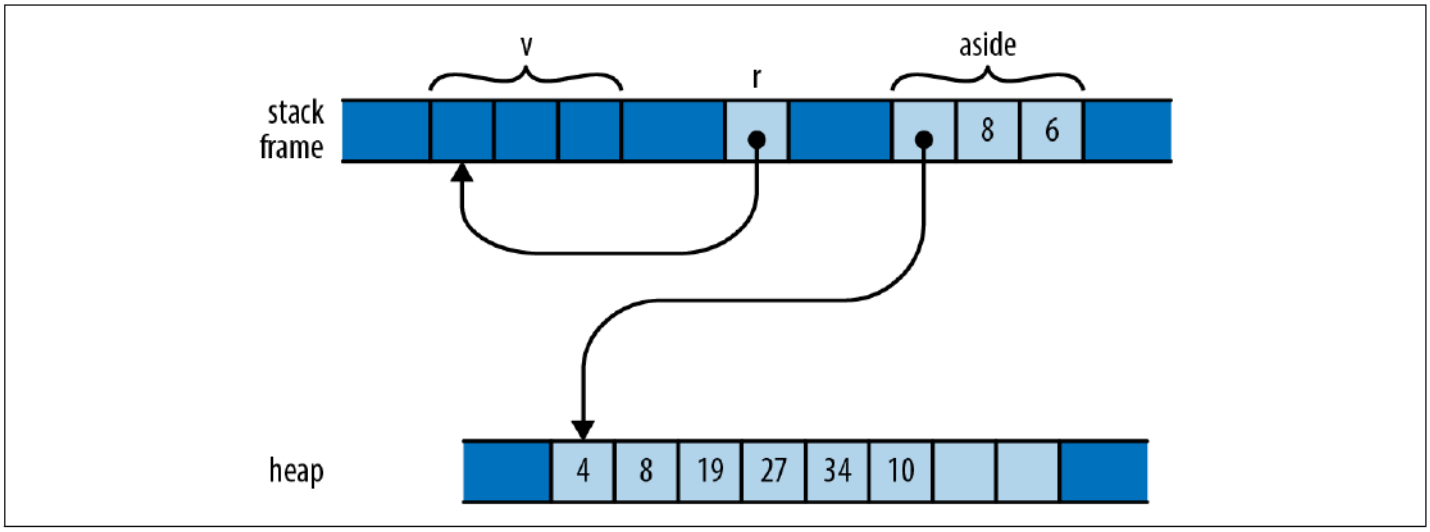

Figure 5-7. A reference to a vector that has been moved away

```
let mut wave = Vec::new();
let head = [0.0, 1.0];
let tail = [0.0, -1.0];
extend(&mut wave, &head);
extend(&mut wave, &tail);
println!("{:?}", wave);
extend(&mut wave, &wave);//wave变量不能同时作为可变借用和不可变借用
```

在生命周期内只能可以拥有多个不可变借用，和一个可变借用

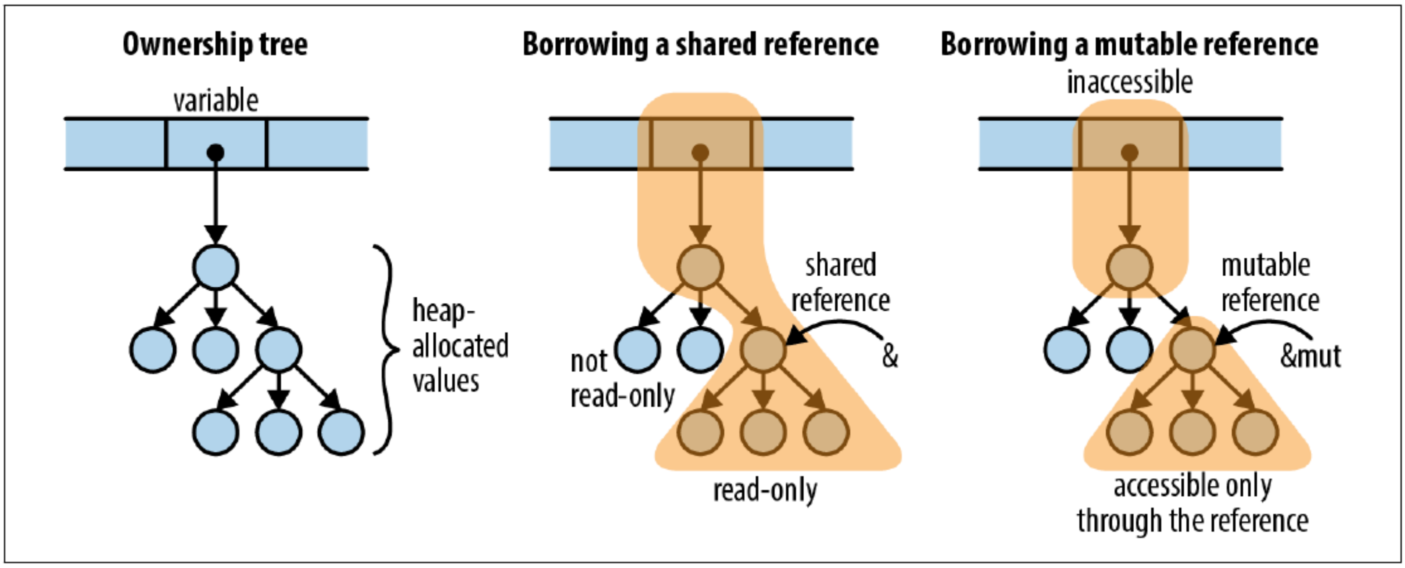

Figure 5-9. Borrowing a reference affects what you can do with other values in the same ownership tree

```rust
let mut x = "a".to_string();
let r1 = &x;
let r2 = &x;
let mut y = x;//move
y.push_str("b");
println!("{}",y);
println!("{},{},{}", r1, r2, x);//error

let mut x = 10;
let r1 = &x;
let r2 = &x;
let mut y = x;//copy
y+=10;
println!("{}",y);
println!("{},{},{}", r1, r2, x);//ok

let mut x = 10;
let r1 = &x;
let r2 = &x;
x+=10; //error 借用后的无法对其再进行赋值
println!("({}, {}, {}", r1, r2, x);


let mut w = (107,109);
let r =&mut w;
let r0 =&r.0;
let m1 = &mut r.1; //错误：不能同时多次借用'r'为可变的
println!("({:?}, {}, {})", r, r0, m1);

let mut v = (136,139);
let m = &mut v;
let m0 = &mut m.0;
*m0 = 137;
let r1 = &m.1;
v.1;// 错误 使用被借用v，不可变借用可以只读访问，可变借用只能&引用访问
println!("r1 is {}", r1);
```

trait object

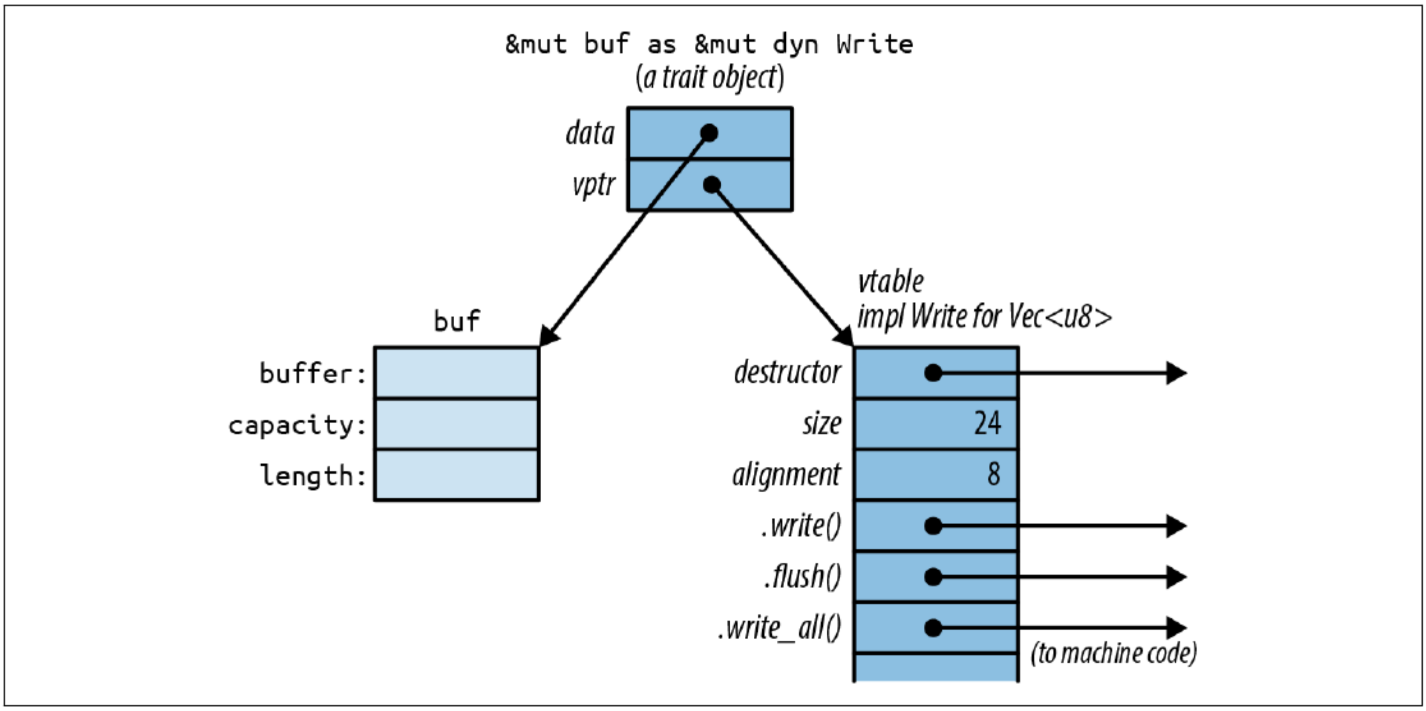

Figure 11-1. Trait objects in memory

通过box<dyn trait> 可以引用不确定大小的值，只要实现trait

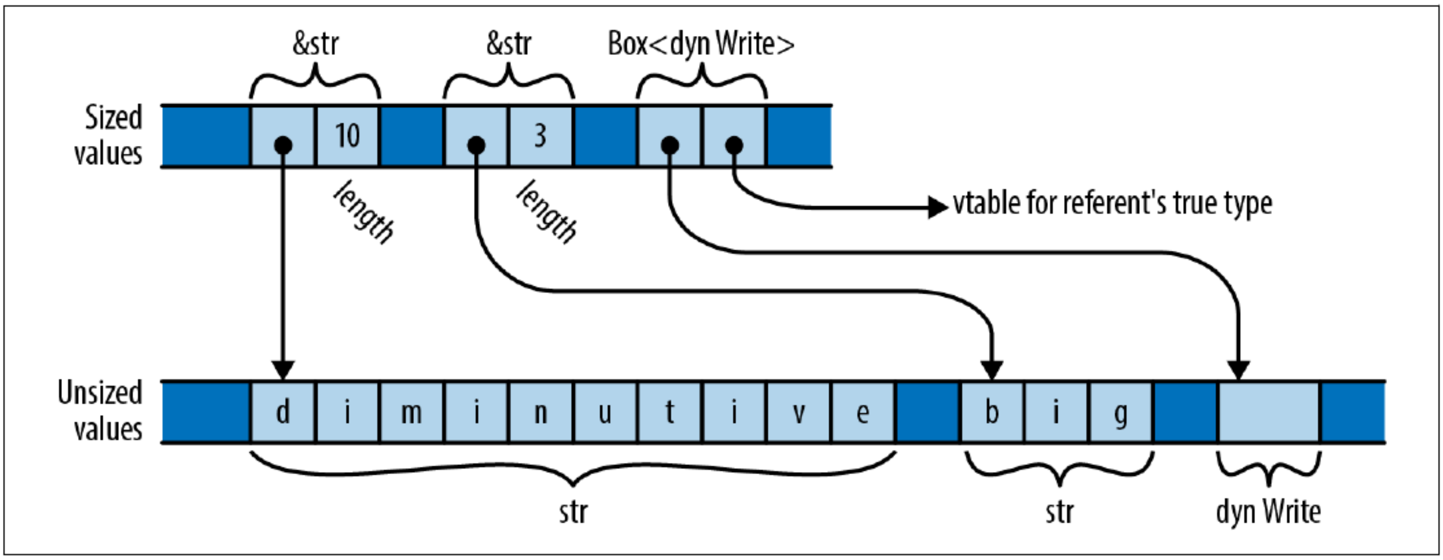

Figure 13-1. References to unsized values

闭包

a. 指向外部的变量

b. 外部的变量拷贝到内部

c. 使用普通函数，不调用外部变量

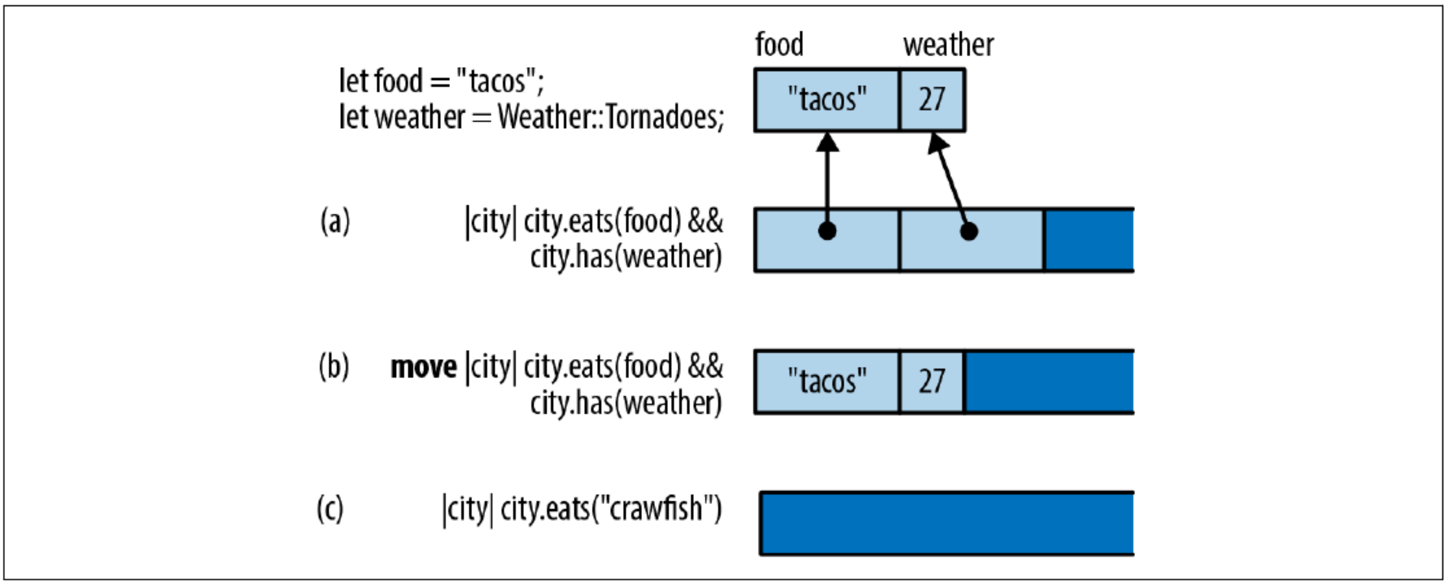

Figure 14-1. Layout of closures in memory

变量的3种状态再闭包中也是一样的，可以将闭包视作struct

ownship

mutable reference

immutable reference

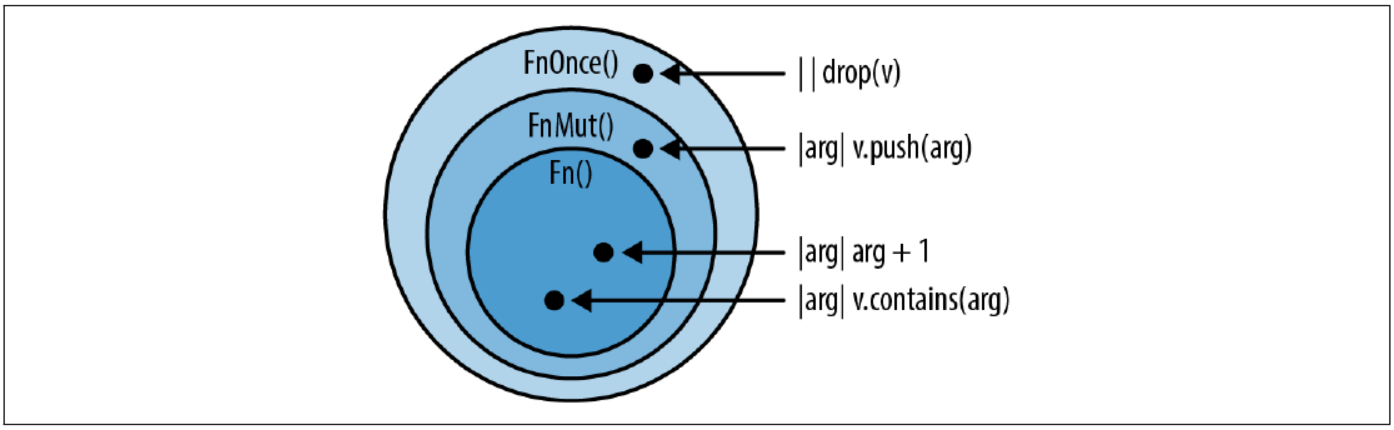

Figure 14-2. Venn diagram of the three closure categories

VecDeque 在内存中的结构

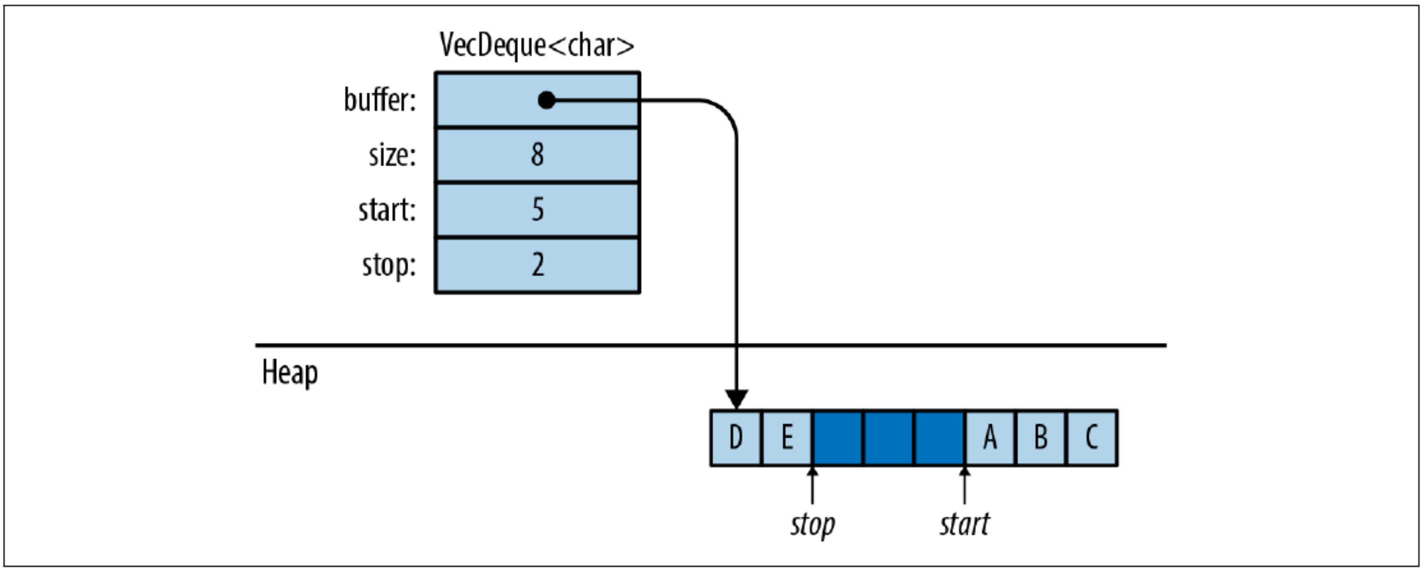

Figure 16-3. How a `VecDeque` *is stored in memory*

HashMap 在内存中的结构

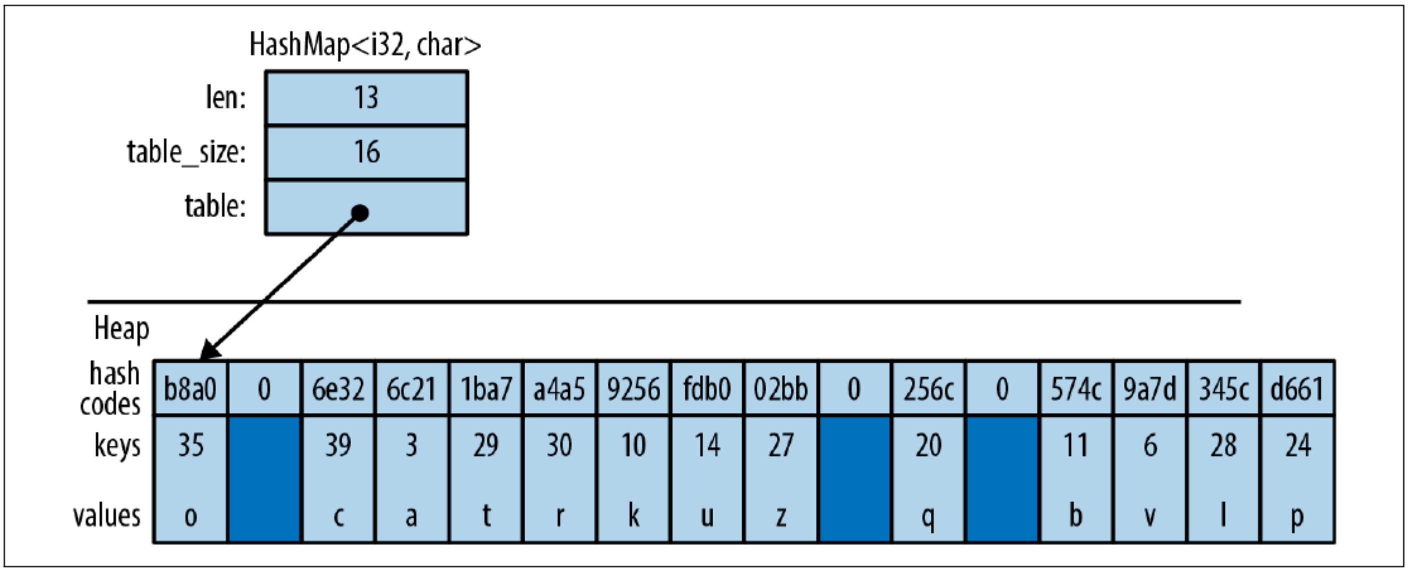

Figure 16-4. A `HashMap` *in memory*

BTreeMap

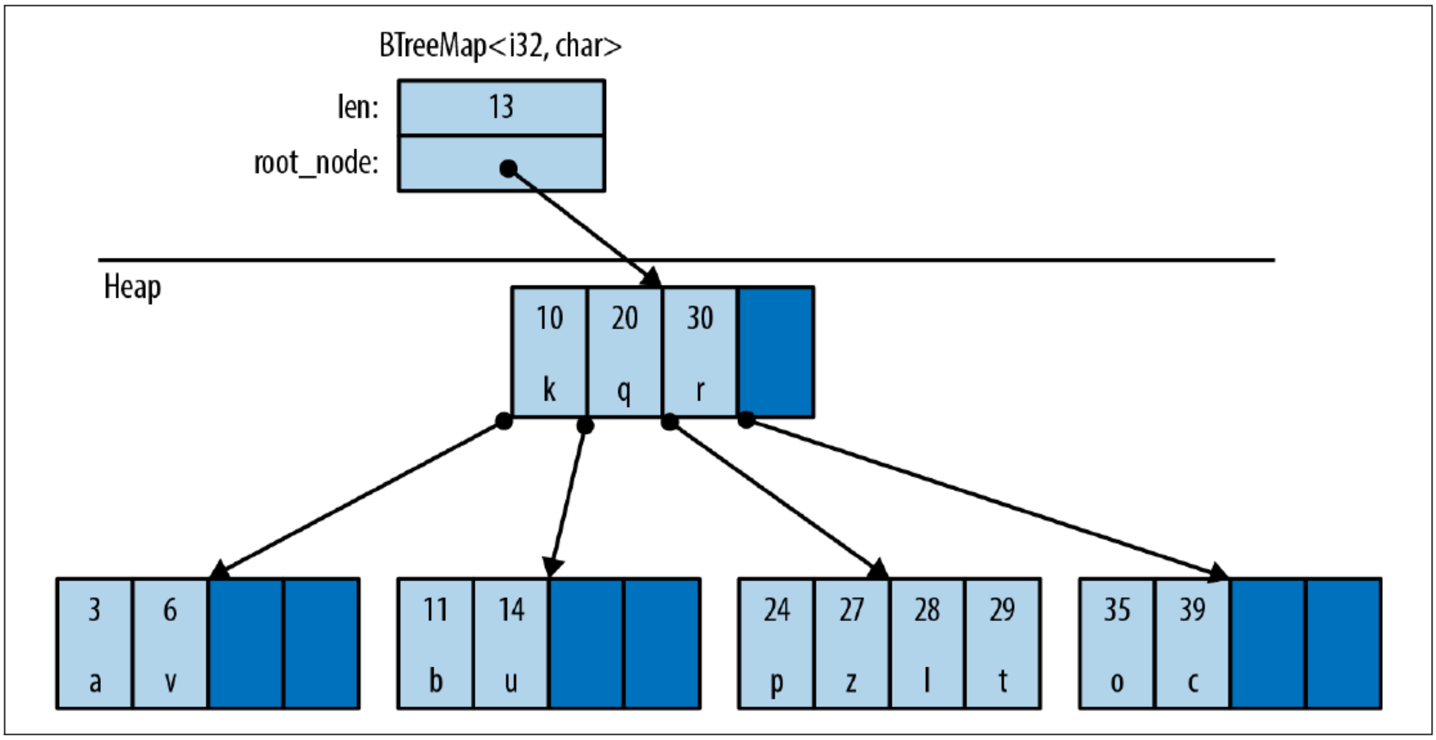

Figure 16-5. A `BTreeMap` in memory

自引用

栈上智能指针指向自身数据

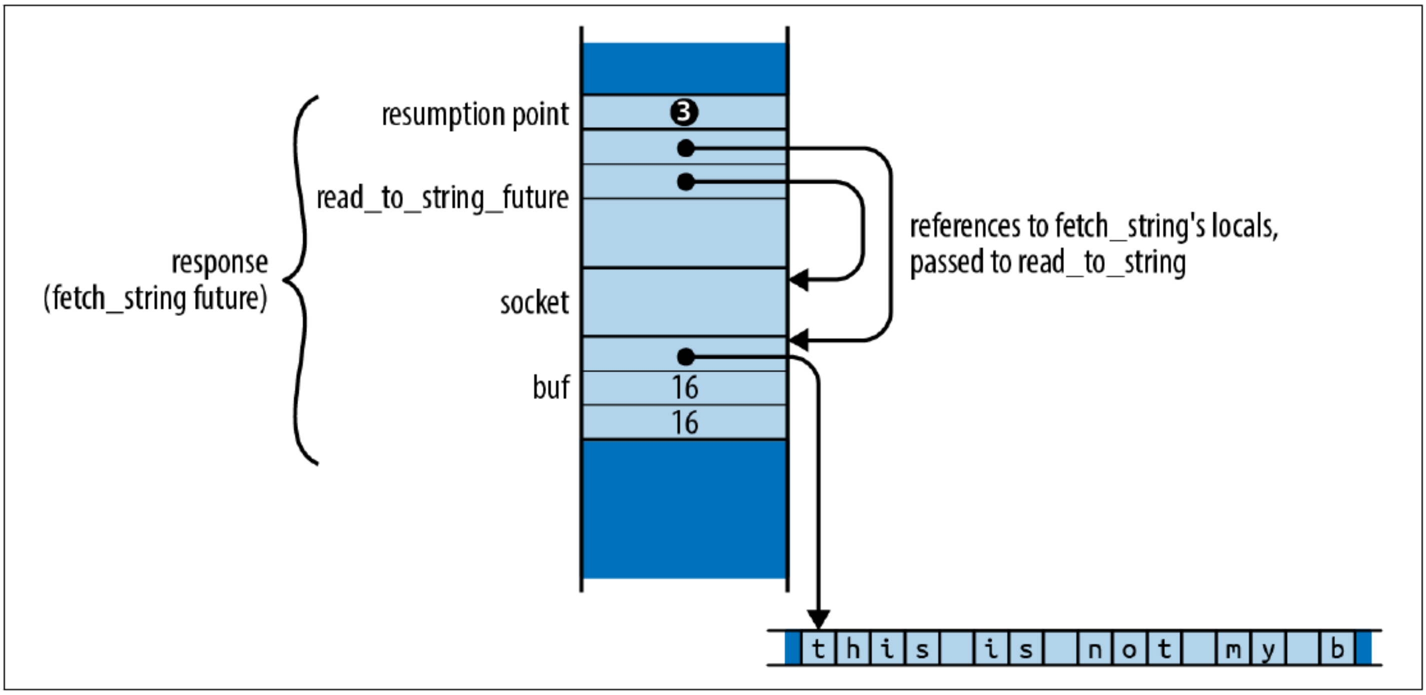

Figure 20-6. The same future, in the midst of awaiting `read_to_string`

当move的过程中，栈空间会拷贝，现在的指针指向了原来的栈空间的地址，一般使用pin来解决自引用

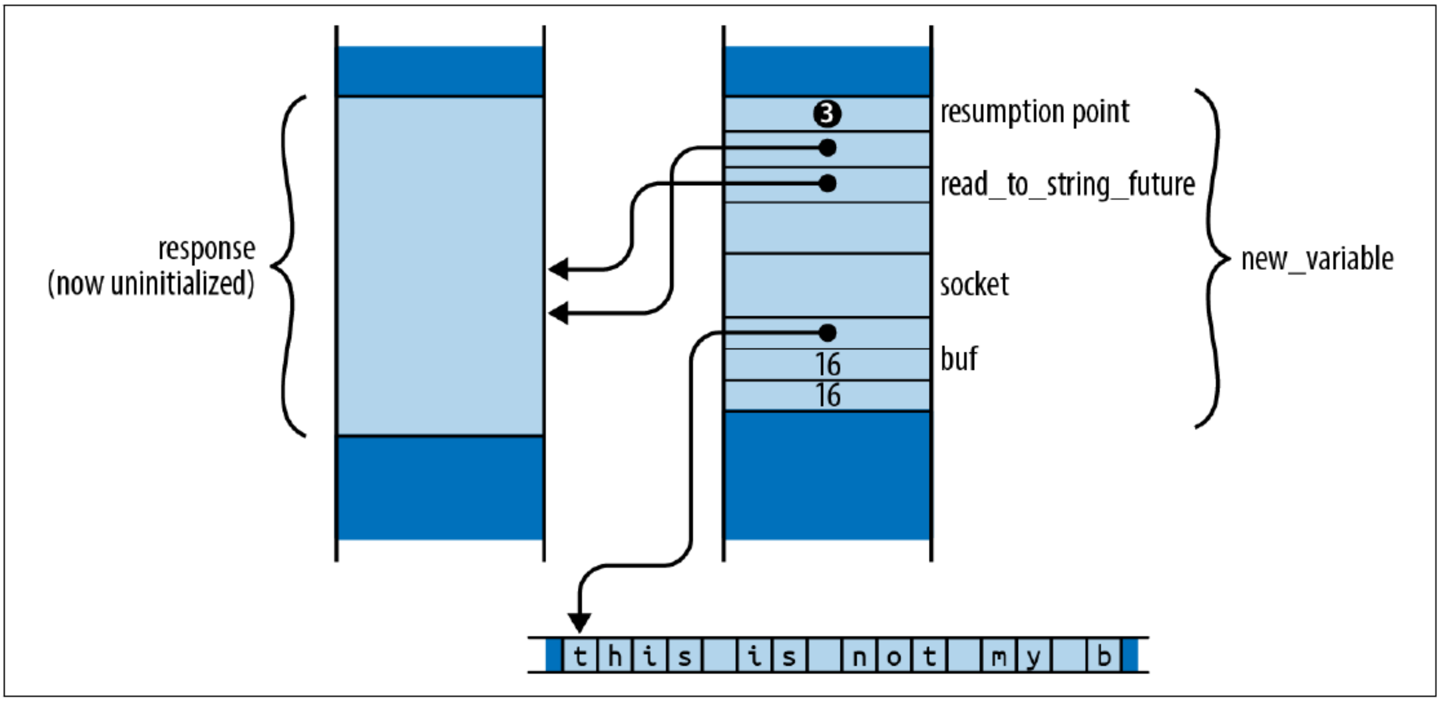

Figure 20-7. `fetch_string`'s future, moved while borrowed (Rust prevents this)