Rust 程序设计

张 伟

(zhangw.sei@pku.edu.cn)

北京大学 计算机学院

2025年2～6月

第 3 章：所有权与所有权转移

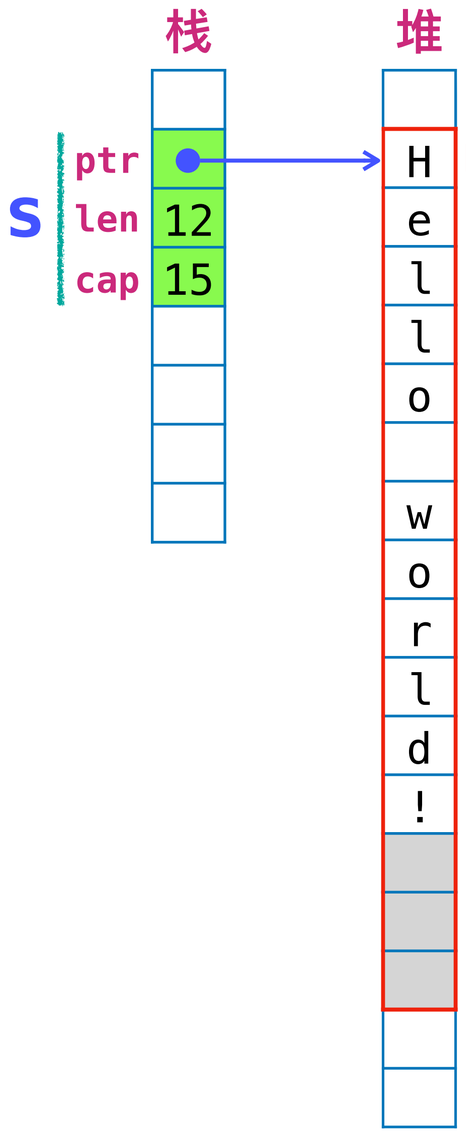
Ownership and Moves

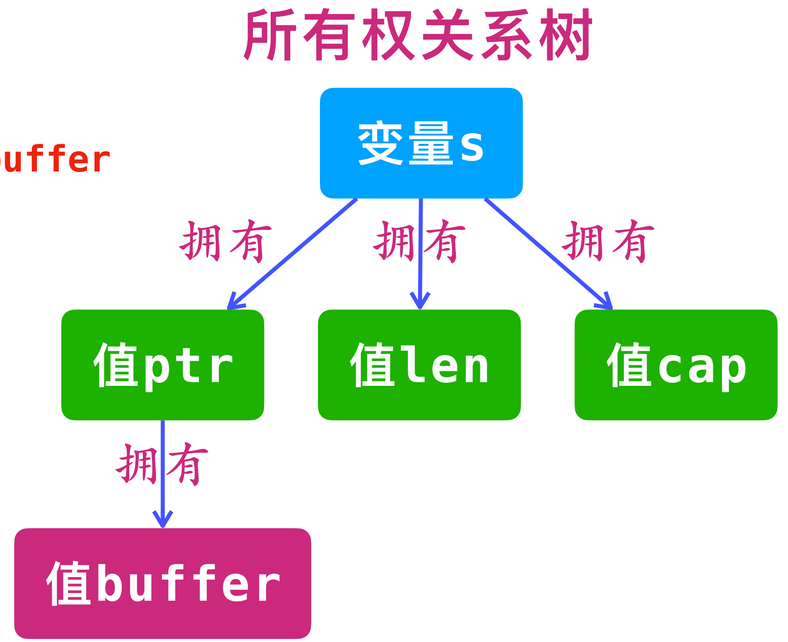
C++中的所有权：一个示例

```cpp
#include <iostream>

int main(){
    // 下面的这一对花括号定义了一个代码块（Code Block）
    {
        // 变量 s 的作用域在当前代码块内
        std::string s = "Hello world!";

        std::cout << s.length() << "\n"; //=> 12
        std::cout << s.capacity() << "\n"; //=> 15
    }
    // 在这里，无法访问到变量 s
}
```

C++中的所有权：一个示例

```
01 #include <iostream>
02
03 int main(){
04     // 下面的这一对花括号定义了一个代码块（Code Block）
05     {
06         // 变量 s 的作用域在当前代码块内
07         std::string s = "Hello world!";
08
09         std::cout << s.length() << "\n"; //=> 12
10         std::cout << s.capacity() << "\n"; //=> 15
11     }
12     // 在这里，无法访问到变量 s
13 }
```





当程序执行到第08～10行时
变量s在内存中的排布见右图

所有权的含义

当一个值的所有者不存在时
那么，这个值也不应该存在

在这个示例中，当执行到11行后
- 因为变量s已经不存在
- 则： 栈中三个值不再有意义
- 且： 堆中buffer空间被释放

在C++中制造一个悬空的指针（dangling pointer）

```cpp
#include <iostream>

int main(){
    // 声明一个指针类型的变量
    std::string *ptr;

    // 下面的这一对花括号定义了一个代码块（Code Block）
    {
        // 变量 s 的作用域在当前代码块内
        std::string s = "Hello world!";

        std::cout << s.length() << "\n"; //=> 12
        std::cout << s.capacity() << "\n"; //=> 15

        ptr = &s; //把变量 s 的地址赋值给 ptr
    }
    // 在这里，无法访问到变量 s
    // 但是，可以通过 ptr 访问到 s 原本占用的栈/堆空间
    std::cout << *ptr << "\n"; //=> Hello world!
}
```

C++的这种灵活性是你所需要的吗？

C++中的所有权贯彻的并不彻底

把Memory Safety的责任

完全交给程序员

不是一种好的选择

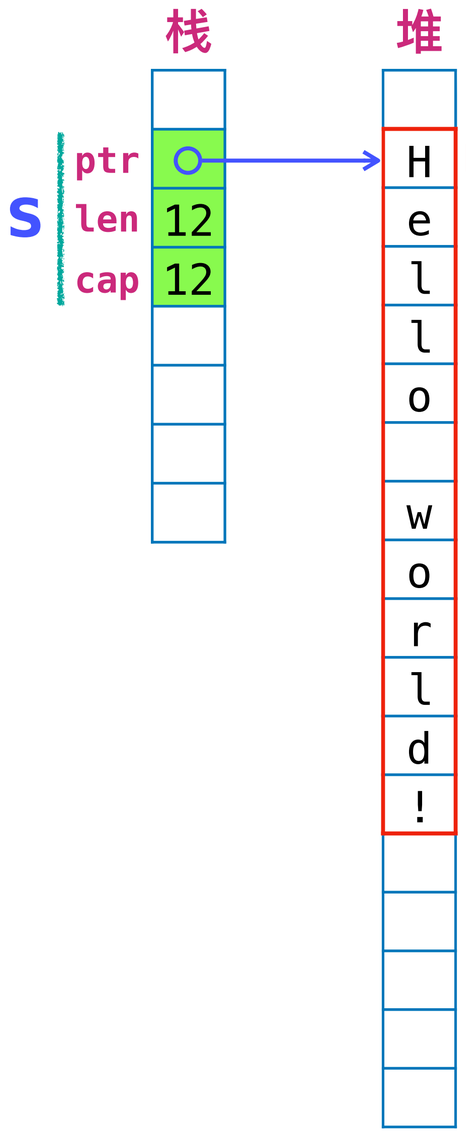
因为程序员会犯错

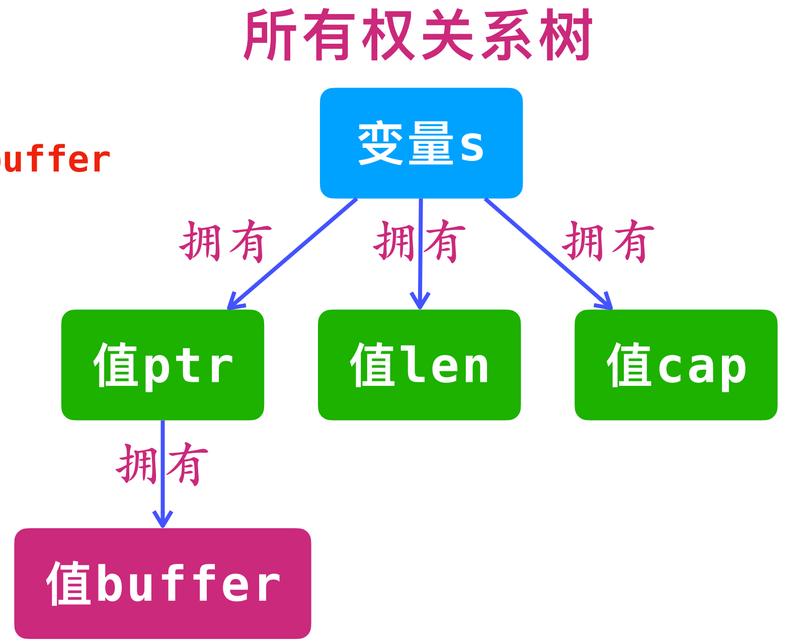
Rust中的所有权：一个示例

```rust


fn main() {
    // 下面的这一对花括号定义了一个代码块（Code Block）
    {
        // 变量 s 的作用域在当前代码块内
        let s = String::from("Hello world!");

        println!("{}", s.len()); //=> 12
        println!("{}", s.capacity()); //=> 12
    }
    // 在这里，无法访问到变量 s
}
```

Rust中的所有权：一个示例

```
01
02
03 fn main() {
04     // 下面的这一对花括号定义了一个代码块（Code Block）
05     {
06         // 变量 s 的作用域在当前代码块内
07         let s = String::from("Hello world!");
08
09         println!("{}", s.len()); //=> 12
10         println!("{}", s.capacity()); //=> 12
11     }
12     // 在这里，无法访问到变量 s
13 }
```





所有权的含义

当一个值的所有者不存在时
那么，这个值也不应该存在

在这个示例中，当执行到11行后
因为变量s已经不存在
则： 栈中三个值不再有意义
且： 堆中buffer空间被释放

当程序执行到第08～10行时
变量s在内存中的排布见右图

尝试在Rust中制造一个悬空的指针

```rust
fn main() {
    // 这里，声明了一个 &String 类型的变量
    let ptr: &String;

    // 下面的这一对花括号定义了一个代码块（Code Block）
    {
        // 变量 s 的作用域在当前代码块内
        let s = String::from("Hello world!");

        println!("{}", s.len()); //=> 12
        println!("{}", s.capacity()); //=> 12

        ptr = &s;
    }
    // 在这里，无法访问到变量 s
    println!("{}", ptr);
}
```

```
error[E0597]: `s` does not live long enough
  --> src/main.rs:13:15
   |
13 |         ptr = &s;
   |               ^^ borrowed value does not live long enough
14 |     }
   |     - `s` dropped here while still borrowed
15 |     // 在这里，无法访问到变量 s
16 |     println!("{}", ptr);
   |                     --- borrow later used here
```

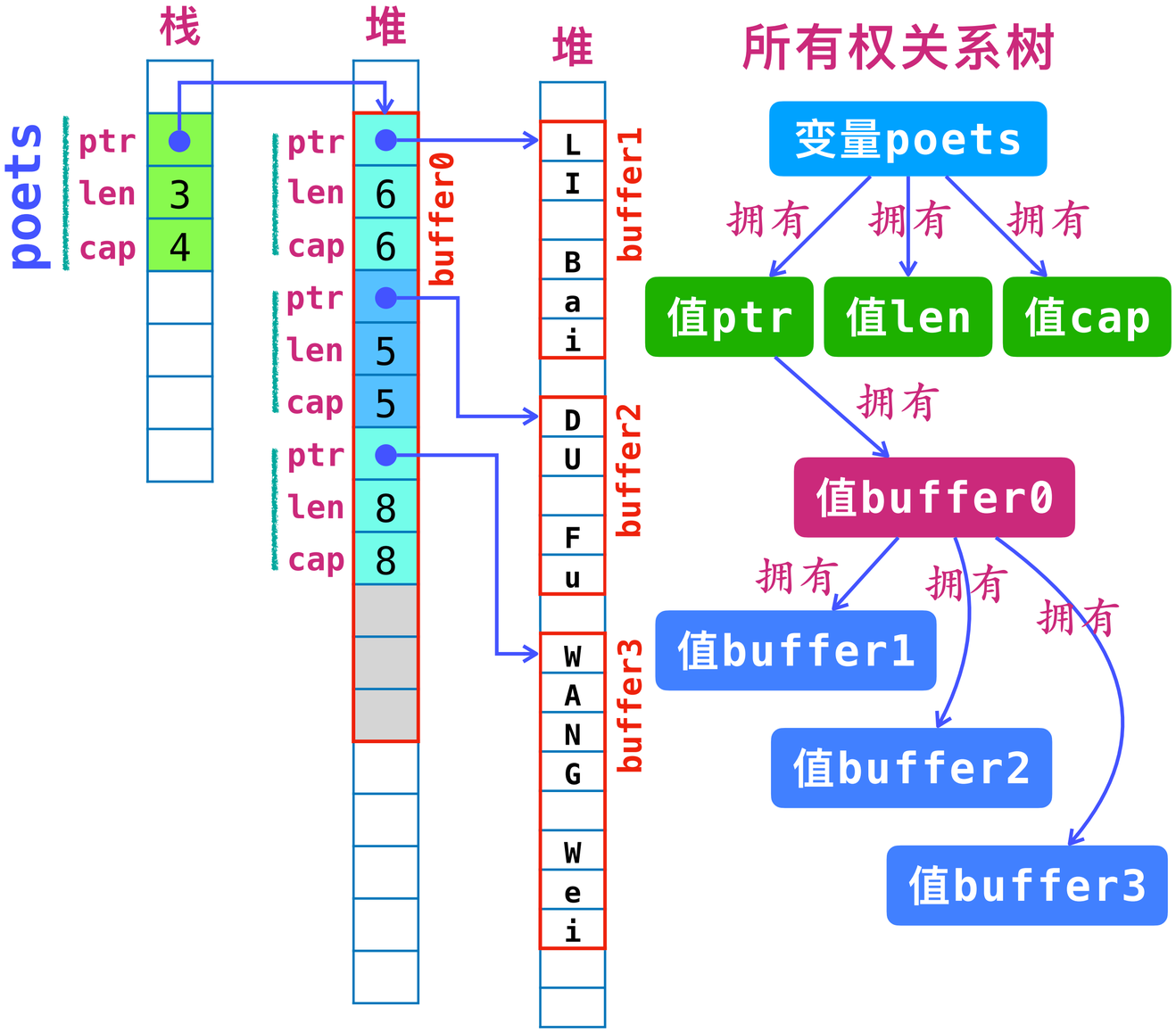
你对Rust编译器
感觉如何

Rust中的所有权：一个更复杂的示例

```
01 fn main() {
02     let mut poets = Vec::new();
03
04     poets.push(String::from("LI Bai"));
05     poets.push(String::from("DU Fu"));
06     poets.push(String::from("WANG Wei"));
07
08     println!("{}",poets.capacity()); //=> 4
09
10     for poet in &poets {
11         println!("{}", poet);
12     }
13 }
```

当程序执行到第07～12行时

变量poets在内存中的排布见右图



Rust中的所有权

在任意时刻

1	一个值具有唯一一个所有者
2	每一个变量，作为根节点，出现在一棵所有权关系树中
3	当一个变量离开当前作用域后 它所有权关系树中的所有值都无法再被访问 其中，所有存在于堆中的值，所占空间会被自动释放

Rust中的所有权：一些扩展/软化措施

1	**所有权转移**：一个值的所有权可以被转移给其他所有者
2	**简单变量豁免**：对于具有简单类型的变量（整数、浮点数、字符等），所有权规则不适用。称这些类型为**Copy types**
3	**引用计数指针类型**：Rust标准库提供了两种引用计数指针类型（ref-counted pointer types），允许一个值具有多个所有者（但需要满足一定的限制）
4	**borrow a ref to a value**：在不改变所有权的情况下，通过引用（ref），在满足一定限制的情况下，访问一个值

所有权转移

在Rust中，对于no-copy type的值而言，当发生如下操作时	
1	把这个值，赋给（assign to）一个变量
2	把这个值，作为参数（argument），传入某一次函数调用
3	把这个值，在函数调用中返回（即，作为函数调用的返回值）
这个值不会被拷贝后赋给目的变量，而会发生所有权转移	

你可能会很惊讶

为什么Rust要去改变

这些已经约定俗成的基本操作的含义?

实际情况并非如此

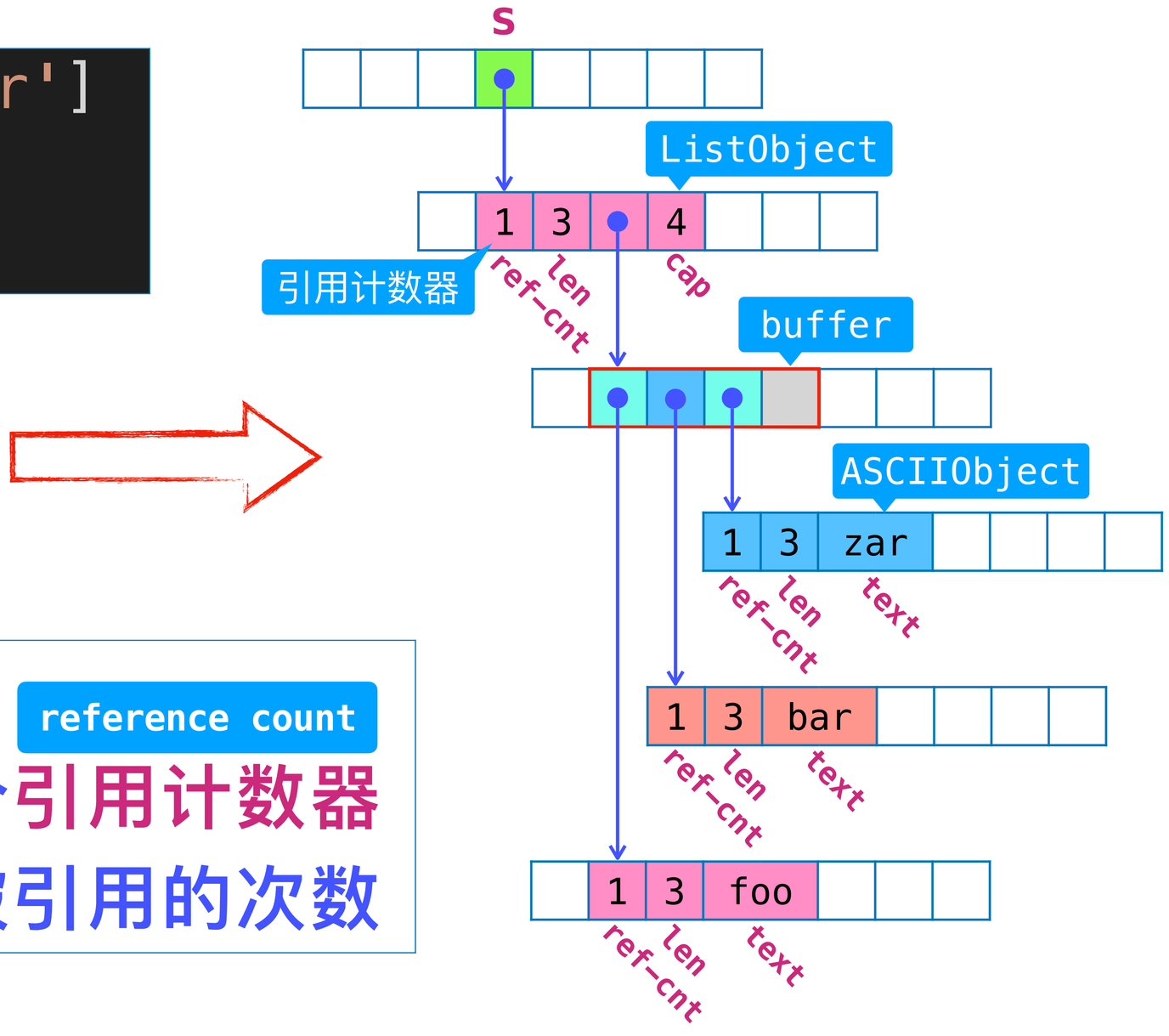
如果你去考察不同语言中这些基本操作的含义

就会发现：哪里有什么约定俗成

Python中的赋值：一个示例

```
01  s = ['foo', 'bar', 'zar']
02  t = s
03  u = s
```

当刚执行完**第01行语句**时
内存排布情况如右图所示



在Python中

reference count

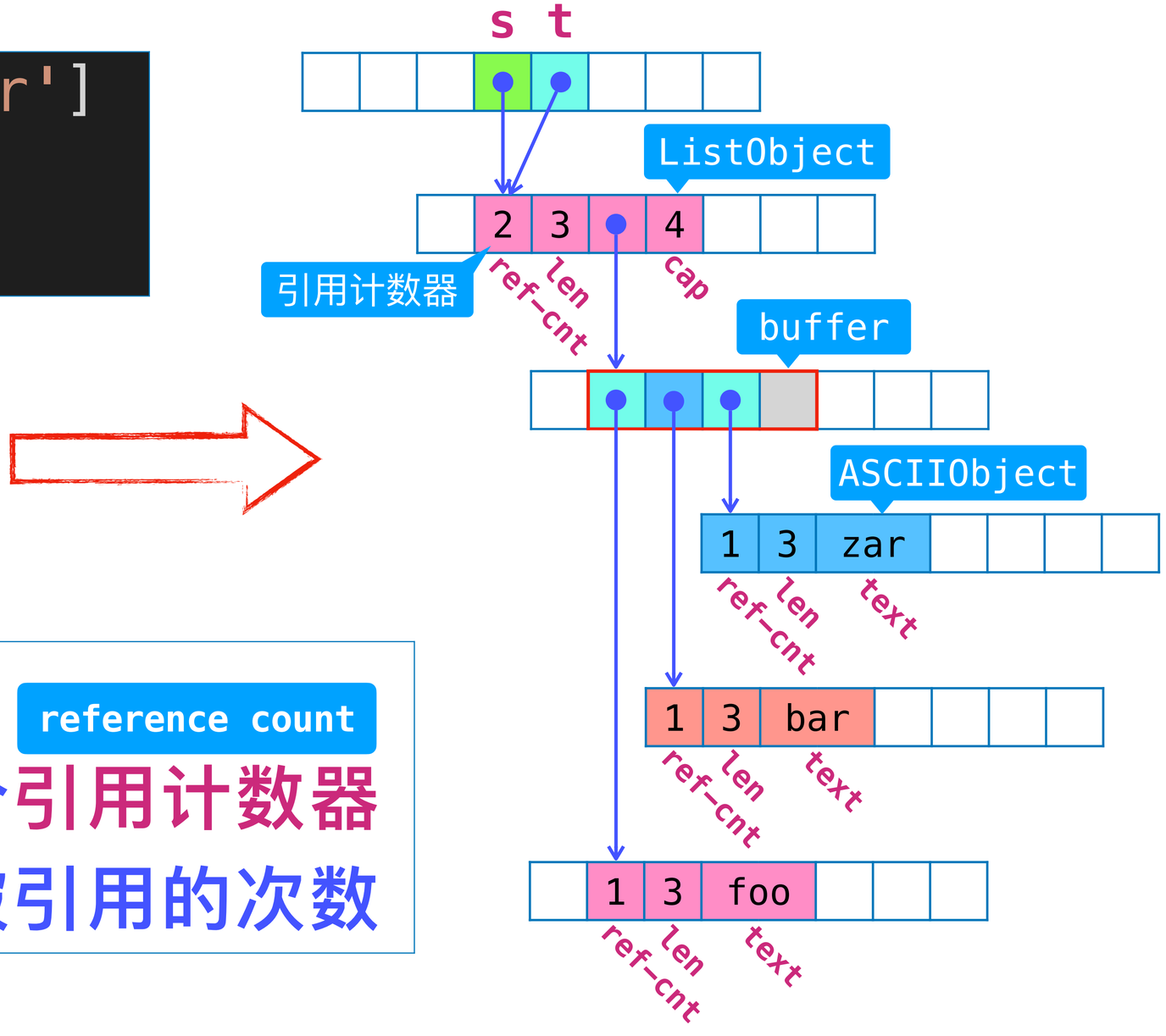
每一object都具有一个**引用计数器**

表示：该object当前被引用的次数

Python中的赋值：一个示例

```
01  s = ['foo', 'bar', 'zar']
02  t = s
03  u = s
```

当刚执行完第02行语句时
内存排布情况如右图所示



在Python中

reference count

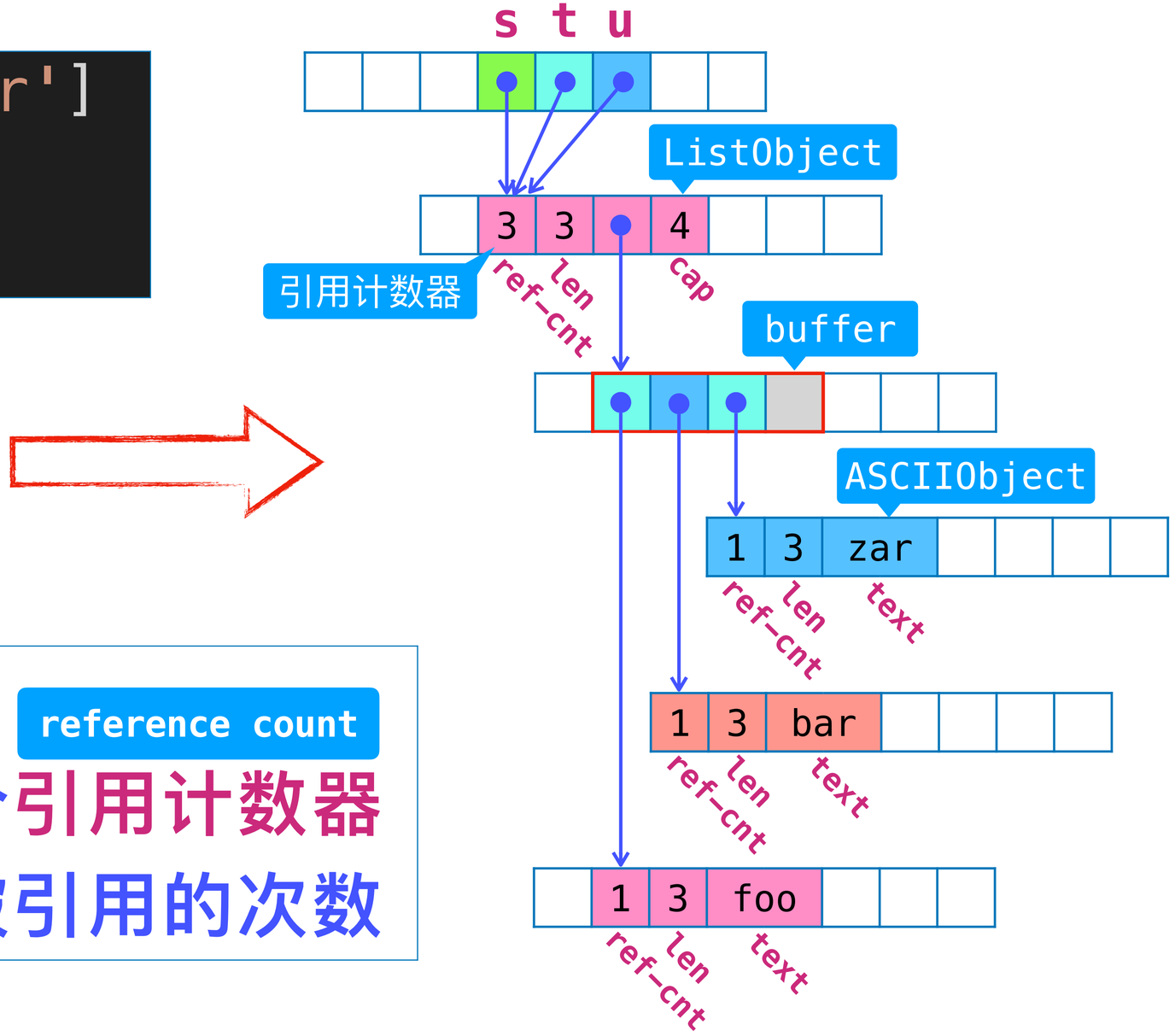
每一object都具有一个引用计数器

表示：该object当前被引用的次数

Python中的赋值：一个示例

```
01  s = ['foo', 'bar', 'zar']
02  t = s
03  u = s
```

当刚执行完第03行语句时
内存排布情况如右图所示



在Python中

reference count

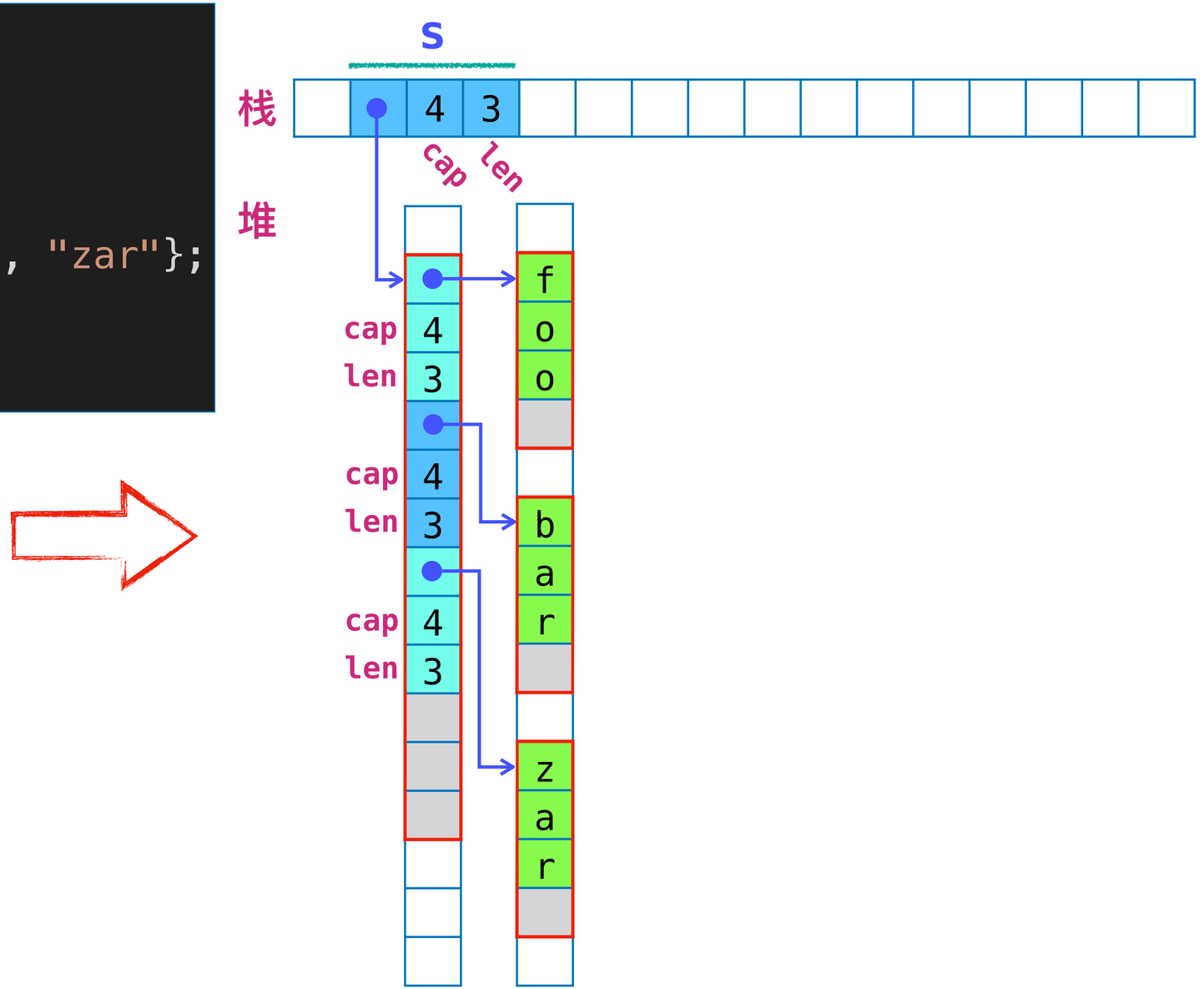
每一object都具有一个引用计数器

表示：该object当前被引用的次数

C++中的赋值：一个示例

```
01 #include <vector>
02 #include <string>
03 using namespace std;
04
05 int main(){
06     vector<string> s = {"foo", "bar", "zar"};
07     vector<string> t = s;
08     vector<string> u = s;
09 }
```

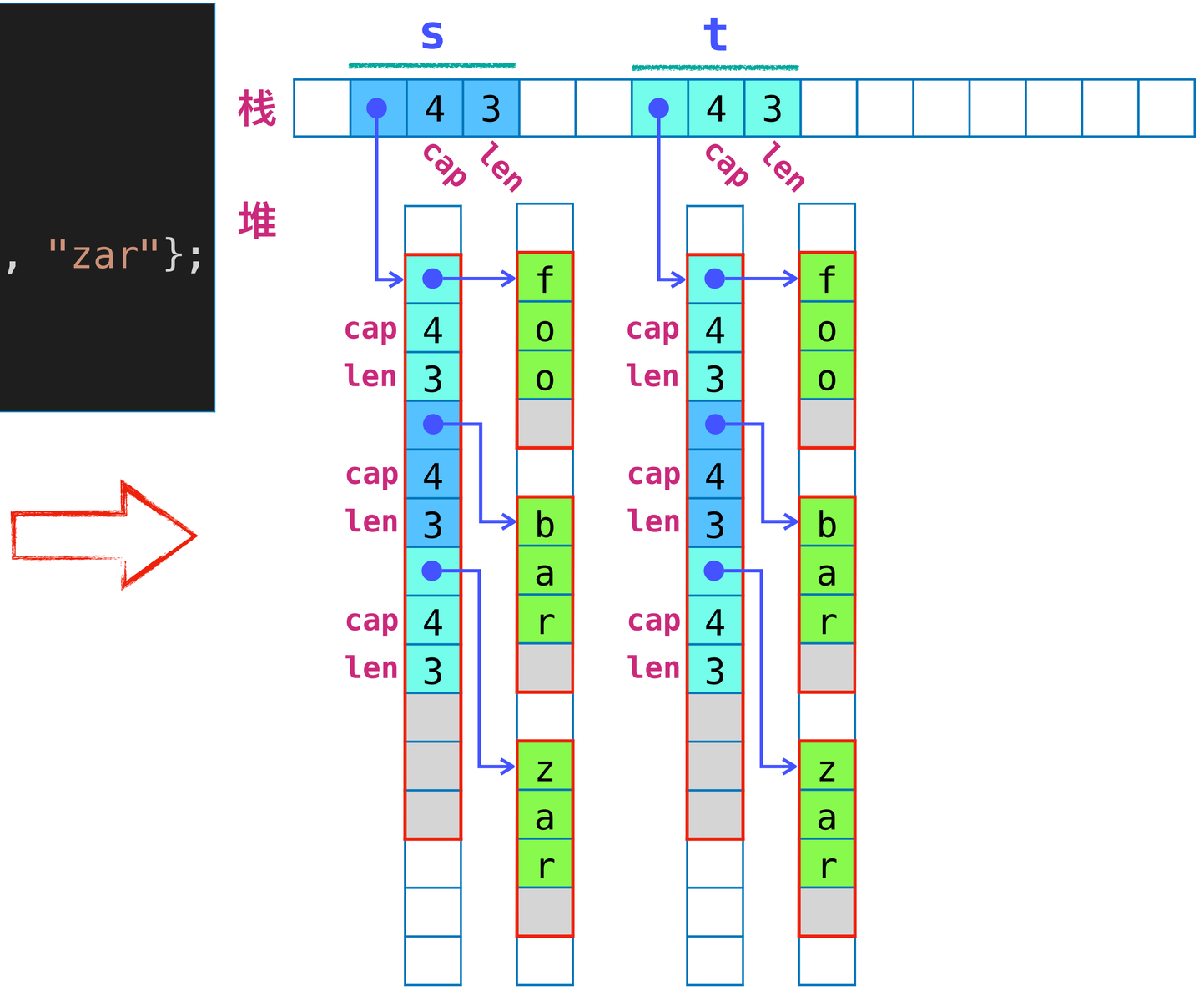
当刚执行完第06行语句时
内存排布情况如右图所示



C++中的赋值：一个示例

```
01 #include <vector>
02 #include <string>
03 using namespace std;
04
05 int main(){
06     vector<string> s = {"foo", "bar", "zar"};
07     vector<string> t = s;
08     vector<string> u = s;
09 }
```

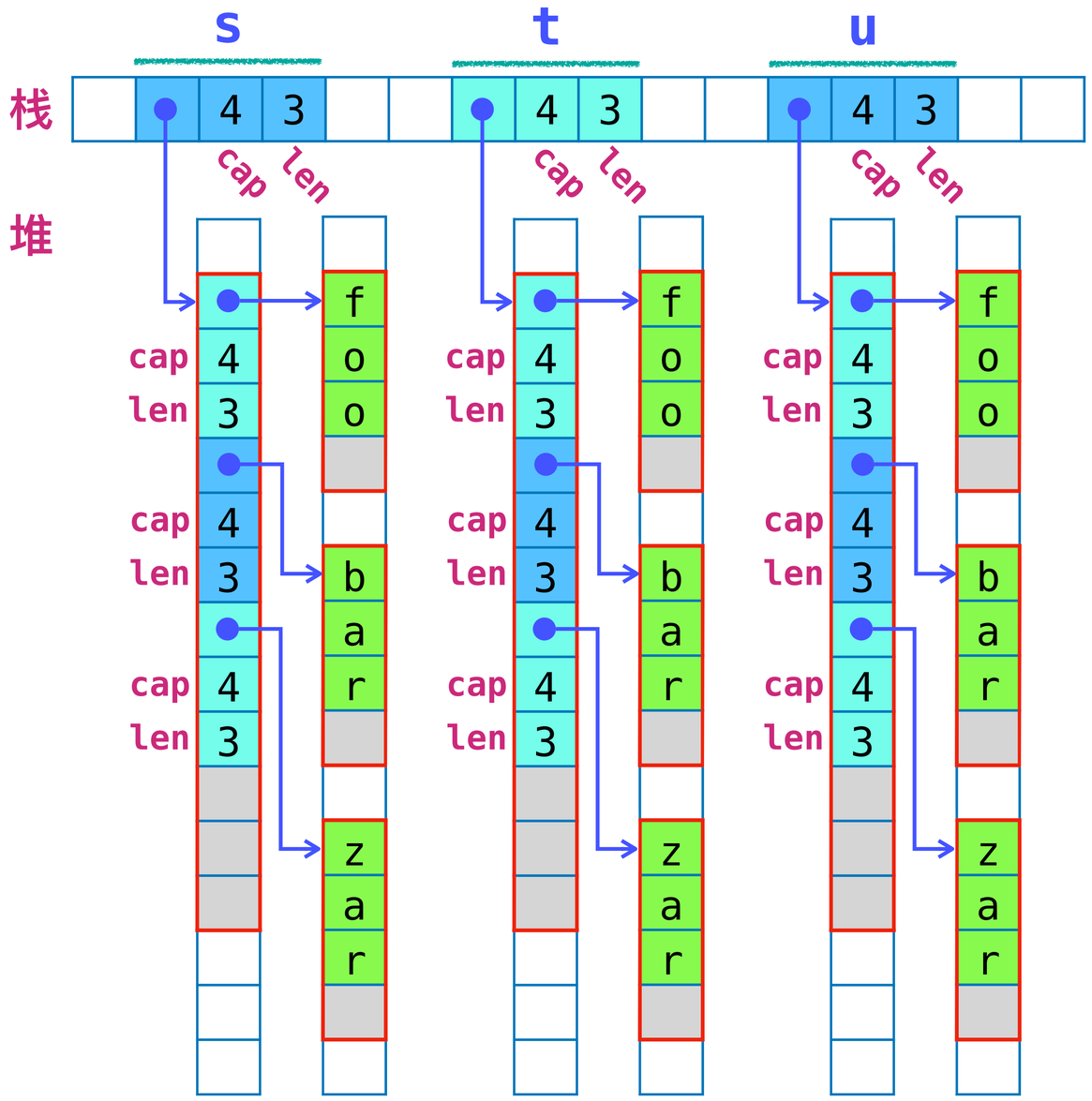
当刚执行完第07行语句时
内存排布情况如右图所示



C++中的赋值：一个示例

```
01 #include <vector>
02 #include <string>
03 using namespace std;
04
05 int main(){
06     vector<string> s = {"foo", "bar", "zar"};
07     vector<string> t = s;
08     vector<string> u = s;
09 }
```

当刚执行完第08行语句时
内存排布情况如右图所示



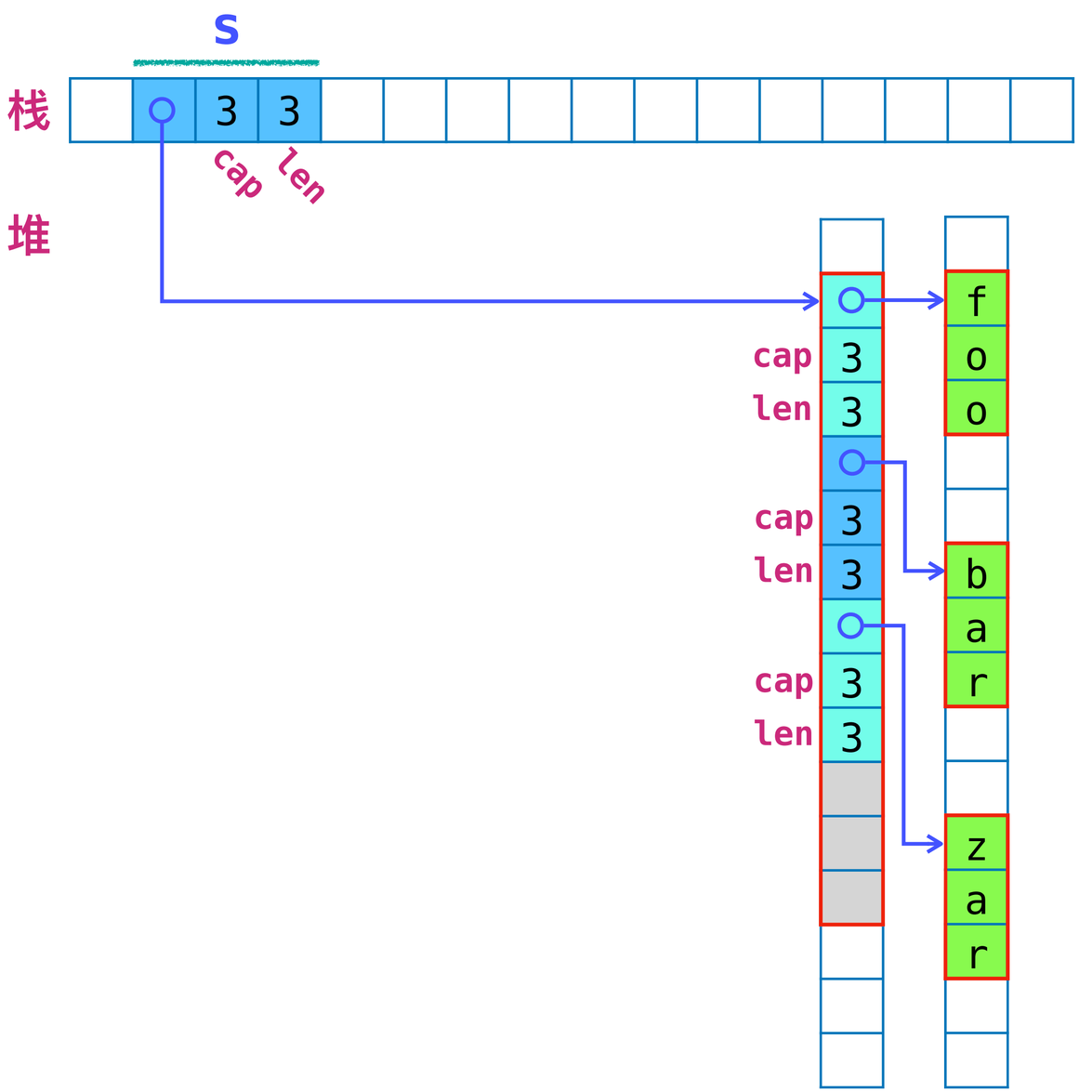
Python中的赋值 VS C++中的赋值

	Python	C++
赋值成本	低 （增加引用计数）	高 （深层复制）
内存管理成本	高 （运行时垃圾回收） （循环引用难处理）	低

Rust中的赋值：一个示例

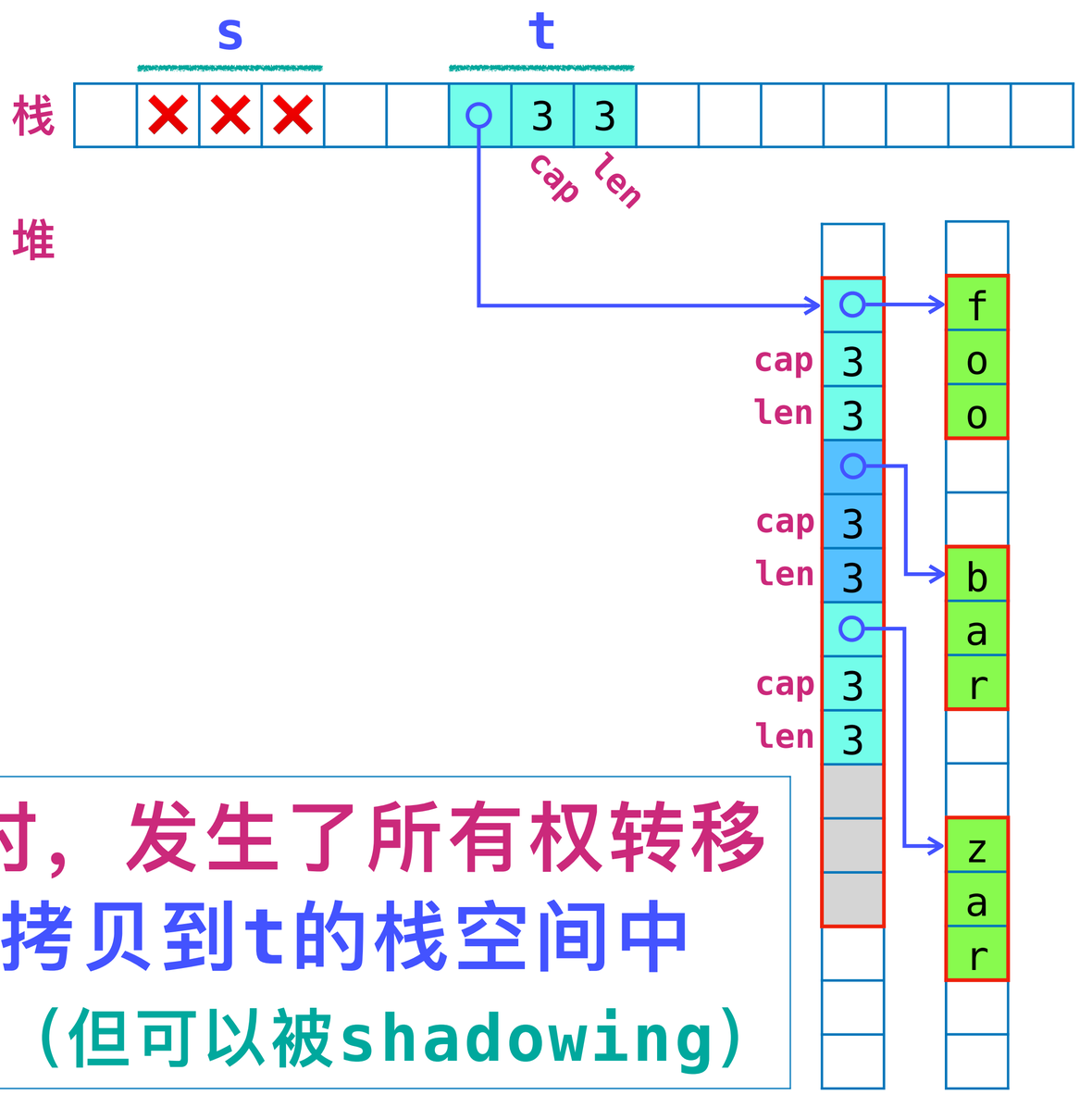
```
01 fn main() {
02     let s = vec![String::from("foo"),
03                  String::from("bar"),
04                  String::from("zar")];
05     let t = s;
06   //let u = s;
07 }
```

当刚执行完第04行语句时
内存排布情况如右图所示



Rust中的赋值：一个示例

```
01 fn main() {
02     let s = vec![String::from("foo"),
03                  String::from("bar"),
04                  String::from("zar")];
05     let t = s;
06   //let u = s;
07 }
```



当刚执行完第05行语句时
内存排布情况如右图所示

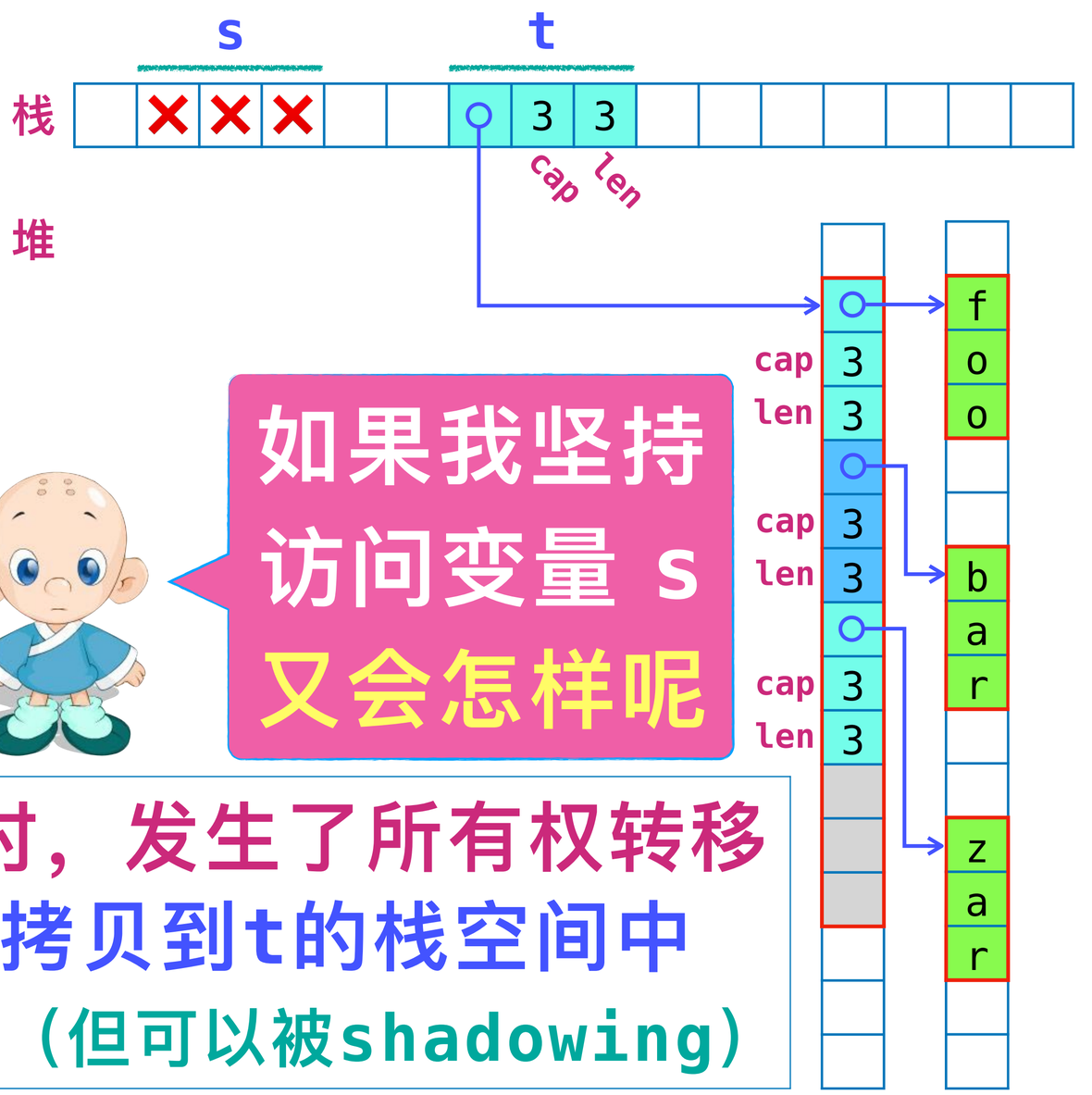
执行第05行语句时，发生了所有权转移

- s栈空间的值被拷贝到t的栈空间中
- s无法再被读取（但可以被shadowing）

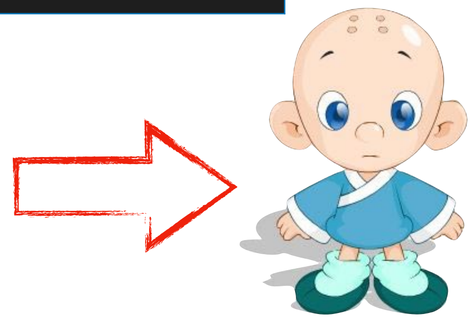
如果s被声明为mut
则可以为s重新赋值

Rust中的赋值：一个示例

```
01 fn main() {
02     let s = vec![String::from("foo"),
03                  String::from("bar"),
04                  String::from("zar")];
05     let t = s;
06   //let u = s;
07 }
```



当刚执行完第05行语句时
内存排布情况如右图所示



如果我坚持
访问变量 s
又会怎样呢

如果s被声明为mut
则可以为s重新赋值

执行第05行语句时，发生了所有权转移

- s栈空间的值被拷贝到t的栈空间中
- s无法再被读取（但可以被shadowing）

Rust中的赋值：一个示例

```
01 fn main() {
02     let s = vec![String::from("foo"),
03                  String::from("bar"),
04                  String::from("zar")];
05     let t = s;
06     let u = s;
07 }
```

如果我坚持访问变量 s 又会怎样呢

为什么要这样设计呢

```
error[E0382]: use of moved value: `s`
 --> src/main.rs:6:13
  |
2 |     let s = vec![String::from("foo"),
  |         - move occurs because `s` has type `Vec<String>`, which does not implement the `Copy` trait
...
5 |     let t = s;
  |             - value moved here
6 |     let u = s;
  |             ^ value used here after move
```

Python中的赋值 VS C++中的赋值 VS C++中的赋值

	Python	C++	Rust
赋值成本	低 （增加引用计数）	高 （深层拷贝）	低 （仅拷贝栈空间）
内存管理成本	高 （运行时垃圾回收） （循环引用难处理）	低	低

而且，可以在Rust中实现Python和C++中的赋值行为

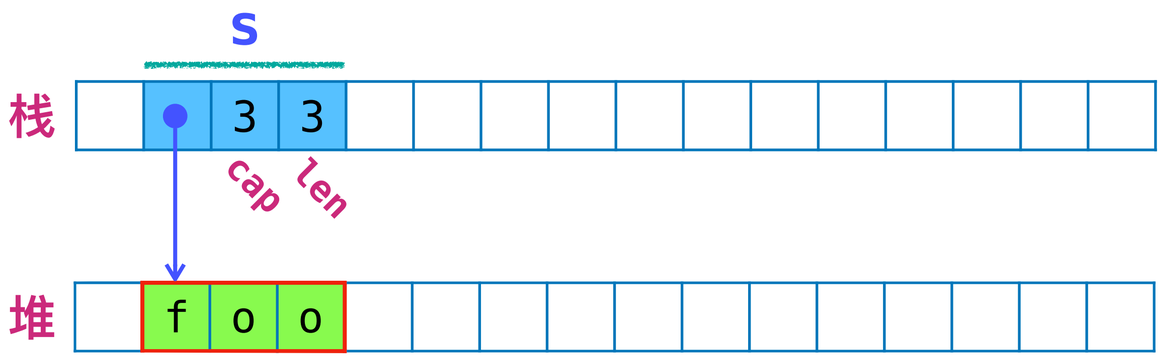
实现
C++赋值行为
	`let s = vec![ ... ];`
	`let t = s.clone();`
	`let u = s.clone();`

实现 Python赋值行为	使用Rust标准库提供的引用计数指针类型 让一个值具有多个所有者（有限制，稍后介绍）

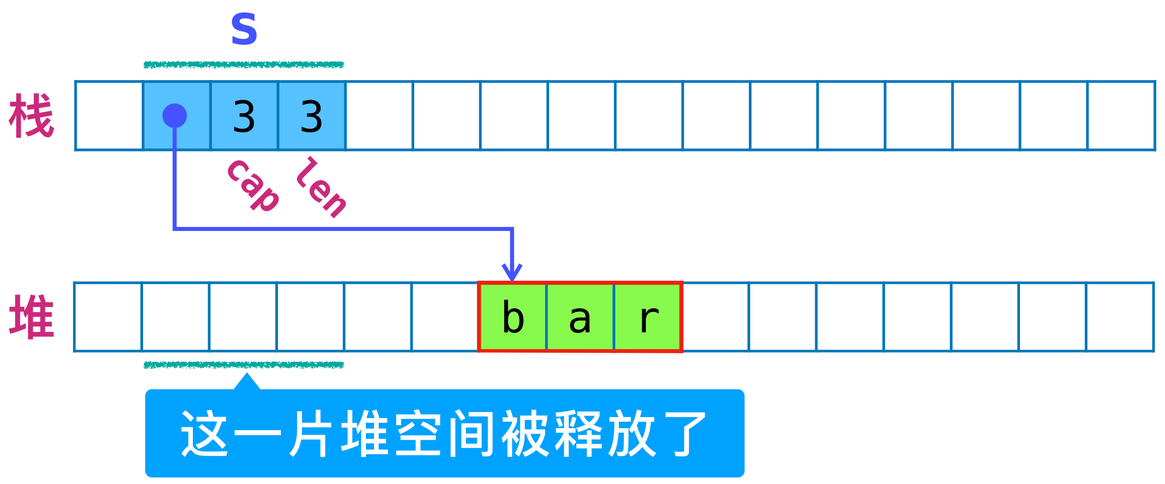
为一个已经有值的变量重新赋值

```
01 fn main() {
02     let mut s = String::from("foo");
03     s = String::from("bar");
04     println!("{}", s);
05 }
```

当刚执行完第02行语句时



当刚执行完第03行语句时



所有权转移（不适用Copy type）：发生的场景

1	变量赋值
2	变量/值作为参数传入函数调用
3	变量/值在函数调用中返回
4	构造一个元组类型的值
...	还有很多其他情况：马上就讲

所有权转移：条件语句

```rust
fn main() {
    let x = vec![10, 20, 30];
    let c = 10;

    if c < 0 {
        foo(x);
    } else {
        println!("{} ⩾ 0", c);
    }

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

fn foo(vs: Vec<i32>) {
    // 这里什么都没有
}
```

若变量有可能在条件语句的某一个分支中被剥夺所有权

即使运行时没有经过该分支

条件语句后也不能读该变量

```
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:11:22
   |
2  |     let x = vec![10, 20, 30];
   |         - move occurs because `x` has type `Vec<i32>`, which does not
implement the `Copy` trait
...
6  |         foo(x);
   |             - value moved here
...
11 |     println!("{:?}", x);
   |                      ^ value borrowed here after move
```

所有权转移：循环语句

```
01 fn main() {
02     let x = vec![10, 20, 30];
03     let mut len = x.len();
04
05     while len > 0 {
06         foo(x);
07
08         len -= 1;
09     }
10 }
11
12 fn foo(vs: Vec<i32>) {}
```

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:6:13
...
6 |   foo(x);
  |       ^ value moved here, in previous iteration of loop
```

```
01 fn main() {
02     let mut x = vec![10, 20, 30];
03     let mut len = x.len();
04
05     while len > 0 {
06         foo(x);
07         x = vec![10, 20, 30];
08         len -= 1;
09     }
10 }
11
12 fn foo(vs: Vec<i32>) {}
```

//这是一个合法的Rust程序

所有权转移：数组、向量、切片

```rust
fn main() {
    let mut v = Vec::new();
    for i in 1 .. 10 {
        v.push(i.to_string());
    }

    let third = v[2];
    println!("{}", third);
}
```

Rust不允许仅通过赋值语句把数组/向量/切片类似值中某位置上元素的所有权转移

多数情况下不必转移所有权

取得元素的引用可能就足够

这也是Rust编译器给出的建议

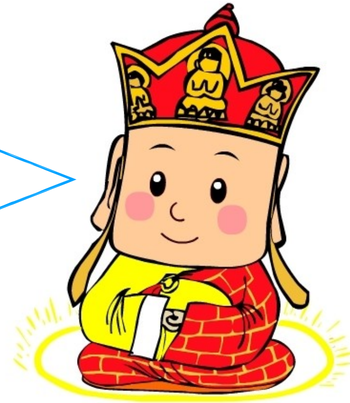
```
error[E0507]: cannot move out of index of `Vec<String>`
 --> src/main.rs:7:17
  |
7 |     let third = v[2];
  |                 ^^^^
  |                 |
  |                 move occurs because value has type `String`, which does not implement the `Copy` trait
  |                 help: consider borrowing here: `&v[2]`
```

所有权转移：数组、向量、切片

如果我就是要把数组/向量/切片中
某一个元素的所有权转移出来呢

别任性哈，有话好好说嘛
谁以前还不是一个宝宝呢

具体问题具体分析
看看是否有解决方案



从向量中转移元素的所有权：remove方法

```
01 fn main() {
02     let mut v = vec![String::from("abc"),
03                      String::from("def"),
04                      String::from("ghi"),
05                      String::from("jkl")];
06     println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]
07
08     let e = v.remove(1);
09     println!("{:?}", v); //=> ["abc", "ghi", "jkl"]
10     println!("{}", e);   //=> def
11 }
```

这不是我想要的；能否v的长度不变，但被转移的元素不可读

以你对Rust的了解，祂会提供你这样没有原则的灵活性吗
我觉得，可能周五的JavaScript课更适合你，考虑一下

从向量中转移元素的所有权：remove方法

```rust
fn main() {
    let mut v = vec![String::from("abc"),
                     String::from("def"),
                     String::from("ghi"),
                     String::from("jkl")];
    println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]

    let e = v.remove(1);
    println!("{:?}", v); //=> ["abc", "ghi", "jkl"]
    println!("{}", e);   //=> def
}
```

向量的remove方法成本较高，可能涉及大量元素的移动

若向量中元素顺序不重要，可用另一方法swap_remove

从向量中转移元素的所有权：swap_remove方法

```
01 fn main() {
02     let mut v = vec![String::from("abc"),
03                      String::from("def"),
04                      String::from("ghi"),
05                      String::from("jkl")];
06     println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]
07
08     let e = v.swap_remove(1);
09     println!("{:?}", v); //=> ["abc", "jkl", "ghi"]
10     println!("{}", e);   //=> def
11 }
```

swap_remove	把向量中特定位置的元素删除 把向量末尾元素移动到该位置

从向量中转移元素的所有权：pop方法

```
01 fn main() {
02     let mut v = vec![String::from("abc"),
03                      String::from("def"),
04                      String::from("ghi"),
05                      String::from("jkl")];
06     println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]
07
08     let e = v.pop().expect("空向量，pop个空气啊！");
09     println!("{:?}", v); //=> ["abc", "def", "ghi"]
10     println!("{}", e);   //=> jkl
11 }
```

pop方法只能弹出向量的末尾元素

从向量/数组/切片中转移元素的所有权：replace函数

```
01 fn main() {
02     let mut v = vec![String::from("abc"),
03   //let mut v =      [String::from("abc"),
04                       String::from("def"),
05                       String::from("ghi"),
06                       String::from("jkl")];
07
08   //let v = &mut v;
09
10     println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]
11
12     let e = std::mem::replace(&mut v[1], String::from("dog"));
13     println!("{:?}", v); //=> ["abc", "dog", "ghi", "jkl"]
14     println!("{}", e);   //=> def
15 }
```

std::mem::replace函数

```
pub fn replace<T>(dest: &mut T, src: T) -> T
```

[–] Moves `src` into the referenced `dest`, returning the previous `dest` value.

Neither value is dropped.

- If you want to replace the values of two variables, see `swap`.
- If you want to replace with a default value, see `take`.

Examples

A simple example:

Run

```
use std::mem;

let mut v: Vec<i32> = vec![1, 2];

let old_v = mem::replace(&mut v, vec![3, 4, 5]);
assert_eq!(vec![1, 2], old_v);
assert_eq!(vec![3, 4, 5], v);
```

Function std::mem::swap 1.0.

```
pub fn swap<T>(x: &mut T, y: &mut T)
```

[–] Swaps the values at two mutable locations, witho

- If you want to swap with a default or dummy
- If you want to swap with a passed value, retu

Examples

```
use std::mem;

let mut x = 5;
let mut y = 42;

mem::swap(&mut x, &mut y);

assert_eq!(42, x);
assert_eq!(5, y);
```

std::mem::take函数

```
pub fn take<T>(dest: &mut T) -> T
where
    T: Default,
```

限制：T必须是一种具有缺省值的类型

[–] Replaces `dest` with the default value of `T`, returning the previous `dest` value.

```
01 fn main() {
02     let mut v = vec![String::from("abc"),
03                      String::from("def"),
04                      String::from("ghi"),
05                      String::from("jkl")];
06
07     println!("{:?}", v); //=> ["abc", "def", "ghi", "jkl"]
08
09     let e = std::mem::take(&mut v[1]);
10     println!("{:?}", v); //=> ["abc", "", "ghi", "jkl"]
11     println!("{}", e);   //=> def
12 }
```

更显式地标记一个元素上是否有值

```
01 fn main() {
02     let mut v = vec![Some(String::from("abc")),
03                      Some(String::from("def")),
04                      Some(String::from("ghi")),
05                      Some(String::from("jkl"))];
06
07     println!("{:?}", v); //=> [Some("abc"), Some("def"), Some("ghi"), Some("jkl")]
08
09     let e1 = std::mem::take(&mut v[1]);
10     println!("{:?}", v);  //=> [Some("abc"), None, Some("ghi"), Some("jkl")]
11     println!("{:?}", e1); //=> Some("def")
12
13     let e2 = std::mem::take(&mut v[2]);
14     println!("{:?}", v);  //=> [Some("abc"), None, None, Some("jkl")]
15     println!("{:?}", e2); //=> Some("ghi")
16 }
```

把向量/数组的所有权转移给循环语句

```
01 fn main() {
02     let v = vec![String::from("abc"),
03   //let v =      [String::from("abc"),
04                   String::from("def"),
05                   String::from("ghi")];
06
07     for mut s in v {
08         s.push(';');
09         println!("{}", s); //=> abc;
10     }                      //=> def;
11                            //=> ghi;
12     // 这里不能再读取v了; 其实, 在循环体内就已经无法读取v了
13 }
```

v的所有权被转移给循环语句

Copy Types：所有权的法外之地

在Rust中，下面的类型都是Copy Types	
1	语言自带的所有数字类型：整数类型、浮点数类型
2	char、bool
3	若干其他类型
4	元素类型为Copy Type的数组
5	所有元素类型均为Copy Type的元组

对于其他的用户自定义数据类型（目前只简单介绍了struct）

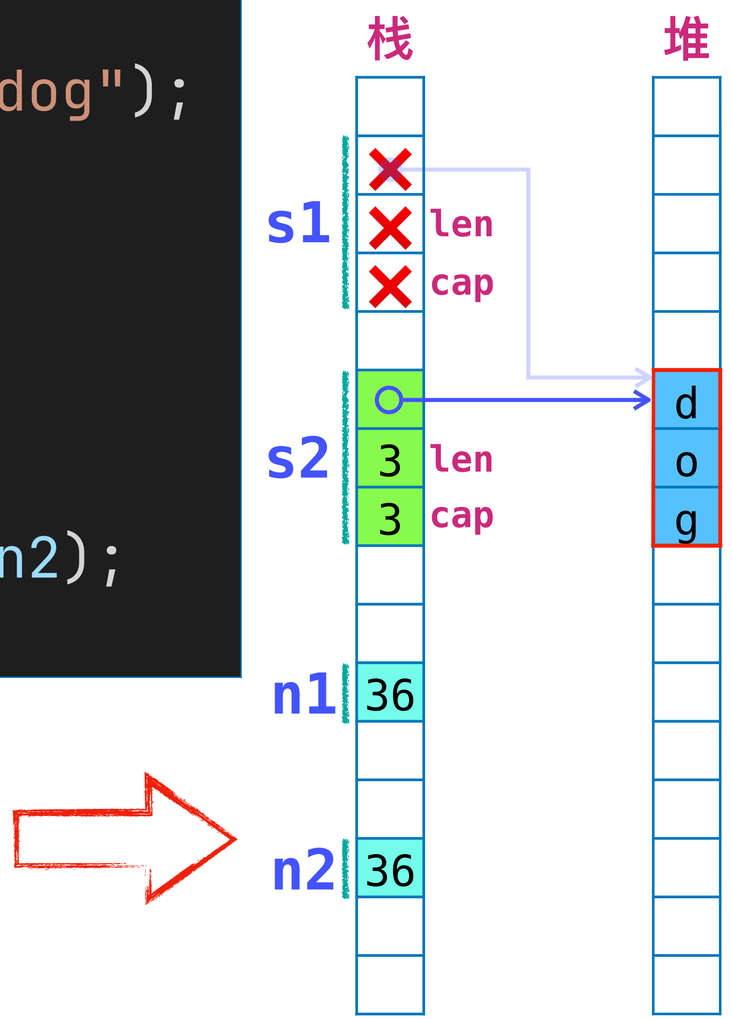
缺省情况下，都不是Copy Type

但是，在满足特定条件时，可以将其声明为Copy Type

Copy Types的行为：以赋值操作为例

```
01 fn main() {
02     let s1 = String::from("dog");
03     let s2 = s1;
04
05     let n1: i32 = 36;
06     let n2 = n1;
07
08     println!("{}, {}", s2, n2);
09 }
```

当刚执行完第06行语句时
内存排布情况如右图所示



String不是Copy Type
let s2 = s1; 之后
s1值的所有权转移给s2
s1不可再被读取

i32 是 Copy Type
let n2 = n1; 之后
n1的值被复制给n2
n1可继续被读取

以上即是
copy/no-copy type
的区别

Copy Types 与 自定义数据类型（以struct为例）

缺省情况下，一个struct类型不是 copy type

```
01 fn main() {
02
03     struct Label { number: u32 }
04     fn print(l: Label) { println!("{}", l.number); }
05
06     let l = Label { number: 3 };
07     print(l);
08     println!("{}", l.number);
09 }
```

```
error[E0382]: borrow of moved value: `l`
 --> src/main.rs:8:20
  |
6 |     let l = Label { number: 3 };
  |         - move occurs because `l` has type `Label`, which does not implement the `Copy` trait
7 |     print(l);
  |           - value moved here
8 |     println!("{}", l.number);
  |                    ^^^^^^^^ value borrowed here after move
```

Copy Types 与 自定义数据类型（以struct为例）

但如果struct类型包含的所有分量的类型都是copy type
那么，　可以通过attribute将该类型声明为copy type

Copy Clone的区别：后面会讲

```
01 fn main() {
02     #[derive(Copy, Clone)]
03     struct Label { number: u32 }
04     fn print(l: Label) { println!("{}", l.number); }
05
06     let l = Label { number: 3 };
07     print(l); //=> 3
08     println!("{}", l.number); //=> 3
09 }
```

Copy Types 与 自定义数据类型（以struct为例）

~~但如果struct类型包含的所有分量的类型都是copy type~~
那么，可以通过attribute将该类型声明为copy type

```
01 fn main() {
02     #[derive(Copy, Clone)]
03     struct Label { number: u32, name: String }
04     fn print(l: Label) { println!("{}", l.number); }
05
06     let l = Label { number: 3, name: String::from("dog") };
07     print(l);
08     println!("{}", l.number);
09 }
```

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> src/main.rs:2:14
  |
2 |     #[derive(Copy, Clone)]
  |              ^^^^
3 |     struct Label { number: u32, name: String }
  |                                 ------------ this field does not implement `Copy`
```

Copy Types 与 自定义数据类型（以struct为例）

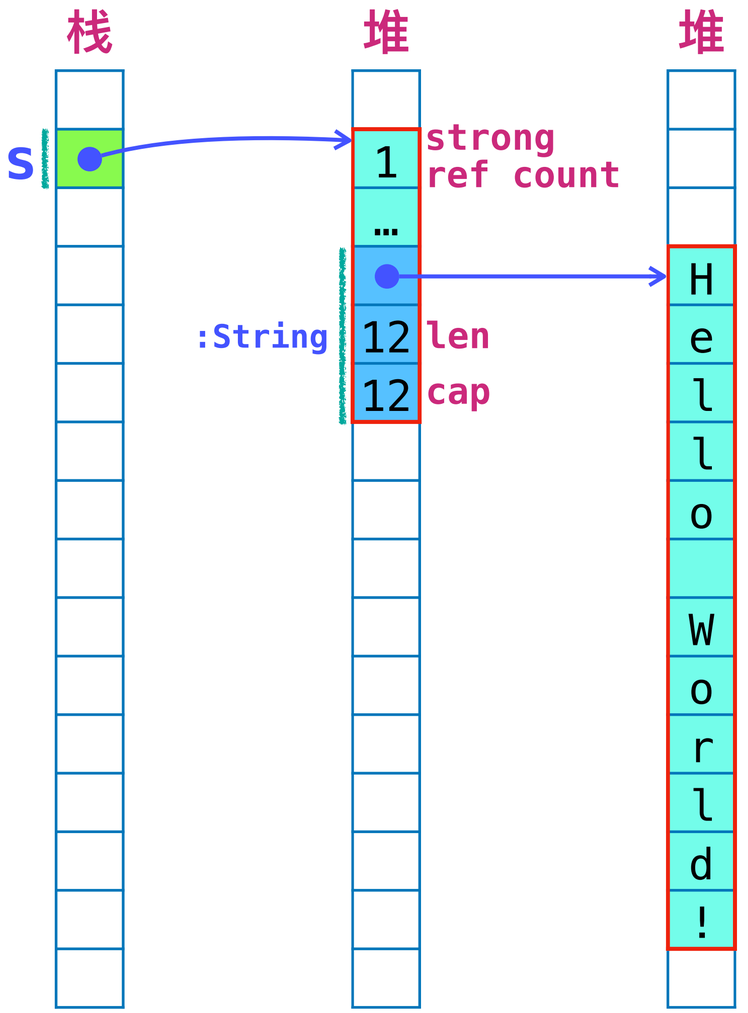
其他自定义类型

能否/如何声明为copy type的条件/方式

与struct类型类似

共享所有权：引用计数指针类型 Rc Arc

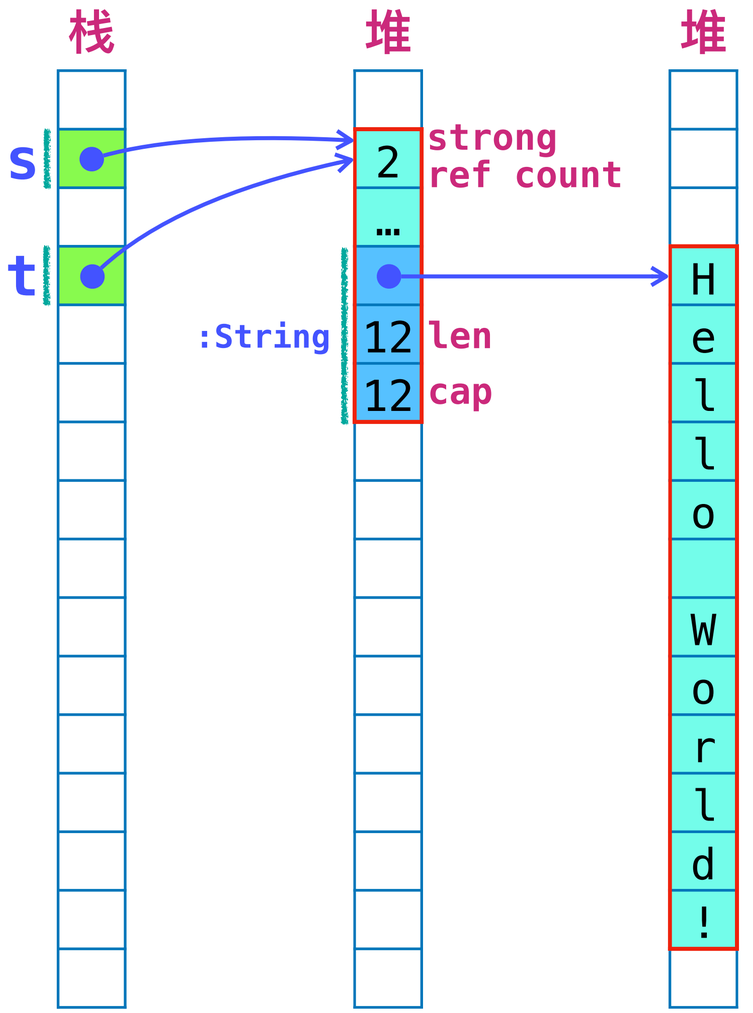
```
01 use std::rc::Rc;
02
03 fn main() {
04     // 以下语句可以不必写类型声明；编译器会进行类型推断
05     let s: Rc<String> = Rc::new(String::from("Hello World!"));
06     let t: Rc<String> = s.clone(); // Method-call syntax
07     let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax
08
09     println!("{}", s); //=> Hello World!
10     println!("{}", t); //=> Hello World!
11     println!("{}", u); //=> Hello World!
12
13     println!("{}", s.contains("ello")); //=> true
14     println!("{:?}", t.find("World")); //=> Some(6)
15
16     /*
17      * 可以在一个 Rc<T> 类型的值上
18      * 直接调用 T 类型的值上的方法。
19      */
20 }
```



当刚执行完第05行语句时
内存排布情况如右图所示

共享所有权：引用计数指针类型 Rc Arc

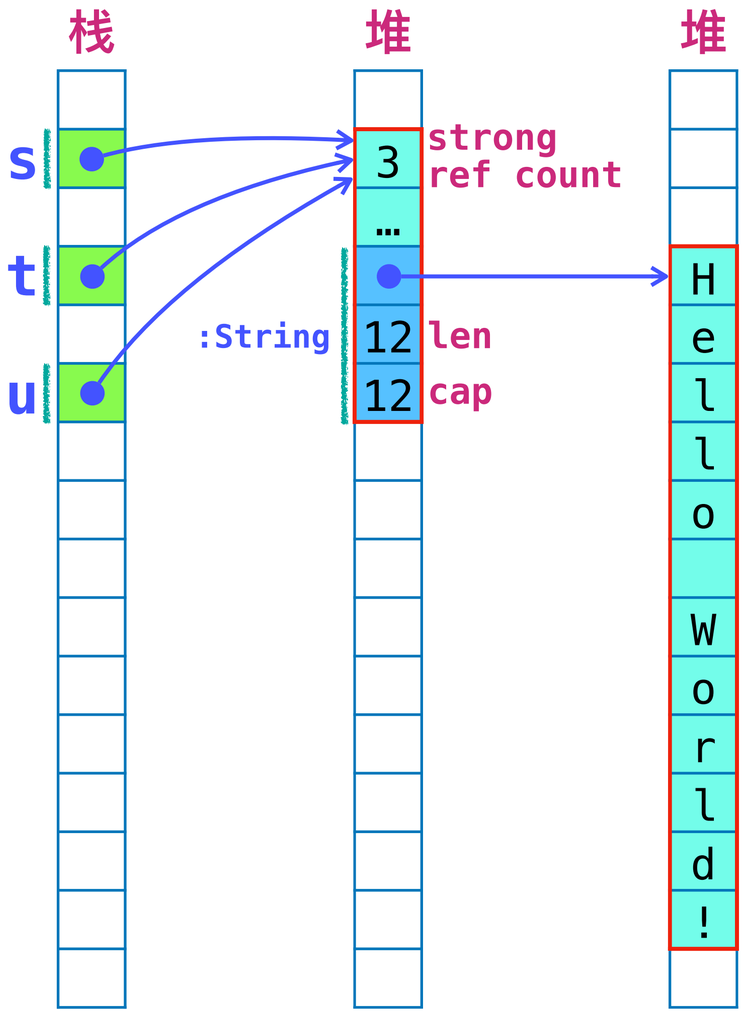
```
01 use std::rc::Rc;
02
03 fn main() {
04     // 以下语句可以不必写类型声明；编译器会进行类型推断
05     let s: Rc<String> = Rc::new(String::from("Hello World!"));
06     let t: Rc<String> = s.clone(); // Method-call syntax
07     let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax
08
09     println!("{}", s); //=> Hello World!
10     println!("{}", t); //=> Hello World!
11     println!("{}", u); //=> Hello World!
12
13     println!("{}", s.contains("ello")); //=> true
14     println!("{:?}", t.find("World")); //=> Some(6)
15
16     /*
17      * 可以在一个 Rc<T> 类型的值上
18      * 直接调用 T 类型的值上的方法。
19      */
20 }
```



当刚执行完第06行语句时
内存排布情况如右图所示

共享所有权：引用计数指针类型 Rc Arc

```
01 use std::rc::Rc;
02
03 fn main() {
04     // 以下语句可以不必写类型声明；编译器会进行类型推断
05     let s: Rc<String> = Rc::new(String::from("Hello World!"));
06     let t: Rc<String> = s.clone(); // Method-call syntax
07     let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax
08
09     println!("{}", s); //=> Hello World!
10     println!("{}", t); //=> Hello World!
11     println!("{}", u); //=> Hello World!
12
13     println!("{}", s.contains("ello")); //=> true
14     println!("{:?}", t.find("World")); //=> Some(6)
15
16     /*
17      * 可以在一个 Rc<T> 类型的值上
18      * 直接调用 T 类型的值上的方法。
19      */
20 }
```

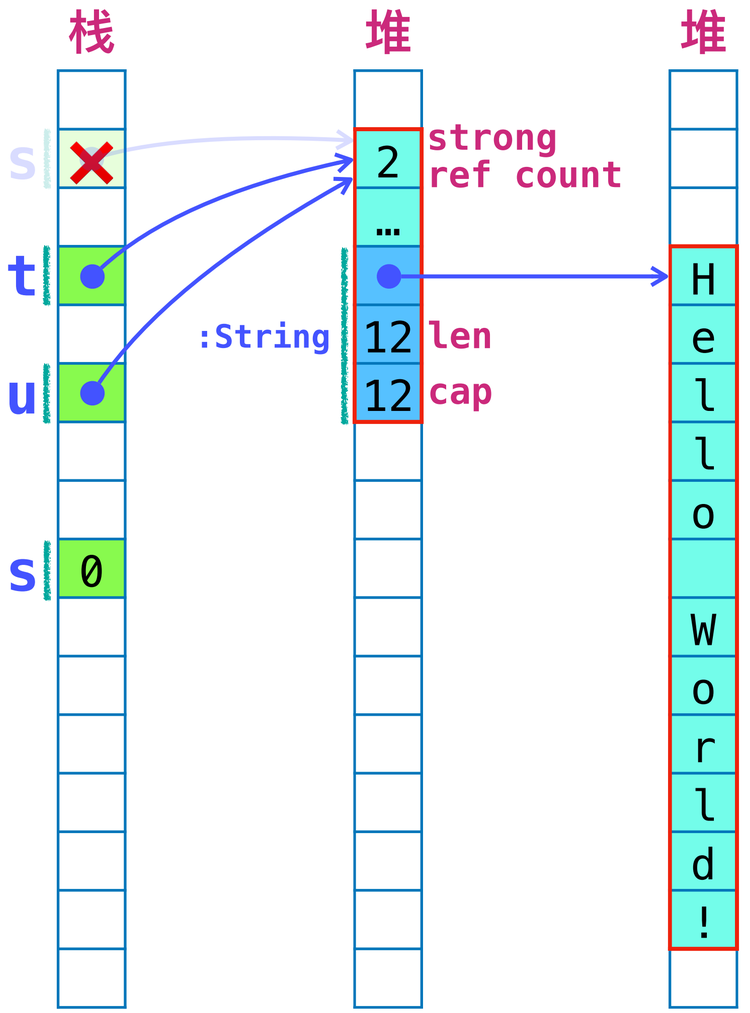


当刚执行完第07行语句时
内存排布情况如右图所示

共享所有权：引用计数指针类型 Rc Arc

```rust
use std::rc::Rc;

fn main() {
    // 以下语句可以不必写类型声明；编译器会进行类型推断
    let s: Rc<String> = Rc::new(String::from("Hello World!"));
    let t: Rc<String> = s.clone(); // Method-call syntax
    let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax

    println!("{}", s); //=> Hello World!
    println!("{}", t); //=> Hello World!
    println!("{}", u); //=> Hello World!

    println!("{}", s.contains("ello")); //=> true
    println!("{:?}", t.find("World")); //=> Some(6)

    let s = 0;
    let t = 1;
    let u = 2;

}
```

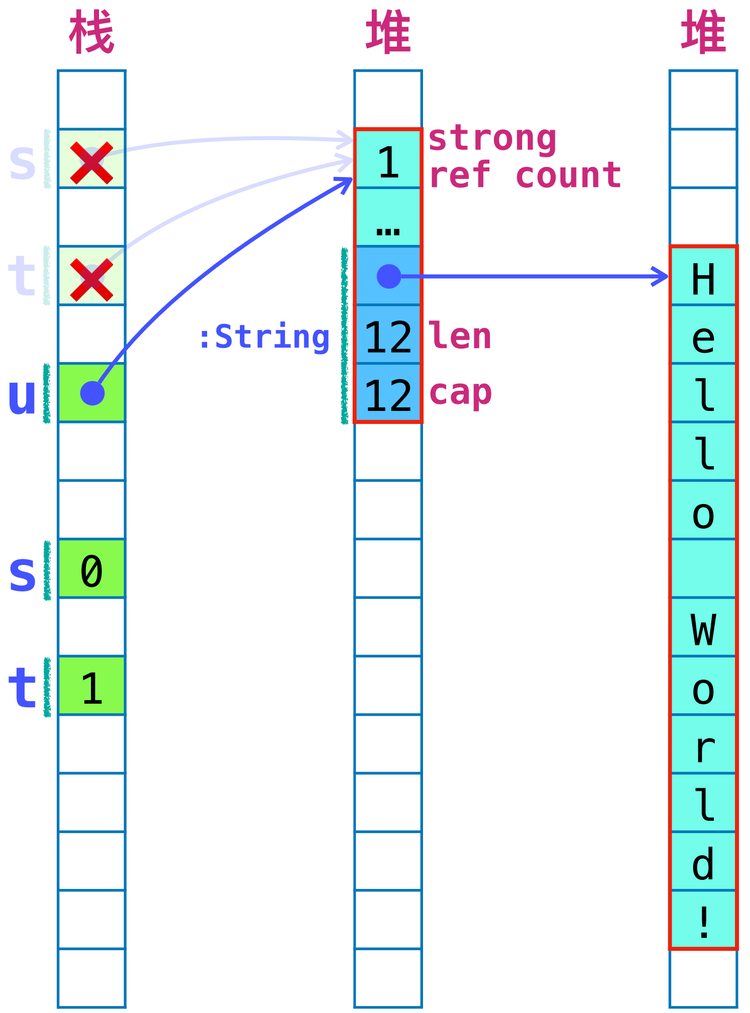
当刚执行完第16行语句时
内存排布情况如右图所示



共享所有权：引用计数指针类型 Rc Arc

```rust
use std::rc::Rc;

fn main() {
    // 以下语句可以不必写类型声明；编译器会进行类型推断
    let s: Rc<String> = Rc::new(String::from("Hello World!"));
    let t: Rc<String> = s.clone(); // Method-call syntax
    let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax

    println!("{}", s); //=> Hello World!
    println!("{}", t); //=> Hello World!
    println!("{}", u); //=> Hello World!

    println!("{}", s.contains("ello")); //=> true
    println!("{:?}", t.find("World")); //=> Some(6)

    let s = 0;
    let t = 1;
    let u = 2;

}
```

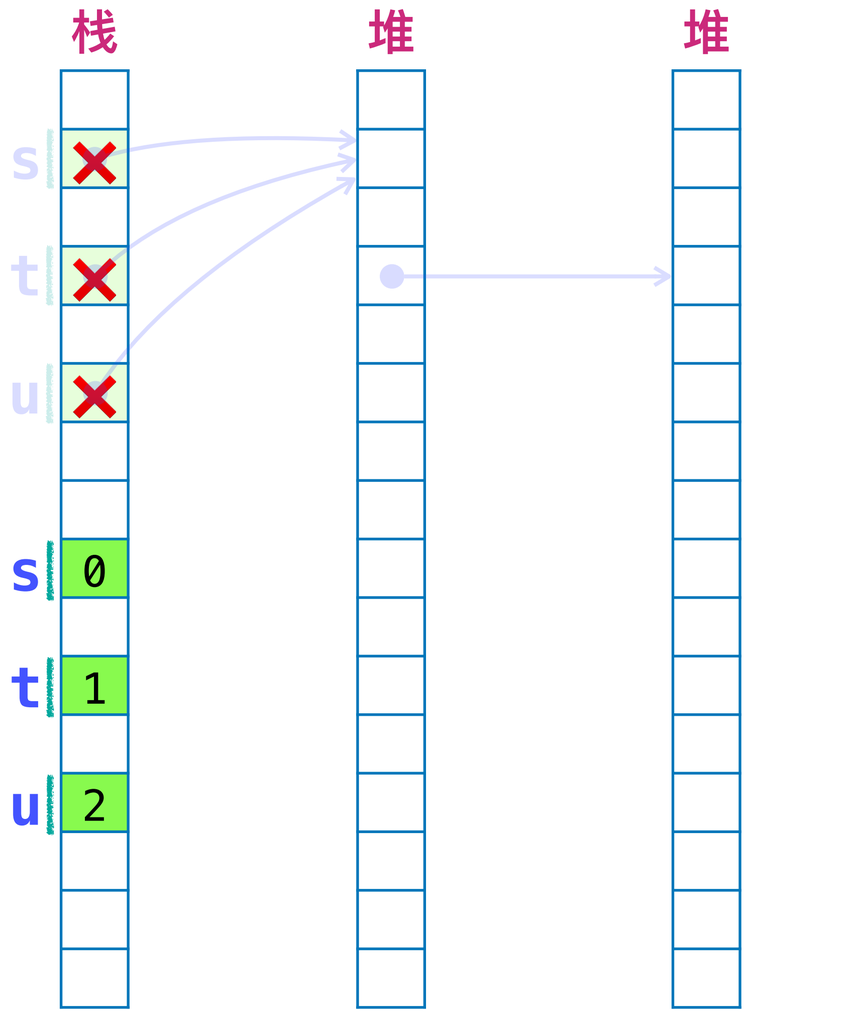
当刚执行完第17行语句时
内存排布情况如右图所示



共享所有权：引用计数指针类型 Rc Arc

```
01 use std::rc::Rc;
02
03 fn main() {
04     // 以下语句可以不必写类型声明；编译器会进行类型推断
05     let s: Rc<String> = Rc::new(String::from("Hello World!"));
06     let t: Rc<String> = s.clone(); // Method-call syntax
07     let u: Rc<String> = Rc::clone(&s); //Fully qualified syntax
08
09     println!("{}", s); //=> Hello World!
10     println!("{}", t); //=> Hello World!
11     println!("{}", u); //=> Hello World!
12
13     println!("{}", s.contains("ello")); //=> true
14     println!("{:?}", t.find("World")); //=> Some(6)
15
16     let s = 0;
17     let t = 1;
18     let u = 2;
19
20 }
```

当刚执行完第18行语句时
内存排布情况如右图所示



共享字符串在堆中占用的空间被全部释放
因为引用计数器的值变为零了

共享所有权：引用计数指针类型 Rc Arc

但是，事情还没有完

```rust
use std::rc::Rc;

fn main() {
    let s = Rc::new(String::from("Hello World!"));

    s.push_str(" and dog.");

    println!("{}", s);
}
```

被Rc拥有的值，不可修改

```
error[E0596]: cannot borrow data in an `Rc` as mutable
 --> src/main.rs:7:5
  |
7 |     s.push_str(" and dog.");
  |     ^^^^^^^^^^^^^^^^^^^^^^^ cannot borrow as mutable
  |
  = help: trait `DerefMut` is required to modify through a dereference, but it is not implemented for `Rc<String>`
```

共享所有权：引用计数指针类型 Rc Arc

Rc 与 Arc 的区别	
Rc	non-thread-safe，但是速度快
Arc	thread-safe，但是速度慢

使用建议

始终用Rc，除非编译器告诉你用Arc

（当在多线程环境下使用了Rc，会被编译器检查出来）

Rust真是太麻烦了
这样不让做，那也不让做，婆婆妈妈的！

你生气了，你不是真的生气了吧
你生气你就说嘛，你不说我怎么知道你生气了呢
&&**$().>>**5%@@34$%*9……

shut up! 😠😠😠

自由 C/C++

Rust中的所有权：一些扩展/软化措施

1	所有权转移：一个值的所有权可以被转移给新的所有者
2	简单变量豁免：对于具有简单类型的变量（整数、浮点数、字符等），所有权规则不适用。称这些类型为Copy types
3	引用计数指针类型：Rust标准库提供了两种引用计数指针类型（ref-counted pointer types），允许一个值具有多个所有者（但需要满足一定的限制）
4	borrow a ref to a value：在不改变所有权的情况下，通过引用（ref），在满足一定限制的情况下，访问一个值

第 3 章：所有权与所有权转移

Ownership and Moves

THE END

2024.03.18：讲到这里