Rust ownership

1. 内存自动释放机制：在作用域结束的时候，自动释放

```
fn main() {
    let a = String::from("hello");
    foo(a);
    let b = a; // error: 在将a传入foo之后，在foo结束的时候内存被释放
}

fn foo(s: String) {
    // todo
}

```

2. 转移（move）：相当于两个变量共享同一个内存（浅复制），转移之后原变量不能继续用

```
fn main() {
    let a = String::from("hello");
    let b = a;
    println!("{}", a); // error, a被转移
}

fn main() {
    let a = String::from("hello");
    let b = a.clone(); // deep clone
    println!("{}", a); // a依然可用，用clone方法深复制
}

fn main() {
    let a = 1;
    let b = a;
    println!("{}", a); // a 是栈空间里的变量，不会被转移
}
```

不会发生转移的类型（发生copy）

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain types that are also `Copy`. For example, `(i32, i32)` is `Copy`, but `(i32, String)` is not.

3. 不可变借用

```
// example 1
fn main() {
    let a = String::from("hello");
    foo(&a);
    let b = a; // 借用之后可以进行转移
}

fn foo(s: &String) {
    // todo
}

// example 2
fn main() {
    // 允许有多个借用者
    let a = String::from("hello");
    foo(&a); // success
    koo(&a); // success
}
fn foo(s: &String) {}
fn koo(s: &String) {}

// example 3
fn main() {
    let a = String::from("hello");
    foo(&a);
}
```

```
fn foo(s: &String) {
    s.push_str(" world"); // error: immutable borrow
}
```

4. 可变借用

```
fn main() {
    let mut a = String::from("abc");
    foo(&mut a); // 相当于c++的引用，a的值已经被改变
    println!("{}", a);
}

fn foo(s: &mut String) {
    s.push_str("def");
}

fn main() {
    let mut a = String::from("hello");
    foo(&mut a);
    let b = a; // 可变借用在借用之后可以转移
    println!("{}", b);
}

fn foo(s: &mut String) {
    s.push_str(" world");
}

fn main() {
    let mut a = String::from("hello");
    foo(&mut a); // 多个借用者
    koo(&mut a);
}

fn foo(s: &mut String) {
    println!("{}", s);
}
```

```rust

fn koo(s: &mut String) {
    println!("{}", s);
}

fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s; // error: 在同一个scope里面，最多允许存在一个可变借用者
    let r2 = &mut s;

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

let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem，不可变借用可以有多个，但是在存在不可变借用的时候，不允许在同一个scope有可变借用
let r3 = &mut s; // BIG PROBLEM
println!("{}, {}, and {}", r1, r2, r3);

// 例外的情况
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// r1 and r2 are no longer used after this point
let r3 = &mut s; // no problem
println!("{}", r3); // r1和r2在第一个println之后没有再使用，这时候r3是可以安全的改变数据的，因此没有问题
```