TinTin

Office Hour 2

老师： Mike Tang
助教： Tulem

TinTin

Tulem.eth

Rust

所有权

01 —— 02

Understanding Rust Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector, so it's important to understand how ownership works.

I 课程关键内容回顾

TinTin

本节课程主要讲解所有权、引用（可变 不可变）

- 1、什么是内存安全?
- 2、什么是所有权?
- 3、栈内存和堆内存分别是什么?
- 4、所有权规则?
- 5、为什么要引入String类型?
- 6、 内存如何对String类型进行分配?
- 7、 变量和数据的交互方式有哪些?
- 8、 所有权与函数有哪些行为?
- 9、 引用从何而来?
- 10、可变引用与不可引用如何区分?

TinTin

一、所有权的提出

- 什么是“内存安全”?
- 很多语言的一个弊端就是默许我们写出“内存不安全”的代码，这也就会更容易产生bug。比如像下面这两个我们熟知的例子：
- 悬垂指针(Dangling pointers): 指向无效数据的指针（当我们了解数据在内存中如何存储之后，这个就很有意义）。
- 重复释放(Double frees): 试图对同一块内存地址释放两次，这会导致“未定义行为”。

TinTin

二、什么是所有权

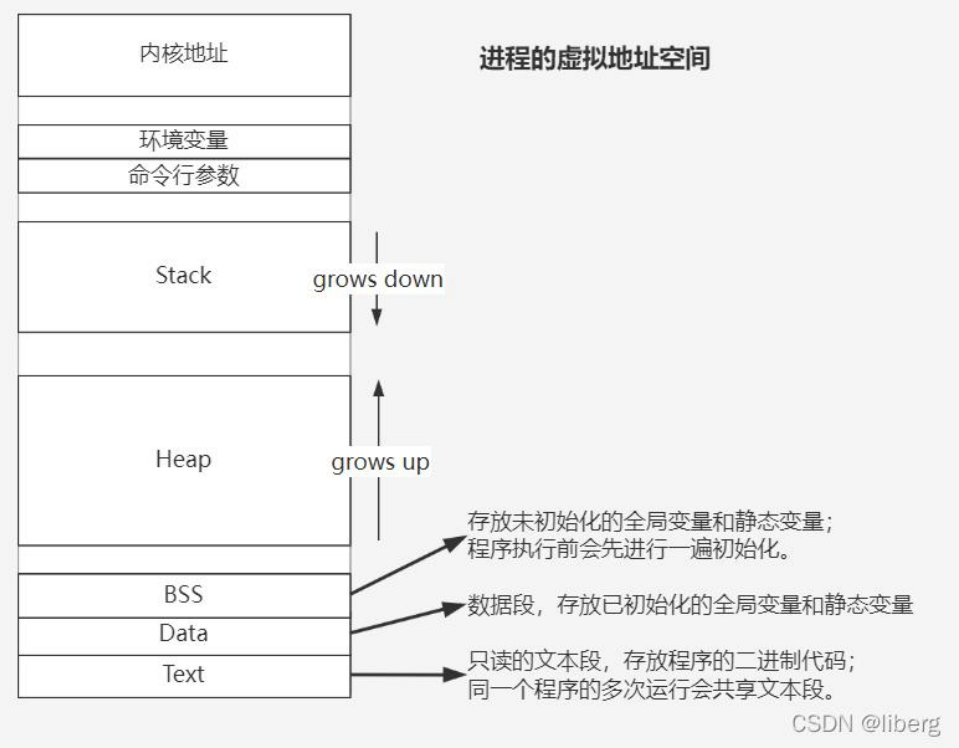
- 一句话解释：谁（获得着）控制这个数据的生死的权利。
- 在Rust里，每一个值都有一个决定其生命周期的唯一的所有者(owner)。

- 所有权的存在原因？

  - 跟踪代码的哪些部分正在使用heap的哪些数据
  - 最小化heap上的重复数据
  - 清理heap上未使用的数据以免空间不足

TinTin

三、栈内存与堆内存



- 全局变量、字符串字面量位于Data段
- Box::new()强制在堆上分配
- String::from将字符串字面量从内存中的代码区（ROData段）复制一份到堆，并用栈上分配的变量指向堆内存。
- 局部变量分配在栈上

I 课程关键内容回顾

TinTin

栈内存

- 固定大小类型的数据：
- 裸指针、普通引用、胖指针(除了指针外还包含其他元数据信息，智能指针也是一种带有额外功能的胖指针，而胖指针实际上又是Struct结构)
- 布尔值
- char
- 各种整数、浮点数
- 数组(Rust数组的元素数据类型和数组长度都是固定不变的)
- 元组

数据存放

堆内存

- 动态大小的类型的数据：
- Vec 动态大小数组
- String 可变字符串
- 其数据部分分布在堆中(被称为allocate buffer)，并在栈中留下胖指针(Struct方式实现)指向实际的数据，栈中的那个胖指针结构是静态大小的。

TinTin

四、所有权规则

- 每个值都有一个变量，这个变量是该值的所有者
- 每个值同时只能有一个所有者
- 当所有者超过作用域（scope）时，该值就会被删除

TinTin

五、以String类型为例，演示内存释放

- 以String类型值演示的原因：

  可以使用from函数从字符串字面值创建出String类型

```
let mut x = String::from("Tintinland Rust入门");    // 这类字符串是可以被修改的
```

TinTin

六、内存分配

- 字符串字面值：编译时就知道其内容和大小，所以其文本内容直接被硬编码到了最终可执行文件里

- String类型：由于其具有可变性，所以需要在heap上分配内存来保存编译时未知的文本内容

  操作系统需要在程序运行时来请求内存：执行`String::from`代码的时候实现堆内存请求

- 用完String之后，需要使用某种方式将内存返回给操作系统

  例如：Java采用的方式——GC垃圾回收机制

  Rust采用的方式：对于某个值来说，当拥有它的变量走出作用范围时，内存会立即自动的交换给操作系统

  · drop函数：当变量走出作用域的时候，Rust会自动调用drop函数

TinTin

七、变量和数据的交互方式

- （1）移动 Move（浅拷贝+令拷贝前的栈数据失效）

  好处：防止二次释放

  - 基础数据类型（u32、f32）被直接压入栈中
  - 复杂数据类型（String）复制栈数据，但并未复制堆数据（只有堆内存才会移动所有权）

（2）克隆Clone（针对堆上面的数据）

- 深度拷贝：使用clone方法

（3）复制Copy（针对栈上面的数据）

- Copy trait可以用于像整数这样完全存放在stack上面的类型，将自己确定的数据在stack上可以快速的复制一遍。
- 如果一个类型实现了Copy这个trait，那么旧的变量在赋值后，仍然可用。
- 如果一个类型或者该类型的一部分实现了Drop trait，那么Rust不允许它再去实现Copy trait了

TinTin

八、所有权与函数

- （1）值传递给函数会发生的行为
  - 复制（i32）实现了Copy trait，往函数里传的是x的副本
  - 移动（String）变量传入到函数里面之后，它就不再有效了

- （2）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
  - 一个变量的所有权总是遵循同样的模式
    - 把一个值赋给其他变量时就会发生移动
    - 当一个包含heap的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

- 默认做复制操作的有：
- 所有的整数类型，比如u32,
- 布尔类型，bool
- 浮点数类型：f32, f64
- 字符类型 char
- 由以上类型组成的元组类型 Tuple，如（i32, i32, char）
- 其它类型，默认都是做值的移动操作。

TinTin

九、如何让函数使用某个值，但不获得其所有权？

- 解决方案：引用（Reference）
  - &符号就表示引用：允许你引用某些值而不取得其所有权
  - 借用：我们把引用作为函数参数的这个行为叫做借用

```
fu main(){
    let s1 = String::from("Rust");
    let len = get_length(&s1);

    println("{}的长度为：{}",s1,len);
}

fn get_length(s:&String) -> usize{
    s.len;
}
```

注意：不可以修改借用的东西！原因：它和变量一样，引用默认也是不可变immutable的！

TinTin

十、可变引用与不可变引用

```
fn main(){
    let mut s = String::from("Rust");
    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;
    println!("{} {} {}",r1,r2,r3);
}
```

- 不可变引用（&T），所有者可以读取引用指向的数据，但不能修改数据。

  &x 是对变量x的不可变引用。

- 可变引用（&mut T），所有者可以读取引用指向的数据，并能够修改指向的数据。

  &mut x是对变量的可变引用。

  存在原因：目前为止，如果要对一个变量绑定的值进行修改，我们只有拥有那个值的所有权才行。

  而很多时候我们无法拥有其所有权，这是就引入了可变引用，以此来修改指向的那个值。

  - 限制1：在特定作用域内，对某一块数据，只能有一个可变的引用。（好处：编译时防止数据竞争）

    解决方案：创建新的作用域，来允许非同时的创建多个可变引用。
  - 限制2：不可以同时拥有一个可变引用和一个不可变引用。但是，多个不可变的引用是可以的。

TinTin

十、课堂案例

1
```
fn main() {
    let a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
}
```

2
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
}
```

3
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{b}");
    println!("{a}");
}
```

4
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;

    println!("{a}");
    println!("{b}");
}
```

5
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;
}
```

而下面的代码不能编译： 6
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;

    println!("{b}");
}
```

而下面的代码又能编译： 7
```
fn main() {
    let mut a = 10u32;
    let b = &mut a;
    *b = 20;
    let c = &a;

    println!("{c}");  // 加了一行这个
}
```

下面的代码又不能编译： 8
```
fn main() {
    let mut a = 10u32;
    let c = &a;          // 这句位置放在了这里
    let b = &mut a;
    *b = 20;

    println!("{c}");
}
```

下面的代码可以编译： 9
```
fn main() {
    let mut a = 10u32;
    let c = &a;
    let b = &mut a;
    *b = 20;

    println!("{b}");     // 这里换成了打印b
}
```

TinTin

用&和&mut来改进函数的定义

第一个例子是将字符串的不可变引用传进函数参数。

```
fn foo(s: &String) {
    println!("in fn foo: {s}");
}
fn main() {
    let s1 = String::from("I am a superman.");
    foo(&s1);
    println!("{s1}");
}
```

然后我们试试将字符串的可变引用传进函数，并修改字符串的内容。

```
fn foo(s: &mut String) {
    s.push_str(" You are batman.");
}
fn main() {
    let mut s1 = String::from("I am a superman.");
    println!("{s1}");
    foo(&mut s1);
    println!("{s1}");
}
```

- 其实函数的形参接受实参的过程，就是变量绑定值的过程，跟前面那些普通的变量绑定是一个道理。不过函数的参数是这个新函数的局部变量。从代码可以看到，Rust的代码非常清晰。如果一个函数参数接受的是可变引用，或所有权参数，那么它里面的逻辑一般都会对引用的资源进行修改。如果一个函数参数只接受不可变引用，那么它里面的逻辑，就一定不会修改被引用的资源。就是这么清晰，太利于代码的阅读了。

TinTin

总结一下：

- 引用的规则：

  在任何给定时刻，只能满足下列条件之一：

  1、一个可变的引用

  2、任意数量不可变的引用

  3、一个资源的可变引用与不可变引用的作用域不能交叠（overlap），即不能同时存在

- 引用必须得一直有效，悬空指针会在编译时直接不通过。

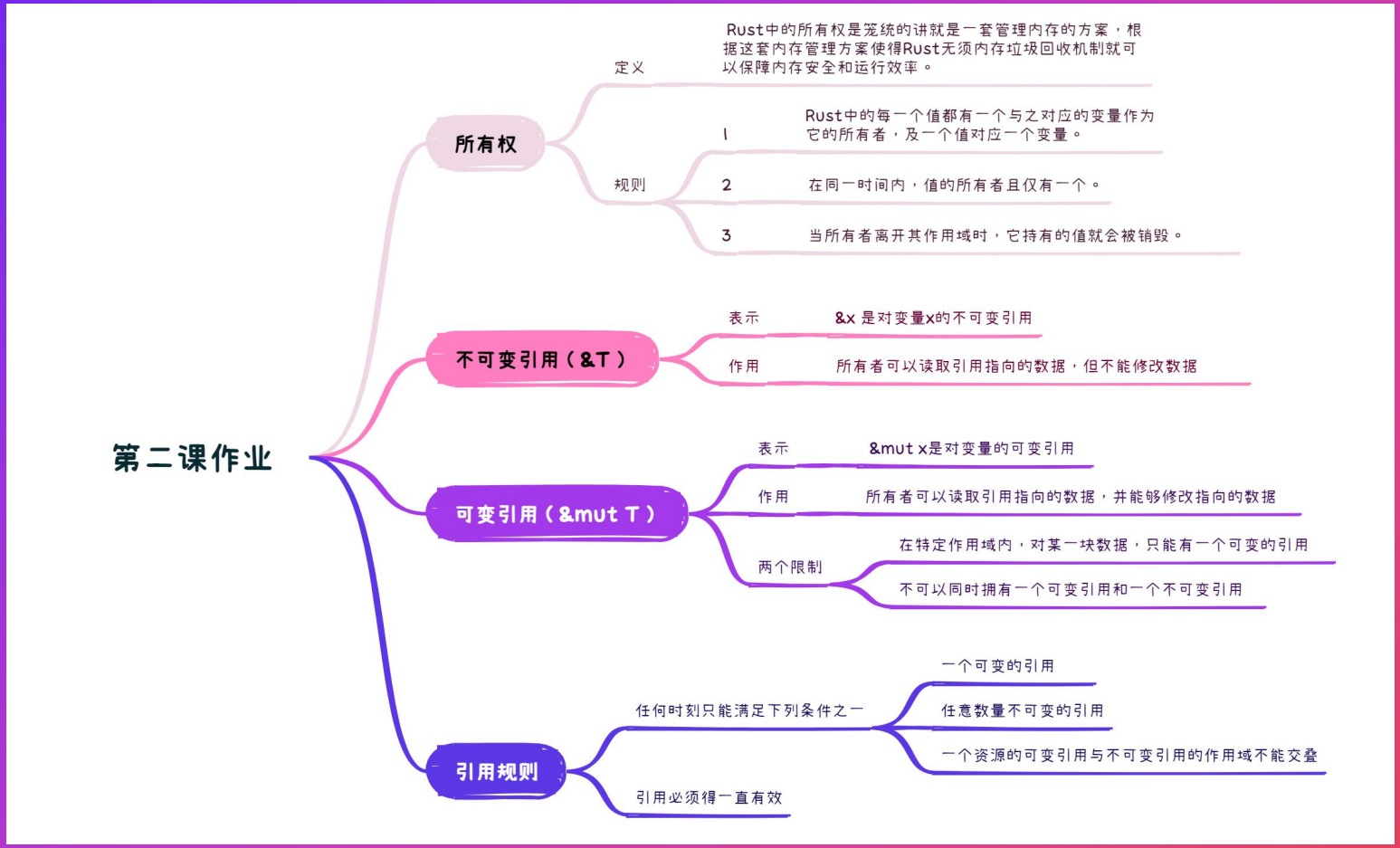
TinTin

题目：

模仿老师的思路，自己对所有权、不可变引用、可变引用这三者的规则或特性做一个集中的总结，写一个笔记列表。

I 作业点评

TinTin



I 优秀作业展示

提交的作业写的都很认真严谨，对作业梳理清晰完整。

优秀作业：@林哲民 ❤❤

TinTin

Week02

Table of content

Homework

Description

模仿老师的思路，自己对所有权、不可变引用、可变引用这三者的规则或特性做一个集中的总结，写一个笔记列表。

Ownership

从一个简单的 String 例子出发

```
fn main() {
    let a = String::from("Hello");
    println!("{a}");
}
```

但是想组出一个 `Hello World` 的时候会出现编译错误(borrow of moved value: `a`)

```
fn main() {
    let a = String::from("Hello");
    let b = a + " World"; // Error
    // let b = a.clone() + " World"; // Correct
    println!("{a}");
    println!("{b}");
}
```

即使将例子简化也会出现一样的错误代码

```
fn main() {
    let a = String::from("Hello");
    let b = a; // Error
    // let b = a.clone(); // Correct
    println!("{a}");
    println!("{b}");
}
```

这时，会发现换个类型的值则符合预期的正常执行。

为什么呢?
在深入研读 The Rust Programming Language，了解下列几点后

1. Rust 借鉴并改进 C++ 的 RAII(Resource Acquisition Is Initialization)
   i. 不需要像 C 一样完全手动管理内存（由程序员呼叫 free）。
   ii. 值会在离开 scope 的时候释放其内存。
2. Memory(内存) 中 Heap(堆) 和 Stack(栈) 相关知识
   i. 固定大小(or 编译期间可以确定大小)类型的值，编译器一般默认将值放在 Stack，如 i32。
   ii. 动态大小(or 编译期间不可确定大小)类型的值，编译器一般默认将值放在 Heap，由一个 Stack 上的变数指向其地址，如 Vec。
3. Rust 中许多类型默认实现 `Copy` trait，是一种 bitwise 的复制方式(i.e. 浅拷贝)，如 i32, bool, char，但 String 类型不属于这个范畴。
   i. String 内部是 Vec<u8>，浅拷贝会产生 2 变数指向同地址，需要 clone (i.e. 深拷贝) 才能连同 Heap 的值复制。

这时我们会发现，Stack 上的值都由编译器管理好了，运行过程中会随着函数呼叫一同增减，困难的在 Heap 上，也就是动态大小的值。
当然，可以默认使用深拷贝，但这非常损耗效能；再来是默认浅拷贝的话，依照 RAII，离开 scope 时又该哪个变数负责回收那同一个内存呢?
毕竟 double-free 也不是我们想要的...

如果我们从 RAII 出发，默认浅拷贝，加上唯一性的概念，有机会得出和 Rust 所有权规则相同的方式。
这里，在不使用 clone 的方式，我们给出了一个可运行的例子，并标注相对应所有权的概念。

```
fn main() {
    let a = String::from("Hello"); // a has the ownership of the String value.
    let b = a; // String value of a is moved to b here. Now b has the ownership and a becomes invalid.
    // println!("{a}"); // Should not use an invalid variable.
    println!("{b}");
} // String value of b is dropped here.
```

最终，我们可以得到如下数条所有权规则，规则 1 & 2 保证了值被唯一一个变数所拥有；规则 3 保证拥有者离开 scope 时释放值的内存，完成内存管理。

所有权规则

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Reference

如果 Rust 只有所有权，想必不是很方便，特别是作为函数的参数时...毕竟很多时候我们会续着函数呼叫后继续使用该变数，不应该因为所有权机制而被强迫将该值作为返回值的一部分。 因此，借用(Borrwing) 机制，即创建引用(Reference) 因应而生。引用的概念也是借鉴 C++ 而来，但 Rust 默认引用和值都是 immutable。
虽然上课时老师使用的词是「引用」，本文尽量按照编译器的讯息解释，所以「借用」和「引用」两个词会穿插使用，但不影响理解。

从一个小例子开始

```
fn main() {
    let a;
    let b = 10;
    a = 80;
    println!("{a}");
```

https://github.com/crosserclaws/share/tree/rust-quick-start-guide/rust-quick-start-guide/week02

I 推荐

TinTin

视频推荐：

1、TinTinLand：Rust入门课 —— Mike老师
2、b站：Rust编程语言入门教程 —— 杨旭老师
3、Youtube：Easy Rust: bite-sized Rust tutorials —— mithradates

文档推荐：

1、博客教程：https://rust-book.junmajinlong.com/ch1/00.html
2、微软教程：https://learn.microsoft.com/en-us/training/paths/rust-first-steps/
3、电子书教程：https://kaisery.github.io/trpl-zh-cn/
4、知乎教程：https://zhuanlan.zhihu.com/p/410291415
5、社区教程：https://fasterthanli.me/articles/a-half-hour-to-learn-rust

Q&A

TinTin

答疑讨论

THANKS

Twitter | YouTube | Discord