程序设计训练之 Rust 编程语言

第二讲：所有权与结构化数据

韩文弢

清华大学计算机科学与技术系

2022 年 8 月 17 日

1

所有权

资源管理的需求

- 内存使用的**安全**和**性能**
- 内存资源
  - 全局对象：事先分配的内存空间段，程序启动时分配，结束时回收。
  - 局部对象：分配在栈上，进入函数时分配，退出函数时回收。
  - 动态对象：分配在堆上，需要时分配，不需要时回收。
- 分配的内存资源什么时候不需要？怎么处理？
  - 用户指定：逻辑精确，编程复杂，性能好
  - 垃圾回收：逻辑模糊，编程简单，性能差

C++ 的做法

- 相比较于 C 进步的地方：
  - `new/delete` 操作符：与面向对象结合，在分配和回收空间的同时完成对象的构造或析构。
  - 拷贝和移动构造语义：根据需要实现内存资源所有权的变换。
    - 拷贝构造：在语义上实现一个对象变两个对象。
    - 移动构造：在语义上实现将一个对象的资源转移给另一个对象。
- 依然存在的问题：
  - 指针和引用无法保证所指向的对象始终有效。
  - 空指针、悬垂指针、双重释放等问题导致运行时错误。

Rust 的做法

引入**所有权 (ownership)** 的概念：

- Rust 作为一门编程语言最大的新特性是提出了显式的所有权。
- 所有权（几乎都）在编译时检查，因此运行时的开销很小。
- Rust 初学者会经常发现自己在和 Rust 编译器的借用检查器斗争，努力使得代码能够编译通过。

所有权规则

- Rust 中的每个值都有所有者 (owner)。
- 同一时刻只有一个所有者。
- 当所有者失效，值也将被丢弃。

所有权与变量绑定

- 变量绑定拥有数据的所有权。
  - 一份数据同时只能有一个所有者。
- 如果一个绑定超出作用域，其绑定的数据将被自动释放。
  - 对于在堆上分配的数据，这意味着释放空间。

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];
    v1.pop();
    v1.push(4);
    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```

移动语义

```rust
let v1 = vec![1, 2, 3];

// Ownership of the Vec object moves to v2.
let v2 = v1;

println!("{}", v1[2]); // error: use of moved value `v1`
```

- `let v2 = v1;`
  - 拷贝数据代价高昂，不希望默认这样来做。
  - 数据不能有多个所有者。
  - 解决方案：把向量的所有权移交给 `v2`，并将 `v1` 置为无效的状态。
- `println!("{}", v1[2]);`
  - 由于 `v1` 已经不再是有效的变量绑定，因此会出错。
- Rust 能够在编译时发现这个问题，从而抛出编译错误。

移动语义（续）

- 移动所有权是编译时的语义，不涉及程序运行时的数据移动。
- 移动是默认行为（通过赋值），不需要像 C++ 那样用 `std::move` 来显式指定。
  - Rust 也可以通过 `std::mem::{replace, take, swap, drop}` 等函数来实现高级所有权管理。

所有权的转移

- 并不是任何时候都想移交所有权。
- 如果每次传递参数都要移交所有权，代码会变得十分繁琐。

```
fn vector_length(v: Vec<i32>) -> Vec<i32> {
    // Do whatever here,
    // then return ownership of `v` back to the caller
    v
}
```

- 可以想到这种方法不具有可扩展性。
  - 如果涉及的变量越多，函数的返回类型就会越长。

借用

- 与其移交所有权，不如进行**借用** (borrow)。
- 可以通过对变量取引用来借用变量中的数据的所有权，此时所有权本身并没有发生变化。
  - 当引用超过作用域，借用也随之结束。
  - 原来的变量依然拥有对数据的所有权。

```
let v = vec![1, 2, 3];

// v_ref is a reference to v.
let v_ref = &v;

// use v_ref to access the data in the vector v.
assert_eq!(v[1], v_ref[1]);
```

借用（续）

- 带来的问题：会给原来的变量增加限制。
- 当一个变量有引用存在时，不能移交它所绑定的数据的所有权。
  - 因为所有权移交会导致引用失效。

```rust
let v = vec![1, 2, 3];
// v_ref is a reference to v.
let v_ref = &v;
// Moving ownership to v_new would invalidate v_ref.
// error: cannot move out of `v` because it is borrowed
let v_new = v;
// Cancel the effect of NLL (non-lexical lifetime)
println!("{:?}", v_ref);
```

非词法生命周期

```
fn main() {
    let v = vec![1, 2, 3];
    let v_ref = &v;
    let v_new = v;
}
```

- 如果 `v_ref` 存活到函数结尾，以上代码不符合引用存在时不能移交所有权的要求。
- 但是，鉴于后续没有再使用，`v_ref` 的生命周期可以提前结束。

非词法生命周期 (NLL, non-lexical lifetime)

- 2018 版新特性
- 对象或引用的生命周期取决于控制流图，而不是词法作用域。
- 能够使得借用检查更加灵活，便于编写程序。
- 详细内容可参见 The Rust RFC Book。

借用与函数

```rust
/// `length` only needs `vector` temporarily, so it is borrowed.
fn length(vec_ref: &Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
}
fn main() {
    let vector = vec![];
    length(&vector);
    println!("{:?}", vector); // this is fine
}
```

- 注意 length 里的类型：vec_ref 以引用的形式传递，类型是 &Vec<i32>。
- 引用默认是不可变的，与变量绑定相同。
- 在超过作用域后，借用结束（length 的结尾）。

可变借用

```rust
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push(vec_ref: &mut Vec<i32>, x: i32) {
    vec_ref.push(x);
}
fn main() {
    let mut vector: Vec<i32> = vec![];
    let vector_ref: &mut Vec<i32> = &mut vector;
    push(vector_ref, 4);
}
```

- 变量还可以通过**可变**引用来进行借用：`&mut vec_ref`
  - 此时，`vec_ref` 是对一个可变的 `Vec` 的引用。
  - 类型是 `&mut Vec<i32>`，而不是 `&Vec<i32>`。

借用与绑定

```rust
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push2(vec_ref: &mut Vec<i32>, x: i32) {
    // error: cannot move out of borrowed content.
    let vector = *vec_ref;
    vector.push(x);
}

fn main() {
    let mut vector = vec![];
    push2(&mut vector, 4);
}
```

- 错误：不能通过解引用然后绑定给变量，这样做会引起数据的所有权转移（同时引用还没有失效）。

借用（续）

- Rust 在大多数情况下会自动解引用，但是有些情况需要显式解引用。
  - 往解引用后的结果里写入内容。
  - 其他可能会引起歧义的情况。

```rust
let mut a = 5;
let ref_a = &mut a;
*ref_a = 4;
println!("{}", *ref_a + 4);
// ==> 8
```

Copy 类型

- Rust 定义了 Copy 特型 (trait)，表示一种类型可以拷贝，而不是用默认的移动语义。
  - 通常这样的类型都是轻量级的，拷贝行为是按位进行的（联系 C++ 中默认的拷贝构造行为）。
  - 大多数基本类型是 Copy 类型（i32、f64、char、bool 等等）。
  - 包含引用的类型不是 Copy 类型（例如，Vec、String）。

```
let x: i32 = 12;
let y = x; // `i32` is `Copy`, so it's not moved :D
println!("x still works: {}, and so does y: {}", x, y);
```

借用规则

思考借用的内在逻辑，体会以下的规则：

- 不能在某个对象不存在后继续保留对它的引用。
- 一个对象可以同时存在多个不可变引用（`&T`）。
- 或者仅有一个可变引用（`&mut T`）。
- 以上两者不能同时存在。

借用的作用

- 考虑迭代器的场景：在修改集合的同时进行迭代访问会引起迭代器失效。
- 这种代码在 C++ 和 Java 等语言中都是可以写出来的。
  - 但是会引发错误，例如 Java 中在运行时会抛 `ConcurrentModificationException` 异常。

```
let mut vs = vec![1,2,3,4];
for v in &vs {
    vs.pop();
    // ERROR: cannot borrow `vs` as mutable because
    // it is also borrowed as immutable
}
```

- `pop` 需要以可变方式借用 `vs` 来修改数据。
- 但是 `vs` 正在以不可变的方式被循环借用。

借用的作用（续）

- 在释放后使用 (use-after-free)
- 这种写法在 C++ 里是能通过编译的，但是可能会导致运行时错误。

```
let y: &i32;
{
    let x = 5;
    y = &x; // error: `x` does not live long enough
}
println!("{}", *y);
```

借用的作用（续）

- 完整的错误信息：

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:13
  |
5 |         y = &x; // error: `x` does not live long enough
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     println!("{}", *y);
  |                     -- borrow later used here
```

- 通过借用机制能够在编译时解决大多数的内存安全问题。

例子：向量

- 向量有三种迭代方式：

```rust
let mut vs = vec![0,1,2,3,4,5,6];

// Borrow immutably
for v in &vs { // Can also write `for v in vs.iter()`
    println!("I'm borrowing {}.", v);
}

// Borrow mutably
for v in &mut vs { // Can also write `for v in vs.iter_mut()`
    *v = *v + 1;
    println!("I'm mutably borrowing {}.", v);
}
```

例子：向量（续）

```
// Take ownership of the whole vector
for v in vs { // Can also write `for v in vs.into_iter()`
    println!("I now own {}! AHAHAHAHA!", v);
}

// `vs` is no longer valid
```

- 以不可变方式借用
- 以可变方式借用
- 获得所有权

切片

- 切片是一种特殊形态的引用，表示引用序列中的一个片段。
- 切片构造的语法是 `&x[s..t]`，其中 `s` 和 `t` 还可以根据情况省略。
  - 也可以使用 `&x[s..=t]` 的语法。
- 可变性以及引用的约束条件对切片同样适用。

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
assert_eq!(slice, &[2, 3]);
```

2

结构化数据

结构化数据

- Rust 有两种创建结构化数据类型的方式：
  - 结构体 `struct`：像 C/C++ 那样的结构体，用于保存数据。
  - 枚举 `enum`：像 OCaml，数据可以是几种类型之一。
- 结构体和枚举都可以有若干实现块 `impl`，用于定义相应类型的方法。

结构体的声明

- 类似于 C++ 的语法
  - 用 `name: type` 来声明结构体的域 (field)，也称为成员变量、字段。

```
struct Point {
    x: i32,
    y: i32,
}
```

- 按照惯例，结构体用 `CamelCase` 命名方式，里面的域用 `snake_case` 命名方式。
- 结构体用下面的方式来初始化（也叫实例化）。

```
let origin = Point { x: 0, y: 0 };
```

结构体的访问

- 结构体的域可以用点记号来访问。
- 结构体不能是部分初始化的。
  - 必须在创建时给所有的域赋值，也可以先声明一个未初始化的结构体，后续再进行初始化。

```
let mut p = Point { x: 19, y: 8 };
p.x += 1;
p.y -= 1;

let q: Point;
q = Point { x: 1, y: 2 };
```

结构体与可变性

- 结构体没有域级的可变性控制。
- 可变性是变量绑定的属性，跟类型无关。
- 域级的可变性（内部可变性）可以通过 `Cell` 类型来实现，后面会讲。

```
struct Point {
    x: i32,
    mut y: i32, // Illegal!
}
```

结构体的访问权限

- 结构体在它所在的模块的名字空间里。
  - Point 的完整名字 (fully qualified name) 是 foo::Point。
- 结构体的域默认是私有的。
  - 可以通过 pub 关键字变成公有。
- 私有域只能在结构体所在的模块内访问。

```
mod foo {
    pub struct Point {
        pub x: i32,
        y: i32,
    }
}
fn main() {
    let b = foo::Point { x: 12, y: 12 };  // error: y is private
}
```

结构体的访问权限（续）

```
mod foo {
    pub struct Point {
        pub x: i32,
        y: i32,
    }

    // Creates and returns a new point
    pub fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }
}
```

- 函数 new 和结构体 Point 在同一个模块内，因此可以访问它的私有域。

结构体的更新语法

- 结构体初始化时可以用 `..s` 从 `s` 中获取部分或者全部的域（拷贝还是移动?）。
- 所有没有在初始化列表里指定的域都从目标结构体里获取。
- 两个结构体的类型必须是一致的。
  - 从不同类型结构体获得相同类型的域也是不行的。

```
struct Foo { a: i32, b: i32, c: i32, d: i32, e: i32 }

let mut x = Foo { a: 1, b: 1, c: 2, d: 2, e: 3 };
let x2 = Foo { e: 4, .. x };

// Useful to update multiple fields of the same struct:
x = Foo { a: 2, b: 2, e: 2, .. x };
```

元组结构体

- 元组结构体 (tuple struct) 是结构体的一种形态，有结构体名字，但没有域的名字。
- 可以像元组那样通过数字来访问域，例如 `x.0`、`x.1` 等等。
- 也可以通过 `match` 来进行匹配。

```
struct Color(i32, i32, i32);

let mut c = Color(0, 255, 255);
c.0 = 255;
match c {
    Color(r, g, b) => println!("({}, {}, {})", r, g, b)
}
```

元组结构体的用途

- 可用来创建新的类型，而不仅仅只是一个别名。
  - 被称为“新类型”模式 (“newtype” pattern)。
- 两种类型在结构上是相同的，但是并不等价（不是同一种类型）。

```
// Not equatable
struct Meters(i32);
struct Yards(i32);

// May be compared using `==`, added with `+`, etc.
type MetersAlias = i32;
type YardsAlias  = i32;
```

单位元结构体（零大小的类型）

- 可以声明零大小的结构体。
  - 这样的结构体没有域。
- 这种结构体也是可以实例化的。
- 通常被用来作为其他数据结构的标记类型。
  - 例如，可以用来指示一个容器保存的数据的类型。

```
struct Unit;

let u = Unit;
```

枚举

- 枚举 (enum)，是**和类型** (sum type)，用来表示可以是多选一的数据。
  - 相对地，结构体和元组都是**积类型** (product type)。
- Rust 的枚举比 C/C++、Java 等语言中的枚举要强很多。
- 枚举的每种变体 (variant) 可以：
  - 没有数据（单位元变体）
  - 有命名的数据域（结构体变体）
  - 有不命名的有序数据域（元组变体）

```
enum Resultish {
    Ok,
    Warning { code: i32, message: String },
    Err(String)
}
```

枚举（续）

- 枚举的变体存在于枚举本身的名字空间中：`Resultish::Ok`
  - 可以使用 `use Resultish::*` 把所有变体引入当前的名字空间。
- 与其他类型相同，也可以对枚举进行匹配。

```
match make_request() {
    Resultish::Ok =>
        println!("Success!"),
    Resultish::Warning { code, message } =>
        println!("Warning: {}!", message),
    Resultish::Err(s) =>
        println!("Failed with error: {}", s),
}
```

- 枚举的构造器 (constructor) 可以像函数一样使用，例如 `Resultish::Ok`。
- 后面介绍迭代器和闭包时会很有用。

递归类型

- 考虑试图创建一种类似函数式编程的 `List` 类型：

```
enum List {
    Nil,
    Cons(i32, List),
}
```

递归类型（续）

- 前面的定义在编译时会出现无穷大小的问题。
- 结构体和枚举默认情况下是内联存储的，因此不能递归。
  - 它们的元素正常情况下不使用引用来存储，但可以显式指定。

```
enum List {
    Nil,
    Cons(i32, List),
}
```

递归类型编译错误

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:2:5
  |
2 |     enum List {
  |     ^^^^^^^^^ recursive type has infinite size
3 |         Nil,
4 |         Cons(i32, List),
  |                   ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List` repr
  |
4 |         Cons(i32, Box<List>),
  |                   ++++    +

For more information about this error, try `rustc --explain E0072`.
```

Box 简介

- Box<T> 是指向堆上对象的指针，作为对象的唯一所有者。
  - Box 唯一拥有它的数据（T 类型），不能创建别名。
- Box 在超过作用域时会自动销毁。
- 通过 Box::new() 来创建 Box。

```
let boxed_five = Box::new(5);

enum List {
    Nil,
    Cons(i32, Box<List>), // OK!
}
```

- 后续讲指针时会进一步讲解。

方法

```
impl Point {
    pub fn distance(&self, other: Point) -> f32 {
        let (dx, dy) = (self.x - other.x, self.y - other.y);
        ((dx.pow(2) + dy.pow(2)) as f32).sqrt()
    }
}
fn main() {
    let p = Point { x: 1, y: 2 };
    p.distance();
}
```

- 结构体和枚举的方法可以实现在 `impl` 代码块里。
- 和域相同，方法也通过点记号进行访问。
- 可以用 `pub` 将方法声明为公开的，`impl` 代码块本身不需要是 `pub` 的。
- 对枚举和对结构体是一样的。

方法与所有权

方法的第一个参数（名字为 `self`）决定这个方法需要的所有权种类。

- `&self`：方法**借用**对象的值。
  - 一般情况下尽量使用这种方式，类似于 C++ 中的常成员函数。
- `&mut self`：方法**可变地借用**对象的值。
  - 在方法需要修改对象时使用，类似于 C++ 中的普通成员函数。
- `self`：方法获得对象的所有权。
  - 方法会消耗掉对象，同时可以返回其他的值。

方法与所有权（例）

```rust
impl Point {
    fn distance(&self, other: Point) -> f32 {
        let (dx, dy) = (self.x - other.x, self.y - other.y);
        ((dx.pow(2) + dy.pow(2)) as f32).sqrt()
    }
    fn translate(&mut self, x: i32, y: i32) {
        self.x += x;
        self.y += y;
    }
    fn mirror_y(self) -> Point {
        Point { x: -self.x, y: self.y }
    }
}
```

关联函数

```rust
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }
}

fn main() {
    let p = Point::new(1, 2);
}
```

- 关联函数与方法类似，但是没有 `self` 参数。
  - 调用时使用名字空间语法：`Point::new()`，而不是 `Point.new()`。
  - 类似 C++ 中的静态成员函数。
- 一般会创建一个名为 `new` 的关联函数起到构造函数的作用。
  - Rust 没有内置的构造函数语法，也不会自动构造。

实现

- 方法、关联函数和函数不能重载。
  - 例如，`Vec::new()` 和 `Vec::with_capacity(capacity: usize)` 都是 `Vec` 的构造函数。
- 方法不能被继承。
  - Rust 中结构体和枚举用的是合成 (compose) 的办法。
  - 特型 (trait) 具有基本的继承功能。

3

模式匹配

结构体的匹配

- 可以用 match 语句对结构体进行解构。

```
pub struct Point {
    x: i32,
    y: i32,
}

match p {
    Point { x, y } => println!("({}, {})", x, y)
}
```

结构体的匹配（续）

- 一些结构体匹配的技巧：

```
match p {
    Point { y: y1, x: x1 } => println!("({}, {})", x1, y1)
}
```

```
match p {
    Point { y, .. } => println!("{}", y)
}
```

- 匹配时域不一定要按照结构体声明时的顺序。
- 将结构体的域列出来，和对应的变量名做绑定。
  - 可以使用 `struct_field: new_var_binding` 的语法来改变绑定的变量名。
- 忽略部分域：使用 `..` 忽略所有没有提到名字的域。

以引用方式匹配

- 使用 `ref` 可以在匹配时获得一个变量的引用（否则是绑定，直接获得所有权）。

```
let x = 17;

match x {
    ref r => println!("Of type &i32: {}", r),
}
```

以引用方式匹配（续）

- 使用 `ref mut` 以可变引用的方式进行匹配。
  - 仅当被匹配的对象是 `mut` 的。

```
let mut x = 17;

match x {
    ref r if x == 5 => println!("{}", r),
    ref mut r => *r = 5
}
```

if let 语句

- 如果只需要单个匹配分支，用 `if let` 语法会比较便捷。
- 以之前的 `Resultish` 类型为例：

```
enum Resultish {
    Ok,
    Warning { code: i32, message: String },
    Err(String),
}
```

if let 语句（续）

- 假设希望在出错时报告错误，其他情况什么也不做。

```
match make_request() {
    Resultish::Err(_) => println!("Total and utter failure."),
    _ => println!("ok."),
}
```

- 可以将上述代码用 if let 改写为：

```
if let Resultish::Err(s) = make_result() {
    println!("Total and utter failure: {}", s);
} else {
    println!("ok.");
}
```

while let 语句

- 还有一个类似的 `while let` 语句，作用是循环迭代直至匹配条件失败。

```rust
let mut v = vec![1, 2, 3];
while let Some(x) = v.pop() {
    println!("{}", x);
}
```

内部绑定

- 对于更复杂的数据结构，可以用 @ 创建内部元素的变量绑定。

```
#[derive(Debug)]
enum A { None, Some(B) }
#[derive(Debug)]
enum B { None, Some(i32) }

fn foo(x: A) {
    match x {
        a @ A::None                => println!("a is A::{:?}", a),
        ref a @ A::Some(B::None) => println!("a is A::{:?}", *a),
        A::Some(b @ B::Some(_))  => println!("b is B::{:?}", b),
    }
}
```

内部绑定（续）

```
foo(A::None);                // ==> x is A::None
foo(A::Some(B::None));       // ==> a is A::Some(None)
foo(A::Some(B::Some(5)));    // ==> b is B::Some(5)
```

模式匹配的穷尽性

- match 的所有分支对于模式来说必须是穷尽的。

```
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
            // error: not exhaustive
        }
    }
    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

模式匹配的使用场合

- 模式匹配语句 `match`
- 变量绑定，包括 `let`、`if let`、`while let`
- `for` 循环
- 函数和闭包的参数

for 循环的模式匹配

```rust
let v = vec![1, 2, 3];
for (i, x) in v.iter().enumerate() {
    println!("v[{i}] = {x}");
}
```

- 在 `for` 循环的 `for` 和 `in` 之间描述循环变量时可以使用模式匹配。

函数参数的模式匹配

```rust
fn tuple_add((a, b): (i32, i32)) -> i32 {
    a + b
}
fn main() {
    tuple_add((1, 2));
}
```

描述函数参数时可以使用模式匹配。

4

小结

本讲小结

- 所有权与借用
  - 一个变量同时只能存在以下两类引用之一：
    - 多个不可变引用
    - 一个可变引用
  - 函数的参数和返回值与变量绑定的规则相同。
- 结构化数据类型
  - 结构体：一种积类型
  - 枚举：一种和类型
- 复杂的模式匹配

下一讲：标准库