

程序设计训练 (Rust 语言)



第 8 讲：高级特性与编程语言综述

韩文弢

清华大学

2024—2025 学年度夏季学期

本讲内容

8.1 内存模型	2
8.2 不安全代码	6
8.3 外部函数接口	17
8.4 宏	26
8.5 编程语言综述	32
8.6 小结	45

8.1 内存模型

8.1.1 内存布局

Rust 是内存安全、无垃圾回收器的编程语言，目前尚无严格定义的内存模型，内存布局大体如下：

- 代码区 (code)
 - 存放程序的二进制代码。
 - 代码区通常是只读的。
- 栈 (stack)
 - 存放函数的局部变量、函数参数、返回地址等。
 - 栈是有限的，通常 1MB 到 8MB。
- 堆 (heap)
 - 存放动态分配的数据。
 - 堆的大小可以在运行时动态调整，只受系统内存大小的限制。
- 静态区 (static)
 - 存放全局变量、常量、静态变量等。
 - 静态区在程序启动时分配，程序结束时释放。

8.1.2 位置表达式与值表达式

- 位置表达式 (place expressions) 是表示内存位置的表达式。
 - 如：局部变量、静态变量、解引用、数组索引、字段引用等。
 - 可以出现在赋值的左边¹。
- 其余都是值表达式 (value expressions)。
 - 如：字面量、函数调用、运算表达式等。
 - 不能出现在赋值的左边。

¹历史上，位置表达式被称为左值 (lvalues)，值表达式被称为右值 (rvalues)。

8.1.3 移动语义（复制语义）

当位置表达式在值表达式上下文中被求值时，或在模式中被值绑定时，此时该值会被**保存进**当前表达式代表的内存地址。

- 非 Copy 类型：值会被**移动**过来，原位置表达式不再有效。
- Copy 类型：值会被**复制**过来，原位置表达式仍然有效。

8.2 不安全代码

8.2.1 Rust 的安全性检查

- Rust 的安全性检查能够帮助程序员写出安全的代码，但是检查偏保守。
- 有的 Rust 程序从语义上来说安全的，但不能通过编译。
- Rust 提供了**不安全**特性，可以绕开编译器的限制。

8.2.2 不安全特性的用处

- Rust 的不安全特性可用于：
 - 与其他编程语言（例如 C 语言）进行交互。
 - 编写标准库没有提供的底层抽象。
 - 使用不具备所有权语义的类型。
 - 实现标准库 `std` 中的组件。

8.2.3 不安全特性的用法

- 所有不安全的代码都要用 `unsafe` 关键字标注。
- 不安全代码可以放在一个 `unsafe` 代码段中。
- 也可以把整个函数标注为不安全的。
 - 调用不安全函数的代码需要放在 `unsafe` 代码段中。

```
1 unsafe fn foo() { }  
2  
3 fn main() {  
4     unsafe {  
5         foo();  
6     }  
7 }
```



8.2.4 不安全的特型

- 特型也可以设计为不安全的。
 - 此时，特型的 `impl` 代码段也要标注为不安全的。

```
1 unsafe trait Sync { }  
2 unsafe impl Sync for Foo { }
```



- **安全**的操作是指不会违反 Rust 的所有权、类型检查、内存安全规则的操作。
- 尽管可能有危害，下列操作在 Rust 中还是认为是安全的：
 - 死锁
 - 内存泄露
 - 整数溢出
 - 退出时没有运行销毁代码

8.2.6 不安全

- **不安全**的操作是指那些有可能引起危害，且 Rust 编译器无法通过分析来鉴别的操作。
- 以下操作是不好的（可能会引起危害），在 Rust 里只可能在 `unsafe` 代码段中发生：
 - 数据竞争
 - 解析非法指针
 - 读取未初始化的内存
 - 创建非法的基础类型数据
 -

8.2.7 不安全的 Rust

- 在 `unsafe` 代码段中可以做下列事情：
 - 对裸指针解引用
 - 修改静态变量
 - 实现 `unsafe` 特型
 - 调用 `unsafe` 函数
- `unsafe` 不会关闭借用检查器或者让代码破坏常规的 Rust 语义。
- 正常安全的东西在 `unsafe` 中仍然是安全的。

8.2.8 unsafe 用在哪里？

- unsafe 实际上无处不在。
- 因为 std 中很多类型是建立在 unsafe 的基础上的。
 - Vec
 - Cell/RefCell
 - Box
 - Rc/Arc
 - Send/Sync
 - HashMap
 -

8.2.9 不安全代码的正确性

- std 使用 unsafe 代码，如何保证它的正确性？
- 编写 std 时用 unsafe 会**非常小心**。
- 目前 std 的安全性还没有做形式化证明，只能相信它是安全的。
- 形式化证明是一个很重要的编程语言的研究方向。

8.2.10 常见的不安全功能

- 裸指针（只有解引用是不安全的）
- 未初始化的内存
- 类型的重解释转换（transmutation, 等同于 C++ 中的 `reinterpret_cast`）
- 不受限的生命周期

8.3 外部函数接口

8.3.1 外部函数接口

- FFI (Foreign Function Interface)
- FFI 表示从一种语言去调用另一种语言写的代码。
- 对 Rust 来说，这意味着：
 - Rust 调用 C
 - C 调用 Rust
 - 其他语言（例如 Python）调用 Rust
- 为什么主要是要实现跟 C 语言相互调用？
 - 因为 C 语言是 FFI 实际上的标准。

8.3.2 从 Rust 调用 C

- 动机
 - 调用已有的库，这些库用 Rust 重新实现代价太大。
 - 与其他语言对接，很多语言都有 C 的绑定。
- 在 Rust 里调用外部函数是不安全的。
 - 因为 C 是不安全的语言。

8.3.3 从 Rust 调用 C 的做法

- 把 C 代码编译成静态库 (.a / .lib)。
 - `cc -c -o foo.o foo.c`
 - `ar rcs libfoo.a foo.o`
- 或者编译成动态库 (.so / .dylib / .dll)。
 - `cc -c -fPIC -o foo.o foo.c`
 - `cc -shared -fPIC -o libfoo.so foo.o`

在 C 中：

```
1 int32_t foo() { return 10; }
```



在 Rust 中：

```
1 #[link(name = "foo", kind = "static")] // links libfoo.a.
2 extern {
3     fn foo() -> i32;
4 }
5
6 #[link(name = "foo")] // links libfoo.so.
7 extern {              // By default, this is also
8     fn foo() -> i32;  // statically linked.
9 }
```



8.3.5 从 Rust 调用 C 的例子

调用外部函数是不安全的：

```
1 fn main() {  
2     println!("foo: {}", unsafe { foo() });  
3 }
```



8.3.6 从 C 调用 Rust

- 动机
 - 用 Rust 编写安全的代码。
 - 用一种更好用的语言来写 C 代码的一部分。
- Rust 有自己的内存布局和调用规范，和 C 的不同。
- 为了能够从 C 调用 Rust，需要遵循 C 的规范。

```
1 #[repr(C)]  
2 pub enum Color { Red = 1, Blue, Green = 5, Yellow }  
3  
4 #[repr(C)]  
5 pub struct Bikeshed { height: f64, area: f64 }  
6  
7 extern "C" pub fn paint(bs: Bikeshed, c: Color) { /* ... */ }
```



Rust

使用新的结构体来隐藏 C 不支持的特性：

```
1 struct X<T>(T); // C 没有泛型  
2  
3 #[repr(C)]  
4 pub struct Xi32(X<i32>); // 这个结构体隐藏了类型参数
```



Rust

8.3.8 从 C 调用 Rust

- 把 Rust 代码编译成静态库或动态库。
 - 对于静态库的方式，在 C 代码编译时链接。
 - 对于动态库的方式，在 C 代码中动态装载。
- 可以使用 rust-bindgen 自动产生 Rust 代码对应的 C 头文件。

8.4 宏

- 宏 (macros) 是预先定义好的文本替换规则。
- C/C++ 语言的构建过程分为**预处理**、编译、汇编、链接四步，其中预处理要做的事情就是把宏展开。
- C/C++ 的宏的问题：只是做直接的词法单元 (tokens) 级别的替换，没有更多的信息。

```
1 #define SUB(x, y) x - y
2 int c = -SUB(2, 3 + 4);
```



会展开成

```
1 int c = -2 - 3 + 4;
```



8.4.3 C 的宏的问题

```
1 #define SWAP(x, y) do { \  
2     (x) = (x) ^ (y);    \  
3     (y) = (x) ^ (y);    \  
4     (x) = (x) ^ (y);    \  
5 } while (0) // 很奇怪的定义多行内容的写法  
6  
7 int x = 10;  
8 SWAP(x, x); // x 是 0，而不是 10
```



- Rust 中，宏是语法扩展功能的组成部分。
 - `#[foo]` 和 `#![foo]`，属性
 - `foo! arg`，一般是 `foo!(...)`、`foo![...]` 或 `foo!{...}`，宏展开
 - `foo! arg arg`，只有 `macro_rules! name { definition }`，声明式的宏定义
- Rust 的宏可以由两种不同的方式来定义：
 - 声明式的宏（declarative macros）
 - 过程式的宏（procedural macros）
- 宏完成的功能是从一棵抽象语法树（abstract syntax tree, AST）转换成另一棵抽象语法树。

```
1 macro_rules! myvec {
2     ( $(          // 开始一段重复出现的内容
3         $elem:expr // 每次重复匹配一个表达式
4     ) ,          // 以逗号分隔
5     *            // 零次或多次重复
6 ) => {
7     { // 花括号用于创建恰好一个语法单位（代码段）
8         let mut v = Vec::new();
9
10        $(          // 展开一次重复
11            v.push($elem); // 展开的内容
12        ) *          // 零次或多次重复
13
14        v            // 产生的代码段最后返回 v
15    }
16 }
17
18
19 println!("{:?}", myvec![3, 4]);
```



8.5 编程语言综述

8.5.1 什么是编程语言？

- 编程语言（programming languages），是用来定义计算机程序的形式语言。
- 它是一种被标准化的交流技巧，用来向计算机发出指令，一种能够让程序员准确地定义计算机所需要使用数据的计算机语言，并精确地定义在不同情况下所应当采取的行动。
- 最早的编程语言是在计算机发明之前产生的，当时是用来控制机器的（例如，提花织布机、自动演奏钢琴）。

8.5.2 编程语言的历史

- 1955 年, FORTRAN: 主要用于科学计算
- 1958 年, Lisp: 函数式语言, 主要用于人工智能
- 1959 年, COBOL: 用于商业系统
- 1962 年, Simula: 第一个支持面向对象的语言
- 1964 年, BASIC: 适合初学者使用
- 1969 年, C: 早期的系统编程语言
- 1970 年, Pascal: 适合教学使用
- 1983 年, C++: 功能丰富的系统编程语言
- 1991 年, Python: 易用的解释型语言

8.5.3 常见编程语言

- C：面向底层系统编程
- C++：增加更多的编程方式
- Rust：强调安全性
- Java：通过虚拟机在不同平台上运行
- Python：语法方便，动态解释执行
- JavaScript：与浏览器紧密集成
- Haskell：纯函数式编程语言

8.5.4 C++ 语言：发展历史

- 作者：Bjarne Stroustrup
- 发展历史
 - 1979 年，C with Classes，为了平衡软件开发效率（如 Simula 语言）与软件运行速度（如 BCPL 语言），特性有类、派生、内联、默认参数等。
 - 1982 年，演进为 C++，新特性有虚函数、函数与操作符重载、引用、常量、类型安全的内存分配（new/delete）、单行注释（//）。
 - 1984 年，基于流的输入输出库（iostream），吸收了用操作符进行 I/O 的想法。
 - 1989 年，C++ 2.0 发布，新特性有多重继承、抽象类、静态成员函数、常成员函数、保护级别的成员等。

8.5.4 C++ 语言：发展历史

- 1990 年，增加模板、异常、名字空间、新式类型转换（各种 `cast`）、布尔类型。
- 1998 年，C++98 标准发布（ISO/IEC 14882:1998，第一个标准）。2003 年，发布小更新 C++03 标准。
- 2011 年，C++11 标准发布，增加大量新特性（如：右值引用与移动语义、类型推断、Lambda 函数、哈希表等），C++ 进入现代（Modern C++）。后续发布两个小更新 C++14 和 C++17。
- 2020 年，C++20 标准发布，增加概念、模块等新特性。

8.5.5 Java 与 Python

- Java
 - 作者：James Gosling
 - 出现年份：1995 年
 - 特点：面向对象（基于类）、指令式、反射、泛型、函数式、并发
 - 注意：JVM (Java Virtual Machine)
- Python
 - 作者：Guido van Rossum
 - 出现年份：1991 年
 - 特点：面向对象、指令式、结构化、反射、函数式
 - 注意：GIL (Global Interpreter Lock)

8.5.6 新兴编程语言

- Zig: Better C, 编译期泛型
- Mojo: Better Python, C++性能, AI
- Kotlin: Better Java, 安卓平台使用最广泛的开发语言
- Swift: 苹果平台的首要开发语言
- 仓颉: 鸿蒙的生态语言
- MoonBit: AI 云原生开发平台

8.5.7 Zig 与 Mojo

- Zig
 - 作者：Andrew Kelley
 - 出现年份：2016 年
 - 特点：指令式、函数式、泛型、并发、反射
 - 注意：Allocator + Option type 内存管理（类似用指针加布尔变量的结构体指示地址是否有效）
- Mojo
 - 作者：Chris Lattner
 - 出现年份：2023 年
 - 特点：指令式、函数式、泛型
 - 注意：基于 MLIR，面向 AI 应用程序

8.5.8 Swift 与 Kotlin

- Kotlin
 - 作者：JetBrains
 - 出现年份：2011 年
 - 特点：面向对象（基于类）、指令式、函数式、结构化、反射、泛型、并发
 - 注意：安卓生态，与 Java 互操作
- Swift
 - 作者：Chris Lattner
 - 出现年份：2014 年
 - 特点：面向对象（基于类）、指令式、函数式、结构化、声明式 (SwiftUI)、并发
 - 注意：苹果生态，ARC (Automatic Reference Counting)

8.5.9 仓颉

- 作者：冯新宇
- 出现年份：2019 年
- 特点：面向对象（基于类）、函数式、指令式、泛型、并发
- Try-with-resources
 - 类似 try-catch-finally 做异常处理
 - 资源在发生异常或表达式结束后都会自动释放
- 全并发 GC
 - 消除 STW 停顿
 - GC 线程与应用线程轻量同步
 - 安全点 + 内存屏障
 - 减少内存碎片
 - 基于指针跳动 (bumping-pointer) 的对象分配方式
 - 指针标记降低 GC 开销

- 作者：张宏波
- 出现年份：2022 年
- 特点：面向对象、函数式、指令式、泛型、异步
- AI 原生平台
 - 扁平化语言设计，适用于 Transformer 架构
 - MoonBit AI 助手，智能代码生成，支持实时测试
- 开箱即用的工具链
 - 程序语言、编译器、构建系统和集成开发环境的协同设计保证整个系统的一致性
 - 兼顾编译时性能及运行时性能

8.5.11 编程语言的选择

- 根据任务类型来选择编程语言
- 考察的方面
 - 行业背景（语言的生态支持）
 - 运行环境（服务器、桌面、移动设备）
 - 对性能的要求
 - 是否需要分布式
 - 开发者和维护者的背景

8.6 小结

8.6.1 本讲小结

- 内存模型
- 不安全特性 `unsafe`
- 外部函数调用 (FFI)
- 宏
- 编程语言综述

8.6.2 课程总结

- 学习一门新的语言。
- 通过实践锻炼解决问题的能力。
- 感受编程语言的设计理念。