程序设计训练之 Rust 编程语言

第八讲：高级特性与编程语言综述

韩文弢

清华大学计算机科学与技术系

2023 年 7 月 13 日

1

不安全代码

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 程序从语义上来说是安全的，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。

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。

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。

```
unsafe fn foo() { }

fn main() {
    unsafe {
        foo();
    }
}
```

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。

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

安全

- **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

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，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

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。

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
  - ……

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。

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)
- 不受限的生命周期

2

外部函数接口

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。

从 Rust 调用 C

- 动机
  - 调用已有的库（例如 OpenSSL），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。

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`

从 Rust 调用 C 的例子

- 在 C 中：

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

- 在 Rust 中：

```rust
#[link(name = "foo", kind = "static")]  // links libfoo.a.
extern {
    fn foo() -> i32;
}

#[link(name = "foo")]  // links libfoo.so.
extern {               // By default, this is also
    fn foo() -> i32;   // statically linked.
}
```

从 Rust 调用 C 的例子

- 调用外部函数是不安全的：

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

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。

使用 C 的规范

```rust
#[repr(C)]
pub enum Color { Red = 1, Blue, Green = 5, Yellow }

#[repr(C)]
pub struct Bikeshed { height: f64, area: f64 }

extern "C" pub fn paint(bs: Bikeshed, c: Color) { /* ... */ }
```

- 使用不透明的结构体来隐藏 C 不支持的特性：

```rust
struct X<T>(T); // C doesn't have generics.

#[repr(C)]
pub struct Xi32(X<i32>); // This struct hides the type parameter.
```

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。

3

宏

宏

- 宏 (macros) 是预先定义好的文本替换规则。
- C/C++ 语言的编译分为**预处理**、编译、链接三步，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。

C 的宏的问题

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

会展开成

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

C 的宏的问题

```
#define SWAP(x, y) do { \
    (x) = (x) ^ (y);    \
    (y) = (x) ^ (y);    \
    (x) = (x) ^ (y);    \
} while (0)  // Also, creating multiple statements is weird.
int x = 10;
SWAP(x, x);  // `x` is now 0 instead of 10
```

Rust 的宏的设计

- Rust 中，宏是语法扩展 (syntax extensions) 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。

声明式的宏举例

```
macro_rules! myvec {
    (   $(              // Start a repetition
            $elem:expr  // Each repetition matches one expr
        ) ,             // Separated by commas
          *             // Zero or more repetitions
    ) => {
        { // Braces so we output only one AST (block kind)
            let mut v = Vec::new();

            $(                  // Expand a repetition
                v.push($elem); // Expands once for each input rep
            ) *                 // No sep; zero or more reps

            v                   // Return v from the block.
        }
    }
}
println!("{:?}", myvec![3, 4]);
```

4

编程语言综述

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）。

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

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

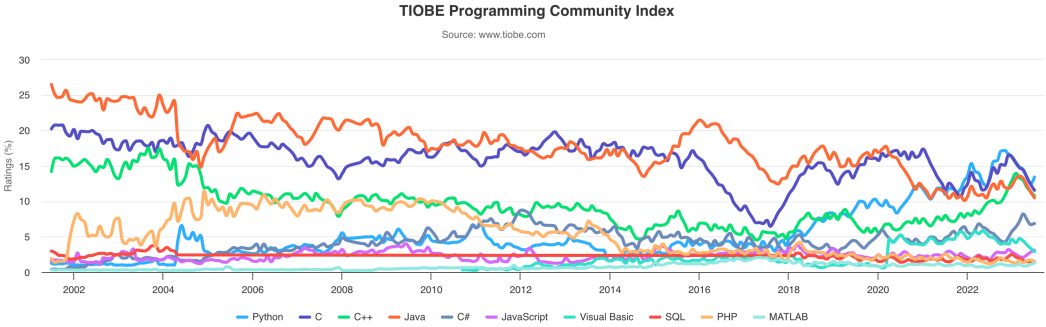
C++ 语言：发展历史

- 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。
  - 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。

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)

TIOBE 编程语言指数



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

5

小结

本讲小结

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

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。