# 第四章 函数

本章要点：如何定义函数；传递参数的多种方式；调用本模块、外部模块、其他编程语言定义的函数；闭包的定义和使用。

通过前面几章内容的介绍，我们现在能够用Rust写一些简单的程序。本章我们将介绍函数。函数是Rust编程语言的重要组成部分。函数能够帮助我们封装过程，实现代码重用。结合Rust中的高阶函数、闭包、泛型、trait对象，可以让我们在Rust中实现函数式编程，达到更高的抽象水平。

## 4.1 函数声明与定义

函数在Rust中十分常见。在一个能够生成可执行文件的Rust程序中，我们需要main函数作为该程序执行时的入口。一个main函数的定义如代码片段ch4\_1.rs所示。

*//这是一个main函数实例*

fn main() {

    println!("demo that defines a main function");

}

代码片段ch4\_1.rs

### 4.1.1 函数的基本语法

**1. 函数的声明。**

不同于C语言，Rust不支持函数的声明。Rust将函数作用域管理的主要工作交给了模块（module）和Crate。Rust中的函数可以在其定义所属模块的任意位置被调用。我们也可以通过设置模块的可见性，以及对模块的引用，控制函数在其他模块中的作用域。

**2. 函数的定义。**

在Rust中，我们用fn关键字表示函数定义的起始。之后是函数名、参数列表、返回值类型和函数体。Rust中的函数名通常采用蛇形命名法：用下划线‘\_’分割单词，所有字母小写。参数列表用‘()’表示，内部用逗号‘,’分隔每个参数，每个参数以‘参数名:类型’的格式定义。参数列表可以为空。返回值类型用一个箭头指出，格式为‘-> 返回值类型’。函数体在‘{}’内，是表示函数执行过程的语句块。除函数体外，一个函数定义的其余部分被称作函数签名。一个函数的定义如代码片段ch4\_2.rs所示，这个例子定义了一个返回参数增加1的函数，参数和返回值的类型都为i32。

*//这是一个函数定义的实例*

fn plus\_one\_i32 (param1: i32) -> i32 {

    return param1 + 1;

}

代码片段ch4\_2.rs

3**. 函数的返回值。**

在Rust中，我们用return语句来表示函数要返回的变量，如上面的代码片段ch4\_2.rs所示；也可以在语句块的最后位置，用一个表达式作为返回值，如代码片段ch4\_3.rs所示。

*//这是一个表达式作返回值的实例*

fn plus\_one\_i32 (param1: i32) -> i32 {

    param1 + 1

}

代码片段ch4\_3.rs

return语句能够让控制流提前返回，而表达式作返回值时没有这一功能，如代码片段ch4\_4.rs所示。

fn abs\_i32(x: i32) -> i32 {

    if x > 0 {

*//这里不能用表达式‘x’代替return语句*

*//因为表达式无法让控制流提前返回*

        return x;

    }

    -x

}

代码片段ch4\_4.rs

Rust允许函数签名中不标注返回值类型。此时，Rust会给这个函数赋予缺省返回值类型：空元组类型‘()’。代码片段ch4\_1.rs中的main函数就是这样的一个例子。

Rust函数只允许唯一的返回值。如果我们想要同时返回多个值，可以用元组实现，如代码片段ch4\_5.rs所示，定义了一个函数，以一个Vec<i32>变量为参数，返回其算术平均值和几何平均值。

*//求i32数组v的几何平均数和算数平均数*

fn mean\_arithmetic\_geometric(v: Vec<i32>) -> (f64, f64) {

*//初始化几何平均数为0和算术平均数为1*

**let** **mut** arithmetic\_mean = 0f64;

**let** **mut** geometric\_mean = 1f64;

*//对v中的每个元素求和、求积*

    for x in &v {

        arithmetic\_mean += \*x as f64;

        geometric\_mean \*= \*x as f64;

    }

*//求得几何平均数和算数平均数*

    arithmetic\_mean /= v.len() as f64;

    geometric\_mean = geometric\_mean.powf(1f64/v.len() as f64);

*//以元组形式返回几何平均数和算数平均数*

    (arithmetic\_mean, geometric\_mean)

}

代码片段ch4\_5.rs

Rust函数返回值的类型也是唯一的。这是因为Rust编译器在编译函数时需要确定返回值在栈中需要分配的空间。我们在需要返回实现了某些特定trait的变量，但只有在运行时才能确定这些变量类型的时候，就要注意这一条规则。下面的代码片段ch4\_6.rs（a）定义了一个trait Authors,以及实现了该trait的类型Paper和Book。Authors定义的方法get\_authors(&self)让我们能够获取文献的作者信息。

**trait** Authors {

    fn get\_authors(&self) -> Vec<String>;

}

**struct** Paper {

    title: String,

    authors: Vec<String>,

    doc\_id: String,

}

impl Authors for Paper {

    fn get\_authors(&self) -> Vec<String> {

        return self.authors.clone();

    }

}

**struct** Book {

    name: String,

    authors: Vec<String>,

    publisher: String,

}

impl Authors for Book {

    fn get\_authors(&self) -> Vec<String> {

        return self.authors.clone();

    }

}

代码片段ch4\_6.rs（a）

现在，我们设计了一个函数get\_authors(x: i32)来模拟查询文献作者的过程，如代码片段ch4\_6.rs（b）所示，当参数x为1时返回paper，其他情况时返回book。为了限制调用者只能够访问返回值的作者信息，我们用impl Authors作为返回值类型。非常遗憾的是，尽管这段代码看上去完美的实现了我们预想的功能，但是它无法通过编译。因为在编译这段代码的时候，编译器无法确定函数返回值类型在栈上的大小，即便Paper和Author这两个类型结构等价。

*// fn get\_authors(x: i32) -> impl Authors {*

*//     let paper = Paper {*

*//         title: String::from("Paper Title"),*

*//         authors: vec![String::from("a")],*

*//         doc\_id: String::from("1")*

*//     };*

*//     let book = Book {*

*//         name: String::from("ProgrammingInRust"),*

*//         authors: vec![String::from("b")],*

*//         publisher: String::from("c")*

*//     };*

*//     if x == 1 {*

*//         return paper;*

*//     }*

*//     else {*

*//         return book;*

*//     };*

*// }*

代码片段ch4\_6.rs（b）

Rust的智能指针Box类型给代码片段ch4\_6.rs（b）中遇到的问题提供了解决方案，如代码片段ch4\_6.rs（c）所示，我们将返回值类型修改为Box<dyn Authors>。这样我们在栈上只保存一个指向堆区的指针作为返回值，而堆上的返回值到运行时再去确定具体的数据类型，从而保证了函数返回值类型的唯一性。

fn get\_boxed\_authors(x: i32) -> Box<dyn Authors> {

**let** paper = Paper {

        title: String::from("Paper Title"),

        authors: vec![String::from("a")],

        doc\_id: String::from("1")

    };

**let** book = Book {

        name: String::from("ProgrammingInRust"),

        authors: vec![String::from("b")],

        publisher: String::from("c")

    };

    if x == 1 {

        return Box::new(paper);

    }

    else {

        return Box::new(book);

    };

}

代码片段ch4\_6.rs（c）

### 4.1.2 函数定义的Shadow机制

在Rust中，我们可以在不同的作用域中定义同名的函数。这两个作用域可以是不相关的，也可以具有嵌套关系。如代码片段ch4\_7.rs所示，我们在两个具有嵌套关系的作用域中分别定义了两个ch4\_3.rs中的plus\_one\_i32函数，它们具有不同的参数列表和函数体。

*// 位于ch4\_7.rs作用域*

fn plus\_one\_i32 (param1: i32) -> i32 {

    println!("ch4\_7.rs scope");

    param1 + 1

}

fn main() {

*// 位于ch4\_7.rs-main作用域*

    fn plus\_one\_i32 (param1: &i32) -> i32 {

        println!("ch4\_7.rs-main scope");

        param1 + 1

    }

*// 用位于ch4\_7.rs-main作用域的plus\_one\_i32*

    println!("main: {}", plus\_one\_i32(&1));

*// 调用位于ch4\_7.rs作用域的plus\_one\_i32*

    println!("main: {}", self::plus\_one\_i32(0));

}

代码片段ch4\_7.rs

代码片段ch4\_7.rs的运行结果如下所示：

ch4\_7.rs-main scope

main: 2

ch4\_7.rs scope

main: 1

从代码片段ch4\_7.rs的实现及其运行结果可以看出，在main函数中定义的plus\_one\_i32函数通过Shadow机制将外部定义的plus\_one\_i32函数定义遮蔽，从而在main函数中无法直接调用main函数外部定义的plus\_one\_i32函数。但是在main函数作用域中，外部定义的plus\_one\_i32函数并不是不可见的，它仍然可以通过显示标注crate符号被调用。

## 4.2 参数传递

函数参数在函数与上下文环境的互动中起着重要的作用。运用传递参数的不同方式，可以让函数获取上下文中的信息，以及改变上下文。

### 4.2.1 按值传递参数

1**. 不可变按值传递参数。**

下面的代码片段ch4\_8.rs实现了返回String类型参数s的最后一个字符的u8编码。参数s的定义的传参方式就是按值传递参数。

*//取参数s串最后一位的u8编码*

fn tail(s: String) -> u8 {

    if !s.is\_empty() {

        s.as\_bytes()[s.len() - 1]

    }

    else {

        0

    }

}

fn main() {

**let** s = String::from("ProgrammingInRust");

    assert\_eq!(tail(s), 't' as u8);

*// tail(s)获得了s的所有权，*

*// tail(s)调用结束后将不能再使用s。*

}

代码片段ch4\_8.rs

按值传递参数会将参数的所有权转移到函数中。在函数返回后，调用者因不再拥有参数的所有权，无法被再次使用。

如果把代码片段ch4\_8.rs中函数的参数类型替换为i32或者其他一些简单的数据类型，如代码片段ch4\_9.rs所示。我们会发现在函数调用结束后能够再次使用这个变量。难道是这个变量的所有权又被归还了？情况不是这样的！

*// 取i32类型参数x的个位数字*

fn tail(x: i32) -> i32 {

    x % 10

}

fn main() {

**let** one = 1;

    assert\_eq!(tail(one), 1);

*// 这里，变量one在调用tail(one)后可以被再次使用。*

    assert\_eq!(tail(one), 1);

}

代码片段ch4\_9.rs

在按值传递参数时，Rust会将对应的变量做一次浅拷贝，把这份拷贝作为参数传递给函数。对于像i32，i64，f32，f64这些仅使用栈上资源的类型来说，浅拷贝与深拷贝无异。而需要使用堆资源的类型，如String，在进行浅拷贝时，不会复制堆上的资源，为了避免在释放资源时发生内存重复释放的问题，需要剥夺原变量的所有权。下面的MIR片段4-8和MIR片段4-9验证了前文。在语言实现的层面上，这一差异体现在i32，i64，f32，f64这些仅使用栈上资源的类型实现了Copy triat，而String等使用堆资源的类型没有也无法实现Copy triat。

//对应代码片段ch4\_8.rs

bb0: {

\_1 = <String as From<&str>>::from(const "Program mingInRust") -> bb1;

// let s = String::from("ProgrammingInRust");

}

bb1: {

\_5 = move \_1; // 浅拷贝s，所有权转移

\_4 = tail(move \_5) -> bb2; // tail(s);

}

MIR片段4-8

//对应代码片段ch4\_9.rs

bb0: {

\_1 = const 1\_i32; // let one = 1;

\_5 = const 1\_i32;

\_4 = tail(move \_5) -> bb1; // tail(one);

}

MIR片段4-9

2**. 可变的按值传递参数。**

如代码片段ch4-10.rs，在参数名称前加上mut关键字，可以让传递的参数在函数中可变。

*// 给参数s1拼接"Rust"并输出*

fn append(**mut** s1: String) {

    s1.extend(['R', 'u', 's', 't'].iter());

    println!("{}", s1);

}

fn main() {

**let** s1 = String::from("ProgrammingIn");

    append(s1);

}

代码片段ch4\_10.rs

按值传递参数的可变性与调用者提供的变量的可变性无关，函数中对这一类参数的修改也不会改变调用者的上下文。这是因为函数获得的参数是调用者提供的变量的浅拷贝，与调用者的上下文无关。

### 4.2.2 引用传递参数

1**. 不可变引用传递参数。**

如代码片段ch4\_11.rs，在参数类型前加上&符号实现不可变的引用传递。引用传递参数能够让调用者保持参数变量的所有权，在函数调用结束后继续使用该参数。

*// 取参数s串最后一位的u8编码*

fn tail(s: &String) -> u8 {

    if !s.is\_empty() {

        s.as\_bytes()[s.len() - 1]

    }

    else {

        0

    }

}

fn main() {

**let** s = String::from("ProgrammingInRust");

    assert\_eq!(tail(&s), 't' as u8);

*// 引用传递参数时，调用者能够保持参数变量的所有权*

    assert\_eq!(tail(&s), 't' as u8);

}

代码片段ch4\_11.rs

观察代码片段ch4\_11.rs对应的MIR片段4-11，可以发现，在按引用传递参数s时，会将调用者创建的引用进行浅拷贝，函数tail获得该引用拷贝的所有权。由于引用类型不占用堆空间的资源，故浅拷贝时并不会剥夺原变量的所有权。这样，原变量的所有权仍在调用者手中，调用者能够继续使用变量s及其引用。

bb0: {

\_1 = <String as From<&str>>::from(const "Programm ingInRust") -> bb1;

// let s = String::from("ProgrammingInRust");

}

bb1: {

\_6 = &\_1; // 创建s的一个不可变引用

\_5 = \_6; // 浅拷贝该引用

\_4 = tail(move \_5) -> [return: bb2, unwind: bb9];

// tail函数获得该引用拷贝的所有权

}

MIR片段4-11

2**. 可变引用传递参数。**

如代码片段ch4\_11.rs所示，在参数类型前加上&mut实现可变引用传递。

*// 给参数s1拼接"Rust"并输出*

fn append(s1: &**mut** String) {

    s1.extend(['R', 'u', 's', 't'].iter());

    println!("{}", s1);

}

fn main() {

**let** **mut** s1 = String::from("ProgrammingIn");

    append(&**mut** s1);

    println!("{}", s1);

}

可变引用的传递

带来副作用的&mut

### 4.2.3 生命周期参数

生命周期参数的用法与意义

生命周期参数的使用场景

生命周期参数的特殊使用情况

### 4.2.4 参数的模式匹配

通配符忽略参数

解构元组

### 4.2.5 Trait作为参数传递

## 4.3 函数调用

### 4.3.1 函数调用的基本语法

### 4.3.2 跨模块函数调用

### 4.3.3 外部函数接口绑定（FFI）

`extern`关键字的使用

FFI只能在unsafe代码块中使用

利用C实现变长参数列表

## 4.4 闭包

### 4.4.1 闭包简介与使用

闭包简介及使用

闭包的基本语法

闭包类型的自动推导

闭包的调用

每个闭包都有一个自己的类型

### 4.4.2 环境变量的捕获

Unboxed Closure

闭包的三个特性：Fn, FnOnce, FnMut

被捕获环境变量的所有权

move关键词

### 4.4.3 闭包作参数

### 4.4.4 闭包作返回值

## 4.5 发散函数

发散函数的定义与使用