# 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。

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，即复制这些变量在栈上占用的资源，然后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基本类型来说，浅拷贝相当于创建了一个新的变量。而对于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中调用tail()函数的部分

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中调用tail()函数的部分

bb0: {

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

\_5 = const 1\_i32; // 创建一个新的变量

\_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。这是因为函数参数是调用者提供的变量的浅拷贝，相当于使用let语句重新绑定一次变量，在完成绑定后参数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\_12.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\_rust(s1: &**mut** String) {

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

    println!("{}", s1);

}

fn main() {

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

    append\_rust(&**mut** s1);

*// 对可变引用参数的修改具有副作用*

    assert\_eq!(s1, String::from("ProgrammingInRust"));

}

代码片段ch4\_12.rs

在函数体中，对可变引用参数的修改也会同时改变调用者相关变量的值，产生副作用。

### 4.2.3函数签名中的生命周期参数

Rust编译器通过分析变量的生命周期来实现安全的内存管理，避免悬垂指针的产生。借用的出现让生命周期的分析变得复杂。Rust编译器中的借用检查器专门来检查借用相关的变量生命周期。为控制借用检查的设计和分析的复杂度，Rust的借用检查器不进行跨函数的借用检查，而以函数为粒度进行借用检查。当程序中出现跨函数的借用时，在一些情况下需要显示标注生命周期参数来帮助Rust借用检查器获得上下文借用信息。如代码片段ch4\_13.rs所示，函数签名中的<’a>是生命周期泛型参数声明。返回值和参数都标注了生命周期参数’a，表明返回值的生命周期长度不超过参数的生命周期长度，且参数的生命周期有重叠的部分。即：返回值的生命周期取v1，v2生命周期的交集，且v1，v2生命周期的交集非空。

*// 返回和最大的数组引用*

fn bigger\_sum<**'a**>(v1: &**'a** [i32], v2: &**'a** [i32]) -> &**'a** [i32] {

    if v1.iter().sum::<i32>() > v2.iter().sum() {

        v1

    }

    else {

        v2

    }

}

fn main() {

**let** v1 = [1, 2, 3]; *//\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**let** v; *//* |

*// scope               //* |

    { *//* |

**let** v2 = [1, 2]; *//\_\_\_\_\_       v1'*

        v = bigger\_sum(&v1, &v2); *//* |*--v'-*| |

*//     v2'* | |

        println!("{:?}", v); *//\_\_\_\_\_*|*-----*| |

    } *//* |

*// v2只在scope内有效 //\_\_\_\_\_\_\_\_\_\_\_\_* |

*// 尽管运行时的返回值是&v1，但是借用检查器将v的生命*

*// 周期取为v1和v2的交集，使得v也只在scope中有效*

*// 下面的这行代码将导致编译错误*

*// println!("{:?}", v);*

}

代码片段ch4\_13.rs

通常情况下，当函数的返回值是引用时，返回值的生命周期一定与参数相关，且相关的参数一定是引用。否则将会产生悬垂指针，这在Rust中是无法通过编译的。

如果函数的参数列表中有多个参数引用，就像代码片段ch4\_13.rs，在不显示标注生命周期参数的情况下，借用检查器将认为这些参数都与返回值有关，并且默认这些参数引用的生命周期各不相同，且没有交集。这是相当保守的静态检查。也许我们可以通过一些静态分析技术推断出返回值可能只和某个参数有关，这样就不需要去显式标注证明周期参数了。但在Rust的借用检查器真正使用了相关技术前，我们需要为安全性做出一点编程效率的牺牲，通过显式标注生命周期参数来辅助借用检查。

函数签名中也可以有多个显示标注的生命周期参数。如果不声明它们的关系，默认情况下编译器将认为它们没有交集。我们可以在生命周期参数声明中标注它们的关系。将代码片段ch4\_13.rs稍作修改，如代码片段ch4\_14.rs所示。

fn bigger\_sum<**'a**, **'b**:**'a**>(v1: &**'a** [i32], v2: &**'b** [i32]) -> &**'a** [i32] {

    if v1.iter().sum::<i32>() > v2.iter().sum() {

        v1

    }

    else {

        v2

    }

}

代码片段ch4\_14.rs

函数签名中的<’a, ‘b:’a>声明了两个生命周期参数’a和’b，生命周期关系’b:’a表示生命周期’a是’b的子集。返回值生命周期只能’a和’b中的较小者’a。如果返回值生命周期是’b，借用检查器将报错。

在函数定义中，并不是所有的跨函数借用行为需要标注生命周期参数。以下的两类情况不需要显示标注生命周期参数：

1. 参数列表中只有一个参数引用时，返回值引用的生命周期参数与该参数的一致；
2. 每一个参数引用都可以使用默认的生命周期参数，且返回值与之无关。

### 4.2.4 参数的模式匹配

在Rust中，函数参数是模式的一种，可以将匹配的值绑定到参数变量上。比较常见的应用场景是利用模式匹配解构元组、结构体、枚举类型、数组。代码片段ch4\_15提供了函数参数解构元组和结构体的示例。

*// 计算一个二维向量的模*

*// 模式匹配解构元组*

fn module\_i32((x, y): &(i32, i32)) -> f64 {

    return f64::sqrt((x.pow(2) + y.pow(2)) as f64)

}

**struct** Vec2I32{

    x: i32,

    y: i32

}

*// 模式匹配解构结构体*

fn module\_vec\_2\_i32(Vec2I32{x, y}: &Vec2I32) -> f64 {

    return f64::sqrt((x.pow(2) + y.pow(2)) as f64)

}

fn main() {

**let** v = (3, 4);

    assert\_eq!(module\_i32(&v), 5f64);

**let** v = Vec2I32{x: 3, y: 4};

    assert\_eq!(module\_vec\_2\_i32(&v), 5f64);

}

代码片段ch4\_15.rs

函数签名中还可以使用通配符来忽略在函数体中不会被使用到的参数，如代码片段ch4\_16.rs所示，对上面的代码片段ch4\_15.rs中的函数替换为了对一个二维向量x轴投影的实现，它们都将参数中的向量y分量忽略。

*// 计算一个二维向量的x轴投影*

*// 忽略y分量*

fn projection\_x((x, \_): &(i32, i32)) -> (i32, i32) {

    return (\*x, 0)

}

#[derive(PartialEq, Debug)]

**struct** Vec2I32{

    x: i32,

    y: i32

}

fn projection\_x\_vec\_2\_i32(Vec2I32{x, ..}: &Vec2I32) -> Vec2I32 {

    Vec2I32 { x: \*x, y: 0 }

}

fn main() {

**let** v = (3, 4);

    assert\_eq!(projection\_x(&v), (3, 0));

**let** v = Vec2I32{x: 3, y: 4};

    assert\_eq!(projection\_x\_vec\_2\_i32(&v), Vec2I32 { x: 3, y: 0 });

}

代码片段ch4\_16.rs

## 4.3 函数调用

### 4.3.1 函数调用的基本语法

函数定义所在的模块允许在任何位置调用该函数，即调用与定义的顺序无关。通过将函数名+参数列表构成一个表达式，我们可以调用一个已经被定义的函数。同时，我们可以用一个赋值语句接受函数的返回值。如代码片段ch4\_13.rs所示，main函数中调用了本模块的bigger\_some()函数，将v1和v2作为参数，让变量v接受返回值。

### 4.3.2 跨模块函数调用

在调用其他模块的函数时，我们需要先标识该函数所属的模块。我们可以在函数调用前用use语句声明模块路径，或者在调用函数时显示标注模块路径，如代码片段ch4\_17.rs所示。

fn main() {

**let** \_ = std::fs::File::create("ProgrammingInRust.txt");

    use std::fs::File;

**let** \_ = File::create("ProgrammingInRust.txt");

}

代码片段ch4\_17.rs

调用外部模块的函数的前提条件是，该外部模块函数的可见性是公开的。关于Rust的模块系统，本书的第八章有更详细的介绍。

### 4.3.3 外部函数接口绑定

外部函数接口（Foreign Function Interface）是指一种编程语言的程序所使用的其他编程语言的函数。所有的编程语言在计算机上都以二进制码的形式运行，为了能够让不同的编程语言调用其他编程语言的函数接口，并且具有一致的调用约定，应用程序二进制接口（Application Binary Interface）应运而生。应用程序二进制接口是在操作系统基础上形成的系统运行时必须遵守的变成约定，包括调用约定、命名规范、数据类型、虚拟地址空间布局、程序装载和动态链接等方面。

随着计算机技术的发展，出现了许多不同的应用程序二进制接口。Rust目前支持下面这些应用程序二进制接口：

* C
* Rust
* stdcall
* aapcs
* cdecl
* fastcall
* vectorcall
* rust-intrinsic
* system
* win64
* sysv64

我们以C语言为例，用静态链接的方式向Rust程序提供外部函数接口。首先，我们用C语言实现代码片段中ch4\_2.rs 的plus\_one\_i32函数，如代码片段add\_one.c所示：

**int** add\_one(**int** x) {

    return x + 1;

}

代码片段add\_one.c

这里我们选择在x86\_64-unknown-linux-gnu目标平台上创建C的静态库add\_one.lib，在终端中运行下面两条命令：

gcc -c add\_one.c

ar -rc libadd\_one.a add\_one.o

然后实现一个Rust程序，调用库的add\_one函数。如代码片段ch4\_18.rs所示， extern语句块完成对C语言实现的add\_one函数的外部函数接口绑定。Rust中所有的FFI都只能在unsafe代码块中被调用。

use std::ffi::c\_int;

#[link(name = "add\_one")]

**extern** {

    fn add\_one(x: c\_int) -> c\_int;

}

fn main() {

**let** x = 1;

    unsafe {

        assert\_eq!(add\_one(x), 2);

    }

}

代码片段ch4\_18.rs

最后用下面的命令调用rustc生成可执行文件。其中-L参数用于告知链接器链接库所在文件夹的路径。

rustc src/bin/ch4\_18.rs -o target/debug/ffi -L csrc/add\_one

利用FFI我们可以通过其他编程语言实现Rust中没有的语言特性。例如：C语言中的变长参数列表。代码片段sum\_pos\_int.c和ch4\_19.rs通过变长参数列表实现一个对正整数求和的FFI，并由Rust程序调用。

#include <stdarg.h>

*// 计算一系列正数的值。参数列表以非正数结束。*

**int** sum\_pos\_int(**int** x1, ...) {

    va\_list ap;

    va\_start(ap, x1);

**int** sum = x1;

**int** tmp = 0;

    while ((tmp = va\_arg(ap, **int**)) > 0)

        sum += tmp;

    va\_end(ap);

    return sum;

}

代码片段sum\_pos\_int.c

use std::ffi::{c\_int};

#[link(name = "sum\_pos\_int")]

**extern** {

    fn sum\_pos\_int(x1: c\_int, ...) -> c\_int;

}

fn main() {

**let** x1 = 1;

**let** x2 = 2;

**let** x3 = 0;

    unsafe {

        assert\_eq!(sum\_pos\_int(x1, x2, x3), 3);

    }

}

代码片段ch4\_19.rs

## 4.4 闭包

### 4.4.1 闭包简介与使用

在Rust中，闭包（Closure）是不通过参数就能捕获自身（词法作用域）外部环境变量的函数。闭包可以作为变量的值，作为函数参数和函数返回值。闭包的概念最初在λ演算中提出，而λ演算是函数式编程的基础。闭包赋予了Rust能够实现比函数更高层次的代码复用能力和多样化的行为。

一个闭包的定义由|参数列表|->返回值类型{语句块}构成。闭包在不作为函数的参数或返回值时，其定义中的参数类型和返回值类型可以省略，编译器将根据闭包被调用的上下文自动推导它们的类型。闭包的语句块只有一行时，如果没有显示标注返回值类型，可以进一步省略语句块的花括号。

闭包的调用语法与函数的调用语法相同。代码片段ch4\_20.rs提供了闭包定义和调用的一个示例，其中定义了一个返回值是传入参数自增1的闭包。在定义时代码片段ch4\_20.rs并没有给出闭包incr的参数类型和返回值类型，编译器根据最近的上下文中的调用推导出闭包的类型。闭包类型的自动推导只能进行一次，故闭包的类型在推导确定后不能被再次改变。

fn main() {

**let** incr = |x|

        x + 1;

*// 显示标注参数类型和返回值类型*

**let** incr\_same = |x: i32| -> i32 {

        x + 1

    };

    assert\_eq!(incr(1), 2);

    assert\_eq!(incr\_same(1), 2);

*// 闭包的类型在推导确定后不能再改变*

*// assert\_eq!(incr(1.0f64), 2.0f64);*

}

代码片段ch4\_20.rs

闭包的类型按名称等价。即每一个不同名的闭包类型不一，即使像代码片段ch4\_20.rs中的闭包incr和incr\_same，它们拥有同样的函数签名和内部实现，但是Rust编译器认为它们的类型不同。在MIR片段4-20中，可以看到Rust编译器为这incr和incr\_same的类型标注中用代码位置区分二者的类型，并分别为它们生成了一段MIR。

fn main() -> () {

…

bb0: {

…

// 调用incr(1)

\_5 = <[closure@[src/main.rs:2:16: 2:19](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)] as Fn<(i32,)>>::call(move \_6, move \_7) -> bb1;

}

…

bb3: {

…

// 调用incr\_same(1)

\_25 = <[closure@[src/main.rs:7:21: 7:36](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)] as Fn<(i32,)>>::call(move \_26, move \_27) -> bb4;

}

…

}

// incr和incr\_same各自有一段MIR代码

fn main::{closure#0}(\_1: &[closure@[src/main.rs:4:16: 4:19](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)], \_2: i32) -> i32 {

//…

}

fn main::{closure#1}(\_1: &[closure@[src/main.rs:8:21: 8:36](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)], \_2: i32) -> i32 {

//…

}

MIR片段4-20

闭包在Rust中的实现是基于实现了Fn, FnMut, FnOnce三种trait中一种的匿名结构体。在本书正式介绍trait的有关内容之前，在这里我们只需要知道trait用于进行类型行为的抽象。代码片段ch4\_21.rs是Rust对Fn, FnMut, FnOnce三种trait的相关定义。根据定义，这三个trait间的集合关系满足Fn ⸧ FnMut ⸧ FnOnce。下面将通过闭包对环境变量的捕获介绍这三种trait。

pub **trait** FnOnce<Args> {

*/// The returned type after the call operator is used.*

    #[lang = "fn\_once\_output"]

    #[stable(feature = "fn\_once\_output", since = "1.12.0")]

**type** Output;

*/// Performs the call operation.*

    #[unstable(feature = "fn\_traits", issue = "29625")]

**extern** "rust-call" fn call\_once(self, args: Args) -> Self::Output;

}

pub **trait** FnMut<Args>: FnOnce<Args> {

*/// Performs the call operation.*

    #[unstable(feature = "fn\_traits", issue = "29625")]

**extern** "rust-call" fn call\_mut(&**mut** self, args: Args) -> Self::Output;

}

pub **trait** Fn<Args>: FnMut<Args> {

*/// Performs the call operation.*

    #[unstable(feature = "fn\_traits", issue = "29625")]

**extern** "rust-call" fn call(&self, args: Args) -> Self::Output;

}

代码片段ch4\_21.rs

### 4.4.2 环境变量的捕获

闭包能够捕获环境中已经定义的变量，并在其语句块中使用。闭包可以通过引用，可变引用和转移所有权这三种方式捕获环境变量。

**1. 引用捕获环境变量。**

闭包如果捕获了复制语义类型的环境变量，并且闭包内没有修改该变量的值，闭包将通过不可变引用形式使用环境变量。代码片段ch4\_21.rs中的闭包捕获环境变量i，并将其与参数x的和作为闭包的返回值。

fn main() {

**let** x = 1;

**let** i = 10;

**let** incr = |x|

        x + i;

    assert\_eq!(incr(x), 11);

}

代码片段ch4\_22.rs

观察代码片段ch4\_21.rs对应的MIR片段4-21，在调用闭包incr前，MIR创建了一个变量i的不可变引用\_3 = &i。在调用闭包时，被捕获的环境变量和参数分别通过两个元组\_8和\_9作为参数供闭包使用。

fn main() -> () {

//…

bb0: {

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

\_2 = const 10\_i32; //let i = 10;

\_4 = &\_2; //创建变量i的不可变引用

Deinit(\_3);

(\_3.0: &i32) = move \_4; //创建被捕获变量的元组(&x)

\_8 = &\_3;

Deinit(\_9);

(\_9.0: i32) = const 1\_i32;//创建参数的元组(i)

\_7 = <[closure@[...](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)] as Fn<(i32,)>>::call(move \_8, move \_9) -> bb1; //调用闭包incr，实现Fn trait

//…

}

MIR片段4-22

从MIR片段4-21中的闭包incr调用可以看出，编译器为闭包incr自动实现了Fn trait。如果闭包通过不可变引用的方式捕获环境变量，那么编译器会为其实现Fn trait。

**2.可变引用捕获环境变量。**

当闭包捕获了复制语义类型的环境变量，并在闭包内部修改了该变量的值，编译器将会创建该环境变量的一个可变引用供闭包使用。该可变引用的词法作用域在闭包定义处开始，在最后一处闭包调用结束。根据所有权机制，在该可变引用的此法作用域内，都将无法在闭包外部使用该变量。如果该闭包作为当前作用域所属函数的返回值，那么在闭包定义后的代码块都将无法继续使用被捕获的变量。

代码片段ch4\_22.rs定义的闭包incr将捕获的环境变量i和参数x相加，并将运算结果保存到i的值中。

fn main() {

**let** x = 1;

**let** **mut** i = 10;

**let** **mut** incr = |x| //--+

        i += x; // |

// | 被捕获变量i的可变借用的词法作用域

//*error: cannot assign to i because it is borrowed*

// *i = 1;* |

// |

    incr(x); //----------+

    assert\_eq!(i, 11);

    i = 1; *// ok*

    assert\_eq!(i, 1);

}

代码片段ch4\_23.rs

根据MIR片段4-22，闭包incr通过可变引用访问和修改环境变量i的值。编译器为该闭包自动实现了FnMut trait。如果闭包通过不可变引用的方式捕获环境变量，那么编译器会为其实现FnMut trait。

fn main() -> () {

//…

bb0: {

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

\_2 = const 10\_i32; //let i = 10;

\_4 = &mut \_2; //创建变量i的可变引用

Deinit(\_3);

(\_3.0: &mut i32) = move \_4; //创建被捕获变量的元组(&x)

\_6 = &mut \_3;

Deinit(\_7);

(\_7.0: i32) = const 1\_i32; //创建参数的元组(i)

\_5 = <[closure@[...](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)] as FnMut<(i32,)>>::call\_mut (move \_6, move \_7) -> bb1; //调用闭包incr

}

//…

}

MIR片段4-23

**3. 转移所有权捕获环境变量。**

对于移动语义类型的变量，闭包在定义时将获取其所有权。

代码片段ch4\_23.rs中定义了一个对环境变量s1: String, s2 :String求较大者的闭包max\_string。根据闭包中调用的max方法的参数列表(self, Other: Self)可以知道，max方法按值传递获得参数，即需要获取s1，s2的所有权。故闭包max\_string在被定义时将获取环境变量s1和s2的所有权。那么，在闭包max\_string定义后，因为main函数不再持有s1，s2的所有权，我们将无法对s1,s2进行其他操作。

fn main() {

**let** s1 = "ProgrammingInRust".to\_string();

**let** s2 = "RustProgrammingLanguage".to\_string();

**let** bigger\_string = || {

        s1.max(s2)

    };

    assert\_eq!(

        bigger\_string(),

        "RustProgrammingLanguage".to\_string()

);

*// error: borrow of moved value "s1"*

*// println!("{:?}", s1);*

*// error: borrow of moved value "s2";*

*// s2.as\_str();*

*// error: closure cannot be invoked more than once because*

*// it moves the variable `s1` out of its environment.*

*// assert\_eq!(*

*//     max\_string(),*

*//     "RustProgrammingLanguage".to\_string()*

*// );*

}

代码片段ch4\_24.rs

观察MIR片段4-23可以验证上文，在闭包调用时直接通过move操作将变量s1和s2移入闭包，闭包获得变量s1和s2的所有权。闭包max\_string也无法被再次调用，因为调用时变量s1和s2的所有权就被移入了该闭包，在第二次调用时将无法重新获取s1和s2的所有权。编译器为闭包max\_string自动实现了FnOnce trait。如果闭包通过转移所有权的方式捕获环境变量，那么编译器会为其实现FnOnce trait。

bb1: {

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

\_1 = slice::<impl [i32]>::into\_vec::<std::alloc::Global>(move \_2) -> bb2;

}

bb2: {

//…

(\_8.0: std::vec::Vec<i32>) = move \_1; //转移v所有权

\_13 = move \_8;

\_12 = <[closure@[...](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)] as FnOnce<()>>::call\_once(move \_13, move \_14) -> bb3;

}

MIR片段4-24

1. **move对闭包捕获环境变量的影响。**

闭包在定义时可以使用move关键字强制获取被捕获变量的所有权。

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

发散函数是一种特殊的函数，在执行时不返回，也不继续向下执行官，而是直接结束当前进程。当程序在运行中遇到一些不可恢复的错误状态时，可以通过调用发散函数完成结束程序运行前的收尾工作和错误信息打印，然后结束程序。如代码片段ch4\_x.rs。

fn kill() -> ! {

eprintln!("a < 0");

    panic!("killed in diverging function")

}

fn main() {

**let** a = -1;

    if a < 0 {

        kill();

    }

}

代码片段ch4\_x.rs

发散函数的存在完善了Rust的类型系统，使得Rust函数的返回值类型不仅包括正常值，也包括了异常值。

## 4.6 小结