第 5 章 函　　数

函数作为程序逻辑的载体，是仓颉中一个非常重要的概念，本章将讲解函数最基本的一些特性，更复杂的特性会在后续章节介绍。

5.1　函数的定义


6min

典型的函数定义，示例代码如下：

```
func add(a: Int64, b: Int64): Int64 {
    return a + b
}
```

从这个示例里可以归纳出函数定义的基本规则如下。

(1) 使用关键字 func 标识函数，表示函数定义的开始。

(2) func 关键字后是函数的名称，名称可以是任意的合法标识符。

(3) 在小括号内是可选的参数列表。

(4) 小括号后是可选的函数返回值类型。

(5) 大括号内是函数体。

上面的定义表明，这是一个名称为 add 的函数，参数列表为两个 Int64 类型的参数 a 和 b，该函数的返回值类型为 Int64 类型，在函数体中把 a 和 b 相加后返回。

说明：从 0.28.4 版本开始，main 被认为是一个关键字，仓颉中的 main 函数被当作一种特殊的函数，定义 main 函数时前面不再需要 func 关键字。

5.2　参数及函数调用


10min

1. 形参、实参及函数调用

函数的参数列表是可选的，一个函数可以没有参数，也可以拥有 1 个或者多个参数，这里的参数又被称为形式参数，简称形参，在函数调用时，传入的实际参数被称为实参。形参

在定义时,可以被认为使用了 let 修饰,也就是说在函数体内不能修改形参的值,否则编译时可能会提示 error: cannot assign to value which is an initialized 'let' constant 的错误信息。函数调用的示例代码如下:

```
//Chapter5/call_demo.cj

func add(param1: Int64,param2: Int64): Int64 {     //这里 param1 和 param2 为函数的形参
    return param1 + param2
}

main() {
    var a = 1
    var b = 2
    let c = add(a, b)                              //这里 a 和 b 为函数的实参
    println(c)
}
```

编译后运行该示例,输出如下:

```
3
```

2. 非命名参数及命名参数

形参分为两类,分别是非命名参数和命名参数。非命名参数的定义方式为 p:T,其中,p 是参数名称,T 是参数类型,中间使用冒号分隔。以示例 call_demo.cj 源码中定义的 add 函数为例,该函数使用的就是非命名参数,非命名参数不能设置默认值。

命名参数的定义方式为 p!:T,其中,p 是参数名称,T 是参数类型,中间使用叹号加冒号分隔,命名参数还可以设置默认值,示例代码如下:

```
func add(param1!: Int64, param2!: Int64 = 2): Int64 {
    return param1 + param2
}
```

在调用时,命名参数的实参需要使用 p:e 的形式,其中,p 是命名参数名称,e 是实参。对于有多个命名参数的函数,调用时的传参顺序可以和定义时的参数顺序不一致;如果命名参数设置了默认值,则在调用时该参数可以不传实参,在函数体内会使用该参数的默认值作为实参的值,如果传递了实参,则实参的值将会覆盖默认值。参数列表可以同时包括非命名参数和命名参数,当两者共存时,要先定义非命名参数,后定义命名参数,示例代码如下:

```
//Chapter5/multi_params_demo.cj

func lunch(name: String,drink!: String = "牛奶",fruits!: String = "苹果",
    stapleFood!: String = "米饭",count!: Int64 = 1,takeOut!: Bool = false): String {
    return "Name: ${name} Drink: ${drink} Fruits: ${fruits} StapleFood: ${stapleFood} Count:
 ${count} TakeOut: ${takeOut}"
}

main() {
```

```
    var menu = lunch("小王",drink: "牛奶",fruits: "苹果",stapleFood: "米饭",count: 1,
takeOut: false)              //传递所有参数的调用示例
    println(menu)

    menu = lunch("小王")     //有默认值的命名参数可以不用传参,使用默认值作为参数值
    println(menu)

    menu = lunch("小张",stapleFood: "面条",drink: "咖啡")
                             //命名参数的传参顺序可以和定义时的顺序不一致
    println(menu)
}
```

该示例包括两个函数,一个是 main 函数,另一个是 lunch 函数,lunch 函数包括 6 个参数,第 1 个是非命名参数,其余的参数均为有默认值的命名参数。在 main 函数里对 lunch 函数进行了 3 次调用,演示了命名参数的使用场景。假如 lunch 全部使用非命名参数,那么每次调用 lunch 函数时都需要把 6 个参数写全,并且要保证顺序一致,这样写起来比较烦琐,而且容易出错;如果使用了命名参数,就不用按照顺序传参,传参时指明了参数名称,不容易出错,更重要的是可以利用默认参数,只需把不是默认值的参数传递过去,提高了开发效率。编译后运行该示例,输出如下:

```
Name:小王 Drink:牛奶 Fruits:苹果 StapleFood:米饭 Count:1 TakeOut:false
Name:小王 Drink:牛奶 Fruits:苹果 StapleFood:米饭 Count:1 TakeOut:false
Name:小张 Drink:咖啡 Fruits:苹果 StapleFood:面条 Count:1 TakeOut:false
```

5.3 返回值类型

函数被调用后得到的值的类型是返回值类型,在函数定义时可以显式指定返回值类型,也可以不指定返回值类型,而是由编译器推导确定。在显式定义返回值类型时,要求函数体的类型、函数体中所有 return e 表达式中 e 的类型是返回值类型的子类型。如果定义的返回值类型和实际类型不一致,编译器则会提示错误 type of function body is incompatible with return type,错误示例代码如下:

```
func add(param1: Int64,param2: Int64): Int64 {
    return (param1,param2)
}
```

在未显式指定返回值类型时,编译器会根据函数体的类型及函数体中所有的 return 表达式来共同推导出函数的返回值类型,如下例所示,会推导出返回类型为 Int64 类型:

```
func add(param1: Int64,param2: Int64) {
    return param1 + param2
}
```

如果编译器不能推导出返回值类型,或者推导出的返回值类型不一致(例如多个 return e

表达式中的 e 的类型不一致),编译器则会报错。

5.4 函数体

1. 函数体的类型

函数体本身也有类型,它的类型是函数体中最后一项的类型,若最后一项为表达式,则函数体的类型是此表达式的类型,若最后一项为变量定义、函数声明或函数体为空,则函数体的类型为 Unit,示例代码如下:

```
func add(param1: Int64,param2: Int64): Int64 {
   param1 + param2
}
```

在这个实例中,最后一项是两个 Int64 类型相加的表达式,这个表达式是 Int64 类型的,也表示函数体是 Int64 类型的,这样就和定义的返回值类型匹配起来了。

2. return 表达式

在函数体中,执行到 return 表达式时,会立刻终止函数的执行并返回,返回的类型需要和函数定义的返回值类型一致。return 表达式有两种形式,一种是 return expr,expr 为表达式,此时要求 expr 的类型和函数定义的返回值类型一致;另一种是 return,等价于 return(),因为()是 Unit 的字面量,所以要求函数定义的返回值类型是 Unit。

3. 函数局部变量对全局变量的遮盖

定义在源文件顶层的变量是全局变量,定义在函数内部的变量是局部变量,如果函数定义在全局变量的定义后面,则全局变量的作用范围也包括这个函数内部。这时,如果在函数内部定义了一个和全局变量同名的局部变量,局部变量就会对全局变量形成遮盖,也就是从同名局部变量定义开始,到函数的结束,用到这个变量的地方,都会使用局部变量,示例代码如下:

```
//Chapter5/local_var_demo.cj

var gloablVar: Int64 = 1

main() {
    println(gloablVar)
    var gloablVar: Int64 = 5
    println(gloablVar)
}
```

编译后运行该示例,输出如下:

```
1
5
```

从示例可以看出,第 1 次打印时还没有定义局部变量,所以打印出的是全局变量的值,

第 2 次打印时，因为已经定义了局部变量，对同名全局变量形成了遮盖，所以打印的是局部变量的值。

5.5 嵌套函数（局部函数）

在源文件顶层定义的函数称为全局函数，在函数内部也可以定义函数，称为嵌套函数或者局部函数，局部函数的作用范围从局部函数定义开始到外层函数的结束，示例代码如下：

```
//Chapter5/nested_func_demo.cj

main() {
    //嵌套函数 incNum
    func incNum(num: Int64) {
        return num + 1
    }

    var count: Int64 = 5
    count = incNum(count)
    println(count)
}
```

编译后运行该示例，输出如下：

```
6
```

类似局部变量对全局变量的遮盖，局部函数的变量对外层函数的变量及全局变量也构成遮盖关系，示例代码如下：

```
//Chapter5/local_func_var_demo.cj

//定义全局变量 gloablVar
var gloablVar: Int64 = 1

main() {
    println(gloablVar)
    //局部变量 gloablVar 遮盖了全局变量 gloablVar
    var gloablVar: Int64 = 5

    func localFunc() {
        println(gloablVar)
        //嵌套函数 localFunc 中的变量 gloablVar 遮盖了上层的同名变量
        var gloablVar: Int64 = 15
        println(gloablVar)
    }

    localFunc()
}
```

编译后运行该示例,输出如下:

```
1
5
15
```

在这个示例中,一共有 3 个打印语句,第 1 次打印时,只定义了全局变量,所以打印出 1;第 2 次打印时,已经定义了全局函数内的同名局部变量,打印的是该局部变量,所以打印出 5;第 3 次打印时,在局部函数里也定义了同名变量,形成了对外层变量的遮盖,所以打印出 15。