1. 定义函数：

带有两个Int参数，返回Int的函数：

fun sum(a: Int, b: Int): Int {

return a + b

}

将表达式作为函数体，返回值类型自动推断的函数：

fun sum(a: Int, b: Int) = a + b

函数返回无意义的值：

fun printSum(a: Int, b: Int): Unit {

println("sum of $a and $b is ${a + b}")

}

Unit返回类型可以省略：

fun printSum(a: Int, b: Int) {

println("sum of $a and $b is ${a + b}")

}

1. 定义变量：

一次赋值（只读）的局部变量：

val a: Int = 1 // 立即赋值

val b = 2 // 自动推断出 `Int` 类型

val c: Int // 如果没有初始值类型不能省略

c = 3 // 明确赋值

可变变量：

var x = 5 // 自动推断出 `Int` 类型

x += 1

1. 注释：

// 这是一个行注释

/\* 这是一个多行的 块注释。 \*/

1. 使用字符串模板：

var a = 1

// 模板中的简单名称：

val s1 = "a is $a"

a = 2

// 模板中的任意表达式：

val s2 = "${s1.replace("is", "was")}, but now is $a"

1. 使用条件表达式

在kotlin中，if是一个表达式，即它会返回一个值，因此就不需要三元运算符

// 传统用法

**var** max = a **if** (a < b) max = b

// With else

**var** max: Int **if** (a > b) {

max = a

} **else** { max = b }

// 作为表达式

**val** max = **if** (a > b) a **else** b

If分支可以是代码块，最后的表达式作为该块的值

**val** max = **if** (a > b) {

print("Choose a")

a

} **else** { print("Choose b")

b

}

注意：如果使用if作为表达式而不是语句（返回它的值或它赋给变量），该表达式需要有else分支

1. 使用空值及null检测：

当某个变量的值可以为null的时候，必须在声明处的类型后添加？来标识该引用可为空

如果str的内容不是数字返回null

**fun** parseInt(str: String): Int? { // …… }

判断是否为空，也可以使用str！=null

1. 使用类型检测及自动类型转换

is运算符检测一个表达式是否某类型的一个实例。如果一个不可变的布局变量或属性已经判断为某类型，那么检测后的分支中可以直接当做该类型使用，无需显示转换:

fun getStringLength(obj: Any): Int? {

if (obj is String) {

// `obj` 在该条件分支内自动转换成 `String`

return obj.length

}

// 在离开类型检测分支后，`obj` 仍然是 `Any` 类型

return null

}

还可以：

fun getStringLength(obj: Any): Int? {

if (obj !is String) return null

// `obj` 在这一分支自动转换为 `String`

return obj.length

}

或者：

fun getStringLength(obj: Any): Int? {

// `obj` 在 `&&` 右边自动转换成 `String` 类型

if (obj is String && obj.length > 0) {

return obj.length

}

return null

}

1. 使用for循环：

For循环可以对任何提供迭代器的对象进行遍历，相当于像C#中的foreach循环：

**for** (item **in** collection) print(item)

循环体可以是一个代码块：

**for** (item: Int **in** ints) {

// ……

}

For可以循环遍历任何提供了迭代器的对象。即：

- 有一个成员函数或者扩展函数interator（）

- 有一个成员函数或者扩展函数next（）

- 有一个成员函数或者扩展函数hasNext（）返回boolean

这三个函数都需要标记为operator

通过索引遍历一个数组或者一个list：

**for** (i **in** array.indices) {

print(array[i])

}

也可以使用库函数withIndex：

**for** ((index, value) **in** array.withIndex()) {

println("the element at $index is $value")

}

val items = listOf("apple", "banana", "kiwi")

for (item in items) {

println(item)

}

或者：

val items = listOf("apple", "banana", "kiwi")

for (index in items.indices) {

println("item at $index is ${items[index]}")

}

1. 使用while循环：

和java的用法一样

val items = listOf("apple", "banana", "kiwi")

var index = 0

while (index < items.size) {

println("item at $index is ${items[index]}")

index++

}

1. 使用when表达式：

When取代了类C余姚的switch操作符：

**when** (x) {

1 -> print("x == 1")

2 -> print("x == 2")

**else** -> {

// 注意这个块 print("x is neither 1 nor 2")

}

}

when将它的参数和所有的分支条件顺序比较，直到某个分支满足条件。When既可以被当做表达式使用也可以被当做语句使用。如果它被当做表达式，符合条件的分支的值就是整个表达式的值。如果当做语句使用，则忽略个别分支的值

如果其他分支都不满足条件将会求值else分支。如果when作为一个表达式使用，则必须有else分支，除非编译器能够检测出所有的可能情况都已经覆盖了。

多个条件分支放在一起：

**when** (x) {

0, 1 -> print("x == 0 or x == 1")

**else** -> print("otherwise")

}

用任意表达式作为分支条件：

**when** (x) {

parseInt(s) -> print("s encodes x")

**else** -> print("s does not encode x")

}

也可以检测一个值在（in）或者不在（！in）一个区间或者集合中：

**when** (x) {

**in** 1..10 -> print("x is in the range")

**in** validNumbers -> print("x is valid")

!**in** 10..20 -> print("x is outside the range")

**else** -> print("none of the above")

}

检测一个值是（is）或者不是（！is）一个特定类型的值。

注意：由于智能转换，可以访问该类型的方法和属性而无需任何额外的检测

**fun** hasPrefix(x: Any) = **when**(x) {

**is** String -> x.startsWith("prefix")

**else** -> **false**

}

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

1. 使用区间（range）

使用in运算符来检测某个数字是否在指定区间内：

val x = 10

val y = 9

if (x in 1..y+1) {

println("fits in range")

}

检测某个数字是否在指定区间外：

val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {

println("-1 is out of range")

}

if (list.size !in list.indices) {

println("list size is out of valid list indices range too")

}

区间迭代：

for (x in 1..5) {

print(x)

}

数列迭代：

for (x in 1..10 step 2) {

print(x)

}

println()

for (x in 9 downTo 0 step 3) {

print(x)

}

1. 创建基本类及其实例：

val rectangle = Rectangle(5.0, 2.0) // 不需要“new”关键字

val triangle = Triangle(3.0, 4.0, 5.0)

基本数据类型：

在kotlin中 ==比较值，====比较值和地址（int要在128以上才会比较地址）

这是完整的位运算列表（只用于 Int 和 Long）：

* shl(bits) – 有符号左移 (Java 的 <<)
* shr(bits) – 有符号右移 (Java 的 >>)
* ushr(bits) – 无符号右移 (Java 的 >>>)
* and(bits) – 位与
* or(bits) – 位或
* xor(bits) – 位异或
* inv() – 位非

字符用char类型表示，它们不能直接当做数字

**fun** check(c: Char) { **if** (c == 1) { // 错误：类型不兼容 // …… } }

支持这几个转义序列：\t、 \b、\n、\r、\'、\"、\\ 和 \$。 编码其他字符要用 Unicode 转义序列语法：'\uFF00'。

数组在kotlin中使用Array类来表示，它定义了get和set函数（按照运算符重载约定这会转变为[]）和size属性

**class** Array<T> **private** **constructor**() {

**val** size:

Int **operator** **fun** **get**(index: Int): T

**operator** **fun** **set**(index: Int, value: T): Unit

**operator** **fun** iterator(): Iterator<T>

// …… }

可以使用库函数arrayof（）创建一个数组并传递元素值给它

Arrayof（1，2,3）创建了array[1,2,3]

或者可以用arrayOfNulls（）创建一个指定大小，所有元素为空的数组

还可以用Array构造函数，用作参数的函数能够返回给定索引的每个元素的初始值：

// 创建一个 Array<String> 初始化为 ["0", "1", "4", "9", "16"]

**val** asc = Array(5, { i -> (i \* i).toString() })

[]运算符代表调用成员函数get（）和set（）

注意：与java不同的是，kotlin中数组是不型变的，这意味着kotlin不让我们把Array<String>赋值给Array<Any>，以防止可能的运行失败（但是可以使用Array<out Any>）

Kotlin也有无装箱开销的专门的类来表示原生类型数组：ByteArray，ShortArray，IntArray等等。这些类和Array并没有继承关系，但是它们有同样的方法属性集，也有相应的工厂方法：

**val** x: IntArray = intArrayOf(1, 2, 3)

x[0] = x[1] + x[2]

字符串：

字符串用String类型表示，字符串是不可变的，还是可以用s[i]

字符串字面值：

Kotlin有两种类型的字符串字面值：转移字符串可以有转义字符，以及原生字符串可以包含换行和任意文本。

转义字符串：

**val** s = "Hello, world!\n"

原生字符串使用三个引号(“””)分界符括起来，内部没有转义并且可以包含换行和其他字符

**val** text = """

for (c in "foo") print(c)

"""

可以通过trimMargin（）函数去除前导空格：

**val** text = """

|Tell me and I forget.

|Teach me and

I remember.

|Involve me and I learn.

|(Benjamin Franklin)

""".trimMargin()

默认|用作边界前缀，我们可以选择其他字符串并作为参数传入，比如：trimMargin（“>”）

字符串模板

**val** i = 10 **val** s = "i = $i" // 求值结果为 "i = 10"

或者用花括号括起来的任意表达式:

**val** s = "abc"

**val** str = "$s.length is ${s.length}" // 求值结果为 "abc.length is 3"

原生字符串和转义字符串内部都支持模板。 如果你需要在原生字符串中表示字面值 $ 字符（它不支持反斜杠转义），你可以用下列语法：

**val** price = """

${'$'}9.99

"""

返回和跳转：

Kotlin有三种结构化跳转表达式：

- return。

- break

- continue

在Kotlin中任何表达式都可以用标签（label）来标记。标签的格式为标识符后跟@符号，例如：abc@

要为一个表达式加标签，只要在其前面加标签即可：

loop@ **for** (i **in** 1..100) {

// ……

}

现在，我们可以用标签限制 break 或者continue：

loop@ **for** (i **in** 1..100) {

**for** (j **in** 1..100) {

**if** (……) **break**@loop

}

}

标签限制的break跳转到刚好位于该标签指定的循环后面的执行点。

Continue继续标签指定的循环的下一次迭代。

标签处返回：

Kotlin有函数字面量，布局函数和对象表达式。因袭Kotlin的函数可以被嵌套。标签限制的return允许我们从外层函数返回

**fun** foo() {

ints.forEach {

**if** (it == 0) **return** // nonlocal return from inside lambda directly to the caller of foo()

print(it)

}

}

这个 return 表达式从最直接包围它的函数即 foo 中返回。 （注意，这种非局部的返回只支持传给[内联函数](http://www.kotlincn.net/docs/reference/inline-functions.html)的 lambda 表达式。） 如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return。

**fun** foo() {

ints.forEach lit@{

**if** (it == 0) **return**@lit

print(it)

}

}

现在，它只会从 lambda 表达式中返回。通常情况下使用隐式标签更方便。 该标签与接受该 lambda 的函数同名。

**fun** foo() {

ints.forEach {

**if** (it == 0) **return**@forEach

print(it)

}

}

或者用一个匿名函数替代lambda表达式。匿名函数内部的return语句将从该匿名函数自身返回。

**fun** foo() {

ints.forEach(**fun**(value: Int) {

**if** (value == 0) **return** // local return to the caller of the anonymous fun, i.e. the forEach loop

print(value)

})

}

当要返一个回值的时候，解析器优先选用标签限制的 return，即

**return**@a 1

意为“从标签 @a 返回 1”，而不是“返回一个标签标注的表达式 (@a 1)”。