彻底掌握Kotlin

字符串操作

数字类型

标准库函数

A

B

C

字符串操作

substring

- 字符串截取，substring函数支持IntRange类型（表示一个整数范围的类型）的参数，until创建的范围不包括上限值。

```
1   const val NAME = "Jimmy's friend"
2   fun main() {
3       val index: Int = NAME.indexOf( char: '\'')
4       //NAME.substring(0,index)
5       val str: String = NAME.substring( range: 0 until index)
6       println(str)
7   }
```

split

- split函数返回的是List集合数据，List集合又支持解构语法特性，它允许你在一个表达式里给多个变量赋值，解构常用来简化变量的赋值。

```
1   const val NAMES = "jack,jacky,jason"
2   fun main() {
3       val data:List<String> = NAMES.split(...delimiters: ',')
4       //data[0]
5       val (origin:String, dest:String, proxy:String) = NAMES.split(...delimiters: ',')
6       println("$origin $dest $proxy")
7   }
```

replace

- 字符串替换

```
fun main() {
    //加密替换一个字符串
    val str1 = "The people's Republic of China."
    //第一个参数是正则表达式，用来决定要替换哪些字符
    //第二个参数是匿名函数，用来确定该如何替换正则表达式搜索到的字符
    val str2 : String = str1.replace(Regex( pattern: "[aeiou]")){
        when(it.value){
            "a" -> "8" ^replace
            "e" -> "6" ^replace
            "i" -> "9" ^replace
            "o" -> "1" ^replace
            "u" -> "3" ^replace
            else -> it.value ^replace
        }
    }
    println(str1)
    println(str2)
}
```

字符串比较

- 在Kotlin中，用==检查两个字符串中的字符是否匹配，用===检查两个变量是否指向内存堆上同一对象，而在Java中==做引用比较，做结构比较时用equals方法。

```
1  fun main() {
2      val str1 = "Jason"
3      val str2 : String = "jason".capitalize()
4
5      println("$str1,$str2")
6      println(str1 == str2)
7      println(str1 === str2)
8  }
```

forEach

- 遍历字符

```
1  fun main() {
2      "The people's Republic of China.".forEach {
3          print("$it*")
4      }
5  }
```

数字类型

安全转换函数

- Kotlin提供了toDoubleOrNull和toIntOrNulll这样的安全转换函数，如果数值不能正确转换，与其触发异常不如干脆返回null值。

```
1  fun main() {
2      //抛出异常
3      //val number1:Int = "8.98".toInt()
4      val number1:Int? = "8.98".toIntOrNull()
5      println(number1)
6  }
```

Double类型格式化

- 格式化字符串是一串特殊字符，它决定该如何格式化数据。

```
1  fun main() {
2      val s : String = "%.2f".format( ...args: 8.956756)
3      println(s)
4  }
```

Double转Int

- 精度损失与四舍五入

```
1  import kotlin.math.roundToInt
2
3  fun main() {
4      //精度损失
5      println(8.956756.toInt())
6      //四舍五入
7      println(8.956756.roundToInt())
8  }
```

标准库函数

apply

- apply函数可看作一个配置函数，你可以传入一个接收者，然后调用一系列函数来配置它以便使用，如果提供lambda给apply函数执行，它会返回配置好的接收者。

```
1  import java.io.File
2
3  fun main() {
4      //配置一个File实例
5      val file1 = File( pathname: "E://i have a dream_copy.txt")
6      file1.setReadable(true)
7      file1.setWritable(true)
8      file1.setExecutable(false)
9      //使用apply
10     val file2 : File = File( pathname: "E://i have a dream_copy.txt").apply {
11         setReadable(true)
12         setWritable(true)
13         setExecutable(false)
14     }
15 }
```

apply

- 可以看到，调用一个个函数类配置接收者时，变量名就省掉了，这是因为，在lambda表达式里，apply能让每个配置函数都作用于接收者，这种行为有时又叫做相关作用域，因为lambda表达式里的所有函数调用都是针对接收者的，或者说，它们是针对接收者的隐式调用。

let

- let函数能使某个变量作用于其lambda表达式里，让it关键字能引用它。let与apply比较，let会把接收者传给lambda，而apply什么都不传，匿名函数执行完，apply会返回当前接收者，而let会返回lambda的最后一行。

```
fun main() {
    //求集合里第一个数的平方值
    val result:Int = listOf(3, 2, 1).first().let {
        it * it
    }
    println(result)
    //如果不用let函数，你需要把第一个数赋给一个变量
    val firstElement:Int = listOf(3,2,1).first()
    val result2:Int = firstElement * firstElement
}
```

```
//使用let
fun formatGreeting(guestName: String?): String {
    return guestName?.let{ it: String
        "Welcome, $it."
    } ?: "What's your name?"
}
//不用let
fun formatGreeting2(guestName: String?): String {
    return if(guestName != null){
        "Welcome, $guestName."
    }else {
        "What's your name?"
    }
}
```

run

- 光看作用域行为，run和apply差不多，但与apply不同，run函数不返回接收者，run返回的是lambda结果，也就是true或者false。

```
1   import java.io.File
2
3   fun main() {
4       //查看某个文件是否包含某一个字符串
5       val file = File( pathname: "E://i have a dream_copy.txt")
6       val result : Boolean  = file.run { this: File
7           readText().contains( other: "great")
8       }
9       println(result)
10  }
```

run

- run也能用来执行函数引用

```
fun main() {
    val result2:Boolean = "The people's Republic of China.".run(::isLong)
    println(result2)
    //当有多个函数调用，run的优势就显而易见了
    "The people's Republic of China."
        .run(::isLong)
        .run(::showMessage)
        .run(::println)
}

fun isLong(name:String) = name.length >= 10

fun showMessage(isLong:Boolean):String{
    return if(isLong){
        "Name is too long."
    }else{
        "Please rename."
    }
}
```

with

- with函数是run的变体，他们的功能行为是一样的，但with的调用方式不同，调用with时需要值参作为其第一个参数传入。

```
1  fun main() {
2      val isTooLong: Boolean = with( receiver: "The people's Republic of China."){
3          length >= 10
4      }
5  }
```

also

- also函数和let函数功能相似，和let一样，also也是把接收者作为值参传给lambda，但有一点不同：also返回接收者对象，而let返回lambda结果。因为这个差异，also尤其适合针对同一原始对象，利用副作用做事，既然also返回的是接收者对象，你就可以基于原始接收者对象执行额外的链式调用。

```
1   import java.io.File
2
3   fun main() {
4       var fileContents:List<String>   //没有初始化
5       File( pathname: "E://i have a dream_copy.txt")
6           .also { it: File
7               println(it.name)
8           }.also { it: File
9               fileContents = it.readLines() //初始化
10          }
11      println(fileContents)
12  }
```

takeIf

- 和其他标准函数有点不一样，takeIf函数需要判断lambda中提供的条件表达式，给出true或false结果，如果判断结果是true，从takeIf函数返回接收者对象，如果是false，则返回null。如果**需要判断某个条件是否满足，再决定是否可以赋值变量或执行某项任务**，takeIf就非常有用，概念上讲，takeIf函数类似于if语句，但它的优势是可以直接在对象实例上调用，避免了临时变量赋值的麻烦。

```
1   import java.io.File
2
3   fun main() {
4       val fileContents:String? = File(pathname:"E://i have a dream_copy.txt")
5           .takeIf { it.canRead() && it.canWrite() }
6           ?.readText()
7       //不用takeIf函数
8       val file = File(pathname:"E://i have a dream_copy.txt")
9       val fileContents2:String? = if (file.canRead() && file.canWrite()) {
10          file.readText()
11      } else {
12          null
13      }
14  }
```

takeUnless

- takeIf辅助函数takeUnless，只有判断你给定的条件结果是false时，takeUnless才会返回原始接收者对象。

```
1   import java.io.File
2
3   fun main() {
4       val fileContents3: String? = File( pathname: "E://i have a dream_copy.txt")
5           .takeUnless { it.isHidden }
6           ?.readText()
7       println(fileContents3)
8   }
```