

这样就能保证不管传入的参数是什么，这段代码始终都是安全的。

由此可以看出，即使是如此简单的一小段代码，都有产生空指针异常的潜在风险，那么在大型项目中，想要完全规避空指针异常几乎是不可能的事情，这也是它高居各类崩溃排行榜的原因。

### 2.7.1 可空类型系统

然而，Kotlin 却非常科学地解决了这个问题，它利用编译时判空检查的机制几乎杜绝了空指针异常。虽然编译时判空检查的机制有时候会导致代码变得比较难写，但是不用担心，Kotlin 提供了一系列的辅助工具，让我们能轻松地处理各种判空情况。下面我们就逐步开始学习吧。

还是回到刚才的 `doStudy()` 函数，现在将这个函数再翻译回 Kotlin 版本，代码如下所示：

```
fun doStudy(study: Study) {  
    study.readBooks()  
    study.doHomework()  
}
```

这段代码看上去和刚才的 Java 版本并没有什么区别，但实际上它是没有空指针风险的，因为 Kotlin 默认所有的参数和变量都不可为空，所以这里传入的 `Study` 参数也一定不会为空，我们可以放心地调用它的任何函数。如果你尝试向 `doStudy()` 函数传入一个 `null` 参数，则会如图 2.30 所示的错误。

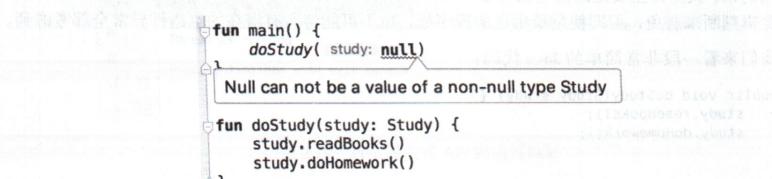


图 2.30 向 `doStudy()` 方法传入 `null` 参数

也就是说，Kotlin 将空指针异常的检查提前到了编译时期，如果我们的程序存在空指针的风险，那么在编译的时候会直接报错，修正之后才能成功运行，这样就可以保证程序在编译时期不会出现空指针异常了。

看到这里，你可能产生了巨大的疑惑，所有的参数和变量都不可为空？这可真是前所未有的事情，那如果我们的业务逻辑就是需要某个参数或者变量为空该怎么办呢？不用担心，Kotlin 提供了另外一套可为空的类型系统，只不过在使用可为空的类型系统时，我们需要在编译时将所有潜在的空指针异常都处理掉，否则代码将无法编译通过。

那么可为空的类型系统是什么样的呢？很简单，就是在类名的后面加上一个问号。比

表示不可为空的整型,而 Int?就表示可为空的整型;String 表示不可为空的字符串,而 String?就表示可为空的字符串。

回到刚才的 doStudy()函数,如果我们希望传入的参数可以为空,那么就应该将参数的类型由 Study 改成 Study?,如图 2.31 所示。

```
fun main() {
    doStudy(study: null)
}

fun doStudy(study: Study?) {
    study.readBooks()
    study.doHomework()
}
```

图 2.31 允许 Study 参数为空

可以看到,现在在调用 doStudy()函数时传入 null 参数,就不会再提示错误了。然而你会发现,在 doStudy()函数中调用参数的 readBooks()和 doHomework()方法时,却出现了一个红色下划线的错误提示,这又是为什么呢?

其实原因也很明显,由于我们将参数改成了可为空的 Study?类型,此时调用参数的 readBooks()和 doHomework()方法都可能造成空指针异常,因此 Kotlin 在这种情况下不允许编译通过。

那么该如何解决呢?很简单,只要把空指针异常都处理掉就可以了,比如做个判断处理,如下所示:

```
fun doStudy(study: Study?) {
    if (study != null) {
        study.readBooks()
        study.doHomework()
    }
}
```

现在代码就可以正常编译通过了,并且还能保证完全不会出现空指针异常。

其实学到这里,我们就已经基本掌握了 Kotlin 的可空类型系统以及空指针检查的机制,但是为了在编译时期就处理掉所有的空指针异常,通常需要编写很多额外的检查代码才行。如果每处检查代码都使用 if 判断语句,则会让代码变得比较啰嗦,而且 if 判断语句还处理不了全局变量的判空问题。为此,Kotlin 专门提供了一系列的辅助工具,使开发者能够更轻松地进行判空处理,下面我们就来逐个学习一下。

## 2.7.2 判空辅助工具

首先学习最常用的?.操作符。这个操作符的作用非常好理解,就是当对象不为空时正常调用

相应的方法，当对象为空时则什么都不做。比如以下的判空处理代码：

```
if (a != null) {  
    a.doSomething()  
}
```

这段代码使用?.操作符就可以简化成：

```
a?.doSomething()
```

了解了?.操作符的作用，下面我们来看一下如何使用这个操作符对 doStudy() 函数进行优化，代码如下所示：

```
fun doStudy(study: Study?) {  
    study?.readBooks()  
    study?.doHomework()  
}
```

可以看到，这样我们就借助?.操作符将 if 判断语句去掉了。可能你会觉得使用 if 语句来进行判空处理也没什么复杂的，那是因为目前的代码还非常简单。当以后我们开发的功能越来越复杂，需要判空的对象也越来越多的时候，你就会觉得?.操作符特别好用了。

下面我们再来学习另外一个非常常用的?:操作符。这个操作符的左右两边都接收一个表达式，如果左边表达式的结果不为空就返回左边表达式的结果，否则就返回右边表达式的结果。观察如下代码：

```
val c = if (a != null) {  
    a  
} else {  
    b  
}
```

这段代码的逻辑使用?:操作符就可以简化成：

```
val c = a ?: b
```

接下来我们通过一个具体的例子来结合使用?.和?:这两个操作符，从而让你加深对它们的理解。

比如现在我们要编写一个函数用来获得一段文本的长度，使用传统的写法就可以这样写：

```
fun getTextLength(text: String?): Int {  
    if (text != null) {  
        return text.length  
    }  
    return 0  
}
```

由于文本是可能为空的，因此我们需要先进行一次判空操作。如果文本不为空就返回它的长度，如果文本为空就返回 0。

这段代码看上去也并不复杂，但是我们却可以借助操作符让它变得更加简单，如下所示：

```
fun getTextLength(text: String?) = text?.length ?: 0
```

这里我们将`?.`和`?:`操作符结合到了一起使用，首先由于`text`是可能为空的，因此我们在调用它的`length`字段时需要使用`?.`操作符，而当`text`为空时，`text?.length`会返回一个`null`值，这个时候我们再借助`?:`操作符让它返回`0`。怎么样，是不是觉得这些操作符越来越好用了呢？

不过 Kotlin 的空指针检查机制也并非总是那么智能，有的时候我们可能从逻辑上已经将空指针异常处理了，但是 Kotlin 的编译器并不知道，这个时候它还是会编译失败。

观察如下的代码示例：

```
var content: String? = "hello"

fun main() {
    if (content != null) {
        printUpperCase()
    }
}

fun printUpperCase() {
    val upperCase = content.toUpperCase()
    println(upperCase)
}
```

这里我们定义了一个可为空的全局变量`content`，然后在`main()`函数里先进行一次判空操作，当`content`不为空的时候才会调用`printUpperCase()`函数，在`printUpperCase()`函数里，我们将`content`转换为大写模式，最后打印出来。

看上去好像逻辑没什么问题，但是很遗憾，这段代码一定是无法运行的。因为`printUpperCase()`函数并不知道外部已经对`content`变量进行了非空检查，在调用`toUpperCase()`方法时，还认为这里存在空指针风险，从而无法编译通过。

在这种情况下，如果我们想要强行通过编译，可以使用非空断言工具，写法是在对象的后面加上`!!`，如下所示：

```
fun printUpperCase() {
    val upperCase = content!!.toUpperCase()
    println(upperCase)
}
```

这是一种有风险的写法，意在告诉 Kotlin，我非常确信这里的对象不会为空，所以不用你来帮我做空指针检查了，如果出现问题，你可以直接抛出空指针异常，后果由我自己承担。

虽然这样编写代码确实可以通过编译，但是当你想要使用非空断言工具的时候，最好提醒一下自己，是不是还有更好的实现方式。你最自信这个对象不会为空的时候，其实可能就是一个潜在空指针异常发生的时候。

最后我们再来学习一个比较与众不同的辅助工具——`let`。`let` 既不是操作符，也不是什么关键字，而是一个函数。这个函数提供了函数式 API 的编程接口，并将原始调用对象作为参数传递到 Lambda 表达式中。示例代码如下：

```
obj.let { obj2 ->
    // 编写具体的业务逻辑
}
```

可以看到，这里调用了 `obj` 对象的 `let` 函数，然后 Lambda 表达式中的代码就会立即执行，并且这个 `obj` 对象本身还会作为参数传递到 Lambda 表达式中。不过，为了防止变量重名，这里我将参数名改成了 `obj2`，但实际上它们是同一个对象。这就是 `let` 函数的作用。

`let` 函数属于 Kotlin 中的标准函数，在下一章中我们将会学习更多 Kotlin 标准函数的用法。

你可能就要问了，这个 `let` 函数和空指针检查有什么关系呢？其实 `let` 函数的特性配合`?.`操作符可以在空指针检查的时候起到很大的作用。

我们回到 `doStudy()` 函数当中，目前的代码如下所示：

```
fun doStudy(study: Study?) {
    study?.readBooks()
    study?.doHomework()
}
```

虽然这段代码我们通过`?.`操作符优化之后可以正常编译通过，但其实这种表达方式是有点啰嗦的，如果将这段代码准确翻译成使用 `if` 判断语句的写法，对应的代码如下：

```
fun doStudy(study: Study?) {
    if (study != null) {
        study.readBooks()
    }
    if (study != null) {
        study.doHomework()
    }
}
```

也就是说，本来我们进行一次 `if` 判断就能随意调用 `study` 对象的任何方法，但受制于`?.`操作符的限制，现在变成了每次调用 `study` 对象的方法时都要进行一次 `if` 判断。

这个时候就可以结合使用`?.`操作符和 `let` 函数来对代码进行优化了，如下所示：

```
fun doStudy(study: Study?) {
    study?.let { stu ->
        stu.readBooks()
        stu.doHomework()
    }
}
```

我来简单解释一下上述代码，`?.`操作符表示对象为空时什么都不做，对象不为空时就调用 `let` 函数，而 `let` 函数会将 `study` 对象本身作为参数传递到 Lambda 表达式中，此时的 `study`

对象肯定不为空了，我们就能放心地调用它的任意方法了。

另外还记得 Lambda 表达式的语法特性吗？当 Lambda 表达式的参数列表中只有一个参数时，可以不用声明参数名，直接使用 `it` 关键字来代替即可，那么代码就可以进一步简化成：

```
fun doStudy(study: Study?) {
    study?.let {
        it.readBooks()
        it.doHomework()
    }
}
```

在结束本小节内容之前，我还得再讲一点，`let` 函数是可以处理全局变量的判空问题的，而 `if` 判断语句则无法做到这一点。比如我们将 `doStudy()` 函数中的参数变成一个全局变量，使用 `let` 函数仍然可以正常工作，但使用 `if` 判断语句则会提示错误，如图 2.32 所示。

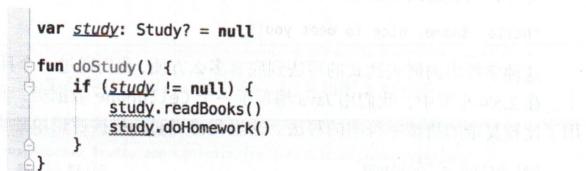


图 2.32 使用 `if` 判断语句对全局变量进行判空

之所以这里会报错，是因为全局变量的值随时都有可能被其他线程所修改，即使做了判空处理，仍然无法保证 `if` 语句中的 `study` 变量没有空指针风险。从这一点上也能体现出 `let` 函数的优势。

好了，最常用的 Kotlin 空指针检查辅助工具大概就是这些了，只要能将本节的内容掌握好，你就可以写出更加健壮、几乎杜绝空指针异常的代码了。

## 2.8 Kotlin 中的小魔术

到目前为止，我们已经学习了很多 Kotlin 方面的编程知识，相信现在的你已经有能力进行一些日常的 Kotlin 开发工作了。在结束本章内容之前，我们再来学习几个魔术类的小技巧，虽说是小技巧，但是相信我，它们一定会给你带来巨大的帮助。

### 2.8.1 字符串内嵌表达式

字符串内嵌表达式是我认为 Java 最应该支持的功能，因为大多数现代高级语言是支持这个非常方便的功能的，但是 Java 直到今天都还不支持，至于为什么，我也想不明白，或许 Java 的开发团队有不这么做的原因和道理吧。

不过值得高兴的是，Kotlin 从一开始就支持了字符串内嵌表达式功能，弥补了 Java 在这一点上的遗憾。在 Kotlin 中，我们不需要再像使用 Java 时那样傻傻地拼接字符串了，而是可以直接将表达式写在字符串里面，即使是构建非常复杂的字符串，也会变得轻而易举。

本书到目前为止，我都还没有使用过字符串内嵌表达式的写法，一直在使用传统的加号连接符来拼接字符串。在学完本节的内容之后，我们就会永远和加号连接符的写法说“再见”了。

首先来看一下 Kotlin 中字符串内嵌表达式的语法规则：

```
"hello, ${obj.name}. nice to meet you!"
```

可以看到，Kotlin 允许我们在字符串里嵌入 \${} 这种语法结构的表达式，并在运行时使用表达式执行的结果替代这一部分内容。

另外，当表达式中仅有一个变量的时候，还可以将两边的大括号省略，如下所示：

```
"hello, $name. nice to meet you!"
```

这种字符串内嵌表达式的写法到底有多么方便，我们通过一个具体的例子来学习一下就知道了。在 2.5.4 小节中，我们用 Java 编写了一个 Cellphone 数据类，其中 `toString()` 方法里就使用了比较复杂的拼接字符串的写法。这里我将当时的拼接逻辑单独提炼了出来，代码如下：

```
val brand = "Samsung"  
val price = 1299.99  
println("Cellphone(brand=" + brand + ", price=" + price + ")")
```

可以看到，上述字符串中一共使用了 4 个加号连接符，这种写法不仅写起来非常吃力，很容易写错，而且在代码可读性方面也很糟糕。

而使用字符串内嵌表达式的写法就变得非常简单了，如下所示：

```
val brand = "Samsung"  
val price = 1299.99  
println("Cellphone(brand=$brand, price=$price)")
```

很明显，这种写法不管是在易读性还是易写性方面都更胜一筹，是 Kotlin 更加推崇的写法。这个小技巧会给我们以后的开发工作带来巨大的便利。

### 2.8.2 函数的参数默认值

接下来我们开始学习另外一个非常有用的小技巧——给函数设定参数默认值。

其实之前在学习次构造函数用法的时候我就提到过，次构造函数在 Kotlin 中很少用，因为 Kotlin 提供了给函数设定参数默认值的功能，它在很大程度上能够替代次构造函数的作用。

具体来讲，我们可以在定义函数的时候给任意参数设定一个默认值，这样当调用此函数时就不会强制要求调用方为此参数传值，在没有传值的情况下会自动使用参数的默认值。

给参数设定默认值的方式也很简单，观察如下代码：

```
fun printParams(num: Int, str: String = "hello") {  
    println("num is $num , str is $str")  
}
```

可以看到，这里我们给 `printParams()` 函数的第二个参数设定了一个默认值，这样当调用 `printParams()` 函数时，可以选择给第二个参数传值，也可以选择不传，在不传的情况下就会自动使用默认值。

现在我们在 `main()` 函数中调用一下 `printParams()` 函数来进行测试，代码如下：

```
fun printParams(num: Int, str: String = "hello") {  
    println("num is $num , str is $str")  
}  
  
fun main() {  
    printParams(123)  
}
```

注意，这里并没有给第二个参数传值。运行一下代码，结果如图 2.33 所示。

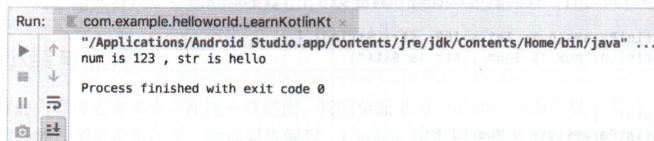


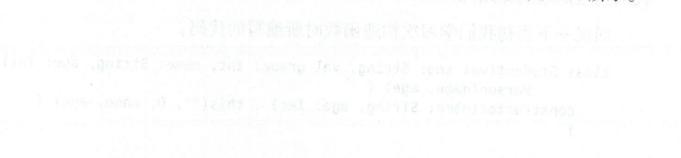
图 2.33 `str` 参数使用默认值的打印结果

可以看到，在没有给第二个参数传值的情况下，`printParams()` 函数自动使用了参数的默认值。

当然上面这个例子比较理想化，因为正好是给最后一个参数设定了默认值，现在我们将代码改成给第一个参数设定默认值，如下所示：

```
fun printParams(num: Int = 100, str: String) {  
    println("num is $num , str is $str")  
}
```

这时如果想让 `num` 参数使用默认值该怎么办呢？模仿刚才的写法肯定是行不通的，因为编译器会认为我们想把字符串赋值给第一个 `num` 参数，从而报类型不匹配的错误，如图 2.34 所示。



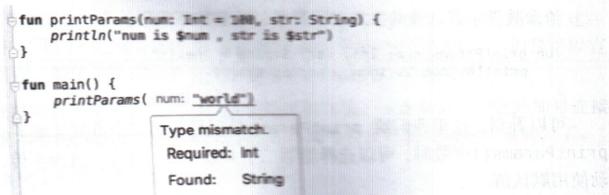


图 2.34 类型不匹配错误提示

不过不用担心，Kotlin 提供了另外一种神奇的机制，就是可以通过键值对的方式来传参，从而不必像传统写法那样按照参数定义的顺序来传参。比如调用 printParams() 函数，我们还可以这样写：

```
printParams(str = "world", num = 123)
```

此时哪个参数在前哪个参数在后都无所谓，Kotlin 可以准确地将参数匹配上。而使用这种键值对的传参方式之后，我们就可以省略 num 参数了，代码如下：

```

fun printParams(num: Int = 100, str: String) {
    println("num is $num , str is $str")
}

fun main() {
    printParams(str = "world")
}

```

重新运行一下程序，结果如图 2.35 所示。

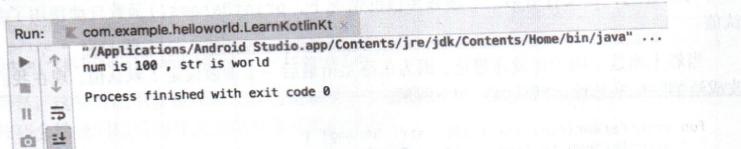


图 2.35 num 参数使用默认值的打印结果

现在你已经掌握了如何给函数设定参数默认值，那么为什么说这个功能可以在很大程度上替代次构造函数的作用呢？

回忆一下当初我们学习次构造函数时所编写的代码：

```

class Student(val sno: String, val grade: Int, name: String, age: Int) :
    Person(name, age) {
    constructor(name: String, age: Int) : this("", 0, name, age) {
    }
}

```

```
    constructor() : this("", 0) {
}
```

上述代码中有一个主构造函数和两个次构造函数，次构造函数在这里的作用是提供了使用更少参数来对 `Student` 类进行实例化的方式。无参的次构造函数会调用两个参数的次构造函数，并将这两个参数赋值成初始值。两个参数的次构造函数会调用 4 个参数的主构造函数，并将缺失的两个参数也赋值成初始值。

这种写法在 Kotlin 中其实是不必要的，因为我们完全可以通过只编写一个主构造函数，然后给参数设定默认值的方式来实现，代码如下所示：

```
class Student(val sno: String = "", val grade: Int = 0, name: String = "", age: Int = 0) :
    Person(name, age)
```

在给主构造函数的每个参数都设定了默认值之后，我们就可以使用任何传参组合的方式来对 `Student` 类进行实例化，当然也包含了刚才两种次构造函数的使用场景。

由此可见，给函数设定参数默认值这个小技巧的作用还是极大的。

## 2.9 小结与点评

本章的内容可着实不少，在这一章里面，我们全面学习了 Kotlin 编程中最主要的知识点，包括变量和函数、逻辑控制语句、面向对象编程、Lambda 编程、空指针检查机制，等等。虽然这还远不足以涵盖 Kotlin 的所有内容，但是这里我要祝贺你，现在你已经有足够的实力使用 Kotlin 来学习 Android 程序开发了！

因此，从下一章开始，我们将正式踏上 Android 开发学习之旅。不过在这之后的每一章里，我都会结合相应章节的内容穿插讲解一些 Kotlin 进阶方面的知识，从而让你在 Android 和 Kotlin 两方面都能够持续不断地进步。那么稍事休息，让我们继续前行吧！