掌握 Kotlin 中的标准库函数: run、with、let、also 和 apply

> Kotlin中的一些标准库函数非常相似,以致于我们不确定要使用哪个函数。这里我将介绍一种简单的方法来清楚地区分它们之间的差异以及如何选择使用哪个函数。

作用域函数

下面我将关于 **run**、**with**、**T.run**、**T.let**、**T.also** 和 **T.apply** 这些函数，并把它们称为作用域函数，因为我注意到它们的主要功能是为调用者函数提供内部作用域。

说明作用域最简单的方式是 **run** 函数

```
fun test() {
    var mood = "I am sad"

    run {
        val mood = "I am happy"
        println(mood) // I am happy
    }
    println(mood)  // I am sad
}
```

在这个例子中，在 **test** 函数的内部有一个分隔开的作用域 (run的内部)，在这个作用域内部完全包含一个在输出之前的 **mood** 变量被重新定义并初始化为 **I am happy** 的操作实现。

这个作用域函数本身似乎看起来不是很有用。但是这还有一个比作用域有趣一点是，它返回一些东西，是这个作用域内部的最后一个对象。

因此，以下的内容会变得更加整洁，我们可以将 **show()** 方法应用到两个 View 中，而不需要去调用两次 show() 方法。

```
run {
    if (firstTimeView) introView else normalView
}.show()
```

作用域函数的三个属性特征

为了让作用域函数更有趣，让我把他们的行为分类成三个属性特征。我将会使用这些属性特征来区分他们每一个函数。

1、普通函数 VS 扩展函数 (Normal vs. extension function)

如果我们对比 **with** 和 **T.run** 这两个函数的话，他们实际上是十分相似的。下面使用他们实现相同的功能的例子.

```
with(webview.settings) {
    javaScriptEnabled = true
    databaseEnabled = true
}
// similarly
webview.settings.run {
    javaScriptEnabled = true
    databaseEnabled = true
}
```

然后他们之间唯一不同在于 **with** 是一个普通函数，而 **T.run** 是一个扩展函数。

那么问题来了，它们各自使用的优点是什么?

想象一下如果 **webview.settings** 可能为空，那么下面两种方式实现如何去修改呢?

```
// Yack!(比较丑陋的实现方式)
with(webview.settings) {
      this?.javaScriptEnabled = true
      this?.databaseEnabled = true
```

```
    }
}
// Nice.(比较好的实现方式)
webview.settings?.run {
    javaScriptEnabled = true
    databaseEnabled = true
}
```

在这种情况下，显然 **T.run** 扩展函数更好，因为我们可以在使用它之前进行空检查。

2、this VS it 参数(This vs. it argument)

如果我们对比 **T.run** 和 **T.let** 两个函数也是非常的相似，唯一的区别在于它们接收的参数不一样。下面显示了两种功能的相同逻辑。

```
stringVariable?.run {
      println("The length of this String is $length")
}
// Similarly.
stringVariable?.let {
      println("The length of this String is ${it.length}")
}
```

如果你查看过 **T.run** 函数声明，你就会注意到 **T.run** 仅仅只是被当做了 **block: T.()** 扩展函数的调用块。因此，在其作用域内，**T** 可以被 **this** 指代。在编码过程中，在大多数情况下 **this** 是可以被省略的。因此我们上面的示例中，我们可以在 **println** 语句中直接使用 $length 而不是 ${this.lenght}. 所以我把这个称之为传递 **this** 参数。

然而对于 **T.let** 函数的声明，你将会注意到 **T.let** 是传递它自己本身到函数中 **block: (T)**。因此这个类似于传递一个 **lambda** 表达式作为参数。它可以在函数作用域内部使用 **it** 来指代. 所以我把这个称之为传递 **it** 参数。

从上面看，似乎 **T.run** 比 **T.let** 更加优越，因为它更隐含，但是 **T.let** 函数具有一些微妙的优势，如下所示:

1. **T.let** 函数提供了一种更清晰的区分方式去使用给定的变量函数/成员与外部类函数/成员。
2. 例如当 **it** 作为函数的参数传递时，**this** 不能被省略，并且 **it** 写起来比 **this** 更简洁，更清晰。

3. **T.let** 允许更好地命名已转换的已使用变量，即可以将 **it** 转换为其他有含义名称，而 **T.run** 则不能，内部只能用 **this** 指代或者省略。

```
stringVariable?.let {
      nonNullString ->
      println("The non null string is $nonNullString")
}
```

3、返回this VS 其他类型 (Return this vs. other type)

现在，让我们看看 **T.let** 和 **T.also** ，如果我们看看它的内部函数作用域，它们都是相同的。

```
stringVariable?.let {
      println("The length of this String is ${it.length}")
}
// Exactly the same as below
stringVariable?.also {
      println("The length of this String is ${it.length}")
}
```

然而，他们微妙的不同在于他们的返回值 **T.let** 返回一个不同类型的值，而 **T.also** 返回 **T** 类型本身，即这个。

这两个函数对于函数的链式调用都很有用，其中 **T.let** 让您演变操作，而 **T.also** 则让您对相同的变量执行操作。

简单的例子如下:

```
val original = "abc"
// Evolve the value and send to the next chain
original.let {
    println("The original String is $it") // "abc"
    it.reversed() // evolve it as parameter to send to next let
}.let {
    println("The reverse String is $it") // "cba"
    it.length  // can be evolve to other type
}.let {
    println("The length of the String is $it") // 3
}
// Wrong
// Same value is sent in the chain (printed answer is wrong)
original.also {
```

```
    println("The original String is $it") // "abc"
    it.reversed() // even if we evolve it, it is useless
}.also {
    println("The reverse String is ${it}") // "abc"
    it.length  // even if we evolve it, it is useless
}.also {
    println("The length of the String is ${it}") // "abc"
}
// Corrected for also (i.e. manipulate as original string
// Same value is sent in the chain
original.also {
    println("The original String is $it") // "abc"
}.also {
    println("The reverse String is ${it.reversed()}") // "cba"
}.also {
    println("The length of the String is ${it.length}") // 3
}
```

T.also 似乎看上去没有意义，因为我们可以很容易地将它们组合成一个功能块。仔细思考，它有一些很好的优点。

1. 它可以对相同的对象提供非常清晰的分离过程，即创建更小的函数部分。
2. 在使用之前，它可以非常强大的进行自我操作，从而实现整个链式代码的构建操作。

当两者结合在一起使用时，即一个自身演变，一个自我保留，它能使一些操作变得更加强大。

```
// Normal approach
fun makeDir(path: String): File  {
    val result = File(path)
    result.mkdirs()
    return result
}
// Improved approach
fun makeDir(path: String) = path.let{ File(it) }.also{ it.mkdirs() }
```

回顾所有属性特征

通过回顾这3个属性特征，我们可以非常清楚函数的行为。让我来说明 **T.apply** 函数，由于我并没有以上函数中提到过它。**T.apply** 的三个属性如下

1. 它是一个扩展函数

2. 它是传递this作为参数
3. 它是返回 this (即它自己本身)

因此，使用它，可以想象下它可以被用作:

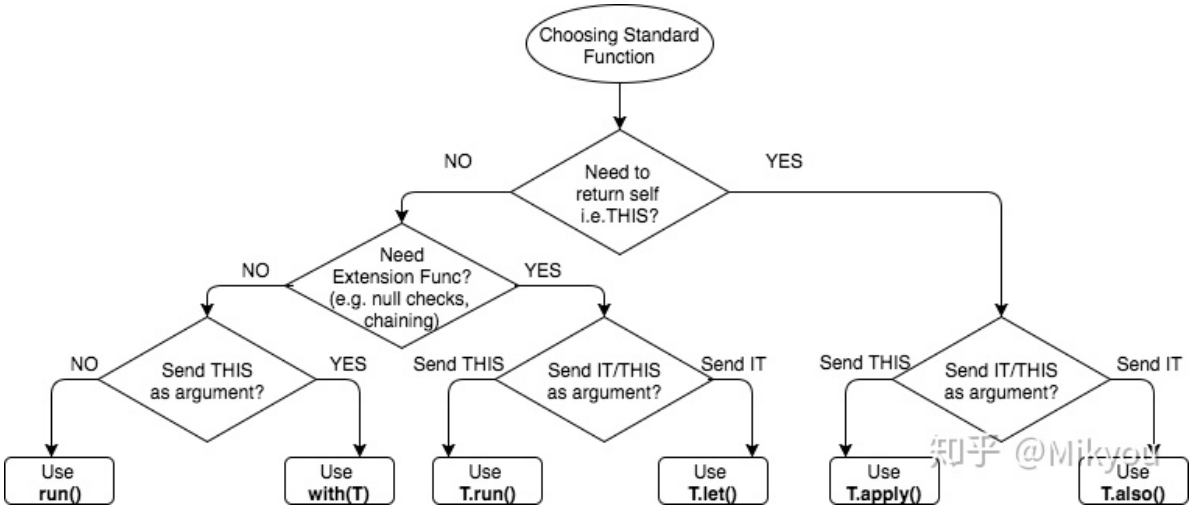
```
// Normal approach
fun createInstance(args: Bundle) : MyFragment {
    val fragment = MyFragment()
    fragment.arguments = args
    return fragment
}
// Improved approach
fun createInstance(args: Bundle)
              = MyFragment().apply { arguments = args }
```

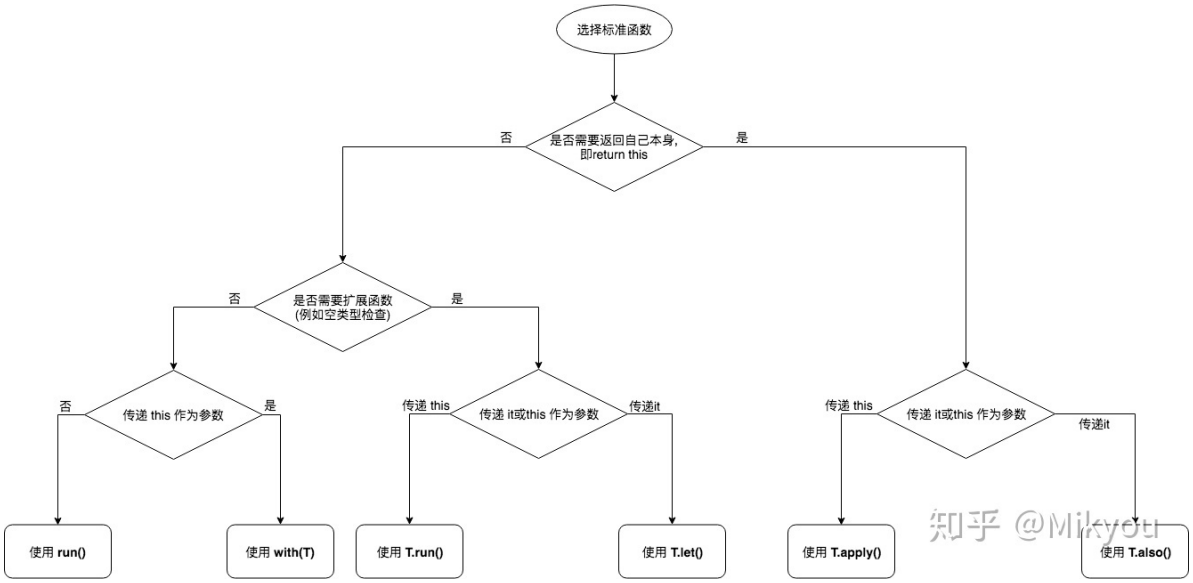
或者我们也可以让无链对象创建链式调用。

```
// Normal approach
fun createIntent(intentData: String, intentAction: String): Intent {
    val intent = Intent()
    intent.action = intentAction
    intent.data=Uri.parse(intentData)
    return intent
}
// Improved approach, chaining
fun createIntent(intentData: String, intentAction: String) =
        Intent().apply { action = intentAction }
                .apply { data = Uri.parse(intentData) }
```

函数的选用

因此，显然有了这3个属性特征，我们现在可以对功能进行相应的分类。基于此，我们可以在下面构建一个决策树，以帮助确定我们想要使用哪个函数，来选择我们需要的。





希望上面的决策树能够更清晰地说明功能，并简化你的决策，使你能够适当掌握这些功能的使用。

译者有话说

1、为什么我要翻译这篇博客?

我们都知道在 **Kotlin** 中 **Standard.Kt** 文件中短短不到100来行库函数源码，但是它们作用是非常强大，可以说它们是贯穿于整个Kotlin开发编码过程中。使用它们能让你的

代码会更具有可读性、更优雅、更简洁。善于合理使用标准库函数，也是衡量你对 **Kotlin** 掌握程度标准之一，因为你去看一些开源 **Kotlin** 源码，随处可见的都是使用各种标准库函数。

但是这些库函数有难点在于它们的用法都非常相似，有的人甚至认为有的库函数都是多余的，其实不然，每个库函数都是有它的实际应用场景。虽然有时候你能用一种库函数也能实现相同的功能，但是也许那并不是最好的实现方式。

相信很多初学者对于这些标准库函数也是傻傻分不清楚(曾经的我也是)，但是这篇博客非常一点在于它提取出了这些库函数三个主要特征:**是否是扩展函数、是否传递this或it做为参数**(在函数内部表现就是this和it的指代)、**是否需要返回调用者对象本身**，基于特征就可以进行分类，分类后相应的应用场景也就一目了然。这种善于提取特征思路还是值得学习的。

2、关于使用标准库函数需要补充的几点。

1、建议尽量不要使用多个标准库函数进行嵌套，不要为了简化而去做简化，否则整个代码可读性会大大降低，一会是 **it** 指代，一会又是 **this** 指代，估计隔一段时间后连你自己都不知道指代什么了。

2、针对上面译文的 **let** 函数和 **run** 函数需要补充下，他们之所以能够返回其他类型的值，其原理在于 **lambda** 表达式内部返回最后一行表达式的值，所以只要最后一行表达式返回不同的对象，那么它们就返回不同类型，表现上就是返回其他类型。

3、关于 **T.also** 和 **T.apply** 函数为什么都能返回自己本身，是因为在各自 **lambda** 表达式内部最后一行都调用 **return this**，返回它们自己本身，这个 **this** 能被指代调用者，是因为它们都是扩展函数特性。

原文

- Mastering Kotlin standard functions: run, with, let, also and apply
- [译]掌握Kotlin中的标准库函数: run、with、let、also和apply