

数，除了使用最基本的 if 语句之外，还可以借助 Kotlin 内置的 max() 函数，如下所示：

```
val a = 10
val b = 15
val larger = max(a, b)
```

这种代码看上去简单直观，也很容易理解，因此好像并没有什么优化的必要。

可是现在如果我们想要在 3 个数中获取最大的那个数，应该怎么写呢？由于 max() 函数只能接收两个参数，因此需要先比较前两个数的大小，然后再拿较大的那个数和剩余的数进行比较，写法如下：

```
val a = 10
val b = 15
val c = 5
val largest = max(max(a, b), c)
```

有没有觉得代码开始变得复杂了呢？3 个数中获取最大值就需要使用这种嵌套 max() 函数的写法了，那如果是 4 个数、5 个数呢？没错，这个时候你就应该意识到，我们是可以对 max() 函数的用法进行简化的。

回顾一下，我们之前在第 7 章的 Kotlin 课堂中学过 vararg 关键字，它允许方法接收任意多个同等类型的参数，正好满足我们这里的需求。那么我们就可以新建一个 Max.kt 文件，并在其自定义一个 max() 函数，如下所示：

```
fun max(vararg nums: Int): Int {
    var maxNum = Int.MIN_VALUE
    for (num in nums) {
        maxNum = max(maxNum, num)
    }
    return maxNum
}
```

可以看到，这里 max() 函数的参数声明中使用了 vararg 关键字，也就是说现在它可以接收任意多个整型参数。接着我们使用了一个 maxNum 变量来记录所有数的最大值，并在一开始将它赋值成了整型范围的最小值。然后使用 for-in 循环遍历 nums 参数列表，如果发现当前遍历的数字比 maxNum 更大，就将 maxNum 的值更新成这个数，最终将 maxNum 返回即可。

仅仅经过这样的一层封装之后，我们在使用 max() 函数时就会有翻天覆地的变化，比如刚刚同样的功能，现在就可以使用如下的写法来实现：

```
val a = 10
val b = 15
val c = 5
val largest = max(a, b, c)
```

这样我们就彻底摆脱了嵌套函数调用的写法，现在不管是求 3 个数的最大值还是求 N 个数的最大值，只需要不断地给 max() 函数传入参数就可以了。

不过，目前我们自定义的 `max()` 函数还有一个缺点，就是它只能求 N 个整型数字的最大值，如果我还想求 N 个浮点型或长整型数字的最大值，该怎么办呢？当然你可以定义很多个 `max()` 函数的重载，来接收不同类型的参数，因为 Kotlin 中内置的 `max()` 函数也是这么做的。但是这种方案实现起来过于烦琐，而且还会产生大量的重复代码，因此这里我准备使用一种更加巧妙的做法。

Java 中规定，所有类型的数字都是可比较的，因此必须实现 `Comparable` 接口，这个规则在 Kotlin 中也同样成立。那么我们就可以借助泛型，将 `max()` 函数修改成接收任意多个实现 `Comparable` 接口的参数，代码如下所示：

```
fun <T : Comparable<T>> max(vararg nums: T): T {
    if (nums.isEmpty()) throw RuntimeException("Params can not be empty.")
    var maxNum = nums[0]
    for (num in nums) {
        if (num > maxNum) {
            maxNum = num
        }
    }
    return maxNum
}
```

可以看到，这里将泛型 `T` 的上界指定成了 `Comparable<T>`，那么参数 `T` 就必然是 `Comparable<T>` 的子类型了。接下来，我们判断 `nums` 参数列表是否为空，如果为空的话就主动抛出一个异常，提醒调用者 `max()` 函数必须传入参数。紧接着将 `maxNum` 的值赋值成 `nums` 参数列表中第一个参数的值，然后同样是遍历参数列表，如果发现了更大的值就对 `maxNum` 进行更新。

经过这样的修改之后，我们就可以更加灵活地使用 `max()` 函数了，比如说求 3 个浮点型数字的最大值，同样也变得轻而易举：

```
val a = 3.5
val b = 3.8
val c = 4.1
val largest = max(a, b, c)
```

而且现在不管是双精度浮点型、单精度浮点型，还是短整型、整型、长整型，只要是实现 `Comparable` 接口的子类型，`max()` 函数全部支持获取它们的最大值，是一种一劳永逸的做法。

而如果你想获取 N 个数的最小值，实现的方式也是类似的，只需要定义一个 `min()` 函数就可以了，这个功能就当做课后习题留给你来完成吧。

12.8.2 简化 Toast 的用法

我们在本书中已经使用过太多次 `Toast`，相信你已经非常熟悉了，但是用了这么久，你有没有觉得 `Toast` 用法其实有些烦琐呢？

首先回顾一下 `Toast` 的标准用法吧，如果想要在界面上弹出一段文字提示需要这样写：

```
Toast.makeText(context, "This is Toast", Toast.LENGTH_SHORT).show()
```

是不是很长的一段代码？而且曾经不知道有多少人因为忘记调用最后的 `show()` 方法，导致 `Toast` 无法弹出，从而产生一些千奇百怪的 bug。

由于 `Toast` 是非常常用的功能，每次都需要编写这么长的一段代码确实让人很头疼，这个时候你就应该考虑对 `Toast` 的用法进行简化了。

我们来分析一下，`Toast` 的 `makeText()` 方法接收 3 个参数：第一个参数是 `Toast` 显示的上下文环境，必不可少；第二个参数是 `Toast` 显示的内容，需要由调用方进行指定，可以传入字符串和字符串资源 id 两种类型；第三个参数是 `Toast` 显示的时长，只支持 `Toast.LENGTH_SHORT` 和 `Toast.LENGTH_LONG` 这两种值，相对来说变化不大。

那么我们就可以给 `String` 类和 `Int` 类各添加一个扩展函数，并在里面封装弹出 `Toast` 的具体逻辑。这样以后每次想要弹出 `Toast` 提示时，只需要调用它们的扩展函数就可以了。

新建一个 `Toast.kt` 文件，并在其中编写如下代码：

```
fun String.showToast(context: Context) {
    Toast.makeText(context, this, Toast.LENGTH_SHORT).show()
}

fun Int.showToast(context: Context) {
    Toast.makeText(context, this, Toast.LENGTH_SHORT).show()
}
```

这里分别给 `String` 类和 `Int` 类新增了一个 `showToast()` 函数，并让它们都接收一个 `Context` 参数。然后在函数的内部，我们仍然使用了 `Toast` 原生 API 用法，只是将弹出的内容改成了 `this`，另外将 `Toast` 的显示时长固定设置成 `Toast.LENGTH_SHORT`。

那么经过这样的扩展之后，我们以后在使用 `Toast` 时可以变得多么简单呢？体验一下就知道了，比如同样弹出一段文字提醒就可以这么写：

```
"This is Toast".showToast(context)
```

怎么样，比起原生 `Toast` 的用法，有没有觉得这种写法畅快多了呢？另外，这只是直接弹出一段字符串文本的写法，如果你想弹出一个定义在 `strings.xml` 中的字符串资源，也非常简单，写法如下：

```
R.string.app_name.showToast(context)
```

这两种写法分别调用的就是我们刚才在 `String` 类和 `Int` 类中添加的 `showToast()` 扩展函数。

当然，这种写法其实还存在一个问题，就是 `Toast` 的显示时长被固定了，如果我现在想要使用 `Toast.LENGTH_LONG` 类型的显示时长该怎么办呢？要解决这个问题，其实最简单的做法就是在 `showToast()` 函数中再声明一个显示时长参数，但是这样每次调用 `showToast()` 函数时都要

额外多传入一个参数，无疑增加了使用复杂度。

不知道你现在有没有受到什么启发呢？回顾一下，我们在第2章学习Kotlin基础语法的时候，曾经学过给函数设定参数默认值的功能。只要借助这个功能，我们就可以在不增加`showToast()`函数使用复杂度的情况下，又让它可以支持动态指定显示时长了。修改`Toast.kt`中的代码，如下所示：

```
fun String.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}

fun Int.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}
```

可以看到，我们给`showToast()`函数增加了一个显示时长参数，但同时也给它指定了一个参数默认值。这样我们之前所使用的`showToast()`函数的写法将完全不受影响，默认会使用`Toast.LENGTH_SHORT`类型的显示时长。而如果你想要使用`Toast.LENGTH_LONG`的显示时长，只需要这样写就可以了：

```
"This is Toast".showToast(context, Toast.LENGTH_LONG)
```

相信我，这样的`Toast`工具一定会给你的开发效率带来巨大的提升。

12.8.3 简化 Snackbar 的用法

`Snackbar`是我们在本章中学习的新控件，它和`Toast`的用法基本类似，但是又比`Toast`稍微复杂一些。

先来回顾一下`Snackbar`的常规用法吧，如下所示：

```
Snackbar.make(view, "This is Snackbar", Snackbar.LENGTH_SHORT)
    .setAction("Action") {
        // 处理具体的逻辑
    }
    .show()
```

可以看到，`Snackbar`中`make()`方法的第一个参数变成了`View`，而`Toast`中`makeText()`方法的第一个参数是`Context`，另外`Snackbar`还可以调用`setAction()`方法来设置一个额外的点击事件。除了这些区别之外，`Snackbar`和`Toast`的其他用法都是相似的。

那么对于这种结构的API，我们该如何进行简化呢？其实简化的方式并不固定，接下来我将演示的写法也只是我个人认为比较不错的一种。

由于`make()`方法接收一个`View`参数，`Snackbar`会使用这个`View`自动查找最外层的布局，用于展示`Snackbar`。因此，我们就可以给`View`类添加一个扩展函数，并在里面封装显示`Snackbar`

的具体逻辑。新建一个 Snackbar.kt 文件，并编写如下代码：

```
fun View.showSnackbar(text: String, duration: Int = Snackbar.LENGTH_SHORT) {
    Snackbar.make(this, text, duration).show()
}

fun View.showSnackbar(resId: Int, duration: Int = Snackbar.LENGTH_SHORT) {
    Snackbar.make(this, resId, duration).show()
}
```

这段代码应该还是很好理解的，和刚才的 `showToast()` 函数比较相似。只是我们将扩展函数添加到了 `View` 类当中，并在参数列表上声明了 `Snackbar` 要显示的内容以及显示的时长。另外，`Snackbar` 和 `Toast` 类似，显示的内容也是支持传入字符串和字符串资源 id 两种类型的，因此这里我们给 `showSnackbar()` 函数进行了两种参数类型的函数重载。

现在想要使用 `Snackbar` 显示一段文本提示，只需要这样写就可以了：

```
view.showSnackbar("This is Snackbar")
```

假如 `Snackbar` 没有 `setAction()` 方法，那么我们的简化工作到这里就可以结束了。但是 `setAction()` 方法作为 `Snackbar` 最大的特色之一，如果不能支持的话，我们编写的 `showSnackbar()` 函数也就变得毫无意义了。

这个时候，神通广大的高阶函数又能派上用场了，我们可以让 `showSnackbar()` 函数再额外接收一个函数类型参数，以此来实现 `Snackbar` 的完整功能支持。修改 `Snackbar.kt` 中的代码，如下所示：

```
fun View.showSnackbar(text: String, actionText: String? = null,
                     duration: Int = Snackbar.LENGTH_SHORT, block: (() -> Unit)? = null) {
    val snackbar = Snackbar.make(this, text, duration)
    if (actionText != null && block != null) {
        snackbar.setAction(actionText) {
            block()
        }
    }
    snackbar.show()
}

fun View.showSnackbar(resId: Int, actionResId: Int? = null,
                     duration: Int = Snackbar.LENGTH_SHORT, block: (() -> Unit)? = null) {
    val snackbar = Snackbar.make(this, resId, duration)
    if (actionResId != null && block != null) {
        snackbar.setAction(actionResId) {
            block()
        }
    }
    snackbar.show()
}
```

可以看到，这里我们给两个 `showSnackbar()` 函数都增加了一个函数类型参数，并且还增加

了一个用于传递给 `setAction()` 方法的字符串或字符串资源 id。这里我们需要将新增的两个参数都设置成可为空的类型，并将默认值都设置成空，然后只有当两个参数都不为空的时候，我们才去调用 Snackbar 的 `setAction()` 方法来设置额外的点击事件。如果触发了点击事件，只需要调用函数类型参数将事件传递给外部的 Lambda 表达式即可。

这样 `showSnackbar()` 函数就拥有比较完整的 Snackbar 功能了，比如本小节最开始的那段示例代码，现在就可以使用如下写法进行实现：

```
view.showSnackbar("This is Snackbar", "Action") {
    // 处理具体的逻辑
}
```

怎么样，和 Snackbar 原生 API 的用法相比，我们编写的 `showSnackbar()` 函数是不是要明显简单好用得多？

在本章的 Kotlin 课堂中，我带着你一共编写了 3 个工具方法，分别应用了顶层函数、扩展函数以及高阶函数的知识，当然还用到了像 `vararg`、参数默认值等技巧。Kotlin 给我们提供了太多出色的特性，因此在你学完了这么多特性之后，能否将它们灵活运用就成为了至关重要的事情。本节课里所实现的 3 个工具方法只能算是开胃菜，我非常期待未来你能编写出许多自己的工具方法，将 Kotlin 提供给我们的优秀特性充分发挥出来。

好了，关于 Kotlin 的内容就先讲到这里，下面我们将再次进入本书的特殊环节——Git 时间，学习一下关于 Git 的高级用法。

12.9 Git 时间：版本控制工具的高级用法

现在的你对于 Git 应该完全不会感到陌生了吧？通过之前两次 Git 时间的学习，你已经掌握了很多 Git 中常用的命令，像提交代码这种简单的操作相信肯定是最不倒你的。

打开终端界面，进入 MaterialTest 这个项目的根目录，然后执行提交操作：

```
git init
git add .
git commit -m "First Commit."
```

这样就将准备工作完成了，下面就让我们开始学习关于 Git 的高级用法。

12.9.1 分支的用法

分支是版本控制工具中比较高级且比较重要的一个概念，它主要的作用就是在现有代码的基础上开辟一个分叉口，使得代码可以在主干线和分支线上同时进行开发，且相互之间不会影响。分支的工作原理如图 12.21 所示。

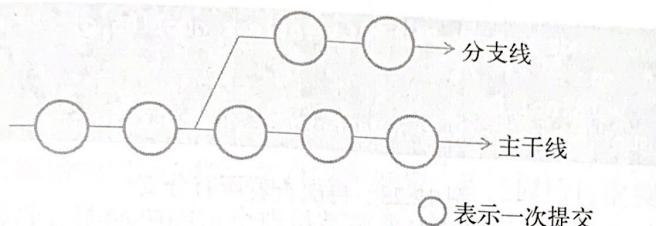


图 12.21 分支的工作原理示意图

你也许会有疑惑，为什么需要建立分支呢？只在主干线上进行开发不是挺好的吗？没错，通常情况下，只在主干线上进行开发是完全没有问题的。不过，一旦涉及发布版本的情况，如果不建立分支的话，你就会非常地头疼。举个简单的例子吧，比如说你们公司研发了一款不错的软件，最近刚刚完成，并推出了 1.0 版本。但是领导是不会让你们闲着的，马上提出了新的需求，让你们投入到 1.1 版本的开发工作当中。过了几个星期，1.1 版本的功能已经完成了一半，但是这个时候突然有用户反馈，之前上线的 1.0 版本发现了几个重大的 bug，严重影响软件的正常使用。领导也相当重视这个问题，要求你们立刻修复这些 bug，并对 1.0 版本进行更新，但这个时候你就非常为难了，你会发现根本没法去修复。因为现在 1.1 版本已经开发一半了，如果在现有代码的基础上修复这些 bug，那么更新的 1.0 版本将会带有一半 1.1 版本的功能！

进退两难了是不是？但是如果你使用了分支的话，就完全不会存在这个让人头疼的问题。你只需要在发布 1.0 版本的时候建立一个分支，然后在主干线上继续开发 1.1 版本的功能。当在 1.0 版本上发现任何 bug 的时候，就在分支线上进行修改，然后发布新的 1.0 版本，并记得将修改后的代码合并到主干线上。这样的话，不仅可以轻松解决 1.0 版本存在的 bug，而且保证了主干线上的代码也已经修复了这些 bug，当 1.1 版本发布时，就不会有同样的 bug 存在了。

说了这么多，相信你也已经意识到分支的重要性了，那么我们马上来学习一下如何在 Git 中操作分支吧。

分支的英文是 branch，如果想要查看当前的版本库当中有哪些分支，可以使用 git branch 这个命令，结果如图 12.22 所示。

```
guolin@MacBook-Pro:MaterialTest guolin$ git branch
* master
guolin@MacBook-Pro:MaterialTest guolin$
```

图 12.22 查看所有分支

由于目前 MaterialTest 项目中还没有创建过任何分支，因此只有一个 master 分支存在，这也是前面所说的主干线。接下来我们尝试创建一个分支，命令如下：

```
git branch version1.0
```

这样就创建了一个名为 version1.0 的分支，我们再次输入 git branch 这个命令来检查一下，结果如图 12.23 所示。

```
guolindeMacBook-Pro:MaterialTest guolin$ git branch
* master
  version1.0
guolindeMacBook-Pro:MaterialTest guolin$
```

图12.23 再次查看所有分支

可以看到，果然有一个叫作version1.0的分支出现了。你会发现，master分支的前面有一个“*”号，说明目前我们的代码还是在master分支上的，那么怎样才能切换到version1.0这个分支上呢？其实也很简单，只需要使用checkout命令即可，如下所示：

```
git checkout version1.0
```

再次输入git branch来进行检查，结果如图12.24所示。

```
guolindeMacBook-Pro:MaterialTest guolin$ git branch
  master
* version1.0
guolindeMacBook-Pro:MaterialTest guolin$
```

图12.24 查看切换分支后的结果

可以看到，我们已经把代码成功切换到version1.0这个分支上了。

需要注意的是，在version1.0分支上修改并提交的代码将不会影响到master分支。同样的道理，在master分支上修改并提交的代码也不会影响到version1.0分支。因此，如果我们在version1.0分支上修复了一个bug，在master分支上这个bug仍然是存在的。这时将修改的代码一行行复制到master分支上显然不是一种聪明的做法，最好的办法就是使用merge命令来完成合并操作，如下所示：

```
git checkout master
git merge version1.0
```

仅仅使用这样简单的两行命令，就可以把在version1.0分支上修改并提交的内容合并到master分支上了。当然，在合并分支的时候还可能出现代码冲突的情况，这个时候你就需要静下心来，慢慢解决这些冲突，Git在这里就无法帮助你了。

最后，当我们不再需要version1.0这个分支的时候，可以使用如下命令将这个分支删除：

```
git branch -D version1.0
```

12.9.2 与远程版本库协作

可以这样说，如果你是一个人在开发，那么使用版本控制工具就远远无法发挥出它真正强大的功能。没错，所有版本控制工具最重要的一个特点就是可以使用它来进行团队合作开发。每个人的电脑上都会有一份代码，当团队的某个成员在自己的电脑上编写完成了某个功能后，就将代

码提交到服务器，其他的成员只需要将服务器上的代码同步到本地，就能保证整个团队所有人的代码都相同。这样的话，每个团队成员就可以各司其职，大家共同来完成一个较为庞大的项目。

那么如何使用 Git 来进行团队合作开发呢？这就需要有一个远程的版本库，团队的每个成员都从这个版本库中获取最原始的代码，然后各自进行开发，并且以后每次提交的代码都同步到远程版本库上就可以了。另外，团队中的每个成员都要养成经常从版本库中获取最新代码的习惯，不然的话，大家的代码就很有可能经常出现冲突。

比如说现在有一个远程版本库的 Git 地址是 <https://github.com/example/test.git>，就可以使用如下命令将代码下载到本地：

```
git clone https://github.com/example/test.git
```

之后如果你在这份代码的基础上进行了一些修改和提交，那么怎样才能把本地修改的内容同步到远程版本库上呢？这就需要借助 push 命令来完成了，用法如下所示：

```
git push origin master
```

origin 部分指定的是远程版本库的 Git 地址，master 部分指定的是同步到哪一个分支上，上述命令就完成了将本地代码同步到 <https://github.com/example/test.git> 这个版本库的 master 分支上的功能。

知道了将本地的修改同步到远程版本库上的方法，接下来我们看一下如何将远程版本库上的修改同步到本地。Git 提供了两种命令来完成此功能，分别是 fetch 和 pull。fetch 的语法规则和 push 是差不多的，如下所示：

```
git fetch origin master
```

执行完这个命令后，就会将远程版本库上的代码同步到本地。不过同步下来的代码并不会合并到任何分支上，而是会存放到一个 origin/master 分支上，这时我们可以通过 diff 命令来查看远程版本库上到底修改了哪些东西：

```
git diff origin/master
```

之后再调用 merge 命令将 origin/master 分支上的修改合并到主分支上即可，如下所示：

```
git merge origin/master
```

而 pull 命令则是相当于将 fetch 和 merge 这两个命令放在一起执行了，它可以从远程版本库上获取最新的代码并且合并到本地，用法如下所示：

```
git pull origin master
```

也许你现在对远程版本库的使用还是感觉比较抽象，没关系，因为暂时我们只是了解了一下命令的用法，还没进行实践，在第 15 章当中，你将会对远程版本库的用法有更深一层的认识。

12.10 小结与点评

学完了本章的所有知识，你有没有觉得无比兴奋呢？反正我是这么觉得的。本章我们的收获实在是太多了，一开始创建了一个什么都没有的空项目，经过一章的学习，最后实现了一个功能如此丰富、界面如此华丽的应用，还有什么事情比这个更让我们有成就感吗？

本章中我们充分利用了Material库、AndroidX库以及一些开源项目，实现了一个高度Material化的应用程序。能将这些库中的相关控件熟练掌握，你的Material Design技术就算是合格了。

不过说到底，我仍然还是在以开发者的思维给你讲解Material Design，侧重于如何去实现这些效果。而实际上，Material Design的设计思维和设计理念才是更加重要的东西。当然，这部分内容其实应该是UI设计人员去学习的，如果你也感兴趣的话，可以参考一下Material Design的官方网站：<https://material.io/>。

至于本章的Kotlin课堂，我们并没有学习什么新的知识，而是通过编写几个工具方法的示例来引导你学会对Kotlin的各种特性进行灵活运用。知识好学，但是思维却是很难培养的，也希望经过本节课的学习能让你引发更多的思考。

除此之外，在本章的Git时间中，我们继续对Git的用法进行了更深一步的探究，相信你对分支和远程版本库的使用都有了一定层次的了解。

现在你已经足足学习了12章的内容，对Android应用程序开发的理解应该比较深刻了。那么掌握了这么多的知识，就可以开发出一款好的应用程序了吗？说实话，现在的你还差了些火候，因为你还不知道该如何搭建一个出色的代码架构体系。当然这也是我们下一章中即将学习的内容了——高级程序开发组件Jetpack。

学会本章的知识后，你就能更从容地面对本章所涉及的“如何使用Material Design设计思维和设计理念”这一部分了。当然，这需要你对Material Design的理论知识有一定的理解，否则你可能会觉得有些吃力。但只要你坚持学习，相信你一定能掌握它！

好了，本章就到这里，希望你能够喜欢。在本章的最后，我们来做一个小结：回顾一下本章所讲的内容，你对Material Design有了哪些新的认识？

本章首先从Material Design的理论入手，介绍了Material Design的理论基础，然后讲解了Material Design的视觉语言，接着介绍了Material Design的组件，最后讲解了Material Design的交互设计。通过本章的学习，你对Material Design有了一个全面的认识，为以后设计自己的应用打下了坚实的基础。

12.9.2 与适配器类的动作