Kotlin的Lambda表达式，大多数人学了个皮毛

扔物线 开源中国 6月13日

听说……Kotlin 可以用 Lambda？

```
val sum = { x: Int, y: Int -> x + y }
```

不错不错。Java 8 也有 Lambda，挺好用的。

听说……Kotlin 的 Lambda 还能当函数参数？

```
items.fold(0, {
  acc: Int, i: Int ->
  print("acc = $acc, i = $i, ")
  val result = acc + i
  println("result = $result")
  result
})
```

啊挺好挺好，我也来写一个！

```
items.map(num -> {
  num * 2
})
```

哎，报错了？我改！

```
items.map(num: Int -> {
  num * 2
})
```

哎？

我……再改？

```
items.map(num: (Int) -> {
  num * 2
})
```

我……再……改？

```
items.map((num: Int) -> {
  num * 2
})
```

啊！！！！！！！！！！！

Kotlin 的高阶函数

Kotlin 很方便，但有时候也让人头疼，而且越方便的地方越让人头疼，比如 Lambda 表达式。很多人因为 Lambda 而被 Kotlin 吸引，但很多人也因为 Lambda 而被 Kotlin 吓跑。其实大多数已经用了很久 Kotlin 的人，对 Lambda 也只会简单使用而已，甚至相当一部分人不靠开发工具的自动补全功能，根本就完全不会写 Lambda。今天我就来跟大家唠一唠 Lambda。不过，要讲 Lambda，我们得先从 Kotlin 的高阶函数——Higher-Order Function 说起。

在 Java 里，如果你有一个 a 方法需要调用另一个 b 方法，你在里面调用就可以；

```
1.  int a(){
```

```
2.  return b(1);
3.  }
4.  a();
```

而如果你想在 a 调用时动态设置 b 方法的参数，你就得把参数传给 a，再从 a 的内部把参数传给 b：

```
1.  int a(int param){
2.  return b(param);
3.  }
4.  a(1);// 内部调用 b(1)
5.  a(2);// 内部调用 b(2)
```

这都可以做到，不过……如果我想动态设置的不是方法参数，而是方法本身呢？比如我在 a 的内部有一处对别的方法的调用，这个方法可能是 b，可能是 c，不一定是谁，我只知道，我在这里有一个调用，它的参数类型是 int，返回值类型也是 int，而具体在 a 执行的时候内部调用哪个方法，我希望可以动态设置：

```
1.  int a(??? method){
2.  return method(1);
3.  }
4.  a(method1);
5.  a(method2);
```

或者说，我想把方法作为参数传到另一个方法里，这个……可以做到吗？

不行，也行。在 Java 里是不允许把方法作为参数传递的，但是我们有一个历史悠久的变通方案：接口。我们可以通过接口的方式来把方法包装起来：

```
1.  publicinterfaceWrapper{
2.  int method(int param);
3.  }
```

然后把这个接口的类型作为外部方法的参数类型：

```
1.  int a(Wrapper wrapper){
2.  return wrapper.method(1);
3.  }
```

在调用外部方法时，传递接口的对象来作为参数：

```
1.  a(wrapper1);
2.  a(wrapper2);
```

如果到这里你觉得听晕了，我换个写法你再感受一下：

我们在用户发生点击行为的时候会触发点击事件：

```
// 注：这是简化后的代码，不是 View.java 类的源码
publicclassView{
OnClickListener mOnClickListener;
...
publicvoid onTouchEvent(MotionEvent e){
...
    mOnClickListener.onClick(this);
...
}
}
```

所谓的点击事件，最核心的内容就是调用内部的一个 OnClickListener 的 onClick() 方法：

```
publicinterfaceOnClickListener{
void onClick(View v);
}
```

而所谓的这个 OnClickListener 其实只是一个壳，它的核心全在内部那个 onClick() 方法。换句话说，我们传过来一个 OnClickListener：

```
OnClickListener listener1 =newOnClickListener(){
@Override
void onClick(View v){
    doSomething();
}
};
view.setOnClickListener(listener1);
```

本质上其实是传过来一个可以在稍后被调用的方法（onClick()）。只不过因为 Java 不允许传递方法，所以我们才把它包进了一个对象里来进行传递。

而在 Kotlin 里面，函数的参数也可以是函数类型的：

```
fun a(funParam:Fun):String{
return funParam(1);
}
```

当一个函数含有函数类型的参数的时候——这句话有点绕啊——如果你调用它，你就可以——当然你也必须——传入一个函数类型的对象给它；

```
fun b(param:Int):String{
return param.toString()
}
a(b)
```

不过在具体的写法上没有我的示例这么粗暴。

首先我写的这个 Fun 作为函数类型其实是错的，Kotlin 里并没有这么一种类型来标记这个变量是个「函数类型」。因为函数类型不是一「个」类型，而是一「类」类型，因为函数类型可以有各种各样不同的参数和返回值的类型的搭配，这些搭配属于不同的函数类型。例如，无参数无返回值（() -> Unit）和单 Int 型参数返回 String （Int -> String）是两种不同的类型，这个很好理解，就好像 Int 和 String 是两个不同的类型。所以不能只用 Fun 这个词来表示「这个参数是个函数类型」，就好像不能用 Class 这个词来表示「这个参数是某个类」，因为你需要指定，具体是哪种函数类型，或者说这个函数类型的参数，它的参数类型是什么、返回值类型是什么，而不能笼统地一句说「它是函数类型」就完了。

所以对于函数类型的参数，你要指明它有几个参数、参数的类型是什么以及返回值类型是什么，那么写下来就大概是这个样子：

```
fun a(funParam:(Int)->String):String{
return funParam(1)
}
```

看着有点可怕。但是只有这样写，调用的人才知道应该传一个怎样的函数类型的参数给你。

同样的，函数类型不只可以作为函数的参数类型，还可以作为函数的返回值类型：

```
fun c(param:Int):(Int)->Unit{
...
}
```

这种「参数或者返回值为函数类型的函数」，在 Kotlin 中就被称为「高阶函数」——Higher-Order Functions。

这个所谓的「高阶」，总给人一种神秘感：阶是什么？哪里高了？其实没有那么复杂，高阶函数这个概念源自数学中的高阶函数。在数学里，如果一个函数使用函数作为它的参数或者结果，它就被称作是一个「高阶函数」。比如求导就是一个典型的例子：你对 f(x) = x 这个函数求导，结果是 1；对 $f(x) = x^2$ 这个函数求导，结果是 2x。很明显，求导函数的参数和结果都是函数，其中 f(x) 的导数是 1 这其实也是一个函数，只不过是一个结果恒为 1 的函数，所以——啊讲岔了，总之， Kotlin 里，这种参数有函数类型或者返回值是函数类型的函数，都叫做高阶函数，这只是个对这一类函数的称呼，没有任何特殊性，Kotlin 的高阶函数没有任何特殊功能，这是我想说的。

另外，除了作为函数的参数和返回值类型，你把它赋值给一个变量也是可以的。

不过对于一个声明好的函数，不管是你要把它作为参数传递给函数，还是要把它赋值给变量，都得在函数名的左边加上双冒号才行：

```
a(::b)
val d =::b
```

这……是为什么呢？

双冒号 ::method 到底是什么？

如果你上网搜，你会看到这个双冒号的写法叫做函数引用 Function Reference，这是 Kotlin 官方的说法。但是这又表示什么意思？表示它指向上面的函数？那既然都是一个东西，为什么不直接写函数名，而要加两个冒号呢？

因为加了两个冒号，这个函数才变成了一个对象。

什么意思？

Kotlin 里「函数可以作为参数」这件事的本质，是函数在 Kotlin 里可以作为对象存在——因为只有对象才能被作为参数传递啊。赋值也是一样道理，只有对象才能被赋值给变量啊。但 Kotlin 的函数本身的性质又决定了它没办法被当做一个对象。那怎么办呢？Kotlin 的选择是，那就创建一个和函数具有相同功能的对象。怎么创建？使用双冒号。

在 Kotlin 里，一个函数名的左边加上双冒号，它就不表示这个函数本身了，而表示一个对象，或者说一个指向对象的引用，但，这个对象可不是函数本身，而是一个和这个函数具有相同功能的对象。

怎么个相同法呢？你可以怎么用函数，就能怎么用这个加了双冒号的对象：

```
b(1)// 调用函数
d(1)// 用对象 a 后面加上括号来实现 b() 的等价操作
(::b)(1)// 用对象 :b 后面加上括号来实现 b() 的等价操作
```

但我再说一遍，这个双冒号的这个东西，它不是一个函数，而是一个对象，一个函数类型的对象。

对象是不能加个括号来调用的，对吧？但是函数类型的对象可以。为什么？因为这其实是个假的调用，它是 Kotlin 的语法糖，实际上你对一个函数类型的对象加括号、加参数，它真正调用的是这个对象的 invoke() 函数：

```
d(1)// 实际上会调用 d.invoke(1)
```

```
(::b)(1)// 实际上会调用 (::b).invoke(1)
```

所以你可以对一个函数类型的对象调用 invoke()，但不能对一个函数这么做：

```
b.invoke(1)// 报错
```

为什么？因为只有函数类型的对象有这个自带的 invoke() 可以用，而函数，不是函数类型的对象。那它是什么类型的？它什么类型也不是。函数不是对象，它也没有类型，函数就是函数，它和对象是两个维度的东西。

包括双冒号加上函数名的这个写法，它是一个指向对象的引用，但并不是指向函数本身，而是指向一个我们在代码里看不见的对象。这个对象复制了原函数的功能，但它并不是原函数。

> 这个……是底层的逻辑，但我知道这个有什么用呢？

这个知识能帮你解开 Kotlin 的高阶函数以及接下来我马上要讲的匿名函数、Lambda 相关的大部分迷惑。

比如我在代码里有这么几行：

```
fun b(param:Int):String{
return param.toString()
}
val d =::b
```

那我如果想把 d 赋值给一个新的变量 e：

```
val e = d
```

我等号右边的 d，应该加双冒号还是不加呢？

不用试，也不用搜，想一想：这是个赋值操作对吧？赋值操作的右边是个对象对吧？d 是对象吗？当然是了，b 不是对象是因为它来自函数名，但 d 已经是个对象了，所以直接写就行了。

匿名函数

我们继续讲。

要传一个函数类型的参数，或者把一个函数类型的对象赋值给变量，除了用双冒号来拿现成的函数使用，你还可以直接把这个函数挪过来写：

```
a(fun b(param:Int):String{
return param.toString()
});
val d = fun b(param:Int):String{
return param.toString()
}
```

另外，这种写法的话，函数的名字其实就没用了，所以你可以把它省掉：

```
a(fun(param:Int):String{
return param.toString()
});
val d = fun(param:Int):String{
return param.toString()
}
```

这种写法叫做匿名函数。为什么叫匿名函数？很简单，因为它没有名字呗，对吧。等号左边的不是函数的名字啊，它是变量的名字。这个变量的类型是一种函数类型，具体到我们的示例代码来说是一种只有一个参数、参数类型是 Int、并且返回值类型为 String 的函数类型。

另外呢，其实刚才那种左边右边都有名字的写法，Kotlin 是不允许的。右边的函数既然要名字也没有用，Kotlin 干脆就不许它有名字了。

所以，如果你在 Java 里设计一个回调的时候是这么设计的：

```
publicinterfaceOnClickListener{
void onClick(View v);
}
publicvoid setOnClickListener(OnClickListener listener){
this.listener = listener;
}
```

使用的时候是这么用的：

```
view.setOnClickListener(newOnClickListener(){
@Override
void onClick(View v){
    switchToNextPage();
}
});
```

到了 Kotlin 里就可以改成这么写了：

```
fun setOnClickListener(onClick:(View)->Unit){
```

```
2.   this.onClick = onClick
3.   }
4.   view.setOnClickListener(fun(v:View):Unit){
5.     switchToNextPage()
6.   })
```

简单一点哈？另外大多数（几乎所有）情况下，匿名函数还能更简化一点，写成 Lambda 表达式的形式：

```
1.   view.setOnClickListener({ v:View->
2.     switchToNextPage()
3.   })
```

Lambda 表达式

终于讲到 Lambda 了。

如果 Lambda 是函数的最后一个参数，你可以把 Lambda 写在括号的外面：

```
1.   view.setOnClickListener(){ v:View->
2.     switchToNextPage()
3.   }
```

而如果 Lambda 是函数唯一的参数，你还可以直接把括号去了：

```
1.   view.setOnClickListener { v:View->
2.     switchToNextPage()
3.   }
```

另外，如果这个 Lambda 是单参数的，它的这个参数也省略掉不写：

```
1.   view.setOnClickListener {
2.     switchToNextPage()
3.   }
```

哎，不错，单参数的时候只要不用这个参数就可以直接不写了。

其实就算用，也可以不写，因为 Kotlin 的 Lambda 对于省略的唯一参数有默认的名字：it：

```
1.   view.setOnClickListener {
2.     switchToNextPage()
3.     it.setVisibility(GONE)
```

```
4.  }
```

有点爽哈？不过我们先停下想一想：这个 Lambda 这也不写那也不写的……它不迷茫吗？它是怎么知道自己的参数类型和返回值类型的？

靠上下文的推断。我调用的函数在声明的地方有明确的参数信息吧？

```
1.  fun setOnClickListener(onClick:(View)->Unit){
2.  this.onClick = onClick
3.  }
```

这里面把这个参数的参数类型和返回值写得清清楚楚吧？所以 Lambda 才不用写的。

所以，当你要把一个匿名函数赋值给变量而不是作为函数参数传递的时候：

```
1.  val b = fun(param:Int):String{
2.  return param.toString()
3.  }
```

如果也简写成 Lambda 的形式：

```
1.  val b ={ param:Int->
2.  return param.toString()
3.  }
```

就不能省略掉 Lambda 的参数类型了：

```
1.  val b ={
2.  return it.toString()// it 报错
3.  }
```

为什么？因为它无法从上下文中推断出这个参数的类型啊！

如果你出于场景的需求或者个人偏好，就是想在这里省掉参数类型，那你需要给左边的变量指明类型：

```
1.  val b:(Int)->String={
2.  return it.toString()// it 可以被推断出是 Int 类型
3.  }
```

另外 Lambda 的返回值不是用 return 来返回，而是直接取最后一行代码的值：

```
val b:(Int)->String={
  it.toString()// it 可以被推断出是 Int 类型
}
```

这个一定注意，Lambda 的返回值别写 return，如果你写了，它会把这个作为它外层的函数的返回值来直接结束外层函数。当然如果你就是想这么做那没问题啊，但如果你是只是想返回 Lambda，这么写就出错了。

另外因为 Lambda 是个代码块，它总能根据最后一行代码来推断出返回值类型，所以它的返回值类型确实可以不写。实际上，Kotlin 的 Lambda 也是写不了返回值类型的，语法上就不支持。

现在我再停一下，我们想想：匿名函数和 Lambda……它们到底是什么？

Kotlin 里匿名函数和 Lambda 表达式的本质

我们先看匿名函数。它可以作为参数传递，也可以赋值给变量，对吧？

但是我们刚才也说过了函数是不能作为参数传递，也不能赋值给变量的，对吧？

那为什么匿名函数就这么特殊呢？

因为 Kotlin 的匿名函数不——是——函——数。它是个对象。匿名函数虽然名字里有「函数」两个字，包括英文的原名也是 Anonymous Function，但它其实不是函数，而是一个对象，一个函数类型的对象。它和双冒号加函数名是一类东西，和函数不是。

所以，你才可以直接把它当做函数的参数来传递以及赋值给变量：

```
a(fun (param:Int):String{
return param.toString()
});
val a = fun (param:Int):String{
return param.toString()
}
```

同理，Lambda 其实也是一个函数类型的对象而已。你能怎么使用双冒号加函数名，就能怎么使用匿名函数，以及怎么使用 Lambda 表达式。

这，就是 Kotlin 的匿名函数和 Lambda 表达式的本质，它们都是函数类型的对象。Kotlin 的 Lambda 跟 Java 8 的 Lambda 是不一样的，Java 8 的 Lambda 只是一种便捷写法，本质上并没有功能上的突破，而 Kotlin 的 Lambda 是实实在在的对象。

在你知道了在 Kotlin 里「函数并不能传递，传递的是对象」和「匿名函数和 Lambda 表达式其实都是对象」这些本质之后，你以后去写 Kotlin 的高阶函数会非常轻松非常舒畅。

Kotlin 官方文档里对于双冒号加函数名的写法叫 Function Reference 函数引用，故意引导大家认为这个引用是指向原函数的，这是为了简化事情的逻辑，让大家更好上手 Kotlin；但这种逻辑是有毒的，一旦你信了它，你对于匿名函数和 Lambda 就怎么也搞不清楚了。

对比 Java 的 Lambda

再说一下 Java 的 Lambda。对于 Kotlin 的 Lambda，有很多从 Java 过来的人表示「好用好用但不会写」。这是一件很有意思的事情：你都不会写，那你是怎么会用的呢？Java 从 8 开始引入了对 Lambda 的支持，对于单抽象方法的接口——简称 SAM 接口，Single Abstract Method 接口——对于这类接口，Java 8 允许你用 Lambda 表达式来创建匿名类对象，但它本质上还是在创建一个匿名类对象，只是一种简化写法而已，所以 Java 的 Lambda 只靠代码自动补全就基本上能写了。而 Kotlin 里的 Lambda 和 Java 本质上就是不同的，因为 Kotlin 的 Lambda 是实实在在的函数类型的对象，功能更强，写法更多更灵活，所以很多人从 Java 过来就有点搞不明白了。

另外呢，Kotlin 是不支持使用 Lambda 的方式来简写匿名类对象的，因为我们有函数类型的参数嘛，所以这种单函数接口的写法就直接没必要了。那你还支持它干嘛？

不过当和 Java 交互的时候，Kotlin 是支持这种用法的：当你的函数参数是 Java 的单抽象方法的接口的时候，你依然可以使用 Lambda 来写参数。但这其实也不是 Kotlin **增加**了功能，而是对于来自 Java 的单抽象方法的接口，Kotlin 会为它们额外创建一个把参数替换为函数类型的桥接方法，让你可以间接地创建 Java 的匿名类对象。

这就是为什么，你会发现当你在 Kotlin 里调用 View.java 这个类的 setOnClickListener() 的时候，可以传 Lambda 给它来创建 OnClickListener 对象，但你照着同样的写法写一个 Kotlin 的接口，你却不能传 Lambda。因为 Kotlin 期望我们直接使用函数类型的参数，而不是用接口这种折中方案。

总结

好，这就是 Kotlin 的高阶函数、匿名函数和 Lambda。简单总结一下：

- 在 Kotlin 里，有一类 Java 中不存在的类型，叫做「函数类型」，这一类类型的对象在可以当函数来用的同时，还能作为函数的参数、函数的返回值以及赋值给变量；
- 创建一个函数类型的对象有三种方式：双冒号加函数名、匿名函数和 Lambda；
- 一定要记住：双冒号加函数名、匿名函数和 Lambda 本质上都是函数类型的对象。在 Kotlin 里，匿名函数不是函数，Lambda 也不是什么玄学的所谓「它只是个代码块，没法归

类」，Kotlin 的 Lambda 可以归类，它属于函数类型的对象。

当然了这里面的各种细节还有很多，这个你可以自己学去，我就不管你了。下期内容是 Kotlin 的扩展属性和扩展函数，关注我，不错过我的任何新内容。大家拜拜~

视频版本

本文在 B 站有对应的视频，如果你喜欢看视频版本，可以在哔哩哔哩搜索「扔物线」或者直接扫描下面的二维码前往观看：

B 站搜索「扔物线」
或扫描二维码
观看本文视频

作者介绍

扔物线（朱凯），中国唯一得到 Google 官方认证的 Android 开发专家和 Kotlin 开发专家（GDE），前 Flipboard Android 工程师，Android 高级技术分享网站 HenCoder (hencoder.com) 作者 和 Kotlin 学习网站码上开学 (kaixue.io) 创办人。

推荐阅读

推理引擎Paddle Inference改造三要点，ERNIE时延降低81.3%
又说骚话，Linus再次拒绝Intel CPU漏洞补丁
AI换脸或受《民法典》严监管
PHP 8性能怎么样？
AI复原100年前京城老视频，靠这三个开源工具