

题目解答 | 期中考试版本参考实现

2022-02-28 朱涛

《朱涛 · Kotlin编程第一课》

[课程介绍 >](#)



讲述：朱涛

时长 06:34 大小 6.03M



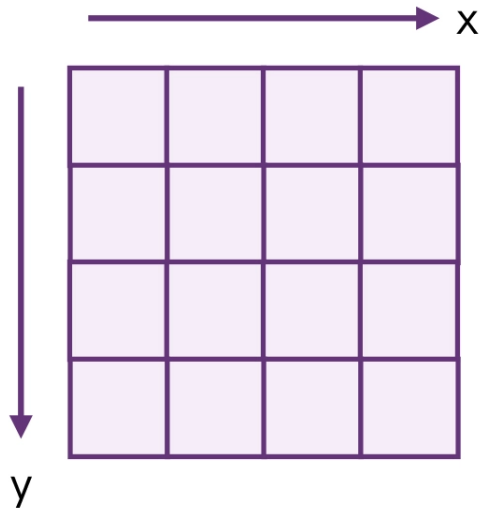
你好，我是朱涛。上节课我给你布置了一份考试题，你完成得怎么样了呢？这节课呢，我会来告诉你我是如何用 **Kotlin** 来做这个图片处理程序的，供你参考。

由于上节课我们已经做好了前期准备，所以这里我们直接写代码就行了。

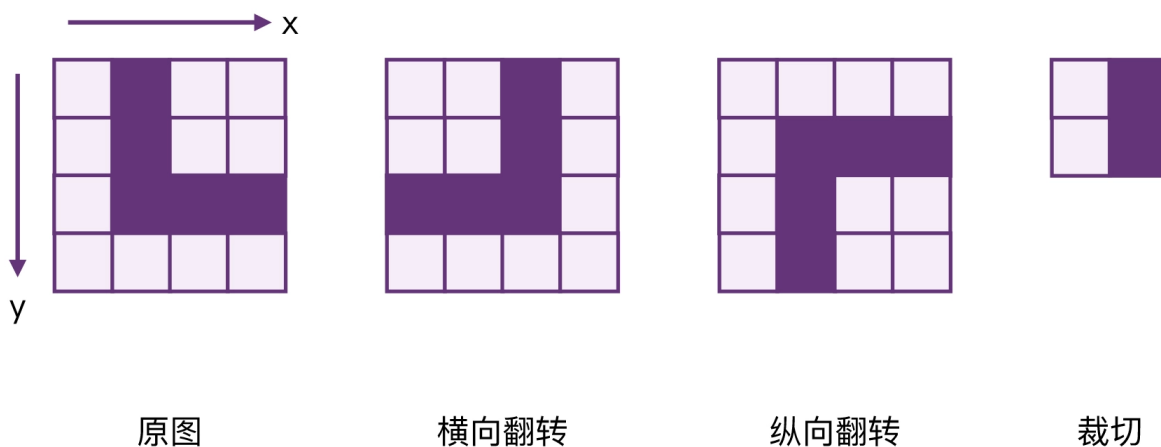
1.0 版本

对于图片反转和裁切的这个问题，如果一开始你就去想象一个大图片，里面有几万个像素点，那你可能会被吓到。但是，如果你将数据规模缩小，再来分析的话，你会发现这个问题其实很简单。

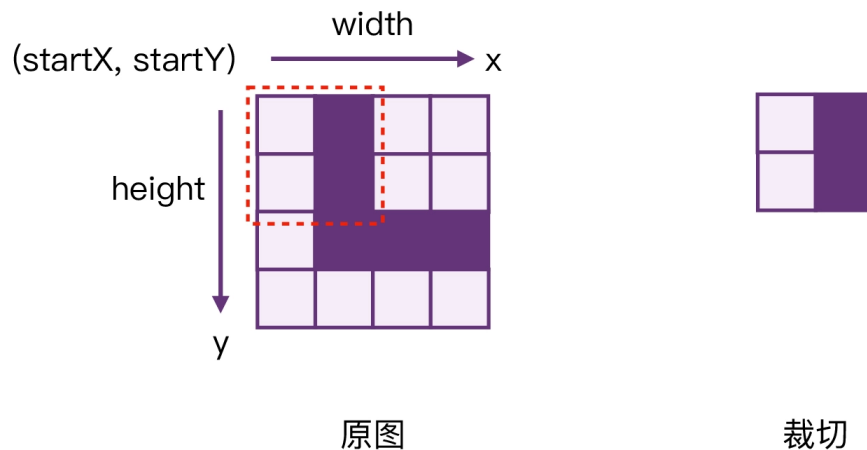
这里，我们就以一张 **4X4** 像素的照片为例来分析一下。



这其实就相当于一个抽象的模型，如果我们基于这张 4X4 的照片，继续分析翻转和裁切，就会容易很多。我们可以来画一个简单的图形：



上面这张图，从左到右分别是原图、横向翻转、纵向翻转、裁切。其中，翻转看起来是要复杂一些，而裁切是最简单的。



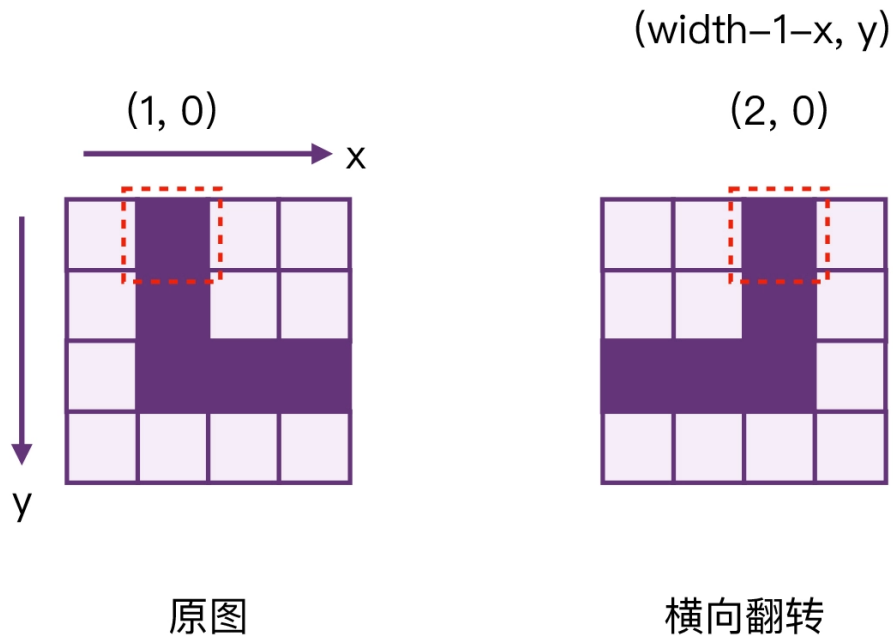
我们先来处理裁切。对于裁切，其实只需要将图片当中某个部分的像素拷贝到内存，然后存储成为一张新图片就行了。

[复制代码](#)

```
1 /**
2  * 图片裁切
3  */
4 fun Image.crop(startY: Int, startX: Int, width: Int, height: Int): Image {
5     val pixels = Array(height) { y ->
6         Array(width) { x ->
7             getPixel(startY + y, startX + x)
8         }
9     }
10    return Image(pixels)
11 }
```

以上代码中，我们创建了一个新数组 `pixels`，它的创建方式是通过 `Lambda` 来实现的，而 `Lambda` 当中最关键的逻辑，就是 `getPixel(startY + y, startX + x)`，也就是从原图当中取像素点。

这代码是不是比你想象中简单很多？其实，图片的翻转也是一样的。只要我们能**找出坐标的对应关系**，代码也非常简单。



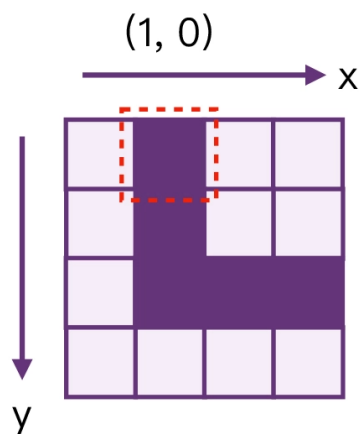
可以看到，对于原图的 $(1, 0)$ 这个像素点来说，它横向翻转以后就变成了 $(2, 0)$ 。所以，对于 (x, y) 坐标来说，横向翻转以后，就应该变成 $(\text{width}-1-x, y)$ 。找到这个对应关系以后，我们就直接抄代码了！

复制代码

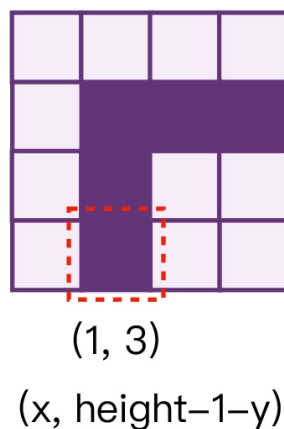
```
1  /**
2   * 横向翻转图片
3   */
4  fun Image.flipHorizontal(): Image {
5      val pixels = Array(height()) { y ->
6          Array(width()) { x ->
7              getPixel(y, width() - 1 - x)
8          }
9      }
10     return Image(pixels)
11 }
```

可见，以上这段代码几乎跟裁切是一模一样的，只是说，裁切要限制宽高，而翻转则是跟原图保持一致。

看到这里，相信你也马上就能想明白纵向翻转的代码该如何写了！



原图



纵向翻转



我们还是以 $(1, 0)$ 这个像素点为例，在纵向翻转以后就变成了 $(1, 3)$ ，它们的转换规则是 $(x, \text{height}-1-y)$ 。

复制代码

```
1  /**
2   * 纵向翻转图片
3   */
4  fun Image.flipVertical(): Image {
5      val pixels = Array(height()) { y ->
6          Array(width()) { x ->
7              getPixel(height() - 1 - y, x) // 改动这里
8          }
9      }
10     return Image(pixels)
11 }
```

所以说，只要我们能找到中间的转换关系，纵向翻转的代码，只需要在横向翻转的基础上，改动一行即可。

单元测试

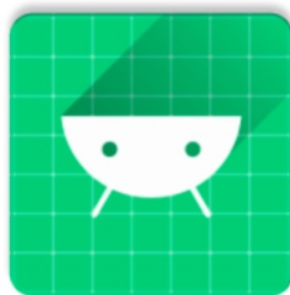
其实针对图像算法的单元测试，我们最好的方式，就是准备一些现有的图片案例。比如说，我们随便找一张图，用其他的软件工具，对它进行翻转、裁切，然后存储起来。比如还是这四张图：



原图



横向翻转

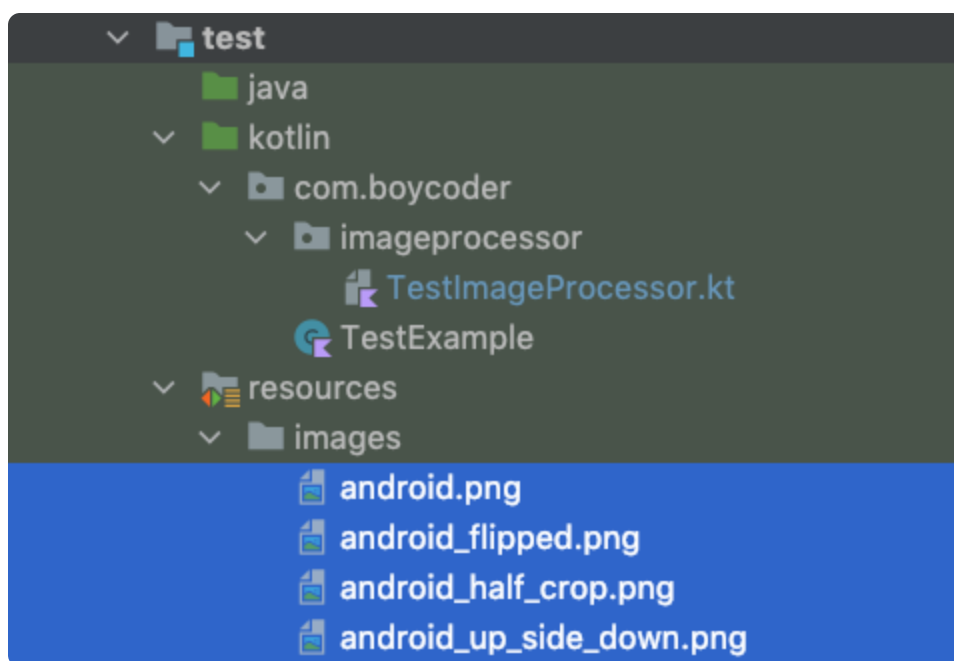


纵向翻转



裁切

我们可以把处理后的图片保存在单元测试的文件夹下，方便我们写对应的测试用例。



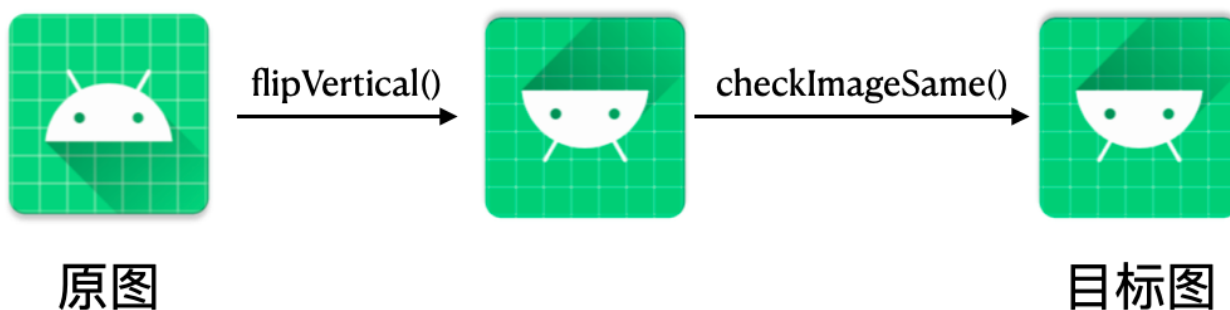
那么，有了这些图片之后，我想你应该就能想到要怎么办了。这时候，你只需要写一个图片像素对比的方法 `checkImageSame()` 就好办了。

复制代码

```
1 private fun checkImageSame(picture: Image, expected: Image) {
2     assertEquals(picture.height(), expected.height())
3     assertEquals(picture.width(), expected.width())
4     for (row in 0 until picture.height()) {
5         for (column in 0 until picture.width()) {
6             val actualPixel = picture.getPixel(row, column)
7             val expectedPixel = expected.getPixel(row, column)
8             assertEquals(actualPixel, expectedPixel)
9         }
10    }
11 }
```

其实，这个函数的思路也很简单，就是逐个对比两张图片之间的像素，看看它们是不是一样的，如果两张图所有的像素都一样，那肯定就是一样的。

有了这个方法以后，我们就可以快速实现单元测试代码了。整体流程大致如下：



复制代码

```
1 @Test
2 fun testCrop() {
3     val image = loadImage(File("${TEST_BASE_PATH}android.png"))
4     val height = image.height() / 2
5     val width = image.width() / 2
6     val target = loadImage(File("${TEST_BASE_PATH}android_half_crop.png"))
7
8     val crop = image.crop(0, 0, width, height)
9     checkImageSame(crop, target)
10 }
11
12 @Test
13 fun testFlipVertical() {
14     val origin = loadImage(File("${TEST_BASE_PATH}android.png"))
15     val target = loadImage(File("${TEST_BASE_PATH}android_up_side_down.png"))
16     val flipped = origin.flipVertical()
17     checkImageSame(flipped, target)
18 }
19
20 @Test
21 fun testFlipHorizontal() {
22     val origin = loadImage(File("${TEST_BASE_PATH}android.png"))
23     val target = loadImage(File("${TEST_BASE_PATH}android_flipped.png"))
24     val flipped = origin.flipHorizontal()
25     checkImageSame(flipped, target)
26 }
```

有了单元测试，我们就再也不用担心以后改代码的时候，不小心改出问题了。

好，那么到这里，1.0 版本就算是完成了。我们接着来看看 2.0 版本。

2.0 版本

2.0 版本的任务，我们需要支持下载网络上面的图片，并且还要能够存起来。由于这是一个比较耗时的操作，我们希望它是一个挂起函数。

关于下载 HTTP 的图片，其实，我们借助 OkHttp 就可以简单实现。下面我们来看看代码。

补充：为了不偏离主题，我们不考虑 HTTPS 的问题。

 复制代码

```
1 fun downloadSync() {
2     logX("Download start!")
3     val okHttpClient = OkHttpClient().newBuilder()
4         .connectTimeout(10L, TimeUnit.SECONDS)
5         .readTimeout(10L, TimeUnit.SECONDS)
6         .build()
7
8     val request = Request.Builder().url(url).build()
9     val response = okHttpClient.newCall(request).execute()
10
11     val body = response.body
12     val responseCode = response.code
13
14     if (responseCode >= HttpURLConnection.HTTP_OK &&
15         responseCode < HttpURLConnection.HTTP_MULT_CHOICE &&
16         body != null
17     ) {
18         // 1, 注意这里
19         body.byteStream().apply {
20             outputStream().use { fileOut ->
21                 copyTo(fileOut)
22             }
23         }
24     }
25     logX("Download finish!")
26 }
```


以上代码中，有一个地方是需要注意的，我用注释标记了。也就是当我们想要把网络流中的数据存起来的时候，我们可以借助 Kotlin 提供的 **IO 扩展函数** 快速实现，这样不仅方便，而且不用担心 `FileOutputStream` 调用 `close()` 的问题。这个部分的代码，在 Java 当中，是要写一堆模板代码的。

下载本身的功能实现以后，挂起函数的封装也就容易了。

 复制代码

```
1 suspend fun downloadImage(url: String, outputFile: File): Boolean {
2     return withContext(Dispatchers.IO) {
3         try {
4             downloadSync()
5         } catch (e: Exception) {
6             println(e)
7             // return@withContext 不可省略
8             return@withContext false
9         }
10        // return@withContext 可省略
11        return@withContext true
12    }
13 }
```

这里，我们可以直接用 **withContext**，让下载的任务直接分发到 IO 线程。

代码写到这里，2.0 版本要求的功能基本上就算是完成了。这样一来，我们就可以在 `main` 函数当中去调用它了。

 复制代码

```
1 fun main() = runBlocking {
2     val url = "http://xxxx.jpg"
3     val path = "${BASE_PATH}downloaded.png"
4
5     downloadImage(url, File(path))
6
7     loadImage(File(path))
8         .flipVertical()
9         .writeToFile(File("${BASE_PATH}download_flip_vertical.png"))
10
11     logX("Done")
12 }
13
14 // 将内存图片保存到硬盘
15 fun Image.writeToFile(outputFile: File): Boolean {
```

```

16         return try {
17             val width = width()
18             val height = height()
19             val image = BufferedImage(width, height, BufferedImage.TYPE_INT_RGB)
20             for (x in 0 until width) {
21                 for (y in 0 until height) {
22                     val awtColor = getPixel(y, x)
23                     image.setRGB(x, y, awtColor.rgb)
24                 }
25             }
26             ImageIO.write(image, "png", outputFile)
27             true
28         } catch (e: Exception) {
29             println(e)
30             false
31         }
32     }
33
34     /*
35     输出结果：
36     =====
37     Download start!
38     Thread:DefaultDispatcher-worker-1
39     =====
40     =====
41     Download finish!
42     Thread:DefaultDispatcher-worker-1
43     =====
44     =====
45     Done
46     Thread:main
47     =====
48     */

```

通过运行结果，我们会发现图片下载的任务，已经被分发到 IO 线程池了，而其他的代码仍然在主线程之上。

小结

其实，课程进行到这里，你就会发现，Kotlin 和 Java、C 之类的语言的编程方式是完全不一样的。Kotlin 提供了丰富的扩展函数，在很多业务场景下，Kotlin 是可以大大减少代码量的。

另外，你也会发现，当你熟悉 Kotlin 协程以后，它的使用一点都不难。对于上面的代码，我们通过 `withContext(Dispatchers.IO)` 就能切换线程，之后，我们就可以在协程作用域当中随意调用了！

思考题


你觉得，我们在 `downloadImage()` 这个挂起函数内部，直接写死 `Dispatchers.IO` 的方式好吗？如果换成下面这种写法，会不会更好？为什么？

 复制代码

```
1 suspend fun downloadImage(  
2     coroutineContext: CoroutineContext = Dispatchers.IO,  
3     url: String,  
4     outputFile: File  
5 ): Boolean {  
6  
7     return withContext(coroutineContext) {  
8         try {  
9             downloadSync()  
10        } catch (e: Exception) {  
11            println(e)  
12            return@withContext false  
13        }  
14        return@withContext true  
15    }  
16 }
```

分享给需要的人，Ta订阅超级会员，你最高得 **50** 元

Ta单独购买本课程，你将得 **20** 元

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 期中考试 | 用Kotlin实现图片处理程序

[下一篇](#) 19 | Channel：为什么说Channel是“热”的？

精选留言 (7)

 写留言



Allen

思考题中关于调度器的使用不太好，一般的使用方式是通过暴露参数的方式让使用者来传。

通过传参的方式有两个好处：

1. 增加了代码的灵活性和可用性。
2. 有利于单元测试。

作者回复: 嗯，各有利弊。



PoPlus

2022-03-01

暂时没想到 `context` 动态传入有什么好处，按理说下载图片只需要 IO 就行了，希望老师能解惑。

作者回复: 其实，这个问题没有标准答案。我只是提出这个问题，希望大家思考这中间的差异。因为，有的时候，更多的灵活性也并不意味着是好事。

动态传入的方式，有利有弊。利：更灵活；弊，调用方可能传错Context。

总的来说，还是要结合使用场景来分析。有些场景下，我们更加重视灵活性，就选择动态传入，有的场景下，我们都希望尽可能的不出问题。



Allen

2022-02-28

这个下载图片的实现和我们普通开线程下载的主要区别就是使用同步的写法（不需要写 `callback`）来实现了异步的操作。

作者回复: 是的。



Geek_Adr

2022-03-12

灵活会带来API难用，增加犯错的概率，但又不能把灵活丢掉

兼得的方法：先给一套默认最佳实现，满足大多数情况，使用者清晰知道默认实现不满足时再给灵活的方案

我认为可配有默认实现的方式更好

作者回复: 很好的思路。



白乾涛

2022-03-06

```
class Image(private val pixels: Array<Array<Color>>){
    val height: Int = pixels.size
    val width: Int = pixels[0].size
    private fun getPixel(y: Int, x: Int): Color = pixels[y][x]

    fun flipHorizontal() = changelImage { y, x -> getPixel(y, width - 1 - x) } //横向翻转图片

    fun flipVertical() = changelImage { y, x -> getPixel(height - 1 - y, x) } //纵向翻转图片

    fun crop(startY: Int, startX: Int, width: Int, height: Int): Image =
        tolImage(height, width) { y, x -> getPixel(startY + y, startX + x) } //图片裁切
}

fun Image.changelImage(init: (Int, Int) -> Color): Image = tolImage(height, width, init)

fun loadImage(imageFile: File): Image =
    ImageIO.read(imageFile)
        .let { tolImage(it.height, it.width) { x, y -> Color(it.getRGB(x, y)) } }

fun tolImage(height: Int, width: Int, init: (Int, Int) -> Color): Image =
    Array(height) { y -> // 创建一个 Array，元素类型也是 Array
        Array(width) { x -> // 创建一个 Array，元素类型是 Color
            init(x, y) // 每个元素都是在 init() 中初始化的
        }
    }.let { Image(it) }
```

作者回复: 这代码改进的很不错，赞！



曾帅

2022-03-02

这个问题 或许 可以根据项目来，有些项目可能有自己的 线程池 或者一些 生命周期 的要求，就可以根据业务需求进行传参进行处理。如果没有这些需求的话，直接用 IO 也是可以的。

作者回复: 是的，要结合具体的场景来分析。



L先生

2022-03-01

我觉得会好点，给调用者选择的权利，并且赋予默认值，方便单元测试

作者回复: 嗯，后者的灵活性更强。

