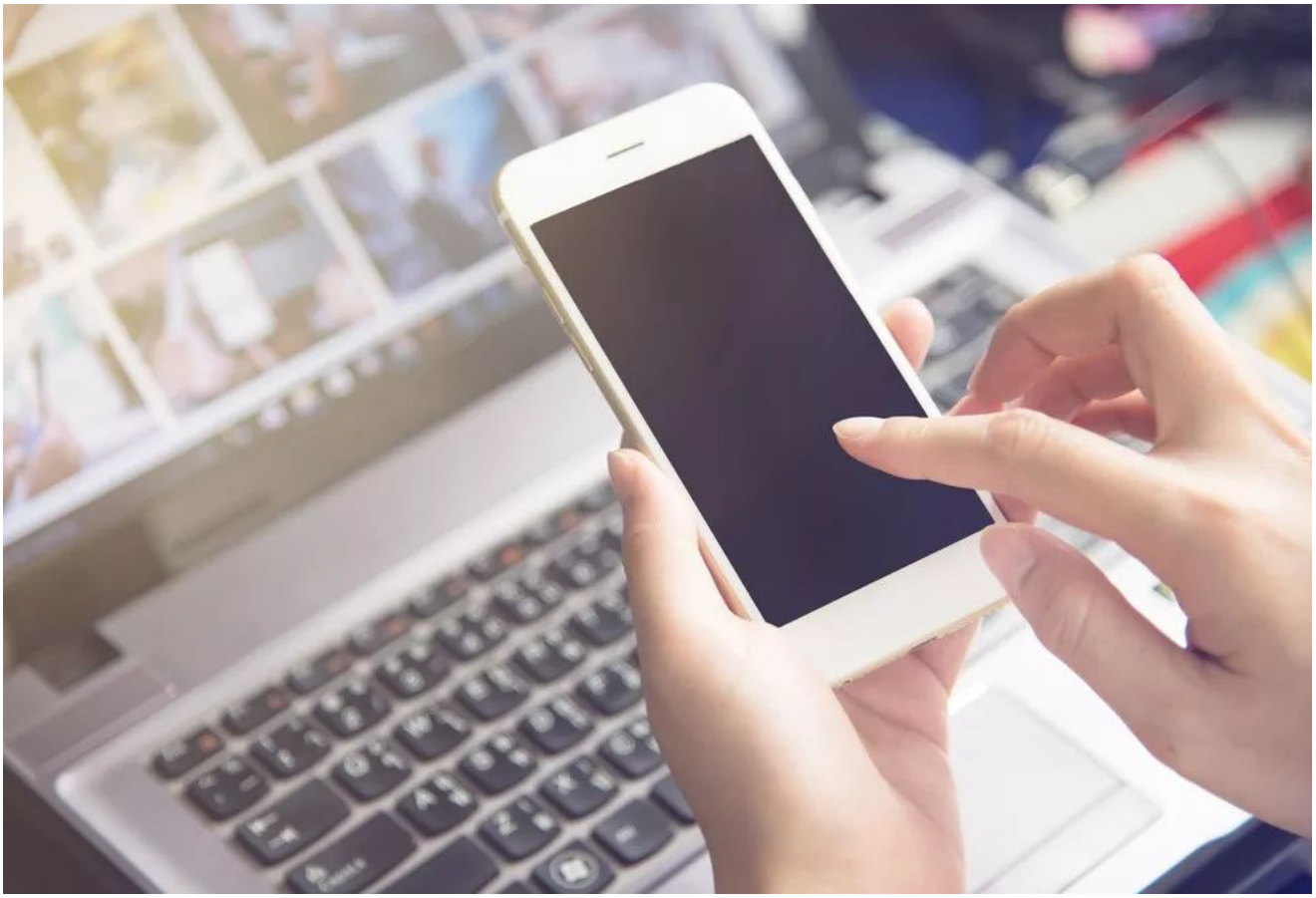
匠修

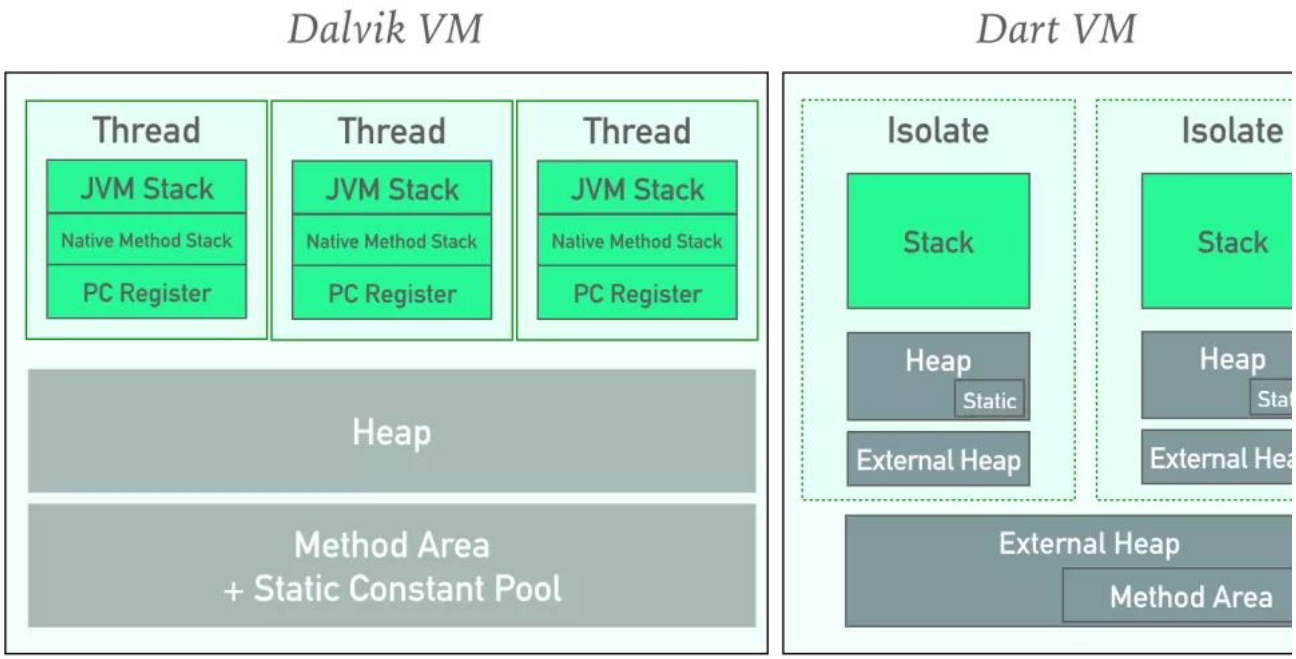


阿里妹导读：闲鱼技术团队一直在探索如何使用Flutter来统一移动App开发。移动设备上的资源有限，内存使用成了日常开发中的常见问题。那么，Flutter是如何使用内存，又会对Native App的内存带来哪些影响呢？本文将简单介绍Flutter内存机制，结合测试和闲鱼技术团队的开发实践，对普遍关心的Bitmap内存使用，View绘制内存使用方面做一些探索。

Dart RunTime简介

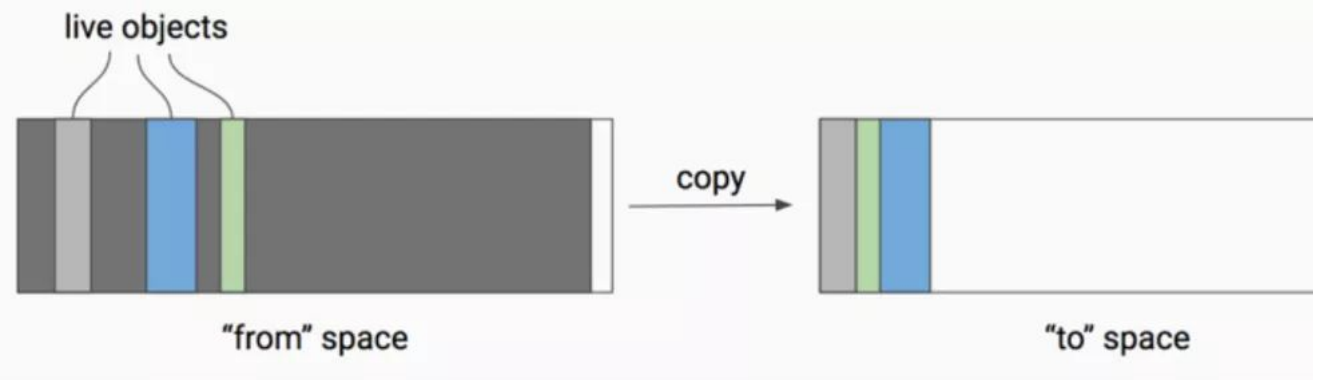
Flutter Framework使用Dart语言开发，所以App进程中需要一个Dart运行环境（VM），和Android Art一样，Flutter也对Dart源码做了AOT编译，直接将Dart源码编译成了本地字节码，没有了解释执行的过程，提升执行性能。这里重点关注Dart VM内存分配(Allocate)和回收(GC)相关的部分。

和Java显著不同的是Dart的"线程"(Isolate)是不共享内存的，各自的堆(Heap)和栈(Stack)都是隔离的，并且是各自独立GC的，彼此之间通过消息通道来通信。Dart天然不存在数据竞争和变量状态同步的问题，整个Flutter Framework Widget的渲染过程都运行在一个isolate中。

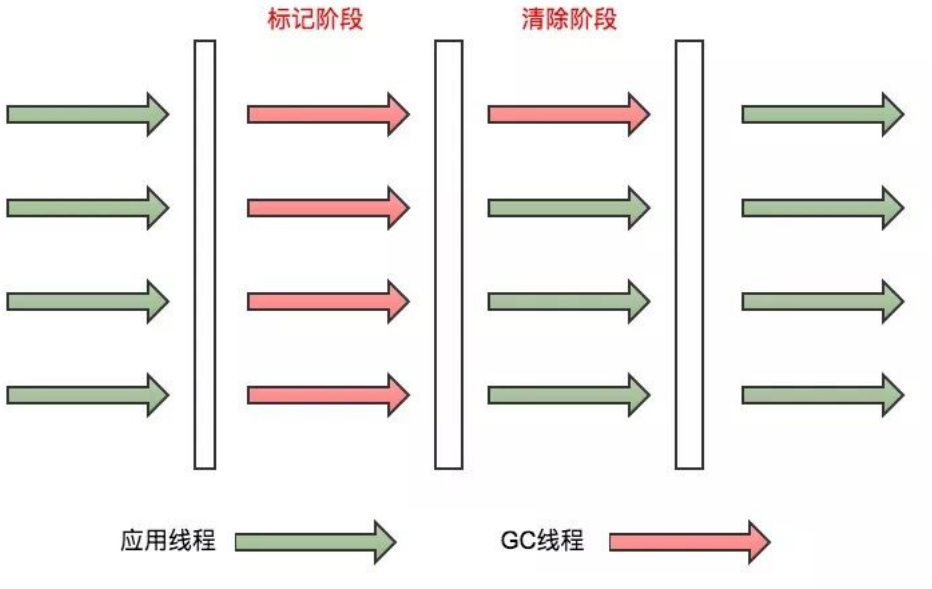


Dart VM将内存管理分为新生代(New Generation)和老年代(Old Generation)。

- 新生代(New Generation): 通常初次分配的对象都位于新生代中，该区域主要是存放内存较小并且生命周期较短的对象，比如局部变量。新生代会频繁执行内存回收(GC)，回收采用“复制-清除”算法，将内存分为两块(图中的from 和 to)，运行时每次只使用其中的一块(图中的from)，另一块备用(图中的to)。当发生GC时，将当前使用的内存块中存活的对象拷贝到备用内存块中，然后清除当前使用内存块，最后，交换两块内存的角色。



- 老年代(Old Generation): 在新生代的GC中“幸存”下来的对象，它们会被转移到老年代中。老年代存放生命力周期较长，内存较大的对象。老年代通常比新生代要大很多。老年代的GC回收采用“标记-清除”算法，分成标记和清除两个阶段。在标记阶段会触发停顿(stop the world)，多线程并发的完成对垃圾对象的标记，降低标记阶段耗时。在清理阶段，由GC线程负责清理回收对象，和应用线程同时执行，不影响应用运行。



可以看到，Dart VM借鉴了很多JVM的思路，Dart中产生内存泄露的方式也和Java类似，Java中很多排查内存泄露的思路和防止内存泄露的编程方法应该也可以借鉴过来。

Image内存初探

对图片的合理使用和优化是UI编程的重要部分，Flutter提供了Image Widget，我们可以方便地使用：

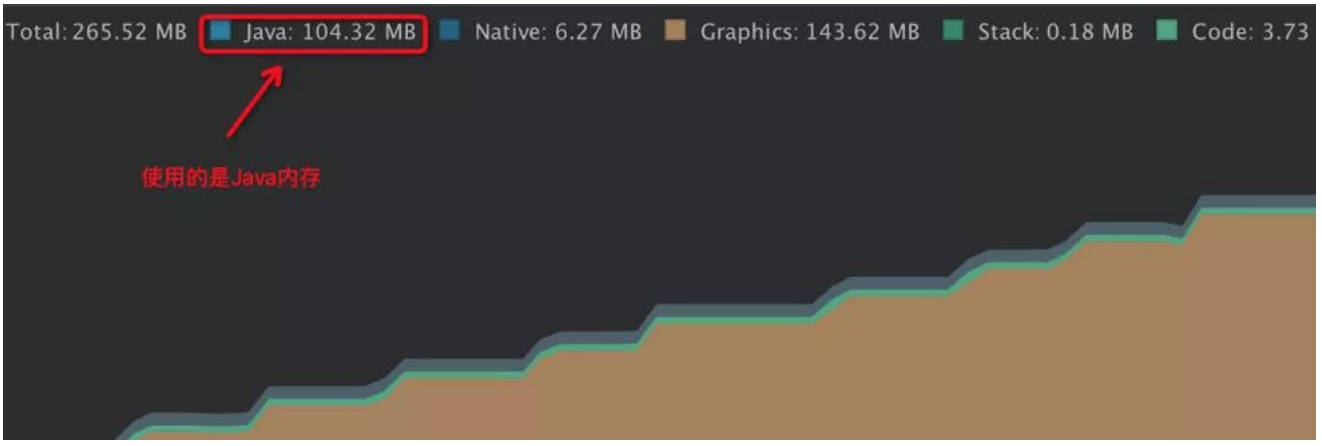
```
//使用本地图片
new Image.asset("images/xxxx.jpg");

//使用网络图片
new Image.network("https://xxxxxx");
```

我们知道Android将内存分为Java虚拟机内存和Native内存，各大厂商都对Java虚拟机内存有一个上限限制，到达上限就会触发OOM异常，而对Native内存的使用没有太严格的限制，现在的手机内存都很大，一般有较大的Native内存富余。那么Android中ImageView使用的是Java虚拟机内存还是Native内存呢？

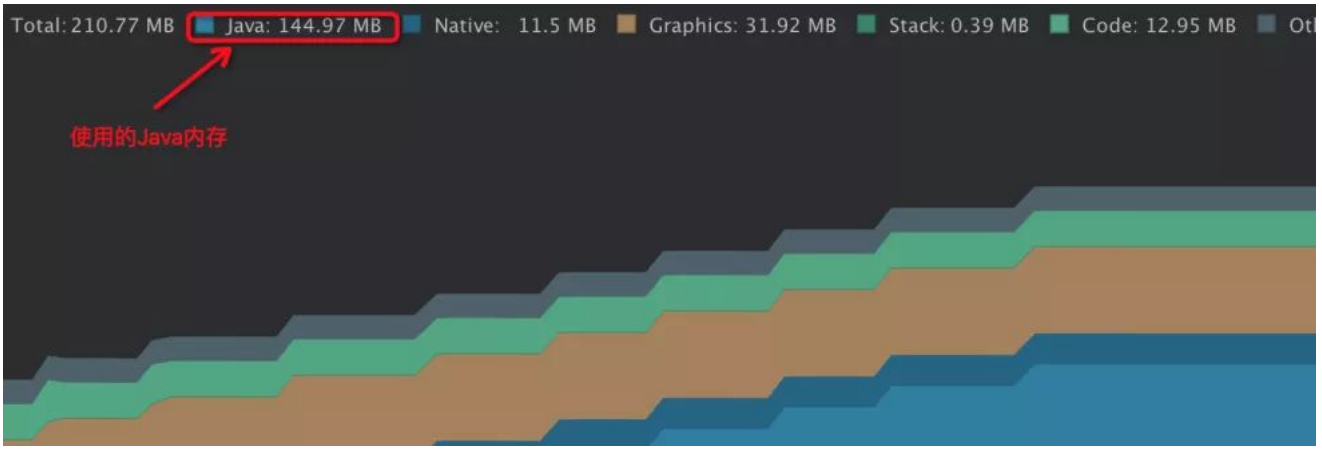
我们可以来做一个测试：在一个界面上，每点击一次，就在上面堆加一张图片。为了防止后面的图片完全覆盖前面的图片而出现优化的情况，每次都缩小几个像素，这样就不会出现完全覆盖。

打开Android Profiler，一张一张添加图片，观察内存数据。分别测试了Android的6.0，7.0和8.0系统，结果如下：

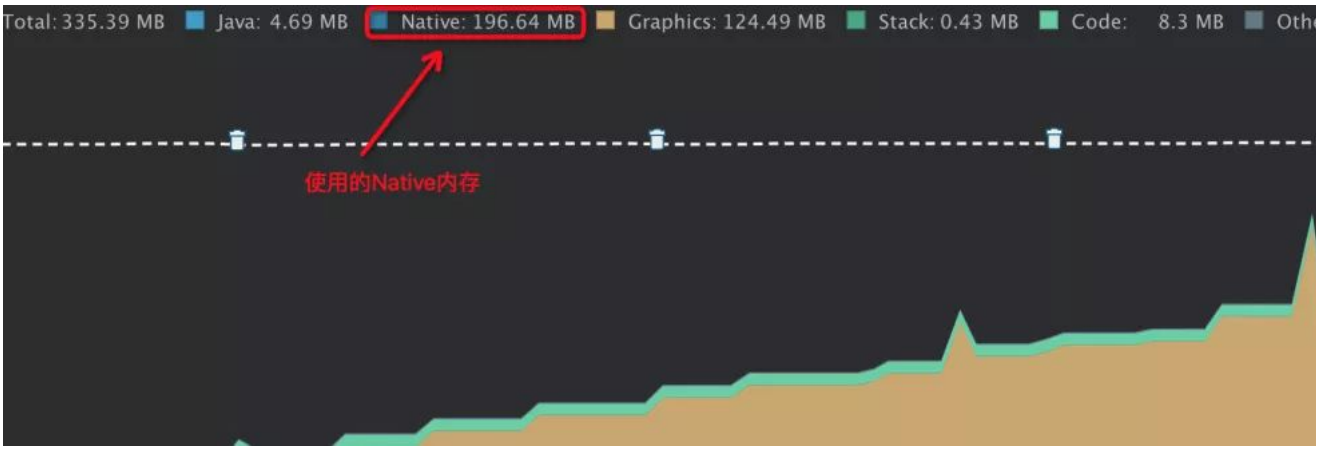
Android 6.0(Google Nextus5)



Android 7.0(Meizu pro5)



Android 8.0(Google pixel)



在测试中,随着图片一张张增加，Android 6.0 和 7.0都是Java部分的内存在增长，而Android 8.0则是Native部分的内存在增长。由此有结论，Android原生的ImageView在6.0和7.0版本中使用的Java虚拟机内存，而在Android 8.0中则使用的Native内存。

而Flutter Image Widget使用的是哪部分内存呢？我们用Flutter界面来做相同的测试。Flutter Engine的Debug版本和Release版本存在很大的性能差异，所以我们测试最好使用Release版本，但是，Release版本的Apk又不能使用Android profiler来观察内存，所以我们需要在Debug版本的Apk中打包一个Release版本的Flutter Engine, 可以修改flutter tool中的flutter.gradle来实现：

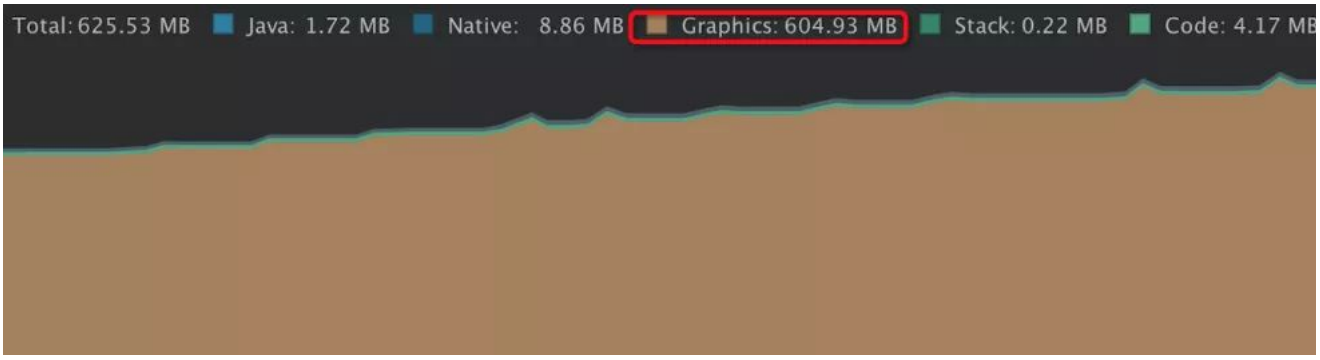
```
//不做判断，强制改为打包release版本的engine
private static String buildModeFor(buildType) {
    // if (buildType.name == "profile") {
    //     return "profile"
    // } else if (buildType.debuggable) {
    //     return "debug"
    // }
    return "release"
}
```

相同地，我们向Flutter界面中添加图片并用Android Profiler来观察内存,测试使用的dart代码：

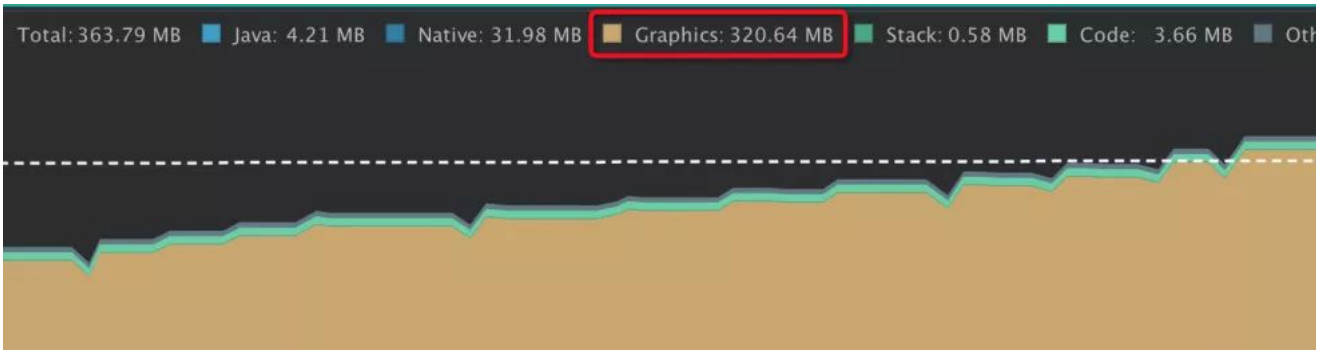
```
class StackImageState extends State<StackImages> {
  var images = <String>[];
  var index = 0;

  @override
  Widget build(BuildContext context) {
    var widgets = <Widget>[];

    for (int i = 0; i <= index; i++) {
      var pos = i - (i ~/ 103) * 103;
      widgets.add(new Container(
          child: new Image.asset("images/${pos}.jpg", fit: BoxFit.cover),
          padding: new EdgeInsets.only(top: i * 2.0)));
    }

    widgets.add(new Center(
        child: new GestureDetector(
            child: new Container(
                child: new Text("添加图片(${index})",
                    style: new TextStyle(color: Colors.red)),
                color: Colors.green,
                padding: const EdgeInsets.all(8.0)),
            onTap: () {
              setState(() {
                index++;
              });
            })));
    return new Stack(
        children: widgets, alignment: AlignmentDirectional.topCenter);
  }
}
```

得到的结果是:

Android 6.0



Android 8.0



可以看到，Flutter Image使用的内存既不属于Java虚拟机内存也不属于Native内存，而是Graphics内存(在Meizu pro5设备上也不属于Graphics,事实上Meizu pro5设备不能归类Flutter Image所使用的内存)，官方对Graphics内存的解释是：

图 2. Memory Profiler 顶部的内存计数图例

内存计数中的类别如下所示：

- **Java**：从 Java 或 Kotlin 代码分配的对象内存。
- **Native**：从 C 或 C++ 代码分配的对象内存。

  即使您的应用中不使用 C++，您也可能会看到此处使用的一些原生内存，因为 Android 框架使用原生内存代表您处理各种任务，如处理图像
  他图形时，即使您编写的代码采用 Java 或 Kotlin 语言。
- **Graphics**：图形缓冲区队列向屏幕显示像素（包括 GL 表面、GL 纹理等等）所使用的内存。（请注意，这是与 CPU 共享的内存，不是 GPU
  存。）

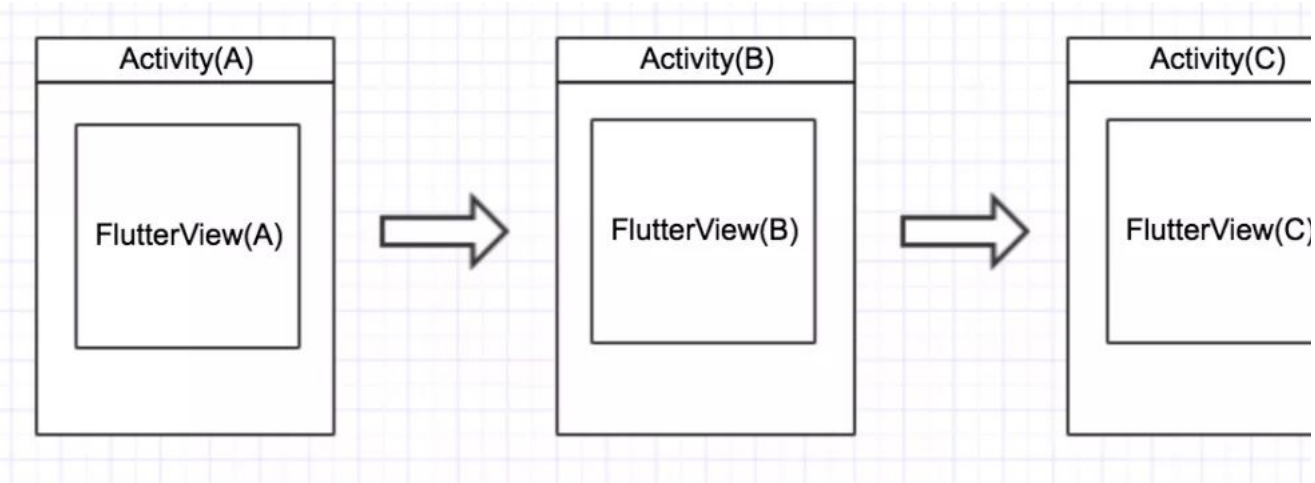
那么至少Flutter Image所使用的内存不会是Java虚拟机内存，这对不少Android设备都是一个好消息，这意味着使用Flutter Image没有OOM的风险，能够较好的利用Native内存。

使用Image的时候，建立一个内存缓存池是个好习惯，Flutter Framework提供了一个ImageCache来缓存加载的图片，但它不同于Android Lru Cache，不能精确的使用内存大小来设定缓存池容量，而是只能粗略的指定最大缓存图片张数。

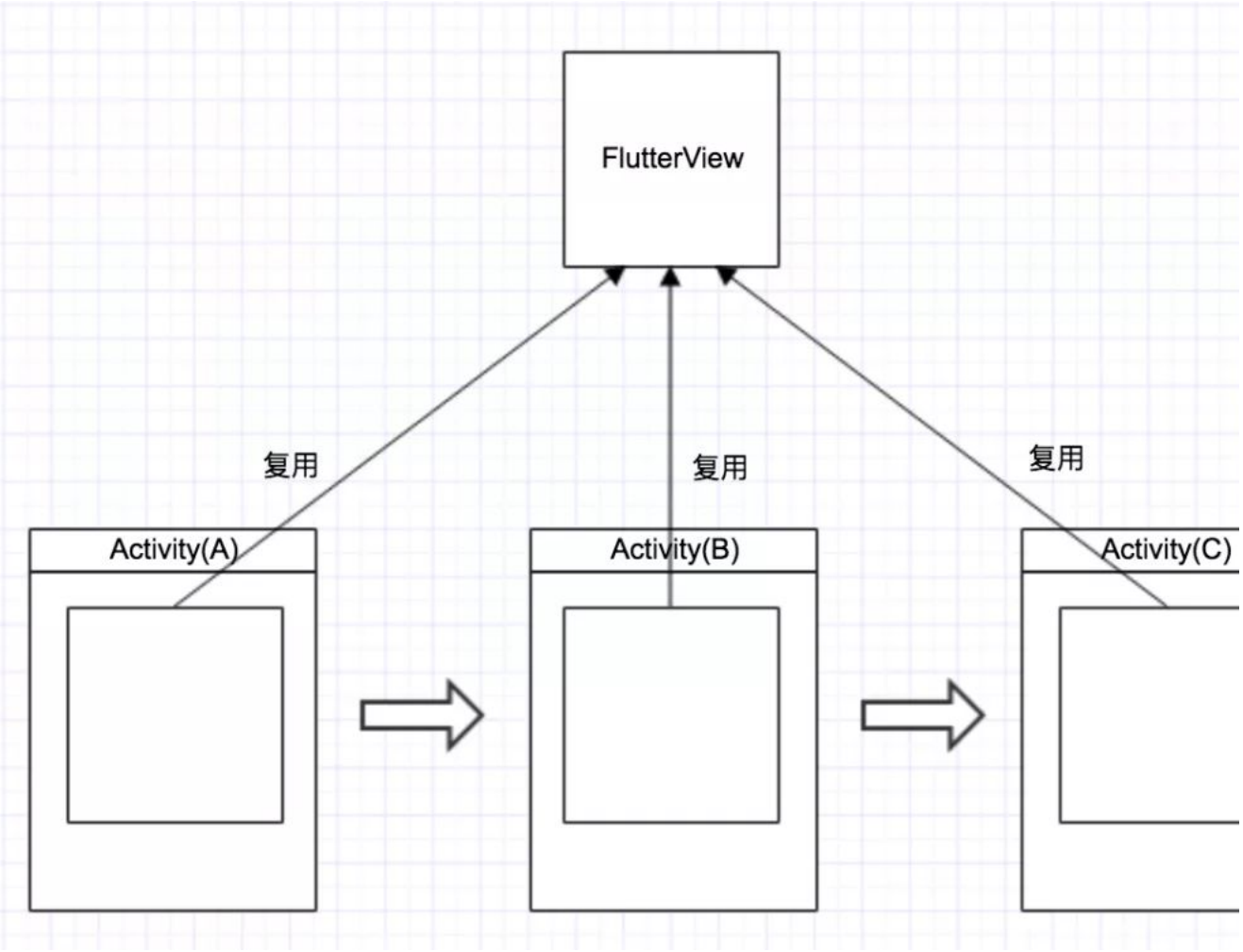
FlutterView内存初探

Flutter设计之初是想统一Android和IOS的界面编程，所以理想的基于Flutter的apk只需要提供一个MainActivity做入口即可，后面所有的页面跳转都在FlutterView中管理。但是，如果是一个已有规模的app接入Flutter开发，我们不可能将已有的Activity页面都用Flutter重新实现一遍，这时候就需要考虑本地页面和Flutter页面之间的跳转交互了。iOS可以方便的管理页面栈，但是Android就很复杂(Android有任务栈机制，低内存Activity回收机制等)，所以通常我们还是使用Activity作为页面容器来展示flutter页面。这时有两种选择，可以每次启动一个Activity就启动一个新的FlutterView，也可以启动Activity的时候复用已有的FlutterView。

不复用FlutterView



复用FlutterView



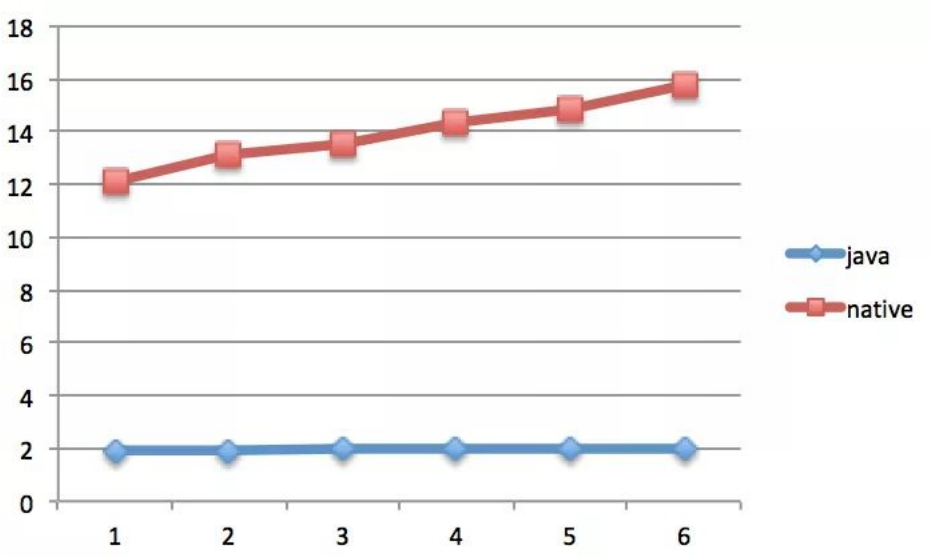
Flutter Framework中FlutterView是绑定Activity使用的，要复用FlutterView就必须能够把FlutterView单独拎出来使用。所幸现在FlutterView和Activity耦合程度并不很深，最关键的地方是FlutterNativeView必须attach一个Activity:

```
//attach到当前Activity
mNativeView.attachViewAndActivity(this, activity);
```

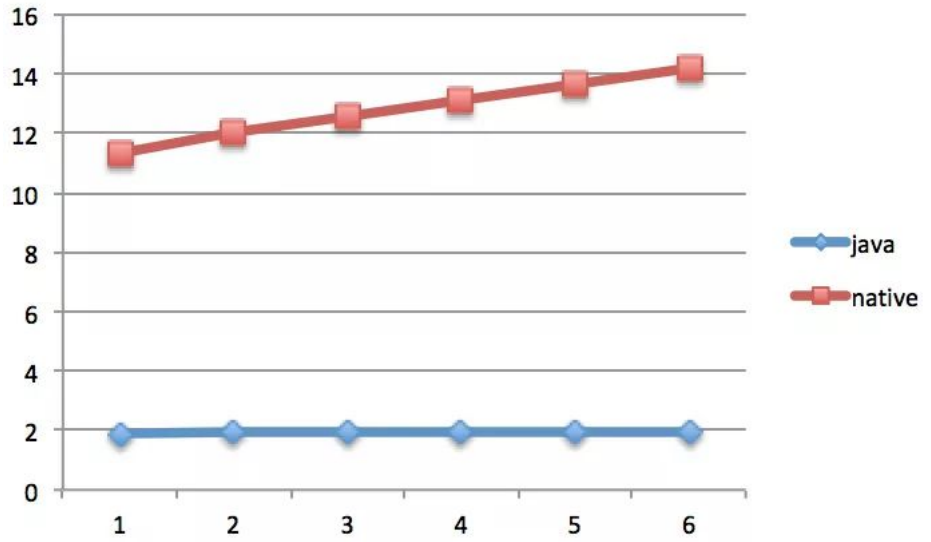
初始化FlutterView时必须传入一个Activity，当其他Activity复用FlutterView时再调用该Attach方法即可。这里有个问题，就是FlutterView中必须保存一个Activity引用，这个一个内存泄露隐患，我们可以在FluterView detach时候将MainActivity传入，因为通常整个App交互过程中MainActivity都是一直存在的，可以避免其他Activity泄露。

为了更好的权衡两种方法的利弊，我们先用空页面来测试一下当页面增加时内存的变化：

不复用FlutterView时，页面增加时内存变化



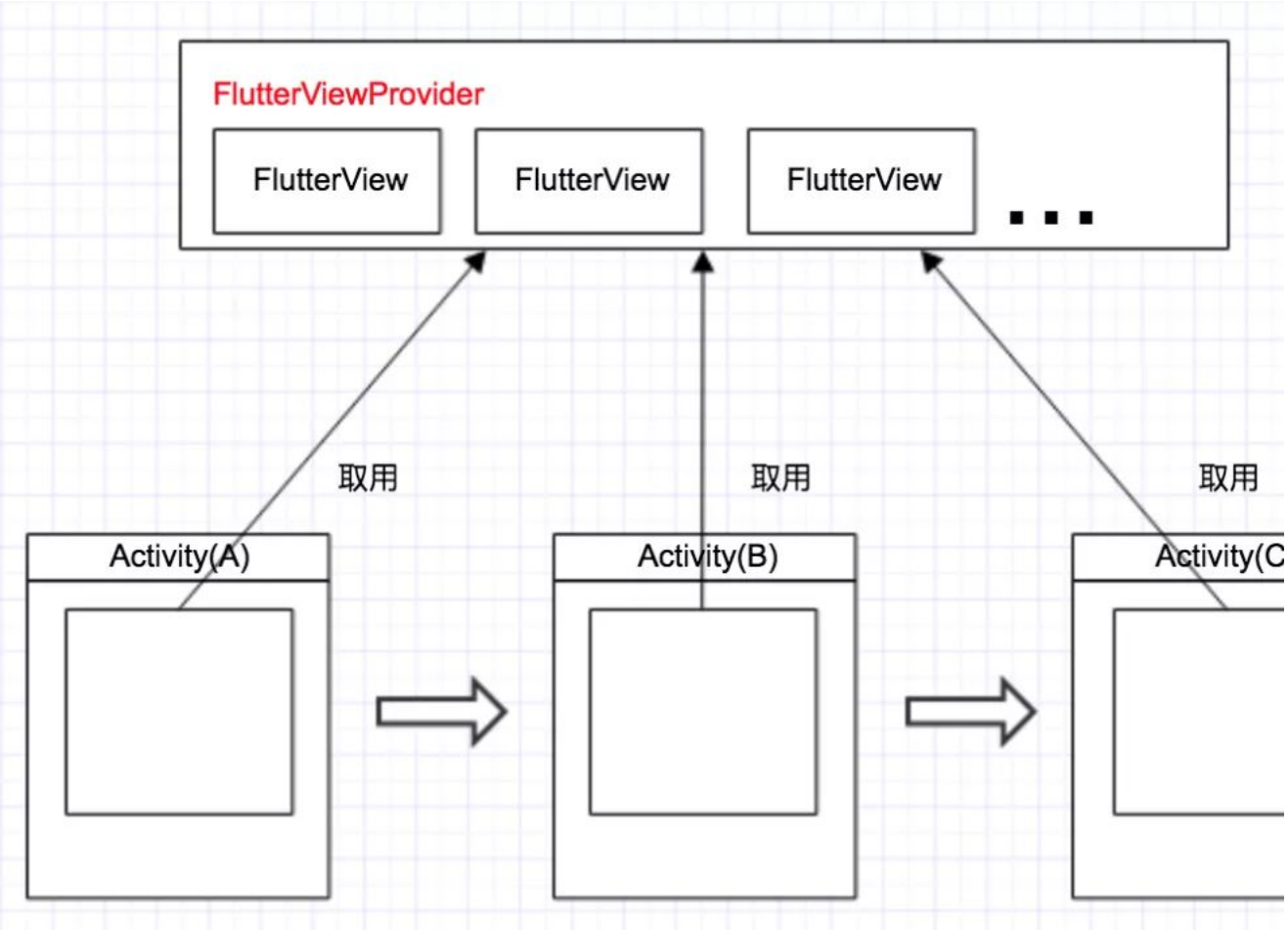
复用FlutterView时，页面增加时内存变化



不复用FlutterView时平均打开一个页面(空页面)，Java内存增长0.02M,Native内存增长0.73M。复用FlutterView时平均打开一个页面(空页面),Java内存增长0.019M,Native内存增长0.65M。可见复用FlutterView在内存使用上是有优势的，但主要复用的还是Native部分的内存。复用FlutterView必然带来额外的一些复杂逻辑，有时候为了逻辑简单，后期维护上的方便，牺牲一些相对不太珍贵的Native内存也是值得的。

复用单个FlutterView有时会有些“意外”，比如当Activity切换时，就不得不将当前FlutterView detach掉给后面新建的Activity使用，当前界面就会空白闪动，有个想法是可以将当前界面截屏下来遮挡住后面的界面变化，这种方式有时会带来额外的适配问题。

FlutterView复用与否不是绝对的，有时候可以使用一些综合性折中方案，比如，我们可以建立一个FlutterViewProvider,里面维护N个可复用的FlutterView，如图：



这样的好处是，可以存在一定程度上的复用，又可以避免只有一个FlutterView出现的一些尴尬问题。

FlutterView的首帧渲染耗时较高，在Debug版本有明显感受，大概会黑屏2秒，release版本会好很多。但我们观察Cpu曲线，发现还是一个较为耗时的过程。有一种体验优化的思路是，我们可以预先让将要使用的FlutterView加载好首帧，这样，在真正使用的时候就很快了，可以先建立一个只有1个像素的窗口，在这个窗口里面完成FlutterView首帧渲染，代码如下：

```
final WindowManager wm = mFakeActivity.getWindowManager();
final FrameLayout root = new FrameLayout(mFakeActivity);

//一个像素足矣
FrameLayout.LayoutParams params = new FrameLayout.LayoutParams(1, 1);
root.addView(flutterView,params);
WindowManager.LayoutParams wlp = new WindowManager.LayoutParams();
wlp.width = 1;
wlp.height = 1;
wlp.flags |= WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE;
wlp.flags |= WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE;
wm.addView(root,wlp);

final FlutterView.FirstFrameListener[] listenerRef = new FlutterView.FirstFrameListener[1];
    listenerRef[0] = new FlutterView.FirstFrameListener() {
        @Override
       public void onFirstFrame() {
        //首帧渲染完后取消窗口
            wm.removeView(root);
          flutterView.removeFirstFrameListener(listenerRef[0]);
       }
    };

flutterView.addFirstFrameListener(listenerRef[0]);
String appBundlePath = FlutterMain.findAppBundlePath(mFakeActivity.getApplicationContext());
flutterView.runFromBundle(appBundlePath, null, "main", true);
```

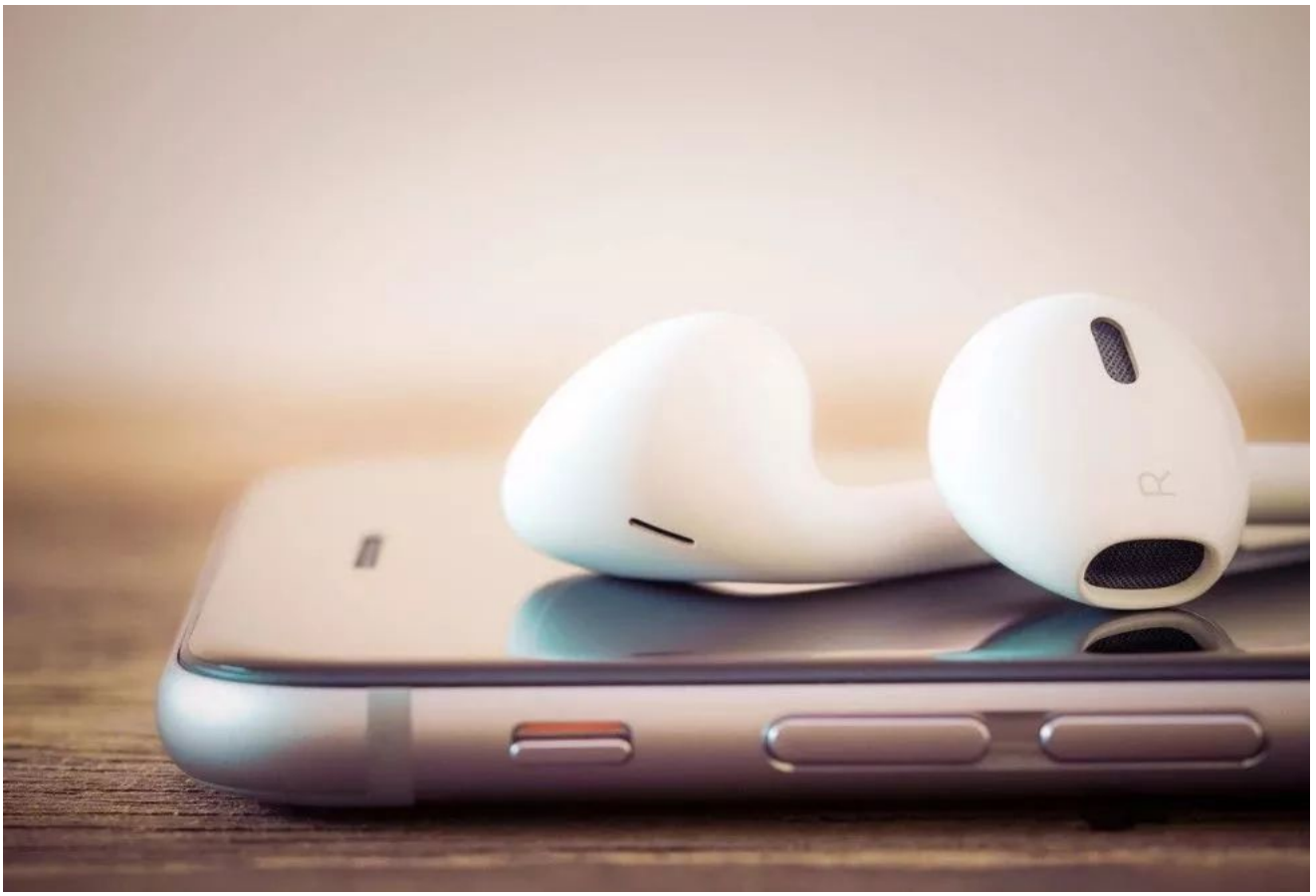
以上就是闲鱼团队在Flutter的应用过程中的一些实践，希望有更多的新技术尝试和技术挑战的同学，请在下面留言告诉我们。

End

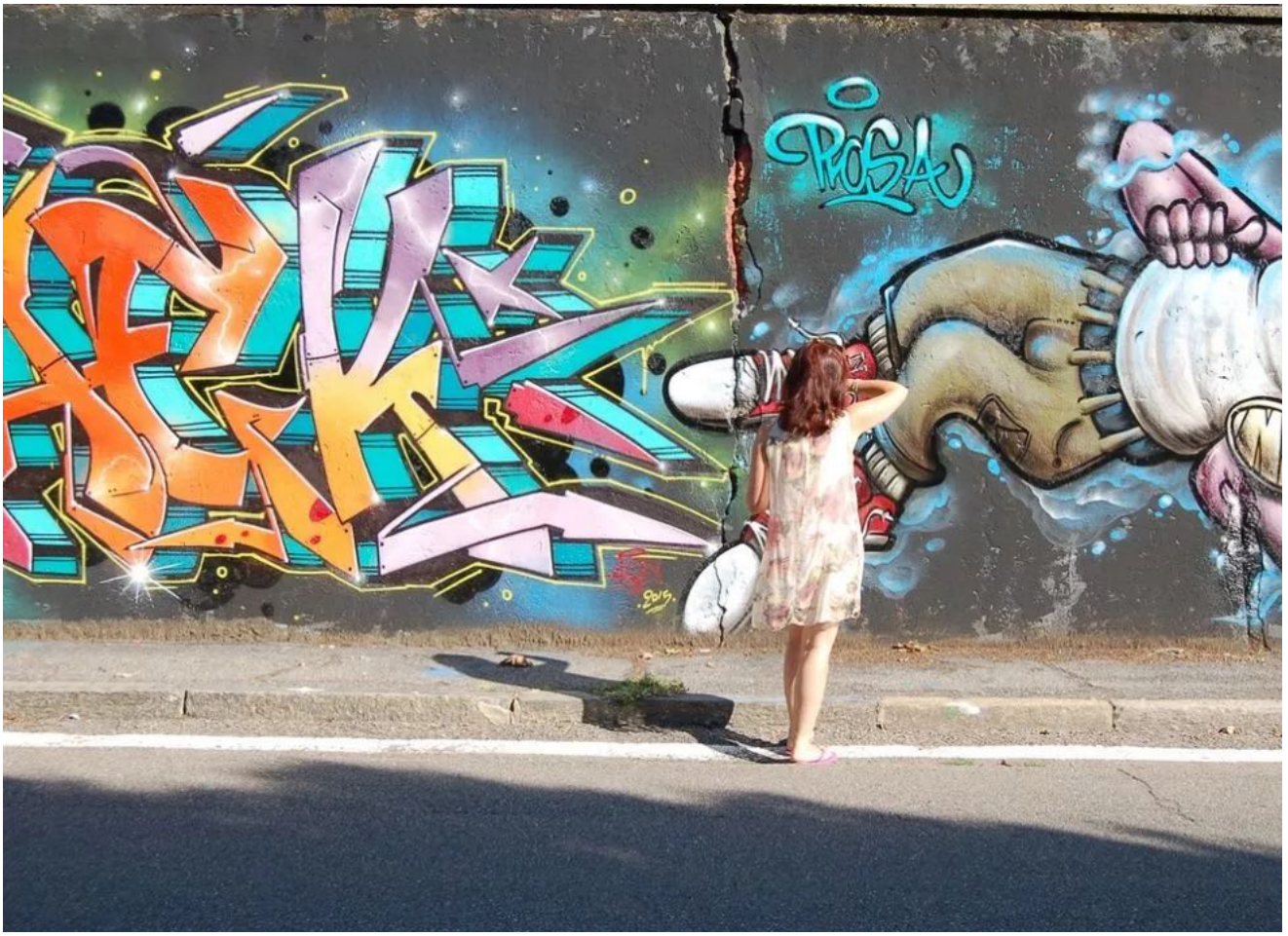
你可能还喜欢

点击下方图片即可阅读

前端 Leader 如何做好团队规划？



基于TensorFlow，人声识别如何在端上实现？



AI设计师“鹿班”核心技术公开：

如何1秒设计8000张海报？

关注「阿里技术」

把握前沿技术脉搏