性能测试

Android篇

1.启动性能

1.1 应用第一次启动：冷启动

应用第一次启动也就是我们常说的**冷启动**,

这时候你的应用程序的进程是没有创建的. 这也是大部分应用的使用场景.用户在桌面上点击你应用的 icon 之后,首先要创建进程,然后才启动 MainActivity.这时候adb shell am start -w packagename/MainActivity 返回的结果,就是标准的应用程序的启动时间

（注意:Android 5.0 之前的手机是没有 WaitTime 这个值的)

```
adb shell am start -W com.meizu.media.painter/com.meizu.media.painter.PainterMainActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.mei
zu.media.painter/.PainterMainActivity }
Status: ok
Activity: com.meizu.media.painter/.PainterMainActivity
ThisTime: 355
TotalTime: 355
WaitTime: 365
Complete
```

- ThisTime：表示一连串启动 Activity 的最后一个 Activity 的启动耗时
- TotalTime：表示新应用启动的耗时，包括新进程的启动和 Activity 的启动，但不包括前一个应用 Activity pause 的耗时
- WaitTime：就是总的耗时，包括前一个应用 Activity pause 的时间和新应用启动的时间

最后总结一下，如果只关心某个应用自身启动耗时，参考TotalTime；如果关心系统启动应用耗时，参考WaitTime；如果关心应用有界面Activity启动耗时，参考ThisTime

1.2 应用非第一次启动

如果是你按Back键，并没有将应用进程杀掉的话，那么执行上述命令就会快一些，因为不用创建进程了，只需要启动一个Activity即可。这也就是我们说的应用热启动。

参考：Android 中如何计算 App 的启动时间?

页面加载时间

也可以通过 `am start` 、 `logcat | grep ActivityManager` 或者 `代码插桩` 的方式获得，不过需要注意的是：

业务功能对应的网络消耗时间也是非常重要的一部分

我们不仅仅要关注**native层响应时间的长短**，更要关心**每个业务到底要调用多少个接口，接口的响应时间，H5页面还要了解其中css，js，png等是否根据网络做了不同策略的调整，是否被压缩了**。时间数据仅仅是最终的一个展现。

2.内存性能

2.1 内存占用

- Android的每个应用程序都会使用一个专有的Dalvik虚拟机实例来运行，也就是说每个应用程序都是在属于自己的进程中运行的。
- 内存主要分为：Native、Dalvik、Other

PS:不同的系统API这个命令的返回值可能会有一些不一致

```
shell@msm8916_32:/ $ dumpsys meminfo com.yaya.mmbang
Applications Memory Usage (kB):
Uptime: 26885572 Realtime: 111884890

** MEMINFO in pid 15168 [com.yaya.mmbang] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap        0        0        0        0    10956     9196     1427
  Dalvik Heap    22236    22192        0        0    28792    21158     7634
 Dalvik Other     4342     4268        0        0
        Stack      280      280        0        0
       Ashmem     6652     6652        0        0
    Other dev     8057     6964        4        0
     .so mmap     2475     2184        0        0
    .apk mmap        1        0        0        0
    .dex mmap      511       36      316        0
   Other mmap        5        4        0        0
     Graphics     2400     2400        0        0
           GL    12288    12288        0        0
      Unknown     5662     5656        0        0
        TOTAL    64909    62924      320        0    39748    30354     9061
```

- Android系统中和dalvik有关的配置信息:

```
shell@msm8916_32:/ $ cat /system/build.prop | grep dalvik
dalvik.vm.heapgrowthlimit=128m #受控情况下的heap堆最大值
dalvik.vm.heapsize=256m #不受控情况下的heap堆最大值
dalvik.vm.heapstartsize=8m #初始分配的heap堆大小
dalvik.vm.heaptargetutilization=0.75
dalvik.vm.heapminfree=2m
dalvik.vm.heapmaxfree=8m
```

- VSS、RSS、PSS、USS

```
VSS- Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
RSS- Resident Set Size 实际使用物理内存（包含共享库占用的内存）
PSS- Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
USS- Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）
```

- 当然还有其他的方式，比如 `procrank` (能获取到USS更精准，不过需要root)

2.2 内存溢出

所谓内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是会产生内存溢出的问题。

- **什么是heap?**

> Heap空间由程序控制，程序员可以使用malloc、new、free、delete等函数调用来操作这片地址空间。Heap为程序完成各种复杂任务提供内存空间，所以空间比较大，一般为几百MB到几GB。正是因为Heap空间由程序员管理，所以容易出现使用不当导致严重问题。C/C++申请的内存空间在native heap中，而java申请的内存空间则在dalvik heap中

- **进程内存空间和RAM之间的关系**

> 进程的内存空间只是虚拟内存（或者叫作逻辑内存），而程序的运行需要的是实实在在的内存，即物理内存（RAM）。在必要时，操作系统会将程序运行中申请的内存（虚拟内存）映射到RAM，让进程能够使用物理内存。

- **Android的 java程序为什么容易出现OOM**

> 这个是因为Android系统对dalvik的vm heapsize作了硬性限制，当java进程申请的java空间超过阈值时，就会抛出OOM异常（这个阈值可以是48M、24M、16M等，视机型而定），可以通过adb shell getprop | grep dalvik.vm.heapsize查看此值。 **也就是说，程序发生OMM并不表示RAM不足，而是因为程序申请的java heap对象超过了dalvik vm heapsize**。也就是说，在RAM充足的情况下，也可能发生OOM。

- **Android如何应对RAM不足**

> java程序发生OMM并不是表示RAM不足，如果RAM真的不足，会发生什么呢？这时Android的memory killer会起作用，当RAM所剩不多时，memory killer会杀死一些优先级比较低的进程来释放物理内存，让高优先级程序得到更多的内存。我们在分析log时，看到的进程被杀的log
> `06-12 11:29:24.493 907-1229/? I/ActivityManager: Process com.yaya.mmbang (pid 5569) has died.`

- **如何查看RAM使用情况**

> 可以使用adb shell cat /proc/meminfo查看RAM使用情况

- **Bitmap分配在native heap还是dalvik heap上?**

> 大家都知道，过多地创建bitmap会导致OOM异常，且native heapsize不受dalvik限制，所以可以得出结论：
>
> Bitmap只能是分配在dalvik heap上的，因为只有这样才能解释bitmap容易导致OOM。

- **java程序如何才能创建native对象**

> 必须使用jni，而且应该用C语言的malloc或者C++的new关键字

- **3种测量内存的工具**：

> - Memory Monitor：跟踪整个app的内存变化情况。
> - Heap Viewer：查看当前内存快照，便于对比分析哪些对象有可能发生了泄漏。
> - Allocation Tracker：追踪内存对象的来源

参考：Android内存管理详细介绍

2.3 垃圾回收&内存泄漏

Java内存泄漏指的是进程中某些对象（垃圾对象）已经没有使用价值了，但是它们却可以直接或间接地引用到gc roots导致无法被GC回收。无用的对象占据着内存空间，使得实际可使用内存变小，形象地说法就是内存泄漏了。

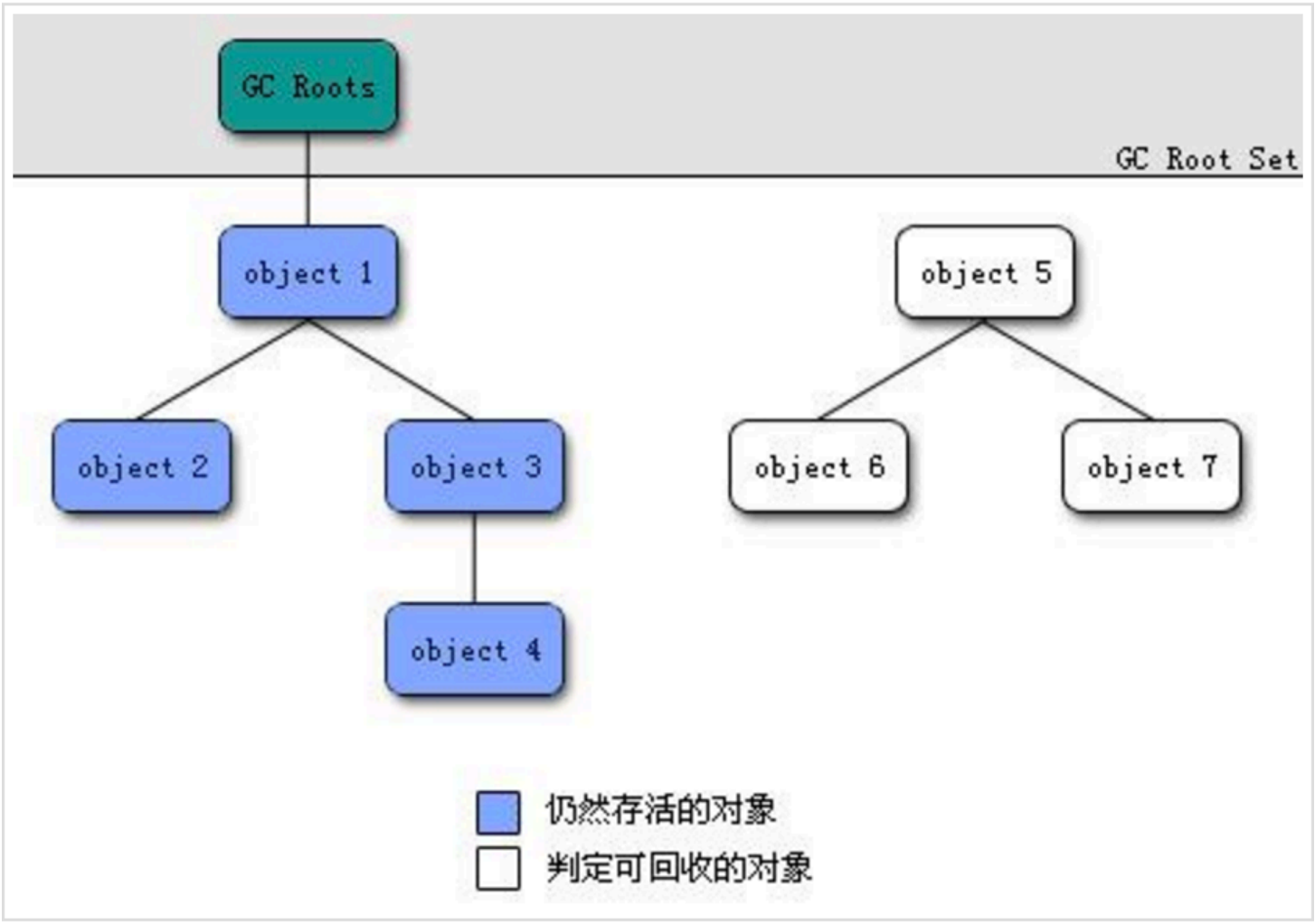
垃圾回收机制

在java中垃圾回收是系统自动完成的。

从GC Roots（每种具体实现对GC Roots有不同的定义）作为起点，向下搜索它们引用的对象，可以生成一棵引用树，树的节点视为可达对象，反之视为不可达。

在Java语言中，可以作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中的引用对象。
- 方法区中的类静态属性引用的对象。
- 方法区中的常量引用的对象。
- 本地方法栈中JNI（Native方法）的引用对象



引用类型分类

引用类型有四种类型分别是：强引用（Strong Reference）、软引用(soft Reference)、弱引用（Weak Reference）、虚引用（Phantom Reference），其中强引用是我们常用到的。下面来说明下他们什么时候会被垃圾回收机制回收。

- **强引用**： 这种最顽强，只要有一个引用存在，永远都不会被回收掉。
- **软引用**： 一般是指还有用，但是非必须的对象。在内存空间不足的情况下，会回收掉此部分内存，如果还不够则会抛出内存溢出异常。
- **弱引用**： 一般指非必须的对象，比软引用还要弱，它只能生存到下一次垃圾回收前，如果一旦发生垃圾回收，它将会被回收掉。
- **虚引用**： 最弱的引用关系，无法通过虚引用来取得一个对象的实例。为一个对象设置虚引用的唯一目的就是能够在这个对象被回收的时候收到一个系统通知。

Stop the world 概念

> 因为垃圾回收的时候，需要整个的引用状态保持不变，否则判定是判定垃圾，等我稍后回收的时候它又被引用了，这就全乱套了。所以，GC的时候，其他所有的程序执行处于暂停状态，卡住了。 幸运的是，这个卡顿是非常短（尤其是新生代），对程序的影响微乎其微 （关于其他GC比如并发GC之类的，在此不讨论）。 所以GC的卡顿问题由此而来，也是情有可原，暂时无可避免。

GC 的类型

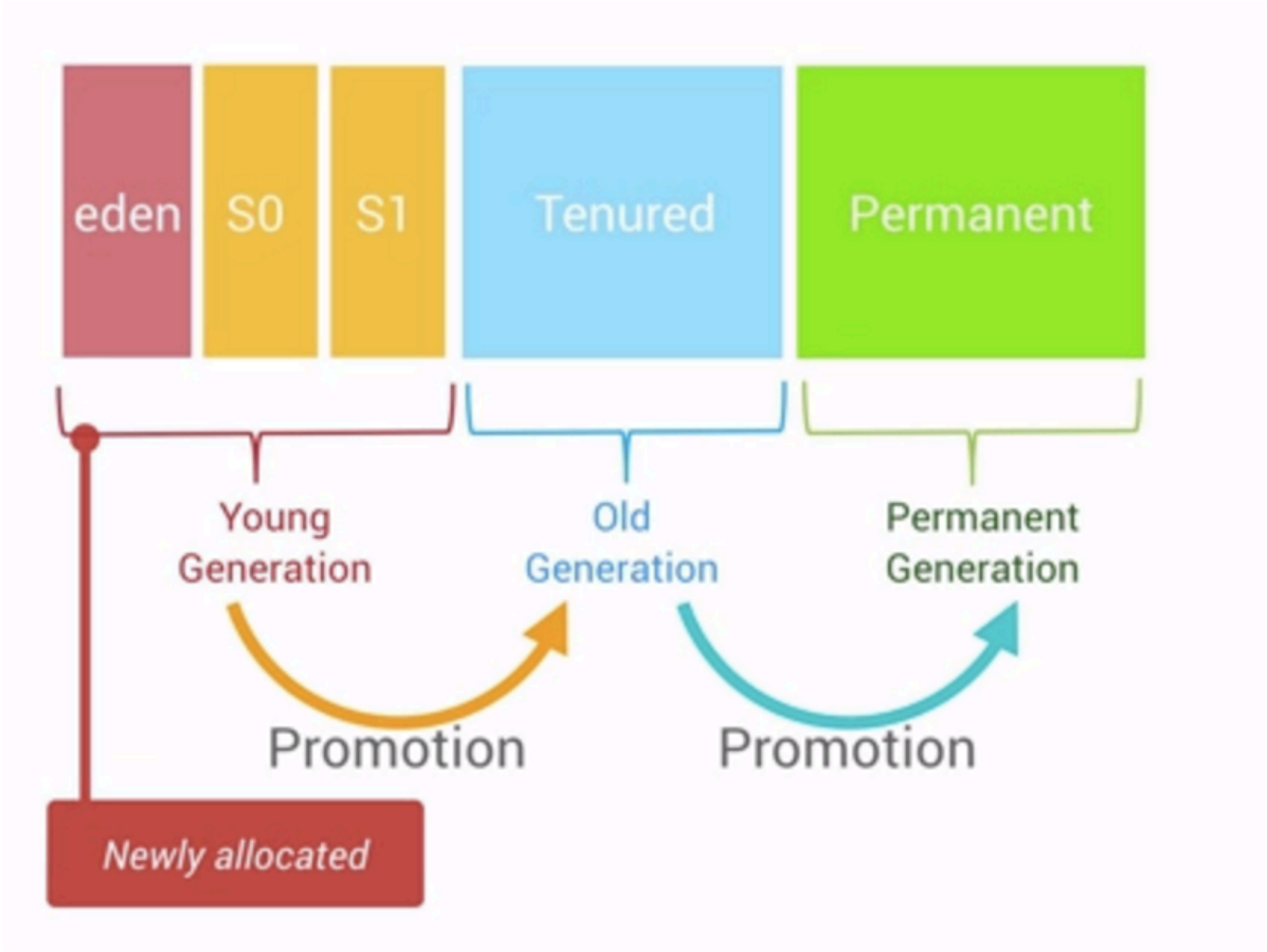
> Davlik虚拟机定义了四种类的GC,它们的含义如下所示：

```
GC_FOR_MALLOC：表示是在堆上分配对象时内存不足触发的GC。

GC_CONCURRENT：表示是在已分配内存达到一定量之后触发的GC。

GC_EXPLICIT：表示是应用程序调用System.gc、VMRuntime.gc接口或者收到SIGUSR1信号时触发的GC。

GC_BEFORE_OOM：表示是在准备抛OOM异常之前进行的最后努力而触发的GC。
```

> 实际上， `GC_FOR_MALLOC` 、 `GC_CONCURRENT` 和 `GC_BEFORE_OOM` 三种类型的GC都是在分配对象的过程触发的。我们在Android的logcat中可以看到GC触发的log。

三级Generation的内存模型

> **Android系统里面有一个Generational Heap Memory的模型，系统会根据内存中不同的内存数据类型分别执行不同的GC操作**。例如，最近刚分配的对象会放在Young Generation区域，这个区域的对象通常都是会快速被创建并且很快被销毁回收的，同时这个区域的GC操作速度也是比Old Generation区域的GC操作速度更快的。
>
> 前面提到过每次GC发生的时候，所有的线程都是暂停状态的。**GC所占用的时间和它是哪一个Generation也有关系，Young Generation的每次GC操作时间是最短的，Old Generation其次，Permanent Generation最长。**执行时间的长短也和当前Generation中的对象数量有关，遍历查找20000个对象比起遍历50个对象自然是要慢很多的。



虽然Google的工程师在尽量缩短每次GC所花费的时间，但是特别注意GC引起的性能问题还是很有必要。如果不小心在最小的for循环单元里面执行了创建对象的操作，这将很容易引起GC并导致性能问题。通过Memory Monitor我们可以查看到内存的占用情况，每一次瞬间的内存降低都是因为此时发生了GC操作，如果在短时间内发生大量的内存上涨与降低的事件，这说明很有可能这里有性能问题。我们还可以通过Heap and Allocation Tracker工具来查看此时内存中分配的到底有哪些对象。

参考：JAVA垃圾回收机制

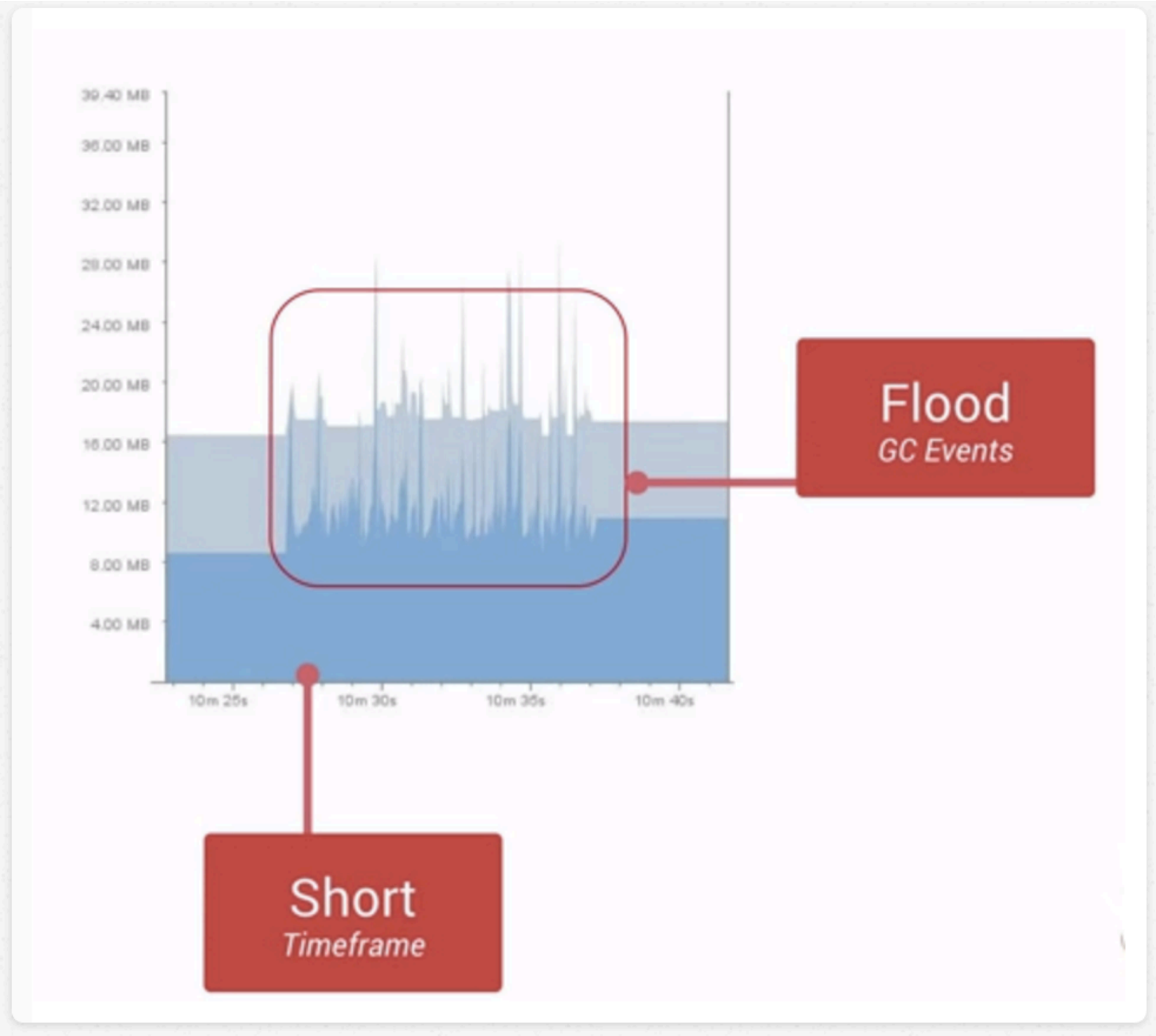
3.流畅度

3.1 频繁GC引起的流畅度问题

通常来说，单个的GC并不会占用太多时间，但是大量不停的GC操作则会显著占用帧间隔时间(16ms)。如果在帧间隔时间里面做了过多的GC操作，那么自然其他类似计算，渲染等操作的可用时间就变得少了 导致频繁GC的两个原因：

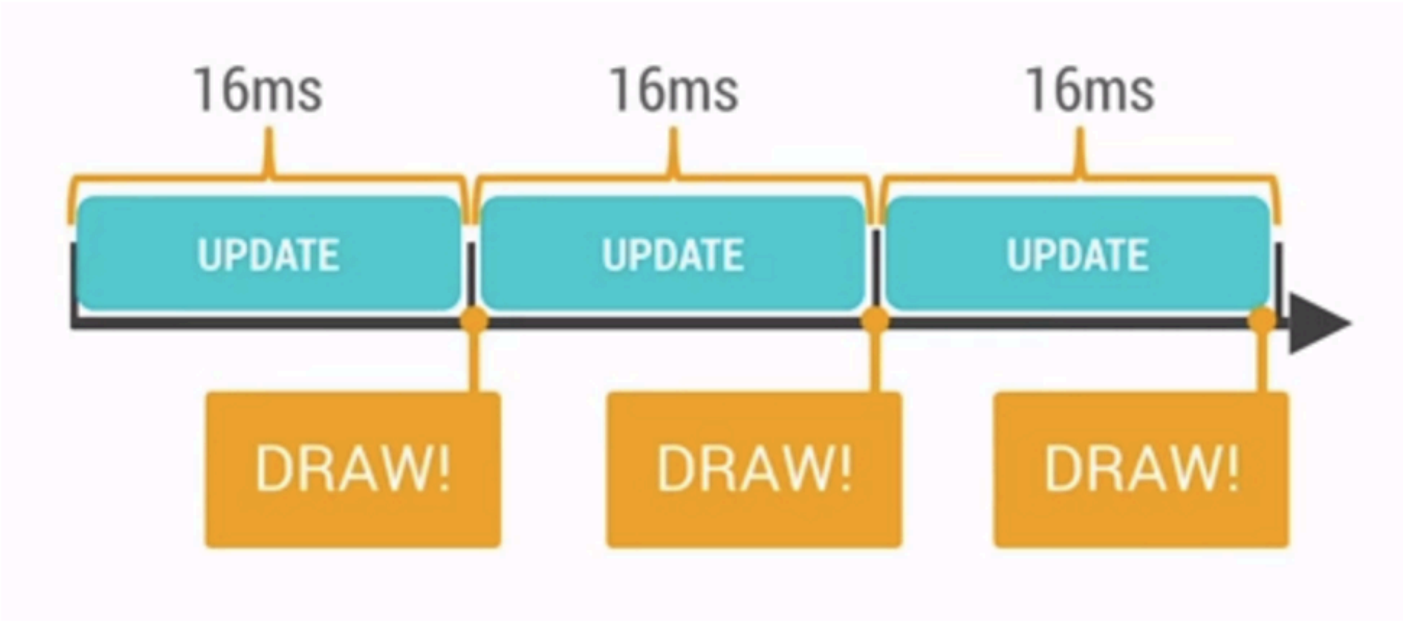
- Memory Churn内存抖动，内存抖动是因为大量的对象被创建又在短时间内马上被释放。
- 瞬间产生大量的对象会严重占用Young Generation的内存区域，当达到阀值，剩余空间不够的时候，也会触发GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加Heap的压力，从而触发更多其他类型的GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。

解决上面的问题有简洁直观方法，如果你在Memory Monitor里面查看到短时间发生了多次内存的涨跌，这意味着很有可能发生了内存抖动。

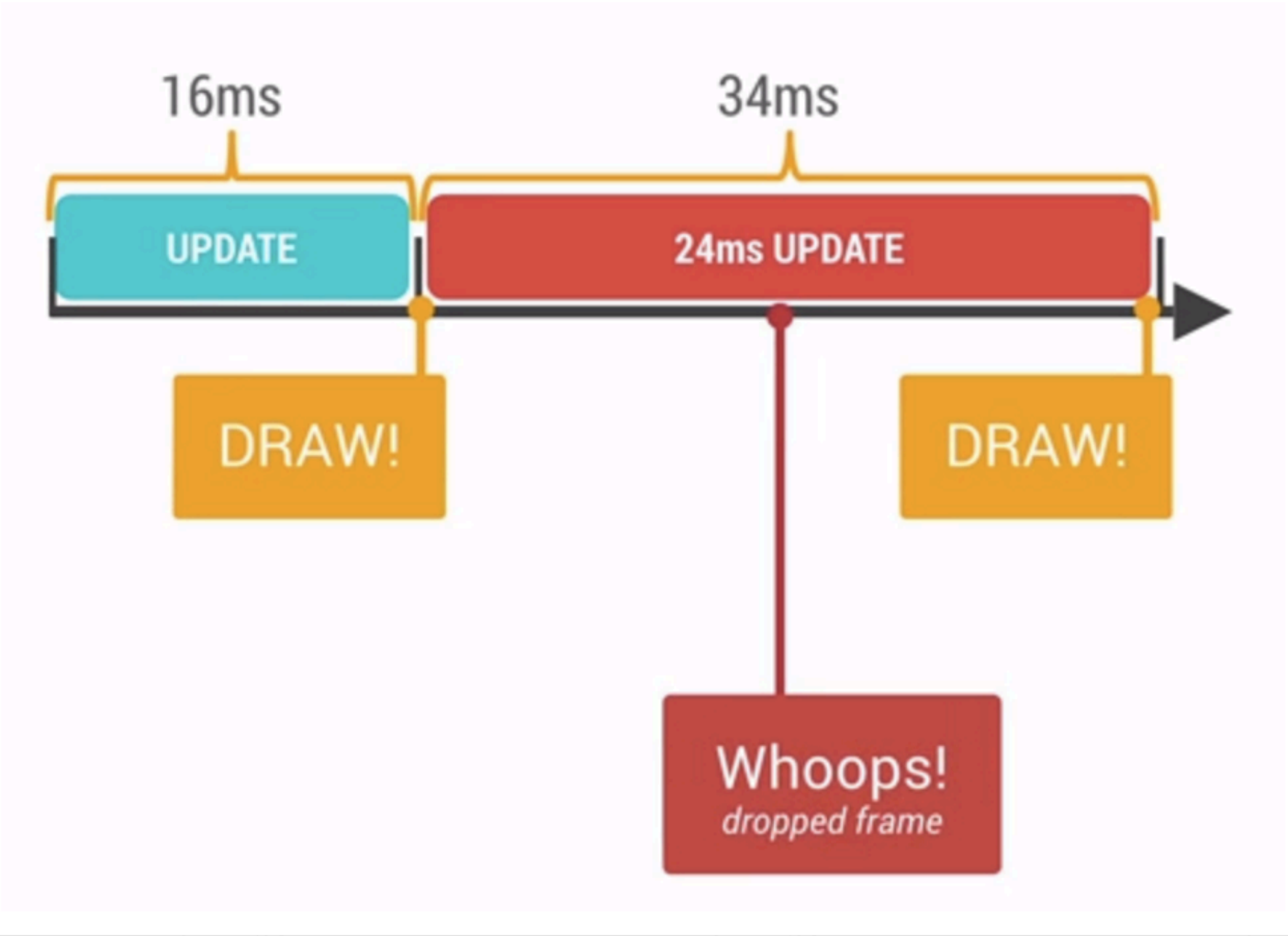


3.2 渲染性能

大多数用户感知到的卡顿等性能问题的最主要根源都是因为渲染性能。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的60fps，为了能够实现60fps，这意味着程序的大多数操作都必须在16ms内完成。

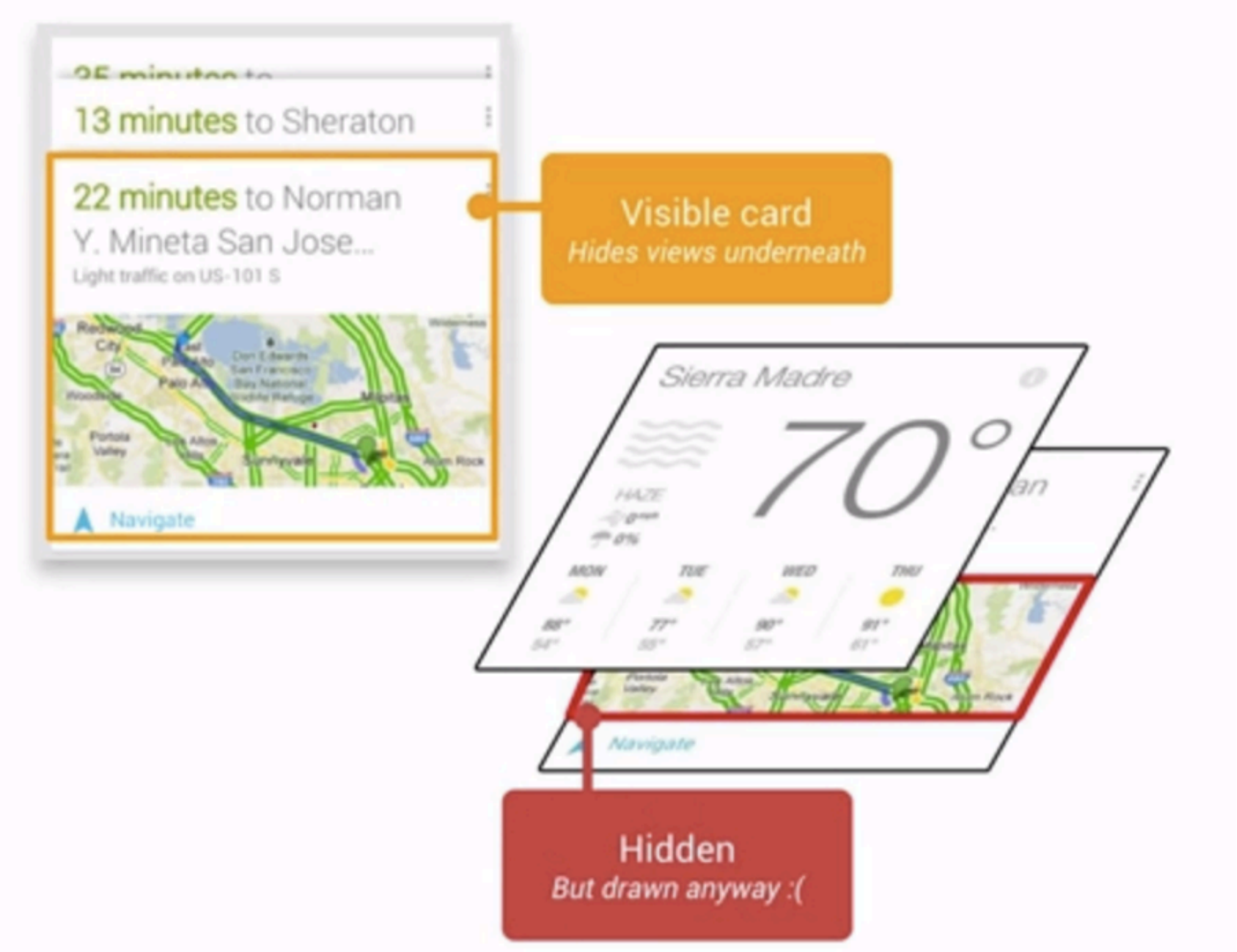


如果你的某个操作花费时间是24ms，系统在得到VSYNC信号的时候就无法进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面。



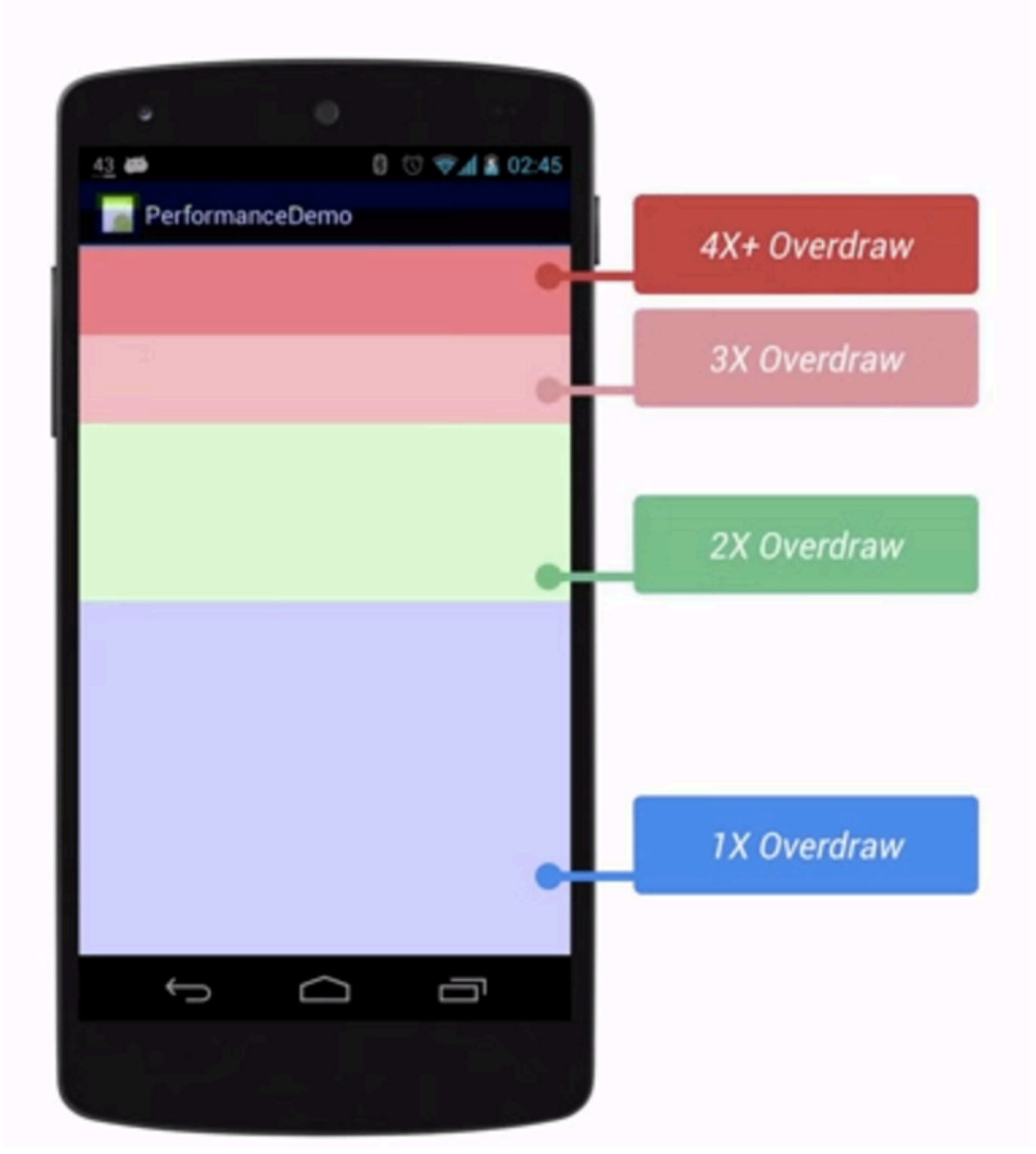
3.2.1 过度绘制（Show GPU Overdraw）

Overdraw(过度绘制)描述的是屏幕上的某个像素在同一帧的时间内被绘制了多次。



在多层次的UI结构里面，如果不可见的UI也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的CPU以及GPU资源。

工具：Show GPU Overdraw



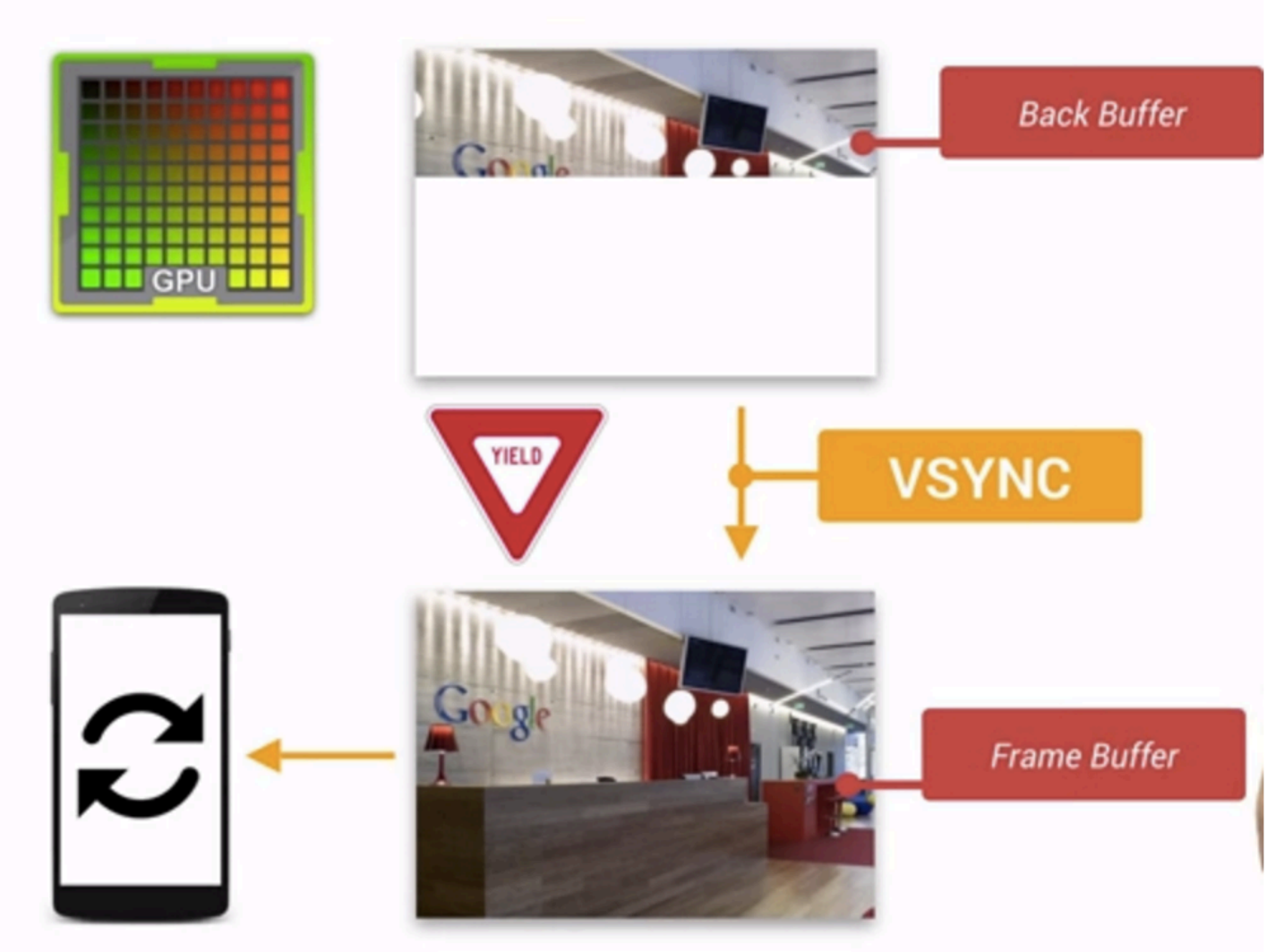
我们可以通过手机设置里面的开发者选项，打开Show GPU Overdraw的选项，可以观察UI上的Overdraw情况。

3.2.2 VSYNC（Profile GPU Rendering）

在讲解VSYNC之前，我们需要了解两个相关的概念：

- Refresh Rate：代表了屏幕在一秒内刷新屏幕的次数，这取决于硬件的固定参数，例如60Hz。
- Frame Rate：代表了GPU在一秒内绘制操作的帧数，例如30fps，60fps。

Tearing（画面撕裂）



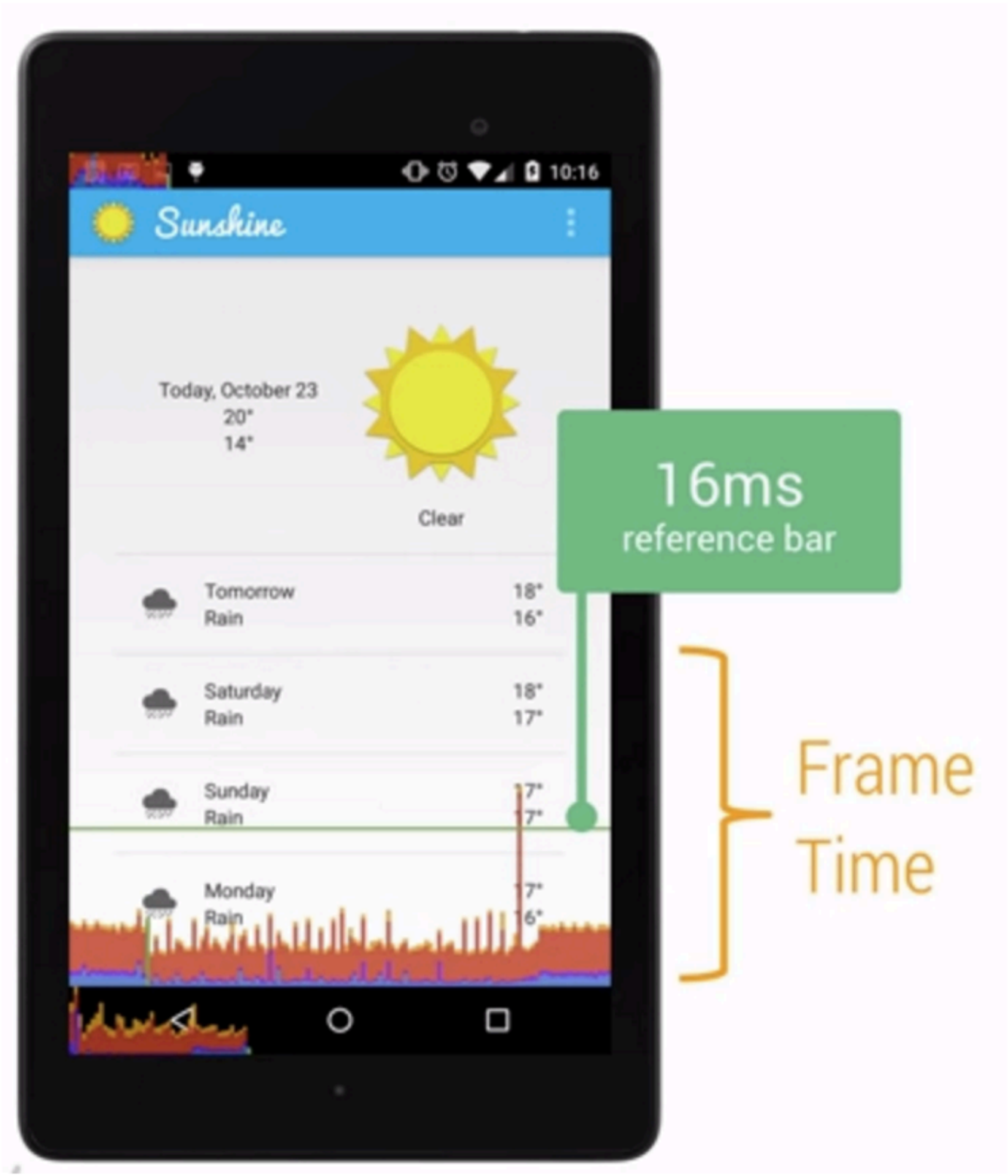
不幸的是，刷新频率和帧率并不是总能够保持相同的节奏。如果发生帧率与刷新频率不一致的情况，就会容易出现Tearing的现象(画面上下两部分显示内容发生断裂，来自不同的两帧数据发生重叠)。

VSYNC

通常来说，帧率超过刷新频率只是一种理想的状况，在超过60fps的情况下，GPU所产生的帧数据会因为等待VSYNC的刷新信息而被Hold住，这样能够保持每次刷新都有实际的新的数据可以显示。但是我们遇到更多的情况是帧率小于刷新频率。

理解图像渲染里面的双重与三重缓存机制，这个概念比较复杂，请移步查看这里，还有这里。

工具：Profile GPU Rendering



中间有一根绿色的横线，代表16ms，我们需要确保每一帧花费的总时间都低于这条横线，这样才能够避免出现卡顿的问题。

每一条柱状线都包含三部分，蓝色代表测量绘制Display List的时间，红色代表OpenGL渲染Display List所需要的时间，黄色代表CPU等待GPU处理的时间。

> GPU会获取图形数据进行渲染，然后硬件负责把渲染后的内容呈现到屏幕上，他们两者不停的进行协作。

3.2.3 Why 60fps?

我们通常都会提到60fps与16ms，可是知道为何会是以程序是否达到60fps来作为App性能的衡量标准吗？这是因为人眼与大脑之间

的协作无法感知超过60fps的画面更新。

12fps大概类似手动快速翻动书籍的帧率，这明显是可以感知到不够顺滑的。24fps使得人眼感知的是连续线性的运动，这其实是归功于运动模糊的效果。24fps是电影胶圈通常使用的帧率，因为这个帧率已经足够支撑大部分电影画面需要表达的内容，同时能够最大的减少费用支出。但是低于30fps是无法顺畅表现绚丽的画面内容的，此时就需要用到60fps来达到想要的效果，当然超过60fps是没有必要的。

开发app的性能目标就是保持60fps，这意味着每一帧你只有16ms=1000/60的时间来处理所有的任务。

参考：Android性能优化典范

4.CPU

CPU利用率是指：

> CPU执行非系统空闲进程的时间 / CPU总的执行时间。

CPU使用率过高造成的影响：

> 会使整个手机无法响应用户，***整体性能降低，手机发烫，引起应用ANR***等等一系列问题。

如果出现ANR需要抓取哪些log

> 抓取logcat的log，搜索ANR等字样；在/data/data下会生成一个traces.txt文件，具体路径可以查看logcat的log，然后使用 `adb pull` 抓取出来；在traces.txt中查看main主线程

两种CPU使用率的计算方式

- top

> top是一段时间内，计算process的cpu jiffies与总的cpu jiffies差值得到。 `seconds = jiffies / Hz`

- cpuinfo

> cpuinfo是一段时间内，计算SystemClock.uptimeMillis()差值

为什么有时候top统计出来的CPU使用率为0%？ 为什么有时候cpuinfo统计出来的CPU使用率超过100%?

Android系统的原因，有时候会出现结果为负数的时候，就会显示为0，而在多核手机上cpu使用率是会出现大于100%的，如果要深入分析的话可以尝试统计jiffies（CPU时间片）

5.流量

5.1 直接从OS中获取

> 思路：获取应用UID，开始测试前，记录一下当前的RX/TX，结束测试时再记录一下RX/TX，然后相减

```
myUID=`adb shell dumpsys package $packageName | grep packageSetting | cut -d "/" -f2 | cut -d "}" -f
1`
RX_bytes_now=`adb shell cat /proc/net/xt_qtaguid/stats | grep $myUID | awk '{rx_bytes+=$6}END{print
rx_bytes}'`
TX_bytes_now=`adb shell cat /proc/net/xt_qtaguid/stats | grep $myUID | awk '{tx_bytes+=$8}END{print
tx_bytes}'`
```

5.2 代理方式

UI界面：Fiddler、Charles

无UI界面：AnyProxy

5.3 tcpdump+ wireshark

没有尝试过，可以自行百度，据说这个拿到的数据比较全面

6.电量

6.1 功耗仪（安捷伦）

精准度最高，但费用消耗庞大，并且使用不方便。无法做自动化

6.2 结合cpu等各种数据最终计算出电量消耗，单位是mA

精准度不如功耗仪，这个公式我这里就不能给出了

6.3 直接从OS中获取。

精准度最低，单位是%，可以同时看到充电状态，温度等一些信息

```
adb shell dumpsys battery | grep level | awk -F: '{print $2}' | tr -d " "
```

7.monkey

使用不同的策略：其实就是根据自己的策略（各种操作比重不同）来制定脚本，包括也可以简单的二次开发，现在流行的做法就是去读取当前所有的Views，然后去做遍历，保证monkey可以在每个Activity上面都执行的到。

使用不同渠道商的脚本：现在各个渠道商都是有自己的monkey脚本来做测试的，如果不通过那么一样耶会被退回来，那么与其这样，不如提前去做。

修复所有的bug：那么这个就是标准了，0 crash和 0 ANR。这两个都是不允许的。