那些年我们用过的显示性能指标

• 发布于 5 个月前 • 作者 Bugly_Tony (/user/115290) • 3201 次浏览 • 来自 技术

> 注：Google 在自己文章中用了 Display Performance 来描述我们常说的流畅度，为了显得有文化，本文主要用“显示性能”一词来代指“流畅度”（虽然两者在概念上有细微差别）。

从 Android 诞生的那一刻起，流畅度就为众人所关注。一时之间，似乎所有人都在讨论 Android 和 iOS 谁的流畅度更好。但是，毫不夸张的说，流畅度绝对是 Android 众多性能维度中最为奇葩的一个。因为，为了刻画这一性能维度，业界设计了各式各样的指标来对其进行衡量。可以说弄清了这些指标我们就明白了什么是流畅度，可是这似乎并不太容易。

笔者简单搜集了一些业界中提及的显示性能指标，大家可以来品评一下：

> **指标名称**：FPS
> **相关资料**：Android性能测试之fps获取 (http://blog.csdn.net/itfootball/article/details/43084527)

> **指标名称**：Aggregate frame stats（N 多个指标）
> **相关资料**：Testing Display Performance (http://developer.android.com/intl/zh-cn/training/testing/performance.html)

> **指标名称**：Jankiness count、Max accumulated frames、Frame rate
> **相关资料**：JankTestBase.java
> (https://android.googlesource.com/platform/frameworks/testing/+/master/uiautomator_test_libraries/src/com/android/uiautomator/platform/JankTestBase.java)

> **指标名称**：SM、Skipped frames
> **相关资料**：Android应用性能评测调优 (http://www.csdn.net/article/2015-06-12/2824949)

面对如此之多的显示性能指标，想必大家也会跟笔者一样，心中难免疑惑丛生。其实，我们只需要依次弄清楚以下三个哲学问题，所有的问题也许就会迎刃而解：

- 你是谁——这些指标具体反映了什么问题
- 你从哪儿来——这些指标数值是怎么得到的
- 你要到哪儿去——这些指标如何落地来指导优化

因此，本文将尝试依次从上诉三个问题来逐步分析和探讨各个显示性能指标。

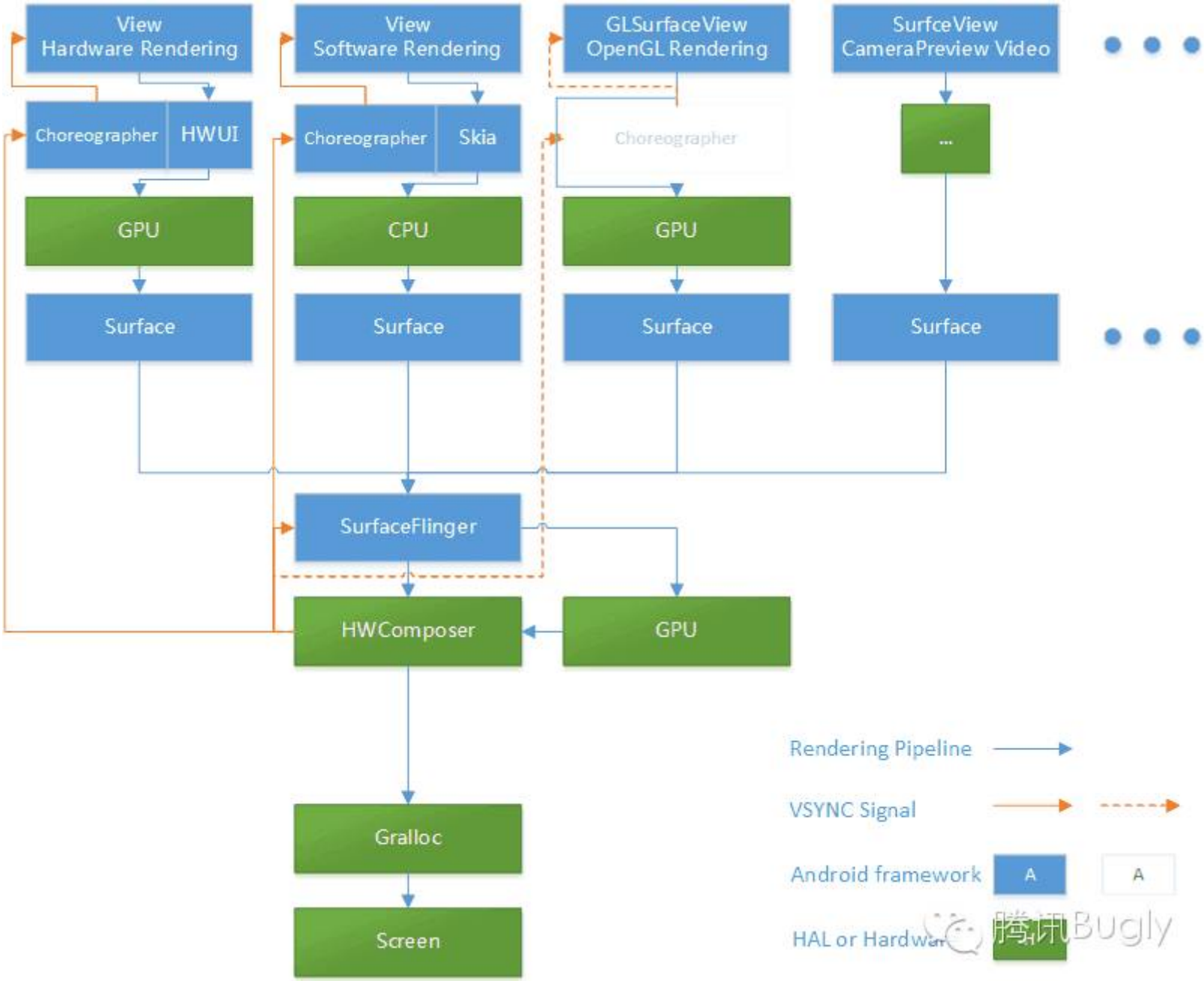
##Step 1：你是谁——这些指标具体反映了什么问题##
总所周知，脱离了具体的应用背景，所有的指标都是没有意义的。所以，为了彻底弄清楚各个显示性能指标的具体身份，我们势必得从 Android 的图像渲染流程说起。

具体展开之前，首先需要说明的是，为了降低复杂程度和本章篇幅，在这个环节之中，我们只讨论图像渲染流程中的各个具体环节所对应的指标有哪些。而指标的具体定义，由第二章《你从哪儿来——这些指标数值是怎么得到的》进行讨论。

Android 图像渲染流程

下图是笔者结合各类资料（主要是是源码及官方文档），在根据自己的理解梳理出的几种常见场景下的图像渲染流程：

> PS 1：笔者个人技术水平有限，若存在理解有误的地方还望指正。
> PS 2：本文主要讨论的 Android 源码为 Android 6.0



备注：基于 OpenGL 的应用可以使用 Choreographer 中的 VSYNC 信号来进行图像渲染工作的安排。

上面这幅图涉及的概念较多，要完全吃透估计得费不少时间。不过好在我们只是想弄明白显示性能各种指标的含义，所以我们只需要理清下面两大关系即可：

SurfaceFlinger、HWComposer与Surface的关系

- **Surface**：可以理解为Android系统中的一个基本显示单元。只要使用Android任意一种API绘图，绘制的结果都将反映在Surface上。

SurfaceFlinger：服务运行在System进程中，用来统一管理系统的帧缓冲区设备，其主要作用是将系统中的大部分Surface进行合成。SurfaceFlinger主要使用GPU进行Surface的合成，合成的结果将形成一个FrameBuffer。

HWComposer：即Hardware Composer HAL，其作用是将SurfaceFlinger通过GPU合成的结果与其他Surface一起最终形成BufferQueue中的一个Buffer。此外，HWComposer可以协助SurfaceFlinger进行Surface的合成，但是否进行协助是由HWComposer决定的。

值得注意的是，有的Surface不由WindowManager管理，将直接作为HWComposer的输入之一与SurfaceFlinger的输出做最后的合成。

Choreographer、SurfaceFlinger、HWComposer与VSYNC的关系

- **VSYNC**：Vertical Synchronization的缩写，它的作用是使GPU的渲染频率与显示器的刷新频率（一般为固定值）同步从而避免出现画面撕裂的现象。
- **HWComposer**：VSYNC信号主要由HWComposer通过硬件触发。
- **Choreographer**：当收到VSYNC信号时，Choreographer将按优先级高低依次去调用使用者通过postCallback提前设置的回调函数，它们分别是：优先级最高的CALLBACK_INPUT、优先级次高的CALLBACK_ANIMATION以及优先级最低的CALLBACK_TRAVERSAL。
- **SurfaceFlinger**：Surface的合成操作也时基于VSYNC信号进行的。

简单来说，Android 图像渲染流程主要由以下特征：

1. 我们可以简单把 Android 图像渲染架构分为应用（Surface）、系统（SurfaceFlinger）、硬件（Screen）三个层级，其中绘制在应用层，合成及提交上屏在系统层，显示在硬件层；
2. 无论应用（Surface）、系统（SurfaceFlinger）、硬件（Screen）都是当且仅当绘制内容发生改变，才会对绘制内容进行处理；
3. 系统中的 SurfaceFlinger 以及绝大部分 Surface 都是按照 VSYNC 信号的节奏来安排自己的任务；
4. 目前，绝大部分 Surface 都属于 Hardware Rendering。

各个指标在 Android 图像渲染流程所代表的意义

大致梳理了 Android 的图像渲染流程之后，我们需要做的一件事情，就是看看上面提到的指标，都对应了渲染流程的哪些阶段，这样对于我们了解各个指标所反映的具体物理意义及其优势劣势都有极大帮助。再次强调，在这个环节之中，我们的讨论仅限于只讨论指标所对应的渲染流程的具体阶段，各指标的具体定义由第二章具体展开。

系统层级（SurfaceFlinger）的显示性能指标

- 基础数据：SurfaceFlinger 合成次数
- 指标意义：
  - 系统合成帧率：FPS
- 特别说明：
  - SurfaceFlinger 仅在显示区域内的 Surface 有提交内容更新时才会进行合成（上屏），因此，系统合成帧率低并不一定意味着图像显示性能差，有可能是因为当前并没有任何的内容更新所导致。
  - 若显示区域内的某个待测 Surface 持续进行更新时， SurfaceFlinger的合成（上屏）的频率可以在某种程度上反映该 Surface 的显示性能，但从理论上分析该指标并不一定准确。这是因为，若显示区域内尚存在其他 Surface，它们也会影响 SurfaceFlinger 的合成（上屏）的行为，从而干扰结果。
  - 若某个 Surface 的合成不在 SurfaceFlinger 中进行（如 Camera Preview），则该 Surface 的显示性能无法用这类指标进行衡量。

应用层级（Surface）的显示性能指标

- 基础数据：绘制过程中每一帧的关键时间点（如开始绘制时间、结束绘制时间等）
- 指标意义：
  - 应用绘制帧率：Frame rate
  - 应用绘制轮询频率：SM
  - 应用绘制超时（跳帧）的次数：Aggregate frame stats、Jankiness count、Skipped frames
  - 应用绘制超时（跳帧）的幅度：Aggregate frame stats、Max accumulated frames、Skipped frames
- 特别说明：

- 与 SurfaceFlinger 类似， Surface也仅在有内容更新时才会进行绘制，因此，绘制频率低并不一定意味着图像显示性能差，有可能是因为当前并没有任何的内容更新所导致。
- 如 SM、Skipped frames 这类指标，由于其基础数据取自 Choreographer，若 某些 Surface 的绘制不依赖于 Choreographer ，则这些指标无法衡量该 Surface 的显示性能。
- 如 Aggregate frame stats、Jankiness count、Max accumulated frames、Frame rate 这类指标， 由于其基础数据仅在硬件绘制（Hardware Rendering）过程中进行统计，属于 HWUI 的功能，所以非硬件绘制的 Surface 自然无法使用这类指标进行衡量。

小结

评价显示性能的各个指标，可以按其在图像渲染流程中的作用，分为以下两类：

1. 系统层级的指标仅有 FPS 一根独苗，它的限制是 Surface 的和合成需要在 SurfaceFlinger中进行；
2. 应用层级的指标较多，它们之中又可以分为两类： 1） SM、Skipped frames 需要 Surface 依赖 Choreographer进行绘制，才能正常工作； 2） Aggregate frame stats、Jankiness count、Max accumulated frames、Frame rate 属于 HWUI 的功能， 需要 Surface 的绘制由 HWUI 进行才能进行分析。

Step 2：你从哪儿来——这些指标数值是怎么得到的

第一章的内容仅仅是站在整个图像绘制流程的高度来简单分析各个指标的，本章将进一步分析各个指标的基础数据来源以及具体计算方式。

基础数据：系统层级（SurfaceFlinger）的合成（上屏）的次数

前面说到，在 Android 系统中，SurfaceFlinger 扮演了系统中所有 Surface 的管理者的角色，当应用程序所对应的 Surface 更新之后，绝大多数的 Surface 都将在 SurfaceFlinger 之中完成了合并的工作之后，最终才会在 Screen 上显示出来。

当然， SurfaceFlinger 的执行也是由 VSYNC 信号驱动的，这也决定了每秒钟合成次数的上限就是 60 次。当 SurfaceFlinger 接收到 Surface 更新通知的时候，将会由 SurfaceFlinger::handleMessageRefresh 函数进行处理，其中包含重建可见区域、初始化、合成等步骤。这里，我们主要关注 SurfaceFlinger::doComposition() 这个方法。

```
void SurfaceFlinger::handleMessageRefresh() {
  ...
  if (CC_UNLIKELY(mDropMissedFrames && frameMissed)) {
    // Latch buffers, but don't send anything to HWC, then signal another
    // wakeup for the next vsync
    preComposition();
    repaintEverything();
  } else {
    preComposition();
    rebuildLayerStacks();
    setUpHWComposer();
    doDebugFlashRegions();
    doComposition(); //重点关注对象
    postComposition();
  }
  ...
}
```

在 doComposition 中，完成 Surface 的合成之后，都会调用 DisplayDevice::flip()，它会使用变量 mPageFlipCount 统计我们进行合成的次数，这个变量就是我们统计 FPS 的核心原始数据。mPageFlipCount 记录了 SurfaceFlinger 一共进行了多少次合成，也可以简单理解为，SurfaceFlinger 向屏幕提交了多少帧的数据。

```
void SurfaceFlinger::doComposition() {
  ATRACE_CALL();
  const bool repaintEverything = android_atomic_and(0, &mRepaintEverything);
  for (size_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {
    const sp<DisplayDevice>& hw(mDisplays[dpy]);
    if (hw->isDisplayOn()) {
      ...
      hw->flip(hw->swapRegion);//重点关注对象
      ...
    }
    // inform the h/w that we're done compositing
    hw->compositionComplete();
  }
  postFramebuffer();
}

void DisplayDevice::flip(const Region& dirty) const {
  ...
  mPageFlipCount++;
}
```

不仅如此， Android 还为我们获取这个基础数据提供了比较方便的方法。通过执行 adb 命令：service call SurfaceFlinger 1013，我们就可以得出当前的 mPageFlipCount。

```
C:\Users\xiaosongluo>adb shell
shell@cancro:/ $ su
su
root@cancro:/ # service call SurfaceFlinger 1013
service call SurfaceFlinger 1013
Result: Parcel(00aea4f4    '....')
```

FPS 的计算方法

根据 FPS 的定义，我们不难逆推得出 FPS 的计算方法：

在 t1 时刻获取 mPageFlipCount 的数值 v1，在在 t2时刻获取 mPageFlipCount 的数值 v2，FPS 的计算公式：

```
FPS = (v2 - v1) / (t2 - t1);
```

需要注意的是：mPageFlipCount 的原始数据是 16 进制的，一般而言计算之前需要先进行进制转换。

基础数据：应用层级（Surface）的绘制过程中每一帧的关键时间点（FrameInfo）

> 请大家先注意 FrameInfo 是由 Android 6.0（具体来讲是 Android M Preview） 引入到 HWUI 模块中的统计功能。 因此，目前来讲绝大多数系统上的大多数应用都暂时无法获取这一基础数据。不过 This IsTheFuture。

我们再来仔细瞧瞧 Google 给出的显示性能测试的十全大补丸 《Testing Display Performance : Aggregate frame stats》(http://developer.android.com/intl/zh-cn/training/testing/performance.html#aggregate) 。其中，特别值得关注的是 adb shell dumpsys gfxinfo <PACKAGE_NAME> framestats 这一条命令。通过这条命令，我们获取每一帧绘制过程中每个关键节点的耗时情况，从而仔细的分析潜在的性能问题。

不得不说，按照 Google 给出的这种测试方法进行测试得到的显示性能数据是非常全面的。

这些基础数据都是记录在 FrameInfo (https://android.googlesource.com/platform/frameworks/base/+/master/libs/hwui/FrameInfo.cpp) 之中，由 CanvasContext (https://android.googlesource.com/platform/frameworks/base/+/master/libs/hwui/renderthread/CanvasContext.cpp) 在doFrame（）时进行记录。相关的主要源码如下：

```
//源码：FrameInfo.cpp
#include "FrameInfo.h"
#include <cstring>

namespace android {
  namespace uirenderer {
    const std::string FrameInfoNames[] = {
      "Flags",
      "IntendedVsync",
      "Vsync",
      "OldestInputEvent",
      "NewestInputEvent",
      "HandleInputStart",
      "AnimationStart",
      "PerformTraversalsStart",
      "DrawStart",
      "SyncQueued",
      "SyncStart",
      "IssueDrawCommandsStart",
      "SwapBuffers",
      "FrameCompleted",
    };

    void FrameInfo::importUiThreadInfo(int64_t* info) {
      memcpy(mFrameInfo, info, UI_THREAD_FRAME_INFO_SIZE * sizeof(int64_t));
    }
  } /* namespace uirenderer */
} /* namespace android */
```

Aggregate frame stats 指标的计算方法

首先需要说明的是 Aggregate frame stats 不是一个指标，而是一系列指标集合。我们来看一个具体的 Aggregate frame stats 的例子：

Stats since: 752958278148ns
Total frames rendered: 82189
Janky frames: 35335 (42.99%)
90th percentile: 34ms
95th percentile: 42ms
99th percentile: 69ms
Number Missed Vsync: 4706
Number High input latency: 142
Number Slow UI thread: 17270
Number Slow bitmap uploads: 1542
Number Slow draw: 23342

以上统计信息的实现可以详见源码：GfxMonitorImpl.java (https://android.googlesource.com/platform/frameworks/janktesthelper/+/master/src/main/java/android/support/test/jank/internal/GfxMonitorImpl.java)

在 Android M 以上的系统上，上述信息的获取十分方便（事实上也只有这些系统能够获取这些信息）。仅需要执行以下命令即可：

```
adb shell dumpsys gfxinfo <PACKAGE_NAME>
```

Jankiness count、Max accumulated frames、Frame rate 指标的计算方法

首先需要说明的是：Jankiness count、Max accumulated frames、Frame rate 与 Aggregate frame stats的基础数据并不一致，它们的基础属于来源于 gfxinfo（Profile data in ms）。

只是在 Android M 中 gfxinfo（Profile data in ms）的基础数值来源于 FrameInfo，详见源码：FrameInfoVisualizer (https://android.googlesource.com/platform/frameworks/base/+/master/libs/hwui/FrameInfoVisualizer.cpp)。但在更早的系统之上， gfxinfo（Profile data in ms）的数值也可以获取。

这里需要特别指出的是， gfxinfo（Profile data in ms）只保存了 Surface 最近渲染的128帧的信息，因此，Jankiness count、Max accumulated frames、Frame rate 也仅仅是针对这 128 帧数据所计算出来的结果,它们的具体含义分别是：

- Jankiness count：根据相邻两帧绘制时间的差值，“估计”是否存在跳帧并进行跳帧次数的统计；
- Max accumulated frames： 根据相邻两帧绘制时间的差值，“估计”这 128 帧绘制过程中可能形成的最大连续跳帧数；
- Frame rate：计算所得平均（绘制）帧率。

如果你对具体的计算过程感兴趣，可以参考详见源码：JankTestBase (https://android.googlesource.com/platform/frameworks/testing/+/master/uiautomator_test_libraries/src/com/android/uiautomator/platform/JankTestBase.java)

基础数据：应用层级（Surface）的绘制过程中每一帧的关键时间点（Choreographer）

先说一句有点绕口的话： Choreographer 是依据 Choreographer 绘制的 Surface 在 UI 绘制过程中最为核心的机制。

Choreographer 的工作机制简单来说就是，使用者首先通过 postCallback 在 Choreographer 中设置的自己回调函数：

- CALLBACK_INPUT：优先级最高，和输入事件处理有关。
- CALLBACK_ANIMATION：优先级其次，和Animation的处理有关。
- CALLBACK_TRAVERSAL：优先级最低，和UI等控件绘制有关。

那么，当 Choreographer 接收到 VSYNC 信号时，Choreographer 会调用 doFrame 函数依次对上述借口进行回调，从而进行渲染。

那么显然，doFrame 的执行效率（次数、频率）也就是我们需要的显示性能数据。而这样的基础数据，Choreographer 自身也进行了记录。如下面代码中，jitterNanos 记录了绘制前后两帧所间隔的时间差， 而 skippedFrames 则记录了 jitterNanos 这段时间 doFrame 错过了多少个 VSYNC 信号，即跳过了多少帧。

```
// Set a limit to warn about skipped frames.
// Skipped frames imply jank.
private static final int SKIPPED_FRAME_WARNING_LIMIT = SystemProperties.getInt("debug.choreographer.skipwarning", 30);

void doFrame(long frameTimeNanos, int frame) {
  ...
  final long jitterNanos = startNanos - frameTimeNanos;
  if (jitterNanos >= mFrameIntervalNanos) {
    final long skippedFrames = jitterNanos / mFrameIntervalNanos;
    if (skippedFrames >= SKIPPED_FRAME_WARNING_LIMIT) {
      Log.i(TAG, "Skipped " + skippedFrames + " frames! " + "The application may be doing too much work on its main thread.");
    }
    ...
    final long lastFrameOffset = jitterNanos % mFrameIntervalNanos;
    ...
    frameTimeNanos = startNanos - lastFrameOffset;
  }
  ...
}
```

上述数据的获取并不是那么的直接，所以需要一定的手段。方法一共有三种，都不难：

- **Logcat 方案**

> 缺点：该方案需要系统授权 “Adb Root” 权限，用于修改系统属性；对于丢帧信息只能统计分析，无法进行实时处理。
> 优点：设置完成后，可以获取系统中所有应用各自的绘制丢帧情况（丢帧发生的时间以及连续丢帧的数量）。

其实，仔细观察代码，我们就可以注意到 Choreographer 源码中本身就有输出的方案：

```
if (skippedFrames >= SKIPPED_FRAME_WARNING_LIMIT) {
  Log.i(TAG, "Skipped " + skippedFrames + " frames! " + "The application may be doing too much work on its main thread.");
}
```

唯一阻碍我们获取数值的是：skippedFrames 的数值只有大于 SKIPPED_FRAME_WARNING_LIMIT 才会输出相关的警告。而 SKIPPED_FRAME_WARNING_LIMIT 的数值可以由系统参数 debug.choreographer.skipwarning 来设定。

> 注意：初始条件下，系统中不存在 debug.choreographer.skipwarning 参数，因此 SKIPPED_FRAME_WARNING_LIMIT 将取默认值 30。因此，正常情况下，我们能够看见上诉 Log 出现的机会极少。

因此，如果我们修改（设定）系统属性 debug.choreographer.skipwarning 为 1，Logcat 中将打印出每一次丢帧的Log。需要说明的是，由于为 SKIPPED_FRAME_WARNING_LIMIT 赋值的代码段由 Zygote 在系统启动阶段加载，而其他应用都是在拷贝复用 Zygote 中的设定，因此设定系统属性后需要重启 Zygote 才能使得上述设定生效。

具体的设置方法如下：

```
setprop debug.choreographer.skipwarning 1
setprop ctl.restart surfaceflinger; setprop ctl.restart zygote
```

设定完成以后，我们可以直接通过 Logcat 中的信息得到系统中所有应用的绘制丢帧信息，包括丢帧发生的时间以及连续丢帧的数量。不过由于 Logcat 信息的滞后性，以上信息我们几乎只能进行在测试完成后进行统计分析，而无法进行实时处理。

- **Choreographer.FrameCallback 方案**

> 缺点：该方案需要将测试代码与待测应用打包在一起，因此理论上仅能测试自己开发的应用。
> 优点：可以对丢帧信息进行实时处理

我们先来看看 Choreographer.FrameCallback (http://developer.android.com/intl/zh-cn/reference/android/view/Choreographer.FrameCallback.html) 的定义。

> Implement this interface to receive a callback when a new display frame is being rendered. The callback is invoked on theLooper thread to which the Choreographeris attached.

通过这个接口，我们可以在每一帧被渲染的时候记录下它开始渲染的时间，这样在下一帧被处理是，我们不仅可以判断上一帧在渲染过程中是否出现掉帧，而整个过程都是实时处理的，这为我们可以及时获取相关的调用栈信息来辅助定位潜在的性能缺陷有极大的帮助。

- **代码注入方案**

> 缺点：该方案需要通过注入程序为指定应用注入测试代码，因此需要系统为注入程序授权 "应用Root” 权限。
> 优点：与 Choreographer.FrameCallback 方案一致。

该方案可以简单理解为通过注入的方式来实现与 Choreographer.FrameCallback 方案一样的目的。因此，这里我们主要讨论两者在实现方式上的区别。

显而易见，我们需要注入的对象是 Choreographer ，因此理论上任何第三方应用都是可以被注入的。但是随着 Android 系统对"应用Root” 权限管理越来越严格，所以该方案可用的范围越来越小。

SM 指标的计算方法

根据定义，SM 其实类似于 FPS，它被设计为可以衡量应用平均每秒执行 doFrame（ ）的次数。我们可以认为它是在衡量 Surface 渲染轮询的次数。

针对 **Logcat 方案**，我们只需统计测试过程中目标进程一共掉了多少帧，由于对于绝大多数应用在没有丢帧的情况下会针对每一次 VSYNC 信号执行一次 doFrame（ ），而 VSYNC 绝大多数情况下每秒会触发 60 次，因此我们可以反向计算得出 SM 的数值：

```
SM = (60* totalSeconds - totalSkippedFrames) / totalSeconds;
```

针对 **Choreographer.FrameCallback 方案** 以及 **代码注入方案**，我们需要在代码中自己进行统计输出（可以是设计成实时的，也可以设计成测试结束后进行统计计算的）。

Skipped frames 指标的计算方法

这个指标的就是指当前应用在丢帧发生时的丢帧帧数。

针对 **Logcat 方案**， 该数值直接在 Logcat 中输出，并且带有时间信息。

```
04-18 16:31:24.957 I/Choreographer(24164): Skipped 4 frames!  The application may be doing too much work on its main thread.
04-18 16:31:25.009 I/Choreographer(24164): Skipped 2 frames!  The application may be doing too much work on its main thread.
```

针对 **Choreographer.FrameCallback 方案** 以及 **代码注入方案**，我们可能很方便的通过计算前后两帧开始渲染的时间差获得这一数值，同样方便。同样与 **Logcat 方案** 不同的是，它也是可以设计成实时计算的。

小结

通过对各个显示性能指标的分析，我们可以知道，虽然目前指标众多，但其实有本质区别的指标确很少：

- 系统层面：
  - 合成（上屏）帧率：FPS
- 应用层面：
  - 跳帧次数：Aggregate frame stats、Jankiness count、Skipped frames
  - 跳帧幅度：Aggregate frame stats、Max accumulated frames、Skipped frames
  - 绘制帧率：Frame rate
  - 绘制轮询频率：SM

更为重要的是，我们从上述的分析中知道了各个指标都有着自己的优势和不足，这也从根本上决定了它们各自有各自的用法。

Step 3：你要到哪儿去——这些指标如何落地来指导优化

其实指标的用法也是多种多样的，为了方便讨论，我们仅从日常监控、缺陷定位以及数据上报三个方面来讨论各个显示性能指标是如何落地的。

日常监控

- FPS：数据形式最为直观（FPS 是最早的显示性能指标，而且在多个平台中都有着类似的定义），且对系统平台的要求最低（API level 1），游戏、视频等连续绘制的应用可以考虑选用，但不适用于绝大多数非连续绘制的应用；
- SM：数据形式与 FPS 类似，可以很好的弥补 FPS 无法准确刻画非连续绘制的应用显示性能的缺陷；
- Aggregate frame stats：除了对系统平台有较高的要求以外，其采集方式最为简单（系统自带功能）；
- Skipped frames：与 Aggregate frame stats 类似， 信息量相对较少，但可适用范围更广

> 特别说明：Jankiness count、Max accumulated frames、Frame rate 只统计了128帧的信息（约2~3秒），而且 Jankiness count、Max accumulated frames对于掉帧情况的计算并非是一个准确值，因此这些指标都不太适用于日常监控

举个栗子，笔者服务的某个产品使用如下的一些指标来监控产品与竞品的性能变化情况：

测试指标	**场景1**	**场景2**	**场景3**	**场景4**
FPS	58	58	58	58
SM	59	59	59	59
Num of 6+ Skipped Frames	0	0	0	0
Num of 3+ Skipped Frames	0	0	2	0

> 备注：
>
> 1. Num of x+ Skipped Frames 代表测试过程中发生连续丢 x 帧（及以上）的次数;
> 2. 至于为什么我们选择关注连续丢 3 帧以及连续丢 6 帧的的次数，在【缺陷定位】部分有相关的分析讨论；

缺陷定位

- Skipped frames：基于 Choreographer.FrameCallback 方案实现的 Skipped frames 指标，可以在卡顿 (http://bugly.qq.com/)出现的时刻获取应用堆栈信息，可以在一定程度上进行缺陷定位

> 特别说明：
>
> 1. FrameInfo 相关指标无法直接进行缺陷定位 (http://bugly.qq.com/)，但 FrameInfo 当中包含了大量详尽的绘制基础数据，对于缺陷定位也有较大帮助；
> 2. 关于缺陷定位 (http://bugly.qq.com/)过程中连续掉帧阈值的选取，可参考维基百科中提到几个重要的帧率数值：
>
> - 12 fps：由于人类眼睛的特殊生理结构，如果所看画面之帧率高于每秒约10-12帧的时候，就会认为是连贯的
> - 24 fps：有声电影的拍摄及播放帧率均为每秒24帧，对一般人而言已算可接受
> - 30 fps：早期的高动态电子游戏，帧率少于每秒30帧的话就会显得不连贯，这是因为没有动态模糊使流畅度降低
> - 60 fps：在实际体验中，60帧相对于30帧有着更好的体验
>
> 以上各数据分别对应： 0 帧、1帧、2.5帧、5~6帧。（这就是为啥选择3/6的原因）

举个栗子， 笔者的同事万大师（yuwan）基于上述原理自研了一款性能分析工具。该工具在集成于待测应用之后，可以自动保存如下的性能缺陷信息：

```
Frame lost：... （连续丢帧数量，一般我们会设定一个阈值，例如 6 帧以上我们才会进行记录）
User action：...（当前用户操作）
Stack trace： ...（当前堆栈信息）
```

有了这个工具之后，我们可以收集应用的各个潜在“卡顿”点，用于进一步的分析和优化。

数据上报

- Aggregate frame stats：除了对系统平台有较高的要求以外，其采集方式最为简单（系统自带功能）、数据也比较清晰，相信基于这类指标实现性能数据上报是特别方便的
- Skipped frames ：基于 Choreographer.FrameCallback 方案实现的 Skipped frames 指标，采集方式简单，实现基础性能数据上报、卡顿数据上报也是很方便的

这方面应用，笔者所服务的产品暂时没有涉及，就不举例子了。如果各位观众感兴趣，建议参考 Android ANR 的设计理念。

小结

发现了没有 Skipped frames 的用处很大有没有？ 而且通读全篇，你会发现 Aggregate frame stats、Jankiness count、Max accumulated frames 这些指标都有提供类似的功能。

至于为什么，这就不是本文需要讨论的内容了，如果大家比较感兴趣，笔者这里给出两份相关的链接以供各位参考：

Fps Versus Frame Time (http://www.mvps.org/directx/articles/fps_versus_frame_time.htm)
量化和优化用户与 Android 设备之间的交互 (https://software.intel.com/zh-cn/android/articles/quantify-and-optimize-the-user-interactions-with-android-devices)

友情附赠： 现有显示性能指标对比

本来写到这里本文的主要内容就应该结束了。但是如果不对比一下显示性能指标神马的，总会让人觉得缺少了一些什么。

> 友情提示：下述内容相对主观，建议各位读者依据项目情况自行进行选择。

指标名称	指标意义	基础数据来源	采集方式	适用系统	适用应用	用途
FPS	系统合成帧率	SurfaceFlinger	adb shell	略	略	监控
Aggregate frame stats	应用跳帧次数、幅度	FrameInfo	adb shell	最低23	HW Rendering	监控/上报
Jankiness count	（估算）应用跳帧次数	FrameInfo（128帧）	adb shell	略	HW Rendering	定位
Max accumulated frames	（估算）应用跳帧幅度	FrameInfo（128帧）	adb shell	略	HW Rendering	定位
Frame rate	应用绘制帧率	FrameInfo（128帧）	adb shell	略	HW Rendering	定位
SM	应用绘制轮询频率	Choreographer	多种方式	最低16	SW/HW Rendering 及 部分 OpenGL Rendering	监控
Skipped frames	应用跳帧次数、幅度	Choreographer	多种方式	最低16	SW/HW Rendering 及 部分 OpenGL Rendering	监控/定位/上报

如果您意犹未尽，可以关注我的公众账号：

腾讯 Bugly () 是一款专为移动开发者打造的质量监控工具，帮助开发者快速，便捷的定位线上应用崩溃 (http://bugly.qq.com/)的情况以及解决方案。智能合并功能帮助开发同学把每天上报的数千条 Crash (http://bugly.qq.com/) 根据根因合并分类，每日日报会列出影响用户数最多的崩溃 (http://bugly.qq.com/)，精准定位功能帮助开发同学定位到出问题的代码行，实时上报可以在发布后快速的了解应用的质量情况，适配最新的 iOS, Android 官方操作系统，鹅厂的工程师都在使用，快来加入我们吧…