

Flutter 性能监控工具

写完 Flutter APP 后，在发布之前，需要对 APP 的性能进行测试，这部分就讲一下 Flutter 的性能监控工具。

本次所讲的 Flutter 性能监控工具有两个：

1. PerformanceOverlay
2. Observatory

PerformanceOverlay 介绍

PerformanceOverlay 是在 app 上显示性能统计数据的浮窗。PerformanceOverlay 是一个 Widget，有几个属性，用于控制开启哪些功能。

PerformanceOverlay 分析

PerformanceOverlay 的源码是：

```
~/flutter/packages/flutter/lib/src/widgets/performance_overlay.dart
```

可以看出PerformanceOverlay有以下属性：

1. optionsMask

用于标记哪些功能打开的flag，有四个值：

- displayRasterizerStatistics：显示光栅化器统计信息 (GPU)
- visualizeRasterizerStatistics：可视化光栅化器统计信

息(GPU)

- displayEngineStatistics : 显示引擎统计 (CPU)
- visualizeEngineStatistics : 可视化引擎统计 (CPU)

2. rasterizerThreshold

光栅化的阈值，用于捕获SkPicture跟踪以进行进一步分析，它的值代表每隔几帧捕获一次，默认是0，代表功能是关闭的

3. checkerboardRasterCachelImages

检查缓存图片的情况

4. checkerboardOffscreenLayers

检查不必要的setlayer

开启 PerformanceOverlay

开启 Performance Overlay 的方法有两种：

1. 使用 IDE 里集成的 Flutter Inspector 工具
2. 使用代码设置

1、使用 IDE 里集成的 Flutter Inspector 工具

Flutter Inspector 是集成在 IDE 里的一个强大的工具，Performance Overlay 的功能也集成在 Flutter Inspector 里。

需要我们运行 Flutter APP 后才可以使用的，

1. 在 Android Studio 里

选择 View > Tool Windows > Flutter Inspector，就可以打开 Flutter Inspector，可以看到有很多功能，如下图：

然后选择 Performance Overlay。

2. 在 VS Code里

通过 View > Command Palette 或者 cmd+p 打开命令板，输入 performance 然后选择 Toggle Performance Overlay 就行。

2. 使用代码设置

通过给 MaterialApp 或者 WidgetsApp 设置参数来开启 PerformanceOverlay 的功能。

要将MaterialApp 或者 WidgetsApp 的showPerformanceOverlay 属性设置为true，如下：

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      showPerformanceOverlay: true,  
      title: 'My Awesome App',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(title: 'My Awesome App'),  
    );  
  }  
}
```

在 PerformanceOverlay 里查看 GPU、CPU 统计信息

如下图：

这个浮窗有两个图形：

1. 上面的是 GPU thread

表示在 GPU 线程上生成每帧需要的时间。

2. 下面的 UI thread

表示在 UI 线程上生成每帧需要的时间。

3. 纵轴

图中的纵轴代表时间，每个图形都被分成三格，每小格代表 16ms，如果图中超过这三条线之一，那么您的运行频率低于 60Hz。

4. 横轴

横轴代表帧。该图仅在应用程序绘制时更新，因此如果它处于空闲状态，该图将停止移动。每个图形绘制的都是该线程最后 300 帧 的数据。

为了保证 60FPS，每帧耗费的时间应该是小于 16ms 的，看上图中绿色的粗线条，代表的是当前帧的数据，如果当前帧的数据符合预期（<16ms），那么就是绿色的，如果不符合，就是红色的，如下图：



遇到红色的就需要去具体分析，因为会造成 APP 卡顿：

1. 如果 GPU thread 是红色的

那么可能是绘制的图形过于复杂，或者是执行了过多的 GPU 操作。

2. 如果 UI thread 是红色的

说明肯定是 Dart 代码里有耗时操作，导致阻塞了 UI 操作。

3. 如果两个都是红色的

建议从 UI thread，也就是 Dart 代码查起。

GPU 问题定位

辅助定位 GPU 问题的，就是 PerformanceOverlay 的下面两个功能：

1. checkerboardRasterCacheImages -- 检查缓存图片的情况

还有一个拖慢 GPU 渲染速度的是没有给静态图像做缓存，导致每次 build 都会重新绘制。我们可以把静态图像加到 RepaintBoundary 中，引擎会自动判断图像是否复杂到需要 repaint boundary。

2. checkerboardOffscreenLayers -- 检查不必要的 setlayer

setlayer 是 Canvas 里的操作，非常耗性能，所以要尽量避免这个操作。你可能会好奇，我根本没用到这个，是的，我们一般是不会使用这个函数的，所以当你开启了这个检查后，会发现啥也没有，正好说明了你没有用到 setlayer。我们一般都是被动的使用，比如需要剪切、透明的操作，用到一些 widget

的时候，这些 widget 会用到 `setlayer` 方法，一旦遇到这种情况，我们要想一下是否一定要这么做，能不能通过其他方式实现。

开启这两个功能的方法：

```
MaterialApp(  
  showPerformanceOverlay: true,  
  checkerboardOffscreenLayers: true, //使用了  
saveLayer的图像会显示为棋盘格式并随着页面刷新而闪烁  
  checkerboardRasterCacheImages: true, // 做了缓  
存的静态图像图片在刷新页面使不会改变棋盘格的颜色；如果棋盘  
格颜色变了，说明被重新缓存，这是我们要避免的  
  ...  
);
```

GPU 优化

1. 一些效果尽量设置在子 Widget 上，而不是父 Widget

例如，要实现一个混合图层的半透明效果，如果把透明度设置在顶层 Widget 上，CPU 会把每个子Widget的图层渲染出来，在执行 `saveLayer` 操作保存为一个图层，最后给这个图层设置透明度，但是 `saveLayer` 的开销很大，所以官方给出建议：首先确认这些效果是否真的有必要；如果有必要，应该把透明度设置到每个子Widget上，而不是父Widget。裁剪操作也是类似。

UI问题定位

可以使用 Observatory 里的 timeline 的功能分析。

Flutter 性能监控工具 -- Observatory 介绍

Flutter 的宣传说，使用 Flutter 可以达到 60FPS，但是这并不意味，不管你怎么写，都能达到 60FPS，如果有耗时的操作，可能会阻塞 UI 的渲染，如果内存使用过多，也有可能会 OOM，所以性能的好坏，是保证 Flutter 能否达到 60FPS 的关键，在对 Flutter 进行性能优化之前，我们先看下如何来监测 Flutter 的性能。

Flutter 的三种构建模式(build modes)

Flutter 有三种构建模式，适用于不同的场景：

1.Debug

顾名思义，就是调试模式，在调试模式下：

1. Flutter 的断言 (Assertions) 功能是开的
2. Flutter 的 Observatory 是开的，Observatory 是用于分析和调试 Dart 代码的工具，用于 Dart 的 debugger
3. 扩展的服务功能 (Service extensions) 是开启的，如第二点的 Observatory 的服务还有性能的服务等。
4. JIT 编译模式，可以使用 Hot Reload，为了快速开发

可以看到，在 Debug 模式下，为了 debug 和快速开发，牺牲了性能，所以 Debug 模式都是用在开发阶段。
而且模拟器只能运行 Debug 模式。

用命令行：

```
$flutter run
```

2.Release

顾名思义，就是要发布了，在 Release 模式下，要追求最高的性能和最小的安装包，所以会：

1. 断言 (Assertions) 功能关闭
2. 没有 Debugging 的信息
3. Debugger 的功能关闭
4. AOT 编译, 为了快速启动, 快速执行和更小的安装包大小。
5. 扩展的服务功能 (Service extensions) 关闭

所以, Release 模式是 APP 要发布的时候才用。

Release 模式只能跑在真机上。

用命令行:

```
$flutter run --release
```

或者

```
$flutter build
```

3.Profile

Profile 是专门监控性能的模式, 在 Debug 模式下, 不能实际反应应用的性能, 而在 Release 模式下, 却没有监控的功能, 所以就诞生了 Profile 模式, Profile 模式和 Release 模式更接近, Profile 和 Release 都采用的 AOT 编译, 所以都不能用 Hot Reload, 但是 Profile 相对于 Release, 多了如下的功能:

1. 一些扩展的服务功能 (Service extensions) 是打开的, 例如监控性能的浮层等。
2. Tracing 是打开的, Observatory 也可以连接到进程

Profile 模式只能跑在真机上。

用命令行:

```
$flutter run --profile
```


使用 Observatory 来监控性能

Observatory 是用于分析和调试 Dart 代码的工具，因为 Flutter 自带 Dart VM，所以也可以用 Observatory。

1. 命令行启动 Observatory

Debug 下启用 Observatory:

```
$flutter run
```

Profile 下启用 Observatory:

```
$flutter run --profile
```

运行完命令后，会看到如下的信息：

```
$ flutter run --profile
Initializing gradle...
0.8s
Resolving dependencies...
6.6s
Launching lib/main.dart on ALP AL00 in profile
mode...
Gradle task 'assembleProfile'...
Gradle task 'assembleProfile'... Done
21.2s
Built build/app/outputs/apk/profile/app-
profile.apk (66.5MB).
Installing build/app/outputs/apk/app.apk...
5.4s
D/mali_winsys(18612): EGLint
new_window_surface(egl_winsys_display *, void *,
EGLSurface, EGLConfig, egl_winsys_surface **,
EGLBoolean) returns 0x3000

An Observatory debugger and profiler on ALP AL00
is available at http://127.0.0.1:57535/
For a more detailed help message, press "h". To
quit, press "q".
```

这一句:

```
available at http://127.0.0.1:57535/
```

打开 <http://127.0.0.1:57535/> 这个网址，就会看到如下的界面：

Observatory 支持如下的功能：

1. Allocation Profile

2. Code Coverage
3. CPU Profile
4. Debugger
5. Evaluating Expressions
6. Heap Map
7. Isolate
8. Metrics
9. User and VM Tags

部分功能的截图如下：

2.Flutter Inspector 启动

Flutter Inspector 是一个强大的工具，要想使用 Flutter Inspector，就得先运行 Flutter APP。

运行 Flutter APP后：

1. 在 Android Studio 里

选择 View > Tool Windows > Flutter Inspector，就可以打开 Flutter Inspector，可以看到有很多功能，如下图：

然后选择 Open observatory。

2. 在 VS Code里

通过 View > Command Palette 或者 cmd+p 打开命令板，输入 Open observatory 然后选择 Open observatory就行。



如果看不到，说明你没有运行 Flutter APP，得以 Start Debugging 的方式运行。

Flutter 性能监控工具 -- Observatory 使用

前面主要讲了如何打开 Observatory，现在讲一下如何使用 Observatory 里的各个功能来分析性能问题，

下图是 Observatory 的主页面：

最上面的蓝色导航栏

表示当前所在的位置

1. `vm@ws://127.0.0.1:50579/ws`：表示当前连接的VM

鼠标放在这个上面，会看到这个，表示当前 APP 的 `main()`：

2. 右边的 Refresh：刷新数据

VM

这里显示当前 VM 的信息

1. name：当前 VM 的名字
2. version：Dart 的版本，APP build 的时间，运行在哪个平台上
3. embedder：嵌入的平台
4. started at：VM 启动时的时间戳
5. uptime：VM 已运行的时长
6. refreshed at：上次采样数据的时间

7. pid : 进程 ID
8. peak memory : APP 运行时用的峰值内存
9. current memory : APP 当前用的内存
10. native zone memory : native 原生内存
11. native heap memory : native 堆内存
12. native heap allocation count : native 堆对象数量
13. flag : 一些标记位
14. timeline : 工具
15. native memory profile :

Isolates

这里是显示 APP 里的 Isolates, 每个 APP 都有一个 root Isolates。

Isolates 是用一个饼图显示, 这个饼图有很多组成部分, 我们把这些叫做 tags, 这些 tags 用来代表在 VM 里不同的执行列表, 包括 User TAG 和 VM TAG:

- User TAG

用户自定义的 TAG, 使用方式如下:

```
import 'dart:developer';

var customTag = new UserTag('MyTag');

// Save the previous tag when installing the
// custom tag.
var previousTag = customTag.makeCurrent();

// your code here

// Restore the previous tag.
previousTag.makeCurrent();
```

- VM TAG

VM TAG 是系统定义的，在整个 UI 中使用，有：

1. CompileOptimized, CompileScanner, CompileTopLevel, CompileUnoptimized：编译 Dart 代码
2. GCNewSpace：新生代的垃圾收集
3. GCOldSpace：老生代的垃圾收集
4. Idle：不是 VM TAG，但是用来标识 Isolate 空闲的占比
5. Native：执行Native代码，dart: io 库使用的 C++ 代码 或者 平台相关代码(Platform Channel)
6. Runtime：执行 Runtime 代码
7. Dart：执行 自己的业务 Dart 代码
8. VM：创建 isolate，和其他未被覆盖的部分

图像的右边还有一系列链接，代表着不同的功能。

debug

可以设置断点并调试您的应用程序。

如何设置 debugger, 看 <https://dart-lang.github.io/observatory/debugger.html>

class hierarchy

显示应用程序的类层次结构。

cpu profile

显示当前 isolate 的 CPU 使用数据

图表的下部按 CPU 占用比例做了一个列表，反映的是函数的调用次数和执行时间（划重点）。一般排在前面的函数（这些函数是？有待学习）都不是我们写的 dart 代码。如果你发现自己的某个函数调用占比反常，那么可能存在问题。

具体使用的地址：<https://dart-lang.github.io/observatory/cpu-profile.html>

cpu profile (table)

和 cpu profile 一样，但是是以表格的形式来展示

allocation profile

显示 isolate 已分配的内存。通过这个面板你能看到新生代/老生代的内存大小和占比；每个类型所占用的内存大小。

Heap 堆，动态分配的 Dart 对象所在的内存空间

- New generation: 新创建的对象，一般来说对象比较小，生命周期短，如local 变量。在这里GC活动频繁
- Old generation: 从GC中存活下来的New generation将会提拔到老生代Old generation，它比新生代空间大，更适合大的对象和生命周期长的对象

通过这个面板你能看到新生代/老生代的内存大小和占比；每个类型所占用的内存大小。

为了 debug 的方便，我们可以获取到某段时间的内存分配情况：点击 Reset Accumulator 按钮，把数据清零，执行一下要测试的程序，点击刷新。

为了检查内存泄露，我们可以点击 GC 按钮，手动执行 GC。

- Accumulator Size:自点击Reset Accumulator以来，累加对象占用内存大小
- Accumulator Instances: 自点击Reset Accumulator以来，累加实例个数
- Current Size: 当前对象占用内存大小
- Current Instances: 当前对象数量

具体使用的地址：<https://dart-lang.github.io/observatory/allocation-profile.html>

heap snapshot

堆快照

heap map

将分配的内存显示为颜色块

heap map 面板能查看 old generation 中的内存状态，它以颜色显示内存块。每个内存页面(page of memory)为256 KB，每页由水平黑线分隔。

例如，蓝色表示字符串，绿色表示双精度表。可用空间为白色，指令（代码）为紫色。如果启动垃圾收集（使用“分配配置文件”屏幕中的GC按钮），堆映射中将显示更多空白区域（可用空间）。将光标悬停在上面时，顶部的状态栏显示有关光标下像素所代表的对象的信息。显示的信息包括该对象的类型，大小和地址。

当你看到白色区域中有很多分散的其它颜色，说明存在内存碎片化，可能是内存泄露导致的。

具体使用看：<https://dart-lang.github.io/observatory/heap-map.html>

metrics

包含您应用中收集的指标。

persistent handles

查看强对象和弱对象

ports

端口

logging

设置Log的级别

