Android Studio Memory Profiler

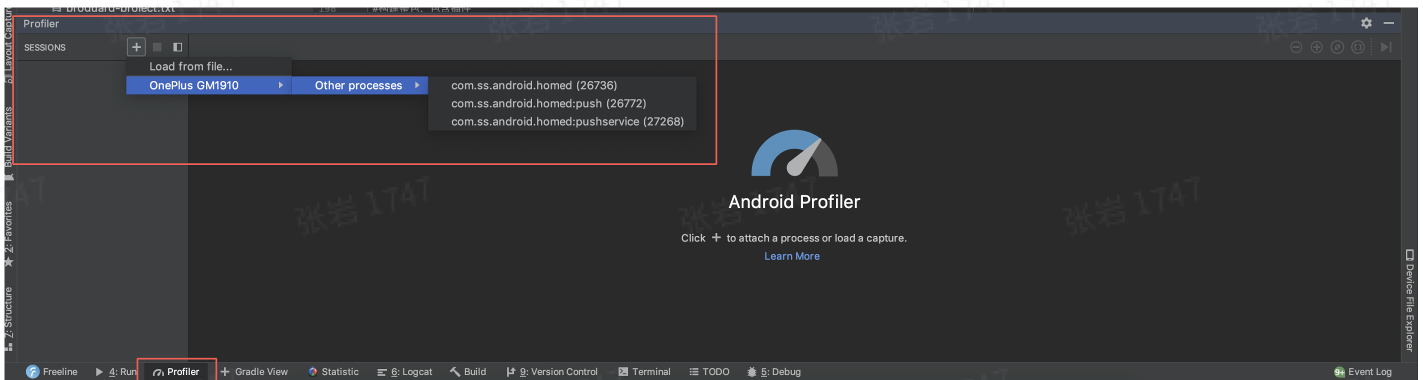
Memory Profiler 干啥的

Memory Profiler 是 Android Profiler 中的一个组件，可帮助我们识别可能会导致应用卡顿、冻结甚至崩溃的内存泄露和内存抖动。它显示一个应用内存使用量的实时图表，让您可以捕获堆转储、强制执行垃圾回收以及跟踪内存分配。

简单来说：就是我们用来精细分析内存使用情况的工具

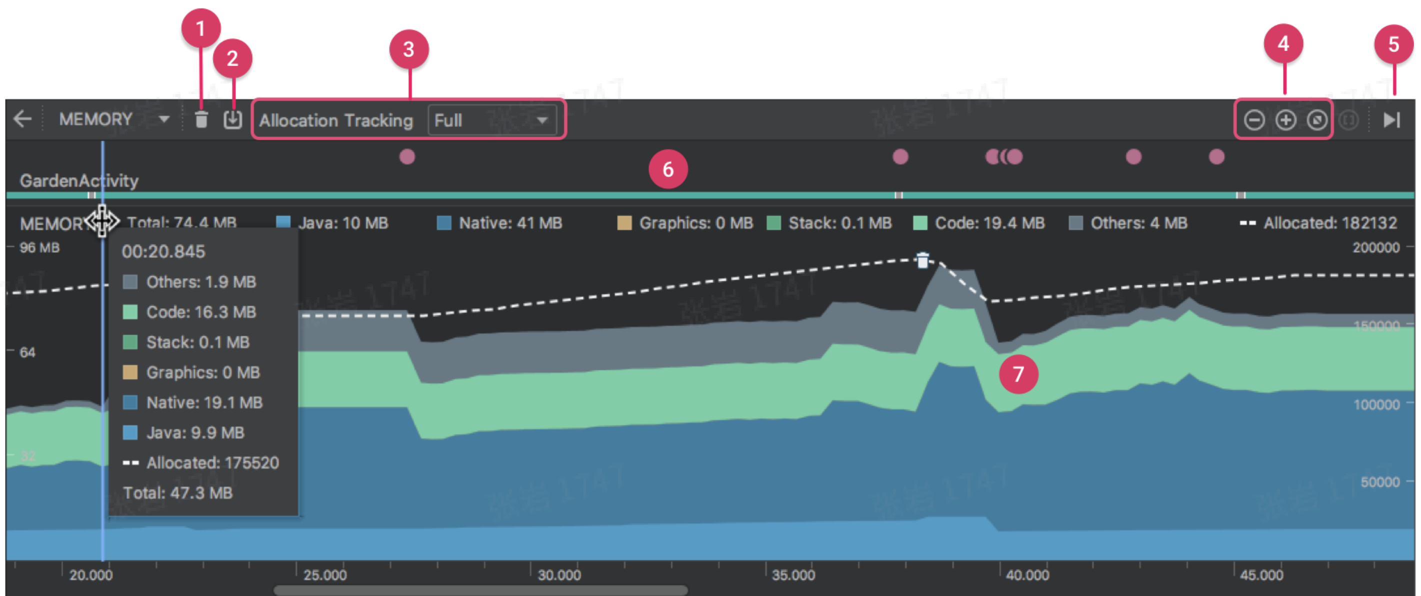
Memory Profiler怎么打开

1. View > Tool Windows > Profiler
2. 手机连接Adb调试
3. 选择SESSIONS 选择设备和对应的调试进程

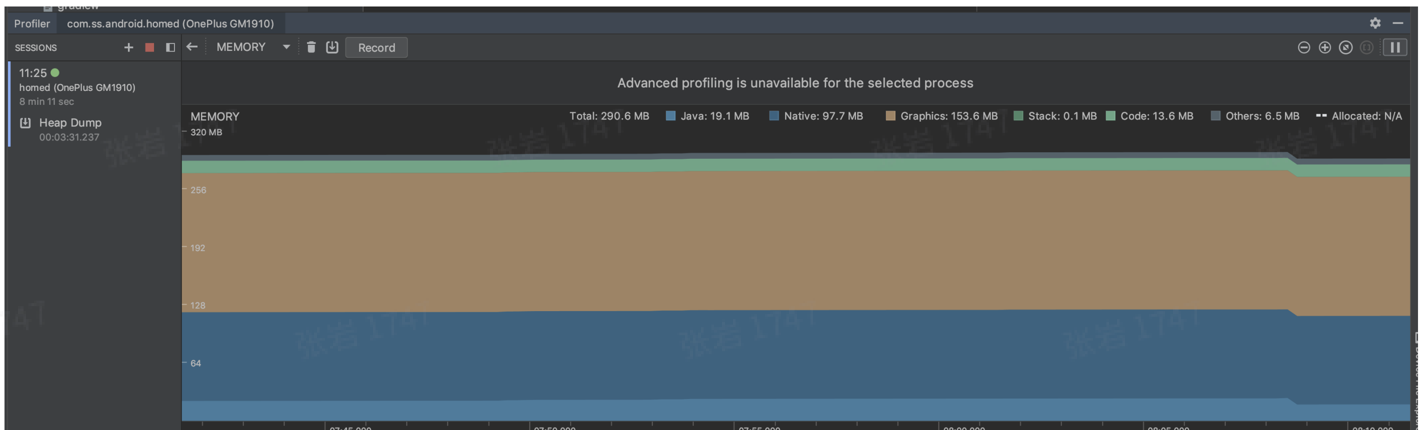


Memory Profiler面板

老版Android studio

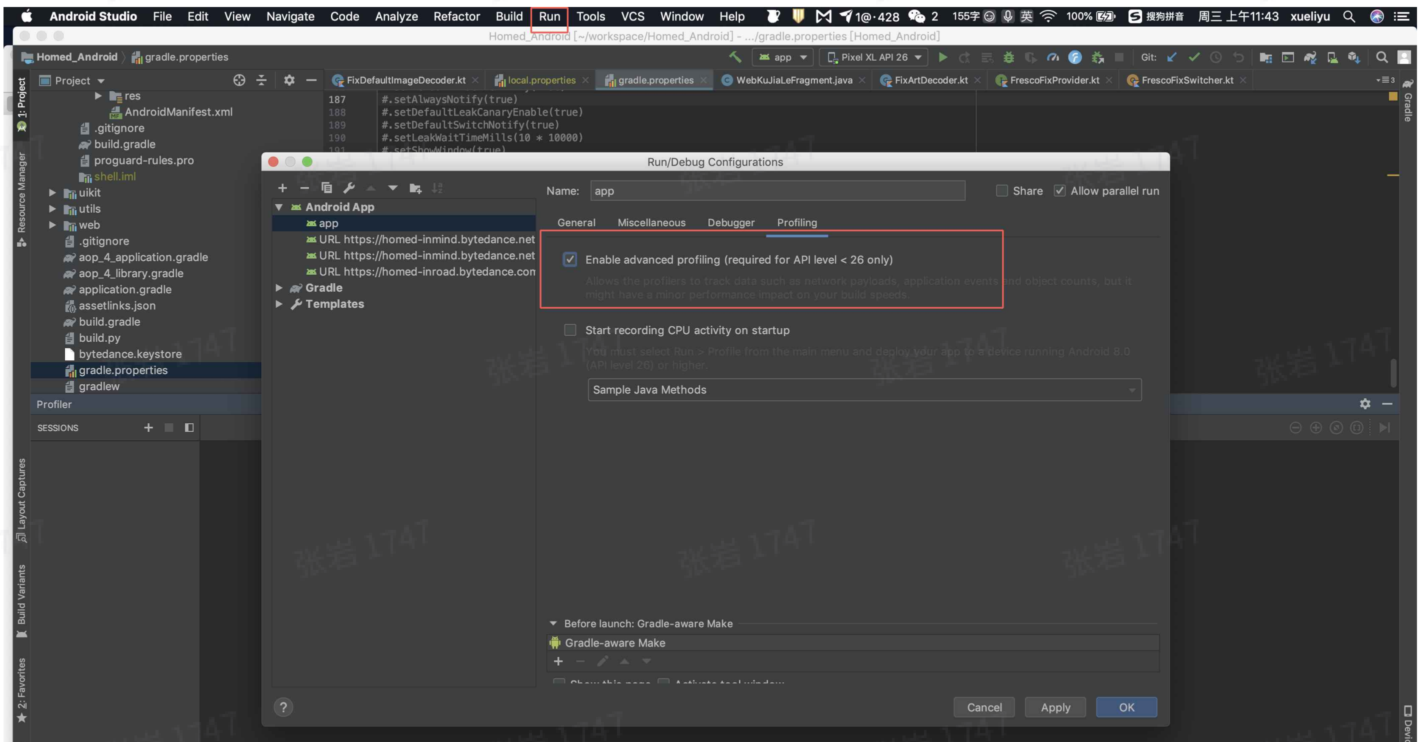


住小帮面板：

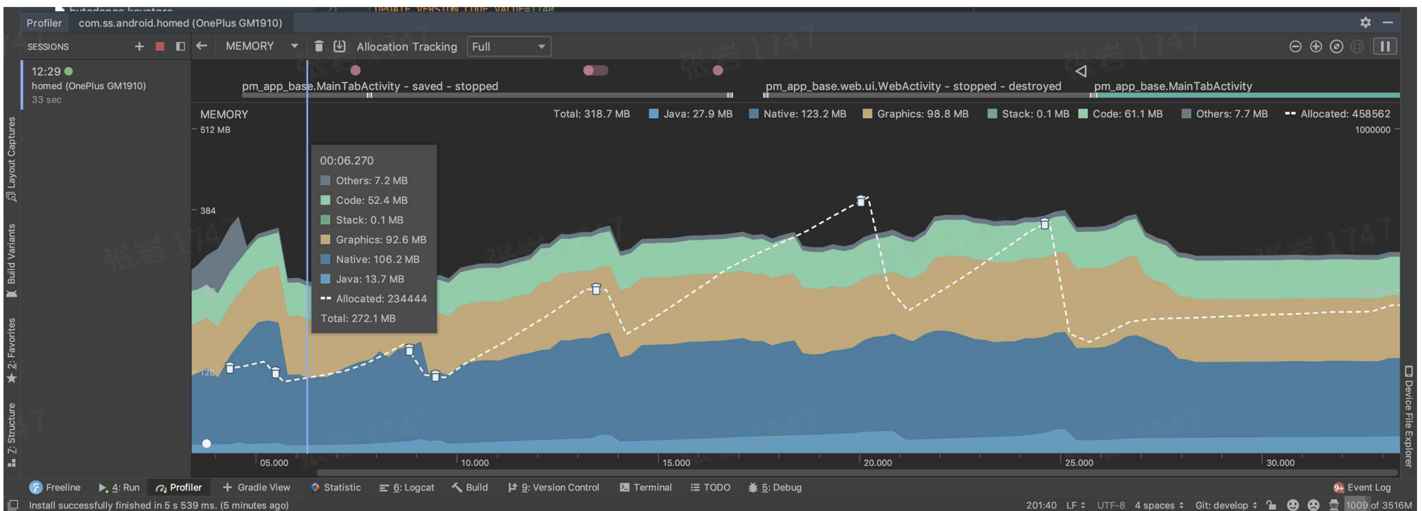


1. 强制GC
2. 捕获堆栈
3. 用于指定分析器多久捕获一次内存分配的下拉菜单（Full，Sampled，Off）
    a. Full：全量采样，严重影响App执行
    b. Simpled：定期采样，也会影响App执行，住小帮这种量级的app，其实也很严重
    c. Off：关闭内存分配采样；还是会影响App执行（惊喜不）
4. 用于缩放时间轴的按钮
5. 用于跳转到实时内存数据的按钮
6. 事件时间轴，显示活动状态、用户输入事件和屏幕旋转事件
7. 内存使用量时间轴
    a. 虚线，表示分配的对象数
    b. 一个堆叠图表，显示每个内存类别当前使用多少内存
    c. 垃圾桶，代表一次GC

启用高级分析



开启后住小帮的面板



这个功能默认是关闭了，我们也一般不开启这个高级分析，因为他会让程序的兼容性变差

比如住小帮开启后：

com.ss.android.homed E/StudioProfiler: JVMTI error: 103(JVMTI_ERROR_ILLEGAL_ARGUMENT)

内存顶部面板

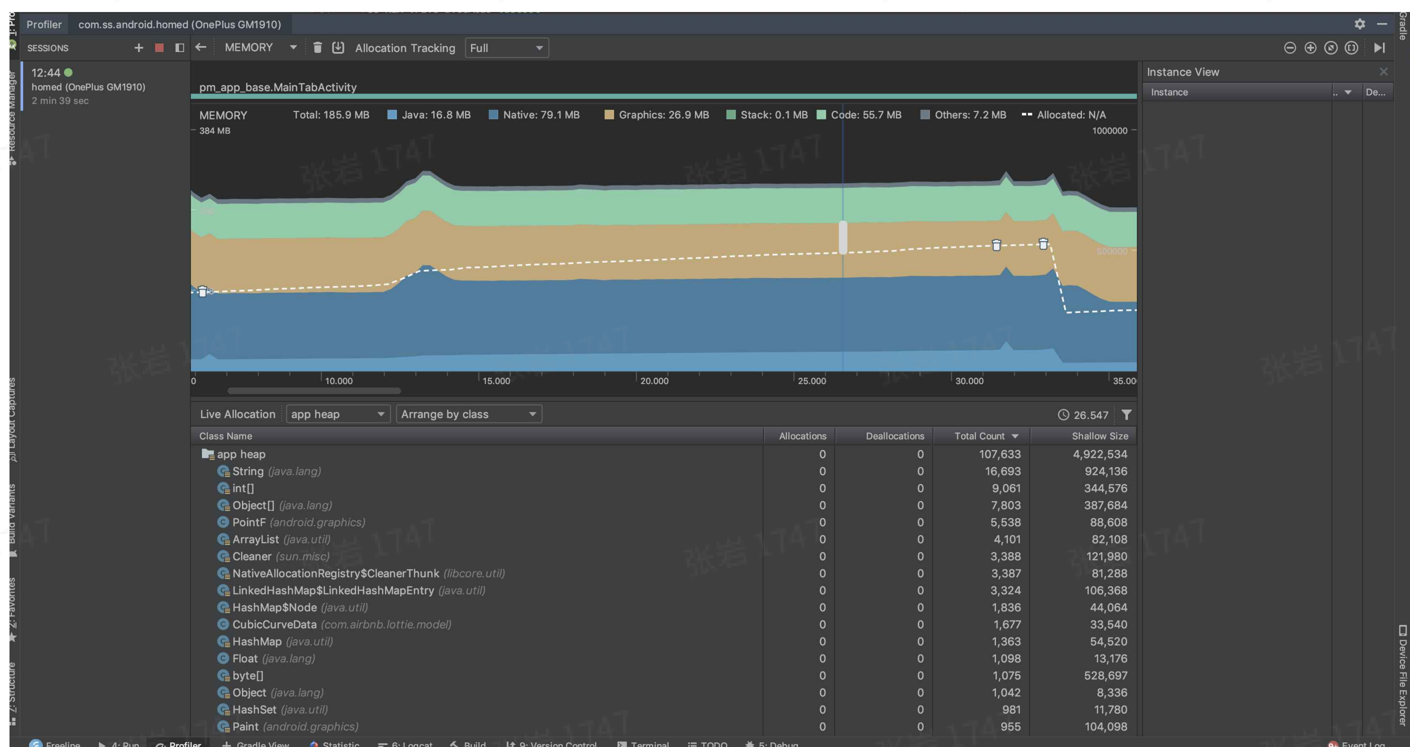
张岩 1747

Total: 126.21MB Java: 6.3MB Native: 71.48MB Graphics: 37.95MB Stack: 0.72MB Code: 7.54MB Others: 2.23MB Allocated: 106028

- Total：总内存
- java：java和kotlin分配对象占用的内存
- Native：C 或 C++ 代码分配的对象的内存
- 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 (cpu共享，非GPU)
- Stack：原生堆栈和 Java 堆栈使用的内存 （这个和线程使用情况相关性很多）
- Code：处理代码和资源（如 dex 字节码、经过优化或编译的 dex 代码、.so 库和字体）的内存，所以为啥我们不喜欢加字体库
- Others：系统不确定如何分类的内存 我们不关心
- Allocated： Java/Kotlin 对象数

查看内存分配

Android 7.1 以上的设备，单击堆叠图就能获取当前的内存分配情况



如果我们选择某个具体类型的对象

在面板中：

Instance View：当前内存中所有的该类型的对象实例

Allocation call stack：分配的调用栈（最后的Thread，就是它被分配的线程）

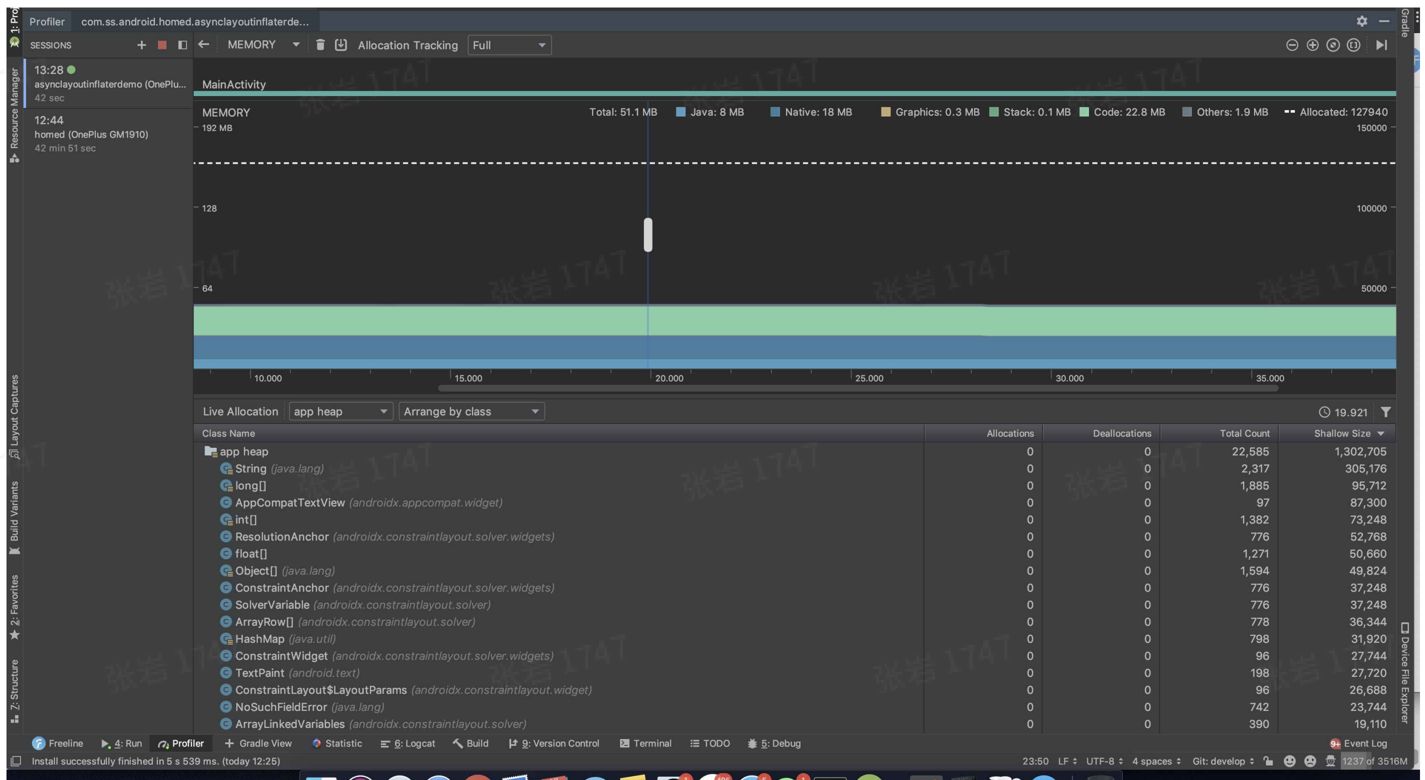
所有我们简单的对比两个切片之间的对象变化，可以初略的看出我们再对象上的开销；比如上图中：虚线在一个持续的上升，然后触发gc，对象回落，继续上升；

两个切面中：

Native和java 的内存在gc后产生的回落波动，在继续上升，这个与我们后台的很多统计、push请求、轮训都息息相关，这个可以视作为一次正常的GC流程；
但我们需要关注的有几点：

1. 这种看试正常的GC流程真的正常吗？
2. GC点产生的突起你又注意到了吗？

静态稳定，这才是我们想要的，但往往商业项目中各种需求，各种不合理的实现，过度的监控，都打破了我们要的，这也就是性能优化的终极命题，我们如何做让我们的应用趋近于这样的稳态



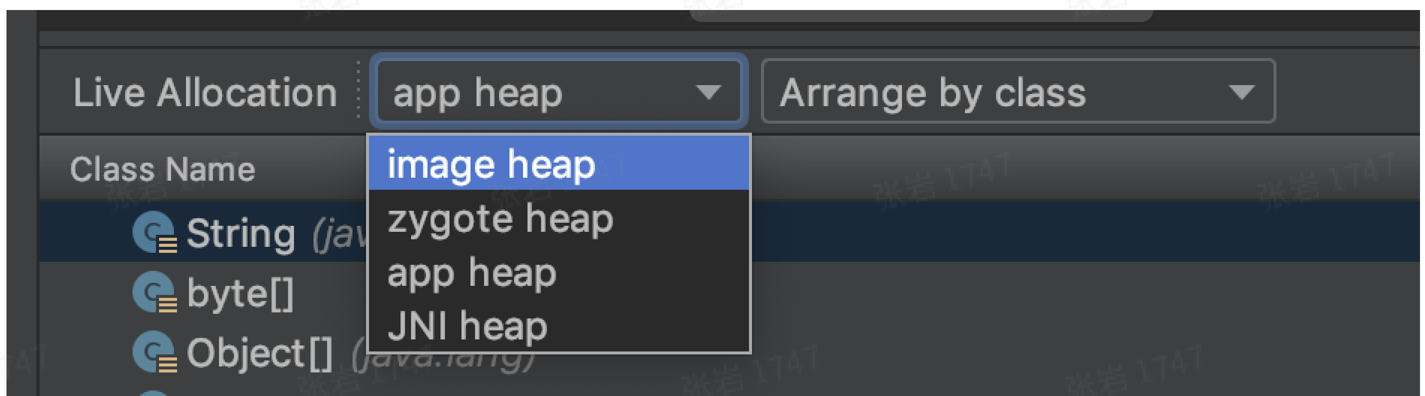
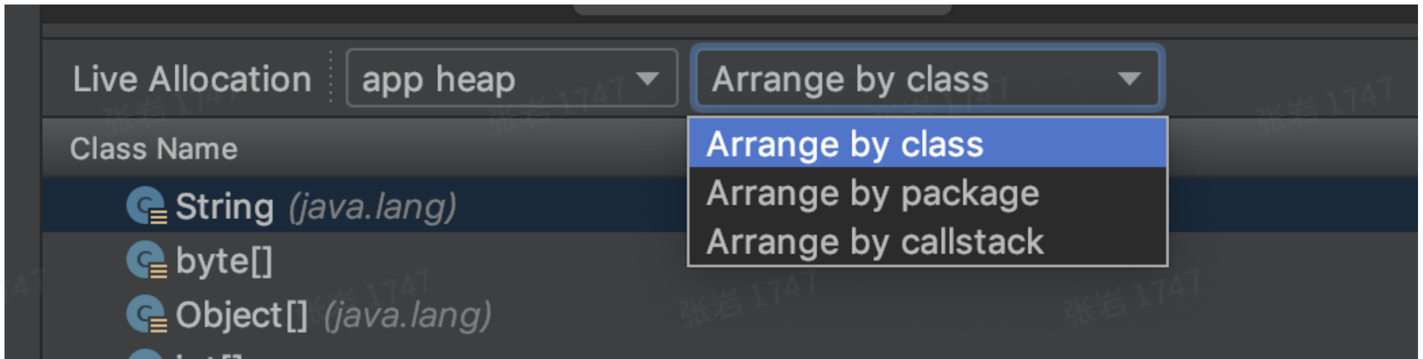


Image heap：系统启动映像，启动期间预加载的类，不会移动或消失，可以理解为常量（不关心）
Zygote heap： Android 系统中派生出来的（了解Android App启动的对Zygote进程应该不陌生）（奴不关心）
App heap：主堆内存，这是重点

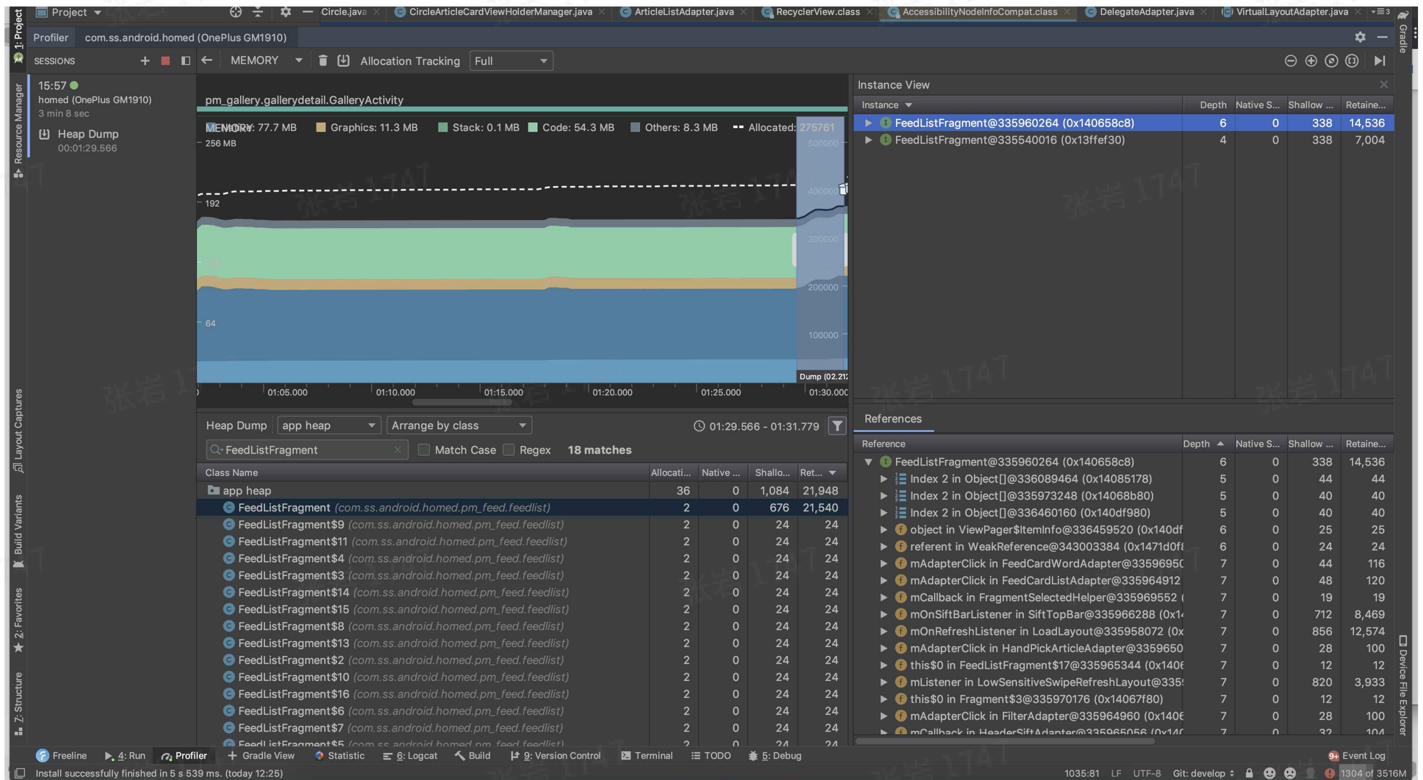
JNI heap：显示 Java 原生接口 (JNI) 引用被分配和释放到什么位置的堆，不关注（触发有大量的自有jni设计）



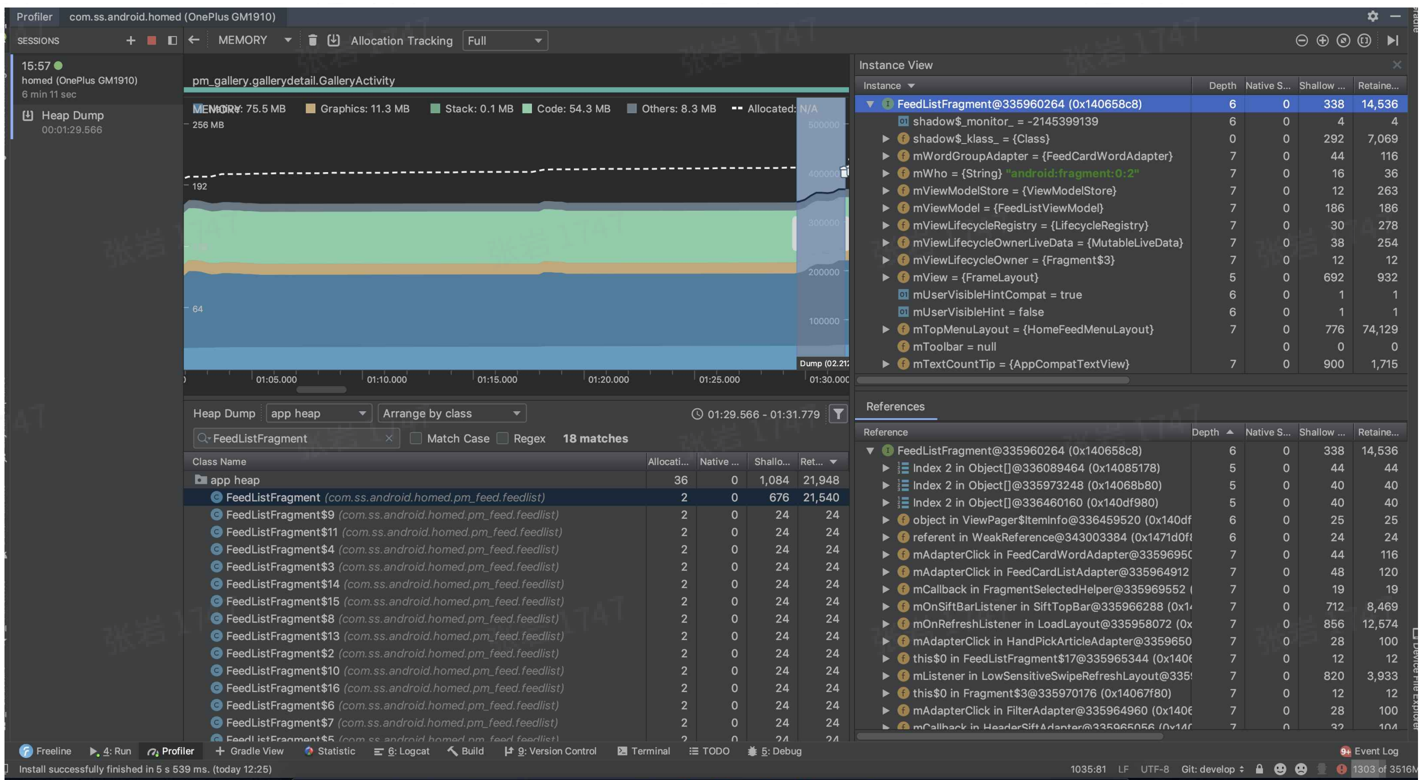
Arrange by class：根据class排列
Arrange by package：根据包名排列
Arrange by callstack：根据线程排列
支持搜索的

下面我们来看一下我们关心的

先看一个例子：FeedListFragment 为例



张岩 1747



Class Name	Allocations	Native Size	Shallow Size	Retained Size
app heap	36	0	1,084	21,948
FeedListFragment (com.ss.android.homed.pm_feed.feed	2	0	676	21,540
FeedListFragment$9 (com.ss.android.homed.pm_feed.fe	2	0	24	24

Instance View

00:01:31.576

	Depth	Native Size	Shallow Size	Retained Size
stFragment@335960264 (0x140658	6	0	338	14,536
shadow$_monitor_ = -2145399139	6	0	4	4
shadow$_klass_ = {Class}	0	0	292	7,069

先解释几个指标：

Instance View：当前堆内的实例个数（不是多个就是泄漏，这个需要结合整体的数据构成）

References：引用该实例的引用（例子里：就是持有BaseFragment的所有引用）

Allocalitons：堆中的分配数。也就是实例个数

Native Size：占用Native的内存大小

Shallow Size：该类型占用的Java总内存

Retained Size：为改类型保留的所有Java内存

Depth：从任意 GC 根到选定实例的最短跳数

我们就来分析一次：

张岩 1747

FeedListFragment 属于 HomeFeedFragment 没有问题。但为什么HomedFeedFragment会被

从正常执行逻辑上虽然是基本不会出现泄漏的，但是这样的代码在HomeFeedFragment destroy后就出现泄漏了，这些监听的持方都是单例的；

```
        FeedService feedService = FeedService.getInstance();
        if (feedService != null) {
            feedService.setLoginStatusListener(this);
            feedService.setShowPointCallback(this);
            feedService.setPopupPullListener(this);
            feedService.setTopTipPopupGuideListener(this);
        }
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        mViewPager.removeOnPageChangeListener(this);
        FeedService feedService = FeedService.getInstance();
        if (feedService != null) {
            feedService.removeLoginStatusListener( iLoginStatusListener: this);
            feedService.setHasDisplayPop(false);
            feedService.setPopupPullListener(null);
            feedService.setIsDestroy(true);
        }
        if (mHandler != null) {
            if (mShowTopPopupRunnable != null) {
                mHandler.removeCallbacks(mShowTopPopupRunnable);
            }
            if (mDismissTopPopupRunnable != null) {
                mHandler.removeCallbacks(mDismissTopPopupRunnable);
            }
        }
    }
```

关于这个工具

整体来看Memory Profiler是一个比较重型的内存风险工具，优势在于全面，系统，他基本可以做到所有我们能想到的内存分析工作；类似于MAT，Android Monitor等；适用于比较系统的排查风险代码，可以做到排查已经出现的内存风险，且能发现没有出现的但高风险的代码缺陷（比如我们的例子）；劣势也非常的明显，该工具暂时不能对高风险的泄漏作出自动的检查，排查起来需要借助堆叠图，或者叫明显的表现；

另外说一下

leakcanary：是一个较为轻量级的泄漏检查工具，其主要优势在于能自动的检测Activity，Fragment等组件的的泄漏情况；

StrictMode：严苛模式，也支持泄漏的检测，与leakcanary的原理基本一致

为什么我们讨厌GIF动图