性能优化

性能优化是一个APP不可或缺并需不断重复的工作，性能优化的深度是一个优秀APP的重要凭证。

不会有人喜欢用一个卡顿的app，卡顿的app会大大增加用户卸载的几率。

JAVA的 强引用、弱引用、软引用和虚引用

强引用(Strong Reference)

> 当内存空间不足时，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止

1.

普通变量引用

```
var str : String = "hahaha"
```

2.

内部类对外部类的引用

3.

静态变量的引用（记得不要用静态变量去引用一个activity）

软引用(Soft Reference)

> 如果内存空间不足了，就会回收这些对象的内存。

```
var str = "66"
var softReference = SoftReference(str)
```

弱引用(Weak Reference)

> 只要引起gc了，就会回收所指向的内存

```
var weakReference = WeakReference(Activity)
//用的时候：
weakReference.get()
```

上面的写法是如果实在需要引用activity或者context时可以采用的方法。

以上方法不会影响activity的正常回收：

```
String str = new String("hhh");
//建立弱引用
WeakReference<String> weakReference = new WeakReference<>(str);
System.out.println(weakReference.get());
//触发gc
System.gc();
//可以看到，当str的对“String("hhh")这个对象”的强引用还在时，gc不会回收String("hhh")这个对象，对应的弱引用还可以继续使用
```

```
System.out.println(weakReference.get());
//删除对“String("hhh")这个对象”的引用
str = null;
//触发gc
System.gc();
//可见当String的强引用删除后，只剩一个弱引用了（weakReference），jvm不顾弱引用的感受就随便地把String释放了
System.out.println(weakReference.get());
//运行效果：
//hh
//hh
//null
```

同理，activity的回收也像这个String一样能被正常回收

虚引用(PhantomReference)

> 相当于没有引用，任何时候都有可能回收
>
> 监听gc回收的时候可以用

但是kotlin貌似放弃了这样的引用，毕竟kotlin的变量大部分都不能为空，那gc还咋回收。

讲这个的目的主要是想讲清楚Handler的内存泄漏的问题。

为什么Handler会导致内存泄漏

> handler发送的消息在当前handler的消息队列中，如果此时activity finish掉了，那么消息队列的消息依旧会由handler进行处理，若此时handler声明为内部类（非静态内部类），我们知道内部类天然持有外部类的实例引用，这样在GC垃圾回收机制进行回收时发现这个Activity居然还有其他引用存在，因而就不会去回收这个Activity，进而导致activity泄露。

如何写一个安全的Handler（不会内存泄漏）

> 在此之前先明确几个前提：
>
> 1. 【非静态内部类】会持有外部类的【强引用】
> 2. 【静态内部类】不会持有外部类的引用
> 3. kotlin中的内部类默认为静态类（与java不同），如果需要使用外部类的东西，需要传进来才能用

```
class MainActivity : AppCompatActivity() {
    //首先它是一个静态的内部类，没有对外部的引用
    //其次，对mainActivity的引用是弱引用，不会影响activity的正常回收
    //再尔，?可以使得取不到activity时不干任何事情，不会抛异常
    class MyHandler(mainActivity: MainActivity) : Handler(){
        private val weakReference: WeakReference<MainActivity> =
WeakReference(mainActivity)
        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            when(msg.what){
                0 -> weakReference.get()?.tv?.text = msg.obj.toString()
            }
        }
    }
    lateinit var myHandler : MyHandler
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        myHandler = MyHandler(this)

        tv.setOnClickListener{
            Thread(Runnable {
                Thread.sleep(1000)
                val msg = Message()
                msg.what = 0
                msg.obj = "UI刷新"
                myHandler.sendMessage(msg)
            }).start()
        }

    }

}
```

这种handler就是绝对安全的，可以随便传给其他的fragment，activity或者thread，不会影响原activity的回收。

还有一个办法防止activity内存泄漏就是每一次退出activity时在ondestroy里handler.removeMessage()一下，这样这个handler的message就会被清空（如果不是空的则会由handler执行完，这样就会有activity的引用），这样handler就直接被回收了，因为没引用了。

线程间的通信

在Android开发时我们难免会遇到许多异步操作，线程间的通信之类的问题。那么线程间的通信的原理是什么呢?

每一个线程都有一个Looper（可以理解为死循环）和一个MessageQueue，你只要执行了Looper.prepare()和Looper.loop()，就会开启这个死循环，这个死循环是在不断的判断消息队列是否为空，如果非空则用handlerMessage顺序执行收到的消息。

```
class MainActivity : AppCompatActivity() {
    lateinit var handler1 : Handler
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        Thread{
            Looper.prepare()
            handler1 = Handler{
                Log.d("HaHa",Thread.currentThread().name + " got the message : "
+ it.obj.toString())
                false
            }
            Looper.loop()
        }.start()

        Thread{
            for(i in 1..10){
                val msg = Message()
                msg.obj = i
```

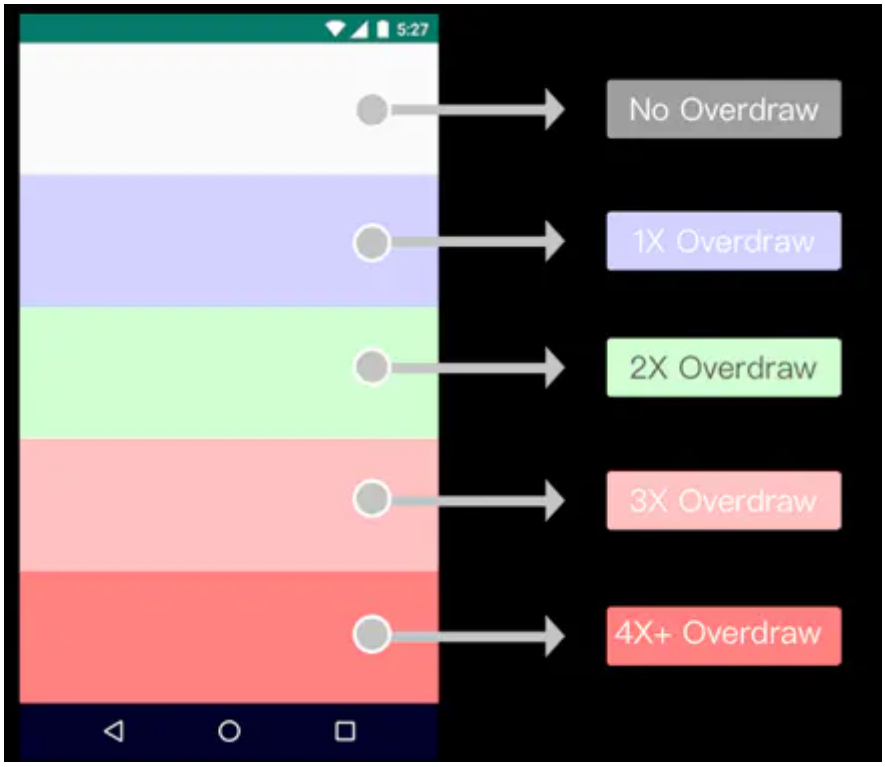
```
                Thread.sleep(1000)
                Log.d("HaHa",Thread.currentThread().name + " sendMessage")
                handler1.sendMessage(msg)
            }
        }.start()

    }
}
```

看起来这种线程不是顺序执行一件事情的，而是收到消息才执行对应的事情。

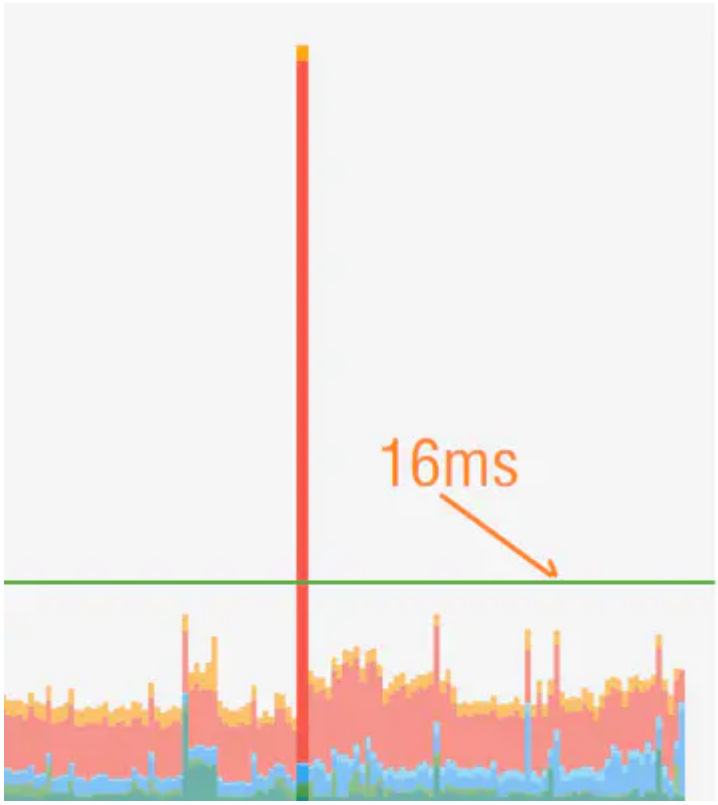
如何调试自己app的性能

1. **查看布局深度（开发者选项》调试GPU过度绘制）**



这样可以看到布局的深度（像素颜色表示该像素被绘制了多少次），尽量避免过深的布局出现

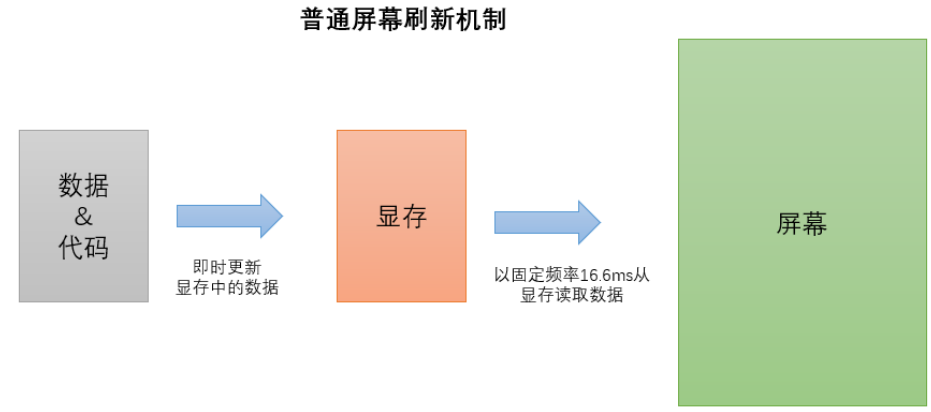
2. **查看每一帧的绘制耗时（开发者选项》GPU呈现模式分析）**



绿色线为16ms的基准线，尽量保持每一帧都在基准线下方。如果超过这个基准线就会发生丢帧

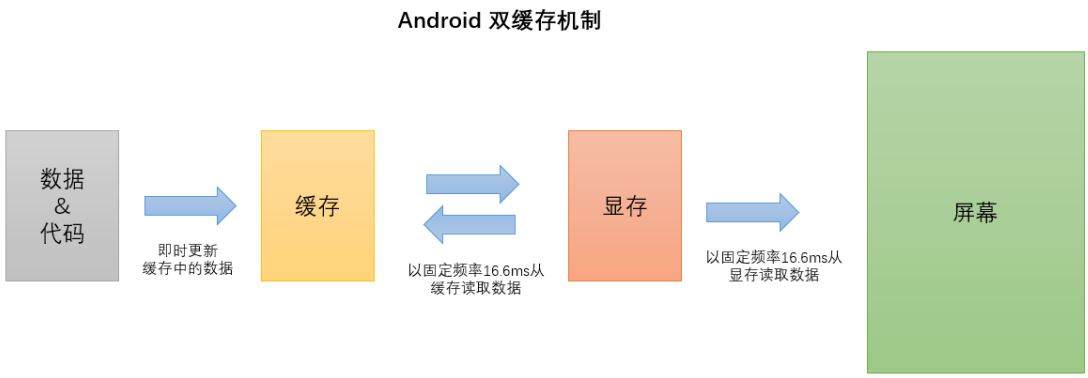
到这里有必要讲一下Android的屏幕刷新机制

1. 普通的屏幕刷新机制



这样会导致屏幕刷新时显存可能有一部分是上一帧的画面，有一部分是这一帧的画面，从而降低用户体验

2. Android双缓存屏幕刷新机制



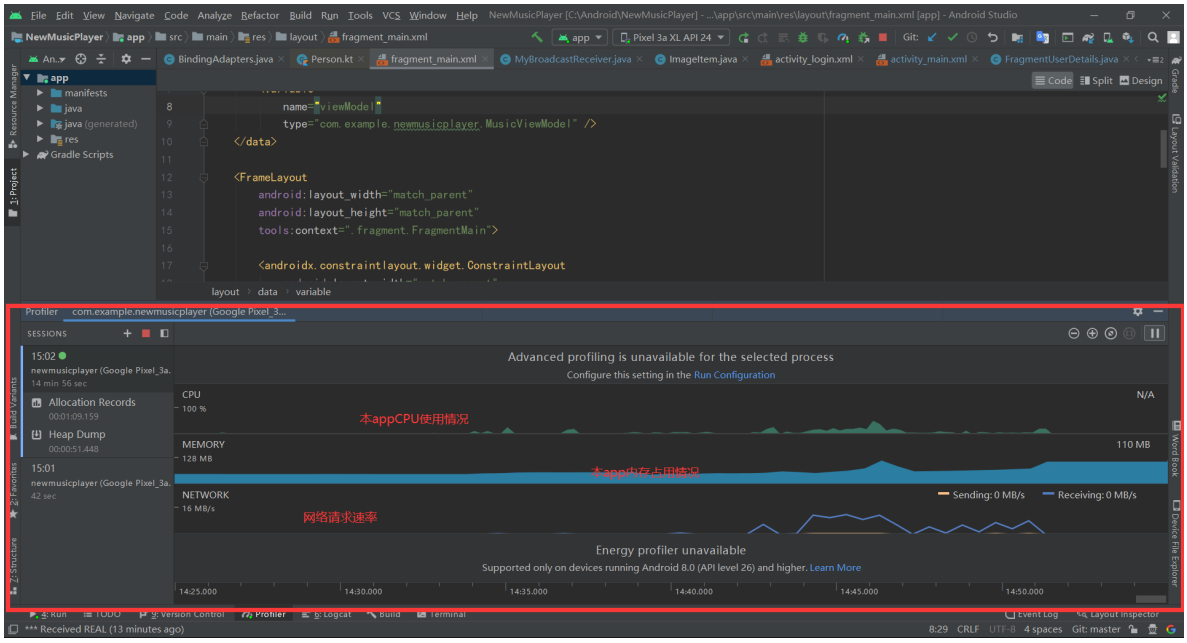
当缓存正在被写入数据时，缓存会被锁定，显存在到达刷新周期时，检测到缓存正在被写入绘制信息时，会放弃此次数据交换。

这样大家就知道丢帧的原理了吧？所以避免view绘制太长时间。

在app中多使用懒加载一些布局（等到用户看到了，不得不显示了，才加载），这样子可以避免一瞬时加载过多东西而造成的卡顿现象。而布局在未初始化时也可以使用下面的这种占位图，这样的好处是能让用户预先熟悉应用布局，从而提升应用体验。

3. **使用Android studio自带的工具：Profiler**

> 官方使用文档： https://developer.android.google.cn/studio/profile/memory-profiler
>
> 要养成看官方文档的好习惯哦（虽然我还没养成 逃）

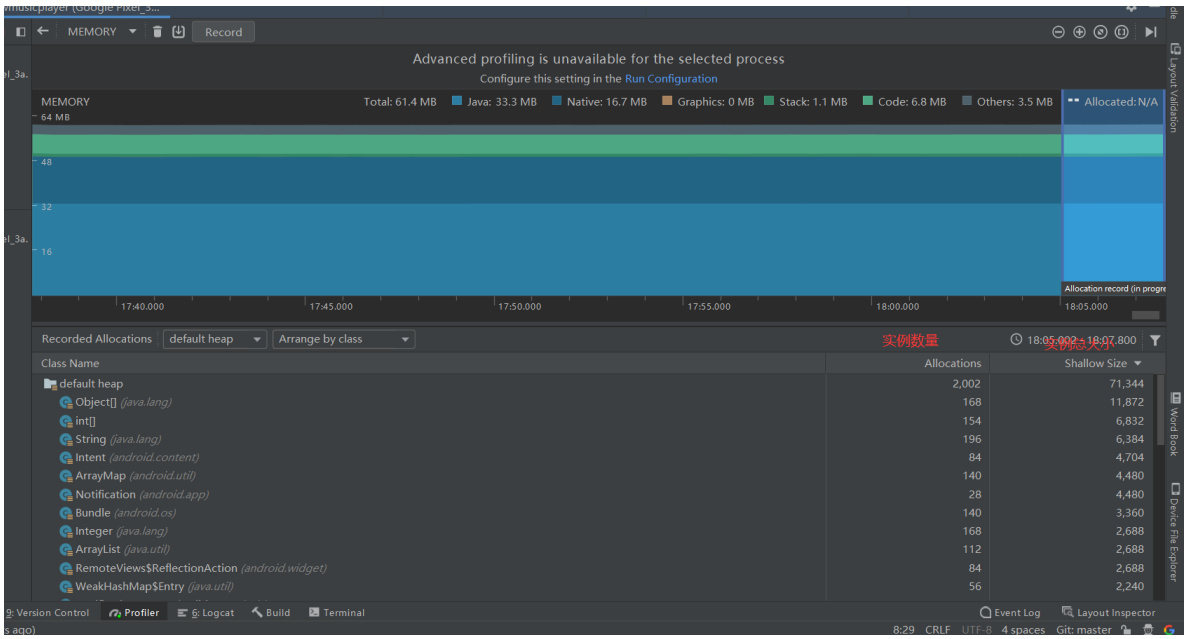


关于内存部分：

> Java：从 Java 或 Kotlin 代码分配的对象的内存。
>
> Native：从 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。（请注意，这是与 CPU 共享的内存，不是 GPU 专用内存。）
>
> Stack：您的应用中的原生堆栈和 Java 堆栈使用的内存。这通常与您的应用运行多少线程有关。
>
> Code：您的应用用于处理代码和资源（如 dex 字节码、经过优化或编译的 dex 代码、.so 库和字体）的内存。
>
> Others：您的应用使用的系统不确定如何分类的内存。
>
> Allocated：您的应用分配的 Java/Kotlin 对象数。此数字没有计入 C 或 C++ 中分配的对象。

内存详细：

> Allocations：堆中的实例数。
>
> Shallow Size：此堆中所有实例的总大小（以字节为单位）。其实算是比较真实的java堆内存
>
> Retained Size：为此类的所有实例而保留的内存总大小（以字节为单位）。这个解释并不准确，因为
>
> Retained Size会有大量的重复统计
>
> native size：8.0之后的手机会显示，主要反应Bitmap所使用的像素内存（8.0之后，转移到了native）


MEMORY
Record
Advanced profiling is unavailable for the selected process
Configure this setting in the Run Configuration
MEMORY
Total: 61.4 MB
Java: 33.3 MB
Native: 16.7 MB
Graphics: 0 MB
Stack: 1.1 MB
Code: 6.8 MB
Others: 3.5 MB
Allocated: N/A
64 MB
48
32
16
17:40.000
17:45.000
17:50.000
17:55.000
18:00.000
18:05.000
Allocation record (in progre
Recorded Allocations
default heap
Arrange by class
实例数量
实例占大小
Class Name
Allocations
Shallow Size
default heap 2,002 71,344
Object[] (java.lang) 168 11,872
int[] 154 6,832
String (java.lang) 196 6,384
Intent (android.content) 84 4,704
ArrayMap (android.util) 140 4,480
Notification (android.app) 28 4,480
Bundle (android.os) 140 3,360
Integer (java.lang) 168 2,688
ArrayList (java.util) 112 2,688
RemoteViews$ReflectionAction (android.widget) 84 2,688
WeakHashMap$Entry (java.util) 56 2,240
9: Version Control
Profiler
6: Logcat
Build
Terminal
Event Log
Layout Inspector
8:29
CRLF
UTF-8
4 spaces
Git: master