15.内存泄漏&内存溢出

15.1.什么是OOM & 什么是内存泄漏以及原因

内存泄漏

什么是

没有用的对象资源任与GC-Root保持可达路径，导致系统无法进行回收。

原因

1 非静态内部类默示持有外部类的引用，如非静态handler持有activity的引用

2.接收器、监听器注册没取消造成的内存泄漏，如广播，eventsbus

3.Activity 的 Context 造成的泄漏，可以使用 ApplicationContext

4.单例中的static成员间接或直接持有了activity的引用

5.资源对象没关闭造成的内存泄漏（如： Cursor、File等）

6.全局集合类强引用没清理造成的内存泄漏（特别是 static 修饰的集合）

内存溢出

根据java的内存模型会出现内存溢出的内存有堆内存、 方法区内存、虚拟机栈内存、native方法区内存， 一般说的OOM基本都是针对堆内存；

1、对于堆内存溢出主的根本原因有两种

（1）app进程内存达到上限

（2）手机可用内存不足，这种情况并不是我们app消耗了很多内存， 而是整个手机内存不足

2、对于app内存达到上限只有两种情况

（1）申请内存的速度超出gc释放内存的速度.往内存中加载超大文件，加载的文件或者图片过大造成或者循环创建大量对象

（2）内存出现泄漏，gc无法回收泄漏的内存，导致可用内存越来越少

15.2.Thread是如何造成内存泄露的，如何解决？

垃圾回收器不会回收GC Roots以及那些被它们间接引用的对象

非静态内部类会持有外部类的引用。Thread 会长久地持有 Activity 的引用，使得系统无法回收 Activity 和它所关联的资源和视图。

使用静态内部类/匿名类，不要使用非静态内部类/匿名类。

该养成为thread设置退出逻辑条件的习惯。

15.3. Handler导致的内存泄露的原因以及如何解决

原因:1.Java中非静态内部类和匿名内部类都会隐式持有当前类的外部引用

2.我们在Activity中使用非静态内部类初始化了一个Handler, 此Handler就会持有当前Activity的引用。

3.我们想要一个对象被回收，那么前提它不被任何其它对象持有引用， 所以当我们Activity页面关闭之后,存在引用关系： "未被处理 / 正处理的消息 -> Handler实例 -> 外部类", 如果在Handler消息队列 还有未处理的消息 / 正在处理消息时 导致Activity不会被回收，从而造成内存泄漏

解决方案: 1.将Handler的子类设置成 静态内部类,使用WeakReference弱引用持有Activity实例

2.当外部类结束生命周期时，清空Handler内消息队列

15.4.如何加载Bitmap防止内存溢出

1.对图片进行内存压缩；

2.高分辨率的图片放入对应文件夹；

3.内存复用

4.及时回收

15.5.MVP中如何处理Presenter层以防止内存泄漏的

首先 MVP 会出现内存泄漏是因为 Presenter 层持有 View 对象，一般我们会把 Activity 做为 View 传递到 Presenter，Presenter 持有 View对象，Activity 退出了但是没有回收出现内存泄漏。

解决办法： 1.Activity onDestroy() 方法中调用 Presenter 中的方法，把 View 置为 null

2.使用 Lifecycle

16.性能优化

https://www.jianshu.com/u/7f26e9b13731 https://github.com/liuyangbajin

https://www.jianshu.com/p/8bd39de7323f

16.1.内存优化

https://juejin.cn/post/6844903618642968590#heading-0

> 首先你要确保你的应用里没有存在内存泄漏，然后再去做其他的内存优化。内存泄漏和图片优化相对来说比较容易定位问题，且优化后效果也非常明显，性价比非常高。

事实上很多优化都是这样，比如减包大小的优化，也是要先分析出主要大头祸首，比如可能你的包里包含了一张3M大小的无用图片，如果你没找到这种祸首，可能你做了大量的工作去想办法减少无用代码等，最终可能只有几百K的收益。

1、内存泄漏和图片优化

1.1 避免内存泄漏

1.1.1 注意代码中常见内存泄漏场景

Handler内存泄漏

在Activity中使用非静态内部类初始化了一个Handler,此Handler就会持有当前Activity的引用。Activity页面关闭之后,存在引用关系。非静态内部类和匿名内部类都会隐式持有当前类的外部引用。

单例造成的内存泄漏

单例持有 Context 对象，如果 Activity 中调用 getInstance 方法并传入 this 时，singleTon 就持有了此 Activity 的引用，当退出 Activity 时，Activity 就无法回收，造成内存泄漏

资源性对象未关闭，注册对象未注销

例如Bitmap等资源未关闭会造成内存泄漏，例如BraodcastReceiver、EventBus未注销造成的内存泄漏，我们应该在Activity销毁时及时注销。

MVP中的内存泄漏

Presenter 层持有 View 对象，一般我们会把 Activity 做为 View 传递到 Presenter，Presenter 持有 View对象，Activity 退出了但是没有回收出现内存泄漏。

ViewPager+fragment内存泄露

List里一直有Fragment的引用，Fragment无法回收造成内存泄漏 在重写的PagerAdapter的getItem()方法中，return new yourFragment()解决此问题

在 MVP 的架构中，通常 Presenter 要同时持有 View 和 Model 的引用，如果在 Activity 退出的时候，Presenter 正在进行一个耗时操作，那么 Presenter 的生命周期会比 Activity 长，导致 Activity 无法回收，造成内存泄漏

避免在循环里创建对象，建立合适的缓存复用对象，避免在onDraw里创建对象

1.1.2 使用工具查找内存泄漏具体位置

LeakCanary

在Activity执行完onDestroy()之后，将它放入WeakReference中，然后将这个WeakReference类型的Activity对象与ReferenceQueque关联。这时再从ReferenceQueque中查看是否有没有该对象，如果没有，执行gc，再次查看，还是没有的话则判断发生内存泄露了。最后用HAHA这个开源库去分析dump之后的heap内存

Profiler

使用App后，Android Profiler中先触发GC，然后dump内存快照，之后点击按package分类，就可以迅速查看到你的App目前在内存中残留的class,点击class即可在右边查看到对应的实例以及引用对象。

1.2 图片优化

Bitmap 内存占用的计算

1.对图片进行内存压缩；

包括图片质量的压缩， RGB_565法，采样率压缩，Matrix比例压缩

2.高分辨率的图片放入对应文件夹；

不同dpi占用内存不一样,假设这张图片是ARGB_8888的，那这张图片占的内存就是 width * height * 4个字节或者 width * height * inTargetDensity /inDensity * 4

3.缓存

LruCache & DiskLruCache

4.及时回收

5.ARTHook非侵入式之图片检查

ARTHook 监控加载的图片是否过大的ImageView，可以在debug阶段发出警告，方便及早发现过大的图片。

1 使用Epic来进行Hook

2 DexposedBridge.hookAllConstructors(ImageView.class, new XC_MethodHook()

3 ImageHook中，图标宽高都大于view的2倍以上，则警告，当宽高度等于0时，说明ImageView还没有进行绘制，使用ViewTreeObserver进行大图检测的处理。

2、内存占用分析优化

2.1静态内存分析优化

确保打开每一个主要页面的主要功能，然后回到首页，进开发者选项去打开"不保留后台活动"。然后，将我们的app退到后台，GC，dump出内存快照。最后，我们就可以将对dump出的内存快照进行分析，看看有哪些地方是可以优化的，比如加载的图片、应用中全局的单例数据配置、静态内存与缓存、埋点数据、内存泄漏等等。

问题1： App首页的主图有两张(一张是保底图，一张是动态加载的图)，都比较大，而且动态加载的图回来后，保底图并没有及时被释放 优化：首先是对首页的主图进行颜色通道的改变以及压缩，可以大大降低这两张图所占的内存，然后在动态加载图回来后及时释放掉保底图 -5M

问题2： 首页底部的轮播背景图占用内存1.6M，且在图片加载回来后，背景图一直没有置空 优化：首先一般来说对背景图的质量并没有很高的要求，所以这张背景图是可以被成倍压缩的，并且在图片加载回来后，背景图要及时的释放掉。同时首页的多张轮播图以及其他图片都可以进行颜色模式的改变以及质量压缩。 -1.6M -4M

3时间选择库

4： SharePreference在内存里占用了700K的内存 优化：由于SP中的东西是会一次性加载到内存里并且保存为静态的，直到App进程结束才会被销毁，所以SP中千万别放大的对象，别图一时方便把对象序列化成json后保存到SP里，优化点就是把已经保存在SP中的一些较大的json字符串或者对象迁移到文件或者数据库缓存。 -400K

2.2 动态内存分析优化

利用Android Profiler实时观察进入每个页面后的内存变化情况，对产生的内存较大波峰做分析, dump出2个页面的内存快照文件，然后利用MAT的对比功能，找出每个页面相对于上个页面内存里主要增加了哪些东西，做针对性优化。

问题1：在内存里发现两个极少概率出现的empty view，占用了接近2M的内存 优化：用ViewStub对empty view做了懒加载，对于这些没有马上用到的资源要做延迟加载，还有很多大概率不会出现的View更加要做懒加载。 -2M

问题2：发现详情页的轮播大图的Viewpager用的Adapter是FragmentPagerAdapter，导致了所有的page都会被保存，当图片页数多的时候，往后翻内存会不断上升。 优化：这种页数多的ViewPager使用FragmentStatePagerAdapter来替代，它只会保留前后pager,在页数多的时候可以 节省大量内存。

内存抖动：

没创建一个对象就会分配一个内存,可用内存就少用了一块,当程序占用的内存达到一定的临界值

就会出发GC 内存频繁分配和回收导致内存不稳定

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

解决

减少不合理的对象创建

ondraw、getView中对象的创建尽量进行复用。

避免在循环中不断创建局部变量。

实时监控

LeakCanary

hprof 文件裁剪

LeakCanary 用 shark 组件来实现裁剪 hprof 文件功能，在 shark-cli 工具中，我们可以通过添加 strip-hprof 选项来裁剪 hprof 文件，它的实现思路是：通过将所有基本类型数组替换为空数组（大小不变）。

实时监控

LeakCanary 主要是为测试环境开发，它会在 Activity 或者 Fragment 的 destory 生命周期后，可以检测 Activity 和 Fragment 是否被回收，来判断它们是否存在泄露的情况。

Matrix-ResourceCanary

Matrix 主要也是在 hprof 文件裁剪 和 实时监控 这两方面做了一些优化。

hprof 文件裁剪

Matrix 的裁剪思路主要是将除了部分字符串和 Bitmap 以外实例对象中的 buffer 数组。 之所以保留 Bitmap 是因为 Matirx 有个检测重复 Bitmap 的功能，会对 Bitmap 的 buffer 数组做一次 MD5 操作来判断是否重复。

监控原理

Matrix 是基于 LeakCanary 上进行二次开发，所以基本是一致的，主要增加了一些误报的优化，比如：

多次检测到相同的可疑对象，才认定为泄露对象，参数可配置。

增加一个哨兵对象，用于判断是否有 GC 操作，因为调用 Runtime.getRuntime().gc() 只是建议虚拟机进行 GC 操作，并不一定会进行。

避免重复检测相同的对象

Matrix 虽然是基于 LeakCnary，但额外增加了一些配置选项，可以用于生产环境，比如 dump 模式，支持手动触发 dump，自动 dump，和不进行 dump，可以根据不同的环境，使用不同的模式。

除此之前，还有检测时间间隔等等。

16.2.启动优化

https://github.com/liuyangbajin/Performance----------

https://my.oschina.net/u/660720/blog/3188893

统计启动时间的方式

https://www.jianshu.com/p/59a2ca7df681

adb shell am start -S -R 10 -W com.ctg.and.mob.cuservice/com.ctg.and.mob.cuservice.mvp.ui.activity.MainActivity

TotalTime代表当前Activity启动时间，将多次TotalTime加起来求平均即可得到启动这个Activity的时间

缺点

应用的启动过程往往不只一个Activity，有可能是先进入一个启动页，然后再从启动页打开真正的首页。某些情况下还有可能中间经过更多的Activity，这个时候需要将多个Activity的时间加起来。

将多个Activity启动时间加起来并不完全等于用户感知的启动时间。例如在启动页可能是先等待某些初始化完成或者某些动画播放完毕后再进入首页。使用命令行统计的方式只是计算了Activity的启动以及初始化时间，并不能体现这种等待任务的时间。

解决

AMS会通过Zygote创建应用程序的进程后,执行Application构造方法 –> attachBaseContext() –> onCreate() –> Activity onCreate() –>–> onStart() –> onResume() –> 测量布局绘制显示在界面上–>onWindowFocusChanged()绘制完毕 在Application的attachBaseContext()方法中开始计算冷启动计时，然后在真正首页Activity的onWindowFocusChanged()中停止冷启动计时。

代码

设置一个Util类专门做计时,Application的attachBaseContext()开始,首页Activity的 onWindowFocusChanged()结束

具体代码方法耗时分析工具

1.手动记录

每个方法前记录System.currentTimeMillis(),之后System.currentTimeMillis()进行相减

2.AOP

Aspect统计 Application 中所有方法的耗时，AspectJ 其实就是一种 AOP 框架

2.1、连接点（JoinPoint）

JPoint 是一个程序的关键执行点，也是我们关注的重点。它就是指被拦截到的点（如方法、字段、构造器等等）。

2.2、切入点（PointCut）

对 JoinPoint 进行拦截的定义。PointCut 的目的就是提供一种方法使得开发者能够选择自己感兴趣的 JoinPoint。

2.3、通知（Advice）

1）、Before：PointCut 之前执行。 2）、After：PointCut 之后执行。 3）、Around：PointCut 之前、之后分别执行。

3.Android Profiler得到启动过程的CPU过程，看到线程的具体代码耗时

https://www.jianshu.com/p/e0d2b6347414

Run -> Edit Configurations-> Sample Java Methods，可以定位到Java代码大致定位到启动CPU耗时的原因

启动优化方案

https://juejin.cn/post/6844903919580086280

https://www.jianshu.com/p/bef74a4b6d5e

https://juejin.cn/post/6844903459951476744#heading-6

https://blog.csdn.net/qian520ao/article/details/81908505?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromBaidu-1.control&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromBaidu-1.control

https://zhuanlan.zhihu.com/p/158683369

1.启动闪屏主题设置

https://www.jianshu.com/p/436b91175826

白屏原因:

1.从zygote进程fork出一个新进程 2.创建Application类并初始化，主要是执行Application.onCreate()方法； 3.创建并初始化入口Activity类，并在窗口上绘制UI；

设置android:windowBackground属性

windowBackground属性可以配置任意drawable， 与你启动页一样的图片。做到视觉上的无缝过渡.

2.application初始化内存,异步改造。

子线程来分担主线程的任务，并减少运行时间

2.线程池和启动器

线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom，所以我们使用线程池的方式去执行异步。

使用FixedThreadPool：创建一个定长的线程池，可控制线程最大的并发数，超出的部分任务，会在队列中等待。

利用CPU设置线程池数量

// 获得当前CPU的核心数 private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();

// 设置线程池的核心线程数2-4之间,但是取决于CPU核数 private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1, 4));

// 创建线程池 ExecutorService executorService = Executors.newFixedThreadPool(CORE_POOL_SIZE);

2.线程池

通过线程池处理初始化任务的方式存在三个问题。

代码不够优雅

假如我们有 100 个初始化任务，那像上面这样的代码就要写 100 遍，提交 100 次任务。

无法限制在 onCreate 中完成

有的第三方库的初始化任务需要在 Application 的 onCreate 方法中执行完成，虽然可以用 CountDownLatch 实现等待，但是还是有点繁琐。

无法实现存在依赖关系

有的初始化任务之间存在依赖关系，比如极光推送需要设备 ID，而 initDeviceId() 这个方法也是一个初始化任务。

2.2.启动器

第一步是我们要对代码进行任务化，任务化是一个简称，比如把启动逻辑抽象成一个任务。

第二步是根据所有任务的依赖关系排序生成一个有向无环图，这个图是自动生成的，也就是对所有任务进行排序。

比如我们有个任务 A 和任务 B，任务 B 执行前需要任务 A 执行完，这样才能拿到特定的数据，比如上面提到的 initDeviceId。 假如任务 B 依赖于任务 A，这时候生成的有向无环图就是 ACB，A 和 C 可以提前执行，B 一定要排在 A 之后执行。

第三步是多线程根据排序后的优先级依次执行。

4.延迟执行任务

延迟启动器利用了 IdleHandler 实现主线程空闲时才执行任务，IdleHandler 是 Android 提供的一个类，IdleHandler 会在当前消息队列空闲时才执行任务，这样就不会影响用户的操作了。 假如现在 MessageQueue 中有两条消息，在这两条消息处理完成后，MessageQueue 会通知 IdleHandler 现在是空闲状态，然后 IdleHandler 就会开始处理它接收到的任务。

3.首页

首页部分接口合并为一，减少网络请求次数，降低频率；

首页布局优化

16.3.布局优化

布局加载流程

原理

https://jsonchao.github.io/2020/01/13/%E6%B7%B1%E5%85%A5%E6%8E%A2%E7%B4%A2Android%E5%B8%83%E5%B1%80%E4%BC%98%E5%8C%96%EF%BC%88%E4%B8%8A%EF%BC%89/

1、在setContentView方法中，会通过LayoutInflater的inflate方法去加载对应的布局到android.R.id.content中。

2、inflate方法中首先会调用Resources的getLayout方法去通过IO的方式去加载对应的Xml布局解析器到内存中。

调用链最终掉用的是 AssetManager的openXmlAssetNative,一个Native方法，通过IO流的方式进行。

方法返回XmlResourceParser,XmlResourceParser继承XmlPullParser, 根据提供的布局资源文件id可读取布局文件中view相关属性。

3、inflate，createViewFromTag来创建View的实例，在每次递归完成的时候将这个View添加到父布局中。

内部首先会判断mFactory2是否存在，存在就会使用mFactory2的onCreateView方法区创建视图，否则就会调用mFactory的onCreateView方法，接下来，如果此时的tag是一个Fragment，则会调用mPrivateFactory的onCreateView方法

createViewFromTag 将一些 控件变成兼容性控件 （例如将 TextView 变成 AppCompatTextView ）以便于向下兼容新版本中的效果 ，创建View，使用类加载器创建了对应的Class实例， 根据Class实例获取到了对应的构造器实例， 通过构造器实例constructor的newInstance方法创建了对应的View对象

https://segmentfault.com/a/1190000019101554

LayoutInflater.Factory的意义：

通过 LayoutInflater 创建 View 时候的一个回调，可以通过 LayoutInflater.Factory 来改造或定制创建 View 的过程。

AppCompatActivity 为什么 setFactory

向下兼容新版本中的效果。

Factory与Factory2的区别：

1、Factory2继承与Factory。 2、Factory2比Factory的onCreateView方法多一个parent的参数，即当前创建View的父View。

优化

性能瓶颈在于LayoutInflater.inflater过程，主要包括如下两点：

1 xmlPullParser IO操作，布局越复杂，IO耗时越长。

2 createView 反射，View越多，反射调用次数越多，耗时越长

AsyncLayoutInflater

https://www.jianshu.com/p/d61b513e6814

1将构造好的 InflateRequest 请求放入到队列ArrayBlockingQueue中。

2.异步线程死循环轮训这个队列，当队列中有数据，取出一个 InflateRequest。

3.通过获取 InflateRequest.LayoutInflater 真正地加载 resid 对应的布局文件，最终得到一个 View 对象，并赋值给 InflateRequest.view。

4.通过 UIHandler 将 InflateRequest 回调到主线程中 (ps:这时加载完成的 View 就传到了主线程了)

5.UIHanlder 处理消息，通过 InflateRequest#callback 将加载得到的 View 对象回调给调用层。

因为是异步加载，所以需要注意在布局加载过程中不能有依赖于主线程的操作,AsyncLayoutInflater仅仅只能通过侧面缓解的方式去缓解布局加载的卡顿.

X2C

X2C，它原理是采用APT（Annotation Processor Tool）+ JavaPoet技术来完成编译期间视图xml布局生成java代码，这样布局依然是用xml来写，编译期X2C会将xml转化为动态加载视图的java代码。

工具

局整体耗时监控 ：Aspectj做面向aop的非侵入性的布局整体耗时监控 。

单个视图创建耗时监控： Factory2、Factory本质上他俩就是创建View的一个hook，可以通过这个回调来监控单个View创建耗时情况。

布局绘制流程

工具

Layout Inspector 和 GPU rendering

原理

绘制的过程 CPU准备数据，通过Driver层把数据交给GPU渲染,Display负责消费显示内容

1.CPU主要负责Measure、Layout、Record、Execute的数据计算工作

2.GPU负责Rasterization（栅格化(向量图形的格式表示的图像转换成位图用于显示器)）、渲染,渲染好后放到buffer(图像缓冲区)里存起来.

3.Display（屏幕或显示器）屏幕会以一定的帧率刷新，每次刷新的时候，就会从缓存区将图像数据读取显示出来,如果缓存区没有新的数据，就一直用旧的数据，这样屏幕看起来就没有变

1.在 App 进程中创建PhoneWindow 后会创建ViewRoot。ViewRoot 的创建会创建一个 Surface壳子，请求WMS填充Surface，WMS copyFrom() 一个 NativeSurface。

2.响应客户端事件，创建Layer(FrameBuffer)与客户端的Surface建立连接。

3.copyFrom()的同时创建匿名共享内存SharedClient（每一个应用和SurfaceFlinger之间都会创建一个SharedClient）

4.当客户端 addView() 或者需要更新 View 时，App 进程的SharedBufferClient 写入数据到共享内存ShareClient中,SurfaceFlinger中的 SharedBufferServer 接收到通知会将 FrameBuffer 中的数据传输到屏幕上。

startActivity->ActivityThread.handleLaunchActivity->onCreate ->完成DecorView和Activity的创建->handleResumeActivity->onResume()->DecorView添加到WindowManager->ViewRootImpl.performTraversals()方法，测量（measure）,布局（layout）,绘制（draw）, 从DecorView自上而下遍历整个View树。

https://www.jianshu.com/p/779b5ad22316

优化

5.1 优化布局层级及其复杂度 measure、layout、draw这三个过程都包含的自顶向下的tree遍历耗时，它是由视图层级太深会造成耗时，另外也要避免类似RealtiveLayout嵌套造成的多次触发measure、layout的问题。最后onDraw在频繁刷新时可能多次被触发，因此onDraw不能做耗时操作，同时不能有内存抖动隐患等。

优化思路： 减少View树层级

布局尽量宽而浅，避免窄而深 ConstraintLayout 实现几乎完全扁平化布局，同时具备RelativeLayout和LinearLayout特性，在构建复杂布局时性能更高。

不嵌套使用RelativeLayout

不在嵌套LinearLayout中使用weight

merge标签使用：减少一个根ViewGroup层级

ViewStub 延迟化加载标签，当布局整体被inflater，ViewStub也会被解析但是其内存占用非常低，它在使用前是作为占位符存在，对ViewStub的inflater操作只能进行一次，也就是只能被替换1次。

5.2 避免过度绘制 一个像素最好只被绘制一次。

优化思路： 去掉多余的background，减少复杂shape的使用

避免层级叠加

自定义View使用clipRect屏蔽被遮盖View绘制

5.3 视图与数据绑定耗时

由于网络请求或者复杂数据处理逻辑耗时导致与视图绑定不及时。这里可以从优化数据处理的维度来解决

监控

布局加载监控

使用AspectJ做面向aop的非侵入性的监控。

针对Activity.setContentView监控简单示例：

@Aspect public class PerformanceAop { public static final String TAG = "aop"; @Around("execution(* android.app.Activity.setContentView(..))") public void getSetContentViewTime(ProceedingJoinPoint joinPoint) { Signature signature = joinPoint.getSignature(); String name = signature.toShortString(); long time = System.currentTimeMillis(); try { joinPoint.proceed(); } catch (Throwable throwable) { throwable.printStackTrace(); } Log.i(TAG, name + " cost " + (System.currentTimeMillis() - time)); } }

布局绘制监控

fpsviewer 利用Choreographer.FrameCallback来监控卡顿和Fps的计算，异步线程进行周期采样，当前的帧耗时超过自定义的阈值时，将帧进行分析保存，不影响正常流程的进行，待需要的时候进行展示，定位

Choreographer

1.只有当 App 注册监听下一个 Vsync 信号后才能接收到 Vsync 到来的回调。如果界面一直保持不变，那么 App 不会去接收每隔 16.6ms 一次的 Vsync 事件，但底层依旧会以这个频率来切换每一帧的画面 。即当界面不变时屏幕也会固定每 16.6ms 刷新，但 CPU/GPU 不走绘制流程。

2.当 View 请求刷新时，这个任务并不会马上开始，而是需要等到下一个 Vsync 信号到来时才开始 measure/layout/draw 流程；measure/layout/draw 流程运行完后，界面也不会立刻刷新，而会等到下一个 VSync 信号到来时才进行缓存交换和显示。

3.造成丢帧主要原因：一是遍历绘制 View 树以及计算屏幕数据超过了16.6ms 第2帧的CPU/GPU计算 没能在VSync信号到来前完成，屏幕平白无故地多显示了一次第1帧。 VSync信号还没绘制完毕

Choreographer，意为 舞蹈编导、编舞者。在这里就是指 对CPU/GPU绘制的指导—— 收到VSync信号 才开始绘制，保证绘制拥有完整的16.6ms，避免绘制的随机性。 VSyn

1.创建

mChoreographer，是在ViewRootImpl的构造方法内使用Choreographer.getInstance()创建

1.

//使用当前线程looper创建 mHandler

// 创建一个链表类型CallbackQueue的数组，大小为5，

创建了一个mHandler、VSync事件接收器mDisplayEventReceiver、任务链表数组mCallbackQueues

2.当有绘制请求时通过 postCallback 方法请求下一次 Vsync 信号

首先取对应类型的CallbackQueue添加任务，action就是mTraversalRunnable

ViewRootImpl.scheduleTraversals中

//添加同步屏障，屏蔽同步消息，保证VSync到来立即执行绘制 mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier(); mChoreographer.postCallback(Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null); mTraversalRunnable 移除同步屏障，开始三大绘制流程

postCallback

Android 4.1 之后系统默认开启 VSYNC，在 Choreographer 的构造方法会创建一个 FrameDisplayEventReceiver，scheduleVsyncLocked 方法将会通过它申请 VSYNC 信号。在 DisplayEventReceiver 的构造方法会通过 JNI 创建一个 IDisplayEventConnection 的 VSYNC 的监听者。

VSYNC信号的接受回调是onVsync()，使用mHandler发送消息到MessageQueue中 执行本次的doFrame(),这个message的Runnable就是队列中的所有任务。

3.申请VSync信号接收到后是走 doFrame()方 执行任务是 CallbackRecord的 run 方法：

Choregrapher 注册一个 FrameCallback 回调，那么系统在每一帧开始绘制的时候，会通过 FrameCallback#doFrame(...) 回调出来。我们一般画面的 fps 一般是 60fps，在这个回调中计算对应的 fps 即可。

如果绘制时间超过16.6ms，计算丢掉的帧数

https://juejin.cn/post/6863756420380196877#heading-17

https://ljd1996.github.io/2020/09/07/Android-Choreographer%E5%8E%9F%E7%90%86/

16.4.卡顿优化

https://www.jianshu.com/p/03dd61816051

https://juejin.cn/post/6844904066259091469#heading-31

https://blog.yorek.xyz/android/paid/master/stuck_1/#systrace

https://www.jianshu.com/p/75aa88d1b575

https://blog.csdn.net/JArchie520/article/details/106710663#1.1%E3%80%81%E5%8D%A1%E9%A1%BF%E9%97%AE%E9%A2%98%E4%BB%8B%E7%BB%8D

https://www.jianshu.com/p/03dd61816051

卡顿原因

FPS（帧率）：每秒显示帧数（Frames per Second）。表示图形处理器每秒钟能够更新的次数。高的帧率可以得到更流畅、更逼真的动画。一般来说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。提升至60fps则可以明显提升交互感和逼真感

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。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。

android中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。就会感觉到界面不流畅了（卡了一下）。丢帧导致卡顿产生。

卡顿优化就是监控和分析由于哪些因素的影响导致绘制渲染任务没有在一个vsync的时间内完成。尤其关注连续丢帧点。

卡顿产生的原因是错综复杂的，它涉及到代码、内存、绘制、IO、CPU等等

卡顿监控

1.BlockCanary: 动态检测消息执行耗时。 基于消息机制，向Looper中设置Printer，监控dispatcher到finish之间的操作，满足耗时阀值dump堆栈、设备信息，以通知形式弹出卡顿信息以供分析 非侵入式的性能监控组件，通知形式弹出卡顿信息 AndroidPerformanceMonitor implementation 'com.github.markzhai:blockcanary-android:1.5.0' https://github.com/markzhai/AndroidPerformanceMonitor

2.ANR

ANR-WatchDog

3.单点问题监控

界面打开页面从onCreate到onWindowFocusChanged耗时统计 Activity/Fragement 生命周期耗时统计 我们也需要去监控生命周期的一个耗时，如onCreate、onStart、onResume等 我们也需要去做生命周期间隔的耗时监 用AOP方式进行非侵入式打点通过这三个方面的监控纬度，我们就能够非常细粒度地去检测页面秒开各个方面的情况。自己代码：https://github.com/eleme/lancet 非one way的binder IPC调用耗时统计（hook BinderProxy.transact）,这类统计可以采用AOP方式进行非侵入式打点。 hook art：Epic hook

卡顿分析

卡顿工具 AndroidPerformanceMonitor ANR-WatchDog

原因 https://blog.csdn.net/axi295309066/article/details/72675365

解决 https://juejin.cn/post/6844904066259091469

16.5.网络优化

http://www.odev.top/2020/06/29/Top%E5%9B%A2%E9%98%9F%E5%A4%A7%E7%89%9B%E5%B8%A6%E4%BD%A0%E7%8E%A9%E8%BD%ACAndroid%E6%80%A7%E8%83%BD%E5%88%86%E6%9E%90%E4%B8%8E%E4%BC%98%E5%8C%96/#Java%E5%86%85%E5%AD%98%E5%9B%9E%E6%94%B6%E7%AE%97%E6%B3%95

https://juejin.cn/post/6861856444032466952#heading-0

1. 流量维度

流量消耗过多

3.1.1 自定义事件监听器

请求次数

4.1 数据缓存 4.2 数据压缩 4.3 图片压缩

2.质量维度

模拟数据