commit()是同步的， apply()是异步的

# LeakCanary

实现内存泄漏的主要判断逻辑是这样的。当我们观察的Activity或者Fragment销毁时，我们会使用一个弱引用去包装当前销毁的Activity或者Fragment,并且将它与本地的一个ReferenceQueue队列关联。我们知道如果GC触发了，系统会将当前的引用对象存入队列中。  
如果没有被回收，队列中则没有当前的引用对象。所以LeakCanary会去判断，ReferenceQueue是否有当前观察的Activity或者Fragment的引用对象，第一次判断如果不存在，就去手动触发一次GC，然后做第二次判断，如果还是不存在，则表明出现了内存泄漏。

# LruCache

# 进程与线程的区别

线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；

一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；

进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段，数据集，堆等)及一些进程级的资源(如打开文件和信号等)，某进程内的线程在其他进程不可见；

调度和切换：线程上下文切换比进程上下文切换要快得多。

**根本区别**：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

**资源开销**：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

**包含关系**：如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

**内存分配**：同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

**影响关系**：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

**执行过程**：每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

## 死锁概述

线程死锁是指两个或两个以上的线程互相持有对方所需要的资源，由于synchronized的特性，一个线程持有一个资源，或者说获得一个锁，在该线程释放这个锁之前，其它线程是获取不到这个锁的，而且会一直死等下去，因此这便造成了死锁。

## 死锁产生的条件

* 互斥条件：一个资源，或者说一个锁只能被一个线程所占用，当一个线程首先获取到这个锁之后，在该线程释放这个锁之前，其它线程均是无法获取到这个锁的。
* 占有且等待：一个线程已经获取到一个锁，再获取另一个锁的过程中，即使获取不到也不会释放已经获得的锁。
* 不可剥夺条件：任何一个线程都无法强制获取别的线程已经占有的锁
* 循环等待条件：线程A拿着线程B的锁，线程B拿着线程A的锁。。

## 死锁演示

Okhttp 用到的设计模式

1单例 2. 工厂 3.构造

4.享元

5观察者 6.责任链 7.策略

# App启动流程

## Zygote

Zygote进程负责其他进程的创建和启动，比如创建SystemServer进程，当需要启动一个新的android应用程序的时候，ActivityManagerService就会通过Socket通知Zygote进程为这个应用创建一个新的进程

# 乐观锁的实现方式-CAS（Compare and Swap）

乐观锁（ Optimistic Locking ）在上文已经说过了，其实就是一种思想。相对悲观锁而言，乐观锁假设认为数据一般情况下不会产生并发冲突，所以在数据进行提交更新的时候，才会正式对数据是否产生并发冲突进行检测，如果发现并发冲突了，则让返回用户错误的信息，让用户决定如何去做。

CAS操作包括三个操作数：需要读写的内存位置(V)、预期原值(A)、新值(B)。如果内存位置与预期原值的A相匹配，那么将内存位置的值更新为新值B。如果内存位置与预期原值的值不匹配，那么处理器不会做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS其实就是一个：我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。这其实和乐观锁的冲突检测+数据更新的原理是一样的。**乐观锁是一种思想，CAS只是这种思想的一种实现方式。**

# Volatile

## ****Java 内存模型中的可见性、原子性和有序性****

volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存

## **可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的**

在 Java 中 volatile、synchronized 和 final 实现可见性。

## **原子是世界上的最小单位，具有不可分割性。**

　在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

## **有序性**

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

## Volatile原理

olatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序，在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制

当一个变量定义为 volatile 之后，将具备两种特性：

　　1.保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：[Java内存模型](http://www.cnblogs.com/zhengbin/p/6407137.html" \t "_blank)）来完成。

　　2.禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个**内存屏障**（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。

volatile 性能：

　　volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

# Protobuf

Protobuf是一种平台无关、语言无关、可扩展且轻便高效的序列化数据结构的协议,可以用于网络通信和数据存储

 XML、JSON、ProtoBuf 都具有**数据结构化**和**数据序列化**的能力

 XML、JSON 更注重**数据结构化**，关注人类可读性和语义表达能力。ProtoBuf 更注重**数据序列化**，关注效率、空间、速度，人类可读性差，语义表达能力不足（为保证极致的效率，会舍弃一部分元信息）

 ProtoBuf 的应用场景更为明确，XML、JSON 的应用场景更为丰富。

## 编码格式

TLV 格式是我们比较熟悉的编码格式。

所谓的 TLV 即 Tag - Length - Value。Tag 作为该字段的唯一标识，Length 代表 Value 数据域的长度，最后的 Value 便是数据本身。

## Varints 编码

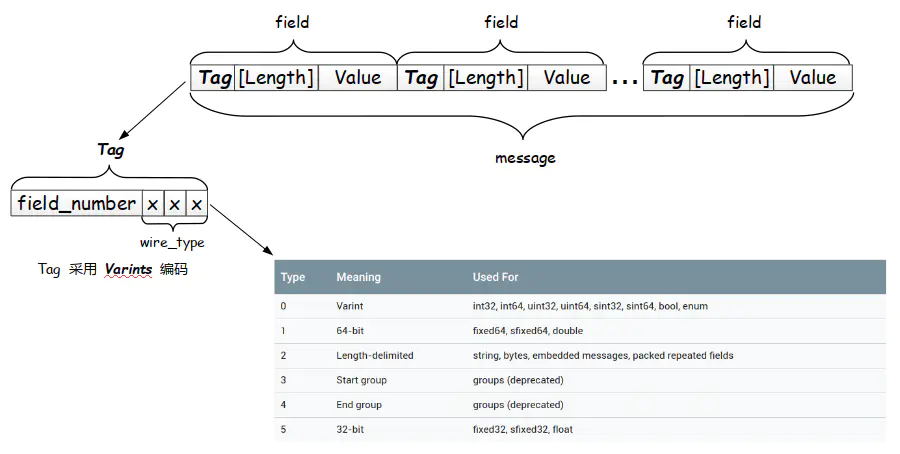
1. 在每个字节开头的 bit 设置了 **msb(most significant bit )**，标识是否需要继续读取下一个字节
2. 存储数字对应的二进制补码
3. 补码的低位排在前面

## ZigZag 编码

在上一节中我们提到了 Varints 编码对负数编码效率低的问题。

为解决这个问题，ProtoBuf 为我们提供了 sint32、sint64 两种类型，当你在使用这两种类型定义字段时，ProtoBuf 将使用 ZigZag 编码，而 ZigZag 编码将解决负数编码效率低的问题。

**ZigZag 编码：有符号整数映射到无符号整数，然后再使用 Varints 编码**

****

（MMKV**1，数据加密**。 在 Android 环境里，数据加密是非常必须的，SP实际上是把键值对放到本地文件中进行存储。如果要保证数据安全需要自己加密，MMKV 使用了 AES CFB-128 算法来加密/解密。  
**2，多进程共享**。系统自带的 SharedPreferences 对多进程的支持不好。现有基于 ContentProvider 封装的实现，虽然多进程是支持了，但是性能低下，经常导致 ANR。考虑到 mmap 共享内存本质上是多进程共享的，MMKV 在这个基础上，深入挖掘了 Android 系统的能力，提供了可能是业界最高效的多进程数据共享组件。  
**3，匿名内存**。 在多进程共享的基础上，考虑到某些敏感数据(例如密码)需要进程间共享，但是不方便落地存储到文件上，直接用 mmap 不合适。而Android 系统提供了 Ashmem 匿名共享内存的能力，它在进程退出后就会消失，不会落地到文件上，非常适合这个场景。MMKV 基于此也提供了 Ashmem(匿名共享内存) MMKV 的功能。  
**4，效率更高**。MMKV 使用protobuf进行序列化和反序列化，比起SP的xml存放方式，更加高效。  
**5，支持从 SP迁移**，如果你之前项目里面都是使用SP，现在想改为使用MMKV，只需几行代码即可将之前的SP实现迁移到MMKV。）

# java虚拟机和Dalvik虚拟机的区别：

|  |  |  |
| --- | --- | --- |
|  | java虚拟机 | Dalvik虚拟机 |
|  | java虚拟机基于**栈**。 基于栈的机器必须使用指令来载入和操作栈上数据，所需指令更多更多 | dalvik虚拟机是基于**寄存器**的 |
|  | java虚拟机运行的是java字节码。（java类会被编译成一个或多个字节码.class文件，打包到.jar文件中，java虚拟机从相应的.class文件和.jar文件中获取相应的字节码） | Dalvik运行的是自定义的.dex字节码格式。（java类被编译成.class文件后，会通过一个dx工具将所有的.class文件转换成一个.dex文件，然后dalvik虚拟机会从其中读取指令和数据） |
|  |  | 常量池已被修改为只使用32位的索引，以 简化解释器。dalvik的堆和栈的参数可以通过-Xms和-Xmx更改 |
|  |  | 一个应用，一个虚拟机实例，一个进程（所有android应用的线程都是对应一个linux线程，都运行在自己的沙盒中，不同的应用在不同的进程中运行。每个android dalvik应用程序都被赋予了一个独立的linux PID(app\_\*)） |

Dalvik和标准Java虚拟机（JVM）之间的首要差别之一，就是Dalvik基于寄存器，而JVM基于栈。

Dalvik和Java之间的另外一大区别就是运行环境——Dalvik经过优化，**允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 Dalvik应用作为一个独立的Linux进程执行。**

（1）虚拟机很小，使用的空间也小；

（2）Dalvik没有**JIT**编译器；

（3）常量池已被修改为只使用32位的索引，以简化解释器；

（4）它使用自己的字节码，而非Java字节码。

# Dalvik和art

Dalvik之所以要被ART替代包含下面几个原因：

* Dalvik是为32位设计的，不适用于64位CPU。
* 单纯的字节码解释加JIT编译的执行方式，性能要弱于本地机器码的执行。
* 无论是解释执行还是JIT编译都是单次运行过程中发生，每运行一次都可能需要重新做这些工作，这样做太浪费资源。
* 原先的垃圾回收机制不够好，会导致卡顿。
* **AOT编译**：Ahead-of-time（AOT）是相对于Just-in-time（JIT）而言的。JIT是在运行时进行字节码到本地机器码的编译，这也是为什么Java普遍被认为效率比C++差的原因。无论是解释器的解释还是运行过程中即时编译，都比C++编译出的本地机器码执行多了一个耗费时间的过程。而AOT就是向C++编译过程靠拢的一项技术：当APK在安装的时候，系统会通过一个名称为dex2oat的工具将APK中的dex文件编译成包含本地机器码的oat文件存放下来。这样做之后，在程序执行的时候，就可以直接使用已经编译好的机器码以加快效率。
* **垃圾回收的改进**：GC（Garbage Collection）是虚拟机非常重要的一个特性，因为它的实现好坏会影响所有在虚拟机上运行的应用。GC实现得不好可能会导致画面跳跃，掉帧，UI响应过慢等问题。ART的垃圾回收机制相较于Dalvik虚拟机有如下改进：
  + 将GC的停顿由2次改成1次
  + 在仅有一次的GC停顿中进行并行处理
  + 在特殊场景下，对于近期创建的具有较短生命的对象消耗更少的时间进行垃圾回收
  + 改进垃圾收集的工效，更频繁的执行并行垃圾收集
  + 对于后台进程的内存在垃圾回收过程进行压缩以解决碎片化的问题

# 常见内存泄露

## 1. 非静态内部类、匿名内部类

非静态内部类、匿名内部类 都会持有外部类的一个引用，如果有一个静态变量引用了非静态内部类或者匿名内部类，导致非静态内部类或者匿名内部类的生命周期比外部类（Activity）长，就会导致外部类在该被回收的时候，无法被回收掉，引起内存泄露, 除非外部类被卸载（JVM自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载，除非使用自定义的类加载器，感兴趣的同学可以研究一下）。

解决办法：   
  将非静态内部类、匿名内部类 改成静态内部类，或者直接抽离成一个外部类。

## 2.静态的View

有时，当一个Activity经常启动，但是对应的View读取非常耗时，我们可以通过静态View变量来保持对该Activity的rootView引用。这样就可以不用每次启动Activity都去读取并渲染View了。这确实是一个提高Activity启动速度的好方法！但是要注意，一旦View attach到我们的Window上，就会持有一个Context(即Activity)的引用。而我们的View有事一个静态变量，所以导致Activity不被回收。

解决办法：   
  在使用静态View时，需要确保在资源回收时，将静态View detach掉

## 3. Handler

  我们知道，主线程的Looper对象不断从消息队列中取出消息，然后再交给Handler处理。如果在Activity中定义Handler对象，那么Handler肯定是持有Activty的引用。而每个Message对象是持有Handler的引用的（Message对象的target属性持有Handler引用），从而导致Message间接引用到了Activity。如果在Activty destroy之后，消息队列中还有Message对象，Activty是不会被回收的。当然了，如果消息正在准备（处于延时入队期间）放入到消息队列中也是一样的。

解决办法：   
  将Handler放入单独的类或者将Handler放入到静态内部类中（静态内部类不会持有外部类的引用）。如果想要在handler内部去调用所在的外部类Activity，可以在handler内部使用弱引用的方式指向所在Activity，这样不会导致内存泄漏。   
  或者在onDestory时，调用相应的方法移除回调和删除消息。

## 4. 监听器（各种需要注册的Listener，Watcher等）

  当我们需要使用系统服务时，比如执行某些后台任务、为硬件访问提供接口等等系统服务。我们需要把自己注册到服务的监听器中。然而，这会让服务持有 activity 的引用，如果程序员忘记在 activity 销毁时取消注册，那就会导致 activity 泄漏了。   
  例如：EditText的一个addTextChangeListener，如果在回调方法里有耗时操作，可能会造成内存泄露

解决办法：   
  在onDestory时，取消注册，editText.removeTextChangedListener

## 5. 资源对象没关闭造成内存泄漏

  当我们打开资源时，一般都会使用缓存。比如读写文件资源、打开数据库资源、使用Bitmap资源等等。当我们不再使用时，应该关闭它们，使得缓存内存区域及时回收。虽然有些对象，如果我们不去关闭，它自己在finalize()函数中会自行关闭。但是这得等到GC回收时才关闭，这样会导致缓存驻留一段时间。如果我们频繁的打开资源，内存泄漏带来的影响就比较明显了。

解决办法：   
  及时关闭资源

## 6. 属性动画

  在使用ValueAnimator或者ObjectAnimator时，如果没有及时做cancel取消动画，就可能造成内存泄露

因为在cancel方法里，最后调用了endAnimation(); ，在endAnimation里，有个AnimationHandler的单例，会持有属性动画对象的引用，如下代码所示；

解决办法：   
  在在onDestory时，调用动画的cancel方法

## 7. RxJava

  在使用RxJava时，如果在发布了一个订阅后，由于没有及时取消，导致Activity/Fragment无法销毁，导致的内存泄露   
解决办法：   
  参考Uber出品的一个开源库AutoDispose的使用，可以参考下文：

## 8. WebView

  在android 5.1及以上版本的代码中，WebView可能会存在内存泄露，

解决办法：   
  在销毁webview前一定要onDetachedFromWindow，我们先将webview从它的父view中移除再调用destroy方法

# App启动流程

* ActivityManagerServices，简称AMS，服务端对象，负责系统中所有Activity的生命周期
* ActivityThread，App的真正入口。当开启App之后，会调用main()开始运行，开启消息循环队列，这就是传说中的UI线程或者叫主线程。与ActivityManagerServices配合，一起完成Activity的管理工作
* ApplicationThread，用来实现ActivityManagerService与ActivityThread之间的交互。在ActivityManagerService需要管理相关Application中的Activity的生命周期时，通过ApplicationThread的代理对象与ActivityThread通讯。
* ApplicationThreadProxy，是ApplicationThread在服务器端的代理，负责和客户端的ApplicationThread通讯。AMS就是通过该代理与ActivityThread进行通信的。
* Instrumentation，每一个应用程序只有一个Instrumentation对象，每个Activity内都有一个对该对象的引用。Instrumentation可以理解为应用进程的管家，ActivityThread要创建或暂停某个Activity时，都需要通过Instrumentation来进行具体的操作。
* ActivityStack，Activity在AMS的栈管理，用来记录已经启动的Activity的先后关系，状态信息等。通过ActivityStack决定是否需要启动新的进程。
* ActivityRecord，ActivityStack的管理对象，每个Activity在AMS对应一个ActivityRecord，来记录Activity的状态以及其他的管理信息。其实就是服务器端的Activity对象的映像。
* TaskRecord，AMS抽象出来的一个“任务”的概念，是记录ActivityRecord的栈，一个“Task”包含若干个ActivityRecord。AMS用TaskRecord确保Activity启动和退出的顺序。如果你清楚Activity的4种launchMode，那么对这个概念应该不陌生。

# 页面卡顿

## 二.哪些原因造成卡顿

 1.UI造成的卡顿

      (1)过于复杂的布局

      (2)过度绘制

      (3)动画

  2.主线程执行了耗时操作

  3.频繁的GC

## 三.怎么优化卡顿？

### 1.UI卡顿优化

#### (1)布局优化

           a.相同层级可实现的页面使用LineraLayout代替RelativeLayout，因为从源码中我们可以知道，

             LineraLayout在没有设置 weight属性的时候onMeasure()只执行一次，而RelativeLayout都要执行两次。

         b.复杂层级的布局可使用ConstraintLayout来减少层级,ConstraintLayout是一个非常强大的布局控件，很多相对复               杂的布局都可以使用ConstraintLayout一个层级就可以实现功能。

         c.使用include标签：

           nclude结合使用，可以减少布局层级，例如一个include的布局文件中

#### (2)过度绘制优化

    a.使用overdraw检测， 设置 -> 开发者选项 -> 调试GPU过度绘制 -> 显示GPU过度绘制，

      通过不同颜色来发现哪些地方过度绘制。

    b.移除不必要的background,比如我们整个页面的背景设置了白色，这样子控件如果再有是白色的就没有必要再设置背景色了。

#### （3）动画优化

    a.有些酷炫的动画往往对性能要求比较高，所以可以采用分级方式，判断手机类型，低端手机减少动画。

### 2.主线程耗时操作操作优化

   (1)主线程也叫UI线程主要的任务是处理用户交互、绘制界面、显示数据、消息处理等工作，如果耗时操作如请求网络数据、操作数据库、读取文件等就不能放在主线程来，可以开启子线程来操作。

 （2）使用线程池来代替单独创建子线程，因为频繁的创建和销毁线程很耗时,创建太多的子线程也会抢占主线程的CPU使用，从而导致卡顿

### 3.频繁的GC优化

   (1)需要频繁操作字符串时使用 StringBuilder 。

 （2）new对象的时候要注意，尽量不要在需要频繁执行的地方New对象。

# 线程池

## 1、什么是线程池

　线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务

一个线程池包括以下四个基本组成部分：  
1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；  
2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；  
3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；  
4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

### 2.常见线程池

①**newSingleThreadExecutor**  
单个线程的线程池，即线程池中每次只有一个线程工作，单线程串行执行任务  
②**newFixedThreadExecutor**(n)  
固定数量的线程池，没提交一个任务就是一个线程，直到达到线程池的最大数量，然后后面进入等待队列直到前面的任务完成才继续执行  
③**newCacheThreadExecutor**（推荐使用）  
可缓存线程池，当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程，当有任务来时，又智能的添加新线程来执行。  
④**newScheduleThreadExecutor**  
大小无限制的线程池，支持定时和周期性的执行线程

java提供的线程池更加强大，相信理解线程池的工作原理，看类库中的线程池就不会感到陌生了。

# serializable和parcelable区别

1. 序列化与反序列化的概念

从广义上讲，数据序列化就是将数据结构或者是对象转换成我们可以存储或者传输的数据格式的一个过程，在序列化的过程中，数据结构或者对象将其状态信息写入到临时或者持久性的存储区中，而在对应的反序列化过程中，则可以说是生成的数据被还原成数据结构或对象的过程。

序列化：得到的字节序列可以方便在网络上传输或者持久化——编码。

反序列化：主要从网络/磁盘读取读取字节序列然后转化为对象或者数据结构。——解码

## Serializable 接口的特点

1. 序列化类的属性没有实现 Serializable 那么在序列化就会报错
2. 在反序列化过程中，它的父类如果没有实现序列化接口，那么将需要提供无参构造函数来重新创建对象。
3. 一个实现 Serializable 接口的子类也是可以被序列化的
4. 静态成员变量是不能被序列化
5. transient 标识的对象成员变量不参与序列化
6. Serializable 在序列化和反序列化过程中大量使用了反射，因此其过程会产生的大量的内存碎片

## Externalizable 接口

Serializable 接口内部序列化是 JVM 自动实现的，如果我们想自定义序列化过程，就可以使用以上这个接口来实现，它内部提供两个接口方法

void writeExternal(ObjectOutput var1) throws IOException;

Void readExternal(ObjectInput var1) throws IOException, ClassNotFoundException;

## Java 的序列化步骤与数据结构分析

序列化算法一般会按步骤做如下事情：

* 将对象实例相关的类元数据输出。
* 递归地输出类的超类描述直到不再有超类。
* 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。
* 从上至下递归输出实例的数据

# View的绘制流程

View 绘制中主要流程分为measure，layout， draw 三个阶段。

measure ：根据父 view 传递的 MeasureSpec 进行计算大小。

layout ：根据 measure 子 View 所得到的布局大小和布局参数，将子View放在合适的位置上。

draw ：把 View 对象绘制到屏幕上。

在介绍发起绘制的入口之前，我们需要先了解Window，ViewRootImpl，DecorView之间的联系。

一个 Activity 包含一个Window，Window是一个抽象基类，是 Activity 和整个 View 系统交互的接口，只有一个子类实现类PhoneWindow，提供了一系列窗口的方法，比如设置背景，标题等。一个PhoneWindow 对应一个 DecorView 跟 一个 ViewRootImpl，DecorView 是ViewTree 里面的顶层布局，是继承于FrameLayout，包含两个子View，一个id=statusBarBackground 的 View 和 LineaLayout，LineaLayout 里面包含 title 跟 content，title就是平时用的TitleBar或者ActionBar，contenty也是 FrameLayout，activity通过 setContent（）加载布局的时候加载到这个View上。ViewRootImpl 就是建立 DecorView 和 Window 之间的联系。

## **measureSpeac**

大概意思是：MeasureSpec 封装了从父View 传递给到子View的布局需求。每个MeasureSpec代表宽度或高度的要求。每个MeasureSpec都包含了size（大小）和mode（模式）

MeasureSpec 一个32位二进制的整数型，前面2位代表的是mode，后面30位代表的是size。mode 主要分为3类，分别是

EXACTLY：父容器已经测量出子View的大小。对应是 View 的LayoutParams的match\_parent 或者精确数值。

AT\_MOST：父容器已经限制子view的大小，View 最终大小不可超过这个值。对应是 View 的LayoutParams的wrap\_content

UNSPECIFIED：父容器不对View有任何限制，要多大给多大，这种情况一般用于系统内部，表示一种测量的状态。(这种不怎么常用，下面分析也会直接忽略这种情况)

# Android中Window与DecorView

kotlin和java的区别

# Tinker热修复原理

Android中有两个主要的Classloader，PathClassLoader和DexClassLoader，它们都继承自BaseDexClassLoader，这两个类加载器的主要区别是：Android系统通过PathClassLoader来加载系统类和主dex中的类。而DexClassLoader则可用于加载指定路径的apk、jar或dex文件。上述两个类都是继承自BaseDexClassLoader