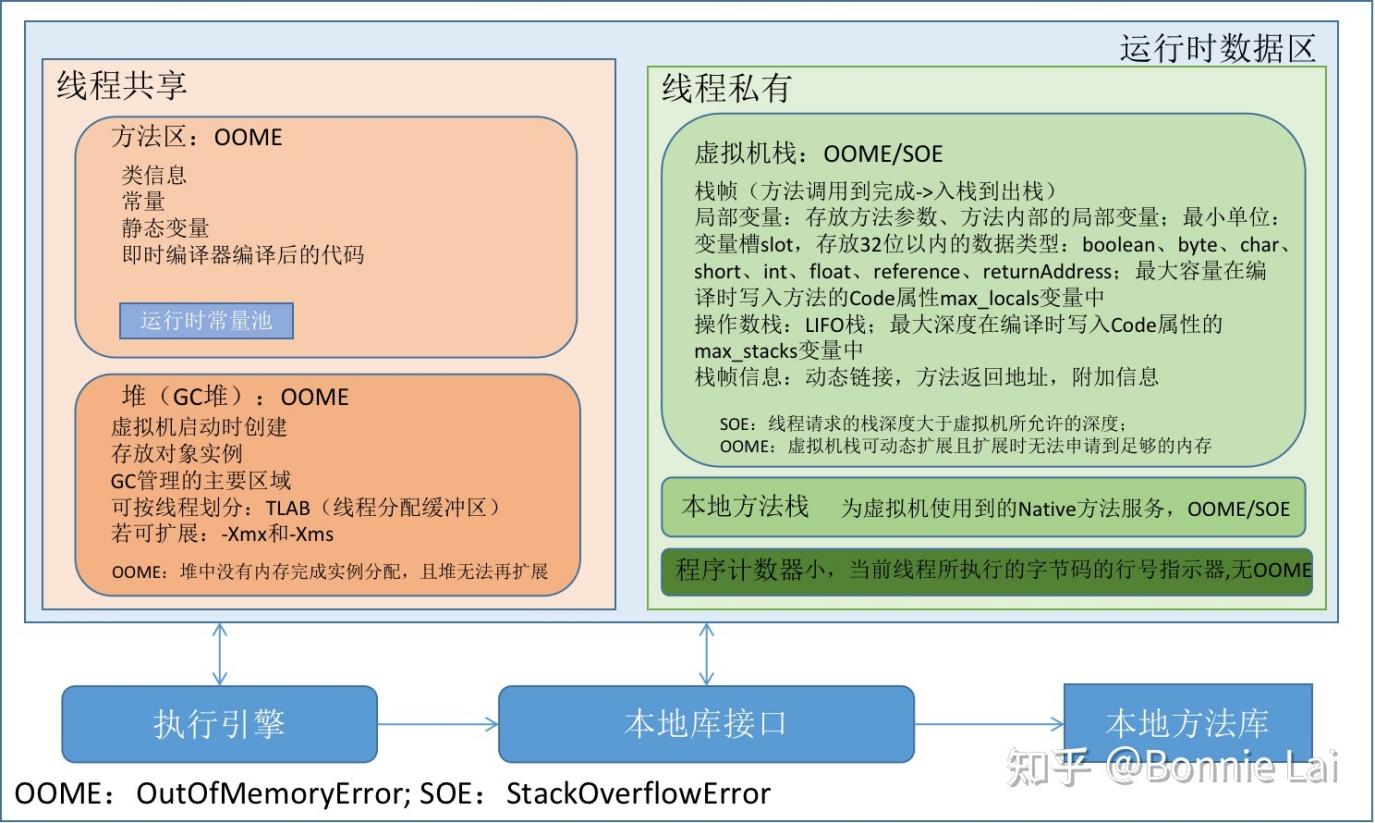
# 一、Java基础

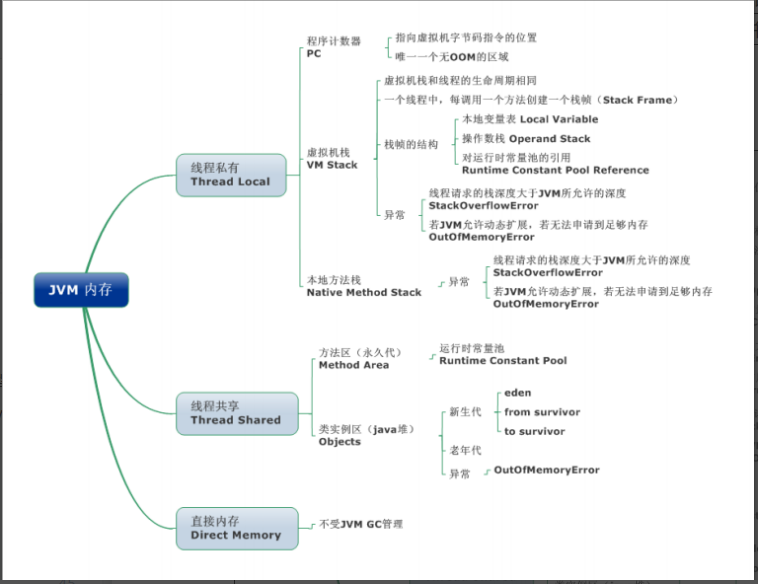
## 1.1虚拟机

运行Java程序的抽象计算机，它是一种计算机设备的规范，可以采用不同的方式进行实现。

JVM的内部体系结构分为三部分，分别是：类装载器（ClassLoader）子系统，运行时数据区，和执行引擎。



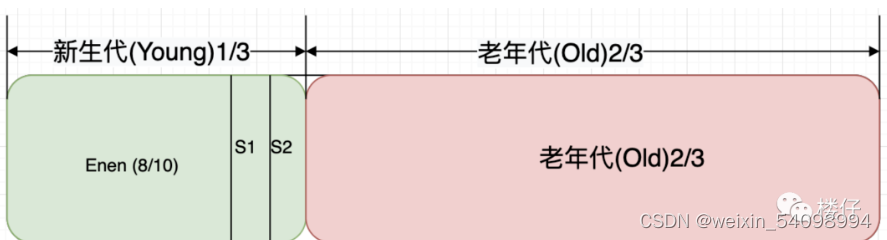
## 1.2 内存模型



### 1.2.1 Java堆

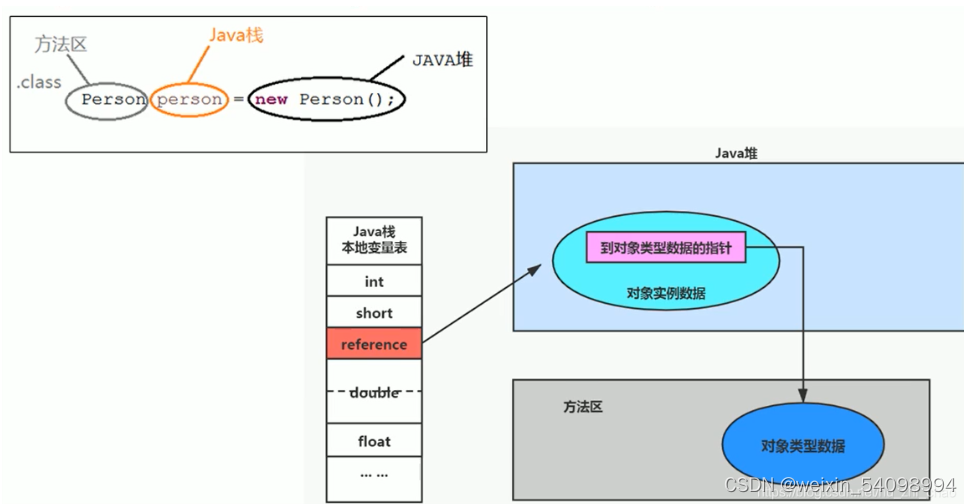
用于存储类的实例，GC主要进行的区域，线程共享，线程不安全，会发生OOM。

Java堆区可以划分为新生代和老年代，新生代又可以进一步划分为Eden区、FromSurvivor区、ToSurvivor区。新生代主要发生MinorGC，垃圾回收用复制清除算法；老年代主要发生FullGC（MajorGC），垃圾回收用标记清除或标记整理算法。



### 1.2.2 方法区

用于存储已被虚拟机加载的类信息、常量、静态变量、编译器编译后的代码等数据。线程共享区域，线程不安全。会发生OOM。



### 1.2.3 JVM栈

存放基本数据类型，以及对象的引用（reference类型，可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和returnAddress类型。

线程私有的，每一个线程都有独享一个虚拟机栈，它的生命周期与线程相同。(新建，就绪，运行，阻塞以及死亡)。会抛出OOM和SOE。

Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

### 1.2.4 本地方法栈

与Java栈类似，处理的是Native方法。会抛出OOM和SOE。

### 1.2.5 程序计数器

于存放下一条指令所在单元的地址的地方。不会OOM。

### 1.2.6 直接内存

来源于NIO，通过存在堆中的DirectByteBuffer操作Native内存，访问直接内存的速度会优于Java堆。即读写性能高。出于性能考虑，读写频繁的场合可能会考虑使用直接内存。

不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域。直接内存是在Java堆外的、直接向系统申请的内存区间。

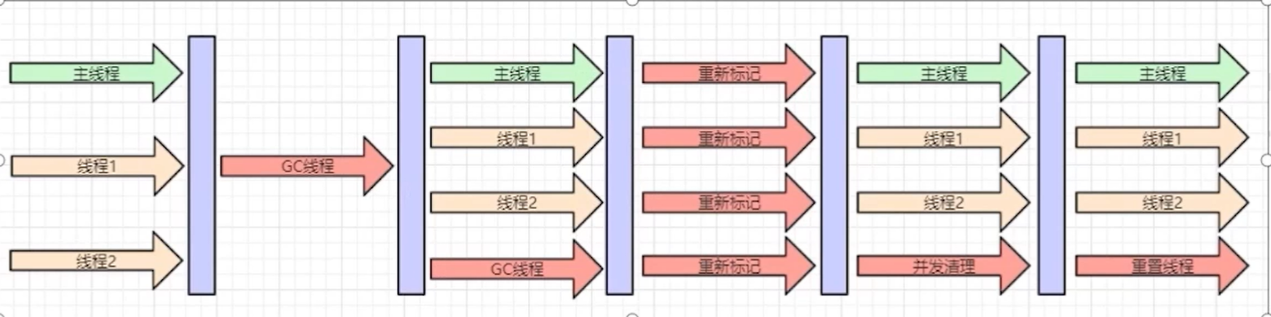
会发生OOM，不受GC管理。如果要回收，需要主动调用Unsafe对象的freeMemory方法，Unsafe是Java底层的类，一般不建议使用。直接内存释放是借助了java的虚引用机制，ByteBuffer的实现类内部，使用了Cleaner（虚引用）。

### 1.2.7 JMM

共享变量存储在主内存(Main Memory)中，每个线程都有一个私有的本地内存（Local Memory），本地内存保存了被该线程使用到的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。

导致了线程间的可见性缺失，可通过加锁或volatile关键字解决。

## 1.3 垃圾回收



CMS垃圾收集器的工作流程，先要挂起所有线程，防止继续产生垃圾。

### 1.3.1 判断可回收

引用计数算法：为对象添加一个引用计数器，当该对象被引用时，计数器+1，反之-1，当对象的计数器为0时表示该对象没有被引用。

可达性分析算法：通过一系列被称之为“GC Roots”的根节点开始，沿着引用链进行搜索，凡是在引用链上的对象都不会被回收。

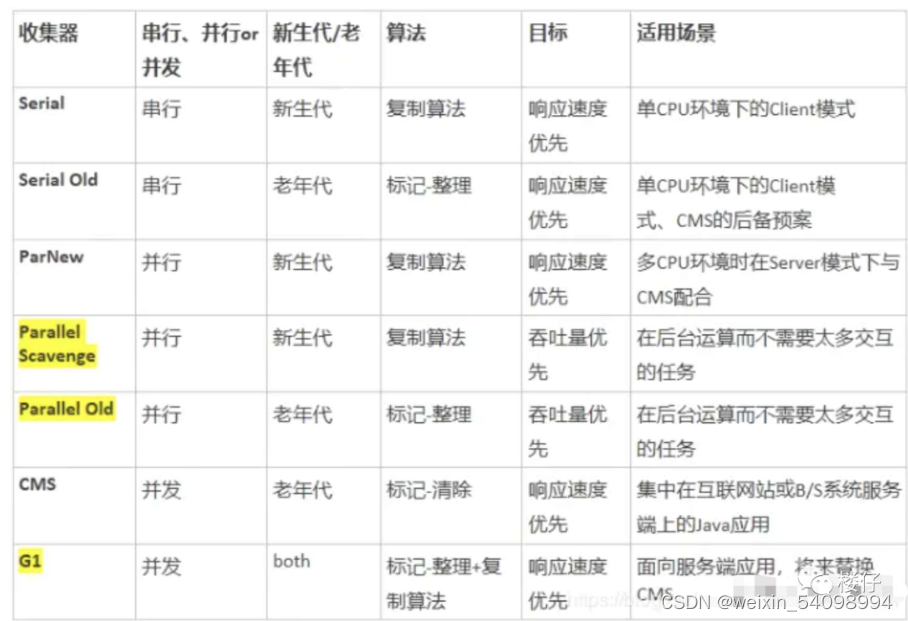
GC Root：

常量、静态变量、JNI对象、临时变量，活跃的Thread类，系统类。

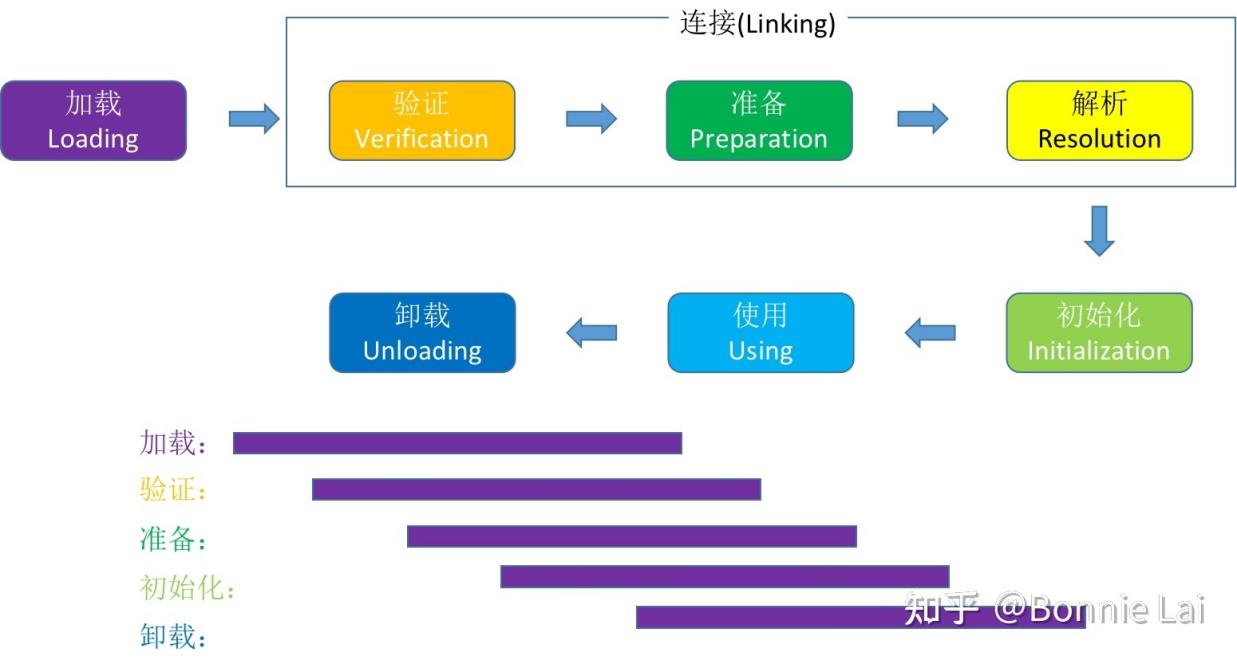
### 1.3.2 垃圾回收算法

标记-清除算法、复制算法、标记-整理算法。

### 1.3.3 垃圾收集器



## 1.4 类加载



### 1.4.1 类加载的时机

* 使用new 关键字实例化对象
* 读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）
* 调用一个类的静态方法的时候
* java.lang.reflect包的方法对类进行反射调用
* 初始化子类
* 虚拟机启动时加载包含main方法的类

### 1.4.2 加载

加载类的二级制字节流，方法区创建类相关静态数据的结构体，在堆中生成Class对象。

### 1.4.3 验证

文件格式、元数据、字节码、符号引用验证。

### 1.4.4 准备

为类的静态变量分配内存，并将其初始化为默认值（0值）。

### 1.4.5 解析

把类中的符号引用转换为直接引用。

### 1.4.6 初始化

类静态变量赋初始值。同一个类加载器下，一个类只会初始化一次。

## 1.5 引用

### 1.5.1 强引用

直接用等号连接实例对象，就是强引用。强引用是一个概念，没有对应的实体类。

### 1.5.2 软引用

对应实体类SoftReference。只有在内存不足时才会回收引用对象。

适合用在缓存中，假如用户访问的系统中需要加载很多图片，内存够用的时候可以缓存很多图片，假如内存不够用了，再把图片先回收掉也无妨，下次需要的时候再加载一次即可。

### 1.5.3 弱引用

对应实体类WeakReference。只要GC就会被回收。

用于防止内存泄漏。

### 1.5.4 虚引用

对应实体类PhantomReference。get方法获取到的永远为null，在任何时候都有可能被回收。

可以配合引用队列来观察对象是否被回收，对象被回收时可以从队列的poll方法获取对象。

## 1.6 并发相关

### 1.6.1 Synchronized

Synchronized是非公平锁、可重入锁、独享锁（互斥锁）、悲观锁。锁升级的机制用于优化Synchronized，其三种状态：偏向锁，一直被一个线程访问，可自动获取；轻量级锁，被另一线程访问，则偏向锁升级为轻量级锁，其他线程自旋等待；重量级锁，自旋等待的次数到达阈值，锁就会膨胀为重量级锁。

修饰静态方法，是用类锁锁住方法；修饰非静态方法，是用对象锁锁住方法。

### 1.6.2 CAS

CAS是英文单词Compare And Swap的缩写，比较并替换。是乐观锁。

CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值V1，要修改的新值V2。更新一个变量的时候，只有当变量的预期值V1和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为V2。

缺点：CPU开销大、不能保证代码块的原子性。ABA问题，解决办法是版本号比较，利用AtomicStampedReference可实现版本号比较的CAS机制。

底层实现是unsafe，最终调用汇编指令lock cmpxcg，完成比较和修改。

### 1.6.3 volatile

保证可见性，不保证原子性。可以防止指令重排。

原理是内存屏障：

* 保证指令重排序时屏障前后的指令不互通。
* 保证对缓存的修改立刻修改到主存。
* 写操作使得其他CPU的缓存失效。

### 1.6.4 AQS

## 1.7 线程

### 1.7.1 ThreadLocal

用于实现一个变量在不同的线程中维护不同值。

实现原理是在线程各自的Thread对象（Thread对象就是各自线程独有的）中维护一个ThreadLocalMap对象，来保存该线程的ThreadLocal对象们（这些对象都是该线程独有的）。ThreadLocalMap内部是用一个Entry数组实现的，Entry是个弱引用对象，弱引用了ThreadLocal对象，key是ThreadLocal对象自身维护的HashCode（初始化是用一个静态的原子整型计算实现的），value则是ThreadLocal对象对应的值。

### 1.7.2 线程池

判断顺序是：先判断核心线程是否已满，满了则判断等待队列是否已满，满了才判断是否放入非核心线程处理，非核心线程已满才触发饱和策略。默认饱和策略是抛出异常，丢弃任务。

## 1.8 集合

### 1.8.1 LruCache

使用LinkedHashMap实现，设置为访问排序。

### 1.8.2 ArrayList

用动态扩容的数组实现。

### 1.8.3 LinkedList

可以当作队列/双端队列使用。用双向链表实现，头节点无数据。

### 1.8.4 HashMap

用数组加链表实现，hash冲突的链表长度大于8则采用红黑树，小于6则恢复成链表。

ConcurrentHashMap在数组变量增加了volatile关键字，保证get时不需要加锁就能获取到正确的值；set方法中使用CAS机制，无冲突的插入则不加锁，在冲突时才给链表加锁，再执行链表操作。

LinkedHashMap，增加了节点类似双向链表的维护，支持遍历时按照插入顺序或访问顺序。

### 1.8.5 PriorityQueue

默认为小根堆。New PriorityQueue<>((x, y) -> y - x)可以创建大根堆。

## 1.9 I/O

## 1.10 泛型

## 1.11 反射

## 1.12 注解

注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。

分为标准注解（Java自带）、元注解（注解的注解）和自定义注解。

自定义注解常用的元注解：

@Retention 标识注解保存的级别，SOURCE为只在源代码中保留，可在编译前使用，多用于代码限制或提示（@Override）；CLASS为只保留到类文件，加载到JVM时被丢弃，为默认值，多用于方法参数的限制（@NonNull，@LayoutRes等）；RUNTIME是保存到运行时，可以反射读取（@BindView）。

@Target 标识注解的应用范围。如类、方法、成员变量、方法参数、包、构造函数等。

## 1.13 编译

# 二、Android基础

## 2.1 虚拟机

相比于JVM基于堆栈，Android虚拟机基于寄存器。

Dalvik虚拟机，支持即时编译JIT（Just In Time），在程序运行的过程中选择热点代码（经常执行的代码）进行编译或优化成odex。以及Interpreter（解释执行），就是一边把字节码翻译成当前硬件平台的机器码一边执行。启动比较快，就是执行效率低下。

Android 5.0后统一使用ART虚拟机（Android Runtime）。采用AOT (ahead of time) ，在App安装的时候就通过AOT编译器将.dex文件通过dex2oat编译为对应的.oat二进制文件，dex中的字节码会被编译成本地机器码。

## 2.2 IPC

进程间通信。

### 2.2.1 Linux IPC

主要有：管道、信号、文件、信号量、共享内存、Message和Socket。

管道通过内核缓冲区实现，一端读一端写，使用简单但只能单向通信。

信号用于通知目标进程状态改变或异常，可以实现软中断，有一定延时性。

文件实现IPC，读写比较灵活，但不够安全，速度较慢。

共享内存是最快的IPC形式，系统调用mmap映射文件实现共享内存。

信号量作为进程间以及同一进程不同线程之间的同步手段，为了规范共享资源的访问规则。

消息队列Message，相比于管道会把数据分成一个个独立的消息体。

Socket可用于不同机器之间的进程间通信。

### 2.2.2 Android IPC

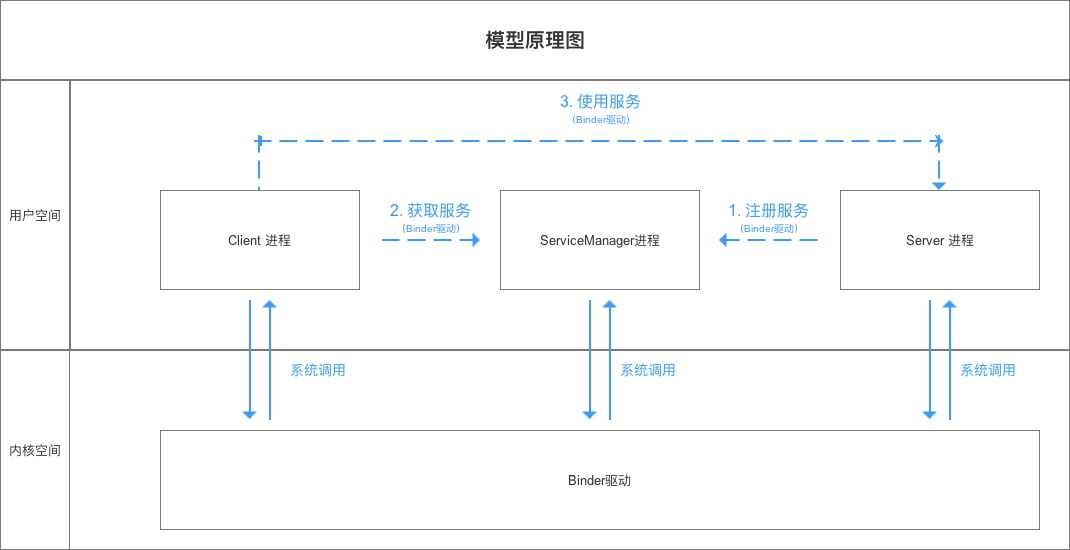
主要有：Binder、文件共享、ContentProvider、Broadcast、Socket。

四大组件以及Messenger的IPC通常都是Binder实现的

### 2.2.3 Binder

在APP进程被fork出来后，执行初始化的过程中会创建自己的Binder线程池。

Binder只能传输1M-8k的数据大小，正好是mmap映射的大小。



ServiceManager用于注册与查询服务，类似路由；Binder驱动运行于内核空间，用于联结三者进程。

IPC过程：

* Server注册服务，向Binder驱动发起请求，最后在serviceManager添加该Server。
* Client获取服务，Client绑定服务，最终获取Binder代理对象。
* Client使用服务，Client发送参数至Server，Server根据需要调用相应的方法，再把结果返回Client进程。

优点：高效（mmap）、安全（ServiceManager会校验进程的uid/pid）、使用简便。

## 2.3 消息机制

### 2.3.1 Handler机制

每个线程都有一个Looper对象，通过ThreadLocal实现。每个Looper有一个消息队列，各个线程的消息会被各个线程的Handler post到各自Looper的消息队列中，Looper会开启一个循环不断取出消息并交给相应的Handler处理，从而Handler将会处理来自各个线程的消息，这样就实现了线程的切换。

子线程使用Handler必须先执行Looper.prepare，以及Looper.loop开始轮询。

消息的延时通过消息的when参数来标识，插入消息队列时会通过when参数排序，所以取出的消息是按照需要执行的顺序进行的。若未到执行时间，则会挂起相应的时间来实现延时。

消息的插入和取出的代码块需要用synchronized锁住，锁为消息队列。消息有消息池，不能直接用构造方法创建。

同步屏障，会设置消息的target为null，开启时只处理异步消息。

Looper的loop开启循环，获取Queue的下一个消息（可能阻塞），然后分发消息。Handler处理消息是如果msg有callback则只执行msg的callback；否则，先处理Handler的callback的handlemessage方法，如果该方法返回true则结束，返回false才会执行Handler的handlemessage回调。

Queue获取下一个消息也会开启一个循环。先是开启epoll，然后进入锁住的代码块，先判断当前是否为屏障（Target==null），是的话就取出最接近的异步消息来处理。处理消息时，先判断是否到执行时间，未到则记录要等待的时间交给下次循环设置给epoll；到时间则标记消息已使用并返回消息。没有消息处理，则会判断是否有需要执行的idleHandler，没有则continue进入下个循环阻塞；有则在锁代码块之外处理idleHandler。

### 2.3.2 HandlerThread

继承Thread，内部维护了自己线程的Looper。

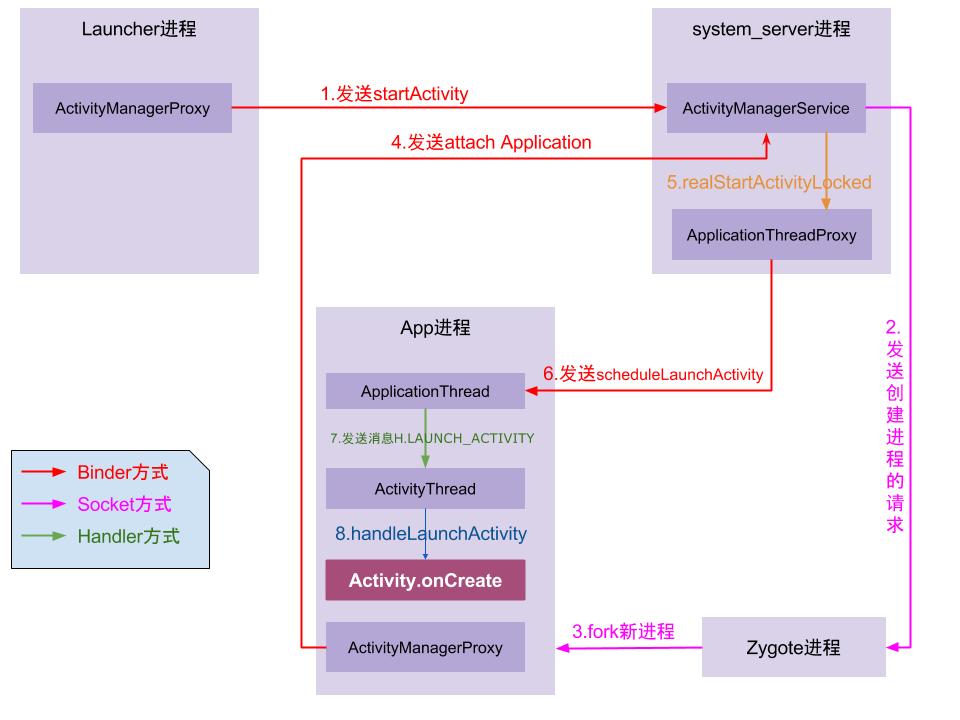
Run方法会准备Looper，在上锁的代码块中，赋值Looper并notifyAll，在getLooper方法中若发现Looper变量为空，则进入一个while循环wait，等待run方法赋值成功才会跳出循环。这样来保证异步线程getLooper能拿到准备好的Looper对象。

## 2.4 APP启动流程

### 2.4.1 全流程

应用启动，可以是从桌面点击图标，也可能是其他进程拉起。

冷启动时需要创建新的应用进程，热启动直接启动Activity。



* 点击桌面App图标，Launcher进程采用Binder IPC向system\_server进程发起startActivity请求；
* system\_server进程接收到请求后，向zygote进程发送创建进程的请求；
* Zygote进程fork出新的子进程，即App进程；
* App进程，通过Binder IPC向system\_server进程发起attachApplication请求；
* system\_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向App进程发送scheduleLaunchActivity请求；
* App进程的binder线程（ApplicationThread）在收到请求后，通过handler向主线程发送LAUNCH\_ACTIVITY消息；
* 主线程在收到Message后，通过消息机制创建目标Activity，并回调onCreate()等方法。

ActivityThread 应用的入口类，系统通过调用main函数，开启消息循环队列。ActivityThread所在线程被称为应用的主线程（UI线程）。

ApplicationThread是ActivityThread的内部类，继承IApplicationThread.Stub，是一个IBinder，是ActivityThread和AMS通信的桥梁，AMS则通过代理调用此App进程的本地方法，运行在Binder线程池。

H继承Handler，在ActivityThread中初始化，即主线程Handler，用于主线程所有消息的处理。本片中主要用于把消息从Binder线程池切换到主线程。

Intrumentation具有跟踪application及activity生命周期的功能，用于监控app和系统的交互。

ActivityManagerServiceAndroid中最核心的服务之一，负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似，因此它在Android中非常重要，它本身也是一个Binder的实现类。

ActivityTaskManagerService管理activity及其容器（task, stacks, displays）的系统服务（Android10中新增，分担了AMS的部分职责）。

ActivityStarter 用于解释如何启动活动。该类收集所有逻辑，用于确定Intent和flag应如何转换为活动以及相关的任务和堆栈。StartActivity会通过ATMS利用工厂类创建ActivityStarter来进行具体实现，在execute方法会进行一些校验和判断权限，包括进程检查，intent检查，权限检查等，后面就会创建 ActivityRecord ，每个 Activity 都会对应一个 ActivityRecord 对象，接着就会调用 startActivityUnchecked 方法对要启动的 Activity 做任务栈管理。在startActivityInner方法中，根据启动模式计算出 flag，然后在根据 flag 等条件判断要启动的 Activity 的 ActivityRecord 是需要新创建 Task 栈 还是加入到现有的 Task 栈，在为 Activity 准备好 Task 栈之后，调用了 mRootWindowContainer.resumeFocuredTasksTopActivities 方法。最终进入ActivityTaskSupervisor的方法startSpecificActivity启动activity。

ActivityStack用来管理系统所有的Activity，内部维护了Activity的所有状态和Activity相关的列表等数据。

ActivityTaskSupervisor负责所有Activity栈的管理。AMS的stack管理主要有三个类，ActivityTaskSupervisor，ActivityStack和TaskRecord。

ClientLifecycleManager客户端生命周期执行请求管理。

ClientTransaction是包含一系列的 待客户端处理的事务 的容器，客户端接收后取出事务并执行。

Android 11的详细启动流程：

<https://juejin.cn/post/7130182223231188999>

<https://juejin.cn/post/7172464885492613128>

<https://juejin.cn/post/7195458962328649788>

### 2.4.2 与Window关联

APP进程创建完成后，首先调用AT的main方法，创建主线程，attach ApplicationThread，与AMS建立联系。

ApplicationThread调用scheduleTransaction开始Activity生命周期的调度。在消息机制处理后，执行handleLaunchActivity，初始化WindowManagerGlobal,，调用performLaunchActivity，创建Activity对象，然后判断Application是否存在，不存在则通过LoadedApk来创建Application对象，执行其onCreate生命周期，之后调用Activity的Attach方法跟Application绑定——在此处创建了PhoneWindow对象，并使其跟WMS建立联系。

接着执行Activity的onCreate生命周期，传入视图树，创建DecorView。

在onResume后，WM会addView添加DecorView，最终由WindowManagerGlobal执行addView。此时会创建RootViewImpl，并调用setView方法。RootViewImpl是真正管理View的核心类，WindowManagerGlobal是RootViewImpl的封装。setView会通过Binder调用WMS的addWindow方法。再调用requestLayout才会执行View的三大回调。

### 2.4.3 USAP机制

Android 10引入，USAP(Unspecialized App Process)。通过prefork的方式提前创建好一批进程，当有应用启动时，直接将已经创建好的进程分配给它，从而省去了fork的动作，因此可以提升性能。进程池的大小维10，参考文章：https://www.jianshu.com/p/392a76aca6d2

## 2.5 绘制原理

### 2.5.1 屏幕刷新原理

CPU计算帧数据，交给GPU渲染图像数据，存到Buffer缓冲区，最后由Display将Buffer内容呈现出来。

双缓存（Frame Buffer和Back Buffer）为了解决画面撕裂，前后缓冲区交替使用。三缓存（新增 Graphic Buffer）是在前者基础上丢帧时启用，为了防止后续丢帧，但会造成画面延迟。

Display根据Vsync垂直同步信号来判断刷新时机。

ViewRootImpl是控制视图绘制的关键类，它通过Window类（通常是PhoneWindow）添加View，在Activity启动时、动画执行时或视图请求刷新时，都会最终调用ViewRootImpl的scheduleTraversals方法调度绘制流程。

Choreographer是实现绘制流程调度的关键类，它通过监听底层的Vsync垂直同步信号来判断何时执行下一次的绘制。卡顿监控通常使用它的postFrameCallback来实现。为了实现绘制任务在下一个Vsync信号到来时立即执行，用到了Handler消息机制的异步消息和同步屏障机制。

SurfaceFlinger管理了Vsync信号的传递。它通过两个延时源，承上启下地控制了上层的绘制和下层的Surface数据合成。对于下一次Vsync信号，由它请求底层硬件产生的，请求的条件包括一个Enable标志以及是否存在Connect来感知信号。Choreographer的监听会设置这样一个Connect来感知信号。

绘制任务正是我们熟悉的measure、layout和draw流程。绘制前获取不到视图宽高。measure/layout/draw 走完后会在VSync到来时进行缓存交换和刷新。

画面静止时，仍然会每16.7ms刷新一次屏幕，但不会有绘制流程。Activity在onResume时创建了所有相关的View对象，但真正的绘制要等下一次Vsync来临才会开始。

### 2.5.2 invalidate刷新视图

调用invalidate方法时，视图并不是只刷新自己，将会把需要刷新的区域标记为dirty，并调用parent的invalidate，从视图树一路往上递归，标记需要刷新的视图节点，最终调用viewRootImpl的方法，刷新整个视图树，告诉特定的视图刷新特定的位置。

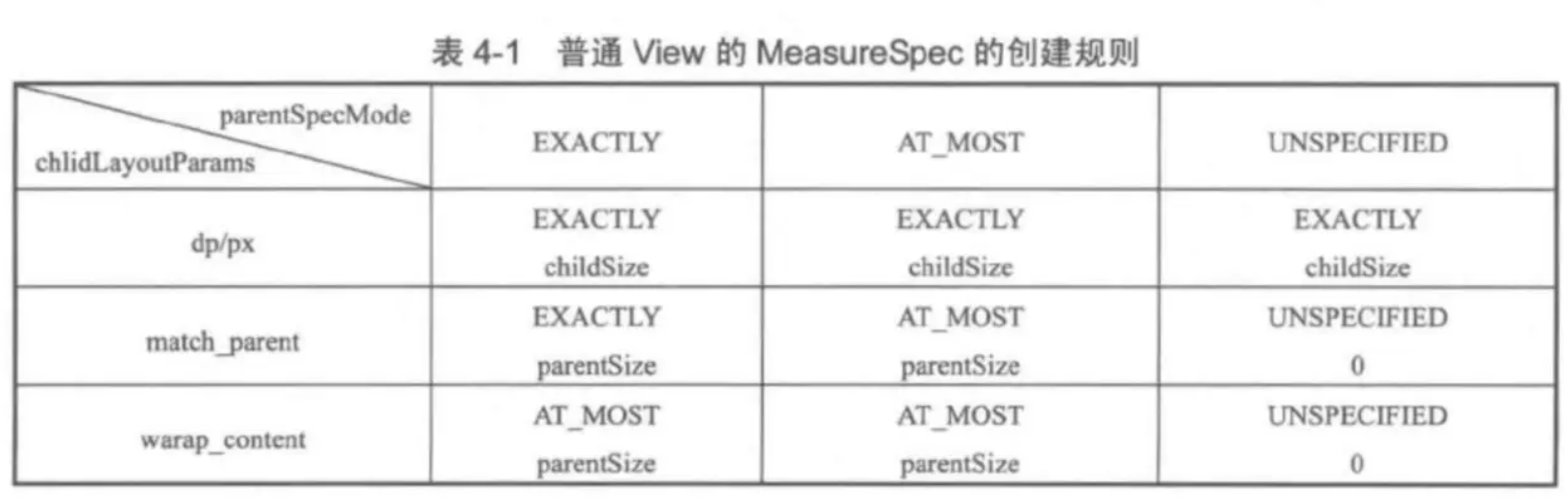
### 2.5.3 UI标准

Dp=dpi/160 \* px

Bitmap.Config.RGB\_565：每个像素用16比特位表示，占2个字节，RGB分量分别使用5位、6位、5位。

Bitmap.Config.ARGB\_8888：每个像素用32比特位表示，占4个字节，由4个8位组成，ARGB分量都是8位。

### 2.5.4 绘制流程



Measure过程是先递归地测量子View，再根据结果测量自己；Layout则是布局子View，子View递归地布局自己的子View。

## 2.6 事件分发

### 2.6.1 ViewGroup事件分发

Down事件：主要用于确定事件分发给哪个视图。先判断是否拦截，如果不拦截，会将子视图按Z值排序，然后开始遍历，寻找符合点击区域条件的子视图，分发后跳出遍历，返回true消费事件；如果拦截，会自己处理。

Move事件：处理时会先判断是否拦截，不拦截则可以利用Down事件保存的视图链直接分发给特定视图。拦截则自己处理事件：取消子视图的处理事件，将会在下一个Move事件到来时自己消费事件（同Down事件处理一样）。

处理滑动冲突，可以根据事件的具体滑动方向判断把事件交给哪个View处理。RequestDisallowInterceptTouchEvent父布局的方法可以防止事件被父布局拦截。解决ViewPager嵌套ListView的滑动冲突，可以在ViewPager的onInterceptTouchEvent回调处理：在Down事件时执行Supervisor并返回false让ListView处理，其他情况返回true；在ListView的dispatchTouchEvent回调，Down事件阻止父布局拦截，Move事件时判断若是横向滑动则取消阻止父布局拦截。

### 2.6.2 事件输入模拟

Linux的minitouch库，uinput模块都可以实现；也可以在Framework层，使用Instrumentation的sendPointerSync方法发送事件；也可以使用InputManager的injectEvent，但需要反射调用。

## 2.7 打包流程



Dx工具生成dex文件，aapt2编译资源文件。

## 2.8 工具与插件

### 2.8.1 Android studio插件

### 2.8.2 Gradle工具

Gradle自定义插件

Module工程需要使用groovy插件，插件入口类需要实现Plugin接口。为了实现Gradle task的功能可以使用Transform API。

若要发布到Maven，需要增加resource目录，MATA\_INF->gradle-plugins下创建properties文件，其文件名前缀即业务方应用插件时的插件名，文件内需指

明入口类的全限定名。

### 2.8.3 JVM TI

通过JVM TI提供的API实现，它是JVM提供的一套后门机制。它是JPDA—— Java 平台调试架构（Java Platform Debugger Architecture）的一环，位于最底层。

它本质上是在JVM内部的许多事件进行了埋点，通过这些埋点可以给外部提供当前上下文的一些信息，甚至可以接受外部的命令来改变下一步的动作。外部程序一般利用C/C++实现一个JVMTIAgent，在Agent里面注册一些JVM事件的回调，当事件发生时JVMTI调用这些回调方法，Agent可以在回调方法里面实现自己的逻辑。JVMTIAgent是以动态链接库的形式被虚拟机加载的。

先通过Agent\_onAttach的JavaVM获取jvmtiEnv环境，然后通过GetPotentialCapabilities方法获取支持的jvmti能力，AddCapabilities方法加入所有支持的能力，最后返回JNI\_OK。然后在jvmtiEventCallbacks中选择需要监听的事件方法，赋值回调方法，接着用环境参数setEventCallbacks注册，最后调用setEventNotificationMode方法，开启对应的监控。

程序需要通过Debug的attachJvmtiAgent方法挂载到虚拟机。但线上不能使用Debug包的Java类，需要使用Native的挂载方法，涉及libopenjdkjvmti库，VMDebug\_nativeAttachAgent方法，调用前还需SetJdwpAllowed方法开启调试，Agent\_onAttach方法获取环境时参数要改用ART\_TI\_VERSION\_1\_2。

## 2.9 NDK

静态注册，利用javah命令生成头文件，方法名是全限定名相关。使用简便，但灵活性不好。

动态注册，通常是重载JNI\_OnLoad(JavaVM \*vm,void \*reserved)方法，自定义方法映射关系，调用方法：env->RegisterNatives(clazz, gMethods, numMethods)，第一个参数是Java对应的类，第二个参数是JNINativeMethod数组，第三个参数是JNINativeMethod数组的长度，也就是需要注册的方法的个数。

Java调用native直接使用native关键字，并注册方法。

Native调用Java，可以使用env的findClass方法，类似反射的原理获取Java类、方法、变量。

JNI方法必须使用C语言标准（extern C）：一是防止重装方法，二是JNIEnv的所有指针都是C语言标准。JNIEXPORT对外暴露，JNICALL标记。

### 2.9.1 Cmake

用于替换makefile的构建文件。

Cmake\_minimum\_required 指定Cmake支持的最低版本。

Add\_subdirectory 添加子cmake文件的目录。

Include\_directories 导入头文件目录。

导入库文件。

Add\_library 构建目标，第一个参数是库名称后缀（libxxx）；第二个参数表示库的类型（STATIC对应静态库.a，SHARED对应动态库.so）；第三个参数表示包含的源文件（cpp），可以使用变量表示多个文件（file(GLOB v \*.c \*.h \*.cpp)则变量v表示所有的.c、.h、.cpp文件名集合 ${v}）。

Find\_library 在Android SDK查找官方库使用变量表示。第一个参数是变量名，第二个参数是库名称。

Target\_link\_libraries 表示链接所有使用的库，可包含变量。

### 2.9.2 图像识别初步

涉及库OpenCV。Mat数据结构用于描述一张图片，类似android的Bitmap。

为了方便，可以先将图片无损压缩到固定大小。

为了使处理更高效，可以先对图片做灰度化处理。即将三维的RGB色值转为一维的灰度色值：f(x, y) = 0.30 \* R(x, y) + 0.59 \* G(x, y) + 0.11 \* B(x, y)

再将灰度色值转化为二进制的黑白两色值，即二值化处理。处理方法：将[0, 255]的灰色值，用一个阈值k，区分成0和255两个值（如k=150，则灰色值大于150则取255，否则取0）。

然后可以进行轮廓检测，即用矩形框将连续的黑点闭包框起来。

若轮廓检测太细致可能搜索不到目标，可以考虑将黑点膨胀。

### 2.9.3 调音初步

使用Fmod库。

System，用于加载、播放声音。需要初始化。

DSP，是声音的数字信号数据抽象，可以修改参数来实现调音。可以通过System针对某个效果来创建。

Channel作为音轨的代码抽象，可以为不同的音轨添加对应的DSP。

### 2.9.3 音视频初步

涉及ffmpeg工具。

一个视频文件可以包含两个视频流。

编码，主要为了压缩视频。主要考虑以下方面来编码：

* 空间冗余，可以把相邻的相同颜色像素点归结为一个记录。
* 时间冗余，可以把相邻的帧描述为差分数据。
* 视觉冗余，可以把人眼不敏感的色值剔除。
* 信息熵冗余，可以把相邻数据值，用哈夫曼算法表示（11112222->4142）。
* 知识冗余，编码时加入常识性认知信息。

编解码标准：国际电联ITU-T（H.264，H.263，H.261），运动静止图像专家组（MPEG系列）。

H.264对应MPEG4 AVC，对应代码“video/avc”（H.265对应代码“video/hevc”）。H.264码流文件分为VCL和NAL，VCL为视频编码层，对应编码后的视频数据序列；NAL为网络提取层，负责用网络要求的方式来打包和解包，类似Android Http实现中对bean的序列化和反序列化。

IPB帧，I帧是关键帧，表示一个完整的图像信息；P帧是前向预测编码帧，记录与前一帧的差别；B帧是双向预测帧，记录与前后帧的差别。

YUV颜色空间是RGB的优化，Y是明亮度，也称灰阶值，U与V表示色调和饱和度。

Y=0.299R+0.587G+0.114B

U=-0.147R-0.289G-0.436B

V=0.615R-0.515G-0.100B

Android摄像头返回的数据是NV21格式的字节流，它是先排布所有像素点的Y数据，再把像素点的V和U数据交替排布。

通常输入输出使用I420格式的字节流，它是先排布所有像素点的Y数据，再排布所有像素点的U数据，最后排布所有像素点的V数据。

MediaCodec是用于访问Android底层的多媒体编解码器，属于硬编码。Ffmpeg属于软编码，兼容性好，但功耗大。

其实现是生产者消费者模式，有一个输入队列和一个输出队列，使用时将数据传入输入队列，在输出队列判断状态，获取结果。

MediaMuxer用于将编码完成的数据混合成一个视频文件容器，之后才能进行编码。

GOP关键帧间隔，相同码率下，GOP越大，则P/B帧越多，画面质量越好，但风险也越高，当一个I帧图像质量比较差时，会影响更多的画面，要等下一个GOP才能恢复画面。

### 2.9.4 直播初步

网络协议RTMP，实时信息传输协议，类似HTTP，是应用层协议。

Librtmp库用C语言实现了socket封装实现RTMP。

音频采集，使用AudioRecord或者OpenSL ES。AudioRecord创建对象时，需要传入声音源（如麦克风）、采样率、声道类型、PCM量化位（每一个声音数据所占bit大小），以及最小缓冲区大小。AAC格式总是双声道。

视频采集，使用MediaProjection。其调用流程，需要先获取系统服务MediaProjectionManager，为了采集屏幕数据，要先取得Intent，startActivityForResult，在onResult中获取MediaProjection对象，再创建一个虚拟屏幕（创建的虚拟屏幕可以传入一个Surface接收数据）。MediaCodec正好可以创建一个Surface来接收输入数据（createInputSurface方法）。

采集数据后，需要进行RTMP封包，RTMP数据流格式与FLV格式类似。音频包数据有编码数据和音频数据，分别需要在数据前拼接0xAF00和0xAF01。

视频包数据有SPS和PPS数据，需要拼接一个0x1700000000；非关键帧数据，需在前面拼接0x2701000000，然后直接发出；关键帧数据，需要在前面拼接0x1701000000，并且在前面发一个SPS与PPS数据再发出关键帧数据。主要通过起始码（占4个byte）之后第一个字节的低5位来判断视频数据类型：5是表示IDR关键帧，7是序列参数集SPS。

## 2.10 系统启动

操作系统被BootLoader引导启动后，先启动内核态的鼻祖进程idle（pid=0），初始化进程、内存管理，加载binder驱动，以及其他驱动；然后启动用户态鼻祖进程init（pid=1）以及内核守护进程kthredd（pid=2）；由init进程解析init.rc，fork出Java进程的鼻祖zygote进程，zygote进程由Java代码和c代码共同完成启动流程。

Zygote初始化，会先启动虚拟机，设置虚拟机参数；然后注册所有系统代码相关的JNI方法；接着通过JNI调用ZygoteInit的main方法，进入Java代码：预加载资源；new ZygoteServer创建Socket服务；fork系统进程；ZygoteServer.runSelectLoop开启循环等待Socket访问。

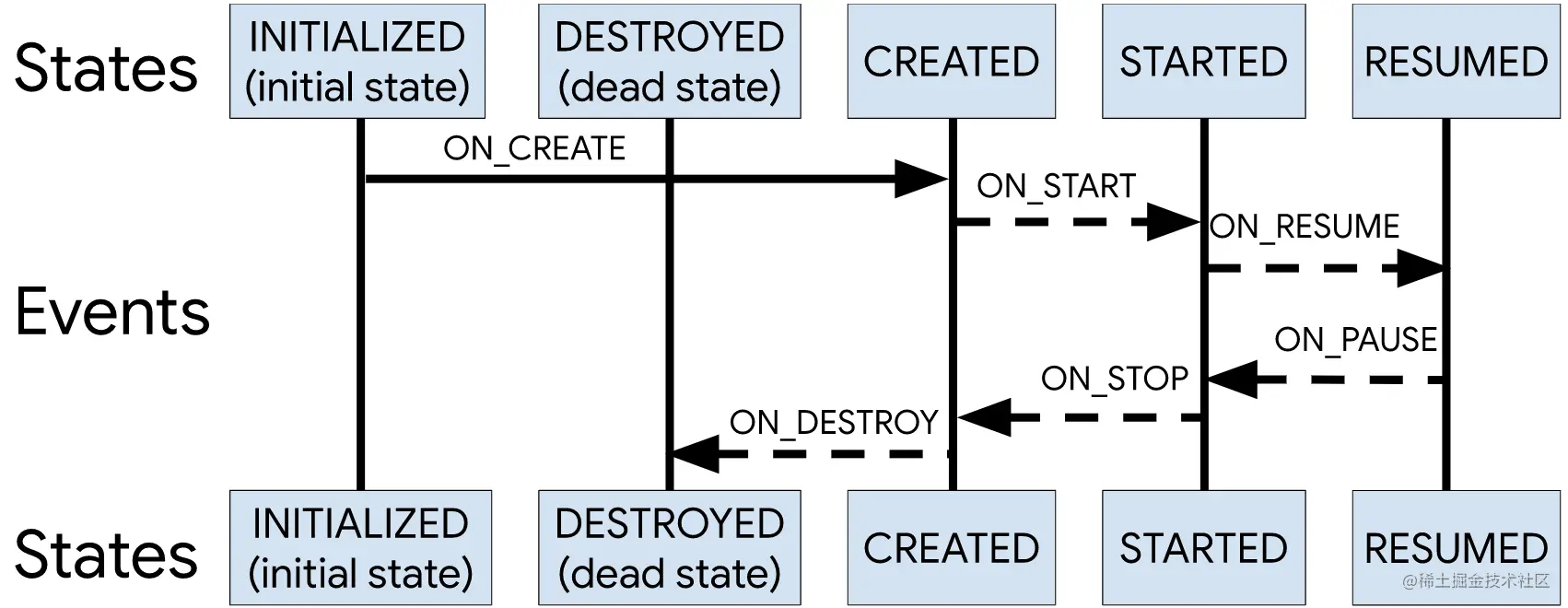
Zygote开始fork SystemServer，通过反射执行SystemServer的初始化流程，关闭Zygote拷贝过来的Socket。SystemServer执行main方法实际执行到run方法，启动系统服务，一共三类：引导服务、核心服务和其他服务。引导服务中会先启动ATMS，通过反射创建对应的lifecycle的对象执行onStart回调，添加Binder至SM；然后启动AMS，创建栈管理ActivityStackSupervisor对象，设置recent task；然后setSystemProcess，添加各种监控Binder到SM，如meminfo、dbinfo、cpuinfo等，以及设置WMS等服务；之后调用AMS的systemReady方法，会开始启动Launcher APP。

# 三、Android进阶

## 3.1 官方库

### 3.1.1 Lifecycle

围绕生命周期事件和生命周期状态来维护Activity和Fragment的生命周期。



Activity实现了LifecycleOwner，主要通过ReportFragment来分发生命周期事件，SDK大于等于29则直接注册生命周期监听回调，SDK小于29则是用add透明Fragment的方式来感知生命周期。生命周期事件交给LifecycleRegistry来处理，用一个Map维护所有观察者。

这个Map是FastSafeIterableMap，可以保证观察者的添加顺序，并且支持遍历时增删观察者数据。回调事件会保证所有事件的顺序一定且粘性。

state的大小关系是：DESTROYED < INITIALIZED < CREATED < STARTED < RESUMED。

### 3.1.2 LiveData

Observe添加观察者，实际上添加的是包装的观察者LifecycleBoundObserver，它同时实现了LifecycleEventObserver接口，利用Lifecycle在onDestroy事件到来时移除用户添加的观察者。

判断观察者是否活跃：至少是STARTED状态。活跃状态改变的回调是在观察者数量从0变1或从1变0的时候。

observeForever添加观察者，实际上添加的是保持活跃状态的AlwaysActiveObserver。

postValue是用线程池把setValue的Runnable抛到主线程处理。

MediatorLiveData是LiveData的子类，可以实现多个数据的观察或者观察数据的转换。可以传入其他LiveData，把多个Observer 包装成多个Source实例，添加到列表mSources中。

### 3.1.3 ViewModel

ViewModel用于代替MVP中的Presenter，用于解决Activity异常销毁需要重建和Presenter需要维护UI的销毁等问题。它的生命周期比Activity长，解决了意外销毁的问题；它不需要持有UI层引用，正是通过LiveData来实现对UI数据响应的。

UI层获取ViewModel是通过ViewModelProvider获取的，它的关键参数是ViewModelStore和Factory（默认工厂用反射创建ViewModel），Activity实现了ViewModelStore，通过NonConfigurationInstance来存储ViewModelStore，它的实例由ActivityClientRecord的lastNonConfigurationInstances存储，所以不受异常销毁Activity的影响。

### 3.1.4 RecyclerView

相比ListView而言，会在滑动时通过缓存来实现ViewHolder的复用。

首先获取的是屏幕内的缓存：ChangedScrape和AttachedScrape；然后获取屏幕外的缓存CachedViews，默认容量是2（可设置setItemViewCacheSize），缓存逻辑是一个先入先出队列；接着是开发者自定义的缓存ViewCacheExtension，通常不用；最后是缓存池，根据viewType存入一个SparseArray，每个viewType的容量是5。

### 3.1.5 Room

### 3.1.6 WorkManager

### 3.1.7 Compose

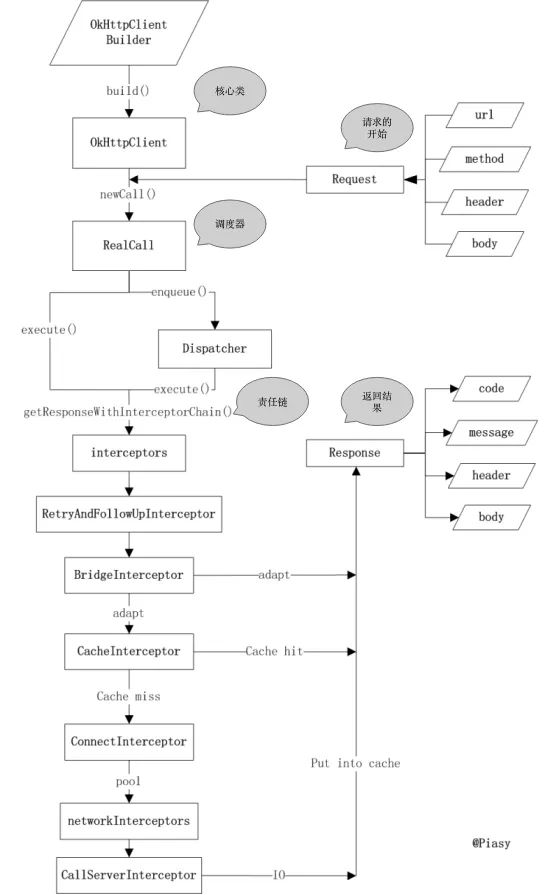
声明式编程，类似Flutter。@Composable注解。

自定义微件，MeasurePolicy将约束传给子微件，layout方法设置子微件的位置，最后调用Layout完成布局。

Modifier修饰，采用装饰器模式设计，最终调用then将所有属性的装饰者层层包装起来。实际测量时调用Modifier的foldOut方法，同样用装饰器模式包装ModifierLayoutNode。

## 3.2 三方库

### 3.2.1 Okhttp



OkHttpClient：这个是整个OkHttp的核心管理类，所有的内部逻辑和对象归OkHttpClient统一来管理，它通过Builder构造器生成。

Request和Response：Request是我们发送请求封装类，而Response是请求的结果，这两个类的定义是完全符合Http协议所定义的请求内容和响应内容。

RealCall ：负责请求的调度；同时负责构造内部逻辑责任链，并执行责任链相关的逻辑，直到获取结果。

拦截器使用责任链模式，最先执行的是addInterceptor传入的拦截器，按传入顺序执行，每个拦截器一次请求只执行一次；而addNetworkdInterceptor传入的拦截器在retryAndFollowUpInterceptor之后执行，它将跟随重试或重定向流程执行，一个请求可能执行多次。

* retryAndFollowUpInterceptor：失败和重定向拦截器。
* BridgeInterceptor：封装request和response拦截器。
* CacheInterceptor：缓存相关的过滤器，负责读取缓存直接返回、更新缓存。
* ConnectInterceptor：连接服务，负责和服务器建立连接这里才是真正的请求网络，DNS和Socket连接都在这里完成。
* CallServerInterceptor——执行流操作(写出请求体、获得响应数据) 负责向服务器发送请求数据、从服务器读取响应数据，进行http请求报文的封装与返回报文的解析。

调度器维护三个队列，准备异步队列、正在执行异步队列、正在执行同步队列。Enqueue一个Call之后，会创建一个AsyncCall来处理，先判断：正在执行异步队列没满（64个）并且同一个Host的请求没满（5个），就放入正在执行的异步队列，并且放入线程池执行，否则放入准备异步队列。完成Call之后会将Call从正在执行异步队列移除，并尝试从准备异步队列移动一个Call到正在执行异步队列并执行。

连接池为了提高连接的使用效率，主要在ConnectInterceptor中使用。通过一个双端队列维护一个连接池，空闲连接允许存在5个，如果地址和路由都符合则会使用连接池中的连接。使用ConnectionPool实现连接池，而且默认支持5个并发KeepAlive，默认链路生命为5分钟。

### 3.2.2 RxJava

主要考虑观察者模式来实现，在通知观察者的路径上，以及观察者回调暴露给使用者编写方法的具体回调实现。

链式调用采用装饰器模式，对于操作符的实现，是不断装饰被观察者，并持有观察者，实现不断链式调用观察者方法。

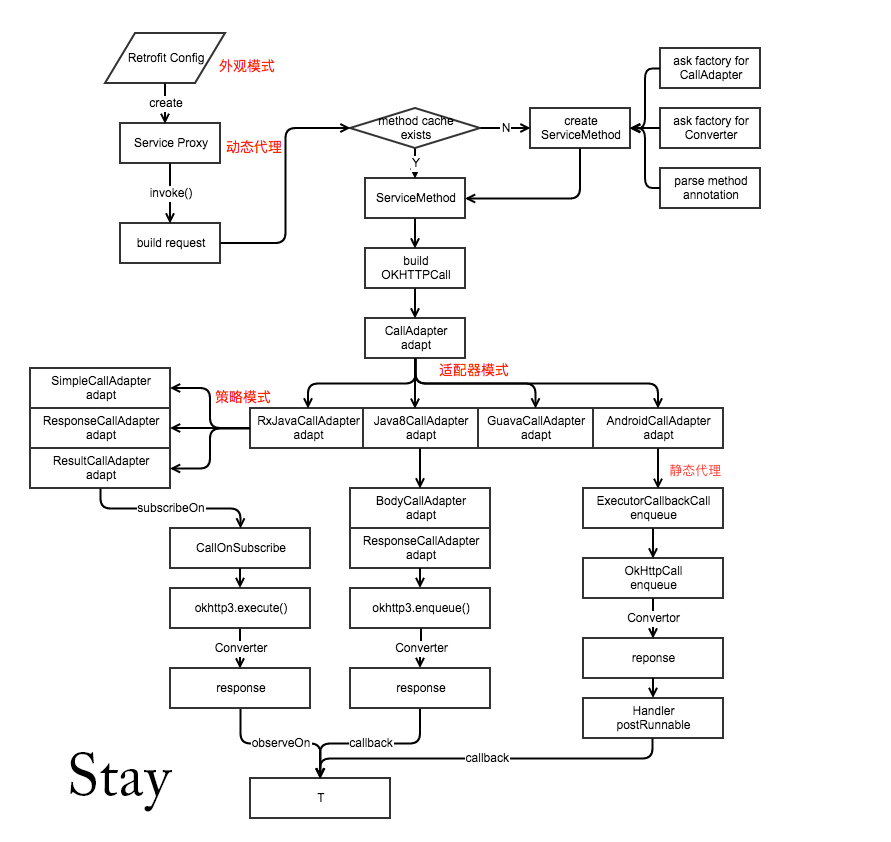
线程切换，subscribeOn只有第一次起作用，observeOn每次都起作用。

背压策略BackPressure：解决被观察者发送事件速度与观察者接收事件速度不匹配的情况。主要引入了一个事件缓存区，缓存被观察者发送的事件，观察者可以按需接收事件。在Rxjava2.0引入Flowable作为被观察者来解决背压问题。

### 3.2.3 Retrofit

用于封装优化Okhttp，做了四件事：请求前，统一配置网络请求头；适配请求的request，使得request得以复用；请求后，线程可按需切换；数据Bean的适配。

涉及设计模式（Builder、外观、代理、策略、适配器）、反射、动态代理等技巧。



### 3.2.4 Glide

缓存机制：活动缓存，存储正在显示的图片资源；内存缓存，LRU缓存，资源与活动缓存互斥；磁盘缓存，LRU缓存，持久化缓存。活动缓存和内存缓存是互斥的，活动缓存是为了消除内存缓存LRU回收导致正在显示的图片被回收的风险。

获取缓存先从活动缓存获取，获取不到则检查内存缓存，内存缓存获取到则把资源剪切到活动缓存，内存缓存获取不到再去磁盘缓存获取，磁盘缓存获取到则把资源复制加入到活动缓存，磁盘缓存获取不到再去网络或本地文件系统获取。

生命周期通过加入透明Fragment来监听Activity的生命周期。onDestroy时回收活动缓存，将资源移动到内存缓存。

相比Picasso，Glide是下载图片后修改大小适应Target后保存，而Picasso是保存原图，内存使用方面Glide更小。Glide默认使用RGB\_565，而Picasso使用ARGB\_8888，Glide还支持GIF图片显示。

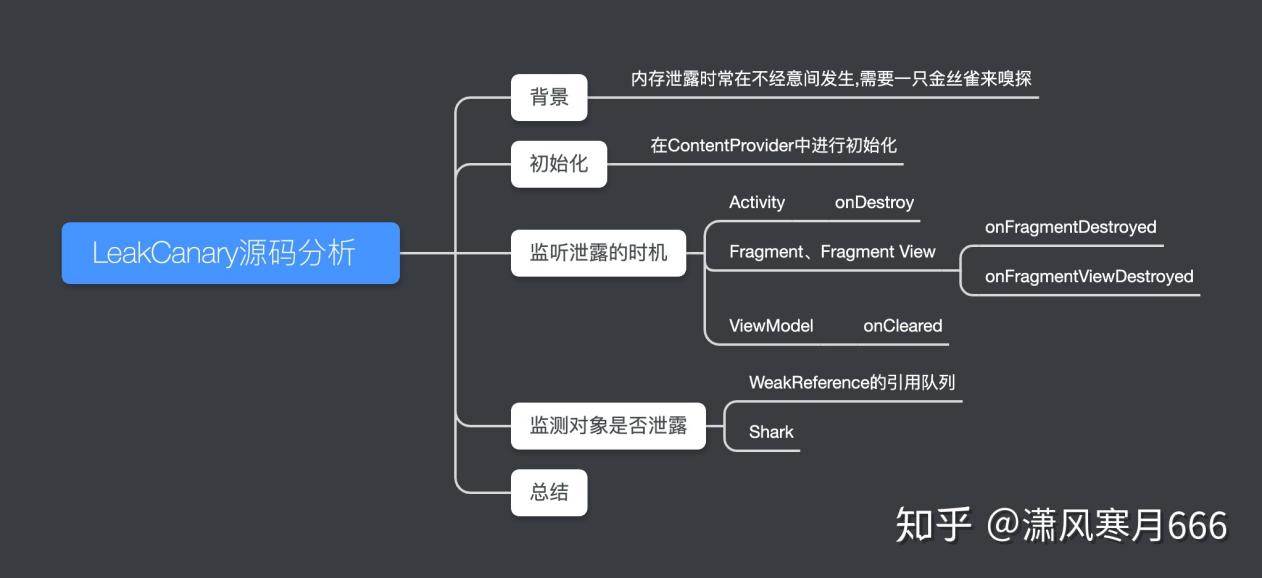
### 3.2.5 LeakCanary

1.\*版本需要在Application代码中初始化，2.\*版本是使用静态注册的ContentProvider自动初始化的。

原理主要是运用引用队列。弱引用和引用队列ReferenceQueue联合使用时，如果弱引用持有的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。在Application注册Activity生命周期回调，onDestroy以后，一旦主线程空闲下来（使用IdleHandler实现），延时5秒执行一个任务：先判断Activity有没有被回收？如果已经回收了，说明没有内存泄漏，如果还没回收，我们进一步确认，手动触发一下gc，然后再判断有没有回收，如果这次还没回收，说明Activity确实泄漏了。

具体实现：在监控对象需要被销毁时调用RefWatcher的watch方法，随机生成一个UUID，加入到RetainedKeys的集合中，这个UUID跟监控对象的弱引用关联起来。判断是否被回收时，先从弱引用队列中poll出所有对象（已回收），用各自关联的UUID删除RetainedKeys中的key，RetainedKeys中还剩下的对象表示对象没被回收。

若无泄漏，还需销毁弱引用对象；有泄漏，则需要堆分析工具Shark分析泄漏的对象，整理引用链，输出给用户。



## 3.3 架构

### 3.3.1 MVC、MVP、MVVM

MVC，Model-View-Controller，职责分类如下：

* Model，模型层，即数据模型，用于获取和存储数据。
* View，视图层，即xml布局
* Controller，控制层，负责业务逻辑。

MVC 的问题点如下：

* Activity/Fragment 责任不明，同时负责View、Controller，就会导致其中代码量大，不满足单一职责。
* Model耦合View，View 的修改会导致 Controller 和 Model 都进行改动，不满足最少知识原则。

MVP，Model-View-Presenter，职责分类如下：

* Model，模型层，即数据模型，用于获取和存储数据。
* View，视图层，即Activity/Fragment
* Presenter，控制层，负责业务逻辑。

MVP 的问题点如下：

* 会引入大量的IView、IPresenter接口，增加实现的复杂度。
* View和Presenter相互持有，形成耦合。

MVVM，Model-View-ViewModel，职责分类如下：

* Model，模型层，即数据模型，用于获取和存储数据。
* View，视图，即Activity/Fragment
* ViewModel，视图模型，负责业务逻辑。

官方Jetpack使用ViewModel+LiveData来实现。

### 3.3.2 AOP思想

AOP面向切面编程，OOP面向对象编程。OOP如果是把问题划分到单个模块的话，那么AOP就是把涉及到众多模块的某一类问题进行统一管理。AOP的优点是加强解耦与提高开发效率。

实现原理主要是代理模式的思想。

织入时机:

* 编译时：当一个类文件被编译时进行织入，这需要特殊的编译器才可以做的到，例如AspectJ的织入编译器。
* 类加载时：使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码。
* 运行时：切面在运行的某个时刻被织入，SpringAOP就是以这种方式织入切面的，原理应该是使用了JDK的动态代理技术。

静态织入：APT，AspectJ、ASM、Javassit

动态织入：Java动态代理，cglib、Javassit

### 3.3.3 APT注解处理

通常需要3个module完成整体功能。包括Annotation module定义注解，本质为library；API module提供业务方调用的方法，本质为library；compiler module处理注解，本质为Gradle plugin。

Compiler依赖Annotation，入口类继承Processor，主要实现init方法，从环境获取Filer、message、util等；实现process方法处理注解，从环境获取相应注解修饰的element集合。可利用Filer创建Java源文件。

API Module可以通过反射创建Compiler模块生成的类，调用具体的方法。

### 3.3.4 字节码插桩ASM

ASM是强大的字节码处理工具，基于一套Java API实现。

ClassWriter类是ASM中的核心API ， 用于生成一个类的字节码。 ClassWriter的visit方法定义一个类。

第一个参数V1\_1是生成的class的版本号，对应class文件中的主版本号和次版本号，即minor\_version和major\_version 。

第二个参数ACC\_PUBLIC表示该类的访问标识。对应class文件中的access\_flags。

第三个参数是生成的类的全限定名。对应class文件中的this\_class。

第四个参数是和泛型相关的，传入null表示这不是一个泛型类。这个参数对应class文件中的Signature属性（attribute）。

第五个参数是当前类的父类的全限定名。这个参数对应class文件中的super\_class。

第六个参数是String[]类型的，传入当前要生成的类的直接实现的接口。这个参数对应class文件中的interfaces 。

MethodVisitor是ClassWriter的visitMethod方法的返回，用于定义方法。

第一个参数是访问标志。这个参数对应method\_info中的access\_flags。

第二个参数是方法的方法名。对于构造方法来说，方法名为<init>。这个参数对应method\_info 中的name\_index，name\_index引用常量池中的方法名字符串。

第三个参数是方法描述符，描述返回值及参数，构造方法描述符为()V。这个参数对应method\_info中的descriptor\_index。

第四个参数是和泛型相关的，传入null表示该方法不是泛型方法。这个参数对应method\_info中的Signature属性。

第五个参数指定方法声明可能抛出的异常。无异常声明抛出，传入null。这个参数对应method\_info中的Exceptions属性。

ClassWriter最后转化为字节流数据，写入类文件即可。

AS插件ASM Bytecode Outline可以辅助处理字节码相关的语句。

### 3.3.5 组件化

代码解耦、提高复用性，支持预部署。

路由优势：多模块分别开发，允许自定义拦截，提供IOC依赖注入容器，提供降级处理，支持InstantRun。

ARouter用懒加载的思想，以及分组思想，解决了大量路由表的注册问题，避免内存被占满。

ARouter在初始化时，若不使用插件，会需要扫描dex文件找到所有的Arouter包名的类，反射创建对应的对象，再执行它们的方法注册路由表。扫描会开启线程池，但对低端手机来说还是很耗时。

使用插件，会把扫描操作移到打包过程中，通过ASM编译时插入代码，减少了初始化耗时，也解决了因加固无法扫描dex文件的问题。

### 3.3.6 插件化

类加载器，BootClassLoader加载Android SDK的类，PathClassLoader默认加载依赖引入的类，DexClassLoader加载未安装的类。双亲委派机制，一是避免重复加载，二是安全，防止核心类被篡改。

DroidPlugin对于启动插件Activity，需要2个Hook点，使用反射和动态代理。先是hook AMS的startActivity方法，替换Intent对象，把原来的Intent存入新的Intent中，启动已经注册到Manifest的Activity；然后在LaunchActivity时，hook mH的Callback接口，获取消息的obj就可以把Intent替换回来。

RePlugin是hook ClassLoader实现的类替换。

### 3.3.7 热修复

AndFix使用DexFile加载补丁包的所有类，找出有特定注解的方法，从而找到已经修复的方法，再用反射获取有问题的原方法，最后通过Native的方法结构体来替换两者。由于不同Android版本、不同厂商有可能有不同的ArtMethod结构体，所以这种方式有兼容性问题。

Sophix解决AndFix的兼容性问题，是使用一个辅助类，用两个相邻方法的地址差测量出ArtMethod结构体的大小，然后将问题方法的结构体数据通过memcpy方法（将某段地址内容复制到特定地址段）替换成修复方法的结构体。

Robust是使用字节码插桩提前在所有方法中插入判断代码和成员变量，热修复时通过注解找到需要修改的方法，注入变量，用代理模式把问题方法做代理修复处理。

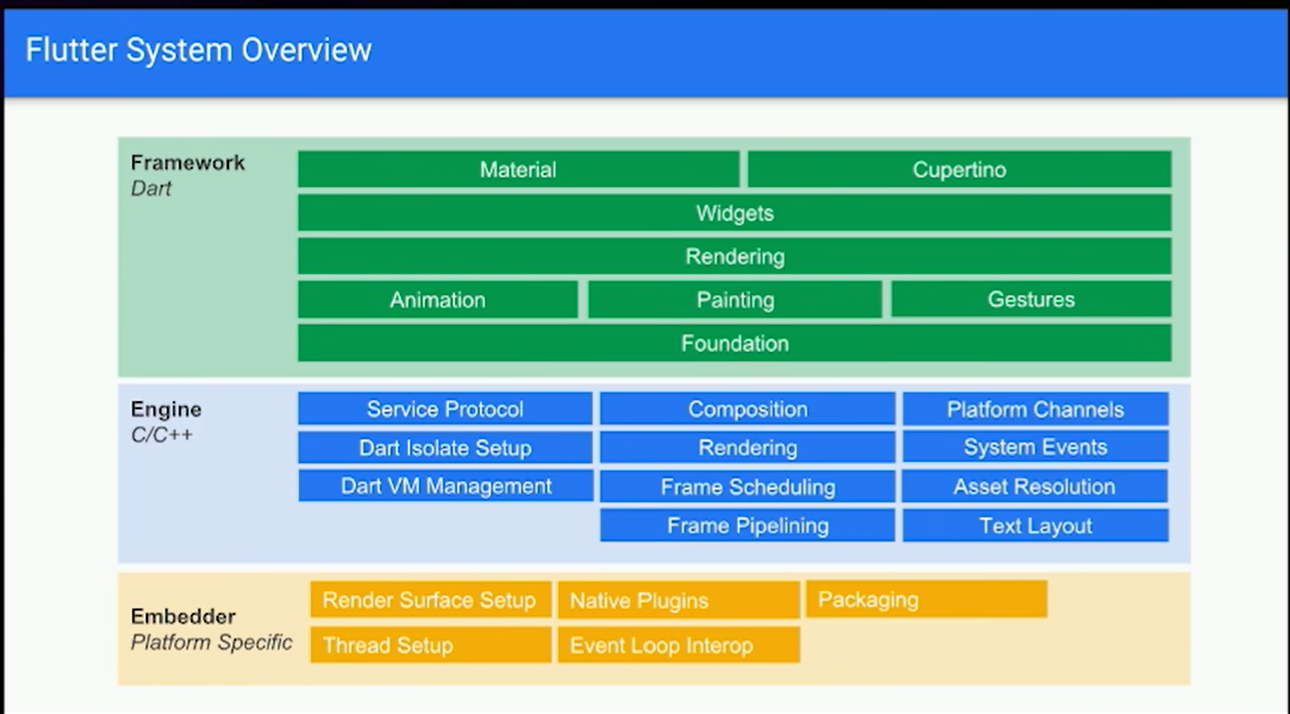
Tinker使用差分包，优化bsdiff工具dexdiff，专门处理dex的差分包。热修复使用InstantRun原理，加载合并后的dex文件，以及AssetManager加载新的资源路径。

## 3.4 跨平台

RN

WebView

### 3.4.3 Flutter



Flutter Engine线程模型由嵌入层管理，Engine需要嵌入层提供四个Task Runner：Platform Task Runner运行在平台上的主线程，对接Engine层，处理平台层的消息；UI Task Runner执行dart root isolate，处理渲染和Vsync信号，将Widget转换成布局树；GPU Task Runner执行GPU指令；IO Task Runner执行资源文件的加载读取等耗时操作。

StatefulWidget的生命周期，创建时先走构造方法，然后创建state；initState，

## 3.5 APM数据监控

### 3.5.1 ANR

超时阈值：

|  |  |  |
| --- | --- | --- |
| **类型** | **前台** | **后台** |
| Service | 20s | 200s |
| Broadcast | 10s | 60s |
| Provider | 10s | |
| Input | 5s | |

Service、Broadcast、Provider的ANR是通过埋“定时炸弹”实现的，时间到没有完成工作就会爆炸，触发ANR；input事件则是“埋雷”，再由下一个input事件去“扫雷”——如果发现上一个事件还没完成则检查是否超时，此时若是超时才会引爆触发ANR。所以只有一个input事件的时候，即使超时执行也不会ANR。

ANR的发生，是由AMS调用AnrHelpler的Anr方法，给进程发送SIGQUIT信号。如果是前台ANR则弹框提示。

Android M(6.0) 版本之后，应用侧不能直接通过监听 data/anr/trace 文件，监控是否发生 ANR。

## 3.6 性能优化

### 3.6.1 内存优化

OOM出现的原因：

* 申请了太大的内存，超出进程的内存限制。
* 内存碎片过多，导致没有足够可用的连续内存。
* 文件描述（句柄fd）过多。
* 线程过多。

内存监控，需要记录程序执行中的动态内存分配。

### 3.6.2 包体积优化

Dex优化，Facebook开源工具ReDex。常量折叠和常量传输优化，之后删除无用代码，可以在处理字节码class文件完成；跨Dex引用的优化，当一个Dex需要“使用”到另一个Dex中的类/方法/变量时，需要在本Dex中保存一份对应的类/方法/变量的id，可以统计每个类的所有引用，计算一个权重，优先放置优先级高的类进入Dex。

### 3.6.3 启动优化

启动任务的有向无环图，然后进行拓扑排序。

懒加载思想。

# 四、设计模式

## 4.1 创建型

### 4.1.1 单例模式

某个类全局只有一个实例，提供全局访问点。

### 4.1.2 简单工厂模式

一个工厂类根据传入的参量决定创建出哪一种产品类的实例。

### 4.1.3 工厂方法模式

抽象工厂方法，通过工厂子类提供对应的产品类实例。

### 4.1.4 抽象工厂模式

进一步抽象产品，一个工厂实例可以生产同一“品牌”的多种产品

### 4.1.5 建造者模式

封装一个复杂对象的创建过程，并可以按步骤构造。主要在创建需要传入多个非必要参数的类时使用。

### 4.1.6 原型模式

通过拷贝现有的实例来创建新的实例。

## 4.2 结构型

### 4.2.1 外观模式

对外提供一个统一的方法，来访问多个子系统中的众多接口。也称门面模式。

### 4.2.2 适配器模式

### 4.2.3 代理模式

使用代理完成功能的调用传递。

静态代理：代理者跟委托者实现同一个接口，代理者持有委托者对象，访问者通过代理者完成委托者的工作以及一些附加工作。代理者和委托者一一对应。

动态代理：通过Proxy.newProxyInstance实现一个代理工厂，传入接口类来生产对应的代理对象，调用InvocationHandler的invoke回调来具体实现实际逻辑。

### 4.2.4 装饰模式

在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）。

抽象构建与原有构建为继承继承，增加抽象装饰继承并持有抽象构建，具体装饰都继承抽象装饰来实现额外功能。

### 4.2.5 组合模式

### 4.2.6 桥接模式

### 4.2.7 享元模式

运用共享技术有效的支持大量细粒度的对象。如消息池，ViewHolder缓存都是享元模式的思想。

## 4.3 行为型

### 4.3.1 策略模式

### 4.3.2 观察者模式

观察者注册到被观察者，让被观察者持有观察者对象，在发生变化时回调观察者的方法。

### 4.3.3 访问者模式

### 4.3.4 责任链模式

### 4.3.5 命令模式

### 4.3.6 迭代器模式

### 4.3.7 解释器模式

### 4.3.8 中介者模式

### 4.3.9 备忘录模式

### 4.3.10 状态模式

### 4.3.11 模板方法模式

# 五、编程语言

## 5.1 kotlin

### 5.1.1 协程

轻量级的线程，异步编程写成同步。

协程上下文，可以通过withContext()来切换。

协程启动模式：DEFAULT，ATOMIC，LAZY，UNDISPATCHED。

协程挂起关键字suspend，实现原理是增加Continuation参数，自动生成代码处理异步和同步代码，通过状态机实现后续代码的持续执行。

### 5.1.2 lateinit与by lazy

lateinit(延迟初始化属性)

一般地，属性声明为非空类型必须在构造函数中初始化，然而这样经常不方便，例如：属性可以通过依赖注入来初始化，或者单元测试的setup方法中初始化，这种情况下，你不能在构造函数内提供一个非空初始器，但你仍然想在类体中引用该属性时避免空检查。为处理这种情况，我们可以使用lateinit。

该修饰只能用于类体中(不是在主构造函数中)声明的var属性，注意是var(可变属性)并且仅当该属性没有自定义getter或setter时，该属性必须是非空类型，并且不能是原生类型。

by lazy(惰性初始化)

惰性初始化是一种常见的模式，直到第一次访问该属性的时候，才根据需要创建对象的一部分，当初始化过程消耗大量资源并且在使用对象时并不总是需要数据时，这个非常有用。

## 5.2 Python

## 5.3 JavaScript

# 六、云游戏

云游戏系统，包含云端、客户端、服务端。功能上，包含调度系统、推拉流系统、操控系统和游戏管理系统。云端需要一个云端程序，采集屏幕的音视频信息，通过推拉流系统的后台服务将画面推给某个特定的客户端；客户端将操控信息，通过操控系统的后台服务，发送给云端程序，使得操控在云端生效；调度系统，主要是接收客户端的云游戏开启请求，选择合适的云机进行调度，唤起该云机的云端程序，首先通过游戏管理系统准备好游戏，然后建立推拉流系统和操控系统的连接，最后运行游戏。

## 6.1 云端

### 6.1.1 云机器

有PC云机和移动端云机。可以通过厂商合作购买。如华为云、阿里云、腾讯云、火山云、百度云和海马云等。

### 6.1.2 云端程序

这里主要涉及移动端云机的程序。可以是ROM内置，也可以只是内置一个系统应用。

云端程序要对接服务端的各个功能系统，包含以下一些功能：采集屏幕音视频，程序需要足够高的权限；推流数据上传，可以上传裸流或编码后的流，主要看推拉流系统的设计；接收及解析操控信息并使其生效，同样需要足够高的权限，多点触控需要看内核支持；对接游戏管理系统，包括游戏的安装，是否提前安装或者镜像挂载快速安装，游戏的启动和退出，游戏账号信息的同步等；另外就是对运行的云机做一些初始化配置，包括输入法配置、桌面配置、屏幕旋转和分辨率配置、音量帧率GPU模式等。

# 七、系统相关

7.1 Android命令行

Android在Framework层就提供了很多Unix命令行工具，即bin执行工具，可查看/system/bin目录了解本机支持的执行工具。它主要是通过binder获取对应服务，通过IBinder::shellCommand调用对应Java层服务的onShellCommand。

Cmd执行程序可参考源码：framework/native/cmds/cmd目录下的代码。

cmd activity即am命令，对应ActivityManagerService的onShellCommand，主要由ActivityManagerShellCommand负责处理具体命令。

