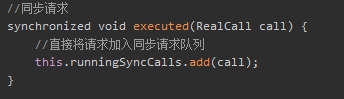
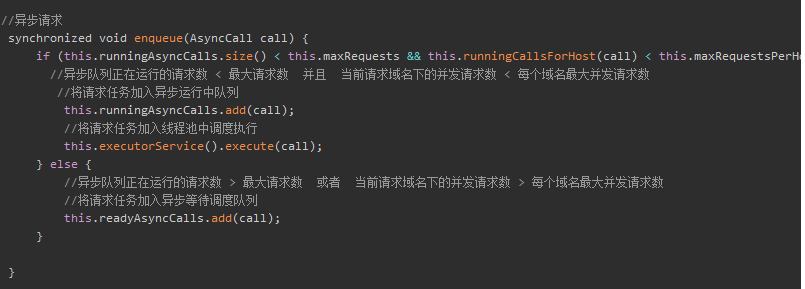
# Android源码方面

1. **Okhttp源码：**
2. **Dispatcher**

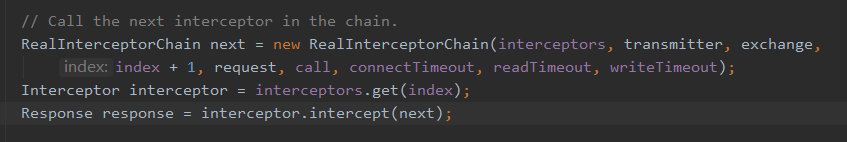
**同步请求：**dispatcher对应同步请求而言，只是将请求放入同步队列，便于后续的取消、完成等操作



**异步请求：**dispatcher对异步请求而言，需要等待队列和运行队列来完成异步调度，在同一域名并发数不超额且总请求数不超额的情况下，调度运行队列，将请求放入线程池运行。

1. **拦截器（Interceptor）**





构建责任链之后，RealInterceptorChain中再创建一个新的责任链，然后责任链的索引加1，再调用intercept()方法，因为拦截器中依然有proceed()，因此，拦截器会一直向下传；执行完之后，再从下到上返回Response。

* 1. **重试与重定向拦截器（RetryAndFollowUpInterceptor）**





（1）发生 Route 或 IO 异常，则进行重试；

（2）如果出现以下情况则不会重试：

·客户端配置了出错不再重试

·无法再次发送 request body

·发生 ProtocolException（协议异常）、InterruptedIOException（中断异常）、SSLHandshakeException（SSL握手异常）、SSLPeerUnverifiedException（SSL握手未授权异常）中的任意一个异常

·没有更多的路线可供尝试

1. 检查是否需要重定向，如果不需要则返回之前的 response，需要则进行重定向，继续进行循环。

**3.2 桥拦截器（BridgeInterceptor）**

桥链拦截器的主要职责是：

请求HTTP前，重新修正请求内容体的编码、压缩方式等

请求HTTP前，读取本次请求路径对应的cookie并设置到HTTP请求头部的cookie字段

请求HTTP后，得到响应对象，读取响应对象头部的的cookie列表并进行缓存

请求HTTP后，得到响应对象，如果响应体内容是经过压缩的，则进行解压缩处理

**3.3 缓存拦截器（CacheInterceptor）**

缓存拦截器的主要职责，即整体流程：

a、从本地读取缓存

b、经过缓存机制类按HTTP缓存机制校验之后，要么缓存过期、要么缓存失效、要么需要再次发起网络请求验证本地缓存是否有效等

c、如果本地缓存有效，且不需要网络再次验证，则直接返回本地缓存

d、如果本地缓存失效或者需要再次发起网络请求验证，则发起网络请求，对响应做如下处理：  
d1：如果本地缓存还有效（即网络上资源没有更新），则使用本地缓存重新构造响应，并更新到本地  
d2：如果没有本地缓存或者网络上资源更新，而且可以缓存，则使用网络响应重新构建响应，并保存到本地  
d3：如果请求方法不支持缓存，则从本地中删除缓存

**缓存存取的实现**常用的策略就是LRU算法（Last Recently Used，最近最少使用算法），LRU优先淘汰最近最久没有使用的，而LRU的数据结构一般采用LinkedHashMap。

**3.4 连接拦截器（ConnectInterceptor）**

主要用来打开与目标服务器的连接，然后继续执行下一个拦截器。

findConnection()方法流程：

（1）判断当前连接是否可用，如果不可用，尝试从连接池中获取可用连接；如果获取的连接不可用，切换路由再次获取可用连接；再不可用，只能重新创建新的连接；

（2）进行 TCP 和 TLS 握手；

（3）最后将新创建的连接放进连接池中；

result.connect()方法流程：

1. okhttp 底层是通过 socket 进行连接的；

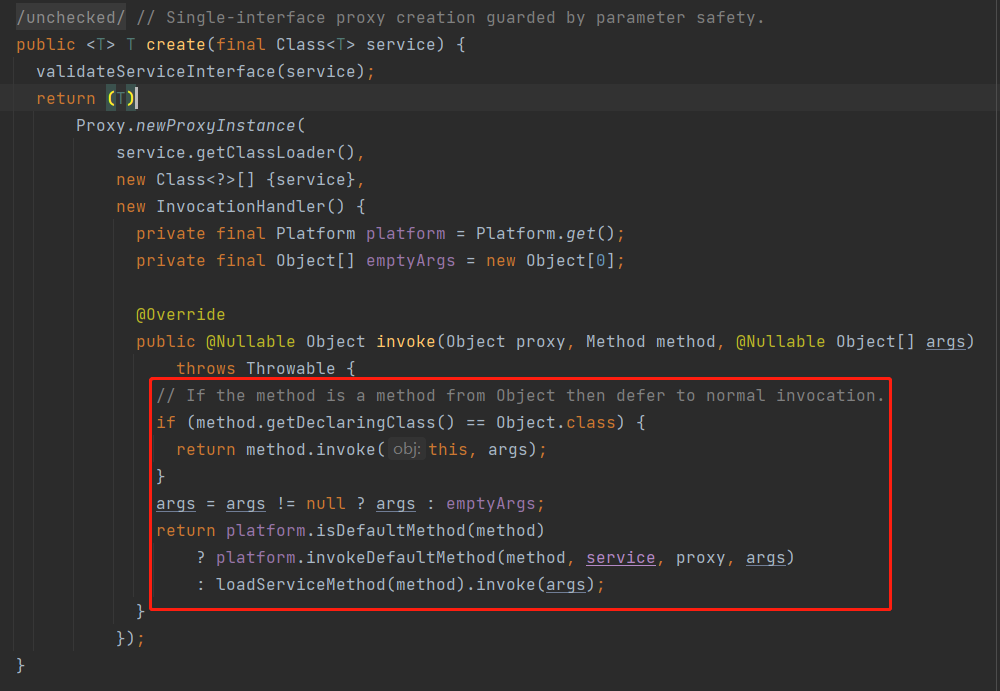
**3.5 服务器请求拦截器（CallServerInterceptor）**

1. **Retrofit源码**

retrofit.build() 配置参数，在create()中使用；

Retrofit.create()：

动态代理：



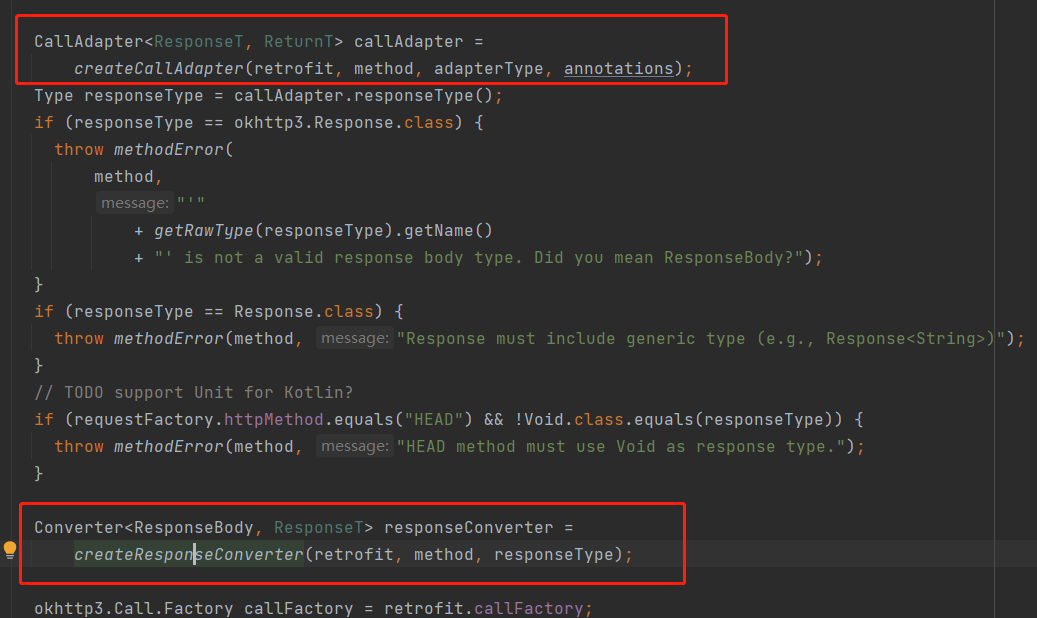
利用动态代理，当调用service.xxx()时，就会执行invoke方法里面的逻辑。

动态代理：



流程：

1. 判断平台；
2. 通过HttpServiceMethod的parseAnnotitions解析请求参数；
3. 执行CallAdatper和Converter；



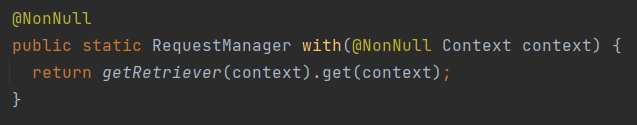
1. 通过RxJava3CallAdapterFactory执行OkHttp请求（Retrofit本身不发起网络请求）

总结：

* 使用构建模式创建Retrofit实例对象
* 使用动态代理模式，通过Retrofit实例对象的create方法动态创建网络请求接口的代理类
* 使用代理类调用方法发起网络请求时，会通过CallAdapterFactory创建的CallAdapter对象的adapt方法调用OkHttpCall的enqueue方法发起网络请求
* OkHttpCall的enqueue方法先根据注解、请求参数等信息构建出OkHttpClient的网络请求Call对象，接着使用Call对象发起网络请求
* 网络请求回来时，使用ServiceMethod的toResponse方法中调用响应内容适配器的具体实现类的convert发放进行解析

Retrofit只是对网络请求进行封装的一个框架，它本身并不发起网络请求，为的是方便我们使用、优化我们网络请求的代码、对网络请求进行扩展（可以自定义请求适配器、响应内容解析器等）

1. **Glide源码**
2. **Glide.with(context)**



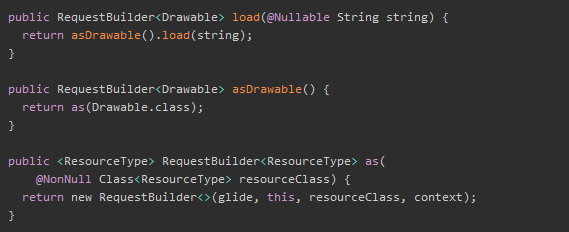


Glide.get(context)实现glide的初始化；

getRequestManagerRetriever()是用来感知组件的生命周期的；

getRetriever(context).get(context)根据不同生命周期对象创建RequestManagerFragment,包括context,activity,fragment,view，用于感知各大组件的生命周期。

1. **Glide.load(url)**



主要是创建一个RequestBuilder，Glide里面的load,error,placeholder,fitCenter等都是RequestBuilder里面的方法。

1. **RequestBuilder**
2. load、transition、apply、error、listener都为保存配置的数据，用于调用into(imageView)后生效；
3. Into(imageView)：

处理图片的加载，包括缓存策略

总结：

· 运行期通过GlideApp的with作为入口返回RequestManager

· 使用RequestManager创建一个图片加载请求构造器RequestBuilder

· 通过RequestBuilder设置图片加载和解析时的一些配置信息：比如playholder、error、apply、transform、listener等

· 通过RequestBuilder的into构建具体图片加载请求对象默认是SingleRequest

· SingleRequest等待ImageView测量得到具体大小之后，在通过图片加载引擎Engine来加载图片

· Engine图片加载引擎根据四级缓存策略：优先从获取资源缓存列表加载；接着从缓存列表加载；再从磁盘缓存加载；最后是从网络加载

· 最后加载完成先回调RequestListener通知图片加载完成；再回调TargetListener通知图片加载完成；最后才回调Target(ImageView的包裹类)将图片设置给ImageView

1. **Glide的生命周期绑定**

· 如果是在子线程，这Glide使用的RequestManager与Application一样的生命周期

· 如果是fragment、activity,则使用FragmentManagerFragment与RequestManager进行关联，通过FragmentManagerFragment的生命周期变化来调度RequestManager对图片加载请求Request采取暂停、重新开始、停止等操作。  
问题：Glide的get操作有哪些优化点？

· 在UI线程中调用，可以避免RequestManager生命周期与Application的一制

· get尽量传递fragment或者activity，这样可以减少通过view找到具体的fragment或者activity的步骤

1. ****ImageViewTarget****

· 通过ViewTreeObserver实现View的大小测量，测量到大小之后回到监听者的onSizeReady告知view的大小已经测量ok

· 通过监听View与Window的绑定关系发起加载图片的请求或者取消加载图片

· 设置加载中的显示图片

· 设置加载失败时显示的图片

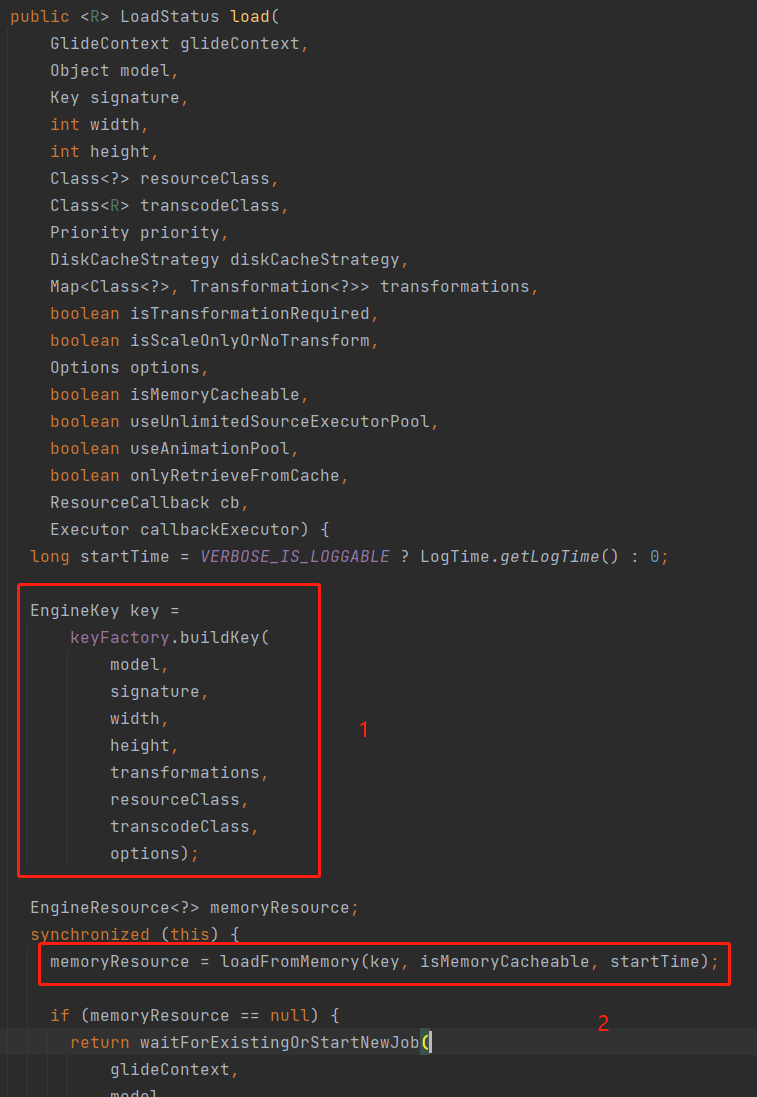
· 设置加载成功时的图片、动效

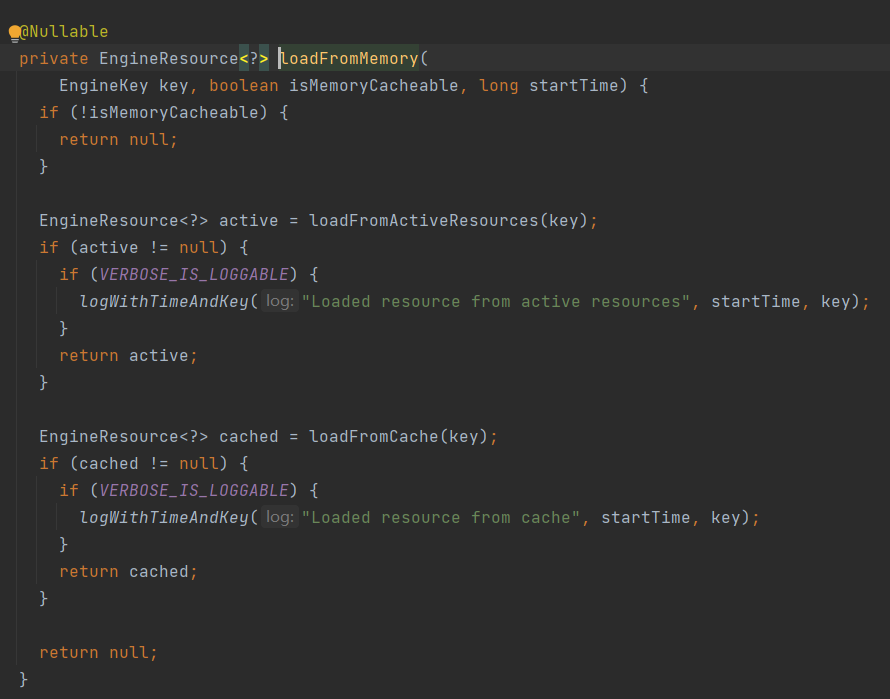
1. **Glide缓存机制**

glide缓存机制分为3级缓存机制，其顺序为WeakReference->LruCache->DiskLrucache->网络。

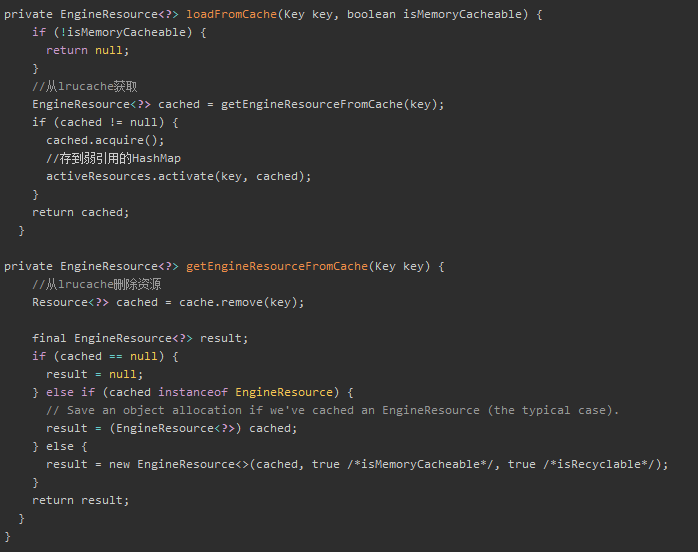
****内存缓存**的主要作用是防止应用重复将图片数据读取到内存当中；而**硬盘缓存**的主要作用是防止应用重复从网络或其他地方下载和读取数据。**

**（1）内存缓存（弱引用+LruCache）**





1. 获取key
2. 判断是否有活跃缓存，没有则去LruCache读取缓存；
3. 通过弱引用的hashmap来存储资源，Key是缓存key，ResourceWeakReference代表资源，它继承WeakReference。首先从弱引用的map获取图片资源，然后通过弱引用的get()方法获取最终需要的对象；



（4）**通过lrucache获取图片资源，如果获取到的话就会从LruCache中删除这张图片，然后会调用**acquire()**方法和**activate()**方法,其中**activate()**是把取到的数据会存到弱引用中，说白了就是把图片从LruCache转移到弱引用**。

这个acquired变量是用来记录图片被引用的次数，调用acquire()方法会让变量加1，调用release()方法会让变量减1。当调用loadFromActiveResources()、loadFromCache()、EngineJob#handleResultOnMainThread()获取图片的时候都会执行acquire()方法；当暂停请求或者加载完毕或者清除资源时会调用release()方法。  
  
注意：从弱引用取缓存，拿到的话，引用计数+1；从LruCache中拿缓存，拿到的话，引用计数也是+1，同时把LruCache缓存转移到弱应用缓存池中；从EngineJob去加载图片，拿到的话，引用计数也是+1，会把图片放到弱引用。反过来，一旦没有地方正在使用这个资源，就会将其从弱引用中转移到LruCache缓存池中。这也说明了正在使用中的图片使用弱引用来进行缓存，暂时不用的图片使用LruCache来进行缓存的功能。

1. **磁盘缓存**

**Glide5大磁盘缓存策略**  
DiskCacheStrategy.DATA: 只缓存原始图片；  
DiskCacheStrategy.RESOURCE:只缓存转换过后的图片；  
DiskCacheStrategy.ALL:既缓存原始图片，也缓存转换过后的图片；对于远程图片，缓存 DATA和 RESOURCE；对于本地图片，只缓存 RESOURCE；  
DiskCacheStrategy.NONE：不缓存任何内容；  
DiskCacheStrategy.AUTOMATIC：默认策略，尝试对本地和远程图片使用最佳的策略。当下载网络图片时，使用DATA；对于本地图片，使用RESOURCE

磁盘缓存就是通过DiskLruCache实现的，根据缓存策略的不同会获取到不同类型的缓存图片。它的逻辑是：先从转换后的缓存中取；没有的话再从原始的（没有转换过的）缓存中拿数据；再没有的话就从网络加载图片数据，获取到数据之后，**再依次缓存到磁盘和弱引用**。

总结：

Glide缓存分为弱引用+ LruCache+ DiskLruCache，其中读取数据的顺序是：弱引用 > LruCache > DiskLruCache>网络；写入缓存的顺序是：网络 --> DiskLruCache--> 弱引用-->LruCache

内存缓存分为弱引用的和 LruCache ，其中正在使用的图片使用弱引用缓存，暂时不使用的图片用 LruCache缓存，这一点是通过 图片引用计数器（acquired变量）来实现的，详情可以看内存缓存的小结。

磁盘缓存就是通过DiskLruCache实现的，根据缓存策略的不同会获取到不同类型的缓存图片。它的逻辑是：先从转换后的缓存中取；没有的话再从原始的（没有转换过的）缓存中拿数据；再没有的话就从网络加载图片数据，获取到数据之后，再依次缓存到磁盘和弱引用。

1. **LruCache原理**

LruCache 是个泛型类，主要原理是：把最近使用的对象用强引用存储在 LinkedHashMap 中，当缓存满时，把最近最少使用的对象从内存中移除，并提供 get/put 方法完成缓存的获取和添加LruCache 是线程安全的，因为使用了 synchronized 关键字。

当调用 put()方法，将元素加到链表头，如果链表中没有该元素，大小不变，如果没有，需调用 trimToSize 方法判断是否超过最大缓存量，trimToSize()方法中有一个 while(true)死循环，如果缓存大小大于最大的缓存值,会不断删除 LinkedHashMap 中队尾的元素，即最少访问的，直到缓存大小小于最大缓存值。当调用 LruCache 的 get 方法时，LinkedHashMap 会调用recordAccess 方法将此元素加到链表头部

1. **其它**

对于一般App来说，Glide完全够用，而对于图片需求比较大的App，为了防止加载大量图片导致OOM，Fresco 会更合适一些。并不是说用Glide会导致OOM，Glide默认用的内存缓存是LruCache，内存不会一直往上涨。

假如让你自己写个图片加载框架，你会考虑哪些问题：

· 异步加载：线程池

· 切换线程：Handler，没有争议吧

· 缓存：LruCache、DiskLruCache

· 防止OOM：软引用、LruCache、图片压缩、Bitmap像素存储位置

· 内存泄露：注意ImageView的正确引用，生命周期管理

1. **LeakCanary解析**

AppWatcher.manualInstall()在主进程中被自动调用

AppWatcherInstall继承自ContentProvider并在AndroidManifest.xml中注册，利用ContentProvider无需显示初始化

内存泄漏检测主要过程：



Activity内存泄漏检测过程：

(1)注册监听Activity生命周期onDestroy事件

(2)在Activity onDestroy事件回调中创建KeyedWeakReference对象，并关联ReferenceQueue

(3)延时5秒检查目标对象是否回收

(4)未回收则开启服务，dump heap获取内存快照hprof文件

(5)解析hprof文件根据KeyedWeakReference类型过滤找到内存泄漏对象

(6)计算对象到GC roots的最短路径，并合并所有最短路径为一棵树

(7)输出分析结果，并根据分析结果展示到可视化页面

1. **Rxjava源码解析**
2. **线程调度器（Schedulers）**

Schedulers.io()

用于IO密集型任务，如读写SD卡文件，查询数据库，访问网络等;

具有线程缓存机制，默认是一个CacheThreadScheduler;

Schedulers.newThread()

为每一个任务创建一个新线程;

不具有线程缓存机制，虽然使用Schedulers.io的地方，都可以使用Schedulers.newThread，但是，Schedulers.newThread的效率没有Schedulers.io高;

Schedulers.computation()

用于CPU 密集型计算任务，即不会被 I/O 等操作限制性能的耗时操作，例如xml,json文件的解析，Bitmap图片的压缩取样等，具有固定的线程池，大小为CPU的核数。不可以用于I/O操作，因为I/O操作的等待时间会浪费CPU。

Schedulers.trampoline()

在当前线程立即执行任务，如果当前线程有任务在执行，则会将其暂停，等插入进来的任务执行完之后，再将未完成的任务接着执行;

Schedulers.single()

拥有一个线程单例，所有的任务都在这一个线程中执行，当此线程中有任务执行时，其他任务将会按照先进先出的顺序依次执行;

Scheduler.from(executor)

指定一个线程调度器，由此调度器来控制任务的执行策略

1. **背压**

背压是指在异步场景中，被观察者发送事件速度远快于观察者的处理速度的情况下，一种告诉上游的被观察者降低发送速度的策略;

支持背压的被观察者为Flowable；

Android中很少用到，除非在线视频流，直播等场景，当画面卡顿已取得的数据失效了，需要抛弃等；

**背压策略模式**

BackpressureStrategy.MISSING

在此策略下，通过Create方法创建的Flowable相当于没有指定背压策略，不会对通过onNext发射的数据做缓存或丢弃处理，需要下游通过背压操作符处理

BackpressureStrategy.ERROR:

在此策略下，如果放入Flowable的异步缓存池中的数据超限了，则会抛出MissingBackpressureException异常;

BackpressureStrategy.BUFFER:

内部维护了一个缓存池SpscLinkedArrayQueue，其大小不限，此策略下，如果Flowable默认的异步缓存池满了，会通过此缓存池暂存数据，它与Observable的异步缓存池一样，可以无限制向里添加数据，不会抛出MissingBackpressureException异常，但会导致OOM;

当缓存区大小存满（默认缓存区大小 = 128）、被观察者仍然继续发送下1个事件时，将缓存区大小设置成无限大，被观察者可无限发送事件 观察者，但实际上是存放在缓存区，但要注意内存情况，防止出现OOM;

BackpressureStrategy.DROP

在此策略下，如果Flowable的异步缓存池满了，会丢掉上游发送的数据;

BackpressureStrategy.LATEST

与Drop策略一样，如果缓存池满了，会丢掉将要放入缓存池中的数据，不同的是，不管缓存池的状态如何，LATEST都会将最后一条数据强行放入缓存池中，来保证观察者在接收到完成通知之前，能够接收到Flowable最新发射的一条数据;

即如果发送了150个事件，缓存区里会保存129个事件（第1-第128 + 第150事件）;

1. **subscribeOn**

大概流程：create-->subsrcibeOn-->observeOn-->subscribe

从subscribe-->

ObservableObserveOn的subscribeActual（调用上游的Obser）-->

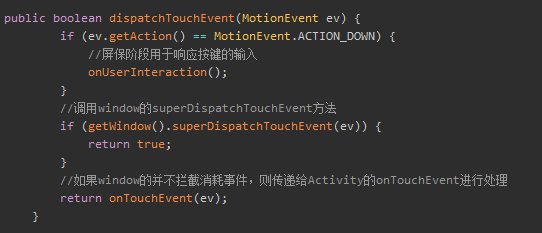
ObservableObserveOn中Observer的onSubscribe（Observer里的一个回调）-->

通过scheduler.scheduleDirect来达到线程切换

ObservableSubscribeOn的subscribe（也就是create里面的object）-->

# Android方面

1. **触摸事件分发机制**
2. Activity事件分发



Activity dispatchTouchEvent的核心思想是：将事件分发给window，如果window拦截消耗事件，则完成事件的分发；如果window不拦截消耗事件，则将事件传递给Activity的onTouchEvent进行处理。

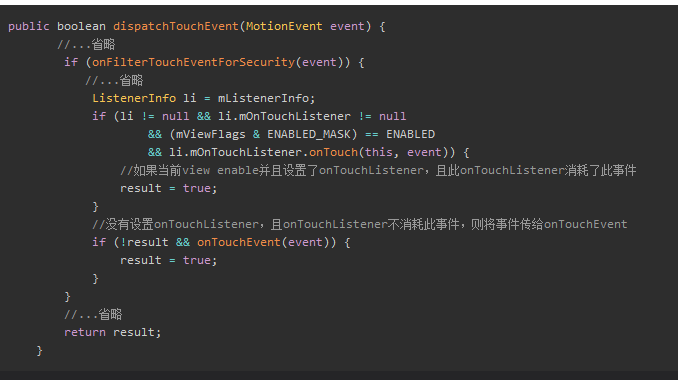
1. ViewGroup事件分发



如果没有子View，则调用ViewGroup的父类View来实现dispatchTouchEvent。

ViewGroup进行进行事件分发时，其核心思想是：  
1、 调用onInterceptTouchEvent判断自己是否要拦截消耗此事件  
2、 如果自己不拦截消耗事件，则遍历子view列表，根据事件坐标找到对应的子view，然后将事件分发给此view，如果此子view是ViewGroup，那么又走ViewGroup的dispatchTouchEvent逻辑；如果此子view是View，那么进入第四阶段  
3、如果自己拦截消耗事件那么调用父类的dispatchTouchEvent，其父类是View，而View的事件分发dispathcTouchEvent请看第四阶段

1. View事件分发



View的dispatchTouchEvent优先将事件传递给onTouchListener，如果onTouchListener消耗此事件，则直接放回true；如果没有设置onTouchListener或者onTouchListener没有消耗此事件，则将事件传递给onTouchEvent

onTouchEvent的核心工作是：  
1、先判断view是否enable，如果不enable则不响应事件，并返回clickable的值  
2、如果设置了touch的代理，并且touch代理消耗了此事件则返回true代表消耗了事件  
3、如果view不可点击，则返回false，代表不消耗此事件  
4、如果view可点击，则返回true，代表消耗此事件。如果设置了长按事件，则在按下事件中延迟500ms响应长按事件；响应点击事件是在松开事件时响应的

1. 如何拦截触摸事件？怎么不响应触摸事件？

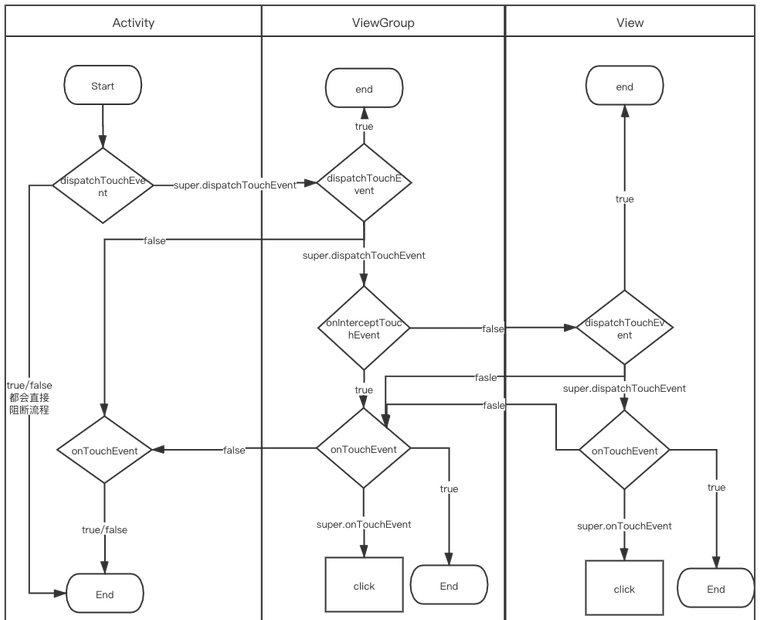
要响应事件，根据View的事件分发流程可以有几种解决方案：  
a、父控件拦截事件，将事件传递给view进行处理  
b、自己拦截事件：

1.可以通过dispatchTouchEvent直接返回true，事件处理再dispatchTouchEvent中处理；

2.onInterceptTouchEvent返回true，并在onTouch或者onTouchEvent中消耗此事件，并返回true

3.不响应最简单的是clickable为false或者enable为false、或者不可见；

1. 嵌套的view，父级要求响应垂直滑动，子级要求响应水平滑动，如何实现？
2. 父级控件拦截事件，滑动时判断水平滚动距离大于垂直滑动距离，则将事件传递给子view响应水平滑动；否则父级控件自己响应垂直滑动  
   b、子级在onInterceptTouchEvent中判断水平滚动距离大于垂直滑动距离，则拦截消耗此事件
3. 流程图



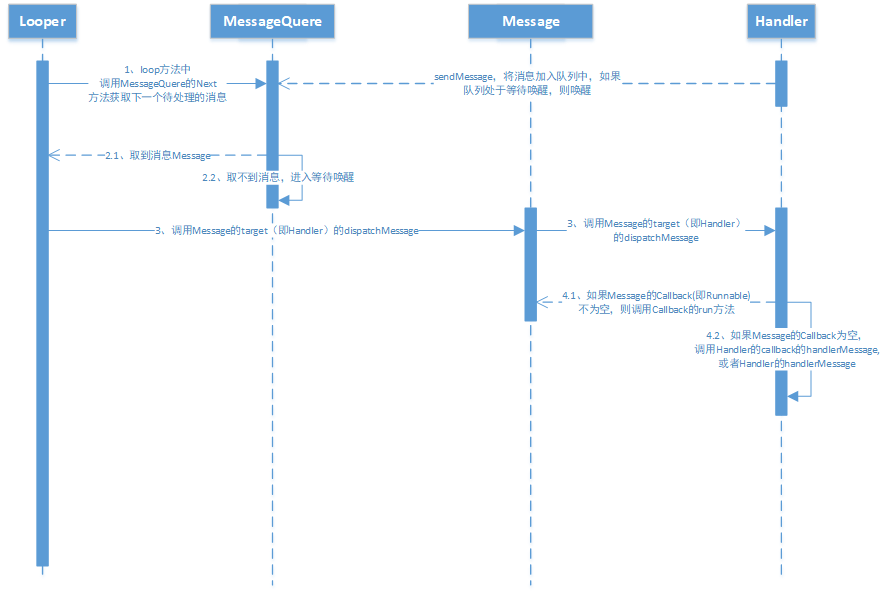
1. **Handler机制**

Handler机制内部有三大关键角色：Handler，Looper，MessageQueue。其中MessageQueue是Looper内部的一个对象，MessageQueue和Looper每个线程有且只有一个，而Handler是可以有很多个的。他们的工作流程是：

（1）用户使用线程的Looper构建Handler之后，通过Handler的send和post方法发送消息

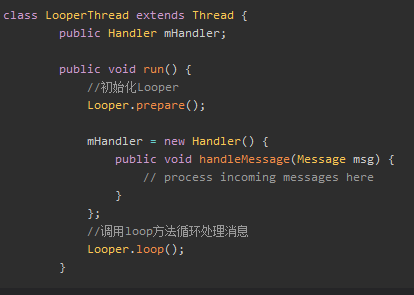
（2）消息会加入到MessageQueue中，等待Looper获取处理

（3）Looper会不断地从MessageQueue中获取Message然后交付给对应的Handler处理



1. **Lopper**

子线程实现Looper代码模板：



主线程不需要Looper.prepare()是因为Looper.prepare()已经在底层中实现了。

Lopper.loop()



a、死循环不停的从消息队列中取消息不会很耗性能吗？  
b、如果消息队列中已经没有待处理的消息，loop方法不就退出了吗，那么后面线程再次有消息发送时怎么办？  
c、为什么需要将回收的消息类放入池中并复用？  
解答：  
a、b：不会很耗性能，如果消息队列为空时，线程会挂起释放CPU时间片，等到有新消息时才会唤醒继续执行（MessageQueue源码分析会知道原因）  
c：消息收发频繁时，就可以缓解创建消息类的内存和CPU开销，可以提升性能

1. **Message**

Message的作用就是承载消息，他的内部有很多的属性用于给用户赋值。同时Message本身也是一个链表结构，无论是在MessageQueue还是在Message内部的回收机制，都是使用这个结构来形成链表。同时官方建议不要直接初始化Message，而是通过Message.obtain()方法来获取一个Message循环利用。

1. **MessageQueue**

每个线程都有且只有一个MessageQueue，他是一个用于承载消息的队列，内部使用链表作为数据结构，所以待处理的消息都会在这里排队。

Message还涉及到一个关键概念：线程休眠。当MessageQueue中没有消息或者都在等待中，则会将线程休眠，让出cpu资源，提高cpu的利用效率。进入休眠后，如果需要继续执行代码则需要将线程唤醒。

1. **Handler**

Handler是作为整个消息机制的消息发起者与处理者，消息在不同的线程通过Handler发送到目标线程的MessageQueue中，然后目标线程的Looper再调用Handler的dispatchMessage方法来处理消息。

问题：

**1. 为什么Looper中的死循环不会阻塞主线程？**

**卡顿**是因为出于某种原因导致的绘制时间过长，而**ANR**的原因是对用户的操作响应超时。  
而Looper中的死循环是为了读取消息，要知道Android应用本质上是消息驱动的，不管是卡顿还是ANR，本质上都是对应Handler或者Handler.Callback的handleMessage()处理消息方法的执行时间太长；而Looper中的死循环是在体系之外的，不在某个Handler的handleMessage()方法体之中，自然也就不会引起卡顿和ANR了。

1. **Handler只能在主线程创建吗？如果不是，那Handler可以在任意线程创建吗？**

Handler可以在任何线程创建，在线程中调用Looper.prepare和Looper.loop即可

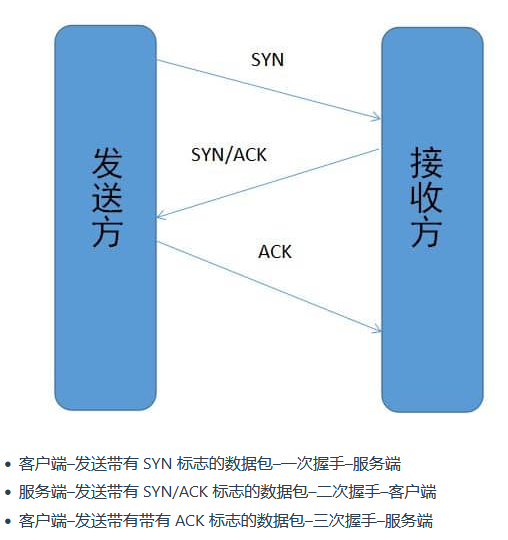
1. **UI只能在主线程改吗？**

不对，严格来说，只能在创建 view 的这个线程，才能更新 ui，如果这个 view 在子线程创建的，那也可以在该子线程更新。

另外，在Activity的onCreate新建线程设置UI也是可以的，原因是在onCreate方法里面调用设置UI的时候，并没有进行实际的绘制流程，因为ViewRootImpl还没有被设置，等到进行真正执行绘制流程的时候，才被渲染出来。

1. **http相关**

**（1）三次握手**



第一次握手：客户端将标志位SYN置为1，随机生成一个值seq=1，接着将数据表发给服务端，此时客户端进入SYN\_SENT（请求建立已发送）状态，等待服务端确认

第二次握手：服务端收到数据包后，有SYN=1知道客户端请求建立连接，服务端将标志位SYN和ACK都置为1，ack=J+1，随机生成一个值seq=K，并将数据包发送给客户端以确认连接请求，接着服务端进入SYN\_RCVD（建立连接已收到）状态

第三次握手：客户端收到确认包后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给服务端，服务端收到之后检测ack是否为K+1，ACK是否为1，如果正确则连接建立成功，客户端和服务端都进入ESTABLISHED状态，完成三次握手，随后客户端和服务端之间可以开始传输数据了。

**（2）TCP四次挥手**



第一次挥手：客户端将FIN置为1，随机生成一个值seq=J，将数据包发送给服务端，用来关闭客户端到服务端的数据发送，此时客户端进入FIN\_WAIT\_1（等待关闭）状态

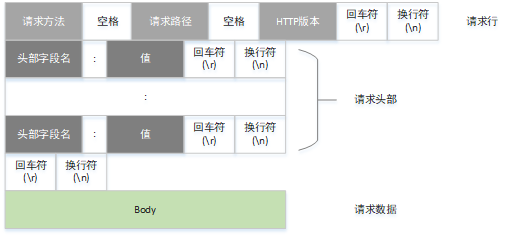
第二次挥手：服务端收到数据包，有FIN=1知道客户端请求断开连接，将ACK置为1，ack=J+1，接着数据包发送回给客户端，表示断开连接的请求已收到，此时服务端进入CLOSE\_WAIT（等待关闭）状态。客户端收到确认包之后，检测ACK是否为1，ack是否为J+1，检查正确之后表明服务端已经收到断开连接的请求，客户端随后进入FIN\_WAIT\_2状态

第三次挥手：服务端将FIN置为1，随机生成一个值seq=K，将数据包发送给客户端，用来关闭服务端到客户端的数据发送，此时服务端进入LAST\_ACK（等待最后确认）状态

第四次挥手：客户端收到服务端的FIN后，表明服务端请求断开数据发送，接着将ACK置为1，ack=K+1并把数据包发送给服务端，此时客户端进入TIME\_WAIT（等待2MSL之后进入CLOSED状态）状态。服务端收到FIN的确认包之后，检测ACK是否为1，ack是否为K+1，如果正确服务端进入CLOSED状态，关闭TCP连接

**1、为什么连接是三次握手，而断开连接是4次挥手？**  
答：这是因为服务端在LISTEN(监听)状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发给客户端，而关闭连接时，当收到客户端的FIN报文时，仅仅表示客户端不在发送数据了但还能接受数据，服务端也未必把全部数据都发送给对方了，所以服务端可以可以立即关闭也可以发送一些数据给客户端后，再发送FIN报文给客户端来表示同意现在关闭连接，所以服务端ACK和FIN一般会分开发送

1. **http请求协议报文格式**



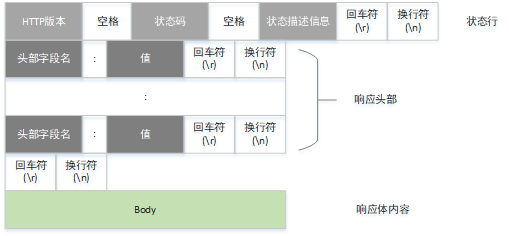
响应协议报文格式分四大部分：  
**1.状态行：**

HTTP版本：如：HTTP/1.1、HTTP/2.0等，表明当前响应内容使用的是哪个HTTP版本，一般与请求时的版本一致

状态码：表明响应的状态：成功（200）、错误等

状态描述信息：对应状态码的描述信息  
**2.响应头部：**  
响应头部是一系列key:value\r\n，具体的将在下面的内容进行讲解  
**3.空行：**  
响应头部后面必须跟一个空行(\r\n)  
**4.响应内容：**  
响应请求的内容，响应成功下，该部分是服务器返回请求需要的数据

1. **http响应协议报文格式**



响应协议报文格式分四大部分：  
**1.状态行：**

HTTP版本：如：HTTP/1.1、HTTP/2.0等，表明当前响应内容使用的是哪个HTTP版本，一般与请求时的版本一致

状态码：表明响应的状态：成功（200）、错误等

状态描述信息：对应状态码的描述信息  
**2.响应头部：**  
响应头部是一系列key:value\r\n，具体的将在下面的内容进行讲解  
**3.空行：**  
响应头部后面必须跟一个空行(\r\n)  
**4.响应内容：**  
响应请求的内容，响应成功下，该部分是服务器返回请求需要的数据

1. **TCP, UDP 协议的区别**



UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 却是一种最有效的工作方式（一般用于即时通信），比如： QQ 语音、 QQ 视频 、直播等等

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。

TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

1. **Activity相关**
2. 启动模式

standard：

系统默认的模式，每次都会创建新的实例；

singleTop：

如果activity的实例已存在于当前任务的顶部，则系统通过调用其onNewIntent()，否则会创建新实例

singleTask

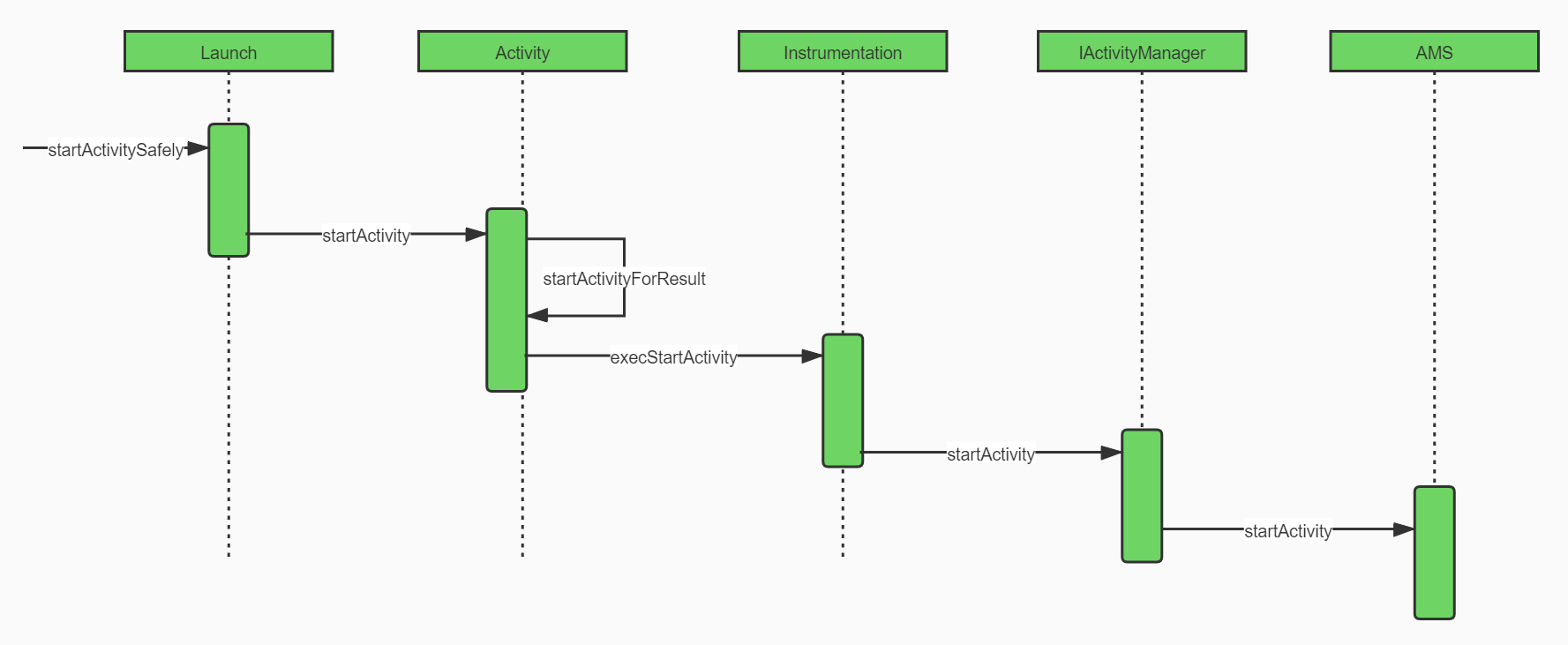
如果 activity 的实例已存在于单独的任务中，则调用其 onNewIntent() 方法，其上面的实例会被移除栈。一次只能存在一个 activity 实例

singleInstance

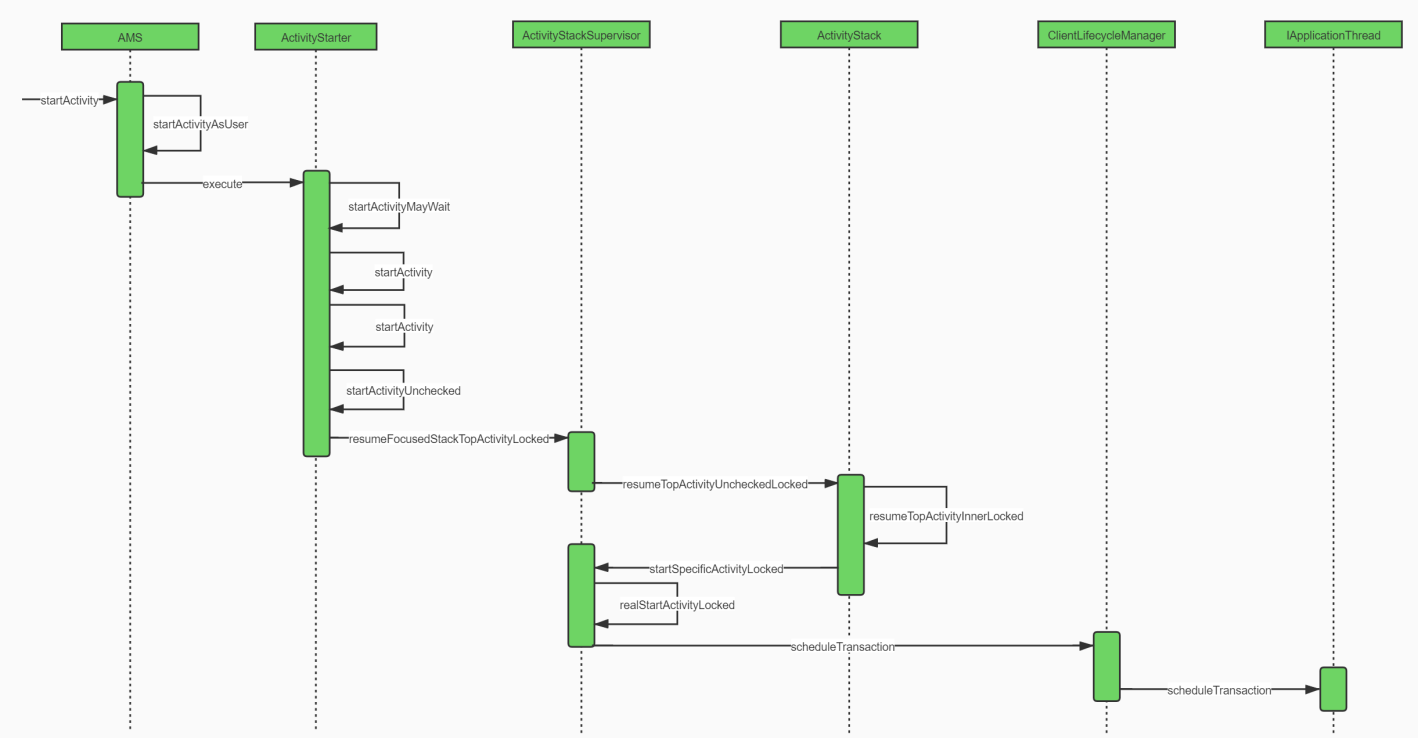
系统会为它创建一个新的任务栈，然后独自在这个新的任务栈中，由于栈内复用的特性，后续的请求均不会创建新的activity，除非这个独特的任务栈被系统销毁了

1. Activity启动流程

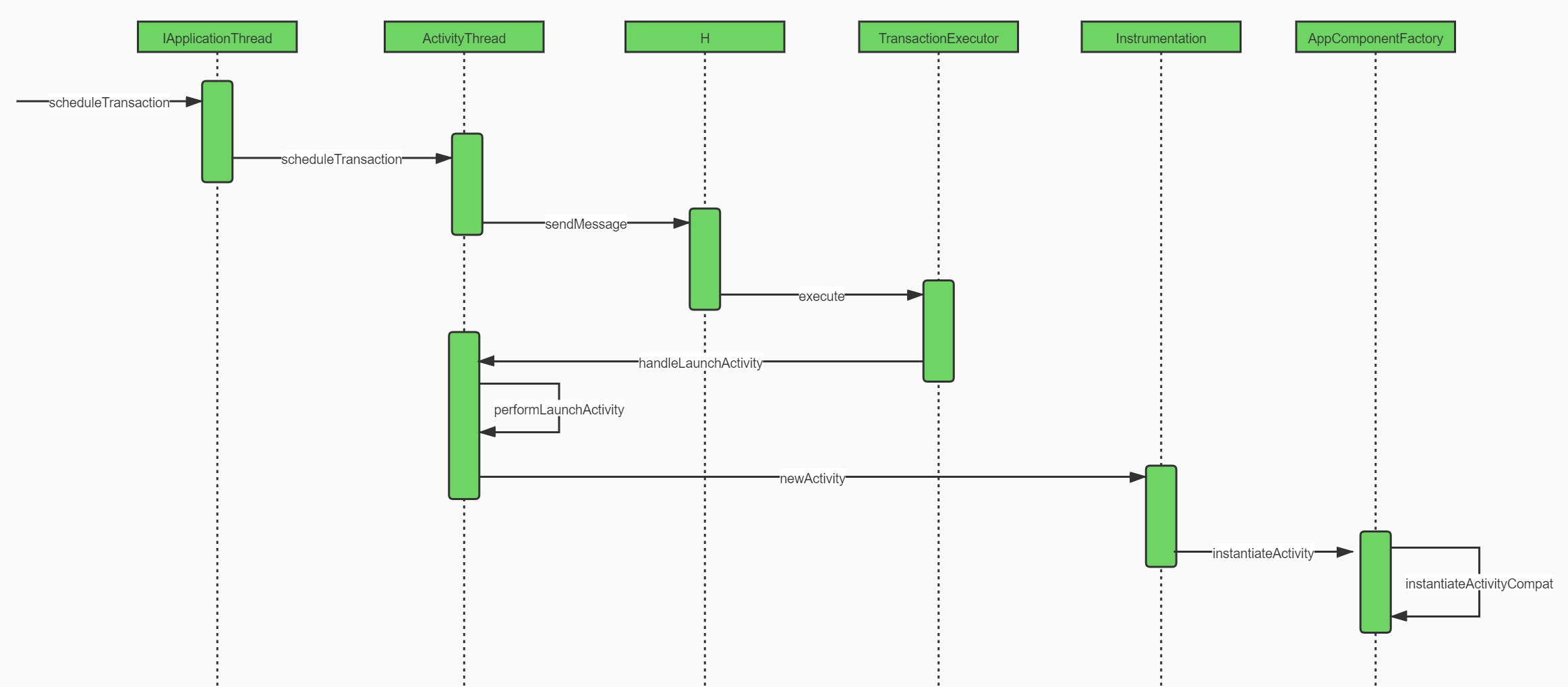
Activity请求AMS的过程



AMS处理请求的过程



ActivityThread创建Activity的过程

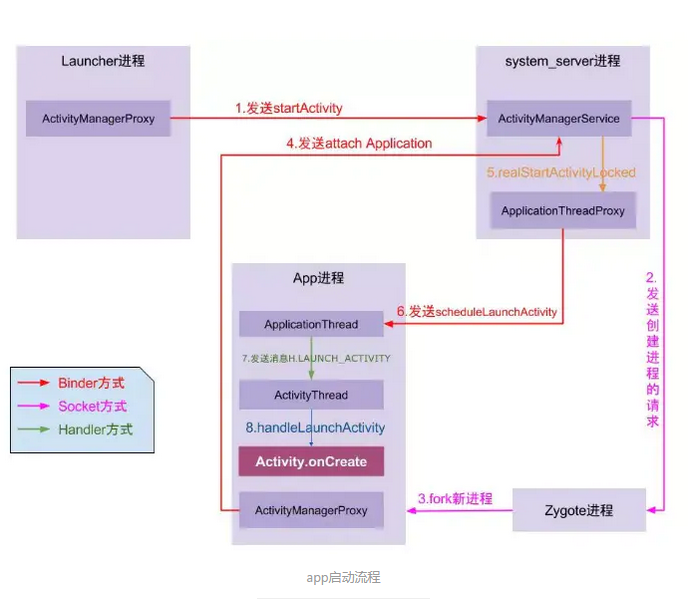


1. execStartActivity里获取跨进程服务，ActivityManagerService类，然后Activity的启动就转移到AMS中；
2. AMS把启动转移到了ActivityTaskManagerService(ATMS)中了，ATMS用于管理Activity及其容器(任务、堆栈、显示等)的系统服务；
3. 启动Activity的操作从客户端 跨进程 转移到 AMS，AMS通过ActivityTaskManagerService、ActivityStarter、ActivityStack、ActivityStackSupervisor 对 Activity任务、activity栈、Activity记录 管理后，又用过跨进程把正在启动过程又转移到了客户端。
4. ApplicationThread把启动Activity的操作，通过mH切到了主线程，走到了ActivityThread的handleLaunchActivity方法；
5. 获取完Activity相关数据后，创建Activity实例，创建Application实例（如果没有的话），使用attach关联上下文，调用Activity生命周期；
6. handleResumeActivity做了以下事情：

通过performResumeActivity方法，内部调用生命周期onStart、onResume；

通过activity.makeVisible方法，添加window、设置window可见。(所以视图的真正可见是在onResume方法之后)

1. app启动流程



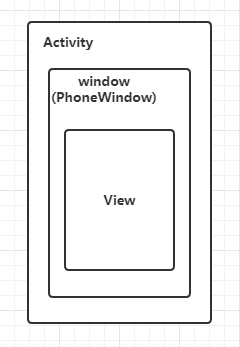
①App发起进程：当从桌面启动应用，则发起进程便是Launcher所在进程;当从某App内启动远程进程，则发送进程便是该App所在进程。发起进程先通过binder发送消息给system\_server进程;

②system\_server进程：调用Process.start()方法，通过socket向zygote进程发送创建新进程的请求;

③zygote进程：在执行ZygoteInit.main()后便进入runSelectLoop()循环体内，当有客户端连接时便会执行ZygoteConnection.runOnce()方法，再经过层层调用后fork出新的应用进程;

④新进程：执行handleChildProc方法，最后调用ActivityThread.main()方法

1. Activity与Window与View的关系



（1）Activity

从 ContextImpl -> startActivity-> scheduleLaunchActivity(AMS) 到最后 ActivityThread.performLaunchActivity -> Activity.attach中创建出PhoneWindow。

Activity并不负责视图控制，它只是控制生命周期和处理事件。真正控制视图的是Window。一个Activity包含了一个Window，Window才是真正代表一个窗口。

Activity就像一个控制器，统筹视图的添加与显示，以及通过其他回调方法，来与Window、以及View进行交互。

（2）Window

ActivityThread.performLaunchActivity -> Activity.attach中初始化了Window。

表示一个窗口的概念，是所有View的直接管理者，任何视图都通过Window呈现(点击事件由Window->DecorView->View; Activity的setContentView底层通过Window完成)

Window是一个抽象类，具体实现是PhoneWindow。PhoneWindow中有个内部类DecorView，通过创建DecorView来加载Activity中设置的布局

创建Window需要通过WindowManager创建，通过WindowManager将DecorView加载其中，并将DecorView交给ViewRoot，进行视图绘制以及其他交互

Window具体实现位于WindowManagerService中，WindowManager和WindowManagerService的交互是通过IPC完成

（3）View

View 的顶级根节点是 DecorView，DecorView 是 FrameLayout的子类。

执行Activity的setContentView方法，内部是调用PhoneWindow的setContentView方法，在PhoneWindow中完成DecorView的创建。流程

1、Activity中的setContentView  
2、PhoneWindow中的setContentView  
3、PhoneWindow中的installDecor

Window 的 实例对象 WindowManager 添加 DecorView，并将 DecorView 交给 ViewRootImpl，ViewRootImpl 是 WindowManager 和 DecorView 的纽带，在起 performTraversals()方法中绘制 View。

1. **Service相关**
2. Servicve启动方式：
3. startService()
4. stopService()
5. bindService()
6. unbindService()
7. 生命周期

startService启动的生命周期

onCreate() 当Service第一次被创建时，由系统调用。

onStartCommand() 当startService方法启动Service时，该方法被调用。

onDestroy() 当Service不再使用时，由系统调用。

bindService启动的生命周期

onCreate() 当Service被创建时，由系统调用。

onBind() 当bindService方法启动Service时，该方法被调用。

onUnbind() 当unbindService方法解除绑定时，该方法被调用。

onDestroy() 当Service不再使用时，由系统调用。

1. IntentService

IntentService的特点：

IntentService会创建单独的worker线程来处理所有的Intent请求。

IntentService会创建单独的worker线程来处理onHandleIntent()方法实现的代码，因此开发者无须处理多线程问题。

1. onStartCommend返回值
2. START\_NOT\_STICKY

如果系统在 onStartCommand() 返回后终止服务，则除非有挂起 Intent 要传递，否则系统不会重建服务。这是最安全的选项，可以避免在不必要时以及应用能够轻松重启所有未完成的作业时运行服务

1. START\_STICKY

如果系统在 onStartCommand() 返回后终止服务，则会重建服务并调用 onStartCommand()，但不会重新传递最后一个 Intent。相反，除非有挂起 Intent 要启动服务（在这种情况下，将传递这些 Intent ），否则系统会通过空 Intent 调用 onStartCommand()。这适用于不执行命令、但无限期运行并等待作业的媒体播放器（或类似服务）

1. START\_REDELIVER\_INTENT

如果系统在 onStartCommand() 返回后终止服务，则会重建服务，并通过传递给服务的最后一个 Intent 调用 onStartCommand()。任何挂起 Intent 均依次传递。这适用于主动执行应该立即恢复的作业（例如下载文件）的服务

1. **ContentProvider相关**

ContentProvider(数据提供者)是应用程序之间共享数据的一种接口机制，是一种更为高级的数据共享方法。

1. **BroadcastReceiver相关**
2. 广播类型
3. 普通广播
4. 系统广播
5. 有序广播

发送的广播被接收者有序的接收，根据接收对象的优先级（Priority属性的值决定，值越大，优先级越高；Priority属性相同时，动态注册的广播优先于静态注册的广播）来决定接受顺序。有序广播可以对广播进行拦截，这样之后的接收者就接受不到广播了，也可以对广播内容进行修改

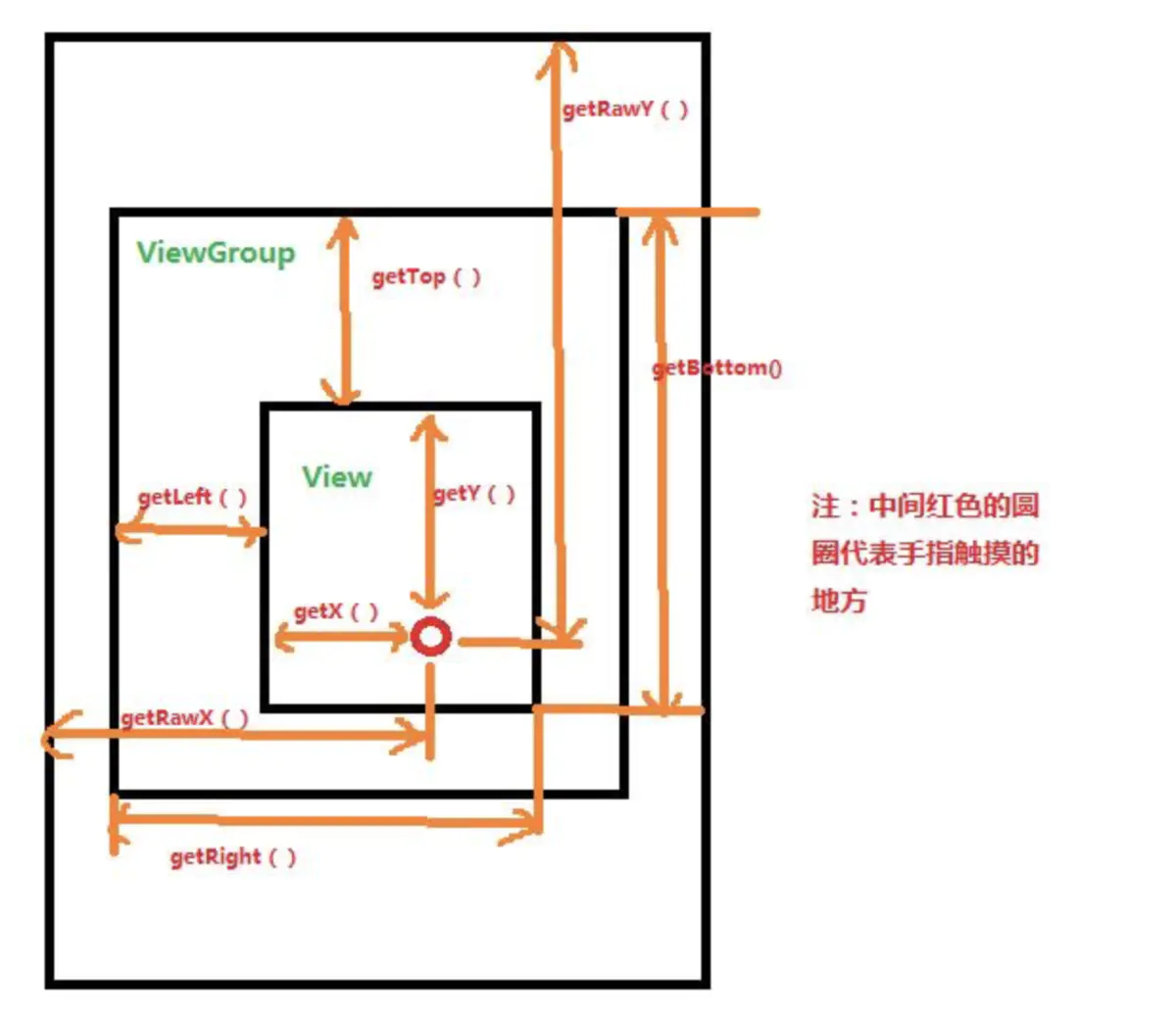
1. 粘性广播

粘性广播一般用来确保重要的状态改变后的信息被持久保存，当下一个注册粘性广播的接收者注册成功后可以获得对应类型的广播之前返回的数据状态；(android 5.0之后将其设置为deprecated,不再推荐应用使用)

1. App应用内广播

用于应用内的广播机制，相对于普通广播，安全性和效率都更高，一般使用LocalBroadcastManager来注册和发送

1. **View相关**
2. **View 的坐标系**



1. **View 的绘制流程**

measure: 判断是否需要重新计算 View 的大小，需要的话则计算

layout: 判断是否需要重新计算 View 的位置，需要的话则计算

draw: 判断是否需要重新绘制 View，需要的话则重绘制

1. **MeasureSpec**

MeasureSpec的值保存在一个int值当中。一个int值有32位，前两位表示模式mode后30位表示大小size。即MeasureSpec = mode + size。

1. View

UNSPECIFIED：无限制，View对尺寸没有任何限制，View设置为多大就应当为多大；

EXACTLY：精准模式，View需要一个精确值，这个值即为MeasureSpec当中的Size；

AT\_MOST：最大模式，View的尺寸有一个最大值，View不可以超过MeasureSpec当中的Size值；

1. 父View

UNSPECIFIED：父布局没有做出限制，子View有自己的尺寸，则使用，如果没有则为0；

EXACTLY：父布局采用精准模式，有确切的大小，如果有大小则直接使用，如果子View没有大小，子View不得超出父view的大小范围；

AT\_MOST：父布局采用最大模式，存在确切的大小，如果有大小则直接使用，如果子View没有大小，子View不得超出父view的大小范围；

1. **Scroller**

用于实现 View 的弹性滑动，Scroller 本身无法让 View 弹性滑动，需要和 View 的 computeScroll 方法配合使用。

1. **进程相关**
2. **进程生命周期**

（1）前台进程

托管用户正在交互的 Activity（已调用 Activity 的 onResume() 方法）

托管某个 Service，后者绑定到用户正在交互的 Activity

托管正在“前台”运行的 Service（服务已调用 startForeground()）

托管正执行一个生命周期回调的 Service（onCreate()、onStart() 或 onDestroy()）

托管正执行其 onReceive() 方法的 BroadcastReceiver

（2）可见进程

托管不在前台、但仍对用户可见的 Activity（已调用其 onPause() 方法）。例如，如果 re前台 Activity 启动了一个对话框，允许在其后显示上一 Activity，则有可能会发生这种情况。

托管绑定到可见（或前台）Activity 的 Service

（3）服务进程

正在运行已使用 startService() 方法启动的服务且不属于上述两个更高类别进程的进程。

（4）后台进程

包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 onStop() 方法）。通常会有很多后台进程在运行，因此它们会保存在 LRU （最近最少使用）列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。

（5）空进程

不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

1. **多进程**

（1）私有进程：android:process=":remote"，以冒号开头，冒号后面的字符串原则上是可以随意指定的。如果我们的包名为“com.biyou.multiprocess”，则实际的进程名 为“com.biyou.multiprocess:remote”。这种设置形式表示该进程为当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中。

（2）全局进程：进程名称不以“:”开头的进程都可以叫全局进程，如android:process=“com.secondProcess”，以小写字母开头，表示运行在一个以这个名字命名的全局进程中，其他应用通过设置相同的ShareUID可以和它跑在同一个进程。

（3）两个应用通过ShareUID运行在同一个进程中是有要求的，就是需要两个应用有相同的UID并且签名也要相同。

一般地，使用多进程会造成如下几方面的影响：

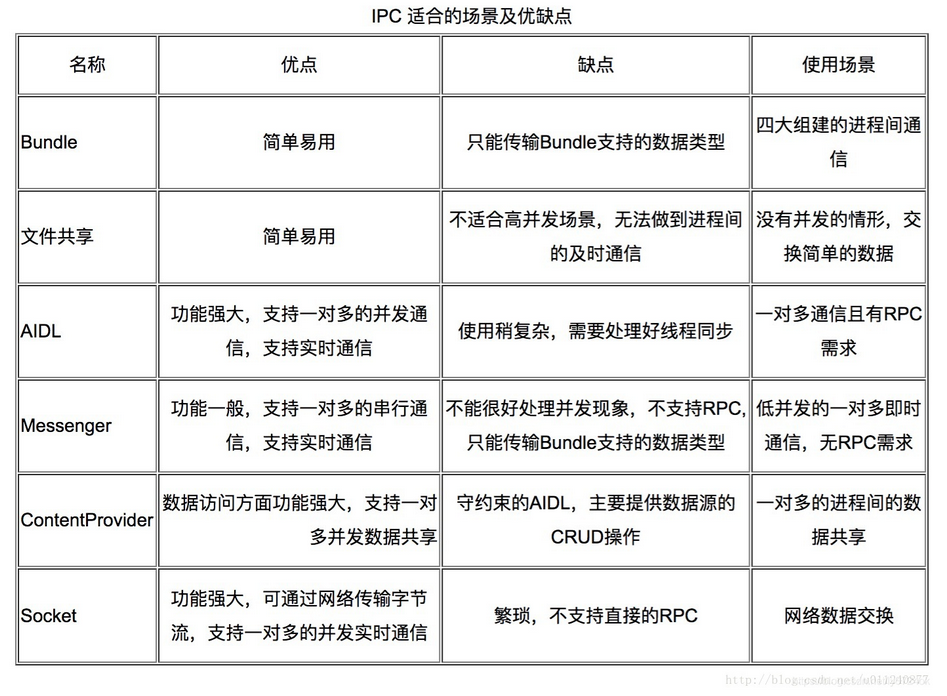
（1） 静态成员和单例模式完全失效。

（2） 线程同步机制完全消失：因为不是同一个内存，那么无论是锁对象还是全局类都无法保证线程同步，因为不同进程锁的不是同一个对象。

（3） SharePreference可靠性下降：SharePreference不支持两个进程同时执行写操作，因为会导致数据丢失,因为SharedPreferences底层是通过读写XML文件实现的，并发写显然会出现问题，甚至读/写多可能出问题。

（4） Application会多次创建：系统在创建进程的同时分配独立的虚拟机（即代表会多次创建Application），当一个组件运行在一个新的进程中，由于 系统要创建新的进程同时分配独立的虚拟机，因此就是一个启动应用的过程。既然重新启动则会创建新的Application。

1. **进程间通信（IPC）**



（1）只有允许不同应用的客户端用 IPC 方式调用远程方法，并且想要在服务中处理多线程时，才有必要使用 AIDL

（2）如果需要调用远程方法，但不需要处理并发 IPC，就应该通过实现一个 Binder 创建接口

（3）如果您想执行 IPC，但只是传递数据，不涉及方法调用，也不需要高并发，就使用 Messenger 来实现接口

（4）如果需要处理一对多的进程间数据共享（主要是数据的 CRUD），就使用 ContentProvider

（5）如果要实现一对多的并发实时通信，就使用 Socket

1. **Binder机制**

（1）为什么使用Binder：

**性能方面：**

Binder相对于传统的Socket方式，更加高效

Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，共享内存方式一次内存拷贝都不需要，但实现方式又比较复杂。

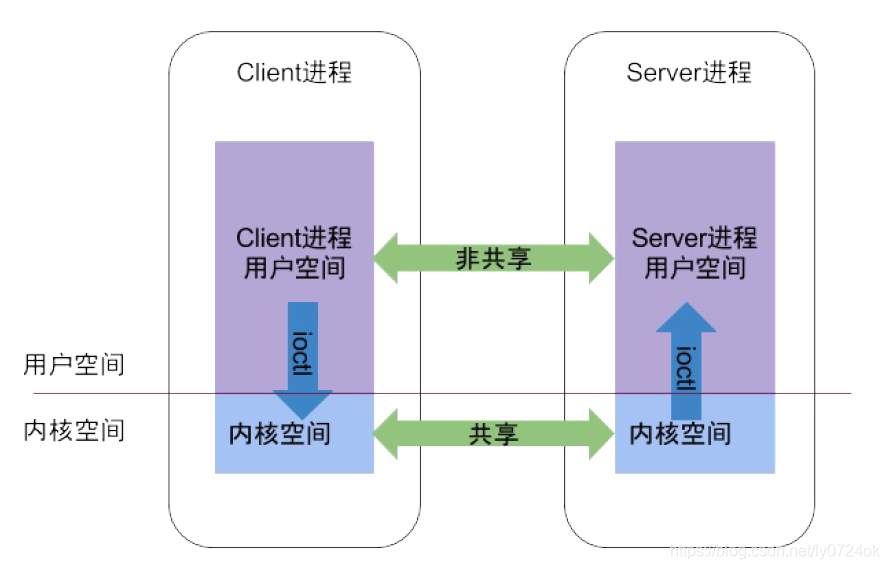
在这里插入图片描述

**安全方面：**

传统的进程通信方式对于通信双方的身份并没有做出严格的验证，比如Socket通信的IP地址是客户端手动填入，很容易进行伪造

Binder机制从协议本身就支持对通信双方做身份校检，从而大大提升了安全性。

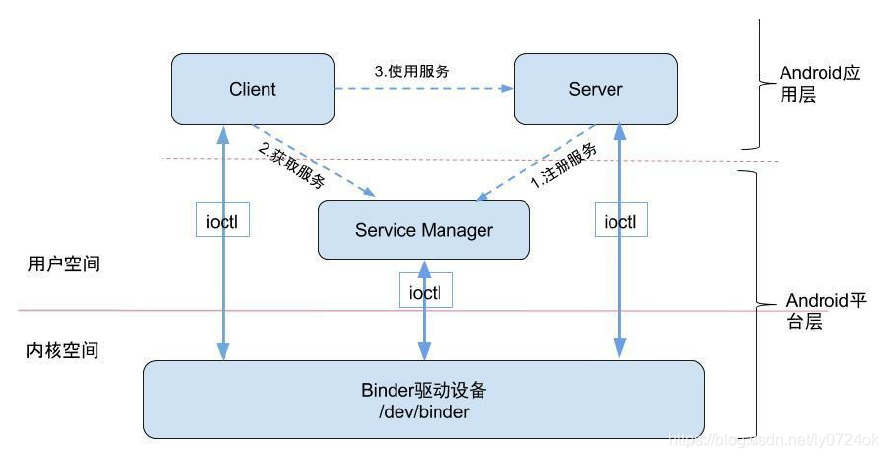
（2）IPC原理：



工作流程:

发送进程通过系统调用，将需要发送的数据拷贝到内核空间的缓冲区中。内核服务程序唤醒接收进程的接收线程，通过系统调用将数据发送到接收进程的用户空间中，完成数据的发送。所以这里通过两次拷贝完成了一次数据的发送。

·Binder通信模型：



**Client进程：**使用服务的进程。

**Server进程：**提供服务的进程。

**ServiceManager进程：**ServiceManager的作用是将字符形式的Binder名字转化成Client中对该Binder的引用，使得Client能够通过Binder名字获得对Server中Binder实体的引用。

**Binder驱动：**驱动负责进程之间Binder通信的建立，Binder在进程之间的传递，Binder引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。

·跨进程通信的基本原理

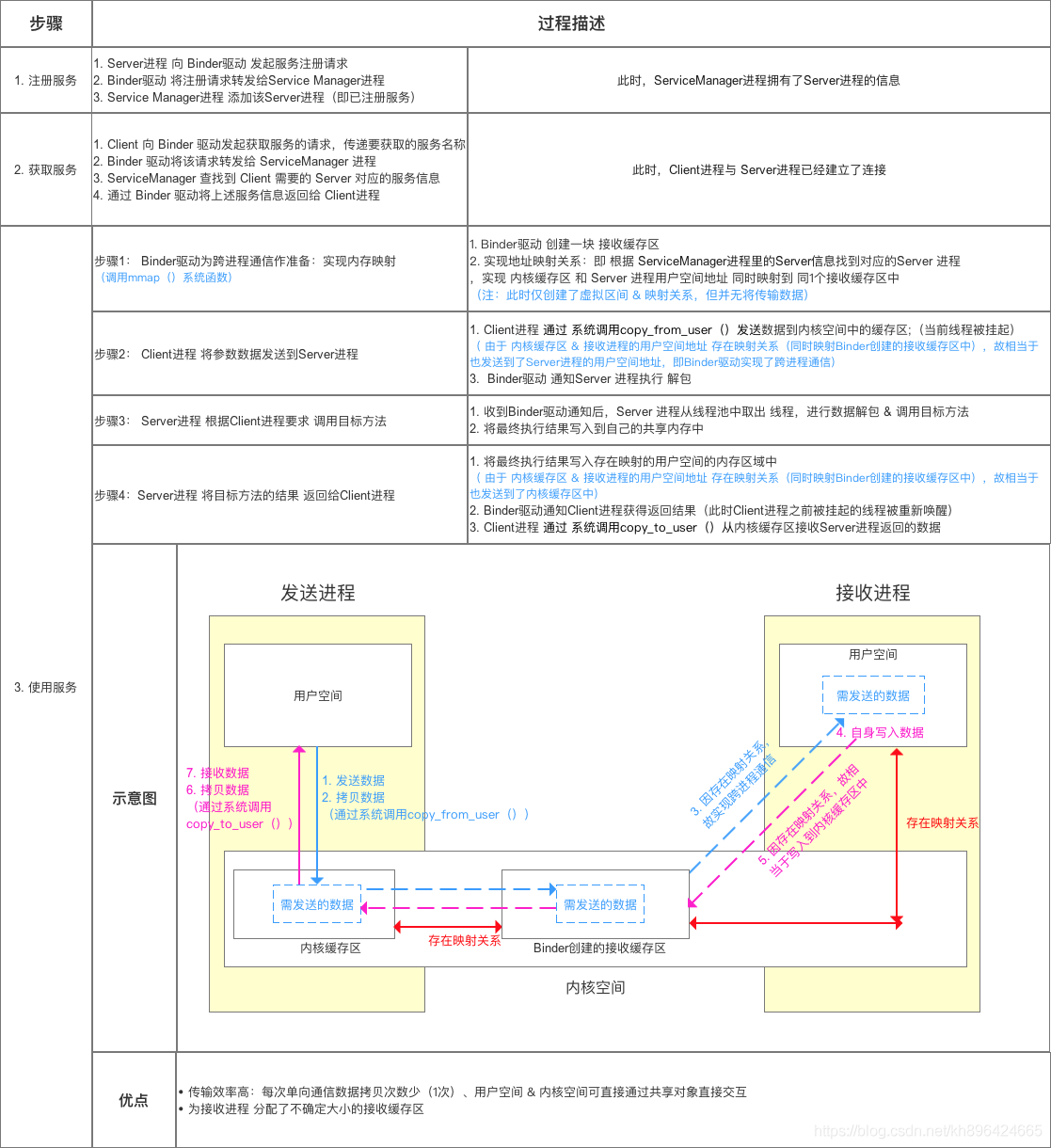
核心原理：内存映射

工作流程：

Binder驱动创建一块接收缓冲区

在内核空间中确定一块用于接收数据的Buffer，然后将用户空间的Buffer与内核空间的Buffer映射到实际的物理内存上，实现用户空间Buffer与内核空间Buffer的同步化

由于内核缓冲区&接收进程的用户地址空间存在映射关系，故相当于也发送到了接收进程的用户地址空间，实现了跨进程通信



1. **进程保活**
2. 进程被杀情况



1. 进程保活方案

* 开启一个像素的 Activity
* 使用前台服务
* 多进程相互唤醒
* JobSheduler 唤醒
* 粘性服务 & 与系统服务捆绑

1. **window相关**
2. Window属性

·变量Z-Order，决定了window的高度。window一共可分为三类：

（1）应用程序窗口：应用程序窗口一般位于最底层，Z-Order在1-99；

（2）子窗口：子窗口一般是显示在应用窗口之上，Z-Order在1000-1999；

（3）系统级窗口：系统级窗口一般位于最顶层，不会被其他的window遮住，如Toast，Z-Order在2000-2999。如果要弹出自定义系统级窗口需要动态申请权限。

Z-Order越大，window越靠近用户，也就显示越高，高度高的window会覆盖高度低的window。

·变量flags，常见的有KEEP\_SCREEN\_ON、FLAG\_FULLSCREEN。

·变量solfInputMode，如我们在微信聊天的时候，点击输入框，当软键盘弹起来的时候输入框也会被顶上去。如果你不想被顶上去，也可以设置为被软键盘覆盖。

1. Window的添加过程

addView-->mGlobal.addView-->ViewRootImpl.setView-->mWindowSession.addToDisplay（WindowManagerService处理创建window等逻辑）



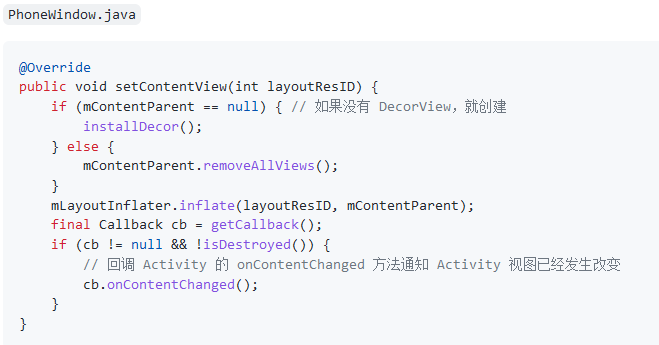
window的添加过程是通过PhoneWindow对应的WindowManagerImpl来添加window，内部会调用WindowManagerGlobal来实现。WindowManagerGlobal会使用viewRootImpl来进行跨进程通信让WMS执行创建window的业务。

每个应用都有一个windowSession，用于负责和WMS的通信，如ApplicationThread与AMS的通信。

1. Activity与Window

在 Activity 的创建过程中，最终会由 ActivityThread 的 performLaunchActivity() 来完成整个启动过程，该方法内部会通过类加载器创建 Activity 的实例对象，并调用 attach 方法关联一系列上下文环境变量。在 Activity 的 attach 方法里，系统会创建所属的 Window 对象并设置回调接口，然后在 Activity 的 setContentView 方法中将视图附属在 Window 上：





这个时候 DecorView 还没有被 WindowManager 正式添加。在 ActivityThread 的 handleResumeActivity 方法中，首先会调用 Activity 的 onResume 方法，接着调用 Activity 的 makeVisible()，完成 DecorView 的添加和显示过程：



1. PopupWindow与Window

showAtLocation-->preparePopup-->invokePopup

（1）根据参数构建popupDecorView

（2）把popupDecorView添加到屏幕上

1. Dialog与Window

Dialog 的 Window 的创建过程和 Activity 类似，创建同样是通过 PolicyManager 的 makeNewWindow 方法完成的，创建后的对象实际就是 PhoneWindow。当 Dialog 被关闭时，会通过WindowManager 来移除DecorView：mWindowManager.removeViewImmediate(mDecor)。



普通 Dialog 必须采用 Activity 的 Context，采用 Application 的 Context 就会报错，是因为应用 token 所导致，应用 token 一般只有 Activity 拥有。系统 Window 比较特殊，不需要 token。

1. Toast与Window

Toast 属于系统 Window ，由于其具有定时取消功能，所以系统采用了 Handler。Toast 的内部有两类 IPC 过程，第一类是 Toast 访问 NotificationManagerService，第二类是 NotificationManagerService 回调 Toast 里的 TN 接口。





1. **混淆相关**

1.检查混淆和追踪异常

开启 Proguard 功能，则每次构建时 ProGuard 都会输出下列文件：

dump.txt  
说明 APK 中所有类文件的内部结构。

mapping.txt  
提供原始与混淆过的类、方法和字段名称之间的转换。

seeds.txt  
列出未进行混淆的类和成员。

usage.txt  
列出从 APK 移除的代码。

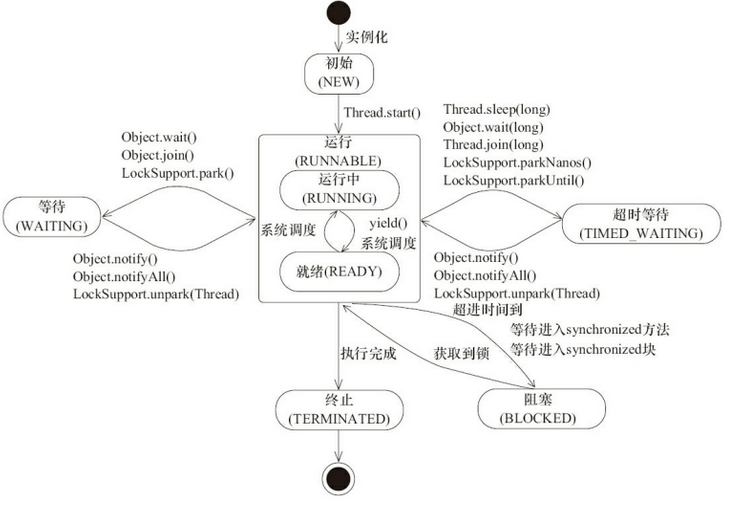
这些文件保存在 /build/outputs/mapping/release/ 中。我们可以查看 seeds.txt 里面是否是我们需要保留的，以及 usage.txt 里查看是否有误删除的代码。 mapping.txt 文件很重要，由于我们的部分代码是经过重命名的，如果该部分出现 bug，对应的异常堆栈信息里的类或成员也是经过重命名的，难以定位问题。我们可以用 retrace 脚本（在 Windows 上为 retrace.bat；在 Mac/Linux 上为 retrace.sh）。它位于 /tools/proguard/ 目录中。该脚本利用 mapping.txt 文件和你的异常堆栈文件生成没有经过混淆的异常堆栈文件,这样就可以看清是哪里出问题了。

# JAVA方面

1. **JAVA并发相关**
2. 进程和线程的关系

线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。线程执行开销小，但不利于资源的管理和保护；而进程正相反。

1. 线程
2. 生命周期



线程创建之后它将处于 **NEW（新建）**状态，调用 start() 方法后开始运行，线程这时候处于 **READY（可运行）**状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 **RUNNING（运行）**状态。

当线程执行 wait()方法之后，线程进入 **WAITING（等待）**状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIMED\_WAITING（超时等待）**状态相当于在等待状态的基础上增加了超时限制，比如通过 sleep（long millis）方法或 wait（long millis）方法可以将 Java 线程置于 TIMED\_WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED（阻塞）**状态。线程在执行 Runnable 的run()方法之后将会进入到 **TERMINATED（终止）**状态

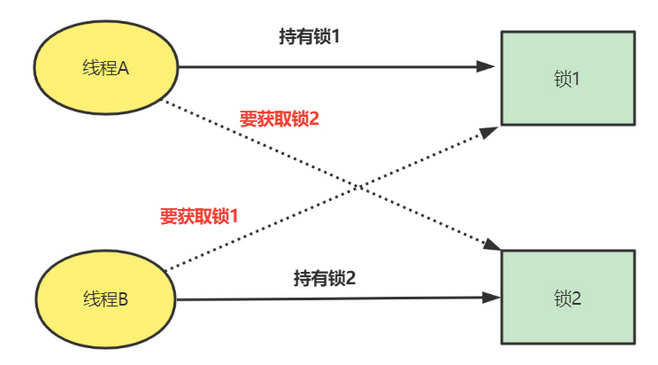
1. 相关问题

·为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

调用 start() 方法方可启动线程并使线程进入就绪状态，直接执行 run() 方法的话不会以多线程的方式执行。

1. 死锁

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。



说明：线程A持有锁1，线程B持有锁2，线程做了一些操作后，线程A想要获取锁2，线程B想要获取锁1，因此线程被无限期地阻塞，因此程序不可能正常终止。

避免死锁：

·保持加锁顺序；

·尽量降低锁的使用粒度 : 用不同的锁，而不是同一个锁；

·避免锁的嵌套；

1. Synchronized和volatile

**① Synchronized**

synchronized 关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行

1. synchronized 关键字的三种使用方式：

·修饰实例方法；

synchronized void method() {

//业务代码

}

·修饰静态方法；

synchronized static void method() {

//业务代码

}

·修饰代码块；

synchronized(this) {

//业务代码

}

总结：

·synchronized 关键字加到 static 静态方法和 synchronized(class) 代码块上都是是给 Class 类上锁。

·synchronized 关键字加到实例方法上是给对象实例上锁。

·尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓存功能！

1. 双重校验锁实现对象单例（线程安全）



这段代码其实是分为三步执行：

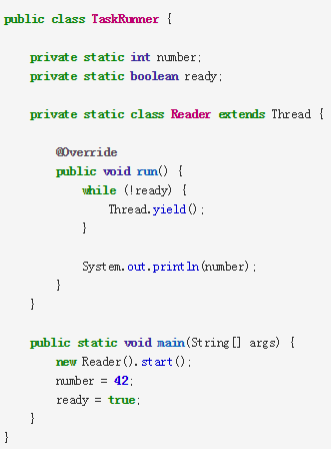
·为 uniqueInstance 分配内存空间

·初始化 uniqueInstance

·将 uniqueInstance 指向分配的内存地址

**② Volatile**

例子：



1. 内存可见性

多线程运行在多个CPU上，而每个线程都会有自己的cache，因此无法保证从主存中读取数据的顺序，即无法保证各个CPU上的线程读取的变量数据一致。

结合上面的程序，主线程在其核心缓存中保留了ready和number的副本，而Reader线程也是同样保留了副本，之后主线程更新缓存值。在大多数现代处理器上，写入请求在发出后不会立即应用。事实上，处理器倾向于将这些写入排在一个特殊的写入缓冲区中。一段时间后，它们会一次性将这些写入应用到主内存中。

因此当主线程更新number和ready变量时，无法保Reader线程会看到什么。换句话说，Reader线程可能会立即看到更新的值，或者有一些延迟，或者根本不会。

1. 防止 JVM 的指令重排

所以，v**olatile 关键字 除了防止 JVM 的指令重排 ，还有一个重要的作用就是保证变量的可见性。**

**③ 并发编程的三个重要特性**

（1）**原子性**：一次操作或者多次操作，要么所有的操作全部都得到执行并且不会受到任何因素的干扰而中断，要么都不执行。synchronized 可以保证代码片段的原子性。

（2）**可见性**：当一个线程对共享变量进行了修改，那么另外的线程都是立即可以看到修改后的最新值。volatile 关键字可以保证共享变量的可见性。

（3）**有序性**：代码在执行的过程中的先后顺序，Java 在编译器以及运行期间的优化，代码的执行顺序未必就是编写代码时候的顺序。volatile 关键字可以禁止指令进行重排序优化。

**④ 说说 synchronized 关键字和 volatile 关键字的区别**

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在！

（1）volatile 关键字是线程同步的轻量级实现，所以 volatile 性能肯定比synchronized关键字要好 。但是 volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块 。

（2）volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都能保证。

（3）volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。

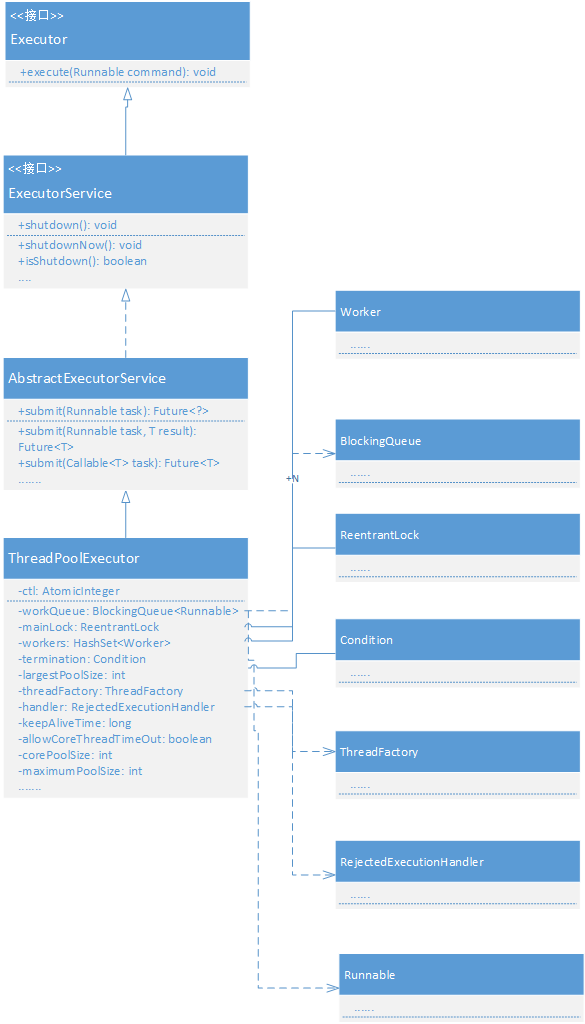
1. 锁

**悲观锁、乐观锁**

悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java 中，synchronized 关键字和 Lock 的实现类都是悲观锁。悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。

而乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。乐观锁在 Java 中是通过使用无锁编程来实现，最常采用的是 CAS 算法，Java 原子类中的递增操作就通过 CAS 自旋实现。乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

1. **线程池**



**ctl**

既表示线程池的状态，也表示线程池的容量

**corePoolSize**

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

**maximumPoolSize**

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize；

**keepAliveTime**

线程池维护线程所允许的空闲时间。当线程池中的线程数量大于corePoolSize的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime；

**unit**

keepAliveTime的单位；

**workQueue**

用来保存等待被执行的任务的阻塞队列，且任务必须实现Runable接口，在JDK中提供了如下阻塞队列：

1、ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务；

2、LinkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene；

3、SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；

4、priorityBlockingQuene：具有优先级的无界阻塞队列；

**threadFactory**

它是ThreadFactory类型的变量，用来创建新线程。默认使用Executors.defaultThreadFactory() 来创建线程。使用默认的ThreadFactory来创建线程时，会使新创建的线程具有相同的NORM\_PRIORITY优先级并且是非守护线程，同时也设置了线程的名称。

**handler**

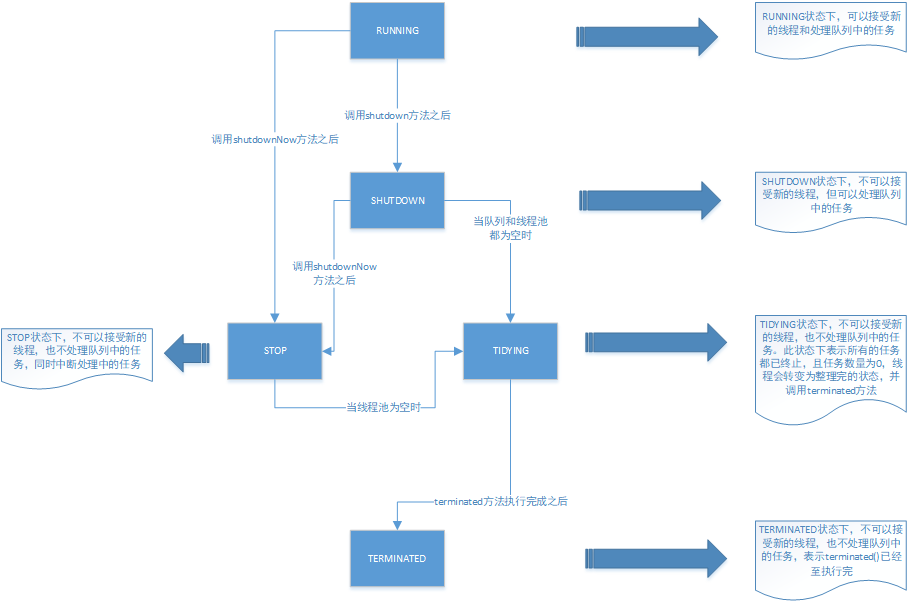
线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

AbortPolicy：直接抛出异常，默认策略；

CallerRunsPolicy：用调用者所在的线程来执行任务；

DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

DiscardPolicy：直接丢弃任务



**RUNNING**

状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。

状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0！

**SHUTDOWN**

状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。

状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

**STOP**

状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。

状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

**TIDYING**

状态说明：当所有的任务已终止，ctl记录的”任务数量”为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。 当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

**TERMINATED**

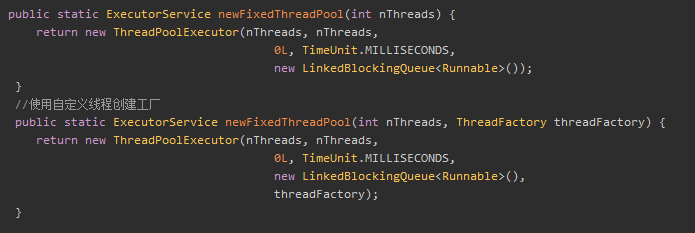
状态说明：线程池彻底终止，就变成TERMINATED状态。

状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED

**五种线程池创建类型**

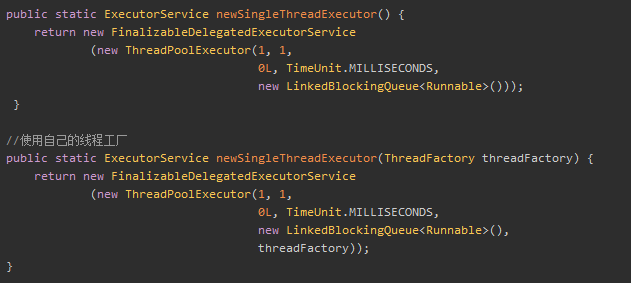
**newFixedThreadPool（固定大小线程池）**

newFixedThreadPool：创建一个核心线程个数和最大线程个数都为 nThreads 的线程池，并且阻塞队列长度为 Integer.MAX\_VALUE，keeyAliveTime=0 说明只要线程个数比核心线程个数多并且当前空闲则回收。代码如下：

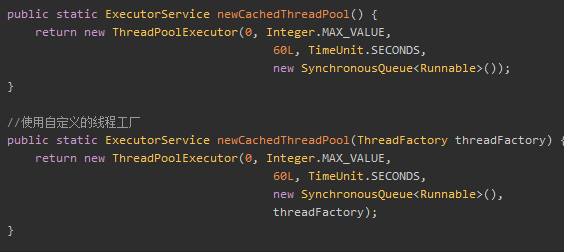


**newSingleThreadExecutor(单个后台线程）**

newSingleThreadExecutor：创建一个核心线程个数和最大线程个数都为1的线程池，并且阻塞队列长度为 Integer.MAX\_VALUE，keeyAliveTime=0 说明只要线程个数比核心线程个数多并且当前空闲则回收。代码如下：

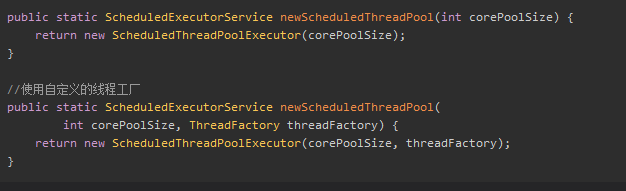
  
  
**newCachedThreadPool（无界线程池，可以进行自动线程回收）**

newCachedThreadPool：创建一个按需创建线程的线程池，初始线程个数为 0，最多线程个数为 Integer.MAX\_VALUE，并且阻塞队列为同步队列，keeyAliveTime=60 说明只要当前线程 60s 内空闲则回收。这个特殊在于加入到同步队列的任务会被马上被执行，同步队列里面最多只有一个任务。代码如下：



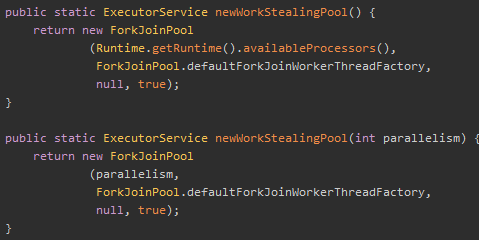
**newScheduledThreadPool (可调度)**

newScheduledThreadPool：创建一个定长线程池，支持定时及周期性任务执行。newScheduledThreadPool 和 其他线程池最大的区别是使用的阻塞队列是 DelayedWorkQueue，而且多了两个定时执行的方法scheduleAtFixedRate和scheduleWithFixedDelay，代码如下：



**newWorkStealingPool（并行操作）**

newWorkStealingPool：JDK1.8新增newWorkStealingPool，适合使用在很耗时的操作，但是newWorkStealingPool不是ThreadPoolExecutor的扩展，它是新的线程池类ForkJoinPool的扩展，但是都是在统一的一个Executors类中实现，由于能够合理的使用CPU进行对任务操作（并行操作），所以适合使用在很耗时的任务中。代码如下：



1、线程池的原理是什么？

线程池的实现原理其实主要是将线程加入线程池时，先判断当前核心线程数是否已达最大值，如果核心线程数没有已达到最大值，那么创建任务work并直接调度并处理任务；如果核心线程数已达到最大值，那么就会加入队列中等待调度。

2、线程池有哪些状态，每种状态分别能做什么？每种状态的走向是怎么样的？

3、线程池最大的容量是多少？

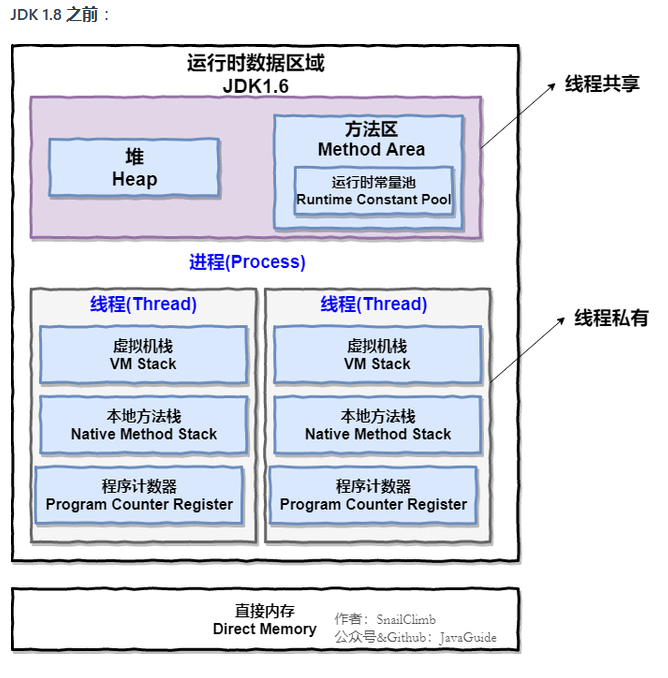
其中包含两个域，高3位表示[线程池](https://so.csdn.net/so/search?q=%E7%BA%BF%E7%A8%8B%E6%B1%A0&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/shanhai3000/article/details/_blank)的5中状态（Running、ShutDown、Stop、Tidying、Terminated），低29位表示线程池的数量。因此，理论上，线程池的最大容量为2^29-1=536870911（5亿多）。  
4、keepAliveTime用来干嘛？是线程多长时间没有得到调度的超时时间吗？

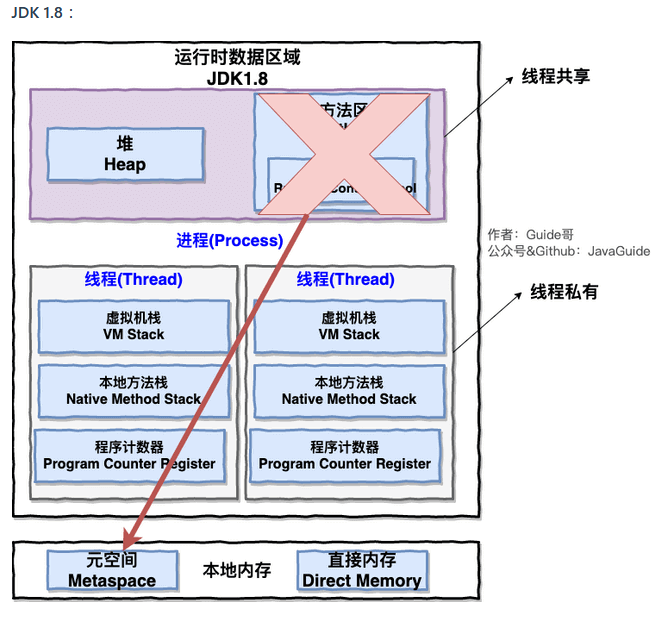
5、线程池是怎么实现线程的复用的？

线程池execute时，如果当前核心线程没满，则直接创建Work创建核心线程直接运行；如果当前核心线程已满且队列未则将线程放入队列中排队等待核心线程的调度。Work中创建线程并执行Worker的run，然后调用runWorker循环不断的从队列中获取等待中的线程进行运行，如果从队列队列为空就会挂起等待队列不为空的信号，这样实现了线程的复用。

6、线程池的核心线程是否是常住的？怎么取消核心线程的常住？

1. **JVM相关**
2. JAVA内存区域
3. 运行时数据区域





线程私有的：

程序计数器

虚拟机栈

本地方法栈

线程共享的：

堆

方法区

直接内存 (非运行时数据区的一部分)

1. 程序计数器

程序计数器主要有两个作用：

·字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

·在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

1. Java 虚拟机栈

方法调用的数据需要通过栈进行传递，每一次方法调用都会有一个对应的栈帧被压入栈中，每一个方法调用结束后，都会有一个栈帧被弹出。

栈由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法返回地址。



运行中栈可能会出现两种错误：

StackOverFlowError： 若栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 错误。

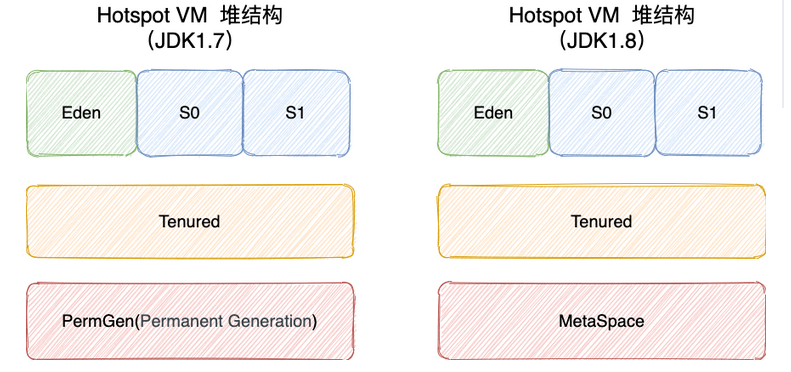
OutOfMemoryError： 如果栈的内存大小可以动态扩展， 如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

1. 本地方法栈

虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

1. 堆

此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。



在 JDK 7 版本及 JDK 7 版本之前，堆内存被通常分为下面三部分：

新生代内存(Young Generation)

老生代(Old Generation)

永久代(Permanent Generation)

JDK 8 版本之后 PermGen(永久) 已被 Metaspace(元空间) 取代，元空间使用的是直接内存。

堆这里最容易出现的就是 OutOfMemoryError 错误，并且出现这种错误之后的表现形式还会有几种，比如：

java.lang.OutOfMemoryError: GC Overhead Limit Exceeded ： 当 JVM 花太多时间执行垃圾回收并且只能回收很少的堆空间时，就会发生此错误。

java.lang.OutOfMemoryError: Java heap space :假如在创建新的对象时, 堆内存中的空间不足以存放新创建的对象, 就会引发此错误。

1. 方法区

当虚拟机要使用一个类时，它需要读取并解析 Class 文件获取相关信息，再将信息存入到方法区。方法区会存储已被虚拟机加载的类信息、字段信息、方法信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

1. 垃圾回收

在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。

1. 判断对象的死亡

**·引用计数法**

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

**·可达性分析算法**

这个算法的基本思想就是通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的，需要被回收。

哪些对象可以作为 GC Roots 呢？

·虚拟机栈(栈帧中的本地变量表)中引用的对象

·本地方法栈(Native 方法)中引用的对象

·方法区中类静态属性引用的对象

·方法区中常量引用的对象

·所有被同步锁持有的对象

1. 垃圾回收算法

**·标记-清除算法**

该算法分为“标记”和“清除”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

①效率问题

②空间问题（标记清除后会产生大量不连续的碎片）

**·标记-复制算法**

它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

**·标记-整理算法**

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

**·分代收集算法**

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

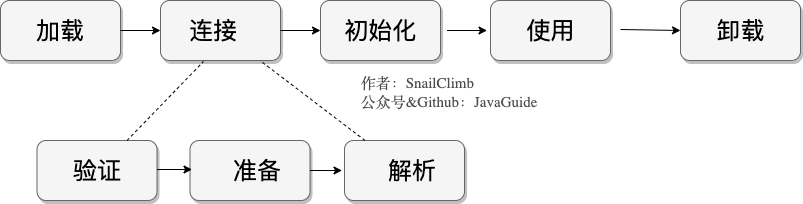
**比如在新生代中，每次收集都会有大量对象死去，所以可以选择”标记-复制“算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。**

1. 类文件结构

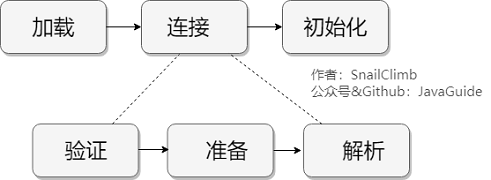


1. 类加载过程

一个类的完整生命周期如下：



系统加载 Class 类型的文件主要三步：**加载->连接->初始化**。连接过程又可分为三步：**验证->准备->解析。**



1. 初始化

初始化阶段是执行初始化方法 <clinit> ()方法的过程，是类加载的最后一步，这一步 JVM 才开始真正执行类中定义的 Java 程序代码(字节码)。

对于初始化阶段，虚拟机严格规范了有且只有 5 种情况下，必须对类进行初始化(只有主动去使用类才会初始化类)：

·当遇到 new 、 getstatic、putstatic 或 invokestatic 这 4 条直接码指令时，比如 new 一个类，读取一个静态字段(未被 final 修饰)、或调用一个类的静态方法时。

当 jvm 执行 new 指令时会初始化类。即当程序创建一个类的实例对象。

当 jvm 执行 getstatic 指令时会初始化类。即程序访问类的静态变量(不是静态常量，常量会被加载到运行时常量池)。

当 jvm 执行 putstatic 指令时会初始化类。即程序给类的静态变量赋值。

当 jvm 执行 invokestatic 指令时会初始化类。即程序调用类的静态方法。

·使用 java.lang.reflect 包的方法对类进行反射调用时如 Class.forname("..."), newInstance() 等等。如果类没初始化，需要触发其初始化。

·初始化一个类，如果其父类还未初始化，则先触发该父类的初始化。

·当虚拟机启动时，用户需要定义一个要执行的主类 (包含 main 方法的那个类)，虚拟机会先初始化这个类。

·MethodHandle 和 VarHandle 可以看作是轻量级的反射调用机制，而要想使用这 2 个调用， 就必须先使用 findStaticVarHandle 来初始化要调用的类。

·当一个接口中定义了 JDK8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。

1. 卸载

卸载类需要满足 3 个要求:

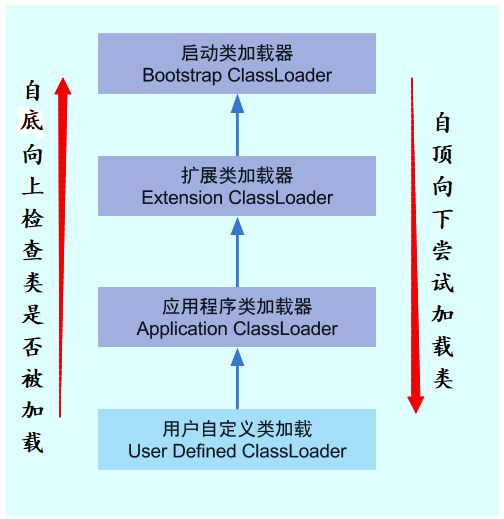
·该类的所有的实例对象都已被 GC，也就是说堆不存在该类的实例对象。

·该类没有在其他任何地方被引用

·该类的类加载器的实例已被 GC

1. 类加载器详解





1. 双亲委派模型

其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式

优势：

具备了一种带有优先级的层次关系，通过这种层级关可以**避免类的重复加载**，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次

1. **单例**



当 SingleTon 第一次被加载时，并不需要去加载 SingleTonHoler，只有当 getInstance() 方法第一次被调用时，才会去初始化 INSTANCE，这种方法不仅能确保线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。getInstance 方法并没有多次去 new 对象，取的都是同一个 INSTANCE 对象。

虚拟机会保证一个类的 <clinit>() 方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的 <clinit>() 方法，其他线程都需要阻塞等待，直到活动线程执行 <clinit>() 方法完毕

缺点在于无法传递参数，如Context等