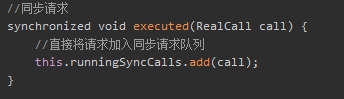
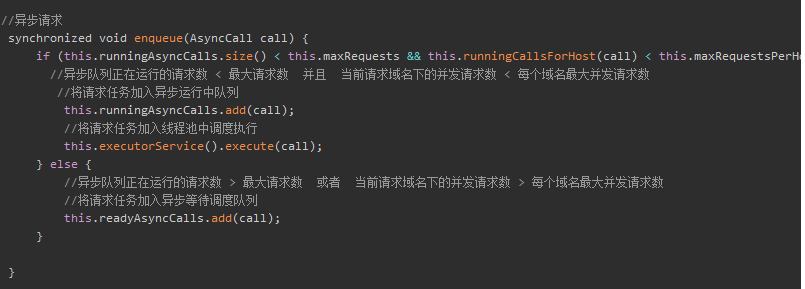
# Android源码方面

1. **Okhttp源码：**
2. **Dispatcher**

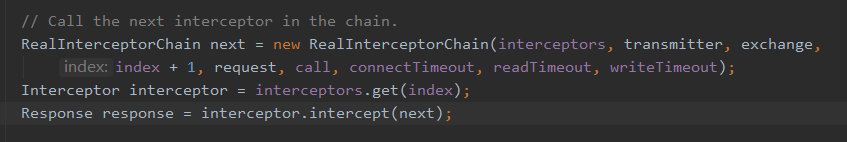
**同步请求：**dispatcher对应同步请求而言，只是将请求放入同步队列，便于后续的取消、完成等操作



**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；

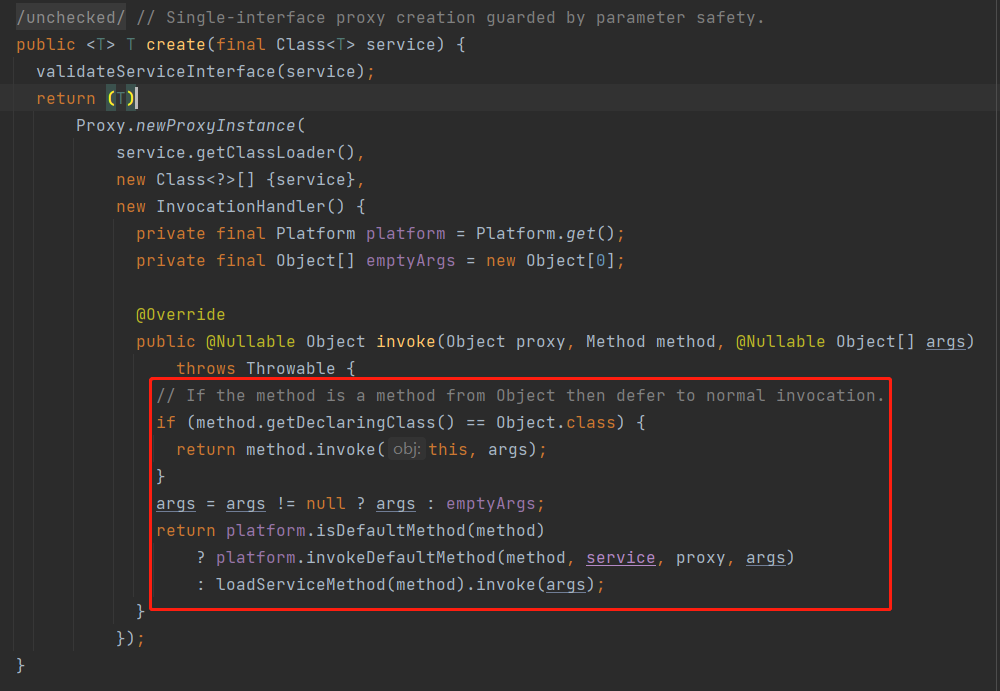
**3.5 服务器请求拦截器（CallServerInterceptor）**

1. **Retrofit源码**

retrofit.build() 配置参数，在create()中使用；

Retrofit.create()：

动态代理：



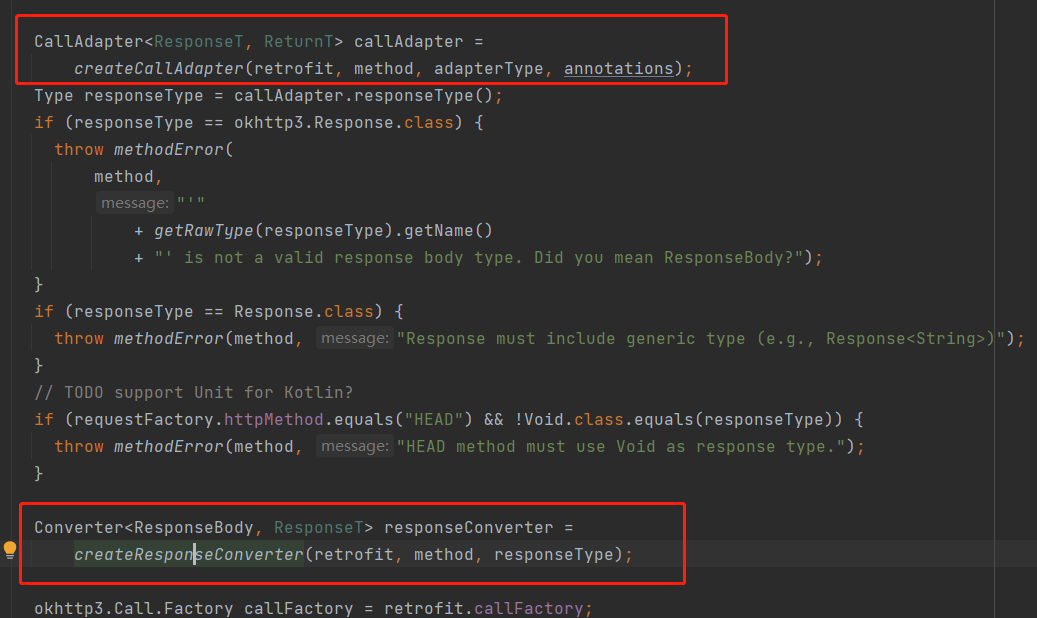
利用动态代理，当调用service.xxx()时，就会执行invoke方法里面的逻辑。

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；



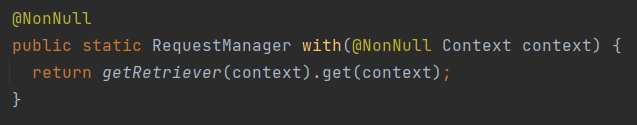
1. 通过RxJava3CallAdapterFactory执行OkHttp请求（Retrofit本身不发起网络请求）

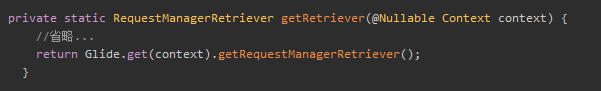
总结：

* 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)**



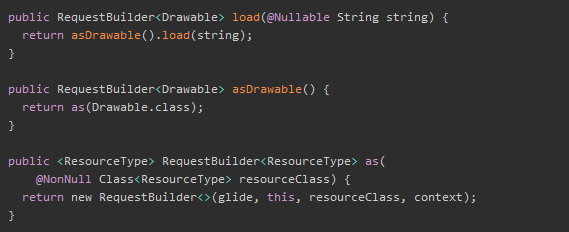


Glide.get(context)实现glide的初始化；

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

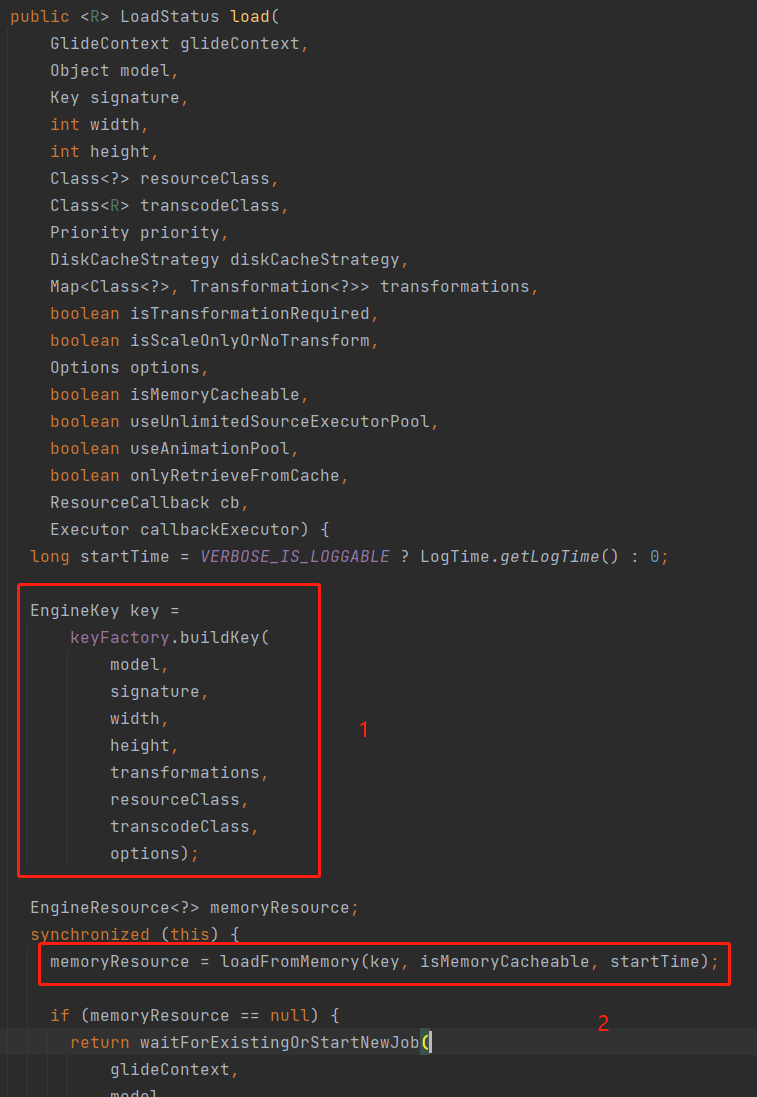
· 设置加载成功时的图片、动效

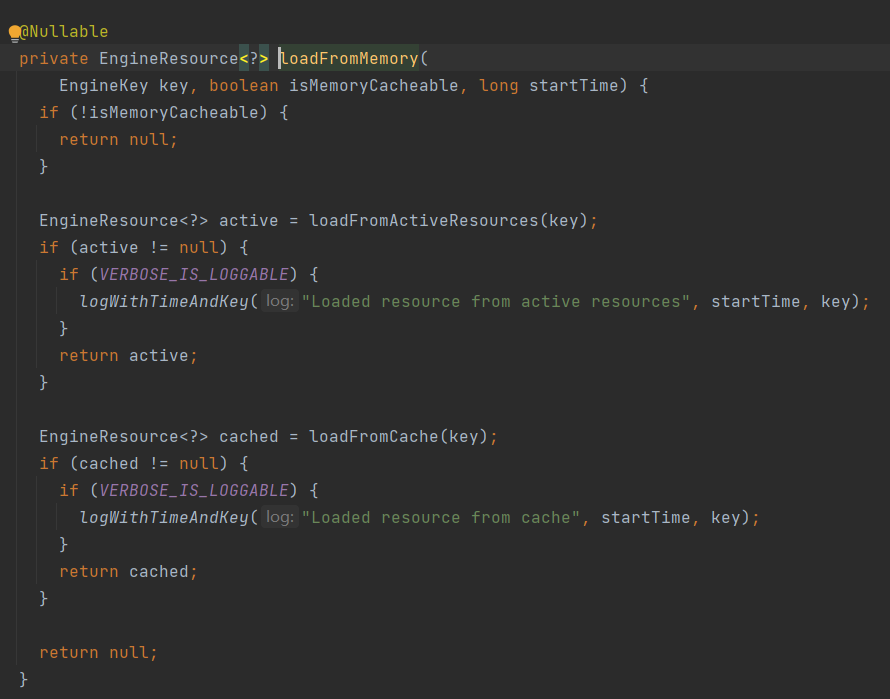
1. **Glide缓存机制**

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）-->

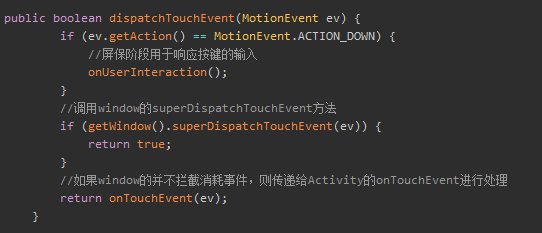
ObservableObserveOn中Observer的onSubscribe（Observer里的一个回调）-->

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。

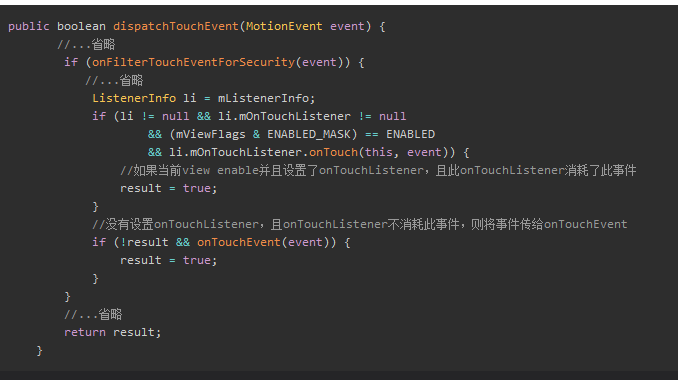
1. ViewGroup事件分发



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？

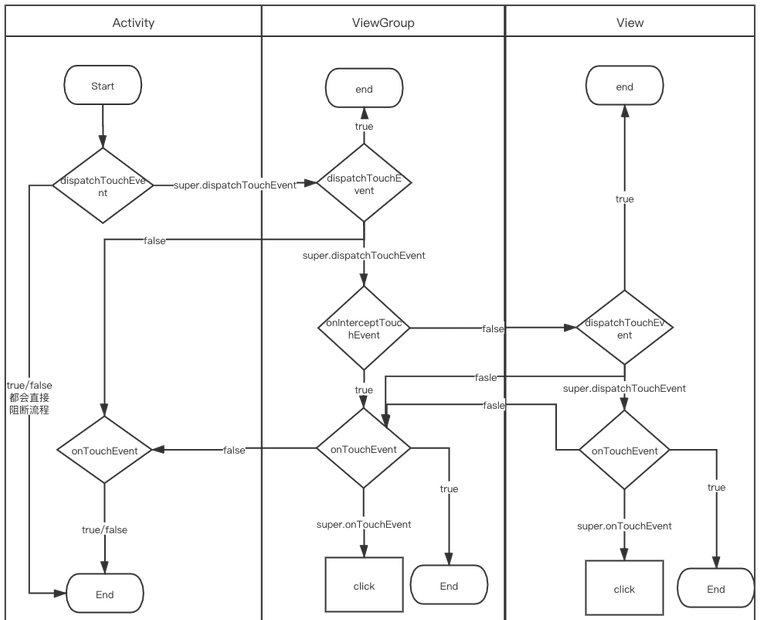
要响应事件，根据View的事件分发流程可以有几种解决方案：  
a、父控件拦截事件，将事件传递给view进行处理  
b、自己拦截事件：

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



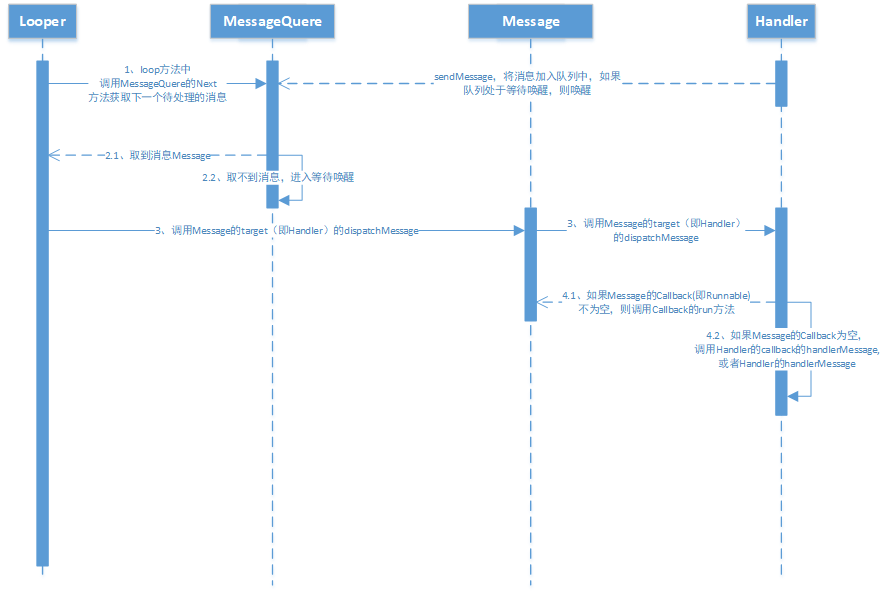
1. **Handler机制**

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

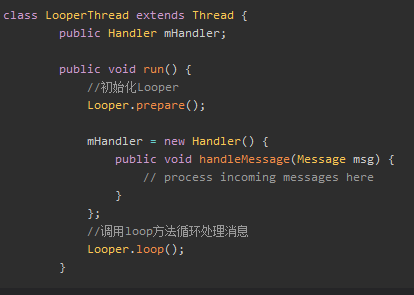
（2）消息会加入到MessageQueue中，等待Looper获取处理

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



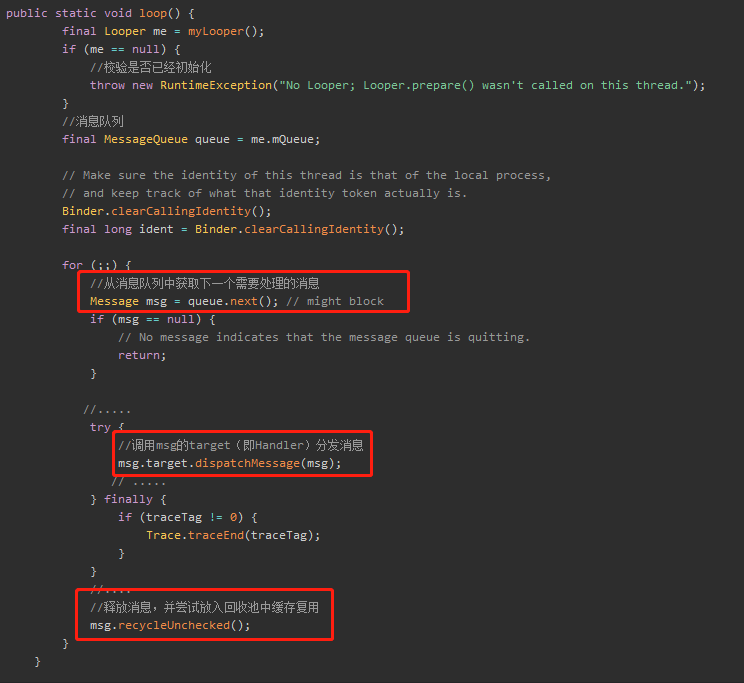
1. **Lopper**

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

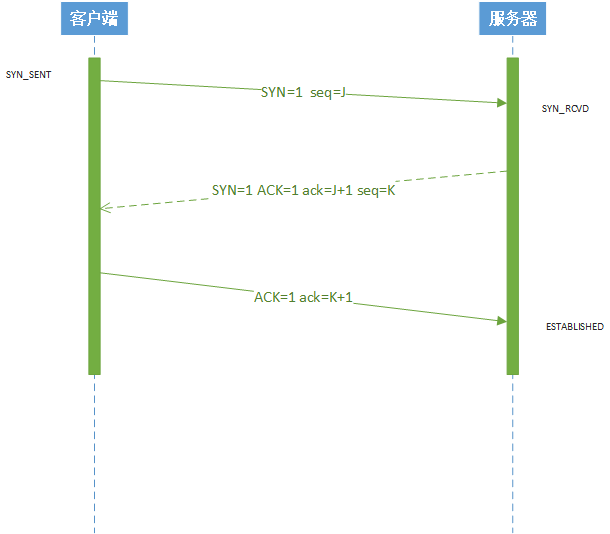
1. **UI只能在主线程改吗？**

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**

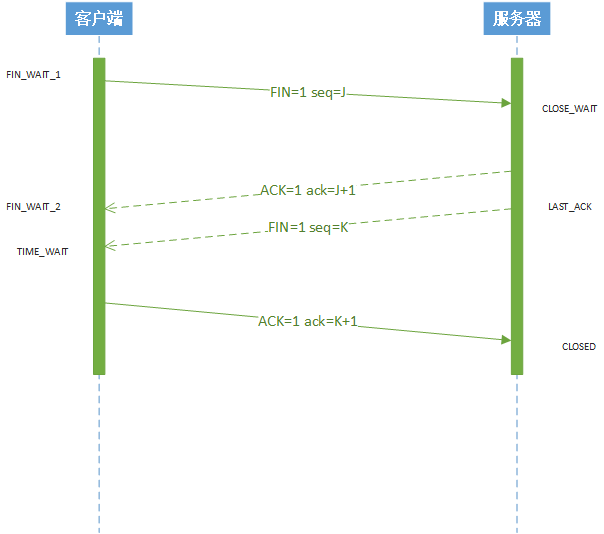


第一次握手：客户端将标志位SYN置为1，随机生成一个值seq=1，接着将数据表发给服务端，此时客户端进入SYN\_SENT（请求建立已发送）状态，等待服务端确认

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

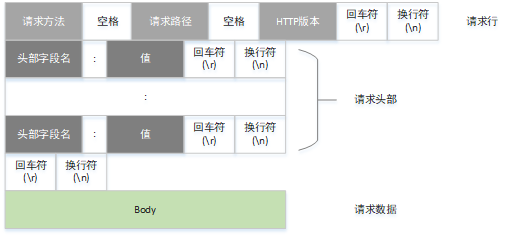
第二次挥手：服务端收到数据包，有FIN=1知道客户端请求断开连接，将ACK置为1，ack=J+1，接着数据包发送回给客户端，表示断开连接的请求已收到，此时服务端进入CLOSE\_WAIT（等待关闭）状态。客户端收到确认包之后，检测ACK是否为1，ack是否为J+1，检查正确之后表明服务端已经收到断开连接的请求，客户端随后进入FIN\_WAIT\_2状态

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**



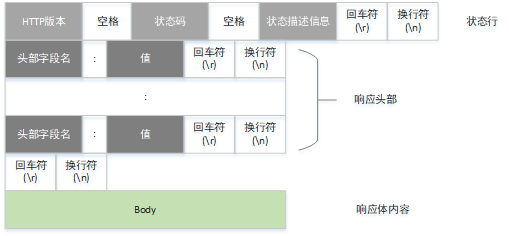
响应协议报文格式分四大部分：  
**1.状态行：**

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. **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

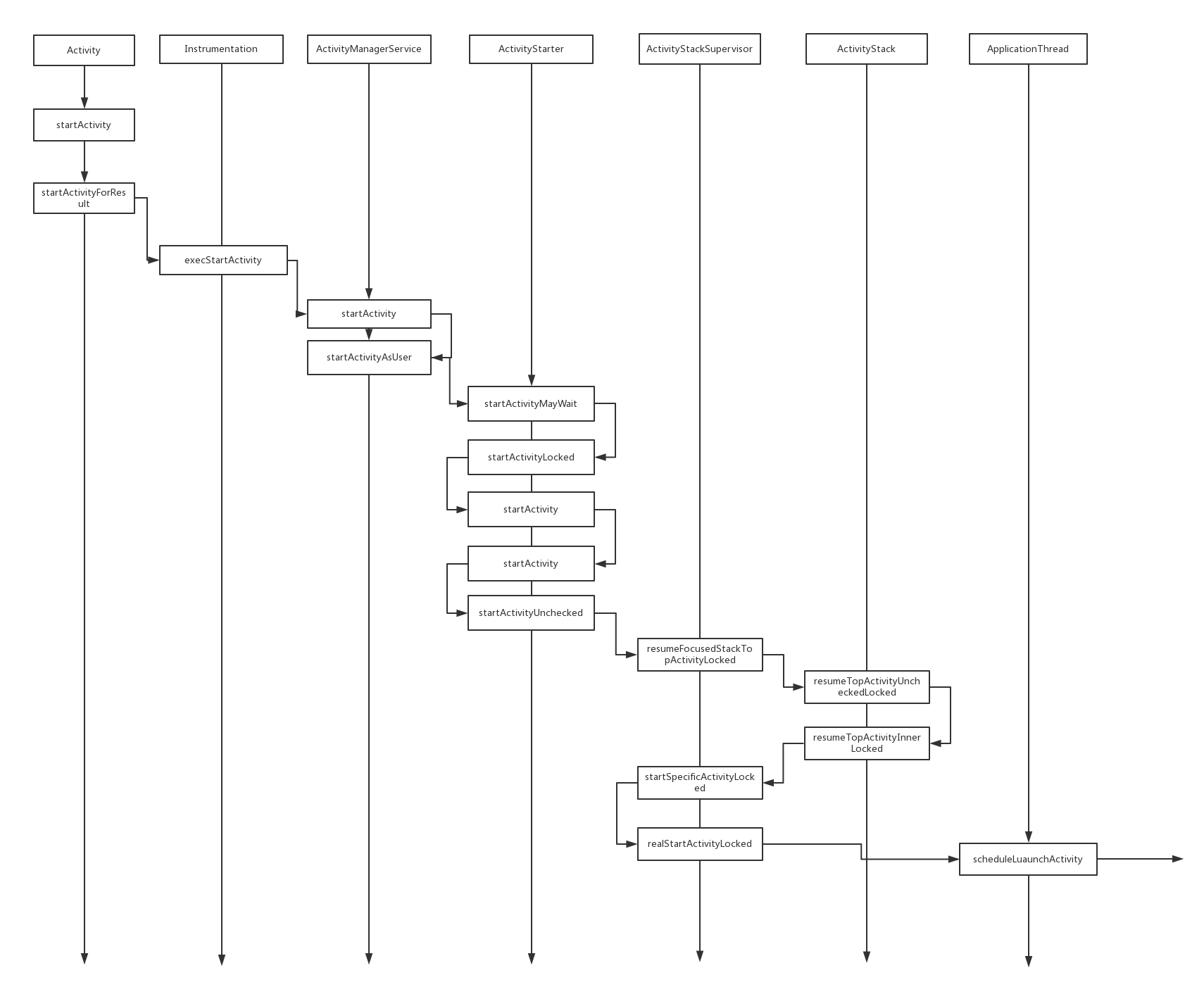
singleTask

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

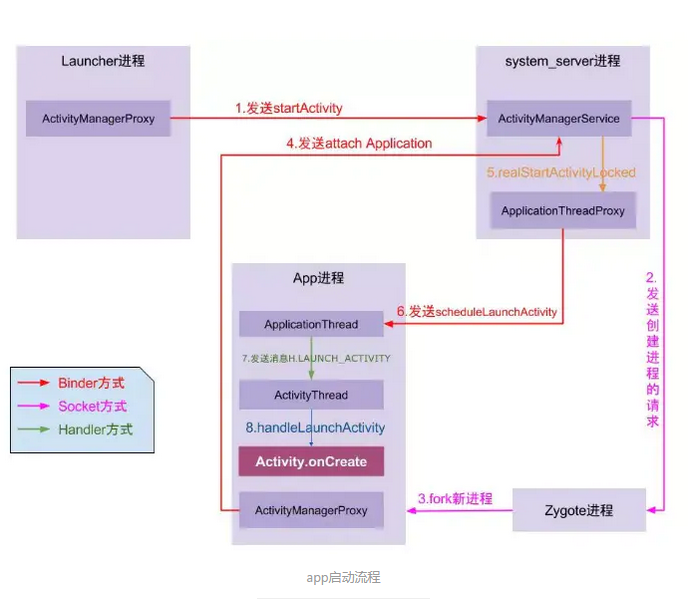


1. execStartActivity里获取跨进程服务，ActivityManagerService类，然后Activity的启动就转移到AMS中；
2. AMS把启动转移到了ActivityTaskManagerService(ATMS)中了，ATMS用于管理Activity及其容器(任务、堆栈、显示等)的系统服务；
3. 启动Activity的操作从客户端 跨进程 转移到 AMS，AMS通过ActivityTaskManagerService、ActivityStarter、ActivityStack、ActivityStackSupervisor 对 Activity任务、activity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



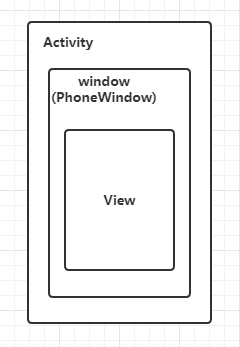
①App发起进程：当从桌面启动应用，则发起进程便是Launcher所在进程;当从某App内启动远程进程，则发送进程便是该App所在进程。发起进程先通过binder发送消息给system\_server进程;

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)

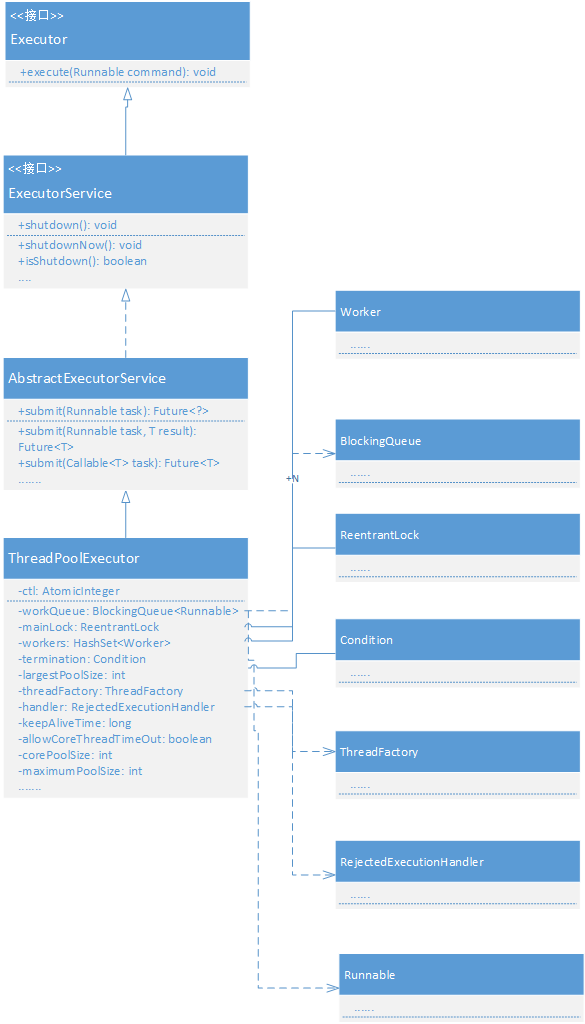
1. App应用内广播

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**

# JAVA方面

1. **JAVA锁**
2. **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**

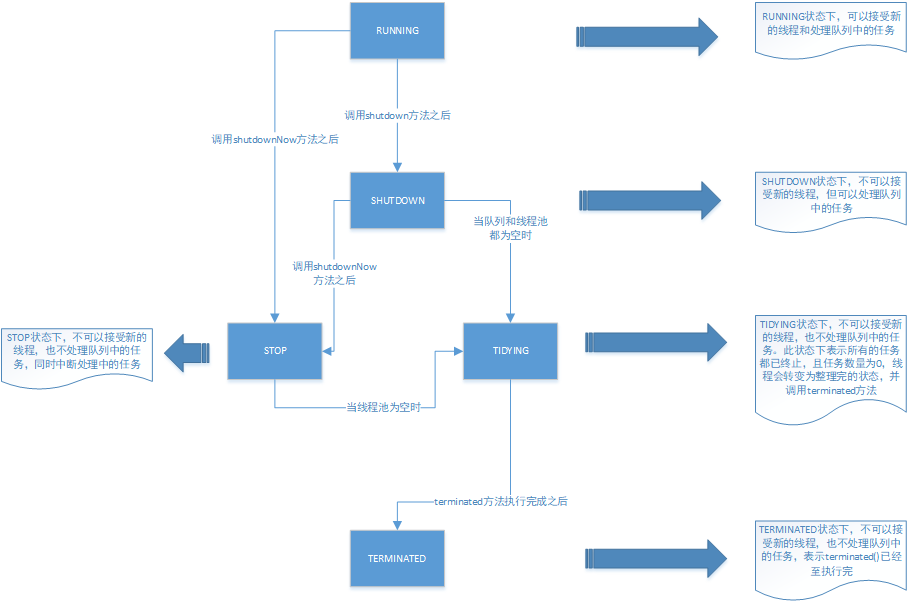
线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

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**

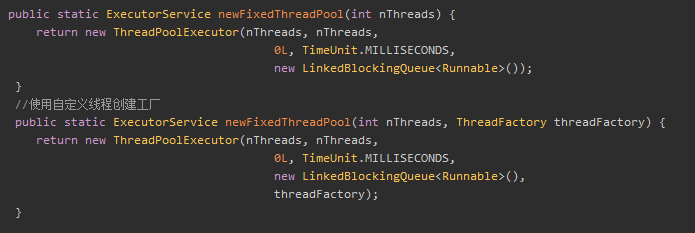
状态说明：线程池彻底终止，就变成TERMINATED状态。

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**

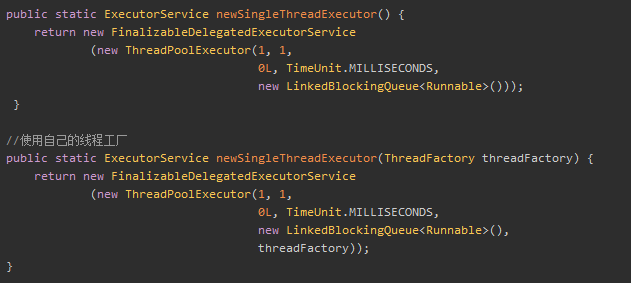
**newFixedThreadPool（固定大小线程池）**

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：

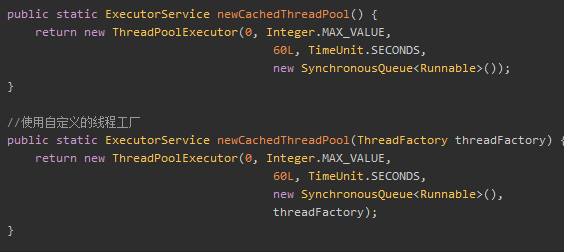


**newSingleThreadExecutor(单个后台线程）**

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：

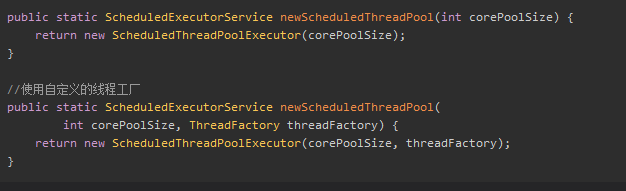
  
  
**newCachedThreadPool（无界线程池，可以进行自动线程回收）**

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：



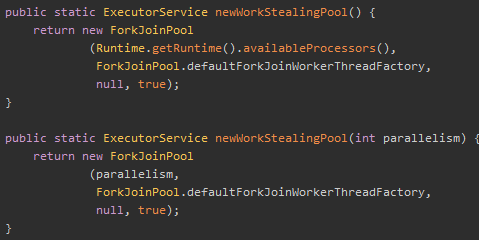
**newScheduledThreadPool (可调度)**

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？