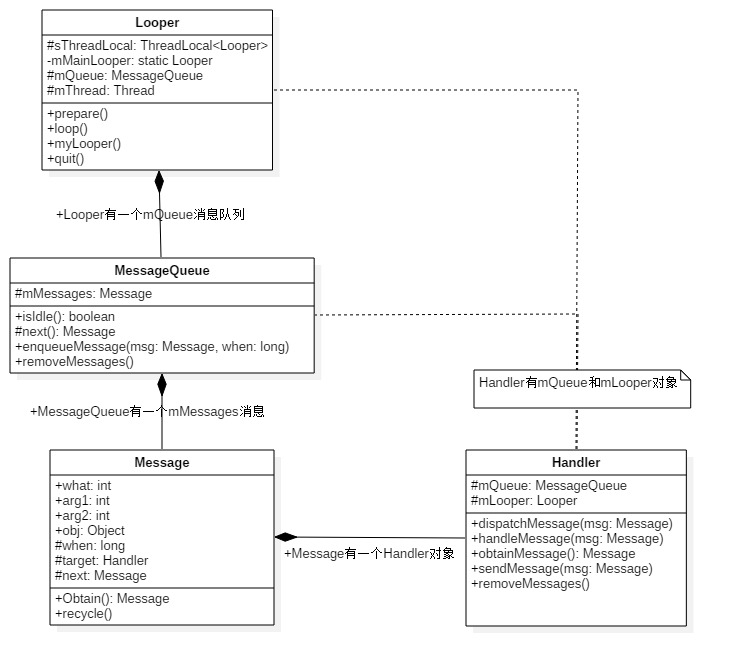
handler机制 :

* handler、mssage、massageQueue、Looper
* handler : 消息辅助类, 主要是发送消息和处理消息
* messgae : 单纯的一个消息
* MessageQueue : 消息队里 , 接收消息 和 处理消息
* Looper : 核心函数loop , 消息循环
* 总结 :
  + 一个线程对应一个Looper,
  + 一个Handler对应一个Looper和一个MessageQueue,
  + 一个Looper对应一个MessageQueue,
  + 一个MessageQueue对应一个Message
  + 一个Message里面有一个用于处理消息的Handler
* 总结 :
  + Handler通过sendMessage()发送Message到MessageQueue队列；
  + Looper通过loop()，不断提取出达到触发条件的Message，并将Message交给target来处理；
  + 经过dispatchMessage()后，交回给Handler的handleMessage()来进行相应地处理。
  + 将Message加入MessageQueue时，处往管道写入字符，可以会唤醒loop线程；如果MessageQueue中没有Message，并处于Idle状态，则会执行IdelHandler接口中的方法，往往用于做一些清理性地工作。
* handler的内存泄漏 ? <http://blog.csdn.net/zhuanglonghai/article/details/38233069>
  + handler若在主线程创建, 并且handlermessage了, 则必须声明为static , 否则被当前aticivty持有, 主线程的looper的生命周期跟当前应用一致, 会导致整个activity被hold住不能回收.
  + 如何处理 ?
    - static
    - 对activity的弱引用
  + 

Activity :

* Activity基础之生命周期
  + 启动Activity - onCreate()—>onStart()—>onResume()
  + Home键退居后台 - onPause()—>onStop()，进入停滞状态 ; 注 : Activity退居后台，且系统内存不足，系统会杀死这个后台状态的Activity，若再次回到这个Activity,则会走onCreate()–>onStart()—>onResume()
  + Activity返回前台： onRestart()—>onStart()—>onResume()，再次回到运行状态
* Activity状态保存 : onSaveInstanceState() 和 onRestoreInstanceState()
  + 当应用遇到意外情况（如：内存不足、用户直接按Home键 \ 屏幕旋转等）由系统销毁一个Activity，onSaveInstanceState() 会被调用
  + onSaveInstanceState() 保存成功 , 对应会去调用onRestoreInstanceState()
* Activity的launchMode
  + standard : 不管有没有在栈中, 都会生成一个新的实例
  + singleTop : 如果发现有对应的Activity实例正位于栈顶，则重复利用，不再生成新的实例
  + singleTask : 栈中有实例, 则将其置于栈顶, 但之前覆盖在其上面的会统统移除掉
  + singleInstance : 一个全新的栈, 并且不会有其他activity实例
* Activity的lanchMode :
  + Standard : 标准模式, 无论栈中是否存在, 统一new一个新的实例于栈中
  + SingleTop : 如果栈顶是该activity,则直接复用, 否则, 在栈中new一个全新的activity
  + SingleTask : 如果栈中有该activity, 则移动该activity至栈顶, 并且移除其 置于其上的所有activity实例
    - 首页的activity一般是SingleTask
  + SingleInstance : 在一个全新的栈里面创建该实例, 一般用于聊天界面

ClassLoader :

* ClassLoader是干嘛的 ?
  + java中所有的类, 必须被装载到jvm中才能运行, 这个装在工具是由jvm中的类装载器完成的,类装载器所做的工作实质是把类文件从硬盘读取到内存中,jvm在加载类的时候, 都是通过ClassLoader的loadClass() 方法来加载class的.
    - 总结, JVM加载类的时候, 是通过ClassLoader的loadClass()方法来加载class的.
* 什么是ClassLoader的双亲委派机制 ?
  + 某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，**依次递归**，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载
* 总结 :
  + java中的类大致分为三种 : 系统类 , 扩展类, 程序员自定义的类
  + 类的装载方式 :
    - 1.隐式装载 : 程序在运行过程中当碰到通过new等方式生成对象时, 隐式调用类装载器加载对应类到JVM中
    - 2.显式装载 : 通过Class.forName()等方法,需要显式加载需要的类
  + 类加载的动态性体现 :
    - 一个程序总是由n多个类组成, java程序启动时, 并不是一次把所有的类全部加载后再运行, 它总是先把保证程序运行的基础类一次性加载到JVM中, 其他类等到JVM用到的时候再加载, 这样的好处是节省了内存的开销
  + Java类装载器 :  ClassLoader只是一个抽象类,
    - BootstrapLoader - 负责加载系统类 (这个是C++来实现的, Java里面调不了)
      * 负责装载JRE的核心类库
    - ExtClassLoader - 负责加载扩展类
      * 负责装载JRE扩展目录ext下的jar类包
    - AppClassLoader - 负责加载应用类
      * 负责装载classpath路径下的类包, 默认情况下使用AppClassLoader装载应用程序的类
  + Java类加载器如何协调工作的 ?
    - Java采用类委托模型机制 :
      * 类装载器有载入类的需求时, 会先请示其parent使用其搜索路径帮忙载入, 如果Parent找不到, 那么才由自己依照自己的搜索路径搜索类
      * 为什么会用全盘委托机制 ?
        + 因为防止恶意基础类例如String加载到JVM将会引起严重的后果
  + 类装载器ClassLoader描述一下JVM加载class文件的原理机制 ?
    - java中类装载器把一个类装入JVM, 经过以下步骤 :
      * 1. 装载 : 查找和导入class文件
      * 2. 链接 : 其中解析步骤是可以选择
        + (a)检查 : 检查载入的class文件数据的正确性
        + (b)准备 : 给类的静态变量分配存储空间
        + (c)解析 : 将符号引用转成直接引用
      * 3.初始化 : 对静态变量 , 静态代码块执行初始化工作
* 总总总总结 :
  + 类装载工作是由ClassLoader和其子类负责. JVM在运行时会产生三个ClassLoader : 根装载器(BootstrapLoader,C++代码实现, Java调用不了) , 扩展类装载器(ExtClassLoader) 和 加载应用类(AppClassLoader)
* Java反射原理 :
  + 类由ClassLoader动态装载Class到JVM , JVM为每个类创建一个Class的实例 , 通过reflect包下的api能动态加载类,调用类方法,访问类的属性.

SharedPreferences :

* （/data/data/<包名>/shared\_prefs/）中
* 使用上commit在主线程中 , 会有ANR, 这个todo, 看下刘俊的分享 主要是anr的解决
* 有commit 和 apply 两种方式, 一种同步 , 一种异步
* 强烈建议不要在sp里面存储特别大的key/value, 有助于减少卡顿/anr
* 请不要高频地使用apply, 尽可能地批量提交;commit直接在主线程操作, 更要注意了
* 不要使用MODE\_MULTI\_PROCESS;
* 高频写操作的key与高频读操作的key可以适当地拆分文件, 由于减少同步锁竞争;
* 不要一上来就执行getSharedPreferences().edit(), 应该分成两大步骤来做, 中间可以执行其他代码.
* 不要连续多次edit(), 应该获取一次获取edit(),然后多次执行putxxx(), 减少内存波动; 经常看到大家喜欢封装方法, 结果就导致这种情况的出现.
* 每次commit时会把全部的数据更新的文件, 所以整个文件是不应该过大的, 影响整体性能;
* 为了避免本地SharedPreference明文可见, 可使用AES对称加密 对 缓存的文字进行加密,
  + 从SharedPreferenceImpl源码来看 , EditorImpl都是Final类型, 因此, 加密过程只能实在读写参数的时候, 因此无需额外封装

（1）每次调用getSharedPreferences时都会创建一个SharedPreferences对象吗？这个对象具体是哪个类对象？

* 这里回答第一个问题，对于一个相同的SharedPreferences name，获取到的都是同一个SharedPreferences对象，它其实是SharedPreferencesImpl对象。

（2）在UI线程中调用getXXX有可能导致ANR吗？

* 这里回答第二个问题，在UI线程中调用getXXX可能会导致ANR。同时可以回答第三个问题，SharedPreferences只能用来存放少量数据，如果一个SharedPreferences对应的xml文件很大的话，在初始化时会把这个文件的所有数据都加载到内存中，这样就会占用大量的内存，有时我们只是想读取某个xml文件中一个key的value，结果它把整个文件都加载进来了，显然如果必要的话这里需要进行相关优化处理

（3）为什么SharedPreferences只适合用来存放少量数据，为什么不能把SharedPreferences对应的xml文件当成普通文件一样存放大量数据？

（4）commit和apply有什么区别？

* 这里就可以回答第四个问题了，commit的写操作是在调用线程中执行的，而apply内部是用一个单线程的线程池实现的，因此写操作是在子线程中执行的

（5）SharedPreferences每次写入时是增量写入吗？

* 这里回答第五个问题，SharedPreferences每次写入都是整个文件重新写入，不是增量写入内存中维护的mMap是增量添加，文件本身每次都全量写入的

AsyncTask :

* 在activity生命周期结束的时候, 如果未关闭AsyncTask, 回报异常, 并且不会执行onPostExecute
* AsyncTask的内部实现 ?
  + 使用Callable + Future 用于获取结果
  + progress + result , 是使用handler去发送结果的
  + 内部使用的线程池 , 并且串行队列, 可以达到先入先出的数组
* AsyncTask的缺点 :
  + 默认的拒绝策略, 和128长度的线程池 , 如果处理大量的任务, 会导致进程直接崩溃
  + 内部是默认串行机制, 如果并行的, 因为doInbackground()访问了他公共资源, 未作同步, 所以gg
  + 必须在主线程初始化, 因为默认的IntentHandler是Looper是主线程的
  + 内存泄漏, 内部会持有外部引用
  + cancel不一定能doInBackground会执行到结束, 只是不执行onPostExecute

Android 线程间通信的几种方式 ?

* 共享内存 volatile
* 文件, 数据库
* handler
* wait, notify, notifyall

Service :

* service运行在主线程
* 问service之startForeground,stopForeground ?
  + 设置前台服务, 因为游戏中心下载的服务需要设置前台服务,后台服务容易被系统杀死
* service的生命周期(有两种) :
  + 常规的startService() 的生命周期函数调用 :
    - onCreate() -> onStartCommand() ps : 这个函数只要调用startService(), 就会call这个函数 -> onDestroy() ; 调用stopSelf() 关闭 或者 context.stopservice()关闭;
  + bindService()的生命周期函数调用 :
    - context.bindService() → onCreate() → onBind() ps : 这里返回的是一个Ibinder用于跨进程通信  → onStartCommand()  → onUnbind() → onDestroy()

IntentService 和 service :

* IntentService : 专门用来处理异步请求的 ; 串行队列
  + 内部原理 : 内部有一个HandlerThread工作线程 , ServiceHandler有一个Looper,不断轮寻读取任务,然后交给onHandleIntent去做耗时任务

HandlerThread原理 :

* 内部拥有Looper, 可以通过Looper构造handler, 与其他线程进行通信.
* 退出的时候, 必须调用quit()函数, 安全退出.

进程优先级 :

* 前台进程(Foreground process)
* 可见进程(Visible process)
* 服务进程(Service process)
* 后台进程(Background process)
* 空进程(Empty process)

Dalvik和JVM的区别 ? class和dex文件的区别 ?

* dvm基于寄存器的虚拟机, jvm基于虚拟栈的虚拟机, dalvik存取速度比jvm块
* android编译完生成-class文件,dex工具把class文件处理成dex文件. 去除class的冗余信息, 效率高

Object的相关方法 ?

* equals 和 == :
  + 如果没有重写equals,equals 和 == 都是同一个含义 , 判断内存地址是否一致
  + 重写后就是按照我们的需求进行比较, 比如内容相等
  + equals相等, 则hashcode一定相当, hashcode相等, equals则不一定相同
* 为什么重写equals 一定要重写hashCode
  + 效率 , 先比较hashcode, 如果一致则继续比较equals, 否则就不同
  + hashcode就是对象的储存地址
* clone : 进行对象拷贝
* getClass返回和当前对象相关的Class对象
* notify,notifyall,wait都是用来对给定对象进行线程同步的
* finalize() :
  + finalize()函数是在JVM回收内存时执行的，但JVM并不保证在回收内存时一定会调用finalize()
  + 如果对象重写了finalize方法, 且方法体不为空, 则会封装成finalizer对象,并添加到finalizer链表.finalizer类会初始化一个低优先级的线程,从引用队列获取finalizer对象,最终调用finalize方法,在c++层.GC时,由于抢占布道资源, 所以有可能不被触发.

String和CharSequence区别, StringBuilder和StringBuffer的区别 :

* StringBuilder - 非线程安全
* StringBuffer - 线程安全 , 线程安全的实现是通过synchronized关键字实现的

android异步的实现 :

* AsyncTask, HandlerThread, Thread, IntentService

compileSdkVersion , minSdkVersion , targetSdkVersion :

* compileSdkVersion : 告诉 Gradle 用哪个 Android SDK 版本编译你的应用
  + 新的 compileSdkVersion 不会被包含到 APK 中：它纯粹只是在编译的时候使用
  + 总是使用最新的 SDK 进行编译。在现有代码上使用新的编译检查可以获得很多好处，避免新弃用的 API ，并且为使用新的 API 做好准备
  + Support Library 要与 compileSdkVersion版本一致
* minSdkVersion : minSdkVersion 则是应用可以运行的最低要求 ,
  + 如果设备的sdkversion 小于这个值, 则不能安装
* targetSdkVersion 是 Android 提供向前兼容的主要依据,将 target 更新为最新的 SDK 是所有应用都应该优先处理的事情
  + 兼容android 版本在应用中表现的新特性, 例如 运行时权限等

RxJava线程池的io() 和 computation()线程池 :

* computation : newFix 核心线程数为CPU核数的有界线程池 , 这个会重用线程资源,  不用的时候就挂起
* io : newCachedThreadPool , 核心线程数为1, 无界的线程池,  这个过了alive时间就会被回收掉, 并且会不断新建线程对象

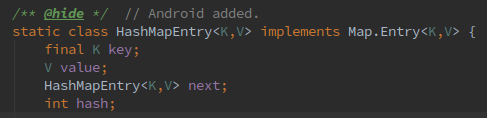
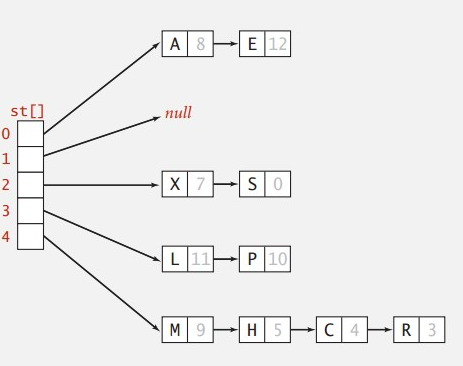
Okhttp的addInterceptor 和 addNetworkInterceptor :

* addInterceptor : (应用拦截器)
  + 不需要担心中间过程的响应, 如重定向和重试
  + 总是只调用一次,即使HTTP响应是从缓存中获取
  + 观察应用程序的初衷, 不关心OkHttp注入的头信息 : If-None-Match
  + 允许短路而不调用Chain.proceed(), 即中止调用
  + 允许重试,使Chain.proceed()调用多次
* addNetworkInterceptor : (网络拦截器)
  + 能够操作中间过程的响应, 如重定向和重试
  + 当网络短路而返回缓存响应时不被调用
  + 只观察在网络上传输的数据
  + 携带请求来访问链接

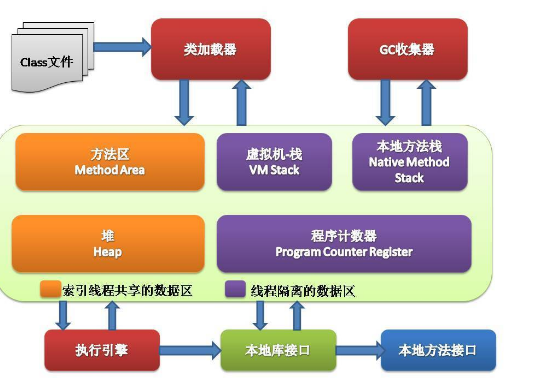
Bitmap的原理 :

* Bitmap的创建 :
  + 最终还是调用的 nativeCreate ,
  + native层创建Bitmap - SKBitmap
  + mPixelStorage保存前面创建Heap对象的弱引用，mPixelRef指向WrappedPixelRef。
  + outBitmap拿到mPixelRef强引用对象，这里理解为拿到SkBitmap对象.
  + 创建Java层Bitmap对象，把该包的包上。
  + native层是通过JNI方法，在Java层创建一个数组对象的，这个数组是对应在Java层的Bitmap对象的buffer数组，所以pixels还是保存在Java堆。
  + 而在native层这里它是通过weak指针来引用的，在需要的时候会转换为strong指针，用完之后又去掉strong指针，这样这个数组对象还是能够被Java堆自动回收。
  + 里面jstrongRef一开始是赋值为null的，但是在bitmap的getSkBitmap方法会使用weakRef给他赋值
* Bitmap的回收 :
  + 一是Java层主动call recycle()方法或者Bitmap析构函数都会调用freePixels，移除Global对象引用，这个对象是Heap上存一堆像素的空间。GC时释放掉。
  + 二是，JNI不再持有Global Reference，并native函数执行后释放掉，但Java层的Bitmap对象还在，只是它的mBuffer和mNativePtr是无效地址，没有像素Heap的Bitmap也就几乎不消耗内存了。至于Java层Bitmap对象什么时候释放，生命周期结束自然free掉了

数据结构 :

* HashMap底层数据结构 , 如何处理冲突 ?
  + HashMap的底层是由数组(Entry[]) + 单链表(Entry) 来实现的, 采用 "拉链法"实现的.除了拉链法解决冲突，还有线性探测法、伪随机数法等. 在1.8, 当链表过长时, 会转化为红黑树, 增加查询的效率
    - "拉链法"的延伸 :
      * 当命中到同一位置之后, 会以链表的形式向后面链接
      * 默认加载因子是 0.75, 这是在时间和空间成本上寻求一种折衷, 增加命中效率
* HashMap, HashSet, HashTable的区别 :
  + 从线程安全来说 :
    - HashTable是线程安全的 , 主要是通过synchronized来实现的
    - HashMap和HashSet是非线程安全的
  + 是内部实现来说 :
    - HashSet 是内部实现 , 是基于HashMap的
    - HashTable虽然也是散列表, 通过拉链法解决"hash冲突",同样也是put<key, value>但跟HashMap还是有以下区别 :
      * 遍历实现方式 : HashMap只支持Iterator, HashTable则支持Enumeration, Iterator
      * HashMap的key, 使用了"hash算法" ; 但是HashTable的key,则仅仅是Object.hashCode()
        + "hash算法" 的实现 是采用了 模运算 + 扰动函数
* Set 和 List的区别, Set如何处理重复数据的?
  + HashSet如何保证不重复的数据的?
    - HashSet的add()函数调用了HashMap的put, 先比较hashCode, 在比较equals函数, 因此一个Object需要重写hashCode和equals这两个函数
* 数组 和 链表 ?
  + 数组 : 占用内存严重, 空间复杂度大. 寻址容易, 但插入和删除困难
  + 链表 : 存储区间离散, 占用内存宽松, 空间复杂度小, 但时间复杂度很大. 寻址困难, 插入和删除容易
* update 2018.2.26
  + 数组静态分配内存，链表动态分配内存；
  + 数组在内存中连续，链表不连续；
  + 数组元素在栈区，链表元素在堆区；
  + 数组利用下标定位，时间复杂度为O(1)，链表定位元素时间复杂度O(n)；
  + 数组插入或删除元素的时间复杂度O(n)，链表的时间复杂度O(1)。
* HashMap 内部实现 ?
  + 散列表的实现
    - 什么是散列表？
      * 可以通过关键字key直接访问到内容value的一种数据结构, 一个value对用多个key值,一个key只能对应一个value
      * 
  + HashMapEntry ,  单链表 + 数组, 单链表存储的key,value
  + Hash冲突是什么 ?
    - 两个不想等的key产生了相同的hash值
  + 拉链法解决是什么意思 ?
    - 
* HashTable的实现 ?
  + 单链表 + 数组 (组成双链表) , put, remove都是synchronized, 因此效率不如concurrentHashMap
  + 也是通过拉链法解决hash冲突的
  + key.value都不为null, 但HashMap是可以的
  + hash是直接调用的object.hashCode();
* ConcurrentHashMap ?
  + 为什么不用Hashtable和Collections.synchronizedMap(hashMap)
    - 因为是对读写进行加锁操作，一个线程在读写元素，其余线程必须等待，性能可想而知
  + 1.6版本 :
    - 分段锁的机制，实现并发的更新操作，底层采用数组+链表+红黑树的存储结构
      * 单词解释 :
        + Segment : 继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶
        + HashEntry : 用来封装映射表的键 / 值对
        + 每个桶是由若干个 HashEntry 对象链接起来的链表
  + 1.8版本 :
    - 抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用node数组+链表+红黑树的存储结构,node代表key,value
  + 为什么使用红黑树 ?
    - 把链表转化为红黑树，提高遍历查询效率
    - 在1.8的实现中，当一个链表中的元素达到8个时，会调用treeifyBin()方法把链表结构转化成红黑树结构
    - 红黑树 :
      * 相对接近平衡的二叉树
      * 每个节点要么红色，要么是黑色
      * 根节点一定是黑色的
      * 每个空叶子节点必须是黑色的
      * 如果一个节点是红色的，那么它的子节点必须是黑色的
      * 从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点
  + 总结 :
    - ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新
  + size()方法 :
    - 1.7的实现 : (不准确的)
      * 先采用不加锁的方式，连续计算元素的个数，最多计算3次：  
        1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；  
        2、如果前后两次计算结果都不同，则给每个Segment进行加锁，再计算一次元素的个数；
    - 1.8的实现 :
      * 使用一个volatile类型的变量baseCount记录元素的个数
      * 元素个数保存baseCount中，部分元素的变化个数保存在CounterCell数组中
      * 通过累加baseCount和CounterCell数组中的数量，即可得到元素的总个数
* java同步之 - AQS是什么 ?
  + AQS内部会保存一个state变量,是通过CAS修改该变量的值, 修改成功表示获取到了该锁,没有修改成功,就表示是state已经是加锁状态了,则添加到等待队列中
* java同步之 - CAS自旋锁 : (atmoic相关类的实现都是自旋锁)
  + 含义 : 一个线程获得锁未释放, 另一个线程不是进入block阻塞状态, 而是进入忙等.
  + 精髓就是,V-内存值, A-预期值, B-修改值;
  + CAS函数有4个参数,1修改的对象,2要修改变量的偏移量,3修改前的值,4预想修改后的值
  + java.util.concurrent.atomic包下的原子操作类都是基于CAS实现的
  + Unsafe是CAS的核心类, atomic原子操作就是调用的该类Unsafe.CAS方法
  + 缺点 :
    - 占据过多的CPU时间, 如果一个锁长时间不释放, 另一个线程会长时间占用CPU
    - 死锁 : 同一个线程两次试图获取自旋锁 , 自己等待自己shifang
    - ABA问题, 初次读取是A,赋值的时候仍是A,那他扔有可能被改变过, 需要带有标记的原子引用类"AtomicStampedReference",它可以保证CAS正确性
* java同步之 - synchronized :
  + 锁的四种状态 :
    - 无锁状态，
    - 偏向锁状态，
    - 轻量级锁状态，
    - 重量级锁状态
  + 用法 :
    - 普通同步方法, 锁对象
    - 静态同步方法, 锁类
    - 同步代码块 , 锁的是括号内的对象
  + 互斥性 :
    - 表示 : 一个线程处于锁状态的情况下, 其他线程试图获取该锁会被阻塞住, 当一个线程释放锁的时候, 会唤醒其他线程
  + 内存可见性 :
    - 线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中
    - 线程获取锁时，JMM会把该线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须从主内存中读取共享变量
* java同步之 - volatile :
  + volatile 特性 :
    - 内存可见性 : 一个线程的修改对其他线程是可见的
    - 在某些特定场景, 也相当于一个轻量级的sychronize, 但必须满足 :
      * 对变量的写不依赖当前值, a++
  + volatile使用场景 :
    - 状态量标记 , boolean值, 但如果设计原子性操作, 但还是需要atomicBoolean, 但性能会有所损耗, 这个得看具体的场景
    - double check 线程安全的单例 ,需要对变量可见
  + volatile可见性 是基于内存屏障实现的.
    - 在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序,
    - JMM为了保证在不同的编译器和CPU上有相同的结果，通过插入特定类型的内存屏障来禁止特定类型的编译器重排序和处理器重排序
* java同步之 - ReentrantLock :
  + 实现了公平锁,非公平锁
  + 定时, 可中断的锁等待
  + 非块结构的加锁
  + 配合condition更加灵活
* JUC集合 - ConcurrentHashMap
  + 为什么不用Hashtable和Collections.synchronizedMap(hashMap)
    - 因为是对读写进行加锁操作，一个线程在读写元素，其余线程必须等待，性能可想而知
  + 1.6版本 :
    - 分段锁的机制，实现并发的更新操作，底层采用数组+链表+红黑树的存储结构
      * 单词解释 :
        + Segment : 继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶
        + HashEntry : 用来封装映射表的键 / 值对
        + 每个桶是由若干个 HashEntry 对象链接起来的链表
  + 1.8版本 :
    - 抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用node数组+链表+红黑树的存储结构,node代表key,value
  + 为什么使用红黑树 ?
    - 把链表转化为红黑树，提高遍历查询效率
    - 在1.8的实现中，当一个链表中的元素达到8个时，会调用treeifyBin()方法把链表结构转化成红黑树结构
    - 红黑树 :
      * 相对接近平衡的二叉树
      * 每个节点要么红色，要么是黑色
      * 根节点一定是黑色的
      * 每个空叶子节点必须是黑色的
      * 如果一个节点是红色的，那么它的子节点必须是黑色的
      * 从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点
  + 总结 :
    - ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新
  + size()方法 :
    - 1.7的实现 : (不准确的)
      * 先采用不加锁的方式，连续计算元素的个数，最多计算3次：  
        1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；  
        2、如果前后两次计算结果都不同，则给每个Segment进行加锁，再计算一次元素的个数；
    - 1.8的实现 :
      * 使用一个volatile类型的变量baseCount记录元素的个数
      * 元素个数保存baseCount中，部分元素的变化个数保存在CounterCell数组中
      * 通过累加baseCount和CounterCell数组中的数量，即可得到元素的总个数

JVM :

* 堆内存 : 用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中
* 栈内存 : 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存
* 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问
* JVM运行时的数据分区 :
  + 程序计数器
  + JVM栈
  + 堆内存
  + 方法区
  + 运行时常量
  + 本地方法栈
* 线程私有 :
  + 程序计数器 :
    - 记录当前正在执行的指令
  + JVM栈 :
    - 栈内存的大小可以设置
    - StackOverFlowError : 栈内存设定为固定值, 超出这个阀值就会抛出此异常
    - OutOfMemoryError : 栈内存动态增长, 当JVM申请的内存大小超过了可用的内存大小就会抛出此异常
  + 本地方法栈 :
    - C栈
* 多个线程共享 :
  + 堆内存 :
    - 垃圾回收 作用于此数据区
  + 方法区 :
    - 方法区中存放类的信息，比如类加载器引用，属性，方法代码和构造方法和常量
  + 运行时常量池 :
    - 一个类或者接口的class文件中常量池表（constant\_pool table）的运行时展示形式
* large heap :
  + android 配置文件路径 : /system/build.prop
  + 开启和关闭的区别 :
    - jvm里面有heapgrowthlimit=192m这个参数
      * 开启 不崩溃, 到heapsize=512m才崩溃
      * 关闭, 到192m, 直接崩溃
* JVM内存模型 :
  + Java堆 : 永久代, 老年代, 新生代 (edon区, survior区(from,to)) eden , from ,to大小默认是8:1:1 , 新生代老年代默认比例是8:1
  + java栈 : 执行java字节码
  + 方法区 :PermGen(存放java类的相关信息) , CodeCache(JIT编译后的本地代码)
  + 本地方法区 : 执行native 方法
  + 程序计数器 : 当前线程执行的字节码的信号指示器
* 判断对象是否能够被回收的两个算法 :
  + 引用计数算法 :
    - java虚拟机没有选用 "引用计数算法" 来管理内存, 主要原因是它很难解决对象之间相互循环引用的问题.
    - 因此 "引用计数算法" 来作为GC回收的临界条件是不准确的
  + 可达性分析算法 :
    - GC Roots作为起点, 查看引用链. 当一个对象GC Roots没有任何引用链的时候, 则会判定为是可回收的对象.
    - 想想leak canary判定内存泄漏
* 垃圾搜集算法 :
  + 标记-清除法 : 效率不高, 产生大量不连续的空间碎片
  + 标记-整理算法 : 产生了连续的空间
  + 复制算法 : 存活的对象复制到另一块, 大大降低了内存空间使用率, HotSpot年青代的回收算法就是如此
  + 分代收集算法 : 当前商业虚拟机都是用的这种方式 , 新生代是复制算法, 老年代是标记整理算法 , 并且比例默认是8:1, 但这个参数可以通过JVM配置进行设置
* G1收集器 :
  + 并行与并发 , 与CMS一样
  + 分代收集
  + 启动后不需要请求更大的java堆, 空间整合: G1运作期间不会产生内存空间碎片
  + 可预测的GC停顿
  + 是一款面向服务端的垃圾收集器,
    - jvm启动的时候, 划分region区域 , 不能超过32mb, 否则会存在问题
    - 这些region区域被映射为 : eden, survivor, old
  + 回收过程 : 初始标记 - 并发标记 - 最终标记 - 筛选回收
* JVM相关 :
  + JVM内存区的图 :
  + 
  + JVM - 内存区 : 划分为
    - 线程共享的数据区 : 堆内存(所有对象的实例都在这分配内存), 方法区(已经被虚拟机加载过来的类信息，常量，静态变量等) - 还包括一个运行常量池, 用于ClassLoader动态加载过来新的常量信息
    - 线程隔离的数据区 : 虚拟机-栈(执行java字节码), 程序计数器(当前线程所执行的字节码的信号指示器), 本地方法栈(执行native方法)
    - update 171222
      * 方法区 : 主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在
      * 栈区 : 方法体内的局部变量都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放
      * 堆区 : 又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收
  + JVM的实现 - HotSpot虚拟机
  + 对象的创建 :
    - a. 先执行 方法区(常量池) - 检索这个类是否由ClassLoader加载进来了, 如果没有的话, 进行类的加载
    - b. 虚拟机对新生对象分配内存,
    - c. 虚拟机对对象进行必要的设置, 例如hash码和GC分代信息
    - d. 最后执行对象init方法, 对对象进行初始化
  + 对象的访问和定位 :
    - 句柄池
    - 直接指针 , JVM是使用的这个方式.  节约指针定位的内存开销
  + 垃圾收集算法 :
    - 标记-清除法 : 效率不高, 产生大量不连续的空间碎片
    - 标记-整理算法 : 产生了连续的空间
    - 复制算法 : 存活的对象复制到另一块, 大大降低了内存空间使用率, HotSpot年青代的回收算法就是如此
    - 分代收集算法 : 当前商业虚拟机都是用的这种方式 , 新生代是复制算法, 老年代是标记整理算法 , 并且比例默认是8:1, 但这个参数可以通过JVM配置进行设置
  + 垃圾收集器 :
    - Serial/Serial Old : 一个运行在年轻代 , 一个运行在老年代 , 会暂停用户线程 , 直到搜集完成
    - ParallelNew : Serial收集器的多线程版本 , 同样也会暂停用户线程, 直到搜集完成, 但如果CPU少的, 会有额外的线程交互开销
      * 多线程各做各的事情(互相无共享状态)
    - Parallel Scavenge/Parallel Old : 多线程收集器, 一个作用与新生代 ,一个作用于老年代. 他关注一个可控的吞吐量 , 需要我们控制好回收次数, 避免内存抖动
      * 多线程各做各的事情(互相无共享状态)
    - Concurrent Mark Sweep : 并发使用标记-清除算法的垃圾收集器．　但他还是会轻微的阻塞用户线程，　所以一定要控制好内存, 防止平凡的GC；初始标记 -> 并发标记 -> 重新标记 -> 并发清理这四个过程
      * 多线程都只做同一件事情,(有共享状态)
    - Garbage-First : 是一款面向服务端的垃圾收集器, 面向大内存、多核系统的收集器,高吞吐量的短暂停时间的收集器.G1将堆分为很多大小相等的Region,在java1.9会替换掉CMS.将Heap分为大小相等的Region，逻辑分代，Marking的大部分是并发的，STW中大部分采取多线程并行执行，采用Copying进行多线程并行收集
* ART和Dalvik的区别 :
  + 程序在运行的时候 , Dalvik是依靠JIT编译器去解释字节码, 编译后的应用代码通过一个解释器在用户的设备上运行 , 让应用能在不同硬件和架构上运行
  + ART应用安装的时候就预编译字节码到机器, AOT编译机制, 移除了解释代码这一过程, 应用将更加高效.
* JMM内存模型 :
  + 重排序
  + 内存屏障
  + happens-before
  + as-if-serial - 不管怎么重排序，程序的执行结果不能被改变
  + 顺序一致性内存模型 - 它是理想化的内存模型 ；一个线程中的所有操作必须按照程序的顺序来执行 ; 在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见 ;
  + JMM - Java内存模型。它是Java线程之间通信的控制机制
  + 可见性 - 可见性一般用于指不同线程之间的数据是否可见
  + 原子性 - 是指一个操作是按原子的方式执行的。要么该操作不被执行；要么以原子方式执行，即执行过程中不会被其它线程中断
  + volatile - 则对该变量的读写就将具有原子性
    - volatile和 synchronize对比 :
      * 在功能上，监视器锁比volatile更强大；在可伸缩性和执行性能上，volatile更有优势。
      * volatile仅仅保证对单个volatile变量的读/写具有原子性；而synchronize锁的互斥执行的特性可以确保对整个临界区代码的执行具有原子性
  + JMM如何实现锁 :
    - 公平锁 - 公平锁是通过“volatile”实现同步的 , 对state的写入
    - 非公平锁 - 通过CAS实现的，CAS就是compare and swap。CAS实际上调用的JNI函数，也就是CAS依赖于本地实现。以Intel来说，对于CAS的JNI实现函数，它保证：(01)禁止该CAS之前和之后的读和写指令重排序。(02)把写缓冲区中的所有数据刷新到内存中。
  + final - final域的读取总是在初始化后, 该过程不能重排序
* JVM相关 :
  + JVM的内存模型 :
    - java堆:存放对象,所有对象的创建都在堆上申请内存，并被GC管理；
    - java栈:用于方法之间的调用，进栈出栈的过程
    - 方法区:
      * PermGen: 存放java类的相关信息 , 如静态变量\成员方法和抽象方法
      * CodeCache: JIT编译之后的本地代码
    - 本地方法栈:用于方法之间的调用，进栈出栈的过程
    - PC寄存器
  + JVM的启动流程 :
    - 1. 配置JVM装载环境 :
      * JVM.dll文件的查找 , 装载
    - 2. 虚拟机参数设置 :
      * 新生代大小, 堆的最小,最大值
    - 3. 设置线程栈大小
    - 4. 执行java main方法
    - 1. 新建jvm实例
    - 2. 加载主类的class
      * java运行方式有两种 : jar方式 和 class方式
    - 3. 查找main方法
    - 4. 执行main方法
  + JVM堆内存区域(包括新生代 , 老年代 , 永久代) :
    - java堆是被所有线程共享的一块内存区域 , 所有对象和数组都在堆上进行内存分配
      * 新生代 :
        + eden区
        + survivor区 :

S0 和 S1

或者from和to

* + - * + 其中eden , from ,to大小默认是8:1:1
      * 老年代 :
      * 永久代 : 1.8后是metaSpace
    - 堆内存初始化过程 :
    - 初始化GC策略 :
      * 一般都是选择两代收集策略 , 新生代复制, 老年代清除-标记
      * 设置新生代,老年代,永久代设置的内存大小进行对齐调整
      * 设置新生代,老年代,永久代的容量 , 一般默认比例是8:1
    - 初始化分代生成器 :
      * 用途是每个生成器保存代签分代的GC算法,内存初始值和最大值
    - 初始化java堆管理器 :
      * GenCollectedHeap是整个Java堆的管理器，负责Java对象的内存分配和垃圾对象的回收
      * 如果当前的GC策略为ConcurrentMarkSweepPolicy,创建GC线程
* Java类的加载过程 : (主要是在内存中进行初始化)
  + HotSpot 对象模型 :
    - oop, klass model
  + 加载过程 : (说实话, 这个没看懂, 暂时这样说吧)
    - 通过classloader加载,读取对应的class文件的二进制数据到虚拟机中进行解析, 解析完成后输出一个底层为instanceKlass代表该class, 再为将要初始化的对象分配内存空间.
* java类的创建过程
  + 这个就说不清楚了, 没有下载opnejdk源码看这块
* 如何触发并执行GC线程 ?
  + VMThread负责调度执行虚拟机内部的JVM线程操作 , 例如GC
  + 其内部也是通过轮询的方式,按优先级调度线程队列 ,
  + 当新生代内存不足的时候, 会触发一次YGC
* GC怎么判断对象死亡 ?
  + 引用计数算法 : 它很难解决对象之间循环引用的问题, 因此用来作为GC回收的临界条件是不准确的
  + 可达性分析算法 : GC Roots没有任何引用链的时候, 则判断是可回收的对象
    - 问谁可以作为根节点对象 ?
      * 虚拟机栈中引用的对象
      * 本地方法栈中引用的对象
      * 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
* JVM垃圾回收几种算法 ?
  + 标记回收法 : 单线程, 会产生内存碎片
  + 标记-压缩回收法 : 先压缩存活的对象到一方, 在进行回收, 提高了内存利用率
  + 复制回收法 : 把现有内存空间分成两部分，gc运行时，它把可到达对象复制到另一半空间，再清空正在使用的空间的全部对象
  + 分代回收发：把内存空间分为两个或者多个域，如年轻代和老年代. 年轻代采用效率比较高的算法 , 老年代采用标记-压缩算法
* java程序, 编译 运行流程 :
  + 源码 -> 编译成包含字节码的class文件
  + class文件 -> 有classLoader在程序运行的时候装载到JVM虚拟机
  + JVM虚拟机再由JIT编译器 , 编译成本机可运行代码
  + 程序运行
* 简述Dalvik虚拟机 ?
  + android基于寄存器定制的虚拟机 , 而JVM是基于栈. 其相当于JVM的优势是, 编译的时候, 花费的时间更短
  + Dalvik执行的时.dex格式的字节码 , 而JVM执行的时.class格式的字节码
  + Dalvik的特点:
    - android平台的核心
    - 支持.dex格式的程序运行
    - .dex专为Dalvik设计的一种压缩格式
    - 减少整体文件尺寸
    - 提高I/O操作的速度
    - 适合内存和处理器速度有限的系统

计算机网络 :

* okhttp的原理
  + 为什么要使用okhttp :
    - Okhttp用来替代HttpURLConnection 和 Apache HTTP Client
    - 支持SPDY,HTTP/2协议(基于TCP的应用层协议,并不是一种用于替代HTTP的协议，而是对HTTP协议的增强) , 共享同一个Socket来处理同一个服务器的所有请求
    - 如果SPDY不可用，则通过连接池来减少请求延时
    - 无缝的支持GZIP来减少数据流量
    - 缓存响应数据来减少重复的网络请求
  + OkHttp支持2.3和以上版本，对于java，需要jdk1.7 ，OkHttp需要依赖Okio包
* HTTP1, HTTP1.1, HTTP2.0, SPDY 协议的区别 :
  + HTTP1 :
    - 一个TCP连接只能处理一个请求 (可以用域名分片提高其并发性)
    - HTTP请求和响应头未压缩
    - HTTP冗余的头部,会重复进行发送
    - 只有客户端才能发送请求, 服务端不能主动推送
  + HTTP1.1 : (如今使用得最多的HTTP协议)
    - 持久化连接以支持连接重用
    - 分块传输编码以支持流式响应
    - 请求管道以支持并行请求处理 (主要是支持并行)
    - 字节服务以支持基于范围的资源请求
    - 改进的更好的缓存机制
  + HTTP/1.x的缺陷 :
    - **连接无法复用**：连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。
      * HTTP/1.0传输数据时，每次都需要重新建立连接，增加延迟。
      * 1.1虽然加入keep-alive可以复用一部分连接，但域名分片等情况下仍然需要建立多个connection，耗费资源，给服务器带来性能压力
    - **Head-Of-Line Blocking**：第一个包被阻塞,带宽无法被充分利用，以及后续健康请求被阻塞
    - **协议开销大 :**  header内容过大, 增加了传输的成本
    - **安全因素**：HTTP/1.x中传输的内容都是明文，客户端和服务端双方无法验证身份
  + SPDY : (2016年google已停止对其开发)
    - 多路复用 :
      * 解决1.x holb的问题
    - 请求优先级 :
      * 允许给每个request设置优先级，这样重要的请求就会优先得到响应
    - header压缩
  + HTTP/2 :
    - 新的二进制格式 :
      * 改进传输性能，实现低延迟和高吞吐量
      * 其中HTTP1.x的首部信息会被封装到Headers帧，而我们的request body则封装到Data帧里面
* header压缩
* 流（stream）和多路复用（MultiPlexing） :
  + 流 :
    - 每个http request都会新建自己的stream，response在同一个stream上返回
  + 多路复用 :
    - 连接共享
* **流量控制（Flow Control）**
* **流优先级（Stream Priority）**
* **Server Push**
* SPDY协议基本特征 :
  + 流复用 :
    - SPDY最牛逼的地方，是允许在一个TCP连接里面，允许无限并发流
  + 请求优先级 :
    - 因为支持并行，所以必须要有优先级机制
  + HTTP header压缩 , 这样包更小
  + 服务器启动流。
    - 服务器启动流能用来分发内容到客户端，而不需要客户端请求它
  + Server hint :
    - 类似于我们的预加载技术 , 但仍需客户端请求
  + 安全防攻击 :
    - 采用了SSL+数据压缩，安全性上有了很大提升
* OKhttp源码原理 :
  + 请求流程 : 通过拦截器发起请求,获取response
* 类模块 :
  + protocol 协议枚举
  + Dispatcher 线程池同步\异步请求的实现
    - 无边界限制的线程池
    - SynchronousQueue阻塞队列 :
    - 同步execute, 异步enqueue
  + ConnectionPool 连接池
    - 线程池是0,1,LinkedBlockingQueue
  + Interceptor 拦截器
    - 针对Request和Response的切面处理
    - 把实际的网络请求、缓存、透明压缩等功能都统一了起来
  + InternalCache , Cache :
    - 主要在Engine.sendRequest()和Engine.readResponse()有用到
  + 重连机制在HttpEngine中实现
* 如何使用Gzip ?
  + 通过拦截器对Request 的body进行gzip的压缩, 减少流量传输
* 平台适应性 ?
* Platform :
  + Android和JdkWithJettyBootPlatform , 通过java反射去实现的
* Http和HTTPS的区别, 算法实现 ?
  + https是在http的基础上, 增加了ssl(安全套接字层)协议 和 TLS(运输层安全)协议 , 对传输数据进行加密和解密. 保证了安全性
    - TSL协议是基于SSL协议3.0基础上设计的
    - SSL协议提供的服务 :
      * 1. 认证用户和服务器
      * 2. 加密传输数据
      * 3. 保证数据完整性
  + Https是使用端口443，而Http使用80
  + https的缺点 :
    - 1. 握手阶段比较费时, 比http长
      * TPS 正慢慢取代 HTTP成为业界主流上层协议, 但HTTPS 在未经优化的情况下, 性能不及 HTTP 的百分之十
    - 2. https连接缓存不如http高效
    - 3.ssl证书需要钱,小型个人网站难以支撑
    - 4.ssl证书需要绑定ip, 不能再同一ip上绑定多个域名
    - 5.https,对于dos攻击,服务器劫持起不到作用. 中间人攻击的话在ca根证书可以控制的情况下会出现
  + http的缺点 :
    - **连接无法复用**：连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。
      * HTTP/1.0传输数据时，每次都需要重新建立连接，增加延迟。
      * 1.1虽然加入keep-alive可以复用一部分连接，但域名分片等情况下仍然需要建立多个connection，耗费资源，给服务器带来性能压力
    - **Head-Of-Line Blocking**：第一个包被阻塞,带宽无法被充分利用，以及后续健康请求被阻塞
    - **协议开销大 :**  header内容过大, 增加了传输的成本
    - **安全因素**：HTTP/1.x中传输的内容都是明文，客户端和服务端双方无法验证身份
  + http协议头部有哪些字段 ?
    - todo 这里可能会不准确
      * 请求头 : 请求行, 消息报头, 请求正文
      * 消息头 : 普通报头, 请求报头 , 响应报头 , 实体报头;
  + HTTP的Get和Post方法的区别 :
    - get - 明文发送内容
    - post - 请求不能被缓存 ; post请求没有长度限制 ; post请求不会保存在浏览器记录中 ; post请求的URL无法保存为浏览器书签
  + 对称加密 和 非对称加密 :
    - 对称加密 : 加密和解密的方式 都是 同样的密钥. DES , AES
    - 非对称加密 : 主要是RSA算法 , 一对密钥 , 公钥 和 私钥 ; 公钥一方对消息教秘 , 只有私钥一方才能对你的消息解密 . RSA , DSA
    - Hash算法 : 单向算法 . MD5 , SHA1
  + TCP/IP 为何要三次握手 ?  释放是4次f
    - 保证了校验数据 , 保证了可靠性
  + TCP 和 UDP ?
    - UDP不可靠, 因为UDP发出消息后, 并不会验证消息是否送达对方
    - 但就传输速度来说的话, UDP协议的速度更快更高
    - 心跳连接, 推送的时TCP包
* OKhttp面试相关 :
  + okhttp一般的作用 :
    - OKhttp一般用于处理Get,Post请求, 基于HTTP文件上传\下载, 加载图片
  + OKhttp请求 :
    - 同步是execute
    - 异步是enqueue , 请求流程 , 没看过源码 实话实说, 但猜想也是用了线程池, 并且异步回调也会自己去封装一个FetureTask,获取异步响应结果
  + OkhttpInterceptor拦截器 :
    - 可以添加请求头
    - 可以响应response,进行重试,
  + Okhttp 优点 :
    - 支持SPDY协议,(最小化网络延迟,基于TCP的应用层协议 , 提升网络速度, 优化用户的网络使用体验)
    - 使用连接池减少请求延迟
    - 利用响应缓存来避免重复的网络请求
  + httpResponseCode :
    - 401 帐号过期 , 403服务器拒绝, 404 notfount, 200 成功 , 500, 502服务器错误
* Daager类似这种依赖注入框架 :
  + **都没有使用发射来实现依赖注入 , 使用的APT**
    - **注 : 在Java平台上很多时候都是通过反射或者动态编译来提供依赖注入**
  + APT , annotationProcessor, JavaPoet关系整理 :
    - annotationProcessor是APT的替代 ,
    - apt的处理要素 : 注解处理器（AbstractProcess）+代码处理（javaPoet）+处理器注册（AutoService）+apt
    - javapoet是用于代码生成的
* Http和HTTPS的区别, 算法实现 ?
  + https是在http的基础上, 增加了ssl(安全套接字层)协议 和 TLS(运输层安全)协议 , 对传输数据进行加密和解密. 保证了安全性
    - TSL协议是基于SSL协议3.0基础上设计的
    - SSL协议提供的服务 :
      * 1. 认证用户和服务器
      * 2. 加密传输数据
      * 3. 保证数据完整性
  + 客户端在使用HTTPS方式与WEB服务器通信时,有以下几个步骤 :
    - 通过url, 要求与服务端建立ssl连接
    - 服务端 → 返回证书信息(包含公钥)
    - 客户端产生随机(对称)密钥
    - 客户端使用公钥对对称密钥加密
    - 客户端 → 发送加密后的对称密钥到服务端
    - 客户端 与 客户端 通过对称密钥加密的密文通信
* Https是使用端口443，而Http使用80



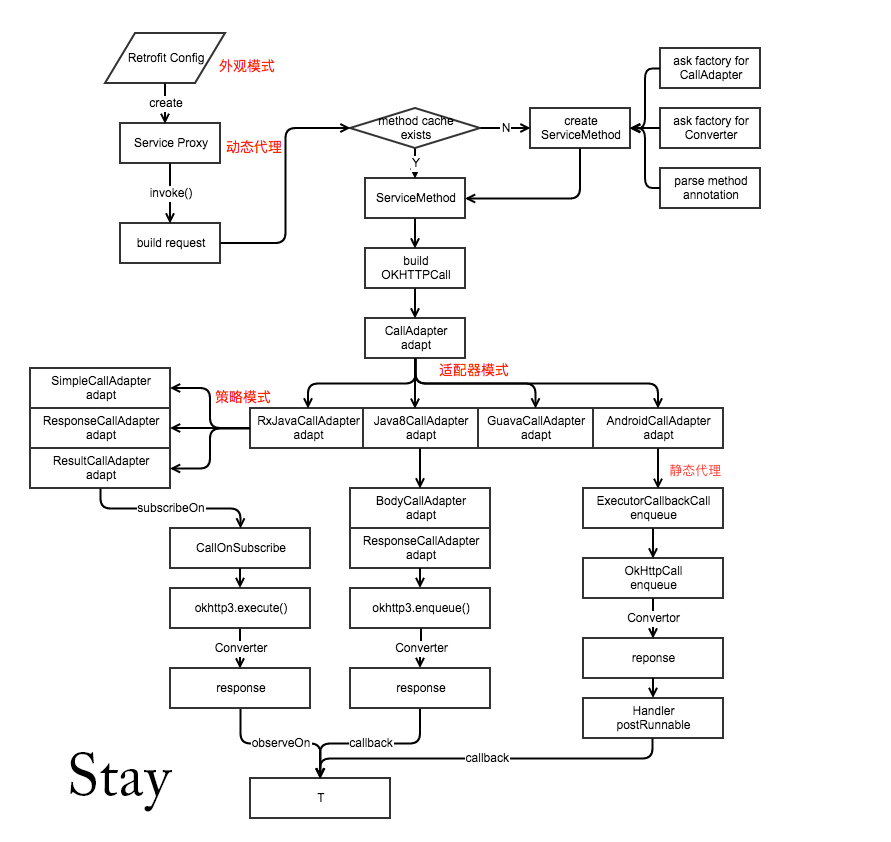
* https的缺点 :
  + 1. 握手阶段比较费时, 比http长
    - TPS 正慢慢取代 HTTP成为业界主流上层协议, 但HTTPS 在未经优化的情况下, 性能不及 HTTP 的百分之十
  + 2. https连接缓存不如http高效
  + 3.ssl证书需要钱,小型个人网站难以支撑
  + 4.ssl证书需要绑定ip, 不能再同一ip上绑定多个域名
  + 5.https,对于dos攻击,服务器劫持起不到作用. 中间人攻击的话在ca根证书可以控制的情况下会出现
* 算法实现 : 客户端 - 服务端 的对称加密 , 如上图
* 对称加密 和 非对称加密 :
  + 对称加密 : 加密和解密的方式 都是 同样的密钥
  + 非对称加密 : 主要是RSA算法 , 一对密钥 , 公钥 和 私钥 ; 公钥一方对消息教秘 , 只有私钥一方才能对你的消息解密
* TCP/IP 为何要三次握手 ?
  + 保证了校验数据 , 保证了可靠性
* TCP 和 UDP ?
  + UDP不可靠, 因为UDP发出消息后, 并不会验证消息是否送达对方
  + 但就传输速度来说的话, UDP协议的速度更快更高
  + 心跳连接, 推送的时TCP包
* http协议头部有哪些字段 ?
  + allow , accept-encoding, content-encoding,content-length, content-type, date, expires, last-modified, location, refresh, set-cookie
  + 总结 : 编码格式, content-type, 以及时间, 是否设置cookie等
* HTTP的Get和Post方法的区别 :
  + get - 明文发送内容
  + post - 请求不能被缓存 ; post请求没有长度限制 ; post请求不会保存在浏览器记录中 ; post请求的URL无法保存为浏览器书签
  + put
  + delete
  + head
  + connect
  + options
  + trace
* 常用请求状态码 :
  + 401 身份认证
  + 403 forbidden
  + 404 找不到网址
  + 500 , 502

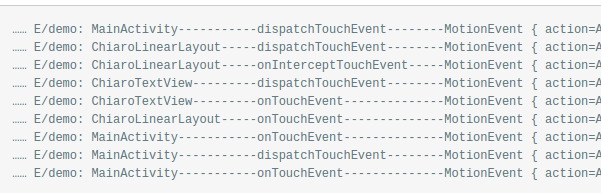
插件化 :

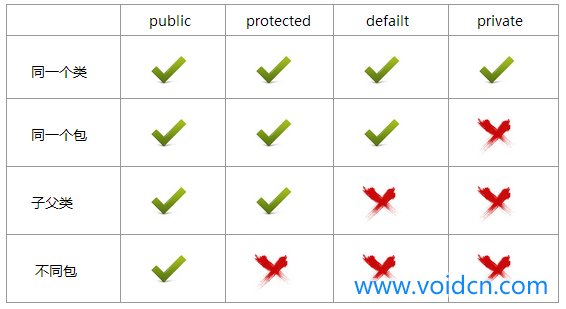
* DroidPlugin原理解析 :
  + 功能 :
    - 可以实现直接加载为安装的apk文件.
    - 应用双开的方式也可以通过DroidPlugin来实现
  + 缺点 :
    - 兼容性太差 , 各厂商的定制ROM对AMS,PMS都做了个性化修改,
    - 对AMS,PMS hook太多,
  + 原理 :
    - 主要是通过DexClassLoader来动态加载插件的ClassLoader在通过反射来获取插件类.(跟JAVA一样一样有双亲委派机制,抽象父类是BaseDexClassLoader)
      * DexClassLoader可以加载jar/apk/dex，可以从SD卡中加载未安装的apk
      * PathClassLoader只能加载系统中已经安装过的apk
    - 资源共享冲突解决 ?
      * 避免了AAPT修改资源ID,使用不同的AssetManager加载各自资源, 不能共享
* Replugin :
  + 通过Gradle脚本编译, 生成占坑文件
  + 只hook , ClassLoader , 没有像DroidPlugin hook太多
  + 兼容性好
* Atlas :
  + 更像组件化, 但doc太抽象了, 没怎么看

性能优化 :

* 图片的管理 :
  + 按需加载图片 , 图片分辨率不能超过控件 , 需做压缩(本地, 客户端压缩 , 请求, 服务端压缩)
  + 565 优先, 避免8888, 内存浪费
  + 图片处理的时候, 及时释放 , 避免在内存中常驻
  + 本地图片压缩处理,减少包体大小
* 内存泄漏场景 :
  + handler, 静态类
  + handler.post , 尤其是delay函数, 一定注意关闭
  + 匿名类持有耗时activity生命周期函数, activity生命周期结束前未释放
  + 一些例如cusor的释放 , bitmap的释放避免在内存中常驻
* review :
  + 1.低级bug的检查,null指针,数组越界等
  + 2.是否存在内存泄漏的代码
  + 3.使用数据结构在当前业务是否最优
  + 4.新增加的res资源是否满足
  + 5.新增compile是否有固定版本, 不能+号
  + 6. 如涉及到自定义view或者新增模块代码, 是否在生命周期结束能够稳定的资源释放
* APK包大小优化 :
  + 1.progard
  + 2.gradle配置shrinkResoures
  + 3.图片压缩webp,.9替换png
  + 4.帧动画或者selector涉及使用本地res的,尽量用代码去实现
  + 5.rom针对屏幕密度去打包资源文件720p,1080p
  + 6.减少jinlibs的大小, 按照CPU的abi去打对应的包
  + 7.对本地res图片进行压缩
    - 基于plugin插件编译时期的压缩 : <https://github.com/chenenyu/img-optimizer-gradle-plugin>
    - tiny运行时对download图片压缩 : <https://github.com/Sunzxyong/Tiny>
* 冷启动优化总结 :
  + Application Oncreate() 初始化异步执行
  + 首页View绘制, 层级优化, 避免过度绘制
  + 异步线程若需与主线程同步结果, 需提前执行异步线程, 减少主线程的等待时间
  + 代码优化 , 优化点 : 数据结构优化, 算法优化 , 是否过多创建了对象, 具体看CPU和内存情况, 避免dalvik并发去清理内存
  + 避免在冷启动期间, 执行动画, 和自定义view的一些measure工作
* Parcelable和Serializable :
  + Parcelable : 内存间传输, 主要是用于AIDL, 比Serializable高效,只能程序内
  + Serializable : 保存或网络传输数据, 主要是数据读写,程序内,或者程序间
  + java还提供了一个序列化方法 :
    - Externalizable的实现 :
      * 必须实现writeExternal()和readExternal()接口
      * 必须定义不带参数的构造函数
* Binder,java层的实现 : <http://weishu.me/2016/01/12/binder-index-for-newer/>
  + binder会对通信双方做身份校验, 安全
  + IBinder/IInterface/Binder/BinderProxy/Stub 关系 :
    - IInterface : 定义远程server应该具有的能力
    - 创建一个aidl接口, 系统会自动生成IInterface(表示远程binder应该具有的能力) , 内部有一个stub 和 proxy类
    - asInterface返回一个IBinder类型, 如果是本地对象(表示同一进程), 他就是binder类型, 如果是代理对象(表示跨进程), 就是proxy类型,
    - Stub本身是一个IBinder（Binder），它本身就是一个能跨越进程边界传输的对象
    - proxy就是一个代理类
* Android的Binder总结 :
  + binder机制是进程间通信(IPC)的方案
  + binder通信是采用C/S架构 , 包含Client, Server, ServiceManager, binder驱动, 主要架构分三层 :
    - java层 : client 对 server端的关系是 使用服务
    - c++层 : 通过addService() 和 getService()从ServiceManager(C++层)获取服务
    - kernel层 : client 和 server进程往往采用ioctl方法跟内核控件的驱动进行交互 ,真正在Client和Server两端建立通信的基础设施便是Binder Driver
      * 问ioctl是什么 ?
        + binder\_ioctl数据操作, kernel层的
    - 
  + binder通信的时候,各层干的事情总结 :
    - kernel  :
      * 创建了全局的哈希链表binder\_procs，用于保存所有的binder\_proc队列
    - native层 :
      * BpBinder(客户端transcat()发送消息) 和 BBinder(onTranscat接收相应的事物) , 都实现了Ibinder接口
  + Binder详细架构图 :
    - 
    - 分析 :
      * ServiceManager :
        + framework层 : framework层的ServiceManager类的实现最终是通过BinderProxy传递给Native层来完成的,
        + C++层 : android binder大管家, 最核心的两个功能为查询和注册服务 , Android系统中的各个服务，都是添加到ServiceManager中进行管理的，而且每个服务都对应一个服务名。当Client获取某个服务时，则通过服务名来从ServiceManager中获取相应的服务
      * IPC流程 :
        + 1. 先将server端的service通过addService注册到ServiceManager里面;
        + 2. 客户端在通过从ServiceManager,getService获取到服务.并且实现AIDL接口MyServiceProxy就能进行通信了
* 进程间通信方式 :
  + 管道(存储-转发方式,2次内存拷贝) / 共享内存 / 消息队列(存储-转发方式,2次内存拷贝) / 信号量 / socket(传输效率很低) / binder
  + 为什么用binder ?
    - 1. Binder能够很好的实现Client-Server架构;
    - 2. Binder的传输效率和可操作性很好
    - 3. Binder机制的安全性很高
* 线程间通信 :
  + 共享内存(JMMjava内存模型, volatile关键字) 和 发送消息(handler)
  + **共享内存 : 线程之间通过读写内存中的公共状态来隐式通信**
  + **消息传递 : 在消息传递的并发模型里, 线程通过明确的发送消息来显示进行通信**
* 线程和进程的区别 :
  + 一个程序运行在一个进程, 一个进程包含多个线程
  + 进程拥有一个完整的虚拟地址空间 , 不依赖于线程而独立存在 , 反之,线程是进程的一部分, 没有自己的地址空间, 与进程内的其他线程一起共享分配给该进程的所有资源
* Retrofit原理？为何用代理？代理的作用是什么？
  + <http://blog.qiji.tech/archives/2267>
  + <https://blog.piasy.com/2016/06/25/Understand-Retrofit/>
  + Retrofit 优点 :
    - 解偶,
    - 与rxjava结合
    - 面向接口编程AOP , 主要是InvocationHandler动态代理
  + 代码实现, :
    - ServiceMethod : 对于一个Api的Interface
    - Callback : 请求数据返回的接口
    - Converter : 这个接口主要的作用就是将HTTP返回的数据解析成Java对象Callback请求数据返回的接口
    - Call : 这个接口主要的作用就是发送一个HTTP请求，Retrofit 默认的实现是 OkHttpCall
    - CallAdapter :　适配器模式，　转化成Rxjava
  + 设计模式 :
    - builder模式 :  retrofit.create()
    - 装饰模式 : ExcutorCallbackCall : 装饰OkHttpCall
    - 动态代理 : serviceMehtod
    - 适配器模式 : CallAdapter
    - 策略模式 : CallAdapter - RxJavaCallAdapter



* synchronized的ReentrantLock的相同和区别 ?
  + 区别 :
    - ReentrantLock 可以被interrupt的同步 , 而synchronized的同步是不能Interrupt的
      * 如果一个线程中断了interrupt会怎么办 ?
        + synchronized会导致线程无线等
        + ReentrantLock支持线程中断, 但需要调用在可能被线程中断的代码块调用lock.lockInterruptibly(); 而不能是 lock.lock()
    - 在资源竞争不激烈的情形下，ReentrantLock性能稍微比synchronized差点点 , 因此单例模式的选择上, 优先选择synchronized
    - 但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态
    - 释放锁的时候 : ReentrantLock需要手动释放 , synchronized是自动释放
    - ReentrantLock可以知道是否获取到该锁, Lock有提供这个接口.synchronized不支持
    - ReentrantLock更灵活, synchronized锁对象, 多个线程必须等待释放. ReentrantLock可以更Condition配合, 更加灵活锁的释放与获取
    - ReentrantLock又区分了公平锁(实现了谁先来谁获取锁)和非公平锁, synchronized只能是非公平锁
    - ReentrantLock又更进一步细分了,读写锁, 读锁是共享锁, 写锁是独占锁.
  + 相同点 :
    - 都可以锁类, 锁对象.
    - 都是可重入锁
* 多线程的方法回顾 ?
  + 线程的五种状态 : 新建状态(New), 就绪状态(Runnable), 运行状态(Running), 阻塞状态(Blocked), 死亡状态(Dead)
  + Object的**wait(), notify()和notifyAll()** , 这个跟锁相关, 因为是锁类或者锁对象, 调用对象的这三个方法, 也就是释放该对象的锁.
  + **yield()**线程让步 : 即线程从running(运行状态)进入runnable(就绪状态) , 但如果有加锁,  yield()是不会释放锁. ps : 就绪状态的话, 就是跟其他线程继续竞争CPU资源
  + **sleep()**线程休眠 : 即当前线程会从“运行状态”进入到“休眠(阻塞)状态 , 时间到了, 它会由“阻塞状态”变成“就绪状态” , sleep不会释放锁, 然而wait()是可以的
  + **jion()** 让“主线程”等待“子线程”结束之后才能继续运行:
  + **interrupt()** 线程中断 :  通过isInterrupted和catch InterruptedException 来中止线程, 线程是自己中止自己, 不能由其他线程来中止, 阻塞状态和运行状态的线程都能够通过标记符来中止 .
  + 线程优先级, 可以设置, 高优先级先执行, 执行完毕才是低优先级线程执行.
* 一个线程里面只有一个Looper , 如果保证一个线程只有一个Looper的 ?
  + 因为Looper的初始化只有调用Looper.prepare() , Looper是通过threadloacl.get来保存来判断是否有初始化.否则就threadLocal , set
  + 关于ThreadLocal :
    - **ThreadLoacl :**<http://droidyue.com/blog/2016/03/13/learning-threadlocal-in-java/index.html>
      * ThreadLocal 是一个关于创建线程局部变量的类
        + 当前线程中, 任何一个点都可以访问到ThreadLocal的值
        + ThreadLocal只能被该线程访问, 一般情况下其他线程访问不到
      * 一个线程只有一个ThreadLocal , 每个线程只存在一个Looper
      * ThreadLocal依附于创建类的持有, 建立在堆内存中
      * 多个线程如果要访问ThreadLocal的值, 可以使用InheritableThreadLocal
      * ThreadLocal不会产生内存泄漏, 因为ThreadLocalMap在选择ThreadLocal的引用的时候 , 是对其实例的弱引用
* 多线程的实现方式 ?
  + 三种，　runnable 和 thread , 继承Callable, 通过FutureTask包装器来创建线程
  + java 又有 ExecutorService 线程池来管理执行线程
  + android多线程有AsyncTask, IntentHandler, IntentService, Handler来实现
* android ANR异常发生的场景:
  + anr产生的类型 :
    - Service Timeout:比如前台服务在20s内未执行完成；后台服务则是200s
    - BroadcastQueue Timeout：比如前台广播在10s内未执行完成
    - ContentProvider Timeout：内容提供者,在publish过超时10s;
    - InputDispatching Timeout: 输入事件分发超时5s，最主要是这个.
  + anr的保存路径 :
    - data/anr/traces.txt
  + anr的文件分析 :
    - 先看trace, 排查内存是否足够,Total和Max; 是否存在线程死锁, 阻塞了主线程
    - 第二看system\_server这个进程, 是否是因为其他进程原因影响了app进程的anr
    - 看log上下文, 主要看anr发生前 :
      * 1. 看ActivityManager打开了哪些页面?(用于排查内存泄漏)
      * 2. 看之前是否有异常发生导致的anr, 例如DeadObjectException binder挂掉了,会导致anr; 其他一些Exception , 例如之前遇到的art一直包一些第三方库gson,OKhttp的log, 最终原因是因为内存泄漏 + 内存管理,大量的inflate view
* 自定义的View和ViewGroup :
  + View的绘制原理 :
    - 由根布局开始,从ViewGroup的measure方法开始, 递归调用View的measure()函数,并且measureSpec参数是父类向下传递的
    - 整个View树的layout也是从根布局开始递归的调用, 这个函数一般是在父ViewGroup里面进行重写的,View没有必要进行重写,onLayout就是一个空方法, 摆放子View
    - 整个View树的layout也是从根布局开始递归的调用,一般重写onDraw()函数 :
  + requestLayout() 函数  : onmeasure() onlayout()
  + invalidate() 函数 : ondraw()
* View 和 ViewGroup的 事件拦截处理机制 :
  + ViewGroup :
    - dispatchTouchEvent(分发TouchEvent) , onInterceptTouchEvent(拦截TouchEvent) , onTouchEvent(处理TouchEvent)
  + View :
    - dispatchTouchEvent(分发TouchEvent)，　onTouchEvent(处理TouchEvent)
  + 具体分发到消费机制 : activity -> viewgroup -> view
* android动画相关 :
  + view动画 : 直接作用于view上面. setacnimation,android有提供默认的位移旋转动画的实现
  + 帧动画 : 占用包大小, 容易OOM, 可以使用属性动画来替代, 或者gif图 , 或者使用VectorDrawable来替换
  + 属性动画 : 可以通过改变View的内部属性来实现动画效果, 并且属性必须有set,get方法, 和invalidate()
* java泛型
  + 泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，以提高代码的重用率
  + Java 的泛型是在 **编译器** 层次实现的 . 仅仅是在编译期进行类型验证，在运行期会进行“类型擦除”，所以在运行期这些泛型参数会全部被擦除掉
  + 运行期能确定类型, 例如List<String> , getGenericSuperclass(), getActualTypeArguments() , 参考Gson, <https://www.zhihu.com/question/36645143>
* java接口和抽象类的区别 :
  + 接口可以被多个类实现, 而抽象类只能单继承
  + 接口和抽象类的成员变量必须是final和static :
    - 1. 接口和抽象类不能实例化, 因此成员变量应该属于class
    - 2. 接口使定义行为的 , 如果任由实现类修改成员变量, 会失去"通用性"
* 软引用 和 弱引用的 区别 ?
  + 软引用 : 在内存不足的时候, 被回收, 其周期也被终止
  + 弱引用 : 在虚拟机执行gc()的时候, 被回收, 其生命周期也被终止
* java权限修饰符访问相关 ?



* 静态代码块 和 构造方法哪个先执行 ?
  + 静态代码块先执行
* java线程池原理 ? <http://www.jianshu.com/p/87bff5cc8d8c>
  + 线程池 和 直接new Thread的好处
    - 降低资源消耗
    - 提高响应速度
    - 提高线程的可管理性
  + ThreadPoolExecutor的参数 :
    - corePoolSize 一般是CPU的核数 ; 线程池中的核心线程数量; 如果当前线程池的线程数 = corePoolSize ；那么新提交的任务将被放在阻塞队列里面 ;
    - maximumPoolSize : 线程池中允许的最大线程数量 , 如果阻塞队列满了. 那么会创建新的线程执行任务 , 提前是当前线程数小于maximumPoolSize
    - keepAliveTime : 线程空闲时的存活时间, 该参数只有在线程大于corePoolSize才有用
    - unit : keepAliveTime的单位
    - workQueue : 阻塞队列 , FIFO顺序执行; 吞吐量从底到高 ArrayBlockingQueue -> LinkedBlockingQuene -> SynchronousQuene(插入必须等待移除, 否则一直阻塞) -> priorityBlockingQuene(具有优先级的无界阻塞) ;
    - threadFactory : 创建线程工厂 , 一般是默认的
    - handler : 线程池饱和策略, 通常是阻塞队列满了, 并且没有空闲的工作线程. 如果继续提交任务, 就需要一种策略来进行处理
      * 1、AbortPolicy：直接抛出异常，默认策略；
      * 2、CallerRunsPolicy：用调用者所在的线程来执行任务；
      * 3、DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
      * 4、DiscardPolicy：直接丢弃任务；
* 线程池的类型 :
  + newFixedThreadPool : corePoolSize == maximumPoolSize ；当线程池没有可执行任务时，也不会释放线程
  + newCachedThreadPool :　newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程可能导致严重的性能问题
  + newSingleThreadExecutor : 线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行
    - new ScheduledThreadPoolExecutor(1, threadFactory)
  + newScheduledThreadPool : 在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据
    - super(corePoolSize, Integer.MAX\_VALUE,  DEFAULT\_KEEPALIVE\_MILLIS, MILLISECONDS,  new DelayedWorkQueue(), threadFactory);
* 线程池的队列的类型 :
  + ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
  + LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
  + PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
  + DelayQueue：一个使用优先级队列实现的无界阻塞队列。
  + SynchronousQueue：一个不存储元素的阻塞队列。
  + LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
  + LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。
* 线程池的几种状态 :
  + RUNNING : 接收新任务，并处理阻塞队列中的任务
  + SHUTDOWN : 不会接收新任务，但会处理阻塞队列中的任务
  + STOP : 不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务
  + TIDYING : 当所有的任务已终止 , 然后会调用terminated()
  + TERMINATED : 线程池彻底终止
* 