## 一,安卓性能优化

### 一》常见优化

1，内存优化。

2，UI优化（布局优化和绘制优化）

3，速度的优化（线程优化/网络优化）

4，电量优化。

5，启动优化。

### 二》listview优化

1，convertView复用

2，viewholder使用

3，图片优化

4，getView()中不要写过多的逻辑代码，不要创建很多的对象，逻辑代码放在别的地方，

5，item的布局减少层级

6，通过getItemViewType实现复杂布局的复用

7，简单布局可以将listview的scrollingCache和animateCache属性设置false。

如果设置为true会提高显示效果，但是需要消耗更多内存和更长的初始化时间

### 三》RecycleView

基本解决了ListView的一些缺陷，比如：需要手动重写ViewHolder，对于item的动画操作也需要很复杂的逻辑去实现等等

但是也有些缺点

不能简单的加头和尾   
2. 不能简单的设置子item的点击事件

所以对于一些简单的列表还是可以用listview的。Recylerview可以用于实现一些复杂的列表。

优化

1，简化item布局

2，图片优化

### 四》内存泄漏

内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。

内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory

Java内存泄漏的根本原因是什么呢？长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是Java中内存泄漏的发生场景。

常见的引起内存泄漏的原因：

1，资源未关闭造成的内存泄漏

对于使用了BraodcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源的使用，数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。动画也要在界面不再对用户可见时停止。

2，监听器没有删除监听

在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，

从而增加了内存泄漏的机会。

3，handler

在activity中new handler时，由于Handler不是静态内部类，所以会持有外部类（Activity）的一个引用。当Handler中有延迟的的任务或者等待执行的任务队列过长时，由于消息持有对Handler的引用，而Handler又持有对其外部类的潜在引用，这条引用关系会一直保持到消息得到处理为止，导致了Activity无法被垃圾回收器回收，从而导致了内存泄露。

注意：非静态类和匿名内部类都会持有外部类的隐式引用。如handler（非静态内部类）对activity（外部类）

解决：1，使用静态内部类+弱引用的方式，静态内部类不会持有外部类的的引用

2，HandlerActivity退出后，移除Handler的所有回调和消息

代码如下图



4、用静态修饰集合引起内存泄漏：

有时候我们需要把一些对象加入到集合容器（例如ArrayList）中，当不再需要当中某些对象时，如果不把该对象的引用从集合中清理掉，也会使得GC无法回收该对象。如果集合是static类型的话，那内存泄漏情况就会更为严重。 因此，当不再需要某对象时，需要主动将之从集合中移除

5，静态内部类引用非静态外部成员变量

尽量不要在静态方法或者静态内部类中使用非静态外部成员变量（包括context )，即使要使用，也要考虑适时把外部成员变量置空；也可以在内部类中使用弱引用来引用外部类的变量。

6，多线程的使用

当在开启一个子线程用于执行一个耗时操作后，此时如果改变配置（例如横竖屏切换）导致了Activity重新创建，一般来说旧Activity就将交给GC进行回收。但如果创建的线程被声明为非静态内部类或者匿名类，那么线程会保持有旧Activity的隐式引用。当线程的run()方法还没有执行结束时，线程是不会被销毁的，因此导致所引用的旧的Activity也不会被销毁，并且与该Activity相关的所有资源文件也不会被回收，因此造成严重的内存泄露。

解决：在Activity退出后主动停止Thread

7，常见的内存开支

内存的开支情况

\*使用枚举通常会比使用静态常量消耗两倍以上的内存，尽可能不使用枚举

\*任何一个Java类，包括匿名类、内部类，都要占用大概500字节的内存空间

\*任何一个类的实例要消耗12-16字节的内存开支，因此频繁创建实例也是会在一定程序上影响内存的

\*使用HashMap时，即使你只设置了一个基本数据类型的键，比如说int，但是也会按照对象的大小来分配内存，大概是32字节，而不是4字节，因此最好使用优化后的数据集合

8，OOM的原因和解决

出现OOM的原因

加载对象过大

相应资源过多，来不及释放

如何解决

在内存引用上做些处理，常用的有软引用、强化引用、弱引用

在内存中加载图片时直接在内存中作处理，如边界压缩

动态回收内存

优化Dalvik虚拟机的堆内存分配

自定义堆内存大小

## 二，源码及设计模式

### 一》LRUCache原理和部分源码解析

图片的三级缓存：也就是加载图片的时候首先从内存缓存中取，如果没有再从文件缓存中取， 如果文件缓存没有取到，就从网络下载图片并且加入内存和文件缓存

LRU：Least Recently Used，最近最少使用算法。当内存缓存达到设定的最大值 时将内存缓存中近期最少使用的对象移除，有效的避免了OOM的出 现。

LruCache中Lru算法：通过LinkedHashMap来实现的。LinkedHashMap继承于HashMap，它使用了一个双向链表来存储Map中的Entry顺序关系，这种顺序有两种，

一种是LRU顺序，

一种是插入顺序，

这可以由其构造函数public LinkedHashMap(int initialCapacity,float loadFactor, boolean accessOrder)指定。所以，对于get、put、remove等操作，LinkedHashMap除了要做HashMap做的事情，还做些调整Entry顺序链表的工作。

LruCache中将LinkedHashMap的顺序设置为LRU顺序来实现LRU缓存，每次调用get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。调用put插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除

部分源码解析：LruCache的初始化构造方法

public LruCache(int maxSize) {

if (maxSize <= 0) {

throw new IllegalArgumentException("maxSize <= 0");

}

this.maxSize = maxSize;

this.map = new LinkedHashMap<K, V>(0, 0.75f, true);//初始容量为零，0.75是加载因子，表示容量达到最大容量的75%的时候会把内存增加一半。最后这个参数至关重要。表示访问元素的排序方式，true表示按照访问顺序排序，false表示按照插入的顺序排序

}

### 二》view的事件分发机制

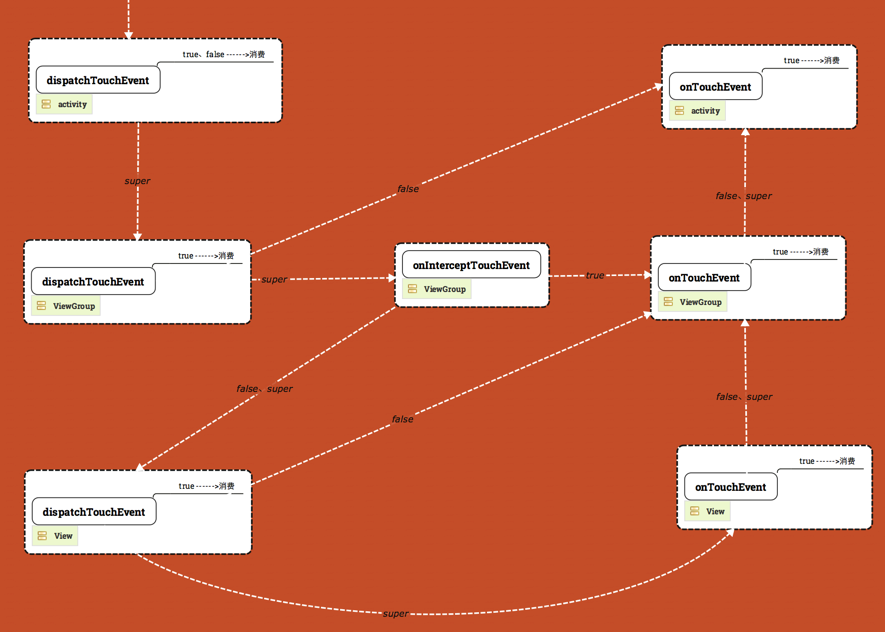
dispatchTouchEvent 分发事件

onInterceptTouchEvent 拦截事件只有viewgroup才有，view和activity没

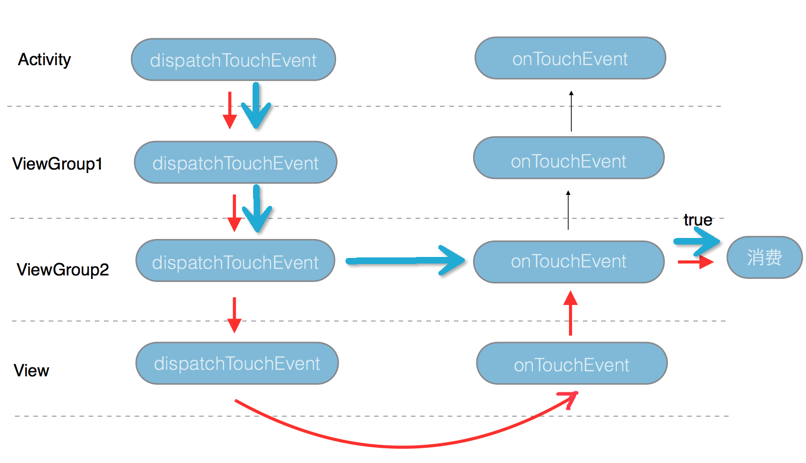
onTouchEvent 处理点击事件

1，图解ACTION\_DOWN 事件分发，如果面试的时候能把下面的图画出来。能增分不少

dispatchTouchEvent和 onTouchEvent的框里有个【true---->消费】的字，表示的意思是如果方法返回true，那么代表事件就此消费，不会继续往别的地方传了，事件终止

2，红色的箭头代表ACTION\_DOWN 事件的流向

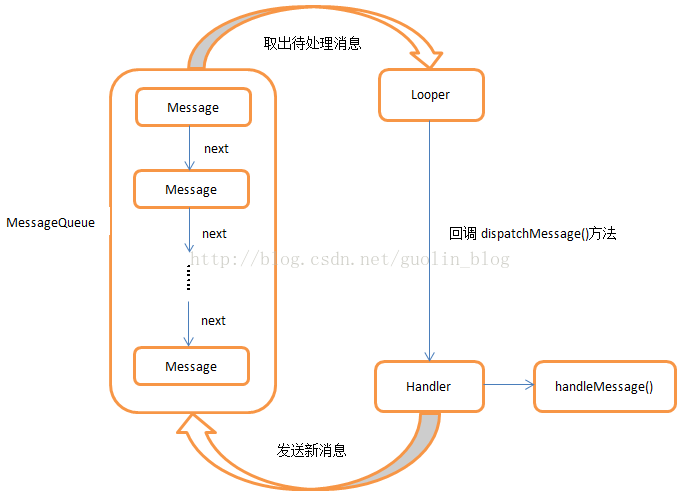
蓝色的箭头代表ACTION\_MOVE 和 ACTION\_UP 事件的流向



### 三》Handler原理机制

Handler主要负责发送和接受消息，Looper负责不断轮询MessageQueue，有新的消息就交给Handler处理，如果轮询不到新的消息，那就自身就处于阻塞状态。

#### 1，Handler简单图解



#### 2，Handler铁三角

Handler android的消息机制就是指Handler机制，Handler机制的运行需要MeeageQueue和Looper的辅助。

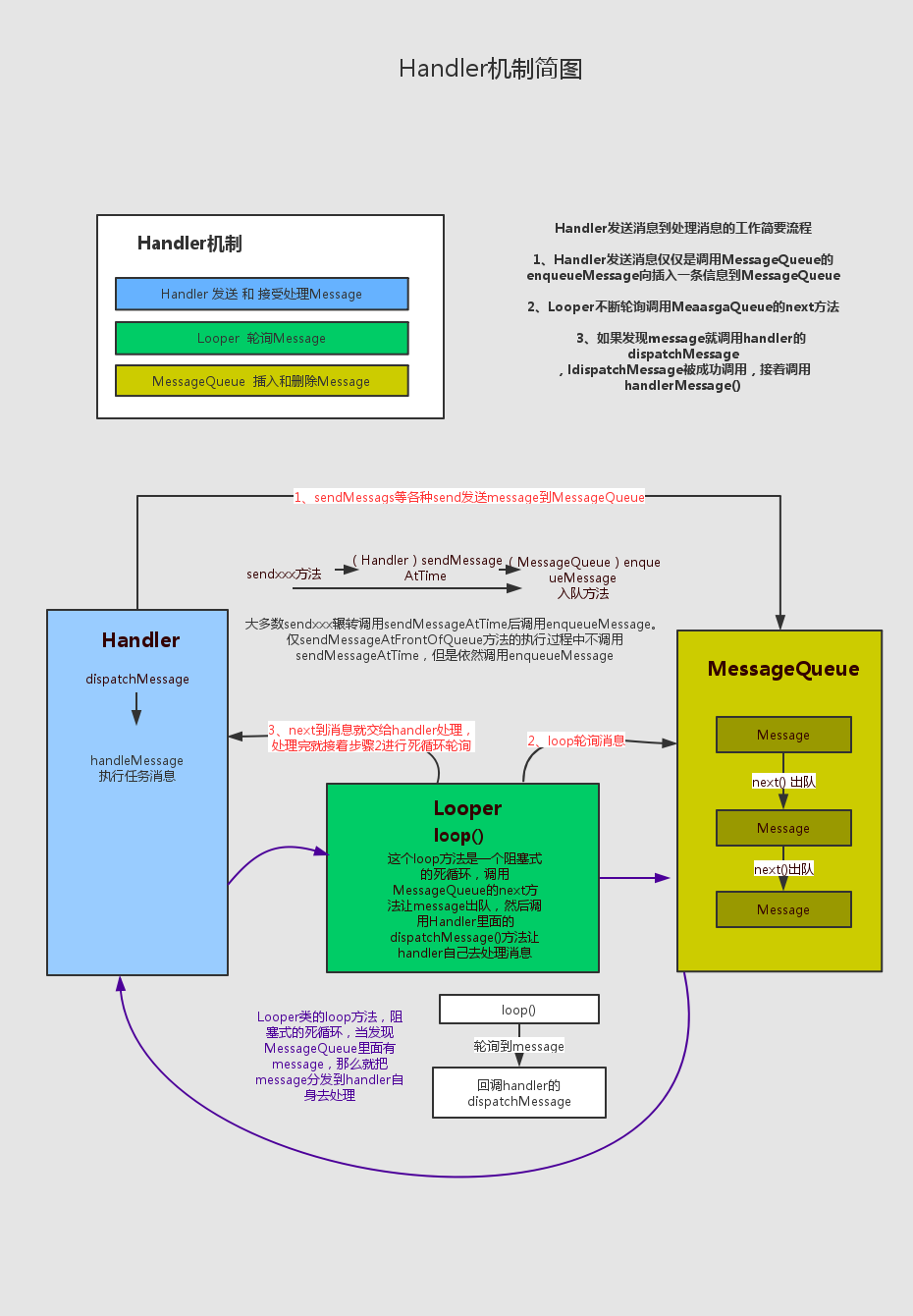
MessageQueue：消息队列，用于将所有收到的消息以队列的形式进行排列，

并提供入队和出队的方法。在Looper的构造函数中创建,因此一个Looper也就对应了一个MessageQueue.

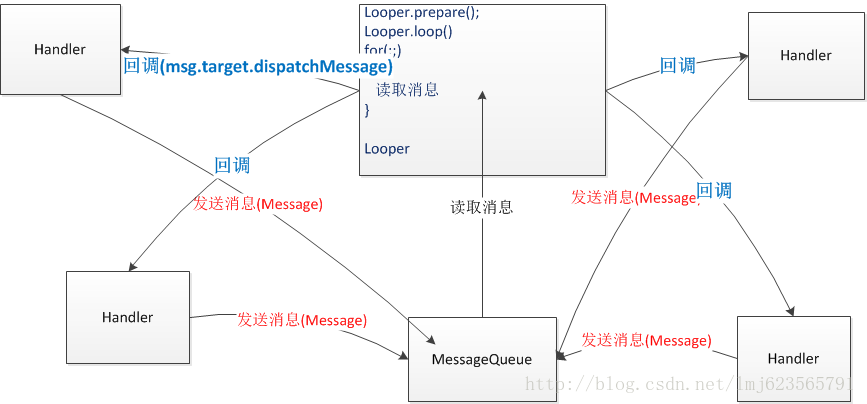
通过enqueueMessage实现消息的入队，通过next方法实现出队

Looper 轮询器 作用：与当前线程绑定，保证一个线程只会有一个Looper实例，同时一个Looper实例也只有一个MessageQueue。Looper 通过loop()方法调用messagequeue的next方法，不断从 MessageQueue中去取消息

#### 3，详解单个handler原理



#### 4，图解多个handler原理



#### 5，Handler的创建流程

1、首先Looper.prepare()在本线程中保存一个Looper实例，然后该实例中保存一个MessageQueue对象；因为Looper.prepare()在一个线程中只能调用一次，所以MessageQueue在一个线程中只会存在一个。

2、Looper.loop()会让当前线程进入一个无限循环，不端从MessageQueue的实例中读取消息，然后回调msg.target.dispatchMessage(msg)方法。

3、Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。

4、Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。

5、在构造Handler实例时，我们会重写handleMessage方法，也就是msg.target.dispatchMessage(msg)最终调用的方法。

好了，总结完成，大家可能还会问，那么在Activity中，我们并没有显示的调用Looper.prepare()和Looper.loop()方法，为啥Handler可以成功创建呢，这是因为在Activity的启动代码中，已经在当前UI线程调用了Looper.prepare()和Looper.loop()方法。

#### 补充，handler机制原理

1，用到的四个对象的关系

MessageQueue：装食物的容器

Message：被装的食物

Handler（msg.target实际上就是Handler）：食物的消费者

Looper：负责分发食物的人

2，handler原理机制

当我们调用handler.sendMessage(msg)方法发送一个Message时，实际上这个Message是发送到与当前线程绑定的一个MessageQueue中，

然后与当前线程绑定的Looper将会不断的从MessageQueue中取出新的Message，

调用msg.target.dispathMessage(msg)方法将消息分发到与Message绑定的handler.handleMessage()方法中。

一个Thread对应多个Handler 一个Thread对应一个Looper和MessageQueue，

Handler与Thread共享Looper和MessageQueue。

Message只是消息的载体，将会被发送到与线程绑定的唯一的MessageQueue中，

并且被与线程绑定的唯一的Looper分发，被与其自身绑定的Handler消费

### 四》安卓源码中用到的设计模式

#### 1，单例模式

保证在应用程序中，一个类Class只有一个实例存在

软键盘管理的 InputMethodManager和安卓系统级别的服务通常用的单例模 式，如WindowsManagerService、ActivityManagerService，更常用的是一个 LayoutInflater的类

#### 2，责任链模式---View事件分发机制

它使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的 耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象 处理它为止

具体体现在View的onTouchEvent方法中返回值的设置，

如果返回false，那么意味着当前的View不会是该次的责任人，将不会对其 持有；如果返回true，此时View会持有该事件并不再向外传递

#### 3，适配器模式----listview的BaseAdapter

将一个类的接口转换成客户希望的另外一个接口

以笔记本电源适配器为例，电源适配器将220V的电压转换到5v。

#### 4，观察者模式---将被观察者和观察者解耦

观察者模式是一种行为类模式，它定义对象间一种一对多的依赖关系，使得每当一个对 象改变状态，则所有依赖于它的对象都会得到通知并被自动更新

DataSetObserver就是一个观察者，它一旦发现BaseAdapter内部数据有变量，就会通过 回调方法DataSetObserver.onChanged和DataSetObserver.onInvalidated来通知 DataSetObserver的实现类。

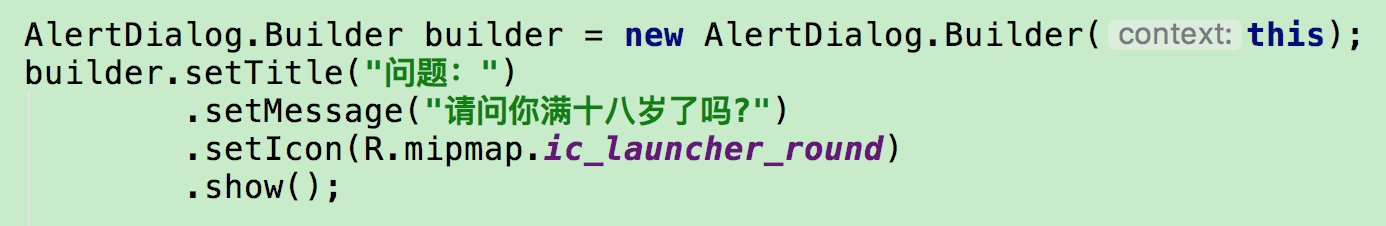
比如安卓开源项目EventBus和Rxjava的核心就是观察者模式

#### 5，工厂模式----BitmapFactory

静态工厂方法在Android中比较明显的例子应该就是BitmapFactory了，通过各种decodeXXX()就可以从不同渠道获得Bitmap对象

#### 6，建造者模式----AlertDialog.Builder

如果一个类的构造需要很多参数，而且这些参数并不都是必须的，那么这种情况下就比较适合Builder模式。比如构建一个AlertDialog，标题、内容、取消按钮、确定按钮、中立按钮，你可能只需要单独设置几个属性即可.



实际使用：如分享弹窗，有的需要朋友圈分享，有的不需要，就用这种建造者模式来链式调用。

#### 7，策略模式----动画的插值器

策略模式就相当于一个影碟机，你往里面插什么碟子，就能放出什么电影

以动画为例，设置不同的插值器对象，就可以得到不同的变化曲线

### 五》手写单例模式

一般情况下直接使用饿汉式就好了，如果明确要求要懒加载（lazy initialization）会倾向于使用静态内部类，如果涉及到反序列化创建对象时会试着使用枚举的方式来实现单例。

1,饿汉式

会在加载类后一开始就被初始化，即使客户端没有调用 getInstance()方法。饿汉式的创建方式在一些场景中将无法使用：譬如 Singleton 实例的创建是依赖参数或者配置文件的

Public class Singleton{

Private static final Singleton instance=new Singleton();

Private Singleton(){}

Public static Singleton getInstance(){

Return instance;

}

}

2,静态内部类

这种写法仍然使用JVM本身机制保证了线程安全问题；由于 SingletonHolder 是私有的，除了 getInstance() 之外没有办法访问它，因此它是懒汉式的；同时读取实例的时候不会进行同步，没有性能缺陷；也不依赖 JDK 版本

Public class Singleton{

Private static class SingletonHolder{

Private static final Singleton INSTANCE=new Singleton();

}

Private Singleton(){}

Public static final Singleton getInstance(){

Return SingletonHolder.INSTANCE;

}

}

3,枚举Enum

我们可以通过EasySingleton.INSTANCE来访问实例，这比调用getInstance()方法简单多了。创建枚举默认就是线程安全的，所以不需要担心double checked locking，而且还能防止反序列化导致重新创建新的对象

Public enum Singleton{

INSTANCE;

}

## 三，集合相关

### 一）LinkedHashMap详解

LinkedHashMap底层数据结构由链表和哈希表组成。

由链表保证元素有序。

由哈希表保证元素唯一

LinkedHashMap的两种排序

1，按插入顺序排序（默认）

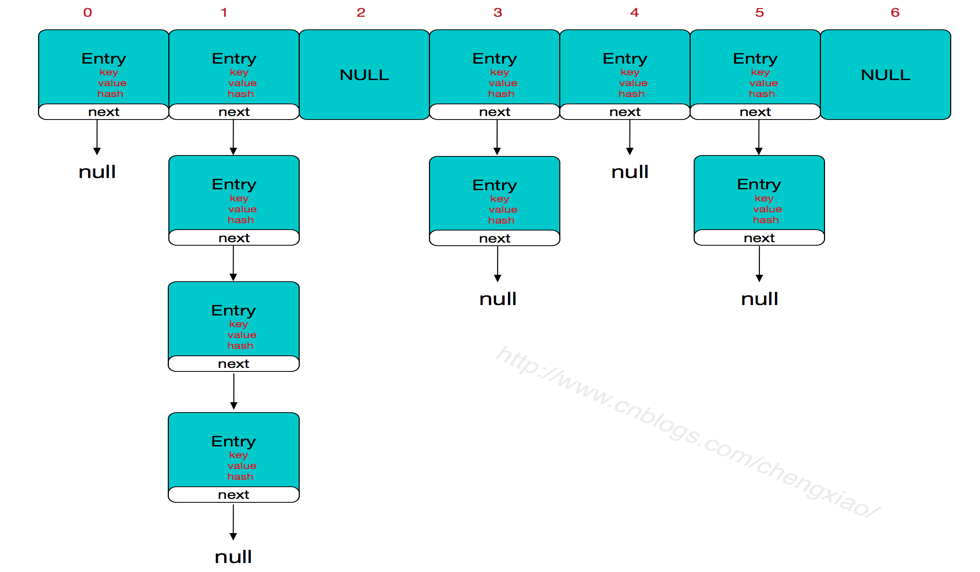
2，访问顺序排序（lru是这种排序）

LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失

### 二）HashMap和Hashtable和ConcurrentHashMap的原理

1，原理

HashMap的原理：hashmap本质数组加链表。通过hash（）方法获取key对应的hash值，然后计算出数组下标，如果多个key对应到同一个下标，就用链表串起来，新插入的在前面。默认长度为16

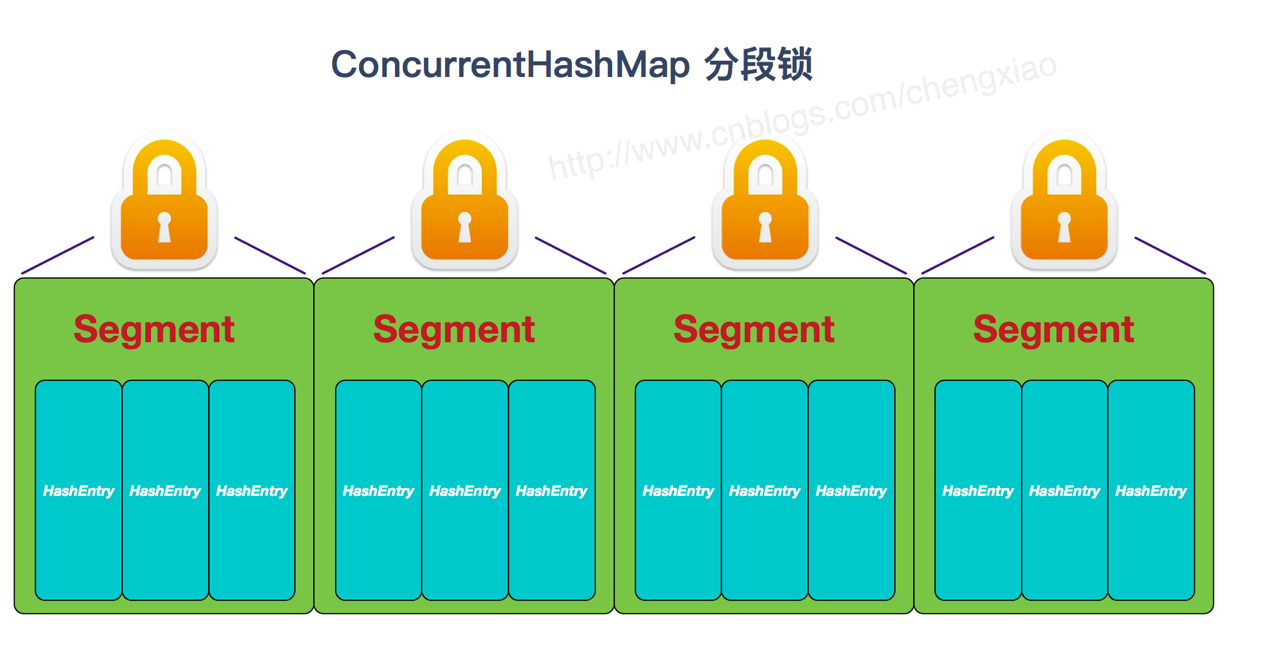
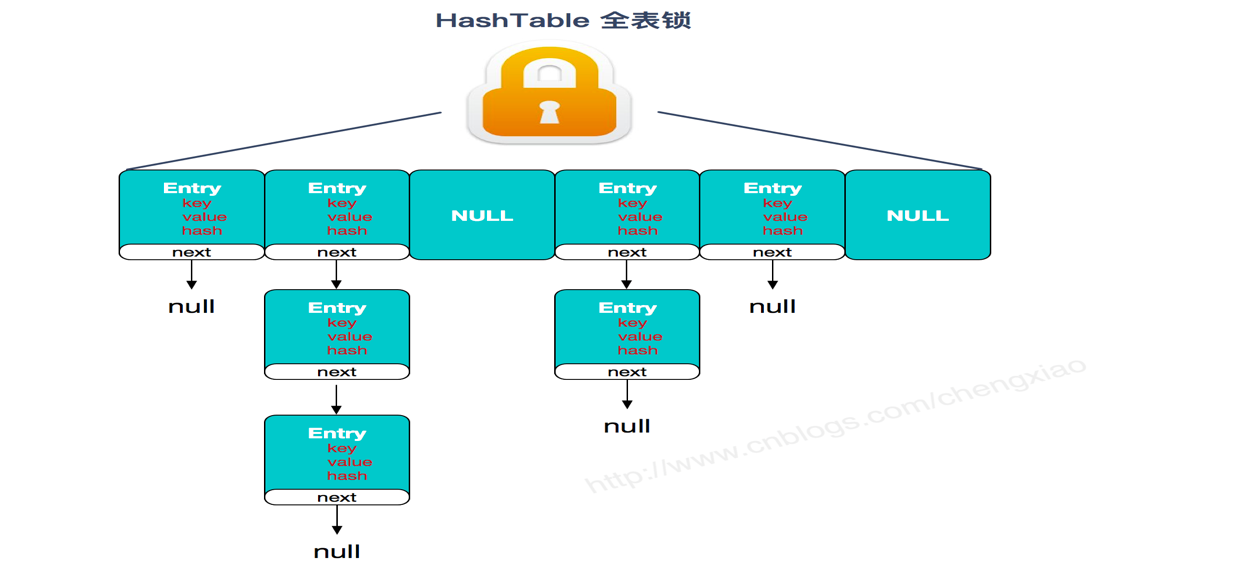


Hashtable的原理：HashTable和HashMap的实现原理几乎一样，差别无非是

1.HashTable不允许key和value为null；

2.HashTable是线程安全的。

但是HashTable线程安全的策略实现代价却太大了，简单粗暴，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。



ConcurrentHashMap原理：在hashMap的基础上，ConcurrentHashMap将数据分为多个segment（类似HashTable），然后每次操作对一个segment加锁，避免多线程锁得几率，提高并发效率。 ConcurrentHashMap的主干是个Segment数组，Segment里维护了一个HashEntry数组，并发环境下，对于不同Segment的数据进行操作是不用考虑锁竞争的。（就按默认的ConcurrentLeve为16来讲，理论上就允许16个线程并发执行

2，Hashmap的源码：

构造函数，空参或者单参时都会调用两个参数的

public HashMap(int initialCapacity, float loadFactor) {

    int capacity = 1;

    while (capacity < initialCapacity)

        capacity <<= 1;

    this.loadFactor = loadFactor;

    threshold = (int)(capacity \* loadFactor);

    table = new Entry[capacity];

    init();

}

有3个关键参数：  
 capacity：容量，就是数组大小  
 loadFactor：比例，用于扩容  
 threshold:=capacity\*loadFactor   最多容纳的Entry数，如果当前元素个数多于这个就要 扩容（capacity扩大为原来的2倍）

Get方法：根据key算hash值，再根据hash值取得数组下标，通过数组下标取出链表，遍历链表用equals取出对应key的value

public V get(Object key) {

    if (key == null)

        return getForNullKey();

    int hash = hash(key.hashCode());

    for (Entry<K,V> e = table[indexFor(hash, table.length)];

         e != null;

         e = e.next) {

        Object k;

        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

            return e.value;

    }

    return null;

}

3，HashMap和ConcurrentHashMap，hashtable的区别

HashMap： 线程不安全，效率高

ConcurrentHashMap： 线程安全，效率高，默认提升16倍

Hashtable： 线程安全，效率低

### 三）SparseArray原理

SparseArray是android里为<Interger,Object>这样的Hashmap而专门写的类,目的是提高内存效率，其核心是折半查找函数（binarySearch）。注意内存二字很重要，因为它仅仅提高内存效率，而不是提高执行效率

它要比 HashMap 节省内存，结构比HashMap简单（SparseArray内部主要使用两个一维数组来保存数据，一个用来存key，一个用来存value）不需要额外的数据结构（主要是针对HashMap中的HashMapEntry而言的）。

//折半查找又叫二分查找(针对有序数组) Arrays.binarySearch(arr,3);的实现原理就是下面

public static int binarySearch(int[] arr, int value) {

int min = 0;

int max = arr.length - 1;

int mid = (min + max) / 2;

while (arr[mid] != value) {

if (arr[mid] > value) {

max = mid - 1;

} else if (arr[mid] < value) {

min = mid + 1;

}

if (min > max) {

return -1;

}

mid = (min + max) / 2;

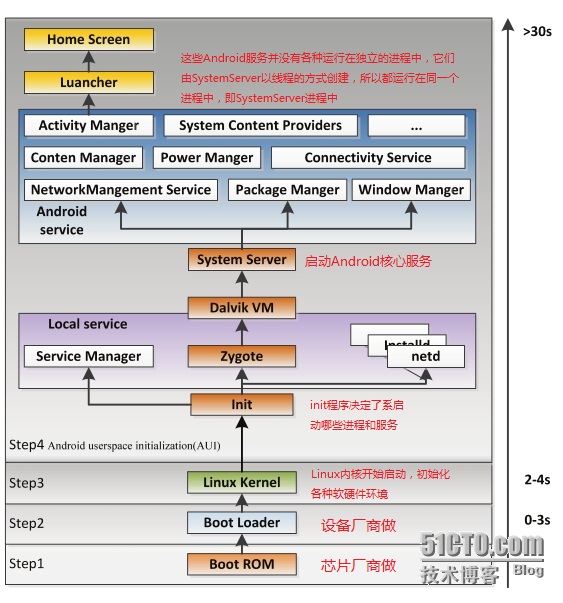
}

return mid;

}

## 四，系统相关

### 一》安卓系统启动流程



Zygote进程 –> SystemServer进程 –> 各种系统服务 –> 应用进程

当所有的服务都启动完毕后，SystemServer会打印出“Making services ready”，然后通过ActivityManager启动Home界面，并发送“ACTION\_BOOT\_COMPLETED”广播消息

### 二》App启动方式、Application生命周期

app的两种启动方式

#### 1，冷启动

系统会重新创建一个新的进程分配给它，所以会先创建和初始化Application类，再创建和初始化MainActivity类（包括一系列的测量、布局、绘制），最后显示在界面上

#### 2，热启动

进程在后台运行，所以热启动就不会走Application这步了，而是直接走MainActivity（包括一系列的测量、布局、绘制），所以热启动的过程只需要创建和初始化一个MainActivity就行了，而不必创建和初始化Application，因为一个应用从新进程的创建到进程的销毁，Application只会初始化一次。

App启动流程

用户点击app后会通知 ActivityManagerService 启动应用的入口 Activity， ActivityManagerService 发现这个应用还未启动，则会通知 Zygote 进程孵化出应用进程，然后在这个应用进程里执行 ActivityThread 的 main 方法。应用进程接下来通知 ActivityManagerService 应用进程已启动，ActivityManagerService 保存应用进程的一个代理对象，这样 ActivityManagerService 可以通过这个代理对象控制应用进程，然后 ActivityManagerService 通知应用进程创建入口 Activity 的实例，并执行它的生命周期函数。

上面的启动流程是 Android 提供的机制，我们只能在创建入口 Activity 的实例这里做文章，正常Main Activity 的启动流程：

-> Application 构造函数

-> Application.attachBaseContext()

-> Application.onCreate()

-> Activity 构造函数

-> Activity.setTheme()

-> Activity.onCreate()

-> Activity.onStart

-> Activity.onResume

-> Activity.onAttachedToWindow

-> Activity.onWindowFocusChanged

#### 3，统计app启动时长的两种方式

##### 3-1，本地通过adb命令行

adb shell am start -w packagename/activity

输入adb shell am start -W com.qcl/com.qcl.MainActivity得到下面结果

Activity: com.qcl/.MainActivity

ThisTime: 83

TotalTime: 83

WaitTime: 94

TotalTime是我们真正的启动时间

##### 3-2，通过收集log（可以用来获取线上启动时间）

开始时间：冷启动在Application的attachBaseContext

热启动在入口activity的onRestart中

结束时间：在入口activity的onWindowFocusChanged

#### 4，app的启动优化： 基于上面的启动流程我们尽量做到如下几点  1.Application的创建过程中尽量少的进行耗时操作  2.如果用到SharePreference,尽量在异步线程中操作  3.减少布局的层次,并且生命周期回调的方法中尽量减少耗时的操作

#### 5，app启动遇见黑屏或者白屏问题

1，产生原因   
其实显示黑屏或者白屏实属正常，这是因为还没加载到布局文件，就已经显示了window窗口背景，黑屏白屏就是window窗口背景

2，解决办法  
通过设置设置Style   
（1）设置背景图Theme   
通过设置一张背景图。 当程序启动时，首先显示这张背景图，避免出现黑屏

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">

    <item name="android:screenOrientation">portrait</item>

    <item name="android:windowBackground">>@mipmap/splash</item>

    <item name="android:windowIsTranslucent">true</item>

    <item name="android:windowNoTitle">true</item>

</style>

设置透明Theme   
通过把样式设置为透明，程序启动后不会黑屏而是整个透明了，等到界面初始化完才一次性显示出来

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">

    <item name="android:windowNoTitle">true</item>

    <item name="android:windowBackground">@android:color/transparent</item>

    <item name="android:windowIsTranslucent">true</item>

    <item name="android:screenOrientation">portrait</item>

  </style>

两者对比：  
 Theme1 程序启动快，界面先显示背景图，然后再刷新其他界面控件。给人刷新不同步感觉。  
 Theme2 给人程序启动慢感觉，界面一次性刷出来，刷新同步

Application和Process进程

当application在Linux平台开启时，系统会给这个application创建一个进程（process）来运行同时分配内存资源给该application，当程序结束运行时，该进程结束系统回收内存资源。

一个Application就是一个应用，在应用启动时，android会启动一个linux进程和一个主线程，这个进程以应用的包名命名，在默认的情况下，这个Application下的Activity、service、provider、receiver等组件都在这个进程中运行

#### 6，Application的生命周期

onCreate（） 在创建应用程序时创建

onTerminate（） 在模拟环境下执行。当终止应用程序对象时调用，不保证一定被调用，当程序是被内核终止以便为其他应用程序释放资源，那么将不会提醒，并且不调用应用程序的对象的onTerminate方法而直接终止进程。

onLowMemory（） 低内存时执行。好的应用程序一般会在这个方法里面释放一些不必要的资源来应付当后台程序已经终止，前台应用程序内存还不够时的情况。

onConfigurationChanged（Configuration newConfig） 配置改变时触发这个方法。

onTrimMemory（int level）程序在进行内存清理时执行​

### 三》Android 架构图



### 四》安卓的5种进程

1. 前台进程：拥有一个正在与用户交互的Activity（onResume被调用）的进程

2. 可见进程：拥有一个不在前台但是对用户依然可见的Activity（onPause被调用）的进程

3. 服务进程：拥有一个通过startService方法启动的服务的进程

4. 后台进程：拥有一个对当前用户不可见的Activity（onStop被调用）的进程

5. 空进程：没有任何活动的应用组件（进程中没有Service也没有Activity）的进程

**如何避免后台进程被杀死：**

1.调用startForegound，让你的Service所在的线程成为前台进程

2.Service的onStartCommond返回START\_STICKY或START\_REDELIVER\_INTENT

3.Service的onDestroy里面重新启动自己

### 五》activity的四种启动模式

standard : 标准模式,每次启动Activity都会创建一个新的Activity实例,并且将其 压入任务栈 栈顶,而不管这个Activity是否已经存在。Activity的启动三回调 (onCreate()->onStart()->onResume())都会执行。

singleTop : 栈顶复用模式.这种模式下,如果新Activity已经位于任务栈的栈顶,那 么此Activity不会被重新创建,所以它的启动三回调就不会执行,同时Activity 的onNewIntent()方法会被回调.如果Activity已经存在但是不在栈顶,

那么作用与standard模式一样.

singleTask: 栈内复用模式.创建这样的Activity的时候,系统会先确认它所需任务 栈已经创建,否则先创建任务栈.然后放入Activity,如果栈中已经有一个Activity实例,那么这个Activity就会被调到栈顶,onNewIntent(),

并且singleTask会清理在当前Activity上面的所有Activity.(clear top)

singleInstance : 加强版的singleTask模式,这种模式的Activity只能单独位于一个任 务栈内,由于栈内复用的特性,后续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

### 六》Service的两种启动方法，有什么区别

1.在Context中通过public boolean bindService(Intent service,ServiceConnection conn,int flags)方法来进行Service与Context的关联并启动，并且Service的生命周期依附于Context(不求同时同分同秒生！但求同时同分同秒屎！！)。

2.通过public ComponentName startService(Intent service)方法去启动一个Service，

此时Service的生命周期与启动它的Context无关。

要注意的是，whatever，都需要在xml里注册你的Service，就像这样:

<service

android:name=".packnameName.youServiceName"

android:enabled="true" />

### 七》广播(Broadcast Receiver)的两种动态注册和静态注册有什么区别。

静态注册：在AndroidManifest.xml文件中进行注册，当App退出后，Receiver仍然可以接收到广播并且进行相应的处理

动态注册：在代码中动态注册，当App退出后，也就没办法再接受广播了

### 八》三类动画

1，帧动画 原理：一张张图片不断的切换，形成动画效果

2，补间动画 只是一个动画效果，组件其实还在原来的位置上，xy的坐标 并没有改变

3，属性动画 修改对象的属性，从而实现动画效果

### 九》进程间通信和同步（进程间通信的四种方式）

1，SP是进程同步的吗?有什么方法做到同步？

[Android](https://www.2cto.com/kf/yidong/Android/)本身的SP可以支持多进程，但是SP不能保证多进程间同步。因为当多个进程同时而又高频的调用SP的commit方法时，就会导致文件被反复覆盖写入，而并没有被及时读取，所以造成进程间数据的不同步

Android的ContentProvider是支持进程同步的，可以利用ContentProvider进行进程的sp同步

#### 1，Activity

  Activity的跨进程访问与进程内访问略有不同。虽然它们都需要Intent对象，但跨进程访问并不需要指定Context对象和Activity的 Class对象，而需要指定的是要访问的Activity所对应的Action（一个字符串）。有些Activity还需要指定一个Uri（通过 Intent构造方法的第2个参数指定）。

       在android系统中有很多应用程序提供了可以跨进程访问的Activity，例如，下面的代码可以直接调用拨打电话的Activity。

Intent callIntent = new  Intent(Intent.ACTION\_CALL, Uri.parse("tel:12345678" );

startActivity(callIntent);

#### 2，Content Provider

可以跨进程访问其他应用程序中的数据（以Cursor对象形式返回），当然，也可以对其他应用程序的数据进行增、删、改操 作

#### 3，广播

可以向android系统中所有应用程序发送广播，而需要跨进程通讯的应用程序可以监听这些广播

#### 4，AIDL

Service和Content Provider类似，也可以访问其他应用程序中的数据，但不同的是，Content Provider返回的是Cursor对象，而Service返回的是Java对象，这种可以跨进程通讯的服务叫AIDL服务

### 十》Dalvik，ART和jvm

Dalvik是Google公司自己设计用于Android平台的Java虚拟机

Dalvik和jvm的区别

Dalvik是基于寄存器的，运行dex文件

JVM是基于栈的，运行java字节码

Art：空间换时间

art模式需要在程序安装时进行预编译，将apk编译解析成机器码，运行速度相比dalvik模式快。

缺点：安装时间稍长，由于进行了预编译，所以会产生机器码，会占用存储空间。

ART和Dalvik区别

Art上应用启动快，运行快，但是耗费更多存储空间，安装时间长，总的来说ART的功效就是"空间换时间"

## 五，多线程相关

### 1，进程和线程的区别

进程：表示一个运行的程序

线程：进程(程序)的执行单元，执行路径

多进程的意义? 提高CPU的使用率

多线程的意义? 提高应用程序的使用率

### 2，多线程的三种实现方式

1，继承Thread类，

2，实现Runnable接口（推荐，方便的实现资源的共享）

3，通过Callable和Future创建线程

### 3，start和run方法的区别

start会先启动线程，再由jvm调用run方法

run方法只是thread的一个普通方法调用，还是在主线程里执行。

### 4，线程池

程序启动一个新线程成本比较高，因为它涉及到要与操作系统进行交互，而使用线程池可以很好的提高性能，尤其是当程序要创建大量生存期很短的线程时，更应该使用线程池。

线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。

JKD5之前，我们手动实现自己的线程池，JDK5以后，java内置支持线程池。

代码演示

//创建一个线程池对象，控制要创建几个线程对象

ExecutorService pool=Executors.newFixedThreadPool(2);

//可以执行Runnable对象或者Callable对象的线程

pool.submit(new MyRunnable());

pool.submit(new MyRunnable());

//结束线程池

pool.shutdown();

### 5，synchronized与Lock的区别

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率

Lock和synchronized有以下几点不同：

　　1）Lock是一个接口，jdk5后出现，而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

5）Lock可以提高多个线程进行读操作的效率。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择

### 6，volatile关键字

volatile关键字修饰变量，用来确保将变量的更新操作通知到其他线程

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序

### 7，ThreadLocal

ThreadLocal类的作用：是为每个线程都创建一个变量副本, 每个线程都可以修改自己所拥有的变量副本, 而不会影响其他线程的副本. 其实这也是解决线程安全的问题的一种方法。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性

ThreadLocal原理：在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本。

如线程中的Handler对象就是通过ThreadLocal来存放的

### 8，死锁

死锁就是指两个或者两个以上的线程在执行过程中，因争夺资源产生的一种相互等待现象。（比如两个人吃饭，一人一根筷子）

嵌套的代码体现

if(x>20){

synchronized(a){

synchronized(b){...}

}

}else{

synchronized(b){

synchronized(a){...}

}

}

### 9，AsyncTask的底层实现

 AsyncTask是对Handler与线程池的封装，AsyncTask的本质是一个线程池，所有提交的异步任务都会在这个线程池中的工作线程内执行，当工作线程需要跟UI线程交互时，工作线程会通过向在UI线程创建的Handler传递消息的方式，调用相关的回调函数，从而实现UI界面的更新

## 六，JVM内存模型与GC回收

#### 1，什么是jvm

JVM是Java Virtual Machine（Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机。

JVM在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行

#### 2，JDK,JRE,JVM的作用及关系

(1)作用

JVM：保证Java语言跨平台

JRE：Java程序的运行环境

JDK：Java程序的开发环境

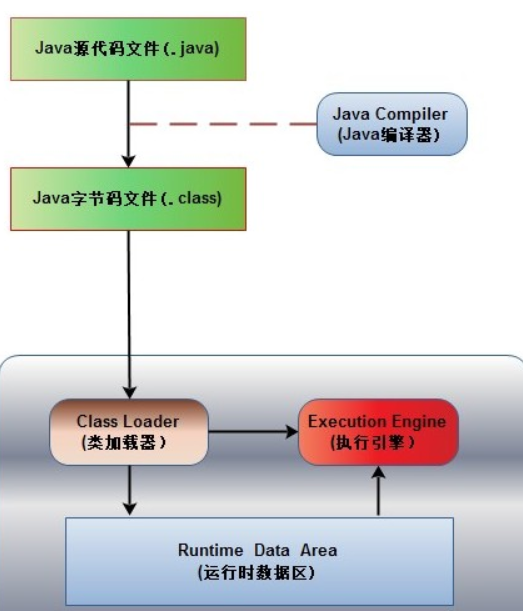
(2)关系

JDK：JRE+工具

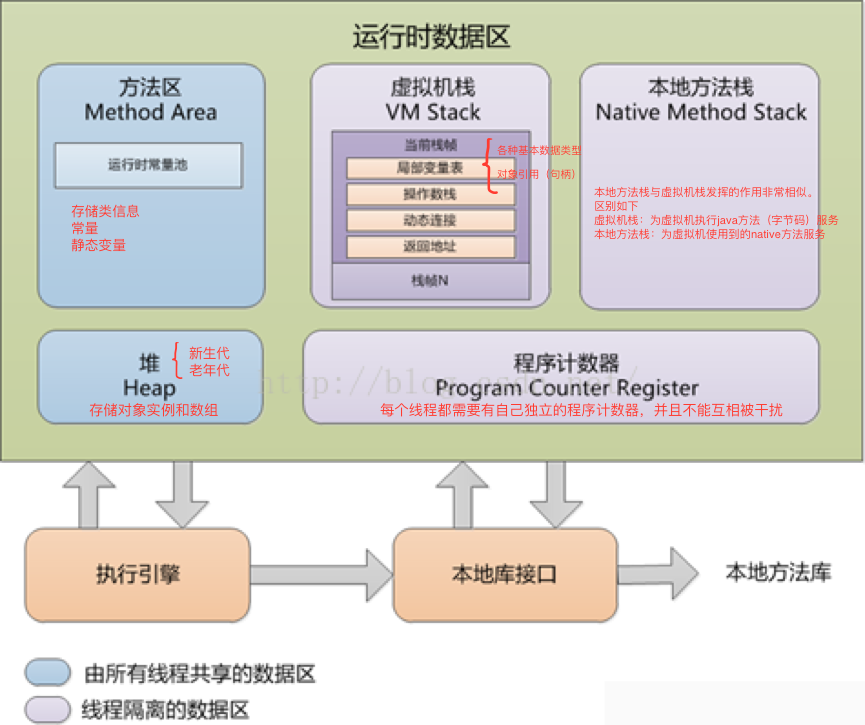
JRE：JVM+类库

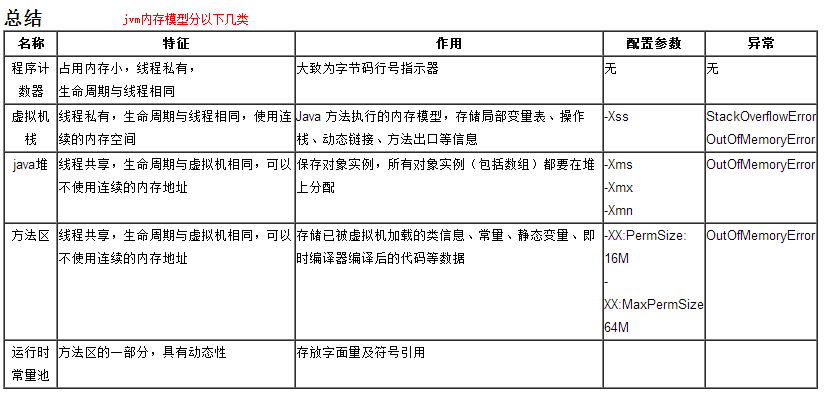
简单而言：使用JDK开发完成的Java程序，交给JRE去运行，由JVM来保证跨平台。

#### 3，java代码具体执行过程如下图



#### 4，jvm内存结构图（即运行时数据区）





程序计数器：是一块较小的内存空间，可以看作是当前线程所执行的字节码的

行号指示器。为了能够使得每个线程都在线程切换后能够恢复在切换之前

的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相

被干扰，独立存储。

如果正在执行的是一个java方法，记录的是正在执行的虚拟机字节码指令的地址

如果正在执行的是个native方法，这个计数器的值为空。

虚拟机栈：为虚拟机执行java方法（字节码）服务。

这里我们用的比较多的是局部变量表

局部变量表所需的内存空间在编译期间完成分配。当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的。在方法运行期间不会改变局部变量表的大小。

本地方法栈：为虚拟机执行本地方法服务。

本地方法栈与虚拟机栈发挥的作用非常相似。区别如下

虚拟机栈：为虚拟机执行java方法（字节码）服务

本地方法栈：为虚拟机使用到的native方法服务

堆：是用来存储对象本身的以及数组（数组引用是存放在Java栈中的）。堆是被所有 线程共享的，在JVM中只有一个堆。是java虚拟机管理的内存最大的一块。

在虚拟机启动时创建，

方法区：和堆一样，是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息，

常量，静态变量和即时编译器编译后的代码数据等。这个区域的内存回收主要是针对

常量池的回收和对类型的卸载。

方法区有以下特点

方法区是线程安全的。

2、方法区的大小不必是固定的，JVM可根据应用需要动态调整。

3、方法区也可被垃圾收集，当某个类不在被使用(不可触及)时，JVM将卸载这个类，进行垃圾收集

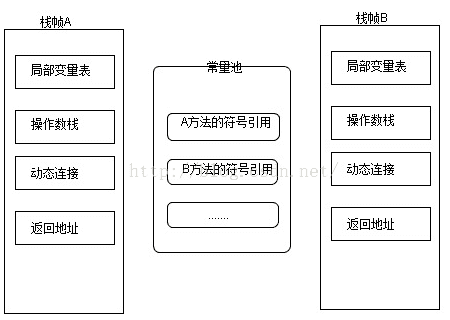
堆和栈的主要区别：

栈--主要存放引用和基本数据类型。

堆--用来存放 new 出来的对象实例和数组。

线程和jvm

每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。

每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过

#### 5，JVM执行程序的过程 1) 加载.class文件  2) 管理并分配内存  3) 执行垃圾收集

#### 6，jvm的生命周期

a) 启动。启动一个Java程序时，一个JVM实例就产生了，任何一个拥有public static void main(String[] args)函数的class都可以作为JVM实例运行的起点   
b) 运行。main()作为该程序初始线程的起点，任何其他线程均由该线程启动。JVM内部有两种线程：守护线程和非守护线程，main()属于非守护线程，守护线程通常由JVM自己使用，java程序也可以表明自己创建的线程是守护线程   
c) 消亡。当程序中的所有非守护线程都终止时，JVM才退出；若安全管理器允许，程序也可以使用Runtime类或者System.exit()来退出

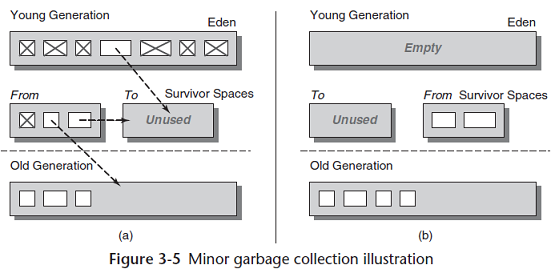
#### 7，GC的回收策略（垃圾回收策略机制）

垃圾回收：主要是对gc堆即Young Generation（年轻代）块和Old Generation（年老代）块内存进行回收，YG用来放新产生的对象，经过几次回收还没回收掉的对象往OG中移动，对YG进行垃圾回收又叫做MinorGC，对 OG垃圾回收又叫MajorGC，两块内存回收互不干涉

 Young generation（年轻代） 更细的划分为Eden和2个survivor space(即幸存区)

1.eden空间: 大部分新对象在这里分配，有些大对象直接在年老区分配，这个区域进行回收后很多情况下会变空。

2. two survivor 空间：即from survivor和to survivor。当eden区满时会将仍然存活的对象复制至其中一个survivor区，当这个区又满时将复制至另外一个survivor区，如下图：



## 七，mvc，mvp，mvvm

三个架构模式:  
MVC：Model-View-Controller，经典模式，很容易理解，主要缺点有两个：  
 1，View对Model的依赖，会导致View也包含了业务逻辑；  
 2，Controller会变得很厚很复杂。  
MVP：Model-View-Presenter，MVC的一个演变模式，将Controller换成了Presenter， 主要为了解决上述第一个缺点，将View和Model解耦，

不过第二个缺点依然没有解决。  
MVVM：Model-View-ViewModel，是对MVP的一个优化模式，

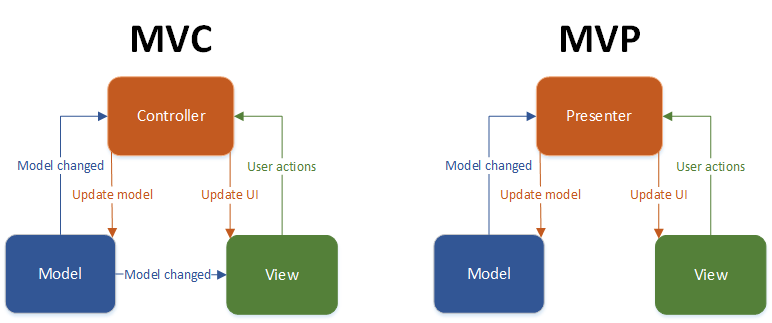
采用了双向绑定：View的变动，自动反映在ViewModel，反之亦然。

MVC, MVP, MMVM用来解决业务逻辑和视图之间的耦合

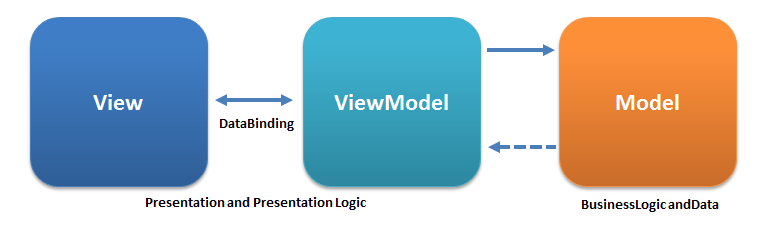
Mvc和mvp的最主要区别：

Mvc中model可以直接和view交互

mvp中model 与view 的交互由presenter完成



Mvvm的图解



Mvc，mvp，mvvm的优缺点

mvc是指用户触发事件的时候，view层会发送指令到controller层，然后controller去通知model层更新数据，model层更新完数据后会直接在view层显示结果。

对android来说 activity几乎承担了view层和controller层两种角色，并且和model层耦合严重，在逻辑复杂的界面维护起来很麻烦。

mvp模式下的activity只承担了view层的角色，controller的角色完全由presenter负责，view层和presenter层的通信通过接口实现，所以VP之间不存在耦合问题，view层与model也是完全解耦了。

presenter复用度高，可以随意搬到任何界面。

mvp模式下还方便测试维护：

但是mvp的问题在于view层和presenter层是通过接口连接，在复杂的界面中，维护过多接口的成本很大。

Mvvm 的优点：

1，和mvp一样，方便进行单元测试

2，便于代码的移植

Mvvm的缺点

1，类会增多

2，viewModel会越来越庞大

3，调用复杂度增加

## 八，多媒体（音视频图片）

### 一》Bitmap相关

#### 1，android 色彩模式说明：

ALPHA\_8： 每个像素占用1byte内存。

RGB\_565: 每个像素占用2byte内存

ARGB\_4444: 每个像素占用2byte内存

ARGB\_8888: 每个像素占用4byte内存

假设一张1024\*1024，模式为ARGB\_8888的图片，那么它占有的内存就是：1024\*1024\*4 = 4MB

Android默认的色彩模式为ARGB\_8888，通常我们优化Bitmap时，当需要做性能优化或者防止OOM（Out Of Memory），我们通常会使用Bitmap.Config.RGB\_565这个配置，因为Bitmap.Config.ALPHA\_8只有透明度，显示一般图片没有意义，Bitmap.Config.ARGB\_4444显示图片不清楚，Bitmap.Config.ARGB\_8888占用内存最多

#### 2,布局生成海报



#### 3，使用Bitmap时的一些注意事项

不用的Bitmap及时释放

if (!bmp.isRecycle()) {

bmp.recycle(); //回收图片所占的内存

bitmap = null;

system.gc(); //提醒系统及时回收

}

虽然调用recycle()并不能保证立即释放占用的内存，但是可以加速Bitmap的内存的释放。  
释放内存以后，就不能再使用该Bitmap对象了，如果再次使用，就会抛出异常。所以一定要保证不再使用的时候释放。比如，如果是在某个Activity中使用Bitmap，就可以在Activity的onStop()或者onDestroy()方法中进行回收

对于 Bitmap 对象在不使用时,我们应该先调用 recycle() 释放内存，然后才把它设置为 null.

因为加载 Bitmap 对象的内存空间，一部分是 java 的，一部分 C 的（因为 Bitmap 分配的底层是通过 JNI 调用的 ）。 而这个 recyle() 就是针对 C 部分的内存释放。

#### 4，圆形bitmap的实现原理

主要靠paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC\_IN));这行代码实现圆形图片

SRC\_IN这种模式，取两个绘制的效果叠加后的交集，第一个绘制的是个圆形，第二个绘制的是个Bitmap，于是交集为圆形，展现的是BItmap，就实现了圆形图片效果

## 九，自定义view

**view绘制流程三部曲**

View的绘制流程：OnMeasure()——>OnLayout()——>OnDraw()

OnMeasure：测量视图大小

OnLayout：确定view位置，进行页面布局

OnDraw：绘制视图

### OnMeasure

**View的onMeasure方法**

默认实现很简单，就是调用setMeasuredDimension()，setMeasuredDimension()可以简单理解就是给mMeasuredWidth和mMeasuredHeight设值，如果这两个值一旦设置了，那么意味着对于这个View的测量结束了，这个View的宽高已经有测量的结果出来了

父View把自己的MeasureSpec传递给子View结合子view自己LayoutParams算出子View的MeasureSpec，然后父容器传递给子容器的，然后让子View用这个MeasureSpec（一个测量要求，比如不能超过多大）去测量自己

如果子View是ViewGroup 那还会递归往下测量。这个计算过程，就是计算出来的父View的MeasureSpec不断往子View传递，结合子View的LayoutParams 一起再算出子View的MeasureSpec，然后继续传给子View，不断计算每个View的MeasureSpec，子View有了MeasureSpec才能测量自己和自己的子View。所有的孩子测量之后，经过一系类的计算之后通过setMeasuredDimension设置自己的宽高。总的来说，父view是等所有的子view测量结束之后，才来测量自己的。

父view的大小减去父view的padding+子view的margin，才能精确的算出子view的MeasureSpec的size。

MeasureSpec一共有三种模式

**1，UPSPECIFIED** : 父容器对于子容器没有任何限制,子容器想要多大就多大（未指定）  
**2，EXACTLY**: 父容器已经为子容器设置了尺寸,子容器应当服从这些边界,不论子容器想要多大的空间（父view确定大小）。  
**3，AT\_MOST**：子容器可以是声明大小内的任意大小（最大只能多大）

**子View的width或height是个精确值**

父EXACTLY: 子size为精确值，mode为EXACTLY

AT\_MOST: size为精确值，mode为EXACTLY

UPSPECIFIED: size为精确值，mode为EXACTLY

**子View的width或height为 MATCH\_PARENT/FILL\_PARENT**

EXACTLY: size为父视图大小，mode为EXACTLY

AT\_MOST: size为父视图大小，mode为AT\_MOST

UPSPECIFIED: size为0其值未定，mode为UPSPECIFIED

**子View的width或height为 WRAP\_CONTENT**

EXACTLY: size为父视图大小，mode为AT\_MOST

AT\_MOST: size为父视图大小，mode为AT\_MOST

UPSPECIFIED: size为0其值未定，mode为UPSPECIFIED

二》OnLayout

自定义viewgroup时，必须重载onlayout方法。重载的目的就是安排子view在父视图的具体位置。

1. OnDraw

## 十，安卓和js的交互

**JsBridge：**安卓和javascript最流行的交互方式，有以下三种方式可以实现交互传值

1，**addJavascriptInterface：**针对Android 4.2（API 17）及以上，只有标有@JavascriptInterface注解的public方法才能从js调用。而对targetSdkVersion为API Level 16及以下的app，js可以调用Java所有的public方法。

通过public void addJavascriptInterface (Object object, String name)方法把对象传递给js，

**2, loadUrl:**可以实现安卓调用js，通过webview.loadUrl("javascript:jsMethod()");字符串里的javascript: 是固定写法，后面的jsMethod可以替换成你js代码里的方法。也可以通过"javascript:jsMethod(" + jsonParams + ")";传递数据给js。这个缺点：如果js方法返回数据，这里会发生重定向。解决办法是用evaluateJavascript

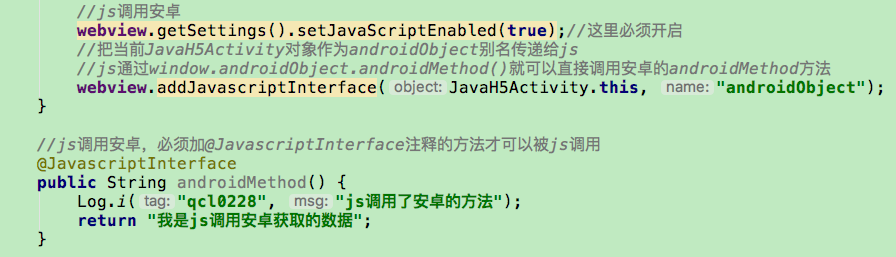
**3，evaluateJavascript：**安卓4.4以后可以实现安卓调用js，安卓可以传数据给js，并且可以获取js方法的返回值。

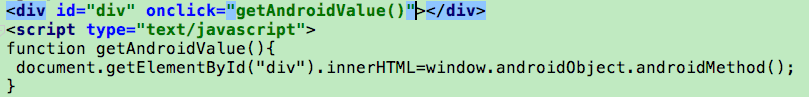
缺点：必须大于api19（4.4）才可以使用

**4，shouldOverrideUrlLoading：**通过这个方法拦截url，并解析url携带的参数，

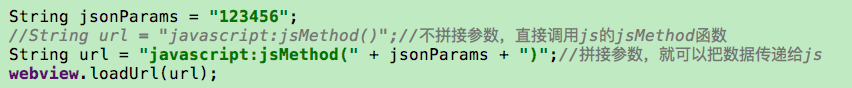
如：qiushi://setH5Info?params={"title"%3A"商品详情"}

### addJavascriptInterface：js调用安卓

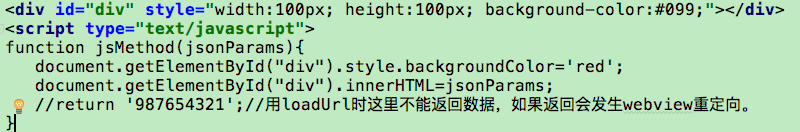


js代码如下

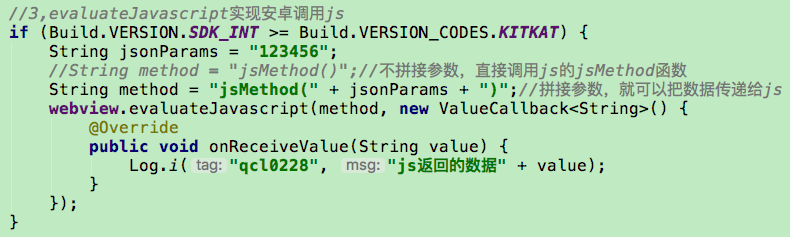
### 二》loadUrl：安卓调用js（单向传递数据，js不能返回数据）



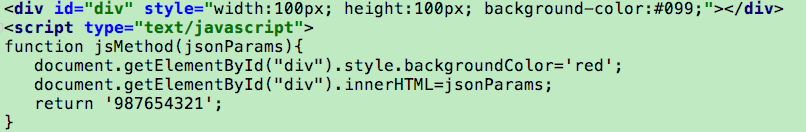
js代码如下



### 三》evaluateJavascript：安卓调用js（双向传递数据）

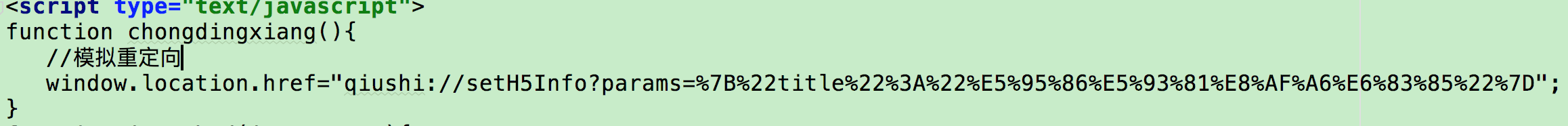


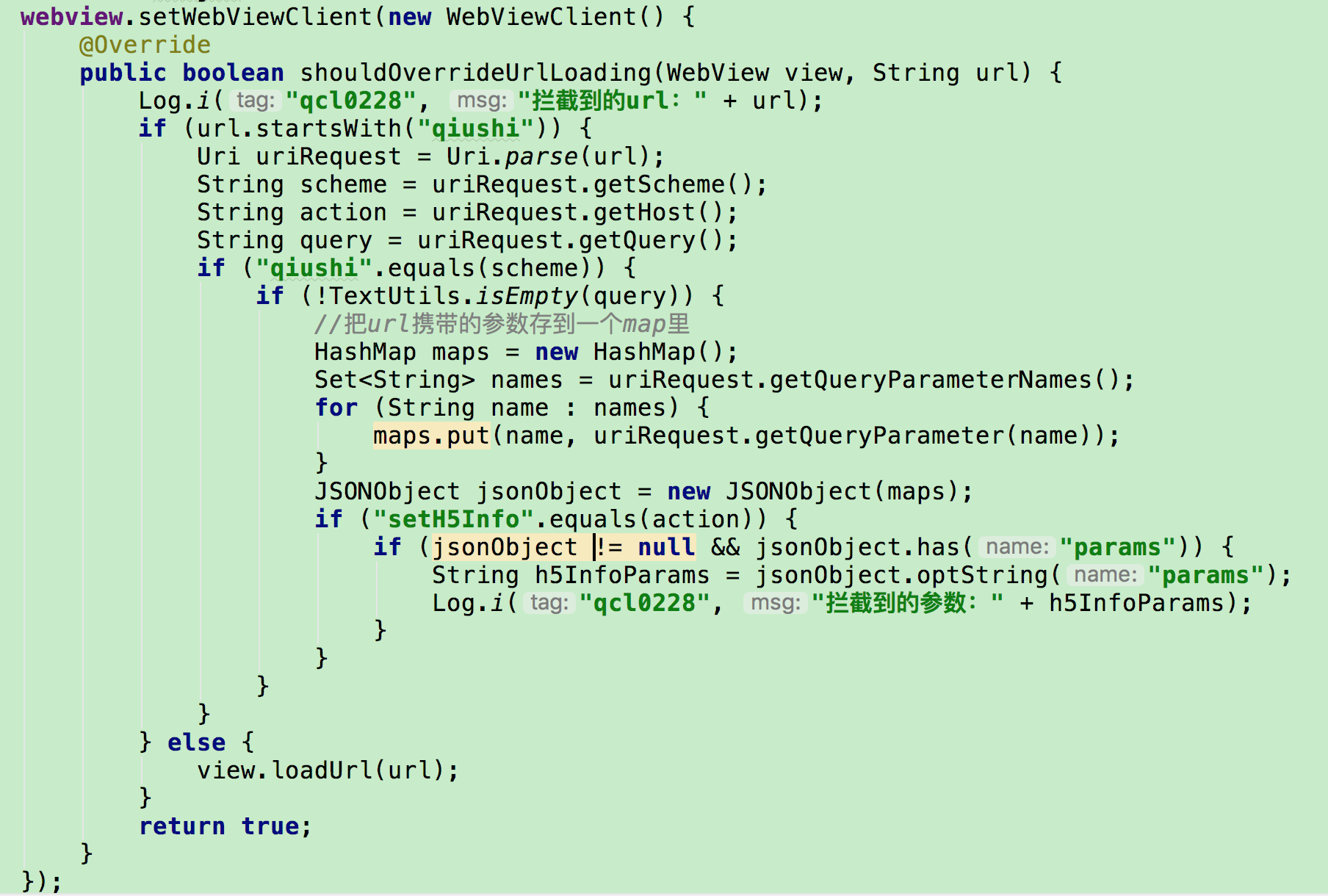
js代码如下



### 四》shouldOverrideUrlLoading：安卓获取js的数据

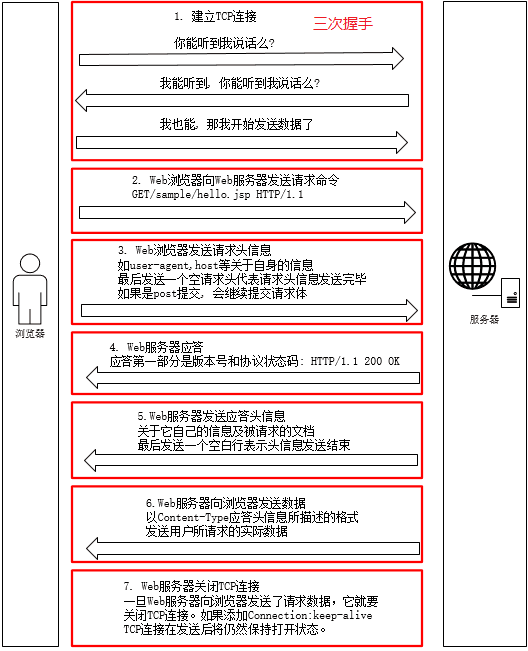
我们这里延时需要借助上面方法，在js实现重定向，只有重定向的url才可以在shouldoverrideurlloading拦截到。





## 十一，Http请求与响应

1，看下http请求的整个步骤图解



一次完整的HTTP通信过程中,客户端与Web服务器之间将完成

下列7个步骤:

1，建立tcp连接

客户端与服务器建立连接需要通过tcp完成，该协议与IP协议共同构建Internet, 即著名的TCP/IP协议族, 因此Internet又被称作是TCP/IP网络.

HTTP是比TCP更高层次的应用层协议, 根据规则, 只有低层协议建立之后才能进行更高层协议的连接, 因此, 首先要建立TCP连接。Tcp需要经过三次握手建立连接。可以拿我们打电话来比喻。如打电话，需要问你好。对方回答你好以后，我们才正式通话

2，客服端向服务器发送请求命令

一旦建立了TCP连接, Web浏览器就会向Web服务器发送请求命令

例如：GET/sample/hello.jsp HTTP/1.1

3， Web浏览器发送请求头（post需要发送请求体，get方法没有）

浏览器发送其请求命令之后, 还要以头信息的形式向Web服务器发送一些别的信息, 这些信息用来描述浏览器自己. 之后浏览器发送了一空白行来通知服务器, 表示它已经结束了该头信息的发送. 若是post请求, 还会在发送完请求头信息之后发送请求体.

4，Web服务器应答

客户机向服务器发出请求后, 服务器会向客户机回送应答.

HTTP/1.1 200 OK

应答的第一部分是协议的版本号和应答状态码

5， Web服务器发送应答头信息

正如客户端会随同请求发送关于自身的信息一样,服务器也会随同应答向用户发送关于它自己的数据及被请求的文档. 最后以一个空白行来表示头信息发送到此结束.

如下面是服务器返回的头信息

Connection: Keep-Alive

Content-Type: text/html; charset=UTF-8

Server: Apache/2.4.6 (CentOS) PHP/5.6.31

6，Web服务器向浏览器发送数据

Web服务器向浏览器发送头信息后, 它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据

7，Web服务器关闭TCP连接

一般情况下, 一旦Web服务器向浏览器发送了请求数据, 它就要关闭TCP连接.

如果浏览器或者服务器在其头信息加入了这行代码Connection:keep-alive

TCP在发送后将仍然保持连接状态. 于是, 浏览器可以继续通过相同的连接发送请求. 保持连接节省了每个请求建立新连接所需的时间, 还节约了网络带宽.

上面的七个步骤可以简单的总结为4步

1)，通过三次握手建立tcp连接

2)，发送请求信息请求头和请求体（get没有请求体）

3)，服务器返回应答头和数据

4)，关闭tcp连接（Connection:keep-alive可保活）

8，网络编程的三要素 ip、端口、协议

举例说明三要素：比如我们送快递。

1，要知道送到那个小区（IP地址，相当于知道送到那台电脑）

2，要知道送到小区里的那一用户家（端口，相当于电脑里的QQ）

3，用什么样的方式给用户（协议，用户需要给我提供姓名电话并签字）

A:IP地址

概述：IP地址是网络中计算机的唯一标识。

a:点分十进制

b:IP地址的组成：网络号段+主机号段。

c:IP地址的分类：

d:dos命令

ipconfig：查看本机ip地址: 192.168.1.139

ping 后面跟ip地址，测试本机与指定的ip地址间的通讯是否正常。

特殊的ip地址

127.0.0.1：回环地址（可以用来表示本机ip地址）

e:InetAddress

通过InetAddress类创建ip地址对象来获取主机名和ip地址值

//这里可以传入主机名或者ip地址

InetAddress ad=InetAddress.getByName("192.168.1.138");

String host=ad.getHostName();//获取主机名

String ip=ad.getHostAddress();//获取ip地址

B:端口

是应用程序的标识。范围：0-65535。其中0-1024是系统或者保留端口，不建议使用。

C:协议

UDP:数据打包,数据大小有限制（64k）,不连接,效率高,不可靠

TCP:建立数据连接通道,数据大小无限制,效率低,可靠

（面试题）UDP与TCP的区别?

1:在注重速度的时候使用UDP。比如：视频聊天时，

因为UDP不需要连接，速度快，每个数据包的大小限制在64k内，不安全，有可能丢失文件

2:在注重安全的时候使用TCP。比如：下载文件时

TCP需要建立连接，通过三次握手完成连接，速度稍慢，连接中不限制数据包的大小， 但是安全

## 十二，即时通讯和推送的3种实现方案

### 1，通过轮询：太low，还消耗性能，消耗流量

### 2，长连接：这个方案可以解决由轮询带来的性能问题，但是还是会消耗手机的电池

并且我们本地的服务很容易被安卓系统在低内存时杀死。长连接需要依赖心跳机制，

实现难度比较大

### 3，采用MQTT协议实现Android推送：耗电量小

MQTT协议较之XMPP更为轻量级，其连接的建立与传输的开销都非常小，非常精简，非常适合大量节点在弱网络环境的场景，发布/订阅的模式也比较易于扩展

XMPP是PC时代的产物，其底层通讯的数据格式的XML，数据冗余性太高(约70%)，比较耗流量，并且在复杂的移动网络环境下会遇到各种各样的问题。

# Java知识点

## 一，计算机资源

经典的冯诺依曼结构把计算机系统抽象成 CPU + 存储器 + IO，那么计算机资源无非就两种：

1. 计算资源

2. 存储资源

CPU：是计算单元，单纯从CPU的角度来说它是一个黑盒，CPU只提供了计算能力， 但是不负责分配计算资源。

计算资源：是操作系统来分配的，也就是常说的操作系统的调度模块，由操作系统按照一 定的规则来分配什么时候由谁来获得CPU的计算资源，比如分时间片

存储资源：就是内存，磁盘这些存储设备的资源

进程是操作系统分配资源的最小单元，

线程是操作系统调度的最小单元

## 二，进程和线程（详细看java分类笔记）

进程表示一个运行的程序

线程：进程(程序)的执行单元，执行路径

多进程的意义?提高CPU的使用率

多线程的意义?提高应用程序的使用率

在Java中实现多线程有两种手段，

1，继承Thread类，

2，实现Runnable接口（推荐，方便的实现资源的共享）

## 三，通过sort方法对数组或集合进行排序

数组排序

int[] a={1,4,-1,5,0};

Arrays.sort(a);

//数组a[]的内容变为{-1,0,1,4,5}

for(int i=0;i<a.length;i++)

System.out.print(a[i]+" ");

集合排序

List<Integer> list = new ArrayList<Integer>();

list.add(1);

list.add(3);

list.add(2);

System.out.println("排序前");

for (Object o : list) {

System.out.print(o);

}

Collections.sort(list);

System.out.println("排序后");

for (Object o : list) {

System.out.print(o);

}

把list转为set进行去重

List<String> list = new ArrayList<String>();

list.add("zhao");

list.add("zhao2");

Collections.addAll(list, "zhao","long","ri");//填充

System.out.println("去重前");

for (Object o : list) {

System.out.print(o+",");

}

Set<String> set=new HashSet<String>();

set.addAll(list);//给set填充

list.clear();//清空list，不然下次把set元素加入此list的时候是在原来的基础上追加元素的

list.addAll(set);//把set的

System.out.println("-------------");

for (Object o : list) {

System.out.print(o+",");

}

## 四，冒泡排序

//冒泡排序

public static void bubbleSort(int[] arr) {

for(int x=0; x<arr.length-1; x++) {

for(int y=0; y<arr.length-1-x; y++) {

if(arr[y] > arr[y+1]) {

int temp = arr[y];

arr[y] = arr[y+1];

arr[y+1] = temp;

}

}

}

}

## 五，集合的体系

Collection(单列集合)

|--List 有序(存储顺序和取出顺序一致)，可重复

|--ArrayList 底层数据结构是数组，查询快，增删慢。

线程不安全，效率高

|--Vector 底层数据结构是数组，查询快，增删慢。

线程安全，效率低。

|--LinkedList 底层数据结构是链表，查询慢，增删快。

线程不安全，效率高。

|--Set 无序,唯一

|--HashSet 无序唯一，底层数据结构是哈希表(是一个元素为链 表的数组)通过hashCode()和equals()保证元素唯一

|--TreeSet 排序唯一，底层数据结构是红黑树(是一个自平衡的 二叉树)

Map(双列集合)

|--HashMap 底层数据结构是哈希表。线程不安全，效率高， 哈希表的作用是用来保证键的唯一性的 |--LinkedHashMap 底层数据结构由链表和哈希表组成。

由链表保证元素有序。

由哈希表保证元素唯一。

|--Hashtable 底层数据结构是哈希表。线程安全，效率低

|--TreeMap 底层数据结构是红黑树。(是一种自平衡的二叉树)

到底使用那种集合

看需求

是否是键值对象形式:

是：Map

键是否需要排序:

是：TreeMap

否：HashMap

不知道，就使用HashMap。

否：Collection

元素是否唯一:

是：Set

元素是否需要排序:

是：TreeSet

否：HashSet

不知道，就使用HashSet

否：List

要安全吗:

是：Vector(其实我们也不用它,后面我们讲解了多线程以后，我在给你回顾用谁)

否：ArrayList或者LinkedList

增删多：LinkedList

查询多：ArrayList

不知道，就使用ArrayList

不知道，就使用ArrayList

所以我们使用最多的就是HashMap、HashSet、ArrayList集合

## 六，常见数据结构

A:堆栈 先进后出(子弹夹的例子)

B:队列 先进先出（过安检的例子）

C:数组 查询快，增删慢

D:链表 查询慢，增删快

E:哈希表；通过hashcode和equals来保证唯一