目录

[Android框架 3](#_Toc516411019)

[Retrofit（http） 3](#_Toc516411020)

[RxJava 4](#_Toc516411021)

[Retrofit + RxJava 5](#_Toc516411022)

[Android性能优化 6](#_Toc516411023)

[数据库优化 6](#_Toc516411024)

[布局优化 6](#_Toc516411025)

[1、抽象布局标签 6](#_Toc516411026)

[2、去除不必要的嵌套和View节点 11](#_Toc516411027)

[3、减少不必要的infalte 11](#_Toc516411028)

[4、其他点 12](#_Toc516411029)

[代码优化 13](#_Toc516411030)

[1、降低执行时间 13](#_Toc516411031)

[2、异步，利用多线程提高TPS 17](#_Toc516411032)

[3、提前或延迟操作，错开时间段提高TPS 17](#_Toc516411033)

[移动网络优化 17](#_Toc516411034)

[1、连接服务器优化策略 17](#_Toc516411035)

[2、获取数据优化策略 18](#_Toc516411036)

[3、其他优化手段 19](#_Toc516411037)

[Android Service 20](#_Toc516411038)

[Started Service 20](#_Toc516411039)

[Bound Service 21](#_Toc516411040)

[1）Extending the Binder class 21](#_Toc516411041)

[2）Using a Messenger 24](#_Toc516411042)

[3）AIDL（Android Interface Definition Language） 28](#_Toc516411043)

[Local Service VS Remote Service 28](#_Toc516411044)

[Service特性 29](#_Toc516411045)

[IntentService 29](#_Toc516411046)

[前台Service 30](#_Toc516411047)

[Android面试题 32](#_Toc516411048)

[Java部分 32](#_Toc516411049)

[1.Switch能否用string做参数？ 32](#_Toc516411050)

[2. equals与==的区别： 32](#_Toc516411051)

[3. Object有哪些公用方法？ 32](#_Toc516411052)

[4. 实际开发中软引用或者弱引用的使用场景： 32](#_Toc516411053)

[5. Hashcode的作用，与 equal 有什么区别 32](#_Toc516411054)

[6. String、StringBuffer与StringBuilder的区别 33](#_Toc516411055)

[7. Override和Overload的含义去区别 33](#_Toc516411056)

[8. 抽象类和接口的区别 33](#_Toc516411057)

[9.解析XML的几种方式的原理与特点：DOM、SAX、PULL 33](#_Toc516411058)

[10.wait()和sleep()的区别 33](#_Toc516411059)

[11.JAVA多态的实现原理 34](#_Toc516411060)

[12.JAVA 垃圾回收与内存分配策略 34](#_Toc516411061)

[13. Java 集合系列问题 35](#_Toc516411062)

[14.什么是反射机制，哪里能用到？ 35](#_Toc516411063)

[15.线程池及其作用 36](#_Toc516411064)

[Android部分 36](#_Toc516411065)

[1. Activity 系列问题 36](#_Toc516411066)

[2. Service系列问题 38](#_Toc516411067)

[3. Handle系列问题 39](#_Toc516411068)

[4. ListView系列问题 40](#_Toc516411069)

[5. JNI系列问题 41](#_Toc516411070)

[6. OOM系列问题 42](#_Toc516411071)

[7. ANR 系列问题 43](#_Toc516411072)

[8. Asynctask问题 44](#_Toc516411073)

[9. Android触摸分发机制 44](#_Toc516411074)

[10. Dalvik虚拟机系列问题 46](#_Toc516411075)

[11. 注册广播接收器有哪几种方式,有什么区别 46](#_Toc516411076)

[12. 显示Intent与隐式Intent的区别 46](#_Toc516411077)

[13. Android中的动画有哪些，区别是什么 46](#_Toc516411078)

[14.不使用动画，怎么实现一个动态的 View？ 47](#_Toc516411079)

[15. Postvalidata与Validata有什么区别？ 47](#_Toc516411080)

[16. 如何自定义ViewGroup？ 47](#_Toc516411081)

[17. View的绘制流程 47](#_Toc516411082)

[18. 数据持久化的四种方式有哪些？ 48](#_Toc516411083)

[19. fragement里面可以再嵌套fragment？ 48](#_Toc516411084)

[20. Socker编程的步骤 48](#_Toc516411085)

[21. Activity中如何动态的添加Fragment 48](#_Toc516411086)

[22. Scrollview怎么判断是否滑倒底部 48](#_Toc516411087)

[23. 什么是 MVC 模式？MVC 模式的好处是什么？ 48](#_Toc516411088)

[24. 应用如何避免被第三方杀掉？ 48](#_Toc516411089)

[25.Context与ApplicationContext的区别，分别用在什么情况下 49](#_Toc516411090)

# Android框架

## Retrofit（http）

<https://www.jianshu.com/p/308f3c54abdd>

1. **public** **class** Example01 {
2. **public** **interface** BlogService {
3. @GET("blog/{id}") //这里的{id} 表示是一个变量
4. Call<ResponseBody> getBlog(/\*\* 这里的id表示的是上面的{id} \*/@Path("id") **int** id);
5. }
7. **public** **static** **void** main(String[] args) **throws** IOException {
8. Retrofit retrofit = **new** Retrofit.Builder()
9. .baseUrl("http://localhost:4567/")
10. .build();
12. BlogService service = retrofit.create(BlogService.**class**);
13. Call<ResponseBody> call = service.getBlog(2);
14. // 用法和OkHttp的call如出一辙
15. // 不同的是如果是Android系统回调方法执行在主线程
16. call.enqueue(**new** Callback<ResponseBody>() {
17. @Override
18. **public** **void** onResponse(
19. Call<ResponseBody> call, Response<ResponseBody> response) {
20. **try** {
21. System.out.println(response.body().string());
22. } **catch** (IOException e) {
23. e.printStackTrace();
24. }
25. }
27. @Override
28. **public** **void** onFailure(Call<ResponseBody> call, Throwable t) {
29. t.printStackTrace();
30. }
31. });
32. }
33. }

## RxJava

<https://blog.csdn.net/sqm8822/article/details/79142458>

1. Observable.from(folders)
2. .flatMap(**new** Func1<File, Observable<File>>() {
3. @Override
4. **public** Observable<File> call(File file) {
5. **return** Observable.from(file.listFiles());
6. }
7. })
8. .filter(**new** Func1<File, Boolean>() {
9. @Override
10. **public** Boolean call(File file) {
11. **return** file.getName().endsWith(".png");
12. }
13. })
14. .map(**new** Func1<File, Bitmap>() {
15. @Override
16. **public** Bitmap call(File file) {
17. **return** getBitmapFromFile(file);
18. }
19. })
20. .subscribeOn(Schedulers.io())
21. .observeOn(AndroidSchedulers.mainThread())
22. .subscribe(**new** Action1<Bitmap>() {
23. @Override
24. **public** **void** call(Bitmap bitmap) {
25. imageCollectorView.addImage(bitmap);
26. }
27. });

## Retrofit + RxJava

<https://www.jianshu.com/p/94f6370ed18d?from=timeline&isappinstalled=1>

先设计interface

1. **public** **interface** GetWeatherService {
2. @GET("weather\_mini")
3. Observable<WeatherEntity> getRxMessage(@Query("city") String city);
4. }

再调用

1. Retrofit retrofit = **new** Retrofit.Builder()
2. .baseUrl("http://wthrcdn.etouch.cn/")//基础URL
3. .addConverterFactory(GsonConverterFactory.create())//设置 Json 转换器
4. .addCallAdapterFactory(RxJavaCallAdapterFactory.create())//RxJava 适配器
5. .build();
7. GetWeatherService weatherService = retrofit.create(GetWeatherService.**class**);
9. weatherService.getRxMessage("北京")
10. .subscribeOn(Schedulers.io())//IO线程加载数据
11. .observeOn(AndroidSchedulers.mainThread())//主线程显示数据
12. .subscribe(**new** Subscriber<WeatherEntity>() {
13. @Override
14. **public** **void** onCompleted() {
16. }
18. @Override
19. **public** **void** onError(Throwable e) {
21. }
23. @Override
24. **public** **void** onNext(WeatherEntity weatherEntity) {
25. Log.e(TAG,"RxJava + Retrofit= " + weatherEntity.getData().getGanmao());
26. }
27. });

# Android性能优化

在性能测试中存在两个概念：

(1). 响应时间

指从用户操作开始到系统给用户以正确反馈的时间。一般包括逻辑处理时间 + 网络传输时间 + 展现时间。对于非网络类应用不包括网络传输时间。展现时间即网页或 App 界面渲染时间。响应时间是用户对性能最直接的感受。

(2). TPS(Transaction Per Second)

TPS为每秒处理的事务数，是系统吞吐量的指标，在搜索系统中也用QPS(Query Per Second)衡量。TPS一般与响应时间反相关。通常所说的性能问题就是指响应时间过长、系统吞吐量过低。

对后台开发来说，也常将高并发下内存泄漏归为性能问题。

对移动开发来说，性能问题还包括电量、内存使用这两类较特殊情况。

性能调优方式

明白了何为性能问题之后，就能明白性能优化实际就是优化系统的响应时间，提高TPS。优化响应时间，提高TPS。方式不外乎这三大类：

(1) 降低执行时间

a. 利用多线程并发或分布式提高 TPS

b. 缓存(包括对象缓存、IO 缓存、网络缓存等)

c. 数据结构和算法优化

d. 性能更优的底层接口调用，如 JNI 实现

e. 逻辑优化

f. 需求优化

(2) 同步改异步，利用多线程提高TPS

(3) 提前或延迟操作，错峰提高TPS

## 数据库优化

待完成

## 布局优化

主要介绍使用抽象布局标签(include, viewstub, merge)、去除不必要的嵌套和View节点、减少不必要的infalte及其他Layout方面可调优点，顺带提及布局调优相关工具(hierarchy viewer和lint)。

### 1、抽象布局标签

(1) <include>标签

include标签常用于将布局中的公共部分提取出来供其他layout共用，以实现布局模块化，这在布局编写方便提供了大大的便利。下面以在一个布局main.xml中用include引入另一个布局foot.xml为例。main.mxl代码如下：

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<RelativeLayout** xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent" **>**
6. **<ListView**
7. android:id="@+id/simple\_list\_view"
8. android:layout\_width="match\_parent"
9. android:layout\_height="match\_parent"
10. android:layout\_marginBottom="@dimen/dp\_80" **/>**
12. **<include** layout="@layout/foot.xml" **/>**
14. **</RelativeLayout>**

其中include引入的foot.xml为公用的页面底部，代码如下：

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<RelativeLayout** xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent" **>**
6. **<Button**
7. android:id="@+id/button"
8. android:layout\_width="match\_parent"
9. android:layout\_height="@dimen/dp\_40"
10. android:layout\_above="@+id/text"**/>**
12. **<TextView**
13. android:id="@+id/text"
14. android:layout\_width="match\_parent"
15. android:layout\_height="@dimen/dp\_40"
16. android:layout\_alignParentBottom="true"
17. android:text="@string/app\_name" **/>**
19. **</RelativeLayout>**

<include>标签唯一需要的属性是layout属性，指定需要包含的布局文件。可以定义android:id和android:layout\_\*属性来覆盖被引入布局根节点的对应属性值。注意重新定义android:id后，子布局的顶结点id就变化了。

(2) <viewstub>标签

viewstub标签同include标签一样可以用来引入一个外部布局，不同的是，viewstub引入的布局默认不会扩张，既不会占用显示也不会占用位置，从而在解析layout时节省cpu和内存。

viewstub常用来引入那些默认不会显示，只在特殊情况下显示的布局，例如进度布局、网络失败显示的刷新布局、信息出错出现的提示布局等。

下面以在一个布局main.xml中加入网络错误时的提示页面network\_error.xml为例。main.mxl代码如下：

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<RelativeLayout** xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent" **>**
6. ……
7. **<ViewStub**
8. android:id="@+id/network\_error\_layout"
9. android:layout\_width="match\_parent"
10. android:layout\_height="match\_parent"
11. android:layout="@layout/network\_error" **/>**
13. **</RelativeLayout>**

其中network\_error.xml为只有在网络错误时才需要显示的布局，默认不会被解析，示例代码如下：

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<RelativeLayout** xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent" **>**
6. **<Button**
7. android:id="@+id/network\_setting"
8. android:layout\_width="@dimen/dp\_160"
9. android:layout\_height="wrap\_content"
10. android:layout\_centerHorizontal="true"
11. android:text="@string/network\_setting" **/>**
13. **<Button**
14. android:id="@+id/network\_refresh"
15. android:layout\_width="@dimen/dp\_160"
16. android:layout\_height="wrap\_content"
17. android:layout\_below="@+id/network\_setting"
18. android:layout\_centerHorizontal="true"
19. android:layout\_marginTop="@dimen/dp\_10"
20. android:text="@string/network\_refresh" **/>**
22. **</RelativeLayout>**

在java中通过(ViewStub)findViewById(id)找到ViewStub，通过stub.inflate()展开ViewStub，然后得到子View，如下：

1. **private** View networkErrorView;
3. **private** **void** showNetError() {
4. // not repeated infalte
5. **if** (networkErrorView != **null**) {
6. networkErrorView.setVisibility(View.VISIBLE);
7. **return**;
8. }
10. ViewStub stub = (ViewStub)findViewById(R.id.network\_error\_layout);
11. networkErrorView = stub.inflate();
12. Button networkSetting = (Button)networkErrorView.findViewById(R.id.network\_setting);
13. Button refresh = (Button)findViewById(R.id.network\_refresh);
14. }
16. **private** **void** showNormal() {
17. **if** (networkErrorView != **null**) {
18. networkErrorView.setVisibility(View.GONE);
19. }
20. }

在上面showNetError()中展开了ViewStub，同时我们对networkErrorView进行了保存，这样下次不用继续inflate。这就是后面第三部分提到的减少不必要的infalte。另外，将一个view设置为GONE不会被解析，从而提高layout解析速度，而VISIBLE和INVISIBLE这两个可见性属性会被正常解析。

viewstub标签大部分属性同include标签类似。

(3) <merge>标签

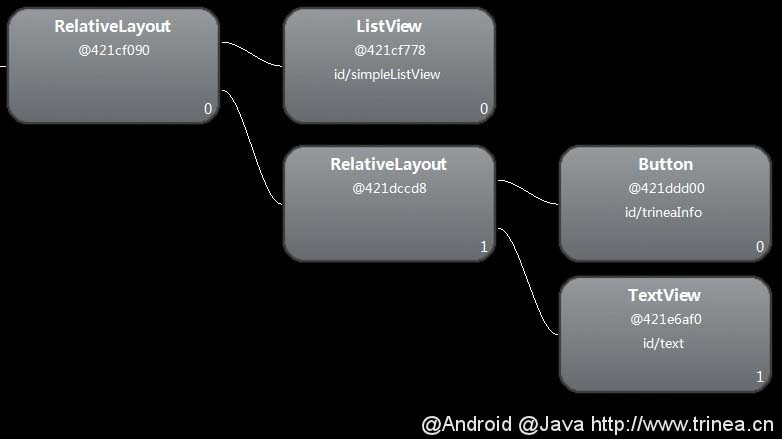
在使用了include后可能导致布局嵌套过多，多余不必要的layout节点，从而导致解析变慢，不必要的节点和嵌套可通过hierarchy viewer(下面布局调优工具中有具体介绍)或设置->开发者选项->显示布局边界查看。

merge标签可用于两种典型情况：

a. 布局顶结点是FrameLayout且不需要设置background或padding等属性，可以用merge代替，因为Activity内容视图的parent view就是个FrameLayout，所以可以用merge消除只剩一个。

b. 某布局作为子布局被其他布局include时，使用merge当作该布局的顶节点，这样在被引入时顶结点会自动被忽略，而将其子节点全部合并到主布局中。

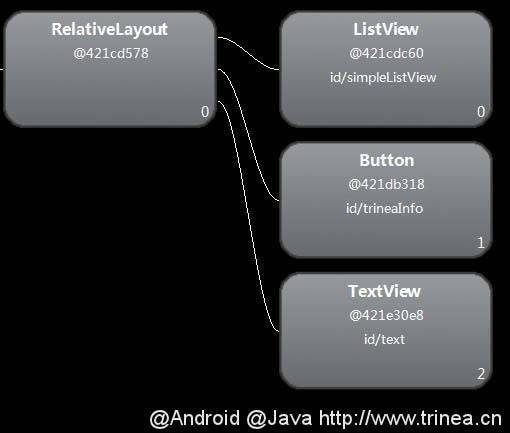
以(1) <include>标签的示例为例，用hierarchy viewer查看main.xml布局如下图：



可以发现多了一层没必要的RelativeLayout，将foot.xml中RelativeLayout改为merge，如下：

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<merge** xmlns:android="http://schemas.android.com/apk/res/android"
3. android:layout\_width="match\_parent"
4. android:layout\_height="match\_parent" **>**
6. **<Button**
7. android:id="@+id/button"
8. android:layout\_width="match\_parent"
9. android:layout\_height="@dimen/dp\_40"
10. android:layout\_above="@+id/text"**/>**
12. **<TextView**
13. android:id="@+id/text"
14. android:layout\_width="match\_parent"
15. android:layout\_height="@dimen/dp\_40"
16. android:layout\_alignParentBottom="true"
17. android:text="@string/app\_name" **/>**
19. **</merge>**

运行后再次用hierarchy viewer查看main.xml布局如下图：



这样就不会有多余的RelativeLayout节点了。

### 2、去除不必要的嵌套和View节点

(1) 首次不需要使用的节点设置为GONE或使用viewstub

(2) 使用RelativeLayout代替LinearLayout

大约在Android4.0之前，新建工程的默认main.xml中顶节点是LinearLayout，而在之后已经改为RelativeLayout，因为RelativeLayout性能更优，且可以简单实现LinearLayout嵌套才能实现的布局。

4.0及以上Android版本可通过设置->开发者选项->显示布局边界打开页面布局显示，看看是否有不必要的节点和嵌套。4.0以下版本可通过hierarchy viewer查看。

### 3、减少不必要的infalte

(1) 对于inflate的布局可以直接缓存，用全部变量代替局部变量，避免下次需再次inflate

如上面ViewStub示例中的

1. **if** (networkErrorView != **null**) {
2. networkErrorView.setVisibility(View.VISIBLE);
3. **return**;
4. }

(2) ListView提供了item缓存，adapter getView的标准写法，如下：

1. @Override
2. **public** View getView(**int** position, View convertView, ViewGroup parent) {
3. ViewHolder holder;
4. **if** (convertView == **null**) {
5. convertView = inflater.inflate(R.layout.list\_item, **null**);
6. holder = **new** ViewHolder();
7. ……
8. convertView.setTag(holder);
9. } **else** {
10. holder = (ViewHolder)convertView.getTag();
11. ……
12. }
13. }
15. /\*\*
16. \* ViewHolder
17. \*
18. \* @author trinea@trinea.cn 2013-08-01
19. \*/
20. **private** **static** **class** ViewHolder {
22. ImageView appIcon;
23. TextView  appName;
24. TextView  appInfo;
25. }

### 4、其他点

(1) 用SurfaceView或TextureView代替普通View

SurfaceView或TextureView可以通过将绘图操作移动到另一个单独线程上提高性能。

普通View的绘制过程都是在主线程(UI线程)中完成，如果某些绘图操作影响性能就不好优化了，这时我们可以考虑使用SurfaceView和TextureView，他们的绘图操作发生在UI线程之外的另一个线程上。

因为SurfaceView在常规视图系统之外，所以无法像常规试图一样移动、缩放或旋转一个SurfaceView。TextureView是Android4.0引入的，除了与SurfaceView一样在单独线程绘制外，还可以像常规视图一样被改变。

(2) 使用RenderJavascript

RenderScript是Adnroid3.0引进的用来在Android上写高性能代码的一种语言，语法给予C语言的C99标准，他的结构是独立的，所以不需要为不同的CPU或者GPU定制代码代码。

(3) 使用OpenGL绘图

Android支持使用OpenGL API的高性能绘图，这是Android可用的最高级的绘图机制，在游戏类对性能要求较高的应用中得到广泛使用。

Android 4.3最大的改变，就是支持OpenGL ES 3.0。相比2.0，3.0有更多的缓冲区对象、增加了新的着色语言、增加多纹理支持等等，将为Android游戏带来更出色的视觉体验。

(4) 尽量为所有分辨率创建资源

减少不必要的硬件缩放，这会降低UI的绘制速度，可借助Android asset studio

## 代码优化

### 1、降低执行时间

这部分包括：缓存、数据存储优化、算法优化、JNI、逻辑优化、需求优化几种优化方式。

(1). 缓存

缓存主要包括对象缓存、IO缓存、网络缓存、DB缓存等。对象缓存能减少内存的分配，IO缓存减少磁盘的读写次数，网络缓存减少网络传输，DB缓存较少Database的访问次数。

在内存、文件、数据库、网络的读写速度中，内存都是最优的，且速度快，所以尽量将需要频繁访问或访问一次消耗较大的数据存储在缓存中。

Android中常使用缓存:

**a.线程池**

1 每次new Thread新建对象性能差。

2 线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。

3 缺乏更多功能，如定时执行、定期执行、线程中断。

相比new Thread，Java提供的四种线程池的好处在于：

1 重用存在的线程，减少对象创建、消亡的开销，性能佳。

2 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。

3 提供定时执行、定期执行、单线程、并发数控制等功能。

Java通过Executors提供四种线程池，分别为：

1 newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

1. ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. **try** {
5. Thread.sleep(index \* 1000);
6. } **catch** (InterruptedException e) {
7. e.printStackTrace();
8. }
10. cachedThreadPool.execute(**new** Runnable() {
12. @Override
13. **public** **void** run() {
14. System.out.println(index);
15. }
16. });
17. }

2 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

1. ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. fixedThreadPool.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

3 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

1. ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
2. scheduledThreadPool.schedule(**new** Runnable() {
4. @Override
5. **public** **void** run() {
6. System.out.println("delay 3 seconds");
7. }
8. }, 3, TimeUnit.SECONDS);

4 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

1. ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. singleThreadExecutor.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

**b.Android图片缓存，Android图片Sdcard缓存，数据预取缓存**

可以使用图片框架如Picasso等

**c.消息缓存**

通过handler.obtainMessage复用之前的message，如下：

1. handler.sendMessage(handler.obtainMessage(0, object));

**d. ListView缓存**

1. @Override
2. **public** View getView(**int** position, View convertView, ViewGroup parent) {
3. ViewHolder holder;
4. **if** (convertView == **null**) {
5. convertView = inflater.inflate(R.layout.list\_item, **null**);
6. holder = **new** ViewHolder();
7. ……
8. convertView.setTag(holder);
9. } **else** {
10. holder = (ViewHolder)convertView.getTag();
11. ……
12. }
13. }
15. /\*\*
16. \* ViewHolder
17. \*
18. \* @author trinea@trinea.cn 2013-08-01
19. \*/
20. **private** **static** **class** ViewHolder {
22. ImageView appIcon;
23. TextView  appName;
24. TextView  appInfo;
25. }

**e. 网络缓存**

数据库缓存http response，根据http头信息中的Cache-Control域确定缓存过期时间。

**f. 文件IO缓存**

使用具有缓存策略的输入流，BufferedInputStream替代InputStream，BufferedReader替代Reader，BufferedReader替代BufferedInputStream.对文件、网络IO皆适用。

**g. layout缓存**

**上节中提到的使用<viewstub>标签**

**h. 其他需要频繁访问或访问一次消耗较大的数据缓存**

(2). 数据存储优化

包括数据类型、数据结构的选择。

**a. 数据类型选择**

字符串拼接用StringBuilder代替String，在非并发情况下用StringBuilder代替StringBuffer。如果你对字符串的长度有大致了解，如100字符左右，可以直接new StringBuilder(128)指定初始大小，减少空间不够时的再次分配。

64位类型如long double的处理比32位如int慢

使用SoftReference、WeakReference相对正常的强应用来说更有利于系统垃圾回收

final类型存储在常量区中读取效率更高

LocalBroadcastManager代替普通BroadcastReceiver，效率和安全性都更高

**b. 数据结构选择**

常见的数据结构选择如：

1 ArrayList和LinkedList的选择，ArrayList根据index取值更快，LinkedList更占内存、随机插入删除更快速、扩容效率更高。一般推荐ArrayList。

2 ArrayList、HashMap、LinkedHashMap、HashSet的选择，hash系列数据结构查询速度更优，ArrayList存储有序元素，HashMap为键值对数据结构，LinkedHashMap可以记住加入次序的hashMap，HashSet不允许重复元素。

3 HashMap、WeakHashMap选择，WeakHashMap中元素可在适当时候被系统垃圾回收器自动回收，所以适合在内存紧张型中使用。

4 Collections.synchronizedMap和ConcurrentHashMap的选择，ConcurrentHashMap为细分锁，锁粒度更小，并发性能更优。Collections.synchronizedMap为对象锁，自己添加函数进行锁控制更方便。

Android也提供了一些性能更优的数据类型，如SparseArray、SparseBooleanArray、SparseIntArray、Pair。

Sparse系列的数据结构是为key为int情况的特殊处理，采用二分查找及简单的数组存储，加上不需要泛型转换的开销，相对Map来说性能更优。不过我不太明白为啥默认的容量大小是10，是做过数据统计吗，还是说现在的内存优化不需要考虑这些东西，写16会死吗，还是建议大家根据自己可能的容量设置初始值。

(3). 算法优化

这个主题比较大，需要具体问题具体分析，尽量不用O(n\*n)时间复杂度以上的算法，必要时候可用空间换时间。

查询考虑hash和二分，尽量不用递归。可以从结构之法 算法之道或微软、Google等面试题学习。

(4). JNI

Android应用程序大都通过Java开发，需要Dalvik的JIT编译器将Java字节码转换成本地代码运行，而本地代码可以直接由设备管理器直接执行，节省了中间步骤，所以执行速度更快。不过需要注意从Java空间切换到本地空间需要开销，同时JIT编译器也能生成优化的本地代码，所以糟糕的本地代码不一定性能更优。

(5). 逻辑优化

这个不同于算法，主要是理清程序逻辑，减少不必要的操作。

(6). 需求优化

这个就不说了，对于sb的需求可能带来的性能问题，只能说做为一个合格的程序员不能只是执行者，要学会说NO。不过不能拿这种接口敷衍产品经理哦。

### 2、异步，利用多线程提高TPS

充分利用多核Cpu优势，利用线程解决密集型计算、IO、网络等操作。

关于多线程可参考：Java线程池

在Android应用程序中由于系统ANR的限制，将可能造成主线程超时操作放入另外的工作线程中。在工作线程中可以通过handler和主线程交互。

### 3、提前或延迟操作，错开时间段提高TPS

(1) 延迟操作

不在Activity、Service、BroadcastReceiver的生命周期等对响应时间敏感函数中执行耗时操作，可适当delay。

Java中延迟操作可使用ScheduledExecutorService，不推荐使用Timer.schedule;

Android中除了支持ScheduledExecutorService之外，还有一些delay操作，如

handler.postDelayed，handler.postAtTime，handler.sendMessageDelayed，View.postDelayed，AlarmManager定时等。

(2) 提前操作

对于第一次调用较耗时操作，可统一放到初始化中，将耗时提前。如得到壁纸wallpaperManager.getDrawable();

## 移动网络优化

### 1、连接服务器优化策略

(1). 不用域名，用 IP 直连

省去 DNS 解析过程，DNS 全名 Domain Name System，解析意指根据域名得到其对应的 IP 地址。 如 http://www.codekk.com 的域名解析结果就是 104.236.147.76。

首次域名解析一般需要几百毫秒，可通过直接向 IP 而非域名请求，节省掉这部分时间，同时可以预防域名劫持等带来的风险。

当然为了安全和扩展考虑，这个 IP 可能是一个动态更新的 IP 列表，并在 IP 不可用情况下通过域名访问。

(2). 服务器合理部署

服务器多运营商多地部署，一般至少含三大运营商、南中北三地部署。配合上面说到的动态 IP 列表，支持优先级，每次根据地域、网络类型等选择最优的服务器 IP 进行连接。对于服务器端还可以调优服务器的 TCP 拥塞窗口大小、重传超时时间(RTO)、最大传输单元(MTU)等。

### 2、获取数据优化策略

(1). 连接复用

节省连接建立时间，如开启 keep-alive。

Http 1.1 默认启动了 keep-alive。对于 Android 来说默认情况下 HttpURLConnection 和 HttpClient 都开启了 keep-alive。只是 2.2 之前 HttpURLConnection 存在影响连接池的 Bug，具体可见：Android HttpURLConnection 及 HttpClient 选择

(2). 请求合并

即将多个请求合并为一个进行请求，比较常见的就是网页中的 CSS Image Sprites。 如果某个页面内请求过多，也可以考虑做一定的请求合并。

(3). 减小请求数据大小

1 对于 POST 请求，Body 可以做 Gzip 压缩，如日志。

2 对请求头进行压缩

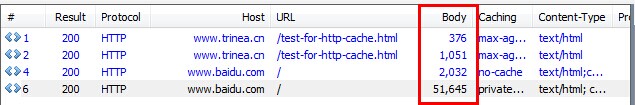
(4). CDN 缓存静态资源

缓存常见的图片、JS、CSS 等静态资源。

(5). 减小返回数据大小

1 压缩

一般 API 数据使用 Gzip 压缩，下图是之前测试的 Gzip 压缩前后对比图。



2 精简数据格式

如 JSON 代替 XML，WebP 代替其他图片格式。关注公众号 codekk，回复 20 查看关于 WebP 的介绍。

3 对于不同的设备不同网络返回不同的内容 如不同分辨率图片大小。

4 增量更新

需要数据更新时，可考虑增量更新。如常见的服务端进行 bsdiff，客户端进行 bspatch。

5 大文件下载

支持断点续传，并缓存 Http Resonse 的 ETag 标识，下次请求时带上，从而确定是否数据改变过，未改变则直接返回 304。

(6). 数据缓存

缓存获取到的数据，在一定的有效时间内再次请求可以直接从缓存读取数据。

### 3、其他优化手段

这类优化方式在性能优化系列总篇中已经有过完整介绍

1. 预取

包括预连接、预取数据。

2. 分优先级、延迟部分请求

将不重要的请求延迟，这样既可以削峰减少并发、又可以和后面类似的请求做合并。

3. 多连接

对于较大文件，如大图片、文件下载可考虑多连接。 需要控制请求的最大并发量，毕竟移动端网络受限。

四、监控

优化需要通过数据对比才能看出效果，所以监控系统必不可少，通过前后端的数据监控确定调优效果。

# Android Service

从Service的启动方式上，可以将Service分为Started Service和Bound Service。无论哪种具体的Service启动类型，都是通过继承Service基类自定义而来。

## Started Service

Started Service相对比较简单，通过context.startService(Intent serviceIntent)启动Service，context.stopService(Intent serviceIntent)停止此Service。当然，在Service内部，也可以通过stopSelf(...)方式停止其本身。当Client调用startService(Intent serviceIntent)启动Service时，Client可以将参数通过Intent直接传递给Service。Service执行过程中，如果需要将参数传递给Client，一般可以通过借助于发送广播的方式（此时，Client需要注册此广播）。

当Client调用startService(Intent serviceIntent)后，如果MyService是第一次启动，首先会执行 onCreate()回调，然后再执行onStartCommand(Intent intent, int flags, int startId)，当Client再次调用startService(Intent serviceIntent)，将只执行onStartCommand(Intent intent, int flags, int startId)，因为此时Service已经创建了，无需执行onCreate()回调。无论多少次的startService，只需要一次stopService()即可将此Service终止，执行onDestroy()函数（其实很好理解，因为onDestroy()与onCreate()回调是相对的）。

1.onCreate(Client首次startService(..)) >> onStartCommand >> onStartCommand - optional ... >> onDestroy(Client调用stopService(..))

注：onStartCommand(..)可以多次被调用，onDestroy()与onCreate()想匹配，当用户强制kill掉进程时，onDestroy()是不会执行的。

2.对于同一类型的Service，Service实例一次永远只存在一个，而不管Client是否是相同的组件，也不管Client是否处于相同的进程中。

3.Service通过startService(..)启动Service后，此时Service的生命周期与Client本身的什么周期是没有任何关系的，只有Client调用stopService(..)或Service本身调用stopSelf(..)才能停止此Service。当然，当用户强制kill掉Service进程或系统因内存不足也可能kill掉此Service。

4.Client A 通过startService(..)启动Service后,可以在其他Client（如Client B、Client C）通过调用stopService(..)结束此Service。

5.Client调用stopService(..)时，如果当前Service没有启动，也不会出现任何报错或问题，也就是说，stopService(..)无需做当前Service是否有效的判断。

6.startService(Intent serviceIntent)，其中的intent既可以是显式Intent，也可以是隐式Intent，当Client与Service同处于一个App时，一般推荐使用显示Intent。当处于不同App时，只能使用隐式Intent。

当Service需要运行在单独的进程中，AndroidManifest.xml声明时需要通过android:process指明此进程名称，当此Service需要对其他App开放时，android:exported属性值需要设置为true(当然，在有intent-filter时默认值就是true)。

## Bound Service

相对于Started Service，Bound Service具有更多的知识点。Bound Service的主要特性在于Service的生命周期是依附于Client的生命周期的，当Client不存在时，Bound Service将执行onDestroy，同时通过Service中的Binder对象可以较为方便进行Client-Service通信。Bound Service一般使用过程如下：

1.自定义Service继承基类Service，并重写onBind(Intent intent)方法，此方法中需要返回具体的Binder对象；

2.Client通过实现ServiceConnection接口来自定义ServiceConnection，并通过bindService (Intent service, ServiceConnection sc, int flags)方法将Service绑定到此Client上；

3.自定义的ServiceConnection中实现onServiceConnected(ComponentName name, IBinder binder)方法，获取Service端Binder实例；

4.通过获取的Binder实例进行Service端其他公共方法的调用，以完成Client-Service通信；

5.当Client在恰当的生命周期（如onDestroy等）时，此时需要解绑之前已经绑定的Service，通过调用函数unbindService(ServiceConnection sc)。

在Bound Service具体使用过程中，根据onBind(Intent intent)方法放回的Binder对象的定义方式不同，又可以将其分为以下三种方式，且每种方式具有不同的特点和适用场景：

### **1）Extending the Binder class**

这是Bound Service中最常见的一种使用方式，也是Bound Service中最简单的一种。

局限：Clinet与Service必须同属于同一个进程，不能实现进程间通信（IPC）。否则则会出现类似于“android.os.BinderProxy cannot be cast to xxx”错误。

下面通过代码片段看下具体的使用：

1. **public** **class** MyBindService **extends** Service {
3. **public** **static** **final** String TAG = "MyBindService";
5. **private** MyBinder mBinder = **new** MyBinder();
7. **public** **class** MyBinder **extends** Binder {
8. MyBindService getService() {
9. **return** MyBindService.**this**;
10. }
11. }
13. @Override
14. **public** IBinder onBind(Intent intent) {
15. Log.w(TAG, "in onBind");
16. **return** mBinder;
17. }
18. }
19. **public** **class** BActivity **extends** Activity {
21. **public** **static** **final** String TAG = "BActivity";
23. **private** Button bindServiceBtn;
24. **private** Button unbindServiceBtn;
26. **private** Button startIntentService;
28. **private** Intent serviceIntent;
30. **private** ServiceConnection sc = **new** MyServiceConnection();
31. **private** MyBinder mBinder;
32. **private** MyBindService mBindService;
33. **private** **boolean** mBound;
35. **private** **class** MyServiceConnection **implements** ServiceConnection {
37. @Override
38. **public** **void** onServiceConnected(ComponentName name, IBinder binder) {
39. Log.w(TAG, "in MyServiceConnection onServiceConnected");
40. mBinder = (MyBinder) binder;
41. mBindService = mBinder.getService();
43. mBound = **true**;
44. }
46. @Override
47. **public** **void** onServiceDisconnected(ComponentName name) {
48. // This is called when the connection with the service has been
49. // unexpectedly disconnected -- that is, its process crashed.
50. Log.w(TAG, "in MyServiceConnection onServiceDisconnected");
51. mBound = **false**;
52. }
54. }
56. @Override
57. **protected** **void** onCreate(Bundle savedInstanceState) {
58. **super**.onCreate(savedInstanceState);
59. setContentView(R.layout.b);
61. bindServiceBtn = (Button) findViewById(R.id.bind\_service);
62. unbindServiceBtn = (Button) findViewById(R.id.unbind\_service);
63. startIntentService = (Button) findViewById(R.id.start\_intentservice);
65. bindServiceBtn.setOnClickListener(**new** View.OnClickListener() {
66. @Override
67. **public** **void** onClick(View v) {
68. Intent intent = **new** Intent(BActivity.**this**, MyBindService.**class**);
69. bindService(intent, sc, Context.BIND\_AUTO\_CREATE);
70. }
71. });
73. unbindServiceBtn.setOnClickListener(**new** View.OnClickListener() {
74. @Override
75. **public** **void** onClick(View v) {
76. excuteUnbindService();
77. }
78. });
80. startIntentService.setOnClickListener(**new** View.OnClickListener() {
81. @Override
82. **public** **void** onClick(View v) {
83. Intent intent = **new** Intent(BActivity.**this**, MyIntentService.**class**);
84. startService(intent);
85. }
86. });
88. }
90. **private** **void** excuteUnbindService() {
91. **if** (mBound) {
92. unbindService(sc);
93. mBound = **false**;
94. }
95. }
97. @Override
98. **protected** **void** onDestroy() {
99. **super**.onDestroy();
100. Log.w(TAG, "in onDestroy");
101. excuteUnbindService();
102. }
103. }

注：在四大基本组件中，需要注意的的是BroadcastReceiver不能作为Bound Service的Client，因为BroadcastReceiver的生命周期很短，当执行完onReceive(..)回调时，BroadcastReceiver生命周期完结。而Bound Service又与Client本身的生命周期相关，因此，Android中不允许BroadcastReceiver去bindService(..)，当有此类需求时，可以考虑通过startService(..)替代。

### **2）Using a Messenger**

Messenger，在此可以理解成”信使“，通过Messenger方式返回Binder对象可以不用考虑Clinet - Service是否属于同一个进程的问题，并且，可以实现Client - Service之间的双向通信。极大方便了此类业务需求的实现。

局限：不支持严格意义上的多线程并发处理，实际上是以队列去处理

下面直接看下具体的使用：

1. **public** **class** MyMessengerService **extends** Service {
3. **public** **static** **final** String TAG = "MyMessengerService";
5. **public** **static** **final** **int** MSG\_FROM\_CLIENT\_TO\_SERVER = 1;
6. **public** **static** **final** **int** MSG\_FROM\_SERVER\_TO\_CLIENT = 2;
8. **private** Messenger mClientMessenger;
9. **private** Messenger mServerMessenger = **new** Messenger(**new** ServerHandler());
11. @Override
12. **public** IBinder onBind(Intent intent) {
13. Log.w(TAG, "in onBind");
14. **return** mServerMessenger.getBinder();
15. }
17. **class** ServerHandler **extends** Handler {
18. @Override
19. **public** **void** handleMessage(Message msg) {
20. Log.w(TAG, "thread name:" + Thread.currentThread().getName());
21. **switch** (msg.what) {
22. **case** MSG\_FROM\_CLIENT\_TO\_SERVER:
23. Log.w(TAG, "receive msg from client");
24. mClientMessenger = msg.replyTo;
26. // service发送消息给client
27. Message toClientMsg = Message.obtain(**null**, MSG\_FROM\_SERVER\_TO\_CLIENT);
28. **try** {
29. Log.w(TAG, "server begin send msg to client");
30. mClientMessenger.send(toClientMsg);
31. } **catch** (RemoteException e) {
32. e.printStackTrace();
33. }
34. **break**;
35. **default**:
36. **super**.handleMessage(msg);
37. }
38. }
39. }
41. @Override
42. **public** **boolean** onUnbind(Intent intent) {
43. Log.w(TAG, "in onUnbind");
44. **return** **super**.onUnbind(intent);
45. }
47. @Override
48. **public** **void** onDestroy() {
49. Log.w(TAG, "in onDestroy");
50. **super**.onDestroy();
51. }
52. }
53. **public** **class** CActivity **extends** Activity {
55. **public** **static** **final** String TAG = "CActivity";
57. **private** Button bindServiceBtn;
58. **private** Button unbindServiceBtn;
59. **private** Button sendMsgToServerBtn;
61. **private** ServiceConnection sc = **new** MyServiceConnection();
62. **private** **boolean** mBound;
64. **private** Messenger mServerMessenger;
66. **private** Handler mClientHandler = **new** MyClientHandler();
67. **private** Messenger mClientMessenger = **new** Messenger(mClientHandler);
69. **private** **class** MyClientHandler **extends** Handler {
70. @Override
71. **public** **void** handleMessage(Message msg) {
72. **if** (msg.what == MyMessengerService.MSG\_FROM\_SERVER\_TO\_CLIENT) {
73. Log.w(TAG, "reveive msg from server");
74. }
75. }
76. }
78. **private** **class** MyServiceConnection **implements** ServiceConnection {
80. @Override
81. **public** **void** onServiceConnected(ComponentName name, IBinder binder) {
82. Log.w(TAG, "in MyServiceConnection onServiceConnected");
83. mServerMessenger = **new** Messenger(binder);
85. mBound = **true**;
86. }
88. @Override
89. **public** **void** onServiceDisconnected(ComponentName name) {
90. // This is called when the connection with the service has been
91. // unexpectedly disconnected -- that is, its process crashed.
92. Log.w(TAG, "in MyServiceConnection onServiceDisconnected");
94. mBound = **false**;
95. }
96. }
98. @Override
99. **protected** **void** onCreate(Bundle savedInstanceState) {
100. **super**.onCreate(savedInstanceState);
101. setContentView(R.layout.c);
103. bindServiceBtn = (Button) findViewById(R.id.bind\_service);
104. unbindServiceBtn = (Button) findViewById(R.id.unbind\_service);
105. sendMsgToServerBtn = (Button) findViewById(R.id.send\_msg\_to\_server);
107. bindServiceBtn.setOnClickListener(**new** View.OnClickListener() {
108. @Override
109. **public** **void** onClick(View v) {
110. Intent intent = **new** Intent(CActivity.**this**, MyMessengerService.**class**);
111. bindService(intent, sc, Context.BIND\_AUTO\_CREATE);
112. }
113. });
115. unbindServiceBtn.setOnClickListener(**new** View.OnClickListener() {
116. @Override
117. **public** **void** onClick(View v) {
118. excuteUnbindService();
119. }
120. });
122. sendMsgToServerBtn.setOnClickListener(**new** View.OnClickListener() {
123. @Override
124. **public** **void** onClick(View v) {
125. sayHello();
126. }
127. });
129. **new** Handler().postDelayed(**new** Runnable() {
130. @Override
131. **public** **void** run() {
132. Intent intent = **new** Intent(CActivity.**this**, MyAlarmBroadcastReceiver.**class**);
133. sendBroadcast(intent);
134. }
135. }, 3 \* 1000);
137. }
139. **public** **void** sayHello() {
140. **if** (!mBound)
141. **return**;
142. // Create and send a message to the service, using a supported 'what' value
143. Message msg = Message.obtain(**null**, MyMessengerService.MSG\_FROM\_CLIENT\_TO\_SERVER, 0, 0);
144. // 通过replyTo把client端的Messenger(信使)传递给service
145. msg.replyTo = mClientMessenger;
146. **try** {
147. mServerMessenger.send(msg);
148. } **catch** (RemoteException e) {
149. e.printStackTrace();
150. }
151. }
153. **private** **void** excuteUnbindService() {
154. **if** (mBound) {
155. unbindService(sc);
156. mBound = **false**;
157. }
158. }
160. @Override
161. **protected** **void** onDestroy() {
162. **super**.onDestroy();
163. Log.w(TAG, "in onDestroy");
164. excuteUnbindService();
165. }
166. }

其中，需要注意的几点是：

1.MyMessengerService自定中，通过new Messenger(new ServerHandler())创建Messenger对象，在onBind(..)回调中，通过调用Messenger对象的getBinder()方法，将Binder返回；

2.Client在ServiceConnection的onServiceConnected(..)的回调中，通过new Messenger(binder)获取到Service传递过来的mServerMessenger；

3.接下来，就可以通过mServerMessenger.send(msg)方法向Service发送message，Service中的Messenger构造器中的Handler即可接收到此信息，在handleMessage(..)回调中处理；

4.至此只是完成了从Client发送消息到Service，同样的道理，想实现Service发送消息到Client，可以在客户端定义一个Handler，并得到相应的Messenger，在Clinet发送消息给Service时，通过msg.replyTo = mClientMessenger方式将Client信使传递给Service；

5.Service接收到Client信使后，获取此信使，并通过mClientMessenger.send(toClientMsg)方式将Service消息发送给Client。

至此，完成了Client - Service之间的双向通信流程。

### **3）AIDL（Android Interface Definition Language）**

一般情况下，Messenger这种方式都是可以满足需求的，当然，通过自定义AIDL方式相对更加灵活。

这种方式需要自己在项目中自定义xxx.aidl文件，然后系统会自动在gen目录下生成相应的接口类文件，接下来整个的流程与Messenger方式差别不大

注：无论哪种方式的Bound Service，在进行unbind(..)操作时，都需要注意当前Service是否处于已经绑定状态，否则可能会因为当前Service已经解绑后继续执行unbind(..)会导致崩溃。这点与Started Service区别很大（如前文所述：stopService(..)无需做当前Service是否有效的判断）。

## Local Service VS Remote Service

Local Service：不少人又称之为”本地服务“，是指Client - Service同处于一个进程；

Remote Service：又称之为”远程服务“，一般是指Service处于单独的一个进程中。

其他使用上上文中基本上都有所述。

## Service特性

1.Service本身都是运行在其所在进程的主线程（如果Service与Clinet同属于一个进程，则是运行于UI线程），但Service一般都是需要进行”长期“操作，所以经常写法是在自定义Service中处理”长期“操作时需要新建线程，以免阻塞UI线程或导致ANR；

2.Service一旦创建，需要停止时都需要显示调用相应的方法（Started Service需要调用stopService(..)或Service本身调用stopSelf(..)， Bound Service需要调用unbindService(..)），否则对于Started Service将处于一直运行状态，对于Bound Service，当Client生命周期结束时也将有此问题。也就是说，Service执行完毕后，必须人为的去停止它。

## IntentService

IntentService是系统提供给我们的一个已经继承自Service类的特殊类，IntentService特殊性是相对于Service本身的特性而言的：

1.默认直接实现了onBind(..)方法，直接返回null，并定义了抽象方法onHandlerIntent(..)，用户自定义子类时，需要实现此方法；

2.默认实现onStartCommand(), 将请求Intent添加到队列中。

3.onHandlerIntent(..)主要就是用来处于相应的”长期“任务的，并且已经自动在新的线程中，用户无语自定义新线程；

4.当”长期“任务执行完毕后（也就是onHandlerIntent(..)执行完毕后），此IntentService将自动结束，无需人为调用方法使其结束；

5.IntentService处于任务时，也是按照队列的方式一个个去处理，而非真正意义上的多线程并发方式。

下面是一个基本的继承自IntentService的自定义Service：

1. **public** **class** MyIntentService **extends** IntentService {
3. **public** **static** **final** String TAG = "MyIntentService";
5. **public** MyIntentService() {
6. **super**(TAG);
7. }
9. **public** MyIntentService(String name) {
10. **super**(name);
11. }
13. @Override
14. **protected** **void** onHandleIntent(Intent intent) {
15. Log.w(TAG, "in onHandleIntent");
16. Log.w(TAG, "thread name:" + Thread.currentThread().getName());
17. }
19. }

onStartCommand提供默认实现，将请求Intent添加到队列中；

## 前台Service

Android中Service接口中还提供了一个称之为”前台Service“的概念。通过Service.startForeground (int id, Notification notification)方法可以将此Service设置为前台Service。在UI显示上，notification将是一个处于onGoing状态的通知，使得前台Service拥有更高的进程优先级，并且Service可以直接notification通信。

下面是一个简单的前台Service使用实例：

1. **public** **class** MyService **extends** Service {
3. **public** **static** **final** String TAG = "MyService";
5. @Override
6. **public** IBinder onBind(Intent intent) {
7. **return** **null**;
8. }
10. @Override
11. **public** **void** onCreate() {
12. **super**.onCreate();
13. Log.w(TAG, "in onCreate");
14. }
16. @Override
17. **public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {
18. Log.w(TAG, "in onStartCommand");
19. Log.w(TAG, "MyService:" + **this**);
20. String name = intent.getStringExtra("name");
21. Log.w(TAG, "name:" + name);

24. Notification notification = **new** Notification(R.drawable.ic\_launcher, "test", System.currentTimeMillis());
25. Intent notificationIntent = **new** Intent(**this**, DActivity.**class**);
26. PendingIntent pendingIntent = PendingIntent.getActivity(**this**, 0, notificationIntesnt, 0);
27. notification.setLatestEventInfo(**this**, "title", "content", pendingIntent);
28. startForeground(1, notification);

31. **return** START\_REDELIVER\_INTENT;
32. }
34. @Override
35. **public** **void** onDestroy() {
36. **super**.onDestroy();
37. Log.w(TAG, "in onDestroy");
38. }
39. }

# Android面试题

## Java部分

### 1.Switch能否用string做参数？

在 Java 7 之前， switch 只能支持 byte 、 short 、 char 、 int 或者其对应的封装类以及 Enum 类型。在 Java 7 中， String 支持被加上了。

### 2. equals与==的区别：

==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同

### 3. Object有哪些公用方法？

方法equals测试的是两个对象是否相等

方法clone进行对象拷贝

方法getClass返回和当前对象相关的Class对象

方法notify,notifyall,wait都是用来对给定对象进行线程同步的

### 4. 实际开发中软引用或者弱引用的使用场景：

**利用软引用和弱引用解决OOM问题**：用一个HashMap来保存图片的路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占用的空间，从而有效地避免了OOM的问题 通过软可及对象重获方法实现Java对象的高速缓存:比如我们创建了一Employee的类，如果每次需要查询一个雇员的信息。哪怕是几秒中之前刚刚查询过的，都要重新构建一个实例，这是需要消耗很多时间的。我们可以通过软引用和 HashMap 的结合，先是保存引用方面：以软引用的方式对一个Employee对象的实例进行引用并保存该引用到HashMap 上，key 为此雇员的 id，value为这个对象的软引用，另一方面是取出引用，缓存中是否有该Employee实例的软引用，如果有，从软引用中取得。如果没有软引用，或者从软引用中得到的实例是null，重新构建一个实例，并保存对这个新建实例的软引用

### 5. Hashcode的作用，与 equal 有什么区别

同样用于鉴定2个对象是否相等的，java集合中有 list 和 set 两类，其中 set不允许元素重复实现，那个这个不允许重复实现的方法，如果用 equal 去比较的话，如果存在1000个元素，你 new 一个新的元素出来，需要去调用1000次 equal 去逐个和他们比较是否是同一个对象，这样会大大降低效率。hashcode实际上是返回对象的存储地址，如果这个位置上没有元素，就把元素直接存储在上面，如果这个位置上已经存在元素，这个时候才去调用equal方法与新元素进行比较，相同的话就不存了，散列到其他地址上

### 6. String、StringBuffer与StringBuilder的区别

String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象 StringBuffer和StringBuilder底层是 char[]数组实现的 StringBuffer是线程安全的，而StringBuilder是线程不安全的

### 7. Override和Overload的含义去区别

Overload顾名思义是重新加载，它可以表现类的多态性，可以是函数里面可以有相同的函数名但是参数名、返回值、类型不能相同；或者说可以改变参数、类型、返回值但是函数名字依然不变。 Override顾名思义就是ride(重写)的意思，在子类继承父类的时候子类中可以定义某方法与其父类有相同的名称和参数，当子类在调用这一函数时自动调用子类的方法，而父类相当于被覆盖（重写）了。

### 8. 抽象类和接口的区别

一个类只能继承单个类，但是可以实现多个接口 接口强调特定功能的实现，而抽象类强调所属关系 抽象类中的所有方法并不一定要是抽象的，你可以选择在抽象类中实现一些基本的方法。而接口要求所有的方法都必须是抽象的

### 9.解析XML的几种方式的原理与特点：DOM、SAX、PULL

DOM：消耗内存：先把xml文档都读到内存中，然后再用DOM API来访问树形结构，并获取数据。这个写起来很简单，但是很消耗内存。要是数据过大，手机不够牛逼，可能手机直接死机

SAX：解析效率高，占用内存少，基于事件驱动的：更加简单地说就是对文档进行顺序扫描，当扫描到文档(document)开始与结束、元素(element)开始与结束、文档(document)结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结束。

PULL：与 SAX 类似，也是基于事件驱动，我们可以调用它的next（）方法，来获取下一个解析事件（就是开始文档，结束文档，开始标签，结束标签），当处于某个元素时可以调用XmlPullParser的getAttributte()方法来获取属性的值，也可调用它的nextText()获取本节点的值。

### 10.wait()和sleep()的区别

sleep来自Thread类，和wait来自Object类

调用sleep()方法的过程中，线程不会释放对象锁。而 调用 wait 方法线程会释放对象锁

sleep睡眠后不出让系统资源，wait让出系统资源其他线程可以占用CPU

sleep(milliseconds)需要指定一个睡眠时间，时间一到会自动唤醒

### 11.JAVA多态的实现原理

抽象的来讲，多态的意思就是同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用） 实现的原理是动态绑定，程序调用的方法在运行期才动态绑定，追溯源码可以发现，JVM 通过参数的自动转型来找到合适的办法。

### 12.JAVA 垃圾回收与内存分配策略

1 垃圾回收是什么？

就是释放那些不再持有引用的对象的内存

2怎么判断一个对象是否需要收集？

A 引用计数（最简单古老的方法）：指将资源（可以是对象、内存或磁盘空间等等）的被引用次数保存起来，当被引用次数变为零时就将其释放的过程

B 对象引用遍历（现在大多数 jvm 使用的方法）：对象引用遍历从一组对象开始，沿着整个对象图上的每条链接，递归确定可到达（reachable）的对象。如果某对象不能从这些根对象中的任何一个到达，则将它作为垃圾收集

C 引用计数缺陷：引用计数无法解决循环引用问题：假设对象A，B都已经被实例化，让A=B,B=A,除此之外这两个对象再无任何引用，此时计数器的值就永远不可能为0，但是引用计数器无法通知gc回收他们

3 Java的四种引用的区别

A 强引用：如果一个对象具有强引用，它就不会被垃圾回收器回收。即使当前内存空间不足，JVM 也不会回收它，而是抛出 OutOfMemoryError 错误，使程序异常终止。如果想中断强引用和某个对象之间的关联，可以显式地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象

B 软引用：在使用软引用时，如果内存的空间足够，软引用就能继续被使用，而不会被垃圾回收器回收，只有在内存不足时，软引用才会被垃圾回收器回收。

C 弱引用：具有弱引用的对象拥有的生命周期更短暂。因为当 JVM 进行垃圾回收，一旦发现弱引用对象，无论当前内存空间是否充足，都会将弱引用回收。不过由于垃圾回收器是一个优先级较低的线程，所以并不一定能迅速发现弱引用对象

D 虚引用：顾名思义，就是形同虚设，如果一个对象仅持有虚引用，那么它相当于没有引用，在任何时候都可能被垃圾回收器回收。

4 介绍垃圾回收机制

A 标记回收法：遍历对象图并且记录可到达的对象，以便删除不可到达的对象，一般使用单线程工作并且可能产生内存碎片

B 标记-压缩回收法：前期与第一种方法相同，只是多了一步，将所有的存活对象压缩到内存的一端，这样内存碎片就可以合成一大块可再利用的内存区域，提高了内存利用率

C 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。这种方法适用于短生存期的对象，持续复制长生存期的对象则导致效率降低。

D 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，年轻代的特点是对象会很快被回收，因此在年轻代使用效率比较高的算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老年的内存空间，老年代则采取标记-压缩算法

5 JAVA 中堆和栈的区别

基本数据类型如**变量**和**对象的引用**都是在栈分配的

堆内存用来存放由new创建的对象和数组

类变量（static修饰的变量）:程序在一加载的时候就在堆中为类变量分配内存，堆中的内存地址存放在栈中

实例变量：当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量，是根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的"物理位置”,实例变量的生命周期--当实例变量的引用丢失后，将被GC（垃圾回收器）列入可回收“名单”中，但并不是马上就释放堆中内存

局部变量: 由声明在某方法，或某代码段里（比如for循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放

### 13. Java 集合系列问题

1 ArrayList、LinkedList、Vector的区别

ArrayList 和Vector底层是采用数组方式存储数据，Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差

LinkedList使用双向链表实现存储，随机存取比较慢

HashMap的底层源码实现：当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

Fail-Fast机制:在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast机制。这一机制在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map

2 HashMap和 HashTable 的区别

HashTable比较老，是基于Dictionary 类实现的，HashTable 则是基于 Map接口实现的 HashTable 是线程安全的， HashMap 则是线程不安全的 HashMap可以让你将空值作为一个表的条目的key或value

### 14.什么是反射机制，哪里能用到？

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

应用场景：

1 逆向代码 ，例如反编译

2 与注解相结合的框架 例如Retrofit

3 单纯的反射机制应用框架 例如EventBus 2.x

4 动态生成类框架 例如Gson

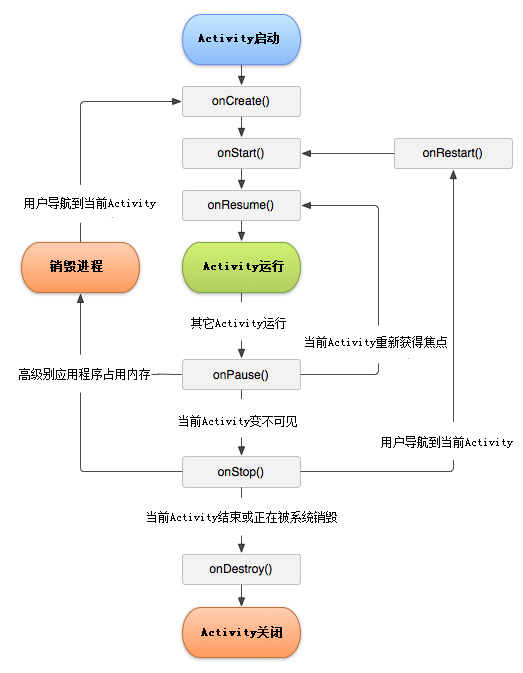
### 15.线程池及其作用

查看本文档中性能优化部分（[12页](#_1、降低执行时间)）

## Android部分

### 1. Activity 系列问题

1.1 绘制Activity生命周期流程图



1.2 介绍下不同场景下Activity生命周期的变化过程

启动Activity： onCreate()--->onStart()--->onResume()，Activity进入运行状态。

Activity退居后台： 当前Activity转到新的Activity界面或按Home键回到主屏： onPause()--->onStop()，进入停滞状态。

Activity返回前台： onRestart()--->onStart()--->onResume()，再次回到运行状态。

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()

锁定屏与解锁屏幕 只会调用onPause()，而不会调用onStop方法，开屏后则调用onResume()

1.3 内存不足时系统会杀掉后台的Activity，若需要进行一些临时状态的保存，在哪个方法进行？

Activity的 onSaveInstanceState() 和 onRestoreInstanceState()并不是生命周期方法，它们不同于 onCreate()、onPause()等生命周期方法，它们并不一定会被触发。当应用遇到意外情况（如：内存不足、用户直接按Home键）由系统销毁一个Activity，onSaveInstanceState() 会被调用。但是当用户主动去销毁一个Activity时，例如在应用中按返回键，onSaveInstanceState()就不会被调用。除非该activity是被用户主动销毁的，通常onSaveInstanceState()只适合用于保存一些临时性的状态，而onPause()适合用于数据的持久化保存。

1.4 onSaveInstanceState()被执行的场景有哪些：

系统不知道你按下HOME后要运行多少其他的程序，自然也不知道activity A是否会被销毁，因此系统都会调用onSaveInstanceState()，让用户有机会保存某些非永久性的数据。以下几种情况的分析都遵循该原则

1 当用户按下HOME键时

2 长按HOME键，选择运行其他的程序时

3 锁屏时

4 从activity A中启动一个新的activity时

6 屏幕方向切换时

1.5 介绍Activity的几中启动模式，并简单说说自己的理解或者使用场景

Activity的启动模式分为四种。（standard、singleTop、singTask、singleInstance）;

启动模式可在AndroidManifest.xml中，通过<activity>标签的android:launchMode属性设置。

一、standard模式

特点：1.Activity的默认启动模式

2.每启动一个Activity就会在栈顶创建一个新的实例。例如：闹钟程序

缺点：当Activity已经位于栈顶时，而再次启动Activity时还需要在创建一个新的实例，不能直接复用。

二、singleTop模式

特点：该模式会判断要启动的Activity实例是否位于栈顶，如果位于栈顶直接复用，否则创建新的实例。 例如：浏览器的书签

缺点：如果Activity并未处于栈顶位置，则可能还会创建多个实例。

三、singleTask模式

特点：使Activity在整个应用程序中只有一个实例。每次启动Activity时系统首先检查栈中是否存在当前Activity实例，如果存在，则直接复用，并把当前Activity之上所有实例全部出栈。例如：浏览器主界面

四、singleInstance模式

特点：该模式的Activity会启动一个新的任务栈来管理Activity实例，并且该实例在整个系统中只有一个。无论从那个任务栈中启动该Activity，都会是该Activity所在的任务栈转移到前台，从而使Activity显示。主要作用是为了在不同程序中共享一个Activity实例。

总结：Activity 的四种启动模式各有特色，在实际开发中，根据实际情况来选择合适的启动方式即可。

### 2. Service系列问题

2.1 注册Service需要注意什么

Service还是运行在主线程当中的，所以如果需要执行一些复杂的逻辑操作，最好在服务的内部手动创建子线程进行处理，否则会出现UI线程被阻塞的问题

2.2 Service与Activity怎么实现通信

方法一：

添加一个继承Binder的内部类，并添加相应的逻辑方法

重写Service的onBind方法，返回我们刚刚定义的那个内部类实例

Activity中创建一个ServiceConnection的匿名内部类，并且重写里面的onServiceConnected方法和onServiceDisconnected方法，这两个方法分别会在活动与服务成功绑定以及解除绑定的时候调用，在onServiceConnected方法中，我们可以得到一个刚才那个service的binder对象，通过对这个binder对象进行向下转型，得到我们那个自定义的Binder实例，有了这个实例，做可以调用这个实例里面的具体方法进行需要的操作了

方法二：

通过BroadCast(广播)的形式 当我们的进度发生变化的时候我们发送一条广播，然后在Activity的注册广播接收器，接收到广播之后更新视图

2.3 介绍源码中binder机制

2.4 IntentService与Service的区别

IntentService是Service的子类，是一个异步的，会自动停止的服务，很好解决了传统的Service中处理完耗时操作忘记停止并销毁Service的问题

会创建独立的worker线程来处理所有的Intent请求；

会创建独立的worker线程来处理onHandleIntent()方法实现的代码，无需处理多线程问题；

所有请求处理完成后，IntentService会自动停止，无需调用stopSelf()方法停止Service；

为Service的onBind()提供默认实现，返回null；

为Service的onStartCommand提供默认实现，将请求Intent添加到队列中；

IntentService不会阻塞UI线程，而普通Serveice会导致ANR异常

Intentservice若未执行完成上一次的任务，将不会新开一个线程，是等待之前的任务完成后，再执行新的任务，等任务完成后再次调用stopSelf()

2.5 *onStartCommand方法*

Int onStartCommand(Intent intent, int flags, int startId)方法。

其中参数flags默认情况下是0，对应的常量名为START\_STICKY\_COMPATIBILITY。startId是一个唯一的整型，用于表示此次Client执行startService(...)的请求请求标识，在多次startService(...)的情况下，呈现0,1,2....递增。另外，此函数具有一个int型的返回值，具体的可选值及含义如下：

START\_NOT\_STICKY：当Service因为内存不足而被系统kill后，接下来未来的某个时间内，即使系统内存足够可用，系统也不会尝试重新创建此Service。除非程序中Client明确再次调用startService(...)启动此Service。

START\_STICKY：当Service因为内存不足而被系统kill后，接下来未来的某个时间内，当系统内存足够可用的情况下，系统将会尝试重新创建此Service，一旦创建成功后将回调onStartCommand(...)方法，但其中的Intent将是null，pendingintent除外。

START\_REDELIVER\_INTENT：与START\_STICKY唯一不同的是，回调onStartCommand(...)方法时，其中的Intent将是非空，将是最后一次调用startService(...)中的intent。

START\_STICKY\_COMPATIBILITY：compatibility version of {@link #START\_STICKY} that does not guarantee that {@link #onStartCommand} will be called again after being killed。此值一般不会使用，所以注意前面三种情形就好。

以上的描述中，”当Service因为内存不足而被系统kill后“一定要非常注意，因为此函数的返回值设定只是针对此种情况才有意义的，换言之，当人为的kill掉Service进程，此函数返回值无论怎么设定，接下来未来的某个时间内，即使系统内存足够可用，Service也不会重启。

### 3. Handle系列问题

3.1 介绍Handle的机制

Handler通过调用sendmessage方法把消息放在消息队列MessageQueue中，Looper负责把消息从消息队列中取出来，重新再交给Handler进行处理，三者形成一个循环

通过构建一个消息队列，把所有的Message进行统一的管理，当Message不用了，并不作为垃圾回收，而是放入消息队列中，供下次handler创建消息时候使用，提高了消息对象的复用，减少系统垃圾回收的次数

每一个线程，都会单独对应的一个looper，这个looper通过ThreadLocal来创建，保证每个线程只创建一个looper，looper初始化后就会调用looper.loop创建一个MessageQueue，这个方法在UI线程初始化的时候就会完成，我们不需要手动创建

3.2 谈谈对HandlerThread的理解

HandlerThread能够新建拥有Looper的线程。这个Looper能够用来新建其他的Handler。（线程中的Looper）需要注意的是，新建的时候需要被回调。

1. **package** com.app;
3. **import** android.os.Bundle;
4. **import** android.os.Handler;
5. **import** android.os.HandlerThread;
6. **import** android.os.Message;
7. **import** android.support.v7.app.AppCompatActivity;
8. **import** android.util.Log;
10. **public** **class** MainActivity **extends** AppCompatActivity {
12. **private** HandlerThread myHandlerThread ;
13. **private** Handler handler ;
15. @Override
16. **protected** **void** onCreate(Bundle savedInstanceState) {
17. **super**.onCreate(savedInstanceState);
18. setContentView(R.layout.activity\_main);
20. //创建一个线程,线程名字：handler-thread
21. myHandlerThread = **new** HandlerThread( "handler-thread") ;
22. //开启一个线程
23. myHandlerThread.start();
24. //在这个线程中创建一个handler对象
25. handler = **new** Handler( myHandlerThread.getLooper() ){
26. @Override
27. **public** **void** handleMessage(Message msg) {
28. **super**.handleMessage(msg);
29. //这个方法是运行在 handler-thread 线程中的 ，可以执行耗时操作
30. Log.d( "handler " , "消息： " + msg.what + "  线程： " + Thread.currentThread().getName()  ) ;
32. }
33. };
35. //在主线程给handler发送消息
36. handler.sendEmptyMessage( 1 ) ;
38. **new** Thread(**new** Runnable() {
39. @Override
40. **public** **void** run() {
41. //在子线程给handler发送数据
42. handler.sendEmptyMessage( 2 ) ;
43. }
44. }).start() ;
46. }
48. @Override
49. **protected** **void** onDestroy() {
50. **super**.onDestroy();
52. //释放资源
53. myHandlerThread.quit() ;
54. }
55. }

### 4. ListView系列问题

4.1 ListView卡顿的原因与性能优化

A 重用converView： 通过复用converview来减少不必要的view的创建，另外Infalte操作会把xml文件实例化成相应的View实例，属于IO操作，是耗时操作。

B 减少findViewById()操作： 将xml文件中的元素封装成viewholder静态类，通过converview的setTag和getTag方法将view与相应的holder对象绑定在一起，避免不必要的findviewbyid操作

C 避免在 getView 方法中做耗时的操作: 例如加载本地 Image 需要载入内存以及解析 Bitmap ，都是比较耗时的操作，如果用户快速滑动listview，会因为getview逻辑过于复杂耗时而造成滑动卡顿现象。用户滑动时候不要加载图片，待滑动完成再加载，可以使用这个第三方库glide

D Item的布局层次结构尽量简单，避免布局太深或者不必要的重绘

E 尽量能保证 Adapter 的 hasStableIds() 返回 true 这样在 notifyDataSetChanged() 的时候，如果item内容并没有变化，ListView 将不会重新绘制这个 View，达到优化的目的

F 在一些场景中，ScollView内会包含多个ListView，可以把listview的高度写死固定下来。 由于ScollView在快速滑动过程中需要大量计算每一个listview的高度，阻塞了UI线程导致卡顿现象出现，如果我们每一个item的高度都是均匀的，可以通过计算把listview的高度确定下来，避免卡顿现象出现

G 使用 RecycleView 代替listview： 每个item内容的变动，listview都需要去调用notifyDataSetChanged来更新全部的item，太浪费性能了。RecycleView可以实现单个item的局部刷新，并且引入了增加和删除的动态效果，在性能上和定制上都有很大的改善

H ListView 中元素避免半透明： 半透明绘制需要大量乘法计算，在滑动时不停重绘会造成大量的计算，在比较差的机子上会比较卡。 在设计上能不半透明就不不半透明。实在要弄就把在滑动的时候把半透明设置成不透明，滑动完再重新设置成半透明。

I 尽量开启硬件加速： 硬件加速提升巨大，避免使用一些不支持的函数导致含泪关闭某个地方的硬件加速。当然这一条不只是对 ListView。

4.2 怎么实现一个部分更新的 ListView？

4.3 怎么实现ListView多种布局？

4.4 ListView与数据库绑定的实现

### 5. JNI系列问题

5.1 如何使用JNI

A JAVA中声明native 方法如private native String printJNI(String inputStr);

B 使用javah工具生成.h头文件这时候头文件中就会自动生成对应的函数JNIEXPORT jstring JNICALL Java\_com\_wenming\_HelloWorld\_printJNI

C 实现JNI原生函数源文件，新建HelloWorld.c文件，对刚才自动生成的函数进行具体的逻辑书写，例如返回一个java叫做HelloWorld的字符串等

D 编译生成动态链接so文件

E Java中调用Sysytem.load方法把刚才的so库加载进来，就可以调用native方法了

5.2 如何通过JNI传递对象

Java的String和C++的对象是不能对等起来的，所以当我们拿到.h文件下面的java对象，会做一次转换我们把java对象转换为C中的对象类型。

### 6. OOM系列问题

6.1 什么OOM？

OOM全称是Out Of Merrory，Android系统的每一个应用程序都设置一个硬性的Dalvik Heap Size最大限制阈值，如果申请的内存资源超过这个限制，系统就会抛出OOM错误

6.2 内存泄漏有哪些场景以及解决方法

A **类的静态变量持有大数据对象** 静态变量长期维持到大数据对象的引用，阻止垃圾回收。

B **非静态内部类存在静态实例** 非静态内部类会维持一个到外部类实例的引用，如果非静态内部类的实例是静态的，就会间接长期维持着外部类的引用，阻止被回收掉。

C **资源对象未关闭** 资源性对象比如（Cursor，File文件等）往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们， 以便它们的缓冲及时回收内存。它们的缓冲不仅存在于java虚拟机内，还存在于java虚拟机外。 如果我们仅仅是把它的引用设置为null,而不关闭它们，往往会造成内存泄露。 解决办法： 比如SQLiteCursor（在析构函数finalize（）,如果我们没有关闭它，它自己会调close()关闭）， 如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。 因此对于资源性对象在不使用的时候，应该调用它的close()函数，将其关闭掉，然后才置为null. 在我们的程序退出时一定要确保我们的资源性对象已经关闭。 程序中经常会进行查询数据库的操作，但是经常会有使用完毕Cursor后没有关闭的情况。如果我们的查询结果集比较小， 对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险，记得try catch后，在finally方法中关闭连接

D **Handler内存泄漏** Handler作为内部类存在于Activity中，但是Handler生命周期与Activity生命周期往往并不是相同的，比如当Handler对象有Message在排队，则无法释放，进而导致本该释放的Acitivity也没有办法进行回收。 解决办法：声明handler为static类，这样内部类就不再持有外部类的引用了，就不会阻塞Activity的释放。如果内部类实在需要用到外部类的对象，可在其内部声明一个弱引用引用外部类，例如：

1. **public** **class** MainActivity **extends** Activity {
2. **private** CustomHandler mHandler;
4. @Override
5. **protected** **void** onCreate(Bundle savedInstanceState) {
6. **super**.onCreate(savedInstanceState);
7. mHandler = **new** CustomHandler(**this**);
8. }
10. **static** **class** CustomHandlerextends Handler {
11. // 内部声明一个弱引用，引用外部类
12. **private** WeakReference<MainActivity > activityWeakReference;
13. **public** MyHandler(MyActivity activity) {
14. activityWeakReference= **new** WeakReference<MainActivity >(activity);
15. }
16. // ... ...
17. }
18. }

E **一些不良代码习惯** 有些代码并不造成内存泄露，但是他们的资源没有得到重用，频繁的申请内存和销毁内存，消耗CPU资源的同时，也引起内存抖动 解决方案 如果需要频繁的申请内存对象和和释放对象，可以考虑使用对象池来增加对象的复用。 例如ListView便是采用这种思想，通过复用converview来避免频繁的GC.

6.3 如何避免 OOM 问题的出现

1. 使用更加轻量的数据结构 例如，我们可以考虑使用ArrayMap/SparseArray而不是HashMap等传统数据结构。通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效，在于他们避免了对key与value的自动装箱（autoboxing），并且避免了装箱后的解箱。

2. 避免在Android里面使用Enum Android官方培训课程提到过“Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.”，具体原理请参考《Android性能优化典范（三）》，所以请避免在Android里面使用到枚举。

3. 减小Bitmap对象的内存占用 Bitmap是一个极容易消耗内存的大胖子，减小创建出来的Bitmap的内存占用可谓是重中之重，，通常来说有以下2个措施： inSampleSize：缩放比例，在把图片载入内存之前，我们需要先计算出一个合适的缩放比例，避免不必要的大图载入。 decode format：解码格式，选择ARGB\_6666/RBG\_545/ARGB\_4444/ALPHA\_6，存在很大差异

4.Bitmap对象的复用 缩小Bitmap的同时，也需要提高BitMap对象的复用率，避免频繁创建BitMap对象，复用的方法有以下2个措施 LRUCache : “最近最少使用算法”在Android中有极其普遍的应用。ListView与GridView等显示大量图片的控件里，就是使用LRU的机制来缓存处理好的Bitmap，把近期最少使用的数据从缓存中移除，保留使用最频繁的数据， inBitMap高级特性:利用inBitmap的高级特性提高Android系统在Bitmap分配与释放执行效率。使用inBitmap属性可以告知Bitmap解码器去尝试使用已经存在的内存区域，新解码的Bitmap会尝试去使用之前那张Bitmap在Heap中所占据的pixel data内存区域，而不是去问内存重新申请一块区域来存放Bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小

4. 使用更小的图片 在涉及给到资源图片时，我们需要特别留意这张图片是否存在可以压缩的空间，是否可以使用更小的图片。尽量使用更小的图片不仅可以减少内存的使用，还能避免出现大量的InflationException。假设有一张很大的图片被XML文件直接引用，很有可能在初始化视图时会因为内存不足而发生InflationException，这个问题的根本原因其实是发生了OOM。

5.StringBuilder 在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”。

4.避免在onDraw方法里面执行对象的创建 类似onDraw等频繁调用的方法，一定需要注意避免在这里做创建对象的操作，因为他会迅速增加内存的使用，而且很容易引起频繁的gc，甚至是内存抖动。

### 7. ANR 系列问题

7.1 什么ANR

ANR全称Application Not Responding，意思就是程序未响应。如果一个应用无法响应用户的输入，系统就会弹出一个ANR对话框，用户可以自行选择继续等待亦或者是停止当前程序。一旦出现下面两种情况，则弹出ANR对话框

出发条件：

应用在5秒内未响应用户的输入事件（如按键或者触摸）

BroadcastReceiver未在10秒内完成相关的处理

7.2 ANR是怎么引起的？

A 主线程中存在耗时的计算

B 主线程被IO操作（从4.0之后网络IO不允许在主线程中）阻塞。-

C 主线程中错误的操作，比如Thread.wait或者Thread.sleep等

7.3 如何避免ANR问题的出现

基本思路就是把一些耗时操作放到子线程中处理

A 使用AsyncTask处理耗时IO操作。

B 降低子线程优先级使用Thread或者HandlerThread时，调用Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND)设置优先级，否则仍然会降低程序响应，因为默认Thread的优先级和主线程相同。

C 使用Handler处理子线程结果，而不是使用Thread.wait()或者Thread.sleep()来阻塞主线程。

D Activity的onCreate和onResume回调中尽量避免耗时的代码

E BroadcastReceiver中onReceive代码也要尽量减少耗时操作建议使用IntentService处理。IntentService是一个异步的，会自动停止的服务，很好解决了传统的Service中处理完耗时操作忘记停止并销毁Service的问题

### 8. Asynctask问题

8.1 AsynTask为什么要设计为只能够一次任务？

最核心的还是线程安全问题，多个子线程同时运行，会产生状态不一致的问题。所以要务必保证只能够执行一次

8.2 AsynTask造成的内存泄露的问题怎么解决?

比如非静态内部类AsynTask会隐式地持有外部类的引用，如果其生命周期大于外部activity的生命周期，就会出现内存泄漏

注意要复写AsynTask的onCancel方法，把里面的socket，file等，该关掉的要及时关掉

在 Activity 的onDestory()方法中调用Asyntask.cancal方法

Asyntask内部使用弱引用的方式来持有Activity

8.3 若Activity已经销毁，此时AsynTask执行完并且返回结果，会报异常吗?

当一个App旋转时，整个Activity会被销毁和重建。当Activity重启时，AsyncTask中对该Activity的引用是无效的，因此onPostExecute()就不会起作用，若AsynTask正在执行，折会报 view not attached to window manager 异常

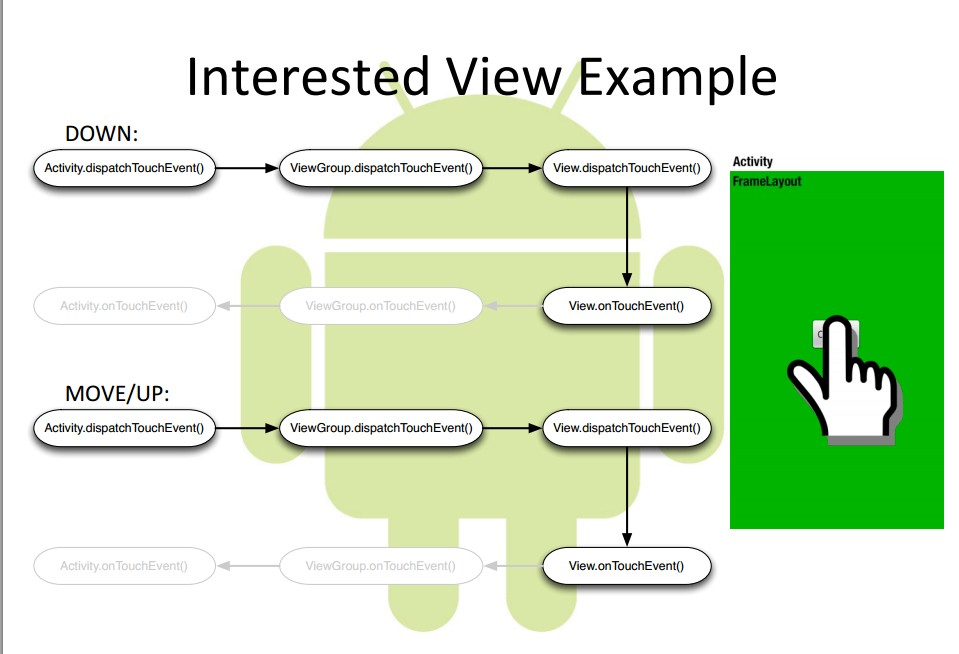
同样也是生命周期的问题，在 Activity 的onDestory()方法中调用Asyntask.cancal方法，让二者的生命周期同步

8.4 Activity销毁但Task如果没有销毁掉，当Activity重启时这个AsyncTask该如何解决？

还是屏幕旋转这个例子，在重建Activity的时候，会回掉Activity.onRetainNonConfigurationInstance()重新传递一个新的对象给AsyncTask，完成引用的更新

### 9. Android触摸分发机制

9.1 介绍触摸事件的分发机制



(1) 事件从Activity.dispatchTouchEvent()开始传递，只要没有被停止或拦截，从最上层的View(ViewGroup)开始一直往下(子View)传递。子View可以通过onTouchEvent()对事件进行处理。

(2) 事件由父View(ViewGroup)传递给子View，ViewGroup可以通过onInterceptTouchEvent()对事件做拦截，停止其往下传递。

(3) 如果事件从上往下传递过程中一直没有被停止，且最底层子View没有消费事件，事件会反向往上传递，这时父View(ViewGroup)可以进行消费，如果还是没有被消费的话，最后会到Activity的onTouchEvent()函数。

(4) 如果View没有对ACTION\_DOWN进行消费，之后的其他事件不会传递过来。

(5) OnTouchListener优先于onTouchEvent()对事件进行消费。

上面的消费即表示相应函数返回值为true。

9.2 View中 setOnTouchListener的onTouch，onTouchEvent，onClick的执行顺序

onTouch() 🡪 onTouchEvent() 🡪 onClick()

### 10. Dalvik虚拟机系列问题

10.1 什么是Dalvik虚拟机？

Dalvik虚拟机是Android平台的核心。它可以支持.dex格式的程序的运行，.dex格式是专为Dalvik设计的一种压缩格式，可以减少整体文件尺寸，提高I/O操作的速度，适合内存和处理器速度有限的系统。

10.2 Dalvik虚拟机的作用是什么？

Dalvik虚拟机主要是完成对象生命周期管理，内存回收，堆栈管理，线程管理，安全和异常管理等等重要功能。

10.3 Dalvik虚拟机与JVM有什么区别

Dalvik 基于寄存器，而 JVM 基于栈。基于寄存器的虚拟机对于更大的程序来说，在它们编译的时候，花费的时间更短。

Dalvik执行.dex格式的字节码，而JVM执行.class格式的字节码。

10.4 每个应用程序对应多少个Dalvik虚拟机

每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行 ，而所有的Android应用的线程都对应一个Linux线程

### 11. 注册广播接收器有哪几种方式,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，也就没办法再接收广播了

### 12. 显示Intent与隐式Intent的区别

对明确指出了目标组件名称的Intent，我们称之为“显式Intent”。 对于没有明确指出目标组件名称的Intent，则称之为“隐式 Intent”。

对于隐式意图，在定义Activity时，指定一个intent-filter，当一个隐式意图对象被一个意图过滤器进行匹配时，将有三个方面会被参考到：

A 动作(Action)

B 类别(Category)

C 数据(Data )

### 13. Android中的动画有哪些，区别是什么

逐帧动画(Drawable Animation)： 加载一系列Drawable资源来创建动画，简单来说就是播放一系列的图片来实现动画效果，可以自定义每张图片的持续时间

补间动画(Tween Animation)： Tween可以对View对象实现一系列简单的动画效果，比如位移，缩放，旋转，透明度等等。但是它并不会改变View属性的值，只是改变了View的绘制的位置，比如，一个按钮在动画过后，不在原来的位置，但是触发点击事件的仍然是原来的坐标。

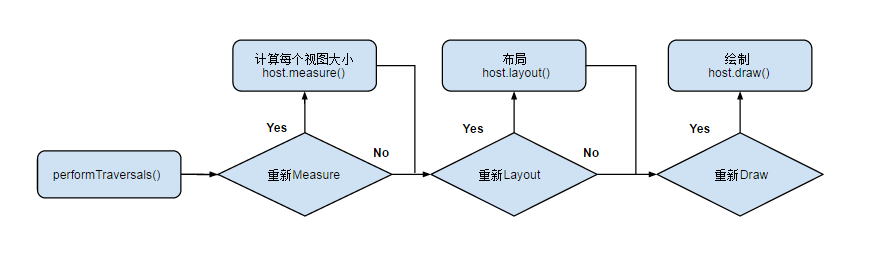
属性动画(Property Animation)： 动画的对象除了传统的View对象，还可以是Object对象，动画结束后，Object对象的属性值被实实在在的改变了

### 14.不使用动画，怎么实现一个动态的 View？

### 15. Postvalidata与Validata有什么区别？

### 16. 如何自定义ViewGroup？

### 17. View的绘制流程



measure()方法，layout()，draw()三个方法主要存放了一些标识符，来判断每个View是否需要再重新测量，布局或者绘制，主要的绘制过程还是在onMeasure，onLayout，onDraw这个三个方法中

1 onMesarue() 为整个View树计算实际的大小，即设置实际的高(对应属性:mMeasuredHeight)和宽(对应属性: mMeasureWidth)，每个View的控件的实际宽高都是由父视图和本身视图决定的。

2 onLayout() 为将整个根据子视图的大小以及布局参数将View树放到合适的位置上。

3. onDraw() 开始绘制图像，绘制的流程如下

a首先绘制该View的背景

b调用onDraw()方法绘制视图本身 (每个View都需要重载该方法，ViewGroup不需要实现该方法)

c如果该View是ViewGroup，调用dispatchDraw ()方法绘制子视图

d绘制滚动条

### 18. 数据持久化的四种方式有哪些？

文件存储： 通过java.io.FileInputStream和java.io.FileOutputStream这两个类来实现对文件的读写，java.io.File类则用来构造一个具体指向某个文件或者文件夹的对象。

SharedPreferences： SharedPreferences是一种轻量级的数据存储机制，他将一些简单的数据类型的数据，包括boolean类型，int类型，float类型，long类型以及String类型的数据，以键值对的形式存储在应用程序的私有Preferences目录（/data/data/<包名>/shared\_prefs/）中，这种Preferences机制广泛应用于存储应用程序中的配置信息。

SQLite数据库： 当应用程序需要处理的数据量比较大时，为了更加合理地存储、管理、查询数据，我们往往使用关系数据库来存储数据。Android系统的很多用户数据，如联系人信息，通话记录，短信息等，都是存储在SQLite数据库当中的，所以利用操作SQLite数据库的API可以同样方便的访问和修改这些数据。

ContentProvider: 主要用于在不同的应用程序之间实现数据共享的功能，不同于sharepreference和文件存储中的两种全局可读写操作模式，内容提供其可以选择只对哪一部分数据进行共享，从而保证我们程序中的隐私数据不会有泄漏的风险

### 19. fragement里面可以再嵌套fragment？

### 20. Socker编程的步骤

### 21. Activity中如何动态的添加Fragment

### 22. Scrollview怎么判断是否滑倒底部

### 23. 什么是 MVC 模式？MVC 模式的好处是什么？

### 24. 应用如何避免被第三方杀掉？

Service设置成START\_STICKY kill 后会被重启（等待5秒左右），重传Intent，保持与重启前一样

通过 startForeground将进程设置为前台进程， 做前台服务，优先级和前台应用一个级别​，除非在系统内存非常缺，否则此进程不会被 kill

双进程Service： 让2个进程互相保护\*\*，其中一个Service被清理后，另外没被清理的进程可以立即重启进程

QQ黑科技: 在应用退到后台后，另起一个只有 1 像素的页面停留在桌面上，让自己保持前台状态，保护自己不被后台清理工具杀死

在已经root的设备下，修改相应的权限文件,将App伪装成系统级的应用 Android4.0系列的一个漏洞，已经确认可行

用C编写守护进程(即子进程) : Android系统中当前进程(Process)fork出来的子进程，被系统认为是两个不同的进程。当父进程被杀死的时候，子进程仍然可以存活，并不受影响。鉴于目前提到的在Android->- Service层做双守护都会失败，我们可以fork出c进程，多进程守护。死循环在那检查是否还存在，具体的思路如下（Android5.0以上的版本不可行）

用C编写守护进程(即子进程)，守护进程做的事情就是循环检查目标进程是否存在，不存在则启动它。

在NDK环境中将1中编写的C代码编译打包成可执行文件(BUILD\_EXECUTABLE)。主进程启动时将守护进程放入私有目录下，赋予可执行权限，启动它即可。

联系厂商，加入白名单

### 25.Context与ApplicationContext的区别，分别用在什么情况下

Application的Context是一个全局静态变量，SDK的说明是只有当你引用这个context的生命周期超过了当前activity的生命周期，而和整个应用的生命周期挂钩时，才去使用这个application的context。

在android中context可以作很多操作，但是最主要的功能是加载和访问资源。在android中有两种context，一种是 application context，一种是activity context，通常我们在各种类和方法间传递的是activity context。

26. 同一个应用程序的不同Activity可以运行在不同的进程中么？如果可以，举例说明；

27. Java中的线程同步有哪几种方式，举例说明；

28. dp, dip, dpi, px, sp是什么意思以及他们的换算公式？layout-sw400dp, layout-h400dp分别代表什么意思；

29. 如何让两个TextView在一个RelativeLayout水平居中显示；

30. 如何画出一个印章的图案

31. 如何实现一个字体的描边与阴影效果

32. 设计一个从网络请求数据，图片，并加载到列表的系统，画出客户端架构并简单的分析下；

33. 设计一个文件的断点续传系统；

34. 设计一个图片缓存加载机制