1. activity的启动过程
2. 调用startActivity()方法 ，然后在方法中调用startActivityForResult()方法
3. 在该方法中获取Instrumentition对象，然后调用Instrumentation中的execStartActivity()的方法，真正执行activity的启动过程的是ActivityManagerNative.getDefault().startActivity(), 执行启动activity之后，有一个代码检查方法：checkStartActivityResult
4. ActivityManagerService继承自ActivityMangerNative,最终启动activity又走到了ActivityMangerService中
5. 经过多次调用activity的启动回到了ApplicationThread的scheduleLauchActivity方法
6. 在方法中又有sendMessage() ，当msg.what等于lauch\_activity后，最终执行perfromLauchActivity()方法
7. 在perfromLauchActivity方法中用instrumentition去new出activity，最后返回activity
8. service的启动过程
9. 启动状态，主要用于执行后台计算

Intent intent =new Intent(this,MyService.class)

startService(intent);

1. 令一种是绑定状态，用于和其它组件进行交互

Intent intent=new Intent(this,MyService.class);

bindService(intent,mServiceConn,BIND\_AUTO\_CREATE);

1. Service既可以处于启动状态也可以同时处于绑定状态
2. service的启动过程从ContextWrapper的startService()开始, 然后调用startServiceCommon方法
3. 在方法中用ActivityManagerNative.getDefault().startService()获取IActivityManger对象，ActivityManagerNative中的内部类ActivityMangerProxy继承至IActivityManger,所以service的启动又交给了它。
4. 在startService()中，使用mRemote.transact()客户端向服务端发起请求，所有请求就到了ActivityMangerNative类中的onTransact方法中
5. 又由于ActivityMangerService继承了ActivityManagerNavite这个抽象类，所有startService()方法在ActivitymangerService中实现。
6. ActivityManagerService并没有直接去处理，而是交给了ActiveService处理，在ActiveService中最终调用app.thread.scheduleCreateService();
7. App.thread是IApplicationThread对象，ApplicationThreadProxy实现了IApplicationThread接口，然后在方法中使用mRemote.transact();就进入到了ApplicationThreadNative中的onTrasact()中的scheduleCreateService()
8. scheduleCreateService是属于ApplicatonThread的,然后在方法中发送一个消息，去调用创建service方法，handleCreateService();
9. MVC ---既模型、视图、控制器3者之间的交互

View层：布局文件

Modle层：本地的数据文件或网络获取的数据体

Controller层：activity

1. MVP

View层：用户界面，如activity、fragment

Model层：本地的数据文件或网络获取的数据体

Presenter层：view和modle之间的桥梁，从modle层检索到数据后，以接口的形式返回给view层，实现view和modle的完全解耦。

1. SQLITE
2. 增

ContentValues values = new ContentValues();

values.put("name", "游天龙");

values.put("phone", "123");

db.insert("person", null, values);

1. 删

int i = db.delete("person", "name = ? ", new String[]{"张三"});

1. 改

ContentValues values = new ContentValues();

values.put("phone",0000);

int i = db.update("person", values, "name = ?", new String[]{"张三"});

1. 查

方法一.

Cursor cursor = db.query("person", null, null, null, null, null, null, null);

while(cursor.moveToNext()){

String name = cursor.getString(cursor.getColumnIndex("name"));

String phone = cursor.getString(cursor.getColumnIndex("phone"));

方法二.

Cursor cursor = db.rawQuery(String sql,String[] args);

1. AsnycTask

AsyncTask(param1,param2,param3)是一个抽象的泛型类，它提供了Params、Progress和Result三个泛型参数，Params表示参数的类型，Progress表示后台任务的执行任务的类型，而Result则表示后台任务的返回结果的类型，如果AsyncTask确实不需要传递具体的类型，那么这三个泛型参数可以用Void代替

1. onPreExecute(),在主线程执行，在异步任务执行之前调用，一般用于做一些准备工作
2. doInBackground(Params…params),在线程池中执行，params参数表示异步任务的输入参数。此方法需要返回计算结果给onPostExecute方法。
3. onProgressUpdate(Progress… values),在主线程中执行，当后台任务执行进度发送改变时此方法调用
4. onPostExecute(Result result),在主线程中执行，在异步任务执行之后，此方法会调用，其中result参数是doInBackgroud方法的返回值。

AsyncTask的工作原理

1. Asynctask调用execute()方法后，经过几次调用，最先开始调用AsnycTask的onPreExecute()方法，然后将asyncTask放入一个串行的线程池中，并且排队执行。然后再用另外一个线程池真正的去执行任务
2. 把AsyncTask的param封装成FutureTask对象，它充当了runnable的作用，FutureTask的run()方法最终会调用WorkerRunnable中的call（）方法，在这里执行AsnycTask的doInBackGround();
3. 接着将doInBackGroud()的结果传递给postResult()方法，在方法中通过mHandler发送一个消息，同时将参数传递给Handler
4. 在Handler的方法中会执行onPostExecute方法（）
5. 线程池的使用（有四种线程池，四种的不同无非在于设置不同的核心线程和最大线程数）
6. ThreadPoolExecutor的构成函数

Public ThreadPoolExecutor(int corePoolSize ,

Int maxnumPoolSize,

Long keepAliveTime,

TimeUnit unit,

BlockkingQueue<Runnable> workQueue,

ThreadFactory threadFactory)

corePoolSize:核心线程数，会在线程池中一直存活

maxnumPoolSize：线程池的最大线程数，当达到这个值后，后续线程会被阻塞

keepAliveTime：非核心线程书的超时 时长，超过这个时间会被回收

timeUnit: keepAliveTime参数的时间单位

threadFactory：线程工程，为线程池创建新的线程

1. threadPoolExecutor.execute(runnable)//启动线程池
2. 使用线程池的优点
3. 重用线程池中的线程，减少程序的性能开销
4. 有效控制线程池的最大并发数，避免大量线程池之间因相互抢夺资源而导致阻塞现象
5. TCP/IP协议
6. 它是用于网络通信的一套协议组合
7. 传统认为TCP/IP分为四层
8. 网络接口层：主要指物理层次，比如电缆
9. 网络层：物理地址与逻辑地址的转换(IP/ICMP)
10. 传输层：为网络提供流量监控，错误控制和确认服务(TCP/UDP)

TCP：是一种面向连接的、可靠的、基于字节流的[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82/4329536)通信协议，通过三次握手实现客户端与服务端连接，四次挥手断开连接

UDP:是一种无连接，传输最大限度64k的数据报

1. 应用层：为网络排错，文件传输，远程控制（HTTP）
2. Http和HttpS的区别
3. 两者都是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
4. https使用密钥和签字的证书响应连接
5. http作用在应用层，https作用在传输层
6. http和https的端口不一样，前者是80，后者是443
7. get和post的区别
8. get参数通过url传递，post放在request body中
9. get请求在url中传递的参数是有长度限制的，而post没有

十一. Socket通信

1.是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口（API）

十二. 启动优化

1. 将启动页的主题里的windowBackGroud属性设置成app启动图
2. Application之做初始化操作，不要在APPlication进行耗时操作，减少LaunchActivity的View层级

十三. 布局优化

1.<include>标签可以将一个指定的布局文件加载到当前的布局文件中

2.<merge>标签一般和<include>标签一起使用从而减少布局的层级

十四. 内存优化

1.减小对象的内存占用

1）使用更加轻量的数据结构（ArrayMap替换HashMap）

2）避免在android中使用枚举

3）减小bitmap对象的内存占用（使用前先压缩图片）

4）使用更小的图片

2.内存对象重复利用

1）复用系统自带的资源（字符串/颜色/图片/动画/布局）

2）ListView/GradView的ConvertView复用、使用单例减少对象的重复创建

If(conVertView==null){

Holder=new ViewHolder();

}else{

Holder=(ViewHolder)convertView.getTag();

}

3）StringBuilder:如果使用Stgring进行字符串多次拼接，会产生多个对象，而StringBuilder是对一个对象多次操作

3.避免对象的内存泄漏

1. 单例模式导致的内存泄漏
2. 静态常量导致的内存泄漏
3. 非静态内部类创建静态常量实例导致的内存溢出
4. Handler导致的内存溢出（创建非静态匿名内部handler,handler长期持有activity的引用，handler又不断的进行消息的轮询，导致activity不能销毁）
5. 线程造成的内存泄漏（在activity中创建的匿名内部线程，在线程还没有执行完毕时，关闭activity,匿名内部类又持有着activity的引用）
6. webview内存泄漏
7. 资源未关闭导致内存泄漏