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();