目录

1、 什么是 Activity ？

一个用户交互界面对应一个 activity，activity 是 Context 的子类，同时实现了 window.callback 和 keyevent.callback， 可以处理与窗体用户交互的事件。

开发常用的有 FragmentActivitiyListActivity ，PreferenceActivity ，TabAcitivty 等

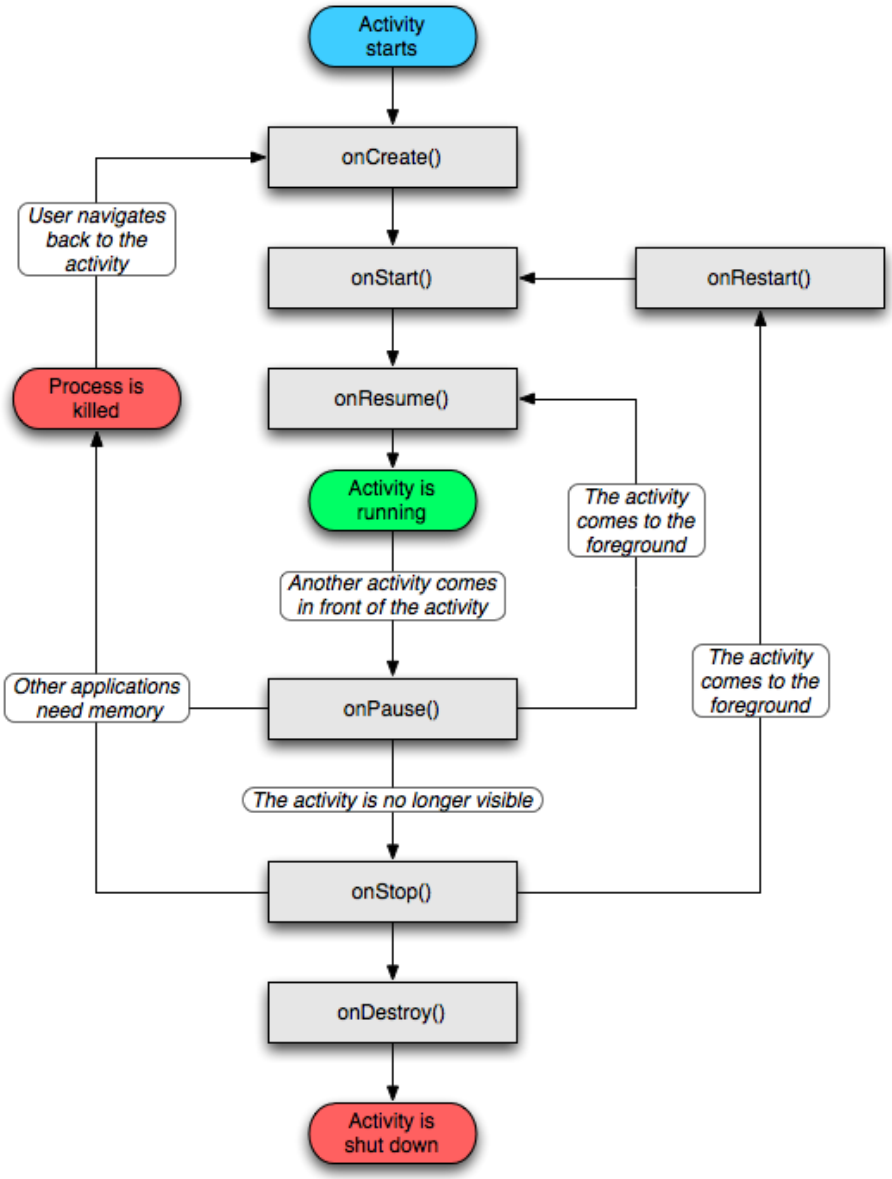
2、 Activity 生命周期 ？

Activity 从创建到销毁有多种状态，从一种状态到另一种状态时会激发相应的回调方法，这些回调方法包括：onCreate、onStart、onResume、onPause、onStop、onDestroy。

这些方法都是两两对应的，onCreate 创建与 onDestroy 销毁; onStart 可见与 onStop 不可见；onResume 可编辑(即焦点)与 onPause；如果界面有共同的特点或者功能的时候，还会自己定义一个 BaseActivity。进度对话框的显示与销毁 。

如果 IntentActivity 处于任务栈的顶端，也就是说之前打开过的 Activity，现在处于 onPause、onStop 状态的话，其他应用再发送 Intent 的话，执行顺序为： onNewIntent，onRestart，onStart，onResume。

提醒： *从 Activity A，打开另一个 Activity B，B 这个 Activity 是在 A 的 onPause 执行后才变成可见状态的，所以为了不影响 B 的显示，最好不要在 onPause 里执行一些耗时操作，可以考虑将这些操作放到 onStop 里，这时 B 已经可见了，（意思是 A ->onPause 后才 B->onCreate，onStart， onResume #转问题 4）*



3、 如何保存 Activity 的状态 ？

Activity 的状态通常情况下系统会自动保存的，只有当我们需要保存额外的数据时才需要使用到这样的功能。

一般来说，调用 onPause() 和 onStop() 方法后的 activity 实例仍然存在于内存中，activity 的所有信息和状态数据不会消失，当 activity 重新回到前台之后，所有的改变都会得到保留。

但是当系统内存不足时，调用 onPause() 和 onStop() 方法后的 activity 可能会被系统摧毁，此时内存中就不会存有该 activity 的实例对象了。如果之后这个 activity 重新回到前台，之前所作的改变就会消失。为了避免此种情况的发生，我们可以覆写 onSaveInstanceState() 方法。

onSaveInstanceState() 方法接受一个 Bundle 类型的参数，开发者可以将状态数据存储到这个 Bundle 对象中，这样即使 activity 被系统摧毁，当用户重新启动这个 activity 而调用它的 onCreate() 方法时，上述的 Bundle 对象会作为实参传递给 onCreate() 方法，开发者可以从 Bundle 对象中取出保存的数据，后利用这些数据将 activity 恢复到被摧毁之前的状态。

需要注意的是 onSaveInstanceState() 方法并不是一定会被调用的，因为有些场景是不需要保存状态数据的。比如用户按下 BACK 键退出 activity 时，用户 显然想要关闭这个 activity，此时是没有必要保存数据以供下次恢复的，也就是 onSaveInstanceState() 方法不会被调用。

如果调用 onSaveInstanceState() 方法，调用将发生在 onPause() 或 onStop() 方法之前。

4、 两个 Activity 之间跳转时必然会执行的是哪几个方法 ？

一般情况下比如说有两个 activity，分别叫 A、B，当在 A 里面激活 B 组件的时候，A 会调用 onPause() 方法，然后 B 调用 onCreate()、onStart()、onResume()。

这个时候 B 覆盖了窗体，A 会调用 onStop() 方法，如果 B 是个透明的，或者是对话框的样式，就不会调用 A 的 onStop() 方法。

5、 横竖屏切换时 Activity 的生命周期 ？

此时的生命周期跟清单文件里的配置有关系。

1) 不设置

Activity 的 android：configChanges 时，切屏会重新调用各个生命周期默认首先销毁当前 activity，然后重新加载。

2) 设置

android：configChanges="orientation|keyboardHidden|screenSize"时，切屏不会重新调用各个生命周期，只会执行 onConfigurationChanged 方法。

通常在游戏开发， 屏幕的朝向都是写死的。

6、 如何将一个 Activity 设置成窗口的样式 ？

只需要给 Activity 配置如下属性即可。

```
1    Android:theme="@android:style/Theme.Dialog"
```

7、 如何退出 Activity ？如何安全退出已调用多个 Activity 的 Application ？

1) 通常情况用户退出一个 Activity 只需按返回键，我们写代码想退出 activity 直接调用 finish() 方法就行。
2) 发送特定广播：
   在需要结束应用时，发送一个特定的广播，每个 Activity 收到广播后，关闭即可。
3) 递归退出
   在打开新的 Activity 时使用 startActivityForResult，然后自己加标志，在 onActivityResult 中处理，递归关闭。
4) FLAG_ACTIVITY_CLEAR_TOP
   其实也可以通过 intent 的 flag 来实现 intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP) 激活一个新的 activity。 此时如果该任务栈中已经有该 Activity，那么系统会把这个 Activity 上面的所有 Activity 干掉。其实相当于给 Activity 配置的启动模式为 SingleTop。
5) 记录打开的 Activity：
   每打开一个 Activity，就记录下来。在需要退出时，关闭每一个 Activity。

8、 Activity 的四种启动模式，singletop 和 singletask 区别是什么 ？一般书签的使用模式是 singletop，那为什么不使用 singletask ？

- **standard 模式**

这是默认模式，每次激活 Activity 时都会创建 Activity 实例，并放入任务栈中。

- **singleTop 模式**

如果在任务的栈顶正好存在该 Activity 的实例，就重用该实例(会调用实例的 onNewIntent())，否则就会创建新的实例并放入栈顶，即使栈中已经存在该 Activity 的实例，只要不在栈顶，都会创建新的实例。

示例用例：*搜索功能。为了更好的用户体验，通常把一个搜索框做成无压回的情况。*

- **singleTask 模式**

如果在栈中已经有该 Activity 的实例，就重用该实例(会调用实例的 onNewIntent())。重用时，会让该实例回到栈顶，因此在它上面的实例将会被移出栈。如果栈中不存在该实例，将会创建新的实例放入栈中。

示例用例：*任何一个入口点活动例如电子邮件客户端的收件箱页面或社交网络的时间轴。主页面的设计一般使用 SingleTask 模式来设计，因为用户点击多次页面的相互跳转后，在点击回到主页，再次点击退出，这时他的实际需求就是要退出程序，而不是一次一次关闭刚才跳转过的页面，最后才退出。*

- **singleInstance 模式**

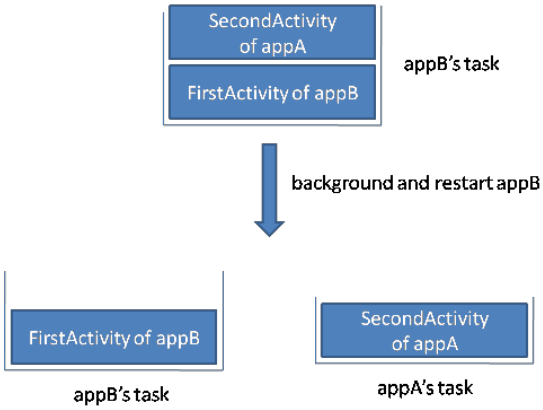
在一个新栈中创建该 Activity 的实例，并让多个应用共享该栈中的该 Activity 实例。一旦该模式的 Activity 实例已经存在于某个栈中，任何应用再激活该 Activity 时都会重用该栈中的实例(会调用实例的 onNewIntent())。其效果相当于多个应用共享一个应用，不管谁激活该 Activity 都会进入同一个应用中。

9、 Activity 之 Task 相关的属性

- **android:allowTaskReparenting**

这个属性用来标记一个 Activity 实例在当前应用退居后台后，是否能从启动它的那个 task 移动到有共同 affinity 的 task，“true”表示可以移动，“false”表示它必须呆在当前应用的 task 中，默认值为 false。如果一个这个 Activity 的<activity>元素没有设定此属性，设定在<application>上的此属性会对此 Activity 起作用。

例：在一个应用中要查看一个 web 页面，在启动系统浏览器 Activity 后，这个 Activity 实例和当前应用处于同一个 task，当我们的应用退居后台之后用户再次从主选单中启动应用，此时这个 Activity 实例将会重新宿主到 Browser 应用的 task 内，在我们的应用中将不会再看到这个 Activity 实例，而如果此时启动 Browser 应用，就会发现，第一个界面就是我们刚才打开的 web 页面，证明了这个 Activity 实例确实是宿主到了 Browser 应用的 task 内。



- **android:alwaysRetainTaskState**

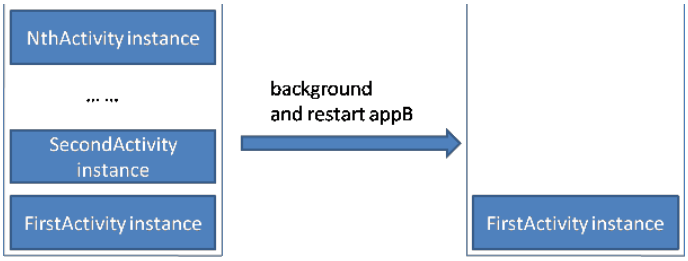
这个属性用来标记应用的 task 是否保持原来的状态，“true”表示总是保持，“false”表示不能够保证，默认为“false”。此属性只对 task 的根 Activity 起作用，其他的 Activity 都会被忽略。

例：默认情况下，如果一个应用在后台呆的太久例如 30 分钟，用户从主选单再次选择该应用时，系统就会对该应用的 task 进行清理，除了根 Activity，其他 Activity 都会被清除出栈，但是如果在根 Activity 中设置了此属性之后，用户再次启动应用时，仍然可以看到上一次操作的界面。Browser 应用程序，有很多状态，比如打开很多的 tab，用户不想丢失这些状态。

- **android:clearTaskOnLaunch**

这个属性用来标记是否从 task 清除除根 Activity 之外的所有的 Activity，“true”表示清除，“false”表示不清除，默认为“false”。同样，这个属性也只对根 Activity 起作用，其他的 Activity 都会被忽略。

例：如果设置了这个属性为 “true”，每次用户重新启动这个应用时，都只会看到根 Activity，task 中的其他 Activity 都会被清除出栈。如果我们的应用中引用到了其他应用的 Activity，这些 Activity 设置了 allowTaskReparenting 属性为“true”，则它们会被重新宿主到有共同 affinity 的 task 中。



- **android:finishOnTaskLaunch**

这个属性和 android:allowTaskReparenting 属性相似，不同之处在于 allowTaskReparenting 属性是重新宿主到有共同 affinity 的 task 中，而 finishOnTaskLaunch 属性是销毁实例。如果这个属性和 android:allowReparenting 都设定为“true”，则这个属性胜出。

- **android:noHistory**

具有此属性标识的 Activity 当导航到其他 Activity 上时，Activity 栈将不记录其自身，简单来说，当 A -> B, 此时 Activity 栈中只含有 B。

10、 Android 中的 Context，Activity，Appliction 有什么区别 ？

- 相同：Activity 和 Application 都是 Context 的子类。

Context 从字面上理解就是上下文的意思,在实际应用中它也确实是起到了管理上下文环境中各个参数和变量的总用，方便我们可以简单的访问到各种资源。

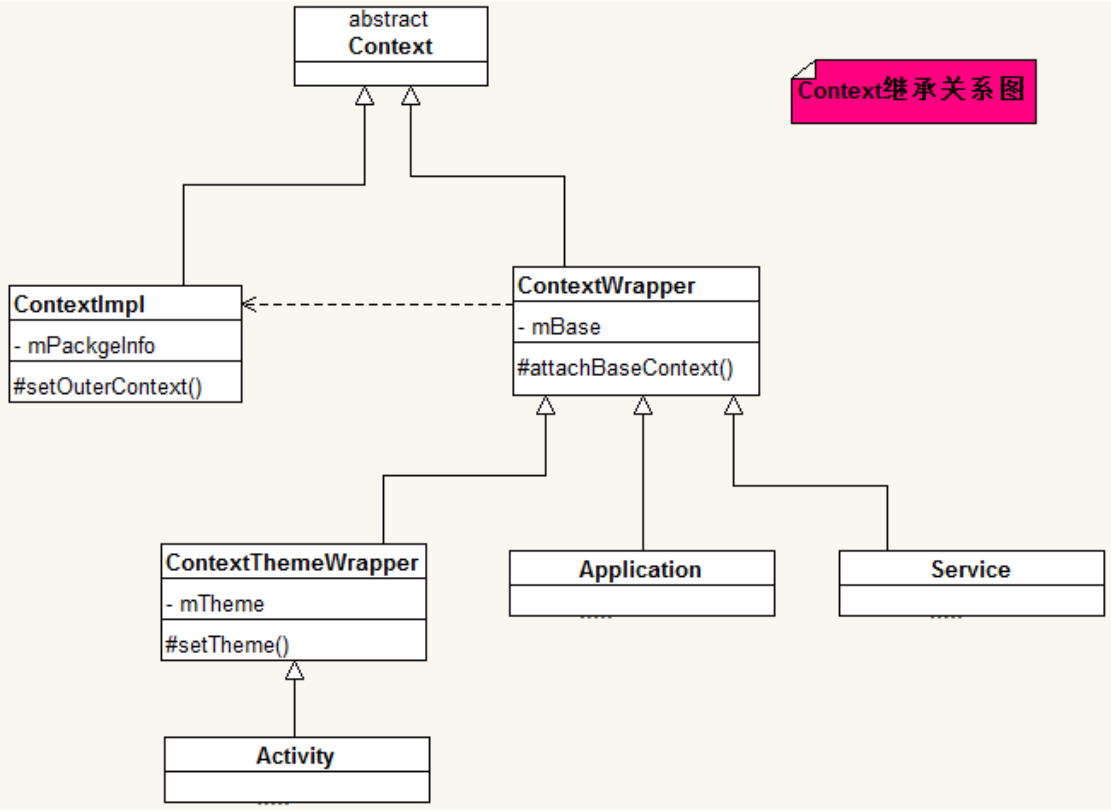
- 不同：维护的生命周期不同。

Context 维护的是当前的 Activity 的生命周期，Application 维护的是整个项目的生命周期。

11、 Context 是什么 ？

它描述的是一个应用程序环境的信息，即上下文，其提供了关于应用环境全局信息的接口。**通过它可以获取应用程序的资源和类，（包括一些应用级别操作，例如：启动一个 Activity，发送广播，接受 Intent）。**

该类是一个抽象(abstract class)类，有两个具体的实现子类：ContextImpl 和 ContextWrapper。ContextWrapper 构造函数中必须包含一个真正的 Context 引用，同时 ContextWrapper 中提供了 attachBaseContext() 用于给 ContextWrapper 对象中指定真正的 Context 对象，调用 ContextWrapper 的方法都会被转向其所包含的真正的 Context 对象。ContextImpl 是 Context 的具体实现类，ContextWrapper 是 Context 的包装类。Activity，Application，Service 虽都继承自 ContextWrapper，但它们初始化的过程中都会创建 ContextImpl 对象，由 ContextImpl 实现 Context 中的方法。



一个应用程序有几个 Context

Context 数量=Activity 数量+Service 数量+1。Broadcast Receiver，Content Provider 并不是 Context 的子类，他们所持有的 Context 都是其他地方传过去的，所以并不计入 Context 总数。

Context 的功能

弹出 Toast、启动 Activity、启动 Service、发送广播、操作数据库等等。

```
1    TextView tv = new TextView(getContext());
2    ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(), ...);
```

```
3    AudioManager am = (AudioManager) getContext().
4                          getSystemService(Context.AUDIO_SERVICE);
5    getApplicationContext().getSharedPreferences(name, mode);
6    getApplicationContext().getContentResolver().query(uri, ...);
7    getContext().getResources().getDisplayMetrics().widthPixels * 5 / 8;
8    getContext().startActivity(intent);
9    getContext().startService(intent);
10   getContext().sendBroadcast(intent);
```

如何获取 Context

1) View.getContext,返回当前 View 对象的 Context 对象，通常是当前正在展示的 Activity 对象。
2) Activity.getApplicationContext,获取当前 Activity 所在的(应用)进程的 Context 对象，通常我们使用 Context 对象时，要优先考虑这个全局的进程 Context。
3) ContextWrapper.getBaseContext():用来获取一个 ContextWrapper 进行装饰之前的 Context，可以使用这个方法，这个方法在实际开发中使用并不多，也不建议使用。
4) Activity.this 返回当前的 Activity 实例，如果是 UI 控件需要使用 Activity 作为 Context 对象，但是默认的 Toast 实际上使用 ApplicationContext 也可以。

getApplication() 和 getApplicationContext()

它们是同一个对象，getApplicationContext() 得到的结果就是 Application 本身的实例。getApplication() 方法的语义性非常强，一看就知道是用来获取 Application 实例的，但是这个方法只有在 Activity 和 Service 中才能调用的到。如果在一些其它的场景，比如 BroadcastReceiver 中也想获得 Application 的实例，这时需要借助 getApplicationContext() 方法。

12、 Task & Back Tack

由同一个应用启动的 Activity 默认都在同一个任务栈中（Task）。

13、 appA 调用 appB 的 Activity，Task 的情况 ？

- **默认情况：**
  appB 的 Activity 好像是嵌入到了 appA 的 Task 中，但是不影响 appB 的正常运行，appB 有自己的 Task。
- **FLAG_NEW_TASK：**
  appB 的 Activity 不嵌入到 appA 的 Task 中，而是加入到 appB 自己的 Task。
- **FLAG_ACTIVITY_CLEAR_TOP：**
  当 Intent 对象包含这个标记时，如果在栈中发现存在 Activity 实例，则清空这个实例之上的 Activity，使其处于栈顶。
- **FLAG_ACTIVITY_SINGLE_TOP：**
  在使用默认的“standard”启动模式下，如果没有在 Intent 使用到 FLAG_ACTIVITY_SINGLE_TOP 标记，那么它将关闭后重建，如果使用了这个 FLAG_ACTIVITY_SINGLE_TOP 标记，则会使用已存在的实例。

14、 如何获取当前 Activity 实例对象 ？

ActivityLifecycleCallbacks

Activity 生命周期回调，Application 通过此接口提供了一套回调方法，用于让开发者对 Activity 的生命周期事件进行集中处理。
重写 Application 的 onCreate()方法，或在 Application 的无参构造函数内，调用 Application.registerActivityLifecycleCallbacks() 方法，并实现 ActivityLifecycleCallbacks 接口。
但是这个要求 API 14+ （Android 4.0+）以上使用，幸好我们这个最低支持，满足需求。

15、 启动 Activity 的方法 ？

- Intent 显示和隐式启动
- 使用 adb shell am 命令

16、 TaskAffinity 是什么 ？

标识 Activity 任务栈名称的属性：TaskAffinity，默认为应用包名。

17、 如果新 Activity 是透明主题时，旧 Activity 会不会走 onStop

不会。

18、 Service 与 Activity 之间通信的几种方式 ？

1) **binder 持有 Activity 的 Handler 对象，或者持有一个 Activity 的回调接口 callbackInterface.**

```
public class WeakReferenceHandlerActivity extends Activity {
    private MyHandler mHandler;
    public static int FLAG = 10;
    public ProgressBar progressBar;
    private Button bindButton;
    private UseHandlerService.MyBinder myBinder;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.handler_layout);
        progressBar = (ProgressBar)this.findViewById(R.id.progress);
        bindButton = (Button)this.findViewById(R.id.bind);
        nextButton = (Button)this.findViewById(R.id.next);
        mHandler = new MyHandler(this);

        bindButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                bindService();
                Log.d("","startDownLoad() --> click");
            }
        });
    }

    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceDisconnected(ComponentName name) {
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            Log.d("","startDownLoad() --> onbind");
            myBinder = (UseHandlerService.MyBinder) service;
            myBinder.setCallBackHandler(mHandler);
            myBinder.startDownLoad();
        }
    };

    @Override
    protected void onPause() {
        unBindService();
        super.onPause();
    }

    public void bindService(){
        Intent bindIntent = new Intent(this, UseHandlerService.class);
        bindService(bindIntent, connection, BIND_AUTO_CREATE);
    }

    public void unBindService(){
        unbindService(connection);
    }

    @Override
    protected void onDestroy() {
        // Remove all Runnable and Message.
        mHandler.removeCallbacksAndMessages(null);
        super.onDestroy();
    }

    public MyHandler getHandler(){
```

```
        return mHandler;
    }

    public class MyHandler extends Handler {
        // WeakReference to the outer class's instance.
        private WeakReference<weakreferencehandleractivity> mOuter;

        public MyHandler(WeakReferenceHandlerActivity activity) {
            mOuter = new WeakReference<weakreferencehandleractivity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            WeakReferenceHandlerActivity outer = mOuter.get();
            if (outer != null) {
                // Do something with outer as your wish.
                if(!outer.isFinishing()){
                    int progress = msg.what;
                    progressBar.setProgress(progress);
                }
            }
        }
    }
}
```

```
public class UseHandlerService extends Service{
    /** 进度条的最大值 */
    public static final int MAX_PROGRESS = 100;
    /** 进度条的进度值 */
    private int progress = 0;
    /** 更新进度的回调接口 */
    private MyHandler myHandler;
    private MyBinder mBinder = new MyBinder();
    private String TAG = "UseHandlerService";
    /**
     * 注册回调接口的方法，供外部调用
     * @param onProgressListener
     */
    public void setHandler(MyHandler handler) {
        myHandler = handler;
    }
    /**
     * 增加 get()方法，供 Activity 调用
     * @return 下载进度
     */
    public int getProgress() {
        return progress;
    }
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "onCreate() executed");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d(TAG, "onStartCommand() executed");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy() executed");
```

```
    }

    /**
     * 返回一个 Binder 对象
     */
    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    public class MyBinder extends Binder{
        /**
         * 获取当前 Service 的实例
         * @return
         */
        public UseHandlerService getService(){
            return UseHandlerService.this;
        }

        public void setCallBackHandler(MyHandler handler){
            myHandler = handler;
        }

        public void startDownLoad(){
            Log.d("","startDownLoad() inBinder-->");
            new Thread(new Runnable() {
                public void run() {
                    while(progress < MAX_PROGRESS){
                        progress += 5;
                        Log.d("","startDownLoad() run-->");
                        //进度发生变化通知调用方
                        if(myHandler != null){
                            myHandler.sendEmptyMessage(progress);
                        }
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }).start();
        }
    }
}
```

2) **AIDL 实现，完美支持 IPC**

3) **startService 和 broadcastReceiver 共同实现。**

19、 Fragment 异常 bug 攻略

1) getActivity() 空指针

情况:

在调用了 getActivity()时，当前的 Fragment 已经 onDetach()了宿主 Activity。比如：在 pop 了 Fragment 之后，该 Fragment 的异步任务仍然在执行，并且在执行完成后调用了 getActivity()方法，这样就会空指针。

解决办法:

- 更"安全"的方法：对于 Fragment 已经 onDetach 这种情况，我们应该避免在这之后再去调用宿主 Activity 对象，比如取消这些异步任务.
- 在 Fragment 基类里设置一个 Activity mActivity 的全局变量，在 onAttach(Activity activity)里赋值，使用 mActivity 代替 getActivity()，保证 Fragment 即使在 onDetach 后，仍持有 Activity 的引用（有引起内存泄露的风险，但是异步任务没停止的情况下，本身就可能已内存泄漏，相比 Crash，这种做法"安全"些），即:

```
protected Activity mActivity;
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    this.mActivity = activity;
}

/**  support 23 的库，上面的方法会提示过时，可以用下面的方法代替 */
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    this.mActivity = (Activity)context;
}
```

2) 异常：Can not perform this action after onSaveInstanceState

情况：

在离开当前 Activity 等情况下，系统会调用 onSaveInstanceState() 保存当前 Activity 的状态、数据等，直到再回到该 Activity 之前（onResume()之前），如果执行了 Fragment 事务，就会抛出该异常！(一般是其他 Activity 的回调让当前页面执行事务的情况，会引发该问题)。

解决办法：

- 该事务使用 commitAllowingStateLoss()方法提交，但是有可能导致该次提交无效！(宿主 Activity 被强杀时)
- 利用 onActivityForResult()/onNewIntent()，可以做到事务的完整性，不会丢失事务：

```
// ReceiverActivity 或 其子 Fragment:
void start(){
   startActivityForResult(new Intent(this, SenderActivity.class), 100);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
     super.onActivityResult(requestCode, resultCode, data);
     if (requestCode == 100 && resultCode == 100) {
         // 执行 Fragment 事务
     }
 }

// SenderActivity 或 其子 Fragment:
void do() { // 操作 ReceiverActivity（或其子 Fragment）执行事务
    setResult(100);
    finish();
}
```

3) 未必靠谱的出栈方法 remove()

情况：

在 add 的同时将 Fragment 加入回退栈：addToBackStack(name)的情况下，它并不能真正将 Fragment 从栈内移除，如果你在 2 秒后（确保 Fragment 事务已经完成）打印 getSupportFragmentManager().getFragments()，会发现该 Fragment 依然存在，并且依然可以返回到被 remove 的 Fragment，而且是空白页面。

解决办法：

- 如果没有将 Fragment 加入回退栈，remove 方法可以正常出栈。
- 如果加入了回退栈，popBackStack() 系列方法才能真正出栈

4) Fragment 转场动画（v4 包）

如果想让出栈动画运作正常的话，需要使用 Fragment 的 onCreateAnimation 中控制动画。

```
@Override
public Animation onCreateAnimation(int transit, boolean enter, int nextAnim) {
    // 此处设置动画
}
```

如果想从当前 Fragment 进入一个新的 Fragment，并且同时要关闭当前 Fragment。由于数据结构是栈，所以正确做法是先 pop，

再 add，但是转场动画会有覆盖的不正常现象，你需要特殊处理，不然会闪屏！

如果遇到 Fragment 的 mNextAnim 空指针的异常（通常是在 Fragment 被重启的情况下），首先需要检查是否操作的 Fragment 是否为 null；其次是否在 Fragment 转场动画还没结束时，就执行了其他事务等方法；解决思路就是延迟一个动画时间再执行事务，或者临时将该 Fragment 设为无动画。

5) 使用 FragmentPagerAdapter+ViewPager 的注意事项

- 使用 FragmentPagerAdapter+ViewPager 时，切换回上一个 Fragment 页面时（已经初始化完毕），不会回调任何生命周期方法以及 onHiddenChanged()，只有 setUserVisibleHint(boolean isVisibleToUser)会被回调，所以如果想进行一些懒加载，需要在这里处理。
- 在给 ViewPager 绑定 FragmentPagerAdapter 时，new FragmentPagerAdapter(fragmentManager) 的 FragmentManager，一定要保证正确，如果 ViewPager 是 Activity 内的控件，则传递 getSupportFragmentManager()，如果是 Fragment 的控件中，则应该传递 getChildFragmentManager()。只要记住 ViewPager 内的 Fragments 是当前组件的子 Fragment 这个原则即可。

20、 Fragment 使用建议（第三方库 Fragmentation）

1) 对 Fragment 传递数据，建议使用 setArguments(Bundle args)，而后在 onCreate 中使用 getArguments()取出，在 “内存重启”前，系统会保存数据，不会造成数据的丢失。和 Activity 的 Intent 恢复机制类似。
2) 使用 newInstance(参数) 创建 Fragment 对象，优点是调用者只需要关系传递的哪些数据，而无需关心传递数据的 Key 是什么。
3) 如果需要在 Fragment 中用到宿主 Activity 对象，建议在基类 Fragment 定义一个 Activity 的全局变量，在 onAttach 中初始化，这不是最好的解决办法，但这可以有效避免一些意外 Crash。
4) show() 和 hide() 最终是让 Fragment 的 View setVisibility(true 还是 false)，不会调用生命周期；replace() 会销毁视图，即调用 onDestoryView、onCreateView 等一系列生命周期；add() 和 replace() 不要在同一个阶级的 FragmentManager 里混搭使用。如果有很高的概率会再次使用当前的 Fragment，建议使用 show() 和 hide()；如果有大量图片，这时更好的方式可能是 replace，配合图片框架在 Fragment 视图销毁时，回收其图片所占的内存。
5) 当使用 add() + show()，hide() 跳转新的 Fragment 时，旧的 Fragment 回调 onHiddenChanged()，不会回调 onStop() 等生命周期方法，而新的 Fragment 在创建时是不会回调 onHiddenChanged()。
6) 对于宿主 Activity，getSupportFragmentManager()获取的 FragmentActivity 的 FragmentManager 对象；
对于 Fragment，getFragmentManager() 是获取的是父 Fragment（如果没有，则是 FragmentActivity）的 FragmentManager 对象；而 getChildFragmentManager() 是获取自己的 FragmentManager 对象。

21、 Activity 与 Fragment 通信方案

1) handler 方案：

```
1   public class MainActivity extends FragmentActivity{
2       //申明一个 Handler
3       public Handler mHandler = new Handler(){
4           @Override
5            public void handleMessage(Message msg) {
6                 super.handleMessage(msg);
7                  ...相应的处理代码
8            }
9       }
10      ...相应的处理代码
11  }
12
13  public class MainFragment extends Fragment{
14      //保存 Activity 传递的 handler
15      private Handler mHandler;
16      @Override
17      public void onAttach(Activity activity) {
18           super.onAttach(activity);
19           //这个地方已经产生了耦合，若还有其他的 activity，这个地方就得修改
20           if(activity instance MainActivity){
21                mHandler = ((MainActivity)activity).mHandler;
22           }
23      }
24      ...相应的处理代码
```

```
25    }
```

该方案存在的缺点：

- Fragment 对具体的 Activity 存在耦合，不利于 Fragment 复用。
- 不利于维护，若想删除相应的 Activity，Fragment 也得改动。
- 没法获取 Activity 的返回数据。

2) 广播方案：

该方案的缺点：

- 用广播解决此问题有点大材小用了，广播的意图是用在一对多，接收广播者是未知的情况
- 广播性能肯定会差
- 传播数据有限制（必须得实现序列化接口才可以）

3) EventBus 方案：

该方案的缺点：

- EventBus 是用反射机制实现的，性能上会有问题
- EventBus 难于维护代码
- 没法获取 Activity 的返回数据

4) 接口方案：

```
1     // MainActivity 实现 MainFragment 开放的接口
2     public class MainActivity extends FragmentActivity implements FragmentListener{
3         @override
4         public void toH5Page(){ }
5         ...其他处理代码省略
6     }
7
8     public class MainFragment extends Fragment{
9
10        public FragmentListener mListener;
11        // MainFragment 开放的接口
12        public static interface FragmentListener{
13            //跳到 h5 页面
14            void toH5Page();
15        }
16
17        @Override
18        public void onAttach(Activity activity) {
19            super.onAttach(activity);
20            // 对传递进来的 Activity 进行接口转换
21            if(activity instance FragmentListener){
22                mListener = ((FragmentListener)activity);
23            }
24        }
25        ...其他处理代码省略
26    }
```

该方案的缺点：

- 需要为每对 Activity 与 Fragment 交互定义交互接口，包括为接口的命名，新定义的接口相应的 Activity 还得实现，相应的 Fragment 还得进行强制转换。

5) 模拟 javascript function 方案：（链接）

设计思路：

- 用一个类来模拟 Javascript 中的一个 Function

Function 就是此类，它是一个基类，每个 Functioon 实例都有一个 mFuncName 既然是方法（或者函数）它就有有参数和无参数之分

FunctionWithParam 是 Function 的子类，代表有参数的方法类，方法参数通过泛型解决

FunctionNoParamAndResult 是 Function 的子类，代表无参无返回值的方法类

- 一个可以存放多个方法（或者函数）的类

Functions 类就是此类,下面简单介绍下 Functions 有 4 个主要方法:

addFunction(FunctionNoParamAndResult function) 添加一个无参无返回值的方法类

addFunction(FunctionWithParam function) 添加一个有参无返回值的方法类

invokeFunc(String funcName) 根据 funcName 调用一个方法

invokeFunc(String funcName,Param param) 根据 funcName 调用有参无返回值的方法类

22、 ContentProvider 如何实现数据 ？

ContentProvider 是应用程序之间共享数据的接口。使用的时候首先自定义 一个类继承 ContentProvider，然后覆写 query、insert、update、delete 等方法。之后在 AndroidManifest 文件中进行注册。

android 系统下不同程序数据默认是不能共享访问，需要去实现一个类去继承 ContentProvider，把自己的数据通过 uri 的形式共享出去。第三方可以通过 ContentResolver 来访问该 Provider。

ContentProvider

```
public class StudentProvider extends ContentProvider {

    private final String TAG = "main";
    private StudentDAO studentDao = null;
    private static final UriMatcher URI_MATCHER = new UriMatcher(
            UriMatcher.NO_MATCH);
    private static final int STUDENT = 1;
    private static final int STUDENTS = 2;
    static {
        //添加两个 URI 筛选
        URI_MATCHER.addURI("com.example.contentproviderdemo.StudentProvider",
                "student", STUDENTS);
        //使用通配符#，匹配任意数字
        URI_MATCHER.addURI("com.example.contentproviderdemo.StudentProvider",
                "student/#", STUDENT);
    }

    public StudentProvider() {
    }

    @Override
    public boolean onCreate() {
        // 初始化一个数据持久层
        studentDao = new StudentDAO(getContext());
        Log.i(TAG, "---->>onCreate()被调用");
        return true;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        Uri resultUri = null;
        //解析 Uri，返回 Code
        int flag = URI_MATCHER.match(uri);
        if (flag == STUDENTS) {
            long id = studentDao.insertStudent(values);
            Log.i(TAG, "---->>插入成功, id="+id);
            resultUri = ContentUris.withAppendedId(uri, id);
        }
        return resultUri;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        int count = -1;
        try {
            int flag = URI_MATCHER.match(uri);
            switch (flag) {
            case STUDENT:
                // delete from student where id=?
```

```
50                  //单条数据，使用 ContentUris 工具类解析出结尾的 Id
51                  long id = ContentUris.parseId(uri);
52                  String where_value = "id = ?";
53                  String[] args = { String.valueOf(id) };
54                  count = studentDao.deleteStudent(where_value, args);
55                  break;
56              case STUDENTS:
57                  count = studentDao.deleteStudent(selection, selectionArgs);
58                  break;
59              }
60          } catch (Exception e) {
61              e.printStackTrace();
62          }
63          Log.i(TAG, "---->>删除成功,count="+count);
64          return count;
65      }
66
67      @Override
68      public int update(Uri uri, ContentValues values, String selection,
69              String[] selectionArgs) {
70          int count = -1;
71          try {
72              int flag = URI_MATCHER.match(uri);
73              switch (flag) {
74              case STUDENT:
75                  long id = ContentUris.parseId(uri);
76                  String where_value = " id = ?";
77                  String[] args = { String.valueOf(id) };
78                  count = studentDao.updateStudent(values, where_value, args);
79                  break;
80              case STUDENTS:
81                  count = studentDao.updateStudent(values, selection,
82                          selectionArgs);
83                  break;
84              }
85          } catch (Exception e) {
86              e.printStackTrace();
87          }
88          Log.i(TAG, "---->>更新成功，count="+count);
89          return count;
90      }
91
92      @Override
93      public Cursor query(Uri uri, String[] projection, String selection,
94              String[] selectionArgs, String sortOrder) {
95          Cursor cursor = null;
96          try {
97              int flag = URI_MATCHER.match(uri);
98              switch (flag) {
99              case STUDENT:
100                 long id = ContentUris.parseId(uri);
101                 String where_value = " id = ?";
102                 String[] args = { String.valueOf(id) };
103                 cursor = studentDao.queryStudents(where_value, args);
104                 break;
105             case STUDENTS:
106                 cursor = studentDao.queryStudents(selection, selectionArgs);
107                 break;
108             }
109         } catch (Exception e) {
110             e.printStackTrace();
111         }
112         Log.i(TAG, "---->>查询成功，Count="+cursor.getCount());
113         return cursor;
```

```
    }


    @Override
    public String getType(Uri uri) {
        int flag = URI_MATCHER.match(uri);
        String type = null;
        switch (flag) {
        case STUDENT:
            type = "vnd.android.cursor.item/student";
            Log.i(TAG, "----->>getType return item");
            break;
        case STUDENTS:
            type = "vnd.android.cursor.dir/students";
            Log.i(TAG, "----->>getType return dir");
            break;
        }
        return type;
    }
    @Override
    public Bundle call(String method, String arg, Bundle extras) {
        Log.i(TAG, "------>>"+method);
        Bundle bundle=new Bundle();
        bundle.putString("returnCall", "call 被执行了");
        return bundle;
    }
}
```

ContentResolver

```
public class MyTest extends AndroidTestCase {

    public MyTest() {
    }

    public void insert() {
        ContentResolver contentResolver = getContext().getContentResolver();
        Uri uri = Uri
                .parse("content://com.example.contentproviderdemo.StudentProvider/student");
        ContentValues values = new ContentValues();
        values.put("name", "Demo");
        values.put("address", "HK");
        Uri returnuir = contentResolver.insert(uri, values);
        Log.i("main", "--------------->" + returnuir.getPath());
    }

    public void delete() {
        ContentResolver contentResolver = getContext().getContentResolver();
        // 删除多行：content://com.example.contentproviderdemo.StudentProvider/student
        Uri uri = Uri
                .parse("content://com.example.contentproviderdemo.StudentProvider/student/2");
        contentResolver.delete(uri, null, null);
    }

    public void deletes() {
        ContentResolver contentResolver = getContext().getContentResolver();
        Uri uri = Uri
                .parse("content://com.example.contentproviderdemo.StudentProvider/student");
        String where = "address=?";
        String[] where_args = { "HK" };
        contentResolver.delete(uri, where, where_args);
    }

    public void update() {
```

```
35          ContentResolver contentResolver = getContext().getContentResolver();
36          Uri uri = Uri
37                  .parse("content://com.example.contentproviderdemo.StudentProvider/student/2");
38          ContentValues values = new ContentValues();
39          values.put("name", "李四");
40          values.put("address", "上海");
41          contentResolver.update(uri, values, null, null);
42      }
43
44      public void updates() {
45          ContentResolver contentResolver = getContext().getContentResolver();
46          Uri uri = Uri
47                  .parse("content://com.example.contentproviderdemo.StudentProvider/student");
48          ContentValues values = new ContentValues();
49          values.put("name", "王五");
50          values.put("address", "深圳");
51          String where = "address=?";
52          String[] where_args = { "beijing" };
53          contentResolver.update(uri, values, where, where_args);
54      }
55
56      public void query() {
57          ContentResolver contentResolver = getContext().getContentResolver();
58          Uri uri = Uri
59                  .parse("content://com.example.contentproviderdemo.StudentProvider/student/2");
60          Cursor cursor = contentResolver.query(uri, null, null, null, null);
61          while (cursor.moveToNext()) {
62              Log.i("main",
63                      "--------------->"
64                              + cursor.getString(cursor.getColumnIndex("name")));
65          }
66      }
67
68      public void querys() {
69          ContentResolver contentResolver = getContext().getContentResolver();
70          Uri uri = Uri
71                  .parse("content://com.example.contentproviderdemo.StudentProvider/student");
72          String where = "address=?";
73          String[] where_args = { "深圳" };
74          Cursor cursor = contentResolver.query(uri, null, where, where_args,
75                  null);
76          while (cursor.moveToNext()) {
77              Log.i("main",
78                      "--------------->"
79                              + cursor.getString(cursor.getColumnIndex("name")));
80          }
81      }
82
83      public void calltest() {
84          ContentResolver contentResolver = getContext().getContentResolver();
85          Uri uri = Uri
86                  .parse("content://com.example.contentproviderdemo.StudentProvider/student");
87          Bundle bundle = contentResolver.call(uri, "method", null, null);
88          String returnCall = bundle.getString("returnCall");
89          Log.i("main", "--------------->" + returnCall);
90      }
91  }
```

23、 说说 ContentProvider 、 ContentResolver 、 ContentObserver 之间的关系

ContentProvider 内容提供者,用于对外提供数据

ContentResolver.notifyChange(uri)发出消息

ContentResolver 内容解析者,用于获取内容提供者提供的数据

ContentObserver 内容监听器,可以监听数据的改变状态

ContentResolver.registerContentObserver()监听消息。

24、 Uri 介绍

为系统的每一个资源命名一个名字，比方说通话记录：

- 每一个 ContentProvider 都拥有一个公共的 URI，这个 URI 用于表示这个 ContentProvider 所提供的数据。
- Android 所提供的 ContentProvider 都存放在 android.provider 包中。 将其分为 A，B，C，D 4 个部分：

  A：标准前缀，用来说明一个 Content Provider 控制这些数据，无法改变的；"content://"；

  B：URI 的标识，用于唯一标识这个 ContentProvider，外部调用者可以根据这个标识来找到它。它定义了是哪个 Content Provider 提供这些数据。对于第三方应用程序，为了保证 URI 标识的唯一性，它必须是一个完整的、小写的类名。这个标识在元素的 authorities 属性中说明：一般是定义该 ContentProvider 的包和类的名称；

  C：路径（path），通俗的讲就是你要操作的数据库中表的名字，或者你也可以自己定义，记得在使用的时候保持一致就可以了；"content://com.bing.provider.myprovider/tablename"；

  D：如果 URI 中包含表示需要获取的**记录的 ID**；则就返回该 id 对应的数据，如果没有 ID，就表示返回全部；"content://com.bing.provider.myprovider/tablename/#" #表示数据 id。

25、 为什么要用 ContentProvider ？它和 sql 的实现上有什么差别 ？

ContentProvider 屏蔽了数据存储的细节，内部实现对用户完全透明，用户只需要关心操作数据的 uri 就可以了,ContentProvider 可以实现不同 app 之间共享。

Sql 也有增删改查的方法，但是 sql 只能查询本应用下的数据库。而 ContentProvider 还可以去增删改查本地文件、 xml 文件的读取等。

26、 Android 的数据存储方式

- File 存储
- SharedPreference 存储
- ContentProvider 存储
- SQLiteDataBase 存储
- 网络存储

27、 如何访问 asserts 资源目录下的数据库 ？

把数据库 db 复制到 /data/data/packagename/databases/目录下，直接访问。

28、 Intent 的作用

Android 中提供了 Intent 机制来协助应用间的交互与通讯。Intent 是一种运行时绑定（runtime binding)机制，它能在程序运行的过程中连接两个不同的组件。通过 Intent，你的程序可以向 Android 表达某种请求或者意愿，Android 会根据意愿的内容选择适当的组件来响应。

activity、service 和 broadcast receiver 之间是通过 Intent 进行通信的，而另外一个组件 **Content Provider** 本身就是一种通信机制，不需要通过 Intent。

- 使用 Context.startActivity() 或 Activity.startActivityForResult()，传入一个 intent 来启动一个 activity。使用 Activity.setResult()，传入一个 intent 来从 activity 中返回结果。
- 将 intent 对象传给 Context.startService()来启动一个 service 或者传消息给一个运行的 service。将 intent 对象传给 Context.bindService()来绑定一个 service。
- 将 intent 对象传给 Context.sendBroadcast()，Context.sendOrderedBroadcast()，或者 Context.sendStickyBroadcast()等广播方法，则它们被传给 broadcast receiver。

29、 显式和隐式 Intent

显式 Intent

直接指定类名进行跳转。

隐式 Intent

不明确指出我们想要启动哪一个活动，而是指定一系列更为抽象的 action 和 category 等信息，然后交由系统去分析这个 Intent，并帮我们找出合适的组件去启动。

30、 隐式 Intent 的匹配内容

- **Action（动作）：用来表现意图的行动**

如果程序的 manifest.xml 中的某一个 Activity 的 IntentFilter 中定义了 Action，恰好与目标 Action 匹配，且其 IntentFilter 中没有定义其它的 Type 或 Category 过滤条件，那么这个 Activity 就匹配了。但是如果手机中有两个以上的程序匹配，那么就会弹出一个对话可框来提示说明。

- **category（类别）：用来表现动作的类别**

一般不要去在 Intent 中设置它，如果是写 Intent 的接收者，就在 manifest.xml 的 Activity 的 IntentFilter 中包含 android.category.DEFAULT，这样所有不设置 Category 的 Intent 都会与这个 Category 匹配。只有<action>和<category>中的内容同时能够匹配上 Intent 中指定的 action 和 category 时，这个活动才能响应 Intent。如果使用的是 DEFAULT 这种默认的 category，在稍后调用 startActivity()方法的时候会自动将这个 category 添加到 Intent 中。

- **data（数据）：表示与动作要操纵的数据**
- **type（数据类型）：对于 data 范例的描写**

Data 是用一个 uri 对象来表示的，uri 代表数据的地址，属于一种标识符。通常情况下，我们使用 action+data 属性的组合来描述一个意图：做什么。

Type 属性用于明确指定 Data 属性的数据类型或 MIME 类型，但是通常来说，当 Intent 不指定 Data 属性时，Type 属性才会起作用，否则 Android 系统将会根据 Data 属性值来分析数据的类型，所以无需指定 Type 属性。

data 和 type 属性一般只需要一个，通过 setData 方法会把 type 属性设置为 null，相反设置 setType 方法会把 data 设置为 null，如果想要两个属性同时设置，要使用 Intent.setDataAndType()方法。

- **extras（扩展信息）：扩展信息**

是其它所有附加信息的集合。使用 extras 可以为组件提供扩展信息，比如，如果要执行“发送电子邮件”这个动作，可以将电子邮件的标题、正文等保存在 extras 里，传给电子邮件发送组件。

- **Flags（标志位）：期望这个意图的运行模式**

31、 IntentFilter

对于显式 Intent，它的接受者已被指定，所以系统会自动把这个 Intent 发给指定的组件。但是对于隐式 Intent，由于并没有指定其组件名属性，所以系统不知道该把它发给哪个组件名，于是系统就直接将其发出去，算是所有的组件都有权接收，这就需要定义一个组件可以接收到哪些 Intent，所以就引入了 IntentFilter

32、 Intent 传递数据时，可以传递哪些类型数据 ？

基本数据类型、String 以及他们的数组形式，实现了 Serializable 和 Parcelable 接口的对象。

33、 Serializable 和 Parcelable 的区别

1) 在使用内存的时候，Parcelable 类比 Serializable 性能高，所以推荐使用 Parcelable 类。
2) Serializable 在序列化的时候会产生大量的临时变量，从而引起频繁的 GC。
3) Parcelable 不能使用在要将数据存储在磁盘上的情况。尽管 Serializable 效率低点，但在这种情况下，建议使用 Serializable 。

实现方式：

Serializable

```
1    public class Person implements Serializable {
2
3        private static final long serialVersionUID = -3139325922167935911L;
4        private int age;
5
6        public int getAge() {
7            return age;
8        }
9
10       public void setAge(int age) {
11           this.age = age;
12       }
13   }
```

Parcelable

```
1    public class MyParcelable implements Parcelable {
```

```
    private int mData;
    private String mStr;

    public int describeContents() {
        return 0;
    }

    // 写数据进行保存
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(mData);
        out.writeString(mStr);
    }

    // 用来创建自定义的 Parcelable 的对象
    public static final Parcelable.Creator<MyParcelable> CREATOR
            = new Parcelable.Creator<MyParcelable>() {
        public MyParcelable createFromParcel(Parcel in) {
            return new MyParcelable(in);
        }

        public MyParcelable[] newArray(int size) {
            return new MyParcelable[size];
        }
    };

    // 读数据进行恢复
    private MyParcelable(Parcel in) {
        mData = in.readInt();
        mStr = in.readString();
    }
}
```

34、 BroadcastReceiver

内部通信实现机制:通过 Android 系统的 Binder 机制实现通信。

无序广播：完全异步，逻辑上可以被任何广播接收者接收到。

优点是效率较高。

缺点是一个接收者不能将处理结果传递给下一个接收者，并无法终止广播 intent 的传播。

有序广播：按照被接收者的优先级顺序，在被接收者中依次传播。比如有三个广 播接收者 A,B,C,优先级是 A > B > C。那这个消息先传给 A,再传给 B,最 后传给 C。每个接收者有权终止广播,比如 B 终止广播,C 就无法接收到。此外 A 接收到广播后可以对结果对象进行操作,当广播传给 B 时,B 可以从结果对象 中取得 A 存入的数据。

在通过 Context.sendOrderedBroadcast() 时我们可以指定 resultReceiver 广播接收者，这个接收者是最终接收者，通常情况下如果比他优先级更高的接收者如果没有终止广播，那么他的 onReceive 会被执行两次，第一次是正常的按照优先级顺序执行，第二次是作为最终接收者接收。如果比他优先级高的接收者终止了广播，那么他依然能接收到广播。

Android 4.0 之后,如果系统自动关闭广播接收者所在进程,在广播中的 action 跟广播接收者的 action 匹配时,系统会启动该广播所在的进程,但是如果是用户手动关闭该进程,则不会自启动,只有等用户手动开启，广播接收者所在进程如果从来没有启动过,那么广播接收者不会生效。

35、 BroadcastReceiver 注册方式

静态注册：在清单文件中注册，接收者只要 app 在系统中运行则一直可以接收到广播消息。

```
<receiver android:name=".BroadcastReceiver1" >
    <intent-filter>
        <action android:name="android.intent.action.CALL" >
        </action>
    </intent-filter>
</receiver>
```

动态注册：在代码中注册，接收者当注册的 Activity 或者 Service 被销毁，就接收不到广播了。

```
receiver = new BroadcastReceiver();
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(CALL_ACTION);
context.registerReceiver(receiver, intentFilter);
```

36、 LocalBroadcastManager

Android v4 兼容包提供 android.support.v4.content.LocalBroadcastManager 工具类，帮助大家在自己的进程内进行局部广播发送与注册，使用它比直接通过 sendBroadcast(Intent) 发送系统全局广播有以下几点好处。

1) 因广播数据在本应用范围内传播，你不用担心隐私数据泄露的问题。
2) 不用担心别的应用伪造广播，造成安全隐患。
3) 相比在系统内发送全局广播，它更高效。

注意：*LocalBroadcastManager 方式发送的应用内广播，只能通过 LocalBroadcastManager 动态注册的 ContextReceiver 才有可能接收到（静态注册或其他方式动态注册的 ContextReceiver 是接收不到的）。*

37、 Service 是否在 main thread 中执行，service 里面是否能执行耗时的操作 ？

默认情况,如果没有显示的指 service 所运行的进程, Service 和 activity 是运行在当前 app 所在进程的 main thread(UI 主线程)里面。 service 里面不能执行耗时的操作(网络请求，拷贝数据库，大文件) 。特殊情况，可以在清单文件配置 service 执行所在的进程，让 service 在另外的进程中执行。

```
<service android:name="com.baidu.location.f"
         android:enabled="true"
         android:process=":remote" />
```

38、 Service 的启动方式以及生命周期

startService

当第一次调用 startService 的时候执行的方法依次为 onCreate()、onStartCommand()，当使用 stopService() 关闭 Service 的时候会调用 onDestroy() 方法，如果是调用者自己直接退出而没有调用 stopService() 的话，Service 会一直在后台运行。如果当前要启动的 Service 已经存在了，那么就不会再次创建该 Service，当然也不会调用 onCreate() 方法。

bindService

bindService() 的时候，执行的方法为 onCreate()、 onBind()，调用 unbindService() 解除绑定的时候会执行 onUnbind()、onDestory()。**onRebind() 只有在 onUnbind() 方法返回 true 的情况下会执行。**

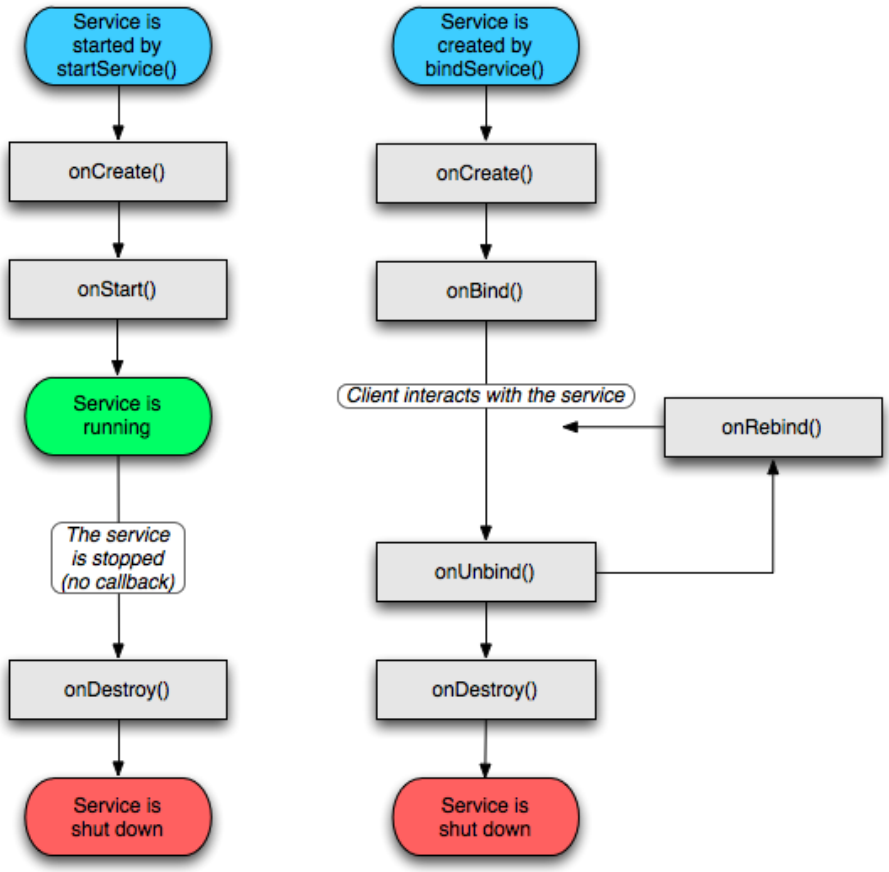
如果先 bindService()，那么调用 startService() 的时候就直接运行 Service 的 onStart() 方法；如果先是 startService()，那么 startService() 的时候就直接运行 onBind() 方法。

如果 service 运行期间调用了 bindService()，这时候再调用 stopService() 的话，service 不会调用 onDestroy() 方法，只有调用 bindService() 方法，service 才会被销毁 。

多次调用 startService() 会多次调用 onStart() 方法，而多次调用 stopService() 只会调用一次 bindService() 方法。

多次调用 bindService() 只会调用一次 onBind() 方法，而多次调用 unbindService() 会抛出异常。

注意：不管是 start 还是 bind，如果当前要启动的 Service 已经存在了，那么就不会再次创建该 Service，当然也不会调用 onCreate() 方法。



39、 Activity、Intent、Service 的关系

Activity 和 Service 都是 ContextWrapper（Context 类的子类） 的子类，Activity 负责用户界面的显示和交互，Service 负责后台任务的处理。Activity 和 Service 之间可以通过 Intent 传递数据，因此可以把 Intent 看作是通信使者。

40、 如何提高 service 的优先级，使 Service 服务不被杀死 ？

1) 提高 Service 的优先级：这个只能说在系统内存不足需要回收资源的时候，优先级较高，不容易被回收。
2) 提高 Service 所在进程的优先级：效果不是很明显。
3) 在 onDestroy 方法里重启 service：这个算有效的方法，但是直接干掉进程的时候，onDestroy 方法都进不来，更别想重启了。
4) broadcast 广播：和第 3 种一样，没进入 onDestroy，就不知道什么时候发广播了，另外，在 Android4.4 以上，程序完全退出后，就不好接收广播了，需要在发广播的地方特定处理。
5) 放到 System/app 底下作为系统应用：这个也就是平时玩玩，没多大的实际意义。
6) Service 的 onStartCommand 方法，返回 START_STICKY，这个也主要是针对系统资源不足而导致的服务被关闭，还是有一定的道理的。
7) setForeground(true)：Service 设置了 foreground 那么他就和正在运行的 Activity 类似优先级得到了一定的提高。这并不能保证你得 Service 永远不被杀掉，只是提高了他的优先级。
8) 通过 JNI 实现守护进程：通过 JNI 的方式（NDK 编程），fork 出一个子线程作为守护进程，轮询监听服务状态。守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。而守护进程的会话组和当前目录，文件描述符都是独立的。后台运行只是终端进行了一次 fork，让程序在后台执行，这些都没有改变。

41、 service 如何定时执行 ？

当启动 service 进行后台任务的时候，一般做法是启动一个线程，然后通过 sleep 方法来控制进行定时的任务，如轮询操作，消息推送，但是这样容易被系统回收。service 被回收是我们不能控制的，但是我们可以控制 service 的重启活动。在 service 的 onStartCommand 方法中可以返回一个参数来控制重启活动。

根据 AlarmManager 的工作原理，AlarmManager 会定时的发出一条广播，在项目里面注册这个广播，并重写 onReceive 方法，在这个方法里面启动一个 service，然后在 service 里面进行网络的访问操作，当获取到新消息的时候进行推送，同时再设置一个

AlarmManager 进行下一次的轮询，当本次轮询结束的时候可以 stopSelf 结束该 service。这样即使这一次的轮询失败了，也不会影响到下一次的轮询，如此就能保证推送任务不会中断。

42、 Service 的 onStartCommand 方法的返回值

START_NOT_STICKY：如果系统在 onStartCommand()方法返回之后杀死这个服务，那么直到接受到新的 Intent 对象，这个服务才会被重新创建，系统不会自动重启服务。这是最安全的选项，用来避免在不需要的时候运行你的服务。

START_STICKY：如果系统在 onStartCommand() 返回后杀死了这个服务，系统就会重新创建这个服务并且调用 onStartCommand() 方法，但是它不会重新传递最后的 Intent 对象，系统会用一个 null 的 Intent 对象来调用 onStartCommand() 方法，如果在此期间没有任何启动命令被传递到 service，那么参数 Intent 将为 null。这适用于不执行命令的媒体播放器（或类似的服务），它只是无限期的运行着并等待工作的到来。

START_REDELIVER_INTENT：系统在 onStartCommand() 方法返回后，服务被异常 kill 掉，就会重新创建这个服务，并将最后一次传送给服务的 Intent 对象发给 onStartCommand() 方法。任意等待中的 Intent 对象会依次被发送。这适用于那些应该立即恢复正在执行的工作的服务，如下载文件。

START_STICKY_COMPATIBILITY：START_STICKY 的兼容版本，但不保证服务被 kill 后一定能重启。

43、 Service 里面弹出 Toast

弹 Toast 的条件是有一个 Context，而 Service 本身就是 Context 的子类，因此在 Service 里面弹 Toast 是完全可以的。比如我们在 Service 中完成下载任务后可以弹一个 Toast 通知用户。

44、 IntentService

- 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 内置的是 HandlerThread 作为异步线程，每一个交给 IntentService 的任务都将以队列的方式逐个被执行到，一旦队列中有某个任务执行时间过长，那么就会导致后续的任务都会被延迟处理
- 正在运行的 IntentService 的程序相比起纯粹的后台程序更不容易被系统杀死，该程序的优先级是介于前台程序与纯后台程序之间的。

```
1    public class MIntentService extends IntentService {
2
3        public MIntentService(){
4            super("MIntentService");
5        }
6
7        /**
8         * Creates an IntentService.  Invoked by your subclass's constructor.
9         * @param name Used to name the worker thread, important only for debugging.
10        */
11       public MIntentService(String name) {
12           super(name);
13       }
14
15       @Override
16       public void onCreate() {
17           Log.e("MIntentService", "onCreate");
18           super.onCreate();
19       }
20
21       @Override
22       public int onStartCommand(Intent intent, int flags, int startId) {
23           Log.e("MIntentService", "onStartCommand");
24           return super.onStartCommand(intent, flags, startId);
25       }
26
27       @Override
```

```
28        protected void onHandleIntent(Intent intent) {
29            Log.e("MIntentService", Thread.currentThread().getName() + "--" +
30    intent.getStringExtra("info") );
31            for(int i = 0; i < 100; i++){ //耗时操作
32                Log.i("onHandleIntent",  i + "--" + Thread.currentThread().getName());
33            }
34        }
35
36        @Override
37        public void onDestroy() {
38            Log.e("MIntentService", "onDestroy");
39            super.onDestroy();
40        }
41    }
```

45、 Activity 调用 Service 中的方法都有哪些方式 ？

- **Binder：**

如果服务是提供给自有应用专用的，并且 Service(服务端)与客户端相同的进程中运行（常见情况），则应通过扩展 Binder 类并从 onBind() 返回它的一个实例来创建接口。客户端收到 Binder 后，可利用它直接访问 Binder 实现中以及 Service 中可用的公共方法。如果我们的服务只是自有应用的后台工作线程，则优先采用这种方法。 不采用该方式创建接口的唯一原因是，服务被其他应用或不同的进程调用。

步骤：

1) 创建 BindService 服务端，继承自 Service 并在类中，创建一个实现 IBinder 接口的实例对象并提供公共方法给客户端调用；
2) 从 onBind() 回调方法返回此 Binder 实例；
3) 在客户端中，从 onServiceConnected() 回调方法接收 Binder，并使用提供的方法调用绑定服务。

```
1     /**
2      * 服务端
3      */
4     public class LocalService extends Service{
5         private final static String TAG = "LocalService";
6         private int count;
7         private boolean quit;
8         private Thread thread;
9         private LocalBinder binder = new LocalBinder();
10
11        /** 创建Binder对象，返回给客户端即Activity使用，提供数据交换的接口 */
12        public class LocalBinder extends Binder {
13            // 声明一个方法，getService（提供给客户端调用）
14            LocalService getService() {
15                // 返回当前对象LocalService,这样我们就可在客户端端调用Service的公共方法了
16                return LocalService.this;
17            }
18        }
19
20        /** 把Binder类返回给客户端 */
21        @Nullable
22        @Override
23        public IBinder onBind(Intent intent) {
24            return binder;
25        }
26
27        @Override
28        public void onCreate() {
29            super.onCreate();
30            Log.i(TAG, "Service is invoke Created");
31            thread = new Thread(new Runnable() {
32                @Override
33                public void run() {
34                    // 每间隔一秒count加1 ，直到quit为true。
```

```
                while (!quit) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    count++;
                }
            }
        });
        thread.start();
    }

    /** 公共方法 */
    public int getCount(){
        return count;
    }

    /** 解除绑定时调用 */
     @Override
    public boolean onUnbind(Intent intent) {
        Log.i(TAG, "Service is invoke onUnbind");
        return super.onUnbind(intent);
    }

    @Override
    public void onDestroy() {
        Log.i(TAG, "Service is invoke Destroyed");
        this.quit = true;
        super.onDestroy();
    }
}
```

```
/**
 * 客户端
 */
public class BindActivity extends Activity {
    protected static final String TAG = "BindActivity";
    Button btnBind;
    Button btnUnBind;
    Button btnGetDatas;
    /**
     * ServiceConnection 代表与服务的连接，它有两个方法：
     * onServiceConnected 是在操作者在连接一个服务成功时被调用，
     * onServiceDisconnected 是在服务崩溃或被杀死导致的连接中断时被调用
     */
    private ServiceConnection conn;
    private LocalService mService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bind);
        btnBind = (Button) findViewById(R.id.BindService);
        btnUnBind = (Button) findViewById(R.id.unBindService);
        btnGetDatas = (Button) findViewById(R.id.getServiceDatas);
        //创建绑定对象
        final Intent intent = new Intent(this, LocalService.class);
        conn = new ServiceConnection() {
            /**
             * 与服务器端交互的接口方法 绑定服务的时候被回调，在这个方法获取绑定 Service
             * 传递过来的 IBinder 对象，通过这个 IBinder 对象，实现宿主和 Service 的交互。
             */
            @Override
```

```
32                public void onServiceConnected(ComponentName name, IBinder service) {
33                    Log.d(TAG, "绑定成功调用：onServiceConnected");
34                    // 获取 Binder
35                    LocalService.LocalBinder binder = (LocalService.LocalBinder) service;
36                    mService = binder.getService();
37                }
38                /**
39                 * 当取消绑定的时候被回调。但正常情况下是不被调用的，它的调用时机是当 Service
40                 * 服务被意外销毁时，例如内存的资源不足时这个方法才被自动调用。
41                 * 注意:当客户端取消绑定时，系统“绝对不会”调用该方法。
42                 */
43                @Override
44                public void onServiceDisconnected(ComponentName name) {
45                    mService=null;
46                }
47            };
48
49            // 开启绑定
50            btnBind.setOnClickListener(new View.OnClickListener() {
51                @Override
52                public void onClick(View v) {
53                    Log.d(TAG, "绑定调用：bindService");
54                    //调用绑定方法
55                    bindService(intent, conn, Service.BIND_AUTO_CREATE);
56                }
57            });
58            // 解除绑定
59            btnUnBind.setOnClickListener(new View.OnClickListener() {
60                @Override
61                public void onClick(View v) {
62                    Log.d(TAG, "解除绑定调用：unbindService");
63                    // 解除绑定
64                    if(mService!=null) {
65                        mService = null;
66                        unbindService(conn);
67                    }
68                }
69            });
70            // 获取数据
71            btnGetDatas.setOnClickListener(new View.OnClickListener() {
72                @Override
73                public void onClick(View v) {
74                    if (mService != null) {
75                        // 通过绑定服务传递的 Binder 对象，获取 Service 暴露出来的数据
76                        Log.d(TAG, "从服务端获取数据：" + mService.getCount());
77                    } else {
78                        Log.d(TAG, "还没绑定呢，先绑定,无法从服务端获取数据");
79                    }
80                }
81            });
82        }
83    }
```

- **Messenger：**

Messenger 可以翻译为信使，通过它可以在不同的进程中共传递 Message 对象(Handler 中的 Messenger，因此 Handler 是 Messenger 的基础)，在 Message 中可以存放我们需要传递的数据，然后在进程间传递。如果需要让接口跨不同的进程工作，则可使用 Messenger 为服务创建接口，客户端就可利用 Message 对象向服务发送命令。同时客户端也可自定义 Messenger，以便服务回传消息。这是执行进程间通信 (IPC) 的最简单方法，因为 Messenger 会在单一线程中创建包含所有请求的队列，也就是说 Messenger 是以串行的方式处理客户端发来的消息，这样我们就不必对服务进行线程安全设计了。

步骤：

1) 服务实现一个 Handler，由其接收来自客户端的每个调用的回调；

2) Handler 用于创建 Messenger 对象 (对 Handler 的引用)；

3) Messenger 创建一个 IBinder，服务通过 onBind() 使其返回客户端；

4) 客户端使用 IBinder 将 Messenger（引用服务的 Handler）实例化，然后使用 Messenger 将 Message 对象发送给服务；

5) 服务在其 Handler 中（在 handleMessage() 方法中）接收每个 Message。

```
/**
 * 服务端进程
 */
public class MessengerService extends Service {

    /** Command to the service to display a message */
    static final int MSG_SAY_HELLO = 1;
    private static final String TAG ="MessengerService" ;

    /**
     * 用于接收从客户端传递过来的数据
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Log.i(TAG, "thanks,Service had receiver message from client!");
                    //回复客户端信息,该对象由客户端传递过来
                    Messenger client=msg.replyTo;
                    //获取回复信息的消息实体
                    Message replyMsg=Message.obtain(null,MessengerService.MSG_SAY_HELLO);
                    Bundle bundle=new Bundle();
                    bundle.putString("reply","ok~,I had receiver message from you! ");
                    replyMsg.setData(bundle);
                    //向客户端发送消息
                    try {
                        client.send(replyMsg);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    /**
     * 创建 Messenger 并传入 Handler 实例对象
     */
    final Messenger mMessenger = new Messenger(new IncomingHandler());

    /**
     * 当绑定 Service 时,该方法被调用,将通过 mMessenger 返回一个实现
     * IBinder 接口的实例对象
     */
    @Override
    public IBinder onBind(Intent intent) {
        Log.i(TAG, "Service is invoke onBind");
        return mMessenger.getBinder();
    }
}
```

```
/**
 * 与服务器交互的客户端
 */
public class ActivityMessenger extends Activity {
    /** 与服务端交互的 Messenger */
    Messenger mService = null;
```

```
7
8       /** Flag indicating whether we have called bind on the service. */
9       boolean mBound;
10
11      /*** 用于接收服务器返回的信息  */
12      private Messenger mRecevierReplyMsg= new Messenger(new ReceiverReplyMsgHandler());
13
14      private static class ReceiverReplyMsgHandler extends Handler{
15          private static final String TAG = "ActivityMessenger";
16
17          @Override
18          public void handleMessage(Message msg) {
19               switch (msg.what) {
20                    //接收服务端回复
21                    case MessengerService.MSG_SAY_HELLO:
22                         Log.i(TAG, "receiver message from service:"
23                                   + msg.getData().getString("reply"));
24                         break;
25                    default:
26                         super.handleMessage(msg);
27               }
28          }
29      }
30
31      /**
32       * 实现与服务端链接的对象
33       */
34      private ServiceConnection mConnection = new ServiceConnection() {
35          public void onServiceConnected(ComponentName className, IBinder service) {
36              /**
37               * 通过服务端传递的 IBinder 对象,创建相应的 Messenger
38               * 通过该 Messenger 对象与服务端进行交互
39               */
40              mService = new Messenger(service);
41              mBound = true;
42          }
43
44          public void onServiceDisconnected(ComponentName className) {
45              // This is called when the connection with the service has been
46              // unexpectedly disconnected -- that is, its process crashed.
47              mService = null;
48              mBound = false;
49          }
50      };
51
52      public void sayHello(View v) {
53          if (!mBound) return;
54          // 创建与服务交互的消息实体 Message
55          Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
56          // 把接收服务器端的回复的 Messenger 通过 Message 的 replyTo 参数传递给服务端
57          msg.replyTo = mRecevierReplyMsg;
58          try {
59              //发送消息
60              mService.send(msg);
61          } catch (RemoteException e) {
62              e.printStackTrace();
63          }
64      }
65
66      @Override
67      protected void onCreate(Bundle savedInstanceState) {
68          super.onCreate(savedInstanceState);
69          setContentView(R.layout.activity_messenager);
70          Button bindService= (Button) findViewById(R.id.bindService);
```

```
        Button unbindService= (Button) findViewById(R.id.unbindService);
        Button sendMsg= (Button) findViewById(R.id.sendMsgToService);

        bindService.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d("zj","onClick-->bindService");
                //当前 Activity 绑定服务端
                bindService(new Intent(ActivityMessenger.this, MessengerService.class),
                            mConnection, Context.BIND_AUTO_CREATE);
            }
        });

        //发送消息给服务端
        sendMsg.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sayHello(v);
            }
        });

        unbindService.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Unbind from the service
                if (mBound) {
                    Log.d("zj","onClick-->unbindService");
                    unbindService(mConnection);
                    mBound = false;
                }
            }
        });
    }
}
```

```
<service android:name=".messenger.MessengerService"
         android:process=":remote"/>
```

- **AIDL:**

由于 Messenger 是以串行的方式处理客户端发来的消息，如果当前有大量消息同时发送到 Service(服务端)，Service 仍然只能一个个处理，这也就是 Messenger 跨进程通信的缺点了，因此如果有大量并发请求，Messenger 就会显得力不从心了，这时 AIDL（Android 接口定义语言）就派上用场了， 但实际上 Messenger 的跨进程底层实现方式就是 AIDL，只不过 android 系统帮我们封装成透明的 Messenger 罢了 。因此，如果我们想让服务同时处理多个请求，则应该使用 AIDL。 在此情况下，服务必须具备多线程处理能力，并采用线程安全式设计。使用 AIDL 必须创建一个定义编程接口的 .aidl 文件。Android SDK 工具利用该文件生成一个实现接口并处理 IPC 的抽象类，随后可在服务内对其进行扩展。

46、 前台服务以及通知发送

前台服务被认为是用户主动意识到的一种服务，因此在内存不足时，系统也不会考虑将其终止。 前台服务必须为状态栏提供通知，状态栏位于"正在进行"标题下方，这意味着除非服务停止或从前台删除，否则不能清除通知。例如将从服务播放音乐的音乐播放器设置为在前台运行，这是因为用户明确意识到其操作。 状态栏中的通知可能表示正在播放的歌曲，并允许用户启动 Activity 来与音乐播放器进行交互。官方提供了两个方法，分别是 startForeground() 和 stopForeground()。

```
/**
 * 服务端
 */
public class ForegroundService extends Service {

    /** id 不可设置为 0,否则不能设置为前台 service */
    private static final int NOTIFICATION_DOWNLOAD_PROGRESS_ID = 0x0001;

```

```
9       private boolean isRemove=false;//是否需要移除
10
11      /** Notification */
12      public void createNotification(){
13          //使用兼容版本
14          NotificationCompat.Builder builder=new NotificationCompat.Builder(this);
15          //设置状态栏的通知图标
16          builder.setSmallIcon(R.mipmap.ic_launcher);
17          //设置通知栏横条的图标
18          builder.setLargeIcon(BitmapFactory.decodeResource(getResources(),
19                                          R.drawable.screenflash_logo));
20          //禁止用户点击删除按钮删除
21          builder.setAutoCancel(false);
22          //禁止滑动删除
23          builder.setOngoing(true);
24          //右上角的时间显示
25          builder.setShowWhen(true);
26          //设置通知栏的标题内容
27          builder.setContentTitle("I am Foreground Service!!!");
28          //创建通知
29          Notification notification = builder.build();
30          //设置为前台服务
31          startForeground(NOTIFICATION_DOWNLOAD_PROGRESS_ID,notification);
32      }
33
34      @Override
35      public int onStartCommand(Intent intent, int flags, int startId) {
36          int i=intent.getExtras().getInt("cmd");
37          if(i==0){
38              if(!isRemove) {
39                  createNotification();
40              }
41              isRemove=true;
42          }else {
43              //移除前台服务
44              if (isRemove) {
45                  stopForeground(true);
46              }
47              isRemove=false;
48          }
49
50          return super.onStartCommand(intent, flags, startId);
51      }
52
53      @Override
54      public void onDestroy() {
55          //移除前台服务
56          if (isRemove) {
57              stopForeground(true);
58          }
59          isRemove=false;
60          super.onDestroy();
61      }
62
63      @Nullable
64      @Override
65      public IBinder onBind(Intent intent) {
66          return null;
67      }
68  }
```

```
1   /**
2    * 客户端
3    */
```

```
4    public class ForegroundActivity extends Activity {
5
6        @Override
7        protected void onCreate(Bundle savedInstanceState) {
8            super.onCreate(savedInstanceState);
9            setContentView(R.layout.activity_foreground);
10           Button btnStart= (Button) findViewById(R.id.startForeground);
11           Button btnStop= (Button) findViewById(R.id.stopForeground);
12           final Intent intent = new Intent(this,ForegroundService.class);
13           btnStart.setOnClickListener(new View.OnClickListener() {
14               @Override
15               public void onClick(View v) {
16                   intent.putExtra("cmd",0);//0,开启前台服务,1,关闭前台服务
17                   startService(intent);
18               }
19           });
20           btnStop.setOnClickListener(new View.OnClickListener() {
21               @Override
22               public void onClick(View v) {
23                   intent.putExtra("cmd",1);//0,开启前台服务,1,关闭前台服务
24                   startService(intent);
25               }
26           });
27       }
28   }
```

47、 ListView 外层嵌套 ScrollView 的处理

嵌套出现的问题： ListView 不能显示正常的条目，只显示一条或二条。

原因： 由于 ListView 在 ScrollView 中无法正确计算它的大小，故只显示一行。

解决方案：

1) 重写自定义 ListView, 并覆盖 onMeasure() 方法，达到使 ListView 适应 ScrollView 的效果。
2) 动态设置 ListView 的高度，不需要重写 ListView。只需要在 setAdapter 之后调用如下方法即可（这时最好给 ListView 之外嵌套一层 LinearLayout，不然有时候这种方法会失效）：

```
1    public void setListViewHeightBasedOnChildren(ListView listView) {
2            // 获取 ListView 对应的 Adapter
3            ListAdapter listAdapter = listView.getAdapter();
4            if (listAdapter == null) {
5                return;
6            }
7            int totalHeight = 0;
8            for (int i = 0, len = listAdapter.getCount(); i < len; i++) {
9                // listAdapter.getCount()返回数据项的数目
10               View listItem = listAdapter.getView(i, null, listView);
11               // 计算子项 View 的宽高
12               listItem.measure(0, 0);
13               // 统计所有子项的总高度
14               totalHeight += listItem.getMeasuredHeight();
15           }
16           ViewGroup.LayoutParams params = listView.getLayoutParams();
17           params.height = totalHeight + (listView.getDividerHeight() * (listAdapter.getCount() -
18   1));
19           // listView.getDividerHeight()获取子项间分隔符占用的高度
20           // params.height 最后得到整个 ListView 完整显示需要的高度
21           listView.setLayoutParams(params);
22       }
```

3) 在 xml 文件中，直接将 ListView 的高度写死。
4) 避免 ScrollView 嵌套 ListView，比如使用 ListView 的 addHeader() 函数来实现预期效果或者利用布局的特性达到预期效果。

48、 ListView 的优化

1) 复用 convertView，对 convetView 进行判空，当 convertView 不为空时重复使用，为空则初始化，从而减少了很多不必要的 View 的创建、减少 findViewById 的次数。
2) 避免在 getView 方法中做耗时操作。
3) 采用 ViewHolder 模式缓存 item 条目的引用。
4) 给 listView 设置滚动监听器 根据不同状态 不同处理数据 分批分页加载 根据 listView 的状态去操作，比如当列表快速滑动时不去开启大量的异步任务去请求图片。
5) listview 每个 item 层级结构不要太复杂。
6) listview 每个 item 中异步加载图片，并对图片加载做优化，(关于 Listview 分页加载和图片异步加载思路请看接下来的文章内容)。
7) listview 每个 item 中不要创建线程。
8) 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。
9) 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。
10) 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 可以实现当个 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。
11) 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。

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder viewHolder;
    //判断是否有缓存
    if (convertView == null) {
        //通过 LayoutInflate 实例化布局
        viewHolder = new ViewHolder();
        convertView = mInflater.inflate(R.layout.item_layout, parent, false);
        viewHolder.ivIcon = (ImageView) convertView.findViewById(R.id.iv_icon);
        viewHolder.tvTitle = (TextView) convertView.findViewById(R.id.tv_title);
        convertView.setTag(viewHolder);
    } else {
       //通过 tag 找到缓存的布局
        viewHolder = (ViewHolder) convertView.getTag();
    }
    NewsBean newsBean = newsBeanList.get(position);

    String urlString = newsBean.newsIconUrl;
    viewHolder.ivIcon.setTag(urlString); // 将 ImageView 与 url 绑定
    //普通异步加载
    // mImageLoader.showImageByThread(viewHolder.ivIcon,urlString);
    mImageLoader.showImageByAsyncTask(viewHolder.ivIcon,urlString);
    viewHolder.tvTitle.setText(newsBean.newsTitle);
    return convertView;
}

//使用 ViewHolder
private static class ViewHolder {
    private TextView tvTitle;
    private ImageView ivIcon;
}
```

49、 打开套有 ScrollView 的 ListView 页面布局，默认起始位置不是最顶部 ？

1) 把套在里面的 ListView 不让获取焦点即可。listview.setFocusable(false); **注意：在 xml 布局里面设置 android：focusable=“false”不生效。**

2) 使用 myScrollView.smoothScrollTo(0,0);

50、 ListView 上拉加载和下拉刷新怎么实现 ？

实现 OnScrollListener 接口重写 onScrollStateChanged 和 onScroll 方法。
使用 onScroll 方法实现"滑动" 后处理检查是否还有新的记录，如果有，调用 addFooterView，添加记录到 adapter，adapter 调 notifyDataSetChanged 更新数据，如果没有记录了，把自定义的 mFooterView 去掉。使用 onScrollStateChanged 可以检测是否滚到最后一行且停止滚动然后执行加载。

51、 ListView 失去焦点怎么处理 ？

在 ListView 子布局里面写，可以解决焦点失去的问题
android:descendantFocusability="blocksDescendants"

52、 ListView 图片异步加载实现思路 ？

1) 先从内存缓存中获取图片显示（内存缓冲）。
2) 获取不到的话从 SD 卡里获取（SD 卡缓冲，，从 SD 卡获取图片是放在子线程里执行的，否则快速滑屏的话会不够流畅）。
3) 都获取不到的话从网络下载图片并保存到 SD 卡同时加入内存并显示（视情况看是否要显示）。

53、 ListView 里有 Button 就点不动 ？

原因是 button 强制获取了 item 的焦点，只要设置 button 的 focusable 为 false 即可。

54、 ViewHolder 内部类为什么要声明成 static 的呢

如果声明成员类不要求访问外围实例，就要始终把 static 修饰符放在它的声明中，使它成为静态成员类，而不是非静态成员类。因为非静态成员类的实例会包含一个额外的指向外围对象的引用，保存这份引用要消耗时间和空间，并且导致外围类实例符合垃圾回收时仍然被保留。如果没有外围实例的情况下，也需要分配实例，就不能使用非静态成员类，因为非静态成员类的实例必须要有一个外围实例。

55、 ListView 如何显示多种类型的 Item ？

Adapter 提供了 getViewTypeCount() 和 getItemViewType(int position) 两个方法。在 getView 方法中我们可以根据不同的 viewtype 加载不同的布局文件。

56、 ListView 如何定位到指定位置 ？

可以通过 ListView 提供的 lv.setSelection(listView.getPosition()); 方法。

57、 RecyclerView 的适配器

```
1    public class BookBaseAdapter extends RecyclerView.Adapter<BookBaseAdapter.ViewHolder>{
2
3        private List<Book> mBookList;
4
5        static class ViewHolder extends RecyclerView.ViewHolder{
6            ImageView bookImage;
7            TextView bookname;
8
9            public ViewHolder(View view) {
10               super(view);
11               bookImage = (ImageView) view.findViewById(R.id.book_iamge);
12               bookname = (TextView) view.findViewById(R.id.book_name);
13           }
14       }
15
16       public BookBaseAdapter(List<Book> mBookList) {
17           this.mBookList = mBookList;
18       }
19
20        //加载 item 的布局  创建 ViewHolder 实例
21       @Override
22       public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
23           View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.book,parent,false);
```

```
            ViewHolder holder = new ViewHolder(view);
            return holder;
        }

        //对 RecyclerView 子项数据进行赋值
        @Override
        public void onBindViewHolder(ViewHolder holder, int position) {
            Book book = mBookList.get(position);
            holder.bookname.setText(book.getName());
            holder.bookImage.setImageResource(book.getImageId());
        }

        //返回子项个数
        @Override
        public int getItemCount() {
            return mBookList.size();
        }
    }
```

```
    //初始化 RecyclerView
    RecyclerView recyclerView = (RecyclerView) findViewById(R.id.recycler_view);
    //创建 LinearLayoutManager 对象 这里使用 LinearLayoutManager 是线性布局的意思
    LinearLayoutManager layoutManager = new LinearLayoutManager(this);
    //设置 RecyclerView 布局
    recyclerView.setLayoutManager(layoutManager);
    //设置为垂直布局，这也是默认的
    layoutManager.setOrientation(OrientationHelper. VERTICAL);
    //设置 Adapter
    BookBaseAdapter adapter = new BookBaseAdapter(mlsit);
    recyclerView.setAdapter(adapter);
    //设置分隔线
    recyclerView.addItemDecoration( new DividerGridItemDecoration(this ));
    //设置增加或删除条目的动画
    recyclerView.setItemAnimator( new DefaultItemAnimator());
```

58、 RecyclerView 回收和复用机制

- **问题**

RecyclerView 的回收复用机制的内部实现都是由 Recycler 内部类实现，假设有一个 20 个 item 的 RecyclerView,每五个占满一个屏幕，在从头滑到尾的过程中，onCreatViewHolder 会调用多少次?

```
    public final class Recycler {
        final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<>();
        ArrayList<ViewHolder> mChangedScrap = null;
        //这个是本篇的重点
        final ArrayList<ViewHolder> mCachedViews = new ArrayList<ViewHolder>();

        private final List<ViewHolder>
                mUnmodifiableAttachedScrap = Collections.unmodifiableList(mAttachedScrap);

        private int mRequestedCacheMax = DEFAULT_CACHE_SIZE;
        int mViewCacheMax = DEFAULT_CACHE_SIZE;
        //这个也是本篇的重点
        RecycledViewPool mRecyclerPool;

        private ViewCacheExtension mViewCacheExtension;

        static final int DEFAULT_CACHE_SIZE = 2;
    }
```

mAttachedScrap：用于缓存显示在屏幕上的 item 的 ViewHolder，场景好像是 RecyclerView 在 onLayout 时会先把 children 都移除掉，再重新添加进去，所以这个 List 应该是用在布局过程中临时存放 children 的，反正在 RecyclerView 滑动过程中不会在这里面来找复用的 ViewHolder 就是了。

mChangedScrap： 这个没理解是干嘛用的，看名字应该跟 ViewHolder 的数据发生变化时有关吧，在 RecyclerView 滑动的过程中，也没有发现到这里找复用的 ViewHolder，所以这个可以先暂时放一边。

mCachedViews：这个就重要得多了，滑动过程中的回收和复用都是先处理的这个 List，这个集合里存的 ViewHolder 的原本数据信息都在，所以可以直接添加到 RecyclerView 中显示，不需要再次重新 onBindViewHolder()。

mUnmodifiableAttachedScrap： 不清楚干嘛用的，暂时跳过。

mRecyclerPool：这个也很重要，但存在这里的 ViewHolder 的数据信息会被重置掉，相当于 ViewHolder 是一个重创新建的一样，所以需要重新调用 onBindViewHolder 来绑定数据。

mViewCacheExtension：这个是留给我们自己扩展的，好像也没怎么用，就暂时不分析了。

- **总结**

RecyclerView 滑动场景下的回收复用涉及到的结构体两个：mCachedViews 和 RecyclerViewPool。

mCachedViews 优先级高于 RecyclerViewPool，回收时，最新的 ViewHolder 都是往 mCachedViews 里放，如果它满了，那就移出一个扔到 ViewPool 里好空出位置来缓存最新的 ViewHolder。

复用时，也是先到 mCachedViews 里找 ViewHolder，但需要各种匹配条件，概括一下就是只有原来位置的卡位可以复用存在 mCachedViews 里的 ViewHolder，如果 mCachedViews 里没有，那么才去 ViewPool 里找。

在 ViewPool 里的 ViewHolder 都是跟全新的 ViewHolder 一样，只要 type 一样，有找到，就可以拿出来复用，重新绑定下数据即可。

- **三问三答**

Q1: 如果向下滑动，新一行的 5 个卡位的显示会去复用缓存的 ViewHolder，第一行的 5 个卡位会移出屏幕被回收，那么在这个过程中，是先进行复用再回收？还是先回收再复用？还是边回收边复用？也就是说，新一行的 5 个卡位复用的 ViewHolder 有可能是第一行被回收的 5 个卡位吗?

答：先复用再回收，新一行的 5 个卡位先去目前的 mCachedViews 和 ViewPool 的缓存中寻找复用，没有就重新创建，然后移出屏幕的那行的 5 个卡位再回收缓存到 mCachedViews 和 ViewPool 里面，所以新一行 5 个卡位和复用不可能会用到刚移出屏幕的 5 个卡位。

Q2: 在这个过程中，为什么当 RecyclerView 再次向上滑动重新显示第一行的 5 个卡位时，只有后面 3 个卡位触发了 onBindViewHolder() 方法，重新绑定数据呢？明明 5 个卡位都是复用的。

答：滑动场景下涉及到的回收和复用的结构体是 mCachedViews 和 ViewPool，前者默认大小为 2，后者为 5。所以，当第三行显示出来后，第一行的 5 个卡位被回收，回收时先缓存在 mCachedViews，满了再移出旧的到 ViewPool 里，所有 5 个卡位有 2 个缓存在 mCachedViews 里，3 个缓存在 ViewPool，至于是哪 2 个缓存在 mCachedViews，这是由 LayoutManager 控制。上面讲解的例子使用的是 GridLayoutManager，滑动时的回收逻辑则是在父类 LinearLayoutManager 里实现，回收第一行卡位时是从后往前回收，所以最新的两个卡位是 0、1，会放在 mCachedViews 里，而 2、3、4 的卡位则放在 ViewPool 里。

所以，当再次向上滑动时，第一行 5 个卡位会去两个结构体里找复用，之前说过，mCachedViews 里存放的 ViewHolder 只有原本位置的卡位才能复用，所以 0、1 两个卡位都可以直接去 mCachedViews 里拿 ViewHolder 复用，而且这里的 ViewHolder 是不用重新绑定数据的，至于 2、3、4 卡位则去 ViewPool 里找，刚好 ViewPool 里缓存着 3 个 ViewHolder，所以第一行的 5 个卡位都是用的复用的，而从 ViewPool 里拿的复用需要重新绑定数据，才会这样只有三个卡位需要重新绑定数据。

Q3：接下去不管是向上滑动还是向下滑动，滑动几次，都不会再有 onCreateViewHolder() 的日志了，也就是说 RecyclerView 总共创建了 17 个 ViewHolder，但有时一行的 5 个卡位只有 3 个卡位需要重新绑定数据，有时却又 5 个卡位都需要重新绑定数据，这是为什么呢?

答：有时一行只有 3 个卡位需要重新绑定的原因跟 Q2 一样，因为 mCachedView 里正好缓存着当前位置的 ViewHolder，本来就是它的 ViewHolder 当然可以直接拿来用。而至于为什么会创建了 17 个 ViewHolder，那是因为再第四行的卡位要显示出来时，ViewPool 里只有 3 个缓存，而第四行的卡位又用不了 mCachedViews 里的 2 个缓存，因为这两个缓存的是 6、7 卡位的 ViewHolder，所以就需要再重新创建 2 个 ViewHodler 来给第四行最后的两个卡位使用。

59、 Android 自定义 view

- **构造函数**

```
1    public SketchView(Context context) { // 构造函数 1
2        super(context);
3    }
4
5    public SketchView(Context context, AttributeSet attrs) { // 构造函数 2
6        super(context, attrs);
```

```
    }

    public SketchView(Context context, AttributeSet attrs, int defStyleAttr) { // 构造函数 3
        super(context, attrs, defStyleAttr);
    }
```

构造函数 1：

在 Java 代码中新建 View 时使用。

构造函数 2：

在 xml 布局文件中使用需要改构造函数，其中诸如宽高以及 margin 等布局属性都会存放在 AttributeSet 参数里，用于自定属性的时候使用。

构造函数 3：

比构造函数 2 多了一个 defStyleAttr 参数，defStyleAttr 指定的是在 Theme Style 中定义的一个 attr，其类型是 reference，主要在 obtainStyledAttributes() 方法中。

- **测量：onMeasure()**

```
        @Override
        protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
            int widthMode = MeasureSpec.getMode(widthMeasureSpec);
            int widthSize = MeasureSpec.getSize(widthMeasureSpec);

            int heightMode = MeasureSpec.getMode(heightMeasureSpec);
            int heightSize = MeasureSpec.getSize(heightMeasureSpec);

            int measuredHeight, measuredWidth;

            if (widthMode == MeasureSpec.EXACTLY) {
                measuredWidth = widthSize;
            } else {
                measuredWidth = SIZE;
            }

            if (heightMode == MeasureSpec.EXACTLY) {
                measuredHeight = heightSize;
            } else {
                measuredHeight = SIZE;
            }

            setMeasuredDimension(measuredWidth, measuredHeight);
        }
```

MeasureSpec 是一个 32 位的 int，高 2 位代表 SpecMode，低 30 位代表 SpecSize。SpecMode 表示测量模式，SpecSize 指在某种测量模式下规格的大小。

specMode 有三种状态：UNSPECIFIED，EXACTLY（相当于 matchparent 和精确值这两种模式），ATMOST（wrap_content）。

如果父是 EXACTLY，parentSize，那么子如果是：

- 具体的值 size：那子的 MeasureSpec 就是 EXACTLY,size；
- MATCH_PARENT：那子的 MeasureSpec 就是 EXACTLY,parentSize；
- WRAPCONTENT：那子的 MeasureSpec 就是 ATMOST,parentSize。

如果父是 ATMOST，parentSize，那么子如果是：

- 具体的值 size：那子的 MeasureSpec 就是 EXACTLY,size；
- MATCHPARENT：那子的 MeasureSpec 就是 ATMOST,parentSize；
- WRAPCONTENT：那子的 MeasureSpec 就是 ATMOST,parentSize。

总结：对于普通 View 的 MeasureSpec 由父容器的 MeasureSpec 和自身的 LayoutParams 来共同决定。

对于普通 View 的 measure 来说，是由父 View 也就是 ViewGroup 的 measureChildWithMargins 方法来触发的。ViewGroup 的 onMeasure 方法是交给子类自己实现的。不同的 viewgroup 子类肯定布局都不一样，那 onMeasure 索性就全部交给他们自己实现好了。

- **布局：onLayout()**

```
protected void onLayout(boolean changed, int l , int t, int r, int b) {
    if (mOrientation == VERTICAL) {
        layoutVertical(l, t, r , b);
    } else {
        layoutHorizontal(l, t, r , b);
    }
}
```

onLayout 方法里主要是具体摆放子 view 的位置，水平摆放或者垂直摆放，所以在单纯的自定义 view 是不需要重写 onLayout 方法，不过需要注意的一点是，**子 view 的 margin 属性是否生效就要看 parent 是否在自身的 onLayout 方法进行处理，而 view 的 padding 属性是在 onDraw 方法中生效的。**

layout 是确定本身 view 的位置，而 onLayout 是确定所有子元素的位置。layout 里面就是通过 serFrame 方法设定本身 view 的四个顶点的位置。这 4 个位置一确定， view 的位置就固定了，然后就调用 onLayout 来确定子元素的位置。view 和 viewgroup 的 onlayout 方法都没有写，都留给我们自己给子元素布局。

- **绘制：onDraw()**

步骤：

1) 绘制背景，(canvas)
2) 保存画布当前状态为绘制 fading edges 做准备
3) 绘制自己。(onDraw)
4) 绘制 children(dispatchDraw)
5) 绘制 fading edges 并还原画布状态
6) 绘制装饰（onDrawScrollBars）

如果是 view，draw 绘制其自身。如果是 VIewGroup，draw 绘制自身外，还要绘制其子元素。

viewgroup 中的 dispatchDraw 用于遍历子 view 并调用子 view 的 draw 方法。自定义属性

要给 view 支持自定义属性，需要在 values/attrs.xml 文件里定义一个 name 为自己定义 view 名字的 declare-styleable。

```
<resources>
    <declare-styleable name="SketchView">
        <attr name="background_color" format="color"/>
        <attr name="size" format="dimension"/>
    </declare-styleable>
</resources>
```

```
public SketchView(Context context, AttributeSet attrs, int defStyleAttr) {
      super(context, attrs, defStyleAttr);
      TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.SketchView,
                                                    defStyleAttr, R.style.AppTheme);
      custom_size = a.getDimensionPixelSize(R.styleable.SketchView_size, SIZE);
      custon_background = a.getColor(R.styleable.SketchView_background_color, DEFAULT_COLOR);
      a.recycle();
}
```

- **注意事项**

1) 如果是继承 view 或者 viewGroup，让 view 支持 wrap_content。
2) 如果有必要，让 view 支持 padding。
3) View 中如果有动画或者线程，要在 onDetachedFromWindow 中及时停止。当 view 的 Activity 退出或者当前 view 被 remove 时，调用它。

60、 Activity 中获取 View 的宽高

因为 activity 的生命周期和 view 不同步，在 oncreate 中无法获取到 view 的宽高，获取宽高的方法：

1) onWindowFocusChanged：view 已经初始化完毕，宽高已经准备好。当 Activity 得到焦点和失去焦点均会被调用，所以它会调用多次。
2) 通过 view.post，将一个 runnable 投递到消息队列尾部，等待 looper 调用时，view 已经初始化好。

```
protected void onStart(){
    super.onStart();
```

```
        view.post(new Runnable(){
            public void run(){
                int width = view.getMeasuredWidth();
                int height = view.getMeasuredHeight();
            }
        })
    }
```

3) ViewTreeObserver：使用 ViewTreeObserver 众多回调接口来完成，如 OnGlobalLayoutListener，当 view 树状态发生改变时或内部 view 可见性发生改变时会回调。

```
    ViewTreeObserver obserber = view.getViewTreeObserver();
    obserber.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener(){
        public void onGlobalLayout(){
            if(observer.isAlive()){ //判断 ViewTreeObserver 是否存活，如果存活的话移除这个观察者
                obserber.removeOnGlobalLayoutListener(this);
                int width = view.getMeasuredWidth();
                int height = view.getMeasuredHeight();
            }
        }
    })
```

4) 通过 view 进行 measure 来得到 view 的宽高。

```
    int width = MeasureSpec.makeMeasureSpec(100,Measure.EXACTLY);//确定值
    int height= MeasureSpec.makeMeasureSpec(100,Measure.EXACTLY);//确定值
    view.measure(width,height);
    对于 wrap_content:
    int width = MeasureSpec.makeMeasureSpec((1<<30)-1,Measure.AT_MOST);//wrap_content
    int height= MeasureSpec.makeMeasureSpec((1<<30)-1,Measure.AT_MOST);//wrap_content
    view.measure(width,height);
```

61、 刷新 view 的方法

三个方法：requestLayout()、invalidate()、postInvalidate()，其实 invalidate 和 postInvalidate 这两个方法作用是一样的，唯一不同的是 invalidate 用在主线程，而 postInvalidate 用在异步线程。

```
    @override
    public void requestLayout() {
        if (!mHandlingLayoutInLayoutRequest) {
            checkThread();
            mLayoutRequested = true;
            scheduleTraversals();
        }
    }
```

```
    void invalidate() {
        mDirty.set(0, 0, mWidth, mHeight);
        if (!mWillDrawSoon) {
            scheduleTraversals();
        }
    }
```

- requestLayout 会调用 measure 和 layout 等一系列操作，然后根据布局是否发生改变，surface 是否被销毁，来决定是否调用 draw，也就是说 requestlayout 肯定会调用 measure 和 layout，但不一定调用 draw。
- invalidate 只会调用 draw，而且肯定会调，即使什么都没有发生改变，它也会重新绘制。

62、 自定义 View 状态的保存

和 Activity 的状态保存类似，都是在 OnSaveInstanceState 保存，在 OnRestoreInstanceState 将数据安全取出。

63、 setWillNotDraw 方法的作用

该方法在 view 里，用于设置标志位。如果你的自定义 view 不需要 draw 的话，就可以设置这个方法为 true。这样系统知道你这

个 view 不需要 draw 可以优化执行速度。viewgroup 一般都默认设置这个为 true，因为 viewgroup 多数都是只负责布局，不负责 draw 的。而 view 这个标志位 默认一般都是关闭的。

64、 View 中 getWidth() 和 getMeasuredWidth() 的区别

getMeasuredWidth()：只要一执行完 setMeasuredDimension() 方法，就有值了，并且不再改变。

getWidth()：必须执行完 onMeasure() 才有值，可能发生改变。 如果 onLayout 没有对子 View 实际显示的宽高进行修改，那么 getWidth() 的值 等于 getMeasuredWidth() 的值。

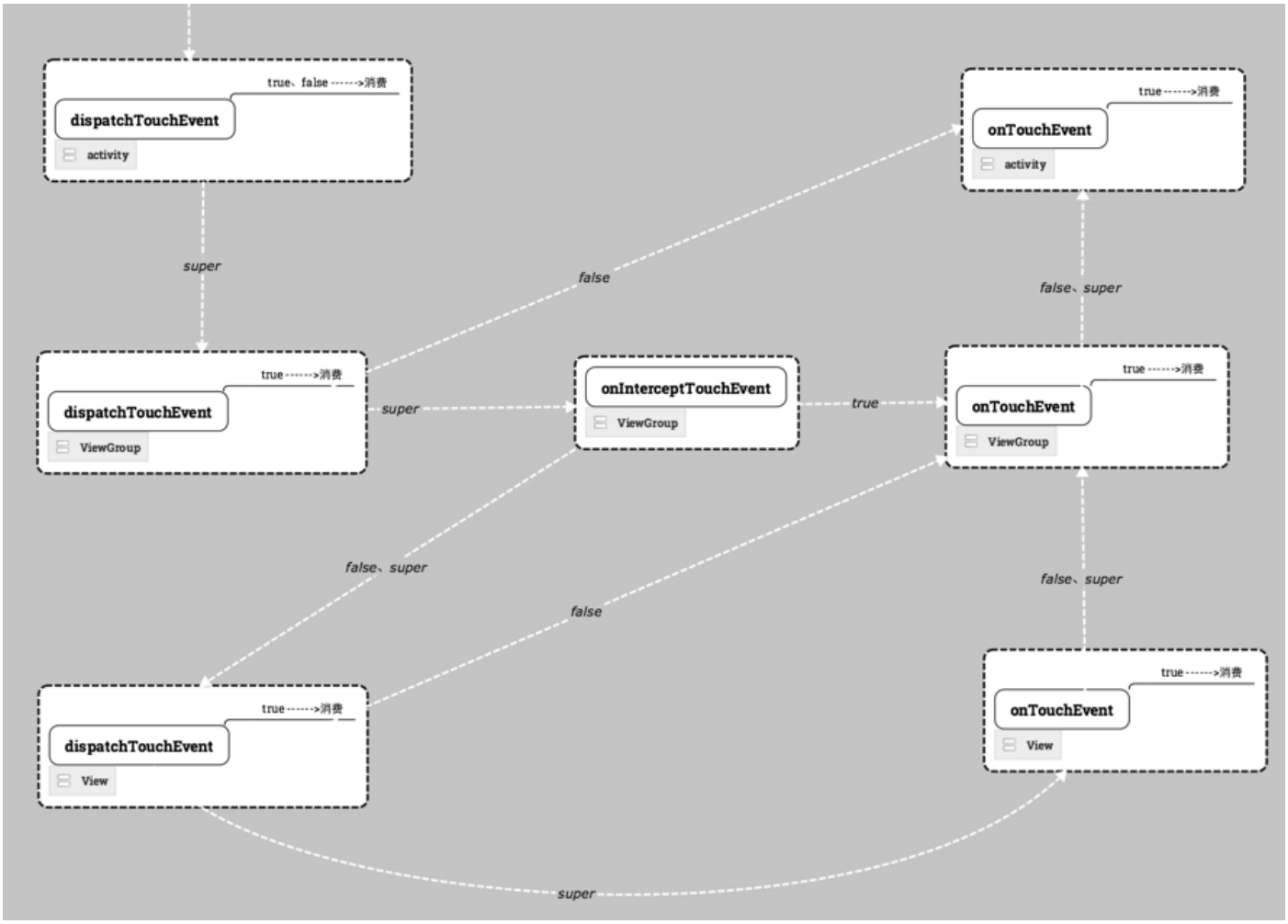
65、 加载高清大图

BitmapRegionDecoder：用于显示图片的某一块矩形区域。

```
1    // 获得图片的宽、高
2    BitmapFactory.Options tmpOptions = new BitmapFactory.Options();
3    tmpOptions.inJustDecodeBounds = true; // 只读取图片，不加载到内存中
4    BitmapFactory.decodeStream(inputStream, null, tmpOptions);
5    int width = tmpOptions.outWidth;
6    int height = tmpOptions.outHeight;
7    // 设置显示图片的中心区域
8    BitmapRegionDecoder bitmapRegionDecoder = BitmapRegionDecoder.newInstance(inputStream, false);
9    BitmapFactory.Options options = new BitmapFactory.Options();
10   options.inPreferredConfig = Bitmap.Config.RGB_565;
11   Bitmap bitmap = bitmapRegionDecoder.decodeRegion(
12         new Rect(width / 2 - 100, height / 2 - 100, width / 2 + 100, height / 2 + 100), options);
13   mImageView.setImageBitmap(bitmap);
```

使用 BitmapRegionDecoder 去加载 assets 中的图片，调用 bitmapRegionDecoder.decodeRegion 解析图片的中间矩形区域，返回 bitmap，最终显示在 ImageView 上。

66、 Android 事件分发机制

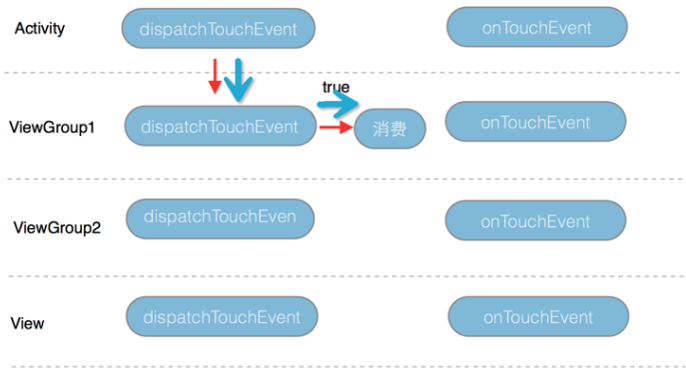


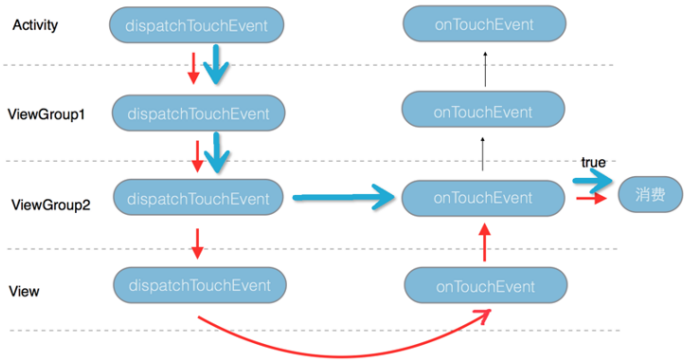
事件分发规律：

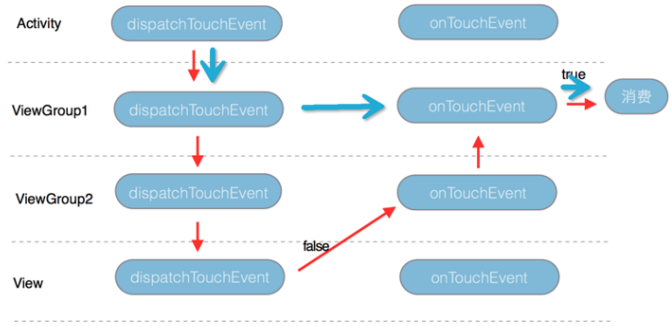
- 对于 dispatchTouchEvent，onTouchEvent，return true 是终结事件传递。return false 是回溯到父 View 的 onTouchEvent 方法。
- ViewGroup 想把自己分发给自己的 onTouchEvent，需要拦截器 onInterceptTouchEvent 方法 return true 把事件拦截下来。
- ViewGroup 的拦截器 onInterceptTouchEvent 默认是不拦截的，所以 return super.onInterceptTouchEvent()=return false；
- View 没有拦截器，为了让 View 可以把事件分发给自己的 onTouchEvent，View 的 dispatchTouchEvent 默认实现（super）就是把事件分发给自己的 onTouchEvent。

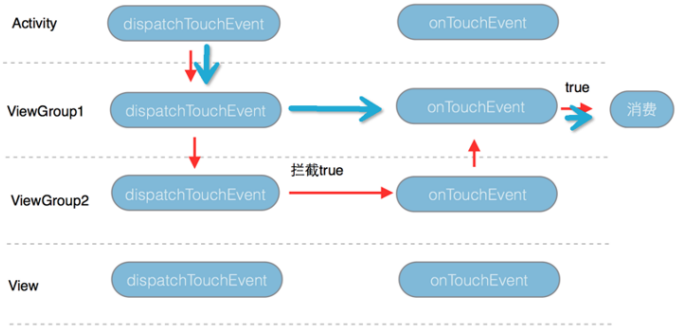
ACTION_MOVE、ACTION_UP 总结：

ACTION_DOWN 事件在哪个控件消费了（return true），那么 ACTION_MOVE 和 ACTION_UP 就会从上往下（通过 dispatchTouchEvent）做事件分发往下传，就只会传到这个控件，不会继续往下传，如果 ACTION_DOWN 事件是在 dispatchTouchEvent 消费，那么事件到此为止停止传递，如果 ACTION_DOWN 事件是在 onTouchEvent 消费的，那么会把 ACTION_MOVE 或 ACTION_UP 事件传给该控件的 onTouchEvent 处理并结束传递。









67、 Android 中有哪几种类型的动画 ？

3.0 以前，android 支持两种动画模式，tween animation（补间动画 view animation），frame animation（帧动画 drawable animation），在 android3.0 中又引入了一个新的动画系统：property animation（属性动画）。

- View Animation

补间动画，给出两个关键帧，通过一些算法将给定的属性值在给定的时间内通过两个关键帧间进行渐变。

View animation 只能应用于 View 对象，只支持一部分属性，如缩放（scale）、旋转（rotate）、平移（translate）、渐变（alpha）。它只是改变了 View 对象绘制的位置，而没有改变 View 对象本身。如：平移之后原先 View 的位置仍可点击。

XML 方式：

用 XML 定义的动画放在**/res/anim/**文件夹内，XML 文件的根元素可以为<alpha>,<scale>,<translate>,<rotate>,interpolator 元素或<set>（表示以上几个动画的集合，set 可以嵌套）。默认情况下，所有动画是同时进行的，可以通过 **startOffset** 属性设置各个动画的开始偏移（开始时间）来达到动画顺序播放的效果。

```
1    <?xml version="1.0" encoding="utf-8"?>
2    <translate xmlns:android="http://schemas.android.com/apk/res/android"
3        android:fromXDelta="0"
```

```
    android:toXDelta="-80"
    android:fromYDelta="0"
    android:toYDelta="-80"
    android:duration="2000"
    android:fillBefore="true">
</translate>
```

```
ImageView spaceshipImage = (ImageView)findViewById(R.id.spaceshipImage);
Animation hyperspaceJumpAnimation = AnimationUtils.loadAnimation(this, R.anim.hyperspace_jump);
spaceshipImage.startAnimation(hyperspaceJumpAnimation);
```

JAVA 方式：

```
translateAnim = new TranslateAnimation(Animation.ABSOLUTE, 0, Animation.ABSOLUTE, -80,
          Animation.ABSOLUTE, 0, Animation.ABSOLUTE, -80);
translateAnim.setDuration(2000);
translateAnim.setFillBefore(true);
ImageView spaceshipImage = (ImageView)findViewById(R.id.spaceshipImage);
spaceshipImage.startAnimation(translateAnim);
```

- **Drawable Animation**

帧动画，就像 GIF 图片，通过一系列 Drawable 依次显示来模拟动画的效果。

使用方式：

必须以<animation-list>为根元素，以<item>表示要轮换显示的图片，duration 属性表示各项显示的时间。XML 文件要放在/res/drawable/目录下。

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

```
imageView = (ImageView) findViewById(R.id.imageView1);
imageView.setBackgroundResource(R.drawable.drawable_anim);
anim = (AnimationDrawable) imageView.getBackground();
anim.start();
```

- **Property Animation**

属性动画，更改对象的实际属性。Property Animation 不止可以应用于 View，还可以应用于任何对象。Property Animation 只是表示一个值在一段时间内的改变，当值改变时要做什么事情完全是你自己决定的。Property Animator 包括 ValueAnimator 和 ObjectAnimator。

- **ValueAnimator**

ValueAnimator 包含 Property Animation 动画的所有核心功能，如动画时间，开始、结束属性值，相应时间属性值计算方法等。ValueAnimator 需要实现 ValueAnimator.onUpdateListener 接口，根据属性值执行相应的动作，这个接口只有一个函数 onAnimationUpdate()，最终通过回调中 ValueAnimator 对象的 getAnimatedValue()函数可以得到当前的属性值。

```
ValueAnimator animation = ValueAnimator.ofFloat(0f, 1f);
animation.setDuration(1000);
animation.addUpdateListener(new AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
        Log.i("update", ((Float) animation.getAnimatedValue()).toString());
    }
});
animation.setInterpolator(new CycleInterpolator(3));
animation.start();
```

监听动画变化时四个状态：

```
public static interface AnimatorListener {
    void onAnimationStart(Animator animation);
    void onAnimationEnd(Animator animation);
    void onAnimationCancel(Animator animation);
    void onAnimationRepeat(Animator animation);
}
```

- **ObjectAnimator**

继承自 ValueAnimator，要指定一个对象及该对象的一个属性，当属性值计算完成时自动设置为该对象的相应属性，不需要像 ValueAnimator 一样根据属性值再做操作。ObjectAnimator 的限制：

1) 对象应该有一个 setter 函数：set<PropertyName>（驼峰命名法）。
2) 像 ofFloat 之类的工厂方法，第一个参数为对象名，第二个为属性名，后面的参数为可变参数，如果 values 参数只设置了一个值的话，那么会假定为目的值，属性值的变化范围为当前值到目的值，为了获得当前值，该对象要有相应属性的 getter 方法：get<PropertyName>。
3) 如果有 getter 方法，其应返回值类型应与相应的 setter 方法的参数类型一致

```
tv=(TextView)findViewById(R.id.textview1);
btn=(Button)findViewById(R.id.button1);
btn.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ObjectAnimator oa=ObjectAnimator.ofFloat(tv, "alpha", 0f, 1f);
        oa.setDuration(3000);
        oa.start();
    }
});
```

注：根据应用动画的对象或属性的不同，可能需要在 onAnimationUpdate 函数中调用 invalidate()函数刷新视图。

- **AnimationSet**

AnimationSet 组合动画，并可设置组中动画的时序关系，如同时播放，顺序播放等。

```
AnimatorSet bouncer = new AnimatorSet();
bouncer.play(anim1).before(anim2);
bouncer.play(anim2).with(anim3);
bouncer.play(anim2).with(anim4)
bouncer.play(anim5).after(amin2);
animatorSet.start();
```

- **TypeEvalutors**

根据属性的开始、结束值与 TimeInterpolation 计算出的因子计算出当前时间的属性值，android 提供了以下几个 evalutor：

1) IntEvaluator：属性的值类型为 int；
2) FloatEvaluator：属性的值类型为 float；
3) ArgbEvaluator：属性的值类型为十六进制颜色值；
4) TypeEvaluator：一个接口，可以通过实现该接口自定义 Evaluator。

自定义 FloatEvaluator：

```
public class FloatEvaluator implements TypeEvaluator {
    public Object evaluate(float fraction, Object startValue, Object endValue) {
        float startFloat = ((Number) startValue).floatValue();
        return startFloat + fraction * (((Number) endValue).floatValue() - startFloat);
    }
}
animation. setEvaluator(new FloatEvaluator());
```

- **TimeInterplator**

Time interplator 定义了属性值变化的方式，如线性均匀改变等。

AccelerateInterpolator	加速，开始时慢中间加速
DecelerateInterpolator	减速，开始时快然后减速

AccelerateDecelerateInterolator	先加速后减速，开始结束时慢，中间加速
AnticipateInterpolator	反向，先向相反方向改变一段再加速播放
AnticipateOvershootInterpolator	反向加回弹，先向相反方向改变，再加速播放，会超出目的值然后缓慢移动至目的值
BounceInterpolator	跳跃，快到目的值时值会跳跃，如目的值 100，后面的值可能依次为 85，77，70，80，90，100
Cyclelinterpolator	循环，动画循环一定次数，值的改变为一正弦函数：Math.sin(2 * mCycles * Math.PI * input)
LinearInterpolator	线性，线性均匀改变
OvershottInterpolator	回弹，最后超出目的值然后缓慢改变到目的值
TimeInterpolator	一个接口，允许你自定义 interpolator，以上几个都是实现了这个接口

```
1    animation. setInterpolator(new AccelerateInterpolator());
```

- **Keyframes**

keyFrame 是一个 时间/值 对，通过它可以定义一个在特定时间的特定状态，即关键帧，而且在两个 keyFrame 之间可以定义不同的 Interpolator，就好像多个动画的拼接，第一个动画的结束点是第二个动画的开始点。KeyFrame 是抽象类，要通过 ofInt()，ofFloat()，ofObject()获得适当的 KeyFrame，然后通过 PropertyValuesHolder.ofKeyframe 获得 PropertyValuesHolder 对象。

```
1    Keyframe kf0 = Keyframe.ofInt(0, 400);
2    Keyframe kf1 = Keyframe.ofInt(0.25f, 200);
3    Keyframe kf2 = Keyframe.ofInt(0.5f, 400);
4    Keyframe kf4 = Keyframe.ofInt(0.75f, 100);
5    Keyframe kf3 = Keyframe.ofInt(1f, 500);
6    PropertyValuesHolder pvhRotation = PropertyValuesHolder.ofKeyframe("width", kf0, kf1, kf2, kf4,
7    kf3);
8    ObjectAnimator rotationAnim = ObjectAnimator.ofPropertyValuesHolder(btn2, pvhRotation);
9    rotationAnim.setDuration(2000);
```

- **LayoutTransaction**

自定义 ViewGroup 子元素入场出场动画。

APPEARING	当一个元素在其父元素中变为 Visible 时对这个元素应用动画
CHANGE_APPEARING	当一个元素在其父元素中变为 Visible 时，因系统要重新布局有一些元素需要移动，对这些要移动的元素应用动画
DISAPPEARING	当一个元素在其父元素中变为 GONE 时对其应用动画
CHANGE_DISAPPEARING	当一个元素在其父元素中变为 GONE 时，因系统要重新布局有一些元素需要移动，这些要移动的元素应用动画

```
1    LayoutTransaction mTransitioner = new LayoutTransition();
2    ObjectAnimator animOut = ObjectAnimator.ofFloat(null, "rotation", 0f, 90f, 0f);
3    mTransitioner.setAnimator(LayoutTransition.DISAPPEARING, animOut);
4    linearLayout.setLayoutTransition(mTransitioner);
```

68、 帧动画在使用时需要注意什么 ？

1) OOM 问题：当图片数量较多较大时就容易出现 OOM，尽量避免使用帧动画。
2) 要在代码中调用 Imageview 的 setBackgroundResource 方法，如果直接在 XML 布局文件中设置其 src 属性当触发动画时会 FC（Force Close）。
3) 在动画 start()之前要先 stop()，不然在第一次动画之后会停在最后一帧，这样动画就只会触发一次。
4) 不要在 onCreate 中调用 start，因为 AnimationDrawable 还没有完全跟 Window 相关联，如果想要界面显示时就开始动画的话，可以在 onWindowFoucsChanged() 中调用 start()。

69、 UI 布局优化

1) 能使用 linearlayout 的，尽量不用 RelativeLayout。没有设置 LinearLayout 的 weight 属性的话， Linearlayout 代码只会执行 onMeasure 一次。RelativeLayout 的 onMeasure()会执行两次 child 的 measure()方法。
2) 尽可能减少布局的嵌套层级，如果需要 LinearLayout 结合其他的 layout 才能实现的布局，建议还是使用 RelativeLayout，因为 viewGroup 的嵌套使用，也会使减慢布局的绘制。
3) 不要设置不必要的背景，避免过度绘制。

4) <include>：对于复用性比较高的布局使用<include>方式，方便引入和管理。
   - 当 include 两个相同的布局时，可以在 include 时指定一个新的 id，用来区别，即用新的 id 将原来的指定的 id 冲掉。
   - 如果想要用<include>标签重写布局属性，为了其他的布局属性能生效，必须要同时重写 android:layoutheight 和 android:layoutwidth 属性。
5) <viewstub>：当一些界面很少情况下使用的时候可以使用<viewstub>进行懒加载，主要使用场景：ViewStub 加载完成后替换成其他的布局，之后一直显示。多次控制显示与隐藏的场景还是要通过 setVisibility()来设置。
   - ViewStub 引用布局时使用 layout 属性，取值@layout/xxxxx。
   - InflateId 表示给被引用的布局的 id，也就是被控制显示或隐藏的布局的 id。如果不写 InflateId ,如果需要的话也可以直接在被引用的布局中给出 id 属性。
   - inflate() 方法只能被调用一次，如果再次调用会报异常信息 ViewStub must have a non-null ViewGroup viewParent。
   - 如果想 控制/修改 被填充布局中的内容并重复显示被填充的 view，就用 try 将 viewStub.inflate() 以及修改内容的代码包裹起来，并在 catch 中 setVisibility。
   - 在 xml 中定义 ViewStub 节点时，内部不能包含其他节点，也就是说，ViewStub 是一个自闭合节点，如果一个布局或者 view 想通过 ViewStub 显示，只能定义在单独的 xml 文件中。
6) <merge>：使用 merge 能够减少视图的节点数,达到减少视图在绘制过程消耗的时间，达到提高 UI 性能的效果。merge 要作为一个布局文件的根节点。
   - <merge />只可以作为 xml layout 的根节点。
   - 用 FrameLayout 作为界面的根布局时，要用<merge>标签作为根节点，因为 View 树的 ContentView 本身就是个 FrameLayout。
   - 用 RelateLayout 或 Linearlayout 作为界面根布局时，界面中某些可复用的或逻辑独立的布局用<include>导入，<include>导入的布局可以考虑用<merge>作为根节点。<merge>根节点内的控件布局取决于<include>这个布局的父布局是哪个布局，并且由父布局控制<include>这个布局中控件的排放。
7) 使用 Hierarchy Viewer 工具检测，查看 View 布局。用该工具你能发现你的 Layout 性能的瓶颈。选择你的手机或者模拟器里运行的某个进程，然后通过该工具展示布局树。在每一个布局块里，信号灯代表了它的测量、布局和绘制的性能，帮助你分析潜在的问题。
8) 通过 Lint 工具的分析功能，能够检测出一些布局方面的不足，方便我们进行优化定位。
   - 使用 compound drawables —— 一个包含了 ImageView 与 TextView 的 LinearLayout 可以被当作一个 compound drawable 来处理。
   - 使用 merge 根框架 —— 如果 FramLayout 仅仅是一个纯粹的（没有设置背景，间距等）布局根元素，我们可以使用 merge 标签来当作根标签。
   - 无用的分支 —— 如果一个 layout 并没有任何子组件，那么可以被移除，这样可以提高效率。
   - 无用的父控件 —— 如果一个 layout 只有子控件，没有兄弟控件，并且不是一个 ScrollView 或者根节点，而且没有设置背景，那么我们可以移除这个父控件，直接把子控件提升为父控件。
   - 深层次的 layout —— 尽量减少内嵌的层级，考虑使用更多平级的组件 RelativeLayout or GridLayout 来提升布局性能，默认最大的深度是 10。

70、 ThreadLocal（在多线程环境下，如何防止自己的变量被其它线程篡改）

ThreadLocal（线程本地存储）为变量在每个线程中创建了一个副本，这样每个线程可以访问自己内部的副本变量，线程之间互不影响。其内部有点像 HashMap，可以保存"key : value"键值对，但是一个 ThreadLocal 只能保存一个。

ThreadLocal 类提供的几个方法：

```
1    public T get() { }
2    public void set(T value) { }
3    public void remove() { }
4    protected T initialValue() { }
```

get() 方法是用来获取 ThreadLocal 在当前线程中保存的变量副本；

set() 用来设置当前线程中变量的副本；

remove() 用来移除当前线程中变量的副本；

initialValue() 是一个 protected 方法，一般是用来在使用时进行重写的，它是一个延迟加载方法。

原理分析：

set(T value) 和 get() 方法的源码：

```
1     public void set(T value) {
2         Thread t = Thread.currentThread();
3         ThreadLocalMap map = getMap(t);
4         if (map != null)
5             map.set(this, value);
6         else
7             createMap(t, value);
8     }
9
10    public T get() {
11        Thread t = Thread.currentThread();
12        ThreadLocalMap map = getMap(t);
13        if (map != null) {
14            ThreadLocalMap.Entry e = map.getEntry(this);
15            if (e != null) {
16                @SuppressWarnings("unchecked")
17                T result = (T)e.value;
18                return result;
19            }
20        }
21        return setInitialValue();
22    }
23
24    ThreadLocalMap getMap(Thread t) {
25        return t.threadLocals;
26    }
```

每个线程中都有一个 ThreadLocalMap 数据结构，当执行 set 方法时，其值是保存在当前线程的 threadLocals 变量中，当执行 get 方法时，是从当前线程的 threadLocals 变量获取。
所以在线程 1 中 set 的值，对线程 2 来说是摸不到的，而且在线程 2 中重新 set 的话，也不会影响到线程 1 中的值，保证了线程之间不会相互干扰。

ThreadLoalMap

```
1     static class Entry extends WeakReference<ThreadLocal<?>> {
2         /** The value associated with this ThreadLocal. */
3         Object value;
4
5         Entry(ThreadLocal<?> k, Object v) {
6             super(k);
7             value = v;
8         }
9     }
```

在 ThreadLoalMap 中，初始化一个大小 16 的 Entry 数组，Entry 对象用来保存每一个 key-value 键值对，只不过这里的 key 永远都是 ThreadLocal 对象，通过 ThreadLocal 对象的 set 方法，把 ThreadLocal 对象自己当做 key，放进了 ThreadLoalMap 中。

hash 冲突

ThreadLoalMap 中插入一个 key-value 的实现：

```
1     public void set(T value) {
2         Thread t = Thread.currentThread();
3         ThreadLocalMap map = getMap(t);
4         if (map != null)
5             map.set(this, value);
6         else
7             createMap(t, value);
8     }
9
10    public T get() {
11        Thread t = Thread.currentThread();
12        ThreadLocalMap map = getMap(t);
13        if (map != null) {
```

```
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }

    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
```

每个 ThreadLocal 对象都有一个 hash 值 threadLocalHashCode，每初始化一个 ThreadLocal 对象，hash 值就增加一个固定的大小 0x61c88647。

在插入过程中，根据 ThreadLocal 对象的 hash 值，定位到 table 中的位置 i，过程如下：

1) 如果当前位置是空的，那么正好，就初始化一个 Entry 对象放在位置 i 上；

2) 不巧，位置 i 已经有 Entry 对象了，如果这个 Entry 对象的 key 正好是即将设置的 key，那么重新设置 Entry 中的 value；

3) 很不巧，位置 i 的 Entry 对象，和即将设置的 key 没关系，那么只能找下一个空位置；

这样的话，在 get 的时候，也会根据 ThreadLocal 对象的 hash 值，定位到 table 中的位置，然后判断该位置 Entry 对象中的 key 是否和 get 的 key 一致，如果不一致，就判断下一个位置

可以发现，set 和 get 如果冲突严重的话，效率很低，因为 ThreadLoalMap 是 Thread 的一个属性，所以即使在自己的代码中控制了设置的元素个数，但还是不能控制其它代码的行为。

避免内存泄露

当使用 ThreadLocal 保存一个 value 时，会在 ThreadLocalMap 中的数组插入一个 Entry 对象，按理说 key-value 都应该以强引用保存在 Entry 对象中，但在 ThreadLocalMap 的实现中，key 被保存到了 WeakReference 对象中。

ThreadLocal 在没有外部强引用时，发生 GC 时会被回收，继承了弱引用的 key 会被回收，而如果创建 ThreadLocal 的线程一直持续运行，那么这个 Entry 对象中的 value 就有可能一直得不到回收，发生内存泄露。

```
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
```

为了最小化减少内存泄露的可能性和影响，在 ThreadLocal 调用 get 和 set 的时候都会清除线程 Map 里所有 key 为 null 的 value。

如果调用 remove 方法，肯定会删除对应的 Entry 对象。

如果使用 ThreadLocal 的 set 方法之后，没有显示的调用 remove 方法，就有可能发生内存泄露，所以养成良好的编程习惯十分重要，使用完 ThreadLocal 之后，记得调用 remove 方法。

```
    ThreadLocal<String> localName = new ThreadLocal();
    try {
        localName.set("占小狼");
        // 其它业务逻辑
    } finally {
        localName.remove();
    }
```

71、 Handler 的原理

Looper：消息封装的载体，内部包含了 MessageQueue，负责从 MessageQueue 取出消息，然后交给 Handler 处理。

Handler：封装了消息的发送，也负责接收消。内部会跟 Looper 关联。

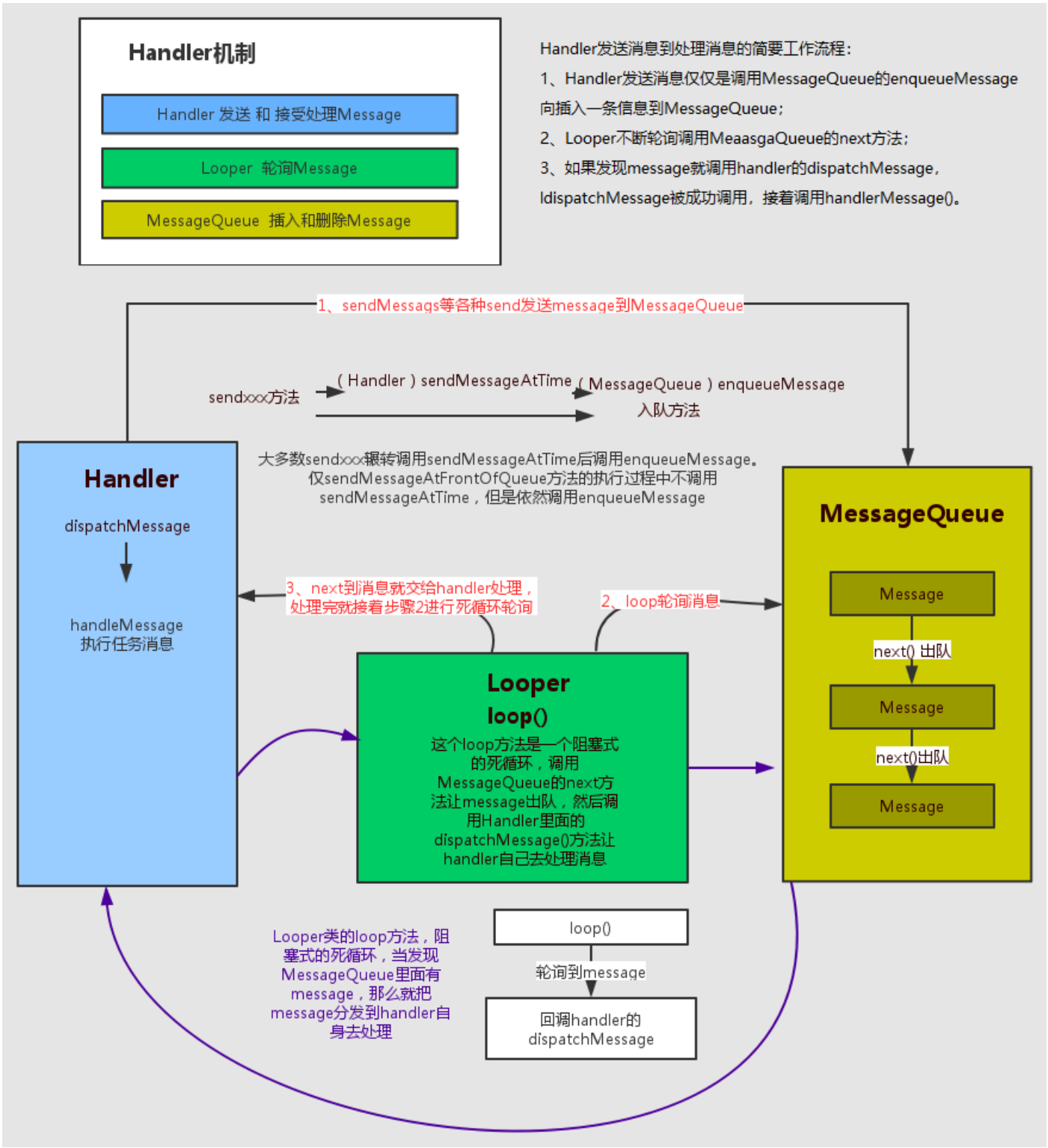
MessageQueue：消息队列，负责存储消息，有消息过来就存储起来，Looper 会循环的从 MessageQueue 读取消息。

Message：需要传递的信息。

简要流程：

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，不断从 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 相关联。
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。

注：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。



- **Looper**

prepare() 方法：

```
1    public static final void prepare() {
2         if (sThreadLocal.get() != null) {
3             throw new RuntimeException("Only one Looper may be created per thread");
4         }
5         sThreadLocal.set(new Looper(true));
6    }
```

sThreadLocal 是一个 ThreadLocal 对象，可以在一个线程中存储变量。源码中判断了 sThreadLocal 是否为 null，为 null 则将一个 Looper 的实例放入 ThreadLocal，否则抛出异常。这也就说明了 **Looper.prepare() 方法不能被调用两次**，同时也保证了一个线程中只有一个 Looper 实例。Looper 在构造方法中，创建了一个 MessageQueue（消息队列）。

loop() 方法：

```
1    public static void loop() {
2            final Looper me = myLooper();
3            if (me == null) {
4                throw new RuntimeException("No Looper; Looper.prepare()"+
5                                           "wasn't called on this thread.");
6            }
7            final MessageQueue queue = me.mQueue;
8            ……（内容省略）
9            for (;;) {
10               Message msg = queue.next(); // might block
11               if (msg == null) {
12                   // No message indicates that the message queue is quitting.
13                   return;
14               }
15               ……（内容省略）
16               msg.target.dispatchMessage(msg);
17               ……（内容省略）
18               msg.recycle();
19           }
20   }
```

myLooper() 返回 sThreadLocal 存储的 Looper 实例，如果 me 为 null 则抛出异常，也就是说 looper 方法必须在 prepare 方法之后运行。拿到该 looper 实例中的 mQueue（消息队列），进入死循环，有消息则取出交给 Message 的 target 对象（即 handler 对象）的 dispatchMessage 方法去处理，最后把分发后的 Message 回收到消息池，以便重复利用。

- **Handler**

无参构造函数：

```
1    public Handler() {
2        this(null, false);
3    }
4
5    public Handler(Callback callback, boolean async) {
6        //匿名类、内部类或本地类都必须申明为 static，否则会警告可能出现内存泄露
7        ……（内容省略）
8        //必须先执行 Looper.prepare()，才能获取 Looper 对象，否则为 null.
9        mLooper = Looper.myLooper();  //从当前线程的 TLS 中获取 Looper 对象
10       if (mLooper == null)  throw new RuntimeException("");
11       mQueue = mLooper.mQueue; //消息队列，来自 Looper 对象
12       mCallback = callback;  //回调方法
13       mAsynchronous = async; //设置消息是否为异步处理方式
14   }
```

有参构造函数：

多了个 Looper 参数，可指定 Looper。

消息分发机制：

```
1    public void dispatchMessage(Message msg) {
2        if (msg.callback != null) {
```

```
3             // 当 Message 存在回调方法，回调 msg.callback.run()方法；
4             handleCallback(msg);
5         } else {
6             if (mCallback != null) {
7                 // 当 Handler 存在 Callback 成员变量时，回调方法 handleMessage()；
8                 if (mCallback.handleMessage(msg))  return;
9             }
10            // Handler 自身的回调方法 handleMessage()
11            handleMessage(msg);
12        }
13    }
```

1) 当 Message 的回调方法不为空时，则回调方法 msg.callback.run()，其中 callBack 数据类型为 Runnable；

2) 当 Handler 的 mCallback 成员变量不为空时，则回调方法 mCallback.handleMessage(msg)；

3) 调用 Handler 自身的回调方法 handleMessage()，该方法默认为空，Handler 子类通过覆写该方法来完成具体的逻辑。

消息发送：

sendEmptyMessage

sendEmptyMessageDelayed

sendMessageDelayed

sendMessageAtTime

sendMessageAtFrontOfQueue

post

postAtFrontOfQueue

enqueueMessage

Handler.sendEmptyMessage() 等系列方法最终调用 MessageQueue.enqueueMessage(msg, uptimeMillis)，将消息添加到消息队列中，其中 uptimeMillis 为系统当前的运行时间，不包括休眠时间。

- **MessageQueue**

next() 方法：

```
1     Message next() {
2         ……（内容省略）
3         for (;;) {
4             // 阻塞操作，当等待 nextPollTimeoutMillis 时长，或者消息队列被唤醒，都会返回
5             nativePollOnce(ptr, nextPollTimeoutMillis);
6             ……（内容省略）
7                 if (msg != null) {
8                     ……（内容省略）
9                     msg = msg.next;
10                }
11            ……（内容省略）
12        }
13    }
```

当 nativePollOnce()返回后，next() 从 mMessages 中提取一个消息。

enqueueMessage 方法：

MessageQueue 是按照 Message 触发时间的先后顺序排列的，队头的消息是将要最早触发的消息。当有消息需要加入消息队列时，会从队列头开始遍历，直到找到消息应该插入的合适位置，以保证所有消息的时间顺序。

removeMessages 方法：

采用了两个 while 循环，第一个循环是从队头开始，移除符合条件的消息，第二个循环是从头部移除完连续的满足条件的消息之后，再从队列后面继续查询是否有满足条件的消息需要被移除。

- **Message**

recycle() 方法：

用于把消息回收到消息池，静态变量 sPool 的数据类型为 Message，通过 next 成员变量，维护一个消息池；静态变量 MAX_POOL_SIZE 代表消息池的可用大小；消息池的默认大小为 50。

obtain() 方法：

从消息池取 Message，都是把消息池表头的 Message 取走，再把表头指向 next。

72、 Linux 现有的所有进程间 IPC 方式

进程间通信（IPC，Inter-Process Communication），指至少两个进程或线程间传送数据或信号的一些技术或方法。

进程是计算机系统分配资源的最小单位(严格说来是线程)。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。

进程一般指的是一个执行单元，它拥有独立的地址空间，也就是一个应用或者一个程序。线程是 CPU 调度的最小单元，是进程中的一个执行部分或者说是执行体，两者之间是包含与被包含的关系。

为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。使用进程间通信的两个应用可以被分类为客户端和服务器（主从式架构），客户端进程请求数据，服务端回复客户端的数据请求。进程间通信技术包括消息传递、同步、共享内存和远程过程调用（Remote Procedure Call，缩写是 RPC）。IPC 是一种标准的 Unix 通信机制。

管道：在创建时分配一个 page 大小的内存，缓存区大小比较有限；

消息队列：信息复制两次，额外的 CPU 消耗；不合适频繁或信息量大的通信；

共享内存：无须复制，共享缓冲区直接付附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；

套接字：作为更通用的接口，传输效率低，主要用于不通机器或跨网络的通信；

信号量：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段；

信号: 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等。

73、 Binder 的优点

- **从性能的角度**

数据拷贝次数：Binder 数据拷贝只需要一次，而管道、消息队列、Socket 都需要 2 次，但共享内存方式一次内存拷贝都不需要；从性能角度看，Binder 性能仅次于共享内存。

- **从稳定性的角度**

Binder 是基于 C/S 架构的，Server 端与 Client 端相对独立，稳定性较好；而共享内存实现方式复杂，没有客户与服务端之别，需要充分考虑到访问临界资源的并发同步问题，否则可能会出现死锁等问题；从这稳定性角度看，Binder 架构优越于共享内存。

- **从安全的角度**

Android 作为一个开放式的平台，应用程序的来源广泛，因此确保终端的安全是非常重要的。Linux 传统的 IPC 没有任何安全措施，完全依赖上层协议来确保，具体有以下两点表现：第一，传统 IPC 的接收方无法获得对方可靠的 UID/PID（用户 ID/进程 ID），从而无法鉴别对方身份，使用传统 IPC 时只能由用户在数据包里填入 UID/PID，但这样不可靠，容易被恶意程序利用；第二，传统 IPC 的访问接入点是开放的，无法建立私有通信，只要知道这些接入点的程序都可以建立连接，这样无法阻止恶意程序通过猜测接收方的地址获得连接。

进程的 UID 是鉴别进程身份的重要标志，前面提到 C/S 架构，Android 系统中对外只暴露 Client 端，Client 端将任务发送给 Server 端，Server 端会根据权限控制策略，判断 UID/PID 是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用户选择是否运行。

74、 AIDL 原理和实例

为使应用程序之间能够彼此通信，Android 提供了 IPC （Inter Process Communication，进程间通信）的一种独特实现： AIDL （Android Interface Definition Language， Android 接口定义语言）。

AIDL 支持的数据类型：

1) 所有基础类型（int, char, 等）
2) String，List，Map，CharSequence 等类
3) 其他 AIDL 接口类型
4) 所有 Parcelable 的类

实例流程：

1) 创建工程（普通创建项目的方式）

新建两个 module：server（包名：com.android.server）和 client（包名：com.android.client）。

2) 创建 AIDL 文件：

打开名为 server 的 module，在项目/app/src/main/java 上右键，新建一个 AIDL 文件（依次选择 New->AIDL->AIDL File），取名为 IAdditionService，此时生成 app/src/main/aidl/IAdditionService.aidl 文件。如：

```
1    package com.android.server;
```

```
2
3     // Interface declaration （接口声明）
4     interface IAdditionService {
5         int add(in int value1, in int value2);
6     }
```

选择 Build->Make Project，会在 app/build/generated/source/aidl/debug/com/android/server 这个路径（debug 或者 release）里自动生成对应的 IAdditionService.java 这个文件。

```
1     package com.android.server;
2
3     public interface IAdditionService extends android.os.IInterface {
4         /**
5          * Local-side IPC implementation stub class.
6          */
7         public static abstract class Stub extends android.os.Binder
8                 implements com. android.server.IAdditionService {
9             ......
10        }
11
12        public int add(int value1, int value2) throws android.os.RemoteException;
13    }
```

上面的接口中有一个 Stub 类，Stub（桩）指用来替换一部分功能的程序段。桩程序可以用来模拟已有程序的行为（比如一个远端机器的过程）或是对将要开发的代码的一种临时替代。实现了远程服务的接口，以便你能使用它，就好像此服务是在本地一样。Stub 分析：

```
1     public static abstract class Stub extends android.os.Binder
2             implements com. android.server.IAdditionService {
3         private static final java.lang.String DESCRIPTOR = "com.android.server.IAdditionService";
4
5         /**
6          * Construct the stub at attach it to the interface.
7          */
8         public Stub() {
9             this.attachInterface(this, DESCRIPTOR);
10        }
11
12        /**
13         * Cast an IBinder object into an com.android.server.IAdditionService interface,
14         * generating a proxy if needed.
15         */
16        public static com.android.server.IAdditionService asInterface(android.os.IBinder obj) {
17            if ((obj == null)) {
18                return null;
19            }
20            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
21            if (((iin != null) && (iin instanceof com.android.server.IAdditionService))) {
22                return ((com.android.server.IAdditionService) iin);
23            }
24            return new com.android.server.IAdditionService.Stub.Proxy(obj);
25        }
26
27        @Override
28        public android.os.IBinder asBinder() {
29            return this;
30        }
31
32        @Override
33        public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)
34                    throws android.os.RemoteException {
35            java.lang.String descriptor = DESCRIPTOR;
36            switch (code) {
```

```
                case INTERFACE_TRANSACTION: {
                    reply.writeString(descriptor);
                    return true;
                }
                case TRANSACTION_add: {
                    data.enforceInterface(descriptor);
                    int _arg0;
                    _arg0 = data.readInt();
                    int _arg1;
                    _arg1 = data.readInt();
                    int _result = this.add(_arg0, _arg1);
                    reply.writeNoException();
                    reply.writeInt(_result);
                    return true;
                }
                default: {
                    return super.onTransact(code, data, reply, flags);
                }
            }
        }

        private static class Proxy implements com.android.server.IAdditionService {
            ......
        }

        static final int TRANSACTION_add = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    }
```

Stub 类中：

DESCRIPTOR 常量是 Binder 的唯一标识；

TRANSACTION_add 用来标识我们在接口中定义的方法的；

asInterface 方法用于将服务端的 Binder 对象转换为客户端所需要的接口对象，该过程区分进程，如果进程一样，就返回服务端 Stub 对象本身，否则呢就返回封装后的 Stub.Proxy 对象。

onTransact 方法是运行在服务端的 Binder 线程中的，当客户端发起远程请求后，在底层封装后会交由此方法来处理。通过 code 来区分客户端请求的方法，需要注意的是，如果该方法返回 false 的换，客户端的请求就会失败。一般可以用来做权限控制。

```
    private static class Proxy implements com.android.server.IAdditionService {
        private android.os.IBinder mRemote;

        Proxy(android.os.IBinder remote) {
            mRemote = remote;
        }

        @Override
        public android.os.IBinder asBinder() {
            return mRemote;
        }

        public java.lang.String getInterfaceDescriptor() {
            return DESCRIPTOR;
        }

        @Override
        public int add(int value1, int value2) throws android.os.RemoteException {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            int _result;
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                _data.writeInt(value1);
                _data.writeInt(value2);
                mRemote.transact(IAdditionService.Stub.TRANSACTION_add, _data, _reply, 0);
```

```
27                _reply.readException();
28                _result = _reply.readInt();
29            } finally {
30                _reply.recycle();
31                _data.recycle();
32            }
33            return _result;
34        }
35    }
```

Stub 的代理类 Proxy 中实现了 IAdditionService 接口中的具体方法，该方法 add() 运行在客户端。当客户端发起远程请求时，_data 会写入参数，然后调用 transact 方法发起 RPC（远程过程调用）请求，同时挂起当前线程，然后服务端的 onTransact 方法就会被调起，直到 RPC 过程返回后，当前线程继续执行，并从_reply 取出返回值（如果有的话），并返回结果。

注：定义 AIDL 文件只是方便 sutido 帮我们生成所需的 Binder 类，AIDL 并不是必须的文件，因为这个 Binder 类也可以手写出来。

3) 实现远程服务：

远程服务端代码，在 app/src/main/java/com/android/server 路径中新建一个 Service，叫 AdditionService.java，作为服务端。

```
1     package com.android.server;
2
3     public class AdditionService extends Service {
4
5         public AdditionService() {
6         }
7
8         @Override
9         public IBinder onBind(Intent intent) {
10            return new IAdditionService.Stub() {
11                /*
12                 * Implement com.android.server.IAdditionService.add(int, int)
13                 * 实现了 add 方法
14                 */
15                @Override
16                public int add(int value1, int value2) throws RemoteException {
17                    return value1 + value2;
18                }
19            };
20        }
21    }
```

为了实现服务，需要让这个类中的 onBind 方法返回一个 IBinder 类的对象。这个 IBinder 类的对象就代表了远程服务的实现。这里要用到自动生成的子类 IAdditionService.Stub。在其中必须实现之前在 AIDL 文件中定义的 add() 函数。

在 Manifest 文件中声明服务：

```
1     <service
2         android:name=".AdditionService"
3         android:exported="true"
4         android:process=":server">
5         <intent-filter>
6             <action android:name="com.lazyor.server.AdditionService" />
7         </intent-filter>
8     </service>
```

其中 process 用于指定当前服务所运行的进程，这里为 server。

4) “暴露”服务：

切换到名为 client 的 module，将服务端创建的 aidl 文件连同包名一起拷贝到 client/app/src/main/aidl/目录下。

服务端目录：server/app/src/main/aidl/**com/android/server/**IAdditionService.aidl

客户端目录：client/app/src/main/aidl/**com/android/server/**IAdditionService.aidl

注：使用 AIDL 进行客户端和服务端的通信需要满足一个条件，就是服务器端的各个 AIDL 文件须要被拷贝到客户端的相同包名下，不然会不成功。

在客户端的 Activity 下创建内部类 AdditionServiceConnection，其继承自 ServiceConnection。Xml 布局中包含两个 EditText（输入）、一个 Button（响应事件）和一个 TextView（结果）。

```
package com.lazyor.client;

public class MainActivity extends AppCompatActivity {

    IAdditionService additionService;
    AdditionServiceConnection additionServiceConnection;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initService();

        EditText etValue1 = findViewById(R.id.value1);
        EditText etValue2 = findViewById(R.id.value2);
        Button btnAdd = findViewById(R.id.add);
        TextView tvResult = findViewById(R.id.result);

        btnAdd.setOnClickListener(v -> {
            int v1 = Integer.parseInt(etValue1.getText().toString());
            int v2 = Integer.parseInt(etValue2.getText().toString());
            int result = 0;
            try {
                result = additionService.add(v1, v2);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
            tvResult.setText(Integer.valueOf(result).toString());
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        releaseService();

    }

    /** 内部类用于连接到服务 */
    class AdditionServiceConnection implements ServiceConnection {

        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            additionService = IAdditionService.Stub.asInterface(service);
            Toast.makeText(MainActivity.this, "Service connected", Toast.LENGTH_LONG).show();
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
            additionService = null;
            Toast.makeText(MainActivity.this, "Service disconnected", Toast.LENGTH_LONG).show();
        }
    }

    /** 连接服务 */
    private void initService() {
        additionServiceConnection = new AdditionServiceConnection();
        // 注意在 Android 5.0 以后，不能通过隐式 Intent 启动 service，必须制定包名
        Intent intent = new Intent();
        intent.setPackage("com.lazyor.server");
```

```
62            intent.setAction("com.lazyor.server.AdditionService");
63            bindService(intent, additionServiceConnection, Context.BIND_AUTO_CREATE);
64        }
65
66        /** 断开服务 */
67        private void releaseService() {
68            unbindService(additionServiceConnection);
69            additionServiceConnection = null;
70        }
71    }
```

之后先运行 Server 后运行 client。

75、 Android 内存泄漏

- **根搜索算法**

Android 虚拟机的垃圾回收采用的是根搜索算法。GC 会从根节点（GC Roots）开始对 heap 进行遍历。最后部分没有直接或者间接引用到 GC Roots 的就是需要回收的垃圾，会被 GC 回收掉。根搜索算法相比引用计数法更好地解决了循环引用的问题。举个例子，Activity 有 View 的引用，View 也有 Activity 的引用。当 Activity finish 掉之后，Activity 和 View 的循环引用已成孤岛，不再引用到 GC Roots，无需断开也会被回收掉。

- **内存泄漏**

进程中某些对象（垃圾对象）已经没有使用价值了，但是它们却可以直接或间接地引用到 gc roots 导致无法被 GC 回收。无用的对象占据着内存空间，使得实际可使用内存变小，形象地说法就是内存泄漏了。

- **场景**

- 单例造成的内存泄露

当传入的是 Activity 或者 Service 等短生命周期对象的 Context 时，单例持有对该对象 Context 的引用，导致该对象无法被正常回收。

```
1     public class AppManager {
2         private static AppManager instance;
3         private Context context;
4         private AppManager(Context context) {
5             this.context = context;
6         }
7         public static AppManager getInstance(Context context) {
8             if (instance != null) {
9                 instance = new AppManager(context);
10            }
11            return instance;
12        }
13    }
```

解决方案：

1) 将该属性的引用方式改为弱引用；
2) 如果传入 Context，使用 ApplicationContext。
3) 当 Activity 销毁的时候，在单例中移除掉对 Activity 的引用

- 非静态内部类和匿名类的静态实例

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

解决方案：

1) 将内部类变成静态内部类；
2) 如果有强引用 Activity 中的属性，则将该属性的引用方式改为弱引用；
3) 在业务允许的情况下，当 Activity 执行 onDestory 时，结束这些耗时任务。

- 类的静态变量持有大数据对象（如 Activity Context）

静态变量长期维持到大数据对象的引用，阻止垃圾回收。如某些静态对象持有对 Activity Context 引用，导致 Activity 关闭的时候无法正常回收。

解决方案：

1) 使用 Application Context 代替 Activity Context，因为 Application Context 会随着应用程序的存在而存在，而不依赖于 activity 的生命周期；
2) 对 Context 的引用不要超过它本身的生命周期，慎重的对 Context 使用“static”关键字。Context 里如果有线程，一定要在 onDestroy() 里及时停掉。

- 资源对象未关闭

资源对象如 Cursor、File、Socket，在使用后没有及时关闭。
解决方案：
1) 在使用结束资源对象后及时关闭，如在 finally 中关闭，避免在异常情况下资源对象未被释放。

- 注册对象未反注册

未反注册会导致观察者列表里维持着对象的引用，阻止垃圾回收。
解决方案：
1) 在 Activity 执行 onDestory 时，调用反注册。

- WebView 造成的内存泄露

WebView 本身存在内存泄漏的风险
解决方案：
1) 不要使用 WebView 对象时，应该调用它的 destory()函数来销毁它，并释放其占用的内存；
2) 为 webView 开启另外一个进程，通过 AIDL 与主线程进行通信，WebView 所在的进程可以根据业务的需要选择合适的时机进行销毁，从而达到内存的完整释放。

- Handler 临时性内存泄露

Handler 通过发送 Message 与主线程交互，Message 发出之后是存储在 MessageQueue 中的，有些 Message 并不是马上就被处理的。在 Message 中存在一个 target，是 Handler 的一个引用，如果 Message 在 Queue 中存在的时间越长，就会导致 Handler 无法被回收。如果 Handler 是非静态的，则会导致 Activity 或者 Service 不会被回收。
由于 AsyncTask 内部也是 Handler 机制，同样存在内存泄漏的风险。此种内存泄露，一般是临时性的。
解决方案：
1) 可以把 Handler 类放在单独的类文件中，或者使用静态内部类便可以避免泄露;
2) 如果想在 Handler 内部去调用所在的 Activity，那么可以在 handler 内部使用弱引用的方式去指向所在 Activity，使用 Static + WeakReference 的方式来达到断开 Handler 与 Activity 之间存在引用关系的目的。

- 集合中对象没清理造成的内存泄漏

当把一些对象的引用加入到集合容器（比如 ArrayList）中时，并没有把它的引用从集合中清理掉。如果这个集合是 static 的话，就会导致大量的对象无法回收。所以要在退出程序之前，将集合里的东西 clear，然后置为 null，再退出程序。
解决方案：
1) 在 Activity 退出之前，将集合里的东西 clear，然后置为 null，再退出程序。

- 构造 Adapter 时，没有使用缓存的 ConvertView

初始时 ListView 会从 Adapter 中根据当前的屏幕布局实例化一定数量的 View 对象，同时 ListView 会将这些 View 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 List Item 的 View 对象会被回收，然后被用来构造新出现的最下面的 List Item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View ConvertView 就是被缓存起来的 List Item 的 View 对象(初始化时缓存中没有 View 对象则 ConvertView 是 null)。

- **检测**

- 静态检测

依靠编辑器和 IDEA 来检查，比如 Android Studio 可以检测出 IO 或者 Socket 未关闭的情况。

- 动态监测

LeakCanary
Android Monitor
MAT 工具

76、 LruCache 与 DiskLruCache 缓存

- **LruCache**

采用近期最少使用算法，内部采用的是 LinkedHashMap 作存储，缓存在内存中。
构造函数：

```
1     public LruCache(int maxSize) {
2         if (maxSize <= 0) {
3             throw new IllegalArgumentException("maxSize <= 0");
4         }
5         this.maxSize = maxSize;
6         this.map = new LinkedHashMap<K, V>(0, 0.75f, true);
7     }
```

定义了缓存的最大值，并且调用了 LinkedHashMap 的三个参数的构造方法，保证按照访问顺序来排列元素，生成一个 LinkedHashMap 对象，赋值给 map。LinkedHashMap 的第三个参数 accessOrder：true 表示基于访问的顺序来排列，即最近访问的 Node 放置在链表的头部，false 表示按照插入的顺序来排列；

get 方法：

首先会根据 key 查找 map 中是否存在对应的 Value（即对应 key 值的缓存），如果找到，直接命中，返回此份缓存；如果没有找到，会调用 create（）方法去尝试创建一个 Value，这里的 create（）默认返回 null，要我们自己重写。但是正常情况下，不需要去重写 create 方法，因为一旦 get 不到缓存，就应该去网络请求数据并缓存起来。

put 方法：

计算新增加的大小，加入 size（size += safeSizeOf(key, value);），然后把 key-value 放入 map 中，如果是更新旧的数据（map.put(key, value) 会返回之前的 value），则还要减去旧数据的大小，并调用 entryRemoved(false, key, previous, value) 方法通知旧数据被更新为新的值，最后也是调用 trimToSize(maxSize) 修整缓存的大小。

sizeOf 方法：

使用 LRU 算法，说明我们需要设定缓存的最大大小，而缓存对象的大小在不同的缓存类型当中的计算方法是不同的，计算的方法通过 protected int sizeOf(K key, V value)实现，我们要缓存 Bitmap 对象，则需要重写这个方法，并返回 bitmap 对象的所有像素点所占的内存大小之和。

- **DiskLruCache**

采用近期最少使用算法，内部采用的是 LinkedHashMap 作存储，缓存在本地文件中。

open 方法：

```
1     public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)
```

创建 DiskLruCache 的实例，接收四个参数。

参数 1：数据的缓存地址；

缓存地址通常都会存放在 /sdcard/Android/data/<application package>/cache 这个路径下面，还需要考虑手机没有 SD 卡，或者 SD 正好被移除了的情况：

```
1     public File getDiskCacheDir(Context context, String uniqueName) {
2         String cachePath;
3         if (Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
4                 || !Environment.isExternalStorageRemovable()) {
5             cachePath = context.getExternalCacheDir().getPath();
6         } else {
7             cachePath = context.getCacheDir().getPath();
8         }
9         return new File(cachePath + File.separator + uniqueName);
10    }
```

当 SD 卡存在或者 SD 卡不可被移除的时候，就调用 getExternalCacheDir() 方法来获取缓存路径，否则就调用 getCacheDir() 方法来获取缓存路径。前者获取到的就是 /sdcard/Android/data/<application package>/cache 这个路径，而后者获取到的是 /data/data/<application package>/cache 这个路径。

参数 2：当前应用程序的版本号；

参数 3：指定同一个 key 可以对应多少个缓存文件，基本都是传 1；

参数 4：指定最多可以缓存多少字节的数据。

写入缓存：

DiskLruCache.Editor，借助 DiskLruCache 的 edit() 方法来获取实例。

```
1     public void setBitmapToLocal(Context context,String url, InputStream inputStream) {
2         BufferedOutputStream out = null;
```

```
        BufferedInputStream in = null;
        try {
            DiskLruCache.Editor editor = getDiskLruCache(context).edit(getMD5String(url));
            if (editor != null) {
                OutputStream outputStream = editor.newOutputStream(0);
                in = new BufferedInputStream(inputStream, 8 * 1024);
                out = new BufferedOutputStream(outputStream, 8 * 1024);
                int b;
                while ((b = in.read()) != -1) {
                    out.write(b);
                }
                editor.commit();
            }
            mDiskLruCache.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

读取缓存：

借助 DiskLruCache 的 get() 方法实现。

```
    public Bitmap getBitmapFromLocal(String url) {
        try {
            DiskLruCache.Snapshot snapShot = mDiskLruCache.get(getMD5String(url));
            if (snapShot != null) {
                InputStream is = snapShot.getInputStream(0);
                Bitmap bitmap = BitmapFactory.decodeStream(is);
                return bitmap;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

journal 文件：

DiskLruCache 自动生成的日志文件，主要记录缓存的操作。

open 方法初始化 DiskLruCache 对象时：

1) 首先判断是否有日志文件，如果有日志文件说明之前有缓存过信息，对上次的缓存信息做处理。通过日志文件头信息去判断之前缓存是否可用；
2) 解析之前缓存信息到 LinkedHashMap，对读到的上次缓存信息做处理，计算 size，把没有调用 Edit.commit() 的缓存剔除掉。

具体操作：

每当调用一次 DiskLruCache 的 edit() 方法时，都会向 journal 文件中写入一条 DIRTY 记录，表示正准备写入一条缓存数据，但写入结果未知。接着调用 commit() 方法写入缓存成功后，这时会向 journal 中写入一条 CLEAN 记录，意味着这条脏数据已经被写入成功，调用 abort() 方法表示写入缓存失败，这时会向 journal 中写入一条 REMOVE 记录。也就是说，每一行 DIRTY 的 key，后面都应该有一行对应的 CLEAN 或者 REMOVE 的记录，否则这条数据就是脏的，会被自动删除掉。

读日志文件：

1) 读日志文件的头部信息，标记，缓存版本，应用版本，进而判断日志文件是否过期
2) 把日志文件的缓存记录读取到 lruEntries，map 中

处理缓存信息

1) 计算整个缓存文件的大小
2) 把正在被编辑的 key（上次保存缓存的时候没有调用 Edit.commit()）,可以认为是没有写成功的缓存，重置掉（相应的缓存文件删除，并且从 lruEntries 中删除）

每条日志文件有四种情况：

CLEAN（调用了 edit（）之后，保存了缓存，并且调用了 Edit.commit（）了）、

DIRTY（缓存正在编辑，调用 edit（）函数)、
REMOVE（缓存写入失败)、
READ（读缓存)。

77、 ANR 解析

ANR：Application Not Responding（应用程序无响应）

基本原因：

APP 阻塞了 UI 线程。在 android 系统中每个 App 只有一个 UI 线程，是在 App 创建时默认生成的，UI 线程默认初始化了一个消息循环来处理 UI 消息，ANR 往往就是处理 UI 消息超时了。

1) View 的按键事件或者触摸事件在特定的时间（5 秒）内无法得到响应。
2) BroadcastReceiver 的 onReceive() 函数运行在主线程中，在特定的时间（10 秒）内无法完成处理。
3) Service 的各个生命周期函数在特定时间（20 秒）内无法完成处理。

具体原因：

1) 耗时的网络访问；
2) 大量的数据读写；
3) 数据库操作；
4) 硬件操作（比如 camera)；
5) 调用 thread 的 join() 方法、sleep() 方法、wait() 方法或者等待线程锁的时候；
6) service binder 的数量达到上限；
7) system server 中发生 WatchDog ANR；
8) service 处理耗时任务，导致超时无响应；
9) 其他线程持有锁，导致主线程等待超时；
10) 其它线程终止或崩溃导致主线程一直等待。

解决方法：

1) 避免在主线程执行耗时操作，所有耗时操作应新开一个子线程完成，然后再在主线程更新 UI。
2) 不要在 broadcastReciever 的 onRecieve() 方法中干活，BroadcastReceiver 要执行耗时操作时应启动一个 service，将耗时操作交给 service 来完成。
3) 避免在 Intent Receiver 里启动一个 Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。如果你的应用程序在响应 Intent 广播时需要向用户展示什么，你应该使用 Notification Manager 来实现。
4) 使用如 Systrace 和 Traceview 这样的性能工具分析 app 响应的瓶颈。

78、 SQLite 数据库优化

sqlite 每个数据库都是以单个文件的形式存在，这些数据都是以 B-Tree 的数据结构形式存储在磁盘上。同时 sqlite 更改数据的时候默认一条语句就是一个事务，有多少条数据就有多少次磁盘操作。

数据库性能上

1) 批量事务插入，提升数据插入的性能；
2) 执行单条 sql 优于多条 sql；
3) 读和写操作是互斥的，写操作过程中可以休眠让读操作进行；
4) 使用索引，让读取变快，缺点是数据插入修改的时间增加，因为需要维护索引的变化；
5) 使用联合索引，过多的索引同时也会减慢读取的速度，如：省市区县查询、年月日查询等这种有层级关系的查询，就可以使用联合索引；
6) 提前将字段的 index 映射好，在查询等时候，指定要查询的字段，减少 getColumnIndex 的时间，可以缩短一半的时间；
7) 增加查询条件，使用 limit，这样搜索到结果后，后面的数据就不会再查询了。

数据库设计上

1) 通过冗余换取查询速度；
2) 避免大数据多表的联合查询；
3) 减少数据来提升查询速度，如：下拉操作时，先清除旧数据，再插入新数据保证数据库中的数据总量小，提升查询速度。

79、 SQLite 的锁机制和 WAL 技术

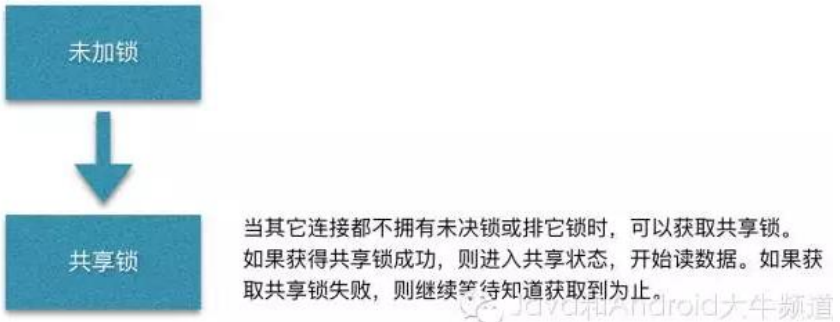
SQLite 基于锁来实现并发控制。SQLite 的锁是粗粒度的，**当一个连接要写数据库时，所有其它的连接都被锁住，**直到写连接结束

它的事务。

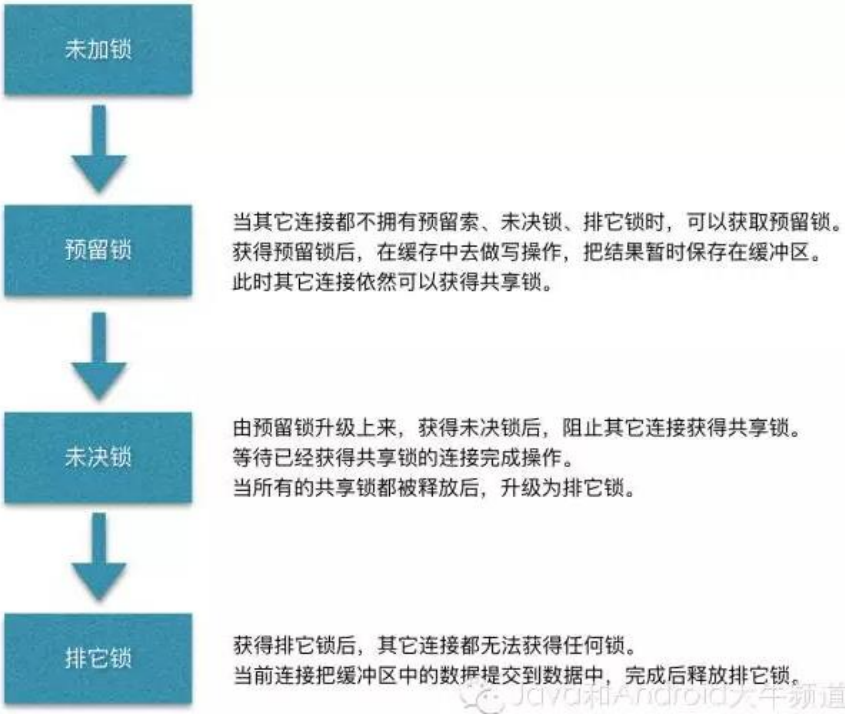
SQL 使用锁逐步提升机制，对应锁的等级逐步提升，等级越高权限就越大：

- **未加锁：**未和数据库建立连接、已建立连接但是还没访问数据库、已用 BEGIN 开始了一个事务但未开始读写数据库，处于这些情形时是未加锁状态。
- **共享：**连接需要从数据库中读取数据时，需要申请获得一个共享锁，如果获得成功，则进入共享状态。
- **预留：**连接需要写数据库时，首先申请一个预留锁，一个数据库同时只能有一个预留锁，预留锁可以与共享锁共存。获得预留锁后进入预留状态，这时会先在缓冲区中进行需要的修改、更新操作，操作后的结果依然保存在缓冲区中，未真正写入数据库。
- **未决：**连接从预留升为排它前，需要先升为未决，这时其它连接就不能获得共享锁了，但已经拥有共享锁的连接仍然可以继续正常读数据库，此时，拥有未决锁的连接等待其它拥有共享锁的连接完成工作并释放其共享锁后，提成到排它锁。
- **排它：**连接需要提交修改时，需要将预留锁升为排它锁，这时其它连接都无法获得任何锁，直到当前连接的排它状态结束。

一个连接读数据的流程：



一个连接写数据的流程：



死锁：

执行顺序	连接 A	连接 B
1	BEGIN;	
2		BEGIN;
3		INSERT INTO foo VALUES（‘x’）;
4	SELECT * FROM foo;	
5		COMMIT;

执行顺序	连接 A	连接 B
6		SQL error: database is locked
7	INSERT INTO foo VALUES（‘x’）;	
8	SQL error: database is locked	

执行顺序 3：连接 B 要执行写操作，获得预留锁

执行顺序 4：连接 A 要执行读操作，获得共享锁

执行顺序 5：连接 B 要提交修改，预留锁升级为未决锁

执行顺序 6：连接 B 想要升级为排它锁，必须先等待连接 A 释放共享锁

执行顺序 7：连接 A 要执行写操作，需要获得预留锁

执行顺序 8：连接 A 获得预留锁失败，必须先等待连接 B 释放未决锁

于是连接 A 和连接 B 相互等待对方，发生死锁。

可以通过正确的使用事务类型来解决以上死锁问题。

WAL 技术：Write Ahead Log（预写日志）

修改并不直接写入到数据库文件中，而是写入到另外一个称为 WAL 的文件中；如果事务失败，WAL 中的记录会被忽略，撤销修改；如果事务成功，它将在随后的某个时间被写回到数据库文件中，提交修改。

80、 热修复技术（链接）

- **QQ 空间超级补丁技术**

超级补丁技术基于 DEX 分包方案，使用了多 DEX 加载的原理，大致的过程就是：把 BUG 方法修复以后，放到一个单独的 DEX 里，插入到 dexElements 数组的最前面，让虚拟机去加载修复完后的方法。超级补丁技术为保证 ART 不出现地址错乱问题，需要将所有关联的类全部加入到补丁中。

当 patch.dex 中包含 Test.class 时就会优先加载，在后续的 DEX 中遇到 Test.class 的话就会直接返回而不去加载，这样就达到了修复的目的。

但是有一个问题是，当两个调用关系的类不在同一个 DEX 时，就会产生异常报错。在 APK 安装时，虚拟机需要将 classes.dex 优化成 odex 文件，然后才会执行。在这个过程中，会进行类的 verify 操作，如果调用关系的类都在同一个 DEX 中的话就会被打上 CLASS_ISPREVERIFIED 的标志，然后才会写入 odex 文件。

所以，为了可以正常的进行打补丁修复，必须避免类被打上 CLASS_ISPREVERIFIED 标志，具体的做法就是单独放一个类在另外 DEX 中，让其他类调用。

优势：

1) 没有合成整包（和微信 Tinker 比起来），产物比较小，比较灵活
2) 可以实现类替换，兼容性高。（某些三星手机不起作用）

不足：

1) 不支持即时生效，必须通过重启才能生效。
2) 为了实现修复这个过程，必须在应用中加入两个 dex！dalvikhack.dex 中只有一个类，对性能影响不大，但是对于 patch.dex 来说，修复的类到了一定数量，就需要花不少的时间加载。对手淘宝这种航母级应用来说，启动耗时增加 2s 以上是不能够接受的事。
3) 在 ART 模式下，如果类修改了结构，就会出现内存错乱的问题。为了解决这个问题，就必须把所有相关的调用类、父类子类等等全部加载到 patch.dex 中，导致补丁包异常的大，进一步增加应用启动加载的时候，耗时更加严重。

- **微信 Tinker**

微信针对 QQ 空间超级补丁技术的不足提出了一个提供 DEX 差量包，整体替换 DEX 的方案。主要的原理是与 QQ 空间超级补丁技术基本相同，区别在于不再将 patch.dex 增加到 elements 数组中，而是差量的方式给出 patch.dex，然后将 patch.dex 与应用的 classes.dex 合并，然后整体替换掉旧的 DEX，达到修复的目的。

优势：

1) 合成整包，不用在构造函数插入代码，防止 verify，verify 和 opt 在编译期间就已经完成，不会在运行期间进行
2) 性能提高。兼容性和稳定性比较高。
3) 开发者透明，不需要对包进行额外处理。

不足：

1) 与超级补丁技术一样，不支持即时生效，必须通过重启应用的方式才能生效。
2) 需要给应用开启新的进程才能进行合并，并且很容易因为内存消耗等原因合并失败。
3) 合并时占用额外磁盘空间，对于多 DEX 的应用来说，如果修改了多个 DEX 文件，就需要下发多个 patch.dex 与对应的 classes.dex 进行合并操作时这种情况会更严重，因此合并过程的失败率也会更高。

- **阿里百川 HotFix**

HotFix 热修复服务基于 AndFix 技术上，再易用性和安全性上做了优化，另外还提供了 patch 包加密解密校验、App 安全校验、版本控制管理、加载修复的安全性等服务。AndFix 不同于 QQ 空间超级补丁技术和微信 Tinker 通过增加或替换整个 DEX 的方案，提供了一种运行时在 Native 修改 Filed 指针的方式（hack），实现方法的替换，达到即时生效无需重启，对应用无性能消耗的目的。其定位于紧急 bug 修复的场景下。

优势：

1) BUG 修复的即时性
2) 补丁包同样采用差量技术，生成的 PATCH 体积小
3) 对应用无侵入，几乎无性能损耗

不足：

1) 不支持新增字段，以及修改<init>方法，也不支持对资源的替换。
2) 由于厂商的自定义 ROM，对少数机型暂不支持。

81、 Webview 解析

WebView 是一个基于 webkit 引擎、展现 web 页面的控件。

- **作用：**

1) 显示和渲染 Web 页面；
2) 直接使用 html 文件（网络上或本地 assets 中）作布局；
3) 可和 JavaScript 交互调用。

- **加载 url**

```
// 方式 1. 加载一个网页：
webView.loadUrl("http://www.google.com/");

// 方式 2：加载 apk 包中的 html 页面
webView.loadUrl("file:///android_asset/test.html");

// 方式 3：加载手机本地的 html 页面
webView.loadUrl("content://com.android.htmlfileprovider/sdcard/test.html");

// 方式 4： 加载 HTML 页面的一小段内容
WebView.loadData(String data, String mimeType, String encoding)
// 参数说明：
// 参数 1：需要截取展示的内容。内容里不能出现 ‘#’，‘%’，‘\’，‘?’ 这四个字符，若出现了需
用 %23, %25, %27, %3f 对应来替代，否则会出现异常
// 参数 2：展示内容的类型
// 参数 3：字节码
```

- **WebView 的状态**

```
// 激活 WebView 为活跃状态，能正常执行网页的响应
webView.onResume() ;

// 当页面被失去焦点被切换到后台不可见状态，需要执行 onPause
// 通过 onPause 动作通知内核暂停所有的动作，比如 DOM 的解析、plugin 的执行、JavaScript 执行。
webView.onPause();

// 当应用程序(存在 webview)被切换到后台时，这个方法不仅仅针对当前的 webview 而是全局的全应用程序的 webview
// 它会暂停所有 webview 的 layout，parsing，javascripttimer。降低 CPU 功耗。
webView.pauseTimers()
// 恢复 pauseTimers 状态
webView.resumeTimers();

// 销毁 Webview
```

```
// 在关闭了 Activity 时，如果 Webview 的音乐或视频，还在播放。就必须销毁 Webview
// 但是注意：webview 调用 destory 时,webview 仍绑定在 Activity 上
// 这是由于自定义 webview 构建时传入了该 Activity 的 context 对象
// 因此需要先从父容器中移除 webview,然后再销毁 webview:
rootLayout.removeView(webView);
webView.destroy();
```

- **关于前进 / 后退网页**

```
// 是否可以后退
Webview.canGoBack()
// 后退网页
Webview.goBack()

// 是否可以前进
Webview.canGoForward()
// 前进网页
Webview.goForward()

// 以当前的 index 为起始点前进或者后退到历史记录中指定的 steps
// 如果 steps 为负数则为后退，正数则为前进
Webview.goBackOrForward(intsteps)
```

- **常见用法：Back 键控制网页后退**

问题：在不做任何处理前提下 ，浏览网页时点击系统的"Back"键,整个 Browser 会调用 finish()而结束自身

目标：点击返回后，是网页回退而不是推出浏览器

解决方案：在当前 Activity 中处理并消费掉该 Back 事件

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if ((keyCode == KEYCODE_BACK) && mWebView.canGoBack()) {
        mWebView.goBack();
        return true;
    }
    return super.onKeyDown(keyCode, event);
}
```

- **清除缓存数据**

```
// 清除网页访问留下的缓存
// 由于内核缓存是全局的因此这个方法不仅仅针对 webview 而是针对整个应用程序.
Webview.clearCache(true);

// 清除当前 webview 访问的历史记录
// 只会 webview 访问历史记录里的所有记录除了当前访问记录
Webview.clearHistory();

// 这个 api 仅仅清除自动完成填充的表单数据，并不会清除 WebView 存储到本地的数据
Webview.clearFormData();
```

- **WebSettings 类**

作用：对 WebView 进行配置和管理

前提：添加访问网络权限（AndroidManifest.xml）

```
<uses-permission android:name="android.permission.INTERNET"/>
```

生成一个 WebView 组件：

```
//方式 1：直接在在 Activity 中生成
WebView webView = new WebView(this)

//方法 2：在 Activity 的 layout 文件里添加 webview 控件：
WebView webview = (WebView) findViewById(R.id.webView1);
```

WebSettings 常见方法：

```
//声明 WebSettings 子类
WebSettings webSettings = webView.getSettings();

//如果访问的页面中要与 Javascript 交互，则 webview 必须设置支持 Javascript
webSettings.setJavaScriptEnabled(true);
// 若加载的 html 里有 JS 在执行动画等操作，会造成资源浪费（CPU、电量）
// 在 onStop 和 onResume 里分别把 setJavaScriptEnabled() 给设置成 false 和 true 即可

//支持插件
webSettings.setPluginsEnabled(true);

//设置自适应屏幕，两者合用
webSettings.setUseWideViewPort(true); //将图片调整到适合 webview 的大小
webSettings.setLoadWithOverviewMode(true); // 缩放至屏幕的大小

//缩放操作
webSettings.setSupportZoom(true); //支持缩放，默认为 true。是下面那个的前提。
webSettings.setBuiltInZoomControls(true); //设置内置的缩放控件。若为 false，则该 WebView 不可缩放
webSettings.setDisplayZoomControls(false); //隐藏原生的缩放控件

//其他细节操作
webSettings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK); //关闭 webview 中缓存
webSettings.setAllowFileAccess(true); //设置可以访问文件
webSettings.setJavaScriptCanOpenWindowsAutomatically(true); //支持通过 JS 打开新窗口
webSettings.setLoadsImagesAutomatically(true); //支持自动加载图片
webSettings.setDefaultTextEncodingName("utf-8");//设置编码格式
```

设置 WebView 缓存:

当加载 html 页面时，WebView 会在/data/data/包名目录下生成 database 与 cache 两个文件夹。

请求的 URL 记录保存在 WebViewCache.db，而 URL 的内容是保存在 WebViewCache 文件夹下。

是否启用缓存:

```
//优先使用缓存:
WebView.getSettings().setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);
//缓存模式如下：
//LOAD_CACHE_ONLY: 不使用网络，只读取本地缓存数据
//LOAD_DEFAULT: （默认）根据 cache-control 决定是否从网络上取数据。
//LOAD_NO_CACHE: 不使用缓存，只从网络获取数据.
//LOAD_CACHE_ELSE_NETWORK，只要本地有，无论是否过期，或者 no-cache，都使用缓存中的数据。

//不使用缓存:
WebView.getSettings().setCacheMode(WebSettings.LOAD_NO_CACHE);
```

离线加载:

```
if (NetStatusUtil.isConnected(getApplicationContext())) {
    webSettings.setCacheMode(WebSettings.LOAD_DEFAULT);//根据 cache-control 决定是否从网络上取数据。
} else {
    webSettings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);//没网，则从本地获取，即离线加载
}

webSettings.setDomStorageEnabled(true); // 开启 DOM storage API 功能
webSettings.setDatabaseEnabled(true);   //开启 database storage API 功能
webSettings.setAppCacheEnabled(true);//开启 Application Caches 功能

String cacheDirPath = getFilesDir().getAbsolutePath() + APP_CACAHE_DIRNAME;
webSettings.setAppCachePath(cacheDirPath); //设置  Application Caches 缓存目录
```

- **WebViewClient 类**

作用：处理各种通知 & 请求事件

shouldOverrideUrlLoading()

作用：打开网页时不调用系统浏览器， 而是在本 WebView 中显示；在网页上的所有加载都经过这个方法。

onPageStarted()

作用：开始载入页面调用的，我们可以设定一个 loading 的页面，告诉用户程序在等待网络响应。

onPageFinished()

作用：在页面加载结束时调用。我们可以关闭 loading 条，切换程序动作。

onLoadResource()

作用：在加载页面资源时会调用，每一个资源（比如图片）的加载都会调用一次。

onReceivedError()

作用：加载页面的服务器出现错误时（如 404）调用。App 里面使用 webview 控件的时候遇到了诸如 404 这类的错误的时候，若也显示浏览器里面的那种错误提示页面就显得很丑陋了，这个时候 app 就需要加载一个本地的错误提示页面，即 webview 如何加载一个本地的页面。

onReceivedSslError()

作用：处理 https 请求，默认是不处理 https 请求的，页面显示空白。

```
// 复写 shouldOverrideUrlLoading()方法，使得打开网页时不调用系统浏览器， 而是在本 WebView 中显示
webView.setWebViewClient(new WebViewClient(){
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url);
        return true;
    }
    @Override
    public void  onPageStarted(WebView view, String url, Bitmap favicon) {
        // 设定加载开始的操作
    }
    @Override
    public void onPageFinished(WebView view, String url) {
        // 设定加载结束的操作
    }
    @Override
    public boolean onLoadResource(WebView view, String url) {
        //设定加载资源的操作
    }
    @Override
    public void onReceivedError(WebView view, int errorCode, String description, String failingUrl){
        switch(errorCode) {
            // 写一个 html 文件（error_handle.html），用于出错时展示给用户看的提示页面
            // 将该 html 文件放置到代码根目录的 assets 文件夹下
            case HttpStatus.SC_NOT_FOUND:
                view.loadUrl("file:///android_assets/error_handle.html");
                break;
        }
    }
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
        handler.proceed();              //表示等待证书响应
        // handler.cancel();            //表示挂起连接，为默认方式
        // handler.handleMessage(null); //可做其他处理
    }
    // 特别注意：5.1 以上默认禁止了 https 和 http 混用，以下方式是开启
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        mWebView.getSettings().setMixedContentMode(WebSettings.MIXED_CONTENT_ALWAYS_ALLOW);
    }
});
```

- **WebChromeClient 类**

作用：辅助 WebView 处理 Javascript 的对话框，网站图标，网站标题等等。

onProgressChanged

作用：获得网页的加载进度并显示

onReceivedTitle

作用：获取 Web 页中的标题

onJsAlert

作用：支持 javascript 的警告框

onJsConfirm

作用：支持 javascript 的确认框

onJsPrompt

作用：支持 javascript 输入框，点击确认返回输入框中的值，点击取消返回 null。

```
webView.setWebChromeClient(new WebChromeClient(){
    @Override
    public void onProgressChanged(WebView view, int newProgress) {
        if (newProgress < 100) {
            String progress = newProgress + "%";
            progress.setText(progress);
        }
    }
    @Override
    public void onReceivedTitle(WebView view, String title) {
        titleview.setText(title);
    }
    @Override
    public boolean onJsAlert(WebView view, String url, String message, final JsResult result)  {
        new AlertDialog.Builder(MainActivity.this)
                .setTitle("JsAlert")
                .setMessage(message)
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.confirm();
                    }
                })
                .setCancelable(false)
                .show();
        return true;
    }
    @Override
    public boolean onJsConfirm(WebView view, String url, String message, final JsResult result) {
        new AlertDialog.Builder(MainActivity.this)
                .setTitle("JsConfirm")
                .setMessage(message)
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.confirm();
                    }
                })
                .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.cancel();
                    }
                })
                .setCancelable(false)
                .show();
        return true;
    }
    @Override
    public boolean onJsPrompt(WebView view, String url, String message, String defaultValue,
                              final JsPromptResult result) {
        final EditText et = new EditText(MainActivity.this);
        et.setText(defaultValue);
        new AlertDialog.Builder(MainActivity.this)
                .setTitle(message)
                .setView(et)
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
```

```
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.confirm(et.getText().toString());
                    }
                })
                .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        result.cancel();
                    }
                })
                .setCancelable(false)
                .show();
        return true;
    }
});
```

- **避免 WebView 内存泄露**

1) 不在 xml 中定义 Webview ，而是在需要的时候在 Activity 中创建，并且 Context 使用 getApplicationgContext()。

```
LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
            ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.MATCH_PARENT);
mWebView = new WebView(getApplicationContext());
mWebView.setLayoutParams(params);
mLayout.addView(mWebView);
```

2) 在 Activity 销毁（WebView）的时候，先让 WebView 加载 null 内容，然后移除 WebView，再销毁 WebView，最后置空。

```
@Override
protected void onDestroy() {
    if (mWebView != null) {
        mWebView.loadDataWithBaseURL(null, "", "text/html", "utf-8", null);
        mWebView.clearHistory();
         ((ViewGroup) mWebView.getParent()).removeView(mWebView);
        mWebView.destroy();
        mWebView = null;
    }
    super.onDestroy();
}
```

82、 WebView 与 JS 的交互方式

- **Android 调用 JS 代码的方法有 2 种:**

1) 通过 WebView 的 loadUrl()，自动加载 js 代码，利用 WebChromeClient 做弹出框等交互。

```
// 设置与 Js 交互的权限
webSettings.setJavaScriptEnabled(true);
// 设置允许 JS 弹窗
webSettings.setJavaScriptCanOpenWindowsAutomatically(true);
// 格式规定为:file:///android_asset/文件名.html，自动加载 js 代码
mWebView.loadUrl("file:///android_asset/javascript.html");
```

2) 通过 WebView 的 evaluateJavascript()。

该方法的执行不会使页面刷新；Android 4.4 后才可使用。

```
// Android 版本变量
final int version = Build.VERSION.SDK_INT;
// 因为该方法在 Android 4.4 版本才可使用，所以使用时需进行版本判断
if (version < 18) {
    mWebView.loadUrl("javascript:callJS()");
} else {
    mWebView.evaluateJavascript ("javascript:callJS()", new ValueCallback<String>() {
        @Override
        public void onReceiveValue(String value) {
```

```
            //此处为 js 返回的结果
        }
    });
}
```

- **JS 调用 Android 代码的方法有 3 种：**

1) 通过 WebView 的 addJavascriptInterface() 进行对象映射。(存在严重的漏洞问题)

定义一个与 JS 对象映射关系的 Android 类：AndroidtoJs。

```
public class AndroidtoJs extends Object {
    // 定义 JS 需要调用的方法
    // 被 JS 调用的方法必须加入@JavascriptInterface 注解
    @JavascriptInterface
    public void hello(String msg) {
        System.out.println("JS 调用了 Android 的 hello 方法");
    }
}
```

将需要调用的 JS 代码以.html 格式放到 src/main/assets 文件夹里。

```
<script>
    function callAndroid(){
        // 由于对象映射，所以调用 test 对象等于调用 Android 映射的对象
        test.hello("js 调用了 android 中的 hello 方法");
    }
</script>
<body>
    //点击按钮则调用 callAndroid 函数
    <button type="button" id="button1" onclick="callAndroid()"></button>
</body>
```

在 Android 里通过 WebView 设置 Android 类与 JS 代码的映射。

```
// 设置与 Js 交互的权限
webSettings.setJavaScriptEnabled(true);

// 通过 addJavascriptInterface()将 Java 对象映射到 JS 对象
//参数 1：Javascript 对象名
//参数 2：Java 对象名
mWebView.addJavascriptInterface(new AndroidtoJs(), "test");// AndroidtoJS 类对象映射到 js 的 test 对象

// 加载 JS 代码
// 格式规定为:file:///android_asset/文件名.html
mWebView.loadUrl("file:///android_asset/javascript.html");
```

2) 通过 WebViewClient 的 shouldOverrideUrlLoading() 方法回调拦截 url。

a) Android 通过 WebViewClient 的回调方法 shouldOverrideUrlLoading () 拦截 url；

b) 解析该 url 的协议；

c) 如果检测到是预先约定好的协议，就调用相应方法。

```
<script>
    function callAndroid(){
        /*约定的 url 协议为：js://webview?arg1=111&arg2=222*/
        document.location = "js://webview?arg1=111&arg2=222";
    }
</script>
<body>
    //点击按钮则调用 callAndroid 函数
    <button type="button" id="button1" onclick="callAndroid()"></button>
</body>
```

```
Uri uri = Uri.parse(url);
// 如果 url 的协议 = 预先约定的 js 协议，就解析往下解析参数
```

```
    if (uri.getScheme().equals("js")) {
        // 如果 authority  = 预先约定协议里的 webview，即代表都符合约定的协议
        // 所以拦截 url,下面 JS 开始调用 Android 需要的方法
        if (uri.getAuthority().equals("webview")) {
            // 执行 JS 所需要调用的逻辑
            // 可以在协议上带有参数并传递到 Android 上
            HashMap<String, String> params = new HashMap<>();
            Set<String> collection = uri.getQueryParameterNames();
        }
        return true;
    }
```

如果 JS 想要得到 Android 方法的返回值，只能通过 WebView 的 loadUrl （）去执行 JS 方法把返回值传递回去：

```
    // Android：MainActivity.java
    mWebView.loadUrl("javascript:returnResult(" + result + ")");

    // JS：javascript.html
    function returnResult(result){
        alert("result is" + result);
    }
```

3) 通过 WebChromeClient 的 onJsAlert()、onJsConfirm()、onJsPrompt() 方法回调拦截 JS 对话框 alert()、confirm()、prompt() 消息。

- **总结**

类型	调用方式	优点	缺点	使用场景
Android 调用 JS	loadUrl （）	方便简洁	效率低、获取返回值麻烦	不需要获取返回值，对性能要求较低时
	evaluateJavascript （）	效率高	向下兼容性差 (仅Android 4.4以上可用)	Android 4.4以上
JS 调用 Android	通过addJavascriptInterface （）进行添加对象映射	方便简洁	Android 4.2以下存在漏洞问题	Android 4.2以上相对简单互调场景
	通过 WebViewClient.shouldOverrideUrlLoading ()回调拦截 url	不存在漏洞问题	使用复杂：需要进行协议的约束；从 Native 层往 Web 层传递值比较繁琐	不需要返回值情况下的互调场景 (iOS主要使用该方式)
	通过 WebChromeClient的onJsAlert()、onJsConfirm()、onJsPrompt （）方法回调拦截JS对话框消息	不存在漏洞问题	使用复杂：需要进行协议的约束；	能满足大多数情况下的互调场景

83、 WebView 使用漏洞

- **任意代码执行漏洞**

addJavascriptInterface 接口引起远程代码执行漏洞

原因：当 JS 拿到 Android 这个对象后，就可以调用这个 Android 对象中所有的方法，包括系统类（java.lang.Runtime 类，Java 反射机制），从而进行任意代码执行。如可以执行命令获取本地设备的 SD 卡中的文件等信息从而造成信息泄露。

```
    function execute(cmdArgs) {
        // 步骤 1：遍历 window 对象，目的是为了找到包含 getClass （）的对象
        // 因为 Android 映射的 JS 对象也在 window 中，所以肯定会遍历到
        for (var obj in window) {
            if ("getClass" in window[obj]) {
                // 步骤 2：利用反射调用 forName（）得到 Runtime 类对象
                alert(obj);
                return  window[obj].getClass().forName("java.lang.Runtime")
                // 步骤 3：以后，就可以调用静态方法来执行一些命令，比如访问文件的命令
                getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs);
                // 从执行命令后返回的输入流中得到字符串，有很严重暴露隐私的危险。
                // 如执行完访问文件的命令之后，就可以得到文件名的信息了。
```

```
            }
        }
    }
```

解决方案：

4.2 版本之后，被调用的函数以 @JavascriptInterface 进行注解从而避免漏洞攻击；

4.2 版本之前，采用拦截 prompt() 进行漏洞修复，过滤掉 Object 类的方法。

- **密码明文存储漏洞**

原因：WebView 默认开启密码保存功能，开启后，在用户输入密码时，会弹出提示框：询问用户是否保存密码。密码会被明文保到 /data/data/com.package.name/databases/webview.db。

```
mWebView.setSavePassword(true)
```

解决方案：

关闭密码保存提醒

- **域控制不严格漏洞**

原因：A 应用可以通过 B 应用导出的 Activity 让 B 应用加载一个恶意的 file 协议的 url，从而可以获取 B 应用的内部私有文件，从而带来数据泄露威胁。

解决方案：

WebView 中 getSettings 类的方法：

1) setAllowFileAccess：设置是否允许 WebView 使用 File 协议。

对于不需要使用 file 协议的应用，禁用 file 协议；

对于需要使用 file 协议的应用，禁止 file 协议加载 JavaScript。

```

    setAllowFileAccess(true);
    // 禁止 file 协议加载 JavaScript
    if (url.startsWith("file://")
        setJavaScriptEnabled(false);
    else
        setJavaScriptEnabled(true);

```

2) setAllowFileAccessFromFileURLs：设置是否允许通过 file url 加载的 Js 代码读取其他的本地文件。

在 Android 4.1 前默认允许；在 Android 4.1 后默认禁止。

3) setAllowUniversalAccessFromFileURLs：设置是否允许通过 file url 加载的 Javascript 可以访问其他的源(包括 http/https 等)。

在 Android 4.1 前默认允许（setAllowFileAccessFromFileURLs 不起作用)；在 Android 4.1 后默认禁止。

4) setJavaScriptEnabled：设置是否允许 WebView 使用 JavaScript（默认是不允许)。

即使把 setAllowFileAccessFromFileURLs 和 setAllowUniversalAccessFromFileURLs 都设置为 false，通过 file URL 加载的 javascript 仍然有方法访问其他的本地文件：符号链接跨源攻击。

通过 javascript 的延时执行和将当前文件替换成指向其它文件的软链接就可以读取到被符号链接所指的文件。具体攻击步骤：

a) 把恶意的 js 代码输出到攻击应用的目录下，随机命名为 xx.html，修改该目录的权限；

b) 修改后休眠 1s，让文件操作完成；

c) 完成后通过系统的 Chrome 应用去打开该 xx.html 文件

d) 等待 4s 让 Chrome 加载完成该 html，最后将该 html 删除，并且使用 ln -s 命令为 Chrome 的 Cookie 文件创建软连接。

84、 Dalvik 虚拟机与 Java 虚拟机的区别

1) Java 虚拟机运行的是 Java 字节码，Dalvik 虚拟机运行的是 Dalvik 字节码。传统的 Java 程序经过编译，生成 Java 字节码保存在 class 文件中，java 虚拟机通过解码 class 文件中的内容来运行程序。而 Dalvik 虚拟机运行的是 Dalvik 字节码，所有的 Dalvik 字节码由 Java 字节码转换而来，并被打包到一个 DEX(Dalvik Executable)可执行文件，Dalvik 虚拟机通过解释 Dex 文件来执行这些字节码。

2) Dalvik 可执行文件体积更小。SDK 中有一个叫 dx 的工具负责将 java 字节码转换为 Dalvik 字节码。消除其中的冗余信息，重新组合形成一个常量池，所有的类文件共享同一个常量池。由于 dx 工具对常量池的压缩，使得相同的字符串，常量在 DEX

文件中只出现一次，从而减小了文件的体积。

3) Java 虚拟机与 Dalvik 虚拟机架构不同。java 虚拟机基于栈架构。程序在运行时虚拟机需要频繁的从栈上读取或写入数据。这过程需要更多的指令分派与内存访问次数，会耗费不少 CPU 时间，对于像手机设备资源有限的设备来说，这是相当大的一笔开销。Dalvik 虚拟机基于寄存器架构，数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式快的多。

85、 ART 虚拟机（Android Runtime）与 Dalvik 虚拟机的区别

Dalvik 是依靠一个 Just-In-Time (JIT)编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。在 dalvik 中，如同其他大多数 JVM 一样，都采用的是 JIT 来做及时翻译(动态翻译)，将 dex 或 odex 中并排的 dalvik code（或者叫 smali 指令集）**运行态**翻译成 native code 去执行，JIT 的引入使得 dalvik 提升了 3~6 倍的性能。

ART 完全改变了这套做法，在应用安装时就预编译字节码到机器语言，这一机制叫 Ahead-Of-Time (AOT）编译。在移除解释代码这一过程后，应用程序执行将更有效率，启动更快。在 ART 中，完全抛弃了 dalvik 的 JIT，使用了 AOT 直接在安装时将其完全翻译成 native code，这一技术的引入，使得虚拟机执行指令的速度又一重大提升。

ART 优点：

1) 系统性能的显著提升。
2) 应用启动更快、运行更快、体验更流畅、触感反馈更及时。
3) 更长的电池续航能力。
4) 支持更低的硬件。

ART 缺点：

1) 更大的存储空间占用，可能会增加 10%-20%。
2) 更长的应用安装时间。

- **垃圾回收机制**

Dalvik 的 GC 的过程：

1) 当 gc 被触发时候，其会去查找所有活动的对象，这个时候整个程序与虚拟机内部的所有线程就会挂起，这样目的是在较少的堆栈里找到所引用的对象，需要注意的是这个回收动作和应用程序**非并发。**
2) gc 对符合条件的对象进行标记。
3) gc 对标记的对象进行回收。
4) 恢复所有线程的执行现场继续运行。

ART 的 GC 的过程：

1) GC 将会锁住 Java 堆，扫描并进行标记。
2) 标记完毕释放掉 Java 堆的锁，并且挂起所有线程。
3) GC 对标记的对象进行回收。
4) 恢复所有线程的执行现场继续运行。
5) 重复 2-4 直到结束。

区别：

Dalvik：当 pause 了之后，GC 势必是相当快速的，但是如果出现 GC 频繁并且内存吃紧，势必会导致 UI 卡顿、掉帧、操作不流畅等。

ART：将其非并发过程改变成了部分并发。

内存管理：

Dalvik：内存碎片化严重，这也是 Mark and Sweep 算法带来的弊端。

ART：将 Java 分了一块空间命名为 Large-Object-Space，这块内存空间的引入用来专门存放 large object。同时 ART 又引入了 moving collector 的技术，即将不连续的物理内存块进行对齐，对齐了后内存碎片化就得到了很好的解决，Large-Object-Space 的引入一是因为 moving collector 对大块内存的位移时间成本太高，二是提高内存的利用率。

86、 APK 编译打包流程

产出为.class 文件。Java 编译器对工程本身的 java 代码进行编译，这些 java 代码有三个来源：app 的源代码，由资源文件生成的 R 文件(aapt 工具)，以及有 aidl 文件生成的 java 接口文件(aidl 工具)。

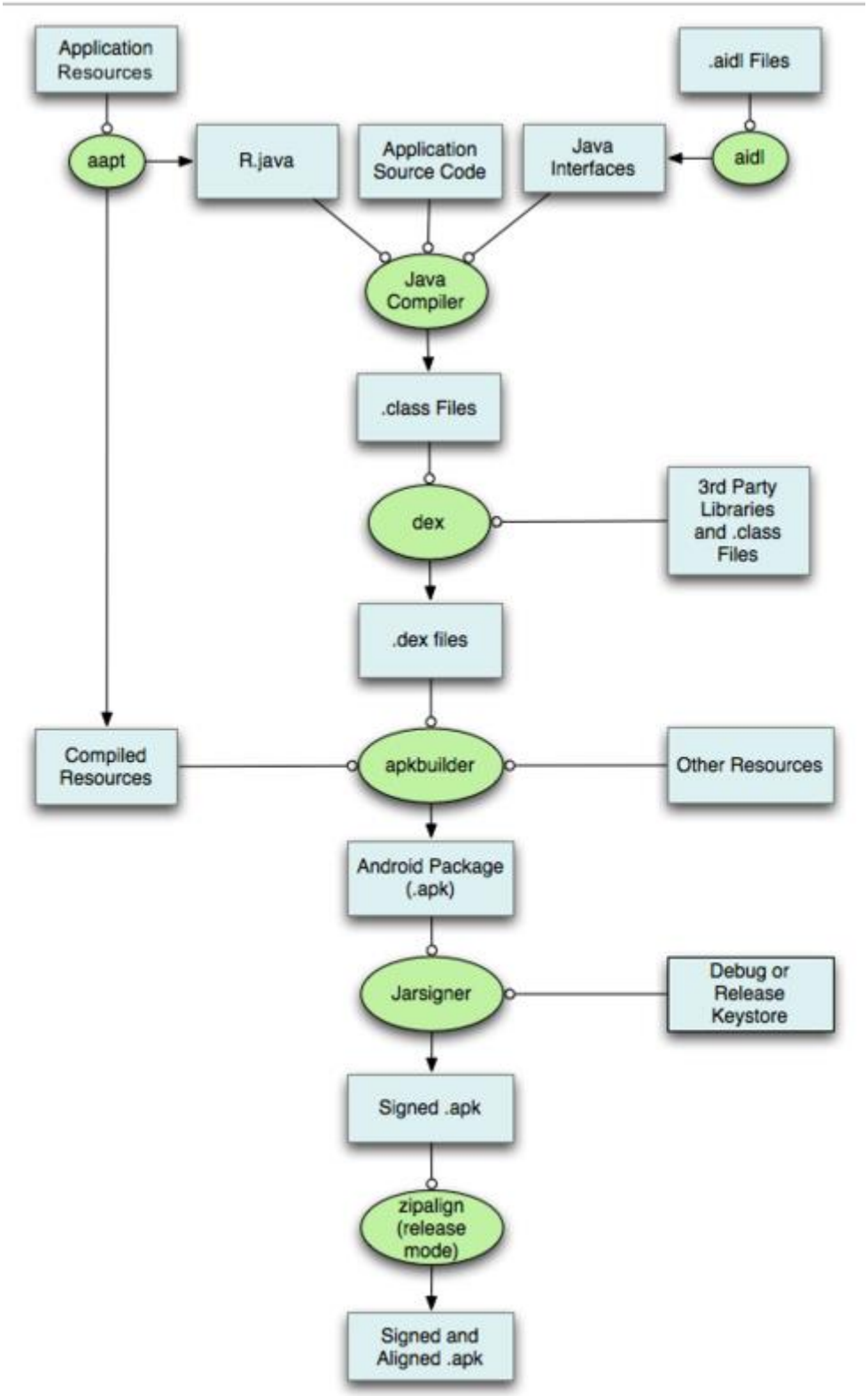
产出为.dex 文件。class 文件和依赖的第三方库文件通过 dex 工具生成 Delvik 虚拟机可执行的.dex 文件，包含了所有的 class 信息，包括项目自身的 class 和依赖的 class。

产出为未经签名的.apk 文件。apkbuilder 工具将.dex 文件和编译后的资源文件生成未经签名对齐的 apk 文件。这里编译后的资源

文件包括两部分，一是由 aapt 编译产生的编译后的资源文件，二是依赖的三方库里的资源文件。

生成最终的 apk 文件。分别由 Jarsigner 和 zipalign 对 apk 文件进行签名和对齐。

总结为：编译-->DEX-->**打包**-->**签名和对齐。**



87、 APK 文件结构和安装过程

- **APK 文件结构**

assets 目录：用于存放需要打包到 APK 中的静态文件，和 res 的不同点在于，assets 目录支持任意深度的子目录，用户可以根据自己的需求任意部署文件夹架构，而且 res 目录下的文件会在.R 文件中生成对应的资源 ID，assets 不会自动生成对应的 ID，访问的时候需要 AssetManager 类。

lib 目录：这里存放应用程序依赖的 native 库文件，一般是用 C/C++编写，这里的 lib 库可能包含多种不同类型，根据 CPU 型号的不同，大体可以分为 ARM，ARM-v7a，arm64-v8a，MIPS，mips64，X86，x86_64。目前市场上使用的移动终端大多是基于 ARM 或者 ARM-V7a 架构的。

res 目录：res 是 resource 的缩写，这个目录存放资源文件，存在这个文件夹下的所有文件都会映射到 Android 工程的.R 文件中，生成对应的 ID，访问的时候直接使用资源 ID 即 R.id.filename，res 文件夹下可以包含多个文件夹，其中 anim 存放动画文件；drawable 目录存放图像资源；layout 目录存放布局文件；values 目录存放一些特征值，colors.xml 存放 color 颜色值，dimens.xml 定义尺寸值，string.xml 定义字符串的值，styles.xml 定义样式对象；xml 文件夹存放任意 xml 文件，在运行时可以通过 Resources.getXML() 读取；raw 是可以直接复制到设备中的任意文件，他们无需编译。

META-INF 目录：保存应用的签名信息，签名信息可以验证 APK 文件的完整性。AndroidSDK 在打包 APK 时会计算 APK 包中所有文件的完整性，并且把这些完整性保存到 META-INF 文件夹下，应用程序在安装的时候首先会根据 META-INF 文件夹校验 APK 的完整性，这样就可以保证 APK 中的每一个文件都不能被篡改。以此来确保 APK 应用程序不被恶意修改或者病毒感染，有利于确保 Android 应用的完整性和系统的安全性。META-INF 目录下包含的文件有 CERT.RSA，CERT.DSA，CERT.SF 和 MANIFEST.MF，其中 CERT.RSA 是开发者利用私钥对 APK 进行签名的签名文件，CERT.SF，MANIFEST.MF 记录了文件中文件的 SHA-1 哈希值。

AndroidManifest.xml：是 Android 应用程序的配置文件，是一个用来描述 Android 应用"整体资讯"的设定文件，简单来说，相当于 Android 应用向 Android 系统"自我介绍"的配置文件，Android 系统可以根据这个"自我介绍"完整地了解 APK 应用程序的资讯，每个 Android 应用程序都必须包含一个 AndroidManifest.xml 文件，且它的名字是固定的，不能修改。我们在开发 Android 应用程序的时候，一般都把代码中的每一个 Activity，Service，Provider 和 Receiver 在 AndroidManifest.xml 中注册，只有这样系统才能启动对应的组件，另外这个文件还包含一些权限声明以及使用的 SDK 版本信息等等。程序打包时，会把 AndroidManifest.xml 进行简单的编译，便于 Android 系统识别，编译之后的格式是 AXML 格式，如下图 Figure3 所示：

classes.dex：传统的 Java 程序，首先先把 Java 文件编译成 class 文件，字节码都保存在了 class 文件中，Java 虚拟机可以通过解释执行这些 class 文件。而 Dalvik 虚拟机是在 Java 虚拟机进行了优化，执行的是 Dalvik 字节码，而这些 Dalvik 字节码是由 Java 字节码转换而来，一般情况下，Android 应用在打包时通过 AndroidSDK 中的 dx 工具将 Java 字节码转换为 Dalvik 字节码。dx 工具可以对多个 class 文件进行合并，重组，优化，可以达到减小体积，缩短运行时间的目的。

resources.arsc：用来记录资源文件和资源 ID 之间的映射关系，用来根据资源 ID 寻找资源。Android 的开发是分模块的，res 目录专门用来存放资源文件，当在代码中需要调用资源文件时，只需要调用 findviewbyId()就可以得到资源文件，每当在 res 文件夹下放一个文件，aapt 就会自动生成对应的 ID 保存在.R 文件，我们调用这个 ID 就可以，但是只有这个 ID 还不够，.R 文件只是保证编译程序不报错，实际上在程序运行时，系统要根据 ID 去寻找对应的资源路径，而 resources.arsc 文件就是用来记录这些 ID 和资源文件位置对应关系的文件。

- **APK 安装过程**

/data/app：存放用户安装的 APK 的目录，安装时，把 APK 拷贝于此。

/data/data：应用安装完成后，在/data/data 目录下自动生成和 APK 包名（packagename）一样的文件夹，用于存放应用程序的数据。

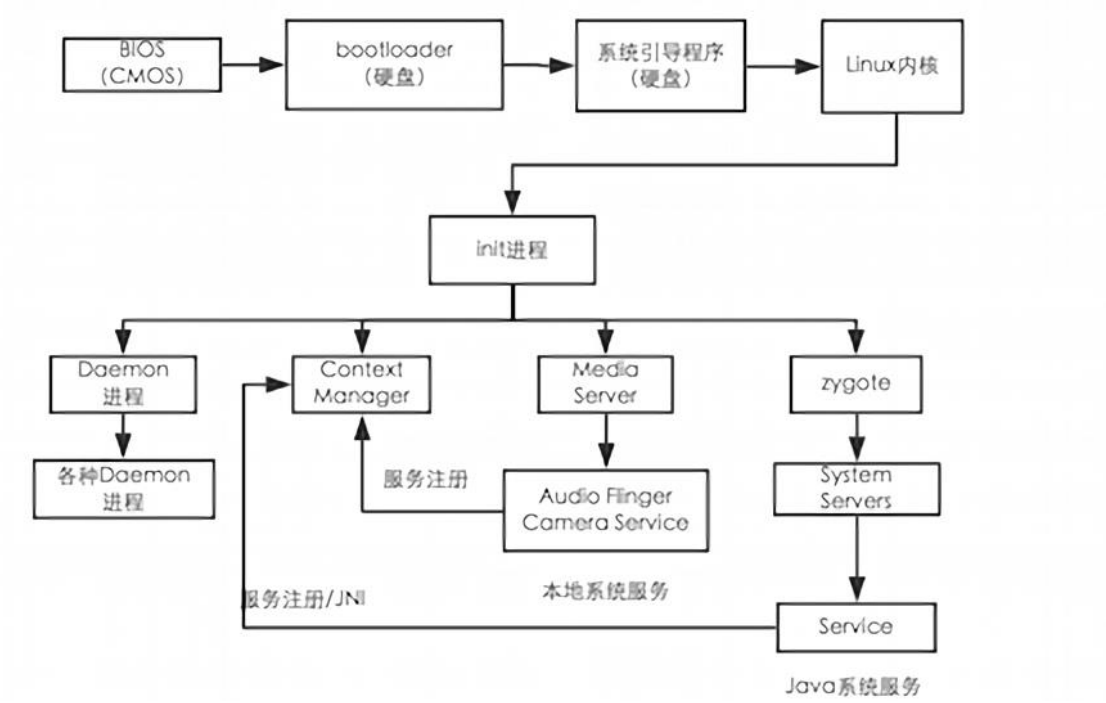
/data/dalvik-cache：存放 APK 的 odex 文件，便于应用启动时直接执行。

首先，复制 APK 安装包到/data/app 下，然后校验 APK 的签名是否正确，检查 APK 的结构是否正常，进而解压并且校验 APK 中的 dex 文件，确定 dex 文件没有被损坏后，再把 dex 优化成 odex，使得应用程序启动时间加快，同时在/data/data 目录下建立与 APK 包名相同的文件夹，如果 APK 中有 lib 库，系统会判断这些 so 库的名字，查看是否以 lib 开头，是否以.so 结尾，再根据 CPU 的架构解压对应的 so 库到/data/data/packagename/lib 下。

APK 安装的时候会把 DEX 文件解压并且优化位 odex，odex 在原来的 dex 文件头添加了一些数据，在文件尾部添加了程序运行时需要的依赖库和辅助数据，使得程序运行速度更快。

88、 Android 从开机到打开第一个应用发生了什么?

- **启动过程**



1) 当我们开机的时候，此时通电了，此时会产生一个确定的复位时序，保证 CPU 是第一个被复位的器件，然后 CPU 开始执行第一条指令，该指令是位于固定的内存地址上的，所谓的引导程序。
2) BIOS 从我们的磁盘中寻找我们操作系统引导程序，将引导程序加载到内存中，执行，然后将我们的 Linux 内核加载到内存中。
3) 启动 init 进程，这个进程是用户态所有进程的祖先。
4) init 进程首先通过 fork 创建子进程，首先是 Daemon 进程也就是守护进程，包括 USB 守护进程，Debug 进程，无线通信连接守护进程。
5) fork 出 Context Manager，通过上图我们可以看出，android 系统提供的服务都要向其进行注册，然后其它的进程才可以调用这个服务。
6) Media Server，这个服务是本地服务，不是 Java 服务，不需要再通过 dalvik 虚拟机装载执行，本地服务单独开启了一个进程。这里包括 Audio Flinger 和 Camera Service。
7) Zygote，所有 Android 应用程序的祖先，其用来缩短应用程序的加载时间，具体怎么节省，下面详解。
8) System Server 是 android 系统中的一个核心进程，提供了管理应用程序生命周期，地理位置信息等各种服务。同时其需要将自身注册到 Context Manager.但是我们的服务管理器是基于 C 语言的，所以我们需要 JNI 本地编程接口。这个时候 Activity Manager Service 会运行 HOME 应用。

- **BIOS**

PC 启动:

BIOS 程序固化在主板上的一块芯片上，是连接计算机硬件与操作系统的桥梁，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，一般是汇编程序。BIOS 的启动，是由硬件完成的，Intel 80x86 系列的 cpu 的硬件都设计为加电（即开机瞬间）就进入 16 位实模式状态运行，此时将 cpu 的硬件逻辑设计为强行将 CS 的值设置为 0xFFFF，IP 的值设置为 0x0000，这样 CS:IP 就指向了 0xFFFF0 这个位置，而这个位置就是 BIOS 程序的入口地址。

Android 启动：

Android 系统虽然也是基于 linux 系统的，但是由于 Android 属于嵌入式设备，并没有像 pc 那样的 BIOS 程序。取而代之的是 Bootloader——系统启动加载器。它类似于 BIOS，在系统加载前，用以初始化硬件设备，建立内存空间的映像图，为最终调用系统内核准备好环境。Android 手机会将固态存储设备 ROM 预先映射到 0xFFFF0 地址上 2，当开机加电的时候，cpu 就会从该地址执行/boot 分区下的 Bootloader 程序，载入 linux 内核到 RAM 中。

- **init 进程**

init 进程是第一个启动的用户进程，其它用户进程都是其子进程，或者其子进程的子进程。

作用：

1) 子进程终止处理。
2) 应用程序访问设备驱动时，生成设备结点文件。
3) 提供属性服务，保存系统运行所需要的环境变量。

init 运行

1) init 进程会先注册一些消息处理器（子进程退出的信号处理函数）；
2) 创建一些文件夹并挂载设备以及启动所需要的文件目录（socket 文件，虚拟内存文件）；
3) 在 dev 目录下生成设备节点文件，然后将标准输入，输出，错误输出重定向到/dev/null，将日志输出设备重定向为/dev/__kmsg__；
4) init 进程生成输出设备之后，开始解析 init.rc 脚本文件。init.rc 通过函数 parse_config_file 来解析，读取分析之后，生成服务列表和动作列表，服务列表和动作列表会分别注册到 service_list 和 action_list 中
5) 调用 device_init 函数，创建利用 Uevent 与 Linux 内核交互的 socket；
6) 之后调用 property_init 函数初始化和属性相关的资源，在共享内存区域，创建并初始化属性值，对于全局属性的修改，只有 init 进程可以修改，当要修改的时候，需要预先向其提交申请，然后 init 进程通过之后，才会去修改属性值，提交申请的过程会创建一个 socket 用来接收提交的申请。执行到这系统将 **Android 系统的 Logo 显示在桌面上。**
7) 设置事件处理循环的监视事件，注册在 POLL 中的文件描述符（内核利用文件描述符来访问文件）会在 poll 函数中等待事件，如果事件发生，则从 poll 函数中跳出并处理事件。

init.rc 文件的处理过程

init.rc 文件分析函数，通过 read_file 函数和 parse_config 函数来分析读入的字符串。（AIL ，Android Init Language）

init.rc 文件大致上分为两个部分，一部分是以“on”关键字开头的动作列表，另一部分是以”service”关键字开头的服务列表。

动作列表：主要设置环境变量，生成系统运行所需的文件或目录，修改相应的权限，并挂载和系统运行相关的目录。在挂载文件的时候，主要挂载/system 和/data 两个目录，两个目录挂载完毕，android 根文件系统就准备好了。根文件系统大致可分为 shell 使用程序、system 目录（提供库和基础应用）、data 目录（保存用户应用和数据）。Android 采用闪存设备，其采用了 yaffs2 文件系统，启动的时候要挂载到/system 和/data 目录下。然后是 on boot 段落，该部分设置应用程序终止条件，应用程序驱动目录和文件权限。为各应用制定 OOM（内存溢出）调整值，OOM 用来监视内核分配给应用程序的内存，当内存不足的时候，应用程序会被终止执行。

服务列表：init.rc 脚本文件，service 段落用来记录 init 进程启动的进程，由 init 进程启动的子进程或者是一次性程序，系统相关的 Daemon 进程。

创建设备节点文件：

和 Linux 相同，应用程序通过驱动程序访问硬件设备，设备节点文件是设备驱动的逻辑文件，应用程序通过设备节点文件来访问设备驱动程序。

设备节点有两种创建方式：

1、根据预先定义的设备信息，创建设备节点文件；

2、在系统运行中，当设备插入时，init 进程会接收这一事件，为插入设备动态创建设备节点文件。当设备插入的时候内核会加载相应的驱动程序，而后驱动程序会调用启动函数 probe，将主、次设备号类型保存到/sys 文件系统中。然后发出 uevent，并传递给守护进程，uevent 是内核向用户控件进程传递信息的信号系统，内核通过 uevent 将信息传递到用户空间，守护进程会根据 uevent 来读取设备信息，创建设备节点文件。对于一些采用冷插拔的设备，会监听设备的 uevent，然后调用其函数来创建设备节点文件。

何为设备节点文件

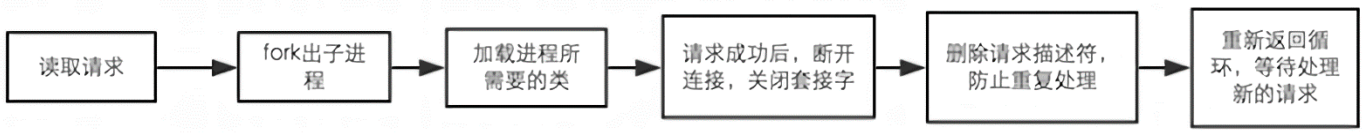
Linux 对于系统中的设备都会抽象成一个文件，内核为了高效的管理已经被打开的文件，通过一个文件描述符来表示，在 Linux 中 Everything is file，通过这种方式，对于启动的时候，对于文件描述符，0 代表标注输入，1 表示标准输出，2 是错误处理，然后设备文件，socket 文件都会获得一个文件描述符来表述它。但是 Linux 会对其做相应的限制，同时可能会对一些进程进行限制，限

制给进程分配多少个文件描述符。

- **Zygote 创建过程**

Zygote 本身是一个 Native 的应用程序，与驱动、内核无关，且 Zygote 由 init 进程根据 init.rc 文件中的配置项创建（fork）。

1) 首先生成了一个 AppRuntime 对象并调用了 start 函数，该类继承自 AndroidRuntime，即调用的 start 函数在父类中。runtime.start("com.android.internal.os.ZygoteInit", startSystemServer)的执行过程：
   1、 创建虚拟机 startVm：调用 Jni 的虚拟机创建函数；
   2、 注册 JNI 函数 startReg：java 中使用的一些函数是 native 方式实现的，需提前注册这些函数；
2) 虚拟机初始化之后，运行 ZygoteInit 类（通过路径查找找到），现在程序的执行转向了虚拟机 Java 代码的执行，首先执行 main 函数。
   1、 建立 IPC 通信服务端 registerZygoteSocket：zygote 与系统其它程序的通信没有使用 Binder，而是采用基于 AF_UNIX 类型的 Socket，该函数的使命就是建立这个 Socket，用来接收从 Activity Manager 来的应用启动请求；
   2、 预加载类和资源 preloadClasses、preloadResources：将应用程序框架中的类、平台资源（图像，XML 信息，字符串）预先加载到内存中，以此提升程序的执行速度。在 Zygote 创建的时候加载资源，生成信息链接等，以后再有应用启动，fork 出一个子进程和父进程共享这些信息，而不需要重新加载，同时也共享虚拟机，因为虚拟机在初始化和创建的过程是很耗时的。
   3、 启动 system_server（startSystemServer）：该进程为 framework 的核心，应用启动需要 server 的参与，所以在启动应用之前先启动了这些 System server，然后启动 Server Thread 来执行 android framework 的服务，同时这些服务在启动的时候都是需要通过 JNI 向服务管理器 Context Manager 注册。
   4、 等待请求 runSelectLoopMode：处理客户连接和客户请求，客户在 zygote 中用 ZygoteConnection 对象来表示，客户的请求由 ZygoteConnection 的 runOnce 来处理。Zygote 此时处在轮询监听 Socket 的状态，当有请求到达时，读取请求，fork 出子进程，加载进程所需要的类，然后执行程序的 main 函数，代码转给了 Dalvik Vm，我们的应用程序也就启动起来了。这个时候将会关闭套接字，删除请求描述符，防止出现重复启动



这里的虚拟机可以理解成：可以解析 Java 字节码的内存上的一条条指令，各个应用共享这个虚拟机，也就是都是知道虚拟机的内存中地址，传递进 java 字节码，然后解析字节码，将字节码实际代表的操作反应出来，多个应用程序可以交替使用，通过寄存器记录彼此执行后的状态。fork 出一个进程之后，开始装载应用程序相应的类，然后执行。

89、 安卓各大版本的差异

- **安卓 4.X**

1) 引入“Holo”界面，在设计追求简约上面充满了浓浓的工程师风格，慢慢脱离苹果风格，而且在往后版本中也开始注重对界面的设计。
2) 重新恢复开源，第三方刷机包开始变多。

- **安卓 5.X**

1) “Material Design”中文名 材料设计，安卓界面开发采用卡片化，扁平化，在原来的 XY 轴的基础上添加 Z 轴的设计理念。
2) 添加更多类型的传感器。
3) 添加卡片显示的后台进程查看
4) 添加通知栏浮动通知
5) 添加了新的摄影技巧以及虚拟摄像机 API，为开发者提供更丰富的摄像头控制
6) Android 运行时由 Android 核心库集和 Dalvike 虚拟机改成 Android 核心库集和 ART。两者的区别就是 Dalvike 虚拟机采用了一种被称为 JIT（just-in-time）的解释器进行动态编译,而 ART 模式则在用户安装 App 是进行预编译 AOT（Ahead-of-time）。将 android5.X 的运行速度提高了 3 倍左右。

- **安卓 6.X**

1) 动态权限的出现，这是对安卓开发最大变化。
2) Doze 电量管理功能，在“Doze”模式下，手机会在一段时间未检测到移动时，让应用休眠清杀后台进程减少功耗，谷歌表示，当屏幕处于关闭状态，平均续航时间提高 30%，这个区别于 IOS 的墓碑机制。在安卓开发，需要后台运行时，最好在前台留有进程，防止被误杀。

3) 从 Android6.X 起，Ecilpse ADT 不再更新支持 Android 开发。
4) 谷歌正式将指纹识别加入系统底层，开发相关的 API，加大指纹开发的安全性。
5) 谷歌还加入了 Android Pay 进一步强化移动支付，同时也是为了对抗 Apple Pay。

- **安卓 7.X**

1) 原生的分屏模式的加入
2) Doze 电量管理的优化
3) 更便捷的通知栏，自动将多条通知合并。
4) 引入了全新的 VulkanAPI 图形处理器 API，可以大幅减少系统动画对 CPU 的占用。
5) 支持 app 应用签名 v2 的打包方式（在 AS2.2 后，在打包签名应用时，可勾选 jar 打包（v1）和全应用打包（v2），详情自行百度）

- **安卓 8.X**

1) 安装未知来源的第三方开关被移出，变成了每次安装未知的第三方都要手动授权。
2) 通知功能的改变，应用收到通知时，会在应用的右上角显示一个红点，长按会跳出一个弹出菜单。
3) 画中画功能的加入。
4) 支持自动填写的功能。

- **Android P（预览版）**

1) WIFI RTT 进行室内高精度定位。
2) 对凹口屏幕的支持，提供 API 供开发者开发。
3) 对多摄像头的开发支持。
4) 处理图像解码，提供 ImageDecoder 替换原来 BitmapFactory
5) 加大了对 Kotlin 的支持，对编译器进行优化

- **Android Pie（正式版）**

1) 动态电量变化。利用机器学习技术对系统资源进行有限分配。
2) 文本识别与 Smart Linkify。利用机器学习模型，能够识别出类似日期或者航班这样的信息。此外，Smart Linkify 还允许开发者通过 Linkify API 使用文本识别模块完成多项操作。
3) 新增神经网络 API1.1。增加了 9 个新算子的支持，分别是 Pad、BatchToApaceND、SpaceToBatchND、TransPose、Strided Slice、Mean、Div、Sub 和 Squeeze。
4) 凹口屏的支持
5) 增加文本放大镜
6) 默认使用 HTTPS
7) 隐私权限的优化
8) 通过 WI-FI RTT 室内定位

90、 Android6.0 权限适配

- **安卓 6.0 新权限系统分类有两种：**

普通权限（normal）：这个权限类型并不直接威胁到用户的隐私，可以直接在 manifest 清单里注册，系统会帮我们默认授权的。

```
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
android.permission.ACCESS_NETWORK_STATE
android.permission.ACCESS_NOTIFICATION_POLICY
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_WIMAX_STATE
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.BROADCAST_STICKY
android.permission.CHANGE_NETWORK_STATE
android.permission.CHANGE_WIFI_MULTICAST_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.CHANGE_WIMAX_STATE
android.permission.DISABLE_KEYGUARD
android.permission.EXPAND_STATUS_BAR
android.permission.FLASHLIGHT
android.permission.GET_ACCOUNTS
android.permission.GET_PACKAGE_SIZE
android.permission.INTERNET
android.permission.KILL_BACKGROUND_PROCESSES
android.permission.MODIFY_AUDIO_SETTINGS
android.permission.NFC
android.permission.READ_SYNC_SETTINGS
android.permission.READ_SYNC_STATS
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.REORDER_TASKS
android.permission.REQUEST_INSTALL_PACKAGES
android.permission.SET_TIME_ZONE
android.permission.SET_WALLPAPER
android.permission.SET_WALLPAPER_HINTS
android.permission.SUBSCRIBED_FEEDS_READ
android.permission.TRANSMIT_IR
android.permission.USE_FINGERPRINT
android.permission.VIBRATE
android.permission.WAKE_LOCK
android.permission.WRITE_SYNC_SETTINGS
com.android.alarm.permission.SET_ALARM
com.android.launcher.permission.INSTALL_SHORTCUT
com.android.launcher.permission.UNINSTALL_SHORTCUT
```

危险权限（dangerous）：这个可以直接给 app 访问用户一些敏感的数据，不仅需要在 manifest 清单里注册，同时在使用的时

候，需要向系统请求授权。

Permission Group	Permissions
android.permission-group.CALENDAR	• android.permission.READ_CALENDAR • android.permission.WRITE_CALENDAR
android.permission-group.CAMERA	• android.permission.CAMERA
android.permission-group.CONTACTS	• android.permission.READ_CONTACTS • android.permission.WRITE_CONTACTS • android.permission.GET_ACCOUNTS
android.permission-group.LOCATION	• android.permission.ACCESS_FINE_LOCATION • android.permission.ACCESS_COARSE_LOCATION
android.permission-group.MICROPHONE	• android.permission.RECORD_AUDIO
android.permission-group.PHONE	• android.permission.READ_PHONE_STATE • android.permission.CALL_PHONE • android.permission.READ_CALL_LOG • android.permission.WRITE_CALL_LOG • com.android.voicemail.permission.ADD_VOICEMAIL • android.permission.USE_SIP • android.permission.PROCESS_OUTGOING_CALLS
android.permission-group.SENSORS	• android.permission.BODY_SENSORS
android.permission-group.SMS	• android.permission.SEND_SMS • android.permission.RECEIVE_SMS • android.permission.READ_SMS • android.permission.RECEIVE_WAP_PUSH • android.permission.RECEIVE_MMS • android.permission.READ_CELL_BROADCASTS
android.permission-group.STORAGE	• android.permission.READ_EXTERNAL_STORAGE • android.permission.WRITE_EXTERNAL_STORAGE

- **申请权限：**

1) 申明该权限（manifest）

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

2) 检查是否已经有该权限
3) 如果没有则进行申请权限
4) 接收申请成功或者失败回调

```
public class PermissionActivity extends AppCompatActivity {

    private static final int Location_Permission = 0x01;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_permission);

        if (Build.VERSION.SDK_INT >= 23) {
```

```
            requestPermission();
        }
    }

    private void requestPermission() {
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) !=
            PackageManager.PERMISSION_GRANTED) {
            //询问用户是否拒绝过，如果没有，该方法返回 false，如果被拒绝过，该方法返回 true
            if (ActivityCompat.shouldShowRequestPermissionRationale(this,
                Manifest.permission.ACCESS_COARSE_LOCATION)) {
                // 用户拒绝过这个权限了，应该提示用户，为什么需要这个权限
                new AlertDialog.Builder(this)
                        .setTitle("友好提醒")
                        .setMessage("没有定位权限将不能搜索附近蓝牙，请把定位权限赐给我吧！")
                        .setPositiveButton("赏你", new DialogInterface.OnClickListener() {
                            @Override
                            public void onClick(DialogInterface dialog, int which) {
                                dialog.cancel();
                                // 用户同意继续申请
                            }
                        })
                        .setNegativeButton("不给", new DialogInterface.OnClickListener() {
                            @Override
                            public void onClick(DialogInterface dialog, int which) {
                                dialog.cancel();
                                // 用户拒绝申请
                            }
                        }).show();
            } else {
                // 申请授权。
                ActivityCompat.requestPermissions(this, new
                          String[]{Manifest.permission.ACCESS_COARSE_LOCATION}, Location_Permission);
            }
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == Location_Permission) {
            if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                //权限被用户同意,做相应的事情
            } else {
                //权限被用户拒绝，做相应事情
            }
        }
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    }
}
```

91、 应用程序启动过程分析（链接）

在 Android 系统中，有两种操作会引发 Activity 的启动：

1) 用户点击应用程序图标时，Launcher 会为我们启动应用程序的主 Activity；

2) 应用程序的默认 Activity 启动起来后，它又可以在内部通过调用 startActvity 接口启动新的 Activity。

Activity 的启动方式：一种是显式的，一种是隐式的。

两种 Activity 的启动方式要借助于应用程序框架层的 ActivityManagerService 服务进程。ActivityManagerService 不但负责启动 Activity 和 Service，还负责管理 Activity 和 Service。

根 Activity 的启动过程：

在 Android 系统中，所有的 Activity 组件都保存在一个堆栈中，其中，后面启动的 Activity 组件位于前面启动的 Activity 组件的上面。 用户在执行一个操作时，通常需要与一系列相关的 Activity 组件进行交互，这些相关的 Activity 组件在系统的 Activity 堆栈中用一个任务（Task）来描述。

Mainctiity 组件是由 Launcher 组件来启动的，而 Launcher 组件又是通过 Activity 管理服务 Activily ManagerService 来启动 MainActivity 组件的。由于 MainActivity 组件，Launcher 组件和 Activity ManagerService 是分别运行在不同的进程中的。因此，MainActivity 组件的启动过程就涉及到了三个进程。这三个进程是通过 Binder 进程间通信机制来完成 MainActivity 组件的启动过程的。

启动的过程：

1) **Launcher 组件向 ActivityManagerService 发送一个启动 MainActivity 组件的进程间通信请求（Binder 通信）。**

Launcher.startActivitySafely -> Activity.startActivity -> Activity.startActivityForResult -> Instrumentation.execStartActivity -> ActivityManagerProxy.startActivity -> ActivityManagerService.startActivity

a) Launcher.startActivitySafely：Laucher 组件如何获取 Intent 中所包含的启动 Activity 的信息：action-“android.intent.action.MAIN”和 category-“android.intent.category.LAUNCHER”？PackageManagerService 安装应用程序的过程中，会对它的配置文件 AndroidManifest.xml 进行解析，从而得到它里面的组件信息。系统在启动完成之后，就会将 Launcher 组件启动起来，Laucher 组件在启动过程中，会向 PackageManagerService 查询所有 Action 名称等于 Intent.ACTION MAIN，并且 Category 名称等于 Intent.CATEGORY_LAUNCHER 的 Activity 组件，最后为每一个 Activity 组件创建一个快捷图标，并且将它们的信息与各自的快捷图标关联起来，以便用户点击它们时可以将对应的 Activity 启动。

b) **mInstrumentation：**Activity 类的成员变量，类型是 Intrumentation，用来监控应用程序和系统的交互操作。
mMainThread：Activity 类的成员变量，类型为 ActivityThread，用来描述一个应用程序进程。系统每当启动一个应用程序进程时，都会在它里面加载一个 ActivityThread 类实例，并且会将这个 ActivityThread 类实例保存在每一个在该进程中启动的 Activity 组件的父类 Activity 的成员变量 mMainThread 中。ActivityThread 类的成员函数 getApplicationThread 用来获取它内部的一个类型为 ApplicationThread 的 Binder 本地对象。这里将它传递给 ActivityManagerService，这样 ActivityManagerService 接下来就可以通过它来通知 Launcher 组件进入 Paused 状态。
mToken：Activity 类的成员变量，类型为 IBinder，它是一个 Binder 代理对象，指向了 ActivityManagerService 中一个类型为 ActivityRecord 的 Binder 本地对象。每一个已经启动的 Activity 组件在 ActivityManagerService 中都有一个对应的 ActivityRecord 对象，用来维护对应的 Activity 组件的运行状态以及信息。这里将它传递给 ActivityManagerService，这样 ActivityManagerService 接下来就可以获得 Launcher 组件的详细信息。

c) Instrumentation.execStartActivity：ActivityManagerNative.getDefault 返回 ActivityManagerService 的代理对象，即 ActivityManagerProxy 接口。

d) ActivityManagerProxy.startActivity：通过 ActivityManagerProxy 内部的一个 Binder 的代理对象 mRemote 向 ActivityManagerService 发起一个类型为 START_ACTIVITY_TRANSACTION 的进程间通信。传递的参数：
参数 caller：指向 Launcher 组件所运行在的应用程序进程的 ApplicationThread 对象；
参数 intent：即将要启动的 MainActivity 组件的信息；
参数 resultTo：指向 ActivityManagerService 内部的一个 ActivityRecord 对象，里面保存了 Launcher 组件的详细信息。

2) **ActivityManagerService 首先将要启动的 MainActivity 组件的信息保存下来，然后再向 Launcher 组件发送一个进人中止状态的进程间通信请求。**

ActivityStack.startActivityMayWait -> ActivityStack.startActivityLocked -> ActivityStack.startActivityUncheckedLocked -> ActivityStack.resumeTopActivityLocked -> ActivityStack.startPausingLockedA –> pplicationThreadProxy.schedulePauseActivity -> ApplicationThread.schedulePauseActivity -> ActivityThread.queueOrSendMessage -> H.handleMessage -> ActivityThread.handlePauseActivity ->

a) ActivityStack.startActivityMayWait：到 PackageManagerService 中解析参数 intent 的内容，得到 MainActivity 的相关信息，保存在 aInfo 变量中，来作为准备要启动的 Activity 的相关信息。

b) ActivityStack.startActivityLocked：ActivityStack 类的成员变量 mHIstory，用来描述系统的 Activity 组件堆栈。在这个堆栈中，每一个已经启动的 Activity 组件都使用一个 ActivityRecord 对象来描述。ActivityStack 类中得到了请求 ActivityManagerService 执行启动 Activity 组件操作的源 Activity 组件，以及要启动的目标 Activity 组件的信息。它们分别保存在 ActivityRecord 对象 sourceRecord 和 r 中。

c) ActivityStack.startActivityUncheckedLocked：在默认情况下，目标 Activily 组件是与源 Activily 组件运行在同一个任务中的。然而，如果源 Activity 组件将目标 Activity 组件的启动标志值的 FLAG_ACTIVITY_NEW_TASK 位设置为 1，并且源 Activily 组件不需要知道目标 Activity 组件的运行结果，那么 ActivityManagerService 就会将目标 Activity 组件运行在另外一个不同的任务中。这个不同的任务可能是一个新创建的任务，也可能是一个已经存在的任务。
这里会为目标 Activity 组件创建一个专属任务，并且将它保存在参数 r 的成员变量 task 中。接着将新创建的专属任务交给 ActivityManagerServier 来管理。

d) ActivityStack.resumeTopActivityLocked：首先看要启动的 Activity 是否就是当前处理 Resumed 状态的 Activity，如果是的话，那就什么都不用做，直接返回就可以了；否则再看一下系统当前是否休眠状态，如果是的话，再看看要启动的 Activity 是否就是当前处于堆栈顶端的 Activity，如果是的话，也是什么都不用做。上面两个条件都不满足的话，首先要把当前处于 Resumed 状态的 Activity 推入 Paused 状态，然后才可以启动新的 Activity。但是在将当前这个 Resumed 状态的 Activity 推入 Paused 状态之前，首先要看一下当前是否有 Activity 正在进入 Pausing 状态，如果有的话，当前这个 Resumed 状态的 Activity 就要稍后才能进入 Paused 状态了，这样就保证了所有需要进入 Paused 状态的 Activity 串行处理。

e) ActivityStack.startPausingLocked：通过一个类型为 ApplicationThread 的 Binder 本地对象，向 Launcher 组件所在的应用程序进程发送一个终止 Launcher 组件的通知，以便 Launcher 有机会执行一些数据保存的操作。

f) ApplicationThreadProxy.schedulePauseActivity：ApplicationThreadProxy 类内部的一个 Binder 代理对象 mRemote 向 Launcher 组件所在的应用程序进程发送一个类型为 SCHEDULE_PAUSE_ACTIVITY_TRANSACTION 的进程间通讯请求。

g) ApplicationThread.schedulePauseActivity：处理类型为 SCHEDULE_PAUSE_ACTIVITY_TRANSACTION 的进程间通讯请求。

h) ActivityThread.queueOrSendMessage：将相关信息组装成一个 msg，然后通过 mH 成员变量发送出去，mH 的类型是 H，继承于 Handler 类，是 ActivityThread 的内部类，因此，这个消息最后由 H.handleMessage 来处理。

i) ActivityThread.handlePauseActivity：将 Binder 引用 token 转换成 ActivityRecord 的远程接口 ActivityClientRecord，然后做了三个事情：1. 如果 userLeaving 为 true，则通过调用 performUserLeavingActivity 函数来调用 Activity.onUserLeaveHint 通知 Activity，用户要离开它了；2. 调用 performPauseActivity 函数来调用 Activity.onPause 函数，在 Activity 的生命周期中，当它要让位于其它的 Activity 时，系统就会调用它的 onPause 函数；3. 调用 QueueWork 类的静态成员函数 waitToFinish 等待完成前面的数据写入操作。
最后，通知 ActivityManagerService，这个 Activity 已经进入 Paused 状态了，ActivityManagerService 现在可以完成未竟的事情，即启动 MainActivity 了。

3) **Launcher 组件进入到中止状态之后，就会向 ActivityManagerService 发送一个已进入中止状态的进程间通信请求，以便 ActivityManagerService 可以继续执行启动 Mainactivitya 组件的操作。**
ActivityManagerProxy.activityPaused -> ActivityManagerService.activityPaused -> ActivityStack.activityPaused -> ActivityStack.completePauseLocked -> ActivityStack.resumeTopActivityLokced

a) ActivityManagerProxy.activityPaused：ActivityManagerProxy 类内部的一个 Binder 代理对象 mRemote 向 ActivityManagerService 发送一个类型为 ACTIVITY_PAUSE_TRANSACTION 的进程间通讯请求。

b) ActivityManagerService.activityPaused：处理进程间通讯请求。

c) ActivityStack.activityPaused：Launcher 组件已经进入 Pause 状态。

d) ActivityStack.resumeTopActivityLokced：当前在堆栈顶端的 Activity 是即将要启动的 MainActivity，这里通过调用 topRunningActivityLocked 将它取回来，保存在 next 变量中。之前最后一个 Resumed 状态的 Activity，即 Launcher，到这里已经处于 Paused 状态了，因此，mResumedActivity 为 null。

4) **ActivityManagerService 发现用来运行 Main Activity 组件的应用程序进程不存在，因此，它就会先启动一个新的应用程序进程。**
ActivityStack.startSpecificActivityLocked -> ActivityManagerService.startProcessLocked

a) ActivityStack.startSpecificActivityLocked：取回来的 app 为 null。在 Activity 应用程序中的 AndroidManifest.xml 配置文件中，我们没有指定 Application 标签的 process 属性，系统就会默认使用 package 的名称。每一个应用程序都有自己的 uid。ActivityManagerService 在启动一个 Activity 组件时，首先会以它的 uid 和进程名称来检查系统中是否存在一个对应的应用程序进程。如果存在，就会直接通知这个应用程序进程将该 Activity 组件启动起来；否则，就会先以这个用户 ID 和进程名称来创建一个应用程序进程，然后再通知这个应用程序进程将该 Activity:组件启动起来。
这里为第一次启动，所以不可能存在所需的应用程序进程。

b) ActivityManagerService.startProcessLocked：再次检查是否已经有以 process + uid 命名的进程存在，在当前情景中，返回值 app 为 null，因此，后面会创建一个 ProcessRecord，并存保存在成员变量 mProcessNames 中，最后调用另一个 startProcessLocked 函数。这里主要是调用 Process.start 接口来创建一个新的进程，新的进程会导入 android.app.ActivityThread 类，并且执行它的 main 函数。

5) **新的应用程序进程启动完成之后，就会向 ActivityManagerService 发送一个启动完成的进程间通信请求，以便 ActivityManagerService 可以继续执行启动 MainActiviyt 组件的操作。**
ActivityThread.main -> ActivityManagerProxy.attachApplication -> ActivityManagerService.attachApplication

a) ActivityThread.main：新的应用程序进程在启动时,主要做了两件事情。
第一件事情是在进程中创建一个 ActivityThread 对象，并且调用它的成员函数 attach 向 ActivityManagerService 发送一个启动完成通知。在 ActivityThread 类的成员函数 attach 中，调用 ActivityManagerNative 类的静态成员函数 getDefault 来获得 ActivityManagerService 的一个代理对象，调用它的成员函数 attachApplication 向 ActivityManagerService 发送一个进程间通信请求，并且将前面所创建的 ApplicationThread 对象传递给 ActivityManagerService。
第二件事情是调用 Looper 类的静态成员函数 prepareMainLooper 创建一个消息循环，并且在 ActivityManagerService 发送启动完成通知之后，使得当前进程进入到这个消息循环中。

b) ActivityManagerProxy.attachApplication：ActivityManagerProxy 类内部的一个 Binder 代理对象 mRemote 向 ActivityManagerService 发送一个类型为 ATTACH_APPLICATION_TRANSACTION 的进程间通讯请求。

c) ActivityManagerService.attachApplication：接收到新的应用程序进程发送过来的类型为 ATTACH_APPLICATION_TRANSACTION 的进程间通讯请求之后，就知道新的应用程序进程已经启动。、

6) **ActivityManagerservice 将第 2 步保存下来的 MainActivity 组件的信息发送给第 4 步创建的应用程序进程，以便它可以将 MainActivity 组件启动起来。**
ActivityManagerService.attachApplicationLocked –> ActivityStack.realStartActivityLocked -> ApplicationThreadProxy.scheduleLaunchActivity -> ApplicationThread.scheduleLaunchActivity -> ActivityThread.queueOrSendMessage -> H.handleMessage -> ActivityThread.handleLaunchActivity -> ActivityThread.performLaunchActivity -> MainActivity.onCreate

a) ActivityManagerService.attachApplicationLocked：ProcessRecord 对象 app 用来描述新创建的应用程序进程，将它的成员变量 thread 设置为参数 thread 所指向的一个 ApplicationThread 代理对象。ActivityManagerService 就可以通过这个 ApplicationThread 代理对象来和新创建的应用程序进程进行通信。
得到位于 Activit 组件堆栈顶端的一个 ActivityRecord 对象 hr（即将要启动的 MainActivity 组件），接着检查这个 Activity 组件的用户 D 和进程名称是否与 ProcessRecord 对象 app 所描述的应用程序进程的用户 ID 和进程名称一致。如果一致，那么就说 ActivityRecord 对象 hr 所描述的 Activity 组件是应该在 ProcessRecord 对象 app 所描述的应用程序进程中启动的。

b) ActivityStack.realStartActivityLocked：将 Activity 组件添加到 app 所描述的应用程序进程的 Activity 组件列表中

c) ApplicationThreadProxy.scheduleLaunchActivity：ApplicationThreadProxy 类内部的一个 Binder 代理对象 mRemote 向前面创建的应用程序进程发送一个类型为 SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION 的进程间通讯请求。

d) ApplicationThread.scheduleLaunchActivity：创建一个 ActivityClientRecord 实例，并且初始化它的成员变量，然后调用 ActivityThread 类的 queueOrSendMessage 函数进一步处理。

e) ActivityThread.queueOrSendMessage：函数把消息内容放在 msg 中，然后通过 mH 把消息分发出去，最后会调用 H 类的 handleMessage 函数。

f) ActivityThread.handleLaunchActivity：首先调用 performLaunchActivity 函数来加载这个 Activity 类，即 MainActivity，然后调用它的 onCreate 函数，最后回到 handleLaunchActivity 函数时，再调用 handleResumeActivity 函数来使这个 Activity 进入 Resumed 状态，即会调用这个 Activity 的 onResume 函数，这是遵循 Activity 的生命周期的。

g) ActivityThread.performLaunchActivity：首先收集要启动的 Activity 的相关信息，主要 package 和 component 信息；然后通过 ClassLoader 将 MainActivity 类加载进来；接下来是创建 Application 对象，这是根据 AndroidManifest.xml 配置文件中的 Application 标签的信息来创建的；紧接着创建 Activity 的上下文信息，并通过 attach 方法将这些上下文信息设置到 MainActivity 中去；最后还要调用 MainActivity 的 onCreate 函数（通过 mInstrumentation 的 callActivityOnCreate 函数来间接调用， mInstrumentation 在这里的作用是监控 Activity 与系统的交互操作，相当于是系统运行日志）。

92、 子 Activity 组件在进程内的启动过程

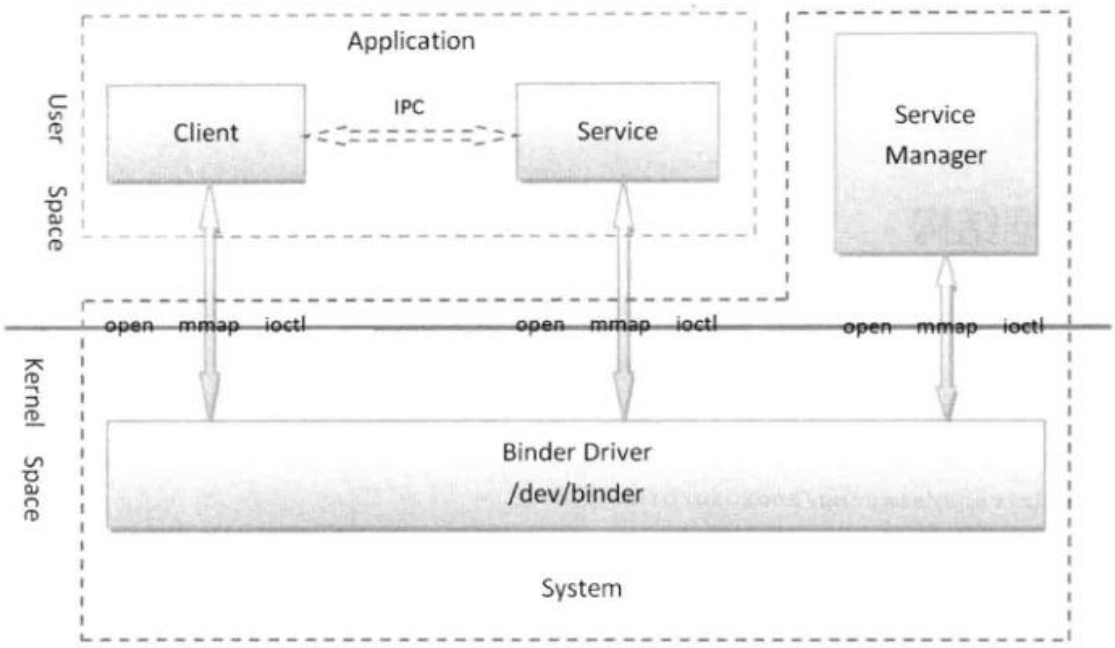
1) MainActivity 组件向 ActivityManagerService 发送一个启动 SubActivityInProcess 组件的进程间通信请求；
2) ActivityManagerService 首先将要启动的 SubActivityInProcess 组件的信息保存下来，然后再向 MainActivity 组件发送一个进入中止状态的进程间通信请求；
3) MainActivity 组件进入到中止状态之后，就会向 ActivityManagerService 发送一个已进入中止状态的进程间通信请求，以便 ActivityManagerService 可以继续执行启动 SubActivityInProcess 组件的操作；
4) ActivityManagerService 发现用来运行 SubActivityInProcess 组件的应用程序进程已经存在，因此它就会将第 2 步保存下来的

SubActivityInProcess 组件信息发送给该应用程序进程，以便它可以将 SubActivityInProcess 组件启动起来。

93、 子 Activity 组件在新进程中的启动过程

1) MainActivity 组件向 ActivityManagerService 发送一个启动 SubAcuivityInNewProcess 组件的进程间通信请求；
2) ActivityManagerService 首先将要启动的 SubAcuivityInNewProcess 组件的信息保存下来，然后再向 MainActivity 组件发送一个进入中止状态的进程间通信请求；
3) MainActivity 组件进入到中止状态之后，就会向 ActivityManagerService 发送一个已进入中止状态的进程间通信请求，以便 ActivityManagerService 可以继续执行启动 SubActivityInNewprocess 组件的操作；
4) ActivityManagerService 发现用来运行 SubActivityInNewProcess 组件的应用程序进程不存在，因此它就会先启动一个新的应用程序进程；
5) 新的应用程序进程启动完成之后，就会向 ActivityManagerService 发送一个启动完成的进程间通信请求，以便 ActivityManagerService 可以继续执行启动 SubActivityInNewProcess 组件的操作；
6) ActivityManagerService 将第 2 步保存下来的 SubActivityInNewProcess 组件信息发送给第 4 步创建的应用程序进程，以便它可以将 SubActivityInNewProcess 组件启动起来。

94、 Binder 进程间通讯机制（Native）



- **基本函数介绍：**

函数 open：打开设备文件/dev/binder 来获得一个文件描述符，之后通过文件描述符来和 Binder 驱动程序交互，继而和其他进程执行 Binder 进程间通信。

函数 mmap：把设备文件映射到进程的地址空间，然后才可以使用 Binder 进程间通信。

函数 ioctl：设备驱动程序中对设备的 I/O 通道进行管理的函数，控制 I/O 设备，提供了一种获得设备信息和向设备发送控制参数的手段。

- **基本类描述：**

Service 组件和 Client 组件分别使用模板类 BnInterface 和 BpInterface 来描述，前者称为 Binder 本地对象，后者称为 Binder 代理对象。**Binder 本地对象（BBinder）和 Binder 代理对象（BpBinder）分别对应于 Binder 驱动程序中的 Binder 实体对象（binder_node）和 Binder 引用对象（binder_ref）。**

BnInterface：继承了 **BBinder** 类，BBinder 类为 Binder 本地对象提供了抽象的进程间通信接口。

BBinder：继承了 **IBinder** 类，而 IBinder 类又继承了 RefBase 类。**继承了 Refbase 类的子类的对象均可以通过强指针和弱指针来维护它们的生命周期。**BBinder 有两个重要的成员函数 transact 和 onTransact：

1) transact：当一个 Binder 代理对象通过 Binder 驱动程序向一个 Binder 本地对象发出一个进程间通信请求时，Binder 驱动程序就会调用该 Binder 本地对象的成员函数 transact 来处理该请求。
2) onTransact：由 BBinder 的子类，即 Binder 本地对象类来实现的。它负责分发与业务相关的进程间通信请求。事实上，与业务相关的进程间通信请求是由 Binder 本地对象类的子类，即 Service 组件类来负责处理的。

BpInterface：继承了 **BpRefBase** 类，而 BpRefBase 类为 Binder 代理对象提供了抽象的进程间通信接口。

BpRefBase：继承了 **RefBase** 类，其内部的成员变量 rEmote，它指向一个 **BpBinder** 对象，可以通过成员函数 remote 来获取。

BpBinder：实现了 BpRefBase 类的进程间通信接口。BpBinder 有两个重要的成员函数 handle 和 transact：

1) handle：可以获取 Client 组件的句柄值。每一个 Client 组件在 Binder 驱动程序中都对应有一个 Binder 引用对象，而每一个 Binder 引用对象都有一个句柄值，其中，Client 组件就是通过这个句柄值来和 Binder 驱动程序中的 Binder 引用对象建立对应关系的。
2) transact：用来向运行在 Server 进程中的 Service 组件发送进程间通信请求，这是通过 Binder 驱动程序间接实现的。BpBinder 类的成员函数 transact 会把 BpBinder 类的成员变量 mHandle，以及进程间通信数据发送给 Binder 驱动程序，这样 Binder 驱动程序就能够根据这个句柄值来找到对应的 Binder 引用对象（**binder_ref**），继而找到对应的 Binder 实体对象（**binder_node**），最后就可以将进程间通信数据发送给对应的 Service 组件了。

IPCThreadState：**BBinder** 和 **BpBinder** 通过它来和 Binder 驱动程序交互。每一个 Binder 线程的内部都有一个 IPCThreadState 对象。IPCThreadState 的成员函数 transact 和成员变量 mProcess：

1) transact：用来和 Binder 驱动程序交互，内部实现是 talkWithDriver，它一方面负责向 Binder 驱动程序发送进程间通信请求，另一方面又负责接收来自 Binder 驱动程序的进程间通信请求。
2) mProcess：指向一个 **ProcessState** 对象，其负责初始化 Binder 设备，即打开设备文件/dev/binder（open），以及将设备文件/dev/binder 映射到进程的地址空间（mmap）。ProcessState 对象在进程范围内是唯一的，因此，Binder 线程池中的每一个线程都可以通过它来和 Binder 驱动程序建立连接。

- **Biner 实体对象、Binder 引用对象、Binder 本地对象和 Binder 代理对象的交互过程**

1) 运行在 Client 进程中的 Binder 代理对象通过 Binder 驱动程序向运行在 Server 进程中的 Binder 本地对象发出一个进程间通信请求，Binder 驱动程序接着就根据 Client 进程传递过来的 Binder 代理对象的句柄值来找到对应的 Binder 引用对象。
2) Binder 驱动程序根据前面找到的 Binder 引用对象找到对应的 Binder 实体对象，并且创建一个事务(binder transaction)来描述该次进程间通信过程。
3) Binder 驱动程序根据前面找到的 Binder 实体对象来找到运行在 Server 进程中的 Binder 本地对象，并且将 Client 进程传递过来的通信数据发送给它处理。
4) Binder 本地对象处理完成 Client 进程的通信请求之后，就将通信结果返回给 Binder 驱动程序，Binder 驱动程序接着就找到前面所创建的一个事务。
5) Binder 驱动程序根据前面找到的事务的相关属性来找到发出通信请求的 Client 进程，并且通知 Client 进程将通信结果返回给对应的 Binder 代理对象处理。

- **ServiceManager**

Binder 进程间通信机制上下文管理者(Context manager)，同时负责管理系统中的 Service 组件，并且向 Client 组件提供获 Service 代理对象的服务。ServiceManager 由 init 进程负责启动，它们均在系统启动时启动。

启动过程：

1) 调用函数 binder_open 打开设备文件/dev/binder，以及将它映射到本进程的地址空间；
2) 调用函数 binder_become_context_manager 将自己注册为 Binder 进程间通信机制的上下文管理者；
3) 调用函数 binder_loop 来循环等待和处理 Client 进程的通信请求。

获取代理对象：

ServiceManager 代理对象的类型为 BpServiceManager，它用来描述一个实现了 IServiceManager 接口的 Client 组件。

IServiceManager 接口四个成员函数：

1) getService 和 checkService：用来获取 Service 组件的代理对象；
2) addService：用来注册 Service 组件；
3) listService：用来获取注册在 ServiceManager 中的 Service 组件名称列表。

defaultServiceManager：

采用单例模式，来获取 ServiceManager 对象，当 gDefaultServiceManager 存在，则直接返回，否则创建一个新对象。

```
1    gDefaultServiceManager = interface_cast<IServiceManager>(
2                    ProcessState::self()->getContextObject(NULL));
```

全局变量 gDefaultServiceManager 是一个类型为 ServiceManager 的强指针，它指向进程内的一个 BpServiceManager 对象，即一个 ServiceManager 代理对象。

1) ProcessState::self()：调 ProcessState 类的静态成员函数 self 获得进程内的一个 ProcessState 对象（也是单例模式）。

2) getContextObject()：创建 Binder 代理对象（BpBinder），对于 handle=0 的 BpBinder 对象，存在则直接返回，不存在才创建。

3) interface_cast<IServiceManager>()：模板函数，等价于 IServiceManager::asInterface()，用于获取 BpServiceManager 对象。IServiceManager::asInterface() 等价于 new BpServiceManager(BpBinder)。

BpServiceManage（用于和 ServiceManager 进程通信）：

1) 通过继承接口 IServiceManager 实现了接口中的业务逻辑函数；

2) 通过成员变量 mRemote= new BpBinder(0) 进行 Binder 通信工作。

3) BpBinder 通过 handler 来指向所对应 BBinder，在整个 Binder 系统中 handle=0 代表 ServiceManager 所对应的 BBinder。

- **Service 组件**

Service 组件是在 Server 进程中运行的。Server 进程启动时，会先将它里面的 Service 组件注册到 ServiceManager 中去，接着再启动一个 Binder 线程池来等待和处理 Client 进程的通信请求。

注册 Service 组件：

在将 Service 组件注册到 ServiceManager 之前，需要先获得一个 ServiceManager 代理对象（BpServiceManage），之后通过它将 Service 组件注册到 ServiceManager 中。

```
1    void MediaPlayerService::instantiate() {
2        defaultServiceManager()->addService(String16("media.player"), new MediaPlayerService());
3    }
```

Client 进程和 Server 进程的一次进程间通信过程：

1) Client 进程将进程间通信数据封装成一个 Parcel 对象，以便可以将进程间通信数据传递给 Binder 驱动程序。

2) Client 进程向 Binder 驱动程序发送一个 BC_TRANSACTION 命令协议。Binder 驱动程序根据协议内容找到目标 Server 进程之后，就会向 Client 进程发送一个 BR_TRANSACTION_COMPLETE 返回协议，表示它的进程间通信请求已经被接受。Client 进程接收到 Binder 驱动程序发送给它的 BR_TRANSACTION_COMPLETE 返回协议，并且对它进行处理之后，就会再次进入到 Binder 驱动程序中去等待目标 Server 进程返回进程间通信结果。

3) Binder 驱动程序在向 Client 进程发送 BR_TRANSACTION_COMPLETE 返回协议的同时，也会向目标 Server 进程发送一个 BR_TRANSACTION 返回协议，请求目标 Server 进程处理该进程间通信请求。

4) Server 进程接收到 Binder 驱动程序发来的 BR_TRANSACTION 返回协议，并且对它进行处理之后，就会向 Binder 驱动程序发送一个 BC_REPLY 命令协议。Binder 驱动程序根据协议内容找到目标 Client 进程之后，就会向 Server 进程发送一个 BR_TRANSACTION_COMPLETE 返回协议，表示它返回的进程间通信结果已经收到了。Server 进程接收到 Binder 驱动程序发送给它的 BR_TRANSACTION_COMPLETE 返回协议，并且对它进行处理之后，一次进程间通信过程就结束了。接着它会再次进入到 Binder 驱动程序中去等待下一次进程间通信请求。

5) Binder 驱动程序向 Serve 进程发送 BR_TRANSACTION_COMPLETE 返回协议的同时，也会向目标 Client 进程发送一个 BR_REPLY 返回协议，表示 Server 进程已经处理完成它的进程间通信请求了，并且将进程间通信结果返回给它。

启动 Binder 线程池：

一个进程是通过调用其内部的 ProcessState 对象的成员函数 startThreadPool 来启动一个 Binder 线程池的。

```
1    int main(int argc, char** argv)
2    {
3        sp<ProcessState> proc(ProcessState::self());      // 获得一个 ProcessState 实例
4        sp<IServiceManager> sm = defaultServiceManager(); // 得到一个 ServiceManager 对象
5        MediaPlayerService::instantiate();                // 初始化 MediaPlayerService 服务
6        ProcessState::self()->startThreadPool();          // 启动 Binder 线程池
7        IPCThreadState::self()->joinThreadPool();         // 将当前线程加入到已启动 Binder 线程池中去等待和
8                                                          // 处理来自 Client 进程的进程间通信请求
9    }
```

一个 Binder 线程的生命周期：

1) 将自己注册到 Binder 线程池中；

2) 在一个无限循环中不断地等待和处理进程间通信请求；

3) 退出 Binder 线程池。

- **Service 代理对象的获取过程**

Service 组件将自己注册到 ServiceManager 中之后，它就在 Server 进程中等待 Client 进程将进程间通信请求发送过来。Client 进程为了和运行在 Server 进程中的 Service 组件通信，首先要获得它的一个代理对象，这是通过 ServiceManager 提供的 Service 组

件查询服务来实现的。

为了创建一个 FregService 代理对象（这里假设 FregService 为要交互的服务端），即一个 BpFregService 对象，首先要通过 Binder 驱动程序来获得一个引用了运行在 FregServer 进程中的 FregService 组件的 Binder 引用对象的句柄值，然后再通过这个句柄值创建一个 Binder 代理对象，即一个 BpBinder 对象，最后将这个 Binder 代理对象封装成一个 FregService 代理对象。

ServiceManager 代理对象的成员函数 getService 提供了获取一个 Service 组件的代理对象的功能，而 ServiceManager 代理对象可以通过 Binder 库提供的函数 defaultServiceManager 来获得。在调用 ServiceManager 代理对象的成员函数 getService 来获得一个 Service 组件的代理对象时，需要指定这个 Service 组件注册到 ServiceManager 中的名称。

ServiceManager 代理对象的成员函数 getService 的 Binder 进程间通信过程：

1) FregClient 进程将进程间通信数据，即要获得其代理对象的 Service 组件 FregService 的名称，封装在一个 Parcel 对象中，用来传递给 Binder 驱动程序。
2) FregClient 进程向 Binder 驱动程序发送一个 BC_TRANSACTION 命令协议，Binder 驱动程序根据协议内容找到 ServiceManager 进程之后，就会向 FregClient 进程发送一个 BC_TRANSACTION_COMPLETE 返回协议，表示它的进程间通信请求已经被接受。FregClient 进程接收到 Binders 驱动程序发送给它的 BC_TRANSACTION_COMPLETE 返回协议，并且对它进行处理之后，就会再次进入到 Binder 驱动程序中去等待 ServiceManager 进程将它要获取的 Binder 代理对象的句柄值返回来。
3) Binder 驱动程序在向 FregClien 进程发送 BC_TRANSACTION_COMPLETE 返回协议的同时，也会向 ServiceManager 进程发送一个 BR_TRANSACTION 返回协议，请求 ServiceManager 进程执行一个 CHECK_SERVICE_TRANSACTION 操作。
4) ServiceManager 进程执行完成 FregClient 进程请求的 CHECK_SERVICE_TRANSACTION 操作之后，就会向 Binder 驱动程序发送一个 BC_REPLY 命令协议，协议内容包含了 Service 组件 FregService 的信息。Binder 驱动程序根据协议内容中的 Service 组件 FregService 的信息为 FregClient 进程创建一个 Binder 引用对象，接着就会向 ServiceManager 进程发送一个 BC_TRANSACTION_COMPLETE 返回协议，表示它返回的 Service 组件 FregService 的信息已经收到了。ServiceManager 进程接收到 Binder 驱动程序发送给它的 BC_TRANSACTION_COMPLETE 返回协议，并且对它进行处理之后，一次进程间通信过程就结束了，接着它会再次进入到 Binder 驱动程序中去等待下一次进程间通信请求。
5) Binder 驱动程序在向 ServiceManager 进程发送 BC_TRANSACTION_COMPLETE 返回协议的同时，也向 FregClient 进程发送一个 BR_REPLY 返回协议，协议内容包含了前面所创建的一个 Binder 引用对象的句柄值，这时候 FregClient 进程就可以通过这个句柄来创建一个 Binder 代理对象。

95、 Binder 进程间通讯机制（Java 接口）

- **ServiceManager 的 Java 代理对象的获取**

ServiceManager 的 Java 代理对象的类型为 ServiceManagerProxy，其实现了 IServiceManager 接口，ServiceManager 继承于 IInterface 接口，有四个成员函数：

1) getService 和 checkService：用来获取 Java 服务的代理对象；
2) addService：用来注册 Java 服务；
3) listService：用来获取注册在 ServiceManager 中的 Java 服务名称列表。

ServiceManager 的 Java 代理对象的成员变量 mRemote：类型为 Binder，指向的是一个 BinderProxy 对象。BinderProxy 类用来描述一个 Java 服务代理对象，它实现了 IBinder 接口。

BinderProxy 类的成员变量 mObject：指向 C++层中的一个 Binder 代理对象，通过这里将 java 服务代理对象和 c++层的 Binder 代理对象关联。

ServiceManager：ServiceManager 的 Java 代理对象是由它创建。

1) **静态成员变量 sServiceManager：**类型为 ServiceManagerProxy，实现了其它静态成员函数 getService、checkService、addService 和 listService。
2) **静态成员函数 getIServiceManager：**用来创建一个 ServiceManager 的 Java 代理对象并保存在静态成员变量 **sServiceManager** 中。在创建的过程中，会通过 ServiceManagerNative 类的静态成员函数 asInterface 将一个句柄值等于 0 的 Java 服务代理对象封装成一个 ServiceManagerProxy 对象。

ServiceManagerNative：

类似于 C++层中的 BnServiceManager 类，其继承了 Java 层中的 Binder 类，用来实现 Java 层中的 ServiceManager 服务。同样，Java 层中的 Binder 类类似于 C++层中的 BBinder 类，是用来实现一个 Java 服务的。Java 层中的 Binder 类内部有一个类型为 int 的成员变量 mObject，它指向 C++层中的一个 Binder 本地对象。

- **Java 服务接口的定义和解析**

通过 AIDL 来定义 Java 服务接口。

- **Java 服务的启动过程**

Java 服务一般是运行在 Android 系统进程 System 或者 Android 应用程序进程中的，它们在启动之前，同样需要将自己注册到 ServiceManager 中，以便 Client 进程可以通过 ServiceManager 来获得它们的 Java 服务代理对象。

由于 Android 系统进程 System 和 Android 应用程序进程在启动时会在内部启动一个 Binders 线程池，因此，运行在它们里面的 Java 服务在启动时，只需要将自己注册到 ServiceManager 中就可以了。

- **Java 服务代理对象的获取过程**

Java 服务将自己注册到 ServiceManager 中之后，Android 应用程序就可以通过 ServiceManager 来获得它的一个 Java 服务代理对象了。有了这个 Java 服务代理对象之后，Android 应用程序就可以使用相应的 Java 服务了。

首先通过 ServiceManager 类的静态成员函数 getService 来获得一个名称为指定名称的 Java 服务代理对象，接着再调用 IFregService.Stub 类的静态成员函数 asInterface 将这个 Java 服务代理对象封装成一个实现了 IFregService 接口的 Java 服务代理对象。

- **Java 服务的调用过程**

参考 AIDL 的 Client 调用 Service 的过程。

96、 SoundPool 和 MediaPlayer 的区别

SoundPool：适合短促且对反应速度比较高的情况(游戏音效或按键声等)。

MediaPlayer：适合比较长且时间要求不高的情况。

- **SoundPool**

SoundPool 类是 Android 用于管理和播放应用程序音频资源的类。

1) SoundPool 的创建：

```
1    public SoundPool(int maxStream, int streamType, int srcQuality)
```

maxStream：同时播放的流的最大数量。如果其数量超过流的最大数目，SoundPool 会基于优先级自动停止先前播放的流。

streamType：流的类型，一般为 STREAM_MUSIC。

srcQuality：采样率转化质量，当前无效果，使用 0 作为默认值。

2) 加载音频资源：

流的加载过程是 MediaPlayer 服务将音频解压为原始 16 位 PCM 数据的过程，该过程为异步的，所以初始化后不能立即播放，需要等待一点时间，也因此在播放时不需要忍受解压缩带来的延迟。

```
1    int load(AssetFileDescriptor afd, int priority)    // 通过一个 AssetFileDescriptor 对象
2    int load(Context context, int resId, int priority) // 通过一个资源 ID
3    int load(String path, int priority)                // 通过指定的路径加载
4    int load(FileDescriptor fd, long offset, long length, int priority) // 通过 FileDescriptor 加载
```

参数 priority：优先级运行是从低到高。数字越大优先级越高。

返回值：一个非 0 的 soundID，用于播放时指定特定的音频。

3) 播放控制：

```
1    final int play(int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)
```

播放指定音频的音效，并返回一个 streamID 。

参数 soundID：load 函数返回的声音 ID。

参数 leftVolume 和 rightVolume：左右声道音量设置(range = 0.0 to 1.0)。

参数 priority：流的优先级，值越大优先级高，当同时播放数量超出了最大支持数时，SoundPool 对该流的处理；

参数 loop：循环播放的次数，0 为值播放一次，-1 为无限循环，其他值为播放 loop+1 次。

参数 rate：播放的速率，范围 0.5-2.0(0.5 为一半速率，1.0 为正常速率，2.0 为两倍速率)。

```
1    final void pause(int streamID)    // 暂停指定播放流的音效
2    final void resume(int streamID)   // 继续播放指定播放流的音效
3    final void stop(int streamID)     // 终止指定播放流的音效
```

4) 释放资源：

```
final boolean unload(int soundID)    // 卸载一个指定的音频资源
final void release()                 // 释放 SoundPool 中的所有音频资源
```

- **MediaPlayer**

用于控制播放音频/视频文件和流。

1) MediaPlayer 的创建：

```
MediaPlayer mp = new MediaPlayer();
```

当所播放的音频文件在 sd 卡中或者是网络音频等的时候，需要通过 setDataSource 设置需要播放的音频，使用这种方式新建的对象需要先使播放器进入 prepare 状态。

```
MediaPlayer mp = MediaPlayer.create(this, R.raw.music);
```

直接播放项目 res 文件中的音频，不需要使播放器进入 prepare 状态，因为 create 已经做了这个操作。

2) 添加需要播放的音频文件：

res 文件夹下的 raw 文件：

```
MediaPlayer.create(this, R.raw.test);
```

src/main 下的 assets 文件：

```
AssetFileDescriptor fileDescriptor = getAssets().openFd("music.mp3");
mediaPlayer.setDataSource(fileDescriptor.getFileDescriptor(),fileDescriptor.getStartOffset(),
                          fileDescriptor.getLength());
```

网络上的音频文件：

```
mp.setDataSource("http://www.xxxxxxx/music.mp3");
```

本地 sd 卡下的文件：

```
mp.setDataSource("/sdcard/music.mp3");
```

3) 控制播放器进入 prepare 状态：

prepare() 和 prepareAsync()：提供了同步和异步两种方式设置播放器进入 prepare 状态，需要注意的是，如果 MediaPlayer 实例是由 create 方法创建的，那么第一次启动播放前不需要再调用 prepare（）了，因为 create 方法里已经调用过了。

4) 控制播放器开始和停止播放：

isPlaying()：可以用来检测 MediaPlayer 对象是否处在 Started 状态。

start()：开始播放。

pause() 和 stop()：暂停和停止播放。一旦 MediaPlayer 对象处于 Stoped 状态，不能开始播放，直到调用 prepare() 或者 prepareAsync() 方法来设置 MediaPlayer 对象进入 Prepared 状态以后，才能再次进入 Started 状态。

5) 释放资源：

release()：释放播放器占用的资源，一旦确定不再使用播放器时应当尽早调用它释放资源。

6) 其他的方法：

seekTo()：定位方法，可以让播放器从指定的位置开始播放，需要注意的是该方法是个异步方法，也就是说该方法返回时并不意味着定位完成，尤其是播放的网络文件，真正定位完成时会触发 OnSeekComplete.onSeekComplete()，如果需要是可以调用 setOnSeekCompleteListener(OnSeekCompleteListener)设置监听器来处理的。

reset()：可以使播放器从 Error 状态中恢复过来，重新会到 Idle 状态。

setLooping()：设置循环播放

setOnCompletionListener(MediaPlayer.OnCompletionListener listener)：播放完成监听。

setOnErrorListener(MediaPlayer.OnErrorListener listener)：播放错误监听

97、 视频展示控件 SurfaceView、TextureView、GLSurfaceView、SurfaceTexture

- **SurfaceView**

SurfaceView 继承自 View，并提供了一个独立的绘图层，你可以完全控制这个绘图层，比如说设定它的大小，所以 SurfaceView 可以嵌入到 View 结构树中，需要注意的是，由于 SurfaceView 直接将绘图表层绘制到屏幕上，所以和普通的 View 不同的地方就

在于它不能执行 Transition，Rotation，Scale 等转换，也不能进行 Alpha 透明度运算。

SurfaceView 的 Surface 排在 Window 的 Surface(也就是 View 树所在的绘图层)的下面，SurfaceView 嵌入到 Window 的 View 结构树中就好像在 Window 的 Surface 上强行打了个洞让自己显示到屏幕上，而且 SurfaceView 另起一个线程对自己的 Surface 进行刷新。需要注意的是 SurfaceHolder.Callback 的所有回调方法都是在主线程中回调的。

SurfaceView、SurfaceHolder、Surface 的关系可以概括为以下几点：

1) SurfaceView 是拥有独立绘图层的特殊 View。
2) Surface 就是指 SurfaceView 所拥有的那个绘图层，其实它就是内存中的一段绘图缓冲区。
3) SurfaceView 中具有两个 Surface，也就是我们所说的双缓冲机制
4) SurfaceHolder 顾名思义就是 Surface 的持有者，SurfaceView 就是通过过 SurfaceHolder 来对 Surface 进行管理控制的。并且 SurfaceView.getHolder 方法可以获取 SurfaceView 相应的 SurfaceHolder。
5) Surface 是在 SurfaceView 所在的 Window 可见的时候创建的。我们可以使用 SurfaceHolder.addCallback 方法来监听 Surface 的创建与销毁的事件。

Surface 的渲染可以放到单独线程去做，渲染时可以有自己的 GL context。这对于一些游戏、视频等性能相关的应用非常有益，因为它不会影响主线程对事件的响应。但它也有缺点，因为这个 Surface 不在 View hierachy 中，它的显示也不受 View 的属性控制，所以不能进行平移，缩放等变换，也不能放在其它 ViewGroup 中，一些 View 中的特性也无法使用。

- **SurfaceTexture**

SurfaceTexture 和 SurfaceView 不同的是，它对图像流的处理并不直接显示，而是转为 GL 外部纹理，因此可用于图像流数据的二次处理（如 Camera 滤镜，桌面特效等）。比如 Camera 的预览数据，变成纹理后可以交给 GLSurfaceView 直接显示，也可以通过 SurfaceTexture 交给 TextureView 作为 View heirachy 中的一个硬件加速层来显示。首先，SurfaceTexture 从图像流（来自 Camera 预览，视频解码，GL 绘制场景等）中获得帧数据，当调用 updateTexImage()时，根据内容流中最近的图像更新 SurfaceTexture 对应的 GL 纹理对象，接下来，就可以像操作普通 GL 纹理一样操作它了。

- **TextureView**

TextureView 专门用来渲染像视频或 OpenGL 场景之类的数据的，而且 TextureView **只能用在具有硬件加速的 Window 中**，如果使用的是软件渲染，TextureView 将什么也不显示。也就是说对于没有 GPU 的设备，TextureView 完全不可用。

TextureView 有两个相关类 SurfaceTexture、Surface：

1) Surface 就是 SurfaceView 中使用的 Surface，就是内存中的一段绘图缓冲区。
2) SurfaceTexture 用来捕获视频流中的图像帧的，视频流可以是相机预览或者视频解码数据。SurfaceTexture 可以作为 android.hardware.camera2, MediaCodec, MediaPlayer, 和 Allocation 这些类的目标视频数据输出对象。可以调用 updateTexImage()方法从视频流数据中更新当前帧，这就使得视频流中的某些帧可以跳过。
3) TextureView 可以通过 getSurfaceTexture()方法来获取 TextureView 相应的 SurfaceTexture。但是最好的方式还是使用 TextureView.SurfaceTextureListener 监听器来对 SurfaceTexture 的创建销和毁进行监听，因为 getSurfaceTexture 可能获取的是空对象。

- **GLSurfaceView**

GLSurfaceView 作为 SurfaceView 的补充，可以看作是 SurfaceView 的一种典型使用模式。在 SurfaceView 的基础上，它加入了 EGL 的管理，并自带了渲染线程。另外它定义了用户需要实现的 Render 接口，提供了用 Strategy pattern 更改具体 Render 行为的灵活性。作为 GLSurfaceView 的 Client，只需要将实现了渲染函数的 Renderer 的实现类设置给 GLSurfaceView 即可。

98、 JNI

JNI 全称 Java Native Interface，Java 本地化接口，可以通过 JNI 调用系统提供的 API。Java 和 C/C++不同 ，它不会直接编译成平台机器码，而是编译成虚拟机可以运行的 Java 字节码的.class 文件，通过 JIT 技术即时编译成本地机器码，所以有效率就比不上 C/C++代码。

JNI：JNI 是一套编程接口，用来实现 Java 代码与本地的 C/C++代码进行交互；

NDK： NDK 是 Google 开发的一套开发和编译工具集，可以生成动态链接库，主要用于 Android 的 JNI 开发；

- **JNI 在 Android 中作用：**

JNI 可以调用本地代码库(即 C/C++代码)，并通过 Dalvik 虚拟机与应用层和应用框架层进行交互，Android 中 JNI 代码主要位于应用层和应用框架层；

应用层: 该层是由 JNI 开发，主要使用标准 JNI 编程模型；

应用框架层: 使用的是 Android 中自定义的一套 JNI 编程模型，该自定义的 JNI 编程模

- **Java 语言执行流程：**

编译字节码：Java 编译器编译 .java 源文件，获得.class 字节码文件；

装载类库：使用类装载器装载平台上的 Java 类库，并进行字节码验证；

Java 虚拟机：将字节码加入到 JVM 中，Java 解释器和即时编译器同时处理字节码文件，将处理后的结果放入运行时系统；

调用 JVM 所在平台类库：JVM 处理字节码后，转换成相应平台的操作，调用本平台底层类库进行相关处理；

Java 一次编译到处执行: JVM 在不同的操作系统都有实现，Java 可以一次编译到处运行，字节码文件一旦编译好了，可以放在任何平台的虚拟机上运行。

- **JNI 数据类型映射**

jni.h 头文件就是为了让 C/C++ 类型和 Java 原始类型相匹配的头文件定义。

Java 类型	本地类型	Jni 类型	域描述符	描述
int	long	jint/jsize	I	signed 32 bit
long	_int64	jlong	J	signed 64 bit
byte	signed char	jbyte	B	signed 8 bit
boolean	unsigned char	jboolean	Z	unsigned 8 bit
char	unsigned short	jchar	C	unsigned 16 bit
short	short	jshort	S	signed 16 bit
float	float	jfloat	F	32 bit
double	double	jdouble	D	64 bit
void	void	void	V	N/A
Object		jobject	Ljava/lang/Object;	任何 Java 对象，或者没有对应 java 类型的对象
Class		jclass		Class 对象
String		jstring	Ljava/lang/String;	字符串对象
Object[]		jobjectArray	[Ljava/lang/Object;	任何对象的数组
int[]		jintArray	[I	整型数组
long[]		jlongArray	[J	长整型数组
byte[]		jbyteArray	[B	比特型数组
boolean[]		jbooleanArray	[Z	布尔型数组
char[]		jcharArray	[C	字符型数组
short[]		jshortArray	[S	短整型数组
float[]		jfloatArray	[F	浮点型数组
double[]		jdoubleArray	[D	双浮点型数组

实例：

```
1    // Java 层方法   ——>  JNI 函数签名
2    String getString()  ——>  Ljava/lang/String;
3    int sum(int a, int b)  ——>  (II)I
4    void main(String[] args) ——> ([Ljava/lang/String;)V
```

注：如果使用 javah 工具来生成对应的 native 代码，就不需要手动编写对应的类型转换了。

- **JNIEnv**

JNIEnv 是 jni.h 文件最重要的部分，它的本质是指向函数表指针的指针（JavaVM 也是），函数表里面定义了很多 JNI 函数，同时它也是区分 C 和 C++环境的，在 C 语言环境中，JNIEnv 是 strut JNINativeInterface* 的指针别名。

```
1    struct _JNIEnv;
2    struct _JavaVM;
3    typedef const struct JNINativeInterface* C_JNIEnv;
4    #if defined(__cplusplus)
5    typedef _JNIEnv JNIEnv;   //C++中的 JNIEnv 类型
6    typedef _JavaVM JavaVM;
7    #else
8    typedef const struct JNINativeInterface* JNIEnv;  //C 语言的 JNIEnv 类型
9    typedef const struct JNIInvokeInterface* JavaVM;
10   #endif
```

1) JNIEnv 是一个指针，指向一组 JNI 函数，通过这些函数可以实现 Java 层和 JNI 层的交互，就是说通过 JNIEnv 调用 JNI 函数可以访问 Java 虚拟机，操作 Java 对象；
2) 所有本地函数都会接收 JNIEnv 作为第一个参数；（不过 C++的 JNI 函数已经对 JNIEnv 参数进行了封装，不用写在函数参数上）
3) 用作线程局部存储，不能在线程间共享一个 JNIEnv 变量，也就是说 JNIEnv 只在创建它的线程有效，不能跨线程传递；相同的 Java 线程调用本地方法，所使用的 JNIEnv 是相同的，一个 native 方法不能被不同的 Java 线程调用；

C 语言和 C++的 JNIEnv

C 语言的 JNIEnv 就是 const struct JNINativeInterface*，而 JNIEnv* env 就等价于 JNINativeInterface** env，env 实际是一个二级指针，所以想要得到 JNINativeInterface 结构体中定义的函数指针，就需要先获取 JNINativeInterface 的一级指针对象 *env，然后才能通过一级指针对象调用 JNI 函数，例如：(*env)->NewStringUTF(env, "hello");

```
struct JNINativeInterface {
    void*       reserved0;
    void*       reserved1;
    void*       reserved2;
    void*       reserved3;

    jint        (*GetVersion)(JNIEnv *);
    jclass      (*DefineClass)(JNIEnv*, const char*, jobject, const jbyte*, jsize);
    jclass      (*FindClass)(JNIEnv*, const char*);
    jmethodID   (*FromReflectedMethod)(JNIEnv*, jobject);
    jfieldID    (*FromReflectedField)(JNIEnv*, jobject);
    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedMethod)(JNIEnv*, jclass, jmethodID, jboolean);
    jclass      (*GetSuperclass)(JNIEnv*, jclass);
    jboolean    (*IsAssignableFrom)(JNIEnv*, jclass, jclass);
    /* spec doesn't show jboolean parameter */
    jobject     (*ToReflectedField)(JNIEnv*, jclass, jfieldID, jboolean);
    //……定义了一系列关于 Java 操作的函数
}
```

C++的 JNIEnv 是 _JNIEnv 结构体，而 _JNIEnv 结构体定义了 JNINativeInterface 的结构体指针，内部定义的函数实际上是调用 JNINativeInterface 的函数，所以 C++的 env 是一级指针，调用时不需要加 env 作为函数的参数，例如：env->NewStringUTF("hello");

```
struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
    const struct JNINativeInterface* functions;
#if defined(__cplusplus)
    jint GetVersion()
    { return functions->GetVersion(this); }

    jclass DefineClass(const char *name, jobject loader, const jbyte* buf, jsize bufLen)
    { return functions->DefineClass(this, name, loader, buf, bufLen); }

    jclass FindClass(const char* name)
    { return functions->FindClass(this, name); }

    jmethodID FromReflectedMethod(jobject method)
    { return functions->FromReflectedMethod(this, method); }

    jfieldID FromReflectedField(jobject field)
    { return functions->FromReflectedField(this, field); }

    jobject ToReflectedMethod(jclass cls, jmethodID methodID, jboolean isStatic)
    { return functions->ToReflectedMethod(this, cls, methodID, isStatic); }

    jclass GetSuperclass(jclass clazz)
    { return functions->GetSuperclass(this, clazz); }
    //……
}
```

- **JavaEnv 和 JavaVM 的关系**

1) 每个进程只有一个 JavaVM（理论上一个进程可以拥有多个 JavaVM 对象，但 Android 只允许一个），每个线程都会有一个 JNIEnv，大部分 JNIAPI 通过 JNIEnv 调用；也就是说，JNI 全局只有一个 JavaVM，而可能有多个 JNIEnv；
2) 一个 JNIEnv 内部包含一个 Pointer，Pointer 指向 Dalvik 的 JavaVM 对象的 Function Table，JNIEnv 内部的函数执行环境来源于 Dalvik 虚拟机；
3) Android 中每当一个 Java 线程第一次要调用本地 C/C++ 代码时，Dalvik 虚拟机实例会为该 Java 线程产生一个 JNIEnv 指针；
4) Java 每条线程在和 C/C++ 互相调用时，JNIEnv 是互相独立，互不干扰的，这样就提升了并发执行时的安全性；
5) 当本地的 C/C++ 代码想要获得当前线程所想要使用的 JNIEnv 时，可以使用 Dalvik VM 对象的 JavaVM* jvm->GetEnv() 方法，该方法会返回当前线程所在的 JNIEnv*；
6) Java 的 dex 字节码和 C/C++ 的 .so 同时运行 Dalvik VM 之内，共同使用一个进程空间；

- **JNI 的两种注册方式**

✓ **静态注册**

原理：根据函数名建立 Java 方法和 JNI 函数的一一对应关系。其有两个**宏定义**关键词 JNIEXPORT 和 JNICALL，主要是注明该函数是 JNI 函数，当虚拟机加载 so 库时，如果发现函数含有这两个宏定义时，就会链接到对应的 Java 层的 native 方法。

1) 先编写 Java 的 native 方法；
2) 然后用 javah 工具生成对应的头文件，执行命令 javah packagename.classname 可以生成由（包名+类名）命名的 jni 层头文件，或执行命名 javah -o custom.h packagename.classname，其中 custom.h 为自定义的文件名；
3) 实现 JNI 里面的函数，再在 Java 中通过 System.loadLibrary 加载 so 库即可。

JNI 的调用函数的定义是按照一定规则命名的：

JNIEXPORT 返回值 JNICALL Java_全路径类名_方法名_参数签名(JNIEnv* , jclass, 其它参数);

1) 其中 Java_ 是为了标识该函数来源于 Java。经检验（不一定正确），如果是重载的方法，则有“参数签名”，否则没有；另外如果使用的是 C++，在函数前面加上 extern “C”（表示按照 C 的方式编译），函数命名后面就不需要加上“参数签名”；
2) 包名或类名或方法名中含下划线 _ 要用 _1 连接；
3) 重载的本地方法后面要用双下划线 __ 连接；
4) 参数签名的斜杠 “/” 改为下划线 “_” 连接，分号 “;” 改为 “_2” 连接，左方括号 “[” 改为 “_3” 连接；

优点：

实现比较简单，可以通过 javah 工具将 Java 代码的 native 方法直接转化为对应的 native 层代码的函数。

缺点：

1) javah 生成的 native 层函数名特别长，可读性很差；
2) 后期修改文件名、类名或函数名时，头文件的函数将失效，需要重新生成或手动改，比较麻烦；
3) 程序运行效率低，首次调用 native 函数时，需要根据函数名在 JNI 层搜索对应的本地函数，建立对应关系，有点耗时。

✓ **动态注册**

原理：直接告诉 native 方法其在 JNI 中对应函数的指针。通过使用 JNINativeMethod 结构来保存 Java native 方法和 JNI 函数关联关系。

1) 先编写 Java 的 native 方法；
2) 编写 JNI 函数的实现（函数名可以随便命名）；
3) 利用结构体 JNINativeMethod 保存 Java native 方法和 JNI 函数的对应关系；
4) 利用 registerNatives(JNIEnv* env)注册类的所有本地方法；
5) 在 JNI_OnLoad 方法中调用注册方法；
6) 在 Java 中通过 System.loadLibrary 加载完 JNI 动态库之后，会调用 JNI_OnLoad 函数，完成动态注册。

99、 自定义 ijkPalyer 播放器

- **CustomTextureView**

重写 TextureView，适配视频的宽高和旋转。

```
1        public void adaptVideoSize(int videoWidth, int videoHeight) {
2            // 调整视频的宽高
3            requestLayout();
4        }
```

```
5
6         @Override
7         public void setRotation(float rotation) {
8             // 调整视频的旋转角度
9             requestLayout();
10        }
11
12        @Override
13        protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
14            // 如果显示的 TextureView 和本身的位置是有 90 度的旋转的，就需要交换宽高参数
15            // 根据测量的宽高适配：
16            // 1.当宽高都是具体值时，根据宽高比较小的去适配；
17            // 2.当宽是具体值时，根据宽来计算高，如果高指定为 match_parent 且超过最大值，则根据高计算宽；
18            // 3.当高是具体值时，根据高来计算宽，如果宽指定为 match_parent 且超过最大值，则根据宽计算高；
19            // 4.其他情况根据宽高是否为 match_parent，来进行宽高的计算和适配。
20        }
```

- **AbstractVideoPlayerController**

控制器抽象类，继承于 FrameLayout。

```
1     public abstract class AbstractVideoPlayerController extends FrameLayout implements View.OnTouchListener
2     {
3         // 设置播放器
4         public void setAbstractVideoPlayer(ICustomeVideoPlayer customeVideoPlayer) {}
5
6         // 开启或者关闭更新进度的计时器
7         protected void startUpdateProgressTimer() {}
8         protected void cancelUpdateProgressTimer() {}
9
10        // 设置标题、未播放时显示图片、总时长
11        public abstract void setTitle(String title);
12        public abstract void setImage(@DrawableRes int resId);
13        public abstract void setLenght(long length);
14
15        // 当播放器的播放状态或者模式（全屏和非全屏）发生变化，更新状态 UI 和控制器界面。
16        protected abstract void onPlayStateChanged(int playState);
17        protected abstract void onPlayModeChanged(int playMode);
18
19        // 重置控制器，将控制器恢复到初始状态。
20        protected abstract void reset();
21        // 更新进度，包括进度条进度，展示的当前播放位置时长，总时长等。
22        protected abstract void updateProgress();
23
24        // 根据手势实时更新进度条，亮度，音量
25        protected abstract void showChangePosition(long duration, int newPositionProgress);
26        protected abstract void showChangeBrightness(int newBrightnessProgress);
27        protected abstract void showChangeVolume(int newVolumeProgress);
28        // 根据手势取消更新进度条，亮度，音量
29        protected abstract void hideChangePosition();
30        protected abstract void hideChangeVolume();
31        protected abstract void hideChangeBrightness();
32
33        @Override
34        public boolean onTouch(View v, MotionEvent event) {
35            // 全屏状态下可用，播放、暂停、缓冲状态下可用
36            // 根据手势进行计算和判断，对进度、亮度、音量进行更新
37            // 主要调用以上 show 和 hide 的六个方法进行处理
38        }
39    }
```

- **CustomeVideoPlayerController**

播放器控制器，控制进度、亮度、音量等。

```
public abstract class CustomeVideoPlayerController extends AbstractVideoPlayerController extends
        NiceVideoPlayerController implements View.OnClickListener,SeekBar.OnSeekBarChangeListener {
    // 初始化视频控制图层的布局，包括播放、暂停、进度条、亮度、音量等；
    // 设置播放链接；
    // 实现抽象类的方法，并根据实际情况更新控制界面的 UI；
    // 对电池状态做监听，更新电池状态图标。
}
```

- **CustomeVideoPlayerManager**

视频播放器管理器，设置视频播放器 CustomeVideoPlayer，对 CustomeVideoPlayer 的生命周期进行管理，以及对返回键事件的监听。

- **IVideoPlayer**

视频播放器接口类。

```
public interface IVideoPlayer {
    // 设置视频 Url（可以是本地，也可以是网络视频），以及 headers
    void setUp(String url, Map<String, String> headers);

    // 播放和暂停
    void start();
    void start(long position);
    void pause();

    // 视频跳转、设置音量、播放速度、从上次的节点继续播放
    void seekTo(long pos);
    void setVolume(int volume);
    void setSpeed(float speed);
    void continueFromLastPosition(boolean continueFromLastPosition);

    // 播放器的播放状态
    boolean isIdle();
    boolean isPreparing();
    boolean isPrepared();
    boolean isBufferingPlaying();
    boolean isBufferingPaused();
    boolean isPlaying();
    boolean isPaused();
    boolean isError();
    boolean isCompleted();

    // 播放器的模式
    boolean isFullScreen();
    boolean isNormal();

    int getMaxVolume();
    int getVolume();
    long getDuration();
    long getCurrentPosition();
    int getBufferPercentage(); // 获取视频缓冲百分比
    float getSpeed(float speed);
    long getTcpSpeed(); // 获取网络加载速度
    void enterFullScreen();
    boolean exitFullScreen();
    // 此处只释放播放器（如果要释放播放器并恢复控制器状态需要调用{@link #release()}方法）
    void releasePlayer();
    // 释放 INiceVideoPlayer，释放后，内部的播放器被释放掉，同时如果在全屏模式下会退出
    // 并且控制器的 UI 也应该恢复到最初始的状态.
    void release();
}
```

- **CustomeVideoPlayer**

视频播放器。

```
public class CustomeVideoPlayer extends FrameLayout implements IVideoPlayer,
        TextureView.SurfaceTextureListener {
    // 通过 TextureView.SurfaceTextureListener 来监听 SurfaceTexture 是否已经准备好
    // 调用 IJK 中的 MediaPlayer API 进行视频播放以及相关操作
    mMediaPlayer = new IjkMediaPlayer();

     // 设置监听
    mMediaPlayer.setOnPreparedListener(mOnPreparedListener);
    mMediaPlayer.setOnVideoSizeChangedListener(mOnVideoSizeChangedListener);
    mMediaPlayer.setOnCompletionListener(mOnCompletionListener);
    mMediaPlayer.setOnErrorListener(mOnErrorListener);
    mMediaPlayer.setOnInfoListener(mOnInfoListener);
    mMediaPlayer.setOnBufferingUpdateListener(mOnBufferingUpdateListener);
    // 设置 dataSource
    mMediaPlayer.setDataSource(mContext.getApplicationContext(), Uri.parse(mUrl), mHeaders);
    if (mSurface == null) {
        mSurface = new Surface(mSurfaceTexture);
    }
    mMediaPlayer.setSurface(mSurface);
    mMediaPlayer.prepareAsync();

    ……
    其他的视频播放操作
    最终这些操作大多都交由 IjkMediaPlayer 处理
}
```

- **VideoPlayActivity**

```
public class VideoPlayActivity extends AppCompatActivity {

    private CustomeVideoPlayer mCustomeVideoPlayer;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play);
        init();
    }

    private void init() {
        mCustomeVideoPlayer = (CustomeVideoPlayer) findViewById(R.id.video_player);
        mCustomeVideoPlayer.setPlayerType(CustomeVideoPlayer.TYPE_IJK); // IjkPlayer or MediaPlayer
        String videoUrl = "http://tanzi27niu.cdsb.mobi/2017-05-17_17-33-30.mp4";
//        videoUrl = Environment.getExternalStorageDirectory().getPath().concat("/上班.mp4");
        mCustomeVideoPlayer.setUp(videoUrl, null);
        CustomeVideoPlayerController controller = new CustomeVideoPlayerController(this);
        controller.setTitle("上班");
        controller.setLenght(98000);
        Glide.with(this)
                .load("http://tanzi27niu.cdsb.mobi/2017-05-17_17-30-43.jpg")
                .placeholder(R.drawable.img_default)
                .crossFade()
                .into(controller.imageView());
        mCustomeVideoPlayer.setController(controller);
    }

    @Override
    protected void onStop() {
        super.onStop();
        CustomeVideoPlayerManager.instance().releaseCustomeVideoPlayer();
    }

    @Override
    public void onBackPressed() {
```

```
37            if (CustomeVideoPlayerManager.instance().onBackPressd()) return;
38            super.onBackPressed();
39        }
40    }
```