1.四大组件

Activity

> 一个应用程序的显示组件，绘制用户界面的窗口，在屏幕上提供一个区域，可填满整个屏幕，也可浮动或小窗，允许用户进行交互操作

1. 四种启动模式与任务栈(Task)

   > 在AndroidManifest.xml -> 中设置启动模式 - android:launchMode="singleInstance/singleTop/singleTask"
   >
   > standard、singleTop、singleTask针对的任务栈都是当前对应的app进程的，而singleInstance指向整个系统。

   1. 任务栈Task　用于存放Activity组件，LIFO，一个任务栈包含了Activity的集合，系统可通过Task有序管理Activities，只有栈顶Activity可以跟用户进行交互。
   2. Standard　默认配置的标准启动模式，每次启动一个Activity都会创建一个实例，无论其是否已存在于栈中。
   3. SingleTop　栈顶复用模式，假设ActivityA已位于栈顶，此时调用onNewIntent再创建ActivityA时将复用该ActivityA不再创建新实例

      > 应用场景：当前要跳转的页面已经在栈顶时，如消息通知跳转

   4. SingleTask　栈内复用模式，假设ActivityA已位于栈内，此时再创建ActivityA时将复用该ActivityA且将其上方的Activity全部出栈

      > 应用场景：有一个专用主页面作为基础的app，如Activity+ViewPager+多Fragment的情况。
      >
      > > 如果其他APP进程开启了Activity1，此时会创建新的任务栈
      >
      > > 如果以该模式启动的Activity1已经活动在后台的某任务栈中，启动后后台任务栈会一起切换到前台

   5. SingleInstance　全局唯一模式，采用一个单独的Task栈管理该Activity，具有全局唯一性，主要用于解决多应用程序共享Activity的问题。

      > 应用场景：系统内部应用如电话、短信等，通过Intent传播时会固定的调用这些应用。

2. 创建Activity的方式
   1. 自定义Activity类名，继承AppCompatActivity
   2. 重写onCreate()方法，调用setContentView()设置要显示的视图
   3. 在AndroidManifest.xml中注册并配置该Activity
   4. 启动Activity
      1. startActivity()

```
// 显式启动
Intent intent1 = new Intent(MainActivity.this,
SecondActivity.class);
startActivity(intent1);
// 隐式启动
Intent intent2 = new Intent();
```

```
intent2.setAction("actionStr");
intent2.setCategory("categoryStr");
startActivity(intent2);
<activity android:name=".SecondActivity">   // manifest.xml
    <intent-filter>
        <!-- 只能设置一个action，可设置多个category，只有action&category同时匹配时可响应Intent -->
        <action android:name="actionStr" />
        <category android:name="categoryStr" />
    </intent-filter>
</activity>
```

2. startActivityForResult(Intent intent, int requestCode, Bundle options)

   以指定的请求码requestCode启动Activity，新Activity结束后根据requestCode回调到旧Activity的`onActivityResult()`获取并处理返回结果

3. startActivities(Intent[] intents, Bundle options)

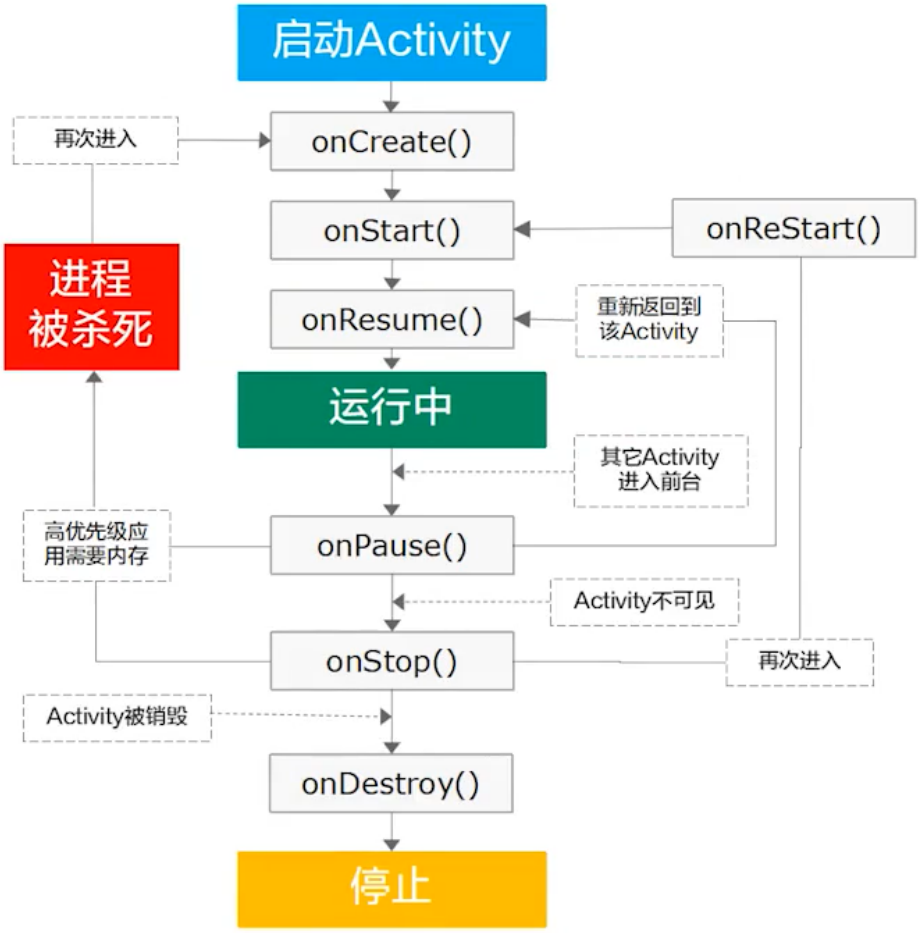
   先启动最后一个Activity，返回时依次创建并启动前一个Activity。如intent对应ABC，则启动顺序为CBA，应用场景：首页广告跳转，消息推送等

5. 关闭Activity，调用finish()或手机返回键、home键

3. 生命周期管理

> 四种状态：
>
> - 运行状态 当前Activity位于前台，用户可见，可获得焦点
> - 暂停状态 其他Activity位于前台，该Activity可见但无法获得焦点
> - 停止状态 Activity不可见，失去焦点
> - 销毁状态 Activity结束，Task栈内清除
>
> 根据回调方法划分为三个生存期：
>
> - **完整生存期** Activity在onCreate()和onDestroy()之间的生存期。
> - **可见生存期** Activity在onStart()和onStop()之间的生存期。
> - **前台生存期** Activity在onResume()和onPause()之间的生存期。

1. onCreate()　首次创建Activity时调用
2. onStart()　在Activity即将对用户可见之前调用
3. onResume()　在Activity即将开始与用户交互之前调用
4. onPause()　当系统即将开始另一个Activity时调用
5. onStop()　当Activity对用户不再可见时调用
6. onDestroy()　在Activity被销毁前调用
7. onRestart()　当Activity已停止并即将再次启动前调用



4. 生命周期方法回调实例
    1. 按下返回键
        - 启动应用 onCreate - onStart - onResume
        - 按下返回键 onPause - onStop - onDestroy
        - 从近期任务再次启动 onCreate - onStart - onResume
    2. 按下Home建
        - 启动应用 同上
        - 按下Home键 onPause - onStop
        - 从近期任务再次启动 onRestart - onStart - onResume
    3. 按下Home键后调整字体大小
        - 启动应用 同上
        - 按下Home键 同上
        - 调整字体大小 无回调
        - 从近期任务再次启动 onDestroy - onCreate - onStart - onResume

> Configuration改变：屏幕旋转、字体大小改变、语言变化、键盘显隐、切换主题/字体 -> 资源配置改变 -> 重启应用onDestroy - onCreate
>
> 应对措施：
>
> - 保存和恢复 onSaveInstanceState()/onRestoreInstanceState()
> - configChanges　AndroidManifest中声明Configuration改变时的处理，再重写onConfigurationChanged()方法自定义处理

5. 如何提升进入页面的速度
    1. 应用启动时间：从点击图标开始，创建一个进程，直到看见界面

2. 应用启动关键流程：创建进程 - 创建、初始化Application类 - 创建Activity类 - **onCreate** - 配置主题信息 - **onStart** - **onResume** - **measure/layout/draw** - 显示
3. 优化措施：
    1. 耗时任务异步处理
    2. 布局优化　减少布局层次、Merge标签、Viewstub、自定义控件
    3. 不可见视图延时加载　资源分开初始化、addView、页面分开加载

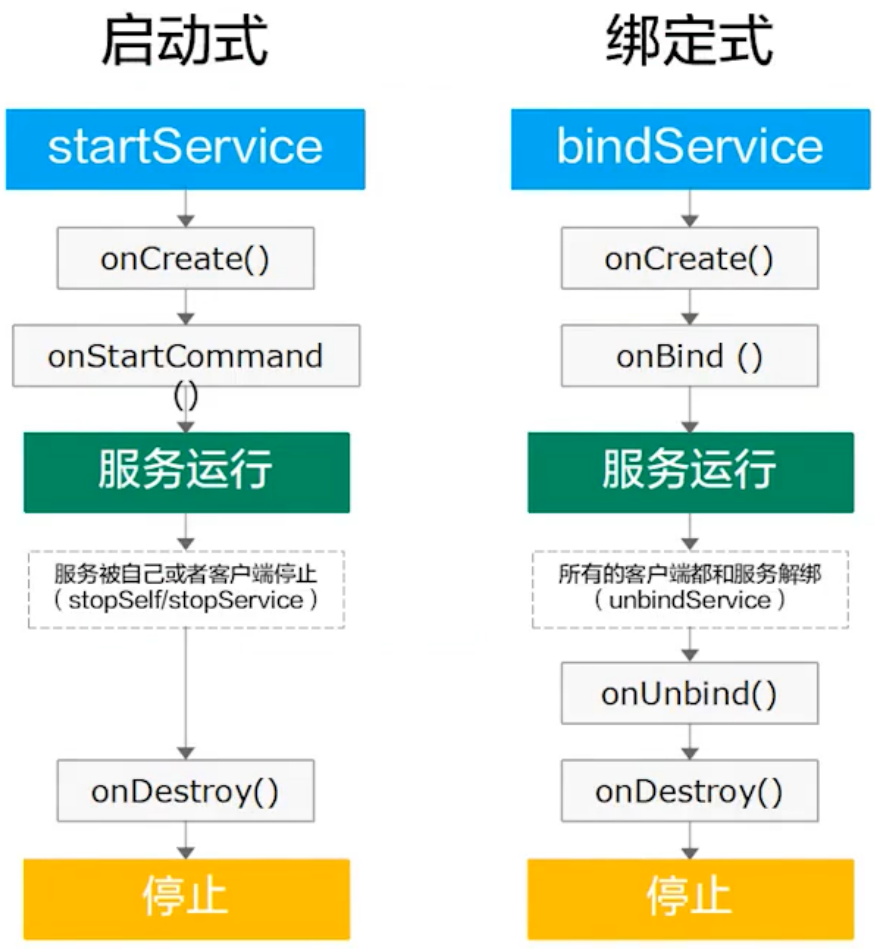
6. 内存泄漏问题

    `static、Handler、ServiceConnection`等在Activity中引发的内存泄漏问题，常基于生命周期解决，即`onDestroy`中进行置空、解绑等。

Service

`Service`实现程序后台运行，适用于执行不需要与用户交互且要求长期运行的任务。其运行不依赖于任何用户界面。服务并非运行在独立进程中，而依赖于创建服务时所在的APP进程。Service运行在主线程，在ANR机制中，Service响应时长不能超过20s，并不能进行耗时操作，可如果使用Thread异步处理，则可进行耗时操作。

1. 生命周期回调
    1. onCreate 创建Service时调用，完成一次性设置
    2. onStartCommand 通过startService启动服务时调用，一旦执行，服务便可后台无限期运行，用于计数，服务被调用的次数
    3. onBind 通过bindService与服务绑定时调用，通过返回IBinder提供一个接口供客户端与服务进行通信
    4. onUnbind 客户端与服务端解绑后调用
    5. onDestroy 在Service被销毁前调用，清理所有资源如注册的监听器、接收器等



2. Service类型
    1. 前台服务 `startForegroundService()`，前台服务可一直保持运行状态，必须使用通知
    2. 后台服务　优先级较低，内存不足易被杀死
        - 不可交互后台服务`startService`　独立于Activity运行，保持一个单例，自行结束或被叫停；onCreate - onStartCommand - onDestroy
        - 可交互后台服务`bindService`　绑定于Activity运行，Activity结束时会被叫停；onCreate - onBind - onUnbind - onDestroy
        - 混合型服务`start|bind同时使用`　一直在后台运行，保持单例，必须显式stop停止，unbind不会停止Service；onCreate - onStartCommand - onBind - onUnbind - onDestroy
    3. AIDL跨进程服务　实现跨进程通信的服务
    4. 调用`bindService`启动Service时应保证在某处会调用`unbindService`解绑；调用`startService`一定要保证`stopService`出现。
3. 绑定Service
    1. 扩展Binder类　使用场景：应用程序私用，并于客户端运行同一进程
        1. 服务端提供客户端可调用的方法
        2. 扩展类返回服务实例
    2. Messenger　使用场景：不同进程间通信；服务每次处理一个请求
        1. 客户端使用IBinder将Message实例化
        2. Message对象向服务发送命令
    3. AIDL文件　使用场景：不同进程间通信；服务每次处理多个请求
        1. android接口定义语言
        2. 将对象解析为系统可识别的形态
4. 使用方法

    Service组件需要在AndroidManifest中注册

```
<service android:name=".LocalService"/> // 1.在AndroidManifest中注册
startService(Intent);   // 2.启动Service
bindService(Intent, ServiceConnection, Int);
unBindService(ServiceConnection);   // 3.解绑，绑定启动时需要成对出现
stopService(Intent);    // 4.暂停，开始启动时需要成对出现
```

5. 与Activity通信

    基于`IBinder`实现，通常使用其实现类`Binder`并通过强制转换完成操作。

```
bindService(Intent, ServiceConnection, Int);

public class LocalService extends Service {
    private final IBinder binder = new ServiceBinder();
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }
    public class ServiceBinder extends Binder {
        LocalService getLocalService() {
            return LocalService.this;
        }
    }
```

```
}

ServiceConnection connection = new ServiceConnection() {    // //
bindService()方法中的参数之一，用于对Service进行操作
    // Activity和Service绑定时调用
    @Override
    public void onServiceConnected(ComponentName name, IBinder binder) {
        // 基于Binder拿到我们要的Service
        service = ((LocalService.ServiceBinder)binder).getLocalService();
        // 干你需要干的事情
    }
    // Activity和Service解绑时调用
    @Override
    public void onServiceDisconnected(ComponentName name) {
        service = null;
    }
};
/* Int: - BIND_AUTO_CREATE     收到绑定需求，若Service尚未创建则立即创建
 - BIND_DEBUG_UNBIND     用于测试使用
 - BIND_NOT_FOREGROUND      不允许此绑定将目标服务进程提升到前台调度优先级
 一般使用BIND_AUTO_CREATE
*/
```

BroadcastReceiver

广播，应用程序间的喇叭，通信的手段，如发生电量低、插入耳机等事件系统发出广播，能响应广播的程序将做相应动作。发送广播使用`Intent`，接收广播使用`BroadcastReceiver`。

> `onReceive`方法中不应添加过多逻辑或长时间耗时操作，耗时操作不允许超过10s，Receiver不允许开启线程，运行较长时APP报错

1. 广播分类
    1. 标准广播

        完全异步执行，广播发出后所有Receiver都在同一时刻接收到该消息，无法被截断。

        ```
        public class MyBroadcastReceiver extends BroadcastReceiver {
            @Override
            public void onReceiver(Context context, Intent intent) {
                ...
            }
        }
        // manifest.xml
        <receiver android:name=".MyBroadcastReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.example.broadcasttest.MY_BROADCAST" />
            </intent-filter>
        </receiver>
        // onCreate
        Intent intent = new Intent("com.example.broadcasttest.MY_BROADCAST");
        sendBroadcast(intent);
        ```

    2. 有序广播

        同步执行，广播发出后按照优先级排序顺序传递广播，可被截断或修改广播信息

```
// manifest.xml
<receiver
    android:name=".MyBroadcastReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter android:priority="100">  // android:priority指定广播优先级
        <action android:name="com.example.broadcasttest.MY_BROADCAST" />
    </intent-filter>
</receiver>
sendOrderedBroadcast(intent)    // 发送有序广播
abortBroadcast()    // 终止广播向下发送，低优先级的Receiver将无法接收到该广播
setResultExtras(Bundle) // 将处理结果存入Broadcast传给下一个接收者
```

2. 广播注册方式

    1. 动态注册　非常驻型，依赖注册组件的生命周期，注册/注销需要成对出现，灵活、自由控制，但必须在APP启动后才能运行

        新建一个类继承自`BroadcastReceiver`，并重写`onReceive`方法；在`onCreate/onDestory`中分别进行`register/unregister`操作

        > 动态广播的生命周期较灵活，资源消耗少，响应速度快；动态广播生命周期与其注册处的Avtivity具体操作有关。

        ```
        inner class TimeChangeReceiver() : BroadcastReceiver(){ // 继承BroadcastReceiver并重写onReceiver()
            override fun onReceive(context: Context?, intent: Intent?){
            }
        }
        lateinit var timeChangeReceiver : TimeChangeReceiver    //Receiver对象
        // onCreate()
        val intentFileter = IntentFilter()
        intentFileter.addAction("android.intent.action.TIME_TICK")
        timeChangeReceiver = TimeChangeReceiver()
        registerReceiver(timeChangeReceiver,intentFileter)  //注册该广播接收器
        // onDestroy()
        unregisterReceiver(timeChangeReceiver)  //取消注册该广播接收器
        ```

    2. 静态注册　在Manifest.xml中注册，常驻型，程序关闭亦可激活广播，存在安全性、流畅性问题，8.0系统后，所有隐式广播都不允许使用静态注册方式接收 [隐式广播：未具体指定发送给哪个应用程序的广播]

        动态注册的优先级始终大于静态注册

        > 静态广播一直存在，消耗资源较大，耗电量大；若使用静态广播进行注册，则每接受一次信息就会被销毁，下一次需要重建

        ```
        <receiver android:name=".MyBroadcastReceiver"
                  android:enabled="true"
                  android:exported="true"
                  <intent-filter>
                    <action android:name="BROADCASTRECEIVER"/>
                  </intent-filter>
        </receiver>
        ```

3. 本地广播

系统全局广播可以被其他APP获取，存在安全性隐患；若广播只在APP内使用，可使用本地广播。实现更高效（无需进程间通信）、无需担心其他APP能接受本APP广播的安全问题，本地广播无法静态注册

本地广播`LocalBroadcast`使用`Handler`的消息传输机制；全局广播`Broadcast`使用`Binder`机制

```
private LocalReceiver localReceiver;
private LocalBroadcastManager localBroadcastManager;    // 本地广播管理器
// onCreate
localBroadcastManager = LocalBroadcastManager.getInstance(this);
Intent intent = new Intent("com.example.broadcasttest.LOCAL_BROADCAST");
localBroadcastManager.sendBroadcast(intent);    // 发送本地广播
intentFilter.addAction("com.example.broadcasttest.LOCAL_BROADCAST");
localBroadcastManager.registerReceiver(localReceiver, intentFilter);    // 注册本地广播接收器
// onDestroy
localBroadcastManager.unregisterReceiver(localReceiver);
```

ContentProvider

`ContentProvider`主要用于在不同应用程序间实现数据共享的功能，是**跨程序共享数据**的标准方式。可以选择性共享数据，保证隐私数据没有泄露风险。

1. 使用现有`ContentProvider`读取和操作响应APP中的数据
    1. ContentResolver的基本用法

        APP想要访问Provider中共享的数据，需要借助`ContentResolver`类，使用`Context.getContentResolver`获取实例，提供CRUD方法对数据进行操作。

    2. 内容URI标准格式写法

        `content://com.example.app.provider/table1`，内容URI = authority + path，authority常使用APP包名，path表示APP中的数据表名，`ContentResolver`接受Uri对象作为参数，可以清楚表达操作某APP内某张表的意图。

        `content://com.example.app.provider/table1/1`：期望访问com.example.app这个APP的table1表中id为1的数据

        - `*`：匹配任意长度字符 `#`：匹配任意长度数字

```
Uri uri = Uri.parse("content://com.example.app.provider/table1");
// query
Cursor cursor = getContentResolver().query(uri, projection, selection, selectionArgs, sortOrder);
// uri:指定某APP下某一张表 projection:指定查询的列名 selection:指定where的约束条件 selectionArgs:为selection提供具体值 sortOrder:指定排序方式
if(cursor != null) {
    while(cursor.moveToNext()) {
        String column1 = cursor.getString(cursor.getColumnIndex("column1"));
        int column2 = cursor.getInt(cursor.getColumnIndex("column2"));
    }
    cursor.close();
}
// insert
ContentValues values = new ContentValues();
values.put("column1", "text");
values.put("column2", 1);
```

```
getContentResolver().insert(uri, values);
// update
ContentValues values = new ContentValues();
values.put("column1", "");
getContentResolver().update(uri, values, "column1 = ? and column2 = ?", new
String[] {"text", "1"});
// delete
getContentResolver().delete(uri, "column2 = ?", new String[] {"1"});
```

2. 自定义`ContentProvider`给APP数据提供外部访问接口

```
public class MyProvider extends ContentProvider {
    public boolean onCreate()   // 初始化ContentProvider时调用，通常完成DB创建、
升级等，返回true表示初始化成功
    public Cursor query()   // 从ContentProvider中查询数据，结果放在Cursor中返回
    public Uri insert() // 向ContentProvider中添加一条数据，返回一个用于表示该新记
录的URI
    public int update() // 更新已有数据，返回本次更新受影响的数据行数
    public int delete() // 删除数据，返回被删除的数据行数
    public String getType() // 根据传入的内容URI返回相应的MIME类型
}
```

3. UriMatches

Intent

在各组件间传递数据的信使，用于启动Activity|Service、发送广播等。代表执行某一动作的意图，需指定目标同时目标也需要设置相关属性保证能够响应，IntentFilter即各组件注册响应能力

传递数据的载体，除了传递基本类型key-value外，通常将数据打包封装为Bundle对象发送，复杂数据类型序列化需要实现Parcelable

1. Intent类型
    1. 显式Intent　知道要启动的组件名，只有一个组件能处理该Intent

    ```
    Intent(Context packageContext, Class<?> cls)
    // param1:启动Activity的上下文 | param2:指定想要启动的Activity
    Context.startActivity(intent: Intent)
    ```

    2. 隐式Intent　声明要执行的常规操作，允许多个应用响应该Intent

    > 并不明确指出想要启动哪一个Activity，而制定一系列更抽象的action、category信息，交由系统分析该Intent，找到合适的Activity启动。
    >
    > 在配置文件中某activity内使用标签添加想要该activity相应的action、category
    >
    > 可用于启动其他程序的Activity，如在本程序打开浏览器等。只需要指定希望响应该Intent的action、category等，在startActivity(intent)后，可以响应该Intent的所有应用程序都会给出答复。

    3. 启动Service必须使用显式Intent，隐式Intent启动Service存在安全隐患（无法确定哪些服务会响应Intent，且用户无法知悉哪些服务已启动），Android5.0开始使用隐式Intent调用bindService()，系统将引发异常、

2. 隐式Intent的匹配规则
    1. Action　Intent指定的Action必须与过滤器中某个Action匹配
    2. Category　Intent的每个Category必须与过滤器中某个Category匹配

3. Data（包括URL和数据类型）URL和数据类型是否指定、是否匹配分为4种匹配类型

3. Intent的信息传递

    1. 基础类型　8种基本数据类型+String+CharSequence
    2. 传递复杂对象
        1. Serializable　序列化，将一个对象转换为可存储或可传输状态
            1. Java序列化接口，使用简单但开销大，序列化正反过程需要很多IO操作
            2. 保存对象到本地文件、数据库、网络流
            3. 数据持久化保存；数据存储在磁盘或网络传输数据
        2. Parcelable　解包打包，将一个完整对象分解，每一部分都是Intent支持的基础类型
            1. Android首选序列化方式，使用麻烦但效率高
            2. 主要用于Android跨进程通信时对内存数据的序列化，跨进程传输数据必须序列化
            3. 不同组件及不同程序间高效传输数据；需要自定义如何打包解包
        3. 优先选择Parcelable接口，有数据存储或网络传输需求再考虑Serializable接口
    3. 传递数据的风险
        1. 传递数据量过大，如list.size过大，可能导致ActivityB无法启动，抛出TransactionToolLargeException异常
            - 限制传递的数据量（多个传输对象共用1M的缓冲区）
            - 调整数据传输方式（static、单例、数据库）
        2. 接受端无法创建对象
            - 将类作为公共类放入系统中
            - 封装为jar包

> Intent中使用putExtra存放的数据最终封装进入`Bundle`，即Activity间数据传递的媒介

```
Intent intent = new Intent(this, SecActivity.class);    // 定义要跳转的目标Activity
intent.putExtra("parameter key", "parameter value");    // 将数据放入Intent
startActivity(intent);  //不带回调方法的启动
startActivityForResult(intent, REQUEST_CODE);   //带回调方法的启动
// 如果要从目标Activity中接收返回的数据，就必须重写onActivityResult方法，根据requestCode判断是否为本Activity发出的启动请求。
@Override
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
    if(requestCode == REQUEST_CODE) {
        // 进行相应的处理
    }
    super.onActivityResult(requestCode, resultCode, data);
}
// 从Intent中拿取参数
getIntent().getStringExtra(key);
getIntent().getBooleanExtra(key);
// 通过setResult进行数据的回调
setResult(resultCode);
setResult(resultCode, intent);
// 目标Activity重写onKeyDown，设置返回跳转Activity的数据
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.ACTION_DOWN && event.getAction() == KeyEvent.ACTION_DOWN) { //按下返回键
        Intent intent = new Intent();
        intent.putExtra("parameter key", "parameter value");
```

```
        // setResult(resultCode);
        setResult(resultCode, intent);  // 携带数据的方式
    }
    return super.onKeyDown(keyCode, event);
}
```

Fragment

> 一种嵌入在Activity中的UI片段，主要用于兼顾平板与手机不同屏幕尺寸的自适应性，也可用于减轻Activity，提高加载效率。

1. 添加Fragment
    1. 静态添加Fragment

       使用标签在布局中静态添加，通过android:name显式声明要添加的全路径Fragment

       ```
       <!-- .xml布局文件中 -->
       <fragment
           ...
           android:name="com.xxx.xxx.XxxFragment"/>
       ```

    2. 动态添加Fragment

       通过调用`SupportFragmentManager`并开启事务的方式，将Fragment替换到R.id.xxx的容器内

       ```
       supportFragmentManager.beginTransaction()
           .replace(R.id.xxx, XxxFragment)
           .commit()
       ```

2. 生命周期

   不管Fragment直接载体是什么，其最终都将依赖于Activity，Fragment的生命周期受依赖Activity生命周期的影响。

   `OnCreateView|onDestroyView`间的方法可能会执行N次，因为FragmentTransaction可以执行N次`remove|replace`操作先后加载不同Fragment。新加载的Fragment会取代之前的Fragment切换到前台，旧Fragment视图就会被销毁；若加载新Fragment的操作有添加到回退栈中，当返回时旧Fragment就重回前台，此时会重走`onCreateView -> onDestroyView`的流程。

   1. onAttach()　与Activity建立关联时调用
   2. onCreateView()　当Fragment创建视图(加载布局)时调用
   3. onActivityCreated()　确保关联Activity创建完毕时调用
   4. onDestroyView()　Fragment视图被移除时调用
   5. onDetach() 与Activity解除关联时调用

```
supportFragmentManager.commit {
    setReorderingAllowed(true)
    addToBackStack(null)     // 本次操作添加到回退栈
    repleace(R.id.xxx, FragmentNew())
}
```

FragmentTransaction 的回退栈中保留的是事务而非具体的 Fragment 实例，能响应返回事件的是我们向其中提交的事务，具体的响应结果就是将该事务撤销，恢复到之前的状态

2.UI组件

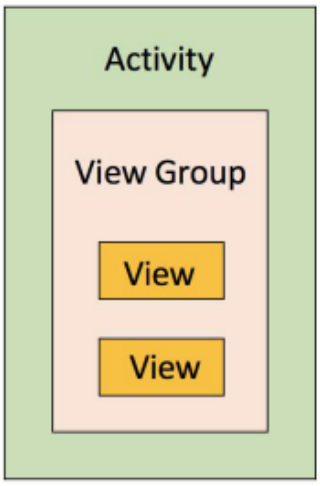
触摸事件分发机制

被分发的对象 -> 用户触摸屏幕产生的点击事件：按下、滑动、抬起、取消，被封装成MotionEvent对象

1. 分发事件、触发场景及单次事件流中触发的次数
    1. MotionEvent.ACTION_DOWN　在屏幕按下时　1次
    2. MotionEvent.ACTION_MOVE　在屏幕上滑动时　0次或多次
    3. MotionEvent.ACTION_UP　在屏幕上抬起时　0次或1次
    4. MotionEvent.ACTION_CANCEL　滑动超出控件边界时　0次或1次

    > 按下、滑动、抬起、取消几种事件组成一个事件流，以按下为开始，中间可能经过若干次滑动，以抬起/取消为结束。
    >
    > 安卓事件分发处理过程中，主要对按下事件作分发，进而找到能够处理按下事件的组件，对于后续事件则直接分发给能处理按下事件的组件。

2. 分发事件的组件，包括Activity、View、View Group。三者的一般结构为：Activity包括了ViewGroup，ViewGroup又可以包含多个View。



    1. Activity　安卓视图类　MainActivty
    2. ViewGroup　View的容器，可以包含若干View　各种布局类
    3. View UI类组件的基类　按钮、文本框
3. 分发的核心方法

    `dispatchTouchEvent()|onTouchEvent()|onInterceptTouchEvent()`，在Activity/View中不存在`onInterceptTouchEvent()`方法。

    1. `dispatchTouchEvent()`

        首先交由子View的`dispatchTouchEvent`或`onTouch`处理，次之则交由自身`onTouchEvent`处理。

    2. `ViewGroup.onInterceptTouchEvent()`

        一般情况下该方法返回False即不拦截，若自定义ViewGroup希望拦截事件，不希望事件继续向子View传播，则重写该方法返回TRUE。

    3. `onTouchEvent`

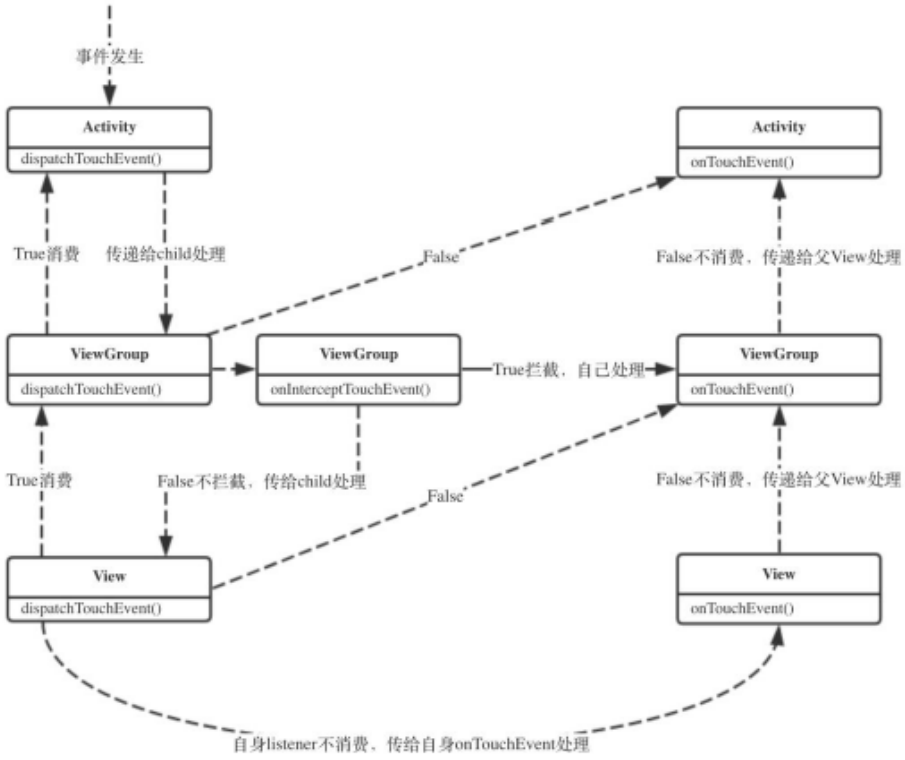
        对事件进行处理，返回TRUE表示已处理，FALSE表示未处理需要继续传递事件。一般默认为FALSE。`View.onTouchEvent`中若设置了点击监听则调用其onClick方法。

4. 事件分发过程

向下传播：Activity包括Layout，事件从Activity向Layout传播；Layout包含若干View，事件从Layout向子View传播。向上传播与之相反。

> 如果某组件的`dispatchTouchEvent()`返回了TRUE，则表示该组件已经对事件进行了处理，不用继续调用其余组件的分发方法，停止分发。如果某组件该方法返回FALSE，则表示该组件无法处理该事件，按照规则继续分发。不重写该方法的情况下，除部分特殊组件，其余组件默认返回False。

```
/* Activity.dispatchTouchEvent()
事件传递给Activity后，先将事件分发给子View处理。若经过子View层层传递或处理后，事件被消费，则Activity.dispatchTouchEvent也返回True，表示事件被消费；如果事件未被消费，则调用自身onTouchEvent处理。如果onTouchEvent消费了事件则返回True，若没有则返回false并作为dispatch的返回值，使调用者明确Activity未消费事件，需要继续处理。
*/
public boolean dispatchTouchEvent(MotionEvent ev) {
    if(child.dispatchTouchEvent(ev)) {
        return true;    // 子View消费了该事件，返回TRUE
    } else {
        return onTouchEvent(ev);    // 子View未消费该事件，调用自身onTouchEvent处理
    }
}
/* ViewGroup.dispatchTouchEvent()
ViewGroup首先尝试onInterceptTouchEvent拦截事件，成功则由ViewGroup自身对象onTouchEvent消费事件，否则交由子View进行分发处理。
> ViewGroup中其实不存在onTouchEvent()，由于ViewGroup继承自View，因此ViewGroup对象调用View.onTouchEvent()。
> 实际onInterceptTouchEvent返回TRUE表示拦截时，调用super.dispatchTouchEvent(View)的方法，进而由该方法调用onTouchEvent
*/
public boolean dispatchTouchEvent(MotionEvent ev) {
    // 核心部分伪代码
    if(!onInterceptTouchEvent(ev)) {
        return child.dispatchTouchEvent;    // 不拦截，传递给子View进行分发处理
    } else {
        return onTouchEvent(ev);    // 拦截事件，交由自身对象onTouchEvent处理
    }
}
/* View.dispatchTouchEvent()
在View.onTouchEvent中，如果设置了onClickListener监听对象，则会调用其onClick方法。同时设置了onTouchListener和onClickListener对象时，OnTouchListener.onTouch优先级高于onClickListener.onClick
*/
public boolean dispatchTouchEvent(MotionEvent ev) {
    if(mOnTouchListener != null && mOnTouchListener.onTouch(this, event)) {
// 如果该对象的监听成员变量不为空，则调用其onTouch方法
        return true;    // 若onTouch返回TRUE，表示事件被消费
    }
    return onTouchEvent(ev);    // 调用自身onTouchEvent处理
}
```



5. 事件分发的主体思路

   由父组件不断向子组件分发，若子组件能处理则立刻返回；若子组件都不处理，则传递到底层的子组件再向上层返回，类似递归过程。事件向下传播，类似于布局由外向内传播。

ViewGroup

1. LinearLayout

   线性布局，控件在线性方向上依次排列，orientation指定线性方向为vertical/horizontal，layout_gravity指定布局中对齐方式，layout_weight使用比例指定控件大小

2. RelativeLayout

   相对布局，通过相对定位方式指定位置。

3. FrameLayout

   帧布局，所有控件都默认摆放在布局的左上角。

4. ConstraintLayout

View

visibility: visible表示可见；invisible表示不可见，但任然占据位置与大小；gone表示不可见且不占据任何空间

1. TextView
    1. gravity指定文字的对齐方式，可以用"|"指定多个值。
    2. textColor/textSize等等...
2. Button
    1. textAllCaps="false" 指定系统保留原始文字，而不是全部转化为大写[Button组件]
    2. 可以使用单抽象方法接口+Lambda表达式注册点击监听事件，也可以通过实现接口方式注册。
3. EditText
    1. maxLines指定EditText的最大行数，保证拉伸长度限制。
4. ImageView

1. 调用setImageResource(Id)动态更改图片。

5. ProgressBar
    1. 用于显示一个进度条，表示正在加载一些数据。
    2. style="?android:attr/progressBarStyleHorizontal"指定为水平进度条，android:max指定最大值
    3. 可根据progressBar.progress属性动态更改进度条进度。
6. AlertDialog
    1. 在当前界面弹出一个对话框，置顶于所有元素之上，用于提示一些重要内容或警告信息。

```
AlertDialog.Builder(this).apply {
    setTitle("This is Dialog")  //标题
    setMessage("Something important")   //信息内容
    setCancelable(false)    //不可取消
    setPositiveButton("OK"){
        dialog, which ->
    }
    setNegativeButton("Cancel"){
        dialog, which ->
    }
    show()
}
```

7. RecyclerView

    RecyclerView相对于ListView的扩展性提升源于对布局排列的管理

> ListView的扩展性较差、运行效率偏低，只能实现纵向滚动效果，无法横向滚动。ListView自身管理布局排列，而RecyclerView令LayoutManager代为管理。LayoutManager提供了一系列扩展接口比如GridLayoutManager网格布局、StaggeredGridLayoutManager瀑布流布局等，只需要实现相应接口就能定制不同布局。

    1. 定制Adapter

```
class FruitAdapter(val fruitList:List<Fruit>) :
RecyclerView.Adapter<FruitAdapter.ViewHolder>() {
    /*
    内部类ViewHolder继承自RecyclerView.ViewHolder，用于缓存子项控件
     */
    inner class ViewHolder(view : View) : RecyclerView.ViewHolder(view){
        val fruitImage : ImageView = view.findViewById(R.id.fruitImage)
        val fruitName : TextView = view.findViewById(R.id.fruitName)
    }
    /*
    用于创建ViewHolder实例，首先渲染相应布局文件，随后将View实例传入ViewHolder
     */
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.fruit_item,parent,f
alse)
        return ViewHolder(view)
    }
    /*
    用于对RecyclerView的子项赋数据值，在每个子项被滚动到屏幕内时执行
     */
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
```

```
            val fruit = fruitList[position]
            holder.fruitName.text = fruit.name
            holder.fruitImage.setImageResource(fruit.imgId)
        }
        override fun getItemCount(): Int {
            return fruitList.size
        }
    }
```

ViewStub|Include|Merge

1. ViewStub

   布局优化的方式之一，适用于延迟加载场景

   相较于`View.GONE`，更加轻量级，本身是一个不可见不占用位置的View，资源消耗比较小

   只有调用了`ViewStub.inflate()`时才加载布局实例化，加载后ViewStub会被替代为实际的布局，因此，ViewStub 一直存在于视图层次结构中直到调用了 `setVisibility(int)` 或 `inflate()`。

   注意事项：

   1. ViewStub只能inflate一次，之后ViewStub对象将被置空，它所占用的空间就会被新的布局替换
   2. ViewStub只能用来inflate一个布局文件，而非某个具体的View
   3. 控制显隐的目标是布局文件，而非某个View
   4. ViewStub的属性会在inflate后传递给其相应的布局

2. Include

   公共布局复用，若一个xml多处被使用，可抽取为单独的xml布局

   复杂布局结构更清晰，若界面复杂可将一个xml布局拆分为多个子xml布局，使用include引用，使结构更清晰

3. merge

   减少布局层次，加快视图的绘制，提高UI性能

   merge标签内含控件设置属性与merge外部ViewGroup控件提供的属性对应

   merge的子元素会直接替换include标签，可减少一层布局

   注意事项：

   1. merge必须在布局文件根节点上
   2. merge不是ViewGroup|View，相当于声明了一些视图等待被添加
   3. 对merge标签设置的所有属性都是无效的

3.数据存储与持久化

文件存储

最基本的数据存储方式，不对存储内容进行任何格式化处理，适合存储**简单的文本数据或二进制数据。**若用此方式存储较复杂结构化数据，则需要自定义一套格式规范。

1. 使用方法：

   1. `Context.openFileOutput(fileName,MODE)`，将数据存储到指定文件中，操作模式有MODE_PRIVATE和MODE_APPEND两种，分别为覆盖和追加方式。

```
String inputText = "xxxxxxxx"
try{
    val output = openFileOutput("data", Context.MODE_PRIVATE)   // 返回FileOutputStream对象
    val writer = BufferedWriter(OutputStreamWriter(output))
    writer.use{ // 扩展函数，调用者作为上下文，可直接使用it，保证执行完毕后自动关闭外层流。
        it.write(inputText)
    }
}catch( e: IOException){
    e.printStackTrace()
}
```

2. `Context.openFileInput(filename)`，从指定文件中读取数据。

```
FileInputStream fis = null;
BufferedReader br = null;
StringBuilder sb = new StringBuilder();
try {
    fis = openFileInput("data");    // 返回FileInputStream对象
    br = new BufferedReader(new InputStreamReader(fis));
    String line = "";
    while((line = br.readLine()) != null) {
        sb.append(line);
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if(br != null) {
        try {
            br.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
return sb.toString();
```

SharedPreference

SharedPreferences使用**键值对方式**存储数据，只支持Java基本数据类型。不支持自定义数据类型，应用内数据共享，使用简单

MODE_PRIVATE　默认操作模式，只有当前APP才可对该SP文件进行读写

1. 三种获取SP对象的方法
    1. Context.getSharedPreferences -> param1：文件名 param2：操作模式
    2. Activity.getSharedPreferences -> 只接收一个操作模式参数，默认以当前Activity类名作为SP文件名
    3. PreferenceManager.getDefaultSharedPreferences -> 只接收一个Context参数，默认以APP包名作为前缀命名SP文件
2. 获取SP对象后存储数据
    1. 调用`SP.edit()`获取一个`SharedPreferences.Editor`对象
    2. 向`Editor`对象中添加数据`putXXX`
    3. 调用`apply()`方法将添加的数据提交，完成数据存储

```
// SharedPreference.edit()每次都返回一个新Editor对象，实现类EditorImpl中有一个缓存Map，apply时先将缓存Map写入内存Map，再将内存Map写入XML文件。
SharedPreferences.Editor editor = getSharedPreferences("data", MODE_PRIVATE).edit();
editor.putString("name", "Tom");
editor.putInt("age", 28);
editor.putBoolean("married", false);
editor.apply();
```

3. 从SP对象中读取数据

   `SP.Editor`对象的`putXXX`方法对应有`getXXX(param1, param2)`方法，param1：Key，param2：若未找到对应值时的返回默认值

```
SharedPreferences sp = getSharedPreferences("data", MODE_PRIVATE);
String name = sp.getString("name", "");
int age = sp.getInt("age", 0);
boolean married = sp.getBoolean("married", false);
```

4.网络请求

Retrofit

1. Retrofit的设计基于以下几点
   1. 同一APP中大多数网络请求指向同一个服务器域名
   2. 服务器接口常根据功能进行归类
   3. 调用者并不需要关心网络具体通信细节
2. 特点
   1. Retrofit只需要配置好一个根路径，在指定服务器接口地址时只需要使用相对路径
   2. 允许对服务器接口归类，同属一类的服务器接口定义到同一个接口文件中
   3. 只需要在接口文件声明方法、返回值，通过注解方式指定对应服务器接口及相应参数，Retrofit将自动发起请求并解析数据为返回值类型，无需了解通信细节
   4. 发起请求时Retrofit会自动开启子线程，数据回调至Callback后会自动切换回主线程
3. 基本使用

```
interface AppService {
    @GET("get_data.json")   // @GET注解表示GET请求，请求的相对地址为get_data.json
    fun getAppData(): Call<List<App>>   // 返回值必须为Call类型，通过泛型指定响应数据的转换类
}
val retrofit = Retrofit.Builder()   // 使用建造者模式构建出retrofit对象
    .baseUrl("")    //指定网络请求根路径
    .addConverterFactory(GsonConverterFactory.create()) // 指定Retrofit解析数据使用的转换库
    .build()
val appService = retrofit.create(AppService::class.java)    // 创建具体Service接口的动态代理对象
appService.getAppData().enqueue(object: Callback<List<App>> {   // 传入Callback匿名对象执行数据回调
    override fun onResponse(call: Call<List<App>>) {
    }
    override fun onFailure(call: Call<List<App>>, t: Throwable) {
    }
})
```

4. 注解类型

    1. @Path()注解声明相对路径中的参数，自动替换到占位符位置 GET http://base_url/page/get_data.json

    ```
    @GET("{page}/get_data.json")
    fun getData(@Path("page") page: Int): Call<Data>
    ```

    2. @Query()注解声明键值对参数 GET http://base_url/get_data.json?u=xxx&t=xxx

    ```
    @GET("get_data.json")
    fun getData(@Query("u") user: String, @Query("t") token: String):
    Call<Data>
    ```

    3. 除@GET外，还提供了@POST|@PUT|@PATCH|@DELETE注解分别对应不同的HTTP请求

    ResponseBody表示能够接受任意类型的响应数据且不会对数据进行解析，常用于不关心响应数据而是操作服务器数据的情况

    ```
    @DELETE("data/{id}")
    fun deleteData(@Path("id") id: String): Call<ResponseBody>
    ```

    4. @Body注解用于POST请求中的body部分，发出请求时自动将对象数据转换为JSON格式文本

    ```
    @POST("data/create")
    fun createData(@Body data: Data): Call<ResponseBody>
    ```

    5. @Headers注解用于指定header参数，但只用于静态声明；动态指定header需要使用@Header参数并作为方法参数

    ```
    @Headers("User-Agent: okhttp", "Cache-Control: max-age=0")
    @GET("get_data.json")
    fun getData(): Call<Data>
    fun getData(@Header("User-Agent") userAgent: String): Call<Data>
    ```

5. 单例Retrofit构建器

```
object ServiceCreator {
    private const val BASE_URL = ""
    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    fun <T> create(serviceClass: Class<T>): T =
retrofit.create(serviceClass)
}
val appService = ServiceCreator.create(AppService::class.java)
```

5.Jetpack组件

Lifecycle

具有生命周期感知能力的组件，用于解决生命周期管理的一致性问题

> 普通组件在使用过程中常需要依赖系统组件的生命周期，如在onCreate()中进行初始化，在onStop()中停止组件，或在onDestroy()中销毁。工作繁琐，存在大量生命周期维护的代码，且不及时释放资源可能导致内存泄漏；无法保证组件会在 Activity/Fragment 停止之前启动组件，从而导致内存泄漏。

lifecycle组件管理不依赖页面的生命周期的回调方法，同时当页面生命周期改变时，能即时收到通知。

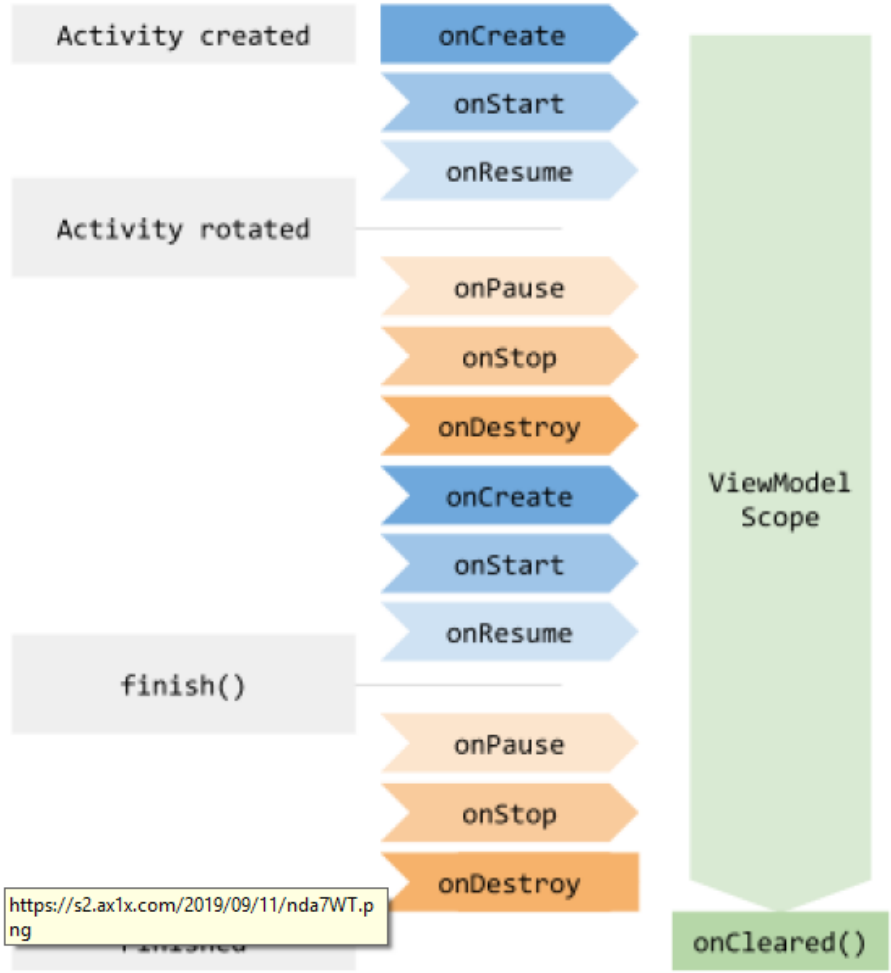
1. Lifecycle有关对象
    1. Lifecycle 持有组件生命周期状态与事件的信息的类
    2. LifecycleOwner Lifecycle的提供者，通过实现LifecycleOwner接口访问Lifecycle生命周期对象
    3. LifecycleObserver Lifecycle观察者，可使用`LifecycleOwner.addObserver()`注册，被注册后的Observer即可观察到Owner的生命周期事件
2. 使用方法

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    void onResume() {
    }
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    void onPause() {
    }
}
// onCreate()
getLifecycle().addObserver(new MyObserver());
// MyObserver通过注解@OnLifecycleEvent可观察Activity的生命周期变化
```

ViewModel

以感知生命周期的形式存储和管理视图相关的数据

1. ViewModel特点
    1. Activity被销毁时，可使用`onSaveInstanceState()`恢复数据，但仅支持少量序列化、反序列化数据，不适用大量数据；ViewModel不仅支持大量数据，还不需要序列化、反序列化操作
    2. Activity/Fragment主要用于显示视图数据，若负责DB或网络加载数据等操作，逻辑过多导致臃肿；ViewModel可以将视图数据相关逻辑和View分离开
    3. ViewModel持有的数据支持在配置改变(如屏幕旋转)中存活
    4. ViewModel在同一个Activity的Fragment间可共享数据
2. ViewModelk的生命周期



3. 创建方法

```
ViewModelProviders.of(this).get(MyViewModel::class.java)    // this指ViewModel绑定的系统组件上下文，如Activity、Fragment
```

LiveData

可观察的数据持有对象，且拥有生命周期感知能力

当数据更新后，LiveData通知其处于Active状态的观察者，若某观察者处于Paused、Destroyed状态则不会收到通知，因此不存在内存泄漏问题，当重新恢复到Resumed状态时会重新收到最新数据

1. 主要方法

```
observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super T> observer);  // 添加观察者对该数据进行观察
setValue(T value);  // 设置数据
getValue(): T   // 获取数据
postValue(T value); // 在主线程更新数据
```

2. 数据变化方法

    1. `Transformations.map()`

        期望在LiveData对象分发给观察者前修改其存储的数据

        ```
        MutableLiveData<String> mutableLiveData = new MutableLiveData<>();
        ```

```
LiveData transformedLiveData = Transformations.map(mutableLiveData, new
Function<String, Object> {
    @Override
    public Object apply(String name) {
        ...
    }
});
transformedLiveData.observe(this, new Observer() {
    @Override
    public void onChanged(@Nullable Object o) {
        ...
    }
});
mutableLiveData.postValue("...");
```

2. `Transformations.switchMap()`

   想手动控制监听其中一个数据变化，且能随时切换监听，必须返回一个LiveData对象

```
mutableLiveData1 = new MutableLiveData<>();
mutableLiveData2 = new MutableLiveData<>();
liveDataSwitch = new MutableLiveData<Boolean>();
LiveData transLiveData = Transformations.switchMap(liveDataSwitch, new
Function<Boolean, LiveData<String>>() {
    @Override
    public LiveData<String> apply(Boolean input) {
        if(input) {
            return mutableLiveData1;
        } else {
            return mutableLiveData2;
        }
    }
})
```

6.项目架构

MVC(Model - View - Controller)

1. 分层职责
   1. 模型层Model：负责数据的加载和存储
   2. 视图层View：负责界面的展示
   3. 控制层Controller：负责业务逻辑的处理
2. 事件流向

   View 产生**事件**，通知到Controller，Controller中进行一系列逻辑处理，之后通知Model去更新数据，Model更新数据后，再将**数据结构**通知给View更新界面。
3. 缺点
   1. 在Android中，因为xml布局能力很弱，View的很多操作是在Activity/Fragment中的，而**业务逻辑同样也是写在Activity/Fragment中。**
   2. Activity/Fragment 责任不明，同时负责View、Controller，导致代码量大，不满足单一职责，逻辑混乱。
   3. Model直接操作View，View的修改导致Controller、Model都需要相应改动

MVP(Model - View - Presenter)

MVP解决了MVC的问题：View责任明确，逻辑不再写在Activity中，而是在Presenter中；Model不再持有View。

Android MVP本质是**面向接口编程，实现了依赖倒置**原则，Model与View无直接引用，一切通过Presenter中转，通常使用Contract接口包含IView、IPresenter子接口的方式进行管理

1. 分层职责
    1. 模型层Model：数据逻辑的处理
    2. 视图层View：处理用户事件、渲染视图
    3. 展示层Presenter：业务逻辑的处理
    4. MVP中使用Bean表示数据模型，Model层用于从指定数据源中获取数据
2. 事件流向

    View产生**事件**，通知给Presenter，Presenter中进行逻辑处理后，通知Model更新数据，Model更新数据后，通知**数据结构**给Presenter，Presenter再通知 View更新界面。
3. 实现思路
    1. UI逻辑抽象成IView接口，由具体的Activity实现类来完成。且调用Presenter进行逻辑操作。
    2. 业务逻辑抽象成IPresenter接口，由具体的Presenter实现类来完成。逻辑操作完成后调用IView接口方法刷新UI。
4. 缺点
    1. 引入大量的IView、IPresenter、IContract接口，增加实现的复杂度，项目文件数量过多。
    2. Presenter除了应用逻辑外，存在大量手动同步逻辑，使类笨重，难以维护。
    3. 解决了View层责任不明的问题，但并没有解决代码耦合的问题，View和Presenter之间相互持有。

MVVM(Model - View - ViewModel)

本质是**数据驱动**，把解耦做的更彻底，ViewModel不持有view 。

1. 分层职责
    1. View层：视图处理、响应事件
    2. Model层：数据模型，用于获取和存储数据，**Repository仓库，包含本地持久数据和服务端数据。**
    3. ViewModel层：业务逻辑的处理，Jetpack ViewModel + Jetpack LivaData
2. 事件流向

    View 产生**事件**，通知ViewModel进行逻辑处理，通知 Model 更新数据，Model 更新数据后，通知**数据结构**给 ViewModel，ViewModel **自动**通知 View 更新界面。将Model与View进行关联，通过ViewModel从Model获取数据，使用DataBinding以观察者模式数据驱动进行View自动刷新。
3. 实现思路
    1. View层包含了我们平时写的Activity/Fragment/布局文件等与界面相关的东西。
    2. ViewModel层用于持有和UI元素相关的数据，以保证这些数据在屏幕旋转时不会丢失，并且还要提供接口给View层调用以及和仓库层进行通信。
    3. 仓库层的主要工作是判断调用方请求的数据应该是从本地数据源中获取还是从网络数据源中获取，并将获取到的数据返回给调用方。本地数据源可以使用数据库、SharedPreferences等持久化技术来实现，而网络数据源则通常使用Retrofit访问服务器提供的Webservice接口来实现。
    4. 向外暴露的获取LiveData的方法应返回LiveData类型，即不可变的，而不是MutableLiveData，好处是避免数据在外部被更改。
    5. ViewModel对象应该比更新的相应View对象存在的时间更长，因此ViewModel实现中不得包含对View对象的直接引用，包括Context。

4. 优点

    1. 低耦合，VIew独立于Model变化和修改
    2. 可重用，一个ViewModel可以绑定到不同的View上

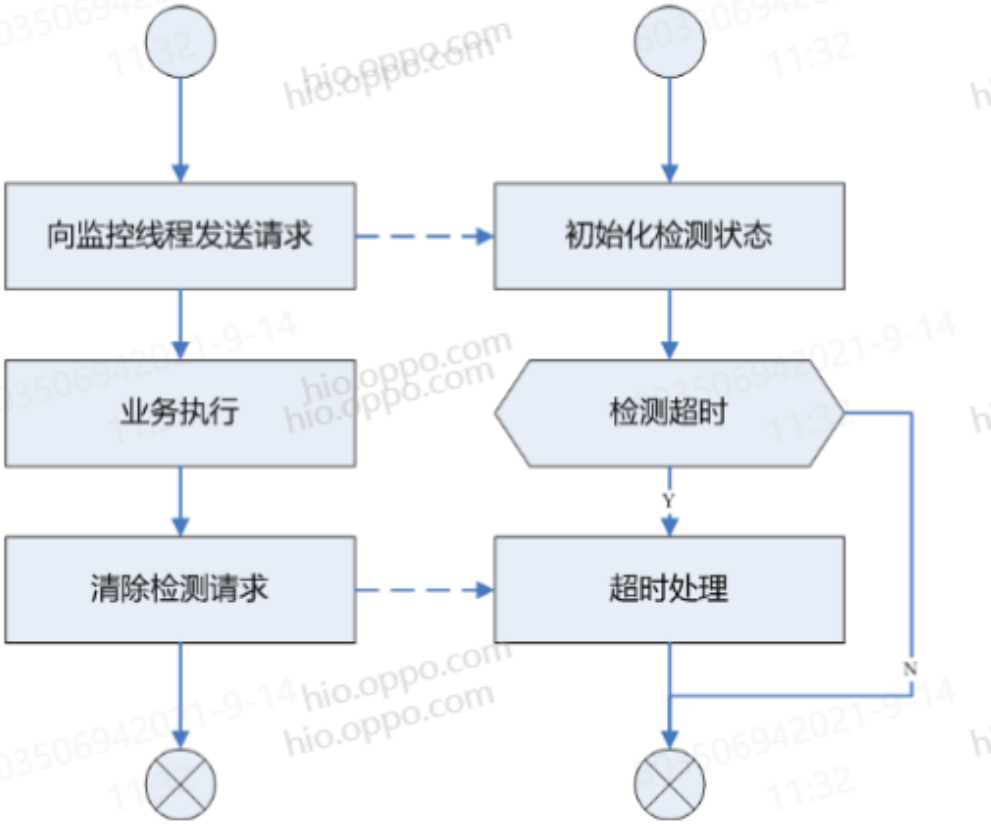
7.其它

Android ANR

ANR(Application Not Responding)，监控应用、系统响应性的机制

1. 监控线程工作原理

    通过监控线程进行ANR工作，主线程进行耗时操作时首先向监控线程发送请求，监控线程检测状态后进入耗时检测



2. ANR分类

    1. 应用ANR 按键响应超时(5S)、广播处理超时(10S)、服务处理超时(20S)
    2. 系统ANR　Watchdog超时(60S)

3. ANR产生原因

    1. 应用本身　耗时操作、死循环、线程阻塞、线程挂起
    2. 其他进程　CPU被抢占

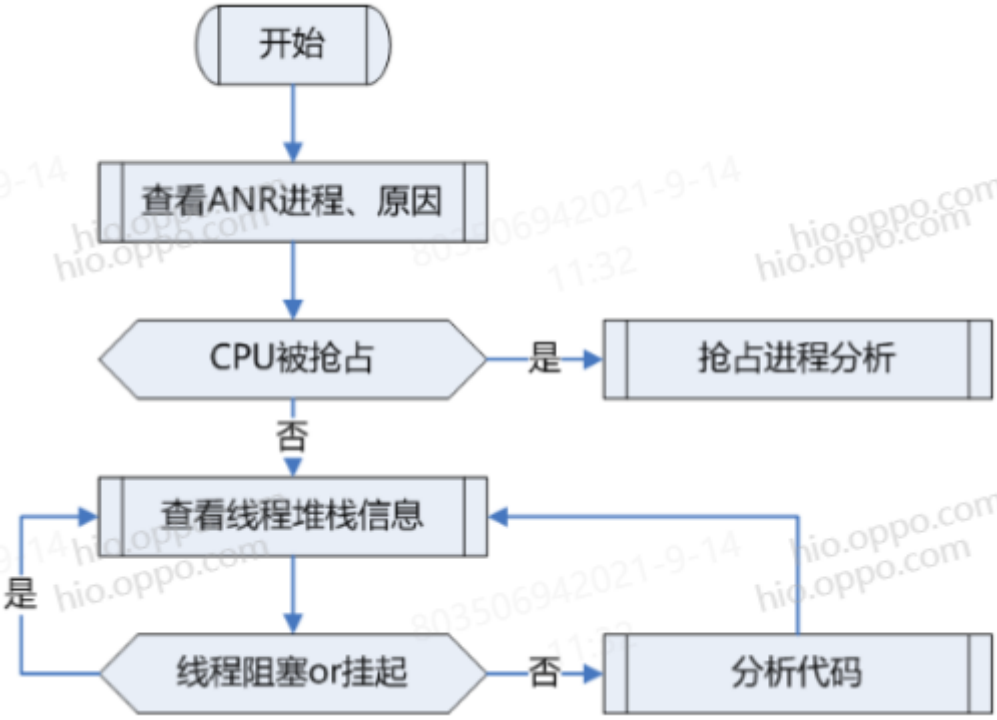
4. ANR现场(日志)

    1. apps/android.txt　出错进程、出错原因、CPU占用率
    2. anr/traces.txt　堆栈信息

5. ANR分析方法

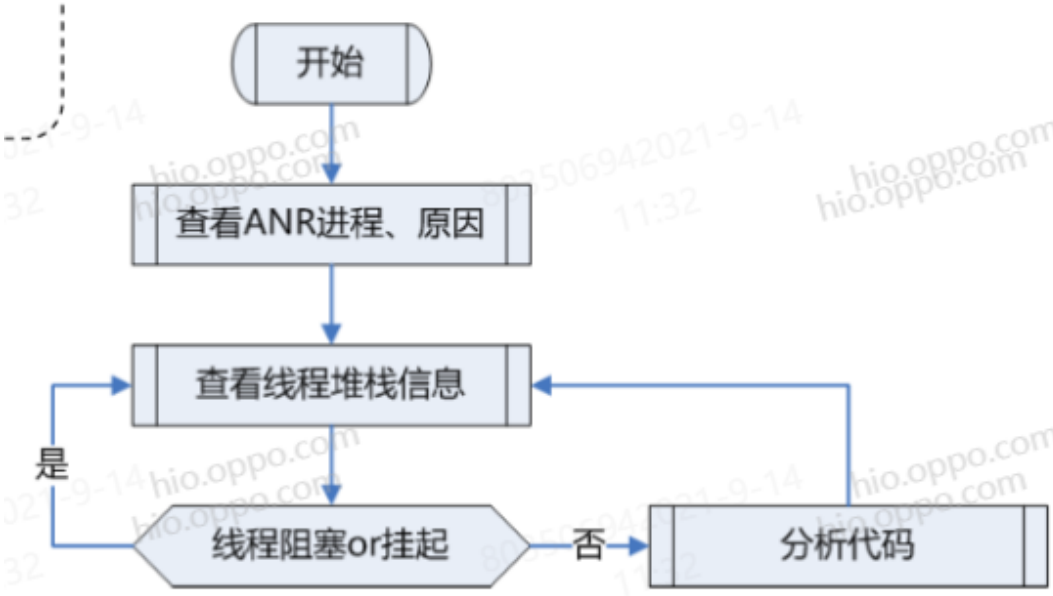
    1. 应用ANR分析方法

        1. 查找ANR进程、原因信息
        2. 检查CPU负载情况
        3. 查看调用堆栈、分析代码



2. 系统ANR分析方法

    1. 查找watchdog信息
    2. 查看调用堆栈、分析代码



6. 广播ANR分析

    1. 应用广播ANR的三种判断状态

        1. 收到通知 IN_ENQUEUEING = 1
        2. 消息入队 IN_HANDLING=2
        3. 消息处理 onReceive()
        4. 处理完成 IN_FINISHING=3

    广播超时ANR表示系统处理问题的状态为0、3

    2. adb命令分析

        1. 查看广播注册信息 adb shell dumpsys activity broadcasts
        2. 分析广播处理流程的log，利用log判断广播分发流程，判断阻塞在系统端或应用端

            1. adb shell dumpsys activity log broadcast 1 // 打开广播log

2. adb shell dumpsys activity log broadcast 0 // 关闭广播log

7. ANR问题优化方向
    1. 避免在主线程执行复杂耗时操作
    2. 广播接收不进行复杂操作
    3. 服务生命周期函数不进行复杂操作
    4. 编码注意边界条件控制，避免死循环
    5. 设计及编码阶段避免出现同步/死锁

ADB

Android Debug Bridge，连接手机与计算机，可在PC端控制手机的命令行工具

1. 安装|卸载应用
    1. `adb install <PC端APK所在路径>` 安装应用
    2. `adb uninstall <应用包名>` 卸载应用，可在手机上直接操作
2. 文件管理
    1. `adb pull <remote> [local]` 手机文件复制到电脑，为设备里文件路径，[local]为计算机目录，可省略并默认复制到当前目录
    2. `adb push [local] <remote>` 将计算机[local]目录文件拷贝到手机文件路径内
    3. `adb shell cd <dir>` 切换路径
    4. `adb shell mkdir [options] <dir_name>` 创建目录
3. 日志打印
    1. `adb shell logcat -b all > [local]` 自动抓取全部log并存放到计算机目录[local]中

8.Notification

每个 app 可以自定义通知的样式和内容等，它会显示在系统的通知栏等区域。用户可以打开抽屉式通知栏查看通知的详细信息。在实际生活中，Android Notification 机制有很广泛的应用，例如 IM app 的新消息通知，资讯 app 的新闻推送等等。

基本使用

1. `NotificationManager`

    对通知进行管理，通过`Context.getSystemService()`获取到`NotificationManger`对象。

    ```
    NotificationManager manager = (NotificationManager)
    getSystemService(Context.NOTIFICATION_SERVICE);
    ```

2. `NotificationCompat.Builder`

    为解决API不稳定性|新老版本兼容的问题，使用`NotificationCompat.Builder`构造器创建`Notification`对象，Android8.0之后还需要传入ChannelId。

    ```
    //创建Notification，传入Context和channelId
    Notification notification = new NotificationCompat.Builder(this,
    "channelId")
        .xxxxx()
        .build();
    ```

3. `Builder.setPriority()`

    通过设置`NotificationConfig`中提供的常量值，给定通知重要程度。

    - PRIORITY_DEFAULT：表示默认重要程度，和不设置效果一样

- PRIORITY_MIN：表示最低的重要程度。系统只会在用户下拉状态栏的时候才会显示
- PRIORITY_LOW：表示较低的重要性，系统会将这类通知缩小，或者改变显示的顺序，将排在更重要的通知之后。
- PRIORITY_HIGH：表示较高的重要程度，系统可能会将这类通知方法，或改变显示顺序，比较靠前
- PRIORITY_MAX：最重要的程度， 会弹出一个单独消息框，让用户做出响应。

4. `NotificationManger.notify(int notifyId, Notification notification)`

   使用该方法显示该通知，NOTIFY_ID需要全局唯一，否则同ID的新通知将覆盖旧通知。Android8.0后需要在此之前先为`NotifactionManager`创建`NotificationChannel`

   ```
   notificationManger.notify(1, notification);
   ```

5. 让通知从状态栏消失

   1. `Builder().setAutoCancel(true)` 点击后自动消失
   2. `notificationManger.cancel(int notifyId)` 在跳转后的目标Activity中动态取消

NotificationChannel

从避免垃圾推送消息的角度出发，Android 8.0系统开始，Google引入了通知渠道这个概念。**每条通知都要属于一个对应的渠道**。每个App都可以自由地创建当前App拥有哪些通知渠道，但是这些**通知渠道的控制权都是掌握在用户手上**的。用户可以自由地选择这些通知渠道的重要程度，是否响铃、是否振动、或者是否要关闭这个渠道的通知。

> 我希望可以即时收到支付宝的收款信息，因为我不想错过任何一笔收益，但是我又不想收到支付宝给我推荐的周围美食，因为我没钱只吃得起公司食堂。这种情况，支付宝就可以创建两种通知渠道，一个收支，一个推荐，而我作为用户对推荐类的通知不感兴趣，那么我就可以直接将推荐通知渠道关闭，这样既不影响我关心的通知，又不会让那些我不关心的通知来打扰我了。

通知渠道一旦创建之后就不能再修改

1. 创建`NotificationChannel`

   ```
   //创建通知渠道ID
   String channelId = "musicNotification";
   //创建通知渠道名称
   String channelName = "音乐播放器通知栏";
   //创建通知渠道重要性
   int importance = NotificationManager.IMPORTANCE_DEFAULT;
   NotificationChannel channel = new NotificationChannel(channelId,
   channelName, importance);
   //为NotificationManager设置通知渠道
   notificationManager.createNotificationChannel(channel);
   ```

2. 重要程度

   数值越高，提示权限就越高，最高的支持发出声音和悬浮通知

```
public class NotificationManager {
    public static final int IMPORTANCE_NONE = 0;
    public static final int IMPORTANCE_MIN = 1;
    public static final int IMPORTANCE_LOW = 2;
    public static final int IMPORTANCE_DEFAULT = 3;
    public static final int IMPORTANCE_HIGH = 4;
    public static final int IMPORTANCE_MAX = 5;
    public static final int IMPORTANCE_UNSPECIFIED = -1000;
}
```

3. 删除`NotificationChannel`

```
notificationManager.deleteNofiticationChannel(int chatChannelId);
```

PendingIntent

使用`PendingIntent`进行通知点击跳转功能。

1. Intent|PendingIntent

   PendingIntent可以看做是对Intent的包装，通过名称可以看出**PendingIntent用于处理即将发生的意图，而Intent用来用来处理马上发生的意图。**可以将PendingIntent看做是延迟执行的Intent。

2. 创建PendingIntent

```
PendingIntent.getBroadcast(context, requestCode, intent, flags)
PendingIntent.getService(context, requestCode, intent, flags)
PendingIntent.getActivity(context, requestCode, intent, flags)
PendingIntent.getActivities(context, requestCode, intent, flags)
 /*
其中flags属性参数用于确定PendingIntent的行为，常传0：
FLAG_ONE_SHOT：  表示返回的PendingIntent仅能执行一次，执行完后自动消失
FLAG_NO_CREATE：  表示如果描述的PendingIntent不存在，并不创建相应的PendingIntent，而是返回NULL
FLAG_CANCEL_CURRENT：  表示相应的PendingIntent已经存在，则取消前者，然后创建新的PendingIntent
FLAG_UPDATE_CURRENT：  表示更新的PendingIntent，如果构建的PendingIntent已经存在，则替换它，常用。
FLAG_IMMUTABLE：  设置Intent在send的时候不能更改
*/
```

   获取到PendingIntent实例后，通过`Builder().setContentIntent(PendingIntent intent)`方法，构建一个PendingIntent。

完整实例

```
Notification notification;
NotificationManager manager;
private final static int NOTIFY_ID = 100;
private void showNotification() {
    manager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    Intent hangIntent = new Intent(this, MainActivity.class);

    PendingIntent hangPendingIntent = PendingIntent.getActivity(this, 1001,
hangIntent, PendingIntent.FLAG_UPDATE_CURRENT);
```

```
    String CHANNEL_ID = "your_custom_id";//应用频道Id唯一值， 长度若太长可能会被截断，
    String CHANNEL_NAME = "your_custom_name";//最长40个字符，太长会被截断
    //Android 8.0 以上需包添加渠道
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationChannel notificationChannel = new
NotificationChannel(CHANNEL_ID,
            CHANNEL_NAME, NotificationManager.IMPORTANCE_LOW);
        manager.createNotificationChannel(notificationChannel);
    }
    notification = new NotificationCompat.Builder(this, CHANNEL_ID)
    .setContentTitle("这是一个猫头")
    .setContentText("点我返回应用")
    .setWhen(System.currentTimeMillis())    //设置通知被创建的时间
    .setSmallIcon(R.mipmap.ic_launcher)
    .setContentIntent(hangPendingIntent)    //点击通知时发送的intent
    .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.mipmap.head))
    .setAutoCancel(true)    //点击通知后自动清除该通知
    .build();
    manager.notify(NOTIFY_ID, notification);
}
```