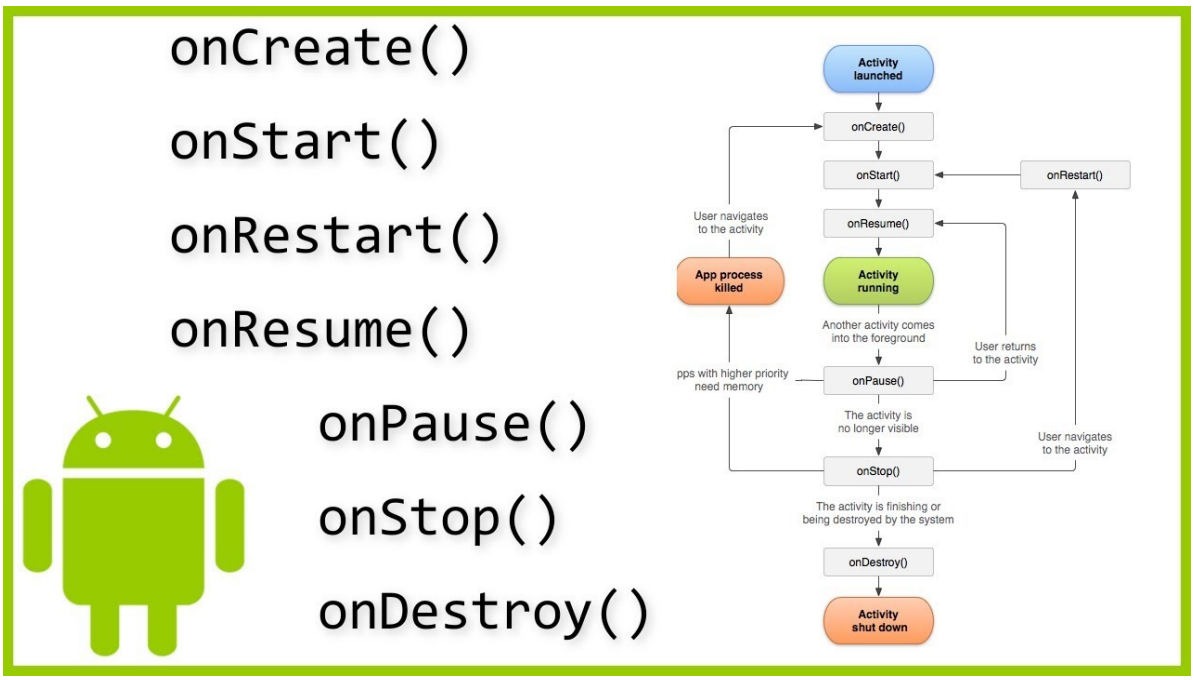
基础

一、 四大组件

1.1 Activity

1.1.1 生命周期:



1.1.2 Intent与序列化:

Intent:

显示:

```
    Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
    startActivity(intent);
```

隐式:

并不明确指出我们想要启动哪一个活动，而是指定了一系列更为抽象的 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活动去启动。

```
打开 AndroidManifest.xml，添加如下代码:
<activity android:name=".SecondActivity" >
     <intent-filter>
         <action android:name="com.example.activitytest.ACTION_START" />
         <category android:name="android.intent.category.DEFAULT" />
     </intent-filter>
</activity>
```

在标签中指明了当前活动可以响应 com.example.activitytest.ACTION_ START 这个 action，而标签则包含了一些**附加信息**，更精确地指明了当前的活动能够响应的 Intent 中还可能带有的 category。只有和中的内容**同时能够匹配**上 Intent 中指定的 action 和 category 时，这个活动才能响应该 Intent。

```
    Intent intent = new Intent("com.example.activitytest.ACTION_START");
    startActivity(intent);
```

android.intent.category.DEFAULT 是一种默认的 category，在调用 startActivity()方法的时候会自动将这个 category 添加到 Intent 中。

每个 Intent 中只能指定一个 action，但却能指定多个 category。这里指定了一个自定义的 category。

```
    Intent intent = new Intent("com.example.activitytest.ACTION_START");
    intent.addCategory("com.example.activitytest.MY_CATEGORY");
    startActivity(intent);
```

```
再添加一个 category 的声明，如下所示：
<activity android:name=".SecondActivity" >
    <intent-filter>
         <action android:name="com.example.activitytest.ACTION_START" />
         <category android:name="android.intent.category.DEFAULT" />
         <category android:name="com.example.activitytest.MY_CATEGORY"/>
     </intent-filter>
</activity>
```

更多隐式：

使用隐式 Intent，我们不仅可以启动自己程序内的活动，还可以启动其他程序的活动，这使得 Android多个应用程序之间的功能共享成为了可能。比如说你的应用程序中需要展示一个网页，这时你没有必要自己去实现一个浏览器，而是只需要调用系统的浏览器来打开这个网页就行了。

```
 Intent intent = new Intent(Intent.ACTION_VIEW);
 intent.setData(Uri.parse("http://www.baidu.com"));
 startActivity(intent);
```

这里我们首先指定了 Intent 的 action 是 Intent.ACTION_VIEW，这是一个 Android 系统内置的动作，其常量值为 android.intent.action.VIEW。然后通过 Uri.parse()方法，将一个网址字符串解析成一个 Uri 对象，再调用 Intent 的 setData()方法将这个 Uri 对象传递进去。

```
<activity android:name=".ThirdActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http" />
    </intent-filter>
</activity>
```

向下一个活动传递数据：

```
//FirstActivity
 String data = "Hello SecondActivity";
 Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
 intent.putExtra("extra_data", data);
 startActivity(intent);
```

```
//SecondActivity
 Intent intent = getIntent();
 String data = intent.getStringExtra("extra_data");
```

返回数据给上一个活动:

startActivityForResult()方法接收两个参数，第一个参数还是 Intent，第二个参数是请求码，用于在之后的回调中判断数据的来源。

```
 //FirstActivity
 Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
 startActivityForResult(intent, 1);

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
         case 1:
             if (resultCode == RESULT_OK) {
                 String returnedData = data.getStringExtra("data_return");
                 Log.d("FirstActivity", returnedData);
             }
         break;
     default:
    }
}
```

onActivityResult()方法带有三个参数，第一个参数 requestCode，即我们在启动活动时传入的请求码。第二个参数 resultCode，即我们在返回数据时传入的处理结果。第三个参数data，即携带着返回数据的 Intent。由于在一个活动中有可能调用 startActivityForResult()方法去启动很多不同的活动，每一个活动返回的数据都会回调到 onActivityResult()这个方法中，因此我们首先要做的就是通过**检查 requestCode 的值来判断数据来源。**确定数据是从SecondActivity 返回的之后，我们再通过**resultCode 的值来判断处理结果是否成功。**最后从 data中取值并打印出来，这样就完成了向上一个活动返回数据的工作。

```
//SecondActivity
 Intent intent = new Intent();
 intent.putExtra("data_return", "Hello FirstActivity");
 setResult(RESULT_OK, intent);
 finish();
```

我们还是构建了一个 Intent，只不过这个 Intent 仅仅是用于传递数据而已，它没有指定任何的“意图”。紧接着把要传递的数据存放在 Intent 中，然后调用了 **setResult()方法。**这个方法非常重要，是专门用于向上一个活动返回数据的。setResult()方法接收两个参数，第一个参数用于向上一个活动返回处理结果，一般只使用 RESULT_OK 或 RESULT_CANCELED 这两个值，第二个参数则把带有数据的 Intent 传递回去，然后调用了 finish()方法来销毁当前活动。

总结：启动活动时使用 startActivityForResult方法，第二个活动使用setResult方法传递带数据的intent，第一个活动使用onActivityResult接收数据。

序列化:

目的:

（1）永久的保存对象数据(将对象数据保存在文件当中,或者是磁盘中

（2）通过序列化操作将对象数据在网络上进行传输(由于网络传输是以字节流的方式对数据进行传输的.因此序列化的目的是将对象数据转换成字节流的形式)

（3）将对象数据在进程之间进行传递(Activity之间传递对象数据时,需要在当前的Activity中对对象数据进行序列化操作.在另一个Activity中需要进行反序列化操作讲数据取出)

两种实现：

1.Implements Serializable 接口 (声明一下即可)

2.Implements Parcelable 接口(不仅仅需要声明,还需要实现内部的相应方法)

Android序列化总结 - 简书 (jianshu.com)

1.1.3 启动模式和FLAG：

启动模式一共有 4 种，分别是 **standard**、**singleTop**、**singleTask** 和 **singleInstance**，可以在 AndroidManifest.xml 中通过给标签指定 android:launchMode属性来选择启动模式。

standard 是活动默认的启动模式，在不进行显式指定的情况下，所有活动都会自动使用这种启动模式。每当启动一个新的活动，它就会在返回栈中入栈，并处于栈顶的位置。对于使用 standard 模式的活动，系统不会在乎这个活动是否已经在返回栈中存在，每次启动都会创建该活动的一个新的实例。

singleTop在启动活动时如果发现返回栈的栈顶已经是该活动，则认为可以直接使用它，不会再创建新的活动实例。

singleTask每次启动该活动时系统首先会在返回栈中检查是否存在该活动的实例，如果发现已经存在则直接使用该实例，并把在这个活动之上的所有活动统统出栈，如果没有发现就会创建一个新的活动实例。

singleInstance 模式的活动会启用一个新的返回栈来管理这个活动。程序中有一个活动是允许其他程序调用的，实现其他程序和我们的程序可以共享这个活动的实例。

Intent的常用Flag参数：

FLAG_ACTIVITY_CLEAR_TOP：例如现在的栈情况为：A B C D 。D此时通过intent跳转到B，如果这个intent添加FLAG_ACTIVITY_CLEAR_TOP 标记，则栈情况变为：A B。如果没有添加这个标记，则栈情况将会变成：A B C D B。也就是说，如果添加了FLAG_ACTIVITY_CLEAR_TOP标记，并且目标Activity在栈中已经存在，则将会把位于该目标activity之上的activity从栈中弹出销毁。这跟上面把B的Launch mode设置成singleTask类似。

FLAG_ACTIVITY_NEW_TASK：例如现在栈1的情况是：A B C。C通过intent跳转到D，并且这个intent添加了FLAG_ACTIVITY_NEW_TASK 标记，如果D这个Activity在Manifest.xml中的声明中添加了Task affinity，并且和栈1的affinity不同，系统首先会查找有没有和D的Task affinity相同的task栈存在，如果有存在，将D压入那个栈，如果不存在则会新建一个D的affinity的栈将其压入。如果D的Task affinity默认没有设置，或者和栈1的affinity相同，则会把其压入栈1，变成：A B C D，这样就和不加FLAG_ACTIVITY_NEW_TASK 标记效果是一样的了。 注意如果试图从非activity的非正常途径启动一个activity，比如从一个service中启动一个activity，则intent比如要添加FLAG_ACTIVITY_NEW_TASK 标记。

FLAG_ACTIVITY_NO_HISTORY：例如现在栈情况为：A B C。C通过intent跳转到D，这个intent添加FLAG_ACTIVITY_NO_HISTORY标志，则此时界面显示D的内容，但是它并不会压入栈中。如果按返回键，返回到C，栈的情况还是：A B C。如果此时D中又跳转到E，栈的情况变为：A B C E，此时按返回键会回到C，因为D根本就没有被压入栈中。

FLAG_ACTIVITY_SINGLE_TOP：和上面Activity的 Launch mode的singleTop类似。如果某个intent添加了这个标志，并且这个intent的目标activity就是栈顶的activity，那么将不会新建一个实例压入栈中。

1.2 Service

服务（Service）是 Android 中实现程序后台运行的解决方案，它非常适合去执行那些不需要和用户交互而且还要求长期运行的任务。

服务的运行不依赖于任何用户界面，即使程序被切换到后台，或者用户打开了另外一个应用程序，服务仍然能够保持正常运行。

服务并不是运行在一个独立的进程当中的，而是依赖于创建服务时所在的应用程序进程。当某个应用程序进程被杀掉时，所有依赖于该进程的服务也会停止运行。

服务并不会自动开启线程，所有的代码都是默认运行在主线程当中的。也就是说，我们需要在服务的内部手动创建子线程，并在这里执行具体的任务，否则就有可能出现主线程被阻塞住的情况。

1.2.1 定义服务

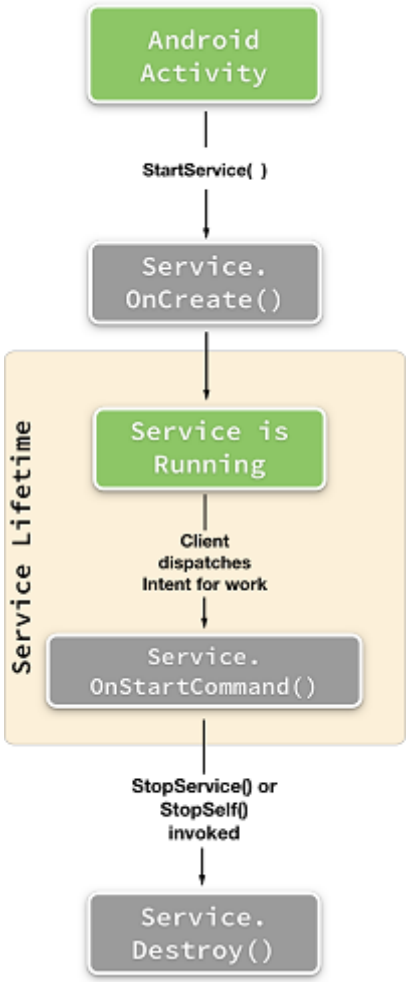
其中 onCreate()方法会在服务创建的时候调用，onStartCommand()方法会在每次服务启动的时候调用，onDestroy()方法会在服务销毁的时候调用。

通常情况下，如果我们希望服务一旦启动就立刻去执行某个动作，就可以将逻辑写在 onStartCommand()方法里。而当服务销毁时，我们又应该在 onDestroy()方法中去回收那些不再使用的资源。

另外需要注意，每一个服务都需要在 AndroidManifest.xml 文件中进行注册才能生效，这是 **Android 四大组件共有的特点**。

```
public class MyService extends Service {
    private Thread mThread;
    public MyService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        mThread = new Thread() {
            @Override
            public void run() {
                try {
                    while (true) {
                        //等待停止线程
                        if (this.isInterrupted()) {
                                throw new InterruptedException();
                            }
                        //耗时操作。
                        System.out.println("执行耗时操作");
                     }
```

```
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
            }
        };
        mThread.start();
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        mThread.interrupt();
    }
}
```



1.2.2 启动/停止服务

```
Intent startIntent = new Intent(this, MyService.class);
startService(startIntent); // 启动服务

Intent stopIntent = new Intent(this, MyService.class);
stopService(stopIntent); // 停止服务
```

1.2.2 活动和服务通信

新建了一个 DownloadBinder 类，并让它继承自 Binder，然后在它的内部提供了开始下载以及查看下载进度的方法。接着，在 MyService 中创建了 DownloadBinder 的实例，然后在 onBind()方法里返回了这个实例.

```
//Service
public class MyService extends Service {
     private DownloadBinder mBinder = new DownloadBinder();
     class DownloadBinder extends Binder {
         public void startDownload() {
         Log.d("MyService", "startDownload executed");
         }
         public int getProgress() {
            Log.d("MyService", "getProgress executed");
            return 0;
        }
 }

 @Override
 public IBinder onBind(Intent intent) {
    return mBinder;
 }
 ...
}
```

首先创建了一个 ServiceConnection 的匿名类，在里面重写了 onServiceConnected()方法和 onServiceDisconnected()方法，这两个方法分别会在活动与服务成功绑定以及活动与服务的连接断开的时候调用。

建出了一个 Intent 对象，然后调用 bindService()方法将 MainActivity 和 MyService 进行绑定。bindService()方法接收 3 个参数，第一个参数就是刚刚构建出的 Intent 对象，第二个参数是前面创建出的 ServiceConnection 的实例，第三个参数则是一个标志位.

```
private MyService.DownloadBinder downloadBinder;
private ServiceConnection connection = new ServiceConnection() {
    @Override
     public void onServiceDisconnected(ComponentName name) {
    }
     @Override
     public void onServiceConnected(ComponentName name, IBinder service) {
         downloadBinder = (MyService.DownloadBinder) service;
         downloadBinder.startDownload();
         downloadBinder.getProgress();
     }
 };
 Button bindService = (Button) findViewById(R.id.bind_service);
 Button unbindService = (Button) findViewById(R.id.unbind_service);
 bindService.setOnClickListener(this);
 unbindService.setOnClickListener(this);

 @Override
 public void onClick(View v) {
    switch (v.getId()) {
     ...
     case R.id.bind_service:
```

```
            Intent bindIntent = new Intent(this, MyService.class);
            bindService(bindIntent, connection, BIND_AUTO_CREATE); // 绑定服务
            break;
        case R.id.unbind_service:
            unbindService(connection); // 解绑服务
            break;
        default:
            break;
        }
    }
```

1.2.3 跨进程，binder/aidl

IPC：Inter-Process Communication，进程间的通信或跨进程通信。简单点理解，一个应用可以存在多个进程，但需要数据交换就必须用IPC；或者是二个应用之间的数据交换。

Binder：Binder是Android的一个类，它实现了IBinder接口。从IPC角度来说，Binder是Android中的一种跨进程通信方式。通过这个Binder对象，客户端就可以获取服务端提供的服务或数据，这里的服务包括普通服务和基于AIDL的服务。

AIDL: Android Interface Definition Language 用于生成可以在Android设备上两个进程之间进行进程间通信（interprocess communication,IPC）的代码。

1.2.4 IntentService

服务中的代码都是默认运行在主线程当中的，如果直接在服务里去处理一些耗时的逻辑，就很容易出现ANR（Application Not Responding）的情况。

IntentService简单地创建一个异步的、会自动停止的服务.

```
public class MyIntentService extends IntentService {
    public MyIntentService() {
    super("MyIntentService"); // 调用父类的有参构造函数
 }
 @Override
 protected void onHandleIntent(Intent intent) {
    // 打印当前线程的 id
    Log.d("MyIntentService", "Thread id is " + Thread.currentThread().
getId());
 }
 @Override
    public void onDestroy() {
    super.onDestroy();
    Log.d("MyIntentService", "onDestroy executed");
 }
}
```

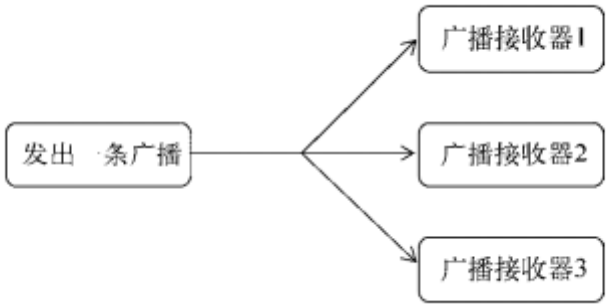
首先要提供一个无参的构造函数，并且必须在其内部调用父类的有参构造函数。然后要在子类中去实现 onHandleIntent()这个抽象方法，在这个方法中可以去处理一些具体的逻辑，而且不用担心 ANR 的问题，因为这个方法已经是在子线程中运行的了。

1.3 BroadcastReceiver

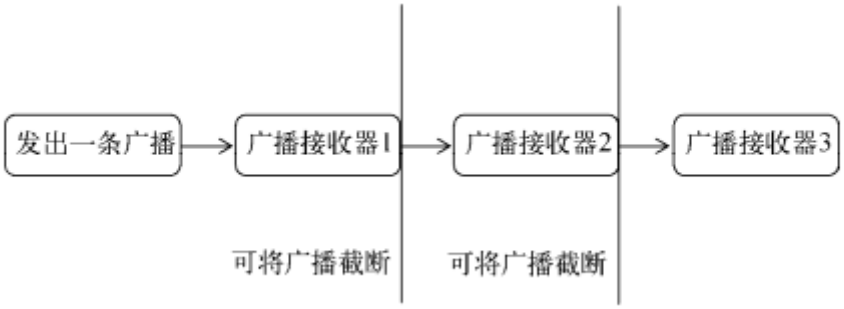
Android 中的每个应用程序都可以对自己感兴趣的广播进行注册，这样该程序就只会接收到自己所关心的广播内容，这些广播可能是来自于系统的，也可能是来自于其他应用程序的。Android 提供了一套完整的 API，允许应用程序自由地发送和接收广播。

1.3.1 标准广播和有序广播

标准广播（Normal broadcasts）是一种完全异步执行的广播，在广播发出之后，所有的广播接收器几乎都会在同一时刻接收到这条广播消息，因此它们之间没有任何先后顺序可言。这种广播的效率会比较高，但同时也意味着它是无法被截断的。



有序广播（Ordered broadcasts）则是一种同步执行的广播，在广播发出之后，同一时刻只会有一个广播接收器能够收到这条广播息，当这个广播接收器中的逻辑执行完毕后，广播才会继续传递。所以此时的广播接收器是有先后顺序的，优先级高的广播接收器就可以先收到广播消息，并且前面的广播接收器还可以截断正在传递的广播，这样后面的广播接收器就无法收到广播消息了。



1.3.2 接收系统广播

Android 内置了很多系统级别的广播，我们可以在应用程序中通过监听这些广播来得到各种系统的状态信息。比如手机开机完成后会发出一条广播，电池的电量发生变化会发出一条广播，时间或时区发生改变也会发出一条广播，等等。

动态注册的广播始终快于静态注册的广播，不管优先级谁高。

1.动态注册

只需要新建一个类，让它继承自 BroadcastReceiver，并重写父类的 onReceive()方法就行了。这样当有广播到来时，onReceive()方法就会得到执行，具体的逻辑就可以在这个方法中处理。

2.静态注册

编写类继承 BroadcastReceiver，重写onReceive()方法，AndroidManifest.xml文件中添加标签。

1.3.3 LocalBroadcastManager

发出的广播只能够在应用程序的内部进行传递，并且广播接收器也只能接收来自本应用程序发出的广播

```
localBroadcastManager = LocalBroadcastManager.getInstance(this); // 获取实例

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");
localReceiver = new LocalReceiver();
localBroadcastManager.registerReceiver(localReceiver, intentFilter); // 注册本地广播监听器
```

基本上就和动态注册广播接收器以及发送广播的代码是一样的。只不过现在首先是通过LocalBroadcastManager的 getInstance()方法得到了它的一个实例，然后在注册广播接收器的时候调用的是 LocalBroadcastManager 的registerReceiver()方法，在发送广播的时候调用的是LocalBroadcastManager的sendBroadcast()方法。

1.4 ContentProvide

内容提供器（Content Provider）主要用于在**不同的应用程序之间实现数据共享的功能**，它提供了一套完整的机制，允许一个程序访问另一个程序中的数据，同时还能保证被访数据的安全性。目前，使用内容提供器是 Android 实现跨程序共享数据的标准方式。

内容提供器的用法一般有两种，一种是使用现有的内容提供器来读取和操作相应程序中的数据，另一种是创建自己的内容提供器给我们程序的数据提供外部访问接口。

1.4.1 增删改查

对于每一个应用程序来说，如果想要访问内容提供器中共享的数据，就一定要借助**ContentResolver** 类，可以通过 Context 中的 getContentResolver()方法获取到该类的实例。ContentResolver 中提供了一系列的方法用于对数据进行 CRUD 操作，其中 insert()方法用于添加数据，update()方法用于更新数据，delete()方法用于删除数据，query()方法用于查询数据。

不同于 SQLiteDatabase，ContentResolver 中的增删改查方法都是不接收表名参数的，而是使用一个 **Uri** 参数代替，这个参数被称为内容 URI。内容 URI 给内容提供器中的数据建立了唯一标识符，它主要由两部分组成：**authority** 和 **path**。**authority 是用于对不同的应用程序做区分的**，一般为了避免冲突，都会采用程序包名的方式来进行命名。比如某个程序的包名是 com.example. app，那么该程序对应的 authority 就可以命名为 com.example.app. provider。**path 则是用于对同一应用程序中不同的表做区分的**，通常都会添加到 authority 的后面。比如某个程序的数据库里存在两张表：table1 和 table2，这时就可以将 path 分别命名为/table1 和/table2，然后把 authority 和 path进行组合，内容 URI 就变成了 com.example.app.provider/table1 和 com.example.app.provider/table2。不过，目前还很难辨认出这两个字符串就是两个内容 URI，我们还需要在字符串的头部加上协议声明。因此，内容 URI 最标准的格式写法如下：

content://com.example.app.provider/table1

content://com.example.app.provider/table2

在得到了内容 URI 字符串之后，我们还需要将它解析成 Uri 对象才可以作为参数传入

```
Uri uri = Uri.parse("content://com.example.app.provider/table1")
```

只需要调用 Uri.parse()方法，就可以将内容 URI 字符串解析成 Uri 对象了。

现在我们就可以使用这个 Uri 对象来查询 table1 表中的数据，代码如下所示：

```
Cursor cursor = getContentResolver().query( uri, projection, selection,
selectionArgs, sortOrder);
```

查询完成后返回的仍然是一个 Cursor 对象，这时我们就可以将数据从 Cursor 对象中逐个读取出来了。读取的思路仍然是通过移动游标的位置来遍历 Cursor 的所有行，然后再取出每一行中相应列的数据

```
if (cursor != null) {
     while (cursor.moveToNext()) {
         String column1 = cursor.getString(cursor.getColumnIndex("column1"));
         int column2 = cursor.getInt(cursor.getColumnIndex("column2"));
     }
     cursor.close();
}
```

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

1.4.2 ContenetObserver

ContentObserver——内容观察者，目的是观察(捕捉)特定Uri引起的数据库的变化，继而做一些相应的处理，它类似于数据库技术中的触发器(Trigger)，当ContentObserver所观察的Uri发生变化时，便会触发它。触发器分为表触发器、行触发器，相应地ContentObserver也分为“表ContentObserver、“行ContentObserver，当然这是与它所监听的Uri MIME Type有关的。

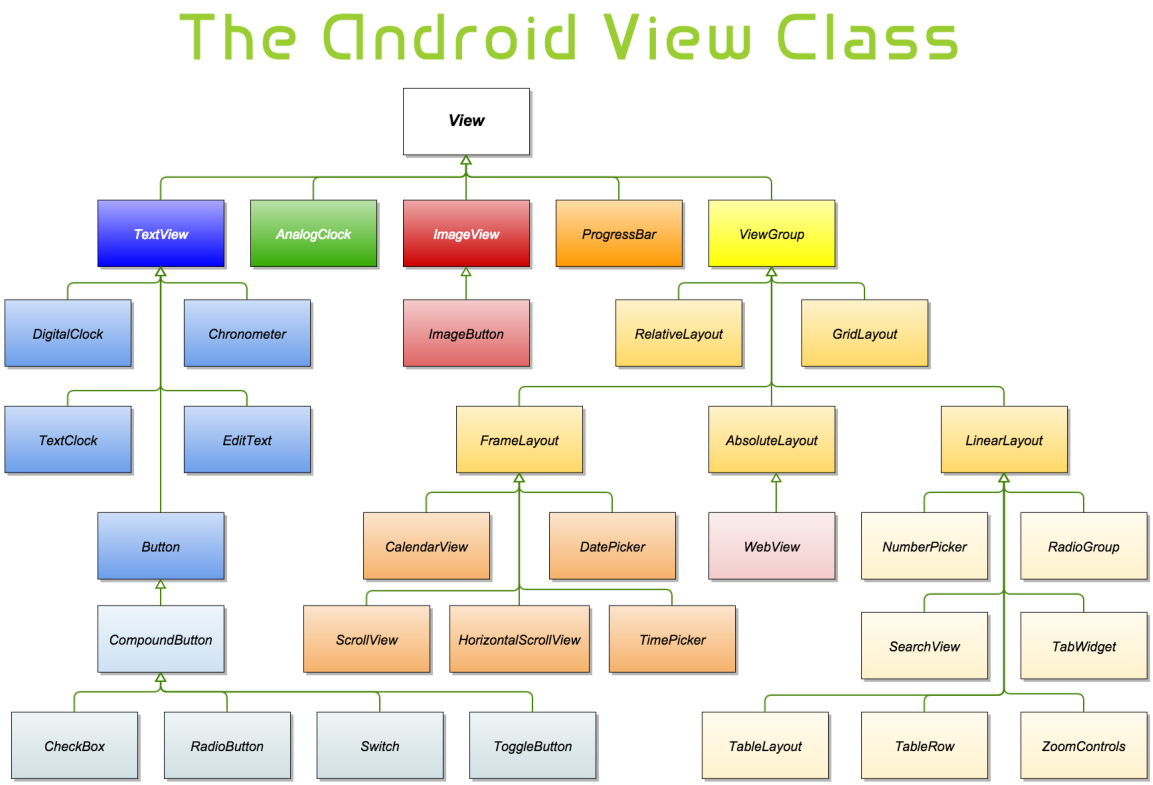
1.4.3 自定义ContentProvider

新建一个类去继承 ContentProvider 的方式来创建一个自己的内容提供器。ContentProvider 类中有 6 个抽象方法，我们在使用子类继承它的时候，需要将这 6 个方法全部重写。

```
public class MyProvider extends ContentProvider {
     @Override
     public boolean onCreate() {
         return false;
     }
     @Override
     public Cursor query(Uri uri, String[] projection, String selection,
String[]
         selectionArgs, String sortOrder) {
         return null;
     }
     @Override
     public Uri insert(Uri uri, ContentValues values) {
        return null;
     }
     @Override
     public int update(Uri uri, ContentValues values, String selection, String[]
     selectionArgs) {
        return 0;
     }
```

```
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
    @Override
    public String getType(Uri uri) {
        return null;
    }
}
```

二、UI



2.1 ViewGoup 五大布局

2.1.1 LinearLayout

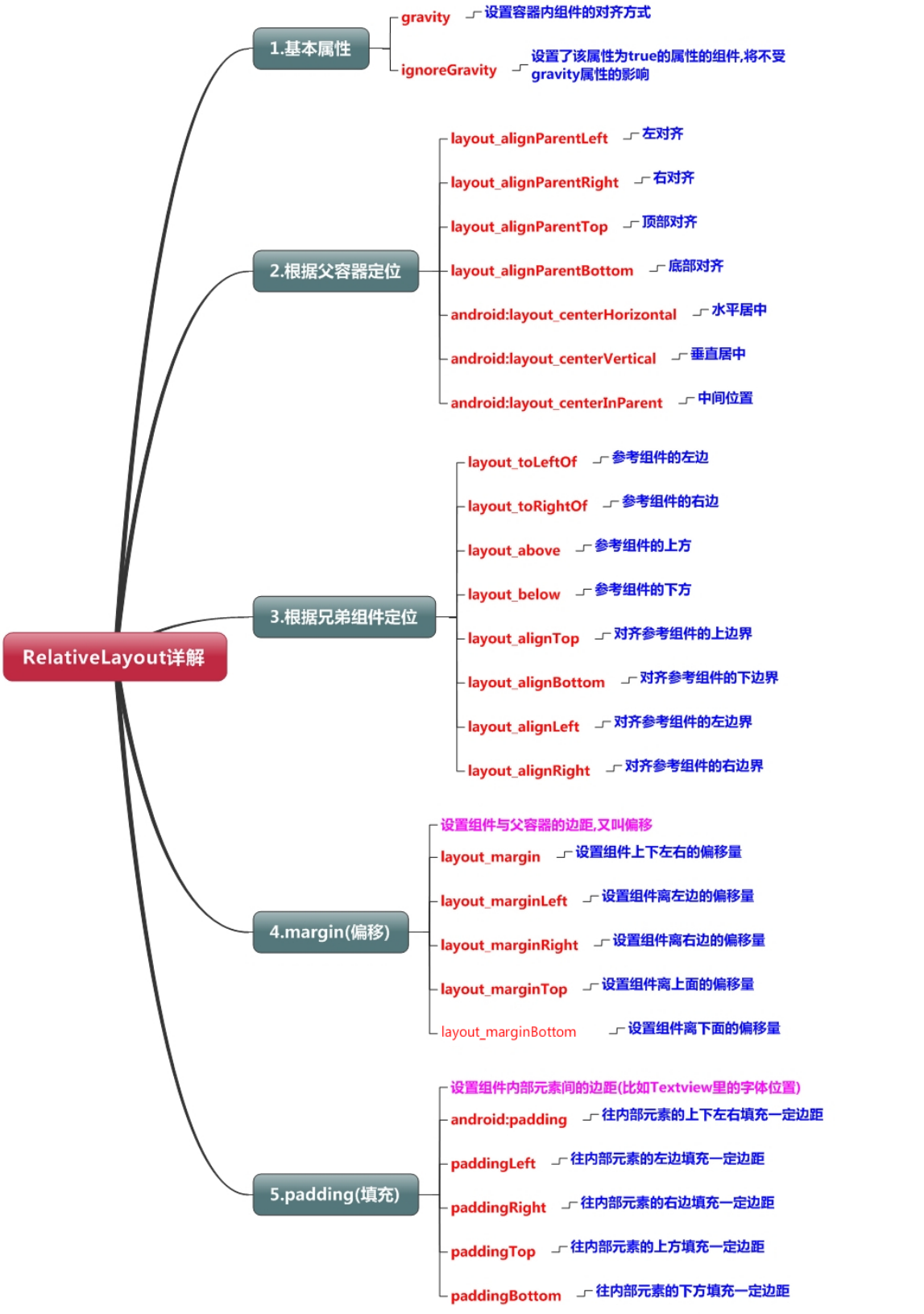
android:orientation 属性指定排列方向

android:layout_weight 属性指定权重

android:gravity用于指定**文字**在控件中的对齐方式，而 android:layout_gravity 用于指定**控件在布局中**的对齐方式。

android:layout_gravity 的可选值和 android:gravity 差不多，但是需要注意，当LinearLayout 的排列方向是 horizontal 时，只有垂直方向上的对齐方式才会生效，因为此时水平方向上的长度是不固定的，每添加一个控件，水平方向上的长度都会改变，因而无法指定该方向上的对齐方式。

2.1.2 RelativeLayout



2.1.3 FrameLayout

设计FrameLayout是为了显示单一项widget。默认把他们放到这块区域的左上角,而这种布局方式却没有任何的定位方式,通常，不建议使用FrameLayout显示多项内容；因为它们的布局很难调节。不用layout_gravity属性的话，多项内容会重叠；使用layout_gravity的话，能设置不同的位置。
layout_gravity可以使用如下取值：

top ： 将对象放在其容器的顶部，不改变其大小.

bottom：将对象放在其容器的底部，不改变其大小.

left：将对象放在其容器的左侧，不改变其大小.

right：将对象放在其容器的右侧，不改变其大小.

center_vertical：将对象纵向居中，不改变其大小.

2.1.4 TableLayout

TableLayout继承了 LinearLayout，因此它的本质依然是线性布局管理器。每次向TableLayout中添加一个TableRow，该TableRow就是一个表格行，TableRow也是容器，因此它也可以不断地添加其他组件，每添加一个子组件该表格就增加一列。如果直接TableLayout中添加组件，那么这个组件将直接占用一行。

表格三种属性

- Shrinkable：如果某个列被设为Shrinkable，那么该列的所有单元格的宽度可以被收缩，以保证该表格能适应父容器的宽度。
- Stretchable：如果某个列被设为Stretchable，那么该列的所有单元格的宽度可以被拉伸，以保证组件能完全填满表格空余空间。
- Collapsed：如果某个列被设为Collapsed，那么该列的所有单元格会被隐藏。

2.1.5 AbsoluteLayout

AbsoluteLayout(绝对布局)：

AbsoluteLayout(绝对布局)之所以把这个放到最后,是因为绝对布局,我们基本上都是不会使用的,我们开发的应用需要在很多的机型上面进行一个适配,如果你使用了这个绝对布局的话,可能你在4寸的手机上是显示正常的,而换成5寸的手机,就可能出现偏移和变形,所以的话,这个还是不建议使用了

AbsoluteLayout(绝对布局)常用属性：

控制大小:

android:layout_width:组件宽度 android:layout_height:组件高度

控制位置:
android:layout_x:设置组件的X坐标 android:layout_y:设置组件的Y坐标

2.2 基础控件

2.2.1 TextView

android:text 指定 TextView 中显示的文本内容

android:gravity 来指定文字的对齐方式

android:textSize 属性可以指定文字的大小 sp

通过 android:textColor 属性可以指定文字的颜色

在 Android 中字体大小使用 sp 作为单位。

2.2.2 Button

系统会对 Button 中的所有英文字母自动进行大写转换

禁用：android:textAllCaps="false"

2.2.3ImageView

图片通常都是放在以“drawable”开头的目录下

android:src="@drawable/img_1 " 属性指定要显示图片

代码中也可以更改图片：imageView.setImageResource(R.drawable.img_2);

2.2.4EditText

android:hint="Type something here" 提示文字

android:maxLines 指定了 EditText 的最大行数为两行，这样当输入的内容超过两行时，文本就会向上滚动

android:inputType="password" 指定输入框类型（可改变键盘默认页面）

2.2.5 ProgressBar

ProgressBar 用于在界面上显示一个进度条，表示我们的程序正在加载一些数据

```
<ProgressBar
 android:id="@+id/progress_bar"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 />
```

会看到屏幕中有一个圆形进度条正在旋转。

可以通过 android:visibility进行指定进度条可见性，可选值有 3 种：visible、invisible 和 gone。

visible 表示控件是可见的，这个值是默认值，不指定 android:visibility 时，控件都是可见的。

invisible 表示控件不可见，但是它仍然占据着原来的位置和大小，可以理解成控件变成透明状态了。

gone 则表示控件不仅不可见，而且不再占用任何屏幕空间。

2.2.6 AlertDialog

```
AlertDialog.Builder dialog = new AlertDialog.Builder (MainActivity.this);
     dialog.setTitle("This is Dialog");
     dialog.setMessage("Something important.");
     dialog.setCancelable(false);
     dialog.setPositiveButton("OK", new DialogInterface.OnClickListener() {
         @Override
         public void onClick(DialogInterface dialog, int which) {
         }
     });
     dialog.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
         @Override
         public void onClick(DialogInterface dialog, int which) {
         }
     });
     dialog.show();
```

2.2.7 ProgressDialog

ProgressDialog 和 AlertDialog 有点类似，都可以在界面上弹出一个对话框，都能够屏蔽掉其他控件的交互能力。

```
ProgressDialog progressDialog = new ProgressDialog(MainActivity.this);
     progressDialog.setTitle("This is ProgressDialog");
     progressDialog.setMessage("Loading...");
     progressDialog.setCancelable(true);
     progressDialog.show();
```

2.3 高级控件

2.3.1 RecyclerView

先定义了一个内部类 ViewHolder，ViewHolder 要继承自 RecyclerView.ViewHolder。然后 ViewHolder 的构造函数中要传入一个 View 参数，这个参数通常就是 RecyclerView 子项的最外层布局，那么我们就可以通过 findViewById()方法来获取到布局中的实例了。

Adapter 中也有一个构造函数，这个方法用于把要展示的数据源传进来，并赋值给一个全局变量，我们后续的操作都将在这个数据源的基础上进行。

Adapter 是继承自 RecyclerView.Adapter 的，那么就必须重写onCreateViewHolder()、onBindViewHolder()和 getItemCount()这 3 个方法。

onCreateViewHolder()方法是用于创建 ViewHolder 实例的，我们在这个方法中将 item 布局加载进来，然后创建一个 ViewHolder 实例，并把加载出来的布局传入到构造函数当中，最后将ViewHolder 的实例返回。

onBindViewHolder()方法是用于对 RecyclerView 子项的数据进行赋值的，会在每个子项被滚动到屏幕内的时候执行，这里我们通过 position 参数得到当前项的 Item实例，然后再将数据设置到 ViewHolder 的 TextView 当中即可。

getItemCount()方法就非常简单了，它用于告诉 RecyclerView 一共有多少子项，直接返回数据源的长度就可以了。

```
implementation "androidx.recyclerview:recyclerview:1.1.0"
```

```
public class ItemAdapter extends RecyclerView.Adapter<ItemAdapter.ViewHolder> {
     private List<Item> mItemList;
     Context mContext;
     static class ViewHolder extends RecyclerView.ViewHolder {
         TextView itemName;
         public ViewHolder(View view) {
            super(view);
            itemName = (TextView) view.findViewById(R.id.item_name);
         }
     }

 public ItemAdapter(List<Item> itemList,Context context) { //构造函数绑定数据源
    mItemList = itemList;
    mContext = context;
 }

 @Override
 public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
```

```
    View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_item, parent,
false);
    ViewHolder holder = new ViewHolder(view);
    return holder;
 }
 @Override
 public void onBindViewHolder(ViewHolder holder, int position) {
    Item item = mItemList.get(position);
    holder.mName.setText(item.getName());
 }
 @Override
 public int getItemCount() {
   return mItemList.size();
 }
}
```

```
 RecyclerView recyclerView = (RecyclerView) findViewById(R.id.recycler_view);
 LinearLayoutManager layoutManager = new LinearLayoutManager(this);
 recyclerView.setLayoutManager(layoutManager);
 ItemAdapter adapter = new ItemAdapter(itemList);
 recyclerView.setAdapter(adapter);
```

为每一个条目注册点击事件

```
public class ItemAdapter extends RecyclerView.Adapter<FruitAdapter.ViewHolder> {
    private List<Item> mItemList;
    static class ViewHolder extends RecyclerView.ViewHolder {
        TextView itemName;
        View itemView;              //*
        public ViewHolder(View view) {
           super(view);
           itemName = (TextView) view.findViewById(R.id.item_name);
        }
    }

 public ItemAdapter(List<Item> itemList) {
   mitemList = itemList;
 }

 @Override
 public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_item, parent,
false);
    final ViewHolder holder = new ViewHolder(view);

    return holder;
 }
 @Override
 public void onBindViewHolder(ViewHolder holder, int position) {
    Item item = mItemList.get(position);
    holder.mName.setText(item.getName());
    holder.ItemView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
```

```
                int position = holder.getAdapterPosition();
                Item item = mItemList.get(position);
                ...         //点击后逻辑
            }
            });
    }
    @Override
    public int getItemCount() {
       return mItemList.size();
    }
}
```

在holder中存储条目view，onBindViewHolder时直接为这个view设置点击事件。

更新数据源

```
adapter.notifyDataSetChanged();
```

2.3.2 Toolbar

顶部导航栏

2.3.3 ViewPager

ViewPager 是android support V4 包中类，这个类可以让用户左右切换当前的View，轮播图？APP欢迎界面？

1. ViewPager直接继承自ViewGroup类，它是一个容器类，可以在其中添加其他的View
2. ViewPager类需要一个PagerAdapter适配器，来为他提供数据。
3. ViewPager经常和Fragment一起使用，并且提供了专门的FragmentPagerAdapter和FragmentStatePagerAdapter类

2.3.4 WebView

混合开发，直接将网页放入WebView容器显示

2.4 自定义View

测量——onMeasure()：决定View的大小
布局——onLayout()：决定View在ViewGroup中的位置
绘制——onDraw()：如何绘制这个View。

2.5 Fragment

碎片（Fragment）是一种可以嵌入在活动当中的 UI 片段，它能让程序更加合理和充分地利用大屏幕的空间，因而在平板上应用得非常广泛

2.5.1 生命周期

`onAttach()`：Fragment和Activity相关联时调用。可以通过该方法获取Activity引用，还可以通过getArguments()获取参数。

`onCreate()`：Fragment被创建时调用

`onActivityCreated()`：当Activity完成onCreate()时调用

`onStart()`：当Fragment可见时调用。

`onResume()`：当Fragment可见且可交互时调用

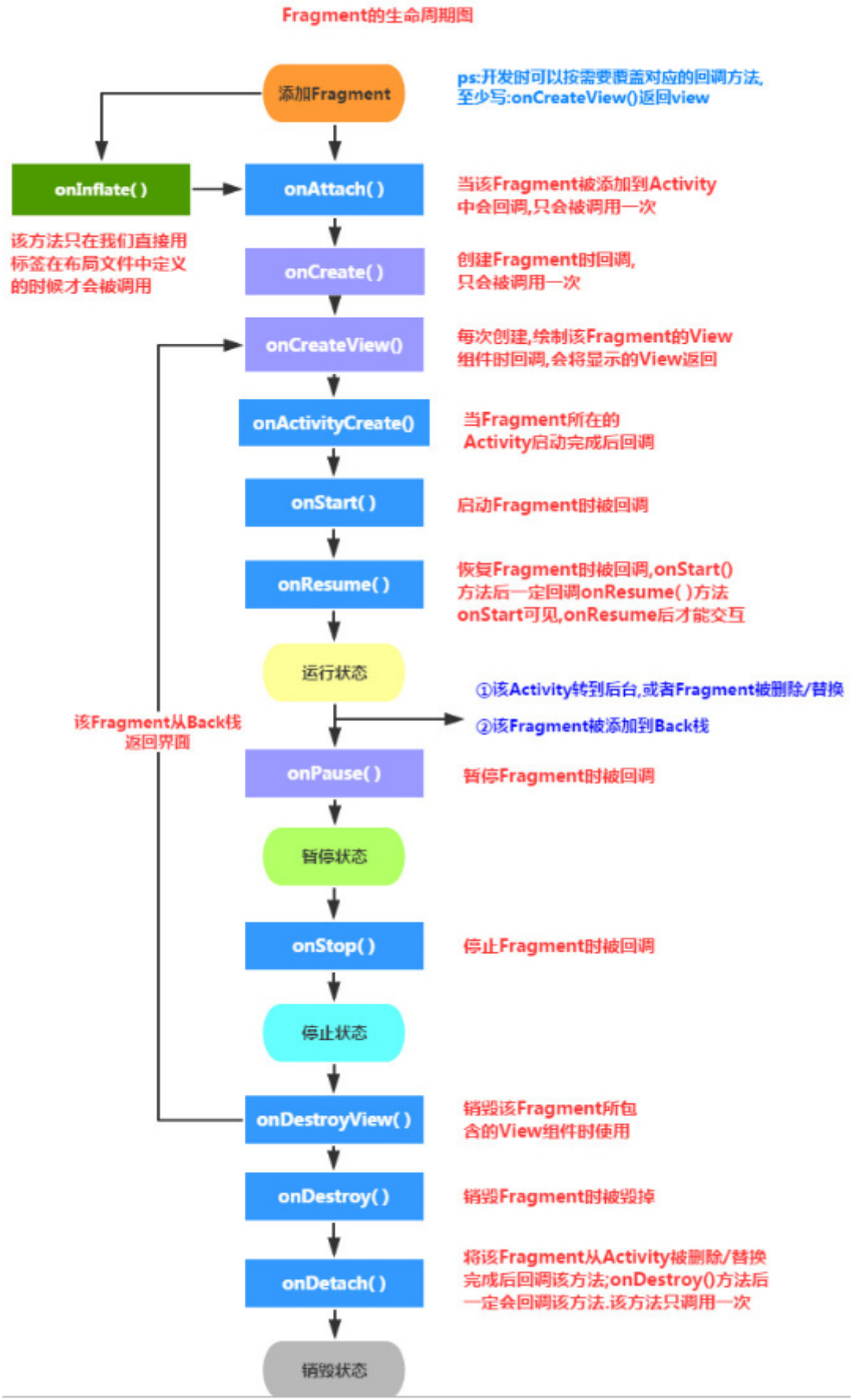
`onPause()`：当Fragment不可交互但可见时调用。

`onStop()`：当Fragment不可见时调用。

`onDestroyView()`：当Fragment的UI从视图结构中移除时调用。

`onDestroy()`：销毁Fragment时调用。

`onDetach()`：当Fragment和Activity解除关联时调用。



2.5.2 基础用法

静态加载

- 定义Fragment的xml布局文件
- 自定义Fragment类，继承Fragment类或其子类，同时实现onCreate()方法，在方法中，通过inflater.inflate加载布局文件，接着返回其View

- 在需要加载Fragment的Activity对应布局文件中`<fragment>`的name属性设为全限定类名，即包名.fragment
- 最后在Activity调用setContentView()加载布局文件即可

静态加载一旦添加就不能在运行时删除

动态加载

- 获得FragmentManager对象，通过`getSupportFragmentManager()`
- 获得FragmentTransaction对象，通过`fm.beginTransaction()`
- 调用`add()`方法或者`repalce()`方法加载Fragment；
- 最后调用`commit()`方法提交事务

2.5.3 Fragment与Activity通信

如果**Activity中包含自己管理的Fragment的引用**，可以通过引用直接访问所有的Fragment的public方法

如果**Activity中未保存任何Fragment的引用**，那么没关系，每个Fragment都有一个唯一的TAG或者ID，可以通过`getFragmentManager.findFragmentByTag()`或者`findFragmentById()`获得任何Fragment实例，然后进行操作

在Fragment中可以通过`getActivity`得到当前绑定的Activity的实例，然后进行操作。

三、数据存储

3.1 SharePreference

不同于文件的存储方式，SharedPreferences 是使用键值对的方式来存储数据的。也就是说，当保存一条数据的时候，需要给这条数据提供一个对应的键，这样在读取数据的时候就可以通过这个键把相应的值取出来。

首先需要获取到 SharedPreferences 对象

1.Context 类中的 getSharedPreferences()方法

2.Activity 类中的 getPreferences()方法

3.PreferenceManager 类中的 getDefaultSharedPreferences()方法

3.2 文件存储

3.3 Sqlite

room mvc

3.4 ContentProvider

只提供外部操作本应用数据的接口，并不实际存储，实际存储的是数据库。

四、线程/进程

4.1 进程

4.1.1 进程优先级

前台进程（Foreground process）。它表明用户正在与该进程进行交互操作，android系统依据下面的条件来将一个进程标记为前台进程：

- 该进程持有一个用户正在与其交互的Activity（也就是这个activity的生命周期方法走到了onResume()方法）。
- 该进程持有一个Service，并且这个Service与一个用户正在交互中的Activity进行绑定。
- 该进程持有一个前台运行模式的Service（也就是这个Service调用了startForegroud()方法）。
- 该进程持有一个正在执行生命周期方法（onCreate()、onStart()、onDestroy()等）的Service。
- 该进程持有一个正在执行onReceive()方法的BroadcastReceiver。

  一般情况下，不会有太多的前台进程。杀死前台进程是操作系统最后无可奈何的做法。当内存严重不足的时候，前台进程一样会被杀死。

可见进程（Visible process）。它表明虽然该进程没有持有任何前台组件，但是它还是能够影响到用户看得到的界面。android系统依据下面的条件将一个进程标记为可见进程：

- 该进程持有一个非前台Activity，但这个Activity依然能被用户看到（也就是这个Activity调用了onPause()方法）。例如，当一个activity启动了一个对话框，这个activity就被对话框挡在后面。
- 该进程持有一个与可见（或者前台）Activity绑定的Service。

服务进程（Service process）。除了符合前台进程和可见进程条件的Service，其它的Service都会被归类为服务进程。

后台进程（Background process）。持有不可见Activity（调用了onStop()方法）的进程即为后台进程。通常情况下都会有很多后台进程，当内存不足的时候，在所有的后台进程里面，会按照LRU（最近使用）规则，优先回收最长时间没有使用过的进程。

空进程（Empty process）。不持有任何活动组件的进程。保持这种进程只有一个目的，就是为了缓存，以便下一次启动该进程中的组件时能够更快响应。当资源紧张的时候，系统会平衡进程缓存和底层的内核缓存情况进行回收。

4.1.2 LowMemoryKiller

OOM全称Out Of Memory，是Linux当中，内存保护机制的一种。该机制会监控那些占用内存过大，尤其是瞬间很快消耗大量内存的进程，为了防止内存耗尽而内核将该进程杀掉。

当Kernel遇到OOM的时候，可以有2种选择：

1） 产生kernelpanic（死机）

2） 启动OOM killer，选择一个或多个“合适”的进程，干掉那些选择中的进程，从而释放内存。

在Android中，及时用户退出当前应用程序后，应用程序还是会存在于系统当中，这是为了方便程序的再次启动。但是这样的话，随着打开的程序的数量的增加，系统的内存就会不足，从而需要杀掉一些进程来释放内存空间。至于是否需要杀进程以及杀什么进程，这个就是由Android的内部机制LowMemoryKiller机制来进行的。Andorid的Low Memory Killer是在标准的linux lernel的OOM基础上修改而来的一种内存管理机制。当系统内存不足时，杀死不必要的进程释放其内存。

名称	触发条件	机制原理
OOM	分配内存时发生页面错误，没有剩余内存阀值可控制（就是机器中的 RAM 已经几乎没有了）	基于多个标准给每个进程算分，分数最高的进程将被杀死
LowMemoryKiller	低于某个阀值之后，就正式启动 LowMemoryKiller（6.0 中，剩余内存低于 315M，LowMemoryKiller 就开始杀进程）	以系统定义的阀值为标准，有多层控制，当处于低于某个阀值值，会自动运行处理。

4.2 线程

4.2.1 线程和线程池使用

在Java中，已经提供了2种实现线程的方式。

继承Thread类

```
class MyThread extends Thread{
    @Override
    public void run() {
        //耗时操作
    }
}
new MyThread().start();
```

实现Runnable接口

```
new Thread(new Runnable() {
    @Override
    public void run() {
        //耗时操作
    }
}).start();
```

两种方式虽然都可以实现，但是两者还是有一定的区别，因为Java的单继承性质，继承了Thread类则不能在继承其他的类，而实现Runnable接口则没有这种限制。另外在多线程去操作同一个资源的时候，也应该选择使用实现Runnable接口的方法来实现。

线程池

针对多线程并发的问题，引入线程池的好处就比较明显了：

- 可以控制同一时间内线程的最大并发数，合理利用系统资源，提高应用性能。
- 当任务完成后，可以重用已经创建的的线程，避免频繁创建新的线程而导致的GC(垃圾回收机制)。
- 通过Executors工具类，可以更方便的使用线程池，控制线程的最大并发数、线程的定时任务、单线程的顺序执行等。

4.2.2 Handler / Looper /Message

1.MessageQueue

MessageQueue顾名思义，指的就是消息队列，说这个之前我们首先需要知道什么是Message，比如说我们在UI界面时点击一个按钮，或者是接收到了一条广播，其实都算是一条Message，这些事件都被封装成一条Message被添加到了MessageQueue队列当中，因为我们知道一个线程在一段时间只能对一种操作进行相关的处理，因此这些消息的处理就要有先后顺序，因此采用MessageQueue来管理，也就是消息队列。消息队列其实就是一堆需要处理的Message而形成的传送带。一旦有消息发送进来,那么直接执行enqueueMessage()方法。也就是将消息压入到队列当中，一旦线程空闲下来，那么直接从MessageQueue中取出消息，使得消息出队。

2.Looper

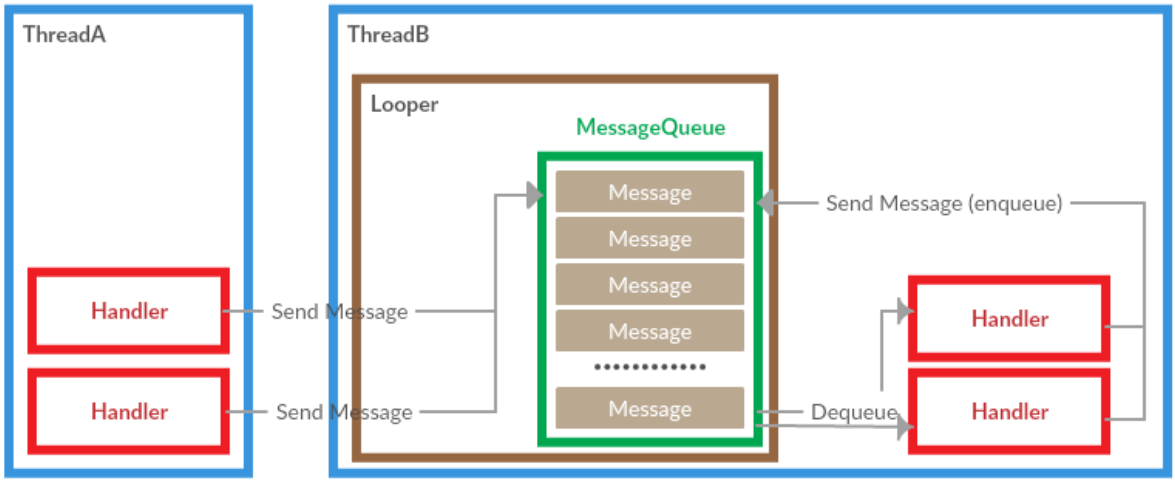
Looper的主要作用是与当前线程形成一种绑定的关系，同时创建一个MessageQueue，这样保证一个线程只能持有一个Looper和MessageQueue同时Looper使得MessageQueue循环起来，就好比水车和水一样，MessageQueue好比水车，当他有了Looper的时候，那么水车会随着水去转动也就是说Looper为MessageQueue提供了活力，使其循环起来，循环的动力往往就少不了Thread。一般而言子线程一般是没有MessageQueue，因此为了使线程和消息队列能够关联那么就需要有Looper来完成了。

默认情况下android中新诞生的线程是没有开启消息循环的。（主线程除外，主线程系统会自动为其创建Looper对象，开启消息循环。） Looper对象通过MessageQueue来存放消息和事件。一个线程只能有一个Looper，对应一个MessageQueue。 主线程（即UI线程）自身就有message loop，不需要创建，而其他线程就需要手动创建，使用prepare()创建loop，使用 loop()来启动loop，直到loop停止。

```
ChildrenLooperThread childerLooperThread=new ChildrenLooperThread();
childerLooperThread.start();
public class ChildrenLooperThread extends Thread{
     @Override
     public void run() {
          Looper.prepare();//创建Looper
          mHandler=new Handler(){
                @Override
                public void handleMessage(Message msg) {
                     super.handleMessage(msg);
                    //...
                }
          };
          Looper.loop();//运行
          super.run();
     }
}
```

3.Handler 与 Message

handler起到了处理MQ上的消息的作用（只处理由自己发出的消息），即通知MQ它要执行一个任务(sendMessage)，并在loop到自己的时候执行该任务(handleMessage)，整个过程是异步的。handler创建时会关联一个looper，默认的构造方法将关联当前线程的looper，不过这也是可以set的。




```
        Handler handler = new Handler(new Handler.Callback() {
            @Override
            public boolean handleMessage(@NonNull Message msg) {
                switch (msg.what){
                    case 1:
                        String data = (String)msg.obj;
                        //TODO:处理数据、更新UI
                        break;
                    default:
                        break;
                }
                return false;
            }
        });
        new Thread(new Runnable() {
            @Override
            public void run() {
                //TODO:线程代码
                //TODO:getData
                String rs = "data";
                Message message = Message.obtain();
                message.what = 1;
                message.obj = rs;
                handler.sendMessage(message);
            }
        });
```

- arg1 和 arg2 都是Message自带的用来传递一些轻量级存储int类型的数据，比如进度条的数据等。
- obj 是Message自带的Object类型对象，用来传递一些对象。兼容性最高避免对齐进行类型转换等。
- replyTo 是作为线程通信的时候使用.
- what 用户自定义的消息码让接受者识别消息种类,int类型。

【注意】: 获得Message的构造方法最好的方式是调用Message.obtain() 和 Handler.obtainMessage() 方法。以便能够更好被回收池所回收。

4.2.3 Asynctask（已被弃用）

Asynctask是Android中的一个自带的轻量级异步类，通过他可以轻松的实现工作线程和UI线程之间的通信和线程切换（其实也只能在工作和ui线程之间切换）

不需使用“任务线程（如继承`Thread`类） + `Handler`”的复杂组合，采用线程池的缓存线程 + 复用线程，避免了频繁创建 & 销毁线程所带来的系统资源开销。

Asynctask是一个抽象类，所以需要创建他的子类，一般重写他的四个方法:

```
//这个就是要在后台做的工作,他将运行在后台工作线程上
@WorkerThread
protected abstract Result doInBackground(Params... params);

//这个时开始执行前的操作，运行在主线程上
@MainThread
protected void onPreExecute() ;

//doInBackground完成后调用
@MainThread
protected void onPostExecute(Result result)

//实时更新，通过在doInBackground中调用publishProgress()方法
@MainThread
protected void onProgressUpdate(Progress... values)
```

4.2.4 Loader

Android 3.0 中引入了 Loader 机制，让开发者能轻松在 Activity 和 Fragment 中异步加载数据,Loader 机制一般用于数据加载，特别是用于加载 ContentProvider 中的内容，比起 Handler + Thread 或者 AsyncTask 的实现方式，Loader 机制能让代码更加的简洁易懂，而且是 Android 3.0 之后最推荐的加载方式。

Loader 机制的 使用场景 有:

- 展现某个 Android 手机有多少应用程序
- 加载手机中的图片和视频资源
- 访问用户联系人

五、网络

5.1 HttpClient（Android6.0已被弃用）

5.2 HttpURLConnection

网络权限:

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

首先需要获取到 HttpURLConnection 的实例，一般只需 new 出一个 URL 对象，并传入目标的网络地址，然后调用一下 openConnection()方法即可，如下所示:

```
URL url = new URL("http://www.baidu.com");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
```

在得到了 HttpURLConnection 的实例之后，我们可以设置一下 HTTP 请求所使用的方法。常用的方法主要有两个：GET 和 POST。GET 表示希望从服务器那里获取数据，而 POST 则表示希望提交数据给服务器。写法如下：

```
connection.setRequestMethod("GET");
```

接下来就可以进行一些自由的定制了，比如设置连接超时、读取超时的毫秒数，以及服务器希望得到的一些消息头等。这部分内容根据自己的实际情况进行编写，示例写法如下：

```
connection.setConnectTimeout(8000);

connection.setReadTimeout(8000);
```

之后再调用 getInputStream()方法就可以获取到服务器返回的输入流了，剩下的任务就是对输入流进行读取，如下所示：

```
InputStream in = connection.getInputStream();
```

最后可以调用 disconnect()方法将这个 HTTP 连接关闭掉，如下所示：

```
connection.disconnect();
```

5.3 OKhttp

GitHub - square/okhttp: Square's meticulous HTTP client for the JVM, Android, and GraalVM.

依赖：

```
implementation("com.squareup.okhttp3:okhttp:4.9.2")
```

5.3.1 异步Get请求

```
String url = "http://wwww.baidu.com";
OkHttpClient okHttpClient = new OkHttpClient();
final Request request = new Request.Builder()
        .url(url)
        .get()//默认就是GET请求，可以不写
        .build();
Call call = okHttpClient.newCall(request);
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: ");
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, "onResponse: " + response.body().string());
```

```
    }
});
```

5.3.2 Post提交Json

```
MediaType mediaType = MediaType.parse("text/x-markdown; charset=utf-8");
String requestBody = "I am Jdqm.";
Request request = new Request.Builder()
        .url("https://api.github.com/markdown/raw")
        .post(RequestBody.create(mediaType, requestBody))
        .build();
OkHttpClient okHttpClient = new OkHttpClient();
okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, response.protocol() + " " +response.code() + " " +
response.message());
        Headers headers = response.headers();
        for (int i = 0; i < headers.size(); i++) {
            Log.d(TAG, headers.name(i) + ":" + headers.value(i));
        }
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

5.4 JSON

5.4.1 Gson

依赖:

```
implementation 'com.google.code.gson:gson:2.8.8'
```

对象序列化:

```
//创建对象
ToJsonBeanOne toJsonBeanOne = new ToJsonBeanOne(1, "小熊", 21);
//将对象转换为json数据
Gson gson = new Gson();
gson.toJson(toJsonBeanOne);
```

对象反序列化:

```
Gson gson = new Gson();
//将json数据转换为对象
ToJsonBeanOne beanOne = gson.fromJson(jsonString, ToJsonBeanOne.class);
```

List序列化:

```
Gson gson = new Gson();
List<Integer> list = Arrays.asList(1, 2, 3);
gson.toJson(list);
```

List反序列化（泛型）：

```
Type type = new TypeToken<List<ToJsonBeanOne>>(){}.getType();
List<ToJsonBeanOne> beanOnes = gson.fromJson(jsonString, type);
```

5.4.2 FastJson

依赖：

```
implementation 'com.alibaba:fastjson:1.1.54.android'
```

序列化：

```
String jsonString = JSON.toJSONString(obj);
```

反序列化：

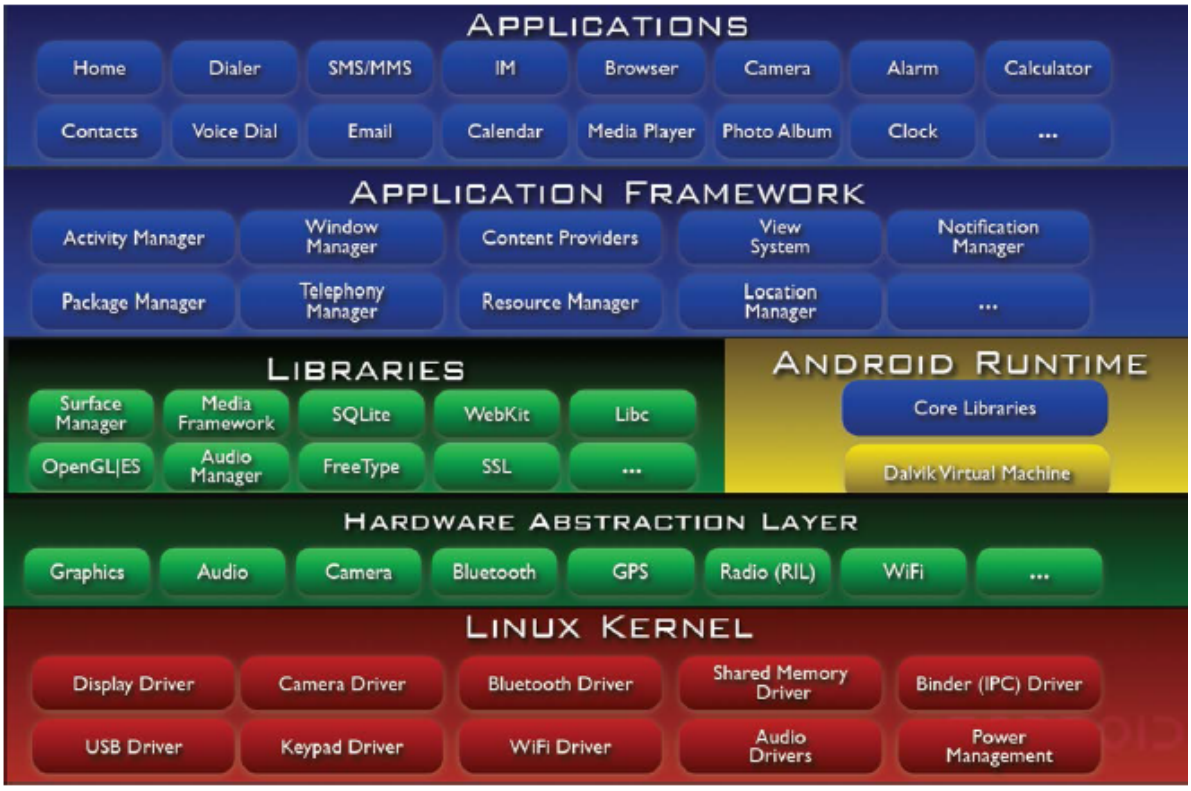
```
VO vo = JSON.parseObject("...", VO.class);
```

泛型反序列化：

```
import com.alibaba.fastjson.TypeReference;

List<VO> list = JSON.parseObject("...", new TypeReference<List<VO>>() {});
```

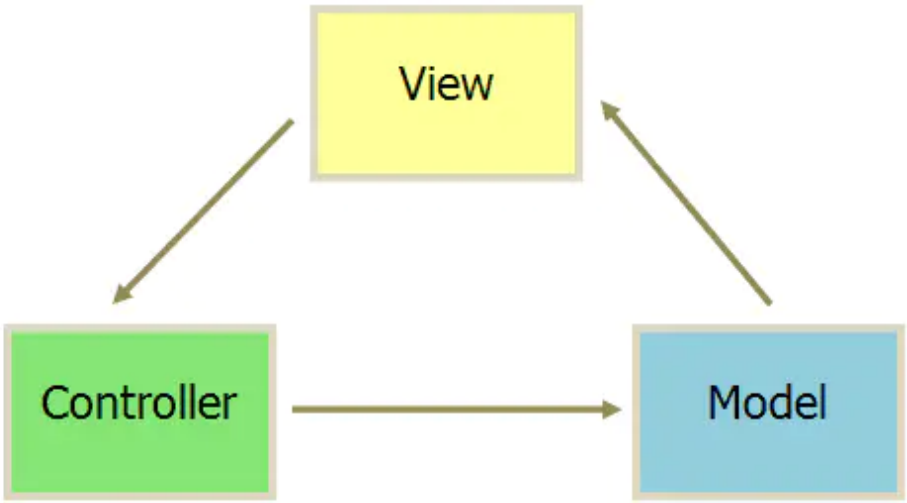
六、其他

1.Android architecture



2.一些名词解释

MVC

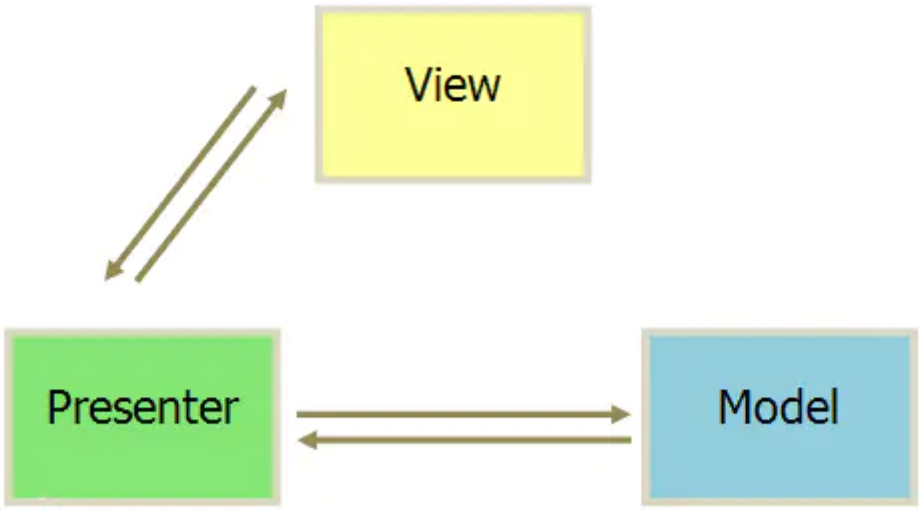


Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。
通常模型对象负责在数据库中存取数据。

View（视图）是应用程序中处理数据显示的部分。
通常视图是依据模型数据创建的。

Controller（控制器）是应用程序中处理用户交互的部分。
通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVP



MVP 是从经典的模式MVC演变而来，它们的基本思想有相通的地方Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。

MVVM

MVVM是Model-View-ViewModel的简写。它本质上就是MVC 的改进版。MVVM 就是将其中的View 的状态和行为抽象化，让我们将视图 UI 和业务逻辑分开。

模型

*模型*是指代表真实状态内容的领域模型（面向对象），或指代表内容的数据访问层（以数据为中心）。

视图

就像在MVC和MVP模式中一样，视图是用户在屏幕上看到的结构、布局和外观（UI）。

视图模型

*视图模型*是暴露公共属性和命令的视图的抽象。MVVM没有MVC模式的控制器，也没有MVP模式的 presenter，有的是一个*绑定器*。在视图模型中，绑定器在视图和数据绑定器之间进行通信。

绑定器

声明性数据和命令绑定隐含在MVVM模式中。

ANR

ANR是指在Android上，应用程序响应不够灵敏时，系统会向用户显示的一个对话框。用户可以选择"等待"而让程序继续运行，也可以选择"强制关闭"。所以一个流畅的合理的应用程序中不能出现anr，而让用户每次都要处理这个对话框。因此，在程序里对响应性能的设计很重要，这样系统不会显示ANR给用户。默认情况下，在android中Activity的最长执行时间是5秒，BroadcastReceiver的最长执行时间则是10秒。

ORM

对象关系映射（英语：**Object Relational Mapping**），是一种程序设计技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的"虚拟对象数据库"。对于数据的操作，我们无需再去编写原生sql，取代代之的是基于面向对象的思想去编写类、对象、调用相应的方法等，ORM会将其转换/映射成原生SQL。

OOM

OOM，全称"Out Of Memory"，来源于java.lang.OutOfMemoryError

当JVM因为没有足够的内存来为对象分配空间并且垃圾回收器也已经没有空间可回收时，就会抛出这个error（注：非exception，因为这个问题已经严重到不足以被应用处理）

NDK

NDK 是一组使您能将 C 或 C++嵌入到 Android 应用中的工具。能够在 Android 应用中使用原生代码对于想执行以下一项或多项操作的开发者特别有用：

- 在平台之间移植其应用。
- 重复使用现有库，或者提供其自己的库供重复使用。
- 在某些情况下提高性能，特别是像游戏这种计算密集型应用

JNI

JNI，全称为Java Native Interface，即Java本地接口，JNI是Java调用Native 语言的一种特性。通过JNI可以使得Java与C/C++进行交互。

- 在Java代码中调用C/C++等语言的代码
- 在C/C++代码中调用Java代码

Native语言

一个Native Method就是一个Java调用非Java代码的接口。方法的实现由非Java语言实现，比如C或C++。

内存泄露

申请使用完的内存没有释放，导致虚拟机不能再次使用该内存，此时这段内存就泄露了，因为申请者不用了，而又不能被虚拟机分配给别人用。

内存溢出

申请的内存超出了JVM能提供的内存大小，此时称之为溢出

URI

统一资源标识符（Uniform Resource Identifier，URI)是一个用于标识某一互联网资源名称的字符串。该种标识允许用户对任何的资源通过特定的协议进行交互操作。URI由包括确定语法和相关协议的方案所定义。

Web上可用的每种资源 -HTML文档、图像、视频片段、程序等 - 由一个通用资源标识符（Uniform Resource Identifier, 简称"URI"）进行定位。

AIDL

AIDL是Android中IPC（Inter-Process Communication）方式中的一种，AIDL是Android Interface definition language的缩写，一般AIDL的作用是让你可以在自己的APP里绑定一个其他APP的service，这样你的APP可以和其他APP交互。

热补丁

热补丁，又称为patch，指能够修复软件漏洞的一些代码，是一种快速、低成本修复产品软件版本缺陷的方式。让应用能够在无需重新安装的情况实现更新，帮助应用快速建立动态修复能力。总的来说，在Android上的期望是，在无需打扰用户手动安装的情况下，实现应用的更新。

Dalvik虚拟机

Dalvik是Google公司自己设计用于Android平台的Java虚拟机，它是Android平台的重要组成部分，支持dex格式（Dalvik Executable）的Java应用程序的运行。dex格式是专门为Dalvik设计的一种压缩格式，适合内存和处理器速度有限的系统。Google对其进行了特定的优化，使得Dalvik具有高效、简洁、节省资源的特点。从Android系统架构图知，Dalvik虚拟机运行在Android的运行时库层。

Dalvik模式与ART模式

Dalvik模式：因为Dalvik虚拟机的存在，Android系统的开发者只需使用谷歌提供的SDK即可较为轻松的按照一套“规则”创建APP，不用顾忌硬件、驱动等问题，在**每次**执行应用的时候Dalvik虚拟机都会将程序的语言由高级语言编译为机器语言，这样当前设备才能够运行这一应用。

ART(Androidruntime)模式与Dalvik模式最大的不同在于，在启用ART模式后，系统在安装应用的时候会进行一次**预编译**，在安装应用程序时会先将代码转换为机器语言**存储在本地**，这样在运行程序时就不会每次都进行一次编译了，执行效率也大大提升。

Dex文件

Android程序一般使用Java语言开发，但是Dalvik虚拟机并不支持直接执行JAVA字节码，因此要进行编译生成的.class文件进行翻译，解释，压缩等处理，这个处理过程是由dx进行处理，处理完成后生成的生成会以.dex结尾，称为Dex文件。Dex文件格式是专为Dalvik设计的一种压缩格式。所以可以简单的理解为：Dex文件是很多.class文件处理后的纹理，最终可以在Android运行时环境中执行。

3.Android应用安装及运行过程：

（1）应用的apk包被安装器调用，检查应用签名后，解析apk中的AndroidManifest，记录权限声明和四大组件等信息，解压出lib中的so库；
（2）桌面应用收到应用安装信息，解析apk获得应用图标显示出来，并存储启动Activity信息；
（3）当桌面点击该应用运行时，调用启动Activity
（4）由系统启动应用的进程，默认进程名是应用安装时写在AndroidManifest中的包名
（5）应用进程加载apk中dex里的代码，加载必要的资源
（6）显示启动Activity页面

4.热补丁运行流程

热补丁所做的改动都集中在应用进程启动后，在加载代码和资源时做必要的替换，大体流程是引导类，检查是否有必要升级，需要的话替换代码和资源。
（1）引导类Application
应用进程能控制的代码入口是Application，一般在attachBaseContext中和onCreate中处理。Amigo提供编译插件，在基本不改变现有代码的情况下，替换Application类为Amigo类，在attachBaseContext中校验当前正在使用的版本，判断是否需要升级，从新apk中解压代码和so，最后再调用应用原Application；而Tinker也替换Application，但通过提供注解，生成类似的代理来实现，需要修改使用了自己Application类的代码。另外在Tinker中dex是补丁包，需要在独立进程中重新合成新dex，之后才能使用。
（2）代码替换
在attachBaseContext中反射获取LoadedApk，并替换mClassloader为AmigoClassLoader，继承DexClassLoader，在创建时添加了新的dex，并指定了新so的路径，覆写了findResource，loadClass方法。
（3）资源替换
在attachBaseContext中反射获取资源引用，创建新的AssetManager并替换。

5.多渠道打包理解

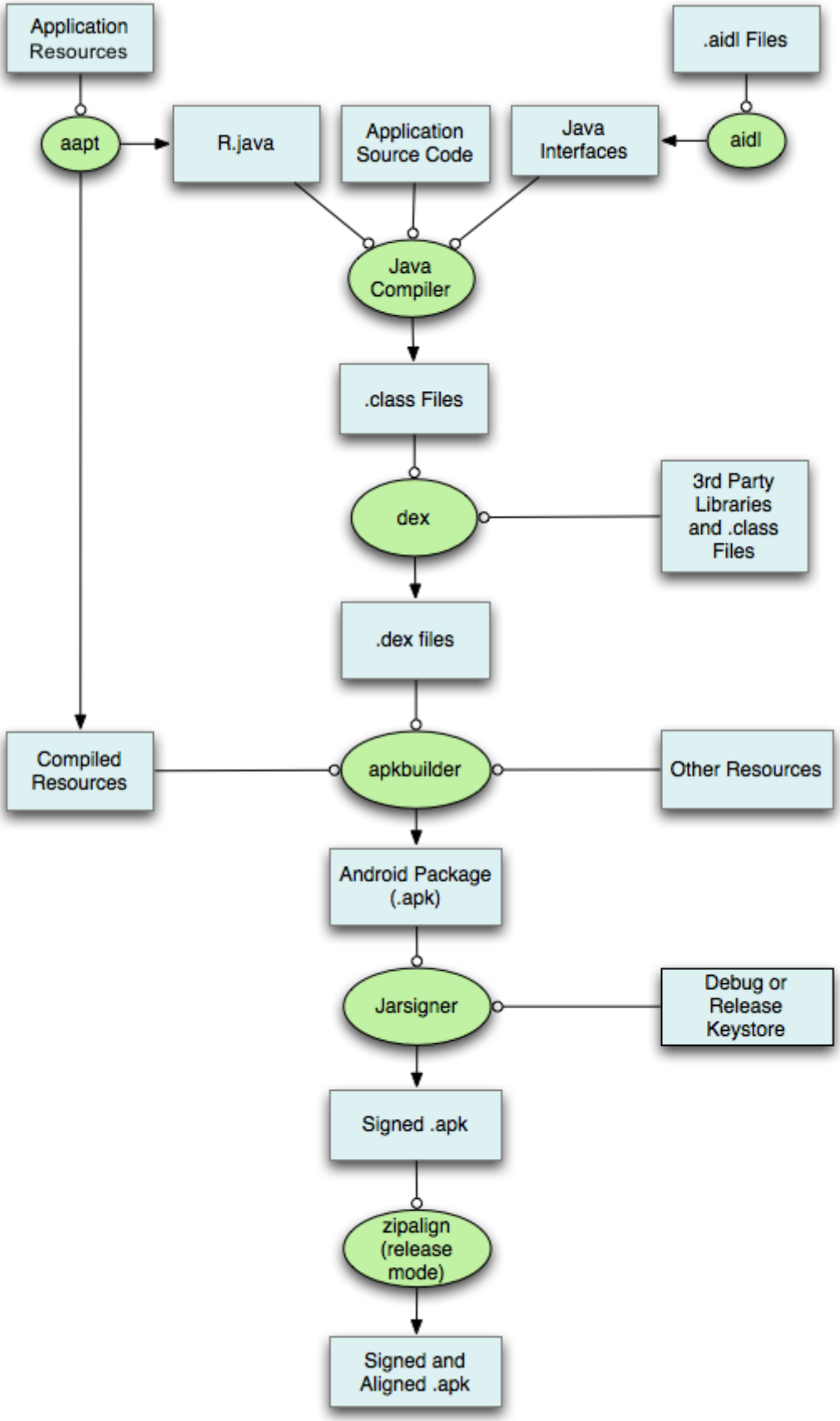
1.统计用户安装APP来源

2.批量修改生成的apk文件名

3.可更改包名

4.生成不同应用名称或图标

6.APK打包流程


Application Resources
.aidl Files
aapt
R.java
Application Source Code
Java Interfaces
aidl
Java Compiler
.class Files
dex
3rd Party Libraries and .class Files
.dex files
Compiled Resources
apkbuilder
Other Resources
Android Package (.apk)
Jarsigner
Debug or Release Keystore
Signed .apk
zipalign (release mode)
Signed and Aligned .apk

7.运行时异常

运行时异常指运行时动态检查和处理的异常，如NullPointerException、IndexOutOfBoundException、BufferOverflowException等。

非运行时异常，在编码过程中就需要try catch或throw的异常，如IOException、SQLException、FileNotFoundException等。

七、框架

1.Lifecycle 生命周期管理

2.navigation 碎片导航

3.PagerBottomTabStrip 底部导航栏

4.SmartRefreshLayout 下拉/上拉刷新

5.MVVMHabit MVVM框架

Room

自增id：

```
@PrimaryKey(autoGenerate = true)
```

Glide

Tip：

基本布局

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <androidx.appcompat.widget.Toolbar
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#EDEDED"
        >
        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:id="@+id/toolbar_title"
```

```
            android:gravity="center"
            android:textSize="18sp"
            android:textStyle="bold"
            />

    </androidx.appcompat.widget.Toolbar>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:id="@+id/main_activity_content"
        />

    <com.ashokvarma.bottomnavigation.BottomNavigationBar
        android:id="@+id/bottom_navigation_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"/>

</LinearLayout>
```

沉浸式状态栏

theme：

```
<item name="android:windowTranslucentStatus">true</item>
```

底部边框

新建xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android" >
    <item> <!-- 格子间的连接处颜色 -->
        <shape>
            <solid android:color="#dddddd" />
        </shape>
    </item>
    <item android:bottom="1dp"> <!--设置底部有边框-->
        <shape>
            <solid android:color="#ffffff" />  <!-- 主体背景颜色 -->
        </shape>
    </item>
</layer-list>
```

EditText边框

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- 角度 -->
    <corners android:radius="100dp"/>
    <!-- 填充色 -->
    <solid android:color="#ffffff"/>
    <!-- 描边 设置线宽及颜色 -->
    <stroke android:color="#cccacb"
        android:width="1dp"/>
</shape>
```

EditText取消下划线

```
android:background="@null"
```

拨打电话

```
mPhoneBtn.setOnClickListener(v->{
    String number = "18223345669";
    Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:"+number));
    startActivity(intent);
});
```

调用相册/相机

```
mAlbumBtn.setOnClickListener(v->{//相册
    Intent intent = new Intent();
    intent.setType("image/*");
    intent.setAction(Intent.ACTION_GET_CONTENT);
    startActivityForResult(intent, 1);

});

 mPhotographBtn.setOnClickListener(v->{//相机
            startActivityForResult(new
Intent("android.media.action.IMAGE_CAPTURE"), 2);
        });

 @Override
    public void onActivityResult(int requestCode, int resultCode, @Nullable
Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if(requestCode == 1 && data != null){//相册选相片
            Bitmap bitmap= null;
            try {
                mBitmap = BitmapFactory.decodeStream

(getActivity().getContentResolver().openInputStream(data.getData()));
```

```
                mPhotoView.setImageBitmap(mBitmap);
                mPhotoLayout.setVisibility(View.VISIBLE);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
        }
        if(requestCode == 2 && data != null){//拍照
            mBitmap = (Bitmap) data.getExtras().get("data");
            mPhotoView.setImageBitmap(mBitmap);
            mPhotoLayout.setVisibility(View.VISIBLE);
        }

    }
```

黄油刀

1.添加依赖

```
//gradle
classpath 'com.jakewharton:butterknife-gradle-plugin:10.2.1'
```

```
//gradle.app
apply plugin: 'com.jakewharton.butterknife'//android下

implementation 'com.jakewharton:butterknife:10.2.1'
annotationProcessor 'com.jakewharton:butterknife-compiler:10.2.1'
```

2.注册Bind

```
//Activity中绑定
public class MainActivity extends AppCompatActivity{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //绑定初始化ButterKnife
        ButterKnife.bind(this);
    }
}
```

```
//Fragment中绑定
public class ButterknifeFragment extends Fragment{
    private Unbinder unbinder;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment, container, false);
        //返回一个Unbinder值（进行解绑），注意这里的this不能使用getActivity()
        unbinder = ButterKnife.bind(this, view);
        return view;
    }

    /**
```

```
     * onDestroyView中进行解绑操作
     */
    @Override
    public void onDestroyView() {
        super.onDestroyView();
        unbinder.unbind();
    }
}
```

```
//Adapter中绑定
public class MyAdapter extends BaseAdapter {

  @Override
  public View getView(int position, View view, ViewGroup parent) {
    ViewHolder holder;
    if (view != null) {
      holder = (ViewHolder) view.getTag();
    } else {
      view = inflater.inflate(R.layout.testlayout, parent, false);
      holder = new ViewHolder(view);
      view.setTag(holder);
    }

    holder.name.setText("Donkor");
    holder.job.setText("Android");
    // etc...
    return view;
  }

  static class ViewHolder {
    @BindView(R.id.title) TextView name;
    @BindView(R.id.job) TextView job;

    public ViewHolder(View view) {
      ButterKnife.bind(this, view);
    }
  }
}
```

3.使用

```
@BindView(R.id.express_item) RelativeLayout mExpress;
```

Fragment跳转

```
    public void  replaceFragment(Fragment fragment){
        FragmentManager fragmentManager =
getActivity().getSupportFragmentManager();
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        transaction.replace(R.id.main_activity_content,fragment);
        transaction.addToBackStack(null);
        transaction.commit();
    }
```

TextView 字体加粗

```
mTextView.setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));//加粗
mTextView.setTypeface(Typeface.defaultFromStyle(Typeface.NORMAL));//取消加粗
```

RecyclerView 单选一个条目

```
package cn.edu.swu.oldcontact.ui.life;

import android.graphics.Typeface;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;

import java.util.List;

import butterknife.BindView;
import butterknife.ButterKnife;
import cn.edu.swu.oldcontact.R;
import cn.edu.swu.oldcontact.javaBean.ContactClassifyItem;

public class LifeClassifyAdapter extends
RecyclerView.Adapter<LifeClassifyAdapter.ViewHolder> {
    private List<ContactClassifyItem> mItemList;


    static class ViewHolder extends RecyclerView.ViewHolder {
        @BindView(R.id.text)
        TextView mItemView;

        public ViewHolder(View view) {
            super(view);
            ButterKnife.bind(this,view);
        }
    }

    public LifeClassifyAdapter(List<ContactClassifyItem> itemList) {
        mItemList = itemList;
    }

    public interface OnItemClickListener {
        void onClick(int position);
    }

    private OnItemClickListener listener;

    public void setOnItemClickListener(OnItemClickListener listener) {
        this.listener = listener;
    }
```

```
    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
        View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.recycler_life_classify
, parent, false);
        ViewHolder holder = new ViewHolder(view);
        return holder;
    }

    @Override
    public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
        ContactClassifyItem item = mItemList.get(position);
        holder.mItemView.setText(item.getTitle());

        if(item.getFlag() == 0){

 holder.mItemView.setTypeface(Typeface.defaultFromStyle(Typeface.NORMAL));
        }else {

 holder.mItemView.setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));
        }

        holder.itemView.setOnClickListener(v->{

            if (listener != null) {
                listener.onClick(position);
            }

            for(int i=0;i<mItemList.size();i++){
                mItemList.get(i).setFlag(0);
            }
            mItemList.get(position).setFlag(1);
            notifyDataSetChanged();
        });
    }

    @Override
    public int getItemCount() {
        return mItemList.size();
    }
}
```

BottomSheetDialog

1.新建dialog布局

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="80dp">

    <TextView
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:gravity="center"
        android:text="社交"
        android:textSize="24sp" />

    <TextView
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:gravity="center"
        android:text="生活"
        android:textSize="24sp" />

</LinearLayout>
```

2.初始化

```
BottomSheetDialog bottomSheetDialog = new BottomSheetDialog(this);
bottomSheetDialog.setContentView(R.layout.dialog_bottom_sheet);
bottomSheetDialog.show();
```

下拉刷新

SmartRefreshLayout: 下拉刷新、上拉加载、RefreshLayout、OverScroll，Android智能下拉刷新框架，支持越界回弹，具有极强的扩展性，集成了几十种炫酷的Header和 Footer。 (gitee.com)

依赖

```
implementation  'com.scwang.smart:refresh-layout-kernel:2.0.3'
implementation  'com.scwang.smart:refresh-header-classics:2.0.3'
implementation  'com.scwang.smart:refresh-footer-classics:2.0.3'
```

添加布局文件

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.life.LifeFragment"
    android:background="#F1F1f1"
    >

    <com.scwang.smart.refresh.layout.SmartRefreshLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/refreshLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <com.scwang.smart.refresh.header.ClassicsHeader
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"/>

    <androidx.recyclerview.widget.RecyclerView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/life_recycler"
        />
        <com.scwang.smart.refresh.footer.ClassicsFooter
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </com.scwang.smart.refresh.layout.SmartRefreshLayout>

</RelativeLayout>
```

3.添加初始化代码

```
mRefresh.setRefreshHeader(new ClassicsHeader(getContext()));
mRefresh.setRefreshFooter(new ClassicsFooter(getContext()));
mRefresh.setOnRefreshListener(new OnRefreshListener() {
    @Override
    public void onRefresh(RefreshLayout refreshlayout) {
        refreshlayout.finishRefresh(2000/*,false*/);//传入false表示刷新失败
    }
});
mRefresh.setOnLoadMoreListener(new OnLoadMoreListener() {
    @Override
    public void onLoadMore(RefreshLayout refreshlayout) {
        refreshlayout.finishLoadMore(2000/*,false*/);//传入false表示加载失败
    }
});
```

Glide圆角图片

依赖:

```
implementation  'com.github.bumptech.glide:glide:3.7.0'
```

实现类:

```
package cn.edu.swu.oldcontact.utils;

import android.content.Context;
import android.content.res.Resources;
import android.graphics.Bitmap;
import android.graphics.BitmapShader;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.RectF;

import com.bumptech.glide.load.engine.bitmap_recycle.BitmapPool;
import com.bumptech.glide.load.resource.bitmap.BitmapTransformation;

/**
 * Glide 圆角 Transform
```

```
 */

public class GlideRoundTransform extends BitmapTransformation {

    private static float radius = 0f;

    /**
     * 构造函数 默认圆角半径 4dp
     *
     * @param context Context
     */
    public GlideRoundTransform(Context context) {
        this(context, 4);
    }

    /**
     * 构造函数
     *
     * @param context Context
     * @param dp 圆角半径
     */
    public GlideRoundTransform(Context context, int dp) {
        super(context);
        radius = Resources.getSystem().getDisplayMetrics().density * dp;
    }

    @Override
    protected Bitmap transform(BitmapPool pool, Bitmap toTransform, int
outWidth, int outHeight) {
        return roundCrop(pool, toTransform);
    }

    private static Bitmap roundCrop(BitmapPool pool, Bitmap source) {
        if (source == null) return null;

        Bitmap result = pool.get(source.getWidth(), source.getHeight(),
Bitmap.Config.ARGB_8888);
        if (result == null) {
            result = Bitmap.createBitmap(source.getWidth(), source.getHeight(),
Bitmap.Config.ARGB_8888);
        }

        Canvas canvas = new Canvas(result);
        Paint paint = new Paint();
        paint.setShader(new BitmapShader(source, BitmapShader.TileMode.CLAMP,
BitmapShader.TileMode.CLAMP));
        paint.setAntiAlias(true);
        RectF rectF = new RectF(0f, 0f, source.getWidth(), source.getHeight());
        canvas.drawRoundRect(rectF, radius, radius, paint);
        return result;
    }

    @Override
    public String getId() {
        return getClass().getName() + Math.round(radius);
    }
}
```

使用：

```
Glide.with(context)
        .load(item.getBgImg())
        .transform(new CenterCrop(context), new GlideRoundTransform(context,10))
        .into(holder.mImg);
```

Fragment销毁

相当于按下返回键

```
mBack.setOnClickListener(v->{
    getActivity().onBackPressed();
});
```

Fragment中获取Activity控件

```
getActivity().findViewById()
```

Activity中获取Fragment控件

```
mFragment.getView().findViewById()
```

List倒序

```
Collections.reverse(itemList);
```

Id转Bitmap

```
Bitmap bitmap = BitmapFactory. decodeResource
(mContext.getResources(),item.getHeadImg());
```

Id转Drawable

```
1.根据资源名获取资源ID(name:资源名；"drawable":资源类型；"entry.dsa":包名)
int imgID = getResources().getIdentifier(name, "drawable","entry.dsa");

2.根据资源id转Drawable
Drawable drawable = getResources().getDrawable(imgID);
```

NestedScrollView嵌套RecyclerView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".ui.life.LifeContentFragment">

    <androidx.core.widget.NestedScrollView
        android:id="@+id/sv"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:scrollbars="none">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
        >
            <androidx.recyclerview.widget.RecyclerView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@+id/img_recycler"
                />


            <RelativeLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:paddingBottom="10dp"
                android:background="@drawable/edit_bottom_border">
                <TextView
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:id="@+id/title"
                    android:textSize="24sp"
                    android:text="测试标题"
                    android:padding="10dp"
                    android:textColor="@color/black"
                    />

                <TextView
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:id="@+id/content"
                    android:textSize="18sp"
                    android:padding="10dp"
                    android:layout_below="@id/title"
                    android:textColor="@color/black"
                    android:text="这里是测试内容"
                    />

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
```

```
                    android:id="@+id/time"
                    android:textSize="14sp"
                    android:text="编辑于11/20 21:12"
                    android:layout_below="@id/content"
                    />

            </RelativeLayout>

            <androidx.recyclerview.widget.RecyclerView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginTop="10dp"
                android:id="@+id/comment_recycler"/>

        </LinearLayout>

    </androidx.core.widget.NestedScrollView>
</LinearLayout>
```

```
RecyclerView commentRecyclerView = (RecyclerView)
view.findViewById(R.id.comment_recycler);
   commentRecyclerView.setLayoutManager(new LinearLayoutManager(getContext()){
       @Override
       public boolean canScrollVertically() {
           //解决ScrollView里存在多个RecyclerView时滑动卡顿的问题
           //如果你的RecyclerView是水平滑动的话可以重写canScrollHorizontally方法
           return false;
       }
   });
   //解决数据加载不完的问题
   commentRecyclerView.setNestedScrollingEnabled(false);
   commentRecyclerView.setHasFixedSize(true);
   //解决数据加载完成后, 没有停留在顶部的问题
   commentRecyclerView.setFocusable(false);
   LifeContentCommentAdapter adapter2 = new
LifeContentCommentAdapter(lifeCommentList,getContext());
   commentRecyclerView.setAdapter(adapter2);
```

ImageView铺满

```
android:scaleType="fitXY"
```

底部导航栏不被键盘顶上来

Manifest.xml:

```
<activity
    android:name=".MainActivity"
    android:windowSoftInputMode="adjustPan">
```

动态申请权限

权限：

所属权限组	权限
日历	READ_CALENDAR
日历	WRITE_CALENDAR
相机	CAMERA
联系人	READ_CONTACTS
联系人	WRITE_CONTACTS
联系人	GET_ACCOUNTS
位置	ACCESS_FINE_LOCATION
位置	ACCESS_COARSE_LOCATION
麦克风	RECORD_AUDIO
电话	READ_PHONE_STATE
电话	CALL_PHONE
电话	READ_CALL_LOG
电话	WRITE_CALL_LOG
电话	ADD_VOICEMAIL
电话	USE_SIP
电话	PROCESS_OUTGOING_CALLS
传感器	BODY_SENSORS
短信	SEND_SMS
短信	RECEIVE_SMS
短信	READ_SMS
短信	RECEIVE_WAP_PUSH
短信	RECEIVE_MMS
存储	READ_EXTERNAL_STORAGE
存储	WRITE_EXTERNAL_STORAGE

```
public void requestPower() {
//判断是否已经赋予权限
    if (ContextCompat.checkSelfPermission(this,
            Manifest.permission.上表权限字符)
            != PackageManager.PERMISSION_GRANTED) {
```

```
        //如果应用之前请求过此权限但用户拒绝了请求，此方法将返回 true。
        if (ActivityCompat.shouldShowRequestPermissionRationale(this,
                Manifest.permission.上表权限字符)) {//这里可以写个对话框之类的项向用户解释为什么要申请权限，并在对话框的确认键后续再次申请权限.它在用户选择"不再询问"的情况下返回false
        } else {
        //申请权限，字符串数组内是一个或多个要申请的权限，1是申请权限结果的返回参数，在onRequestPermissionsResult可以得知申请结果
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.上表权限字符,}, 1);
        }
    }
}
```

uCorp裁剪图片

```
/**
 * 裁剪图片
 *
 * @param sourceUri
 */
private void startUCrop(Uri sourceUri) {
    File outDir =
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
    if (!outDir.exists()) {
        outDir.mkdirs();
    }
    File outFile = new File(outDir, System.currentTimeMillis() + ".jpg");
    UCrop.Options options = new UCrop.Options();
    //裁剪后图片保存在文件夹中
    Uri destinationUri = Uri.fromFile(outFile);
    UCrop uCrop = UCrop.of(sourceUri, destinationUri);//第一个参数是裁剪前的uri,第二个参数是裁剪后的uri
    uCrop.withAspectRatio(16, 9);//设置裁剪框的宽高比例
    //下面参数分别是缩放,旋转,裁剪框的比例
    options.setAllowedGestures(com.yalantis.ucrop.UCropActivity.ALL,
com.yalantis.ucrop.UCropActivity.NONE, com.yalantis.ucrop.UCropActivity.ALL);
    options.setToolbarTitle("移动和缩放");//设置标题栏文字
    options.setCropGridStrokeWidth(2);//设置裁剪网格线的宽度(我这网格设置不显示，所以没效果)
    //options.setCropFrameStrokeWidth(1);//设置裁剪框的宽度
    options.setMaxScaleMultiplier(3);//设置最大缩放比例
    //options.setHideBottomControls(true);//隐藏下边控制栏
    options.setShowCropGrid(true);  //设置是否显示裁剪网格
    //options.setOvalDimmedLayer(true);//设置是否为圆形裁剪框
    options.setShowCropFrame(true); //设置是否显示裁剪边框(true为方形边框)
    options.setToolbarWidgetColor(Color.parseColor("#ffffff"));//标题字的颜色以及按钮颜色
    options.setDimmedLayerColor(Color.parseColor("#AA000000"));//设置裁剪外颜色
    options.setToolbarColor(Color.parseColor("#000000")); // 设置标题栏颜色
    options.setStatusBarColor(Color.parseColor("#000000"));//设置状态栏颜色
    options.setCropGridColor(Color.parseColor("#ffffff"));//设置裁剪网格的颜色
    options.setCropFrameColor(Color.parseColor("#ffffff"));//设置裁剪框的颜色
    uCrop.withOptions(options);
    /*//裁剪后保存到文件中
```

```
    Uri destinationUri = Uri.fromFile(new
File(Environment.getExternalStorageDirectory() + "/myxmpp/" + "test1.jpg"));
    UCrop uCrop = UCrop.of(sourceUri, destinationUri);
    UCrop.Options options = new UCrop.Options();
    //设置裁剪图片可操作的手势
    options.setAllowedGestures(UCropActivity.SCALE, UCropActivity.ROTATE,
UCropActivity.ALL);
    //设置toolbar颜色
    options.setToolbarColor(ActivityCompat.getColor(this, R.color.orange2));
    //设置状态栏颜色
    options.setStatusBarColor(ActivityCompat.getColor(this, R.color.orange2));
    //是否能调整裁剪框
    options.setFreeStyleCropEnabled(true);
    options.setToolbarWidgetColor(Color.parseColor("#ffffff"));//标题字的颜色以及按
钮颜色
    options.setDimmedLayerColor(Color.parseColor("#AA000000"));//设置裁剪外颜色
    options.setToolbarColor(Color.parseColor("#000000")); // 设置标题栏颜色
    options.setStatusBarColor(Color.parseColor("#000000"));//设置状态栏颜色
    options.setCropGridColor(Color.parseColor("#ffffff"));//设置裁剪网格的颜色
    options.setCropFrameColor(Color.parseColor("#ffffff"));//设置裁剪框的颜色
    //options.setShowCropFrame(false); //设置是否显示裁剪边框(true为方形边框)
    uCrop.withOptions(options);*/
    uCrop.start(getActivity());
}
```

```
@Override
public void onActivityResult(int requestCode, int resultCode, @Nullable Intent
data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode == 1 && data != null){//相册选
        Uri sourceUri = data.getData();
        startUCrop(sourceUri);
    }
}
```

ImageView宽度填充

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:scaleType="fitXY"
    android:adjustViewBounds="true"
    android:layout_gravity="center"
    android:id="@+id/img"/>
```

获取时间

```
private String getDate() {
    Date date = new Date(System.currentTimeMillis());
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年-MM月dd日-HH时mm分ss
秒 E");
    String sim = dateFormat.format(date);

    return sim;
}
```

List排序

```
public static void userListSort(List<User> list){
    Comparator<User> comparator = new Comparator<User>() {
        @Override
        public int compare(User details1, User details2) {
            //排序规则，按照价格由大到小顺序排列("<"),按照价格由小到大顺序排列(">"),
            if(details1.getIntegral() < details2.getIntegral())
                return 1;
            else {
                return -1;
            }
        }
    };
    Collections.sort(list, comparator);
}
```

TextView滚动显示

```
mContent.setMovementMethod(ScrollingMovementMethod.getInstance());
```

EditText失去焦点

```
mEdit.clearFocus();
```

Git命令

代码上传:

```
$ git init      #初始化仓库
$ git add .     #添加当前目录所有文件到缓存
$ git commit -m "xxx"
$ git remote add origin "git@xxxx"(远程git仓库地址)
$ git pull origin master(main)  #注意远程分支名称
$ git push -u origin master
```

快捷键

1.shift+enter 下插一行

2.ctrl + alt + L　整理代码