Activity

活动的生存期

- `onCreate()`：创建活动
- `onStart()`：活动由不可见变为可见
- `onResume()`：由可见变为可以和用户交互，即准备好和用户交互
- `onPause()`：在系统准备调用其他活动的时候调用
- `onStop()`：活动由可见变为不可见
- `onDestory()`：销毁活动
- `onRestart`；当活动处于停止状态时，调用该方法可以用于重新启动活动
- 完整的生存期：`onCreate()`~`onDestory()`
- 可见生存期：`onStart()`~`onStop()`
- 前台生存期：`onResume()`~`onPause()`
- 活动的开始到结束过程：创建->不可见->可见->不可交互->可交互->不可交互->不可见->销毁活动

活动的启动模式

- 默认启动模式`standard`：系统默认是standard模式，这种模式就是每创建一个活动，不会在乎返回栈中是否已经存在，每次启动都会创建一个该活动的实例
- `singleTop`：这种模式在启动活动时，发现返回栈的栈顶已经存在该活动时，不会创建该活动新的实例，直接使用
- `singleTask`：上述问题仅仅解决了在栈顶存在相同活动的实例，但是还可能存在相同的实例。怎么样达到在整个程序中只存在某个活动的一个实例呢？`singleTask`正是解决这种问题的方法。
- `singleInstance`：如果某个活动的启动模式被设为该模式时，则在该活动被启动时，会单独存在于另外一个的返回栈中，即另外创建一个返回栈来管理该活动，如下：

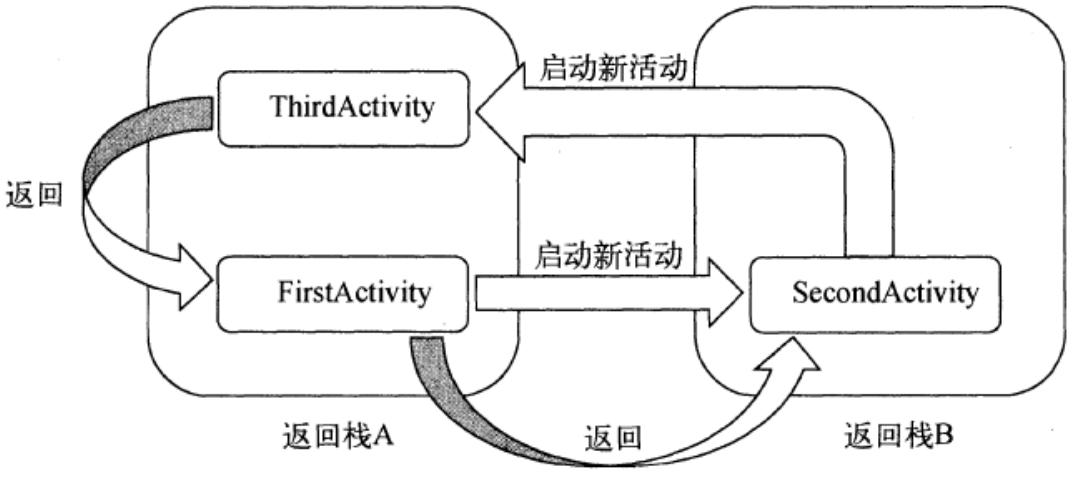

图 2.41　singleInstance 模式示意图

Fragment

- 碎片类`Fragment`
- 碎片的使用

- 新建用于碎片的XML，如left_fragment.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<Button
android:id="@+id/button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:text="Button"
/>
</LinearLayout>
```

- 新建一个碎片LeftFragment类，继承于Fragment

```
public class LeftFragment extends Fragment {
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
View view = inflater.inflate(R.layout.left_fragment, container, false);
return view;
}
}
```

- 在activity_main.xml中使用碎片，

```
<fragment
android:id="@+id/left_fragment"
android:name="com.example.fragmenttest.LeftFragment"
android:layout_width="0dp"
android:layout_height="match_parent"
android:layout_weight="1" />
```

- 动态添加碎片
  - 新建XML，如another_right_fragment.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:background="#ffff00"
android:layout_width="match_parent"
android:layout_height="match_parent">
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:textSize="20sp"
android:text="This is another right fragment"
/>
</LinearLayout>
```

- 新建AnotherRightFragment类，继承于Fragment，使用上述xml文件

```
public class AnotherRightFragment extends Fragment {
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
View view = inflater.inflate(R.layout.another_right_fragment,
container,
false);
return view;
}
}
```

- 在activity_main.xml中新建或者添加一个FrameLayout

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

- 在Framelayout中使用AnotherRightFragment碎片的对象，达到动态添加碎片

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction transaction = fragmentManager.beginTransaction();
transaction.replace(R.id.right_layout, new AnotherRightFragment());
transaction.commit();
```

- 碎片和活动之间的通信
  - 在activity活动中新建一个碎片对象，方法如下：

```
RightFragment rightFragment = (RightFragment)
getSupportFragmentManager()
.findFragmentById(R.id.right_fragment);
```

得到上述碎片的对象之后，就可以使用碎片类中的方法

  - 碎片中使用活动的方法

```
MainActivity activity = (MainActivity) getActivity();
```

通过上述方法得到活动的实例对象后就可以使用活动中方法了

- 碎片的生命周期
  - 运行状态

当一个碎片是可见的，并且其所关联的活动正处于运行状态，该碎片也处于运行状态

  - 暂停状态

当一个活动进入暂停状态时，去其相关的可见碎片将会进入到暂停状态

  - 停止状态

当一个活动进入停止状态时候，与其相关的碎片就会进入停止状态

- 销毁状态

  碎片总是依附于活动存在的，当活动被销毁时，与其相关的碎片也会进入到销毁状态

- 碎片周期中使用的方法：
  - `onAttach()`：当碎片和活动建立关联的时候调用
  - `onCreateView()`：当碎片加载布局时调用
  - `onActivityCreated()`：确保与碎片相关联的活动一定已经创建完毕的时候调用
  - `onDestroyView()`：当与碎片关联的布局被移除的时候调用
  - `onDetach()`：当活动和碎片解除关联的时候调用

Service

- 服务：服务是安卓实现程序后台运行的解决方案，适合执行不需要和用户交互并且长期执行的任务；其次服务中所有代码均是在主线程中执行的，所以需要创建新的子线程去执行具体的任务
- 安卓多线程编程：
  - 继承Thread类：

    ```
    class MyThread extends Thread {
    @Override
    public void run() {
    // 处理具体的逻辑
    }
    }
    ```

    启动线程：

    ```
    new MyThread().start()
    ```

  - 实现Runnable接口

    ```
    class MyThread implements Runnable {
    @Override
    public void run() {
    // 处理具体的逻辑
    }
    }
    ```

    启动线程：

    ```
    MyThread myThread = new MyThread();
    new Thread(myThread).start();
    ```

  - 匿名类：

```
new Thread(new Runnable()
            {
                @override
                public void run()
                {
                    //处理具体的逻辑
                }
            }).start();
```

- 子线程中更新UI
  - 新建一个`Handler`对象，并重写`Handler`类中的`handleMessage(Message msg)`方法，根据接收到的msg消息进行处理。`Handler`专门用来处理消息的，可以发送消息，也可以处理消息，发送一般使用`sendMessage`，处理一般使用`handleMessage`

```
public Handler hander =new Handler(){
    @override
    public void handleMessage(Message msg){
        //处理逻辑
    }

}
```

  - 子线程中新建Message消息，然后通过上述的`Handler`对象进行发送消息。`Message`用于在线程之间传递消息

```
new Thread(new Runnable(){
    @override
    public void run()
    {
        Message msg = new Message();
        msg.what=1;
        hander.sendMessage(msg);//发送消息
    }
}).start();
```

- `AsyncTask`类，这是一个安卓中专门用于处理异步消息的抽象类，需要创建子类并继承，其有三个泛型参数

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {
...
}
```

  - 此外，还需要实现`AsyncTask`类中的如下几个抽象方法
    - `onPreExecute()`：在后台任务开始之前调用
    - `doInBackground(Params...)`：在子线程中执行；用来处理耗时的长时间的任；不能进行UI操作，如有需要，则使用`publishProgress (Progress...)`
    - `onProgressUpdate(Progress...)`：用于接收`publishProgress(Progress...)`传递过来的消息，并进行UI操作
    - `onPostExecute(Result)`:当后台任务执行结束时并通过return返回时，会执行该方法；`doInBackground()`返回的值就是`onPostExecute(Result)`的参数值
  - 启动任务：`new DownloadTask().execute()`

示例一（服务的生命周期，服务和活动通信）调试成功　工程名：ServiceTest

示例二 （服务的前台服务）调试失败　工程名：ServiceTest

示例三 （服务的IntentService）调试成功 工程名：ServiceTest

- 开启服务：服务启动之后，其生命周期独立于启动该服务的组件，并且可以在后台无限期运行，即使启动服务的组件销毁了，也不影响活动。因此，需要通过`stopSelf()`进行销毁服务
  - 启动方法：`startService()`，参数为Intent对象
- 停止服务：`stopService()`
- 绑定服务：`bindService()`,第一个参数是Intent类型，第二个参数是ServiceConnection，第三个参数是BIND_AUTO_CREATE，通过这一步将活动与服务进行关联。相当于在活动中取得了，服务绑定的对象，然后就可以通过绑定的服务调用相应的功能
- 取消绑定服务：`unbindService()`,只有一个参数，参数是ServiceConnection
- 服务的生命周期：
  - `onCreate()`：该方法会在startService之后调用；然后回调下面的onStartCommand；当后面再次调用startService时，如果服务已经创建过，就不会调用onCreate，直接调用onStartCommand；不管后面startService多少次，由于只存在一个服务，因此只需要调用一次stopService或者stopSelf就可以达到停止活动的效果。
  - `onStartCommand()`:该方法会在startService之后调用；然后回调该函数；当后面再次调用startService时，如果服务已经创建过，就不会调用onCreate，直接调用onStartCommand，
  - `onBind()`:该方法会在bindService之后调用；但前提是如果服务还没创建过，会优先调用`onCreate()`；接着调用该函数会返回IBinder的实例对象；最后就可以与服务之间进行通信了
  - `onDestory()`：当调用unbindService后被调用
- `IntentService`类：该类中可以用于创建自动在子线程中运行的程序，提供在子线程中运行的函数为`onHandleIntent()`函数，该函数就是在子线程中运行的，并且还可以在运行完该函数中的逻辑后，自动的调用另一个函数onDestory，停止掉服务。使用方法：新建一个继承于IntentService的类，在该类中重写`onHandleIntent`方法与`onDestory`方法。然后在主线程中通过调用startService就能直接开启服务。
  - 通过这种方法相比手动在继承于Service的类中的onStartcommand方法中通过下述方式要方便多，因为这种方式在run函数运行完后，不会停止服务，需要接着调用stopService或者stopSelf才能将服务停止。

```
public void onStartcommand() {
    new Thread(new Runable(){
    @override
    void run() {

        //添加子线程中运行的逻辑
        stopSelf();//停止服务
    }
}).start();
}
```

BroadcastReceiver

- 接收系统广播
  - 动态注册监听网络变化 调试成功

- 在代码中注册，接收系统的消息
- 创建一个广播接收器，即创建一个类`BroadcastReceiver`，继承于`Receiver`，并重写`onReceiver`方法
- 建立`Interfilter`对象，并将该对象添加`action`，`action`的值就是需要传递的广播，即要传递什么广播
- 接收器中注册`Interfilter`对象

```
public class MainActivity extends AppCompatActivity {
    private IntentFilter intentFilter;
    private NetworkChangeReceiver networkChangeReceiver;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        intentFilter = new IntentFilter();

intentFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        networkChangeReceiver = new NetworkChangeReceiver();
        registerReceiver(networkChangeReceiver, intentFilter);
    }
@Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(networkChangeReceiver);
    }
class NetworkChangeReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "network changes",
Toast.LENGTH_SHORT).show();
    }
}
}
```

  - 静态注册实现开机启动 调试失败
    - 新建一个广播接收器类，该类继承于`BroadcastReceiver`
    - 其次一定要在AndroidManifest.xml文件中注册静态广播接收器，下述中的`android.intent.action.BOOT_COMPLETED`就为该接收器接收的广播值

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

- 发送自定义广播
  - 发送标准广播 调试成功
    - 首先创建一个广播接收器：可以通过上述静态注册方式；此外接收的值一定要是自定义的待传播的广播值

        - 构建Intent对象，并将要发送的广播值传入；
        - 使用`sendBroadcast()`方法将上述的Intent对象发送出去；这样所有监听上述广播值的广播接收器都能接收到该广播

```
Intent intent = new
Intent("com.example.broadcasttest.MY_BROADCAST");
intent.setComponent(new
ComponentName(getPackageName(),"com.example.broadcastt.MyBroadcastR
eceiver"));
sendBroadcast(intent);
```

    - 应用程序内发送的广播是可以被其他应用程序接收到的
    - 发送有序广播 调试成功
        - 使用`sendOrderedBroadcast(intent,null)`发送有序广播
        - 其次，还可以设置接收器接收的先后顺序`android:priority="100"`；并且先接收到的接收器还可以截断广播，让其不能继续传播`abortBroadcast()`，如果在某个接收器的`onReceive()`中添加了`abortBroadcast()`方法，则表明该接收器接受完广播之后就终止传递了

```
<receiver
android:name=".MyBroadcastReceiver"
android:enabled="true"
android:exported="true">
<intent-filter android:priority="100">
<action android:name="com.example.broadcasttest.MY_BROADCAST" />
</intent-filter>
</receiver>
```

- 使用本地广播
    - 为了避免发出的广播被应用程序之外的广播接收器接收，安卓系统中引入了本地广播机制
    - 使用`LocalBroadcastManager`对广播进行管理
    - `LocalBroadcastManager`类中提供`regiserReceiver()`方法用于注册广播接收器，此外还提供`sendBroadcast()`发送广播

```
package com.example.broadcastt;

import androidx.appcompat.app.AppCompatActivity;
import androidx.localbroadcastmanager.content.LocalBroadcastManager;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    private IntentFilter intentFilter;
    private LocalReceiver localReceiver;
    private LocalBroadcastManager localBroadcastManager;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        localBroadcastManager =
LocalBroadcastManager.getInstance(this);
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new
Intent("com.example.broadcastt.LOCAL_BROADCAST");
                //intent.setComponent(new
ComponentName(getPackageName(),"com.example.broadcastt.MyBroadcastRecei
ver"));
                localBroadcastManager.sendBroadcast(intent);
            }
        });

        intentFilter = new IntentFilter();

 intentFilter.addAction("com.example.broadcastt.LOCAL_BROADCAST");
        localReceiver = new LocalReceiver();

 localBroadcastManager.registerReceiver(localReceiver,intentFilter);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        localBroadcastManager.unregisterReceiver(localReceiver);
    }

    class LocalReceiver extends BroadcastReceiver{
        @Override
        public void onReceive(Context context, Intent intent) {
            Toast.makeText(context,"received local
broadcast",Toast.LENGTH_SHORT).show();
        }
    }
}
```

ContentProvider

- 解决的问题是:不同应用app之间的数据传递和共享，android中正式使用ContentProvider将app的数据分享出去
- 安卓运行时权限
- 提供数据给其他应用程序-ContentProvider类，下面是一些在使用过程中，继承于ContentProvider类的子类中需要根据app中特定的数据库进行重写的方法：
    - `insert()`:第一个参数是Uri地址，第二个参数是ContentVaules类型的数据值；依据该Url地址将数据添加进去
    - `delete()`：第一个是Uri地址，根据该Uri地址将地址上的数据删除
    - `query()`：第一个参数同样是Uri地址，根据该地址查询数据

    - `update()`：第一个参数是Uri地址，第二个参数是待更新进去的数据，根据地址将待更新的数据更新进去
- 访问其他程序中的数据-ContentResolver类，要访问ContentProvider

Intent

- 显示`Intent`

```
Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
startActivity(intent);//启动活动
```

- 隐式`Intent`
    - 启动其他程序：

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.baidu.com"));
startActivity(intent);//启动百度
Intent intent = new Intent(Intent.ACTION.DIAL);
intent.setData(Uri.parse("tel:10086"));
startActivity(intent);//启动拨打电话
```

    - 传递数据给下一个活动：

```
//发送
Intent intentSend = new Intent(MainActivity.this,FirstActivity.class);
intentSend.putExtra("extraData","您好");
startActivity(intentSend);
//接收
Intent intentReceive = getIntent();
String res = intentReceive.getStringExtra("extraData");
```

    - 返回数据给上一个活动(当需要在一个活动中得到新打开活动关闭后返回的数据时，需要使用`startActivityForResult`启动活动)
        - 通过`startActivityForResult`，可以用于启动一个活动并且可以在被启动的活动被销毁时，能够返回一个结果给上一个活动；该方法具有两个参数
            - 参数1，是一个Intent对象
            - 参数2，是一个请求码，用于在后面的回调函数中判断数据的来源
        - 在活动中添加重写`onActivityResult`，用于接收被启动活动传输来的数据

```
protected void onActivityResult(int requestCode, int resultCode,
@Nullable Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        switch (requestCode) {
            case 1:
                if (requestCode == RESULT_FIRST_USER) {
                    String res_return =
data.getStringExtra("data_return");
                    Log.d("MainActivity",res_return);
                }
                break;
            default:
                break;
        }
    }
```

- 在被启动活动中返回数据给上一个活动：

```
Intent intent = new Intent();
intent.putExtra("data_return" ,"你好 MainActivity");
setResult(RESULT_OK,intent);
finish();
```

数据存储方案

保存在内存中的数据是瞬时状态的，保存在存储设备中的数据是持久状态的，持久化技术提供了一种机制可以实现数据在瞬时状态和持久化状态进行转换。下面是三种持久化技术：

- 文件存储
  - 将数据存储到文件中
  - 从文件中读取数据
- TIPS:
  - 查看从内存中写入的文件：
    - 进入到cmd界面，进入到Android 文件夹下的tools文件，执行monitor命令
    - 弹出Android Device Monitor软件，查看File Explorer下的文件，如果没有权限，继续下面的步骤修改权限
    - 执行cmd命令，进入Android下的platform-tools文件夹，执行adb shell，紧接着设置su模式(强调;这一步需要运行Android 6模拟器，才能进入模拟器)
    - 下面执行chmod 文件名 777，通过设置权限就可以查看到文件
- SharedPreferences存储
  - 将数据存储到SharedPreferences中
    - SharedPreferences对象的获取方法
      - Context类中的getSharedPreferences()方法
      - Activity中的getPreferences()方法
      - 静态方法：通过构建一个SharedPerference.Editor对象，然后通过该对象添加数据，最后通过apply将数据提交
  - 从SharedPreferences中读取数据
    - 构建SharedPreference对象，通过该对象提供的getString ,getInt,getBoolean方法得到存储的数据
- SQLite数据库存储

    - 创建数据库
    - 升级数据库
    - 添加数据库
    - 更新数据
    - 删除数据
    - 查询数据
    - 使用SQL操作数据库
        - 同样可以实现上面的所有操作
    - 使用LitePal操作数据库
        - 配置LitePal
        - 创建和升级数据库
        - 使用LitePal添加数据
        - 使用LitePal更新数据
        - 使用LitePal删除数据
        - 使用LitePal查询数据

Git

- 创建代码仓库`git init`：在要创建的文件夹中执行该命令
- 添加到仓库
    - `git add .`(添加所有文件)
    - `git add build.grandle`（添加指定文件）
    - `git add app`（添加整个app目录下的文件）
- 提交`git commit -m "First commit"`
- 查看文件变动`git diff app/src/main/java/com/example/providertest/MainActivity.java`
    - 减号代表删除的部分
    - 加号代表添加的部分
- 撤销未添加的修改(这种情况只适合于没有执行add命令的文件)

  `git checkout app/src/main/java/com/example/providertest/MainActivity.java`
- 对已添加的文件进行撤回
    - `git reset HEAD app/src/main/java/com/example/providertest/MainActivity.java`，这一步使得该文件变为未添加状态
    - 撤销修改：

      `git checkout app/src/main/java/com/example/providertest/MainActivity.java`
    - 通过上面两步可以实现对已经添加的文件进行修改
- 查看提交记录
    - 查看所有的提交信息：`git log`
    - 查看一条提交信息：

      `git log 1fa380b502a00b82bfc8d84c5ab5e15b8fbf7dac -1`
    - 查看具体提交中修改了哪些内容：

`git log 1fa380b502a00b82bfc8d84c5ab5e15b8fbf7dac -1 -p`，输出来的信息中，减号代表删除的部分，加号代表添加的部分

手机多媒体

- 使用通知
  - 通知的基本使用
  - 通知的进阶使用
  - 通知的高级使用

网络技术

- `WebView`
- 使用HTTP协议访问网络
  - 使用`HttpURLConnection`：
    -
  - 使用第三方库`OkHttp`

其他

- 取随机数

```
Random random = new Random();
int data = random.nextInt(5);//取0~5之间的随机数，不包括5
```

- `StringBuilder`，字符串累加

```
StringBuilder builder = new StringBuilder();
builder.append("xue");
builder.append("guo");
builder.append("fei");
```

UI设计

- 布局
  - 线性布局：`LinearLayout`
    - `orientation`：用于指定控件的位置排列方向，包括`vertical`和`horizontal`
    - `gravity`：用于设置控件中内容的位置，即内容在控件内部的相对位置，可以设置的参数有：`top`，`bottom`，`center_vertical`,`center_horizontal`，`left`，`right`，`start`，`end`，`clip_horizontal`，`clip_vertical`，`fill`，`fill_vertical`，`fill_horizontal`
    - `layout_gravity`：用于指定若干控件在布局中的对齐方式；需要注意的是，当LinearLayout的布局方向是垂直时，则该项只能在水平方向上设置才能生效；当LinearLayout的布局方向是水平时，则该项只能在垂直方向上设置才能生效。可以设置的值有：`top`，`bottom`，`center_vertical`,`center_horizontal`，`left`，

`right`, `start`, `end`, `clip_horizontal`, `clip_vertical`, `fill`, `fill_vertical`, `fill_horizontal`

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="match_parent">
<Button
android:id="@+id/button1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="top"
android:text="Button 1" />
<Button
android:id="@+id/button2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_vertical"
android:text="Button 2" />
<Button
android:id="@+id/button3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="bottom"
android:text="Button 3" />
</LinearLayout>
```

- `layout_weight`：用于设置在某方向上的权重，当控件水平放置时，并且`layout_width`均设置为0时，按照该项的值除以所有控件该项值的总和，所得的结果就是对应控件的宽度。如

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="match_parent">
<EditText
android:id="@+id/input_message"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:hint="Type something"
/>
<Button
android:id="@+id/send"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:text="Send"
/>
</LinearLayout>
```

当其中一个的layout_width设置为wrap_content，另外一个的layout_width设置为0且layout_weight设置为1时，达到的效果是前者按照设为wrap_content的控件去分配，剩下的控件全部分配给后者

    - 相对布局：`RelativeLayout`
        - 相对于父布局：
            - `layout_alignParentLeft`:位于父布局的最左边，值为true或者false，下面一样
            - `layout_alignParentTop`：位于父布局的最上边
            - `layout_alignParentRight`：位于父布局的最边
            - `layout_alignParentBottom`：位于父布局的最下边
            - `layout_centerInParent`：位于父布局的最中间
        - 相对于控件的布局：
            - `layout_above`：在某控件的上面
            - `layout_below`：在某控件的下面
            - `layout_toLeftOf`：在某控件的左边
            - `layout_toRightOf`：在某控件的右边
        - 控件与控件的对齐：
            - `layout_alignLeft`：此控件的左边缘与另一个控件的右边缘对齐
            - `layout_alignRight`：此控件的右边缘与另一个控件的右边缘对齐
            - `layout_alignTop`：此控件的顶边缘与另一个控件的顶边缘对齐
            - `layout_alignBottom`：此控件的底边缘与另一个控件的底边缘对齐
    - 帧布局：`FrameLayout`
    - 百分比布局：`PercentFrameLayout`
- 高级控件
    - `ListView`
        - `id`
        - `layout_width`
        - `layout_height`
        - 使用

          通过ListView控件设置适配器，适配器加载了数据，达到在ListView控件上显示数据
            - 重写`getView`方法
            - 使用`LayoutInflater`加载布局，得到View
    - `RecyclerView`
        -
- 基础控件
    - `TextView`(用于在界面上显示一段信息)
        - `id`
        - `layout_width`
        - `layout_height`
        - `text`：文字
        - `textSize`：文字的尺寸，如20sp
        - `textColor`：文字的颜色，如`#00ff00`代表绿色
        - `gravity`：文字的对齐方式，如center
    - `Button`
        - `id`

        - `layout_width`
        - `layout_height`
        - `text`
        - `textAllCaps`：系统中对text的文字默认均是大写，如果不想要默认的效果，这一项可以设为false
        - 使用：基于匿名类

```
Button button = (Button)findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener(){
    @override
    public void click(View v) {
        //在此处添加逻辑
    }
});
```

    - `EditText`(允许用户在控件内输入和编辑内容，并可以用于程序中的处理)
        - `id`
        - `layout_width`
        - `layout_height`
        - `hint`:用于提示用户输入的一段文字，当用户输入时会消失
        - `maxLines`：最大行数
        - 使用

```
EditText editText = (EditText)findViewById(R.id.EditText);
//得到EditText中输入的文字，并转换为字符串
String inputText = editText.getText().toString();
```

    - `ImageView`
        - `id`
        - `layout_width`
        - `layout_height`
        - `src`：在控件上显示的图片
        - 使用：

```
ImageView imageView = (ImageView)findViewById(R.id.ImageView);
//动态添加图片
imageView.setImageResource(R.drawble.img_2);
```

    - `ProgressBar`：进度条
        - `id`
        - `layout_width`
        - `layout_height`
        - `style`：如`?android:attr/progressBarStyleHorizontal`表示为水平进度条
        - `max`:进度条的最大值

- 使用

```
int peogress = progressBar.getProgress();//得到当前进度条的值
progressBar.setProgress(50);//设置进度条的值
```

- `AlertDialog`：用于弹出对话框，能够屏蔽掉与其他控件交互

```
AlertDialog.Builder dialog = new
AlertDialog.Builder(MainActivity.this);
dialog.setTitle("this is dialog");//设置标题
dialog.setMessage("something important!");//设置消息
dialog.setCancelable(false);
//两个按钮是和否
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();
```

- `progressDialog`:可以用于显示进度的对话框

```
ProgressDialog progressDialog = new ProgressDialog(this);
progressDialog.setTitle("This is ProgressDialog");
progressDialog.setMessage("something is loading");
progressDialog.show();
```