Android Service

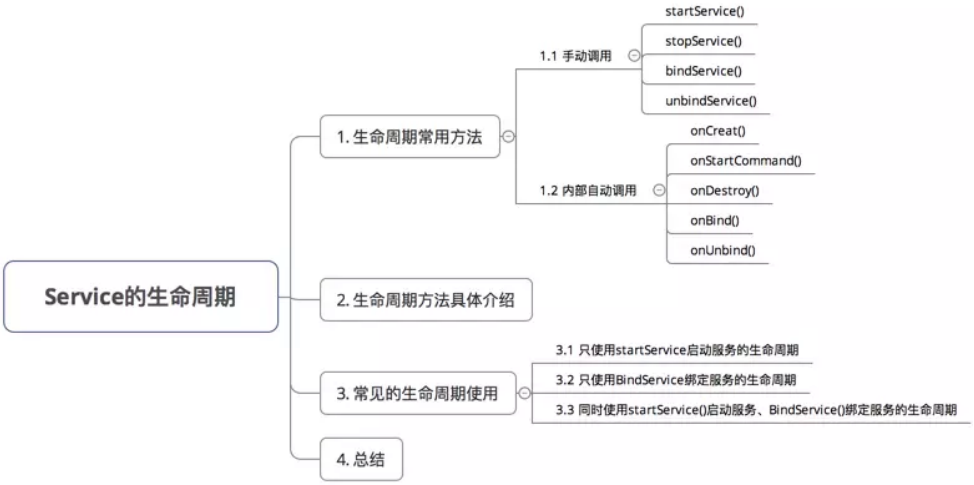
deepwaterooo

2021 年 12 月 15 日

目录

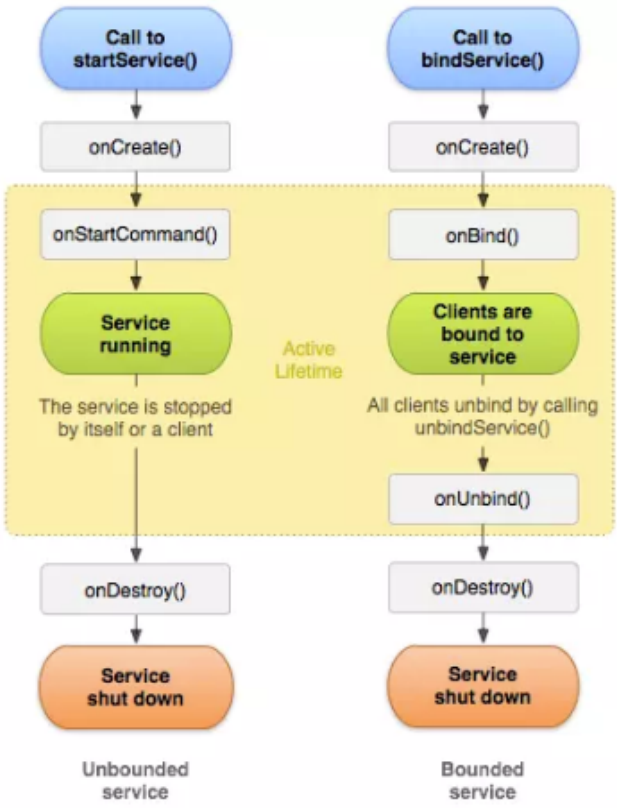
1 Android：Service 生命周期完全解析

- 目录



1.1 生命周期常用方法

官方说明图



- 在 Service 的生命周期里，常用的有：
  - 4 个手动调用的方法

手动调用方法	作用
startService()	启动服务
stopService()	关闭服务
bindService()	绑定服务
unbindService()	解绑服务

- 5 个自动调用的方法

内部自动调用的方法	作用
onCreate()	创建服务
onStartCommand()	开始服务
onDestroy()	销毁服务
onBind()	绑定服务
onUnbind()	解绑服务

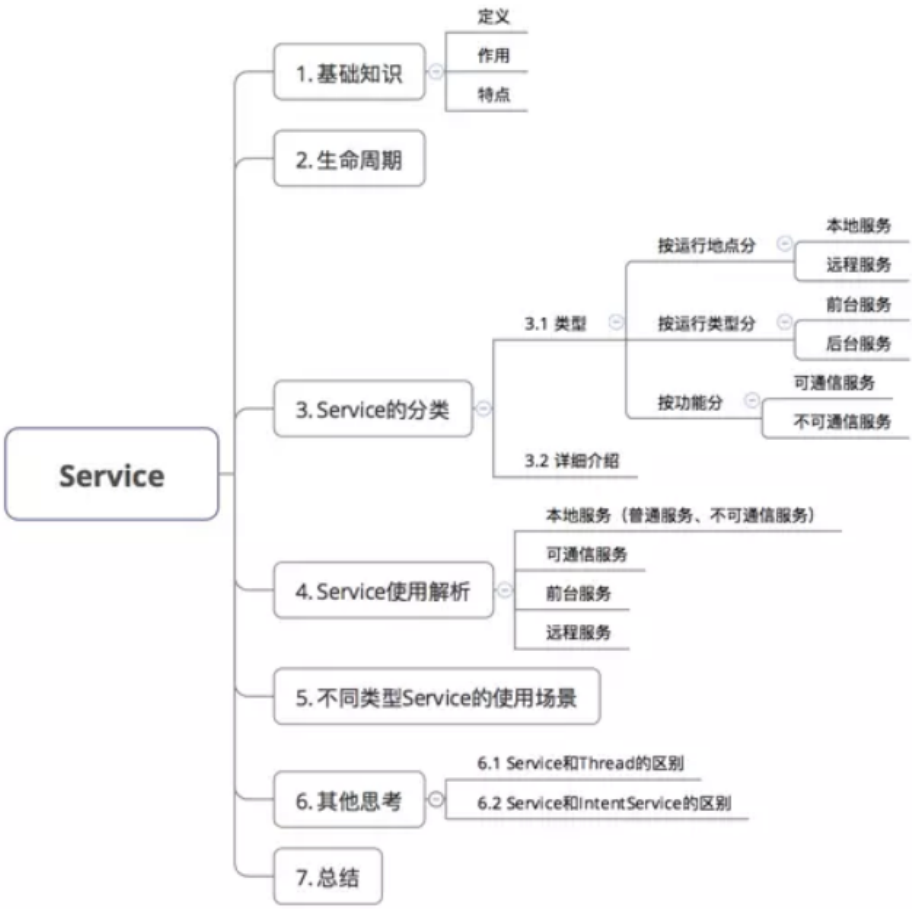
2 Android 四大组件：一份全面 & 简洁的 Service 知识讲解攻略

- https://www.jianshu.com/p/d963c55c3ab9

2.1 前言

Service 作为 Android 四大组件之一，应用非常广泛本文将提供一份全面 & 简洁的 Service 知识讲解攻略，希望你们会喜欢

2.2 目录



2.3 简介

- 定义：服务，是 Android 四大组件之一，属于 计算型组件
- 作用：提供需在后台长期运行的服务
  - 如：复杂计算、音乐播放、下载等
- 特点：无用户界面、在后台运行、生命周期长

2.4 生命周期

- 具体请看前一章文章：Android：Service 生命周期最全面解析
  - https://www.jianshu.com/p/8d0cde35eb10

2.5 类型

- Service 可按照运行地点、运行类型 & 功能进行分类，具体如下

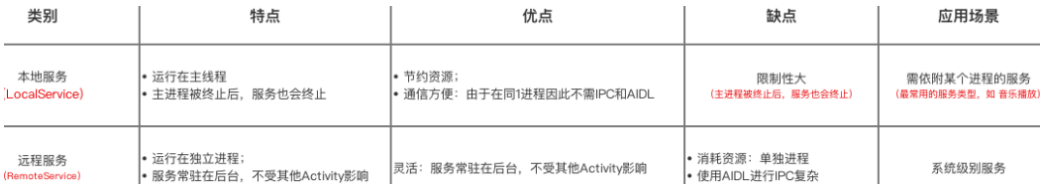
2.5.1 具体分类

- 按运行地点分类
  - 本地服务
  - 远程服务
- 按运行类型分类
  - 前台服务
  - 后台服务
- 按功能分类
  - 可通信服务
  - 不可通信服务

2.5.2 详细介绍

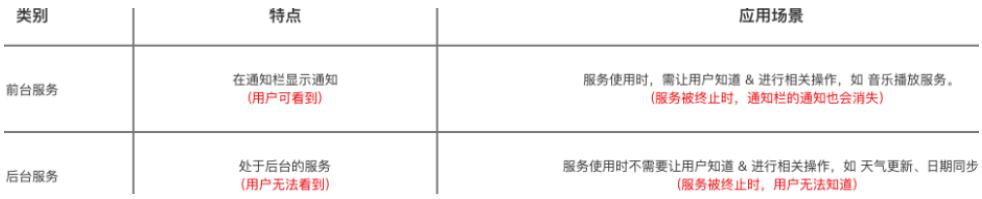
- 按运行地点分类

类型	特点	优点	缺点
本地	-运行在主线程 -主进程被禁止后，服务也会终止	-节约资源 -通信方便： 因在同一进程因此不需 IPC 和 AIDL	-限制性大 主进程被禁止后， 服务也会终止
远程	-运行在独立进程 -服务常驻在后台， 不受其它 activity 影响	-灵活： 服务常驻在后台， 不受其它 activity 影响	-消耗资源：单独进程 -使用 AIDL 进行 IPC 复



类别	特点	优点	缺点	应用场景
本地服务 (LocalService)	• 运行在主线程 • 主进程被终止后，服务也会终止	• 节约资源； • 通信方便：由于在同1进程因此不需IPC和AIDL	限制性大 (主进程被终止后，服务也会终止)	需依附某个进程的服务 (最常用的服务类型，如 音乐播放)
远程服务 (RemoteService)	• 运行在独立进程； • 服务常驻在后台，不受其他Activity影响	灵活：服务常驻在后台，不受其他Activity影响	• 消耗资源：单独进程 • 使用AIDL进行IPC复杂	系统级别服务

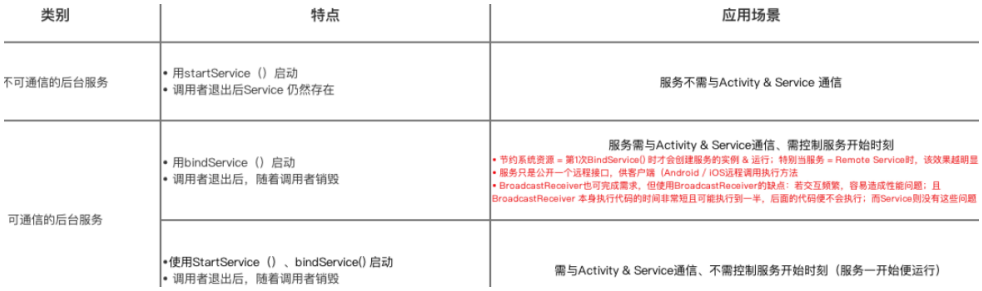
- 按运行类型分类



类别	特点	应用场景
前台服务	在通知栏显示通知 (用户可看到)	服务使用时，需让用户知道 & 进行相关操作，如 音乐播放服务。 (服务被终止时，通知栏的通知也会消失)
后台服务	处于后台的服务 (用户无法看到)	服务使用时不需要让用户知道 & 进行相关操作，如 天气更新、日期同步 (服务被终止时，用户无法知道)

类型	特点	应用场景
前台服务	- 在通知栏显示通知 （用户可看到）	服务使用时需让用户知道并进行相关操作，如音乐播放 （服务被终止时，通知栏的通知也会消失）
后台服务	- 处于后台的服务 （用户无法看到）	服务使用时不需让用户知道并进行相关操作，如天气更新、日期同步 （服务被终止时，用户无法知道）

- 按功能分类



类型	特点	应用场景
不可通信服务	-用 startService() 启动 -调用者退出后 service 仍然存在	服务不需与 Activity & Service 通信
可通信服务	-用 bindService() 启动 -调用者退出后，随着调用者销毁	服务需与 Activity & Service 通信 需控制服务开启时刻 * 备注
可通信服务	-使用 startService()、bindService() 启动 -调用者退出后，随着调用者销毁	需与 Activity & Service 通信 不需控制服务开启时刻 服务一开始便运行

- 备注
  - 节约系统资源 = 第一次 bindService() 时才会创建服务的实例 & 运行；特别当服务 =remote service 时，该效果越明显
  - 服务只是公开一个远程接口，供客户端 Android/iOS 远程调用执行方法
  - BroadcastReceiver 也可完成需求，但使用 BroadcastReceiver 的缺点：若交互频繁，容易造成性能问题；且 BroadcastReceiver 本身执行代码的时间非常短且可能执行到一半，后面的代码便不会执行；而 Service 则没有这些问题

2.6 使用讲解

- 下面，我将介绍每种 Service 的具体使用
- 具体请看文章：Android：（本地、可通信的、前台、远程）Service 使用全面介绍
  - https://www.jianshu.com/p/e04c4239b07e

2.7 其他思考

2.7.1 Service 与 Thread 的区别

- 结论：Service 与 Thread 无任何关系
- 之所以有不少人会把它们联系起来，主要因为 Service 的后台概念
  - 后台：后台任务运行完全不依赖 UI，即使 Activity 被销毁 / 程序被关闭，只要进程还在，后台任务就可继续运行

- 关于二者的异同，具体如下图：

类型	相同点	不同点	
	作用	运行线程	运行范围
Service	执行 异步 操作	主线程 (不能处理耗时操作，否则会出现ANR)	进程 • 完全不依赖UI / Activity，只要进程还在，Service就可继续运行 • 所有Activity都可 与 Service关联 获得Binder实例 & 操作其中的方法 • 若要处理耗时操作，则在Service里创建Thread子线程执行
Thread		工作线程	Activity • 即 依赖于某个Activity • 在一个Activity中创建的子线程，另一个Activity无法对其进行操作； • Activity 很难控制 Thread • 当Activity被销毁后，就无法再获取到之前创建的子线程实例

不同点

运行线程	运行范围
主线程 (不能处理耗时操作，否则会出现ANR)	进程 • 完全不依赖UI / Activity，只要进程还在，Service就可继续运行 • 所有Activity都可 与 Service关联 获得Binder实例 & 操作其中的方法 • 若要处理耗时操作，则在Service里创建Thread子线程执行
工作线程	Activity • 即 依赖于某个Activity • 在一个Activity中创建的子线程，另一个Activity无法对其进行操作； • Activity 很难控制 Thread • 当Activity被销毁后，就无法再获取到之前创建的子线程实例

- 注：一般会将 Service 和 Thread 联合着用，即在 Service 中再创建一个子线程（工作线程）去处理耗时操作逻辑，如下代码：

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // 新建工作线程
    new Thread(new Runnable() {
        @Override
        public void run() {
            // 开始执行后台任务
        }
    }).start();
    return super.onStartCommand(intent, flags, startId);
}
class MyBinder extends Binder {
    public void service_connect_Activity() {
        // 新建工作线程
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 执行具体的下载任务
            }
        }).start();
    }
}
```

2.7.2 Service 和 IntentService 的区别

具体请看文章：Android 多线程：IntentService 用法 & 源码

- https://www.jianshu.com/p/8a3c44a9173a Android 多线程解析：IntentService（含源码解析）

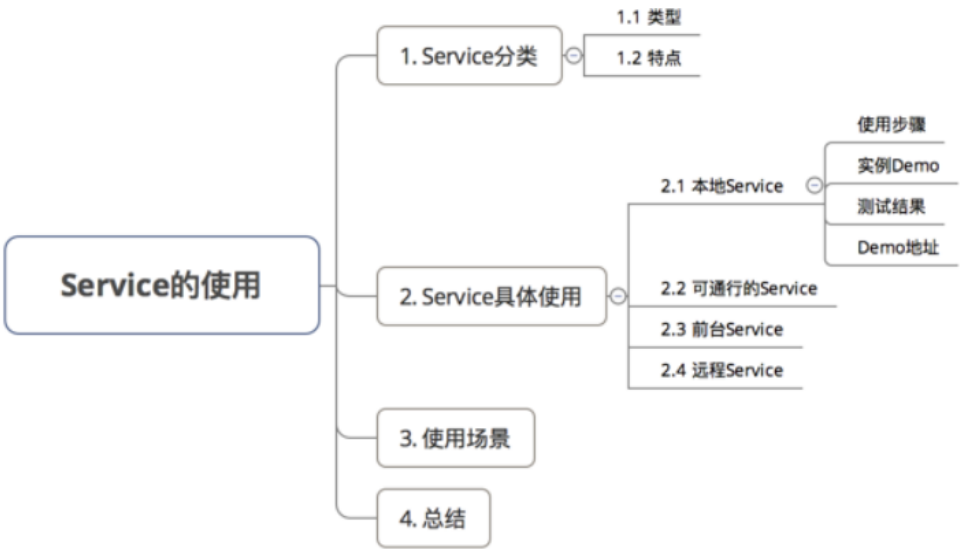
3 Android:（本地、可通信的、前台、远程）Service 使用全面介绍

- https://www.jianshu.com/p/e04c4239b07e

3.1 前言

Service 作为 Android 四大组件之一，应用非常广泛本文将介绍 Service 最基础的知识：Service 的生命周期如果你对 Service 还未了解，建议先阅读我写的文章：Android 四大组件：Service 史上最全面解析

3.2 目录



3.3 Service 分类

3.3.1 Service 的类型

- 按运行地点分类
  - 本地服务
  - 远程服务
- 按运行类型分类
  - 前台服务
  - 后台服务
- 按功能分类
  - 可通信服务
  - 不可通信服务

3.3.2 特点

- 详见前一章的三个表格

3.4 具体使用解析

3.4.1 本地 Service

- 这是最普通、最常用的后台服务 Service。

1. 使用步骤
   - 步骤 1：新建子类继承 Service 类
     - 需重写父类的 onCreate()、onStartCommand()、onDestroy() 和 onBind() 方法
   - 步骤 2：构建用于启动 Service 的 Intent 对象
   - 步骤 3：调用 startService() 启动 Service、调用 stopService() 停止服务
   - 步骤 4：在 AndroidManifest.xml 里注册 Service
2. 实例 Demo 接下来我将用一个实例 Demo 进行本地 Service 说明
   - 建议先下载 Demo 再进行阅读：（carson.ho 的 Github 地址）Demo_for_Service
     - https://github.com/Carson-Ho/Demo_Service/tree/5e2a70cf2d75c56bbfa1abc0ead
   - 步骤 1：新建子类继承 Service 类
     - 需重写父类的 onCreate()、onStartCommand()、onDestroy() 和 onBind()
     - MyService.java

```
public class MyService extends Service {
    // 启动 Service 之后，
    // 就可以在 onCreate() 或 onStartCommand() 方法里去执行一些具体的逻辑
    // 由于这里作 Demo 用，所以只打印一些语句
    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println(" 执行了 onCreat()");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        System.out.println(" 执行了 onStartCommand()");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        System.out.println(" 执行了 onDestory()");
    }
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

   - 步骤 2：在主布局文件设置两个 Button 分别用于启动和停止 Service
     - activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.demo_service.MainActivity">
    <Button
        android:layout_centerInParent="true"
```

```
        android:id="@+id/startService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 启动服务" />
    <Button
        android:layout_centerInParent="true"
        android:layout_below="@+id/startService"
        android:id="@+id/stopService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 停止服务" />
</RelativeLayout>
```

- 步骤 3：构建 Intent 对象，并调用 startService() 启动 Service、stopService 停止服务
  - MainActivity.java

```
public class MainActivity extends AppCompatActivity
    implements View.OnClickListener {

    private Button startService;
    private Button stopService;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startService = (Button) findViewById(R.id.startService);
        stopService = (Button) findViewById(R.id.stopService);
        startService.setOnClickListener(this);
        startService.setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            // 点击启动 Service Button
            case R.id.startService:
                // 构建启动服务的 Intent 对象
                Intent startIntent = new Intent(this, MyService.class);
                // 调用 startService() 方法-传入 Intent 对象, 以此启动服务
                startService(startIntent);
            // 点击停止 Service Button
            case R.id.stopService:
                // 构建停止服务的 Intent 对象
                Intent stopIntent = new Intent(this, MyService.class);
                // 调用 stopService() 方法-传入 Intent 对象, 以此停止服务
                stopService(stopIntent);
        }
    }
}
```

- 步骤 4：在 AndroidManifest.xml 里注册 Service
  - AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="scut.carson_ho.demo_service">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        //注册 Service 服务
        <service android:name=".MyService">
        </service>
    </application>
</manifest>
```

- Androidmanifest 里 Service 的常见属性说明

属性	说明	备注
android:name android:label android:icon	Service 的类名 Service 的名字 Service 的图标	若不设置，默认为 Servic
android:permission	申明此 Service 的权限	有提供了该权限的应用才 或连接此服务
android:process	表示该服务是否在另一个进程中运行（远程服务)	不设置默认为本地服务； remote 则设置成远程服务
android:enabled	系统默认启动	true：Service 将会默认被 不设置则默认为 false
android:exported	该服务是否能够被其他应用程序所控制或连接	不设置默认此项为 false

3. 测试结果

4. Demo 地址

- Carson.ho 的 Github 地址：Demo_for_Service
  - https://github.com/Carson-Ho/Demo_Service/tree/5e2a70cf2d75c56bbfa1abc0ead

3.4.2 可通信的服务 Service

- 上面介绍的 Service 是最基础的，但只能单机使用，即无法与 Activity 通信
- 接下来将在上面的基础用法上，增设" 与 Activity 通信”的功能，即使用绑定 Service 服务（Binder 类、bindService()、onBind()、unbindService()、onUnbind()）

1. 实例 Demo 接下来我将用一个实例 Demo 进行可通信的服务 Service 说明

- 建议先下载 Demo 再进行阅读：（carson.ho 的 Github 地址）Demo_for_Service
  - https://github.com/Carson-Ho/Demo_Service/tree/719e3b9ffd5017c334cdfdaf45b
- 步骤 1：在新建子类继承 Service 类，并新建一个子类继承自 Binder 类、写入与 Activity 关联需要的方法、创建实例

```
public class MyService extends Service {
    private MyBinder mBinder = new MyBinder();
    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println(" 执行了 onCreat()");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        System.out.println(" 执行了 onStartCommand()");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        System.out.println(" 执行了 onDestory()");
    }
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
```

```
        System.out.println(" 执行了 onBind()");
        //返回实例
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        System.out.println(" 执行了 onUnbind()");
        return super.onUnbind(intent);
    }
    //新建一个子类继承自 Binder 类
    class MyBinder extends Binder {
        public void service_connect_Activity() {
            System.out.println("Service 关联了 Activity, 并在 Activity 执行了 Service 的方法");
        }
    }
}
```

- 步骤 2：在主布局文件再设置两个 Button 分别用于绑定和解绑 Service

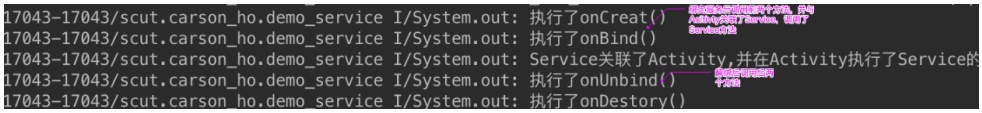
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.demo_service.MainActivity">
    <Button
        android:layout_centerInParent="true"
        android:id="@+id/startService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 启动服务" />
    <Button
        android:layout_centerInParent="true"
        android:layout_below="@+id/startService"
        android:id="@+id/stopService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 停止服务" />
    <Button
        android:layout_centerInParent="true"
        android:layout_below="@id/stopService"
        android:id="@+id/bindService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 绑定服务" />
    <Button
        android:layout_centerInParent="true"
        android:layout_below="@id/bindService"
        android:id="@+id/unbindService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=" 解绑服务"
        />
</RelativeLayout>
```

- 步骤 3：在 Activity 通过调用 MyBinder 类中的 public 方法来实现 Activity 与 Service 的联系
  - 即实现了 Activity 指挥 Service 干什么 Service 就去干什么的功能
  - MainActivity.java

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private Button startService;
    private Button stopService;
    private Button bindService;
    private Button unbindService;
    private MyService.MyBinder myBinder;
```

```
    // 创建 ServiceConnection 的匿名类
    private ServiceConnection connection = new ServiceConnection() {
            // 重写 onServiceConnected() 方法和 onServiceDisconnected() 方法
            // 在 Activity 与 Service 建立关联和解除关联的时候调用
            @Override
            public void onServiceDisconnected(ComponentName name) {
            }
            // 在 Activity 与 Service 解除关联的时候调用
            @Override
            public void onServiceConnected(ComponentName name, IBinder service) {
                // 实例化 Service 的内部类 myBinder
                // 通过向下转型得到了 MyBinder 的实例
                myBinder = (MyService.MyBinder) service;
                // 在 Activity 调用 Service 类的方法
                myBinder.service_connect_Activity();
            }
        };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startService = (Button) findViewById(R.id.startService);
        stopService = (Button) findViewById(R.id.stopService);
        startService.setOnClickListener(this);
        stopService.setOnClickListener(this);
        bindService = (Button) findViewById(R.id.bindService);
        unbindService = (Button) findViewById(R.id.unbindService);
        bindService.setOnClickListener(this);
        unbindService.setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            // 点击启动 Service
        case R.id.startService:
            // 构建启动服务的 Intent 对象
            Intent startIntent = new Intent(this, MyService.class);
            // 调用 startService() 方法-传入 Intent 对象，以此启动服务
            startService(startIntent);
            break;
            // 点击停止 Service
        case R.id.stopService:
            // 构建停止服务的 Intent 对象
            Intent stopIntent = new Intent(this, MyService.class);
            // 调用 stopService() 方法-传入 Intent 对象，以此停止服务
            stopService(stopIntent);
            break;
            // 点击绑定 Service
        case R.id.bindService:
            // 构建绑定服务的 Intent 对象
            Intent bindIntent = new Intent(this, MyService.class);
            // 调用 bindService() 方法，以此停止服务
            bindService(bindIntent,connection,BIND_AUTO_CREATE);
            // 参数说明
            // 第一个参数:Intent 对象
            // 第二个参数: 上面创建的 Serviceconnection 实例
            // 第三个参数: 标志位
            // 这里传入 BIND_AUTO_CREATE 表示在 Activity 和 Service 建立关联后自动创建 Service
            // 这会使得 MyService 中的 onCreate() 方法得到执行，但 onStartCommand() 方法不会执行
            break;
            // 点击解绑 Service
        case R.id.unbindService:
            // 调用 unbindService() 解绑服务
            // 参数是上面创建的 Serviceconnection 实例
            unbindService(connection);
            break;
        default:
            break;
        }
    }
}
```

2. 测试结果



3. Demo

   - carson.ho 的 Github 地址：Demo_for_Service
     - https://github.com/Carson-Ho/Demo_Service/tree/719e3b9ffd5017c334cdfdaf45b

3.4.3 前台 Service

- 前台 Service 和后台 Service（普通）最大的区别就在于：
  - 前台 Service 在下拉通知栏有显示通知，但后台 Service 没有；
  - 前台 Service 优先级较高，不会由于系统内存不足而被回收；后台 Service 优先级较低，当系统出现内存不足情况时，很有可能会被回收

1. 具体使用

   - 用法很简单，只需要在原有的 Service 类对 onCreate() 方法进行稍微修改即可，如下图：

```
@Override
public void onCreate() {
    super.onCreate();
    System.out.println(" 执行了 onCreat()");

    // 添加下列代码将后台 Service 变成前台 Service
    // 构建" 点击通知后打开 MainActivity" 的 Intent 对象
    Intent notificationIntent = new Intent(this,MainActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(this,0,notificationIntent,0);
    // 新建 Builer 对象
    Notification.Builder builer = new Notification.Builder(this);
    builer.setContentTitle(" 前台服务通知的标题");// 设置通知的标题
    builer.setContentText(" 前台服务通知的内容"); // 设置通知的内容
    builer.setSmallIcon(R.mipmap.ic_launcher); // 设置通知的图标
    builer.setContentIntent(pendingIntent);    // 设置点击通知后的操作
    Notification notification = builer.getNotification();// 将 Builder 对象转变成普通的 notification
    startForeground(1, notification);// 让 Service 变成前台 Service, 并在系统的状态栏显示出来
}
```

2. 测试结果

   - 运行后，当点击 Start Service 或 Bind Service 按钮，Service 就会以前台 Service 的模式启动（通知栏上有通知），如下图

3.4.4 远程 **Service**

- 具体请看我写的另外一篇文章：Android：远程服务 Service（含 AIDL & IPC 讲解）
  - https://www.jianshu.com/p/34326751b2c6

3.5 使用场景

- 通过上述描述，你应该对 Service 类型及其使用非常了解；
- 那么，我们该什么时候用哪种类型的 Service 呢？
- 各种 Service 的使用场景请看下图：

类别	应用场景	备注
本地服务	需要依附某个进程的服务，如音乐播放	最常用、最普通
远程服务	系统级别服务	
前台服务	服务使用时需要让用户知道并进行相关操作，如音乐播放服务。	服务被终止的时候，通知栏的通知也会消失
后台服务	服务使用时不需要让用户知道并进行相关操作，如天气更新，日期同步	
不可通信的后台服务	启动在后台长期运行的服务、不需要与Activity & Service进行通信	使用StartService（）启动
可通信的后台服务	启动在后台长期运行的服务、需要与Activity & Service进行通信、需要控制服务开始时刻	1. 使用BindService()启动 2. 第一次BindService() 时才会创建服务的实例并运行它，节约很多系统资源，特别是当服务是Remote Service时，该效果会越明显 3. 服务只是公开一个远程接口，供客户端（Android / iOS远程调用执行方法。4. BroadcastReceiver也可以完成需求，但使用BroadcastReceiver的缺点：若交互频繁，容易造成性能问题；且BroadcastReceiver 本身执行代码的时间非常短且可能执行到一半，后面的代码便不会执行；而Service则没有这些问题
	启动在后台长期运行的服务、需要与Activity & Service进行通信、不需要控制服务开始时刻（服务一开始便运行）	使用BindService() &StartService（）启动

类型	特点	优点	缺点
本地	-运行在主线程 -主进程被禁止后，服务也会终止	-节约资源 -通信方便： 因在同一进程因此不需 IPC 和 AIDL	-限制性大 主进程被禁止后， 服务也会终止
远程	-运行在独立进程 -服务常驻在后台， 不受其它 activity 影响	-灵活： 服务常驻在后台， 不受其它 activity 影响	-消耗资源：单独进程 -使用 AIDL 进行 IPC 复

类别	特点	优点	缺点	应用场景
本地服务 (LocalService)	• 运行在主线程 • 主进程被终止后，服务也会终止	• 节约资源； • 通信方便：由于在同1进程因此不需IPC和AIDL	限制性大 (主进程被终止后，服务也会终止)	需依附某个进程的服务 (最常用的服务类型，如 音乐播放)
远程服务 (RemoteService)	• 运行在独立进程； • 服务常驻在后台，不受其他Activity影响	灵活：服务常驻在后台，不受其他Activity影响	• 消耗资源：单独进程 • 使用AIDL进行IPC复杂	系统级别服务

- 按运行类型分类

类别	特点	应用场景
前台服务	在通知栏显示通知 (用户可看到)	服务使用时，需让用户知道 & 进行相关操作，如 音乐播放服务。 (服务被终止时，通知栏的通知也会消失)
后台服务	处于后台的服务 (用户无法看到)	服务使用时不需要让用户知道 & 进行相关操作，如 天气更新、日期同步 (服务被终止时，用户无法知道)

类型	应用场景	备注
本地服务 远程服务	需依附某个进程的服务，如音乐播放 系统级别服务	最常用、最普通
前台服务 后台服务	服务使用时需让用户知道并进行相关操作，如音乐播放 服务使用时不需让用户知道并进行相关操作，如天气更新、日期同步	服务被终止时，通 服务被终止时，用
不可通信服务	启动在后台长期运行的服务， 不需与 Activity & Service 通信	使用 startService
可通信服务	启动在后台长期运行的服务， 需与 Activity & Service 通信 需控制服务开启时刻	用 bindService() * 备注
可通信服务	启动在后台长期运行的服务， 需与 Activity & Service 通信 不需控制服务开启时刻	使用 startService

- 备注
  - 用 bindService() 启动
  - 第一次 bindService() 时才会创建服务的实例并运行它，节约很多系统资源，特别是当服务是 Remote Service 时，该效果会更明显
  - 服务只是公开一个远程接口，供客户端 Android、iOS 远程调用执行方法
  - BroadcastReceiver 也可完成需求，但使用 BroadcastReceiver 的缺点：若交互频繁，容易造成性能问题；且 BroadcastReceiver 本身执行代码的时间非常短且可能执行到一半，后面的代码便不会执行，而 Service 则没有这些问题

4 Android 多线程解析：IntentService（含源码解析）

- https://www.jianshu.com/p/8a3c44a9173a

4.1 前言

- 多线程的应用在 Android 开发中是非常常见的，常用方法主要有：
  - 继承 Thread 类
  - 实现 Runnable 接口
  - AsyncTask
  - Handler
  - HandlerThread
  - IntentService
- 今天，我将全面解析多线程其中一种常见用法：IntentService

4.2 定义

- Android 里的一个封装类，继承四大组件之一的 Service

4.3 作用

- 处理异步请求 & 实现多线程

4.4 使用场景

- 线程任务需按顺序、在后台执行
  - 最常见的场景：离线下载
  - 不符合多个数据同时请求的场景：所有的任务都在同一个 Thread looper 里执行

4.5 使用步骤

- 步骤 1：定义 IntentService 的子类
  - 需传入线程名称、复写 onHandleIntent() 方法
- 步骤 2：在 Manifest.xml 中注册服务
- 步骤 3：在 Activity 中开启 Service 服务

4.6 实例应用

- 步骤 1：定义 IntentService 的子类
  - 传入线程名称、复写 onHandleIntent() 方法

```
public class myIntentService extends IntentService {
    // 在构造函数中传入线程名字
    public myIntentService() {
        // 调用父类的构造函数
        // 参数 = 工作线程的名字
        super("myIntentService");
    }
   /**
     * 复写 onHandleIntent() 方法
     * 根据 Intent 实现 耗时任务 操作
     **/
    @Override
```

```
    protected void onHandleIntent(Intent intent) {
        // 根据 Intent 的不同，进行不同的事务处理
        String taskName = intent.getExtras().getString("taskName");
        switch (taskName) {
            case "task1":
                Log.i("myIntentService", "do task1");
                break;
            case "task2":
                Log.i("myIntentService", "do task2");
                break;
            default:
                break;
        }
    }
    @Override
    public void onCreate() {
        Log.i("myIntentService", "onCreate");
        super.onCreate();
    }
   /**
     * 复写 onStartCommand() 方法
     * 默认实现 = 将请求的 Intent 添加到工作队列里
     **/
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i("myIntentService", "onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public void onDestroy() {
        Log.i("myIntentService", "onDestroy");
        super.onDestroy();
    }
}
```

- 步骤 2：在 Manifest.xml 中注册服务

```
<service android:name=".myIntentService">
            <intent-filter >
                <action android:name="cn.scu.finch"/>
            </intent-filter>
        </service>
```

- 步骤 3：在 Activity 中开启 Service 服务

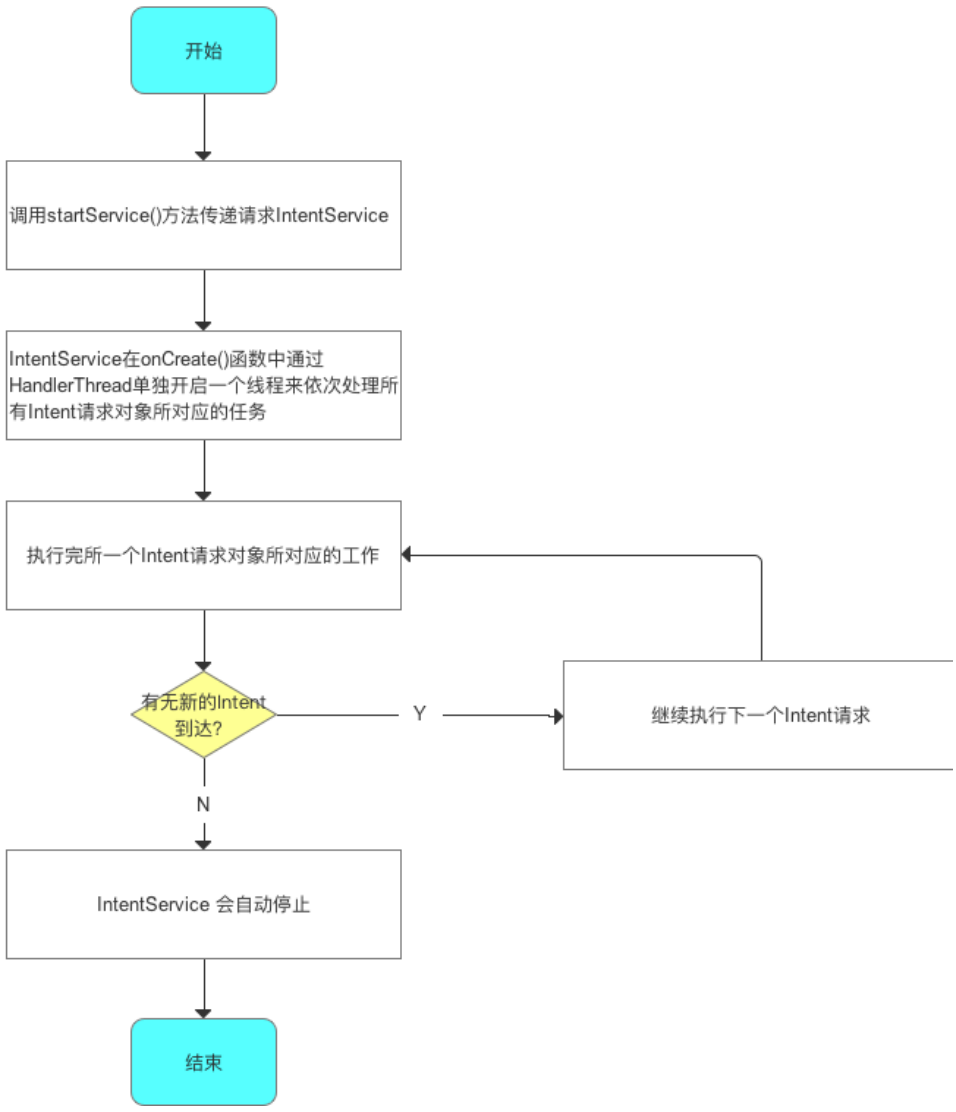
```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 同一服务只会开启 1 个工作线程
        // 在 onHandleIntent () 函数里，依次处理传入的 Intent 请求
        // 将请求通过 Bundle 对象传入到 Intent，再传入到服务里
        // 请求 1
        Intent i = new Intent("cn.scu.finch");
        Bundle bundle = new Bundle();
        bundle.putString("taskName", "task1");
        i.putExtras(bundle);
        startService(i);
        // 请求 2
        Intent i2 = new Intent("cn.scu.finch");
        Bundle bundle2 = new Bundle();
        bundle2.putString("taskName", "task2");
        i2.putExtras(bundle2);
        startService(i2);
        startService(i);  //多次启动
    }
}
```

- 测试结果

```
Tag Text
myIntentService onCreate
myIntentService onStartCommand
myIntentService onStartCommand
myIntentService do task1
myIntentService onStartCommand
myIntentService do task2
myIntentService do task1
myIntentService onDestory
```

4.7 源码分析

- IntentService 的源码工作流程如下：



- 特别注意：若启动 IntentService 多次，那么每个耗时操作则以队列的方式在 IntentService 的 onHandleIntent 回调方法中依次执行，执行完自动结束
- 接下来，我们将通过源码分析解决以下问题：
  - IntentService 如何单独开启 1 个新的工作线程

- IntentService 如何通过 onStartCommand() 将 Intent 传递给服务 & 依次插入到工作队列中

4.7.1 IntentService 如何单独开启 1 个新的工作线程

- 主要分析内容 = IntentService 源码中的 onCreate() 方法

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

    // 1. 通过实例化 andlerThread 新建线程 & 启动；故 使用 IntentService 时，不需额外新建线程
    // HandlerThread 继承自 Thread，内部封装了 Looper
    HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
    thread.start();

    // 2. 获得工作线程的 Looper & 维护自己的工作队列
    mServiceLooper = thread.getLooper();
    // 3. 新建 mServiceHandler & 绑定上述获得 Looper
    // 新建的 Handler 属于工作线程 ->> 分析 1
    mServiceHandler = new ServiceHandler(mServiceLooper);
}
/**
 * 分析 1：ServiceHandler 源码分析
 **/
 private final class ServiceHandler extends Handler {
     // 构造函数
     public ServiceHandler(Looper looper) {
     super(looper);
   }
    // IntentService 的 handleMessage () 把接收的消息交给 onHandleIntent() 处理
    @Override
     public void handleMessage(Message msg) {

      // onHandleIntent 方法在工作线程中执行
      // onHandleIntent() = 抽象方法，使用时需重写 ->> 分析 2
      onHandleIntent((Intent)msg.obj);
      // 执行完调用 stopSelf() 结束服务
      stopSelf(msg.arg1);
    }
}
/**
 * 分析 2： onHandleIntent() 源码分析
 * onHandleIntent() = 抽象方法，使用时需重写
 **/
@WorkerThread
protected abstract void onHandleIntent(Intent intent);
```

4.7.2 IntentService 如何通过 onStartCommand() 将 Intent 传递给服务 & 依次插入到工作队列中

```
/**
 * onStartCommand () 源码分析
 * onHandleIntent() = 抽象方法，使用时需重写
 **/
public int onStartCommand(Intent intent, int flags, int startId) {
    // 调用 onStart () ->> 分析 1
    onStart(intent, startId);
    return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
}
/**
  * 分析 1：onStart(intent, startId)
  **/
  public void onStart(Intent intent, int startId) {
    // 1. 获得 ServiceHandler 消息的引用
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    // 2. 把 Intent 参数 包装到 message 的 obj 发送消息中，
    //这里的 Intent  = 启动服务时 startService(Intent) 里传入的 Intent
```

```
    msg.obj = intent;
    // 3. 发送消息，即 添加到消息队列里
    mServiceHandler.sendMessage(msg);
}
```

4.8 总结

从上面源码可看出：IntentService 本质 = Handler + HandlerThread：

- 通过 HandlerThread 单独开启 1 个工作线程：IntentService
- 创建 1 个内部 Handler ： ServiceHandler
- 绑定 ServiceHandler 与 IntentService
- 通过 onStartCommand() 传递服务 intent 到 ServiceHandler、依次插入 Intent 到工作队列中 & 逐个发送给 onHandleIntent()
- 通过 onHandleIntent() 依次处理所有 Intent 对象所对应的任务
  - 因此我们通过复写 onHandleIntent() & 在里面根据 Intent 的不同进行不同线程操作即可

4.9 注意事项

1. 工作任务队列 = 顺序执行

   即若一个任务正在 IntentService 中执行，此时你再发送 1 个新的任务请求，这个新的任务会一直等待直到前面一个任务执行完毕后才开始执行

   - 原因：
     - 由于 onCreate() 只会调用一次 = 只会创建 1 个工作线程；
     - 当多次调用 startService(Intent) 时（即 onStartCommand （）也会调用多次），其实不会创建新的工作线程，只是把消息加入消息队列中 & 等待执行。
     - 所以，多次启动 IntentService 会按顺序执行事件

   若服务停止，则会清除消息队列中的消息，后续的事件不执行

2. 不建议通过 bindService() 启动 IntentService

   原因：

   ```
   // 在 IntentService 中，onBind()`默认返回 null
   @Override
   public IBinder onBind(Intent intent) {
       return null;
   }
   ```

   - 采用 bindService() 启动 IntentService 的生命周期如下：

     ```
     onCreate() -->> onBind() -->> onUnbind()-->> onDestory()
     ```

   - 即，并不会调用 onStart() 或 onStartcommand()，故不会将消息发送到消息队列，那么 onHandleIntent() 将不会回调，即无法实现多线程的操作
     - 此时，你应该使用 Service，而不是 IntentService

4.10 对比

4.10.1 IntentService 与 Service 的区别

类型	运行线程	结束服务操作
Service	主线程 不能处理耗时操作，否则会出现 ANR	需主动调用 stopService()
IntentService	创建一个工作线程处理多线程任务	不需要 在所有 Intent 被处理完后系统会自动关闭服务

- 备注：
  - IntentService 为 Service 的 onBind() 提供了默认实现：返回 null
  - IntentService 为 Service 的 onStartCommand() 提供了默认实现：将请求的 intent 添加到队列中

4.10.2 IntentService 与其他线程的区别

类型	不同点		
	线程属性	作用	线程优先级
IntentService	类似 后台线程 (内部采用了HandlerThread实现)	后台服务 (继承了 Service)	高 (不容易被系统杀死)
其他线程	普通线程	普通多线程作用	低 (若进程中没有活动的四大组件，则该线程的优先级非常低，容易被系统杀死)

4.11 总结

- 本文主要全面介绍了多线程 IntentService 用法 & 源码
- 接下来,我会继续讲解 Android 开发中关于多线程的知识,包括继承 Thread 类、实现 Runnable 接口、Handler 等等，有兴趣可以继续关注 Carson_Ho 的安卓开发笔记

5 Android：远程服务 Service（含 AIDL & IPC 讲解）

- https://www.jianshu.com/p/34326751b2c6

5.1 前言

- Service 作为 Android 四大组件之一，应用非常广泛
- 本文将介绍 Service 其中一种常见用法：远程 Service

5.2 远程服务与本地服务的区别

- 远程服务与本地服务最大的区别是：远程 Service 与调用者不在同一个进程里（即远程 Service 是运行在另外一个进程）；而本地服务则是与调用者运行在同一个进程里
- 二者区别的详细区别如下图：

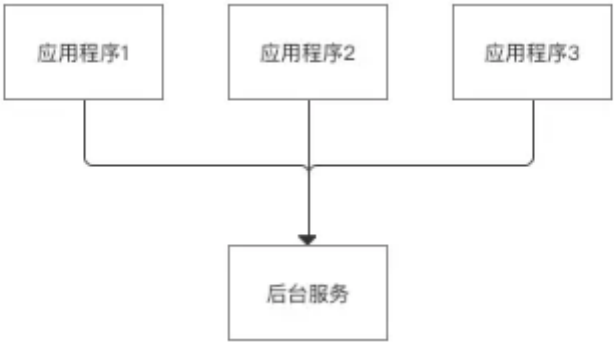
类别	特点	优点	缺点	应用场景
本地服务（LocalService）	1. 运行在主线程 2. 主进程被终止后，服务也会终止。	1. 节约资源； 2. 通信方便：由于在同一进程因此不需要IPC和AIDL。	限制性大：主进程被终止后，服务也会终止。	需要依附某个进程的服务，如音乐播放
远程服务（RemoteService）	1. 运行在独立进程； 2. 服务常驻在后台，不受其他Activity影响	灵活：服务常驻在后台，不受其他Activity影响	1. 消耗资源：单独进程 2. 使用AIDL进行IPC复杂	系统级别服务

类型	特点	优点	缺点	应
本地 Local Service	1 运行在主线程 2 主进程被禁止后，服务也会终止	1 节约资源 2 通信方便：在同一进程 => 不需 IPC 和 AIDL	限制性大: 主进程被禁止后，服务也会终止	需（如
远程 Remote Service	1 运行在独立进程 2 服务常驻在后台，不受其它 activity 影响	灵活： 服务常驻在后台，不受其它 activity 影响	1 消耗资源：单独进程 2 使用 AIDL 进行 IPC 复杂	系

5.3 使用场景

- 多个应用程序共享同一个后台服务（远程服务）
  - 即一个远程 Service 与多个应用程序的组件（四大组件）进行跨进程通信

5.4 使用场景



5.5 具体使用

- 为了让远程 Service 与多个应用程序的组件（四大组件）进行跨进程通信（IPC），需要使用 AIDL
  - **IPC ： Inter-Process Communication** ，即跨进程通信
  - **AIDL ： Android Interface Definition Language** ，即 Android 接口定义语言；
    - 用于让某个 Service 与多个应用程序组件之间进行跨进程通信，从而可以实现多个应用程序共享同一个 Service 的功能。

- 在多进程通信中，存在两个进程角色（以最简单的为例）：服务器端和客户端
- 以下是两个进程角色的具体使用步骤：
  - 服务器端（**Service**）
    - 步骤 1：新建定义 AIDL 文件，并声明该服务需要向客户端提供的接口
    - 步骤 2：在 Service 子类中实现 AIDL 中定义的接口方法，并定义生命周期的方法（onCreate()、onStartCommand()、onBind()、onUnbind()、onDestory()）
    - 步骤 3：在 AndroidMainfest.xml 中注册服务 & 声明为远程服务
  - 客户端（**Client**）
    - 步骤 1：拷贝服务端的 AIDL 文件到目录下
    - 步骤 2：使用 Stub.asInterface 接口获取服务器的 Binder，根据需要调用服务提供的接口方法
    - 步骤 3：通过 Intent 指定服务端的服务名称和所在包，绑定远程 Service
- 接下来，我将用一个具体实例来介绍远程 Service 的使用

5.6 具体实例

- 实例描述：客户端远程调用服务器端的远程 Service
- 具体使用：

5.6.1 服务器端（**Service**）

新建一个服务器端的工程：Service - server

- 先下 Demo 再看，效果会更好：Github_RemoteService_Server
- 步骤 1. 新建一个 AIDL 文件
  - New ==> AIDL ==> AIDL File
- 步骤 2. 在新建 AIDL 文件里定义 Service 需要与 Activity 进行通信的内容（方法），并进行编译（Make Project）

```
// 在新建的 AIDL_Service1.aidl 里声明需要与 Activity 进行通信的方法
package scut.carson_ho.demo_service;
interface AIDL_Service1 {
    void AIDL_Service();
}
//AIDL 中支持以下的数据类型
//1. 基本数据类型
//2. String 和 CharSequence
//3. List 和 Map ,List 和 Map 对象的元素必须是 AIDL 支持的数据类型;
//4. AIDL 自动生成的接口（需要导入-import）
//5. 实现 android.os.Parcelable 接口的类（需要导入-import)
```

编译

- 步骤 3：在 Service 子类中实现 AIDL 中定义的接口方法，并定义生命周期的方法（onCreate()、onBind()、onUnbind() etc）
  - MyService.java

```
/**
 * onStartCommand () 源码分析
 * onHandleIntent() = 抽象方法，使用时需重写
 **/
public int onStartCommand(Intent intent, int flags, int startId) {
    // 调用 onStart () ->> 分析 1
    onStart(intent, startId);
    return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
}
/**
 * 分析 1: onStart(intent, startId)
 **/
public void onStart(Intent intent, int startId) {
    // 1. 获得 ServiceHandler 消息的引用
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    // 2. 把 Intent 参数 包装到 message 的 obj 发送消息中，
    //这里的 Intent  = 启动服务时 startService(Intent) 里传入的 Intent
    msg.obj = intent;
    // 3. 发送消息，即 添加到消息队列里
    mServiceHandler.sendMessage(msg);
}
```

- 步骤 4：在 AndroidMainfest.xml 中注册服务 & 声明为远程服务

```
<service
    android:name=".MyService"
    android:process=":remote"  //将本地服务设置成远程服务
    android:exported="true"      //设置可被其他进程调用
    //该 Service 可以响应带有 scut.carson_ho.service_server.AIDL_Service1 这个 action 的 Intent。
    //此处 Intent 的 action 必须写成" 服务器端包名.aidl 文件名”
    <intent-filter>
        <action android:name="scut.carson_ho.service_server.AIDL_Service1"/>
    </intent-filter>
</service>
```

- 至此，服务器端（远程 Service）已经完成了。

5.6.2 客户端（**Client**）

新建一个客户端的工程：Service - Client

- 先下 Demo 再看，效果会更好：Github_RemoteService_Client
- 步骤 1：将服务端的 AIDL 文件所在的包复制到客户端目录下（Project/app/src/main），并进行编译
  - 注：记得要原封不动地复制!! 什么都不要改!
- 步骤 2：在主布局文件定义" 绑定服务”的按钮
  - MainActivity.xml

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="scut.carson_ho.service_client.MainActivity">
    <Button
        android:layout_centerInParent="true"
```

```
        android:id="@+id/bind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text=" 绑定服务"
        />
</RelativeLayout>
```

- 步骤 3：在 MainActivity.java 里
  - 使用 Stub.asInterface 接口获取服务器的 Binder；
  - 通过 Intent 指定服务端的服务名称和所在包，进行 Service 绑定；
  - 根据需要调用服务提供的接口方法。
  - MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    private Button bindService;
    // 定义 aidl 接口变量
    private AIDL_Service1 mAIDL_Service;
    // 创建 ServiceConnection 的匿名类
    private ServiceConnection connection = new ServiceConnection() {
        // 重写 onServiceConnected() 方法和 onServiceDisconnected() 方法
        // 在 Activity 与 Service 建立关联和解除关联的时候调用
        @Override
        public void onServiceDisconnected(ComponentName name) {
        }
        // 在 Activity 与 Service 建立关联时调用
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            // 使用 AIDLService1.Stub.asInterface() 方法获取服务器端返回的 IBinder 对象
            // 将 IBinder 对象传换成了 mAIDL_Service 接口对象
            mAIDL_Service = AIDL_Service1.Stub.asInterface(service);
            try {
                // 通过该对象调用在 MyAIDLService.aidl 文件中定义的接口方法, 从而实现跨进程通信
                mAIDL_Service.AIDL_Service();
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bindService = (Button) findViewById(R.id.bind_service);
        // 设置绑定服务的按钮
        bindService.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                // 通过 Intent 指定服务端的服务名称和所在包, 与远程 Service 进行绑定
                // 参数与服务器端的 action 要一致, 即" 服务器包名.aidl 接口文件名"
                Intent intent = new Intent("scut.carson_ho.service_server.AIDL_Service1");

                // Android5.0 后无法只通过隐式 Intent 绑定远程 Service
                // 需要通过 setPackage() 方法指定包名
                intent.setPackage("scut.carson_ho.service_server");

                // 绑定服务, 传入 intent 和 ServiceConnection 对象
                bindService(intent, connection, Context.BIND_AUTO_CREATE);
            }
        });
    }
}
```

5.6.3 测试结果

```
1414-1414/scut.carson_ho.service_client I/System.out: 点击了[绑定服务]按钮
1695-1695/scut.carson_ho.service_server:remote I/System.out: 执行了onCreat()
1695-1695/scut.carson_ho.service_server:remote I/System.out: 执行了onBind()
1695-1706/scut.carson_ho.service_server:remote I/System.out: 客户端通过AIDL与远程后台成功通信
```

- 从上面测试结果可以看出：
  - 打印的语句分别运行在不同进程（看语句前面的包名）；
  - 客户端调用了服务端 Service 的方法
- 即客户端和服务端进行了跨进程通信

5.6.4 Demo 地址

- 客户端：Github_RemoteService_Client
- 服务端：Github_RemoteService_Server

6 Android 屏幕适配：最全面的解决方案

- https://www.jianshu.com/p/ec5a1a30694b

6.1 前言

- Android 的屏幕适配一直以来都在折磨着我们 Android 开发者，本文将结合：
  - Google 的官方权威适配文档
  - 郭霖：Android 官方提供的支持不同屏幕大小的全部方法
    - https://blog.csdn.net/guolin_blog/article/details/8830286
  - Stormzhang：Android 屏幕适配
  - 鸿洋：Android 屏幕适配方案
    - https://blog.csdn.net/lmj623565791/article/details/45460089
  - 凯子：Android 屏幕适配全攻略 (最权威的官方适配指导)
    - https://blog.csdn.net/zhaokaiqiang1992/article/details/45419023
  - 自身的思考 & 实践
- 给你带来一种全新、全面而逻辑清晰的 Android 屏幕适配思路，只要你认真阅读，保证你能解决 Android 的屏幕适配问题！

6.2 目录

6.3 定义

- 使得某一元素在 Android 不同尺寸、不同分辨率的手机上具备相同的显示效果

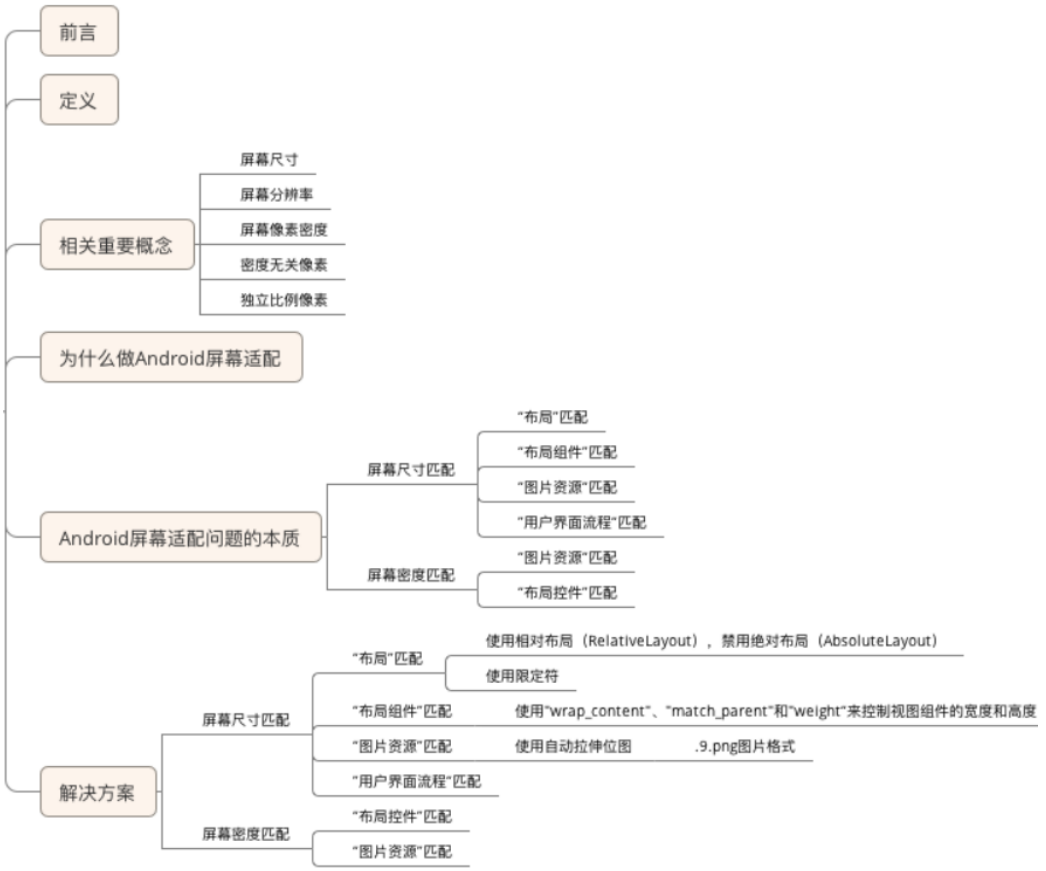


图 1: Android 屏幕适配解决方案

6.4 相关重要概念

6.4.1 屏幕尺寸

- 含义：手机对角线的物理尺寸
- 单位：英寸（inch），1 英寸 =2.54cm
  - Android 手机常见的尺寸有 5 寸、5.5 寸、6 寸等等

6.4.2 屏幕分辨率

- 含义：手机在横向、纵向上的像素点数总和
  - 一般描述成屏幕的" 宽 x 高"=AxB
  - 含义：屏幕在横向方向（宽度）上有 A 个像素点，在纵向方向（高）有 B 个像素点
  - 例子：1080x1920，即宽度方向上有 1080 个像素点，在高度方向上有 1920 个像素点
- 单位：px（pixel），1px=1 像素点
  - UI 设计师的设计图会以 px 作为统一的计量单位
- Android 手机常见的分辨率：320x480、480x800、720x1280、1080x1920

6.4.3 屏幕像素密度

- 含义：每英寸的像素点数
- 单位：dpi（dots per ich）
  - 假设设备内每英寸有 160 个像素，那么该设备的屏幕像素密度 =160dpi
- 安卓手机对于每类手机屏幕大小都有一个相应的屏幕像素密度：

密度类型	代表的分辨率（px）	屏幕像素密度（dpi）
低密度（ldpi）	240x320	120
中密度（mdpi）	320x480	160
高密度（hdpi）	480x800	240
超高密度（xhdpi）	720x1280	320
超超高密度（xxhdpi）	1080x1920	480

1. 屏幕尺寸、分辨率、像素密度三者关系
   - 一部手机的分辨率是宽 x 高，屏幕大小是以寸为单位，那么三者的关系是：

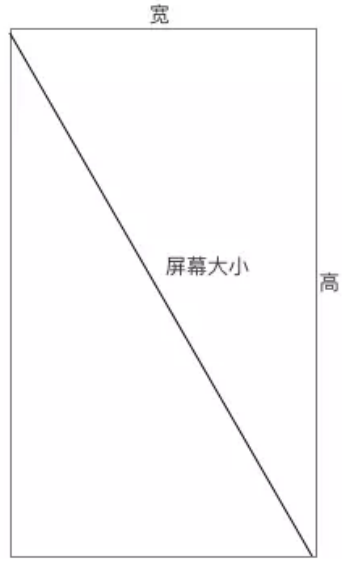
$$\text{密度（单位/dpi）} = \frac{\sqrt{(\text{宽}^2+\text{高}^2)}\ \text{单位（/px）}}{\text{屏幕大小}\quad \text{单位（/inch）}}$$

讲解

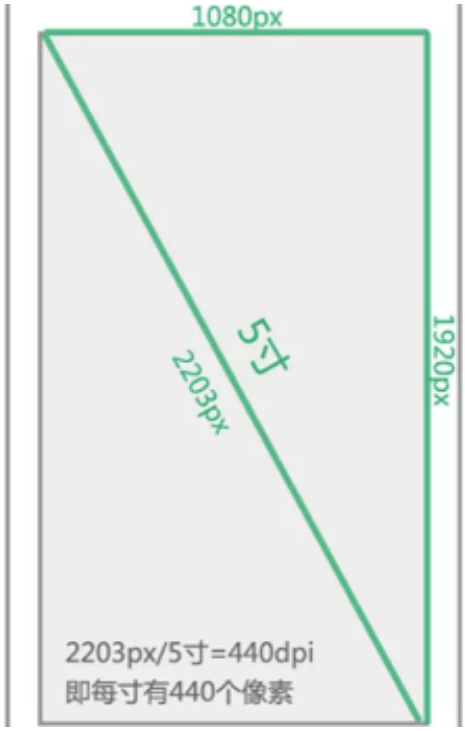
1. 密度即每英寸的像素点

2. 勾股定理求出手机的对角线物理尺寸

3. 再除以屏幕大小即可



   - 假设一部手机的分辨率是 1080x1920（px），屏幕大小是 5 寸，问密度是多少？



6.4.4 密度无关像素

- 含义：density-independent pixel，叫 dp 或 dip，与终端上的实际物理像素点无关。
- 单位：dp，可以保证在不同屏幕像素密度的设备上显示相同的效果
  - Android 开发时用 dp 而不是 px 单位设置图片大小，是 Android 特有的单位
  - 场景：假如同样都是画一条长度是屏幕一半的线，如果使用 px 作为计量单位，那么在 480x800 分辨率手机上设置应为 240px；在 320x480 的手机上应设置为 160px，二者设置就不同了；如果使用 dp 为单位，在这两种分辨率下，160dp 都显示为屏幕一半的长度。
- dp 与 px 的转换
  - 因为 ui 设计师给你的设计图是以 px 为单位的，Android 开发则是使用 dp 作为单位的，那么我们需要进行转换：

密度类型	代表的分辨率（px）	屏幕密度（dpi）	换算（px/dp）	比例
低密度（ldpi）	240x320	120	1dp=0.75px	3
中密度（mdpi）	320x480	160	1dp=1px	4
高密度（hdpi）	480x800	240	1dp=1.5px	6
超高密度（xhdpi）	720x1280	320	1dp=2px	8
超超高密度（xxhdpi）	1080x1920	480	1dp=3px	12

- 在 Android 中，规定以 160dpi（即屏幕分辨率为 320x480）为基准：1dp=1px

6.4.5 独立比例像素

- 含义：scale-independent pixel，叫 sp 或 sip
- 单位：sp

- Android 开发时用此单位设置文字大小，可根据字体大小首选项进行缩放
- 推荐使用 12sp、14sp、18sp、22sp 作为字体设置的大小，不推荐使用奇数和小数，容易造成精度的丢失问题；小于 12sp 的字体会太小导致用户看不清

6.5 为什么要进行 **Android** 屏幕适配

由于 Android 系统的开放性，任何用户、开发者、OEM 厂商、运营商都可以对 Android 进行定制，于是导致：

- Android 系统碎片化：小米定制的 MIUI、魅族定制的 flyme、华为定制的 EMUI 等等
  - 当然都是基于 Google 原生系统定制的
- Android 机型屏幕尺寸碎片化：5 寸、5.5 寸、6 寸等等
- Android 屏幕分辨率碎片化：320x480、480x800、720x1280、1080x1920
  - 据友盟指数显示，统计至 2015 年 12 月，支持 Android 的设备共有 27796 种
- 当 Android 系统、屏幕尺寸、屏幕密度出现碎片化的时候，就很容易出现同一元素在不同手机上显示不同的问题。
  - 试想一下这么一个场景：
  - 为 4.3 寸屏幕准备的 UI 设计图，运行在 5.0 寸的屏幕上，很可能在右侧和下侧存在大量的空白；而 5.0 寸的 UI 设计图运行到 4.3 寸的设备上，很可能显示不下。

为了保证用户获得一致的用户体验效果：

- 使得某一元素在 Android 不同尺寸、不同分辨率的手机上具备相同的显示效果
- 于是，我们便需要对 Android 屏幕进行适配。

6.6 屏幕适配问题的本质

- 使得" 布局"、" 布局组件"、" 图片资源"、" 用户界面流程" 匹配不同的屏幕尺寸
  - 使得布局、布局组件自适应屏幕尺寸；
  - 根据屏幕的配置来加载相应的 UI 布局、用户界面流程
- 使得" 图片资源" 匹配不同的屏幕密度

6.6.1 屏幕尺寸匹配

问题：如何进行屏幕尺寸匹配？

1. " 布局" 匹配
   - 本质 **1**：使得布局元素自适应屏幕尺寸

   (a) 使用相对布局（RelativeLayout），禁用绝对布局（AbsoluteLayout）
   开发中，我们使用的布局一般有：
   - 线性布局（Linearlayout）
   - 相对布局（RelativeLayout）
   - 帧布局（FrameLayout）

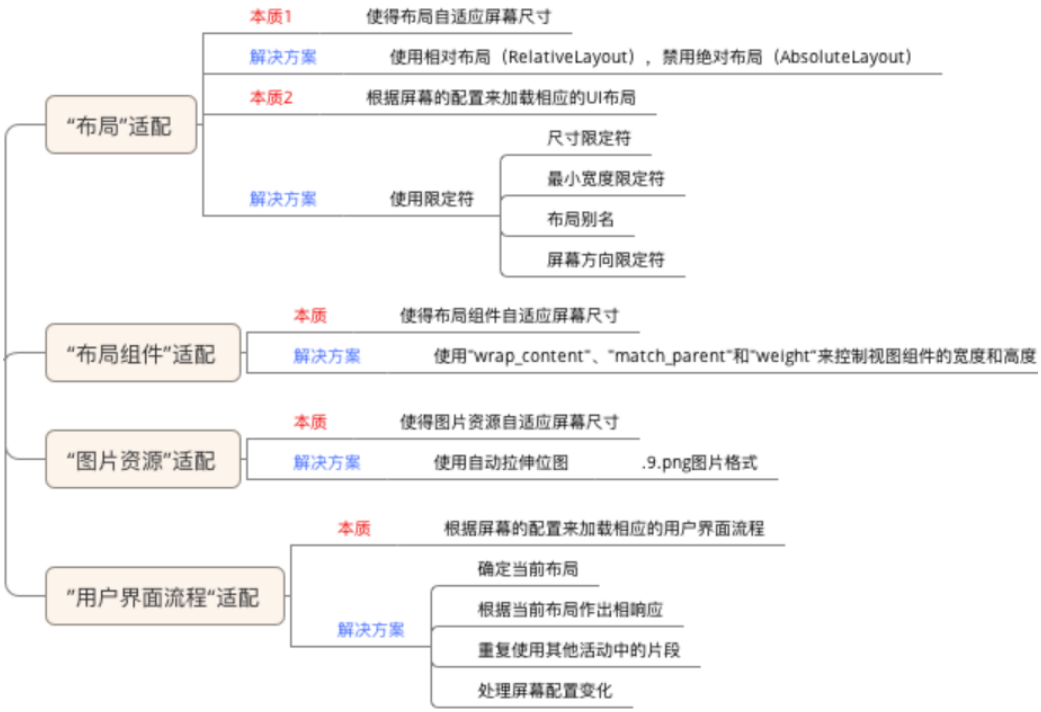


图 2: 屏幕尺寸适配解决方案

- 绝对布局（AbsoluteLayout）
  - 由于绝对布局（AbsoluteLayout）适配性极差，所以极少使用。

对于线性布局（Linearlayout）、相对布局（RelativeLayout）和帧布局（FrameLayout）需要根据需求进行选择，但要记住：

- RelativeLayout
  - 布局的子控件之间使用相对位置的方式排列，因为 RelativeLayout 讲究的是相对位置，即使屏幕的大小改变，视图之前的相对位置都不会变化，与屏幕大小无关，灵活性很强
- LinearLayout
  - 通过多层嵌套 LinearLayout 和组合使
  - 用"wrap_content" 和"match_parent" 已经可以构建出足够复杂的布局。但是 LinearLayout 无法准确地控制子视图之间的位置关系，只能简单的一个挨着一个地排列

所以，对于屏幕适配来说，使用相对布局（RelativeLayout）将会是更好的解决方案

- 本质 **2**：根据屏幕的配置来加载相应的 **UI** 布局
  - 应用场景：需要为不同屏幕尺寸的设备设计不同的布局

(b) 使用限定符

- 作用：通过配置限定符使得程序在运行时根据当前设备的配置（屏幕尺寸）自动加载合适的布局资源

限定符类型：

- 尺寸（size）限定符
- 最小宽度（Smallest-width）限定符

- 布局别名
- 屏幕方向（Orientation）限定符

i. 尺寸（size）限定符
使用场景：当一款应用显示的内容较多，希望进行以下设置：
- 在平板电脑和电视的屏幕（>7 英寸）上：实施" 双面板" 模式以同时显示更多内容
- 在手机较小的屏幕上：使用单面板分别显示内容

因此，我们可以使用尺寸限定符（layout-large）通过创建一个文件

```
res/layout-large/main.xml
```

来完成上述设定：
- 让系统在屏幕尺寸 >7 英寸时采用适配平板的双面板布局
- 反之（默认情况下）采用适配手机的单面板布局

文件配置如下：
- 适配手机的单面板（默认）布局：res/layout/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="match_parent"
      />
</LinearLayout>
```

- 适配尺寸 >7 寸平板的双面板布局:：res/layout-large/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp"
    />
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent"
    />
</LinearLayout>
```

请注意：
- 两个布局名称均为 main.xml，只有布局的目录名不同：第一个布局的目录名为：layout，第二个布局的目录名为：layout-large，包含了尺寸限定符（large）
- 被定义为大屏的设备 (7 寸以上的平板) 会自动加载包含了 large 限定符目录的布局，而小屏设备会加载另一个默认的布局

但要注意的是，这种方式只适合 **Android 3.2** 版本之前。

ii. 最小宽度（Smallest-width）限定符
- 背景：上述提到的限定符"large" 具体是指多大呢？似乎没有一个定量的指标，这便意味着可能没办法准确地根据当前设备的配置（屏幕尺寸）自动加载合适的布局资源
- 例子：比如说 large 同时包含着 5 寸和 7 寸，这意味着使用"large" 限定符的话我没办法实现为 5 寸和 7 寸的平板电脑分别加载不同的布局

于是，在 Android 3.2 及之后版本，引入了最小宽度（Smallest-width）限定符

- 定义：通过指定某个最小宽度（以 dp 为单位）来精确定位屏幕从而加载不同的 UI 资源
- 使用场景:
  - 你需要为标准 7 英寸平板电脑匹配双面板布局（其最小宽度为 600 dp），在手机（较小的屏幕上）匹配单面板布局
- 解决方案：
  - 您可以使用上文中所述的单面板和双面板这两种布局，但您应使用 sw600dp 指明双面板布局仅适用于最小宽度为 600 dp 的屏幕，而不是使用 large 尺寸限定符。
    - sw xxxdp，即 small width 的缩写，其不区分方向，即无论是宽度还是高度，只要大于 xxxdp，就采用次此布局
- 例子：
  - 使用了 layout-sw 600dp 的最小宽度限定符，即无论是宽度还是高度，只要大于 600dp，就采用 layout-sw 600dp 目录下的布局

代码展示：

- 适配手机的单面板（默认）布局：res/layout/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />
</LinearLayout>
```

- 适配尺寸 >7 寸平板的双面板布局：res/layout-sw600dp/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="400dp"
          android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.ArticleFragment"
          android:layout_width="fill_parent" />
</LinearLayout>
```

- 对于最小宽度 600 dp 的设备
  - 系统会自动加载 layout-sw600dp/main.xml（双面板）布局，否则系统就会选择 layout/main.xml（单面板）布局
  - （这个选择过程是 Android 系统自动选择的）

iii. 使用布局别名

设想这么一个场景

当你需要同时为 Android 3.2 版本前和 Android 3.2 版本后的手机进行屏幕尺寸适配的时候，由于尺寸限定符仅用于 Android 3.2 版本前，最小宽度限定符仅用于 Android 3.2 版本后，所以这会带来一个问题，为了很好地进行屏幕尺寸的适配，你需要同时维护 layout-sw600dp 和 layout-large 的两套 main.xml 平板布局，如下：

- 适配手机的单面板（默认）布局：res/layout/main.xml
- 适配尺寸 >7 寸平板的双面板布局（Android 3.2 前）：res/layout-large/main.xml

- 适配尺寸 >7 寸平板的双面板布局（Android 3.2 后）res/layout-sw600dp/main.xml

最后的两个文件的 xml 内容是完全相同的，这会带来：文件名的重复从而带来一些列后期维护的问题
于是为了要解决这种重复问题，我们引入了**--**布局别名
还是上面的例子，你可以定义以下布局：

- 适配手机的单面板（默认）布局：res/layout/main.xml
- 适配尺寸 >7 寸平板的双面板布局：res/layout/main_twopanes.xml

然后加入以下两个文件，以便进行 Android 3.2 前和 Android 3.2 后的版本双面板布局适配：

- res/values-large/layout.xml（Android 3.2 之前的双面板布局）

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

- res/values-sw600dp/layout.xml（Android 3.2 及之后的双面板布局）

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

注：

- 最后两个文件有着相同的内容，但是它们并没有真正去定义布局，它们仅仅只是将 main 设置成了 @layout/main_twopanes 的别名
- 由于这些文件包含 large 和 sw600dp 选择器，因此，系统会将此文件匹配到不同版本的 >7 寸平板上：
  - a. 版本低于 3.2 的平板会匹配 large 的文件
  - b. 版本高于 3.2 的平板会匹配 sw600dp 的文件

这样两个 layout.xml 都只是引用了 @layout/main_twopanes，就避免了重复定义布局文件的情况

iv. 屏幕方向（Orientation）限定符

- 使用场景：根据屏幕方向进行布局的调整
- 取以下为例子：
  - 小屏幕, 竖屏: 单面板
  - 小屏幕, 横屏: 单面板
  - 7 英寸平板电脑，纵向：单面板，带操作栏
  - 7 英寸平板电脑，横向：双面板，宽，带操作栏
  - 10 英寸平板电脑，纵向：双面板，窄，带操作栏
  - 10 英寸平板电脑，横向：双面板，宽，带操作栏
  - 电视，横向：双面板，宽，带操作栏
- 方法是
  - 先定义类别：单/双面板、是否带操作栏、宽/窄
    - 定义在 res/layout/ 目录下的某个 XML 文件中
  - 再进行相应的匹配：屏幕尺寸（小屏、7 寸、10 寸）、方向（横、纵）
    - 使用布局别名进行匹配
- 在 **res/layout/** 目录下的某个 **XML** 文件中定义所需要的布局类别
  - （单/双面板、是否带操作栏、宽/窄）
  - res/layout/onepane.xml:(单面板)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="match_parent" />
</LinearLayout>
```

– res/layout/onepane_with_bar.xml:(单面板带操作栏)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
          android:id="@+id/linearLayout1"
          android:gravity="center"
          android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
              android:layout_height="wrap_content"
              android:layout_width="wrap_content"
              android:src="@drawable/logo"
              android:paddingRight="30dp"
              android:layout_gravity="left"
              android:layout_weight="0" />
        <View android:layout_height="wrap_content"
             android:id="@+id/view1"
             android:layout_width="wrap_content"
             android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
            android:background="@drawable/button_bg"
            android:layout_height="match_parent"
            android:layout_weight="0"
            android:layout_width="120dp"
            style="@style/CategoryButtonStyle"/>
    </LinearLayout>
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="match_parent"
    />
</LinearLayout>
```

– res/layout/twopanes.xml:(双面板，宽布局)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="400dp"
          android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.ArticleFragment"
          android:layout_width="fill_parent" />
</LinearLayout>
```

– res/layout/twopanes_narrow.xml:(双面板，窄布局)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="200dp"
```

```
            android:layout_marginRight="10dp"/>
        <fragment android:id="@+id/article"
            android:layout_height="fill_parent"
            android:name="com.example.android.newsreader.ArticleFragment"
            android:layout_width="fill_parent" />
</LinearLayout>
```

- 使用布局别名进行相应的匹配
  - (屏幕尺寸（小屏、7 寸、10 寸)、方向（横、纵）)
  - res/values/layouts.xml:（默认布局）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

  - 可为 resources 设置 bool，通过获取其值来动态判断目前已处在哪个适配布局
  - res/values-sw600dp-land/layouts.xml
    *（大屏、横向、双面板、宽-Andorid 3.2 版本后）

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

  - res/values-sw600dp-port/layouts.xml
    *（大屏、纵向、单面板带操作栏-Andorid 3.2 版本后）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

  - res/values-large-land/layouts.xml
    *（大屏、横向、双面板、宽-Andorid 3.2 版本前）

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

  - res/values-large-port/layouts.xml
    *（大屏、纵向、单面板带操作栏-Andorid 3.2 版本前）

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

  - 这里没有完全把全部尺寸匹配类型的代码贴出来，大家可以自己去尝试把其补充完整

2. " 布局组件" 匹配

   本质：使得布局组件自适应屏幕尺寸

   - 做法: 使用"wrap_content"、"match_parent" 和"weight" 来控制视图组件的宽度和高度
     - "wrap_content"
       * 相应视图的宽和高就会被设定成所需的最小尺寸以适应视图中的内容
     - "match_parent"(在 Android API 8 之前叫作"fill_parent")
       * 视图的宽和高延伸至充满整个父布局

- "weight"
  * 1. 定义：是线性布局（Linelayout）的一个独特比例分配属性
  * 2. 作用：使用此属性设置权重，然后按照比例对界面进行空间的分配，公式计算是：控件宽度 = 控件设置宽度 + 剩余空间所占百分比宽幅
  * 具体可以参考这篇文章,讲解得非常详细 http://mobile.51cto.com/abased-375428.htm

通过使用"wrap_content"、"match_parent" 和"weight" 来替代硬编码的方式定义视图大小 & 位置，你的视图要么仅仅使用了需要的那边一点空间，要么就会充满所有可用的空间，即按需占据空间大小，能让你的布局元素充分适应你的屏幕尺寸

3. " 图片资源" 匹配

本质：使得图片资源在不同屏幕密度上显示相同的像素效果

- 做法：使用自动拉伸位图：**Nine-Patch** 的图片类型
  - 假设需要匹配不同屏幕大小，你的图片资源也必须自动适应各种屏幕尺寸
    * 使用场景：一个按钮的背景图片必须能够随着按钮大小的改变而改变。
    * 使用普通的图片将无法实现上述功能, 因为运行时会均匀地拉伸或压缩你的图片
- 解决方案：使用自动拉伸位图（nine-patch 图片），后缀名是.9.png，它是一种被特殊处理过的 PNG 图片，设计时可以指定图片的拉伸区域和非拉伸区域；使用时，系统就会根据控件的大小自动地拉伸你想要拉伸的部分
  - 1. 必须要使用.9.png 后缀名，因为系统就是根据这个来区别 nine-patch 图片和普通的 PNG 图片的；
  - 2. 当你需要在一个控件中使用 nine-patch 图片时, 如

```
android:background="@drawable/button"
```

系统就会根据控件的大小自动地拉伸你想要拉伸的部分

4. " 用户界面流程" 匹配

- 使用场景：我们会根据设备特点显示恰当的布局，但是这样做，会使得用户界面流程可能会有所不同。
- 例如，如果应用处于双面板模式下，点击左侧面板上的项即可直接在右侧面板上显示相关内容；而如果该应用处于单面板模式下，点击相关的内容应该跳转到另外一个 Activity 进行后续的处理。

本质：根据屏幕的配置来加载相应的用户界面流程

- 做法: 进行用户界面流程的自适应配置：
  - 确定当前布局
  - 根据当前布局做出响应
  - 重复使用其他活动中的片段
  - 处理屏幕配置变化
- 步骤 1：确定当前布局
  - 由于每种布局的实施都会稍有不同，因此我们需要先确定当前向用户显示的布局。例如，我们可以先了解用户所处的是" 单面板" 模式还是" 双面板" 模式。要做到这一点，可以通过查询指定视图是否存在以及是否已显示出来。

```
public class NewsReaderActivity extends FragmentActivity {
    boolean mIsDualPane;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);
        View articleView = findViewById(R.id.article);
        mIsDualPane = articleView != null &&
                      articleView.getVisibility() == View.VISIBLE;
    }
}
```

  - 这段代码用于查询" 报道" 面板是否可用，与针对具体布局的硬编码查询相比，这段代码的灵活性要大得多。

- 步骤 2：根据当前布局做出响应
  - 有些操作可能会因当前的具体布局而产生不同的结果。
  - 例如，在新闻阅读器示例中，如果用户界面处于双面板模式下，那么点击标题列表中的标题就会在右侧面板中打开相应报道；但如果用户界面处于单面板模式下，那么上述操作就会启动一个独立活动：

```
@Override
public void onHeadlineSelected(int index) {
    mArtIndex = index;
    if (mIsDualPane) {
        /* display article on the right pane */
        mArticleFragment.displayArticle(mCurrentCat.getArticle(index));
    } else {
        /* start a separate activity */
        Intent intent = new Intent(this, ArticleActivity.class);
        intent.putExtra("catIndex", mCatIndex);
        intent.putExtra("artIndex", index);
        startActivity(intent);
    }
}
```

- 步骤 3：重复使用其他活动中的片段
  - 多屏幕设计中的重复模式是指，对于某些屏幕配置，已实施界面的一部分会用作面板；但对于其他配置，这部分就会以独立活动的形式存在。
    - 例如，在新闻阅读器示例中，对于较大的屏幕，新闻报道文本会显示在右侧面板中；但对于较小的屏幕，这些文本就会以独立活动的形式存在。
  - 在类似情况下，通常可以在多个活动中重复使用相同的 Fragment 子类以避免代码重复。例如，在双面板布局中使用了 ArticleFragment：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.HeadlinesFragment"
          android:layout_width="400dp"
          android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
          android:layout_height="fill_parent"
          android:name="com.example.android.newsreader.ArticleFragment"
          android:layout_width="fill_parent" />
</LinearLayout>
```

  - 然后又在小屏幕的 Activity 布局中重复使用了它：

```
ArticleFragment frag = new ArticleFragment();
getSupportFragmentManager().beginTransaction().
    add(android.R.id.content, frag).commit();
```

- 步骤 4：处理屏幕配置变化
  - 如果我们使用独立 Activity 实施界面的独立部分，那么请注意，我们可能需要对特定配置变化（例如屏幕方向的变化）做出响应，以便保持界面的一致性。
    * 例如，在运行 Android 3.0 或更高版本的标准 7 英寸平板电脑上，如果新闻阅读器示例应用运行在纵向模式下，就会在使用独立活动显示新闻报道；但如果该应用运行在横向模式下，就会使用双面板布局。
  - 也就是说，如果用户处于纵向模式下且屏幕上显示的是用于阅读报道的活动，那么就需要在检测到屏幕方向变化（变成横向模式）后执行相应操作，即停止上述活动并返回主活动，以便在双面板布局中显示相关内容：

```
public class ArticleActivity extends FragmentActivity {
    int mCatIndex, mArtIndex;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCatIndex = getIntent().getExtras().getInt("catIndex", 0);
        mArtIndex = getIntent().getExtras().getInt("artIndex", 0);
        // If should be in two-pane mode, finish to return to main activity
        if (getResources().getBoolean(R.bool.has_two_panes)) {
            finish();
            return;
        }
        ...
}
```

通过上面一系列步骤，我们就完全可以建立一个可以根据用户界面配置进行自适应的应用程序 App 了。

6.6.2 屏幕密度匹配

问题：如何进行屏幕密度匹配？

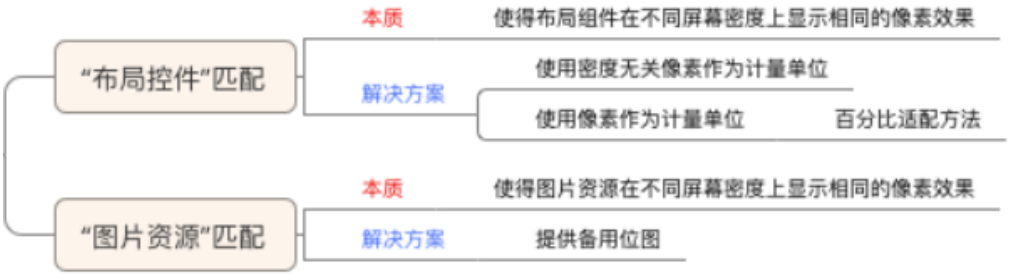


图 3: 屏幕密度匹配解决方案

1. " 布局控件" 匹配

   本质：使得布局组件在不同屏幕密度上显示相同的像素效果

   (a) 使用密度无关像素作为计量单位

   - 由于各种屏幕的像素密度都有所不同，因此相同数量的像素在不同设备上的实际大小也有所差异，这样使用像素（px）定义布局尺寸就会产生问题。
   - 因此，请务必使用密度无关像素 dp 或 ** 独立比例像素 sp ** 单位指定尺寸。

   密度无关像素

   - 含义：density-independent pixel，叫 dp 或 dip，与终端上的实际物理像素点无关。
   - 单位：dp，可以保证在不同屏幕像素密度的设备上显示相同的效果

  - Android 开发时用 dp 而不是 px 单位设置图片大小，是 Android 特有的单位
  - 场景：假如同样都是画一条长度是屏幕一半的线，如果使用 px 作为计量单位，那么在 480x800 分辨率手机上设置应为 240px；在 320x480 的手机上应设置为 160px，二者设置就不同了；如果使用 dp 为单位，在这两种分辨率下，160dp 都显示为屏幕一半的长度。
- dp 与 px 的转换
  - 因为 ui 给你的设计图是以 px 为单位的，Android 开发则是使用 dp 作为单位的，那么该如何转换呢？

密度类型	代表的分辨率（px）	屏幕密度（dpi）	换算（px/dp）	比例
低密度（ldpi）	240x320	120	1dp=0.75px	3
中密度（mdpi）	320x480	160	1dp=1px	4
高密度（hdpi）	480x800	240	1dp=1.5px	6
超高密度（xhdpi）	720x1280	320	1dp=2px	8
超超高密度（xxhdpi）	1080x1920	480	1dp=3px	12

在 Android 中，规定以 160dpi（即屏幕分辨率为 320x480）为基准：1dp=1px

独立比例像素

- 含义：scale-independent pixel，叫 sp 或 sip
- 单位：sp
  - Android 开发时用此单位设置文字大小，可根据用户的偏好文字大小/字体大小首选项进行缩放
  - 推荐使用 12sp、14sp、18sp、22sp 作为字体设置的大小，不推荐使用奇数和小数，容易造成精度的丢失问题；小于 12sp 的字体会太小导致用户看不清

所以，为了能够进行不同屏幕像素密度的匹配，我们推荐：

- 使用 dp 来代替 px 作为控件长度的统一度量单位
- 使用 sp 作为文字的统一度量单位

可是，请看以下一种场景：

- Nexus5 的总宽度为 360dp，我们现在在水平方向上放置两个按钮，一个是 150dp 左对齐，另外一个是 200dp 右对齐，那么中间留有 10dp 间隔；但假如同样地设置在 Nexus S（屏幕宽度是 320dp），会发现，两个按钮会重叠，因为 320dp<200+150dp

从上面可以看出，由于 Android 屏幕设备的多样性，如果使用 dp 来作为度量单位，并不是所有的屏幕的宽度都具备相同的 dp 长度

- 再次明确，屏幕宽度和像素密度没有任何关联关系

所以说，dp 解决了同一数值在不同分辨率中展示相同尺寸大小的问题（即屏幕像素密度匹配问题），但却没有解决设备尺寸大小匹配的问题。（即屏幕尺寸匹配问题）

- 当然，我们一开始讨论的就是屏幕尺寸匹配问题，使用 match_parent、wrap_content 和 weight，尽可能少用 dp 来指定控件的具体长宽，大部分的情况我们都是可以做到适配的。

那么该如何解决控件的屏幕尺寸和屏幕密度的适配问题呢？

从上面可以看出：

- 因为屏幕密度（分辨率）不一样，所以不能用固定的 px
- 因为屏幕宽度不一样，所以要小心的用 dp

因为本质上是希望使得布局组件在不同屏幕密度上显示相同的像素效果，那么，之前是绕了个弯使用 dp 解决这个问题，那么到底能不能直接用 px 解决呢？

- 即根据不同屏幕密度，控件选择对应的像素值大小

(b) 百分比适配方法

- 以某一分辨率为基准，生成所有分辨率对应像素数列表
- 将生成像素数列表存放在 res 目录下对应的 values 文件下
- 根据 UI 设计师给出设计图上的尺寸，找到对应像素数的单位，然后设置给控件即可

i. 步骤 1：以某一分辨率为基准，生成所有分辨率对应像素数列表
现在我们以 320x480 的分辨率为基准：

- 将屏幕的宽度分为 320 份，取值为 x1~x320
- 将屏幕的高度分为 480 份，取值为 y1~y480

然后生成该分辨率对应像素数的列表，如下图：

- lay_x.xml（宽）

```
<?xml version="1.0" encoding="utf-8"?>
<resources><dimen name="x1">1.0px</dimen>
    <dimen name="x2">2.0px</dimen>
    <dimen name="x3">3.0px</dimen>
    <dimen name="x4">4.0px</dimen>
    <dimen name="x5">5.0px</dimen>
    <dimen name="x6">6.0px</dimen>
    <dimen name="x7">7.0px</dimen>
    <dimen name="x8">8.0px</dimen>
    <dimen name="x9">9.0px</dimen>
    <dimen name="x10">10.0px</dimen>
    ...
    <dimen name="x300">300.0px</dimen>
    <dimen name="x301">301.0px</dimen>
    <dimen name="x302">302.0px</dimen>
    <dimen name="x303">303.0px</dimen>
    <dimen name="x304">304.0px</dimen>
    <dimen name="x305">305.0px</dimen>
    <dimen name="x306">306.0px</dimen>
    <dimen name="x307">307.0px</dimen>
    <dimen name="x308">308.0px</dimen>
    <dimen name="x309">309.0px</dimen>
    <dimen name="x310">310.0px</dimen>
    <dimen name="x311">311.0px</dimen>
    <dimen name="x312">312.0px</dimen>
    <dimen name="x313">313.0px</dimen>
    <dimen name="x314">314.0px</dimen>
    <dimen name="x315">315.0px</dimen>
    <dimen name="x316">316.0px</dimen>
    <dimen name="x317">317.0px</dimen>
    <dimen name="x318">318.0px</dimen>
    <dimen name="x319">319.0px</dimen>
    <dimen name="x320">320px</dimen>
</resources>
```

- lay_y.xml（高）

```
<?xml version="1.0" encoding="utf-8"?>
<resources><dimen name="y1">1.0px</dimen>
    <dimen name="y2">2.0px</dimen>
    <dimen name="y3">3.0px</dimen>
    <dimen name="y4">4.0px</dimen>
    ...
    <dimen name="y480">480px</dimen>
</resources>
```

找到基准后，是时候把其他分辨率补全了，现今以写 1080x1920 的分辨率为例：

- 因为基准是 320x480，所以 1080/320=3.375px，1920/480=4px，所以相应文件应该是

  – lay_x.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources><dimen name="x1">3.375px</dimen>
```

```
    <dimen name="x2">6.65px</dimen>
    <dimen name="x3">10.125px</dimen>
    ...
    <dimen name="x320">1080px</dimen>
</resources>
```

– lay_y.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources><dimen name="y1">4px</dimen>
    <dimen name="y2">8px</dimen>
    <dimen name="y3">12px</dimen>
    <dimen name="y4">16px</dimen>
    ...
    <dimen name="y480">1920px</dimen>
</resources>
```

用上面的方法把你需要适配的分辨率的像素列表补全吧 ~
作为程序猿的我们当然不会做手写的这些蠢事!!! 多谢 @ 鸿洋大神提供了自动生成工具（内置了常用的分辨率），大家可以直接点击这里下载
注：工具默认基准为 400*320，当然对于特殊需求，通过命令行指定即可：

```
java -jar 文件名.jar 基准宽 基准高 额外支持尺寸 1 的宽，额外支持尺寸 1 的高 _ 额外支持尺寸 2 的宽，额外支持尺寸
```

例如：需要设置的基准是 800x1280，额外支持尺寸：735x1152 ；3200x4500；

```
java -jar 文件名.jar 800 1280 735，1152_3200,4500
```

ii. 步骤 2：把生成的各像素数列表放到对应的资源文件
将生成像素数列表（lay_x.xml 和 lay_y.xml）存放在 res 目录下对应的 values 文件（注意宽、高要对应），如下图：

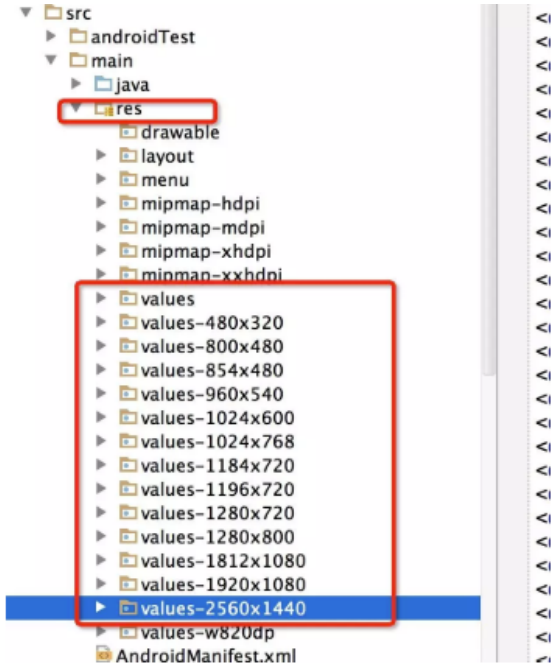


图 4: res 目录下对应的 values 文件

注：

- 分辨率为 480x320 的资源文件应放在 res/values-480x320 文件夹中；同理分辨率为 1920x1080 的资源文件应放在 res/values-1920x1080 文件夹中。（其中 values-480x320 是分辨率限定符）

- 必须在默认 values 里面也创建对应默认 lay_x.xml 和 lay_y.xml 文件，如下图
  - lay_x.xml

```
<?xml version="1.0" encoding="utf-8">
<resources>
    <dimen name="x1">1.0dp</dimen>
    <dimen name="x2">2.0dp</dimen>
    ...
</resources>
```

- 因为对于没有生成对应分辨率文件的手机，会使用默认 values 文件夹，如果默认 values 文件夹没有（即没有对应的分辨率、没有对应 dimen）就会报错，从而无法进行屏幕适配。
  - （注意对应单位改为 dp，而不同于上面的 px。因为不知道机型的分辨率，所以默认分辨率文件只好默认为 x1=1dp 以保证尽量兼容（又回到 dp 老方法了），这也是这个解决方案的一个弊端）

iii. 步骤 3：根据 UI 设计师给出某一分辨率设计图上的尺寸，找到对应像素数的单位，然后设置给控件即可
如下图：

```
<FrameLayout >
    <Button
        android:layout_gravity="center"
        android:gravity="center"
        android:text="@string/hello_world"
        android:layout_width="@dimen/x160"
        android:layout_height="@dimen/y160"/>
</FrameLayout>
```

使用上述的适配方式，应该能进行 90% 的适配了，但其缺点还是很明显：

- 由于实际上还是使用 px 作为长度的度量单位，所以和 google 的要求使用 dp 作为度量单位会有所背离
- 必须尽可能多的包含所有分辨率，因为这个是使用这个方案的基础，如果有某个分辨率缺少，将无法完成该屏幕的适配
- 过多的分辨率像素描述 xml 文件会增加软件包的大小和维护的难度

2. " 图片资源" 匹配 本质：使得图片资源在不同屏幕密度上显示相同的像素效果

- 做法：提供备用位图（符合屏幕尺寸的图片资源）
  - 由于 Android 可在各种屏幕密度的设备上运行，因此我们提供的位图资源应该始终可以满足各类密度的要求：

密度类型	代表的分辨率（px）	系统密度（dpi）
低密度（ldpi）	240x320	120
中密度（mdpi）	320x480	160
高密度（hdpi）	480x800	240
超高密度（xhdpi）	720x1280	320
超超高密度（xxhdpi）	1080x1920	480

- 步骤 1：根据以下尺寸范围针对各密度生成相应的图片。
  - 比如说，如果我们为 xhdpi 设备生成了 200x200 px 尺寸的图片，就应该按照相应比例地为 hdpi、mdpi 和 ldpi 设备分别生成 150x150、100x100 和 75x75 尺寸的图片

即一套分辨率 = 一套位图资源（这个当然是 Ui 设计师做了）

- 步骤 2：将生成的图片文件放在 res/ 下的相应子目录中 (mdpi、hdpi、xhdpi、xxhdpi)，系统就会根据运行您应用的设备的屏幕密度自动选择合适的图片

- 步骤 3：通过引用 @drawable/id，系统都能根据相应屏幕的屏幕密度（dpi）自动选取合适的位图。

注：

- 如果是.9 图或者是不需要多个分辨率的图片，放在 drawable 文件夹即可
- 对应分辨率的图片要正确的放在合适的文件夹，否则会造成图片拉伸等问题。

更好地方案解决" 图片资源" 适配问题

上述方案是常见的一种方案，这固然是一种解决办法，但缺点在于：

- 每套分辨率出一套图，为美工或者设计增加了许多工作量
- 对 Android 工程文件的 apk 包变的很大

那么，有没有一种方法：

- 保证屏幕密度适配
- 可以最小占用设计资源
- 使得 apk 包不变大（只使用一套分辨率的图片资源）

下面我们就来介绍这个方法：

- 只需选择唯一一套分辨率规格的图片资源

方法介绍

(a) 先来理解下 Android 加载资源过程

Android SDK 会根据屏幕密度自动选择对应的资源文件进行渲染加载（自动渲染）

- 比如说，SDK 检测到你手机的分辨率是 320x480（dpi=160），会优先到 drawable-mdpi 文件夹下找对应的图片资源；但假设你只在 xhpdi 文件夹下有对应的图片资源文件（mdpi 文件夹是空的），那么 SDK 会去 xhpdi 文件夹找到相应的图片资源文件，然后将原有大像素的图片自动缩放成小像素的图片，于是大像素的图片照样可以在小像素分辨率的手机上正常显示。
- 具体请看http://blog.csdn.net/xiebudong/article/details/37040263

所以理论上来说只需要提供一种分辨率规格的图片资源就可以了。

那么应该提供哪种分辨率规格呢？

- 如果只提供 ldpi 规格的图片，对于大分辨率（xdpi、xxdpi）的手机如果把图片放大就会不清晰

所以需要提供一套你需要支持的最大 dpi 分辨率规格的图片资源，这样即使用户的手机分辨率很小，这样图片缩小依然很清晰。那么这一套最大 dpi 分辨率规格应该是哪种呢？是现在市面手机分辨率最大可达到 1080X1920 的分辨率（dpi=xxdpi=480）吗？

(a) xhdpi 应该是首选

原因如下：

- xhdpi 分辨率以内的手机需求量最旺盛
  - 目前市面上最普遍的高端机的分辨率还多集中在 720X1080 范围内（xhdpi），所以目前来看 xhpdi 规格的图片资源成为了首选

- 节省设计资源 & 工作量
  - 在现在的 App 开发中（iOS 和 Android 版本），有些设计师为了保持 App 不同版本的体验交互一致，可能会以 iPhone 手机为基础进行设计，包括后期的切图之类的。
  - 设计师们一般都会用最新的 iPhone6 和 iPhone5s（5s 和 5 的尺寸以及分辨率都一样）来做原型设计，所有参数请看下图

机型	分辨率（px）	屏幕尺寸（inch）	系统密度（dpi）
iPhone 5s	640X1164	4	332
iPhone 6	1334x750	4.7	326
iPhone 6 Plus	1080x1920	5	400

iPhone 主流的屏幕 dpi 约等于 320, 刚好属于 xhdpi，所以选择 xhdpi 作为唯一一套 dpi 图片资源，可以让设计师不用专门为 Android 端切图，直接把 iPhone 的那一套切好的图片资源放入 drawable-xhdpi 文件夹里就好，这样大大减少的设计师的工作量！

额外小 **tips**

- ImageView 的 ScaleType 属性
  - 设置不同的 ScaleType 会得到不同的显示效果，一般情况下，设置为 centerCrop 能获得较好的适配效果。
- 动态设置
  - 使用场景:有些情况下,我们需要动态的设置控件大小或者是位置,比如说 popwindow 的显示位置和偏移量等

这时我们可以动态获取当前的屏幕属性，然后设置合适的数值

```
public class ScreenSizeUtil {
    public static int getScreenWidth(Activity activity) {
        return activity.getWindowManager().getDefaultDisplay().getWidth();
}
    public static int getScreenHeight(Activity activity) {
      return activity.getWindowManager().getDefaultDisplay().getHeight();
   }
}
```