

# Android Service

deepwaterooo

2021 年 12 月 15 日

## 目录

- 1 Android: Service 生命周期完全解析 1**
  - 1.1 生命周期常用方法 1
- 2 Android 四大组件：一份全面 & 简洁的 Service 知识讲解攻略 2**
  - 2.1 前言 2
  - 2.2 目录 3
  - 2.3 简介 3
  - 2.4 生命周期 3
  - 2.5 类型 3
    - 2.5.1 具体分类 4
    - 2.5.2 详细介绍 4
  - 2.6 使用讲解 5
  - 2.7 其他思考 5
    - 2.7.1 Service 与 Thread 的区别 5
    - 2.7.2 Service 和 IntentService 的区别 6
- 3 Android:（本地、可通信的、前台、远程）Service 使用全面介绍 7**
  - 3.1 前言 7
  - 3.2 目录 7
  - 3.3 Service 分类 7
    - 3.3.1 Service 的类型 7
    - 3.3.2 特点 7
  - 3.4 具体使用解析 8
    - 3.4.1 本地 Service 8
    - 3.4.2 可通信的服务 Service 10
    - 3.4.3 前台 Service 13
    - 3.4.4 远程 Service 14
  - 3.5 使用场景 14
- 4 Android 多线程解析: IntentService（含源码解析） 15**
  - 4.1 前言 16
  - 4.2 定义 16
  - 4.3 作用 16
  - 4.4 使用场景 16
  - 4.5 使用步骤 16
  - 4.6 实例应用 16
  - 4.7 源码分析 18
    - 4.7.1 IntentService 如何单独开启 1 个新的工作线程 19

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 . . . . . 19

4.8 总结 . . . . . 20

4.9 注意事项 . . . . . 20

4.10对比 . . . . . 21

4.10.1IntentService 与 Service 的区别 . . . . . 21

4.10.2IntentService 与其他线程的区别 . . . . . 21

4.11 总结 . . . . . 21

5 Android: 远程服务 Service (含 AIDL & IPC 讲解) 21

5.1 前言 . . . . . 21

5.2 远程服务与本地服务的区别 . . . . . 22

5.3 使用场景 . . . . . 22

5.4 使用场景 . . . . . 22

5.5 具体使用 . . . . . 22

5.6 具体实例 . . . . . 23

5.6.1 服务器端 (Service) . . . . . 23

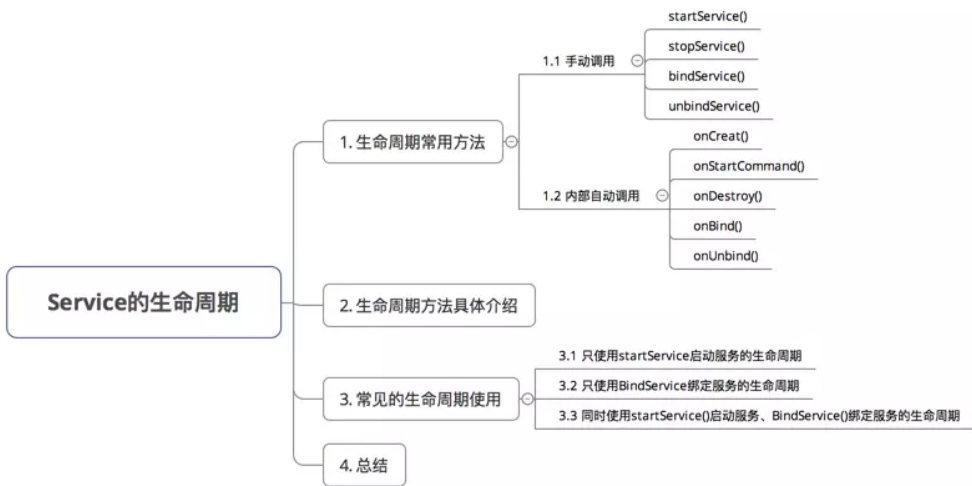
5.6.2 客户端 (Client) . . . . . 24

5.6.3 测试结果 . . . . . 26

5.6.4 Demo 地址 . . . . . 26

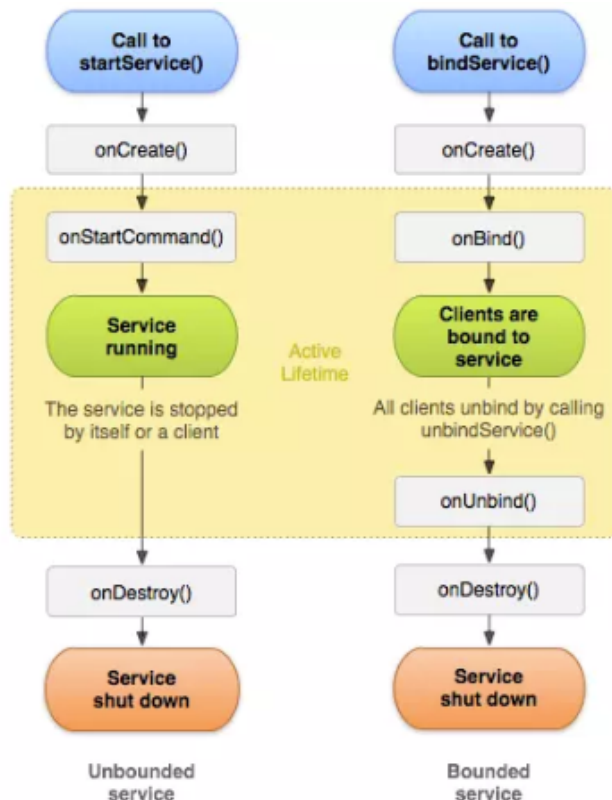
# 1 Android: Service 生命周期完全解析

• 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

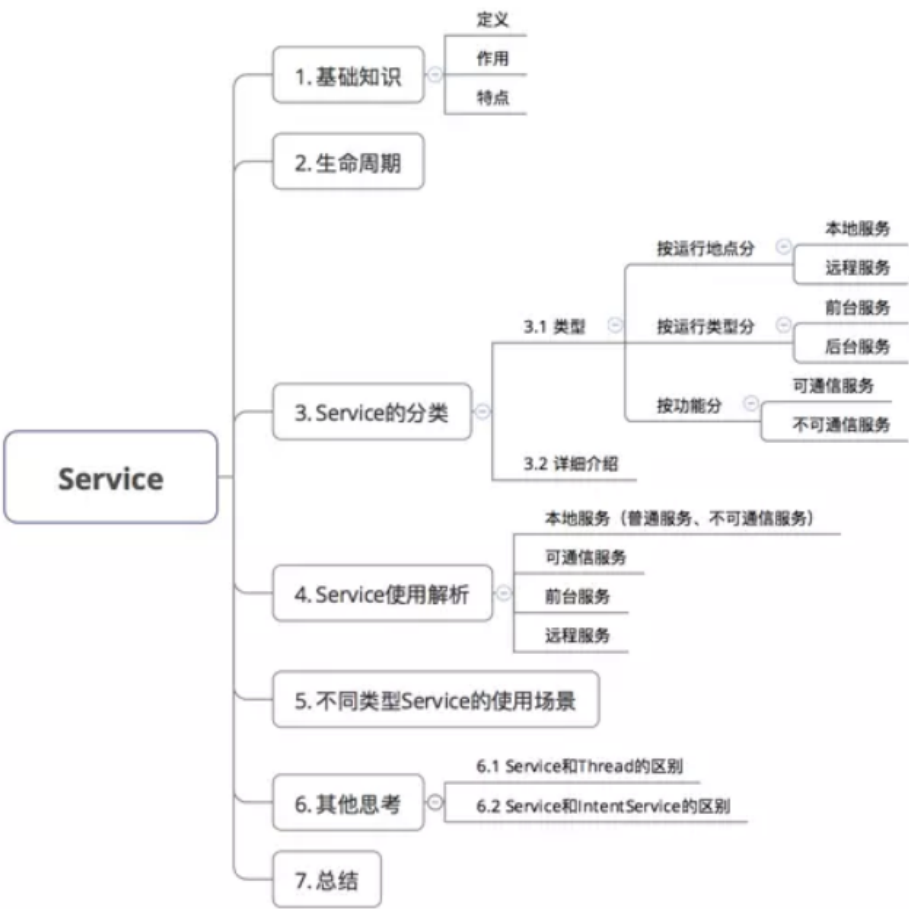
## 2 Android 四大组件：一份全面 & 简洁的 Service 知识讲解攻略

- <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 <small>LocalService</small>	<ul style="list-style-type: none"><li>• 运行在主线程</li><li>• 主进程被终止后，服务也会终止</li></ul>	<ul style="list-style-type: none"><li>• 节约资源；</li><li>• 通信方便：由于在同1进程因此不需IPC和AIDL</li></ul>	限制性大 <small>(主进程被终止后，服务也会终止)</small>	需依附某个进程的服务 <small>(最常用的服务类型，如 音乐播放)</small>
远程服务 <small>RemoteService</small>	<ul style="list-style-type: none"><li>• 运行在独立进程；</li><li>• 服务常驻在后台，不受其他Activity影响</li></ul>	灵活：服务常驻在后台，不受其他Activity影响	<ul style="list-style-type: none"><li>• 消耗资源：单独进程</li><li>• 使用AIDL进行IPC复杂</li></ul>	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 <small>(用户可看到)</small>	服务使用时，需让用户知道 & 进行相关操作，如 音乐播放服务。 <small>(服务被终止时，通知栏的通知也会消失)</small>
后台服务	处于后台的服务 <small>(用户无法看到)</small>	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 <small>(服务被终止时，用户无法知道)</small>

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

类别	特点	应用场景
不可通信的后台服务	<ul style="list-style-type: none"><li>• 用startService () 启动</li><li>• 调用者退出后Service 仍然存在</li></ul>	服务不需与Activity & Service 通信
可通信的后台服务	<ul style="list-style-type: none"><li>• 用bindService () 启动</li><li>• 调用者退出后, 随着调用者销毁</li></ul>	服务需与Activity & Service通信、需控制服务开始时刻 <small>• 节约系统资源 = 第1次bindService() 时才会创建服务的实例 &amp; 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</small>
	<ul style="list-style-type: none"><li>• 使用StartService () 、bindService() 启动</li><li>• 调用者退出后, 随着调用者销毁</li></ul>	需与Activity & Service通信、不需控制服务开始时刻 (服务一开始便运行)

类型	特点	应用场景
不可通信 后台服务	-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 后台服务	-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 后台服务	-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 <ul style="list-style-type: none"> <li>完全依赖UI / Activity, 只要进程还在, Service就可继续运行</li> <li>所有Activity都可 与 Service关联 获得Binder实例 &amp; 操作其中的方法</li> <li>若要处理耗时操作, 则在Service里创建Thread子线程执行</li> </ul>
Thread		工作线程	Activity <ul style="list-style-type: none"> <li>即 依赖于某个Activity</li> <li>在一个Activity中创建的子线程, 另一个Activity无法对其进行操作;</li> <li>Activity 很难控制 Thread</li> <li>当Activity被销毁后, 就无法再获取到之前创建的子线程实例</li> </ul>

### 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 <ul style="list-style-type: none"> <li>完全依赖UI / Activity, 只要进程还在, Service就可继续运行</li> <li>所有Activity都可 与 Service关联 获得Binder实例 &amp; 操作其中的方法</li> <li>若要处理耗时操作, 则在Service里创建Thread子线程执行</li> </ul>
工作线程	Activity <ul style="list-style-type: none"> <li>即 依赖于某个Activity</li> <li>在一个Activity中创建的子线程, 另一个Activity无法对其进行操作;</li> <li>Activity 很难控制 Thread</li> <li>当Activity被销毁后, 就无法再获取到之前创建的子线程实例</li> </ul>

- 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)

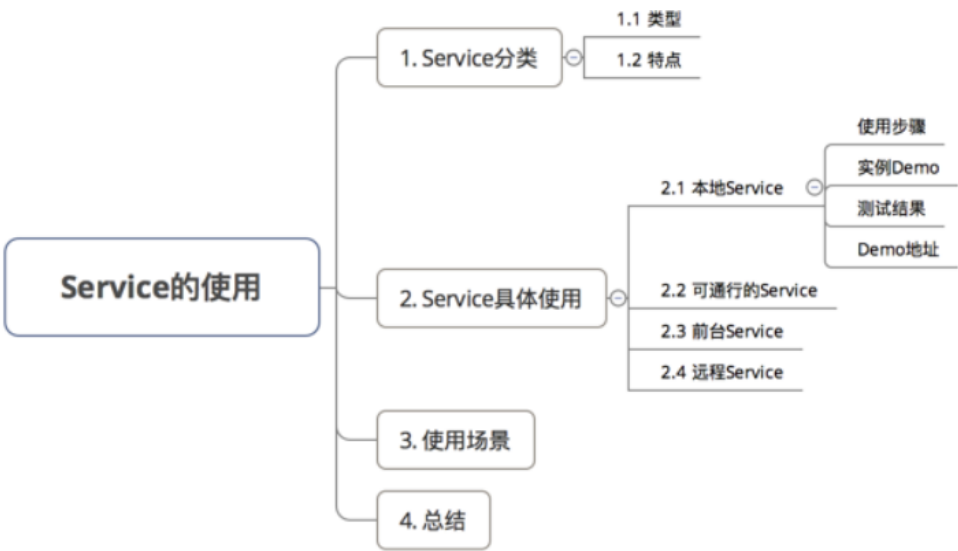
### 3 Android:（本地、可通信的、前台、远程）Service 使用全面介绍

• <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](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 onCreate()");
    }
    @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 onDestroy()");
    }
    @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);
        stopService.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:supportRtl="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	Service 的类名	
android:label	Service 的名字	若不设置，默认为 Service 的名字
android:icon	Service 的图标	
android:permission	申明此 Service 的权限	有提供了该权限的应用才能连接此服务
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 将会默认启动；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

```
11-04 22:06:05.815 8707-8707/scut.carson_ho.demo_service I/System.out: 执行了onCreat()
11-04 22:06:05.815 8707-8707/scut.carson_ho.demo_service I/System.out: 执行了onStartCommand()
11-04 22:06:06.463 8707-8707/scut.carson_ho.demo_service I/System.out: 执行了onDestroy()
```

4. Demo 地址

- Carson.ho 的 Github 地址: Demo\_for\_Service
  - [https://github.com/Carson-Ho/Demo\\_Service/tree/5e2a70cf2d75c56bbfa1abc0ead](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](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 onCreate()");
    }
    @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 onDestroy()");
    }
    @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

```

17043-17043/scut.carson_ho.demo_service I/System.out: 执行了onCreat()
17043-17043/scut.carson_ho.demo_service I/System.out: 执行了onBind()
17043-17043/scut.carson_ho.demo_service I/System.out: Service关联了Activity,并在Activity执行了Service的
17043-17043/scut.carson_ho.demo_service I/System.out: 执行了onUnbind()
17043-17043/scut.carson_ho.demo_service I/System.out: 执行了onDestory()

```

### 3. Demo

- carson.ho 的 Github 地址: Demo\_for\_Service
  - [https://github.com/Carson-Ho/Demo\\_Service/tree/719e3b9ffd5017c334cdfdaf45b](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 onCreate()");

    // 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 Builder 对象
    Notification.Builder builder = new Notification.Builder(this);
    builder.setContentTitle(" 前台服务通知的标题");// 设置通知的标题
    builder.setContentText(" 前台服务通知的内容");// 设置通知的内容
    builder.setSmallIcon(R.mipmap.ic_launcher); // 设置通知的图标
    builder.setContentIntent(pendingIntent); // 设置点击通知后的操作
    Notification notification = builder.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 <small>(LocalService)</small>	<ul style="list-style-type: none"> <li>运行在主线程</li> <li>主进程被终止后，服务也会终止</li> </ul>	<ul style="list-style-type: none"> <li>节约资源；</li> <li>通信方便：由于在同1进程因此不需IPC和AIDL</li> </ul>	限制性大 <small>(主进程被终止后，服务也会终止)</small>	需依附某个进程的服务 <small>(最常用的服务类型，如 音乐播放)</small>
远程服务 <small>(RemoteService)</small>	<ul style="list-style-type: none"> <li>运行在独立进程；</li> <li>服务常驻在后台，不受其他Activity影响</li> </ul>	灵活：服务常驻在后台，不受其他Activity影响	<ul style="list-style-type: none"> <li>消耗资源：单独进程</li> <li>使用AIDL进行IPC复杂</li> </ul>	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 <small>(用户可看到)</small>	服务使用时，需让用户知道 & 进行相关操作，如 音乐播放服务。 <small>(服务被终止时，通知栏的通知也会消失)</small>
后台服务	处于后台的服务 <small>(用户无法看到)</small>	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 <small>(服务被终止时，用户无法知道)</small>

类型	应用场景	备注
本地服务	需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	服务使用时需让用户知道并进行相关操作，如音乐播放	服务被终止时，通
后台服务	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
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

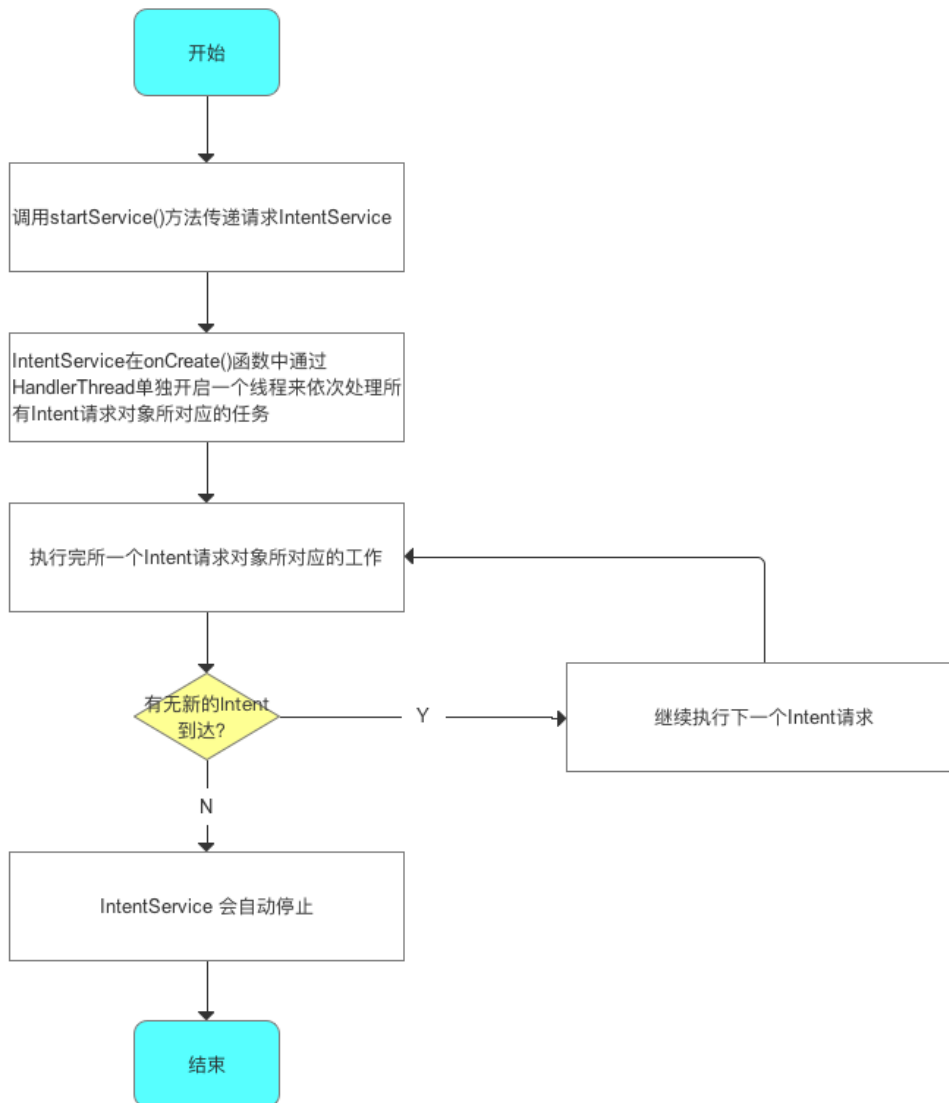
---

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()->> onDestroy()
```

---

- 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：

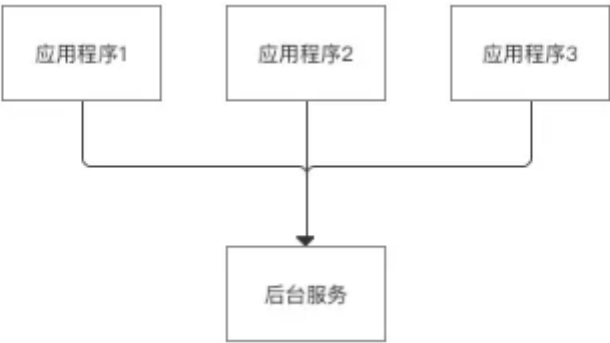
类别	特点	优点	缺点	应用场景
本地服务 (LocalService)	1. 运行在主线程 2. 主进程被终止后，服务也会终止。	1. 节约资源； 2. 通信方便：由于在同一进程因此不需要IPC和AIDL。	限制性大：主进程被终止后，服务也会终止。	需要依附某个进程的服务，如音乐播放
远程服务 (RemoteService)	1. 运行在独立进程； 2. 服务常驻在后台，不受其他 Activity影响	灵活：服务常驻在后台，不受其他 Activity影响	1. 消耗资源：单独进程 2. 使用AIDL进行IPC复杂	系统级别服务

类型	特点	优点	缺点	应用
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 (女
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。

- 在多线程通信中，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 AndroidManifest.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 AndroidManifest.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 AIDL_Service1.Stub.asInterface() 方法获取服务器端返回的 IBinder 对象
            // 将 IBinder 对象转换成了 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
                // 参数与服务端端的 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