- Android training: IntentService
- Android 面试题之Service

Service 之 LocalService

Service 按进程划分可以分为本地服务(LocalService)和远程服务(RemoteService),这篇文章主要来介绍一些 Service 本地服务的知识，主要是以下两点。

- 运行在 UI 线程的普通 Service（两种启动方式）
- 运行在后台线程 IntentService

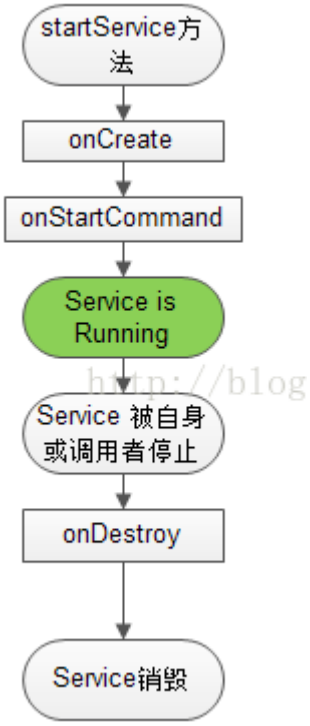
运行在 UI 线程的普通 Service

调用者和 service 在同一个进程并且都运行在 UI 线程，所以不能进行耗时的操作，如果要进行耗时操作，可以在 service 里面创建 Thread 来执行。**任何 Activity 都可以控制同一个 Service， 而系统也只会创建一个对应的 Service 实例。**

第一种启动方式：：通过 `startService` 方式开启服务

- 1 定义一个类继承 `Service`
- 2 `AndroidManifest.xml` 文件中配置 `Service`
- 3 使用 `context` 的 `startService(Intent)` 方法启动 `service`
- 4 不使用时，调用 `stopService(Intent)` 方法停止服务

使用 `startService` 方式启动的生命周期



StartService 方法的特点：

- 1、使用 `startService()` 方法启用服务，调用者与服务之间没有关连，即使调用者退出了，服务仍然运行。
- 2、 `onCreate()` 该方法在服务被创建时调用，**该方法只会被调用一次**，无论调用多少次 `startService()` 或 `bindService()` 方法，服务也只被创建一次（也就是只有一个实例）。
- 3、 `onStartCommand()` 只有采用 `Context.startService()` 方法启动服务时才会回调该方法。该方法在服务开始运行时被调用。多次调用 `startService()` 方法尽管不会多次创建服务，但 `onStartCommand()` 方法会被多次调用。
- 4、无论通过 `startService()` 方法启动了多少次服务，只需要调用一次 `Context.stopService()` 方法结束服务（Service 只有一个实例），服务结束时会调用 `onDestroy()` 方法, 或者调用 `stopSelf` 停止。
- 5、通过 `StartService()` 方法启动的 `service`， `Service` 类中的 `onBind` 方法返回 `null` 即可
- 6、 `startService()` 方法和 `stopService()` 方法都是异步执行的，并且是串行执行，只有当第一次 `onStartCommand()` 执行完毕才会执行第二次的

> start 方式启动者和服务没有交互，一般用于网络上传或者下载，操作完成后，自动停止。

例子： 创建 `StartService` 类

```
public class StartService extends Service{
    private static final String TAG = "StartServiceTag";
    boolean threadRunning = true; // 停止线程的标识
    @Override
    public void onCreate() {
```

```
        super.onCreate();
        Log.i(TAG, "StartService onCreate called");
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        Log.i(TAG, "StartService onBind called");
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i(TAG, "StartService onStartCommand called");
        new  Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    for(int i = 0; i < 100 && threadRunning; i++) {
                        Thread.sleep(1000);
                        Log.i(TAG, "i = " + i);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        threadRunning = false;
        Log.i(TAG, "StartService onDestroy called");
    }
}
```

我们在 `onStartCommand` 方法里启动了一个线程，每隔一秒打印一次 Log。编写一个测试类，启动和停止服务，比较简单。

测试场景一：多次点击 `start` 按钮，`onCreate` 不会重新调用，而 `onStartCommand` 会多次调用。

测试场景二：点击 `start` 按钮后，退出 `Activity`，`Log` 继续打印，说明 `Service` 继续运行。

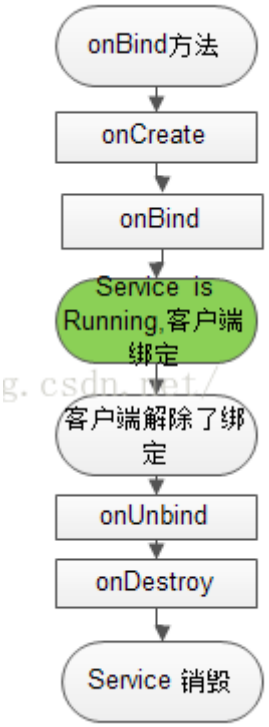
测试场景三：点击 `start`，然后点击 `stop`，Log 打印一部分后停止。

第二种启动方式: 通过 `bind` 的方式启动

- 1，定义一个类继承 `Service`
- 2，在 `AndroidManifest.xml` 文件中注册 `service`
- 3，使用 `context` 的 `bindService(Intent,ServiceConnection,int)` 方法启动 `service`

- 4，不再使用时，调用 `unbindService(ServiceConnection)` 方法停止该服务

使用 bind 方式启动的生命周期



bind 方法的特点

- 1、一个 `activity` 通过 `bindService()` 绑定服务后会调用 `Service` 的 `onCreate()` 和 `onBind()` 方法， `onBind()` 方法会返回一个 `IBander` 对象的实例给 `activity` 的 `ServiceConnection` 的 `onServiceConnected()` 方法中，标志着 `Activity` 与 `Service` 建立了绑定连接，此时当客户端任意一个 `ActivityB` 想要再次 `bindservice` 的时候， `service` 不会再走 `onCreate()` 和 `onBind()` 方法，而是直接拿到 `IBinder` 的实例（这个实例是 `APP` 中所有 `Activity` 共享的），所以在 `ActivityB` 的 `ServiceConnection` 的 `onServiceConnected()` 方法中会直接获得 `IBinder` 的实例,此时如果在 `ActivityB` 中解绑 `Service` ，不会触发 `Service` 的 `unBindService()` 和 `onDestory()` 方法，这两个方法是当 `Service` 没有任何 `activity` 与之绑定的时候才会调用，此时还有 `ActivityA` 与之绑定
- 2、不能多次调用 `unBindService()` ，会抛出异常，所以在调用该方法时需要判断。
- 3、如果 `service` 的 `onBind()` 方法返回 `null` ， `service` 依然会启动，但是没有和 `activity` 绑定上，但但是此时仍然要用 `unBindService` 方法停止服务。

实例: 使用 bind 方式启动

创建 BindService 类

```
public class BindService extends Service {
    private static final String TAG = "BindServiceTag";
    private int count;
    private boolean quit;
```

```
    //定义onBind 方法返回的对象
    private MyBinder binder = new MyBinder();

    public class MyBinder extends Binder {
        //获取Service运行状态,count
        public int getCount() {
            return count;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        //绑定该Service时回调的方法
        Log.i(TAG, "service is binded");
        return binder;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        Log.i(TAG, "service is created");
        //启动一条线程，动态修改count状态值
        new Thread() {
            @Override
            public void run() {
                while (!quit) {
                    try {
                        Thread.sleep(1000);
                        count++;
                    } catch (InterruptedException e) {
                    }
                }
            }
        }.start();
    }

    @Override
    public void onDestroy() {
        //service关闭之前回调
        super.onDestroy();
        this.quit = true;
        Log.i(TAG, "service is destroy");
    }

    @Override
    public boolean onUnbind(Intent intent) {
        //断开连接时回调
        Log.i(TAG, "service onUnbind");
        return true;
    }
}
```

下面是我们的测试类，同时也包含了 start 方式的测试代码

```
public class ServiceTestActivity extends AppCompatActivity {
    private static final String TAG = "BindServiceTag";
    // start 的方式
    private Button startServiceBtn;
    private Button stopServiceBtn;
    // bind 的方式
    private Button bindServiceBtn;
    private Button unbindServiceBtn;
    private Button getStatusBtn;
    private Context mContext;


    //保持所启动Service的IBinder对象
    BindService.MyBinder binder;



    //定义一个ServiceConnection对象
    private ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            Log.i(TAG,"service connected");
            //获取Service的onBind方法所返回的MyBinder对象
            binder = (BindService.MyBinder) iBinder;
        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            Log.i(TAG,"Service disconnected");
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_test);
        init();
    }

    private void init() {
        mContext = ServiceTestActivity.this;
        startServiceBtn = (Button)findViewById(R.id.start_service_btn);
        startServiceBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent intent = new Intent(mContext, StartService.class);
                mContext.startService(intent);
            }
        });
        stopServiceBtn = findViewById(R.id.stop_service_btn);
        stopServiceBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent intent = new Intent(mContext, StartService.class);
                mContext.stopService(intent);
            }
```

```
        });
        bindServiceBtn = (Button)findViewById(R.id.bind_service_btn);
        bindServiceBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // 绑定服务
                Intent bindIntent = new Intent(mContext, BindService.class);
                bindService(bindIntent, connection, Service.BIND_AUTO_CREATE);
            }
        });

        unbindServiceBtn = (Button)findViewById(R.id.unbind_service_btn);
        unbindServiceBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // 解除绑定
                unbindService(connection);
            }
        });
        // 获取 service 状态
        getStatusBtn = (Button)findViewById(R.id.get_status_btn);
        getStatusBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                //获取显示Servcie的count值
                Toast.makeText(mContext, "count value is " +
binder.getCount(),Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

- **bindService(Intent service,ServiceConnection conn,int flags);**
  - `service` :指定要启动的 service,也就是 intent 类型。
  - conn：一个 `ServiceConnection` 对象，该对象用于监听访问者与Service之间的连接 情况。
    - 连接成功调用 `onServiceConnected(ComponentName name, IBinder service)` 方法；异常终止如内存不足，则调用 `onServiceDisconnected(ComponentName name);`
    - 当调用者主动通过 `unBindService()` 方法断开连接时，`onServiceDisconnected(ComponentName name)` 不会调用，调用者退出也不会调用。
  - `Flags` :当 `flags` 不等于 `BIND_AUTO_CREATE` 时， `bindService` 不会自动启动 `service` ，当 `flag` 等于 0 时，调用 `bindService` 前 `service` 未启动，一段时间后 `service` 被启动，此时系统还会尝试之前的 bind 动作。

> 通过 `startService()` 和 `stopService()` 启动关闭服务。**适用于服务和 Activity 之间没有调用交互的情况。**如果**客户端要与服务进行通信**，要使用 `bindService` 方法， `onBind()` 方法必须返回 `Ibinder` 对象。

运行在后台线程 IntentService

如果我们没有为某个操作指定特定的线程，那么大部分的操作任务都会执行在一个叫做 `UI Thread` 的特殊线程中。但是在 `UI` 线程中执行耗时任务可能会影响界面的响应性能，会影响用户体验，甚至可能导致 `ANR` 错误。为了避免这样的问题，我们可以使用 `IntentService` 执行后台任务。

创建后台任务

`IntentService` 可以在后台处理耗时任务并且 `IntentService` 的执行不受 `UI` 生命周期的影响，以此来确保 `AsycnTask` 能够顺利运行。但是 `IntentService` 有下面几个局限性

- 不可以和 `UI` 做交互，为了把它执行的结果体现在 `UI` 上，需要把结果返回给`Activity`。
- 工作任务队列是顺序执行的，如果一个任务正在 `IntentService` 中执行，此时你再发送一个新的任务请求，这个新的任务请求会一直等待直到前面的一个任务执行完毕才开始执行。
- 正在执行的任务无法打断。

下面是一个示例

```
private const val ACTION_ONE = "ramon.lee.fourcomponent.service.action.ONE"

private const val EXTRA_PARAM1 = "PARAM1"
private const val EXTRA_PARAM2 = "PARAM2"

/**
 * An [IntentService] subclass for handling asynchronous task requests in
 * a service on a separate handler thread.
 */
class MyIntentService : IntentService("MyIntentService") {

    override fun onHandleIntent(intent: Intent?) {
        when (intent?.action) {
            ACTION_ONE -> {
                val param1 = intent.getStringExtra(EXTRA_PARAM1)
                val param2 = intent.getStringExtra(EXTRA_PARAM2)
                handleActionOne(param1, param2)
            }
        }
    }

    private fun handleActionOne(param1: String, param2: String) {
        Log.i(TAG, "Current thread: ${Thread.currentThread().name} param1 = $param1
param2 = $param2")
    }

    companion object {
        private const val TAG = "MyIntentService"
        /**
         * Starts this service to perform action Foo with the given parameters. If
         * the service is already performing a task this action will be queued.
         *
         * @see IntentService
         */
        @JvmStatic
        fun handleActionOne(context: Context, param1: String, param2: String) {
            val intent = Intent(context, MyIntentService::class.java).apply {
                action = ACTION_ONE
                putExtra(EXTRA_PARAM1, param1)
                putExtra(EXTRA_PARAM2, param2)
            }
            context.startService(intent)
        }
```

```
    }
}
```

> 需要调用 IntentService("MyIntentService") IntentService 指定一个名字

常见问题

如何提高 Service 的优先级?

- 在 AndroidManifest.xml 中对 `intent-filter` 设置 `android:priority`,最大值是 1000，值越大优先级越高，也适用于广播。

```
<intent-filter android:priority="1000" >
    ...
</intent-filter>
```

- 在 `onCommandStart` 中将服务设置为前台服务,如何启动一个前台服务根据 Android 的版本启动方式有所不同，这里介绍 9 如何启动一个前台服务。

> https://developer.android.com/guide/components/foreground-services

首先需要声明权限，这个权限在声明后会默认赋予

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

然后在 `onStartCommand` 中需要调用 `startForeground`

```
// 启动一个前台服务
val pendingIntent: PendingIntent =
    Intent(this, MainActivity::class.java).let { notificationIntent ->
        PendingIntent.getActivity(this, 0, notificationIntent, 0)
    }

// Create the NotificationChannel
val name = "channel name"
val descriptionText = "description text"
val importance = NotificationManager.IMPORTANCE_DEFAULT
val mChannel = NotificationChannel(CHANNEL_ID, name, importance)
mChannel.description = descriptionText
// Register the channel with the system; you can't change the importance
// or other notification behaviors after this
val notificationManager = getSystemService(NOTIFICATION_SERVICE) as NotificationManager
notificationManager.createNotificationChannel(mChannel)

val notification: Notification = Notification.Builder(this, CHANNEL_ID)
    .setContentTitle("Start foreground service")
    .setContentText("hello foreground")
    .setSmallIcon(R.drawable.ic_launcher_background)
```

```
        .setContentIntent(pendingIntent)
        .setTicker("ticker")
        .build()

    // Notification ID cannot be 0.
    startForeground(ONGOING_NOTIFICATION_ID, notification)
```

移除前台服务,调用 `stopForeground()`. 接收一个 boolean 值，表示是否同时移除 Notification，注意移除后 Service 仍然在运行，如果 Service 停止运行，那么 notification 也会被移除。

- `onStartCommand` 方法，手动返回 `START_STICKY`
- 在 `onDestroy` 方法里发广播重启 `service`。`service +broadcast` 方式，就是当 `service` 走 `ondestory` 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 `service`。（第三方应用或是在setting里-应用-强制停止时，APP进程就直接被干掉了，`onDestroy` 方法都进不来，所以无法保证会执行）
- 监听系统广播判断 `Service` 状态。通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的 `Service` 是否还存活。
- `Application` 加上 `Persistent`, 这种应用会顽固地运行于系统之中，从系统一启动，一直到系统关机。 属性。(https://my.oschina.net/youranhongcha/blog/269591)

Service 的 onStartCommand 方法有几种返回值?各代表什么意思?

https://stackoverflow.com/questions/9093271/start-sticky-and-start-not-sticky

- `START_STICKY`:如果 `service` 进程被 kill 掉,保留 `service` 的状态为开始状态,但不保留递送的 `intent` 对象。随后系统会尝试重新创建 service,由于服务状态为开始状态,所以创建服务后一定会调用 `onStartCommand(Intent,int,int)` 方法。如果在此期间没有任何启动命令被传递到 `service`,那么参数 Intent 将为 `null`。
- `START_NOT_STICKY`：“非粘性的”。使用这个返回值时,如果在执行完 `onStartCommand` 后,服务被异常 kill 掉,系统不会自动重启该服务。
- `START_REDELIVER_INTENT`:重传 Intent。使用这个返回值时,如果在执行完 `onStartCommand` 后,服务被异 常 kill 掉,系统会自动重启该服务,并将 `Intent` 的值传入。
- `START_STICKY_COMPATIBILITY`：`START_STICKY` 的兼容版本,但不保证服务被 kill 后一定能重启。https://www.javaer101.com/en/article/15357575.html

Service 的 onRebind(Intent) 方法什么时候调用?

https://blog.csdn.net/fenggering/article/details/53116311

当我们既想绑定一个 `Service`（为了实现 `Activity` 和 `Service` 的交互）又想在 `Activity` 停止时，`Service` 不会停止，我们可以先 `StartService`，然后再 `BindService()`。

这样的话，当 `Activity` 退出的时候，`Service` 的 `onUnbind()` 方法就会被调用，但 `Sercvice` 并不会停止，然后我们可以再进入 `Activity` 重新绑定该 `Service`，这个时候 `Service` 就会调用 `onRebind()` 方法。`onRebind()` 方法被调用还有个前提是先前的 `onUnbind()` 方法返回值为true