第7章 后台服务

杨刚
中国人民大学

7.1 Service简介

- Service
  - Android系统的服务组件，适用于开发没有用户界面且长时间在后台运行的应用功能
  - 因为手机硬件性能和屏幕尺寸的限制，通常Android系统仅允许一个应用程序处于激活状态并显示在手机屏幕上，而暂停其他处于未激活状态的程序

7.1 Service简介

- Service
  - Android系统需要一种后台服务机制
    - 没有用户界面
    - 其它应用组件能够启动后台服务
    - 当用户切换到另外的应用场景，service能够继续在后台运行
    - 一个组件能够绑定到一个service并且与之交互，例如，一个service可能会处理网络操作，播放音乐，操作文件I/O或者与内容提供者（content provider）交互，但是所有这些活动都是在后台进行

7.1 Service简介

- Service的优势
  - 没有用户界面，更加有利于降低系统资源的消耗
  - Service比Activity具有更高的优先级，因此在系统资源紧张时，Service不会被Android系统优先终止
  - 即使Service被系统终止，在系统资源恢复后Service也将自动恢复运行状态，可以认为Service是在系统中永久运行的组件
  - Service除了可以实现后台服务功能，还可以用于进程间通信（Inter Process Communication，IPC），解决不同Android应用程序进程之间的调用和通讯问题

7.1 Service简介

- Service的分类
  - 本地服务（Local Service ）：用于应用程序内部；
  - 远程服务（Remote Service）：用于android系统内部的应用程序之间。可以定义接口并把接口暴露出来，以便其他应用进行操作。客户端建立到服务对象的连接，并通过那个连接来调用服务。

7.1 Service简介

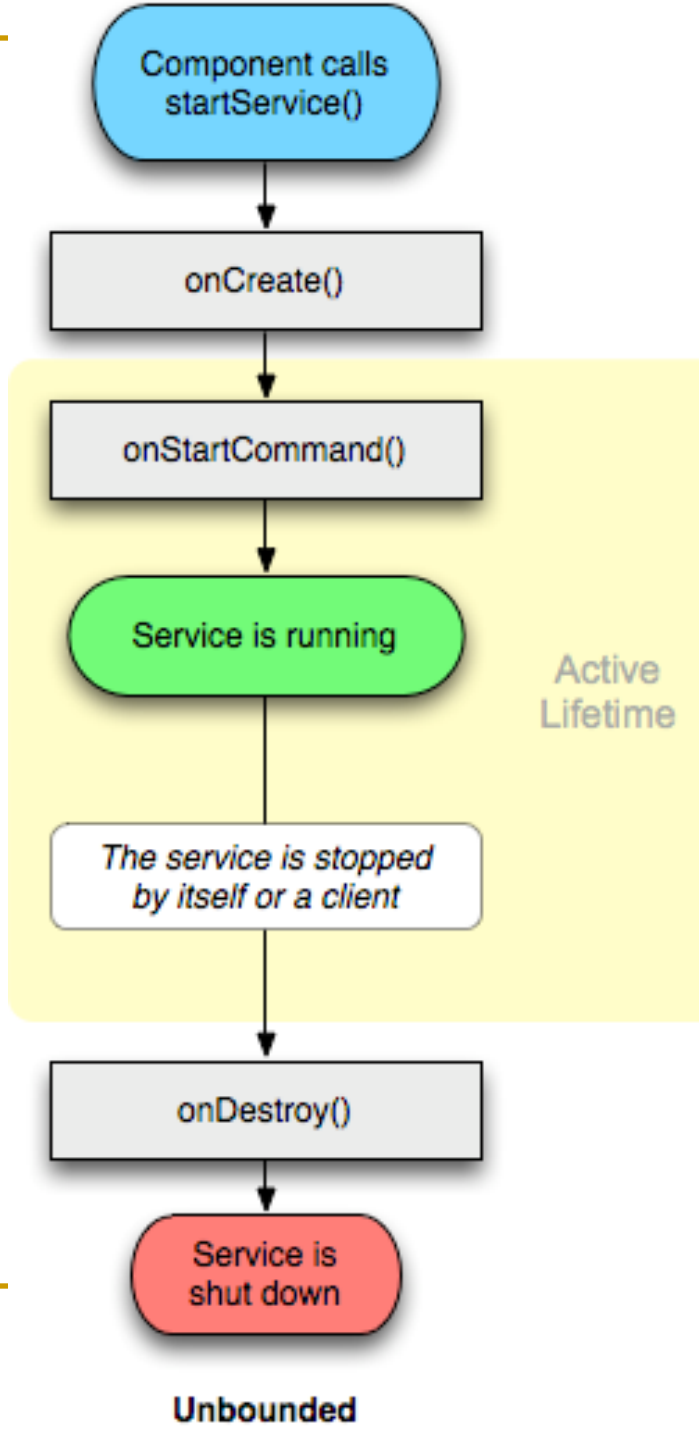
- Service生命周期
  - onCreate()函数
    - Service的生命周期开始，完成Service的初始化工作
  - onStart() 函数
    - 启动线程
  - onDestroy() 函数
    - Service的生命周期结束，释放Service所有占用的资源

7.1 Service简介

- Service生命周期
  - Service生命周期包括
    - 完整生命周期从onCreate()开始到onDestroy()结束，在onCreate()中完成Service的初始化工作，在onDestroy()中释放所有占用的资源
    - 活动生命周期从onStart()开始，但没有与之对应的“停止”函数，因此可以粗略的认为活动生命周期是以onDestroy()标志结束
    - Service的使用方式一般有两种
      - 启动方式
      - 绑定方式

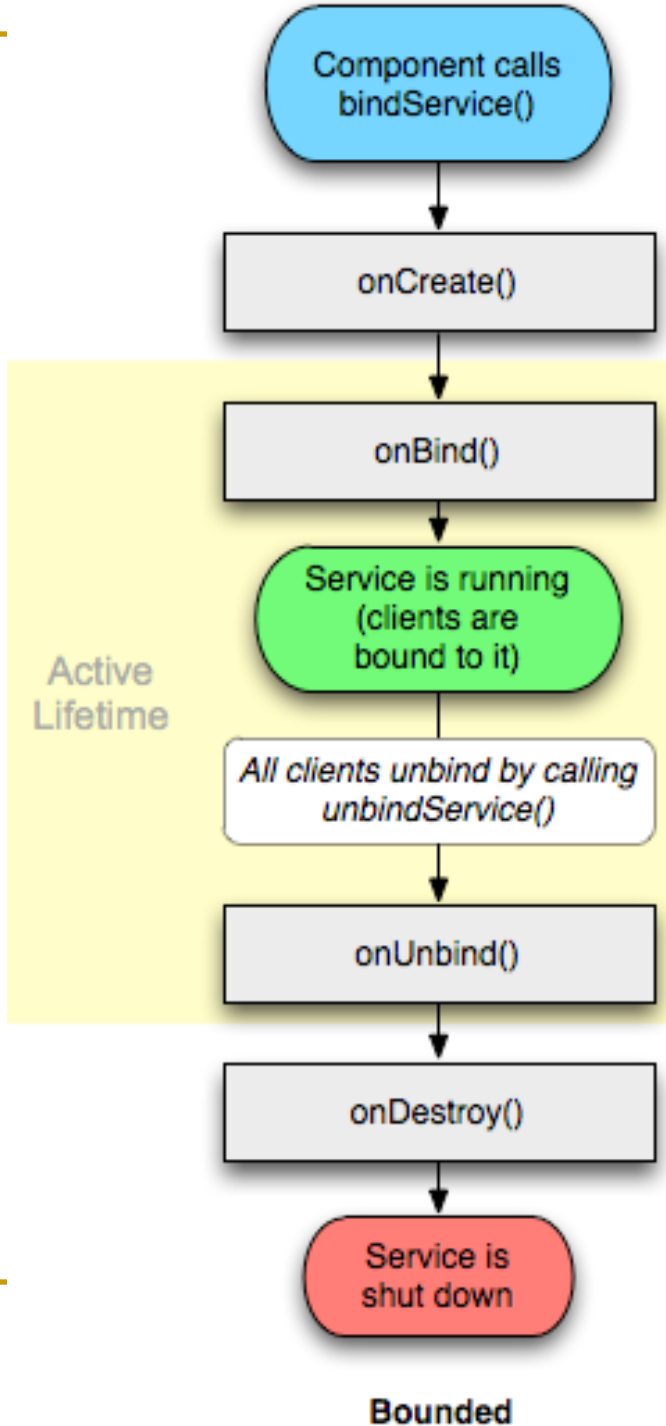
7.1 Service简介

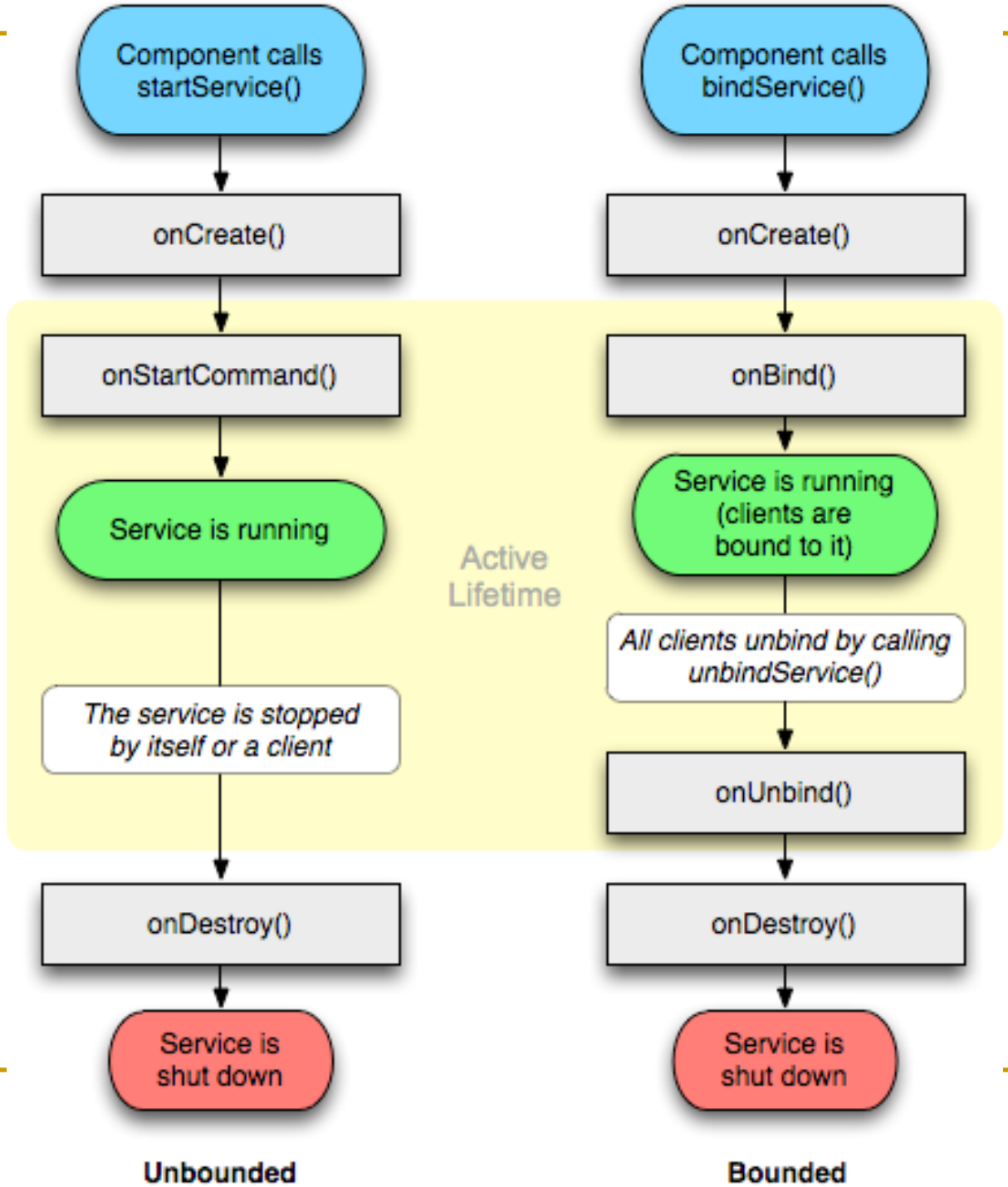
- 启动方式
  - 通过调用Context.startService()启动Service，通过调用Context.stopService()或Service.stopSelf()停止Service。因此，Service一定是由其它的组件启动的，但停止过程可以通过其它组件或自身完成
  - 在启动方式中，启动Service的组件不能够获取到Service的对象实例，因此无法调用Service中的任何函数，也不能够获取到Service中的任何状态和数据信息
  - 能够以启动方式使用的Service，需要具备自管理的能力，而且不需要从通过函数调用获取Service的功能和数据

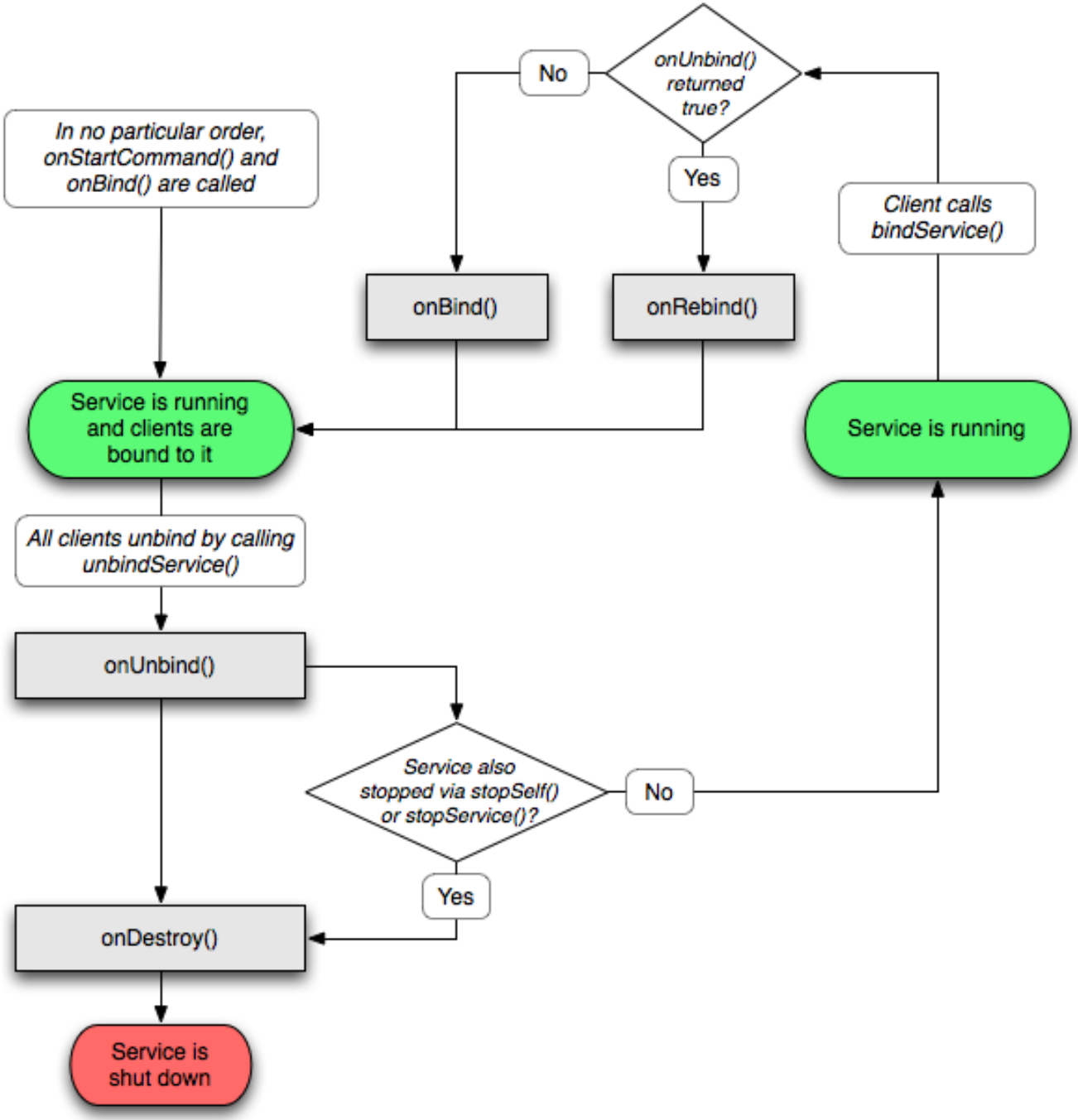


7.1 Service简介

- 绑定方式
  - Service的使用是通过服务链接（Connection）实现的，服务链接能够获取Service的对象实例，因此绑定Service的组件可以调用Service中实现的函数，或直接获取Service中的状态和数据信息
  - 使用Service的组件通过Context.bindService()建立服务链接，通过Context.unbindService()停止服务链接
  - 如果在绑定过程中Service没有启动，Context.bindService()会自动启动Service，而且同一个Service可以绑定多个服务链接，这样可以同时为多个不同的组件提供服务




Component calls startService()
onCreate()
onStartCommand()
Service is running
The service is stopped by itself or a client
onDestroy()
Service is shut down
Unbounded
Component calls bindService()
onCreate()
onBind()
Service is running (clients are bound to it)
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
Service is shut down
Bounded
Active Lifetime


In no particular order, onStartCommand() and onBind() are called
Service is running and clients are bound to it
All clients unbind by calling unbindService()
onUnbind()
Service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service is shut down
onUnbind() returned true?
No
Yes
onBind()
onRebind()
Client calls bindService()
Service is running

7.1 Service简介

- 启动方式和绑定方式的结合
  - 这两种使用方法并不是完全独立的，在某些情况下可以混合使用
    - 以MP3播放器为例，在后台的工作的Service通过Context.startService()启动某个音乐播放，但在播放过程中如果用户需要暂停音乐播放，则需要通过Context.bindService()获取服务链接和Service对象实例，进而通过调用Service对象实例中的函数，暂停音乐播放过程，并保存相关信息
    - 在这种情况下，如果调用Context.stopService()并不能够停止Service，需要在所有的服务链接关闭后，Service才能够真正的停止

7.2 本地服务

- 本地服务的调用者和服务都在同一个程序中，是不需要跨进程就可以实现服务的调用
- 本地服务涉及服务的建立、启动和停止，服务的绑定和取消绑定，以及如何在线程中实现服务
- 7.2.1 服务管理
  - 服务管理主要指服务的启动和停止
  - Service是一段在后台运行、没有用户界面的代码

7.2 本地服务

- 7.2.1 服务管理
  - 为了使Service工作，一般需要重载onCreate()、onStart()和onDestroy()。Android系统在创建Service时，会自动调用onCreate()，用户一般在onCreate()完成必要的初始化工作，例如创建线程、建立数据库链接等
  - 在Service关闭前，系统会自动调用onDestroy()函数释放所有占用的资源。通过Context.startService(Intent)启动Service，onStart()则会被调用，重要的参数通过参数Intent传递给Service
  - 当然，不是所有的Service都需要重载这三个函数，可以根据实际情况选择需要重载的函数

7.2 本地服务

- 7.2.1 服务管理
  - 在完成Service代码和在AndroidManifest.xml文件中注册后，下面来说明如何启动和停止Service。有两种方法启动Service，显式启动和隐式启动
  - 显式启动需要在Intent中指明Service所在的类，并调用startService(Intent)启动Service，示例代码如下

```
1  final Intent serviceIntent = new Intent(this, RandomService.class);
2  startService(serviceIntent);
```

    - Intent指明了启动的Service所在类为RandomService

7.2 本地服务

- 7.2.1 服务管理
  - 隐式启动则需要在注册Service时，声明Intent-filter的action属性

```
1   <service android:name=".RandomService">
2       <intent-filter>
3           <action android:name="com.example.RandomService" />
4       </intent-filter>
5   </service>
```

7.2 本地服务

- 7.2.1 服务管理
  - 在隐式启动Service时，需要设置Intent的action属性，这样则可以在不声明Service所在类的情况下启动服务。隐式启动的代码如下

```
1  final Intent serviceIntent = new Intent();
2  serviceIntent.setAction("com.example.RandomService");
```

  - 如果Service和调用服务的组件在同一个应用程序中，可以使用显式启动或隐式启动，显式启动更加易于使用，且代码简洁。但如果服务和调用服务的组件在不同的应用程序中，则只能使用隐式启动

7.2 本地服务

- 7.2.1 服务管理
  - 无论是显式启动还是隐式启动，停止Service的方法都是相同的，将启动Service的Intent传递给stopService(Intent)函数即可，示例代码如下

```
1  stopService(serviceIntent);
```

  - 在首次调用startService(Intent)函数启动Service后，系统会先后调用onCreate()和onStart()
  - 如果是第二次调用startService(Intent)函数，系统则仅调用onStart()，而不再调用onCreate()
  - 在调用stopService(Intent)函数停止Service时，系统会调用onDestroy()
  - 无论调用过多少次startService(Intent)，在调用stopService(Intent)函数时，系统仅调用一次onDestroy()

7.2 本地服务

7.2.2 使用线程

- 较好的解决方法是将耗时的处理过程转移到子线程上，这样可以缩短主线程的事件处理时间，从而避免用户界面长时间失去响应
- “耗时的处理过程”一般指复杂运算过程、大量的文件操作、存在延时的网络通讯和数据库操作等等（常用的如查询升级信息）
- 线程是独立的程序单元，多个线程可以并行工作。在多处理器系统中，每个中央处理器（CPU）单独运行一个线程，因此线程是并行工作的
- 但在单处理器系统中，处理器会给每个线程一小段时间，在这个时间内线程是被执行的，然后处理器执行下一个线程，这样就产生了线程并行运行的假象

7.2 本地服务

- 7.2.2 使用线程
  - 无论线程是否真的并行工作，在宏观上可以认为子线程是独立于主线程的，且能与主线程并行工作的程序单元
  - 在Java语言中，建立和使用线程比较简单，首先需要实现Java的Runnable接口，并重载run()函数，在run()中放置代码的主体部分

```
1   private Runnable backgroudWork = new Runnable(){
2         @Override
3         public void run() {
4            //过程代码
5         }
6      };
```

7.2 本地服务

7.2.2 使用线程

- 当线程在run()方法返回后，线程就自动终止了
- 当然，也可以调用stop()在外部终止线程，但这种方法并不推荐使用，因为这方法并不安全，有一定可能性会产生异常
- 最好的方法是通知线程自行终止，一般调用interrupt()方法通告线程准备终止，线程会释放它正在使用的资源，在完成所有的清理工作后自行关闭

```
1 workThread.interrupt();
```

- 其实interrupt()方法并不能直接终止线程，仅是改变了线程内部的一个布尔值，run()方法能够检测到这个布尔值的改变，从而在适当的时候释放资源和终止线程

7.2 本地服务

- 7.2.2 使用线程
  - 使用Handler更新用户界面
    - Handler允许将Runnable对象发送到线程的消息队列中，每个Handler实例绑定到一个单独的线程和消息队列上
    - 当用户建立一个新的Handler实例，通过post()方法将Runnable对象从后台线程发送给GUI线程的消息队列，当Runnable对象通过消息队列后，这个Runnable对象将被运行

7.2 本地服务

- 7.2.3 服务绑定
  - 以绑定方式使用Service，能够获取到Service实例，不仅能够正常启动Service，还能够调用Service中的公有方法和属性
  - 为了使Service支持绑定，需要在Service类中重载onBind()方法，并在onBind()方法中返回Service实例

7.2 本地服务

- 7.2.3 服务绑定
  - 当Service被绑定时，系统会调用onBind()函数，通过onBind()函数的返回值，将Service实例返回给调用者
  - onBind()函数的返回值必须符合IBinder接口
  - IBinder是用于进程内部和进程间过程调用的轻量级接口，定义了与远程对象交互的抽象协议，使用时通过继承Binder的方法来实现

7.2 本地服务

- 7.2.3 服务绑定
  - 调用者通过bindService()函数绑定服务
    - 调用者通过bindService()函数绑定服务，并在第1个参数中将Intent传递给bindService()函数，声明需要启动的Service
    - 第3个参数Context.BIND_AUTO_CREATE表明只要绑定存在，就自动建立Service
    - 同时也告知Android系统，这个Service的重要程度与调用者相同，除非考虑终止调用者，否则不要关闭这个Service

```
1    final Intent serviceIntent = new Intent(this,MathService.class);
2    bindService(serviceIntent,mConnection,Context.BIND_AUTO_CREATE);
```

7.2 本地服务

- 7.2.3 服务绑定
  - bindService()函数的第2个参数是ServiceConnnection
    - 当绑定成功后，系统将调用ServiceConnnection的onServiceConnected()方法
    - 当绑定意外断开后，系统将调用ServiceConnnection中的onServiceDisconnected方法
    - 因此，以绑定方式使用Service，调用者需要声明一个ServiceConnnection，并重载内部的onServiceConnected()方法和onServiceDisconnected方法

7.2 本地服务

- 7.2.3 服务绑定
  - 取消绑定仅需要使用unbindService()方法，并将ServiceConnnection传递给unbindService()方法
  - 但需要注意的是，unbindService()方法成功后，系统并不会调用onServiceConnected()，因为onServiceConnected()仅在意外断开绑定时才被调用

```
1  unbindService(mConnection);
```

Service实例: 微信抢红包插件

- 实现的功能
  - 可以抢屏幕上显示的所有红包，同类插件往往只能获取最新的一个红包。
  - 智能跳过已经戳过的红包，避免频繁点击影响正常使用。
  - 红包日志功能，方便查看抢过的红包内容。
  - 性能优化，感受不到插件的存在，不影响日常聊天。
  - 由于这是一份教学代码，项目的文档和注释都比较完整，代码适合阅读。
- 演示

实现原理

- 1. 抢红包流程的逻辑控制
  - 这个插件通过一个Stage类来记录当前对应的阶段。
  - Stage类被设计成单例并惰性实例化，因为一个Service不需要也不应该处在不同的阶段。对外暴露阶段常量和entering和getCurrentStage两个方法，分别记录和获取当前的阶段。

实现原理

```
public class Stage {

    private static Stage stageInstance;

    public static final int FETCHING_STAGE = 0, OPENING_STAGE = 1, FETCHED_STAGE = 2, O

    private int currentStage = FETCHED_STAGE;

    private Stage() {}

    public static Stage getInstance() {
        if (stageInstance == null) stageInstance = new Stage();
        return stageInstance;
    }

    public void entering(int _stage) {
        stageInstance.currentStage = _stage;
    }

    public int getCurrentStage() {
        return stageInstance.currentStage;
    }
}
```

实现原理

阶段	说明
FETCHING_STAGE	正在读取屏幕上的红包，此时不应有别的操作
FETCHED_STAGE	已经结束一个FETCH阶段，屏幕上的红包都已加入待抢队列
OPENING_STAGE	正在拆红包，此时不应有别的操作
OPENED_STAGE	红包成功抢到，进入红包详情页面

- 程序以**FETCHED_STAGE** 开始，将屏幕上的红包加入待抢队列：

→FETCHED_STAGE → FETCHING_STAGE → FETCHED_STAGE →

- 处理待抢队列中的红包：

→[CLICK] → OPENING_STAGE → [CLICK] → OPENED_STAGE → [BACK] → FETCHED_STAGE → （抢到）

→ [CLICK] → OPENING_STAGE → [BACK] → FETCHED_STAGE → （没抢到）

实现原理

- 根据阶段选择不同的入口

```
switch (Stage.getInstance().getCurrentStage()) {
    case Stage.OPENING_STAGE:
        // .......
        Stage.getInstance().entering(Stage.FETCHED_STAGE);
        performGlobalAction(GLOBAL_ACTION_BACK);
        break;
    case Stage.OPENED_STAGE:
        Stage.getInstance().entering(Stage.FETCHED_STAGE);
        performGlobalAction(GLOBAL_ACTION_BACK);
        break;
    case Stage.FETCHED_STAGE:
        if (nodesToFetch.size() > 0) {
            AccessibilityNodeInfo node = nodesToFetch.remove(nodesToFetch.size() - 1);
            if (node.getParent() != null) {
                // .......
                Stage.getInstance().entering(Stage.OPENING_STAGE);
                node.getParent().performAction(AccessibilityNodeInfo.ACTION_CLICK);
            }
            return;
        }
        Stage.getInstance().entering(Stage.FETCHING_STAGE);
        fetchHongbao(nodeInfo);
        Stage.getInstance().entering(Stage.FETCHED_STAGE);
        break;
}
```

实现原理

- 屏幕内容检测和自动化点击的实现
  - 和其他插件一样，这里使用的是Android API提供的AccessibilityService。这个类位于android.accessibilityservice包内，该包中的类用于开发无障碍服务，提供代替或增强的用户反馈。
  - AccessibilityService 服务在后台运行，等待系统在发生 AccessibilityEvent 事件时回调。这些事件指的是用户界面上显示发生的状态变化， 比如焦点变更、按钮按下等等。服务可以请求“查询当前窗口中内容”的能力。 开发辅助服务需要继承该类并实现其抽象方法。

实现原理

- 配置AccessibilityService

在这个例子中，我们需要监听的事件是当红包来或者滑动屏幕时引起的屏幕内容变化，和点开红包时窗体状态的变化，因此我们只需要在配置XML的accessibility-service标签中加入一条

```
android:accessibilityEventTypes="typeWindowStateChanged|typeWindowContentChanged"
```

或在onAccessibilityEvent回调函数中对事件进行一次类型判断

```
final int eventType = event.getEventType();
if (eventType == AccessibilityEvent.TYPE_WINDOW_STATE_CHANGED
     || eventType == AccessibilityEvent.TYPE_WINDOW_CONTENT_CHANGED) {
         // ...
}
```

除此之外，由于我们只监听微信，还需要指定微信的包名

```
android:packageNames="com.tencent.mm"
```

为了获取窗口内容，我们还需要指定

```
android:canRetrieveWindowContent="true"
```

实现原理

- 获取红包所在的节点

```
AccessibilityNodeInfo nodeInfo = event.getSource();

AccessibilityNodeInfo nodeInfo = getRootInActiveWindow();
```

  - 这里返回的AccessibilityNodeInfo是窗体内容的节点，包含节点在屏幕上的位置、文本描述、子节点Id、能否点击等信息。从AccessibilityService的角度来看，窗体上的内容表示为辅助节点树，虽然和视图的结构不一定一一对应。换句话说，自定义的视图可以自己描述上面的辅助节点信息。当辅助节点传递给AccessibilityService之后就不可更改了，如果强行调用引起状态变化的方法会报错。
  - 在聊天页面，每个红包上面都有“领取红包”这几个字，我们把它作为识别红包的依据。因此，如果你收到了这四个字的文本消息，那插件会做出误判，并没有什么办法可以解决。

实现原理

- 获取红包所在的节点
  - AccessibilityNodeInfo的API中有一个findAccessibilityNodeInfosByText方法允许我们通过文本来搜索界面中的节点。匹配是大小写敏感的，它会从遍历树的根节点开始查找。API文档中特别指出，为了防止创建大量实例，节点回收是调用者的责任，这一点会在接下来的部分中讲到。

```
List<AccessibilityNodeInfo> node1 = nodeInfo.findAccessibilityNodeInfosByText("领取红包");
```

实现原理

- 对节点进行操作

AccessibilityNodeInfo同样暴露了一个API——performAction来对节点进行点击或者其他操作。处于安全性考虑，只有这个操作来自AccessibilityService时才会被执行。

```
nodeInfo.performAction(AccessibilityNodeInfo.ACTION_CLICK);
```

不过，我们在调试时发现"领取红包"的mClickable属性为false，说明点击的监听加在它父辈的节点上。通过getParent获取父节点，这个节点是可以点击的。

我们还需要全局的返回操作，最方便的办法就是performGlobalAction，不过注意这个方法是API 16后才有的。

```
performGlobalAction(GLOBAL_ACTION_BACK);
```

实现原理

- 获取屏幕上的所有红包
  - 和其他插件最大的区别是，这个插件的逻辑是获取屏幕上所有的红包节点，去掉已经获取过的之后，将待抢红包加入队列，再将队列中的红包一个个打开。

实现原理

- 判断红包节点是否已被抢过
  - 实现这一点是编写时最大的障碍。对于一般的Java对象实例来说，除非被GC回收，实例的Id都不会变化。我最初的想法是通过正则表达式匹配下面的十六进制对象id来表示一个红包。

```
android.view.accessibility.AccessibilityNodeInfo@2a5a7c; .......
```

  - 但在测试中，队列中的部分红包没有被戳开。进一步观察发现，新的红包节点和旧的红包节点id出现了重复，且出现概率较大。由于GC日志正常，我推测存在一个实例池的设计。获取当前窗体节点树的时候，从一个可重用的实例池中获取一个AccessibilityNodeInfo实例。在接下来的获取时，仍然从实例池中获取节点实例，这时可能会重用之前的实例。这样的设计是有好处的，可以防止每次返回都创建大量的实例，影响性能。AccessibilityNodeProvider的源码表明了这样的设计。
  - 也就是说，为了标识一个唯一的红包，只用实例id是不充分的。这个插件插件采用的是红包内容+节点实例id的hash来标记。因为同一屏下，同一个节点树下的节点id是一定不会重复的，滑动屏幕后新红包的内容和节点id同时重复的概率已经大大减小。更改标识策略后，实测中几乎没有出现误判。

实现原理

- 将新出现的红包加入待抢队列

我们维护了两个列表，分别记录待抢红包和抢过的红包标识。

```
private List<AccessibilityNodeInfo> nodesToFetch = new ArrayList<>();

private List<String> fetchedIdentifiers = new ArrayList<>();
```

在每次读取聊天屏幕的时候，会检查这个红包是否在fetchedIdentifiers队列中，如果没有，则加入nodesToFetch队列。

```
for (AccessibilityNodeInfo cellNode : fetchNodes) {
    String id = getHongbaoHash(cellNode);
    /* 如果节点没有被回收且该红包没有抢过 */
    if (id != null && !fetchedIdentifiers.contains(id)) {
        nodesToFetch.add(cellNode);
    }
}
```

实现原理

- 打开队列中的红包
  - 通过红包打开后显示的文本判断这个红包是否可以抢，进行接下来的操作。

实现原理

- 判断红包节点是否被重用
  - 这也是实现时的一个坑。前面提到了实例池的设计，当我们把红包们加入待抢队列，戳完一个红包再回来时，队列中的其他红包节点可能已被回收重用，如果再去点击这个节点，显然没有什么卵用。为了解决这个问题，我们只能退而求其次，在点开前做一次检查。如果发现被重用了，就舍弃这个节点，在下一轮fetch的阶段重新加入待抢队列。确认没有重用立即打开，并把节点hash加入fetchedIdentifiers队列。这里如果node失效getHongbaoHash会返回null。

```
AccessibilityNodeInfo node = nodesToFetch.remove(nodesToFetch.size() - 1);
if (node.getParent() != null) {
    String id = getHongbaoHash(node);
    if (id == null) return;
    fetchedIdentifiers.add(id);
    Stage.getInstance().entering(Stage.OPENING_STAGE);
    node.getParent().performAction(AccessibilityNodeInfo.ACTION_CLICK);
}
```

实现原理

- 判断红包节点是否被重用
  - 可以看出这并不是很有效率的解决方案。在实测中，有时队列中中红包失效后被舍弃，但没有新的AccessibilityEvent发生，接下来的操作都被挂起了。在戳过较多红包之后，这种情况表现得尤为明显，必须要显式地改变窗体内容才能解决。

```
AccessibilityNodeInfo node = nodesToFetch.remove(nodesToFetch.size() - 1);
if (node.getParent() != null) {
    String id = getHongbaoHash(node);
    if (id == null) return;
    fetchedIdentifiers.add(id);
    Stage.getInstance().entering(Stage.OPENING_STAGE);
    node.getParent().performAction(AccessibilityNodeInfo.ACTION_CLICK);
}
```

实现原理

- 根据红包类型选择操作
  - 红包被戳开前会进行查重。戳开后如果界面上出现了“拆红包”几个字，说明红包还没有被别人抢走，立刻点击“拆红包”并将stage标记为OPENED_STAGE。
  - 此时，另三种情况表明抢红包失败了，直接返回，接下来状态会被标记为FETCHED_STAGE。
    - “过期”，说明红包超过有效时间
    - “手慢了”，说明红包发完但没抢到
    - “红包详情”，说明你已经抢到过

实现原理

- 防止加载红包时返回
  - 戳开红包和红包加载完之间有一个“正在加载”的过渡动画，会触发onAccessibilityEvent回调方法。如果在加载完之前判断，上述文本都找不到，默认被标记为FETCHE_STAGE并触发返回。因此，我们要在返回前特殊判定这种情形。
  - 我们引入了TTL来记录尝试次数，并返回错误值-1。如果到达MAX_TTL时红包还没有加载出来就舍弃这个红包。

```
Stage.getInstance().entering(Stage.OPENING_STAGE);
ttl += 1;
return -1;
```

项目文档

- https://github.com/geeeeeeeeek/WeChatLuckyMoney
- 详细的源码和文档。
- 欢迎Star、Fork、Follow。