Service

2018年4月23日 8:23

服务是Android中实现程序后台运行的解决方案，适合去执行那些不需要和用户交互而且还要求长期运行的任务。（服务的运行不依赖于任何用户界面）
服务是依赖于创建服务时所在的应用程序进程，所以当某个应用程序进程被杀掉时，所有依赖于该进程的服务也会停止服务

1、线程

……

①定义：新建一个类继承自Thread，然后重写父类的run（）方法，在里面编写耗时逻辑

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

启动：new出MyThread的实例，然后调用start（）方法

```
new MyThread（）.start（）；
```

……

②定义：实现Runnable接口的方式来定义线程

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

启动： MyThread myThread = new MyThread（）；
new Thread（myThread）.start（）；

……

③使用匿名类（常用）

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

……

Android中更新应用程序里的UI元素，必须在主线程中进行，否则会异常！！！

>>> 异步消息处理机制

4个部分组成：Message、Handler、MessageQueue和Looper

- ✓ Message
  在线程之间传递的消息，可以在内部携带少量的信息，用于在不同线程之间交换数据。（arg1和arg2携带一些整型数据，obj字段携带一个Object对象）
- ✓ Handler
  处理者，用于发送和处理消息。发送消息一般是使用Handler的sendMessage（）方法，而发出的消息经过一系列辗转处理后，最终 会传递到Handler的handleMessage（）方法中
- ✓ MessageQueue
  消息队列，用于存放所有通过Handler发送的消息。这部分消息会一直存在于消息队列中，等待被处理。每个线程中只会有一个MessageQueue对象
- ✓ Looper
  Looper是每个线程中的MessageQueue的管家，调用Looper的loop（）方法后，就会进入到一个无限循环当中，然后每当发现MessageQueue中存在一条消息，就会将它取出，并传递到Handler的handleMessage（）方法中，每个线程也只会有一个Looper对象

流程：

1) 在主线程当中创建一个Handler对象，并重写handMessage（）方法
2) 当子线程中需要进行UI操作时，创建一个Message对象，并通过Handler将这条消息发送出去
3) 消息被添加到MessageQueue的队列中等待被处理，Looper也会一直尝试从MessageQueue中取出待处理消息
4) 分发回Handler的handleMessage（）方法中

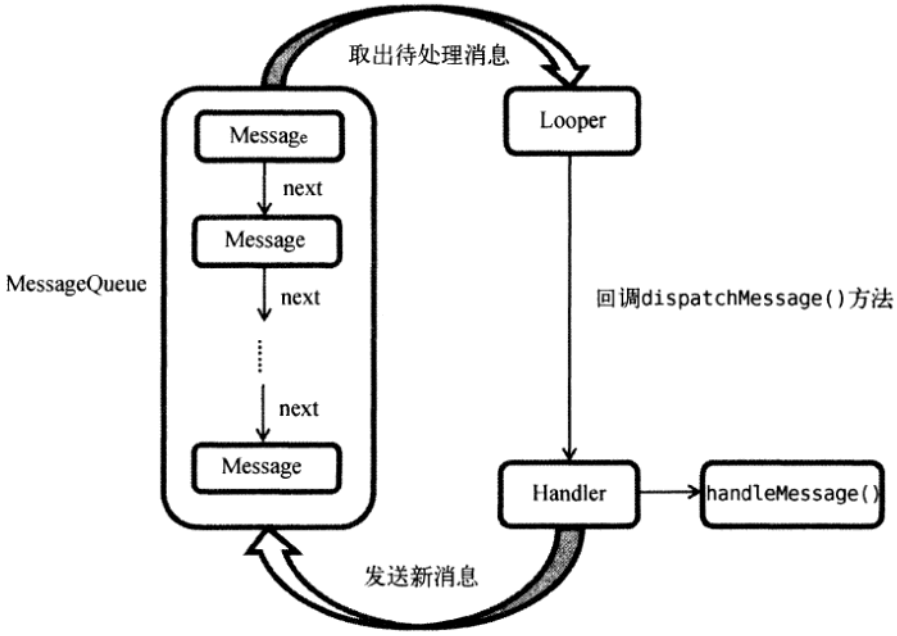

图 10.4　异步消息处理机制流程示意图

>>>使用AsyncTask（从子线程切换到主线程）

AsyncTask是一个抽象类，要创建一个子类去继承，可以指定3个泛型参数

- Params：在执行AsyncTask时需要传入的参数，可用于在后台任务中使用
- Progress：后台任务执行时，需要在界面上显示当前的进度，使用指定的泛型作为进度单位
- Result：任务执行完毕后，对寄过进行返回，使用指定的泛型作为返回值类型

class DownloadTask extends AsyncTask<void，Integer，Boolean>{ … }

重写AsyncTask中方法完成对任务的定制：

- onPreExecute（）：在后台任务开始执行之前调用，用于进行一些界面上的初始化（显示一个进度条对话框）

- doInBackground(Paeams...)：处理所有的耗时任务
- onProgressUpdate（Progress...）：对界面元素进行相应的更新
- onPostExecute（Result）：提醒任务执行的结果，以及关闭掉进度条对话框等

使用AsyncTask，在doInBackground（）方法中执行具体的耗时任务，在onProgressUpdate（）方法中进行UI操作，在onPostExecute（）方法中执行一些任务的收尾工作

启动任务： new DownloadTask（）.execute（）；

2、服务

定义服务，重写一些方法

```
@Override
    public void onCreate(){   //在服务第一次创建的时候调用
        super.onCreate();
    }
```

//服务一旦启动就去执行某个动作，逻辑写在onStartCommand（）方法里

```
    @Override
    public int onStartCommand(Intent intent,int flag,int startId){  //在每次服务启动的时候调用
        return super.onStartCommand(intent,flag,startId);
    }

    @Override
    public void onDestroy(){    //在服务销毁的时候调用
        super.onDestroy();
    }
```

启动及停止服务

```
@Override
    public void onClick(View view) {
        switch (view.getId()){
            case R.id.start_service:
                Intent startIntent = new Intent(this,MyService.class);
                startService(startIntent);  //启动服务
                break;
            case R.id.stop_service:
                Intent stopIntent = new Intent(this,MyService.class);
                stopService(stopIntent); //停止服务
                break;
            default:
                break;
        }
    }
```

在任何一个位置调用stopSelf（）方法就能让这个服务停止下来

活动和服务进行通信

活动指挥服务，用onBind（）方法，创建一个专门的Binder对象

在活动中根据具体的场景来调用DownloadBinder中的任何public（）方法，即实现了指挥服务干什么，服务就去干什么的功能（DownloadBinder继承Binder类）

```
private MyService.DownloadBinder downloadBinder;
private ServiceConnection connection = new ServiceConnection() { //匿名类，在活动与服务成功绑定以及解除绑定的时候调用
//MainActivity.java
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            downloadBinder = (MyService.DownloadBinder) iBinder;  //得到downloadBinder实例
            downloadBinder.startDownload();
            downloadBinder.getProgress();
        }

        @Override
        public void onServiceDisconnected(ComponentName componentName) {

        }
    };
//MainActivity.java
case R.id.bind_service:
        Intent bindIntent = new Intent(this,MyService.class);
        bindService(bindIntent,connection,BIND_AUTO_CREATE); //绑定服务，第一个参数是Intent对象，第二个参数是ServiceConnection的实例，第三个参数是一个标志位，BIND_AUTO_CREATE表示在活动和服务进行绑定后自动创建服务
       break;
case R.id.unbind_service:
       Intent unbindService = new Intent(this,MyService.class);
       unbindService(connection); //解绑服务
       break;
```

3、生命周期

- ✧ 调用Context的startService（）方法，相应的服务就会启动起来，并回调onStartCommand（）方法，服务一旦启动，就会一直处于运行状态。不过每个服务都只有存在一个实例，所以不管调用多少startService（）方法，只需调用一次stop Service（）或stopSelf（），服务即停止

- ✧ 调用Context的bindService（）来获取一个服务的持久连接，这时就会回调服务中的onBind（）方法，若服务未创建过，则onCreate（）方法会先于onBind（）方法执行。之后，调用方获取到onBind（）方法里返回的IBinder对象的实例，这样则可和服务进行通信，且只要调用方和服务之间的连接没有断开，服务就会一直保持运行状态。

- ✧ 一个服务只要被启动或者被绑定了之后，就会一直处于运行状态，必须要让以上两种条件同时不满足，服务才能被销毁。所以，这种情况下要同时调用stopService（）和unbingdService（）方法，onDestroy（）方法才会执行。

4、小扩展

① 前台服务（用法类似于通知）

前台服务和普通服务最大的区别在于，它会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，类似于通知的效果。

```
Intent intent = new Intent(this,MainActivity.class);
        PendingIntent pi = PendingIntent.getActivity(this,0,intent,0);
        Notification notification = new NotificationCompat.Builder(this)  //构建notification实例
                .setContentTitle("This is content title")
                .setContentText("This is content text")
                .setWhen(System.currentTimeMillis())
                .setSmallIcon(R.mipmap.ic_launcher)
                .setLargeIcon(BitmapFactory.decodeResource(getResources(),R.mipmap.ic_launcher))
                .setContentIntent(pi)
                .build();
        startForeground(1,notification); //显示通知，让MyService变成一个前台服务，并在系统状态栏显示出来
```

② 使用IntentService

服务在执行完毕后自动停止

```
//MyIntentService.java
  @Override
    protected void onHandleIntent(@Nullable Intent intent) {  //可以处理一些具体的逻辑（在子线程中运行的）
        //打印当前线程的id
        Log.d("MyIntentService","Thread id is " + Thread.currentThread().getId());
    }
//MainActivity.java
case R.id.start_intent_service:
                //打印主线程的id
                Log.d("MainActivity","Thread id is" + Thread.currentThread().getId());
                Intent intentService = new Intent(this,MyIntentService.class);
                startService(intentService);
                break;
//AndroidManifest.xml
<service android:name=".MyIntentService"/>
```