



#10 服务与多线程





什么是Service

Service(服务)是一种可以在后台执行长时间运行操作而没有用户界面的应用组件。

服务可由其他应用组件启动（如Activity），服务一旦被启动将在后台一直运行，即使启动服务的组件（Activity）已销毁也不受影响。

例如：退出播放器界面之后，继续在后台播放音乐
有没有其他例子？





什么是Service

Service可以理解为没有专属界面（UI）的Activity。通过Service可以使程序在退出之后仍然能够对事件或用户操作做出反应，或者在后台继续运行某些程序功能。

Android赋予Services比处于非前台的Activities有更高的优先级，所以它们的进程不会轻易被系统杀掉。

在某些极端的情况下（例如为前台Activity提供资源），Service可能会被杀掉，但是只要有足够的资源，系统会自动重启Service。





什么是Service

不同的是Activity拥有前台运行的用户界面，而Service不能自己运行，需要通过某个Activity或者其他Context对象来调用。

Service是由其他Service，Activity或者Broadcast Receiver开始，停止和控制。

Service在后台运行，它不能与用户直接进行交互。在默认情况下，**Service运行在应用程序进程的主线程之中。**





如何启动Service

- 模式一：通过startService启动
- 模式二：通过bindService启动





模式一：通过startService启动

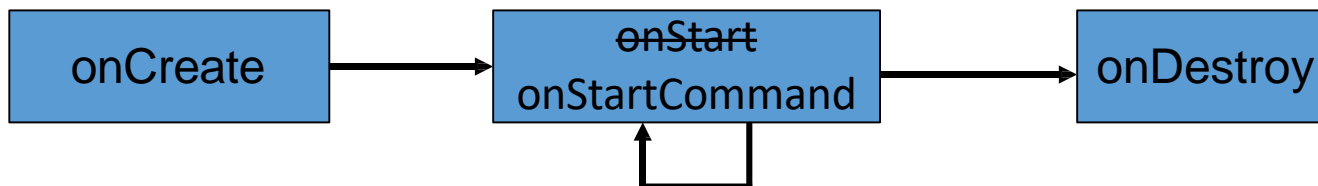
- 用于实现应用程序自己的一些耗时任务，比如查询升级信息，并不占用应用程序比如Activity所属线程，而是单开线程后台执行，这样用户体验比较好。
- startService()启动和stopService()关闭服务，Service与访问者之间基本不存在太多关联，因此Service和访问者之间无法通讯和数据交换。





模式一：通过startService启动

- 通过startService启动的Service的生命周期状态（一旦启动，各自无关）



- 调用Context.startService()启动，
调用Context.stopService()结束，
调用Service.stopSelf() 或 Service.stopSelfResult()停止
- 不论调用了多少次startService()方法，需要调用一次stopService()来停止服务





模式一：通过startService启动

开启Service

- 在Activity中可以通过startService(Intent)开启一个Service。与Activity跳转类似。

```
Intent intent = new Intent(this, MyService.class);
```

```
startService(intent);
```

- 其中MyService类是开发者自定义的继承Service的子类。
- 当第一次启动Service时，先后调用了onCreate(),onStart()方法。当停止Service时，则执行onDestroy()方法。
- 若Service已经启动，当再次启动Service时，不会再执行onCreate()方法，而是直接执行onStart()方法。





模式一：通过stopService停止

关闭Service

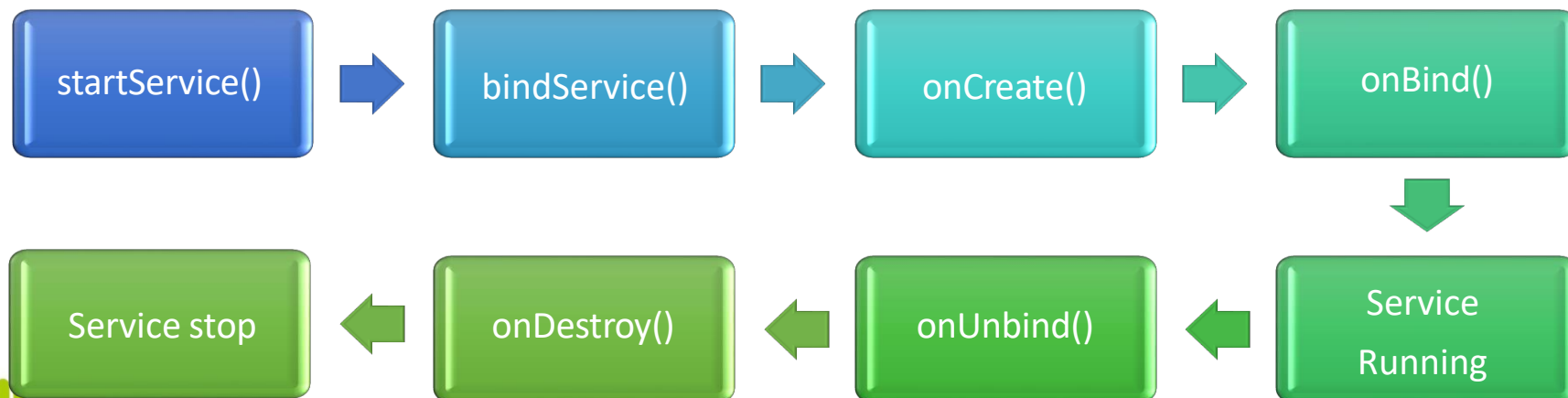
- 在Activity中通过stopService(Intent)关闭Service ;
Intent `intent` = new Intent(this, `MyService.class`);
stopService(`intent`);
- 或者在Service中通过stopSelf()关闭自身





模式二：通过bindService启动

- 通过Context.bindService()方法启动服务
- 通过Context.unbindService()关闭服务
- 多个客户端可以绑定至同一个服务。如果服务此时还没有加载，bindService()会先加载它。
- bindService启动的Service的生命周期





模式二：通过bindService启动

- 与模式一启动的Service无法进行通信和数据交换不同。若Service和访问者之间需要进行方法调用或数据交换，则应该使用：

`bindService(intent service, ServiceConnection conn, int flags)`

service：通过Intent指定要启动的Service；

conn：该对象用于监听访问者与Service之间的连接情况。
当访问者连接成功时将回调ServiceConnection对象的
`onServiceConnected(ComponentName name, IBinder service)`方法；

flags：绑定时是否自动创建Service。（自动或不自动创建）



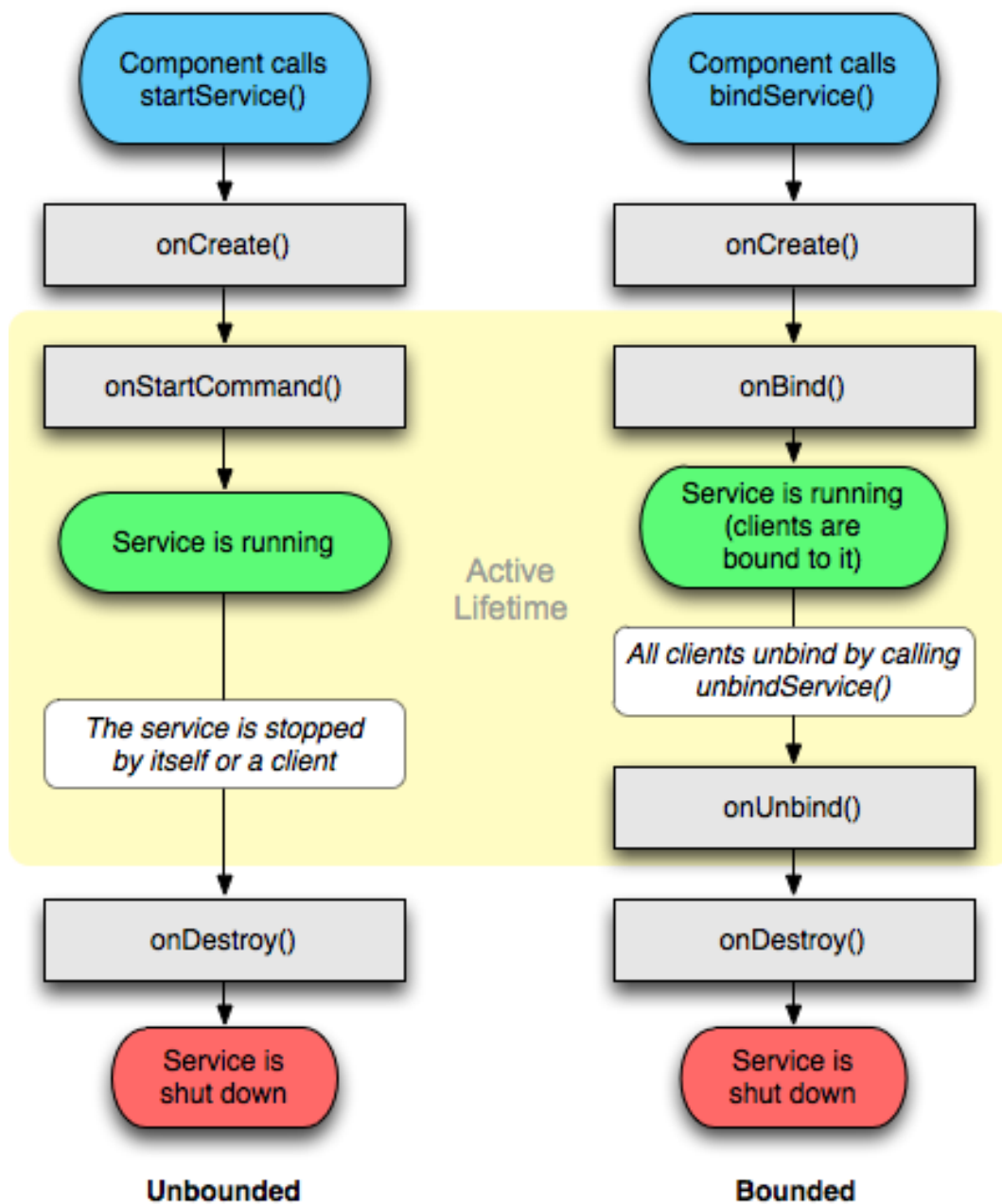


模式二：通过bindService启动

- onBind()只有采用Context.bindService()方法启动服务时才会回调该方法。该方法在调用者与服务绑定时被调用。
- 当调用者与服务**已经绑定**，多次调用Context.bindService()方法并不会导致该方法被多次调用。
- 采用Context.bindService()方法启动服务时，**只能调用onUnbind()方法解除调用者与服务解除**，服务结束时调用onDestroy()方法。



可被其他应用程序复用，比如天气预报服务，其他应用程序不需要再写这样的服务，调用已有的即可。



两种启动模式对比





如何实现一个Service

建立Android工程

- Activity : ServiceApplication.java。
程序入口，例程将在 这个Activity中启动Service。
- Service : MyService.java(继承Service的子类)
在大多数情况，需要重写onCreate和onStartCommand方法





Service实现

onStartCommand方法

- 在使用startService方法启动Service时被调用，在Service的生命周期内会多次被调用。
- onStartCommand方法代替了Android 2.0之前一直使用的onStart方法。
- 通过onStartCommand方法，可以明确告诉操作系统，在用户调用stopService或者stopSelf方法显式停止之前被操作系统杀死的Service重启的时候要执行的操作。





Service实现

MyService.java

```
public class MyService extends Service {
    @Override // 必须实现的方法
    public IBinder onBind(Intent intent) {
        return null; }
    @Override // 被启动时回调该方法
    public int onStartCommand(Intent intent, int flags, int
startId) {
        return Service.START_STICKY; }
    @Override
    public boolean onUnbind(Intent intent) {
        return super.onUnbind(intent); }
    @Override// 被关闭之前回调该方法
    public void onDestroy() {
        super.onDestroy(); }
}
```





Service实现

ServiceApplication.java

```
public class ServiceApplication extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        // 启动指定的Service  
        Intent intent = new Intent(this, MyService.class);  
        startService(intent);  
    }  
}
```





Service实现

在AndroidManifest.xml中注册这个Service

如果没有在此定义服务名称、访问权限，服务就无法被正确运行

```
<application android:icon="@drawable/icon"
    android:label="@string/app_name">
    <activity android:name=".ServiceApplication"
        android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service android:name=".MyService"
        android:process="serviceApplication"/>
</application>
```





绑定Activity和服务

- 由于Service没有界面，所以用户控制Service需要通过另外一个Activity来接收用户输入。
- 通过绑定 Activity 与 Service，可以实现 Activity 与 Service 之间的交互。例如在Activity中控制音乐播放 Service对音乐进行开始/停止，快进/快退等操作。





绑定Activity和服务

Activity和服务交互示意图



对于Service的onBind()方法所返回IBinder对象来说，它可以被当成该Service组件所返回的代理对象，Service允许客户端通过该IBinder对象来访问Service内部的数据，实现客户端与Service之间的通信。





跨进程调用

Service运行在远程进程中

Android系统中，各种应用程序都运行在自己的进程中，操作系统也对进程空间进行保护，一个进程不能直接访问另一个进程的内存空间，进程之间一般无法进行数据交换。**所以进程间进行数据交互需要利用Android操作系统提供的进程间通讯（IPC）机制来实现。**





IPC机制——IBinder

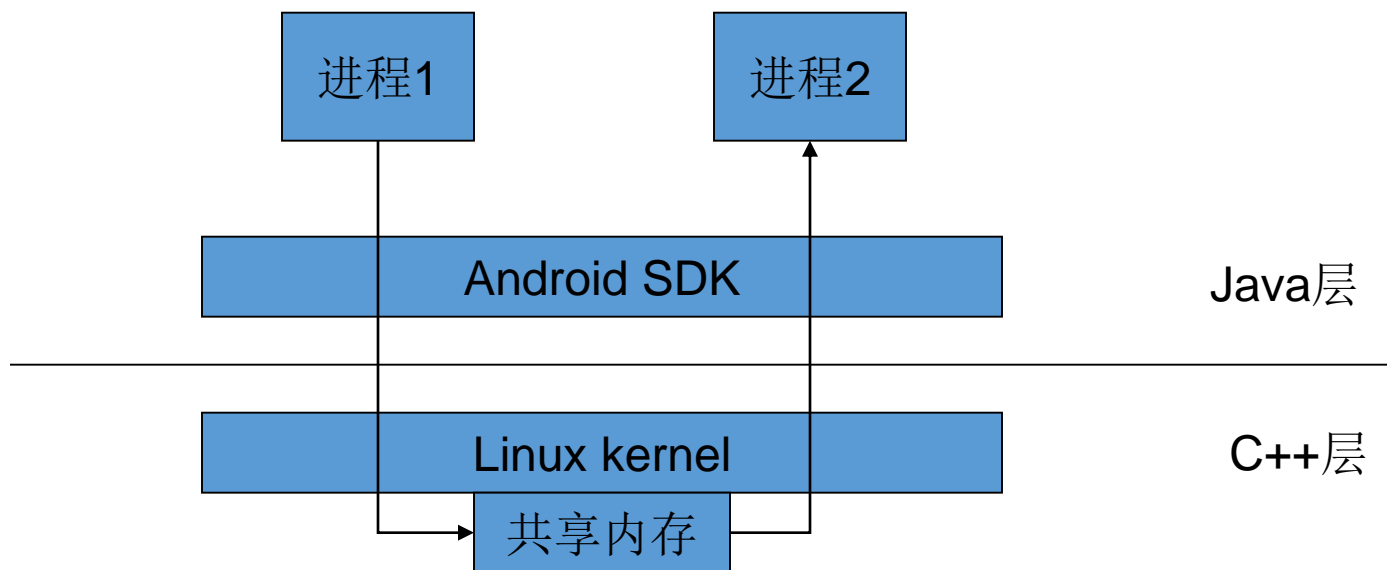
- IBinder是远程对象的基本接口，是为高效率进行**进程间通讯**设计的轻量级远程过程调用机制的核心。
- 该接口描述了与远程对象交互的抽象协议。
- 通常我们使用的时候并不是直接实现这个接口，而是继承自Binder父类。





IPC机制——IBinder

- Binder实质上是以IPC（Inter-Process Communication，进程间通信）框架为基础。可以简单按下图理解，其实质就是通过共享内存实现进程间的通讯。





IPC机制——IBinder

- IBinder的主要API是transact()，与它对应另一方法是Binder.onTransact()。
- 第一个方法使你可以向远端的IBinder对象发送发出调用，第二个方法使你自己的远程对象能够响应接收到的调用。
- IBinder的API都是同步执行的，比如transact()直到对方的Binder.onTransact()方法调用完成后才返回。调用发生在进程内时无疑是这样的，而在进程间时，在IPC的帮助下，也是同样的效果。





IPC机制——IBinder

通过transact()发送的数据是Parcel，Parcel是一种一般的缓冲区，除了有数据外还带有一些描述它内容的元数据。

元数据用于管理IBinder对象的引用，这样就能在缓冲区从一个进程移动到另一个进程时保存这些引用。

这样就保证了当一个IBinder被写入到Parcel并发送到另一个进程中，如果另一个进程把同一个IBinder的引用回发到原来的进程，那么这个原来的进程就能接收到发出的那个IBinder的引用。

这种机制使IBinder和Binder像唯一标志符那样在进程间管理。





IPC机制——IBinder

- onTransact(int code, Parcel data, Parcel reply,int flags)
- transact(int code, Parcel data, Parcel reply,int flags)
- 通过onTransact和transact交互的数据都被封装在Parcel中
- 由于IBinder只提供了一个消息传递接口，只能通过int类型的输入参数code对消息进行识别和判断





IPC机制——IBinder

Binder机制还支持进程间的递归调用。

例如，进程A执行自己的IBinder的transact()调用进程B的Binder，而进程B在其Binder.onTransact()中又用transact()向进程A发起调用，那么进程A在等待它发出的调用返回的同时，还会用Binder.onTransact()响应进程B的transact()。

总之Binder造成的结果就是让我们感觉到跨进程的调用与进程内的调用没什么区别。





IPC机制——IBinder

要实现IBinder来支持远程调用，应从Binder类派生一个类。Binder实现了IBinder接口。可自己实现，也可跟据需要由开发包中的工具生成。

这个工具叫AIDL，通过AIDL语言定义远程对象的方法，然后用AIDL工具生成Binder的派生类，然后就可使用之
[AIDL资料链接](#)





绑定Activity和服务

- MyService.java

```
public class MyService extends Service {  
    private IBinder mBinder = new MyBinder();  
    @Override  
    public IBinder onBind(Intent intent) {  
        // 必须实现的接口  
        return mBinder;  
    }  
    public class MyBinder extends Binder  
    {  
        @Override  
        protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws  
            RemoteException {  
            switch (code) {  
                //服务端处理  
            }  
            return super.onTransact(code, data, reply, flags);  
        }  
    }  
}
```





绑定Activity和服务

• ServiceApplication.java

```
public class ServiceApplication extends Activity {
    private IBinder mBinder;
    private ServiceConnection mConnection;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mConnection = new ServiceConnection(){
            @Override
            public void onServiceConnected(ComponentName name, IBinder service) {
                mBinder = service;
            }
            @Override
            public void onServiceDisconnected(ComponentName name) {
                mConnection = null;
            }
        };
        Intent intent = new Intent(this, MyService.class);
        startService(intent);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }
}
```





耗时操作的处理

Service本身存在两个问题

- Service不是一个单独的进程，它和应用程序在同一个进程中。
- Service不是一个线程，所以应该避免在Service里面进行耗时的操作；

把耗时的操作直接放在Service的onStart方法中，会出现Application Not Responding！

Service不是一个线程，不能直接处理耗时的操作。如果有耗时操作在Service里，就必须开启一个单独的线程来处理





IntentService

- 如果有耗时的操作，可以：
 1. Service里面开启新线程
 2. 使用IntentService来处理，IntentService内部有一个工作线程来处理耗时操作
- IntentService使用队列的方式将请求的Intent加入队列，然后开启一个worker thread(线程)来处理队列中的Intent，对于异步的startService请求，IntentService会处理完成一个之后再处理第二个，每一个请求都会在一个单独的worker thread中处理，不会阻塞应用程序的主线程。





IntentService

- 创建单独的worker线程来处理所有的Intent请求；
- 创建单独的worker线程来处理onHandleIntent()方法实现的代码，开发者无需处理多线程问题；
- 当所有请求处理完成后，IntentService会自动停止，开发者无需调用stopSelf()停止该Service；
- 提供了一个onBind()方法的默认实现，它返回null
- 提供了一个onStartCommand()方法的默认实现，它将Intent先传送至工作队列，然后从工作队列中每次取出一个传送至onHandleIntent()方法，在该方法中对Intent对相应的处理





IntentService

```
public class ServiceDemoActivity extends Activity {  
    /** Called when the activity is first created. */  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        startService(new Intent(this, MyService.class));  
        //主界面阻塞，最终会出现Application not responding  
        //连续两次启动IntentService，会发现应用程序不会阻塞，而且第二次请求会在第  
        一个请求结束之后运行(这个证实了IntentService采用单独的线程每次只从队列中拿出一个请求  
        进行处理)  
        startService(new Intent(this, MyIntentService.class));  
        startService(new Intent(this, MyIntentService.class));  
    }  
}
```





IntentService

```
public class MyService extends Service {  
    @Override  
    public void onCreate() {  
        super.onCreate();  
    }  
    @Override  
    public int onStartCommand(Intent intent, int flags, int startId)  
    {  
        super.onStartCommand(intent, flags, startId);  
        //经测试，Service里面是不能进行耗时的操作的，必须要手动开启一个工作线程来处理耗时操作  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                // 此处进行耗时的操作，这里只是简单地让线程睡眠了1s  
                try { Thread.sleep(1000); } catch (Exception e) {  
                    e.printStackTrace();  
                }  
            }  
        }).start();  
        return START_STICKY;  
    }  
    @Override  
    public IBinder onBind(Intent intent) {  
        return null;  
    }  
}
```





IntentService

```
public class MyIntentService extends IntentService { public
    MyIntentService() {
        super("yyyyyyyyyyyy");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        // 经测试，IntentService里面是可以进行耗时的操作的
        //IntentService使用队列的方式将请求的Intent加入队列，然后开启一个worker
        thread（线程）来处理队列中的Intent
        //对于异步的startService请求，IntentService会处理完成一个之后再处理第二个
        System.out.println("onStart"); try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("睡眠结束");
    }
}
```





多线程应用背景

1. 为了提供良好的用户体验，我们必须保证程序有高响应性，所以不能在UI线程中进行耗时的计算或I/O操作。
2. Android操作系统在下面情况下会强制关闭程序
 - UI线程在5秒内没有响应
 - 广播对象不能再10秒内完成onReceive方法





多线程应用场景

- 在工程实现中，需要采用多线程的方法，将大规模的计算放在后台线程中进行计算，然后将计算结果再显示到前台。
- 例子：在后台下载网络图片，下载完成后更新UI显示在屏幕上。





多线程基础

//新建线程

```
Thread mThread = new Thread()  
{  
    @Override  
    public void run()  
    {  
        timeConsumingProcess();  
    }  
};  
mThread.start();
```





新建用户线程

- 由于Android操作系统的线程安全机制，不能在非UI线程中重绘UI，所以在用户线程中进行类似更改进度条，修改TextView文字等操作都会造成程序强制关闭（FC）
- Android提供了两种方法解决上述问题：
 1. Handler
 2. AsyncTask





Handler机制

- Android操作系统在UI线程中，缺省维护该MessageQueue和一个Looper。
- Looper伪码

```
while(true)
{
    Msg = getFirstMessage(MessageQueue)
    if(Msg != null)
        processMessage()
}
```





消息系统模型

Send/Post Message

接收消息到
消息队列

消息队列

Message

Message

Message

.....

GetMessage

唤醒

消息循环
Loop

没有消息
挂起

DispatchMessage

HandleMessage





Handler机制

- Looper通过一个死循环，当有消息Message加入队列时，通过FIFO的顺序处理消息。
- 一个Message中包括了处理Message的Handler对象还有消息内容。
- 这种机制对应这设计模式中的命令模式
- Handler与UI线程是同一个线程，所以我们在用户线程中完成计算之后，可以通过向消息队列加入一个消息，通知特定的Handler去更改UI。





Handler实现

```
/* ServiceApplication.java */
Handler mHandler = new Handler()
{
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch(msg.what)
        { //在此处判断消息类型并更新UI }
        }
};

Thread mThread = new Thread()
{
    @Override
    public void run()
    {
        timeConsumingProcess(); //进行耗时操作
        //定义接收消息的Handler对象，并将消息加入队列
        mHandler.obtainMessage(type).sendToTarget(); }
};
```

与UI同一线程的消息处理器Handler，专门负责处理非UI线程发送过来的各种消息，更新UI。

非UI线程负责耗时工作，将不同类型的消息发送给上面定义的Handler。





AsyncTask概述

- AsyncTask是在Android SDK 1.5之后推出的一个方便编写后台线程与UI线程交互的辅助类。
- AsyncTask的内部实现是一个线程池，每个后台任务会提交到线程池中的线程执行，然后使用Thread+Handler的方式调用回调函数（对程序员透明）。
- AsyncTask抽象出后台线程运行的五个状态，分别是：
1、准备运行，2、正在后台运行，3、进度更新，
4、完成后台任务，5、取消任务

对于这五个阶段，AsyncTask提供了五个回调函数。





AsyncTask可重载的回调函数

1. 准备运行：onPreExecute()

该回调函数在任务被执行之后立即由UI线程调用。这个步骤通常用来建立任务，在用户接口（UI）上显示进度条。

2. 正在后台运行：doInBackground(Params...)

该回调函数由后台线程在onPreExecute()方法执行结束后立即调用。通常在这里执行耗时的后台计算。计算的结果必须由该函数返回，并被传递到onPostExecute()中。

在该函数内也可以使用publishProgress(Progress...)来发布一个或多个进度单位(unitsofprogress)。这些值将会在onProgressUpdate(Progress...)中被发布到UI线程。





AsyncTask可重载的回调函数

3. 进度更新：onProgressUpdate(Progress...),该函数由UI线程在publishProgress(Progress...)方法调用完后被调用。一般用于动态地显示一个进度条。

4. 完成后台任务：onPostExecute(Result),当后台计算结束后调用。后台计算的结果会被作为参数传递给这一函数。

5、取消任务：onCancelled()，在调用AsyncTask的cancel()方法



时调用



AsyncTask的构造函数

- 三个模板参数 <Params type, Progress type, Result type>
 - 1.Params , 传递给后台任务的参数类型。
 - 2.Progress , 后台计算执行过程中, 进度单位 (progress units) 的类型。(就是后台程序已经执行了百分之几了。)
 - 3.Result , 后台执行返回的结果的类型。
- AsyncTask并不总是需要使用上面的全部3种类型。标识不使用的类型很简单, 只需要使用Void类型即可。





AsyncTask使用

ServiceApplication.java

```
class MyTask extends AsyncTask<Integer,Integer,Integer>
{
    @Override
    protected Integer doInBackground(Integer... params) {
        //重写该函数，实现后台处理大规模计算
        return null;
    }
    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
        //重写该回调函数，更新UI
    }
}
```

很清晰的一套模板方法设计模式，只需要重写关键事件，不用去了解底层的多线程并发的实现机制





AsyncTask更新进度条的例子

1.在布局里放置进度条，在代码里定义进度条
`ProgressBar mProgBar ;`

2.定义class MyTask extends
`AsyncTask<Integer,Integer,String>`

3.在MyTask 的doInBackground里执行耗时操作，在适当的时候调用publishProgress(n)来设置进度条进度。





AsyncTask更新进度条的例子

4. 在MyTask的onProgressUpdate (Integer... progress)里具体设置进度条进度，例如

```
mProgBar.setProgress(progress[0]);
```

注意：这个函数在doInBackground调用publishProgress时触发，虽然调用时只有一个参数，但是这里取到的是一个数组，所以要用progresss[0]来取值，第n个参数就用progress[n]来取值。

5. 定义并运行：

```
MyTask mt = new MyTask();  
mt.execute(100);
```



A large crowd of 3D human figures, mostly light blue, with one prominent red figure in the foreground on the left, raising its right hand. The figures are stylized and have a soft, rounded appearance. The background is a light blue gradient.

Questions?

