Android AsyncTask 完全解析，带你从源码的角度彻底理解

视频教程：http://www.jikexueyuan.com/study/5/lid/22.html?hmsr=wenku_asynctask

我们都知道，Android UI 是线程不安全的，如果想要在子线程里进行 UI 操作，就需要借助 Android 的异步消息处理机制。之前我也写过了一篇文章从源码层面分析了 Android 的异步消息处理机制，感兴趣的朋友

可以参考 **Android Handler、Message 完全解析，带你从源码的角度彻底理解** 。

不过为了更加方便我们在子线程中更新 UI 元素，Android 从1.5版本就引入了一个 AsyncTask 类，使用它就可以非常灵活方便地从子线程切换到 UI 线程，我们本篇文章的主角也就正是它了。

AsyncTask 很早就出现在 Android 的 API 里了，所以我相信大多数朋友对它的用法都已经非常熟悉。不过今天我还是准备从 AsyncTask 的基本用法开始讲起，然后我们再来一起分析下 AsyncTask 源码，看看它是如何实现的，最后我会介绍一些关于 AsyncTask 你所不知道的秘密。

AsyncTask 的基本用法

首先来看一下 AsyncTask 的基本用法，由于 AsyncTask 是一个抽象类，所以如果我们想使用它，就必须要创建一个子类去继承它。在继承时我们可以为 AsyncTask 类指定三个泛型参数，这三个参数的用途如下：

1. Params

在执行 AsyncTask 时需要传入的参数，可用于在后台任务中使用。

2. Progress

后台任务执行时，如果需要在界面上显示当前的进度，则使用这里指定的泛型作为进度单位。

3. Result

当任务执行完毕后，如果需要对结果进行返回，则使用这里指定的泛型作为返回值类型。

因此，一个最简单的自定义 AsyncTask 就可以写成如下方式：

[java]view plaincopy

```
1  class DownloadTask extends AsyncTask<Void, Integer, Boolean> {
2  ……
3  }
```

这里我们把 AsyncTask 的第一个泛型参数指定为 Void，表示在执行 AsyncTask 的时候不需要传入参数给后台任务。第二个泛型参数指定为 Integer，表示使用整型数据来作为进度显示单位。第三个泛型参数指定为 Boolean，则表示使用布尔型数据来反馈执行结果。

当然，目前我们自定义的 DownloadTask 还是一个空任务，并不能进行任何实际的操作，我们还需要去重写 AsyncTask 中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下四个：

1. onPreExecute()

这个方法会在后台任务开始执行之间调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

2. doInBackground(Params...)

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。任务一旦完成就可以通过 return 语句来将任务的执行结果进行返回，如果 AsyncTask 的第三个泛型参数指定的是 Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行 UI 操作的，如果需要更新 UI 元素，比如说反馈当前任务的执行进度，可以调用 publishProgress(Progress...)方法来完成。

3. onProgressUpdate(Progress...)

当在后台任务中调用了 publishProgress(Progress...)方法后，这个方法就很快会被调用，方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对 UI 进行操作，利用参数中的数值就可以对界面元素进行相应的更新。

4. onPostExecute(Result)

当后台任务执行完毕并通过 return 语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些 UI 操作，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

因此，一个比较完整的自定义 AsyncTask 就可以写成如下方式：

[java]view plaincopy

```
4   class DownloadTask extends AsyncTask<Void, Integer, Boolean> {
5
6   @Override
7   protected void onPreExecute() {
8   progressDialog.show();
9   }
10
11  @Override
12  protected Boolean doInBackground(Void... params) {
13  try {
14  while (true) {
15  int downloadPercent = doDownload();
```

```
16  publishProgress(downloadPercent);
17  if (downloadPercent >= 100) {
18  break;
19  }
20  }
21  } catch (Exception e) {
22  return false;
23  }
24  return true;
25  }
26
27  @Override
28  protected void onProgressUpdate(Integer... values) {
29  progressDialog.setMessage("当前下载进度：" + values[0] + "%");
30  }
31
32  @Override
33  protected void onPostExecute(Boolean result) {
34  progressDialog.dismiss();
35  if (result) {
36  Toast.makeText(context, "下载成功", Toast.LENGTH_SHORT).show();
37  } else {
38  Toast.makeText(context, "下载失败", Toast.LENGTH_SHORT).show();
39  }
40  }
41  }
```

这里我们模拟了一个下载任务，在 doInBackground()方法中去执行具体的下载逻辑，在 onProgressUpdate()方法中显示当前的下载进度,在 onPostExecute()方法中来提示任务的执行结果。如果想要启动这个任务，只需要简单地调用以下代码即可：

```
[java]view plaincopy
```

```
42  new DownloadTask().execute();
```

以上就是 AsyncTask 的基本用法，怎么样，是不是感觉在子线程和 UI 线程之间进行切换变得灵活了很多？我们并不需求去考虑什么异步消息处理机制，也不需要专门使用一个 Handler 来发送和接收消息，只需要调用一下 publishProgress()方法就可以轻松地从子线程切换到 UI 线程了。

分析 AsyncTask 的源码

虽然 AsyncTask 这么简单好用，但你知道它是怎样实现的吗？那么接下来，我们就来分析一下 AsyncTask 的源码，对它的实现原理一探究竟。注意这里我选用的是 Android 4.0的源码，如果你查看的是其它版本的源码，可能会有一些出入。

从之前 DownloadTask 的代码就可以看出，在启动某一个任务之前，要先 new 出它的实例，因此，我们就先来看一看 AsyncTask 构造函数中的源码，如下所示：

[java]view plaincopy

```
43  public AsyncTask() {
44  mWorker = new WorkerRunnable<Params, Result>() {
45  public Result call() throws Exception {
46  mTaskInvoked.set(true);
47  Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
48  return postResult(doInBackground(mParams));
49  }
50  };
51  mFuture = new FutureTask<Result>(mWorker) {
52  @Override
53  protected void done() {
54  try {
55  final Result result = get();
56  postResultIfNotInvoked(result);
57  } catch (InterruptedException e) {
58  android.util.Log.w(LOG_TAG, e);
59  } catch (ExecutionException e) {
60  throw new RuntimeException("An error occured while executing doInBackground()",
```

```
61  e.getCause());
62  } catch (CancellationException e) {
63  postResultIfNotInvoked(null);
64  } catch (Throwable t) {
65  throw new RuntimeException("An error occured while executing "
66  + "doInBackground()", t);
67  }
68  }
69  };
70  }
```

这段代码虽然看起来有点长，但实际上并没有任何具体的逻辑会得到执行，只是初始化了两个变量，mWorker 和 mFuture，并在初始化 mFuture 的时候将 mWorker 作为参数传入。mWorker 是一个 Callable 对象，mFuture 是一个 FutureTask 对象，这两个变量会暂时保存在内存中，稍后才会用到它们。

接着如果想要启动某一个任务，就需要调用该任务的 execute()方法，因此现在我们来看一看 execute()方法的源码，如下所示：

[java]view plaincopy

```
71  public final AsyncTask<Params, Progress, Result> execute(Params... params) {
72  return executeOnExecutor(sDefaultExecutor, params);
73  }
```

简单的有点过分了，只有一行代码，仅是调用了 executeOnExecutor()方法，那么具体的逻辑就应该写在这个方法里了，快跟进去瞧一瞧：

[java]view plaincopy

```
74  public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
75  Params... params) {
76  if (mStatus != Status.PENDING) {
77  switch (mStatus) {
78  case RUNNING:
79  throw new IllegalStateException("Cannot execute task:"
80  + " the task is already running.");
```

```
81  case FINISHED:
82  throw new IllegalStateException("Cannot execute task:"
83  + " the task has already been executed "
84  + "(a task can be executed only once)");
85  }
86  }
87  mStatus = Status.RUNNING;
88  onPreExecute();
89  mWorker.mParams = params;
90  exec.execute(mFuture);
91  return this;
92  }
```

果然，这里的代码看上去才正常点。可以看到，在第 15 行调用了 onPreExecute()方法，因此证明了 onPreExecute()方法会第一个得到执行。可是接下来的代码就看不明白了，怎么没见到哪里有调用 doInBackground()方法呢？别着急，慢慢找总会找到的，我们看到，在第 17 行调用了 Executor 的 execute()方法，并将前面初始化的 mFuture 对象传了进去，那么这个 Executor 对象又是什么呢？查看上面的 execute()方法，原来是传入了一个 sDefaultExecutor 变量，接着找一下这个 sDefaultExecutor 变量是在哪里定义的，源码如下所示：

[java]view plaincopy

```
93  public static final Executor SERIAL_EXECUTOR = new SerialExecutor();
94  ……
95  private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
```

可以看到，这里先 new 出了一个 SERIAL_EXECUTOR 常量，然后将 sDefaultExecutor 的值赋值为这个常量，也就是说明，刚才在 executeOnExecutor()方法中调用的 execute()方法，其实也就是调用的 SerialExecutor 类中的 execute()方法。那么我们自然要去看看 SerialExecutor 的源码了，如下所示：

[java]view plaincopy

```
96  private static class SerialExecutor implements Executor {
97  final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
98  Runnable mActive;
99
```

```
100 public synchronized void execute(final Runnable r) {
101 mTasks.offer(new Runnable() {
102 public void run() {
103 try {
104 r.run();
105 } finally {
106 scheduleNext();
107 }
108 }
109 });
110 if (mActive == null) {
111 scheduleNext();
112 }
113 }
114
115 protected synchronized void scheduleNext() {
116 if ((mActive = mTasks.poll()) != null) {
117 THREAD_POOL_EXECUTOR.execute(mActive);
118 }
119 }
120 }
```

SerialExecutor 类中也有一个 execute()方法，这个方法里的所有逻辑就是在子线程中执行的了，注意这个方法有一个 Runnable 参数，那么目前这个参数的值是什么呢？当然就是 mFuture 对象了，也就是说在第 9 行我们要调用的是 FutureTask 类的 run()方法，而在这个方法里又会去调用 Sync 内部类的 innerRun()方法，因此我们直接来看 innerRun()方法的源码：

[java]view plaincopy

```
121 void innerRun() {
122 if (!compareAndSetState(READY, RUNNING))
123 return;
```

```
124 runner = Thread.currentThread();
125 if (getState() == RUNNING) { // recheck after setting thread
126 V result;
127 try {
128 result = callable.call();
129 } catch (Throwable ex) {
130 setException(ex);
131 return;
132 }
133 set(result);
134 } else {
135 releaseShared(0); // cancel
136 }
137 }
```

可以看到，在第 8 行调用了 callable 的 call()方法，那么这个 callable 对象是什么呢？其实就是在初始化 mFuture 对象时传入的 mWorker 对象了，此时调用的 call()方法，也就是一开始在 AsyncTask 的构造函数中指定的，我们把它单独拿出来看一下，代码如下所示：

[java]view plaincopy

```
138 public Result call() throws Exception {
139 mTaskInvoked.set(true);
140 Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
141 return postResult(doInBackground(mParams));
142 }
```

在 postResult()方法的参数里面，我们终于找到了 doInBackground()方法的调用处，虽然经过了很多周转，但目前的代码仍然是运行在子线程当中的，所以这也就是为什么我们可以在 doInBackground()方法中去处理耗时的逻辑。接着将 doInBackground()方法返回的结果传递给了 postResult()方法，这个方法的源码如下所示：

[java]view plaincopy

```
143 private Result postResult(Result result) {
144 Message message = sHandler.obtainMessage(MESSAGE_POST_RESULT,
```

```
145 new AsyncTaskResult<Result>(this, result));
146 message.sendToTarget();
147 return result;
148 }
```

如果你已经熟悉了异步消息处理机制,这段代码对你来说一定非常简单吧。这里使用 sHandler 对象发出了一条消息，消息中携带了 MESSAGE_POST_RESULT 常量和一个表示任务执行结果的 AsyncTaskResult 对象。这个 sHandler 对象是 InternalHandler 类的一个实例，那么稍后这条消息肯定会在 InternalHandler 的 handleMessage()方法中被处理。InternalHandler 的源码如下所示：

[java]view plaincopy

```
149 private static class InternalHandler extends Handler {
150 @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
151 @Override
152 public void handleMessage(Message msg) {
153 AsyncTaskResult result = (AsyncTaskResult) msg.obj;
154 switch (msg.what) {
155 case MESSAGE_POST_RESULT:
156 // There is only one result
157 result.mTask.finish(result.mData[0]);
158 break;
159 case MESSAGE_POST_PROGRESS:
160 result.mTask.onProgressUpdate(result.mData);
161 break;
162 }
163 }
164 }
```

这里对消息的类型进行了判断，如果这是一条 MESSAGE_POST_RESULT 消息，就会去执行 finish()方法,如果这是一条 MESSAGE_POST_PROGRESS 消息,就会去执行 onProgressUpdate()方法。那么 finish()方法的源码如下所示：

[java]view plaincopy

```
165 private void finish(Result result) {
```

```
166 if (isCancelled()) {
167 onCancelled(result);
168 } else {
169 onPostExecute(result);
170 }
171 mStatus = Status.FINISHED;
172 }
```

可以看到，如果当前任务被取消掉了，就会调用 onCancelled()方法，如果没有被取消，则调用 onPostExecute()方法，这样当前任务的执行就全部结束了。

我们注意到，在刚才 InternalHandler 的 handleMessage()方法里，还有一种 MESSAGE_POST_PROGRESS 的消息类型，这种消息是用于当前进度的，调用的正是 onProgressUpdate()方法，那么什么时候才会发出这样一条消息呢？相信你已经猜到了，查看 publishProgress()方法的源码，如下所示：

[java]view plaincopy

```
173 protected final void publishProgress(Progress... values) {
174 if (!isCancelled()) {
175 sHandler.obtainMessage(MESSAGE_POST_PROGRESS,
176 new AsyncTaskResult<Progress>(this, values)).sendToTarget();
177 }
178 }
```

非常清晰了吧！正因如此，在 doInBackground()方法中调用 publishProgress()方法才可以从子线程切换到 UI 线程，从而完成对 UI 元素的更新操作。其实也没有什么神秘的，因为说到底，AsyncTask 也是使用的异步消息处理机制，只是做了非常好的封装而已。

读到这里，相信你对 AsyncTask 中的每个回调方法的作用、原理、以及何时会被调用都已经搞明白了吧。

关于 AsyncTask 你所不知道的秘密

不得不说，刚才我们在分析 SerialExecutor 的时候，其实并没有分析的很仔细，仅仅只是关注了它会调用 mFuture 中的 run()方法，但是至于什么时候会调用我们并没有进一步地研究。其实 SerialExecutor 也是 AsyncTask 在3.0版本以后做了最主要的修改的地方，它在 AsyncTask 中是以常量的形式被使用的，因此在整个应用程序中的所有 AsyncTask 实例都会共用同一个 SerialExecutor。下面我们就来对这个类进行更加详细的分析，为了方便阅读，我把它的代码再贴出来一遍：

[java]view plaincopy

```
179 private static class SerialExecutor implements Executor {
180 final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
181 Runnable mActive;
182
183 public synchronized void execute(final Runnable r) {
184 mTasks.offer(new Runnable() {
185 public void run() {
186 try {
187 r.run();
188 } finally {
189 scheduleNext();
190 }
191 }
192 });
193 if (mActive == null) {
194 scheduleNext();
195 }
196 }
197
198 protected synchronized void scheduleNext() {
199 if ((mActive = mTasks.poll()) != null) {
200 THREAD_POOL_EXECUTOR.execute(mActive);
201 }
202 }
203 }
```

可以看到，SerialExecutor 是使用 ArrayDeque 这个队列来管理 Runnable 对象的，如果我们一次性启动了很多个任务，首先在第一次运行 execute()方法的时候，会调用 ArrayDeque 的 offer()方法将传入的 Runnable 对象添加到队列的尾部，然后判断 mActive 对象是不是等于 null，第一次运行当然是等于 null 了，于是会调用 scheduleNext()方法。在这个方法中会从队列的头部取值，并赋值给 mActive 对象，然后调用 THREAD_POOL_EXECUTOR 去执行取出的取出的 Runnable 对象。之后如何又有新的任务被执行，同样还会调用 offer()方法将传入的 Runnable 添加到队列的尾部，但是再去给 mActive 对象做非空检查的时候就会发现 mActive

对象已经不再是 null 了，于是就不会再调用 scheduleNext()方法。

那么后面添加的任务岂不是永远得不到处理了？当然不是，看一看 offer()方法里传入的 Runnable 匿名类，这里使用了一个 try finally 代码块，并在 finally 中调用了 scheduleNext()方法，保证无论发生什么情况，这个方法都会被调用。也就是说，每次当一个任务执行完毕后，下一个任务才会得到执行，SerialExecutor 模仿的是单一线程池的效果，如果我们快速地启动了很多任务，同一时刻只会有一个线程正在执行，其余的均处于等待状态。**Android 照片墙应用实现，再多的图片也不怕崩溃** 这篇文章中例子的运行结果也证实了这个结论。

不过你可能还不知道，在 Android 3.0之前是并没有 SerialExecutor 这个类的，那个时候是直接在 AsyncTask 中构建了一个 sExecutor 常量，并对线程池总大小，同一时刻能够运行的线程数做了规定，代码如下所示：

[java]view plaincopy

```
204 private static final int CORE_POOL_SIZE = 5;
205 private static final int MAXIMUM_POOL_SIZE = 128;
206 private static final int KEEP_ALIVE = 10;
207 ……
208 private static final ThreadPoolExecutor sExecutor = new ThreadPoolExecutor(CORE_POOL_SIZE,
209 MAXIMUM_POOL_SIZE, KEEP_ALIVE, TimeUnit.SECONDS, sWorkQueue, sThreadFactory);
```

可以看到，这里规定同一时刻能够运行的线程数为 5 个，线程池总大小为 128。也就是说当我们启动了 10 个任务时，只有 5 个任务能够立刻执行，另外的 5 个任务则需要等待，当有一个任务执行完毕后，第 6 个任务才会启动，以此类推。而线程池中最大能存放的线程数是 128 个，当我们尝试去添加第 129 个任务时，程序就会崩溃。

因此在3.0版本中 AsyncTask 的改动还是挺大的，在3.0之前的 AsyncTask 可以同时有5个任务在执行，而3.0之后的 AsyncTask 同时只能有1个任务在执行。为什么升级之后可以同时执行的任务数反而变少了呢？这是因为更新后的 AsyncTask 已变得更加灵活，如果不想使用默认的线程池，还可以自由地进行配置。比如使用如下的代码来启动任务：

[java]view plaincopy

```
210 Executor exec = new ThreadPoolExecutor(15, 200, 10,
211 TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
212 new DownloadTask().executeOnExecutor(exec);
```

这样就可以使用我们自定义的一个 Executor 来执行任务，而不是使用 SerialExecutor。上述代码的效果允许在同一时刻有 15 个任务正在执行，并且最多能够存储 200 个任务。

转载出处：http://blog.csdn.net/guolin_blog/article/details/11711405