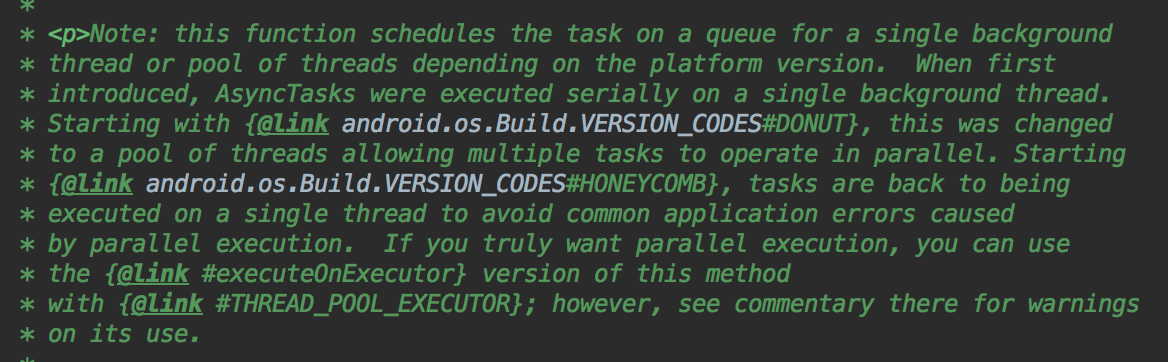
# **你真的了解AsyncTask吗？**

说到AsyncTask，它几乎能够用最简单的方式将操作异步执行，再呈现给UI线程。你不需要自己写一个线程，然后通过Handler去将结果返回给UI线程。只要简单的重写  
onPreExecute，doInBackground，onProgressUpdate,onPostExecute四个方法，然后调用execute方法，是不是超级简单。可是，你了解AsyncTask是如何操作你的任务的吗？它是如何封装Handler将异步任务执行结果返回给UI线程的？使用AsyncTask有哪些需要注意的？本文从源码分析AsyncTask的工作原理，部分内容来自源码。

### 任务执行方式

目前的AsyncTask默认的任务处理是在单线程中顺序执行，之前有过一段时间可以在线程池中执行，不信你看execute方法的注释：



我简单的翻译一下，execute方法将队列中的任务在一个后台的单线程或线程池中执行。AsyncTask的第一个版本是顺序执行。在1.6（DONUT）版本后，改成多任务的线程池中执行。但在3.2（HONEYCOMB）后，为了避免一些线程同步的错误，又改回在单线程中执行。如果想在线程池中执行，可以这样：

new AsynTask().executeOn(AsyncTask.THREAD\_POOL\_EXECUTOR,"")

显然这种方法，并不建议。

既然说到AsyncTask.THREAD\_POOL\_EXECUTOR，它是什么呢？

public static final Executor THREAD\_POOL\_EXECUTOR =

new ThreadPoolExecutor(CORE\_POOL\_SIZE,

MAXIMUM\_POOL\_SIZE,

KEEP\_ALIVE,

TimeUnit.SECONDS,

sPoolWorkQueue,

sThreadFactory);

THREAD\_POOL\_EXECUTOR是一个线程池的执行器(有关线程池的可以参考 [这篇文章](" \t "_blank) )。在这里你只要了解它是一个核心线程数量是CPU数+1，最大线程数量是2\*CPU数量+1就可以了。

话说回来，单线程顺序执行是如何执行的？请看：

private static volatile Executor sDefaultExecutor = SERIAL\_EXECUTOR;

这个sDefaultExecutor是AsyncTask任务执行器。看过源码你会发现有这样一个方法：

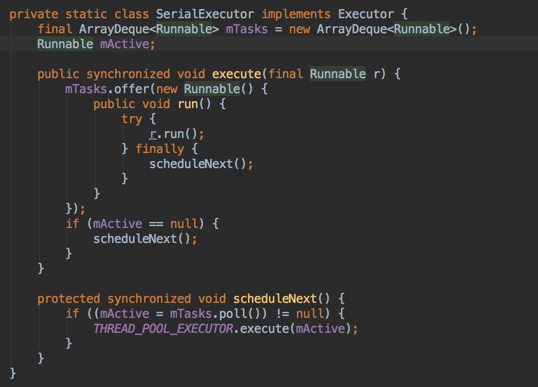
/\*\* @hide \*/public static void setDefaultExecutor(Executor exec) {

sDefaultExecutor = exec;

}

sDefaultExecutor是可以设置的，只不过你调用不了，被隐藏了（@hide）。

那么SERIAL\_EXECUTOR是什么呢？它是一个SerialExecutor的实例。



可以看到，在execute中会调用offer方法会将Runnable r包装一下放到ArrayDeque队列里，包装的新Runnable保证原来的Runnable执行之后会去取队列里的下一个Runnable，从而不会导致中断。  
scheduleNext做了什么呢？可以看到scheduleNext是从队列中取出Runnable然后交给THREAD\_POOL\_EXECUTOR执行。也就是说SerialExecutor只是将任务按先后顺序排列到队列中，真正执行任务的是THREAD\_POOL\_EXECUTOR。

#### 任务执行过程

在你调用execute的时候会这样：

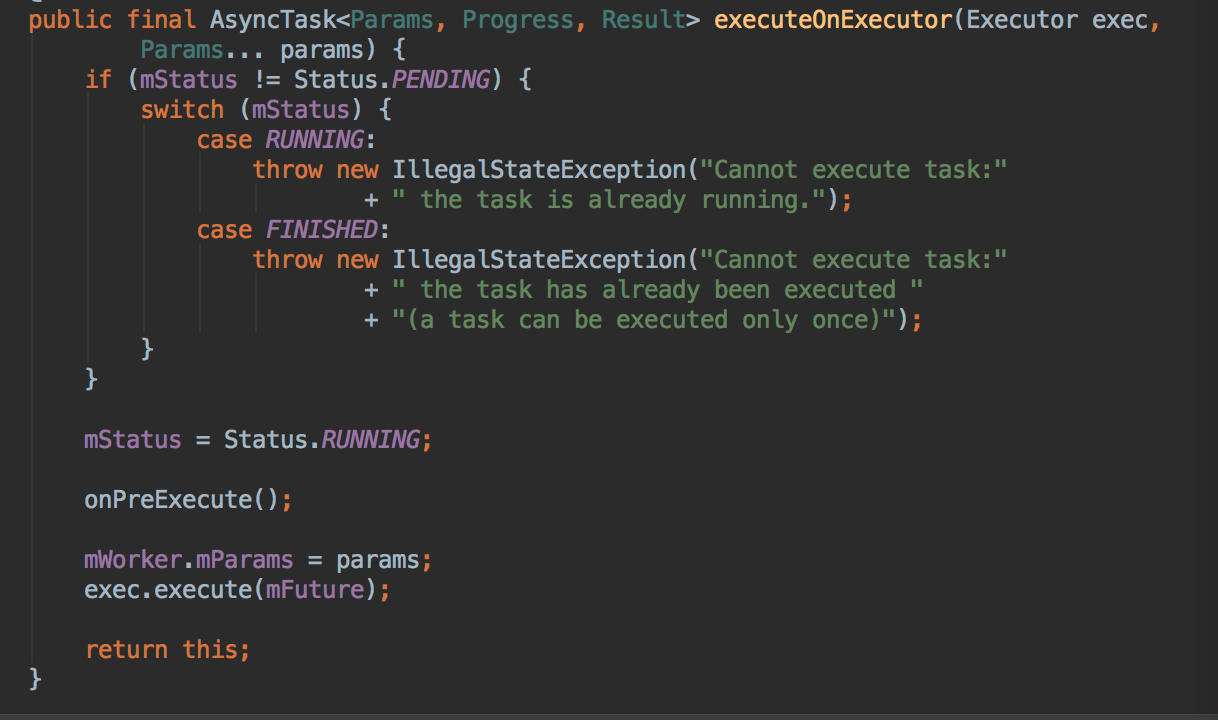
@MainThreadpublic final AsyncTask<Params, Progress, Result> execute(Params... params) {

return executeOnExecutor(sDefaultExecutor, params);

}

你看，最后还是会调用到executeOnExecutor，默认传了一个SERIAL\_EXECUTOR。并且，看见那个@MainThread了吧，execute一定在主线程调用。

请看executeOnExecutor:



每个AsyncTask都有一个Status，代表这个AsyncTask的状态，Status是一个枚举变量，每一个状态在这个Task的生命周期里赋值一次，也就是这个Task一定会经历 PENDING -> RUNNING -> FINISHED 的过程。  
PENDING代表Task还没有被执行，RUNNING代表当前任务正在执行，FINISHED代表的是onPostExecute方法已经执行完了，而不是doInBackground。

/\*\*

\* Indicates the current status of the task. Each status will be set only once

\* during the lifetime of a task.

\*/

public enum Status {

PENDING,

RUNNING,

FINISHED,

}

话说回到executeOnExecutor中，如果当前的Task的状态不是PENDING，那么就会抛出异常。也就是同一个Task，你只能execute一次，直到它的异步任务执行完成，你才可以再次调用他的execute方法，否则一定会报错。  
然后调用onPreExecute方法，之后会提交给SERIAL\_EXECUTOR执行。但是这个mWorker是什么？mFuture是什么？

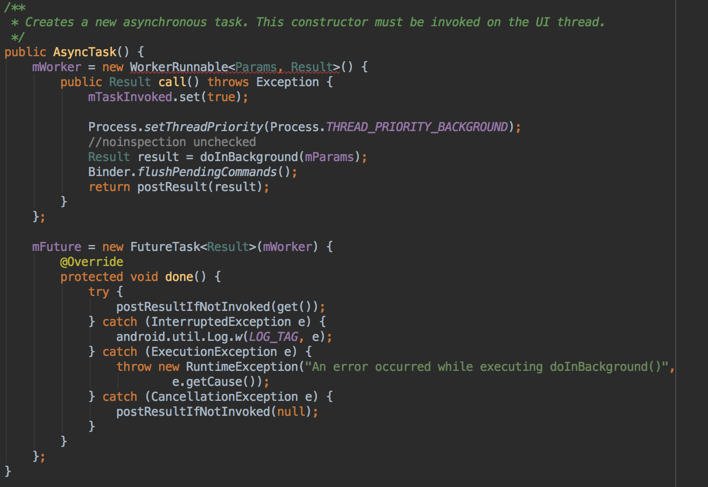
**mWorker**  
mWorker是WorkerRunnable的具体实现，实现Callable接口，相当于一个能够保存参数，返回结果的Runnable。

private static abstract class WorkerRunnable<Params, Result> implements Callable<Result> {

Params[] mParams;

}

当你创建一个AsyncTask的时候就会创建mWorker和mFutureTask。



可以看到，mWorker的call方法主要的工作是设置call是否被调用，调用你重写的doInBackground方法，获得Result(这个Result的类型就是你声明AsyncTask时传入的类型)，再将Result调用postResult方法返回。关于postResult请往下看。

**mFuture**  
可以看到mFuture中有一个postResultIfNotInvoked(get());方法，通过get方法获得mWorker的执行结果，然后调用postResultIfNotInvoked方法，由于某些原因，mWorker的call可能没有执行，所以在postResultIfNotInvoked中能够保证postResult一定会执行一次，要不在mWorker的call中执行，要不在postResultIfNotInvoked中执行。

private void postResultIfNotInvoked(Result result) {

final boolean wasTaskInvoked = mTaskInvoked.get();

if (!wasTaskInvoked) {

postResult(result);

}

}

那么这个postResult是干什么的?

private Result postResult(Result result) {

@SuppressWarnings("unchecked")

Message message = getHandler().obtainMessage(MESSAGE\_POST\_RESULT,

new AsyncTaskResult<Result>(this, result));

message.sendToTarget();

return result;

}

可以看到postResult实际上是获得了一个AsyncTask内部的一个Handler，将result包装在AsyncTaskResult中，并将它放在message发送给Handler。

那么AsyncTaskResult是如何封装的？

private static class AsyncTaskResult<Data> {

final AsyncTask mTask;

final Data[] mData;

AsyncTaskResult(AsyncTask task, Data... data) {

mTask = task;

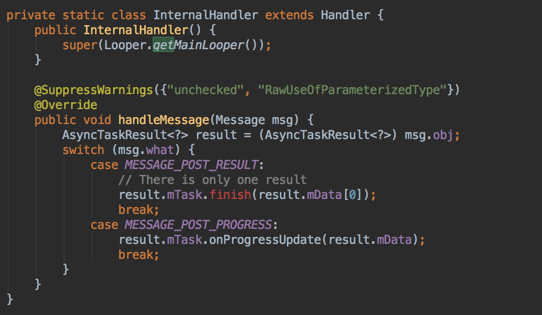
mData = data;

}

}

可以看到包含AsyncTask的实例（mTask）和数据（mData）。当将任务执行的结果返回时，mData保存的是Result，当更新进度的时候mData保存的是和Progress类型一样的数据。你可以往下看。

先说一说Handler。



每个AsyncTask都会获得一个InternalHandler的实例。可以看到，InternalHandler绑定到了主线程的Looper中(关于Looper与Handler的关系，可以参考[这篇文章](http://www.jianshu.com/p/27924ef1ea8f" \t "_blank))，所以你在异步线程中执行的结果最终都可以通过InternalHandler交给主线程处理。再看handlerMessage方法，获得AsyncTaskResult对象，如果传的是MESSAGE\_POST\_RESULT类型，就调用AsyncTask的finish方法(别忘了result.mTask其实就是当前的AsyncTask)。

finish做了什么？

private void finish(Result result) {

if (isCancelled()) {

onCancelled(result);

} else {

onPostExecute(result);

}

mStatus = Status.FINISHED;

}

可以看到，判断你是否取消了任务，取消则优先执行onCancelled回调，否则执行onPostExecute，并更改Task的状态。

如果是一个MESSAGE\_POST\_PROGRESS，就会执行onProgressUpdate方法。那MESSAGE\_POST\_PROGRESS的信息是谁去发送的呢？请看：

protected final void publishProgress(Progress... values) {

if (!isCancelled()) {

getHandler().obtainMessage(MESSAGE\_POST\_PROGRESS,

new AsyncTaskResult<Progress>(this, values)).sendToTarget();

}

}

也就是当你调用publishProgress的时候，会将传递的values包装成AsyncTaskResult，AsyncTaskResult的mData会保存进度的数据，将message发送给handler。

有个方法需要说明一下，就是cancle方法。

public final boolean cancel(boolean mayInterruptIfRunning) {

mCancelled.set(true);

return mFuture.cancel(mayInterruptIfRunning);

}

作用是设置被取消的状态，然后取消FutureTask的执行。当task已经执行完了，或已经被取消，或因为某些原因不能被取消，会返回false。如果任务已经执行，那么根据mayInterruptIfRunning决定是否打断（interrupt）当前正在执行Task的线程。  
调用这个方法会在doInBackground返回后回调onCancelled方法，并且onPostExecute不会执行，所以当你需要取消Task的时候记得在doInBackground通过isCancelled检查返回值。

### 注意事项

**1.**由于AsyncTask是单线程顺序执行的，所以不要用AsyncTask执行耗时太久的操作，如果有很多耗时太久的线程，最好使用线程池。  
**2.**onPreExecute、onProgressUpdate、onPostExecute都是在UI线程调用的，doInBackground在后台线程执行。  
**3.**调用cancel方法取消任务执行，这个时候onPostExecute就不会执行了，取而代之的是cancel方法，所以为了尽快的退出任务的执行，在doInBackground中调用isCancelled检查是否取消的状态。  
**4.**其他

* AsyncTask类一定要在主线程加载
* AsyncTask类的实例一定在主线程创建
* execute方法一定在主线程调用
* 不要主动调用onPreExecute等方法
* 任务只能在完成前执行一次。