简介

AsyncTask是一种轻量级的异步任务类，它可以在线程池中执行后台任务，然后把执行的进度和最终结果传递给主线程并在主线程中更新UI。从实现上来说，AsyncTask封装了Thread和Handler，通过AsyncTask可以更方便地执行后台任务以及在主线程访问UI，但AsyncTask不适合特别耗时的任务，特别耗时任务建议使用线程池。
AsyncTask类声明如下：

```
public abstract class AsyncTask<Params, Progress, Result> {

    //主线程执行，异步任务开始前执行
    onPreExecute();

    //子线程执行，执行异步任务，此方法中可通过publishProgress在主线程更新任务进度，publishProgress方法会调用onProgressUpdate方法。任务结束，还会在主线程调用onPostExecute方法
    doInBackground(Params...params);

    //主线程执行，当后台任务执行进度发生改变时调用
    onProgressUpdate(Progress...values);

    //在主线程中执行，当后台任务结束后调用
    onPostExecute(Result result);

    //主线程中执行，当后台任务被取消时执行。onCancelled调用后，不会调用onPostExecute方法
    onCancelled();
}
```

注意事项

1. AsyncTask的类必须在主线程中加载，即第一次访问AsyncTask必须发生在主线程。在Android 4.1及以上版本被系统自动完成。在Android 5.0源码中，在ActivityThread的main方法，会调用AsyncTask的init方法。
2. AsyncTask的对象必须在主线程中创建
3. execute方法必须在UI线程调用
4. 不要在程序中直接调用onPreExecute、onPostExecute、doInBackground和onProgressUpdate方法
5. 一个AsyncTask对象只能执行一次，即只能调用一次execute方法，否则会报异常
6. 在Android 1.6之前，AsyncTask是串行执行任务，Android 1.6版本AsyncTask开始采用线程池处理并行任务。但从Android 3.0开始，为了避免AsyncTask的并发错误，又采用了一个线程来串行执行任务。但是可以通过AsyncTask的executeOnExecutor方法来并行执行任务。

AsyncTask工作原理

先从AsyncTask的execute方法开始分析

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

public final AsyncTask<Params, Progress, Result>
executeOnExecutor(Executor exec, Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute
task:" + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute
task:" + " the task has already been executed " + " (a task can be
executed only once)");
        }
    }
    mStatus = Status.RUNNING;
    onPreExecute();
    mWorker.mParams = params;
    exec.execute(mFuture);
    return this;
}
```

在上面的代码中，sDefaultExecutor实际上是一个串行的线程池，一个进程中所有的AsyncTask全部在这个串行的线程池中排队执行。在executeOnExecutor方法中，AsyncTask的onPreExecute方法最先执行，然后线程池开始执行。线程池的执行过程如下所示：

```
public static final Executor SERIAL_EXECUTOR = new
SerialExecutor();
private static volatile Executor sDefaultExecutor =
SERIAL_EXECUTOR;

private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;

    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
           public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
           }
        });
        if (mActive == null) {
            scheduleNext();
        }
    }

    protected synchronized void scheduleNext() {
        if (mActive = mTask.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);
        }
    }
}
```

系统先会把Params参数封装为FutureTask对象，FutureTask是一个并发类，这里充当了Runnable的作用。接着FutureTask会交给SerialExecutor的execute方法去处理。execute方法会FutureTask对象插入到任务队列mTasks中。当一个AsyncTask任务执行完后，会继续调用scheduleNext()执行下一个任务。
AsyncTask中有两个线程池（SerialExecutor和THREAD_POOL_EXECUTOR)和一个Handler(InternalHandler)，其中线程池SerialExecutor用于任务的排队，而线程池THREAD_POOL_EXECUTOR用于真正地执行任务，InternalHandler用于将切换主线程。在AsyncTask的构造方法中有如下一段代码，由于FutureTask的run方法会调用mWorker的call方法，因此mWorker的call方法最终会在线程池中执行。

```
mWorker = new WorkerRunnable<Params, Result>() {
    public Result call() throws Exception {
        mTaskInvoked.set(true);


Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
        return postResult(doInBackground(mParams));
    }
};
```

mWoker的call方法首先将mTaskInvoked设为true，表示当前任务已经被调用，然后执行AsyncTask的doInBackground方法，接着将其返回值传递给postResult方法。

```
private Result postResult(Result result) {
    Message message = sHandler.obtainMessage(MESSAGE_POST_RESULT,
new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

postResult会通过sHandler发送一个MESSAGE_POST_RESULT的消息，sHandler的定义如下：

```
private static final InternalHandler sHandler = new
InternalHandler();

private static class InternalHandler extends Handler {

    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult result = msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}
```

可以发现，sHandler是一个静态的Handler对象，为了能够将执行环境切换到主线程，这就要求sHandler这个对象必须在主线程中创建。由于静态成员会在加载类的时候进行初始化，因此这就变相要求AsyncTask的类必须在主线程中加载。sHandler收到MESSAGE_POST_RESULT这个消息后会调用AsynsTask的finish方法。

```
private void finish(Result result) {
    if (isCancelled()) {
        onCancelled(result);
    } else {
        onPostExecute(result);
    }
    mStatus = Status.FINISHED;
}
```

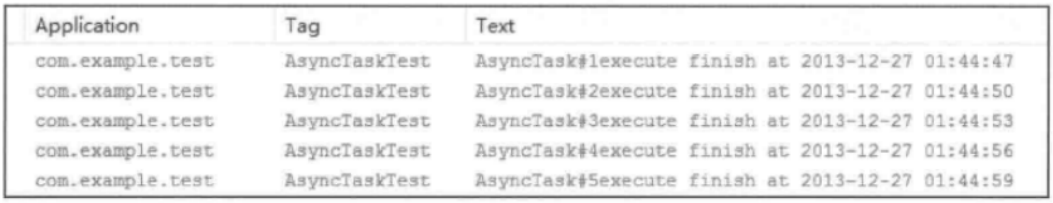
AsyncTask的finish方法逻辑比较简单，如果任务被取消了，那么就调用onCancelled方法，否则就调用onPostExecute方法，可以看到doInBackground的返回结果会传递给onPostExecute方法，到这里AsyncTask的整个工作过程就分析完毕了。

验证AsyncTask默认串行执行任务

1. 单击按钮的时候，同时执行5个AsyncTask任务，每个任务休眠3秒来模拟耗时操作，同时将每个AsyncTask执行结束的时间打印出来，这样就能观察AsyncTask到底是串行执行还是并行执行。

```
@Override
public void onClick(View v) {
    new MyAsyncTask("AsyncTask#1").execute("");
    new MyAsyncTask("AsyncTash#2").execute("");
    new MyAsyncTask("AsyncTash#3").execute("");
    new MyAsyncTask("AsyncTash#4").execute("");
    new MyAsyncTask("AsyncTash#5").execute("");
}

private static class MyAsyncTask extends AsyncTask<String, Integer,
String> {
    private String mName = "AsyncTask";

    public MyAsyncTask(String name) {
        super();
        mName = name;
    }

    @Override
    protected String doInBackground(String... params) {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return mName;
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        SimpleDateFormat df = new SimpleDataFormat("yyyy-MM-dd
HH:mm:ss");
        Log.e(TAG, result = "execute finish at " + df.formate(new
Date()));
    }
}
```

在Android 4.1.1上串行执行如下所示：



Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:44:47
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:44:50
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:44:53
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:44:56
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:44:59

在Android 2.3.3上并行执行如下所示：

Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:45:39
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:45:39

在Android 3.0以以上版本使用executeOnExecutor方法，并行执行。

```
@TargetApi(Build.VERSION_CODES.HONEYCOME)
@Override
public void onClick(View v) {
    new
MyAsyncTask("AsyncTask#1").executeOnExecutor(AsyncTask.THREAD_POOL_
EXECUTOR, "");
    new
MyAsyncTask("AsyncTask#2").executeOnExecutor(AsyncTask.THREAD_POOL_
EXECUTOR, "");
    new
MyAsyncTask("AsyncTask#3").executeOnExecutor(AsyncTask.THREAD_POOL_
EXECUTOR, "");
    new
MyAsyncTask("AsyncTask#4").executeOnExecutor(AsyncTask.THREAD_POOL_
EXECUTOR, "");
    new
MyAsyncTask("AsyncTask#5").executeOnExecutor(AsyncTask.THREAD_POOL_
EXECUTOR, "");
}
```

在Android 4.1.1上并行执行如下所示：

Application	Tag	Text
com.example.test	AsyncTaskTest	AsyncTask#1execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#2execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#4execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#3execute finish at 2013-12-27 01:52:40
com.example.test	AsyncTaskTest	AsyncTask#5execute finish at 2013-12-27 01:52:40