一、AsyncTask 示例

```
    class WorkTask extends AsyncTask<Integer, Integer, String> {

        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            Log.d(TAG, "onPreExecute");
        }

        @Override
        protected String doInBackground(Integer... integers) {

            Log.d(TAG, String.format("doInBackground:计算 %d 到 %d 之
和", integers[0], integers[1]));

            int sum = 0;
            for (int i = integers[0]; i <= integers[1]; i++) {
                try {
                    sum = sum + i;
                    publishProgress(i);
                    Thread.sleep(1000); // 模拟耗时操作
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (isCancelled()) break;
            }

            return String.valueOf(sum);
        }

        @Override
        protected void onProgressUpdate(Integer... values) {
            super.onProgressUpdate(values);
            Log.d(TAG, "onProgressUpdate: " + values[0]);
        }

        @Override
        protected void onPostExecute(String s) {
            super.onPostExecute(s);
            Log.d(TAG, "onPostExecute:计算结果 = " + s);
        }
    }

    // 创建实例，并执行，求 1 到 5 的和
    WorkTask wt = new WorkTask();
    wt.execute(1, 5);
```

示例是一个求和程序，用 Thread.sleep 来模拟耗时的操作，执行结果如下：

```
04-01 16:12:02.041 7203-7203/com.example.test.myapplication D/jeff:
onPreExecute
04-01 16:12:02.041 7203-7341/com.example.test.myapplication D/jeff:
doInBackground:计算 1 到 5 之和
04-01 16:12:02.051 7203-7203/com.example.test.myapplication D/jeff:
onProgressUpdate: 1
04-01 16:12:03.061 7203-7203/com.example.test.myapplication D/jeff:
onProgressUpdate: 2
04-01 16:12:04.061 7203-7203/com.example.test.myapplication D/jeff:
onProgressUpdate: 3
04-01 16:12:05.061 7203-7203/com.example.test.myapplication D/jeff:
onProgressUpdate: 4
04-01 16:12:06.061 7203-7203/com.example.test.myapplication D/jeff:
onProgressUpdate: 5
04-01 16:12:07.051 7203-7203/com.example.test.myapplication D/jeff:
onPostExecute:计算结果 = 15
```

从结果看，依次回调 AsyncTask 的 onPreExecute、doInBackground、onProgressUpdate、onPostExecute 方法，其中 doInBackground 在子线程执行（7203-7341），其他放都执行在主线程（7203-7203）。

二、AsyncTask 回调方法

```
    /**
     * Override this method to perform a computation on a
background thread. The
     * specified parameters are the parameters passed to {@link
#execute}
     * by the caller of this task.
     *
     * This method can call {@link #publishProgress} to publish
updates
     * on the UI thread.
     *
     * @param params The parameters of the task.
     */
    @WorkerThread
    protected abstract Result doInBackground(Params... params);

    /**
     * Runs on the UI thread before {@link #doInBackground}.
```

```
     *
     */
    @MainThread
    protected void onPreExecute() {
    }

    /**
     * <p>Runs on the UI thread after {@link #doInBackground}. The
     * specified result is the value returned by {@link
#doInBackground}.</p>
     *
     * <p>This method won't be invoked if the task was cancelled.
</p>
     *
     * @param result The result of the operation computed by {@link
#doInBackground}.
     *
     */
    @SuppressWarnings({"UnusedDeclaration"})
    @MainThread
    protected void onPostExecute(Result result) {
    }

    /**
     * Runs on the UI thread after {@link #publishProgress} is
invoked.
     * The specified values are the values passed to {@link
#publishProgress}.
     *
     * @param values The values indicating progress.
     *
     */
    @SuppressWarnings({"UnusedDeclaration"})
    @MainThread
    protected void onProgressUpdate(Progress... values) {
    }

    /**
     * <p>Runs on the UI thread after {@link #cancel(boolean)} is
invoked and
     * {@link #doInBackground(Object[])} has finished.</p>
     *
     * <p>The default implementation simply invokes {@link
#onCancelled()} and
     * ignores the result. If you write your own implementation, do
not call
     * <code>super.onCancelled(result)</code>.</p>
     *
```

```
     * @param result The result, if any, computed in
     *               {@link #doInBackground(Object[])}, can be null
     */
    @SuppressWarnings({"UnusedParameters"})
    @MainThread
    protected void onCancelled(Result result) {
        onCancelled();
    }

    /**
     * <p>Applications should preferably override {@link
#onCancelled(Object)}.
     * This method is invoked by the default implementation of
     * {@link #onCancelled(Object)}.</p>
     *
     * <p>Runs on the UI thread after {@link #cancel(boolean)} is
invoked and
     * {@link #doInBackground(Object[])} has finished.</p>
     */
    @MainThread
    protected void onCancelled() {
    }
```

- **doInBackground**： 这是一个抽象方法，所以必须要重写，运行在工作线程，执行耗时操作，可以传递参数，在该方法中调用 publishProgress 方法来通知进度，onProgressUpdate 会接收进度。
- **onPreExecute**： 执行在 UI 线程，在 doInBackground 前执行，一般做一些准备工作。
- **onPostExecute**： 执行在 UI 线程，在 doInBackground 后执行，接收 doInBackground 返回值，需要注意的是，如果任务被取消，改方法将不会被回调。
- **onProgressUpdate**： 执行在 UI 线程，接收进度值，在 doInBackground 中调用 onProgressUpdate 方法后回调，如下载进度。
- **onCancelled(Result)**： 执行在 UI 线程，AsyncTask 调用 cancel 方法之后回调该方法，可以接收 doInBackground 返回值，即时没有执行完。
- **onCancelled**： 同 onCancelled(Result) 方法，区别在于不接收返回值。

三、任务状态

```
    public enum Status {
        /**
         * Indicates that the task has not been executed yet.
         */
        PENDING,
        /**
         * Indicates that the task is running.
         */
        RUNNING,
        /**
         * Indicates that {@link AsyncTask#onPostExecute} has
finished.
         */
        FINISHED,
    }
```

- **PENDING**： 未执行
- **RUNNING**： 执行中
- **FINISHED**： 任务结束

四、执行流程分析

1. 构造方法

```
    public AsyncTask(@Nullable Looper callbackLooper) {
        mHandler = callbackLooper == null || callbackLooper ==
Looper.getMainLooper()
            ? getMainHandler()
            : new Handler(callbackLooper);

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

Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                    //noinspection unchecked
                    result = doInBackground(mParams);
                    Binder.flushPendingCommands();
                } catch (Throwable tr) {
                    mCancelled.set(true);
                    throw tr;
                } finally {
                    postResult(result);
                }
                return result;
            }
        };

        mFuture = new FutureTask<Result>(mWorker) {
            @Override
            protected void done() {
                try {
                    postResultIfNotInvoked(get());
                } catch (InterruptedException e) {
                    android.util.Log.w(LOG_TAG, e);
                } catch (ExecutionException e) {
                    throw new RuntimeException("An error occurred
while executing doInBackground()",
                            e.getCause());
                } catch (CancellationException e) {
                    postResultIfNotInvoked(null);
                }
            }
        };
    }
```

三个很重要的变量，mHandler、mWorker、mFuture，

```
    private static class InternalHandler extends Handler {
        public InternalHandler(Looper looper) {
            super(looper);
        }

        @SuppressWarnings({"unchecked",
"RawUseOfParameterizedType"})
        @Override
        public void handleMessage(Message msg) {
            AsyncTaskResult<?> result = (AsyncTaskResult<?>)
msg.obj;
            switch (msg.what) {
                case MESSAGE_POST_RESULT:
                    // There is only one result
                    result.mTask.finish(result.mData[0]);
                    break;
                case MESSAGE_POST_PROGRESS:
                    result.mTask.onProgressUpdate(result.mData);
                    break;
            }
        }
    }
```

Handler 常用来多线程之间传递消息，AsyncTask 内部实现 InternalHandler，用来发送和处理消息 MESSAGE_POST_RESULT、MESSAGE_POST_PROGRESS，对应 AsyncTask 的回调方法 onProgressUpdate 和 onPostExecute，这两个方法都是要在 UI 线程回调的。

mWorker 是 WorkerRunnable 的实例，WorkerRunnable 实现了 Callable 接口，可以返回执行结果，耗时方法 doInBackground 在 WorkerRunnable 的 call 方法中执行，执行完后得到执行结果，执行 postResult 方法，将结果传递到 UI 线程。

mFuture 是 FutureTask 的实例，用来执行任务 mWorker，并重写了 done 方法，执行 postResultIfNotInvoked。FutureTask 这里不再展开，简单的说，就是 FutureTask 实现了 Runnable 和 Future 接口，可以执行 Runnable 和 Future 任务，可以返回执行结果。

```
    private void postResultIfNotInvoked(Result result) {
        final boolean wasTaskInvoked = mTaskInvoked.get();
        if (!wasTaskInvoked) {
            postResult(result);
        }
    }
```

FutureTask 的 done 方法在任务执行过程中最后回调，这里重写的 done 方法中的 postResultIfNotInvoked 方法 只有当任务没有正常调用时才会执行，如果正常调用， mTaskInvoked.set(true); 将忽略。

2. 任务执行 execute

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

    @MainThread
    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;
    }
```

AsyncTask 调用 execute 方法来执行任务，如 wt.execute(1, 5)。我们重点分析一下 executeOnExecutor 这个方法，其中第一个参数是 Executor，异步任务执行器，用来执行异步任务，默认用的是 sDefaultExecutor，且看 sDefaultExecutor 是如何生成的：

```
    private static volatile Executor sDefaultExecutor =
SERIAL_EXECUTOR;

    /**
     * An {@link Executor} that executes tasks one at a time in
serial
     * order.  This serialization is global to a particular
process.
     */
    public static final Executor SERIAL_EXECUTOR = new
SerialExecutor();


    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 = mTasks.poll()) != null) {
                THREAD_POOL_EXECUTOR.execute(mActive);
            }
        }
    }
```

sDefaultExecutor 是一个静态变量，所以当我们提交多个 AsyncTask 任务时，都是由同一个 sDefaultExecutor 进行调度的，sDefaultExecutor 由 SERIAL_EXECUTOR 直接复制，即 SerialExecutor 内部类实例，从字面意思来看是一个串行异步执行器，简单分析一下 SerialExecutor 的实现，看一下是不是串行执行的。

SerialExecutor 中维护了一个任务队列 mTasks，假设我们第一次执行任务 execute，这时将该任务 R1 插入到任务队列 mTasks 中去，因为是第一次执行，所以 mActive 为 null，所以接着会执行 scheduleNext 方法，mTasks 去除头部元素，赋值给 mActive，即我们刚添加的任务 R1，此时 mActive 肯定不为空，THREAD_POOL_EXECUTOR 来执行 mActive。如果此时 mActive 任务未执行完，又添加了一个任务 R2 ，同样会插入到 mTasks 中，不同的是 mActive 此时不为空，不会执行 scheduleNext，会一直等待 R1 任务完成，然后执行到 finally，即再次调用 scheduleNext，这时去除任务 R2并赋值给 mActive，然后执行 R2，如果此时有任务 R3、R4 ... 加入，将按照之前的流程执行，如果没有任务新增，R2 执行完就结束了。

这么看来确认是无论执行多少个任务，都是串行执行的。那么如果想让多个任务同时执行呢？这个问题后面再看。

继续回到执行流程上，如果当前状态不是 PENDING，将抛出异常。所以一个 AsyncTask 任务只能执行一次，如 wt.execute(1, 5) 只能执行一次。否则正常执行，将当前状态置位 RUNNING，执行回调 onPreExecute，然后执行任务 mFuture，即 WorkerRunnable 的 call 中执行 doInBackground。

3. 进度更新

在简述 AsyncTask 的回调方法是提到在 doInBackground 方法中可以调用 publishProgress 更新进度，就是通过 Handler 来处理的，

```
    @WorkerThread
    protected final void publishProgress(Progress... values) {
        if (!isCancelled()) {
            getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
                    new AsyncTaskResult<Progress>(this,
values)).sendToTarget();
        }
    }
```

4. 任务结束

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

判断任务是否被取消，如果已取消，调用 onCancelled，否则调用 onPostExecute，所以这两个方法只有一个会被调用。最后设置状态 FINISHED。

五、串行、并行

前面已经分析到 AsyncTask 的多任务默认是串行执行的，可能是 Google 处于性能的考虑，但在实际的很多场景是需要并行执行的，那我们怎么实现呢?

THREAD_POOL_EXECUTOR

```
    private static final int CPU_COUNT =
Runtime.getRuntime().availableProcessors();
    // We want at least 2 threads and at most 4 threads in the core
pool,
    // preferring to have 1 less than the CPU count to avoid
saturating
    // the CPU with background work
    private static final int CORE_POOL_SIZE = Math.max(2,
Math.min(CPU_COUNT - 1, 4));
    private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
    private static final int KEEP_ALIVE_SECONDS = 30;

    private static final ThreadFactory sThreadFactory = new
ThreadFactory() {
        private final AtomicInteger mCount = new AtomicInteger(1);

        public Thread newThread(Runnable r) {
            return new Thread(r, "AsyncTask #" +
mCount.getAndIncrement());
        }
    };

    private static final BlockingQueue<Runnable> sPoolWorkQueue =
            new LinkedBlockingQueue<Runnable>(128);

    /**
     * An {@link Executor} that can be used to execute tasks in
parallel.
     */
    public static final Executor THREAD_POOL_EXECUTOR;

    static {
        ThreadPoolExecutor threadPoolExecutor = new
ThreadPoolExecutor(
                CORE_POOL_SIZE, MAXIMUM_POOL_SIZE,
KEEP_ALIVE_SECONDS, TimeUnit.SECONDS,
                sPoolWorkQueue, sThreadFactory);
        threadPoolExecutor.allowCoreThreadTimeOut(true);
        THREAD_POOL_EXECUTOR = threadPoolExecutor;
    }
```

THREAD_POOL_EXECUTOR 其实就是 AsyncTask 内部实现的一个线程池，可以用来并行执行多任务。关于线程池我们这里不再展开，只要记住 THREAD_POOL_EXECUTOR 可以并行执行多任务。

并行任务

在示例中我们使用的是 AsyncTask 默认的 Executor 实例 sDefaultExecutor 来执行，wt.execute(1, 5)，我们来执行两个任务，

```
    WorkTask wt = new WorkTask();
    wt.execute(1, 5);

    WorkTask wt2 = new WorkTask();
    wt2.execute(6, 10);
```

执行结果：

```
04-01 19:20:15.791 11879-11879/com.example.test.myapplication
D/jeff: onPreExecute
04-01 19:20:15.791 11879-11879/com.example.test.myapplication
D/jeff: onPreExecute
04-01 19:20:15.791 11879-12345/com.example.test.myapplication
D/jeff: doInBackground:计算 1 到 5 之和
04-01 19:20:15.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 1
04-01 19:20:16.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 2
04-01 19:20:17.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 3
04-01 19:20:18.801 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 4
04-01 19:20:19.801 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 5
04-01 19:20:20.801 11879-11879/com.example.test.myapplication
D/jeff: onPostExecute:计算结果 = 15
04-01 19:20:20.801 11879-12355/com.example.test.myapplication
D/jeff: doInBackground:计算 6 到 10 之和
04-01 19:20:20.801 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 6
04-01 19:20:21.801 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 7
04-01 19:20:22.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 8
04-01 19:20:23.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 9
04-01 19:20:24.811 11879-11879/com.example.test.myapplication
D/jeff: onProgressUpdate: 10
04-01 19:20:25.811 11879-11879/com.example.test.myapplication
D/jeff: onPostExecute:计算结果 = 40
```

第一个任务执行完才开始执行第二个任务，显示是并行执行，也验证了我们之前的分析结论。

现在我们不使用默认的 Executor，把它换成 AsyncTask 中定义的 THREAD_POOL_EXECUTOR，修改代码如下，

```
    WorkTask wt = new WorkTask();
    wt.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, 1, 5);

    WorkTask wt2 = new WorkTask();
    wt2.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, 6, 10);
```

执行结果：

```
04-01 19:24:33.041 12674-12674/com.example.test.myapplication
D/jeff: onPreExecute
04-01 19:24:33.041 12674-12674/com.example.test.myapplication
D/jeff: onPreExecute
04-01 19:24:33.051 12674-12849/com.example.test.myapplication
D/jeff: doInBackground:计算 1 到 5 之和
04-01 19:24:33.051 12674-12850/com.example.test.myapplication
D/jeff: doInBackground:计算 6 到 10 之和
04-01 19:24:33.061 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 1
04-01 19:24:33.061 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 6
04-01 19:24:34.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 2
04-01 19:24:34.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 7
04-01 19:24:35.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 3
04-01 19:24:35.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 8
04-01 19:24:36.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 9
04-01 19:24:36.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 4
04-01 19:24:37.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 10
04-01 19:24:37.051 12674-12674/com.example.test.myapplication
D/jeff: onProgressUpdate: 5
04-01 19:24:38.061 12674-12674/com.example.test.myapplication
D/jeff: onPostExecute:计算结果 = 40
04-01 19:24:38.061 12674-12674/com.example.test.myapplication
D/jeff: onPostExecute:计算结果 = 15
```

显然，任务是并行执行的，所以 AsyncTask 也是可以用来进行多任务并行执行的。我们这里使用的是 AsyncTask 内部实现的线程池，当然我们也可以自己实现，这里就不在详述。

结束