# [AsyncTask异步任务类](https://www.cnblogs.com/yc211/p/9878621.html)

#### 目录介绍

* 01.先看下AsyncTask用法
* 02.AsyncTask源码深入分析
  + 2.1 构造方法源码分析
  + 2.2 看execute(Params... params)方法
  + 2.3 mWorker和mFuture的创建过程
* 03.异步机制的实现
* 04.不同的SDK版本区别
* 05.AsyncTask的缺陷和问题
  + 5.1 AsyncTask对应线程池
  + 5.2 AsyncTask生命周期问题
  + 5.3 AsyncTask内存泄漏问题
  + 5.4 AsyncTask结果丢失问题
  + 5.5 AsyncTask并行还是串行问题

### 好消息

* 博客笔记大汇总【16年3月到至今】，包括Java基础及深入知识点，Android技术博客，Python学习笔记等等，还包括平时开发中遇到的bug汇总，当然也在工作之余收集了大量的面试题，长期更新维护并且修正，持续完善……开源的文件是markdown格式的！同时也开源了生活博客，从12年起，积累共计47篇[近20万字]，转载请注明出处，谢谢！
* **链接地址：**[**https://github.com/yangchong211/YCBlogs**](https://github.com/yangchong211/YCBlogs)
* 如果觉得好，可以star一下，谢谢！当然也欢迎提出建议，万事起于忽微，量变引起质变！

### 问题答疑

* AsyncTask是如何实现异步机制的，底层原理是什么？
* AsyncTask调用execute方法时，如果不是运行在主线程中会出现什么情况，如何解决？
* 为什么异步任务对象不能执行多次，即不能创建一个对象执行多次execute方法？
* doInBackground这个方法可以做什么操作？它是在主线程中还是工作线程中？为什么？
* AsyncTask任务是否可以被中途取消？为什么？
* AsyncTask对应线程池是如何操作的？它有什么弊端，为什么现在几乎很少用呢？
* AsyncTask的执行策略是并行还是串行的？
* **带着问题去看这篇文章，相信看完之后你对异常AsyncTask有了初步理解……**

### 01.先看下AsyncTask用法

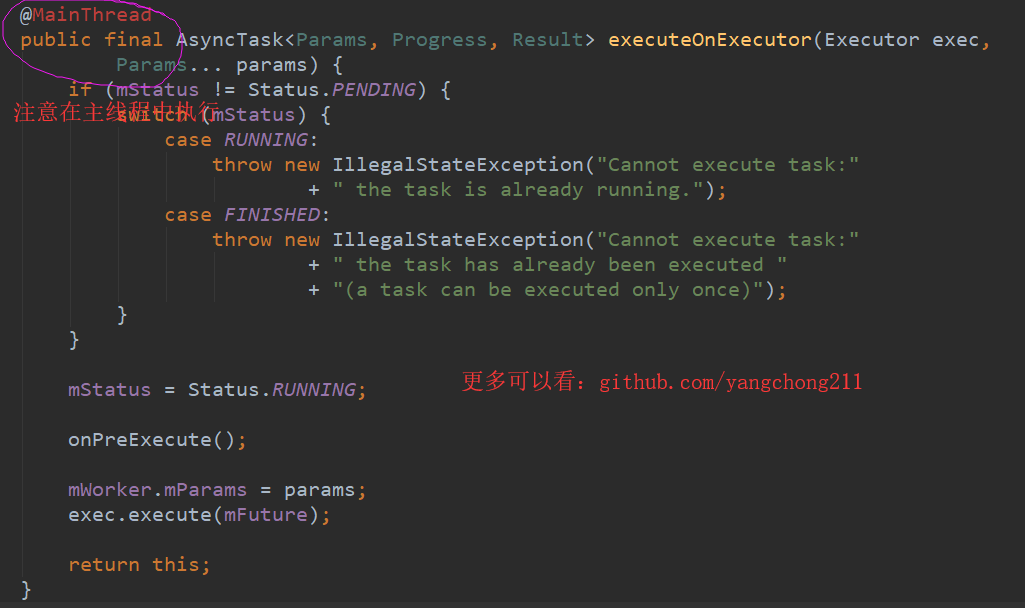
* 来看一下AsyncTask的基本使用，代码如下所示
  + 定义了自己的MyAsyncTask并继承自AsyncTask；并重写了其中的是哪个回调方法：onPreExecute()，onPostExecute（），doInBackground();
* class MyAsyncTask extends AsyncTask<Integer, Integer, Integer> {
* @Override
* protected void onPreExecute() {
* super.onPreExecute();
* Log.i(TAG, "onPreExecute...(开始执行后台任务之前)");
* }
* @Override
* protected void onPostExecute(Integer i) {
* super.onPostExecute(i);
* Log.i("TAG", "onPostExecute...(开始执行后台任务之后)");
* }
* @Override
* protected Integer doInBackground(Integer... params) {
* Log.i(TAG, "doInBackground...(开始执行后台任务)");
* return 0;
* }
* }
* 开始调用异步任务
* new MyAsyncTask().execute();

### 02.AsyncTask源码深入分析

#### 2.1 构造方法源码分析

* 源代码如下所示，主要是看AsyncTask(@Nullable Looper callbackLooper)中的代码
  + 这里面只是初始化了两个成员变量：mWorker和mFuture他们分别是：WorkerRunnable和FutureTask，对于熟悉java的逗比应该知道这两个类其实是java里面线程池先关的概念。
  + 异步任务的构造方法主要用于初始化线程池先关的成员变量
* //创建一个新的异步任务。必须在UI线程上调用此构造函数
* public AsyncTask() {
* this((Looper) null);
* }
* //创建一个新的异步任务。必须在UI线程上调用此构造函数
* public AsyncTask(@Nullable Handler handler) {
* this(handler != null ? handler.getLooper() : null);
* }
* 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);
* }
* }
* };
* }

#### 2.2 看execute(Params... params)方法

* 看一下execute方法
  + 发现该方法中添加一个@MainThread的注解，通过该注解，可以知道我们在执行AsyncTask的execute方法时，只能在主线程中执行
* @MainThread
* public final AsyncTask<Params, Progress, Result> execute(Params... params) {
* return executeOnExecutor(sDefaultExecutor, params);
* }
* 如果execute方法不是运行在主线程中会出现什么情况呢？
  + 执行，但是并没有什么区别，程序还是可以正常执行。但是onPreExecute方法是与开始执行的execute方法是在同一个线程中的，所以如果在子线程中执行execute方法，一定要确保onPreExecute方法不执行刷新UI的方法，否则将会抛出异常。
* new Thread(new Runnable() {
* @Override
* public void run() {
* Log.i("tag", Thread.currentThread().getId() + "");
* new MAsyncTask().execute();
* }
* }).start();
* Log.i("tag", "mainThread:" + Thread.currentThread().getId() + "");
* @Override
* protected void onPreExecute() {
* super.onPreExecute();
* //更新UI
* title.setText("潇湘剑雨");
* Log.i(TAG, "onPreExecute...(开始执行后台任务之前)");
* }
  + 异常如下所示，在子线程中执行execute方法，那么这时候如果在onPreExecute方法中刷新UI，会报错，即子线程中不能更新UI。
* Process: com.example.aaron.helloworld, PID: 659
* android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
* 接着看看executeOnExecutor这个方法源码
  + 具体的内部实现方法里：首先判断当前异步任务的状态，其内部保存异步任务状态的成员变量mStatus的默认值为Status.PENDING,所以第一次执行的时候并不抛出这两个异常，那么什么时候回进入这个if判断并抛出异常呢，通过查看源代码可以知道，当我们执行了execute方法之后，如果再次执行就会进入这里的if条件判断并抛出异常
  + 在executeOnExecutor中若没有进入异常分之，则将当前异步任务的状态更改为Running，然后回调onPreExecute()方法，这里可以查看一下onPreExecute方法其实是一个空方法，主要就是为了用于我们的回调实现，同时这里也说明了onPreExecute（）方法是与execute方法的执行在同一线程中。
* 然后将execute方法的参数赋值给mWorker对象那个，最后执行exec.execute(mFuture)方法，并返回自身。
  + 
* 模拟测试一下抛出异常的操作
  + 看到我们定义了一个AsyncTask的对象，并且每次执行点击事件的回调方法都会执行execute方法，当我们点击第一次的时候程序正常执行，但是当我们执行第二次的时候，程序就崩溃了。
* final MyAsyncTask mAsyncTask = new MyAsyncTask();
* title.setOnClickListener(new View.OnClickListener() {
* @Override
* public void onClick(View v) {
* new Thread(new Runnable() {
* @Override
* public void run() {
* Log.i("tag", Thread.currentThread().getId() + "");
* mAsyncTask.execute();
* }
* }).start();
* Log.i("tag", "mainThread:" + Thread.currentThread().getId() + "");
* }
* });
  + 若这时候第一次执行的异步任务尚未执行完成则会抛出异常：
* Cannot execute task:the task is already running.
  + 若第一次执行的异步任务已经执行完成，则会抛出异常：
* Cannot execute task:the task has already been executed (a task can be executed only once)
* 然后看一下exec.execute(mFuture)的实现
  + 这里的exec其实是AsyncTask定义的一个默认的Executor对象：
* private static volatile Executor sDefaultExecutor = SERIAL\_EXECUTOR;
  + 那么，SERIAL\_EXECUTOR又是什么东西呢？
* public static final Executor SERIAL\_EXECUTOR = new SerialExecutor();
  + 继续查看SerialExecutor的具体实现：
* 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);
* }
* }
* }
  + 可以发现其继承Executor类其内部保存着一个Runnable列表，即任务列表，在刚刚的execute方法中执行的exec.execute(mFuture)方法就是执行的这里的execute方法。
  + 这里具体看一下execute方法的实现：
    - 1）首先调用的是mTasks的offer方法，即将异步任务保存至任务列表的队尾
    - 2）判断mActive对象是不是等于null，第一次运行是null，然后调用scheduleNext()方法
    - 3）在scheduleNext()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。
    - 4）在这之后如果再有新的任务被执行时就等待上一个任务执行完毕后才会得到执行，所以说同一时刻只会有一个线程正在执行。
    - 5）这里的THREAD\_POOL\_EXECUTOR其实是一个线程池对象。

#### 2.3 构造方法中mWorker和mFuture的创建过程

* 看一下执行过程中mWorker的执行逻辑：
  + 可以看到在执行线程池的任务时，我们回调了doInBackground方法，这也就是我们重写AsyncTask时重写doInBackground方法是后台线程的原因。
* mWorker = new WorkerRunnable<Params, Result>() {
* public Result call() throws Exception {
* mTaskInvoked.set(true);
* Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);
* //noinspection unchecked
* Result result = doInBackground(mParams);
* Binder.flushPendingCommands();
* return postResult(result);
* }
* };
* 看一下执行过程中mFuture的执行逻辑
* 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);
* }
* }
* };
  + 这里具体看一下postResultIfNotInvoked方法：
* private void postResultIfNotInvoked(Result result) {
* final boolean wasTaskInvoked = mTaskInvoked.get();
* if (!wasTaskInvoked) {
* postResult(result);
* }
* }
  + 其内部还是调用了postResult方法：
* private Result postResult(Result result) {
* @SuppressWarnings("unchecked")
* Message message = getHandler().obtainMessage(MESSAGE\_POST\_RESULT,
* new AsyncTaskResult<Result>(this, result));
* message.sendToTarget();
* return result;
* }
  + 这里可以看到起调用了内部的Handler对象的sendToTarget方法，发送异步消息

### 03.异步机制的实现

* 看AsyncTask内部定义了一个Handler对象
  + 内部的handleMessage方法，有两个处理逻辑，分别是：更新进入条和执行完成，这里的更新进度的方法就是我们重写AsyncTask方法时重写的更新进度的方法，这里的异步任务完成的消息会调用finish方法
* private static class InternalHandler extends Handler {
* public InternalHandler() {
* super(Looper.getMainLooper());
* }
* @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
* @Override
* public void handleMessage(Message msg) {
* AsyncTaskResult<?> result = (AsyncTaskResult<?>) 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;
* }
* }
* }
* 然后看看调用finish方法做了什么
  + 首先会判断当前任务是否被取消，若被取消的话则直接执行取消的方法，否则执行onPostExecute方法，也就是我们重写AsyncTask时需要重写的异步任务完成时回调的方法。
* private void finish(Result result) {
* if (isCancelled()) {
* onCancelled(result);
* } else {
* onPostExecute(result);
* }
* mStatus = Status.FINISHED;
* }
* 既然有处理消息的，那么肯定有发送消息的。
  + 可以从构造方法中看到，当通过执行doInBackground方法拿到结果后，最后在finally执行发送该消息逻辑
* 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MESSAGE\_POST\_PROGRESS这个消息发送是处理进度，需要在工作线程中
* @Override
* protected ReusableBitmap doInBackground(Void... params) {
* // enqueue the 'onDecodeBegin' signal on the main thread
* publishProgress();
* return decode();
* }
* @WorkerThread
* protected final void publishProgress(Progress... values) {
* if (!isCancelled()) {
* getHandler().obtainMessage(MESSAGE\_POST\_PROGRESS,
* new AsyncTaskResult<Progress>(this, values)).sendToTarget();
* }
* }

### 04.不同的SDK版本区别

* 调用AsyncTask的execute方法不能立即执行程序的原因析及改善方案通过查阅官方文档发现，AsyncTask首次引入时，异步任务是在一个独立的线程中顺序的执行，也就是说一次只执行一个任务，不能并行的执行，从1.6开始，AsyncTask引入了线程池，支持同时执行5个异步任务，也就是说时只能有5个线程运行，超过的线程只能等待，等待前的线程某个执行完了才被调度和运行。换句话说，如果个进程中的AsyncTask实例个数超过5个，那么假如前5都运行很长时间的话，那么第6个只能等待机会了。这是AsyncTask的一个限制，而且对于2.3以前的版本无法解决。如果你的应用需要大量的后台线程去执行任务，那么只能放弃使用AsyncTask，自己创建线程池来管理Thread。不得不说，虽然AsyncTask较Thread使用起来方便，但是它最多只能同时运行5个线程，这也大大局限了它的作用，你必须要小心设计你的应用，错开使用AsyncTask时间，尽力做到分时，或者保证数量不会大于5个，否就会遇到上次提到的问题。可能是Google意识到了AsynTask的局限性了，从Android3.0开始对AsyncTask的API做出了一些调整：每次只启动一个线程执行一个任务，完了之后再执行第二个任务，也就是相当于只有一个后台线在执行所提交的任务。

### 05.AsyncTask的缺陷和问题

#### 5.1 AsyncTask对应线程池

* Asynctask对应的线程池ThreadPoolExecutr都是进程范围内共享的，都是static的，所以是Asynctask控制着进程范围内所有的子类实例。由于这个限制的存在，当使用默认线程池时，如果线程数超过线程池的最大容量，线程池就会爆掉(3.0后默认串行执行，不会出现个问题)。针对这种情况，可以尝试自定义线程池，配合Asynctask使用。
* 关于默认线程池：
  + AsyncTask里面线程池是一个核心线程数为CPU + 1，最大线程数为CPU \* 2 + 1，工作队列长度为128的线程池，线程等待队列的最大等待数为28，但是可以自定义线程池。线程池是由AsyncTask来处理的，线程池允许tasks并行运行，需要注意的是并发情况下数据的一致性问题，新数据可能会被老数据覆盖掉类似volatile变量。所以希望tasks能够串行运行的话，使用SERIAL\_EXECUTOR。

#### 5.2 AsyncTask生命周期问题

* 很多开发者会认为一个在Activity中创建的AsyncTask随着Activity的销毁而销毁。然而事实并非如此。AsynTask会一直执行，直到doInBackground()方法执行完毕，然后，如果cancel(boolean)被调用,那么onCancelled(Result result)方法会被执行；否则，执行onPostExecuteResult result)方法。如果我们的Activity销毁之前，没有取消AsyncTask，这有可能让我们的应用崩溃(crash)。因为它想要处理的view已经不存在了。所以，我们是必须确保在销毁活动之前取消任务。总之，我们使用AsyncTask需要确保AsyncTask正确的取消。

#### 5.3 AsyncTask内存泄漏问题

* 如果AsyncTask被声明为Activity的非静态的内部类，那么AsyncTask会保留一个对Activity的引用。如果Activity已经被销毁，AsyncTask的后台线程还在执行，它将续在内存里保留这个引用，导致Activity无法被回收，引起内存泄漏。

#### 5.4 AsyncTask结果丢失问题

* 屏幕旋转或Activity在后台被系统杀掉等情况会导致Actvity的重新创建，之前运行的AsyncTask会持有一个之前Activity的引用，这个引用已经无效，这时调用onPostExecute()再去更新界面将不再生效。

#### 5.5 AsyncTask并行还是串行问题

* 在Android1.6之前的版本，AsyncTask是串行的，在1.6-2.3的版本，改成了并行的。在2.3之后的版本又做了修改，可以支持并行和串行，当想要串行执行时，直接行execute()方法，如果需要并行执行时，执行executeOnExecutor(Executor)。

### 关于其他内容介绍

#### 01.关于博客汇总链接

* 1.[技术博客汇总](https://www.jianshu.com/p/614cb839182c)
* 2.[开源项目汇总](https://blog.csdn.net/m0_37700275/article/details/80863574)
* 3.[生活博客汇总](https://blog.csdn.net/m0_37700275/article/details/79832978)
* 4.[喜马拉雅音频汇总](https://www.jianshu.com/p/f665de16d1eb)
* 5.[其他汇总](https://www.jianshu.com/p/53017c3fc75d)

#### 02.关于我的博客

* 我的个人站点：[www.yczbj.org](http://www.yczbj.org/)，[www.ycbjie.cn](http://www.ycbjie.cn/)
* github：<https://github.com/yangchong211>
* 知乎：<https://www.zhihu.com/people/yang-chong-69-24/pins/posts>
* 简书：<http://www.jianshu.com/u/b7b2c6ed9284>
* csdn：<http://my.csdn.net/m0_37700275>
* 喜马拉雅听书：<http://www.ximalaya.com/zhubo/71989305/>
* 开源中国：<https://my.oschina.net/zbj1618/blog>
* 泡在网上的日子：<http://www.jcodecraeer.com/member/content_list.php?channelid=1>
* 邮箱：yangchong211@163.com
* 阿里云博客：<https://yq.aliyun.com/users/article?spm=5176.100-> 239.headeruserinfo.3.dT4bcV
* segmentfault头条：<https://segmentfault.com/u/xiangjianyu/articles>