# AsyncTask的使用——四个方法？

AsyncTask 运用的场景就是我们需要进行一些耗时的操作，耗时操作完成后更新主线程，或者在操作过程中对主线程的UI进行更新。

@Deprecated 这个类已经被弃用了。

被弃用后，Android给出了两个替代的建议：

1）java.util.concurrent包下的相关类，如Executor，ThreadPoolExecutor，FutureTask。

2）kotlin并发工具，那就是协程 - Coroutines了。

## （1）了解AsyncTask异步任务

目的是让与UI线程交互的子线程变得更容易。

先来看看AsyncTask的定义：

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

**三种泛型类型分别代表“启动任务执行的输入参数”、“后台任务执行的进度”、“后台计算结果的类型”。**在特定场合下，并不是所有类型都被使用，如果没有被使用，可以用java.lang.Void类型代替。

一个异步任务的执行一般包括以下几个步骤：

1.执行异步任务：execute(Params... params)，执行一个异步任务，需要我们在代码中调用此方法，触发异步任务的执行。

2.执行后台任务前调用：onPreExecute()，在execute(Params... params)被调用后立即执行，一般用来在执行后台任务前对UI做一些标记。

3.执行后台任务：doInBackground(Params... params)，在onPreExecute()完成后立即执行，用于执行较为费时的操作，此方法将接收输入参数和返回计算结果。在执行过程中可以调用publish Progress(Progress... values)来更新进度信息。

4.更新进度信息：onProgressUpdate(Progress... values)，在调用publish Progress(Progress... values)时，此方法被执行，直接将进度信息更新到UI组件上。

5.任务结束回调：onPostExecute(Result result)，当后台操作结束时，此方法将会被调用，计算结果将做为参数传递到此方法中，直接将结果显示到UI组件上。

在使用的时候，有几点需要格外注意：

1.异步任务的实例必须在UI线程中创建。

2.execute(Params... params)方法必须在UI线程中调用。

3.不要手动调用onPreExecute()，doInBackground(Params... params)，

onProgressUpdate(Progress... values)，onPostExecute(Result result)这几个方法。

4.不能在doInBackground(Params... params)中更改UI组件的信息。

5.一个任务实例只能执行一次，如果执行第二次将会抛出异常。

实例：

public class MainActivity extends Activity {

private static final String TAG = "ASYNC\_TASK";

private Button execute;

private Button cancel;

private ProgressBar progressBar;

private TextView textView;

private MyTask mTask;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

execute = (Button) findViewById(R.id.execute);

execute.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

//注意每次需new一个实例,新建的任务只能执行一次,否则会出现异常

mTask = new MyTask();

mTask.execute("http://www.baidu.com");

execute.setEnabled(false);

cancel.setEnabled(true);

}

});

cancel = (Button) findViewById(R.id.cancel);

cancel.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

//取消一个正在执行的任务,onCancelled方法将会被调用

mTask.cancel(true);

}

});

progressBar = (ProgressBar) findViewById(R.id.progress\_bar);

textView = (TextView) findViewById(R.id.text\_view);

}

private class MyTask extends AsyncTask<String, Integer, String> {

//onPreExecute方法用于在执行后台任务前做一些UI操作

@Override

protected void onPreExecute() {

Log.i(TAG, "onPreExecute() called");

textView.setText("loading...");

}

//doInBackground方法内部执行后台任务,不可在此方法内修改UI

@Override

protected String doInBackground(String... params) {

Log.i(TAG, "doInBackground(Params... params) called");

try {

HttpClient client = new DefaultHttpClient();

HttpGet get = new HttpGet(params[0]);

HttpResponse response = client.execute(get);

if (response.getStatusLine().getStatusCode() == HttpStatus.SC\_OK) {

HttpEntity entity = response.getEntity();

InputStream is = entity.getContent();

long total = entity.getContentLength();

ByteArrayOutputStream baos = new ByteArrayOutputStream();

byte[] buf = new byte[1024];

int count = 0;

int length = -1;

while ((length = is.read(buf)) != -1) {

baos.write(buf, 0, length);

count += length;

//调用publishProgress公布进度,最后onProgressUpdate方法将被执行

publishProgress((int) ((count / (float) total) \* 100));

//为了演示进度,休眠500毫秒

Thread.sleep(500);

}

return new String(baos.toByteArray(), "gb2312");

}

} catch (Exception e) {

Log.e(TAG, e.getMessage());

}

return null;

}

//onProgressUpdate方法用于更新进度信息

@Override

protected void onProgressUpdate(Integer... progresses) {

Log.i(TAG, "onProgressUpdate(Progress... progresses) called");

progressBar.setProgress(progresses[0]);

textView.setText("loading..." + progresses[0] + "%");

}

//onPostExecute方法用于在执行完后台任务后更新UI,显示结果

@Override

protected void onPostExecute(String result) {

Log.i(TAG, "onPostExecute(Result result) called");

textView.setText(result);

execute.setEnabled(true);

cancel.setEnabled(false);

}

//onCancelled方法用于在取消执行中的任务时更改UI

@Override

protected void onCancelled() {

Log.i(TAG, "onCancelled() called");

textView.setText("cancelled");

progressBar.setProgress(0);

execute.setEnabled(true);

cancel.setEnabled(false);

}

}

}

---------------------------------------------------------------------------------------------------------------------------------

AsyncTask是对Thread+Handler良好的封装，在android.os.AsyncTask代码里仍然可以看到Thread和Handler的踪迹。下面就向大家详细介绍一下AsyncTask的执行原理。

按住取消之后，可以看到onCancelled()方法将会被调用，onPostExecute(Result result)方法将不再被调用。

---------------------------------------------------------------------------------------------------------------------------------

概括来说，当我们调用execute(Params... params)方法后，execute方法会调用onPreExecute()方法，然后由ThreadPoolExecutor实例sExecutor执行一个FutureTask任务，这个过程中doInBackground(Params... params)将被调用，如果被开发者覆写的doInBackground(Params... params)方法中调用了publishProgress(Progress... values)方法，则通过InternalHandler实例sHandler发送一条MESSAGE\_POST\_PROGRESS消息，更新进度，sHandler处理消息时onProgressUpdate(Progress... values)方法将被调用；如果遇到异常，则发送一条MESSAGE\_POST\_CANCEL的消息，取消任务，sHandler处理消息时onCancelled()方法将被调用；如果执行成功，则发送一条MESSAGE\_POST\_RESULT的消息，显示结果，sHandler处理消息时onPostExecute(Result result)方法被调用。

经过上面的介绍，相信朋友们都已经认识到AsyncTask的本质了，它对Thread+Handler的良好封装，减少了开发者处理问题的复杂度，提高了开发效率，希望朋友们能多多体会一下。

---------------------------------------------------------------------------------------------------------------------------------

# 2、AsyncTask内存泄漏问题--弱引用、静态内部类、cancel()；

（1）一个AsyncTask对象只能创建一次,也就是说只能调用一次excute()方法,否则会报运行时异常。

（2）AsyncTask必须在主线程中创建,在主线程中加载.

（3）内存泄漏

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

（4） 结果丢失

屏幕旋转或Activity在后台被系统杀掉等情况会导致Activity的重新创建，之前运行的AsyncTask（非静态的内部类）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。

1. 使用弱引用和静态内部类的方案

public class AsyncTaskActivity extends Activity {

private AsyncTask task;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

/\* 用匿名内部类的方式创建\*/

task = new AsyncTask() {

@Override

protected Object doInBackground(Object[] params) {

for(int i=0;i<10;i++){

Log.i("task","i="+i);

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

if(isCancelled()){

break;

}

}

return null;

}

@Override

protected void onPostExecute(Object o) {

super.onPostExecute(o);

Log.i("task","执行结束了");

}

@Override

protected void onCancelled() {

super.onCancelled();

Log.i("task","执行了取消");

}

};

task.execute();

//从log可以看到，上个task和下面的task是串行执行的

new MyTask(this).execute();

}

@Override

protected void onDestroy() {

super.onDestroy();

task.cancel(true);

//这里不做取消MyTask的任务是为了验证weakReference是否起作用

}

public void doSomething(){

//为保险，还是需要判断下当前activity是否已经销毁，因为weakReference修饰的对象并不是马上就能被回收

Log.i("AsyncActivity","异步任务完成，更新UI");

}

static class MyTask extends AsyncTask<String,Integer,String>{

private WeakReference<Activity> weakAty;

public MyTask(Activity activity){

weakAty = new WeakReference<Activity>(activity);

}

@Override

protected String doInBackground(String... params) {

for(int i=0;i<100;i++){

Log.i("Mytask","i="+i);

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

if(isCancelled()){

break;

}

}

return null;

}

@Override

protected void onPostExecute(String s) {

super.onPostExecute(s);

Log.i("Mytask","执行结束了");

AsyncTaskActivity mActivity;

if((mActivity= (AsyncTaskActivity) weakAty.get())!=null){

mActivity.doSomething();

}

}

@Override

protected void onCancelled() {

super.onCancelled();

Log.i("Mytask","执行了取消");

}

}

}

------------------------------------------------------------------------------------------------------------------------------

以上方案也是存在一个问题：

静态类中使用的变量需要静态，有何隐患。

静态类是一直在内存中存在的，占用一定的内存，所以静态变量和类不能过多，否则会导致内存的不足。 但必要时也是不可避免的。

# 3、Cancel()方法的使用

AsyncTask不会不考虑结果而直接结束一个线程。调用cancel()其实是给AsyncTask设置一个"canceled"状态。这取决于你去检查AsyncTask是否已经取消，之后决定是否终止你的操作。对于mayInterruptIfRunning——它所作的只是向运行中的线程发出interrupt()调用。在这种情况下，你的线程是不可中断的，也就不会终止该线程。

那么该如何结束线程呢？

可见.cancel()是给AsyncTask设置一个"canceled"的状态，那么想要终止异步任务，就需要在异步任务当中结束。

@Override

public void onProgressUpdate(Integer... value) {

// 判断是否被取消

if(isCancelled()) return;

.........

}

@Override

protected Integer doInBackground(Void... mgs) {

// Task被取消了，马上退出

if(isCancelled()) return null;

.......

// Task被取消了，马上退出

if(isCancelled()) return null;

}

...

另外结束异步任务的条件：

if(loadAsyncVedio!=null && !loadAsyncVedio.isCancelled()

&& loadAsyncVedio.getStatus() == AsyncTask.Status.RUNNING){

loadAsyncVedio.cancel(true);

loadAsyncVedio = null;

}

loadAsyncVedio（异步任务）

如此，便可以有效及时的结束异步任务