AsyncTask详解

版本：2018.9.9-1(1:0)

问题汇总

1. AsyncTask是什么
2. AsyncTask为什么只适合做耗时较短的任务?
3. AsyncTask基本使用
4. AysncTask的优点(3)
5. AysncTask的缺点(4)
6. AsyncTask的内存泄漏如何解决?
7. AsyncTask和Activity生命周期不同步
8. AsyncTask的任务结果会丢失
9. AsyncTask的三个泛型参数的作用?
10. AsyncTask的4个核心方法的作用?
11. AsyncTask的onPreExecute
12. AsyncTask的doInBackground
13. AsyncTask的onProgressUpdate
14. AsyncTask的onPostExecute
15. AysncTask Android3.0之前的特点
16. AysncTask Android7.0版本的特点
17. AsyncTask如何指定采用其他线程池去执行任务?
18. AsyncTask的机制原理概述
19. AsyncTask的内部的静态线程池?
20. AsyncTask(API28)默认采用的串行线程池内部机制是怎么样的?
21. AsyncTask内部的线程池的特点?
22. AsyncTask内部的线程池采用的阻塞队列LinkedBlockingQueue有什么特点?
23. ArrayDeque的特点
24. AsyncTask如何设置采用并发方法去执行任务?
25. AsyncTask内部只有一个静态线程池，如何做到AsyncTask1串行执行任务和AsyncTask2并发执行?
26. AsyncTask的构造方法和内部机制
27. onProgressUpdate、onPostExecute、onCancelled都执行在主线程(API28)?
28. 为什么AsyncTask要使用实现Callable接口的方式去创建线程?
29. onPostExecute()获取的返回值是哪里来的?
30. execute的内部源码和流程
31. onPreExecute()一定执行在主线程?
32. AysncTask的流程

33. AsyncTask的cancel()原理
34. onCancelled()方法什么时候会被回调?
35. cancel(boolean mayInterruptIfRunning)的参数有什么用?
36. AsyncTask并没有串行线程池

基本使用(17)

1、AsyncTask是什么

> 1. 诞生背景：线程执行耗时操作，任务完成后去更新UI，可以通过 `Handler` 实现。如果同时有多个任务同时执行，就会出现代码臃肿。
> 2. AsyncTask用于 使得异步任务更加简单,代码更清晰
> 3. AsyncTask是一个 抽象的泛型类
> 4. 只适合做耗时比较短的操作。
> 5. 总之是一个封装了线程池和handler的异步框架。

2、AsyncTask为什么只适合做耗时较短的任务?

> 1. AsyncTask和Activity生命周期不同步
> 2. 内存泄漏：长时间持有外部类的引用。

3、AsyncTask基本使用

> 1-继承AsyncTask。指定泛型参数，实现4个核心方法。

```
public class MyAsyncTask extends AsyncTask<String, Integer, String> {
    private static final String TAG = "MyAsyncTask";
    /**================================
     * 1、任务开始前的准备工作。
     *      UI线程中
     *=================================*/
    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        Log.i(TAG, "加载的准备工作");
    }
    /**================================
     * 2、后台任务。
     *     1. 线程池中
     *     2. 调用publishProgress去更新进度
     *=================================*/
    @Override
    protected String doInBackground(String... strings) {
        Log.i(TAG, strings[0]);
        for (int i = 0; i < 20; i++) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            publishProgress(100 * (i + 1)/20);
        }
        return "加载成功";
    }
    /**================================
     * 3、更新进度。
     *     1. 处于UI线程。
     *     2. 参数是AsyncTask的第二个参数，为publishProgress中传递的参数
     *=================================*/
    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
        Log.i(TAG, "progress = " + values[0]);
    }
    /**================================
     * 4、收尾工作。任务完成后回调。
     *     1.  处于UI线程。
     *=================================*/
    @Override
    protected void onPostExecute(String s) {
        super.onPostExecute(s);
        Log.i(TAG, s);
    }
}
```

> 2-使用AsyncTask

```
MyAsyncTask asyncTask = new MyAsyncTask();
asyncTask.execute("任务开始");

// 取消任务，会打断正在执行的任务。
asyncTask.cancel(true);
// 取消任务，不会打断
asyncTask.cancel(false);
```

4、AysncTask的优点(3)

> 优点：简单，快捷，过程可控

5、AysncTask的缺点(4)

> 1. 生命周期：Activity中的AsyncTask不会随Acitivity的销毁而销毁。AsyncTask会一直运行到 `doInBackground()` 方法执行完毕，然后会执行 `onPostExecute()` 方法。如果Acitivity销毁时，没有执行 `onCancelled()` ，AysncTask在结束后操作UI时出现崩溃
> 2. 内存泄漏：如果AsyncTask被声明为Acitivity的非静态的内部类，会拥有Activity的引用。在Acitivity已经被销毁后，AsyncTask后台线程仍在执行，则会导致Acitivity无法被回收，造成内存泄漏。
> 3. 结果丢失: 屏幕旋转或Acitivity在后台被系统杀掉等情况下，Acitivity会重新创建。之前运行的AsyncTask持有的Acitivity引用会失效，导致更新UI的操作无效。

4. 并行还是串行：1.6之前，AsyncTask是串行的；在1.6至2.3版本，AsyncTask是并行的；在3.0及以上版本中，AsyncTask支持串行和并行(默认串行)-execute()方法就是串行执行，executeOnExecutor(Executor)就是并发执行

6、AsyncTask的内存泄漏如何解决?

1. 原因是内部类持有外部类的引用
2. 需要采用静态内部类
3. 并且在组件结束后，调用 `asyncTask.cancel(true);` 进行取消

生命周期

7、AsyncTask和Activity生命周期不同步

1. 并不会更随Activity的生命周期而销毁
2. 必须要通过 `cancel` 才能停止AsyncTask
3. 如果Activity已经停止，但是AsyncTask还在执行并且去更新UI，就会出现空指针异常。

8、AsyncTask的任务结果会丢失

1. 旋转屏幕和内存不足会导致Activity的重建。AsyncTask会持有之前控件的引用，导致崩溃。(结果丢失)
2. 如果AsyncTask持有了之前Activity的引用，因此不会Activity不会被销毁，但是会导致结果丢失。

三个泛型参数

9、AsyncTask的三个泛型参数的作用?

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

1. Params: 任务开始时参数类型
    1. execute()的入参类型
    2. doInBackground()的入参类型
2. Progress：更新进度时传递的参数类型
    1. 更新进度时传递的参数类型: publishProgress()的入参类型
    2. onProgressUpdate()的入参类型
3. Result：任务结束时的参数类型
    1. doInBackground()的返回值类型
    2. onPostExecute()的入参类型
4. 不需要的参数用void即可

四个核心方法

10、AsyncTask的4个核心方法的作用?

具有4个核心方法：onPreExecute()、doInBackground(Params... params)、onProgresUpdate(Progress... value)、onPostExecute(Result result)

1. onPreExecute(): 运行在执行execute()的线程中。任务前的准备工作(对UI进行一些标记等)
2. doInBackground()：线程池中执行，在onPreExecute后执行耗时操作。过程中可以调用 `publishProgress` 更新进度
3. onProgresUpdate: 主线程中。在 `doInBackground` 中调用 `publishProgress` 后，会调用该方法，会在UI上更新进度。
4. onPostExecute: 主线程中。任务执行完成后的收尾工作，result参数就是doInBackground最后返回的值

11、AsyncTask的onPreExecute

```
@Override
protected void onPreExecute() {
    super.onPreExecute();
    Log.i(TAG, "加载的准备工作");
}
```

1. 任务开始前的初始化工作
2. UI线程

12、AsyncTask的doInBackground

```
@Override
protected String doInBackground(String... strings) {
    Log.i(TAG, strings[0]);
    for (int i = 0; i < 20; i++) {
        publishProgress(i);
    }
    return "加载成功";
}
```

> 1. 后台耗时操作。
> 2. 线程池中：不能操作UI
> 3. 入参类型是AsynTask的第一个泛型参数Params指定，为execute()传递的参数。
> 4. 返回值是onPostExecute的入参。第三个泛型参数Result指定。
> 5. 更新进度条需要调用：publishProgress

13、AsyncTask的onProgressUpdate

```
@Override
protected void onProgressUpdate(Integer... values) {
    super.onProgressUpdate(values);
    Log.i(TAG, "progress = " + values[0]);
}
```

> 1. 根据进度可以更新UI
> 2. UI线程中
> 3. 入参是publishProgress的参数

14、AsyncTask的onPostExecute

```
@Override
protected void onPostExecute(String s) {
    super.onPostExecute(s);
    Log.i(TAG, s);
}
```

> 1. 任务完成后的收尾工作
> 2. UI线程中
> 3. 参数为doInBackground的返回值

串行or并行

15、AysncTask Android3.0之前的特点

> 1. 内部的ThreadPoolExecutor：核心线程数5个，最大线程数量128，非核心线程等待时间1s，采用阻塞队列 `LinkedBlockingQueue` 容量为10.
> 2. 1.6之前，AsyncTask是串行的；在1.6至2.3版本，AsyncTask是并行的
> 3. 缺点：AynsncTask最多能同时容纳138个任务(128+10)，超过后会抛出 `RejectedExecutionException` 异常

16、AysncTask Android7.0版本的特点

> 1. 串行处理：Android3.0及以上版本使用 `SerialExecutor` 作为默认的线程，会将任务串行的处理，保证一个时间段只有一个任务在执行。不会再出现之前的执行饱和策略的情况。
> 2. 在3.0及以上版本中，也可以使用并行处理 `asyncTask.executeOnExecutor(Asynctask.THREAD_POOL_EXECUTOR, "")`
> 3. THREAD_POOL_EXECUTOR就是采用以前的threadPoolExecutor，但核心线程数和最大线程数由CPU的核数计算得到，阻塞队列依旧是 `LinkedBlockingQueue` 且容量提升到 `128`
> 4. 当然也可以在 `.executeOnExecutor` 中传入其他几种线程池

17、AsyncTask如何指定采用其他线程池去执行任务?

> asyncTask.executeOnExecutor(Executor exec, null);

原理(18)

1、AsyncTask的机制原理概述

> 1. 内部是一个静态的线程池，AsyncTask的子类提交的异步任务，都会到这个线程池中执行。

2. 线程池中的工作线程会执行 `doInBackground` 执行后台异步任务。
3. 任务状态的改变，会发送消息到内部的 `InternalHandler` 中，并且根据消息去执行对应的回调函数。

内部线程池

2、AsyncTask的内部的静态线程池?

```
// CPU数量
private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
// 核心线程数(2~4)CPU数量1~3时，为2。如果CPU数量=4,就为3.如果CPU数量>4,为4。
private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1, 4));
// 最大线程数(CPU * 2 + 1)
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
// 线程存活时间30s
private static final int KEEP_ALIVE_SECONDS = 30;
// 线程工厂
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());
    }
};
// 任务队列，采用LinkedBlockingQueue。默认大小为128。
private static final BlockingQueue<Runnable> sPoolWorkQueue = new LinkedBlockingQueue<Runnable>(128);
/**========================================
 * AsyncTask线程池内部的线程池
 * 1. 在静态代码块中进行初始化
 * 2. 核心线程数2~4，最大线程数CPU数量*2+1，保活时间30s，采用128的有界阻塞队列
 * 3. 核心线程也开启超时。(保活时间30s)
 *========================================*/
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);
    // 开启核心线程的超时
    threadPoolExecutor.allowCoreThreadTimeOut(true);
    THREAD_POOL_EXECUTOR = threadPoolExecutor;
}
```

3、AsyncTask(API28)默认采用的串行线程池内部机制是怎么样的?

1. AsyncTask默认串行执行任务。
2. 内部具有SerialExecutor，但是底层和并发执行任务采用的是同一个线程池 `THREAD_POOL_EXECUTOR`

```
// 1、SerialExecutor，串行执行任务
public static final Executor SERIAL_EXECUTOR = new SerialExecutor();
// 2、默认采用串行Executor
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
// 3、SerialExecutor实现了Executor接口
private static class SerialExecutor implements Executor {
    // 4、内部采用了ArrayDeque。(无容量限制、非线程安全)
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;
    // 5、execute，将任务投递到ArrayDeque(双向队列)的尾部，调用scheduleNext()执行任务
    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (mActive == null) {
            scheduleNext();
        }
    }
    // 6、从双线队列的头部取出任务，并通过AsyncTask内部的静态线程池，去执行。
    protected synchronized void scheduleNext() {
        if ((mActive = mTasks.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);
        }
    }
}
```

4、AsyncTask内部的线程池的特点?

> 1. 总容量: CPU数量 * 2 + 1 + 128
> 2. 核心线程和非核心线程都具有保活时间30s
> 3. 所有AsyncTask以及子类的任务，都在该静态线程池中进行。

5、AsyncTask内部的线程池采用的阻塞队列LinkedBlockingQueue有什么特点?

> 1. 是链表组成的有界阻塞队列
> 2. 若构造时不指定队列缓存区大小，默认无穷大。一旦生产速度>消费速度，会导致内存耗尽。
> 3. AsyncTask中默认大小为128。

6、ArrayDeque的特点

> 1. 继承自AbstractCollection，实现Deque接口
> 2. 没有容量限制的双线队列
> 3. 非线程安全，需要使用synchronized等方法进行保护

7、AsyncTask如何设置采用并发方法去执行任务?

```
asyncTask.executeOnExecutor(Asynctask.THREAD_POOL_EXECUTOR, "");
```

8、AsyncTask内部只有一个静态线程池，如何做到AsyncTask1串行执行任务和AsyncTask2并发执行?

> 1. 内部的SerialExectuor
> 2. AsyncTask1采用ArrayDeque队列存储任务，并交给内部并发线程池去执行。
> 3. 任务执行完毕后，才会执行下一个任务。

构造方法

9、AsyncTask的构造方法和内部机制

> 1. 采用Callable方式创建线程，在call方法中执行具体的任务：执行doInBackground，并且回调onPostExecute
> 2. 采用主线程Looper或者自定义Looper创建内部InternelHandler。因为采用@hide进行注解，所以开发者无法使用这些API，所以内部Handler一定是主线程的。反射的方法在API28已经被禁止。

```java
// AsyncTask.java
private static abstract class WorkerRunnable<Params, Result> implements Callable<Result> {
    Params[] mParams;
}
// 实现Callable
private final WorkerRunnable<Params, Result> mWorker;
// FutureTask去包装Callable
private final FutureTask<Result> mFuture;

// AsyncTask.java
public AsyncTask() {
    this((Looper) null);
}
// AsyncTask.java
@hide
public AsyncTask(@Nullable Handler handler) {
    this(handler != null ? handler.getLooper() : null);
}
// AsyncTask.java
@hide
public AsyncTask(@Nullable Looper callbackLooper) {
    // 1、采用主线程Looper或者用给定的Looper创建InternalHandler
    mHandler = callbackLooper == null || callbackLooper == Looper.getMainLooper()
            ? getMainHandler()
            : new Handler(callbackLooper);
    // 2、实现WorkerRunnable，实现了Callable的call方法，也就是任务执行的地方。
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {
            mTaskInvoked.set(true);
            Result result = null;
            try {
                // 3、设置线程优先级为: THREAD_PRIORITY_BACKGROUND，后台优先级，相比于一般优先级会更慢，但是不会影响到UI。
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                // 4、调用doInBackground，传入参数，并且获取岛返回至
                result = doInBackground(mParams);
            } finally {
                // 5、去回调onPostExecute方法。
                postResult(result);
            }
            return result;
        }
    };
    mFuture = new FutureTask<Result>(mWorker) {
        @Override
        protected void done() {
            // 保证方法postResult(result)会被调用，如果没有调用Callable的call方法。会去回调onCancelled()。
            postResultIfNotInvoked(get());
        }
    };
}

// AsyncTask.java
private Result postResult(Result result) {
    // 1、将任务执行的结果result，投递到内部的消息队列中
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
            new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}

// AsyncTask.java-在Looper所在线程中去回调onCancelled()、onPostExecute()或者onProgressUpdate()
private static class InternalHandler extends Handler {
    public InternalHandler(Looper looper) {
        super(looper);
    }
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // 1、执行AsyncTask的finish方法
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                // 2、回调onPregressUpdate
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
```

```
        }
    }
    // AsyncTask.java-在Looper所在线程中回调onCancelled和onPostExecute
    private void finish(Result result) {
        // 1、调用cancel取消了AsyncTask后，会去回调onCancelled()方法
        if (isCancelled()) {
            onCancelled(result);
        } else {
            // 2、一般情况是在doInBackground后执行onPostExecute方法
            onPostExecute(result);
        }
        mStatus = AsyncTask.Status.FINISHED;
    }
```

10、onProgressUpdate、onPostExecute、onCancelled都执行在主线程(API28)?

> 错误！不准确！常规方法的确是在主线程，但是使用反射去调用@hide的构造方法，可以去使用子线程的Looper去构造内部Handler。
>
> 1. 都执行在创建内部Handler所使用的Looper所在的线程中。
> 2. 如果采用的是主线程的Looper，就是运行在主线程。
> 3. 如果采用的是其他线程的Looper，就在相应线程中。

11、为什么AsyncTask要使用实现Callable接口的方式去创建线程?

> 1. 创建线程一共三种方式：继承Thread、实现Runnable接口、实现Callable
> 2. 只有才用Callable接口，才能获取到任务执行的返回值。
> 3. 调用 `asyncTask.get()` 方法能获取到call()的返回值，也就是doInbackground的返回值。

12、onPostExecute()获取的返回值是哪里来的?

> 1. 实现的Callable的call()方法中，执行doInBackground获取到结果。
> 2. 调用 `postResult(result)` 去回调 `onPostExecute()`

```
public Result call() throws Exception {
   // 1、获取doinBakcground的返回值
    Result result = doInBackground(mParams);
   // 2、调用postResult方法
    postResult(result);
    return result;
}
```

execute

13、execute的内部源码和流程

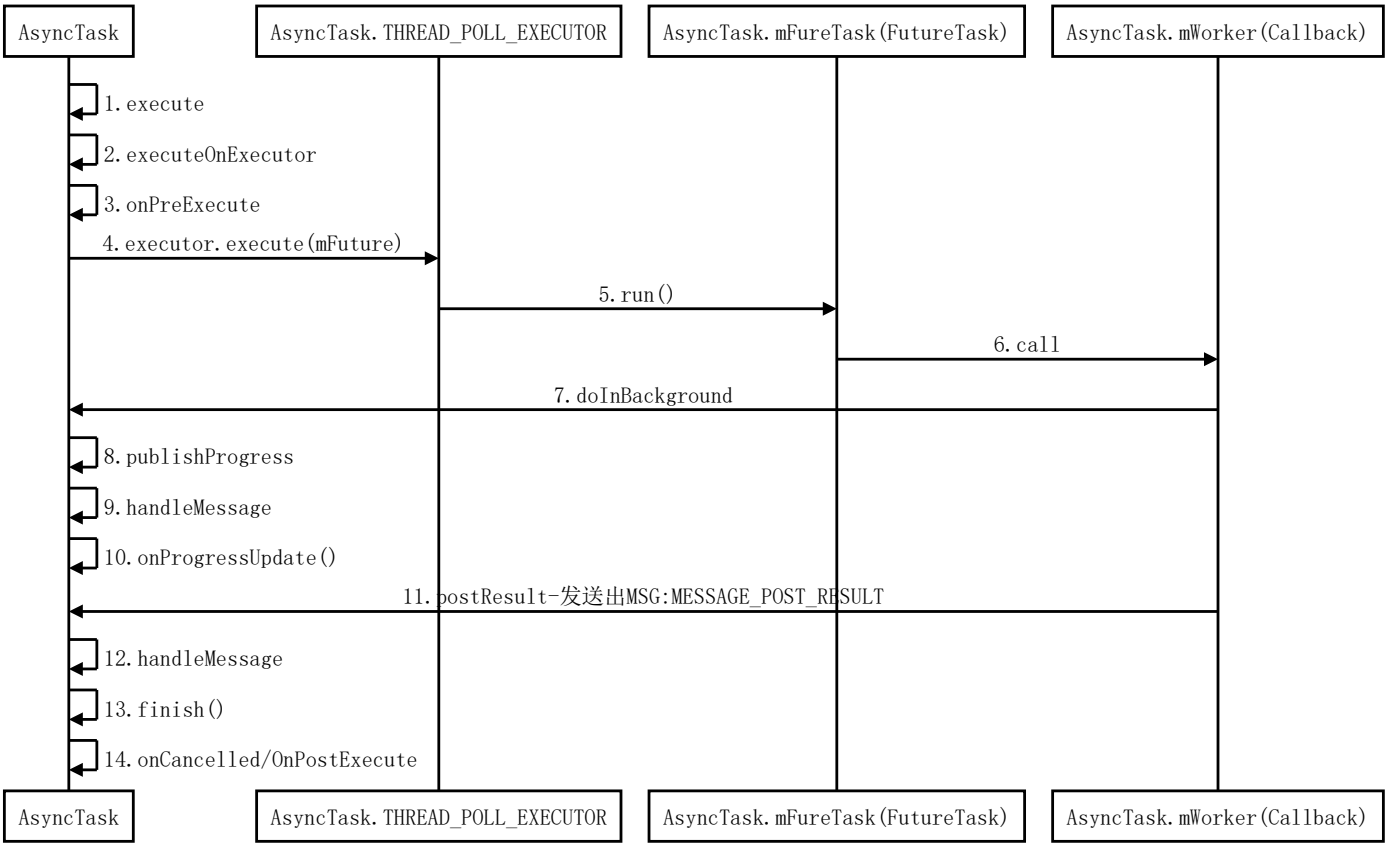
```
// AsyncTask.java-调用默认串行方式执行任务。
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
// AsyncTask.java
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec, Params... params) {
    mStatus = AsyncTask.Status.RUNNING;
    // 1、回调onPreExecute方法，也就是execute方法在哪个线程中执行，onPreExecute()方法就在哪个线程。
    onPreExecute();
    mWorker.mParams = params;
    // 2、Executor.execute()去执行任务
    exec.execute(mFuture);
    return this;
}

// AsyncTask.java-call方法就是任务执行的地方。
private final WorkerRunnable<Params, Result> mWorker = new WorkerRunnable<Params, Result>() {
    public Result call() throws Exception {
        Result result = null;
        try {
            // 1、设置线程优先级为: THREAD_PRIORITY_BACKGROUND，后台优先级，相比于一般优先级会更慢，但是不会影响到UI。
            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
            // 2、调用doInBackground，传入参数，并且获取岛返回至
            result = doInBackground(mParams);
        } finally {
            // 3、去回调onPostExecute方法。
            postResult(result);
        }
        return result;
    }
};
// AsyncTask.java-进行进度更新的操作，会到InternalHandler中执行
protected final void publishProgress(Progress... values) {
    if (!isCancelled()) {
        // 1、消息类型：MESSAGE_POST_PROGRESS
        getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
                new AsyncTaskResult<Progress>(this, values)).sendToTarget();
    }
}

// AsyncTask.java-回调onProgressUpdate()更新进度，在结束时回调onPostExecute()
private static class InternalHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // 1、执行AsyncTask的finish方法
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                // 2、回调onPregressUpdate
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}
```

> 1. execute()中执行executeOnExecutor()
> 2. 执行onPreExecute()
> 3. 通过内部静态线程池去执行任务。
> 4. 在Call中执行doInBackground()，如果调用了publishProgress()会到内部Handler中去执行onProgressUpdate()方法
> 5. 后台任务执行完毕，会去回调onPostExecute()方法

14、onPreExecute()一定执行在主线程?

> 错误!
>
> 1. execute()方法在哪个线程，onPreExecute()就执行在哪个线程

15、AysncTask的流程



cancel

16、AsyncTask的cancel()原理

> 1. 底层调用mFuture的cancel()去取消任务
> 2. publicProgress不会再去更新进度
> 3. 结束时会直接回调onCancelled()而不是onPostExecute()

```
// AsyncTask.java
public final boolean cancel(boolean mayInterruptIfRunning) {
  // 1、publicProgress不会再去更新进度
    mCancelled.set(true);
  // 2、调用mFuture的cancel()去取消任务
    return mFuture.cancel(mayInterruptIfRunning);
}
// AsyncTask.java-用于取消任务
public final boolean isCancelled() {
        return mCancelled.get();
}
// AsyncTask.java-不会再去更新进度
protected final void publishProgress(Progress... values) {
        if (!isCancelled()) {
            getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
                    new AsyncTaskResult<Progress>(this, values)).sendToTarget();
        }
}
// AsyncTask.java-不会去执行onPostExecute()会直接回调onCancelled()
private void finish(Result result) {
        if (isCancelled()) {
            onCancelled(result);
        } else {
            onPostExecute(result);
        }
        mStatus = Status.FINISHED;
}
```

17、 onCancelled()方法什么时候会被回调?

> 1. 调用 `cancel()` 之后。
> 2. 不会去执行onPostExecute()，而是直接执行onCancelled()

18、cancel(boolean mayInterruptIfRunning)的参数有什么用?

> 1. 如果为true，无论正在执行什么，会立即终止任务。
> 2. 如果为false，会让线程中执行的任务结束后，终止该任务(是等待时间片结束就停止，而不是整个Task结束了，才停止。)
> 3. 比如有个Thread.sleep()
>   1. true: 会直接中断sleep然后停止任务。
>   2. false：会等待sleep结束，然后停止任务。

知识补充(1)

1、AsyncTask并没有串行线程池

> 1. 无论是串行方式还是并发方式都是在同一个静态线程池中执行。
> 2. 区别在于：串行方式采用自定义SerialExecutor，内部使用ArrayDeque队列.
> 3. 将任务投递到队列中，交给线程池执行，执行完毕后，再取出下一个任务并且执行。