android多线程-AsyncTask之工作原理深入解析(下)

原创 2016年09月07日 22:45:05 标签：android / 多线程 / AsyncTask / 工作原理 2527

关联文章:
Android 多线程之HandlerThread 完全详解
Android 多线程之IntentService 完全详解
android多线程-AsyncTask之工作原理深入解析(上)
android多线程-AsyncTask之工作原理深入解析(下)

上篇分析AsyncTask的一些基本用法以及不同android版本下的区别，接着本篇我们就来全面剖析一下AsyncTask的工作原理。在开始之前我们先来了解一个多线程的知识点——Callable<V> 、Future<V>和FutureTask类

一、理解Callable<V> 、Future<V>以及FutureTask类

Callable<V>

Callable的接口定义如下:

```
1  public interface Callable<V> {
2        V   call()   throws Exception;
3  }
```

Callable接口声明了一个名称为call()的方法，该方法可以有返回值V，也可以抛出异常。Callable也是一个线程接口，它与Runnable的主要区别就是Callable在线程执行完成后可以有返回值而Runnable没有返回值，Runnable接口声明如下：

```
1  public interface Runnable {
2      public abstract void run();
3  }
```

那么Callable接口如何使用呢，Callable需要和ExcutorService结合使用，其中ExecutorService也是一个线程池对象继承自Executor接口，对于线程池的知识点不了解可以看看我的另一篇文章,这里就不深入了，接着看看ExecutorService提供了那些方法供我们使用：

```
1  <T> Future<T> submit(Callable<T> task);
2  <T> Future<T> submit(Runnable task, T result);
3  Future<?> submit(Runnable task);
```

- submit(Callable task)，传递一个实现Callable接口的任务，并且返回封装了异步计算结果的Future。
- submit(Runnable task, T result)，传递一个实现Runnable接口的任务，并且指定了在调用Future的get方法时返回的result对象。
- submit(Runnable task)，传递一个实现Runnable接口的任务，并且返回封装了异步计算结果的Future。

因此我们只要创建好我们的线程对象（实现Callable接口或者Runnable接口），然后通过上面3个方法提交给线程池去执行即可。Callable接口介绍就先到这，再来看看Future时什么鬼。

Future<V>

Future接口是用来获取异步计算结果的，说白了就是对具体的Runnable或者Callable对象任务执行的结果进行获取(get()),取消(cancel()),判断是否完成等操作。其方法如下：

```
public interface Future<V> {
    //取消任务
    boolean cancel(boolean mayInterruptIfRunning);

    //如果任务完成前被取消，则返回true。
    boolean isCancelled();

    //如果任务执行结束，无论是正常结束或是中途取消还是发生异常，都返回true。
    boolean isDone();

    //获取异步执行的结果，如果没有结果可用，此方法会阻塞直到异步计算完成。
    V get() throws InterruptedException, ExecutionException;

    // 获取异步执行结果，如果没有结果可用，此方法会阻塞，但是会有时间限制，
    //如果阻塞时间超过设定的timeout时间，该方法将返回null。
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
 TimeoutException;

}
```

总得来说Future有以下3点作用：

- 能够中断执行中的任务
- 判断任务是否执行完成
- 获取任务执行完成后额结果。

但是Future只是接口，我们根本无法将其创建为对象，于官方又给我们提供了其实现类FutureTask，这里我们要知道前面两个接口的介绍都只为此类做铺垫，毕竟AsncyTask中使用到的对象是FutureTask。

FutureTask

先来看看FutureTask的实现：

```
public class FutureTask<V> implements RunnableFuture<V> {
```

显然FutureTask类实现了RunnableFuture接口，我们再看一下RunnableFuture接口的实现：

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

从接口实现可以看出，FutureTask除了实现了Future接口外还实现了Runnable接口，因此FutureTask既可以当做Future对象也可是Runnable对象，当然FutureTask也就可以直接提交给线程池来执行。接着我们最关心的是如何创建FutureTask对象，实际上可以通过如下两个构造方法来构建FutureTask

```
public FutureTask(Callable<V> callable) {
}
public FutureTask(Runnable runnable, V result) {
}
```

从构造方法看出，我们可以把一个实现了Callable或者Runnable的接口的对象封装成一个FutureTask对象，然后通过线程池去执行，那么具体如何使用呢？简单案例，CallableDemo.java代码如下：

```

package com.zejian.Executor;
import java.util.concurrent.Callable;
/**
 * Callable接口实例 计算累加值大小并返回
 */
public class CallableDemo implements Callable<Integer> {

    private int sum;
    @Override
    public Integer call() throws Exception {
```

```
        for(int i=0 ;i<5000;i++){
            sum=sum+i;
        }
        System.out.println("Callable子线程计算结束！");
        return sum;
    }
}
```

3 CallableTest.java测试代码如下：

```
package com.zejian.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;
public class CallableTest {

public static void main(String[] args) {
//第一种使用方式
//      //创建线程池
//      ExecutorService es = Executors.newSingleThreadExecutor();
//      //创建Callable对象任务
//      CallableDemo calTask=new CallableDemo();
//      //提交任务并获取执行结果
//      Future<Integer> future =es.submit(calTask);
//      //关闭线程池
//      es.shutdown();

    //第二中使用方式

    //创建线程池
    ExecutorService es = Executors.newSingleThreadExecutor();
    //创建Callable对象任务
    CallableDemo calTask=new CallableDemo();
    //创建FutureTask
    FutureTask<Integer> futureTask=new FutureTask<>(calTask);
    //执行任务
    es.submit(futureTask);
    //关闭线程池
    es.shutdown();
    try {
        Thread.sleep(2000);
    System.out.println("主线程在执行其他任务");

    if(futureTask.get()!=null){
        //输出获取到的结果
        System.out.println("futureTask.get()-->"+futureTask.get());
    }else{
        //输出获取到的结果
        System.out.println("futureTask.get()未获取到结果");
    }

    } catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println("主线程在执行完成");
}
}
```

代码非常简单，注释也很明朗，这里我们分析一下第2种执行方式，先前声明一个CallableDemo类，该类实现了Callable接口，接着通过call方法去计算sum总值并返回。然后在测试类CallableTest中，把CallableDemo实例类封装成FutureTask对象并交给线程池去执行，最终执行结果将封装在FutureTask中，通过FutureTask#get()可以获取执行结果。第一种方式则是直接把Callable实现类丢给线程池执行，其结果封装在Future实例中，第2种方式执行结果如下：

```
Callable子线程开始计算啦!
主线程在执行其他任务
```

```
futureTask.get()-->12497500
主线程在执行完成
```

ok~，到此我们对Callable、Future和FutureTask就介绍到这，有了这个知识铺垫，我们就可以愉快的撩开AsyncTask的内部工作原理了。

二、AsyncTask的工作原理完全解析

在上篇中，使用了如下代码来执行AsyncTask的异步任务：

```
new AysnTaskDiff("AysnTaskDiff-1").execute("");
```

从代码可知，入口是execute方法，那我们就先看看execute的源码：

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

很明显execute方法只是一个壳子，直接调用了`executeOnExecutor(sDefaultExecutor, params)`，其中sDefaultExecutor是一个串行的线程池，接着看看sDefaultExecutor内部实现：

```
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();

//串行线程池类，实现Executor接口
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() { //插入一个Runnble任务
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        //判断是否有Runnable在执行,没有就调用scheduleNext方法
        if (mActive == null) {
            scheduleNext();
        }
    }

    protected synchronized void scheduleNext() {
      //从任务队列mTasks中取出任务并放到THREAD_POOL_EXECUTOR线程池中执行.
      //由此也可见任务是串行进行的。
        if ((mActive = mTasks.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);
        }
    }
}
```

从源码可以看出，ArrayDeque是一个存放任务队列的容器（mTasks），任务Runnable传递进来后交给SerialExecutor的execute方法处理，SerialExecutor会把任务Runnable插入到任务队列mTasks尾部，接着会判断是否有Runnable在执行,没有就调用scheduleNext方法去执行下一个任务，接着交给THREAD_POOL_EXECUTOR线程池中执行，由此可见SerialExecutor并不是真正的线程执行者，它只是是保证传递进来的任务Runnable（实例是一个FutureTask）串行执行，而真正执行任务的是THREAD_POOL_EXECUTOR线程池，当然该逻辑也体现AsyncTask内部的任务是默认串行进行的。顺便看一下THREAD_POOL_EX

```
//CUP核数
private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
//核心线程数量
private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
//最大线程数量
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
//非核心线程的存活时间1s
private static final int KEEP_ALIVE = 1;
//线程工厂类
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());
    }
};
//线程队列，核心线程不够用时，任务会添加到该队列中，队列满后，会去调用非核心线程执行任务
private static final BlockingQueue<Runnable> sPoolWorkQueue =
        new LinkedBlockingQueue<Runnable>(128);

/**
 * An {@link Executor} that can be used to execute tasks in parallel.
 * 创建线程池
 */
public static final Executor THREAD_POOL_EXECUTOR
        = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE,
                TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
```

ok~，关于sDefaultExecutor，我们先了解到这，回到之前execute方法内部调用的executeOnExecutor方法的步骤，先来看看executeOnExecutor都做了些什么事？其源码如下：

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
            Params... params) {
   //判断在那种状态
   if (mStatus != Status.PENDING) {
       switch (mStatus) {
           case RUNNING:
               throw new IllegalStateException("Cannot execute task:"
                       + " the task is already running.");
           case FINISHED://只能执行一次!
               throw new IllegalStateException("Cannot execute task:"
                       + " the task has already been executed "
                       + "(a task can be executed only once)");
       }
   }

   mStatus = Status.RUNNING;
   //onPreExecute()在此执行了！！！
   onPreExecute();
   //参数传递给了mWorker.mParams
   mWorker.mParams = params;
   //执行mFuture任务，其中exec就是传递进来的sDefaultExecutor
   //把mFuture交给线程池去执行任务
   exec.execute(mFuture);

   return this;
    }
```

从executeOnExecutor方法的源码分析得知，执行任务前先会去判断当前AsyncTask的状态，如果处于RUNNING和FINISHED状态就不可再执行，直接抛出异常，只有处于Status.PENDING时，AsyncTask才会去执行。然后onPreExecute()被执行的，该方法可以用于线程开始前做一些准备工作。接着会把我们传递进来的参数赋值给 mWorker.mParams ，并执行开始执行mFuture任务，那么mWorker和mFuture到底是什么？先看看mWorker即WorkerRunnable的声明源码：

```
//抽象类
private static abstract class WorkerRunnable<Params, Result> implements Callable<Result>
```

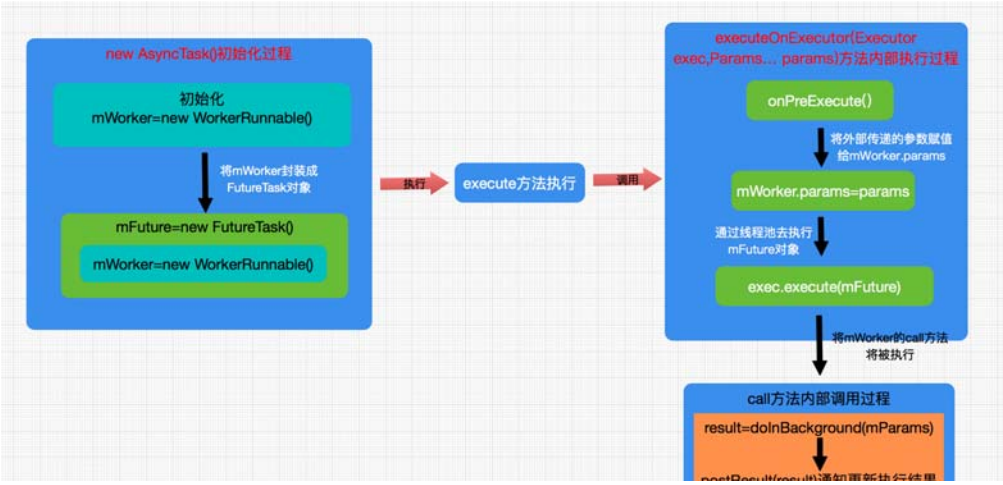
```
        Params[] mParams;
    }
```

WorkerRunnable抽象类实现了Callable接口，因此WorkerRunnable本质上也算一个Callable对象，其内部还封装了一个mParams的数组参数，因此我们在外部执行execute方法时传递的可变参数最终会赋值给WorkerRunnable的内部数组mParams，这些参数最后会传递给doInBackground方法处理，这时我们发现doInBackground方法也是在WorkerRunnable的call方法中被调用的，看看其源码如下：


```
public AsyncTask() {
   //创建WorkerRunnable mWorker，本质上就是一个实现了Callable接口对象
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {
            //设置标志
            mTaskInvoked.set(true);

         Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
            //执行doInBackground，并传递mParams参数
            Result result = doInBackground(mParams);
            Binder.flushPendingCommands();
            //执行完成调用postResult方法更新结果
            return postResult(result);
        }
    };
//把mWorker（即Callable实现类）封装成FutureTask实例
//最终执行结果也就封装在FutureTask中
    mFuture = new FutureTask<Result>(mWorker) {
        //任务执行完成后被调用
        @Override
        protected void done() {
            try {
             //如果还没更新结果通知就执行postResultIfNotInvoked
                postResultIfNotInvoked(get());
            } catch (InterruptedException e) {
                android.util.Log.w(LOG_TAG, e);
            } catch (ExecutionException e) {
                throw new RuntimeException("An error occurred while executing doInBackgr
ound()",
                        e.getCause());
            } catch (CancellationException e) {
                //抛异常
                postResultIfNotInvoked(null);
            }
        }
    };
}
```

可以看到在初始化AsyncTask时，不仅创建了mWorker（本质实现了Callable接口的实例类）而且也创建了FutureTask对象，并把mWorker对象封装在FutureTask对象中，最后FutureTask对象将在executeOnExecutor方法中通过线程池去执行。给出下图协助理解：



AsynTask在初始化时会创建mWorker实例对象和FutureTask实例对象，mWorker是一个实现了Callable线程接口并封装了传递参数的实例对象，然后mWorker实例会被封装成FutureTask实例中。在AsynTask创建后，我们调用execute方法去执行异步线程，其内部又直接调用了executeOnExecutor方法，并传递了线程池exec对象和执行参数，该方法内部通过线程池exec对象去执行mFuture实例，这时mWorker内部的call方法将被执行并调用doInBackground方法，最终通过postResult去通知更新结果。关于postResult方法,其源码如下：


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

显然是通过Handler去执行结果更新的，在执行结果成返回后，会把result封装到一个AsyncTaskResult对象中，最后把MESSAGE_POST_RESULT标示和AsyncTaskResult存放到Message中并发送给Handler去处理，这里我们先看看AsyncTaskResult的源码：

```
private static class AsyncTaskResult<Data> {
        final AsyncTask mTask;
        final Data[] mData;

        AsyncTaskResult(AsyncTask task, Data... data) {
            mTask = task;
            mData = data;
        }
    }
```

显然AsyncTaskResult封装了执行结果的数组以及AsyncTask本身，这个没什么好说的，接着看看AsyncTaskResult被发送到handler后如何处理的。

```
private static class InternalHandler extends Handler {
    public InternalHandler() {
        //获取主线程的Looper传递给当前Handler，这也是为什么AsyncTask只能在主线程创建并执行的
原因
        super(Looper.getMainLooper());
    }

    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
    //获取AsyncTaskResult
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            //执行完成
            case MESSAGE_POST_RESULT:
                // There is only one result
                result.mTask.finish(result.mData[0]);
                break;
                //更新进度条的标志
            case MESSAGE_POST_PROGRESS:
            //执行onProgressUpdate方法，自己实现。
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}
```

从Handler的源码分析可知，该handler绑定的线程为主线线程，这也就是为什么AsyncTask必须在主线程创建并执行的原因了。接着通过handler发送过来的不同标志去决定执行那种结果，如果标示为MESSAGE_POST_RESULT则执行AsyncTask的finish方法并传递执行结果给该方法，finish方法源码如下：

```
private void finish(Result result) {
    if (isCancelled()) {//判断任务是否被取消
```

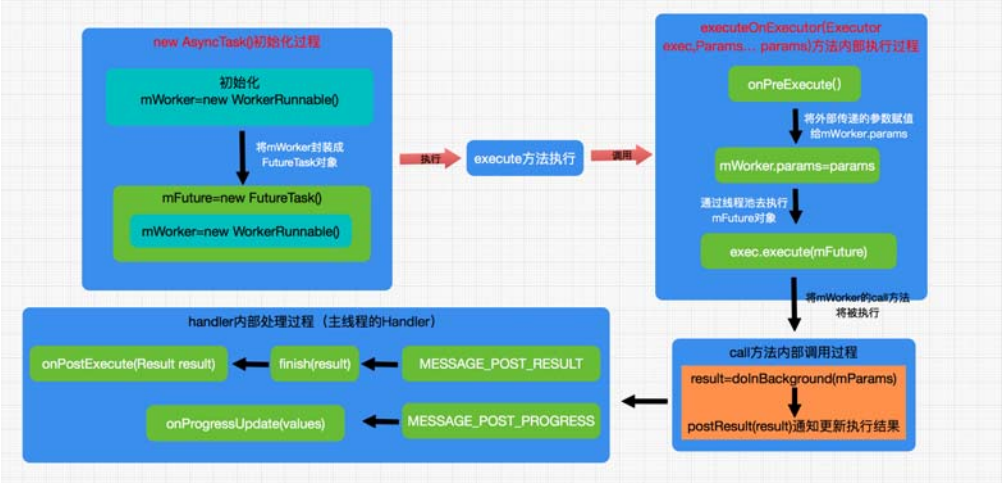
```
            onPostExecute(result);
        }
        //更改AsyncTask的状态为已完成
        mStatus = Status.FINISHED;
    }
```

该方法先判断任务是否被取消，如果没有被取消则去执行onPostExecute(result)方法，外部通过onPostExecute方法去更新相关信息，如UI，消息通知等。最后更改AsyncTask的状态为已完成。到此AsyncTask的全部流程执行完。

这里还有另一种标志MESSAGE_POST_PROGRESS，该标志是我们在doInBackground方法中调用publishProgress方法时发出的，该方法原型如下：

```
protected final void publishProgress(Progress... values) {
    if (!isCancelled()) {
    //发送MESSAGE_POST_PROGRESS，通知更新进度条
        getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
                new AsyncTaskResult<Progress>(this, values)).sendToTarget();
    }
}
```

ok~，AsyncTask的整体流程基本分析完，最后来个总结吧：当我们调用execute(Params… params)方法后，其内部直接调用executeOnExecutor方法，接着onPreExecute()被调用方法，执行异步任务的WorkerRunnable对象(实质为Callable对象)最终被封装成FutureTask实例，FutureTask实例将由线程池sExecutor执行去执行，这个过程中doInBackground(Params… params)将被调用（在WorkerRunnable对象的call方法中被调用），如果我们覆写的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；最后如果FutureTask任务执行成功并返回结果，则通过postResult方法发送一条MESSAGE_POST_RESULT的消息去执行AsyncTask的finish方法，在finish方法内部onPostExecute(Result result)方法被调用，在onPostExecute方法中我们可以更新UI或者释放资源等。这既是AsyncTask内部的工作流程，可以说是Callable+FutureTask+Executor+Handler内部封装。结尾我们献上一张执行流程，协助大家理解整个流程：



好~，本篇到此结束。。。

Android 多线程之HandlerThread 完全详解
Android 多线程之IntentService 完全详解
android多线程-AsyncTask之工作原理深入解析(上)
android多线程-AsyncTask之工作原理深入解析(下)

主要参考资料：
https://developers.android.com
《android开发艺术探索》