发件人：方堃 fangkun119@icloud.com
主题：CH14 HalfSync/HalfAsync
日期：2017年5月2日 下午12:03
收件人：

CH14 HalfSync/HalfAsync

2017年4月30日 星期日
下午4:51

用途

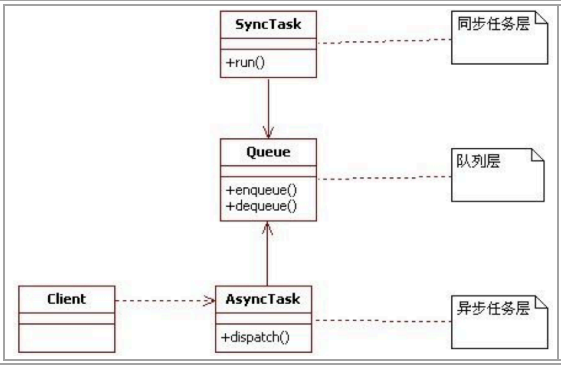
Half-sync/Half-async模式集成了同步编程和异步编程的优势，它通过同步任务和异步任务的共同协作来完成一个计算，既保持了同步编程的简单性，又充分发挥异步编程在提高系统并发性方面的优势。该模式就如一个称职的管理者，知晓不同下属各自的优势和弱点，在指派任务的时候能够根据下属的优势和弱点扬长避短，并使各个下属相互协作，共同完成工作。

设计思

Half-sync/Half-async模式是一个分层架构。它包含3个层次：异步任务层、同步任务层和队列层。Half-sync/Half-async模式的核心思想是如何将系统中的任务进行恰当的分解，使各个子任务落入合适的层次中。

低级的任务（如与用户界面有关的任务）或者耗时较短的任务可以安排在异步任务层。而高级的任务（如数据库访问）或者耗时较长的任务可以安排在同步任务层。而异步任务层和同步任务层这两层之间的协作通过队列层进行解耦（Decoupling）：队列层负责异步任务层和同步任务层之间的数据交换。

类图



AsyncTask：异步任务，负责接收来自客户端的输入，对其进行初步处理，并通过队列与相应的同步任务通信。其主要方法及职责如下。

- **dispatch：**对输入进行初步处理，并构造相应消息放入队列由相应的同步任务进行处理。

Queue：队列，异步任务层和同步任务层进行通信的中介。其主要方法及职责如下。

- **enqueue：**消息入队列。
- **dequeue：**消息出队列。

SyncTask：同步任务，负责处理队列中的消息所对应的计算。其主要方法及职责如下。

- **run：**执行同步任务。

时序

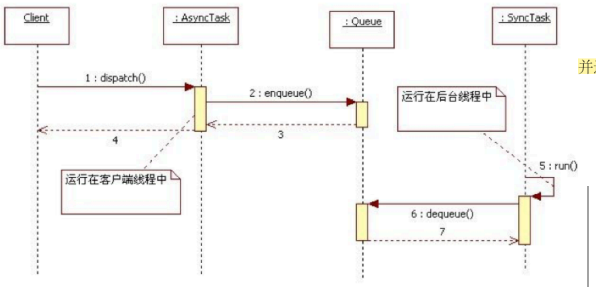


图14-2. Half-sync/Half-async模式的序列图

第1步：客户端代码调用AsyncTask的dispatch方法。

第2、3步：dispatch方法对其输入进行初步处理后，构造相应消息放入队列，并返回。

第4步：dispatch方法返回。

第5步：同步任务的run方法被JVM或者线程池调用。

第6、7步：同步任务的run方法取队列中的消息，并根据消息执行相应操作。

使用权衡

优势

Half-sync/Half-async模式既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性。Half-sync/Half-async模式通过把低级任务或者耗时较短的任务安排在异步层，减少了客户端的等待，有利于提升系统的吞吐率。而高级任务或耗时较长的任务被安排在同步层，这使得我们可以在不影响客户端代码处理性能的情况下保持了同步编程的简单性。

独立的并发访问控制策略。Half-sync/Half-async模式的分层架构使得各个层次的代码可以有各自的并发访问控制策略。这似乎很显而易见，但是它却蕴含了Half-sync/Half-async模式的几个重要的应用场景。例如，在Java Swing中，为了避免GUI组件（如按钮）死锁（Dead lock），Swing的GUI层的代码特意采用单线程模型（Swing的GUI层代码运行在Event Loop线程中）。这样就可以避免使用锁，也就自然地避免了死锁。但是，这同时也带来另外的问题：如果要在Swing的Event Loop线程中执行耗时较长的任务（如从服务器上下载大文件），那么GUI就会被"冻住"而无法响应用户操作。为了解决这个问题，JDK 1.6引入了SwingWorker类。该类应用了Half-sync/Half-async模式。SwingWorker就像一个转换器，其一端（异步层）连接着Swing的单线程GUI层，另一端（同步层）连接着多线程的应用代码层（如从服务器上下载文件的代码）。这样，耗时较长的应用代码仍然可以多线程执行，比如多个线程分段从服务器上下载大文件，而不影响Swing GUI层的单线程模式。

另外一个典型的例子是，Half-sync/Half-async模式的异步层对接的是多线程环境，因此该层通常需要并发访问控制措施，如显式锁。而同步层是一个单线程环境，因此它无须任何并发访问控制即可保证线程安全。例如，同步层的代码需要调用一些非线程安全的API。而我们又不想因此引入显式锁，同时又要保证线程安全。此时，这种异步层对接多线程环境，同步层采用单线程模型的Half-sync/Half-async模式就可以派上用场。这种场景可以使我们用单线程编程这种方式去编写同步层的代码，从而使同步层的代码进一步简单化。

问题

Half-sync/Half-async模式可以看成是Producer-Consumer模式的一个实例。AsyncTask和SyncTask分别相当于Producer-Consumer模式的Producer和Consumer参与者。Producer-Consumer模式在实际应用中常见的一个问题是Consumer处理"产品"的速率往往比Producer"生产"产品"的速率慢。如果同步层的处理速率过慢的情况持续存在，那么队列就会逐渐积压。如果Producer和Consumer采用有界队列进行通信，那么当队列积压到队列满的时候就会导致Producer的处理速率受到影响。这是因为Producer在"生产"好"产品"将其放入队列的时候，需要等待队列非满（由满变成不满的状态）。如果Producer和Consumer采用无界队列进行通信，那么队列持续积压到一定程度可能就导致内存溢出，这是因为队列中存储的大量元素占用了大量内存无法被垃圾回收（Garbage collect）掉。

因此，在使用Half-sync/Half-async模式的时候，需要根据实际情况在有界队列和无界队列之间做出选择。选择有界队列有个隐含的好处：当队列积压到满的情况下，Producer会因其需要等待队列非满而降低处理速率，从而在一定程度上减轻了Consumer的处理负担[1]。有界队列可以使用java.util.concurrent.ArrayBlockingQueue。ArrayBlockingQueue用于存储队列元素的存储空间是预先分配的，这意味着它在使用过程中内存开销较小（无须动态申请存储空间）。无界队列可以使用java.util.concurrent.LinkedBlockingQueue。LinkedBlockingQueue用于存储队列元素的存储空间是在其使用过程中动态分配的，因此它可能会增加JVM垃圾回收的负担。

防止层过

同步层中的高级任务往往涉及I/O这种比较慢的操作，如网络读/写、数据库操作以及文件读/写等。这些任务执行的快慢往往取决于其依赖的外部资源（如数据库服务器）。而这些外部资源又往往不在我们的控制范围之内。因此要避免同步层处理过慢，还要从同步层自身的设计入手。例如，某同步任务需要通过广域网发送请求给服务器。网速、对端服务器性能等因素在极端的情况下可能导致执行该同步任务的线程由于等待对端响应而全部被阻塞，从而使得队列中的新增元素没有空闲线程去处理。这种情形下，该同步任务可能需要改用非阻塞I/O（如基于Java NIO的API），并采用专门的线程去处理I/O等待事件。

可复用

```
public abstract class AsyncTask<V> {

    // 相当于Half-sync/Half-async模式的同步层：用于执行异步层提交的任务。
    private volatile Executor executor;
    private final static ExecutorService DEFAULT_EXECUTOR;

    static {
     DEFAULT_EXECUTOR = new ThreadPoolExecutor(1, 1, 8, TimeUnit.HOURS,
         new LinkedBlockingQueue<Runnable>(),  new ThreadFactory()  {

           @Override
           public Thread newThread(Runnable r) {
             Thread thread = new Thread(r, "AsyncTaskDefaultWorker");
```

```
        Thread thread = new Thread(r, "AsyncTask#" +
                mCount.getAndIncrement());
        // 使该线程在JVM关闭时自动停止
        thread.setDaemon(true);
        return thread;
       }

      },  new RejectedExecutionHandler()  {

            /**
       * 该RejectedExecutionHandler支持重试。
       * 当任务被ThreadPoolExecutor拒绝时，
       * 该RejectedExecutionHandler支持
       * 重新将任务放入ThreadPoolExecutor
       * 的工作队列（这意味着，此时客户端代码
       * 需要等待ThreadPoolExecutor的队列非满）。
       */
      @Override
      public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
       if (! executor.isShutdown())  {
        try {
         executor.getQueue(). put(r);
        } catch (InterruptedException e) {
         ;
        }
       }

      }

     });
}

public AsyncTask(Executor executor) {
 this.executor = executor;
}

public AsyncTask()  {
 this(DEFAULT_EXECUTOR);
}

public void setExecutor(Executor executor) {
 this.executor = executor;
}

/**
 * 留给子类实现耗时较短的任务，默认实现什么也不做。
 *
 * @param params
 *          客户端代码调用dispatch方法时所传递的参数列表
 */
protected void onPreExecute(Object...  params) {
 // 什么也不做
}
/**
 * 留给子类实现。用于实现同步任务执行结束后所需执行的操作。默认实现什么也不做。
 *
 * @param result
 *          同步任务的处理结果
 */
protected void onPostExecute(V result) {
 // 什么也不做
}

protected void onExecutionError(Exception e) {
 e.printStackTrace();
}

/**
 * 留给子类实现耗时较长的任务（同步任务），由后台线程负责调用。
 *
 * @param params
 *          客户端代码调用dispatch方法时所传递的参数列表
 * @return 同步任务的处理结果
 */
protected abstract V doInBackground(Object...  params);

/**
 * 对外（其子类）暴露的服务方法。该类的子类需要定义一个比该方法命名更为具体的服务方法（如
 * downloadLargeFile）。
 * 该命名具体的服务方法（如downloadLargeFile）可直接调用该方法。
 *
 * @param params
 *          客户端代码传递的参数列表
 * @return 可借以获取任务处理结果的Promise（参见第6章，Promise模式）实例。
 */
protected Future<V> dispatch(final Object...  params) {
 FutureTask<V> ft = null;

 // 进行异步层初步处理
 onPreExecute(params);

 Callable<V> callable = new Callable<V>()  {
  @Override
  public V call()  throws Exception {
   V result;
   result = doInBackground(params);
   return result;
  }

 };
```

```
    ft = new FutureTask<V>(callable) {

     @Override
     protected void done()  {
      try {
       onPostExecute(this.get());
      } catch (InterruptedException e) {
       onExecutionError(e);
      } catch (ExecutionException e) {
       onExecutionError(e);
      }
     }
    };

    // 提交任务到同步层处理
    executor.execute(ft);

    return ft;
   }

  }
```

使用上码

使用AsyncTask类来实现Half-sync/Half-async模式，应用代码只需要创建一个AsyncTask类的子类（或者匿名子类），并在子类中完成以下几个“填空”任务。

1. 【必需】定义一个含义具体的服务方法名，该方法可直接调用父类的dispatch方法。
2. 【必需】实现父类的抽象方法doInBackground方法。
3. 【可选】根据应用的实际需要覆盖父类的onPreExecute方法、onPostExecute方法和onExecutionException方法。

使用上码的例

```
public class SampleAsyncTask {

 public static void main(String[]  args) {
  XAsyncTask task = new XAsyncTask();
  List<Future<String>> results = new LinkedList<Future<String>>();

  try {
   results.add(task.doSomething("Half-sync/Half-async",  1));

   results.add(task.doSomething("Pattern",  2));

   for (Future<String> result : results) {
    Debug.info(result.get());
   }
   Thread.sleep(200);
  } catch (Exception e) {

   e.printStackTrace();
  }

 }

 private static class XAsyncTask extends AsyncTask<String> {

  @Override
  protected String doInBackground(Object...  params) {
   String message = (String) params[0];
   int sequence = (Integer) params[1];
   Debug.info("doInBackground:" + message);
   return "message " + sequence + ":" + message;
  }

  @Override
  protected void onPreExecute(Object...  params) {
   String message = (String) params[0];
   int sequence = (Integer) params[1];
   Debug.info("onPreExecute:[" + sequence + "]" + message);
  }

  public Future<String> doSomething(String message, int sequence) {
   if (sequence < 0) {
    throw new IllegalArgumentException("Invalid sequence:" + sequence);
   }
   return this.dispatch(message, sequence);
  }
 }
}
```

```
[2015-04-22 20:26:19.333][INFO][main]: onPreExecute:[1]Half-sync/Half-async
[2015-04-22 20:26:19.335][INFO][main]: onPreExecute:[2]Pattern
[2015-04-22 20:26:19.336][INFO][AsyncTaskDefaultWorker]: doInBackground:async
[2015-04-22 20:26:19.336][INFO][main]: message 1:Half-sync/Half-async
[2015-04-22 20:26:19.337][INFO][AsyncTaskDefaultWorker]: doInBackground:Pattern
[2015-04-22 20:26:19.337][INFO][main]: message 2:Pattern
```

JDK例

Java Swing为了避免其GUI组件（如按钮）出现死锁，而特意采用单线程模型。所有的GUI组件都是运行在唯一的一个线程，即Event Loop线程中的。在Event Loop线程中运行耗时较长的任务会导致用户界面被“冻住”而无法响应用户操作。为了解决这个问题，JDK 1.6引入了SwingWorker类。该类应用了Half-sync/Half-async模式。SwingWorker使得一些耗时较短的任务运行在Event Loop线程中，而耗时较长的任务运行在其维护的后台线程中。因此，SwingWorker相当于Half-sync/Half-async模式的AsyncTask参与者实例。其doInBackground方法相当于AsyncTask参与者的dispatch方法。而SwingWorker内部维护的后台线程则相当于SyncTask参与者实例，它们负责真正执行耗时较长的任务。

例子	电子书3638