发件人：**方堃** fangkun119@icloud.com
主题：CH06 Promise
日期：2017年5月2日 下午12:00
收件人：

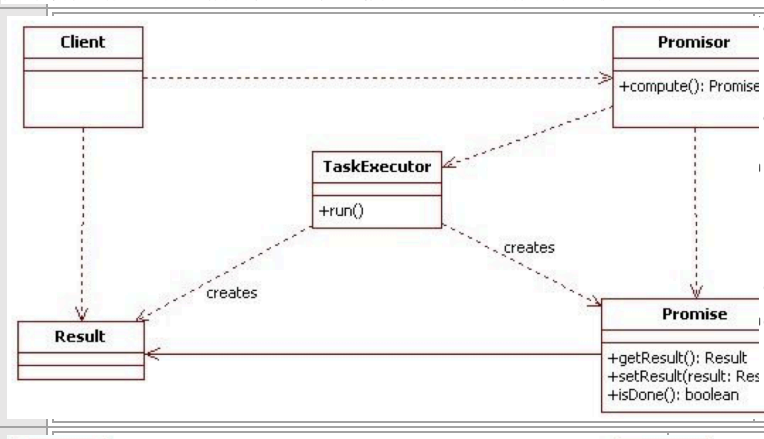
CH06 Promise

2017年4月24日 星期一
上午8:44

用途

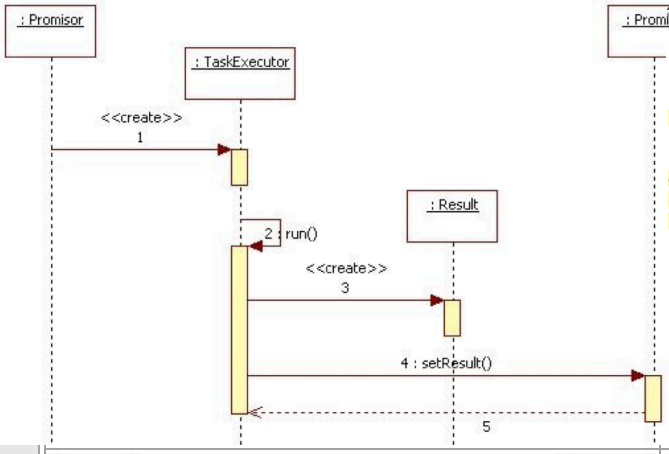
Promise模式是一种异步编程模式[1]。它使得我们可以先开始一个任务的执行，并得到一个用于获取该任务执行结果的凭据对象，而不必等待该任务执行完毕就可以继续执行其他操作。等到我们需要该任务的执行结果时，再调用凭据对象的相关方法来获取。这样就避免了不必要的等待，增加了系统的并发性。这好比我们去小吃店，同时点了鸭血粉丝汤和生煎包。当我们点餐付完款后，我们拿到手的其实只是一张

类图



- **Promisor**：负责对外暴露可以返回Promise对象的异步方法，并启动异步任务的执行。其主要方法及职责如下：
  - **compute**：启动异步任务的执行，并返回用于获取异步任务执行结果的凭据对象。
- **Promise**：包装异步任务处理结果的凭据对象。负责检测异步任务是否处理完毕、返回和存储异步任务处理结果。其主要方法及职责如下。
  - **getResult**：获取与其所属Promise实例关联的异步任务的执行结果。
  - **setResult**：设置与其所属Promise实例关联的异步任务的执行结果。
  - **isDone**：检测与其所属Promise实例关联的异步任务是否执行完毕。
- **Result**：负责表示异步任务处理结果。具体类型由应用决定。
- **TaskExecutor**：负责真正执行异步任务所代表的计算，并将其计算结果设置到相应的Promise实例。其主要方法及职责如下
  - **run**：执行异步任务所代表的计算。

时序

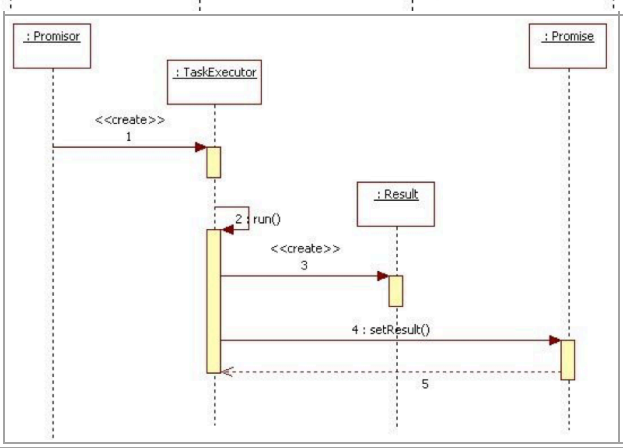


客户代码视角：Promisor#compute()是入口

第1步：客户端代码调用Promisor的异步方法compute。

第2、3步：compute方法创建Promise实例作为该方法的返回值，并返回。

第4步：客户端代码调用其所得到的Promise对象的getResult方法来获取异步任务处理结果。如果此时异步任务执行尚未完成，则getResult方法会阻塞（即调用方代码的运行线程暂时处于阻塞状态）。



任务执行视角：TaskExecutor是真正在后台异步执行任务的线程

第1步：Promisor的异步方法compute创建TaskExecutor实例。

第2步：TaskExecutor的run方法被执行（可以由专门的线程或者线程池[2]来调用run方法）。

第3步：run方法创建表示其执行结果的Result实例。

第4、5步：run方法将其处理结果设置到相应的Promise实例上。

优缺点

Promise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：有关异步编程的细节，如创建新的线程或者提交任务到线程池执行等细节，都被封装在Promisor参与者实例中，而Promise的客户端代码则无须关心这些细节，其编码方式与同步编程并无本质上差别。这点正如清单6-1代码所展示的，客户端代码仅仅需要调用FTPClientUtil的newInstance静态方法，再调用其返回值的get方法，即可获得一个初始化完毕的FTP客户端实例。这本质上还是同步编程。当然，客户端代码也不能完全无视Promise模式的异步编程这一特性：为了减少客户端代码在调用Promise的getResult方法时出现阻塞的可能，客户端代码应该尽可能早地调用Promisor的异步方法，并尽可能晚地调用Promise的getResult方法。这当中间隔的时间可以由客户端代码用来执行其他操作，同时这段时间可以给TaskExecutor用于执行异步任务。

Promise模式一定程度上屏蔽了异步、同步编程的差异。前文我们一直说Promisor对外暴露的compute方法是个异步方法。事实上，如果compute方法是一个同步方法，那么Promise模式的客户端代码的编写方式也是一样的。也就是说，无论compute方法是一个同步方法还是异步方法，Promise客户端代码的编写方式都是一样的。例如，本章案例中FTPClientUtil的newInstance方法如果改成同步方法，我们只需要将其方法体中的语句new Thread（task）.start(); 改为task.run(); 即可。而该案例中的其他代码无须更改。这就在一定程度上屏蔽了同步、异步编程的差异。

而这可以给代码调试或者问题定位带来一定的便利。比如，我们的本意是要将compute方法设计成一个异步方法，但在调试代码的时候发现结果不对，那么我们可以尝试临时将其改为同步方法。若此时原先存在的问题不再出现，则说明问题是compute方法被编码为异步方法后所产生的多线程并发访问控制不正确导致的。

实现要领

异常传递

异常不能跨线程捕捉，通过由客户调用的getResult()函数来把异常从TaskExecutor所在线程传递到客户线程

如果Promisor的compute方法是个异步方法，那么客户端代码在调用完该方法后异步任务可能尚未开始执行。另外，异步任务运行在自己的线程中，而不是compute方法的调用方线程中。因此，异步任务执行过程中产生的异常无法在compute方法中抛出。为了让Promise模式的客户端代码能够捕获到异步任务执行过程中出现的异常，一个可行的办法是让TaskExecutor在执行任务捕获到异常后，将异常对象“记录”到Promise实例的一个专门的实例变量上，然后由Promise实例的getResult方法对该实例变量进行检查。若该实例变量的值不为null，则getResult方法抛出异常。这样，Promise模式的客户端代码通过捕获getResult方法抛出的异常即可“知道”异步任务执行过程中出现的异常。JDK中提供的类java.util.concurrent.FutureTask就是采用这种方法对compute异步方法的异常进行处理的。

支持轮询

提供一个isDone()方法

客户端代码对Promise的getResult的调用可能由于异步任务尚未执行完毕而阻塞，这实际上也是一种等待。虽然我们可以通过尽可能早地调用compute方法并尽可能晚地调用getResult方法来减少这种等待的可能性，但是它仍然可能会出现。某些场景下，我们可能根本不希望进行任何等待。此时，我们需要在调用Promise的getResult方法之前确保异步任务已经执行完毕。因此，Promise需要暴露一个isDone方法用于检测异步任务是否已执行完毕。JDK提供的类java.util.concurrent.FutureTask的isDone方法正是出于这种考虑，它允许我们在“适当”的时候才调用Promise的getResult方法（相当于FutureTask的get方法）。

异步任务并发量

问题	每次调用Promisor#compute()函数(对应例中FTPClientUtil.newInstance())都产生一个线程，并发量可能过大
解决	用ThreadPoolExecutor来创建线程，控制线程数量

ThreadPoolExecutor

```
public class FTPClientUtil {
 private volatile static ThreadPoolExecutor threadPoolExecutor;

 static {
  threadPoolExecutor = new ThreadPoolExecutor(1,Runtime.getRuntime()
      .availableProcessors()  * 2,
      60,
      TimeUnit.SECONDS,
      new ArrayBlockingQueue<Runnable>(10),  new ThreadFactory()  {
       public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setDaemon(true);
        return t;
       }
      },  new ThreadPoolExecutor.CallerRunsPolicy());
 }
}
```

使用线程池中的线程：用threadPoolExecutor.execute(task)来替代new Thread(task).start()

```
public static Future<FTPClientUtil> newInstance(final String ftpServer,
    final String userName, final String password) {

 Callable<FTPClientUtil> callable = new Callable<FTPClientUtil>()  {

  @Override
  public FTPClientUtil call()  throws Exception {
   FTPClientUtil self = new FTPClientUtil();
   self.init(ftpServer, userName, password);
   return self;
  }

 };
 final FutureTask<FTPClientUtil> task = new FutureTask<FTPClientUtil>(
     callable);

 threadPoolExecutor.execute(task);
 return task;
}

private void init(String ftpServer, String userName, String password)
    throws Exception {
 //省略与清单6-2中相同的代码
}

public void upload(File file) throws Exception {
  //省略与清单6-2中相同的代码
 }

 public void disconnect()  {
  //省略与清单6-2中相同的代码
 }

}
```

可复用代码

只有Future<V>可以复用

JDK1.5开始提供的接口java.util.concurrent.Future可以看成是Promise模式中Promise参与者的抽象，其声明如下：

```
public interface Future<V>
```

该接口的类型参数V相当于Promise模式中的Result参与者。该接口定义的方法及其与Promise参与者相关方法之间的对应关系如表6-1所示。

表6-1. 接口java.util.concurrent.Future与Promise参与者的对应关系

接口java.util.concurrent.Future的方法	Promise参与者的方法	功能

get()	getResult()	获取异步任务的执行结果
isDone()	isDone()	检查异步任务是否执行完毕

接口java.util.concurrent.Future的实现类java.util.concurrent.FutureTask可以看作Promise模式的Promise参与者实例。

JDK例

JAX-WS 2.0 API中用于支持调用Web Service的接口javax.xml.ws.Dispatch就使用了Promise模式。该接口用于异步调用Web Service的方法声明如下：

```
Response<T> invokeAsync(T msg)
```

该方法不等对端服务器给响应就返回了（即实现了异步调用Web Service），从而避免了Web Service客户端进行不必要的等待。而客户端需要其调用的Web Service的响应时，可以调用invokeAsync方法的返回值的相关方法来获取。invokeAsync的返回值类型为javax.xml.ws.Response，它继承自java.util.concurrent.Future。因此，javax.xml.ws.Dispatch相当于Promise模式中的Promisor参与者实例，其异步方法invokeAsync（T msg）的返回值相当于Promise参与者实例。

例子

场景 步编程，使得FTP客户端实例初始化和本地文件上传这两个任务能够并发执行，减少不必要的等待。另一方面，我们不希望这种异步编程增加了代码编写的复杂性。这时，Promise模式就可以派上用场了：先开始FTP客户端实例的初始化，并得到一个获取FTP客户端实例的凭据对象。在不必等待FTP客户端实例初始化完毕的情况下，每生成一个本地文件，就通过凭据对象获取FTP客户端实例，再通过该FTP客户端实例将文件上传到目标服务器上。代码如清单6-1所示[3]。

代码

```
public class DataSyncTask implements Runnable {                    使用Promise的线程
 private final Map<String, String> taskParameters;

 public DataSyncTask(Map<String, String> taskParameters) {
  this.taskParameters = taskParameters;
 }

 @Override
 public void run()  {
  String ftpServer = taskParameters.get("server");
  String ftpUserName = taskParameters.get("userName");
 String password = taskParameters.get("password");

 //先开始初始化FTP客户端实例
 Future<FTPClientUtil> ftpClientUtilPromise = FTPClientUtil.newInstance(
     ftpServer, ftpUserName, password);
 //查询数据库生成本地文件
 generateFilesFromDB();

 FTPClientUtil ftpClientUtil = null;
 try {
  // 获取初始化完毕的FTP客户端实例
  ftpClientUtil = ftpClientUtilPromise.get();
 } catch (InterruptedException e) {
  ;
 } catch (ExecutionException e) {
  throw new RuntimeException(e);
 }

 // 上传文件
 uploadFiles(ftpClientUtil);
  //省略其他代码
}

private void generateFilesFromDB()  {
 // 省略其他代码
}

private void uploadFiles(FTPClientUtil ftpClientUtil) {
    Set<File> files = retrieveGeneratedFiles();
    for (File file : files) {
     try {
      ftpClientUtil.upload(file);
     } catch (Exception e) {
      e.printStackTrace();
     }
    }

   }

  private Set<File> retrieveGeneratedFiles()  {
   Set<File> files = new HashSet<File>();
   // 省略其他代码
   return files;
  }

 }
//模式角色: Promise.Promisor、Promise.Result                    Promisor，内嵌了Promisor.compute，Promise
public class FTPClientUtil {
 private final FTPClient ftp = new FTPClient();

 private final Map<String, Boolean> dirCreateMap = new HashMap<String, Bool
ean>();

 private FTPClientUtil()  {

 }
```

```
//模式角色：Promise.Promisor.compute
public static Future<FTPClientUtil> newInstance(final String ftpServer,
    final String userName, final String password) {

 Callable<FTPClientUtil> callable = new Callable<FTPClientUtil>()  {

  @Override
  public FTPClientUtil call()  throws Exception {
   FTPClientUtil self = new FTPClientUtil();
   self.init(ftpServer, userName, password);
   return self;
  }

 };

 //task相当于模式角色：Promise.Promise
 final FutureTask<FTPClientUtil> task = new FutureTask<FTPClientUtil>(
     callable);

 /*
 下面这行代码与本案例的实际代码并不一致，这是为了讨论方便。
 下面新建的线程相当于模式角色：Promise.TaskExecutor
 */
 new Thread(task). start();
 return task;
}

private void init(String ftpServer, String userName, String password)
    throws Exception {

 FTPClientConfig config = new FTPClientConfig();
 ftp.configure(config);

 int reply;
 ftp.connect(ftpServer);

 System.out.print(ftp.getReplyString());

 reply = ftp.getReplyCode();

 if (! FTPReply.isPositiveCompletion(reply))  {
  ftp.disconnect();
  throw new RuntimeException("FTP server refused connection.");
 }
 boolean isOK = ftp.login(userName, password);
  if (isOK) {
  System.out.println(ftp.getReplyString());
 } else {
  throw new RuntimeException("Failed to login." + ftp.getReplyString());
 }
 reply = ftp.cwd("~/subspsync");
 if (! FTPReply.isPositiveCompletion(reply))  {
  ftp.disconnect();
  throw new RuntimeException("Failed to change working directory.reply:"
      + reply);
 } else {
  System.out.println(ftp.getReplyString());
 }
 ftp.setFileType(FTP.ASCII_FILE_TYPE);
}
public void upload(File file) throws Exception {
 InputStream dataIn = new BufferedInputStream(new FileInputStream(file),
     1024 * 8);
 boolean isOK;
 String dirName = file.getParentFile(). getName();
 String fileName = dirName + '/' + file.getName();
 ByteArrayInputStream checkFileInputStream = new ByteArrayInputStream(
     "". getBytes());
 try {
  if (! dirCreateMap.containsKey(dirName))  {
   ftp.makeDirectory(dirName);
   dirCreateMap.put(dirName, null);
  }
  try {
   isOK = ftp.storeFile(fileName, dataIn);
  } catch (IOException e) {
   throw new RuntimeException("Failed to upload " + file, e);
  }
  if (isOK) {
   ftp.storeFile(fileName + ". c",  checkFileInputStream);
  } else {
   throw new RuntimeException("Failed to upload " + file + ", reply:" +
","  + ftp.getReplyString());
  }
 } finally {
  dataIn.close();
 }
}
public void disconnect()  {
 if (ftp.isConnected())  {
  try {
   ftp.disconnect();
  } catch (IOException ioe) {
    // 什么也不做
   }
  }
 }
}
```