发件人：方堃 fangkun119@icloud.com
主题：CH11 Serial Thread Confinement
日期：2017年5月2日 下午12:01
收件人：

CH11 Serial Thread Confinement

2017年4月30日 星期日
下午3:50

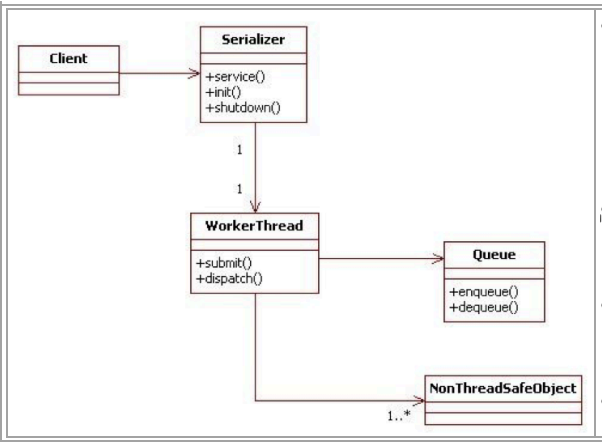
功能

如果并发任务的执行涉及某个非线程安全对象，而我们又希望因此而引入锁，那么我们可以考虑使用Serial Thread Confinement模式。

Serial Thread Confinement模式的核心思想是通过将多个并发的任务存入队列实现任务的串行化，并为这些串行化的任务创建唯一的一个工作者线程进行处理。因此，这个唯一的工作者线程所访问的非线程安全对象由于只有一个线程访问它，对其的访问自然无须加锁，从而避免了锁的开销及由锁可能引发的问题。

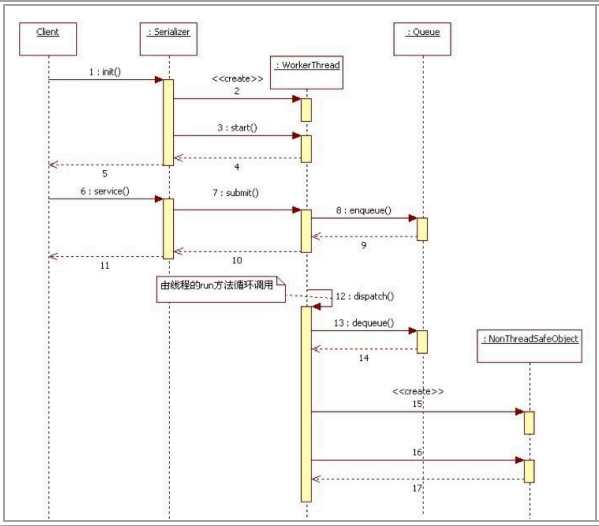
当然，如果我们对并发任务访问的非线程安全对象进行加锁，也能实现任务的串行化从而实现线程安全，另外Serial Thread Confinement模式串行化并发任务所使用的队列本身也会涉及锁。因此，Serial Thread Confinement模式的本质是使用一个开销更小的锁（串行化并发任务时所用队列涉及的锁）去替代另一个可能的开销更大的锁（为保障并发任务所访问的非线程安全对象可能引入的锁）。

类图



- **Serializer：** 负责对外暴露服务接口以及工作者线程的生命周期管理，并将客户端对其的并发服务调用转换为相应的任务以实现服务串行化。其主要方法及职责如下：
  - **service：** 表示Serial Thread Confinement模式对外暴露的服务方法，该方法将客户端对其的并发调用串行化。
  - **init：** 初始化Serial Thread Confinement模式对外暴露的服务。该方法可以启动工作者线程。
  - **shutdown：** 停止Serial Thread Confinement模式对外暴露的服务。该方法可以停止工作者线程。
- **WokerThread：** 工作者线程，负责接收Serializer提交的并发任务以及这些任务的执行。其主要方法及职责如下。
  - **submit：** 用于Serializer提交并发任务，并将这些任务串行化。
  - **dispatch：** 用于执行串行化的任务。
- **Queue：** Serializer和WorkerThread间的缓冲区，用于实现并发任务的串行化。其主要方法及职责如下。
  - **enqueue：** 用于并发任务入队列。
  - **dequeue：** 用于任务出队列。
- **NonThreadSafeObject：** 工作者线程执行任务时所需访问的非线程安全对象。

时序



操作1:框架初始化（主线程中）
1.Serializer#init()
2.create WorkerThread()
3.WorkerThrea#start()

操作2：提交任务（主线程中）
1.Serializer#service()
2.WorkerThread#submit()
3.Queue#enqueue()

操作3：执行任务（WorkerThread中）
1.WorkerThread#run()调用WorkerThread#dispatch()
2.Queue#dequeue
3.create NonThreadSafeObjecrt
4.执行具体任务

使用考点

Serial Thread Confinement模式可以帮助我们在不使用锁的情况下实现线程安全。但是，在实际使用时需要注意Serial Thread Confinement模式自身的开销：将任务串行化所涉及的进出队列以及Serializer创建向WorkerThread提交的任务对象这些动作都有时间和空间的开销。Serial Thread Confinement模式的本质是通过使用一个开销更小的锁来替代另一个开销更大的锁以实现线程安全。因此，实际应用时我们需要注意比较锁的开销：如果采用锁去保障对某个非线程安全对象的访问的线程安全，那么这个锁的开销比起Serial Thread Confinement模式中使用的队列涉及的锁哪个开销更大些？

Serial Thread Confinement模式的典型应用场景包括以下两个。

- **需要使用非线程安全对象，但又不希望引入锁：** 任务的执行涉及非线程安全对象，如果采用锁去保证对这些对象访问的线程安全，这些锁的开销比起将任务通过队列中转涉及锁的开销更大的话，那么我们可以使用Serial Thread Confinement模式。本章案例就属于这种场景。
- **任务的执行涉及I/O操作，但我们不希望过多的I/O线程增加上下文切换：** I/O操作，如文件I/O、网络I/O会增加系统的上下文切换。因此，如果涉及I/O的线程越多，那么系统的处理效率可能反而越低。这是由于过多的上下文切换消耗了过多的系统资源。本章案例之所以采用单线程去处理文件下载，也是出于减少上下文切换的考虑。

任务处结果

如果客户端代码关心任务的处理结果，那么我们可以借用Promise模式（参见第6章）。此时，我们可以让Serializer的service方法返回一个Promise（java.util.concurrent.Future）实例，客户端代码通过该实例可以获取相应任务的处理结果。但是，这样做需要注意一点：多个客户端线程共享同一个Serializer实例意味着多个线程会等待同一个线程（Serializer实例所管理的工作者线程）的处理结果。如果WorkerThread处理过慢，或者客户端代码在Serializer的service方法调用与获取任务的处理结果之间的时间间隔太短，使得WorkerThread没有足够的时间执行相应任务，那么这可能导致客户端线程等待时间过长。当然，我们也可以考虑生成多个Serializer实例。这样，每个Serializer实例的客户端线程在获取任务处理结果时若需要等待也只是等待其调用Serializer实例对应的工作者线程。不过这样一来，我们又需要考虑工作者线程的数量是否会太多。

可复用代码

1.AbstractSerializer

```
public abstract class AbstractSerializer<T, V> {
 private final TerminatableWorkerThread<T, V> workerThread;

 public AbstractSerializer(BlockingQueue<Runnable> workQueue,
     TaskProcessor<T, V> taskProcessor) {
  workerThread = new TerminatableWorkerThread<T, V>(workQueue, taskProcess
 }
```

```
/**
 * Serial Thread Confinement模式Serializer参与者可复用实现。
 *
 * @author Viscent Huang
 *
 * @param <T>
 *          Serializer向WorkerThread所提交的任务对应的类型
 * @param <V>
 *          service方法的返回值类型
 */
```

```
protected abstract T makeTask(Object ... params);
```

```
/**
```

```
protected Future<V> service(Object...  params) throws InterruptedExc
 T task = makeTask(params);
 Future<V> resultPromise = workerThread.submit(task);
 return resultPromise;
}
```

```
/**
 * 该类对外暴露的服务方法。该类的子类需要定义一个命名含义比该方法更为具体的方法（如
 * downloadFile）。
 * 含义具体的服务方法（如downloadFile）可直接调用该方法。
 *
 * @param params
 *          客户端代码调用该方法时所传递的参数列表
 * @return 可借以获取任务处理结果的Promise（参见第6章，Promise模式）实例。
 * @throws InterruptedException
 */
```

```
public void init()  {
 workerThread.start();
}
```

```
/**
 * 初始化该类对外暴露的服务。
 */
```

```
public void shutdown()  {
 workerThread.terminate();
}


}
```

```
/**
 * 停止该类对外暴露的服务。
 */
```

2.WorkerThread

```
public class TerminatableWorkerThread<T, V> extends AbstractTerminatableThread {
 private final BlockingQueue<Runnable> workQueue;

 //负责真正执行任务的对象
 private final TaskProcessor<T, V> taskProcessor;

public TerminatableWorkerThread(BlockingQueue<Runnable> workQueue,
    TaskProcessor<T, V> taskProcessor) {
 this.workQueue = workQueue;
 this.taskProcessor = taskProcessor;
}
```

```
public Future<V> submit(final T task) throws InterruptedExc
 Callable<V> callable = new Callable<V>()  {

  @Override
  public V call()  throws Exception {
   return taskProcessor.doProcess(task);
  }
 };

 FutureTask<V> ft = new FutureTask<V>(callable);
 workQueue.put(ft);

 terminationToken.reservations.incrementAndGet();
 return ft;
}
```

接收并行任务，并将其串行化。

```
@param task
          任务
@return 可借以获取任务处理结果的Promise（参见第6章，Promise模式）实例。
@throws InterruptedException
```

```
@Override
protected void doRun()  throws Exception {
 Runnable ft = workQueue.take();
 try {
  ft.run();
 } finally {
  terminationToken.reservations.decrementAndGet();
 }

}

}
```

执行任务的逻辑

3.TaskProcessor

```
public interface TaskProcessor<T, V> {
 /**
  * 对指定任务进行处理。
  *
  * @param task
  *            任务
  * @return 任务处理结果
  * @throws Exception
  */
 V doProcess(T task) throws Exception;
}
```

T：提交的任务
V：任务执行的结果

4.ReusableCodeExample

利用本章的可复用代码实现Serial Thread Confinement模式，应用程序只需要完成以下几件事情。

1.【必需】定义Serializer提交给WorkerThread的任务对应的类型。

2.【必需】定义AbstractSerializer的子类，并实现其父类定义的makeTask抽象方法。另外该子类需要定义一个名字含义比service更为具体的服务方法。该方法可直接调用其父类的service方法。

3.【必需】定义TaskProcessor接口的实现类。

```
public class ReusableCodeExample {                                    main()

 public static void main(String[]  args) throws InterruptedException,
     ExecutionException {
  SomeService ss = new SomeService();
  ss.init();
  Future<String> result = ss.doSomething("Serial Thread Confinement",  1);

  Thread.sleep(50);

  System.out.println(result.get());

  ss.shutdown();

 }
```

```
 private static class Task {
  public final String message;
  public final int id;

  public Task(String message, int id) {
   this.message = message;
   this.id = id;
  }
 }
```

```
 @Override
 protected Task makeTask(Object...  params) {
  String message = (String) params[0];
  int id = (Integer) params[1];
  return new Task(message, id);
 }
```

```
   public Future<String> doSomething(String message, int id)
       throws InterruptedException {
    Future<String> result = null;

    result = service(message, id);

    return result;
   }

  }

 }
```

JDK例 Swing中的实用工具类SwingUtilities就使用了Serial Thread Confinement模式。该类的invokeLater方法使得多个应用线程能够并发提交与Swing GUI有关的任务。而这些任务仅由唯一的一个线程（即Swing的Event Loop线程）去负责执行。从而保障了Swing GUI层的线程安全。这里，SwingUtilities类相当于Serializer参与者，其invokeLater方法相当于Serializer参与者的service方法。而Swing Event Loop线程则相当于WorkerThread参与者。

例子

场景 某系统需要支持从内网的某台FTP服务器上下载一批文件的功能。该功能的实现会用到一款开源FTP客户端组件。该FTP客户端组件并非线程安全，因此如果多个线程共享其实例可能引起数据错乱。另外，系统中会有多个线程需要从服务器上下载文件，并且我们不希望这些需要下载文件的线程等待要下载的文件下载完毕后才能进行其他处理。因此，这里我们需要使用异步编程。如果我们采用多个线程去并发下载一批文件，并使其中的每个线程都持有一个特有的FTP客户端实例（即使用Thread Specific Storage模式，参见第10章）。那么，线程安全是得到保障了。但这样意味着某一时刻该系统与对端FTP服务器建立了多个网络连接。而这种情况除非必要的时候，否则我们不希望其出现。当然，使用锁也能够实现FTP客户端的线程安全，但是，这样一来从服务器上下载文件其实是串行的（逐一下载文件）。并且，文件下载过程中涉及的网络I/O、文件I/O都会引起上下文切换，为保障FTP客户端线程安全而引入的锁也增加了额外的上下文切换，从而增加系统的负担。

> 功能本身是线程不安全的（被多个线程调用），同时也不适合并发执行（不希望开多个ftp连接，吞吐量考虑）

好处 因此，这里我们可以使用Serial Thread Confinement模式。它可以帮助我们达成以下几个目标。

一、异步编程。这使得客户端线程不必等待要下载的文件下载完毕便可以继续其他处理。

二、不借助锁而实现线程安全。这使得负责文件下载的工作者线程可以安全地使用上述FTP客户端实例，而又不会增加不必要的上下文切换。

代码

```
/*
 * 实现FTP文件下载。
 * 模式角色: SerialThreadConfiment.Serializer
 */
public class MessageFileDownloader {

 // 模式角色: SerialThreadConfiment.WorkerThread
 private final WorkerThread workerThread;

 public MessageFileDownloader(String outputDir, final String ftpServer,
     final String userName, final String password) {

  workerThread = new WorkerThread(outputDir, ftpServer, userName, password);

 }

 public void init()  {
  workerThread.start();
 }

 public void shutdown()  {
  workerThread.terminate();
 }

 public void downloadFile(String file) {
  workerThread.download(file);
 }
 // 模式角色: SerialThreadConfiment.WorkerThread
 private static class WorkerThread extends AbstractTerminatableThread {

  // 模式角色: SerialThreadConfiment.Queue
  private final BlockingQueue<String> workQueue;
  private final Future<FTPClient> ftpClientPromise;
  private final String outputDir;
```

```
  public WorkerThread(String outputDir, final String ftpServer,
      final String userName, final String password) {
   this.workQueue = new ArrayBlockingQueue<String>(100);
   this.outputDir = outputDir + '/';

   this.ftpClientPromise = new FutureTask<FTPClient>(
       new Callable<FTPClient>()  {

        @Override
        public FTPClient call()  throws Exception {
         FTPClient ftpClient = initFTPClient(ftpServer, userName,
password);
         return ftpClient;
        }

       });

   new Thread((FutureTask<FTPClient>) ftpClientPromise). start();
  }
  public void download(String file) {
   try {
    workQueue.put(file);
    terminationToken.reservations.incrementAndGet();
   } catch (InterruptedException e) {
    ;
   }
  }
  private FTPClient initFTPClient(String ftpServer, String userName,
      String password) throws Exception {
   FTPClient ftpClient = new FTPClient();

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

   int reply;
   ftpClient.connect(ftpServer);

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

   reply = ftpClient.getReplyCode();

   if (! FTPReply.isPositiveCompletion(reply))  {
    ftpClient.disconnect();
    throw new RuntimeException("FTP server refused connection.");
   }

   boolean isOK = ftpClient.login(userName, password);
   if (isOK) {
    System.out.println(ftpClient.getReplyString());

   } else {
    throw new RuntimeException("Failed to login."
        + ftpClient.getReplyString());
   }

   reply = ftpClient.cwd("~/messages");
   if (! FTPReply.isPositiveCompletion(reply))  {
    ftpClient.disconnect();
    throw new RuntimeException("Failed to change working
directory.reply:" + reply);
   } else {
    System.out.println(ftpClient.getReplyString());
    }

    ftpClient.setFileType(FTP.ASCII_FILE_TYPE);
    return ftpClient;

   }

   @Override
   protected void doRun()  throws Exception {
    String file = workQueue.take();

    OutputStream os = null;
    try {
     os = new BufferedOutputStream(new FileOutputStream(outputDir +
file));
     ftpClientPromise.get(). retrieveFile(file, os);
    } finally {
     if (null != os) {
      try {
       os.close();
      } catch (Exception e) {
       e.printStackTrace();
      }
     }
     terminationToken.reservations.decrementAndGet();
    }


   @Override
   protected void doCleanup(Exception cause) {
    try {
     ftpClientPromise.get(). disconnect();
    } catch (IOException e) {
```

```
        e.printStackTrace();
        System.out.println(ftpClient.getReplyString());
       }

       ftpClient.setFileType(FTP.ASCII_FILE_TYPE);
       return ftpClient;

      }

      @Override
      protected void doRun()  throws Exception {
       String file = workQueue.take();

       OutputStream os = null;
       try {
        os = new BufferedOutputStream(new FileOutputStream(outputDir +
file));
        ftpClientPromise.get(). retrieveFile(file, os);
       } finally {
        if (null != os) {
         try {
          os.close();
         } catch (Exception e) {
          e.printStackTrace();
         }
        }
        terminationToken.reservations.decrementAndGet();
       }


      @Override
      protected void doCleanup(Exception cause) {
       try {
        ftpClientPromise.get(). disconnect();
       } catch (IOException e) {
        e.printStackTrace();
       } catch (InterruptedException e) {
        e.printStackTrace();
       } catch (ExecutionException e) {
        e.printStackTrace();
       }
      }

     }
    }
```

```
public class SampleClient {
 private static final MessageFileDownloader DOWNLOADER;

 static {
  DOWNLOADER = new MessageFileDownloader("/home/viscent/tmp/incoming",
      "192.168.1.105",  "datacenter",  "abc123");
  DOWNLOADER.init();
 }

 public static void main(String[]  args) {
  DOWNLOADER.downloadFile("abc.xml");
  // 执行其他操作
 }

}
```

使用 Microsoft OneNote for Mac 创建。