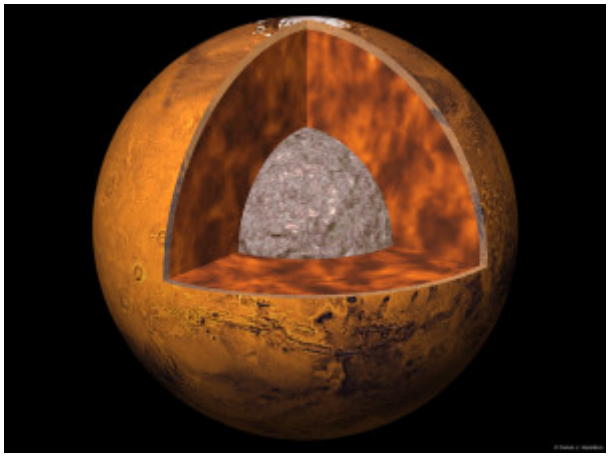
剖析同步器



原文链接 **作者：**Jakob Jenkov **译者：**丁一

虽然许多同步器（如锁，信号量，阻塞队列等）功能上各不相同，但它们的内部设计上却差别不大。换句话说，它们内部的的基础部分是相同（或相似）的。了解这些基础部件能在设计同步器的时候给我们大大的帮助。这就是本文要细说的内容。

注：本文的内容是哥本哈根信息技术大学一个由Jakob Jenkov，Toke Johansen和Lars Bjørn参与的M.Sc.学生项目的部分成果。在此项目期间我们咨询Doug Lea是否知道类似的研究。有趣的是在开发Java 5并发工具包期间他已经提出了类似的结论。Doug Lea的研究，我相信，在《Java Concurrency in Practice》一书中有描述。这本书有一章“剖析同步器”就类似于本文，但不尽相同。

大部分同步器都是用来保护某个区域（临界区）的代码，这些代码可能会被多线程并发访问。要实现这个目标，同步器一般要支持下列功能：

1. 状态
2. 访问条件
3. 状态变化
4. 通知策略
5. Test-and-Set方法
6. Set方法

并不是所有同步器都包含上述部分，也有些并不完全遵照上面的内容。但通常你能从中发现这些部分的一或多个。

状态

同步器中的状态是用来确定某个线程是否有访问权限。在Lock中，状态是boolean类型的，表示当前Lock对象是否处于锁定状态。在BoundedSemaphore中，内部状态包含一个计数器（int类型）和一个上限（int类

型），分别表示当前已经获取的许可数和最大可获取的许可数。BlockingQueue的状态是该队列中元素列表以及队列的最大容量。

下面是Lock和BoundedSemaphore中的两个代码片段。

```
01 public class Lock{
02   //state is kept here
03   private boolean isLocked = false;
04   public synchronized void lock()
05   throws InterruptedException{
06     while(isLocked){
07       wait();
08     }
09     isLocked = true;
10   }
11   ...
12 }
```

```
01 public class BoundedSemaphore {
02   //state is kept here
03   private int signals = 0;
04   private int bound   = 0;
05
06   public BoundedSemaphore(int upperBound){
07     this.bound = upperBound;
08   }
09   public synchronized void take() throws InterruptedException{
10     while(this.signals == bound) wait();
11     this.signal++;
12     this.notify();
13   }
14   ...
15 }
```

访问条件

访问条件决定调用test-and-set-state方法的线程是否可以对状态进行设置。访问条件一般是基于同步器状态的。通常是放在一个while循环里，以避免虚假唤醒问题。访问条件的计算结果要么是true要么是false。

Lock中的访问条件只是简单地检查isLocked的值。根据执行的动作是“获取”还是“释放”，BoundedSemaphore中实际上有两个访问条件。如果某个线程想“获取”许可，将检查signals变量是否达到上限；如果某个线程想“释放”许可，将检查signals变量是否为0。

这里有两个来自Lock和BoundedSemaphore的代码片段，它们都有访问条件。注意观察条件是怎样在while循环中检查的。

```
01 public class Lock{
02   private boolean isLocked = false;
03   public synchronized void lock()
04   throws InterruptedException{
05     //access condition
06     while(isLocked){
07       wait();
08     }
09     isLocked = true;
10   }
```

```
  ...
}
```

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }
  public synchronized void take() throws InterruptedException{
    //access condition
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }
  public synchronized void release() throws InterruptedException{
    //access condition
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

状态变化

一旦一个线程获得了临界区的访问权限，它得改变同步器的状态，让其它线程阻塞，防止它们进入临界区。换而言之，这个状态表示正有一个线程在执行临界区的代码。其它线程想要访问临界区的时候，该状态应该影响到访问条件的结果。

在Lock中，通过代码设置isLocked = true来改变状态，在信号量中，改变状态的是signals–或signals++;

这里有两个状态变化的代码片段：

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    //state change
    isLocked = true;
  }

  public synchronized void unlock(){
    //state change
    isLocked = false;
    notify();
  }
}
```

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

```

```
  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    //state change
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    //state change
    this.signals--;
    this.notify();
  }
}
```

通知策略

一旦某个线程改变了同步器的状态，可能需要通知其它等待的线程状态已经变了。因为也许这个状态的变化会让其它线程的访问条件变为true。

通知策略通常分为三种：

1. 通知所有等待的线程
2. 通知N个等待线程中的任意一个
3. 通知N个等待线程中的某个指定的线程

通知所有等待的线程非常简单。所有等待的线程都调用的同一个对象上的wait()方法，某个线程想要通知它们只需在这个对象上调用notifyAll()方法。

通知等待线程中的任意一个也很简单，只需将notifyAll()调用换成notify()即可。调用notify方法没办法确定唤醒的是哪一个线程，也就是“等待线程中的任意一个”。

有时候可能需要通知指定的线程而非任意一个等待的线程。例如，如果你想保证线程被通知的顺序与它们进入同步块的顺序一致，或按某种优先级的顺序来通知。想要实现这种需求，每个等待的线程必须在其自有的对象上调用wait()。当通知线程想要通知某个特定的等待线程时，调用该线程自有对象的notify()方法即可。饥饿和公平中有这样的例子。

下面是通知策略的一个例子（通知任意一个等待线程）：

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      //wait strategy - related to notification strategy
      wait();
    }
    isLocked = true;
```

```
  }

  public synchronized void unlock(){
    isLocked = false;
    notify(); //notification strategy
  }
}
```

Test-and-Set方法

同步器中最常见的有两种类型的方法，test-and-set是第一种（set是另一种）。Test-and-set的意思是，调用这个方法的线程检查访问条件，如若满足，该线程设置同步器的内部状态来表示它已经获得了访问权限。

状态的改变通常使其它试图获取访问权限的线程计算条件状态时得到false的结果，但并不一定总是如此。例如，在读写锁中，获取读锁的线程会更新读写锁的状态来表示它获取到了读锁，但是，只要没有线程请求写锁，其它请求读锁的线程也能成功。

test-and-set很有必要是原子的，也就是说在某个线程检查和设置状态期间，不允许有其它线程在test-and-set方法中执行。

test-and-set方法的程序流通常遵照下面的顺序：

1. 如有必要，在检查前先设置状态
2. 检查访问条件
3. 如果访问条件不满足，则等待
4. 如果访问条件满足，设置状态，如有必要还要通知等待线程

下面的ReadWriteLock类的lockWrite()方法展示了test-and-set方法。调用lockWrite()的线程在检查之前先设置状态(writeRequests++)。然后检查canGrantWriteAccess()中的访问条件，如果检查通过，在退出方法之前再次设置内部状态。这个方法中没有去通知等待线程。

```
public class ReadWriteLock{
    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

    ...

    public synchronized void lockWrite() throws InterruptedException{
      writeRequests++;
      Thread callingThread = Thread.currentThread();
      while(! canGrantWriteAccess(callingThread)){
        wait();
      }
      writeRequests--;
      writeAccesses++;
      writingThread = callingThread;
```

```
    }

    ...
}
```

下面的BoundedSemaphore类有两个test-and-set方法：take()和release()。两个方法都有检查和设置内部状态。

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

set方法

set方法是同步器中常见的第二种方法。set方法仅是设置同步器的内部状态，而不先做检查。set方法的一个典型例子是Lock类中的unlock()方法。持有锁的某个线程总是能够成功解锁，而不需要检查该锁是否处于解锁状态。

set方法的程序流通常如下：

1. 设置内部状态
2. 通知等待的线程

这里是unlock()方法的一个例子：

```
public class Lock{
  private boolean isLocked = false;

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

有点CAS的味道.

suli
2016/04/14 3:22下午

关于通知策略的讲解，和条件队列的wait/notify,await/signal 混淆概念了吧，实际上查看AQS源码会发现，同步器的通知实际上是由已经获取状态的线程对未获取状态的线程的唤醒，底层使用的LockSupport.park/unpark方式，而条件队列的await/signal 等api是依赖同步器的阻塞和唤醒机制的。

suli
2016/04/14 3:35下午

删除评论