一個模擬 Hoare Monitor 的 Java 設計樣式
A Design Pattern for Simulating Hoare Monitor in Java

陳玉龍
Yu-Lung Chen

國立台灣科技大學電機研究所
台北市基隆路四段 43 號
yulung@pps.ee.ntust.edu.tw

陳 恭
Kung Chen

國立台灣科技大學資訊管理系
台北市基隆路四段 43 號
chenk@cs.ntust.edu.tw

摘 要

Java 語言內建有多線程的功能，並提供依 Hoare monitor 設計的機制來協調線程的執行以確保共有資料之一致性。但 Java 並未提供 Hoare monitor 中的條件變數的功能，使得在協調線程時，為避免隨機喚醒可能造成的問題，常需採取效率不彰的全體喚醒的方式。針對此問題，本文提出一模擬 Hoare monitor 的設計樣式，讓多線程的 Java 程式在協調時，不僅不必盲目地喚醒全部的線程，而且所有依 Hoare monitor 設計的演算法都可直接地引用。

關鍵字：爪哇，線程協調，設計樣式。

Abstract

Java provides explicit multi-threading capabilities through thread classes and a language level synchronization construct based on Hoare's monitor. However, the condition variable mechanism that coordinates the passage of threads through the monitor is missing in Java. The wait/notify methods of Java do not work with a particular condition variable. Consequently, programmers are forced to use the highly inefficient notifyAll method to avoid the problems that may result from the arbitrary selection among threads waiting for a shared resource by the notify method. This paper presents a design pattern for simulating Hoare monitor in Java. Using this pattern, programmers can not only define condition variables to coordinate threads for different purposes without using notifyAll, but also adapt algorithms based on Hoare monitor to Java easily.

Key Words ： Java, Thread Synchronization, Hoare Monitor, Design Pattern.

本論文國科會計畫編號 ：NSC86-2213-E-011-081

1. 簡介

多線程(multi-threaded)程式的執行過程中，由於有多個線程可同時處於執行狀態，必須採用適當之線程協調(thread synchronization)機制，以避免諸線程在存取共有資料時所可能導致的資料不一致性[1]。在眾多線程協調機制中， Hoare 教授所設計的monitor[2]深獲好評，為許多系統採用。其作法是將共有資料及其操作程序集中封裝，限定諸線程必須依序透過這些程序來存取共有資料以達操作互斥(mutual exclusion)之要求；其次， monitor 中可定義條件變數(condition variable)來表達各種需要協調的狀況，以等待(wait)或喚醒(signal or notify)特定線程的方式供各線程協調彼此的步調。

Java 語言[3]內建有多線程的功能，並提供依 Hoare monitor 設計的機制來協調線程的執行以確保共有資料之一致性。但仔細考察 Java monitor 的設計[4]後，我們發現其與 Hoare monitor 間存有一些差異，其中最主要的是 Java monitor 並沒有提供條件變數的功能，影響所及，在撰寫 Java 多線程程式時，不能直接沿用依 Hoare monitor 所設計的諸多演算法，定義不同的條件變數來處理有不同目的線程間的協調工作，而必須採用全體喚醒(notifyAll)此一效率較差的方式來完成線程協調。

針對此一問題，本論文提出一設計樣式(design pattern)[5]的解決之道，包括一供製造條件變數的 class 、一共用的 monitor class 與應用這兩個 classes 來定義各種 monitors 的樣式規則。此設計樣式簡單明瞭，模組性(modularity)高；依此方案，凡需要依不同條件來進行線程協調的 Java 程式，都可以直接而且有效率地表達出來。

本篇論文的組織如下：第二節簡單描述 Java monitor 的運作原則，指出其與 Hoare monitor 的差異所在，並介紹 Java 線程協調的設計樣式、可能造成

效率低落的原因與相關研究。第三節提出以 Java 模擬 Hoare monitor 的設計樣式。第四節則舉兩個古典問題來闡明此一樣式。最後於第五節做一結論。

2. Java Monitor

Java 爲一物件導向程式語言，除了少數原始型態外，所有的資料皆以 class 定義、以物件的形式存在。Monitor 將資料及其操作程序集中封裝的作法與物件一樣，所以 Java 很自然地以物件來實現 monitor 。實際上， Java 的每個物件都伴隨有一個 lock ，物件若含有以 synchronized 修飾符所界定的 methods ，此 lock 就會啓動，該物件即成爲一 monitor ，限制僅有取得 lock 的線程可以執行其 synchronized methods 。換言之， Java 物件皆可爲 monitor, 物件中的 synchronized methods 則相當於 monitor procedures 。

除了 lock 之外， Java 的每個物件還伴隨有一個 wait set 供 wait 、 notify 和 notifyAll 三方法使用。當線程在取得 lock 後，執行 synchronized methods 時，若呼叫 wait() ，會被放入 wait set 中 ,並釋放 lock 允許其它線程進入 monitor 中；若呼叫 notify()，Java 會隨機取回在 wait set 中的某一線程，放回 run queue 中，允許它跟其它線程競爭 CPU 的使用權；若呼叫 notifyAll()，則所有在 wait set 中的線程都會被喚醒來競爭 CPU 的使用權。其中 wait 與 notify 和 Hoare monitor 中的 wait 與 signal 類似， notifyAll 則相當於某些系統中的 broadcast[6]。

Java monitor 與 Hoare monitor 有兩個主要不同點。第一，每個 Java monitor(物件)只有一個 wait set ，而 Hoare monitor 中可定義多個條件變數，每個條件變數都有專屬的 wait set ； Java 的 wait 與 notify 方法操作的對象物件固定，就是 monitor 本身，而 Hoare monitor 中的 wait 與 signal 方法則是作用於條件變數(c.wait(), c.notify())。所以線程因故需要等待時， Hoare monitor 可透過條件變數來提供多個 wait sets ，而 Java monitor 僅有一 wait set 可用。換句話說， Java monitor 僅支援單一條件。

其次， Java monitor 的 notifying 線程在 notify 一個 waiting 線程後，被 notified 的線程並不會立即取得 CPU 的使用權，而必須與其它的線程一起競爭 CPU 的使用權。 Hoare monitor 則假定當線程去 signal 一個 waiting 線程後，會把控制權轉給被 signal 的線程，等被 signal 的線程離開 monitor 後，再把控制權交回 signaling 線程。

Synchronization Patterns

從以上的描述，我們可以對比 Hoare 與 Java 的線程協調樣式如下。

Hoare synchronization pattern:

```
void Hoare_monitor_procedure() {
      if (condition is false ) someCond.wait();
      ...other operations...
      someCond.signal();
}
```

Java synchronization pattern:

```
void synchronized JavaMethod() {
      while (condition is false) wait();
      ...other operations...
      notifyAll();
}
```

由於 Java monitor 中被 notified 的線程並不會立即取得 CPU 的使用權，所以必須採用 while 迴圈來測試，以防其它線程在 notified 的線程繼續執行之前將條件更改了。但爲何要用 notifyAll 而不用 notify? 因爲 Java 僅支援單一 wait set ，所有的 waiting 線程不論其等待原因爲何，都放在同一個 wait set 中，而 notify 隨機選取一線程，並不保證能選到所希望喚醒的線程。所以只好將所有在 wait set 的線程以 notifyAll 全部喚醒，然後再利用 while 判斷式挑選，若挑選的線程不是所希望的，就再度放回 wait set ，若爲所希望的線程，則此線程會跳出 while 迴圈，繼續執行程式。

例：定義一 single buffer 供生產者與消費者共用，其 monitor code 如下：

```
class BufferMonitor {
      private int count=0;
      private Object buffer;     // buffer size = 1

      public synchronized void put(Object x){
            while (count == 1) wait();     // wait1
            buffer=x;
            count=1;
            notify();
      }
      public synchronized Object get(){
            Object x;
```

```
        while (count == 0) wait();     // wait2
        x=buffer;
        count=0;
        notify();
        return x;
    }
}
```

假設有三個線程： ProducerA 、 ProducerB 和 ConsumerC，前兩個會反覆呼叫 put，第三個則反覆呼叫 get 。考慮以下的執行順序： ProducerA 先執行完 put 後，輪到 ProducerB 執行 put 時，它會在 wait1 處等待；接著如果 ProducerA 再次執行 put，也會停在 wait1 處等待。然後 ConsumerC 執行 get，假設在離開前 notify 選到 ProducerB，但接著 ConsumerC 又再次進入 get 中而被暫停在 wait2 處；然後 ProducerB 繼續執行至 notify()時，若選到 ProducerA 而不是預期的 ConsumerC ，結果 ProducerA 因爲 while 的 condition 仍然成立， 而再度進入 wait set 中，而 ProducerB 接著進入 put 後也停在 wait1 處，造成三個線程都停在 wait set 中，形成死鎖(deadlock)。

爲了避免因喚醒非預期線程而造成困擾，故採用 notifyAll 盲目地喚醒所有線程，這樣的的作法顯然會導致效率的低落，尤其是當 wait set 中有很多等待的線程時，系統花在篩選與搬移線程的時間必然相當可觀，實不可取。

相關研究

Birrell[6]稱這種不加區分而喚醒全部線程的作法爲 spurious wakeup 。 Cargill[7]則針對 Java 的情形而稱其爲 haphazard notification，他的主要關切並不是條件變數而是喚醒對象的完全掌控，甚至包括按時間先後的順序來喚醒等待中的線程，所以他的解法較爲複雜，模組性也不如我們的作法。

3. 模擬 Hoare Monitor

本節詳述我們所提出的模擬 Hoare mointor 的方案，包括一供製造條件變數的 class 、一共用的 monitor class 與應用這兩個 classes 來定義各種 monitors 的樣式規則。茲分述如下。

Condition Class

從上一節的分析我們知道必須要提供定義條件變數的功能，以便處理不同需求的線程協調；而實際上，條件變數提供的就是 wait set ，所以我們選擇以 Java monitor 的方式來定義條件變數：

```
class Condition
    private   int condCounter=0;
    public synchronized void condWait(Monitor m){
            condCounter++;
            m.releaseLock();
            wait();
    }
    public synchronized void condSignal(Monitor m){
            if (condCounter > 0)
              condWaiter--;
              notify();
              m.signalFlag = true;
            }
    }
    public synchronized boolean queue()
            return (condCounter > 0) ? true : false;
    }
}
```

此 Condition class 本身就是個 Java monitor，內含一資料 condCouniter，記載在其 wait set 中的線程數；兩個 synchronized methods — condWait 和 condSignal ，是用來模擬 Hoare monitor 的 wait 和 signal，因爲我們的條件變數是獨立於 monitors，所以 condWait 和 condSignal 都得傳入一 monitor 參數，以建立兩者的關連。另外有一個 queue method 用來判斷 wait set 內是否有線程。

Monitor Class

其次，我們定義一基本的 Monitor class，提供所有 monitors 都需要的互斥功能，任一特定的 monitor 便可從此 class 繼承這些操作程序。

```
class Monitor {
    private boolean lock = false;
    private   int counter = 0;
    public boolean signalFlag = false;
    public synchronized void setLock() {
        if (lock) {
            counter++;
            wait();
        } else lock = true;
    }
```

```
    public synchronized void releaseLock() {
        if (!signalFlag) {
            if (counter != 0) {
                counter--;
                notify();
            } else lock = false;
        } else signalFlag = false;
    }
}
```

此 Monitor class 以一個 lock 變數來限制線程進入 monitor ：每個線程都必須先呼叫並完成 setLock ，才能執行某個 monitor 的操作程序；若已有線程在 mointor 內，則必須在 wait set 內等待。如果不使用此 lock 變數來達到互斥，而逕自將 monitor 的操作程序宣告爲 synchronized methods ，則可能在使用條件變數的時候構成 nested monitors 而導致死鎖[8]。

此外，線程在離開 monitor 的程序前，或執行 condWait 時，必須呼叫 releaseLock 以讓出 monitor 的使用權。但有可能在此之前，該線程已執行了 condSignal 且透過 notify 選了一線程來繼續其未完成的 monitor 操作程序，可是 Java 的 notify 並不立即轉移控制權，爲避免因這樣而允許兩線程同時進入 monitor ，我們定義了一個變數 signalFlag 來確保只會讓一個線程進入 monitor 。也就是說，等待在條件變數的 wait set 的線程要比等待在 monitor 的 wait set 的線程優先取得繼續執行的權利。這一作法也使得我們在決定是否要等待時，可以跟 Hoare monitor 一樣只用 if 敘述而不必用 while 迴圈。

特定的 monitors

有了上述兩個 classes 後，我們就可以很容易地以 Java 來按 Hoare monitor 的方式定義各種特定的 monitors 。首先，將要定義的 monitor 宣告成上述 Monitor class 的 subclass ，以繼承 lock 等確保線程互斥的成員。其次，可應需要定義若干個 Condition 物件作爲條件變數之用。接著就按照下列樣式定義所需的 monitor 程序。

```
returnType   method(...parameters...) {
    setLock();
    ...operation...
    // may include if (!condition) c.condWait(this) and
    //   c'.condSignal(this) for some conditions c, c'
    releaseLock();
```

注意這些 monitor 程序都不是 synchronized methods ，其間之互斥性是透過 lock 變數的操作而達成：程序開始時呼叫 setLock ，結束前呼叫 releaseLock 。如因某條件 c 而須等待時，可呼叫 c.condWait ；若要喚醒在條件變數 c'等待的線程，可呼叫 c'.condSignal 。

4. 實作範例

依我們的樣式，很容易地就可以用 Java 來實現 Hoare monitors 。這裡從 Hoare 的原始論文[2]選擇兩個古典的線程協調問題來作爲實作範例。

問題一： Bounded buffer Problem

此問題是要設計一容量有限的 buffer 供多個線程交換資料。其中一類線程(生產者)不斷地製造物件以放到 buffer 內，另一類線程(消費者)則不斷地從 buffer 取走物件。

列表一是依我們的樣式，將 Hoare 的演算法以 Java 實現的 monitor 程式，外觀和實質上都與原來的算法非常類似：生產者呼叫 append 存放物件至 buffer 並共用條件變數 nonempty 的 wait set ，消費者則呼叫 remove 以從 buffer 取出物件並使用條件變數 nonfull 的 wait set 。

問題二： Readers and Writers Problem

此問題主要是要避免多個 Readers 和 Writers 對某些共有資料同時作存取時可能造成的資料不一致性。基本上，許多 readers 可以同時讀取此資料，但當有 Writer 要更新此資料時，其它的 readers 與 writers 都不能存取；但爲了避免 readers 一個接一個地去讀資料而造成 writers 無法去更新資料，規定一旦有 writer 等著要寫，就不允許再有新的 readers 去讀資料，而且目前正在讀的 readers 結束後，就得讓一等待的 writer 去更新資料。

列表二是依我們的樣式，將 Hoare 的演算法以 Java 實現的 monitor 程式，同樣地，這個 monitor 在外觀和實質上都與 Hoare 原來的演算法非常類似：readers 在讀資料前後必須分別呼叫 startRead 和 endRead ，共用條件變數 OKtoread 的 wait set ，writers 則必須分別呼叫 startWrite 和 endWrite ，共用條件變數 OKtowrite 的 wait set 。

5. 結論

Java的線程協調機制與Hoare monitor類似但不盡相同，由於缺少定義條件變數的功能，在喚醒等待中的線程時，不能喚醒特定的線程，而必須要以效率較差的全體喚醒的方式來避免像死鎖等問題。本論文提出一以 Java 模擬 Hoare monitor 的設計樣式，此設計樣式簡單明瞭、模組性高，可以讓多線程的 Java 程式在協調時，不僅不必盲目地喚醒全部的線程，而且所有依 Hoare monitor 設計的演算法都可直接了當地引用。

參考文獻

[1] A. Silberchatz and P.B. Galvin, Operating System Concepts, Fourth Edition, Addison-Wesley, 1994.

[2] C.A.R. Hoare, Monitors: An Operating System Structuring Concept. CACM 17:10, pp.549-557, 1974.

[3] K. Arnold and J. Gosling, The Java Programming Language. Addison-Wesley, 1996.

[4] J. Gosling et al., The Java Language Specification. Addison-Wesley, 1996. Chapter 17.

[5] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[6] A.D. Birrell, An Introduction to Programming with Threads. Technical Report #35, Systems Research Center, Digital Equipment, Palo Alto, CA, 1989.

[7] T. Cargill, Specific Notification for Java Thread Synchronization. Conference on Pattern Language of Programming '96, Allerton Park, IL, Sept. 1996.

[8] D. Lea, Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, 1997.

列表一： Bounded Buffer Monitor

```
class BoundedBufferMonitor
                    extends Monitor{

  private Object buffer[];
  private int count=0,lastPointer=0;
  private Condition nonempty,nonfull;

  public BoundedBufferMonitor() {
    buffer = new Object[N];
    nonempty  = new Condition();
    nonfull = new Condition();
  }

  public void append(Object x) {
    setLock();

    if (count == N)
          nonfull.condWait(this);
    buffer[lastPointer] = x;
    lastPointer =
              (lastPointer+1)mod N;
    count++;
    nonempty.condSignal(this);

    releaseLock();
  }

  public Object remove(){
    Object x;
    setLock();

    if (count == 0)
          nonempty.condWait(this);
    x = buffer[lastPointer];
    lastPointer =
          (lastPointer-count) mod N;
    count--;
    nonfull.Signal(this);

    releaseLock();
    return x;
  }
}
```

列表二: Readers and Writers Mointor

```
class RWMonitor extends Monitor {
  private  int readerCount = 0;
  private  boolean busy = false;
  private Condition OKtoread,
                    OKtowrite;

  public RWMonitor() {
    OKtoread  = new Condition();
    OKtowrite = new Condition();
  }

  public void startRead() {
    setLock();

    if (busy||OKtowrite.queue())
         OKtoread.condWait(this);
    readerCount++;
    OKtoread.condSignal(this);

    releaseLock();
  }

  public void endRead() {
    setLock();

    readerCount--;
    if (readerCount==0)
      OKtowrite.condSignal(this);

    releaseLock();
  }

  public void startWrite() {
    setLock();

    if (busy || readerCount!=0)
        OKtowrite.condWait(this);
    busy=true;

    releaseLock();
  }

  public void endWrite() {
    setLock();

    busy = false;
    if (OKtoread.queue())
         OKtoread.condSignal(this);
    else  OKtowrite.condSignal(this);

    releaseLock();
  }
}
```