第 5 章

極簡同步化的技巧

在前面兩章中，我們討論過讓物件具有 thread 安全性的方法，能夠讓它們在同一時間於兩個或以上的 thread 中使用。thread 的安全性是好的 thread 程式設計中最重要的項目；race condition 是非常難以複製與改正的。

在本章中，我們會藉由兩個相關的議題完成對資料同步化與 thread 安全性的討論。開始是 Java 記憶體模型的討論，它定義了變數實際上是如何被 thread 存取。這個模型有著一些令人驚訝的效果；我們會對前面章節澄清的議題之一就是何謂 thread 被模型化成指令的清單。在解釋完記憶體模型之後，我們會討論到 volatile 變數如何融入以及為何它們在多個 thread 間能夠安全的使用。這個議題全部都與避免同步化有關。

之後會檢視資料同步化的其他途徑：使用 atomic 的 class。這一組於 J2SE 5.0 引進的 class 能夠讓特定型別資料於特定的操作上被定義為 atomic 化的。這些 class 在操作上提供了一個相當好的資料抽象概念，同時也會防止可能由操作本身所引發出的 race condition。這些 class 在採用不同的方式以達成同步化這方面也是很有意思：相較於直接明確的將資料存取同步化，它們使用一種方式能夠允許 race condition 的發生，但確保 race condition 都是良性的。因此，這些 class 能夠自動的避免明確的同步化。

能避免同步化嗎？

threaded 程式的開發者通常都對同步化有妄想症狀。有許多關於因為過度或不正確的同步化導致程式效能很差的恐怖故事在坊間流傳。如果對某特定的 lock 有非常多的競爭，取得該 lock 會因為下面兩個因素讓成本非常的高昂：

- 在許多 virtual machine 的實作上取得有競爭與無競爭 lock 的程式寫作方式是不相同的。取得有競爭的 lock 需要在 virtual machine 的層次上執行更多的程式碼。無論如何，相反的說法也是對的：取得無競爭的 lock 是相當廉價的作業。
- 在有競爭的 lock 能被取得之前，現存的持有者必須要先將其釋放。要取得有競爭 lock 的 thread 總是必須要等到 lock 被釋放開。

有競爭與無競爭的 Lock

競爭（contended）與無競爭（uncontended）此兩個術語是指有多少的 thread 在某特定的 lock 上操作。沒有被任何 thread 持有的 lock 是個無競爭的 lock：第一個要試著取得它的 thread 會立即的成功。

當某個 thread 嚐試要取得一個已經被其他 thread 持有的 lock 時，這個 lock 就變成了有競爭的 lock。一個有競爭的 lock 至少有一個 thread 在等著它；也有可能是很多個。注意到有競爭的 lock 會於 thread 不再要等待取得它時變成個沒有競爭的 lock。

以實用的觀點來看，上述的第二條是最顯著的：如果其他人佔有了 lock，你就必須等它，這會讓程式的效能大幅的下滑。我們會在第十四章討論到與 thread 效能有關的作業。

這個情況會讓程式設計師想嚐試在程式中限制同步化。這是個不錯的主意；你當然會比出現無謂的運算還更不想見到無謂的同步化出現在程式中。但是否有辦法能夠全然的避掉同步化？

我們已經看過在某種情況下這答案可以是“Yes”：你可以對 instance 變數使用 volatile 關鍵字。這些變數無法不完整的儲存，所以在讀取它們的時候，你知道你是在讀取有效的值：最近一次被存入此變數的值。在本章稍後，我們會看到其他能夠以特定的 class 來讓無同步化資料存取也能夠被接受的例子。

但那也就是可以避免同步化唯一的情形。在其他的狀況中，如果有多個 thread 存取相同的一組資料，你必須明確的同步化所有對該資料的存取以防止各種的 race condition。

這個原因與電腦將程式最佳化有關係。電腦會執行兩種主要的最佳化動作：建構暫存器來保持資料與重排（reordering）述句。

暫存器的效應

電腦有一定數量的主記憶體用來儲存與程式有關的資料。在宣告一個變數的時候（像是在數個 class 中都有用到的 done 旗標），電腦在一旁設置一個特定的記憶體位置來保持該變數的值。

大多數的 CPU 都能夠直接的操作保持在主記憶中的資料。其他的 CPU 則只能讀寫主記憶體的位置；這些電腦必須要將資料從主記憶體中讀到暫存器，對暫存器操作，然後將資料存回記憶體。就算是能夠對主記憶體中的資料直接操作的 CPU 通常也是有一組可以保持資料的暫存器，且對暫存器上的資料操作通常比對主記憶體上的資料操作要快上許多。因此，在電腦執行程式碼時，到處都有在使用暫存器。

從邏輯的觀點上來看，每個 thread 都有自己的一組暫存器。當作業系統將某 thread 指派給 CPU 時，它會把該 thread 特有的資訊載入 CPU 的暫存器中；在指派不同的 thread 給 CPU 之前，它會將暫存器的資訊存下來。所以 thread 間決不會共用到保持在暫存器的資料。

讓我們來看一下這是如何套用在 Java 程式上。在要終結一個 thread 的時候，我們通常會使用 done 旗標。此 thread（或 runnable 物件）帶有像下面這樣的程式碼：

```
public void run() {
    while (!done) {
        foo();
    }
}
public void setDone() {
    done = true;
}
```

假設說我們是這樣的宣告 done：

```
private boolean done = false;
```

這會將某個特定的記憶體位置（例如說 0xff12345）與 done 變數結合起來，並將此記憶體位置的值設為 0（false 值在機器上的表示法）。

然後 run() 這個 method 會被編譯成一組指令：

```
Begin method run
Load register r1 with memory location 0Xff12345
Label L1:
Test if register r1 == 1
If true branch to L2
Call method foo
Branch to L1
Label L2:
End method run
```

在此同時，setDone() 看起來會像是下面這樣：

```
Begin method setDone
Store 1 into memory location 0xff12345
End method setDone
```

問題在這裡：run() 絕不會將記憶體位置 0xff12345 的內容重新載入暫存器 r1。因此 run() 這個 method 永遠不會停止。

然而，假設我們將 done 這樣定義：

```
private volatile boolean done = false;
```

現在 run() 在邏輯上看起來像是這樣：

```
Begin method run
Label L1:
Test if memory location 0xff12345 == 1
If true branch to L2
Call method foo
Branch to L1
Label L2:
End method
```

使用 volatile 關鍵字能夠確保變數節不會保持在記憶體中。這能夠保證變數是真正的分享於 thread 之間【註】。

還記得我們可能會將此程式碼實作成同步化對 done 旗標的存取（而不是設定 done 旗標為 volatile）。這會可行是因為同步化邊界標示出 virtual machine 必須要將它的暫存器視為無效的。當 virtual machine 進入 synchronized 的 method 或區段的時候，它必須要重新載入本來已經 cached 到自有暫存器上的資料。在 virtual machine 離開同步化的 method 或區段值之前，它必須要把自有暫存器存入主記憶體中。

重排述句的效應

開發者經常希望可藉由述句的執行順序來避開同步化。假設我們決定要紀錄打字遊戲在數個回合間的總分。resetScore() 這個 method 就有可能會被寫成像下面這樣：

註　只要不違反我們所要求的語意，virtual machine 還是可以用暫存器來處理 volatile 變數。這只是個必須遵守的原則，而不是實際施行的規範。

```
public int currentScore, totalScore, ◘nalScore
public void resetScore(boolean done) {
    totalScore += currentScore;
    if (done) {
        ◘nalScore = totalScore;
        currentScore = 0;
    }
}

public int getFinalScore() {
    if (currentScore == 0)
        return ◘nalScore;
    return -1;
}
```

這會引發 race condition，因為 thread t1 與 thread t2 的執行順序可能會是這樣：

```
Thread1: Update total score
Thread2: See if currentScore == 0
Thread2: Return -1
Thread1: Update ◘nalScore
Thread1: Set currentScore == 0
```

這不是程式邏輯必然的結果。如果週期性的檢查分數，這一次會得到 -1，但下一次會得到正確得結果。依據程式的需求，這可能會是非常可以接受的。

然而，你無法依賴像這樣排列過的執行順序。virtual machine 可能會決定在它指派最終結果前將 0 儲存於 currentScore 是更有效率的。這樣的決策是在執行期間根據執行程式的硬體所作出的。在這種情形下，跑出的是這個次序：

```
Thread1: Update total score
Thread1: Set currentScore == 0
Thread2: See if currentScore == 0
Thread2: Return ◘nalScore
Thread1: Update ◘nalScore
```

現在 race condition 就會引發一個問題：回傳錯誤的最終分數。注意到不管變數是否有定義為 volatile 都沒有差別：包含有 volatile 變數的述句也會跟其他述句一樣被重新排過。

此處唯一能夠幫上忙的就是同步化。如果 resetScore() 與 getFinalScore() 是 synchronized 過的，method 中的述句是否是被重排過就不重要了，因為同步化能夠防止執行 method 的 thread 交錯。

synchronized 的區段同樣也能夠防止述句的重排。virtual machine 不能將述句從 synchronized 區段移動到 synchronized 區段之外。然而，要注意到反過來說就不對了：在 synchronized 區段之前的述句可以被移動到區段內，且在 synchronized 區段後面的述句也可以被移動到區段內。

雙重檢查的 Locking

這個 design pattern 在第一次被提出時受到相當的重視，但它現在已經完全的被唾棄了。但它還是偶爾又會被拿出來，所以在此將細節提出來以滿足好奇心。

有一個例子是開發者受誘以 lazy initialization 來避開同步化。在此種典型中，有個帶有參考、要花時間建構的物件，所以開發者將物件的建構給延後：

```
Foo foo;
public void useFoo() {
    if (foo == null) {
        synchronized(this) {
            if (foo == null)
                foo = new Foo();
        }
    }
    foo.invoke();
}
```

在此處開發者的目標是要在一旦 foo 物件被初始化之後防止同步化。很不幸的，這個 pattern 並不行，原因是因為我們剛剛所看過的那個問題。特別是 foo 的值可以在 foo 的 constructor 被呼叫前就儲存；之後另一個 thread 進入 useFoo() 會在 foo 的 constructor 完成之前呼叫 foo.invoke()。假使 foo 是個 volatile 的 primitive（但不是個 primitive 物件），這就還算能正常運作，只要你不在乎 foo 會被初始化一次以上（且對 foo 多次的初始化也保證會產出相同的值）。

想要知道雙重檢查（double-checked）的 locking 這個 pattern 以及 Java 記憶體模型的處置方法，請見 *http://www.cs.umd.edu/~pugh/java/memoryModel/*。

Atomic 變數

同步化的目的是在防止 race condition 導致資料在不一致或異動中途狀態被使用到。多個 thread 會在程式段由同步化保護的期間被禁止作競爭。這並不意味著產出或者 thread 的執行的順序是命中注定的：thread 間可能在同步化的程式碼區段之前就開始競爭了。且如果 thread 正在等待相同的同步化 lock，thread 執行 synchronized 程式碼的順序是由 lock 的授予（一般來說這是由平台所決定且結果是非注定的）來決定的。

這是微妙但又重要的一點：並不是所有的 race condition 都應該要避免。只有在無 thread 安全性的程式段中的 race condition 才會被認為是個問題。我們可以使用兩種辦法中的一種來改正這個問題。我們可以用 synchronized 的程式碼來防止 race condition 的發生，或是將程式設計成無須同步化（或僅使用最少的同步化）就具有 thread 安全性。

我們相信兩種技術你都有試過。在第二個情形中，儘可能縮小同步化的範圍並重新組織程式碼以讓有程式安全性的段落能夠被移出 synchronized 區段之外是很重要的。使用 volatile 變數又是另一個例子；如果有夠多的程式碼能被移出 synchronized 區段之外，也就不需要作同步化了。

這意味著在同步化與 volatile 變數之間有個平衡關係存在。依據程式的演算法來決定可以使用兩種技巧中的哪一種也不重要；事實上是可以將程式設計成兩種都用。當然，這個平衡是傾向一邊的：volatile 變數僅能安全的用在單一的載入或儲存操作。這個限制導致 volatile 變數的運用是不常見的。

J2SE 5.0 提供了一組 atomic class 來處理更複雜的情況。相對於只能夠作單一的 atomic 操作（像是載入或儲存），這些 class 能夠讓多個操作被 atomic 化的對待。這聽起來像是不太重要的加強版，但是一個簡單的 atomic 化「比對後設定（compare-and-set）」操作就能夠讓 thread 去“抓住旗標”。反過來說，這讓它能實作出 locking 的機制：事實上，ReentrantLock 只使用 atomic 的 class 就作出很多這樣的功能。理論上那是有可能不靠 Java 的同步化機制就實作出我們目前所辦到的一切。

在這一節中，我們會來檢視這些 atomic class。這些 atomic 的 class 有兩個用途。首先是比較簡單的，它提供 class 以對單一的資料片段執行 atomic 操作。舉例來說，有一個 volatile 的 integer 無法使用 ++ 運算子是因為 ++ 運算帶有多個指令。然而 AtomicInteger class 帶有一個可以讓它持有的 integer 能夠 atomic 化的被遞增（還是沒有用到同步化）。

其次，更複雜的，使用 atomic class 建構出完全不需要同步化的複雜程式碼。需要存取兩個或以上 atomic 變數的程式碼（或者是對單一的 atomic 變數執行兩個或以上的操作）通常都需要被 synchronized 過以便兩者的操作能夠被當作是一個 atomic 的單元。無論如何，藉著使用與 atomic class 所使用相同方式的程式技巧，你也可以避開同步化來設計演算法以執行這些多重的操作運算。

Atomic Class 的概觀

有四個基本的 atomic 型別是由 AtomicInteger、AtomicLong、AtomicBoolean、與 AtomicReference 這四個 class 來處理 integer、long、boolean、與物件。這些 class 都有提供兩個 constructor。預設的 constructor 以零值、false、false、或 null 來依資

料型別初始化物件。另一個 constructor 是以程式設計師所指定的值來初始與建構變數。set() 與 get() 提供 volatile 變數所具有的功能性：能夠 atomic 化的設定與取得值。get() 與 set() 也能夠確保讀寫是從主記憶體上執行的。

這些 class 的 getAndSet() 提供新的功能性。這個 method 能夠 atomic 化的在回傳原始值的時候設定變數成新的值，完全都不需要用到任何的同步化 lock。要知道只在 Java 層次使用 get 與 set 的運算子而沒有使用同步化是不可能模擬出這樣的 atomic 化功能性。如果這是不可能的，那它又是怎麼實作出來的？這個功能性是透過使用 user-level 的 Java 程式無法存取的原生 method 來達成的。你也可以自行撰寫原生的 method 來達成此功能，但是特定平台的問題是相當的令人生畏。此外，因為 atomic 的 class 是 Java 的核心 class，它們並不會有關於使用者自訂原生 method 的安全性問題。

compareAndSet() 與 weakCompareAndSet() 是有條件性的修改程序。這兩個 method 都要取用兩個參數－在 method 啟動時預期資料會具有的值，以及要把資料所設定成的值。method 只會在變數具有預期值的時候才會將它設定成新值。如果目前值不等於預期值，該變數不會被更動且 method 回傳 false。如果目前值等於預期值會回傳 boolean 的 true 值，在這種情況下，值會被設定成新值。這個 method 的 weak 型式基本上也是一樣，但是少了一項保證：如果 method 回傳的是 false 值，該變數不會被更動，但是這並不表示現有值不是預期值。這個 method 不管初始值是否為預期值都可能會無法更新該值。

AtomicInteger 與 AtomicLong 兩個 class 提供額外的 method 來支援 integer 與 long 資料型別。有意思的是這些便利性的 method 在內部是使用所提供的「比對後設定」功能性來實作的。無論如何，這些 method 是重要且常用的。

incrementAndGet()、decrementAndGet()、getAndIncrement()、與 getAndDecrement() 提供了前置遞增（pre-increment）、前置遞減、後遞增（post-increment）、與後遞減的功能性。它們之所以必須是因為 Java 的遞增與遞減運算子都是多重載入與儲存操作的語意縮寫（syntactic sugar）;這些操作在 volatile 變數上並不是 atomic 的。使用 atomic 的 class 能夠讓你 atomic 化的處理這些操作。

addAndGet() 與 getAndAdd() 提供「前置」與「後」的運算子給指定值（delta 值）的加法運算用。這些 method 讓程式能夠對變數增或減一個指定值－包括了負值，所以不需要一個相對的減法運算 method。

atomic package 有支援更複雜的變數型別嗎？是也不是。目前是沒有對 atomic 字元或浮點變數的實作。你可以使用 AtomicInteger 來處理字元，但是使用 atomic 的浮點數字需要 atomic 化帶有唯讀浮點數值的受管理物件。我們會在本章稍後檢視此一情形。

有一些支援 array 與變數的 class 已經包含在其他的物件中。然而，沒有額外的功能性是由這些 class 所提供的，所以對複雜型別的支援是很小的。對 array 來說，一次只有一個索引的變數可以異動；並沒有功能可以對整個 array 作 atomic 化的異動。atomic 的 array 是使用 AtomicIntegerArray、AtomicLongArray、與 AtomicReferenceArray 這些 class 來模型化。這些 class 的行為如同其所組成之資料型別的 array，但是 array 的大小必須要在建構時指定好，且在操作過程中必須提供索引。並沒有 class 實作 boolean 的 array。這只是小小的不方便，因為這樣的 array 可以用 AtomicIntegerArray 來模擬。

最後由兩個 class 來完成我們對 atomic class 的概述。AtomicMarkableReference 與 AtomicStampedReference 能夠讓 mark 或 stamp 跟在任何的物件參考上。更精確的說，AtomicMarkableReference 提供一種包括物件參考結合 boolean 的資料結構，而 AtomicStampedReference 提供一種包括物件參考結合 integer 的資料結構。

這些 class 的基本 method 在本質上是相同的，只有稍稍的修改過以容許兩個值（參考以及 stamp 或 mark）。get() 現在需要傳入一個 array 做為參數；stamp 或 mark 被儲存在 array 的第一個元素而參考被正常的傳回。其他的 get method 就是回傳參考、mark、或 stamp。set() 與 compareAndSet() 需要額外的參數來代表 mark 或 stamp。最後，這些 class 帶有 attemptMark() 或 attemptStamp() 用來依據期待的參考設定 mark 或 stamp。

使用 Atomic 的 Class

如同我們前面所提的，是有可能（理論上）將每個程式或 class 到目前為止已經實作出來的部分改成只用 atomic 變數。老實說，這沒有那麼簡單。atomic class 並不是同步化工具的直接替代品－使用它們會需要對程式作更複雜的重新設計，就算是在某些簡單的 class 中也是一樣。要能夠有更好的了解，讓我們將 ScoreLabel 這個 class 【註】修改成只使用 atomic 變數：

```
package javathreads.examples.ch05.example1;

import javax.swing.*;
import java.awt.event.*;
import java.util.concurrent.*;
```

註　ScoreLabel 同時也是我們第一個使用 J2SE 5.0 的 generic 功能的範例。你會看到在角括號中的 parameterized 程式碼；在這個 class 中 <CharacterSource> 是個 generic 的參考。更進一步的細節可以在 David Flanagan 與 Brett McLaughlin 合著的《Java 5.0 Tiger：程式高手秘笈》中看到（歐萊禮;中譯本已經發行）。

```
import java.util.concurrent.atomic.*;
import javathreads.examples.ch05.*;

public class ScoreLabel extends JLabel implements CharacterListener {
    private AtomicInteger score = new AtomicInteger(0);
    private AtomicInteger char2type = new AtomicInteger(-1);
    private AtomicReference<CharacterSource> generator = null;
    private AtomicReference<CharacterSource> typist = null;

    public ScoreLabel (CharacterSource generator, CharacterSource typist) {
        this.generator = new AtomicReference(generator);
        this.typist = new AtomicReference(typist);

        if (generator != null)
             generator.addCharacterListener(this);
        if (typist != null)
             typist.addCharacterListener(this);
    }

    public ScoreLabel () {
        this(null, null);
    }

    public void resetGenerator(CharacterSource newGenerator) {
        CharacterSource oldGenerator;

        if (newGenerator != null)
            newGenerator.addCharacterListener(this);

        oldGenerator = generator.getAndSet(newGenerator);
        if (oldGenerator != null)
            oldGenerator.removeCharacterListener(this);
    }

    public void resetTypist(CharacterSource newTypist) {
        CharacterSource oldTypist;

        if (newTypist != null)
            newTypist.addCharacterListener(this);

        oldTypist = typist.getAndSet(newTypist);
        if (oldTypist != null)
            oldTypist.removeCharacterListener(this);
    }
```

```
    public void resetScore() {
        score.set(0);
        char2type.set(-1);
        setScore();
    }

    private void setScore() {
        // 此 method 稍晚會在第七章解釋
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                setText(Integer.toString(score.get()));
            }
        });
    }

    public void newCharacter(CharacterEvent ce) {
        int oldChar2type;

        // 前一個字母沒有正確鍵入：扣一分
        if (ce.source == generator.get()) {
            oldChar2type = char2type.getAndSet(ce.character);

            if (oldChar2type != -1) {
                score.decrementAndGet();
                setScore();
            }
        }
        // 如果字母是無關的：扣一分
        // 如果字母不相符：扣一分
        else if (ce.source == typist.get()) {
            while (true) {
                oldChar2type = char2type.get();

                if (oldChar2type != ce.character) {
                    score.decrementAndGet();
                    break;
                } else if (char2type.compareAndSet(oldChar2type, -1)) {
                    score.incrementAndGet();
                    break;
                }
            }

            setScore();
        }
    }
}
```

當你將這個 class 與之前的作比較時，就會發現變動不只是將前面由同步化所保護的變數代換成 atomic 變數而已。移除掉同步化也對演算法產生不同方式的影響。我們在此作出了三種修改：簡單的變數代換、演算法的變更、與重新嚐試的操作。

每一種修改的要點都是在於保持 class 的 synchronized 版本語意的完整。synchronized 程式碼的語意依賴於實現程式碼所有的效應。確保變數是由程式碼以 atomic 化的更動運用是不夠的：還必須要確保程式碼的最終效應與 synchronized 版本是一致的。我們接下來會討論到不同種類的修改以觀察其中對上述需求所具有的意涵。

變數代換

你可能作出最簡單的修改是就將之前 synchronized 的 method 所用的變數代換成 atomic 變數。這就是在我們對 resetScroe() 中所作的事情：score 與 char2type 變數已經被改成 atomic 變數，且 method 只有把它們重新初始化而已。

很有意思的是，將此兩個變數一起作變更的動作並不是 atomic 化的完成：還是有可能會讓 char2type 變數的變更在完成前就變更了 score。這聽起來會是一個問題，但實際上不是，因為我們還是保持住這個 class 在 synchronized 版本上的語意。之前對 ScoreLabel 這個 class 的實作有著類似的 race condition，那有可能導致如果在 resetScore() 被呼叫時傾聽者還跟在資源上所會發生分數沒有更新的情況。

在之前的實作中，resetScore() 與 newCharacter() 都是 synchronized，但是這只意味著兩者不會同時的執行。被拖延住的 newCharacter() 呼叫還是可能因為到達的順序或者取得 lock 的順序而脫序執行（以 resetScore() 的觀點來說）。所以打字鍵入的事件可能會等到 resetScore() 完成後才會被傳遞，但屆時這傳遞到的只是個已經過時的事件。這與此次的實作有著相同的問題，就是在 resetScore() 中同時變更兩個變數這個動作並沒有 atomic 化的處理。

要記得同步化的目的不是要防止所有的 race condition；它要防的是有問題的 race condition。此次對 resetScore() 的實作所出現的 race condition 並不被認為是個問題。稍後在本章的任一例子中我們都會建構此打字遊戲的 atomic 化變更分數與字母版本。

變更演算法

第二種變更是發生在 resetGenerator() 與 resetTypist() 這兩個 method 新的實作中。之前對 resetGenerator() 與 resetTypist() 所作，將兩者的同步化 lock 分離的嚐試是個不錯的主意。這兩個都沒有變動到 score 或者 char2Type 變數。事實上，它們甚至也沒有變動到相互共用的變數－resetGenerator() 的同步化 lock 只是用來

保護此 method 不受多個 thread 同時的叫用。resetTypist() 也是這樣；事實上，這兩個 method 的問題都一樣，所以我們只討論 resetGenerator()。很不幸的，將 generator 變數變成 AtomicReference 會引發多個我們曾經解決過的潛在問題。

問題會發生的原因是因為由 resetGenerator() 所封裝的狀態不只是 generator 變數的值而已。讓 generator 變數變成 AtomicReference 表示我們知道對該變數的操作會是 atomic 化的發生。但當我們要從 resetGenerator() 中完全的移除掉同步化的時候，我們還必須要確保整個由此 method 所封裝住的狀態還是一致的。

在這個例子中，這個狀態包或了在字母來源產生器之上 ScoreLabel 物件的登記（this 物件）。在這個 method 完成後，我們要確保 this 物件只有登記一次到唯一一個產生器上（被指派到 generator 的 instance 變數上的那一個）。

思考一下當兩個 thread 同時呼叫 resetGenerator() 會發生什麼事情。在此討論中，現有的產生器是 generatorA；某一個 thread 以 generatorB 產生器呼叫 resetGenerator()；而另一個 thread 以稱為 generatorC 的產生器來呼叫此 method。

前一個範例看起來像是下面這樣：

```
if (generator != null)
    generator.removeCharacterListener(this);
generator = newGenerator;
if (newGenerator != null)
    newGenerator.addCharacterListener(this);
```

在這個程式碼中，兩個 thread 同時要求 generatorA 移除 this 物件：作用上它會被移除兩次。ScoreLabel 物件同樣也會加入 generatorB 與 generatorC。這兩個效應都是錯的。

因為前一個範例是 synchronized 過的，會防止這樣的錯誤發生。在我們沒有 synchronized 的程式碼中，則必須這樣做：

```
if (newGenerator != null)
    newGenerator.addCharacterListener(this);
oldGenerator = generator.getAndSet(newGenerator);
if (oldGenerator != null)
    oldGenerator.removeCharacterListener(this);
```

這個程式碼的效應必須要仔細考量。當它被兩個 thread 同時呼叫時，ScoreLabel 物件會被 generatorB 與 generatorC 登記。各 thread 之後會 atomic 化的設定目前的產生器。因為它們是同時的執行，可能會有不同的結果。假設第一個 thread 先執行；它會從 getAndSet() 取回 generatorA 然後將 ScoreLabel 物件從 generatorA 的傾聽

器中移除。第二個 thread 從 getAndSet() 取回 generatorB 並從 generatorB 的傾聽器移除 ScoreLabel。如果第二個 thread 先執行，變數會稍有不同，但結果永遠會是一樣的：不管哪一個物件被指派給 generator 的 instance 變數，它就是 ScoreLabel 物件所傾聽的那一個（唯一的一個）。

此處會有一個副作用影響到其他的 method。因為在交換之後傾聽器會從舊的資料來源中移除掉，且傾聽器會在交換前被加入到新的資料來源，它現在會有可能接收到既不是現有的產生器也不是打字鍵入來源所產出的字元。之前 newCharacter() 會檢查是否來源為產生器的來源，並在不是的時候會假設來源是打字鍵入來源。現在就不再是這樣了。newCharacter() 現在必須要在處理它之前確認字母的來源；它必須要忽略掉來自不正確的傾聽器的字母。

重新嚐試的操作

在此範例中newCharacter() 具有最多的改變。如同之前所提過的，第一個修改是將事件根據不同的字母來源給分離開來。這個 method 現在不能假設當來源不是產生器的時候就會是打字鍵入：它必須要丟棄任何不是來自於所屬來源的事件。

對產生器事件的處理只有小幅度的修改。首先，getAndSet() 是用來 atomic 化的用新值交換字母。其次，玩家直到交換之後才會被扣分。這是因為沒有辦法可以確保之前的字母是什麼，直到 getAndSet() 的交換完成後。此外，分數也必須要 atomic 化的遞減，因為它可能會被多個同時到達的事件給變更過。對於字母與分數的變更並不是被 atomic 化的處理掉：還是存有 race condition。然而，它也不會是問題。我們需要正確的更新分數以獎勵或處罰玩家。如果玩家在分數更新前遇到非常短暫的延遲並不會是個問題。

對打字鍵入事件的處理則更為複雜。我們需要檢查是否鍵入的字母是正確的。如果不是，玩家就會被處罰。這是藉由 atomic 化的遞減分數來達成的。如果字母被正確的鍵入，玩家無法被立即的給予獎賞。相對的，char2type 變數要先被更新。分數只有在 char2type 被正確更新時才會更新。如果更新的操作失敗了，這代表著在我們處理此事件時有其他的事件已經被處理過（於其他的 thread 中）－且其他的操作是成功處理完的。

其他的操作是成功的處理完另一個事件是什麼意思？它代表我們必須從頭再處理事件。我們是如此的作出假設：假設說正在使用的該變數值不會被變更且程式碼完成時也是如此的話，所有已經被我們設定具有特定值的變數就確實應該是那個值沒錯。但因為這與其他的 thread 衝突到，這些假設就被破壞了。藉由從頭重新嚐試處理事件，就會好像從未遇到衝突一樣。

那也就是為何這一段程式碼是包裝在一個無窮迴圈中的原因：程式不會離開迴圈直到事件被成功的處理掉。顯然在多個事件間有著 race condition；迴圈確保沒有一個事件會被漏掉或者處理超過一次以上。只要我們確實只處理有效的事件一次，事件被處理的順序就不重要了：在處理完每個事件後，資料就會保持在完好的狀態。注意到當我們使用同步化的時候，也是有同樣的情形：多個事件並沒有以指定的順序執行；它們只是以授予 lock 的順序來執行。

atomic 變數的目的只是在避免同步化的效能因素。然而，atomic 變數又是如何能夠在置於無窮迴圈中的時候還比較快呢？當然，答案是技術上來說那並不是個無窮的迴圈。額外的重複迴圈只會發生在 atomic 操作失敗的時候，這是因為與其他的 thread 發生衝突。要發生一個真正的無窮迴圈的話，那會需要無窮的衝突。如果使用同步化的話，這也會是個問題：無數的 thread 要存取 lock 同樣也會讓程式無法正常的操作。另外一方面，第十四章會討論到 atomic class 與同步化間的效能差異通常也沒有大到要特別注意。

如同我們在此範例中所可以發現到的，是有必要平衡同步化與 atomic 變數的使用。在使用同步化的時候，thread 在取得 lock 之前會被 block 住不能執行。這能夠讓程式因為其他的 thread 被阻擋不能執行而 atomic 化的來執行。當使用 atomic 變數時，thread 是能夠平行的執行相同的程式碼。atomic 變數的目的不是要消除不具有 thread 安全性的 race condition；它們的目的是要讓程式碼具有 thread 安全性，所以就不用特地去防止 race condition。

通告與 Atomic 變數

是否在需要條件變數的功能時也能夠使用 atomic 變數？使用 atomic 變數來實作條件變數的功能性是有可能的，但必不一定有效率。同步化－以及等待與通知機制－是以控制 thread 的狀態來實作出的。如果 thread 無法取得 lock 的話就會被 block 住不能執行，且會被設定為等待的狀態直到特定的條件發生為止。atomic 變數並不會 block 住 thread 的執行。事實上，由非 synchronized 的 thread 所執行的程式碼可能會被設定到更複雜的迴圈中以便能夠重新嚐試失敗的動作。換言之，是有可能以 atomic 變數來實作出條件變數的功能性，但 thread 會不停的在等待所需的條件同時來回運作著。

這並不代表如果需要條件變數的功能性時就應該要避免 atomic 變數。再一次，還是要取得平衡。是有可能在程式中不一定要運用通告與使用同步化的部份來使用 atomic 變數。也是有可能將整個程式實作以 atomic 變數以及用分離的函式庫來送出通告－在內部使用條件變數的函式庫。當然，於某些情況下讓 thread 在等待的時候來回執行也不會是個問題。

最後一個方案正是打字遊戲所處的情況。首先，只有兩個 thread－動畫元件的 thread 與字母產生器的 thread－需要等待條件。第二，等待的處理只會在遊戲停止的時候發生。程式已經在動畫的兩個 frame 之間等待；使用相同的迴圈與間隔來等待完間重新啟動遊戲並不會產生對效能的重大影響。第三，在 Start 按鈕按下時等待 100 個毫秒（動畫的 frame 間隔週期）應該不會讓玩家有感覺；任何注意到此延遲的玩家應該也同時會注意到動畫本身的延遲。

下面是只使用 atomic 變數來實作出的動畫元件；它會在使用者停止遊戲的時候來回的執行。對隨機字母產生器有類似的實作可以從線上範例取得。

```
package javathreads.examples.ch05.example2;

import java.awt.*;
import javax.swing.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import javathreads.examples.ch05.*;

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas
                    implements CharacterListener, Runnable {

    private AtomicBoolean done = new AtomicBoolean(true);
    private AtomicInteger curX = new AtomicInteger(0);
    private AtomicInteger tempChar = new AtomicInteger(0);
    private Thread timer = null;

    public AnimatedCharacterDisplayCanvas() {
        startAnimationThread();
    }

    public AnimatedCharacterDisplayCanvas(CharacterSource cs) {
        super(cs);
        startAnimationThread();
    }

    private void startAnimationThread() {
        if (timer == null) {
            timer = new Thread(this);
            timer.start();
        }
    }

    public void newCharacter(CharacterEvent ce) {
        curX.set(0);
        tempChar.set(ce.character);
```

```
            repaint();
        }

        protected void paintComponent(Graphics gc) {
            char[] localTmpChar = new char[1];
            localTmpChar[0] = (char) tempChar.get();
            int localCurX = curX.get();

            Dimension d = getSize();
            int charWidth = fm.charWidth(localTmpChar[0]);
            gc.clearRect(0, 0, d.width, d.height);
            if (localTmpChar[0] == 0)
                return;

            gc.drawChars(localTmpChar, 0, 1,
                         localCurX, fontHeight);
            curX.getAndIncrement();
        }

        public void run() {
            while (true) {
                try {
                    Thread.sleep(100);
                    if (!done.get()) {
                        repaint();
                    }
                } catch (InterruptedException ie) {
                    return;
                }
            }
        }

        public void setDone(boolean b) {
            done.set(b);
        }
    }
```

如同前一個範例，使用 atomic 變數不只是將由同步化保護的變數代換成 atomic 變數就好：演算法同樣也需要以能夠讓任何 race condition 具有 thread 安全性的形式來調整。在動畫元件中，建構動畫之 thread 的程式碼更是要這樣做。在前一個範例中，當 setDone() 被呼叫時就會建構這個 thread。我們可以不動該 method 的程式碼並使用 atomic 的參考變數來儲存此 thread 物件；只有成功儲存 atomic 參考的 thread 才會去呼叫新 thread 的啟動 method。然而，將此功能性以於該物件的 constructor 才會呼叫的 private method 中建構與啟動 thread 是比較容易的（因為 constructor 不會被多個 thread 所呼叫到）。

newCharacter() 只是部分的 atomic。個別變數的操作，像是 curX 與 tempChar 的指派，都因為使用到 atomic 變數而是 atomic 的。然而，這兩個指派動作一起來的時候就不是 atomic 的。如果其他的 thread 同時呼叫 newCharacter() 也不會是問題；兩個 method 的呼叫會設定 curX 變數為零，且字母變數也會被指派為由第二個 thread 所執行該 method 要求的字母。在此 method 與 paintComponent() 間還是有 race condition，但或許不太會被注意到。這個 race condition 會讓 paintComponent() 產生假的遞增。這意味著新的字母會從第二個動畫 frame 開始畫起－第一個動畫 frame 會被跳過－－個不太可能會被玩家注意到的效應。

paintComponent() 同樣也不是完全的 atomic，但也跟 newCharacter() 一樣，所有的 race condition 都是可接受的。它是不可能跟自己產生衝突的，因為 paintComponent() 只會被視窗系統所呼叫且只會來自單一的 thread。所以，沒有理由要保護只會被 paintComponent() 用到的變數。paintComponent() 會載入與 newCharacter() 共用的資料進暫用的變數。如果這些變數剛好在 paintComponent() 呼叫的時候被變更，這不會是個問題，因為另外一個 repaint() 的要求也會由 newCharacter() 所發出。這樣的結果最多也只不過是個假的 frame。

run() 這個 method 類似我們之前的版本，它會在 done 旗標是 false 的時候每隔 100 毫秒呼叫 repaint()。然而，如果 done 旗標被設定為 true，此 thread 還是會每隔 100 個毫秒醒來一次。這意味著這個程式每隔 100 毫秒執行一次“什麼都不作”的 task。這個 thread 在動畫執行的期間每隔 100 毫秒執行一次；就算是遊戲停止也還是這樣。在另外一方面，動畫的復原就不再是立即的執行：使用者最多會等到 100 毫秒才會看到動畫重新啟動。這可以藉由從 setDone() 呼叫 repaint() 來解決，但這個範例並不需要這麼做。動畫 frame 間的延遲是 100 毫秒。如果開始動畫的 100 毫秒延遲會被注意到，那麼 frame 間的 100 毫秒延遲也就同樣的會被注意到。

現在 setDone() 的實作簡單多了。它不再需要建構動畫的 thread，因為它現在是由元件的建構期間所完成的。且它也不需要通知動畫的 thread 關於 done 旗標被更動的事情。

這個實作主要的好處是在於此元件中不再有任何的同步化。在遊戲沒有進行的時候是會有些 threading 的消耗，但它還是比遊戲進行時要小。其他的程式可能會有不同的背景。如同我們之前所提過的，開發者不只面對使用同步化技巧或 atomic 變數的選擇而已；它們還需要在兩者之間取得平衡。為了要了解這個平衡，最好能多在實例中將兩者都使用到。

使用 Atomic 變數的摘要

這些範例展示出數種 atomic 變數的正規使用方式；我們也用了許多的技巧來延伸由 atomic 變數所提供的 atomic 操作。以下是這些技巧的摘要。

樂觀的同步化

在使用 atomic 變數的範例中所發生的事情是所謂的「沒有白吃的午餐」：此程式碼避開了同步化，但它在所執行的工作量上付出了代價。你可以把這想做是個“樂觀同步化”（此一詞修改自資料庫管理的術語）：抓住被保護變數的程式碼會作出此一瞬間沒有其他修改的假設。然後程式碼就計算出該變數新的值並嚐試更新該變數。如果有其他的 thread 同時修改了變數，這個更新就失敗且程式必須重新執行這些步驟（用變數最新修改過的值）。

這個 atomic 的 class 於實作內部使用這個技巧，且我們也在範例中對 atomic 變數有多個操作時使用這個技巧。

資料交換

資料交換是在取得舊值的同時 atomic 化的設定新值的能力。這是由使用 getAndSet() 來達成的。使用這個 method 能夠確保只有一個 thread 能夠取得並使用該值。

如果更複雜的資料交換時該如何？如果值的設定要依據舊值又該如何？這可以由把 get() 與 compareAndSet() 放在迴圈中來處理。get() 用來取得舊值，以計算新值。此變數以使用 compareAndSet() 來設定成新的值－只有在舊值沒有被更動的時候才會設定為新值。如果 compareAndSet() 失敗，這整個操作可以再重新試過，因為目前的 thread 在失敗時都沒有動到任何資料。雖然有叫用過 get()、計算過新值、與資料交換並不是個別的 atomic，如果它成功的話，這個順序可以被認為是 atomic 的，因為它只在沒有別的 thread 更動該值時才會成功。

比較與設定

比較與設定是只有在目前值是預期值的時候才 atomic 化設定值的能力。compareAndSet() 這個 method 就是用來處理這個狀況。這是在 atomic 層級提供條件式支援能力的重要 method。這個基本的功能甚至能夠用來實作出由 mutex 所提供的同步化能力。

如果更複雜的比較該如何？如果比較是要依據舊值或外來值該如何？跟前面一樣，這個案例可由把 get() 與 compareAndSet() 放在迴圈中來處理。get() 可以用來取得舊值，以用來比較或者只拿來達成 atomic 的交換。複雜的比較可以用來觀察是否要繼續操作。然後使用 compareAndSet() 來在目前值沒有被更動的情況下設定新值。如果操作失敗整個操作就會重新試過。同前，如果它成功的話，這個順序可以被認為是 atomic 的，因為它只在符合操作開始時的值才會被設定。

進階 atomic 資料型別

雖然 atomic class 可用的資料型別清單數量是相當的大，但不是完整的。atomic 的 package 並沒有支援字元與浮點數型別。雖然它有支援一般物件型別，但並沒有對更複雜的物件型別，像是 string 所需的操作。然而，我們可以將資料型別包裝進唯讀的資料物件來對任何的新型別實作以 atomic 的支援。然後此資料物件藉由改變 atomic 參考至新的資料物件，就可以被 atomic 化的作更動。這僅在埋入資料物件的值不會被任何方式變動才有效。任何對資料物件的異動必須僅能以改變參考到不同的物件上完成－舊物件的值是不改變的。所有由資料物件所封裝的值，不管是直接或非直接的，必須是唯讀的才能使這個技巧有效運作。

因此，不可能 atomic 化的改變浮點數值，但卻有可能 atomic 化的改變物件參考到不同的浮點數值。只要浮點數的值是唯讀的，這個技巧就會是具有 thread 安全性的。記住上面的方法，我們就可以實作出浮點數值的 atomic class：

```
package javathreads.examples.ch05;

import java.lang.*;
import java.util.concurrent.atomic.*;

public class AtomicDouble extends Number {
    private AtomicReference<Double> value;

    public AtomicDouble() {
        this(0.0);
    }

    public AtomicDouble(double initVal) {
        value = new AtomicReference<Double>(new Double(initVal));
    }

    public double get() {
        return value.get().doubleValue();
    }

    public void set(double newVal) {
        value.set(new Double(newVal));
    }

    public boolean compareAndSet(double expect, double update) {
        Double origVal, newVal;

        newVal = new Double(update);
        while (true) {
            origVal = value.get();
```

```
        if (Double.compare(origVal.doubleValue(), expect) == 0) {
            if (value.compareAndSet(origVal, newVal))
                return true;
        } else {
            return false;
        }
    }
}

public boolean weakCompareAndSet(double expect, double update) {
    return compareAndSet(expect, update);
}

public double getAndSet(double setVal) {
    Double origVal, newVal;

    newVal = new Double(setVal);
    while (true) {
        origVal = value.get();

        if (value.compareAndSet(origVal, newVal))
            return origVal.doubleValue();
    }
}

public double getAndAdd(double delta) {
    Double origVal, newVal;

    while (true) {
        origVal = value.get();
        newVal = new Double(origVal.doubleValue() + delta);
        if (value.compareAndSet(origVal, newVal))
            return origVal.doubleValue();
    }
}

public double addAndGet(double delta) {
    Double origVal, newVal;

    while (true) {
        origVal = value.get();
        newVal = new Double(origVal.doubleValue() + delta);
        if (value.compareAndSet(origVal, newVal))
            return newVal.doubleValue();
    }
}
```

```
    public double getAndIncrement() {
        return getAndAdd((double) 1.0);
    }

    public double getAndDecrement() {
        return getAndAdd((double) -1.0);
    }

    public double incrementAndGet() {
        return addAndGet((double) 1.0);
    }

    public double decrementAndGet() {
        return addAndGet((double) -1.0);
    }

    public double getAndMultiply(double multiple) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get();
            newVal = new Double(origVal.doubleValue() * multiple);
            if (value.compareAndSet(origVal, newVal))
                return origVal.doubleValue();
        }
    }

    public double multiplyAndGet(double multiple) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get();
            newVal = new Double(origVal.doubleValue() * multiple);
            if (value.compareAndSet(origVal, newVal))
                return newVal.doubleValue();
        }
    }
}
```

在新的 AtomicDouble 這個 class 中，我們使用了 atomic 參考物件來包裝一個 double 的浮點數值。因為 Double class 已經包裝一個 double 值，所以就不需要建構新的 class；此 Double 用來維護住該 double 值。

get() 現在必須使用兩個對 method 的呼叫來取得 double 值－它必須要取得 Double 物件，那是用來取得 double 浮點數值。取得 Double 物件型別顯然是 atomic 的，因為我們是使用 atomic 參考物件來持有該物件。無論如何，整體技巧能夠運作都有賴於資料是唯讀的：它不能被更改。如果資料不是唯讀的，讀出資料就不會是 atomic 的，且兩個併用的 method 也不會被視為是 atomic 的。

set() 是用來改變值的。因為被封裝的值是唯讀的，我們必須建構新的 Double 物件而不是改變舊值。如同 atomic 參考本身的操作一樣，這會是 atomic 的，因為我們是運用 atomic 參考物件來改變參考的值。

compareAndSet() 是以之前提過的複雜比對後設定技巧來實作出。getAndSet() 是以之前提過的複雜資料交換技巧實作出。而其他的method－相加、乘法等－也是以複雜的資料交換技巧來實作。我們沒有於本章直接的展示出這個 class 的範例，但會於第十五章中用到它。現在這個 class 是個對新的與複雜的資料型別實作出 atomic 支援相當好的 framework。

大量資料的異動

在之前的範例中，我們只有對個別的變數作 atomic 化的設定；還沒有作到對一群資料 atomic 化的設定。在那些對一個以上的變數作設定的例子中，我們並沒有考慮到要把它們當作一組來作 atomic 化的設定。無論如何，atomic 化的設定一組變數可以由建構包裝這些要被更動值的物件來達成；之後這些值就可以藉由 atomic 化的更動對這些值的 atomic 參考來作到同時的改變。這樣的運作方式完全就像 AtomicDouble 這個 class 一樣。

再一次的，這只有在值沒有以任何方式直接改變的情況下才會有效。任何對資料物件的變更是經由改變參考到不同的物件上來達成的－舊的物件值必須沒有被更動過。不管是直接或間接包裝的值都必須是唯讀的才能讓這個技巧有效運作。

以下是個用來保護兩個變數的 atomic class：分數與字母變數。我們可以使用這個 class 來開發 atomic 化修改分數與字母變數的打字遊戲：

```
package javathreads.examples.ch05.example3;

import java.util.concurrent.atomic.*;

public class AtomicScoreAndCharacter {
    public class ScoreAndCharacter {
        private int score, char2type;

        public ScoreAndCharacter(int score, int char2type) {
```

```
        this.score = score;
        this.char2type = char2type;
    }

    public int getScore() {
        return score;
    }

    public int getCharacter() {
        return char2type;
    }
}

private AtomicReference<ScoreAndCharacter> value;

public AtomicScoreAndCharacter() {
    this(0, -1);
}

public AtomicScoreAndCharacter(int initScore, int initChar) {
    value = new AtomicReference<ScoreAndCharacter>
                (new ScoreAndCharacter(initScore, initChar));
}

public int getScore() {
    return value.get().getScore();
}

public int getCharacter() {
    return value.get().getCharacter();
}

public void set(int newScore, int newChar) {
    value.set(new ScoreAndCharacter(newScore, newChar));
}

public void setScore(int newScore) {
    ScoreAndCharacter origVal, newVal;

    while (true) {
        origVal = value.get();
        newVal = new ScoreAndCharacter
                    (newScore, origVal.getCharacter());
        if (value.compareAndSet(origVal, newVal)) break;
    }
}
```

```
public void setCharacter(int newCharacter) {
    ScoreAndCharacter origVal, newVal;

    while (true) {
        origVal = value.get();
        newVal = new ScoreAndCharacter
                    (origVal.getScore(), newCharacter);
        if (value.compareAndSet(origVal, newVal)) break;
    }
}

public void setCharacterUpdateScore(int newCharacter) {
    ScoreAndCharacter origVal, newVal;
    int score;

    while (true) {
        origVal = value.get();
        score = origVal.getScore();
        score = (origVal.getCharacter() == -1) ? score : score-1;

        newVal = new ScoreAndCharacter (score, newCharacter);
        if (value.compareAndSet(origVal, newVal)) break;
    }
}

public boolean processCharacter(int typedChar) {
    ScoreAndCharacter origVal, newVal;
    int origScore, origCharacter;
    boolean retValue;

    while (true) {
        origVal = value.get();
        origScore = origVal.getScore();
        origCharacter = origVal.getCharacter();

        if (typedChar == origCharacter) {
            origCharacter = -1;
            origScore++;
            retValue = true;
        } else {
            origScore--;
            retValue = false;
        }

        newVal = new ScoreAndCharacter(origScore, origCharacter);
```

```
            if (value.compareAndSet(origVal, newVal)) break;
        }
        return retValue;
    }
}
```

如同在 AtomicDouble 這個 class 中一樣，getScore() 與 getCharacter() 能運作是因為被封裝的值是以唯讀的方式處理。set() 必須要建構新的物件來封裝要被儲存的新值。

setScore() 與 setCharacter() 是使用高等資料交換計數來實作出的。這是因為此實作技術上是交換資料而不只是設定資料而已。雖然只有改變一個部分的封裝資料，我們還是得讀取沒有要被改變的資料（用來確保它實際上沒有被變更）。且因為我們必須要對整組的資料作 atomic 化的修改－保證讓不想要修改的資料不會被更動到－所以必須要將程式碼實作成資料交換。

setCharacterUpdateScore() 與 processCharacter() 實作此計分系統的核心。第一個 method 設定要被鍵入的字母，並在玩家沒有正確的鍵入前一個字母時作出處罰。第二個 method 比對鍵入的字母與目前產生出的字母。如果兩者相符，此字母被設定成非字母的值，且分數會被遞增。如果兩者並不相符，就直接扣分。有意思的是雖然這兩個 method 是如此的複雜，但它們還是 atomic 的，因為所有的計算都是在暫存的變數上完成，且所有的值都是使用資料交換來作 atomic 化的異動。

以進階 atomic 資料型別來執行大量資料異動，可能會用到大量的物件。對每個交易動作都需要建構一個新的物件，而不管有多少的變數需要修改。每個 atomic 的比對後設定操作在失敗而必須重新嚐試的時候也需要建構新的物件。再說一次，atomic 變數的使用必須與使用同步化之間取得平衡。是否可以接受所有暫存物件的建構？此技巧是否有比同步化好？或者說這是個折衷方案？答案與該程式有關。

如同這些技巧所展示出來的，使用 atomic 變數是有點複雜。這複雜性發生在使用多個 atomic 變數時、對單一 atomic 變數的多重操作、或在一個必須要 atomic 的程式段中兩者兼備。在許多情況中，因為只是想要用 atomic 變數來作單一的操作，像是更新分數，它們是很容易使用的。

而在其他的情況下，使用這一類極簡同步化並不是個好主意。它可能會變得很複雜，使得程式碼難以維護或者讓開發者間很難接手。對同步化可能會是問題的大量 method 叫用，改以極簡同步化的好處還是可議的。對於那些在 class 或子系統中相信找到是由同步化所引發問題的讀者來說，重新審議過這個主題是個好主意－如果只是想要以極簡同步化來獲得較佳的滿意度的話。

Thread 區域變數

任何的 thread 都可以在任意的時間定義該 thread 私用的區域變數。其他的 thread 可以定義相同的變數以建構該變數自有的拷貝。這意味著 thread 的區域變數無法用在 thread 間分享狀態；對某 thread 私用的變數所作的改變並不會反映在其他 thread 所持有的拷貝上。但這也代表對該變數的存取絕不需要同步化，因為它是不可能讓多個 thread 同時存取。當然，thread 的區域變數還有其他的用途，但它們最常見的用途還是能夠讓多個 thread 保有自有的資料而不是對共用的資料來競爭同步化的 lock。

thread 的區域變數是以 java.lang.ThreadLocal 這個 class 來模型化：

```
public class ThreadLocal<T> {
    protected T initialValue();
    public T get();
    public void set(T value);
    public void remove();
}
```

在一般的使用中，會 subclass 此 ThreadLocal 並 override 過 initialValue() 來回傳應該在 thread 第一次存取此變數時回傳的值。它的 subclass 很少需要 override 過 ThreadLocal 其他的 method；相反的，這些 method 是用來做為特定 thread 值的 getter/setter 之 pattern。

有一個使用 thread 的區域變數以避免同步化的例子是個別 thread 使用的 cache。以下面的 class 來說：

```
package javathreads.examples.ch05.example4;

import java.util.*;

public abstract class Calculator {

    private static ThreadLocal<HashMap> results = new ThreadLocal<HashMap>()
{
        protected HashMap initialValue() {
            return new HashMap();
        }
    };

    public Object calculate(Object param) {
        HashMap hm = results.get();
        Object o = hm.get(param);
        if (o != null)
              return o;
        o = doLocalCalculate(param);
```

```
            hm.put(param, o);
            return o;
        }

        protected abstract Object doLocalCalculate(Object param);
    }
```

thread 區域物件被宣告成 static 因此物件本身（也就是此例中的 results 變數）是在 thread 間共用。當 thread 區域變數的 get() 被呼叫時，thread 的 local class 內部機制會回傳指派給特定 thread 的特定物件。該物件的初始值是從 extend 過 ThreadLocal 的 class 的 initialValue() 所回傳；當你在建構 thread 區域變數時，你要負責實作此 method 以回傳適當（屬於該 thread）的物件。

當範例中的 calculate() 被呼叫時，會參考 thread 的私用 hash map 以判斷值是否之前就被計算過了。如果是的話，該值就會被回傳；不然的話，會執行計算且新值會儲存於 hash map 中。因為對此 map 的存取只會來自一個 thread，我們就能夠使用 HashMap 物件而不是 Hashtable 物件（或者要同步化 has map）。

這個方式僅在計算因為取得 hash map 本身需要對所有的 thread 作同步化而非常昂貴的時候才有意義。如果從 thread 的 get() 所回傳的參考需要長時間持有的話，這種設計就值得多加探討，不然的話此參考就需要被同步化很長一段時間。不是如此，則你就只是以一個同步化的呼叫來取代另外一個罷了。一般來說，ThreadLocal 的效能在過去是相當的差，雖然這情況在 JDK 1.4 中已經得到改善，且在 J2SE 5.0 中改變更多。

另外一個此技巧也很有用的情況是處理不具 thread 安全性的 class。如果每個 thread 都初始在 thread 區域變數中必要的物件，它就有一份可以安全存取的拷貝。

可繼承的 Thread 區域變數

由 thread 中的 thread 區域變數所儲存的值之間是無關的。當一個新的 thread 被建立的時候，它會有一份新的 thread 區域變數拷貝，且這些變數的值是由 thread 私用 subclass 的 initialValue( ) 所傳回。

此想法的另一個變換是 InheritableThreadLocal 這個 class：

```
package java.lang;
public class InheritableThreadLocal extends ThreadLocal {
     protected Object childValue(Object parentValue);
}
```

這個 class 能夠讓子代 thread 繼承父代 thread 的 thread 區域變數值；也就是說，當 thread 區域變數的 get() 被子代 thread 所呼叫時，它所回傳的值會與該 method 被父代呼叫時所回傳的一樣。

如果想要的話，還可以使用 childValue() 來更進一步的將此行為參數化。當子代 thread 呼叫 thread 區域變數的 get() 時，get() 會查詢父代 thread 上的值。然後將該值傳遞給 childValue() 並回傳該值。預設上，childValue() 只是傳回它的參數，所以不會發生轉換。

摘要

在本章中，我們檢視過一些同步化的高等技巧。學習到關於 Java 記憶體模型以及為何它會妨礙某些同步化技巧如同預期般的運作。這會產生對 volatile 變數更好的認知以及了解到為何 Java 所推出的同步化規則是如此的難以改變。

我們同樣也檢視過伴隨 J2SE 5.0 而來的 atomic package。這是一種可以避開同步化的途徑，但為了它也是要付出代價的：在 atomic package 中的 class 有一種天性導致經常要為此而必須改變演算法（特別是在同時使用多個 atomic 變數時）。建構一個直到所需的結果達成時才會離開的迴圈是實作 atomic 變數時常見的方法。

範例的 Class

下面列出本章範例的 class 名稱與 Ant 的 target：

說明	主要的 Java class	Ant target
使用 atomic ScoreLabel 的 Swing Type Tester	javathreads.examples.ch05.example1.SwingTypeTester	ch5-ex1
使用 atomic animation canvas 的 Swing Type Tester	javathreads.examples.ch05.example2.SwingTypeTester	ch5-ex2
使用 atomic score 與 character class 的 Swing Type Tester	javathreads.examples.ch05.example3.SwingTypeTester	ch5-ex3
使用 thread 區域變數的計算測試	javathreads.examples.ch05.example4.CalculatorTest	ch5-ex4

計算器測試需要命令列參數來設定同時執行的 thread 數目，在 Ant 的指令中，它是以下面這個屬性來定義：

```
<property name="CalcThreadCount" value="10"/>
```