剖面導向程式設計(AOP/AOSD)簡介

Last revised: May 14, 2007

政大資科系 陳 恭 副教授

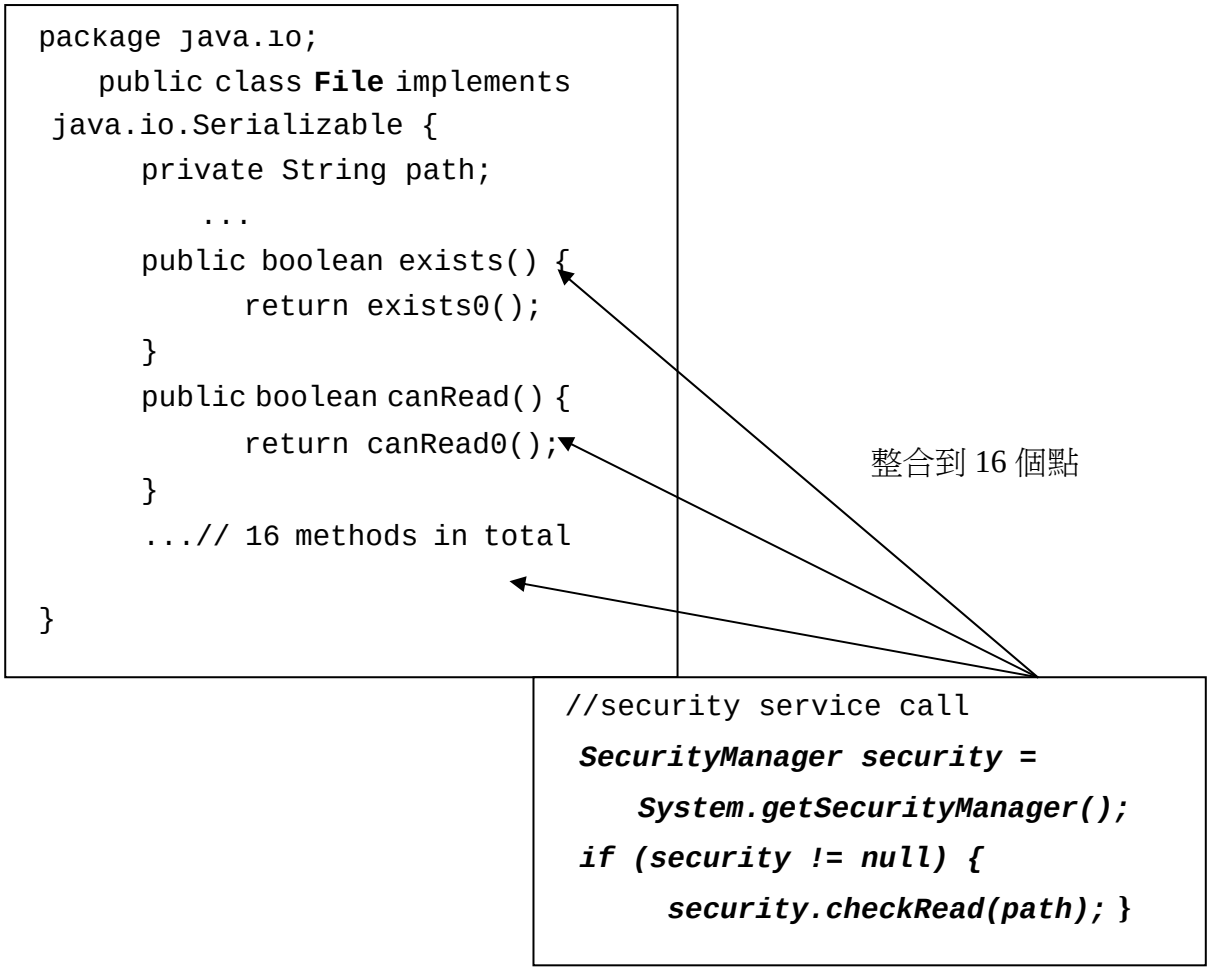
一般應用程式的需求，除了業務面的功能性需求外，也會有許多諸如授權檢查與記錄(log)等的非功能性需求。這類需求在實現上有一個特色，就是它們往往散布在應用程式的各個功能模組，不容易集中維護。軟體開發與設計的文獻中，稱這類需求為應用程式的橫跨性關注(Crosscutting concerns)[1]。現有的程式機制(programming mechanism)對於它們並無法提供有效的封裝與模組化支援，使得實現此類需求的程式碼必須重複出現於功能模組中，與其他功能性程式碼糾結在一起；即便是主流的物件導向程式設計在這方面也有所欠缺：雖然實現這類需求的程式碼可以集中開發維護，但是需要它們的各個地方還是要有引用的程式碼。例如，下面的程式碼摘自 Java 程式庫 java.io.File，我們看到完全相同的引用 Java 安全服務(JAAS SecurityManager)[2]的程式碼(斜體部分)重複達 16 次之多。

```
package java.io;
   public class File implements java.io.Serializable {
         private String path;
         ...
         public boolean exists() {
            // JAAS security service call
            SecurityManager security = System.getSecurityManager();
               if (security != null) {
                  security.checkRead(path);
               }
               return exists0();
           }
         public boolean canRead() {
             // JAAS security service call
            SecurityManager security = System.getSecurityManager();
               if (security != null) {
                  security.checkRead(path);
               }
               return canRead0();
           }
      … // repeat for 16 times
}
```

範例中斜體部分的程式碼如果需要修改，必須 16 個地方都要改；還要注意，修改時不會影響到其他檔案處理的部份，增加了程式維護的複雜度。最好能從 java.io.File

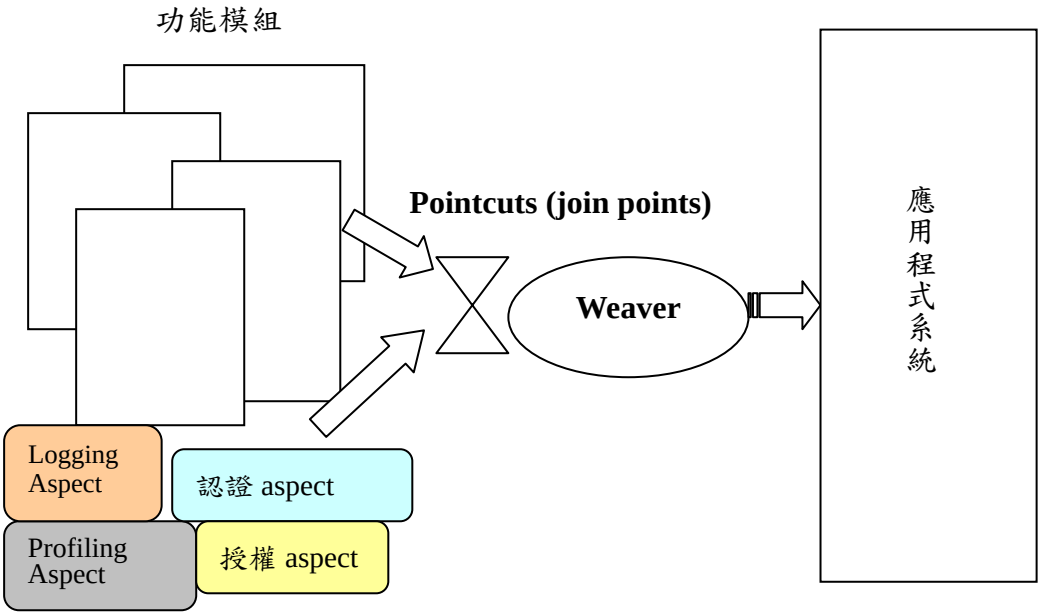
這個 class 中將這段安全檢查程式碼分離出來，成為單獨的一個模組，再用一種新的方式來整合這個模組與原來的 java.io.File class(將該模組放置到那 16 個點)。若能做到這樣的話，那不論是要修改這個模組本身，或是修改 java.io.File，都會比較單純，比較符合模組化程式設計的原則。

這個想法就是新一代的程式設計方法—*剖面導向程式設計*(Aspect-Oriented Programming, AOP) [3]，的基本出發點。我們用下列改自上面程式範例的示意圖來說明這個想法。原先重複的那段程式碼已移出 File class，集中在另一個模組，之後透過程式整合技術將這兩個模組結合，實現完整的功能。如果要修改安全檢查模組，只要修改此模組本身，不會有原來那樣寫法的問題發生。



簡言之，從 AOP 的觀點來看，應用程式除了功能模組外，還有許多像安全需求等的橫跨性關注；實現這些橫跨性關注的程式碼應該要從功能模組中分離出來，自成一模組，並稱之為**剖面**(aspect)。剖面與功能模組之間的介接點稱為連結點(join point)，通常一個剖面可以對應到多個連結點。一組相關的連結點與由所謂切點(pointcut)來定義，並透過稱之為織入(weaving)的機制將剖面程式碼整合入功能模組中，從而合成完整程式，滿足系統整體需求。這樣實現橫跨性關注的程式碼就可以集中封裝於適當的模組

中，避免掉程式碼糾結與重複的問題。下圖展示以 AOP 技術與工具組合應用系統各類程式模組的方式。



圖一：以 AOP 方式組合成應用系統的各類模組

AOP 這種想法自 1997[2]年提出來後，即受到許多軟體工程與程式語言的學者專家所重視，引起了非常多的迴響，很多的研究成果已陸續發表出來，Communication of ACM 在 2001 年 10 月份也出了 AOP 的專刊[3]。支援 AOP 的語言與工具也不少，像 AspectJ[4]、AspectC++[5]、Aspect C#[6]、JBoss AOP[7]與 AspectWerkz[8]。其中又以初期由 Xerox Palo Alto 實驗室發展出的基於 Java 的 AOP 程式語言—AspectJ 最為盛行，目前 AspectJ 已經進入 1.5 版(或稱 AspectJ 5)，由 Eclipse 組織與 IBM UK 負責研發。此外，近年來頗受實務界好評的 Spring framework[9]也以 runtime library 的方式支援巨觀層級(transaction and persistence)的 AOP。

以下我們以 AspectJ 為例來進一步明 AOP 的基本想法。AspectJ 程式分為 class 模組與 aspect 模組，Aspect 模組主要是定義 pointcuts 與 advice。Pointcutc 規範 aspect 與應用系統介接的 join points(簡單講，pointcut 為 join points 的集合)，advice 則定義在那些點要執行的程式碼，每個 advice 都需要一個 pointcut，而 advice 本身像是一個沒有名字的方法。二者共同界定在何時以及如何實現應用程式的橫跨性需求。Advice 分為 before、after、around 三種不同的執行模式，before 在 join point 之前、after 在 joint point 之後執行，around 比較特別，可以選擇取代 join point 的執行或是修改執行的結果。以下為一個簡單的 profiling aspect 範例，它會記錄名為 pkg.Test class 的所有方法被呼叫的次數。

```
public aspect CallCount {
  public int callCount; // 計數器
```

```
    pointcut methodCall(): call(* pkg.Test.*(..) );
    before(): methodCall() { //advice
       callCount++;
    }
  }
```

此範例顯示 pointcut 中可透過 wildcard 字元選取多個 join points，而且 pointcut 可以取名以便於重用。CallCount aspect 中的 before advice 會在程式中，每一個呼叫 pkg.Test class 的任一個方法的呼叫點執行，累加 callCount 的數值。

AspectJ 語言中的 aspects 在設計上沿用了許多 class 的功能，像是我們可以在 aspect 內宣告欄位變數(如 callCount)與一般方法。此外，我們也可以定義 abstract aspects 與 aspect inheritance，這主要是透過 abstract pointcuts 來達到的。以下是一個常見 aspect inheritance 的使用方式的範例：

```
public abstract aspect AccessControlCheck {
  abstract pointcut pc(Data d);
  abstract boolean constraint(Data d);
  void around(Data d) : pc(d) {
      if (constraint(d)) proceed(d);  // granted & resumed
      else forwardToExceptionHandler(..);//access denied
  }  // end around
   }
public aspect UpdateConstraint extends AccessControlCheck {
  pointcut pc(Data d):
      (execution(public void Customer.update*(..) || //union
       execution(public void Order.update*(..))) && args(d);
   boolean constraint(Data d) { //'d' got from pointcut
     ... /// evaluate the access constraint
  }
}
```

在這個範例中，AccessControlCheck aspect 中宣告了一個 abstract pointcut, 一個 abstract method 和一個 around advice。其中"Pointcut pc(Data d)"表示要從它選到的 join point 中取出型態為 Data 的物件，這通常是傳給這個 join point 的方法的引數(argument)。注意，before 與 after advice 都不會影響它們 pointcut 所選到的方法的執行，但 around advice 不同，around advice 基本上是會取代所選到的方法，不過 around advice 執行中，也可以透過呼叫了 proceed()來執行原來被擱置的選取的方法。所以在本例中，這個 around advice 會依據執行 constraint(d)的結果，決定是否要執行原來被擱置的選取的方法，進而達到存取控管的目的。不過這還需要另外一個 UpdateConstraint aspect 共同完成，它繼承 AccessControlCheck，並提供 poincut pc 和 Constraint 方法的定義。這裡，pc 指定的是 Customer 或 Order classes 中以 Update 開頭命名的方法，並且透過 args(d)將傳給這些方法的引數取出，傳給繼承來的 around

advice。

但 Aspects 也有一些與 class 不同之處，最主要的差別在於 aspect instantiation 之處。我們可以從 class"製造"物件，但是我們並不可以從 aspect 直接"製造"出 aspect instance。Aspect instance 的製造必須由 AspectJ runtime 負責執行。以上範例中宣告的 aspects 都是所謂 static aspect，只有一個 instance，由 AspectJ runtime 在應用程式載入時製造出來。我們也可以透過比較複雜的宣告 aspect 方式(例如 perthis)，要求 AspectJ runtime 從一個 aspect 製造出多個 aspect instances，這部份請參照 AspectJ 先關文件。

最後，AspectJ 除了以 advice 方式動態地去影響應用程式的執行過程，也可以用靜態的方式去改變應用程式中的 class 構造。譬如說，在一個既有的 class 中增加欄位或是方法，並進而改變這個 class 所實現的 interface。例如：

```
interface Iface { public int m(); }
aspect A {
  declare parents: Cls implement Iface;
  private int Cls.i =10; //在 class Cls 中增加一 field
  public void Cls.m() { //在 class Cls 中增加一 method
    return cls.i
  }
}
```

資訊補充站

目前多數的文獻與網站是以Aspect-Oriented Software Development (AOSD) 命名，各種相關訊息可以在 http://aosd.net 找到。每年 3 月舉辦的International Conference on Aspect-Oriented Software Development迄今已舉辦了 6 屆，論文集由ACM出版[10]。歐盟幾所大學並成立了AOSD-Europe進行各項研究計畫(http://aosd-europe.net/)。亞洲地區的一些從事AO先關研究的學者也在 2005 年建立社群與網站(http://www.cse.cuhk.edu.hk/~aoasia/)，開始進行對話；並透過舉辦Asian Workshop on AOSD來促進相關學者的交流。2005 AOAsia 1 在台北舉行，2006 AOAsia 2 在東京舉行，今年 2007 AOAsia 3 則將於 7 月 27 日在北京舉行。

參考文獻

[1] Walter Hürsch and Cristina Videira Lopes, *Separation of Concerns*, Technical Report, no. NU-CCS-95-03, 1995.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, in ECOOP '97 Object-Oriented Programming 11th European Conference," Finland (M. Aksit and S. Matsuoka, eds.), vol. 1241, pp. 220-242, New York, NY: Springer-Verlag, 1997.

[3] Communications of the ACM, *Special Issue on AOP*, vol. 44, no. 10, October 2001.

[4] AspectJ website: http://www.eclipse.org/aspectj/

[5] AspectC++ website: http://www.aspectc.org
[6] AspectC# website: www.dsg.cs.tcd.ie/index.php?category_id=169
[7] JBoss AOP website: http://www.jboss.org/products/aop
[8] AspectWerkz website: http://aspectwerkz.codehaus.org/index.html
[9] Spring Framework website: http://www.springframework.org
[10] ACM, *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007, ACM Press. ACM digital library-Proceedings.