第二章
深入類別載入器

如果陽春白雪真的是曲高和寡，
我希望有一天我能用下里巴人的方式
來引導他們學會陽春白雪。

前言

程式設計師在開發應用程式的時候，常常被老闆耳提面命，要求寫出來的程式要有彈性，容易擴充，甚至要求要能”Plug and Play”，相信不少程式設計師聽到這些要求就非常頭痛。在本文之中，筆者把這種需求稱做對於『動態性』的需求。有了動態性，我們的應用程式就可以在不用全盤重新編譯的情況下更新系統，或者在不用停止主程式運作的情況下(尤其是您的系統必須 24 小時運轉，一停止就會造成巨大損失時)，除去系統中原有的 bug，或者是增加原本不具備的新功能。

一般來說，常見的程式語言先天上並不具有動態性的本質，如 C、C++本身就不具備動態性。因此，為了讓這些本身不具有動態性的程式語言具有某種程度的動態性，就必須依賴底層的作業系統提供一些機制來實現動態性，Windows 作業系統底下的動態聯結函式庫(Dynamic Linking Library)和 Unix 底下的共享物件(Share Object)這是這樣的例子。但是，要運用這些底層作業系統所提供的機制，程式設計師必須多費一些功夫來撰寫額外的程式碼(例如 Windows 平台上需要使用 LoadLibrary()與 GetProcAddress()兩個 Win32 API 來完成動態性的需求)，這些額外撰寫的程式碼也會因為作業平台的不同而不同，畢竟這些額外的程式碼與程式本身的運作邏輯甚少關聯，所以維護起來雖不算複雜，但仍有其難度。

相對來說，Java 是一個本質上就具有『動態性』的程式語言。在一般使用 Java 程式語言來開發應用程式的工程師眼中，很少有機會能夠察覺 Java 因為具備了動態性之後所帶來的優點和特性，甚至根本不曾利用過這個 Java 先天就具有的特性。這不是我們的錯，而是因為這個動態的本質被巧妙地隱藏起來，使得使用 Java 的程式設計師在不知不覺中用到了動態性而不自知。

程式語言本質上就具備動態性的優點在於，程式設計師不需要撰寫額外的程式碼，所以比較沒有跨平台的問題。而缺點則是底層的黑箱幫我們搞定了一大堆的瑣碎工作，一旦程式設計師想要”御駕親征”，自己完全主控動態性的時候，就必須了解更多細部的運作機制。

本章的主角是類別載入器。在 Java 中，講到暗中幫助程式設計師，讓使用 Java 所撰寫的程式具備動態性的『黑手』，非類別載入器莫屬。筆者將帶大家深入類別載入器的運作機制，讓諸位讀者了解類別載入器如何達成動態性。了解

動態性之所以能夠順利運行的來龍去脈之後，我們還要看看如何運用動態性做出一些巧妙且實際的功能。

為何要自己全盤掌控動態性?

學習 Java 的朋友一定都知道，Java 是一種天生就具有動態連結能力的技術。Java 把每個類別的宣告、介面的宣告，在編譯器處理之後，全部變成一個個小的執行單位(類別檔, .class)，一旦我們指定一個具有 public static void main(String args[])方法的類別作為起點開始運作之後，Java 虛擬機器會找出所有在執行時期需要的執行單位，並將他們載入記憶體之中，彼此互相交互運作。儘管本質上是一堆類別檔，但是在記憶體之中，變成了一個"邏輯上"為一體的 Java 應用程式。所以，嚴格的來說，每個類別檔對 Java 虛擬機器來說，都是一個獨立的動態聯結函式庫，只不過它的副檔名不是 .dll 或 .so，而是 .class 罷了。 因為這種特性，所以我們可以在不重新編譯其他 Java 程式碼的情況下，只修改有問題的執行單位，並放入檔案系統之中，等到**下次該 Java 虛擬機器重新啟動時**，這個邏輯上的 Java 應用程式就會因為載入了新修改的 .class 檔，自己的功能也做了更新。這是一個最基本的動態性功能。

但是，如果您用過支援 JSP/Servlet 的高檔 Web Server(或稱 Web Container)，或是高檔的 Application Server 裡的 EJB Container，他們一定會提供一個名為 Hot Deployment 的功能，這個功能的意思是說，您可以在 Web Server 不關閉的情況下，放入已經編譯好的新 servlet 以取代舊的 servlet，下一次的 HTTP request 時，就會自動釋放舊的 servlet 所代表的類別檔，而重新載入新的 servlet 所代表的類別檔，同理，如果主角換成 EJB Container，Hot Deployment 可以讓元件部署者不用關閉 Application Server，就能夠將舊的 EJB(也是一堆類別檔的集合)換成新版的 EJB。

Hot Deployment 的功能並非每家廠商都能提供，只有還算高檔的產品才有。接下來眼尖的朋友就會問了:「慢著，前面說新的類別必須要等到"**下次該 Java 虛擬機器重新啟動時**"才會重新載入，可是有些 Web Server 或 EJB Container 本身就是用 Java 所撰寫，他們也是在 Java 虛擬機器上面執行，那麼，他們如何在 Java 虛擬機器不重新啟動的情況下具有 Hot Deployment 的功能? 好吧!就算可以做到讀取新版本的功能好了，在不關閉 Java 虛擬機器的情況下，類別所佔用的記憶體將無法釋放，那麼記憶體裡頭肯定充滿了同一個類別的新舊版本，如果 Hot Deployment 的次數太多，記憶體不會爆掉嗎?」。這是一個非常好的問題，也是本章之所以存在的理由。

了解了類別載入器的來龍去脈，您將可以讓您的程式具有強大的動態性 - 在 Java 虛擬機器不重新啟動的情況下做出具有載入最新類別檔的功能;不關閉 Java 虛擬機器的情況下，釋放類別所佔用的記憶體，讓記憶體不會因為充滿了同一個類別的多個版本而面臨記憶體不足的窘境。

我們在不知不覺中用到動態性

假設我們撰寫了三個 Java 類別，分別是 Main、A、以及 B。他們的程式碼分別如下所示:

檔案:A.java

```
public class A
{
	public void print()
	{
		System.out.println("Using Class A") ;
	}
}
```

檔案:B.java

```
public class B
{
	public void print()
	{
		System.out.println("Using Class B") ;
	}
}
```

檔案:Main.java

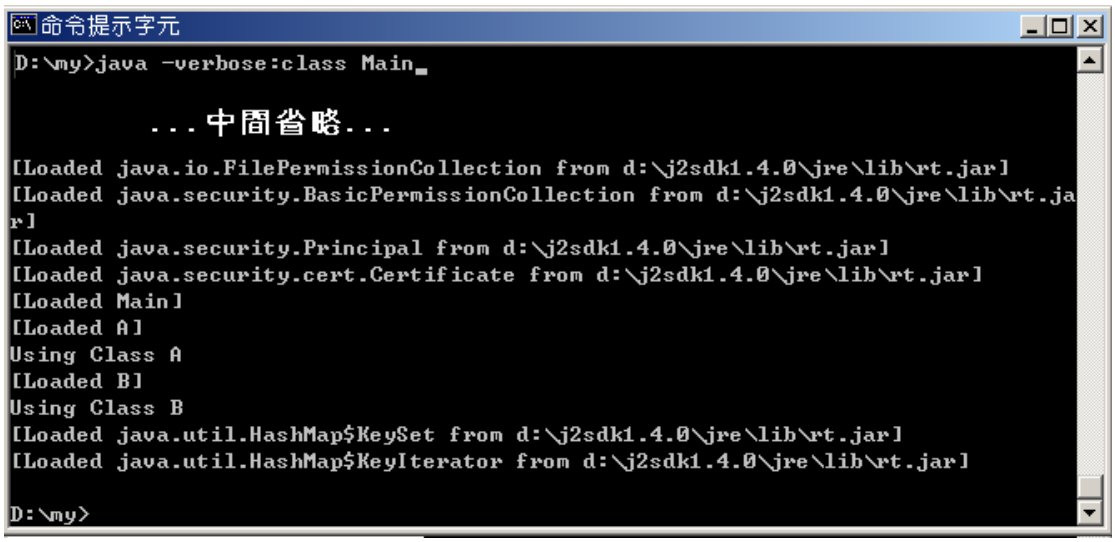
```
public class Main
{
	public static void main(String args[])
	{
		A a1 = new A() ;
		a1.print() ;
		B b1 = new B() ;
		b1.print() ;
	}
}
```

這三個檔案編譯好之後，所在目錄的內容如下圖所示:

接著，我們執行指令:

java **–verbose:class** Main

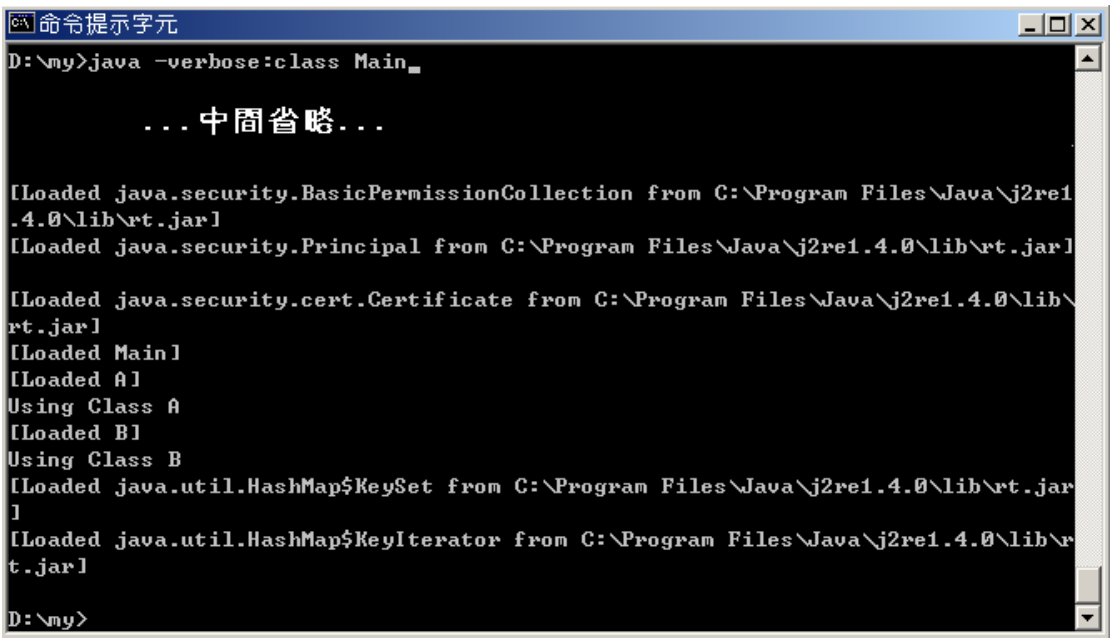
螢幕上會產生一長串的輸出結果，我們擷取最後的部分呈現出來，如下圖所示:



螢幕上的輸出告訴我們，類別載入器(class loader)在背景偷偷的運作，除了將 Java 程式運作時所需要的基礎類別函式庫(又叫核心類別函式庫， Core Classes)載入記憶體(位於**<JRE 所在位置>\lib\rt.jar** 之中)，我們的程式所用到的類別 Main.class、A.class、以及 B.class 也都偷偷被類別載入器載入了記憶體之中。類別載入器的功用，就是把類別從靜態的硬碟裡(.class 檔)，複製一份放到記憶體之中，並做一些初始化的工作，讓這個類別”活起來”，其他人就能夠使用它的功能。

有關於基礎類別函式庫，在第一章的時候曾經提到，java.exe 是利用幾個基本原則來尋找 Java Runtime Environment(JRE)，然後把類別檔(.class)直接轉交給 JRE 執行之後，java.exe 就功成身退。類別載入器也是構成 JRE 的其中

一個重要成員，所以最後類別載入器就會自動從所在之 JRE 目錄底下的 \lib\rt.jar 載入基礎類別函式庫。所以在上圖裡，一定是因為 java.exe 定位到 c:\j2sdk1.4.0\jre，所以才會有此輸出結果。因此，如果 java.exe 定位到其他的 JRE，輸出結果就會有稍許的不同，如下圖所示:



上圖就是 java.exe 定位到位於 C:\Program Files\Java\j2re1.4.0 這個 JRE 時，所輸出的結果，各位可以看到，類別載入器改由從 C:\Program Files\Java\j2re1.4.0\lib\rt.jar 之中載入。

預先載入與依需求載入

上述的螢幕輸出還透露了一個訊息，就是 - 我們自己所撰寫的類別(A.class 與 B.class)只會在"用到"的時候才載入(Main.class 是起始類別，所以一定比 A.class 和 B.class 優先載入)，而不是像基礎類別函式庫(位於 rt.jar 之中的類別)一次一股腦地全部載入記憶體之中，所以螢幕上才會輸出先載入了 A.class，然後印出"Using Class A"，再印出載入了 B.class 的訊息，然後再印出"Using Class B"。

像基礎類別函式庫這樣的載入方法我們叫做預先載入(pre-loading)，這是因為基礎類別函式庫裡頭的類別大多是 Java 程式執行時所必備的類別，所以為了不要老是做浪費時間的 I/O 動作(讀取檔案系統，然後將類別檔載入記憶體之中)，預先載入這些類別會讓 Java 應用程式在執行時速度稍微快一些。相對來說，我們自己所撰寫的類別之載入方式，叫做依需求載入(load-on-demand)，也就是 Java 程式真正用到該類別的時候，才真的把類別檔從檔案系統之中載入記憶體。

看到上述的說明，大家就會立刻有了結論:『只要參考到特定類別，類別載入器就會自動幫我們載入類別。』這個結論對嗎? 我們來試試修改後的

Main.java:

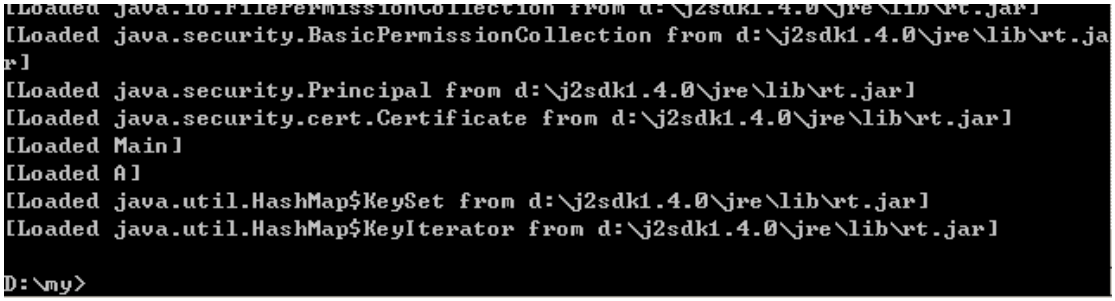
檔案:Main.java

```
public class Main
{
    public static void main(String args[])
    {
        A a1  = new A() ;

        B b1  ;

    }
}
```

執行

java –verbose:class Main

之後，螢幕上的輸出如下:



這個輸出告訴我們，只有單獨宣告(如: B 類別)而已，是不會促使類別載入器幫我們載入類別的，只有實體化指令(**new XXX()**)才會讓類別載入器幫我們載入該類別。

依需求載入的方式，可以讓執行時期所佔用的記憶體變小，這是因為我們的 Java 程式可能是由數以百計的類別所構成，但是不是每一種類別都會在執行的時候使用，因此依需求載入的方式，可以讓需要的類別載入記憶體，而不需要的類別不會被載入。這種機制這在記憶體不多的裝置上特別有用，因為 Java 最初就是為了嵌入式系統而設計，嵌入式裝置擁有的記憶體通常很小，所以 Java 採這種設計方式有其歷史因素。底下就是一個利用依需求載入功能來減少記憶體用量的例子:

檔案:Office.java

```
public class Office
{
    public static void main(String args[])
    {
        if(args[0].equals("Word"))
```

```
        {
            Word w = new Word() ;
            w.print() ;
        }else if(args[0].equals("Excel"))
        {
            Excel e = new Excel() ;
            e.print() ;
        }
    }
}
```

檔案:Word.java

```
public class Word
{
    public void print()
    {
        System.out.println("Using Word") ;
    }
}
```

檔案:Excel.java

```
public class Excel
{
    public void print()
    {
        System.out.println("Using Excel") ;
    }
}
```

假設您有一個主程式叫 Office，而 Office 又有兩個主要的應用程式 Word 及 Excel，兩個應用程式執行時都需要佔用很大的記憶體空間，而一般很少人同時用到這兩個應用程式，這時利用上述 Office.java 的程式寫法，就可以省去很多記憶體。當我們執行指令:

java Office Word

時，程式就會載入 Word.class，如下圖:

```
D:\my>java -verbose:class Office Word
```

```
[Loaded java.security.Principal from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.security.cert.Certificate from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded Office]
[Loaded Word]
Using Word
[Loaded java.util.HashMap$KeySet from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.util.HashMap$KeyIterator from d:\j2sdk1.4.0\jre\lib\rt.jar]

D:\my>
```

而執行指令:

java Office Excel

時，程式就會載入 Excel.class，如下圖:

```
D:\my>java -verbose:class Office Excel
```

```
[Loaded java.security.Principal from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.security.cert.Certificate from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded Office]
[Loaded Excel]
Using Excel
[Loaded java.util.HashMap$KeySet from d:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.util.HashMap$KeyIterator from d:\j2sdk1.4.0\jre\lib\rt.jar]

D:\my>
```

依需求載入的優點是節省記憶體，但是仍有其缺點。舉例來說，當程式第一次用到該類別的時候，系統就必須花一些額外的時間來載入該類別，使得整體執行效能受到影響，尤其是由數以萬計的類別所構成的 Java 程式。可是往後需要用到該類別時，由於類別在初次載入之後就會被永遠存放在記憶體之中，直到 Java 虛擬機器關閉，所以不再需要花費額外的時間來載入。

總的來說，就彈性上和速度上的考量，如此的設計所帶來的優點(彈性和省記憶體)遠超過額外載入時間的花費(只有第一次用到時)，因此依需求載入的設計是明智的選擇。

讓 Java 程式具有動態性的兩種方法

Java 本質上具有的動態性，帶來了記憶體的節省、程式的彈性、以及一些額外的載入時間。 既然談到了動態性，『那麼，上述的程式是"有彈性"的寫法嗎? 』一些寫程式的老手一定會這樣問。嚴格來說，上述的程式只達到了一點點的彈性，就是讓使用者可以在執行的時候選擇載入哪個類別。接下來，筆者將示範讓程式具有更多彈性的做法。

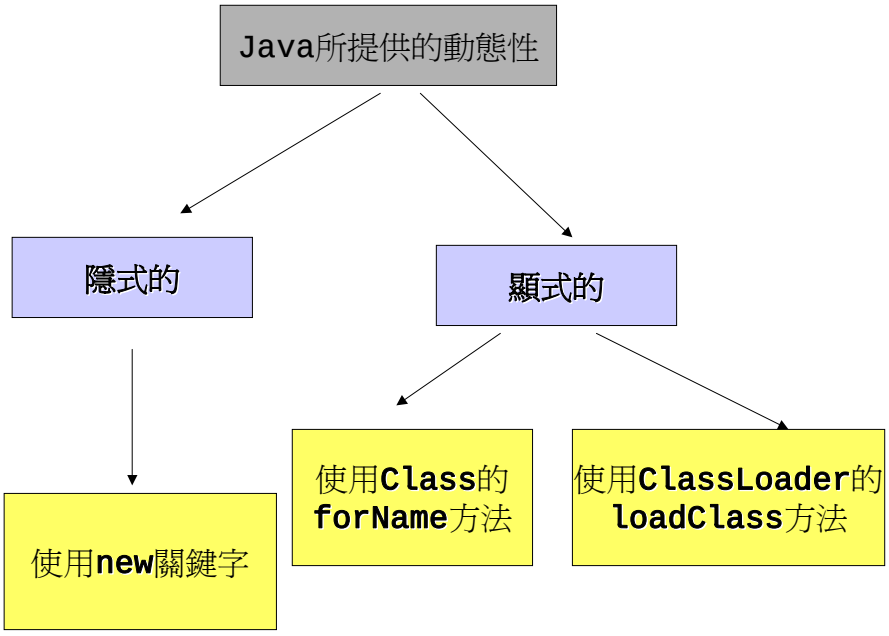
要讓程式具有彈性，就必須利用 Java 所提供的動態性來完成。Java 提供兩種方法來達成動態性。一種是隱式的(implicit)，另一種是顯式的(explicit)。這兩種方式底層用到的機制完全相同，差異只有在程式設計師所使用的程式碼有所不同，隱式的方法可以讓您在不知不覺的情況下就使用，而顯式的方法必須加入一些額外的程式碼。您可以把這兩種方法和 Win32 應用程式呼叫動態聯結函

式庫(Dynamic Linking Library)時的兩種方法(implicit 與 explicit)來類比，意思幾乎相同。

隱式的(implicit)方法我們已經談過了，也就是當程式設計師用到 new 這個 Java 關鍵字時，會讓類別載入器依需求載入您所需要的類別，這種方式使用了隱式的(implicit)方法，筆者在前面提到『**一般使用 Java 程式語言來開發應用程式的工程師眼中，很少有機會能夠察覺 Java 因為具備了動態性之後所帶來的優點和特性，甚至根本不曾利用過這個 Java 先天就具有的特性。這不是我們的錯，而是因為這個動態的本質被巧妙地隱藏起來，使得使用 Java 的程式設計師在不知不覺中用到了動態性而不自知**』，就是因為如此。但是，隱式的(implicit)方法仍有其限制，無法達成更多的彈性，遇到這種情況，我們就必須動用顯式的(explicit)方法來完成。

顯式的方法，又分成兩種方式，一種是藉由 java.lang.Class 裡的 forName()方法，另一種則是藉由 java.lang.ClassLoader 裡的 loadClass()方法。您可以任意選用其中一種方法。

Java提供的動態性



用顯式的方法來達成動態性:使用 Class.forName()方法

使用顯式的方法來達成動態性，意味著我們要自己動手處理類別載入時的細節部分。處理細節部分雖然需要撰寫一些額外的程式碼，但是可以讓程式變的更具彈性，我們以上面這個 Office.java、Word.java、以及 Excel.java 的例子來說，這個程式雖然很有彈性(可以讓執行程式的人在執行時期決定要載入哪個類

別)，但是，如果我們新增了 Access.java 和 PowerPoint.java 這兩個新類別時，Office.java 裡的主程式就必須增加兩個 if ... else 的迴圈。身為一個強調程式維護性的工程師，接下來要問的一定是:「那麼，有沒有更好的方法，可以在不修改主程式的情況下增加主程式的功能?」有的，使用顯式的方法所達成的動態性，可以增加程式的彈性，並達成我們不希望修改主程式的需求。程式碼如下所示:

檔案:Assemblyjava

```
public interface Assembly
{
	public void start() ;
}
```

檔案:Office.java

```
public class Office
{
	public static void main(String args[]) throws Exception
	{
		Class c = Class.forName(args[0]) ;
		Object o = c.newInstance() ;
		Assembly a = (Assembly) o ;
		a.start() ;
	}
}
```

檔案:Word.java

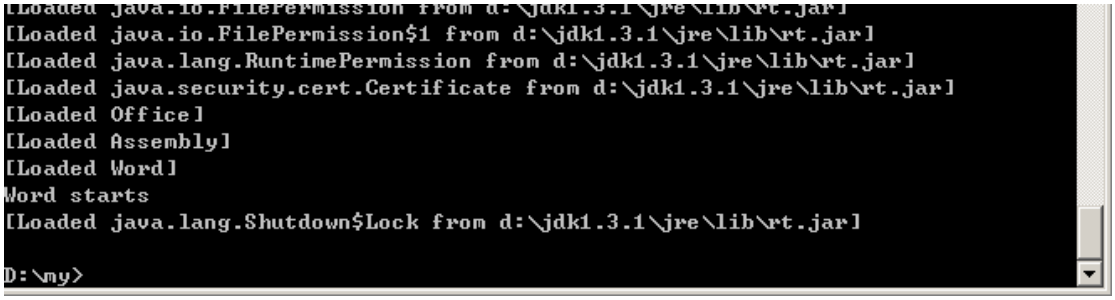
```
public class Word implements Assembly
{
	public void start()
	{
		System.out.println("Word starts") ;
	}
}
```

檔案:Excel.java

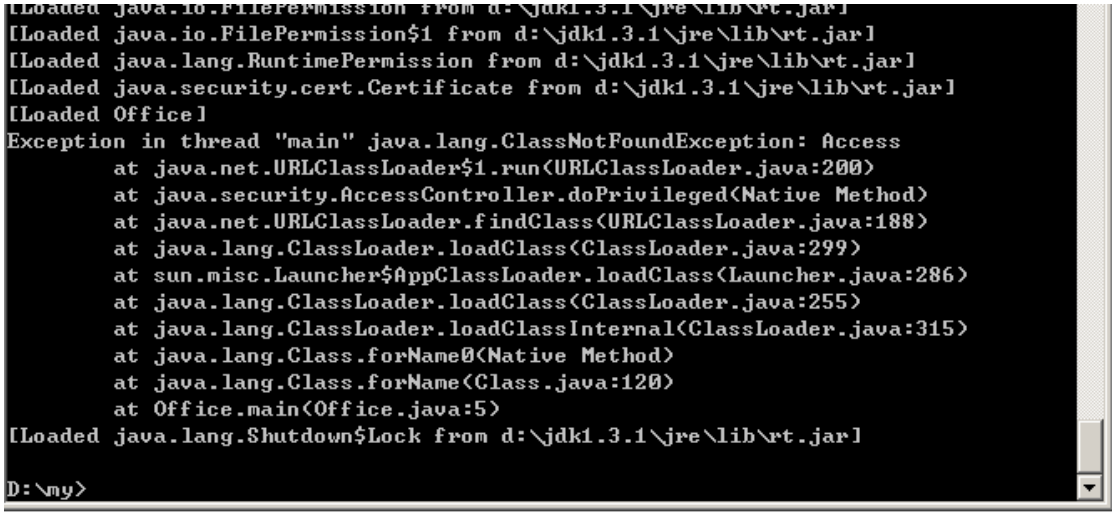
```
public class Excel implements Assembly
{
	public void start()
```

```
    {
        System.out.println("Excel starts") ;
    }
}
```

如此一來，我們的主程式 Office.java 只要編譯之後，往後只要叫用:
java Office Word 或 java Office Excel
就可以動態載入我們需要的類別，如下圖所示:



除此之外，輸入
java Office Access
的時候，雖然會出現錯誤訊息(因為我們還沒有完成 Access.java)，如下圖所示:



但是，一旦往後我們完成了 Access(Access.java)這個類別，或是 PowerPoint(PowerPoint.java)，只要他們都實作了 Assembly 這個介面，我們

就可以在不修改主程式(Office.java)的情況下，新增主程式的功能:

檔案:Access.java

```
public class Access implements Assembly
{
      public void start()
      {
            System.out.println("Access starts") ;
      }
}
```

檔案:PowerPoint.java

```
public class PowerPoint implements Assembly
{
      public void start()
      {
            System.out.println("PowerPoint starts") ;
      }
}
```

如果您用過 JDBC 撰寫資料庫程式，使用 Class.forName()來動態載入類別的功能，正是 JDBC 裡頭用來動態載入 JDBC 驅動程式(JDBC Software driver)的方式。

注意: 請仔細端看加入 –verbose:class 之後的螢幕輸出，您會看到 Assembly.class 也被系統載入了。在此您可以發現，interface 如同 class 一般，會由編譯器產生一個獨立的類別檔(.class)，當類別載入器載入類別時，如果發現該類別繼承了其他類別，或是實作了其他介面，就會先載入代表該介面的類別檔，也會載入其父類別的類別檔，如果父類別也有其父類別，也會一併優先載入。換句話說，類別載入器會依繼承體系最上層的類別往下依序載入，直到所有的祖先類別都載入了，才輪到自己載入。舉例來說，如果有個類別 C 繼承了類別 B、實作了介面 I，而 B 類別又繼承自 A 類別，那麼載入的順序如下圖:

```
[Loaded java.security.cert.Certificate from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded A]
[Loaded B]
[Loaded I]
[Loaded C]
```

如果您親自搜尋 Java 2 SDK 說明檔內部對於 Class 這個類別的說明，您可以發現其實有兩個 forName()方法，一個是只有一個參數的(就是之前程式之中所使用的):

public static Class forName(String className)

另外一個是需要三個參數的:

public static Class forName(String name, boolean initialize,
ClassLoader loader)

這兩個方法，最後都是連接到原生方法 forName0()，其宣告如下:

private static native Class forName0(String name, boolean initialize, ClassLoader loader) throws ClassNotFoundException;

只有一個參數的 forName()方法，最後叫用的是:

forName0(className, true, ClassLoader.getCallerClassLoader());

而具有三個參數的 forName()方法，最後叫用的是:

forName0(name, initialize, loader);

其中，關於名為 initialize 這個參數的用法，請看範例程式:

檔案:Office.java

```
public class Office
{
    public static void main(String args[]) throws Exception
    {
        Class c = Class.forName(args[0],true,null) ;
        Object o = c.newInstance() ;
        Assembly a = (Assembly) o ;
        a.start() ;
    }
}
```

執行時輸入

java Office Word

螢幕上的輸出為:

```
[Loaded java.io.FilePermission$1 from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded java.lang.RuntimePermission from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded java.security.cert.Certificate from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded Office]
Exception in thread "main" java.lang.ClassNotFoundException: Word
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:195)
        at Office.main(Office.java:5)
[Loaded java.lang.Shutdown$Lock from d:\jdk1.3.1\jre\lib\rt.jar]
```

之所以產生畫面上的錯誤訊息，是因為我們在類別載入器的部分給的參數是 null，導致系統不知道用哪個類別載入器來載入類別檔，因而發生錯誤。如果把程式修改如下:

檔案:Office.java

```
public class Office
{
```

```
    public static void main(String args[]) throws Exception
    {
        Office off = new Office() ;
        Class c = Class.forName(args[0],true,off.getClass().getClassLoader()) ;
        Object o = c.newInstance() ;
        Assembly a = (Assembly) o ;
        a.start() ;
    }
}
```

輸出就正常了:

```
[Loaded java.security.cert.Certificate from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded Office]
[Loaded Assembly]
[Loaded Word]
Word starts
[Loaded java.lang.Shutdown$Lock from d:\jdk1.3.1\jre\lib\rt.jar]
```

從這裡我們可以知道，forName()方法的第三個參數，是用來指定載入類別的類別載入器的，只有一個參數的 forName()方法，由於在內部使用了 ClassLoader.getCallerClassLoader()來取得載入呼叫他的類別所使用的類別載入器，和我們自己寫的程式有相同的效用。(**注意，ClassLoader.getCallerClassLoader()是一個 private 的方法，所以我們無法自行叫用，因此必須要自己產生一個 Office 類別的實體，再去取得載入 Office 類別時所使用的類別載入器**)。

那麼 forName()的第二個參數的效用為何? 我們修改主程式如下:

檔案:Office.java

```
public class Office
{
    public static void main(String args[]) throws Exception
    {
        Office off = new Office() ;
        System.out.println("類別準備載入") ;
        Class c = Class.forName(args[0],true,off.getClass().getClassLoader()) ;
        System.out.println("類別準備實體化") ;
        Object o = c.newInstance() ;
        Object o2 = c.newInstance() ;
    }
}
```

而 Word.java 之中則加入靜態初始化區塊:

檔案:Word.java

```
public class Word implements Assembly
{
	static
	{
		System.out.println("Word static initialization") ;
	}
	public void start()
	{
		System.out.println("Word starts") ;
	}
}
```

執行:

java Office Word

時的結果如下:

如果 forName()方法的第二個參數給的是 false:

檔案:Office.java

```
public class Office
{
	public static void main(String args[]) throws Exception
	{
		Office off = new Office() ;
		System.out.println("類別準備載入") ;
		Class c = Class.forName(args[0],false,off.getClass().getClassLoader()) ;
		System.out.println("類別準備實體化") ;
		Object o = c.newInstance() ;
		Object o2 = c.newInstance() ;
	}
}
```

則輸出變成:

```
[Loaded java.security.cert.Certificate from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded Office]
類別準備載入
[Loaded Assembly]
[Loaded Word]
類別準備實體化
Word static initialization
[Loaded java.lang.Shutdown$Lock from d:\jdk1.3.1\jre\lib\rt.jar]
```

使用 true 和 false 會造成不同的輸出結果，您看出端倪了嗎? 過去在很多 Java 的書本上提到靜態初始化區塊(static initialization block)時，都會說「**靜態初始化區塊是在類別第一次載入的時候才會被呼叫那僅僅一次。**」 可是從上面的輸出，卻發現即使類別被載入了，其靜態初始化區塊也沒有被呼叫，而是在第一次叫用 newInstance()方法時，靜態初始化區塊才真正被叫用。所以嚴格來說，應該改成「**靜態初始化區塊是在類別第一次被實體化的時候才會被呼叫那僅僅一次。**」

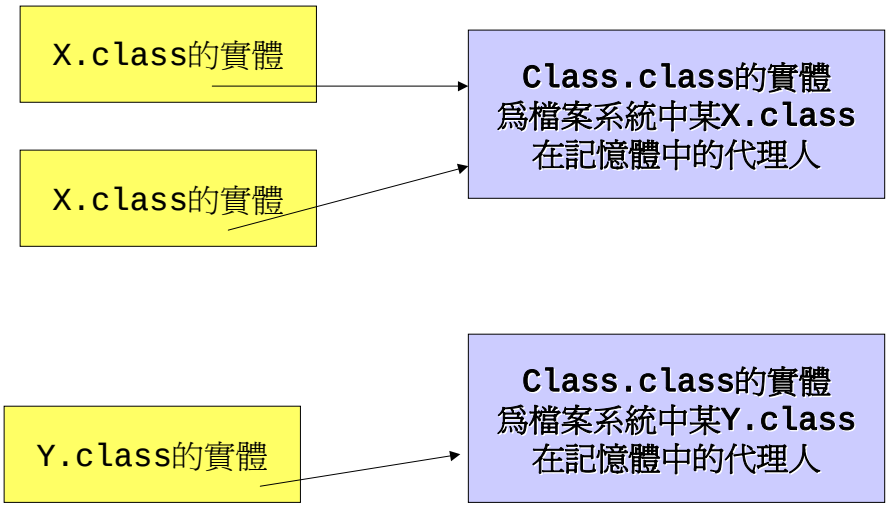
所以，我們得到以下結論:不管您使用的是 new 來產生某類別的實體、或是使用只有一個參數的 forName()方法，內部都隱含了"載入類別+呼叫靜態初始化區塊"的動作。而使用具有三個參數的 forName()方法時，如果第二個參數給定的是 false，那麼就只會命令類別載入器載入該類別，但不會叫用其靜態初始化區塊，只有等到整個程式第一次實體化某個類別時，靜態初始化區塊才會被叫用。

用顯式的方法來達成動態性:直接使用類別載入器

要直接使用類別載入器幫我們載入類別，首先必須先取得指向類別載入器的參考才行，而要取得指向類別載入器的參考之前，必須先取得某物件所屬的類別。

相信在大家的認知裡，都有個基本的概念，就是:「類別是一個樣版，而物件就是根據這個樣版所產生出來的實體」。在 Java 之中，每個類別最後的老祖宗都是 Object，而 Object 裡有一個名為 getClass()的方法，就是用來取得某特定實體所屬類別的參考，這個參考，指向的是一個名為 Class 類別 (Class.class) 的實體，您無法自行產生一個 Class 類別的實體，因為它的建構式被宣告成 private，這個 Class 類別的實體是在類別檔(.class)第一次載入記憶體時就建立的，往後您在程式中產生任何該類別的實體，這些實體的內部都會有一個欄位記錄著這個 Class 類別的所在位置。概念上如下圖所示:

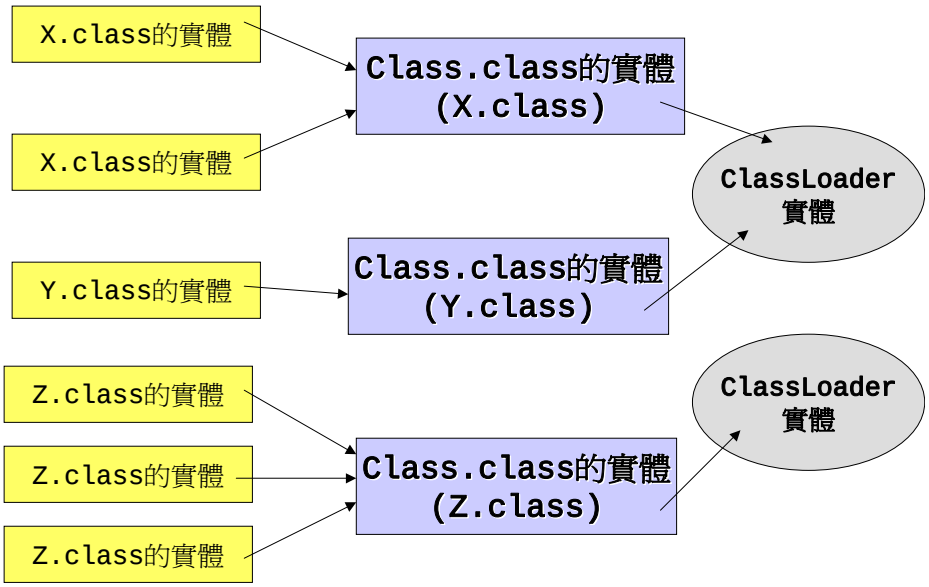
類別實體與Class.class



基本上，我們可以把每個 Class 類別的實體，當作是某個類別在記憶體中的代理人。每次我們需要查詢該類別的資料(如其中的 field、method 等)時，就可以請這個實體幫我們代勞。事實上，Java 的 Reflection 機制，就大量地利用 Class 類別。去深入 Class 類別的原始碼，我們可以發現 Class 類別的定義中大多數的方法都是原生方法(native method)。

在 Java 之中，每個類別都是由某個類別載入器(ClassLoader 的實體)來載入，因此，Class 類別的實體中，都會有欄位記錄著載入它的 ClassLoader 的實體(注意:如果該欄位是 null，並不代表它不是由類別載入器所載入，而是代表這個類別由靴帶式載入器(bootstrap loader,也有人稱 root loader)所載入，只不過因為這個載入器並不是用 Java 所寫成，所以邏輯上沒有實體)。其概念如下圖所示:

類別實體,Class.class,與ClassLoader



從上圖我們可以得之，系統裡同時存在多個 ClassLoader 的實體，而且一個類別載入器不限於只能載入一個類別，類別載入器可以載入多個類別。所以，只要取得 Class 類別實體的參考，就可以利用其 getClassLoader()方法籃取得載入該類別之類別載入器的參考。getClassLoader()方法最後會呼叫原生方法 getClassLoader0()，其宣告如下:

```
private native ClassLoader getClassLoader0();
```

最後，取得了 ClassLoader 的實體，我們就可以叫用其 loadClass()方法幫我們載入我們想要的類別。因此，我們把程式碼修改如下:

檔案:Office.java

```
public class Office
{
	public static void main(String args[]) throws Exception
	{
		Office off = new Office() ;
		System.out.println("類別準備載入") ;
		ClassLoader loader = off.getClass().getClassLoader() ;
		Class c = loader.loadClass(args[0]) ;
			System.out.println("類別準備實體化") ;
		Object o = c.newInstance() ;
		Object o2 = c.newInstance() ;
	}
}
```

執行

java Office Word
之後，結果如下:

```
[Loaded java.security.cert.Certificate from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded Office]
類別準備載入
[Loaded Assembly]
[Loaded Word]
類別準備實體化
Word static initialization
[Loaded java.lang.Shutdown$Lock from d:\jdk1.3.1\jre\lib\rt.jar]
```

從這裡我們也可以看出，直接使用 ClassLoader 類別的 loadClass()方法來載入類別，只會把類別載入記憶體，並不會叫用該類別的靜態初始化區塊，而必須等到第一次實體化該類別時，該類別的靜態初始化區塊才會被叫用。這種情形與使用 Class 類別的 forName()方法時，第二個參數傳入 false 幾乎是相同的結果。

上述的程式其實還有另外一種寫法如下所示:

檔案:Office.java

```
public class Office
{
    public static void main(String args[]) throws Exception
    {
        Class cb = Office.class ;
        System.out.println("類別準備載入") ;
        ClassLoader loader = cb.getClassLoader() ;
        Class c = loader.loadClass(args[0]) ;
        System.out.println("類別準備實體化") ;
        Object o = c.newInstance() ;
        Object o2 = c.newInstance() ;
    }
}
```

直接在程式裡頭使用 **Office.class**，就是直接取得某特定實體所屬類別的參考，用起來比起產生 Office 的實體，再用 getClass()取出，這個方法方便的多，也較省記憶體。

自己建立類別載入器來載入類別

在此之前，當我們談到使用類別載入器來載入類別時，都是使用既有的類別載入器來幫我們載入我們所指定的類別。那麼，我們可以自己產生類別載入器來幫我們載入類別嗎？答案是肯定的。利用 Java 本身提供的 java.net.URLClassLoader 類別就可以做到，範例如下:

檔案:Office.java

```
import java.net.* ;
```

```
public class Office
{
	public static void main(String args[]) throws Exception
	{
		URL u = new URL("file:/d:/my/lib/") ;
		URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
		Class c = ucl.loadClass(args[0]) ;
		Assembly asm = (Assembly) c.newInstance() ;
		asm.start() ;
	}
}
```

在這個範例中，我們自己產生 java.net.URLClassLoader 的實體來幫我們載入我們所需要的類別。但是載入前，我們必須告訴 URLClassLoader 去哪個地方尋找我們所指定的類別才行，所以我們必須給它一個 URL 類別所構成的陣列，代表我們希望它去搜尋的所有位置。URL 可以指向網際網路上的任何位置，也可以指向我們電腦裡的檔案系統(包含 JAR 檔)。在上述範例中，我們希望 URLClassLoader 到 d:\my\lib\這個目錄下去尋找我們需要的類別，所以指定的 URL 為"file:/d:/my/lib/"。其實，如果我們請求的位置是主要類別(有 public static void main(String arga[])方法的那個類別)的相對目錄，我們可以在 URL 的地方只寫"file:lib/"，代表相對於目前的目錄。

我們再把程式修改如下:

檔案:Office.java

```
import java.net.* ;
public class Office
{
	public static void main(String args[]) throws Exception
	{
		URL u = new URL("file:/d:/my/lib/") ;
		URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
		Class c = ucl.loadClass(args[0]) ;
		Assembly asm = (Assembly) c.newInstance() ;
		asm.start() ;

		URL u1 = new URL("file:/d:/my/lib/") ;
		URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
		Class c1 = ucl1.loadClass(args[0]) ;
		Assembly asm1 = (Assembly) c1.newInstance() ;
		asm1.start() ;
```

```
    }
}
```

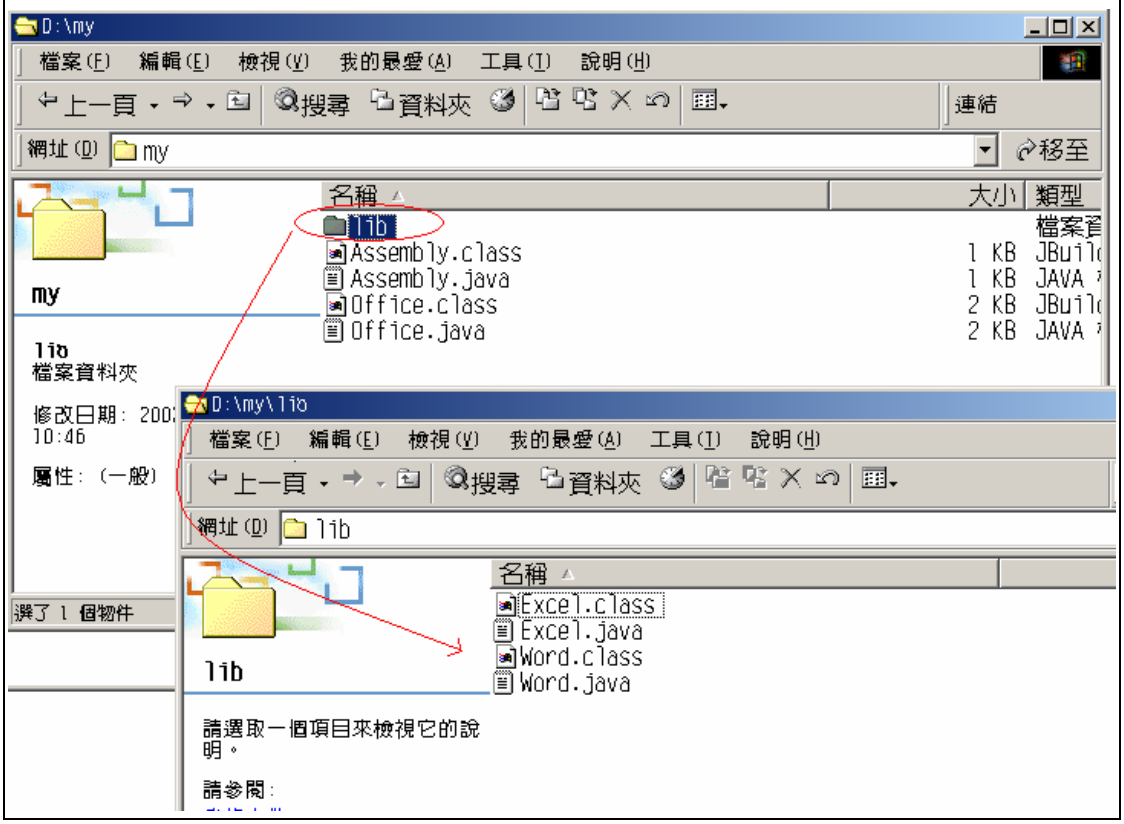
您將發現螢幕輸出如下:

```
D:\my>java Office Word
Word static initialization
Word starts
Word static initialization
Word starts

D:\my>
```

您可以發現，同樣一個類別，卻被不同的 URLClassLoader 分別載入，而且分別初始化一次。也就是說，在一個虛擬機器之中，相同的類別被載入了兩次。

注意!

此範例中，我們的目錄結構如下圖:



類別被哪個類別載入器載入?

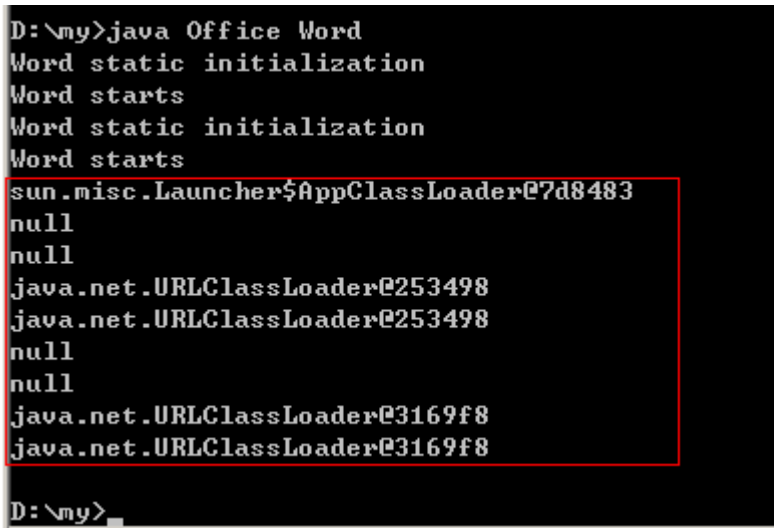
我們將上述的程式碼稍作修改，修改後的程式碼如下:

檔案:Office.java

```
import java.net.* ;
public class Office
{
    public static void main(String args[]) throws Exception
```

```
    {
        URL u = new URL("file:/d:/my/lib/") ;
        URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
        Class c = ucl.loadClass(args[0]) ;
        Assembly asm = (Assembly) c.newInstance() ;
        asm.start() ;

        URL u1 = new URL("file:/d:/my/lib/") ;
        URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
        Class c1 = ucl1.loadClass(args[0]) ;
        Assembly asm1 = (Assembly) c1.newInstance() ;
        asm1.start() ;

        System.out.println(Office.class.getClassLoader()) ;
        System.out.println(u.getClass().getClassLoader()) ;
        System.out.println(ucl.getClass().getClassLoader()) ;
        System.out.println(c.getClassLoader()) ;
        System.out.println(asm.getClass().getClassLoader()) ;

        System.out.println(u1.getClass().getClassLoader()) ;
        System.out.println(ucl1.getClass().getClassLoader()) ;
        System.out.println(c1.getClassLoader()) ;
        System.out.println(asm1.getClass().getClassLoader()) ;
    }
}
```

執行後輸出結果如下圖:



從輸出中我們可以得知，Office.class 由 AppClassLoader(又稱做 System

Loader，系統載入器)所載入，URL.class 與 URLClassLoader.class 由 Bootstrap Loader 所載入(注意:輸出 null 並非代表不是由類別載入器所載入。在 Java 之中，所有的類別都必須由類別載入器載入才行，只不過 Bootstrap Loader 並非由 Java 所撰寫而成，而是由 C++實作而成，因此以 Java 的觀點來看，邏輯上並沒有 Bootstrap Loader 的類別實體)。而 Word.class 分別由兩個不同的 URLClassLoader 實體載入。至於 Assembly.class，本身應該是由 AppClassLoader 載入，但是由於多型(Polymorphism)的關係，所指向的類別實體(Word.class)由特定的載入器所載入，導致列印在螢幕上的內容是其所參考的類別實體之類別載入器。Interface 這種型態本身無法直接使用 new 來產生實體，所以在執行 getClassLoader()的時候，叫用的一定是所參考的類別實體的 getClassLoader()，要知道 Interface 本身由哪個類別載入器載入，您必須使用底下程式碼:

Assembly.class.getClassLoader()

如果把這行程式加在上述程式之中，您會發現其輸出結果為

```
sun.misc.Launcher$AppClassLoader@7d8483
```

這個輸出告訴您，Assembly.class 是由 AppClassLoader 載入。

一切都是由 Bootstrap Loader 開始：類別載入器的階層體系

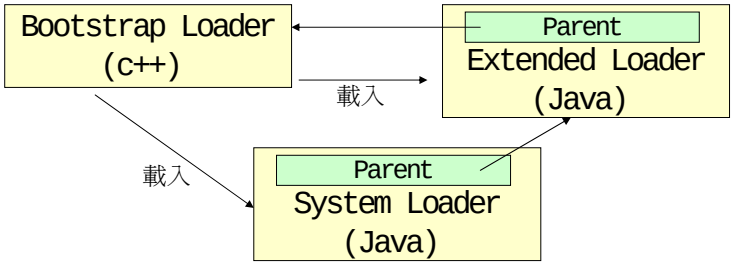
注意!
如果您參考其他資料，您將發現在講解類別載入器時，資料中常常會提到「類別載入器的階層體系」(classloader hierarchy)。請注意，這裡所說的繼承體系，並非我們一般所指的類別階層體系(即父類別與子類別構成的體系)。「類別載入器的階層體系」的意涵另有所指，請不要混淆了。筆者接下來將為您解釋「類別載入器的階層體系」所代表的真正意涵。

在前面我們曾經提過，Java 程式在編譯之後會產生許多的執行單位(.class 檔)，當我們執行主要類別時(有 public static void main(String arga[])方法的那個類別)，才由虛擬機器一一載入所有需要的執行單位，變成一個邏輯上為一體的 Java 應用程式。因此接下來，我們將細部討論這整個流程。

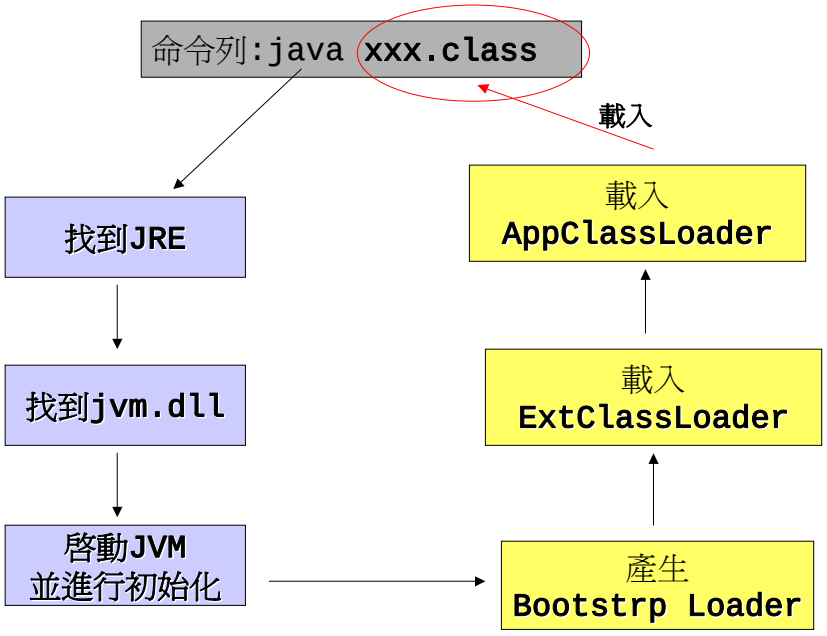
當我們在命令列輸入 java xxx.class 的時候，java.exe 根據我們之前所提過的邏輯找到了 JRE(Java Runtime Environment)，接著找到位在 JRE 之中的 jvm.dll(真正的 Java 虛擬機器)，最後載入這個動態聯結函式庫，啟動 Java 虛擬機器。這個動作的詳細介紹請回頭參閱第一章。

虛擬機器一啟動，會先做一些初始化的動作，比方說抓取系統參數等。一旦初始化動作完成之後，就會產生第一個類別載入器，即所謂的 Bootstrap Loader，Bootstrap Loader 是由 C++所撰寫而成(所以前面我們說，以 Java

的觀點來看，邏輯上並不存在 Bootstrap Loader 的類別實體，所以在 Java 程式碼裡試圖印出其內容的時候，我們會看到的輸出為 null)，這個 Bootstrap Loader 所做的初始工作中，除了也做一些基本的初始化動作之外，最重要的就是載入定義在 sun.misc 命名空間底下的 Launcher.java 之中的 ExtClassLoader(因為是 inner class ，所以編譯之後會變成 Launcher$ExtClassLoader.class)，並設定其 Parent 為 null，代表其父載入器為 Bootstrap Loader。然後 Bootstrap Loader 再**要求載入**定義於 sun.misc 命名空間底下的 Launcher.java 之中的 AppClassLoader(因為是 inner class，所以編譯之後會變成 Launcher$AppClassLoader.class)，並設定其 Parent 為之前產生的 ExtClassLoader 實體。這裡要請大家注意的是，**Launcher$ExtClassLoader.class 與 Launcher$AppClassLoader.class 都是由 Bootstrap Loader 所載入，所以 Parent 和由哪個類別載入器載入沒有關係。**我們可以用下圖來表示:



AppClassLoader 在 Sun 官方文件中常常又被稱做系統載入器(System Loader)，但是在本文中為了避免混淆，所以還是稱作 AppClassLoader。最後一個步驟，是由 AppClassLoader 負責載入我們在命令列之中所輸入的 xxx.class(注意:實際上 xxx.class 很可能由 ExtClassLoader 或 Bootstrap Loader 載入，請參考底下「**委派模型**」一節)，然後開始一個 Java 應用程式的生命週期。上述整個流程如下圖所示:



這個由 Bootstrap Loader ➔ ExtClassLoader ➔ AppClassLoader，就是我們所謂「類別載入器的階層體系」。我們可以用底下的程式碼證明這一點:

檔案:test.java

```
public class test
{
	public static void main(String args[])
	{
		ClassLoader cl = test.class.getClassLoader() ;
		System.out.println(cl) ;
		ClassLoader cl1 = cl.getParent() ;
		System.out.println(cl1) ;
		ClassLoader cl2 = cl1.getParent() ;
		System.out.println(cl2) ;
	}
}
```

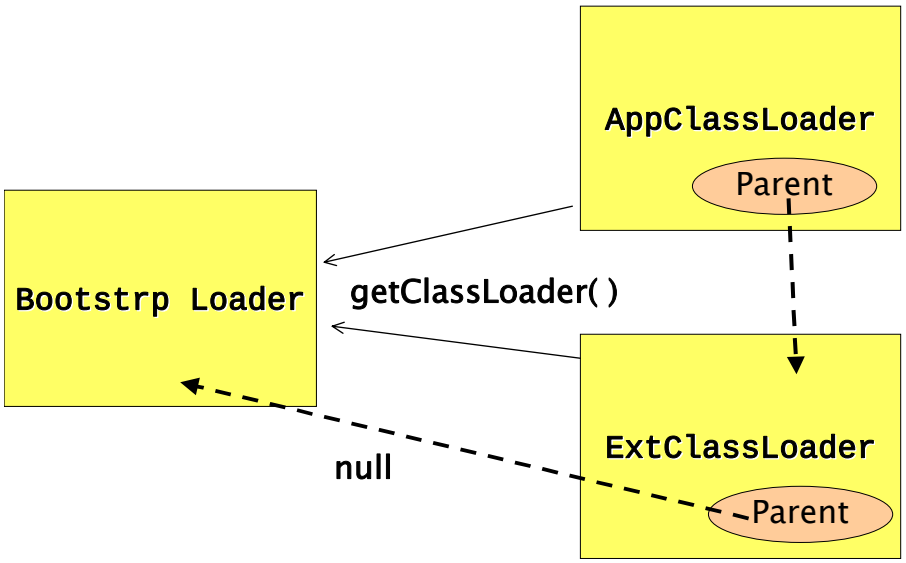
輸出結果如下:

```
D:\my>java test
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$ExtClassLoader@ac738
null

D:\my>
```

如果在上述程式中，如果您使用程式碼:

cl.getClass().getClassLoader()及 **cl1.getClass().getClassLoader()**，您會發

現印出的都是 null，這代表它們都是由 Bootstrap Loader 所載入。這裡也再次強調，**類別載入器由誰載入(這句話有點詭異，類別載入器也要由類別載入器載入，這是因為除了 Bootstrap Loader 之外，其餘的類別載入器皆是由 Java 撰寫而成)，和它的 Parent 是誰沒有關係，Parent 的存在只是為了某些特殊目的，這個目的我們將在稍後作解釋。**這三個主要載入器的關係如下圖所示:



在此要請大家注意的是，AppClassLoader 和 ExtClassLoader 都是 URLClassLoader 的子類別。由於它們都是 URLClassLoader 的子類別，所以它們也應該有 URL 作為搜尋類別檔的參考，由原始碼中我們可以得知，AppClassLoader 所參考的 URL 是從系統參數 java.class.path 取出的字串所決定，而 java.class.path 則是由我們在執行 java.exe 時，利用 –cp 或 -classpath 或 CLASSPATH 環境變數所決定。我們可以用底下程式碼測試之:

檔案:test.java

```
public class test
{
	public static void main(String args[])
	{
		String s = System.getProperty("java.class.path");
		System.out.println(s) ;
	}
}
```

輸出結果如下:

```
D:\my>java test
.

D:\my>java -classpath e:\test;d:\my;. test
e:\test;d:\my;.

D:\my>set CLASSPATH=f:\;.

D:\my>java test
f:\;.

D:\my>java -classpath e:\test;d:\my;. test
e:\test;d:\my;.

D:\my>
```

從這個輸出結果，我們可以看出，在預設情況下，AppClassLoader 的搜尋路徑為”.”(目前所在目錄)，如果使用-classpath 選項(與-cp 等效)，就可以改變 AppClassLoader 的搜尋路徑，如果沒有指定-classpath 選項，就會搜尋環境變數 CLASSPATH。如果同時有 CLASSPATH 的環境設定與-classpath 選項，則以-classpath 選項的內容為主，CLASSPATH 的環境設定與-classpath 選項兩者的內容不會有加成的效果。

至於 ExtClassLoader 也有相同的情形，不過其搜尋路徑是參考系統參數 java.ext.dirs。我們可以用底下程式碼測試:

檔案:test.java

```
public class test
{
	public static void main(String args[])
	{
		String s = System.getProperty("java.ext.dirs");
		System.out.println(s) ;
	}
}
```

輸出結果如下:

```
D:\my>path=c:\winnt\system32\

D:\my>java test
C:\Program Files\Java\j2re1.4.0\lib\ext

D:\my>path=d:\jdk1.3.1\bin

D:\my>java test
d:\jdk1.3.1\jre\lib\ext

D:\my>path=d:\j2sdk1.4.0\bin

D:\my>java test
d:\j2sdk1.4.0\jre\lib\ext

D:\my>_
```

輸出結果告訴我們，系統參數 java.ext.dirs 的內容，會指向 java.exe 所選擇的 JRE 所在位置下的\lib\ext 子目錄。系統參數 java.ext.dirs 的內容可以在一開始下命列的時候來更改，如下:

```
D:\my>java test
d:\jdk1.3.1\jre\lib\ext

D:\my>java -Djava.ext.dirs=c:\winnt\ test
c:\winnt\

D:\my>_
```

最後一個類別載入器是 Bootstrap Loader，我們可以經由查詢由系統參數 sun.boot.class.path 得知 Bootstrap Loader 用來搜尋類別的路徑。請使用底下的程式碼測試之:

檔案:test.java

```
public class test
{
	public static void main(String args[])
	{
		String s = System.getProperty("sun.boot.class.path");
		System.out.println(s) ;
	}
}
```

輸出結果如下:

```
D:\my>java test
d:\jdk1.3.1\jre\lib\rt.jar;d:\jdk1.3.1\jre\lib\i18n.jar;d:\jdk1.3.1\jre\lib\sunr
sasign.jar;d:\jdk1.3.1\jre\classes

D:\my>
```

系統參數 sun.boot.class.path 的內容可以在一開始下命列的時候來更改，如下:

```
D:\my>java test
d:\jdk1.3.1\jre\lib\rt.jar;d:\jdk1.3.1\jre\lib\i18n.jar;d:\jdk1.3.1\jre\lib\sunr
sasign.jar;d:\jdk1.3.1\jre\classes

D:\my>java -Dsun.boot.class.path=c:\winnt\ test
c:\winnt\

D:\my>
```

從這三個類別載入器的搜尋路徑所參考的系統參數的名字中，其實還透漏了一個訊息。請回頭看到 java.class.**path** 與 sun.boot.class.**path**，也就是說，AppClassLoader 與 Bootstrap Loader 會搜尋它們所指定的位置(或 JAR 檔)，如果找不到就找不到了，AppClassLoader 與 Bootstrap Loader 不會遞迴式地搜尋這些位置下的其他路徑或其他沒有被指定的 JAR 檔。反觀 ExtClassLoader，所參考的系統參數是 java.ext.**dirs**，意思是說，他會搜尋底下的所有 JAR 檔以及 classes 目錄，作為其搜尋路徑(**所以您會發現上面我們在測試的時候，如果加入 -Dsun.boot.class.path=c:\winnt 選項時，程式的起始速度會慢了些，這是因為 c:\winnt 目錄下的檔案很多，必須花額外的時間來列舉 JAR 檔**)。

在命令列下參數時，使用 –classpath / -cp / 環境變數 CLASSPATH 來更改 AppClassLoader 的搜尋路徑，或者用 –Djava.ext.dirs 來改變 ExtClassLoader 的搜尋目錄，兩者都是有意義的。可是用–Dsun.boot.class.path 來改變 Bootstrap Loader 的搜尋路徑是無效。這是**因為 AppClassLoader 與 ExtClassLoader 都是各自參考這兩個系統參數的內容而建立，當您在命令列下變更這兩個系統參數之後， AppClassLoader 與 ExtClassLoader 在建立實體的時候會參考這兩個系統參數，因而改變了它們搜尋類別檔的路徑;而系統參數 sun.boot.class.path 則是預設與 Bootstrap Loader 的搜尋路徑相同，就算您更改該系統參與，與 Bootstrap Loader 完全無關**。如果您手邊有原始碼，以 JDK 1.4.x 為例，請參考**<原始碼根目錄>\hotspot\src\share\vm\runtime\os.cpp** 這個檔案裡頭的 **os::set_boot_path** 方法，您將看到程式碼片段:

```
static const char classpathFormat[] =
    "%/lib/rt.jar:"
    "%/lib/i18n.jar:"
    "%/lib/sunrsasign.jar:"
    "%/lib/jsse.jar:"
    "%/lib/jce.jar:"
    "%/lib/charsets.jar:"
    "%/classes";
```

JDK 1.4.x 比 JDK 1.3.x 新增了一些核心類別函式庫(例如 jsse.jar 與 jce.jar)，所以如果您測試時用的是 1.4.x 版的 JDK，sun.boot.class.path 內容應該如下:

```
D:\my>path d:\j2sdk1.4.0\bin

D:\my>java test
d:\j2sdk1.4.0\jre\lib\rt.jar;d:\j2sdk1.4.0\jre\lib\i18n.jar;d:\j2sdk1.4.0\jre\li
b\sunrsasign.jar;d:\j2sdk1.4.0\jre\lib\jsse.jar;d:\j2sdk1.4.0\jre\lib\jce.jar;d:
\j2sdk1.4.0\jre\lib\charsets.jar;d:\j2sdk1.4.0\jre\classes

D:\my>_
```

更重要的是，AppClassLoader 與 ExtClassLoader 在整個虛擬機器之中只會存有一份，一旦建立了，其內部所參考的搜尋路徑將不再改變，也就是說，即使我們在程式裡利用 System.setProperty()來改變系統參數的內容，仍然無法更動 AppClassLoader 與 ExtClassLoader 的搜尋路徑。因此，**執行時期動態更改搜尋路徑的設定是不可能的事情。如果因為特殊需求，有些類別的所在路徑並非在一開始時就能決定，那麼除了產生新的類別載入器來輔助我們載入所需的類別之外，沒有其他方法了。**

以下範例說明即使更改系統參數 java.class.path，仍然無法變更 AppClassLoader 載入類別時所使用的搜尋路徑。

檔案:test.java

```
public class test
{
	public static void main(String args[])
	{
		String old = System.getProperty("java.class.path");
		System.out.println(old) ;

		System.setProperty("java.class.path","d:\\test");
		String s = System.getProperty("java.class.path");
		System.out.println(s) ;

		testlib tl = new testlib() ;
		tl.print() ;
	}
}
```

檔案:testlib.java

```
public class testlib
{
	public void print()
	{
		System.out.println("I get loaded") ;
	}
```

```
}
```

(注意: test.java 與 testlib.java 放在同一個目錄下)

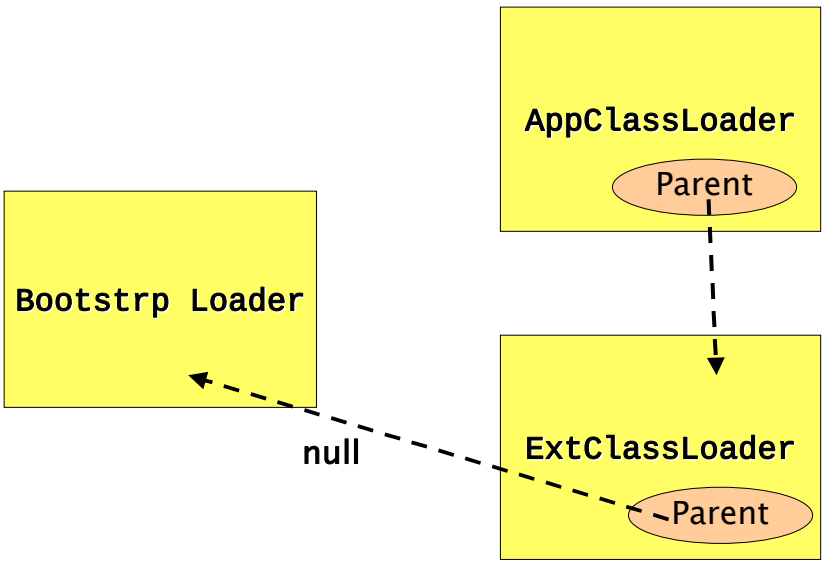
執行結果為:

```
D:\my>java test
.
d:\test
I get loaded

D:\my>_
```

因為我們並沒有在 d:\test 這個路徑底下放上 testlib.class，所以如果更改系統參數可以影響類別載入器的話，理論上應該無法正常地載入 testlib.class，可是螢幕輸出確告訴我們 testlib.class 被成功地載入。這個範例足以證明 AppClassLoader 完全不受系統參數 java.class.path 的更改而產生影響，您一樣可以試著測試改變 java.ext.dirs 來測試它對 ExtClassLoader 的影響。

委派模型

前面我們曾經提過「Bootstrap Loader 所做的初始工作中，除了也做一些基本的初始化動作之外，，最重要的就是載入定義在 sun.misc 命名空間底下的 Launcher.java 之中的 ExtClassLoader，並設定其 Parent 為 null，然後 Bootstrap Loader 再載入定義在 sun.misc 命名空間底下的 Launcher.java 之中的 AppClassLoader，並設定其 Parent 為之前產生的 ExtClassLoader 實體。」這個動作產生了所謂的「類別載入器的階層體系」，如下圖所示:



而之所以有階層體系的存在，是為了實現委派模型。所謂的委派模型，用簡單的話來講，就是**「類別載入器有載入類別的需求時，會先請示其 Parent 使用**

其搜尋路徑幫忙載入，如果 Parent 找不到，那麼才由自己依照自己的搜尋路徑搜尋類別」，請注意，**這句話具有遞迴性**。底下將用幾個小測試來說明這句話的意涵。測試前，我們將撰寫兩個類別: test.java 與 testlib.java，內容如下:

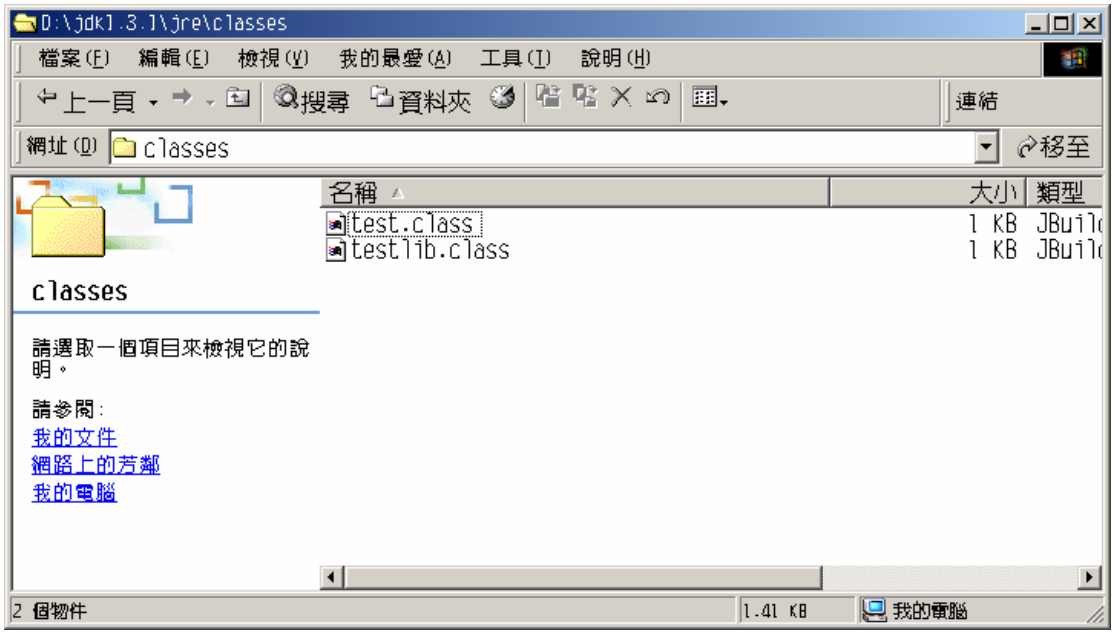
檔案:test.java

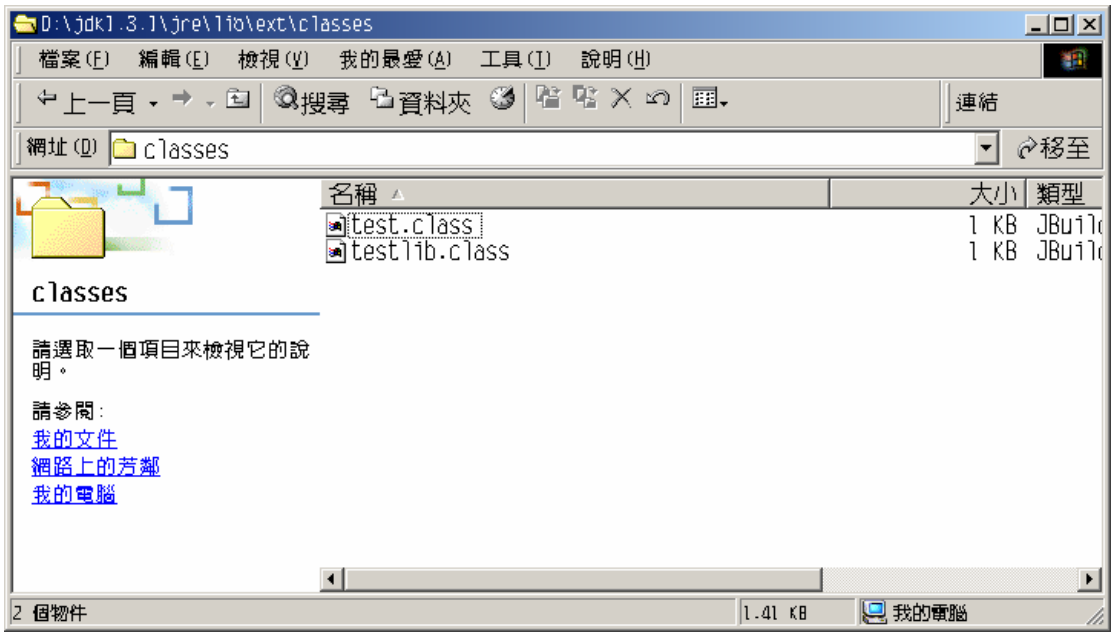
```
public class test
{
	public static void main(String args[])
	{
		System.out.println(test.class.getClassLoader()) ;
		testlib tl = new testlib() ;
		tl.print() ;
	}
}
```

檔案:testlib.java

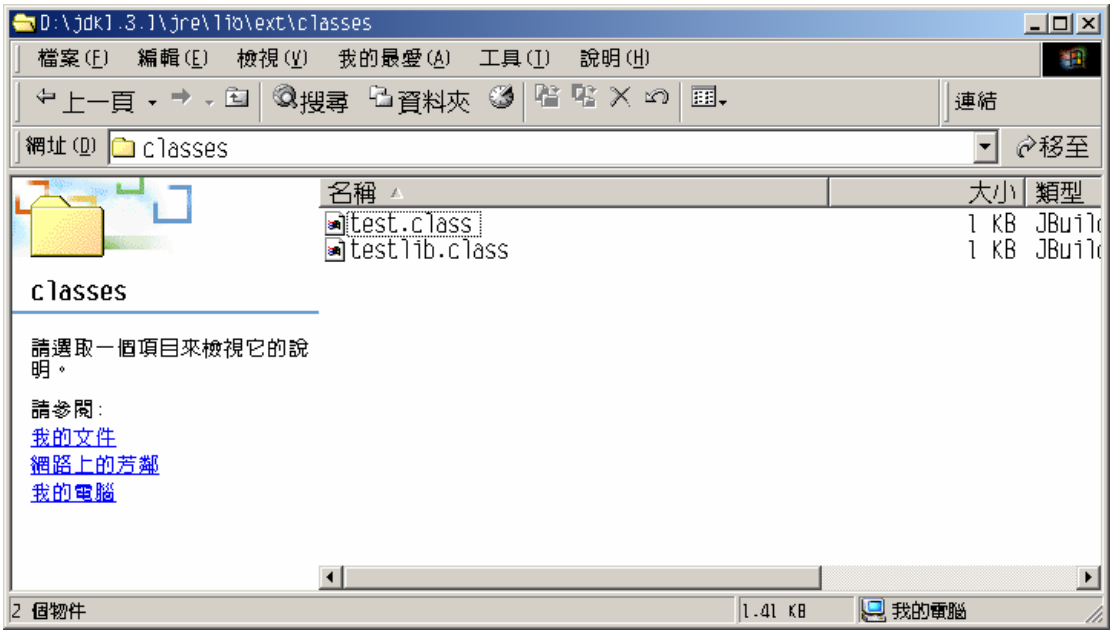
```
public class testlib
{
	public void print()
	{
		System.out.println(this.getClass().getClassLoader()) ;
	}
}
```

這兩個類別的作用就是印出載入它們的類別載入器是誰。將它們編譯成 test.class 與 testlib.class 之後，我們再複製兩份，分別至於**<JRE 所在目錄>\classes** 底下(注意，您的系統下應該沒有此目錄，您必須自己建立)與**<JRE 所在目錄>\lib\ext\classes**(注意，您的系統下應該沒有此目錄，您必須自己建立)底下，如下圖所示:





而 **d:\my** 底下也保有一份:



然後我們切換到 d:\my 目錄下，輸入 **java test** 開始進行測試(注意，此測試指令和路徑在底下的測試中將永遠不會改變)，測試前，我們會先用表格表示目前 test.class 與 testlib.class 的分布情況，然後解說不厭其煩地解說執行結果的成因，有些執行結果會發生錯誤，我們也會說明如何解決(或者是根本無法解決)。

測試一:

<JRE 所在目錄>\classes 底下 test.class testlib.class
<JRE 所在目錄>\lib\ext\classes 底下 test.class testlib.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
null
null

D:\my>
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。由於**<JRE 所在目錄>\classes** 目錄為

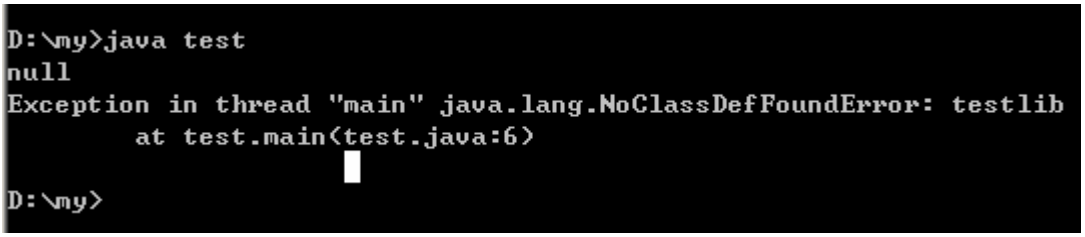
Bootstrap Loader 的搜尋路徑之一，所以 Bootstrap Loader 找到了 test.class，因此將它載入。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 Bootstrap Loader 所載入，所以 testlib.class 內定是由 Bootstrap Loader 根據其搜尋路徑來尋找，因為 testlib.class 也位於 Bootstrap Loader 可以找到的路徑下，所以也被載入了。

最後我們看到 test.class 與 testlib.class 都是由 Bootstrap Loader(null)載入。

測試二:

<JRE 所在目錄>\classes 底下 test.class
<JRE 所在目錄>\lib\ext\classes 底下 test.class testlib.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:



從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。由於**<JRE 所在目錄>\classes** 目錄為 Bootstrap Loader 的搜尋路徑之一，所以 Bootstrap Loader 找到了 test.class，因此將它載入。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 Bootstrap Loader 所載入，所以 testlib.class 內定是由 Bootstrap Loader 根據其搜尋路徑來尋找，但是因為 Bootstrap Loader 根本找不到 testlib.class(被我們刪除了)，而 Bootstrap Loader 又沒有 Parent，所以無法載入 testlib.clss。

輸出告訴我們，test.class 由 Bootstrap Loader 載入，且最後印出的訊息是 NoClassDefFoundError，代表無法載入 testlib.class。這個問題沒有比較簡單的方法能夠解決，但是仍然可以透過較複雜的 Context Class Loader 來解決。這個技巧在本文中不討論。

測試三:

<JRE 所在目錄>\classes 底下 testlib.class
<JRE 所在目錄>\lib\ext\classes 底下 test.class testlib.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
sun.misc.Launcher$ExtClassLoader@ac738
null
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。但是 Bootstrap Loader 無法在其搜尋路徑下找到 test.class(被我們刪掉了)，所以 ExtClassLoader 只得自己搜尋。因此ExtClassLoader在其搜尋路徑**<JRE所在目錄>\lib\ext\classes**底下找到 test.class，因此將它載入。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 ExtClassLoader 所載入，所以 testlib.class 內定是由 ExtClassLoader 根據其搜尋路徑來尋找，但是因為 ExtClassLoader 有 Parent，所以要先由 Bootstrap Loader 先幫忙尋找，testlib.class 位於 Bootstrap Loader 可以找到的路徑下，所以被 Bootstrap Loader 載入了。

最後我們看到 test.class 由 ExtClassLoader 載入，而 testlib.class 則是由 Bootstrap Loader(null)載入。

測試四:

<JRE 所在目錄>\classes 底下
<JRE 所在目錄>\lib\ext\classes 底下 test.class testlib.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
sun.misc.Launcher$ExtClassLoader@ac738
sun.misc.Launcher$ExtClassLoader@ac738

D:\my>
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。但是 Bootstrap Loader 無法在其搜尋路徑下找到 test.class(被我們刪掉了)，所以 ExtClassLoader 只得自己搜尋。因此 ExtClassLoader 在其搜尋路徑**<JRE 所在目錄>\lib\ext\classes** 底下找到 test.class，因此將它載入。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 ExtClassLoader 所載入，所以 testlib.class 內定是由 ExtClassLoader 根據其搜尋路徑來尋找，ExtClassLoader 一樣要請求其 Parent 先試著載入，但是 Bootstrap Loader 根本找不到 testlib.class(被我們刪除了)，所以只能由 ExtClassLoader 自己來，ExtClassLoader 在其搜尋路徑**<JRE 所在目錄>\lib\ext\classes** 底下找到 testlib.class，因此將它載入。

最後我們看到 test.class 與 testlib.clas 都是由 ExtClassLoader 載入。

測試五:

<JRE 所在目錄>\classes 底下
<JRE 所在目錄>\lib\ext\classes 底下 test.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
sun.misc.Launcher$ExtClassLoader@ac738
Exception in thread "main" java.lang.NoClassDefFoundError: testlib
        at test.main(test.java:6)

D:\my>
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。但是 Bootstrap Loader 無法在其搜尋路徑下找到 test.class(被我們刪掉了)，所以 ExtClassLoader 只得自己搜尋。因此 ExtClassLoader 在其搜尋路徑**<JRE 所在目錄>\lib\ext\classes** 底下找到

test.class，因此將它載入。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 ExtClassLoader 所載入，所以 testlib.class 內定是由 ExtClassLoader 根據其搜尋路徑來尋找，ExtClassLoader 一樣要請求其 Parent 先試著載入，但是 Bootstrap Loader 根本找不到 testlib.class(被我們刪除了)，所以只能由 ExtClassLoader 自己來，ExtClassLoader 也無法在自己的搜尋路徑中找到 testlib.class，所以產生錯誤訊息。

輸出告訴我們，test.class 由 ExtClassLoader 載入，且最後印出的訊息是 NoClassDefFoundError，代表無法載入 testlib.class。要解決問題，我們必須讓 ExtClassLoader 找的到 testlib.class 才行，所以請使用選項 –Djava.ext.dirs=<路徑名稱> 來指定，請注意，ExtClassLoader 只會自動搜尋底下的 classes 子目錄或是 JAR 檔，其他的子目錄或其他類型的檔案一概不管。此外，這個錯誤亦可以透過 Context Class Loader 的技巧來解決。

測試六:

<JRE 所在目錄>\classes 底下
<JRE 所在目錄>\lib\ext\classes 底下 testlib.class
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$ExtClassLoader@ac738

D:\my>
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。Bootstrap Loader 無法在其搜尋路徑下找到 test.class(被我們刪掉了)，所以轉由 ExtClassLoader 來搜尋。ExtClassLoader 仍無法在其搜尋路徑**<JRE 所在目錄>\lib\ext\classes** 底下找到 test.class，最後只好由 AppClassLoader 載入它自己在搜尋路徑底下找到的 test.class。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 AppClassLoader 所載入，所以 testlib.class 內定是由 AppClassLoader 根據其搜尋路徑來尋找，AppClassLoader 一樣要請求其 Parent 先試著載入，但是 Bootstrap Loader 根本找不到 testlib.class(被我們刪除了)，所以回頭轉

由 ExtClassLoader 來搜尋，ExtClassLoader 在其搜尋路徑**<JRE 所在目錄>\lib\ext\classes** 底下找到 testlib.class，因此將它載入。

最後我們看到 test.class 由 AppClassLoader 載入，而 testlib.class 則是由 ExtClassLoader 載入。

測試七:

<JRE 所在目錄>\classes 底下
<JRE 所在目錄>\lib\ext\classes 底下
d:\my 底下 test.class testlib.class

輸出結果如下圖所示:

```
D:\my>java test
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$AppClassLoader@7d8483

D:\my>
```

從輸出我們可以看出，當 AppClassLoader 要載入 test.class 時，先請其 Parent，也就是 ExtClassLoader 來載入，而 ExtClassLoader 又請求其 Parent，即 Bootstrap Loader 來載入 test.class。但是 Bootstrap Loader 無法在其搜尋路徑下找到 test.class(被我們刪掉了)， ExtClassLoader 只無法在其搜尋路徑下找到 test.class(被我們刪掉了)，最後只好由 AppClassLoader 載入它在自己搜尋路徑底下找到的 test.class。接著在 test.class 之內有載入 testlib.class 的需求，由於 test.class 是由 AppClassLoader 所載入，所以 testlib.class 內定是由 AppClassLoader 根據其搜尋路徑來尋找，但是 Bootstrap Loader 根本找不到 testlib.class(被我們刪除了)，ExtClassLoader 也找不到 testlib.class(被我們刪除了)，所以回頭轉由 AppClassloader 來搜尋，AppClassLoader 在其搜尋路徑底下找到 testlib.class，因此將它載入。

最後我們看到 test.class 和 testlib.class 都是由 AppClassLoader 載入。

類別載入體系

從以上的測試，我們可以知道一件事，就是:「類別載入器可以看到 Parent 所載入的所有類別，但是反過來並非除此」。所以在前面的測試二中，test.class 由 Bootstrap Loader 載入，且明明 AppClassLoader 可以找到，但是 Bootstrap Loader 就是看不到。如果您有用過 JDBC API 的經驗，您就會開始感到奇怪。

這是因為我們的 JDBC API 介面類別(屬於核心類別函式庫)都是由 Bootstrap Loader 或是 ExtClassLoader 來載入，可是 JDBC driver 常常是藉由 ExtClassLoader 或 AppClassLoader 來載入。假設 JDBC API 介面類別由 Bootstrap Loader 載入，而 JDBC driver 是藉由 ExtClassLoader 來載入，不就發生了與測試二相同的情形，那麼 JDBC 是如何克服這個問題的呢? 其實不只是 JDBC，在 Java 領域中只要分成 **API**(Application Programming Interface，公開制定，會成為核心類別函式庫(由 Bootstrap Loader 載入)或擴充類別函式庫(由 ExtClassLoader 載入))與 **SPI**(Service Provide Interface，由特定廠商撰寫，會成為擴充類別函式庫(由 ExtClassLoader 載入)或應用程式(由 AppClassLoader 載入)的一部分)的函式庫，都會遇到此問題，比方說 JDBC 或 JNDI 就是最好的例子。解決這個問題的方法是透過 Context Class Loader，不過本章不對此問題進行討論。

如果您對整個類別載入的方式仍有所疑問，請容筆者重新解釋一下之前程式

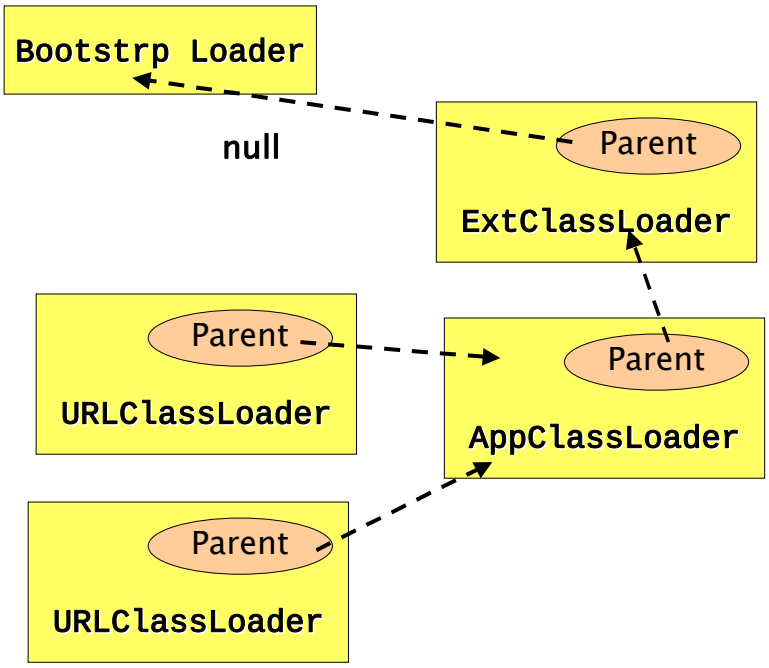
檔案:Office.java

```
import java.net.* ;
public class Office
{
	public static void main(String args[]) throws Exception
	{
		URL u = new URL("file:/d:/my/lib/") ;
		URLClassLoader ucl = new URLClassLoader(new URL[]{ u }) ;
		Class c = ucl.loadClass(args[0]) ;
		Assembly asm = (Assembly) c.newInstance() ;
		asm.start() ;

		URL u1 = new URL("file:/d:/my/lib/") ;
		URLClassLoader ucl1 = new URLClassLoader(new URL[]{ u1 }) ;
		Class c1 = ucl1.loadClass(args[0]) ;
		Assembly asm1 = (Assembly) c1.newInstance() ;
		asm1.start() ;

		System.out.println(Office.class.getClassLoader()) ;
		System.out.println(u.getClass().getClassLoader()) ;
		System.out.println(ucl.getClass().getClassLoader()) ;
		System.out.println(c.getClassLoader()) ;
		System.out.println(asm.getClass().getClassLoader()) ;

		System.out.println(u1.getClass().getClassLoader()) ;
```

```
            System.out.println(ucl1.getClass().getClassLoader()) ;
            System.out.println(c1.getClassLoader()) ;
            System.out.println(asm1.getClass().getClassLoader()) ;

            System.out.println(Assembly.class.getClassLoader()) ;
        }
}
```

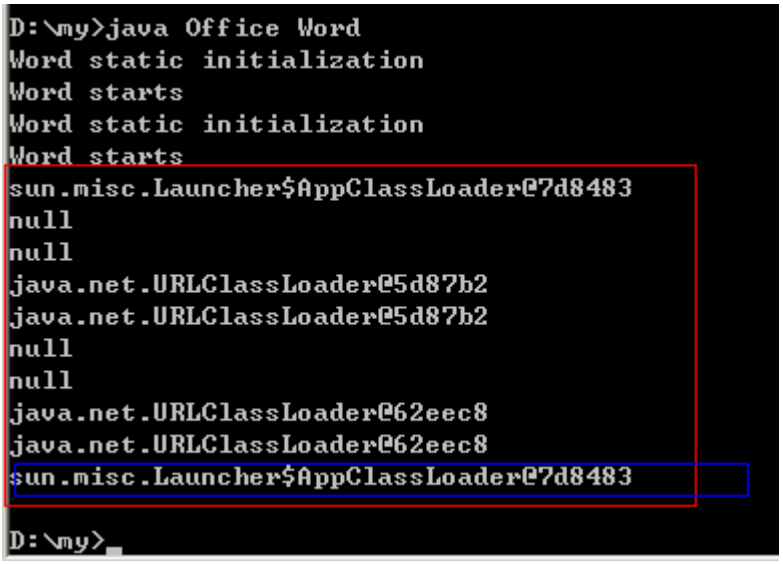
的執行結果。

在這個程式中，整個 Java 虛擬機器共產生五個類別載入器，如下圖所示:

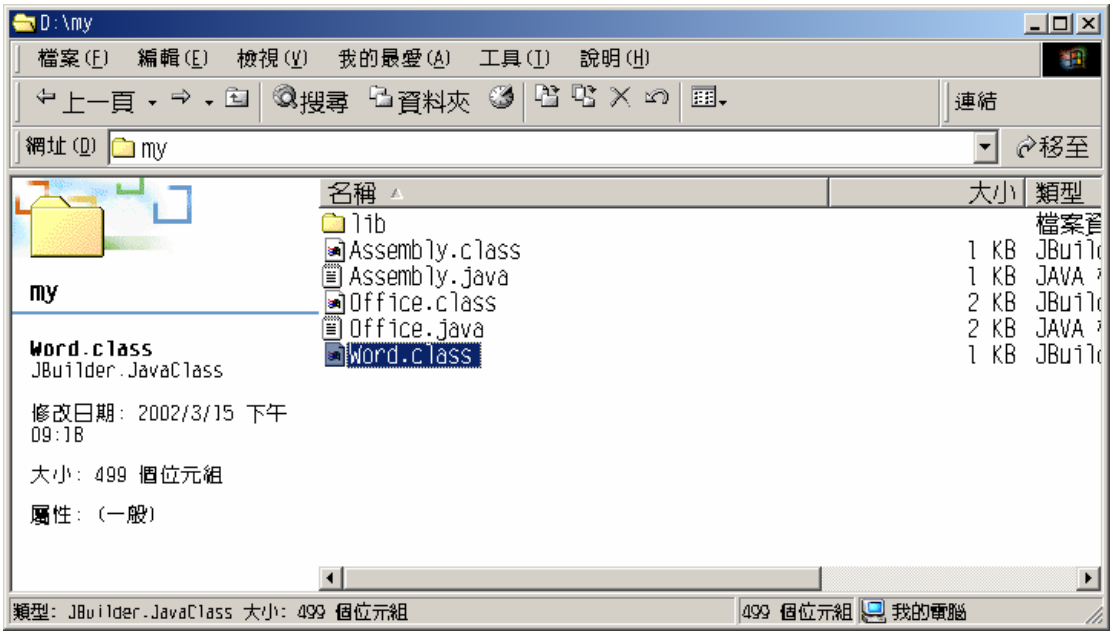


整個程序是這樣子的: 一開始的時候，我們在命令列輸入 java office Word，所以 AppClassLoader 必須負責載入 Office.class，由於其 Parent(ExtClassLoader)與 Parent 的 Parent(Bootstrap Loader)都無法在其搜尋路徑找到 Office.class，所以最後是由 AppClassLoader 載入 Office.class。我們在 Office.class 之中，需要建立 URL 與 URLClassLoader 的類別實體，預設使用的類別載入器是當時所在的類別本身的類別載入器，也就是利用 ClassLoader.getCallerClassLoader()取得的類別載入器(**注意，ClassLoader.getCallerClassLoader()是一個 private 的方法，所以我們無法自行叫用，這是 new 運算子本身隱函的呼叫機制中自行使用的)**。接下來，AppClassLoader 仍然會去請求其 Parent(ExtClassLoader)與 Parent 的 Parent(Bootstrap Loader)來載入，其中，Bootstrap Loader 在**<JRE 所在目錄>\lib\rj.jar** 之中找到 URL.class 與 URLClassLoader.class，所以理所當然由 Bootstrap Loader 負責載入。最後，我們請求 URLClassLoader 到相對路徑下的 lib\子目錄載入 Word.class，載入 Word.class 之前，必須先載入

Assembly.class，所以 URLClassLoader 會請求其 Parent(AppClassLoader)、Parent 的 Parent(ExtClassLoader)與 Parent 的 Parent 的 Parent(Bootstrap Loader)來載入，所以最後會由 AppClassLoader 載入 Assembly.class，而由 URLClassLoader 載入 Word.class。然後因為 Assembly.class 的實體(asm 與 asm1)參考到的事 Word.class 的實體(c 與 c1)，所以最後印出結果是 URLClassLoader 載入的，但是如果我們直接用 Assembly.getClassLoader()，就會顯示出是 AppClassLoader 載入的。所以輸出如下圖所示:



但是，如果我們特意把 Word.class 放在 AppClassLoader 找的到的地方(例如與 office.class 放在同一個目錄)，



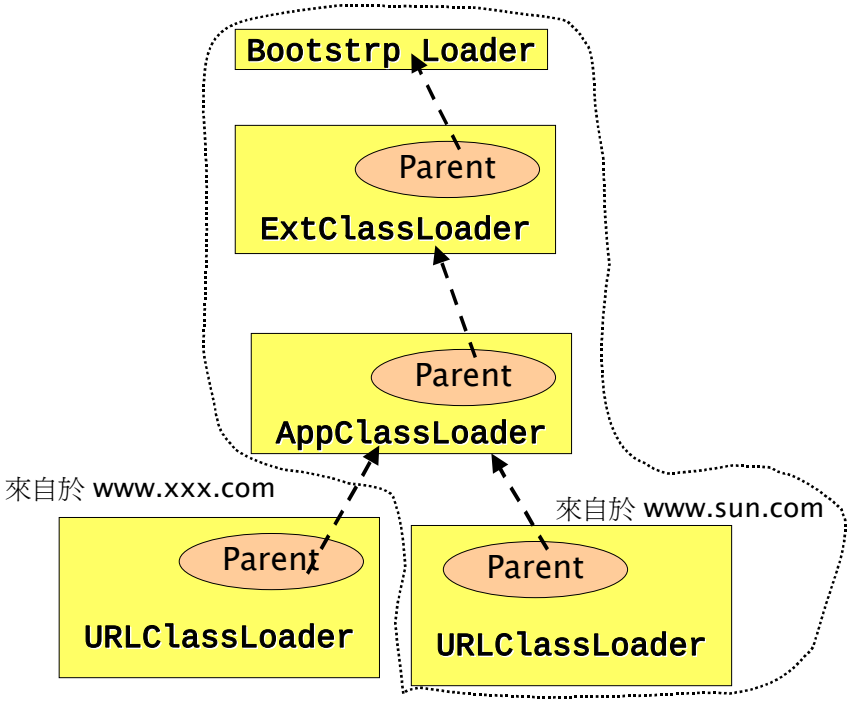
就會產生底下結果:

```
D:\my>java Office Word
Word static initialization
Word starts
Word starts
sun.misc.Launcher$AppClassLoader@7d8483
null
null
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$AppClassLoader@7d8483
null
null
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$AppClassLoader@7d8483
sun.misc.Launcher$AppClassLoader@7d8483

D:\my>
```

這是因為 URLClassLoader 委派給 AppClassLoader 來載入類別時，AppClassLoader 在自己的搜尋路徑下找到了 Word.class，所以輸出的時候告訴我們 Word.class 皆是由 AppClassLoader 載入。

類別載入器的功用

從上面的種種測試和說明，我們了解了類別載入器和其載入機制是一個非常複雜的系統，那麼為何要設計這麼複雜的系統呢? 除了可以達到動態性之外，其實最重要的原因莫過於安全性。我們以下面這張圖來說明:



這張圖說明了兩件事情，第一，假設我們利用 URLClassLoader 到網路上的任何地方下載了其他的類別，URLClassLoader 都不可能下載 AppClassLoader、ExtClassLoader、或者 Bootstrap Loader 可以找到的同名類別(指全名，套件名稱+類別名稱)，因此，蓄意破壞者根本沒有機會植入有問題的程式碼於我們的

電腦之中(除非蓄意破壞者能潛入您的電腦，置換掉您電腦內的類別檔，但是這已經不是 Java 所涉及的安全問題了，而是作業系統本身的安全問題)。第二，類別載入器無法看到其他相同階層之類別載入器所載入的類別，如上圖所示，圖中虛線索框起來的部分意指從 www.sun.com 下載程式碼的類別載入器所能看到的類別。告訴我們從 www.sun.com 載入的類別，無法看到 www.xxx.com 載入的類別，這除了意味著不同的類別載入器可以載入完全相同的類別之外，也排除了誤用或惡意使用別人程式碼的機會。

總結

我們花了很長的篇幅，才探討完 Java 裡頭類別載入器的特性，不知道您是否對類別載入器有了深入的認識呢? 類別載入器是 Java 平台上最神秘，也是最有趣的一個組件。透過類別載入器，除了可以達成程式的動態性之外，更能夠做到無懈可擊的安全性(附帶一提，Java 的安全架構是由 Sun Microsystems 裡頭優秀的華裔科學家 宮力(Gong Li)博士所設計，這真是華人的驕傲)。市面上許多提到安全性的書籍，或多或少都會提到類別載入器，大多數都會以特別一個章節來解說類別載入器，如果您對安全性的問題有興趣，您可以參考 O'Reilly 所出版的 Java Security 2/e 一書:

中文版網址:

http://www.oreilly.com.tw/chinese/java/java_security2.html

原文版網址:

http://www.oreilly.com/catalog/javasec2/