Chapter

1

初探

在這個電腦發展日新月異的時代，軟體產品不斷推陳出新、讓人應接不暇，軟體需求更是變幻莫測，難以捉摸。作為技術人員，我們在軟體開發過程中常常會遇到程式碼重複的問題，只好對系統進行大量改動，這不但帶來很多額外工作，而且會給產品帶來不必要的風險。因此，良好、穩固的軟體架構就顯得至關重要。設計模式正是為了解決這些問題，它針對各種場景提供了適合的程式碼模組的重用及擴展解決方案。

設計模式最早於 1994 年由 Gang Of Four（四人幫）提出，並以物件導向語言 C++ 作為範例，如今已大量應用於 Java、C# 等物件導向語言所開發的程式中。其實設計模式和程式語言並不是密切相關的，因為程式語言只是人與電腦溝通的媒介，它們可以用自己的方式去實現某種設計模式。從某種意義上講，設計模式並不是指某種具體的技術，而更像是一種思想，一種格局。本書將以時下流行的物件導向程式語言 Java 為例，對 23 種設計模式逐一拆解、分析。

在學習設計模式之前，得先搞清楚到底什麼是物件導向。我們生活的現實世界裡充滿了各種物件，如大自然中的山川河流、花鳥魚蟲，現代文明中的高樓大廈、車水馬龍，我們每天都要面對它們，與它們溝通、互動，這是對物件導向最簡單的理解。為了將現實世界重現於電腦世界中，我們想了各種方法針對這些物件建立數位模型，但是理想很「豐滿」，而現實很「骨感」，我們永遠無法包羅萬象。人們在「造物」的過程中發現，各種模型並非孤立存在的，它們之間有著千絲萬縷的關聯，於是便出現了物件導向所特有的程式方法。我們利用封裝、繼承、多

碁峰 www.gotop.com.tw

型的方式去建模，從而大量減少重複程式碼、降低模組間耦合，像拼積木一樣組裝了整個「世界」。這裡提到的「封裝」「繼承」和「多型」便是物件導向的三大特性，它們是掌握設計模式不可或缺的先決條件與理論基礎，我們必須要對其進行全面透徹的理解。

1.1　封裝

想要理解封裝，可以先觀察一下現實世界中的事物，比如膠囊對於各類混合藥物的封裝；錢包對於現金、身份證及銀行卡的封裝；電腦機殼對於主機板、CPU 及記憶體等配件的封裝等。

由此可見，封裝在我們生活中隨處可見。我們舉一個現實生活中常見的例子。如圖 1-1 所示，注意餐盤中的可樂杯，其中的飲料是被裝在杯子裡面的，杯子的最上面封上蓋子，只留有一個孔用於插吸管，這其實就是封裝。

封裝隱藏了杯子內部的飲料，也許還會有冰塊，而對於杯子外部來說只留有一個「介面」用於存取。這樣的做法是否多此一舉？又會帶來什麼好處呢？

圖 1-1　飲料的封裝

首先是方便、快捷，只有這樣我們才能拿著飲料杯四處行走，隨吸隨飲，而不至於把飲料灑得到處都是，因為零散的資料缺乏集中管理，難以引用、讀取。其次是封裝後的可樂更加乾淨、衛生，可以防止外部的灰塵落入，杯子裡面以關鍵字「private」宣告的可樂會成為內部的私有化物件，因此能防止外部隨意存取，避免造成資料汙染。最後，對外暴露的吸管介面帶來了極大便利，顧客在喝可樂時根本不需要關心杯子的內部物件和工作機制，如杯子中的冰塊如何讓可樂降溫、杯體內部的氣壓如何變化、氣壓差又是如何導致可樂流出等實現細節對顧客完全是不可見的，留給顧客的操作其實非常簡單，只需呼叫「吸」這個公有方法就可以喝到冰涼的可樂了。

我們再來分析一下對電腦主機的封裝，它必然需要一個機殼把各種配件封裝進去，如主機板、CPU、記憶體、顯示卡、硬碟等。一方面，機殼發揮保護作用，防止

碁峯 www.gotop.com.tw

異物（如老鼠、昆蟲等）進入內部而破壞電路；另一方面，機殼也不是完全封閉的，它一定對外預留有一些使用介面，如開機按鈕、USB 介面等，這樣使用者才能夠使用電腦，電腦主機的類別結構如圖 1-2 所示。

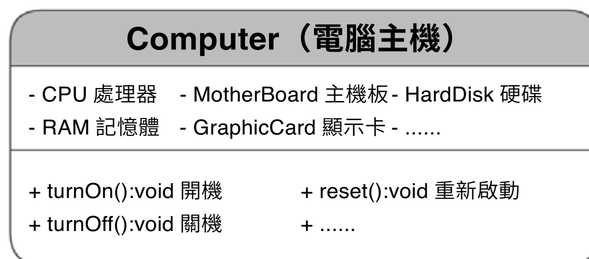


圖 1-2 電腦主機的類別結構

封裝的概念在歷史發展中也非常多見，其實它就是隨著時間的推移對前人經驗和技術產物的逐漸堆疊和組合的結果。舉個例子，早期的槍設計得非常簡陋，發射一發子彈需要很長時間去準備，裝填時要先把火藥倒入槍管內，然後裝入鉛彈，最後用棍子戳實後才能發射；而下一次發射還要再重複這一過程，耗時費力。為了解決這個問題，人們開始思考，既然彈藥裝填如此困難，那不如把彈頭和火藥組合後封裝在彈殼裡，這樣只要撞擊彈殼底部，彈頭就會因火藥爆炸的力量而發射出去，裝入槍膛的子彈便可發出，如圖 1-3 所示。

從彈藥到子彈的發展其實就是對彈藥的「封裝」，因此大大提高了裝彈效率。其實一次裝一發子彈還是效率欠佳，如果再進一步，在子彈外再封裝一層彈夾的話則會更顯著地提升效率。我們可以定義一個資料結構「堆疊」來模擬這個彈夾，保證最早壓入（push）的子彈最後彈出（pop），這就是堆疊結構「先進後出，後進先出」的特點。如此一來，子彈打完後只需更換彈夾就可以了。至此，封裝的層層堆疊又上了一個層次，在機槍被發明出來之後冷兵器時代就徹底結束了。

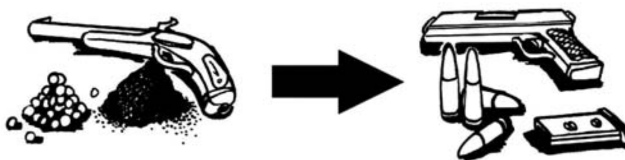


圖 1-3 彈藥的發展

碁峰 www.gotop.com.tw

在 Java 程式語言中，一對大括號「{}」就是類別的外殼、邊界，它能夠將類別的各種屬性及行為包裹起來，將它們封裝在類別內部並固化成一個整體。封裝好的類別如同一個黑匣子，外部無法看到內部的構造及運轉機制，而只能使用其暴露出來的屬性或方法。需要注意的是，我們千萬不要過度設計、過度封裝，更不要東拉西扯、亂攀親戚，像是把檯燈、輪子、茶杯等物品封裝在一起，或者在電腦主機裡封裝一個算盤。如果把一些不相干的物件硬生生封裝在一起，就會使程式碼變得莫名其妙，難於維護與管理，所謂「物極必反，過猶不及」，所以封裝一定要適度。

1.2 繼承

繼承是非常重要的物件導向特性，如果沒有它，程式碼量會變得非常龐大且難以維護、修改。繼承可以使父類別的屬性和方法延續到子類別中，這樣子類別就不需要重複定義，並且子類別可以透過重寫來修改繼承而來的方法實現，或者透過追加達到屬性與功能擴展的目的。從某種意義上講，如果說類別是物件的樣板，那麼父類別（或超類別）則可以被看作樣板的樣板。

生物一代一代延續是靠什麼來保持父輩的特徵呢？沒錯，答案就是遺傳基因 DNA，如圖 1-4 所示。正所謂「龍生龍鳳生鳳，老鼠的兒子會打洞」，如果沒有這個遺傳機制，程式碼的數量就會急速膨脹，很多功能、資源都會出現重複定義的情況，這樣就會造成極大的冗餘和資源的浪費，所以受自然界的啟發，物件導向就有了繼承機制。

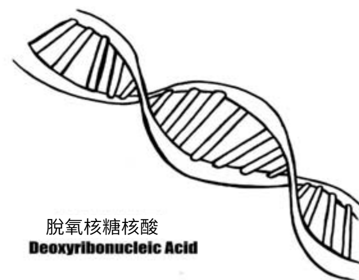


圖 1-4　生物的遺傳基因

舉個例子，兒子從父親那裡繼承了一些東西，就不需要透過別的方式獲得了，如繼承家產。再舉個例子，我們知道，狗是人類忠實的朋友，它們在一萬多年的進化過程中不斷繁衍，再加上人類的培育，衍生出許多品種，如圖 1-5 所示。

碁峰 www.gotop.com.tw

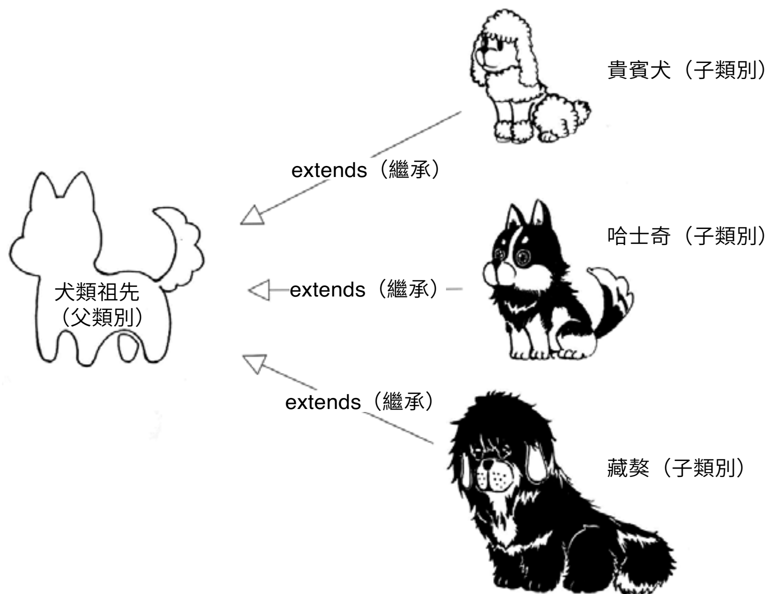


圖 1-5 犬類別的繼承

基於圖 1-5 所示的繼承關係，我們思考一下如何用程式碼來建模。倘若為每個犬類品種都定義一個類別並封裝各自的屬性和方法，這顯然不行，因為類別一多勢必會造成程式碼泛濫。其實，不管是什麼犬類品種，它們都有某些共同的特徵與行為，如吠叫行為等，所以我們需要把犬類別共有的基因抽離出來，並封裝到一個犬類別祖先中以供後代繼承，請參看程式 1-1。

程式 1-1 犬類別的祖先 Dog

```
1.  public class Dog {
2.    protected String breeds;// 品種
3.    protected boolean sex;// 性別
4.    protected String color;// 毛色
5.    protected int age;// 年齡
6.
7.    public Dog(String breeds) {
8.      this.age = 0; // 初始化為 0 歲
9.      this.breeds = breeds; // 初始化犬類品種
10.   }
11.
12.   public void bark(){// 吠叫
13.     System.out.println(" 汪汪汪 ");
14.   }
15.
```


碁峰
www.gotop.com.tw

透過這場對話，我們對電腦和周邊裝置以及它們之間的關係有了更深刻的認識。電腦中裝了一個 USB 介面，這就是「封裝」，而鍵盤、滑鼠及攝影機都是 USB 介面的實現類別，從廣義上理解這就是一種「繼承」，所以電腦的 USB 介面就能接駁各式各樣的 USB 裝置，這就是「多型」。我們來看它們的類別結構，如圖 1-10 所示。

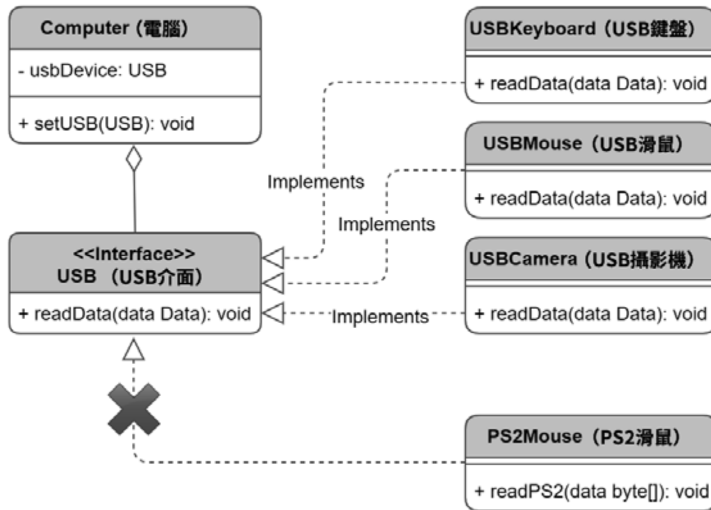


圖 1-10　現代電腦的類別結構

透過對電腦介面的抽象化、標準化，我們對各個模組重新分類、規劃，併合理封裝，最終實現電腦與周邊裝置的徹底解耦。多型化的周邊裝置使電腦功能更加強大、靈活、可擴展、可取代。其實這就是設計模式中非常重要的一種「策略模式」，介面的定義是解決耦合問題的關鍵所在。但對於一些老舊的介面裝置模組，我們暫時還無法使用，正如同上面故事裡那個可憐的 PS2 滑鼠。

我們都知道有一種裝置叫轉換器，它能輕鬆地將老舊的介面裝置轉接到新的介面，以達到相容的目的，這就是「轉接器模式」。這些設計模式後續都會被講到，我們會由淺入深、一步一個腳印地逐個解析。讀者一定要邊學邊思考，理論一定要與實踐結合才能舉一反三、融會貫通，如此才能合理有效地利用設計模式設計出更加優雅、健壯、靈活的應用程式。

碁峰 www.gotop.com.tw

這樣的程式邏輯看似沒問題，但其實在多執行緒模式下是有缺陷的。試想如果是並行請求的話，程式第 10 行的判空邏輯就會同時成立，這樣就會多次實例化太陽，並且對 sun 進行多次賦值（覆蓋）操作，這違背了單例的理念。我們再來改良一下，把請求方法加上 synchronized（同步鎖）讓其同步，如此一來，某執行緒呼叫前必須獲取同步鎖，呼叫完後會釋放鎖給其他執行緒用，也就是給請求排隊，一個接一個按順序來，請參看程式 2-6。

程式 2-6　太陽類別 Sun

```
1.  public class Sun {
2.
3.      private static Sun sun;// 這裡不進行實例化
4.
5.      private Sun(){// 構造方法私有化
6.
7.      }
8.
9.      public static synchronized Sun getInstance() {// 此處加入同步鎖
10.         if (sun == null) {// 如果無日才造日
11.             sun = new Sun();
12.         }
13.         return sun;
14.     }
15.
16. }
```

如程式 2-6 所示，我們將太陽類別 Sun 中第 9 行的 getInstance() 改成了同步方法，如此可避免多執行緒陷阱。然而這樣的做法是要付出一定代價的，試想，執行緒還沒進入方法內部便不管三七二十一直接加鎖排隊，會造成執行緒阻塞，資源與時間被白白浪費。我們只是為了實例化一個單例物件而已，不必如此興師動眾，使用 synchronized 讓所有請求排隊等候。所以，要保證多執行緒並行下邏輯的正確性，同步鎖一定要加得恰到好處，其位置是關鍵所在，請參看程式 2-7。

程式 2-7　太陽類別 Sun

```
1.  public class Sun {
2.
3.      private volatile static Sun sun;
4.
5.      private Sun(){// 構造方法私有化
6.
7.      }
8.
```


碁峯 www.gotop.com.tw

```
9.      public static Sun getInstance() {// 華山入口
10.         if (sun == null) {// 觀日台入口
11.             synchronized(Sun.class){// 觀日者進行排隊
12.                 if (sun == null) {
13.                     sun = new Sun();// 旭日東升
14.                 }
15.             }
16.         }
17.         return sun; //……陽光普照，其餘人不必再造日
18.     }
19. }
```

如程式 2-7 所示，我們在太陽類別 Sun 中第 3 行對 sun 變數的定義不再使用 find 關鍵字，這意味著它不再是常量，而是需要後續賦值的變數；而關鍵字 volatile 對靜態變數的修飾則能保證變數值在各執行緒存取時的同步性、唯一性。需要特別注意的是，對於第 9 行的 getInstance() 方法，我們去掉了方法上的關鍵字 synchronized，使大家都可以同時進入方法並對其進行開發。

請仔細閱讀每行程式碼的注釋，有些人（執行緒）起早就是為了觀看日出，那麼這些人會透過第 10 行的判空邏輯進入觀日台。而在第 11 行又加上了同步塊以防止多個執行緒進入，這就類似於觀日台是一個狹長的走廊，大家排隊進入。隨後在第 12 行又進行一次判空邏輯，這就意味著只有隊伍中的第一個人造了太陽，有幸看到了日出的第一縷陽光，而後面的人則統統離開，直到第 17 行得到已經造好的太陽，如圖 2-2 所示。

圖 2-2　觀日台

隨後發生的事情就不難想見了，太陽高高升起，實例化操作完畢，起晚的人們都無須再進入觀日台，直接獲取太陽實例就可以了，陽光普照大地，將溫暖灑向人間。

碁峰 www.gotop.com.tw

大家注意到沒有，我們一共用了兩個嵌套的判空邏輯，這就是懶載入模式的「雙檢鎖」：外層放寬入口，保證執行緒並行的高效性；內層加鎖同步，保證實例化的單次執行。如此裡應外合，不僅達到了單例模式的效果，還完美地保證了構建過程的執行效率，一舉兩得。

2.4 大道至簡

相較於「懶漢模式」，其實在大多數情況下我們通常會更多地使用「餓漢模式」。原因在於這個單例遲早是要被實例化占用記憶體的，延遲懶載入的意義並不大，加鎖解鎖反而是一種資源浪費，同步更是會降低 CPU 的利用率，使用不當的話反而會帶來不必要的風險。越簡單的包容性越強，而越複雜的反而越容易出錯。我們來看單例模式的類別結構，如圖 2-3 所示。單例模式的角色定義如下。

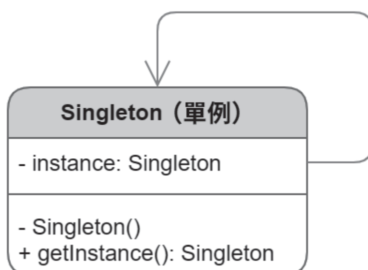


圖 2-3　單例模式的類別結構

- Singleton（單例）：包含一個自己的類別實例的屬性，並把構造方法用 private 關鍵字隱藏起來，對外只提供 getInstance() 方法以獲得這個單例物件。

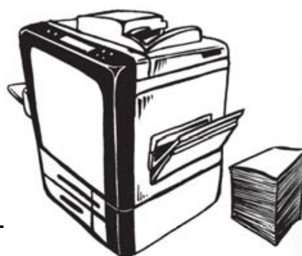
除了「餓漢」與「懶漢」這兩種單例模式，其實還有其他的實現方式。但萬變不離其宗，它們統統都是由這兩種模式發展、衍生而來的。我們都知道 Spring 框架中的 IoC 容器很好地幫我們託管了業務物件，如此我們就不必再親自動手去實例化這些物件了，而在預設情況下我們使用的正是框架提供的「單例模式」。誠然，究其程式碼實現當然不止如此簡單，但我們應該追本溯源，抓住其本質的部分，理解其核心的設計思想，再針對不同的應用場景做出相應的調整與變動，結合實踐舉一反三。

碁峰 www.gotop.com.tw

Chapter

3

原型



原型模式（Prototype），在製造業中通常是指大批次生產開始之前研發出的概念模型，並基於各種參數指標對其進行檢驗，如果達到了品質要求，即可參照這個原型進行批次生產。原型模式達到以原型實例建立副本實例的目的即可，並不需要知道其原始類別，也就是說，原型模式可以用物件建立物件，而不是用類別建立物件，以此達到效率的提升。

3.1　原件與副本

在講原型模式之前，我們得先搞清楚什麼是類別的實例化。相信大家一定見過活字印章，如圖 3-1 所示，當我們調整好需要的日期（初始化參數），再輕輕一蓋（呼叫構造方法），一個實例化後的日期便躍然紙上了，這個過程正類似於類別的實例化。

圖 3-1　印章實例化的過程

碁峰
www.gotop.com.tw

其實構造一個物件的過程是耗時耗力的。想必大家一定有過列印和影印的經驗。為了節省成本，我們通常會用印表機把電子文件列印到 A4 紙上（原型實例化過程），再用影印機把這份紙質文稿複製多份（原型複製過程），這樣既簡單又有效率。那麼，對於第一份列印出來的原文稿，我們可以稱之為「原型文件」，而對於影印過程，我們則可以稱之為「原型複製」，如圖 3-2 所示。

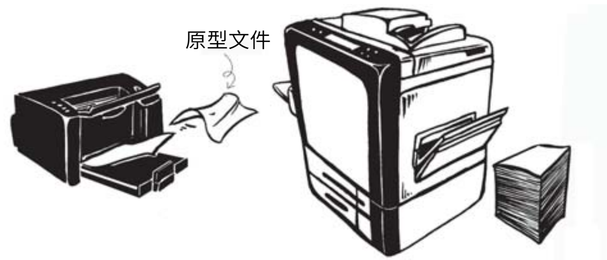


圖 3-2　對原文件的影印

3.2　卡頓的遊戲

想必大家已經明白了類別的實例化與複製之間的區別，兩者都是在造物件，但方法絕對是不同的。原型模式的目的是從原型實例複製出新的實例，對於那些有非常複雜的初始化過程的物件或者是需要耗費大量資源的情況，原型模式是更好的選擇。理論還需與實踐結合，下面開始實戰部分，假設我們準備設計一個空戰遊戲的程式，如圖 3-3 所示。

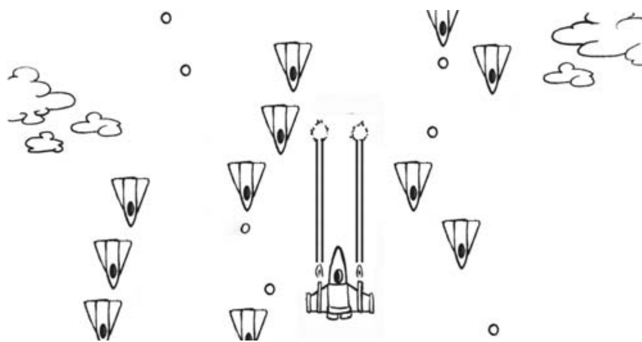


圖 3-3　空戰遊戲

碁峯 www.gotop.com.tw

除此之外，工廠方法模式是基於多元化產品的構造方法發展而來的，它開闢了產品多元化的生產模式，不同的產品可以交由不同的專業工廠來生產，例如皮鞋由皮鞋工廠來製造，汽車則由汽車工廠來製造，專業化分工明確，如圖 4-1 所示。

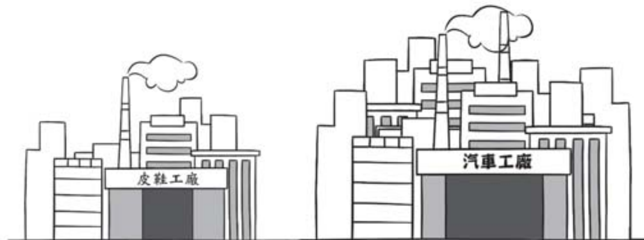


圖 4-1　專業化的工廠

4.2　遊戲角色建模

在製造產品之前，我們得先為它們建模。我們依舊以空戰遊戲來舉例，通常這類遊戲中主角飛機都擁有強大的武器裝備，以應對敵眾我寡的遊戲局面，所以敵人的種類就應當多樣化，以帶給玩家更加豐富多樣的遊戲體驗。於是我們增加了一些敵機、坦克，遊戲畫面如圖 4-2 所示。

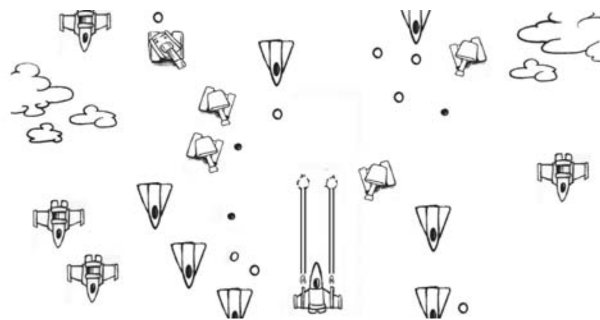


圖 4-2　空戰遊戲

如圖 4-2 所示，遊戲中敵人的種類有飛機和坦克，雖然它們之間的區別比較大，但總有一些共同的屬性或行為，例如一對用來描述位置狀態的座標，以及一個展示（繪製）方法，以便將自己繪製到相應的地圖位置上。好了，現在我們使用抽象類別來定義所有敵人的父類別，請參看程式 4-1。

碁峯
www.gotop.com.tw