第十三章

類別、物件和 tie

我要支持全世界的群眾，對抗階級的不平等。

— William E. Galdstone
1886 年 6 月 28 號在英國利物浦的演講

13.0 簡介

除了參考和模組以外，Perl 在 5.000 加入了物件。Perl 並不強迫你一定要使用標準的資料型態，可是它包含許多你可以使用的型態。這幫助許多使用者以他們想要的方式去作他們的工作。

你可以不使用物件去寫程式，這不像 JAVA 的程式都是由物件所組成。如果想以這種方式來設計程式，可以透過物件導向的手法來寫 Perl 程式。Perl 支援類別（class）、物件（object）、單一和多重繼承（inheritance）、實體成員函式（instance method）、類別成員函式（class method）、被覆蓋的成員函式（overridden method）的存取、建構函式（constructor）、解構函式（destructor）、運算子溢載（operator overloading）、自動載入的代理成員函式（proxy method）、委派（delegation）、物件的繼承階層、資源回收（garbage collection, GC）。

你可以根據需要的情況，來決定使用物件導向技巧的多寡。繫結（tie）是 Perl 裡唯一一定要使用物件導向的部分。但雖然如此，只有實作模組的人需要注意這個；一般的使用者不需要知道內部的運作情形。繫結在章節 13.15 有討論，它讓你輕易地攔截存取變數的動作。例如可以使用繫結去製造一個以 key 或值搜尋的雜湊。

在表面意義之下

如果你問十個人什麼是物件導向？你會得到十個不同的答案。人們討論像是抽象（abstraction）和封裝（encapuslation）這類的議題，試著去區隔這些物件導向程式語言的最基本單元，將其具名與撰文立著。但並不是全部的物件導向語言都提供相同的特徵，但是他們仍被視作是物件導向。當然這更產生了更多的論文和書籍。

我們將依循在 Perl 說明文件、perlobj(1) 說明書、《Perl 程式設計》第五章中所使用的專門用語。物件是附屬於類別的變數。成員函式是由函式與類別（或物件）所構成。在 Perl 之，一個類別就是一個套件 —— 通常也是個模組。物件是個參考，它指向已被宣告在類別內的某樣東西。bless 函式將參考對象（referent）與類別關聯在一起，這動作得要一個（或兩個）引數，第一個引數是個參考，指向被 bless 的東西，第二個引數（不一定會有）則是 bless 的目的地。

```
$object = {};                       # 雜湊參考
bless($object, "Data::Encoder");    # bless $object 到 Data::Encoder 類別內
bless($object);                     # bless $object 到目前的套件內
```

類別名就是套件名（這個範例中的 Data::Encoder）。因為類別是模組，所以 Data::Encoder 類別的程式碼存在於 Data/Encoder.pm 的檔案當中。如同傳統的模組，目錄結構的存在完全是為了方便；它不包含任何有關繼承、變數分享或任何其他的東西。和傳統的模組不同的是，一個物件模組很少會使用 Exporter。存取動作只能經由成員函式的呼叫，而不是匯入的函式或變數。

一旦物件被 bless 到別的地方，它的參考呼叫 ref 函式時，會傳回類別的名字，而不是參考對象的基本型態：

```
$obj = [3,5];
print ref($obj), " ", $obj->[1], "\n";
bless($obj, "Human::Cannibal");
print ref($obj), " ", $obj->[1], "\n";

ARRAY 5
Human::Cannibal 5
```

如你所見，一個參考即使被 bless 了，還是可以對它做解參考（dereference）的動作。通常，物件常被當作是已 bless 的雜湊參考來實作。你可以使用任何類型的參考，但雜湊參考是最有彈性的。它們讓你在物件中可以隨意命名資料欄位。

```
$obj->{Stomach} = "Empty";   # 存取物件內容
$obj->{NAME}    = "Thag";
 # 大寫的欄位名為標準輸出(可選擇)
```

雖然 Perl 允許你這麼做，可是任何類別外的程式碼都可以直接去存取物件的所有內容是一種不好的寫作方式。我想每個人都會同意，最好是有個代碼（handle）指向物件的資料，因此你看不到，而只能經由指定的成員函式存取資料。這讓類別的維修者要改變類別的內容時，不需要去改變每個使用到這個類別的應用程式碼。

成員函式

使用 -> 呼叫成員函式，這裡我們用引數 "data" 呼叫 $object 的 encode() 成員函式，且把傳回值存放在 $encoded 中。

```
$encoded = $object->encode("data");
```

這是個「物件成員函式」（object method），因為我們是在物件當中呼叫成員函式。我們也可以有「類別成員函式」（class method），就是用類別名稱呼叫成員函式。

```
$encoded = Data::Encoder->encode("data");
```

引用成員函式會呼叫相對應類別中的函式，該函式以物件成員函式的參考或類別成員函式的字串這兩種型態，作為初始引數被傳遞。章節 13.7 說明如何呼叫在執行時期才會被決定的成員函式。

大部分的類別提供建構函式，它可以傳回新的物件。不像一些物件導向語言，Perl 的建構函式沒有特別被命名。事實上你可以用任何喜歡的名稱為它命名。C++ 程式設計者傾向於將 Perl 裡的建構函式命名為 new。我們建議你根據解決的問題來有意義地命名建構函式。例如在 Tk 裡延伸到 Perl 的建構函式，會依據他們創造的視窗元件（widget）被命名。比較不普及的做法是，將匯出的函式取和類別相同的名稱；相關細節請參考章節 13.14 的範例「溢載的 StrNum 類別」。

下面是一個典型的建構函式：

```
sub new {
    my $class = shift;
    my $self  = {};         # 分配新的雜湊給物件
    bless($self, $class);
    return $self;
}
```

呼叫建構函式：

```
$object = Class->new();
```

如果沒有任何的繼承或其他在這個情況後面運作的行為的話，這和下面的式子實際上是相同的：

```
$object = Class::new("Class");
```

new() 函式的第一個引數是類別名稱，它是新的參考要被 bless 到的目的地。建構函式應該將該字串當作第二個引數傳遞給 bless()。

章節 13.1 也討論會傳回被 bless 的參考的函式。建構函式不一定要是類別成員函式。會傳回新物件的物件成員函式是很有用的，如章節 13.6 討論的。

解構函式是一個副常式，當物件的參考對象的記憶體被回收時，它會被執行。不像建構函式，你沒有命名它的選擇。解構函式必須命名為 DESTROY。DESTROY 如果存在，它將會在記憶體回收時，馬上被全部的物件呼叫。解構函式的使用與否是可以自由選擇的，請看章節 13.2 的介紹。

一些語言允許編譯器限制存取類別的成員函式。Perl 不是如此 —— 它允許程式碼去呼叫物件的任何成員函式。類別的作者應該清楚地說明公共（public）的成員函式（這些可能會被使用到），而且類別的使用者應該避開無正式出處（私有）的成員函式。

Perl 沒有去區別可在類別呼叫的成員函式（即類別成員函式）和可在物件呼叫的成員函式（即實體成員函式）。如果你想要某個成員函式只能被類別呼叫，請參照下面的敘述這麼做：

```
sub class_only_method {
    my $class = shift;
    die "class method called on object" if ref $class;
    # 更多的程式碼在此
}
```

如果你想要某個成員函式只能像是一個實體成員函式被呼叫，請參照下面的敘述這麼做：

```
sub instance_only_method {
    my $self = shift;
    die " instance method called on class" unless ref $self;
    # 更多的程式碼在此
}
```

如果程式碼的內容是在物件內呼叫未定義的成員函式，Perl 不會在編譯時期就指出錯誤；程式會在執行時期才引發例外。同樣地，傳遞一個非質數到一個只接受質數的成員函式，編譯器無法捕捉到這個情況。成員函式只是一個函式呼叫，它（們）的套件在執行時期才被決定。如同所有間接的函式，它們沒有原型的檢查 —— 因為這動作是在編譯時期發生。即使成員函式呼叫察覺到原型，Perl 的編譯器沒有辦法自動去檢查它是不是正確的型態，或是關於函式的引數範圍合不合法。Perl 原型是被使用來限制函式的引數情況，而不是檢查引數的範圍。章節 10.11 詳細說明了 Perl 對原型的獨特看法。

你可以使用 AUTOLOAD 技巧去捕捉不存在的成員函式呼叫，這可以避免因未定義成員函式而引發例外。在章節 13.11 有一個關於這的應用程式。

繼承

繼承（inheritance）定義類別的階層（hierarchy）。呼叫類別中未定義的成員函式，會針對指定名稱的成員函式的階層加以搜尋，搜尋的順序是第一個找到符合名稱的成員函式會被使用。繼承的意思是類別可位在其他類別之上，所以使用者不需要一再地寫相同的程式。這是軟體再利用的一個方式。

一些語言對於繼承提供特別的語法。在 Perl，每個類別（套件）可以把超類別（superclass，階層中的上一層類別）的串列放進套件全域變數 @ISA 中。當一個物件類別的未定義成員函式被呼叫，程式會在執行時期搜尋這個串列。如果列在 @ISA 中的套件沒有要被呼叫的成員函式，而套件有自己的 @ISA，Perl 會遞迴地尋找第一個套件擁有的 @ISA。

如果階層搜尋失敗，相同的檢查會再次執行，這一次會尋找 AUTOLOAD 成員函式。假設，$ob 是屬於類別 P，搜尋 $ob->meth() 的順序是：

- P::meth
- 遞迴地在 @P::ISA 的所有套件 S 中尋找任何 S::meth
- UNIVERSAL::meth
- P::AUTOLOAD 副常式
- 遞迴地在 @P::ISA 的所有套件 S 中尋找任何 S::AUTOLOAD()
- UNIVERSAL::AUTOLOAD 副常式

大部分的類別在它們的 @ISA 陣列當中只有一個項目，這種情況稱作單一繼承 (single inheritance)。類別的 @ISA 中有一個以上的元素表示多重繼承 (multiple inheritance)。對於多重繼承，各方的看法不一，但總而言之，Perl 支援多重繼承就是了。

章節 13.9 討論繼承的原理，以及如何設計一個類別使它可以容易地衍生出子類別。章節 13.10 說明一個子類別 (subclass) 如何在它的超類別內呼叫被覆蓋的成員函式。

Perl 不支援資料數值的繼承。一個類別可以，但不應該直接碰觸其他類別的資料，這會破壞封套 (envelope)，也會毀壞抽象性 (abstraction)。如果依循章節 13.10 和章節 13.12 的建議，這就不會是太大的問題。

間接物件的標號方式

呼叫成員函式的間接標號方式：

```
$lector = new Human::Cannibal;
feed $lector "Zak";
move $lector "New York";
```

其實就是下列敘述的替代語法：

```
$lector = Human::Cannibal->new();
$lector ->feed("Zak");
$lector ->move("New York");
```

這種間接的物件標號方式可能會特別吸引英語的使用者，與熟悉 C++ 的程式撰寫者（因為這與他們使用 new 的方式相同）。但不要被迷惑了。它有兩個嚴重的問題。一個是它依循著反覆無常的規則，就像 print 和 printf 的檔案代碼狹槽（slot）：

```
printf STDERR "stuff here\n";
```

這個狹槽如果要被填滿，它必須包含一個符號、一個區塊、或一個純數變數名稱；它不能是任何舊式的先數表示式（expression）。這有可能導致一個令人迷惑的問題，如下面兩列：

```
move $obj->{FIELD};                # 可能是錯的
move $ary[$i];                     # 可能是錯的
```

令人驚訝的是，他們會被分析（parse）成：

```
$obj->move->{FIELD};               # 驚訝吧！
$ary->move->[$i];                  # 驚訝吧！
```

而不是：

```
$obj->{FIELD}->move();             # 你希望是如此
$ary[$i]->move;                    # 你希望是如此
```

第二個問題是在編譯時期 Perl 必須推測 name 和 move 是函式或成員函式。通常 Perl 會取得正確的解答，但當它不是時，會取得一個函式呼叫，卻把它當成員函式編譯它。這會產生難以捉摸的錯誤，而且非常難以解決。使用 -> 不會發生這些令人困惑的問題，所以我們建議你使用這個方式。

物件術語的一些註解

在物件導向的世界，許多文字只描述少許的概念。如果你寫過其它物件導向語言的程式，你會想要知道熟悉的名詞和概念對應於 Perl 的那個部份。

舉個例子，類別的物件實體和這些物件的成員函式稱為實體成員函式。對每個物件都有特殊意義的資料欄位，通常被稱為實體資料（instanec data）或物件屬性（object attribute），對類別的所有成員都具有相同意義的資料欄位稱為類別資料（class data）、類別屬性（class attribute）、或靜態資料成員（static data member）。

基底類別（base class）、通用類別（generic class）、超類別都是指相同的概念 —— 繼承階層裡的父階層；反過來，衍生類別（derived class）、特殊類別（specific class）、子類別則是與之相對的關係 —— 繼承階層裡的子階層。

C++ 有靜態成員函式（static method）、虛擬成員函式（virtual method）、實體成員函式，但 Perl 只有類別成員函式和物件成員函式（object method）。實際上，Perl 只有成員函式，至於成員函式是作為類別成員函式或物件成員函式，則是單純依用法而定。你可以在物件上呼叫類別成員函式（需要一個字串引數），或在類別上呼叫物件成員函式（需要一個參考），反之亦然，但如果這樣做，不要期望有合理的結果。

C++ 程式設計師腦中的全域（類別）建構函式和解構函式，分別對應於模組的初始化程式碼和每個模組的 END{} 區塊。

以 C++ 的觀點來看，Perl 的所有模組都是虛擬的，它們從不檢查引數的函式原型，不像檢查內建函式和使用者定義的函式那樣。原型在編譯時期被編譯器檢查。不到執行時期，就無法決定成員函式呼叫的函式。

隱含在哲學之外的意義

Perl 的 OO 程式設計給你許多自由：你可以用不止一種方法完成一件事情（可以 bless 任何資料型態以製作物件），可以檢查和修改你根本就沒寫過的類別（新增函式到它們的套件中），還可以用這些技巧寫一個令人一籌莫展的程式 —— 如果你真的想這麼做的話。

彈性較少的程式語言通常比較有限制。許多語言都盲目地致力於加強私有性、編譯時期的型態檢查、複雜的函式特色。Perl 沒有提供這類東西的相關物件，如果發現 Perl 物件導向的執行有點奇怪，那是因為你習慣了其他語言的語法，所以會覺得 Perl 奇怪。有些程式設計者想要讓物件有完全的私有性：perltoot(1) 說明書描述如何 bless closure 以產生物件，這會讓物件和 C++ 之中的物件一樣有私有性。

Perl 的物件是對的，而且對的很與眾不同！

相關資料

一般談論物件導向程式設計的資料文獻很少論及 Perl。Perl 隨附的說明文件是學習物件導向程式設計的好地方，尤其是物件的使用說明 perltoot(1)。perlobj(1)。《Perl 程式設計》第五章。你可能需要 perlbot(1)，因為它包含許多物件導向的技巧。

《Perl 高等程式設計》第七章、第八章有關物件導向程式設計的討論，描述前人遭遇的 Perl 物件問題。

13.1 建立一個物件

問題

想為你的使用者建立一個產生新物件的方法。

解答

製作一個建構函式。Perl 的建構函式不但初始化它的物件，且會為它配置記憶體（以匿名雜湊的方式）。另一方面，C++ 建構函式被呼叫時，記憶體已經為其被配置好了。其餘物件導向的語言可能會將 C++ 的建構函式稱為初始函式（initializer）。

這是 Perl 的標準物件建構函式：

```
sub new {
    my $class = shift;
    my $self  = { };
    bless($self, $class);
    return $self;
}
```

這是和上列敘述同義的寫法：

```
sub new { bless( { }, shift ) }
```

討論

任何成員函式都會像建構函式的運作一樣，去配置和初始化一個新物件。有一件最重要的事是，在被 bless 之前，參考並不是一個物件。下面是一個最簡單的建構函式，雖然不一定特別有用：

```
sub new { bless({}) }
```

加入一些初始值：

```
sub new {
    my $self = { };  # 配置匿名雜湊
    bless($self);
    # 初始化兩個樣本屬性/資料成員/欄位
    $self->{START} = time();
    $self->{AGE}   = 0;
    return $self;
}
```

這個建構函式使用只有一個引數的 bless 函式，所以並不是非常有用，這種狀況下一定是把物件 bless 到目前的套件。這表示它不能有效地被繼承，它建立的物件一定是被 bless 到 new 函式被編譯進去的類別之中。在繼承的情況下，此類別不一定要是被引用的成員函式所屬的類別。

要解決這問題，令建構函式專注於它的第一個引數。對類別成員函式而言，這是個套件名稱，將它當作第二個引數傳遞給 bless：

```
sub new {
    my $classname  = shift;         # 我們要建構的類別
    my $self      = {};             # 配置新的記憶體
    bless($obref, $classname);      # 標記正確的型態
    $self->{START}  = time();       # 初始化資料欄位
    $self->{AGE}    = 0;
    return $self;                   # 再傳送回去
}
```

這樣，建構函式就能正確地被子類別繼承。

你可能想要將記憶體配置和 bless 的步驟從實體資料初始化的步驟中分離出來。簡單的類別不需要這個動作，但它會讓繼承變得更容易；請參考章節 13.10。

```
sub new {
    my $classname  = shift;         # 我們要建構的類別
    my $self       = {};            # 配置新的記憶體
    bless($self, $classname);       # 標記正確的型態
    $self->_init(@_);               # 以剩下的引數去呼叫 _init
    return $self;
}

# 將成員函式變為初始化欄位「私有」的,
# 這會將 START 設定為目前的時間, 將 AGE 設為 0,
# 若以引數呼叫, _init 會把引數當作是初始化物件的 key+value 對。
sub _init {
    my $self = shift;
    $self->{START} = time();
    $self->{AGE}   = 0;
    if (@_) {
        my %extra = @_;
        @$self{keys %extra} = values %extra;
    }
}
```

相關資料

perltoot(1)、perlobj(1)。《Perl 程式設計》第五章。本書章節 13.6、13.9、13.10。

13.2 清除一個物件

問題

當物件已經不再被使用的情況下，你想要執行特別的程式碼。這種情況是會出現的，因為物件有時是與外在世界的連繫介面 —— 或包含循環的資料結構 —— 且在執行完之後必須被清除。你必須移除暫存的檔案、切斷循環的鏈結、切斷與基座的連接、或殺掉大量產生的副行程（subprocess）。

解答

建立一個名為 DESTORY 的成員函式。當不再有指向物件的參考或系統關機時，它都將會被呼叫。你不須做任何記憶體回收的動作，交給結束程式碼（finalization code）去做就好了，如果它對類別是有意義的。

```
sub DESTROY {
    my $self = shift;
    printf("$self dying at %s\n", scalar localtime);
}
```

討論

每個敘述都有開始和結束。物件敘述的開始是它的建構函式，當物件存在時會被明確地呼叫。它結束時的敘述是解構函式，當一個物件結束執行時，解構函式就會被呼叫。任何物件的清除碼（clean-up code）都被放在該物件的解構函式中，這個解構函式的名稱就是 DESTROY。

為什麼解構函式不能有隨意的名字？因為建構函式是以名稱被呼叫，而解構函式則不是。經由 Perl 的資源回收（GC）系統，清除動作會自動發生。為了知道呼叫了什麼東西，Perl 堅持解構函式的名稱是 DESTROY。若同時有一個以上的物件結束，Perl 不保證以何種順序呼叫解構函式。

為什麼 DESTROY 都是大寫字母？在 Perl 中，完全是大寫字母的函式名稱，表示這個函式會被 Perl 自動地呼叫。除了 DESTROY，還有 BEGIN、END、AUTOLOAD，以及被繫結的物件所使用的成員函式（請參考章節 13.15），如 STORE、FETCH。

使用者並不關心解構函式什麼時候被呼叫。在沒有任何 GC 格式的語言當中，這是靠不住的，所以程式設計者必須呼叫解構函式去清除記憶體和狀態，且要知道去做這件事的正確時間，這是件很艱鉅的任務。

由於 Perl 會自動做記憶體管理，在 Perl 裡的物件解構函式幾乎不會用到。即使用到了，這個呼叫的動作不只是多餘，且是危險的。當物件不再被使用時，解構函式會被執行時期系統呼叫。因為 Perl 會處理單純的問題，例如記憶體回收，所以大部分的類別不需要解構函式。

Perl 裡以參考為主的資源回收系統唯一沒有為你做的工作，是當你的資料結構有循環性的情況時，例如：

```
$self->{WHATEVER} = $self;
```

在這個情況下，如果期望程式不會浪費掉記憶體，你必須手動刪除自我參考（self-reference）。在這種允許「也許會導致錯誤」的情況下，這是我們能採取的最好的做法；章節 13.13 提供這個問題一個不錯的解答。儘管如此，還要確保當程式完成時，它的物件解構函式會適當地被呼叫。在直譯器停止時，GC 的清除指令會馬上被執行。任何物件都無法逃過最後被清除的命運。除非程式永不跳出，否則你要保證物件最後都能夠被清除。如果正執行嵌入（embed）其他應用程式的 Perl，第二個 GC 指令會很頻繁地發生 —— 在每回直譯器停止時。

若一個程式經由 exec 呼叫而跳出，DESTROY 不會被呼叫。

相關資料

perltoot(1)、perlobj(1)。《Perl 程式設計》第五章「關於垃圾回收（資源回收）」的部分。本書章節 13.10、13.13。

13.3 處理實體資料

問題

物件內的每個資料屬性，有時候稱為資料成員（member）或特質（property），在存取時需要它們自己的成員函式。如何寫這些函式去操作這些物件的實體資料（instance data）呢？

解答

解決方法有兩種：一是寫取得和設定成員函式來改變物件雜湊內適當的 key，如下：

```
sub get_name {
    my $self = shift;
    return $self->{NAME};
}

sub set_name {
    my $self      = shift;
    $self->{NAME} = shift;
}
```

或者視它們是否被傳遞兩個引數，製作一個成員函式來完成這兩件（取得、設定）工作：

```
sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}
```

有時候設定一個新值的時候，傳回先前的值是很有用的：

```
sub age {
    my $self = shift;
    my $prev = $self->{AGE};
    if (@_) { $self->{AGE} = shift }
    return $prev;
}
# 呼叫取得與設定函式
$obj->age( 1 + $obj->age );
```

討論

成員函式是你用來實作物件公共介面（public interface）的工具。一個正確的類別不應支持任何人到它的內部閒逛。每個資料屬性有一個成員函式可以去更新、取回它、或兩者都有。如果使用者寫的程式碼如下：

```
$him = Person->new();
$him->{NAME} = "Sylvester";
$him->{AGE}  = 23;
```

就可以侵入介面，然後獲得他們要取得的資料。

對於私有的資料，最好不要提供可以存取它們的成員函式。

如果使用一個功能嚴謹的介面，你就可以自由地修改內部的表示法，而不需要害怕會破壞程式碼。這個介面允許你去執行任何範圍的檢查和處理任何資料的重新格式化或轉換。

這個特殊版本的 name 成員函式可以用來說明一切：

```
use Carp;
sub name {
    my $self = shift;
    return $self->{NAME} unless @_;
    local $_ = shift;
    croak "too many arguments" if @_;
    if ($^W) {
        /[^\s\w'-]/         && carp "funny characters in name";
        /\d/                && carp "numbers in name";
        /\S+(\s+\S+)+/      || carp "prefer multiword name";
        /\S/                || carp "name is blank";
    }
    s/(\w+)/\u\L$1/g;       # 強制第一個字母為大寫
    $self->{NAME} = $_;
}
```

若使用者或是其他繼承的類別，經由界面直接存取 NAME 欄位，你不能之後來再加上這一類的程式碼。若只使用間接的方式來存取全部的資料屬性，則你日後還有選擇性。

如果你習慣用 C++ 物件，你就會習慣從成員函式內，就像取得簡單變數一樣地取得物件的資料成員。CPAN 的 Alias 模組就提供這個功能，而且讓物件可以呼叫私有成員函式，但在類別外被分流（folk）出來的物件則不能。

以下是用 Alias 模組來建立 Person 的示範。當更新這些實體變數時，你會在雜湊中自動更新變數值所在的欄位。

```
package Person;

# 這和之前作的一樣...
sub new {
     my $that  = shift;
     my $class = ref($that) || $that;
     my $self = {
           NAME  => undef,
           AGE   => undef,
           PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
use vars qw($NAME $AGE $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return    $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return    $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return    @PEERS;
}

sub exclaim {
    my $self = attr shift;
```

```
    return sprintf "Hi, I'm %s, age %d, working with %s",
            $NAME, $AGE, join(", ", @PEERS);
}

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}
```

你需要使用 use var，因為 Alias 用相同的名稱當作欄位來操作套件全域變數。要在 use strict 在執行時使用全域變數，你必須預先宣告它們。這些變數是在包含 attr() 呼叫的程式碼區塊內的區域變數，就像使用 local 一樣。這表示它們仍然被看成擁有暫存值的套件全域變數。

相關資料

perltoot(1)、perlobj(1)、perlbot(1)。CPAN 裡 Alias 模組的說明文件。《Perl 程式設計》第五章。本書章節 13.11、13.12。

13.4 處理類別資料

問題

你需要一個成員函式可以被整個類別（而不只是一個物件）呼叫。這可能是程序上的要求，或者它可能是一個被類別裡所有實體分享的全域資料屬性。

解答

物件成員函式是以參考作為第一個引數，而類別成員函式則是以包含類別名稱的字串作為它的第一個引數。類別成員函式存取的是套件資料，而不是物件資料，就像這裡的 population 成員函式一樣：

```
package Person;

$Body_Count = 0;

sub population { return $Body_Count }

sub new {                                   # 建構函式
    $Body_Count++;
    return bless({}, shift);
}

sub DESTROY { --$BodyCount }                # 解構函式

# 稍後，使用者就可以這樣用了：
package main;

for (1..10) { push @people, Person->new }
printf "There are %d people alive.\n", Person->population();
```

```
There are 10 people alive.
```

討論

每個物件本身存放著自己完整的狀態資訊。在物件內資料屬性的值和相同類別的其他實體的屬性值無關。例如設定 her 的性別，不會對 his 的性別做任何更改，因為他們是不同狀態的不同物件：

```
$him = Person->new();
$him->gender("male");

$her = Person->new();
$her->gender("female");
```

有些程式設計者喜歡使用字首大寫的全域變數，有些則偏好在成員函式影響類別資料而不是實體資料的時候，使用字首大寫字母的名稱。這有一個使用類別成員函式 Max_Bound 的範例：

```
FixedArray->Max_Bounds(100);                # 此設定適用於整個類別
$alpha = FixedArray->new();
printf "Bound on alpha is %d\n", $alpha->Max_Bounds();
100
$beta = FixedArray->new();
$beta->Max_Bounds(50);                      # 此設定適用於整個類別
printf "Bound on alpha is %d\n", $alpha->Max_Bounds();
50
```

實作起來很簡單：

```
package FixedArray;
$Bounds = 7;  # 預設值
sub new { bless( {}, shift ) }
sub Max_Bounds {
    my $proto  = shift;
    $Bounds    = shift if @_;          # 允許更新
    return $Bounds;
}
```

要讓數值只能被讀取，只要移除更新的部份即可，如：

```
sub Max_Bounds { $Bounds }
```

如果你非常堅持，那你可以只讓 $Bound 私有於包含類別的檔案範圍內的區域變數，這樣一來就沒有人能夠用 $FixedArray::Bounds 去找到它的值，這些值必須經由成員函式介面才能被存取。

這有個技巧可以建立可調整的類別：把物件資料存放在物件的名稱空間，把類別資料存放在類別的名稱空間（套件變數或檔案範圍的區域變數）。只有類別成員函式可以直接存取類別內的屬性。物件成員函式應該只能存取實體資料。如果物件成員函式需要存取類別資料，它的建構函式應該把指向該資料的參考存放在物件中。例如：

```
sub new {
    my $class = shift;
    my $self = bless({}, $class);
    $self->{Max_Bounds_ref} = \$Bounds;
    return $self;
}
```

相關資料

perltoot(1)、perlobj(1)、perlbot(1)。《Perl 程式設計》第五章「類別內容與物件」小節。本書章節 13.3，章節 13.14 的範例「溢載的 FixNum 類別」之的 places 成員函式。

13.5 把類別當作結構使用

問題

你習慣用比 Perl 的陣列和雜湊更複雜的結構性資料型態，如 C 的結構（struct）或 Pascal 的記錄（record）。你聽說 Perl 的類別可以媲美它們，但你又不是一個物件導向程式設計師。

解答

使用 Class::Struct 模組去宣告類似 C 的結構：

```
use Class::Struct;          # 載入建立結構的模組

struct Person => {          # 為 "Person" 定義
    name   => '$',          #     name 欄位為純量
    age    => '$',          #     age 欄位為純量
    peers  => '@',          #     peers 欄位為陣列(參考)
};

my $p = Person->new();      # 配置空的 Person 結構

$p->name("Jason Smythe");                   # 設定 name 欄位
$p->age(13);                                # 設定 age 欄位
$p->peers( ["Wilbur", "Ralph", "Fred" ] );  # 設定 peers 欄位

# 或者你也可以這樣做：
```

```
@{$p->peers} = ("Wilbur", "Ralph", "Fred");

# 取得不同的值，包括第零位的朋友
printf "At age %d, %s's first friend is %s.\n",
    $p->age, $p->name, $p->peers(0);
```

討論

Class::Struct::struct 函式可以快速建立類似結構的類別。它以第一個引數為名創造類別，並給這個類別一個名為 new 的建構函式和每個欄位的資料存取函式。

在結構的設計定義上，key 是欄位的名稱，值（value）則是資料型態（type）。型態可以是三個基本型態的其中一種：'$' 表示數量，'@'表示陣列，'%' 表示雜湊。每個存取函式（accessor）被呼叫時可以沒有引數（可以將目前的值取回），或是用一個引數（可以設定值）。欄位型態是雜湊或陣列時，沒有引數的成員函式呼叫會傳回指向陣列或雜湊的參考，一個引數的呼叫會取回該欄位的值【註】，兩個引數的呼叫則會設定該欄位的值。

型態甚至可以是另一個具名結構的名稱 —— 或是任何類別 —— 此具名結構提供一個名為 new 的建構函式。

```
use Class::Struct;

struct Person => {name => '$',      age  => '$'};
struct Family => {head => 'Person', address => '$', members => '@'};

$folks  = Family->new();
$dad    = $folks->head;
$dad->name("John");
$dad->age(34);

printf("%s's age is %d\n", $folks->head->name, $folks->head->age);
```

註　　但是，若它是一個參考，則會被當作一個新的、可型態檢查的集合體使用。

如果你想要對欄位值做更多參數檢查，就寫一個你自己的存取函式去覆蓋掉預設的存取成員函式版本。假設你要確定年齡值只包含數字，而且要落在合理的人類年齡範圍內。你可能會寫一個像這樣的函式：

```
sub Person::age {
    use Carp;
    my ($self, $age) = @_;
    if    (@_  > 2) {  confess "too many arguments" }
    elsif (@_ == 1) {  return $struct->{'age'}      }
    elsif (@_ == 2) {
        carp "age `$age' isn't numeric"   if $age !~ /^\d+/;
        carp "age `$age' is unreasonable" if $age > 150;
        $self->{'age'} = $age;
    }
}
```

如果只想在 -w 命令列旗標被使用時提供警告，可以檢查 $^W 變數：

```
if ($^W) {
    carp "age `$age' isn't numeric"   if $age !~ /^\d+/;
    carp "age `$age' is unreasonable" if $age > 150;
}
```

如果你使用了 -w，而且想要讓使用者自行決定是否要顯示警告訊息，做法如下。不要被指標箭頭弄糊塗了，它是間接的函式呼叫，不是成員函式呼叫。

```
my $gripe = $^W ? \&carp : \&croak;
$gripe->("age `$age' isn't numeric")   if $age !~ /^\d+/;
$gripe->("age `$age' is unreasonable") if $age > 150;
```

在內部，類別是以雜湊來實作，就如同大部分的類別一樣。這使得程式碼容易被處理和除錯，想想看在除錯器（debugger）列印結構的效果，你就會明白了。Class::Struct 模組也提供陣列的表示法，只要指定方括號（而不是大括號 {}）內的欄位就可以了：

```
struct Family => [head => 'Person', address => '$', members => '@'];
```

根據經驗，建議你選擇陣列的表示法來代替雜湊，這可以降低物件 10% 到 50% 的記憶體使用量，提高 33% 的存取時間。這個方式的代價是除錯訊息會比較少，而且在寫一個覆蓋函式時，例如上面的 Person::age，會有更多內在系統資源的消耗。

如果選擇將物件用陣列來表示，將會使它難以使用繼承的動作。在這裡倒不是重點，因為 C 式的結構反而讓你更容易了解集合（aggregation）的觀念。

Perl 5.005 版的 use fields pragma 提供以雜湊表達陣列的速度和空間，以及在編譯時期的物件欄位名稱檢查。

如果所有的欄位都是同一個型態，不需要這麼寫：

```
struct Card => {
    name    => '$',
    color   => '$',
    cost    => '$',
    type    => '$',
    release => '$',
    text    => '$',
};
```

可以使用 map 去縮短它：

```
struct Card => { map { $_ => '$'} qw(name color cost type release text) } ;
```

如果你是個喜歡把型態加在檔案名稱前的 C 程式設計師，你只要倒轉它們的順序，就可以產生下列的結果：

```
struct hostent => { reverse qw{
    $ name
    @ aliases
    $ addrtype
    $ length
    @ addr_list
}};
```

你甚至可以用 #define 的精神來製作別名，這允許你在多個別名下存取相同的欄位資料。在 C 可以宣告：

```
#define h_type h_addrtype
#define h_addr h_addr_list[0]
```

在 Perl 可以試試下面的方式：

```
# 製造 (hostent object)->type() 與 (hostent object)->addrtype()
*hostent::type = \&hostent::addrtype;

# 製造 (hostenv object)->addr() 與(hostenv object)->addr_list(0)
sub hostent::addr { shift->addr_list(0,@_) }
```

如你所看到的，你可以簡單地將成員函式加入類別，或將函式加入套件，只要在正確的名稱空間宣告一個副常式就可以，不一定要在定義、或衍生該類別的檔案內，或作任何困難複雜的事情。不過，衍生它會比較容易一點：

```
package Extra::hostent;
use Net::hostent;
@ISA = qw(hostent);
sub addr { shift->addr_list(0,@_) }
1;
```

這可在標準 Net::hostent 類別中找到，一點也不費事。你可以看看這個模組的原始程式碼，也許你會得到不同的啟發。

相關資料

perltoot(1)、perlobj(1)、perlbot(1)。標準 Class::Struct 模組的說明文件。標準 Net::hostent 模組的原始程式碼。CPAN 的 Alias 模組的說明文件。本書章節 13.3。

13.6 複製物件

問題

想要寫一個可以在已存在物件上被呼叫的建構函式。

解答

像這樣開始你的建構函式：

```
my $proto  = shift;
my $class  = ref($proto) || $proto;
my $parent = ref($proto) && $proto;
```

$class 變數將會包含作為 bless 目的地的類別。$parent 變數會是 false，或是你所複製（clone）的物件。

討論

有時候你需要與目前物件相同型態的另一個物件，你可以這樣做：

```
$ob1 = SomeClass->new();
# 一些操作後
$ob2 = (ref $ob1)->new();
```

但這不是非常清楚明確，若有個建構函式可以在類別或已存在的物件上被呼叫就會比較清楚了。此建構函式若為類別成員函式（即它在類別被呼叫）會傳回一個以預設初始化的新物件，若為實體成員函式（在物件被呼叫）會傳回一個新物件，這個新物件是在它被呼叫的物件內被初始化：

```
$ob1 = Widget->new();
$ob2 = $ob1->new();
```

下面這個版本的 new 會把這情形列入考慮：

```
sub new {
    my $proto  = shift;
    my $class  = ref($proto) || $proto;
    my $parent = ref($proto) && $proto;

    my $self;
    # 檢查是否會遮蔽 new，使 @ISA 看不到
    if (@ISA && $proto->SUPER::can('new')) {
        $self = $proto->SUPER::new(@_);
    } else {
        $self = {};
        bless ($self, $proto);
    }
```

```
    bless($self, $class);

    $self->{PARENT}  = $parent;
    $self->{START}   = time();   # 初始化資料欄位
    $self->{AGE}     = 0;
    return $self;
}
```

初始化不意味著只是簡單地從來源複製數值。如果你寫了一個鏈結串列（linked list）或是二進位樹類別，當建構函式以實體成員函式被呼叫的時候，它可以傳回一個鏈結到串列和樹的新物件。

相關資料

perlobj(1)。《Perl 程式設計》第五章。本書章節 13.1、13.9、13.13。

13.7 間接地呼叫成員函式

問題

如何呼叫在執行時期才會知道其名稱的成員函式。

解答

將成員函式名稱以字串存放在純量變數內，然後在要使用真正的成員函式名稱的地方，將之放在箭頭運算子的右邊來使用它：

```
$methname = "flicker";
$obj->$methname(10);         # 呼叫 $ob->flicker(10);

# 於物件上依名稱呼叫三個成員函式
foreach $m ( qw(start run stop) ) {
    $obj->$m();
}
```

討論

有時候你需要呼叫的成員函式的名稱已經存放在某個地方，你沒辦法得到那個成員函式的位址（adress），但可以儲存它的名稱。如果有一個純量變數 $meth 包含成員函式的名稱，你可以在物件 $crystal 上，以 $crystal->$meth 呼叫那個成員函式。

```
@methods = qw(name rank serno);
%his_info = map { $_ => $ob->$_() } @methods;

# 等於這個：

%his_info = (
    'name'  => $ob->name(),
    'rank'  => $ob->rank(),
    'serno' => $ob->serno(),
);
```

如果你想不出取得成員函式位址的方法，應該試著重新思考演算法。例如，不要用 \$ob->method()（因為反斜線會直接被對應到成員函式的傳回值或數值上），而改用：

```
my $fnref = sub { $ob->method(@_) };
```

現在，間接呼叫它的時候可以使用：

```
$fnref->(10, "fred");
```

然後就可以正確真正地呼叫：

```
$obj->method(10, "fred");
```

即使 $obj 已經超過範圍也可以正常運作。這個做法是比較明確的。

被 UNIVERSAL can() 傳回的程式碼參考也許不能像是個間接的成員函式呼叫一樣被使用，因為當用在一個任意類別的物件時，這不一定是個合法的成員函式。

例如，這段程式碼就很模稜兩可：

```
$obj->can('method_name')->($obj_target, @arguments)
   if $obj_target->isa( ref $obj );
```

這問題在於被 can 傳回的程式碼參考，它可能不是一個在 $obj_target 可以被呼叫的合法成員函式。最安全的作法，可能是只以布林表示式測試 can() 成員函式。

相關資料

perlobj(1)。本書章節 11.8。

13.8 決定子類別的成員關係

問題

想要知道一個物件是否是某個類別的實體或是該類別的子類別。也許你想要決定某個成員函式是否可以在某個物件上被呼叫。

解答

使用 UNIVERSAL 類別的成員函式：

```
$obj->isa("HTTP::Message");                 # 為物件成員函式
HTTP::Response->isa("HTTP::Message");       # 為類別成員函式

if ($obj->can("method_name")) { .... }      # 檢查成員函式的正確性
```

討論

假如所有的物件追根究底都是來自同一個類別，會不會比較方便？這樣一來就可以給每個物件同一個成員函式，而不用把它加到個別的 @ISA內。雖然你看不到它，但 Perl 會把套件 UNIVERAL 當成是在 @ISA 末端一個額外的元素。

在 5.003 版，沒有成員函式被預先定義在 UNIVERSAL 內，但你可以把想要用的成員函式放進去。然而在 5.004 版，UNIVERSAL 已經有一些成員函式在裡面。這被建立在 Perl 二進位程式碼內，所以它們不需要額外的時間去載入。預先定義的成員函式包括 isa、can、和 VERSION。isa 成員函式告訴你一個物件或類別是否「是」另一個，而不需要自己去詳查階層：

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

通常試試成員函式呼叫是最好的，但這樣的檢查有時讓人覺得很麻煩，因為太受限制了。

can 成員函式代表物件或類別被呼叫，會回報它的字串引數是否是在該類別內可以呼叫的成員函式。事實上它傳回的是一個指向成員函式的函式參考：

```
$his_print_method = $obj->can('as_string');
```

最後 VERSION 成員函式會檢查類別（或物件的類別）是否有一個名為 $VERSION 的套件全域變數，例如這裡的敘述：

```
Some_Module->VERSION(3.0);
$his_vers = $obj->VERSION();
```

通常我們不會呼叫 VERSION。記住在 Perl 當中，一個全是大寫字母的函式名稱，意味著這函式會被 Perl 自動呼叫。在這個例子中，當你用了以下的敘述，Perl 就會自動呼叫 VERSION：

```
use Some_Module 3.0;
```

如果想將版本檢查加入先前所說的 Person 類別，可以把下面的敘述加到 Person.pm 當中：

```
use vars qw($VERSION);
$VERSION = '1.01';
```

然後在使用者程式碼中宣告 use Person 1.01;，以確保你載入的版本為 1.01 或更高的版本。這和載入正確的版本數字是不同的，它只是至少要有那麼高的版本。很遺憾地，目前還沒有支援可同時安裝模組的多個版本。

相關資料

perlfunc(1)。標準 UNIVERSAL 模組的說明文件。《Perl 程式設計》第三章的 use 關鍵字。

13.9 寫一個可被繼承的類別

問題

你不確定已經設計的類別是否嚴謹地可以被繼承。

解答

在類別當中使用「空子類別測試」。

討論

假設已經完成一個名為 Person 的類別，這類別有個建構函式 new，以及 age、name 等成員函式。你可以這麼實作它：

```
package Person;
sub new {
    my $class = shift;
    my $self  = { };
    return bless $self, $class;
}
sub name {
    my $self = shift;
    $self->{NAME} = shift if @_;
    return $self->{NAME};
}
sub age {
```

```
    my $self = shift;
    $self->{AGE} = shift if @_;
    return $self->{AGE};
}
```

可以像這樣使用該類別：

```
use Person;
my $dude = Person->new();
$dude->name("Jason");
$dude->age(23);
printf "%s is age %d.\n", $dude->name, $dude->age;
```

現在看到另一個類別 Employee：

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

很簡短，只是載入 Person 類別，並指定 Employee 繼承所有 Person 的成員函式。由於 Employee 自己並沒有成員函式，所有它的成員函都是來自 Person 類別，Employee 的表現就像是 Person 一樣。

設定一個像這樣的空類別被稱作空基底類別測試法（empty base class test），即建立一個衍生類別，這個類別只是單純繼承基底類別，不具其他功能。如果原來的基底類別設計得當，新的衍生類別才可以被使用，就像是原來類別的替身。這表示之前的程式碼如果只是類別的名稱改變了，一切仍然能夠正常運作：

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

所謂設計得當，是說我們只使用兩個引數的 bless、避免類別物件資料被直接存取、而且不匯出任何東西。在之前定義的 Person::new() 函式當中，我們小心地照這樣做。我們在建構函式中使用一些套件資料，但指向這的參考被存放在物件本身，其他成員函式存取套件資料就會經由這個參考，所以應該沒有問題。

為什麼我們說 Person::new()「函式」，它實際上不是個成員函式嗎？成員函式只是一個函式，而這個函式要求的第一個引數是類別名稱（套件）或者是物件（bless 的參考）。Person::new 函式是 Person->new 成員函式和 Employee->new 成員函式兩者最後呼叫的函式。雖然成員函式呼叫和函式呼叫非常像，但他們是不一樣的。如果你把它們當作是相同的，很快地你的程式就會什麼功能都沒有，而只剩下一個殘破的空殼。不同之處在於：第一，實際上它們的呼叫規則是不同的，成員函式呼叫會多取得一個引數；第二，函式呼叫沒有繼承的動作，但成員函式呼叫有。

成員函式呼叫	函式呼叫
Person->new()	Person::new("Person")
Employee->new()	Person::new("Employee")

如果你習慣這樣呼叫：

```
$him = Person::new();              # 錯!
```

你將會遇上難以捉摸的問題，因為函式沒有取得 Person 的引數，所以它不能被 bless 到被傳遞進來的類別中。更糟的是你可能想要試著去呼叫 Employee::new()，但沒有這個函式！這是個對繼承的成員函式的呼叫。

所以打算呼叫一個成員函式時，不要使用函式呼叫。

相關資料

perltoot(1)、perlobj(1)、perlbot(1)。《Perl 程式設計》第五章。本書章節 13.1、13.10。

13.10 存取被覆蓋的成員函式

問題

你的建構函式覆蓋了超類別的建構函式，但你想呼叫超類別原來的建構函式。

解答

使用 SUPER 類別。

```
sub meth {
    my $self = shift;
    $self->SUPER::meth();
}
```

討論

在類似 C++ 的程式語言裡，建構函式沒有實際配置記憶體但會初始化物件，所有基底類別的建構函式會自動被呼叫。在類似 Java 和 Perl 的語言，你則需要自己去呼叫它們。

在特定的類別中呼叫成員函式，可以用 $self->SUPER::meth()。它會在特定的基底類別中尋找，而且只在被覆蓋的成員函式（overriden method）內是有效的。比較這三個呼叫：

```
$self->meth();              # 呼叫第一個找到的 meth()
$self->Where::meth();       # 呼叫 Where 套件中的 meth()
$self->SUPER::meth();       # 呼叫被覆蓋的 meth()
```

一般的使用者應該限制他們只能使用第一種方式；第二種方式？也許可以，但不建議使用；最後一個則必須要在被覆蓋的成員函式內才能用。

覆蓋別人的建構函式應該要呼叫它的超類別原來的建構函式去配置和 bless 物件，而它自己則只負責初始化資料欄位。將配置物件的程式碼與初始化物件的程式碼分開來是很合理的，我們會以一個底線開頭的名稱來命名它，慣例上這表示一個私有的成員函式。你可以把它想成是「請勿打擾」的標示。

```
sub new {
    my $classname  = shift;         # 我們要建立的類別
    my $self       = $classname->SUPER::new(@_);
    $self->_init(@_);
    return $self;                   # 並傳送回去
}

sub _init {
```

```
    my $self = shift;
    $self->{START}   = time();   # 初始化資料欄位
    $self->{AGE}     = 0;
    $self->{EXTRA}   = { @_ };   # 所有其它的東西
```

SUPER::new 和 _init 都已經被任何一個剩下的引數呼叫。這方式讓使用者能夠傳遞其他的欄位起始值進來，如：

```
$obj = Widget->new( haircolor => red, freckles => 121 );
```

至於這些使用者的參數要不要存放在他們自己的雜湊中，則由你決定。

注意，SUPER 只能在第一個被覆蓋的成員函式運作。如果 @ISA 陣列中有好幾個類別，它只會取得第一個。手動操作 @ISA 也是可以的，但實在沒有必要這麼麻煩。

```
my $self = bless {}, $class;
for my $class (@ISA) {
    my $meth = $class . "::_init";
    $self->$meth(@_) if $class->can("_init");
}
```

這個程式碼假定全部的超類別是以 _init 來初始化它們的物件，而不是在建構函式內，它也假定雜湊參考只是被它的物件使用。

相關資料

perltoot(1)、perlobj(1) 當中 SUPER 類別的討論。《Perl 程式設計》第五章「執行方法（成員函式）」的部份。

13.11 使用 AUTOLOAD 產生屬性成員函式

問題

你的物件需要存取函式（accessor method）去設定或取得它自己的資料欄位，但你不想每一次都要把它們全部寫出來。

解答

將 Perl 的 AUTOLOAD 成員函式當成代理成員函式產生器（proxy method generator）一樣地小心使用，這樣你每次想要加入一個新的資料欄位時，就不用自己去創造它們。

討論

Perl 的 AUTOLOAD 機制會攔截所有可能尚未定義的成員函式呼叫。為了檢查不被允許的資料名稱，我們會把被允許欄位的串列存放在雜湊中。AUTOLOAD 會去查證被存取的欄位在那個雜湊中。

```
package Person;
use strict;
use Carp;
use vars qw($AUTOLOAD %ok_field);

# 授權四個屬性欄位
for my $attr ( qw(name age peers parent) ) { $ok_field{$attr}++; }

sub AUTOLOAD {
    my $self = shift;
    my $attr = $AUTOLOAD;
    $attr =~ s/.*::/;
    return unless $attr =~ /[^A-Z]/;  # 跳過 DESTROY 和所有大寫的成員函式
    croak "invalid attribute method: ->$attr()" unless $ok_field{$attr};
    $self->{uc $attr} = shift if @_;
    return $self->{uc $attr};
}
sub new {
    my $proto  = shift;
    my $class  = ref($proto) || $proto;
    my $parent = ref($proto) && $proto;
    my $self = {};
    bless($self, $class);
    $self->parent($parent);
    return $self;
}
1;
```

這個類別支援名為 new 的建構函式，和四個屬性成員函式：name、age、peers、和 parent。使用這個模組：

```
use Person;
my ($dad, $kid);
$dad = Person->new;
$dad->name("Jason");
$dad->age(23);
$kid = $dad->new;
$kid->name("Rachel");
$kid->age(2);
printf "Kid's parent is %s\n", $kid->parent->name;
Kid's parent is Jason
```

在製造繼承樹的時候，這會變得難以處理。假定你想要建立一個 Employee 類別，它包含 Person 類別的每個資料屬性，再加上兩個新的屬性：salary 和 boss。Employee 類別不能依賴繼承而來的 Person::AUTOLOAD 來決定 Employee 的屬性成員函式，所以每個類別需要有自己的 AUTOLOAD 函式。它只會檢查類別的已知屬性欄位，但發生錯誤時，它會呼叫被覆蓋的超類別版本。

例如，請看這個版本的 AUTOLOAD：

```
sub AUTOLOAD {
    my $self = shift;
    my $attr = $AUTOLOAD;
    $attr =~ s/.*::///;
    return if $attr eq 'DESTROY';

    if ($ok_field{$attr}) {
        $self->{uc $attr} = shift if @_;
        return $self->{uc $attr};
    } else {
        my $superior = "SUPER::$attr";
        $self->$superior(@_);
    }
}
```

如果屬性不在我們的合法名單上，我們會把它傳遞給我們的上層結構，希望它可以解決這個問題。但你不能繼承這個 AUTOLOAD；每個類別必需有自己的 AUTOLOAD，因為不經由物件直接存取類別資料是不明智的。

甚至更糟的情況是，如果 A 類別繼承 B 和 C 兩個類別，這兩者都定義了自己的 AUTOLOAD，一個未定義的成員函式在 A 被呼叫，這動作將只會選中兩個父類別的其中一個。

我們可以試著來解決這方面的限制，但 AUTOLOAD 最後會像是七拼八湊的函式。對於更複雜的情況其實有更好的成員函式。

相關資料

perltoot(1) 中使用 AUTOLOAD 的範例。《Perl 程式設計》第五章。本書上冊章節 10.15、下冊章節 13.12。

13.12 解決資料繼承的問題

問題

你想繼承一個已經存在的類別，並加入一些成員函式以增加它的功能，但不知道超類別使用哪些資料欄位。要如何在不破壞上層類別的前提下，在物件雜湊裡安全地創造自己的名稱空間？

解答

在你自己的欄位名稱前加上你的類別名稱和分隔符號，例如一個底線或兩個。

討論

在一般 Perl OO 策略下，存在一個令人苦惱的問題，就是類別真正的表示法必須公開，這跟必須保持抽象化的原則相抵觸。這使得子類別必須用很不自然的方式來遞迴地跟超類別溝通。

假定我們是喜歡使用物件導向語言的使用者，並且每個人總是使用雜湊來存放物件，這樣雖然規避了類別使用陣列表示法但繼承自使用雜湊模型類別的問題（這個問題的解答是用集合（aggregation）和委派的觀念，如 perlbot(1) 所述）。即使如此，一個繼承的類別也不能在雜湊安全地使用一個 key。即使我們同意只使用成員函式呼叫去存取我們不能設定的屬性，要如何知道我們設定的不是超類別正在使用的 key？設想你要去使用一個 count 欄位，但你不知道你的更上層的類別是不是正在使用相同的東西。如果上層類別正嘗試相同的動作，你使用 _count 去指定名義上的私有性也是沒有幫助的。

合理的方法是在你自己的資料成員前加上套件名稱。因此如果 Employee 類別想要一個 age 欄位，為了安全的理由，你應該使用 Employee_age。這有一個存取函式的範例：

```
sub Employee::age {
    my $self = shift;
    $self->{Employee_age} = shift if @_;
    return $self->{Employee_age};
}
```

根據章節 13.5 敘述的 Class::Struct 模組的精神，這裡有一個對於這問題更嚴謹的解答。試想一個檔案有下列的敘述：

```
package Person;
use Class::Attributes;  # 請見下面說明
mkattr qw(name age peers parent);
```

以及這個敘述：

```
package Employee;
@ISA = qw(Person);
use Class::Attributes;
mkattr qw(salary age boss);
```

注意到它們兩者都有一個 age 屬性了嗎？如果它們被邏輯上分開，那我們就不能使用 $self->{age}，即使是我們自己在模組內！這個 Class::Attributes::mkattr 函式解決了這個問題：

```
package Class::Attributes;
use strict;
use Carp;
use Exporter ();
use vars qw(@ISA @EXPORT);
@ISA = qw(Exporter);
@EXPORT = qw(mkattr);
sub mkattr {
    my $hispack = caller();
    for my $attr (@_) {
        my($field, $method);
        $method = "${hispack}::$attr";
        ($field = $method) =~ s/:/_/g;
        no strict 'refs';
        *$method = sub {
            my $self = shift;
            confess "too many arguments" if @_ > 1;
            $self->{$field} = shift if @_;
            return $self->{$field};
        };
    }
}
1;
```

這麼一來 $self->{Person_age} 和 $self->{Employee_age} 還是分開的。有趣的是 $obj->age 只會取得第一個。現在可以寫 $obj->Person::age 和 $obj->Employee::age 以茲區別，但除了在極度要求的情況下，一個好的 Perl 程式碼不應該使用雙冒號去詳細說明一個正確的套件。如果真的被迫得這樣去做，或許該函式庫可以被設計地更好。

如果不想要用這個方式去寫它，可以另外從 Person 類別的內部，只要使用 age($self) 敘述，將可以取得 Person 的版本，若從 Employee 類別的內部，age($self) 可以取得 Employee 的版本。但那是因為它是個函式呼叫，而不是成員函式呼叫。

相關資料

Perl 5.005 的標準 use fields 和 use base pragma 的說明文件。本書上冊章節 10.14。

13.13 循環的資料結構

問題

有個自我參考的資料結構，當它不再被使用的時候，Perl 以參考為主的 GC 系統不會注意到這個情形，所以你要防止程式浪費了記憶體。

解答

創造一個非循環的容器（container）物件，這物件儲存一個指標，指向自我參考的資料結構。定義 DESTROY 成員函式，讓物件的類別能夠中斷自我參考的循環。

討論

許多有趣的資料結構都有包含指回它們自己的參考。例如下面的程式碼：

```
$node->{NEXT} = $node;
```

有了循環性，就可以逃過 Perl 的 GC 系統。當程式存在時，解構函式在程式跳出時會被呼叫，但有時候你可能不想等那麼久的時間才做這些事。

循環的鏈結串列和自我參考很相似。每個程式碼包含一個前端指標、一個後端指標和一個節點值。如果用參考來實作它，且沒有外部的參考指向它的節點時，會得到一連串循環的參考，則資料結構不會自動被 GC 系統收集。

讓每個節點都成為 Ring 類別內的一個實體不能解決這個問題。你希望 Perl 會將它清除掉（就像清除其它的資料結構那樣），只要你的物件是個結構，包含指向真正循環的參考，就可以如你所願。參考將會被存放在 DUMMY 欄位：

```
package Ring;

# 回傳空的環結構
sub new {
    my $class = shift;
    my $node  = { };
    $node->{NEXT} = $node->{PREV} = $node;
    my $self  = { DUMMY => $node, COUNT => 0 };
    bless $self, $class;
    return $self;
}
```

這是包含在環裡面的節點，而不是被傳回的環物件上。這表示類似下面的程式碼不會發生記憶體漏失（memory leak）的情形：

```
use Ring;

$COUNT = 1000;
for (1 .. 20) {
    my $r = Ring->new();
    for ($i = 0; $i < $COUNT; $i++) { $r->insert($i) }
}
```

即使有一千個結點，為每個結點建立二十個環，每個環都會在新的環被創造出來之前被拋除。比起釋放字串的記憶體，類別的使用者不再需要去釋放環的記憶體，因為這會自動地發生，就好像它的動作已經被指定了。

然而類別的執行者必須擁有相對於環的解構函式，它會自己刪除那些節點：

```
# 當 Ring 被清除時，破壞它所含的環結構
sub DESTROY {
    my $ring = shift;
    my $node;
    for ( $node  =  $ring->{DUMMY}->{NEXT};
          $node !=  $ring->{DUMMY};
          $node  =  $node->{NEXT} )
    {
            $ring->delete_node($node);
    }
```

```
    $node->{PREV} = $node->{NEXT} = undef;
}

# 刪除環結構的一個節點
sub delete_node {
    my ($ring, $node) = @_;
    $node->{PREV}->{NEXT} = $node->{NEXT};
    $node->{NEXT}->{PREV} = $node->{PREV};
    --$ring->{COUNT};
}
```

這有一些其它的成員函式，你可以用在你的 Ring 類別上。注意隱藏在物件內部的循環內，真正的工作如何擴展：

```
# $node = $ring->search( $value ) : 找出 $node 在環結構內的 $value
sub search {
    my ($ring, $value) = @_;
    my $node = $ring->{DUMMY}->{NEXT};
    while ($node != $ring->{DUMMY} && $node->{VALUE} != $value) {
          $node = $node->{NEXT};
    }
    return $node;
}

# $ring->insert( $value ) : 插入 $value 到環結構
sub insert_value {
    my ($ring, $value) = @_;
    my $node = { VALUE => $value };
    $node->{NEXT} = $ring->{DUMMY}->{NEXT};
    $ring->{DUMMY}->{NEXT}->{PREV} = $node;
    $ring->{DUMMY}->{NEXT} = $node;
    $node->{PREV} = $ring->{DUMMY};
    ++$ring->{COUNT};
}

# $ring->delete_value( $value ) : 依據值，由環結構中刪除節點
sub delete_value {
    my ($ring, $value) = @_;
```

```
    my $node = $ring->search($value);
    return if $node == $ring->{DUMMY};
    $ring->delete_node($node);
}

1;
```

相關資料

perlobj(1)。《Introduction to Algorithm》這本書（作者：Cormen、Leiserson、Rivest，由 MIT Press/McGraw-Hill 出版）的第 206~207 頁推導出的演算法。《Perl 程式設計》第五章「關於垃圾回收（資源回收）」的部分。

13.14 運算子溢載

問題

想要在一個類別內已經寫好的物件，使用熟悉的運算子，像是 == 或 +，或想要去為一個物件定義列印的插入（interpolation）。

解答

使用 use overload pragma，下面有對兩個最常用最有用的運算子做溢載的例子：

```
use overload ('<=>' => \&threeway_compare);
sub threeway_compare {
    my ($s1, $s2) = @_;
    return uc($s1->{NAME}) cmp uc($s2->{NAME});
}

use overload ( '""'  => \&stringify );
sub stringify {
    my $self = shift;
```

```
    return sprintf "%s (%05d)",
           ucfirst(lc($self->{NAME})),
           $self->{IDNUM};
}
```

討論

使用內建型態的時候，某些運算子會被使用，如 + 是「加」，. 是字串串接。使用 use overload pragma，可以自訂這些運算子，因此它們可以在物件內做一些特別的事情。

這個 pragma 需要一個運算子/函式呼叫的串列，例如：

```
package TimeNumber;
use overload '+' => \&my_plus,
             '-' => \&my_minus,
             '*' => \&my_star,
             '/' => \&my_slash;
```

藉著 TimeNumber 類別的物件，現在這四個運算子可以被使用，且串列中的函式可以被呼叫。這些函式可以做你希望作的任何事情。

這有一個 + 運算子的溢載範例，它用在一個包含小時、分、和秒的物件上。它假設兩個運算元都在一個類別內，這個類別有一個 new 成員函式可以像物件成員函式一樣被呼叫：

```
sub my_plus {
my($left, $right) = @_;
    my $answer = $left->new();
    $answer->{SECONDS} = $left->{SECONDS} + $right->{SECONDS};
    $answer->{MINUTES} = $left->{MINUTES} + $right->{MINUTES};
    $answer->{HOURS}   = $left->{HOURS}   + $right->{HOURS};

    if ($answer->{SECONDS} >= 60) {
        $answer->{SECONDS} %= 60;
        $answer->{MINUTES} ++;
    }

    if ($answer->{MINUTES} >= 60) {
```

```
        $answer->{MINUTES} %= 60;
        $answer->{HOURS}   ++;
    }

    return $answer;

}
```

當物件自己本身反映出數字結構時，例如複數、無窮數、向量、或矩陣，最好溢載數字運算子，否則程式碼會變得難以瞭解，而且會誤導使用者。想像一個類別塑造一個「區域」，如果可以把一個區域加到另一個裡面去，那難道不能從其中扣除一個區域嗎？如你所看到的，對於非數學的東西使用溢載的運算子是很奇怪的。

你可以使用 == 或 eq 去比較物件（任何參考），但這只能告訴你位址是否相同（使用 == 大概會比 eq 快十倍）。因為一個物件是高階的概念，而不是一個未處理的機器位址，你必須要定義要符合哪些條件兩個物件才算「相等」。

即使對於非數字的類別，也有兩個運算子：比較運算子和字串插入運算子是常常會被溢載的。<=> 和 cmp 運算子都可以被溢載，但前者被溢載的狀況比較常見。一旦 <=> 運算子被定義，你就可以使用 ==、!=、<、<=、>、>=。同樣地，如果溢載了 cmp，就能使用 lt、gt，和其他字串比較運算子。

字串插入運算子利用 " " 的名稱來運作，也就是兩個雙引號。每回字串轉換被呼叫時，這個運算子就會被觸發，例如位在雙引號或倒引號（`）內、或當它被傳遞給 print 函式時。

你可以閱讀 Perl 的 overload pragma 的說明文件。Perl 的運算子溢載有一些精巧的功能，例如：字串和數字轉換的成員函式、可自動產生你缺少的成員函式、必要時反轉（reverse）運算元，例如 5+$a 的 $a 是一個物件，就得做這樣的處理。

範例:溢載的 StrNum 類別

這兒有一個 StrNum 類別，它讓你可以將數字運算子用在字串上 —— 是的，我們正要做的是剛剛才建議你別這麼做的事，因為總有其他背景的程式設計師希望能在字串上運作 + 和 ==。這是運算子溢載一個簡單的示範，但平時因為執行的考量，我們肯定不會在程式中用上這樣的技巧。這也是一個建構函式與類別同名的有趣範例，這是 C++ 和 Python 程式設計師常會遇到的情況。

```
#!/usr/bin/perl
# show_strnum — 運算子溢載範例
use StrNum;

$x = StrNum("Red"); $y = StrNum("Black");
$z = $x + $y; $r = $z * 3;
print "values are $x, $y, $z, and $r\n";
print "$x is ", $x < $y ? "LT" : "GE", " $y\n";

values are Red, Black, RedBlack, and RedBlackRedBlackRedBlack
Red is GE Black
```

這個類別在範例 13-1。

範例 13-1. StrNum

```
package StrNum;

use Exporter ();
@ISA = 'Exporter';
@EXPORT = qw(StrNum);  # 不常見

use overload        (
        '<=>'   => \&spaceship,
        'cmp'   => \&spaceship,
        '""'    => \&stringify,
        'bool'  => \&boolify,
        '0+'    => \&numify,
        '+'     => \&concat,
        '*'     => \&repeat,
);

# 建構函式
sub StrNum {
    my ($value) = @_;
    return bless \$value;
}
```

```
sub stringify { ${ $_[0] } }
sub numify    { ${ $_[0] } }
sub boolify   { ${ $_[0] } }

# 溢載了 <=>，所以 <、== 等等都可以用
sub spaceship {
    my ($s1, $s2, $inverted) = @_;
    return $inverted ? $$s2 cmp $$s1 : $$s1 cmp $$s2;
}

# 這會用到 stringify
sub concat {
    my ($s1, $s2, $inverted) = @_;
    return StrNum ($inverted ? ($s2 . $s1) : ($s1 . $s2));
}

# 這會用到 stringify
sub repeat {
    my ($s1, $s2, $inverted) = @_;
    return StrNum ($inverted ? ($s2 x $s1) : ($s1 x $s2));
}
1;
```

範例:溢載的 FixNum 類別

這個類別使用運算子溢載去控制輸出的十進位數字。不過在操作上依然使用完整的精密度。places() 成員函式可以被使用在類別或特定的物件上，設定輸出的地方是在十進位小數點的右邊何處。

```
#!/usr/bin/perl
# demo_fixnum — 運算子溢載範例
use FixNum;

FixNum->places(5);

$x = FixNum->new(40);
$y = FixNum->new(12);
```

```
print "sum of $x and $y is ", $x + $y, "\n";
print "product of $x and $y is ", $x * $y, "\n";

$z = $x / $y;
printf "$z has %d places\n", $z->places;
$z->places(2) unless $z->places;
print "div of $x by $y is $z\n";
print "square of that is ", $z * $z, "\n";
```

```
sum of STRFixNum: 40 and STRFixNum: 12 is STRFixNum: 52
product of STRFixNum: 40 and STRFixNum: 12 is STRFixNum: 480
STRFixNum: 3 has 0 places
div of STRFixNum: 40 by STRFixNum: 12 is STRFixNum: 3.33
square of that is STRFixNum: 11.11
```

這個類別在範例 13-2。它只溢載加法、乘法、和除法這些數學運算子，其他則是空間關係（spaceship）運算子，用於處理比較、字串插入運算子、數字轉換運算子。為利於除錯，字串插入運算子做了特別的處理以便於辨認。

範例 13-2. FixNum

```
package FixNum;

use strict;

my $PLACES = 0;

sub new {
    my $proto   = shift;
    my $class   = ref($proto) || $proto;
    my $parent  = ref($proto) && $proto;

    my $v = shift;
    my $self = {
        VALUE  => $v,
        PLACES => undef,
    };
```

```
    if ($parent && defined $parent->{PLACES}) {
        $self->{PLACES} = $parent->{PLACES};
    } elsif ($v =~ /(\.\d*)/) {
        $self->{PLACES} = length($1) - 1;
    }  else {
        $self->{PLACES} = 0;
    }
    return bless $self, $class;
}

sub places {
    my $proto = shift;
    my $self  = ref($proto) && $proto;
    my $type  = ref($proto) || $proto;

    if (@_) {
        my $places = shift;
        ($self ? $self->{PLACES} : $PLACES) = $places;
    }
    return $self ? $self->{PLACES} : $PLACES;
}

sub _max { $_[0] > $_[1] ? $_[0] : $_[1] }
use overload '+'    => \&add,
             '*'    => \&multiply,
             '/'    => \&divide,
             '<=>'  => \&spaceship,
             '""'   => \&as_string,
             '0+'   => \&as_number;

sub add {
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} + $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}

sub multiply {
```

範例 13-2. FixNum（續）

```
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} * $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}

sub divide {
    my ($this, $that, $flipped) = @_;
    my $result = $this->new( $this->{VALUE} / $that->{VALUE} );
    $result->places( _max($this->{PLACES}, $that->{PLACES} ));
    return $result;
}

sub as_string {
    my $self = shift;
    return sprintf("STR%s: %.*f", ref($self),
        defined($self->{PLACES}) ? $self->{PLACES} : $PLACES,
            $self->{VALUE});
}

sub as_number {
    my $self = shift;
    return $self->{VALUE};
}

sub spaceship {
    my ($this, $that, $flipped) = @_;
    $this->{VALUE} <=> $that->{VALUE};
}

1;
```

相關資料

標準 use overload pragma 以及 Math::BigInt、Math::Complex 模組的說明文件。《Perl 程式設計》第七章。

13.15 使用 tie 創造特別的變數

問題

想要在變數或代碼上增加特別的程序。

解答

使用 tie 函式指定你的變數物件。

討論

任何一個人在 Perl 下曾經使用過 DBM 檔案都已經用過被繫結（tie）的物件。或許使用物件最聰明的方式是讓使用者不曾注意到它們。使用 tie 可以將變數或代碼連結到一個類別內，在這之後，某些名稱的物件成員函式對於被繫結的變數或代碼的存取動作將會被阻止。

最重要的 tie 成員函式有：FETCH 截斷讀的存取、STORE 截斷寫的存取、建構函式（TIESCALAR、TIEARRAY、TIEHASH、TIEHANDLE 其中一個）。

使用者的程式碼	執行的程式碼
tie $s, "SomeClass"	SomeCass->TIESCALAR()
$p = $s	$p = $obj->FETCH()
$s = 10	$obj->STORE(10)

$obj 從哪裡來呢？tie 會觸發類別的 TIESCALAR 建構函式的呼叫。Perl 儲存回傳的物件，並在後來的存取中暗中地使用它。

這有一個 tie 類別的範例，它實作一個數值環。每次變數被讀取，在環中的下一個值會被顯示出來。當它被寫入時，一個新值會被增加到環裡面去。這有一個範例：

```
#!/usr/bin/perl
# demo_valuering — 示範 tie 類別
use ValueRing;
tie $color, 'ValueRing', qw(red blue);
print "$color $color $color $color $color $color\n";
red blue red blue red blue

$color = 'green';
print "$color $color $color $color $color $color\n";
green red blue green red blue
```

簡單的實作在範例 13-3。

範例 13-3. ValueRing

```
package ValueRing;

# 繫結純量的建構函式
sub TIESCALAR {
    my ($class, @values) = @_;
    bless  \@values, $class;
    return \@values;
}

# 截斷讀的存取
sub FETCH {
    my $self = shift;
    push(@$self, shift(@$self));
    return $self->[-1];
}
```

```
# 截斷寫的存取
sub STORE {
    my ($self, $value) = @_;
    unshift @$self, $value;
    return $value;
}

1;
```

這個範例可能不會令人讚賞，但它說明如何簡單地去寫出可能很複雜的繫結。對於使用者而言，$color 只是一般變數，不是物件。所有的奧妙都隱藏在繫結之下。你不必因為使用了純量，而用純量參考，這裡我們使用陣列參考，你可以使用任何你想用的方式。因為雜湊參考是最有彈性的物件表示法，所以它常被使用在和被繫結的東西上。

至於陣列和雜湊，想當然爾這類的操作是更複雜了。一直到 5.004 版出現之後，才有被繫結的代碼出現，在這之前，被繫結的陣列在使用上有點限制，但被繫結的雜湊則一定有支援。因為有如此多的物件成員函式需要去支援被繫結的雜湊，大部分的使用者選擇從標準 Tie::Hash 模組繼承預設的成員函式。

接下來有許多使用繫結的有趣範例。

Tie 範例: Outlaw $_

這個奇怪的繫結類別被使用在變數 $_，用來指出不合法的非區域化的使用。我們不用 use（因為這會引發類別的 import() 成員函式），而改用 no 來載入，然後呼叫極少使用的 unimport() 成員函式。如下：

```
no UnderScore;
```

然後所有非區域化的全域變數 $_ 都會造成一個例外。

你可以對這個模組做個小小測試。

```
#!/usr/bin/perl
# nounder_demo — 示範如何由你的程式禁止 $_
no UnderScore;
@tests = (
```

```
    "Assignment"  => sub { $_ = "Bad" },
    "Reading"     => sub { print },
    "Matching"    => sub { $x = /badness/ },
    "Chop"        => sub { chop },
    "Filetest"    => sub { -x },
    "Nesting"     => sub { for (1..3) { print } },
);

while ( ($name, $code) = splice(@tests, 0, 2) ) {
    print "Testing $name: ";
    eval { &$code };
    print $@ ? "detected" : "missed!";
    print "\n";
}
```

結果如下:

```
Testing Assignment: detected
Testing Reading: detected
Testing Matching: detected
Testing Chop: detected
Testing Filetest: detected
Testing Nesting: 123missed!
```

最後一個不見了的原因是它已經被 for 迴圈適當地區域化，所以它被認為是安全的。

UnderScore 模組在範例 13-4。注意它有多麼短小。這個模組在它的初始化程式碼中自己作 tie 的動作。

範例 13-4. UnderScore

```
package UnderScore;
use Carp;
sub TIESCALAR {
    my $class = shift;
    my $dummy;
    return bless \$dummy => $class;
}
```

```
sub FETCH { croak "Read access to \$_ forbidden"  }
sub STORE { croak "Write access to \$_ forbidden" }
sub unimport { tie($_, __PACKAGE__) }
sub import { untie $_ }
tie($_, __PACKAGE__) unless tied $_;
1;
```

你不能在你的程式裡混用 use 和 no 來呼叫這個類別，因為它們都發生在編譯時期，而不是執行時期。若要再次使用 $_，用 local 來將它區域化。

Tie 範例:製作一個永遠可以追加資料的雜湊

下面的類別會產生一個雜湊，而這個雜湊的 key 存放在一個陣列內。

```
#!/usr/bin/perl
# appendhash_demo — 示範可自動追加的神奇雜湊
use Tie::AppendHash;
tie %tab, 'Tie::AppendHash';

$tab{beer} = "guinness";
$tab{food} = "potatoes";
$tab{food} = "peas";

while (my($k, $v) = each %tab) {
    print "$k => [@$v]\n";
}
```

這是結果：

```
food => [potatoes peas]
beer => [guinness]
```

為了使這個類別容易運作，我們使用了與標準模組繫結在一起的雜湊，如範例 13-5。為了將它們繫結在一起，我們載入 Tie::Hash 模組，然後繼承 Tie::StdHash 類別 —— 是的，它們是不同的兩個名稱。檔案 Tie/Hash.pm 提供 Tie::Hash 和 Tie::StdHash 兩個類別，它們有些微的不同。

範例 13-5. Tie::AppendHash

```
package Tie::AppendHash;
use strict;
use Tie::Hash;
use Carp;
use vars qw(@ISA);
@ISA = qw(Tie::StdHash);
sub STORE {
    my ($self, $key, $value) = @_;
    push @{$self->{$key}}, $value;
}
1;
```

Tie 範例：不分大小寫的雜湊

下面這個雜湊繫結名為 Tie::Folded，它提供不分 key 的大小寫雜湊。

```
#!/usr/bin/perl
# folded_demo — 示範不分大小寫的雜湊
use Tie::Folded;
tie %tab, 'Tie::Folded';

$tab{VILLAIN} = "big";
$tab{herOine} = "red riding hood";
$tab{villain} = "bad wolf";

while ( my($k, $v) = each %tab){
print "$k is $v\n";
}
```

下面是這個程式的輸出結果：

```
heroine is red riding hood
villain is big bad wolf
```

因為我們必須去攔截更多的存取，所以在範例 13-6 的類別比起範例 13-5 會有點複雜。

範例 13-6. Tie::Folded

```
package Tie::Folded;
use strict;
use Tie::Hash;
use vars qw(@ISA);
@ISA = qw(Tie::StdHash);
sub STORE {
    my ($self, $key, $value) = @_;
    return $self->{lc $key} = $value;
    }
sub FETCH {
    my ($self, $key) = @_;
    return $self->{lc $key};
}
sub EXISTS {
    my ($self, $key) = @_;
    return exists $self->{lc $key};
}
sub DEFINED {
    my ($self, $key) = @_;
    return defined $self->{lc $key};
}
1;
```

Tie 範例：以 key 或值尋找資料的雜湊

這有一個雜湊讓你利用 key 或數值來尋找成員。它利用一個存放成員函式來做這件事，不但可以使用 key 去存放值，它也可以使用值去存放 key。

一般來說，如果參考被當成存放的值，會有問題發生，因為你不能用參考來做為 key。Perl 的標準 Tie::RefHash 類別可以避免發生這個問題；我們將會繼承它，因此我們也可以避免這個難題。

```
#!/usr/bin/perl -w
# revhash_demo — 示範以 key 或值尋找資料的雜湊
use strict;
use Tie::RevHash;
```

```
my %tab;
tie %tab, 'Tie::RevHash';
%tab = qw{
    Red         Rojo
    Blue        Azul
    Green       Verde
};
$tab{EVIL} = [ "No way!", "Way!!" ];

while ( my($k, $v) = each %tab ) {
    print ref($k) ? "[@$k]" : $k, " => ",
        ref($v) ? "[@$v]" : $v, "\n";
}
```

執行時，revhash_demo 產生這個結果：

```
[No way! Way!!] = EVIL
EVIL => [No way! Way!!]
Blue => Azul
Green => Verde
Rojo => Red
Red => Rojo
Azul => Blue
Verde => Green
```

這個模組在範例 13-7。

範例 13-7. Tie::RevHash

```
package Tie::RevHash;
use Tie::RefHash;
use vars qw(@ISA);
@ISA = qw(Tie::RefHash);
sub STORE {
    my ($self, $key, $value) = @_;
    $self->SUPER::STORE($key, $value);
    $self->SUPER::STORE($value, $key);
}
```

```
sub DELETE {
    my ($self, $key) = @_;
    my $value = $self->SUPER::FETCH($key);
    $self->SUPER::DELETE($key);
    $self->SUPER::DELETE($value);
}

1;
```

Tie 範例：利用代碼計算存取次數

這有一個繫結檔案代碼的範例：

```
use Counter;
tie *CH, 'Counter';
while (<CH>) {

    print "Got $_\n";
}
```

執行時，程式會列印 Got 1、Got 2、Got 3 ...，一直到世界末日，除非你中斷它或將電腦重新開機。Counter 在範例 13-8。

範例 13-8. Counter

```
package Counter;
sub TIEHANDLE {
    my $class = shift;
    my $start = shift;
    return bless \$start => $class;
}
sub READLINE {
    my $self = shift;
    return ++$$self;
}
1;
```

Tie 範例:多個 sink 的檔案代碼

最後這個範例是，一個被繫結的代碼藉由將標準輸出和標準錯誤配對，來實作一個類似 T 型的功能。

```
use Tie::Tee;
tie *TEE, 'Tie::Tee', *STDOUT, *STDERR;
print TEE "This line goes both places.\n";
```

或者更詳盡的：

```
#!/usr/bin/perl
# demo_tietee
use Tie::Tee;
use Symbol;

@handles = (*STDOUT);
for $i ( 1 .. 10 ) {
    push(@handles, $handle = gensym());
    open($handle, ">/tmp/teetest.$i");
}

tie *TEE, 'Tie::Tee', @handles;
print TEE "This lines goes many places.\n";
```

Tie/Tee.pm 檔案在範例 13-9。

範例 13-9. Tie::Tee

```
package Tie::Tee;

sub TIEHANDLE {
    my $class   = shift;
    my $handles = [@_];

    bless $handles, $class;
    return $handles;
}
```

```
sub PRINT {
    my $href = shift;
    my $handle;
    my $success = 0;

    foreach $handle (@$href) {
        $success += print $handle @_;
    }

    return $success == @$href;
}

1;
```

相關資料

perlfunc(1) 的 tie 函式，perltie(1)。《Perl 程式設計》第五章「使用 tie 的變數」的部分。