第17章 重新审视面向对象：类

如果说广义函数是对象系统的动词，那么类就是名词。如同我在前面一章里提到的，Common Lisp程序中所有的值都是某个类的实例。更进一步，所有的类都被组织成以类T为根的单一层次体系。

类层次的体系由两个主要的类家族构成，即内置的类和用户定义的类。到目前为止，你已经学过的代表数据类型的类，诸如INTEGER、STRING和LIST这样的类，都是内置的。它们生活在类层次体系中它们自己的区域里，按照适当的子类和基类关系组织在一起，并且由那些我在本书中到目前为止讨论过的函数所管理。你不能创建这些类的子类，但是正如你在前一章里看到的，你可以定义特化在它们之上的方法，从而有效地扩展那些类的行为。[1]

但是当你想要创建新的名词时，例如前一章里用来表示银行帐户的那些类，你就需要定义你自己的类。这就是本章的主题。

17.1 DEFCLASS

你可以使用DEFCLASS宏来创建用户定义的类。由于一个类关联的行为是通过定义广义函数和特化在该类上的方法决定的，DEFCLASS的责任仅仅是将类定义为一种数据类型。

一个类作为数据类型的三个方面是它的名字、它与其他类的关系以及构成该类实例的那些槽（slot）的名字。[2]一个DEFCLASS的基本形式很简单。

```
(defclass name (direct-superclass-name*)
  (slot-specifier*))
```

什么是“用户定义的类”

术语“用户定义的类”不是来自语言标准的术语。从技术上来讲，当我说“用户定义的类”时，我指的是那些属于STANDARD-OBJECT的子类并且其元类（metaclass）是STANDARD-CLASS的类。但由于我不打算谈论你可以定义不是STANDARD-OBJECT的子类并且其元类不是

① 为一个已有的类定义新的方法，对于那些曾经使用诸如C++和Java这样的静态类型语言的人们来说可能听起来有些奇怪。在这些语言里，类的所有方法必须被定义为该类定义的一部分。但是具有使用诸如Smalltalk和Objective-C这类动态类型面向对象语言经验的程序员们则不觉得为已有类添加新行为有任何奇怪之处。

② 在其他面向对象语言里，“槽”可能被称为字段（field）、成员变量（member variable）或属性（attribute）。

STANDARD-CLASS的那些类的方式，你根本不需要关心这点。“用户定义的”并不是一个用来描述这些类的完美术语，因为实现可能以相同的方式定义了特定的类。不过，将它们称为标准类可能会带来更多的困惑，因为诸如INTEGER和STRING这样的内置类也是标准的，它们是由语言标准定义的但却没有扩展STANDARD-OBJECT。更复杂的事情在于，用户也有可能定义不是STANDARD-OBJECT子类的新类。特别的是，宏DEFSTRUCT同样定义了新的类，但那在很大程度上是为了向后兼容——DEFSTRUCT出现在CLOS之前，并且当CLOS被集成进语言时曾被改进用于定义类。因此在本章里，我将只谈论那些由DEFCLASS定义的使用默认的STANDARD-CLASS作为元类的类，并且由于缺少一个更好的术语，我将把它们称为“用户定义的”。

与函数和变量一样，你可以使用任何符号作为一个新类的名字。[1]类的名字与函数和变量的名字处在独立的名字空间里，因此你可以让类、函数和变量全部带有相同的名字。你将使用类名作为MAKE-INSTANCE的参数，该函数用来创建用户定义类的新实例。

那些*direct-superclass-name*指定了该新类将成为其子类的那些类。如果没有基类被列出，那么新类将直接成为STANDARD-OBJECT的子类。任何列出的类必须是其他用户定义的类，这确保了每一个新类都将最终追溯到STANDARD-OBJECT。STANDARD-OBJECT又是T的子类，因此，所有用户定义的类都是同样含有全部内置类的单一类层次体系的一部分。

在暂时省略槽描述符的情况下，前一章里你用到的某些类的DEFCLASS形式可能看起来像这样：

```
(defclass bank-account () ...)

(defclass checking-account (bank-account) ...)

(defclass savings-account (bank-account) ...)
```

我将在17.8节里讨论在*direct-superclass-name*中列出多于一个直接基类的含义。

17.2　槽描述符

DEFCLASS形式的大部分是由槽描述符的列表组成的。每个槽描述符定义的槽都属于该类的每个实例。实例中的每个槽都是一个可以保存值的位置，该位置可以通过函数SLOT-VALUE来访问。SLOT-VALUE接受一个对象和一个槽的名字作为参数并返回给定对象中该命名槽的值。它可以和SETF一起使用来设置对象中某个槽的值。

一个类也从它的所有基类中继承槽描述符，因此，实际存在于任何对象中的槽的集合是一个类的DEFCLASS形式中指定的所有槽和它的全部基类中指定的槽的并集。

在最小情况下，一个槽描述符可以只是一个名字。例如，你可以将bank-account类定义为带有两个槽customer-name和balance，如下所示：

---

① 跟为函数和变量命名一样，你可以使用任何符号作为类名的这个说法并不是很正确，你不能使用由语言标准所定义的名字。你将在第21章里看到如何避免这样的名字冲突。

```
(defclass bank-account ()
  (customer-name
   balance))
```

该类的每个实例都含有两个槽，一个用来保存该账户所属客户的名字而另一个用来保存当前余额。借助该定义，你可以用**MAKE-INSTANCE**来创建新的bank-account对象。

```
(make-instance 'bank-account) → #<BANK-ACCOUNT @ #x724b93ba>
```

MAKE-INSTANCE的参数是想要实例化的类的名字，而返回的值就是新的对象。[①]一个对象的打印形式取决于广义函数**PRINT-OBJECT**。在本例中，可应用的方法是由实现提供的特化在**STANDARD-OBJECT**上的方法。每一个对象都可以被打印成随后可被读回的形式，因此**STANDARD-OBJECT**打印方法使用了#<>语法，这将导致读取器在它试图读取该对象时报错。打印表示的其余部分是由实现定义的，但通常是一些类似于上面所显示的输出，其中包括该类的名字和一些诸如该对象的内存地址这样的可区别值。在第23章里，你将看到一个关于如何定义**PRINT-OBJECT**方法的例子，它使得一个特定类的对象可以被打印成更具说明性的形式。

使用刚刚给出的bank-account定义，创建出的新对象将带有未绑定（unbound）的槽。任何尝试获取未绑定槽的值的操作都将会报错，因此你必须在读取一个槽之前先设置它：

```
(defparameter *account* (make-instance 'bank-account))  → *ACCOUNT*
(setf (slot-value *account* 'customer-name) "John Doe") → "John Doe"
(setf (slot-value *account* 'balance) 1000)             → 1000
```

现在你可以访问这些槽的值了：

```
(slot-value *account* 'customer-name) → "John Doe"
(slot-value *account* 'balance)       → 1000
```

17.3 对象初始化

由于你不能对一个带有未绑定槽的对象做太多事，因此如果可以创建带有预先初始化槽的对象就非常好。Common Lisp提供了三种方式来控制槽的初始值。前面两种是通过在**DEFCLASS**形式中向槽描述符添加选项来实现的：通过:initarg选项，你可以指定一个随后作为**MAKE-INSTANCE**的关键字形参的名字并使该参数的值保存在槽中。另一个选项:initform可以让你指定一个Lisp表达式在没有:initarg参数传递给**MAKE-INSTANCE**时为该槽计算一个值。最后，为了完全控制初始化过程，你可以在广义函数**INITIALIZE-INSTANCE**上定义一个方法，它将被**MAKE-INSTANCE**调用。[②]

---

① **MAKE-INSTANCE**的参数实际上既可以是一个类的名字，也可以是一个由函数**CLASS-OF**或**FIND-CLASS**返回的类对象。

② 另一种影响槽的初始值的方式是通过**DEFCLASS**的:default-initargs选项。当一个特定的**MAKE-INSTANCE**调用没有给定该值时，该选项用来指定将被求值的形式以及提供特定初始化形参的参数。目前你不需要担心:default-initargs。

包含诸如:initarg或:initform等选项的槽描述符被写成以槽的名字开始后跟选项的列表。例如，如果你想要修改bank-account的定义，从而允许**MAKE-INSTANCE**的调用者传递客户名和初始余额，并为余额提供一个零美元的默认值，你可以写成这样：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

现在你可以创建一个账户并同时指定所有的槽值：

```
(defparameter *account*
  (make-instance 'bank-account :customer-name "John Doe" :balance 1000))

(slot-value *account* 'customer-name) → "John Doe"
(slot-value *account* 'balance)       → 1000
```

如果你没有提供:balance参数给**MAKE-INSTANCE**，那么balance的**SLOT-VALUE**将通过求值由:initform选项指定的形式而得到。但如果你没有指定:customer-name参数，那么customer-name槽将是未绑定的，并且在你设置它之前尝试读取它的操作将会报错。

```
(slot-value (make-instance 'bank-account) 'balance)       → 0
(slot-value (make-instance 'bank-account) 'customer-name) → error
```

如果你想要确保在创建账户的同时也提供客户名，那么你可以在初始化形式中产生一个错误，因为它只在没有提供初始化参数时被求值一次。你还可以使用初始化形式在每次它们被求值时生成一个不同的值——初始化形式对于每个对象都被重新求值。为了体会这些技术，你可以修改customer-name槽描述符并添加一个新的槽account-number，它被初始化为一个永远递增的计数器的值。

```
(defvar *account-numbers* 0)

(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))))
```

多数时候，:initarg和:initform选项的组合可以很好地初始化一个对象。不过，尽管初始化形式可以是任何Lisp表达式，但它却无法访问正在初始化的对象，因此它不能基于一个槽的值来初始化另一个槽。对于这种情况你需要在广义函数**INITIALIZE-INSTANCE**上定义一个方法。

基于它们的:initarg和:initform选项，在**STANDARD-OBJECT**上特化的**INITIALIZE-INSTANCE**主方法负责槽的初始化工作。由于你不想干扰这些工作，最常见的添加定制初始化代码

的方式是定义一个特化在你的类上的:after方法。[1]例如，假设你想要添加一个account-type槽并需要根据该账户的初始余额将其设置成:gold、:silver或:bronze这些值中的一个。你可以将你的类定义改成下面这样，其中添加了一个没有选项的account-type槽：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))
   account-type))
```

然后你可以为**INITIALIZE-INSTANCE**定义一个:after方法，根据保存在balance槽中的值来设置account-type槽。[2]

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
    (setf (slot-value account 'account-type)
          (cond
            ((>= balance 100000) :gold)
            ((>= balance 50000) :silver)
            (t :bronze)))))
```

为了保持该方法的形参列表与广义函数一致，形参列表中的&key是必不可少的。广义函数**INITIALIZE-INSTANCE**指定的形参列表包含了**&key**，从而允许个别方法可以指定它们自己的关键字参数，但其对特定的关键字参数却没有要求。这样，每一个方法都必须指定**&key**，哪怕它们没有指定任何**&key**参数。

另一方面，如果特化在某个特定类上的**INITIALIZE-INSTANCE**方法指定了一个**&key**参数，那么在创建该类的实例时，该参数就成为了一个**MAKE-INSTANCE**的合法参数。例如，有时在你开户时银行会支付一定比例的初始余额作为奖励，那么你可以像下面这样使用一个接受关键字参数来指定奖励百分比的**INITIALIZE-INSTANCE**方法来实现这一点：

```
(defmethod initialize-instance :after ((account bank-account)
                                       &key opening-bonus-percentage)
  (when opening-bonus-percentage
    (incf (slot-value account 'balance)
```

① 在Common Lisp中向**INITIALIZE-INSTANCE**添加一个:after方法，相当于在Java或C++中定义一个构造函数或是在Python中定义一个__init__方法。

② 在未习惯使用附加方法之前，你可能会犯的错误是，在**INITIALIZE-INSTANCE**上定义了一个方法而没有使用:after限定符。如果你这样做了，你将得到一个覆盖了默认方法的新的主方法。你可以使用函数**REMOVE-METHOD**和**FIND-METHOD**来移除不想要的主方法。某些开发环境可能提供图形用户接口来实现同样的事情。

```
(remove-method #'initialize-instance
  (find-method #'initialize-instance () (list (find-class 'bank-account))))
```

```
        (* (slot-value account 'balance) (/ opening-bonus-percentage 100)))))
```

通过定义这个**INITIALIZE-INSTANCE**方法，你使:opening-bonus-percentage在创建bank-account时成为了**MAKE-INSTANCE**的合法参数。

```
CL-USER> (defparameter *acct* (make-instance
                                'bank-account
                                 :customer-name "Sally Sue"
                                 :balance 1000
                                 :opening-bonus-percentage 5))
*ACCT*
CL-USER> (slot-value *acct* 'balance)
1050
```

17.4　访问函数

从**MAKE-INSTANCE**到**SLOT-VALUE**，你有了用于创建和管理你的类实例的所有工具。你想要做的其他任何事都可以用这两个函数来实现。不过，每一位了解优秀的面向对象编程实践原则的人都知道，直接访问一个对象的槽（或字段或成员变量）可能导致脆弱的代码。问题在于直接访问槽会将你的代码过于紧密地绑定到你的类的具体结构上。例如，假设你打算改变bank-account的定义，不再保存数值形式的当前余额，而是保存一个带有时间戳的提款和存款列表。在你改变了类定义来移除该槽或是保存新的列表到旧的槽时，直接访问balance槽的代码将很可能被打断。另一方面，如果你定义了一个用来访问该槽的balance函数，那么随后你可以重定义它，在类的内部表示改变的情况下保留其行为。并且使用这样一个函数的代码将无需修改而继续工作。

另一个使用访问函数而不是直接通过**SLOT-VALUE**来访问槽的优点在于，它可以让你限制外部代码修改槽的方式。[①]对于bank-account类的用户来说能够得到当前余额就可以了，但你可能想让余额的所有修改通过你将提供的其他函数访问到，例如deposit和withdraw。如果客户知道他们被假定只通过已发布的函数型API来管理对象，那么你可以提供一个balance函数但不使它成为可**SETF**的，如果你想让余额是只读的。

最后，使用访问函数可以使你的代码更整齐，因为它帮助你在大量情况下都不必使用相当繁琐的**SLOT-VALUE**。

很容易定义一个函数来读取balance槽的值。

```
(defun balance (account)
  (slot-value account 'balance))
```

不过，如果你知道你打算定义的bank-account的子类，那么将balance定义成一个广义函数可能是个好主意。通过这种方式，你可以在balance上为这些子类提供不同的方法或使用附加

① 当然，提供一个访问函数并不能真的产生任何限制，因为其他代码仍然可以使用**SLOT-VALUE**来直接访问槽。Common Lisp并没有提供C++和Java这些语言里所提供的严格对象封装。不过，如果一个类的设计者提供了访问函数而你忽略了它们仍然使用**SLOT-VALUE**，那么你最好知道你在做什么。你也可以使用我将在第21章里讨论的包系统，其更清楚地说明了某些槽不用于直接访问，方法就是不导出这些槽的名字。

方法来扩展其定义。因此你可能写出下面的定义：

```
(defgeneric balance (account))

(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

正如我已讨论过的，你不希望调用者直接设置余额。但对于其他槽，诸如customer-name，你可能也想提供一个函数来设置它们。定义这样的函数最简洁的方式是将其定义为**SETF**函数。

SETF函数是一种扩展**SETF**的方式，其定义了一种新的位置类型使其知道如何设置它。**SETF**函数的名字是一个两元素列表，其第一个元素是符号setf而第二个元素是一个符号，通常是一个用来访问该**SETF**函数将要设置的位置的函数名。**SETF**函数可以接受任何数量的参数，但第一个参数总是赋值到位置上的值。[①]例如，你可以定义**SETF**函数像下面这样设置bank-account中的customer-name槽：

```
(defun (setf customer-name) (name account)
  (setf (slot-value account 'customer-name) name))
```

在求值该定义之后，一个类似

```
(setf (customer-name my-account) "Sally Sue")
```

的表达式将被编绎成一个对你刚刚定义**SETF**函数的调用，其中"Sally Sue"作为第一个参数而my-account的值作为第二个参数。

当然，和读取函数一样，你希望你的**SETF**函数是广义的，因此你将实际像下面这样定义它：

```
(defgeneric (setf customer-name) (value account))

(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))
```

并且你当然也想为customer-name定义一个读取函数。

```
(defgeneric customer-name (account))

(defmethod customer-name ((account bank-account))
  (slot-value account 'customer-name))
```

这允许你写出下面的表达式：

```
(setf (customer-name *account*) "Sally Sue") → "Sally Sue"

(customer-name *account*)                    → "Sally Sue"
```

编写这些访问函数没有什么困难的，但是完全手工编写它们就跟Lisp风格不太吻合了。因此，**DEFCLASS**提供了三个槽选项，从而允许你为一个特定的槽自动创建读取和写入函数。

---

① 定义一个**SETF**函数，比如说(setf foo)，其带来的一种后果是，如果你还定义了对应的访问函数，在这种情况下是foo，那么你将可以在这个新的位置类型上使用构建在**SETF**之上的所有修改宏，比如**INCF**、**DECF**、**PUSH**和**POP**。

:read选项指定广义函数的名字，该函数只接受一个对象参数。当**DEFCLASS**被求值时，如果广义函数不存在则创建它。然后，为它添加一个方法，此方法基于新类特化一个参数并返回该槽的值。该名字可以是任意的，但通常将其命名成与槽本身相同的名字。这样，代替了前面给出的那些显式编写的balance广义函数的方法，你可以将bank-account中的balance槽的槽描述符修改成下面这样：

```
(balance
 :initarg :balance
 :initform 0
 :reader balance)
```

:write选项用来创建设置一个槽的值的广义函数和方法。该函数和方法按照**SETF**函数的要求创建，接受新值作为其第一个参数并把它作为结果返回，因此你可以通过提供一个诸如(setf customer-name)这样的名字来定义**SETF**函数。例如，你可以将槽描述符改变成下面的样子来为customer-name提供等价于前面所写的读取和写入方法：

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :reader customer-name
 :writer (setf customer-name))
```

由于经常同时需要用于读取和写入的函数，因此**DEFCLASS**还提供了一个选项:accessor来同时创建读取函数和对应的**SETF**函数。取代刚刚给出的槽描述符，一般情况下还可以写成这样：

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :accessor customer-name)
```

最后，还有一个你应当知道的槽选项是:documentation选项，使用它你可以提供一个字符串来记录一个槽的用途。将所有这些放在一起，并为account-number和account-type槽添加了读取方法，现在bank-account类的**DEFCLASS**形式看起来像下面这样：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name.")
    :accessor customer-name
    :documentation "Customer's name")
   (balance
    :initarg :balance
    :initform 0
    :reader balance
    :documentation "Current account balance")
   (account-number
    :initform (incf *account-numbers*)
    :reader account-number
    :documentation "Account number, unique within a bank.")
   (account-type
    :reader account-type
    :documentation "Type of account, one of :gold, :silver, or :bronze.")))
```

17.5 WITH-SLOTS 和 WITH-ACCESSORS

尽管使用访问函数将使代码更易于维护，但使用它们仍然有些繁琐。当编写那些实现一个类底层行为的方法时，情况将会更严重，这时你可能特别想直接访问槽来设置那些没有写入函数的槽，或是得到一些槽的值而无需为其定义读取函数。

这就是**SLOT-VALUE**适用的场合，不过它仍然很繁琐。更糟糕的是，一个多次访问同一个槽的函数或方法可能会产生大量对访问函数和**SLOT-VALUE**的调用。例如，就算是下面这个相当简单的方法也充满了对balance和**SLOT-VALUE**的调用，该方法在bank-account账户余额低于某个最小值时对其科以罚款：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (balance account) *minimum-balance*)
    (decf (slot-value account 'balance) (* (balance account) .01))))
```

而如果你打算直接访问槽值以避免运行附加的方法，它会变得更加混乱。

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (slot-value account 'balance) *minimum-balance*)
    (decf (slot-value account 'balance) (* (slot-value account 'balance) .01))))
```

两个标准宏**WITH-SLOTS**和**WITH-ACCESSORS**可以减轻这种混乱情况。两个宏都创建了一个代码块，在其中，简单的变量名可用于访问一个特定对象的槽。**WITH-SLOTS**提供了对槽的直接访问，就像**SLOT-VALUE**那样，而**WITH-ACCESSORS**提供了一个访问方法的简称。

WITH-SLOTS的基本形式如下所示：

```
(with-slots (slot*) instance-form
  body-form*)
```

每一个*slot*元素可以是一个槽的名字，它也用作一个变量名；或是一个两元素列表，第一个元素是一个用作变量的名字，第二个元素则是对应槽的名字。*instance-form*被求值一次来产生将要访问其槽的对象。在代码体内，这些变量名的每一次出现都被翻译成一个对**SLOT-VALUE**的调用，该对象和适当的槽名作为其参数。[①]这样，你可以像下面这样编写access-low-balance-penalty：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots (balance) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

或者使用两元素列表形式，像这样：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots ((bal balance)) account
```

① 由**WHITH-SLOTS**和**WITH-ACCESSORS**提供的“变量”名并不是真正的变量，它们是使用一种特殊类型的宏来实现的，这种宏称为符号宏（symbol macro），它允许一个简单的名字被展开成任意代码。在语言中引入符号宏主要是为了支持**WITH-SLOTS**和**WITH-ACCESSORS**，但你也可以将它们用于自己的目的。我将在第20章讨论它们的更多细节。

```
    (when (< bal *minimum-balance*)
      (decf bal (* bal .01)))))
```

如果你已经用一个:accessor而不只是:reader定义了balance，那么还可以使用**WITH-ACCESSORS**。**WITH-ACCESSORS**形式和**WITH-SLOTS**相同，除了槽列表的每一项都必须是包含一个变量名和一个访问函数名字的两元素列表。在**WITH-ACCESSORS**的主体中，对一个变量的引用等价为对相应访问函数的调用。如果访问函数是可以**SETF**的，那么该变量也可以。

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-accessors ((balance balance)) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

上面的代码中第一个balance是变量的名字，第二个是访问函数的名字，它们不必相同。例如，你可以编写一个方法来合并两个账户，其中使用两个**WITH-ACCESSORS**调用，每个账户一个。

```
(defmethod merge-accounts ((account1 bank-account) (account2 bank-account))
  (with-accessors ((balance1 balance)) account1
    (with-accessors ((balance2 balance)) account2
      (incf balance1 balance2)
      (setf balance2 0))))
```

是使用**WITH-SLOTS**还是使用**WITH-ACCESSORS**，与在**SLOT-VALUE**和一个访问函数之间进行选择是一样的：提供类基本功能的底层代码可以使用**SLOT-VALUE**和**WITH-SLOTS**直接修改那些不被访问函数支持的槽，或是为了显式地避免那些可能定义在访问函数上的附加方法所带来的影响。但你通常都应当使用访问函数或**WITH-ACCESSORS**，除非有特定的理由不这样做。

17.6　分配在类上的槽

你需要知道的最后一个槽选项是:allocation。:allocation的值可以是:instance或:class，如果没有指定则默认为:instance。当一个槽带有:class分配选项时，该槽只有单一值存储在类中并且被所有实例所共享。

尽管如此，:class槽和:instance槽的访问方法相同——通过**SLOT-VALUE**或访问函数来访问，这意味着你只能通过该类的一个实例来访问该槽的值，尽管它实际并没有保存在实例中。:initform和:initarg选项本质上也具有相同的效果，只是:initform将在类定义时而不是每次创建实例时求值。另一方面，传递:initarg给**MAKE-INSTANCE**会设置该值，从而影响该类的所有实例。

由于不能在没有该类实例的情况下访问一个类分配的槽，类分配的槽事实上并不等价于诸如Java、C++和Python这些语言里的静态字段或类字段。[①]而且，类分配的槽主要用来节省空间。

① 元对象协议（Meta Object Protocol, MOP）其不是语言标准的一部分，但被多数Common Lisp实现支持，它提供了一个函数class-prototype，该函数可以返回一个类的实例用来访问类槽。如果你正在使用一个支持MOP的实现，并且刚好在翻译一些来自其他语言的带有大量对静态字段或类字段使用的代码，那么该函数将给你一种简化翻译过程的方式。但这并不是惯用的做法。

如果你打算创建一个类的许多实例且所有实例都打算带有对同一个对象的引用，比如一个共享的资源池，那么可以通过使该槽成为类分配的槽而节省由每个实例都带有它们自己的引用所产生的开销。

17.7 槽和继承

正如我在前面章节里讨论的，类通过广义函数机制继承了来自其基类的行为，即一个在类A上进行特化的方法不仅可以应用在A的直接实例上，还可以应用在A的子类的实例上。类也可以从它的基类中继承槽，但手法上稍有不同。

在Common Lisp中，一个给定对象只能拥有一个带有给定名字的槽。尽管如此，一个给定类的继承层次关系中可能有多个类指定了具有同一个特定名字的槽。这既可能是因为一个子类包含了与其父类所指定的槽具有相同名字的槽描述符，也可能是多个基类指定了具有相同名字的槽。

Common Lisp处理这些情形的方式是，将来自新类和所有其基类的同名描述符合并在一起，并为每个唯一的槽名创建单一的描述符。当合并描述符时，不同的槽选项有不同的处理方式。例如，由于一个槽只能有单一的默认值，那么如果多个类指定了:initform，新类将使用来自最相关类的那一个。这允许子类可以指定一个与它本该继承的不同的默认值。

另一方面，:initargs不需要是互斥的，槽描述符中的每个:initarg选项都将创建一个可用来初始化该槽的关键字形参。多个参数不会产生冲突，因此新的槽描述符将含有所有的:initargs。**MAKE-INSTANCE**的调用者可以使用任何一个:initargs来初始化该槽。如果调用者传递了多个关键字参数来初始化同一个槽，那么会使用**MAKE-INSTANCE**调用中最左边的参数。

继承得到的:reader、:writer和:accessor选项不会包含在合并了的槽描述符中，因为由基类的**DEFCLASS**创建的这些方法已经可以用在新类上。不过，新类可以通过提供它自己的:reader、:writer或:accessor选项来定义其自己的访问函数。

最后，:allocation选项和:initform一样，由指定该槽的最相关的类决定。这样，有可能一个类的所有实例共享了一个:class槽，而它的子类的每个实例可能带有相同名字的自己的:instance槽。随后一个子类的子类可能将其重定义回:class槽，从而使该类的所有实例再次共享单一的槽。在后面的这种情况下，由子类的子类的实例共享的槽和由最初的基类共享的槽是不同的。

例如你有下面的类：

```
(defclass foo ()
  ((a :initarg :a :initform "A" :accessor a)
   (b :initarg :b :initform "B" :accessor b)))

(defclass bar (foo)
  ((a :initform (error "Must supply a value for a"))
   (b :initarg :the-b :accessor the-b :allocation :class)))
```

当实例化类bar时，你可以使用继承了的初始化参数:a来为槽a指定值，事实上必须这样做才能避免出错，因为由bar提供的:initform覆盖了继承自foo的那一个。为了初始化b槽，可以使

用继承的初始化参数:b或者新的初始化参数:the-b。不过，由于bar中的b槽带有:allocation选项，指定的值将保存在由bar的所有实例共享的槽中。同样的槽既可以使用在foo上特化的广义函数b的方法来访问，也可以使用直接在bar上特化的广义函数the-b的新方法来访问。为了访问foo或bar上的a槽，你将继续使用广义函数a。

通常，合并槽定义可以工作得很好。尽管如此，你需要关注当使用多重继承时，两个碰巧带有相同名字的无关的槽会被合并成新类中的单一的槽。这样，当在不同类上特化的方法应用在一个扩展了这些类的类上时，它们可能最终操作在同一个槽上。这实际上并不是太大的问题，因为你可以使用即将在第21章里学到的包（package）系统，来避免互不相关的代码中的名字冲突。

17.8　多重继承

到目前为止，你看到的所有类都只有单一的直接基类。Common Lisp也支持多重继承——一个类可以有多个直接基类，从所有这些类中继承可应用的方法和槽描述符。

多重继承并没有在本质上改变任何目前为止我所谈及的继承机制——每个用户定义的类已经带有多个基类，因为它们全部扩展至**STANDARD-OBJECT**，而后者扩展了**T**，所以它们至少有两个基类。多重继承带来的问题是一个类可以有超过一个的直接基类。这使得类的特化性概念在用于构造一个广义函数的有效方法以及合并继承的槽描述符时，会变得更加复杂。

这就是说，如果每个类都只有一个直接基类，那么决定类的特化性将极其简单。一个类及其所有基类可以排序成一条直线，从该类开始，后接它的直接基类，然后是后者的直接基类，最后一直上溯到**T**。但是当一个类有多个直接基类时，这些基类通常是彼此互不相关的。确实，如果一个类是另一个类的子类，那么你不会同时需要它们的直接子类。在这种情况下，子类比其基类更加相关这一规则不足以排序所有的基类。因此，Common Lisp提供了第二条规则，根据**DEFCLASS**的直接基类列表中列出的顺序来排序不相关的基类，更早出现在列表中的类被认为比列表中后面的类更相关。这条规则被认为有些随意，但确实可以允许每个类都有一个线性的类优先级列表（class precedence list），它可被用于检测某个基类是否比其他基类更相关。尽管如此，要注意并不存在所有类的全序。每个类都有其自己的类优先级列表，而同一个类可能以不同的顺序出现在不同类的类优先级列表中。

为了了解它是如何工作的，我们向银行应用中添加一个类money-market-account。一个货币市场账户组合了来自支票账户和储蓄账户的特征：客户可以填写支票，也可以挣得利息。你可以像下面这样定义它：

```
(defclass money-market-account (checking-account savings-account) ())
```

money-market-account的类优先级列表如下所示：

```
(money-market-account
 checking-account
 savings-account
 bank-account
 standard-object
 t)
```

注意该列表是怎样同时满足两条规则的：每个类都出现在它所有基类之前，并且checking-account和savings-account按照**DEFCLASS**中指定的顺序出现。

该类没有定义自己的槽，但是它会继承来自其两个直接基类的槽，包括两个直接基类继承自其基类的槽。同样，任何应用在类优先级列表中的任何类上的方法也将应用在money-market-account对象上。由于同一个槽的所有槽描述符都被合并了，因此money-market-account从bank-account中两次继承相同的槽描述符是不会有什么问题的。[①]

当不同的基类提供了完全无关的槽和行为时，多继承最容易理解。例如，money-market-account将从checking-account中继承用于处理支票的槽和行为,而从savings-account中继承用于计算利息的槽和行为。你不需要为只从一个或另一个基类继承的方法和槽担心类优先级列表。

尽管如此，也有可能从不同的基类中继承同一广义函数的不同方法。在这种情况下，类优先级列表将发挥其作用。例如，假设银行应用定义了一个广义函数print-statement来生成月对账单。假设已经有了在checking-account和savings-account上特化的print-statement方法。这两个方法对于money-market-account实例来说都是可应用的，但在checking-account上特化的方法被认为比在savings-account上特化的方法更加相关，因为checking-account在money-market-account的类优先级列表中出现在savings-account之前。

假设继承到的方法都是主方法并且你还没有定义任何其他方法，那么如果你在money-market-account上调用print-statement,则在checking-account上特化的方法将被使用。不过，这并不一定可以给你想要的行为，因为你可能希望货币市场账户的对账单中同时含有来自支票账户和储蓄账户对账单的元素。

可以用几种方式来修改用于money-market-accout的print-statement的行为。一种直接的方式是定义一个在money-market-account上特化的新的主方法。这可以让你更好地控制新行为,但可能需要比我即将讨论的其他选项要求更多的新代码。问题在于,当你可以使用**CALL-NEXT-METHOD**来“向上”调用下一个最相关方法时，也就是在checking-account上特化的那个方法，就没有办法来调用一个特定的不太相关的方法，例如在savings-account上特化的方法。因此，如果你想要重用那些打印对账单中savings-account部分的代码，就需要将那些代码分解成单独的函数，它随后可同时被money-market-account和savings-account的print-statement方法直接调用。

另一种可能性是，编写所有三个类的主方法去调用**CALL-NEXT-METHOD**，然后在money-market-account上特化的方法将使用**CALL-NEXT-METHOD**来调用在checking- account上特化的方法。当后者再调用**CALL-NEXT-METHOD**时，它将会运行savings-account上的方法，因为根据money-market-account的类优先级列表，它将是下一个最相关的方法。

① 换句话说，Common Lisp不会遇到像C++那样的宝石继承（diamond inheritance）问题。在C++中，当一个类子类了两个同时从公共基类继承了一个成员变量的类时，底下的类继承了成员变量两次，这导致了大量的混乱。

当然，如果你打算依赖一种编码约定（即所有方法都调用**CALL-NEXT-METHOD**）来确保所有可应用的方法都能在某一点处运行，那么应该考虑使用附加方法。在这种情况下，除了为checking-account和savings-account定义print-statement之上的主方法，还可以将这些方法定义成:after方法，同时在bank-account上定义单一的主方法。然后，调用在money-market-account上的print-statement将首先打印出由在bank-account上特化的主方法输出的基本账户对账单，接着是由在savings-account和checking-account上特化的:after方法输出的细节。如果你想要添加特定于money-market-account的细节，那么可以定义一个在money-market-account上特化的:after方法，它将在最后运行。

使用附加方法的优点在于，它使得哪个方法对于实现广义函数负主要责任，哪个方法只贡献附加的一点儿功能变得非常清楚了。其缺点在于，你无法良好地控制这些附加方法的运行顺序。如果你想要对账单中的checking-account部分在savings-account部分之前打印，那么就必须改变money-market-account，子类化这些类的顺序。但这种改变相当大，可能会影响其他方法和继承的槽。一般而言，如果你发现自己纠结于把直接基类列表的顺序作为调节特定方法行为的手段，那么你可能需要重新考虑你的设计。

另一方面，如果你并不关心具体的顺序，而只想让它在多个广义函数中是一致的，那么使用附加方法可能刚好合适。例如，如果在print-statement之外还有一个print-detailed-statement广义函数，你可以将这两个函数都实现为bank-account的不同子类上的:after方法，这样它们中正规和细化的对帐单部分的顺序将是相同的。

17.9　好的面向对象设计

以上就是Common Lisp对象系统的主要特性。如果你有丰富的面向对象编程经验，那么就能看到Common Lisp的特性是怎样来实现好的面向对象设计的。不过，如果你缺乏面向对象经验，那么就可能需要花费一些时间来吸取面向对象的思想。遗憾的是，这个主题相当大，已经超出了本书的讨论范围。或者正如Perl的对象系统手册页中所写："现在你只需买一本面向对象设计方法学的书，并在接下来的六个月里埋头苦读就好了。"或者你可以阅读本书后面的一些实用章节，在那里你将看到这些特性在实践中的使用方法的一些示例。不过，目前你需要暂且放下所有这些面向对象理论，转而学习另一个相当不同的主题：如何更好地使用Common Lisp强大但有时晦涩难懂的**FORMAT**函数。