第16章

重新审视面向对象：广义函数

16

Lisp的发明比面向对象编程的兴起早了几十年[1]，新的Lisp程序员们有时会惊奇地发现，原来Common Lisp竟是一门非常彻底的面向对象语言。Common Lisp之前的几个Lisp语言开发于面向对象还是一个崭新思想的年代，而那时有许多实验在探索将面向对象的思想（尤其是Smalltalk中所展现的）合并到Lisp中的方式。作为Common Lisp标准化过程的一部分，这些实验中的一些被合成在一起，以Common Lisp Object System（即CLOS）的名义出现。ANSI标准将CLOS合并到了语言之中，因此单独提及CLOS就不再有任何实际意义了。

CLOS为Common Lisp贡献的特性既有那些必不可少的，也有那些相对难懂的Lisp的“语言作为语言的构造工具”这一哲学的具体表现。本书无法对所有这些特性全部加以介绍，但在本章和下一章里，我将描述其中最常用的特性，并给出关于Common Lisp对象的概述。

你应当从一开始就注意到，Common Lisp的对象系统体现了与许多其他语言中相当不同的面向对象的原则。如果你能够深刻理解面向对象背后的基本思想，那么将会感谢Common Lisp在实现这些思想时所采用的强大和通用的方式。另一方面，如果你的面向对象经历很大程度上来自单一语言，那么你可能会发现Common Lisp的观点多少有些另类。你应当试图避免假设只存在一种方式令一门语言支持面向对象。[2]如果你几乎没有面向对象编程经验，那么也应当不难理解这里

① 现在，Simula通常被认为是第一个面向对象的语言，其发明于20世纪60年代早期，只比McCarthy的第一个Lisp晚了几年。尽管如此，直到20世纪80年代Smalltalk的第一个广泛使用的版本发布以后，面向对象才真正起飞，几年以后C++才得以发布。Smalltalk从Lisp那里获得了许多灵感，并将它与来自Simula的思想组合起来，产生出一种动态的面向对象语言，C++则组合了Simula和C——另一种相当静态的语言，从而得到了一个静态的面向对象语言。这种早期的分道扬镳导致了许多关于面向对象的定义的困惑。来自C++阵营的人们倾向于认为C++的特定方面，例如严格的数据封装是面向对象的关键特征。不过来自Smalltalk阵营的人们则认为C++的许多特性只是C++的特性而已，并不属于面向对象的核心内容。事实上，据说Smalltalk的发明者Alan Kay就曾说过：“我发明了术语面向对象（object oriented），而我可以告诉你C++并不是我头脑里所想的东西。”

② 有些人反对将Common Lisp作为面向对象语言。特别是那些将严格数据封装视为面向对象关键特征的人们，通常是诸如C++、Eiffel或Java这类相对静态语言的拥护者，他们不认为Common Lisp是真正面向对象的。当然，如果按照那样的定义，就算是Smalltalk这种无可争议的最早的和最纯粹的面向对象语言也不再是面向对象的了。另外，那些将消息传递视为面向对象关键特征的人们也不会很高兴，因为Common Lisp在声称自己是面向对象的同时，其面向广义函数的设计提供了纯消息传递所无法提供的自由度。

的解释，不过文中偶尔比较其他语言做同样事情的方式的内容，你就只好跳过不看了。

16.1　广义函数和类

面向对象的基本思想在于一种组织程序的强大方式：定义数据类型并将操作关联在那些数据类型上。特别是，你希望产生一种操作并让其确切行为取决于该操作所涉及的一个或多个对象的类型。所有关于面向对象的介绍中使用的经典例子，是可应用于代表各种几何图形的对象的draw操作。draw操作的不同实现可用于绘制圆、三角形和矩形，而对draw的调用将实际绘制出圆、三角形或矩形，具体取决于draw操作所应用到的对象类型。draw的不同实现被分别定义，并且新的版本可以被定义来绘制其他图形，而无需修改调用方或是任何其他draw实现的代码。这一面向对象风格称为“多义性”，源自希腊语polymorphism，意思是“多种形式”，因为单一的概念性操作，诸如绘制一个对象，可以带有许多不同的具体形式。

Common Lisp和今天的多数面向对象语言一样都是基于类的，所有对象都是某个特定类的实例。[①]一个对象的类决定了它的表示，诸如**NUMBER**和**STRING**这样的内置类带有不透明的表示，只能通过管理这些类型的标准函数来访问；而用户自定义类的实例，如同你将在下一章里看到的，由称为槽（slot）的命名部分组成。

类通过层次结构组织在一起，形成了所有对象的分类系统。一个类可以定义成另一个类的子类（subclass），后者称为它的基类（superclass）。一个类从它的基类中继承（inherit）其定义的一部分，而一个类的实例也被认为是其基类的实例。在Common Lisp中，类的层次关系带有一个单根，即类T，它是其他类的所有直接或间接基类。这样，Common Lisp中的每一个数据都是T的一个实例。[②]Common Lisp也支持多继承（multiple inheritance），即单一的类可以拥有多个直接基类。

在Lisp家族之外，几乎所有的面向对象语言都遵循了由Simula建立的基本模式：类所关联的行为由属于一个特定类的方法（method）或成员函数（member function）定义。在这些语言里，在一个特定对象上调用一个方法，然后该对象所属的类决定运行什么代码。这种方法调用的模型称为消息传递（message passing），这是来自Smalltalk的术语。从概念上来讲，在一个消息传递系统中，方法调用开始于向被调用方法所操作的对象发送一个消息，其中含有需要运行的方法名和任何参数。该对象随后使用其类来查找与该消息中的名字所关联的方法并运行它。由于每个类对于一个给定名字都有它自己的方法，因此发送相同的消息到不同的对象可以调用不同的方法。

早期的Lisp对象系统以类似的方式工作，提供了一个特殊函数SEND，用于向特定对象发送消息。尽管如此，这种方式并不完全令人满意，因为它使得方法调用不同于正常的函数调用。句法意义上的方法调用应写成

```
(send object 'foo)
```

① 基于原型的语言是另一种面向对象的语言类型。在这些语言里，JavaScript可能是最流行的例子，它的对象通过克隆一个原型对象来创建。该克隆可以随后被修改并用作其他对象的原型。

② 作为常量的T和作为类的T，除了刚好具有相同的名字以外没有特别的关系。作为值的T是类**SYMBOL**的一个直接实例，并且只是间接地成为作为类的T的一个实例。

而不是

```
(foo object)
```

更重要的是，由于方法不是函数，它们无法作为参数传递给像**MAPCAR**这样的高阶函数。如果一个人想要使用**MAPCAR**在一个列表的所有元素上调用方法，他不得不写成

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

而不是

```
(mapcar #'foo objects)
```

最终，工作在Lisp对象系统上的人们通过创建一种新的称为广义函数（generic function）的函数类型而将方法和函数统一在一起。广义函数不但解决了上面描述的问题，它还为对象系统开放了新的可能性，包括许多在消息传递对象系统中基本无法实现的特性。

广义函数是Common Lisp对象系统的核心，也是本章其余部分的主题。虽然我不可能在不提到类的情况下谈论广义函数，但目前我将把注意力集中在如何定义和使用广义函数上。在下一章里，我将向你展示如何定义你自己的类。

16.2 广义函数和方法

广义函数定义了抽象操作，指定了其名字和一个参数列表，但不提供实现。例如，下面就是你可能定义广义函数draw的方式，它将用来在屏幕上绘制不同的形状：

```
(defgeneric draw (shape)
  (:documentation "Draw the given shape on the screen."))
```

我将在下一节里讨论**DEFGENERIC**的语法，目前只需注意该定义并不含有任何实际代码。

广义函数的广义性至少在理论上体现在，它可以接受任何对象作为参数。[1]不过，广义函数本身并不能做任何事。如果你只是定义广义函数，那么无论用什么参数来调用它，它都将会报错。广义函数的实际实现是由方法（method）提供的。每一个方法提供了广义函数用于特定参数类的实现。也许在一个基于广义函数的系统和一个消息传递系统之间最大的区别在于方法并不属于类，它们属于广义函数，其负责在一个特定调用中检测哪个或哪些方法将被运行。

方法通过特化那些由广义函数所定义的必要参数，来表达它们可以处理的参数类型。例如，在广义函数draw中，你可以定义一个方法来特化shape参数，使其用于circle类的实例对象；而另一个方法则将shape特化成triangle类的实例对象。去掉实际的绘图代码以后，它们如下所示：

```
(defmethod draw ((shape circle))
  ...)

(defmethod draw ((shape triangle))
  ...)
```

① 这里和其他地方一样，对象意味着任何Lisp数据——Common Lisp并不像一些语言里那样区分对象和“基本”数据类型。Common Lisp中的所有数据都是对象，并且任何对象都是某个类的实例。

当一个广义函数被调用时，它将那些被传递的实际参数与它的每个方法的特化符进行比较，找出可应用（applicable）的方法，即那些特化符与实际参数相兼容的方法。如果你调用draw并传递一个circle的实例，那么在circle类上特化了shape的方法将是可应用的；而如果你传递了一个triangle实例，那么在triangle上特化了shape的方法将被应用。在简单的情况下，只有一个方法是可应用的，并且它将处理该调用。在复杂的情况下，可能有多个方法均可应用，它们随后将被组合起来——我将在16.5节里进行讨论，成为一个有效的（effective）方法来处理该调用。

你可以用两种方式来特化参数。通常你将指定一个类，其参数必须是该类的实例。由于一个类的实例也被视为该类的所有基类的实例，因此一个带有特化了某个特定类的参数的方法可以被应用在对应参数无论是该特定类的直接实例或是该类的任何子类的实例上。另一种类型的特化符是所谓的**EQL**特化符，其指定了方法所应用的特定对象。

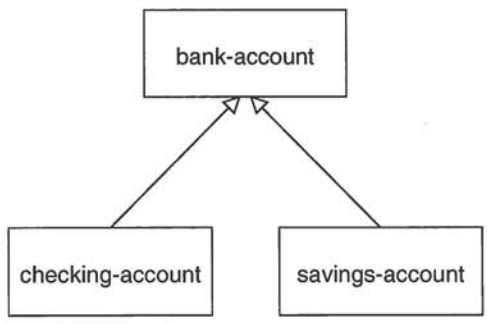
当一个广义函数只具有特化在单一参数上的方法并且所有特化符都是类特化符时，调用广义参数的结果跟在一个消息传递系统下调用方法的结果非常相似——操作的名字与调用时对象的类的组合决定了哪个方法被运行。

尽管如此，相反的方法查找顺序带来了消息传递系统所没有的可能性。广义函数支持特化在多个参数上的方法，提供了一个使多继承更具有可管理性的框架，并且允许你使用声明性的构造来控制方法如何组合成有效方法，从而在无须使用大量模板代码的情况下直接支持几种常用的设计模式。我将很快讨论到这些主题。但首先你需要了解两个用来定义广义函数的宏**DEFGENERIC**和**DEFMETHOD**的一些基础。

16.3　DEFGENERIC

为了给你一个关于这些宏和它们所支持的不同功能的大致印象，我将向你展示一些可能作为一个银行应用，或者说，一个相当幼稚的银行应用的一部分来编写的代码。重点在于观察一些语言特性而不是学习如何实际编写银行软件。例如，这些代码甚至并不打算处理像多种货币、审查跟踪以及事务集成这样的问题。

由于我不准备在下一章之前讨论如何定义新的类，因此目前你可以假设特定的类已经存在了。假设你已有一个bank-account类和它的两个子类checking-account以及savings-account。类层次关系如下所示：



第一个广义函数将是withdraw，它将账户余额减少指定数量。如果余额小于提款量，它将报错并保持余额不变。你可以从通过DEFGENERIC定义该广义函数开始。

除了缺少函数体之外，DEFGENERIC的基本形式与DEFUN相似。DEFGENERIC的形参列表指定了那些定义在该广义函数上的所有方法都必须接受的参数。在函数体的位置上，DEFGENERIC可能含有不同的选项。一个你应当总是带有的选项是:documentation，它提供了一个用来描述该广义函数用途的字符串。由于广义函数是纯抽象的，让用户和实现者了解它的用途将是重要的。因此，你可以像下面这样定义withdraw：

```
(defgeneric withdraw (account amount)
  (:documentation "Withdraw the specified amount from the account.
Signal an error if the current balance is less than amount."))
```

16.4 DEFMETHOD

现在你开始使用DEFMETHOD来定义实现了withdraw的方法。[①]

方法的形参列表必须与它的广义函数保持一致。在本例中，这意味着所有定义在withdraw上的方法都必须刚好有两个必要参数。在更一般的情况下，方法必须带有由广义函数指定的相同数量的必要和可选参数，并且必须可以接受对应于任何&rest或&key形参的参数。[②]

由于提款的基本操作对于所有账户都是相同的，因此你可以定义一个方法，其在bank-account类上特化了account参数。你可以假设函数balance返回当前账户的余额并且可被用于同SETF（因此也包括DECF）一起来设置余额。函数ERROR是一个用于报错的标准函数，我将在第19章里讨论其进一步的细节。使用这两个函数，你可以像下面这样定义出一个基本的withdraw方法：

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn."))
  (decf (balance account) amount))
```

如同这段代码显示的，DEFMETHOD的形式比DEFGENERIC更像是一个DEFUN形式。唯一的区别在于必要形参可以通过将形参名替换成两元素列表来进行特化。其中第一个元素是形参名，而第二个元素是特化符，其要么是类的名字要么是EQL特化符，其形式我将很快讨论到。形参名可

① 从技术上来讲，你可以完全跳过DEFGENERIC。如果你用DEFMETHOD定义了一个方法而没有定义相关的广义函数，那么广义函数将被自动创建。但是显式地定义广义函数是好的形式，因为它给你一个好的位置来文档化你想要的行为。

② 一个方法“接受”由其广义函数定义的&key和&rest参数的方式可以是使用&rest形参，使用相同的&key形参或是将&allow-other-keys与&key一起指定。方法也可以指定广义函数的形参列表中所没有的&key形参，当广义函数被调用时，任何由广义函数指定的&key参数或任何可应用的方法将被接受。

这种一致性规则带来的一个后果是，同一个广义函数上的所有方法将同样带有一致的形参列表。Common Lisp不支持诸如C++和Java那样的某些静态类型语言里支持的方法重载（method overloading），在那里相同的名字可被用于带有不同形参列表的方法。

以是任何东西——它不需要匹配广义函数中使用的名字，尽管经常是使用相同的名字。

该方法在每当withdraw的第一个参数是bank-account的实例时被应用。第二个形参amount被隐式特化到T上，而由于所有对象都是T的实例，它不会影响该方法的可应用性。

现在假设所有现金账户都带有透支保护。这就是说，每个现金账户都与另一个银行账户相关联，该账户将在现金账户的余额本身无法满足提款需求时被提款。你可以假设函数overdraft-account接受checking-account对象并返回代表了关联账户的bank-account对象。

这样，相比从标准的bank-account对象中提款，从checking-account对象中提款需要一些额外的步骤。你必须首先检查提款金额是否大于该账户的当前余额，如果大于，就将差额转给透支账户。然后你可以像处理标准的bank-account对象那样进行处理。

因此，你要做的是在withdraw上定义一个特化在checking-account上的方法来处理该传递过程，然后再让特化在bank-account上的方法接手。这样一个方法如下所示：

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))
  (call-next-method))
```

函数**CALL-NEXT-METHOD**是广义函数机制的一部分，用于组合可应用的方法。它指示控制应当从该方法传递到特定于bank-account的方法上。[1]当它不带参数被调用时，就像这里的这样，下一个方法将以最初传递给广义函数的参数被调用。它也可以带参数被调用，这些参数随后被传递给下一个方法。

你不必在每一个方法中调用**CALL-NEXT-METHOD**。尽管如此，如果你不这样做的话，新的方法将负责完全实现你想要的广义函数行为。假如你有一个bank-account的子类proxy-account，它并不实际跟踪自己的余额而是将提款请求代理到其他账户，那么你可以写一个如下的方法（假设有一个函数proxied-account的方法可以返回代理的账户）：

```
(defmethod withdraw ((proxy proxy-account) amount)
  (withdraw (proxied-account proxy) amount))
```

最后，**DEFMETHOD**还允许你通过使用**EQL**特化符来创建特化在一个特定对象上的方法。例如，假设该银行应用被部署在某个腐败银行上。假设变量*account-of-bank-president*保存了一个特定银行账户的引用，如同其名字显示的，该账户属于该银行的总裁。进一步假设变量*bank*代表该银行整体，而函数embezzle可以从银行中偷钱。银行总裁可能会让你“修复”withdraw来特别处理他的账户。

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
```

① **CALL-NEXT-METHOD**大致相当于Java中在super上调用一个方法，或是在Python或C++中使用一个显式的类限定方法或函数名。

```
    (when (plusp overdraft)
      (incf (balance account) (embezzle *bank* overdraft))))
  (call-next-method))
```

不过需要注意到，EQL特化符中提供了特化对象的形式，在本例中是变量*account-of-bank-president*，其只在DEFMETHOD被求值时求值一次。在定义方法时该方法将特化*account-of-bank-president*的值。随后，改变该变量将不会改变该方法。

16.5 方法组合

在一个方法体之外，CALL-NEXT-METHOD没有任何意义。在一个方法之内，它被广义函数机制定义，用来在每次广义函数使用的所有应用于特定调用的方法被调用时构造一个有效方法。这种通过组合可应用的方法来构造有效方法的概念是广义函数概念的核心，并且是让广义函数可以支持消息传递系统里所没有的机制的关键。因此，值得更进一步地观察究竟发生了什么。那些在他们的意识中带有根深蒂固的消息传递模型思想的人们应当尤其注意这点，因为广义函数相比消息传递完全颠覆了方法的调度过程，使得广义函数而不是类成为了主要推动者。

从概念上讲，有效方法由三步构造而成：首先，广义函数基于被传递的实际参数构造一个可应用的方法列表。其次，这个可应用方法的列表按照它们的参数特化符中的特化程度（specificity）排序。最后，根据排序后列表中的顺序来取出这些方法并将它们的代码组合起来以产生有效方法。[①]

为了找出可应用的方法，广义函数将实际参数与它的每一个方法中的对应参数特化符进行比较。当且仅当所有特化符均和对应的参数兼容，一个方法便是可应用的。

当特化符是一个类的名字时，如果该名字是参数的实际类名或是它的一个基类的名字，那么该特化符将是兼容的。（再次强调，不带有显式特化符的形参将隐式特化到类T上从而与任何参数兼容。）一个EQL特化符当且仅当参数和特化符中所指定的对象是同一个时才是兼容的。

由于所有参数都将在对应的特化符中被检查，它们都会影响一个方法是否是可应用的。显式地特化了超过一个形参的方法被称为多重方法（multimethod）。我将在16.8节中讨论它们。

在可应用的方法被找到以后，广义函数机制需要在将它们组合成一个有效方法之前对它们进行排序。为了确定两个可应用方法的顺序，广义函数从左到右比较它们的参数特化符，[②]并且两个方法中第一个不同的特化符将决定它们的顺序，其中带有更加特定的特化符的方法排在前面。

由于只有可应用的方法在排序，你可以看出所有由类特化符命名的类对应的参数实际上都是它们的实例。在典型情况下，如果两个类特化符不同，那么其中一个将是另一个的子类。在那种情况下，命名了子类的特化符将被认为是更加相关的。这就是为什么在checking-account上特化了account的方法被认为比在bank-account上特化它的方法是更加相关的。

① 尽管构造有效方法的过程听起来很费时，但在开发快速的Common Lisp实现过程中有相当多的努力被用于使上述过程更有效率，一种策略是缓存有效方法以便未来在相同参数类型上的调用可以被直接处理。

② 事实上，比较特化符的顺序可以通过DEFGENERIC的选项:argument-precedence-order来定制，尽管该选项很少被用到。

多重继承稍微复杂化了特化性的概念，因为实际参数可能是两个类的实例，而两者都不是对方的子类。如果这样的类被用于参数特化符，那么广义函数就无法只通过子类比它们的基类更加相关这一规则来决定它们的顺序。在下一章里，我将讨论特化性的概念如何被扩展用于处理多重继承。目前我要说明的只是存在一个确定的算法来决定类特化符的顺序。

最后，**EQL**特化符总是比任何类特化符更加相关，并且由于只有可应用的方法被考虑，如果对于一个特定形参有多于一个方法带有EQL特化符，那么它们一定全部带有相同的EQL特化符。这样对这些方法的比较将取决于其他参数。

16.6　标准方法组合

现在，你理解了找出可应用的方法并对它们进行排序的方式，你可以更进一步来观察最后一步——排序的方法列表是如何被组合成单一有效方法的。默认情况下，广义函数使用一种称为标准方法组合（standard method combination）的机制。标准方法组合将方法组合在一起，从而使**CALL-NEXT-METHOD**像你看到的那样工作——最相关的方法首先运行，然后每个方法可以通过**CALL-NEXT-METHOD**将控制传递给下一个最相关的方法。

不过，这里面还有更多的细节。到目前为止，我所讨论过的所有方法都称为主方法（primary method）。如同其名字所显示的，主方法被用于提供一个广义函数的主要实现。标准方法组合也支持三种类型的辅助方法：`:before`、`:after`和`:around`。附加方法定义是用**DEFMETHOD**像一个主方法那样写成的，但是它带有一个方法限定符（method qualifier），其命名了方法的类型，介于方法名和形参列表之间。例如，一个在类`bank-account`上特化了`account`形参的`withdraw`的`:before`方法以下面的定义开始：

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

每种类型的附加方法以不同的方式组合到有效方法之中。所有可应用的`:before`方法（不只是最相关的）都将作为有效方法的一部分来运行。如同其名字所显示的，这些`:before`方法将在最相关的主方法之前以最相关者优先的顺序来运行。这样，`:before`方法可用来做任何需要确保主方法可以运行的准备工作。例如，你可以使用特化在`checking-account`上的`:before`方法像下面这样来实现对现金账户的透支保护：

```
(defmethod withdraw :before ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft))))
```

这个`:before`方法相比于一个主方法有三个优点。其中之一是它使得该方法改变`withdraw`整体行为的方式变得非常直观，但它并不打算影响主要的行为或是改变返回的结果。

下一个优点在于，一个特化在比`checking-account`更相关类上的主方法将不会影响该`:before`方法，从而使得`checking-account`子类的作者可以更容易地扩展`withdraw`的行为而同时保存它的一部分老的行为。

最后，由于:before方法不需要调用CALL-NEXT-METHOD来将控制传递给其余的方法，所以就不可能因为忘记这点而引入bug。

其他的附加方法同样以它们的名字所建议的方式融合进有效方法。所有的:after方法都在主方法之后以最相关者最不优先的顺序运行，也就是说与:before方法相反。这样，:before和:after方法组合在一起创建了一系列嵌套包装在由主方法所提供的核心功能周边的环境。每一个更相关的:before方法将有机会设置环境以便不太相关的:before方法和主方法得以成功运行，而每一个更相关的:after方法将有机会在所有主方法和更不相关的:after之后进行清理工作。

最后，除了它们运行在所有其他方法的外围，:around将以非常类似主方法的方式被组合。这就是说，来自最相关:around方法的代码将在其他任何代码之前运行。在:around方法的主体中，CALL-NEXT-METHOD将指向下一个最相关的:around方法的代码，或是在最不相关的:around的方法中指向由:before方法、主方法和:after方法组成的复合体。几乎所有的:around方法都会含有一个对CALL-NEXT-METHOD的调用，因为如果不这样做的话，:around的方法就会完全劫持广义函数中除了最相关:around方法之外的所有方法的实现。

这种类型的方法劫持偶尔也会被用到，但是典型的:around方法通常被用于建立一些其他方法得以运行的动态上下文。例如绑定一个动态变量，或是建立一个错误处理器（我将在第19章里讨论这一点）。差不多一个:around方法不去调用CALL-NEXT-METHOD的唯一场合就是当它返回一个缓存自之前对CALL-NEXT-METHOD的调用时。不管怎么说，一个没有调用CALL-NEXT-METHOD的:around的方法有责任正确实现广义函数在方法可能应用到的所有类型参数下的语义，包括未来定义的子类。

附加方法只是一种更简洁和具体地表达特定常用模式的便利方式。它们并不能让你做到任何通过将带有额外努力的主方法与一些代码约定和额外输入相组合所不能做到的事情。也许它们最大的好处在于它们提供了一个扩展广义函数的统一框架。通常库将定义一个广义函数并提供默认的主方法，然后允许该库的用户通过定义适当的附加方法来定制它的行为。

16.7 其他方法组合

在标准方法组合之外，该语言还指定了九种其他的内置方法组合，也称为简单内置方法组合。你还可以自定义方法组合，尽管这是一个相对难懂的特性并且超出了本书的范围。我将简要介绍简单内置组合的功能。

所有的简单组合都遵循了相同的模式：和调用最相关主方法并让它通过CALL-NEXT-METHOD来调用次相关主方法的方式有所不同，简单方法组合通过将所有主方法的代码一个接一个地全部包装在一个由方法组合的名字所给出的函数、宏或特殊操作符的调用中来产生一个有效方法。9种组合分别以下列操作符来命名：+、AND、OR、LIST、APPEND、NCONC、MIN、MAX和PROGN。另外简单组合只支持两种方法：按照刚刚描述的方式进行组合的主方法，以及:around方法，它和标准方法组合中的:around方法具有相似的工作方式。

例如，一个使用“+”方法组合的广义函数将返回其有主方法返回的结果之和。注意到由于这些宏的短路行为，**AND**和**OR**方法组合不一定会运行所有主方法——使用**AND**组合的广义函数将在一个方法返回**NIL**时立即返回，否则将返回最后一个方法的值。类似地，**OR**组合将返回第一个由任何方法返回的非**NIL**的值。

为了定义一个使用特定方法组合的广义函数，你可以在**DEFGENERIC**形式中包含一个:method-combination选项。连同该选项所提供的值是你想要使用的方法组合的名字。例如，为了定义一个广义函数priority，其使用“+”方法组合返回所有单独方法的返回值之和，你可以写成下面这样：

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination +))
```

默认情况下，所有这些方法组合以最相关者优先的顺序组合主方法。尽管如此，你可以通过在**DEFGENERIC**形式中的方法组合名之后包含关键字:most-specific-last来逆转这一顺序。该顺序在你使用“+”组合的时候可能无关紧要，除非方法带有副作用，但是出于演示的目的，你可以像下面这样使用最相关者最不优先的顺序来改变priority：

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination + :most-specific-last))
```

定义在使用这些组合之一的广义函数上的主方法必须被限定在该方法组合的名字上。这样，一个特定于priority的主方法可能看起来像这样：

```
(defmethod priority + ((job express-job)) 10)
```

这可以使你清楚地看到一个属于特定类型广义函数的方法定义。

所有简单内置方法组合也支持:around方法，其工作方式与标准方法组合中的:around方法类似：最相关的:around方法在任何其他方法之前运行，而**CALL-NEXT-METHOD**被用于将控制传递给越来越不相关的:around方法，直到到达组合的主方法。:most-specific-last选项并不影响:around方法的顺序。并且如同我前面提到的，内置方法组合不支持:before或:after方法。

和标准方法组合一样，这些方法组合不允许你做到任何你不能手工做到的事情。但是它们确实可以允许你表达你所想要的事情，并且让语言来帮助你将所有东西组织在一起从而使你的代码更加简洁且更富有表达性。

这就是说，很可能在99%的时间里标准方法组合是你需要的东西。在其中剩下的1%的事件里，可能99%的情况将被一个简单内置方法组合处理。如果你遇到了1%中的1%的情况，其中没有内置组合可以满足需要，那么你可以在你喜爱的Common Lisp参考中查询**DEFINE-METHOD-COMBINATION**。

16.8　多重方法

显式地特化了超过一个广义函数的必要形参的方法称为多重方法。多重方法是广义函数和消息传递真正有区别的地方。多重方法无法存在于消息传递语言之中是因为它们不属于一个特定的

类；相反，每一个多重方法都定义了一个给定广义函数的部分实现，并且当广义函数以匹配所有该方法的特化参数被调用时采用。

多重方法与方法重载

曾经使用过诸如Java和C++这些静态类型消息传递语言的程序员们可能认为，多重方法听起来类似于这些语言中一种称为方法重载（method overloading）的特性。不过事实上这两种语言特性相当不同，因为重载的方法是在编译期被选择的，其所基于的是编译期的参数类型而不是运行期。为了观察方法重载的工作方式，考虑下面的两个Java类：

```
public class A {
  public void foo(A a) { System.out.println("A/A"); }
  public void foo(B b) { System.out.println("A/B"); }
}
public class B extends A {
  public void foo(A a) { System.out.println("B/A"); }
  public void foo(B b) { System.out.println("B/B"); }
}
```

现在考虑当你从下面的类中运行main方法时将会发生什么。

```
public class Main {
  public static void main(String[] argv) {
    A obj = argv[0].equals("A") ? new A() : new B();
    obj.foo(obj);
  }
}
```

当你告诉Main来实例化A时，它像你可能期待的那样打印出“A/A”。

```
bash$ java com.gigamonkeys.Main A
A/A
```

不过，如果你告诉Main来实例化B，那么obj的真正类型将只有一半被实际分发。

```
bash$ java com.gigamonkeys.Main B
B/A
```

如果重载的方法像Common Lisp的多重方法那样工作，那么上面将打印出“B/B”。在消息传递语言里有可能手工实现多重分发，但这将与消息传递模型背道而驰，因为多重分发的方法中的代码并不属于任何一个类。

多重方法对于所有这些情形都很完美，而在一个消息传递语言里你将很难决定一个特定行为应该属于哪个类。一个鼓在用鼓棒敲它的时候产生的声音究竟是由鼓的类型还是棒的类型决定的？当然，两者都是。为了在Common Lisp中对这种情况建模，你可以简单地定义一个接受两个参数的广义函数beat。

```
(defgeneric beat (drum stick)
  (:documentation
   "Produce a sound by hitting the given drum with the given stick."))
```

然后，你可以定义不同的多重方法来实现用于你所关心的不同组合的beat。例如：

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)
(defmethod beat ((drum snare-drum) (stick brush)) ...)
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)
(defmethod beat ((drum tom-tom) (stick brush)) ...)
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

多重方法不能帮助处理组合爆炸。如果你需要建模五种类型的鼓和六种类型的鼓棒，并且每一种组合都产生不同的声音，那么不存在更好的办法。无论是否使用多重方法你都需要三十种不同的方法来实现所有的组合。多重方法使你免于手工编写大量用于分发的代码，而让你使用与处理特化在单一参数上的方法相同的内置多态分发技术。[①]

多重方法还可以使你免于将一组类互相关联在一起。在鼓/棒示例中，鼓类的实现不需要知道任何关于不同类型鼓棒的信息，而鼓棒类也不需要知道任何关于不同类型鼓的信息。多重方法将完全无关的类联系在一起来描述它们的组合行为，而不要求这些类彼此之间的任何互操作。

16.9　未完待续……

我已经介绍了广义函数的基础，并且还介绍了超出范围之外的内容，即Common Lisp对象系统中的动词。在下一章里，我将向你展示如何定义你自己的类。

① 在没有多重方法的语言里，你必须手工编写分发代码来实现依赖于超过一个对象的类的行为。流行的Visitor的设计模式的目的就是结构化一系列单一分发的方法调用从而提供多重分发。尽管如此，它要求有一种彼此知道的类。Visitor模式在被用作分发超过两个对象时还会快速地陷入组合爆炸。