



**在**第4章里，我讨论了Lisp读取器是如何将文本名字转化成用来传递给求值器的对象的，它们是一种称为符号的对象。实践表明，拥有专门用来表示各种名字的内置数据类型，对于多种编程任务都是有用的。<sup>①</sup>不过，这并不是本章的主题。在本章里，我将讨论一个处理名字方面的问题：如何避免彼此无关的人开发的代码出现名字冲突。

举个例子，假设你正在编写一个程序并决定在其中使用一个第三方库。你不想为了避免这些名字与你自己的程序中所使用的名字相冲突，而要知道那个库中的每一个函数、变量、类和宏的名字。你更希望这个库中的大多数名字和你程序中的名字是彼此无关的，哪怕是它们碰巧使用了相同的文本表示。同时，你希望这个库所定义的特定名字就是被用来访问的——这些名字构成了它的公共API，你会在自己的程序中用到它们。

在Common Lisp中，这一名字空间问题最后归结到，如何控制读取器将文本名称转化成符号这个问题上：如果你希望相同的名字可以被求值器同等看待，那么需要确保读取器使用相同的符号来表示每个名字。同样地，如果你希望两个名字被视作是不同的，甚至在它们刚好具有相同的文本名字时，那么就需要让读取器分别创建不同的符号来表示它们。

## 21.1 读取器是如何使用包的

在第4章里，我简单讨论了Lisp读取器是如何将名字转化成符号的，但我刻意忽略了大部分细节。现在是时候从更近的角度来看看这个过程中究竟发生了什么了。

我先来描述读取器所理解的名字的语法，以及该语法和包之间的关系。目前，你可以将包想像成一个字符串和符号的映射表。如同你在下一节里将会看到的，实际的映射过程比一个简单的查找表稍微灵活一些，但这对读者来说通常意义不大。每个包也都有一个名字，该名字可用来通过函数**FIND-PACKAGE**找到对应的包。

读取器用来访问包中的名字-符号映射的两个关键函数是**FIND-SYMBOL**和**INTERN**。这两个函数都接受一个字符串和一个可选的包。当包未指定时，包参数默认为全局变量**\*PACKAGE\***的值，也称为当前包。

<sup>①</sup> 依赖于符号数据类型的编程，被恰如其名地称为符号计算。它和基于数值的编程正好相反。一个所有程序员都应当熟悉的、主要的符号计算程序的例子是编译器，它将一个程序的文本视为符号数据并将其转化成一种新的形式。

**FIND-SYMBOL**在包中查找名为给定字符串的符号并将其返回，如果没有找到任何符号则返回**NIL**。**INTERN**也会返回一个已有的符号，否则它会创建一个以该字符串命名的新符号并将其添加到包里。

你所使用的大多数名字都是非限定的（unqualified），就是说名字里不带冒号。当读取器读到这样的名字时，它先将名字中所有未转义的字母转换成大写形式，然后将得来的字符串传给**INTERN**，从而将该名字转化成一个符号。这样，每当读取器读到相同的包里面相同的名字时，它都将得到相同的符号对象。这是很重要的，因为求值器使用符号的对象标识来决定一个给定函数所指向的函数、变量或其他程序元素。因此，诸如(*hello-world*)这样的表达式得以调用一个特定的*hello-world*函数的原因就在于，读取器在读取对该函数的调用和定义该函数的**DEFUN**形式时返回了相同的符号。

含有单冒号或双冒号的名字是包限定的名字。当读取器读取包限定的名字时，它将名字在冒号处拆开，前一部分作为包的名字，后一部分作为符号的名字。读取器查找适当的包并用它来将符号名转化成一个符号对象。

只含单冒号的名字必须指向一个外部符号（external symbol）——一个被包导出（export）作为公共接口来使用的符号。如果命名的包不含有一个给定名字的符号，或是含有该符号但并未导出，那么读取器会产生一个错误。含双冒号的名字可以指向命名包中的任何符号，尽管这通常不是个好主意——导出符号的集合定义了一个包的公共接口，而如果你不遵守包作者关于哪些名字是公开的而哪些名字是私有的约定，那么在使用时肯定会遇到麻烦。另一方面，有时一个包的作者可能忽略了导出一个确实应当开放给公众的符号。在这种情况下，含双冒号的名字可以让你无需等待该包的下一个版本的发布即可完成手头的工作。

读取器理解的符号语法的另外两点分别是关键字符和未进入（uninterned）包的符号。关键字符在书写上以一个冒号开始。这类字符在名为**KEYWORD**的包中创建并自动导出。更进一步，当读取器在**KEYWORD**包中创建一个符号时，它也会定义一个以该符号作为其名字和值的常量。这就是你可以在参数列表中使用关键字而无需引用它们的原因——当它们出现在一个值的位置上时，它们求值到自身。这样：

```
(eql ':foo :foo) → T
```

和所有符号一样，关键字符的名字在它们被创建之前就被读取器全部转换成大写形式了。名字中不包含前导冒号。

```
(symbol-name :foo) → "FOO"
```

未进入的字符在写法上以“`#:`”开始。这些名字（在去掉“`#:`”后）被正常地转换成大写形式并被转化成符号，但这些符号还没有进入任何包，每当读取器读到一个带有“`#:`”的名字时，它都会创建一个新的符号。这样：

```
(eql '#:foo '#:foo) → NIL
```

一般不需要自行书写这种语法，但有时当你打印一个含有由**GENSYM**所返回的符号的S-表达

式时就会看到它。

```
(gensym) → #:G3128
```

## 21.2 包和符号相关的术语

如同我先前提到的，包所实现的从名字到符号的映射比简单的查找表更加灵活。在核心层面上，每一个包都含有一个从名字到符号的查找表，但还有其他几种方式通过一个给定包中的非限定名字可以访问一个符号。为了更有意义地谈论这些方法，你需要了解一些词汇表。

首先，所有可在在一个给定包中通过**FIND-SYMBOL**找到的符号被称为在该包中是可访问的(*accessible*)。换句话说，一个包中可访问的符号是，该包为当前包时非限定名字指向的符号。

可以通过两种方式访问一个符号。前一种方式要求包的名字-符号表中含有该符号的项，这时我们称该符号存在(*present*)于该包中。当读取器让一个新符号进入一个包时，该符号会添加到包的名字-符号表中。该符号首先停留的包称作该符号的主包(*home package*)。

另一种方式是当某个包继承一个符号时，该符号在该包中就是可访问的。一个包通过使用(*use*)其他包来继承这些包中的符号。在被使用的包中，只有外部(*external*)符号才能被继承。可以通过在包中导出(*export*)一个符号来使其成为外部符号。导出操作除了可以使包的其他使用者继承符号以外，还可以使其能够通过带有单冒号的限定名称来引用，如同你在上一节里所看到的那样。

为了保证从名字到符号的映射的确定性，包系统只允许每个名字在给定的包中指向单一符号。这就是说，一个包不能同时有一个本身定义的符号和一个同名的继承得来的符号，或是同时从不同的包中继承两个具有相同名字的不同符号。不过，你可以通过使其中一个可访问的符号成为隐蔽(*shadow*)符号来解决冲突，这可以使其他同名的符号变得不可访问。每一个包都在它们的名字-符号列表之外维护了一个隐蔽符号的列表。

一个已有的符号可以通过将其添加到另一个包的名字-符号表中，来导入(*import*)到这个包。这样，同一个符号可以同时存在于多个包中。有时，导入符号只是因为希望它们在被导入的包中可以直接访问而无需使用它们的主包。其他时候，导入符号则是因为只有存在的符号才可以被导出或是成为隐蔽符号。举个例子，如果一个包需要使用两个带有同名外部符号的包，那么其中一个符号必须导入该包，以便可添加到该包的隐蔽符号列表并使另一个符号成为不可访问的。

最后，一个已有的符号可以从一个包中退出(*unintern*)，这会导致它被清除出名字-符号表，并且如果它是一个隐蔽符号，也会被清除出隐蔽符号列表。你可能想让一个符号从一个包中退出，从而消除该符号和一个来自你想使用的包中的外部符号之间的冲突。一个不存在于任何包中的符号称为未进入(*uninterned*)的符号，它不能被读取器读取，并且将采用`#:foo`语法进行打印。

## 21.3 ·三个标准包

在下一节里，我将向你展示如何定义你自己的包，包括如何让一个包使用另一个包，如何导出、隐蔽和导入符号。不过首先让我们看一些你已经在使用的包。最初启动Lisp时，**\*PACKAGE\***

的值通常是COMMON-LISP-USER包，有时也叫做CL-USER。<sup>①</sup>CL-USER使用了包COMMON-LISP，后者导出了语言标准定义的所有名字。因此，当在REPL中键入一个表达式时，所有诸如标准函数、宏、变量之类的名字都将转化成从COMMON-LISP包中导出的符号，而其他名字进入到COMMON-LISP-USER包中。例如，名字\*PACKAGE\*是从COMMON-LISP包中导出的，如果你想要查看\*PACKAGE\*的值，可以输入：

```
CL-USER> *package*
#<The COMMON-LISP-USER package>
```

这是因为COMMON-LISP-USER使用了COMMON-LISP，或者也可以使用一个包限定的名字：

```
CL-USER> common-lisp:*package*
#<The COMMON-LISP-USER package>
```

甚至可以使用COMMON-LISP的昵称CL：

```
CL-USER> cl:*package*
#<The COMMON-LISP-USER package>
```

21

但是\*x\*不是COMMON-LISP中的符号，因此如果你输入：

```
CL-USER> (defvar *x* 10)
*x*
```

那么读取器将把DEFVAR作为COMMON-LISP中的符号来读取，同时把\*x\*作为COMMON-LISP-USER中的符号来读取。

REPL不能在COMMON-LISP包中启动，因为你不能在这个包中添加新符号。COMMON-LISP-USER是作为一个“模板”包来提供的，在其中你可以创建自己的名字，同时还能轻松地访问COMMON-LISP的所有符号。<sup>②</sup>通常情况下，你定义的所有包都将使用COMMON-LISP，因此不需要写出类似下面的代码：

```
(cl:defun (x) (cl:+ x 2))
```

第三个标准包是KEYWORD包，这个包被Lisp读取器用来创建以冒号开始的名字。这样，你也可以使用显式的包限定方式来引用任何关键字符串：

① 每一个包都有一个正式名字以及零个或多个昵称（nickname），昵称可用在任何需要用到包名的地方，例如带有包限定的名字，或是在一个DEFPACKAGE或IN-PACKAGE形式中引用那个包。

② COMMON-LISP-USER也允许提供对从其他语言实现所定义的包导出的符号的访问。尽管这在本意上是为了给用户提供方便——它使得实现相关的功能易于访问，但它也给Lisp初学者带来许多疑惑：Lisp会抱怨重定义某些语言标准并未涉及的符号名。要想看到在一个特定的实现中COMMON-LISP-USER都从哪些包中继承了符号，可以在REPL中求值下列表达式：

```
(mapcar #'package-name (package-use-list :cl-user))
```

而要想查出一个符号最初来源于哪个包，可以求值下列表达式：

```
(package-name (symbol-package 'some-symbol))
```

同时把some-symbol替换成你想要的符号。例如：

```
(package-name (symbol-package 'car)) → "COMMON-LISP"
(package-name (symbol-package 'foo)) → "COMMON-LISP-USER"
```

继承自实现定义的包的符号将返回一些其他的值。

```
CL-USER> :a
:A
CL-USER> keyword:a
:A
CL-USER> (eql :a keyword:a)
T
```

## 21.4 定义你自己的包

使用COMMON-LISP-USER包对于在REPL中进行尝试是好的，不过一旦你开始编写实际的程序就会需要定义新的包，这样不同的程序可以加载到同一个Lisp环境中，而不会破坏彼此的名字。而当你编写可能用于不同环境中的库时，你会想要定义分开的包并导出那些构成了库的公共API的符号。

尽管如此，在开始定义包之前，理解包无法做到的一件事是很重要的。包无法提供对于谁可以调用什么函数或访问什么变量的直接控制。它们只提供对于名字空间的基本控制，这是通过控制读取器将文本名字转换成符号对象来完成的。但在后面求值器中，当符号被解释成一个函数、变量或其他任何东西的名字时，包机制就无能为力了。因此，谈论把一个函数或变量从一个包中导出是没有意义的。你可以导出符号以便特定的名字更加易于访问，但包系统并不允许你限制这些名字如何被使用。<sup>①</sup>

记住上述这件事，你可以开始学习如何定义包并将它们捆绑在一起了。你可以通过宏`DEFPACKAGE`来定义新的包，在创建包的同时还能指定它使用哪些包，导出哪些包，以及它从其他包里导入什么符号，还可以创建隐蔽符号来解决冲突。<sup>②</sup>

假设你正在使用包来编写一个将E-mail消息组织进一个可搜索数据库的程序，我将在此背景下描述所有有关的选项。这个程序是完全假想出来的，包括我将引用的其他库在内。要点在于，查看如何组织用于这样一个程序的包。

所需的第一个包是提供了整个应用程序命名空间的包，你需要命名你的函数、变量，等等，而无需担心和不相关的代码产生命名冲突。因此你最好用`DEFPACKAGE`定义一个新的包。

如果应用程序写得足够简单，没有用到超出语言本身所能提供的库，那么你可以定义一个如下的简单包：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

这段代码定义了一个包，名为`COM.GIGAMONKEYS.EMAIL-DB`，其继承了由COMMON-LISP包

<sup>①</sup> 这与Java的包系统不同，Java的包系统在提供类的名字空间的同时还引入了Java的访问控制机制。非Lisp语言中包系统最接近Common Lisp的语言是Perl。

<sup>②</sup> `DEFPACKAGE`进行的所有处理也都可以通过管理包对象的函数来完成。尽管如此，由于一个包在通常情况下都要在使用前被完全定义，所以这些函数很少用到。另外，`DEFPACKAGE`可以采用正确的顺序来完成所有的包管理操作。例如，`DEFPACKAGE`可以在使用那些用到的包之前将有关符号添加到隐蔽符号列表中。

导出的所有符号。<sup>①</sup>

事实上，有几种选择用于表示包的名字，如同你将看到的，**DEFPACKAGE**中符号的名字也有几种表示方法。包和符号都是用字符串来命名的。不过，在**DEFPACKAGE**形式中，可以使用字符串描述符（string designator）来指定包和符号的名字。字符串描述符要么是一个字符串，代表其自身；要么是一个符号，代表其名字；要么是一个字符，代表一个含有该字符的单字符串。像上面的**DEFPACKAGE**那样使用关键字符，是一种允许把名字书写成小写字母的常用风格，读取器将把名字转换成大写形式。也可以用字符串来书写**DEFPACKAGE**，若这样就需要全部使用大写形式。事实上，由于读取器的大小写转换约定，多数符号和包真正的名字都是大写的。<sup>②</sup>

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```

你也可以使用非关键字符——**DEFPACKAGE**中的名字不会被求值，但是随后读取**DEFPACKAGE**形式的操作将使这些符号进入到当前的包中，这在某种程度上泄露了名字空间，并可能也会在以后你试图使用该包时带来问题。<sup>③</sup>

为了读取这个包中的代码，你需要使用**IN-PACKAGE**宏来使其成为当前包：

```
(in-package :com.gigamonkeys.email-db)
```

如果你在REPL中输入这个表达式，它将改变\*PACKAGE\*的值，从而影响REPL读取后续表达式的方式，直到你通过另一个**IN-PACKAGE**调用来改变它。类似地，如果你在用**LOAD**加载的文件或**COMPILE-FILE**编译的文件中包含了一个**IN-PACKAGE**，那么它将改变当时的包，从而影响文件中后续表达式的读取方式。<sup>④</sup>

通过将当前包设置为**COM.GIGAMONKEYS.EMAIL-DB**包，除了那些继承自**COMMON-LISP**的名字以外，你几乎可以使用任何你想要使用的名字来实现任何目的。这样，你可以定义一个新的、与先前定义在**COMMON-LISP-USER**中的函数共存的**hello-world**函数。这是已有函数的行为：

```
CL-USER> (hello-world)
hello, world
NIL
```

<sup>①</sup> 在许多Lisp实现里，如果你只是使用**COMMON-LISP**包，那么**:use**子句是可选的。如果省略了它，包将自动从一个由具体实现所定义的包列表中继承名字，而这个包列表通常都包括**COMMON-LISP**。尽管如此，如果总是显式地指定你想要使用的包列表，那么代码将会变得更加具有可移植性。那些不愿意打字的人可以使用包的昵称，从而将其写成(**:use :cl**)。

<sup>②</sup> 使用关键字来代替字符串还有另一个优点：Allegro支持一种“现代模式”的Lisp，其中读取器并不做大小写转换，并且在带有大写名称的**COMMON-LISP**包以外，它还提供了一个使用小写字母的**common-lisp**包。严格来讲，这种Lisp并不符合Common Lisp标准，因为所有标准中定义的名字都是被定义成大写的。但如果你使用关键字符来编写**DEFPACKAGE**形式，那么它将可以同时工作在Common Lisp和与之相近的另一种模式下。

<sup>③</sup> 一些人使用“#：“语法的未进入的符号来代替关键字符。这通过在关键字包中未进入任何符号来节省一小部分内存，在**DEFPACKAGE**通过它实现（或代码展开）之后，符号会变成垃圾。然而，差异是如此微小，以至于这被归结为美学范畴。

<sup>④</sup> 使用**IN-PACKAGE**而不是仅仅用**SETF**来修改\*PACKAGE\*的原因在于，**IN-PACKAGE**可以展开成在文件被**COMPILE-FILE**编译时和文件加载时都会执行的代码，从而在编译期就可以改变读取器读取文件其余部分的方式。

现在你可以用`IN-PACKAGE`切换到新的包上。<sup>①</sup>注意提示符的改变——具体的形式取决于开发环境，但在SLIME中默认提示符由包名的简化版本构成。

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

你可以在这个包里定义一个新的hello-world：

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

然后用如下方式测试它：

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

现在切换回CL-USER：

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

旧的函数行为没有被干扰：

```
CL-USER> (hello-world)
hello, world
NIL
```

## 21.5 打包可重用的库

尽管工作在E-mail数据库上，但你可能需要编写几个与存取文本相关而与E-mail本身无关的函数。你可能意识到这些函数会对其他程序有用，并且决定将它们重新打包成一个库。你应当定义一个新的包，这次将导出一些特定的名字，从而使其对于其他包可见。

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

你再次使用了COMMON-LISP包，因为你需要在COM.GIGAMONKEYS.TEXT-DB中访问标准函数。`:export`子句指定了COM.GIGAMONKEYS.TEXT-DB外部的名字，这些名字可以被所有使用它的包直接访问。因此，在定义了这个包以后，你可以将主应用程序包的定义作如下改变：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

---

<sup>①</sup> 在SLIME的REPL缓冲区里，你也可以使用REPL快捷键来改变当前包。键入一个逗号，然后在Command:提示下输入`change-package`。

现在COM.GIGAMONKEYS.EMAIL-DB中编写的代码可以同时使用非限定名字来引用COMMON-LISP和COM.GIGAMONKEYS.TEXT-DB中导出的符号。所有其他的名字都将继续直接进入COM.GIGAMONKEYS.EMAIL-DB包中。

## 21.6 导入单独的名字

现在假设你找到了一个可以处理E-mail消息的第三方函数库。该库的API中所使用的名字是从COM.ACME.EMAIL包中导出的，因此你可以通过使用这个包来轻松获得对这些名字的访问权限。但假设你只需要用到这个库的一个函数，且其他的导出符号跟你已经使用（或计划使用）的符号有冲突。<sup>①</sup>在这种情况下，你可以使用`DEFPACKAGE`中的`:import-from`子句只导入所需要的那一个符号。例如，如果想要使用的函数的名字为`parse-email-address`，那么可以把`DEFPACKAGE`作如下改写：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

现在COM.GIGAMONKEYS.EMAIL-DB包中任何代码里出现的`parse-email-address`都将被读取为COM.ACME.EMAIL中的同名符号。如果需要从单个包中导入多个符号，那只需在单个`:import-from`子句中的包名之后包含多个名字即可。一个`DEFPACKAGE`也可以含有多个`:import-from`子句，以便从不同的包中分别导入符号。

有时你可能会遇到相反的情况——一个包可能导出了一大堆你想要使用的名字，但还有少数是你不需要的。不用将所有你需要的符号都列在一个`:import-from`子句中，你可以直接使用这个包，同时把不需要继承的符号放在一个`:shadow`子句里。例如，假设COM.ACME.TEXT包导出了许多用于文本处理的函数和类的名字，更进一步假设多数这些函数和类都是你想要用在代码中的，但其中一个名字`build-index`跟你已经使用的名字冲突了。你可以通过隐蔽这个符号将来自COM.ACME.TEXT的`build-index`符号设置为不可见的。

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index))
```

这个`:shadow`子句导致创建了一个新的名为BUILD-INDEX的符号，且这个符号直接添加到COM.GIGAMONKEYS.EMAIL-DB的名字-符号映射表中。如果读取器读到了名字BUILD-INDEX，将把它转化成COM.GIGAMONKEYS.EMAIL-DB表中的符号，而不是继承自COM.ACME.TEXT的那个符号了。这个新的符号也会添加到作为COM.GIGAMONKEYS.EMAIL-DB包的一部分的隐蔽符号列

<sup>①</sup> 在开发过程中，如果你试图使用一个包，它导出了一些与你当前所在包的已有符号同名的符号，那么Lisp将会立即报错并通常会提供给你一个再启动，把所用到的包中的那个符号去掉。关于这个过程的更多细节，请参见21.8节。

表中。所以，如果你以后使用了另一个同样导出了BUILD-INDEX符号的包，包系统将会知道这里没有冲突，也就是说你总是希望使用来自COM.GIGAMONKEYS.EMAIL-DB的符号，而不是从其他包里继承的同名符号。

当你想要使用两个导出了同样名字的包时，一个类似的情形出现了在这种情况下，读取器在读到文本名字时不可能知道你究竟想要使用哪一个继承的符号，你必须通过隐蔽有冲突的名字来消除歧义。如果你根本就不需要使用这个名字，那可以通过:shadow子句将该名字屏蔽掉，从而在包中创建出一个同名的新符号来。但如果你确实想要使用一个继承来的符号，那么你需要通过:shadowing-import-from子句来消除歧义。和:import-from子句一样，:shadowing-import-from子句由一个包名及紧接着需要从那个包中导入的名字所构成。举个例子，如果COM.ACME.TEXT导出了一个名字SAVE，它与从COM.GIGAMONKEYS.TEXT-DB中导出的名字相冲突，那么可以使用如下的`DEFPACKAGE`形式来消除歧义：

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

## 21.7 打包技巧

前面已经讨论了一些常见情形下用包来管理名字空间的方法，而关于如何使用包的另一层面内容也是值得讨论的——关于如何组织代码来使用不同的包的基本技巧。在本节里，我将讨论一些关于如何组织代码的概括性规则，也即相对于那些通过`IN-PACKAGE`来使用包的代码来说，应该在哪里保存`DEFPACKAGE`形式呢？

因为包要被读取器使用，因此一个包必须在加载或编译一个含有切换到该包的`IN-PACKAGE`表达式的文件之前就被定义。包也必须定义在可能用到它的其他`DEFPACKAGE`形式之前。举个例子，如果你打算在COM.GIGAMONKEYS.EMAIL-DB中使用COM.GIGAMONKEYS.TEXT-DB包，那么COM.GIGAMONKEYS.TEXT-DB的`DEFPACKAGE`必须在COM.GIGAMONKEYS.EMAIL-DB的`DEFPACKAGE`之前被求值。

确保包在它们被用到之前总是存在的，最佳方法是把所有的`DEFPACKAGE`定义放在与需要在这些包里读取的源代码分开的文件里。一些人喜欢针对每个单独的包都创建一个形如`foo-package.lisp`的文件，而另一些人则创建单个`packages.lisp`来包含一组相关的包的所有`DEFPACKAGE`形式。两种思路都是合理的，尽管每个包一个文件的方法也对组织并根据包之间的依赖关系以正确的顺序加载单独的文件提出了要求。

不论用哪种方式，一旦所有的`DEFPACKAGE`形式都从那些用到它们的代码中分离出来了，你就可以调整加载文件的顺序，让那些含有`DEFPACKAGE`的文件在编译或加载任何其他文件之前进

行加载。对于简单的程序，可以手工完成这件事：简单地加载那些含有`DEFPACKAGE`形式的文件，其中有可能需要先用`COMPILE-FILE`编译。然后加载使用这些包的文件，可再次预先选用`COMPILE-FILE`编译它们。不过，需要注意的是，直到你用`LOAD`加载那些以源代码的形式或是`COMPILE-FILE`输出的文件形式存在的包定义之前，包都是不存在的。因此，如果你正在编译所有的文件，仍然必须先加载所有的包定义，然后再编译那些需要从这些包里读取符号的文件。

完全手工来做这些事很快就会令人厌烦。对于简单的程序，你可以通过编写一个`load.lisp`文件自动完成这些步骤，该文件里含有以正确顺序排列的适当的`LOAD`和`COMPILE-FILE`调用，然后你只需加载那个文件就好了。对于更复杂的程序，你会希望使用一种系统定义功能（system definition facility）以正确的顺序来加载和编译文件。<sup>①</sup>

另一个关键的概括性规则是每个文件里应该只含有一个`IN-PACKAGE`形式，并且它应当是该文件中除注释以外的第一个形式。含有`DEFPACKAGE`形式的文件应当以`(in-package "COMMON-LISP-USER")`开始，而所有其他的文件都应当含有一个属于某个包的`IN-PACKAGE`形式。

如果你违反了这个规则，在文件中间切换当前包，那么就会迷惑那些没有注意到第二个`IN-PACKAGE`的读者。另外，许多Lisp开发环境，尤其是诸如SLIME这种基于Emacs的环境，是通过查看`IN-PACKAGE`来决定与Common Lisp通信时所使用的包的。如果每个文件里有多个`IN-PACKAGE`，也可能会干扰这些工具。

另一方面，多个文件以相同的包来读取，每个文件都使用相同的`IN-PACKAGE`形式，这是没有问题的。问题只是你想要怎样组织代码。

关于打包技巧的最后一点是如何给包命名。所有包名都存在于扁平的名字空间里——包名只是字符串，而不同的包必须带有文本上可区分的名字。这样，你就得考虑包名字冲突的可能性。如果你只使用自己开发的包，那么可以随意地使用短名称。但如果你正在计划使用第三方库或是发布你的代码给其他程序员使用，那么需要遵守一个可以令不同包之间名字冲突最小的命名约定。最近，许多程序员都采用了Java风格的命名方式，如同本章里使用的那些包名一样，它由一个逆向的Internet域名后跟一个点和一个描述性的字符串组成。

## 21.8 包的各种疑难杂症

一旦你熟悉了包，就不会再花许多时间来思考它们了。其实本来也没什么可思考的。不过，一些困扰初级Lisp程序员的疑难杂症，使得包系统看起来比它实际的情况更加复杂和不友好了。

排名第一的疑难杂症通常出现在使用REPL的时候。当你正在寻找一些定义了特别感兴趣的函数的库时，你试图像下面这样调用它们中的一个：

```
CL-USER> (foo)
```

然后，伴随以下错误信息，程序会进入调试器：

21

<sup>①</sup> 所有可通过本书的Web站点获得的那些来自“实践”章节的代码，都使用了ASDF系统定义库。我将在第32章里讨论ASDF。

```

attempt to call 'FOO' which is an undefined function.
[Condition of type 'UNDEFINED-FUNCTION']
Restarts:
 0: [TRY AGAIN] Try calling FOO again.
 1: [RETURN-VALUE] Return a value instead of calling FOO.
 2: [USE-VALUE] Try calling a function other than FOO.
 3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
 4: [ABORT] Abort handling SLIME request.
 5: [ABORT] Abort entirely from this (lisp) process.

```

啊！原来你忘了使用那个库的包。于是你退出调试器并试图使用该库的包来得到对名字FOO的访问，然后再调用该函数。

```
CL-USER> (use-package :foolib)
```

但这次出现了以下错误信息，并再次进入了调试器：

```

Using package 'FOOLIB' results in name conflicts for these symbols: FOO
[Condition of type PACKAGE-ERROR]
Restarts:
 0: [CONTINUE] Unintern the conflicting symbols from the 6→
 'COMMON-LISP-USER' package.
 1: [ABORT] Abort handling SLIME request.
 2: [ABORT] Abort entirely from this (lisp) process.

```

啊？问题在于第一次调用foo的时候，读取器读取名字foo，在求值器接手并发现这个新创建的符号不是一个函数的名字之前就使其进入了CL-USER包。这个新的符号随后又和FOOLIB包中导出的同名符号相冲突了。如果你在试图调用foo之前记得`USE-PACKAGE FOOLIB`，那么读取器就可以将foo读取成一个继承而来的符号，从而就不会在CL-USER中创建一个新符号了。

不过，现在还为时不晚，因为调试器给出的第一个再启动将会以正确的方式来修复：它将使foo符号从COMMON-LISP-USER中退出，把CL-USER包恢复到调用foo之前的状态，从而使`USE-PACKAGE`得以进行并使继承来的foo在CL-USER中可用。

这类问题也会发生在加载和编译文件时。举个例子，如果你定义了一个包MY-APP，其中的代码打算使用来自包FOOLIB的函数名字，但是当你在`(in-package :my-app)`下编译文件时却忘记了`:use FOOLIB`，这样，读取器会将这些原本打算从FOOLIB中读取的符号改为在MY-APP中创建新符号。当你试图运行编译后的代码时，就会得到函数未定义的错误。如果你随后试图重定义MY-APP包，加上`:use FOOLIB`，那么就会得到符号冲突的错误。解决方案是一样的：选择再启动来从MY-APP中消除有冲突的符号。然后，你需要重新编译MY-APP包中的代码，这样它们就可以指向那些继承的名字了。

下一个疑难杂症本质上是第一个的相反形式。在这种情况下，你已经定义了一个用到了诸如FOOLIB的包，再次假设它是MY-APP。现在你开始编写MY-APP中的代码。尽管你使用FOOLIB是为了引用foo函数，但FOOLIB同时还导出了其他一些符号。如果你把其中一个导出的诸如bar的符号用作了自己代码里的某个函数的名字，那么Lisp就不会报错。相反，你的函数的名字将会是FOOLIB导出的那个符号，这会破坏FOOLIB中bar的定义。

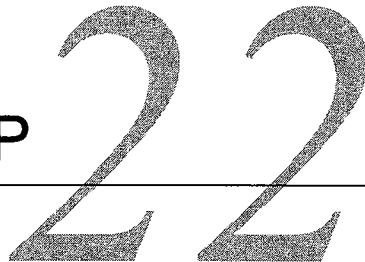
这个问题的危害更大，因为它不会明显地报错。从求值器的观点来看，这只是要求将一个新

函数关联到一个旧的名字上，有时这是完全合法的。唯一的可疑之处只在于，做这个重定义的代码是在一个与函数名符号所在的包名不同的\*PACKAGE\*下读取的，但求值器不必关心这点。不过在多数Lisp环境下，你会得到一个关于“redefining BAR, originally defined in ...”的警告。应当留心那些警告。如果你破坏了一个库中的定义，那么可以通过使用LOAD重新加载库的代码来恢复它。<sup>①</sup>

最后一个包相关的疑难杂症相对来说比较简单，但它给多数Lisp程序员至少带来了几次麻烦：你定义了一个使用COMMON-LISP同时还可能有其他一些库的包。然后，在REPL中你切换到那个包里去做一些事。紧接着你决定完全退出Lisp环境并试图调用(quit)。不过，quit并不是来自COMMON-LISP包的名字，它被具体实现定义在了某个实现相关的包里了，后者往往是COMMON-LISP-USER包所使用的。解决方案很简单：切换回CL-USER包然后再退出，或者使用SLIME的REPL快捷键quit，它可以使你免去记忆特定Common Lisp实现的退出函数究竟是exit还是quit的烦恼。

至此，你基本完成了对Common Lisp的了解。在下一章里，我将讨论扩展形式的LOOP宏的细节。在那之后，本书的其余部分都将面向“实践”：一个垃圾过滤器、一个用来解析二进制文件的库以及一个带有Web接口的流式MP3服务器的各部分。

<sup>①</sup> 某些Common Lisp实现，包括Allegro和SBCL，提供了一种“锁定”特定包中符号的机制，这一机制可以确保只有当诸如DEFUN、DEFVAR和DEFCLASS这类定义形式所在的主包是当前包时才能顺利通过。



**第**7章曾简要讨论了扩展形式的LOOP宏。正如当时提到的，LOOP本质上提供了用来编写迭代构造的专用语言。

这看起来可能有些无聊——发明一种全新的语言只为编写循环。但如果你深入思考循环结构在程序中的各种用法，就会发现事实上这样做是合理的。任何规模的程序都会包含相当数量的循环。虽然它们不都是完全一样的，但也不是完全不同的。循环里有很多固定的编码模式，尤其是在循环前后包含代码时——为循环做准备的模式，确保循环过程所需的模式，以及当循环结束时所需操作的模式。LOOP语言捕捉了这些模式，让你可以直接表达各种循环。

LOOP宏由许多部分组成——LOOP反对者们的一个主要论点就是说它过于复杂。在本章里，我将从头介绍LOOP，系统化地介绍它的多个组成部分以及彼此间配合使用的方式。

## 22.1 LOOP 的组成部分

你可以在一个LOOP中做下列事情：

- 以数值或多种数据结构为步长来做循环；
- 在循环的过程中收集、计数、求和以及求最大值或最小值；
- 执行任意Lisp表达式；
- 决定何时终止循环；
- 条件执行上述内容。

另外，LOOP还提供了用于下列事务的语法：

- 创建用于循环内部的局部变量；
- 指定任意Lisp表达式在循环开始前和结束后运行。

LOOP的基本结构是一个子句集合，其中每个子句都以一个循环关键字（loop keyword）开始。<sup>①</sup>每个子句被LOOP宏解析的方式取决于具体的关键字。第7章已经介绍了一些主要的关键字，包括for、collecting、summing、counting、do以及finally。

<sup>①</sup> “循环关键字”这个名字取得并不是很好，因为循环关键字并不是正常意义上的KEYWORD包中的关键字。事实上，来自任何包的任何符号，只要有适当的名字就可以了，LOOP宏只关心它们的名字。不过，通常情况下它们都被写成不带有包限定符的形式并在当前包下被读取（必要时会创建新符号）。

## 22.2 迭代控制

大多数所谓的迭代控制子句都以循环关键字 `for` 或是它的同义词 `as`<sup>①</sup> 开始，后接一个变量的名字。变量名后面的内容取决于 `for` 子句的类型。

一个 `for` 子句的下级子句可以对下列内容进行迭代：

- 数字范围（以指定的间隔向上或向下）；
- 由单独的项组成的列表；
- 构成列表的点对单元；
- 向量的元素（包括诸如字符串和位向量这样的向量子类型）；
- 哈希表的键值对；
- 一个包中的符号；
- 对给定形式反复求值得到的结果。

循环可以含有多个 `for` 子句，其中每个子句都可以命名其自己的循环变量。如果循环含有多个 `for` 子句，其中任何一个子句达到结束条件循环都会终止。例如，下面的循环：

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

将迭代至多10次，但在 `list` 含有少于10项时会提前终止。

## 22.3 计数型循环

算术迭代子句可以控制循环体的执行次数，它通过在一个整数范围上步进来到做到这点，每前进一步就执行一次循环体。这些子句由 `for`（或 `as`）之后紧跟下列介词短语中的1~3个构成：起始短语、终止短语以及步长短语。

起始短语指定了该子句的变量初始值。它由介词 `from`、`downfrom` 或 `upfrom` 之一后接一个提供初值（一个数字）的形式所构成。

终止短语指定了循环的终止点。它由介词 `to`、`upto`、`below`、`downto` 或 `above` 之一后接一个提供终值的形式所构成。当使用 `upto` 和 `downto` 时，循环体将在变量通过终止点时终止（通过以后不会再次求值循环体）；而当使用 `below` 和 `above` 时，它会提前一次迭代终止循环。

步长短语由介词 `by` 和一个形式所构成，该形式必须求值为一个正数。变量将按照该数在每次迭代时步进（向上或向下，取决于其他短语）或是在其默认时每次步进1。

你必须至少指定一个上述介词短语。默认值是从零开始，每次迭代时加1，然后一直加下去，或者更有可能的是直到其他某个子句终止了循环。你可以通过添加适当的介词短语来修改这些默认值中的任何一个或全部。唯一需要注意的是如果要逐步递减的话，不存在默认的初始值，必须

22

<sup>①</sup> 因为当初 **LOOP** 的目标之一就是允许循环表达式可以被写成类似英语的语法，所以许多关键字都有一些同义词，它们对于 **LOOP** 来说处理方法相同，但在不同的语境下可以更接近英语的语法习惯。

使用`from`或`downfrom`来指定一个。因此，形式：

```
(loop for i upto 10 collect i)
```

会收集到11个整数（从零到十），而下面这个形式的行为则是未定义的：

```
(loop for i downto -10 collect i) ; wrong
```

以下是一种正确的写法：

```
(loop for i from 0 downto -10 collect i)
```

另外要注意，由于`LOOP`是一个宏，运行在编译期，它需要能够完全基于这些介词而不是一些形式的值来决定变量步进的方向，因为形式的值到运行期才会知道。因此，形式

```
(loop for i from 10 to 20 ...)
```

可以工作得很好，因为默认就是递增步进。但形式：

```
(loop for i from 20 to 10 ...)
```

将不知道是从20向下数到10。更糟糕的是，它不会给你报错——由于*i*已经大于10了，所以循环根本不会执行。此时，你必须写成：

```
(loop for i from 20 downto 10 ...)
```

或是

```
(loop for i downfrom 20 to 10 ...)
```

最后，如果你只是想让一个循环重复特定的次数，那么可以将下列形式中的一个子句

```
for i from 1 to number-form
```

替换成如下所示的一个`repeat`子句：

```
repeat number-form
```

这些子句在效果上是等价的，只是`repeat`子句没有创建显式的循环变量。

## 22.4 循环集合和包

用于迭代列表的`for`子句比算术子句更简单一些。这种子句只支持两个介词短语，`in`和`on`。一个下列形式的短语

```
for var in list-form
```

将在求值*list-form*所产生的列表的所有元素上推动变量*var*。

```
(loop for i in (list 10 20 30 40) collect i) → (10 20 30 40)
```

有时这个子句会在一个`by`短语的辅助下使用，`by`短语指定了一个用来在列表中向下移动的函数，其默认值是`CDR`，但它可以是任何接受一个列表并返回其子列表的函数。例如，可以像下面这样收集一个列表中相隔的元素：

```
(loop for i in (list 10 20 30 40) by #'cddr collect i) → (10 30)
```



on介词短语被用来在构成列表的点对单元上步进变量var。

```
(loop for x on (list 10 20 30) collect x) → ((10 20 30) (20 30) (30))
```

该短语也接受一个介词by：

```
(loop for x on (list 10 20 30 40) by #'caddr collect x) → ((10 20 30 40) (30 40))
```

循环向量（包括字符串和位向量）的元素与循环列表的元素类似，只是要使用介词across来代替in。<sup>①</sup>例如：

```
(loop for x across "abcd" collect x) → (#\a #\b #\c #\d)
```

迭代哈希表或包稍微复杂一些，因为哈希表和包可能包含需要迭代的不同值的集合——哈希表中的键或值，以及包中不同类型的符号。两种迭代都遵循相同的模式。基本的模式如下所示：

```
(loop for var being the things in hash-or-package ...)
```

对于哈希表来说，things的可能值是hash-keys和hash-values，它们使var与哈希表中连续的键或值绑定。hash-or-package形式只被求值一次并产生一个值，它必须是一个哈希表。

要想在包上迭代，things可以是symbols、present-symbols和external-symbols，它们使var被绑定到包的每个可访问的符号、当前存在的符号（即在包里创建的或导入进该包的符号），或是每一个从该包中导出的符号上。hash-or-package被求值并产生一个包的名字（这会导致用FIND-PACKAGE来查找）或包对象本身。for子句的许多部分也可用同义词。在the的位置上可用each，你还可以用of来代替in，另外你也可以将things写成单数形式（例如，hash-key或symbol）。

最后，由于你经常需要在一个哈希表上同时迭代键和值，哈希表子句还在其结尾处支持using子句。

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

这两个循环都可以将k绑定到哈希表的每个键上，再把v绑定到对应的值上。注意using子句的第一个元素必须写成单数形式。<sup>②</sup>

## 22.5 等价-然后迭代

如果其他的for子句都无法确切支持你所需要的变量步进形式，那么你可以通过等价-然后>equals-then) 子句来完全控制步进的方式。这个子句跟DO循环中的绑定语句很相似，但更接近Algol语言的语法。完整的形式如下所示：

<sup>①</sup> 你可能想知道为什么LOOP不使用同样的介词，然后自己检查当前究竟是循环列表还是向量。这是LOOP作为一个宏所带来的另一个后果：列表或者向量的值直到运行期才会知道，但LOOP作为一个宏必须在编译期生成代码，并且LOOP的设计者们想要它生成极其高效的代码。为了能够生成用来访问的高效的代码，比如说一个向量，它需要在编译期知道这个值在运行期将是一个向量。因此，需要采用不同的介词。

<sup>②</sup> 不要问我为什么LOOP的设计者们没有使用不带括号的风格来表示using子句。

```
(loop for var = initial-value-form [ then step-form ] ...)
```

和通常一样，*var*是需要步进的变量名。它的初值在首次迭代之前通过求值*initial-value-form*而获取到。在每一次后续迭代中，*step-form*被求值，然后它的值成为了*var*的新值。如果子句中没有*then*部分，那么*initial-value-form*将在每次迭代中重新求值以提供新值。注意，这和一个没有步长形式的DO绑定子句的行为是不同的。

*step-form*可以引用其他的循环变量，包括由循环中其他后续for子句所创建的循环变量。例如：

```
(loop repeat 5
  for x = 0 then y
  for y = 1 then (+ x y)
  collect y) → (1 2 4 8 16)
```

不过，每个for子句都是以它们各自出现的顺序来逐个求值的，因此在前面的循环中，第二次迭代时x会在y改变（变成1）之前被设置为y的值。但是y随后会被设置为它的旧值（1）与x的新值之和。如果for子句的顺序反过来，那么结果将会改变。

```
(loop repeat 5
  for y = 1 then (+ x y)
  for x = 0 then y
  collect y) → (1 1 2 4 8)
```

不过，通常你都想让多个变量的步长形式在任何变量被赋予新值之前计算完毕（类似于DO步进其变量的方式）。在这种情况下，你可以将多个for子句连在一起，将除第一个以外的for全部替换成and。你已经在第7章通过LOOP计算Fibonacci的示例里见过它的公式了。这是基于前面两个例子的另一个变体：

```
(loop repeat 5
  for x = 0 then y
  and y = 1 then (+ x y)
  collect y) → (1 1 2 3 5)
```

## 22.6 局部变量

尽管循环所需要的主要变量通常都会在for子句中隐式声明，但有时也需要额外的变量，这些变量可以使用with子句来声明。

```
with var [= value-form]
```

*var*将成为一个局部变量的名字，它会在循环结束时被删除。如果with子句含有一个 = *value-form*部分，那么变量将会在循环的首次迭代之前初始化为*value-form*的值。

多个with子句可以同时出现在一个循环里。每个子句将根据其出现的顺序独立求值，并且其值将在处理下一个子句之前完成赋值，从而允许后面的变量可以依赖于已经声明过的变量。完全无关的变量可以用and连接每个声明并放在一个with子句中。

## 22.7 解构变量

一个尚未谈及的LOOP宏的非常有用的特性，是其解构赋值给循环变量的列表值的能力。这可以让你取出赋值给循环变量的列表里的值，类似于DESTRUCTURING-BIND的工作方式但没有那么复杂。基本上，你可以将任何出现在for或with子句中的循环变量替换成一个符号树，这样，原本赋值到简单变量上的列表值将改为解构到以树中的符号所命名的变量上。一个简单的例子如下所示：

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
      do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

这棵树还可以包含带点的列表，这时点之后的名字将像一个`&rest`参数那样处理，被绑定到列表的其余元素上。这对于for/on类循环来说特别有用，因为值总是一个列表。例如，下面这个循环（第8章曾用它来输出一个逗号分隔的列表）：

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ", "))
```

也可以写成这样：

```
(loop for (item . rest) on list
      do (format t "~a" item)
      when rest do (format t ", "))
```

如果你想要忽略一个解构列表中的值，可以用NIL代替相应变量的名字。

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) → (1 3 5)
```

如果解构列表含有比列表中的值更多的变量，那么多余的变量将被设置为NIL，这使得所有变量本质上都像是`&optional`参数。不过，没有任何跟`&key`参数等价的东西。

## 22.8 值汇聚

值汇聚（value accumulation）语句可能是LOOP中最有用的部分。尽管迭代控制语句提供了一个表示循环基本结构的简洁的语法，但它们本质上与DO、DOLIST和DOTIMES所提供的机制是等价的。

另一方面，值汇聚子句为循环过程中涉及值汇聚的常见循环用法提供了一套简洁表示法。每个汇聚子句都以一个动词后接下列模式开始：

```
verb form [ into var ]
```

每次通过循环时，汇聚子句会对form求值并将其按照由verb所决定的方式保存起来。通过into子句，这些值被保存在名为var的变量里。该变量对于循环来说是局部的，就像它是被一个

with子句所声明的那样。如果没有into下级子句，那么汇聚子句将汇聚出一个作为整个循环表达式返回值的默认值。

可用的动词包括collect、append、nconc、count、sum、maximize和minimize。还有它们对应的进行时形式的同义词：collecting、appending、nconcing、counting、summing、maximizing和minimizing。

collect子句会构造一个列表，列表中包含以代码中的顺序排列的所有form的值。这是一个特别有用的构造，因为手工编写一个像LOOP那样有效率的列表来收集代码非常困难。<sup>①</sup>与collect相关的动词是append和nconc。这两个动词都将值汇聚到一个列表上，但它们所汇聚的值本身也必须是列表，然后像函数APPEND或NCONC<sup>②</sup>那样将所有列表汇聚成单个列表。

其余的汇聚子句都用来汇聚数值。动词count统计form为真的次数，sum收集所有form的值之和，maximize收集它所看到的最大值，而minimize则收集最小值。例如，假设你定义了一个变量\*random\*，它含有一个随机数列表。

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

那么下面的循环将返回关于这些数的一个含有多种统计信息的列表：

```
(loop for i in *random*
      counting (evenp i) into evens
      counting (oddp i) into odds
      summing i into total
      maximizing i into max
      minimizing i into min
      finally (return (list min max total evens odds)))
```

<sup>①</sup> 难点在于必须跟踪列表的尾部并通过SETF尾部的CDR来向列表添加新的点对。一个由(loop for i upto 10 collect i)所生成代码的手写等价版本如下所示：

```
(do((list nil) (tail nil) (i 0 (1+ i)))
    ((> i 10) list)
    (let ((new (cons i nil)))
        (if (null list)
            (setf list new)
            (setf (cdr tail) new))
        (setf tail new)))
```

当然，你很少需要编写像这样的代码。可以使用LOOP，也可以（如果出于某种原因你不想使用LOOP的话）使用标准的收集值的PUSH/NREVERSE。

<sup>②</sup> 回顾一下，NCONC是APPEND的破坏性版本——安全使用nconc子句的场合仅限于你正在收集的值都是全新的列表，而且该列表没有跟其他列表共享任何结构。例如，下面的代码是安全的：

```
(loop for i upto 3 nconc (list i i)) → (0 0 1 1 2 2 3 3)
```

而这个将给你带来麻烦：

```
(loop for i on (list 1 2 3) nconc i) → undefined
```

它将很可能进入一个无限循环，因为由(list 1 2 3)所产生的列表被破坏性地修改以指向它自身。其实这个行为也难以保证，因为其行为根本没有定义。

## 22.9 无条件执行

虽然值汇聚构造非常有用，但如果有机会在循环体中执行任意代码的话，LOOP就不是一个很好的通用迭代机制了。

在一个循环之内执行任意代码的最简单方式是使用do子句。与之前讨论过的那些带有介词和进一步子句的其他子句相比，do具有一种Yoda式的简洁性。<sup>①</sup>一个do子句由单词do（或doing）后接一个或多个Lisp形式构成，这些形式将在do子句开始运行时全部被求值。do子句结束于一个循环的闭合括号或是下一个循环关键字。

例如，为了打印出数字1到10，可以这样来写：

```
(loop for i from 1 to 10 do (print i))
```

另一个更有趣的立即执行的形式是return子句。这个子句由单词return后接单个Lisp形式组成，当该形式被求值时，得到的结果将立即作为整个循环的值返回。

使用Lisp的常规控制流操作符，例如RETURN和RETURN-FROM，也可以从循环中的do子句里跳出。注意return子句总是从临近的LOOP表达式里返回，而do子句中的RETURN或RETURN-FROM可以从任意封闭的表达式中返回。举个例子，比较下列表达式

```
(block outer
  (loop for i from 0 return 100) ; 100 returned from LOOP
  (print "This will print")
  200) → 200
```

和

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 returned from BLOCK
  (print "This won't print")
  200) → 100
```

上述do和return子句统称为无条件执行子句。

## 22.10 条件执行

由于do子句可以包含任意Lisp形式，所以可以在这里使用任何Lisp表达式，包括IF和WHEN这样的控制构造。下面就是只打印1到10之间所有偶数的一种循环的写法：

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

不过，有时你会需要循环子句层面上的条件控制。例如，假设你只想用一个summing子句来求和从1到10之间的偶数。你不可能用一个do子句来写出这样的循环，因为没有办法在一个正规的Lisp形式里“调用”sum i。对于类似这种情况，就需要用到LOOP自己的条件表达式，如下所示：

<sup>①</sup> “No! Try not. Do ... or do not. There is no try.”（不！不要试。要么做要么不做。没有机会可试）——Yoda, *The Empire Strikes Back*（《星球大战5》）。

```
(loop for i from 1 to 10 when (evenp i) sum i) → 30
```

LOOP提供了三种条件构造，它们全部遵循下面的基本模式：

*conditional test-form loop-clause*

其中的*conditional*可以是if、when或unless，*test-form*可以是任何正规Lisp形式，而*loop-clause*则可以是一个值汇聚子句（count、collect，等等）、一个无条件执行子句或是另一个条件执行子句。多个循环子句可以通过and连接成单一条件。

还有一点儿语法糖：在第一个循环子句里，测试形式之后，可以使用变量it来指代由测试形式所返回的值。例如，下面的循环可以收集在列表some-list中查找键时所找到的在哈希表some-hash中对应的非空值：

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

条件子句在每次通过循环时都会执行。if或when子句会在它的*test-form*求值为真时执行其*loop-clause*。unless子句可以把测试反过来，仅当*test-form*为NIL时才执行*loop-clause*。LOOP的if和when关键字Common Lisp中同名的关键词含义不同，在这里它们是同义词——行为上没有区别。

下面这个相当傻的循环演示了几种不同形式的LOOP条件子句。函数update-analysis将在每次通过循环时被调用，其参数是条件子句中的汇聚子句最后更新的不同变量的值。

```
(loop for i from 1 to 100
      if (evenp i)
          minimize i into min-even and
          maximize i into max-even and
          unless (zerop (mod i 4))
              sum i into even-not-fours-total
          end
          and sum i into even-total
      else
          minimize i into min-odd and
          maximize i into max-odd and
          when (zerop (mod i 5))
              sum i into fives-total
          end
          and sum i into odd-total
      do (update-analysis min-even
                           max-even
                           min-odd
                           max-odd
                           even-total
                           odd-total
                           fives-total
                           even-not-fours-total))
```

## 22.11 设置和拆除

LOOP语言设计者早就预见到：循环在实际使用中总是以一些设置初始环境的代码开始，循

环结束后还会有更多的代码来处理由循环所计算出来的值。举一个简单的Perl例子<sup>①</sup>，如下所示：

```
my $evens_sum = 0;
my $odds_sum = 0;
foreach my $i (@list_of_numbers) {
    if ($i % 2) {
        $odds_sum += $i;
    } else {
        $evens_sum += $i;
    }
}
if ($evens_sum > $odds_sum) {
    print "Sum of evens greater\n";
} else {
    print "Sum of odds greater\n";
}
```

这段代码中的循环是foreach语句。但foreach本身并不能独立工作：循环体中的代码引用了循环开始前的两行代码中声明的变量。<sup>②</sup>而循环所做的所有工作假如果没有了后面那个if语句的话也就毫无意义了，if语句在循环结束后输出结果。在Common Lisp中，LOOP结构也是一个可以返回值的表达式，因此通常更需要做的一件事是在循环结束之后生成一个有用返回值。

所以，LOOP的设计者说，应该提供一种方式将原本应该放在循环中的那些代码也塞进循环里。这样，LOOP就提供了两个关键字，initially和finally，用于引入那些原本会运行在循环主体以外的代码。

在initially或finally之后，这些子句由所有需要在下一个循环子句开始之前或者循环结束之后运行的多个Lisp形式所组成。所有的initially形式会被组合成一个“序言”，在所有局部循环变量被初始化以后和循环体开始之前运行一次。所有的finally形式则被简单地组合成一个“尾声”，在循环体的最后一次迭代结束以后运行。序言和尾声部分的代码都可以引用局部循环变量。

就算循环迭代了零次，序言部分也总是会运行。但是循环有可能在下列任何情况发生时不会运行尾声：

- 执行了一个return子句。
- RETURN、RETURN-FROM或其他控制构造的传递操作在循环体中的一个Lisp形式中被调用<sup>③</sup>。
- 循环被一个always、never或thereis子句终止，我将在下一节里讨论这种情况。

在尾声部分的代码中，RETURN或RETURN-FROM可被用来显式提供一个循环的返回值。这个显式的返回值将比其他汇聚或终止测试子句所提供的值具有更高的优先级。

另外，为了使RETURN-FROM从一个特定的循环中返回（这在嵌套的LOOP表达式中是有用的），

<sup>①</sup> 我并不是故意选择Perl的，这个例子在任何语法基于C的语言里看起来都差不多。

<sup>②</sup> 在Perl里，如果你没有使用use strict的话，Perl会允许你随意使用未经声明的变量。但你应当总是在Perl中使用use strict。Python、Java或C中的等价代码将总是会要求声明变量。

<sup>③</sup> 你可以使用局部宏LOOP-FINISH让整个循环从循环体中的某段Lisp代码中直接正常返回，同时有机会执行循环的尾声部分。

你可以使用循环关键字named为LOOP命名。如果一个named子句出现在一个循环中，那么它必须是第一个子句。举一个简单的例子，假设lists是一个列表的列表，而你想要在这些嵌套的列表中查到匹配某些特征的项。你可以像下面这样使用一对嵌套的循环来找到它：

```
(loop named outer for list in lists do
  (loop for item in list do
    (if (what-i-am-looking-for-p item)
        (return-from outer item))))
```

## 22.12 终止测试

尽管for和repeat子句提供了控制循环次数的方法，但有时你需要更早地中断循环。你已经知道do子句里的return子句、RETURN或RETURN-FROM形式可以立即终止循环。但正如存在一些用来汇聚值的通用模式那样，也存在用来决定何时终止循环的通用模式。在LOOP中这些模式是由终止子句while、until、always、never和thereis来提供的。它们全都遵循相同的模式：

*loop-keyword test-form*

五种子句都会在每次通过迭代时对*test-form*求值，然后基于得到的值来决定是否终止循环。它们的区别在于，如果终止循环的话需要什么条件以及如何决定。

循环关键字while和until代表了“温和的”终止子句。当它们决定终止循环时，控制会传递到尾声部分，并跳过循环体的其余部分。尾声部分随后会返回一个值或是做任何想做的事情来结束循环。while子句在测试形式首次为假时终止循环；相反地，until子句在测试形式首次为真时停止循环。

另一个温和的终止形式是由LOOP-FINISH宏所提供的。这是一个正规的Lisp形式，并非一个循环子句，因此它可以用在一个do子句的Lisp形式中的任何地方。它也会导致立即跳转到循环的尾声部分。这在是否跳出循环的判断难以用一个简单的while或until子句来表达时是有用的。

另外三个子句即always、never和thereis，采用极端偏执的方式来终止循环。它们立即从循环中返回，不但跳过任何连续的循环子句而且还跳过尾声部分。它们还为整个循环提供了默认的返回值，哪怕是它们没有导致循环终止。尽管如此，如果循环不是因为这些终止测试中的一个而终止的话，那么尾声部分还是有机会运行，并返回一个值以代替终止子句所提供的默认值的。

由于这些子句提供了它们自己的返回值，因此它们不能跟汇聚类子句配合使用，除非汇聚子句带有一个into下级子句。编译器（或解释器）应当在编译期报告此类错误。always和never子句仅返回布尔值，因此在你需要用一个循环表达式来构成谓词时，它们将是最有用的。你可以使用always来确认循环的每次迭代过程中测试形式均为真。相反地，never测试每次迭代中测试形式均为假。如果测试形式失败了（在always子句中返回NIL或是在never子句中返回非NIL），那么循环将被立即终止，并返回NIL。如果循环可以一直运行直到完成，那么就会提供默认值T。

举个例子，如果你想要测试一个列表numbers中的所有数都是偶数，可以写成这样：

```
(if (loop for n in numbers always (evenp n))
      (print "All numbers even."))

```

下面是等价的另一种写法：

```
(if (loop for n in numbers never (oddp n))
      (print "All numbers even."))

```

`thereis`子句被用来测试是否测试形式“曾经”为真。一旦测试形式返回了一个非`NIL`的值，那么循环就会终止并返回该值。如果循环得以运行到完成，那么`thereis`子句会提供默认值`NIL`。

```
(loop for char across "abc123" thereis (digit-char-p char)) → 1
```

```
(loop for char across "abcdef" thereis (digit-char-p char)) → NIL
```

## 22.13 小结

现在你已经看到了`LOOP`功能的所有主要特性。只要你遵循下列规则就可以将我所讨论过的任何子句组合在一起：

- 如果有`named`子句的话，它必须是第一个子句。
- 在`named`子句后面是所有的`initially`、`with`、`for`和`repeat`子句。
- 然后是主体子句：有条件和无条件的执行、汇聚和终止测试。<sup>①</sup>
- 以任何`finally`子句结束。

`LOOP`宏将展开成完成下列操作的代码：

- 初始化所有由`with`或`for`子句声明的局部变量，以及由汇聚子句创建的隐含局部变量。提供初始值的形式按照它们在循环中出现的顺序进行求值。
  - 执行由任何`initially`子句（序言部分）所提供的形式，以它们出现在循环中的顺序来执行。
  - 迭代，同时按照下面一段文字所描述的过程来执行循环体的代码。
  - 执行由任何`finally`子句（尾声部分）所提供的形式，以它们出现在循环中的顺序来执行。
- 当循环在迭代时，循环体被执行的方式是首先步进那些迭代控制变量，然后以出现在循环中的顺序执行任何有条件或无条件的执行、汇聚或终止测试子句。如果循环中的任何子句终止了循环，那么循环体的其余部分将被跳过，然后整个循环可能在运行了尾声部分以后返回。

这基本上就是所有的内容了。<sup>②</sup>本书后面的代码中将频繁用到`LOOP`，因此有必要对它多些了解。除此之外，用不用它就完全取决于你了。

有了这些基础，就可以进入本书其余部分的实践性章节了。首先是编写一个垃圾过滤器。

22

① 一些Common Lisp实现允许你交替使用主体子句和`for`子句，但这在严格来讲是未定义的，并且另一些实现会拒绝这样的循环。

② 关于`LOOP`，我尚未讨论过的一个方面是用来声明循环变量类型的语法。当然，我也还没有讨论过`LOOP`之外的类型声明。我将在第32章里谈及这个一般主题。对于它们与`LOOP`配合使用的细节，请参考你所喜爱的Common Lisp手册。

# 实践：垃圾邮件过滤器



**2**002年，Paul Graham在把Viaweb卖给Yahoo之后腾出一些时间写了《一个处理垃圾邮件的计划》的随笔<sup>①</sup>，这引起了垃圾邮件过滤技术的一场小革命。在Graham的文章发表之前，多数垃圾邮件过滤器都基于手工编写的规则：如果标题中带有×××，那么它可能是垃圾邮件；如果正文的一行中有三个或更多词是全部大写的，那么它可能是垃圾邮件。Graham花了几个月时间试图编写这样一个基于规则的过滤器，并最终意识到这是一个非常折磨人的任务。

为了找出垃圾邮件的每一个特点，你必须站在垃圾邮件发送者的角度来思考。而且坦白地说，我想在这件事上花尽可能少的时间。

为了避免站在垃圾邮件发送者的角度来思考，Graham决定尝试从非垃圾邮件中区分垃圾邮件，通过统计哪些词出现在哪一类邮件中来做到这点。这个过滤器将持续跟踪特定单词出现在垃圾邮件和正常邮件中的频率，然后基于该频率分析新邮件中的单词，从而计算出该邮件是垃圾邮件还是正常邮件。他把这个方法称为贝叶斯（Bayesian）过滤，这种统计技术可以将个别词汇的频率组合成一个整体上的可能性。<sup>②</sup>

## 23.1 垃圾邮件过滤器的核心

在本章里，你将实现一个垃圾邮件过滤引擎的核心。我们不会编写完整的垃圾邮件过滤程序，而是会把精力集中在对新邮件进行分类以及训练过滤器上。

这个应用比较大，因此有必要定义一个包来避免名字冲突。在从本书网站下载到的源代码中，我使用了包名COM.GIGAMONKEYS.SPAM，这个包同时用到标准COMMON-LISP包和来自第15章的COM.GIGAMONKEYS.PATHNAMES包：

```
(defpackage :com.gigamonkeys.spam
  (:use :common-lisp :com.gigamonkeys.pathnames))
```

任何含有这个应用程序代码的文件都应当以下面这行开始：

<sup>①</sup> 此文可从<http://www.paulgraham.com/spam.html>找到，也收录在*Hackers & Painters: Big Ideas from the Computer Age* (O'Reilly, 2004) 一书中。

<sup>②</sup> 关于Graham所描述的技术是否真的是“贝叶斯”一直以来有些不同的看法。不过这个名字已经广为流传并成为谈论垃圾邮件过滤时“统计”的代名词。

```
(in-package :com.gigamonkeys.spam)
```

你可以使用相同的包名或者将com.gigamonkeys替换成你所控制的某个域。<sup>①</sup>

还可以通过在REPL中输入相同的形式来切换到这个包，从而测试你所编写的函数。在SLIME中这将使提示符从CL-USER>变成SPAM>，像下面这样：

```
CL-USER> (in-package :com.gigamonkeys.spam)
#<The COM.GIGAMONKEYS.SPAM package>
SPAM>
```

定义了包以后，就可以开始实际的编码工作了。你要实现的主函数有一个简单的任务——接受一封邮件的文本作为参数并将该邮件分类成垃圾邮件、有用信息或不确定。实现这个基本函数很简单，可以通过后面要编写的其他函数来定义它。

```
(defun classify (text)
  (classification (score (extract-features text))))
```

从里向外，分类邮件的第一步是从文本中提取出特征词并传递给score函数。score将计算出一个值，该值随后通过函数classification分成三类——垃圾邮件、有用信息或不确定中的一类。在这三个函数中，classification是最简单的。可以假设score在邮件为垃圾邮件时返回一个接近1的值，对正常邮件返回接近0的值，而在不确定时返回接近0.5的值。

因此你可以像下面这样实现classification：

```
(defparameter *max-ham-score* .4)
(defparameter *min-spam-score* .6)

(defun classification (score)
  (cond
    ((<= score *max-ham-score*) 'ham)
    ((>= score *min-spam-score*) 'spam)
    (t 'unsure)))
```

23

函数extract-features也几乎是一样直接，尽管它需要更多一些的代码。目前你所提取的特征是出现在文本中的单词。对于每个单词，需要跟踪它在垃圾邮件中出现的次数和在正常邮件中出现的次数。为方便地将这些数据与单词本身保存在一起，可以定义一个类word-feature，该类带有三个槽。

```
(defclass word-feature ()
  ((word
    :initarg :word
    :accessor word
    :initform (error "Must supply :word")
    :documentation "The word this feature represents.")
   (spam-count
    :initarg :spam-count
    :accessor spam-count
    :initform 0
    :documentation "Number of spams we have seen this feature in.")
   (ham-count
    :initarg :ham-count
```

<sup>①</sup> 尽管如此，并不推荐你使用一个以com.gigamonkeys开头的包来分发该应用，因为你并不控制那个域。

```
:accessor ham-count
:initform 0
:documentation "Number of hams we have seen this feature in.")))
```

把所有的特征数据库保存在一个哈希表中，可以方便地查找代表给定特征的对象。可以定义一个特殊变量`*feature-database*`来保存对这个哈希表的引用。

```
(defvar *feature-database* (make-hash-table :test #'equal))
```

应当使用`DEFVAR`而不是`DEFPARAMETER`来定义它，因为你不希望你在开发过程中重新加载了含有该定义的文件后被`*feature-database*`重置，你不想丢失那些保存在`*feature-database*`中的数据。当然，这意味着如果你确实想要清空这个特征数据库，那你不能只是重新求值`DEFVAR`形式。所以应该定义一个`clear-database`函数。

```
(defun clear-database ()
  (setf *feature-database* (make-hash-table :test #'equal)))
```

为了查找给定邮件中的特征，代码要提取出每一个词，继而在`*feature-database*`中查找对应的`word-feature`对象。如果`*feature-database*`不含有这个特征，那么它将创建一个代表该词的新`word-feature`。可以将这些逻辑封装在一个函数`intern-feature`中，它接受一个单词并返回对应的特征，必要时会创建它。

```
(defun intern-feature (word)
  (or (gethash word *feature-database*)
      (setf (gethash word *feature-database*)
            (make-instance 'word-feature :word word))))
```

可以使用正则表达式从邮件文本中提取出某个词。例如，使用Edi Weitz编写的Common Lisp可移植的Perl兼容的正则表达式（CL-PPCRE）库，`extract-words`的代码可以这样写：<sup>①</sup>

```
(defun extract-words (text)
  (delete-duplicates
   (cl-ppcre:all-matches-as-strings "[a-zA-Z]{3,}" text)
   :test #'string=))
```

现在为了实现`extract-features`需要将`extract-words`和`intern-feature`放在一起。由于`extract-words`返回了一个字符串的列表，而你想要的是一个列表中的每个字符串都被转换成对应的`word-feature`对象，因此正好可以使用`MAPCAR`。

```
(defun extract-features (text)
  (mapcar #'intern-feature (extract-words text)))
```

可以在REPL中测试这些函数：

```
SPAM> (extract-words "foo bar baz")
("foo" "bar" "baz")
```

同时确保`DELETE-DUPLICATES`能够工作：

```
SPAM> (extract-words "foo bar baz foo bar")
("baz" "foo" "bar")
```

还可以测试`extract-features`。

---

<sup>①</sup> CL-PPCRE的一个版本包含于本书的源代码中，或者你可以从Weitz的站点<http://www.weitz.de/cl-ppcre/>下载。

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE @ #x71ef28da> #<WORD-FEATURE @ #x71e3809a>
 #<WORD-FEATURE @ #x71ef28aa>)
```

正如你所看到的，打印任意对象的默认方法输出的信息太简单。对于这个程序来说，最好可以更清晰地打印出word-feature对象。第17章提到过，所有对象的打印都是由广义函数PRINT-OBJECT来实现的。因此，为了改变word-feature的打印方式，你只需定义一个特化在word-feature上的PRINT-OBJECT方法。为了让这样的方法实现起来更简单，Common Lisp提供了一个宏PRINT-UNREADABLE-OBJECT。<sup>①</sup>

PRINT-UNREADABLE-OBJECT基本形式如下所示：

```
(print-unreadable-object (object stream-variable &key type identity)
  body-form*)
```

其中的object参数是一个求值到被打印对象的表达式。在PRINT-UNREADABLE-OBJECT主体中，stream-variable被绑定到一个流，可以向其中打印你想要的任何东西。打印到该流中的任何东西都将被PRINT-UNREADABLE-OBJECT输出并封装在不可读对象的标准语法#<>中。<sup>②</sup>

通过关键字参数type和identity，PRINT-UNREADABLE-OBJECT还可以让你包含对象的类型和一个对象标识的指示。如果它们是非NIL的，那么输出将以对象类的名字开始并以对象的标识结束，这与STANDARD-OBJECT的默认PRINT-OBJECT方法所打印的形式相似。对于word-feature来说，需要定义一个PRINT-OBJECT方法来包含其类型而不是标识，并同时带有word、ham-count和spam-count等槽的值。这个方法如下所示：

```
(defmethod print-object ((object word-feature) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (word ham-count spam-count) object
      (format stream "~s :hams ~d :spams ~d" word ham-count spam-count))))
```

再在REPL中测试extract-features时，可以更清楚地看到那些被提取出的特征。

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE "baz" :hams 0 :spams 0>
 #<WORD-FEATURE "foo" :hams 0 :spams 0>
 #<WORD-FEATURE "bar" :hams 0 :spams 0>)
```

## 23.2 训练过滤器

现在有了跟踪单独特征的方式，几乎可以开始实现score了。但首先你需要编写用来训练垃圾邮件过滤器的代码，这样score才会有数据可用。为此要定义一个函数train，它接受一些文本和一个指示邮件类型（ham或spam）的符号，然后递增文本中出现的所有特征的ham或spam

<sup>①</sup> 使用PRINT-UNREADABLE-OBJECT的主要原因是，在某人试图可读地打印你的对象时，它会负责报一个适当的错误，例如在使用FORMAT指令~S时。

<sup>②</sup> PRINT-UNREADABLE-OBJECT也会在打印控制变量\*PRINT-READABLY\*为真时报错。这样，一个完全由PRINT-UNREADABLE-OBJECT形式组成的PRINT-OBJECT方法将正确实现遵守\*PRINT-READABLY\*协议的PRINT-OBJECT。

计数器，它们代表目前所处理过的有用信息或垃圾邮件信息的全局计数。你可以再次采用自顶向下的方法通过其他尚不存在的函数来实现它。

```
(defun train (text type)
  (dolist (feature (extract-features text))
    (increment-count feature type))
  (increment-total-count type))
```

你已经编写了extract-features，因此下一个需要编写的是increment-count，它接受一个word-feature和一个邮件类型并递增该特征的相应槽。由于递增这些计数器的逻辑对于不同类型的对象不应变化，因此可以将它写成一个正规函数。<sup>①</sup>因为你将ham-count和spam-count都定义成带有一个:accessor选项，因此可以使用INCF和由DEFCLASS创建的访问函数来递增相应的槽。

```
(defun increment-count (feature type)
  (ecase type
    (ham (incf (ham-count feature)))
    (spam (incf (spam-count feature)))))
```

其中的**ECASE**构造是**CASE**的一个变体，两者都类似于源自Algol语言的case语句（在C和它的后裔中重命名成switch了）。它们都求值其第一个参数——键形式，然后找出第一个元素（键）EQL相等的子句。在本例中，这意味着当变量type被求值时，得到了作为increment-count的第二个参数所传递的值。

键不会被求值。换句话说，type的值将与Lisp读取器作为**ECASE**形式一部分所读取的字面对象进行比较。在这个函数中，这意味着键是符号ham和spam，而不是任何名为ham和spam的变量的值。因此，如果increment-count像这样

```
(increment-count some-feature 'ham)
```

被调用，那么type的值将是符号ham，而**ECASE**的第一个分支将被求值并且对应特性的ham计数将会递增。另一方面，如果它像这样

```
(increment-count some-feature 'spam)
```

被调用，那么第二个分支将运行，从而递增spam计数。这里符号ham和spam在调用increment-count时被加了引号，否则它们就会作为变量被求值。但是当它们出现在**ECASE**中时却没加引号，因为**ECASE**不对键求值。<sup>②</sup>

<sup>①</sup> 如果你以后决定需要为不同的类编写不同版本的increment-feature，那么你可以将increment-count重定义成一个广义函数，而将该函数定义成一个特化在word-feature上的方法。

<sup>②</sup> 从技术上来讲，一个**CASE**或**ECASE**的每个子句中的键都将被解释成一个列表指示符，一个指定了对象列表的对象。一个单一的非列表对象被当作列表指示符对待时相当于一个只含有一个对象的列表，而一个列表将指代它本身。这样，每个子句可以有多个键，**CASE**和**ECASE**将选择键列表中含有键形式的值的子句。例如，如果你想要把good作为ham的同义词而把bad作为spam的同义词，那么你可以像下面这样来编写increment-count：

```
(defun increment-count (feature type)
  (ecase type
    ((ham good) (incf (ham-count feature)))
    ((spam bad) (incf (spam-count feature)))))
```

**ECASE**中的**E**代表“无遗漏的”(exhaustive)或“错误”(error)，意味着当键值是任何列出的键之外的东西时，**ECASE**应当报错。正常的**CASE**相对宽松，当没有匹配的子句时返回**NIL**。

为了实现increment-total-count，需要决定将计数保存在哪里。目前，使用两个特殊变量\*total-spams\*和\*total-hams\*会比较好。

```
(defvar *total-spams* 0)
(defvar *total-hams* 0)

(defun increment-total-count (type)
  (ecase type
    (ham (incf *total-hams*))
    (spam (incf *total-spams*))))
```

应当使用**DEFVAR**来定义这两个变量，理由与用在\*feature-database\*时相同——它们将在你运行程序期间始终保持其中的数据，你不想只是因为在开发过程中重新加载了你的代码就扔掉这些数据。但是你希望在重置\*feature-database\*之后可以顺便重置这两个变量。因此，应当按照如下方式在clear-database中添加几行：

```
(defun clear-database ()
  (setf
    *feature-database* (make-hash-table :test #'equal)
    *total-spams* 0
    *total-hams* 0))
```

23

## 23.3 按单词来统计

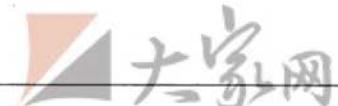
一个统计型垃圾邮件过滤器的核心当然是那些基于统计结果计算概率的函数。讨论这些计算的数学原理<sup>①</sup>超出了本书的范围，有兴趣的读者可以参考Gray Robinson的几篇论文。<sup>②</sup>我将把焦点集中在它们是怎样实现的。

统计计算的起点是测量值的集合——保存在\*feature-database\*、\*total-spams\*和\*total-hams\*中的频率数据。假设用于训练的邮件集合在统计上有代表性，那么可以将观察到的频率视为未来邮件中同样特征在有用信息和垃圾邮件信息中出现的概率。

分类邮件的基本方法是提取其中的特征，计算含有该特征的邮件是垃圾邮件的概率，然后再

① 从数学的角度来说，本章中有时对概率一词较宽松的用法可能会冒犯严肃的统计学家。不过，由于即便是该用法的赞成者，其中还进一步划分成贝叶斯论者和频率论者，也无法对概率究竟是什么达成统一意见，因此我不会担心这一点。毕竟这是一本关于编程而不是统计学的书。

② 本章参考的Robinson的文章有“A Statistical Approach to the Spam Problem”（发表在Linux Journal上并可从<http://www.linuxjournal.com/article.php?sid=6467>获得，一个简化版本发表在Robinson的博客<http://radio.weblogs.com/0101454/stories/2002/09/16/spamDetection.html>上）以及“Why Chi? Motivations for the Use of Fisher's Inverse Chi-Square Procedure in Spam Classification”（可从<http://garyrob.blogs.com/whychi93.pdf>获得）。另一篇可能有用的文章是“Handling Redundancy in Email Token Probabilities”（可从<http://garyrob.blogs.com/handlingtokenredundancy94.pdf>获得）。SpamBayes项目（<http://spambayes.sourceforge.net/>）的存档邮件列表里也含有许多关于测试垃圾过滤器的不同算法和思想的有用信息。



将所有这些概率综合成该邮件的一个整体评分，带有许多垃圾邮件特征和很少有用特征的邮件将得到接近于1的评分，而带有许多有用特征和很少垃圾邮件特征的邮件的评分会接近于0。

第一个统计函数用来计算一个含有给定特征的邮件是垃圾邮件的概率。从某种角度看，一个含有该特征的给定邮件是垃圾邮件的概率，就是含有该特征的垃圾邮件与含有该特征的所有邮件的比值。这样，就可以用如下方式来计算它：

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (/ spam-count (+ spam-count ham-count))))
```

该函数的值可能被任何邮件是一封垃圾邮件或有用邮件的总体概率所影响。例如，假设你通常获得的正常邮件是垃圾邮件的9倍。那么一个完全中立的特征将在每9个正常邮件后出现在一个垃圾邮件里，从而这个函数计算出1/10的垃圾邮件概率。

但你更感兴趣的是一个给定特征出现在一封垃圾邮件中的概率，与收到垃圾邮件或正常邮件的整体概率无关。这样，你需要将垃圾邮件数量除以接受训练的垃圾邮件总数，将有用邮件数量除以有用邮件总数。为了避免发生除零错误，如果`*total-spams*`或`*total-hams*`两者任何一个为零，那么你应当将相应的频率视为零。（很明显，如果垃圾邮件或有用消息的任一总数为零，那么相应的每特征计数也将是零，因此你可以将结果频率视为零而不会带来不良影响。）

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (let ((spam-frequency (/ spam-count (max 1 *total-spams*)))
          (ham-frequency (/ ham-count (max 1 *total-hams*))))
        (/ spam-frequency (+ spam-frequency ham-frequency)))))
```

这个版本还有另一个问题——它没有在每单词概率上计入到达并分析的消息数量。假设你已经训练了2000条消息，一半是垃圾邮件而另一半正常。现在考察两个只出现在垃圾邮件中的特征。一个出现在所有1000条邮件中，而另一个仅出现一次。根据当前`spam-probability`的定义，两个特征的出现预测一个邮件是垃圾邮件的概率是相等的，都是1。

尽管如此，那个仅出现一次的特征很可能实际上是一个中性的特征，它很明显在无论是垃圾邮件还是有用信息中都很罕见，在2000条消息中仅出现一次。如果你训练了另外2000条邮件，它很可能又出现了一次，这次出现在一条正常邮件中，使得它突然成为了垃圾邮件可能性为0.5的一条中性特征。

所以看起来你想要计算一个概率，它以某种方式影响了进入到每个特征中的数据点数。Robinson在他的论文中推荐了一个基于贝叶斯概念的函数，将观察到的数据与先验知识或假设相合并。基本上，你以一个假设的先验概率开始计算新的概率并在添加新信息之前给假设的概率一个权重。Robinson的函数如下所示：

```
(defun bayesian-spam-probability (feature &optional
                                      (assumed-probability 1/2)
                                      (weight 1))
  (let ((basic-probability (spam-probability feature))
        (data-points (+ (spam-count feature) (ham-count feature))))
    (/ (+ (* weight assumed-probability)
```

```
(* data-points basic-probability))
(+ weight data-points))))
```

Robinson建议把1/2作为assumed-probability的值，把1作为weight的值。使用这些值，一个出现在一条垃圾邮件中而没有出现在有用邮件中的特征具有0.75的bayesian-spam-probability，一个出现在10条垃圾邮件而没有出现在有用邮件中的特征具有大约0.955的bayesian-spam-probability，而一条匹配了1000条垃圾邮件却没有有用邮件的特征将具有大约0.9995的垃圾邮件概率。

## 23.4 合并概率

现在你可以计算在一条消息中所找到的每一个单独特征的bayesian-spam-probability，实现score函数的最后一步是找出一种方式，将大量的概率个体合并成介于0和1之间的单个值。

如果单独的特征概率是彼此无关的，那么从数学上来讲可以将它们相乘从而得到一个合并的概率。但是它们实际上不可能是彼此无关的，特定的特征很可能会一起出现，而其他一些却从不这样。<sup>①</sup>

Robinson提议使用由统计学家R. A. Fisher发明的概率组合方法。在不讨论为什么它的技术可以奏效的具体细节的前提下，方法是这样的：首先你通过将所有概率相乘来把它们组合在一起。这给了你一个接近于0的远低于最初概率集合中概率的值，然后取该值的对数并乘以-2。Fisher在1950年证明，如果这些单独的概率是彼此无关的并且来自于0和1之间的统一分布，那么得到的值将满足卡方(chi-square,  $\chi^2$ )分布。该值和概率数量的两倍可以输入到一个反向卡方分布函数中，然后返回一个反映了通过组合相同数量的随机选择概率得到越来越大的值的可能性。当这个反向卡方分布函数返回一个较低的概率时，这意味着在单独的概率中存在相当多的低概率。(要么是许多相对的低概率，要么是少量非常低的概率。)

23

为了使用这个概率来检测一个给定的邮件是否是垃圾邮件，你从一个空假设(null hypothesis)开始，一个你想要击倒的稻草人。这个空假设是被分类的消息事实上只是一个特性的随机集合。如果它是的话，那么单独的概率，即每个特征出现在一个垃圾邮件中的可能性也将是随机的。这就是说，一个特征的随机选择将通常含有一些经常出现在垃圾邮件中的特征和另一些很少出现在垃圾邮件中的特征。如果你根据Fisher的方法合并这些随机选择的概率，那么你将得到一个中间的合并值，然后反向卡方分布函数将很可能返回一个比较高的值，事实也正是如此。但如果反向卡方分布返回了一个非常低的概率，这意味着产生该合并值的那些概率不太可能是随机选择的，而可能是里面有太多的低概率值。因此你可以拒绝这个空假设而采纳另一个替代假设：所有引入的特征来自一个有偏的样本——一个带有少量高垃圾邮件概率特征和许多低垃圾邮件特征的样本。换句话说，它一定是条有用的邮件。

尽管如此，Fisher方法并不是对称的，因为反向卡方分布函数对于由给定数量的随机选择的

<sup>①</sup> 从技术上来讲，对一些事实上无关的概率进行非无关的概率合并，这称为原生贝叶斯(Naive Bayesian)。Graham最初发表的建议本质上是一个原生贝叶斯分类器，其中带有一些“经验驱动”的常量因子。

概率返回的合并后的概率，将比你从合并实际概率中得到的值大得多。这种非对称性的用法对你是有利的，因为当你拒绝空假设时你知道更好的假设是什么。当你用Fisher方法合并单独的垃圾邮件概率时，而它告诉你有很高的概率表明空假设是错误的——邮件并不是一个单词的随机集合，那么这意味着该邮件很可能是有用的。返回的数值就算并非该邮件是有用邮件的字面概率，至少也是对它有用程度的一个好的衡量。相反，对于单独的有用概率的Fisher合并可以给你关于该邮件垃圾邮件程度的一个衡量。

为了得到一个最终的评分，你需要将这两个指标合并成单一的值，从而给一个范围是从0到1的组合的有用程度-垃圾邮件程度评分。Robinson所推荐的方法是将有用程度和垃圾邮件程度之间差异的一半与1/2相加，换句话说，就是垃圾邮件程度和1减去有用程度的平均值。这在两个评分相反（高的垃圾邮件程度和低的有用程度，或者反过来）时可以带来很好的效果，这时你将得到一个接近0或1的强烈的指示值。但是当垃圾邮件程度和有用程度的评分都高或都低时，你将得到一个接近1/2的最终值，从而得到一个“不确定”的分类。

实现这一模型的score函数如下所示：

```
(defun score (features)
  (let ((spam-probs ()) (ham-probs ()) (number-of-probs 0))
    (dolist (feature features)
      (unless (untrained-p feature)
        (let ((spam-prob (float (bayesian-spam-probability feature) 0.0d0)))
          (push spam-prob spam-probs)
          (push (- 1.0d0 spam-prob) ham-probs)
          (incf number-of-probs))))
    (let ((h (- 1 (fisher spam-probs number-of-probs)))
          (s (- 1 (fisher ham-probs number-of-probs))))
      (/ (+ (- 1 h) s) 2.0d0))))
```

你接受一个特征的列表并循环它们，构建起两个概率的列表，一个列出含有每个特征的消息是垃圾邮件的概率，而另一个列出含有每个特征的邮件是有用邮件的概率。作为一项优化，你也可以在循环过程中统计概率的数量，然后将这个计数传给fisher从而避免在fisher本身再次对它们计数。当单独的概率中含有许多来自随机文本的低概率值时，由fisher返回的值也将非常低。这样，一个低的fisher垃圾邮件概率评分意味着存在许多有用的特征。将这个评分减去1就得到该邮件是有用邮件的概率。相反，从有用概率中减去fisher评分将得到该邮件是垃圾邮件的概率。将这两个概率组合在一起就可以给你一个介于0和1之间的整体垃圾邮件程度评分。

在循环内部，你可以使用函数untrained-p来跳过那些从邮件中提取出的从未在训练中出现过的特征。这些特征将具有值为0的垃圾邮件计数和有用计数。untrained-p函数非常简单。

```
(defun untrained-p (feature)
  (with-slots (spam-count ham-count) feature
    (and (zerop spam-count) (zerop ham-count))))
```

剩下的唯一一个新的函数是fisher本身。假设你已经有了一个inverse-chi-square函数，那么fisher在概念上很简单。

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
```

```
(inverse-chi-square
  (* -2 (log (reduce #'* probs)))
  (* 2 (number-of-probs))))
```

不幸的是，在这个相当直接的实现中有一个小问题。尽管使用REDUCE是一个将数字列表相乘的简洁方法，但在这个特定应用中乘积将会过小而无法表示成一个浮点数。在这种情况下，结果将会下溢到0。而且，因为概率的乘积下溢，所有努力都将白费，因为对0求LOG要么报错，要么在某些实现中得到一个特殊的负无穷大值，这将使得所有后续的计算在本质上都变成无意义的。这在本函数中尤其不幸，因为当输入的概率值较低（接近0）时，fisher方法最为敏感，并且因此非常容易导致乘积下溢。

幸运的是，你可以运用一点儿高中数学知识来避免这个问题。一个乘法的对数等价于所有因数的对数之和，因此不用将所有概率相乘然后取对数，你可以将每个概率的对数相加。并且，由于REDUCE接受一个:key关键字参数，你可以用它来完成整个计算。把下面的写法：

```
(log (reduce #'* probs))
```

写成这样：

```
(reduce #'+ probs :key #'log)
```

## 23.5 反向卡方分布函数

23

本节中的inverse-chi-square实现是Rebinson所写的一个Python版本的相当直接的转换。讲解该函数的确切数学含义超出了本书的范围，但通过思考你传递给fisher的值将怎样影响结果可以得到一个关于其作用的直观印象：你传给fisher的低概率值越多，概率的乘积将会越小。一个小的乘积的对数将会是一个绝对值较大的负数。这样，传递给fisher的低概率值越多，它传递给inverse-chi-square的值就越大。当然，引入的概率数量也会影响传递给inverse-chi-square的值。由于概率的定义是小于或等于1的，一个乘积中的概率越多，它的结果就会越小并且传给inverse-chi-square的值就会越大。这样，在fisher合并值相比进入它的概率数量异乎寻常地大时，inverse-chi-square将返回一个较低的概率。下面的函数精确地做到了这点：

```
(defun inverse-chi-square (value degrees-of-freedom)
  (assert (evenp degrees-of-freedom))
  (min
    (loop with m = (/ value 2)
          for i below (/ degrees-of-freedom 2)
          for prob = (exp (- m)) then (* prob (/ m i))
          summing prob)
    1.0))
```

回忆第10章里EXP计算e的给定参数次方。这样，value的值越大，prob的初始值将会越小。但只要m大于自由度的数量，这个初始值随后就将不断地被每个自由度微调。由于inverse-chi-square返回的值应当是另一个概率，有必要用MIN来固定返回值，因为乘法和指数计算中的边界错误可能导致LOOP返回一个稍大于1的和。

## 23.6 训练过滤器

你编写classify和train来接受一个字符串参数，因此你可以轻松地在REPL中测试它们。如果你还没有这样做过，那么你应当通过在REPL中求值一个IN-PACKAGE形式，或是使用SLIME的快捷命令change-package将当前包切换到你编写这些代码所在的包中。在你输入包名的时候，按Tab将会根据你的Lisp所知道的包来自动补全包名。现在你可以调用任何属于垃圾邮件应用一部分的函数了。你应当首先确保数据库为空。

```
SPAM> (clear-database)
```

现在你可以用一些文本来训练过滤器。

```
SPAM> (train "Make money fast" 'spam)
```

然后看分类器是怎样判断的。

```
SPAM> (classify "Make money fast")
SPAM
SPAM> (classify "Want to go to the movies?")
UNSURE
```

尽管最终你所关心的只是那个分类，但可以看到原始的评分也是很有用的。得到两个值而不会干扰任何其他代码的最简单方法是改变classification从而返回多个值。

```
(defun classification (score)
  (values
    (cond
      ((<= score *max-ham-score*) 'ham)
      ((>= score *min-spam-score*) 'spam)
      (t 'unsure))
    score))
```

你可以做出这个改变，然后只重新编译这一个函数。classify返回classification所返回的任何东西，因此它也将返回两个值。但由于主返回值和以前相同，这两个函数的那些只需要一个值的调用者将不会受到影响。现在当你测试classify时，能够精确地看到进入到分类中的评分。

```
SPAM> (classify "Make money fast")
SPAM
0.863677101854273D0
SPAM> (classify "Want to go to the movies?")
UNSURE
0.5D0
```

现在你可以看到，如果你用更多的一些有用邮件来训练过滤器的话将发生什么。

```
SPAM> (train "Do you have any money for the movies?" 'ham)
1
SPAM> (classify "Make money fast")
SPAM
0.7685351219857626D0
```

它仍然是垃圾邮件，只是不太确定，因为“money”在有用邮件中也出现了。

```
SPAM> (classify "Want to go to the movies?")
HAM
0.17482223132078922D0
```

而现在它被清楚地识别成有用的邮件，这要感谢单词“movies”的存在，这是一个有用的特征。

不过，你可能并不真的想手工训练这个过滤器，你真正喜欢的是一种简单的方式来指向一堆文件并在其上训练它。除此之外，因为你想要测试该过滤器实际工作的效果，你会希望随后使用它来分类另外一些已知类型的文件并查看分类的效果。因此，在本章中你将要编写的最后一点代码是一套测试系统，它在一个已知类型的邮件库上测试该过滤器，使用其中的一定比例用于训练，然后在其余部分测量该过滤器在分类时的精度。

## 23.7 测试过滤器

为了测试该过滤器，你需要一个已知类型的邮件库。你可以使用邮箱中的邮件，或者从Web上获得一个可用的邮件库。例如，SpamAssassin邮件库<sup>①</sup>含有手工分类成垃圾邮件、稍微有用和十分有用的几千条邮件。为了更容易地使用你所拥有的文件，可以定义一个驱动在一个文件/类型对的数组上的测试平台。你可以定义一个函数来接受文件名和类型，并像下面这样将其添加到消息库中：

```
(defun add-file-to-corpus (filename type corpus)
  (vector-push-extend (list filename type) corpus))
```

其中corpus的值应当是一个带有填充指针的可调整向量。例如，你可以像这样创建一个新的库：

```
(defparameter *corpus* (make-array 1000 :adjustable t :fill-pointer 0))
```

如果你的有用邮件和垃圾邮件已经分别放在了不同的目录中，那么你可能想要一次性将一个目录中的所有文件作为相同的类型添加到库中。下面这个函数使用了第15章的list-directory函数，它实现了上述想法：

```
(defun add-directory-to-corpus (dir type corpus)
  (dolist (filename (list-directory dir))
    (add-file-to-corpus filename type corpus)))
```

例如，假设你有一个mail目录，它含有两个子目录spam和ham，分别包含相应类型的消息。你可以像下面这样把这两个目录中的所有文件添加到\*corpus\*中：

```
SPAM> (add-directory-to-corpus "mail/spam/" 'spam *corpus*)
NIL
SPAM> (add-directory-to-corpus "mail/ham/" 'ham *corpus*)
NIL
```

23

<sup>①</sup> 包括SpamAssassin邮件库在内的几个垃圾邮件库可以从<http://nexp.cs.pdx.edu/~psam/cgi-bin/view/PSAM/CorpusSets>获得。

现在你需要一个函数来测试分类器。基本的策略是选择库中的一个随机片段来训练，然后通过把库的其余部分分类来测试这个库，将由classify返回的分类与已知的分类进行比较。你主要想知道的是分类器的精度——究竟有多少百分比的消息被正确分类了？但你还可能对错误分类的消息以及错误的方向感兴趣——究竟是假阳性更多还是假阴性更多？为了方便对分类器的行为不断地进行分析，你应当定义测试函数来构造一个原始结果的列表，随后你可以用任何方法来分析它。

主测试函数如下所示：

```
(defun test-classifier (corpus testing-fraction)
  (clear-database)
  (let* ((shuffled (shuffle-vector corpus))
         (size (length corpus))
         (train-on (floor (* size (- 1 testing-fraction)))))
    (train-from-corpus shuffled :start 0 :end train-on)
    (test-from-corpus shuffled :start train-on)))
```

这个函数从清空特征数据库开始。<sup>①</sup>然后，它对整个库进行“洗牌”，使用一个你将很快实现的函数，基于其testing-fraction参数来找出用于训练的消息和将被保留用来测试的消息。两个辅助函数train-from-corpus和test-from-corpus都将带有关键字参数:start和:end，从而允许它们对给定消息库的一个子序列进行操作。

train-from-corpus函数相当简单——简单地在库的适当部分中循环，它使用DESTRUCTURING-BIND从每个元素中解出文件名和类型，然后将命名文件的文本和类型传给train。由于某些邮件消息，尤其是那些带有附件的，通常会比较大，你应当限制它从消息中获取的字符数量。它使用一个你将很快实现的函数start-of-file来获取文本，该函数接受一个文件名和一个最大的字符数来返回相应的文本。train-from-corpus如下所示：

```
(defparameter *max-chars* (* 10 1024))

(defun train-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) do
        (destructuring-bind (file type) (aref corpus idx)
          (train (start-of-file file *max-chars*) type))))
```

除了要返回一个含有每个分类结果的列表，从而可以稍后来分析它们之外，函数test-from-corpus和上述函数相似。这样，你应当同时捕捉由classify返回的分类和评分数据，然后将文件名、实际类型、由classify返回的类型以及评分收集在一个列表中。为了使结果更好理解，你可以在列表中放置一些关键字来指示每个值的含义。

```
(defun test-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) collect
        (destructuring-bind (file type) (aref corpus idx)
          (multiple-value-bind (classification score)
```

<sup>①</sup> 如果你想要进行一个测试而不想干扰已有的数据库，那么你可以用一个LET绑定\*feature-database\*、\*total-spams\*和\*total-hams\*，但这样的话在测试结束之后，你就没有办法查看数据库了，除非你把所用到的这些值返回到函数中。

```
(classify (start-of-file file *max-chars*))  
(list  
  :file file  
  :type type  
  :classification classification  
  :score score))))
```

## 23.8 一组工具函数

为了完成test-classifier的实现，你还需要编写两个事实上跟垃圾邮件过滤没有特别关系的工具函数，shuffle-vector和start-of-file。

实现shuffle-vector的简单有效的方法是使用Fisher-Yates算法<sup>①</sup>。你可以从实现一个函数nshuffle-vector开始，它可以就地重排一个向量。这个名字遵循与诸如NCONC和NREVERSE等其他破坏性函数相同的命名规则。如下所示：

```
(defun nshuffle-vector (vector)  
  (loop for idx downfrom (1- (length vector)) to 1  
        for other = (random (1+ idx))  
        do (unless (= idx other)  
              (rotatef (aref vector idx) (aref vector other))))  
  vector)
```

23

非破坏性版本简单地复制最初的向量，然后将它传给破坏性版本。

```
(defun shuffle-vector (vector)  
  (nshuffle-vector (copy-seq vector)))
```

另一个工具函数start-of-file也是非常直接的，只有一点特别。把一个文件的内容读取到内存中最有效的方式是创建一个适当大小的数组并使用READ-SEQUENCE来填充其内容。因此，你应该创建一个字符数组，其长度要么是文件的大小要么是你想要读取的字符的最大数量，后者相对小一些。不幸的是，如同第14章里提到的，在处理字符流时，函数FILE-LENGTH完全没有很好地定义，因为一个文件中编码的字符个数可能同时取决于使用的字符编码和文件中的特定文本。在最坏的情况下，精确测量文件中字符数的唯一方法是实际读取整个文件。这样，在处理字符流时FILE-LENGTH存在歧义。而在多数实现中，FILE-LENGTH总是返回文件的字节数，这可能大于可从文件中读取的字符数。

不过，READ-SEQUENCE可以返回实际读取的字符数。因此，你可以尝试读取由FILE-LENGTH报告的字符数，并在实际读取的字符数较少时返回一个子串。

```
(defun start-of-file (file max-chars)  
  (with-open-file (in file)  
    (let* ((length (min (file-length in) max-chars))  
          (text (make-string length)))
```

<sup>①</sup> 这个算法以发明了概率合并方法的同一个Fisher和Frank Yates来命名，后者是Fisher的*Statistical Tables for Biological, Agricultural and Medical Research* (Oliver & Boyd, 1938) 一书的共同作者。根据Knuth的说法，他们首次公开发表了对该算法的描述。

```
(read (read-sequence text in)))
(if (< read length)
  (subseq text 0 read)
  text))))
```

## 23.9 分析结果

现在编写一些代码来分析由`test-classifier`生成的结果。回顾一下，`test-classifier`返回了由`test-from-corpus`所返回的列表，其中每个元素是一个代表了文件分类结果的plist。这个plist含有该文件的名字、文件的实际类型、分类以及由`classify`所返回的评分。编写分析性代码的第一步是编写一个函数，该函数可以返回一个符号来指明一个给定结果究竟是正确的、假阳性的、假阴性的、错过的有用消息或错过的垃圾邮件消息。你可以使用`DESTRUCTURING-BIND`从一个单独的结果列表中取出`:type`和`:classification`元素（使用`&allow-other-keys`来告诉`DESTRUCTURING-BIND`忽略任何其他的键值对），然后使用嵌套的`ECASE`将不同的配对转换成单一符号。

```
(defun result-type (result)
  (destructuring-bind (&key type classification &allow-other-keys) result
    (ecase type
      (ham
        (ecase classification
          (ham 'correct)
          (spam 'false-positive)
          (unsure 'missed-ham)))
      (spam
        (ecase classification
          (ham 'false-negative)
          (spam 'correct)
          (unsure 'missed-spam))))))
```

你可以在REPL中测试这个函数。

```
SPAM> (result-type '(:FILE #p"foo" :type ham :classification ham :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type spam :classification spam :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type ham :classification spam :score 0))
FALSE-POSITIVE
SPAM> (result-type '(:FILE #p"foo" :type spam :classification ham :score 0))
FALSE-NEGATIVE
SPAM> (result-type '(:FILE #p"foo" :type ham :classification unsure :score 0))
MISSED-HAM
SPAM> (result-type '(:FILE #p"foo" :type spam :classification unsure :score 0))
MISSED-SPAM
```

有了这个函数，你就可以方便地以多种方式切分`test-classifier`的结果了。例如，你可以从为每种结果类型定义谓词函数开始。

```
(defun false-positive-p (result)
  (eql (result-type result) 'false-positive))
```

```
(defun false-negative-p (result)
  (eql (result-type result) 'false-negative))

(defun missed-ham-p (result)
  (eql (result-type result) 'missed-ham))

(defun missed-spam-p (result)
  (eql (result-type result) 'missed-spam))

(defun correct-p (result)
  (eql (result-type result) 'correct))
```

有了这些函数，你可以轻易地使用我在第11章里讨论的列表和序列操作函数来解出并统计特定类型的结果。

```
SPAM> (count-if #'false-positive-p *results*)
6
SPAM> (remove-if-not #'false-positive-p *results*)
(:FILE #p"ham/5349" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999983107355541d0)
(:FILE #p"ham/2746" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.6286468956619795d0)
(:FILE #p"ham/3427" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9833753501352983d0)
(:FILE #p"ham/7785" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9542788587998488d0)
(:FILE #p"ham/1728" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.684339162891261d0)
(:FILE #p"ham/10581" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999924537959615d0))
```

你还可以使用`result-type`返回的符号作为哈希表或alist中的键。例如，你可以编写一个函数来打印结果中每种类型的个数和比例，该函数使用alist将每种类型和一个额外的符号`total`映射到一个计数上。

```
(defun analyze-results (results)
  (let* ((keys '(total correct false-positive
                        false-negative missed-ham missed-spam))
         (counts (loop for x in keys collect (cons x 0))))
    (dolist (item results)
      (incf (cdr (assoc 'total counts)))
      (incf (cdr (assoc (result-type item) counts))))
    (loop with total = (cdr (assoc 'total counts))
          for (label . count) in counts
          do (format t "~&~@(~a~):~20t~5d~,5t: ~6,2f%~%" 
                     label count (* 100 (/ count total)))))))
```

当给该函数传递一个由`test-classifier`生成的结果列表时，它将给出下面的输出：

```
SPAM> (analyze-results *results*)
Total:           3761 : 100.00%
Correct:         3689 : 98.09%
False-positive:  4 : 0.11%
False-negative:  9 : 0.24%
Missed-ham:     19 : 0.51%
Missed-spam:    40 : 1.06%
NIL
```

而在分析的最后，你可能想要了解一个单独的消息被分类成某种类型的原因。下面的函数将

为你显示这点：

```
(defun explain-classification (file)
  (let* ((text (start-of-file file *max-chars*))
         (features (extract-features text))
         (score (score features))
         (classification (classification score)))
    (show-summary file text classification score)
    (dolist (feature (sorted-interesting features))
      (show-feature feature)))))

(defun show-summary (file text classification score)
  (format t "~&~a" file)
  (format t "~2%~a~2%" text)
  (format t "Classified as ~a with score of ~,5f~%" classification score))

(defun show-feature (feature)
  (with-slots (word ham-count spam-count) feature
    (format
      t "~&~2t~a~30thams: ~5d; spams: ~5d;~,10tprob: ~,f~%"
      word ham-count spam-count (bayesian-spam-probability feature)))))

(defun sorted-interesting (features)
  (sort (remove-if #'untrained-p features) #'< :key #'bayesian-spam-probability))
```

## 23.10 接下来的工作

很明显，你可以用这些代码做更多的事。为了将它变成一个真正的垃圾邮件过滤应用程序，你需要找到一种方式来将它集成到你正常的电子邮件基础服务框架中。一种使它方便地与几乎任何电子邮件客户端相集成的思路是，编写一点儿代码来使它成为一个POP3代理。这是多数电子邮件客户端从邮件服务器上获取邮件所使用的协议，这样一个代理将从你的实际POP3服务器中获取邮件并为你的电子邮件客户端提供服务，在这个过程中它要么将垃圾邮件标记一个你的电子邮件客户端过滤器可以理解的信头，要么直接把它放在一边。当然你可能还需要一种方式来与过滤器沟通有关错误分类的信息。只要你把它设置成一个服务器，你就可以提供一个Web接口。第26章将谈及如何编写Web接口，第29章将为不同的应用构建Web接口。

或者你可能想要改进基本分类——一个可能的起点是令`extract-features`更加专业。特别地，你可以使分词器更聪明地处理电子邮件的内部结构，即可以为出现在消息体和消息头中的单词解出不同类型的特征。你还可以解码包括Base64和Quoted Printable在内的多种类型的消息编码，因为垃圾邮件发送者经常使用这些编码来扰乱它们的消息。

但是我将把这些改进留给你。现在你已准备好继续前进来构建一个流式MP3服务器，先从编写一个解析二进制文件的通用库开始。

# 实践：解析二进制文件



**在**本章里，我将向你介绍如何构建一个库，用来编写那些读取和写入二进制文件的代码。你将在第25章里使用这个库来编写一个ID3标签的解析器，ID3标签是用来保存MP3文件中诸如艺术家和专辑名这类元数据的机制。这个库同样也是一个关于如何使用宏来为语言扩展新的控制构造的示例，它将通用语言转化成了一种用于处理特定问题的专用语言，在本例中是读取和写入二进制数据。由于你将循序渐进地开发这个库，包括几个部分可用的版本，因此看起来你将会编写很多代码。但是当一切都已完成时，整个库的规模将会少于150行代码，而其中最长的宏也只有20行。

## 24.1 二进制文件

24

在一个足够低的抽象层面上，所有文件都是“二进制”的，因为看起来它们只是含有一些以二进制形式编码的数字罢了。不过，通常会把所谓“文本”文件和“二进制”文件区别看待，前者的所有数字都可以被解释成表示人类可读文本的字符，而后者所含有的数据如果被解释成字符的话，将会得到不可打印的字符。<sup>①</sup>

二进制文件格式通常被设计成可以简洁高效地进行解析，这是它们相对于基于文本的格式的主要优点。为了同时满足这些要求，它们通常采用可以轻易映射到程序内存中数据结构的磁盘结构来保存。<sup>②</sup>

即将编写的这个库将提供一种简单的方式，来定义那些在二进制文件所定义的磁盘结构和内存中Lisp对象之间的映射关系。使用这个库，编写一个可以读取二进制文件的程序将会很容易，先将其转化成可管理的Lisp对象，然后再写回到另一个正确格式化的二进制文件中。

- 
- ① 在ASCII编码中，前32个字符是不可打印的控制字符，最初用来控制终端服务器的行为，让其做到诸如通过喇叭发声、回退一个字符、换行以及将光标移到行首之类的操作。在这32个控制字符串里，通常只有三个可以在文本文件中看到：换行、回车以及水平制表（tab）。
  - ② 某些二进制文件格式确实是内存数据结构。在许多操作系统中，将一个文件直接映射进内存是有可能的，然后诸如C语言这样的底层语言就可以将含有文件内容的内存区域当作任何其他的内存来处理。写入该内存区域的数据在解除映射时会被写回到文件中。不过，这些格式都是平台相关的，因为即便像整数这样的简单数据类型在内存中的表示方式也取决于程序所运行的硬件。这样，任何倾向于可移植的文件格式都必须为其所有数据类型定义规范的数据表示，使其可以映射到特定类型的机器或编程语言的实际内存中的数据表示上。

## 24.2 二进制格式基础

读写二进制文件的起始点是打开一个用于读写单个字节的文件。如同第14章里讨论的那样，**OPEN**和**WITH-OPEN-FILE**都接受一个关键字参数：**:element-type**，它可以控制流传输的基本单元。当你在处理二进制文件时，需要把该参数设定为(**unsigned-byte 8**)。通过：**:element-type**打开的输入流将在每次传给**READ-BYTE**时返回一个介于0到255之间的整数。同样地，你可以通过向**WRITE-BYTE**传递介于0到255之间的数字来向一个(**unsigned-byte 8**)输出流写入字节。

在单独字节的层面之上，多数二进制格式都使用了一小组基本数据类型——以多种方式编码的数字、文本字符串以及位字段等，然后再复合成更复杂的结构。所以你的首要任务是定义一个用来编写那些读写给定二进制格式中使用的基本数据类型的框架。

先举一个简单的例子，假设你正在处理一个将无符号16位整数作为基本数据类型的二进制格式。为了读取这样一个整数，你需要读取两个字节，将一个字节乘以256，也就是 $2^8$ ，再跟另一个字节相加，从而将它们组合成单个整数。举个例子，假设指定这个16位量的二进制格式是以**big-endian**<sup>①</sup>形式保存的，那么以最重要字节优先的顺序，你可以用下面的函数来读取这样一个数：

```
(defun read-u2 (in)
  (+ (* (read-byte in) 256) (read-byte in)))
```

不过，Common Lisp提供了一种更便利的方式来处理这些按位处理。函数**LDB**，就是加载字节(**load byte**)的意思，可用来从一个整数中解出和设置(通过**SETF**)任意数量的连续位。<sup>②</sup>整数中的位数量和它们的位置由**BYTE**函数所创建的一个位描述符所指定。**BYTE**接受两个参数，即需要解出(或设置)的位数量以及最右边那一位相对整数中最不重要位来说以零开始的位置。**LDB**接受一个字节描述符和需要解出位数据的那个整数，然后返回由解出的位所代表的整数。这样，你就可以解出一个整数的最不重要的八位元：

```
(ldb (byte 8 0) #xabcd) → 205 ; 205 is #xcd
```

为了得到下一个八位元，可以使用字节描述符(**byte 8 8**)，如下所示：

```
(ldb (byte 8 8) #xabcd) → 171 ; 171 is #xab
```

可以将**LDB**与**SETF**配合使用来设置一个保存在可**SETF**的位置上的整数的指定位。

```
CL-USER> (defvar *num* 0)
*NUM*
CL-USER> (setf (ldb (byte 8 0) *num*) 128)
```

① 术语**big-endian**和它的反义词**little-endian**来自Jonathan Swift的*Gulliver's Travels* (《格列佛游记》)，用来表达一个字节的数字在诸如内存和文件中保存时所采用的字节顺序。例如，数字43981，其十六进制为abcd，当表示成16位量时由ab和cd两个字节所组成。对于一台电脑来说，只要各方意见一致，以何种顺序保存这两个字节都是无关紧要的。当然，尽管你可以在同样好的两种方式中任意选择，但可以保证的是并非人人都同意。要想了解关于此事的更多隐情，以及术语**big-endian**和**little-endian**是最早以这种含义应用在哪里的，可以阅读Danny Cohen的“On Holy Wars and a Plea for Peace”，地址是<http://khavrinen.lcs.mit.edu/wollman/ien-137.txt>。

② **LDB**跟与之相关的函数**DPB**均是来自DEC PDP-10计算机的汇编函数，它们本质上做相同的事情。无论特定Common Lisp实现使用的是何种内部表示法，两个函数都运行在以二进制补码表示的整数上。

```

128
CL-USER> *num*
128
CL-USER> (setf (ldb (byte 8 8) *num*) 255)
255
CL-USER> *num*
65408

```

因此，也可以这样来编写read-u2：<sup>①</sup>

```
(defun read-u2 (in)
  (let ((u2 0))
    (setf (ldb (byte 8 8) u2) (read-byte in))
    (setf (ldb (byte 8 0) u2) (read-byte in)))
  u2))
```

为了把一个数字写成16位整数，需要解出单独的8位字节并逐个地写它们。为了解出单独的字节，只需以同样的字节描述符来使用LDB就可以了。

```
(defun write-u2 (out value)
  (write-byte (ldb (byte 8 8) value) out)
  (write-byte (ldb (byte 8 0) value) out))
```

当然，你也可以用许多其他的方式来编码整数——使用不同的字节数、不同的尾部处理(endianess)以及有符号或无符号的格式。

24

## 24.3 二进制文件中的字符串

文本字符串是另一种可能在许多二进制格式中遇到的基本数据类型。当你逐字节地读取文件时，不能直接读写字符串，你需要每次一个字节地对它们进行解码或者编码，就像你对二进制编码的数字所做的那样。并且正如你可以用多种方式来编码一个整数一样，你也可以用多种方式来编码一个字符串。不过最起码来讲，二进制格式必须指定究竟需要编码多少个单独的字符。

为了将字节转化成字符，你既需要知道字符的编码(code)，也需要知道编码方式(encoding)。一个字符编码定义了从正整数到字符之间的映射。映射表中的每个数字被称为一个代码点(code point)。例如，ASCII就是一种字符编码，它将0到127之间的数字映射到了拉丁字母表的一些特定

<sup>①</sup> Common Lisp也提供了用来对整数进行移位和处理掩码的函数，这种方式可能对C和Java程序员来说更熟悉些。例如，你还可以用第三种方式编写read-u2，像下面这样使用那些函数：

```
(defun read-u2 (in)
  (logior (ash (read-byte in) 8) (read-byte in)))
```

该函数几乎跟下面的Java方法完全等价：

```
public int readU2 (InputStream in) throws IOException {
  return (in.read() << 8) | (in.read());
}
```

名字LOGIOR和ASH是LOGical Inclusive OR（逻辑同或）和Arithmetic SHift（算术移位）的简称。ASH以给定的位数对一个整数进行移位，当其第二个参数为正时左移，为负时右移。LOGIOR通过对整数的每个位做逻辑或来将它们合并在一起。另一个函数LOGAND可以做按位与，这可用来掩盖特定的位。尽管如此，对于本章和接下来的章节里你将用到的各种按位操作，使用LDB和BYTE将是既便利又符合习惯的Common Lisp风格。

字符上。另一方面，字符编码定义了代码点在诸如文件这种基于字节的媒体中是如何被表示成一个字节序列的。对于那些使用八位或者更少位的编码，例如ASCII和ISO-8859-1，编码方式是相当直接的——每一个数值刚好编码成单个字节。

相对直接的是纯粹的双字节编码方式，例如UCS-2，它在16位值和字符之间做映射。双字节编码方式比单字节编码方式更复杂的唯一原因就是你可能还需要知道那些16位的值究竟是编码成big-endian还是little-endian格式的。

变长的编码方式对于不同的数值使用不同数量的八位元，这会使其更加复杂但却令它们在许多时候更加紧凑。例如，UTF-8是一种设计用于Unicode字符代码的编码方式，它使用单个八位元来编码0到127之间的值，同时使用至多四个八位元来编码最多1 114 111个不同的值。<sup>①</sup>

由于在Unicode字符集中0到127的代码点映射到与ASCII相同的字符上，一段UTF-8编码的只由ASCII字符构成的文本与ASCII编码的结果是相同的。另一方面，对于几乎完全由UTF-8中需要用四个字节来表示的字符构成的文本，如果用直接的双字节编码方式来编码的话，反而会更紧凑。

Common Lisp提供了两个函数用来在数值的字符代码和字符对象之间进行转换：**CODE-CHAR**接受一个数值代码并返回一个字符，而**CHAR-CODE**则接受一个字符并返回其数值的代码。语言标准并未指定一个实现必须使用的字符编码方式，因此并不保证你可以表示有可能作为一个Lisp字符被编码进给定二进制文件格式中的每一个字符。不过，几乎所有的现代Common Lisp实现都使用ASCII、ISO-8859-1或Unicode作为其原生的字符编码。由于Unicode是ISO-8859-1的超集，而ISO-8859-1则是ASCII的超集，如果你正在使用一个支持Unicode的Lisp平台，那么CODE-CHAR和CHAR-CODE将可以直接用来转换这三种编码方式中的任何一种。<sup>②</sup>

除了指定字符编码方式以外，一个字符串的编码工作还必须指定如何编码字符串的长度。有三种技术通常用在二进制文件格式中。

最简单的方式是不编码，而是让它成为字符串在更大的结构中某个位置上的隐含值：一个文件中的特定元素可能总是一个特定长度的字符串，或者一个字符串可能是一个变长数据类型中的最后一个元素，结构的总长度决定了有多少剩余字节可被用来作为字符串数据读取。这两种方式都用在了ID3标签中，正如你将在下一章里看到的那样。

另外两种技术可用来在无需依赖上下文的情况下编码变长的字符串。一种方式是先编码字符串的长度再跟上字符数据——解析器先读取一个整数值（以某种特定的整数格式）再读取相应数量的字符。另一种方式是先写入字符数据后跟一个不可能出现在字符串中的定界符，例如空字符。

不同的表示法各自具有不同的优点和缺点，但当你已经在处理指定的二进制格式时，你就无法控制究竟使用哪种编码方式了。不过，没有哪种编码方式比其他方式是特别难以读写的。举一

<sup>①</sup> UTF-8最初被设计用来表示31位的字符代码，并在每个代码点上使用至多六个字节。不过，Unicode代码点的最大值是#x10ffff，因此一个UTF-8编码的Unicode字符在每个代码点上只需至多四个字节就够了。

<sup>②</sup> 如果你需要解析一个用到了其他字符编码的文件格式，或者需要通过一个非Unicode的Common Lisp实现来解析一个含有任意Unicode字符串的文件，那么你总是可以在内存中将这些字符串表示成整数代码点的向量。它们不会成为Lisp字符串，因为你无法使用字符串函数来管理或比较它们，但你可以像对任意向量那样对它们做任何事情。

个例子，下面是一个用来读取空字符结尾的ASCII字符串的函数，假设你的Lisp实现使用了ASCII，或是诸如ISO-8859-1或完全的Unicode这两个它的超集作为原生字符编码：

```
(defconstant +null+ (code-char 0))

(defun read-null-terminated-ascii (in)
  (with-output-to-string (s)
    (loop for char = (code-char (read-byte in))
          until (char= char +null+) do (write-char char s))))
```

其中的**WITH-OUTPUT-TO-STRING**宏是我在第14章里提到过的，这是一种在你不知道长度的情况下构造字符串的简单方式。它创建了一个**STRING-STREAM**并将其绑定到特定的变量名上，这里是s。所有写入流的字符都被收集到一个字符串中，并随后作为**WITH-OUTPUT-TO-STRING**形式的值返回。

为了写回一个字符串，你只需将字符转换回可以用**WRITE-BYTE**来写的数值形式，然后再在字符串内容后面写入一个空终止符即可。

```
(defun write-null-terminated-ascii (string out)
  (loop for char across string
        do (write-byte (char-code char) out))
  (write-byte (char-code +null+) out))
```

如同这些示例所显示的，读写二进制文件中基本元素的主要智力挑战是理解究竟该如何解释出现在一个文件中的字节并将其映射到Lisp数据类型。如果一个二进制格式是良好定义的，那么这将是一个相当直接的命题。事实上正如它们所说的，编写函数来读写一个特定的编码根本就是小事一桩。

24

现在你可以转而考虑读写更复杂的磁盘结构，以及如何将它们映射到Lisp对象上的问题了。

## 24.4 复合结构

二进制格式通常用来表示那些可以轻易映射到内存数据结构上的数据。因此，不难理解那些复合的磁盘结构通常是以一种接近于编程语言定义内存中数据结构的方式来定义的。通常，一个复合的磁盘结构由一些命名的部分所组成，每个部分其本身要么是诸如数字或字符串这样的基本类型，要么是另一个复合结构，或者可能是这些值的一个集合。

例如，一个定义在2.2版本规范中的ID3标签包括：一个三字符的ISO-8859-1字符串（始终是“ID3”）的头部，两个用来指定规范的主版本和修订号的单字节无符号整数，八位的布尔旗标以及四个以特定于ID3规范的编码方式编码整个标签长度的字节。紧接着头部的是一个帧的列表，每个帧都有其自己的内部结构。在帧之后是填满头部所指定的标签长度所需的数量相当的空字节。

如果你以面向对象的眼光来看的话，复合结构会和类很像。例如，你可以编写一个类来表示ID3标签。

```
(defclass id3-tag ()
  ((identifier :initarg :identifier :accessor identifier)
   (major-version :initarg :major-version :accessor major-version))
```

```
(revision      :initarg :revision      :accessor revision)
(flags        :initarg :flags        :accessor flags)
(size         :initarg :size         :accessor size)
(frames       :initarg :frames       :accessor frames)))
```

这个类的实例将成为保存ID3标签的完美仓库。随后你可以编写函数来读写该类的实例。例如，假设已有了用来读取适当基本数据类型的特定的其他函数，那么函数read-id3-tag如下所示：

```
(defun read-id3-tag (in)
  (let ((tag (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) tag
      (setf identifier      (read-iso-8859-1-string in :length 3))
      (setf major-version   (read-u1 in))
      (setf revision        (read-u1 in))
      (setf flags           (read-u1 in))
      (setf size            (read-id3-encoded-size in))
      (setf frames          (read-id3-frames in :tag-size size)))
    tag))
```

函数write-id3-tag具有类似的结构，你需要使用适当的write-\*函数来输出那些保存在id3-tag对象中的值。

不难看出你应该怎样编写一个适当的类来表示一个规范中的所有复合数据类型，以及用于每个类和必要的基本类型的read-foo和write-foo函数。但是很容易也可以看出所有用来读和写的函数都将会非常相似，区别仅在于指定它们要读取的类型和它们所保存在槽中的名字。这实在太浪费笔墨了，尤其是当你发现在ID3规范中它只用了四行文本来描述一个ID3标签的结构，而你已经写了八行代码却还没写到write-id3-tag。

你真正想要的是一种以类似规范中伪代码的形式来描述像ID3标签这样的结构的方式，随后这些描述可以被展开成定义了id3-tag类的代码，以及在磁盘上的字节和类实例之间相互转换的函数。听起来这正是宏的任务。

## 24.5 设计宏

由于你已经对宏需要生成怎样的代码有了大致的想法，根据第8章归纳的宏编写过程，下一步就是要切换视角，转而思考这样一个宏的具体调用将会是怎样的。因为目标是可以书写像ID3规范中的伪代码一样紧凑的东西，所以你可以从那里开始。指定一个ID3标签头部的方式如下所示：

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size                 4 * %0xxxxxxxx
```

在规范的写法里，这意味着一个ID3标签的“文件标识符”是ISO-8859-1编码的字符串“ID3”。版本部分由两个字节构成，对于当前版本的规范来说，其中第一个字节的值为2，第二个字节是0。用于保存旗标的槽有8个位，其中除了前两个以外都是0，其长度是4个字节，每个字节的最重要的位上都是0。

还有一些信息没有被上面的伪代码覆盖到。例如，编码了长度的4个字节究竟是如何被解释

为用几行文字来描述的。同样地，规范用文字描述了怎样才能编写代码来读和写由这个伪代码所指定的一个ID3标签。这样，你应该可以写出该伪代码的一个S-表达式版本并将其展开成原本需要手写的类和函数的定义，比如说可能是类似下面这样：

```
(define-binary-class id3-tag
  ((file-identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)
   (frames (id3-frames :tag-size size))))
```

这个形式的基本思想是定义一个类似于由`DEFCLASS`所定义的`id3-tag`类，但和指定诸如`:initarg`和`:accessor`之类的东西所不同的是，每个槽描述符由槽的名字——`file-identifier`、`major-version`等，以及关于该槽在磁盘中如何表示的信息所构成。由于目前这些都还只是一点儿随想，所以你不必担心宏`define-binary-class`究竟是如何对诸如`(iso-8859-1-string :length 3)`、`u1`、`id3-tag-size`和`(id3-frames :tag-size size)`这些表达式进行处理的。对你来说，只要每个表达式都含有对于如何读写一个特定数据编码的必要信息就可以了。

## 24.6 把梦想变成现实

24

那么，对于优美代码的幻想就到此为止吧。现在你需要开始编写`define-binary-class`了——编写代码将那个关于ID3标签的样子的简洁表达方式转化成实际可用的代码：在内存中表示它、从磁盘中读取以及将其写入磁盘。

首先，你应该为这个库定义一个包。下面是你可以从本书Web站点上下载到的版本中的包定义文件：

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.binary-data
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :define-binary-class
           :define-tagged-binary-class
           :define-binary-type
           :read-value
           :write-value
           :*in-progress-objects*
           :parent-of-type
           :current-binary-object
           :+null+))
```

其中的`COM.GIGAMONKEYS.MACRO-UTILITIES`包里含有第8章的宏`with-gensyms`和`once-only`。

由于你已经有了想要生成的代码的手写版本，编写这样一个宏应该不会太难。可以分而治之，先写一个只生成`DEFCLASS`形式的`define-binary-class`版本。

如果回过头来观察那个`define-binary-class`形式，你将看到它接受两个参数：名字

id3-tag以及一个槽描述符的列表，后者的每一个都是两元素列表。你需要从这些材料中构造出适当的DEFCLASS形式来。很明显地，define-binary-class形式与一个正确的DEFCLASS形式之间最大的区别就在槽描述符中。来自define-binary-class的单个槽描述符如下所示：

```
(major-version u1)
```

但这并不是一个合法的DEFCLASS槽描述符。相反，你需要类似下面的东西：

```
(major-version :initarg :major-version :accessor major-version)
```

其实很简单。首先定义一个简单的函数将一个符号转换成对应的关键字符串。

```
(defun as-keyword (sym) (intern (string sym) :keyword))
```

现在定义一个函数，其接受一个define-binary-class槽描述符并返回一个DEFCLASS槽描述符。

```
(defun slot->defclass-slot (spec)
  (let ((name (first spec)))
    `',(name :initarg ,(as-keyword name) :accessor ,name)))
```

在你使用IN-PACKAGE调用切换到新包以后，你可以在REPL中测试这个函数。

```
BINARY-DATA> (slot->defclass-slot '(major-version u1))
(MAJOR-VERSION :INITARG :MAJOR-VERSION :ACCESSOR MAJOR-VERSION)
```

看起来不错。现在define-binary-class的第一个版本可以轻松搞定了。

```
(defmacro define-binary-class (name slots)
  `'(defclass ,name ()
    ,(mapcar #'slot->defclass-slot slots)))
```

这是一个简单的模板风格的宏。通过插入类的名字和槽描述符列表，define-binary-class生成一个DEFCLASS形式，其中槽描述符列表的构造方法是将函数slot->defclass-slot应用到define-binary-class形式的槽描述符列表的每个元素上。

为了查看这个宏究竟生成了什么代码，你可以在REPL中求值下面的表达式。

```
(macroexpand-1 '(define-binary-class id3-tag
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size)))))
```

为了更好的可读性，这里对得到的结果稍微重新格式化了一下，它应当看起来很眼熟，因为正是你早些时候手工编写的那个类定义：

```
(defclass id3-tag ()
  ((identifier      :initarg :identifier      :accessor identifier)
   (major-version  :initarg :major-version  :accessor major-version)
   (revision       :initarg :revision       :accessor revision)
   (flags          :initarg :flags          :accessor flags)
   (size           :initarg :size           :accessor size)
   (frames         :initarg :frames         :accessor frames)))
```

## 24.7 读取二进制对象

下一步你需要让`define-binary-class`也能生成一个函数以读取这个新类的实例。回顾你之前写的`read-id3-tag`函数，看起来有些滑稽，因为`read-id3-tag`的存在并不是很正常——为了读取每一个槽的值，你不得不调用一个不同的函数。更不用说函数`read-id3-tag`的名字，尽管来自你所定义的类的名字，但其本身却并不是`define-binary-class`的参数，因此没有办法像类名那样直接插入到模板中。

你可以通过设计并遵循一个命名约定来处理这两个问题，让宏可以基于槽描述符中的类型名来找出需要调用的函数名。不过，这将需要`define-binary-class`来生成名字`read-id3-tag`，这是有可能的但不是个好主意。创建全局定义的宏通常应当仅使用那些由调用者传递给它们的名字；背后生成名字的宏可能会在生成的名字和其他地方使用的名字刚好同名时导致难以预测且难以调试的名字冲突。<sup>①</sup>

你可以避免这些不便，只要你注意到所有这些读取一个特定类型值的函数都有相同的基本目的：从一个流中读取指定类型的值。说白了它们都是单个通用操作的实例。除此之外，对于“通用”(generic)的使用应当让你直接想到问题的解决方案：与其定义一堆互不相关的、名字各不相同的函数，还不如定义一个广义函数`read-value`，以及特定用来读取不同类型值的方法。

这就是说，不必定义函数`read-iso-8859-1-string`和`read-u1`，你可以将`read-value`定义成一个接受两个必要参数，一个类型和一个流，甚至是一些关键字参数的广义函数。

24

```
(defgeneric read-value (type stream &key)
  (:documentation "Read a value of the given type from the stream."))
```

通过指定`&key`而不带有任何实际关键字参数，你可以允许不同的方法定义它们自己的`&key`参数而不做具体要求。不过这意味着每个特化在`read-value`上的方法都将在它们的形参列表中至少包括`&key`或`&rest`参数中的一个，这样才能与广义函数兼容。

然后，你定义使用`EQL`特化符将类型参数特化在你想要读取的类型名上的方法。

```
(defmethod read-value ((type (eql 'iso-8859-1-string)) in &key length) ...)
(defmethod read-value ((type (eql 'u1)) in &key) ...)
```

接下来，你就可以让`define-binary-class`生成一个特化在类型名`id3-tag`上的`read-value`方法了，而该方法可以通过调用把适当的槽类型作为第一个参数的`read-value`来实现。你想要生成的代码如下所示：

```
(defmethod read-value ((type (eql 'id3-tag)) in &key)
  (let ((object (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) object
      (setf identifier (read-value 'iso-8859-1-string in :length 3))
      (setf major-version (read-value 'u1 in))
      (setf revision (read-value 'u1 in))))
```

<sup>①</sup> 不幸的是，语言本身在这个观点上并没有提供一个好的榜样：宏`DEFSTRUCT`被`DEFCCLASS`所取代，因此我不打算讨论它。宏`DEFSTRUCT`可以基于给定结构的名字来生成新的函数名，其不良示例导致了许多初级的宏编写者效仿。



```
(setf flags          (read-value 'u1 in))
(setf size           (read-value 'id3-encoded-size in))
(setf frames         (read-value 'id3-frames in :tag-size size)))
(object))
```

因此，就像你需要一个函数来将define-binary-class槽描述符转化成DEFCLASS槽描述符那样，现在你也需要一个接受define-binary-class槽描述符作为参数并生成适当SETF形式的函数，也就是说，接受形式

```
(identifier (iso-8859-1-string :length 3))
```

并返回下面结果的函数：

```
(setf identifier (read-value 'iso-8859-1-string in :length 3))
```

不过，上面的代码和DEFCLASS的槽描述符有一点儿区别：它包含了对一个变量in的引用，该变量是来自read-value方法的方法参数而并非来源于槽描述符。它不一定非叫做in，但无论你使用什么名字，它都必须跟你用在方法参数列表和其他read-value调用中的名字相同。眼下你可以避开关于这个名字来源的问题，定义slot->read-value来接受一个流变量作为第二个参数。

```
(defun slot->read-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(setf ,name (read-value ',type ,stream ,@args))))
```

函数normalize-slot-spec用来正则化槽描述符的第二个元素，将类似u1这样的符号转化成列表(u1)，从而让DESTRUCTURING-BIND可以解析它。如下所示：

```
(defun normalize-slot-spec (spec)
  (list (first spec) (mklist (second spec))))
```

```
(defun mklist (x) (if (listp x) x (list x)))
```

你可以使用各种类型的槽描述符来测试slot->read-value。

```
BINARY-DATA> (slot->read-value '(major-version u1) 'stream)
(SETF MAJOR-VERSION (READ-VALUE 'U1 STREAM))
BINARY-DATA> (slot->read-value '(identifier (iso-8859-1-string :length 3)) 'stream)
(SETF IDENTIFIER (READ-VALUE 'ISO-8859-1-STRING STREAM :LENGTH 3))
```

有了这些函数，你就可以将read-value添加到define-binary-class中了。如果你取一个手写的read-value方法并去掉任何特定类相关的内容，那么你将得到这样一个骨架：

```
(defmethod read-value ((type (eql ...)) stream &key)
  (let ((object (make-instance ...)))
    (with-slots (...) object
      ...
      object)))
```

所有要做的就是将这个骨架添加到define-binary-class模板中，把其中的省略号部分替换成适当的名字和代码。你也可能会想要把变量type、stream和object替换成由符号生成的名字以避免潜在的槽名字冲突，<sup>①</sup>这可以通过使用第8章的with-gensyms宏来实现。

<sup>①</sup> 从技术上来讲，type或object不可能与槽名字冲突，最坏情况是它们会在WITH-SLOTS形式中被掩盖掉。不过，简单地用GENSYM来生成一个宏模板中用到的所有局部变量肯定是无害的。

另外，由于一个宏必须展开成单一形式，你需要在DEFCLASS和DEFMETHOD的外面包装一些形式。**PROGN**习惯上用来让宏可以展开成多个定义，因为当它出现在一个文件的顶层时可以得到文件编译器的特殊对待，第20章曾讨论过这点。

所以，你可以将define-binary-class改成下面这样：

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ())
      ,(mapcar #'slot->defclass-slot slots))

      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let ((,objectvar (make-instance ',name)))
          (with-slots ,(mapcar #'first slots) ,objectvar
            ,(,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)
              ,objectvar))))
```

## 24.8 写二进制对象

生成用来写一个二进制类实例的代码将会做类似的处理。首先，你可以定义一个write-value广义函数。

```
(defgeneric write-value (type stream value &key)
  (:documentation "Write a value as the given type to the stream."))
```

其次，定义一个助手函数，将一个define-binary-class槽描述符转换成使用write-value来输出槽数据的代码。和slot->read-value函数一样，这个助手函数需要接受流变量的名字作为一个参数。

```
(defun slot->write-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(write-value ',type ,stream ,name ,@args)))
```

现在你可以在define-binary-class宏里添加一个write-value模板。

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ())
      ,(mapcar #'slot->defclass-slot slots))

      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let ((,objectvar (make-instance ',name)))
          (with-slots ,(mapcar #'first slots) ,objectvar
            ,(,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)
              ,objectvar)))

      (defmethod write-value ((,typevar (eql ',name)) ,streamvar ,objectvar &key)
        (with-slots ,(mapcar #'first slots) ,objectvar
          ,(,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

## 24.9 添加继承和标记的结构

尽管这个版本的define-binary-class能够处理独立的结构，但二进制文件格式通常定义了一些可以自然地采用子类和基类来建模的磁盘结构。因此你可能想要扩展define-binary-class来支持继承。

一个相关的用在许多二进制格式中的技术是存在于一些磁盘上的结构，其确切类型只有在读取了一些用来指示如何解析后续字节的数据以后才能决定。例如，ID3标签中的大量帧全都共享了一个由字符串标识和长度所构成的统一的头结构。为了读取一个帧，你需要先读取标识符再用它的值来检测你正在查看的是哪一种帧类型，以及如何解析该帧的主体。

当前的define-binary-class宏没有办法处理这种类型的读取操作，你可以使用define-binary-class来定义一个代表每种帧类型的类，但如果你没有至少读取标识符部分的话就无法知道这是哪个类型的帧。而如果其他代码读取了标识符以检测用来传给read-value的类型，那么这会打断read-value的运行，因为它需要读取构成它所实例化的类实例的全部数据。

你可以为define-binary-class添加继承来解决这个问题，并编写另一个宏define-tagged-binary-class用来定义那些“抽象”类。后者并不直接被实例化，而是可以被那些知道如何读取足够数据来决定创建何种类型的类的read-value方法所特化。

为define-binary-class添加继承的第一步是为该宏添加一个参数来接受一个基类的列表。

```
(defmacro define-binary-class (name (&rest superclasses) slots) ...)
```

然后，在DEFCLASS模板中插入该值以取代原先的空列表。

```
(defclass ,name ,superclasses
  ...)
```

不过，你还需要改变read-value和write-value方法，这样在定义基类时所生成的方法才可以被那些由子类生成的方法用来读写继承的槽。

当前的read-value工作方式尤其有问题，因为它在填入内容之前就要实例化对象。很明显，你不可能让方法通过读取基类的字段来实例化一个对象，同时让子类的方法去实例化并填充另一个不同的对象。

你可以通过将read-value划分成两部分来解决这个问题：一部分用来实例化正确类型的对象，而另一部分则用来填充一个已存在对象的槽。写的方面其实更简单，但你可以使用同样的技术。

因此，你可以定义两个新的广义函数read-object和write-object，它们都接受一个已存在的对象和一个流。定义在这些广义函数上的方法将用来读写特定于它们所特化的对象所属的类的槽。

```
(defgeneric read-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Fill in the slots of object from stream."))
```

```
(defgeneric write-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Write out the slots of object to the stream."))
```

把这些广义函数定义成使用带有:most-specific-last选项的PROGN方法组合的形式，继而你可以定义特化在object的每个二进制类上的方法，并让它们只处理实际定义在该类中的槽。PROGN方法组合将合并所有可应用的方法并让继承体系中最不相关的类首先运行，接着读写定义在该类中的槽，然后特化在下一个最不相关子类上的方法再运行，依此类推。而由于所有对于特定类的重量级操作现在都由read-object和write-object来完成了，你甚至不需要再定义特化了的read-value和write-value方法。你可以定义默认方法，其中假设类型参数就是一个二进制类的名字。

```
(defmethod read-value ((type symbol) stream &key)
  (let ((object (make-instance type)))
    (read-object object stream)
    object))

(defmethod write-value ((type symbol) stream value &key)
  (assert (typep value type))
  (write-object value stream))
```

注意你是怎样将MAKE-INSTANCE用作一个通用的对象工厂的。尽管通常情况下由于确切知道想要实例化的类，你会使用一个引用了的符号作为第一个参数来调用MAKE-INSTANCE，但你也可以使用任何求值成一个类名的表达式来调用这个函数。本例则使用了read-value方法中的type参数。

define-binary-class中实际的改变相对较少，现在是定义read-object和write-object而不是read-value和write-value上的方法了。

```
(defmacro define-binary-class (name superclasses slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

24

## 24.10 跟踪继承的槽

目前的定义适用于很多情形。不过，它无法处理一种相当普遍的情形，即子类需要引用其槽规范中所继承的槽的情形。例如，在当前的define-binary-class定义下，你可以像这样定义单个类：

```
(define-binary-class generic-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)
   (data (raw-bytes :bytes size))))
```

在data规范中，对size的引用可以按照你预想的方式来进行，因为这些表达式是在该对象的全部槽的**WITH-SLOTS**的封装下读写data槽的。不过，如果你试图将上面的类像这样分开定义在两个槽里：

```
(define-binary-class frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))

(define-binary-class generic-frame (frame)
  ((data (raw-bytes :bytes size))))
```

你将在编译generic-frame定义时得到一个编译期警告，然后在你试图使用它时得到一个运行期错误，因为在特化于generic-frame的read-object和write-object方法中没有以词法形态出现的变量size。

你需要做的是跟踪由每个二进制类定义的槽，并将通过继承得到的槽包含在read-object和write-object方法的**WITH-SLOTS**形式中。

跟踪这类信息最简单的方法是从命名类的符号下手。如同我在第13章里讨论过的，每个符号对象都有一个与之关联的属性列表，属性列表可通过函数SYMBOL-PLIST和GET来访问。你可以把任意的键值对用GET的SETF添加到一个符号的属性表中，从而将这些信息与该符号关联起来。举个例子，如果二进制类foo定义了三个槽x、y和z，那么在跟踪这一事实时，你可以采用下面的表达式将一个slots键添加到符号foo的属性表中，值为(x y z)：

```
(setf (get 'foo 'slots) '(x y z))
```

你希望这份备忘能够作为求值foo的define-binary-class的一部分。不过，对于在何处放置这个表达式仍然不甚明了。如果你在计算宏的展开式时对其求值，那么它将在你编译define-binary-class形式时被求值，但当你以后加载了含有编译后代码的文件时就不会再求值了。另一方面，如果你将该表达式包含到展开式中，那么它将不会在编译期被求值，这意味着如果你编译了一个带有几个define-binary-class形式的文件，在编译过程中关于这些类都定义了哪些槽的信息将是不可见的，直到整个文件被加载以后才会出现，而这时已经太晚了。

这就是我在第20章里讨论的用特殊操作符EVAL-WHEN处理的问题。通过将一个形式封装在EVAL-WHEN中，你可以控制它是在编译期还是编译后代码的加载期运行，或是在两个时期都运行。你希望在编译一个宏形式的过程中窃取一些信息，并且希望在编译后的形式被加载时仍然有效。对于这样的需求，你应当把它包装在一个类似下面这样的EVAL-WHEN中：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get 'foo 'slots) '(x y z)))
```

然后把EVAL-WHEN包含在宏所生成的展开式中。这样，你可以将下列形式添加到由define-binary-class生成的展开式中，从而保住一个二进制类和它的直接基类的槽信息：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'slots) ',(mapcar #'first slots))
  (setf (get ',name 'superclasses) ',superclasses))
```

现在你可以定义三个助手函数来访问这些信息。第一个函数简单地返回由一个二进制类直接定义的槽。让该函数返回列表的复本是个好主意，因为你不希望其他代码在二进制类已经被定义之后再去修改其槽列表。

```
(defun direct-slots (name)
  (copy-list (get name 'slots)))
```

下一个函数返回从其他二进制类中继承的槽。

```
(defun inherited-slots (name)
  (loop for super in (get name 'superclasses)
        nconc (direct-slots super)
        nconc (inherited-slots super)))
```

最后，你可以定义一个函数，其返回包含所有直接定义和继承得到的槽名称的列表。

```
(defun all-slots (name)
  (nconc (direct-slots name) (inherited-slots name)))
```

当你在计算**define-binary-class**形式的展开式时，你想要生成包含所有由新类及其全部基类定义的槽的名字的**WITH-SLOTS**形式。不过，你不能在生成展开式的时候使用**all-slots**，因为所需的信息只有在展开式被编译以后才可用。反之，你应当使用下面的函数，它接受传递给**define-binary-class**的类描述符和基类列表，并用它们来计算所有新类的槽列表：

```
(defun new-class-all-slots (slots superclasses)
  (nconc (mapcan #'all-slots superclasses) (mapcar #'first slots)))
```

一旦定义了这些函数，你就可以改变**define-binary-class**来保存当前被定义类的信息，并用已保存的基类的槽信息来生成你想要的**WITH-SLOTS**形式，如下所示：

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots , (new-class-all-slots slots superclasses) ,objectvar
           ,(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots , (new-class-all-slots slots superclasses) ,objectvar
           ,(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

## 24.11 带有标记的结构

一旦可以定义二进制类来扩展其他二进制类，你就可以定义一个新的宏来定义那些表示“带有标记”的结构的类了。读取带有标记的结构的策略是定义一个特化的read-value方法，它知道如何读取结构开始部分的值并使用这些值来决定哪个子类将被实例化。然后它用MAKE-INSTANCE生成该类的一个实例，同时将已经读取的值作为起始参数来传递，接着再将该对象传给read-object，从而由该对象实际所属的类来决定如何读取结构的其余部分。

这个新的宏define-tagged-binary-class看起来像是带有附加的一个:dispatch选项的define-binary-class，该选项指定一个求值到某二进制类名的形式。:dispatch形式在由带有标记的类定义的槽名称被绑定到从文件中所读取到的值的上下文中被求值。它返回的类必须接受对应于由带有标记的类定义的槽名称的起始参数。如果:dispatch形式总是求值到该标记类的子类的名字上，那么这个要求可以直接满足。

举个例子，假设你有一个函数find-frame-class，它将一个字符串标识符映射到代表特定类型的ID3帧的二进制类上，那么你可以定义一个带有标记的二进制类id3-frame，如下所示：

```
(define-tagged-binary-class id3-frame ()
  ((id    (iso-8859-1-string :length 3))
   (size  u3))
  (:dispatch (find-frame-class id)))
```

define-tagged-binary-class的展开式将含有一个DEFCLASS和一个就像define-binary-class的展开式那样的write-object方法，但它没有read-object方法，而是含有一个看起来像下面这样的read-value方法：

```
(defmethod read-value ((type (eql 'id3-frame)) stream &key)
  (let ((id (read-value 'iso-8859-1-string stream :length 3))
        (size (read-value 'u3 stream)))
    (let ((object (make-instance (find-frame-class id) :id id :size size)))
      (read-object object stream)
      object)))
```

由于define-tagged-binary-class和define-binary-class的展开式除了读方法以外都是相同的，你可以将它们的共同点分离出来放在一个助手宏define-generic-binary-class里，它接受读方法作为一个参数并将其插入到自己的展开式里。

```
(defmacro define-generic-binary-class (name (&rest superclasses) slots read-method)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       ,read-method))
```

```
(defmethod write-object progn ((,objectvar ,name) ,streamvar)
  (declare (ignorable ,streamvar))
  (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
    ,(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))
```

现在你可以同时定义define-binary-class和define-tagged-binary-class来展开成一个对define-generic-binary-class的调用了。下面是一个新版本的define-binary-class，当其完全展开时可以生成和之前的版本相同的代码：

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-object progn ((,objectvar ,name) ,streamvar)
        (declare (ignorable ,streamvar))
        (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
          ,(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))))
```

而下面是define-tagged-binary-class的定义以及它所用到的两个新的助手函数：

```
(defmacro define-tagged-binary-class (name (&rest superclasses) slots &rest options)
  (with-gensyms (typevar objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let* ,(mapcar #'(lambda (x) (slot->binding x streamvar)) slots)
          (let ((,objectvar
                  (make-instance
                    ,(or (cdr (assoc :dispatch options))
                          (error "Must supply :dispatch form."))
                    ,(mapcan #'slot->keyword-arg slots)))
                (read-object ,objectvar ,streamvar)
                ,objectvar))))
      (defun slot->binding (spec stream)
        (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
          `',(name (read-value ',type ,stream ,@args))))
      (defun slot->keyword-arg (spec)
        (let ((name (first spec)))
          `',(as-keyword name) ,name))))
```

24

## 24.12 基本二进制类型

尽管define-binary-class和define-tagged-binary-class令复合结构的定义变得简单了，但你仍然不得不手工编写用于基本数据类型的read-value和write-value方法。你可以决定保持现状，指定该库的用户必须编写适当的read-value和write-value方法来支持他们的二进制类所使用的基本类型。

不过，除了针对如何编写合适的read-value/write-value对写些文档以外，你还可以提供一个宏来自动地做到这点。这样做的另一个优点是让define-binary-class所创建的抽象更加圆满。目前，define-binary-class依赖于以特殊方式定义的read-value和write-value

方法，但这只是一种实现细节。通过定义一个对基本类型生成read-value和write-value方法的宏，你可以将那些细节隐藏在你所控制的抽象层面上。如果你以后决定改变define-binary-class的实现，那么你可以改变你的基本类型定义宏来满足新的需求而无需对使用二进制格式库的代码做任何改变。

所以你应当定义最后一个宏define-binary-type，它将生成用来读写代表已有类的实例的值，而不是由define-binary-class定义的类的实例的值。

举一个简单的例子，考虑一个用在id3-tag类中的类型，一个以ISO-8859-1编码的定长字符串。如以往一样，假设你的Lisp使用的原生字符集是ISO-8859-1或它的一个超集，这样你就可以使用CODE-CHAR和CHAR-CODE来将字节和字符相互转化了。

和以往一样，你的目标是编写一个宏，你可以仅表达必要的用来生成所需代码的信息。在本例中，共有4个部分的本质信息：类型名iso-8859-1；应当被read-value和write-value方法所接受的&key参数，在这里是length；从流中做读操作的代码；向一个流中做写操作的代码。

下面是一个含有这四部分信息的表达式：

```
(define-binary-type iso-8859-1-string (length)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in)))))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

现在你只需要一个宏来接受上面的形式，再将两个DEFMETHOD的形式一起封装到一个PROGN中就可以了。如果你像下面这样定义了define-binary-type的参数列表：

```
(defmacro define-binary-type (name (&rest args) &body spec) ...)
```

那么在宏里，参数spec将是一个含有读写器定义的列表。随后你可以通过ASSOC使用标签:reader和:writer来解出spec中的元素，再用DEFSTRUCTURING-BIND来取出每个元素的REST部分。<sup>①</sup>

从这里开始，剩下的问题只是将解出来的值插入到read-value和write-value方法的反引用模板中了。

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (with-gensyms (type)
    `(progn
      ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
        `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
```

<sup>①</sup> 使用ASSOC来解出spec的:reader和:WRITER元素，可以使define-binary-type的用户以任何顺序包含这些元素。如果你要求:reader元素必须总是第一个，那么你可以使用(rest (first spec))来解出读取器，再用(rest (second spec))来解出写入器。不过，只要你要求使用:reader和:writer关键字来改进define-binary-type形式的可读性，那么你就总是可以使用它们来解出正确的数据来。

```

        ,@body))
  ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
    `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
      ,@body)))))

```

注意反引用模板是如何嵌套的：最外层的模板以反引用的PROGN形式开始。这个模板由符号PROGN和两个逗号解除反引用的DESTRUCTURING-BIND表达式组成。这样，外层模板是通过求值DESTRUCTURING-BIND表达式并插入得到的值来实现的。每一个DESTRUCTURING-BIND表达式又含有另外的反引用模板，它生成插入到外层模板的方法定义。

有了这个宏，之前给出的define-binary-type形式将展开成下面的代码：

```

(progn
  (defmethod read-value ((#:g1618 (eql 'iso-8859-1-string)) in &key length)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in)))))
      string))
  (defmethod write-value ((#:g1618 (eql 'iso-8859-1-string)) out string &key length)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))

```

当然，现在你已经让这个漂亮的宏可用来自定义二进制类型了，不过它似乎还是多做了一些事。目前，应该只需要一个小的改进，就可以让它在你开始使用这个库来处理诸如ID3标签这样的实际格式时，成为相当有用的工具了。

和其他二进制格式一样，ID3标签使用的许多基本类型都是同一个主题下的变体，例如一个、两个、三个和四个字节的无符号整数。你当然可以用define-binary-type来逐个定义每个类型，或者你也可以将读写n字节无符号整数的通用算法分离成助手函数。

但是假设你已经定义了一个二进制类型unsigned-integer，它接受一个:bytes参数来指定一次读写多少个字节。使用这个类型，你可以用(unsigned-integer :bytes 1)的一个类型说明符来指定一个表示单字节无符号整数的槽。但假如一个特定的二进制格式指定了许多这样类型的槽，那么如果可以将其定义成一个代表同样类型的新类型（比如说u1）就会很方便了。结果证明，改变define-binary-type来支持两个形式是很容易的，一个是由:reader和:writer对构成的长形式，另一个是用已有类型来定义新二进制类型的短形式。使用一个短形式的define-binary-type，你可以像下面这样定义u1：

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
```

它将展开成下面的代码：

```

(progn
  (defmethod read-value ((#:g161887 (eql 'u1)) #:g161888 &key)
    (read-value 'unsigned-integer #:g161888 :bytes 1))
  (defmethod write-value ((#:g161887 (eql 'u1)) #:g161888 #:g161889 &key)
    (write-value 'unsigned-integer #:g161888 #:g161889 :bytes 1)))

```

为了同时支持长短两种形式的define-binary-type调用，需要基于spec参数的值来做区分。如果spec是两项的，那么它将代表一个长形式的调用，其中的两项应当分别是:reader



和:writer规范，你可以像之前那样处理。另一方面，如果spec只有一项，那么这个唯一的项应当是一个类型说明符，需要有区别地进行处理。你可以使用ECASE在spec的LENGTH上做切换，并随后解析spec来生成可分别用于长短两种形式的适当的展开式。

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (ecase (length spec)
    (1
      (with-gensyms (type stream value)
        (destructuring-bind (derived-from &rest derived-args) (mklist (first spec))
          `(progn
              (defmethod read-value ((,type (eql ',name)) ,stream &key ,@args)
                (read-value ',derived-from ,stream ,@derived-args))
              (defmethod write-value ((,type (eql ',name)) ,stream ,value &key ,@args)
                (write-value ',derived-from ,stream ,value ,@derived-args)))))))
    (2
      (with-gensyms (type)
        `(progn
            ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
              `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
                  ,@body))
            ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
              `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
                  ,@body)))))))

```

## 24.13 当前对象栈

在下一章里，你将会用到的最后一点儿功能是在读取和写入过程中获得当前二进制对象的方式。在更一般的情况下，当你读写嵌套的复合对象时，能够获得当前正在读写的任何层面的对象将是非常有用的。多亏有了动态变量和:around方法，你可以仅用几行代码来添加这一增强特性。一开始，你应当定义一个用来保存当前正在读取或写入的对象栈的动态变量。

```
(defvar *in-progress-objects* nil)
```

然后，你可以在read-object和write-object上定义:around方法，从而将正在被读写的对象在调用**CALL-NEXT-METHOD**之前推送到该变量里。

```
(defmethod read-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))

(defmethod write-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))
```

注意，你是如何把\*in-progress-objects\*重绑定到一个头部带有新项的列表上而不是将其赋予新值的。以这种方式的话，在LET形式结束，**CALL-NEXT-METHOD**返回以后，\*in-progress-objects\*的旧值将被恢复，从而相当于把对象从栈上弹出了。

定义了这两个方法之后，你还可以写出两个用来获取当前进度栈中特定对象的便利的函数。函数current-binary-object将返回栈的头部，也就是read-object或write-object最近被调用的那个对象。另一个函数parent-of-type接受一个应当是某个二进制类的名字的参数并返回最近推入栈中的该类型的对象，它使用TYPEP函数来测试一个给定的对象是否为一个特定类型的实例。

```
(defun current-binary-object () (first *in-progress-objects))

(defun parent-of-type (type)
  (find-if #'(lambda (x) (typep x type)) *in-progress-objects))
```

这两个函数可以用于在read-object和write-object调用的动态上下文中被调用的任何代码中。下一章将介绍关于current-binary-object用法的一个例子。<sup>①</sup>

现在你终于有了用来装备ID3解析库的所有工具，因此你可以进入下一章来做这件事了。

<sup>①</sup> ID3格式并不需要parent-of-type函数，因为它是一个相对扁平的结构。该函数主要用于解析一个带有深层嵌套结构的格式时，其解析过程依赖于保存在更高层结构中的信息。例如，在Java类文件格式中，顶层类文件结构含有一个常量池，负责该类文件中其他子结构中用到的数值映射到解析这些子结构时所需的常量值上。如果你正在编写一个类文件解析器，那么可以在读写那些子结构的代码中使用parent-of-type来获得顶层类文件对象并从中得到那个常量池。

# 实践：ID3解析器

25

**有**了一个解析二进制数据的库以后，你就可以开始编写一些代码来读写实际的二进制格式了，首先是ID3标签。ID3标签用来在MP3音频文件中嵌入元数据。处理ID3标签将是对二进制数据处理库的一个好的测试，因为ID3格式是一个真实的文件格式——工程权衡和特定设计选择的混合体，不管怎么说确实可以满足需要。万一你不了解文件共享领域的革命也不要紧，下面是关于ID3标签是什么以及它们与MP3文件之间关系的简要介绍。

MP3，也称为MPEG Audio Layer 3，是一种用来保存压缩的音频数据的格式，由Fraunhofer IIS的研究者们所设计并由Moving Picture Experts Group标准化，后者是由国际标准化组织（ISO）和国际电工技术委员会（IEC）所组成的联合委员会。不过，MP3格式本身只定义了如何保存音频数据。只要你所有的MP3文件都被单一的应用程序所管理，能够将元数据外部保存并跟踪元数据所关联的文件，那么就不会有太大的问题。不过，当人们开始在Internet上通过诸如Napster这样的文件共享平台相互传递独立的MP3文件时，他们很快发现需要一种方式将元数据嵌入进MP3文件本身。

由于MP3标准已经定案并且相当数量的软件和硬件已经知道如何解析已有的MP3格式了，所以任何在MP3文件中嵌入信息的方法都必须对MP3解码器不可见。ID3应运而生。

最初的ID3格式由程序员Eric Kemp所发明，它由连接到一个MP3文件结尾处的128个字节所构成，大多数MP3软件都会忽略它。它包括四个30字符的字段，分别用于歌曲标题、专辑标题、艺术家名和一个评论，一个四字节的年份字段以及一个单字节的风格代码。Kemp提供了对于前80个风格代码的标准含义。Nullsoft公司，一个流行的MP3播放器Winamp的发明者，后来又向其中添加了另外60种风格。

这个格式易于解析但明显带有很多局限性。它没有办法编码长度超过30字符的名字。它受限于256种风格，并且风格代码必须被所有ID3敏感的软件的用户认可才行。起初甚至没有办法编码一个特定MP3文件的CD音轨号，直到另一个程序员Michael Mutschler提议将音轨号嵌入到评论字段中，用一个空字节使其与评论的其余部分隔开，以便已有的、倾向于读取每个文本字段第一个空字符之前内容的ID3软件将其忽略。Kemp的版本现在被称为ID3v1，而Mutschler的版本是ID3v1.1。

尽管有上述局限性，但版本1确实提供了一个对于元数据问题的部分解决方案，因此它们被许多MP3压制程序（它们必须将ID3标签放进MP3文件里）和MP3播放器（它们将解出ID3标签中

的信息并显示给用户) 所采纳。<sup>①</sup>

可是到了1998年, 这些限制已经令人难以忍受了, 于是一个由Martin Nilsson领导的新的小组开始了设计全新标签模式的工作, 其成果后来被称为ID3v2。ID3v2格式极其灵活, 允许包含多种多样的信息, 同时几乎没有长度限制。它还利用了MP3格式的特定细节, 从而允许将ID3v2标签放置在一个MP3文件的开始处。

不过, ID3v2标签在解析方面相比版本1标签来说是一项挑战。在本章里, 你将使用前一章的二进制数据解析库来开发可以读写ID3v2标签的代码。或者至少你将有一个合理的开始——ID3v1太简单了, 而从完全过分工程化的角度来看, ID3v2又太复杂了。实现其规范中的每一处细节, 尤其是当你想要支持已规范化的所有三个版本时, 将需要相当多的工作。不过, 你可以忽略掉规范中的许多很少被实际使用的特性。对于初学者来说, 目前你可以忽略掉整个版本2.4, 因为它尚未被广泛采纳, 并且相比版本2.3来说, 它基本上只是增加了更多不需要的灵活性。我将把注意力集中在版本2.2和2.3上, 因为它们都已被广泛使用并且互相之间的区别大到了足够让事情保持有趣的程度。

## 25.1 ID3v2 标签的结构

在开始写代码之前, 需要熟悉ID3v2标签的整体结构。标签以一个含有关于整个标签的信息的头部开始。这个头部的最初三个字节以ISO-8859-1字符集编码了字符串“ID3”, 它们是字节73、68和51。接下来的两个字节编码了代表当前标签所符合的ID3规范的主版本和修订号。它们的后面又跟了一个字节, 其单独的位被视为标志位。这意味着这些单独标志的含义依赖于规范的版本。一些标志可以影响整个标签其余部分的解析方式。所谓“主版本”实际上是用来记录规范的副版本, 而“修订号”则是规范的子副版本。这样, “主版本”字段对于一个遵守2.3.0规范的标签来说就是3。修订号字段总是零, 因为每一个新的ID3v2规范都在副版本号上跳跃, 子副版本始终零。正如你将会看到的, 这个保存在标签主版本字段上的值将对你解析标签其余部分的方式产生深远的影响。

标签头部的最后一个字段是一个整数, 它以四个字节进行编码但只用了每个字节的前七位, 给出了整个标签不包括头部在内的长度。在版本2.3标签里, 头部可能还跟有几个扩展头部字段, 否则, 标签数据的其余部分将被划分成多个帧。不同类型的帧保存不同类型的信息, 从诸如歌曲名这类简单的文本信息到嵌入的图像。每个帧以一个含有字符标识符和长度的头部开始。在版本2.3中, 帧头还含有总长两字节的标志位, 以及取决于某个标志位的一个可选的单字节代码, 它用来指示帧的其余部分是如何加密的。

帧是带有标记的数据结构的一个完美例子。为了知道如何解析一个帧的主体, 你需要读取它的头部并使用标识符来检测你正在读取的帧的类型。

<sup>①</sup> 所谓压制 (ripping) 是将一张音乐CD中的某支歌曲转化成你硬盘中的一个MP3文件的过程。近年来, 大多数压制软件也都可以自动地从诸如Gracenote (也就是Compact Disc Database [CDB]) 或FreeDB这些在线数据库中获取关于歌曲的信息, 然后再以ID3标签的形式嵌入到MP3文件中。

ID3标签头中没有包含关于一个标签中究竟有多少个帧的直接指示。标签头只告诉你标签有多大，但由于许多标签都是变长的，因此要找出标签中含有的帧的数量，唯一方法就是实际去读取这些帧数据。另外，由标签头所给出的大小可能会超过帧数据的实际字节数，帧后面可能跟有足够的数量的空字节用以将标签补齐到指定的大小。这个设计使得标签编辑器可以在修改标签时无需重写整个MP3文件。<sup>①</sup>

因此，你面对的主要问题是在读取ID3头部时。检测你正在读取的是版本2.2还是2.3的标签，然后读取帧数据，并在读取了标签长度范围内的所有标签或是遇到补白字节的时候停下来。

## 25.2 定义包

和目前你开发的其他库一样，你应该把在本章里编写的代码放进它自己的包里。这里需要引用第24章和第15章的二进制数据和路径名的库，并且你也希望导出那些构成该包公共API的函数名。下面的包定义做到了所有这些事：

```
(defpackage :com.gigamonkeys.id3v2
  (:use :common-lisp
        :com.gigamonkeys.binary-data
        :com.gigamonkeys.pathnames)
  (:export
   :read-id3
   :mp3-p
   :id3-p
   :album
   :composer
   :genre
   :encoding-program
   :artist
   :part-of-set
   :track
   :song
   :year
   :size
   :translated-genre))
```

和往常一样，你可以并且也应该将包名中的com.gigamonkeys部分改成你自己的域。

## 25.3 整数类型

首先，你可以定义读写几种ID3格式会用到的基本类型的二进制类型，包括不同长度的无符号整数以及四种字符串。

<sup>①</sup> 几乎所有的文件系统都具有覆盖一个文件中已有字节的能力，但也有少数文件系统允许在一个文件的开始或中间位置添加或删除数据而无需重写文件的其余部分。由于ID3标签通常存放在一个文件的开始处，为了重写一个ID3标签而不干扰文件的其余部分，你必须将旧标签替换成一个长度完全相同的新标签。通过在写入ID3标签时带有特定数量的补白，你就有机会更好地做到这点。如果新标签带有比最初标签更多的数据，你就可以使用较少的补白，而如果变得更短了就使用更多的补白。

ID3用到了编码在一到四个字节中的无符号整数。如果你第一次编写一个通用的unsigned-integer二进制类型，其中接受读取的字节数作为一个参数，那么你随后可以再用短形式的define-binary-type来定义特定的类型。通用的unsigned-integer类型如下所示：

```
(define-binary-type unsigned-integer (bytes)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* 8 (1- bytes)) to 0 by 8 do
        (setf (ldb (byte 8 low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* 8 (1- bytes)) to 0 by 8
      do (write-byte (ldb (byte 8 low-bit) value) out))))
```

现在，你可以使用短形式的define-binary-type像下面这样为ID3格式里用到的每种大小的整数分别定义一个类型：

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
(define-binary-type u2 () (unsigned-integer :bytes 2))
(define-binary-type u3 () (unsigned-integer :bytes 3))
(define-binary-type u4 () (unsigned-integer :bytes 4))
```

另一个你需要用来读写的类型是用在头部中的28位值。这个值使用28位而非诸如32位这样的8的倍数来编码，因为一个ID3标签中不能含有在字节#xff后跟一个前三位为1的字节的模式，这对于MP3解码器来说有另外的特殊含义。ID3头部的其他字段也都不允许含有这样的字节序列，但如果你将标签大小编码成一个正规的unsigned-integer的话，就有可能会出问题了。为了避免这种可能性，这个大小被编码成只使用每个字节的底下7位，并让最上面一位总是零。<sup>①</sup>

这样，除了你传给LDB的字节说明符的大小应当是7而不是8之外，它可以像unsigned-integer那样进行读写。这种相似性表明，假如你为已有的unsigned-integer二进制类型添加一个参数bits-per-byte，那么你就可以用短形式的define-binary-type直接定义出一个新类型id3-tag-size。除了bits-per-byte被用在旧版本的所有硬编码了数字8的位置上之外，这个新版本的unsigned-integer和旧版本非常像。如下所示：

```
(define-binary-type unsigned-integer (bytes bits-per-byte)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte do
        (setf (ldb (byte bits-per-byte low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte
      do (write-byte (ldb (byte bits-per-byte low-bit) value) out))))
```

那么id3-tag-size的定义就很简单了。

```
(define-binary-type id3-tag-size () (unsigned-integer :bytes 4 :bits-per-byte 7))
```

<sup>①</sup> ID3头部后跟的帧数据也可能潜在地含有这一不合法的序列。可以使用一种不同的模式来避免其出现，即通过打开标签头上的某个标记位来控制。本章中的代码并不考虑该标记位被设定的可能性，它在实际上也很少被用到。

你还需要改变u1到u4的定义，像下面这样明确指定每个字节里读取8位：

```
(define-binary-type u1 () (unsigned-integer :bytes 1 :bits-per-byte 8))
(define-binary-type u2 () (unsigned-integer :bytes 2 :bits-per-byte 8))
(define-binary-type u3 () (unsigned-integer :bytes 3 :bits-per-byte 8))
(define-binary-type u4 () (unsigned-integer :bytes 4 :bits-per-byte 8))
```

## 25.4 字符串类型

ID3格式中另一个常用的基本类型是字符串。前一章讨论了处理二进制文件中的字符串时必须考虑的一些问题，例如字符编码和字符编码方式之间的区别。

ID3使用两种不同的字符编码ISO 8859-1和Unicode。ISO 8859-1也称为Latin-1，是一种八位字符编码，它用西欧语言中用到的字符扩展了ASCII。换句话说，在ASCII和ISO 8859-1中，从0到127之间的代码点映射到相同的字符上，但ISO 8859-1还提供了最大到255的其余代码点的映射。Unicode是设计用于为世界上所有语言的几乎每一个字符提供代码点的字符编码。Unicode是ISO 8859-1的超集，正如ISO 8859-1是ASCII的超集那样。从0到255的代码点在ISO 8859-1和Unicode中都映射到相同的字符上。（因此，Unicode也是ASCII的一个超集。）

由于ISO 8859-1是一个8位字符编码，它使用每字符一个字节的方式进行编码。对于Unicode字符串来说，ID3使用带有前导字符序标记的UCS-2编码方式。<sup>①</sup>我将很快讨论什么是一个字节序标记。

读写这两种编码方式不是问题，不过是以不同的格式读写无符号整数罢了，而你已经写好了做这件事的代码。难点在于如何将这些数值转化成Lisp字符对象。

你所使用的Lisp实现很可能使用了Unicode或ISO 8859-1作为其内部字符编码。而由于从0到255之间的所有值在ISO 8859-1和Unicode中都映射到相同的字符上，所以你可以使用Lisp的CODE-CHAR和CHAR-CODE函数来转化两个编码中的这些值。不过，如果你的Lisp仅支持ISO 8859-1，那么你只能把前255个Unicode字符表示成Lisp字符。换句话说，在这样的Lisp实现里，如果你试图处理一个用到Unicode字符串并且其中含有代码点超出255的字符的ID3标签，那么你在试图把代码点转化成一个Lisp字符时会遇到错误。目前，先假设你正在使用一个基于Unicode的Lisp或者你不会处理任何含有超出ISO 8859-1范围的字符的文件。

字符串编码所带来的另一个问题，是如何得知需要将多少个字节解释成字符数据。ID3使用了前一章提到的两种策略——一些字符串是采用空字符结尾的，而另一些字符串出现在你可以决定读取多少个字节的位置上，要么是因为那个位置上的字符串总是具有相同的长度，要么是因为字符串处在一个总长度已知的符合结构的结尾处。不过需要注意的是，字节的数量不一定与字符串中字符的数量相同。

考虑了所有这些特征，ID3格式使用四种方式来读写字符串——由两种字符编码方式和两种字符串数据分界方式排列组合而成。

<sup>①</sup> 在ID3v2.4中，UCS-2被替换成几乎等价的UTF-16，并且UTF-16BE和UTF-8被增加为附加的编码方式。

很明显，读写字符串的很多业务逻辑将会非常相似。因此，你可以从定义两种二进制类型开始，一种用于读取指定（字符）长度的字符串，而另一种用来读取带有终止符的字符串。这两种类型利用的read-value和write-value的类型参数是由另外的代码提供的，你可以让字符类型来读取一个关于其类型的参数。这种技术你在本章里还会多次看到。

```
(define-binary-type generic-string (length character-type)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (read-value character-type in)))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-value character-type out (char string i)))))

(define-binary-type generic-terminated-string (terminator character-type)
  (:reader (in)
    (with-output-to-string (s)
      (loop for char = (read-value character-type in)
            until (char= char terminator) do (write-char char s))))
  (:writer (out string)
    (loop for char across string
          do (write-value character-type out char)
          finally (write-value character-type out terminator))))
```

有了这些类型，读取ISO 8859-1字符串就很容易了。由于传递给generic-string的read-value和write-value方法的character-type参数必须是二进制类型的名字，因此你需要定义一个iso-8859-1-char二进制类型。这也给了你一个很好的机会用来放置一些用于一致性检查的代码，检查你所读写字符的代码点。

```
(define-binary-type iso-8859-1-char ()
  (:reader (in)
    (let ((code (read-byte in)))
      (or (code-char code)
          (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (if (<= 0 code #xff)
          (write-byte code out)
          (error
            "Illegal character for iso-8859-1 encoding: character: ~c with code: ~d"
            char code)))))
```

现在，使用define-binary-type的短形式来定义ISO 8859-1字符串类型就很简单了，如下所示：

```
(define-binary-type iso-8859-1-string (length)
  (generic-string :length length :character-type 'iso-8859-1-char))

(define-binary-type iso-8859-1-terminated-string (terminator)
  (generic-terminated-string :terminator terminator
                            :character-type 'iso-8859-1-char))
```

读取UCS-2字符串只是稍微复杂一些。其复杂性源自你可以用两种方式来编码一个UCS-2代码点：大端字节序（big-endian）或小端字节序（little-endian）。因此UCS-2字符串以两个附加的字节开始，称为字节序标记（byte order mark），它由以big-endian或little-endian形式编码的数值#xffff构成。当读取一个UCS-2字符串时，你需要读取这个字节序标记，然后根据其值来读取big-endian或little-endian的字符。这样，你将需要两个不同的UCS-2字符类型。但是你只需要一个版本的一致性检查代码，因此你可以像下面这样来定义一个参数化的二进制类型：

```
(define-binary-type ucs-2-char (swap)
  (:reader (in)
    (let ((code (read-value 'u2 in)))
      (when swap (setf code (swap-bytes code)))
      (or (code-char code) (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (unless (<= 0 code #xffff)
        (error "Illegal character for ucs-2 encoding: ~c with char-code: ~d"
               char code))
      (when swap (setf code (swap-bytes code)))
      (write-value 'u2 out code))))
```

其中的swap-bytes函数可以像下面这样来定义，它利用了LDB函数可被SETF和ROTATEF的特点：

```
(defun swap-bytes (code)
  (assert (<= code #xffff))
  (rotatef (ldb (byte 8 0) code) (ldb (byte 8 8) code))
  code)
```

使用ucs-2-char，你可以定义两个用作通用字符串函数的character-type参数的字符类型。

```
(define-binary-type ucs-2-char-big-endian () (ucs-2-char :swap nil))

(define-binary-type ucs-2-char-little-endian () (ucs-2-char :swap t))
```

然后你需要一个函数，它基于字节序标记的值来返回具体所使用的字符类型。

```
(defun ucs-2-char-type (byte-order-mark)
  (ecase byte-order-mark
    (#xffff 'ucs-2-char-big-endian)
    (#xfffe 'ucs-2-char-little-endian)))
```

现在，你可以定义UCS-2编码字符串的长度和终止符定界的字符串类型了。它们将读取字节序标记，并用这个标记来决定究竟向read-value和write-value的character-type参数传递哪个UCS-2字符变体。其余唯一的亮点是，你需要根据字节序标记将代表字节个数的length参数转化成需要读取的字符数。

```
(define-binary-type ucs-2-string (length)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in))
          (characters (1- (/ length 2)))))
      (read-value
       'generic-string in
       :length characters
       :character-type (ucs-2-char-type byte-order-mark))))
```

```
(:writer (out string)
  (write-value 'u2 out #xeff)
  (write-value
    'generic-string out string
    :length (length string)
    :character-type (ucs-2-char-type #xeff)))))

(define-binary-type ucs-2-terminated-string (terminator)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in)))
      (read-value
        'generic-terminated-string in
        :terminator terminator
        :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xeff)
    (write-value
      'generic-terminated-string out string
      :terminator terminator
      :character-type (ucs-2-char-type #xeff))))
```

## 25.5 ID3 标签头

基本类型定义完成以后，就可以切换到更高层次的视角并开始定义二进制类来表示ID3标签整体和单独的帧了。

如果你是首次接触ID3v2.2规范，那么你将看到标签的基本结构是如下所示的头部：

25

ID3/file identifier	"ID3"
ID3 version	\$02 00
ID3 flags	%xx000000
ID3 size	4 * %0xxxxxxxx

其后跟帧数据和补白。由于你已经定义了读写头部所有字段的二进制类型，定义读取一个ID3标签的整个头部的类只是将已有的成果合并在一起罢了。

```
(define-binary-class id3-tag ()
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)))
```

如果你手头有一些MP3文件的话，那么你可以测试目前的这些代码，同时也看看你的MP3都含有哪些版本的ID3标签。首先你编写一个函数，从一个文件的开始处读取刚刚定义的这个id3-tag。不过，请注意ID3标签不一定出现在一个文件的开始处，尽管目前它们总是这样的。为了在一个文件的其他位置上找到ID3标签，你可以在文件中搜索字节序列73、68、51（也即字符串“ID3”）。<sup>①</sup>目前你可以简单地假设这些标签总是出现在文件的开始处。

<sup>①</sup> ID3格式的2.4版也支持在一个标签的结尾处放置一个脚标，这使得一个附加在文件结尾处的标签可以更容易地被找到。

```
(defun read-id3 (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (read-value 'id3-tag in)))
```

在这个函数的基础上可以构造一个函数，它接受一个文件名并打印出连同文件名在内的标签中的信息。

```
(defun show-tag-header (file)
  (with-slots (identifier major-version revision flags size) (read-id3 file)
    (format t "~-a ~d.~d ~8,'0b ~d bytes -- ~a~%"
            identifier major-version revision flags size (enough-namestring file))))
```

它可以打印出类似下面这样的输出：

```
ID3V2> (show-tag-header "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
ID3 2.0 00000000 2165 bytes -- Kitka/Wintersongs/02 Byla Cesta.mp3
NIL
```

当然，为了检测你的MP3库里哪个版本的ID3是最普遍的，如果有一个函数能返回一个给定目录下所有MP3文件的汇总将是更有用的。你可以用第15章里我们定义的walk-directory函数轻松地写出这个函数。首先定义一个助手函数来测试一个给定的文件名是否带有mp3扩展名。

```
(defun mp3-p (file)
  (and
    (not (directory-pathname-p file))
    (string-equal "mp3" (pathname-type file))))
```

然后你可以将show-tag-header、mp3-p和walk-directory组合起来，打印出给定目录下每个MP3文件的ID3头的汇总。

```
(defun show-tag-headers (dir)
  (walk-directory dir #'show-tag-header :test #'mp3-p))
```

不过，如果你有许多MP3文件，你可能只想知道你的MP3收藏中每个版本的ID3标签分别有多少个。为了得到这个信息，可以写一个像下面这样的函数：

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
              (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
           (walk-directory dir #'count-version :test #'mp3-p)))
      versions))
```

另一个你将在第29章里用到的函数是用来测试给定文件是否以一个ID3标签开始的函数，可以像下面这样来定义它：

```
(defun id3-p (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (string= "ID3" (read-value 'iso-8859-1-string in :length 3))))
```

## 25.6 ID3帧

如同之前所讨论的，一个ID3标签从整体上被划分成了多个帧，每个帧都具有类似于整个标

签的内部结构，都以一个指示了该帧类型和字节长度的头开始。帧头的结构在ID3格式的版本2.2和版本2.3之间稍微有些变化，而最终还要同时处理两种形式。刚开始，你可以集中在解析版本2.2的帧上。

一个版本2.2的帧头由三个编码一个三字符ISO 8859-1字符串的字节和一个三字节的无符号整数所构成，后者指定了该帧的字节长度，其中不包括6字节的头部。字符串表明该帧的类型，从而决定了你解析帧长度后面其他数据的方式。这正好是你定义的`define-tagged-binary-class`宏所适用的场合。你可以定义一个带有标签的类来读取帧头，并随后使用一个从ID映射到类名的函数派发到适当的具体类上。

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

现在你可以开始实现具体的帧类了。不过，规范里定义了许多帧类——版本2.2中共有63个，后续版本里还有更多。即便将那些共享了同样的基本结构的帧类型视为等价的，最后你仍然可以在版本2.2中得到24种不同的帧类型，但是它们中只有很少的一些是被“广泛使用的”。因此，与其立即开始为每个帧类型定义类，还不如从编写一个通用帧类开始，这个类可以让你读取一个标签中的帧而无需解析帧里面的数据。这将给你一种方式来找出你想要处理的MP3文件中实际上都有哪些帧。反正你最终也需要这样一个类，因为规范中允许实验性帧的存在，对于这些帧你可以无需解析地读取它们。

由于帧头中的大小字段可以告诉你一个帧究竟有多少个字节，因此你可以定义一个`generic-frame`类，它扩展`id3-frame`并增加了一个字段`data`用来保存一个字节数组。25

```
(define-binary-class generic-frame (id3-frame)
  ((data (raw-bytes :size size))))
```

其中数据字段的类型`raw-bytes`只用来保存一个字节数组。你可以像下面这样来定义它：

```
(define-binary-type raw-bytes (size)
  (:reader (in)
    (let ((buf (make-array size :element-type '(unsigned-byte 8))))
      (read-sequence buf in)
      buf))
  (:writer (out buf)
    (write-sequence buf out)))
```

现阶段，你希望所有的帧都被读取为`generic-frames`，这样你可以定义用在`id3-frame`的`:dispatch`表达式中的`find-frame-class`函数，让它总是返回`generic-frame`，无论帧的`id`是什么。

```
(defun find-frame-class (id)
  (declare (ignore id))
  'generic-frame)
```

现在你需要修改`id3-tag`，让其可以读取头部字段后面的那些帧。读取帧数据的唯一难点是：尽管标签头告诉了你该标签有多少字节，但这个数值还包括了跟在帧数据之后的补白。标签头无

法告诉你该标签含有多少帧，因此知道你遇到补白的唯一办法，就是在你期待一个帧标识符的时候却找到了一个空字节。

为此，你可以定义一个二进制类型 `id3-frames`，它负责读取一个标签的其余部分，创建代表它所发现的所有帧的对象；并且跳过任何补白。这个类型接受标签大小作为一个参数，该参数可用来避免读取到超过标签结尾的位置上。但是读取代码也将需要检测跟在帧数据之后的补白的开始位置。因此不能直接在 `id3-frames` 的 `:reader` 部分直接调用 `read-value`，你应当使用一个函数 `read-frame`，你将定义它在检测到补白时返回 `NIL`，而在其他时候返回一个使用 `read-value` 来读取到的 `id3-frame` 对象。假设你已定义了 `read-frame`，并让它在前一个帧的结尾处读取额外的一个字节来检测补白的开始，那么你可以像下面这样来定义 `id3-frame` 二进制类型：

```
(define-binary-type id3-frames (tag-size)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame in)
          while frame
          do (decf to-read (+ 6 (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
          for frame in frames
          do (write-value 'id3-frame out frame)
          (decf to-write (+ 6 (size frame)))
          finally (loop repeat to-write do (write-byte 0 out))))
```

你可以使用这个类型来为 `id3-tag` 增加一个帧槽。

```
(define-binary-class id3-tag ()
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size))))
```

## 25.7 检测标签补白

现在剩下的就只是实现 `read-frame` 了。这有一点儿麻烦，因为实际从流中读取字节的代码位于 `read-frame` 的数层以下。

你真正想要在 `read-frame` 中做的是读取一个字节并在它为空时返回 `NIL`，否则使用 `read-value` 来读取一个帧。不幸的是，如果你在 `read-frame` 中读取了这个字节，那么在被 `read-value` 读取时它就不再可用用了。<sup>①</sup>

<sup>①</sup> 字符流支持两个函数，`PEEK-CHAR` 和 `UNREAD-CHAR`，这两个函数中的任何一个都是对于该问题的完美解决方案，但是二进制流不支持任何等价的函数。

看起来这是一个使用状况系统的好机会。你可以在从流中进行读取的底层代码中检查空字节，并在你读到一个空字节时抛出一个状况。`read-frame`随后可以处理该状况并在读取更多字节以前将栈回退。这个方法不但是一个检测标签的补白开始位置的良好解决方案，还是一个将状况系统用于错误处理之外目的的例子。

可以从定义一个状况类型开始，它将从底层代码接收信号并被上层代码处理。这个状况并不需要任何槽，你只需要一个可区分的状况类来确保没有其他的代码可能抛出或处理它即可。

```
(define-condition in-padding () ())
```

接下来需要定义一个二进制类型，其`:reader`部分读取指定数量的字节，它先读一个字节并在该字节为空时抛出一个`in-padding`状况，否则继续按照`iso-8859-1-string`来读取其余的字节，并将得到的结果与前面读取的第一个字节组合起来。

```
(define-binary-type frame-id (length)
  (:reader (in)
    (let ((first-byte (read-byte in)))
      (when (= first-byte 0) (signal 'in-padding))
      (let ((rest (read-value 'iso-8859-1-string in :length (1- length))))
        (concatenate
          'string (string (code-char first-byte)) rest))))
  (:writer (out id)
    (write-value 'iso-8859-1-string out id :length length)))
```

如果你重定义了`id3-frame`，使其`id`槽的类型从`iso-8859-1-string`变成`frame-id`，那么每当`id3-frame`的`read-value`方法读到一个空字节而非帧的开始处时，状况就会被抛出。

```
(define-tagged-binary-class id3-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

现在`read-frame`需要做的只是将对`read-value`的调用包装在`HANDLER-CASE`中，`HANDLER-CASE`处理`in-padding`状况并返回`NIL`。

```
(defun read-frame (in)
  (handler-case (read-value 'id3-frame in)
    (in-padding () nil)))
```

定义了`read-frame`之后，你就可以读取一个完整的版本2.2的ID3标签了，其中的帧用`generic-frame`的实例来表示。在25.11节里，你将在REPL中做一些实验来检测需要实现的帧类。但首先让我们添加对版本2.3的ID3标签的支持。

## 25.8 支持 ID3 的多个版本

目前，`id3-tag`是用`define-binary-class`定义的，但如果你想要支持多个版本的ID3，那么使用一个`define-tagged-binary-class`将更加合理，因为它可以派发`major-version`的值。看起来所有版本的ID3v2都具有相同的结构，包括大小字段。因此你可以像下面这样来定义一个带有标签的二进制类，其定义了基本的结构并派发到适当版本相关的子类上。

```
(define-tagged-binary-class id3-tag ()
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size))
  (:dispatch
   (ecase major-version
     (2 'id3v2.2-tag)
     (3 'id3v2.3-tag))))
```

版本2.2和2.3的标签在两方面上有所区别。首先，一个版本2.3标签的头部可能被至多四个可选的扩展头部字段所扩展，这可以通过flags字段的值来检测到。其次，帧格式在版本2.2和版本2.3之间发生了变化，这意味着你将使用不同的类来表示版本2.2的帧和对应的版本2.3的帧。

由于新的id3-tag类以最初为了表示版本2.2标签所写的那个为基础，所以新的id3v2.2-tag类的定义比较简单也就不奇怪了，它继承了来自新的id3-tag类的大部分槽并添加了一个缺失的槽frames。由于版本2.2和2.3使用了不同的帧格式，所以你必须将id3-frames类型改成根据所读取的帧类型进行参数化选择的形式。目前，假设你将通过为id3-frames类型描述符添加一个:frame-type参数来做到这点，如下所示：

```
(define-binary-class id3v2.2-tag (id3-tag)
  ((frames (id3-frames :tag-size size :frame-type 'id3v2.2-frame))))
```

id3v2.3类带有可选的字段，因此会稍微复杂一些。4个可选字段中的前3个将在flag的第6位被设置时包含在标签中。它们包括一个用来指定扩展头大小的四字节的整数，两个字节的标志位，以及另一个用来指定标签中含有多少个字节补白的四字节的整数。<sup>①</sup>第4个可选字段，当第15个扩展的头标志位被设置时会被包含进来，它是标签其余部分的一个四字节的循环冗余校验（CRC）。

二进制数据处理库没有提供对于二进制类中的可选字段的任何特别的支持，但是看起来正规的参数化的二进制类型就足够好了。你可以使用一个类型的名字和一个代表是否实际读写该类型的变量来参数化地定义这个新类型。

```
(define-binary-type optional (type if)
  (:reader (in)
    (when if (read-value type in)))
  (:writer (out value)
    (when if (write-value type out value))))
```

使用if作为参数的名字可能看起来有些奇怪，但它使得这个optional类型描述符变得更加容易理解了。举个例子，下面是使用了optional槽的id3v2.3-tag的定义：

```
(define-binary-class id3v2.3-tag (id3-tag)
  ((extended-header-size (optional :type 'u4 :if (extended-p flags)))
   (extra-flags          (optional :type 'u2 :if (extended-p flags)))
   (padding-size         (optional :type 'u4 :if (extended-p flags)))
   (crc                  (optional :type 'u4 :if (crc-p flags extra-flags)))
   (frames               (id3-frames :tag-size size :frame-type 'id3v2.3-frame))))
```

<sup>①</sup> 如果一个标签带有扩展的头部，那么可以用这个值来检测帧数据的结束位置。不过，如果这个扩展的头部没有使用，那么就继续使用老方法，不值得添加新代码以不同的方式来做这件事。

其中extended-p和crc-p是用来测试特定标志位是否被传递的助手函数。为了测试一个整数中的个别位是否被设置了，你可以使用另一个位操作函数LOGBITP。它接受一个索引和一个整数，并在该整数中的指定位被设置时返回真。

```
(defun extended-p (flags) (logbitp 6 flags))

(defun crc-p (flags extra-flags)
  (and (extended-p flags) (logbitp 15 extra-flags)))
```

和版本2.2的标签类一样，帧槽被定义为类型id3-frames，其中帧类型的名字作为参数传递。尽管如此，你需要对id3-frames和read-frame做一些小的改动以使其支持额外的frame-type参数。

```
(define-binary-type id3-frames (tag-size frame-type)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame frame-type in)
          while frame
          do (decf to-read (+ (frame-header-size frame) (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
          for frame in frames
          do (write-value frame-type out frame)
          (decf to-write (+ (frame-header-size frame) (size frame)))
          finally (loop repeat to-write do (write-byte 0 out)))))

(defun read-frame (frame-type in)
  (handler-case (read-value frame-type in)
    (in-padding () nil)))
```

25

改动发生在对read-frame和write-value的调用中，这里你需要传递frame-type参数，并且在计算帧大小的时候，需要使用一个函数frame-header-size来代替字面数值6，因为帧头的大小在版本2.2和2.3之间发生了改变。由于该函数在结果上的区别取决于帧的类，所以像下面这样将其定义成一个广义函数是合理的：

```
(defgeneric frame-header-size (frame))
```

下一节，在定义了新的帧类以后，你将在该广义函数上定义必要的方法。

## 25.9 版本化的帧基础类

之前你定义了单一的基础类用于所有的帧类型，现在，你需要两个类id3v2.2-frame和id3v2.3-frame。其中id3v2.2-frame类和最初的id3-frame类完全相同。

```
(define-tagged-binary-class id3v2.2-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

另一方面，`id3v2.3-frame`需要更多的修改。帧标识符和大小字段在版本2.3里各自从3个字节增加到4个字节，而另有两个字节的标志位被添加进来。另外，和版本2.3的标签一样，帧可以含有可选字段，具体由三个帧标志位的值来控制。<sup>①</sup>考虑到这些变化，可以像下面这样定义版本2.3的帧基础类以及相关的助手函数：

```
(define-tagged-binary-class id3v2.3-frame ()
  ((id           (frame-id :length 4))
   (size         u4)
   (flags        u2)
   (decompressed-size (optional :type 'u4 :if (frame-compressed-p flags)))
   (encryption-scheme (optional :type 'ul :if (frame-encrypted-p flags)))
   (grouping-identity (optional :type 'ul :if (frame-grouped-p flags))))
  (:dispatch (find-frame-class id)))

(defun frame-compressed-p (flags) (logbitp 7 flags))

(defun frame-encrypted-p (flags) (logbitp 6 flags))

(defun frame-grouped-p (flags) (logbitp 5 flags))
```

有了这两个函数，现在你可以实现广义函数`frame-header-size`上的方法了。

```
(defmethod frame-header-size ((frame id3v2.2-frame)) 6)

(defmethod frame-header-size ((frame id3v2.3-frame)) 10)
```

版本2.3帧中的可选字段并不在这些计算中作为帧头的一部分而计入，因为它们已经被包括在帧的`size`值中了。

## 25.10 版本化的具体帧类

在最初的定义中，`id3-frame`是`generic-frame`的子类。但现在`id3-frame`已经被替换成了两个版本相关的基础类`id3v2.2-frame`和`id3v2.3-frame`。所以，你需要定义两个新版本的`generic-frame`，为每个基础类定义一个。定义这些类的一种方法如下所示：

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame)
  ((data (raw-bytes :size size)))

(define-binary-class generic-frame-v2.3 (id3v2.3-frame)
  ((data (raw-bytes :size size))))
```

不过，这里面不太好的一点是这两个类除了基类以外其余部分都相同。在本例中，由于它们只有唯一的附加字段，所以看起来还不算太坏。但如果你将这种思路用在其他具体的帧类上，尤

<sup>①</sup> 这些标志位，除了控制是否包含可选字段以外，还可以影响标签中其余部分的解析方式。特别地，如果第七个标志位被设定，那么实际的帧数据将使用zlib压缩算法进行压缩。而如果第六个标志位被设定，那么数据将被加密。在实践中这些选项很少出现，但如果真的出现的话，目前只能忽略它们。不过如果你打算实现一个产品级的ID3库，那么就不得不涉及到这些领域了。一个简单的不完整解决方案是改变`find-frame-class`来接受第二个参数并向其传递所有标志位。如果帧被压缩或加密了，那么你应当实例化一个通用帧来保存数据。

其是那些带有更复杂的内部结构但在两个ID3版本上却又完全相同的帧类，那么这些重复定义将浪费很多时间。

另一种你实际应当采用的思路是，将generic-frame类定义为一个合成类（mixin）：一个用来作为基类的类，它和一个版本相关的基类可以共同使用来产生一个具体的版本相关的帧类。这种思路唯一的难点是，如果generic-frame没有扩展任何一个帧基础类的话，那么你就无法在其定义中访问size槽。因此，你必须使用前一章结尾处讨论的current-binary-object函数来访问你正在读写的对象并将其传递给size。并且需要考虑整个帧的大小在字节数上的区别，尤其当版本2.3的帧里含有任何可选字段时。因此，你需要定义一个广义函数data-bytes以及相应的在版本2.2和2.3的帧下都可以正确工作的方法。

```
(define-binary-class generic-frame ()
  ((data (raw-bytes :size (data-bytes (current-binary-object))))))

(defgeneric data-bytes (frame))

(defmethod data-bytes ((frame id3v2.2-frame))
  (size frame))

(defmethod data-bytes ((frame id3v2.3-frame))
  (let ((flags (flags frame)))
    (- (size frame)
        (if (frame-compressed-p flags) 4 0)
        (if (frame-encrypted-p flags) 1 0)
        (if (frame-grouped-p flags) 1 0))))
```

然后你可以扩展版本相关的基础类和generic-frame类，来定义出版本相关的通用帧类。

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame generic-frame) ())
(define-binary-class generic-frame-v2.3 (id3v2.3-frame generic-frame) ())
```

有了这些类的定义，现在你可以重定义find-frame-class函数，根据标识符的长度来返回正确的版本化的类。

```
(defun find-frame-class (id)
  (ecase (length id)
    (3 'generic-frame-v2.2)
    (4 'generic-frame-v2.3)))
```

## 25.11 你实际需要哪些帧

有了使用通用帧来同时读取版本2.2和2.3标签的能力，就可以开始实现那些代表你所关心的特定帧的类了。不过，在就此深入下去之前，应该先停下来思考一下究竟哪些帧是你所关心的，因为正如我之前所提到的，ID3标签规范中指定了许多几乎从不使用的帧。当然，你所关心的帧取决于你正在编写哪种类型的应用。如果你最关心的是从已有的ID3标签中解出信息，那么你只需实现那些含有你所关心的信息的帧所对应的类即可。另一方面，如果你打算编写一个ID3标签编辑器，那么你可能需要实现所有的帧。

与其猜测哪些帧是最有用的，还不如使用已有的代码在REPL中实际测试一下，找出在你的MP3文件中实际用到了哪些帧。你需要从一个id3-tag的实例开始，可以通过read-id3函数得到它。

```
ID3V2> (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
#<ID3V2.2-TAG @ #x727b2912>
```

由于可能会多次用到这个对象，所以最好把它保存在一个变量里。

```
ID3V2> (defparameter *id3*
          (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
```

现在你可以看到它有多少个帧。

```
ID3V2> (length (frames *id3*))
11
```

看起来并不是很多——让我们具体看看它们是什么。

```
ID3V2> (frames *id3*)
(#<GENERIC-FRAME-V2.2 @ #x72dabdda> #<GENERIC-FRAME-V2.2 @ #x72dabec2>
 #<GENERIC-FRAME-V2.2 @ #x72dabfa2> #<GENERIC-FRAME-V2.2 @ #x72dac08a>
 #<GENERIC-FRAME-V2.2 @ #x72dac16a> #<GENERIC-FRAME-V2.2 @ #x72dac24a>
 #<GENERIC-FRAME-V2.2 @ #x72dac32a> #<GENERIC-FRAME-V2.2 @ #x72dac40a>
 #<GENERIC-FRAME-V2.2 @ #x72dac4f2> #<GENERIC-FRAME-V2.2 @ #x72dac632>
 #<GENERIC-FRAME-V2.2 @ #x72dac7b2>)
```

好吧，几乎看不到什么有用的信息。你其实更想知道这些帧都是什么类型的。换句话说，你想知道这些帧的ID，这可以通过下面这样一个简单的MAPCAR来实现：

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

如果你在ID3v2.2规范中查找这些标识符，那么你将发现所有那些带有字母T开头标识符的帧都是文本信息帧并且都具有相似的结构。而COM是评论帧的标识符，其结构也跟文本信息帧相似。这里所辨认出的一些特定的文本信息帧其实是用来表示歌曲标题、艺术家、专辑、音轨、歌曲集的部分、年份、风格、以及编码程序的帧。

当然，这只是一个MP3文件。其他文件也许还用到了其他的帧。全部找出它们并不难。首先定义一个函数将前面的MAPCAR表达式和一个对read-id3的调用组合起来，再将所有这些封装在一个DELETE-DUPLICATES中以保证结果的简洁性。你应当在DELETE-DUPLICATES中使用一个值为#':string='的:test参数，当两个元素是相同的字符串时，把它们视为是等价的。

```
(defun frame-types (file)
  (delete-duplicates (mapcar #'id (frames (read-id3 file))) :test #'string=))
```

这应该得到和之前一样的结果，只不过当该函数使用相同的文件名时每个标识符只有一个。

```
ID3V2> (frame-types "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

然后你可以使用第15章里的walk-directory函数与mp3-p一起来找出一个目录下的每个MP3文件，并将在这些文件上调用frame-types得到的结果组合在一起。我们知道NUNION是

**UNION**函数的回收性版本。由于frame-types对于每个文件都会建立新的列表，所以使用它是安全的。

```
(defun frame-types-in-dir (dir)
  (let ((ids ()))
    (flet ((collect (file)
              (setf ids (nunion ids (frame-types file) :test #'string=)))
           (walk-directory dir #'collect :test #'mp3-p))
      (ids)))
```

现在传给它一个目录的名字，然后它将告诉你该目录下的所有MP3文件总计用到的标识符的集合。根据你的MP3文件多少，该函数可能用掉几秒的时间，但最后你将很可能得到类似下面这样的东西：

```
ID3V2> (frame-types-in-dir "/usr2/mp3/")
("TCON" "COMM" "TRCK" "TIT2" "TPE1" "TALB" "TCP" "TT2" "TP1" "TCM"
 "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

其中的四字母标识符是版本2.2标识符在版本2.3中的等价物。由于保存在这些帧中的信息正是你将在第27章里所需要的，只为实际用到的帧（即下面两节将要讨论的文本信息帧和评论帧）实现具体的类是比较合理的。如果你决定以后还要支持其他的帧类型，那么无非就是将其ID3规范转化成适当的二进制类定义了。

## 25.12 文本信息帧

所有的文本信息帧都由两个字段组成：一个用来指示该帧所采用的字符串编码方式的单字节，以及一个编码在帧的其余字节中的字符串。如果编码方式字节为0，那么字符串将用ISO 8859-1来编码；如果该字节为1，那么字符串将是一个UCS-2字符串。25

你已经定义了代表四种不同类型字符串的二进制类型——两种不同的编码方式，其中每个分别使用不同的字符串定界方法。尽管如此，`define-binary-class`并不直接支持基于对象中的其他值来检测所要读取的值类型。相反，你可以定义一个二进制类型，它接受你传递的编码方式字节的值并读写对应类型的字符串。

定义了这样一个类型之后，你也可以定义它接受两个参数：`:length`和`:terminator`，并通过具体指定的参数来选择正确的字符串类型。为了实现这个新类型，必须首先定义一些助手函数。前两个函数可以根据编码方式字节来返回对应的字符串类型的名字。

```
(defun non-terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-string)
    (1 'ucs-2-string)))

(defun terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-terminated-string)
    (1 'ucs-2-terminated-string)))
```

然后`string-args`函数使用编码方式字节、长度和终止符来决定在`id3-encoded-string`



的：`:reader`和`:writer`中传递给`read-value`和`write-value`的参数。`string-args`的`length`和`terminator`参数中应当总有一个是`NIL`。

```
(defun string-args (encoding length terminator)
  (cond
    (length
      (values (non-terminated-type encoding) :length length)))
    (terminator
      (values (terminated-type encoding) :terminator terminator))))
```

有了这些助手函数，定义`id3-encoded-string`就很简单了。一个需要注意的细节是用在`read-value`和`write-value`调用中的关键字，无论`:length`还是`:terminator`都只是由`string-args`所返回的数据。尽管参数列表中的关键字几乎总是字面关键字，但不一定总是如此。

```
(define-binary-type id3-encoded-string (encoding length terminator)
  (:reader (in)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (read-value type in keyword arg)))
  (:writer (out string)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (write-value type out string keyword arg))))
```

现在可以定义一个名为`text-info`的合成类了，就像你之前定义的`generic-frame`那样。

```
(define-binary-class text-info-frame ()
  ((encoding u1)
   (information (id3-encoded-string :encoding encoding :length (bytes-left 1)))))
```

正如定义`generic-frame`时需要访问帧的大小，本例中为了计算传递给`id3-encoded-string`的`:length`参数也需要同样的信息。由于你接下来在定义的其他类里也需要做类似的计算，所以最好定义另一个助手函数`bytes-left`，它使用`current-binary-object`来得到该帧的大小。

```
(defun bytes-left (bytes-read)
  (- (size (current-binary-object)) bytes-read))
```

现在，就像你定义`generic-frame`合成类一样，你可以使用最少的重复代码来定义两个版本相关的基本类。

```
(define-binary-class text-info-frame-v2.2 (id3v2.2-frame text-info-frame) ())
(define-binary-class text-info-frame-v2.3 (id3v2.3-frame text-info-frame) ())
```

为了启用这些类，你需要修改`find-frame-class`，当ID表明该帧是一个文本信息帧，也即当ID以T开头并且不是TXX或XXXX时，返回适当的类名。

```
(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "XXXX") :test #'string=)))
     (ecase (length name)
```

```
(3 'text-info-frame-v2.2)
(4 'text-info-frame-v2.3)))
(t
(ecase (length name)
(3 'generic-frame-v2.2)
(4 'generic-frame-v2.3))))
```

## 25.13 评论帧

另一个常用的帧类型是评论帧，它就像一个带有额外字段的文本信息帧。和文本信息帧一样，它以代表帧中所采用的字符串编码方式的单个字节开始。该字节后跟一个三字符的ISO 8859-1字符串（无论字符串编码方式字节的值是什么），它代表了评论所使用的语言，以ISO 639-2格式的代码来表示，例如“eng”代表英语，而“jpn”代表日语。这个字段之后是两个根据第一个字节的值所编码的字符串。前一个字符串是以空字节结尾的，含有评论的简要描述。后一个字符串是整个帧的最后部分，保存评论文本。

```
(define-binary-class comment-frame ()
  ((encoding u1)
   (language (iso-8859-1-string :length 3))
   (description (id3-encoded-string :encoding encoding :terminator +null+))
   (text (id3-encoded-string
          :encoding encoding
          :length (bytes-left
                    (+ 1 ; encoding
                       3 ; language
                       (encoded-string-length description encoding t)))))))
```

25

和text-info混合类的定义一样，你可以使用bytes-left来计算最后一个字符串的大小。不过，由于description字段是变长的字符串，在text开始前所需读取的字节数并非常量。更糟糕的是，用来编码text的字节数取决于编码方式。因此，你应当定义一个助手函数，在给定字符串、编码方式和一个指明字符串是否以某个额外字符结尾等参数的情况下，返回用来编码这个字符串所需的字节数。

```
(defun encoded-string-length (string encoding terminated)
  (defun encoded-string-length (string encoding terminated)
    (let ((characters (+ (length string)
                          (if terminated 1 0)
                          (ecase (encoding (0 0) (1 1))))))
      (* characters (ecase encoding (0 1) (1 2))))))
```

然后，和前面一样，你可以定义具体的版本相关的评论帧类并将其嵌入到find-frame-class中。

```
(define-binary-class comment-frame-v2.2 (id3v2.2-frame comment-frame) ())
(define-binary-class comment-frame-v2.3 (id3v2.3-frame comment-frame) ())
(defun find-frame-class (name)
```

```
(cond
  ((and (char= (char name 0) #\T)
        (not (member name '("TXX" "XXXX") :test #'string=)))
   (ecase (length name)
     (3 'text-info-frame-v2.2)
     (4 'text-info-frame-v2.3)))
   ((string= name "COM") 'comment-frame-v2.2)
   ((string= name "COMM") 'comment-frame-v2.3)
   (t
    (ecase (length name)
      (3 'generic-frame-v2.2)
      (4 'generic-frame-v2.3))))))
```

## 25.14 从ID3标签中解出信息

现在你有了读写ID3标签的基本能力，扩展这些代码的方向有很多。如果你想开发一个完整的ID3标签编辑器，那么你需要实现用于所有帧类型的特定类。你还需要定义以一致的方式来管理标签和帧对象的方法。（比如说，如果你改变了一个text-info-frame中的字符串的值，那么就可能需要调整其大小。）目前的代码无法保证这一点。<sup>①</sup>

或者，如果你只需要从MP3文件的ID3标签里解出关于它的特定信息——如同你即将在第27、28和29章里开发一个流式MP3服务器时所做的那样，那么就需要编写函数来查找适当的帧并解出你想要的信息。

最后，为了使其成为产品级的代码，你需要仔细确认ID3规范并处理所有之前没有讨论到的细节。特别地，某些标签和帧中的标志位可能影响标签或帧的读取方式。除非你编写了代码在这些标志位被设定时做正确的处理，否则就可能会有一些ID3标签无法被正确解析。但是本章的代码应当可以解析你实际遇到的几乎所有的MP3文件。

目前你可以再编写几个用来从一个id3-tag中解出个别信息的函数来结束这项工作。你将在第27章或者有可能在用到该库的其他代码里需要这些函数。它们之所以属于该库是因为它们依赖于ID3格式的细节，而这不是库的用户应当关心的。

比如说，为了从一个被解出的id3-tag中获得MP3的歌曲名，你需要查找带有特定标识符的ID3帧并解出其information字段。而另外一些信息，例如歌曲的风格，可能还需要进行后续的解码。幸运的是，所有包含你所关心信息的帧都是文本信息帧。因此，解出一段特定信息的操作基本上可以细化成使用正确的标识符来查找对应的帧。当然，ID3的作者们可能决定将所有标识符从ID3v2.2改为ID3v2.3，所以你必须考虑到这一点。

没有什么太复杂的东西，你只需找出正确的路径从而得到不同的信息就好了。这正是采用交

<sup>①</sup> 确保这类跨字段的一致性是访问广义函数的:after方法的良好应用场合。例如，你可以定义下面的:after方法来确保size与information字符串同步：

```
(defmethod (setf information) :after (value (frame text-info-frame))
  (declare (ignore value))
  (with-slots (encoding size information) frame
    (setf size (encoded-string-length information encoding nil))))
```

互式开发的好机会，跟你之前用来找出需要实现哪些帧的方法大致相同。一开始，需要得到一个 id3-tag 对象来进行后续的操作。假设你手头刚好有一个 MP3 文件，可以像下面这样来使用 read-id3：

```
ID3V2> (defparameter *id3* (read-id3 "Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
ID3V2> *id3*
#<ID3V2.2-TAG @ #x73d04c1a>
```

其中的 Kitka/Wintersongs/02 Byla Cesta.mp3 需要替换成你自己的 MP3 文件名。得到了 id3-tag 对象，你就可以拿它做实验了。例如，可以使用 frames 函数来检出所有帧对象的列表。

```
ID3V2> (frames *id3*)
(#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04dba>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04ea2>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04f9a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05082>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0516a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05252>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0533a>
 #<COMMENT-FRAME-V2.2 @ #x73d0543a>
 #<COMMENT-FRAME-V2.2 @ #x73d05612>
 #<COMMENT-FRAME-V2.2 @ #x73d0586a>)
```

现在假设你想要解出歌曲标题，它很可能就藏在上面的那些帧里。但为了找到它，你需要查找带有 “TT2” 标识符的帧。一个足够简单的方法是像下面这样解出所有的标识符来查看标签中是否含有这样一个帧。

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

第一个帧就是。不过，无法保证它总是第一个帧，因此任何时候都应该通过标识符而不是位置来寻找它。另外也可以直接使用 FIND 函数。

```
ID3V2> (find "TT2" (frames *id3*) :test #'string= :key #'id)
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

现在，为了得到帧中的实际信息，可以这样做：

```
ID3V2> (information (find "TT2" (frames *id3*) :test #'string= :key #'id))
"Byla Cesta^@"
```

晕！那个 ^@ 是 Emacs 打印空字符的方式。在一次从 ID3v1 升级到 ID3v1.1 的行动中，一个文本信息帧的 information 槽，尽管没有正式地被定义为空终止的字符串，却可以含有一个空字符，并且 ID3 读取器本该忽略掉空字符以后的任何字符。因此，你需要一个函数来接受一个字符串并返回该字符串直到第一个空字符之前的内容。使用二进制数据处理库的 +null+ 常量可以轻易做到这一点。

```
(defun upto-null (string)
  (subseq string 0 (position +null+ string)))
```

现在可以得到正确的标题了。

```
ID3V2> (upto-null
  (information (find "TT2" (frames *id3*) :test #'string= :key #'id)))
"Byla Cesta"
```

可以将这些代码直接封装到一个接受id3-tag作为参数的名为song的函数里，然后工作就算是完成了。不过，这些代码与你用来解出其他必要信息（例如专辑名、艺术家和风格）的代码的唯一区别就是标识符。因此，最好可以将代码拆分一下。对于初学者来说，可以编写一个函数，像下面这样通过给定一个id3-tag和一个标识符来查找帧：

```
(defun find-frame (id3 id)
  (find id (frames id3) :test #'string= :key #'id))

ID3V2> (find-frame *id3* "TT2")
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

然后另外一些代码，也就是从text-info-frame中解出具体信息的那部分，可以写在另一个函数里。

```
(defun get-text-info (id3 id)
  (let ((frame (find-frame id3 id)))
    (when frame (upto-null (information frame)))))

ID3V2> (get-text-info *id3* "TT2")
"Byla Cesta"
```

现在song函数的定义就只剩下传递正确的标识符了。

```
(defun song (id3) (get-text-info id3 "TT2"))

ID3V2> (song *id3*)
"Byla Cesta"
```

不过，这个song的定义只适用于版本2.2的标签，因为在版本2.2和2.3之间标识符从“TT2”变成了“TIT2”，所有其他的标签也改变了。考虑到该库的用户在获取歌曲标题这么简单的事情上不应该被迫去关注ID3格式的不同版本，因此你应该帮用户处理好这些细节。一个简单的方法是修改find-frame，让它不只是接受单个标识符，而是像下面这样接受一个标识符的列表：

```
(defun find-frame (id3 ids)
  (find-if #'(lambda (x) (find (id x) ids :test #'string=)) (frames id3)))
```

然后稍微改变get-text-info，使其可以通过&rest参数接受更多的标识符。

```
(defun get-text-info (id3 &rest ids)
  (let ((frame (find-frame id3 ids)))
    (when frame (upto-null (information frame)))))
```

为了允许song同时支持版本2.2和2.3的标签，随后需要改动的只是将版本2.3的标识符添加进来。

```
(defun song (id3) (get-text-info id3 "TT2" "TIT2"))
```

接下来，你只需为那些你想要提供访问函数的字段查找适当的版本2.2和2.3的帧标识符。下



面是一些你将在第27章里用到的函数：

```
(defun album (id3) (get-text-info id3 "TAL" "TALB"))

(defun artist (id3) (get-text-info id3 "TP1" "TPE1"))

(defun track (id3) (get-text-info id3 "TRK" "TRCK"))

(defun year (id3) (get-text-info id3 "TYE" "TYER" "TDRC"))

(defun genre (id3) (get-text-info id3 "TCO" "TCON"))
```

最后的难点是保存在TCO或TCON帧中的genre并不容易看明白。在ID3v1中，风格被保存在单个字节中，使用来自一个固定列表的特别风格进行编码。不幸的是，这些代码继续存在于ID3v2中。如果风格字段的文本是一个位于括号中的数字，那么这个数字将被解释成一个ID3v1风格代码。但话又说回来，这个库的用户可能并不关心这些年代久远的历史。所以，你应该提供一个函数来自动地转换这些风格。下面的函数使用刚刚定义的genre函数来解出实际的风格文本，并检查其是否以一个左括号开始。然后在检测通过时使用一个即将定义的函数来解码版本1的风格代码：

```
(defun translated-genre (id3)
  (let ((genre (genre id3)))
    (if (and genre (char= #\ ( (char genre 0)))
             (translate-v1-genre genre)
             genre)))
```

版本1的风格代码本质上只是一个标准名称数组的索引，因此实现translate-v1-genre的最简单方法就是从风格字符串中解出那个数字，并将其作为访问实际数组的索引。

```
(defun translate-v1-genre (genre)
  (aref *id3-v1-genres* (parse-integer genre :start 1 :junk-allowed t)))
```

然后，你需要做的就是定义这些名字数组了。下面的名字数组包含了80种官方的版本1风格，外加由Winamp发明者所创建的附加风格。

```
(defparameter *id3-v1-genres*
  #(
    ; These are the official ID3v1 genres.
    "Blues" "Classic Rock" "Country" "Dance" "Disco" "Funk" "Grunge"
    "Hip-Hop" "Jazz" "Metal" "New Age" "Oldies" "Other" "Pop" "R&B" "Rap"
    "Reggae" "Rock" "Techno" "Industrial" "Alternative" "Ska"
    "Death Metal" "Pranks" "Soundtrack" "Euro-Techno" "Ambient"
    "Trip-Hop" "Vocal" "Jazz+Funk" "Fusion" "Trance" "Classical"
    "Instrumental" "Acid" "House" "Game" "Sound Clip" "Gospel" "Noise"
    "AlternRock" "Bass" "Soul" "Punk" "Space" "Meditative"
    "Instrumental Pop" "Instrumental Rock" "Ethnic" "Gothic" "Darkwave"
    "Techno-Industrial" "Electronic" "Pop-Folk" "Eurodance" "Dream"
    "Southern Rock" "Comedy" "Cult" "Gangsta" "Top 40" "Christian Rap"
    "Pop/Funk" "Jungle" "Native American" "Cabaret" "New Wave"
    "Psychedelic" "Rave" "Showtunes" "Trailer" "Lo-Fi" "Tribal"
    "Acid Punk" "Acid Jazz" "Polka" "Retro" "Musical" "Rock & Roll"
    "Hard Rock"
```

```
; These were made up by the authors of Winamp but backported into
;; the ID3 spec.
"Folk" "Folk-Rock" "National Folk" "Swing" "Fast Fusion"
"Bebob" "Latin" "Revival" "Celtic" "Bluegrass" "Avantgarde"
"Gothic Rock" "Progressive Rock" "Psychedelic Rock" "Symphonic Rock"
"Slow Rock" "Big Band" "Chorus" "Easy Listening" "Acoustic" "Humor"
"Speech" "Chanson" "Opera" "Chamber Music" "Sonata" "Symphony"
"Booty Bass" "Primus" "Porn Groove" "Satire" "Slow Jam" "Club"
"Tango" "Samba" "Folklore" "Ballad" "Power Ballad" "Rhythmic Soul"
"Freestyle" "Duet" "Punk Rock" "Drum Solo" "A capella" "Euro-House"
"Dance Hall"

; These were also invented by the Winamp folks but ignored by the
;; ID3 authors.
"Goa" "Drum & Bass" "Club-House" "Hardcore" "Terror" "Indie"
"BritPop" "Negerpunk" "Polisk Punk" "Beat" "Christian Gangsta Rap"
"Heavy Metal" "Black Metal" "Crossover" "Contemporary Christian"
"Christian Rock" "Merengue" "Salsa" "Thrash Metal" "Anime" "Jpop"
"Synthpop"))
```

你可能感觉自己在本章里又写了大量代码。但如果你将它们全部放在一个文件里，或是下载了本书Web站点上的版本，你会发现其实并没有多少行——编写这个库的主要难点在于理解ID3格式本身的复杂性。不管怎么说，现在你有了将在第27、28和29章里编写的流式MP3服务器的主体。而另一个主要的基础性内容是一种编写服务器端Web软件的方式，这就是下一章的主题。

# 实践：用AllegroServe 进行Web编程



**本**章将利用开源的AllegroServe Web服务器来学习在Common Lisp中开发Web应用的一种方法。这并不意味着本章是对AllegroServe的完整介绍。我只打算介绍关于Web编程这个大型话题的冰山一角。我的目标是涵盖足够多的AllegroServe基本用法，以确保你可以在第29章里用它来开发一个可以浏览MP3文件库，并将它们以流的方式发送到MP3客户端的应用程序。类似地，本章也为初学者提供了一个关于Web编程的简要介绍。

## 26.1 30 秒介绍服务器端 Web 编程

尽管当今的Web程序开发通常都会用到相当数量的软件框架和不同的协议，但Web编程的核心部分自从它们在20世纪90年代早期被发明以后几乎没有什么变化。对于第29章里将要编写的那些简单应用，你只需理解几个关键的概念就可以了，因此这里将快速地概述一下。有经验的Web程序员可以略读或是干脆跳过本节。<sup>①</sup>

首先，你需要理解Web浏览器和Web服务器在Web编程中的角色。尽管现代浏览器通常带有大量花哨的功能，但Web浏览器的核心功能只是从Web服务器上请求Web页并将它们渲染出来。通常，这些页面是使用超文本标记语言（HTML）来编写的，HTML告诉浏览器如何渲染页面，包括在哪里插入内嵌的图像和指向其他Web页的链接。HTML由带有标签的文本组成，这些标签为文本添加了结构，使浏览器得以渲染页面。一个简单的HTML文档如下所示：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
    <p>This is a picture: </p>
```

<sup>①</sup> Web编程的初学者需要在这篇介绍的基础上补充阅读一两篇更加深入的介绍。你可以在<http://www.jmarshall.com/easy/>上找到一些很好的在线指导。

```
<p>This is a <a href="another-page.html">link</a> to another page.</p>
</body>
</html>
```

图26-1 显示了浏览器是如何渲染这个页面的。

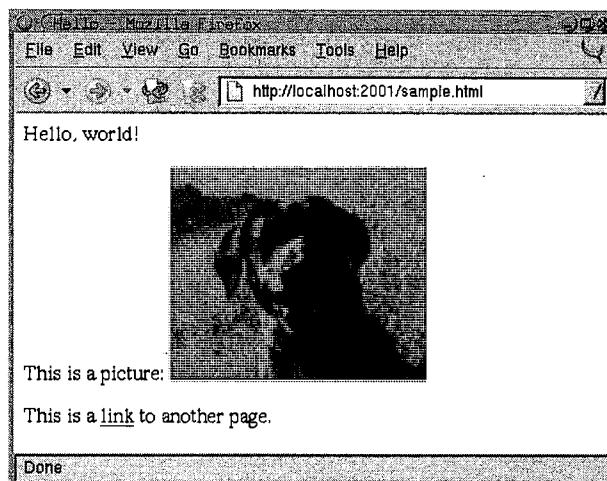


图26-1 网页示例

浏览器和服务器之间使用一种称为超文本传输协议（HTTP）的协议进行通信。尽管你不需要关心该协议的细节，但有必要知道它是由浏览器发起的请求和服务器生成的响应的规范。这就是说，浏览器连接到Web服务器并发送了一个至少包括目标URL和浏览器支持的HTTP版本的请求。浏览器也可以在它的请求中包含数据。这就是浏览器向服务器提交HTML表单的方式。

为了回应一个请求，服务器发送一个包括一系列的头部和一个主体的响应。头部中含有关于主体的信息，诸如数据的类型是什么（比如HTML、纯文本或一个图片），而主体就是数据本身，随后将被浏览器渲染。服务器有时也会发送一个错误响应来告诉浏览器其请求因为某种原因无法正确回应。

情况基本上就是这样。一旦浏览器从服务器那里收到了完整的响应，那么直到下一次浏览器决定从服务器请求一个页面之前，在浏览器和服务器之间将不再有任何通信。<sup>①</sup>这就是Web编程的主要约束所在——无法让运行在服务器上的代码影响用户在浏览器中看到的内容，除非浏览器向服务器发起一个新请求。<sup>②</sup>

有些称为静态页面的Web页只是保存在Web服务器上的HTML文件，在浏览器发出请求时直接被发送出去。另一方面，动态页面是由每次页面被浏览器请求时生成的HTML构成的。一个动

<sup>①</sup> 加载单个Web页面可能实际上会产生多个请求。为了渲染一个含有内嵌图片的页面的HTML，浏览器必须单独地请求每个图片，再将它们分别插入到渲染后的HTML中的适当位置。

<sup>②</sup> Web编程的许多复杂性都是试图解决这个基本限制的结果，目标是提供类似桌面应用那样的用户体验。

态页面可能会在查询数据库的基础上生成并构造出HTML来表示查询的结果。<sup>①</sup>

当针对请求生成响应时，服务器端的代码需要处理四种主要信息。第一种信息是被请求的URL。不过，URL通常被Web服务器本身用来决定使用哪些代码来生成响应。接下来，如果URL中含有一个问号，那么问号之后的所有内容将被视为一个查询字符串，后者通常会被Web服务器忽略，除非将它传给用来生成响应的代码。多数时候查询字符串由一组键/值对组成。来自浏览器的请求也可以POST数据，这些数据通常也由键值对构成。POST数据一般用来提交HTML表单。无论是查询字符串中的键值对还是发送数据中的键值对都被统称为查询参数。

最后，为了将来自同一个浏览器的一系列请求串接在一起，服务器中运行的代码可以设置cookie，并在浏览器的响应中发送一个特殊的头部，里面含有一些不透明数据。一旦cookie被一个特定的服务器所设置，那么浏览器将在每次向该服务器发送请求时都带上这个cookie。浏览器并不关心cookie中的数据，它只是将其回显给服务器，让服务器端的代码按照它们想要的方式来解释。

以上就是99%的服务器端Web编程所依赖的基础元素。浏览器发起一个请求，服务器查找用来处理该请求的代码并运行它，然后代码使用查询参数和cookie来决定要做什么。

## 26.2 AllegroServe

有很多种方式可以用Common Lisp来提供Web内容。至少有三种用Common Lisp写的开源Web服务器，还有诸如mod\_lisp<sup>②</sup>和Lisplets<sup>③</sup>这类可以允许Apache Web服务器或任何Java Servlet容器将请求代理到运行在独立进程中的Lisp服务器上的系统。

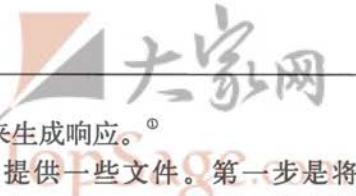
对于本章来说，你将使用开源Web服务器AllegroServe的某个版本，它最初由Franz Inc.的John Foderaro所开发。AllegroServe包含在来自Franz的用于本书的Allegro版本里。如果你没在使用Allegro，那么你可以使用PortableAllegroServe，即一个AllegroServe代码树的友好分支，它还包括一个让PortableAllegroServe得以运行在多数Common Lisp平台上的兼容层。本章和第29章里编写的代码应该可以同时运行在原版的AllegroServe和PortableAllegroServe上。

AllegroServe提供了一个与Java Servlet类似的编程模型。每当浏览器请求一个页面时，AllegroServe会解析请求并查找一个称为实体(entity)的对象来处理该请求。一些作为AllegroServe一部分的实体类知道如何处理静态内容——无论是单独的文件还是一个目录树的内容。而另一些

<sup>①</sup> 不幸的是，“动态”一词在Web世界中被重载了。术语“动态HTML”指的是含有嵌入式代码的HTML，其代码通常采用JavaScript来编写，JavaScript可在不跟Web服务器进行通信的情况下在浏览器中执行。如果谨慎使用，动态HTML可以改进一个基于Web的应用程序的可用性，因为即便在高速的Internet连接下，向一个Web服务器发出请求、接受响应并渲染新页面也需要花费相当长的时间。更加令人困惑的是，动态生成的页面（换句话说，是在服务器上生成的页面）也可以含有动态HTML（运行在客户端的代码）。对于本书中的应用，你将只是动态地生成简单的非动态HTML。

<sup>②</sup> [http://www.fractalconcept.com/asp/html/mod\\_lisp.html](http://www.fractalconcept.com/asp/html/mod_lisp.html)。

<sup>③</sup> <http://lisplets.sourceforge.net/>。



则是我将用本章的多数篇幅进行讨论的，它们运行任意Lisp代码来生成响应。<sup>①</sup>

但在开始之前，你需要知道如何启动AllegroServe并让它提供一些文件。第一步是将AllegroServe加载到你的Lisp映像中。在Allegro中，可以简单地键入(require :aserve)。在其他Lisp环境（也包括Allegro在内）下，可以通过加载portableaserve目录树顶层的文件INSTALL.lisp来加载PortableAllegroServe。加载AllegroServe会创建三个新包：NET.ASERVE、NET.HTML.GENERATOR和NET.ASERVE.CLIENT。<sup>②</sup>

加载了服务器以后，你可以通过NET.ASERVE包中的函数start来启动它。为了可以方便地访问来自NET.ASERVE、COM.GIGAMONKEYS.HTML（一个即将讨论到的新包）以及Common Lisp其余部分的导出符号，你应该像下面这样创建一个新包：

```
CL-USER> (defpackage :com.gigamonkeys.web
  (:use :cl :net.aserve :com.gigamonkeys.html))
#<The COM.GIGAMONKEYS.WEB package>
```

现在使用下面的IN-PACKAGE表达式切换到该包上：

```
CL-USER> (in-package :com.gigamonkeys.web)
#<The COM.GIGAMONKEYS.WEB package>
WEB>
```

现在你可以无需限定符而使用来自NET.ASERVE的导出符号了。函数start用来启动服务器，它接受相当数量的关键字参数，但你现在唯一需要传递的是:port，即监听的端口。你可能需要使用诸如2001这种较大的端口而不是HTTP服务器的标准端口80。因为在类Unix的操作系统里，只有root用户才能监听1024以下的端口。为了在Unix上运行监听80端口的AllegroServe，你需要以root用户启动Lisp，然后使用:setuid和:setgid参数来告诉start在打开端口以后切换到指定的身份。可以像下面这样启动监听端口2001的服务器：

```
WEB> (start :port 2001)
#<WSERVER port 2001 @ #x72511c72>
```

服务器现在在你的Lisp环境中运行了。在启动服务器时或许会看到类似“port already in use”这样的错误提示。这表明端口2001已经被系统里的其他服务器占用了。在这种情况下，最简单的修复方法是使用一个不同的端口，为start提供一个不同的参数，然后在本章其余部分的URL里始终用该值来代替2001。

你可以继续通过REPL与Lisp环境交互，因为AllegroServe启动了它自己的线程来处理来自浏览器的请求。这意味着你至少可以通过REPL来观察当前运行中的服务器，这使得调试和测试工作比面对一个完全黑箱的服务器要容易得多。

假设你正在运行的Lisp环境与你的浏览器是在同一台机器上，那么你可以通过将浏览器指向http://localhost:2001/来检查服务器是否已经启动并运行了。此刻你应该会在浏览器中得到一个页

<sup>①</sup> AllegroServe也提供了一个名叫Webactions的框架，其类似于Java中的JSP。这个框架不是编写代码来生成HTML，而是通过Webactions你可以直接编写本质上是HTML的页面，但其中的某些内容将在页面提供服务时作为代码来运行。在本书里我将不会谈及Webactions。

<sup>②</sup> 加载PortableAllegroServe将为相关的兼容库创建出其他一些包，但你需要关心的只是那三个包。

面未找到的错误信息，因为你还没有发布任何内容。但这个错误信息来自AllegroServe，你可以从页面的底部看到这点。另一方面，如果浏览器显示了一个错误对话框并提示说“The connection was refused when attempting to contact localhost:2001”，那么这意味着要么服务器没有运行，要么是从不同于2001的端口启动的。

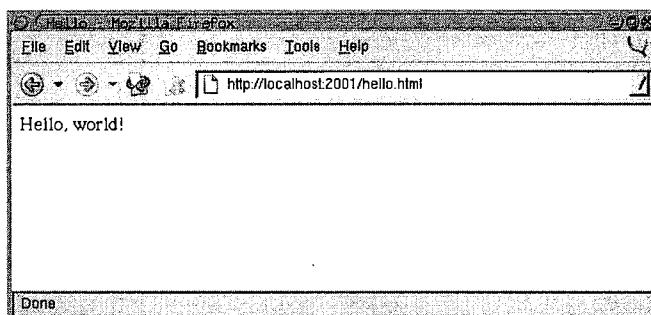
现在你可以发布一些文件了。假设在/tmp/html目录下有一个文件hello.html，其内容如下：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

你可以使用publish-file函数单独地发布它。

```
WEB> (publish-file :path "/hello.html" :file "/tmp/html/hello.html")
#<NET.ASERVE::FILE-ENTITY @ #x725eddea>
```

其中的:path参数将出现在浏览器请求的URL中，而:file参数则是文件系统中的文件名。在求值publish-file表达式之后，可以将浏览器指向http://localhost:2001/hello.html，然后它将显示一个类似图26-2这样的页面。



26

图26-2 http://localhost:2001/hello.html

你也可以使用publish-directory函数来发布整个目录树中的文件。首先让我们使用下面的publish-file调用将已发布的內容清除：

```
WEB> (publish-file :path "/hello.html" :remove t)
NIL
```

现在，你可以使用publish-directory函数发布整个/tmp/html/目录（包括它的所有子目录）了。

```
WEB> (publish-directory :prefix "/" :destination "/tmp/html/")
#<NET.ASERVE::DIRECTORY-ENTITY @ #x72625aa2>
```

在本例中，`:prefix`参数指定了应由该实体接手的URL路径部分的开始。这样，如果服务器收到了一个来自`http://localhost:2001/foo/bar.html`的请求，那么其路径部分是`/foo/bar.html`，以“/”开始。这个路径随后通过将其前缀“/”替换成目标“/tmp/html/”从而变成了一个文件名。同样的道理，`http://localhost:2001/hello.html`也将被转化成一个对文件`/tmp/html/hello.html`的请求。

## 26.3 用AllegroServe生成动态内容

发布生成动态内容的实体几乎和发布静态内容一样简单。函数`publish`和`publish-prefix`是`publish-file`和`publish-directory`对应的动态版本。这两个函数的基本思想是，你可以发布一个函数，它将被调用来生成一个指定URL或带有给定前缀的任何URL的响应。这个函数将用两个参数来调用：一个代表请求的对象以及一个被发布的实体。多数时候你不需要对那个实体对象做任何操作，除非将它和一些后面即将讨论到的宏一起传递。另一方面，你将使用请求对象来获取由浏览器提交的信息，即包含在URL或使用HTML表单发送的数据中的查询参数。

作为使用函数来生成动态内容的简单示例，让我们编写一个在每次请求时生成一个带有不同随机数的页面的函数。

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (format
        (request-reply-stream request)
        "<html>~@
         <head><title>Random</title></head>~@
         <body>~@
           <p>Random number: ~d</p>~@
         </body>~@
       </html>~@
      "
      (random 1000)))))
```

宏`with-http-response`和`with-http-body`是AllegroServe的一部分。前者启动生成一个HTTP响应的过程，并且可以像这样指定诸如返回内容的类型之类的信息。它还可以处理HTTP的其他部分，例如处理`If-Modified-Since`请求。`with-http-body`实际发送HTTP回执头并执行其主体，后者应当含有用来生成响应内容的代码。在`with-http-response`中`with-http-body`之前的地方，你可以添加或修改在回执中发送的HTTP头。函数`request-reply-stream`也是AllegroServe的一部分，它返回一个流用来向浏览器中写入想要的输出。

正如该函数显示的，可以只用`FORMAT`将HTML打印到由`request-reply-stream`返回的流上。在下一节里，我将向你展示更方便的方法来以编程方式生成HTML。<sup>①</sup>

<sup>①</sup> ~@后接一个新行可以告诉`FORMAT`忽略换行之后的所有空白，这样就可以精美地缩进代码而不会在HTML中增加大量的空白。由于HTML中的空白通常会被忽略，因此这不会影响到浏览器，但它可以让产生的HTML看起来更美观。

现在你可以发布这个函数了。

```
WEB> (publish :path "/random-number" :function 'random-number)
#<COMPUTED-ENTITY @ #x7262bab2>
```

参数:`:path`与它在`publish-file`函数中的用法相同，它指定导致该函数被调用的URL的路径部分。`:function`参数用来指定函数的名字或实际的函数对象。像这样使用一个函数的名字可以允许你以后重定义该函数而无需重新发布即可令AllegroServe使用新的函数定义。在求值了`publish`调用以后，可以让浏览器指向`http://localhost:2001/random-number`来得到一个带有一个随机数的页面，如图26-3所示。

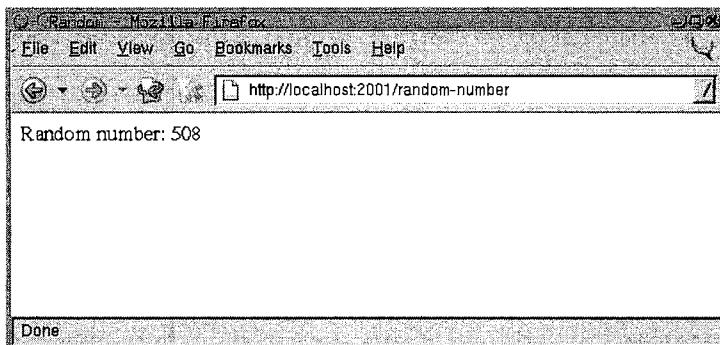


图26-3 `http://localhost:2001/random-number`

## 26.4 生成 HTML

尽管使用`FORMAT`来生成HTML对于到目前为止所讨论的简单页面还不错，但如果要构建更复杂的页面，那么如果有一种更简洁的HTML生成方式就更好了。有几个库可以用来从S-表达式形式的表示生成HTML，其中的`htmlgen`就包含在AllegroServe中。在本章里你将使用一个称为FOO<sup>①</sup>的库，它在很大程度上来自Franz的`htmlgen`，并且你将在第30章和第31章里看到更多关于它的具体实现的细节。不过目前你只需要知道如何使用FOO。

从Lisp里生成HTML是件相当自然的事情，因为本质上S-表达式跟HTML是同构的。你可以用S-表达式来表示HTML元素，方法是将HTML中的每个元素视为一个以适当头元素“标记”的列表，例如一个与HTML标签同名的关键字符串。这样，HTML `<p>foo</p>`就可以用S-表达式`(:p "foo")`来表示了。由于HTML元素嵌套的方式与S-表达式中的列表嵌套方式相同，因此上述表示法可以扩展到更复杂的HTML。例如，下面的HTML

```
<html>
  <head>
    <title>Hello</title>
  </head>
```

26

<sup>①</sup> FOO是来源于FOO Outputs Output的递归伪技术缩略语。

```
<body>
<p>Hello, world!</p>
</body>
</html>
```

可以用下列S-表达式来表示：

```
(:html
  (:head (:title "Hello"))
  (:body (:p "Hello, world!")))
```

带有属性的HTML元素会稍微复杂一些，但也不是无法解决。FOO支持两种在标签中添加属性的方式。一种方式是简单地在列表的第一个元素之后跟上一个键值对。跟在键值对后面的第一个不是关键字的元素代表该HTML元素内容的开始。这样，你可以将下面的HTML

```
<a href="foo.html">This is a link</a>
```

用下列S-表达式来表示：

```
(:a :href "foo.html" "This is a link")
```

FOO支持的另一种语法是将标签名和属性组织在它们自己的列表中，如下所示：

```
((:a :href "foo.html") "This is link.")
```

FOO可以通过这两种方式使用S-表达式来表示HTML。函数emit-html接受一个HTML的S-表达式并输出相应的HTML。

```
WEB> (emit-html '(:html (:head (:title "Hello")) (:body (:p "Hello, world!"))))
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
T
```

不过，emit-html并非总是最有效的HTML生成方式，因为其参数必须是想要生成的HTML的完整S-表达式表示。尽管构造这样一个表示很容易，但它却并不总是高效的。例如，假设你想要生成一个含有10 000个随机数的列表的HTML页面。你可以像下面这样使用一个反引用模板来构造S-表达式并将其传给emit-html：

```
(emit-html
 `(:html
   (:head
     (:title "Random numbers"))
   (:body
     (:h1 "Random numbers")
     (:p ,@(loop repeat 10000 collect (random 1000) collect " "))))
```

不过，这会导致在实际开始生成HTML之前就先要构造出一个含有10 000个元素的列表的树来，而一旦HTML生成出来以后，整个S-表达式就没有任何用处了。为了避免这种低效，FOO还

支持一个宏html，它允许你在一个HTML的S-表达式中嵌入一点儿Lisp代码。

位于html宏的输入中的诸如字符串和数字这样的字面值将被插入到输出的HTML中。同样，符号将被视为对变量的引用，宏所生成的代码会在运行期输出它们的值。这样，下面两个形式

```
(html (:p "foo"))

(let ((x "foo")) (html (:p x)))
```

都将生成下面的代码：

```
<p>foo</p>
```

不以一个关键字符开始的列表形式会被视为代码，并被嵌入到生成的代码中。被嵌入的代码返回的任何值都将被忽略，但是代码可以通过调用html宏本身来产生更多的HTML。例如，为了在HTML中输出一个列表的内容，你可以写成下面这样

```
(html (:ul (dolist (item (list 1 2 3)) (html (:li item))))))
```

它将产生下面的HTML：

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

如果你想输出一个列表形式的值，必须将其包装在伪标签:print中。这样，表达式

```
(html (:p (+ 1 2)))
```

在计算并丢弃值3以后会生成下面的HTML

```
<p></p>
```

为了输出那个3，必须写成下面这样：

```
(html (:p (:print (+ 1 2))))
```

26

或者也可以先计算出该值并将其保存在一个html调用之外的变量里，如下所示：

```
(let ((x (+ 1 2))) (html (:p x)))
```

这样，你就可以使用html宏来生成随机数的列表了，如下所示：

```
(html
  (:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p (loop repeat 10 do (html (:print (random 1000)) "))))))
```

宏版本将比emit-html版本更加高效。你不仅不再需要生成一个代表整个页面的S-表达式，而且emit-html的很多在运行期解释S-表达式的工作现在都可以在宏展开时一次性完成，而不必在每次代码运行时来做了。

你可以通过宏`with-html-output`来控制由`html`和`emit-html`生成的输出被发送到哪里，该宏是FOO库的一部分。这样，你可以使用来自FOO的`with-html-output`和`html`宏来重写`random-number`，如下所示：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random 1000))))))))
```

## 26.5 HTML 宏

FOO还有一个特性，它允许你定义一种HTML“宏”可将任意形式转化成`html`宏可理解的HTML S-表达式。例如，假设你经常发现自己会编写下列形式的页面：

```
(:html
  (:head (:title "Some title"))
  (:body
    (:h1 "Some title")
    ... stuff ...))
```

那么你应该定义一个HTML宏来捕捉这个模式，就像这样：

```
(define-html-macro :standard-page ((&key title) &body body)
  `(:html
    (:head (:title ,title))
    (:body
      (:h1 ,title)
      ,@body)))
```

现在，你可以在你的S-表达式HTML中使用“标签”`:standard-page`了，它将在被解释或编译之前展开。例如，下面的形式

```
(html (:standard-page (:title "Hello") (:p "Hello, world.")))
```

可以生成这样的HTML：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Hello, world.</p>
  </body>
</html>
```

## 26.6 查询参数

当然，生成HTML输出还只是Web编程的一半，另一半是得到来自用户的输入。正如26.1节中讨论过的，当浏览器从Web服务器上请求一个页面时，它可以在URL和POST数据中发送查询参数，两者都是向服务器端代码提供输入的途径。

和多数Web编程框架一样，AllegroServe可以帮你解析这两种输入。等到你发布的函数被调用时，所有来自查询字符串和/或POST数据的键/值对都已被解码并放置在一个alist中，后者可以使用函数`request-query`从请求对象中获取alist。下面的函数可以返回一个页面，其中显示了所有它收到的查询参数：

```
(defun show-query-params (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
            (:title "Query Parameters")
            (if (request-query request)
                (html
                  (:table :border 1
                    (loop for (k . v) in (request-query request)
                          do (html (:tr (:td k) (:td v))))))
                (html (:p "No query parameters.")))))))))

(publish :path "/show-query-params" :function 'show-query-params)
```

如果你给浏览器一个类似下面这样的带有查询字符串的URL

`http://localhost:2001/show-query-params?foo=bar&baz=10`

那么你应该可以得到一个类似图26-4所示的页面。

26

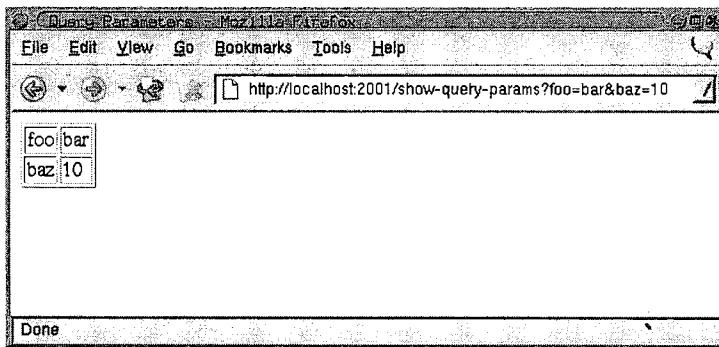


图26-4 `http://localhost:2001/show-query-params?foo=bar&baz=10`

为了生成POST数据，你需要一个HTML表单。下面的函数可以生成一个简单的表单，其数据将发送到`show-query-params`：

```
(defun simple-form (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let ((*html-output* (request-reply-stream request)))
        (html
          (:html
            (:head (:title "Simple Form"))
            (:body
              (:form :method "POST" :action "/show-query-params"
                (:table
                  (:tr (:td "Foo")
                    (:td (:input :name "foo" :size 20)))
                  (:tr (:td "Password")
                    (:td (:input :name "password" :type "password" :size 20))))
                (:p (:input :name "submit" :type "submit" :value "Okay")
                  (:input ::type "reset" :value "Reset")))))))))
  (publish :path "/simple-form" :function 'simple-form))
```

将你的浏览器指向http://localhost:2001/simple-form，然后你应该可以看到一个类似图26-5所示的页面。

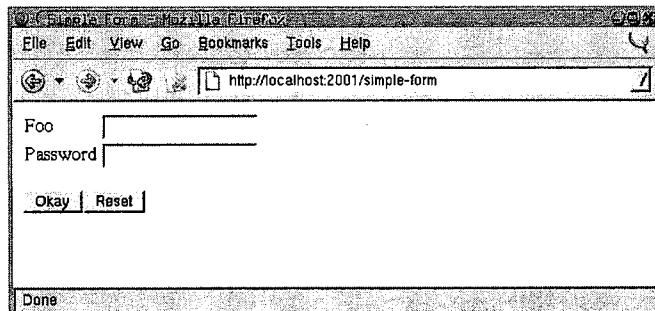


图26-5 http://localhost:2001/simple-form

如果你在表单中填入“abc”和“def”两个值，那么点击Okay按钮应该会把你带到一个类似图26-6所示的页面里。

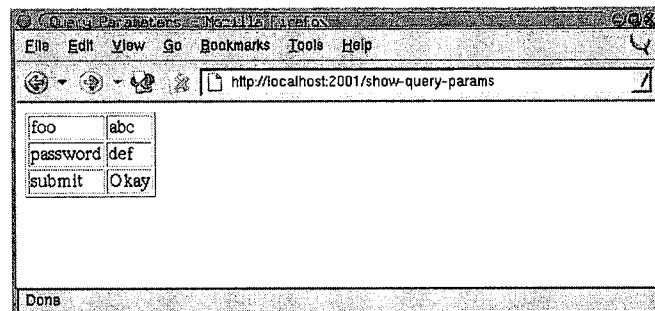


图26-6 提交一个简单表单的结果

尽管如此，多数时候你不需要在所有查询参数上迭代，你只需要提取单独的参数。例如，你可能想要修改random-number，令你传给RANDOM的限制值可以通过一个查询参数来提供。在这种情况下，你可以使用函数request-query-value，它接受一个请求对象和你想要查询的参数名并将其值以字符串的形式返回，或者当没有参数时返回NIL。一个参数化的random-number版本如下所示：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let* ((*html-output* (request-reply-stream request))
             (limit-string (or (request-query-value "limit" request) ""))
             (limit (or (parse-integer limit-string :junk-allowed t) 1000)))
        (html
         (:html
          (:head (:title "Random"))
          (:body
           (:p "Random number: " (:print (random limit))))))))))
```

由于request-query-value可能返回NIL或一个空字符串，在把参数解析成一个用来传给RANDOM的数字时需要同时考虑这两种情况。你可以在绑定limit-string时当没有"limit"查询参数的情况下将它绑定到空字符串""，从而处理NIL的情形。然后，你可以使用带有:junk-allowed参数的PARSE-INTEGER来确保它要么返回NIL（如果不能从给定字符串中解析出整数的话）要么返回一个整数。在26.8节中，你将开发一些宏来使查询参数的提取和到多种类型的转换工作变得更加容易。

## 26.7 cookie

26

在AllegroServe中你可以发送一个Set-Cookie头来告诉浏览器保存一个cookie并将其随着后续请求一起发送，具体方法是，在with-http-response的主体中调用with-http-body之前调用函数set-cookie-header。该函数的第一个参数是请求对象，其余参数都是用来设定cookie中不同属性的关键字参数。其中两个你必须传递的是:name和:value参数，两者都应该是字符串。其他可能影响发送到浏览器的cookie的参数包括:expires、:path、:domain和:secure。

当然，你只需要担心:expires，它控制浏览器应该保存cookie多久。如果:expires是NIL（默认值），那么浏览器只把cookie保存到它被关闭时。其他可能的值是:never，这意味着cookie应当被永远保存下去，或者在一个由GET-UNIVERSAL-TIME或ENCODE-UNIVERSAL-TIME返回的全局时间里被保存。一个值为零的:expire参数告诉客户端立即丢弃已有的cookie。<sup>①</sup>

在设置了一个cookie以后，可以使用函数get-cookie-values得到一个alist，其中含有由浏览器发送的每个cookie对应的键值对。从这个alist中，可以使用ASSOC和CDR来提取单独的cookie值。

下面的函数可以显示出浏览器所发送的所有cookie的名字和值：

<sup>①</sup> 关于其他参数的含义，可参见AllegroServe文档和RFC 2109，这些文档里描述了cookie机制。

```
(defun show-cookies (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request)))
        (html
          (:standard-page
            (:title "Cookies")
            (if (null (get-cookie-values request))
                (html (:p "No cookies."))
                (html
                  (:table
                    (loop for (key . value) in (get-cookie-values request)
                      do (html (:tr (:td key) (:td value)))))))))))
      (publish :path "/show-cookies" :function 'show-cookies)
```

第一次加载页面http://localhost:2001/show-cookies时，它应该会说“No cookies”，如图26-7所示，因为你还没有设置任何cookie。

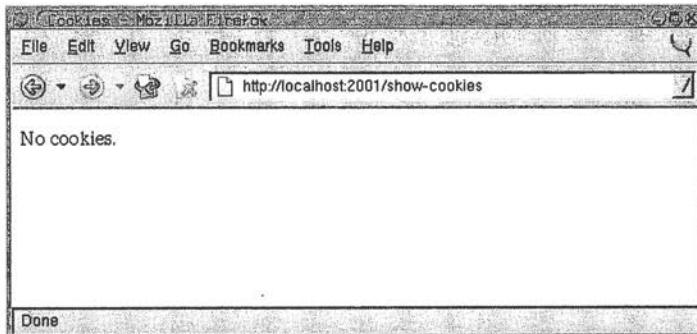


图26-7 没有cookies的http://localhost:2001/show-cookies

为了设置cookie，你需要另外一个函数，例如：

```
(defun set-cookie (request entity)
  (with-http-response (request entity :content-type "text/html")
    (set-cookie-header request :name "MyCookie" :value "A cookie value")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request)))
        (html
          (:standard-page
            (:title "Set Cookie")
            (:p "Cookie set.")
            (:p (:a :href "/show-cookies" "Look at cookie jar.")))))))
  (publish :path "/set-cookie" :function 'set-cookie))
```

如果你输入URL http://localhost:2001/set-cookie，那么你的浏览器应该会显示如图26-8所示的页面。同时，服务器将发送一个Set-Cookie头部，其中带有一个名为“MyCookie”值为“A cookie value”的cookie。如果你点击链接Look at cookie jar，那么你将被带到/show-cookies页面，在

那里你将看到新的cookie，如图26-9所示。由于你并未指定一个`:expires`参数，浏览器将继续在每个请求中发送该cookie，直到你关闭了浏览器。

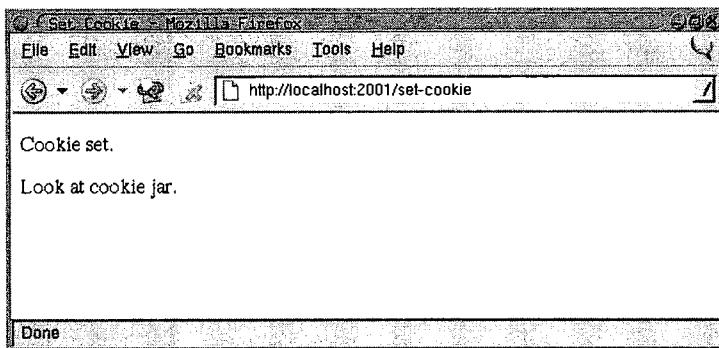


图26-8. http://localhost:2001/set-cookie

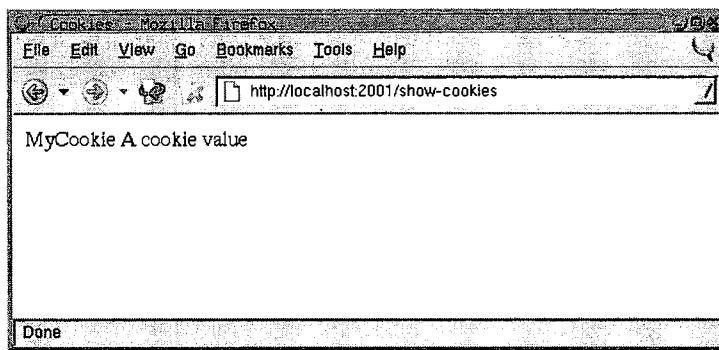


图26-9 设置cookie之后的http://localhost:2001/show-cookies

26

## 26.8 小型应用框架

尽管AllegroServe为你提供了用来编写服务端Web代码的几乎所有基本功能(访问URL的查询字符串和提交数据中的查询参数,设置和获取cookie值的能力,以及生成发给浏览器响应的能力),但这需要写很多令人厌烦的重复性代码。

举个例子,你编写的每一个HTML生成函数都需要带有参数`request`和`entity`,并且它们都会包含对`with-http-response`、`with-http-body`以及(如果你打算使用`FOO`来生成HTML的话)`with-html-output`的调用。然后,在需要获取查询参数的函数里,还会有大量的`request-query-value`调用以及更多的代码来将这些字符串转化成任何你实际需要的类型。最后,你还需要记得`publish`这些函数。

为了减少样板代码的数量,可以在AllegroServer之上编写一个小型的框架,使其可以更容易地用来定义那些处理特定URL请求的函数。

基本的思路是先定义一个宏`define-url-function`，再用它来定义可自动通过`publish`发布的函数。这个宏展开成一个含有适当样板代码的DEFUN，以及在同名的URL下发布该函数的代码。它也负责生成代码来从查询参数和cookie中解出值，并将这些值绑定到声明在函数参数列表中的变量上。这样，`define-url-function`定义的基本形式如下所示：

```
(define-url-function name (request query-parameter*)
  body)
```

其中`body`是产生页面HTML的代码，它将被包装在一个对FOO的`html`宏的调用中，因此对于简单的页面来说，它只含有S-表达式形式的HTML。

在宏的主体中，查询参数变量将被绑定到同名查询参数或来自一个cookie的值上。在最简单的情形下，一个查询参数的值是同名的查询参数或POST数据字段中的字符串。如果查询参数使用列表来指定，还可以指定自动的类型转换、默认值以及是否在cookie中查找并保存该值。`query-parameter`的完整语法如下所示：

```
name | (name type [default-value] [stickiness])
```

其中的`type`必须是一个`define-url-function`可以识别的名字。我将很快讨论如何定义新的类型。`default-value`必须是该给定类型的一个值。最后，如果有`stickiness`，它表示参数的值应当在没有查询参数的情况下从一个适当命名的cookie中获取，并且Set-Cookie头部应当在响应中发送，其中保存了同名cookie的值。这样，在显式地通过一个查询参数的值来指定粘滞参数以后，它将在该页面的后续请求中保持该值，即便没有查询参数被提供。

所使用的cookie名取决于`stickiness`的值：使用值`:global`，cookie将采用与参数相同的命名方式。这样，使用同名的全局粘滞参数的不同函数将共享其值。如果`stickiness`是`:package`，那么cookie的名字将根据参数的名字和函数名所在的包构造出来，这样来自同一个包的函数可以共享一些值，且不必担心被其他包里的函数参数所破坏。最后，一个带有`stickiness`值为`:local`的参数将使用由参数名、函数名所在的包以及函数名构成的cookie，这对该函数而言将是唯一的。

举个例子，你可以使用`define-url-function`来将之前`random-page`的17行定义替换成下面的5行：

```
(define-url-function random-number (request (limit integer 1000))
  (:html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```

如果要限制参数为粘滞的，你可以将`limit`的声明改成`(limit integer 1000 :local)`。

## 26.9 上述框架的实现

我将会自顶向下地解释`define-url-function`的实现。该宏本身如下所示：

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params))))
```

```

`(progn
  (defun ,name (,request ,entity)
    (with-http-response (,request ,entity :content-type "text/html")
      (let* (,@(param-bindings name request params))
        ,@(set-cookies-code name request params)
        (with-http-body (,request ,entity)
          (with-html-output ((request-reply-stream ,request))
            (html ,@body))))))
  (publish :path ,(format nil "/~(~a~)" name) :function ',name))))
```

让我们一点一点地分析它，首先看最初的几行。

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params))))
```

直到这里你才开始生成代码。你可以用**GENSYM**生成一个符号以便在后面的**DEFUN**里把它作为实体参数的名字来使用。然后正则化所有参数，使用下列函数将简单的符号转化成列表形式：

```
(defun normalize-param (param)
  (etypecase param
    (list param)
    (symbol `(.param string nil nil))))
```

换句话说，只用一个符号来声明参数等价于声明不带默认值的非粘滞字符串参数。

接下来是**PROGN**。你必须展开成一个**PROGN**，因为你需要生成代码来做两件事：用**DEFUN**定义函数以及调用**publish**。你应当首先定义该函数，这样如果定义中出现错误，那么该函数将不会被发布。**DEFUN**的前两行只是一些样板代码。

```
(defun ,name (,request ,entity)
  (with-http-response (,request ,entity :content-type "text/html"))
```

现在开始做实际工作。接下来两行将为在**define-url-function**中指定的除**request**以外的参数生成绑定，以及为粘滞性参数调用**set-cookie-header**的代码。当然，实际的工作是由你即将看到的助手函数来完成的。<sup>①</sup>

```
(let* (,@(param-bindings name request params))
  ,@(set-cookies-code name request params))
```

其余的就只是些样板代码了，它们将来自**define-url-function**定义的主体放在适当的**with-http-body**、**with-html-output**和**html**宏的上下文中。然后是对**publish**的调用。

```
(publish :path ,(format nil "/~(~a~)" name) :function ',name))
```

表达式(**format** nil "/~(~a~)" name)在宏展开阶段求值，生成一个由“/”后跟你定义的这个函数名的全小写版本。该字符串随后成为**publish**的:**:path**参数，而函数名则作为:**:function**参数被插入。

26

<sup>①</sup> 你需要使用**LET\***而非**LET**来使参数的默认值形式可以引用更早出现在参数列表中的参数。例如，可以写成下面这样：

```
(define-url-function (request (x integer 10) (y integer (* 2 x))) ...)
```

从而允许当没有显式提供y的值时，则使用x值的两倍。

再看一看用来生成`DEFUN`形式的助手函数。为了生成参数绑定，需要在`params`上循环并收集每个参数的由`param-binding`生成的代码片段。该片段是一个含有需要绑定的变量名和用来计算该变量值的代码的列表。用来计算值的代码的确切形式取决于参数的类型是否为粘滞的以及其默认值，如果有的话。因为你已经正则化了所有的参数，所以你可以在`param-binding`中使用`DESTRUCTURING-BIND`来将各部分取出。

```
(defun param-bindings (function-name request params)
  (loop for param in params
        collect (param-binding function-name request param)))

(defun param-binding (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (let ((query-name (symbol->query-name name))
          (cookie-name (symbol->cookie-name function-name name sticky)))
      `',(name (or
                 (string->type ',type (request-query-value ,query-name ,request))
                 ,(if cookie-name
                      (list `(string->type ',type
                                         (get-cookie-value ,request ,cookie-name)))
                      ,default)))))
```

函数`string->type`用来将那些从查询参数和`cookie`中获取的字符串转化成你想要的类型，它是一个下列形式的广义函数：

```
(defgeneric string->type (type value))
```

为了让一个特定的名字可被用作某个查询参数的类型名，只需在`string->type`上定义一个方法。你需要至少定义一个特化在符号`string`上的方法，因为这是默认类型。当然，这很容易做到。由于浏览器有时会提交带有空字符串的表单，以表明没有值提供给某个特定的变量，你需要采用如下的方法把空字符串转化成`NIL`：

```
(defmethod string->type ((type (eql 'string)) value)
  (and (plusp (length value)) value))
```

可以为应用程序所需的其他类型添加转换方法。例如，为了使`integer`成为一个可用的查询参数类型，从而可以处理`random-page`中的`limit`参数，你可以定义下列方法：

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))
```

另一个在`param-binding`生成的代码中用到的助手函数是`get-cookie-value`，它只是由AllegroServe提供的`get-cookie-values`函数外围的一点儿语法糖：

```
(defun get-cookie-value (request name)
  (cdr (assoc name (get-cookie-values request) :test #'string=)))
```

类似地，用来计算查询参数和`cookie`名的函数也相当直接。

```
(defun symbol->query-name (sym)
  (string-downcase sym))

(defun symbol->cookie-name (function-name sym sticky)
```

```
(let ((package-name (package-name (symbol-package function-name))))
  (when sticky
    (ecase sticky
      (:global
       (string-downcase sym))
      (:package
       (format nil "~(~a:~a~)" package-name sym)))
      (:local
       (format nil "~(~a:~a:~a~)" package-name function-name sym)))))
```

为了生成那些为粘滞性参数设置cookie的代码，需要再次循环参数列表，但这一次只收集来自每个粘滞性参数的代码片段。你可以使用when和collect it这两个LOOP形式来只收集那些由set-cookie-code返回的非空值。

```
(defun set-cookies-code (function-name request params)
  (loop for param in params
        when (set-cookie-code function-name request param) collect it))

(defun set-cookie-code (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (declare (ignore type default))
    (if sticky
        `(when ,name
          (set-cookie-header
           ,request
           :name ,(symbol->cookie-name function-name name sticky)
           :value ,(princ-to-string ,name))))))
```

像这样用助手函数来定义宏的一大优点是，很容易确保生成的代码单独看起来是正确的。例如，你可以检查下面的set-cookie-code：

```
(set-cookie-code 'foo 'request '(x integer 20 :local))
```

是否生成了如下代码：

```
(WHEN X
  (SET-COOKIE-HEADER REQUEST
    :NAME "com.gigamonkeys.web:foo:x"
    :VALUE (PRINC-TO-STRING X)))
```

假设这些代码将会出现在x为变量名的某个上下文中，那么它看起来是正确的。

宏再次使你的代码直击要害，在本例中，就是你想要从请求中解出的数据和你想要生成的HTML。这就是说，该框架并不是一个大而全的Web应用框架，而只是可以让将在第29章编写的简单应用更容易一些的语法糖。

但在此之前，你还需要编写应用程序的功能性部分，相对来说第29章的应用将成为用户界面。下一章，我们要编写一个之前在第3章里编写的数据库的增强版，这一次用它来跟踪从MP3文件中解出的ID3数据。



**本**章来回顾在第3章里首次发掘的思想——从基本Lisp数据结构中构建出一个内存数据库。这次你的目标是保存那些你使用第25章的ID3v2库从一组MP3文件中解出的信息，而后在第28章和第29章里把这个数据库作为一个基于Web的流式MP3服务器的一部分。当然，这次你可以使用自从第3章以来所学到的语言特性来构建一个更专业的版本了。

## 27.1 数据库

第3章的那个数据库的主要问题是它只有一个表，也就是保存在变量`*db*`中的列表。另一个问题是代码并不清楚保存在不同字段中值的类型。在第3章里你避开了这个问题，通过使用相当通用的`EQUAL`方法来比较从数据库中选出的行的不同列的值，但如果你想要保存一些无法用`EQUAL`来比较的值，或者想要对数据库的行排序的话就会遇到麻烦了，因为不存在像`EQUAL`那样通用的比较函数。

这次你将定义一个类`table`来表示单独的数据库表，从而同时解决上述两个问题。每一个`table`实例都由两个槽构成——一个用来保存表的数据，而另一个保存关于表中各列的信息以供各种数据库操作使用。这个类如下所示：

```
(defclass table ()  
  ((rows :accessor rows :initarg :rows :initform (make-rows))  
   (schema :accessor schema :initarg :schema)))
```

和第3章一样，可以使用`plist`来表示单独的行，但这次你将创建一层抽象使得以后可以轻松调整实现细节而不会带来太多麻烦。并且这次你将把行保存在一个向量而非列表中，因为你将要支持的特定操作，比如通过数值索引对行进行随机访问，或要排序一个表，使用向量可以更高效地实现。

初始化`rows`槽的函数`make-rows`可以简单地封装在`MAKE-ARRAY`之外，从而构建一个空的可调整的带有填充指针的向量。

### 相关的包

在本章中你将用于开发代码的包如下所示：

```
(defpackage :com.gigamonkeys.mp3-database
```

```
(:use :common-lisp
      :com.gigamonkeys.pathname
      :com.gigamonkeys.macro-utilities
      :com.gigamonkeys.id3v2)
(:export  :*default-table-size*
          :*mp3-schema*
          :*mp3s*
          :column
          :column-value
          :delete-all-rows
          :delete-rows
          :do-rows
          :extract-schema
          :in
          :insert-row
          :load-database
          :make-column
          :make-schema
          :map-rows
          :matching
          :not-nullable
          :nth-row
          :random-selection
          :schema
          :select
          :shuffle-table
          :sort-rows
          :table
          :table-size
          :with-column-values))
```

其中的`:use`部分可以让你访问那些从第15章、第8章和第25章定义的包中导出的名字所对应的函数和宏，而`:export`部分导出了该库提供的API，你将在第29章里用到它们。

```
(defparameter *default-table-size* 100)

(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

为了表示表的模式 (schema)，你需要定义另一个类`column`，其每个实例都含有关于表中一个列的信息：它的名字、如何比较列中值的等价性和顺序、默认值以及在向表中插入数据或查询表时用来正则化列中值的一个函数。`schema`槽将保存一个`column`对象的列表。该类的定义如下所示：

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)

  (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)

  (comparator
```

```

:reader comparator
:initarg :comparator)

(default-value
:reader default-value
:initarg :default-value
:initform nil)

(value-normalizer
:reader value-normalizer
:initarg :value-normalizer
:initform #'(lambda (v column) (declare (ignore column)) v)))

```

column对象的equality-predicate槽和comparator槽保存着几个函数，用来比较给定列中值的等价性和顺序。这样，含有字符串值的列可以使用**STRING=**作为其equality-predicate的值，用**STRING<**作为其comparator，而当列含有数字时可以使用“=”和“<”。

default-value槽和value-normalizer槽用在向数据库中插入行时，其中value-normalizer也用在查询数据库时。当你向数据库中插入新行时，如果没有给特定的列提供值，那么就可以使用保存在column的default-value槽中的值。然后，该值（无论是默认值还是其他值）将连同列对象一起传递给保存在value-normalizer槽中的函数，从而被正则化。你传递整个列以防value-normalizer函数可能需要使用与该列对象相关联的一些数据。（你将在下一节里看到相关的例子。）在将它们与数据库中的值进行比较之前，你也应该正则化传递给查询的值。

这样，value-normalizer的职责主要是返回一个可被安全和正确地传递给equality-predicate和comparator函数的值。如果value-normalizer不能找出一个适当的返回值，那么它将报错。

向数据库中保存值之前将其正则化也是为了节省内存和CPU时钟周期。举个例子，如果你有一个即将包含字符串值的列，但将会保存在该列中的不同字符串的数量较少，例如MP3数据库中的风格列，那么你可以通过使用value-normalizer来保留这些字符串（将所有**STRING=**的值都转化成单个字符串对象）。这样，无论表中有多少行，你只需要保存相当于所有不同的值那么多的字符串，并且你可以使用**EQL**而非相对缓慢的**STRING=**来比较列中的这些值。<sup>①</sup>

## 27.2 定义模式

这样，为了生成table实例，你需要构建一个column对象的列表。你可以使用**LISP**和**MAKE-INSTANCE**来手工构建这个列表。但你很快就会注意到你经常在生成许多带有相同比较器和等价谓词组合的列对象。这是因为比较器和等价谓词的组合本质上定义了一个列类型。如果可以有一种方式，只需给出这些类型的名字就可以让你表达出给定列是字符串列，而无需每次指定**STRING<**作为其比较符和**STRING=**作为其等价谓词，那就太好了。一种方式是定义一个广义函数

<sup>①</sup> 保留对象的一般理论是，如果你打算多次比较一个特定的值，那么值得花时间先保留它。value-normalizer在你将一个值插入到表中时运行一次，然后，如同你将要看到的，它会在每个查询的开始运行一次。由于查询可能涉及对表中的每一行都调用一次equality-predicate，因此保留这些值的摊余成本将快速地收敛到零。

make-column，如下所示：

```
(defgeneric make-column (name type &optional default-value))
```

现在，你可以在这个广义函数上实现通过EQL特化符特化在type上的方法，并返回填充了适当值的column对象。下面是为类型名string和number定义列类型的广义函数和方法：

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'string<
   :equality-predicate #'string=
   :default-value default-value
   :value-normalizer #'not-nulliable))

(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'<
   :equality-predicate #'=
   :default-value default-value))
```

下面的函数not-nulliable用作string列的value-normalizer，它简单地返回给定值，除非它为NIL而会直接报错：

```
(defun not-nulliable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

这很重要，因为如果STRING<和STRING=在NIL上被调用的话将会报错。在有问题的值进入表之前将其捕捉到，比等到你要使用它们时再报错要好很多。<sup>①</sup>

你在MP3数据库中用到的另一个列类型是interned-string，如同之前讨论的那样，它的值是被保留下来的。由于你需要一个哈希表来保留这些值，你应当定义一个column的子类interned-values-column，它增加了一个槽以保存那个用来做intern操作的哈希表。

为了实现实际的保留过程，你还需要为value-normalizer提供一个:initform以传递函数用来保留该列的interned-values哈希表中的值。并且由于保留这些值的一个主要原因是允许你使用EQL作为等价谓词，你还应该为equality-predicate添加一个值为#'eq的:initform。

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eq))
```

27

<sup>①</sup> 和通常一样，编程书籍中简要讲解的最大牺牲品是正确的错误处理。在产品代码中你可能想要定义你自己的错误类型，例如下面这样的，然后在报错时使用它：

```
(error 'illegal-column-value :value value :column column)
```

接下来，你可能会考虑在哪里放置再启动以便可以从这个状况中恢复。并且在最终给定的应用中，你可以在这些再启动中建立一些状况处理器来进行选择。

```
(value-normalizer :initform #'intern-for-column))

(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-null-value column) hash)
        (setf (gethash value hash) value))))
```

然后你可以定义一个特化在名字interned-string上的make-column方法，来返回一个interned-values-column的实例。

```
(defmethod make-column (name (type (eql 'interned-string)) &optional default-value)
  (make-instance
   'interned-values-column
   :name name
   :comparator #'string<
   :default-value default-value))
```

有了这些定义在make-column上的方法，你现在可以定义一个函数make-schema，它从包括列名、列类型名以及可选默认值的列规范的列表中构建出一个column对象的列表。

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

例如，你可以为那个将用来保存从MP3中解出的数据的表定义下面这样的模式：

```
(defparameter *mp3-schema*
  (make-schema
   '(:file      string)
   (:genre     interned-string "Unknown")
   (:artist    interned-string "Unknown")
   (:album     interned-string "Unknown")
   (:song      string)
   (:track     number 0)
   (:year      number 0)
   (:id3-size  number))))
```

为了生成一个实际用来保存关于MP3的信息的表，你将\*mp3-schema\*作为:schema初始化参数传给MAKE-INSTANCE。

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

### 27.3 插入值

现在你可以开始定义你的第一个表操作insert-row了，它接受由名字和值组成的plist以及一个表，然后在表中添加含有给定值的一行。大量的工作是在一个助手函数normalize-row中完成的，它为每个列构建了一个带有默认值并正则化了的plist，并尽可能使用来自names-and-values的值，否则会使用每个列的default-value。

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows table)))

(defun normalize-row (names-and-values schema)
```

```
(loop
  for column in schema
  for name = (name column)
  for value = (or (getf names-and-values name) (default-value column))
  collect name
  collect (normalize-for-column value column)))
```

值得定义一个单独的助手函数normalize-for-column，它接受一个值和一个column对象并返回正则化的值，因为你将需要在查询参数上作同样的正则化处理。

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

现在可以将这些数据库代码与前面章节的代码组合起来，从而构建一个从MP3文件中解出的数据的数据库了。你可以定义函数file->row，使用来自ID3v2库的read-id3从一个文件中解出ID3标签并将其转化成可以传给insert-row的plist。

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
      :file (namestring (truename file))
      :genre (translated-genre id3)
      :artist (artist id3)
      :album (album id3)
      :song (song id3)
      :track (parse-track (track id3))
      :year (parse-year (year id3))
      :id3-size (size id3))))
```

insert-row可以替你处理这些事情，因此你不必担心值的正则化问题。不过，你确实需要将track和year返回的字符串值转化成数字。ID3标签中的音轨号有时被保存成音轨号的ASCII表示，有时则保存成一个数字后跟左斜杠，然后是该专辑的音轨总数。你只关心实际的音轨号，因此如果有的话，你应当使用PARSE-INTEGER的:end参数来指定它只解析到左斜杠之前的位置。<sup>①</sup>

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\: track))))
```

```
(defun parse-year (year)
  (when year (parse-integer year))))
```

最后，你可以将所有这些函数放在一起，再加上可移植路径名库的walk-directory和ID3v2库的mp3-p函数，从而定义出一个函数：它从给定目录里找出所有MP3文件并解出其中的数据，然后再把这些数据加载到数据库中。

```
(defun load-database (dir db)
  (let ((count 0))
```

<sup>①</sup> 如果任何MP3文件在音轨和年代帧里数据有格式错误，那么PARSE-INTEGER就可能会报错。处理该问题的一种方式是传给PARSE-INTEGER一个值为T的:junk-allowed参数，这将使它忽略掉跟在数字后面的任何非数字的垃圾，或是在字符串中没有数字时返回NIL。或者，如果你想试用一下状况系统，可以定义一个错误类型并在当数据的格式出现错误时在这些函数中报错，同时也建立一些再启动使这些函数得以恢复。

```
(walk-directory
  dir
  #'(lambda (file)
    (princ #\.)
    (incf count)
    (insert-row (file->row file) db))
  :test #'mp3-p)
(format t "~&Loaded ~d files into database." count)))
```

## 27.4 查询数据库

一旦你加载了带有数据的数据库，那么你需要一种方式来查询它。对于MP3应用来说，你需要一个比你在第3章里写得更加专业一些的查询函数。这一次你不仅要找出那些匹配特定条件的行，还要将结果限制在一些特定的列上，或是将结果限制在那些唯一的行上，同时还可能会在特定的列上排序这些行。为了保持关系型数据库的精髓，查询的结果将是一个含有你想要的行和列的新对象table。

你即将编写的查询函数select很大程度上出自结构化查询语言（SQL）的SELECT语句。它接受五个关键字参数：`:from`、`:columns`、`:where`、`:distinct`和`:order-by`。其中`:from`参数是你想要查询的table对象。`:column`参数指定了哪些列应当包含在结果中。其值或者是列名字的列表，或者是单独的列名，或者是默认值`T`，表示返回所有的列。如果你指定`:where`参数的话，那它应当是一个函数，其接受一行并在该行应当包含在结果中时返回真。接下来你将编写两个函数`matching`和`in`，它们可以返回适用于`:where`参数的函数。如果你指定`:order-by`参数的话，那它应当是一个列名的列表，结果将按照命名的列被排序。和`:columns`参数的情况一样，你可以只用一个名字来指定单一的列，这等价于一个含有同样名字的单元素列表。最后，`:distinct`参数是一个布尔值，它表明是否需要从结果中清除重复的行。`:distinct`的默认值为`NIL`。

下面是一些使用`select`的例子：

```
;; Select all rows where the :artist column is "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))

;; Select a sorted list of artists with songs in the genre "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

`select`和它直接用到的助手函数的实现如下所示：

```
(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))

    (when where
      (setf rows (restrict-rows rows where))))
```

```

(unless (eql columns 't)
  (setf schema (extract-schema (mklist columns) schema))
  (setf rows (project-columns rows schema)))

(when distinct
  (setf rows (distinct-rows rows schema)))

(when order-by
  (setf rows (sorted-rows rows schema (mklist order-by)))))

(make-instance 'table :rows rows :schema schema)))
(defun mklist (thing)
  (if (listp thing) thing (list thing)))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column:~a in schema:~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))

(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))

```

当然, select中真正有趣的部分在于你如何实现函数extractor、row-equality-tester和row-comparator。

通过它们的用法你可以看出, 这些函数中的每一个都必须返回函数。例如, project-columns使用由extractor返回的值作为提供给MAP的函数参数。由于project-columns的目标是返回一些仅含有特定列值的行, 你可以推断出extractor将返回接受一行作为参数的函数, 并返回一个仅含有传递的模式中指定的那些列的新行。下面是一种实现它的方式:

```

(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))


```

注意你完成这项工作的方式——在闭包主体之外从模式中解出了所有的名字: 由于闭包将被多次调用, 你会希望它在每次调用时尽可能地少做事。

函数row-equality-tester和row-comparator的实现方式很相似。为了决定两行是否等价, 你需要将用于每一列的相应等价谓词应用在适当的列值上。回顾第22章里的内容, 在一对值测试失败以后, LOOP子句always立即返回NIL, 否则将使整个LOOP返回T。

```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema)))
    #'(lambda (a b)
        (loop for name in names and test in tests
              always (funcall test (getf a name) (getf b name))))))
```

排序两行稍微复杂一些。在Lisp中，比较操作符当它们的第一个参数应该排在第二个参数的前面时返回真，否则返回假。这样，**NIL**可能意味着第二个参数应该被排在第一个参数的前面，或者说明它俩其实相等。你希望你的行比较器具有相同的行为：当第一个行排在第二个的前面时返回真，否则返回假。

这样，为了比较两个行，你应当比较用于排序的列中的值，其中采用每个列的对应比较器。首先以来自第一个行的值作为第一个参数来调用比较器。如果比较器返回真，那就意味着第一行绝对应该排在第二个行的前面，所以你可以立即返回**T**。

但如果列比较器返回了**NIL**，那么你需要检测这是因为第二个值应当排在第一个值的前面，还是因为它们相等。因此你应当以相反的参数再次调用比较器。如果这次比较器返回真，那就意味着第二个列值排在了第一个列值的前面，并且因此第二个行也应该排在第一个行的前面，所以你可以立即返回**NIL**。否则，两个列值就是等价的，那么你应当继续比较下一个列。如果你通过了所有的列而始终没有遇到某个行的值赢得了比较，那么这两行就是等价的，于是你返回**NIL**。一个实现了该算法的函数如下所示：

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names schema))))
    #'(lambda (a b)
        (loop
          for name in column-names
          for comparator in comparators
          for a-value = (getf a name)
          for b-value = (getf b name)
          when (funcall comparator a-value b-value) return t
          when (funcall comparator b-value a-value) return nil
          finally (return nil))))
```

## 27.5 匹配函数

`select`的`:where`参数可以是任何接受行对象并在该行被包括在结果中时返回真的函数。不过在实践中，你很少需要用任意代码来表达查询条件。因此你应当提供两个函数`matching`和`in`，它们将用来构造查询函数，从而允许你表达常用类型的查询，并帮你处理每个列的正确等价性谓词和值正则化器的使用。

主要的查询函数构造器是`matching`，它返回一个函数匹配带有给定列值的行。你在早先的`select`例子里看到过它的用法。例如，对`matching`的如下调用：

```
(matching *mp3s* :artist "Green Day")
```

返回一个函数匹配`:artist`值为“Green Day”的行。你也可以传递多个名字和值，当所有列都匹



配时，返回的函数才算是匹配。例如，下面的例子返回了一个匹配艺术家为“Green Day”和专辑名为“American Idiot”的行的闭包：

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

你必须将整个表对象传给`matching`，因为它需要访问表的模式，以获得它所要匹配的那些列的等价谓词和值正则化器。

你可以从较小的函数中逐步构造出`matching`返回的函数，其中每个底层函数负责匹配一个列的值。为了构造出这些函数，你需要定义函数`column-matcher`，它接受`column`对象和你要匹配的未经正则化的值，并返回接受单一行的函数，它在该行的给定列的值匹配给定值的正则化版本时返回真。

```
(defun column-matcher (column value)
  (let ((name (name column))
        (predicate (equality-predicate column))
        (normalized (normalize-for-column value column)))
    #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

然后，你可以使用函数`column-matchers`为那些你关心的名字和值构造一个列匹配函数的列表：

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
        when value collect
        (column-matcher (find-column name schema) value)))
```

现在你可以实现`matching`了。再次注意：你应尽可能多地在闭包之外做事，以确保有些事情只做一次而不用在表的每一行上都做。

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
      (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

这个函数就像一个闭包的迷宫，但值得花点儿时间来思考一下作为第一类对象的函数会给编程带来多少灵活性。

`matching`的职责是返回一个函数，它将在表的每一行上被调用来检测其是否应被包含在新表中。因此，`matching`返回带有单个参数`row`的闭包。

现在回想一下，函数`EVERY`可以接受谓词函数作为其第一个参数，并且仅当该函数应用在作为`EVERY`第二个参数传递进来的列表的每一个元素上均为真时才返回真。不过在本例中，你传递给`EVERY`的列表本身是一个由函数组成的列表，即列匹配器。你想要知道的是，每个列匹配器在你当前测试的行上被调用时，是否均返回真。因此，作为`EVERY`的谓词参数，你传递了另一个闭包给它，该闭包向列匹配器传递当前行的`FUNCALL`调用。

另一个你偶尔会觉得有用的匹配函数是`in`，它返回一个函数匹配那些特定列从给定的值集合中取值的行。你将定义`in`来接受两个参数：一个列名和一个含有你想要匹配的那些值的表。例如，假设你想要在MP3数据库中找出所有与Dixie Chicks的歌曲同名的歌曲。你可以像下面这样通过

in和一个子select写出这个where字句：<sup>①</sup>

```
(select
  :columns '(:artist :song)
  :from *mp3s*
  :where (in :song
    (select
      :columns :song
      :from *mp3s*
      :where (matching *mp3s* :artist "Dixie Chicks"))))
```

尽管查询更复杂了，但in本身的定义却比matching要简单得多。

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table)))))
    (values (map 'list #'(lambda (r) (getf r column-name)) (rows table)))
    #'(lambda (row)
        (member (getf row column-name) values :test test))))
```

## 27.6 获取结果

select返回了另一个table，因此你需要思考一下如何才能得到表中单独的行和列值。如果你确定将永不改变在表中表达数据的方式，那么就可以直接把表结构作为API的一部分，也就是说，该table中带有一个rows槽，类型为plist构成的向量，然后使用所有正常的Common Lisp函数操作向量和plist来得到表中的值。但这些表示可能确实是以后会改变的内部细节。另外，你也不希望其他代码可以直接操作这些数据结构，例如，你不希望任何人使用SETF在行中放置一个未经正则化的列值。因此，定义一些抽象来提供你想要支持的操作就会是个好主意了。如果你决定以后改变表的内部表示，就只需要改变这些函数和宏的实现。尽管Common Lisp并不能使你绝对避免人们获得“内部”数据，但通过提供一个官方API你至少可以清楚地表明边界在哪里。

你需要对查询结果做的最常见的事情，也许就是在各个行上迭代并解出特定列的值。因此，你需要提供一种方式来同时做到这两件事，而无需直接用rows向量或是GETF来获取一个行中的列值。

目前这些操作都很容易实现，它们几乎就是包装在没有这些抽象时你编写的代码之上的。你

<sup>①</sup> 这个查询也会返回所有Dixie Chicks的歌曲。如果你想把查询限制在除Dixie Chicks之外的其他艺术家的歌曲上，那么就需要一个更复杂的：where函数。由于：where参数可以是任何函数，所以这确实是可能的。你可以通过下列查询移除Dixie Chicks自己的歌曲：

```
(let* ((dixie-chicks (matching *mp3s* :artist "Dixie Chicks"))
      (same-song
        (in :song (select :columns :song :from *mp3s* :where dixie-chicks)))
      (query
        #'(lambda (row)
            (and (not (funcall dixie-chicks row)) (funcall same-song row)))))

  (select :columns '(:artist :song) :from *mp3s* :where query))
```

这样显然不是很方便。如果你打算编写一个需要做很多复杂查询的应用程序，那么你会考虑设计更加复杂的查询语言。

可以提供两种方式在一个表的行上迭代：宏`do-rows`用来提供基本的循环构造，函数`map-rows`可以构造出一个列表，含有将一个函数应用在表的每一行时所得到的结果。<sup>①</sup>

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))

(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

为了得到一行中各列的值，你应该编写一个函数`column-value`，它接受一个行和一个列名并返回对应的值。再一次，这只是对你本该自行编写的代码的简单封装。但如果你以后改变了表的内部表示，那么`column-value`的用户可以不必受到影响。

```
(defun column-value (row column-name)
  (getf row column-name))
```

尽管`column-value`对于获取列的值来说已经足矣，但你会经常想要一次性得到多个列的值。因此你可以提供一点儿语法糖，即宏`with-column-values`，它将一组变量绑定到通过适当的关键字名从一个行中解出的值上。这样，你可以将下面的写法：

```
(do-rows (row table)
  (let ((song (column-value row :song))
        (artist (column-value row :artist))
        (album (column-value row :album)))
    (format t " ~a by ~a from ~a~%" song artist album)))
```

简单替换为：

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t " ~a by ~a from ~a~%" song artist album)))
```

再一次，如果你使用第8章的`once-only`宏，实际的实现并不复杂。

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))

(defun column-bindings (vars row)
  (loop for v in vars collect `',(v (column-value ,row ,(as-keyword v)))))

(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

最后，你应当提供一种抽象方法来获取一个表中所有行的个数，并通过数值索引来访问指定行。

```
(defun table-size (table)
  (length (rows table)))
```

<sup>①</sup> 在Common Lisp被标准化以前，MIT实现的LOOP版本含有一种机制来扩展LOOP语法以支持在新数据结构上的迭代。一些从该代码树上继承了LOOP实现的Common Lisp实现可能仍然支持这一功能，从而使`do-rows`和`map-rows`变得不再是必需的了。

```
(defun nth-row (n table)
  (aref (rows table) n))
```

## 27.7 其他数据库操作

最后，我们来实现其他一些将在第29章里用到的数据库操作。前两个类似于SQL DELETE语句。函数`delete-rows`用来从一个表中删除匹配特定条件的行。和`select`一样，它接受`:from`和`:where`关键字参数。和`select`不同的是，它并不返回新表——它实际修改了作为`:from`参数传递的表。

```
(defun delete-rows (&key from where)
  (loop
    with rows = (rows from)
    with store-idx = 0
    for read-idx from 0
    for row across rows
    do (setf (aref rows read-idx) nil)
    unless (funcall where row) do
      (setf (aref rows store-idx) row)
      (incf store-idx)
    finally (setf (fill-pointer rows) store-idx)))
```

出于对效率的兴趣，你可能想要编写一个单独的函数来从表中删除所有的行。

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

其余的表操作并没有映射到正常的关系型数据库操作中，但它在MP3浏览器应用中非常有用。首先是一个在表中直接排序所有行的函数。

```
(defun sort-rows (table &rest column-names)
  (setf (rows table)
    (sort (rows table) (row-comparator column-names (schema table))))
  table)
```

另一方面，在MP3浏览器应用中，你需要一个直接在表中打乱所有的行的函数，它用到了第23章的`nshuffle-vector`。

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table))
  table)
```

最后，再一次为了MP3浏览器，你应当编写一个函数来选择 $n$ 个随机行，然后作为新表返回。它也用到了`nshuffle-vector`，另外还有一个版本的`random-sample`，后者基于我在第20章里讨论过的Donald Knuth的《计算机程序设计艺术，卷2：半数值算法（第3版）》中的算法S。

```
(defun random-selection (table n)
  (make-instance
   'table
   :schema (schema table)
   :rows (nshuffle-vector (random-sample (rows table) n))))
```

```
(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
        for idx from 0
        do
        (loop
           with to-select = (- n (length selected))
           for remaining = (- (length vector) idx)
           while (>= (* remaining (random 1.0)) to-select)
           do (incf idx))
        (vector-push (aref vector idx) selected)
        when (= (length selected) n) return selected))
```

有了这些代码，你就可以在第29章里构建一个用于浏览MP3文件集合的Web接口了。但在此之前，你还需要实现服务器中使用Shoutcast协议流式播放MP3的部分，这正是下一章的主题。

# 实践：Shoutcast服务器

# 28

**本**章我们来开发基于Web的流式MP3应用的另一个重要部分，也就是实际向诸如iTunes、XMMS<sup>①</sup>和Winamp等客户端发送MP3流的Shoutcast协议。

## 28.1 Shoutcast 协议

Shoutcast协议是由Nullsoft的人发明的，他们也是Winamp MP3软件的开发者。它被设计用来支持Internet音频广播——Shoutcast DJ从他们的个人电脑上将音频数据发送到一个中央Shoutcast服务器上，继而把它以流的形式发送到任何连网听众那里。

你即将构建的服务器实际上只能算半个真正的Shoutcast服务器，尽管你可以使用通常的Shoutcast服务器一样的协议来传送流式MP3给听众，但你的服务器将只能提供那些已经保存在服务器文件系统中的歌曲。

你只需关注Shoutcast协议的两部分：客户端开始接收一个流时所产生的请求，以及回执的格式，其中包括将当前正在播放歌曲的元数据嵌入到流中的机制。

从MP3客户端到Shoutcast服务器的初始请求被格式化成了一个正常的HTTP请求。在回应部分，Shoutcast服务器发送了ICY回执，它看起来就像是一个HTTP回执，只是字符串“ICY”<sup>②</sup>出现在正常HTTP版本字符串的位置上并带有不同的头。在发送了头和一个空行之后，服务器开始流式发送无穷尽的MP3数据。

关于Shoutcast协议唯一的难点是，正在流式发送的歌曲的元数据如何被嵌入到发送给客户端的数据中。Shoutcast设计者面临的问题是要提供一种方式让Shoutcast服务器可以在每次开始播放一首新歌时与客户端沟通新的标题信息，这样客户端就可以将其显示在UI里。（第25章讲过MP3格式本身并不提供对编码元数据的支持。）尽管ID3v2的一个设计目标是使其更适合用在流式MP3中，但Nullsoft的人们还是决定走他们自己的路线，从头发明了一种在服务器和客户端都相当容

① Red Hat 8.0和9.0以及Fedora中附带的XMMS版本已不能如何播放MP3了，这是因为Red Hat的人对MP3相关的解码器存在版权忧虑。为了在这些版本的Linux上得到带有MP3支持的XMMS，你需要从<http://www.xmms.org>上获取源代码并自行编译它。或者，对于其他可能性可以参见<http://www.fedorafaq.org/#xmms-mp3>上的信息。

② 更让人困惑的是，还存在另一个称为Icecast的流协议。看起来在Shoutcast和Icecast协议所使用的ICY头之间不存在明显的关联。

易实现的新格式。这对他们来说也是理想的，因为他们也是自己的MP3客户端的开发者。

他们的方法简单地忽略了MP3数据的结构，在每n个字节里嵌入一个分界的元数据片段。客户端有义务分离出这些元数据，使其不被视为MP3数据。由于发送到不支持该格式的客户端的元数据将导致杂音的出现，服务器仅在客户端的原始请求中包含一个特殊的Icy-Metadata头时才发送元数据。并且为了让客户端知道元数据的发送频率，服务器必须发回一个Icy-Metainit头，其值为在每个相邻的元数据片段之间发送的MP3数据的字节数。

元数据的基本内容是一个形如"StreamTitle='title';"的字符串，其中的title是当前歌曲的标题并且不能带有单引号。这一载荷采用定长的字节数组来编码：先发送一个单字节指示接下来有多少个16字节的块，然后再发送这些块。它们含有作为ASCII字符串的字符串载荷，其中最后一个块使用必要的空字节作为补白。

这样，最小的合法元数据片段是单个字节零，代表没有后续块。如果服务器不需要更新元数据，那么它可以发送这样的一个空片段，但它必须至少发送一个字节才能让客户端不会丢掉实际的MP3数据。

## 28.2 歌曲源

由于Shoutcast服务器必须在客户端连接上的情况下始终保持流式发送歌曲，你需要为服务器提供一个进行操作的歌曲来源。在基于Web的应用中，每个连接的客户端都将拥有一个可以通过Web接口管理的播放列表。但考虑到为了避免过度耦合，应当定义一个接口让Shoutcast服务器用来获得播放的歌曲。现在你可以编写一个该接口的简单实现，然后在第29章里构建一个更复杂的Web应用接口。

### 包 定 义

你将在本章里用于开发代码的包如下所示：

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

28

该接口背后的思想是，Shoutcast服务器将根据从AllegroServe请求对象中解出的ID来查找歌曲源。然后它可以对歌曲源做三件事：

- 获得歌曲源中的当前歌曲

- 告诉歌曲源当前歌曲结束
- 询问歌曲源之前给出的某个歌曲是否仍是当前歌曲

最后一个操作是必要的，因为可能存在某种方式（第29章里就会这样做）在Shoutcast服务器之外管理歌曲源。可以用下列广义函数来表达Shoutcast服务器所需的操作：

```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))

(defgeneric maybe-move-to-next-song (song source)
  (:documentation
    "If the given song is still the current one update the value
  returned by current-song."))
  
(defgeneric still-current-p (song source)
  (:documentation
    "Return true if the song given is the same as the current-song."))

```

函数`maybe-move-to-next-song`如此定义可以允许用单一操作来检查一首歌曲是否为当前歌曲，如果是的话，它就将歌曲源移向下一首歌曲。下一章里当需要实现一个可以安全地从两个不同线程来管理的歌曲源时，这种设计会很重要。<sup>①</sup>

为了表示Shoutcast服务器所需要的关于一首歌曲的信息，你可以定义一个类`song`。其槽用来保存MP3文件的名字，在Shoutcast元数据中发送的标题，以及使你在发送文件时跳过标签部分的ID3标签的大小。

```
(defclass song ()
  ((file :reader file :initarg :file)
   (title :reader title :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

由`current-song`（也就是`still-current-p`和`maybe-move-to-next-song`的第一个参数）返回的值将是一个`song`的实例。

此外，你需要定义一个广义函数，以使服务器可以基于想要的歌曲源类型和请求对象来查找一个歌曲源。其方法将特化在`type`参数上以便返回不同类型的歌曲源，并且从请求对象中将所需的任何信息取出用来检测应返回哪个源。

```
(defgeneric find-song-source (type request)
  (:documentation "Find the song-source of the given type for the given request."))

```

不过，对于本章的目标，可以使用该接口的一个简单实现，让其总是使用相同的对象，即一个可从REPL管理的歌曲对象的简单队列。一开始先定义类`simple-song-queue`和保存该类的一个实例的全局变量`*songs*`。

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer 0))
   (index :accessor index :initform 0)))
```

---

<sup>①</sup> 从技术上来讲，本章的实现也可以从两个线程来管理，即运行着Shoutcast服务器的AllegroServe线程和REPL线程。但目前你可以接受竞争状况。我将在下一章里讨论如何用锁来确保代码是线程安全的。

```
(defparameter *songs* (make-instance 'simple-song-queue))
```

然后，可以在`find-song-source`之上定义一个通过符号`singleton`上的`EQL`特化符特化在`type`上的方法，它返回保存在`*songs*`中的实例。

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

现在只需实现Shoutcast服务器将会用到的三个广义函数上的方法就可以了。

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source))))
```

```
(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))
```

```
(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

另外出于测试的目的，可以提供一种方式向队列中添加歌曲。

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))
```

```
(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
      'song
      :file (namestring (truename file))
      :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
      :id3-size (size id3))))
```

## 28.3 实现 Shoutcast

现在可以开始实现Shoutcast服务器了。由于Shoutcast协议在很大程度上是基于HTTP的，你可以将该服务器实现成AllegroServe中的一个函数。不过，由于你需要与AllegroServe的一些底层特性交互，就不能使用第26章的`define-url-function`宏。你需要编写一个像下面这样的正规函数：

28

```
(defun shoutcast (request entity)
  (with-http-response
    (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
          (request-socket request)
          (find-song-source *song-source-type* request)
          (if wants-metadata-p *metadata-interval*)))))
```

然后像下面这样将该函数发布在路径 /stream.mp3下：<sup>①</sup>

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

在with-http-response调用中，除了通常的request和entity参数以外，你还需要传递:content-type和:timeout参数。其中:content-type参数告诉AllegroServe如何设置它所发送的Content-Type头，而:timeout参数指定了AllegroServe给该函数用来生成回执的秒数。在默认情况下AllegroServe判定每个请求在五分钟后超时。由于你打算通过流发送一个本质上无穷的MP3序列，你需要更多的时间。但我们无法告诉AllegroServer一个请求“永不”超时，所以你应当将这个时间设置成\*timeout-seconds\*的值，其中你可以定义一些诸如10年的秒数这类相当大的值。

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

然后，在with-http-response的主体中导致回执头被发送的with-http-body调用之前，你需要处理AllegroServe将要发送的回执。函数prepare-icy-response封装了必要的处理：将协议字符串从默认的“HTTP”改为“ICY”并添加了Shoutcast特定的头。<sup>②</sup>为了处理iTunes中的一个bug，你还需要告诉AllegroServe不要使用分段传输编码（chunked transfer-encoding）。<sup>③</sup>函数request-reply-protocol-string、request-uri和reply-header-slot-value都是AllegroServe的一部分。

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
    `(((:|icy-metaint| ,(princ-to-string metadata-interval))
      (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
      (:|icy-notice2| "More blah")
      (:|icy-name| "MyLispShoutcastServer")
      (:|icy-genre| "Unknown")
      (:|icy-url| ,(request-uri request))
      (:|icy-pub| "1")))
    do (setf (reply-header-slot-value request k) v)))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))

(defun turn-off-chunked-transfer-encoding (request)
```

- 
- ① 当编写这些代码时，你可能还想对Lisp形式(net.aserve::debug-on :notrap)求值。这告诉AllegroServe不要捕捉由你代码所抛出的异常，从而使你在一个正常的Lisp调试器中调试它们。在SLIME中，和其他任何错误情况一样，这将会弹出一个SLIME调试器。
  - ② Shoutcast头通常以小写字母发送，因此你需要转义那些用来在AllegroServe中标识它们的关键字符的名字，从而避免让Lisp读取器将他们全部转换成大写。这样，你应当写成:|icy-metaint|而不是:icy-metaint。你还可以写成:\i\c\y-\m\e\t\a\i\n\t，但这样非常难看。
  - ③ 函数turn-off-chunked-transfer-encoding使了一点儿诡计。无法在不指定内容长度的情况下通过AllegroServe的官方API来关闭分段传输编码，因为任何声称支持HTTP/1.1的客户端都应当理解它，包括iTunes在内。但这段代码做到了。

```
(setf (request-reply-strategy request)
      (remove :chunked (request-reply-strategy request))))
```

在函数shoutcast的with-http-body中，实际流出的是MP3数据。函数play-songs接受用来发送数据的流、歌曲源以及应当使用的元数据间隔，或者在客户端不想要元数据时为NIL。该流是从请求对象中获取的socket，歌曲源通过调用find-song-source获取到，而元数据间隔则来自全局变量\*metadata-interval\*。歌曲源的类型由变量\*song-source-type\*来控制，目前你可以将其设置成singleton以使用你之前实现的simple-song-queue。

```
(defparameter *metadata-interval* (expt 2 12))

(defparameter *song-source-type* 'singleton)
```

函数play-songs本身并不做太多事。它循环调用做所有粗活的函数play-current，包括发送单个MP3文件的内容、跳过ID3标签以及嵌入ICY元数据。唯一的亮点是你需要跟踪何时发送元数据。

由于你必须以特定的间隔来发送元数据片段，而与你何时碰巧从一个MP3文件切换到下一个文件无关，在每次调用play-current时你需要告诉它下一个元数据何时到期，而当它返回时，它必须告诉你相同的事情，这样它才能将该信息传递到下一个play-current调用里。如果play-current从歌曲源里得到了NIL，那么它也返回NIL，这使play-songs中的循环得以停下来。

除了处理循环以外，play-songs还提供了一个**HANDLER-CASE**，当MP3客户端从服务器上断开并导致对socket的写入失败时，用它捕捉在play-current中抛出的错误。由于**HANDLER-CASE**在**LOOP**之外，对错误进行处理将中断循环，从而允许play-songs返回。

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
    (loop
      (for next-metadata = metadata-interval
           then (play-current
                  stream
                  song-source
                  next-metadata
                  metadata-interval)
           while next-metadata)
      (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

最终，你可以实现play-current了，它用来实际发送Shoutcast数据。基本思想是，你从歌曲源里得到当前的歌曲，打开该歌曲的文件，然后循环地从文件中读取数据并写入到socket中，直到要么遇到文件结尾，要么当前歌曲不再是当前歌曲了。

这里只有两个复杂之处：一个是需要确保在正确的间隔上发送元数据；另一个是如果文件以ID3标签开始，那么你需要跳过它。如果你不过多地考虑I/O性能，那么可以像下面这样来实现play-current：

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
```

```

(defun maybe-move-to-next-song (song song-source)
  (when song
    (let ((metadata (make-icy-metadata (title song))))
      (with-open-file (mp3 (file song) :element-type '(unsigned-byte 8))
        (unless (file-position mp3 (id3-size song))
          (error "Can't skip to position ~d in ~a" (id3-size song) (file song)))
        (loop for byte = (read-byte mp3 nil nil)
              while (and byte (still-current-p song song-source)) do
                (write-byte byte out)
                (decf next-metadata)
            when (and (zerop next-metadata) metadata-interval) do
              (write-sequence metadata out)
              (setf next-metadata metadata-interval))

        (maybe-move-to-next-song song song-source)))
      next-metadata)))

```

该函数从歌曲源中得到当前歌曲，并得到一个缓冲区，含有将要通过传递标题给make-icy-metadata来发送的元数据。接着它打开文件并使用两参数形式的FILE-POSITION跳过ID3标签。然后它开始从文件中读取字节并将它们写到请求的流中。<sup>①</sup>

当到达文件的结尾或是当歌曲源的当前歌曲发生改变时，循环就会中断。同时，无论何时next-metadata得到了零（如果允许发送元数据），那么它就将metadata写入到流中并重置next-metadata。一旦它完成了循环，就会检查歌曲是否仍是歌曲源的当前歌曲。如果是的话，那么这意味着它是因为读取了整个文件才跳出循环的，这时它会告诉歌曲源移动到下一首歌上；否则，它跳出循环是因为有人改变了当前正在播放的歌曲，那么函数就只是返回。无论是哪种情况，它都返回在下一个元数据到期前剩余的字节数，以便传给play-current的下一次调用。<sup>②</sup>

函数make-icy-metadata接受当前歌曲的标题，并生成一个含有正确格式化的ICY元数据片段的字节数组，它的实现也是相当直接的。<sup>③</sup>

```

(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a';" (substitute #\Space #\' title)))
         (blocks (ceiling (length text) 16))
         (buffer (make-array (1+ (* blocks 16))
                           :element-type '(unsigned-byte 8)
                           :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop

```

<sup>①</sup> 多数MP3播放软件都会在用户接口的某个地方显示元数据。不过，Linux上的XMMS程序默认不这样做，为了让XMMS显示Shoutcast元数据，需要按Ctrl+P来打开Preferences面板；接着在Audio I/O Plugins标签栏（在版本1.2.10下是最左边的标签）选择MPEG Layer 1/2/3 Player（libmpg123.so）并按下Configure按钮；然后选择配置窗口中的“流”标签，并在标签底部的SHOUTCAST/Icecast部分里选中“Enable SHOUTCAST/Icecast title streaming”复选框。

<sup>②</sup> 那些从Scheme迁移到Common Lisp的人们可能想知道为什么play-current不是递归地调用其自身。在Scheme中这确实工作得很好，因为Scheme实现在规范要求下必须支持“无限次活跃尾递归”（unbounded number of active tail calls）。Common Lisp实现也允许带有这一属性，但这不是语言标准要求的。因此，Common Lisp习惯上使用循环构造来编写循环，而不是递归。

<sup>③</sup> 和你编写的其他代码一样，这个函数假设你的Lisp实现的内部字符编码方式是ASCII或ASCII的一个超集，因此你可以使用CHAR-CODE将Lisp的CHARACTER对象转化成ASCII数据的字节。

```

for char across text
  for i from 1
    do (setf (aref buffer i) (char-code char)))
  buffer))

```

根据你的具体Lisp实现是如何处理它的流的，以及需要一次服务多少个MP3客户端，这个简单版本的play-current可能足够高效，也可能不是。

这个简单实现的潜在问题是，你不得不为你传输的每个字节调用**READ-BYTE**和**WRITE-BYTE**。有可能每个调用都产生成本相对高昂的系统调用来读写一个字节。即便Lisp实现了自己的带有内部缓冲区的流，从而并非每个**READ-BYTE**和**WRITE-BYTE**都产生系统调用，函数调用本身的成本也仍然存在。特别是在使用所谓的Gray Streams提供用户可扩展流的实现里，**READ-BYTE**和**WRITE-BYTE**可能导致对广义函数的调用，该函数在底层派发到流参数的类上。虽说广义函数派发通常足够高效让你不必担心，但它还是比非广义的函数调用成本更高一些，如果能够避免的话，你绝不想在几分钟里对其调用数百万次。

实现play-current的一种更高效但也更复杂的方式是使用**READ-SEQUENCE**和**WRITE-SEQUENCE**来一次性读写多个字节。你也给自己一个机会将文件读取操作与文件系统的自然块大小相匹配，从而为你带来最佳的磁盘吞吐量。当然，无论你使用多大的缓冲区，跟踪何时发送元数据都将变得更复杂一些。一个使用了**READ-SEQUENCE**和**WRITE-SEQUENCE**的更高效的play-current版本如下所示：

```

(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song)))
            (buffer (make-array *block-size*:element-type '(unsigned-byte 8))))
        (with-open-file (mp3 (file song)) :element-type '(unsigned-byte 8))
          (labels ((write-buffer (start end)
                     (if metadata-interval
                         (write-buffer-with-metadata start end)
                         (write-sequence buffer out :start start :end end)))
                  (write-buffer-with-metadata (start end)
                    (cond
                      ((> next-metadata (- end start))
                       (write-sequence buffer out :start start :end end)
                       (decf next-metadata (- end start)))
                      (t
                        (let ((middle (+ start next-metadata)))
                          (write-sequence buffer out :start start :end middle)
                          (write-sequence metadata out)
                          (setf next-metadata metadata-interval)
                          (write-buffer-with-metadata middle end)))))

                  (multiple-value-bind (skip-blocks skip-bytes)
                      (floor (id3-size song) (length buffer))

                    (unless (file-position mp3 (* skip-blocks (length buffer)))
                      (error "Couldn't skip over ~d ~d byte blocks.")))

```

```
skip-blocks (length buffer)))  
  
(loop for end = (read-sequence buffer mp3)  
      for start = skip-bytes then 0  
      do (write-buffer start end)  
      while (and (= end (length buffer))  
                 (still-current-p song song-source)))  
  
(maybe-move-to-next-song song song-source))))  
next-metadata)))
```

现在你可以用所有这些东西来做点什么了。在下一章里，你将编写一个本章所开发的Shoutcast服务器的Web接口，它使用第27章的MP3数据库作为歌曲源。



**构**建MP3流应用的最后一步是编写一个Web接口，从而允许用户查找他们想听的歌曲，并且将它们添加到一个播放列表中，这样当用户的MP3客户端请求该流的URL时，Shoutcast服务器将会播放指定的歌曲。为了开发应用的这个组件，你需要把一些前面几章的代码整合起来：MP3数据库、第26章的define-url-function宏，当然还有Shoutcast服务器本身。

## 29.1 播放列表

接口背后的基本思想是每个MP3客户端连接到Shoutcast服务器上，获取它们自己的播放列表(playlist)，将其作为Shoutcast服务器所需的歌曲源。播放列表还将提供超出Shoutcast服务器需要之外的功能：用户将通过Web接口来向播放列表中添加歌曲，删除已在播放列表中的歌曲以及通过排序和乱序来重新调整播放列表。

你可以像下面这样来定义一个表示播放列表的类：

```
(defclass playlist ()
  ((id :accessor id :initarg :id)
   (songs-table :accessor songs-table :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx :accessor current-idx :initform 0)
   (ordering :accessor ordering :initform :album)
   (shuffle :accessor shuffle :initform :none)
   (repeat :accessor repeat :initform :none)
   (user-agent :accessor user-agent :initform "Unknown")
   (lock :reader lock :initform (make-process-lock))))
```

播放列表的id是其关键字，你从请求对象中解出它并传递给find-song-source来查询一个播放列表。你实际上并不需要将其保存在playlist对象中，但如果可以从一个任意的播放列表对象中找出它的id是什么的话，这会使调试更加方便。

播放列表的核心是songs-table槽，它用来保存一个table对象。用于这个表的模式将和用于主MP3数据库的模式相同。用来初始化songs-table的函数make-playlist-table十分简单：

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

## 包 定 义

可以使用下列DEFPACKAGE来定义用于本章中代码的包：

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

由于这是一个高阶应用，它用到了许多底层包。它还从ACL-SOCKET包中导入了三个符号，并从MULTIPROCESSING包中导入了其余两个，这是因为它只需要这两个包中导出的5个符号而不需要其他的139个符号。

通过将歌曲的列表保存在一个表中，可以使用第27章的数据库函数来操作播放列表：你可以用insert-row向播放列表中添加歌曲，用delete-rows删除歌曲以及用sort-rows和shuffle-table重排播放列表。

current-song和current-idx槽用来跟踪当前正在播放哪首歌曲：current-song是实际的song对象，而current-idx是song-table中代表当前歌曲的行的索引。你将在29.3节中看到如何在每当current-idx改变时确保更新current-song。

ordering和shuffle槽保存关于songs-table中的歌曲顺序的信息。其中ordering槽保存一个关键字来告诉songs-table在其不是乱序时应当怎样排序。合法的值包括:genre、:artist、:album和:song。shuffle槽保存下列关键字之一：:none、:song或:album。它指定了songs-table应当如何被乱序。

repeat槽也保存一个关键字，:none、:song或:all之一，指定了播放列表的重复模式。如果:repeat是:none，那么在songs-table的最后一首歌播放完以后，current-song回滚到一个默认的MP3上；当:repeat为:song时，播放列表将不断地返回到相同的current-song上；而如果是:all的话，在最后一首歌结束以后current-song将回到第一首歌上。

user-agent槽保存MP3客户端在其对流的请求中发送的User-Agent头。你纯粹是为了Web接口才保存这个值的——User-Agent头标识了产生请求的程序，因此可以将该值显示在列出所有播放列表的页面上，从而容易看出当有多个用户连接时播放列表与连接的对应关系。

最后，lock槽中保存了一个由函数make-process-lock创建的“进程锁”，该函数是Allegro的MULTIPROCESSING包的一部分。你将需要在操作playlist对象的特定函数中用到这个锁，以

确保每次只有一个线程在操作给定的播放列表对象。可以定义下面的宏来包装一组需要在保持一个播放列表锁的情况下进行处理的代码，该宏是来自MULTIPROCESSING的with-process-lock宏构建出来的：

```
(defmacro with-playlist-locked ((playlist) &body body)
  ` (with-process-lock ((lock ,playlist))
      ,@body))
```

其中的with-process-lock宏要求获得对给定进程锁的排他访问，然后再执行其主体Lisp形式，最后再释放锁。在默认情况下with-process-lock允许递归加锁，这意味着同一个线程可以安全地对同一个锁对象加锁多次。

## 29.2 作为歌曲源的播放列表

为了将playlist用作Shoutcast服务器的歌曲源，需要在第28章的广义函数find-song-source上实现一个方法。由于你打算拥有多个播放列表，需要一种方式来为连接到服务器的每个客户端找出那个正确的播放列表来。具体的映射部分很简单——你可以定义一个变量来保存EQUAL哈希表，并用它来将一些标识符映射到playlist对象上。

```
(defvar *playlists* (make-hash-table :test #'equal))
```

你还需要定义一个进程锁来保护对这个哈希表的访问，如下所示：

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

然后定义一个函数根据给定ID来查询一个播放列表，如果必要的话就创建一个新的playlist对象，并用with-process-lock来确保每次只有一个线程在操作哈希表。<sup>①</sup>

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

然后你就可以在该函数和另一个函数playlist-id的基础上实现find-song-source了，它接受AllegroServe请求对象并返回适当的播放列表标识符。find-song-source函数也负责从请求对象中抓取User-Agent字符串并保存在播放列表对象中。

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent))))
```

29

<sup>①</sup> 并发编程的复杂度超出了本书的讨论范围。基本的思想是，如果你有多个控制线程，就像在当前的应用里这样，一些线程运行shoutcast函数而另一些线程回应浏览器的请求，那么你需要确保每次只有一个线程在操作给定的某个对象，以避免当一个线程工作在该对象时另一个线程看到了不一致的状态。例如，在当前这个函数中，如果两个新的MP3客户端正在同时连接，它们都试图添加一项到\*playlists\*中，那么这有可能互相影响。with-process-lock确保了每个线程都可以获得对哈希表的排他访问，以便有足够长的时间来完成它们想做的事。

```
(when user-agent (setf (user-agent playlist) user-agent)))
playlist))
```

接下来的难点是如何实现`playlist-id`，即一个从请求对象中解出标识符的函数。你有很多选项，每个分别对应于不同的用户接口实现。你可以从请求对象中取得任何想要的信息，但无论你决定怎样识别一个客户端，都需要一些方式来让Web接口的用户与正确的播放列表关联在一起。

目前你可以采用一个“勉强可用”的方法，这要求每台连接到服务器的机器上只有一个MP3客户端，并且浏览Web接口的用户就来自运行着MP3客户端的那台机器：你将使用客户机的IP地址作为标识符。通过这种方式，可以为一个请求找出正确的播放列表，而不论请求是来自MP3客户端还是一个Web浏览器。尽管如此，你将在Web接口中提供一种方式来从浏览器中选择一个不同的播放列表，因此这一选择在应用上施加的实际约束是每个客户端IP地址上只能有一个连接的MP3客户端。<sup>①</sup> `playlist-id`的实现如下所示：

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

函数`request-socket`是AllegroServe的一部分，而`remote-host`和`ipaddr-to-dotted`都是Allegro的socket库的一部分。

为了创建可被Shoutcast服务器用作歌曲源的播放列表，需要在`current-song`、`still-current-p`和`maybe-move-to-next-song`上定义将`source`参数特化在`playlist`上的方法。`current-song`方法已经准备好了：通过在`current-song`槽上定义同名的访问函数，会自动地获得了特化在`playlist`上的可返回该槽的值的`current-song`方法。不过，为了使对`playlist`的访问是线程安全的，需要在访问`current-song`槽之前锁定该`playlist`。在本例中，最简单的方法是像下面这样定义一个`:around`方法：

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

实现`still-current-p`也很简单，假设你确保只有在当前的歌曲实际发生了改变时`current-song`才被更新到新的`song`对象上。你再次需要获取一个进程锁以确保可以对`playlist`的状态得到一致的视图。

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

剩下的难点是确保`current-song`槽在正确的时间得到更新。当前的歌曲有几种改变的方式，最明显的一种是当Shoutcast服务器调用`maybe-move-to-next-song`时。但它还可以在歌曲被添加到播放列表时更新，比如当Shoutcast服务器播完了所有歌曲，或者甚至是在播放列表的重复模式被改变时。

---

<sup>①</sup> 这种方法也假设了每个客户机都有独立的IP地址。这个假设在所有用户都在同一个LAN下是成立的，但如果用户是来自一个做网络地址转换的防火墙之后的话就不成立了，如果你想要将这个应用部署在更广的因特网上的话，最好对网络有足够的理解从而找出最适合自己的方法。

与其试图编写特定于上述每种情形的代码来检测是否更新current-song，不如定义一个函数update-current-if-necessary，在current-song中的song对象不再匹配current-idx槽对应的当前应播放文件时，它会更新current-song。然后，如果进行了可能导致这两个槽不同的播放列表操作之后再调用该函数，你就可以确保current-song总是被正确设置了。下面是update-current-if-necessary和它的助手函数：

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                 (file-for-current-idx playlist))
    (reset-current-song playlist)))

(defun file-for-current-idx (playlist)
  (if (at-end-p playlist)
      nil
      (column-value (nth-row (current-idx playlist) (songs-table playlist)) :file)))

(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

你不需要为这些函数加锁，因为它们将只在那些预先加锁过播放列表的函数中调用。

函数reset-current-song引入了又一个亮点：由于想要播放列表对客户端提供无穷的MP3流，你不希望current-song被设置成NIL。相反，当一个播放列表没有歌曲可播时，即当songs-table为空或是当repeat设置成:none时最后一首歌已经播完了，你需要将current-song设定在一个其文件为MP3静音<sup>①</sup>的特殊歌曲上，并且其标题要能够解释为何没有歌曲在播放。下面的一些代码定义了两个参数，\*empty-playlist-song\*和\*end-of-playlist-song\*，每个都被设置成一个以\*silence-mp3\*所命名的文件作为文件并带有适当标题的歌曲：

```
(defparameter *silence-mp3* ...)

(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
   'song
   :file file
   :title title
   :id3-size (if (id3-p file) (size (read-id3 file)) 0)))

(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))
(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist."))
```

reset-current-song会在current-idx没有指向songs-table的任何一行时使用这些参数。否则，它会将current-song设置成代表当前行的一个song对象。

29

<sup>①</sup> 不幸的是，由于MP3格式的授权问题，我不太清楚在没有向Fraunhofer IIS付费的情况下提供一个这样的MP3文件是否合法。我的这个MP3来自Slim Devices的Slimp3的配套软件的一部分。你可以通过访问[http://svn.slimdevices.com/\\*checkout\\*/trunk/server/HTML/EN/html/silentpacket.mp3?rev=2](http://svn.slimdevices.com/*checkout*/trunk/server/HTML/EN/html/silentpacket.mp3?rev=2)。从他们的Subversion库中获得它。或者购买一个Squeezebox，即Slimp3的新的无线版本，然后作为随机软件的一部分，你将得到silentpacket.mp3。或者你还可以查找John Cage的一个长度为4'33"的MP3文件。

```
(defun reset-current-song (playlist)
  (setf
    (current-song playlist)
    (cond
      ((empty-p playlist) *empty-playlist-song*)
      ((at-end-p playlist) *end-of-playlist-song*)
      (t (row->song (nth-row (current-idx playlist)) (songs-table playlist))))))

(defun row->song (song-db-entry)
  (with-column-values (file song artist album id3-size) song-db-entry
    (make-instance
      'song
      :file file
      :title (format nil "~a by ~a from ~a" song artist album)
      :id3-size id3-size)))

(defun empty-p (playlist)
  (zerop (table-size (songs-table playlist)))))


```

现在，你可以实现`maybe-move-next-song`上的方法了，基于播放列表的重复模式将`current-idx`移到其下一个值上，然后调用`update-current-if-necessary`。当`current-idx`已在播放列表的结尾时，你不需要改变`current-idx`，因为你希望它保持在当前值上，这样它就可以指向你添加到播放列表的下一首歌。这个函数必须在操作播放列表前锁定它，因为它是由Shoutcast服务器代码调用的，而后者并没有做任何锁定。

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all (setf (current-idx playlist)
                      (mod (1+ (current-idx playlist))
                           (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist))))
```

## 29.3 操作播放列表

播放列表代码的其余部分被Web接口用来操作`playlist`对象，包括添加和删除歌曲、排序和乱序以及设置重复模式。和上一节的那些助手函数一样，你不需要在这些函数中担心锁定问题，因为你将要看到，锁将被调用它们的Web接口函数获取。

添加和删除基本上是一个`songs-table`的管理问题。你唯一需要做的额外工作是保持`current-song`和`current-idx`同步。例如，无论何时播放列表为空，它的`current-idx`都将是零，而`current-song`将是`*empty-playlist-song*`。如果你向空的播放列表里添加一首歌曲，那么这个零索引将在范围内，因此你应该将`current-song`改变成新添加的歌曲。同样的情况，当你已经播放完一个播放列表中的所有歌曲且`current-song`为`*end-of-playlist-song*`

时，添加一首歌会导致current-song被重置。所有这些实际上意味着，你需要在适当时机调用update-current-if-necessary。

Web接口沟通所需添加歌曲的方式，使播放列表添加歌曲的过程有点儿复杂。我由于下一节里讨论的一些原因，Web接口代码无法只是给你一些简单的判定规则来从数据库中选择歌曲，而是给你一个列的名字和一个值的列表，然后让你从主数据库中添加给定列具有值列表中某个值的所有歌曲。这样，为了添加正确的歌曲，你需要首先构造一个含有你想要的值的表对象，然后将它和歌曲数据库上的in查询一起使用。因此，add-songs看起来像下面这样：

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                'table
                :schema (extract-schema (list column-name) (schema *mp3s*)))))
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist)))
    (update-current-if-necessary playlist)))
```

删除歌曲会简单一些。你只需从songs-table中删除匹配特定条件的歌曲——无论是一个特定歌曲还是一个特定风格、特定艺术家或来自特定专辑的所有歌曲。因此，你可以编写一个delete-song函数，接受一些键值对，用来构造一个你可以传给delete-rows数据库函数的基于matching的:where子句。

当你删除歌曲时会出现的另一个复杂之处是current-idx可能需要改变。假设当前歌曲并非刚刚删除的歌曲之一，那么你希望它仍旧是当前歌曲。但如果在songs-table中在它之前的歌曲被删除，在删除以后它将处在表中的一个不同的位置上。因此在delete-rows调用之后，需要查看含有当前歌曲的行并重设current-idx。如果当前歌曲本身被删除了，在没有其他法子时，你可以将current-idx重设到零。在更新了current-idx之后，调用update-current-if-necessary将会处理current-song的更新。而如果current-idx改变了却仍然指向了同一首歌，那么current-song将保持不变。

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))

(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
         (matcher (matching table :file (file (current-song playlist)))))
    (pos 0))
  (do-rows (row table)
    (when (funcall matcher row)
      (return-from position-of-current pos)))
  (incf pos)))
```

你还可以编写函数来完全清空播放列表，它使用delete-all-rows并且不再需要查找当前歌

曲，因为当前歌曲明显也要被删除。对update-current-if-necessary的调用将使得current-song设置到empty-playlist-song上。

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

排序和乱序播放列表是彼此相关的操作，因为播放列表总是要么排序的要么乱序的。shuffle槽表明播放列表是否应当被乱序，以及如果是的话该怎样做。如果它被设置为:none，那么播放列表将按照ordering槽的值来排序。当shuffle是:song时，播放列表将被随机地调整顺序。而当它被设置成:album时，专辑的列表将被随机调整顺序，但每个专辑中的歌曲仍然以音轨的顺序列出。这样当用户选择一个新的顺序时，Web接口代码调用的sort-playlist函数，就需要在调用实际完成排序工作的order-playlist之前，将ordering设置成你想要的顺序，而将shuffle设置成:none。和delete-songs里的情况一样，你需要使用position-of-current来重设current-idx到当前歌曲的新位置。不过，这时你不需要调用update-current-if-necessary，因为你知道当前歌曲仍然在表中。

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

在order-playlist中，你可以使用数据库函数sort-rows来实际进行排序，基于ordering的值传递一个列的列表来进行排序。

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist))
  (case (ordering playlist)
    (:genre '(:genre :album :track))
    (:artist '(:artist :album :track))
    (:album '(:album :track))
    (:song '(:song))))
```

当用户选择一个新的乱序模式时，Web接口代码调用的函数shuffle-playlist以类似的方式工作，只是它不需要改变ordering的值。这样，当使用:none的shuffle来调用shuffle-playlist时，播放列表将根据最近一次的排序状态进行排序。按歌曲乱序比较简单——只需在songs-table上调用shuffle-table。按专辑乱序稍微复杂一些，但也没什么大不了的。

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))

(defun shuffle-by-song (playlist)
```

```
(shuffle-table (songs-table playlist)))

(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist (column-value album-row :album)))
        (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))

(defun shuffled-album-names (playlist)
  (shuffle-table
    (select
      :columns :album
      :from (songs-table playlist)
      :distinct t)))

(defun songs-for-album (playlist album)
  (select
    :from (songs-table playlist)
    :where (matching (songs-table playlist) :album album)
    :order-by :track))
```

你需要支持的最后一个操作是设置播放列表的重复模式。多数时候，在设置repeat时不需做任何额外的操作，它的值只在maybe-move-to-next-song中用到。不过，你需要在一种情况下作为改变repeat的结果来更新current-song，即当current-idx位于一个非空播放列表的结尾，同时repeat从:song改变成:all。在这种情况下，你希望可以继续播放，要么重复最后一首歌曲，要么从播放列表的起始处开始。因此，你应该在广义函数(setf repeat)上定义一个:after方法。

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
    (ecase value
      (:song (setf (current-idx playlist) (1- (table-size (songs-table playlist))))))
      (:none)
      (:all (setf (current-idx playlist) 0)))
    (update-current-if-necessary playlist)))
```

现在有了你需要的所有底层支持。其余的代码将只是提供一个基于Web的用户接口来浏览MP3数据库和操作播放列表了。这个接口将由3个通过define-url-function定义的主函数构成：一个用于浏览歌曲数据库，一个用于查看和管理单个播放列表，最后一个用来列出所有可用的播放列表。

但在开始编写这三个函数之前，你还需要先写出它们将用到的一些助手函数和HTML宏。

29

## 29.4 查询参数类型

由于将使用define-url-function，你需要在第28章的string->type广义函数上定义一些方法，使得define-url-function可以用来将字符串查询参数转化成Lisp对象。在当前的应用下，你将需要一些方法来将字符串分别转化成整数、关键字符以及一个值的列表。

前两个方法很简单。

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))

(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

最后一个string->type方法稍微复杂一些。出于我即将谈到的一些原因，你会需要生成页面来显示表单，其中含有一个隐含字段，其值是一个字符串的列表。由于你要负责生成这个隐含字段中的值，并且在它提交回来以后还要解析它，因此可以使用任何你认为方便的编码方式。你可以使用函数WRITE-TO-STRING和READ-FROM-STRING，它们使用Lisp的打印机和读取器向字符串中写入数据，以及从字符串中读取数据，除非字符串的打印表示中可能含有引号，和其他在嵌入到一个INPUT元素的值属性时可能带来问题的字符。因此，你需要以某种方式转义这些字符。与其试图引入你自己的转义方法，不如直接使用Base 64，它通常是一种用来保护通过电子邮件发送的二进制数据的编码方式。AllegroServe带有两个函数base64-encode和base64-decode，它们可以为你做Base 64的编码和解码，因此你要做的就只是编写一对函数：一个用来编码Lisp对象，先用WRITE-TO-STRING将其转化成可读的字符串，然后再对其进行Base 64编码；另一个用来从上述编码的字符串中进行base 64解码，然后再把结果传给READ-FROM-STRING。你需要把对WRITE-TO-STRING和READ-FROM-STRING的调用包装在WITH-STANDARD-IO-SYNTAX中，以确保所有可能影响打印机和读取器的变量都被设置在它们的标准值上。不过，由于你打算读取来自网络的数据，必然希望关掉读取器的一个特性，即不需要在读取过程中对任意Lisp代码求值！<sup>①</sup>可以定义你自己的宏with-safe-io-syntax，它将其主体Lisp形式包装在一个将\*READ-EVAL\*绑定到NIL的LET外围的WITH-STANDARD-IO-SYNTAX里。

```
(defmacro with-safe-io-syntax (&body body)
  ` (with-standard-io-syntax
      (let ((*read-eval* nil))
        ,@body)))
```

然后编码和解码函数就很容易写了。

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj)))))

(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

最终，你可以使用这些函数来定义string->type上的方法，为查询参数类型base64-list 定义转换方法。

---

<sup>①</sup> 读取器支持一种语法“#.”，它使接下来的S-表达式在读取期被求值。这在源代码中偶尔会有用，但显然会在你读取不可信任的数据时打开了一个巨大的安全漏洞。不过，你可以通过将\*READ-EVAL\*设置为NIL来关闭该语法，这样以来读取器在遇到“#.”时就会报错。

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

## 29.5 样板 HTML

接下来需要定义一个HTML宏和助手函数，以便让应用中的不同页面获得一致的外观。可以从一个定义了应用中页面基本结构的HTML宏开始。

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
    (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
    (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

出于两个理由，应该将standard-header和standard-footer定义成单独的函数。首先，在开发过程中可以重定义这些函数并立即观察其效果，而不需要重新编译那些使用了:mp3-browser-page宏的函数。其次，可以看出你以后编写的某个页面将不会使用:mp3-browser-page，但却可能仍然需要标准的页头和页脚。该宏如下所示：

```
(defparameter *r* 25)

(defun standard-header ()
  (html
    ((:p :class "toolbar")
     "[" (:a :href (link "/browse" :what "genre") "All genres") "] "
     "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "] "
     "[" (:a :href (link "/browse" :what "artist") "All artists") "] "
     "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists") "] "
     "[" (:a :href (link "/browse" :what "album") "All albums") "] "
     "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "] "
     "[" (:a :href (link "/browse" :what "song" :random *r*) "Random songs") "] "
     "[" (:a :href (link "/playlist") "Playlist") "] "
     "[" (:a :href (link "/all-playlists") "All playlists") "]")))
))

(defun standard-footer ()
  (html
    (:hr)
    ((:p :class "footer") "MP3 Browser v" *major-version* "." *minor-version*)))
```

一些较小的HTML宏和助手函数自动化了其他一些常用的模式。HTML宏:table-row可以让生成HTML中表的单行更加容易。它使用了FOO的一个特性（我将在第31章里提到），即一个&attributes参数，它使任何收集到一个列表中并绑定到&attributes参数上的属性可被宏作为正常S-表达式HTML形式来解析。它看起来像下面这样：



```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

另一个link函数用来生成可用作A元素的HREF属性的应用内部URL，它可从一组键值对中构造出一个查询字符串，并确保所有的特殊字符都被正确地转义。例如，你可以将下面的写法

```
(:a :href "browse?what=artist&genre=Rhythm%26Blues" "Artists")
```

替换为

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

该函数如下所示：

```
(defun link (target &rest attributes)
  (html
    (:attribute
      (:format "~a~@[?~{(~(~a~)=~a~^&~}~]" target (mapcar #'urlencode attributes)))))
```

为了编码用于URL的键和值，你用到了助手函数urlencode，这是一个包装在函数encode-form-urlencoded上的函数，函数encode-form-urlencoded是来自AllegroServe的一个非公开的函数。一方面，这种做法并不是很好。由于名字encode-form-urlencoded并非是NET.ASERVE导出的名字，encode-form-urlencoded将来有可能消失或在你不知道的情况下被重命名。另一方面，使用这个没有导出的函数可以让你立刻完成手头的工作。通过将encode-form-urlencoded封装在你自己的函数中，就将有风险的代码隔离在了一个函数里，将来如果需要的话还可以重写它。

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

最后，你需要:mp3-browser-page用到的CSS样式表mp3-browser.css。由于它并非是动态的，最简单的方法就是用publish-file发布一个静态文件。

```
(publish-file :path "/mp3-browser.css" :file filename :content-type "text/css")
```

一个示例样式表在本书Web站点上与本章配套的源代码放在了一起。你将在本章结尾处定义一个函数来启动这个MP3浏览器应用。它将在完成其他工作的同时顺便发布这个文件。

## 29.6 浏览页

第一个URL函数将生成一个用来浏览MP3数据库的页面。查询参数将告诉它用户正在浏览什么类型的东西，并提供他们感兴趣的数据库元素的查询条件。它将给你一种方式来查询匹配一个特定风格、艺术家或专辑的数据库项。为了增加奇遇的可能性，你还可以提供一种方式来选择匹配项的一个随机子集。当用户在单个歌曲的层面上浏览时，歌曲的标题是一个添加该歌曲到播放列表的链接。否则，每个项都带有链接，可以让用户浏览其他分类所列出的项。例如，如果用户正在浏览风格，其中的项“Blues”包含的链接可浏览所有带有Blues风格的专辑、艺术家和歌曲。而且，浏览页面里还带有一个“Add all”按钮，可将匹配页面中所给条件的每一首歌曲都添加到该用户的播放列表中。该函数如下所示：

```
(define-url-function browse
  (request (what keyword :genre) genre artist album (random integer))

  (let* ((values (values-for-page what genre artist album random))
         (title (browse-page-title what random genre artist album))
         (single-column (if (eql what :song) :file what))
         (values-string (values->base-64 single-column values)))
    (html
      (:mp3-browser-page
        (:title title)
        (:form :method "POST" :action "playlist")
        (:input :name "values" :type "hidden" :value values-string)
        (:input :name "what" :type "hidden" :value single-column)
        (:input :name "action" :type "hidden" :value :add-songs)
        (:input :name "submit" :type "submit" :value "Add all"))
      (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

这个函数首先使用函数values-for-page来获得一个含有它需要表示的值的列表。当用户按歌曲来浏览时，这时what参数为:song，你需要从数据库中选择完成的行。但是当用户按风格、艺术家或专辑名来浏览时，你将只想选择给定分类中不同的值。数据库函数select完成了几乎所有的重活儿，其中values-for-page多数时候负责根据what的值来向select传递正确的参数。这也是在必要时你选择匹配行的一个随机子集的地方。

```
(defun values-for-page (what genre artist album random)
  (let ((values
          (select
            :from *mp3s*
            :columns (if (eql what :song) t what)
            :where (matching *mp3s* :genre genre :artist artist :album album)
            :distinct (not (eql what :song))
            :order-by (if (eql what :song) '(:album :track) what))))
    (if random (random-selection values random) values)))
```

为了生成浏览页的标题，可以将浏览条件传递给下列函数browse-page-title：

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a" artist))
      (when album (format s " on album ~a" album)))))
```

29

一旦有了想要表示的那些值，就需要对它们做两件事。当然，主要的任务是将它们表示出来，这是由do-rows循环来做的，然后把每行的渲染工作交给list-item-for-page。该函数以一种方式来渲染用于:song的行，而用另一种方式来渲染其他类型的行。

```
(defun list-item-for-page (what row)
  (if (eql what :song)
    (with-column-values (song file album artist genre) row
```

```
(html
  (:li
    (:a :href (link "playlist" :file file :action "add-songs") (:b song))
    " from "
    (:a :href (link "browse" :what :song :album album) album)
    " by "
    (:a :href (link "browse" :what :song :artist artist) artist)
    " in genre "
    (:a :href (link "browse" :what :song :genre genre) genre)))
  (let ((value (column-value row what)))
    (html
      (:li value " - "
        (browse-link :genre what value)
        (browse-link :artist what value)
        (browse-link :album what value)
        (browse-link :song what value))))))

(defun browse-link (new-what what value)
  (unless (eql new-what what)
    (html
      "["
      (:a :href (link "browse" :what new-what what value)
        (:format "~(~as~)" new-what))
      "]"))))
```

在browse页上你要做的另一件事是，编写一个带有几个隐含INPUT字段的表单和一个“Add all”提交按钮。你需要使用HTML表单而不是正常的链接来确保应用的无状态性，从而确保拥有所需的信息来回应一个来自该请求本身的需求。由于浏览页中的结果可能是部分随机的，因此你需要向服务器提交相当多的数据才能重构添加到播放列表的歌曲列表。如果你没有允许浏览页返回随机结果的话，就不需要太多的数据，你只需提交一个添加歌曲的请求，采用浏览页使用的任何搜索条件均可。但如果你以这种方式提交一个含有random参数的条件来添加歌曲的话，那么最终你所添加的歌曲将与用户点击“Add all”按钮时在页面中看到的歌曲属于不同的随机集合。

你将使用的解决方案是发回一个含有足够多信息的表单，其中带有一个隐含的INPUT元素，允许服务器可以重构匹配浏览页条件的歌曲列表。该信息就是由values-for-page所返回的值列表以及what参数的值。这就是你用到base64-list参数类型的地方。函数values->base64从values-for-page返回的表中将一个指定列的值解出来并放在一个列表中，然后再从该列表中生成一个base 64编码的字符串嵌入到表单里。

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

当该参数以values查询参数的形式回到一个将values声明为类型base-64-list的URL函数中时，它将被自动转换回一个列表。后面你将很快看到，该列表可被用来构造返回正确

的歌曲列表的查询。<sup>①</sup>当你正在按:song浏览时，你使用来自:file列的值，因为它们可以唯一地识别实际的歌曲，而通过歌曲名可能不行。

## 29.7 播放列表

本节将我们带到了下一个URL函数playlist。这是三个页面中最复杂的一个，它负责显示用户播放列表的当前内容，同时提供操作播放列表的接口。但在大部分繁文缛节都由define-url-function处理的背景下，不难看出函数playlist是如何工作的。下面是其定义的开始部分，只给出了参数列表：

```
(define-url-function playlist
  (request
    (playlist-id string (playlist-id request) :package)
    (action keyword) ; Playlist manipulation action
    (what keyword :file) ; for :add-songs action
    (values base-64-list) ;
    file ; for :add-songs and :delete-songs actions
    genre ; for :delete-songs action
    artist ;
    album ;
    (order-by keyword) ; for :sort action
    (shuffle keyword) ; for :shuffle action
    (repeat keyword)) ; for :set-repeat action
```

除了强制出现的request参数以外，playlist还接受大量的查询参数。从某种程度来讲最重要的是playlist-id，它标识了页面应显示和管理的那个playlist对象。对于这个参数，你可以利用define-url-function的“粘滞参数”特性。正常情况下playlist-id无需显式提供，默认为playlist-id函数所返回的值，也就是浏览器所在客户机的IP地址。不过，通过允许显式地指定该值，用户也可以从未运行他们的MP3客户端的其他机器上管理其播放列表。并且如果该参数被指定过一次，那么define-url-function将通过在浏览器中设置一个cookie来使其成为“粘滞的”。随后你将定义一个URL函数来生成全部已有播放列表的列表，其中用户可以选取一个与他们正在机器上浏览的不同的播放列表。

参数action指定了一些在用户的播放列表对象上所做的操作。该参数的值将会自动地转化成一个关键字符，包括：:add-songs、:delete-songs、:clear、:sort、:shuffle或:set-repeat。其中:add-songs操作用于浏览页中的“Add all”按钮，也用于那些用来添加单独歌曲的链接。其他的操作都用于播放列表页面本身的链接。

<sup>①</sup> 这个解决方案也有其负面效果——如果一个浏览页返回了许多结果，那么大量的数据将在底层来回发送。另外，数据库查询也未必是最有效的。但它确实可以保证应用的无状态性。一个替代的方法是反过来在服务器端保存由browse返回的结果，然后当一个添加歌曲的请求进来时，查找适当的一点儿信息以重建正确的歌曲集。例如，你可以只是将这个值列表保存下来，而不必将它放在表单里发回服务器。或者你可以在生成浏览结果之前将RANDOM-STATE复制下来，以便后面可以重建出同样的“随机”结果。但这个思路也有它自己的问题。你永远不知道用户何时可能点击了它们浏览器的回退按钮，从而返回到一个旧的浏览页然后再点击那个“Add all”按钮。总之，欢迎来到丰富多彩的Web编程世界。

参数file、what和values与:add-songs操作配合使用。通过将values声明为类型base-64-list，define-url-function底层将负责解码由“Add all”形式提交的值。其余的参数按照注释里所描述的方式分别用于其他操作。

现在让我们来查看playlist的函数体。你需要做的第一件事是使用playlist-id来查找一个队列对象，并使用下面的两行来获取该播放列表的锁：

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist)))
```

由于lookup-playlist将在必要时创建一个新的播放列表，它将总是返回一个playlist对象。然后你进行必要的队列处理，派发action参数的值以便调用一个playlist系列的函数。

```
(case action
  (:add-songs      (add-songs playlist what (or values (list file))))
  (:delete-songs   (delete-songs
    playlist
    :file file :genre genre
    :artist artist :album album))
  (:clear          (clear-playlist playlist))
  (:sort           (sort-playlist playlist order-by))
  (:shuffle        (shuffle-playlist playlist shuffle))
  (:set-repeat     (setf (repeat playlist) repeat)))
```

函数playlist中其余的部分就是实际的HTML生成了。你可以再次使用:mp3-browser-page这个HTML宏来确保页面的基本样式匹配应用程序中的其他页面，尽管这一次你要向:header参数传递NIL以避免生成那个H1头。下面是该函数的其余部分：

```
(html
  (:mp3-browser-page
    (:title (:format "Playlist - ~a" (id playlist)) :header nil)
    (playlist-toolbar playlist)
    (if (empty-p playlist)
        (html (:p (:i "Empty.")))
        (html
          ((:table :class "playlist")
            (:table-row "#" "Song" "Album" "Artist" "Genre")
            (let ((idx 0)
                  (current-idx (current-idx playlist)))
              (do-rows (row (songs-table playlist))
                (with-column-values (track file song album artist genre) row
                  (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
                    (html
                      ((:table-row :class row-style)
                        track
                        (:progn song (delete-songs-link :file file))
                        (:progn album (delete-songs-link :album album))
                        (:progn artist (delete-songs-link :artist artist)))
                        (:progn genre (delete-songs-link :genre genre))))))
                  (incf idx))))))))))
```

其中的函数playlist-toolbar生成一个含有到playlist页面的链接的工具栏，以进行多种:action操作。而delete-songs-link可以生成一个带有设置为:delete-songs的:action

参数和其他适当参数的playlist链接，可以分别删除单独的文件、专辑里的所有文件、特定艺术家的文件或是指定风格的文件。

```
(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
      (:p :class "playlist-toolbar"
           (:i "Sort by:")
           " [ "
           (sort-playlist-button "genre" current-sort) " | "
           (sort-playlist-button "artist" current-sort) " | "
           (sort-playlist-button "album" current-sort) " | "
           (sort-playlist-button "song" current-sort) " ] "
           (:i "Shuffle by:")
           " [ "
           (playlist-shuffle-button "none" current-shuffle) " | "
           (playlist-shuffle-button "song" current-shuffle) " | "
           (playlist-shuffle-button "album" current-shuffle) " ] "
           (:i "Repeat:")
           " [ "
           (playlist-repeat-button "none" current-repeat) " | "
           (playlist-repeat-button "song" current-repeat) " | "
           (playlist-repeat-button "all" current-repeat) " ] "
           "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] "))))
  (defun playlist-button (action argument new-value current-value)
    (let ((label (string-capitalize new-value)))
      (if (string-equal new-value current-value)
          (html (:b label))
          (html (:a :href (link "playlist" :action action argument new-value) label)))))

  (defun sort-playlist-button (order-by current-sort)
    (playlist-button :sort :order-by order-by current-sort))

  (defun playlist-shuffle-button (shuffle current-shuffle)
    (playlist-button :shuffle :shuffle shuffle current-shuffle))

  (defun playlist-repeat-button (repeat current-repeat)
    (playlist-button :set-repeat :repeat repeat current-repeat))

  (defun delete-songs-link (what value)
    (html " [ " (:a :href (link "playlist" :action :delete-songs what value) "x") " ] ")))

```

## 29.8 查找播放列表

三个URL函数中的最后一个是最简单的。它可以表示一个列出了当前已创建的所有播放列表的表。通常用户不需要用到这个页面，但在开发过程中它可以给你一个有用系统状态视图。它还提供了一个选择不同的播放列表的机制，即每个播放列表ID都是一个带有显式playlist-id查询参数的playlist页面的链接，该查询参数随后会被playlist URL函数设置成粘滞的。注意，你需要获取\*playlists-lock\*以确保\*playlists\*哈希表在你对其迭代时不会发生改变。

```
(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   (:(:table :class "all-playlists")
    (:table-row "Playlist" "# Songs" "Most recent user agent")
    (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
        (html
         (:table-row
          (:a :href (link "playlist" :playlist-id (id playlist))
              (:print (id playlist)))
          (:print (table-size (songs-table playlist)))
          (:print (user-agent playlist)))))))
```

## 29.9 运行应用程序

这样就完成了。为了使用这个应用程序，只需使用第27章的load-database函数来加载MP3数据库，发布那个CSS样式表，将\*song-source-type\*设置成playlist以便find-song-source可以使用播放列表来代替前面章节里定义的单一歌曲源，最后启动AllegroServe。下面的函数为你完成了所有这些步骤，你只需在两个参数中填入适当的值就可以了：\*mp3-dir\*指定了你的MP3集所在的根目录，而\*mp3-css\*则是CSS样式表的文件名：

```
(defparameter *mp3-dir* ...)
(defparameter *mp3-css* ...)

(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css" :file *mp3-css* :content-type "text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001))
```

当你调用这个函数时，它将在从你的ID3文件中加载ID3信息时打印出一些点。然后，你可以将你的MP3客户端指向下面的URL：

<http://localhost:2001/stream.mp3>

然后再将你的浏览器指向一个好的起始点，比如：

<http://localhost:2001/browse>

这可以让你从默认的风格分类开始浏览。在你为播放列表添加了一些歌曲以后，只要点击MP3客户端的播放按钮，然后它就开始播放第一首歌了。

很明显，你可以从几方面来改进用户接口。假如你的库里有许多MP3，那么通过艺术家或专辑名的第一个字母来浏览就会很有用。或者也许你可以为播放列表页面添加一个“Play whole album”按钮，从而立刻把来自同一张专辑的所有歌曲全部放入播放列表的最顶端，并作为当前播放的歌曲。或者你还可以改变playlist类，当没有歌曲在队列中时不再播放静音，而是从数据库中随机选择一首歌曲来播放。但所有这些思路都属于应用设计的范畴，实际上并不是本书的主题。相反，接下来两章将回到软件基础设施的层面上，探索HTML生成库FOO是如何工作的。

# 实践：HTML生成库， 解释器部分



**在**本章和下一章里，你将审视过去几章里用到的HTML生成器FOO的底层实现。FOO是Common Lisp中相当普遍但在非Lisp语言，即“面向语言”的编程里相对罕见的编程类型的一个示例。相比在函数、类和宏的基础上构建出来的API，FOO提供了一个可以嵌入到你的Common Lisp程序中的领域相关语言。

FOO提供了用于同样S-表达式语言的两个语言处理器。一个是解释器，它接受作为数据的一个FOO“程序”并解释它来生成HTML。另一个是编译器，它将可能嵌入在Common Lisp代码中的FOO表达式编译成生成HTML的Common Lisp代码，并执行这些嵌入的代码。解释器是以函数emit-html的形式出现的，而编译器是作为宏html提供的，你在前面的章节里用到过它们。

在本章，你将首先认识一些在解释器和编译器之间共享的基础设施，然后了解解释器的实现。在下一章里，我将向你展示编译器是如何工作的。

## 30.1 设计一个领域相关语言

设计嵌入式语言需要两个步骤：首先，设计一门可以让你表达想要表达的事物的语言；其次，实现一个或几个处理器，接受一个以该语言表达的“程序”，然后要么进行该语言所表达的操作，要么将该程序转译成具有等价行为的Common Lisp代码。

因此，第一步是设计HTML生成语言。设计好的领域相关语言的关键是在表达能力和简洁性之间找好平衡。举个例子，一个用来生成HTML的表达性强但不太简洁的“语言”是字面HTML字符串的语言。该语言的合法“形式”是那些含有字面HTML的字符串。该“语言”的语言处理器可以通过简单地原样输出它们来处理这些形式。

30

```
(defvar *html-output* *standard-output*)  
  
(defun emit-html (html)  
  "An interpreter for the literal HTML language."  
  (write-sequence html *html-output*))  
  
(defmacro html (html)
```

```
"A compiler for the literal HTML language."  
(`(write-sequence ,html *html-output*))
```

这个“语言”表达性很强，因为它可以表达“任何”你可能生成的HTML。<sup>①</sup>另一方面，该语言没有什么简洁性可言，因为它带给你的是零压缩率，它的输入就是输出。

为了设计一个可以给你一些有用的压缩但又不会牺牲太多表达性的语言，你需要识别输出的细节中那些重复和无关的部分。然后可以让输出的这些方面隐含在该语言的语义中。

例如，由于HTML的结构，每个开放的标签都有一个配对的闭合标签。<sup>②</sup>当你手工编写HTML时，就不得不书写这些闭合标签，但可以通过隐式地生成这些闭合标签从而改进你的HTML生成语言的简洁性。

另一种以牺牲一些表达性为代价来提高简洁性的方式是，让语言处理器负责在元素之间添加适当的空白，包括空行和缩进。当通过编程生成HTML时，你通常不太关心元素前后的换行，以及不同的元素相对于它们的父元素是否正确缩进了。让语言处理器根据一些规则来插入空白就意味着你不需要担心它们了。FOO事实上支持两种模式：一种使用最少量的空白，生成极其高效和紧凑的HTML；另一种可以生成格式良好的HTML，其中不同的元素根据它们的角色相对其他元素缩进或分离开。

另一个最好可以交给语言处理器来做的细节是对特定字符的转义，诸如<、>和&这些字符在HTML中有特殊的含义。显然，如果只是通过将字符串打印到一个流中来生成HTML的话，那么将由你来把字符串中出现的任何这类特殊字符替换成对应的转义序列“&lt;”、“&gt;”和“&amp;”。但如果语言处理器知道哪些字符串将被输出成元素数据的话，那么它就可以帮你自动地转义这些字符。

## 30.2 FOO语言

理论已经讲得足够多了。下面我们来快速浏览一下FOO所实现的语言，然后你将看到两个FOO语言处理器的实现：本章是解释器，第31章是编译器。

和Lisp本身一样，FOO语言的基本语法由构成Lisp对象的Lisp形式定义而成。该语言定义了合法的FOO形式是如何被转化成HTML的。

最简单的FOO形式是诸如字符串、数字和关键字符<sup>③</sup>这样的自求值Lisp对象。你将需要函数self-evaluating-p来测试一个给定对象在FOO的意义下是否是自求值的。

```
(defun self-evaluating-p (form)  
  (and (atom form) (if (symbolp form) (keywordp form) t)))
```

---

① 事实上，它可能是表达性太强了，甚至可以生成那些不太合法的HTML输出。当然，如果你需要生成不十分正确的HTML用来补偿有bug的Web浏览器的话，这也可能成为一个特性。另外，语言处理器通常也会接受那些词法严谨并且形态良好，但运行起来以后却不可避免地产生未定义行为的程序。

② 好吧，其实是几乎所有标签。诸如IMG和BR等特定标签不是这样的。你将在30.7节里处理它们。

③ 在Common Lisp标准所描述的严格语言里，关键字符不是自求值的，尽管它们在事实上确实求值到它们自身。参见语言标准的3.1.2.1.3节或HyperSpec里的简要讨论。

通过使用PRINC-TO-STRING，满足该谓词的对象将把它们转换成字符串，然后在转义任何诸如<、>或&这类保留字符后输出。当值作为属性输出时，字符“”和‘’也需要转义。这样，你可以在一个自求值对象上调用html宏从而将其输出到\*html-output\*（初始绑定到\*STANDARD-OUTPUT\*）。表30-1给出了一些不同的自求值对象是如何被输出的。

表30-1 自求值对象的FOO输出

FOO 形式	生成的 HTML
"foo"	foo
10	10
:foo	FOO
"foo & bar"	foo & bar

当然，多数HTML都由带有标签的元素构成。用来描述每个元素的三部分信息分别是标签、属性集合以及含有文本和/或更多HTML元素的主体。这样，你需要一种方式将这三部分信息表示成Lisp对象，最好是Lisp读取器已经知道如何读取的对象。<sup>①</sup>如果你暂时不考虑属性的话，那么在Lisp列表和HTML元素之间存在一个明显的映射：任何HTML元素均可被表示成一个列表，其FIRST部分是一个名字与该元素的标签名相同的符号，而REST部分是一个代表其他HTML元素的由自求值对象或列表组成的列表。这样：

```
<p>Foo</p> ↔ (:p "Foo")
<p><i>Now</i> is the time</p> ↔ (:p (:i "Now") " is the time")
```

现在唯一的问题是在哪里插入属性。由于多数元素都没有属性，如果可以继续使用前面用于无属性元素的语法就最好了。FOO提供了两种方式来表示带有属性的元素。第一种是简单地将属性包含在列表中紧随符号之后的位置上，其中命名了属性的关键字符串和代表属性值形式的对象交替出现。元素的主体开始于列表中第一个在属性名的位置上却并非关键字符串的那一项。因此：

```
HTML> (html (:p "foo"))
<p>foo</p>
NIL
HTML> (html (:p "foo" (:i "bar") "baz"))
<p>foo <i>bar</i> baz</p>
NIL
HTML> (html (:p :style "foo" "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html (:p :id "x" :style "foo" "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

对于那些喜欢在元素的属性和主体间有更明确界限的人们，FOO还支持另一种语法：如果列表的第一个元素本身是一个以关键字符串为其首元素的列表，那么外层的列表就代表一个以该关

<sup>①</sup> 使用那些Lisp读取器知道如何读取的对象，这并不是一个十分严格的要求。由于Lisp读取器本身是可定制的，你还可以定义一个新的读取器层面的语法来处理新的对象类型。但这样做不值得，并且会带来更多麻烦。

关键字为标签的HTML元素，嵌套列表的REST部分作为其属性，外层列表的REST部分作为其主体。这样，可以像下面这样书写前面两个表达式：

```
HTML> (html ((:p :style "foo") "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html ((:p :id "x" :style "foo") "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

下面的函数测试一个给定对象是否匹配这两种语法：

```
(defun cons-form-p (form &optional (test #'keywordp))
  (and (consp form)
    (or (funcall test (car form))
      (and (consp (car form)) (funcall test (caar form))))))
```

应当将test函数参数化，因为以后你需要在该名字上使用稍微不同的谓词来测试相同的两种语法。

为了完全地抽象掉两种语法变体之间的差异，你可以定义函数parse-cons-form，它接受一个形式并将其解析成3个元素：标签、属性列表和主体列表，然后以多值的形式返回它们。实际求值点对形式的代码将使用该函数而不担心它所采用的语法。

```
(defun parse-cons-form (sexp)
  (if (consp (first sexp))
    (parse-explicit-attributes-sexp sexp)
    (parse-implicit-attributes-sexp sexp)))

(defun parse-explicit-attributes-sexp (sexp)
  (destructuring-bind ((tag &rest attributes) &body body) sexp
    (values tag attributes body)))

(defun parse-implicit-attributes-sexp (sexp)
  (loop with tag = (first sexp)
    for rest on (rest sexp) by #'cddr
    while (and (keywordp (first rest)) (second rest))
    when (second rest)
      collect (first rest) into attributes and
      collect (second rest) into attributes
    end
    finally (return (values tag attributes rest))))
```

现在已经基本规范了语言，你可以考虑如何实际来实现语言的处理器了。怎样才能将一系列的FOO形式转化成你想要的HTML呢？前面提到，需要实现FOO的两个语言处理器：一个解释器负责遍历一棵FOO形式树并直接输出对应的HTML，一个编译器遍历一棵树并将其转化成可以输出同样HTML的Common Lisp代码。无论是解释器还是编译器都将构建在一个共同的代码基础之上，它会支持诸如转义保留字符和生成美观的缩进输出之类的特性，因此我们有理由从这里开始。

### 30.3 字符转义

需要依赖的首要基础设施是知道如何转义那些HTML中带有特殊含义字符的代码。有三个这样的字符一定不能出现在元素或属性值的文本中，它们是<、>和&。在元素文本或属性值中，这些字符必须被替换成字符引用项“&lt;”、“&gt;”和“&amp;”。类似地，在属性值中，用来给值定界的引号必须被转义，把‘’变成“&apos;”把“”变成“&quot;”。此外，任何字符都可被表示成一个数值的字符引用项，后者依次由&、#、一个以十进制数表示的数值代码，以及一个分号组成。这些数值转义项有时用来在HTML中嵌入非ASCII的字符。

#### 包 定 义

由于FOO是一个底层库，你开发的这个包并不依赖很多外部代码。只有通常依赖的来自COMMON-LISP包的名字以及几乎同样经常依赖的来自COM.GIGAMONKEYS.MACRO-UTILITIES的宏生成宏的名字。另一方面，该包需要导出使用FOO的代码所需要的所有名字。下面是来自本书Web站点上下载的源代码中的`DEFPACKAGE`定义：

```
(defpackage :com.gigamonkeys.html
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :with-html-output
           :in-html-style
           :define-html-macro
           :html
           :emit-html
           :&attributes))
```

下面的函数接受单个字符并返回一个含有该字符的字符引用项的字符串：

```
(defun escape-char (char)
  (case char
    (#\& "&amp;")
    (#\< "&lt;")
    (#\> "&gt;")
    (#\' "&apos;")
    (#\" "&quot;")
    (t (format nil "&#~d;" (char-code char)))))
```

你可以使用该函数来作为函数`escape`的基础，`escape`函数接受一个字符串和一个字符序列，然后返回第一个参数的一个副本，其中所有在第二个参数中出现过的字符都被替换成由`escape-char`返回的对应的字符项。

```
(defun escape (in to-escape)
  (flet ((needs-escape-p (char) (find char to-escape)))
    (with-output-to-string (out)
      (loop for start = 0 then (+ pos)
            for pos = (position-if #'needs-escape-p in :start start)
            do (write-sequence in out :start start :end pos)
            when pos do (write-sequence (escape-char (char in pos)) out)
            while pos))))
```

你还可以定义如下两个参数：`*element-escapes*`，它含有需要在正常元素数据中转义的所有字符；`*attribute-escapes*`，它含有在属性值中需要转义的字符集。

```
(defparameter *element-escapes* "<>&")
(defparameter *attribute-escapes* "<>&\\"")
```

下面是一些例子：

```
HTML> (escape "foo & bar" *element-escapes*)
"foo &amp; bar"
HTML> (escape "foo & 'bar'" *element-escapes*)
"foo &amp; 'bar'"
HTML> (escape "foo & 'bar'" *attribute-escapes*)
"foo &amp; &apos;bar&apos;"
```

最后，还需要一个变量`*escapes*`，它绑定到需要进行转义的字符集上。其初值设置成`*element-escapes*`的值，但是当生成属性时，它会被重新绑定在`*attribute-escapes*`的值上。

```
(defvar *escapes* *element-escapes*)
```

## 30.4 缩进打印机

为了生成精美缩进输出，你可以定义一个类`indenting-printer`，它包装在一个输出流的外围，再定义一些函数在使用该类的一个实例来将字符串输出到该流的同时跟踪何时开始新的一行。该类如下所示：

```
(defclass indenting-printer ()
  ((out           :accessor out          :initarg :out)
   (beginning-of-line-p :accessor beginning-of-line-p :initform t)
   (indentation      :accessor indentation    :initform 0)
   (indenting-p       :accessor indenting-p   :initform t)))
```

操作在`indenting-printers`上的主函数是`emit`，它接受该打印机和一个字符串，并将该字符串输出到打印机的输出流上，同时跟踪它何时输出一个换行以重置`beginning-of-line-p`槽。

```
(defun emit (ip string)
  (loop for start = 0 then (1+ pos)
        for pos = (position #\Newline string :start start)
        do (emit/no-newlines ip string :start start :end pos)
        when pos do (emit-newline ip)
        while pos))
```

为了实际输出该字符串，它用到了函数`emit/no-newlines`，后者通过助手函数`indent-if-necessary`来输出任何需要的缩进，然后再将字符串写入该流。该函数也会被其他用来输出一个确定不需要换行的字符串的代码所直接调用。

```
(defun emit/no-newlines (ip string &key (start 0) end)
  (indent-if-necessary ip)
  (write-sequence string (out ip) :start start :end end)
  (unless (zerop (- (or end (length string)) start))
    (setf (beginning-of-line-p ip) nil)))
```

助手函数 `indent-if-necessary` 通过检测 `beginning-of-line-p` 和 `indenting-p` 来决定是否需要输出缩进，以及当两者均为真时输出由 `indentation` 的值所指定数量的空格。使用 `indenting-printer` 的代码可以通过操作 `indentation` 和 `indenting-p` 槽来控制缩进。递增和递减 `indentation` 可以改变前导空格的数量，而将 `indenting-p` 设置为 `NIL` 可以临时关闭缩进。

```
(defun indent-if-necessary (ip)
  (when (and (beginning-of-line-p ip) (indenting-p ip))
    (loop repeat (indentation ip) do (write-char #\Space (out ip)))
    (setf (beginning-of-line-p ip) nil)))
```

`indenting-printer API` 中的最后两个函数是 `emit-newline` 和 `emit-freshline`，两者都用来输出一个换行符，类似于 `FORMAT` 指令 `~%` 和 `~&`。这就是说，唯一的区别在于 `emit-newline` 总是输出一个换行，而 `emit-freshline` 则只有在 `beginning-of-line-p` 为假时才输出。这样，中间没有任何 `emit` 的多个 `emit-freshline` 调用将不会产生一个空行。在一些代码希望生成以换行结束的输出，而另一些代码希望生成以换行开始的输出，但你不希望两者之间产生空行时，这非常有用。

```
(defun emit-newline (ip)
  (write-char #\Newline (out ip))
  (setf (beginning-of-line-p ip) t))

(defun emit-freshline (ip)
  (unless (beginning-of-line-p ip) (emit-newline ip)))
```

有了这些先决条件，现在就可以开始进入FOO处理器的核心地带了。

## 30.5 HTML 处理器接口

现在可以定义将被FOO语言处理器用来输出HTML的接口了。你可以将该接口定义成一组广义函数，因为需要两个实现：一个实际输出HTML，而另一个可被 `html` 宏用来收集一个需要执行的操作的列表，其可被优化并编译成以更高效方式生成同样输出的代码。我把这些广义函数称为后台接口。它由下面8个广义函数构成：

```
(defgeneric raw-string (processor string &optional newlines-p))

(defgeneric newline (processor))

(defgeneric freshline (processor))

(defgeneric indent (processor))

(defgeneric unindent (processor))

(defgeneric toggle-indenting (processor))

(defgeneric embed-value (processor value))

(defgeneric embed-code (processor code))
```

这些函数中有一些明显地有其对应的indenting-printer系列函数，但重要的是理解这些广义函数定义了FOO语言处理器所使用的抽象操作，并且它们并不总是通过对indenting-printer系列函数的调用来实现的。

但也许理解这些抽象操作语义的最简单方式是去查看特化在html-pretty-printer类上的方法的具体实现，该类被用来生成人类可读的HTML。

## 30.6 美化打印机后台

你可以从定义带有两个槽的类开始：一个槽用来保存indenting-printer的实例，另一个槽用来保存制表符宽度，即对于HTML元素的每一层嵌套缩进你想要增加的空格数。

```
(defclass html-pretty-printer ()
  ((printer :accessor printer :initarg :printer)
   (tab-width :accessor tab-width :initarg :tab-width :initform 2)))
```

现在你可以实现特化在html-pretty-printer上的构成后台接口的8个广义函数上的方法了。

FOO处理器使用raw-string函数来输出不需要字符转义的字符串，这要么是因为你实际想要输出一个正常保留的字符，要么是所有的保留字都已经被转义了。通常raw-string以不含有换行的字符串来调用，因此默认的行为是使用emit/no-newlines，除非调用者指定了一个非NIL的newlines-p参数。

```
(defmethod raw-string ((pp html-pretty-printer) string &optional newlines-p)
  (if newlines-p
      (emit (printer pp) string)
      (emit/no-newlines (printer pp) string)))
```

函数newline、freshline、indent、unindent和toggle-indenting实现了对于底层、indenting-printer的相当直接的管理。唯一的亮点是只有当动态变量\*pretty\*为真时，HTML美化打印机才会生成美化的输出。当它是NIL时，你应当生成没有不必要的紧凑HTML。因此除了newline之外，下面的方法全部都会在做任何事之前检查\*pretty\*：<sup>①</sup>

```
(defmethod newline ((pp html-pretty-printer))
  (emit-newline (printer pp)))

(defmethod freshline ((pp html-pretty-printer))
  (when *pretty* (emit-freshline (printer pp)))))

(defmethod indent ((pp html-pretty-printer))
  (when *pretty*
    (incf (indentation (printer pp)) (tab-width pp))))
```

<sup>①</sup> 另一个更纯粹的面向对象的方法是定义两个类，也许是html-pretty-printer和html-raw-printer，然后对于那些只有在\*pretty\*为真时才发挥作用的方法定义特化在html-raw-printer上的空操作（no-op）方法。不过，在本例中，在定义了所有空操作方法以后，你会得到更多的代码，并且随后还需要确保在正确的时候创建了正确类的实例。但一般来讲，使用多态来替换条件语句是一个好的策略。

```
(defmethod unindent ((pp html-pretty-printer))
  (when *pretty*
    (decf (indentation (printer pp)) (tab-width pp))))
```

```
(defmethod toggle-indenting ((pp html-pretty-printer))
  (when *pretty*
    (with-slots (indenting-p) (printer pp)
      (setf indenting-p (not indenting-p)))))
```

最后，函数embed-value和embed-code只被FOO编译器使用。embed-value用来生成将输出Common Lisp表达式值的代码，而embed-code用来嵌入一点代码运行并丢弃其结果。在解释器中，你无法有目的地求值嵌入的Lisp代码，因此这些函数上的相应方法将总是报错。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (error "Can't embed values when interpreting. Value: ~s" value))
```

```
(defmethod embed-code ((pp html-pretty-printer) code)
  (error "Can't embed code when interpreting. Code: ~s" code))
```

### 使用状况系统来解决问题

一个替代的方法是使用EVAL来求值解释器中的Lisp表达式。这种方法的问题在于EVAL无法访问词法环境。因此，无法让类似下面的代码正常工作：

```
(let ((x 10)) (emit-html '(:p x)))
```

其中x是一个词法变量。在运行期传递给emit-html的符号x与同名的词法变量没有特别的关联。Lisp编译器将代码中对x的引用指向该变量，但在代码被编译以后，名字x和该变量之间就不再有必要关联了。这就是当你认为EVAL是一个解决方案时，可能会判断错误的主要原因。

不过，如果x是一个以DEFVAR或DEFPARAMETER声明的动态变量（从而可能会被命名为\*x\*以代替x），那么EVAL就可以得到它的值。这样，允许FOO解释器在某些情况下使用EVAL将是有用的，但总是使用EVAL显然不是个好主意，所以你可以将EVAL跟状况系统一起使用，将两种思想组合在一起从而博采众长。

首先定义当embed-value和embed-code在解释器中调用时你将抛出的一些错误类。

```
(define-condition embedded-lisp-in-interpreter (error)
  ((form :initarg :form :reader form)))
```

```
(define-condition value-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed values when interpreting. Value: ~s" (form c)))))
```

```
(define-condition code-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed code when interpreting. Code: ~s" (form c)))))
```

现在你可以实现`embed-value`和`embed-code`来抛出这些错误，并提供一个将使用`EVAL`来求值Lisp形式的再启动函数。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (restart-case (error 'value-in-interpreter :form value)
    (evaluate ())
    :report (lambda (s)
      (format s "EVAL ~s in null lexical environment." value))
    (raw-string pp (escape (princ-to-string (eval value)) *escapes*) t)))))

(defmethod embed-code ((pp html-pretty-printer) code)
  (restart-case (error 'code-in-interpreter :form code)
    (evaluate ())
    :report (lambda (s)
      (format s "EVAL ~s in null lexical environment." code))
    (eval code))))
```

现在你可以做类似下面的事情：

```
HTML> (defvar *x* 10)
*x*
HTML> (emit-html '(:p *x*))
```

然后你将以下列信息进入调试器：

```
Can't embed values when interpreting. Value: *x*
[Condition of type VALUE-IN-INTERPRETER]
Restarts:
 0: [EVALUATE] EVAL *X* in null lexical environment.
 1: [ABORT] Abort handling SLIME request.
 2: [ABORT] Abort entirely from this process.
```

如果你调用`evaluate`再启动，那么`embed-value`将`EVAL *x*`，得到值10，然后生成下面的HTML：

```
<p>10</p>
```

然后，出于方便的考虑，你可以在特定的情形下提供再启动函数，即可以调用`evaluate`再启动的函数。`evaluate`再启动函数无条件地调用同名的再启动，而`eval-dynamic-variables`和`eval-code`仅当其状况中的Lisp形式是一个动态变量或潜在的代码时才调用再启动。

```
(defun evaluate (&optional condition)
  (declare (ignore condition))
  (invoke-restart 'evaluate))

(defun eval-dynamic-variables (&optional condition)
  (when (and (symbolp (form condition)) (boundp (form condition)))
    (evaluate)))

(defun eval-code (&optional condition)
  (when (consp (form condition))
    (evaluate)))
```

现在你使用**HANDLER-BIND**来设置一个处理器，它自动为你调用**evaluate**再启动。

```
HTML> (handler-bind ((value-in-interpreter #'evaluate)) (emit-html '(:p *x*)))
<p>10</p>
T
```

最后，你可以定义一个宏来提供一个漂亮的语法绑定两种类型的错误的处理器。

```
(defmacro with-dynamic-evaluation ((&key values code) &body body)
  `(handler-bind (
    ,(if values `((value-in-interpreter #'evaluate)))
    ,(if code `((code-in-interpreter #'evaluate))))
   ,@body))
```

一旦定义了这个宏，就可以写出下面的代码：

```
HTML> (with-dynamic-evaluation (:values t) (emit-html '(:p *x*)))
<p>10</p>
T
```

## 30.7 基本求值规则

现在将FOO语言与它的处理器接口连接起来，你所需要的全部就是一个函数，它接受一个对象并处理它，其中调用适当的处理器函数来生成HTML。例如，当给定类似

```
(:p "Foo")
```

的简单形式时，该函数可以在处理器上执行下面的调用序列：

```
(freshline processor)
(raw-string processor "<p" nil)
(raw-string processor ">" nil)
(raw-string processor "Foo" nil)
(raw-string processor "</p>" nil)
(freshline processor)
```

目前你可以定义一个简单的函数，它只是检查一个Lisp形式是否是合法的FOO形式，并在是的情况下将其交给**process-sexp-html**来处理。在下一章里，你将为该函数添加一些额外的代码来允许其处理宏和特殊操作符。但目前它看起来像下面这样：

```
(defun process (processor form)
  (if (sexp-html-p form)
      (process-sexp-html processor form)
      (error "Malformed FOO form: ~s" form)))
```

函数**sexp-html-p**检查一个给定对象是否是合法的FOO表达式，它要么是一个自求值形式，要么是一个正确格式化了的点对。

```
(defun sexp-html-p (form)
  (or (self-evaluating-p form) (cons-form-p form)))
```

自求值形式很容易处理，只需用**PRINC-TO-STRING**将其转化成一个字符串，并转义其中出现在变量\*escapes\*中的字符，该变量前面说过是初始绑定到\*element-escapes\*的值上的。

点对形式则传递给process-cons-sexp-html来处理。

```
(defun process-sexp-html (processor form)
  (if (self-evaluating-p form)
      (raw-string processor (escape (princ-to-string form) *escapes*) t)
      (process-cons-sexp-html processor form)))
```

函数process-cons-sexp-html随后负责输出开放的标签、任何属性、主体以及闭合的标签。这里主要的复杂之处在于为了生成美化的HTML，就需要根据输出的元素类型来输出空行并调整缩进。你可以将HTML中定义的所有元素分为三类：块、段落和内联元素。当输出块元素（例如body和ul）时在它们的开放标签之前和闭合标签之后都要换行，并且它们的内容需要缩进一层。当输出段落元素（例如p、li和blockquote）时在开放标签之前和闭合标签之后都要换行。内联元素只是简单地在行内输出。下面3个参数列出了每种类型的元素：

```
(defparameter *block-elements*
  '(:body :colgroup :dl :fieldset :form :head :html :map :noscript :object
    :ol :optgroup :pre :script :select :style :table :tbody :tfoot :thead
    :tr :ul))

(defparameter *paragraph-elements*
  '(:area :base :blockquote :br :button :caption :col :dd :div :dt :h1
    :h2 :h3 :h4 :h5 :h6 :hr :input :li :link :meta :option :p :param
    :td :textarea :th :title))

(defparameter *inline-elements*
  '(:a :abbr :acronym :address :b :bdo :big :cite :code :del :dfn :em
    :i :img :ins :kbd :label :legend :q :samp :small :span :strong :sub
    :sup :tt :var))
```

函数block-element-p和paragraph-element-p测试一个给定标签是否是对应列表的一个成员。<sup>①</sup>

```
(defun block-element-p (tag) (find tag *block-elements*))

(defun paragraph-element-p (tag) (find tag *paragraph-elements*))
```

其他两个带有它们自己的谓词的分类是那些总是空的元素，例如br和hr，以及空白需要保留的三个元素pre、style和script。前者在生成正规HTML（换句话说，不是XHTML）时需要特别处理，因为它们没有对应的闭合标签。而在输出那三个内部空白需要保留的标签时，你可以临时关闭缩进，这样美化打印机就不会添加任何不属于元素实际内容的空白了。

```
(defparameter *empty-elements*
  '(:area :base :br :col :hr :img :input :link :meta :param))

(defparameter *preserve-whitespace-elements* '(:pre :script :style))
```

---

<sup>①</sup> 你不需要用\*inline-elements\*的谓词，因为你只可能测试块和段落元素。在这里，我只是出于完备性的考虑才包括了该参数。

```
(defun empty-element-p (tag) (find tag *empty-elements*))  
(defun preserve-whitespace-p (tag) (find tag *preserve-whitespace-elements*))
```

当生成HTML时你需要的最后一点儿信息是，你是否在生成XHTML，因为这将影响到你输出空元素的方式。

```
(defparameter *xhtml* nil)
```

有了全部这些信息，现在就可以开始处理一个FOO的点对形式了。你使用parse-cons-form来将列表解析成3个部分：标签符号，一个可能为空的属性键值对以及一个可能为空的主体形式。然后，你用助手函数emit-open-tag、emit-element-body和emit-close-tag来分别输出开放的标签、主体以及闭合的标签。

```
(defun process-cons-sexp-html (processor form)  
  (when (string= *escapes* *attribute-escapes*)  
    (error "Can't use cons forms in attributes: ~a" form))  
  (multiple-value-bind (tag attributes body) (parse-cons-form form)  
    (emit-open-tag processor tag body attributes)  
    (emit-element-body processor tag body)  
    (emit-close-tag processor tag body)))
```

在emit-open-tag中你需要在适当的时候调用freshline，然后用emit-attributes来输出属性。你需要将元素的主体传递给emit-open-tag，这样它在输出XHTML时，就知道究竟是用“/”还是“>”来结束标签。

```
(defun emit-open-tag (processor tag body-p attributes)  
  (when (or (paragraph-element-p tag) (block-element-p tag))  
    (freshline processor))  
  (raw-string processor (format nil "<(~a~)" tag))  
  (emit-attributes processor attributes)  
  (raw-string processor (if (and *xhtml* (not body-p)) "/>" ">")))
```

在emit-attributes中，属性名必须是关键字符，因此它不会被求值，但你应当调用顶层process函数来求值那些属性值，同时将\*escapes\*绑定到\*attribute-escapes\*。为了给指定的布尔属性带来方便，这时属性的值就是属性名本身，如果一个属性的值为T——只是T而非任何其他的真值，那么就将其值替换成该属性的名字。<sup>①</sup>

```
(defun emit-attributes (processor attributes)  
  (loop for (k v) on attributes by #'cddr do  
    (raw-string processor (format nil "~(~a~)='" k))  
    (let ((*escapes* *attribute-escapes*))  
      (process processor (if (eql v t) (string-downcase k) v)))  
    (raw-string processor "'")))
```

30

元素主体的输出过程与属性值的输出类似：你可以在主体上循环调用process来求值其中的

<sup>①</sup> 尽管XHTML要求布尔属性必须用其名字作为值以表示一个真值，但在HTML中简单地包含一个没有值的属性名也是合法的，例如使用<option selected>而非<option selected='selected'>。所有兼容HTML 4.0的浏览器都应当同时理解两种形式，但一些有bug的浏览器对于特定属性可能只理解没有值的那种形式。如果需要为这样的浏览器生成HTML，那么你将修改emit-attributes从而以不同的方式输出那些属性。

每一个Lisp形式。其余的代码用来输出换行并根据元素的类型来调整缩进。

```
(defun emit-element-body (processor tag body)
  (when (block-element-p tag)
    (freshline processor)
    (indent processor))
  (when (preserve whitespace-p tag) (toggle-indenting processor))
  (dolist (item body)
    (process processor item))
  (when (preserve whitespace-p tag) (toggle-indenting processor))
  (when (block-element-p tag)
    (unindent processor)
    (freshline processor)))
```

最后，正如你可能想到的那样，`emit-close-tag`输出闭合的标签（除非不需要闭合标签，例如当主体为空并且你要么在输出XHTML，要么该元素是特殊的空元素之一）。无论是否实际输出闭合标签，你都需要为块和段落元素输出一个结束的换行。

```
(defun emit-close-tag (processor tag body-p)
  (unless (and (or *xhtml* (empty-element-p tag)) (not body-p))
    (raw-string processor (format nil "</~(~a~)>" tag)))
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor)))
```

函数`process`是基本的FOO解释器。为了让其更易于使用，你可以定义函数`emit-html`，它调用`process`并向其传递`html-pretty-printer`和一个需要求值的Lisp形式。你可以定义并使用助手函数`get-pretty-printer`来得到美化打印器对象，该函数在`*html-pretty-printer*`被绑定的情况下返回该变量的值；否则，它生成`html-pretty-printer`的一个新实例，它使用`*html-output*`作为其输出流。

```
(defun emit-html (sexp) (process (get-pretty-printer) sexp))

(defun get-pretty-printer ()
  (or *html-pretty-printer*
      (make-instance
       'html-pretty-printer
       :printer (make-instance 'indenting-printer :out *html-output*))))
```

有了这个函数，就可以将HTML输出到`*html-output*`了。与其将变量`*html-output*`作为FOO公共API的一部分暴露出来，你不如定义一个宏`with-html-output`，让它来为你处理流的绑定。它还可以让你指定是否想要美化HTML输出，默认值是变量`*pretty*`的值。

```
(defmacro with-html-output ((stream &key (pretty *pretty*))) &body body)
  ` (let* ((*html-output* ,stream)
          (*pretty* ,pretty))
      ,@body))
```

因此，如果你想要使用`emit-html`来生成HTML到一个文件里，那么就可以这样来写：

```
(with-open-file (out "foo.html" :direction output)
  (with-html-output (out :pretty t)
    (emit-html *some-foo-expression*)))
```

## 30.8 下一步是什么

在第31章里，你将看到如何实现一个宏来将FOO表达式编译成Common Lisp，这样你就可以将HTML生成代码直接嵌入到Lisp程序中了。你还能扩展FOO语言，通过添加它自己的特殊操作符和宏来使其具有更强的表达能力。

# 实践：HTML生成库， 编译器部分

**现**在你即将看到FOO编译器是如何工作的。编译器和解释器之间的主要区别在于，解释器处理程序并直接产生某些行为，这对于FOO解释器来说就是生成HTML；但编译器处理同样的程序却可以产生以其他语言表示的实现相同行为的代码。在FOO中，编译器是将FOO转译成Common Lisp代码的Common Lisp宏，因此它可以直接嵌入到Common Lisp程序中。编译器通常比解释器更有优势，这是因为编译过程是预先完成的，因此它们可以多花一点儿时间来优化它们生成的代码，使其更加高效。FOO编译器能够做到这点是因为它通过将字面文本尽可能地合并在一起，从而使用比解释器更少量的写入操作来输出同样的HTML。当编译器是一个Common Lisp宏时，你还可以获得额外的优势：让含有嵌入式Common Lisp代码的语言更容易被编译器理解，编译器只需将这些代码识别出来，并直接嵌入到其所生成代码的正确位置上就可以了。FOO编译器将利用这种能力。

## 31.1 编译器

编译器的基本架构包括3层。首先你将实现一个类`html-compiler`，它带有一个用来保存可调整的向量的槽，这种向量用来集聚那些代表在执行`process`函数的过程中调用后台接口广义函数的那些操作码（op）。

随后你将实现后台接口中广义函数上的方法，在向量中保存操作序列。每一个op被表示成一个列表，包括一个命名了该操作的关键字和传递给产生该op的函数的参数。函数`sexp->ops`实现了编译器的第一阶段，在一个FOO形式列表的每个形式上调用带有`html-compiler`实例的`process`函数来编译该列表。

这个编译器保存的op向量随后被传给一个用来优化它的函数，将相邻的`raw-string op`合并成一次性输出的字符串组合的单个op。该优化函数也可以将那些只用于美化打印的op提取出来，这在多数时候会很重要，因为它使你得以合并更多的`raw-string op`。

最后，优化后的op向量被传递给第3个函数`generate-code`，它返回一个实际输出HTML的Common Lisp表达式的列表。当`*pretty*`为真时，`generate-code`生成使用特化在`html-pretty-printer`上的方法的代码来输出美化的HTML；而当`*pretty*`为NIL时，它生成

直接向流\*html-output\*写入的代码。

宏html实际上会生成一个含有两个展开式的主体，一个是在\*pretty\*绑定到T时生成的，而另一个是在\*pretty\*绑定到NIL时生成的。究竟使用哪个展开式取决于\*pretty\*在运行期的值。这样，每个含有对html宏调用的函数都将同时含有生成美化和紧凑输出的代码。

编译器和解释器之间的另一个明显区别是，编译器可以在它生成的代码中嵌入Lisp形式。为了利用这点，你需要修改process函数，使其在处理一个并非FOO形式的表达式时可以调用embed-code和embed-value函数。由于所有的自求值对象都是合法的FOO形式，不需要传递给process-sexp-html的形式只有那些不匹配FOO点对形式语法的列表和非关键字的符号，即唯一的非自求值原子。你可以假设任何非FOO的点对都是用来内联运行的代码，而所有的符号都是其值打算嵌入的变量。

```
(defun process (processor form)
  (cond
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form)         (embed-code processor form))
    (t                   (embed-value processor form))))
```

现在让我们查看编译器的代码。首先你应当创建两个函数，它们略微地抽象了将在编译的前两阶段用来保存op的向量。

```
(defun make-op-buffer () (make-array 10 :adjustable t :fill-pointer 0))

(defun push-op (op ops-buffer) (vector-push-extend op ops-buffer))
```

接下来你可以定义html-compiler类和特化在其上的方法，以实现后台接口。

```
(defclass html-compiler ()
  ((ops :accessor ops :initform (make-op-buffer)))

  (defmethod raw-string ((compiler html-compiler) string &optional newlines-p)
    (push-op `(:raw-string ,string ,newlines-p) (ops compiler)))

  (defmethod newline ((compiler html-compiler))
    (push-op `(:newline) (ops compiler)))

  (defmethod freshline ((compiler html-compiler))
    (push-op `(:freshline) (ops compiler)))

  (defmethod indent ((compiler html-compiler))
    (push-op `(:indent) (ops compiler)))

  (defmethod unindent ((compiler html-compiler))
    (push-op `(:unindent) (ops compiler)))

  (defmethod toggle-indenting ((compiler html-compiler))
    (push-op `(:toggle-indenting) (ops compiler)))

  (defmethod embed-value ((compiler html-compiler) value)
    (push-op `(:embed-value ,value ,*escapes*) (ops compiler)))

  (defmethod embed-code ((compiler html-compiler) code)
    (push-op `(:embed-code ,code) (ops compiler))))
```

有了这些方法，你就可以实现编译器的第一阶段sexp->ops了。

```
defun sexp->ops (body)
  (loop with compiler = (make-instance 'html-compiler)
        for form in body do (process compiler form)
        finally (return (ops compiler))))
```

在目前的阶段里你不需要担心\*pretty\*的值，只需记录下被process调用的所有函数即可。

下面是sexp->ops从一个简单的FOO形式中产生的结果：

```
HTML> (sexp->ops '((:p "Foo")))
#((:FRESHLINE) (:RAW-STRING "<p" NIL) (:RAW-STRING ">" NIL)
  (:RAW-STRING "Foo" T) (:RAW-STRING "</p>" NIL) (:FRESHLINE))
```

下一个阶段optimize-static-output接受一个op的向量并返回一个含有优化后版本的新向量。对于每个:raw-string op，算法是简单的，它将字符串写入到一个临时字符串缓冲区里。这样，相邻的:raw-string op将把需输出的字符串拼接成单个字符串。一旦遇到了一个不是:raw-string的op，你就将当前位置构造的字符串通过助手函数compile-buffer转化成一个交替出现:raw-string和:newline两个op的序列，然后再添加下一个op。这个函数也负责在\*pretty\*为NIL时清除美化打印的op。

```
(defun optimize-static-output (ops)
  (let ((new-ops (make-op-buffer)))
    (with-output-to-string (buf)
      (flet ((add-op (op)
               (compile-buffer buf new-ops)
               (push-op op new-ops)))
        (loop for op across ops do
              (ecase (first op)
                (:raw-string (write-sequence (second op) buf))
                (:newline :embed-value :embed-code) (add-op op))
                ((:indent :unindent :freshline :toggle-indenting)
                 (when *pretty* (add-op op))))
              (compile-buffer buf new-ops)))
        new-ops)))
  (defun compile-buffer (buf ops)
    (loop with str = (get-output-stream-string buf)
          for start = 0 then (1+ pos)
          for pos = (position #\Newline str :start start)
          when (< start (length str))
          do (push-op `(:raw-string ,(subseq str start pos) nil) ops)
          when pos do (push-op `(:newline) ops)
          while pos)))
```

最后一步是将这些op转化成相应的Common Lisp代码。这一阶段也会关注\*pretty\*的值。当\*pretty\*为真时，它生成在\*html-pretty-printer\*上调用后台广义函数的代码，该变量绑定在html-pretty-printer上。当\*pretty\*为NIL时，它生成的代码将直接写入\*html-output\*，后者就是美化打印器用来发送输出的那个流。

实际的函数generate-code很简单。

```
(defun generate-code (ops)
  (loop for op across ops collect (apply #'op->code op)))
```



所有的工作都是由广义函数op->code上的方法来完成的，它们全部使用op名的EQL特化符特化在了参数op上。

```
(defgeneric op->code (op &rest operands))

(defmethod op->code ((op (eql :raw-string)) &rest operands)
  (destructuring-bind (string check-for-newlines) operands
    (if *pretty*
        ` (raw-string *html-pretty-printer* ,string ,check-for-newlines)
        ` (write-sequence ,string *html-output*))))

(defmethod op->code ((op (eql :newline)) &rest operands)
  (if *pretty*
      ` (newline *html-pretty-printer*)
      ` (write-char #\Newline *html-output*)))

(defmethod op->code ((op (eql :freshline)) &rest operands)
  (if *pretty*
      ` (freshline *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :indent)) &rest operands)
  (if *pretty*
      ` (indent *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :unindent)) &rest operands)
  (if *pretty*
      ` (unindent *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :toggle-indenting)) &rest operands)
  (if *pretty*
      ` (toggle-indenting *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))
```

其中两个最有趣的op->code方法是用来为:embed-value和:embed-code这两个op生成代码的方法。在:embed-value方法中，可以根据escapes操作数的值来生成稍有不同的代码，因为当escapes为NIL时你不需要生成对escape的调用。而当\*pretty\*和escapes均为NIL时，你可以生成使用PRINC来直接向流中输出值的代码。

```
(defmethod op->code ((op (eql :embed-value)) &rest operands)
  (destructuring-bind (value escapes) operands
    (if *pretty*
        (if escapes
            ` (raw-string
                *html-pretty-printer* (escape (princ-to-string ,value) ,escapes) t)
            ` (raw-string *html-pretty-printer* (princ-to-string ,value) t))
        (if escapes
            ` (write-sequence (escape (princ-to-string ,value) ,escapes) *html-output*)
            ` (princ ,value *html-output*))))
```

这样，类似下面的代码

```
HTML> (let ((x 10)) (html (:p x)))
<p>10</p>
NIL
```

就能够正常工作了，因为html将(:p x)转译成了类似下面的形式：

```
(progn
  (write-sequence "<p>" *html-output*)
  (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
  (write-sequence "</p>" *html-output*))
```

当上述代码在LET上下文中代替了对html的调用时，你可以得到下面的代码：

```
(let ((x 10))
  (progn
    (write-sequence "<p>" *html-output*)
    (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (write-sequence "</p>" *html-output*)))
```

并且在生成的代码中，对x的引用将变成一个对来自html形式外围LET的词法变量的引用。

另一方面，:embed-code方法有趣是因为它是如此地简单。process传递形式到embed-code，后者又将其插入到:embed-code op中，因此你只需再把它拉出来并返回。

```
(defmethod op->code ((op (eql :embed-code)) &rest operands)
  (first operands))
```

这让类似下面的代码得以工作：

```
HTML> (html (:ul (dolist (x '(foo bar baz)) (html (:li x)))))
<ul>
  <li>FOO</li>
  <li>BAR</li>
  <li>BAZ</li>
</ul>
NIL
```

其中外层的html调用可以展开成类似下面的形式：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz)) (html (:li x)))
  (write-sequence "</ul>" *html-output*)))
```

如果你在DOLIST的主体中展开对html的调用，那么你将得到类似下面的东西：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz))
    (progn
      (write-sequence "<li>" *html-output*)
      (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
      (write-sequence "</li>" *html-output*)))
    (write-sequence "</ul>" *html-output*)))
```

事实上，这些代码将会生成你所见过的输出。

## 31.2 FOO 特殊操作符

你可以就此打住，FOO语言的表达性已经足够用来生成你所关心的几乎任何HTML了。尽管如此，你还可以为该语言添加两个特性，只需要再写一点儿代码就可以使其更加强大：特殊操作符和宏。

FOO中的特殊操作符类似于Common Lisp中的特殊操作符。特殊操作符可用来在语言中表达那些无法通过语言的基本求值规则来表达的事物。或者从另一个角度来看，特殊操作符提供了访问语言求值器所使用的底层机制的能力。<sup>①</sup>

举一个简单的例子，在FOO编译器中，语言求值器使用embed-value函数来生成代码，把变量的值嵌入到输出的HTML中。不过，由于传递给embed-value的只是符号，无法在目前所描述的语言里嵌入任意Common Lisp表达式的值。process函数将点对单元传给了embed-code而非embed-value，因此返回的值被忽略了。通常这就是你所需要的东西，因为在FOO程序中嵌入Lisp代码的主要原因是使用Lisp的控制构造。不过，有时你也希望在生成的HTML中嵌入计算后的值。例如，你可能希望下面的FOO程序可以生成一个含有随机数的段落标签：

```
(:p (random 10))
```

但这样不行，因为代码会运行，然后值被丢掉了。

```
HTML> (html (:p (random 10)))
<p></p>
NIL
```

在目前你所实现的语言中，可以通过在html的外面计算该值，然后再通过一个变量嵌入它，从而绕过这个限制。

```
HTML> (let ((x (random 10))) (html (:p x)))
<p>1</p>
NIL
```

但这样做很麻烦，尤其是当你考虑把(random 10)传递给embed-value而非embed-code时，最初的形式刚好可以完成你想要做的事。因此，你可以定义一个特殊操作符:print，它被FOO语言处理器按照一个和正常FOO表达式不同的规则来处理。确切地说，不是输出一个<print>元素，而是将其主体中的形式传给embed-value。这样，你就可以像下面这样生成一个含有随机数的段落：

```
HTML> (html (:p (:print (random 10))))
<p>9</p>
NIL
```

很明显，这个特殊操作符只在编译的FOO代码中才有用，因为embed-value不能工作在解释器中。另一个可以同时用在解释和编译的FOO代码中的特殊操作符是:format，它让你使用FORMAT函数来产生输出。特殊操作符:format的参数是一个用作格式控制字符串的字符串和任

<sup>①</sup> 对于FOO特殊操作符和宏之间的相似之处，我将在31.3节里讨论到，这与Lisp本身的情况是一样的。事实上，理解了FOO特殊操作符和宏的工作方式，也有助于让你理解Common Lisp的有关设计理念。

何需要插入的参数。当所有`:format`的参数都是自求值对象时，通过将它们传递给`FORMAT`而生成出一个字符串，并且随后像其他任何字符串那样输出该字符串。这允许`:format`形式被用在传递给`emit-html`的FOO中。在编译过的FOO中，`:format`的参数可以是任意Lisp表达式。

其他特殊操作符控制了哪些字符被自动转义以及显式输出换行：`:noescape`特殊操作符使其主体中的所有形式作为正常FOO形式来求值，除了`*escapes*`绑定到`NIL`，`:attribute`可以在`*escapes*`绑定为`*attribute-escapes*`的情况下求值其主体中的Lisp形式。而`:newline`则被转译成输出一个显式换行的代码。

那么，你怎样才能定义特殊操作符呢？对特殊操作符的处理有两个方面：语言处理器如何识别那些使用了特殊操作符的形式，以及在处理每个特殊操作符的时候如何得知需要运行哪些代码？

你可以修改`process-sexp-html`来识别每一个特殊操作符，并用适当的方法来处理它们——特殊操作符在逻辑上是语言实现的一部分，并且它们不会有太多。不过，如果有更加模块化的方法来添加新的特殊操作符就更好了，这并不是为了方便FOO的用户而只是为了方便你自己。

我们将“特殊形式”定义为任何这样的列表，其`CAR`是一个特殊操作符名字的符号。你可以通过将一个非`NIL`值添加到该符号属性表中关键字`html-special-operator`下以标记特殊操作符的名字。因此，可以像下面这样定义函数来测试一个给定形式是否为特殊形式：

```
(defun special-form-p (form)
  (and (consp form) (symbolp (car form)) (get (car form) 'html-special-operator)))
```

实现每个特殊操作符的代码负责按照它们需要的方式提取该列表的其余部分，并按照特殊操作符所要求的语义来做事。假设你定义一个函数`process-special-form`，它接受语言处理器和一个特殊形式，并运行适当的代码来生成处理器对象上的一个调用序列，那么你可以修订顶层的`process`函数，像下面这样来处理特殊形式：

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form) (embed-code processor form))
    (t (embed-value processor form))))
```

必须将`special-form-p`子句放在最前面，因为特殊形式可能看起来在词法上跟正常的FOO表达式一样，就好像Common Lisp的特殊形式也可能看起来像正常函数调用一样。

现在你只需实现`process-special-form`就好了。不用去定义实现了所有特殊操作符的单个函数，而应当转而定义一个宏，允许你像正规函数一样地定义特殊操作符，并负责在特殊操作符的名字的属性表中添加`html-special-operator`项。事实上，你保存在属性表中的值可以是一个实现该特殊操作符的函数。下面就是这个宏：

```
(defmacro define-html-special-operator (name (processor &rest other-parameters)
                                         &body body)
  ` (eval-when (:compile-toplevel :load-toplevel :execute)
      (setf (get ',name 'html-special-operator)
            (lambda (,processor ,@other-parameters) ,@body))))
```

这是一个相当高级的宏类型，但如果你逐行地观察它就会发现，其实也没有什么特别的。为

了了解它的工作方式，我们取该宏的一个简单用例，即特殊操作符: noescape 的定义，然后查看其宏展开式。如果你写出了下面的定义：

```
(define-html-special-operator :noescape (processor &rest body)
  (let ((*escapes* nil))
    (loop for exp in body do (process processor exp))))
```

那么它相当于下面的写法：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ':noescape 'html-special-operator)
    (lambda (processor &rest body)
      (let ((*escapes* nil))
        (loop for exp in body do (process processor exp))))))
```

在第20章里讨论过，**EVAL-WHEN**特殊操作符确保了当你用**COMPILE-FILE**编译时，其主体中的代码产生的效果在编译期可见。如果你想要在一个文件中使用**define-html-special-operator**，然后在同样的文件中使用刚刚定义的特殊操作符的话，那么这个**EVAL-WHEN**将是必要的。

随后的**SETF**表达式将符号:**:noescape**的属性**html-special-operator**设置成一个匿名函数，其参数列表与**define-html-special-operator**中指定的参数列表相同。通过将**define-html-special-operator**定义成参数列表分成两部分的形式，即**processor**和其余的部分，你可以确保所有的特殊操作符都至少接受一个参数。

该匿名函数的主体就是提供给**define-html-special-operator**的主体。该匿名函数的任务是通过在后台接口上作适当的调用来生成正确的HTML，或生成可以生成它们的代码，从而实现该特殊操作符。它还可以使用**process**来求值一个作为FOO形式的表达式。

特殊操作符:**:noescape**尤其简单，它所做的全部就是在\*escapes\*绑定到NIL的情况下，将其主体中的Lisp形式传递给**process**。换句话说，这个特殊操作符禁止了由**process-sexp-html**进行的正常字符转义。

有了以这种方式定义的特殊操作符，**process-special-form**所要做的就是查询特殊操作符名字的属性表中的匿名函数，并将其应用在处理器和Lisp形式的其余部分上。

```
(defun process-special-form (processor form)
  (apply (get (car form) 'html-special-operator) processor (rest form)))
```

现在你可以开始定义其余的5个FOO特殊操作符了。与**:noescape**类似的是**:attribute**，它在\*escapes\*绑定到\*attribute-escapes\*的情况下对其主体中的Lisp形式求值。这个特殊操作符在你想要编写用来输出属性值的助手函数时将会很有用。如果你编写一个类似下面的函数：

```
(defun foo-value (something)
  (html (:print (frob something))))
```

其中的**html**宏用来生成在\*element-escapes\*中转义字符的代码。但如果你正在计划像下面这样来使用**foo-value**：

```
(html (:p :style (foo-value 42) "Foo"))
```

那么你会希望它可以生成使用`*attribute-escapes*`的代码。因此，你可以另行像下面这样来编写该函数：<sup>①</sup>

```
(defun foo-value (something)
  (html (:attribute (:print (frob something)))))
```

`:attribute`的定义如下所示：

```
(define-html-special-operator :attribute (processor &rest body)
  (let ((*escapes* *attribute-escapes*))
    (loop for exp in body do (process processor exp))))
```

下面两个特殊操作符`:print`和`:format`用来输出值。早先讨论过，特殊操作符`:print`用于在编译过的FOO程序里嵌入任意Lisp表达式的值，差不多等价于先用`(format nil ...)`生成一个字符串然后再嵌入它。将`:format`定义为特殊操作符主要是出于方便的考虑。

```
(:format "Foo: ~d" x)
```

比下面这个更好一些：

```
(:print (format nil "Foo: ~d" x))
```

它还有另外一点优势，如果你将`:format`与全部是自求值的参数一起使用，那么FOO可以在编译期求值`:format`而无需等到运行期再做。`:print`和`:format`的定义如下所示：

```
(define-html-special-operator :print (processor form)
  (cond
    ((self-evaluating-p form)
     (warn "Redundant :print of self-evaluating form ~s" form)
     (process-sexp-html processor form))
    (t
     (embed-value processor form)))

(define-html-special-operator :format (processor &rest args)
  (if (every #'self-evaluating-p args)
      (process-sexp-html processor (apply #'format nil args))
      (embed-value processor `(format nil ,@args))))
```

特殊操作符`:newline`强制输出一个字面换行，这有时是有用的。

```
(define-html-special-operator :newline (processor)
  (newline processor))
```

最后，特殊操作符`:progn`和Common Lisp中的`PROGN`特殊操作符相似。它简单地按顺序处理其主体中的Lisp形式。

```
(define-html-special-operator :progn (processor &rest body)
  (loop for exp in body do (process processor exp)))
```

换句话说，下面的形式

```
(html (:p (:progn "Foo" (:i "bar") "baz")))
```

<sup>①</sup> `:noescape`和`:attribute`特殊操作符必须被定义成特殊操作符，这是因为FOO是在编译期决定如何转义的，而不是运行期。这允许FOO在编译期转义字面值，从而比在运行期扫描所有输出要高效得多。

将生成与下面形式相同的代码：

```
(html (:p "Foo" (:i "bar") "baz"))
```

这可能看起来是个奇怪的需要，因为正常的FOO表达式可以在其主体中有任意多个形式。不过，这个特殊操作符将在一种情形里变得非常有用——当编写FOO宏的时候，这将把你带到你需要实现的最后一个语言特性上。

### 31.3 FOO 宏

FOO宏类似于Common Lisp宏。FOO宏是一点儿代码，它接受FOO表达式作为参数并返回一个新的FOO表达式作为结果，后者随后按照正常的FOO求值规则来求值。实际的实现与特殊操作符的实现非常相似。

和特殊操作符一样，你可以定义一个谓词来测试给定Lisp形式是否是一个宏形式。

```
(defun macro-form-p (form)
  (cons-form-p form #'(lambda (x) (and (symbolp x) (get x 'html-macro)))))
```

使用前面定义的函数cons-form-p，是因为你想要允许宏被用在所有非宏的FOO点对形式语法中。不过，你需要传递一个不同的谓词函数，它用来测试形式名是否是一个带有非NIL的html-macro属性的符号。另外，和特殊操作符的实现一样，你将定义一个宏用来定义FOO宏，它负责将一个函数保存在宏名的属性表中，位于关键字html-macro之下。不过，定义一个宏的过程会更加复杂一些，因为FOO支持两种类型的宏。一些你将定义的宏类似于正常的HTML元素，并且可能想要容易地访问一个属性列表。其他的宏只是简单地想要直接访问它们主体的元素。

你可以使这两种类型的宏的区别变得隐性：当你定义一个FOO宏时，参数列表可以包含一个&attributes参数。如果有这个参数的话，那么宏形式将按照正常点对形式来解析，并且宏函数将被传递两个值，一个属性的plist和一个构成宏形式体的表达式列表。没有&attributes参数的宏形式将不会解析属性，并且宏函数将被应用在单一参数上，即一个含有主体表达式的列表。前者对于本质上的HTML模板很有用。例如：

```
(define-html-macro :mytag (&attributes attrs &body body)
  `((:div :class "mytag" ,@attrs) ,@body))

HTML> (html (:mytag "Foo"))
<div class='mytag'>Foo</div>
NIL
HTML> (html (:mytag :id "bar" "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
HTML> (html ((:mytag :id "bar") "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
```

后一种类型的宏对于编写管理其主体中Lisp形式的宏更加有用。这个类型的宏可以作为一种HTML控制构造类型来使用。作为一个简单的例子，考虑下面这个实现了;if构造的宏：

```
(define-html-macro :if (test then else)
  ` (if ,test (html ,then) (html ,else)))
```

该宏允许你写出下面的代码：

```
(:p (:if (zerop (random 2)) "Heads" "Tails"))
```

而不必写成下面这个较长的版本：

```
(:p (if (zerop (random 2)) (html "Heads") (html "Tails"))))
```

为了决定你应该生成哪种类型的宏，你需要一个函数来解析`define-html-macro`的参数列表。该函数返回两个值，`&attributes`参数的名字或者其不存在时为`NIL`，以及一个含有`args`中去掉`&attributes`标记及其后续列表元素后剩下的所有元素的列表。<sup>①</sup>

```
(defun parse-html-macro-lambda-list (args)
  (let ((attr-cons (member '&attributes args)))
    (values
      (cadr attr-cons)
      (nconc (ldiff args attr-cons) (cddr attr-cons)))))

HTML> (parse-html-macro-lambda-list '(a b c))
NIL
(A B C)
HTML> (parse-html-macro-lambda-list '(&attributes attrs a b c))
ATTRS
(A B C)
HTML> (parse-html-macro-lambda-list '(a b c &attributes attrs))
ATTRS
(A B C)
```

参数列表中`&attributes`后面跟着的元素也可以是一个解构参数列表。

```
HTML> (parse-html-macro-lambda-list '(&attributes (&key x y) a b c))
(&KEY X Y)
(A B C)
```

现在你可以开始定义`define-html-macro`了。根据是否指定了`&attributes`参数，你需要生成HTML宏的两种形式之一，因此主宏简单地检测其正在定义哪种类型的HTML宏，并随后调用一个助手函数来生成正确类型的代码。

```
(defmacro define-html-macro (name (&rest args) &body body)
  (multiple-value-bind (attribute-var args)
    (parse-html-macro-lambda-list args)
    (if attribute-var
        (generate-macro-with-attributes name attribute-var args body)
        (generate-macro-no-attributes name args body))))
```

实际生成展开式的函数如下所示：

```
(defun generate-macro-with-attributes (name attribute-args args body)
  (with-gensyms (attributes form-body)
    (if (symbolp attribute-args) (setf attribute-args `(&rest ,attribute-args))))
```

---

<sup>①</sup> 注意`&attributes`只不过是另一个符号罢了，以“`&`”开头的名字本质上没有什么特别之处。

```

`(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'html-macro-wants-attributes) t)
  (setf (get ',name 'html-macro)
    (lambda (,attributes ,form-body)
      (destructuring-bind (,@attribute-args) ,attributes
        (destructuring-bind (,@args) ,form-body
          ,@body))))))

(defun generate-macro-no-attributes (name args body)
  (with-gensyms (form-body)
    `(eval-when (:compile-toplevel :load-toplevel :execute)
      (setf (get ',name 'html-macro-wants-attributes) nil)
      (setf (get ',name 'html-macro)
        (lambda (,form-body)
          (destructuring-bind (,@args) ,form-body ,@body))))))

```

你将定义的宏函数接受一个或两个参数，然后使用`DESTRUCTURING-BIND`来将参数提取出来并绑定到在对`define-html-macro`的调用中所定义的参数上。在两个展开式中，你都需要将宏函数保存在其名字的属性表中的`html-macro`之下，并且在属性`html-macro-wants-attributes`下保存一个布尔值，以指示该宏是否接受`&attributes`参数。你在下面的函数`expand-macro-form`中使用该属性来决定宏函数被调用的方式：

```

(defun expand-macro-form (form)
  (if (or (consp (first form))
           (get (first form) 'html-macro-wants-attributes))
      (multiple-value-bind (tag attributes body) (parse-cons-form form)
        (funcall (get tag 'html-macro) attributes body))
      (destructuring-bind (tag &body body) form
        (funcall (get tag 'html-macro) body))))

```

最后一步是通过在顶层`process`函数的派发`COND`语句里添加一个子句来集成对宏的支持。

```

(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((macro-form-p form) (process processor (expand-macro-form form)))
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form) (embed-code processor form))
    (t (embed-value processor form))))

```

这就是`process`的最终版本。

## 31.4 公共 API

现在，你终于完成了对`html`宏的实现，它就是FOO编译器的主入口点。FOO公共API的其余部分还包括我在上一章里讨论过的`emit-html`和`with-html-output`，以及上一节里讨论过的`define-html-macro`。`define-html-macro`之所以需要成为公共API的一部分，是因为FOO的用户也希望编写自己的HTML宏。另一方面，`define-html-special-operator`不是公共API一部分的理由，是它需要太多的关于FOO内部的知识来定义一个新的特殊操作符。并且应该几乎

没有什么功能是无法使用已有的语言和特殊操作符来完成的。<sup>①</sup>

在我到达html之前，公共API的最后一个元素是另一个宏`in-html-style`。该宏通过设置`*xhtml*`变量来控制FOO生成XHTML还是正常的HTML。将其定义为宏的原因是你希望把设置`*xhtml*`的代码包装在`EVAL-WHEN`中，这样就可以在一个文件里设置它，并让其影响同一个文件中的所有对html宏后续的使用。

```
(defmacro in-html-style (syntax)
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (case syntax
      (:html (setf *xhtml* nil))
      (:xhtml (setf *xhtml* t)))))
```

最后让我们来查看html宏本身。实现html的唯一难点是必须生成那种可同时生成美观和紧凑输出的代码，具体取决于变量`*pretty*`在运行期的值。这样，html需要生成一个含有IF表达式和两个版本代码的展开式，一个在`*pretty*`绑定为真时编译，另一个在它绑定为NIL时编译。更复杂的是，html调用经常会含有嵌入的html调用，就像这样：

```
(html (:ul (dolist (item stuff)) (html (:li item))))
```

如果外层的html展开成一个带有两个版本代码的IF表达式，一个用在`*pretty*`为真时，另一个用在其为假时，那么再把内嵌的html形式也展开成两个版本的话就太傻了。事实上，这将导致代码量呈指数级增长，因为内嵌的html也将展开两次——一次是在`*pretty*`为真的分支里，另一次是在`*pretty*`为假的分支里。如果每个展开式都生成两个版本，那么你总共有4个版本。而如果这个内嵌的html形式还含有另一个内嵌的html形式，那么你最后将得到该代码的8个版本。如果编译器足够聪明，它最终会意识到其生成的多数代码都是没用的，并将清除它们，但即便找出这点也需要相当多的时间，从而减慢了任何使用嵌套html调用的函数的编译时间。

幸运的是，你可以通过生成一个用MACROLET局部重定义html宏的展开式，让其只生成正确类型的代码，从而避免无用代码的爆炸。首先你定义一个助手函数，它接受由`sexp->ops`返回的op向量并在`*pretty*`绑定为指定值的情况下对其应用`optimize-static-output`和`generate-code`，即受`*pretty*`影响的两个阶段，然后再将得到的结果插入到`PROGN`中。（`PROGN`返回`NIL`是为了让输出更简洁。）

```
(defun codegen-html (ops pretty)
  (let ((*pretty* pretty))
    `(progn ,(generate-code (optimize-static-output ops) nil))))
```

有了这个函数，你随后就可以像下面这样来定义html：

```
(defmacro html (&whole whole &body body)
  (declare (ignore body))
  `(if *pretty*
      (macrolet ((html (&body body) (codegen-html (sexp->ops body) t)))
        ...)))
```

<sup>①</sup> 在底层的语言处理基础设施里，到目前为止，还没有充分地通过特殊操作符暴露出来的一个元素是对缩进的处理。如果你想要令FOO更加灵活（尽管这要以增加其API的复杂度为代价），那么你可以添加用于管理底层缩进打印机的特殊操作符。不过看起来解释额外的特殊操作符的代价将远远超出在语言表达性上获得的微小提升。

```
(let ((*html-pretty-printer* (get-pretty-printer))) ,whole)
  (macrolet ((html (&body body) (codegen-html (sexp->ops body) nil)))
    ,whole)))
```

其中的`&whole`参数代表最初的`html`形式，而由于它被插入到两个`MACROLET`主体的展开式中，它将使用每个`html`的新定义重新处理，一个生成美化打印的代码，另一个生成非美化打印的代码。注意变量`*pretty*`被同时用于宏展开期和结果代码运行期。在宏展开期，它被`codegen-html`用来使`generate-code`生成一种或另一种类型的代码。而在运行期，它被顶层`html`宏生成的`IF`表达式用来决定实际上应该运行美化打印还是非美化打印的代码。

## 31.5 结束语

和往常一样，你可以继续研究这些代码，以不同的方式来增强它。一个有趣的方向是使用底层的输出框架来产生其他类型的输出。从本书Web站点上下载的FOO版本中，你将找到一些实现了CSS输出的代码，它们可在解释器和编译器中与HTML输出集成在一起。这是一个有趣的案例，因为CSS的语法不能像HTML那样简单地映射到S-表达式上。不过，如果你实际查看那些代码，将看到定义一种S-表达式语法来表示CSS中的多种结构仍然是可能的。

一项更具雄心的底层处理是添加对生成嵌入式JavaScript的支持。如果做法得当，那么为FOO添加JavaScript支持可以得到两大好处。一个好处是，在定义出一种可映射到JavaScript语法的S-表达式语法以后，你就可以开始编写Common Lisp的宏，为你用来编写客户端代码的语言添加新的构造，然后再将它们编译成JavaScript。另一个好处是，作为从FOO的S-表达式JavaScript到正常JavaScript转换的一部分，你可以轻松处理不同浏览器的JavaScript实现中的那些细微但令人讨厌的区别。这就是说，FOO生成的JavaScript代码要么可以含有适当的条件代码，在不同的浏览器里做不同的事，要么直接根据你想要支持的浏览器来生成不同的代码。然后如果你把FOO用在动态生成的页面中，它可以使用来自User-Agent中的信息来让请求生成针对该浏览器的JavaScript代码。

如果这些事情激发了你的兴趣，那么你应当自行去实现它们，因为这已经是本书实践性内容的最后一章了。在下一章里我将做个总结，同时简要讨论一些我尚未在本书涉及的主题，包括如何查找第三方库，如何优化Common Lisp代码，以及如何交付Lisp应用程序。

# 结论：下一步是什么



**到目前为止，我希望你终于明白本书的书名并不矛盾了。不过，确实有一些领域对你的编程实践非常重要而我并未谈及。例如，我没有提到如何开发图形用户界面（GUI），如何连接关系型数据库，如何解析XML或是如何编写用作多种网络协议客户端的程序。类似地，当你用Common Lisp来编写实际的应用程序时，还有两个非常重要的主题尚未讨论：优化Lisp代码，为交付而打包应用程序。**

显然，我并不打算在这最后的一章里深入讨论所有这些主题，我只想给你一些指点，以便你去探索Lisp编程中你最关注的方面。

## 32.1 查找 Lisp 库

尽管Common Lisp自带的函数、数据类型和宏的标准库规模宏大，但它只提供了通用的编程构造。诸如编写GUI、与数据库进行通信以及解析XML之类的特定任务，都需要用到ANSI标准化语言之外所提供的第三方库。

要获取一个用来做你想做的事情的库，最简单方式可能就是简单地查看Lisp实现。多数实现至少提供了Lisp语言标准里没有指定的一些功能。Common Lisp厂商为证明其价值，尤其倾向于提供用于其实现的附加库。例如，Franz的Allegro Common Lisp企业版就自带了一些库，用于解析XML、进行SOAP通信、生成HTML、连接关系型数据库以及用多种方式构建图形界面。另一个卓越的商业Lisp平台LispWorks也提供了几个类似的库，包括广泛使用的可移植的GUI工具箱API，用来开发能够运行在任何支持LispWorks的操作系统之上的GUI应用程序。

免费和开源的Common Lisp实现通常不包含许多打包的库，而是依赖于可移植的免费和开源库。但是，即便是那些实现，通常也会支持一些语言标准没有涉及的重要领域，例如网络和多线程。

使用与具体实现相关的库的唯一缺点是，它们将你捆绑在了提供这些库的实现上。如果你正在交付面向最终用户的应用程序，或者正在一台你所控制的服务器上部署基于服务器的应用程序，那么这可能不是什么大问题。但如果你想要编写可以共享给其他Lisp程序员的代码，或者只是简单地不想把自己捆绑在特定实现上，那么就有点恼人了。

对于可移植的库，其可移植性要么来源于它们完全是用标准的Common Lisp写成的，要么是

因为它们包含了适当的读取期条件化，从而可以工作在多个实现上<sup>①</sup>，获得它们的最佳途径就是借助Web。虽然说URL变化很快，通常会在打印出来后立即失效，不过可以将下面三个作为当前的最佳起始点。

- Common-Lisp.net (<http://www.common-lisp.net/>) 是一个含有免费和开源Common Lisp项目的站点，它提供了版本控制、邮件列表以及项目页面的Web服务。在该站点启动后的一年半里，注册了将近100个项目。
- Common Lisp Open Code Collection (CLOCC), (<http://clocc.sourceforge.net/>) 是一个较早的免费软件库聚集地，目的是在各个Common Lisp实现之间做到可移植，并试图不依赖于任何未包含在CLOCC的库。
- Cliki (<http://www.cliki.net/>) 是一个提供用Common Lisp编写的各种免费软件的Wiki站点。尽管与其他任何Wiki站点一样，它可能随时发生变化，但它通常可以链接到相当多的库和多种开源Common Lisp实现。该站点运行所依赖的软件也是用Common Lisp写成的。运行Debian或Gentoo发行版的Linux用户也可以轻松地安装越来越多的Lisp库，这些库与发行版的打包工具，即Debian的apt-get和Gentoo的emerge，打包在了一起。

我在这里不会推荐具体的库，因为这些库的情况日新月异。在对Perl、Python和Java的库集合觊觎了多年以后，Common Lisp程序员在过去的几年里也终于开始勇敢地接受挑战，开始为Common Lisp提供应有的开源及商业库了。

近年非常活跃的一个领域是GUI前端。Common Lisp不像Java和C#，但却跟Perl、Python和C的情况相似，不存在于开发GUI的单一方法。相反，这项工作依赖于你所使用的发行版和你想要支持的操作系统。

商业的Common Lisp实现通常提供了某种方式在它们支持的平台上构建GUI。其中LispWorks还提供了CAPI，前面提到过，这是一种可移植的GUI API。

在开源领域，你有几种选择。在Unix上，你可以使用CLX来编写底层的X-Window GUI，CLX是X-Window协议的一个纯Common Lisp的实现，几乎等价于C的xlib库。或者你可以使用几种对GTK和Tk这类高层次API和工具箱的绑定，这与你在Perl和Python里的工作方式差不多。

或者，如果你在寻找一些完全不同的工具，可以看一下Common Lisp Interface Manager (CLIM)。作为Symbolics Lisp Machine GUI框架的后裔，CLIM既强大又复杂。尽管事实上许多商业Common Lisp实现支持它，但并未见其被大量使用。不过在过去的几年里，一种CLIM的开源实现，McCLIM——目前放在Common-Lisp.net上，正在蓬勃地发展，因此也许我们正处于CLIM复苏的边缘。

---

<sup>①</sup> Common Lisp的读取期条件化和宏使得开发可移植库成为可能，这些库本身只是为了在不同实现中为语言标准没有指定的那些功能提供的API之上提供一个通用的API层。第15章的可移植路径名库就是这类库的一个例子，它在具体实现相关的API上尽可能地消除了对于语言标准的不同解释。

## 32.2 与其他语言接口

尽管许多有用的库都可以只用语言标准里指定的特性，以“纯粹的”Common Lisp来编写，并且还有更多的库可以使用由给定实现所提供的非标准功能用Lisp写出来，但有时使用来自C语言等其他语言的库会更为直接。

语言标准并未指定一种机制让Lisp代码得以调用其他语言写成的代码，甚至也没有要求具体实现提供这样一种机制。不过近年来，几乎所有的Common Lisp发行版都支持所谓的“外部函数接口”(Foreign Function Interface或简称FFI)。<sup>①</sup>FFI的基本工作是让你给Lisp足够的信息以便其可以链接外部代码。因此，如果你打算调用一个来自某C库的函数，就需要告诉Lisp如何将传递给该函数的Lisp对象转化成C类型，然后再将该函数的返回值转化回Lisp对象。不过，每个实现都提供了它们自己的FFI，每种FFI都有稍微不同的功能和语法。某些FFI允许从C向Lisp回调，而一些则不允许。Universal Foreign Function Interface (UFFI)项目提供了七八种不同Common Lisp实现上的可移植的FFI兼容层。它的工作方式是定义自己的宏，然后展开成其具体实现上的适当FFI代码。UFFI的思路是选取最底层的共同特征，这意味着它无法充分利用不同实现的FFI的所有特性，但它确实提供了一种构建基本C API外围Lisp封装的好方法。<sup>②</sup>

## 32.3 让它工作，让它正确，让它更快

人们总是说，并分别由Donald Knuth、C.A.R. Hoare和Edsger Dijkstra强调过，过早地进行优化是万恶之源。<sup>③</sup>Common Lisp是一门杰出的语言，使用它你在拥有充分表达能力的同时还可以得到很高的性能。如果你曾经听到过有关Lisp很慢的传言的话，可能会感到惊奇。在Lisp的早期岁月里，当计算机还在使用穿孔卡片来编程时，Lisp的高级特性可能确实让其慢于竞争对手，也即汇编语言和FORTRAN。但那已经是很久以前的事情了。在同一时期，Lisp曾经被用于从创建复杂AI系统到编写操作系统等一系列任务中，并且大量的工作被用来找出如何将Lisp编译成高效的代码。在本节里，我将讨论为什么说Common Lisp是用来编写高性能代码的杰出语言，并介绍相关的一些技术。

带有讽刺意味的是，说Lisp是一门用来编写高性能代码的优秀语言，首要原因正是Lisp编程的动态本质——这正是当初使Lisp的性能难以达到FORTRAN编译器水平的原因。Common Lisp的动态特性使其易于编写高性能代码，因为编写高效代码的第一步是要找到正确的算法和数据结构。

① 外部函数接口基本上等价于Java中的JNI，Perl中的XS或者Python中的扩展模块API。

② 在写这本书时，UFFI的两大缺点是缺少对从C到Lisp回调的支持（很多但并非全部实现的FFI都支持），以及缺少对CLISP的支持。后者的FFI很好但与其他实现的区别很大，以至于无法轻易集成到UFFI模型之中。

③ Knuth过去曾在其出版物中说过许多次，包括在他的1974年ACM图灵奖论文《作为艺术的计算机编程》和他的论文《带有goto语句的结构化程序》中。在他的论文《TeX的错误》中，他将该说法归功于C.A.R. Hoare。而Hoare在一封于2004年发给phobia.com的Hans Genwitz的电子邮件中说他不记得这一说法的起源了，但他可以将其归功于Dijkstra。

Common Lisp的动态特性确保了代码的灵活性，这使其易于尝试不同的方法。给定有限的时间来编写一个程序，如果你不必花很多时间出入死胡同的话，多半会写出一个高性能的版本来。在Common Lisp中，你可以尝试一种思路，如果发现其不可行就立即切换到下一个，而不必花费大量时间来使编译器得以通过你的代码并等待编译完成。你可以先写出一个函数的某个直接而低效的版本（一份代码草图）来检查你的基本思路是否可行，如果是的话再将该函数替换成一个复杂但却高效的实现。并且就算发现整个思路存在问题，你也不必将时间浪费在调整一个不再需要的函数上，这意味着你有更多的时间来找出一个更好的方法。

说Common Lisp是用于开发高性能软件的优秀语言，第二个原因在于，大多数Common Lisp实现都带有成熟的编译器，可产生相当高效的机器码。我将很快谈及如何帮助这些编译器来生成可与C编译器生成的代码相媲美的代码，但不作改进的这些实现已经比那些不成熟的实现或使用更简单的编译器或解释器的语言快得多了。另外，由于Lisp编译器在运行期可用，因此Lisp程序员拥有一些其他语言难以比拟的可能性——程序可以在运行期生成Lisp代码，然后再被编译成机器码来运行。如果生成的代码打算运行足够多次的话，那么由此带来的好处将是巨大的。或者，即便不使用运行期的编译器，闭包也给了你另一种将机器码与运行期数据混合使用的方式。例如，CL-PPCRE正则表达式库运行在CMUCL上的时候，在某些基准测试上比Perl的正则表达式引擎还要快，即便Perl的引擎是用高度优化的C写成的。这在很大程度上是因为在Perl中一个正则表达式被转化成了本质上是字节码的东西，然后被插入到了正则引擎中，而CL-PPCRE直接将正则表达式转译成了编译了的闭包树，这些闭包通过正常的函数调用机制来调用彼此。<sup>①</sup>

不过，即便使用了正确的算法和高质量的编译器，你可能仍然无法达到你所需要的原始速度。那么接下来就要思考性能分析和调优了。在Lisp中，和在任何语言中一样，关键在于首先要进行分析以找到你程序中最花时间的性能瓶颈，然后再考虑如何让这些部分提速。<sup>②</sup>

你有许多种方式来完成性能分析。语言标准提供了一些基本的工具用于测量特定的形式在执行时花了多长时间。特别是，**TIME**宏可以包装在任何形式之外，并返回由其形式所返回的任何值，不过在这之前它会向**\*TRACE-OUTPUT\***打印一条消息，指出它花费了多长时间来运行，以及它使用了多少内存。该消息的确切形式是由具体实现定义的。

可以使用**TIME**来完成一些简单粗暴的分析，缩小你搜索性能瓶颈的范围。例如，假设你有一个花费很长时间来运行的函数，并且它还调用了其他两个类似下面的函数：

```
(defun foo ()
  (bar)
  (baz))
```

<sup>①</sup> CL-PPCRE还利用了另一个我没有讨论过的Common Lisp特性，即编译器宏（compiler macro）。编译器宏是一个特殊类型的宏，它提供了优化一个特定的函数调用的机会，方法是将对该函数的调用转化成更高效的代码。CL-PPCRE为其接受正则表达式参数的函数定义了编译器宏。这个编译器宏通过在编译期解析正则表达式来优化那些带有常量正则表达式的函数调用，而不是将它们留到运行期再处理。关于编译器宏的更多信息，参见你喜爱的任何一本Common Lisp参考书中的**DEFINE-COMPILER-MACRO**部分。

<sup>②</sup> “过早地优化”中的词汇“过早”完全可以被定义成“在分析之前”。请记住就算你可以将一些代码的速度提高到几乎不需要花时间的程度，你的整个程序的性能提升也仅限于那段代码在程序运行时间中所占的比率。

如果你想要查看bar和baz究竟哪一个花了更多的时间，那么可以将foo的定义改成下面这样：

```
(defun foo ()
  (time (bar))
  (time (baz)))
```

现在你可以调用foo了，而Lisp将会打印出两份报告，一个是bar的，另一个是baz的。具体的格式是与实现相关的。下面是它们在Allegro Common Lisp中的样子：

```
CL-USER> (foo)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)      0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time 105 msec
; space allocation:
; 24,172 cons cells, 1,696 other bytes, 0 static bytes
; cpu time (non-gc) 540 msec user, 10 msec system
; cpu time (gc)     170 msec user, 0 msec system
; cpu time (total) 710 msec user, 10 msec system
; real time 1,046 msec
; space allocation:
; 270,172 cons cells, 1,696 other bytes, 0 static bytes
```

当然，如果输出中带有一个标签的话就更易于阅读了。如果你大量使用这种技术，也许值得去定义一个类似下面这样的宏：

```
(defmacro labeled-time (form)
  `(progn
    (format *trace-output* "~2&~a" ',form)
    (time ,form)))
```

如果你在foo中将**TIME**替换成labeled-time，那么将得到下面的输出：

```
CL-USER> (foo)

(BAR)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)      0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time 131 msec
; space allocation:
; 24,172 cons cells, 1,696 other bytes, 0 static bytes

(BAZ)
; cpu time (non-gc) 490 msec user, 0 msec system
; cpu time (gc)     190 msec user, 10 msec system
; cpu time (total) 680 msec user, 10 msec system
; real time 1,088 msec
; space allocation:
; 270,172 cons cells, 1,696 other bytes, 0 static bytes
```

从这些输出中，很明显可以看出foo的大多数时间都花在了baz上。

当然，如果你想要分析的Lisp形式被重复调用的话，那么来自**TIME**的输出就显得笨拙了。你可以使用函数**GET-INTERNAL-REAL-TIME**和**GET-INTERNAL-RUN-TIME**来构造你自己的测量工

具，它们返回一个按照常量 `INTERNAL-TIME-UNITS-PER-SECOND` 的值逐秒递增的数值。`GET-INTERNAL-REAL-TIME` 测量的是“挂钟时间”，也就是实际流逝的时间量，而 `GET-INTERNAL-RUN-TIME` 则测量某种具体实现所定义的值，例如 Lisp 实际执行的时间量，或是 Lisp 执行用户代码而不包括垃圾收集等内部例行公事在内的时间量。下面是一个简单而有用的分析工具，它使用了一些宏和 `GET-INTERNAL-RUN-TIME`：

```
(defparameter *timing-data* ())

(defmacro with-timing (label &body body)
  (with-gensyms (start)
    `(let ((,start (get-internal-run-time)))
       (unwind-protect (progn ,@body)
         (push (list ',label ,start (get-internal-run-time)) *timing-data*)))))

(defun clear-timing-data ()
  (setf *timing-data* ()))

(defun show-timing-data ()
  (loop for (label time count time-per %-of-total) in (compile-timing-data) do
        (format t "~3d~% ~a: ~d ticks over ~d calls for ~d per.~%" 
                %-of-total label time count time-per)))

(defun compile-timing-data ()
  (loop with timing-table = (make-hash-table)
        with count-table = (make-hash-table)
        for (label start end) in *timing-data*
        for time = (- end start)
        summing time into total .
        do
          (incf (gethash label timing-table 0) time)
          (incf (gethash label count-table 0))
        finally
          (return
            (sort
              (loop for label being the hash-keys in timing-table collect
                    (let ((time (gethash label timing-table))
                          (count (gethash label count-table)))
                      (list label time count
                            (round (/ time count)) (round (* 100 (/ time total)))))))
            #'> :key #'fifth))))
```

这个分析器可以让你将 `with-timing` 包装在任意 Lisp 形式之外。每当该形式被执行时，其开始和结束的时刻都将记录下来并关联到你提供的标签上。函数 `show-timing-data` 可以输出一个表，显示在带有不同标签的代码段中分别花费的时间，如下所示：

```
CL-USER> (show-timing-data)
84% BAR: 650 ticks over 2 calls for 325 per.
16% FOO: 120 ticks over 5 calls for 24 per.
NIL
```

你可以明显地从多个角度让这段分析代码变得更专业。此外，你的 Lisp 实现也很有可能提供了它自己的分析工具，并且由于它们具有对实现内部的访问权限，因此能够得到对用户层代码未

必可见的一些信息。

你一旦在代码中发现了瓶颈所在，就可以开始设法调优了。当然，你应当尝试的第一件事是寻找一个更有效的基本算法，这是获得最大性能提升的最佳方法。但假设你已经使用了一个适当的算法，那么接下来就是“代码精修”阶段了，即局部优化你的代码，让其绝对不做任何多余的工作。

Common Lisp中用于代码精修的主要工具是它的各种可选的声明。Common Lisp声明背后的基本思想是，它们用来向编译器提供信息以帮助其生成更好的代码。

举一个简单的例子，考虑下面这个Common Lisp函数：

```
(defun add (x y) (+ x y))
```

我在第10章里提到过，如果你比较这个Lisp函数和看起来等价的C函数之间的性能：

```
int add (int x, int y) { return x + y; }
```

那么你很可能会发现Common Lisp版本慢很多，即便当你的Common Lisp实现号称是带有高质量的原生编译器时。

这是因为Common Lisp版本的函数做了太多的事——Common Lisp编译器甚至不知道a和b的值是数字，因此必须生成代码在运行期检查。并且一旦检测出它们确实是数字，它还需要检测这些数字的类型是整数、比值、浮点数还是复数，然后派发到适当的用于实际类型的加法程序上。并且就算a和b都是整数（你所关心的情形），加法程序也不得不考虑加法的结果可能过大而无法表示成一个fixnum，即单个机器字可表示的数，从而可能不得不分配一个bignum对象。

而在C语言中，由于所有变量的类型都是声明了的，编译器精确地知道a和b将保存何种类型的值。并且由于C语言中的算术在加法的结果过大而无法用返回值的类型表示时只是简单地溢出，不做任何溢出检测，也不会在数学意义上的和过大而无法填入单个机器字时分配一个bignum对象。

这样，尽管Common Lisp代码的行为更有可能从数学意义上讲是正确的，但是C版本很可能直接被编译成一两个机器指令。不过，如果你愿意为Common Lisp编译器提供跟C编译器同样的信息，包括参数和返回值的类型，以及在通用性和错误检查方面接受类似C的妥协的话，那么Common Lisp函数也可以被编译成一两个指令。

这就是声明的用处。声明的主要用法是为了告诉编译器关于变量和其他表达式的类型。例如，你可以通过编写下面的函数来告诉编译器，add的两个参数都是fixnum：

```
(defun add (x y)
  (declare (fixnum x y))
  (+ x y))
```

其中的DECLARE表达式并非Lisp形式，相反，它是DEFUN语法的一部分，并且必须出现在函数体中其他任何代码之前。<sup>①</sup>该声明声称形参x和y传递的参数将总是fixnum的。换句话说，这是一

<sup>①</sup> 声明可以出现在引入新变量的多数Lisp形式中，例如LET、LET\*和DO家族的循环宏。LOOP有其自己的用于声明循环变量类型的语法。第20章里提过的特殊操作符LOCALLY，就是专门用来创建一个可书写声明的作用域的。

个对编译器的承诺，并且编译器被允许生成假设你所言为真的代码。

为了声明返回值的类型，你可以将形式`(+ x y)`包装在`THE`特殊操作符中。该操作符接受一个诸如`FIXNUM`这样的类型说明符和一个形式，从而告诉编译器该形式将求值到给定的类型上。这样，为了给Common Lisp编译器相当于C编译器得到的所有关于`add`的信息，你可以写成下面这样：

```
(defun add (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

不过，即便是这个版本也需要更多的一个声明，才能让Common Lisp编译器像C编译器那样生成快速但危险的代码。`OPTIMIZE`声明被用来告诉编译器如何平衡5个量：被生成的代码的速度；运行期错误检查的程度；代码的内存使用，包括代码本身的大小和运行期的内存占用；随代码提供的调试信息的数量；编译过程本身的速度。`OPTIMIZE`声明由一个或多个列表构成，其中每个列表都含有符号`SPEED`、`SAFETY`、`SPACE`、`DEBUG`和`COMPILE-SPEED`中的一个，以及一个从0到3的数值，包括0和3在内。该数值指定了编译器应当给予对应量的相对权重，其中3代表最重要，而0意味着完全不重要。这样，为了让Common Lisp将`add`编译到跟C编译器差不多的程度，你可以写成下面这样：

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

当然，现在这个Lisp版本将只承担相当于C版本的义务了。如果传递的参数不是`fixnum`或者加法溢出了，那么结果将是数学上错误的或者更坏。另外，如果有人使用错误的参数数量来调用`add`，那么结果可能也会很糟。因此，你应该仅在你的程序已经正常工作以后才使用这类声明，并且应该只在性能分析表明它们可以带来不同的效果时才添加它们。如果没有它们的时候也能得到合理的性能，那么就不要使用它们。但如果性能分析显示你的代码中有一个真正的热点并且你需要对其调优，那么就放手去做吧。由于你可以这样去使用声明，因而很少只出于性能考虑来用C重写代码。FFI可以用来访问已有的C代码，但声明可在需要接近C的性能时使用。当然，你希望给定的一段Common Lisp代码在多大程度上接近于C和C++的性能，完全取决于你想让它们有多像C代码。

Lisp中内置的另一个代码调优工具是函数`DISASSEMBLE`。该函数的确切行为是与实现相关的，因为它依赖于具体实现编译代码的方式——是否编译成机器码、字节码或其他的某种形式。但其基本思想是，它可以向你展示编译一个指定的函数后所生成的代码。

于是，你可以使用`DISASSEMBLE`来查看声明是否对生成的代码产生了任何效果。并且如果你的Lisp实现使用了一个原生编译器，同时你还懂你所在平台上的汇编语言的话，那么就可以精确地看到当你调用一个函数时究竟发生了什么。例如，你可以使用`DISASSEMBLE`来感受没有声明的第一个版本的`add`和最终版本之间的区别。首先，定义并编译最初的版本。

```
(defun add (x y) (+ x y))
```

然后在REPL中用该函数的名字来调用DISASSEMBLE。在Allegro中，它可以显示类似下面的由编译器所生成的类似汇编语言的代码输出：

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x737496f4:
 0: 55          pushl  ebp
 1: 8b ec        movl   ebp,esp
 3: 56          pushl  esi
 4: 83 ec 24    subl   esp,$36
 7: 83 f9 02    cmpl   ecx,$2
10: 74 02       jz    14
12: cd 61       int   $97    ; SYS:::TRAP-ARGERR
14: 80 7f cb 00 cmpb  [edi-53],$0      ; SYS:::C_INTERRUPT-PENDING
18: 74 02       jz    22
20: cd 64       int   $100   ; SYS:::TRAP-SIGNAL-HIT
22: 8b d8        movl  ebx,eax
24: 0b da        orl   ebx,edx
26: f6 c3 03    testb bl,$3
29: 75 0e        jnz   45
31: 8b d8        movl  ebx,eax
33: 03 da        addl  ebx,edx
35: 70 08        jo    45
37: 8b c3        movl  eax,ebx
39: f8           clc
40: c9           leave
41: 8b 75 fc    movl  esi,[ebp-4]
44: c3           ret
45: 8b 5f 8f    movl  ebx,[edi-113]    ; EXCL:::+_2OP
48: ff 57 27    call  *[edi+39]      ; SYS:::TRAMP-TWO
51: eb f3        jmp   40
53: 90           nop
; No value
```

很明显，其中做了很多事。如果你熟悉x86汇编语言的话，就很可能看出具体的内容。现在编译下面的带有完整声明的add版本。

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

现在再次反汇编add并查看这些声明是否产生了什么效果。

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x7374dc34:
 0: 03 c2        addl  eax,edx
 2: f8           clc
 3: 8b 75 fc    movl  esi,[ebp-4]
```

```

6: c3          ret
7: 90          nop
; No value

```

嗯，看来确实有效果。



## 32.4 交付应用程序

另一个对于实践有重要意义但我却没有在本书谈论过的主题，是如何交付用Lisp编写的软件。我忽略该主题的主要原因是有许多种方式可以做到这点，并且具体哪一种是最好的取决于你需要交付的软件类型、目标用户以及所使用的Common Lisp实现。在本节里，我将对其中一些不同的选择做一个概括。

如果你在编写打算共享给其他Lisp程序员的代码，那么发行它们的最直接方式就是提供源代码。<sup>①</sup>你可以将一个简单的库以单个源代码文件的形式来发布，然后程序员们可以用LOAD将其加载到他们的Lisp映像里，或者有可能先用COMPILE-FILE将其编译然后再加载。

那些跨越多个源文件的更加复杂的库和应用提出了更大的挑战——为了加载和编译这些代码，所有文件都需要以正确的顺序来编译和加载。例如，一个含有宏定义的文件必须在你编译任何使用了这些宏的文件之前加载，而一个含有DEFPACKAGE形式的文件也必须在用到该包的任何文件甚至成为READ之前被加载。Lisp程序员将其称为“系统定义”问题，并通常使用所谓的“系统定义机制”或“系统定义工具”来处理它们，后者类似于make或ant这样的构建工具。跟make和ant相似，系统定义工具允许你指定不同文件之间的依赖关系，并帮助你以正确的顺序来加载和编译文件，其间试图只做必要的工作，例如只重新编译那些发生了改变的文件。

近年来，最为广泛使用的系统定义工具是ASDF，它的全称是Another System Definition Facility。<sup>②</sup>ASDF背后的基本思想是，你在ASD文件中定义系统，然后ASDF提供了一些系统上的操作，包括加载或编译它们等。一个系统也可以定义成依赖于其他的系统，后者将在必要时被加载。例如，下面给出了html.asd的内容，它是来自第31章和第32章的FOO库的ASD文件：

```

(defpackage :com.gigamonkeys.html-system (:use :asdf :cl))
(in-package :com.gigamonkeys.html-system)

(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"

```

<sup>①</sup> COMPILE-FILE所产生的FASL文件是与实现相关的，并且不一定能在同一个Common Lisp实现的不同版本间兼容。这样，使用它们就不是分发Lisp代码的一种良好方式了。它们可用于为特定实现的已知版本里运行的应用程序提供补丁。追加一个补丁的方法就是加载这个FASL文件，而由于FASL可以包含任意代码，它可被用来通过提供新的代码定义来升级已有的数据。

<sup>②</sup> ASDF最初是由SBCL的开发者之一Daniel Barlow所编写的，并且长久以来就是SBCL的一部分，也以独立库的形式来分发。它最近已经被采纳并包含在诸如OpenMCL和Allegro等其他实现中。

```

:description "HTML and CSS generation from sexps."
:long-description ""
:components
((:file "packages")
 (:file "html" :depends-on ("packages"))
 (:file "css" :depends-on ("packages" "html")))
:depends-on (:macro-utilities))

```

如果你在变量`asdf:*central-registry*`<sup>①</sup>中所列出的目录里添加了一个到该文件的符号链接，那么你就可以通过键入

```
(asdf:operate 'asdf:load-op :html)
```

来以正确的顺序编译和加载文件`packages.lisp`、`html.lisp`以及`html-macros.lisp`，并首先保证`:macro-utilities`系统已被编译和加载。对于其他的ASD文件示例，你可以查看本书的源代码，即来自每个实用章节的代码都被定义成一个系统，并带有ASD文件中表达的适当的跨系统依赖关系。

你将发现大多数免费和开源的Common Lisp库都带有一个ASD文件。它们中的一些还有可能使用其他系统定义工具，例如相对比较古老的MK:DEFSYSTEM，或者甚至是库作者自己设计的工具，但整个流行趋势是向ASDF发展的。<sup>②</sup>

当然，尽管ASDF让Lisp程序员可以容易地安装Lisp库了，但它对于你想要给不了解或不关心Lisp的最终用户打包一个应用程序却不能带来多大的帮助。如果你正在分发一个纯面向最终用户的应用程序，那么你必定要提供一些东西，让用户可以下载、安装和运行而无需知道任何有关Lisp的知识。你不能期待他们能独立地下载并安装一个Lisp实现。并且你还应该让他们能像运行其他应用程序一样运行你的程序——通过双击Windows或OS X上的一个图标，或者在Unix命令行下输入该程序的名字即可。

不过，C程序通常依赖于作为操作系统一部分的那些构成C“运行时环境”的特定共享库（在Windows上是DLL），Lisp程序与此不同，它必须包含一个Lisp运行时环境，也就是当你启动Lisp时运行的那个程序，其中的某些功能可能是你的应用程序所不需要的。

更复杂的问题在于，“程序”这个概念在Lisp中并没有很好的定义。正如你在本书中所看到的，在Lisp中开发软件是一个不断修改你的Lisp映像中的定义和数据集的增量过程。“程序”只是映像所达到的一个特定状态，通过加载含有创建适当定义和数据的代码的.lisp或.fasl文件来改变。随后你可以将Lisp应用程序分发成一个Lisp运行时环境外加一组FASL文件，以及一个可执行程序负责启动运行时环境、加载FASL文件并以某种方式调用适当的启动函数。不过，由于事实上加载FASL可能需要花很多时间，尤其是当你需要做一些计算来设置环境的状态时，因此多数Common Lisp实现都提供了一种导出映像文件的方式，即将一个运行中的Lisp环境的状态保存在

① 在不支持符号链接的Windows上，其工作方式略有不同，但也是差不多的。

② 另一个工具ASDF-INSTALL，构建在ASDF和MK:DEFSYSTEM之上，提供了一种从网络上自动下载和安装库的简单方式。学习ASDF-INSTALL的最佳途径是阅读Edi Weitz的“A tutorial for ASDF-INSTALL” (<http://www.weitz.de/asdf-install/>)。

一个称为“映像文件”(image file)或“核心”(core)的文件里。当一个Lisp运行时环境启动时，它做的第一件事就是加载映像文件，这比通过加载FASL文件来重建所有状态花费的时间要少得多。

正常情况下，这个映像文件是一个只含有语言所定义的标准包和具体实现所提供的附加包的默认映像。但在多数实现里，你都有某种方法可以指定一个不同的映像文件。这样，不必将一个应用程序打包成一个Lisp运行时环境外加一堆FASL，你可以将其打包成一个Lisp运行时环境外加单个映像文件包含你应用程序的所有定义和数据。然后你所需要的就是一个程序，它可以用适当的映像文件来启动Lisp运行时环境，并调用作为该应用程序入口点的那个函数。

这就是事情开始依赖于具体实现和操作系统的地方了。一些Common Lisp实现，尤其是诸如Allegro和LispWorks这样的商业实现，提供了用来构建这样一个可执行程序的工具。例如，Allegro的企业版提供了一个函数`excl:generate-application`，它可以创建出一个目录，其中含有共享库形式的Lisp运行时环境、一个映像文件以及一个用给定映像启动Lisp运行时环境的可执行程序。类似地，LispWorks专业版中的“delivery”机制允许你构建所有程序的单一可执行文件。在Unix上，通过使用多种免费和开源的实现，你也可以从本质上达到同样的效果，只是使用一个shell脚本来开始可能会更容易一些。

在Mac OS X上，事情甚至变得更奇妙了。由于Mac OS X上的所有应用程序都打包成了.app应用程序捆绑(bundle)，其本质上就是带有特定结构的一个目录，因此将Lisp应用程序的所有部分打包成一个可以双击的.app应用程序捆绑完全没有任何困难。Mikel Evins的Bosco工具可以让创建OpenMCL上的应用程序的.app捆绑变得更容易。

当然，近年来的另一种分发应用程序的流行方式是将其作为服务器端应用程序。这是Common Lisp真正擅长的方式。你可以选取一种最适合你的操作系统和Common Lisp实现组合，并且不需要担心如何打包面向最终用户的应用程序。并且Common Lisp的交互式调试和开发特性也可以调试和升级一个运行中的服务器，这在那些不够动态的语言里要么根本是不可能的，要么也要求你为此构建大量的特定基础设施才可行。

## 32.5 何去何从

就这么多了。欢迎来到Lisp的精彩世界。现在你的最佳选择（如果你还没有开始的话）就是开始亲手编写你自己的Lisp代码。选择一个令你感兴趣的项目，然后用Common Lisp来完成它。然后再做另一个。就这样不断地进行下去。

不过，如果你还需要更进一步的指点，本节提供了一些可供参考的地方。对于初学者来说，可以查看本书位于<http://www.gigamonkeys.com/book/>的Web站点，这里有那些实用章节的源代码、勘误以及指向Web上其他Lisp资源的链接。

另外，除了我在32.1节所提到的站点之外，你可能还需要浏览Common Lisp HyperSpec（也称为HyperSpec或CLHS），一个ANSI语言标准的HTML版本，它由Kent Pitman制作并通过LispWorks发布在<http://www.lispworks.com/documentation/HyperSpec/index.html>。HyperSpec并不是

一个学习向导，但在你未从ANSI购买语言标准的打印副本的情况下，它是你可以获得的关于这门语言的权威指导，并且它更适合于日常使用。<sup>①</sup>

如果你想要接触其他Lisp程序员，那么Usenet的com.lang.lisp新闻组和Freenode IRC网络上的#lisp频道就是两个最主要的会面场所。还有一些Lisp相关的博客，其中的多数都被聚合到了位于<http://planet.lisp.org/>的Planet Lisp站点上。

并且你要保持关注所有这些论坛上关于你所在区域里的本地Lisp用户组的公告——在过去的几年里，Lisp用户群正在世界上的许多城市里陆续出现，从纽约到奥克兰，从科隆到慕尼黑，从日内瓦到赫尔辛基。

如果你想要继续啃书本，那么这里有一些推荐书目。如果你想要一本放在桌面上又厚又精美的参考书，可以选择David Margolies的*ANSI Common Lisp Reference Book* (Apress, 2005年)。<sup>②</sup>

想了解Common Lisp对象系统的更多内容，你可以从Sonya E. Keena的*Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* (Addison-Wesley, 1989年) 开始。然后如果你真的想要成为一个对象技术专家或者只是想要激发灵感的话，可以阅读Gregor Kiczales、Jim des Rivières和Daniel G. Bobrow的*The Art of the Metaobject Protocol* (MIT Press, 1991年)。这本书也称为AMOP，它说明了元对象协议是什么以及你为何需要它，并且还描述了一个被许多Common Lisp实现所支持的元对象协议的事实标准。

两本覆盖通用Common Lisp技术的书籍是Peter Norvig的*Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* (Morgan Kaufmann, 1992年)以及Paul Graham的*On Lisp: Advanced Techniques for Common Lisp* (Prentice Hall, 1994年)。前者提供了对各种人工智能相关技术的全面介绍，其中教授了许多关于如何编写良好Common Lisp代码的内容，而后者在对宏的处理上尤为优秀。

如果你是那种对事情的工作原理究根问底的人，那么Christian Queinnec的*Lisp in Small Pieces* (Cambridge University Press, 1996年) 提供了编程语言理论和使用Lisp实现技术的完美融合。尽管该书主要集中在Scheme而非Common Lisp上，但两者应用的是同样的原则。

那些更喜欢用理论的眼光来看事物的读者，或者读者只是想知道作为MIT的计算机学院新生是什么感觉，可以去看Harold Abelson、Gerald Jay Sussman和Julie Sussman的*Structure and Interpretation of Computer Programs*第二版 (MIT Press, 1996年)，这是使用Scheme来教授重要编程概念的经典计算机科学文献。每个程序员都可以从该书中受益良多，不过要注意的是Scheme

<sup>①</sup> SLIME带有一个Elisp库，它允许你自动跳转到任何由标准所定义的名字在HyperSpec中的对应项上。你还可以下载一份HyperSpec的完整副本以实现本地的离线浏览。

<sup>②</sup> 另一本经典的参考书是Guy Steele的*Common Lisp: The Language* (Digital Press, 1984和1990年)。该书的第一版，也称为CLtL1，在很多年里都是该语言的事实标准。在等待官方的ANSI标准完成时，Guy Steele——标准化委员会的成员之一，决定发布第二版以填补CLtL1和未来标准之间的鸿沟。该书的第二版，现在称为CLtL2，在本质上是标准化委员会在接近完成的一个特定时间里的工作的快照，接近但并不等于最终的标准。因此，CLtL2与最终的标准在一些方面还有所区别，这使得它不是一个很好的日常参考。不过，它是一份极其有用的历史文献，尤其是它说明了某些特性在完成之前被标准所丢弃从而没能成为标准的前因后果，以及为何某些特性采用了标准中所定义的方式。

和Common Lisp之间有许多重要的区别。

一旦你习惯了Lisp的思维方式，就可能想了解它的更多内容。没人可以在不懂Smalltalk的情况下号称自己真正理解了面向对象，因此你可能需要从Adele Goldberg和David Robson的*Smalltalk-80: The Language* (Addison Wesley, 1989年)开始，它是对Smalltalk核心的标准介绍。在这之后，Kent Beck的*Smalltalk Best Practice Patterns* (Prentice Hall, 1997年)是一本面向Smalltalk程序员的完美教材，其中的许多内容都可以应用在任何面向对象的语言里。

另一方面，Bertrand Meyer的*Object-Oriented Software Construction* (Prentice Hall, 1997年)给出了Eiffel发明人对静态语言思想的杰出解释，Eiffel是Simula和Algol的一个后裔，常常被人忽视。它含有很多值得思考的东西，即便是对于那些工作在诸如Common Lisp这类动态语言的程序员来说也是这样。特别是Meyer关于契约式设计 (Design By Contract) 的思想对于应该如何使用Common Lisp的状况系统也有很大参考价值。

尽管不是关于计算机的，但James Surowiecki的*The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations* (Doubleday, 2004年)一书却给出了下面这个问题的一个很好的答案：“如果Lisp这么好的话，那为什么不是每个人都用它呢？”具体参见第53页开始的一节“Plank-Road Fever”。

最后，为了寻找一些乐趣，并且也是为了了解Lisp和Lisp程序员对黑客文化的影响，可以看看Eric S. Raymond所编译的*The New Hacker's Dictionary*第三版 (MIT Press, 1996年)，该书基于Guy Steele早先所编辑的*The Hacker's Dictionary* (Harper & Row, 1983年)。

但是不要让所有这些建议影响你的编程，真正学好一门语言的唯一方法是去实际使用它。如果你已经学到这里了，那么肯定已经准备好要这样做了。那么祝你玩得开心！