第24章

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

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。[1]

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。[2]

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[1]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进行这些按位处理。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。[2]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》），用来表达一个多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：[1]

```
(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（endianness）以及有符号或无符号的格式。

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

---

① 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。[①]

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。[②]

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

① 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。

② 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。

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 * %0xxxxxxx
```

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

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号。

```
(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。[1]

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。

```
(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))
```

① 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**DEFCLASS**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，[①]这可以通过使用第8章的with-gensyms宏来实现。

① 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))))))
```

24.10 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)))
```

24.12 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**DEFMATHOD**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**DEFTRUCTURING-BIND**来取出每个元素的**REST**部分。[①]

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)
```

① 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。[1]

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。

① 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。