第25章

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。[①]

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。

① 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 [CDDB]）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。[1]

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。

---

① 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。[①]

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))
```

① 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。[1]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。

---

① 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#xfeff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
    (#xfeff '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 #xfeff)
    (write-value
     'generic-string out string
     :length (length string)
     :character-type (ucs-2-char-type #xfeff))))

(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 #xfeff)
    (write-value
     'generic-terminated-string out string
     :terminator terminator
     :character-type (ucs-2-char-type #xfeff)))
```

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：

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

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”）。[①]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。

① 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。

```
(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了。[1]

① 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。[①]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))))
```

① 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)))
```

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。[①]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

① 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，这还只是一个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。

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具体类。

```
(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TXXX时，返回适当的类名。

```
(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "TXXX") :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)))))))
```

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" "TXXX") :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。[①]

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

① 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用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" "Polsk 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。