

第十五章 Unicode

- ↓ 第十五章 Unicode
 - ↓ 15.1 制作字符
 - ↓ 15.2 字符语意的效果

如果你还不知道什么是 **Unicode**，那么你很快就会知道了——即使你略过本章，你也会知道——因为利用 **Unicode** 工作日益成为必须。（有些人认为它是一个必须有的恶魔，但实际上它更象是必须存在的好人。不管怎样，它都是必须有的痛苦。）

从历史来看，人们制定字符集来反映在他们自己的文化环境里所需要处理的事物。因为所有不同文化的人群都是懒惰的，他们只想包含那些他们需要的符号，而排除他们不需要的符号。如果我们只和自己文化内部的其他人进行交流，那么这些符号是够用的，但是既然我们开始使用互联网进行跨文化的交流，那么我们会因为原先排除的符号而头疼。在一个美国键盘上键入音调字符已经够难的了。那么应该怎样才能写一个多语言的网页呢？

Unicode 就是答案，或者至少是一部分答案（又见 **XML**）。**Unicode** 是一种包容性的字符集，而不是排斥性的字符集。尽管人们仍然会就 **Unicode** 的各种细节争论不休（而且的确还有许多可以争吵的细节），但是 **Unicode** 的总体目标是让每个人在使用 **Unicode** 的时候都足够高兴（注：或者在一些场合，并非彻底失望。），这样大家就都愿意把 **Unicode** 当作交换文本数据的国际媒介。没有人强迫你使用 **Unicode**，就好象没有人强迫你阅读本章一样（我们希望如此）。我们仍将允许人们在他们自己的文化里使用他们原来的排它的字符集。但是那样的话（如我们所说），移植性就会有问题。

痛苦守恒定律告诉我们，如果我们在一个地方减少痛苦，那么在另外的地方肯定会增加痛苦。在 **Unicode** 的例子里，我们必须经历从字节语义到字符语义的迁移的痛苦。因为由于历史的偶然性，**Perl** 是美国人发明的，而 **Perl** 在历史上就混淆了字节和字符的概念。为了向 **Unicode** 迁移，**Perl** 必须在某种程度上分清这两个概念。

荒谬的是，我们让 **Perl** 分清楚了字符和字节的关系，而却又允许 **Perl** 程序员混淆它们的概念，而依靠 **Perl** 来保持界限，就好象我们允许程序员混淆数字和字串而是依靠 **Perl** 做必要的转换。要扩展可能性，**Perl** 处理 **Unicode** 的方法和处理其他东西的方法是一样的：做正确的事情。通常，我们要实现下面四个目标：

目标 1：

旧的面向字节的程序不能同时破坏它们原来用来运行的旧式面向字节的数据。

目标 2：

旧的面向字节的程序在合适的条件下应该能神奇地开始在新的面向字符的数据上运行。

目标 3：

程序在新的面向字符模式下应该和原来的面向字节的模式运行得一样快。

目标 4：

Perl 应该仍然是一种语言，而不是分裂成一种面向字节的 **Perl** 和一种面向字符的 **Perl**。

把它们合在一起，这些目标实际上是不可能实现的。不过我们已经非常接近了。或者，换句话说，我们仍然在向非常接近目标努力，因为这是一项正在进行的工作。随着 **Unicode** 的不断发展，**Perl** 也在不断发展。但是我们的主要目标是提供一条安全的迁移路径，这样我们在迁移的过程中只需要最少

的注意就可以了。我们是如何做的是下一节的内容。

15.1 制作字符

在 Perl 5.6 以前的版本，所有字串都被看成一个字节序列（注：你可能愿意把它们称做“八位字节”；那样挺好，不过我们认为如此这两个词几乎同义，所以我们仍然使用那个蓝领阶层的叫法）不过，Perl 5.6 以后的版本里，一个字串可能包含一些比一个字节宽的字符。我们现在不把字串看作一个字节序列，而是一个数字序列，这些数字是处于 $0 \dots 2^{32}-1$ （或者在 64 位机上是 $0 \dots 2^{64}-1$ ）之间。这些数字代表抽象的字符，而且在某种意义上数字越大，字符越宽；不过和许多语言不同的是，Perl 没有捆绑在任何特定宽度的字符形式上。Perl 使用一种变长编码（基于 UTF-8），所以这些抽象的数据可能能够，也可能不能够每字节封装一个数字。显然，字符数字 18,446,744,073,709,551,615（也就是 `\x{ffff_ffff_ffff_ffff}`）肯定不能放在一个字节里（实际上，它占了十三个字节），但是如果你的字串里的所有字符都在十进制 $0 \dots 127$ 的范围里，那么它们肯定可以包在一个字节的范围里，因为 UTF-8 在低七位空间里和 ASCII 是一样的。

Perl 只有在它认为有益的时候才使用 UTF-8，所以如果你的字串里的所有字符都落在 $0 \dots 255$ 的范围里，那么很有可能这些字符都封装在一个字节里——但是如果缺乏其他知识，你也无法确定这一点，因为 Perl 在内部根据需要在定长的 8 位字符和变长 UTF-8 字符之间进行转换。关键是你大多数时间不需要关心这些内容，因为这里的字符语意是一种不考虑表现的抽象含义。

不论什么情况下，如果你的字串包含任意大于十进制 255 的数字，那么该字串肯定是以 UTF-8 的形式存储的。更准确地说，它是以 Perl 扩展了的 UTF-8 版本存储的，我们管它叫 `utf8`，一方面是尊重那个名称的用法，另一方面也是为了敲击简单。（并且因为“真正”的 UTF-8 只允许包含 Unicode 联盟确认的字符数字。Perl 的 `utf8` 允许你包含任何你需要的字符数字。Perl 并不在乎你的字符数字是官方认可的还是你认可的。）

我们说过你在大部分时候都不用关心这些，但是人们还是愿意关心。假设你使用一个 `v` 字串代表一个 IPv4 地址：

```
$locaddr = v127.0.0.1;      # 当然是按照字节存储
$oreilly = v204.148.40.9;   # 可能以字节或者以 utf8 方式存储
$badaddr  = v2004.148.40.9  # 当然是以 utf8 方式存储
```

每个人都明白 `$badaddr` 不是一个 IP 地址。所以我们很容易认为如果 O'Reilly 的网络地址被强制表示成 UTF-8 方式，那么它就无法运行了。但是在字串里的字符都是抽象数字，而不是字节。任何使用 IPv4 地址的东西，比如 `gethostbyaddr` 函数，都自动把抽象字符数字转换成一个字节形式（并且对 `$badaddr` 会失效）。

Perl 和真实世界之间的接口需要处理表现形式的细节。现有的接口在最大的可能下都可以不用你告诉它们如何处理而做出正确的动作。但是的确有偶尔的机会需要你给一些接口以某种指导（比如 `open` 函数），而且如果你写你自己的与现实世界的接口，那么这个接口要么是聪明得能够自己分辨事物，要么至少是能够在非缺省特性的时候能够遵循指导。（注：在一些系统上可能存在一次性切换你的所有接口的方法。如果使用了 `-C` 命令行开关，（或者全局的 `$_{@WIDE_SYSTEM_CALLS}` 变量设置为 1，那么所有系统调用都会使用对应的宽字符 API。（这个特性目前只在 Microsoft Windows 上实现。）Linux 社区目前的计划是如果 `$ENV{LC_CTYPE}` 设置为“UTF-8”，那么所有接口都切换到 UTF-8 模式。其他的社区可能采纳其他方法。我们的进展也可能变化。）

因为 Perl 要关心在它自己内部维持透明的字符语意，所以你需要关心字节与字符语意的区别的唯一的地方就是你的接口。缺省时，你所有旧的连接外部世界的 Perl 接口都是面向字节的，因此它们生成和处理面向字节的数据。也就是说，在这个抽象层，你的所有字串都是范围 $0 \dots 255$ 的数字序列，所以如果在程序里没有什么东西强迫它们表示成 `utf8`，你的旧的程序就继续以面向字节的数据为处理对象，就象它们原来的样子。因此可以在上面的目标 1 上画个勾。

如果你希望你的旧程序可以处理新的面向字符的数据，那么你必须设法标记你的面向字符的接口，这样 Perl 就明白在那些接口上准备处理面向字符的数据。一旦你完成这些工作，Perl 就会自动做任何必须的转换工作以保持字符的抽象性。唯一的区别是你已经引入了一些字串到你的程序里面，这些字串标记为可能包含超过 255 的字符，因此，如果你在字串和 utf8 字串之间进行操作，那么 Perl 将在内部先把字节字串转换成 utf8 字串，然后再执行操作。通常，只有你又把它们发送回给一个字节接口的时候，utf8 字串才转换回字节字串，这个时候，如果字串包含大于 255 的字符，那么你就会有一个问题，这个问题可以用多种不同的方法来处理——取决于那个有问题的接口。因此你可以在目标 2 上也画一个勾。

有时候你想把理解字符语意的编码和那些必须以字节语意运行的编码，比如读取或者写出固定大小的块的 I/O 编码，混合起来使用。这种情况下，你可以在面向字节的编码周围放一个 `use bytes` 声明以强制它使用字节语意——甚至是在那些标记为 utf8 的字串上。然后你就要对任何必须的转换负责。不过这是一个强化目标 1 的更严格的本地读取，代价放松目标 2 的全局读取。

目标 3 大部分都实现了，原因部分是通过做字节和 utf8 表示形式之间的懒惰的转换另一部分是因为我们在实现 Unicode 里的那些比较慢的特性（比如大表里的属性查找）的时候做得比较隐蔽。

我们通过牺牲一小部分实现其他目标的接口的兼容性实现了目标 4。从某个角度来看，我们没有把 Perl 分裂成不同的两种 Perl；但是以另外一种方法来看，版本 5.6 的 Perl 是一种分裂了的版本，只是它仍然尊重以前的版本，而且我们认为人们在确信新版本能够处理他们的事物之前不会从更早的版本切换到新的版本。不过新版本总是这个样子的，所以我们允许自己在目标 4 上也打个勾。

15.2 字符语意的效果

字符语意的结果是一个典型的内建操作符将以字符为操作对象，除非它位于 `use bytes` 用法里。不过，即使在 `use bytes` 用法外面，如果该操作符的所有操作数都以 8 位字符的方式存储（也就是说，没有操作符以 utf8 方式存储），那么字符语意就和字节语意没有区别，并且操作符的结果将以 8 位的方式存储在内部。这样，只要你的程序不使用比 Latin-1 更宽的字符，那么这个程序就保留的向后的兼容性。

utf8 用法主要是一个兼容性设备，它打开分析器对 UTF-8 文本和标识的识别。它还可以用于打开一些更是处于实验阶段的 Unicode 支持特性。我们的远期目标是把 utf8 用法变成透明层（no-op）。

`use bytes` 用法可能永远不会成为透明层。它不仅对面向字节的编码是必要的，而且在一个函数上定义面向字节的封装在 `use bytes` 范围外面使用还会有副作用。在我们写这些的时候，唯一定义了封装是 `length`，不过随着时间的推移，会有更多封装出现的。要使用这样的封装，你可以说：

```
use bytes(); # 不加输入 byte 语意地装载封装器 ... $charlen = length("\x{ffff_ffff}"); # 返回 1
$bytelen = byte::length("\x{ffff_ffff}"); # 返回 7
```

在 `use bytes` 声明外边，Perl 版本 5.6 的运行特性（或者至少其期望的运行特性）是这样的：

- 字串和模式现在可以包含值大于 255 的字符；

```
use utf8;
$convergence = " ";
```

假设你有一个可以编辑 Unicode 的编辑器编辑你的程序，这样的字符通常会在文本字串中直接以 UTF-8 字符的方式出现。从现在开始，你必须在你的程序开头 开头声明一个 `use utf8` 以便允许文本中使用 UTF-8。

如果你没有 Unicode 编辑器，那么你还是可以用 `\x` 表示法声明特定的 ASCII 码扩展。Latin-1 范

围的字符可以以 `\x{ab}` 的形式或者 `\xab` 的形式写，但是 如果数字超过两位十六进制数字，那你就必须使用花括号。你可以用 `\x` 后面跟着 花括号括起来的十六进制编码来表示 **Unicode**。比如一个 **Unicode** 笑脸符号是 `\x{263A}`。在 **Perl** 里没有什么语法构造假设 **Unicode** 字符正好就是 16 位， 所以你不能用其他语言那样的 `\u263A` 来表示；`\x{263A}` 是最相近的等价物。

- 在 **Perl** 脚本里的标识符可以包含 **Unicode** 字母数字字符，包括象形文字：

```
use utf8;
$人++      # 又生了一个孩子
```

同样，你需要 `use utf8` （至少目前）才能识别你的脚本中的 **UTF-8**。目前，如果 需要使用规范化的字符形式，你得自己决定——**Perl** （还）不会试图帮助你 规范化变量名。我们建议你把你的程序规范化为正常 **C** 模式（**Normalization Form C**），因为这种形式是有朝一日可能是 **Perl** 的缺省规范化形式。参阅 www.unicode.org 获取最新的有关规范化的技术报告。

- 正则表达式现在匹配字符，而不是字节。比如，点匹配一个字符而不是一个字节。 如果 **Unicode** 协会准备批准 **Tengwar** 语言，那么（尽管这样的字符在 **UTF-8** 里 用四个字节表示），但下面的东西是匹配的：

```
"\N{TENGWAR LETTER SILME NUQUERNA}" =~ /^.$/
```

`\C` 模式用于强制一次匹配是对一个字节的（**C** 里的“**char**”，因此是 `\C`）。用 `\C` 的时候要小心，因为它会令你和你字符串的字符边界不同步，而且你可能会收到 “**Malformed UTF-8 character**” 错误。你不能在方括号里使用 `\C`，因为它不代表任何特定的字符或者字符集。

- 在正则表达式里的字符表匹配字符而不是字节，并且匹配那些在 **Unicode** 属性 数据库里声明的字符属性。因此可以把 `\w` 用于匹配一个象形文字：

```
"人" =~ /\w/
```

- 可以用新的 `\p`（匹配属性）和 `\P`（不匹配属性）构造，把命名 **Unicode** 属性和块范围用做字符表。比如，`\p{Lu}` 匹配任何有 **Unicode** 大写字符属性的字符，而 `\p{M}` 匹配任何标记字符。但字母属性可以忽略花括号，因此标记字符也可以用 `\pM` 匹配。还有许多预定义的字符表可以用，比如 `\p{IsMirrored}` 和 `\p{InTibetan}`：

```
"\N{greek:Iota}" =~ /\p{Lu}/
```

你还可以在放括号字符表里使用 `\p` 和 `\P`。（在版本 5.6 的 **Perl** 里，你需要 使用 `use utf8` 才能令字符属性正确工作。不过这个限制在将来会消失。）参阅 第五章，模式匹配，获取匹配 **Unicode** 属性的细节。

- 特殊的模式 `\X` 匹配任何扩展的 **Unicode** 序列（**Unicode** 标准中的“组合字符序列”），这时候，第一个字符是基础字符，而随后的字符是标记字符，这些标记字符附加在基础字符上。它等效于 `(?:\PM\pM*)`：

```
"o\N{COMBINING TILDE BELOW}" =~ /\X/
```

你不能在方括号里使用 `\X`，因为它可以匹配多个字符，而且它不匹配任何特定的 字符或者字符集。

- `r///` 操作符转换字符，而不是转换字节。要把所有 **Latin-1** 范围以外的字符变成一个问号，你可以说：

```
tr/\0-\x{10ffff}/\0-\xff?;/ # utf8到 latin1 字符
```

- 如果有字符输入，那么大小写转换操作使用 **Unicode** 的大小写转换表。请注意 `uc` 转换成大写，而 `ucfirst` 转换成抬头体（对那些区分这些的语言而言）。通常对应的反斜杠序列有着相

同的语意：

```
$x = "\u$word";    # 把 $word 的第一个字母改成抬头体
$x = "\U$word";    # 大写 $word
  $x = "\l$word";    # 小写 $word 的第一个字母
$x = "L$word";     # 小写 $word
```

需要小心的是，Unicode 的大小写转换表并不准备给每种实例都提供循环映射，尤其是那些大写或者抬头体的字符数和对应小写的字符数不同的语言。正如 Unicode 协会的人所说，尽管大小写属性本身是标准的，大小写映射却只是 报告性的。

- 大多数处理字符串内的位置或者长度的操作符将自动切换成使用字符位置，包括 `chop`，`substr`，`pos`，`index`，`rindex`，`sprintf`，`write`，和 `length`。有意不做切换的操作符包括 `vec`，`pack`，和 `unpack`。不在意这些东西的操作符包括 `chomp`，以及任何其他的把字符串当作一堆二进制位的操作，比如缺省的 `sort` 和处理文件名的操作符。

```
use bytes;
  $bytelen = length("I do 合气道.");    # 15 字节
no bytes;
  $charlen = length("I do 合气道.");    # 只有 9 字符
```

- `pack/unpack` 字符 “c” 和 “C” 不改变，因为它们常用于面向字节的格式。（同样，类似 C 语言里的 “char”。）不过，现在有了一个新的 “U” 修饰词可以在 UTF-8 字符和整数之间做转换：

```
pack("U*", 1, 20, 300, 4000) eq v1.20.300.4000
```

- `chr` 和 `ord` 函数处理字符：

```
chr(1).chr(20).chr(300).chr(400) eq v1.20.300.4000
```

换句话说，`chr` 和 `ord` 类似 `pack("U")` 和 `unpack("U")`，而不是 `pack("C")` 和 `unpack("C")`。实际上，后面两个语句就是你懒得想详 `use bytes` 的时候模拟 字节的 `chr` 和 `ord` 的方法。

- 最后，`scalar reverse` 倒转的是字符，而不是字节：。。。 (略)

如果你看看目录 `PATH——TOPERLLIB/unicode`，你就会找到许多定义上面语意需要的文件。

Unicode 协会规定的 Unicode 属性数据库放在文件 `Unicode.300`（用于 Unicode 3.0）。这个文件已经用 `mktables.PL` 处理成同目录下的许多小 `.pl` 文件了（以及子目录 `Is/`，`In/`，和 `To/`），这些文件或目录中的一部分会被 Perl 自动装载用以实现诸如 `\p`（参阅 `Is/` 和 `In/` 目录）和 `uc`（参阅 `To/` 目录）这样的东西。其他的文件由模块装载，比如 `use charname` 用法（参阅 `Name.pl`）。不过到我们写这些为止，还有一些文件只是放在那里，等着你给它们写一个访问模块：

```
ArabLink.pl
ArabLnkGrp.pl
Bidirectional.pl
Block.pl
Category.pl
CombiningClass.pl
Decomposition.pl
JamoShort.pl
Number.pl
To/Digit.pl
```

一个可读性更好的 Unicode 的概述以及许多超级链接都放在 `PATH_TO_PERLLIB/unicode/Unicode3.html`。

请注意，如果Unicode 协会制定了新的版本，那么这些文件中的一部分的文件名可能会变化，因此你必须四处刺探。你可以用下面的“咒语”找出 `PATH_TO_PERLLIB`:

```
%perl -MConfig -le 'print $config{Privlib}'
```

如果想找到现有所有的关于 Unicode 东西，你应该看看 Unicode 标准，版本 3.0（ISBN 0-201-61633-5）。

- 请注意，“人（Unicode）”可以用了在我们写到这些的时候（也就是说，对于版本 5.6 的 Perl），使用 Unicode 上仍然有一些注意事项。（请检查你的在线文档获取最新信息。）
- 目前的正则表达式编译器不生成多形的操作码。这就意味着在编译模式的时候就要判断某个模式是否匹配 Unicode 字符（基于该模式是否包含 Unicode 字符）而不是在匹配该模式的运行的时候。这方面需要改进成只有待匹配的字串是 Unicode 才相应需要匹配 Unicode。
- 目前没有很简单的方法标记从一个文件或者其他外部数据源读取的数据是 utf8。这方面将是近期注意的主要方面，并且在你读取这些的时候可能已经搞定了。
- 我们没有办法把输入和输出转换成除 UTF-8 以外的编码方式。不过我们准备在最近做这些事情，请检查你的在线文档。
- 把本地化设置和 utf8 一起使用会导致奇怪的结果。目前，我们准备把 8 位的区域信息用于范围在 0..255 的字符，不过我们完全可以证明这样做对那些使用超过上面范围的本地化设置是不正确的（当映射成 Unicode 的时候）。而且这样做还会运行得慢一些。我们强烈建议避免区域设置。

Unicode 很好玩——但是你得正确地定义好玩的东西。

Revision: r1.4 - 05 Sep 2005 - 15:21 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > PerlUnicode

版权 © 1999-2006 归这里所有作者。 [PostgreSQL](#) 的中文文档版权归何伟平所有。
向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)