文件传输协议的国际化

本备忘录状态

本备忘录为 Internet community 指定了一种 Internet 标准跟踪协议。它需要讨论和建议以得到改进。请查阅最新版本的“Internet 正式协议标准”（STD 1）来获得本协议的标准化状态和进程。发布本备忘录不受限制。

版权声明

概要

RFC 959 [RFC959]和 RFC 1123 第 4 节[RFC1123]定义的文件传输协议是 Internet 上使用最久和最广泛是协议之一。这个协议的主要字符集，7 位的 ASCII 代码，在 Internet 发展的早期很适用。不过，随着 Internet 的全球化，需要有比 7 位 ASCII 代码更强大的字符集来支持。

本文致力于 FTP 的国际化（I18N），它包括对 Internet community 能找到的多种字符集和语言的支持。它的完成有赖于 FTP 规范的扩展和适当的国际化支持的建议。

目录

1. 介绍

当 Internet 遍及全球的时候，对 ASCII [ASCII] / Latin-1 [ISO-8859]以外的字符集支持的需求变得越来越迫切了。对 FTP 来说，因为已有很大的安装基础，所以要想不破坏现有的客户端和服务器来满足这个需求极为重要。本文致力于这个需求。文中定义了一个解决方案，它允许在已安装的基础的内部添加新的客户端和服务器。

本文增强了文件传输协议的兼容性，解除了在客户端指令和服务器回应使用的路径名的 7 位编码的限制，推荐使用统一字符集(UCS) ISO/IEC 10646 [ISO-10646]，推荐 UCS 的传输格式(UTF) UTF-8 [UTF-8]，并为语言的谈判定义了一条新指令。

本文的推荐和 IETF 有关字符集和语言的政策中（RFC 2277 [RFC2277]）表示的推荐是一致的。

1.1 必要术语

本文中的关键单词"MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", 和"OPTIONAL" 的解释已在 BCP 14 [BCP14]中描述。

2. 国际化

文件传输协议是在 7 位的 ASCII 和 8 位的 EBCDIC 字符集占统治地位时开发出来的。今天，这些字符集已经不能支持多国系统更广泛的字符的需求了。当前使用的很多字符集提供了比 7 位的 ASCII 码更多的特性，人们认识到，应该有一种便利的方式来把这些字符集关联起来一起表示。为了全球性的工作，或者需要支持多种字符集并在它们之间进行转换，或者使用一个首选的字符集。为了确保全球性的协同工作，本文档推荐用后一种方案，在 NVT ASCII 和 EBCDIC 之外，定义一个可以被所有系统理解的独立的字符集。对 FTP 来说这个字符集就是 ISO/IEC 10646:1993。为了支持全球性的兼容能力，强烈推荐客户端和服务器在交换路径名时使用 UTF-8 编码。不过，

客户端和服务器没有义务执行任何像 STOR 或 RETR 这样操作的文件内容的转换。

存储文件的字符集仍由本地系统决定，它依赖于本地操作系统的性能。在交换路径名之前应该把它们转换到ISO/IEC 10646 格式和 UTF-8 编码。这个方案允许国际的路径名交换，仍然允许向后兼容旧系统，因为ASCII字符集的编码集位置和UTF-8的一个字节序列一样。

2.1和2.2节给出了本文推荐的国际字符集和传输编码的简要描述。关于更多的UTF-8, ISO/IEC 10646, 和 UNICODE[UNICODE]的描述超出了本文的范围，它们可以在RFC 2279 [RFC2279]中找到。

2.1 国际字符集

FTP支持的国际化字符集是ISO 10646:1993定义的统一字符集。该标准合并了许多已存在的国际、国内、社会的字符集标准。ISO/IEC 10646定义了两种形式的编码：UCS-4和UCS-2。UCS-4由四个字节（31位）编码，包含2**31个编码组合，分成128组，每组256个面，每面由256行和256列组成。UCS-2是2个字节（16位）的字符集，由第零面或基础多语言面（BMP）组成。目前，还没有编码集超出2个字节的BMP。

统一字符编码标准2.0 [UNICODE]与ISO/IEC 10646的UCS-2子集一致。统一字符编码标准2.0包括10646的全部字符，10646的修正1-7，和一些编辑与技术上的勘误。

2.2传输编码

UCS转换格式8（UTF-8）是UTF-2或UTF-FSS的转换编码，用来传输国际字符集。UTF-8消除了在解析路径名字符串时对有特定意义的字节值的使用限制，所以它是一个文件安全编码。UTF-8是UCS的8位字符编码。UFT-8的好处是兼容7位的ASCII，所以有特定含义的多个ASCII字符不会影响程序；它对同步错误有免疫力；它的编码规则容易识别；并且它有足够的空间支持大量的字符集。

UTF-8用1-6个字节长的序列表示每一个UCS字符。所有的一个字节的序列标志位是零。所有多于一个字节的序列标志位在第一个字节，是一。从标志位开始，指出UTF-8序列的字节数，后面再跟一个零。例如，对于3个字节的UTF-8序列的标志位是1110。UTF-8里的每个附加字节（继续的字节）包含一个一位，再跟一个零位，作为它们的标志位。附加字节里剩余的位用来指出UCS字符。UCS和UTF-8的关系如下表所示：

UCS-4 范围(十六进制)	UTF-8 字节序列(二进制)
00000000 - 0000007F	0xxxxxxx
00000080 - 000007FF	110xxxxx 10xxxxxx
00000800 - 0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000 - 001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000 - 03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx

	10xxxxxx
04000000 - 7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

UTF-8的另一个优点是它的单一字节序列与ASCII字符集保持一致。该特性允许只支持ASCII的旧客户端仍可以与支持UTF-8编码的新服务器进行传输。

如果把一个字符序列从一个不同的字符集传输到UTF-8的主机有时会出错，因为编码规则很不可靠。它的另一个特性就是客户端和服务器能通过简单的程序来测定字符集是否已转换成了有效的UTF-8字符集。B.1节展示了一个这种检测的编码示例。

3. 路径名

3.1 一般遵从的规范

——抛弃了对路径名交换时7位表示的限制。

——有许多操作系统允许使用空格<SP>，回车<CR>，和换行<LF>字符作为路径名的一部分。在交换路径名时使用这些特殊的命令字符会引起路径名解析不正确。这是因为FTP有关路径名的命令采用这样的格式：

COMMAND <SP> <pathname> <CRLF>

为了允许交换包含这些字符的路径名，把路径名的定义从：

<pathname> ::= <string> ; BNF格式
改为：
pathname = 1*(%x01..%xFF) ; ABNF格式 [ABNF]

为了避免包含这些特殊命令字符的路径名的解析错误，定义了下列规则：

在FTP指令和路径名之间必须只能有一个<SP>。执行时，<SP>字符跟随在初始的<SP>之后，作为路径名的一部分。比如，STOR<SP><SP><SP>foo.bar<CRLF>的路径名是<SP><SP>foo.bar。

目前的执行中，还允许使用多个<SP>字符作为指令和路径名之间的分隔符，必须确保它们仍按单个的<SP>来解析。注意：将3字符的指令（比如CWD，MKD，等）填加一个空格作为四字符的指令的作法不符合本规范。

当<CR>字符作为路径名的一部分时，必须在发送指令这前填充一个<NUL>字符。在接收到的路径名中如果包含<CR><NUL>序列，<NUL>字符必须被剥离掉。这个方法在Telnet协议 [RFC854]的第11和12页作了描述。例如，发送指令STOR<SP>foo<CR><NUL><LF>boo.bar<CRLF>，此前已经把路径名

foo<CR><LF>boo.bar变为了foo<CR><NUL><LF>boo.bar。接收方把<CR>后的<NUL>字符从最初的路径名中分离出去。

——为了保持客户端和服务器的一致，两者必须支持用UTF-8来传输和接收路径名。客户端和服务器也可以让用户额外地选择另一种表示路径名的编码。注意，设置客户端和服务器使用UTF-8以外的字符集和编码已超出了本文的范围。已经验证，这样的方法在确定的操作情况下是值得的，但它是以牺牲执行和操作的品质为代价的。

——路径名是字节序列。有效的UTF-8序列名字的编码假定也是UTF-8格式的。其他名字的字符集未被定义。除非客户端和服务器进行设置，支持一个指定的本国字符集，否则必须对有效的UTF-8字节序列进行检测，以确定路径名是否可以用UTF-8表示。

——为了避免数据丢失，在不能把路径名转换到便于使用的编码集时，客户端和服务器应该使用UTF-8编码的路径名。

——可能有时供给服务器或客户端的编码集/编码不能被测定，在此情况下，应使用自然状态的字节。

3.2 服务器遵从的规范

——服务器对FEAT指令[RFC2389]的回应必须支持UTF-8特征。UTF-8特征是一个包含精确字串“UTF8”的行。该字串不区分大小写，但应使用大写传输。FEAT指令的回应应为：

```
C> feat
S> 211- <any descriptive text>
S>  ...
S>  UTF8
S>  ...
S> 211 end
```

省略号代表包含其他特征的占位符，但不是必需的。强制对特征行进行空一格的缩排[RFC2389]。

——镜像服务器如果想要正确地映射其镜像的站点，该服务器应把从主服务器接收到的路径名进行正确的路径名字节的存储和表示。

3.3 客户端遵从的规范

——无显示的客户端无需遵从有关显示的规范。

—— 被服务器用UTF-8路径名提到的客户端，应能正确地解析UTF-8并且在有效资源的限制下显示路径名。

——客户端必须支持FEAT指令，并能识别“UTF8”特征（已在前面的3.2节定义）来测定服务器是否支持UTF-8编码。

——其他名称的标识符语义尚未定义。如果客户端检测到服务器不支持UTF-8，它应改变到适当的显示方式。客户端如何处理非UTF-8格式关系到执

行结果的质量。它可以尝试采取其他编码，给用户选择采取其他编码的机会，也可以在服务器从一个FTP过程到另一个FTP过程时存储编码。

——字形表示已超出本文讨论的范围。客户端不能将字符显示出来也关系到执行结果的质量。本文推荐采用八位的字节来进行非显示字符的通信，它们用RFC 1738[RFC1738]中定义的URL %HH格式来表示。可是，用UCS十六进制数值时，它们可能被显示为疑问标志，或其他适当的方式。

——许多现有的客户端是用本地字符集来解释8位的路径名。它们仍可用这种方法来解释无效的UTF-8 路径名。

4．语言支持

字符集工作组报告[RFC2130]建议客户端和服务器用“问候语”和“错误消息”来拒绝某种语言。RFC 2130中对使用“错误消息”条件作了规范说明，这意味着对server-PI对user-PI指令的回应会返回解释性的文本消息。

执行者应注意，FTP指令和数字回应是协议的基本元素。同样，它们的使用不受本规范的任何有关表示指导的影响。

问候语和指令回应的支持语言应该是服务器支持的缺省语言，或客户端选择的服务器支持的语言。

可能会使用[MLST]中描述的虚拟主机来达到语言支持的目的。不过，FTP服务器可能不支持虚拟服务器，或虚拟服务器设置为支持一个与语言无关的环境。为了语言特征谈判，本规范定义了一个新的LANG指令。遵守本规范的客户端和服务器必须支持LANG指令。

4．1 LANG指令

新的“LANG”指令加入到FTP指令集中，它允许server-FTP过程能够决定服务器用哪种语言来表示问候语和指令回应的最初部分。与LANG指令关联的参数应是RFC 1766[RFC1766]中定义的一种语言标记之一。如果LANG指令没有参数，服务器会使用缺省的语言。

问候语和回应在语言特征之前用服务器缺省的语言发出。[RFC2277]的第4.5节规定“缺省的语言必须是讲英语的人能理解的语言”。本规范推荐服务器缺省的语言使用ASCII对英语进行编码。文本可以用其他语言扩充。一旦特征谈判完成，server-PI必须用谈判过的语言以UTF-8编码格式返回服务器消息和指令回应的最初部分。server-PI可以使用新的谈判过的语言重新发出已发送的服务器消息。

LANG指令仅影响问候语消息和有关指令回应的解释性文本的表示。无需服务器解释协议的基本元素（FTP指令和数字回应）或在数据连接上的传输数据。

user-PI可以在FTP过程中的任何时候发出LANG指令。为了获得该指令的完整用途，应在认证之前发出。通常，它在HOST指令[MLST]之后发出。注意，HOST或REIN指令[RFC959]的发出会消除LANG的影响。user-PI应有支持进行语言谈判的UTF-8编码的能力。可应用第3节中定义的解释和翻译的指导。

虽然本规范未作强制规定，但user-PI应在LANG指令之前发送一个FEAT指令[RFC2389]。它能让user-PI检测服务器是否支持LANG指令和支持的可选语言。

为了帮助服务器检测已确认连接的客户端是支持本协议的客户端还是一个旧的客户端，user-PI必须在发出其他指令之前（除了FEAT指令[RFC2389]）发出HOST和/或LANG指令。如果user-PI发出HOST指令，并且服务器的缺省语言可接受，就无需发出LANG指令。不过，如果服务器不支持HOST指令，就要发出LANG指令。直到server-PI接收到HOST指令或LANG指令，它才能假定user-PI不遵守本规范。

4．2 LANG指令语法

LANG指令定义如下：

```
lang-command        = "Lang" [(SP lang-tag)] CRLF
lang-tag            = Primary-tag *( "-" Sub-tag)
Primary-tag         = 1*8ALPHA
Sub-tag             = 1*8ALPHA

lang-response       = lang-ok / error-response
lang-ok             = "200" [SP *(%x00..%xFF) ] CRLF
error-response      = command-unrecognized / bad-argument /
                    not-implemented / unsupported-parameter
command-unrecognized  = "500" [SP *(%x01..%xFF) ] CRLF
bad-argument        = "501" [SP *(%x01..%xFF) ] CRLF
not-implemented     = "502" [SP *(%x01..%xFF) ] CRLF
unsupported-parameter = "504" [SP *(%x01..%xFF) ] CRLF
```

“lang”指令的关键字不区分大小写，可以使用任何大小写组合，所以，“LANG”，“lang”，“Lang”和“lAnG”都是等价的指令。

可选项“Lang-tag”给出的参数指出了主要的语言标记和零或多个子标记，这些标记在[RFC1766]中定义。在[RFC1766]中描述的语言标记也是不区分大小写的。如果省略掉该参数，则server-PI必须使用服务器缺省的语言。

server-FTP对“Lang”指令的回应或是“lang-ok”，或是“error-response”。如果server-FTP支持“Lang”指令并支持“lang-tag”的某些形式，它必须发送“lang-ok”。支持的意思是：

——如果server-FTP接收到一个没有参数的“Lang”指令，它应用服务器缺省的语言返回消息和指令回应。

——如果server-FTP接收到一个“Lang”指令只有主标记参数（比如，en，fr，de，ja，zh，等），该语言它可以支持，它应该用主标记标出的语言返回消息和指令的回应。也可能server-FTP只支持与子标记组合的主标记（比如，en-US,en-UK,等）。这种情况下，server-FTP可以决定在过程期间使

用的子标记变量，而server-FTP如何决定已超出了本文讨论的范围。如果server-FTP不能决定一个子标记变量是否适用，它应返回一个“不支持的参数”（504）回应。

——如果server-FTP接收到一个包含主标记和一个或多个子标记参数的“Lang”指令，它应用它支持的语言返回消息和指令回应。可能server-FTP支持“Lang”指令的主标记参数，但是不支持一个或多个子标记，这种情况下，server-FTP可以从已实现的主标记中选择最适当的语言来返回消息的指令的回应，而server-FTP如何选择已超出了本文讨论的范围。如果服务器不能决定子标记是否合适，应返回“不支持的参数”（504）回应。

例如，如果client-FTP发送了一个“LANG en-AU”指令，server-FTP实现的语言标记有en-US和en-UK，它将决定最合适的语言标记是en-UK，并返回“200 不支持en-AU，语言设置为en-UK”。 数字回应作为协议的基本元素并没有改变。与其相关的文本仅是为了对执行的情况进行说明。

符合本规范的客户端和服务器必须支持LANG指令。不过如果旧的或不适应本规范的服务器不能识别或没有实现“Lang”指令，客户端将接收到500或502指令回应。如果“Lang”指令的参数语法不正确，服务器将发送501回应。如果“Lang”指令的参数语法正确但是不能执行，服务器将发送504回应。如上面所提到的，一个参数尽管没有严格正确地匹配语法，但是它可被视为匹配的词汇。

4．3 对LANG指令的Feat回应

支持LANG指令的server-FTP过程，和可支持的语言的消息和指令回应，必须包含在FEAT指令[RFC2389]的回应里，用一个特征行指定LANG指令被支持，并且给出可支持的语言标记列表。FEAT指令的回应格式如下：

```
Lang-feat  = SP ″LANG″ SP lang-fact CRLF
lang-fact  = lang-tag [″*″] *(″;″ lang-tag [″*″])

lang-tag   = Primary-tag *( ″-″ Sub-tag)
Primary-tag= 1*8ALPHA
Sub-tag    = 1*8ALPHA
```

lang-feat回应包含字串“LANG”，其后紧跟语言列表。“LANG”字串不区分大小写，但[RFC2389]中推荐应使用大写传送。lang-feat回应里第一个的空格是FEAT指令必需的，它必须是一个单独的空格，多或少的空格字符均不允许。lang-fect应包含server-FTP支持的lang-tag。FEAT回应必须至少要包含一个lang-tag。lang-tag应遵照前文所描述的形式。当出现可选的星号时，指出当前server-FTP的消息和回应正在使用的lang-tag。

4．3．1 feat示例

```
C> feat
S> 211- <any descriptive text>
S>  ...
S>  LANG EN*
S>  ...
S> 211 end
```

在这个例子中，server-FTP仅支持英语，它也是当前服务器用来发送消息和指令回应的语言（用星号标出）。

```
C> feat
S> 211- <any descriptive text>
S>  ...
S>  LANG EN*;FR
S>  ...
S> 211 end

C> LANG fr
S> 200 Le response sera changez au francais

C> feat
S> 211- <quelconque descriptif texte>
S>  ...
S>  LANG EN;FR*
S>  ...
S> 211 end
```

在这个例子中，在第一个FEAT指令的回应里，英语和法语是服务器支持的语言，用星号标出英语是当前server-FTP正在使用的语言。一个LANG将语言改为法语，FEAT回应显示法语是当前使用的语言。

上面的例子中，省略号指出的占位符是回应中包含的其他特征，但不是必需的。

5．安全考虑

本文对多于一个字节的字符集和新的语言谈判指令进行讨论。本文不会引起安全的问题。

6．感谢

（略——译者注）

7．术语表

BIDI——双向的缩写，从右到左或从左到右的文本的混合状态。

Character Set（字符集）—— 用来表示原文信息的字符集合，每一个字符有一个数值。

Code Set（编码集）——（参看Character Set）

I18N——单词“internationalization”的第一个字母和最后一个字母，中间有18个字母。

UCS-2——ISO/IEC10646的双八位字节统一字符集形式。

UCS-4——ISO/IEC10646的四个八位字节统一字符集形式。

UTF-8——用8个二进制位表示的UCS传输格式。

TF-16——等同于统一字符编码标准。

8．参考资料

[ABNF]

Crocker, D. and P. Overell, "Augmented BNF 语法规范扩展: ABNF", RFC 2234, 1997年十一月。

[ASCII]

ANSI X3.4:1986 编码字符集 - 7位的美国信息交换标准码(7位的ASCII)

[ISO-8859]ISO 8859.

国际标准——信息处理——8-bit 单字节编码图解字符集——第1部分：拉丁字母表No. 1 (1987) -- 第2部分：拉丁字母表No. 2 (1987) -- 第3部分：拉丁字母表No. 3 (1988) -- 第4部分：拉丁字母表No. 4 (1988) -- 第5部分：拉丁/古斯拉夫字母表(1988) -- 第6部分：拉丁/阿拉伯字母表 (1987) -- 第7部分：拉丁/希腊字母表 (1987) -- 第8部分：拉丁/希伯来字母表 (1988) -- 第9部分：拉丁字母表No. 5 (1989) -- 第10部分：拉丁字母表No. 6 (1992)

[BCP14]

Bradner, S., "RFC中用来指定需求级别的关键字", BCP 14, RFC 2119, 1997年三月。

[ISO-10646]

ISO/IEC 10646-1:1993. 国际标准——信息技术——统一多字节编码字符集（UCS）-- 第1部分：体系结构和基本的多语言平台。

[MLST]

Elz, R. and P. Hethmon, "FTP扩展", Work in Progress.

[RFC854]

Postel, J. and J. Reynolds, "Telnet协议规范", STD 8, RFC 854, 1983年五月。

[RFC959]

Postel, J. and J. Reynolds, "文件传输协议(FTP)", STD 9, RFC 959, 1985年十月。

[RFC1123]

Braden, R., "Internet主机必要条件——应用程序和支持", STD 3,

RFC 1123, 1989年十月。

[RFC1738]

Berners-Lee, T., Masinter, L. and M. McCahill, ″统一资源定位器(URL)″, RFC 1738, 1994年十二月。

[RFC1766]

Alvestrand, H., ″语言认证的标记″, RFC 1766, 1995年三月。

[RFC2130]

Weider, C., Preston, C., Simonsen, K., Alvestrand, H.,Atkinson, R., Crispin, M. and P. Svanberg, ″字符集工作组报告″, RFC 2130, 1997年四月。

[RFC2277]

Alvestrand, H., ″ IETF关于字符集和语言的政策″, RFC 2277, 1998年一月。

[RFC2279]

Yergeau, F., ″UTF-8,ISO 10646的一种传输格式″, RFC 2279, 1998年一月。

[RFC2389]

Elz, R. and P. Hethmon, ″文件传输协议的特征谈判机制″, RFC 2389, 1998年八月。

[UNICODE]

The Unicode Consortium, ″统一字符编码标准——版本2.0″, Addison Westley Developers Press,1996年七月。

[UTF-8]

ISO/IEC 10646-1:1993 AMENDMENT 2 (1996). UCS转换格式8 (UTF-8).

9．作者地址

（略——译者注）

附录A ——执行考虑

A．1 一般考虑

——执行者应确保他们的编码对潜在问题的考虑，比如当支持的扩展字符集使用一个空（NULL）字符作为字串的结束。

——执行者应知道有把非UTF-8的路径名解析为合法的UTF-8的可能性。在加密的编码中发生的可能性较小。近来一个非系统的分析发现，EUC编码的日文单词有2.7%的阅读错误；SJIS有0.0005%的阅读错误；其他编码比如ASCII或KOI-8的阅读错误是0%。这种情况在使用短路径名时发生的可能性最高。执行者可能想要查找短路径名中的多个本地字符。充满希望的是，更多的执行开始遵照UTF-8传输编码，猜测编码的情况会越来越少。

——客户端开发者应知道路径名有包含混合字符的可能（比如，//拉丁目录名/希伯来文件名）。他们应准备好处理这些字符集的双向显示（比如，

从右到左显示路径，或从左到右显示文件名）。双向显示的问题已超出了本文讨论的范围，在UNICODE2.0[UNICODE]标准中有比上面的例子更复杂的情况的双向显示的规则。也应注意，在合成时由于要在不同的点插入BIDI控制字符，路径名可以有不同的字节逻辑顺序而显示方式相同。也应注意，混合的字符集会引起字体交换的问题。

——服务器透明地从本地文件系统复制路径名，可以像本地文件一样，等同于是使用UTF-8路径名的本地文件创建者。

——服务器可以支持不同的字符集来标注文件和/或目录，这样不同的路径名可以有不同的字符集。服务器的任务是把所有目录名转换为UTF-8，但是如果转换不成功，将使用它的自然形式。

——有些服务器的操作系统不能指定字符集，但是允许管理员在FTP服务器进行设置。这些服务器可设置使用特殊的映射表（外部的或内部的）。这就允许弹性地为不同的目录定义不同的字符集。

——如果服务器的操作系统不能指定字符集，并且也不能设置FTP服务器，那么服务器就只能简单地使用自然字节来表示文件名。它们可以是ASCII或UTF-8。

——如果服务器是镜像服务器，想让它看起来像被镜像的服务器，就应把从主服务器接收到的文件名精确原样地存储。

A．2 传输考虑

——支持本规范的服务器当从旧的客户端（这个客户端不支持本规范）接收到路径名时，几乎总能判断路径名是UTF-8（参看B.1）还是其他编码集。为了支持这些旧的客户端，服务器 使用非UTF-8编码集作为它的缺省字符集，但是，服务器如何支持非UTF-8已经超出了本文讨论的范围。

——支持本规范的客户端可以通过检测服务器是否支持FEAT指令和UTF8特征（在3.2节中定义）来判断服务器是否支持UTF-8（也就是支持本规范）。如果较新的客户端检测到服务器不支持UTF-8，它将使用缺省的编码集。客户端的开发者应对与旧的服务器关联的路径名进行考虑，可以用UTF-8来存储。但是，服务器如何支持非UTF-8已经超出了本文讨论的范围。

——客户端和服务器应从任意的本地编码转换到UTF-8，或者用户可以存储UTF-8文件名。前一种方法对紧密控制的文件系统（比如PC或MAC）较容易；后一种方法对更自由形式的文件系统（比如Unix）较容易。

——为了实现交互性，注意的焦点应放在用户界面和易操作上。非交互的应用需要一致的和受限的运转状态。

——有许多应用程序在提到文件时使用的是旧有的自然文件名（比如，URL链接）。转换文件名到UTF-8会引起旧有的URL读取失败。服务器方面的一个解决方案是为这样的文件建立相关联的2个不同的路径名。对控制的文件系统可以在服务器内部实现，对自由形式的系统用符号链接实现。该方案可以为非交互的单个文件工作，在一个目录中的所有文件进行非交互传输时，将产生完全相同的复本。交互的用户会收到实际文件数量两倍的文件列表。

附录B——示例代码和例子

B．1 有效的UTF-8检测

下面的程序检测一个字节序列是否是有效的UTF-8。它通过对第一个和随后的字节进行适当的标记检测来确定其是否符合UTF-8格式。然后对UTF-8序列的数据端口进行检测，以确保它遵照适当的许可范围。注意：本程序对未分配的字符没有进行检测。

```
int utf8_valid(const unsigned char *buf, unsigned int len)
{
 const unsigned char *endbuf = buf + len;
 unsigned char byte2mask=0x00, c;
 int trailing = 0;  // trailing (continuation) bytes to follow

 while (buf != endbuf)
 {
   c = *buf++;
   if (trailing)
    if ((c&0xC0) == 0x80)  // Does trailing byte follow UTF-8 format?
    {if (byte2mask)        // Need to check 2nd byte for proper range?
      if (c&byte2mask)     // Are appropriate bits set?
       byte2mask=0x00;
      else
       return 0;
     trailing--; }
    else
     return 0;
   else
    if ((c&0x80) == 0x00)  continue;      // valid 1 byte UTF-8
    else if ((c&0xE0) == 0xC0)            // valid 2 byte UTF-8
          if (c&0x1E)                     // Is UTF-8 byte in
                                          // proper range?
           trailing =1;
          else
           return 0;
    else if ((c&0xF0) == 0xE0)           // valid 3 byte UTF-8
          {if (!(c&0x0F))                // Is UTF-8 byte in
                                         // proper range?
            byte2mask=0x20;              // If not set mask
                                         // to check next byte
```

```
            trailing = 2;}
    else if ((c&0xF8) == 0xF0)          // valid 4 byte UTF-8
          {if (!(c&0x07))               // Is UTF-8 byte in
                                        // proper range?


            byte2mask=0x30;             // If not set mask
                                        // to check next byte
            trailing = 3;}
    else if ((c&0xFC) == 0xF8)          // valid 5 byte UTF-8
          {if (!(c&0x03))               // Is UTF-8 byte in
                                        // proper range?
            byte2mask=0x38;             // If not set mask
                                        // to check next byte
            trailing = 4;}
    else if ((c&0xFE) == 0xFC)          // valid 6 byte UTF-8
          {if (!(c&0x01))               // Is UTF-8 byte in
                                        // proper range?
            byte2mask=0x3C;             // If not set mask
                                        // to check next byte
            trailing = 5;}
    else  return 0;
 }
  return trailing == 0;
}
```

B．2 转换

本节中示例的代码严密地表现了ISO 10646的运算法则，但不是效率最高的UTF-8编码的转换方案。如果想更高效，执行者应使用适当的按位运算的方法。

额外的代码示例和众多映射表可以在Unicode站点找到：HTTP://www.unicode.org 或 FTP://unicode.org。

注意：下面的转换示例假定本地操作系统支持的字符集是不同于UCS2/UTF-16的字符集。有些操作系统已经支持UCS2/UTF-16（特别是Plan 9和Windows NT），这种情况下无需把本地字符集转换成UCS。

B．2．1 从本地字符集转换到UTF-8

把本地字符集转换到UTF-8通常需要两步过程。第一步转换本地字符集到UCS，然后把UCS转换成UTF-8。

第一步过程通过操作一个包含本地字符集和相应的UCS编码的映射表来完成。比如ISO/IEC 8859-8 [ISO-8859]编码的希伯来字母“AVA”是0xE4，

相应的4字节ISO/IEC 10646编码是0x000005D5。

下一步是把UCS字符编码转换到UTF-8编码。下面的程序用来对基于UCS-4的字符编码的正确字节数进行检测和编码。

```
unsigned int ucs4_to_utf8 (unsigned long *ucs4_buf, unsigned int
                           ucs4_len, unsigned char *utf8_buf)

{
 const unsigned long *ucs4_endbuf = ucs4_buf + ucs4_len;
 unsigned int utf8_len = 0;         // return value for UTF8 size
 unsigned char *t_utf8_buf = utf8_buf; // Temporary pointer
                                       // to load UTF8 values

 while (ucs4_buf != ucs4_endbuf)
 {
  if ( *ucs4_buf <= 0x7F)    // ASCII chars no conversion needed
  {
   *t_utf8_buf++ = (unsigned char) *ucs4_buf;
   utf8_len++;
   ucs4_buf++;
  }
  else
   if ( *ucs4_buf <= 0x07FF ) // In the 2 byte utf-8 range
   {
     *t_utf8_buf++= (unsigned char) (0xC0 + (*ucs4_buf/0x40));
     *t_utf8_buf++= (unsigned char) (0x80 + (*ucs4_buf%0x40));
     utf8_len+=2;
     ucs4_buf++;
   }
   else
     if ( *ucs4_buf <= 0xFFFF ) /* In the 3 byte utf-8 range. The
                                   values 0x0000FFFE, 0x0000FFFF
                                   and 0x0000D800 - 0x0000DFFF do
                                   not occur in UCS-4 */
     {
      *t_utf8_buf++= (unsigned char) (0xE0 +
                     (*ucs4_buf/0x1000));
      *t_utf8_buf++= (unsigned char) (0x80 +
                     ((*ucs4_buf/0x40)%0x40));
      *t_utf8_buf++= (unsigned char) (0x80 + (*ucs4_buf%0x40));
      utf8_len+=3;
```

```
 ucs4_buf++;
 }
else
 if ( *ucs4_buf <= 0x1FFFFF ) //In the 4 byte utf-8 range
 {
  *t_utf8_buf++= (unsigned char) (0xF0 +
                 (*ucs4_buf/0x040000));
  *t_utf8_buf++= (unsigned char) (0x80 +
                 ((*ucs4_buf/0x10000)%0x40));
  *t_utf8_buf++= (unsigned char) (0x80 +
                 ((*ucs4_buf/0x40)%0x40));
  *t_utf8_buf++= (unsigned char) (0x80 + (*ucs4_buf%0x40));
  utf8_len+=4;
  ucs4_buf++;

 }
 else
  if ( *ucs4_buf <= 0x03FFFFFF )//In the 5 byte utf-8 range
  {
   *t_utf8_buf++= (unsigned char) (0xF8 +
                  (*ucs4_buf/0x01000000));
   *t_utf8_buf++= (unsigned char) (0x80 +
                  ((*ucs4_buf/0x040000)%0x40));
   *t_utf8_buf++= (unsigned char) (0x80 +
                  ((*ucs4_buf/0x1000)%0x40));
   *t_utf8_buf++= (unsigned char) (0x80 +
                  ((*ucs4_buf/0x40)%0x40));
   *t_utf8_buf++= (unsigned char) (0x80 +
                  (*ucs4_buf%0x40));
   utf8_len+=5;
   ucs4_buf++;
  }
  else
  if ( *ucs4_buf <= 0x7FFFFFFF )//In the 6 byte utf-8 range
   {
     *t_utf8_buf++= (unsigned char)
                    (0xF8 +(*ucs4_buf/0x40000000));
     *t_utf8_buf++= (unsigned char) (0x80 +
                    ((*ucs4_buf/0x01000000)%0x40));
     *t_utf8_buf++= (unsigned char) (0x80 +
                    ((*ucs4_buf/0x040000)%0x40));
```

```
            *t_utf8_buf++= (unsigned char) (0x80 +
                           ((*ucs4_buf/0x1000)%0x40));
            *t_utf8_buf++= (unsigned char) (0x80 +
                           ((*ucs4_buf/0x40)%0x40));
            *t_utf8_buf++= (unsigned char) (0x80 +
                           (*ucs4_buf%0x40));
            utf8_len+=6;
            ucs4_buf++;

          }
 }
 return (utf8_len);
}
```

B．2．2 从UTF-8转换到本地字符集

把UTF-8编码转换到本地字符集就是对上面的程序进行逆操作。第一步把UTF-8编码转换到UCS-4字符集，然后通过一个映射表把UCS-4转换到本地字符集。

为了把UTF-8转换到UCS-4，在UTF-8序列里的自由位（它们没有说明UTF-8序列大小或表示附加的字节数）被连接成一个位串。这些位从最小的有意义位开始被分配到一个四字节的序列里。四字节序列里未分配的位被零填充。下面的程序是把UTF-8转换成UCS-4的代码：

```
int utf8_to_ucs4 (unsigned long *ucs4_buf, unsigned int utf8_len,
                  unsigned char *utf8_buf)
{

const unsigned char *utf8_endbuf = utf8_buf + utf8_len;
unsigned int ucs_len=0;

 while (utf8_buf != utf8_endbuf)
 {

  if ((*utf8_buf & 0x80) == 0x00)  /*ASCII chars no conversion
                                     needed */
  {
   *ucs4_buf++ = (unsigned long) *utf8_buf;
   utf8_buf++;
   ucs_len++;
  }
  else
```

```
if ((*utf8_buf & 0xE0)== 0xC0) //In the 2 byte utf-8 range
{
  *ucs4_buf++ = (unsigned long) (((*utf8_buf - 0xC0) * 0x40)
                + ( *(utf8_buf+1) - 0x80));
  utf8_buf += 2;
  ucs_len++;
}
else
  if ( (*utf8_buf & 0xF0) == 0xE0 ) /*In the 3 byte utf-8
                                      range */
  {
  *ucs4_buf++ = (unsigned long) (((*utf8_buf - 0xE0) * 0x1000)
               + (( *(utf8_buf+1) -  0x80) * 0x40)
               + ( *(utf8_buf+2) - 0x80));
   utf8_buf+=3;
   ucs_len++;
  }
  else
   if ((*utf8_buf & 0xF8) == 0xF0) /* In the 4 byte utf-8
                                      range */
   {
    *ucs4_buf++ = (unsigned long)
                  (((*utf8_buf - 0xF0) * 0x040000)
                  + (( *(utf8_buf+1) -  0x80) * 0x1000)
                  + (( *(utf8_buf+2) -  0x80) * 0x40)
                  + ( *(utf8_buf+3) - 0x80));
    utf8_buf+=4;
    ucs_len++;
   }
   else
    if ((*utf8_buf & 0xFC) == 0xF8) /* In the 5 byte utf-8
                                       range */
    {
     *ucs4_buf++ = (unsigned long)
                   (((*utf8_buf - 0xF8) * 0x01000000)
                   + ((*(utf8_buf+1) - 0x80) * 0x040000)
                   + (( *(utf8_buf+2) -  0x80) * 0x1000)
                   + (( *(utf8_buf+3) -  0x80) * 0x40)
                   + ( *(utf8_buf+4) - 0x80));
     utf8_buf+=5;
     ucs_len++;
```

```
        }
        else
         if ((*utf8_buf & 0xFE) == 0xFC) /* In the 6 byte utf-8
                                            range */
         {
           *ucs4_buf++ = (unsigned long)
                         (((*utf8_buf - 0xFC) * 0x40000000)
                          + ((*(utf8_buf+1) - 0x80) * 0x010000000)
                          + ((*(utf8_buf+2) - 0x80) * 0x040000)
                          + (( *(utf8_buf+3) -  0x80) * 0x1000)
                          + (( *(utf8_buf+4) -  0x80) * 0x40)
                          + ( *(utf8_buf+5) - 0x80));
           utf8_buf+=6;
           ucs_len++;
         }

 }
return (ucs_len);
}
```

B.2.3 ISO/IEC 8859-8示例

本例示范了将ISO/IEC 8859-8字符集映射为UTF-8的方法，和逆映射的方法。如前面所提到的，希伯来字母“VAV”从ISO/IEC 8859-8字符编码0xE4转换到相应的4字节ISO/IEC 10646编码，是通过查找一个转换/映射文件的简单操作完成的。

UCS-4字符编码转换到UTF-8的方法是使用前面描述的ucs4_to_utf8程序：

1．因为UCS-4字符在0x80和0x07FF之间，所以它被映射为2字节的UTF-8序列。

2．第一个字节定义为(0xC0 + (0x000005D5 / 0x40)) = 0xD7。

3．第二个字节定义为(0x80 + (0x000005D5 % 0x40)) = 0x95。

UTF-8编码转换到UCS-4的方法是使用前面描述的utf8_to_ucs4程序：

1．由序列的第一个字节判断，“&”操作符跟随的值0xE0，经运算得到0xC0(0xD7 & 0xE0 = 0xC0)，所以UTF-8是一个2字节序列。

2．四字节的UCS-4字符编码的运算：(((0xD7 - 0xC0) * 0x40) + (0x95 -0x80)) = 0x000005D5。

最后，UCS-4字符编码转换成ISO/IEC 8859-8字符编码（通过ISO/IEC 8859-8和UCS-4匹配的映射表），产生希伯来字母“VAV”的最初编码0xE4。

B．2．4 厂商代码页示例

该例示范了到UTF-8的代码页映射和相反的厂商代码页的映射。两个厂

商之间的代码页映射与上面描述的方法非常类似。比如，把泰国语标准TIS 620-2533映射为PC和MAC代码页，泰国语字母“SO SO”在这两个平台上的字符编码都是0xAB。这个字符可以通过一个转换/映射文件将其映射为UCS-4字符，生成的UCS-4编码是0x0E0B。

UCS-4字符编码转换为UTF-8的方法是使用前面描述的ucs4_to_utf8程序：

1．因为UCS-4字符在0x0800和0xFFFF之间，它将被映射为3字节的UTF-8序列。

2．第一个字节定义为(0xE0 + (0x00000E0B / 0x1000) = 0xE0。

3．第二个字节定义为(0x80 + ((0x00000E0B / 0x40) % 0x40))) = 0xB8。

4．第三个字节定义为(0x80 + (0x00000E0B % 0x40)) = 0x8B。

UTF-8编码转换回UCS-4的方法是使用前面描述的utf8_to_ucs4程序：

1．由序列的第一个字节判断，“&”操作符跟随的值0xF0，经运算得到0xE0 (0xE0 & 0xF0 = 0xE0)，所以UTF-8是一个3字节序列。

2．四字节的UCS-4字符编码的运算：(((0xE0 - 0xE0) * 0x1000) + ((0xB8 - 0x80) * 0x40) + (0x8B -0x80) = 0x0000E0B。

最后，UCS-4字符编码转换成PC或MAC代码页字符代码（通过与UCS-4匹配的代码页的映射表），产生泰国语字母“SO SO”的最初编码0xAB。

B．3 高品质转换服务器的伪代码

```
if utf8_valid(fn)
  {
  attempt to convert fn to the local charset, producing localfn
  if (conversion fails temporarily) return error
  if (conversion succeeds)
  {
    attempt to open localfn
    if (open fails temporarily) return error
    if (open succeeds) return success
  }
  }
attempt to open fn
if (open fails temporarily) return error
if (open succeeds) return success
return permanent error
```

完整的版权声明

（略——译者注）

原文：RFC 2640《Internationalization of the File Transfer Protocol》
译者：comehope 2002年6月
博客：http://www.comehope.com