UTF-8 遍地开花

宣言

1. 本文的目的

本文档包含特殊字符。若你的浏览器不支持渲染，可能会显示为问号、方框或者其他符号。

我们的目标是促进 UTF-8 编码的使用和支持，并推广它成为内存、磁盘存储和通信等用途的默认文本字符串编码。我们相信，这种做法可以提高性能，降低软件的复杂性，并有助于避免多种 Unicode 相关的 bug。我们建议 Unicode（或广义来说，所有文本）的其他编码方式只应该用于少量需要性能优化的情况，主流用户应当避免接触。

一个具体的例子是非常流行的 UTF-16 编码（在 Windows 世界常常误称为“宽字符”或干脆称为“Unicode”）。我们认为它除了在 ICU 之类的专用文本处理库中可以使用以外，不应该在其他程序库的 API 中有一席之地。

本文还建议选择 UTF-8 作为 Windows 应用程序内部的字符串表示方式，尽管由于原生 API 不支持 UTF-8 和历史原因，很少有人采用这种做法。我们认为，即使在这个平台上，本文所述论点的好处依旧胜过缺乏原生支持的所带来的坏处。此外，我们建议永远抛弃“ANSI 代码页”等概念，因为用户应当有权在任何文本字符串中混用任何数量的语言。

在整个行业中，许多本地化相关的 bug 都可归咎于程序员对 Unicode 缺乏认知。然而，我们相信一个不是专注于文本本身的应用程序，理应有着可以无视编码问题的架构。例如，一个文件复制程序不应特地为了支持非英语文件名而用其他方式实现。在这份宣言中，我们也将解释，程序员如果不想关心 Unicode 的所有复杂细节或者字符串内有什么，应该做哪些事情。

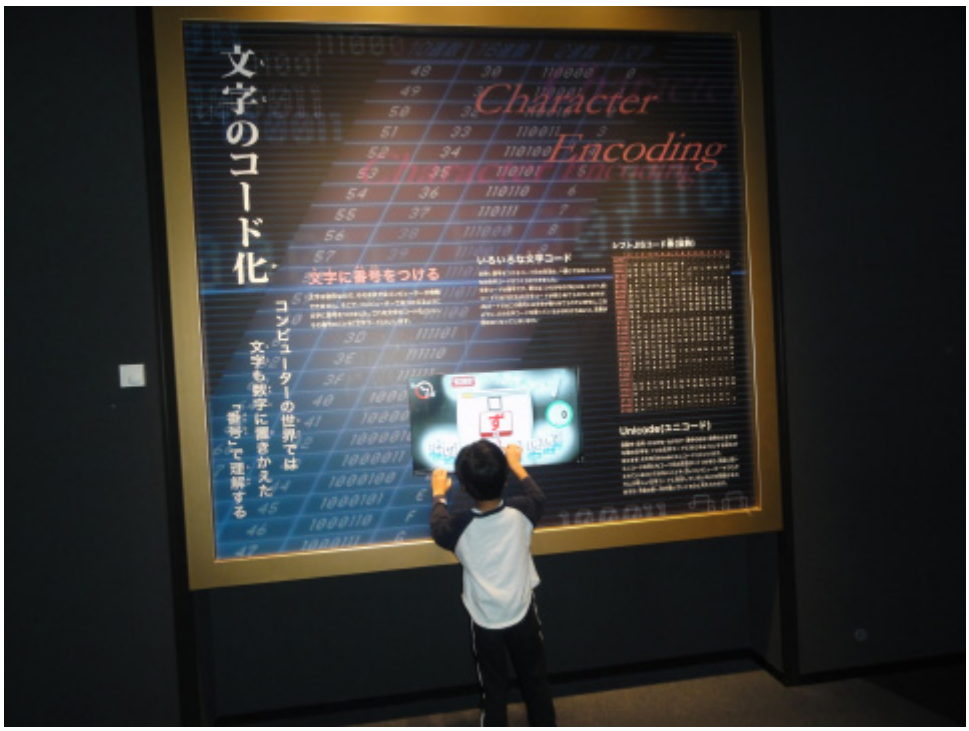
此外，我们想表明，在文本处理场景中，不应过分看重对 Unicode 码位的计数和遍历。许多开发者错误地认为码位就是 ASCII 字符的升级版。这产生了如 Python 字符串的 O(1) 码位访问之类的软件设计决策。但事实却是，Unicode 本质上更为复杂，**Unicode 字符**这个概念没有统一的定义。我们认为没有特别的理由把它定义为 Unicode 码位，而不是 Unicode 字素群、编码单元，或甚至是一些语言的词语。另一方面，把 UTF-8 的编码单元（字节）作为

文本的基本单元似乎很有用，比如在解析常用的文本数据格式有优势。UTF-8 的特性造就了这个结果。字素、编码单元、码位等 Unicode 相关的术语将在第 5 节中解释。对已编码的文本字符串进行操作将在第 7 节中讨论。

2. 背景

1988 年，Joseph D. Becker 发布了第一个 Unicode 提议草案。他的设计天真地建立在 16 比特就足够表示一个字符的假设上。1991 年，Unicode 标准（又名“统一码”、“万国码”）的第一个版本发布，但限制码位为 16 比特长。在随后的几年里，许多系统都增加了 Unicode 支持，并切换到了 UCS-2 编码。Unicode 尤其吸引当时的新技术，如 Qt 框架（1992）、Windows NT 3.1（1993）与 Java（1995）。

然而，人们很快发现每个字符 16 比特并不能满足 Unicode 的需求。1996年，为了确保现有的系统可以支持非 16 比特的字符，人们发明了 UTF-16。但是，它恰恰偏离了选择 16 比特定长编码的初衷。目前，Unicode 收录了超过 109449 个字符，其中大约有 74500 个中日韩表意文字。



名古屋市科学馆。Vadim Zlotnik 摄。

微软经常误把“Unicode”和“宽字符”作为“UCS-2”和“UTF-16”的代名词。此外，由于 Windows API 不能接受 UTF-8 窄字符串，人们必须使用 `UNICODE` 预定义常量来编译代码。微软教育 Windows 的 C++ 程序员必须用“宽字符”来处理 Unicode（更有甚者，用 TCHAR，来让程序员通过改变编译器设定来选择是否干脆完全不支持 Unicode）。这样的混乱让许多 Windows 程序员困惑，到底应该怎样才能正确处理文本。

与此同时，在 Linux 和 Web 世界，大家默许了 Unicode 的最佳编码是 UTF-8。即使它的主要优点在于比起其他文本更能简短地表示英语和计算机语言（如 C++、HTML、XML 等），它在各种常用字符集上的表现也都几乎不逊色于 UTF-16。

3. 常识

- 在 UTF-8 和 UTF-16 编码里，一个码位最多都要占 4 个字节。
- UTF-8 与端序无关。UTF-16 有两种端序：UTF-16LE 和 UTF-16BE（分别是小端序和大端序）。在这里我们统称为 UTF-16。
- 宽字符在某些平台上占 2 个字节，另一些平台上占 4 个字节。
- 对 UTF-8 和 UTF-32 字符串排序可以得到和字典序相同的结果，UTF-16 则不能。（译者注：Unicode 码位的汉字排序是康熙字典序，而不是新华字典序。Unihan 数据库可用于实现拼音排序）
- UTF-8 可以高效表示英文字母和其他 ASCII 字符（每字符一个字节），而 UTF-16 只对一部分亚洲字符集有利（只需 2 字节，UTF-8 需要 3 字节）。这也是为什么 Web 世界最喜好 UTF-8 的原因。英语书写的 HTML/XML 标签掺杂了任何语言的正文文本。西里尔语、希伯来语等其他几个常用 Unicode 区块在 UTF-16 和 UTF-8 都占 2 个字节。
- UTF-16 经常被误以为是定长编码，即使是 Windows 自带的程序也如此：Windows（Vista 之前）的纯文本编辑控件中，占 4 个字节的 UTF-16 字符需要按两下退格键才能删掉。在 Windows 7 中，控制台把此类字符显示成两个乱码字，无论选择什么字体。
- 许多适用于 Windows 的第三方库不支持 Unicode：它们接受窄字符串参数，然后传递给 ANSI API。有时甚至文件名也是如此。这种问题通常无解，因为一个字符串可能无法在任何 ANSI 代码页完整表示（如果它包含 Unicode 各个区块的大杂烩）。Windows 程序员一般会去查询这个文件（如果已经存在）的 8.3 DOS 短文件名来传递给这样的库。如果要创建一个不存在的文件，或者 8.3 路径长度超过了 `MAX_PATH`，又或者操作系统设定里禁用了短文件名的生成，那就无法实现了。
- 在 C++ 中，`std::exception::what()` 只能用 UTF-8 编码来返回 Unicode。`localeconv` 只有使用 UTF-8 才能支持 Unicode。
- UTF-16 在今天仍然流行，即使是在 Windows 世界以外。Qt、Java、C#、Python（CPython 参考实现 3.3 版本之前，见下文）以及 ICU，内部都使用 UTF-16 来表示字符串。

4. 不透明数据参数

我们再谈文件复制程序。在 UNIX 世界里，几乎所有窄字符串都默认是 UTF-8。正因如此，文件复制程序的作者并不需要在意 Unicode。因为参数是作为不透明数据，即 cookie，所

以只要用 ASCII 字符串作为文件名参数测试成功，程序就自动支持了任何语言的文件名。文件复制程序的代码不需要做**任何**改动就能支持外国语言。`fopen()` 函数完美支持 Unicode，`argv` 亦然。

现在让我们看看 Microsoft Windows，基于 UTF-16 架构，是怎么做的。使文件拷贝程序支持多个 Unicode 区块的大杂烩（多语种混合）需要一些奇技淫巧。首先，应用程序必须在编译时打开 Unicode 支持。这样的话，程序不能有 C 语言标准形参的 `main()` 函数。它得能接受 UTF-16 编码的 `argv`。要把一个本为窄字符串写的 Windows 程序改造成支持 Unicode，开发者需要深度重构，照顾好每一个字符串变量。

MSVC 自带的标准库对 Unicode 支持得很糟糕。它把窄字符串参数直接传递到操作系统的 ANSI API，没办法重载。改变 `std::locale` 也没用。没法用 C++ 的标准特性来打开 Unicode 文件名的文件。打开文件的标准方法是：

```
std::fstream fout("abc.txt");
```

解决这个问题的正确方法是用微软自己发明的补救方法来接受宽字符串参数，而这是一个非标准扩展。

在 Windows 中，`HKLM\SYSTEM\CurrentControlSet\Control\Nls\CodePage\ACP` 注册表键可以允许 ANSI API 支持非 ASCII 字符，但是只能选择一个固定的 ANSI 代码页。或许 65001 值可以解决 Windows 的 cookie 问题，然而微软没实现它。如果微软实现了这个 ACP 值，就能够在 Windows 平台上促进 UTF-8 的广泛接受。

对于 Windows 程序员和跨平台库厂商，我们将在“如何在 Windows 上处理文本”一节中讨论我们提出的处理文本字符串和重构程序、增强 Unicode 支持的办法。

5. 字形、字素群和其他 Unicode 概念

这里是根据 Unicode 标准，对字符、码位、编码单元和字素群之定义的摘录。建议参考 Unicode 标准相关章节的详细说明。

码位 (Code point)

Unicode 编码空间的任何数值。[§ 3.4, D10] 例如：U+3243F。

Привет नमस्ते שָׁלוֹם

你看到了几个**字符**?

编码单元 (Code unit)

可以表示一段已编码文本的最小比特组合。[§ 3.9, D77] 例如，UTF-8、UTF-16 和 UTF-32 分别使用 8 比特、16 比特和 32 比特编码单元。上述码位可以编码成四个 UTF-8 编码单元“`f0 b2 90 bf`”，两个 UTF-16 编码单元“`d889 dc3f`”，一个 UTF-32 编码单元“`0003243f`”。请注意，这仅仅是一**组比特**序列；它们存储在面向字节的媒体

上的方式取决于特定编码的端序。当存储上述 UTF-16 码位时，UTF-16BE 会转换为“`d8 89 dc 3f`”，UTF-16LE 会转换为“`89 d8 3f dc`”。

抽象字符 (Abstract character)

一个用于组织、控制或表示文字数据的信息单元。[§3.4, D7] 该标准在 3.1 节进一步说道：

> 对于 Unicode 标准，[…] 收录本身是开放的。因为 Unicode 是一种通用编码，任何抽象字符只要能被编码，就可能成为编码的候选，无论这个字符是否已知。

这个定义的确很抽象。只要是人能想到的任何字符，就**是**一个抽象字符。例如，  **精灵语 tengwar 字母 ungwe** 是一个抽象的字符，虽然它尚不能用 Unicode 表示。

已编码字符 (Encoded character, Coded character)

码位和抽象字符之间的映射。[§3.4, D11] 例如，U+1F428 是一个代表抽象字符  *考拉*的已编码字符。

这种映射是既不是双射，也不是单射，更不是满射：

- 代理字符（surrogate），非字符（noncharacter）和未分配的码位完全不对应抽象字符。
- 一些抽象字符可以由不同的码位进行编码；U+03A9 *希腊大写字母 omega* 和 U+2126 *欧姆符号* 都对应于同一个抽象字符“Ω”，**必须同等对待**。
- 一些抽象字符无法用单个码位进行编码，要用已编码字符的**序列**才能表示它们。例如，表示抽象字符 ю́ **带锐音符的西里尔小写字母 yu** 的唯一方法是使用 U+044E *西里尔小写字母 yu* 和 U+0301 *组合锐音符* 这个序列。

  此外，一些抽象字符不仅有单码位表示方法，**还有**多码位表示方法。抽象字符 ǵ 可以用单一码位 U+01F5 *带锐音符的拉丁小写字母 g* 编码，或者用 <U+0067 *拉丁小写字母 g*, U+0301 *组合锐音符*> 这个序列。

用户观感字符 (User-perceived character)

任何最终用户所认为的字符。这个概念与语言有关。例如，“ch”在英语和拉丁文中是两个字母，而在捷克语和斯洛伐克语中则是一个字母。

字素群 (Grapheme cluster)

“应该放在一起”的已编码的字符序列。[§2.11] 字素群可以近似地看作语言中立的用户观感字符。例如它们可适用于光标移动和选择。

字模 (Glyph)

一种字体中的单个特定形状。字体是字模的集合，由字体设计师设计。文本成型和渲染引擎负责将码位序列转换成为特定字体中字模的序列。这个转换的规则可能很复杂且取决于语言环境设定。这不在 Unicode 标准范围之内。

“字符”可以代表以上任何概念。Unicode 标准用“字符”作为“**已编码字符**”的同义词。[§ 3.4] 编程语言或程序库文档中的“字符”通常是指一个编码单元。如果问最终用户一个字符串中字符 的数量，他会数“用户观感字符”。根据一个程序员对 Unicode 了解程度的不同，他可能认为“字符”是编码单元、码位或者字素群。例如这是 Twitter 计算字符的方式。在我们看来，一个字符串长度函数对字符串“🐭”不一定要返回 1 才算 Unicode 兼容。

6. 亚洲语言文本：UTF-8 与 UTF-16

因此，在 UTF-8 和 UTF-16 中大多数 Unicode 码位都占相同的字节数。包括俄语、希伯来语、希腊语，和所有非 BMP 码位在两种编码中都占了 2 或 4 字节。UTF-16 编码的拉丁字母、一些标点符号和 ASCII 码中的其他字符占了更多空间，而 UTF-8 编码的一部分亚洲字符则占用了更多空间。亚洲程序员难道不能转储 UTF-16，从而使每个字理论上节省 50% 的内存吗?

事实不是这样。仅有人工构造的，且只含有 U+0800 到 U+FFFF 范围内的文本示例，才能节省一半内存。然而计算机之间文本接口的使用远超其他文本数据的用途。这包括 XML、HTTP、文件路径和配置文件——它们几乎都仅使用了 ASCII 字符。而且事实上，UTF-8 在各个亚洲国家广为使用。

在专门储存中文书籍的场合，UTF-16 可能还是一种合理的优化。一旦从这样的数据库取出文本，就应该将其转为和世界接轨的标准编码。无论采用哪种方案，在存储成本较高时，人们往往会使用无损压缩。这样的话，UTF-8 和 UTF-16 占用的储存空间就差不多了。此外，“在上述语言中，一个字符比一个拉丁字符含有的信息量更大，因此其占用更多空间是合情合理的。”（Tronic，UTF-16 是有害的）

这是一个简单实验的结果。第一列是某网页（《日本》，2012 年 1 月 1 日取自日语维基百科）的 HTML 源代码占用的空间。第二列是相应删除标记的纯文本，即“全选、复制、粘贴到纯文本文件”的结果。

	HTML 源代码（与 UTF-8 比较）	**纯文本（与 UTF-8 比较）**
UTF-8	767 KB (0%)	222 KB (0%)
UTF-16	1 186 KB (+55%)	176 KB (−21%)
UTF-8，ZIP 压缩	179 KB (−77%)	83 KB (−63%)
UTF-16LE，ZIP 压缩	192 KB (−75%)	76 KB (−66%)

UTF-16BE，ZIP 压缩	194 KB (−75%)	77 KB (−65%)

可以看出，UTF-16 对于实际数据比 UTF-8 多占用约 50% 的空间，对亚洲语言纯文本也仅仅节省了 20%，并很难超越通用压缩算法。

7. 已编码字符串中的文本操作

流行的基于文本的数据格式（例如 CSV、XML、HTML、JSON、RTF 和计算机程序源代码）常常包含 ASCII 字符作为结构控制元素，并且可以包含 ASCII 和非 ASCII 的文本数据字符串。用可变长编码来操作可能看起来很困难，因为其中继承自 ASCII 的码位比其他码位短，而字符串中已编码字符间的界限不是一望即知。这推动了软件架构选择 UCS-4 定长编码（如 Python v3.3）。其实，这既不必要也不能解决我们知道的任何实际问题。

UTF-8 编码在设计上保证了一个 ASCII 字符或子字符串永远不会匹配到一个多字节编码的字符中间。这在 UTF-16 中也适用。这两个编码中，多字节编码的码位的编码单元会将 MSB 设为 1。

比如要找 HTML 标签的开始记号“<”，或在 UTF-8 编码的 SQL 语句中找到单引号（'）来预防 SQL 注入，这和在全英文 ASCII 文本字符串中做法一样。UTF-8 的编码方式保证了这行得通。具体来说，每一个非 ASCII 字符在 UTF-8 中会编码为一系列字节，每个字节的值都大于 127。一个朴素的算法完全没有机会与之冲突——简单、快速、优雅，没有必要关心编码的字符边界。

此外，你还可以像在简单的字节数组中一样，直接在一个 UTF-8 编码的字符串中搜索 UTF-8 编码的非 ASCII 的子字符串——无需关注码位边界。这要归功于 UTF-8 的另一个设计特点——一个码位编码的起始字节永远不会与其他码位的尾随字节相同。

8. 有关字符计数的更多谣言

前面我们说过，有一种流行的观点认为计数、拆分、索引或以其他方式遍历 Unicode 字符串中的码位应该算是一种频繁和重要的操作。在本节中，我们将更详细地探讨这点。

8.1. 1. 可以在常数时间内对 UTF-16 中的字符计数。

这是那些认为 UTF-16 是定长编码的人的一个常见错误。这是错的。实际上 UTF-16 是可变长度编码。如果你否认非 BMP 字符的存在，请参阅这则常见问题。

8.2. 2. 可以在常数时间内对 UTF-32 中的字符计数。

这和误用“字符”的定义有关。我们确实可以在常数时间内对 UTF-32 中的编码单元和码位计数。然而，码位并不对应于用户观感字符。即使是在 Unicode 形式化的规定中，有些码位

对应于“**已编码的字符**”，有些对应于“**非字符**”。

8.3. 3. 对已编码的字符或码位计数很重要。

我们认为，人们经常夸大了码位的重要性，这通常是由于误解了 Unicode 的复杂性。Unicode 也只是反映了人类语言的复杂性。我们很容易就能数出“Abracadabra”里有几个字符，但再来看这个字符串：

Приве́т नमस्ते שָׁלוֹם

这里包含了 22 (!) 个码位，却只有 16 个字素群。如果将其转换为 NFC 形式，可以减少到 20 个码位。即使这样，其中的码位数量与几乎任何软件工程任务无关。也许把字符串转换为 UTF-32 是唯一一个例外。例如：

- 光标移动和文本选择等，应使用“**字素群**”。
- 在限制输入字段、文件格式、协议或数据库中字符串的长度时，长度单位是某些已知编码的**编码单元**。这里的原因是，无论是在内存、磁盘还是某一特定的数据结构中，任何长度限制都来自于底层给字符串分配的固定大小的储存空间。
- 屏幕上字符串的长度与字符串的码位数量无关，需要从渲染引擎那里取得。即使是等宽字体和终端，一个码位也不一定占一列。POSIX 考虑到了这一点。

8.4. 4. 在 NFC 中每个码位对应一个用户观感字符。

错，因为 Unicode 可以表示几乎无限多的用户观感字符。即使是在实际使用中，大多数字符也没有一个已完全组合的形式。例如，上述例子的 NFD 字符串包含了三种**真实**语言写的三个**真实**词汇，其 NFC 形式也用了 20 个码位。这数量还是远远超过 16 个用户观感字符。

8.5. 5. 取字符串长度的 `length()` 操作必须计算用户观感或已编码字符数量，否则它就是没有正确支持 Unicode。

人们通常用“取字符串长度”操作返回的数值来评判程序库和编程语言对 Unicode 的支持程度。按照这个评判标准，大多数流行的编程语言，如 C#、Java，甚至 ICU 本身都不支持 Unicode。例如，对于这个单字符的字符串“🤦🏼‍♂️”，用 UTF-16 作为内部编码的话，其长度通常是 2；而使用 UTF-8 的语言则会认为其长度是 4。误解的根源在于，这些语言的规范用“字符”来表示一个编码单元，程序员则误以为它是别的什么。

那就是说，这些 API 返回的编码单元数量有最大的实际意义。把一个 UTF-8 字符串写入文件时，重要的就是字节数量。另一方面，计算任何其他意义上的“字符”数量就不是很有用。

9. 我们的结论

UTF-16 是可变长度编码，又太占空间，在各种意义上都是最糟糕的。它只是因历史原因而存在，引发了无数混乱。我们希望能进一步减少它的使用。

相比于对原生平台 API 的互操作性，可移植性、跨平台兼容性和简朴性更加重要。所以，最好的办法是随处使用 UTF-8 窄字符串，在用不支持 UTF-8 而接受宽字符串的平台 API（如 Windows API）时来回转换。与字符串相关的系统 API 打交道时（如 UI 代码和文件系统 API），性能几乎不是问题。在应用程序里统一编码是莫大的优势，我们找不到充足的理由来反驳。

说到性能，计算机往往使用字符串通信（例如 HTTP 报头、XML、SOAP）。很多人认为不应该用文本通信协议，但实际上通信协议几乎都用英语和 ASCII 字符组成，UTF-8 更有优势。对不同类的字符串用不统一的编码使复杂度大大上升，更容易引发问题。

我们尤其认为给 C++ 标准增加 `wchar_t` 是一个错误，给 C++11 增加 Unicode 亦然。我们更想要一个能存储任意 Unicode 数据的**基本执行字符集**。然后，让每一个 `std::string` 或 `char*` 参数都兼容 Unicode。“只要这里接受文本，就能兼容 Unicode”——有了 UTF-8，这就是小菜一碟。

标准库的 facet 有一堆设计缺陷。比如 `std::numpunct`、`std::moneypunct` 和 `std::ctype` 不支持变长编码字符（UTF-8 的非 ASCII 部分和 UTF-16 的非 BMP 部分），以及没有为转换提供足够的信息。这些问题亟待修复：

- `decimal_point()` 和 `thousands_sep()` 应返回字符串而不是单个编码单元。C 语言的 `localeconv` 函数就是这么做的，尽管这函数无法自定义。
- `toupper()` 和 `tolower()` 不应该按一个编码单元为一项，因为这样不支持 Unicode。比如拉丁文 ffl 连字必须转换成 FFL，德文 ß 要转换成 SS（有个大写形式 ẞ，但传统大小写规则不用它）。此外，某些语言（如希腊语）的小写字母有词尾变形，所以转换例程得注意到字母的位置才能正确转换大小写。

10. 如何在 Windows 上处理文本

本节主要谈跨平台库开发和 Windows 编程。Windows 平台的问题是，它的窄字符串系统 API（至今仍）不支持 Unicode。给 Windows API 传递 Unicode 字符串的唯一方法是转换为 UTF-16（又名宽字符串）。

需要注意的是，我们的准则与微软的原版指南截然相反。我们的方法是在临近 API 调用时才转换字符串，而不长期持有宽字符串数据。前面的章节已经说明，这样做可以提升性能和稳定性，保持代码简洁，更有利于与其他软件交互。

- 不要在临近接受 UTF-16 参数的 API 之外的地方用 `wchar_t` 或 `std::wstring`。
- 不要在接受 UTF-16 参数的 API 之外的地方用 `_T("")` 或 `L""` 字面量。
- 不要使用对 `UNICODE` 常量敏感的类型、函数等，如 `LPTSTR`、`CreateWindow()` 和 `_T()` 宏，而要用 `LPWSTR`、`CreateWindowW()` 和显式 `L""` 字面量。
- 然而，要始终定义上 `UNICODE` 和 `_UNICODE`，来避免不小心把 UTF-8 窄字符串传递给 ANSI WinAPI 并且通过编译。可以修改 VS 项目设定，名字叫做“**使用 Unicode 字符集**”。
- 在程序的任何地方，都认为 `std::string` 和 `char*` 变量是 UTF-8。
- 如果你能写 C++，下述的 `narrow()` / `widen()` 转换函数对于内联转换语法非常实用。当然你也可以用别的 UTF-8/UTF-16 转换代码。
- 只用接受宽字符（`LPWSTR`）的 Win32 函数。严禁使用接受 `LPTSTR` 和 `LPSTR` 的函数。这样传递参数：

```
::SetWindowTextW(widen(标准库字符串变量 或 "字符串字面量").c_str())
```

  这种方法使用下述的转换函数。另请参阅，有关转换性能的说明。
- 对于 MFC 字符串：

```
CString someoneElse; // MFC 引入的某字符串。

// 在传递给别的 API 调用前尽早转换掉：
std::string s = str(boost::format("你好 %s\n") % narrow(someoneElse));
AfxMessageBox(widen(s).c_str(), L"错误", MB_OK);
```

- 对于 .NET 开发人员：使用基于 UTF-16 的原生字符串可能难以避免。谨记字符串类的接口严重暴露了其实现细节。例如，`string[index]` 操作可能返回某字符的一部分（UTF-8 字节数组也如此）。当序列化字符串到输出文件或通信设备时，记得要指定 `Encoding.UTF8`。准备好接受转换（比如典型的 ASP.NET Web 应用程序输出 UTF-8 编码的 HTML 时）引起的性能损失。

10.1. 在 Windows 上应付文件、文件名和 fstream

- 始终以 UTF-8 输出文本文件。
- 为了 RAII/OOD，避免使用 `fopen()`。如果非要用，就用 `_wfopen()`，并且遵循前述的 WinAPI 调用约定。
- 不要把 `std::string` 或 `const char*` 文件名传递给 `fstream` 家族的参数。MSVC CRT 不支持 UTF-8 参数，但是有个非标准扩展，这样用：

- 用 `widen` 来把 `std::string` 转换成 `std::wstring`：

```
std::ifstream ifs(widen("你好"), std::ios_base::binary);
```

  如果 MSVC 对 `fstream` 的态度有所悔改，我们终将手工去掉这样的转换。
- 这段代码不跨平台，将来肯定要手动修改。
- 或者可以把转换封装起来。

10.2. 转换函数

本指南使用 Boost.Nowide 库中的转换函数（它还没进入 boost 官方库）：

```
std::string narrow(const wchar_t *s);
std::wstring widen(const char *s);
std::string narrow(const std::wstring &s);
std::wstring widen(const std::string &s);
```

本库还封装了常用 C 和 C++ 库函数，提供了应付文件操作和用 iostream 读写 UTF-8 的方法。

用 Windows 的 `MultiByteToWideChar` 和 `WideCharToMultiByte` 函数也很容易实现上述函数和封装。你还可以用别的（也许更快的）转换例程。

11. 常见问题

1. ### 11.1. Q: 你是不是 Linux 人？你是不是想趁机反对 Windows?

   A: 不，Windows 伴我成长。我主要还是在 Windows 上做开发。我认为微软在字符串领域误入歧途，因为他们比别的厂商更早做了决定。

2. ### 11.2. Q: 你是不是亲英派？你是不是暗自认为英语字母表和英国文化是世界上最好的?

   A: 不，我家乡的语言不在 ASCII 码表中。我认为用单字节能编码 ASCII 字符的格式不一定是亲英派，甚至与人类交互无关。你可以争辩说本来就不应该存在程序源代码、网页、XML 文件、操作系统文件名等机对机文本接口。但既然存在，那么说明文本不一定是给人类看的。

3. **11.3. Q: 你们这帮人到底在想什么？我用 C# 和/或 Java 编程，完全不用考虑什么编码啊。**

   A: 并非如此。C# 和 Java 提供的 `char` 类型都是 16 比特的。恭喜你，它不一定能完整表示一个 Unicode 字符。.NET 索引操作 `str[i]` 也是操作在内部表示单元上，所以这又是一个不完全的抽象。操作子字符串的方法会兴高采烈地把非 BMP 字符一刀两断，返回一个无效的字符串。

   此外，你还要注意在往磁盘上的文件、网络通信、外部设备或其他程序要读取的地方写文本时所使用的编码。无论内容是什么都请使用 `System.Text.Encoding.UTF8` (.NET)，而不要用 `Encoding.ASCII`、UTF-16 或手机 PDU。

   底层框架选错了内部字符串表示的方法，影响了 ASP.NET 这样的 Web 框架：网络应用几乎全要用 UTF-8 来输出（和输入）字符，在大流量的网络应用和服务中，字符串转换导致了显著开销。

4. **11.4. Q: UTF-8 不就是一个兼容 ASCII 的尝试吗？为什么还要用这老古董?**

   A: 不论原来 UTF-8 是否在创造时是作为一个兼容性措施，现在它比任何其它 Unicode 编码更好，也更流行。

5. **11.5. Q: 占用超过两字节的 UTF-16 字符在实际生活中十分罕见，所以 UTF-16 就可以看作是一个定长编码，有一堆好处。我们就不能无视这些字符吗?**

   A: 你不打算让你的软件设计完整支持 Unicode，是在开玩笑吗？那么，既然你打算支持，而非 BMP 字符罕见就不支持，除了给软件测试增加难度，实在没什么好处。然而，真正值得在意的是，真实的应用程序不怎么操作字符串——只是原封不动地传递字符串。这意味着“几乎定长”几乎没有性能优势（参见“性能”），让字符串短一点倒有可能挺重要。

6. **11.6. Q: 只要程序员知道如何使用，为什么不让他们在程序内部选择他们最爱的编码?**

   A: 我们不反对任何编码的正确使用。但是如果同一个类型，比如 `std::string`，根据上下文语境有不同的含义，就引发问题了。有些人会认为它意味着“ANSI 代码页”，另一些人会认为“这代码有问题，不支持非英语文本”。在我们的程序里，它就意味着支持 Unicode 的 UTF-8 字符串。这样的分歧是许多 bug 和苦恼的源泉。我们的世界实在不需要让事情变得更复杂了。整个行业因此出现了一堆有 Unicode 问题的软件。JoelOnSoftware 认为，要想消灭这些有问题的软件，就得让每一个程序员都意识到编码的问题。我们坚信，只要让软件 API 默认使用统一的主流编码，程序员不需要成为一个语言文字专家就能够正确地写一个文件复制程序了。

7. **11.7. Q: 我的应用程序只有图形界面，又不做 IP 通信，也不做文件读写。为什么我调用 Windows API 的时候非要来来回回转换字符串，而不是干脆只用宽字符串变量呢?**

   这是个合理的捷径。确实，这种情况下用宽字符串没有问题。但是如果你将来打算加个配置文件或日志文件，请考虑把字符串都转换成窄字符串。这样可以免除后患。

8. **11.8. Q: 既然你不打算用 Windows 的 `LPTSTR` / `TCHAR` 等宏，为什么还要打开 `UNICODE` 定义呢?**

   A: 这是一个额外的安全措施，防止将 UTF-8 编码的 `char*` 字符串传入那些接受 ANSI 字符串的 Windows API 函数。我们想产生一个编译错误。与这个情况类似，在 Windows 上将一个 `argv[]` 字符串传入 `fopen()` 也是一个难以发现的 bug。Windows 假定用户不会传入非当前码页的文件名。手工测试通常不能发现这种问题，除非你的测试员能熟练地偶尔传入一些中文文件名。然而这还是个错误的程序逻辑。有了 `UNICODE` 定义，这种情况就会出编译错误。

9. **11.9. Q: 认为微软终将停止使用宽字符串是不是太幼稚了?**

   A: 让我们首先看到他们支持 `CP_UTF8` 作为有效代码页的那一天。这不会很难。之后，我们认为 Windows 开发者就找不到理由继续使用宽字符 API。而且，加入对 `CP_UTF8` 的支持就能直接“修复”一些已有不支持 Unicode 的程序和库。

   一些人说加入 `CP_UTF8` 支持会**破坏**现有使用 ANSI API 的程序，据说是因为这样，微软只好创建一个宽字符串 API。这是错的。甚至一些流行的 ANSI 编码也是变长的（例如 Shift JIS），所以正确的代码都能搞定变长编码。微软选择 UCS-2 单纯是历史原因——在 UTF-8 发明前，人们认为 Unicode “只是一个宽的 ASCII”，还认为使用定长编码很重要。

10. **11.10. Q: 你对于 BOM （端序标记）是怎么看的?**

    A: 根据 Unicode 标准 (v6.2, p.30): “UTF-8 既不要求也不推荐使用 BOM”。

    端序问题又是一个避免 UTF-16 的理由。UTF-8 没有端序问题，UTF-8 BOM 的存在只是用来声明这是一个 UTF-8 流。如果只有 UTF-8 是唯一流行的编码（在互联网世界已经是这样），BOM 就是多余的。实际上，现在大多数 UTF-8 的文本文件都省略了 BOM。

    我们无法接受，即使是在像文件连接这样简单的情形下，所有现有的代码都得注意到 BOM 的问题。

11. **11.11. Q: 你如何评价行尾标记?**

    A: 永远使用 `\n` (`0x0a`) 作为行尾标记，即使是在 Windows 上。文件应以二进制模式读

写，这保证了互操作性——一个程序在任何系统上都会给出相同结果。既然 C 和 C++ 标准采用了 `\n` 作为内部行尾表示，这就导致了所有文件会以 POSIX 惯例输出。文件在 Windows 上用“记事本”打开可能会出问题；然而任何像样的文本编辑器都能理解这样的行尾。

我们也偏好 SI 单位、ISO-8601 日期格式，用句点而不是逗号作为小数点。

12. ### 11.12. Q: 那文本处理算法、字节对齐之类的性能问题呢?

A: 使用 UTF-16 真的更好吗？可能是的。ICU 用 UTF-16 是历史因素，这样的话就难以比较。然而，大部分情况下字符串是作为 cookie 对待，不是动不动就排序或反转。一个更稠密的编码就更有利于性能。

13. ### 11.13. Q: 人们误用 UTF-16 并误以为它每个字符就是 16 比特，是不是失误?

A: 不一定。不过确实，各种设计的一个重要特性就是安全性，编码也不例外。

14. ### 11.14. Q: 如果 `std::string` 表示 UTF-8，那用 `std::string` 存储纯文本的时候不会弄混吗?

A: 没有纯文本这种概念。没理由在一个名叫“string”的类里存储仅 ASCII 或 ANSI 代码页编码的文本。

15. ### 11.15. Q: 在 Windows 中传递字符串时，UTF-8 和 UTF-16 之间的转换不会拖慢我的程序吗?

A: 首先，无论用哪种做法，你肯定会做一些转换，要么是系统调用，要么是与世界交互。例如，用 TCP 发送一段文本字符串。而且，那些接受字符串的系统 API 通常是执行原本就慢的任务，比如用户界面或文件系统操作。如果你的程序中全是与系统 API 的交互，我们来看个小实验。

操作系统 API 的一个典型用法就是打开文件。执行以下函数在我的机器上用了 (184 ± 3)µs:

```
void f(const wchar_t* name)
{
    HANDLE f = CreateFile(name, GENERIC_WRITE, FILE_SHARE_READ, 0, CREATE_ALWAYS, 0, 0);
    DWORD written;
    WriteFile(f, "Hello world!\n", 13, &written, 0);
    CloseHandle(f);
}
```

而这个用了 (186 ± 0.7)µs:

```
void f(const char* name)
{
    HANDLE f = CreateFile(widen(name).c_str(), GENERIC_WRITE, FILE_SHARE_READ, 0, CREATE_ALW
    DWORD written;
    WriteFile(f, "Hello world!\n", 13, &written, 0);
    CloseHandle(f);
}
```

（两次都用了 `name="D:\\a\\test\\subdir\\subsubdir\\this is the sub dir\\a.txt"`，取五次平均。我们用了一个优化过的 `widen`，其利用了 C++11 保证 `std::string` 连续储存的特性。）

只有 (1 ± 2)% 的开销。况且，`MultiByteToWideChar` 不是最快的 UTF-8↔UTF-16 转换函数。

16. 11.16. Q: 在 C++ 源码中如何书写 UTF-8 字符串字面量?

A: 如果你对你的软件进行了国际化，所有非 ASCII 字符串会从额外的翻译数据库载入，那就不存在问题。

如果你还想嵌入特殊字符，你可以按如下方法做。在 C++11 中你可以:

```
u8"∃y ∀x ¬(x < y)"
```

对不支持“u8”的编译器你可以像这样硬编码 UTF-8 单元:

```
"\xE2\x88\x83y \xE2\x88\x80x \xC2\xAC(x \xE2\x89\xBA y)"
```

然而最直接的方式就是原样书写字符串，并将源文件保存为 UTF-8 编码:

```
"∃y ∀x ¬(x < y)"
```

不幸的是，MSVC 会将其转为某些 ANSI 代码页，从而损坏字符串。将文件存为**无 BOM**的 UTF-8 可以解决。MSVC 会认为文件已经在正确的代码页，就不会碰你的字符串。（译者注：在 DBCS 下无效，试试在简体中文下输出 `"井\n"`。）然而，这样就无法使用 Unicode 标识符和宽字符串字面量了（反正你也不会用）。

17. 11.17. Q: 我有个基于 char 的大型 Windows 应用程序。使其支持 Unicode 最简单的方法是什么?

保留这些 char。定义 `UNICODE` 和 `_UNICODE`，这样忘了使用 `narrow()` / `widen()` 的情况就会有编译错误。（在 Visual Studio 中勾上“**使用 Unicode 字符集**”选项）找出所有用到 `fstream` 和 `fopen()` 的地方，改成上述的宽字符重载。这样就基本完成了。

如果你用到了不支持 Unicode 的第三方库，例如它会原样将文件名字符串传递给 `fopen()`，你得用上述像 `GetShortPathName()` 这样的工具来替代解决。

18. **11.18. Q: Python 呢？我听说他们为更好支持 Unicode，在 v3.3 改进了不少。**

A: 大概是吧，不过他们不必大费周折也能支持得更好。在 CPython v3.3 参考实现中，他们改了内部字符串表示方法。原来的 UTF-16，根据字符串实际内容，换成了三种可能的编码（ISO-8859-1、UCS-2 或 UCS-4）。在字符串中加入一个非 ASCII 或非 BMP 字符，常常会使整个字符串被隐式转换为别的编码。脚本并不知道字符串内部的实际编码。这样的设计本来是为了优化对 Unicode 码位的**索引操作**。然而对大部分用途来说，我们认为对码位——而不是字素群——的计数或索引并不重要。据我们所知，Python 不支持字素群的计数或索引。

因此，我们反对黑盒化处理字符串，推荐内部采用 UTF-8 来表示字符串，且在 API 中直接暴露 UTF-8 的表示。Python 在此次改进之前，字符串索引操作计数的是编码单元而不是码位。这样本可以简化实现且提升性能，比如脚本用于处理 Web 时。Web 上的文本大多已经是 UTF-8 编码，致使 Python 编程语言在服务端更为适用。你可能会认为这是阻止脚本程序员错切字符串的安全措施，但错把字素群切断不也很危险么？ 即使 Python 现在已经完全支持 Unicode 了，但是我们认为这个少有历史包袱的现代化工具应该在文本处理方面做得更好。

CPython 之外，JPython 和 IronPython 依赖于托管平台（分别是 Java 和 .NET）不幸选择的字符串编码。所以必须特别关照代理对问题。

19. **11.19. Q: 但是为什么用 `std::string`？做一个支持 UTF-8 的字符串类难道不是面向对象的更好方案吗?**

A: 并非每一段字符串相关的代码都涉及到字符串的处理和文本的验证。文件复制程序只需要用一个简单的字节缓冲区即可接受 Unicode 文件名并传递给文件 IO 例程。如果你打算设计一个接受字符串的程序库，只要用简单、标准且轻量的 `std::string` 就好。另一方面，重新发明一个新字符串类，然后逼着每个人都用你的特制接口是不对的。当然，如果某人不仅仅需要来回传递字符串，就应该用合适的文本处理工具。然而，秉承 STL 的容器/算法分离的精神，这样的工具最好和具体用哪个类无关。事实上，还有些人认为 `std::string` 的接口过于臃肿，大部分功能应该从 `std::string` 类里移出来。

20. **11.20. Q: 我已经在用这种方法了。我希望我们的展望成真。我应该怎么办呢?**

A: 去传播福音吧。

重审你的代码，看看哪个库在跨平台支持 Unicode 的代码中使用让你最为头痛。给那个库的作者开一张 bug 报告单。如果你是 C 或 C++ 程序库作者，用 `char*` 和 `std::string`，默

认 UTF-8 编码，并且拒绝支持 ANSI 代码页——因为这东西本来就不支持 Unicode。

如果你是微软员工，请推进支持并实现 `CP_UTF8` 作为窄 API 代码页。

进一步的想法：

- 为常见的第三方库（如 PugiXML、LibTIFF 等）做 UTF-8 支持，并创建一个仓库来放此类补丁。这些库是用标准 C 语言写的，并不在乎 Windows。
- 为 Windows 创建一个标准库函数（如 `fopen()`）、`main()` 和全局环境变量的链接时补丁，用来做参数转换。

12. 关于作者

本宣言的作者是 Pavel Radzivilovsky、Yakov Galka 和 Slava Novgorodov。这是我们基于经验和现实中程序员对于 Unicode 犯下真正的错误，遇到真正的困难的结晶。我们的目的是呼吁加深对文本问题的认识，为业界降低面向 Unicode 编程的难度抛砖引玉，最终提升工程师做出的程序的用户体验。我们与 Unicode 联盟无关。

特别感谢 Glenn Linderman 提供关于 Python 的信息，Markus Künne、Jelle Geerts、Lazy Rui 和 Jan Rüegg 反馈这篇文档的缺陷和拼写错误。

本文大部分灵感来源于 StackOverflow 上 Artyom Beilis 引发的讨论，他是 Boost.Locale 的作者。还有些灵感来源于 VisionMap 的开发约定以及 Michael Hartl 的 tauday.org。

13. 译者注

让应用程序支持 Unicode 是不可阻挡的潮流。然而亚洲的许多软件开发者并没有意识到 Unicode 和 UTF-8 的重要性，制造出很多在非本土语言设定下会崩溃的应用程序。身为亚洲人，我们更应当让自己的语言在互联网时代得到传承。然而对 Unicode 的无知甚至在祸害下一代：小学课本一边炫耀着“我们有 7 万多个汉字”，一边教育着好奇的孩子们“一个汉字占 2 字节”。

我们希望让读者知道，现在是时候抛弃陈旧的错误观念了。支持 Unicode 方可大幅提升用户体验。我们建议选择 UTF-8 而不是 UTF-16、GBK 或 GB18030 作为应用程序的默认编码。

感谢以下译者：

- Star Brilliant
- Dingyuan Wang
- Howard Xiao

- James Swineson
- Kexy Biscuit

如果您对本译文的遣词造句有任何意见或建议，请访问本翻译项目的 GitHub 仓库，通过提交 GitHub issue 的方式与我们取得联系。

14. 延伸阅读

- Unicode 联盟（Unicode 标准, PDF）
- Unicode 国际化组件 (ICU)
- Boost.Locale——C++ 的高质量本地化方案。
- Artyom Beilis 在 StackOverflow 上提问“UTF-16 有害吗？”
- Twitter 如何计数字符

15. 反馈

你可以在 Facebook UTF-8 Everywhere 主页上留下评论和反馈。我们非常感谢你的帮助和反馈。

比特币捐款：1UTF8gQmvChQ4MwUHT6XmydjUt9TsuDRn
捐款将用于研究和推广事业。

原文更新时间: 2016-07-19
译文更新时间: 2016-09-17