为啥需要 Unicode

我们知道计算机其实挺笨的,它只认识 0101 这样的字符串,当然了我们看这样的 01 串时肯定会比较头晕的,所以很多时候为了描述简单都用十进制,十六进制,八进制表示.实际上都是等价的,没啥太多不一样.其他啥文字图片之类的其他东东计算机不认识.那为了在计算机上表示这些信息就必须转换成一些数字.你肯定不能想怎么转换就怎么转,必须得有定些规则.于是刚开始的时候就有 ASCII 字符集(American Standard Code for Information Interchange，"美国信息交换标准码),它使用 7 bits 来表示一个字符,总共表示 128 个字符,我们一般都是用字节(byte,即 8 个 01 串)来作为基本单位.那么怎么当用一个字节来表示字符时第一个 bit 总是 0,剩下的七个字节就来表示实际内容.后来 IBM 公司在此基础上进行了扩展，用 8bit 来表示一个字符，总共可以表示 256 个字符.也就是当第一个 bit 是 0 时仍表示之前那些常用的字符.当为 1 时就表示其他补充的字符.

英文字母再加一些其他标点字符之类的也不会超过 256 个.一个字节表示主足够了.但其他一些文字不止这么多 ,像汉字就上万个.于是又出现了其他各种字符集.这样不同的字符集交换数据时就有问题了.可能你用某个数字表示字符 A,但另外的字符集又是用另外一个数字表示 A.这样交互起来就麻烦了.于是就出现了 Unicode 和 ISO 这样的组织来统一制定一个标准,任何一个字符只对应一个确定的数字.ISO 取的名字叫 UCS(Universal Character Set),Unicode 取的名字就叫 unicode 了.

总结起来为啥需要 Unicodey 就是为了适应全球化的发展,便于不同语言之间的兼容交互,而 ASCII 不再能胜任此任务了.

Unicode 详细介绍

1.容易产生后歧义的两字节

unicode 的第一个版本是用两个字节(16bit)来表示所有字符

.实际上这么说容易让人产生歧义,我们总觉得两个字节就代表保存在计算机中时是两个字节.于是任何字符如果用 unicode 表示的话保存下来都占两个字节.其实这种说法是错误的.

其实 Unicode 涉及到两个步骤,首先是定义一个规范,给所有的字符指定一个唯一对应的数字,这完全是数学问题,可以跟计算机没半毛钱关系.第二步才是怎么把字符对应的数字保存在计算机中,这才涉及到实际在计算机中占多少字节空间.

所以我们也可以这样理解,Unicode 是用 0 至 65535 之间的数字来表示所有字符.其中 0 至 127 这 128 个数字表示的字符仍然跟 ASCII 完全一样.65536 是 2 的 16 次方.这是第一步.第

二步就是怎么把 0 至 65535 这些数字转化成 01 串保存到计算机中.这肯定就有不同的保存方式了.于是出现了 UTF(unicode transformation format),有 UTF-8,UTF-16.

2.UTF-8 与 UTF-16 的区别

UTF-16 比较好理解,就是任何字符对应的数字都用两个字节来保存.我们通常对 Unicode 的误解就是把 Unicode 与 UTF-16 等同了.但是很显然如果都是英文字母这做有点浪费.明明用一个字节能表示一个字符为啥整两个啊.

于是又有个 UTF-8,这里的 8 非常容易误导人,8 不是指一个字节,难道一个字节表示一个字符?实际上不是.当用 UTF-8 时表示一个字符是可变的,有可能是用一个字节表示一个字符,也可能是两个,三个..反正是根据字符对应的数字大小来确定.

于是 UTF-8 和 UTF-16 的优劣很容易就看出来了.如果全部英文或英文与其他文字混合,但英文占绝大部分,用 UTF-8 就比 UTF-16 节省了很多空间.而如果全部是中文这样类似的字符或者混合字符中中文占绝大多数.UTF-16 就占优势了,可以节省很多空间.另外还有个容错问题,等会再讲

看的有点晕了吧,举个例子.假如中文字"汉"对应的 unicode 是 6C49(这是用十六进制表示,用十进制表示是 27721 为啥不用十进制表示呢?很明显用十六进制表示要短点.其实都是等价的没啥不一样.就跟你说 60 分钟和 1 小时一样.).你可能会问当用程序打开一个文件时我们怎么知道那是用的 UTF-8 还是 UTF-16 啊.自然会有点啥标志,在文件的开头几个字节就是标志.

EF BB BF 表示 UTF-8

FE FF 表示 UTF-16.

用 UTF-16 表示"汉"

假如用 UTF-16 表示的话就是 01101100 01001001(共 16 bit,两个字节).程序解析的时候知道是 UTF-16 就把两个字节当成一个单元来解析.这个很简单.

用 UTF-8 表示"汉"

用 UTF-8 就有复杂点.因为此时程序是把一个字节一个字节的来读取,然后再根据字节中开头的 bit 标志来识别是该把 1 个还是两个或三个字节做为一个单元来处理.

0xxxxxxx,如果是这样的 01 串,也就是以 0 开头后面是啥就不用管了 XX 代表任意 bit.就表示把

一个字节做为一个单元.就跟 ASCII 完全一样.

110xxxxx 10xxxxxx.如果是这样的格式,则把两个字节当一个单元

1110xxxx 10xxxxxx 10xxxxxx 如果是这种格式则是三个字节当一个单元.

这是约定的规则.你用 UTF-8 来表示时必须遵守这样的规则.我们知道 UTF-16 不需要用啥字符来做标志,所以两字节也就是 2 的 16 次能表示 65536 个字符.

而 UTF-8 由于里面有额外的标志信息,所有一个字节只能表示 2 的 7 次方 128 个字符,两个字节只能表示 2 的 11 次方 2048 个字符.而三个字节能表示 2 的 16 次方,65536 个字符.

由于"汉"的编码 27721 大于 2048 了所有两个字节还不够,只能用三个字节来表示.

所有要用 1110xxxx 10xxxxxx 10xxxxxx 这种格式.把 27721 对应的二进制从左到右填充 XXX 符号(实际上不一定从左到右,也可以从右到左,这是涉及到另外一个问题.等会说.

刚说到填充方式可以不一样,于是就出现了 Big-Endian,Little-Endian 的术语.Big-Endian 就是从左到右,Little-Endian 是从右到左.

由上面我们可以看出 UTF-8 在局部的字节错误（丢失、增加、改变）不会导致连锁性的错误，因为 UTF-8 的字符边界很容易检测出来，所以容错性较高。

Unicode 版本 2

前面说的都是 unicode 的第一个版本.但 65536 显然不算太多的数字,用它来表示常用的字符是没一点问题.足够了,但如果加上很多特殊的就也不够了.于是从 1996 年开始又来了第二个版本.用四个字节表示所有字符.这样就出现了 UTF-8,UTF16,UTF-32.原理和之前肯定是完全一样的,UTF-32 就是把所有的字符都用 32bit 也就是 4 个字节来表示.然后 UTF-8,UTF-16 就视情况而定了.UTF-8 可以选择 1 至 8 个字节中的任一个来表示.而 UTF-16 只能是选两字节或四字节..由于 unicode 版本 2 的原理完全是一样的,就不多说了.

前面说了要知道具体是哪种编码方式,需要判断文本开头的标志,下面是所有编码对应的开头标志

EF BB BF	UTF-8
FE FF	UTF-16/UCS-2, little endian
FF FE	UTF-16/UCS-2, big endian
FF FE 00 00	UTF-32/UCS-4, little endian.
00 00 FE FF	UTF-32/UCS-4, big-endian.

其中的 UCS 就是前面说的 ISO 制定的标准,和 Unicode 是完全一样的,只不过名字不一样.ucs-2

对应 utf-16,ucs-4 对应 UTF-32.UTF-8 是没有对应的 UCS

UTF-16 并不是一个完美的选择，它存在几个方面的问题：

UTF-16 能表示的字符数有 6 万多，看起来很多，但是实际上目前 Unicode 5.0 收录的字符已经达到 99024 个字符，早已超过 UTF-16 的存储范围；这直接导致 UTF-16 地位颇为尴尬——如果谁还在想着只要使用 UTF-16 就可以高枕无忧的话，恐怕要失望了

UTF-16 存在大小端字节序问题，这个问题在进行信息交换时特别突出——如果字节序未协商好，将导致乱码；如果协商好，但是双方一个采用大端一个采用小端，则必然有一方要进行大小端转换，性能损失不可避免（大小端问题其实不像看起来那么简单，有时会涉及硬件、操作系统、上层软件多个层次，可能会进行多次转换）

另外，容错性低有时候也是一大问题——局部的字节错误，特别是丢失或增加可能导致所有后续字符全部错乱，错乱后要想恢复，可能很简单，也可能会非常困难。（这一点在日常生活里大家感觉似乎无关紧要，但是在很多特殊环境下却是巨大的缺陷）

目前支撑我们继续使用 UTF-16 的理由主要是考虑到它是双字节的，在计算字符串长度、执行索引操作时速度很快。当然这些优点 UTF-32 都具有，但很多人毕竟还是觉得 UTF-32 太占空间了。

反过来 UTF-8 也不完美，也存在一些问题：

文化上的不平衡——对于欧美地区一些以英语为母语的国家 UTF-8 简直是太棒了，因为它和 ASCII 一样，一个字符只占一个字节，没有任何额外的存储负担；但是对于中日韩等国家来说，UTF-8 实在是太冗余，一个字符竟然要占用 3 个字节，存储和传输的效率不但没有提升，反而下降了。所以欧美人民常常毫不犹豫的采用 UTF-8，而我们却老是要犹豫一会儿

变长字节表示带来的效率问题——大家对 UTF-8 疑虑重重的一个问题就是在于其因为是变长字节表示，因此无论是计算字符数，还是执行索引操作效率都不高。为了解决这个问题，常常会考虑把 UTF-8 先转换为 UTF-16 或者 UTF-32 后再操作，操作完毕后再转换回去。而这显然是一种性能负担。

当然，UTF-8 的优点也不能忘了：

字符空间足够大，未来 Unicode 新标准收录更多字符，UTF-8 也能妥妥的兼容，因此不会再出现 UTF-16 那样的尴尬

不存在大小端字节序问题，信息交换时非常便捷

容错性高，局部的字节错误（丢失、增加、改变）不会导致连锁性的错误，因为 UTF-8 的字符边界很容易检测出来，这是一个巨大的优点（正是为了实现这一点，咱们中日韩人民不得不忍受 3 字节 1 个字符的苦日子）

那么到底该如何选择呢？

因为无论是 UTF-8 和 UTF-16/32 都各有优缺点，因此选择的时候应当立足于实际的应用场景。例如在我的习惯中，存储在磁盘上或进行网络交换时都会采用 UTF-8，而在程序内部进行处理时则转换为 UTF-16/32。对于大多数简单的程序来说，这样做既可以保证信息交换时容易实现相互兼容，同时在内部处理时会比较简单，性能也还算不错。（基本上只要你的程序不是 I/O 密集型的都可以这么干，当然这只是我粗浅的认识范围内的经验，很可能会被无情的反驳）

稍微再展开那么一点点……

在一些特殊的领域，字符编码的选择会成为一个很关键的问题。特别是一些高性能网络处理程序里更是如此。这时采用一些特殊的设计技巧，可以缓解性能和字符集选择之间的矛盾。例如对于内容检测/过滤系统，需要面对任何可能的字符编码，这时如果还采用把各种不同的编码都转换为同一种编码后再处理的方案，那么性能下降将会很显著。而如果采用多字符编码支持的有限状态机方案，则既能够无需转换编码，同时又能够以极高的性能进行处理。当然如何从规则列表生成有限状态机，如何使得有限状态机支持多编码，以及这将带来哪些限制，已经又成了另外的问题了。