

深入图解字符集与字符集编码

本文转摘自网友国栋发表在开源中国网站博客上的文章，仅作学习使用。在每章节的前面，会给出相应的文章链接地址。

第一章 charset vs encoding

原文地址: <http://my.oschina.net/goldenshaw/blog/304493>

摘要: charset 与 encoding 的差别在哪? charset=utf-8 与 encoding=utf8 哪种写法更规范? 本文将探讨这两者的联系与区别。

许多时候, 字符集与编码这两个概念常被混为一谈, 但两者是有差别的, 作为深入理解的第一步, 首先要明确: **字符集与字符集编码是两个不同层面的概念**。

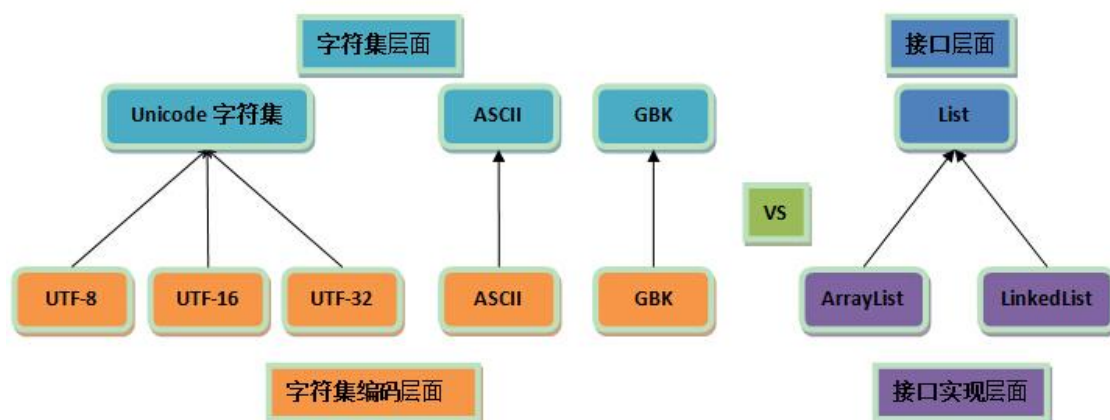
字符集与字符集编码是两个不同层面的概念

| charset 是 character set 的简写, 即字符集。

| encoding 是 charset encoding 的简写, 即字符集编码, 简称编码。

与接口及接口实现的对比

可以把这两者与接口及接口实现做个对比:



从这里可以很清楚地看到,

| 编码是依赖于字符集的, 就像代码中的接口实现依赖于接口一样;

| 一个字符集可以有多个编码实现, 就像一个接口可以有多个实现类一样。

具体例子及规范用法

可以简单看两个例子, 一个自于 html 文件, 用的是 charset:

| `<meta http-equiv="content-type" content="text/html; charset=utf-8">`

另一个来自于 xml 文件, 用的是 encoding:

| `<?xml version="1.0" encoding="UTF-8"?>`

哪一种用法更规范呢? 显然是后者, 它更加准确地区分了字符集与编码的概念。

为什么要严格区分字符集与编码这两个概念?

字符集与编码一对一的情形

有很多的字符编码方案, 一个字符集只有唯一一个编码实现, 两者是一一对应的。比如 GB2312, 这种情况, 无论你怎么去称呼它们, 比如“GB2312 编码”, “GB2312 字符集”, 说来说去其实都是一个东西, 可能它本身就没有特意去做什么区分, 所以无论怎么说都不会错。

为什么一对一是一种普遍的情况呢？

我们以 GB2312 为例，GB=Guo Biao=国标=国家标准，标准出来本来就为了统一，你一个标准弄出 N 个编码实现来，你让人家用哪个呢？

字符集与编码一对多的情形

事情到了 Unicode 这里，变得不一样了，唯一的 Unicode 字符集对应了三种编码：UTF-8, UTF-16, UTF-32。如果还是这么笼统地去称呼，就很容易搞混了。

为什么 Unicode 这么特殊？

人们弄出新的字符集标准，驱动力无外乎是旧的字符集里的字符不够用了。

Unicode 的目标是统一所有的字符集，囊括所有的字符，所以字符集发展到它这里就到头了，再去整什么新的字符集就没必要也不应该了。

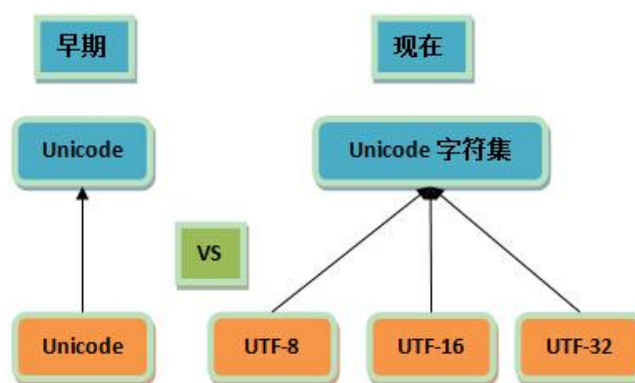
但如果觉得它现有的编码方案不太好呢？在不能弄出新的字符集情况下，只能在编码方面做文章了，于是就有了多个实现，这样一来传统的一一对应关系就打破了。

我们严格地区分字符集与编码两个概念，理由就在这里。

| 指定了编码，它所对应的字符集自然就指定了，编码才是我们最终要关心的。

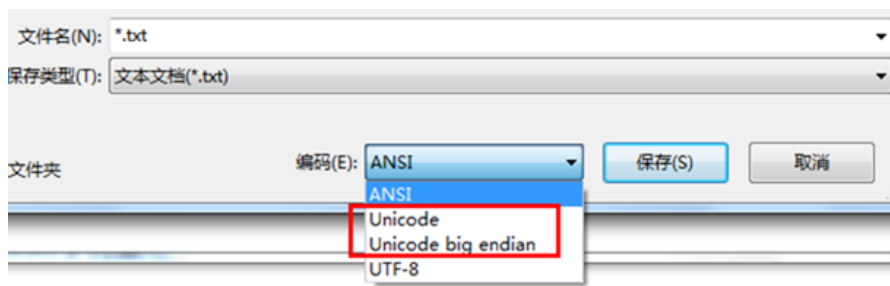
Unicode 早期与现在的对比

让我们来看一个图，它展现了 Unicode 早期与现在的一些差别：



| 注：由于历史方面的原因，你还会在不少地方看到把 Unicode 和 UTF-8 混在一块的情况，
| 这种情况下的 Unicode 通常就是 UTF-16 或者是更早的 UCS-2 编码，
| 在后面的篇章中我们会进一步分析。

下面是“记事本程序”保存时的一个截图，是 Unicode 的一个不规范使用，这里的 Unicode 就是指 UTF-16：



我们现在说了不少 Unicode，但我们还没有准确地定义这个词，我们可能有些疑问，如：

Unicode 是不是就是指 Unicode 字符集呢？

Unicode 编码是不是就是指 UTF-8, UTF-16, UTF-32 这些东西呢？

这些问题没办法简单用是与否回答，在第二篇里会做进一步探讨。

第二章 编号 vs 编码

原文地址: <http://my.oschina.net/goldenshaw/blog/305805>

摘要: 编号是字符到最终编码的一个过渡层与抽象层,起着承上启下的作用,它与最终编码在形式上也常常很相似,在 Unicode 中,码点 (code point) 扮演的正是编号的角色。广义而言,编号其实也是一种编码。

在深入研究**字符集编码**(简称**编码**)之前,我们先引入一个概念:**编号 (code)**,引入它是为了更好地与**编码 (encode)**相区分。

- | 如果你对 Unicode 有深入了解,你也许已经意识到了 Unicode 中码点 (code point) 扮演的正是编号的角色。类似的还有 GB 系列中所谓的区位码。

其实叫什么并不重要,爱咋咋地,我并不关心。但乱叫容易叫混了,比如把码点也叫成 Unicode 编码,这里先把这些归入到编号概念。为区别起见,用黑色加粗的编码特指字符集编码。

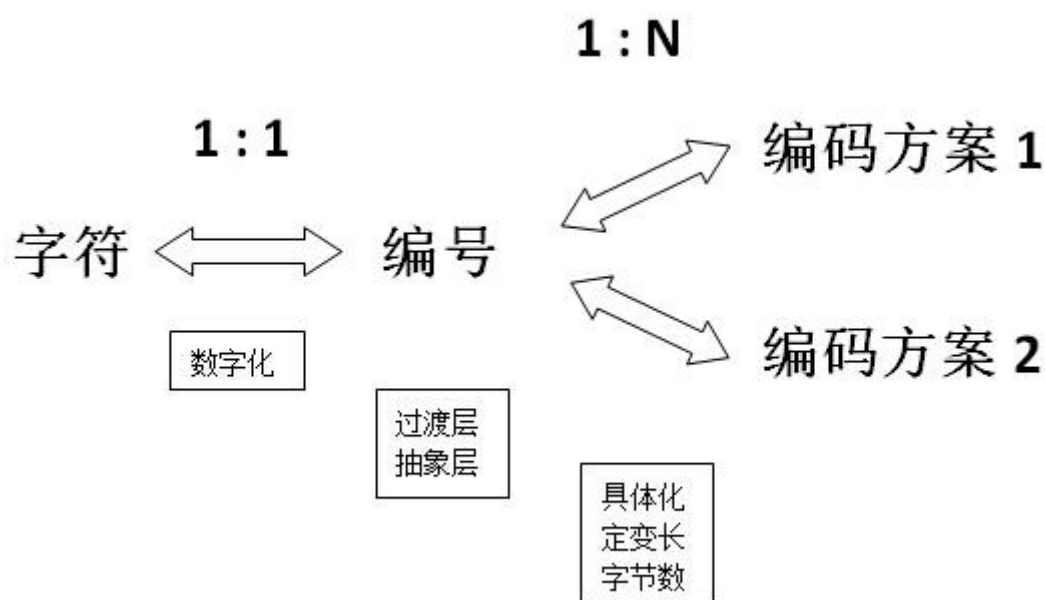
- | 到了后面你甚至会为字符集编码的边界在哪而困惑,为它的准确定义而纠结,
- | 不过到那时你已经属于”难得糊涂“了,编号这一概念你也可以把它丢到爪哇国去了。

编号是什么?

编号可以看作是字符与编码中间的一个抽象层,过渡层:

- | 广义上说,编号也可看成是某种编码;
- | 狭义上说,编码也可视为某种编号。

图: 字符<-->编号<-->编码



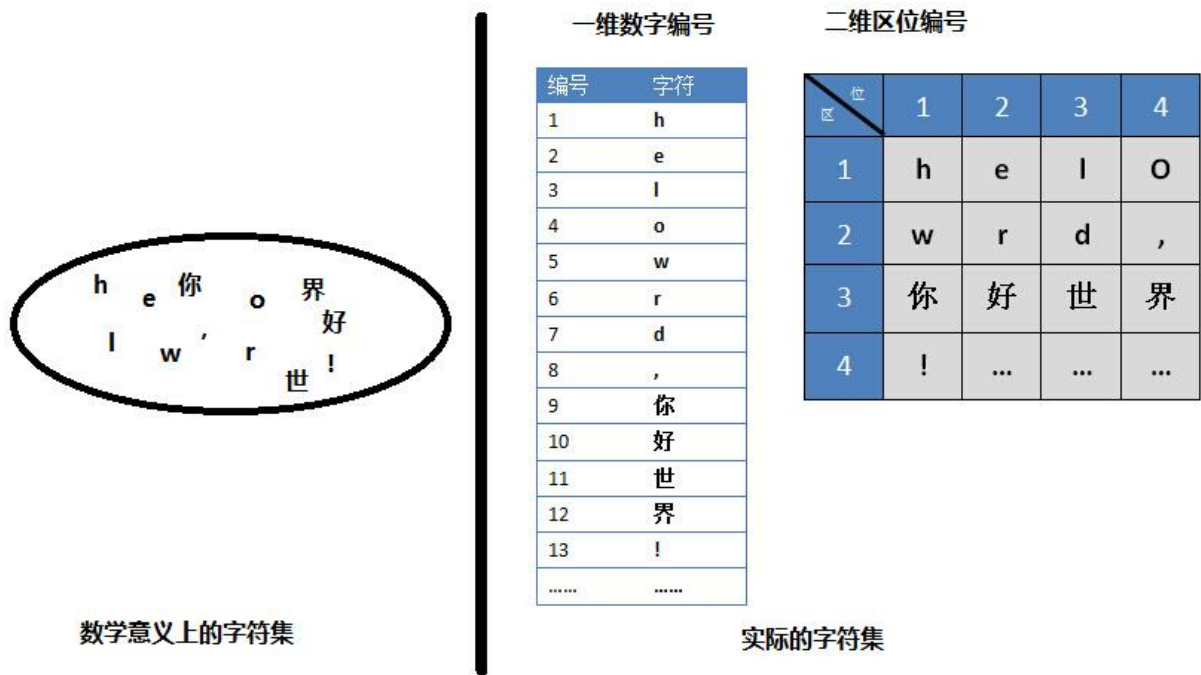
- | 编号与编码的主要区别在于编号不涉及具体使用多少字节来表示、
- | 是用定长还是变长方案等细节问题。编号仅仅是一个抽象的概念。

在进一步比较编号与编码之前,我们先看看编号是如何来的。

字符集通常是带编号的有序集合

- | 数学意义上的集合 (Set): 一组不重复的, 无序的的元素。
- | 一般意义上的字符集: 不重复的, 带编号的有序的字符合集。

图：数学意义上的字符集对比实际的字符集



编号是如何来的？

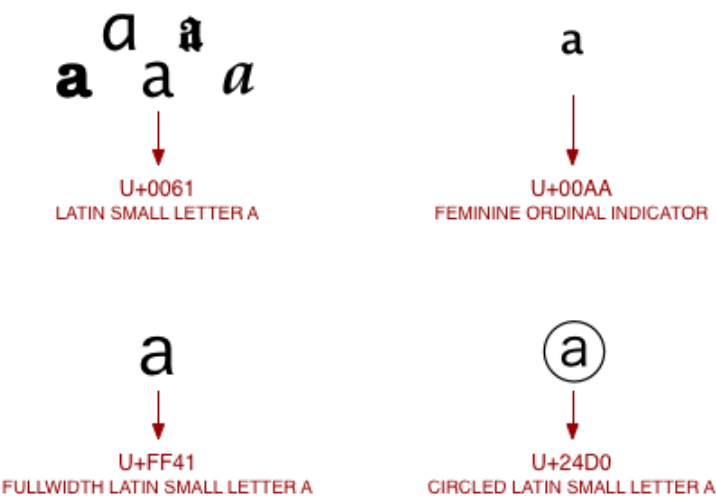
字符被整理出来之后往往数量众多，通常是置于表格之中。出于统计方面的原因，人们通常用一个数字来编号。表格本身就是一种有序的暗含了编号的形式，哪怕你没有明确地为其编号，人们拿到这个表格也会说：“嘿，第一个字符是 XXX！”“这里的一不就是编号吗？”

- 可以把整理的工作视作是由一群完全不懂计算机的语言文字学家来完成的，
- 他们甚至连字节是什么都没听说过。所谓编号是抽象的就是说它仅仅是一个数字而已。

怎样才算不重复？

这里不打算去讨论哲学意义上的等同，你可能会碰到有些字符彼此间长得非常像，在有了编号后，我们可以简单地说，只要是编号不同的两个字符就是不重复的。

图：一些很相似的字符，图片来自 http://wiki.secondlife.com/wiki/Unicode_In_5_Minutes



我们可以看到至少有三个码点上的 a 是非常像的。其中的 U+FF41 中所谓的 FULLWIDTH（全宽）其实就是全角，也即两个半角宽度，我想我们对此都很熟悉。

- 当初 GB2312 出来时，仗着自己编码空间大，把 ASCII 里那些字母符号之类的又重复
- 弄出一套所谓的全角版本来，后来 Unicode 又把这些又再收罗了过去。

编号一定是一个数字吗？

不一定！它可以是数字对，或者你叫它复数，二元数啥的，随便你。但只要它是离散可数量子化的，它自然也可以转换成唯一的一个数字。参见前面图中的二维区位编号，我们用数字对 (1, 1) 编号“h”这个字符。(1, 1) 可以简单转换成 11，然后可以进一步映射到从 0 或者 1 开始的编号。

编号是连续的吗？

有序不意味着连续。这里需要说明的一点是，从前面的叙述来看，编号早于编码，但实际情况是人们通常是一起考虑这两者的，编号反过来会受到编码考虑的影响，这样做只是为了让从编号到编码的映射或者叫转换更加方便。这些影响包括：

- 1、如果按日常习惯，编号通常应该从 1 开始，受编码影响，编号也从 0 开始。
- 2、编号写成十进制是更自然的方式，受编码影响，编号通常也以十六进制形式来书写，并写成固定的位数，不够时就在前面填充 0，比如把 48 写成 0048。Unicode 的码点位数是 4-6 位，不够 4 位的补足 4 位，超过的是多少位就多少位，比如：U+1D11E 就是一个五位数的编号。
- 3、为了以后的扩展方便，编码常常会跳过某些码位，甚至会保留大片的区域未定义或作保留用途。比如 Unicode 有所谓的代理区(surrogate area)，后续我们会进一步了解。编号因此也跳过这些。（其实到了后面你会发现，究竟谁影响谁还真不好说！不过等你明白之时这些已经不重要了。。。）

| 总之一句话就是让映射规则尽可能简单。

图：编号最终与编码几乎一样的一种可能情形，为简单起见，使用十进制。

最初编号	修正后编号	编码 1(定长 2 位)	编码 2(定长 4 位)	字符
1	00	00	0000	h
2	01	01	0001	e
3	02	02	0002	l
4	03	03	0003	o
5	04	04	0004	w
6	05	05	0005	r
7	06	06	0006	d
8	07	07	0007	,
9	08	08	0008	你
10	09	09	0009	好
11	10	10	0010	世
12	11	11	0011	界
13	12	12	0012	!
*** **	*** **	*** **	*** **	*** **

看起来一样，但两者并不是一回事

编码与编号的区别？

你可能会说，那这样它们还有什么区别？你的确可以把编号也说成是编码。

但事情并不总是这样，这种相似性确实迷惑了很多，特别是 Unicode，很多人把码点说成是 Unicode 编码，这种说法本身并没有错，这取决于你如何定义编码，但他们是否意识码点仅仅一种抽象的编码呢？为了区别，Unicode 把最终的具体编码称为 UTF(Unicode Transformation Format)，即所谓的 Unicode 转换格式。

| 所谓转换，其实就是把抽象的数字映射到具体的，最终的编码上来。

Unicode 编码的两个层面



抽象编码层面

把一个字符编码到一个数字（不涉及每个数字用几个字节表示，是用定长还是变长表示等具体细节）

具体编码层面

即 UTF，把抽象编码层面的数字编码成最终的存储形式，需要明确是用定长还是变长；定长的话定几个字节；用变长的话有哪几种字节长度，具体如何去实现等等。（注：在上一层面，字符与数字已经实现一一对应，对数字编码实质就是对字符编码）

什么是编码？（广义）

编码是一个非常宽泛的概念！虽然我们前面一直用编码特指字符集编码，但这只是一种狭义的理解，广义的理解则有很多：

- 文字是对声音的编码
- 照相机，摄像机把光信号编码成图像及视频
- 我们还经常能看到条形码，二维码，这些都是编码

在《编码：隐匿在计算机软硬件背后的语言》（Code: The Hidden Language of Computer Hardware and Software）一书中，参见[豆瓣读书](#)，作者提到了莫尔斯电码(Morse Code)。以及布莱叶盲文(Braille Code)，这些都是编码的例子。

莫尔斯电码 (Morse code)						布莱叶盲文 (Braille code)									
A	.-	J	.-.-.-	S	...	⠁	⠃	⠉	⠑	⠑	⠉	⠑	⠉	⠑	⠉
B	---.	K	-.--	T	-	⠃	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑
C	.-.-.	L	.-..	U	..-	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
D	---.	M	--	V	...-	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
E	.	N	-.	W	-.--	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
F	..-.	O	---	X	-.--	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
G	--.	P	.-.-	Y	-.--	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
H	Q	---.	Z	---.	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉
I	..	R	.-.			⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉	⠑	⠉

在《信息简史》（The Information: A History, A Theory, A Flood）一书中，参见[豆瓣读书](#)，作者提到了一个有趣的“会说话的鼓”的故事，非洲的一些部落成员之间可以用鼓声来交流非常复杂的讯息，在这里就是用鼓声来编码信息。

字符集编码再审视

回到我们的字符集的例子，虽然我们倾向于认为编码就是指最终存储的形式，比如写入文件时或者放在内存时，又或者是在网络传输的过程中。

但如果我们要说，字符集编码这一概念也可以包含抽象层面的编码，那么这样一种说法也并无不妥，只要你能准确区分这两个层面，你怎么去看待它们都是可以的，还是前面那句话，这取决于你如何定义编码，比如我们可以说 GBK 中的区位码难道不是字符集编码吗？这就取决于你如何看待 GBK 编码这一概念了，是狭义的去看待还是广义的去看待。

我不想为字符集编码的准确定义去争论，在我看来，当你说编码时，只要你自己清楚说的是哪个层面就 ok 了，我们在前面引入了编号，目的是为在尚未澄清之前作更好的区分，如果你现在已经清楚了，就可以把编号丢掉了。

事实上，当人们说到 Unicode 编码时，更常是指它的抽象编码层，即码点这一层面，实际上这才是 Unicode 的核心所在。

| Unicode 的核心就是为每个字符提供唯一的一个数字编号。

| Unicode provides a unique number for every character.

Unicode 编号及编码的一个具体事例

让我们来看一个更加具体的示意图：

字符层面				
H 你 🎵				
码点(Code Point) 括号内为十进制	U+0048(72)	U+4F60 (20320)	U+1D11E (119070)	抽象编码层面 (编号)
UTF	UTF-8	UTF-16	UTF-32	
H	U+0048	48	00 48	00 00 00 48
你	U+4F60	E4 BD A0	4F 60	00 00 4F 60
🎵	U+1D11E	F0 9D 84 9E	D8 34 DD 1E	00 01 D1 1E
	变长 1-4 字节	变长 2 或 4 字节	定长 4 字节	具体编码层面

关于码点如何具体转换成各种编码，这个在后面再作讨论。从图上我们可以初步得出一些结论。比如：

- UTF-8 与 UTF-16 都是变长编码，UTF-32 则是定长编码。
- 码点到 UTF-32 的转换最简单，就是在前面垫 0 垫够 4 字节就行了。
- 码点到 UTF-8 的转换，除了最小那个在数值上一样外，其它两个完全看不出两者的关系。
- 码点到 UTF-16 的转换则是最微妙的，可以看出前两个字符 UTF-16 与码点是完全一致的，但那个大码点（准确地说是超过了 U+FFFF 的码点）则有了很大的变化，长度变成了四字节，值也变得很不一样了。

关于 UTF-16 的误解是很多的，部分可能由于它的名字上带了个 16，让人误以为它是 16 位定长的两字节编码。但正像 UTF-8 并不是仅仅是 8 位一样，UTF-16 也不仅仅是 16 位。

| 事实上,UTF-16 的前身 UCS-2 确实是 16 位定长的编码,它跟码点在形式上就是完全一样了,
| 实际我很怀疑那时候压根就没码点这一说法,那时人们甚至也不说 UCS-2,直接就叫 Unicode!
| 时至今日,你依然可以在不少地方看到把 UTF-16 写成 Unicode 的,然后与 UTF-8 并排在一
| 起,显得不伦不类的,当然了,这是有历史原因的。

UTF-16 为何变成变长了？

简单地说，字符扩充了，目前码点的范围是 U+0000~U+10FFFF（按 Unicode 官方的说法，就这样了，以后也不扩充了），U+10FFFF 是多大呢？大概是 111 万，而 16 位定长的话，撑死了也就 6 万多，所以不变就不行了。

在后面的篇章中，将进一步分析定长，变长的问题，见下一章。

第三章 定长与变长

原文地址: <http://my.oschina.net/goldenshaw/blog/307708>

摘要: 本文深入探讨了定长与变长两种实现, 阐述了定长到变长演变的一些权衡与取舍, 并把它与 CAP 理论作了对比。在最后, 还通过自行实现变长方案的方式来演示变长设计上的一些考虑。

☯, 首先, 这并不是图片, 这是一个 unicode 字符, Yin Yang, 即阴阳符, 码点为 U+262F。如果你的浏览器无法显示, 可以查看[这里](#)。这与我们要讨论的主题有何关系呢? 下面我会谈到。

连续式表示带来的分隔难题

计算机的底层表示

在计算机的最底层, 一切都成了 0 和 1, 你也许见过一些极客(Geek)穿着印有一串 0 和 1 的衣服招摇过市, 像是数字化时代的某种图腾。比如, 这么一串 “0001100101101110001111111000...”, 如果它来自某个文本文件保存后的结果, 我们如何从这一串的 0 和 1 中从新解码得到一个个的字符呢? 显然你需要把这一串的 0 和 1 分成一段一段的 0 和 1, 在讲述编码是如何分隔之前, 我们先看看自然语言的分隔问题。

自然语言的分隔问题

大家是否意识到, 我们的中文句子里字词之间也是连续的呢? 英文里说 “hello world”, 我们说 “你好世界”, “我们 不 需要 在 中间 加 空格!” 在古代, 句与句之间甚至都没有分开, 那时还没有标点! 所以有了所谓的断句问题。让我们来看一个例子:

- 民可使由之不可使知之 ——出自《论语 第八章 泰伯篇》

这么一串十个字要如何去分隔并解释呢?

断法一:

- 民可使由之, 不可使知之。
- 解释: 你可以去驱使你的民众, 但不可让他们知道为什么 (不要去教他们知识)
- 评论: 很显然是站在统治阶级立场的一种愚民论调。

断法二:

- 民可, 使由之; 不可, 使知之。
- 解释: 民众可以做的, 放手让他们去做; 不会做的, 教会他们如何做 (又或: 不可以去做的, 让他们明白为何不可以)
- 评论: 这看来是种不错的主张。

显然, 以上的文字是以某种定长或变长的方式组合在一起的, 但是关于它们如何分隔的信息则被丢弃了, 于是在解释时就存在产生歧义可能了。

编码的分隔

自然语言中我们可以使用空格, 标点来减少歧义的发生。在计算机里, 一切都数字化了, 包括所谓的空格, 标点之类的分隔符。

| 老子说 “道生一, 一生二, 二生三, 三生万物”,

| 计算机则是 “二生万物”, 0 和 1 表示了一切。

在空格与标点都被数字化的情况下, 我们在这一串 01 中如何去找出分隔来呢? 显然我们需要外部的约定。

| 8 位 (bit) 一组的字节是最基本的一个约定, 也是文件的基本单位, 文件就是字节的序列。

| 字节显然就是最基础的一个分隔依据。

定长(Fixed-length)的解决方案

定长仅表明段与段之间长度相同，但没说明是多长。有了字节这一基本单位，我们就可以说得更具体，如定长一字节或者定长二字节。

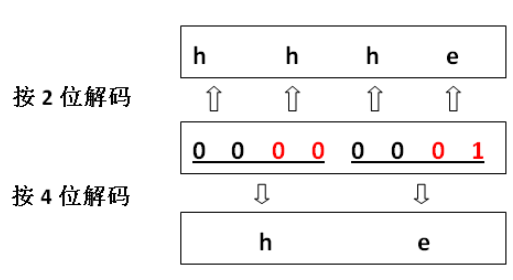
ASCII 编码是最早也是最简单的一种字符编码方案，使用定长一字节来表示一个字符。

下面我们来看一个具体的编码示例，为了方便，采用了十进制，大家看起来也更直观，原理与二进制是一样的。

编号	编码1(定长2位)	编码2(定长4位)	字符
1	00	0000	h
2	01	0001	e
3	02	0002	l
4	03	0003	o
5	04	0004	w
6	05	0005	r
7	06	0006	d
8	07	0007	,
9	08	0008	你
10	09	0009	好
11	10	0010	世
12	11	0011	界
13	12	0012	!
.....

假设我们现在有个文件，内容是 00000001，假如定长 2 位（这里的位指十进制的位）是唯一的编码方案，用它去解码，就会得到“hhhe”（可以对比图上的编码，00 代表 h，所以前 6 个 0 转化成 3 个 h，后面的 01 则转化成 e）。

但是，如果定长 2 位不是唯一的编码方案呢？如上图中的定长 4 位方案，如果我们误用定长 4 位去解码，结果就只能得到“he”（0000 转化为 h，0001 转化为 e）！



毕竟，文件的内容并没有暗示它使用了何种编码！这就好比孔夫子写下“民可使由之不可使知之”时并没有暗示它是 5|5 分隔（民可使由之|不可使知之）还是 2|3|2|3 分隔（民可|使由之|不可|使知之）那样。

如何区分不同的定长（以及变长）编码方式？

答案是：你无法区分！好吧，这么说可能有点武断，有人可能会说 BOM(Byte Order Mark 字节顺序标识)能否算作某种区分手段呢？但也有很多情况是没有 BOM 的。（关于 BOM，我们在后面具体分析 Unicode 编码时再深入去谈）

总之，我想给读者传递的一个信息就是：文本文件作为一种通用的文件，在存储时一般都不会带上其所使用编码的信息。编码信息与文件内容的分离，其实这正是乱码的根源。

我们说无法区分即是基于这一点而言，但另一方面，各种编码方案所形成的字节序列也往往带有某种特征，综合统计学，语言偏好等因素，还是有可能猜测出正确的编码的，比如很多浏览器中都有所谓“编码自动检测”的功能。本章主题主讲定变长，暂时先抛出这一问题，留待后面再讨论。

定长多字节方案是如何来的？

显然，字符集的扩充是主要推动力。定长一字节编码空间撑死了也就 $2^8=256$ 。

| 这点可怜的空间拿到中国来，它能编码的汉字量也就小学一年级的水平。

其实变长多字节方案更早出现，比如 GB2312，采用变长主要为了兼容一字节的 ASCII，汉字则用

两字节表示（这也是迫不得已的事，一字节压根不够用）。随着计算机在全世界的推广，各种编码方案都出来了，彼此之间的转换也带来了诸多的问题。采用某种统一的标准就势在必行了，于是乎天上一声霹雳，Unicode 粉墨登场！

前面已经谈到，Unicode 早期是作为定长二字节方案出来的。它的理论编码空间一下达到了 $2^{16}=65536$ (即 64K，这里 $1K=1024=2^{10}$)。

对于只用到 ASCII 字符的人来说，比如老美，让他们采用 Unicode，多少还是有些怨言的。怎么说呢？比如“he”两个字符，用 ASCII 只需要保存成 6865 (16 进制)，现在则成了 00680065，前面两个毫无作用的 0 怎么看怎么碍眼，原来假设是 1KB 的文本文件现在硬生生就要变成 2KB，1GB 的则变成 2GB！

可是更糟糕的事还在后头，在老美眼中，16 位的空间已经算是天量了！要知道一字节里 ASCII 也仅仅用了一半（后面将会看到，这一特性为各种变长方案能兼容它提供了很大便利！），而且这一半里还有不少控制符。可随着整理工作的深入，人们发现，16 位空间还是不够！！

| 说起来我们的中文可是字符集里面的大头。“茴字有四种写法”，上大人孔乙己的这句名言
| 想必大家还有点印象。据说有些新近整理的汉语字典收录的汉字数量已经高达 10 万级别！
| 我的天！这里很多字怕是孔乙己先生也未必认得了！

那现在该咋办呢？如果还是定长的方案，眼瞅着就要奔着四字节而去了。

| 计算机界有动不动就翻倍的优良传统，比如从 16 位机一下就到 32 位，
| 32 位一下又到了 64 位。当然了，这里面是有各种权衡的，包括硬件方面的。

那些看到把 6865 保存成 00680065 已经很不爽的人，现在你却对他们说，“嘿，伙计，可能你需要进一步存成 0000006800000065...”。容量与效率的矛盾在这时候开始激化。

容量与效率的矛盾

首先，需要明确一下：

- 所谓容量，这里指用几个字节表示一个字符，显然用的字节越多，编码空间越大，能表示更多不同的字符，也即容量越大。
- 所谓效率，当表示一个字符用的字节越多，所占用的存储空间也就越大，换句话说，存储（乃至检索）的效率降低了。

如果说效率是阴，那么容量就是阳。（我没还没忘记自小学语文老师就开始教导的，写作文要遥相呼应。）



我们说定长不是问题，关键是定几位。定少了不够用，定多了太浪费。定得恰到好处？可怎样才算恰好呢？你可能会说，至少要能容纳所有字符吧？但重要的事实是并非所有的字符所有的人都用得上！哪怕用得上，也可能是偶尔用上，多数时候还是用不上！

| 字符之间并不是平等的。用数学的语言来说，每个字符出现的机率不是等概率的，
| 但表示它们却用了同样长度的字节。

如果你对前一篇所发的莫尔斯电码图还有印象，你就会发现，字母 e 只用了一个点（dot）来编码。

其它字母可能觉得不公平，为啥我们就要录入那么多个点和划（dash）才行呢？这里面其实是有统计规律支撑的。e 出现的概率是最大的。z 你能想到什么？zoo 大概很多人能想到，厉害一点可能还能想到 zebra（斑马），Zuckerberg（扎克伯格），别翻字典！你还能想到更多不？但含有的 e 的单词则多了去了。zebra 中不就有个 e 吗，Zuckerberg 中还两个 e 呢！

| 在存储图片时，一个像素点用几个位来表示也是一个很值得考究的问题。
| 你也许听过所谓的 24 位真彩色，这暗示了一个像素使用了高达 3 个字节来表示。

| 24 位的空间可以表示高达 1600 多万种颜色，但各种颜色的出现概率在均衡度上肯定要好于字符。

回到我们的主题，虽然很多字连我们的孔乙己先生见了都要摇头，可还是有少部分人会用到它们，比如一些研究古汉字的学者。有些人取名还喜欢弄些偏僻字，所以很多人口登记方面的系统也有这个超大字符集的需求。

好吧，我们不能不顾这些人需求。那么有可能在定长方案的框架下解决这一容量与效率的矛盾吗？答案是否定的！

| 矛盾是事物发展的动力，下面我们将看到定长方案的简单性使它无法缓和容量与效率的冲突，
| 平衡这一对矛盾的努力最终推动了编码方案从定长演变到变长，
| 事情也由此从简单变得复杂了。

CAP 理论及扩展

CAP 是什么玩意？

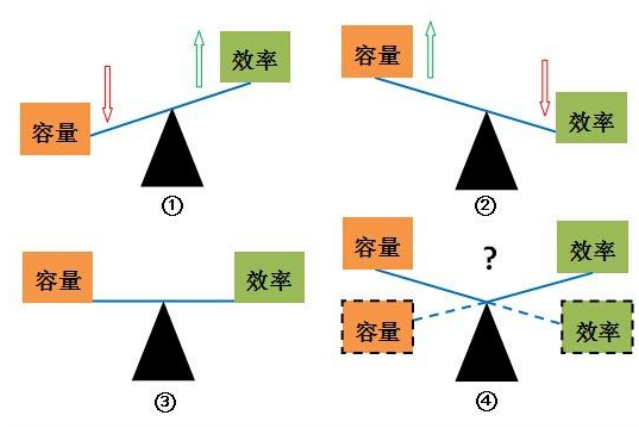
著名的 Brewer 猜想说：对于现代分布式应用系统来说，数据一致性 (Consistency)、系统可用性 (Availability)、服务规模可分区性 (Partitioning) 三个目标 (合称 CAP) 不可同时满足，最多只能选择两个。

你可能要问，这貌似跟我们要讨论的问题风牛马不相及？别着急，我们可以借鉴他的这种思想，扩展到我们的问题上来。

两个维度

我们所应对的问题常常很抽象，有时借助某些隐喻 (metaphor) 可以帮助我们来理解。天平就是一个很好的隐喻。

先看图中四个天平。你叫它跷跷板我也没意见，反正我没打算吃美术这碗饭！（嗯，也许是少个了指针的缘故，希望这空指针的天平不要引发什么异常。）



这幅图表示的就是定长方案下容量与效率的一种约束关系。

- 1、容量小则效率高
- 2、容量大则效率低
- 3、容量与效率均不能让人满意！
- 4、容量与效率均较好，但这是不可能的情形！

让我们具体解释一下：

天平的蓝色横梁一种刚性约束的隐喻。所谓刚性，这里简单理解成不能变形就是了。

天平的两端的容量与效率是它的两个维度，或者说两个自由度。不必去纠结物理学上的定义，简单理解就是它们能自由上下就好了。但我们可以看到，由于受到横梁的约束，两个维度同时向上是不可能的！它们的相互运动呈现出彼消此长的关系。

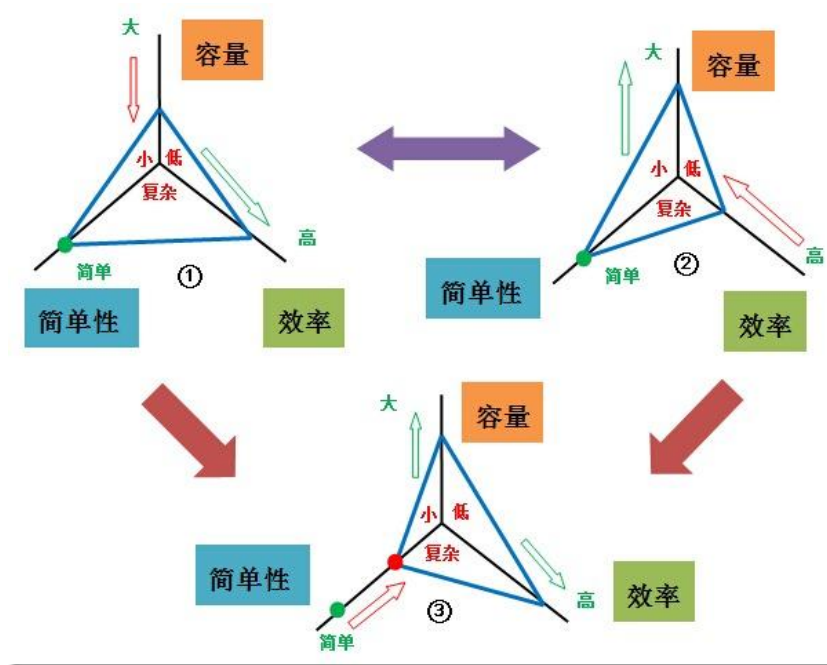
天平可以隐喻很多，比如安全性与便利性。为什么要我录入验证码？为什么支付宝登录与支付要用不一样的密码？为何输入密码还不够，还要什么手机验证码？这些都很不方便呀！当你觉得只有一个前门还不够方便时你又加开了个后门，但你是否想过在方便自己的同时也“方便”了小偷呢？

三个维度

好了，现在要再次拓展一个维度了，以使得它更像 CAP 理论。

还有一个维度在哪呢？我们说容量大是好，效率高是好，我们为何青睐定长方案呢？因为定长它简单，简单当然也是好，复杂就不好了。这就是第三个维度——简单性（你也可以叫成复杂性，意思是一样的）。

先深呼吸一下，让我们再看一个图：



首先这里多了一个维度，天平模型不足以表达了，改用三条互成 120 度的直线表示这三个维度。越往里，红色的字代表是越差的情况；越往外，绿色的字代表是越好的情况。

图中的约束在哪呢？就在蓝色的三角形上，它有一个固定的周长，这就代表了它的约束。也许我们把它想像成一条首尾相接的固定长度的钢丝绳更好，在图中它只是被拉成了三角形。它可以在三个维度上运动，这让它比天平的横梁更灵活，但它的长度不能被拉伸，它不是橡皮绳！！

这幅图能告诉我们什么？

- 图 1 跟图 2，当我们维持简单性不变时，容量与效率的关系其实跟天平模型中是相似的，也是一种彼消此长的关系。

- 图 3，让简单性下降（换句话说就是变复杂了），才能为其它两个维度腾出“余量”来。即是说你要“牺牲”简单性来调和容量与效率的矛盾。

我们能从模型得到什么启示呢？

一、事物的多个方面往往是相互制衡的

在前面的图中，我们用钢丝绳来形象隐喻这种制衡，深刻理解制衡是各种直觉与预见性的前提，当我们作出决定时，便能够预判出可能的后果。深层次的矛盾暗示了我们有些冲突是不可避免的，同时为我们找到正确解决问题的路径指明了方向。

二、制衡的局面暗示了凡事有代价，站在一个全局上去考量，我们常常需要在各方面达成某种平衡与妥协。

| 举个例子，分层会对性能有所损害，但不分层又会带来紧耦合的问题。

| 很多时候，架构就是关于平衡的艺术。

| 如果我们能明白这一点，就不会为无法找到“完美”方案而苦恼。

三、复杂性从根本上是由需求所决定的。我们既要求容量大，又要求效率高，这种要求本身就不简单，因此也无法简单地解决。

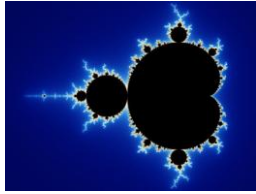
| 你功能做好了，用户说性能还不行；你性能达标了，用户说界面还太丑。。

| 丫的能不能先把首付款付清了再跟哥提要求？！

为什么谈这些理论？

一方面我不想仅仅为谈定变长而只谈定变长，单纯谈理论又往往过于抽象。我想说的一点是“我们在编码上所遇到的困境往往不过是我们软件开发过程中遇到的各种困境的一个缩影”。

下图是所谓的芒德布罗集 (Mandelbrot set), 是分形 (Fractal) 理论中的一个重要概念。



很多问题需要我们站在更高层面上去观察与思考时才能发现它们的某些相似性或者说共性，这些共性被抽象出来也就形成了我们的理论。

不识庐山真面目，只缘身在此山中。——苏轼《题西林壁》

另一方面,因为这种相似性,我们在编码问题上得到启示也能够指导我们去解决其它领域的问题。我想这就是这些理论的意义所在。

调和矛盾的努力，兼容考虑与变长方案的引入

通过前面分析，我们知道，定长二字节方案无法满足容量增长，转向定长四字节又会引发了效率危机，最终，Unicode 编码方案演化成了变长的 UTF-16 编码方案。那么 UTF-8 方案又是如何来的呢？为何不能统一成一个方案呢？搞这么多学起来真头痛！

我们前面提到了有一群 ASCII 死忠对 Unicode 统一使用二字节编码 ASCII 字符始终是有不满的，现在眼见简单的定长方案也不行了，他们中的一些大牛终于忍无可忍。既然决定抛弃定长，他们决定变得更彻底，于是这帮人揭竿而起，捣鼓出了能与 ASCII 兼容的 UTF-8 方案。（注：真实历史也许并非如此，我不是考据癖，以上叙述大家悠着看就是了，别太当真。）

如今装个逼还分高低格，大牛不折腾，谁又知道他们是大牛呢？只是可怜了我们这些码农，你还在苦苦研究 sql，忽如一夜春风来，Nosql 菊花朵朵开。（貌似应开在秋冬季？）

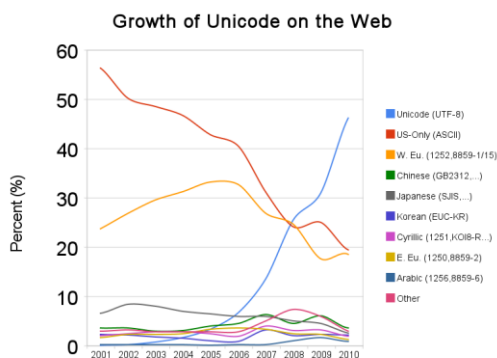
UTF-16 用所谓的**代理对 (surrogate pair)**来编码 U+FFFF 以上的字符。在采用了变长之后，事情变得复杂了，以后我们还将继续探讨**代理区**，**代理对**，**编码单元 (Code Unit)**等一系列由此而来的概念。

这种变化甚至还影响到对字符串长度的定义，比如，在 java 中，你可能认为包含一个字符的字符串它的长度就是 1，但现在，一个字符它的长度也可能是 2！这样的字符也无法用 char 来存储了。

UTF-8 因为能兼容 ASCII 而受到广泛欢迎，但在保存中文方面，要用 3 个字节，有的甚至要 4 个字节，所以在保存中文方面效率并不算太好，与此相对，GB2312，GBK 之类用两字节保存中文字符效率上会高，同时它们也都兼容 ASCII，所以在中英混合的情况下还是比 UTF-8 要好，但在国际化方面及可扩展空间上则不如 UTF-8 了。

其实 GBK 之后又还有 GB18030 标准，采用了 1, 2, 4 字节变长方案，把 Unicode 字符也收录了进来。GB18030 其实是国家强制性标准，但感觉推广并不是很给力。

目前，UTF-8 方案在越来越多的地方有成为一种默认的编码选择的趋势。下面是一张来自 wiki 百科的图片，反映了 UTF-8 在 web 上的增长。



- | 在软件开发的各个环节强制统一采用 UTF-8 编码，依旧是避免乱码问题的最有效措施，
- | 没有之一。技术人员也许更偏爱技术问题技术解决，但不得不承认有时行政手段更加高效！

变长(Variable-length)的编码方案

好了，现在是时候谈谈变长方案的实现问题了，在这里将通过尝试自己设计来探讨这一问题。

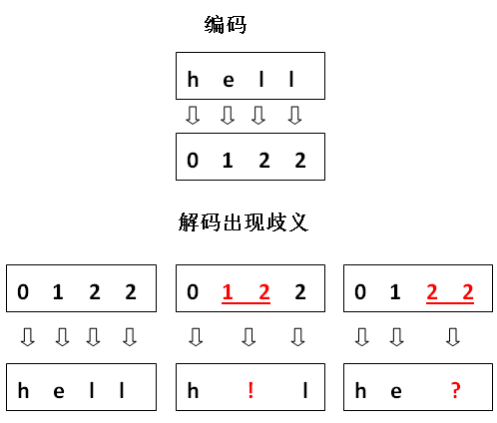
- | 变长设计的核心问题自然就是如何区分不同的变长字节，只有这样才能在解码时不发生歧义。

利用高位作区分

还是以前面的例子来看，我们设计了几种变长方案

编号	变长编码方案1	变长编码方案2	字符
1	0	0	h
2	1	1	e
3	2	2	l
4	3	3	o
5	4	4	w
6	5	50	r
7	6	51	d
8	7	52	,
9	8	53	你
10	9	54	好
11	10	55	世
12	11	56	界
13	12	57	!
.....

第一种方案的想法很美好，它试图跟随编号来自然增长，它还是可以编码的，但在解码时则遇到了困难。让我们来看看。

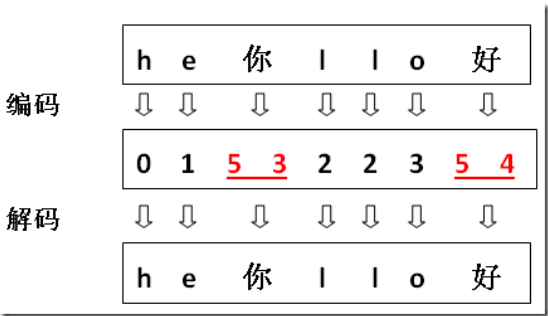


可见，由于低位的码位被“榨干”了，导致单个位与多位间无法区分，所以这种方案是行不通的。

第二种方案，低位空间有所保留，5 及以上的就不要使用了。然后通过引入一条变长解码规则：

- 从左向右扫描，读到 5 以下数字按单个位解码；读到 5 或以上数字时，把当前数字及下一个数字两位一起读上来解码。

让我们来看个实例：



这种方案避免了歧义，因此是可行的方案，但这还是非常粗糙的设计，如果我们想在这串字符中搜索“o”这个字符，它的编码是 3，这样在匹配时也会匹配上 53 中的 3，这种设计会让我们在实现匹配算法时困难重重。我们可以在跟随位上也完全舍弃低位的编码，比如以 55，56，57，58，59，65，66…这样的形式，但这样也会损失更多的有效编码位。

GB2312, GBK, UTF-8 的基本思想也是如此。下面也简单示例一下（0, 1 代表固定值；黑色的 X 代表可以为 0 或 1，为有效编码位）：

GB2312,GBK 编码变长字节间区分的基本原理

0XXXXXXXX1XXXXXXXX1XXXXXXXX0XXXXXXXX

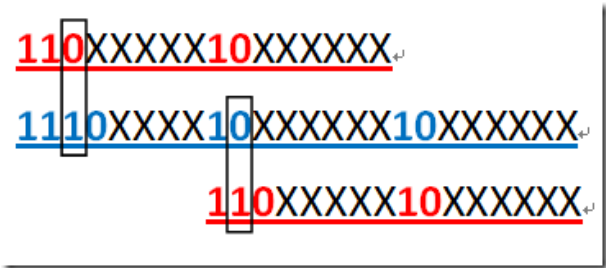
UTF-8 编码变长字节间区分的基本原理

0XXXXXXXX110XXXX10XXXX1110XXXX10XXXX10XXXX11110XXX10XXXX10XXXX10XXXX

注：GBK 第二字节最高位也可能为 0。

其实关键就在于用高位保留位来做区分，缺点就是有效编码空间少了，可以看到三字节的 UTF-8 方式中实际有效的编码空间只剩两字节。但这是变长方案无法避免的。

我们还可以看到，由于最高位不同，多字节中不会包含一字节的模式。对于 UTF-8 而言，二字节的模式也不会包含在三字节模式中，也不会的四字节中；三字节模式也不会四字节模式中，这样就解决上面所说的搜索匹配难题。你可以先想想看为什么，下面的图以二，三字节为例说明了为什么。



可以看到，由于固定位上的 0 和 1 的差别，使得二字节既不会与三字节的前两字节相同，也不会它的后两字节相同。其它几种情况原理也是如此。

利用代理区作区分

让我们再来看另一种变长方案。用所谓代理区来实现。

编号	编码(变长 2 或 4 位)	字符
1	00	h
2	01	e
...
70	69	o
71	90	w
...
80	99	d
81	7080	你
82	7081	好
...
179	7988	世
180	7989	界

高低位	80	81	...	88	89
70	你	好
71
...
78
79	世	界

这里挖出 70-89 间的码位，形成横竖 10*10 的编码空间，使得能再扩展 100 个编码空间。原来 2 位 100 个空间损失了 20 还剩 80，再加上因此而增加的 100 个空间，总共是 180 个空间。这样一种变长方式也就是 UTF-16 所采用的，具体的实现我们留待后面再详述。

好了，关于定变长的问题，就讲到这里，下一篇将继续探讨前面提及但还未深入分析的一些问题。

第四章 Unicode

原文地址: <http://my.oschina.net/goldenshaw/blog/310331>

摘要: 本文系统介绍了 Unicode 方面的一些重要知识, 如码点, 平面, 代理区, 代理对以及 UTF, 用具体的例子讲解了码点到 UTF-8 及 UTF-16 的转换原理与过程。文中还顺便鸟瞰了一下 BMP 字符集, 以此获取更加直观的印象。

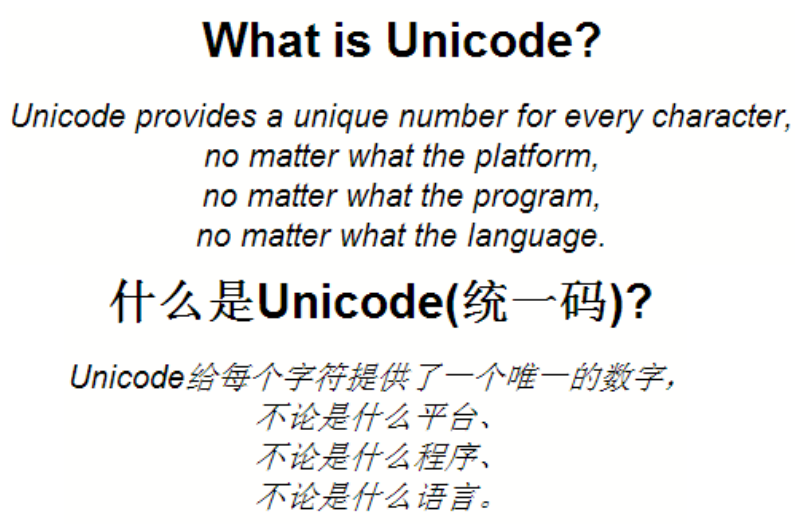
前面谈到不少的 Unicode, 但一直没有系统地谈及 Unicode 的方方面面, 所以本篇文章专门谈谈 Unicode, 当然了, Unicode 是一个庞大的主题, 这里也是拣些重要的方面谈谈而已, 免不了挂一漏万。

什么是 Unicode?

按 Unicode 官方的说法, Unicode 是 Unicode Standard (Unicode 标准) 的简写, 所以 Unicode 即是指 Unicode 标准。

按 wiki 的说法, 它是一个计算机工业标准 (a computing industry standard)。

下图来自 <http://www.unicode.org/standard/WhatIsUnicode.html> 中的截图, 在这里我把中文和英文的合在一起



这样一个所谓的一个唯一的数字在 Unicode 中就叫做码点。

Unicode 中的码点是什么?

字符集通常又叫”编码字符集”(coded charset), 这里的 coded 与”字符集编码”(charset encoding) 中的 encoding 是不同的。

- | 一个是 code, 一个是 encode, 翻译时都可以译成”编码”,
- | 但把 coded charset 译成”编号字符集”也许更不易引发误解。

码点(Code Point)即是这里的 code, 表示的是一种抽象的数字编号。UTF-X 则是最终的 encoding, 这点如不明白, 仍请参见系列第二篇。

码点的表示形式与范围是?

U+[XX]XXXX 是码点的表示形式, X 代表一个十六制数字, 可以有 4-6 位, 不足 4 位前补 0 补足 4 位, 超过则按是几位就是几位。以下是码点的一些具体示例: U+0048, U+4F60, U+1D11E。最后一个是 5 位的码点。

- | 有人可能以为码点只有 4 位, 并常常将它与 UTF-16 的编码搞混, 这些都是对码点的误解。

它的范围目前是 U+0000~U+10FFFF, 理论大小为 $10FFFF+1=11000016$ 。后一个 1 代表是 65536, 因为是 16 进制, 所以前一个 1 是后一个 1 的 16 倍, 所以总共有 $1 \times 16 + 1 = 17$ 个的 65536 的大小, 粗

略估算为 $17 \times 6 \text{ 万} = 102 \text{ 万}$ ，所以这是一个百万级别的数。

| 准确的值是 1114112，一般记为 111 万左右即可。

110000 写成二进制是 10001000000000000000，是一个 21 位的二进制数，我们知道 $2^{10} = \text{K}$ ， $2^{20} = \text{K} \times \text{K} = \text{M}$ ，即百万级别，所以 2^{21} 理论上限是两百万左右。10001000000000000000 大小基本上由第一个 1 决定，所以也就一百万左右，从这里也可印证前面的估算。

| 按照 Unicode 官方的说法，码点范围就这些了，以后也不会再扩充了。

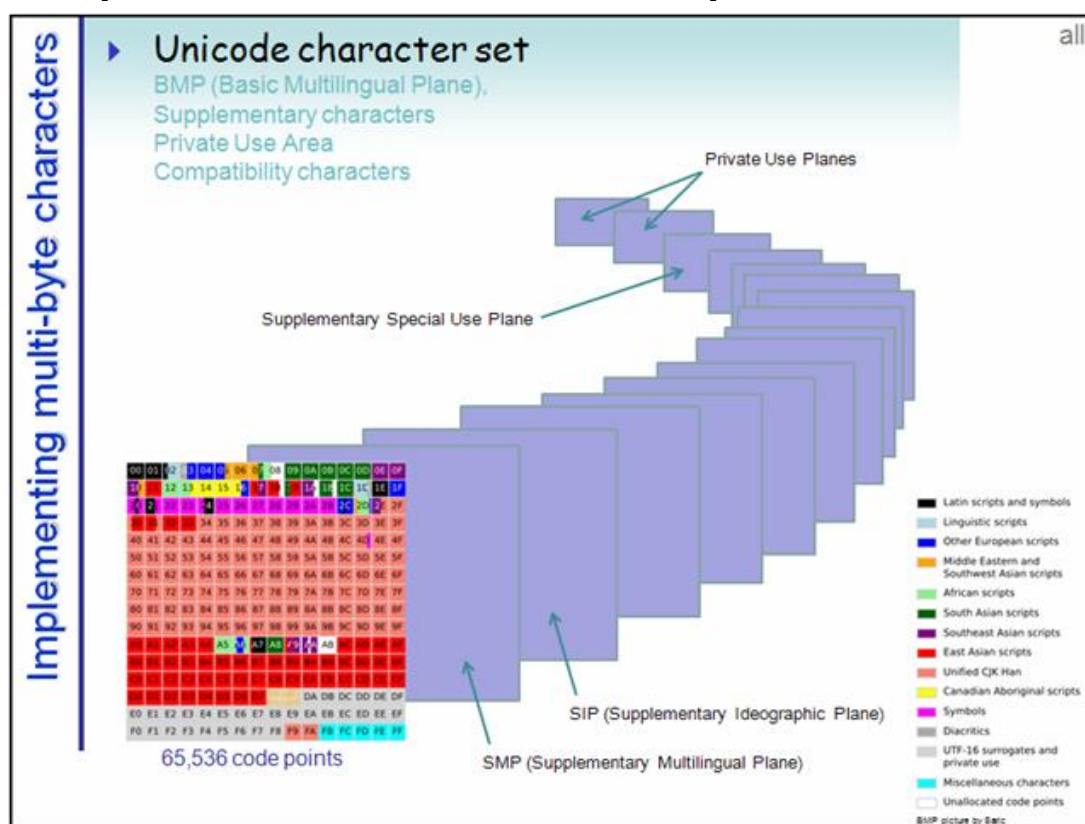
为了更好分类管理如此庞大的码点数，把每 65536 个码点作为一个平面，总共 17 个平面。

平面，BMP，SP

什么是平面？

由前面可知，码点的全部范围可以均分成 17 个 65536 大小的部分，这里的每一个部分就是一个平面 (Plane)。编号从 0 开始，第一个平面称为 Plane 0。

下图来自 <http://rishida.net/docs/unicode-tutorial/part2>



什么是 BMP？

第一个平面即是 BMP (Basic Multilingual Plane 基本多语言平面)，也叫 Plane 0，它的码点范围是 U+0000~U+FFFF。这也是我们最常用的平面，日常用到的字符绝大多数都落在这个平面内。

| 上图中第一个花花绿绿的平面就是 BMP。

UTF-16 只需要用两字节编码此平面内的字符。

| 很多人错误地把 UTF-16 当成定长两字节看待，

| 但只要处理的字符都在这一平面内，一般也不会遇到什么问题。

什么是增补平面？

后续的 16 个平面称为 SP (Supplementary Planes)。显然，这些码点已经是超过 U+FFFF 的了，所以已经超过了 16 位空间的理论上限，对于这些平面内的字符，UTF-16 采用了四字节编码。

| 注：其中很多平面还是空的，还没有分配任何字符，只是先规划了这么多。

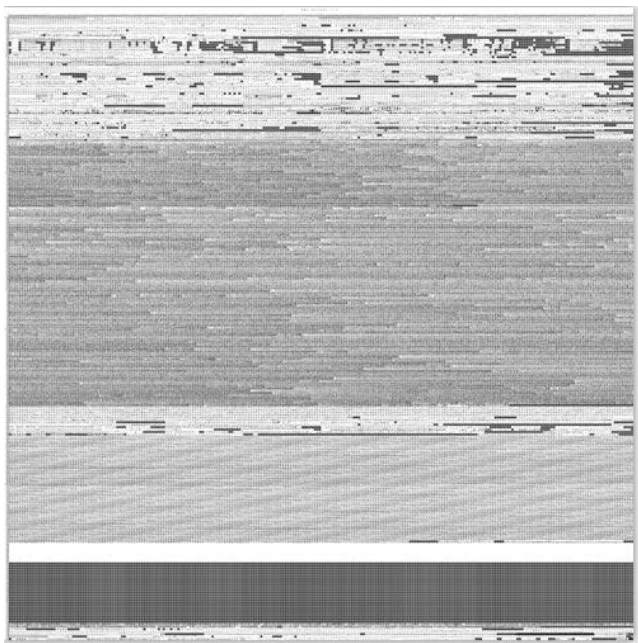
| 另：有些还属于私有的，如上图中的最后两个 Private Use Planes，在此可自定义字符。

鸟瞰 BMP 字符集

Unicode 的字符如此之多，即使是最常用的 BMP，它的码点空间也有 6 万多，如果把这些字符都放到一张图片上，会是什么情况呢？GNU Unifont 就制作了一张这样的图片。见[这里](#)。

提示：打开它需要一点时间，它的像素是 4000×4000 这个级别！

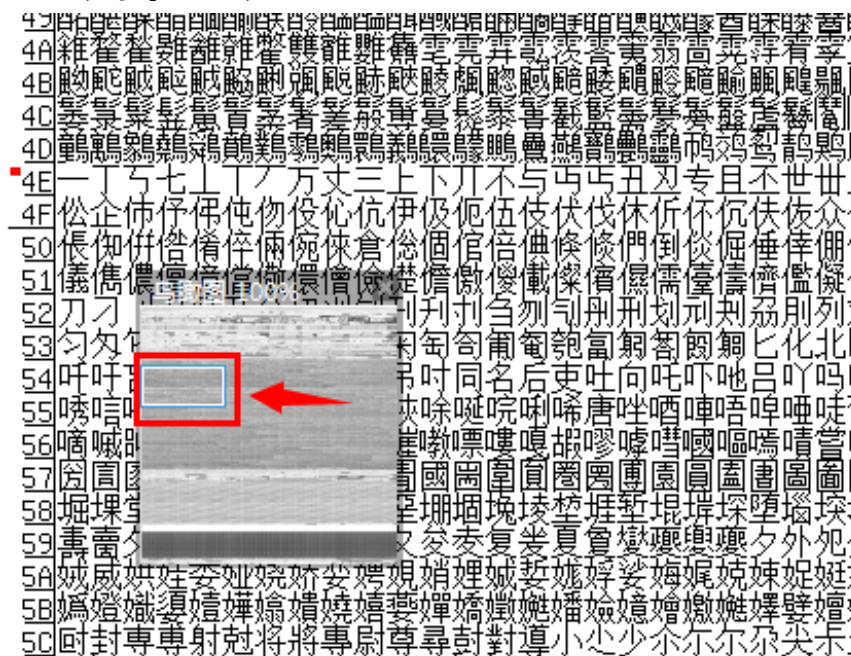
下图是它的一个缩略版本。



这是一个 256×256=65536 的表格，横向纵向都是从 00~FF。

CJK 统一汉字

你可能已经注意到上图中间一大片的区域，没错，它就是我们的汉字，在 Unicode 中，称为 CJK 统一汉字（CJK: Chinese, Japanese, and Korean，中日韩）。我们可以局部放大看一下。



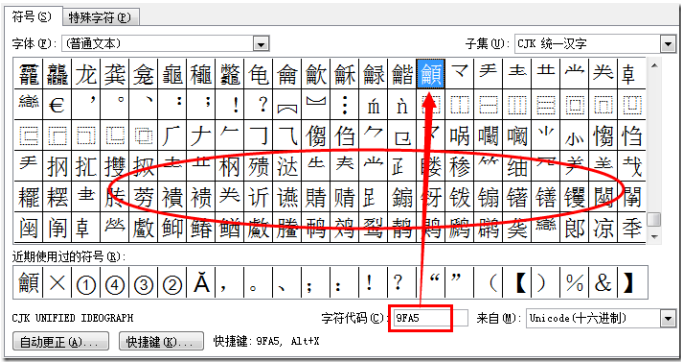
正则表达式 `[\u4E00-\u9FA5]` 来匹配中文的问题在哪？

你可能在不少地方见过这种写法，严格来说这只是 Unicode 最主要的一段中文区域。

你只要稍加计算就可知这一段大小不过是两万多一点，`\u4E00-\u9FA5`（19968–40869），中文怎么可能只有这两万多字呢？

这里的“天字第一号”字 4E00 是哪个字呢？请看上面的图，它就是“一”字，我们还可以看到它上面还有不少的汉字，这就是后来增补的汉字了。所以严格来说，这个上限是不准确的。那么它的

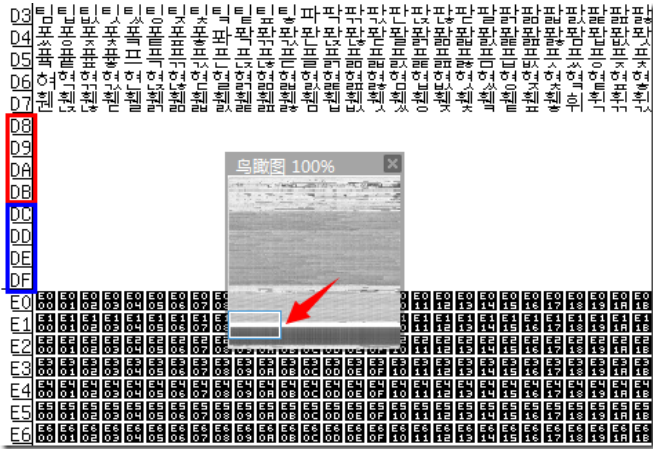
下限又是否准确呢？下面是 Word 的一个插入符号功能的一个截图



可以看到 9FA5 后面也还有不少的汉字，它们中间又还夹杂着一些符号，所以想正确地表示 Unicode 中的汉字还是个不小的挑战。

代理区

你可能还注意到前面的 BMP 缩略图中有一片空白，这白花花一片亮瞎了我们的猿眼的是啥呢？正如标题已经告诉你的，这就是所谓的代理区（Surrogate Area）了。



可以看到这段空白从 D8~DF。其中前面的红色部分 D800 - DBFF 属于高代理区（High Surrogate Area），后面的蓝色部分 DC00 - DFFF 属于低代理区（Low Surrogate Area），各自的大小均为 4×256=1024。

关于代理区的相关用途，我们在讲到 UTF-16 编码时再说。

还可以看到在它之前是韩文的区域，之后 E0 开始到 F8 的则是属于私有的（private），可以在这里定义自己专用的字符。

至此我们对 Unicode 的码点，平面都有了一定的了解，但我们还没有触及一个重要的方面，那就是码点到最终编码的转换，在 Unicode 中，这称为 UTF。

什么是 UTF？

UTF 即是 Unicode 转换格式（Unicode (or UCS) Transformation Format）

- 关于 UCS: Universal Character Set（统一字符集），也称 ISO/IEC 10646 标准，
- 不那么严格的情况下，可以认为它和”Unicode 字符集“这一概念是等价的。
- 如有兴趣的可以自行搜索了解。

码点如何转换成 UTF 的几种形式呢？我想这是大家很关心的问题，再发一次前面的一个图

H 你					字符层面
码点(Code Point)	U+0048(72)	U+4F60 (20320)	U+1D11E (119070)	抽象编码层面 (编号)	
UTF	UTF-8	UTF-16	UTF-32		
H	U+0048	00 48	00 00 00 48		
你	U+4F60	E4 BD A0	00 00 4F 60		
你	U+1D11E	F0 9D 84 9E	D8 34 DD 1E		
	变长 1-4 字节	变长 2 或 4 字节	定长 4 字节		

让我们先从最简单的 UTF-32 说起

UTF-32

我们说码点最大的 10FFFF 也就 21 位，而 UTF-32 采用的定长四字节则是 32 位，所以它表示所有的码点不但毫无压力，反而绰绰有余，所以只要把码点的表示形式以前补 0 的形式补够 32 位即可。这种表示唯一的缺点是占用空间太大。

再来看稍复杂一点的 UTF-8。

UTF-8

UTF-8 是变长的编码方案，可以有 1, 2, 3, 4 四种字节组合。在前面的定长与变长篇章我们提到 UTF-8 采用了高位保留方式来区别不同变长，如下：

0XXXXXXX

110XXXXX10XXXXXX

1110XXXX10XXXXXX10XXXXXX

11110XXX10XXXXXX10XXXXXX10XXXXXX

码点与字节如何对应？

哪些码点用哪种变长呢？可以先把码点变成二进制，看它有多少有效位（去掉前导 0）就可以确定了。

1. 一字节留给了 ASCII，码点 U+0000~U+007F（127）使用一字节。
2. 二字节有效编码位只有 5+6=11 位，最多只有 $2^{11}=2048$ 个编码空间，所以数量众多的汉字是无法容身于此的了。码点 U+0080~U+07FF（2047）使用二字节。
3. 三字节模式可看到光是保留位就达到 4+2+2=8 位，相当一字节，所以只有两字节 16 位有效编码位，它的容量实际也只有 65536。码点 U+0800~U+FFFF 使用三字节编码。我们前面说到，一些汉字字典收录的汉字达到了惊人的 10 万级别。基本上，常用的汉字都落在了这三字节的空间里，这就是我们常说的汉字在 UTF-8 里用三字节表示。当然了，这么说并不严谨，如果这 10 万的汉字都被收录进来的话，那些偏门的汉字自然只能被挤到四字节空间上去了。
4. 四字节的可以看到它的有效位是 3+6+6+6=21 位，前面说到最大的码点 10FFFF 也是 21 位，U+FFFF 以上的增补平面的字符都在这里来表示。

按照 UTF-8 的模式，它还可以扩展到 5 字节，乃至 6 字节变长，但 Unicode 说了码点就到 10FFFF，不扩充了，所以 UTF-8 最多到四字节就足够了。

码点到 UTF-8 如何转换？

那么具体是如何转换呢，其实不难，来看一个汉字”你“(U+4F60) 的转换示意，如下图所示：

U+4F60

0100 1111 0110 0000 16 位二进制形式

0100 1111 0110 0000 按 4+6+6 位分组

1110XXXX 10XXXXXX 10XXXXXX UTF-8 三字节模板

11100100 10111101 10100000 替换有效编码位

E4 BD A0 按字节重新转换成 16 进制

上图显示了一有效位为 15 位的码点到三字节转换的一个基本原理，我们还可看到原来 4F60 中的一头一尾的两个 4 和 0 在转换后还存在于最终的三字节结果中。UTF-8 三字节模式固定了 1110 的开头模式，所以多数汉字总是以 1110 开头，换成 16 进制形式，1110 就是字母 E。

| 如果看到一串的 16 进制有如下的形式 EX XX XX EX XX XX...

| 每三个字节前面都是 E 打头，那么它很可能就是一串汉字的 UTF-8 编码了。

其它变长字节转换道理也类似，其中分组从低位开始，高位如不足则补零。这里就不再示例了。

最后来看最复杂的 UTF-16，在此之前我们先要理解代理区与代理对等概念。

UTF-16

UTF-16 是一种变长的 2 或 4 字节编码模式。对于 BMP 内的字符使用 2 字节编码，其它的则使用 4 字节组成所谓的代理对来编码。

什么是代理区？

在前面的鸟瞰图中，我们看到了一片空白的区域，这就是所谓的代理区（Surrogate Area）了，代理区是 UTF-16 为了编码增补平面中的字符而保留的，总共有 2048 个位置，均分为高代理区（D800 - DBFF）和低代理区（DC00 - DFFF）两部分，各 1024，这两个区组成一个二维的表格，共有 $1024 \times 1024 = 2^{10} \times 2^{10} = 2^4 \times 2^{16} = 16 \times 65536$ ，所以它恰好可以表示增补的 16 个平面中的所有字符。

当然了，说恰好是不对的，因为代理区就是冲着表示增补平面来设计的。

UTF-16 decoder				
Lead \ Trail	DC00	DC01	...	DFFF
D800	010000	010001	...	0103FF
D801	010400	010401	...	0107FF
⋮	⋮	⋮	⋮	⋮
DBFF	10FC00	10FC01	...	10FFFF

什么是代理对？

一个高代理区（即上图中的 Lead（头），行）的加一个低代理区（即上图中的 Trail（尾），列）的编码组成一对即是一个代理对（Surrogate Pair），必须是这种先高后低的顺序，如果出现两个高，两个低，或者先低后高，都是非法的。

在图中可以看到一些转换的例子，如（D8 00 DC 00） \rightarrow U+10000，（DB FF DF FF） \rightarrow U+10FFFF

码点到 UTF-16 如何转换？

分成两部分：

1. BMP 中直接对应，无须做任何转换；

2. 增补平面 SP 中，则需要做相应的计算。其实由上图中的表也可看出，码点就是从上到下，从左到右排列过去的，所以只需做个简单的除法，拿到除数和余数即可确定行与列。

拿到一个码点，先减去 10000_{16} ，再除以 400_{16} （ $=1024_{10}$ ）就是所在行了，余数就是所在列了，再加上行与列所在的起始值，就得到了代理对了。

| $\text{Lead} = (\text{码点} - 10000_{16}) \div 400_{16} + \text{D800}$

| $\text{Trail} = (\text{码点} - 10000_{16}) \% 400_{16} + \text{DC00}$

下面以前面的 U+1D11E 具体示例了代理对的计算：

$\text{Lead} = (1\text{D}11\text{E} - 10000_{16}) \div 400_{16} + \text{DB00} = \text{D}11\text{E} \div 400_{16} + \text{D800} = 34 + \text{D800} = \text{D834}$

$\text{Trail} = (1\text{D}11\text{E} - 10000_{16}) \% 400_{16} + \text{DC00} = \text{D}11\text{E} \% 400_{16} + \text{DC00} = 11\text{E} + \text{DC00} = \text{DD1E}$

所以，码点 U+1D11E 对应的代理对即是 D834 DD1E。

关于 Unicode 的基本知识，就讲到这里。

第五章 代码单元及 length 方

原文地址: <http://my.oschina.net/goldenshaw/blog/311848>

摘要: 本文讲述了 Unicode 中的代码单元这一概念, 并以 java 为例, 阐述其对 `string.length` 方法的影响, 并结合 junit 做了一些具体的测试。

在前一篇文章中已经谈了不少 Unicode 中的重要概念, 但仍还有一些概念没有提及, 一则不想一下说太多, 二则有些概念也无法三言两语就说清楚, 本文在此准备谈一下代码单元及由此引发的一些话题。

什么是代码单元? UTF-8, UTF-16 和 UTF-32 中的 8, 16 和 32 究竟指什么?

代码单元指一种转换格式 (UTF) 中最小的一个分隔, 称为一个代码单元 (Code Unit), 因此, 一种转换格式只会包含整数个单元。

各种 UTF 编码方案下的代码单元

UTF-8 的 8 指的就是最小为 8 位一个单元, 也即一字节为一个单元, UTF-8 可以包含一个单元, 二个单元, 三个单元及四个单元, 对应即是一, 二, 三及四字节。

UTF-16 的 16 指的就是最小为 16 位一个单元, 也即二字节为一个单元, UTF-16 可以包含一个单元和二单元, 对应即是二字节和四个字节。我们操作 UTF-16 时就是以它的一个单元为基本单位的。

同理, UTF-32 以 32 位一个单元, 它只包含这一种单元就够了, 它的一单元自然也就是四字节了。

所以, 现在我们清楚了:

UTF-X 中的数字 X 就是各自代码单元的位数。

你可能要问, Unicode 整出这么多的概念来做什么? 讨论代码单元这犊子有什么用? 下面我将会以 Java 为例来做些说明, 我们首先讨论一个非常普通的方法, `string.length()`, 你可能觉得自己已经完全理解了 `length`, 不就是字符串长度吗? 可是如果深入再问下这个长度究竟怎么来的, 它跟这里的代码单元有什么关系? 你可能就未必能说清楚了, 这里面的东西可能比你想像的要复杂。

| 注: 有些读者的语言背景可能并不是 Java, 但我想这里讨论的情况对于无论是哪种语言平台它都有一定的借鉴意义。任何语言平台它去处理 Unicode 时都必然要面对类似的问题。

Java 中的 `string.length` 究竟指什么?

如果你阅读一下 java 中 `String` 类中的 `length` 方法的说明, 就会注意到以下文字:

| Returns the length of this string. The length is equal to the number of Unicode code units in the string.

| 返回字符串的长度, 这一长度等于字符串中的 Unicode 代码单元的数目。

我们知道 Java 语言里 `String` 在内存中以是 UTF-16 方式编码的, 所以长度即是 UTF-16 的代码单元数目。

BMP 内的字符长度

通过前面的篇章我们知道, UTF-16 保存 BMP 中的字符时使用了两字节, 也即一个代码单元。这就意味着, Java 中所有的 BMP 字符长度都是 1, 无论它是英文还是中文。这一点我想大家都没有疑问。代码示例如下:

```
1| @Test
2| public void testStringLength() {
3|     String str = "hello 你好";
4|     assertEquals(7, str.length());
5| }
```


| 这里我们用了 JUnit 的方式来测试, 如果你对此还不熟悉, 可以简单理解它是对使用

| main 方法来测试的一种更好替代。更多了解，百度一下，你就知道（抑或是谷歌一下，
| hope you are feeling lucky if you have no proxy(like VPN, goAgent, etc)。

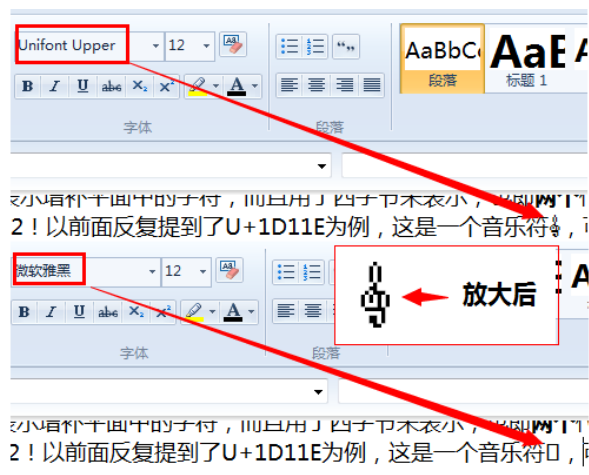
不出所料，“hello 你好”有 5 个英文字符加 2 个中文字符，所以它的长度就是 7。因此，很多人就得出了一个“结论”：“string.length 就是字符数”。但我们知道 UTF-16 同样可以表示增补平面中的字符，而且用了四字节来表示，也即两个代码单元。如果 length 的 API 说明所言不虚的话，那么，一个增补平面中的字符，它的长度将是 2！

增补平面中的字符长度

以前面反复提到的 U+1D11E 为例，这是一个五位的码点，用 UTF-8 编码需要四个字节，用 UTF-16 需要一个代理对，同样是四个字节才能表示。这是一个音乐符（MUSICAL SYMBOL G CLEF），在你的电脑中可能无法正常显示这个字符，因为没有相应的字库文件支持。可参考网站上的显示 <http://www.fileformat.info/info/unicode/char/1d11e/index.htm>。下图则是前面的一个截图：

	UTF-8	UTF-16	UTF-32
 U+1D11E	F0 9D 84 9E	D8 34 DD 1E	00 01 D1 1E

下面的图是我在 Live Writer 上的截图，可以看到如果选用开源中国 oschina 默认的“微软雅黑”字体，是无法正常显示的，显示变成了一个白板。为了显示它，我只能对这个字符特别指定“Unifont Upper”字体它才能正常显示。



| 注：Unifont Upper 字体是我特别去下载的，一般电脑上应该都没有这个字体。
| 微软有个“Arial Unicode MS”，但试了发现它还是只能支持 BMP 中的字符。
| Unifont Upper 可以到以下网址下载 <http://www.unifoundry.com/unifont.html>，是免费的，
| 做的也比较粗糙，大家看上图中放大的效果就知道了，很明显是点阵字体，而且似乎只提供
| 了一种尺寸，一放大就呈锯齿状了。不过这是 GNU 号召一些志愿者为大家提供的，
| 大家都是无偿劳动，有好过没有，这点大家应该都能理解。
| 要想更好，通常要矢量字体，肯定有商业机构提供这些字体，一切不过是钱的问题，
| 我想大家对此也是心知肚明。有没有免费又好用的，这个我就没有费心去找过了。
| 另：由于博客后台不支持增补字符，这个字符在发表后被去掉了。

让我们来看看：

```
1| @Test
2| public void testSupplementaryStringLength() {
3|     // only "one" character here
4|     String str = "??";
5|     assertThat(str.length()).isEqualTo(2);
6|     assertThat(str.length()).isEqualTo(2);
7| }
```

| 这个 Live Writer 的源码插件不能很好地支持增补字符，我拷贝代码到 Live Writer 的源码
| 插件时，发现编码已经丢失了，变成了两个??。以上代码我已经发到 git.oschina.net 上，
| 见[这里](#)。虽然很多系统及软件声称“支持 Unicode”，但在后续的讨论中，我们将始终面对
| 各种尴尬情况，为此不得不截图来说明。

这是 Live Writer 源码插件中的情况，发现编码已经丢失了，变成了两个??。

```
@Test
public void testSupplementaryStringLength() {
    // only "one" character here
    String str = "??";
    assertEquals(str.length(), 2);
}
```

而直接拷贝到 Live Writer 中，虽然没法显示，起码编码没有丢失，还是一个字符，跟前面类似

```
@Test
public void testSupplementaryStringLength() {
    // only "one" character here
    String str = "□";
    assertEquals(str.length(), 1);
}
```

如果特别调整一下这个字符的字体，它还是可以显示的，跟前面的道理一样。

```
@Test
public void testSupplementaryStringLength() {
    // only "one" character here
    String str = "𐀀";
    assertEquals(str.length(), 1);
}
```

我想，这里就是一个活生生的例子：

| 很多软件依然不能很好地支持 Unicode 中的增补字符，甚至连编码也无法正确处理。

从源码插件输出两个问号，我们可以猜测一下，当拷贝发生时，操作系统给了源码插件一段 UTF-16 编码的字节流，其中有一个增补字符是以代理对形式表示的，插件显然不能正确处理代理对，把它当作了两个字符，但单个代理对的编码都是保留的，显然没有什么字符可以对应，于是插件就用两个问号代替了。

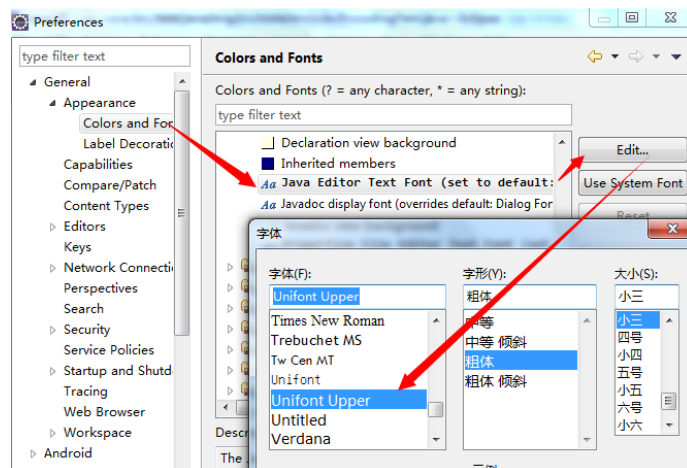
| 以上仅是一种合理猜测，因为没有深入了解 windows 中的拷贝机理及源码插件中的源代码，
| 真正的原因也许不是这样，但可以肯定的一点就是它一定在某个环节出错了。

好了，让我们直接看看代码运行的情况，没法正常显示，我们就截图来说明，虽然免不了繁琐了些。测试是通过了的，一个增补字符它的长度确实变成了 2。我给大家截个图，这是大家电脑上可能显示出来的效果：

```
@Test
public void testSupplementaryStringLength() {
    // only "one" character here
    String str = "𐀀";
    assertEquals(str.length(), 2);
}
```

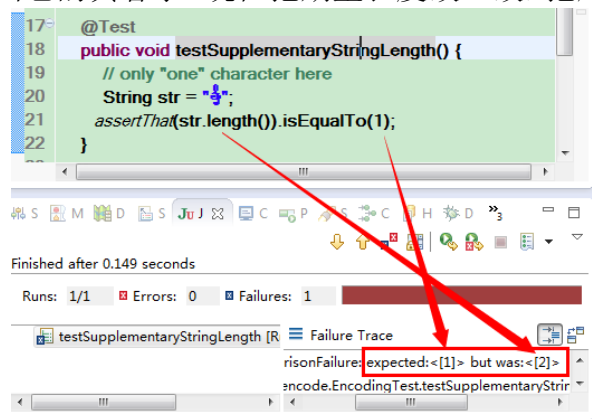
| 一个方框，里面一个问号，当字库里没有这个字时，eclipse 中就会以这样一个符号来表示，
| 这点跟 Live Writer 上显示一个白板又有所不同，但原因是一样的。

为了显示这朵奇葩，还是要设置 Unifont Upper 字体才行：



真是繁琐，其实从前面图中两个引号紧密围住那个无法显示的字符也可以看出，这里确实只有一个字符，我没骗大家，它的长度也的确变成了 2，测试也通过了，正如 API 中所说的以及我们分析的那样。

现在这一字符终于显示出它的真容了。现在把期望长度改一改，把原来的 2 设置成 1，再跑一下：



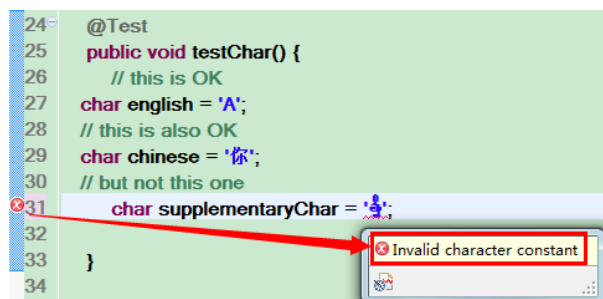
JUnit 华丽丽地报错了！它说期望的长度是 1，但实际的却是 2！

注：图中的红条可直观表示测试失败，如果出现绿条则说明测试通过，这是 JUnit 中的一个约定。

所以现在终于证实了 `string.length` 的 API 上所言不虚，图上的 `str` 只有一个字符，但它的长度却不是 1。它返回的的确就是 UTF-16 的代码单元的数目，而不是我们想像中的所谓“字符数”。

既然已经到这步，我们不妨多看几个例子。

char 类型与增补字符



在上图中，还试图把这个字符赋值给一个 `char` 变量，发现编译器提示出错。为什么呢？因为 `char` 使用了 16 位，而这个字符在 16 位内已经无法表示，所以它放不进一个 `char` 中。可以看到，`char` 可以放一个英文字符，一个中文字符，那是因为这些字符都在 BMP 中，但却无法放置这个音乐符，eclipse 的即时编译立马就报错了：“Invalid character constant”（非法的字符常量）。与此类似，如果一个中文字符来自增补平面，那么它也将无法放入 `char` 中。

另：使用了这种非等宽字体后，代码的对齐显示方面也出了问题，缩进变得不统一。

增补字符的转义表示

另外我们可以以转义的代理对的方式表示这个音乐符号，这样可以避免字库方面的问题。

| 当然了，任何字符都可以用转义的方式表示，而不仅仅限于增补字符。

| 如果你没有安装输入法，可以简单使用\u4F60来表示“你”这个字符。

从前面的篇章中，我们知道 U+1D11E，写成 UTF-16 的代理对是 (D8 34 DD 1E)。

| Java 中的转义表示始终是以\u后接四个 16 进制数字为界的(其实就是 UTF-16 的代码单元)，

| 你不能简单像码点那样写成\u1D11E，这种写法相当于“\u1D11”+”E”，

| 即前面四位 1D11 做转义，后面当成正常的字母 E。如果要转义的字符码点超过 U+FFFF，

| 我们需要两个一对的转义”\uD834\uDD1E”来表示，从这里我们也可看到，

| 所谓的转义表示其实就是 UTF-16 编码。

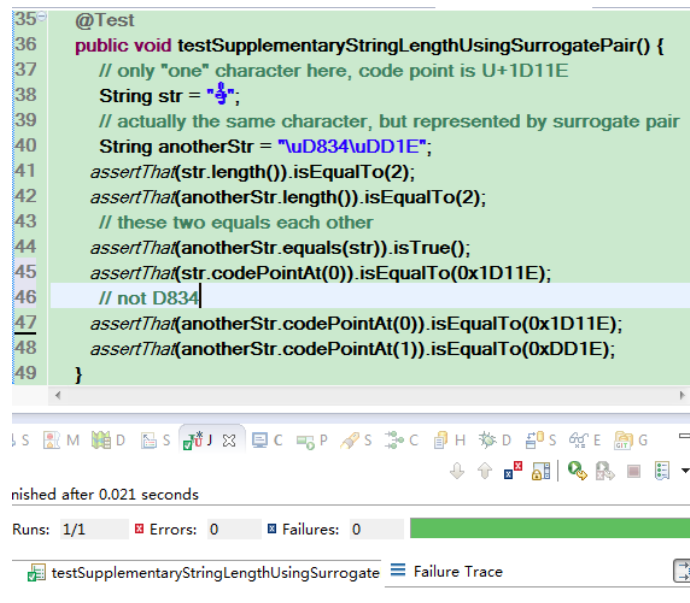
我们可以用“\uD834\uDD1E”的转义方式表示 U+1D11E 这个码点所表示的字符，只是这样的话，不明就里的人可能认为这里有两个字符了，所以输出 2 不奇怪。

| 这也正好从另一角度说明，增补字符需要四个字节，也即两个代码单元才能表示。

测试的代码如下：（注：这是后面补上的，所以跟下图有些差别，没有重新截图）

```
1| @Test
2| public void testSupplementaryStringLengthUsingSurrogatePair() {
3|     // only "one" character here, code point is U+1D11E
4|     // 这里只有一个字符 U+1D11E，由于后台不支持保存增补字符，
5|     // 所以这里的换成了两个问号，可查看 git 上的源码
6|     String str = "??";
7|     // actually the same character, but represented by surrogate pair
8|     // 同样的一个字符，以转义的代理对方式表示
9|     String anotherStr = "\uD834\uDD1E";
10|    assertEquals(str.length(), 2);
11|    assertEquals(anotherStr.length(), 2);
12|    // these two equals each other
13|    assertEquals(anotherStr.equals(str), true);
14|    assertEquals(str.codePointAt(0), 0x1D11E);
15|    // not D834
16|    assertEquals(anotherStr.codePointAt(0), 0x1D11E);
17|    assertEquals(anotherStr.codePointAt(1), 0xDD1E);
18| }
```

在这里还让两个 string 进行了 equals 比较，可以看到那条绿油油的成功指示条，测试是通过的。



另外，上图中还对两个 string 在 index=0 处的码点进行了求值（图中的 codePointAt() 方法），可以看到无论是以字符表示的 str 还是以代理对表示的 anotherStr，它们的码点都是 0x1D11E，这也从另一个侧面证明了它们是同一个字符。

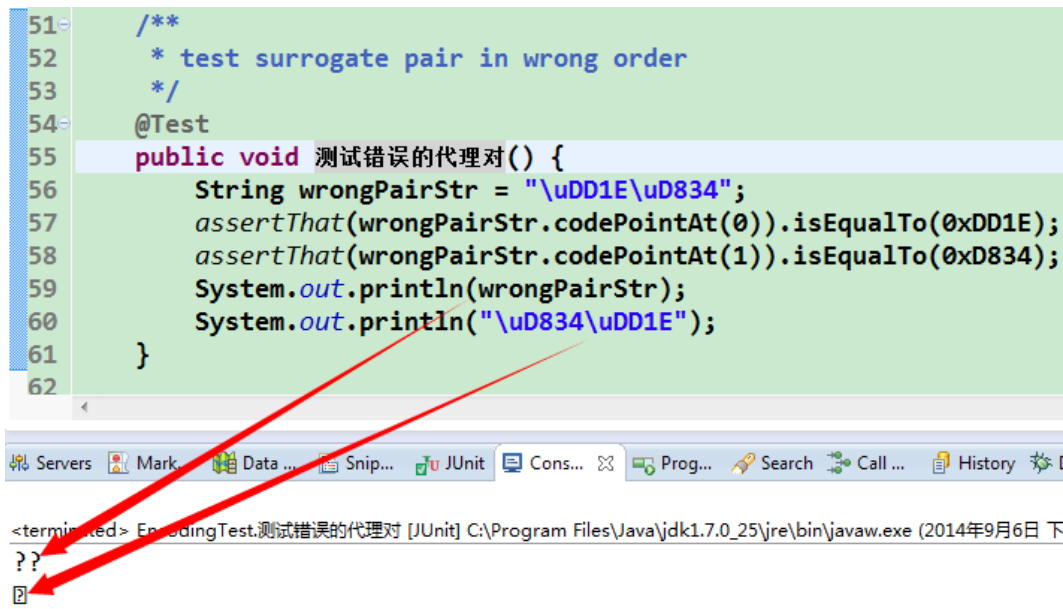
错误的代理对

如果反转了代理对，会是什么情况呢？前面篇章已经谈及，代理对必须严格按照先高后低的顺序来书写，以下代码中把代理对写反了：

```
1| @Test
2| public void 测试错误的代理对() {
3|     String wrongPairStr = "\uD834\uDD1E";
4|     assertThat(wrongPairStr.codePointAt(0)).isEqualTo(0xDD1E);
5|     assertThat(wrongPairStr.codePointAt(1)).isEqualTo(0xD834);
6|     System.out.println(wrongPairStr);
7|     System.out.println("\uD834\uDD1E");
8| }
```

| 上面的测试，方法名直接使用了中文，其实 Java 中是可以支持这种命名方式的，
| 我们也免去了写完英文方法又要写中文注释的麻烦。
| 当然，这样看上去可能让有些人觉得别扭。

让我们实际测试一下：



```
51- /**
52-  * test surrogate pair in wrong order
53-  */
54- @Test
55- public void 测试错误的代理对() {
56-     String wrongPairStr = "\uD834\uDD1E";
57-     assertThat(wrongPairStr.codePointAt(0)).isEqualTo(0xDD1E);
58-     assertThat(wrongPairStr.codePointAt(1)).isEqualTo(0xD834);
59-     System.out.println(wrongPairStr);
60-     System.out.println("\uD834\uDD1E");
61- }
62-
```

<terminated> EncodingTest.测试错误的代理对 [JUnit] C:\Program Files\Java\jdk1.7.0_25\jre\bin\javaw.exe (2014年9月6日 下
??

可以看到，输出了两个问号，在 index=0 处的码点也变成了 0xDD1E，而不是原来的 0x1D11E 了，而正常顺序则只输出一个字符。再一次的由于字库原因，它不能正常显示，我也懒得去调整 console 的字体了，大家明白怎么回事就行了。

结论

以上说了这么多，不外乎就是为了证明一件事，java 中 string.length() 不是你想像中的那样。在前面系列中的第三篇中，我们就已经谈到变长的引入也同时带来了复杂性，其中就说到了它影响到我们对 java 中的 string.length 的理解，现在 length 显得有点尴尬，如果我们真的想确切地知道有几个字符，length 显然是不能给出正确答案的。

目前来说，增补字符使用得还比较少，多数处理的还是 BMP 中的字符，所以哪怕你没有认识清楚它的本质，通常也不会给你带来什么麻烦。但事情是在不断发展的，也许不久的将来，处理这些增补字符的问题就会成为常态。

好了，关于代码单元及 string.length 方法就分析到这里。String 中的许多方法跟编码问题紧密相关，在下一篇中，我们还将分析一下 string.getBytes() 方法并初步探讨一下乱码的问题。

第六章 getBytes 方法及乱码

原文地址: <http://my.oschina.net/goldenshaw/blog/313077>

摘要: 本文主要讲述 `string.getBytes()` 方法, 分析了系统缺省编码的各种陷阱, 并针对测试中出现的乱码作了初步的分析, 对代码页的概念也进行了介绍

在前一篇里我们谈了 Unicode 的代码单元及 `string.length`, 现在接着前面的讨论继续谈 `string.getBytes()` 方法并对乱码的产生作初步分析。

string.getBytes 方法

首先声明一下, 以下讨论如无特别说明, 均是在 Java 语言环境下。如果你用的不是 java, 我只能说声抱歉。但另一方面, 我相信无论是何种语言或平台, 也必然有类似的方法及类似的处理, 而其中的原理也必将是相通的, 当然了, 具体到细节上则可能有些差异。

带参数的调用

首先, `string.getBytes` 它可以带参数去调用, 这是最简单的情形, 如下

```
1| @Test
2| public void testGetBytesGbk() throws UnsupportedEncodingException {
3|     String str = "hello 你好";
4|     assertThat(str.getBytes("GBK").length).isEqualTo(9);
5| }
```

因为 GBK 是变长编码, 对 ASCII 字符采用一字节, 汉字则是两字节, 所以总的长度是 $1*5+2*2=5+4=9$, 所以测试是通过的。

| 注: 本文代码均已经上传到开源中国 oschina 的 git.oschina.net 上, 具体代码见[这里](#)
| 有些代码后来又做了修改, 与下面截图中的一些可能有差异。

无参数的调用

此外, `string.getBytes` 它又可以不带参数去调用, 这是最容易引发误解的, 也是乱码的一大根源。如下面的代码所示, 那么这表示什么呢?

```
1| @Test
2| public void testDefaultGetByte() {
3|     String str = "hello 你好";
4|     assertThat(str.getBytes().length).isEqualTo(9);
5| }
```

有人可能会想, 既然 String 在内存中是以 UTF-16 编码, 是不是指它用 UTF-16 编码时所用的字节呢? 答案是否定的。可能有人已经知道这个问题怎么回事, 他们会说, 没有参数时就使用系统的缺省编码。可是等等, 这里所谓“系统”究竟指什么? 操作系统? 如果你就是这么认为的话, 你可能又错了。

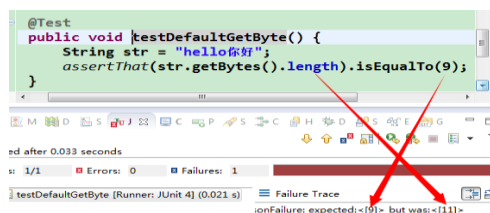
所谓的缺省编码

缺省的编码究竟是哪种? 有句话说得好:

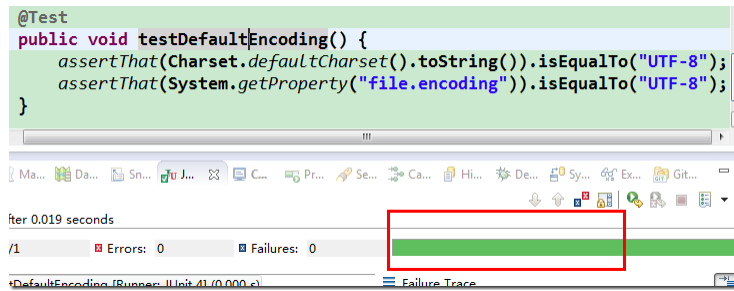
| “是骡子是马, 拉出来溜溜就知道了”

Eclipse 下的缺省编码

“hello 你好”这一串字符, 如果按变长的 GBK 来编码, 英文一个字节, 中文则是两个字节, 所以总的字节长度为 $1*5+2*2=5+4=9$, 让我们简单实验一下: (以下测试如无特殊说明均在 Windows 平台下完成, 我的操作系统是 64 位 win7):



咦？居然测试失败了，红条现身了。怎么回事？Windows 系统缺省不是 GBK 吗？而且它所那个实际值 11，则极大地暗示了使用了 UTF-8 作为缺省，我们知道 BMP 中，一个汉字是三字节，所以 $1*5+3*2=5+6=11$ 。让我们用测试来证实一下：



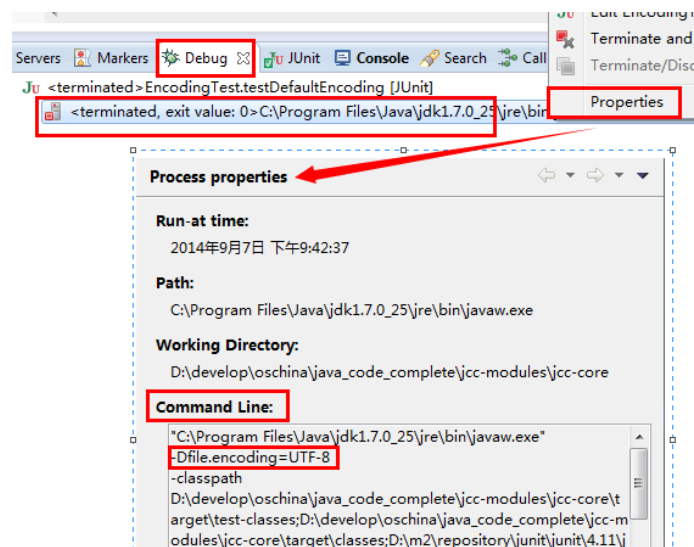
果不其然，绿条显示测试通过，是 UTF-8，怎么回事呢？还是要再次声明一下：

- | 作者一直在 windows 下使用 Live Writer 写着博客，我也有虚拟机，
- | 上面也装了 linux 的 Ubuntu，可是并没有开启，更没在上面作测试，
- | 一切测试都是在 windows 下做的。我向毛主席发誓！！

让我们用调试模式再跑一下此方法，以此获取 eclipse 运行此方法时的一些细节，在此不用设置断点。

- | 在 eclipse 中，可以按下“Ctrl+Shift+向上（下）箭头”快速跳到方法名中，
- | 然后按下“Alt+Shift+D T”快速地以 debug 方式跑一下，
- | “Alt+Shift+D T”表示按下“Alt+Shift+D”后紧接着再按“T”，
- | 是一种更加复杂的快捷键组合方式。
- | 当然了，你也可以鼠标选中方法名—右键—Debug As—JUnit Test。

然后在 Debug 视图中，选中运行的实例—右键—选择“properties”，在弹出的窗口中，我们终于发现了猫腻：



可以看到在 Command Line 中，eclipse 传入了一个额外的参数“-Dfile.encoding=UTF-8”，我们可以大胆猜测一下正是这一参数改变了 string.getBytes 的缺省值！

- | 注：其它平台下是什么情况，我不敢断言。
- | 实际上，eclipse 之前的一些版本是否也是如此，我也说不准，
- | 我目前用的是 win7 下的 64 位 eclipse kepler SR1

命令行中的缺省编码

让我们跳过 eclipse，直接在命令行中验证一下，上图中 eclipse 正好为我们列出了正常运行所需要的一系列 classpath，我们直接拷贝来用。

- | 要是没有 eclipse 生成这一堆 classpath，我才不想去命令行下演示呢，敲这些玩意简直让人抓狂。我们还可以看到 eclipse 中并没有使用“java”这个命令来启动 JVM，它直接就用了 javaw.exe，所以也许你是否设置了 JAVA_HOME 对 eclipse 并没有什么影响。

执行的命令如下：

```
1| java
2| -classpath
3| D:\develop\oschina\java_code_complete\jcc-modules\jcc-core\target\test-
classes;D:\develop\oschina\java_code_complete\jcc-modules\jcc-core\target\classes;D:\m2\repository\junit\junit\4.11\junit-4.11.jar;D:\m2\repository\org\hamcrest\hamcrest-core\1.3\hamcrest-core-1.3.jar;D:\m2\repository\org\assertj\assertj-core\1.5.0\assertj-core-1.5.0.jar;
4| org.junit.runner.JUnitCore
5| org.jcc.core.encode.EncodingTest
```

注：以上列出的 classpath 仅对本机适用，熟悉 maven 的同学可能已经看出 classpath 里的第一项就是源码文件夹（source folder）“test”下的类编译后缺省放置的位置，第二项则是源码文件夹“src”下的类编译后放置的位置，其实对这个例子而言这里并不需要这个，因为这是个纯粹为测试而写的测试类，并没有引用 src 下的任何类。其它的则是用到的 jar 了。还有我把 maven 的缺省库设置在了 D:\m2\repository 下。大家如有兴趣在本地亲自实验，则可按照上图方式拿到 eclipse 正常运行的 Command Line 中的 classpath（它还包含了跟 eclipse 运行有关的一些 jar，在命令行运行时可把那些去掉）。

另：git 上的项目使用 maven 来构建的，如果对此不熟悉，请自行搜索了解。

以上的命令看上去有些乱，把 classpath 去掉的话，就简单一些了：

```
1| java org.junit.runner.JUnitCore org.jcc.core.encode.EncodingTest
```

进一步去掉包名则是

```
2| java JUnitCore EncodingTest
```

就两个参数而已，第一个参数 JUnitCore 类就是有 main 方法要执行的类；第二个参数就是我们的测试类 EncodingTest 了，作为参数传递给 JUnitCore。

这里是作为 string 参数传递进去的，我们可以推测，JUnit 里面的实现自然会用到反射（reflection）之类的技术，另外，测试类中使用了注解（annotation），没有继承任何类，所以可以肯定这一点。

如果你对 JUnit 不太熟悉，甚至用久了 IDE，对命令行已经很陌生，也可自行写个简单的带 main 方法的类来测试。总之，达到一切传入参数由我们掌控的目的即可。

另：如果在命令行下用 junit 来测试，我们无法像在 eclipse 中那样特别指定只测试其中的一个方法，这里对 EncodingTest 类中的所有的方法都进行了测试。

下面是执行的结果，可以看到这下缺省确实是 GBK 了，所以测试失败了：

```
C:\Users\Administrator>java -classpath D:\develop\oschina\java_code_comp
pository\junit\junit\4.11\junit-4.11.jar;D:\m2\repository\org\hamcrest\h
JUnitCore org.jcc.core.encode.EncodingTest
JUnit version 4.11
..??
?
...E..
Time: 0.032
There was 1 failure:
1) testDefaultEncoding(org.jcc.core.encode.EncodingTest)
org.junit.ComparisonFailure: expected:<' [UTF-8]'> but was <' [GBK]'>
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Native
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unkn
```

这里我使用了绿色背景，所以看上去跟传统的黑色背景有些差别

让我们也加上 -Dfile.encoding=UTF-8 再跑一下，果然，最后一行的“OK”表示测试通过了：

```
C:\Users\Administrator>java -Dfile.encoding=UTF-8 -classpath D:\de
target\classes;D:\m2\repository\junit\junit\4.11\junit-4.11.jar;D:\
jar; org.junit.runner.JUnitCore org.jcc.core.encode.EncodingTest
JUnit version 4.11
..??
恒割
....
Time: 0.026
OK (7 tests)
```

图上还用红框圈出两个乱码字符，这点在下面再分析

那么现在一切已经清楚了：

- `string.getBytes` 在没有指定参数的时候，它使用了 JVM 的缺省编码，如果启动 JVM 时没有明确设置编码，那么 JVM 就会使用所在操作系统的缺省编码；但如果启动时明确地设置了编码，那么这一设置将成为 JVM 中的缺省编码！

所以呢，这里的坑还是有些多的，而且坑里的水又比较深的。如果你走路时是那种喜欢仰望星空的哲学家式的人，你一定要会游泳才行呀！

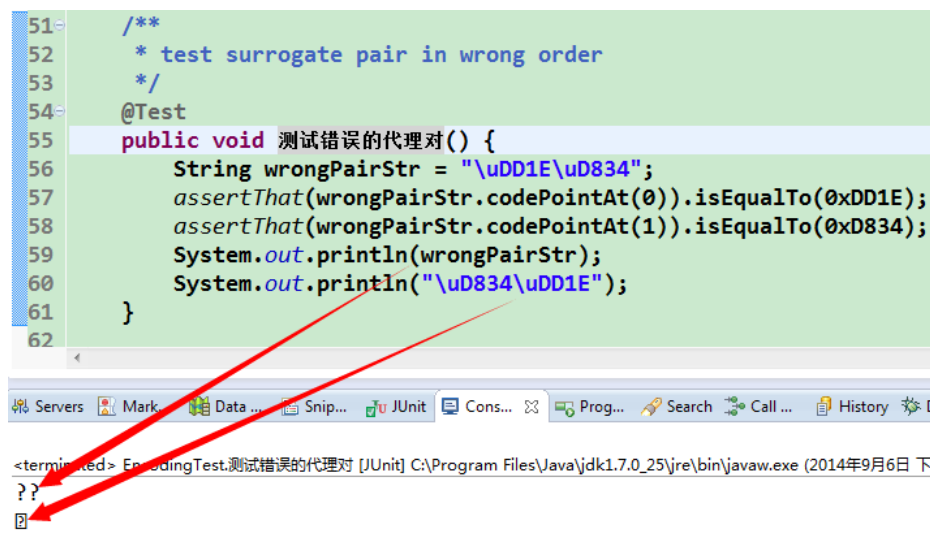
| 至于其它的平台，具体是怎么样的，是否与 java 一样存在不少的“坑”，这个无法一概而论，
| 读者可根据所在平台的具体情况作具体分析。

乱码的初步分析

在前面的最后一张截图中可以看到，出现了两个乱码的字符“亘割”，既来之，我们干脆就见招拆招，分析分析之。我们初步猜测是，当我们设置了 `-Dfile.encoding=UTF-8` 这一参数后，打印流也变成了 UTF-8 来编码，而命令行窗口依然按照 GBK 来解码传递过来的字节流，所以就出现乱码了。

问题回顾

让我们综合来看一下，首先，输出的问号及乱码是前面有一个方法里有打印语句导致的，在那里打印了一个错误的代理对及一个正确的代理对，在前面篇章也曾提及，图如下

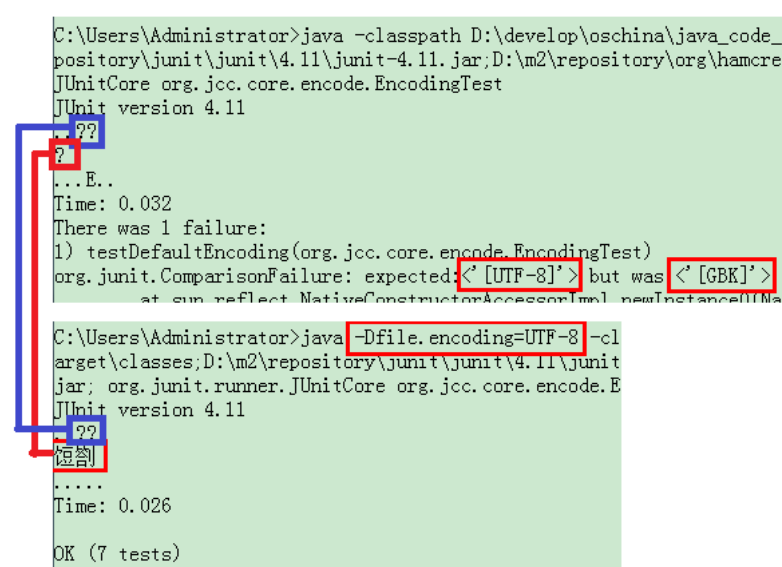


```
51- /**
52-  * test surrogate pair in wrong order
53-  */
54- @Test
55- public void 测试错误的代理对() {
56-     String wrongPairStr = "\uDD1E\uD834";
57-     assertThat(wrongPairStr.codePointAt(0)).isEqualTo(0xDD1E);
58-     assertThat(wrongPairStr.codePointAt(1)).isEqualTo(0xD834);
59-     System.out.println(wrongPairStr);
60-     System.out.println("\uD834\uDD1E");
61- }
62-
```

<terminated> EncodingTest.测试错误的代理对 [JUnit] C:\Program Files\Java\jdk1.7.0_25\jre\bin\javaw.exe (2014年9月6日 下
??

| 当然了，JUnit 是不赞成你使用打印语句的，JUnit 强调自动化测试，所以一切判断应该由
| `assert` 之类的语句去完成，而不应该打印出来，然后靠人眼去看去判断。
| 在下图中，我们能看到，JUnit 在测试成功一个方法后，会输出一个点 (.)，
| 而在失败时则会输出一个 E，而我们的打印流夹杂在其中，打乱了它的输出。

我们再来对比一下两次执行的细节：



```
C:\Users\Administrator>java -classpath D:\develop\oschina\java_code_...
JUnitCore org.jcc.core.encode.EncodingTest
JUnit version 4.11
...??
...E..
Time: 0.032
There was 1 failure:
1) testDefaultEncoding(org.jcc.core.encode.EncodingTest)
org.junit.ComparisonFailure: expected:<' [UTF-8]'> but was <' [GBK]'>
...
C:\Users\Administrator>java -Dfile.encoding=UTF-8 -cl
arget\classes;D:\m2\repository\junit\junit\4.11\junit
jar; org.junit.runner.JUnitCore org.jcc.core.encode.E
JUnit version 4.11
...
Time: 0.026
OK (7 tests)
```

首先无论是 GBK 还是 UTF-8，前面那个错误代理对的打印都输出了两个??，表明都没有找到相应的字符。（图中蓝色部分）

但我们感兴趣的是第二个打印（图中左边红色部分），它以代理对方式实际打印的是那个 U+1D11E 的音乐符，可以看到，在第一个窗口中，还是只有一个问号，可是在第二次我们加入“-Dfile.encoding=UTF-8”后，输出了两个奇怪的字符“𠂔𠂔”，我们自然要问，为什么乱码了？更进一步的，为什么是这两个字呢？

在业余的时间，我喜欢看一些记录片，《重返危机现场》(Seconds from Disaster) 是我喜欢的一个系列，由国家地理频道 (National Geographic Channel) 拍摄，片中对各类事故，如空难，列车出轨，航天飞机爆炸等的发生原因作了精彩而深刻的调查与分析，片头经常出现的一句名言就是：“Disasters don't just happen.”（灾难不会凭空发生），与此类似，乱码也不是无缘无故的，当然了，我们的问题与那些比起来就是小巫见大巫了。
另：如果对空难有特别的兴趣，《空中浩劫》也是相当精彩的一个系列。

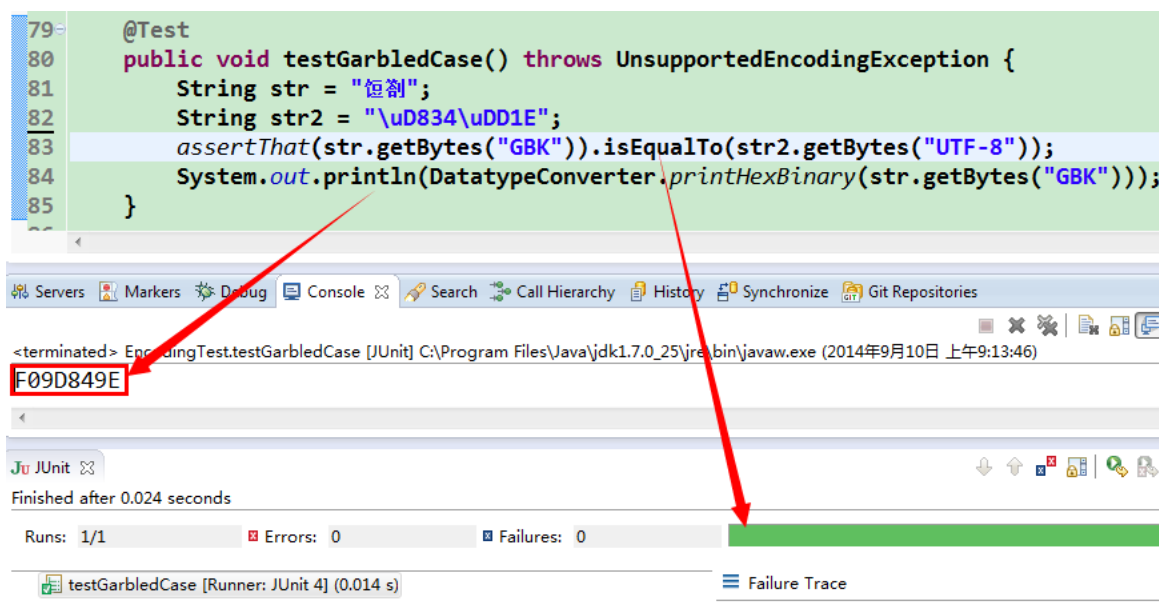
猜想与验证

让我们干脆做个深度历险，把这两个怪怪的字“𠂔𠂔”拷贝到程序中去，如下：

```
1  @Test
2|  public void testGarbledCase() throws UnsupportedOperationException {
3|  String str = "𠂔𠂔";
4|  String str2 = "\uD834\uDD1E";
5|  assertThat(str.getBytes("GBK")).isEqualTo(str2.getBytes("UTF-8"));
6|  System.out.println(DatatypeConverter.printHexBinary(str.getBytes("GBK")));
7|}
```

我敢说没几个人知道如何念这两个字，你们的语文水平也就这样了。
你问我会不会念呀？这个。。怎么说呢，今天天气还不错！其实我也不会啦~

我们猜测它是命令行窗口错误地以 GBK 编码方式去解码一段 UTF-8 的字节流导致的，让我们用测试来验证一下，并获取它的 GBK 编码看看：



可以看到，测试是通过的，我们还打印了 GBK 的字节输出，发现是 F0 9D 84 9E，你是否觉得有点眼熟呢？再次看看前面发过的图：

	UTF-8	UTF-16	UTF-32
U+1D11E	F0 9D 84 9E	D8 34 DD 1E	00 01 D1 1E

其实从测试通过我们就知道，这两个字节数组必然是相等的。那么现在我们也大概能明白这个乱码是怎么一回事了，在此之前我们再说一个概念——代码页。

代码页 (Code Page)

其实这也是处理字符集编码问题时经常遇到的一个概念了，虽然前面一直没怎么提到它，不过这里也不打算多么详细地去讲它：

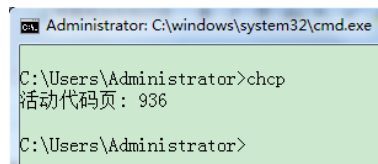
- | 不那么严格地去看，代码页可以看作是字符集编码的同义词，
- | 比如 Code Page 936 就相当于 GBK，而 Code Page 65001 则相当于 UTF-8。

可以通过在命令行窗口中输入“chcp”来查看当前代码页

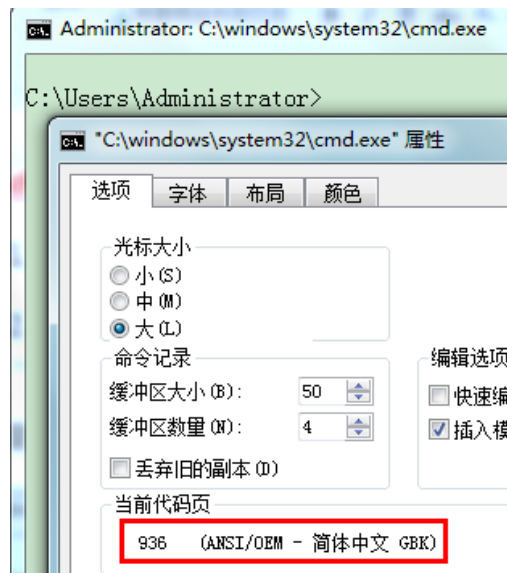
1| chcp=change code page (改变代码页)

- 1、要是不带参数就是输出当前的代码页。
- 2、带参数则另起一个 console，并把此新开的 console 的代码页设置为指定的值。
 - | 注：这一功能在我的电脑上执行时貌似有点问题，有时会开一个新的窗口，
 - | 但窗口与字体都变得很小；有时又没开新窗口。

以下是查看当前活动代码页的一个截图：



还可以在标题栏--右键--属性--选项中查看，如下，可以看到 936 就是 GBK



Code Page 936 就是命令行窗口的缺省值，也即它缺省将使用 GBK 来解码它收到的字节流。

乱码的机理

现在是时候仔细分析一下这次乱码的产生机理了：

- 1、我们在代码中打印了一个代理对，即 U+1D11E 这个码点所代表的一个音乐符，在 JVM 的内存中就是以 UTF-16 的代理对编码形式存在的，可以想像在堆内存中有这么一个字节数组，它的值是 (D8 34 DD 1E)。
- 2、我们在启动 JVM 时加入了“-Dfile.encoding=UTF-8”参数，所以缺省编码就成了 UTF-8。
- 3、当打印发生时，会以缺省编码形式得到向外输出的字节流（字节数组），也即内部某处实质调用了 string.getBytes(“UTF-8”)，这样就得到了一个临时的字节数组 (F0 9D 84 9E)，其实就是 UTF-8 对 U+1D11E 的编码，JVM 向命令行窗口输出这样一个字节数组，自然是希望在命令行中打印出一个音乐符来。
- 4、可是，命令行只是得到这么一串字节流 (F0 9D 84 9E)，这里不包含任何的编码信息，所以它还是愣头愣脑按着自己的缺省 GBK 来解码，它先拿到第一个字节 F0 (11110000)，一看最高位是 1，所以它认为这是一个汉字编码的第一个字节，于是它继续地读入第二个字节 9D，并把 (F0 9D) 合一起去查 GBK 的码表，这一查还真查到一个字，就是“短”了（我们觉得这像是一个乱码，可计算机知道什么呢？），所以它很高兴地向外输出了这么一个字符。至于后面的 (84 9E) 呢，道理是一样

的，所以又输出了另一个字符“劓”。

| 其实通过前面的测试我们就知道了，“劓劓”用 GBK 编码后的字节数组恰恰与 U+1D11E
| 这个码点对应的音乐符以 UTF-8 编码后的字节数组相同，所以这就是故事的全部。

尽管我们以后要对付的乱码问题千差万别，但很多的问题其背后的基本原理与以上的例子没有本质区别。

string.getBytes 的本质

在此也正好先说说 string.getBytes 的本质：

string.getBytes 不过是把一种编码的字节数组转换成另一种编码的字节数组。

- 这里的一种编码在 Java 中就是 UTF-16，这个已经定了，你不用操心，你也改不了！
- 这里的另一种编码则由你来指定，不指定就用缺省，反正得要有，没有还转个球！

所以呢，string.getBytes 其实就是 bytes.getBytes，不过是一堆的 bytes 变来变去。

| 在 Java 中呢，前面的 bytes 其实是限死了的，就是 bytesInUTF16.getBytes (XXX)。
| 怎么说呢，严格地讲，应该是 codeUnitsInUTF16.getBytes (XXX)，
| 但另一方面，code unit 底层也就是两个 bytes，所以你只要指定后面一个参数，
| 即你要把一串已经是 UTF-16 编码的 bytes 变成哪种编码的 bytes。

那么转换的依据又是什么呢？自然就是 bytes 背后都要表示的是相同的抽象字符了。

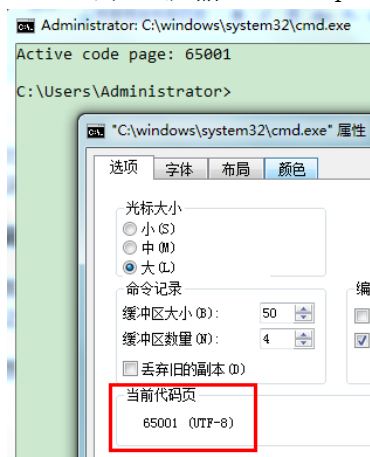
| 比如有一串字节数组表示的是“hello 你好”这 7 个字符，转换成另一种编码的字节数组后，
| 在那种编码中，它所表示的也必须是“hello 你好”这 7 个字符。
| 具体的转换细节，我们在以后的篇章中再详细分析。

让解码与编码一致

既然前面说到，由于命令行窗口采用了 GBK 来解码 UTF-8 的字节流，从而导致了乱码，自然，我们就想，如果把命令行窗口也设置成 UTF-8 编码，事情不就 OK 了吗？让我们来试试。

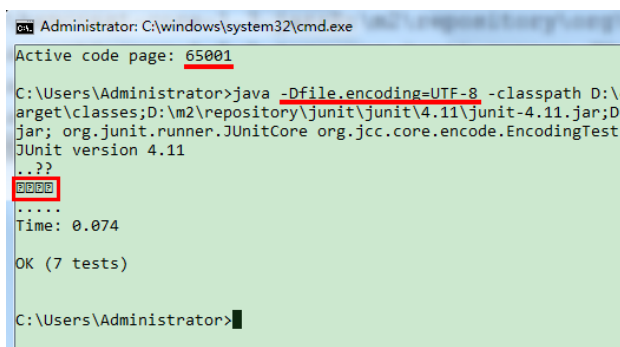
在 CMD 下验证

前面说了，代码页 65001 就是 UTF-8，那么就输入“chcp 65001”，回车，结果如下：



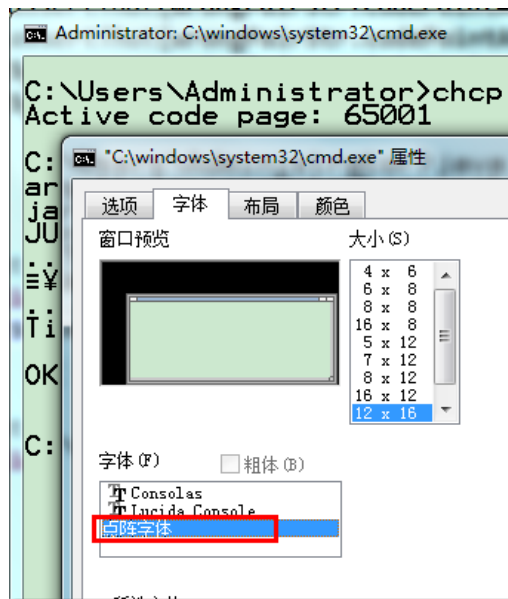
| 为了少截一些图片，图中同时把标题属性窗口也开了

可以看到“Active Code Page: 65001”的字样，同时标题属性窗口也证实了目前是 UTF-8 编码。再次执行前面的命令：

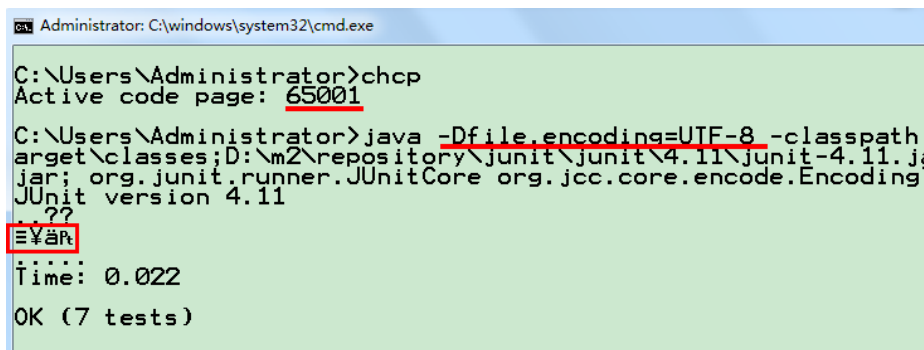


可是情况并不如我们想像那样，可以看到出来四个问号，按理应该只出来一个字符（哪怕不能显示）。

更糟糕的是，如果我们换种字体，输出应该不会受此影响，但事实证明不是如此！下图中把字体从原来的“Consolas”换成了“点阵字体”



换完后的输出结果，变成了几个奇怪的字符！



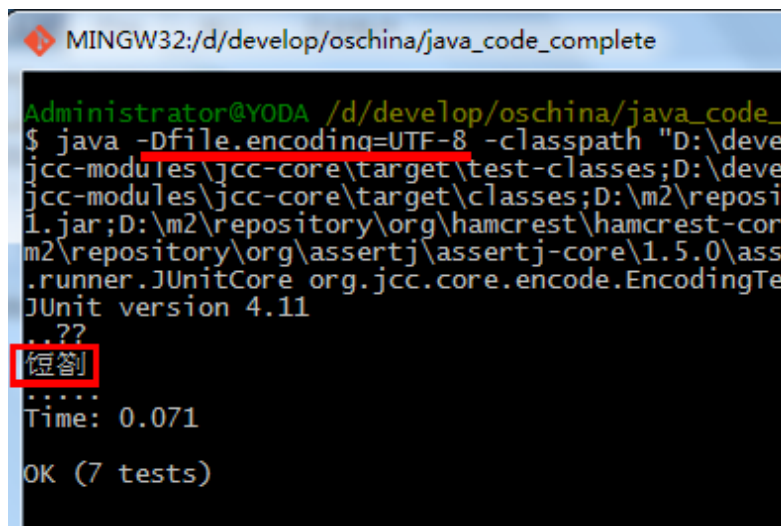
结果完全无法理喻，可能是有 bug，看来在 windows 的命令行窗口下是无法验证这点了。

- | 由此也可看出，乱码真是挺麻烦的一件事，有时问题还不是出自于你，
- | 在这里不打算继续深挖下去了，怕没完没了。

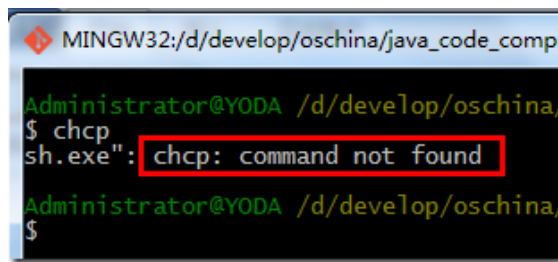
让我们转战其它地方试试。

在 git bash 上验证

首先想到 git bash，让我们看看：（注：这里要对 classpath 里面的内容用双引号括住，因为里面有分号对 git bash 有影响）



老问题，看来它也是用了 GBK，转成 UTF-8 看看：



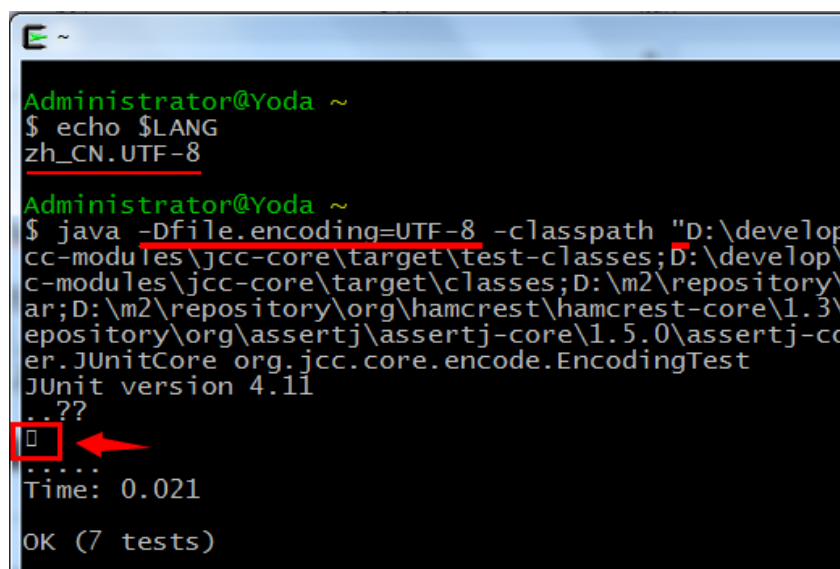
```
MINGW32:/d/develop/oschina/java_code_comp
Administrator@YODA /d/develop/oschina,
$ chcp
sh.exe": chcp: command not found
Administrator@YODA /d/develop/oschina,
$
```

悲剧，不支持这个命令。又不清楚如何调整它的编码，囧，只好作罢。可机子上还装有 cygwin，再一次转战。

有句话是怎么说来着？不要吊死在一棵树上，多找几棵树试试~

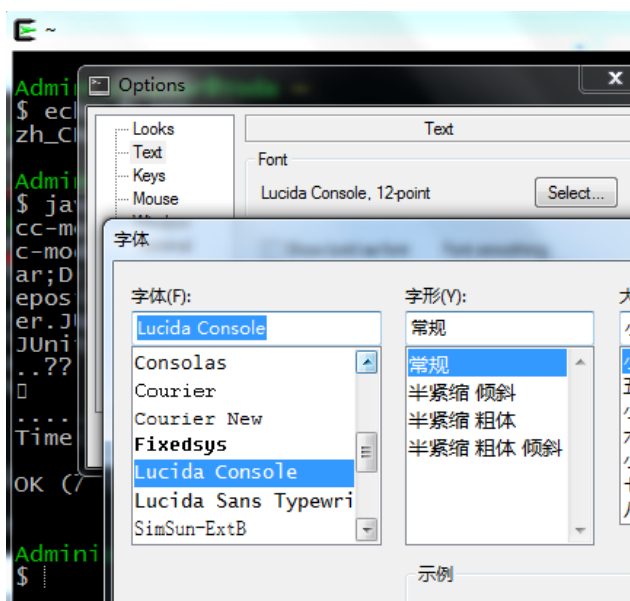
在 cygwin 上验证

输出 \$LANG 时可看到，它缺省已经是 UTF-8（窃喜，正愁不知如何调整呢~），直接上命令（注：这里同样要把 classpath 里面的内容用双引号括住，因为它也是模拟 linux 的 console，所以不括住也会受里面分号的影响）。



```
Administrator@Yoda ~
$ echo $LANG
zh_CN.UTF-8
Administrator@Yoda ~
$ java -Dfile.encoding=UTF-8 -classpath "D:\develop\cc-modules\jcc-core\target\test-classes;D:\develop\cc-modules\jcc-core\target\classes;D:\m2\repository\ar;D:\m2\repository\org\hamcrest\hamcrest-core\1.3\repository\org\assertj\assertj-core\1.5.0\assertj-core.JUnitCore org.jcc.core.encode.EncodingTest
JUnit version 4.11
..??
Time: 0.021
OK (7 tests)
```

这次终于算是正常了，可看到只有一个字符，不过由于字库不支持增补字符的原因而无法显示，调整字体试试？



虽然这里列出了不少字体，至少比命令行窗口下要多得多了，但还是没有我后来下载的支持增补字符的字体，Word 等软件里能列出的很多字体这里也没有，看来对 console 下能用什么字体还是有一些限制的，所以在 console 下显示增补字符这个希望也只能落空了。

非 Windows 平台，linux，mac...

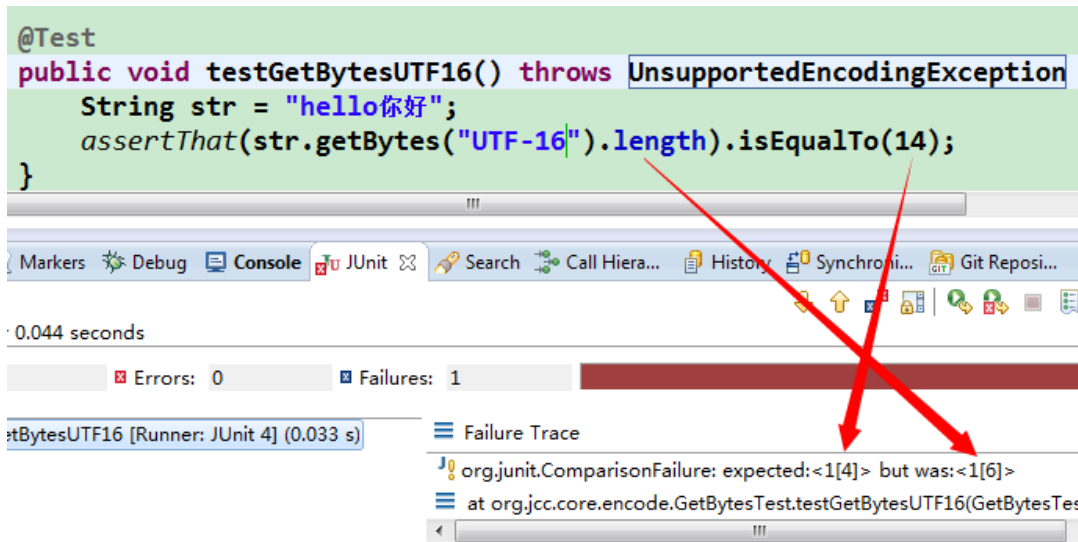
我在这里就不演示了（其实我也不会~），你要是不知道如何去整？

丫的既然已经玩上了高大上的如 Linux，怎么还会搞不掂这些问题呢！你要是说恰巧知识有些盲点，那么俺也不懂，自己问度娘，谷歌，搜叔，必姨，36 娘，雅虎等去，这些亲戚都很愿意回答你的任何问题，你要是还搞不掂，连俺都要鄙视你了：“就这水平还敢玩 Linux，不装逼装 Windows 会死吗？”哥也在用 Windows，哥表示不丢人，用得还挺舒畅。

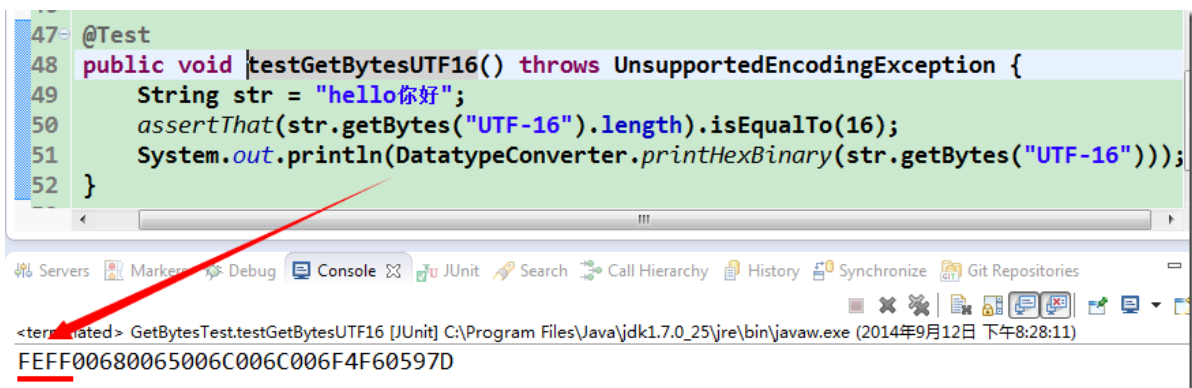
- | 不知道在开源社区说这些会不会招来怨恨？
- | 不过经常我们用的 Windows 倒是挺符合开源里的免费精神，哈哈，你们都懂的~
- | 我倒是听说开源里的那个 free，更加强调的是“自由”而不是免费，呵呵
- | 记得 Linus Torvalds 好像说过，软件就像那个啥，sex？，然后呢，free 更好。
- | Software is like sex: it's better when it's free.
- | 也不知道大湿口里的 free 究竟是自由还是免费抑或是两者兼有之.....

UTF-16 编码的问题

经上所述，虽然八戒会爬树，缺省还是靠不住（八戒毕竟还是公的嘛）。很多的乱码问题，很可能就是这种多变的缺省所害的。所以不能依赖于这些缺省。前面已经做过明确指定 GBK 编码的测试，这次我们使用 UTF-16 再试下，可以先简单计算一下，“hello 你好”7 个字符都在 BMP 中都是两字节，所以 $7 \times 2 = 14$ ，对吧？再跑一下：



尼玛!! 又见红了！咋猜啥啥不是呢？贝利的乌鸦嘴也没这么衰！仔细看看，它说实际是 16，哪里又多出两个字节来？这里也没有什么增补平面的字符呀！没辙了，要么打印出来，要么直接断点查看，我们就简单打印它好了：



元凶终于现身了，就在最头部的地方，楞是多出了两字节“FEFF”，这是啥呢？我想有人看到这里已经明白了，这就是 BOM，在下一篇我们再谈论这个话题。（至于原因嘛，这篇实在已经是太长了！）

第七章 BOM

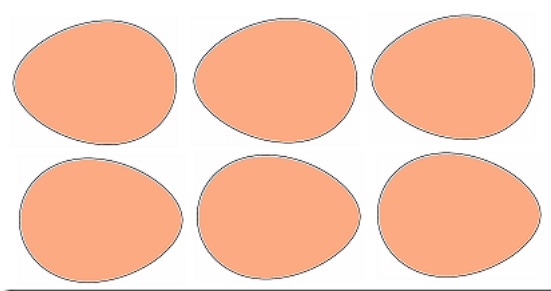
原文地址: <http://my.oschina.net/goldenshaw/blog/323248>

摘要: 本文讨论了 Unicode 中的 BOM 及与 BOM 紧密相关的端法 (endian) 问题。

在前一篇的最后, 留下了一个问题, 即 `string.getBytes("UTF-16")` 会在开头多出两个字节 "FEFF" 来, Unicode 中称之为 BOM, 接下来就让我们一起来了解有关 BOM 的知识, 在此之前我们需要说说有关端法的知识。

什么是端法 (endian) ?

在具体介绍它之前, 让我们先看看鸡蛋的两种摆法:



大家看出区别来了吗? 上面的一排都是尖的一端 (或者说小端) 向着左, 较圆的一端 (或者说大端) 向着右; 而下面一排正好相反。

| 画得不咋的, 大家凑合看就是了, 画出了《蒙娜丽莎》(Mona Lisa) 的达芬奇

| 据说开始学画画时也画过一段时间鸡蛋呢, 有说画了几天的, 也有说画了几年!

如果按照我们从左到右的习惯认为左是前面, 上面可以说是小端在前, 而下面的则是大端在前。有人可能要问, 这与我们的 BOM 有何关系? 我们知道 UTF-16 一个代码单元有两个字节, 如果把一代码单元比作一个鸡蛋, 那么它也有两个端, 一个字节是小端, 另一个则是大端。

大端法 (Big endian)

以两个 UTF-16 的编码 0x0048 与 0x4F60 为例, 如果我们把它们书写成 00 48 4F 60, 这样对我们而言也是非常自然的一种方式, 00 与 4F 都属于高位, 我们又常常说 “高大高大” 的, 高与大总是关系紧密, 自然这样一种高位在前的方式就是大端法 (Big endian) 了。

| 所谓的大端法, 就是大端在前,

| 上面图中下面一排的鸡蛋就是大端在前, 因此这种摆法也可以称为大端法。

那么, 自然的, 与大端法相反的那种就是小端法了。

小端法 (Little endian)

还是以两个 UTF-16 的编码 0x0048 与 0x4F60 为例, 如果我们把它们书写成 48 00 60 4F, 那么这样一种低位在前的方式就是小端法 (Little endian) 了。

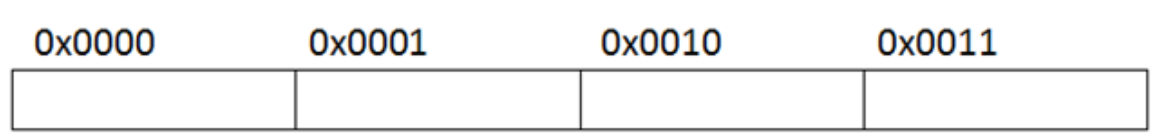
| 所谓的小端法, 就是小端在前,

| 上面图中上面一排的鸡蛋就是小端在前, 因此这种摆法也可以称为小端法。

我估计很多人会有些疑问, 为什么弄出这么一种很 “不自然” 很别扭的方式来呢? 请注意, 我给很不自然里的 “不自然” 三个字打了引号, 而且你可能也注意到了我在前面强调了 “书写” 两字, 其实呢, 大小端法应该是从存储层面考虑的, 在此之前让我们看看我们是如何看待内存布局的。

内存中的存储

如果有一排的格子来表示内存, 我们来给它们编号 (其实就是地址), 那么自然按照从左到右的习惯, 地址编号越来越大, 下面是一个四字节内存的示意图:



那么现在把大端法表示的四字节放进去，结果如下：

0x0000	0x0001	0x0010	0x0011
00	48	4F	60

那么我们发现，就单个编码而言，高位的字节反而放到了低地址上，而低位的字节却放到了高地址上。

- | 如上，高位的字节 00 放到了低地址 0x0000 上，
- | 低位的字节 48 却放到了高地址 0x0001 上。4F60 的情况也与此类似。

这么下来，我们所谓很“自然”的大端法反而有点不自然了。

与此相反，让我们把小端法放入内存，结果如下：

0x0000	0x0001	0x0010	0x0011
48	00	60	4F

那么，与大端法相反，现在它的高低字节反而与地址的高低位自然地对应上了。

- | 所以呢，我们前面说小端“不自然”，那是对于书写时的情况而言，
- | 考虑到存储层面，它看上去倒似乎更自然了。

需要强调的是，所谓大小端仅仅是字节间的关系，

- | 这也暗示了只有多字节情况才会有所谓的端法，而通常又在偶数字节情况下更为普遍，
- | 如 UTF-16，UTF-32，这样才能更好分出“两个端”来。
- | 下面谈到 UTF-8 时将会再度阐述这一问题。

每个单独字节里的 8 个位依然还是高位在前，无论大小端均是如此。下图是小端法单个字节内部以二进制表示的示意图：

0x0000	0x0001	0x0010	0x0011
01001000	00000000	01100000	01001111

- | 当然了，建立在字节抽象层面上的操作已经无需关注字节内部究竟是什么端法了，
- | 甚至已经不存在端法这一说法了。

大小端法的来历

关于 Big endian 和 Little endian，它们是有来头的，下面文字引自[阮一峰的网络日志](#)：

- | 这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。
- | 在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头 (Big-Endian)
- | 敲开还是从小头 (Little-Endian) 敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

所以你明白了为何前面先画了几个鸡蛋来示意。我们当然不会无聊到为鸡蛋从哪头敲开去打仗，不过关于端法哪种好也是争论不休的，在前面我们就谈到了哪种更“自然”的问题。

端法与系统架构

在 Windows 平台下，当使用记事本程序保存文件时，编码里有几个选项：



可以看到一个“Unicode”和“Unicode big endian”，通过以上名称的对比及对大端法的特别标

示，我们可以猜测出，Windows 下缺省是小端法。

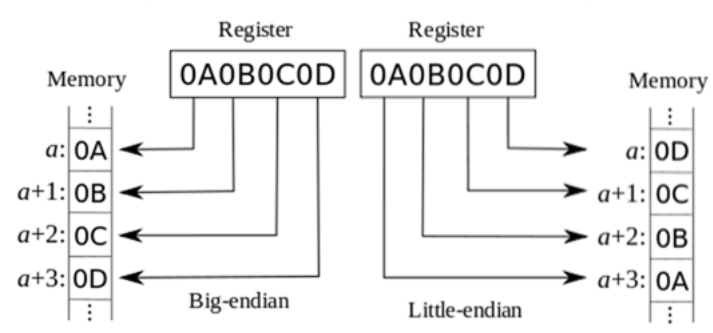
| 注：关于这里的 Unicode，前面篇章中也有提及，实际就是 UTF-16 编码。

Windows 平台为何使用小端法呢？说起来与 CPU 制造商英特尔（Intel）又有很大关系。

| 这两者我们又常叫它们为 Wintel 联盟。（Wintel=Windows+Intel）

内存（Memory）中使用端法其实又是受到寄存器（Register）中使用的端法的影响，因为两者之间经常要来回拷贝数据。英特尔的 CPU 就使用了小端法。

需要强调的另一点是，虽然我们讨论是字符集编码，但端法并不限于此，还可以是其它，比如一个 int，通常是 4 字节，以一个整数 0x0A0B0C0D 为例（以下图片来自 wiki 的截图）：



除此之外，图片文件中也可能会涉及端法的问题，网络传输中也同样有端法的问题，对此有兴趣的可以参考 <http://en.wikipedia.org/wiki/Endianness>。

至于为何英特尔采用了小端法，而其它一些厂商又使用了大端法以及这两种端法到底哪种好等问题，这里就不打算深入下去了，总之，大家知道这个世界比较乱就是了，有兴趣有精力的同学可以自行搜索以了解更多。

| 在 2010 年央视春晚小品《捐助》中，赵本山的徒弟王小利扮演的亲家说：“他刨的不深，我要往祖坟上刨。”针对端法的争论，这里就不打算往祖坟上刨了，另一方面的原因是刨不动了，再刨就到硬件层面上去了，所以呢，非不为也，是不能也！

回到编码的问题，在记事本中以 ANSI 之外的三种编码分别保存一下“hello 你好”，分别命名为 UTF16BE.txt，UTF16.txt，UTF8.txt（分别对应“Unicode big endian”，“Unicode”，“UTF-8”）

以 16 进制查看一下：

UTF16BE.txt																
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	fe	ff	00	68	00	65	00	6c	00	6c	00	6f	4f	60	59	7d
UTF16.txt																
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	ff	fe	68	00	65	00	6c	00	6c	00	6f	00	60	4f	7d	59
UTF8.txt																
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	ef	bb	bf	68	65	6c	6c	6f	e4	bd	a0	e5	a5	bd		

什么是 BOM？

BOM=Byte Order Mark，翻译过来就是“字节顺序标识”，也即是上图中红色框中的部分。

自然地，这里所谓的字节顺序其实就是指使用了哪种端法。

从上图中可以看出：

- UTF-16 BE (Big Endian) 的 BOM 是：FE FF
- UTF-16 LE (Little Endian) 的 BOM 是：FF FE
- UTF-8 的 BOM 是：EF BB BF

| 注：前面说到，getBytes(“UTF-16”)得到的缺省 BOM 是“FEFF”，

| 可见 JVM 中缺省是大端法，这与 Windows 平台下缺省为小端法恰好相反。

UTF-32 的 BOM

虽然前面一直在说的是 UTF-16，但 UTF-32 同样也有 BOM，以下代码是一些测试：

```
1| package org.jcc.core.encode;
2| import java.io.UnsupportedEncodingException;
3| import org.junit.Test;
4| import static javax.xml.bind.DatatypeConverter.printHexBinary;
5| import static org.assertj.core.api.Assertions.assertThat;
6|
7| public class BOMTest {
8|     @Test
9|     public void testBOM() throws UnsupportedEncodingException {
10|         String s = "hello 你好";
11|         // ===== UTF-16
12|         // java 中的缺省是大端
13|
14|         assertThat(printHexBinary(s.getBytes("UTF-16"))).isEqualTo("FEFF00680065006C006C006F4F60597D");
15|         // 注意：特别指明端法后的字节数组将不再带上 BOM
16|
17|         assertThat(printHexBinary(s.getBytes("UTF-16BE"))).isEqualTo("00680065006C006C006F4F60597D");
18|         // 小端法表示的字节数组需要特别使用"UTF-16LE"来获取
19|
20|         assertThat(printHexBinary(s.getBytes("UTF-16LE"))).isEqualTo("680065006C006C006F00604F7D59");
21|
22|         // ===== UTF-8
23|         // java 中，UTF-8 缺省不带 BOM，这点与记事本又不同
24|         // 另：UTF-8 不存在所谓的大小端，全部是大端，BOM 仅仅作为一种所用编码的指示
25|         assertThat(printHexBinary(s.getBytes("UTF-8"))).isEqualTo("68656C6C6FE4BDA0E5A5BD");
26|
27|         // ===== UTF-32
28|         // UTF-32 太长，用短一点的串来测试
29|         String s32 = "he";
30|         // 注意：在本机测试时，缺省情况下也没有 BOM（不同的虚拟机实现可能会不一样!!）
31|         assertThat(printHexBinary(s32.getBytes("UTF-32"))).isEqualTo("0000006800000065");
32|         // 为防止意外，最好明确设置
33|         assertThat(printHexBinary(s32.getBytes("UTF-32BE"))).isEqualTo("0000006800000065");
34|         assertThat(printHexBinary(s32.getBytes("UTF-32LE"))).isEqualTo("6800000065000000");
35|     }
36| }
```

不过，有点遗憾的是，UTF-32 在本机测试时，缺省情况下也没有 BOM 输出，这点与 UTF-16 的情况又不同：

但以上仅是在本机测试的情况，请不要将此作为一个结论，不同的虚拟机实现可能会不一样!!

下图是各种 BOM 的一个汇总（图片截取自 unicode.org）：

Bytes	Encoding Form
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

BOM 与码点 U+FEFF

BOM 其实就是 U+FEFF 这一码点，“EF BB BF”就是这一码点在 UTF-8 下的编码。代码如下：

```
1| @Test
2| public void testBomCodePoint() throws UnsupportedOperationException {
3|     String s = "\uFEFF";
4|     assertThat(printHexBinary(s.getBytes("UTF-8"))).isEqualTo("EFBBBF");
5| }
```

U+FEFF 称为“zero-width non-breaking space”

| 字面义：零宽度非换行空格。

| 也即碰到时把它解释成这样，显示上的实际效果就是啥也没显示。

缩写成“ZWNBS”，如下图所示：



在用作 BOM 之后，Unicode 不再建议这样去解释(deprecated)，而是建议用 U+2060 来代替，U+FEFF 就作为 BOM 的专用。

U+2060 称为“Word Joiner”（字面义：词连接器），缩写为“WJ”，如下图所示：



UTF-8 的 BOM

从前面测试可知，java 中，UTF-8 缺省不带 BOM，这点与记事本又不同：

- | 按 Unicode 组织的说法，UTF-8 可带可不带 BOM，不作强制要求，但不推荐用 BOM，
- | 原因之一是为与 ASCII 的兼容。

另：UTF-8 也不存在所谓的大小端两种情况，统一为大端法，BOM 仅仅作为一种所用编码的指示。

- | UTF-8 中有一字节的情况，这种情况，就没有两端的说法了。
- | 至于另外的二，三，四字节情况，以三字节为例，如果你一定要弄出端法，也不是说不可以，
- | 比如，小端法就是“小-中-大”，大端法就是“大-中-小”。但现实情况是 UTF-8 仅仅采用了一种端法，就是大端法。

UTF-8 中的 BOM 已经偏离了它的本意，而这一点估计也是 Unicode 组织不推荐在 UTF-8 中使用 BOM 的一大原因。

- | 在 eclipse 中，以 UTF-8 保存时就没有 BOM，但它的编辑器也能正确处理带 BOM 的情况。

总结

其实关于端法及 BOM 并没有太多好说的，通常大家知道有这么一回事或者说有那么一些“坑”也就够了，关于 BOM 的话题就谈到这里。

第八章 ASCII 和 ISO-8859-1

原文地址: <http://my.oschina.net/goldenshaw/blog/351949>

摘要: 简单介绍了 ASCII 和 ISO-8859-1 两个常见的字符集 (编码)

在前面其实也谈到了 ASCII 了, 但并没有很具体, 作为一个完整系列的一部分, 还是有必要谈一下, 也作为后面讨论的一些基础。

ASCII

它的全称是 American Standard Code for Information Interchange (美国信息交换标准代码), 是一个 7 位字符编码方案。

下面是它的一张简图 (来自 <http://www.asciitable.com/index/asciifull.gif>):

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

ASCII 定义了 128 个字符, 包括 33 个不可打印的控制字符(non-printing control characters) 和 95 个可打印的字符。

控制字符

32 以下的及最后一个 127 是所谓的控制字符。(0x00~0x1F 以及 0x7F)

| 即上图最左边一列的 32 个字符及最右边一列最后的一个字符 (DEL, 删除)

熟悉的有 0x09 (TAB, horizontal tab, 水平制表符), 0x0A (LF, line feed, ' \n' 换行符), 0x0D (CR, carriage return, ' \r' 回车符), 其它的很多现在已经是废弃不用了。

| 关于回车换行 (' \r\n'), 在屏幕还不普及的时代, 结果输出经常是依赖于所谓的电传打印机, 打印头沿着打印杆从左向右移动并打印出一个个字符。

| 当碰到一个回车符时 (CR, 0x0D, ' \r'), 打印机就指示打印头重新回到最左边的位置上,

| 这即是传统意义上的回车了。(你可以把打印头想像成一辆小车, 回车即是退回原处, 现代意义上的回车则通常包含回车与换行两个动作)

回车符后常跟着一个换行符 (LF, 0x0A, ' \n'), 打印机收到换行符就会指示滚筒滚动, 这样, 打印头就对准了纸张上的新的一行。如果没有换行, 新的打印输出就会重叠在上一行上, 有时走纸不顺畅时也会造成这种后果。

- | 目前，在 Windows 系统上，回车键会产生两个字符 CRLF，一起表示换行。
- | Unix/Linux 之类的则单独用 LF 表示换行，而苹果的 Mac 则单独用 CR 来表示换行。

其它字符

包括空格，字母，数字以及一些常见的标点符号等。

- | 关于空格（SPACE，图中第二列第一个，0x20），它没被归到控制字符类，
- | 当然，你可能会纠结它是否算可打印的。

由于只定义了 $2^7=128$ 个字符，用 7bit 即可完全编码，而一字节 8bit 的容量是 256，所以一字节 ASCII 的编码最高位总是 0，这为后来的编码方案兼容它带来的便利。

ISO-8859-1

ISO-8859-1 又称 Latin-1，是一个 8 位单字节字符集，它把 ASCII 的最高位也利用起来，并兼容了 ASCII，新增的理论空间是 128，但它并没有完全用完：

（截图来自 http://zh.wikipedia.org/wiki/ISO/IEC_8859-1）

ISO/IEC 8859-1																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x																
1x																
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8x																
9x																
Ax	NSP	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

可以看到，新增部分也保留了前面的 32 个位置（中间绿色部分，0x80-0x9F），与前面的 ASCII 部分类似，所以实际只增加了 $128-32=96$ 个，主要是西欧的一些字符，另外可以看到乘号（0xD7）和除号（0xF7）也被包含进来了。

- | 0x00-0x1F、0x7F、0x80-0x9F 在此字符集中未有定义。（即图中的绿色部分）

ISO-8859-1 能与 ASCII 兼容，同时它的适用范围又较广，一些协议或软件把它作为一种缺省编码，当然，现在更好的选择是 UTF-8。

第九章 GB2312, GBK, GB18030

原文地址: <http://my.oschina.net/goldenshaw/blog/352859>

摘要: 关于 GB2312, GBK, GB18030 编码的一些介绍, 还有区位码, 国标码, 机内码间的转换关系。

前面的一些篇章更多谈论了 Unicode 的相关话题, 虽然也有提到 GBK 等编码, 但都没细说, 这里打算系统说一下。GB 系列包括 GB2312, GBK, GB18030。

- | 前面已经提过, GB=Guo Biao=国标=国家标准, 至于所谓的 2312 就是一编号了,
- | 没有其它特别的意义, 18030 类似。GBK 没有编号, 所以它实际上并不是国家标准,
- | 只是一个事实标准, GBK 中 K 指“扩展”的意思。

最早的是 GB2312, 我们从它开始说起。

GB2312

以下为一简介 (官方文档见[这里](#), 用 IE 打开, 它要安装一个 ActiveX 我插呀件~):

- | GB 2312-1980, 全称《信息交换用汉字编码字符集 基本集》, 由国家标准总局于 1980 年 3 月 9 号发布, 1981 年 5 月 1 日实施, 通行于大陆。新加坡等地也使用此编码。它是一个简
- | 化字的编码规范, 也包括其他的符号、字母、日文假名等, 共 7445 个图形字符, 其中汉字
- | 占 6763 个。

上述官网地址无法下载, 如果你想下载, 可试下下面的 FTP 地址 (比标准方案多增加了一些字符):

<ftp://ftp.oreilly.com/examples/cjkvinfo/AppE/gb2312.pdf>

作为一个编码字符集而言, 前面也曾说到, 它采用了所谓的二维区位编号, 下面是一个概览图:

它是一个 94*94 的表格, 理论上有 94*94=8836 个空间。

横的叫区, 竖的叫位, 总共 94 个区, 区和位的编号都从 1 开始。可以看到粗略有三大部分。

94 个区

- 1、中间黑色的主体部分即是汉字区了, 具体为 16-87 区, 共 87-16+1=72 个区, 理论空间为 72*94=6768。

- | 从上图中可以看到中间有 5 个编码为空白 (中间靠右边部分, 55 区最后 5 个位),

| 所以总共有 6768-5=6763 个汉字。

| 一级汉字与二级汉字：

53	侧	闸	眨	栅	榨	乍	炸	诈	摘	斋	宅	窄
54	帧	症	郑	证	芝	枝	支	肢	知	脂	汁	
55	住	注	祝	驻	抓	爪	拽	专	砖	转	撰	篆
56	宁	丌	兀	丐	卅	丕	亘	丞	扁	舜	噩	丨
57	佟	佗	侏	伽	佶	倨	侑	侔	侗	侑	侗	侑
58	淞	冢	冥	讠	讠	讠	讠	讠	讠	讠	讠	讠

| 第 16—55 区：一级汉字，3755 个（以拼音字母排序）

| 第 56—87 区：二级汉字，3008 个（以部首笔画排序）

- 2、最下面的 88-94 区是有待进一步标准化的空白区。
- 3、关于前面的 01-15 区，下图为概览图左上角的局部放大图：

位 区	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
	1	SP	、	。	·	-	˘	˙	˚	々	一	~		...	‘	”	[]	< >	《 》	「 」	『 』	【 】	±	×	÷									
2																	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.
3	!	"	#	¥	%	&	/	()	☆+	, -	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B			
4	あ	い	う	え	お	か	き	く	げ	こ	さ	し	じ	ず	せ	そ	た	だ	ち															
5	ア	アイ	イ	ウ	ウエ	エ	オ	オカ	ガ	キ	グ	ケ	ゴ	サ	ザ	シ	ジ	ズ	セ	ゼ	ゾ	タ	チ	デ										
6	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M	N	Ξ	O	Π	P	Σ	T	Υ	Φ	X	Ψ	Ω									α β	
7	A	B	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я		
8	ā	á	ǎ	à	ē	é	ě	è	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	í	
9				—	—				...	:	:	:	:	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	┐	
10																																		
11																																		
12																																		
13																																		
14																																		
15																																		
16	啊	阿	埃	挨	哎	哀	皑	癌	矮	艾	碍	爱	隘	鞍	氨	安	俺	按	暗	岸	胺	案	肮	昂	盎	凹	熬	熬	翱	袄	傲	奥		
17	薄	雹	堡	堡	饱	宝	抱	报	暴	豹	鲍	爆	杯	碑	悲	卑	北	辈	背	贝	钡	倍	狈	备	惫	焙	被	奔	苯	笨	崩	绷	甬	

- (1). 01-09 区为符号、字母、日文假名等，部分区还有空白位。
| 03 区即是对应 ASCII 字符的全角字符区。输入法的全角模式下输入的即是这些字符。
- (2). 10-15 区也是有待进一步标准化的空白区。

各区的一个具体情况：

- 第 01 区：中文标点、数学符号以及一些特殊字符
- 第 02 区：序号
- 第 03 区：全角西文字符
- 第 04 区：日文平假名
- 第 05 区：日文片假名
- 第 06 区：希腊字母表
- 第 07 区：俄文字母表
- 第 08 区：中文拼音字母表
- 第 09 区：制表符号
- 第 10—15 区：未定义
- 第 16—55 区：一级汉字（以拼音字母排序）
- 第 56—87 区：二级汉字（以部首笔画排序）
- 第 88—94 区：未定义

区位码

在上图中还标出了一个汉字“啊”，它就是 GB2312 方案中的天字第一号汉字，它处于 16 区 01 位上，所以它的区位码即是 1601。

| 所谓区位码就是这一 94*94 的大表格中的行号与列号了，均从 1 开始编号。

| 第一个字符 0101 为“全角空格”（图中显示为 SP (space)）。

国标码

将区位码的区和位分别加上 32 (=0x20) 就得到了国标码。

| “啊”的区位码是 16-01，分别加 32，得到 16+32-01+32=48-33，即是国标码。

| 当然，你通常应该写成 16 进制，48-33 即是 0x30-0x21，

| 所以 3021 即是“啊”十六进制的国标码，使用两字节保存，30 为高字节，21 为低字节。

| 如下：

First Byte	位	1	2	3	4	5	6
0 1 0 0 0 0 1 1	1	SP
0 1 0 0 0 0 1 0	2						
0 1 0 0 0 0 1 1	3	!	"	#	%	%	%
0 1 0 0 1 0 0 0	4	あ	い	う	え	お	か
0 1 0 0 1 0 0 1	5	ア	イ	ウ	エ	オ	カ
0 1 0 0 1 1 0 0	6	A	B	Γ	Δ	E	Z
0 1 0 0 1 1 0 1	7	A	B	Γ	Δ	E	Z
0 1 0 0 1 0 0 0	8	ā	á	ǎ	ǎ	ē	ē
0 1 0 0 1 0 0 1	9						
0 1 0 0 1 0 1 0	10						
0 1 0 0 1 0 1 1	11						
0 1 0 0 1 1 0 0	12						
0 1 0 0 1 1 0 1	13						
0 1 0 0 1 1 1 0	14						
0 1 0 0 1 1 1 1	15						
0 1 1 0 0 0 0 0	16	啊	阿	埃	挨	哎	噍
0 1 1 0 0 0 0 1	17	藎	霍	保	保	恂	全

GB2312 方案规定，对上述表中任意一个图形字符都采用两个字节表示，每个字节均采用七位编码表示。

| 如上图所示，只用了 7 位，这即是说最高位就是 0 了。

但为何不直接采用区位码呢？为什么要加 32 呢？你也许还记得前面说到 ASCII 时，前面 32 个字符是控制码，中文系统自然也不能少了这些控制码，为了不与这些控制码冲突，加上 32 就能跳过它们了。

| 一字节有 128 个空间，128-32=96，实际上，ASCII 中第 127 个也是控制码（DEL，删除），

| 再减去就还有 95 个有效位，再加上区位从 1 开始，又损失了一位，所以最终只有 94 个有效

| 位了，这也是前面为何是一个 94*94 的表格。

国标码的定位实际应该是与 ASCII 一致的，是作为国家信息交换的标准码。从设计上看，它并没打算兼容 ASCII，它已经把 ASCII 中的字符收录了过来，不过是作为所谓的全角字符来看待，但全角英文显示效果其实是很差的，下面是全角英文的一个示例：

| h e l l o , w o r l d

显得非常不紧凑，最终，一种能兼容 ASCII 的存储方案得到了广泛采纳，这就是所谓的机内码了。

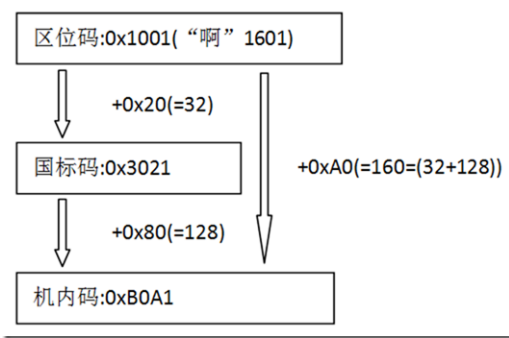
机内码

将国标码高低字节分别加上 0x80 (=128) 就得到了机内码（有时又叫交换码）。128 的二进制形式为 10000000，加 128，简单地讲，就是把国标码最高位置成 1。至于为什么要这样呢？我想你应该也清楚了，就是要兼容 ASCII，ASCII 最高位为 0，国标码加 128 后，高低字节的最高位都成了 1，这样就与 ASCII 区分开来。

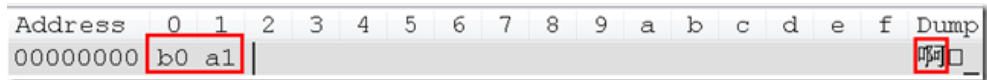
| 将“啊”的国标码 3021 分别加上 0x80，0x30+0x80=0xB0，0x21+0x80=0xA1，

| 所以 B0A1 即是机内码。

如果从区位码算起，那么则是加上 $0x20+0x80=32+128=160=0xA0$ ，也即区位码的区和位分别直接加上 $0xA0$ 即可得到机内码，如下图所示：



如果你新建一个文本文件，录入“啊”字，以 GB2312 编码方式保存(使用 GBK 即可，它兼容 GB2312)，再用十六进制查看，你会发现使用的是机内码：



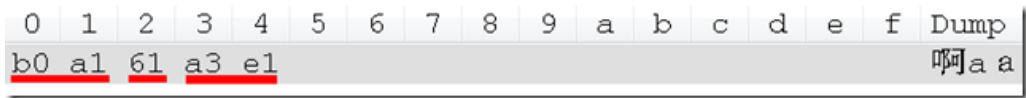
使用代码的测试也可验证这一点：

```
1| @Test
2| public void testAh() throws UnsupportedOperationException {
3|     String ah = "啊";
4|     assertThat(DatatypeConverter.printHexBinary(ah.getBytes("GB2312"))).isEqualTo("B0A1");
5| }
```

虽然我们常把 GB2312 称为国标码，但我们应该清楚，实际存储使用的是机内码，通常说到 GB2312 编码时指的就是这个机内码了。它能兼容 ASCII，是一种变长的编码方案，对 ASCII 中的字符（也即所谓的“半角西文字符”）采用一字节编码，最高位为 0；对区位表中的字符采用两字节编码，且每字节最高位均为 1，以此区分。

自然，全角英文字符就是两字节编码了，跟汉字是一样的。

下面是一个混合了汉字，半角字母 a 和全角字母 a 的编码示例，共 5 个字节：

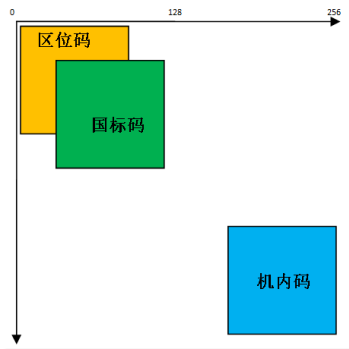


我们说 GB2312 是一个变长编码方案，是站在其兼容 ASCII 编码角度而言，就其方案标准本身定义的字符而言，它是一个双字节定长编码方案。

你可能会想，那国标码还有什么用？

- 我个人觉得，国标码既然称为中文信息交换的标准码，必然要成为“机内”码才有意义，只不过由于各种原因，最终未能如愿。早期的一些系统或者一些小型的嵌入式系统或许采纳了它做为“机内”码。当然以上为个人猜测，仅供参考。
- 另：我在前面的一些文章中谈到区位码时把它与机内码混为一谈，特此更正。

下面是三种码在 256*256 坐标中的位置的一个示意图：



GBK

GBK 是对 GB2312 的一个扩展，兼容 GB2312，因此也兼容 ASCII，也是一个变长编码方案。下面是一个简介：

- | GBK 总体编码范围为 8140-FEFE，首字节在 81-FE 之间，尾字节在 40-FE 之间，
- | 总计 23940 个码位，共收入 21886 个汉字和图形符号，
- | 其中汉字（包括部首和构件）21003 个，图形符号 883 个。

GBK 是国家有关部门与一些信息行业企业等一起合作推出的方案，但并未作为国家标准发布，只是一个事实上的标准，一个过渡方案，为 GB18030 标准作的一个准备。

首字节 (lead byte)

下面是 Windows Code page: 936 (GBK)，第一字节的概况：

- | 来自 <http://msdn.microsoft.com/en-US/goglobal/cc305153.aspx>

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	MUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	NAK 0014	SYN 0015	ETB 0016	CAN 0017	EM 0018	SUB 0019	ESC 001A	FS 001B	GS 001C	RS 001D	US 001E	001F
20	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
30	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
40	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
50	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
60	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
70	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
80	8080	8081	8082	8083	8084	8085	8086	8087	8088	8089	808A	808B	808C	808D	808E	808F
90	9080	9081	9082	9083	9084	9085	9086	9087	9088	9089	908A	908B	908C	908D	908E	908F
A0	A080	A081	A082	A083	A084	A085	A086	A087	A088	A089	A08A	A08B	A08C	A08D	A08E	A08F
B0	B080	B081	B082	B083	B084	B085	B086	B087	B088	B089	B08A	B08B	B08C	B08D	B08E	B08F
C0	C080	C081	C082	C083	C084	C085	C086	C087	C088	C089	C08A	C08B	C08C	C08D	C08E	C08F
D0	D080	D081	D082	D083	D084	D085	D086	D087	D088	D089	D08A	D08B	D08C	D08D	D08E	D08F
E0	E080	E081	E082	E083	E084	E085	E086	E087	E088	E089	E08A	E08B	E08C	E08D	E08E	E08F
F0	F080	F081	F082	F083	F084	F085	F086	F087	F088	F089	F08A	F08B	F08C	F08D	F08E	F08F

- | Code page 936 实质上是 GBK 到 UTF-16 编码的一个转换表，
- | 图中字符下面标注的四位 16 进制数字即是 UTF-16 编码

- 1、上面部分是兼容 ASCII 单字节编码。
- 2、下面阴影部分是双字节编码中的第一个字节，表中作为超链接，可以点击进去查看具体内容。

- | 注：0x80(=128)被用于欧元符。(图中小圆框部分) 0xFF 则保留，实际共有 128-2=126 块。
- | 另：新的 GB18030 标准使用双字节编码欧元符号，去掉了这个单字节编码。

第二字节

前面说到“啊”的机内码是 B0A1，我们点击 B0（上图中红色小框部分）去查看一下：

- | 来自 <http://msdn.microsoft.com/en-US/goglobal/gg675356>

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
40	癩 7646	癩 7647	癩 7648	癩 7649	癩 764A	癩 764B	癩 764C	癩 764D	癩 764E	癩 764F	癩 7650	癩 7651	癩 7652	癩 7653	癩 7654	癩 7655
50	癩 7656	癩 7657	癩 7658	癩 7659	癩 765A	癩 765B	癩 765C	癩 765D	癩 765E	癩 765F	癩 7660	癩 7661	癩 7662	癩 7663	癩 7664	癩 7665
60	癩 7666	癩 7667	癩 7668	癩 7669	癩 766A	癩 766B	癩 766C	癩 766D	癩 766E	癩 766F	癩 7670	癩 7671	癩 7672	癩 7673	癩 7674	癩 7675
70	癩 7676	癩 7677	癩 7678	癩 7679	癩 767A	癩 767B	癩 767C	癩 767D	癩 767E	癩 767F	癩 7680	癩 7681	癩 7682	癩 7683	癩 7684	癩 7685
80	癩 7686	癩 7687	癩 7688	癩 7689	癩 768A	癩 768B	癩 768C	癩 768D	癩 768E	癩 768F	癩 7690	癩 7691	癩 7692	癩 7693	癩 7694	癩 7695
90	癩 7696	癩 7697	癩 7698	癩 7699	癩 769A	癩 769B	癩 769C	癩 769D	癩 769E	癩 769F	癩 76A0	癩 76A1	癩 76A2	癩 76A3	癩 76A4	癩 76A5
A0	癩 76A6	癩 76A7	癩 76A8	癩 76A9	癩 76AA	癩 76AB	癩 76AC	癩 76AD	癩 76AE	癩 76AF	癩 76B0	癩 76B1	癩 76B2	癩 76B3	癩 76B4	癩 76B5
B0	癩 76B6	癩 76B7	癩 76B8	癩 76B9	癩 76BA	癩 76BB	癩 76BC	癩 76BD	癩 76BE	癩 76BF	癩 76C0	癩 76C1	癩 76C2	癩 76C3	癩 76C4	癩 76C5
C0	癩 76C6	癩 76C7	癩 76C8	癩 76C9	癩 76CA	癩 76CB	癩 76CC	癩 76CD	癩 76CE	癩 76CF	癩 76D0	癩 76D1	癩 76D2	癩 76D3	癩 76D4	癩 76D5
D0	癩 76D6	癩 76D7	癩 76D8	癩 76D9	癩 76DA	癩 76DB	癩 76DC	癩 76DD	癩 76DE	癩 76DF	癩 76E0	癩 76E1	癩 76E2	癩 76E3	癩 76E4	癩 76E5
E0	癩 76E6	癩 76E7	癩 76E8	癩 76E9	癩 76EA	癩 76EB	癩 76EC	癩 76ED	癩 76EE	癩 76EF	癩 76F0	癩 76F1	癩 76F2	癩 76F3	癩 76F4	癩 76F5
F0	癩 76F6	癩 76F7	癩 76F8	癩 76F9	癩 76FA	癩 76FB	癩 76FC	癩 76FD	癩 76FE	癩 76FF	癩 7680	癩 7681	癩 7682	癩 7683	癩 7684	癩 7685

- 1、“啊”位于 A1 处，所以它是兼容 GB2312 的。而前面的那些字符就是 GBK 扩展的了。
 - | “啊”下面的 554A 即是它的 UTF-16 编码。
 - | GBK 与 UTF-16 之间编码的转换只能通过查表实现。
- 2、第二字节从 0x40 开始，不是从 0x00 也不是从 0x80 开始。表格只有 12 行。
 - | 因为不是从 0x80 开始，这意味着第二字节最高位也可能是 0。
 - | 这点与 GB2312 不同，GB2312 确保了无论是高低字节最高位均是 1。
- 3、另外 0x7F 和 0xFF 两处保留未定义。
 - | 所以实际有 $12 \times 16 - 2 = 192 - 2 = 190$ 个字符。
 - | 注：并非所有的块里面都是 190 个字符，也有不少是少于 190 的。
 - | 粗略估算可得 $126 \times 190 = 23940$ ，所以 GBK 也就是两万多个字符这样子。

GBK 还是 UTF-8?

GBK 使用两字节保存中文，也能兼容 ASCII，而对常用汉字，UTF-8 都是采用三字节编码，因此无论是全中文还是中英文混合的情况，GBK 保存的效率都要好于 UTF-8。

| 这也不奇怪，毕竟是亲生的。

但它也有些不好的地方，比如它不能支持一些国际性的文字，在国际化，通用性方面它肯定不如 UTF-8；就汉字而言，由于容量空间的限制，它也无法收录更多的汉字了。

| 所以，怎么选择，自己看着办。

GB18030

GB18030 前后发布了两个标准，最新的是 2005 年发布的 GB 18030-2005（《信息技术 中文编码字符集》），2000 年还有一版 GB18030-2000，更多了解可参考[百度百科](#)，或[官网](#)，（注：这个文件比较大）

对于多数用户而言，无需了解太多，这里也不打算详细介绍，下面是一些简介（针对最新的 GB18030-2005）：

- 1、它也是一个多字节编码方案，有一，二，四字节三种变长组合。
- 2、它的编码空间很大，高达 160 万（约数），这甚至超过了 Unicode 规定的 110 万（约数）。
- 3、它兼容 GB2312，基本兼容 GBK（只有很少几处不同）。
- 4、它收录高达 7 万多的汉字，Unicode 中的 CJK 统一汉字，CJK 统一汉字扩充 A，CJK 统一汉字扩充 B 均收录了进来。
- 5、它还支持许多少数民族如藏、蒙古、彝、维吾尔等的文字。

对于普通用户，超大字符集很少用到，通常情况下，如 Windows 系统下你可能要安装 GB18030 的相关插件才能处理及显示那些增补的字符，一般系统默认情况也不会安装能支持完整显示 GB18030 全体字符的字体。

| GB18030 作为一个强制标准，但由于采用了高达四字节的情形，
| 无论是操作系统还是各种应用软件，可能涉及许多调整才能很好地支持，
| 这决不是一件简单的事情。
| 作为国际性标准的 Unicode，BMP 以外的字符的处理与显示都还有很多不完善，
| 所以如果 GB18030 没有得到很好的支持，那也不足为奇了。

关于 GB 系列的编码就说到这里。