

乱码探源

本文摘自网友国栋发表在开源中国个人博客中的文章，仅作个人学习使用。每一章均会注明原文网址。

第一章 文件，文本文件以及编码

原文地址：<http://my.oschina.net/goldenshaw/blog/411022>

摘要：介绍了文件名在操作系统中的编码，以及一些非文本文件中的文本内容所用的编码

在前面的[字符集编码系列](#)中，已经探讨了几大主要的字符集编码。在此基础之上，这里将进一步探讨编码的应用及乱码的根源，我们先从基本的文件说起。

文件

文件（内容）就是字节序列。文本文件也是文件，所以它也是字节序列。

文件名与文件内容

通常说到文件时，指的是文件内容，但文件还有文件名，文件名与文件内容是分开存储的。

你可以在硬盘上新建一个文件，它的大小为 0。如下：

名称	类型	大小
新建文本文档.txt	文本文档	0 KB

但它是具有文件名的，比如上述的“新建文本文档.txt”，保存这些名字自然也要占用空间，只不过它与文件内容是分离的。

这些由操作系统的文件系统模块负责。

文件名是一段文本，因此它会涉及字符集编码。文件内容则视情况而定。

- 1. 文本文件，肯定会涉及字符集编码。
 - 常见的比如 txt, html, xml 以及各种源代码文件等等。
- 2. 非文本文件，比如图片文件 jpg, gif 之类，自然跟字符集编码无关了。
 - 有些文件，比如 Word 的 doc 之类的，混合了图片跟文本在里面，可以想像，
 - 其中的文本部分自然也会牵涉到字符集编码的问题，只不过这些编码不由我们去控制，
 - 我们通常也无须去关心。

文件名编码

文件名是一串文本，因此它必然涉及某种字符集编码，只不过这种编码是由操作系统决定的，我们无权干预。

那么，它用的是什么编码呢？在 Windows 下，可以简单做些实验。我们可以弄些奇怪的文件名如“★★★★.txt”，如下：

名称	类型	大小
★★★★.txt	文本文档	0 KB

结果也能保存。这些字符只在 Unicode 中才有，所以它肯定不是用的 GBK 之类的。

- Windows 下 NTFS 架构文件名使用 UTF-16 编码。
- 但对于 FAT 之类的，则是所谓的“OEM character set”。
- MSDN 上的原文如下：“NTFS stores file names in Unicode. In contrast, the older FAT12, FAT16, and FAT32 file systems use the OEM character set”（NTFS 使用 Unicode 存储文件名。与此相对，老的 FAT12, FAT16 和 FAT32 文件系统使用 OEM 字符集）。参见[这里](#)。
- 注：在 Windows 语境中，UTF-16 通常叫成 Unicode。

结合实验的结果，可以确定，Windows 使用 UTF-16 对文件名进行编码。（我的系统是 Win 7，文件系统为 NTFS）

不过，不同的系统平台可能使用了不同的编码。比如最新的 Linux 平台对文件名采用了 UTF-8 编码，但早期的则不好说，甚至没有一个标准。

如果你不是 Windows 平台，你也可以简单做些实验来大致猜测一下文件系统使用的编码。

文件的上传下载与文件名乱码

由于对文件名没有一个统一的编码，不同系统平台间交换文件时，中文文件名极易发生乱码现象。比如 FTP 上传，网页文件上传及下载等情况下经常能遇到文件名乱码。

不过，需要注意的是，交换过程中，**文件内容**不会发生任何改变。即便是文本文件，也完全是字节传送，不会涉及任何的编解码。

你可能碰到过这样的事，把一个文本文件从 Windows 平台上传到 Linux 平台，并在 Linux 平台下打开时发现乱码了，但这并不意味着文件内容有了什么变化，通常的原因是你的文件是用 GBK 编码的，但 Linux 平台下打开时它缺省可能用的是 UTF-8 编码去读取，因此，你只要调整成正确的编码去读取即可。

在这里，我们讨论了文件名的编码，之后，如无特别说明，谈到编码时均指对文件内容的编码。通常，这是我们更为关心的内容。

非文本文件中的字符集编码

通常，说到字符集编码都是对文本文件而言的，但非文本文件也是可能用到字符集编码的。

比如，Word 用什么编码？word 生成的 doc 或者 docx 虽然不是文本文件，但我们可以想像，它里面可能有图像，又有文字。其中的文字自然也会用到某种编码。只不过，这些都不需要我们去操心。

下面是一个实验，新建一个空白的 doc 文档，录入几个简单字符”Hello 你好”



保存成 doc 文件，再用 notepad++打开，以十六进制形式查看：

48 00 65 00 6c 00 6c 00 6f 00 60 4f 7d 59 0d 00 H.e.l.l.o. O}Y..

如上图，搜索到 hello 几个关键字，我们知道，“H”的码点是 U+0048，而“你”的码点则是 U+4F60，所以，很显然，用的是 UTF-16 LE (Little Endian, 小端序)

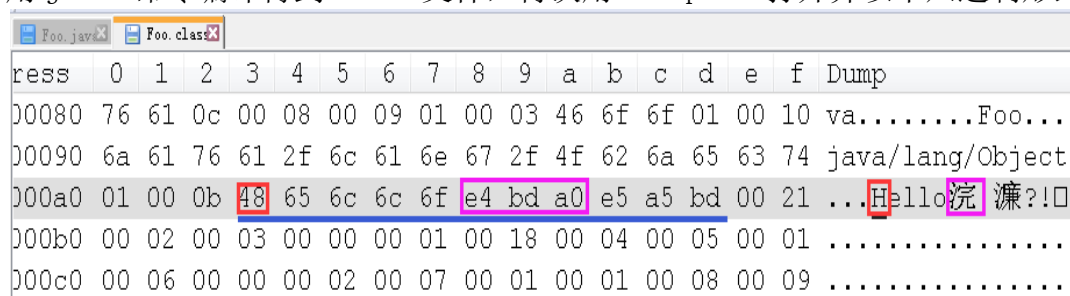
关于端序及 BOM 的相关话题，可参见[深入图解字符集与字符集编码（七）——BOM](#)
注：这只是我个人在本机测试的结果，不代表普遍的结论，不同平台不同版本下的可能会有差异，谁知道呢？我没有去研究过 doc 文件格式的规范，这个 doc 我还是用 WPS 生成的！

又比如，Java 中的 class 文件，它也不是文本文件，通常称为字节码文件。但它里面也会保存 String 的常量，这自然又要牵涉到编码。实际用的是所谓的“modified UTF-8”编码。

简单建立一个 java 文件，定义一个 string 常量”Hello 你好”：

```
public class Foo { static final String HI = "Hello 你好"; }
```

保存并用 javac 命令编译得到 class 文件，再次用 notepad++打开并以十六进制形式查看：



搜索到 Hello 几个关键字，紧接在它们后面的”e4 bd a0”就是”你”的 UTF-8 编码了。

在前面的[深入图解字符集与字符集编码（四）——Unicode](#)中，曾提到过，汉字的 UTF-8 编码通常都是以 e 打头，形如 ex xx xx 这样，这是常用汉字 UTF-8 编码的一个重要特征。

这个” modified UTF-8” 编码与 UTF-8 类似，但有一些差别，它的名字也暗示了这一点。

- | 比如对于 U+0000 它用了两字节来编码；
- | 还有对 U+FFFF 以上的字符它采用了 6 字节编码而非正常 UTF-8 的四字节编码，
- | 实质是对代理对（surrogate pairs）的值进行编码。
- | 详情可参见[这里](#)。

2、文本文件中的字符集编码

文本文件也是文件，所以它也是字节序列。当读取一个文本文件时，最重要的是确定它所使用的编码，只有这样才能正确的解码。

由于牵涉的情况较多，我们将在下一篇讨论这一问题。

第二章 确定文本文件的编码

原文地址：<http://my.oschina.net/goldenshaw/blog/413412>

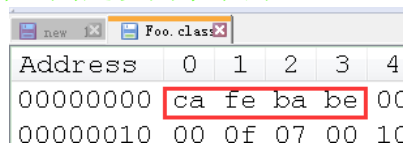
摘要：介绍了如何去确定一个文本文件所使用的编码，特别地以记事本保存“联通”为例进行了深入分析。

在上一篇中，探讨了文件名编码以及非文本文件中的文本内容的编码，在这里，将介绍更为重要的文本文件的编码。

1、混乱的现状

设想一下，如果在保存文本文件时，也同时把所使用的编码的信息也保存在文件内容里，那么，在再次读取时，确定所使用的编码就容易多了。

- | 很多的非文本文件比如图片文件通常会在文件的头部加上所谓的“magic number（魔法数字）”来作为一种标识。所谓的“magic number”，其实它就是一个或几个固定的字节构成的固定值，用于标识文件的种类（类似于签名）。比如 bmp 文件通常会以“42 4D”两字节开头。
- | 又比如 Java 的 class 文件，则是以四字节的“ca fe ba be”打头。（咖啡宝贝？）

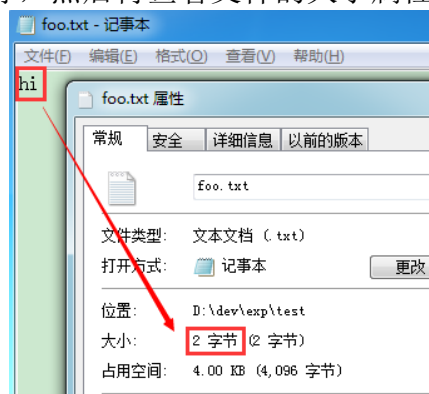


Address	0	1	2	3	4
00000000	ca	fe	ba	be	00
00000010	00	0f	07	00	1c

- | 即便没有文件后缀名，根据这些信息也是确定一个文件类型的手段。
- | 附：关于用 Notepad++ 查看十六进制的问题，这是一个插件，如果没有装，菜单--插件--plugin manager--available--HEX-Editor，装上它。装上后，它通常在工具栏的最右边，一个黑色的大写的斜体的“H”就是它。单击它可以在正常文本与 16 进制间切换。要进一步查看二进制，在文本区，右键--view in--to binary。

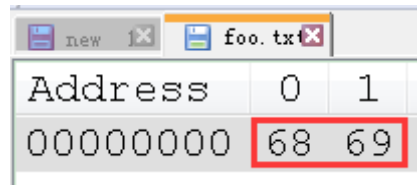
没有编码信息

那么，对于文本文件，有没有这样的好事呢？可以简单建立一个文本文件“foo.txt”，里面输入两个简单的字符，比如“hi”，保存，然后再查看文件的大小属性



然后，我们很遗憾地发现，大小只有 2，也即“hi”两个字符的大小，这意味着没有保存额外的所用编码的信息。

用十六进制形式查看，也可以发现这两个字节就是 hi 两字符的编码：



Address	0	1
00000000	68	69

| 关于字母的 ASCII 编码,可查看[深入图解字符集与字符集编码\(八\)——ASCII 和 ISO-8859-1](#)。

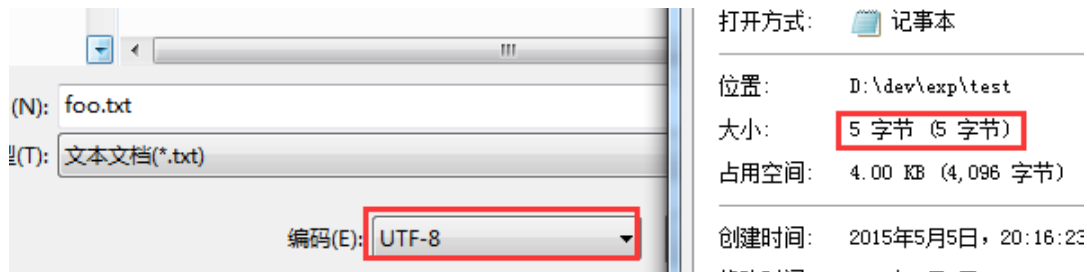
那么，现在很清楚了，文本文件仅仅是内容的字节序列，没有其它额外的信息。

BOM?

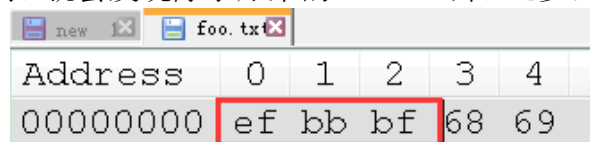
当然，说绝对没有额外信息也不完全正确，在之前的关于 BOM 的介绍中，我们看到 BOM 其实可以看成是一种额外的信息。

| 参见[深入图解字符集与字符集编码（七）——BOM](#)

保持内容不变，简单地“另存为”一下，在编码一栏选择“UTF-8”，再次查看属性将会发现大小变成了 5。



再次查看十六进制形式时，就会发现除了原来的“68 69”外，还多出了 UTF-8 的 BOM: “ef bb bf”



Address	0	1	2	3	4
00000000	ef	bb	bf	68	69

也正是以上三个与内容无关的字节使得大小变成了 5。

这个信息是与所用编码有关的，不过它仅能确定与 Unicode 相关的编码。

| 严格地说，BOM 的目的是用于确定字节序的。

另一方面，对于 UTF-8 而言，现在通常不建议使用 BOM，因为 UTF-8 的字节序是固定的，所以很多的 UTF-8 编码的文本文件其实是没有 BOM 的，不能简单地认为没有 BOM 就不是 UTF-8 了。

| 比如 eclipse 中生成的 UTF-8 文件默认就是不带 BOM 的。

| 微软的笔记本应该也是比较特殊的情况。

综上所述，文本文件通常没有一个特殊的头部信息来供确定所用的编码，另一方面，编码的种类又是五花八门，那么如何去确定编码呢？

2、确定编码的步骤

不妨就以记事本为考察对象，去探究一下它是如何确定编码的。

利用 BOM

前面说了，BOM 作为一种额外的信息，间接地表明了所使用的编码。尽管它原本的意图是要指明字节序，但曲线救国一下也未必不可。况且记事本还主动地为 UTF-8 也写入了 BOM，不加以利用这一信息自然是不明智的。

| 注：对 UTF-16 来说，BOM 是必须的，因为它是存在字节序的，弄反了字节序一个编码就会变成另一个编码了，那就彻底乱套了。不过一般很少用 UTF-16 编码来保存文件的，更多是在内存中使用它作为一种统一的编码。

但对于 UTF-8，很多时候也是没有 BOM 的，记事本遇到 UTF-8 without BOM 时又该怎么办呢？

我猜，我猜，我猜猜猜

如果内容中没有编码信息，又要去确定它使用的编码，这不是为难人是什么？好在“坑蒙拐骗”中的第二招“蒙”可以拿来用用。

“蒙”其实也是要讲点技术含量的，简单点自然就是是模式匹配了，或许一个或几个正则式就完了；复杂点，什么概率论，统计学，大数据统统给它弄上去，那逼格立马就高了有木有？当然了，记事本也就是一跑龙套的...

记事本跟“联通”有仇？

在编码界有这么一个传说：记事本跟“联通”有仇。这是怎么一回事呢？

新建一个文本文件“test.txt”，录入两个汉字“联通”，保存，关闭程序然后再次打开这一文件：



咦，这是什么鬼？咱们的“联通”呢？

深入分析

这其实就是竞猜失败的结果了，准确地讲，记事本把编码给猜成了 UTF-8.

为什么说是猜成 UTF-8 造成的呢？我也没见过源码！接下来会根据出现的现象，已有的证据来作出我们的推论，然后还会做些实验去验证。（没错，这就是科学！）

首先是这样一个事实：当我们保存时，使用的是缺省编码，也就是 GBK。

“联通”两字的 GBK 编码如下：

	0	1	2	3	4
dress					
000000	c1	aa	cd	a8	

然后，是这样一个现象：再次打开时，记事本突然就翻脸不认人了，显示出了一些奇怪的字符。

严格地讲，有三个字符，两个问号及一个 C 一样的字符，后面会分析为何会这样。

以上字节咋一看也没啥子特别的，领头字节也没有恰好等于 UTF-16 或 UTF-8 的 BOM，绝对标准的 GBK 模式，为啥记事本对它另眼相看了呢？

关于 GBK 等编码，可参见[深入图解字符集与字符集编码（九）——GB2312，GBK，GB18030](#)

那么，一个合理的猜测就是记事本可能把它当成了无 BOM 的 UTF-8 编码。

而对于 UTF-8 编码来说，它的编码模式还是很有自己特色的，那么我们换成二进制形式查看以上编码：

Address	0	1	2	3
00000000	11000001	10101010	11001101	10101000

看到以上编码，相信对 UTF-8 编码模式有一些了解的都知道是怎么回事了，我也用颜色的下标标注了关键的部分。

在前面的篇章中，也曾经几次说到过 UTF-8 的编码模式：

0XXXXXX
110XXXX10XXXXXX
1110XXXX10XXXXXX10XXXXXX
11110XXX10XXXXXX10XXXXXX10XXXXXX

见[深入图解字符集与字符集编码（三）——一定长与变长及深入图解字符集与字符集编码](#)

对比一下，不难发现上述两个编码完全符合二字节模式！也难怪记事本犯迷糊了。

| 自然，要做出一些“科学”的发现，你还是需要一定的基础的。

| 所以在这里也给出了很多前面文章的链接。

显示疑云

那么，为何又显示成了那样的效果呢？

既然推断它是 UTF-8 编码，让我们人肉解码一下：

- 前两字节构成一组：11000001 10101010
- 有效的码位有三段：11000001 10101010
- 重组成码点之后是：00000000 01101010

以上码点写成 16 进制是 U+006A，这明明是一个一字节的码点！对应的字母其实是小写字母“j”。

| 所以，这里其实是有错误的。这个码点不应该用二字节来编码。

如果你读过前面关于 Unicode 的篇章，就会明白，对于 UTF-8 编码而言，码点在 U+0000~U+007F（0-127）间的用一字节模式编码。码点在 U+0080~U+07FF（128-2047）间的才用二字节模式编码。

| 别问为什么！这叫乌龟的屁股——龟腚（规定）。

理论上讲，二字节的空间是完全可以囊括一字节编码的那些码点的，各种模式间其实是有重叠与冗余的。但如果一个码点适用于更少字节，那么它应该优先用更少字节的编码模式。

| 通常说 UTF-8 的二字节模式是“110x xxxx, 10xx xxxx”，但并非所有满足这些模式的编码

| 都是合法的 UTF-8 二字节编码。这里面其实是有个坑的。二字节首个码点为 U+0080，对应

| 二字节模式首字节为“1100 0010”，那么，所有合法的二字节模式首字节不应该小于此值。

| 显然，亲爱的微软的写记事本的程序员们，你们偷懒了！

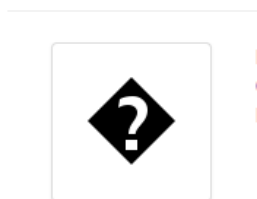
| 你们至少应该可以避免与“联通”结仇。

那么，虽然能够解出相应的码点，但其实是非法的组合，这也就是结果显示出“◆”的原因。这通常是一个明显的解释失败的标志。

| 注：“◆”本身是一个合法的 Unicode 字符，码点为 U+FFFD，

| 对应的 UTF-8 编码为：EF BF BD。如果字体不支持的话，可参见[这里](#)。

(U+FFFD)



| 这可不是“显示不出来”造成的，它本身就长这样。

| 这个码点的含义为“replacement character”（替换字符）。当碰到非法的字节时，显示系

| 统就用它来替换，然后显示这个替换字符来表示发生了替换。所以，那些真正“显示不出

| 来”的东西已经被替换了。（如果文件中本身就包含这个字符的话，替换上去的和原有的实

| 际是无法区分的。）

在这里，程序实际临时在内存中做了替换，如果你对它进行拷贝，得到的也将是它的值，而不是原来的值。

| 所以，显示层面出现了问号（包括早期的 ASCII 中那个问号“？”，U+001A，也常用作替换

| 字符），不代表它不清楚如何显示，而完全是因为最终交给它去渲染的就是“问号”字符。

| （可能是替换上去的，也可能本身就有的。）

在显示层面，原来的值实际已经丢失了。

| 至于为何显示了两个问号，大概是把两个字节当作了两次失败。

| 个人认为显示一个问号也能说得通。

再来看第二组

- 后两字节构成一组：11001101 10101000
- 有效的码位有三段：11001101 10101000
- 重组为码点之后是：00000011 01101000

以上码点写成 16 进制是 U+0368, 对应的字母其实是所谓的 COMBINING LATIN SMALL LETTER C (<— 这里第二个 C 就是 U+0368)。见“<http://www.fileformat.info/info/unicode/char/0368/index.htm>”

- | 显示时，它会紧贴在前面的字母上，这是 Unicode 中个人感觉比较奇葩的一些内容，
- | 如果你有兴趣，这是 [wiki 的一些介绍](#)
- | 前面乱码的截图中，那个怪怪的 C 就是这样来的，感觉好像与前面一个问号是一体的一样。

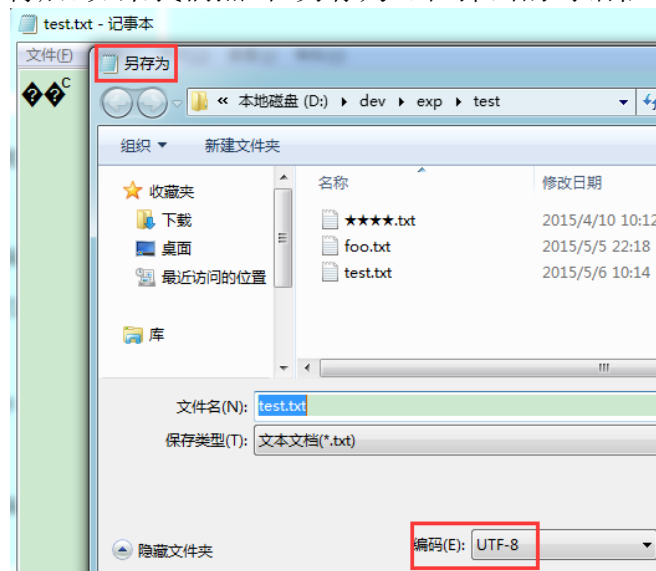


- | “联通”这一案例还真多梗。

至此，冤有头，债有主，一切都水落石出。最终我们的猜测被证实。

直接证据！

其实，在变成怪怪的字符后，如果我们点击“另存为”，在弹出的对话框中会发现编码成了“UTF-8”



- | 这是赤裸裸的证据，直接表明了记事本把编码误判成了 UTF-8！简直是铁证如山呀！

打破模式

删掉 test.txt，从新再建立，这次把联通的好基友“电信”也一起录入，录入“联通电信”四个字，保存并再次打开时记事本就不再抽疯了，因为“电信”两字的 GBK 编码模式与 UTF-8 不能匹配了，读者可自行验证一下，这里就不再贴图展示了。

- | 注：不要直接删除原来的乱码字符并重新录入，前面“另存为”已经表明它已经成了 UTF-8 编码，直接在原文件修改将导致以 UTF-8 编码保存。所以应该删除原文件。

有句话叫“无巧不成书”，记事本跟“联通”有仇，这其实就是一个因为样本太少而误判的典型例子。

缺省编码，ANSI 是个什么玩意

如果既没有 BOM，又无法猜测出所使用的编码，那是否就只能两眼一抹黑了昵？

- | 还好，计算机世界还有件贴心的小棉袄叫“缺省”。

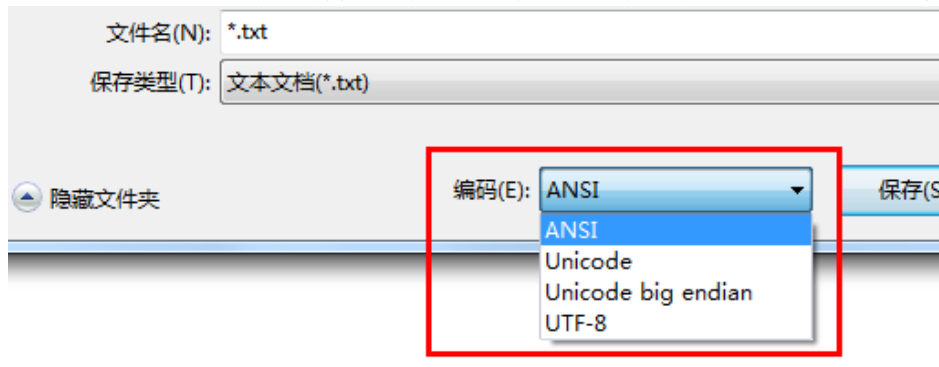
其实，当你保存任何一个文本文件时，指定一个编码是必不可少的一个步骤。

- | 与此类似，读取一个文本文件时，或者说是比如 Java 中 new 一个 Reader 字符流时，
- | 又或者是 string.getBytes 时，你其实都是需要指定一个编码的。

但很多时候，我们并没有感觉到需要这一步骤，原因就是“缺省”在为我们默默地服务。

| 缺省这玩意，怎么说它好呢？当它正常时，你好，我好，大家好。当它不正常时，你甚至不知道哪儿出错了。你过于依赖它，它很可能成为你的定时炸弹。不得不说，很多时候我们其实是抱着炸弹在击鼓传花，还玩得不亦乐乎，直到“轰”的一声，咦？头上什么时候多了个圈？

以记事本为例，当我们新建一个文件并保存时，其实是有个选项的，通常，这里会缺省地选上“ANSI”



那么这里的 ANSI 又是个什么鬼呢？

| ANSI (American National Standards Institute 美国国家标准学会)，
| 与它字面的意思并不相符，它也不是一种真正意义上的编码。

通常把它理解成平台缺省编码，它具体指代什么则通常与平台所在地区的 Windows 发行版本有关。

| 像我们这些火墙内的大陆人民，多数人用的 Windows 版本，ANSI 指的是 GBK；
| 在香港台湾地区，它可能是 Big5；在一些欧洲地区，它则可能是 ISO-8859-1。

除了 ANSI 之外，在这里还有其它的选项。

| 其实在这里短短的一个下拉列表，处处都是坑呀，说多了都是泪。
| 1. ANSI，前面已说，就不说了。
| 2. Unicode。其实是 UTF-16，具体地讲是 UTF-16 little endian (UTF-16 LE)。这个缺省为 Little Endian 也仅是微软平台的缺省。其它平台未必是如此。
| 3. Unicode big endian。与之前类似，就是 UTF-16 big endian (UTF-16 BE)。Unicode 现在的含义太宽泛，可以指 Unicode 字符集，可以指 Unicode 码点，也可以指整个 Unicode 标准。现在看来，把 UTF-16 继续叫成 Unicode 实在是很坑爹，除了容易引发误解，我还真没想到它还能有什么其它好处~
| 4. UTF-8。其实是“带 BOM 的 UTF-8”，而真正推荐的缺省做法是“不带 BOM”。
| 微软就是任性！

还需要注意的是，不同的操作系统对于缺省有不同的策略。

| 比如现在很多的 Linux 的操作系统都把 UTF-8 当成了缺省的编码，
| 无论你在什么地区都是如此。这对于减少混乱还是有帮助的。

因为文件内容没有编码的信息，各个系统平台对于缺省的规定又各不相同，种种情况导致了乱码问题层出不穷，下一篇，将探讨引入编码信息的一些实践。

第三章 引入编码信息的一些实践

原文地址：<http://my.oschina.net/goldenshaw/blog/418150>

摘要：介绍了两种引入编码信息的实践，变相引入及外部指定

前面说到，文本文件中没有编码信息，导致了各种混乱，那么，最关键的就是要指定好所用的编码信息。具体地讲，有以下一些途径。

变相引入

什么是变相引入呢？其实本质与前面提到的一些“文件头”信息是类似的。

xml

我们来看看 xml 文件的例子，你通常能在最开始看到这样的一行：

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

那么这里面，encoding 指明的就是所用编码的信息了。

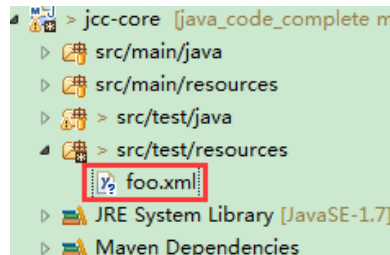
可是，等等!! 为了得到这一编码信息，我得先读取这一文件；可要正确读取文件，我又要先知道编码信息！

| 这成了一个鸡生蛋，蛋生鸡，又或者说是先有鸡还是先有蛋的问题了。

怎么破呢？考虑这一行信息所有字符都是 ASCII 中的字符，那么我们可以先使用最基础的 ASCII 去读取它开头的一些信息，获取到这一编码信息后，再次用这一编码去读取文件即可。

| ASCII 可谓是这样一个始祖鸟或者始祖蛋一样的存在。

可以动手做些实验，先建立一个 xml 文件，比如就叫 foo.xml



内容如下：

```
1| <?xml version="1.0" encoding="UTF-8"?>
2| <foo>向我开炮</foo>
```

然后初步测试读取编码信息

```
1| package org.jcc.core.encode;
2| import static org.assertj.core.api.Assertions.assertThat;
3| import java.io.File;
4| import java.nio.charset.StandardCharsets;
5| import java.util.regex.Matcher;
6| import java.util.regex.Pattern;
7| import org.apache.commons.io.FileUtils;
8| import org.junit.Test;
9| public class EncodingDetectTest {
10| @Test
11| public void testEncodingDetect() throws Exception {
12|     File foo = FileUtils.toFile(getClass().getResource("/foo.xml"));
13|     //以 ASCII 方式读取文件
14|     String content = FileUtils.readFileToString(foo, StandardCharsets.US_ASCII);
15|     // 匹配到首行，并用 group 方式抓取编码的值
16|     Pattern headerPattern = Pattern.compile("<\\?xml[\\s\\S]*encoding=\\\"([^\"]*)\\\"\\?>");
17|     Matcher headerMatcher = headerPattern.matcher(content);
18|     assertThat(headerMatcher.find()).isTrue();
19|     assertThat(headerMatcher.group(1)).isEqualTo("UTF-8");
20|     // 匹配 foo 节点中的内容 “向我开炮”
21|     Pattern fooPattern = Pattern.compile("<foo>([\\s\\S]*)</foo>");
22|     Matcher fooMatcher = fooPattern.matcher(content);
23|     assertThat(fooMatcher.find()).isTrue();
24|     // 四个 UTF-8 字符，每个三字节，共 12 字节。
25|     // 由于最高位都为 1，都不是有效的 ASCII 字节，最终被替换成了 12 个 ? (见乱码探源 2 中的介绍)
26|     assertThat(fooMatcher.group(1)).isEqualTo("????????????");
27| }
28| }
```

| 注：仅为演示用，就写得比较粗糙了。比如直接就把全部内容读取上来了，
| 精细一点应该是读取一行或者读取到所需信息就行了。正则表达式也还可以写得更严谨些。

之后就可以进一步测试了：

```
1| @Test
2| public void testRereadUsingDetectedEncoding() throws Exception {
3|     File foo = FileUtils.toFile(getClass().getResource("/foo.xml"));
4|     String encoding = getXmlEncoding(foo);
5|     // 使用检测到的编码再次读取文件
6|     String content = FileUtils.readFileToString(foo, encoding);
7|     // 这次，内容正确了。
8|     assertEquals(getTextInFooNode(content)).isEqualTo("向我开炮");
9| }
10|
11| private String getXmlEncoding(File foo) throws Exception {
12|     String content = FileUtils.readFileToString(foo, StandardCharsets.US_ASCII);
13|     Pattern headerPattern = Pattern.compile("<\\?xml[\\s\\S]*encoding=\"([^\"]*)\"\\?>");
14|     Matcher headerMatcher = headerPattern.matcher(content);
15|     headerMatcher.find();
16|     return headerMatcher.group(1);
17| }
18|
19| private String getTextInFooNode(String content) {
20|     Pattern fooPattern = Pattern.compile("<foo>([\\s\\S]*)</foo>");
21|     Matcher fooMatcher = fooPattern.matcher(content);
22|     fooMatcher.find();
23|     return fooMatcher.group(1);
24| }
```

这次内容正常了，表明我们的策略是可行的。

html, jsp

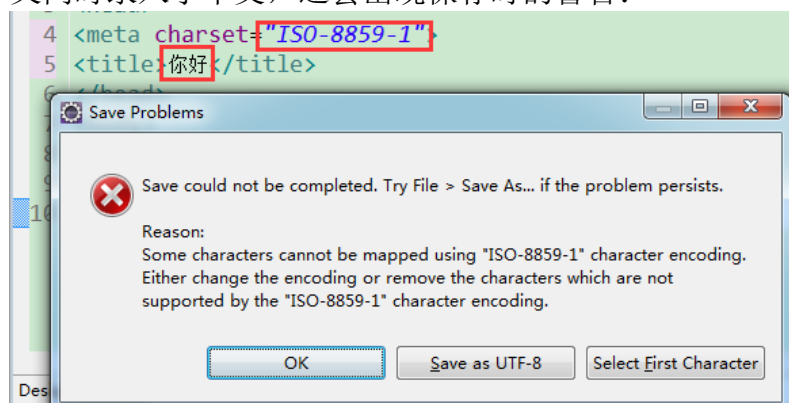
像 Html 文件也常常会这样去引入一些编码的信息，如在 header 里常会包括以下元信息：

```
| <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

又或者是像这样：

```
| <meta charset="UTF-8">
```

智能一点的文本编辑器还会根据这一信息来作为保存时的编码。比如在 Eclipse 中，如果设定了用 ISO-8859-1 编码，又同时录入了中文，还会出现保存时的警告：



自然，像记事本这样傻乎乎的编辑器就没有这么贴心了。这时，保持宣称编码与实际保存用的编码一致就是源文件作者的责任了，否则可能不但没有帮助还会误导编辑器。

ruby, python

像 ruby, python 之类的语言有时会在文件头加上如下声明：

```
| # -*- coding: utf-8 -*-
```

那么，这也算是变相引入的编码信息。自然，这需要相应的源码编辑器及编译（解释）器的支持。

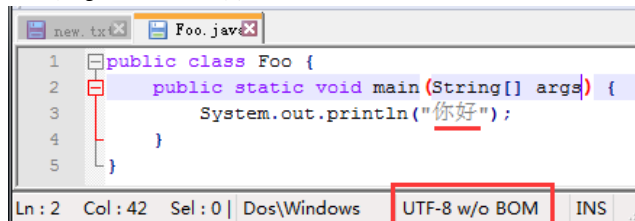
但是，像 java 这样的语言似乎没有这样的约定，那么要怎样才能尽可能避免出错呢？

外部指定

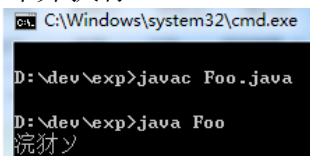
既然没有编码信息，又不打算用变相的方式指定，那么靠谱的方式就是外部显式指定了。

假如没有编码信息？

假如我们用 UTF-8 编码一个 java 源文件：



然后在 cmd 下用 javac 命令手动编译并执行：



我们发现乱码了，没有输出“你好”，而是三个怪字。原因实际上就是 javac 编译器用了缺省的编码，在 Windows 平台，也就是 GBK 去读取源文件。

- | “你好”两个字按 UTF-8 一个字三个字节，总共 6 个字节，
- | 而按 GBK 去解码，则两字节一个字，最终成为三个字。
- | 注：也即生成的 class 文件就已经是有缺陷的了。

明确引入编码参数

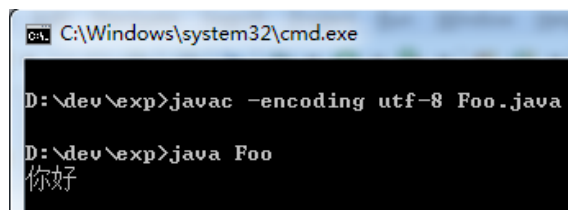
纠正的方法也很简单，就是在编译时显式指定所用的编码：

- | `javac -encoding utf-8 Foo.java`

在加了“encoding”参数后再编译，就能正确的读取源文件从而生成正确的 class 文件，

- | 注：如果你观察一下新生成的 class 文件，会比原来的小 3 个字节。
- | 这与 class 文件中所使用的“modified UTF-8”编码方式有关，
- | 可参考前面乱码探源 1 中的介绍。

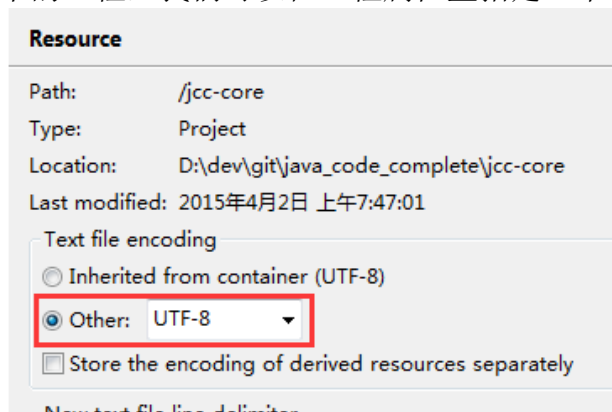
再次执行，就正常了：



在工程中指定

每次编译时都要去指定这一编码是件很繁琐的事，通常是对一整个工程在一开始就设置一个明确的编码。

比如对于一个 Eclipse 下的工程，我们可以在工程属性里指定一个编码，比如用 UTF-8：



这样之后，新建的文本文件如各种源代码文件都会使用这一编码。而当要编译时，也会使用这一编码去读取源文件。

当然，如果我们从外部引入一些文件，编码是不会自动转换的。

- | 比如引入一些 css 文件，话说天下 css 一大抄，你可能是从某网站直接抓取来的，
- | 而很多网站由于历史等原因可能还是用 GBK 等编码。

这时你需要手工转换一下编码，或者用一些批量转换的工具（如果数量很多的话）

- | 手工转的话，比如可以在记事本中先正确打开它，再拷贝到工程中的一个新建文件再保存。
- | 注：编辑时，内容在内存中都是转换成了统一的编码（在 Windows 下，就是 UTF-16），
- | 所以不同编码的文件间互相拷贝也是 OK 的，只是保存时才再次转换成相应的编码。

在构建文件中指定

也可以在构建文件中指定源文件的编码，比如 java 中用 maven 时可以这样指定：

```
1| <project>
2|   // ...
3|   <properties>
4|       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
5|   </properties>
6|   // ...
7| </project>
```

如果你用的是 ant 或者 gradle 之类的，也可自行查查文档要如何设置。至于其它语言平台的构建平台，如 grunt，make 之类，读者可自行去了解。

总而言之，越是明确地设置了编码，才越能避免混乱的出现。

在下一篇，我们再谈下在内存中的编码及相关的 string，字节流及字符流的话题。

第四章 深入理解文本在内存中的编码(1)

原文地址：<http://my.oschina.net/goldenshaw/blog/470946>

摘要：文本在内存中的编码以及 String 类型的本质。

让我们从一个故事开始说起。话说北大是很有哲学传统的，当你准备踏进北大校门时，连门卫都会连问你三个终极哲学问题：

- | 你是谁？你从哪里来？你要到哪里去？

那么这与我们的问题又有何关系呢？我觉得理解内存中的编码的关键在于理解 String 类型，因此我们也来探讨一下 String 的前世今生：String 是谁（什么）？String 从哪里来？String 到哪里去？

当我们能够清晰地回答这三个终极问题时，对文本在内存中的编码也算理解得差不多了。

- | 注：文中将用 Java 平台为例来探讨这些问题。

String 是什么？

要回答这个问题，源码当然是最好的参考。

字符序列(CharSequence)

如果看 String 类型的声明：

```
1| public final class String
2|     implements java.io.Serializable, Comparable<String>, CharSequence {
3|     private final char value[];
4|     // ...
5| }
```

可以看到它实现了所谓的 CharSequence 接口，所以它是一个 char 序列，内部实质是一个 char

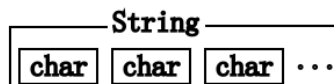
数组。

- | 也即上述代码中的” char value[] “,
- | 也许你觉得” char[] value “的写法更习惯一些, 两者是等价的

如果再看 String 的 length 方法, 事实就更清楚了, 实际上取的是 char 数组的长度:

```
1| public final class String
2|     implements java.io.Serializable, Comparable<String>, CharSequence {
3|     private final char value[];
4|     // ...
5|     public int length() {
6|         return value.length;
7|     }
8| }
```

到现在为止, 可以这样看 String:

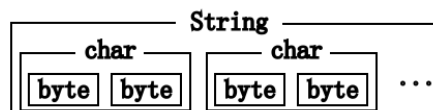


char

现在新的问题是: 什么是 char 呢? Java 中的 char 是一种基础的数据类型, 用于表示字符, 长度为 16 位。

- | 可以把 char 看作是无符号的 16 位短整型。

一个 byte 是 8 位, 那么一个 char 就相当于两个 byte 了, 所以也可以把 String 视为 byte 数组。(这是毫无疑问的, 事实上整个内存就是一个大大的 byte 数组)



那么一个 String 在内存中总是占据偶数个字节, 具体地说是占用 $\text{length()} \times 2$ 个字节。

- | 当然, 单独地拿出 String 中的一个 byte 出来是没有意义的,
- | 你总是需要两个两个一起地操作它们, 但这并不妨碍我们把它看成 byte 数组,
- | 它可以说是有点特殊的 byte 数组。

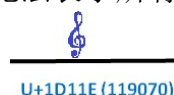
容量问题

显然, 由于 char 是 16 位固定长度, 它的容量总是有限的, 上限是 $2^{16}=65536$, 能表示 0-65535 (0x0000-0xFFFF)。

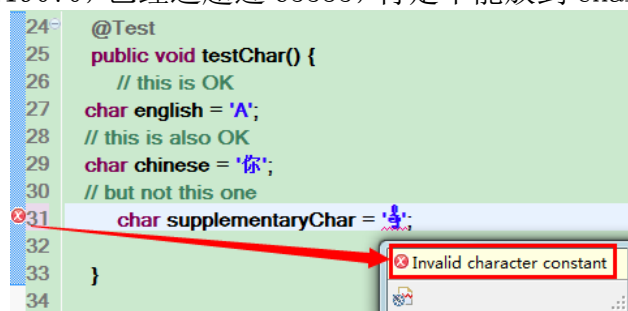
即便满打满算它也只能表示 6 万多个不同字符而已, 另一方面, Unicode 规划的字符空间高达 100 万以上, 最新版本已经定义的字符也超过了 10 万。

- | 规划的码点范围具体为 U+0000-U+10FFFF。
- | char 能表示前面的 U+0000-U+FFFF, 对于 U+10000-U+10FFFF 则无能为力。

所以一个显然的事实就是单个的 char 无法表示所有的字符。比如下面的这个音乐符:



它的十进制的码点为 119070, 已经远超过 65535, 肯定不能放到 char 里, 实验也可证实这一点:



萝卜太多, 坑位不够, 怎么办呢?

解决方案

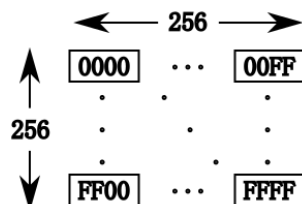
一种方式自然是对 char 进行扩展，比如弄成 32 位的，不过这样会造成很大的内存浪费。

另一种方式就是对后面的那些字符使用两个 char 来表示，也即是所谓的代理对方式，需要注意的是不能跟单个 char 表示的字符冲突。

Java 采用的就是这样方式，其后果是使得 char 无法与”抽象的字符“这一概念划上等号，一一对应的关系被打破了。

我们具体来看下是怎么做的。

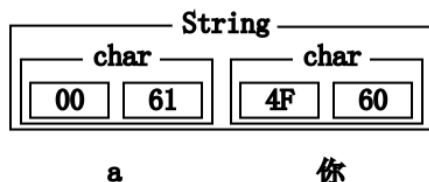
首先 char 有 $256 \times 256 = 65536$ 个空间：



常用的字符都可以在这个空间内表示，包括绝大多数的汉字。

| 比如“a”分配到的编码是“0061”，而“你”分配到的是“4F60”。

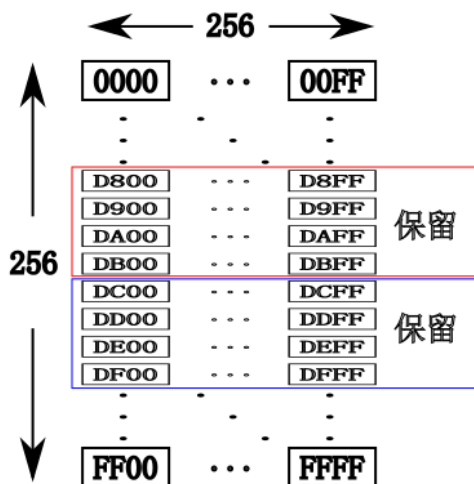
那么一个字符串，比如“a 你”就有两个 char，内存中占 4 个字节：



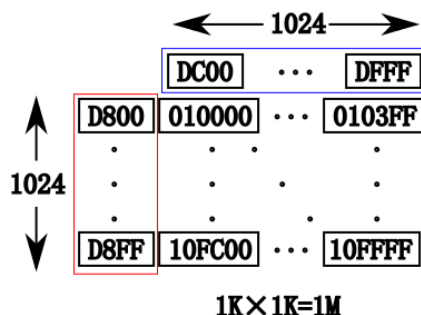
然后对于那个音乐符而言，它的码点为 U+1D11E，有 5 位，当然不能简单直接地分成 0001 和 D11E 两部分。

| 这样会与 U+0001 和 U+D11E 冲突。

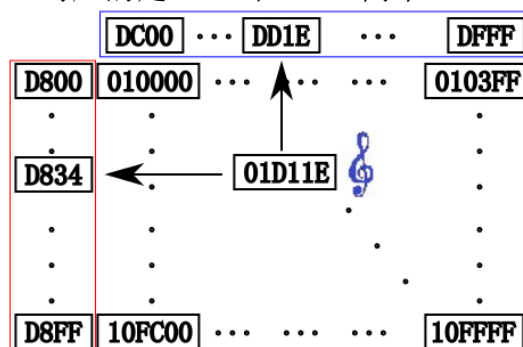
所以首先要保留一些 char，它们单个而言不代表任何抽象的字符，具体地说保留了 D800-DFFF 共 2048 个位置：



然后横竖弄成一张表，能够形成 100 多万种组合（ $1K=1024$ ）：

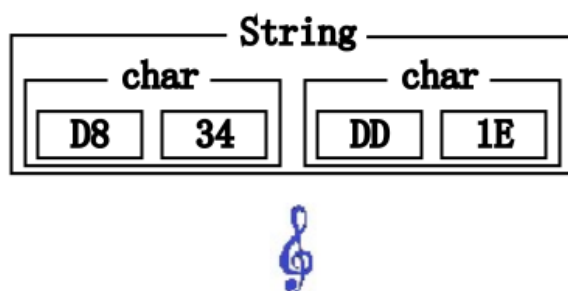


在这种表示方式下，U+1D11E 对应的是 D834 和 DD1E 两个 char：



| 具体的转换方式可见：[深入图解字符集与字符集编码（四）——Unicode](#)

我们就用这两个 char 一起来表示这个字符。这个字符无法放到单个的 char 中，但它可以放到 String 中，因为 String 是 char 数组。



综述

以上其实就是 UTF-16 的编码方式。你经常能听到这样的说法，比如：Java 平台在内存中使用 Unicode 编码。这其实说得很笼统，让我们把它说得更具体一些：

| [Java 中的 String 类型在内存中使用 UTF-16 编码。](#)

String 以 char 作为它的构成单元，这样一个 16 位的 char 也称为 UTF-16 编码的一个代码单元 (code unit)。

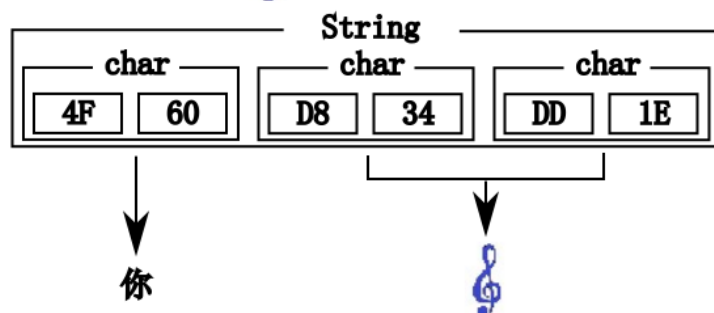
通常，一个 char 对应一个抽象的字符，但也可能需要两个 char 构成一个所谓的代理对才能表示一个抽象的字符。

所以这也导致了一些尴尬的情况，对于一些抽象字符它的长度是 2。

```
"a".length() = 1;  
"你".length() = 1;  
"你".length() = 2;
```

这与我们的直觉不符，又如下面的情况：

```
"你你".length() = 3;
```



两个抽象字符，内部为 3 个 char，所以长度是 3，在内存中则占据了 6 个字节。你可能不是很喜欢这样的 String 类型，但事实就是这样。

| 另可见[深入图解字符集与字符集编码（五）——代码单元及 length 方法](#)

其它选择

自然，你有很多的选择。如果你自己去实现一个语言平台，你当然也可以选择一个其它的编码，比如 UTF-8，甚至是 UTF-32 作为 String 的内部编码。

- | 考虑到 UTF-32 用四字节表示一个字符，通常一个 int 类型也是 4 字节，
- | 那么这种方式几乎可以认为是用一个 int 数组来保存字符。

明白了 String 是什么之后，在下一篇再继续探讨 [String 从哪里来的问题](#)。我们将深入探讨 String 的构造，字节流和字符流以及编码间的转换等问题。

第五章 深入理解文本在内存中的编码(2)

原文地址: <http://my.oschina.net/goldenshaw/blog/470948>

摘要: 深入探讨了 String 的构造，编码间的转换以及字节流，字符流等。

在前面我们探讨了 String 是什么的问题，现在来看 String 从哪来的问题。

String 从哪里来?

所谓从哪里来也可以看作是 String 的构造问题，因此我们会从 String 的构造函数说起。

String 的构造函数

在前面我们知道 String 的内部就是 char[]，因此它可以根据一组 char[] 来构建，String 中有这样的构造函数：

```
| public String(char value[]) {}
```

那么 char[] 又从何而来呢？char 的底层是 byte，String 从根本上来讲还是字节序列，而一个文本文件从根本上来讲它也是字节序列，那是不是直接把一个文本文件按字节读取上来就成了一个 String 呢？

答案是否定的。因为我们知道 String 不但是 byte[]，而且它是一个有特定编码的 byte[]，具体为 UTF-16。

而一个文本文件的字节序列有它自己特定的编码，当然它也可能是 UTF-16，但更可能是如 UTF-8 或者是 GBK 之类的，所以通常要涉及编码间的一个转换过程。我们来看下通过字节序列来构造 String 的几种方式：

```
| public String(byte bytes[]) {}  
| public String(byte bytes[], String charsetName) throws UnsupportedEncodingException {}  
| public String(byte bytes[], Charset charset) {}
```

第一个只有 byte[] 参数的构造函数实质上会使用缺省编码；而剩余的两种方式没有本质的区别。

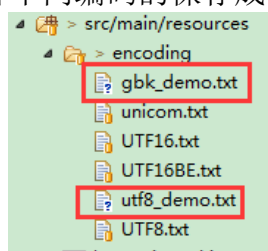
- | 后两种方式的差别在于第二个参数是用更加安全的 Charset 类型还是没那么安全的 String 类型来指明编码。

实质上可以概括为一种构造方式：也即是通过一个 byte[] 和一个编码来构造一个 String。（没有指定则使用缺省）

- | 由于历史的原因，这里沿用了 charset 这种叫法，更加准确的说法是 encoding。
- | 可参见之前的[深入图解字符集与字符集编码（一）——charset vs encoding](#)；

具体示例

录入以下内容“hi 你好”，并以两种不同编码的保存成两个不同文件：



那么，两种字节序列是有些不同的，当然，两个英文字母是相同的。

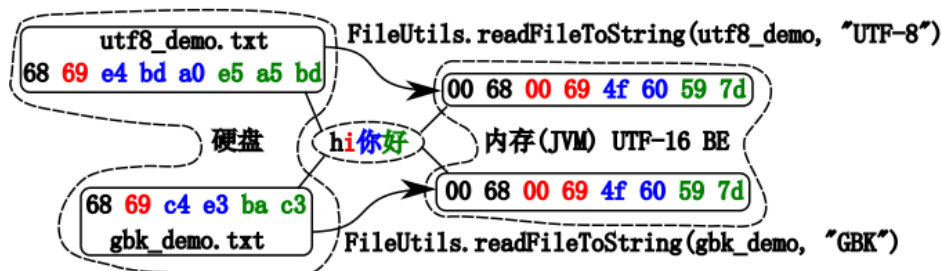
notepad_temp.txt	utf8_demo.txt	gbk_demo.txt	new					
Address	0	1	2	3	4	5	6	7
00000000	68	69	e4	bd	a0	e5	a5	bd

notepad_temp.txt	utf8_demo.txt	gbk_demo.txt	new					
Address	0	1	2	3	4	5	6	7
00000000	68	69	c4	e3	ba	c3		

那么我们如何把它们读取并转换成内存中的 String 呢？当然我们可以用一些工具类，比如 apache common 中的一些：

```
1| @Test
2| public void testReadGBK() throws Exception {
3|     File gbk_demo = FileUtils.toFile(getClass().getResource("/encoding/gbk_demo.txt"));
4|     String content = FileUtils.readFileToString(gbk_demo, "GBK");
5|     assertEquals("hi 你好");
6|     assertEquals(content.length(), 4);
7| }
8|
9| @Test
10| public void testReadUTF8() throws Exception {
11|     File utf8_demo = FileUtils.toFile(getClass().getResource("/encoding/utf8_demo.txt"));
12|     String content = FileUtils.readFileToString(utf8_demo, "UTF-8");
13|     assertEquals("hi 你好");
14|     assertEquals(content.length(), 4);
15| }
```

在这里，file 作为 byte[]，加上我们指定的编码参数，这一参数必须与保存文件时所用的参数一致，那么构造 String 就不成问题了，下图显示了这一过程：



以上四个字节序列都是对四个抽象字符“hi 你好”的编码，转换成 string 后，特定编码统一成了 UTF-16 的编码。

现在，如果我们要进行比较呀，拼接呀，都方便了。如果只是把两个文件作为原始的 byte[] 直接读取上来，那么我们甚至连一些很简单的问题，比如“在抽象的字符层面，这两个文件的内容是不是相同的”，都没办法去回答。

从这个角度来看，string 不过就是统一了编码的 byte[]。

而另一方面，我们看到，构造 string 的这一过程也就是不同编码的 byte[] 转换到统一编码的 byte[] 的过程，我们自然要问：它具体是怎么转换的呢？

转换的过程

让我们一一来分析下：

- 1. UTF-16 BE: 假如文本文件本身的编码就是 UTF-16 BE，那么自然不需要任何的转换了，逐字节拷贝到内存中就是了。

- 2. UTF-16 LE: LE 跟 BE 的差别在于字节序，因此只要每两个字节交换一下位置即可。

关于字节序跟 BOM 的话题，可见[深入图解字符集与字符集编码（七）——BOM](#)

- 3. ASCII 和 ISO_8859_1: 这两种都是单字节的编码，因此只需要前补零补成双字节即可。如上图中 68, 69 转换成 0068, 0069 那样。

• 4. UTF-8: 这是变长编码，首先按照 UTF-8 的规则分隔编码，如把 8 个字节 “68 69 e4 bd a0 e5 a5 bd” 分成 “1|1|3|3” 四个部分：

| 68 | 69 | e4 bd a0 | e5 a5 bd

然后，编码可转换成码点，然后就可以转换成 UTF-16 的编码了。

| 关于码点及 Unicode 的话题，可见[深入图解字符集与字符集编码（四）——Unicode](#)

我们来看一个具体的转换，比如字符 “你” 从 “e4 bd a0” 转换到 “4f 60” 的过程：

UTF-8	e	4	b	d	a	0
二进制	1110 0100	1011 1101	1010 0000			
保留位	1110 0100	1011 1101	1010 0000			
有效位	0100	11 1101	10 0000			
右移		0100 1111	0110 0000			
UTF-16		4	f	6	0	

| 真正的转换代码，未必真要转换到码点，可能就是一些移位操作，移完了就算转换好了。

| 如果涉及增补字符，这个过程还会更加复杂

• 5. GBK: GBK 也是变长编码，首先还是按照 GBK 的编码规则将字节分隔，把 “68 69 c4 e3 ba c3” 分成 “1|1|2|2” 四个部分。

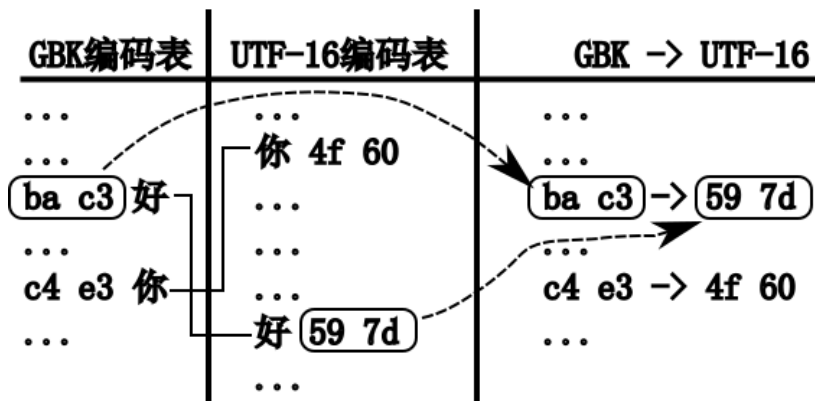
| 68 | 69 | c4 e3 | ba c3

之后，比如对于字符 “好” 来说，编码是如何从 GBK 的 “ba c3” 变成 UTF-16 的 “59 7d” 呢？

| 这一下，我们没法简单地用有限的一条或几条规则就能完成转换了，

| 因为不像 Unicode 的几种编码之间有码点这一桥梁。

这时候只能依靠查表这种原始的方式。首先需要建立一个对应表，要把一个 GBK 编码表和一个 UTF-16 编码表放一块，以字符为纽带，一一找到那些对应，如下图：



| 很明显，由于有众多的字符，要建立这样一个对应表还是蛮大的工作量的。自然，曾经有
| 那么些苦逼的程序员在那里把这些关系一一建立了起来。不过，好在这些工作只须做一次
| 就行了。如果他们藏着掖着，我们就向他们宣扬 “开源” 精神，等他们拿出来共享后，我
| 们再发挥 “拿来主义” 精神。

那么，有了上图中最右边的表之后，转换就能进行了。

| 当然，我们只需要扔给 String 的构造函数一个 byte[]，以及告诉它这是 “GBK” 编码即可，
| 怎么去查不用我们操心，那是 JVM 的事，当然它可能也没有这样的表，它也许只是转手又
| 委托给操作系统去转换。

不支持的情况

如果我们看前面 String 构造函数的声明，有一个会抛出 UnsupportedOperationException（不支持的编码）的异常。如果一个比较小众的编码，JVM 没有转换表，操作系统也没有转换表，String 的构建过程就没法进行下去了，这时只好抛个异常，罢工了。

| 当然了，很多时候抛了异常也许只是粗心把编码写错了而已。

至此，我们基本明白了 String 从哪里来的问题，它从其它编码的 byte[] 转换而来，它自身不过也是某种编码的 byte[] 而已。

字节流与字符流

如果你此时认为前面的 FileUtils.readFileToString(gbk_demo, "GBK") 就是读取到一堆的 byte[]，然后调用构造函数 String(byte[], encoding) 来生成 String，不过，实际过程并不是这样的，通常的方式是使用所谓的“字符流”。

那么什么是字符流呢？它是为专门方便我们读取（以及写入）文本文件而建立的一种抽象。

| 文件始终是字节流，这一点对于文本文件自然也是成立的，你始终可以按照字节流并结合编码的方式去处理文本文件。不过，另外一种更方便处理文本文件的方式是把它们看成是某种抽象的“字符流”。

设想一个很大的文本文件，我们通常不会说一下就把它全部读取上来并指定对应编码来构建出一个 String，更可能的需求是要一个一个字符的读取。

| 比如对于前述的“hi 你好”四个字符，我们希望说，把“h”读取上来，再把“i”读上来，再读“你”，再读“好”，如此这般，至于编码怎么分隔呀，转换呀，我们都不关心。

Reader 跟 Writer 是字符流的最基本抽象，分别用于读跟写，我们先看 Reader。可以用以下方式尝试依次读取字符：

```
1| public void testReader() {
2|     File gbk_demo = FileUtils.toFile(getClass().getResource("/encoding/gbk_demo.txt"));
3|     Reader reader = null;
4|     try {
5|         InputStream is = new FileInputStream(gbk_demo);
6|         reader = new InputStreamReader(is, "GBK");
7|
8|         char c = (char) reader.read();
9|         assertEquals('h', c);
10|        c = (char) reader.read();
11|        assertEquals('i', c);
12|        c = (char) reader.read();
13|        assertEquals('你', c);
14|        c = (char) reader.read();
15|        assertEquals('好', c);
16|    } catch (FileNotFoundException e) {
17|        e.printStackTrace();
18|    } catch (IOException e) {
19|        e.printStackTrace();
20|    } finally {
21|        IOUtils.closeQuietly(reader);
22|    }
23| }
```

| 这里，read() 方法返回是一个 int，把它转换成 char 即可，另一种方式是 read(char cbuf[]) 直接把字符读取到一个 char[]，之后，如果有必要，可以用这个 char[] 去构建 String，因为我们知道 String 其实就是一个 char[]。

很显然，一个 Reader 不但要构建在一个字节流 (InputStream) 基础上，而且它与具体的编码也是息息相关的。

| 编码用于指导分隔底层字节数组，然后再转成 UTF-16 编码的字符。

那么这其实与 String 的构造没有本质的区别，事实上也是如此，一个字符流实质所做的工作依旧是把一种编码的 byte[] 转换成 UTF-16 编码的 byte[]。

| 而那些需要两个 char 才能表示的增补字符，如前面提到的音乐符，事实上你要 read 两次。

| 所以字符流这种抽象还是要打个折扣的，准确地讲是 char 流，而非真正的抽象的字符。

关于 String 从哪里来的话题，就讲到这里，下一篇再继续探讨它[到哪里去的问题](#)。

第六章 深入理解文本在内存中的编码(3)

原文地址: <http://my.oschina.net/goldenshaw/blog/471370>

摘要: 探讨了 String 到 byte[] 的转换, 并结合之前的 new String 作了综合分析。

先讲个小故事, 虽然跟主题有点不太相关哈:

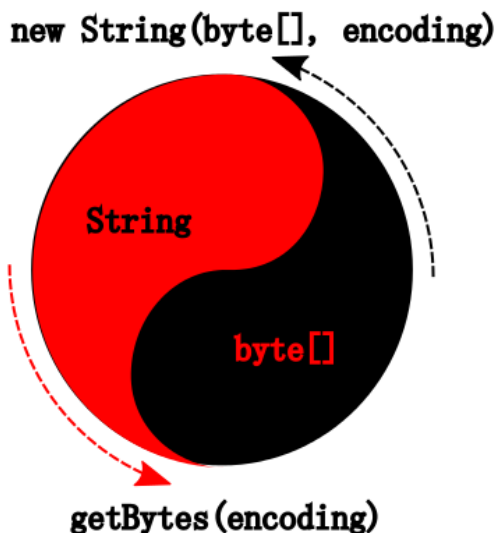
| 唐朝诗人李绅, 身为官员, 脾气暴躁, 瞧不起信教的, 尤其鄙视装逼之僧人, 动不动就对他们拳脚相加。曾扬言: “我可以接见他们, 要能答出来还好, 要是答不出来, 我弄死他!”
| 有一回一个和尚来跟他宣传因果报应, 李绅问: “阿师从哪里来, 到哪里去呢?” 僧答: “贫僧从来处来, 到去处去。” 李绅当时就急了, 撸起袖子, 亮出了手腕: “我去年买了个表!”
| 来自知乎问答 “古人是如何「装逼」的?”, 略有改动。

String 到哪里去?

有了前面僧人的教训, 在这里就不故弄玄虚了, 应该说 String 的去处还是蛮确定的, 那就是到 byte[] 中去, 方式就是通过 getBytes 这一方法。

new String 与 getBytes

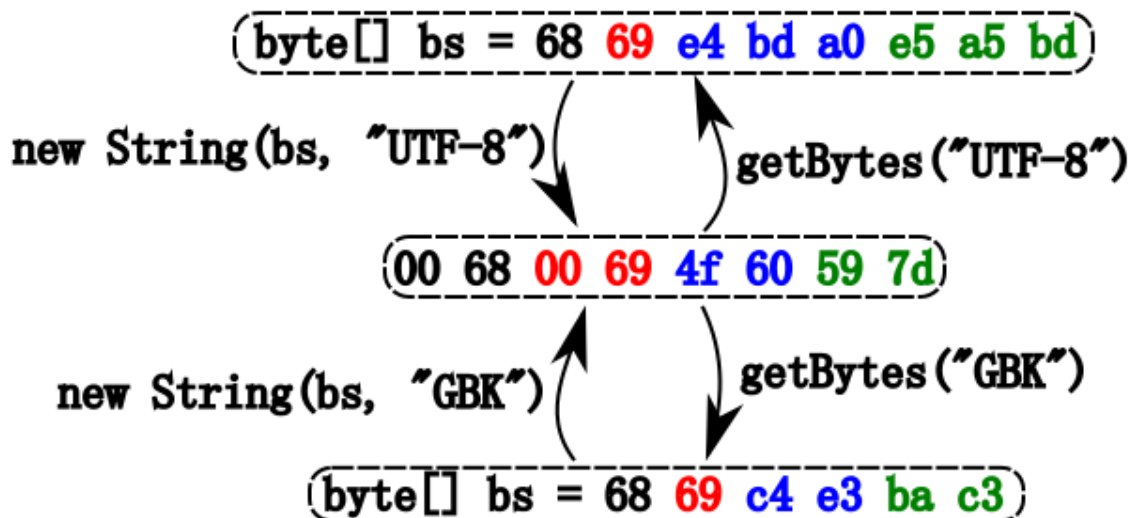
如果说 new String(byte[], encoding) 是从 byte[] 到 String 的过程, 那么 getBytes(encoding) 则正好与之相反: 它是从 String 到 byte[] 的过程。



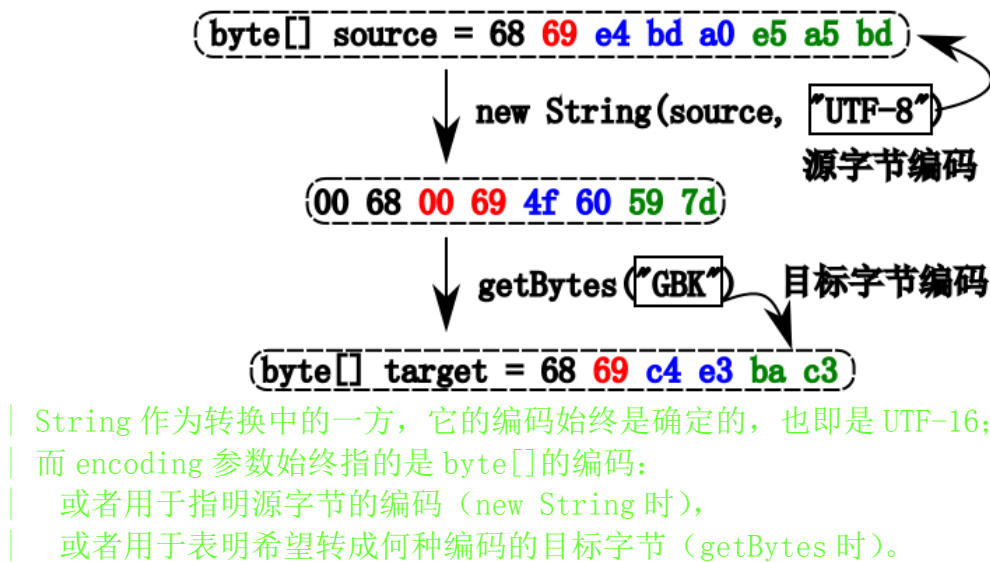
| 或许我们应该说, 它从去处来, 又到来处去。

编码的逆转

显然, 我们一直在说, String 也不过是一堆 byte, getBytes 的过程不过是 UTF-16 编码的 byte[] 再转回去其它编码的 byte[] 的过程。无论是 new String 还是 getBytes, 不过都是在玩编码的转换而已。

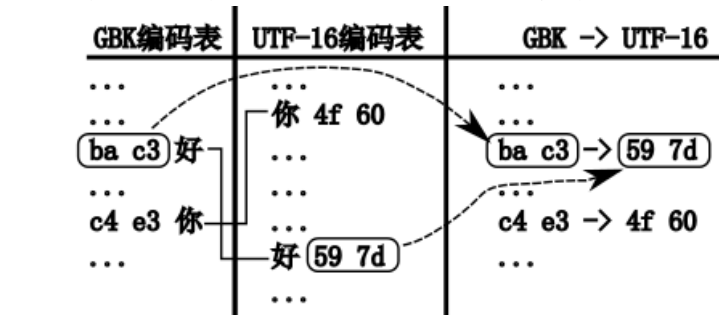


从上图中可以看出 String 作为桥梁，可以把一种编码的字节转换成另一种编码的字节。比如把一串的 UTF-8 编码的字节转换成 GBK 编码的字节的。

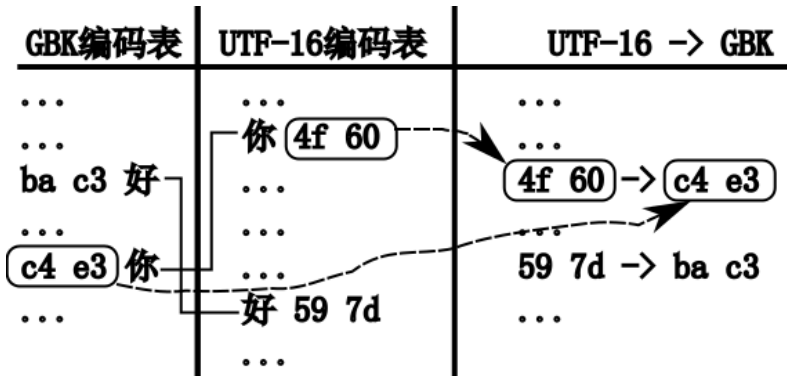


具体转换过程

以 GBK 为例，既然前面说，GBK 转 UTF-16 可以通过查表实现：



那么 UTF-16 转 GBK，我们只需要反查那张表即可。当然，考虑到效率的问题，我们可能需要另一张按 UTF-16 编码排序的表：



| 当然了，这些都不需要我们去操心的。

至于 UTF-16 转成 UTF-8，依旧可以通过码点这一桥梁来进行。

| 这里就不再演示了，与前面码点转 UTF-8 非常类似：

U+4F60

0100 1111 0110 0000 16 位二进制形式

0100 1111 0110 0000 按 4+6+6 位分组

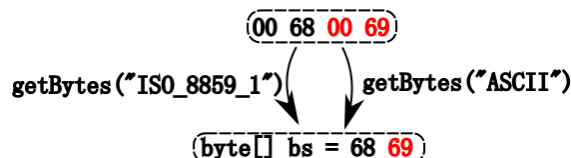
1110XXXX 10XXXXXX 10XXXXXX UTF-8 三字节模板

11100100 10111101 10100000 替换有效编码位

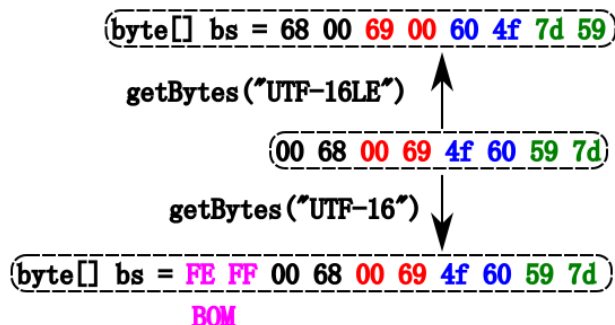
E4 BD A0 按字节重新转换成 16 进制

| 可参见[深入图解字符集与字符集编码（四）——Unicode](#)

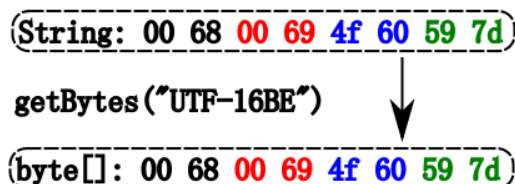
剩下的如转 ISO_8859_1 以及 ASCII 之类的,那就更简单了。如果一段String表示的是 ISO_8859_1 或者 ASCII 中的字符,显然里面每个 char 的高位都是 00,因此只要把这些没用的 00 掐掉就行了。



如果调用 `getBytes("UTF-16")` 呢? 那就不存在转换了,相当于复制了一遍,不过要注意它会带上 BOM, 除非明确指明了端序。



如果调用 `getBytes("UTF-16BE")`, 那么内存中就会出现两组一模一样的字节了。



但是这两者还是有本质的区别的, 原因就在于指向这两者的引用所代表的类型的不同。

| 类型赋予了一串 byte 丰富的内涵, 决定了我们怎么去解释它。

String 是一种有趣得多的类型, 它有明确的编码, 还有丰富多样的方法与之绑定。

而另一方面, byte[] 则要原始单调乏味得多。严格地说, byte[] 只是一堆字节而已, 就编码这个问题而言, 它本身没有与任何编码绑定。

| 当然, 字节间的特征也许能让你断言这不是某种编码,

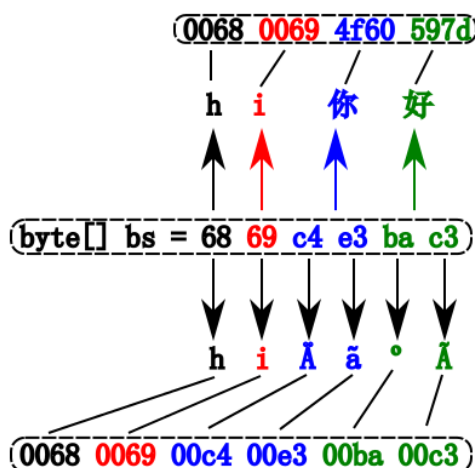
| 但你却不能肯定地说, 一串字节一定是某种编码。

多种解释

我们来看个具体例子, 还是拿前面说到的那串 GBK 编码的 byte 来说吧: 68 69 c4 e3 ba c3。

之所以说它是 GBK, 那是因为我们用 GBK 编码保存文件得到的它。但如果内存中有一段与之一模一样的 byte[], 难道你能说它一定是 GBK 编码吗?

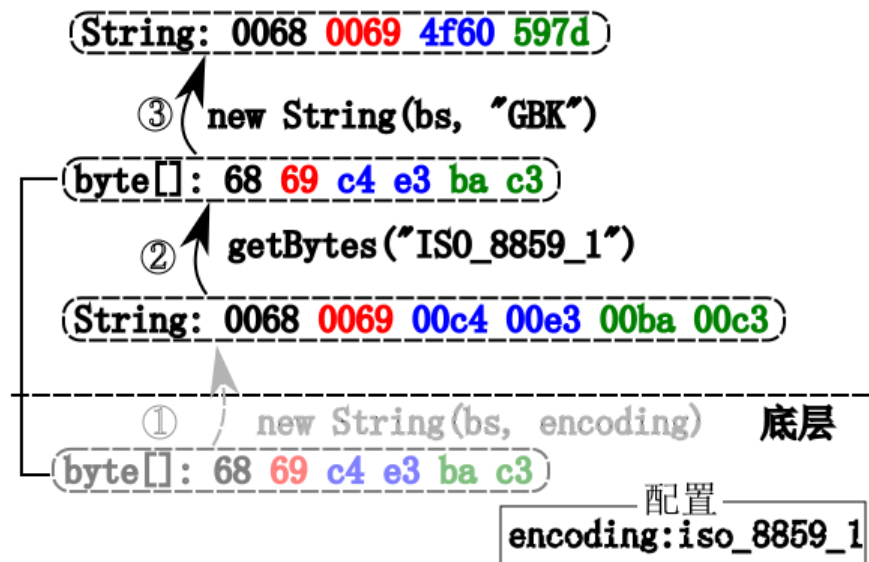
首先可以确定它不可能是 ASCII, 因为有些字节最高位是 1, 那么它有可能是 ISO_8859_1 吗? 这是有可能的。而对它的不同解释会因此在内存中生成不一样的 String:



具体的代码测试也可以反映这一点：

```
1| @Test
2| public void testReadGBKBytesAsISO_8859_1() throws Exception {
3|     File gbk_demo = FileUtils.toFile(getClass().getResource("/encoding/gbk_demo.txt"));
4|     // 当成 ISO_8859_1 来读取
5|     String content = FileUtils.readFileToString(gbk_demo, "ISO_8859_1");
6|     assertThat(content).isEqualTo("hiÃ°Ã");
7|     assertThat(content.length()).isEqualTo(6);
8| }
```

有的时候，你没有办法拿到最底层的 `byte[]`，你直接收到的就是一个 `String`，而这个 `String` 是通过错误的编码构建的，如下所示：

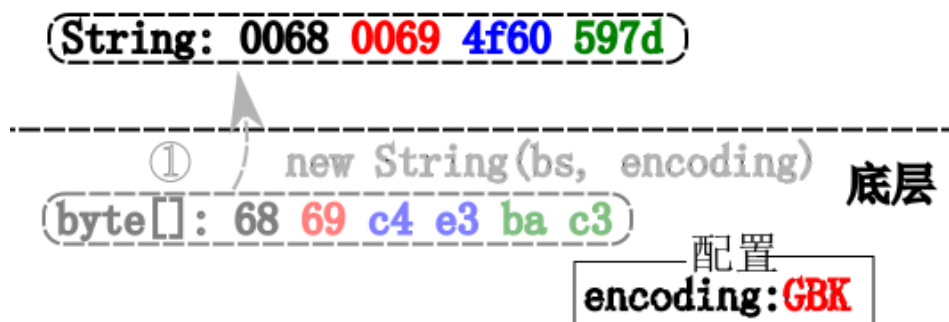


第一步不受我们控制，这时有一种 hack 的方式，也即是通过上面的第二步再度拿回原始的 `byte[]`，再通过第三步传入正确的编码再度构建出 `String`。

代码演示如下：

```
1| @Test
2| public void testISO_8859_1Hack() throws Exception {
3|     File gbk_demo = FileUtils.toFile(getClass().getResource("/encoding/gbk_demo.txt"));
4|     // 当成 ISO_8859_1 来读取
5|     String content = FileUtils.readFileToString(gbk_demo, "ISO_8859_1");
6|     assertThat(content).isEqualTo("hiÃ°Ã");
7|     assertThat(content.length()).isEqualTo(6);
8|
9|     // 再次拿到原始 byte[]，并以新的编码重新构建 String
10|    content = new String(content.getBytes("ISO_8859_1"), "GBK");
11|    assertThat(content).isEqualTo("hi 你好");
12|    assertThat(content.length()).isEqualTo(4);
13| }
```

当然，我们并不鼓励这样繁琐地转来转去，正确的姿势应该是这样的：



通过修改配置，就能一步到位得到正确的 String，不需要曲线救国。

- | 有人可能说，我并没配置过 iso_8859_1 呀，那么这可能是某种缺省编码。
- | 总之，如果你收到被疑似 iso_8859_1 错误编码的 String，
- | 那肯定是某个环节使用了这一编码。

当然，如果你工作在一个遗留系统上，还是要非常慎重地去改变这些缺省的编码配置，因为可能严重冲击到那些依赖于这些缺省设置的代码。

- | 比如上述 hack 方式可能就失效了，而你可能不知道系统到底有多少地方使用了这种 hack。

关于 String 到哪里去的问题，就探讨到这里。