1字节就是1B，  
  
1024B＝1KB  
1024KB＝1MB  
1024MB＝1GB

注意1个换行占2个字节的

1. Gbk与utf-8区别

Gbk的文字编码是双字节表示的，既不论中，英文字都是使用双字节表示的。只不过为了区分中文，将其最高位都定成1

Utf-8 对英文使用8位即一个字节，中文使用24位即3个字节来编码。

所以对于英文符较多的论坛则用utf-8更节约空间。

2 Gbk包含全部的中文字符。

Utf-8则包含全世界的用到得字符。

Gb2312是gbk的子集，gbk是gb18030的子集

Gbk包括中日韩字符的大字符集合

如果是中文的网站，推荐gb2312，gbk有时候有点问题。

为了避免所有乱码问题，应该采用utf-8，将来要支持国际化也非常方便。

GBK、GB2312等与UTF8之间都必须通过Unicode编码才能相互转换：

GBK、GB2312－－Unicode－－UTF8

UTF8－－Unicode－－GBK、GB2312

字符，字节和编码

[原创文章，转载请保留或注明出处：<http://www.regexlab.com/zh/encoding.htm>]

级别：中级

摘要：本文介绍了字符与编码的发展过程，相关概念的正确理解。举例说明了一些实际应用中，编码的实现方法。然后，本文讲述了通常对字符与编码的几种误解，由于这些误解而导致乱码产生的原因，以及消除乱码的办法。本文的内容涵盖了“中文问题”，“乱码问题”。

掌握编码问题的关键是正确地理解相关概念，编码所涉及的技术其实是很简单的。因此，阅读本文时需要慢读多想，多思考。

**引言**

“字符与编码”是一个被经常讨论的话题。即使这样，时常出现的乱码仍然困扰着大家。虽然我们有很多的办法可以用来消除乱码，但我们并不一定理解这些办法的内在原理。而有的乱码产生的原因，实际上由于底层代码本身有问题所导致的。因此，不仅是初学者会对字符编码感到模糊，有的底层开发人员同样对字符编码缺乏准确的理解。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**1. 编码问题的由来，相关概念的理解**

**1.1 字符与编码的发展**

从计算机对多国语言的支持角度看，大致可以分为三个阶段：

|  |  |  |  |
| --- | --- | --- | --- |
|  | **系统内码** | **说明** | **系统** |
| 阶段一 | ASCII | 计算机刚开始只支持英语，其它语言不能够在计算机上存储和显示。 | 英文 DOS |
| 阶段二 | ANSI编码 （本地化） | 为使计算机支持更多语言，通常使用 0x80~0xFF 范围的 2 个字节来表示 1 个字符。比如：汉字 '中' 在中文操作系统中，使用 [0xD6,0xD0] 这两个字节存储。  不同的国家和地区制定了不同的标准，由此产生了 GB2312, BIG5, JIS 等各自的编码标准。这些使用 2 个字节来代表一个字符的各种汉字延伸编码方式，称为**ANSI 编码**。在简体中文系统下，ANSI 编码代表 GB2312 编码，在日文操作系统下，ANSI 编码代表 JIS 编码。  不同 ANSI 编码之间互不兼容，当信息在国际间交流时，无法将属于两种语言的文字，存储在同一段**ANSI 编码**的文本中。 | 中文 DOS，中文 Windows 95/98，日文 Windows 95/98 |
| 阶段三 | UNICODE （国际化） | 为了使国际间信息交流更加方便，国际组织制定了 **UNICODE 字符集**，为各种语言中的每一个字符设定了统一并且唯一的数字编号，以满足跨语言、跨平台进行文本转换、处理的要求。 | Windows NT/2000/XP，Linux，Java |

字符串在内存中的存放方法：

在 ASCII 阶段，**单字节字符串**使用一个字节存放一个字符（SBCS）。比如，"Bob123" 在内存中为：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 42 | 6F | 62 | 31 | 32 | 33 | 00 |
| http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif |
| B | o | b | 1 | 2 | 3 | \0 |

在使用 ANSI 编码支持多种语言阶段，每个字符使用一个字节或多个字节来表示（MBCS），因此，这种方式存放的字符也被称作**多字节字符**。比如，"中文123" 在中文 Windows 95 内存中为7个字节，每个汉字占2个字节，每个英文和数字字符占1个字节：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D6 | D0 | CE | C4 | 31 | 32 | 33 | 00 |
| http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif | http://www.regexlab.com/images/spacer.gif |
| 中 | | 文 | | 1 | 2 | 3 | \0 |

在 UNICODE 被采用之后，计算机存放字符串时，改为存放每个字符在 UNICODE 字符集中的序号。目前计算机一般使用 2 个字节（16 位）来存放一个序号（DBCS），因此，这种方式存放的字符也被称作**宽字节字符**。比如，字符串 "中文123" 在 Windows 2000 下，内存中实际存放的是 5 个序号：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2D | 4E | 87 | 65 | 31 | 00 | 32 | 00 | 33 | 00 | 00 | 00 | ← 在 x86 CPU 中，低字节在前 |
| http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif | | http://www.regexlab.com/images/spacer.gif |
| 中 | | 文 | | 1 | | 2 | | 3 | | \0 | |  |

一共占 10 个字节。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**1.2 字符，字节，字符串**

理解编码的关键，是要把字符的概念和字节的概念理解准确。这两个概念容易混淆，我们在此做一下区分：

|  |  |  |
| --- | --- | --- |
|  | **概念描述** | **举例** |
| 字符 | 人们使用的记号，抽象意义上的一个符号。 | '1', '中', 'a', '$', '￥', …… |
| 字节 | 计算机中存储数据的单元，一个8位的二进制数，是一个很具体的存储空间。 | 0x01, 0x45, 0xFA, …… |
| ANSI 字符串 | 在内存中，如果“字符”是以 **ANSI 编码**形式存在的，一个字符可能使用一个字节或多个字节来表示，那么我们称这种字符串为 **ANSI 字符串**或者**多字节字符串**。 | "中文123" （占7字节） |
| UNICODE 字符串 | 在内存中，如果“字符”是以在 UNICODE 中的序号存在的，那么我们称这种字符串为 **UNICODE 字符串**或者**宽字节字符串**。 | L"中文123" （占10字节） |

由于不同 ANSI 编码所规定的标准是不相同的，因此，对于一个给定的**多字节字符串**，我们必须知道它采用的是哪一种编码规则，才能够知道它包含了哪些“字符”。而对于 **UNICODE 字符串**来说，不管在什么环境下，它所代表的“字符”内容总是不变的。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**1.3 字符集与编码**

各个国家和地区所制定的不同 ANSI 编码标准中，都只规定了各自语言所需的“字符”。比如：汉字标准（GB2312）中没有规定韩国语字符怎样存储。这些 ANSI 编码标准所规定的内容包含两层含义：

1. 使用哪些字符。也就是说哪些汉字，字母和符号会被收入标准中。所包含“字符”的集合就叫做“**字符集**”。
2. 规定每个“字符”分别用一个字节还是多个字节存储，用哪些字节来存储，这个规定就叫做“**编码**”。

各个国家和地区在制定编码标准的时候，“字符的集合”和“编码”一般都是同时制定的。因此，平常我们所说的“字符集”，比如：GB2312, GBK, JIS 等，除了有“字符的集合”这层含义外，同时也包含了“编码”的含义。

“**UNICODE 字符集**”包含了各种语言中使用到的所有“字符”。用来给 UNICODE 字符集编码的标准有很多种，比如：UTF-8, UTF-7, UTF-16, UnicodeLittle, UnicodeBig 等。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**1.4 常用的编码简介**

简单介绍一下常用的编码规则，为后边的章节做一个准备。在这里，我们根据编码规则的特点，把所有的编码分成三类：

|  |  |  |
| --- | --- | --- |
| **分类** | **编码标准** | **说明** |
| 单字节字符编码 | ISO-8859-1 | 最简单的编码规则，每一个字节直接作为一个 UNICODE 字符。比如，[0xD6, 0xD0] 这两个字节，通过 iso-8859-1 转化为字符串时，将直接得到 [0x00D6, 0x00D0] 两个 UNICODE 字符，即 "ÖÐ"。  反之，将 UNICODE 字符串通过 iso-8859-1 转化为字节串时，只能正常转化 0~255 范围的字符。 |
| ANSI 编码 | GB2312, BIG5, Shift\_JIS, ISO-8859-2 …… | 把 UNICODE 字符串通过 ANSI 编码转化为“字节串”时，根据各自编码的规定，一个 UNICODE 字符可能转化成一个字节或多个字节。  反之，将字节串转化成字符串时，也可能多个字节转化成一个字符。比如，[0xD6, 0xD0] 这两个字节，通过 GB2312 转化为字符串时，将得到 [0x4E2D] 一个字符，即 '中' 字。  “ANSI 编码”的特点： 1. 这些“ANSI 编码标准”都只能处理各自语言范围之内的 UNICODE 字符。 2. “UNICODE 字符”与“转换出来的字节”之间的关系是人为规定的。 |
| UNICODE 编码 | UTF-8, UTF-16, UnicodeBig …… | 与“ANSI 编码”类似的，把字符串通过 UNICODE 编码转化成“字节串”时，一个 UNICODE 字符可能转化成一个字节或多个字节。  与“ANSI 编码”不同的是： 1. 这些“UNICODE 编码”能够处理所有的 UNICODE 字符。 2. “UNICODE 字符”与“转换出来的字节”之间是可以通过计算得到的。 |

我们实际上没有必要去深究每一种编码具体把某一个字符编码成了哪几个字节，我们只需要知道“编码”的概念就是把“字符”转化成“字节”就可以了。对于“UNICODE 编码”，由于它们是可以通过计算得到的，因此，在特殊的场合，我们可以去了解某一种“UNICODE 编码”是怎样的规则。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**2. 字符与编码在程序中的实现**

**2.1 程序中的字符与字节**

在 C++ 和 Java 中，用来代表“字符”和“字节”的数据类型，以及进行编码的方法：

|  |  |  |
| --- | --- | --- |
| **类型或操作** | **C++** | **Java** |
| 字符 | wchar\_t | char |
| 字节 | char | byte |
| ANSI 字符串 | char[] | byte[] |
| UNICODE 字符串 | wchar\_t[] | String |
| 字节串→字符串 | mbstowcs(), MultiByteToWideChar() | string = new String(bytes, "encoding") |
| 字符串→字节串 | wcstombs(), WideCharToMultiByte() | bytes = string.getBytes("encoding") |

以上需要注意几点：

1. Java 中的 char 代表一个“UNICODE 字符（宽字节字符）”，而 C++ 中的 char 代表一个字节。
2. MultiByteToWideChar() 和 WideCharToMultiByte() 是 Windows API 函数。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**2.2 C++ 中相关实现方法**

声明一段字符串常量：

|  |
| --- |
| // ANSI 字符串，内容长度 7 字节 char     sz[20] = "中文123";  // UNICODE 字符串，内容长度 5 个 wchar\_t（10 字节） wchar\_t wsz[20] = L"\x4E2D\x6587\x0031\x0032\x0033"; |

UNICODE 字符串的 I/O 操作，字符与字节的转换操作：

|  |
| --- |
| // 运行时设定当前 ANSI 编码，VC 格式 setlocale(LC\_ALL, ".936");  // GCC 中格式 setlocale(LC\_ALL, "zh\_CN.GBK");  // Visual C++ 中使用小写 %s，按照 setlocale 指定编码输出到文件 // GCC 中使用大写 %S fwprintf(fp, L"%s\n", wsz);  // 把 UNICODE 字符串按照 setlocale 指定的编码转换成字节 wcstombs(sz, wsz, 20); // 把字节串按照 setlocale 指定的编码转换成 UNICODE 字符串 mbstowcs(wsz, sz, 20); |

在 Visual C++ 中，UNICODE 字符串常量有更简单的表示方法。如果源程序的编码与当前默认 ANSI 编码不符，则需要使用 #pragma setlocale，告诉编译器源程序使用的编码：

|  |
| --- |
| // 如果源程序的编码与当前默认 ANSI 编码不一致， // 则需要此行，编译时用来指明当前源程序使用的编码 #pragma setlocale(".936")  // UNICODE 字符串常量，内容长度 10 字节 wchar\_t wsz[20] = L"中文123"; |

以上需要注意 #pragma setlocale 与 setlocale(LC\_ALL, "") 的作用是不同的，#pragma setlocale 在编译时起作用，setlocale() 在运行时起作用。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**2.3 Java 中相关实现方法**

字符串类 String 中的内容是 UNICODE 字符串：

|  |
| --- |
| // Java 代码，直接写中文 String string = "中文123";  // 得到长度为 5，因为是 5 个字符 System.out.println(string.length()); |

字符串 I/O 操作，字符与字节转换操作。在 Java 包 java.io.\* 中，以“Stream”结尾的类一般是用来操作“字节串”的类，以“Reader”，“Writer”结尾的类一般是用来操作“字符串”的类。

|  |
| --- |
| // 字符串与字节串间相互转化  // 按照 GB2312 得到字节（得到多字节字符串） byte [] bytes = string.getBytes("GB2312");  // 从字节按照 GB2312 得到 UNICODE 字符串 string = new String(bytes, "GB2312");  // 要将 String 按照某种编码写入文本文件，有两种方法：  // 第一种办法：用 Stream 类写入已经按照指定编码转化好的字节串 OutputStream os = new FileOutputStream("1.txt"); os.write(bytes); os.close();  // 第二种办法：构造指定编码的 Writer 来写入字符串 Writer ow = new OutputStreamWriter(new FileOutputStream("2.txt"), "GB2312"); ow.write(string); ow.close();  /\* 最后得到的 1.txt 和 2.txt 都是 7 个字节 \*/ |

如果 java 的源程序编码与当前默认 ANSI 编码不符，则在编译的时候，需要指明一下源程序的编码。比如：

|  |
| --- |
| E:\>javac -encoding BIG5 Hello.java |

以上需要注意区分源程序的编码与 I/O 操作的编码，前者是在编译时起作用，后者是在运行时起作用。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**3. 几种误解，以及乱码产生的原因和解决办法**

**3.1 容易产生的误解**

|  |  |
| --- | --- |
|  | **对编码的误解** |
| 误解一 | 在将“字节串”转化成“UNICODE 字符串”时，比如在读取文本文件时，或者通过网络传输文本时，容易将“字节串”简单地作为**单字节字符串**，采用每“一个字节”就是“一个字符”的方法进行转化。  而实际上，在非英文的环境中，应该将“字节串”作为 ANSI 字符串，采用适当的编码来得到 UNICODE 字符串，有可能“多个字节”才能得到“一个字符”。  通常，一直在英文环境下做开发的程序员们，容易有这种误解。 |
| 误解二 | 在 DOS，Windows 98 等非 UNICODE 环境下，字符串都是以 ANSI 编码的字节形式存在的。这种以字节形式存在的字符串，必须知道是哪种编码才能被正确地使用。这使我们形成了一个惯性思维：“字符串的编码”。  当 UNICODE 被支持后，Java 中的 String 是以字符的“序号”来存储的，不是以“某种编码的字节”来存储的，因此已经不存在“字符串的编码”这个概念了。只有在“字符串”与“字节串”转化时，或者，将一个“字节串”当成一个 ANSI 字符串时，才有编码的概念。  不少的人都有这个误解。 |

第一种误解，往往是导致乱码产生的原因。第二种误解，往往导致本来容易纠正的乱码问题变得更复杂。

在这里，我们可以看到，其中所讲的“误解一”，即采用每“一个字节”就是“一个字符”的转化方法，实际上也就等同于采用 iso-8859-1 进行转化。因此，我们常常使用 bytes = string.getBytes("iso-8859-1") 来进行逆向操作，得到原始的“字节串”。然后再使用正确的 ANSI 编码，比如 string = new String(bytes, "GB2312")，来得到正确的“UNICODE 字符串”。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**3.2 非 UNICODE 程序在不同语言环境间移植时的乱码**

非 UNICODE 程序中的字符串，都是以某种 ANSI 编码形式存在的。如果程序运行时的语言环境与开发时的语言环境不同，将会导致 ANSI 字符串的显示失败。

比如，在日文环境下开发的非 UNICODE 的日文程序界面，拿到中文环境下运行时，界面上将显示乱码。如果这个日文程序界面改为采用 UNICODE 来记录字符串，那么当在中文环境下运行时，界面上将可以显示正常的日文。

由于客观原因，有时候我们必须在中文操作系统下运行非 UNICODE 的日文软件，这时我们可以采用一些工具，比如，南极星，AppLocale 等，暂时的模拟不同的语言环境。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**3.3 网页提交字符串**

当页面中的表单提交字符串时，首先把字符串按照当前页面的编码，转化成字节串。然后再将每个字节转化成 "%XX" 的格式提交到 Web 服务器。比如，一个编码为 GB2312 的页面，提交 "中" 这个字符串时，提交给服务器的内容为 "%D6%D0"。

在服务器端，Web 服务器把收到的 "%D6%D0" 转化成 [0xD6, 0xD0] 两个字节，然后再根据 GB2312 编码规则得到 "中" 字。

在 Tomcat 服务器中，request.getParameter() 得到乱码时，常常是因为前面提到的“误解一”造成的。默认情况下，当提交 "%D6%D0" 给 Tomcat 服务器时，request.getParameter() 将返回 [0x00D6, 0x00D0] 两个 UNICODE 字符，而不是返回一个 "中" 字符。因此，我们需要使用 bytes = string.getBytes("iso-8859-1") 得到原始的字节串，再用 string = new String(bytes, "GB2312") 重新得到正确的字符串 "中"。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**3.4 从数据库读取字符串**

通过数据库客户端（比如 ODBC 或 JDBC）从数据库服务器中读取字符串时，客户端需要从服务器获知所使用的 ANSI 编码。当数据库服务器发送字节流给客户端时，客户端负责将字节流按照正确的编码转化成 UNICODE 字符串。

如果从数据库读取字符串时得到乱码，而数据库中存放的数据又是正确的，那么往往还是因为前面提到的“误解一”造成的。解决的办法还是通过 string = new String( string.getBytes("iso-8859-1"), "GB2312") 的方法，重新得到原始的字节串，再重新使用正确的编码转化成字符串。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**3.5 电子邮件中的字符串**

当一段 Text 或者 HTML 通过电子邮件传送时，发送的内容首先通过一种指定的**字符编码**转化成“字节串”，然后再把“字节串”通过一种指定的**传输编码**（Content-Transfer-Encoding）进行转化得到另一串“字节串”。比如，打开一封电子邮件源代码，可以看到类似的内容：

|  |
| --- |
| Content-Type: text/plain;         charset="gb2312" Content-Transfer-Encoding: base64  sbG+qcrQuqO17cf4yee74bGjz9W7+b3wudzA7dbQ0MQNCg0KvPKzxqO6uqO17cnnsaPW0NDEDQoNCg== |

最常用的 Content-Transfer-Encoding 有 Base64 和 Quoted-Printable 两种。在对二进制文件或者中文文本进行转化时，Base64 得到的“字节串”比 Quoted-Printable 更短。在对英文文本进行转化时，Quoted-Printable 得到的“字节串”比 Base64 更短。

邮件的标题，用了一种更简短的格式来标注“字符编码”和“传输编码”。比如，标题内容为 "中"，则在邮件源代码中表示为：

|  |
| --- |
| // 正确的标题格式 Subject: =?GB2312?B?1tA=?= |

其中，

* 第一个“=?”与“?”中间的部分指定了字符编码，在这个例子中指定的是 GB2312。
* “?”与“?”中间的“B”代表 Base64。如果是“Q”则代表 Quoted-Printable。
* 最后“?”与“?=”之间的部分，就是经过 GB2312 转化成字节串，再经过 Base64 转化后的标题内容。

如果“传输编码”改为 Quoted-Printable，同样，如果标题内容为 "中"：

|  |
| --- |
| // 正确的标题格式 Subject: =?GB2312?Q?=D6=D0?= |

如果阅读邮件时出现乱码，一般是因为“字符编码”或“传输编码”指定有误，或者是没有指定。比如，有的发邮件组件在发送邮件时，标题 "中"：

|  |
| --- |
| // 错误的标题格式 Subject: =?ISO-8859-1?Q?=D6=D0?= |

这样的表示，实际上是明确指明了标题为 [0x00D6, 0x00D0]，即 "ÖÐ"，而不是 "中"。

|  |
| --- |
| http://www.regexlab.com/images/blue_rule.gif |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | http://www.regexlab.com/images/u_bold.gif | [回页首](http://www.regexlab.com/zh/encoding.htm#main) | | |

**4. 几种错误理解的纠正**

**误解：“ISO-8859-1 是国际编码？”**

非也。iso-8859-1 只是单字节字符集中最简单的一种，也就是“字节编号”与“UNICODE 字符编号”一致的那种编码规则。当我们要把一个“字节串”转化成“字符串”，而又不知道它是哪一种 ANSI 编码时，先暂时地把“每一个字节”作为“一个字符”进行转化，不会造成信息丢失。然后再使用 bytes = string.getBytes("iso-8859-1") 的方法可恢复到原始的字节串。

**误解：“Java 中，怎样知道某个字符串的内码？”**

Java 中，字符串类 java.lang.String 处理的是 UNICODE 字符串，不是 ANSI 字符串。我们只需要把字符串作为“抽象的符号的串”来看待。因此不存在字符串的内码的问题。