1. 概述

本文主要包括以下几个方面：编码基本知识，java，系统软件，url，工具软件等。

在下面的描述中，将以"中文"两个字为例，经查表可以知道其 GB2312 编码是"d6d0 cec4"，Unicode 编码为"4e2d 6587"，UTF 编码就是"e4b8ad e69687"。注意，这两个字没有 iso8859-1 编码，但可以用 iso8859-1 编码来"表示"。

2. 编码基本知识

最早的编码是 iso8859-1，和 ascii 编码相似。但为了方便表示各种各样的语言，逐渐出现了很多标准编码，重要的有如下几个。

2.1. iso8859-1

属于单字节编码，最多能表示的字符范围是 0-255，应用于英文系列。比如，字母'a'的编码为 0x61=97。

很明显，iso8859-1 编码表示的字符范围很窄，无法表示中文字符。但是，由于是单字节编码，和计算机最基础的表示单位一致，所以很多时候，仍旧使用 iso8859-1 编码来表示。而且在很多协议上，默认使用该编码。比如，虽然"中文"两个字不存在 iso8859-1 编码，以 gb2312 编码为例，应该是"d6d0 cec4"两个字符，使用 iso8859-1 编码的时候则将它拆开为 4 个字节来表示："d6 d0 ce c4"（事实上，在进行存储的时候，也是以字节为单位处理的）。而如果是 UTF 编码，则是 6 个字节"e4 b8 ad e6 96 87"。很明显，这种表示方法还需要以另一种编码为基础。

2.2. GB2312/GBK

这就是汉子的国标码，专门用来表示汉字，是双字节编码，而英文字母和 iso8859-1 一致（兼容 iso8859-1 编码）。其中 gbk 编码能够用来同时表示繁体字和简体字，而 gb2312 只能表示简体字，gbk 是兼容 gb2312 编码的。

2.3. unicode

这是最统一的编码，可以用来表示所有语言的字符，而且是定长双字节（也有四字节的）编码，包括英文字母在内。所以可以说它是不兼容 iso8859-1 编码的，也不兼容任何编码。不过，相对于 iso8859-1 编码来说，uniocode 编码只是在前面增加了一个 0 字节，比如字母'a'为"00 61"。

需要说明的是，定长编码便于计算机处理（注意 GB2312/GBK 不是定长编码（汉字双字节，英文单字节）），而 **unicode 又可以用来表示所有字符，所以在很多软件内部是使用 unicode 编码来处理的，比如 java。**

2.4. UTF

考虑到 **unicode 编码不兼容 iso8859-1 编码，而且容易占用更多的空间：因为对于英文字母，unicode 也需要两个字节来表示。**所以 unicode **不便于传输和存储**。因此而产生了 utf 编码，utf 编码兼容 iso8859-1 编码，同时也可以用来表示所有语言的字符，不过，**utf 编码是不定长编码**，每一个字符的长度从 1-6 个字节不等。另外，utf 编码自带简单的校验功能。一般来讲，英文字母都是用一个字节表示，而汉字使用三个字节。

注意，虽然说 utf 是为了使用更少的空间而使用的，但那只是相对于 unicode 编码来说，如果已经知道是汉字，则使用 GB2312/GBK 无疑是最节省的。不过另一方面，值得说明的是，虽然 utf 编码对汉字使用 3 个字节，但即使对于汉字网页，utf 编码也会比 unicode 编码节省，因为网页中包含了很多的英文字符。

3. java 对字符的处理

在 java 应用软件中，会有多处涉及到字符集编码，有些地方需要进行正确的设置，有些地方需要进行一定程度的处理。

3.1. getBytes(charset)

这是 java 字符串处理的一个标准函数，其作用是将字符串所表示的字符按照 charset 编码，并以字节方式表示。注意字符串在 java 内存中总是按 unicode 编码存储的。比如"中文"，正常情况下（即没有错误的时候）存储为"4e2d 6587"，如果 charset 为"gbk"，则被编码为"d6d0 cec4"，然后返回字节"d6 d0 ce c4"。如果 charset 为"utf8"则最后是"e4 b8 ad e6 96 87"。如果是"iso8859-1"，则由于无法编码，最后返回 "3f 3f"（两个问号）。

3.2. new String(charset)

这是 java 字符串处理的另一个标准函数，和上一个函数的作用相反，将字节数组按照 charset 编码进行组合识别，最后转换为 unicode 存储。参考上述 getBytes 的例子，"gbk" 和"utf8"都可以得出正确的结果"4e2d 6587"，但 iso8859-1 最后变成了"003f 003f"（两个问号）。

因为 utf8 可以用来表示/编码所有字符，所以 new String(str.getBytes("utf8"), "utf8") === str，即完全可逆。

3.3. setCharacterEncoding()

该函数用来设置 http 请求或者相应的编码。

对于 request，是指提交内容的编码，指定后可以通过 getParameter()则直接获得正确的字符串，如果不指定，则默认使用 iso8859-1 编码，需要进一步处理。参见下述"表单输入"。值得注意的是在执行 setCharacterEncoding()之前，不能执行任何 getParameter()。java doc 上说明：This method must be called prior to reading request parameters or reading input using getReader()。而且，该指定只对 POST 方法有效，对 GET 方法无效。分析原因，应该是在**执行第一个 getParameter()的时候，java 将会按照编码分析所有的提交内容，而后续的 getParameter()不再进行分析**，所以 setCharacterEncoding()无效。而对于 GET 方法提交表单是，提交的内容在 URL 中，一开始就已经按照编码分析所有的提交内容，setCharacterEncoding()自然就无效。

对于 response，则是指定输出内容的编码，同时，该设置会传递给浏览器，告诉浏览器输出内容所采用的编码。

3.4. 处理过程

下面分析两个有代表性的例子，说明 java 对编码有关问题的处理方法。

3.4.1. 表单输入

User input *(gbk:d6d0 cec4) browser *(gbk:d6d0 cec4) web server iso8859-1(00d6 00d 000ce 00c4) class，需要在 class 中进行处理：getbytes("iso8859-1")为 d6 d0 ce c4，new String("gbk")为 d6d0 cec4，内存中以 unicode 编码则为 4e2d 6587。

l 用户输入的编码方式和页面指定的编码有关，也和用户的操作系统有关，所以是不确定的，上例以 gbk 为例。

l 从 browser 到 web server，可以在表单中指定提交内容时使用的字符集，否则会使用页面指定的编码。而**如果在 url 中直接用?的方式输入参数，则其编码往往是操作系统本身的编码，因为这时和页面无关。上述仍旧以 gbk 编码为例。**

l Web server **接收到的是字节流**，默认时（getParameter）会以 iso8859-1 编码处理之，结果是不正确的，所以需要进行处理。但如果预先设置了编码（通过 request. setCharacterEncoding ()），则能够直接获取到正确的结果。

l 在页面中指定编码是个好习惯，否则可能失去控制，无法指定正确的编码。

3.4.2. 文件编译

假设文件是 gbk 编码保存的，而编译有两种编码选择：gbk 或者 iso8859-1，前者是中文 windows 的默认编码，后者是 linux 的默认编码，当然也可以在编译时指定编码。

Jsp *(gbk:d6d0 cec4) java file *(gbk:d6d0 cec4) compiler read uincode(gbk: 4e2d 6587; iso8859-1: 00d6 00d 000ce 00c4) compiler write utf(gbk: e4b8ad e69687; iso8859-1: *) compiled file unicode(gbk: 4e2d 6587; iso8859-1: 00d6 00d 000ce 00c4) class。所以用 gbk 编码保存，而用 iso8859-1 编译的结果是不正确的。

class unicode(4e2d 6587) system.out / jsp.out gbk(d6d0 cec4) os console / browser。

l 文件可以以多种编码方式保存，中文 windows 下，默认为 ansi/gbk。

l 编译器读取文件时，需要得到文件的编码，如果未指定，则使用系统默认编码。一般 class 文件，是以系统默认编码保存的，所以编译不会出问题，但对于 jsp 文件，如果在中文 windows 下编辑保存，而部署在英文 linux 下运行/编译，则会出现问题。所以需要在 jsp 文件中用 pageEncoding 指定编码。

l Java 编译的时候会转换成统一的 unicode 编码处理，最后保存的时候再转换为 utf 编码。

l 当系统输出字符的时候，会按指定编码输出，对于中文 windows 下，System.out 将使用 gbk 编码，而对于 response（浏览器），则使用 jsp 文件头指定的 contentType，或者可以直接为 response 指定编码。同时，会告诉 browser 网页的编码。如果未指定，则会使用 iso8859-1 编码。对于中文，应该为 browser 指定输出字符串的编码。

l browser 显示网页的时候，首先使用 response 中指定的编码（jsp 文件头指定的 contentType 最终也反映在 response 上），如果未指定，则会使用网页中 meta 项指定中的 contentType。

3.5. 几处设置

对于 web 应用程序，和编码有关的设置或者函数如下。

3.5.1. jsp 编译

指定文件的存储编码，很明显，该设置应该置于文件的开头。例如：<%@page pageEncoding="GBK"%>。另外，对于一般 class 文件，可以在编译的时候指定编码。

3.5.2. jsp 输出

指定文件输出到 browser 是使用的编码，该设置也应该置于文件的开头。例如：<%@ page contentType="text/html; charset= GBK" %>。该设置和 response.setCharacterEncoding("GBK")等效。

3.5.3. meta 设置

指定网页使用的编码，该设置对静态网页尤其有作用。因为静态网页无法采用 jsp 的设置，而且也无法执行 response.setCharacterEncoding()。例如：<META http-equiv="Content-Type" content="text/html; charset=GBK" />

如果同时采用了 jsp 输出和 meta 设置两种编码指定方式，则 jsp 指定的优先。因为 jsp 指定的直接体现在 response 中。

需要注意的是，apache 有一个设置可以给无编码指定的网页指定编码，该指定等同于 jsp 的编码指定方式，所以会覆盖静态网页中的 meta 指定。所以有人建议关闭该设置。

3.5.4. form 设置

当浏览器提交表单的时候，可以指定相应的编码。例如：<form accept-charset= "gb2312">。一般不必不使用该设置，浏览器会直接使用网页的编码。

4. 系统软件

下面讨论几个相关的系统软件。

4.1. mysql 数据库

很明显，要支持多语言，应该将数据库的编码设置成 utf 或者 unicode，而 utf 更适合与存储。但是，如果中文数据中包含的英文字母很少，其实 unicode 更为适合。

数据库的编码可以通过 mysql 的配置文件设置，例如 default-character-set=utf8。还可以在数据库链接 URL 中设置，例如： useUnicode=true&characterEncoding=UTF-8。注意这两者应该保持一致，在新的 sql 版本里，在数据库链接 URL 里可以不进行设置，但也不能是错误的设置。

4.2. apache

appache 和编码有关的配置在 httpd.conf 中，例如 AddDefaultCharset UTF-8。如前所述，该功能会将所有静态页面的编码设置为 UTF-8，最好关闭该功能。

另外，apache 还有单独的模块来处理网页响应头，其中也可能对编码进行设置。

4.3. linux 默认编码

这里所说的 linux 默认编码，是指运行时的环境变量。两个重要的环境变量是 LC_ALL 和 LANG，默认编码会影响到 java URLEncode 的行为，下面有描述。

建议都设置为"zh_CN.UTF-8"。

4.4. 其它

为了支持中文文件名，linux 在加载磁盘时应该指定字符集，例如：mount /dev/hda5 /mnt/hda5/ -t ntfs -o iocharset=gb2312。

另外，如前所述，使用 GET 方法提交的信息不支持 request.setCharacterEncoding()，但可以通过 tomcat 的配置文件指定字符集，在 tomcat 的 server.xml 文件中，形如：<Connector ... URIEncoding="GBK"/>。这种方法将统一设置所有请求，而不能针对具体页面进行设置，也不一定和 browser 使用的编码相同，所以有时候并不是所期望的。

5. URL 地址

URL 地址中含有中文字符是很麻烦的，前面描述过使用 GET 方法提交表单的情况，使用 GET 方法时，参数就是包含在 URL 中。

5.1. URL 编码

对于 URL 中的一些特殊字符，浏览器会自动进行编码。这些字符除了"/?&"等外，还包括 unicode 字符，比如汉子。这时的编码比较特殊。

IE 有一个选项"总是使用 UTF-8 发送 URL"，当该选项有效时，IE 将会对特殊字符进行 UTF-8 编码，同时进行 URL 编码。如果改选项无效，则使用默认编码"GBK"，并且不进行 URL 编码。但是，对于 URL 后面的参数，则总是不进行编码，相当于 UTF-8 选项无效。比如"中文.html?a=中文"，当 UTF-8 选项有效时，将发送链接"%e4%b8%ad%e6%96%87.html?a=\x4e\x2d\x65\x87"；而 UTF-8 选项无效时，将发送链接"\x4e\x2d\x65\x87.html?a=\x4e\x2d\x65\x87"。注意后者前面的"中文"两个字只有 4 个字节，而前者却有 18 个字节，这主要时 URL 编码的原因。

当 web server（tomcat）接收到该链接时，将会进行 URL 解码，即去掉"%"，同时按照 ISO8859-1 编码（上面已经描述，可以使用 URLEncoding 来设置成其它编码）识别。上述例子的结果分别是"\ue4\ub8\uad\ue6\u96\u87.html?a=\u4e\u2d\u65\u87"和"\u4e\u2d\u65\u87.html?a=\u4e\u2d\u65\u87"，注意前者前面的"中文"两个字恢复成了 6 个字符。这里用"\u"，表示是 unicode。

所以，由于客户端设置的不同，相同的链接，在服务器上得到了不同结果。这个问题不少人都遇到，却没有很好的解决办法。所以有的网站会建议用户尝试关闭 UTF-8 选项。不过，下面会描述一个更好的处理办法。

5.2. rewrite

熟悉的人都知道，apache 有一个功能强大的 rewrite 模块，这里不描述其功能。需要说明的是该模块会自动将 URL 解码（去除%），即完成上述 web server（tomcat）的部分功能。有相关文档介绍说可以使用[NE]参数来关闭该功能，但我试验并未成功，可能是因为版本（我使用的是 apache 2.0.54）问题。另外，当参数中含有"?& "等符号的时候，该功能将导致系统得不到正常结果。

rewrite 本身似乎完全是采用字节处理的方式，而不考虑字符串的编码，所以不会带来编码问题。

5.3. URLEncode.encode()

这是 Java 本身提供对的 URL 编码函数，完成的工作和上述 UTF-8 选项有效时浏览器所做的工作相似。值得说明的是，java 已经不赞成不指定编码来使用该方法（deprecated）。应该在使用的时候增加编码指定。

当不指定编码的时候，该方法使用系统默认编码，这会导致软件运行结果得不确定。比如对于"中文"，当系统默认编码为"gb2312"时，结果是"%4e%2d%65%87"，而默认编码为"UTF-8"，结果却是"%e4%b8%ad%e6%96%87"，后续程序将难以处理。另外，这儿说的系统默认编码是由运行 tomcat 时的环境变量 LC_ALL 和 LANG 等决定的，曾经出现过 tomcat 重启后就出现乱码的问题，最后才郁闷的发现是因为修改修改了这两个环境变量。

建议统一指定为"UTF-8"编码，可能需要修改相应的程序。

5.4. 一个解决方案

上面说起过，因为浏览器设置的不同，对于同一个链接，web server 收到的是不同内容，而软件系统有无法知道这中间的区别，所以这一协议目前还存在缺陷。

针对具体问题，不应该侥幸认为所有客户的 IE 设置都是 UTF-8 有效的，也不应该粗暴的建议用户修改 IE 设置，要知道，用户不可能去记住每一个 web server 的设置。所以，接下来的解决办法就只能是让自己的程序多一点智能：根据内容来分析编码是否 UTF-8。

比较幸运的是 UTF-8 编码相当有规律，所以可以通过分析传输过来的链接内容，来判断是否是正确的 UTF-8 字符，如果是，则以 UTF-8 处理之，如果不是，则使用客户默认编码（比如"GBK"），下面是一个判断是否 UTF-8 的例子，如果你了解相应规律，就容易理解。

```
public static boolean isValidUtf8(byte[] b,int aMaxCount){
    int lLen=b.length,lCharCount=0;
    for(int i=0;i<lLen && lCharCount<aMaxCount;++lCharCount){
        byte lByte=b[i++];//to fast operation, ++ now, ready for the following for(;;)
        if(lByte>=0) continue;//>=0 is normal ascii
        if(lByte<(byte)0xc0 || lByte>(byte)0xfd) return false;
        int lCount=lByte>(byte)0xfc?5:lByte>(byte)0xf8?4
            :lByte>(byte)0xf0?3:lByte>(byte)0xe0?2:1;
        if(i+lCount>lLen) return false;
        for(int j=0;j<lCount;++j,++i) if(b[i]>=(byte)0xc0) return false;
```

```
        }
        return true;
}
```

相应地，一个使用上述方法的例子如下：

```
public static String getUrlParam(String aStr,String aDefaultCharset)
throws UnsupportedEncodingException{
        if(aStr==null) return null;
        byte[] lBytes=aStr.getBytes("ISO-8859-1");
        return new String(lBytes,StringUtil.isValidUtf8(lBytes)?"utf8":aDefaultCharset);
}
```

不过，该方法也存在缺陷，如下两方面：

l 没有包括对用户默认编码的识别，这可以根据请求信息的语言来判断，但不一定正确，因为我们有时候也会输入一些韩文，或者其他文字。

l 可能会错误判断 UTF-8 字符，一个例子是"学习"两个字，其 GBK 编码是" \xd1\xa7\xcf\xb0"，如果使用上述 isValidUtf8 方法判断，将返回 true。可以考虑使用更严格的判断方法，不过估计效果不大。

有一个例子可以证明 google 也遇到了上述问题，而且也采用了和上述相似的处理方法，比如，如果在地址栏中输入"http://www.google.com/search?hl=zh-CN&newwindow=1&q=学习"，google 将无法正确识别，而其他汉字一般能够正常识别。

最后，应该补充说明一下，如果不使用 rewrite 规则，或者通过表单提交数据，其实并不一定会遇到上述问题，因为这时可以在提交数据时指定希望的编码。另外，中文文件名确实会带来问题，应该谨慎使用。

6. 其它

下面描述一些和编码有关的其他问题。

6.1. SecureCRT

除了浏览器和控制台与编码有关外，一些客户端也很有关系。比如在使用 SecureCRT 连接 linux 时，应该让 SecureCRT 的显示编码（不同的 session，可以有不同的编码设置）和 linux 的编码环境变量保持一致。否则看到的一些帮助信息，就可能是乱码。

另外，mysql 有自己的编码设置，也应该保持和 SecureCRT 的显示编码一致。否则通过 SecureCRT 执行 sql 语句的时候，可能无法处理中文字符，查询结果也会出现乱码。

对于 Utf-8 文件，很多编辑器（比如记事本）会在文件开头增加三个不可见的标志字节，如果作为 mysql 的输入文件，则必须要去掉这三个字符。（用 linux 的 vi 保存可以去掉这三个字符）。一个有趣的现象是，在中文 windows 下，创建一个新 txt 文件，用记事本打开，输入"连通"两个字，保存，再打开，你会发现两个字没了，只留下一个小黑点。

6.2. 过滤器

如果需要统一设置编码，则通过 filter 进行设置是个不错的选择。在 filter class 中，可以统一为需要的请求或者回应设置编码。参加上述 setCharacterEncoding()。这个类 apache 已经给出了可以直接使用的例子 SetCharacterEncodingFilter。

6.3. POST 和 GET

很明显，以 POST 提交信息时，URL 有更好的可读性，而且可以方便的使用 setCharacterEncoding() 来处理字符集问题。但 GET 方法形成的 URL 能够更容易表达网页的实际内容，也能够用于收藏。

从统一的角度考虑问题，建议采用 GET 方法，这要求在程序中获得参数是进行特殊处理，而无法使用 setCharacterEncoding()的便利，如果不考虑 rewrite，就不存在 IE 的 UTF-8 问题，可以考虑通过设置 URIEncoding 来方便获取 URL 中的参数。

6.4. 简繁体编码转换

GBK 同时包含简体和繁体编码，也就是说同一个字，由于编码不同，在 GBK 编码下属于两个字。有时候，为了正确取得完整的结果，应该将繁体和简体进行统一。可以考虑将 UTF、GBK 中的所有繁体字，转换为相应的简体字，BIG5 编码的数据，也应该转化成相应的简体字。当然，仍旧以 UTF 编码存储。

例如，对于"语言 語言"，用 UTF 表示为"\xE8\xAF\xAD\xE8\xA8\x80 \xE8\xAA\x9E\xE8\xA8\x80"，进行简繁体编码转换后应该是两个相同的 "\xE8\xAF\xAD\xE8\xA8\x80>"。

Java 与 Unicode

博客分类：

- Java 基础

JavaJVM 虚拟机 IDEJDK

Java 与 Unicode：

Java 的 class 文件采用 utf8 的编码方式，JVM 运行时采用 utf16。Java 的字符串是 unicode 编码的。总之，Java 采用了 unicode 字符集，使之易于国际化。

Java 支持哪些字符集：

即 Java 能识别哪些字符集并对它进行正确地处理？查看 Charset 类，最新的 JDK 支持 160 种字符集。可以通过 static 方法 availableCharsets 拿到所有 Java 支持的字符集。

Java 代码

```
1.          assertEquals(160, Charset.availableCharsets().size());
2.          Set<String> charsetNames = Charset.availableCharsets().keySet();

3.          assertTrue(charsetNames.contains("utf-8"));
4.          assertTrue(charsetNames.contains("utf-16"));
5.          assertTrue(charsetNames.contains("gb2312"));
6.          assertTrue(Charset.isSupported("utf-8"));
```

需要在哪些时候注意编码问题？

1.从外部资源读取数据：

这跟外部资源采取的编码方式有关，我们需要使用外部资源采用的字符集来读取外部数据：

Java 代码

```
1.          InputStream is = new FileInputStream("res/input2.data");
```

```
2.          InputStreamReader streamReader = new InputStreamReader(is, "GB18030")
    ;
```

这里可以看到，我们采用了 GB18030 编码读取外部数据，通过查看 streamReader 的 encoding 可以印证：

Java 代码

```
1.          assertEquals("GB18030", streamReader.getEncoding());
```

正是由于上面我们为外部资源指定了正确的编码，当它转成 char 数组时才能正确地进行解码（GB18030 -> unicode）：

Java 代码

```
1.          char[] chars = new char[is.available()];
2.          streamReader.read(chars, 0, is.available());
```

但我们经常写的代码就像下面这样：

Java 代码

```
1.          InputStream is = new FileInputStream("res/input2.data");
2.          InputStreamReader streamReader = new InputStreamReader(is);
```

这时候 InputStreamReader 采用什么编码方式读取外部资源呢？Unicode？不是，这时候采用的编码方式是 JVM 的默认字符集，这个默认字符集在虚拟机启动时决定，通常根据语言环境和底层操作系统的 charset 来确定。可以通过以下方式得到 JVM 的默认字符集：

Java 代码

```
1.          Charset.defaultCharset();
```

为什么要这样？因为我们从外部资源读取数据，而外部资源的编码方式通常跟操作系统所使用的字符集一样，所以采用这种默认方式是可以理解的。

好吧，那么我通过我的 IDE Ideas 创建了一个文件，并以 JVM 默认的编码方式从这个文件读取数据，但读出来的数据竟然是乱码。为何？呵呵，其实是因为通过 Ideas 创建的文件是以 utf-8 编码的。要得到一个 JVM 默认编码的文件，通过手工创建一个 txt 文件试试吧。

2.字符串和字节数组的相互转换

我们通常通过以下代码把字符串转换成字节数组：

Java 代码

```
1.          "string".getBytes();
```

但你是否注意过这个转换采用的编码呢？其实上面这句代码跟下面这句是等价的：

Java 代码

```
1.          "string".getBytes(Charset.defaultCharset());
```

也就是说它根据 JVM 的默认编码（而不是你可能以为的 unicode）把字符串转换成一个字节数组。
反之，如何从字节数组创建一个字符串呢？

Java 代码

```
new String("string".getBytes());
```

同样，这个方法使用平台的默认字符集解码字节的指定数组（这里的解码指从一种字符集到 unicode）。

字符串编码迷思：

Java 代码

```
new String(input.getBytes("ISO-8859-1"), "GB18030")
```

上面这段代码代表什么？有人会说：“把 input 字符串从 ISO-8859-1 编码方式转换成 GB18030 编码方式”。如果这种说法正确，那么又如何解释我们刚提到的 java 字符串都采用 unicode 编码呢？
这种说法不仅是欠妥的，而且是大错特错的，让我们一一来分析，其实事实是这样的：我们本应该用 GB18030 的编码来读取数据并解码成字符串，但结果却采用了 ISO-8859-1 的编码，导致生成一个错误的字符串。要恢复，就要先把字符串恢复成原始字节数组，然后通过正确的编码 GB18030 再次解码成字符串（即把以 GB18030 编码的数据转成 unicode 的字符串）。注意，字符串永远都是 unicode 编码的。

但编码转换并不是负负得正那么简单，这里我们之所以可以正确地转换回来，是因为 ISO8859-1 是单字节编码，所以每个字节被按照原样 转换为 String ，也就是说，虽然这是一个错误的转换，但编码没有改变，所以我们仍然有机会把编码转换回来！

总结：
所以，我们在处理 java 的编码问题时，要分清楚三个概念：Java 采用的编码：unicode，JVM 平台默认字符集和外部资源的编码。

程序示例：

Java 代码

```
public static void main(String args[])
    {
        System.out.println("default charset : "+Charset.defaultChars
et());
        String str = "abc 你好";//string with UTF-8 charset

        byte[] bytes = str.getBytes(Charset.forName("UTF-8"));//conv
ert to byte array with UTF-8 encode
        for (byte b : bytes)
        {
            System.out.print(b + " ");
        }
        System.out.println();
        try
        {
```

```
                    String str1 = new String(bytes, "UTF-8");//to UTF-8 stri
ng
                    String str2 = new String(bytes, "ISO-8859-1");//to ISO-8
859-1 string
                    String str3 = new String(bytes, "GBK");//to GBK string

                    System.out.println(str1);//abc 你好
                    System.out.println(str2);//abc??????
                    System.out.println(str3);//abc 浣犲ソ

                    System.out.println();
                    byte[] bytes2 = str2.getBytes(Charset.forName("ISO-8859-
1"));
                    for (byte b : bytes2)
                    {
                        System.out.print(b + " ");
                    }
                    System.out.println();
                    String str22 = new String(bytes2, "UTF-8");
                    System.out.println(str22);//abc 你好

                    System.out.println();
                    byte[] bytes3 = str3.getBytes(Charset.forName("GBK"));
                    for (byte b : bytes3)
                    {
                        System.out.print(b + " ");
                    }
                    System.out.println();
                    String str33 = new String(bytes3, "UTF-8");
                    System.out.println(str33);//abc 你好
                } catch (UnsupportedEncodingException e)
                {
                    e.printStackTrace();
                }
            }
```

运行结果：

Html 代码

```
        default charset : GBK
        97 98 99 -28 -67 -96 -27 -91 -67
        abc 你好
        abc??????
        abc 浣犲ソ

```

```
7.          97 98 99 -28 -67 -96 -27 -91 -67
8.          abc 你好
9.
10.         97 98 99 -28 -67 -96 -27 -91 -67
11.         abc 你好
```

http://blog.csdn.net/wcnqrm/article/details/5888297

1、JVM 中单个字符占用的字节长度跟编码方式有关，而默认编码方式又跟平台是一一对应的或说平台决定了默认字符编码方式；2、对于单个字符：ISO-8859-1 单字节编码，GBK 双字节编码，UTF-8 三字节编码；因此中文平台(中文平台默认字符集编码 GBK)下一个中文字符占 2 个字节，而英文平台(英文平台默认字符集编码 Cp1252(类似于 ISO-8859-1))。
3、getBytes()、getBytes(encoding)函数的作用是使用系统默认或者指定的字符集编码方式，将字符串编码成字节数组。
Me 结论：编码方式决定字节长度；
在中文平台下，默认的字符集编码是 GBK，此时如果使用 getBytes()或者 getBytes("GBK")，则按照 GBK 的编码规则将每个中文字符用 2 个 byte 表示。所以我们看到"中文"最终 GBK 编码结果就是：-42 -48 -50 -60 。-42 和-48 代表了"中"字，而"-50"和"-60"则代表了"文"字。
在中文平台下，如果指定的字符集编码是 UTF-8，那么按照 UTF-8 对中文的编码规则：每个中文用 3 个字节表示，那么"中文"这两个字符最终被编码成：-28 -72 -83、-26 -106 -121 两组。每 3 个字节代表一个中文字符。
在中文平台下，如果指定的字符集编码是 ISO-8859-1，由于此字符集是单字节编码，所以使用 getBytes("ISO-8859-1")时，每个字符只取一个字节，每个汉字只取到了一半的字符。另外一半的字节丢失了。由于这一半的字符在字符集中找不到对应的字符，所以默认使用编码 63 代替，也就是?。

在英文平台下，默认的字符集编码是 Cp1252(类似于 ISO-8859-1)，如果使用 GBK、UTF-8 进行编码，得到的字节数组依然是正确的(GBK4 个字节，UTF-8 是 6 个字节)。因为在 JVM 内部是以 Unicode 存储字符串的，使用 getBytes(encoding)会让 JVM 进行一次 Unicode 到指定编码之间的转换。对于 GBK，JVM 依然会转换成 4 个字节，对于 UTF-8，JVM 依然会转换成 6 个字节。但是对于 ISO-8859-1，则由于无法转换(2 个字节--->1 个字节，截取了一半的字节)，所以转换后的结果是错误的。

相同的平台下，同一个中文字符，在不同的编码方式下，得到的是完全不同的字节数组。这些字节数组有可能是正确的(只要该字符集支持中文)，也可能是完全错误的(该字符集不支持中文)。
记住：
不要轻易地使用或滥用 String 类的 getBytes(encoding)方法，更要尽量避免使用 getBytes()方法。因为这个方法是平台依赖的，在平台不可预知的情况下完全可能得到不同的结果。如果一定要进行字节编码，则用户要确保 encoding 的方法就是当初字符串输入时的 encoding。
【结论】字节数组转换成字符串，然后转换成字符数组问题：

和 getBytes(encoding)不同，toCharArray()返回的是"自然字符"。但是这个"自然字符"的数目和内容却是由原始的编码方式决定的。来看看里面是如何进行字符串的操作的：

```
String encodedString = new String(content.getBytes(), encoding);
char[] charArray = inStr.toCharArray();
```

可以看到系统首先对原始字符串按照默认的编码方式进行编码，得到一个字节数组，然后按照指定的

新的编码方式进行解码，得到新的编码后的字符串。再转换成对应的字符数组。

由于在中文平台下，默认的字符集编码是 GBK，于是 content.getBytes()得到的是什么呢？就是下面这 4 个字节：
byte[0] = -42 hex string = ffffffd6
byte[1] = -48 hex string = ffffffd0
byte[2] = -50 hex string = ffffffce
byte[3] = -60 hex string = ffffffc4
如果新的 encoding 是 GBK，那么经过解码后，由于一个字符用 2 个字节表示。于是最终的结果就是：

char[0]='中' --- byte[0] + byte[1]
char[1]='文' --- byte[2] + byte[3]
如果新的 encoding 是 ISO-8859-1，那么经过解码后，由于一个字符用 1 个字节表示，于是原来本应该 2 个字节一起解析的变成单个字节解析，每个字节都代表了一个汉字字符的一半。这一半的字节在 ISO-8859-1 中找不到对应的字符，就变成了"?"了，最终的结果：
char[0]='?' ---- byte[0]
char[1]='?' ---- byte[1]
char[2]='?' ---- byte[2]
char[3]='?' ---- byte[3]
如果新的 encoding 是 UTF-8，那么经过解码后，由于一个字符用 3 个字节表示，于是原来 4 个字节的数据无法正常的解析成 UTF-8 的数据，最终的结果也是每一个都变成"?"。
char[0]='?' ---- byte[0]
char[1]='?' ---- byte[1]
char[2]='?' ---- byte[2]
char[3]='?' ---- byte[3]

如果是在英文平台下，由于默认的编码方式是 Cp1252，于是 content.getBytes()得到的字节都是被截去一半的残留字符，所以我们看到在英文平台下，不论指定的 encoding 是 GBK、UTF-8，其结果和 ISO-8859-1 都是一样的。
记住：
这个方法再次证明了 String 的 getBytes()方法的危险性，如果我们使用 new String(str.getBytes(), encoding)对字符串进行重新编码解码时，我们一定要清楚 str.getBytes()方法返回的字节数组的长度、内容到底是什么，因为在接下来使用新的 encoding 进行编码解码时，Java 并不会自动地对字节数组进行扩展以适应新的 encoding。而是按照新的编码方法直接对该字节数组进行解析。
于是结果就像上面的例子一样，同样是 4 个原始字节，有些每 2 个一组进行解析，有些每个一组进行解析，有些每 3 个一组进行解析。其结果就只能看那种编码方式合适了。
【结论】
FileWriter 是字符流输出流，而 OutputStreamWriter 是字节流输出流
①在中文平台下，如果使用 FileWriter，不论你如何设置字符集都不会起作用。因为它采用的是默认的系统字符集。即便你设置了 System.setProperty("file.encoding", "ISO-8859-1")，或者在运行时给予参数-Dfile.encoding=UTF-8 都不会起作用。你会发现它最终还是都已"GB2312"或者"GBK"的方式保存。
在中文平台下，如果使用 OutputStreamWriter，则在后台写入时会把字符流转换成字节流，此时指定的编码字符集就起作用了。可以看到在指定 GBK、UTF-8 的情况下中文可以正常的保存和读取，同

时文件按照我们给定的方式保存了。而对于 ISO-8859-1 则变成了?，这再次证明了采用 ISO-8859-1 是不能保存中文的，而且会因为中文编码在 ISO-8859-1 的编码中找不到对应的字符而默认转换成?。

②在英文平台下，如果使用 FileWriter，不论你如何设置字符集同样都不会起作用。所有的文件都将按照 ISO-8859-1 的编码方式保存，毫无疑问地变成了?。在英文平台下，如果使用 OutputStreamWriter，则只有当我们把字符和文件的编码方式正确设置为 GBK、UTF-8 的情况下，中文才能正确的保存并显示。
③通过上述的实验证明，为了确保在不同的平台下，客户端输入的中文可以被正确地解析、保存、读取。最好的办法就是使用 OutputStreamWriter 配合 UTF-8 编码。
如果不想使用 UTF-8 编码，那么可以考虑使用 GB2312，不建议使用 GBK、GB18030。因为对于某些老式的文本编辑器，甚至不支持 GBK、GB18030 的编码，但是对于 GB2312 则是一定支持的。因为前两者都不是国标但后者是。
④关于 String 的 getBytes()，getBytes(encoding)和 new String(bytes, encoding)这三个方法，非常值得注意：
A.getBytes()：使用平台默认的编码方式(通过 file.encoding 属性获取)方式来将字符串转换成 byte[]。得到的是字符串最原始的字节编码值。
B.getBytes(NAME_OF_CHARSET)：使用指定的编码方式将字符串转换成 byte[]，如果想要得到正确的字节数组，程序员必须给出正确的 NAME_OF_CHARSET。否则得到的就不会得到正确的结果。

C.new String(bytes, encoding)：如果我们在客户端使用 UTF-8 编码的 JSP 页面发出请求，浏览器编码后的 UTF-8 字节会以 ISO-8859-1 的形式传递到服务器端。所以要得到经 HTTP 协议传输的原始字节，我们需要先调用 getBytes("ISO-8859-1")得到原始的字节，但由于我们客户端的原始编码是 UTF-8，如果继续按照 ISO-8859-1 解码，那么得到的将不是一个中文字符，而是 3 个乱码的字符。所以我们需要再次调用 new String(bytes,"UTF-8")，将字节数组按照 UTF-8 的格式，每 3 个一组进行解码，才能还原为客户端的原始字符。
D.String 的 getBytes()、getBytes(NAME_OF_CHARSET)方法都是比较微妙的方法，原则上：传输时采用的是什么编码，我们就需要按照这种编码得到字节。new String(bytes, NAME_OF_CHARSET)则更加需要小心，原则上：客户端采用的是什么编码，那么这里的 NAME_OF_CHARSET 就必须和客户端保持一致。

例如 JSP 页面是 GBK，那么我们接收页面传递而来的参数时就必须使用 new String(parameter.getBytes("ISO-8859-1"), "GBK");如果使用了错误的解码方式，如使用了 UTF-8，那么得到的很有可能就是乱码了。
也就是说：GBK--->ISO-8859-1--->GBK、UTF-8--->ISO-8859-1--->UTF-8 的转换过程是没有问题的。但是 GBK--->ISO-8859-1--->UTF-8、UTF-8--->ISO-8859-1--->GBK 的字节直接转码则可能导致乱码，需要另外的转换过程。
记住：
谨慎地使用 getBytes(NAME_OF_CHARSET)和 new String(bytes, NAME_OF_CHARSET)，除非你很清楚的知道原始的字符编码和传输协议使用的编码。
推荐使用基于服务器的配置、过滤器设置 request/response 的 characterEncoding、content type 属性。还有就是 JSP 页面的 pageEncoding 属性、HTML meta 元素的 content type 属性。尽量避免频繁的在代码中进行字符串转码，即降低了效率又增加了风险。