十分钟搞清字符集和字符编码

2015/03/13 • IT技术 • 806 阅读 • 3 评论 • 字符编码, 字符集

玩转Bootstrap（JS插件篇）
JAVA实现对称加密
jQuery源码解析（DOM与核心模块）
jQuery源码解析（架构与依赖模块）

原文出处： 卢钧轶 欢迎分享原创到伯乐头条

本文将简述字符集，字符编码的概念。以及在遭遇乱码时的一些常用诊断技巧。

背景：字符集和编码无疑是IT菜鸟甚至是各种大神的头痛问题。当遇到纷繁复杂的字符集，各种火星文和乱码时，问题的定位往往变得非常困难。本文就将会从原理方面对字符集和编码做个简单的科普介绍，同时也会介绍一些通用的乱码故障定位的方法以方便读者以后能够更从容的定位相关问题。在正式介绍之前，先做个小申明：如果你希望非常精确的理解各个名词的解释，那么可以查阅wikipedia。本文是博主通过自己理解消化后并转化成易懂浅显的表述后的介绍。

什么是字符集

在介绍字符集之前，我们先了解下为什么要有字符集。我们在计算机屏幕上看到的是实体化的文字，而在计算机存储介质中存放的实际是二进制的比特流。那么在这两者之间的转换规则就需要一个统一的标准，否则把我们的U盘插到老板的电脑上，文档就乱码了；小伙伴QQ上传过来的文件，在我们本地打开又乱码了。于是为了实现转换标准，各种字符集标准就出现了。简单的说字符集就规定了某个文字对应的二进制数字存放方式（编码）和某串二进制数值代表了哪个文字（解码）的转换关系。

那么为什么会有那么多字符集标准呢？这个问题实际非常容易回答。问问自己为什么我们的插头拿到英国就不能用了呢？为什么显示器同时有DVI，VGA，HDMI，DP这么多接口呢？很多规范和标准在最初制定时并不会意识到这将会是以后全球普适的准则，或者处于组织本身利益就想从本质上区别于现有标准。于是，就产生了那么多具有相同效果但又不相互兼容的标准了。

说了那么多我们来看一个实际例子，下面就是屌这个字在各种编码下的十六进制和二进制编码结果，怎么样有没有一种很屌的感觉？

字符集	16进制编码	对应的二进制数据
UTF-8	0xE5B18C	1110 0101 1011 0001 1000 1100
UTF-16	0x5C4C	1011 1000 1001 1000
GBK	0x8CC5	1000 1100 1100 0101

什么是字符编码

字符集只是一个规则集合的名字，对应到真实生活中，字符集就是对某种语言的称呼。例如：英语，汉语，日语。对于一个字符集来说要正确编码转码一个字符需要三个关键元素：字库表（character repertoire）、编码字符集（coded character set）、字符编码（character encoding form）。其中字库表是一个相当于所有可读或者可显示字符的数据库，字库表决定了整个字符集能够展现表示的所有字符的范围。编码字符集，即用一个编码值code point来表示一个字符在字库中的位置。字符编码，将编码字符集和实际存储数值之间的转换关系。一般来说都会直接将code point的值作为编码后的值直接存储。例如在ASCII中A在表中排第65位，而编码后A的数值是0100 0001也即十进制的65的二进制转换结果。

看到这里，可能很多读者都会有和我当初一样的疑问：字库表和编码字符集看来是必不可少的，那既然字库表中的每一个字符都有一个自己的序号，直接把序号作为存储内容就好了。为什么还要多此一举通过字符编码把序号转换成另外一种存储格式呢？其实原因也比较容易理解：统一字库表的目的是为了能够涵盖世界上所有的字符，但实际使用过程中会发现真正用的上的字符相对整个字库表来说比例非常低。例如中文地区的程序几乎不会需要日语字符，而一些英语国家甚至简单的ASCII字库表就能满足基本需求。而如果把每个字符都用字库表中的序号来存储的话，每个字符就需要3个字节（这里以Unicode字库为例），这样对于原本用仅占一个字符的ASCII编码的英语地区国家显然是一个额外成本（存储体积是原来的三倍）。算的直接一些，同样一块硬盘，用ASCII可以存1500篇文章，而用3字节Unicode序号存储只能存500篇。于是就出现了UTF-8这样的变长编码。在UTF-8编码中原本只需要一个字节的ASCII字符，仍然只占一个字节。而像中文及日语这样的复杂字符就需要2个到3个字节来存储。

UTF-8和Unicode的关系

看完上面两个概念解释，那么解释UTF-8和Unicode的关系就比较简单了。Unicode就是上文中提到的编码字符集，而UTF-8就是字符编码，即Unicode规则字库的一种实现形式。随着互联网的发展，对同一字库集的要求越来越迫切，Unicode标准也就自然而然的出现。它几乎涵盖了各个国家语言可能出现的符号和文字，并将为他们编号。详见：Unicode on Wikipedia。Unicode的编号从0000开始一直到10FFFF共分为16个Plane，每个Plane中有65536个字符。而UTF-8则只实现了第一个Plane，可见UTF-8虽然是一个当今接受度最广的字符集编码，但是它并没有涵盖整个Unicode的字库，这也造成了它在某些场景下对于特殊字符的处理困难（下文会有提到）。

UTF-8编码简介

为了更好的理解后面的实际应用，我们这里简单的介绍下UTF-8的编码实现方法。即UTF-8的物理存储和Unicode序号的转换关系。
UTF-8编码为变长编码。最小编码单位（code unit）为一个字节。一个字节的前1-3个bit为描述性部分，后面为实际序号部分。

- 如果一个字节的第一位为0，那么代表当前字符为单字节字符，占用一个字节的空间。0之后的所有部分（7个bit）代表在Unicode中的序号。
- 如果一个字节以110开头，那么代表当前字符为双字节字符，占用2个字节的空间。110之后的所有部分（7个bit）代表在Unicode中的序号。且第二个字节以10开头
- 如果一个字节以1110开头，那么代表当前字符为三字节字符，占用2个字节的空间。110之后的所有部分（7个bit）代表在Unicode中的序号。且第二、第三个字节以10开头
- 如果一个字节以10开头，那么代表当前字节为多字节字符的第二个字节。10之后的所有部分（6个bit）代表在Unicode中的序号。

具体每个字节的特征可见下表，其中x代表序号部分，把各个字节中的所有x部分拼接在一起就组成了在Unicode字库中的序号

Byte 1	Byte 2	Byte3
0xxx xxxx		
110x xxxx	10xx xxxx	
1110 xxxx	10xx xxxx	10xx xxxx

我们分别看三个从一个字节到三个字节的UTF-8编码例子：

实际字符	在Unicode字库序号的十六进制	在Unicode字库序号的二进制	UTF-8编码后的二进制	UTF-8编码后的十六进制
$	0024	010 0100	0010 0100	24
¢	00A2	000 1010 0010	1100 0010 1010 0010	C2 A2
€	20AC	0010 0000 1010 1100	1110 0010 1000 0010 1010 1100	E2 82 AC

细心的读者不难从以上的简单介绍中得出以下规律：

- 3个字节的UTF-8十六进制编码一定是以E开头的
- 2个字节的UTF-8十六进制编码一定是以C或D开头的
- 1个字节的UTF-8十六进制编码一定是以比8小的数字开头的

为什么会出现乱码

先科普下乱码的英文native说法是mojibake

简单的说乱码的出现是因为：编码和解码时用了不同或者不兼容的字符集。对应到真实生活中，就好比是一个英国人为了表示祝福在纸上写了bless（编码过程）。而一个法国人拿到了这张纸，由于在法语中bless表示受伤的意思，所以认为他想表达的是受伤（解码过程）。这个就是一个现实生活中的乱码情况。在计算机科学中一样，一个用UTF-8编码后的字符，用GBK去解码。由于两个字符集的字库表不一样，同一个汉字在两个字符表的位置也不同，最终就会出现乱码。

我们来看一个例子：假设我们用UTF-8编码存储很屌两个字，会有如下转换：

字符	UTF-8编码后的十六进制
很	E5BE88

很	E5BE88
屠	E5B18C

于是我们得到了E5BE88E5B18C这么一串数值。而显示时我们用GBK解码进行展示，通过查表我们获得以下信息：

两个字节的十六进制数值	GBK解码后对应的字符
E5BE	寰
88E5	堝
B18C	砻

解码后我们就得到了寰堝砻这么一个错误的结果，更要命的是连字符个数都变了。

如何识别乱码的本来想要表达的文字

要从乱码字符中反解出原来的正确文字需要对各个字符集编码规则有较为深刻的掌握。但是原理很简单，这里用最常见的UTF-8被错误用GBK展示时的乱码为例，来说明具体反解和识别过程。

第1步 编码

假设我们在页面上看到寰堝砻这样的乱码，而又得知我们的浏览器当前使用GBK编码。那么第一步我们就能先通过GBK把乱码编码成二进制表达式。当然查表编码效率很低，我们也可以用以下SQL语句直接通过MySQL客户端来做编码工作：

```
mysql [localhost] {msandbox} > select hex(convert('寰堝砻' using gbk));
+
----------------------------------------+
| hex(convert('寰堝砻' using gbk))    |
+
----------------------------------------+
| E5BE88E5B18C                           |
+
----------------------------------------+
1 row in set (0.01 sec)
```

第2步 识别

现在我们得到了解码后的二进制字符串E5BE88E5B18C。然后我们将它按字节拆开。

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
E5	BE	88	E5	B1	8C

然后套用之前UTF-8编码介绍章节中总结出的规律，就不难发现这6个字节的数据符合UTF-8编码规则。如果整个数据流都符合这个规则的话，我们就能大胆假设乱码之前的编码字符集是UTF-8

第3步 解码

然后我们就能拿着E5BE88E5B18C用UTF-8解码，查看乱码前的文字了。当然我们可以不查表直接通过SQL获得结果：

```
mysql [localhost] {msandbox} ((none)) > select convert(0xE5BE88E5B18C using utf8);
+------------------------------------+
```

```
| convert(0xE5BE88E5B18C using utf8) |
+------------------------------------+
| 很屌                               |
+------------------------------------+
1 row in set (0.00 sec)
```

常见问题处理之Emoji

所谓Emoji就是一种在Unicode位于\u1F601-\u1F64F区段的字符。这个显然超过了目前常用的UTF-8字符集的编码范围\u0000-\uFFFF。Emoji表情随着IOS的普及和微信的支持越来越常见。下面就是几个常见的Emoji:

那么Emoji字符表情会对我们平时的开发运维带来什么影响呢？最常见的问题就在于将他存入MySQL数据库的时候。一般来说MySQL数据库的默认字符集都会配置成UTF-8（三字节），而utf8mb4在5.5以后才被支持，也很少会有DBA主动将系统默认字符集改成utf8mb4。那么问题就来了，当我们把一个需要4字节UTF-8编码才能表示的字符存入数据库的时候就会报错：ERROR 1366: Incorrect string value: '\xF0\x9D\x8C\x86' for column 。 如果认真阅读了上面的解释，那么这个报错也就不难看懂了。我们试图将一串Bytes插入到一列中，而这串Bytes的第一个字节是\xF0意味着这是一个四字节的UTF-8编码。但是当MySQL表和列字符集配置为UTF-8的时候是无法存储这样的字符的，所以报了错。

那么遇到这种情况我们如何解决呢？有两种方式：升级MySQL到5.6或更高版本，并且将表字符集切换至utf8mb4。第二种方法就是在把内容存入到数据库之前做一次过滤，将Emoji字符替换成一段特殊的文字编码，然后再存入数据库中。之后从数据库获取或者前端展示时再将这段特殊文字编码转换成Emoji显示。第二种方法我们假设用-*-1F601-*-来替代4字节的Emoji，那么具体实现python代码可以参见Stackoverflow上的回答

reference

如何配置Python默认字符集
字符编码笔记：ASCII，Unicode和UTF-8
Unicode中文编码表
Emoji Unicode Table
Every Developer Should Know About The Encoding

相关文章

- 关于字符编码，你所需要知道的
- 字符集和字符编码（Charset & Encoding）
- 字符编码常识及问题解析
- 10分钟学会理解和解决MySQL乱码问题
- MySQL之终端：管理数据库的基本操作
- 理解 MySQL（1）：架构和概念