字符串，那些你不知道的事

发表于 2015-11-20 23:16 更新日期:2015-11-22 14:04

最近在看《Dive Into Python 3》，第四章讲了字符串相关知识，看后才发现，字符串远比我们想象的要复杂多。就像该书所说的

> Everything you thought you knew about strings is wrong.

是的，我之前对字符串的理解都是错的。

也许你会诧异，字符串有什么难的，即便遇到乱码的情况随便 Google 下就能找到解决方法，但是这样你不觉得有种被动的感觉嘛，我觉得和学习任何东西一样，学习编程首要是学习其思想，知道某事物为什么（why）要这么做，至于如何做（how）那只是前辈们提出的解决方案，我们可以参考，随便掌握下来。

本文下面首先讲解字符、字符串、编码、ASCII、Unicode、UTF-8 等一些基本概念，然后会介绍在使用计算机时是如何如编码打交道的，也就是实战部分。
希望大家在阅读完本文后，都能对 string 有一全新的认识。

文章目录

为什么需要字符编码

当我们谈到字符串（string或text）时，你可能会想到“计算机屏幕上的那些字符（characters）与符号（symbols）”，你正在阅读的文章，无非也是由一串字符组成的。但是你也许会发现，你无法给“字符串”一明确定义，但是我们就是知道，就像给你一个苹果，你能说出其名字，但是不能给出准确定义一样。这个问题先放一放，后面我再解释。

我们知道，计算机并不能直接处理操作字符与符号，它只认识 0、1 这两个数字，所以如果想让计算机显示各种各样的字符与符号，就必须定义它们与数字的一一映射关系，也就是我们所熟知的字符编码（character encoding）。你可简单的认为，字符编码为计算机屏幕上显示的字符与这些字符保存在内存或磁盘中的形式提供了一种映射关系。字符编码纷繁复杂，有些专门为特定语言优化，像针对简体中文的编码就有 GB2312，GBK（GB 是 Guajia Biaozhun 的简写）等，针对英文的 ASCII；另一些专门用于多语言环境，像后面要讲到的 UTF-8。

我们可以把**字符编码**看作一种解密密钥（decryption key），当我们收到一段字节流时，无论来自文件还是网络，如果我们知道它是“文本（text）”，那么我们就需要知道采用何种字符编码来**解码**这些字节流，否则，我们得到的只是一堆无意义的符号，像 `锘挎`。

编码所产生的问题

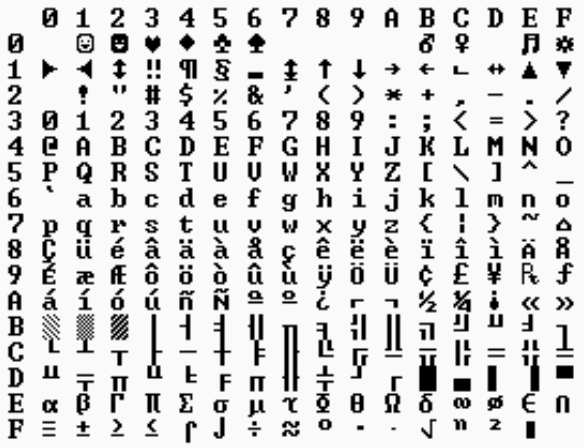
计算机最早起源于以英文为母语的美国，英文中的符号比较少，用七个二进制位就足以表示，现在最常见也是最流行的莫过于 ASCII 编码，该编码使用 0 到 127 之间的数字来存储字符（65表示“A”，97表示“a”）。

USASCII code chart

$b_7 b_6 b_5$ / Bits	000	001	010	011	100	101	110	111
b_4 b_3 b_2 b_1 / Row ↓ Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

ASCII_Code_Chart-Quick_ref_card

我们知道一个字节是 8 位，ASCII 编码其实只使用了其中的低 7 位，还剩下 1 位。在早期，很多 OEM 厂商就想着可以利用 128-255 来表示一些特殊符号，其中比较有名的是 IBM-PC 提出了 OEM 字符集（character set），为欧洲国家的语言提供注音以及一些线画符号（line drawing characters），如下图，利用这些人们可以在计算机上画一些有趣的图形。



ascii-dos

随着计算机的普及，使用计算机的人不再仅仅局限于以英文为母语的国家，不同国家的 OEM 字符集也如雨后春笋，层出不穷。由于0-127的编码已经固定下来，所以它们大都使用128-255来编码自己的符号集。例如，有些地方用 130 表示 `é`，但在以色列表示为 Hebrew letter Gimel `ג`。

这种 OEM 厂商“混战”的情况最终被 ANSI 标准制止，在 ANSI 标准中，对于0-127之间的编码与 ASCII 保持一致，最高位用 0 填充。但是对于128以及之上的编码，争议比较大，不同地区往往不一致，这些不同的编码，导致了不同的 code pages 的产生。

可以看到，只是单字节的编码问题就已经存在这么严重的问题了，像中文（Chinese），日文（Japanese），韩文（Korean）（业界一般称为 CJK）等象形（表意）文字（ideograph-based language），字符数量比较多，1 个字节是放不下的，所以需要更多的字节来进行字符的编码。和单字节编码一样，如果没有统一的规范，不同国家自己定制自己的编码标准，那么不同国家之间是无法进行交流的，所以，需要一个囊括世界上所有字符的编码方案的出现，但是这里还有个问题，如果用多字节来对字符进行编码，那么对于 ASCII 字符来说，是比较浪费空间的，针对这两个问题，聪明的人们提出了一全新的字符编码方案——Unicode。

Unicode_logo

Unicode

Unicode 的全称是 universal character encoding，中文一般翻译为“统一码、万国码、单一码”。

Unicode 主要解决了前面所提到的两个问题：

1. 统一世界上所有字符的编码，使得语言不同的国家也能正常交换信息
2. 提出了一个中间层，使得字符的编码与存储形式分离解耦，这样不同国家就可以采用不同的存储方案，来解决单字节表达字符数有限与多字节编码浪费的矛盾。

Unicode 的关键创新点在于为字符编码与最终的存储形式加了一中间层（术语为 code points），这样，当一种语言有新字符产生时，只需分配新的 code point 即可，具体的存储形式（一个字节还是两个字节，采用大端还是小端）不需要关心。

Unicode 中采用四个字节来定义 code point，每一个 code point 都代表世界上唯一的字符，不会出现同一 code point 在不同国家表示不同字符的情况。比如，`U+0041` 总是代表 `A`，即便某语言中没有这个字符。

Unicode 的存储形式

Unicode 的**存储形式**一般称为 `UTF-*` 编码，其中 UTF 全称为 `Unicode Transformation Format`，常见的有：

UTF-32

UTF-32 编码是 Unicode 最直接的存储方式，用 4 个字节来分别表示 code point 中的 4 个字节，也是 `UTF-*` 编码家族中唯一的一种**定长编码（fixed-length encoding）**。UTF-32 的好处是能够在 `O(1)` 时间内找到第 N 个字符，因为第 N 个字符的编码的起点是 N*4 个字节，当然，劣势更明显，四个字节表示一个字符，别说以英文为母语的人不干，我们中国人也不干了。

UTF-16

UTF-16 最少可以采用 2 个字节表示 code point，需要注意的是，UTF-16 是一种**变长编码（variable-length encoding）**，只不过对于 65535 之内的 code point，采用 2 个字节表示而已。如果想要表示 65535 之上的字符，需要一些 hack 的手段，具体可以参考wiki UTF-16#U.2B10000_to_U.2B10FFFF。很明显，UTF-16 比 UTF-32 节约一半的存储空间，如果用不到 65535 之上的字符的话，也能够在 `O(1)` 时间内找到第 N 个字符。

> UTF-16 与 UTF-32 还有一个不明显的缺点。我们知道不同的计算机存储字节的顺序是不一样的，这也就意味着 `U+4E2D` 在 UTF-16 可以保存为 `4E 2D`，也可以保存成 `2D 4E`，这取决于计算机是采用大端模式还是小端模式，UTF-32 的情况也类似。为了解决这个问题，引入了 **BOM (Byte Order Mark)**，它是一特殊的不可见字符，位于文件的起始位置，标示该文件的字节序。对于 UTF-16 来说，BOM 为 `U+FEFF`（FF 比 FE 大 1），如果某 以 UTF-16 编码的文件以 `FF FE` 开始，那么就意味着字节序为**小端模式**，如果以 `FE EE` 开始，那么就是**大端模式**。

UTF-8

UTF-16 对于以英文为母语的人来说，还是有些浪费了，这时聪明的人们（准确说是Ken Thompson与Rob Pike）又发明了另一个编码——UTF-8。在 UTF-8 中，ASCII 字符采用单字节。其实，UTF-8 前 128 个字符与 ASCII 字符编码方式一致；扩展的拉丁字符像 `ñ`、`ö` 等采用2个字节存储；中文字符采用 3 个字符存储，使用频率极少字符采用 4 个字节存储。由此可见，UTF-8 也是一种**变长编码（variable-length encoding）**。

UTF-8 的编码规则很简单，只有二条：

1. 对于单字节的符号，字节的第一位设为0，后面7位为这个符号的 code point。因此对于英语字母，UTF-8编码和ASCII码是相同的。
2. 对于n字节的符号，第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的 code point。

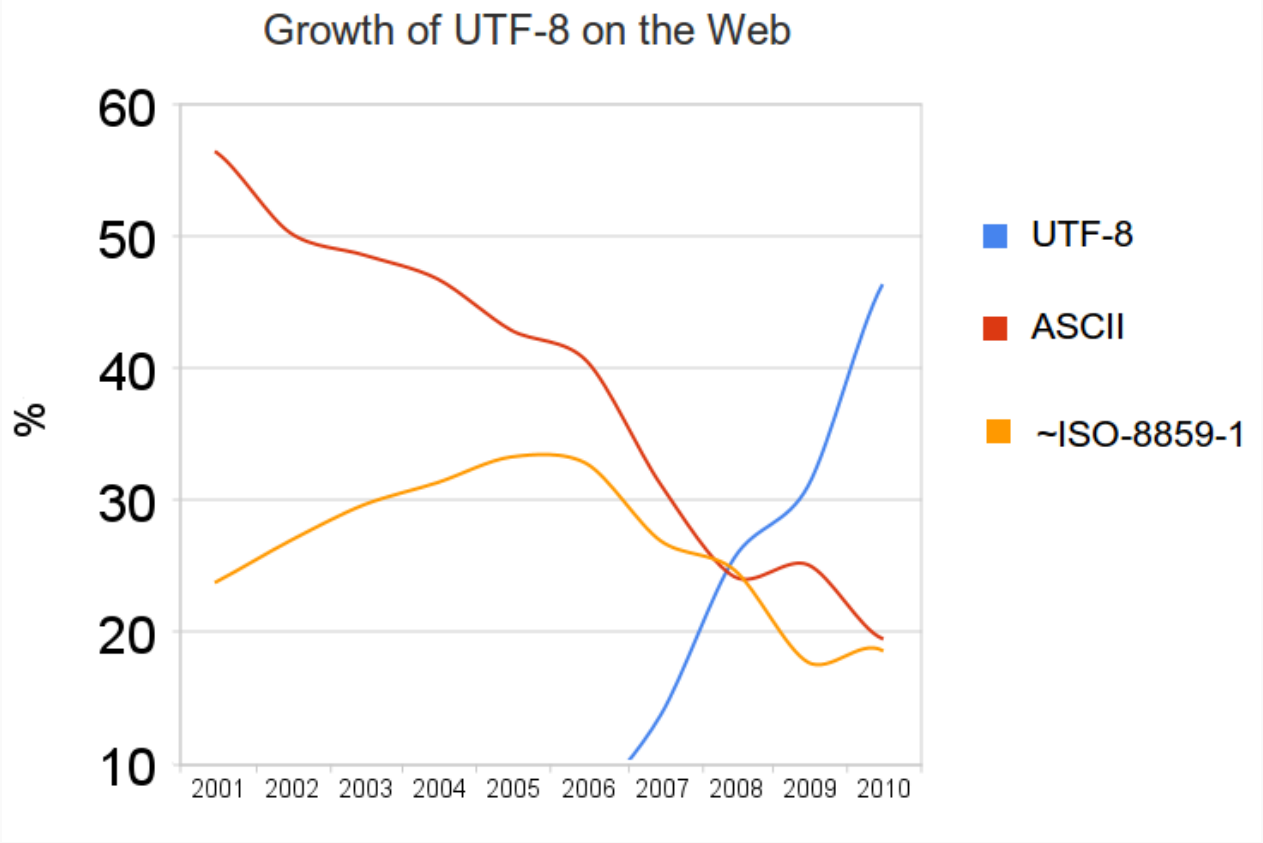
Character		Binary code point	Binary UTF-8	Hexadecimal UTF-8
$	U+0024	0100100	00100100	24
¢	U+00A2	00010100010	11000010 10100010	C2 A2
€	U+20AC	0010000010101100	11100010 10000010 10101100	E2 82 AC
𐍈	U+10348	00001000001101001000	11110000 10010000 10001101 10001000	F0 90 8D 88

UTF-8 编码规则

通过上面这两个规则，UTF-8 就不存在字节顺序在大小端不同的情况，所以用 UTF-8 编码的文件在任何计算机中保存的字节流都是一致的，这是其很重要一优势；UTF-8 的另一大优势在于对 ASCII 字符超节省空间，存储扩展拉丁字符与 UTF-16 的情况一样，存储汉字字符比 UTF-32 更优。

UTF-8 的一劣势是查找第 N 个字符时需要 `O(N)` 的时间，也就是说，字符串越长，就需要更长的时间来查找其中的每个字符。其次是在对字节流解码、字符编码时，需要遵循上面两条规则，比 UTF-16、UTF-32 略麻烦。

随着互联网的兴起，UTF-8 是逐渐成为使用范围最广的编码方案。



UnicodeGrow2b.png

UCS

我们在互联网上查找编码相关资料时，经常会看到 `UCS-2`、`UCS-4` 编码，它们和 `UTF-*` 编码家族是什么关系呢？要想理清它们之间的关系，需要先弄清楚，什么是 **UCS**。

UCS 全称是 **Universal Coded Character Set**，是由 ISO/IEC 10646定义的一套标准字符集，是很多字符编码的基础，UCS 中大概包含 100,000 个抽象字符，每一个

字符都有一唯一的数字编码，称为 code point。

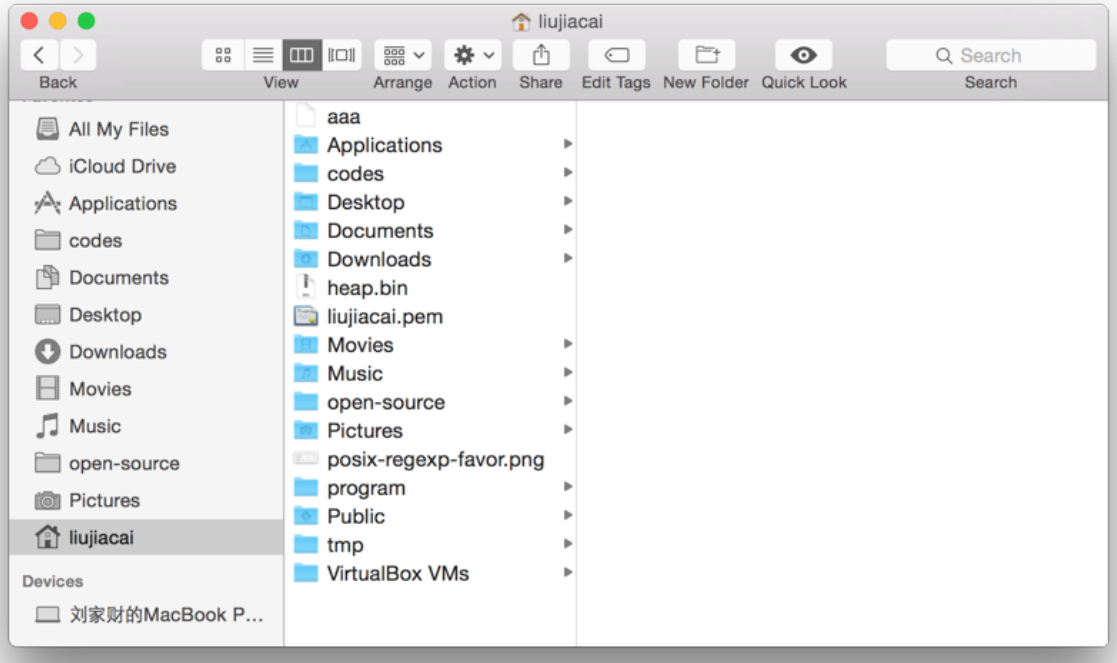
在19世纪八十年代晚期，有两个组织同时在 UCS 的基础上开发一种与具体语言无关的统一的编码方案，这两个组织分别是 IEEE 与 Unicode Consortium，为了保持这两个组织间编码方案的兼容性，两个组织尝试着合作。早期的两字节编码方案叫做“Unicode”，后来改名为“UCS-2”，在研发过程发，发现 16 位根本不能够囊括所有字符，于是 IEEE 引入了新的编码方案——UCS-4 编码，这种编码每个字符需要 4 个字节，这一行为立刻被 Unicode Consortium 制止了，因为这种编码太浪费空间了，又因为一些设备厂商已经对 2 字节编码技术投入大量成本，所以在 1996 年 7 月发布的 Unicode 2.0 中提出了 UTF-16 来打破 UCS-2 与 UCS-4 之间的僵局，UTF-16 在 2000 年被 IEFE 组织制定为RFC 2781标准。

由此可见，`UCS-*` 编码是一历史产物，目前来说，统一编码方案最终的赢家是 `UTF-*` 编码。

实战

操作系统

根据UTF-16 FOR PROCESSING，现在流行的三大操作系统 Windows、Mac、Linux 均采用 UTF-16 编码方案，上面链接也指出，现代编程语言像 Java、ECMAScript、.Net 平台上所有语言等在内部也都使用 UTF-16 来表示字符。



Mac Finder 界面

上图为 Mac 系统文件浏览器 Finder 的界面，其中所有的字符，在内存中都是以

UTF-16 的编码方式存储的。

你也许会问，为什么操作系统都这么偏爱 UTF-16，Stack Exchange 上面有一个精彩的回答，感兴趣的可以去了解

- Should UTF-16 be considered harmful?
- UTF-8 Everywhere

Locale

为了适应多语言环境，Linux/Mac 系统通过 locale 来设置系统的语言环境，下面是我在 Mac 终端输入 `locale` 得到的输出

```
LANG="en_US.UTF-8"           <==主语言的环境
LC_COLLATE="en_US.UTF-8"     <==字串的比较排序等
LC_CTYPE="en_US.UTF-8"       <==语言符号及其分类
LC_MESSAGES="en_US.UTF-8"    <==信息显示的内容，如功能表、错误信息等
LC_MONETARY="en_US.UTF-8"    <==币值格式的显示等
LC_NUMERIC="en_US.UTF-8"     <==数字系统的显示信息
LC_TIME="en_US.UTF-8"        <==时间系统的显示资料
LC_ALL="en_US.UTF-8"         <==语言环境的整体设定
```

locale 按照所涉及到的文化传统的各个方面分成12个大类，上面的输出只显示了其中的 6 类。为了设置方便，我们可以通过设置 `LC_ALL` 、 `LANG` 来改变这 12 个分类熟悉。其优先级关系为

> `LC_ALL` > `LC_*` > `LANG`

设置好 locale，操作系统在进行文本字节流解析时，如果没有明确制定其编码，就用 locale 设定的编码方案，当然现在的操作系统都比较聪明，在用默认编码方案解码不成功时，会尝试其他编码，现在比较成熟的编码测探技术有Mozila 的 UniversalCharsetDetection 与 ICU 的 Character Set Detection 。

编程语言

Java

一般来说，高级编程语言都提供都对字符的支持，像 Java 中的 Character 类就采用 UTF-16 编码方案。

这里有个文字游戏，一般我们说“某某字符串是XX编码”，其实这是不合理的，因为字符串压根就没有编码这一说法，只有**字符**才有，字符串只是字符的一串序列而已。
不过我们平时并没有这么严谨，不过你要明白，当我们说“某某字符串是XX编码”时，知道这其实指的是该字符串中字符的编码就可以了。

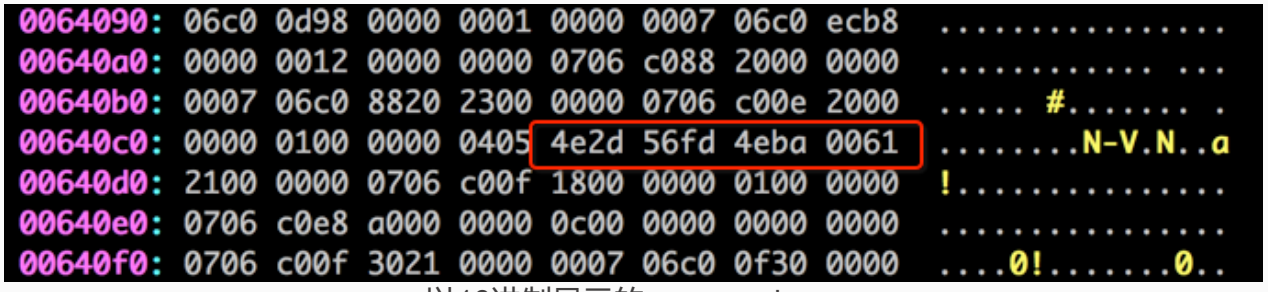
我们可以做个简单的实验来验证 Java 中确实使用 UTF-16 编码来存储字符：

```java
public class EncodingTest {
    public static void main(String[] args) {
        String s  = "中国人a";
        try {
            //线程睡眠，不要让线程退出
            Thread.sleep(10000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在使用 javac 编译这个类时，javac 会按照操作系统默认的编码去解析字节流，如果你保存的源文件编码与操作系统默认不一致，是可能出错的，可以在启动 javac 命令时，附加 `-encoding <encoding>` 选项来指明源代码文件所使用的编码。

```
# 编译生成 .class 文件
javac -encoding utf-8 EncodingTest.java
# 执行该类
java EncodingTest
# 使用 jps 查看其 pid，然后用 jmap 把程序运行时内存的内容 dump 下来
jmap -dump:live,format=b,file=encoding_test.bin <pid>
# 在 Linux/Mac 系统上，使用 `xxd` 命令以十六进制查看该文件，我这里用管道传给了 vim
xxd encoding_test.bin | vim -
```

在 vim 中可以看到下图所示片段



以16进制显示的memory dump

其中我用红框标注部分就是上面 EncodingTest 类中字符串 `s` 的内容，`4e2d` 是“中”的 code point，`56fd` 是“国”的 code point，`4eba` 是“人”的 code point，`0061` 是“a”的 code point。而在 UTF-16 编码中，0-66535之间的字符直接用两个字节存储，这也就证明了 Java 中的 `Character` 是使用 UTF-16 编码的。

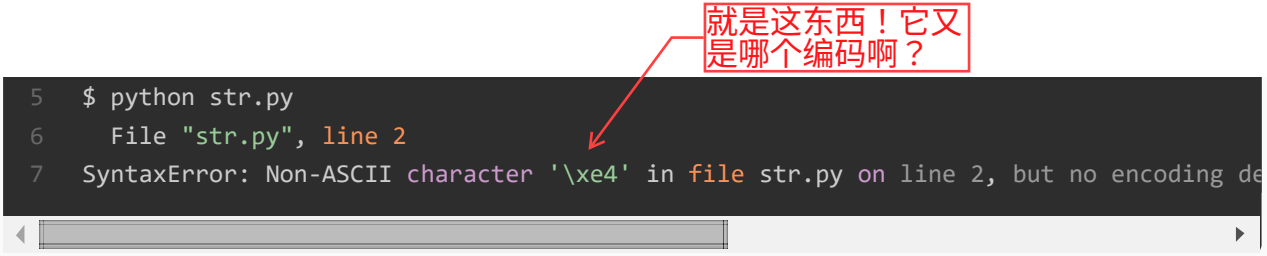
Python

首先说下 Python 解释器如何解析 Python 源程序。

在 Python 2 中，Python 解析器默认用 ASCII 编码来读取源程序，当程序中包含非 ASCII 字符时，解释器会报错，下面实验在我 Mac 上用 python 2.7.6 进行：

```
$ cat str.py
#!/usr/bin/env python
a = "众人过"

```



我们可以通过在源程序起始处用 `coding: name` 或 `coding=name` 来声明源程序所用的编码。

Python 3 中改变了这一行为，解析器默认采用 UTF-8 解析源程序。

按理接下来应该介绍 Python 中对字符的处理了，但是发现这里面东西太多了，介于本文篇幅原因，这里不再介绍，后面有空可以单独写篇文件介绍。大家感兴趣的可以参考下面的文章：

- More About Unicode in Python 2 and 3

JavaScript

- JavaScript's internal character encoding: UCS-2 or UTF-16?

HTML/XML

在我们的 Web 浏览器接收到来自世界各地的 HTML/XML 时，也需要正确的编码方案才能够正常显示网页，在现代的 HTML5 页面，一般通过下面的代码指定

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

4.0.1 之前的 HTML 页面使用下面的代码

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

XML 使用属性标注其编码

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

仔细想想，这里有个矛盾的地方，因为我们需要事先知道某字节流的编码才能正确解析该字节流，而这个字节流的编码是保存在这段字节流中的，和“鸡生蛋，蛋生鸡”的问题有点像，其实这并不是一个问题，因为大部分的编码都是兼容 ASCII 编码的，而这些 HTML/XML 开始处基本都是 ASCII 字符，所以采用浏览器默认的编码方案即可解析出该字节流所声明的编码，在解析出该字节流所用编码后，使用该编码重新解析该字节流。

VIM

- vim_fileencodings_detection

总结

这篇文章我用了周末 2 天时间才完成，在 wiki 上搜索的资料时，需要消耗较大精力去整理，因为各个编码都不是孤立存在的，要想完整了解某编码，需要从起发展历史开始，在不同编码条目间来回切换，才能对其有深入理解。这是我意料之外的。

字符串，这个既熟悉又陌生的东西，相信大家在看本文后，大家都能够对其有一全新的认识。

参考

1. The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
2. Are data type declarators like “int” and “char” stored in RAM when a C program executes?
3. UTF-8 Everywhere
4. 字符编码笔记：ASCII，Unicode和UTF-8
5. Java: a rough guide to character encoding
6. Python Unicode HOWTO
7. How does my computer store things in memory?
8. 谈谈Unicode编码，简要解释UCS、UTF、BMP、BOM等名词

理解计算机 string

0

下一篇：

SF 黑客马拉松赛后回顾