https://www.zhihu.com/question/23374078

计算机内只能保存101010等二进制数据，那么页面上显示的字符是如何显示出来的呢？  
一：字符集（Charset）  
charset = char + set，char 是字符，set是集合，charset就是字符的集合。  
字符集就是是这个编码方式涵盖了哪些字符，每个字符都有一个数字序号。  
二：编码方式（Encoding）  
编码方式就是一个字符要怎样编码成二进制字节序，或者反过来怎么解析。  
也即给你一个数字序号，要编码成几个字节，字节顺序如何，或者其他特殊规则。  
三：字形字体（Font）  
根据数字序号调用字体存储的字形，就可以在页面上显示出来了。  
所以一个字符要显示出来，要显示成什么样子要看字体文件。  
  
综上所述，Unicode 只是字符集，而没有编码方式。  
UTF-8 是一种 Unicode 字符集的编码方式，其他还有 UTF-16，UTF-32 等。  
而有了字符集以及编码方式，如果系统字体是没有这个字符，也是显示不出来的。

作者：匿名用户  
链接：https://www.zhihu.com/question/23374078/answer/55846548  
来源：知乎  
著作权归作者所有，转载请联系作者获得授权。

#############################################################################

作者：龙腾道默默地  
链接：https://www.zhihu.com/question/23374078/answer/121955274  
来源：知乎  
著作权归作者所有，转载请联系作者获得授权。

* **[encoding] 文件编码**（硬件传输存储、系统/应用软件解码）
  + **包装层**。文件编码可以有多层，比如最外面套的可以是传输储/存编码（也可以不套）：
    - data:base64,...
    - [https://..](//link.zhihu.com/?target=https%3A//..).加密传输编码
    - chunked切片传输编码（Transfer-Encoding）
    - gzip压缩编码（Content-Encoding）
  + **文件层**。解读编码分为：
    - **二进制文件（狭义）**，如：exe程序、mp3音乐、mp4视频、pdf文档（没错这不是普通文本文件）
    - **[text/\*]文本文件**，如：txt、md、html、mhtml、css、json、js（所以text/javascript比application/javascript更合理；注意，这里重点是/前面的部分，后面的javascript已经不属于编码的范畴，只是文件内容的含义，与编码没有半毛钱关系。）

其实包装层和文本文件本质上也是二进制文件，因为计算机本身就是二进制的。所以上面提到的二进制其实是狭义的二进制。

* + 单独指出包装层，是为了提示文件编码可以不止一层，否则有人就会困惑一个txt文件为什么编码会是gzip——这只是因为HTTP传输时最外层加了一层压缩，而标注编码总是从外而内的。
  + 而单独指出文本文件，是因为文本文件这种二进制非常基础、常见，而且有很多独有的特性、遵循很多统一的规范。像pdf虽然最终可以显示文字，但这个过程技术上是不透明的。而文本文档可以不依赖于独特的解读软件而过得普遍的浏览。

不过事实上文本文档也有几种不同的编码方式（后文会谈到原因）。最开始文本文件就是ANSI编码，UTF-8是后来出现的（后文会结合字符集解释）。  
简言之，**UTF-8**就是**文件编码**层次的概念，也是文本文件保存时选择的编码类型之一。

* **[character set] 字符集**
  1. **计算机的发明**。由于一个字节（Byte）有8位（bit），类似10101010，因此一个基本的存储单位能够表示2的8次方（即256）种字符，而基本的计算机字符（所谓的半角字符，说白了是发明计算机的美国常用的字符）用128个就表示完了，所以这时候不存在任何问题，一个字节（计算机存储单位）对应一个字符（人类自然语言文字中的单位），这套字符集被称为ASCII，包括了半角英文大小写、半角数字、半角表单符号等。闲得无聊的美国人还用多出来的一位作校验码，用于避免电路不稳定信号传输错误（其实是看着浪费不顺眼）。  
     *（之所以一个字节不是7位，是考虑到2的整数幂，要么4位、要么16位，8位刚好够用，后文会看出幸好如此，否则很难想象有预留的先见之明，毕竟研究各国文化恐怕比发明计算机还麻烦。）*
  2. **后来欧洲语言的扩充**。计算机的世界化过程中，欧洲紧随其后。他们马上发现各种注音的拉丁语系字母无法表示，因此把后128位直接用上了，128+128，皆大欢喜。  
     *（一方面这可以理解为目光短浅，另一方面也确实没有必要浪费更多空间。）*
  3. **世界另一边象形文字的扩充**。以中文为例，虽然比欧洲稍晚，但实际上应当与欧洲理解为第二传播梯队同期的各自闭门造车。由于中文太多，索性用多个计算机字节（两个）表示一个中文字符。为了避免与美国原始体系冲突，我们规定当一个字节在128以内时，还是原义；超出128的，需要与后面一个字节组合解读成一个字符。这样128瞬间变成了128\*256，完全够用了。这套字符集命名以GB（国标）开头，有过几次扩充（GB2312、GBK、GB18030），相互不完全兼容，与欧洲扩展字符集自然也不兼容。注意中国人用计算机也是要处理拉丁文、日韩等文字的，所以其中并不只有中文，只是各国实现的标准不同。同期产生的还有Big5（港澳台地区繁体中文标准）、JIS（日本）等。  
     *（表面上看起来各国各自为营很不合理，实际上一开始就要考虑所有问题会导致各自极其不合理，而事实上各自为营的发展速度和地位上升才是后一步的现实前提。）*
  4. **世界统一标准**。随着互联网的发展，计算机工业不再仅限于处理不同的文字，还必须用同一套方式处理。Universal Character Set，简称UCS，别名Unicode（实际是两个并行起草的字符集，快弄好了才发现彼此干脆合并了协议），应运而生。Unicode包罗万象，并预留好了无限扩展的方案。随着计算机硬件发展，这种通用方案虽然略浪费空间，却极大地节省了人的工作成本。

可见，**Unicode**是**字符集**层面的概念，处理的是每个汉字对应的计算机编号。  
  
  
上面，基本从各自的角度，讲了Unicode作为字符集，与UTF-8作为文件编码的区别。  
下面，谈谈**二者的关联**（如果是无关的不同，也不可能构成混淆）：

* 1. **ASCII字符集**，最初的字符集，对应了ANSI文件编码，一个字符用一个字节表示，严格来说这个时候字符集和文本文件的编码并没有必要作为单独的概念加以区分；
  2. **西欧字符集、日韩、港澳台、国标字符集等**，仍然使用了ANSI文件编码，其中部分字符集一个字符要用两个字节；
  3. **Unicode字符集**，终于在完成了字符集层面的集大成后，开始反过头来规定适合自己的文件编码（保存为ANSI会丢失大量的新增扩展字符），即Unicode Transformation Format（Unicode转换格式，其中每种方案针对不同发展阶段的使用情境设计，在其针对的情况下都能够大幅度地减少常见字的空间占用）：
     + **UTF-32**（= UCS-4 + 未来拓展保留）：直接把毫不考虑空间占用的Unicode字符集编码数字保存起来，文件大很多，优点是每个字符都一定是用4个字节表示（8bit/Byte \* 4Byte = 32bit）。  
       字符集作为信息处理层的规范，统一才是刚需，大大提升开发和运算效率；但是到了储存、网络传输层，这就成为了硬伤。  
       *（事实上百万种文字的预留可能性在实际上会变成半数以上的零补位，直到外星文字出现——而真的出现了，恐怕又完全不够用或要协商甚至被迫采用新标准了，所以这种理论方案实际上没有用处。）*
     + **UTF-16**（= UCS-2 + 配对处理能力）：将2字节字符视为主流的方案，在表示英文时总是浪费一倍，在表示更罕见的字符时可能要超过4位；不论如何，总是2字节作为最小单位（8bit/Byte \* 2Byte = 16bit）。  
       文本编辑器中的UTF-16选项通常写作“Unicode”或“Unicode big endian”（高低位顺序不同，后者与16进制Unicode编码的直观顺序一致）。
     + **UTF-8**：目前最主流的方案。虽然英文字符只有128个，但却占了主要的使用比例，因此默认单字节字符最常见、而将1字节作为编码单位的UTF-8成为了最现实而且通行的方案（8bit/Byte \* 2Byte = 8bit）——既能表示各国文字，又以英文为主要使用情境而极大地节省了传输和存储的负担；说白了就是英文仍然占据主要地位。在中文为主的情况下，不如基于ANSI编码的国标字符集节省空间，也因此UTF-8在国内的推行用了很长的时间，不过使用UTF-8已经越来越利大于弊了。  
       *（推行的速度其实不完全是观念问题，因为标准之为标准，本质与其说在于合理，更在于通行；而通行是以习惯和地位作为基础的。如果人们没有某种行为习惯，又怎么会产生针对这种行为习惯的行为规范呢？）*

实例小结：

Transfer-Encoding: chunked

Content-Encoding: gzip

Content-Type: text/html; charset=UTF-8

以上面这个常见的HTTP报头为例，

* 1. 第一行的意思是，数据传输使用的是分片式的，最开始不能确定一共有多大，需要逐步接收；
  2. 接收完成后，第二行的意思是，收到的文件是被压缩过的，需要先用指定的gzip方式解压缩才能得到文件；
  3. 得到文件本身后，第三行的text部分告知这是一个文本文件；
  4. UTF-8本身是文件编码信息，却写在charset（字符集）的位置，其实是暗示了其从属的Unicode字符集系统，同时又顺便指出了文件编码信息（因为二者是高度绑定的关系），从而正确获取了文件的文字内容；  
     *（非网络环境下，本地文件因为没有HTTP报头，实际上是通过文件开始处的BOM信息确定具体是UTF-8、UTF-16还是UTF-32，其中UTF-8因为高地位无关，可以省略BOM）*
  5. 最后，根据第三行的html部分得知，这个文件应该作为网页来显示。

下面是一个针锋相对的例子：

Content-Length: 1024

Content-Type: text/plain; charset=GB2312

* 1. 第一行表示文件大小是确定的，收到1024字节的数据才算完成接收；
  2. 没有压缩编码提示，所以文件可以直接使用；
  3. 第二行的text表明这是一个文本文件；
  4. GB2312字符集暗示了文件肯定是ANSI编码的，从而配合字符集正确获取了文件的文字内容；
  5. 最后，根据plain得知文件是一个普通文本文件，直接显示即可。

**关于转义（escape）：**  
HTML实体转义、编程中的字符串转义、URL网址转义都称为转义，但前两个属于字符集（char set）的范畴，而后者属于编码（encode）的范畴（编码用于文件储存、文件传输，以及此处的网址传输）。

* + **HTML实体**
  + &#40845; “龍”的Unicode编号是40845
  + 或是：
  + &#x9F8D; 40845的16进制是9F8D，前面加x表示16进制
  + **JavaScript字符串**
  + "龍".charCodeAt(0)===40845;
  + String.fromCharCode(40845)==="龍";
  + 0x9F8D===40845;
  + "龍".charCodeAt(0)===0x9F8D;
  + String.fromCharCode(0x9F8D)==="龍";
  + "\u9F8D"==="龍";
  + **URL**  
      
    [https://www.bing.com/search?q=龍](//link.zhihu.com/?target=https%3A//www.bing.com/search%3Fq%3D%25E9%25BE%258D)  
      
    由于URL在传输中不支持中文，所以上面这个网址会被用UTF-8规则转码成：  
    [https://www.bing.com/search?q=%E9%BE%8D](//link.zhihu.com/?target=https%3A//www.bing.com/search%3Fq%3D%25E9%25BE%258D)  
      
    “龍”的UTF-8编码是“E9 BE 8D”。  
      
    早期百度使用的是GB字符集（ANSI）编码：  
    [https://www.baidu.com/s?ie=gb2312&wd=%FD%88](//link.zhihu.com/?target=https%3A//www.baidu.com/s%3Fie%3Dgb2312%26wd%3D%25FD%2588)  
    “龍”的国标编码是“FD 88”，确实节省了空间，但现在已经非常不建议这样做了。

**为什么业界不推荐中文编程，并应当坚持使用UTF-8编码？**  
阻碍世界范围内的合作开发项目，以及，出现编码错误（旧系统/基层编程环境不支持中文，其他国家用户未必安装了独特的字符集）。