在面试的笔试题里出了一道开放性的题：请简述Unicode与UTF-8之间的关系。一道看似简单的题，能给出满意答案的却寥寥无几

，确实挺失望的。所以今天就结合我以前做过的一个关于字符编码的分享，总结一些与字符编码相关的知识和问题。如果你这方面的 知识已经掌握的足够了，可以忽略这篇文字。但如果你没法很好的回答我上面的面试题，或经常被乱码的问题所困扰，还是不妨一读。

基本常识

1.位和字节

说起编码，我们必须从最基础的说起，**位和字节**(别觉得这个过于简单不值一说，我还真见过很多个不能区分这两者的程序员)。位（bit）是指计算机里存放的二进制值(0/1)，而8个位组合成的“位串”称为一个字节，容易算出，8个位的组合有256（ 2^8）个组合方式，其取值范围是“00000000-11111111”，常用十六进制来表示。比如“01000001”就是一个字节，其对应的十六进制值为“0x41”。

而我们通常所讲的字符编码，就是指**定义一套规则**，将真实世界里的字母/字符与计算机的二进制序列进行相互转化。如我们可以针对上面的字节定义如下的转换规则：

```
01000001（0x41）<-> 65 <-> 'A'
```

即用字位序“01000001”来表示字母’A’。

2.拉丁字符

拉丁字符是当今世界使用最广泛的符号了。通常我们说的拉丁字母，指的的是**基础拉丁字母**,即指常见的”ABCD“等26个英文字母，这些字母与英语中一些常见的符号（如数字，标点符号）称为**基础拉丁字符**，这些基础拉丁字符在使用英语的国家广为流行，当然在中国，也被用来当作汉语拼音使用。在欧洲其它一些非英语国家，为满足其语言需要，在基础拉丁字符的基础上，加上一些连字符，变音字符(如’Á’)，形成了**派生拉丁字母**，其表示的字符范围在各种语言有所不同，而**完整意义上的拉丁字符是指这些变体字符与基础拉丁字符的全集**。是比基础拉丁字符集大很多的一个集合。

编码标准

前文提到，字符编码是一套规则。既然是规则，就必须有标准。下面我就仔细说说常见的字符编码标准。

1.拉丁编码

ASCII的全称是American Standard Code for Information Interchange（美国信息交换标准代码）。顾名思义，这是现代计算机的发明国美国人设计的标准，而美国是一个英语国家，他们设定的**ASCII编码也只支持基础拉丁字符**。ASCII的设计也很简单，**用一个字节（8个位）来表示一个字符，并保证最高位的取值永远为’0’**。即表示字符含义的位数为7位，不难算出其可表达字符数为2^7=128个。这128个字符包括95个可打印的字符（涵盖了26个英文字母的大小写

以及英文标点符号能）与33个控制字符（不可打印字符）。例如下表，就是几个简单的规则对应：

字符类型	字符	二进制	
可打印字符	A	01000001	0›
可打印字符	a	01100001	0›
控制字符	r	00001101	0›
控制字符	n	00001010	0›

前面说到了，ASCII是美国人设计的，只能支持基础拉丁字符，而当计算机发展到欧洲，欧洲其它不只是用的基础拉丁字符的国家（即用更大的派生拉丁字符集）该怎么办呢？

当然，最简单的办法就是将美国人没有用到的**第8位也用上**就好了，这样能表达的字符个数就达到了2^8=256个，相比较原来，增长了一倍，这个编码规则也常被称为**EASCII**。EASCII基本解决了整个西欧的字符编码问题。但是对于欧洲其它地方如北欧，东欧地区，256个字符还是不够用，如是出现了**ISO 8859**,为解决256个字符不够用的问题，**ISO 8859采取的不再是单个独立的编码规则，而是由一系列的字符集（共15个）所组成**，分别称为ISO 8859-n(n=1,2,3…11,13…16,没有12)。其每个字符集对应不同的语言,如ISO 8859-1对应西欧语言，ISO 8859-2对应中欧语言等。其中大家所熟悉的**Latin-1就是ISO 8859-1的别名,它表示整个西欧的字符集范围。需要注意的一点的是，ISO 8859-n与ASCII是兼容的，即其0000000(0x00)-01111111(0x7f)范围段与ASCII保持一致，**

而**10000000（0x80）-11111111(0xFF)范围段被扩展用到不同的字符集。**

2.中文编码

以上我们接触到的拉丁编码，都是单字节编码，即用一个字节来对应一个字符。但这一规则对于其它字符集更大的语言来说，并不适应，比如中文，而是出现了用多个字节表示一个字符的编码规则。常见的中文GB2312（国家简体中文字符集）就是用两个字节来表示一个汉字（注意是表示一个汉字，对于拉丁字母，GB2312还是是用一个字节来表示以兼容ASCII）。我们用下表来说明各中文编码之间的规则和兼容性。

	GB2312	BIG5	GBK	GB18030
作用	国家简体中文字符集	统一繁体字符集	GB2312的扩展，加入对繁体字的支持	中日韩文字的编码
字节数	变字节： 1字节-兼容ASCII 2字节-汉字	2字节	2字节	变字节： 1字节-兼容ASCII 2字节，4字节
范围	能表示7445个符号，包括6763个汉字	能表示21886个符号	能表示21886个符号	能表示27484个符号
兼容性	兼容ASCII 00-7F范围内是一位，和ASCII保持一致	兼容ASCII，但与GB2312有冲突	兼容GB2312	兼容GB2312

对于中文编码，其规则实现上是很简单的，一般都是简单的**字符查表**即可，重要的是要注意其相互之间的**兼容性**问题。如如果选择BIG5字符集编码，就不能很好的兼容GB2312，当做繁转简时有可能导致

个别字的冲突与不一致，但是GBK与GB2312之间就不存在这样的问题。

3.Unicode

以上可以看到，针对不同的语言采用不同的编码，有可能导致冲突与不兼容性，如果我们打开一份字节序文件，如果不知道其编码规则，就无法正确解析其语义，这也是产生乱码的根本原因。有没有一种规则是全世界字符统一的呢？当然有，Unicode就是一种。为了能独立表示世界上所有的字符，Unicode采用**4个字节表示一个字符**,这样理论上Unicode能表示的字符数就达到了2^{31}= 2147483648 = 21 亿左右个字符，完全可以涵盖世界上一切语言所用的符号。我们以汉字”微信“两字举例说明：

```
微 <-> u5fae  <-> 00000000 00000000 01011111 10101110
信 <-> u4fe1  <-> 00000000 00000000 01001111 11100001
```

容易从上面的例子里看出，Unicode对所有的字符编码均需要四个字节，而这对于拉丁字母或汉字来说是浪费的，其前面三个或两个字节均是0,这对信息存储来说是极大的浪费。另外一个问题就是，如何区分Unicode与其它编码这也是一个问题，比如计算机怎么知道四个字节表示一个Unicode中的字符，还是分别表示四个ASCII的字符呢？

以上两个问题，困扰着Unicode，让Unicode的推广上一直面临着困难。直至UTF-8作为Unicode的一种实现后，部分问题得到解决，才得以完成推广使用。说到此，我们可以回答文章一开始提出的问题了，

UTF-8是Unicode的一种实现方式，而Unicode是一个统一标准规范，Unicode的实现方式除了UTF-8还有其它的，比如UTF-16等。

话说当初大牛Ben Thomson吃饭时，在一张餐巾纸上，设计出了UTF-8，然后回到房间，实现了第一版的UTF-8。关于UTF-8的基本规则，其实简单来说就两条（来自阮一峰老师的总结）：

规则1：对于单字节字符，字节的第一位为0，后7位为这个符号的Unicode码，所以对于拉丁字母，UTF-8与ASCII码是一致的。 规则2：对于n字节(n>1)的字符，第一个字节前n位都设为1，第n+1位为0，后面字节的前两位一律设为10， 剩下没有提及的位，全部为这个符号的Unicode编码。

通过，根据以上规则，可以建立一个Unicode取值范围与UTF-8字节序表示的对应关系，如下表，

Unicode 符号范围(十六进制)	UTF-8 编码方式(二进制)
单字节：0000 0000–0000 007F	0xxxxxxx
双字节：0000 0080–0000 07FF	110xxxxx 10xxxxxx
三字节：0000 0800–0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
四字节：0001 0000–0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

举例来说，’微’的Unicode是’u5fae’，二进制表示是”00000000 00000000 01011111 10101110“，其取值就位于’0000 0800-0000

FFFF’之间，所以其UTF-8编码为’**111**00101**10**1111110**10**101110’（加粗部分为固定编码内容）。

通过以上简单规则，UTF-8采取变字节的方式，解决了我们前文提到的关于Unicode的两大问题。同时，作为中文使用者需要注意的一点是**Unicode(UTF-8)与GBK，GB2312这些汉字编码规则是完全不兼容的，也就是说这两者之间不能通过任何算法来进行转换,如需转换，一般通过GBK查表的方式来进行**。

常见问题及解答

1.windows Notepad中的编码ANSI保存选项，代表什么含义？

ANSI是windows的默认的编码方式，对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对Windows简体中文版，如果是繁体中文版会采用Big5码）。所以，**如果将一个UTF-8编码的文件，另存为ANSI的方式，对于中文部分会产生乱码**。

2.什么是UTF-8的BOM？

BOM的全称是Byte Order Mark，BOM是微软给UTF-8编码加上的，用于标识文件使用的是UTF-8编码，即在UTF-8编码的文件起始位置，加入三个字节“EE BB BF”。这是微软特有的，标准并不推荐包含BOM的方式。采用加BOM的UTF-8编码文件，对于一些只支持标准UTF-8编码的环境，可能导致问题。比如，在Go语言编程中，对于包含BOM的代码文件，会导致编译出错。详细可见我的这篇文章。

3.为什么数据库Latin1字符集（单字节）可以存储中文呢？

其实不管需要使用几个字节来表示一个字符，但最小的存储单位都是字节,所以，**只要能保证传输和存储的字节顺序不会乱即可**。作为数据库，只是作为存储的使用的话，只要能保证存储的顺序与写入的顺序一致，然后再按相同的字节顺序读出即可，翻译成语义字符的任务交给应用程序。比如'微'的UTF-8编码是'0xE5 0xBE 0xAE'，那数据库也存储'0xE5 0xBE 0xAE'三个字节，其它应用按顺序从数据库读取，再按UTF-8编码进行展现。这当然是一个看似完美的方案，但是只要写入，存储，读取过程中岔出任何别的编码，都可能导致乱码。

4.Mysql数据库中多个字符集变量（其它数据库其实也类似），它们之间分别是什么关系？

```
mysql> show variables like 'character_set_%';
+--------------------------+----------------------------+
| Variable_name            | Value                      |
+--------------------------+----------------------------+
| character_set_client     | utf8                       |
| character_set_connection | utf8                       |
| character_set_database   | latin1                     |
| character_set_filesystem | binary                     |
| character_set_results    | utf8                       |
| character_set_server     | latin1                     |
| character_set_system     | utf8                       |
| character_sets_dir       | /usr/share/mysql/charsets/ |
+--------------------------+----------------------------+
8 rows in set (0.00 sec)
```

我们分别解释：

character_set_client：客户端来源的数据使用的字符集，用于客户端显式告诉客户端所发送的语句中的的字符编码。

character_set_connection：连接层的字符编码，mysql一般用character_set_connection将客户端的字符转换为连接层表示的字符。

character_set_results:查询结果从数据库读出后，将转换为character_set_results返回给前端。

而我们常见的解决乱码问题的操作：

```
mysql_query('SET NAMES GBK')
```

其相当于将以上三个字符集统一全部设置为GBK，这三者一致时，一般就解决了乱码问题。

character_set_database:当前选中数据库的默认字符集，如当create table时没有指定字符集，将默认选择该字符集。

character_set_database已经character_set_system，一般用于数据库系统内部的一些字符编码，处理数据乱码问题时，我们基本可以忽略。

5.什么情况下，表示信息丢失？

对于mysql数据库，我们可以通过**hex(colname)**函数（其它数据库也有类似的函数，一些文本文件编辑器也具有这个功能），查看实际存储的字节内容，如：

```
mysql> select hex(name) from t;
+------------------+
| hex(name)        |
+------------------+
| 6E616D6531303031 |
| 6E616D65313031   |
| 6E616D65313031   |
| 6E616D65313031   |
| 6E616D65313031   |
| 6E616D65313031   |
| 6E616D65313031   |
| 6E616D65313031   |
+------------------+
8 rows in set (0.00 sec)
```

通过查看存储的字节序，我们可以从根本上了解存储的内容是什么编码了。而当发现**存储的内容全部是'3F'时，就表明存储的内容由于编码问题，信息已经丢失了，无法再找回**。

之所以出现这种信息丢失的情况，一般是将不能相互转换的字符集之间做了转换，比如我们在前文说到，UTF-8只能一个个字节地变成Latin-1，但是根本不能转换的，因为两者之间没有转换规则，Unicode的字符对应范围也根本不在Latin-1范围内，所以只能用'?(0x3F)'代替了。

总结：

本文从基础知识与实际中碰到的问题上，解析了字符编码相关内容。而之所以要从头介绍字符编码的基础知识，是为了更好的从原理上了

解与解决日常碰到的编码问题，只有从根本上了解了不同字符集的规则及其之间的关系与兼容性，才能更好的解决碰到的乱码问题，也能避免由于程序中不正确的编码转换导致的信息丢失问题。