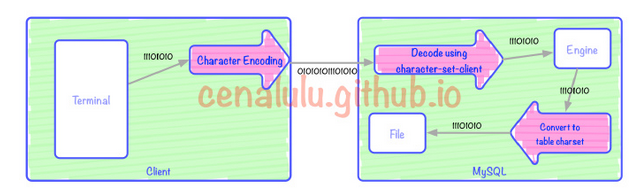
理解和解决MySQL乱码问题

# MySQL存储数据过程

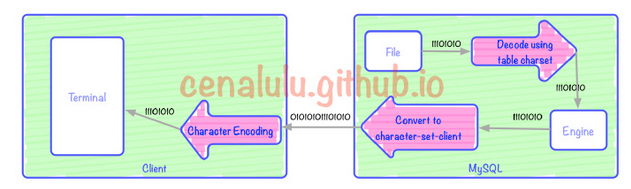
## 存入MySQL经历的编码转换过程



上图中有3次编码/解码的过程（红色箭头）。三个红色箭头分别对应：客户端编码，MySQL Server解码，Client编码向表编码的转换。其中Terminal可以是一个Bash，一个web页面又或者是一个APP。本文中我们假定Bash是我们的Terminal，即用户端的输入和展示界面。图中每一个框格对应的行为如下：

* 在terminal中使用输入法输入
* terminal根据字符编码转换成二进制流
* 二进制流通过MySQL客户端传输到MySQL Server
* Server通过character-set-client解码
* 判断character-set-client和目标表的charset是否一致
* 如果不一致则进行一次从client-charset到table-charset的一次字符编码转换
* 将转换后的字符编码二进制流存入文件中

## MySQL表中取出数据经历的编码转换过程



上图有3次编码/解码的过程（红色箭头）。上图中三个红色箭头分别对应：客户端解码展示，MySQL Server根据character-set-client编码，表编码向character-set-client编码的转换。

* 从文件读出二进制数据流
* 用表字符集编码进行解码
* 将数据转换为character-set-client的编码,使用character-set-client编码为二进制流
* Server通过网络传输到远端client
* client通过bash配置的字符编码展示查询结果

# 造成MySQL乱码的原因

## 存入和取出时对应环节的编码不一致

这个会造成乱码是显而易见的。我们把存入阶段的三次编解码使用的字符集编号为C1,C2,C3（图一从左到右）；取出时的三个字符集依次编号为C1’,C2’,C3’（从左到右）。那么存入的时候bash C1用的是UTF-8编码，取出的时候,C1'我们却使用了windows终端（默认是GBK编码），那么结果几乎一定是乱码。又或者存入MySQL的时候set names utf8(C2)，而取出的时候却使用了set names gbk(C2')，那么结果也必然是乱码

## 单个流程中三步的编码不一致

即上面任意一幅图中的同方向的三步中，只要两步或者两部以上的编码有不一致就有可能出现编解码错误。如果差异的两个字符集之间无法进行无损编码转换（下文会详细介绍），那么就一定会出现乱码。例如：我们的shell是UTF8编码，MySQL的character-set-client配置成了GBK，而表结构却又是charset=utf8，那么毫无疑问的一定会出现乱码。

|  |
| --- |
| master [localhost] {msandbox} (test) > create table charset\_test\_utf8 (id int primary key auto\_increment, char\_col varchar(50)) charset = utf8;  Query OK, 0 rows affected (0.04 sec)    master [localhost] {msandbox} (test) > set names gbk;  Query OK, 0 rows affected (0.00 sec)    master [localhost] {msandbox} (test) > insert into charset\_test\_utf8 (char\_col) values ('中文');  Query OK, 1 row affected, 1 warning (0.01 sec)    master [localhost] {msandbox} (test) > show warnings;  +---------+------+---------------------------------------------------------------------------+  | Level   | Code | Message                                                                   |  +---------+------+---------------------------------------------------------------------------+  | Warning | 1366 | Incorrect string value: '\xAD\xE6\x96\x87' for column 'char\_col' at row 1 |  +---------+------+---------------------------------------------------------------------------+  1 row in set (0.00 sec)    master [localhost] {msandbox} (test) > select id,hex(char\_col),char\_col from charset\_test\_utf8;  +----+----------------+----------+  | id | hex(char\_col)  | char\_col |  +----+----------------+----------+  |  1 | E6B6933FE69E83 | �?��        |  +----+----------------+----------+  1 row in set (0.01 sec) |

# 关于MySQL的编/解码

既然系统之间是按照二进制流进行传输的，那直接把这串二进制流直接存入表文件就好啦。为什么在存储之前还要进行两次编解码的操作呢？

Client to Server的编解码的原因是：MySQL需要对传来的二进制流做语法和词法解析。如果不做编码解析和校验，我们甚至没法知道传来的一串二进制流是insert还是update。

File to Engine的编解码是：为知道二进制流内的分词情况。举个简单的例子：我们想要从表里取出某个字段的前两个字符，执行了一句形如select left(col,2) from table的语句，存储引擎从文件读入该column的值是E4B8ADE69687。那么这个时候如果我们按照GBK把这个值分割成E4B8,ADE6,9687三个字，并那么返回客户端的值就应该是E4B8ADE6；如果按照UTF8分割成E4B8AD,E69687，那么就应该返回E4B8ADE69687两个字。可见，如果在从数据文件读入数据后，不进行编解码的话在存储引擎内部是无法进行字符级别的操作的。

## 关于错进错出

在MySQL中最常见的乱码问题的起因就是把错进错出神话。所谓的错进错出就是，客户端(web或shell)的字符编码和最终表的字符编码格式不同，但是只要保证存和取两次的字符集编码一致就仍然能够获得没有乱码的输出的这种现象。但是，错进错出并不是对于任意两种字符集编码的组合都是有效的。我们假设客户端的编码是C，MySQL表的字符集编码是S。那么为了能够错进错出，需要满足以下两个条件：

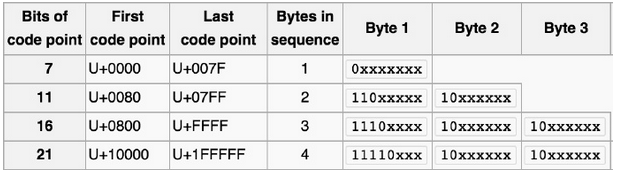
* MySQL接收请求时，从C编码后的二进制流在被S解码时能够无损
* MySQL返回数据是，从S编码后的二进制流在被C解码时能够无损

## 编码无损转换

那么什么是有损转换，什么是无损转换呢？假设我们要把用编码A表示的字符X，转化为编码B的表示形式，而编码B的字形集中并没有X这个字符，那么此时我们就称这个转换是有损的。

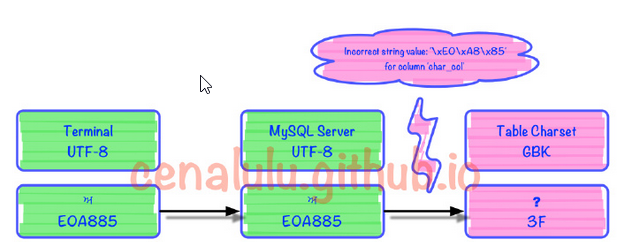
每个字符集所支持的字符数量是有限的，并且各个字符集涵盖的文字之间存在差异。UTF8和GBK所能表示的字符数量范围如下：

* GBK单个字符编码后的取值范围是：8140 - FEFE 其中不包括\*\*7E，总共字符数在27000左右
* UTF8单个字符编码后，按照字节数的不同，取值范围如下表：



由于UTF-8编码能表示的字符数量远超GBK。那么我们很容易就能找到一个从UTF8到GBK的有损编码转换。在MySQL中存储的具体情况如下：

|  |
| --- |
| master [localhost] {msandbox} (test) > create table charset\_test\_gbk (id int primary key auto\_increment, char\_col varchar(50)) charset = gbk;  Query OK, 0 rows affected (0.00 sec)    master [localhost] {msandbox} (test) > set names utf8;  Query OK, 0 rows affected (0.00 sec)    master [localhost] {msandbox} (test) > insert into charset\_test\_gbk (char\_col) values ('ਅ');  Query OK, 1 row affected, 1 warning (0.01 sec)    master [localhost] {msandbox} (test) > show warnings;  +---------+------+-----------------------------------------------------------------------+  | Level   | Code | Message                                                               |  +---------+------+-----------------------------------------------------------------------+  | Warning | 1366 | Incorrect string value: '\xE0\xA8\x85' for column 'char\_col' at row 1 |  +---------+------+-----------------------------------------------------------------------+  1 row in set (0.00 sec)    master [localhost] {msandbox} (test) > select id,hex(char\_col),char\_col,char\_length(char\_col) from charset\_test\_gbk;  +----+---------------+----------+-----------------------+  | id | hex(char\_col) | char\_col | char\_length(char\_col) |  +----+---------------+----------+-----------------------+  |  1 | 3F            | ?        |                     1 |  +----+---------------+----------+-----------------------+  1 row in set (0.00 sec) |

出错的部分是在编解码的第3步时发生的。具体见下图：  


可见MySQL内部如果无法找到一个UTF8字符所对应的GBK字符时，就会转换成一个错误mark（这里是问号）。而每个字符集在程序实现的时候内部都约定了当出现这种情况时的行为和转换规则。例如：UTF8中无法找到对应字符时，如果不抛错那么就将该字符替换成� (U+FFFD)

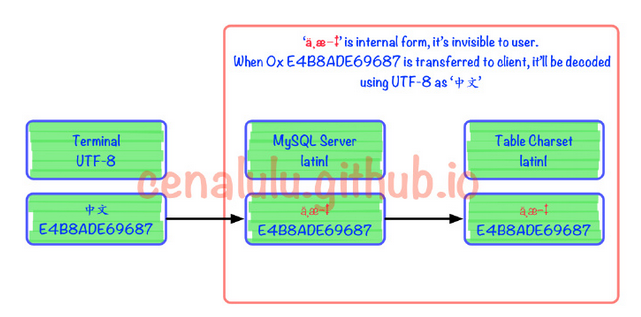
那么是不是任何两种字符集编码之间的转换都是有损的呢？并非这样，转换是否有损取决于以下几点：

* 被转换的字符是否同时在两个字符集中
* 目标字符集是否能够对不支持字符，保留其原有表达形式

关于第一点，刚才已经通过实验来解释过了。这里来解释下造成有损转换的第二个因素。从刚才的例子我们可以看到由于GBK在处理自己无法表示的字符时的行为是：用错误标识替代，即0x3F。而有些字符集（例如latin1）在遇到自己无法表示的字符时，会保留原字符集的编码数据，并跳过忽略该字符进而处理后面的数据。如果目标字符集具有这样的特性，那么就能够实现这节最开始提到的错进错出的效果。

|  |
| --- |
| master [localhost] {msandbox} (test) > create table charset\_test (id int primary key auto\_increment, char\_col varchar(50)) charset = latin1;  Query OK, 0 rows affected (0.03 sec)    master [localhost] {msandbox} (test) > set names latin1;  Query OK, 0 rows affected (0.00 sec)    master [localhost] {msandbox} (test) > insert into charset\_test (char\_col) values ('中文');  Query OK, 1 row affected (0.01 sec)    master [localhost] {msandbox} (test) > select id,hex(char\_col),char\_col from charset\_test;  +----+---------------+----------+  | id | hex(char\_col) | char\_col |  +----+---------------+----------+  |  2 | E4B8ADE69687  | 中文     |  +----+---------------+----------+  2 rows in set (0.00 sec) |

具体流程图如下。可见在被MySQL Server接收到以后实际上已经发生了编码不一致的情况。但是由于Latin1字符集对于自己表述范围外的字符不会做任何处理，而是保留原值。这样的行为也使得错进错出成为了可能。



## 如何避免乱码

理解了上面的内容，要避免乱码就显得很容易了。只要做到"三位一体"，即客户端，MySQL character-set-client，table charset三个字符集完全一致就可以保证一定不会有乱码出现了。

## 修复已经编码损坏的数据

将二进制数据作为中间数据的做法来实现的。由于，MySQL再将有编码意义的数据流，转换为无编码意义的二进制数据的时候并不做实际的数据转换。而从二进制数据准换为带编码的数据时，又会用目标编码做一次编码转换校验。通过这两个特性就相当于在MySQL内部模拟了一次"错出"，将乱码"拨乱反正"了。还是用上面那个例子举例，我们用UTF-8将数据"错进"到latin1编码的表中。现在需要将表编码修改为UTF-8可以使用以下命令。

|  |
| --- |
| mysql> ALTER TABLE charset\_test\_latin1 MODIFY COLUMN char\_col VARBINARY(50);  mysql> ALTER TABLE charset\_test\_latin1 MODIFY COLUMN char\_col varchar(50) character set utf8; |