# 字符集

## 概述

通过命令SHOW CHARSET可以查看MySQL数据库支持的字符集。

对于简体中文，我们习惯使用gbk或gb2312，这两者的区别是：gbk是gb2312的超集，因此可以支持更多的汉字。不过当前的MySQL不支持中文字符集gb18030，因此在有些应用中已经出现gbk不能显示特定中文字体的情况了。而对于繁体中文，big5可能是首选的字符集。

上面叙述的字符集gbk、gb2312和big5都是假设用户的应用程序只涉及这些字符范围所规定的字符，不需要额外的字符。而对于Facebook、Google、Yahoo和网易这些国际化公司，需要存储的字符可能是多种多样的，因此要使用Unicode编码的字符集。另外，为了平台的扩展性，有必要将字符集设置为Unicode编码。例如，笔者之前所在的网游公司成功地将游戏输出到韩国，并在当地取得了不小的成功，不过移植到韩国版本时，程序员花费了很大的精力来修改原字符集所带来的问题。因为原来使用的字符集是gbk，显然不能满足韩国玩家的要求。如果最初设计时采用utf8字符集，就不会有这个烦恼了。

## Unicode

此外，很多程序员和DBA不能区分Unicode和utf8的区别，他们总是认为两者是等价的，即Unicode就是utf8，utf8就是Unicode。其实，两者还是存在很大区别的。这里介绍一下这两者的区别。

Unicode是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一且唯一的二进制编码，以满足跨语言和跨平台进行文本转换和处理的要求。Unicode是1990年开始研发，1994年正式公布的。随着计算机工作能力的增强，Unicode在面世后的十多年里得到普及。需要注意的是，Unicode是字符编码，不是字符集。

Unicode是基于通用字符集（Universal Character Set）的标准进行发展的，同时以书本的形式（The Unicode Standard，目前第五版由Addison-Wesley Professional出版）对外发表。2006年7月的最新版Unicode是5.0版本。

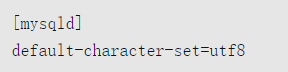
Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode用数字0～0x10FFFF来映射这些字符，最多可以容纳1114112个字符，或者说有1114112个码位。码位就是可以分配给字符的数字。utf8、utf16和utf32都是将数字转换到程序数据的编码方案。

我们可以通过下面的SQL语句来查询MySQL支持的Unicode编码的字符集。

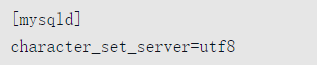
MySQL 5.5数据库共支持ucs2、utf8（utf8mb3）、utf8mb4、utf16，以及utf32五种Unicode编码，而5.5之前的版本只支持ucs2和utf8两种Unicode字符集，即只支持Unicode 3.0的标准。显然MySQL 5.5版本对于Unicode标准已经支持到5.0。utf8目前被视为utf8mb3，即最大占用3个字节空间，而utf8mb4可以视做utf8mb3的扩展。对BMP（Basic Multilingual Plane）字符的存储，utf8mb3和utf8mb4两者是完全一样的，区别只是utf8mb4对扩展字符的支持。

对于Unicode编码的字符集，强烈建议将所有的CHAR字段设置为VARCHAR字段，因为对于CHAR字段，数据库会保存最大可能的字节数。例如，对于CHAR（30），数据库可能存储90字节的数据。

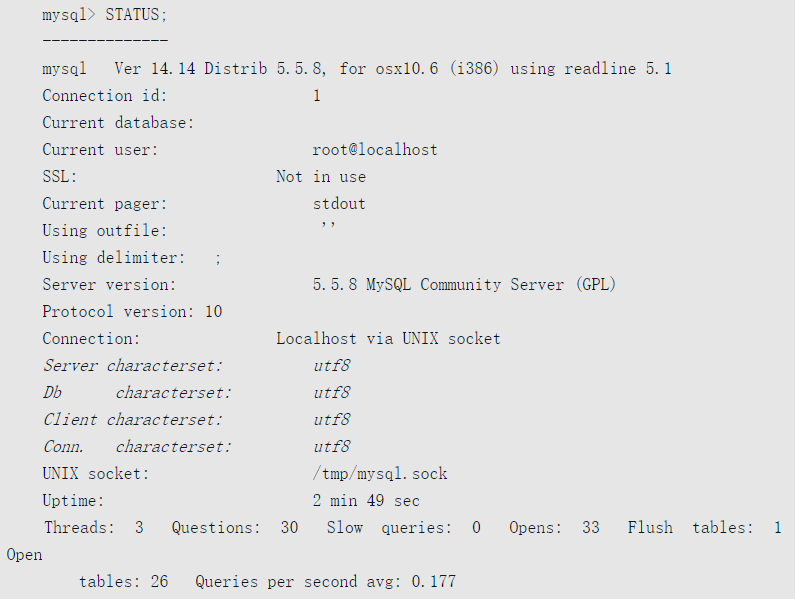
对字符集的设置可以在MySQL的配置文件中完成，例如：



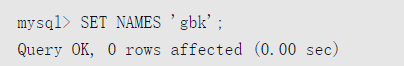
MySQL 5.5版本开始移除了参数default\_character\_set，取而代之的是参数character\_set\_server，因此在配置文件中需进行如下设置：



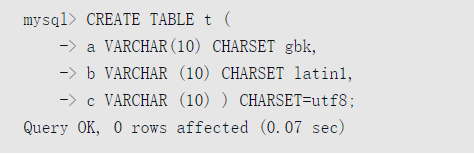
要查看当前使用的字符集，可以使用STATUS命令：



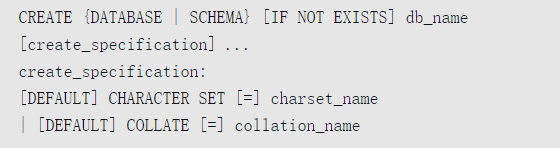
命令SET NAMES可以用来更改当前会话连接的字符集、当前会话的客户端的字符集，以及当前会话返回结果集的字符集，例如：



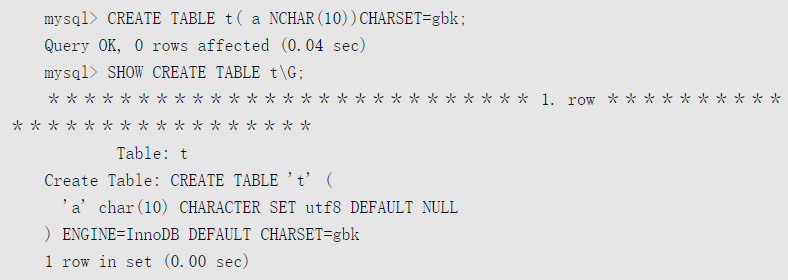
MySQL数据库一个比较“强悍”的地方是，可以细化每个对象字符集的设置，例如：



可以看到，我们创建了表t，a列的字符集是gbk，b列的字符集是latin1，c列的字符集没有在定义时指定，因此采用定义表时的字符集，这里是utf8。如果定义表时没有给出具体的字符集，则采用创建架构（Schema，也称库）时指定的字符集。如果没有在创建架构时指定字符集，则使用数据库配置文件中指定的字符集。创建架构的时候可以指定架构的字符集及排序规则，过程如下：



SQL标准支持NCHAR类型（即National Character Set），在MySQL 5.5数据库中使用utf8来表示这种类型，例如：



可以看到，我们在创建表时将a列定义为NCHAR（10），但是在创建数据库过程中实际使用的是utf8字符集。在别的数据库中，如Microsoft SQL Server，NCHAR可能被视为ucs2字符集。我们在客户端使用N前缀将字符串指定为NCHAR类型，也就是UTF-8类型，例如：



## utf8

## utf8mb4

MySQL在5.5.3之后增加了utf8mb4字符编码，mb4即most bytes 4。简单说utf8mb4是utf8的超集并完全兼容utf8，能够用四个字节存储更多的字符。

但抛开数据库，标准的UTF-8字符集编码是可以用 1~4 个字节去编码21位字符，这几乎包含了是世界上所有能看见的语言了。然而在MySQL里实现的utf8最长使用3个字节，也就是只支持到了 Unicode 中的 基本多文本平面（U+0000至U+FFFF），包含了控制符、拉丁文，中、日、韩等绝大多数国际字符，但并不是所有，最常见的就算现在手机端常用的表情字符emoji和一些不常用的汉字，如“墅”，这些需要四个字节才能编码出来。

注：QQ里面的内置的表情不算，它是通过特殊映射到的一个gif图片。一般输入法自带的就是。

也就是当你的数据库里要求能够存入这些表情或宽字符时，可以把字段定义为 utf8mb4，同时要注意连接字符集也要设置为utf8mb4，否则在 严格模式 下会出现 Incorrect string value: /xF0/xA1/x8B/xBE/xE5/xA2… for column 'name'这样的错误，非严格模式下此后的数据会被截断。

提示：另外一种能够存储emoji的方式是，不关心数据库表字符集，只要连接字符集使用 latin1，但相信我，你绝对不想这个干，一是这种字符集混用管理极不规范，二是存储空间被放大（读者可以想下为什么）。

**utf8转换为utf8mb4**

**“伪”转换**

如果你的表定义和连接字符集都是utf8，那么直接在你的表上执行

ALTER TABLE tbl\_name CONVERT TO CHARACTER SET utf8mb4;

则能够该表上所有的列的character类型变成 utf8mb4，表定义的默认字符集也会修改。连接的时候需要使用set names utf8mb4便可以插入四字节字符。（如果依然使用 utf8 连接，只要不出现四字节字符则完全没问题）。

上面的 convert 有两个问题，一是它不能ONLINE，也就是执行之后全表禁止修改，有关这方面的讨论见 mysql 5.6 原生Online DDL解析；二是，它可能会自动该表字段类型定义，如 VARCHAR 被转成 MEDIUMTEXT，可以通过 MODIFY 指定类型为原类型。

另外 ALTER TABLE tbl\_name DEFAULT CHARACTER SET utf8mb4 这样的语句就不要随便执行了，特别是当表原本不是utf8时，除非表是空的或者你确认表里只有拉丁字符，否则正常和乱的就混在一起了。

最重要的是，你连接时使用的latin1字符集写入了历史数据，表定义是latin1或utf8，不要期望通过 ALTER ... CONVERT ... 能够让你达到用utf8读取历史中文数据的目的，没卵用，老老实实做逻辑dump。所以我才叫它“伪”转换

**character-set-server**

一旦你决定使用utf8mb4，强烈建议你要修改服务端 character-set-server=utf8mb4，不同的语言对它的处理方法不一样，c++, php, python可以设置character-set，但java驱动依赖于 character-set-server 选项。

同时还要谨慎一些特殊选项，如遇到腾讯云CDB连接字符集设置一个坑。个人不建议设置全局init\_connect。

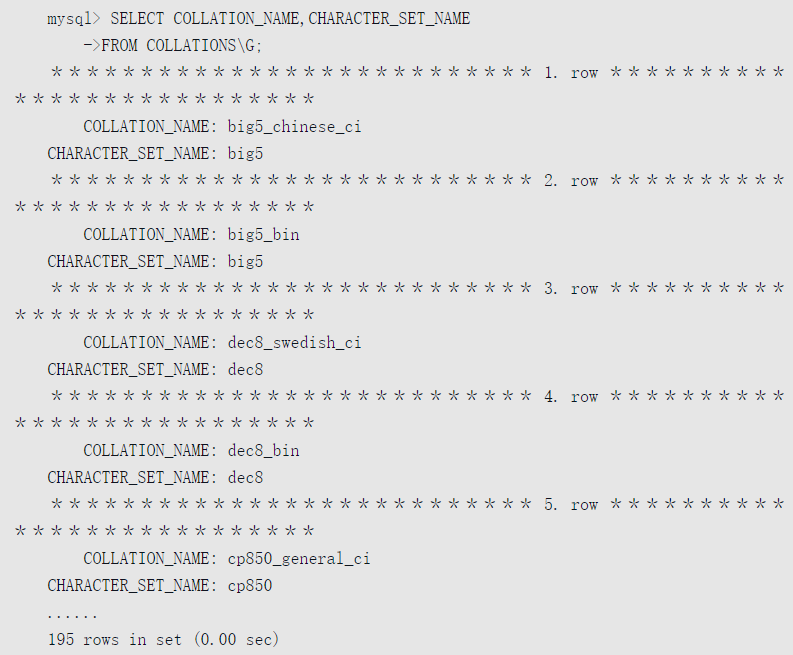
# 校对集

## 概述

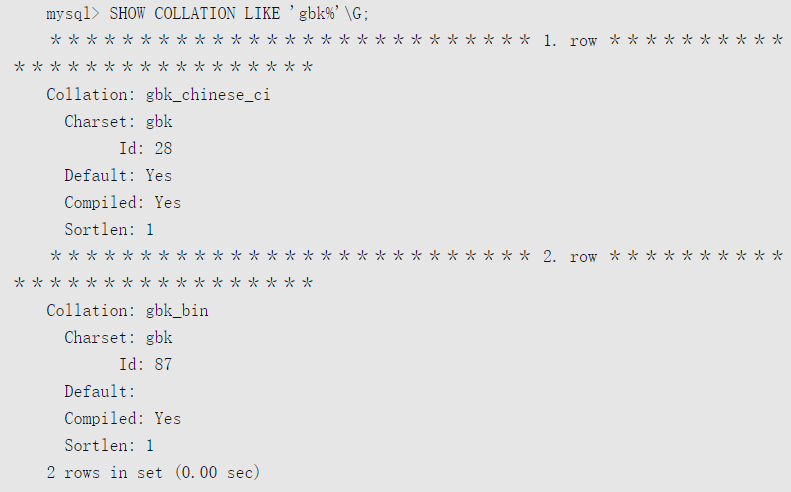
排序规则（Collation）是指对指定字符集下不同字符的比较规则。其特征有以下几点：

1. 两个不同的字符集不能有相同的排序规则。
2. 每个字符集有一个默认的排序规则。
3. 有一些常用的命名规则。如\_ci结尾表示大小写不敏感（case insensitive），\_cs表示大小写敏感（case sensitive），\_bin表示二进制的比较（binary）。

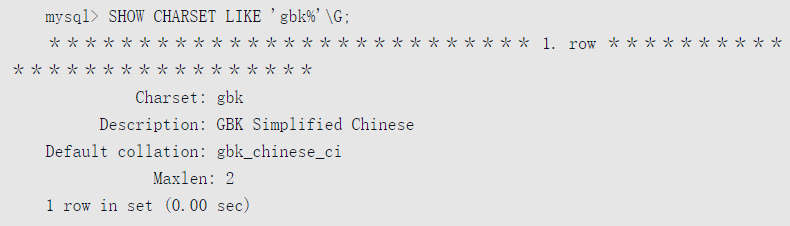
在MySQL数据库中，可以通过命令SHOW COLLATION来查看支持的各种排序规则，也可以通过information\_schema架构下的表COLLATIONS来查看，例如：



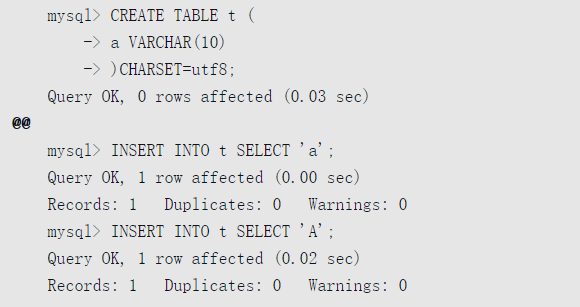
MySQL 5.5数据库支持195种排序规则，这里只列出了一部分排序规则。当然，也可以通过SHOWCOLLATION LIKE命令来过滤想要查看的排序规则，过程如下：



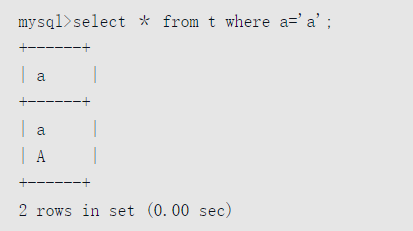
可以看到，对于gbk字符集，有两种排序规则，分别是gbk\_chinese\_ci和gbk\_bin。前面介绍的命令SHOW CHARSET已经显示了每种字符集的默认排序规则，如果用户需要查看gbk字符集的默认排序规则，那么可以使用如下命令：



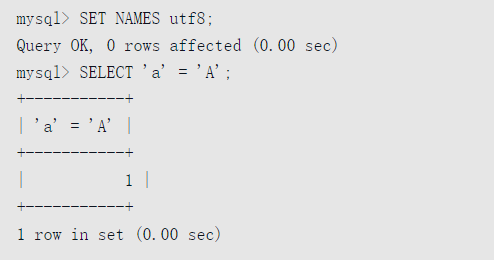
前面介绍了一些命令用来查看当前MySQL支持的排序规则，那排序规则到底有什么用，它对我们的SQL编程又会产生怎样的影响呢？下面通过一个例子来说明排序规则的重要性。首先创建一张表，再插入两条数据，过程如下：



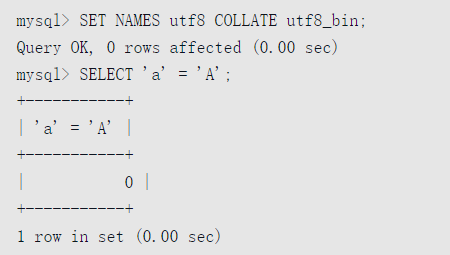
表t非常简单，只有一个列a是VARCHAR类型的。设置表的字符集为utf8，因此列a的字符集也是utf8。我们插入了两条数据，“a”和“A”，然后执行这样的SQL查询：



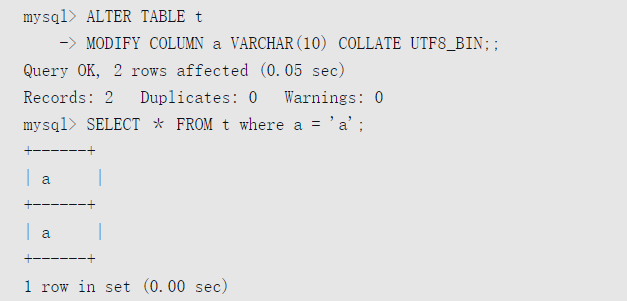
执行SQL查询语句的目的可能只是要寻找小写字符“a”，但是现在返回的却是两条记录，大写字符“A”也在返回结果集中。导致这个问题的原因是utf8字符集默认的排序规则是utf8\_general\_ci。之前介绍过\_ci结尾表示大小写不敏感，因此这个示例中的“a”和“A”被视为一致的字符而返回。在命令行中可以直接对这两个字符进行比较，例如：



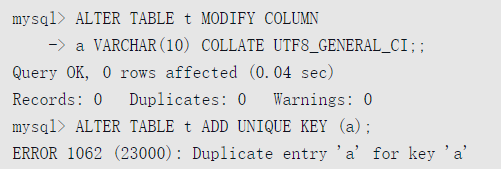
可以看到返回值为1，即认为两个字符的比较结果是相等的。如果需要更改当前会话的排序规则，可以通过命令SET NAMES... COLLATE...来实现，例如：



可以看到，这时数据库会认为“a”和“A”是不同的字符。对于创建的表t如果需要对a列区分大小字符，则可以将a列的排序规则修改为utf8\_bin，例如：



大小写敏感的需求要视应用程序的需求而定，并不总是要求区分大小写字符。例如，对于用户信息表，可能希望在用户注册时不要区分大小写，因为如果区分大小写，那么一个用户可以注册为David，另一个用户可以注册为david或者DaVid。这可能并不是我们通常想看到的一种情况。另外，排序规则不仅影响大小写的比较问题，也影响着索引。例如我们将t表的a列修改为之前的定义，然后再创建一个a列上的唯一索引。



可以看到，不能在a列上创建一个唯一索引，报错中提示有重复数据。索引是B+树，同样需要对字符进行比较，因此在建立唯一索引时由于排序规则对大小写不敏感而导致了错误。

在MySQL数据库中还有一种被称为binary的排序规则，其和\_bin的排序规则大致相同，有些小的区别读者可以在MySQL官方手册中进行查找。

## utf8mb4\_unicode\_ci /utf8mb4\_general\_ci

字符除了需要存储，还需要排序或比较大小，涉及到与编码字符集对应的排序字符集（collation）。ut8mb4对应的排序字符集常用的有 utf8mb4\_unicode\_ci、utf8mb4\_general\_ci，到底采用哪个在 stackoverflow 上有个讨论，What’s the difference between utf8\_general\_ci and utf8\_unicode\_ci

主要从排序准确性和性能两方面看：

**准确性**

utf8mb4\_unicode\_ci 是基于标准的Unicode来排序和比较，能够在各种语言之间精确排序

utf8mb4\_general\_ci 没有实现Unicode排序规则，在遇到某些特殊语言或字符是，排序结果可能不是所期望的。

但是在绝大多数情况下，这种特殊字符的顺序一定要那么精确吗。比如Unicode把ß、Œ当成ss和OE来看；而general会把它们当成s、e，再如ÀÁÅåāă各自都与 A 相等。

**性能**

utf8mb4\_general\_ci 在比较和排序的时候更快

utf8mb4\_unicode\_ci 在特殊情况下，Unicode排序规则为了能够处理特殊字符的情况，实现了略微复杂的排序算法。

但是在绝大多数情况下，不会发生此类复杂比较。general理论上比Unicode可能快些，但相比现在的CPU来说，它远远不足以成为考虑性能的因素，索引涉及、SQL设计才是。 我个人推荐是utf8mb4\_unicode\_ci，将来 8.0 里也极有可能使用变为默认的规则。

这也从另一个角度告诉我们，不要可能产生乱码的字段作为主键或唯一索引。我遇到过一例，以 url 来作为唯一索引，但是它记录的有可能是乱码，导致后来想把它们修复就特别麻烦。

# 字符集转换问题

## key 768 long错误

字符集从utf8转到utf8mb4之后，最容易引起的就是索引键超长的问题。

对于表行格式是 COMPACT或 REDUNDANT，InnoDB有单个索引最大字节数768 的限制，而字段定义的是能存储的字符数，比如 VARCHAR(200) 代表能够存200个汉字，索引定义是字符集类型最大长度算的，即 utf8 maxbytes=3, utf8mb4 maxbytes=4，算下来utf8和utf8mb4两种情况的索引长度分别为600 bytes和800bytes，后者超过了768，导致出错：Error 1071: Specified key was too long; max key length is 767 bytes。

COMPRESSED和DYNAMIC格式不受限制，但也依然不建议索引太长，太浪费空间和cpu搜索资源。

如果已有定义超过这个长度的，可加上前缀索引，如果暂不能加上前缀索引（像唯一索引），可把该字段的字符集改回utf8或latin1。

但是，（ 敲黑板啦，很重要 ），要防止出现 Illegal mix of collations (utf8\_general\_ci,IMPLICIT) and (utf8mb4\_general\_ci,COERCIBLE) for operation '=' 错误：连接字符集使用utf8mb4，但 SELECT/UPDATE where条件有utf8类型的列，且条件右边存在不属于utf8字符，就会触发该异常。表示踩过这个坑。

再多加一个友好提示：EXPLAIN 结果里面的 key\_len 指的搜索索引长度，单位是bytes，而且是以字符集支持的单字符最大字节数算的，这也是为什么 INDEX\_LENGTH 膨胀厉害的一个原因。

## C/C++内存空间分配问题

这是我们这边的开发遇到的一个棘手的问题。C或C++连接MySQL使用的是linux系统上的 libmysqlclient 动态库，程序获取到数据之后根据自定义的一个网络协议，按照mysql字段定义的固定字节数来传输数据。从utf8转utf8mb4之后，c++里面针对character单字符内存空间分配，从3个增加到4个，引起异常。

这个问题其实是想说明，使用utf8mb4之后，官方建议尽量用 varchar 代替 char，这样可以减少固定存储空间浪费（关于char与varchar的选择，可参考 这里）。但开发设计表时 varchar 的大小不能随意加大，它虽然是变长的，但客户端在定义变量来获取数据时，是以定义的为准，而非实际长度。按需分配，避免程序使用过多的内存。

## java驱动使用

Java语言里面所实现的UTF-8编码就是支持4字节的，所以不需要配置 mb4 这样的字眼，但如果从MySQL读写emoji，MySQL驱动版本要在 5.1.13 及以上版本，数据库连接依然是 characterEncoding=UTF-8 。

但还没完，遇到一个大坑。官方手册 里还有这么一段话：

Connector/J did not support utf8mb4 for servers 5.5.2 and newer.

Connector/J now auto-detects servers configured with character\_set\_server=utf8mb4 or treats the Java encoding utf-8 passed

using characterEncoding=... as utf8mb4 in the SET NAMES= calls it makes when establishing the connection. (Bug #54175)

意思是，java驱动会自动检测服务端 character\_set\_server 的配置，如果为utf8mb4，驱动在建立连接的时候设置 SET NAMES utf8mb4。然而其他语言没有依赖于这样的特性。

## 主从复制错误

这个问题没有遇到，只是看官方文档有提到，曾经也看到过类似的技术文章。

大概就是从库的版本比主库的版本低，导致有些字符集不支持；或者人工修改了从库上的表或字段的字符集定义，都有可能引起异常。

## join 查询问题

这个问题是之前在姜承尧老师公众号看到的一篇文章 “MySQL表字段字符集不同导致的索引失效问题”，自己也验证了一下，的确会有问题：

CREATE TABLE t1 (

f\_id varchar(20) NOT NULL,

f\_action char(25) NOT NULL DEFAULT '' COMMENT '',

PRIMARY KEY (`f\_id`),

) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW\_FORMAT=DYNAMIC;

CREATE TABLE t1\_copy\_mb4 (

f\_id varchar(20) CHARACTER SET utf8mb4 NOT NULL,

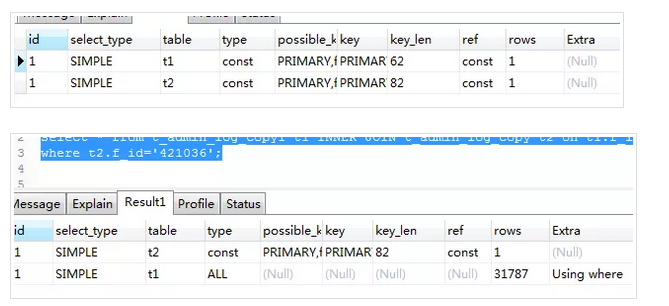
f\_action char(25) NOT NULL DEFAULT '' COMMENT '',

PRIMARY KEY (`f\_id`),

) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW\_FORMAT=DYNAMIC;

EXPLAIN extended select \* from t1 INNER JOIN t1\_copy\_mb4 t2 on t1.f\_id=t2.f\_id where t1.f\_id='421036';

EXPLAIN extended select \* from t1 INNER JOIN t1\_copy\_mb4 t2 on t1.f\_id=t2.f\_id where t2.f\_id='421036';

对应上面1,2 的截图：

其中 2 的warnings 有convert:

(convert(t1.f\_id using utf8mb4) = ‘421036’)

官网能找到这一点解释的还是开头那个地址：

Similarly, the following comparison in the WHERE clause works according to the collation of utf8mb4\_col:

SELECT \* FROM utf8\_tbl, utf8mb4\_tbl WHERE utf8\_tbl.utf8\_col = utf8mb4\_tbl.utf8mb4\_col;

只是索引失效发生在utf8mb4列 在条件左边。（关于MySQL的隐式类型转换，见这里）。