MySQL的优化

[1 概述 5](#_Toc791)

[2 设计 - 字段类型的选择 6](#_Toc10663)

[**2.1** 尽可能占用更少的存储空间 6](#_Toc27293)

[**2.2** 尽可能定长(占用固定的存储空间)数据类型 7](#_Toc11966)

[**2.3** 尽可能用整数代替字符串 8](#_Toc7811)

[3 设计 - 范式 Normal Format, 规范的格式 9](#_Toc26409)

[**3.1** 1NF, 第一范式, 满足原子性 9](#_Toc8196)

[**3.2** 2NF, 在满足第一范式的前提下, 消除对主键的部分依赖 10](#_Toc27905)

[**3.3** 3NF,第三范式,满足第二, 消除对主键传递依赖 11](#_Toc20367)

[**3.4** 为什么要满足范式? 13](#_Toc31430)

[**3.5** 逆范式优化 13](#_Toc12848)

[4 设计 - 存储引擎的选择 15](#_Toc2999)

[**4.1** 结论 15](#_Toc16127)

[**4.2** MyISAM 15](#_Toc26621)

[4.2.1 存储机制 15](#_Toc23855)

[4.2.2 特殊功能 17](#_Toc17155)

[4.2.3 并发支持 17](#_Toc16249)

[**4.3** Innodb 17](#_Toc1456)

[4.3.1 存储机制 17](#_Toc28659)

[4.3.2 特殊功能 19](#_Toc27491)

[4.3.3 并发支持 19](#_Toc14231)

[**4.4** show engines,查看支持的存储引擎 20](#_Toc4064)

[**4.5** memory, 缓存型存储引擎 20](#_Toc9987)

[**4.6** archive, 存档型 21](#_Toc15760)

[5 锁定 23](#_Toc29243)

[**5.1** 锁定类型 23](#_Toc1721)

[**5.2** 锁定粒度 23](#_Toc8049)

[**5.3** 加锁解锁 24](#_Toc18853)

[6 功能 - index-索引(最重要环节) 25](#_Toc18723)

[**6.1** 概述 25](#_Toc32462)

[**6.2** MySQL支持索引类型 25](#_Toc30220)

[6.2.1 普通索引, index 26](#_Toc27010)

[6.2.2 唯一索引, unique index(key) 26](#_Toc23997)

[6.2.3 主键索引, paimary key 26](#_Toc18085)

[6.2.4 全文索引, fulltext index 26](#_Toc26326)

[**6.3** 索引的管理(语法) 26](#_Toc12915)

[6.3.1 创建索引 26](#_Toc21564)

[6.3.2 查看索引, show 27](#_Toc1398)

[6.3.3 删除索引 27](#_Toc185)

[**6.4** explain – 获取执行计划 - 分析索引的使用 28](#_Toc10765)

[**6.5** 使用场景 31](#_Toc17643)

[6.5.1 where检索 31](#_Toc14128)

[6.5.2 order by排序 31](#_Toc16757)

[6.5.3 join, 连接查询时关联字段 32](#_Toc16985)

[6.5.4 查询字段使用索引，索引覆盖 34](#_Toc22465)

[**6.6** 使用语法原则 37](#_Toc10685)

[6.6.1 字段独立 37](#_Toc11741)

[6.6.2 左原则 38](#_Toc24885)

[6.6.3 OR原则 39](#_Toc15926)

[6.6.4 MySQL会自动选择 40](#_Toc24455)

[**6.7** 前缀索引 41](#_Toc14915)

[**6.8** 索引的数据结构(理论) 43](#_Toc30454)

[6.8.1 BTree索引 – 磁盘上存储索引的不二选择 43](#_Toc32598)

[6.8.2 聚簇索引, B+Tree 46](#_Toc31547)

[6.8.3 Hash索引 47](#_Toc5937)

[**6.9** 如何创建索引? 48](#_Toc26706)

[7 功能 - 查询缓存 query cache(了解) 50](#_Toc18964)

[**7.1** 开启查询缓存 50](#_Toc6349)

[**7.2** 设置查询缓存大小 50](#_Toc13914)

[**7.3** 客户端常规执行select查询即可 51](#_Toc20350)

[**7.4** 缓存失效 52](#_Toc16864)

[**7.5** 提示不使用缓存 53](#_Toc14268)

[8 分区,分表(分布式存储策略) 54](#_Toc7119)

[**8.1** 分区, partition 54](#_Toc22675)

[**8.2** 分区算法 56](#_Toc31100)

[8.2.1 hash, 均匀分配(求余) 56](#_Toc6807)

[8.2.2 key, 均匀分配(计算可以的整数值, 再求余) 57](#_Toc20342)

[8.2.3 list, 按业务逻辑分配(列表值条件) 57](#_Toc15670)

[8.2.4 range, 按业务逻辑分配(区间范围条件) 59](#_Toc4107)

[**8.3** 分区支持管理 59](#_Toc16320)

[**8.4** 建议的分表算法 61](#_Toc697)

[**8.5** 用户分表, 水平分表 61](#_Toc9080)

[**8.6** 分区和分表, 都叫水平分表 63](#_Toc9323)

[**8.7** 垂直分表 63](#_Toc22384)

[9 架构 – 一台MySQL不够, 需要多台MySQL协同工作 65](#_Toc15995)

[**9.1** 读写分离 65](#_Toc25595)

[**9.2** 主从复制 65](#_Toc4804)

[**9.3** 负载均衡 66](#_Toc23291)

[**9.4** 安装模拟 67](#_Toc11600)

[9.4.1 准备2台MySQL服务器 67](#_Toc31575)

[9.4.2 配置复制 67](#_Toc12385)

[9.4.3 配置读写分离和负载均衡代理软件 71](#_Toc10581)

[**9.5** 冷热数据分离 77](#_Toc21997)

[**9.6** 高可用, HA, high availiable 77](#_Toc10939)

[**9.7** 服务器配置 78](#_Toc31167)

[9.7.1 max\_connections, 最大连接数 78](#_Toc19669)

[9.7.2 innodb\_buffer\_pool\_size 79](#_Toc23005)

[9.7.3 key\_buffer\_size 79](#_Toc3032)

[10 SQL层面 80](#_Toc26340)

[**10.1** explain, 分析SQL的执行计划 80](#_Toc20026)

[**10.2** 慢SQL查询日志 80](#_Toc3190)

[10.2.1 开启慢查日志 80](#_Toc7162)

[10.2.2 设置时间的临界值 81](#_Toc23026)

[10.2.3 记录慢查询 82](#_Toc14572)

[**10.3** profile机制 82](#_Toc632)

[10.3.1 开启profile 83](#_Toc23548)

[10.3.2 执行SQL 83](#_Toc13444)

[10.3.3 获取概要信息 83](#_Toc5988)

[10.3.4 查看某条SQL详细执行时间 84](#_Toc5711)

[**10.4** limit优化 85](#_Toc7192)

[10.4.1 limit 1 85](#_Toc22413)

[10.4.2 limit 分页 85](#_Toc363)

[**10.5** order 86](#_Toc26378)

[10.5.1 order by rand(), 尽可能少用 86](#_Toc25782)

[10.5.2 order by null 88](#_Toc6028)

[**10.6** 单表对多表查询 89](#_Toc5416)

[11 Iterator告白 90](#_Toc19685)

[12 前++告白 94](#_Toc31536)

[13 global关键字告白 95](#_Toc31295)

# 概述

应用程序架构中, LAMP(LNMP), 最慢的一环, 都是数据库.

PHP:　业务逻辑程序都是相似的．　不会随着数据的增加而变得更加复杂．

 MySQL, 来说，　数据是不断的增加的，　同样的运算，　随着数据的增加而变慢！

如何提升数据库的效率？

通常需要从４个方面考虑．

* 设计方面：存储引擎的选择，字段类型的选项，实体间的关系设计，范式的选择
* 功能方面：索引，查询缓存，分表分区等等．
* 架构层面：（应用程序缓存），负载均衡集群，读写分离的处理，冷热数据分离．
* 应用层面：（应用程序缓存），ＳＱＬ应该如何写？多表联查，单表查询．尽可能避免哪些低效的ＳＱＬ．

# 设计 - 字段类型的选择

建表时, 往往需要考虑字段的类型的问题.

可优化性不强, 需要注意以下的几个原则

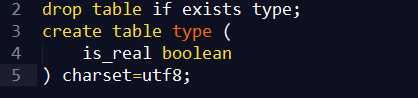
## 尽可能占用更少的存储空间

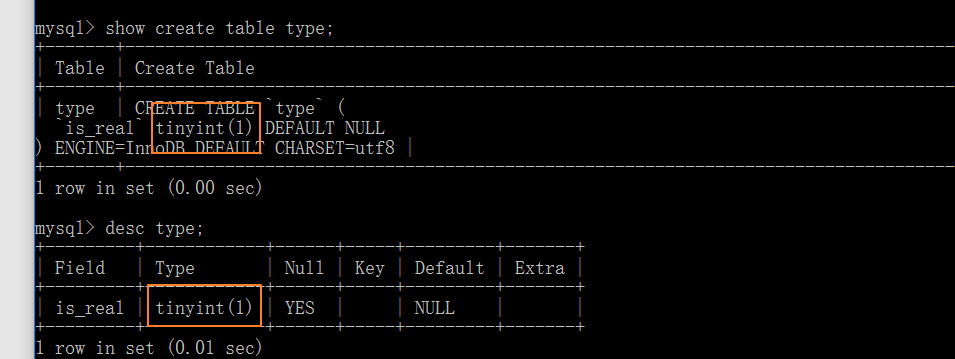
多少字节Byte!

tinyint:1, smallint 2, mediumint 3, int 4, bigint 8

但是小空间带来的问题, 存储量的减少, 范围的减少.

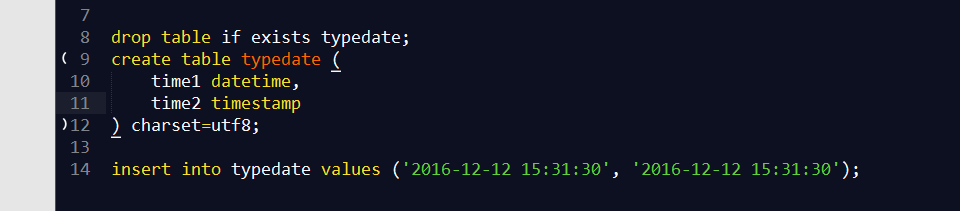
Boolean = tinyint

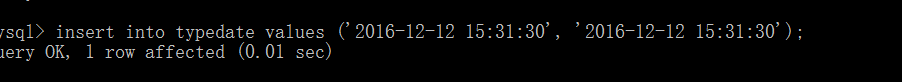




datetime:8 timestamp:4 date time year

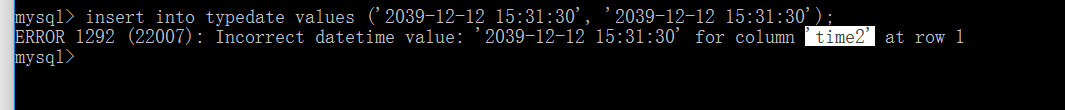
datetime 和 timestamp的选择?







存储 2039年, 超过了unix时间戳范围:



timestamp字段失败了.

timestamp占用4个字节, 存储的时间范围是有限的.1970-2038-1-19 3:11

例如存储登录时间. 就可以用timestamp来存储.

## 尽可能定长(占用固定的存储空间)数据类型

定长的数据类型的运算速度, 要不变长的数据类型速度快.

类型: char(32), varchar(32)

数据:hellokang, hellokang

空间:32, 9+1(1个空间记录长度)

如果在乎存储空间: varchar

在乎的计算效率:char, 最大长度为255

varchar() 于 text 的选择?

varchar, 占用记录的存储空间

text, 不占用记录的存储空间

一条记录的存储的空间有最大的限制, 默认是65535Bytes

浮点(double8, float4) 定点(decimal)

浮点, 更在乎存储空间, 不能保留全部的精度. 定长数据类型, 占用固定的存储空间.

定点, 更在乎精度, 保留精度. 占用非固定的存储空间, 随着有效数位增加, 而占用更多的存储空间.

PHP, float, 他是双精度的double, 占用8个字节!

## 尽可能用整数代替字符串

整数计算能力最强.

典型的 ipv4的存储, 可以转成int. (存储空间小, 整数计算快)

使用起来是字符串, 存储起来是整数

enum, 枚举, 单选:

set 集合, 复选:

# 设计 - 范式 Normal Format, 规范的格式

数据设计数据表是, 通用的规范规则, 帮助设计更优秀的数据表结构.

范式的定义是 层级定义, 1NF, 2NF, 3NF …… 6NF 第一, 第二…第六范式.

当前级别的范式, 都是在满足前一级别的基础上, 在增加规范内容.

通常满足3NF范式即可!

结论:

* 每类实体独立一张表
* 每张表存在独立的自动增长的非业务逻辑ID作为主键
* 使用关联字段或者关联表, 去表示实体间的联系

## 1NF, 第一范式, 满足原子性

字段, 属性应该独立,不可再分的. 就称之为字段满足原子性.

看下面的设计:

表示, 老师 在 某个班级 的那个阶段, 进行授课.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 老师 | 性别 | 班级 | 教室 | 授课时间 |
| 孔子 | 男 | PHP4 | 203 | 2016-01-01 到 2016-3-15 |
| 老子 | 男 | PHP4 | 203 | 2016-3-16 到 2016-6-11 |
| 孔子 | 男 | Java3 | 201 | 2016-03-17 |

观察授课时间字段?

在数据上由两个部分组成: 1开始 2结束 时间

如果在应用程序中, 需要仅仅需要开始的时间. 授课时间的字段,就不满足 逻辑上的原子性.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 老师 | 性别 | 班级 | 教室 | 开始时间 | 结束时间 |
| 孔子 | 男 | PHP4 | 203 | 2016-01-01 | 2016-3-15 |
| 老子 | 男 | PHP4 | 203 | 2016-3-16 | 2016-6-11 |
| 孔子 | 男 | Java3 | 201 | 2016-03-17 | 2016-5-12 |

再思考: 如果应用程序, 就是需要授课时间段, 而没有需要独立的开始时间和结束时间呢?

可见, 授课时间是不需要再拆分的.

拆分后, 满足 逻辑上的原子性. 而不拆分, 也满足新的逻辑上的原子性.

可见, 逻辑上的原子性, 是依赖于当前的应用程序需求的.

物理结构上, 关系型数据库系统的表, 都满足物理的原子性.

二维表, 都满足 第一范式!1NF.

## 2NF, 在满足第一范式的前提下, 消除对主键的部分依赖

增加独立的主键字段!

主键: 表中, 记录的唯一标志. 称之为主键. primary key.

主键 可以由1个或多个字段构成:

如下表: 在不增加字段的情况下, 建立主键:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 老师 | 性别 | 班级 | 教室 | 开始时间 | 结束时间 |
| 孔子 | 男 | PHP4 | 203 | 2016-01-01 | 2016-3-15 |
| 老子 | 男 | PHP4 | 203 | 2016-3-16 | 2016-6-11 |
| 孔子 | 男 | Java3 | 201 | 2016-03-17 | 2016-5-12 |

选择 老师+班级 两个字段组成联合主键.

部分依赖: 在联合主键的情况下, 表中的非主键字段, 依赖于(A字段确定, 就可以确定B字段, 称之为B字段依赖于A字段)联合主键中的部分字段.

例如, 性别字段, 就由老师字段决定, 性别字段部分依赖于主键中的老师字段. 就产生了部分依赖!

消除联合主键, 也就消除了部分依赖的可能性.

增加一个于业务逻辑无关的自动增长的ID字段, 作为记录的主键.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | 老师 | 性别 | 班级 | 教室 | 开始时间 | 结束时间 |
| 23 | 孔子 | 男 | PHP4 | 203 | 2016-01-01 | 2016-3-15 |
| 25 | 老子 | 男 | PHP4 | 203 | 2016-3-16 | 2016-6-11 |
| 57 | 孔子 | 男 | Java3 | 201 | 2016-03-17 | 2016-5-12 |

此时, 主键由ID构成. 没有联合主键, 就没有部分依赖!

就满足了第二范式.

## 3NF,第三范式,满足第二, 消除对主键传递依赖

传递依赖:

字段, 依赖于非主键之外的其他字段. 就成为形成了对主键的传递依赖.

例如:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | 老师 | 性别 | 班级 | 教室 | 开始时间 | 结束时间 |
| 23 | 孔子 | 男 | PHP4 | 203 | 2016-01-01 | 2016-3-15 |
| 25 | 老子 | 男 | PHP4 | 203 | 2016-3-16 | 2016-6-11 |
| 57 | 孔子 | 男 | Java3 | 201 | 2016-03-17 | 2016-5-12 |

性别依赖非ID的, 老师字段. 性别依赖于非主键字段, 产生了传递依赖.

解决思路: 将同一种实体, 使用一张表进行存储, 解决之后, 老师, 班级, 授课信息分别在表中存储, 如果需要, 使用主键字段,进行关联即可:

授课信息表:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | 老师ID | 班级 ID | 开始时间 | 结束时间 |
| 23 | 1 | 11 | 2016-01-01 | 2016-3-15 |
| 25 | 2 | 11 | 2016-3-16 | 2016-6-11 |
| 57 | 1 | 12 | 2016-03-17 | 2016-5-12 |

老师表

|  |  |  |
| --- | --- | --- |
| 老师ID | 老师 | 性别 |
| 1 | 孔子 | 男 |
| 2 | 老子 | 男 |

班级表

|  |  |  |
| --- | --- | --- |
| 班级ID | 班级 ID | 教室 |
| 11 | PHP4 | 203 |
| 12 | Java3 | 201 |

以上的设计就满足了第三范式.

可见, 满足, 每种实体, 一张表, 实体间的关系, 使用关联字段表示, 就可以满足第三范式!

## 为什么要满足范式?

尽可能减少数据冗余.

提升更新的便利性.

## 逆范式优化

为了提升某个操作的速度, 而打破范式, 称之为逆范式!

有种优化策略.

例如:

问题表 question

|  |  |  |  |
| --- | --- | --- | --- |
| 问题ID | 标题 | 发布时间 | 分类ID |
|  |  |  |  |
|  |  |  |  |

分类表: category

|  |  |
| --- | --- |
| 分类ID | 分类标题 |
|  |  |
|  |  |

下面的SQL需求: 查询分类, 及其分类对于的问题数量:

select c\*, count(q.question\_id) as question\_number from category c left join question q using(category\_id) group by c.category\_id;

当, 查询分类时, 总是需要查询到分类对应的问题的数量

每当查询category,就需要连接 question. question的数据量一定会很大.

设计分类表时, 强制增加一个字段, 问题数量:

|  |  |  |
| --- | --- | --- |
| 分类ID | 分类标题 | 问题数量 |
|  |  |  |
|  |  |  |

此时在一张表中就可以获取分类及其对应的问题数量.

效率是提升了, 数据冗余了. 打破了范式. 就是 逆范式的优化方案!

注意, 打破范式, 带来业务逻辑的增加.

必须要设计好再做, 哪些操作会影响当前的数据设计, 必须要同时更新才可以!

慎做!

# 设计 - 存储引擎的选择

逐渐演变成: “选择MySQL 还是 MariaDB?”

## 结论

新版本的MySQL的话, 选择Innodb没错的.

依据基本功能选择.

MyISAM: 擅长处理快速的查询和插入. 典型的web程序的形态.

Innodb: 擅长大量并发, 处理大量的更新操作. 支持事务, 外键约束.

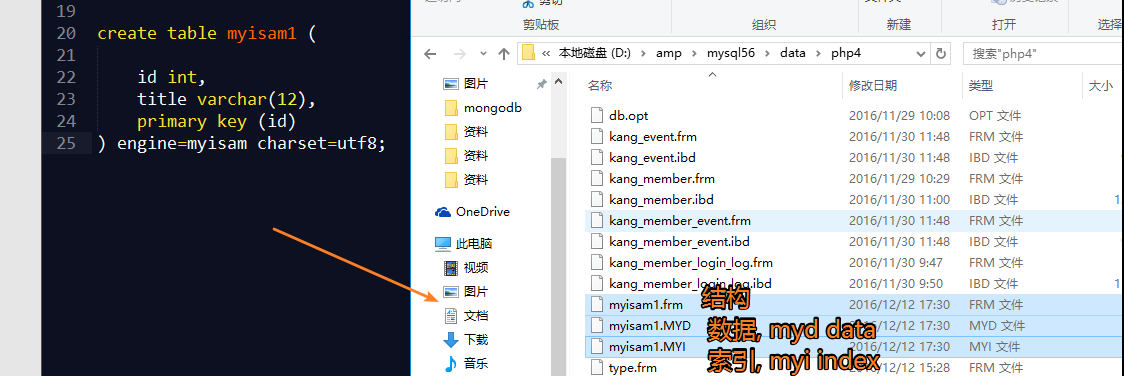
## MyISAM

### 存储机制

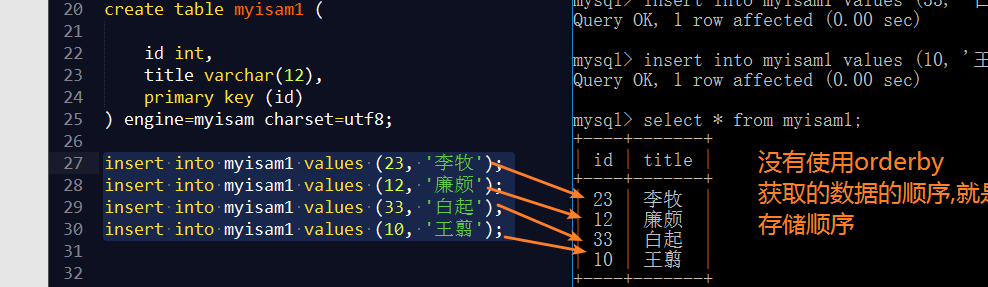
数据和索引, 分开到不同的文件中进行存储

table.myi

table.myd



数据依据于插入顺序进行物理存储的:



由于插入时, 每次不进行排序, 直接插入到表的末尾, 速度比较快.

但是也带来了一个问题. 删除空洞的问题.

如果某些记录被删除了. 原本记录所占用存储空间, 插入是不会利用.

myiam表的这种机制, 导致需要手动的定期去修复空洞.

演示: 创建大点是数据表

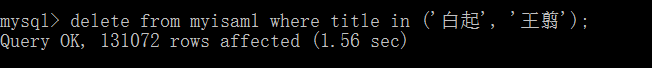
自己复制自己的方式增加大量的重复数据

原始的数据

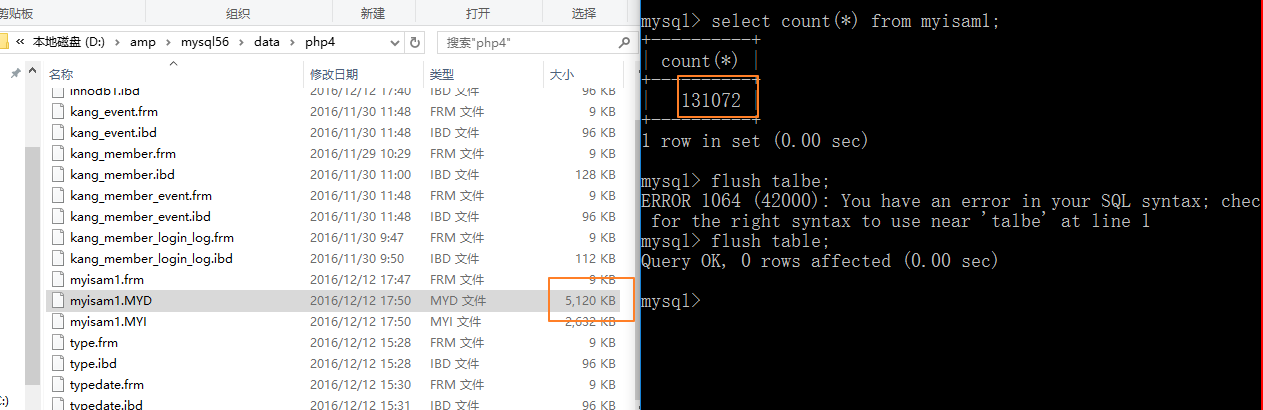


flush table 快速刷新内容到磁盘.

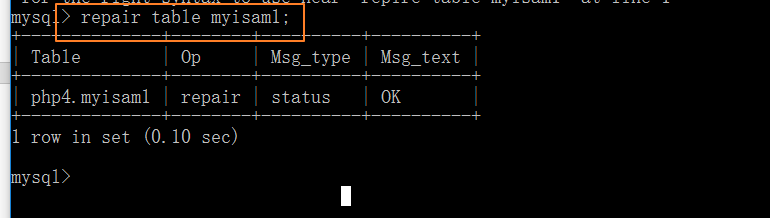
删除一半的数据



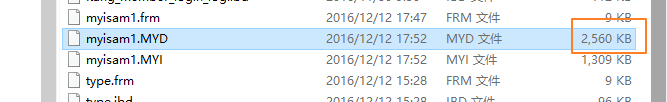
删除的结果



修复该表:使用repair table table-name



文件大小:



### 特殊功能

<=5.6全文索引. (很鸡肋, 不支持中文, 没人用)

数据myd文件支持压缩存储, 最小的使用磁盘空间. (不是很常用, 一旦压缩, 表不可编辑, 只能读取. 磁盘空间成本很低)

### 并发支持

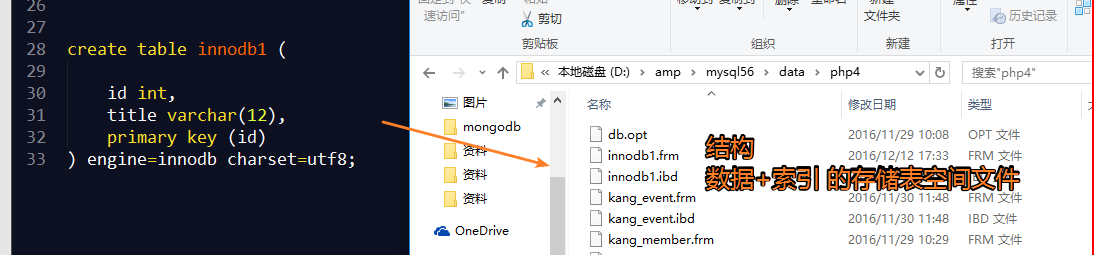
相对差!

仅仅支持: table-level lock, 表级锁定

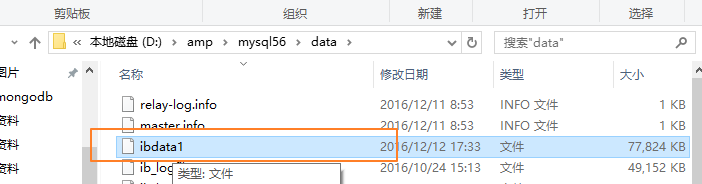
## Innodb

### 存储机制

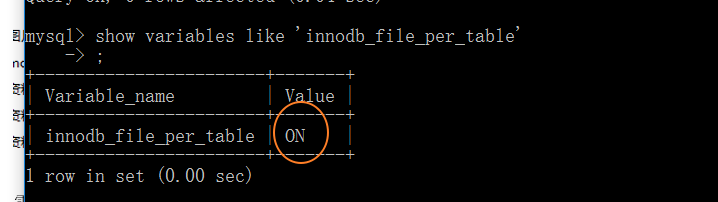
数据和索引是集中(聚簇, 聚集)存储的, 存储在.ibd的表空间文件中:



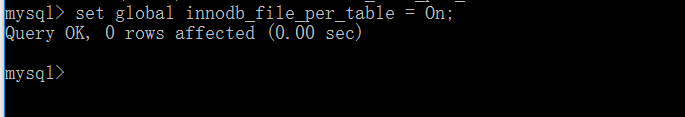
(PS: 表空间文件, 在新版本中, 默认是每张表一个表空间. 稍微早一点的版本, 所有的innodb一个表空间文件.



需要配置,innodb\_file\_per\_table配置项, 每个表一个文件

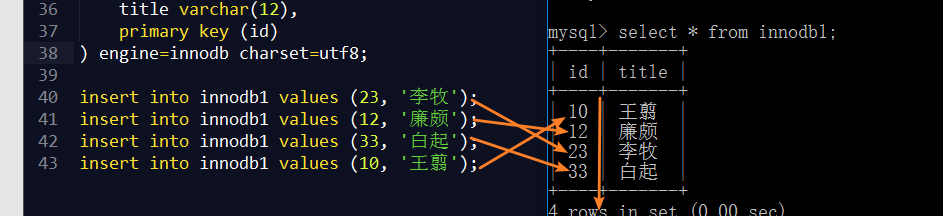


设置为低版本



)

数据的存储顺序, 依据主键排序的



### 特殊功能

事务支持, 外键支持

事务, 典型数据的必备的特征, 用于保证数据的完整性. 通过ACID(原子性,一致性,隔离性,持久性)的支持实现.

外键约束: 典型的用于保证数据关系完整性机制. MySQL层面.

对于程序员来说, 无论数据库层面是否保证的数据关系的完整性, 在应用程序层面一定要保证

### 并发支持

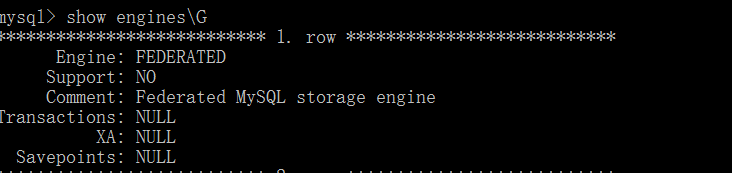
相对好!

支持: table-level lock表级锁定 , row-level lock 行级锁定.

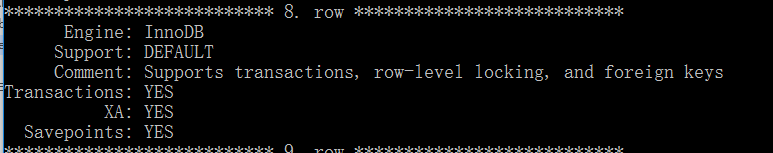
智能选择使用什么级别的锁定!

支持: MVCC机制, 多版本并发控制, 来提升锁定带来的效率问题. 几乎可以做大, 非锁定读操作!

## show engines,查看支持的存储引擎







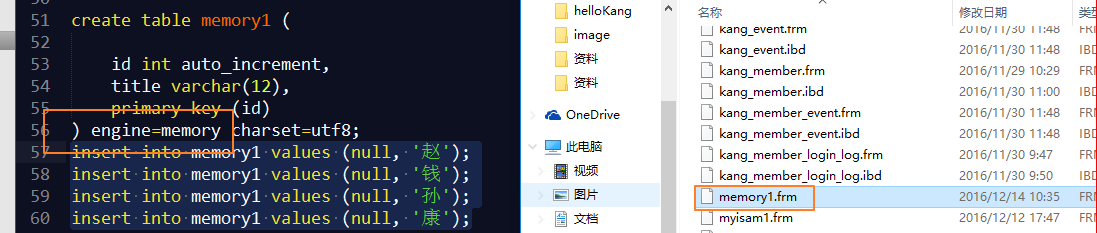
MySQL支持大量的存储引擎.

例如

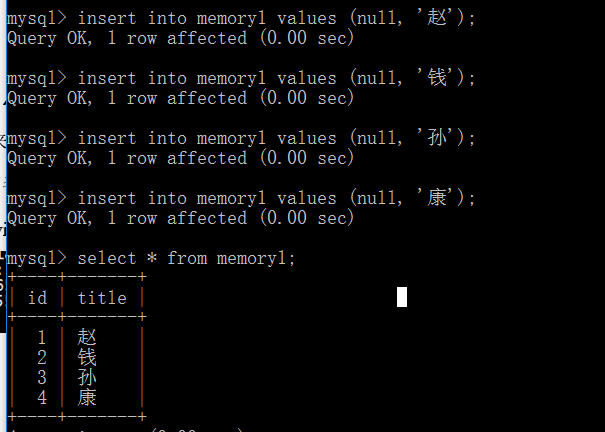
## memory, 缓存型存储引擎

数据和索引, 存储在内存中

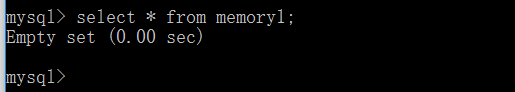
仅仅存在 .frm结构文件!



我们也可以操作该表.



在 服务器重启后, 数据丢失!



不适合做持久化存储, 但适合做缓存系统. mysql内部的临时表也采用该存储引擎存储!

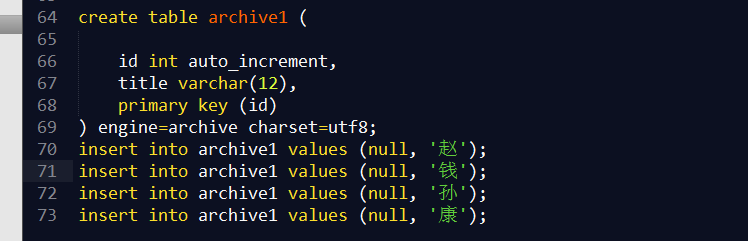
但是, memory表, 不支持 text, blob(二进制数据) 大块的数据!

## archive, 存档型

高速存档型 类似myisam

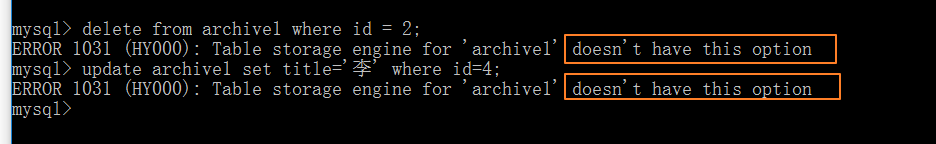
常规操作:

仅支持添加和查询, 更高效.





不支持修改和删除.



适合做, 日志类系统!

# 锁定

通用的机制, 解决资源争用的典型机制. 并发带来了资源争用.

解决资源争用, 典型的方案就是在资源上加锁.

(队列是干什么的? 争用前,为了避免出现争用, 而先排队处理)

资源层面解决狼多肉少就是 锁定

业务逻辑应用层面解决狼多肉少, 就是队列.

通常, 锁定需要队列. 一旦一个资源被锁住, 其他的请求, 就需要排队等待.

## 锁定类型

加锁之后, 对资源的操作限制不同, 带来了不同的锁定类型.

典型的 就是 读锁和写锁.

读锁, s-lock, 共享锁, 资源被读取操作时, 要求增加的锁定类型. 结果: 共享读操作, 阻塞写操作.

写锁, x-lock, 排他锁, 独占锁, 资源被写操作时, 要在增加的锁定. 结果: 阻塞其他的所有操作, 独占这个资源.

## 锁定粒度

一旦加锁, 锁定的范围, 锁定的记录数量. (文件操作时没有这个概念, 文件仅仅锁定整个文件, 只有一种粒度)

表操作有粒度的概念:

表级锁定: 一旦加锁(读和写), 锁定整张表. 例如, 在更新(写)一条记录, 表中的其他记录(整张表)被我独占, 其他客户端,对表不能进行其他操作.

行级锁定: 一旦加锁(读和写), 仅仅锁定当前操作的行. 其他行, 其他的客户端是可能操作的.

## 加锁解锁

自动添加锁定

在MyISAM中 模拟事务, 实现事务的隔离性, 人为加锁!

锁, 先尝试加锁, 如果加锁成功, 再使用资源!

# 功能 - index-索引(最重要环节)

## 概述

索引, 是最常规, 最重要的优化措施. 在不改变设计结构, 不增加分布式的数据库服务器的基础上, 提升查询速度!

性价比最高一种优化措施.

索引(index): 采用完整数据中的部分内容作为关键字, 建立于完整数据位置间的 一种映射关系, 称之为索引.

关键字于记录位置的映射, 就是索引.

例如:

字典的拼音检索: 拼音一个汉字的某个部分属性(完整数据中的部分内容). 确定汉字在第几页(数据位置), 建立了 拼音和页数 之间的对应关系.

数据库系统中, 索引主要用来标识记录, 进而完成, 查找, 排序, 返回数据等工作!

在索引中检索的效率高于在全部的数据中, 原因何在?

1, 索引中仅仅存在部分数据, 体积小, 遍历会更快

2 索引中的关键字是排序的. 提供更快速的检索.

## MySQL支持索引类型

无论何种类型, 都是 关键字和位置的映射.

不同类型的索引, 仅仅是关键字的获取, 对关键字的限定是不同的.

共4中索引类型为:

### 普通索引, index

对关键字没有要求.

### 唯一索引, unique index(key)

要求关键字不能重复.

### 主键索引, paimary key

主键自动成为主键索引.

要求, 关键字不能重复 同时, 关键字不能为null.

### 全文索引, fulltext index

mysql的全文索引不支持中文.

如果需要在中文上, 增加该操作, 必须要使用其他的替代方案!

典型的:

Coreseek(sphinx)

XunSearch(Xapina)

## 索引的管理(语法)

### 创建索引

可以在创建表时, 同时创建索引.



创建索引时, 可以为索引起名.默认的名字于字段名相关! 只有一个主键, 主键不需要名子.

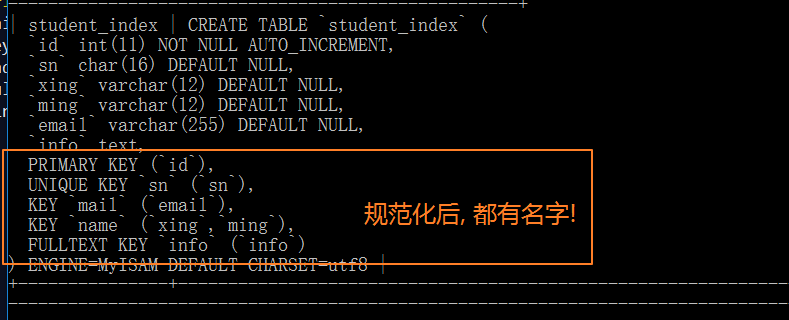
创建索引时, key于index是通用的.除primary key外!

一个索引可以由多个字段构成, 任何类型的都可以. 就是联合索引.

也可以在, 后期修改表结构式, 去创建索引.

与创建一致, 使用 alter table table-name add XXXX;

### 查看索引, show



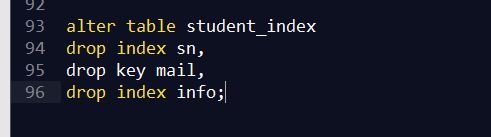
### 删除索引

删除主键:

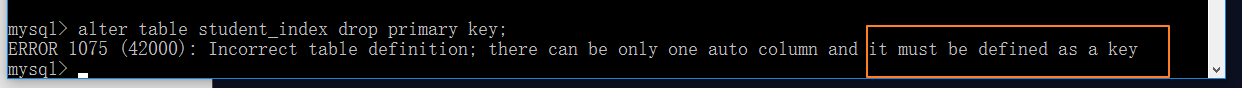
drop primary key

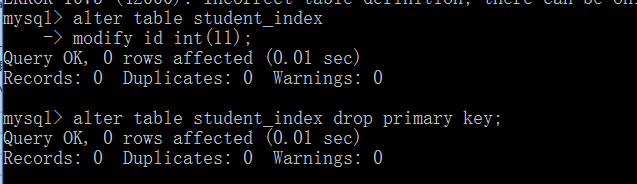
删除其他索引

drop index index-name, 不分类型, 无论是全文, 普通, 唯一, 都是如此!



删除主键索引时, 通常会在主键字段上, 存在AI(auto\_increment)机制, 必须要去掉AI机制, 再删除主键. AI机制依赖key.

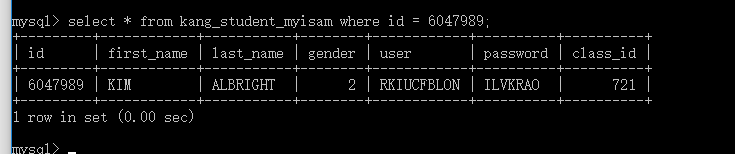




## explain – 获取执行计划 - 分析索引的使用

例如:

查询 利用 ID查询检索:

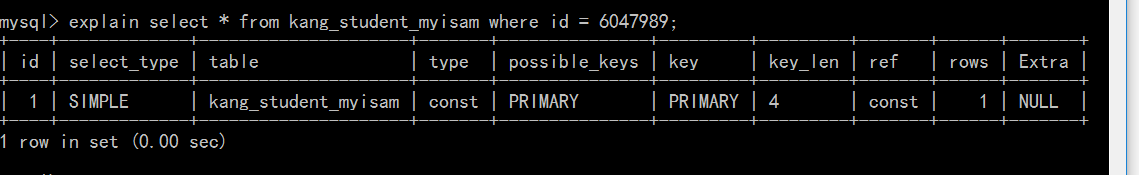


仅仅速度快.

但是是否使用了索引, 看不出来的吧.

使用 explain机制, 就可以判断出来.

在select语句前, 增加 explain:

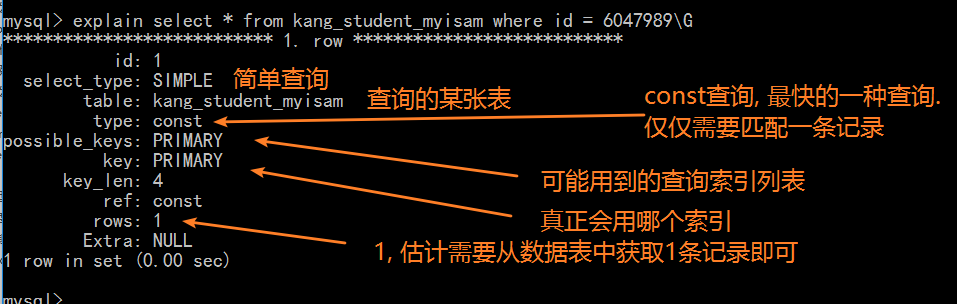


当前explain执行时, SQL就不执行了. 仅仅获取该SQL的执行计划.

如图所示:



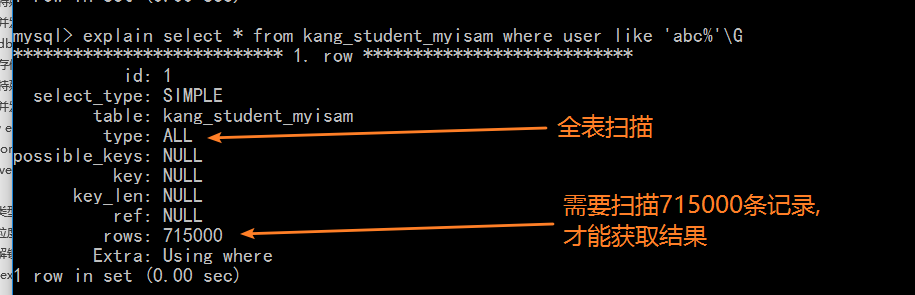
通过执行计划, 可以知道这条SQL的执行基本信息:



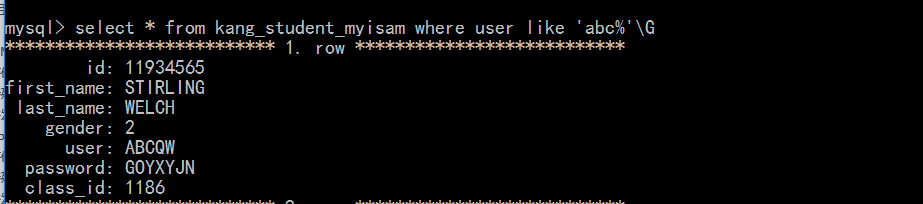
key字段可知, 当前的查询, 使用了索引!

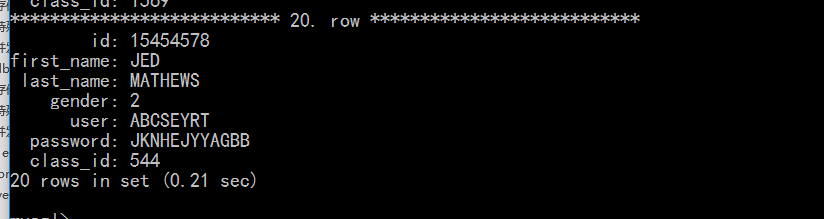
测试效果:

没有使用索引的执行计划



可以直接执行:





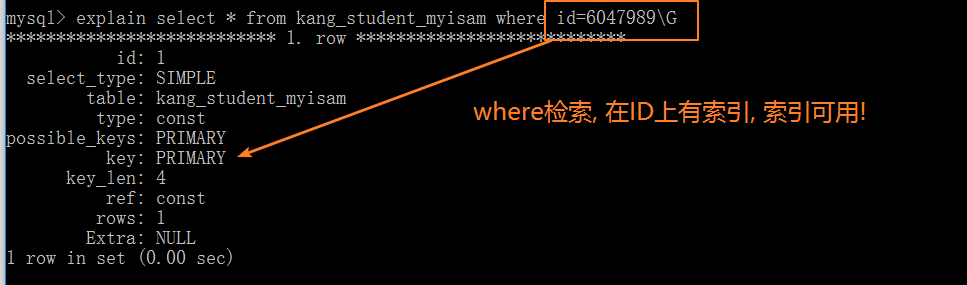
## 使用场景

结语: 在where，orderby, join 所对应的字段后边增加索引．

如果存在, 哪些情况, 可以用到索引功能.

### where检索

查询的字段的上存在索引, 索引是可以起作用

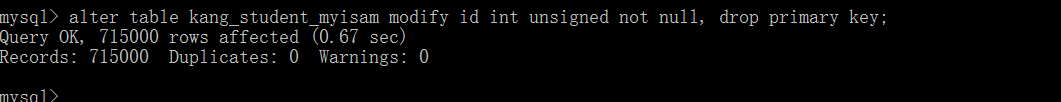


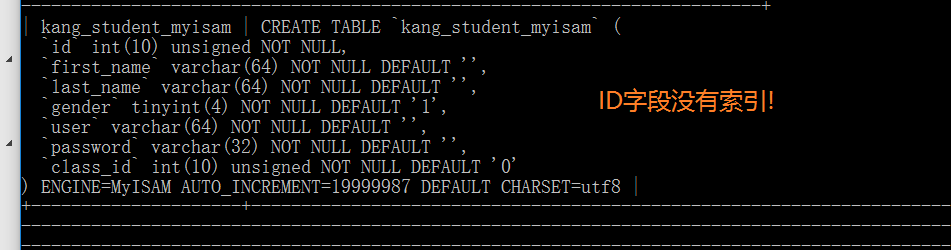
### order by排序

在order by 利用字段进行排序时, 如果字段上存在索引. 索引也会被启用.



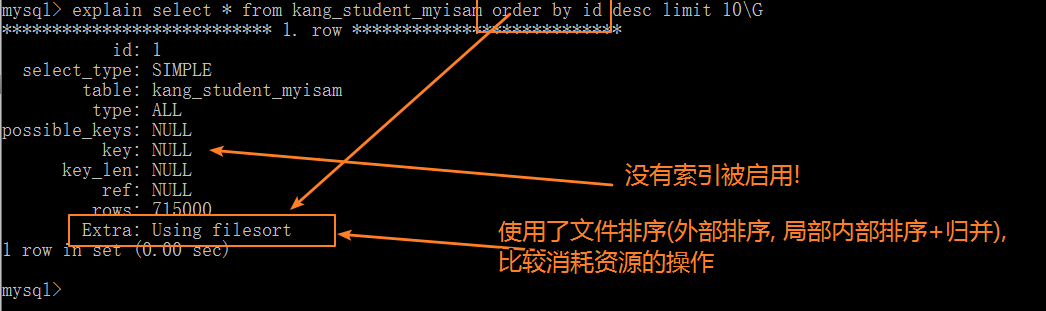
如果删除id字段上的索引, 再执行上面的相同的排序SQL





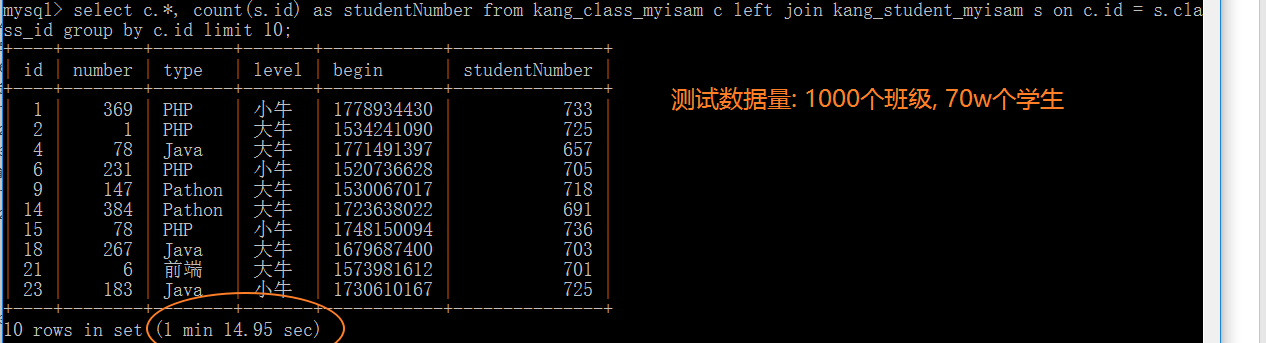
相同的SQL执行:

由于没有id字段上的索引可用,因此, 会使用外部排序完成处理!

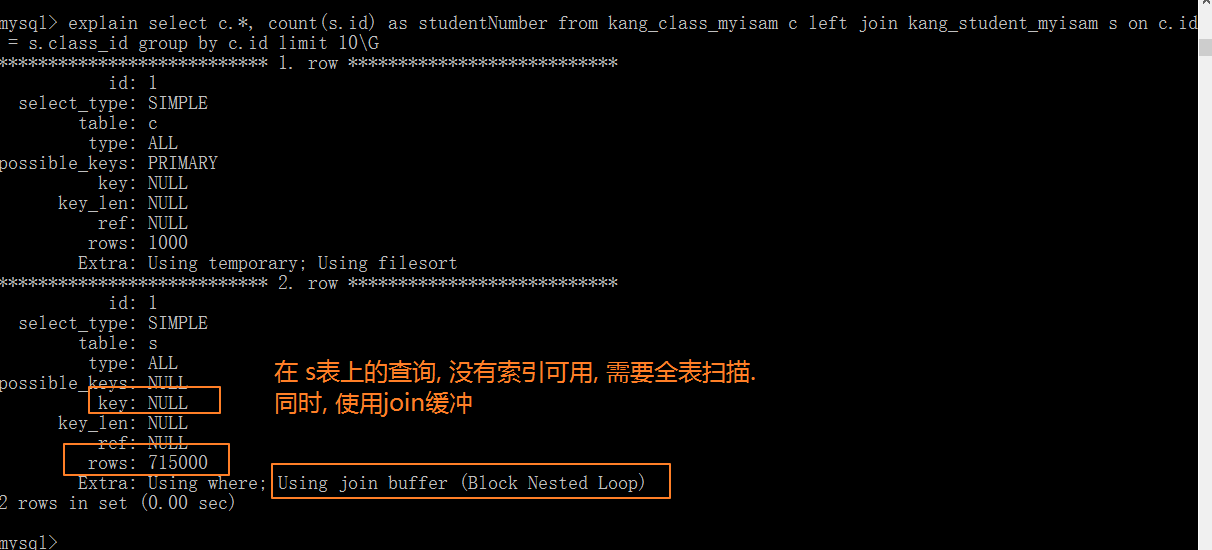


### join, 连接查询时关联字段

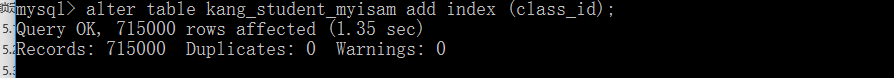
执行join操作时, 在表中进行检索时, 需要利用到关联字段, 例如下面的统计班级人数的查询: on s.class\_id = c.id



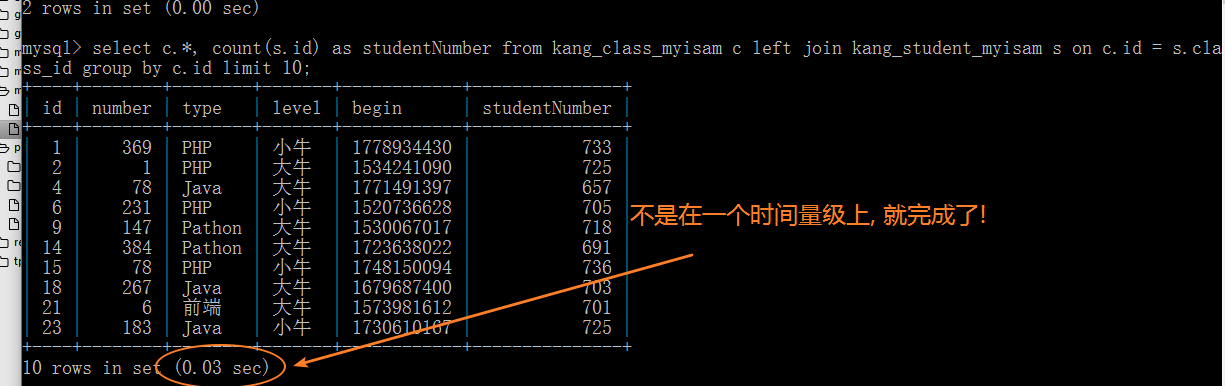
由于在关联字段 s.class\_id上没有索引可用, 导致查询时间过慢:



简单的在 class\_id字段上, 增加一个索引



在做相同的查询:



索引启用了:



### 查询字段使用索引，索引覆盖

索引覆盖: 索引中的关键字, 覆盖了查询需要的字段. 则查询时, 会使用索引完成, 而不是使用全部的记录内容.

在执行计划中, 明确的指标: extra: using index 表示使用了所以覆盖!

select \*

select后使用查询字段, 表示需要返回的字段的数据.

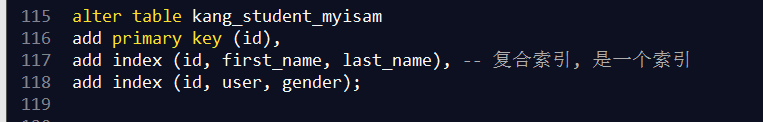
\*, 全部字段.

但如果不需要全部数据, 则可以使用 字段列表的形式.

测试:

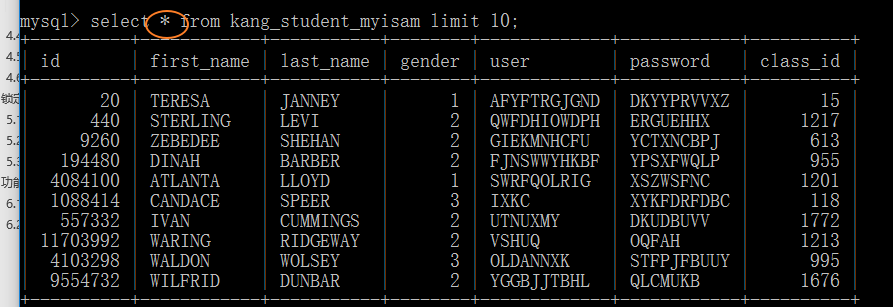
增加了不同的字段组合的索引. 执行查询操作:

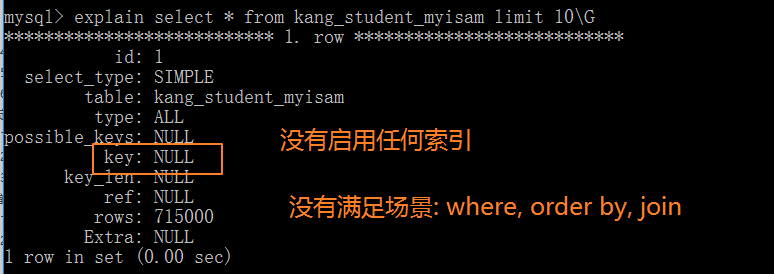
增加不同的索引:



测试查询, 查询的字段是\*, 全部字段: 没有where, order by:

结果没有启用索引.





更改了查询的字段的组合, 不是\*, 而是: id, user, 再测试:



继续测试:





可见, 查询时, 尽可能写少量的字段, 就会最大的可能性, 触发所以覆盖!

如果一个查询, 是基于索引查询得到的, 就称之为, 索引是三星级.

二星级: 一个索引, 可以同时适应于多个场景.

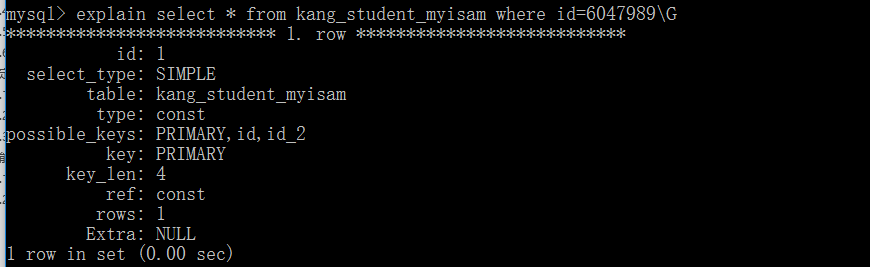
一星级: 一个索引, 仅仅为一件事情而形成.

## 使用语法原则

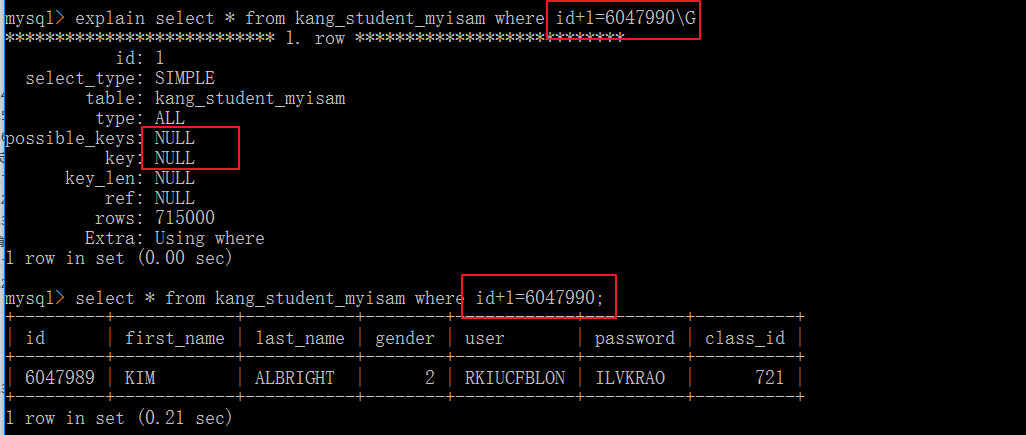
### 字段独立

如果需要使用字段上的索引. 则字段一定要独立出现在等式的一侧. 不能出现在表达式中.

id字段上有索引



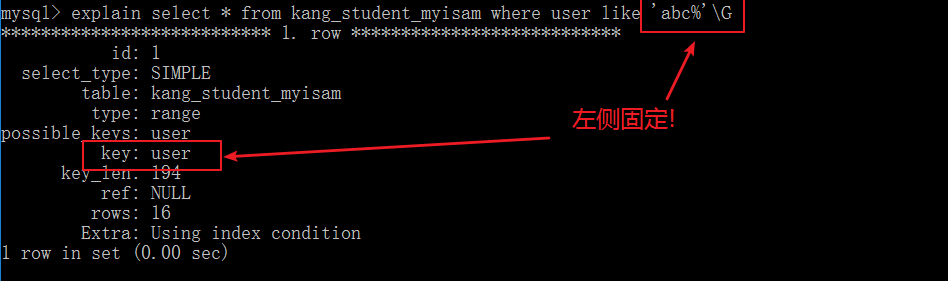
id参与表达式, 同样的内容, 就没有索引可用:



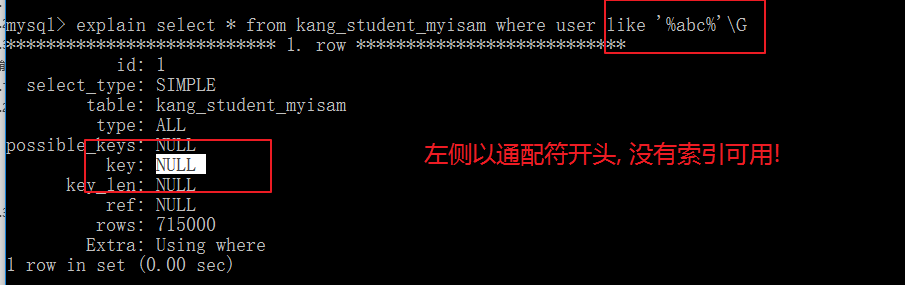
### 左原则

#### like

模糊查询. like的匹配字符串. 要求左侧确定, 才可以使用索引.



不满足:



因此, 在实操中, like 仅仅用来做 左边固定的查询, 例如, 搜索建议.

|  |
| --- |
| PHP |

PHP教程

PHP培训

PHP视频教程

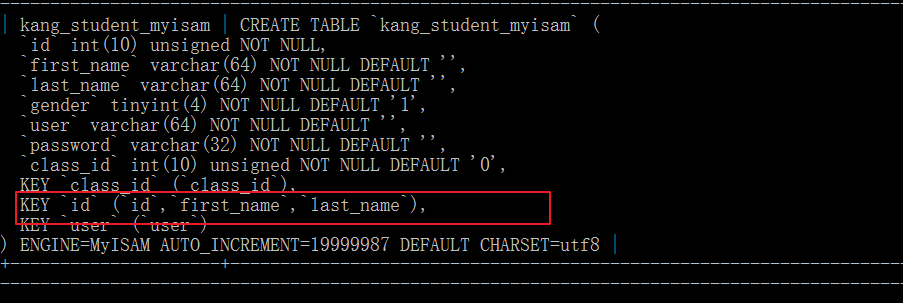
而一旦出现包含, %keyword% 就不要用like完成. 用 全文索引 完成!

#### 复合索引

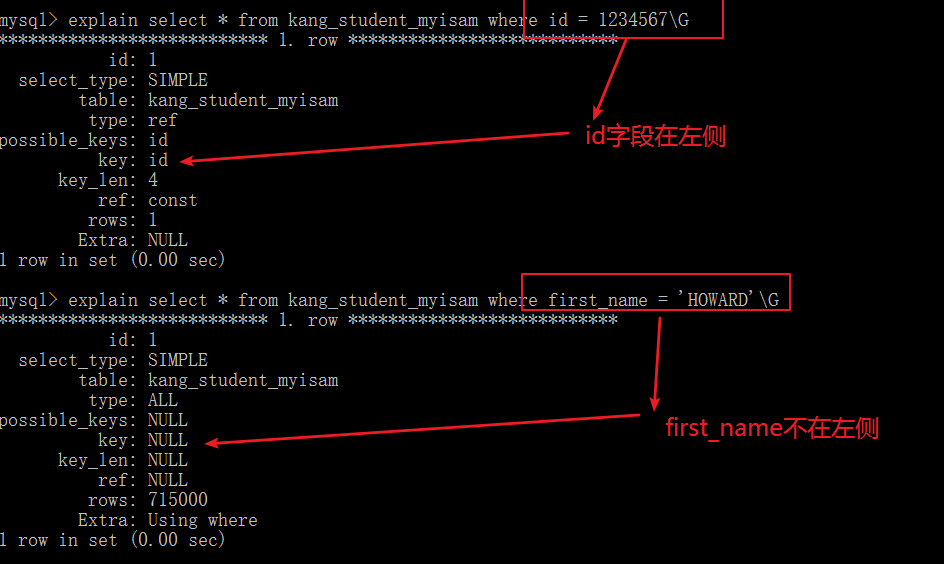
一个索引由多个字段组成.

多个字段间的地位是不一致 , 索引中, 按照左边字段排序, 如果左边字段相同, 再按照右边字段排序.

因此, 在使用复合索引时, 左边字段可以直接用来检索. 而右边字段, 仅仅在左边字段确定的情况下, 才有检索功能:



执行检索, 发现只有id字段上存在可以用的索引



### OR原则

或者的逻辑关系, 要保证, 两端的条件都有索引可用, 才会使用索引

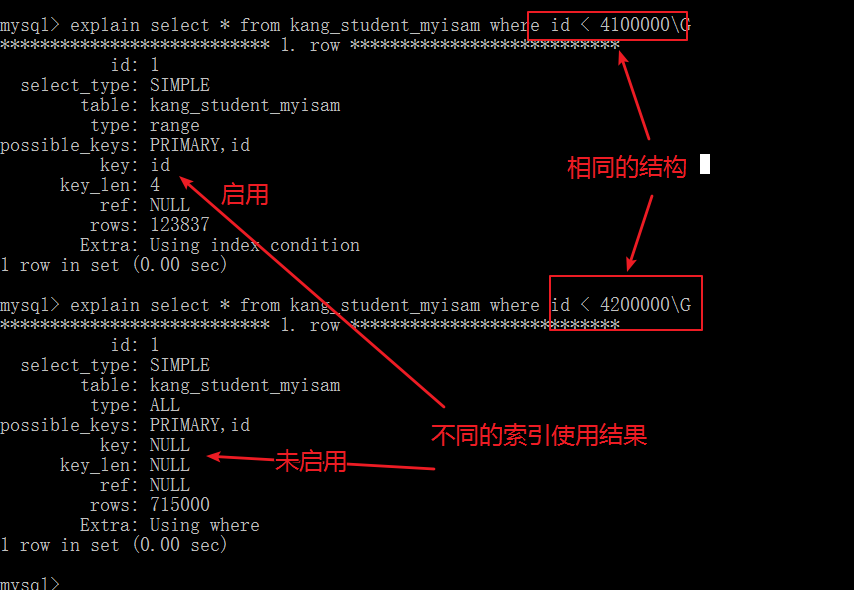
first\_name like ‘abc%’ OR last\_name like ‘xyz%’;

first\_name like ‘abc%’, 有索引.

last\_name like ‘xyz%’, 有索引

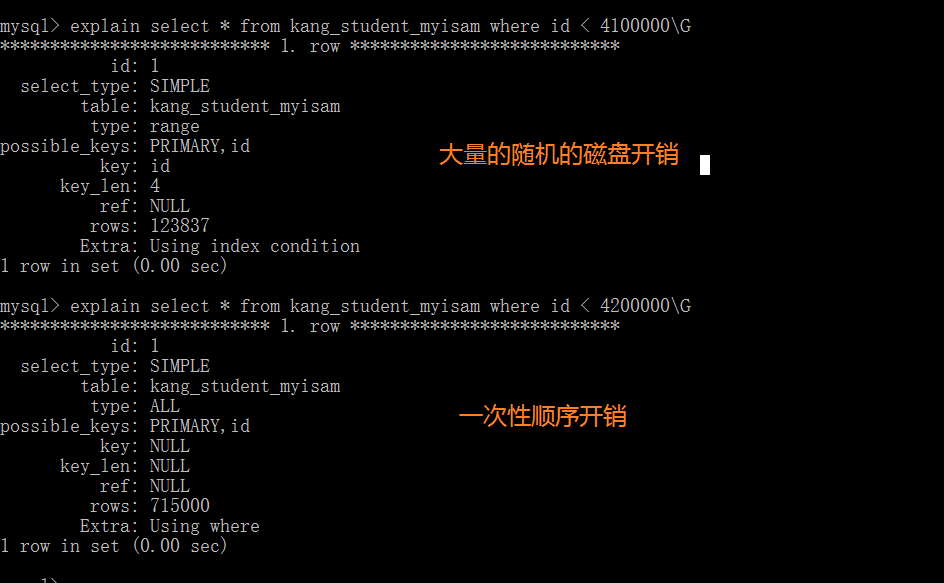
### MySQL会自动选择

注意下面相同结构的SQL, 但是索引有时启用, 有时未启用:



MySQL服务器, 自己判断, 在情况2中, 使用索引可能会更慢, 选择不用索引, 使用, All 全表扫描的方式完成查询.

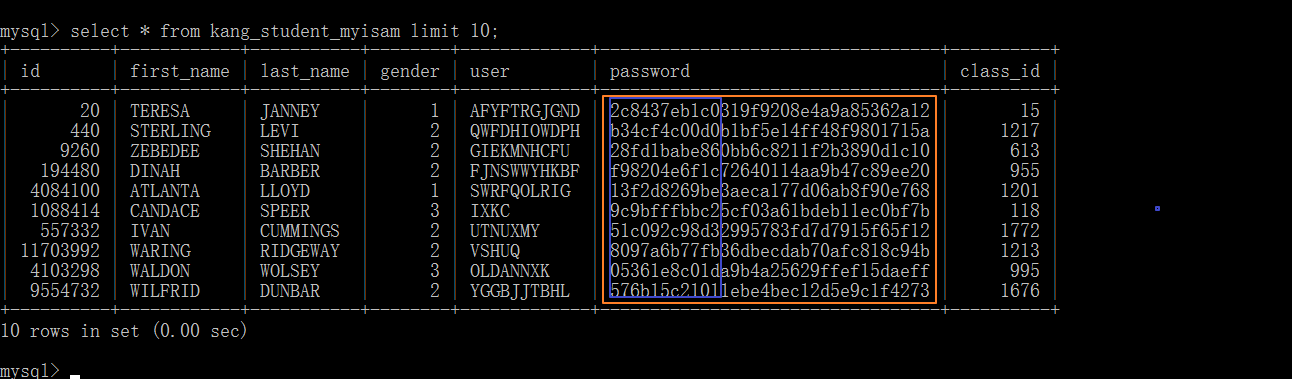
当MySQL判断, 出现了大量的随机的IO(磁盘读写, InOut)开销, 超过了一次性的全表扫描的顺序IO开销. MySQL就会选择更容易的全表扫描完成!



## 前缀索引

在使用索引进行检索时, 有些数据的特征, 是 左边的部分标识度已经足够, 不再需要全部的数据:

例如 hash处理之后又的密码:



在做检索时, 没有必要使用全部的password字段的.

查找密码是: 28fd1babe860bb6c8211f2b3890d1c10

或者密码以: 28fd1babe860bb6

或者密码以: 28fd1b

查找 的记录是一致的.

前缀的标识度, 足够.

既然前部分, 标识度足够, 建立索引时, 就可以使用前缀部分作为关键字即可. 不再需要使用全部数据.

建立索引时 , 关键字由字段的前部分构成的, 就称之为, 前缀索引.

语法:

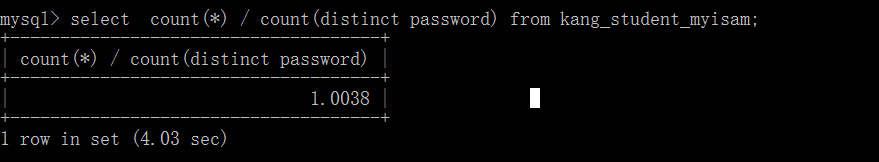
index (字段(前缀长度))

index (password (10)) 使用密码字段的前10个字符, 作为索引的关键字.

实操中, 难在, 分析前缀的长度?

原则: 前缀标识度 与 全部长度的标识度 尽可能一致即可.

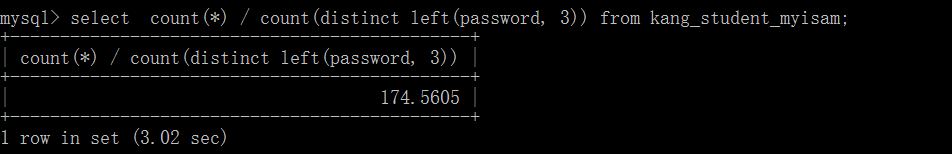
先计算, 使用全部长度的标识度:

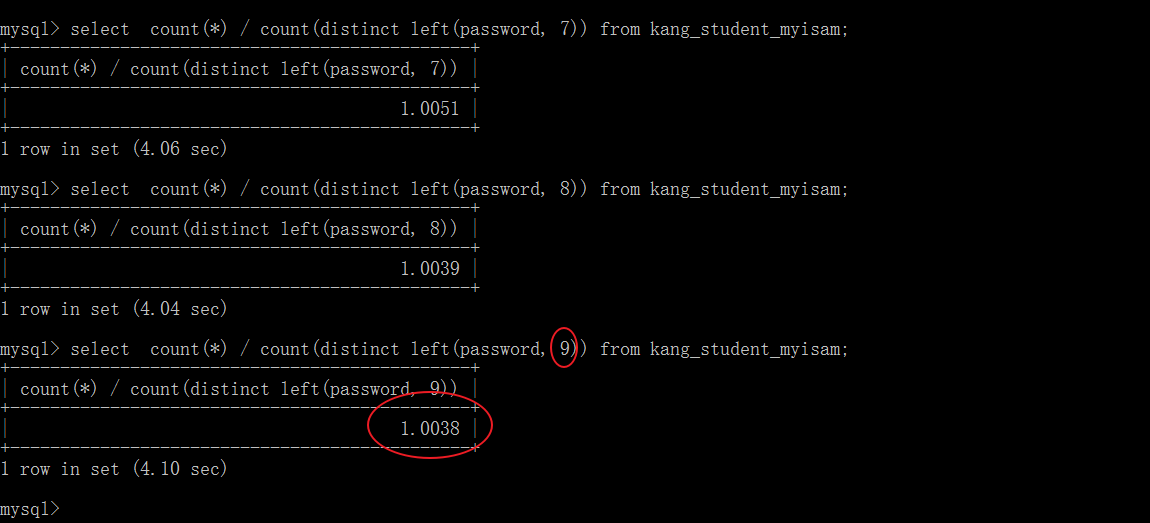


计算特定前缀的标识度:

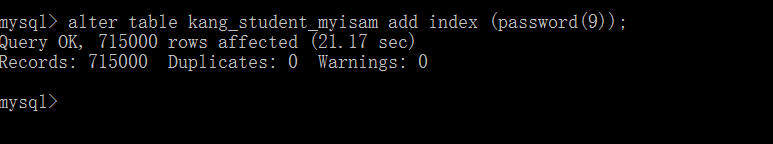
left(), 截取左侧的某个长度字符

count(distinct), 去重复统计





可见, 使用9长度的前缀, 于使用全部密码的前缀 非常接近了, 小于1/10000的差异!



目的, 缩减了关键字的长度, 加快了索引的遍历速度.

## 索引的数据结构(理论)

索引的存储, 是以何种方式, 何种结构, 格式 进行存储的. 这种格式给索引带来的优势是什么. 下面的结构, 不仅仅针对于MySQL, 几乎所有的索引都是下面的结构.

### BTree索引 – 磁盘上存储索引的不二选择

数据库的索引需要存储在磁盘上: .myi.索引文件. (量大, 需要持久存储)

为了加速, 在磁盘上的索引的检索遍历, 被存储于一种称之为较: BTree的结构中.

大体结构如下:

一个Btree的节点, 存储多个排序好的索引关键字



关键子之间, 存在子节点指针, 可以指向子节点, 子节点存储的就是 两个关键字之间的关键字



关键字也是和记录位置对应



最重要的特点: 减少磁盘IO, 增加获取的关键字的数量.

节点的大小, 取决于一次性磁盘IO的数据大小, 512(K)Bytes. 2者保持一致.

可以保证, 一次磁盘IO读取开销, 尽可能多的获取关键字数量.

意味着, 假设, 关键字的大小, 导致一个节点可以存储1000个关键字. 问, 2层的Btree, 可以存储多少个关键字?

1000 + (1000+1)\*1000 = 100w

意味着, 2次磁盘开销 就可以在100w的关键字中进行检索!

BTree的中文名:多路平衡查找树

MySQL的无论创建的(普通, 唯一, 主键, 全文)索引, 都是存在在Btree结构中.

(

辨析Btree不是(Binary Tree 二叉树)



)

### 聚簇索引, B+Tree

B+Tree, 在Btree的基础上, 做了些调整, 变化.

(回忆, innodb的记录的顺序, 依据表中的主键进行排序)

在MySQL的所有表的索引中, 只有一个索引例外, 不是存储在BTree结构中, 就是innodb表的主键索引.

而是, 存储在B+Tree中.

B+Tree是一个聚簇(聚集)结构.

典型的结构特征:

索引的关键字, 于 索引对应的记录 存储在一起.

如图所示:



继续思考, 由于记录和主键存储在一起? 记录的位置是否固定? 不固定.

记录的位置不固定, 那么 其他的索引中的关键字, 如何找到该记录?

innodb的非主键索引(二级索引), 不是聚簇的, 存储记录位置时, 不是存储的磁盘位置, 而是存储的是, 记录的主键值.

如果使用innodb的二级索引进行检索, 先是利用关键字, 确定主键值, 再利用主键值, 在聚簇的主索引上, 找到记录. 需要经过二次检索.

### Hash索引

hash, 利用key可以快速获取一个对应的值, 就是hash算法.

hash索引, 利用关键字, 就可以快速的确定关键字对应的记录位置, 就是hash索引.

hash索引, 利用内存的快速寻址达到的目的.

hash索引, 仅仅在磁盘上的索引被载入内存, 时, 才会被转换成hash结构, 便于在内存中快速确定.

内部优化的措施, 而不是存储磁盘上索引的措施.



## 如何创建索引?

* 需要建索引的表的结构?
* 与该表相关联的表的结构?
* 该表上执行的SQL集合.
* 统计SQL的权重以及频率.

结构+操作 是决定索引的要素.

二星级索引是目标.

基于测试,调试的.

如果实在搞不定:

在where，orderby, join 所对应的字段后边增加索引

# 功能 - 查询缓存 query cache(了解)

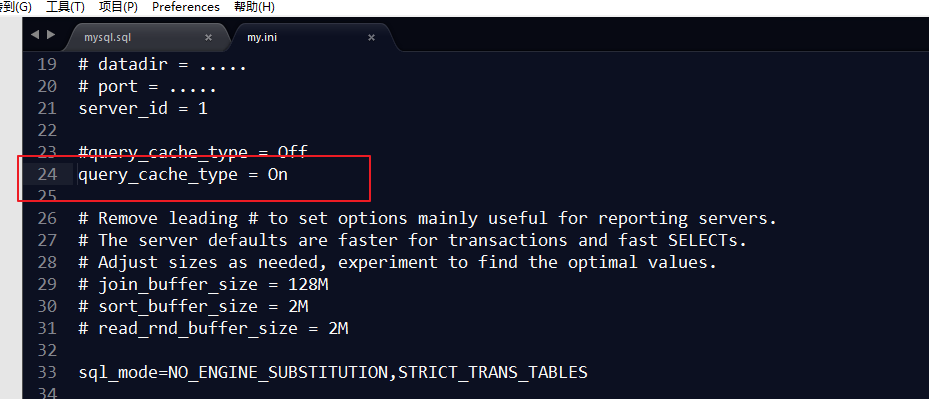
MySQL服务器端, 提供的, 用于缓存查询结果的一块缓存区, 称之为查询缓存, querycache!

当执行的select语句时, 结果就被缓存

下次执行相同的select, 直接缓存中获取. 对于客户端来说是透明的.

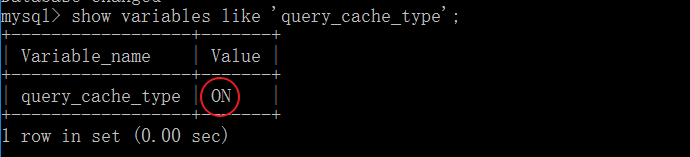
## 开启查询缓存

修改MySQL的配置文件 my.ini(my.cnf, linux)



重启MySQL的服务器

通过查询当前的配置变量值 可以看到已经开启:

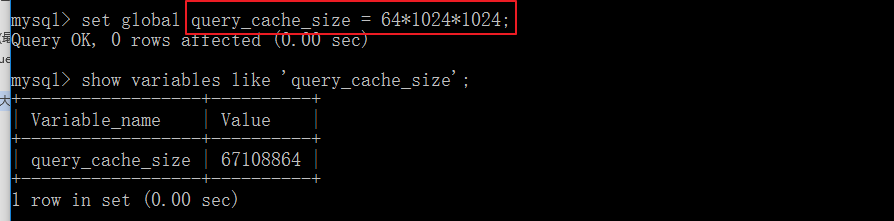


## 设置查询缓存大小

利用配置项, query\_cache\_size完成缓存的大小的配置



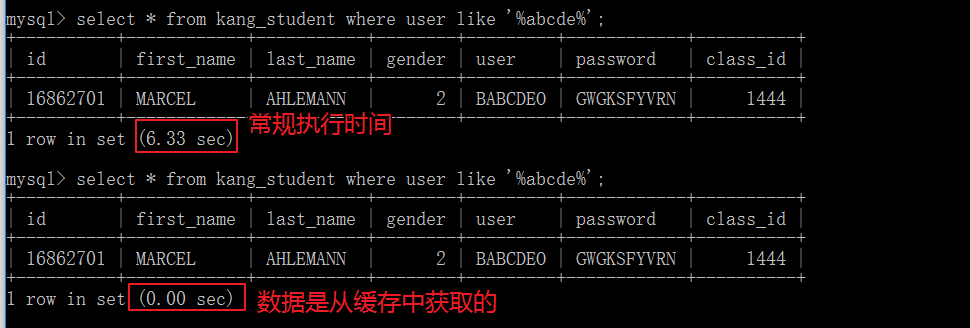
缓存大小设置:



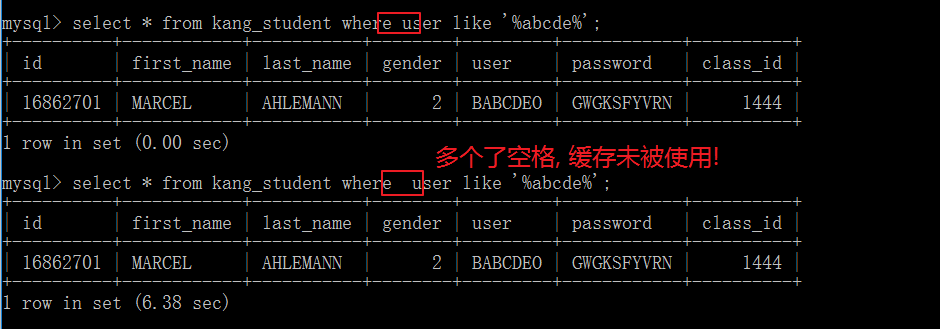
## 客户端常规执行select查询即可

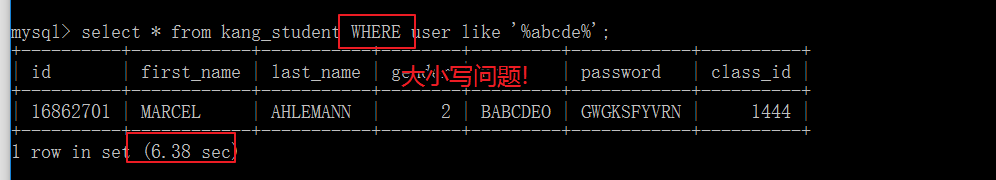
已开启缓存, 查询结果, 就会自动的被放入缓存.

执行相同的SQL的, 则缓存被使用

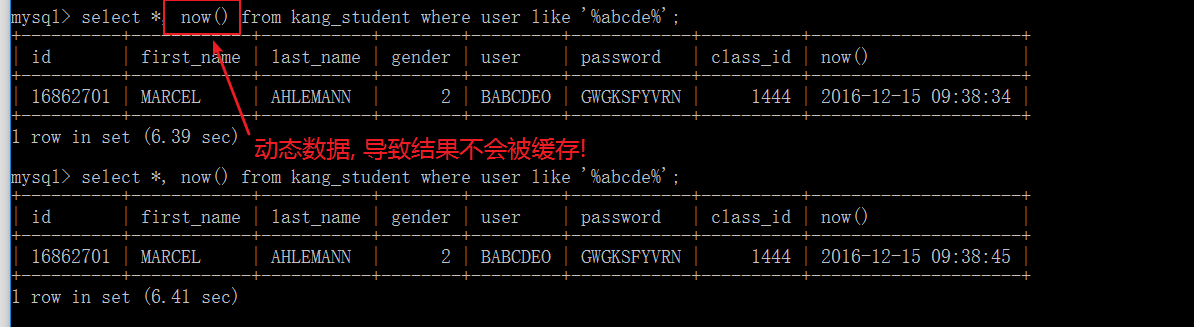


注意, 严格依赖于执行的SQL, 判断是否存在缓存的. 因此, 如果SQL语法出现变化, 例如 空格, 大小写, 都回导致SQL匹配失败, 从而缓存获取失败:





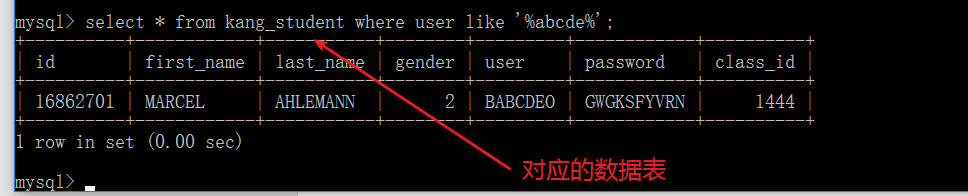
如果查询结果中, 存在动态数, 也是不能被缓存的:



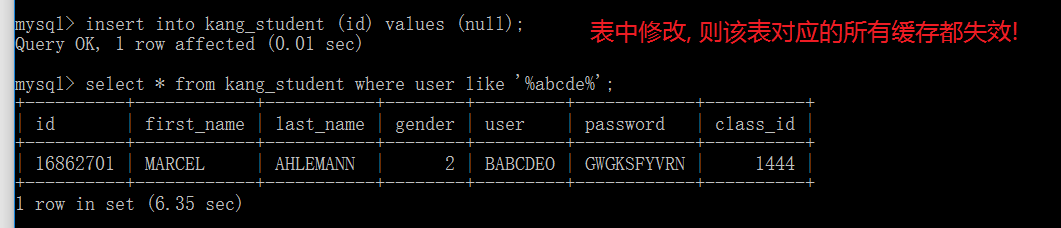
## 缓存失效

没有有效期. 永远不会主动失效.

缓存对应的表, 出现了数据的改动, 则缓存失效.



数据改动:

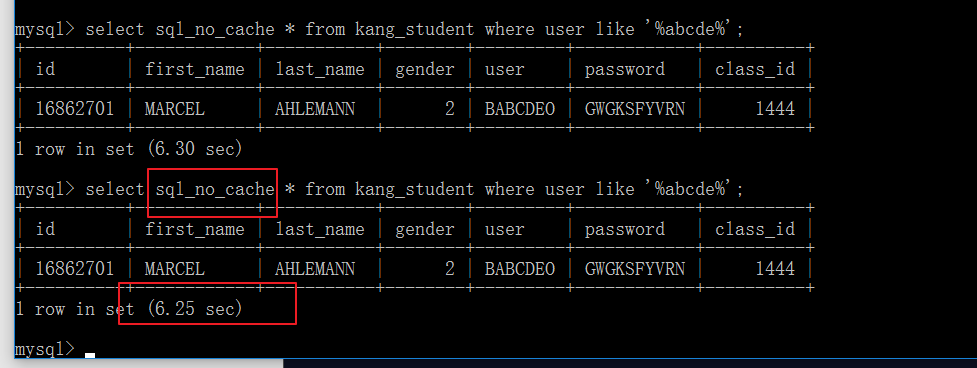


缓存空间不足, 也会使用LRU算法, 去失效缓存!

## 提示不使用缓存

缓存只要开启, select就会被缓存.

如果明确: 某个select的结果, 仅仅使用一次, 结果不需要被缓存. 或者某些使用频率低的结果, 数据量很大. 可以要求select结果不被缓存 ,使用语法hint(提示)完成:



# 分区,分表(分布式存储策略)

当一张表中, 出现很大的数据量时, 将这些数据分别存储到不同的存储文件中, 称之为分区(分表).

MySQL服务器内部提供的策略, 称之为分区, partition.

用户应用程序自定义完成的策略, 称之为分表.

## 分区, partition

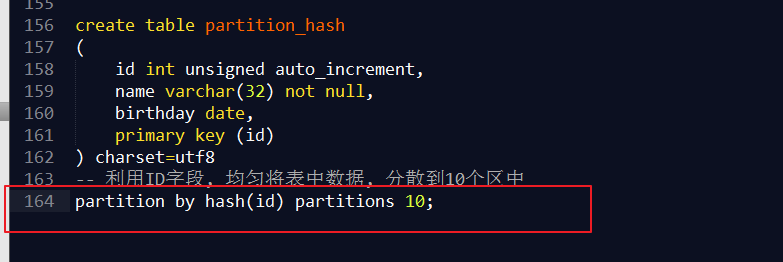
在MySQL5.1之后, MySQL服务器, 自动提供数据表的分区功能.

区: 一个用来存储表中数据和索引的存储机制. (就是一个表)

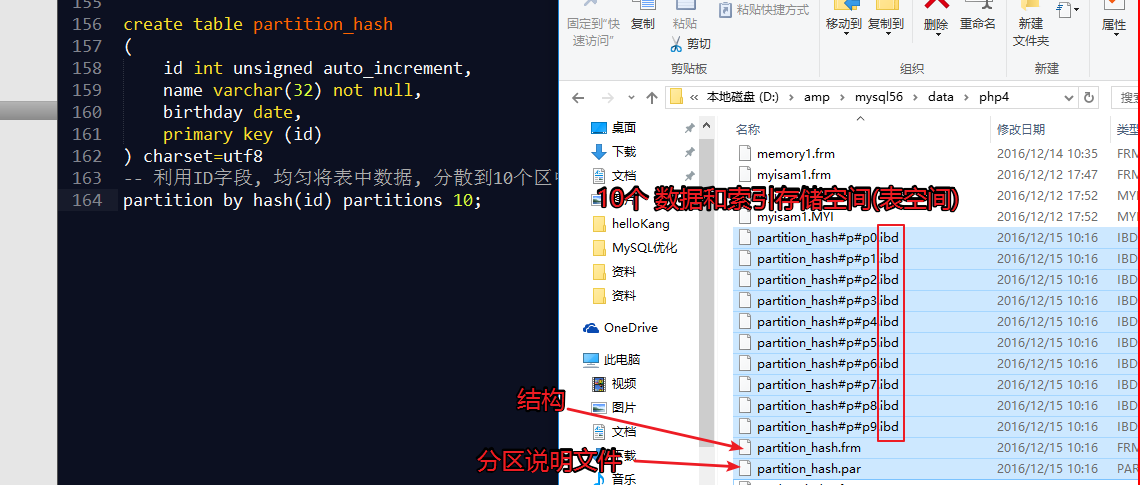
使用特定的分区语法, 可以将表划分成多个分区.

快速演示, 利用主键, 将数据均匀分布在不同的10个区中:(数据没有明显的逻辑界限)

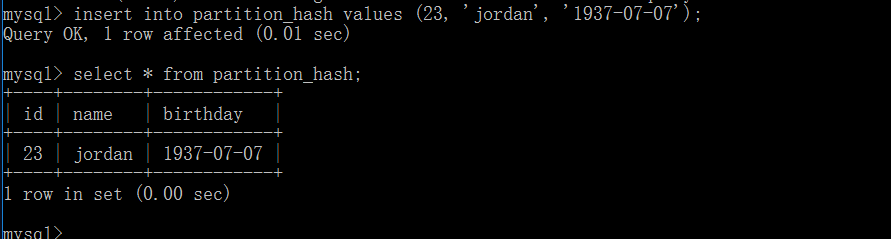
演示语法如下:



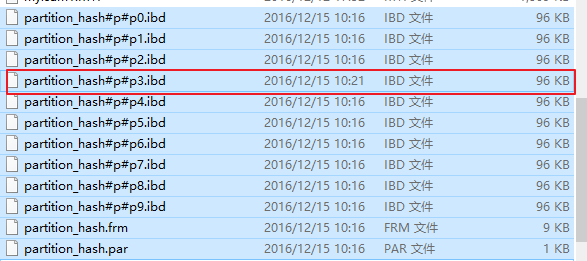
以上的建表语句, 形成的表文件如下:



插入一条记录, 要存储在某个分区中:



存储在了对应的分区p3中



可以发现利用 id%10 完成的分区, 这就是hash分区算法的实现方式!

数据在内部存储时, 使用了不同的存储空间. 但是数据在被客户端使用时, 逻辑上的效果在一张表中.

使用了分区后, 客户端程序不需要发生改动. 基于服务器端的分布式算法.(对比memcahce基于客户端的分布式算法)

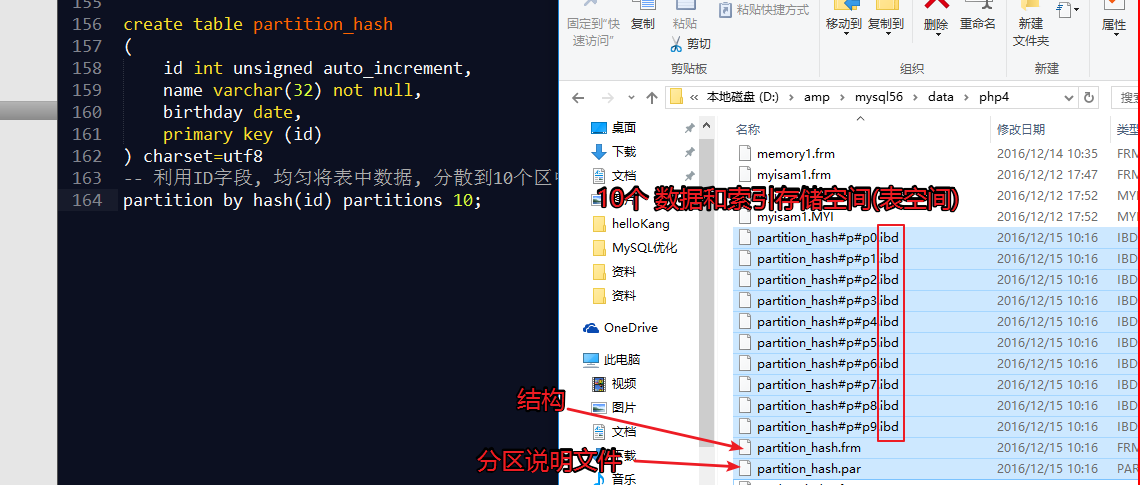


## 分区算法

MySQL服务器提供了4种分区算法为

### hash, 均匀分配(求余)

提供一个整数值字段, 利用该字段的值, 对分区数量进行求余, 划分分区.



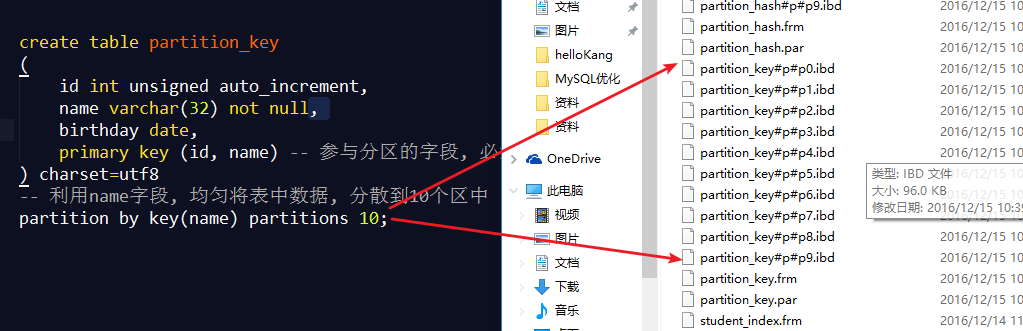
按照ID字段, 均匀分布在10个分区中

### key, 均匀分配(计算可以的整数值, 再求余)

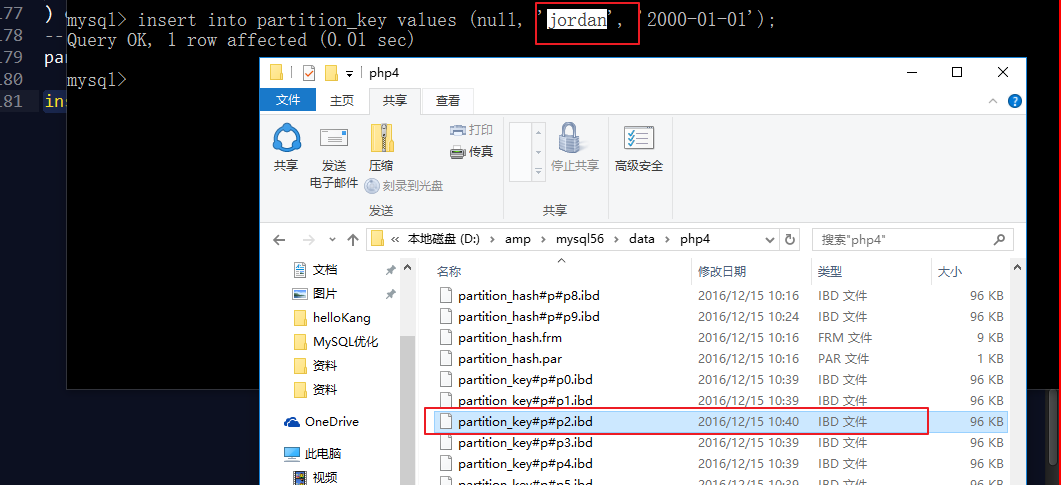
支持非整数字段, 先利用该字段, 计算整数值(类似crc32), 然后, 再求余运算.

key的可以使用任意类型的字段, 进行平均分配.

例如, 使用name字段, 将用户平均分配:



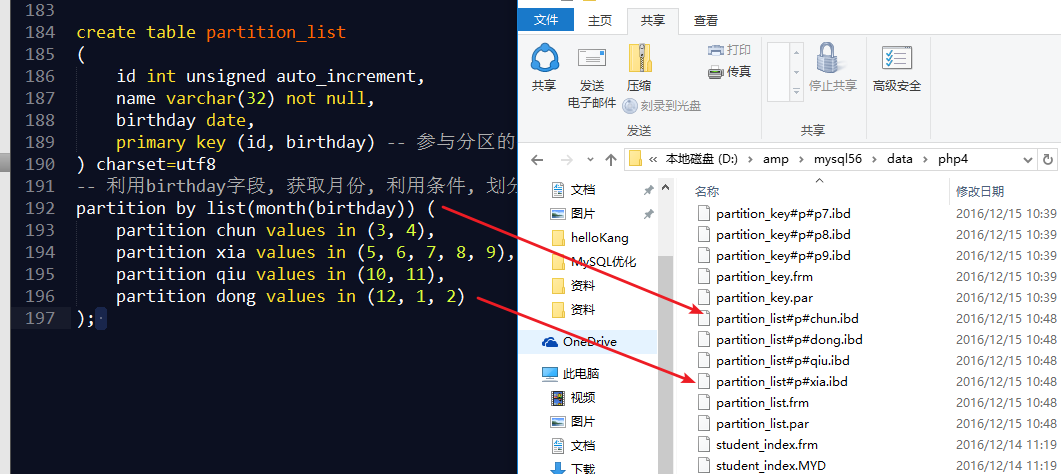
name字段, 可以为字符串类型



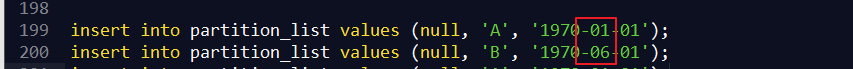
### list, 按业务逻辑分配(列表值条件)

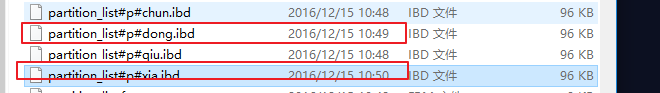
需要提供每个分区所存储数据的逻辑条件. 满足条件的数据, 可以被存储到该分区中:

例如, 按照生日中的月份, 将用户分成 春夏秋冬 各个季节进行存储:



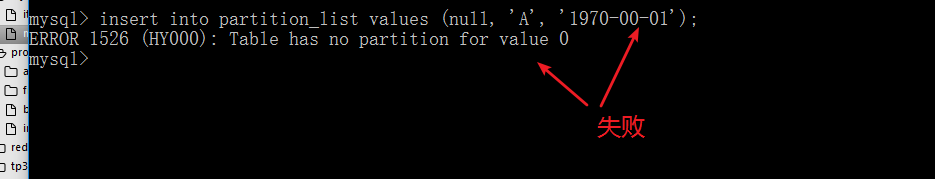
插入数据测试, 明确分区:



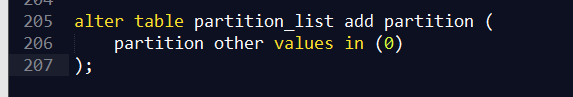


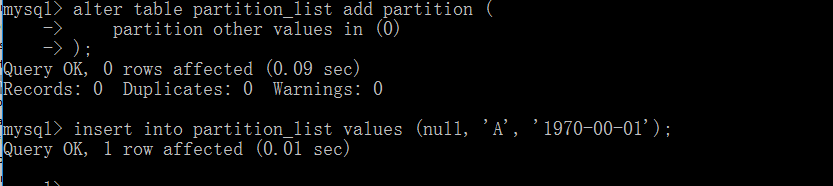
条件分区, 常见问题, 就是条件不完善:

导致, 溢出的条件, 数据无法进入到数据表中.



需要修改分区的定义完成, 增加一个分区 专门存储值为0 的分区:



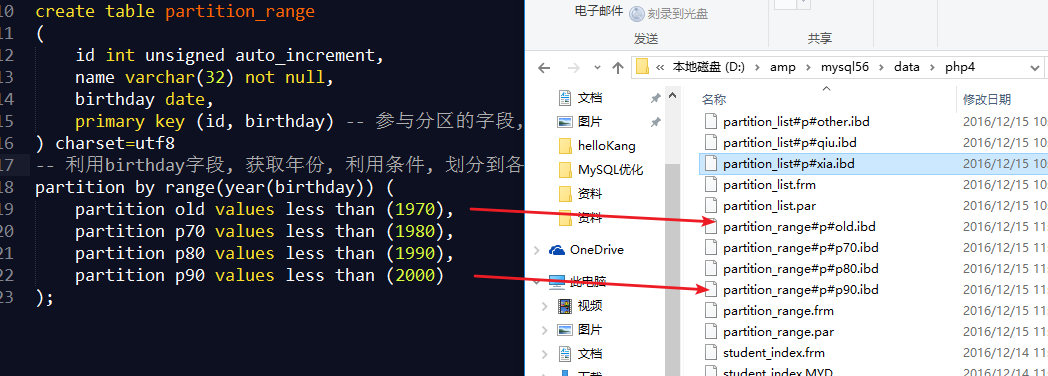


### range, 按业务逻辑分配(区间范围条件)

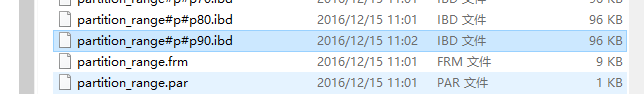
提供一个小于表达式, 作为分区的条件. 仅仅支持小于表达式:

Less than

例如, 利用生日中的年份, 将用户, 划分为80后, 90后等:





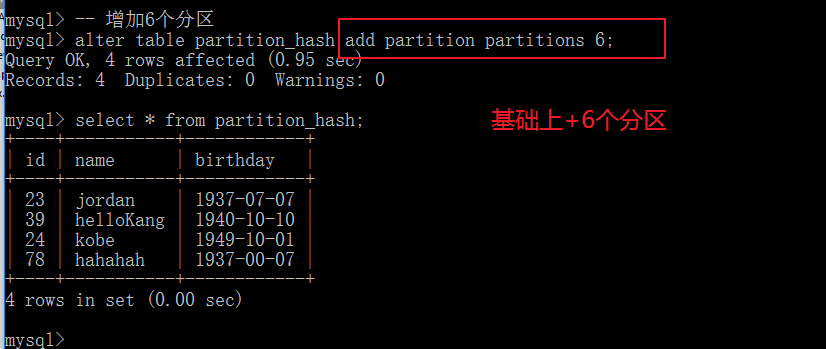


## 分区支持管理

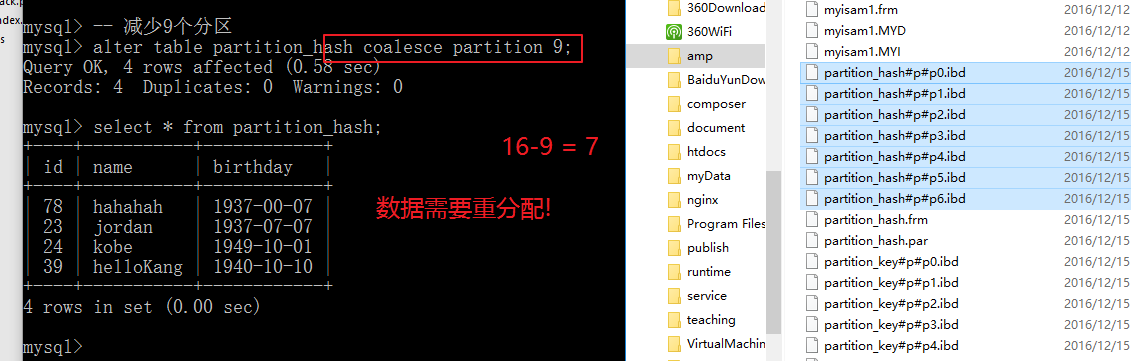
修改 hash, key分区的数量

分区数量更改了, 将数据重新分散到不同的分区中.

增加数量:

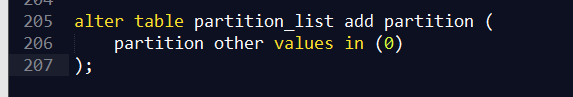


减少(合并分区)数量:



增加或删除 range, list分区

添加分区, add partition指定分区的条件即可



删除分区:

alter table partition\_list drop partition dong;



## 建议的分表算法

对客户端透明!

在新版的MySQL上, 支持, 将不同的分区, 存储在不同的磁盘上, 提示分区的并发性能.

分区是不区别于存储引擎的.

## 用户分表, 水平分表

分表, 将数据分散到不同的数据表中.

体现:

数据库中, 存在多张结构相同的表.

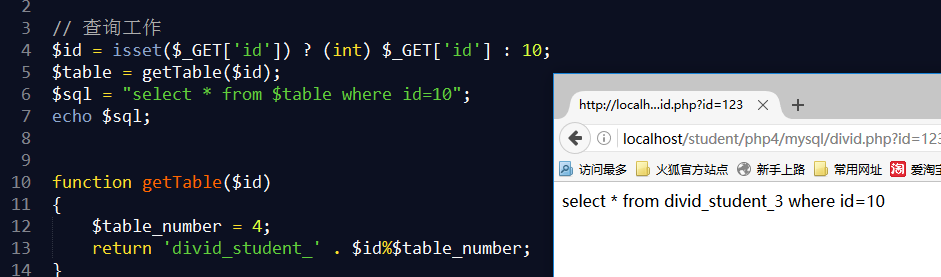
分表算法, 由客户端程序实现.(PHP程序实现)

|  |
| --- |
| create table divid\_student\_0  (  id int unsigned auto\_increment,  name varchar(32) not null,  birthday date,  primary key (id)  ) charset=utf8;  create table divid\_student\_1  (  id int unsigned auto\_increment,  name varchar(32) not null,  birthday date,  primary key (id)  ) charset=utf8;  create table divid\_student\_2  (  id int unsigned auto\_increment,  name varchar(32) not null,  birthday date,  primary key (id)  ) charset=utf8;  create table divid\_student\_3  (  id int unsigned auto\_increment,  name varchar(32) not null,  birthday date,  primary key (id)  ) charset=utf8; |

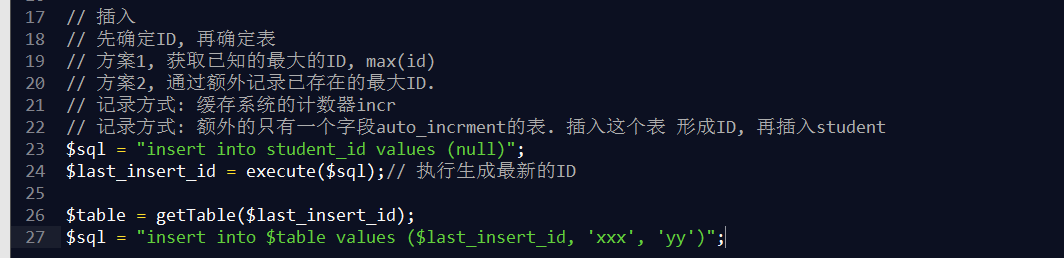
PHP程序, 需要发生改动:

演示最常用的, 使用ID做求余划分:

通过ID计算表名



插入时的处理:



采用分表, 在分区不能满足自定义的业务逻辑时, 才会自定义.

## 分区和分表, 都叫水平分表

将表中的记录, 分散到不同的存储中!

## 垂直分表

将表中的字段, 进行拆分.

主要是基于业务逻辑.

表中的字段, 明显的常用和不常用, 业务逻辑上的分离线. 可以由2张或多张表完存储:

例如:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | 姓名 | 身高 | 籍贯 | 政治面貌 | 年龄 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

姓名, 身高, 年龄, 常用字段(90%)

籍贯, 政治面貌, 不常有字段(10%)

垂直一刀:

常用信息表(90%)

|  |  |  |  |
| --- | --- | --- | --- |
| ID | 姓名 | 身高 | 年龄 |
|  |  |  |  |
|  |  |  |  |

额外信息表(10%)

|  |  |  |
| --- | --- | --- |
| ID | 籍贯 | 政治面貌 |
|  |  |  |
|  |  |  |

# 架构 – 一台MySQL不够, 需要多台MySQL协同工作

多MySQL是何种结构, 称之为如何架构的:

读写分离, 主从复制, 负载均衡.

如图所示

## 读写分离

将读,写操作, 分开到不同的MySQL服务器完成. 称之为读写分离.

典型的web项目, 都是读多写少. 7:1 到 10:1.



## 主从复制

将数据从 写服务器 , 拷贝到读服务器上去.

MysQL提供了repliaction复制技术,. 完成该拷贝操作



## 负载均衡

读多写少. 7:1 到 10:1

读的检索也相对复杂. 通常需要提供更多的服务器完成读操作.

典型的架构, 一主多从. 一写多读.

需要一个算法, 工具, 完成在多台读服务器中, 完成选择某个读服务器的操作, 就称为负载均衡. (轮询就是最常用的负载算法)

实操中, 读写分离算法, 于 负载均衡算法, 都需需要一个专业的代理完成, 而不是交给客户端完成.



演示的过程, 就是安装软件和配置的过程.

## 安装模拟

### 准备2台MySQL服务器

192.168.118.1, 写, 主

192.168.118.128, 读, 从

需要在从服务器端, 可以登录上主服务器.



### 配置复制

#### 配置主服务器118.1: 开启二进制日志, 配置服务器ID

my.ini

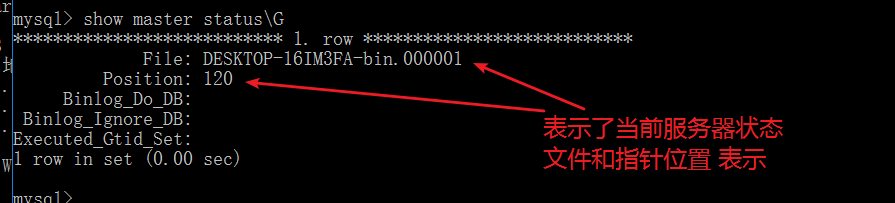


重启MySQL服务器

#### 记录主服务器118.1的状态

show master status

服务器进度状态, 使用(bin-log)日志文件及日志文件的内部指针表示.

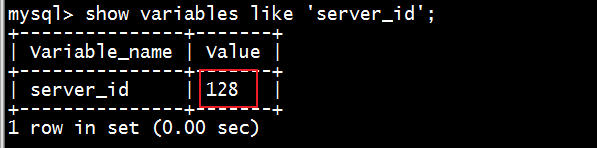


#### 配置从服务器118.128: 配置一个独立的server-id

/usr/local/mysql/my.cnf



重启 mysqld



#### 主服务器上, 增加用于复制的权限帐号

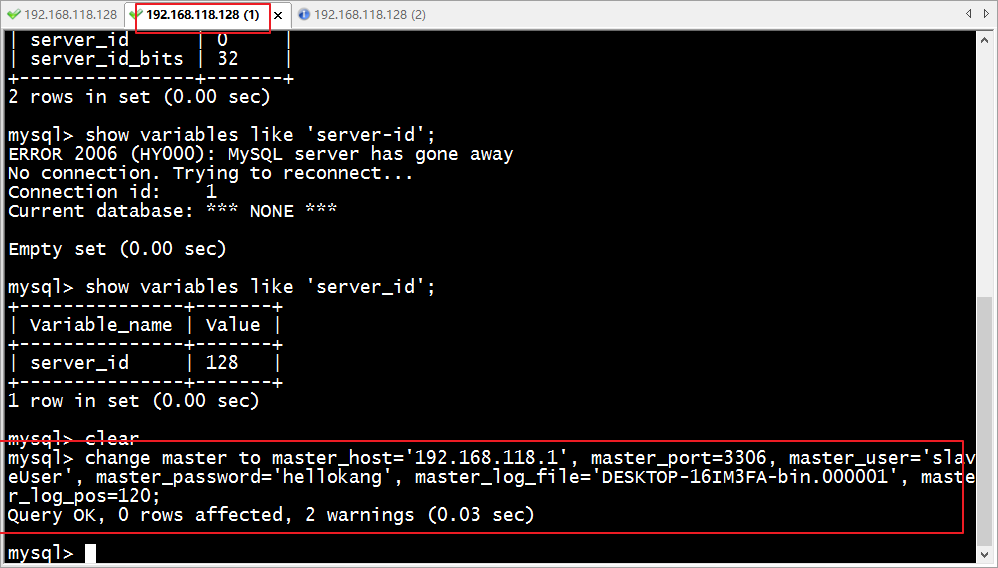
就是在从服务器上, 执行复制的帐号, 拥有全局replication slave权限

(全局权限, 需要设置在\*.\*层面, 整个数据库层面)



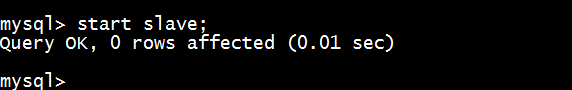
#### 从服务器上, 设置复制来源

在从服务上, 执行change master to指令完成, 由主服务器的主机, 端口, 用户名, 密码等构成



#### 从服务器开启复制

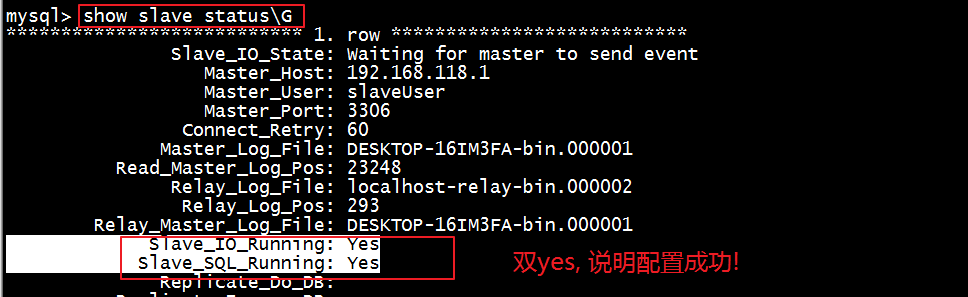
start slave;



(PS, stop slave,关闭复制)

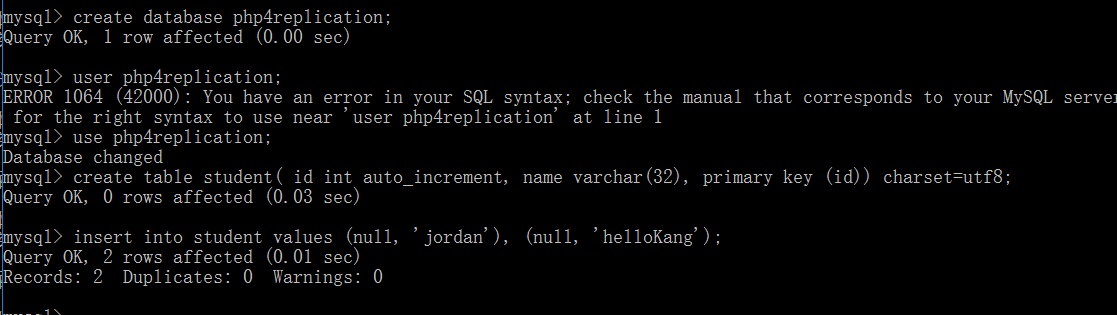
#### 看从服务器状态, 确定当前复制操作结果

show slave status\G



#### 测试复制

主, 写



从, 读

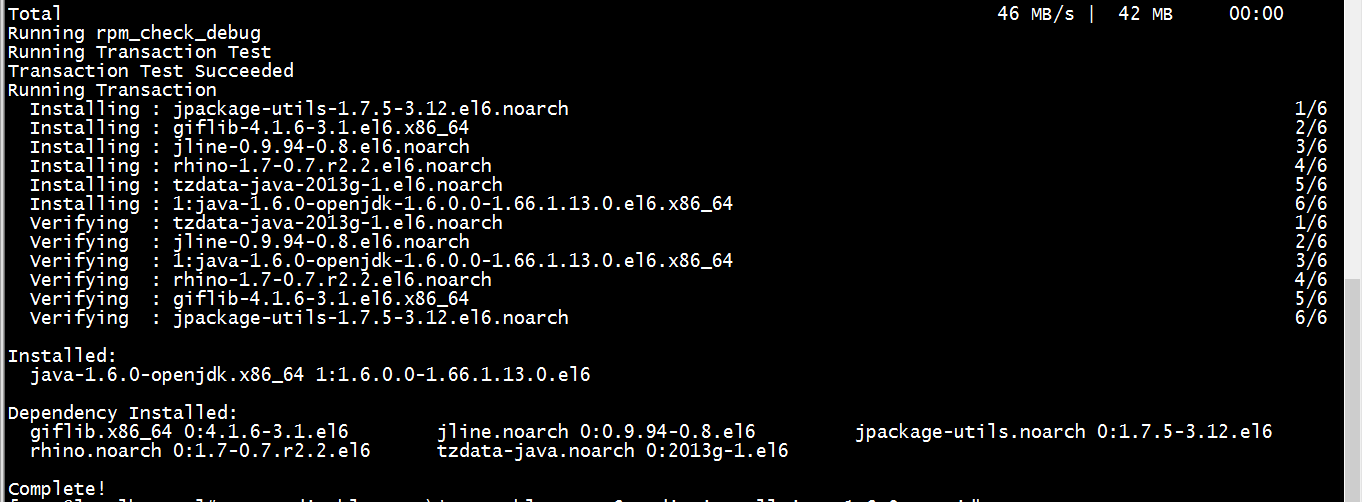


### 配置读写分离和负载均衡代理软件

课堂使用 google amoeba(变形虫)软件进行展示, 其他软件类似

#### yum安装java运行环境 jdk(jre)

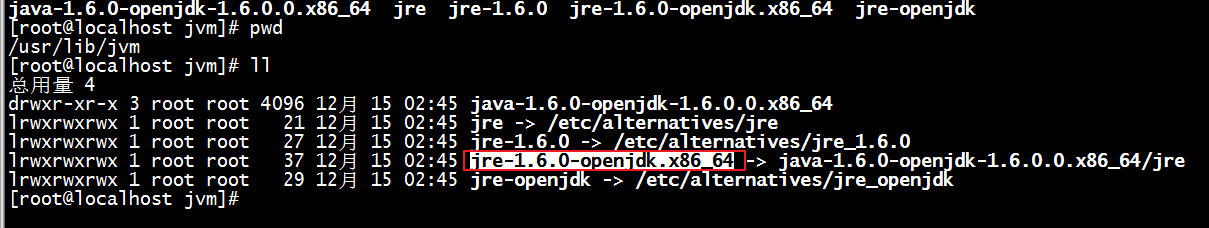
yum --disablerepo=\\* --enablerepo=c6-media install java-1.6.0-openjdk

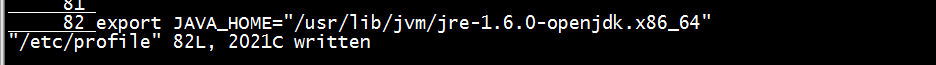


#### 配置环境变量JAVA\_HOME

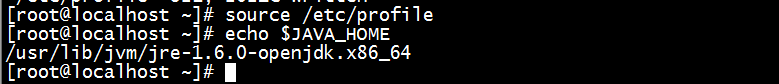
vi /etc/profile

JAVA\_HOME指向下面的目录



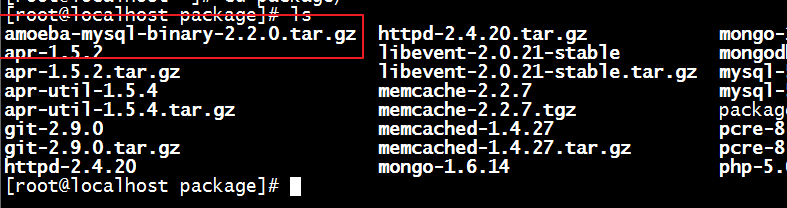


source生效

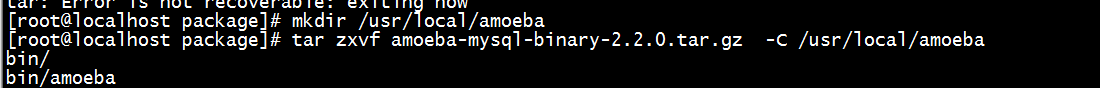


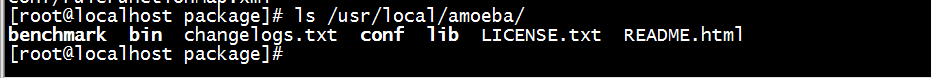
#### 安装amoeba

直接解压执行即可;



解压到, /usr/local/amoeba

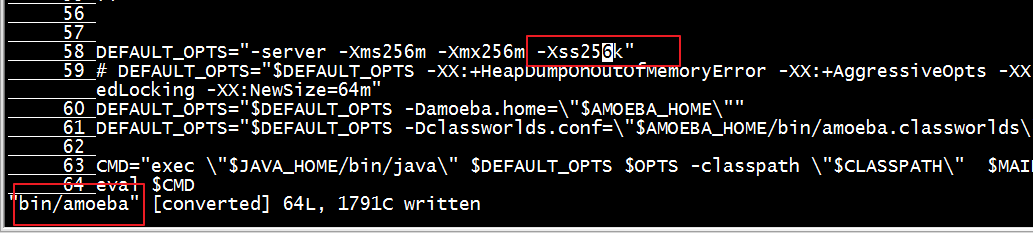




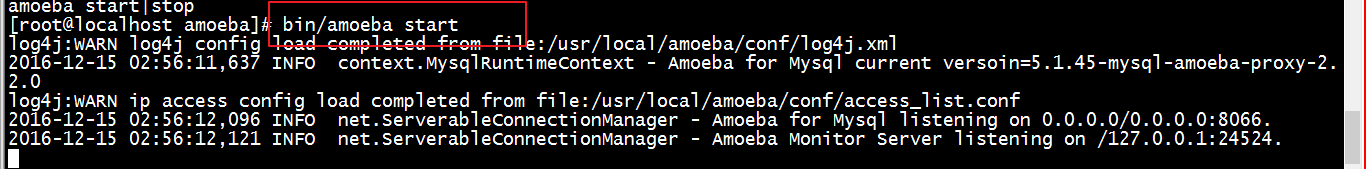
测试执行



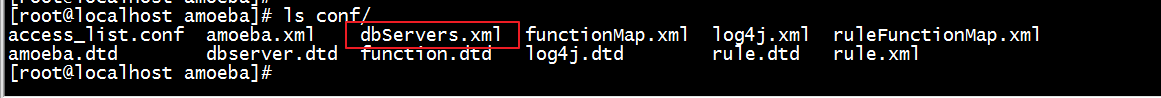
出现一个问题, 修改运行脚本即可:



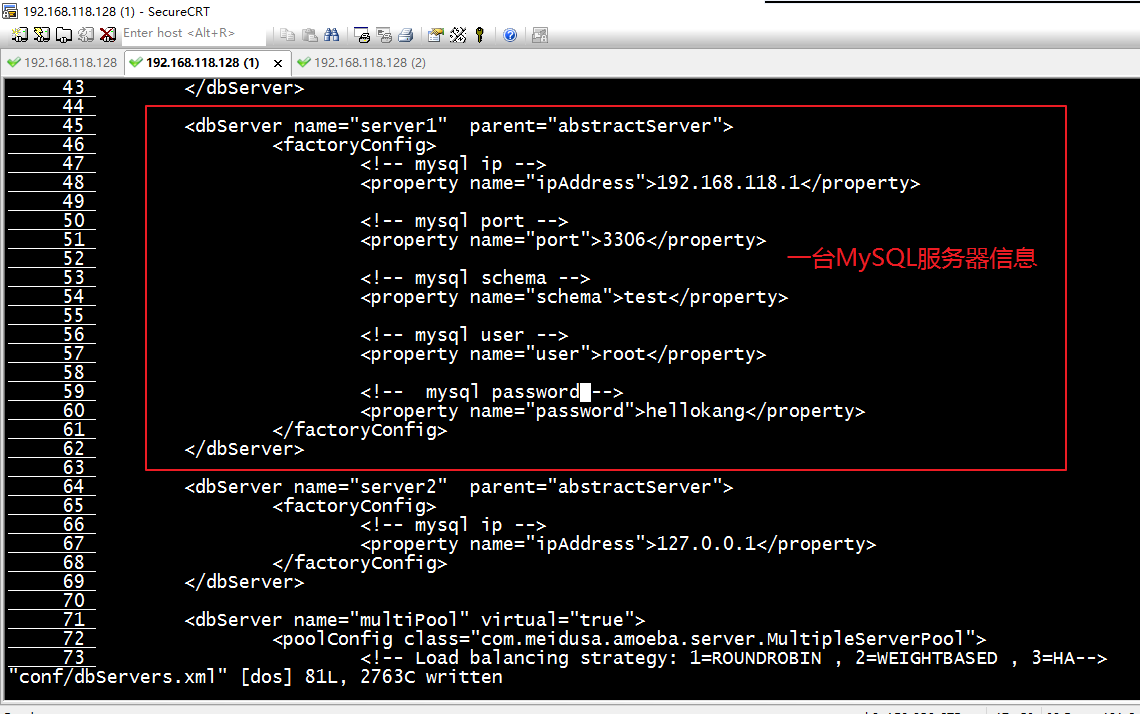
测试运行

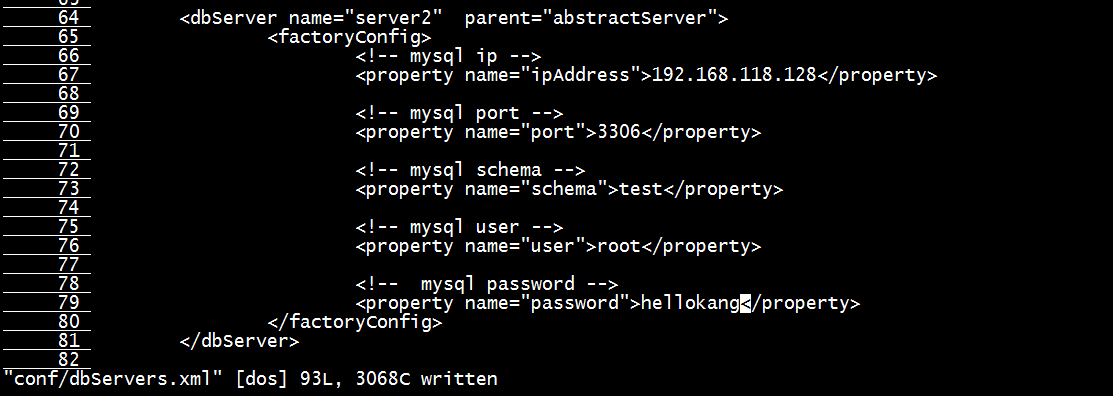


#### 配置 amoeba管理的MySQL服务器

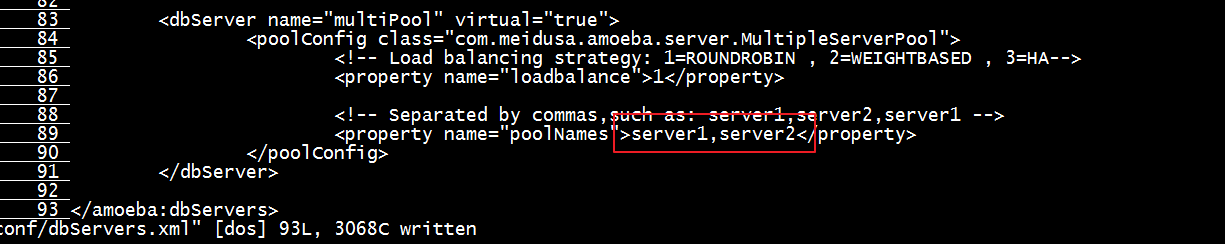


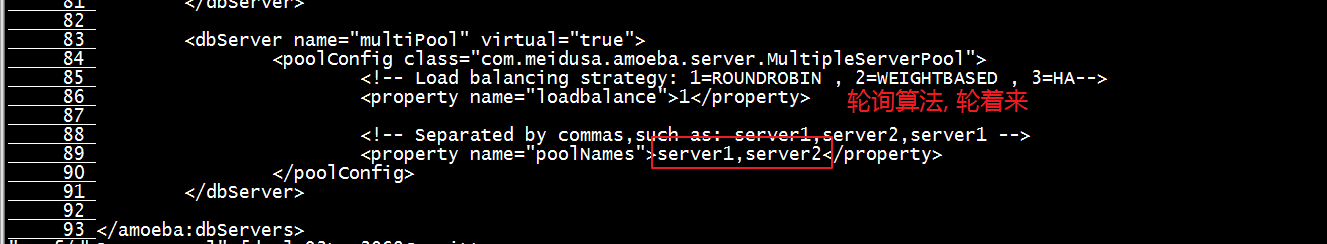
配置多个合理的DBServer段, 操作多台MySQL服务器





通常, 读服务器由多台MySQL服务器组成的集群. 需要在配置服务器时, 指定哪些MySQL服务器组成这个多台的MySQL集群.





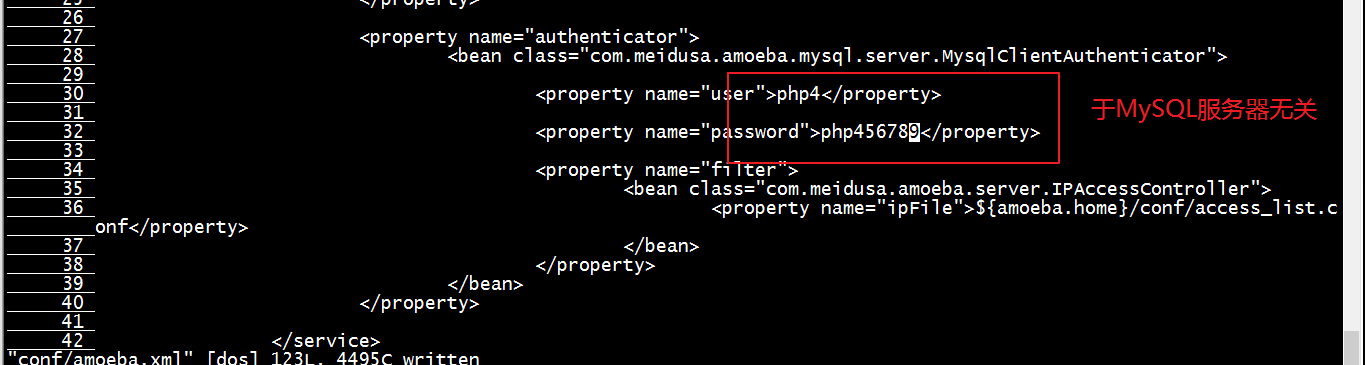
server1, server2, 组成读服务器集群, 采用roundrobin算法, 完成负载!

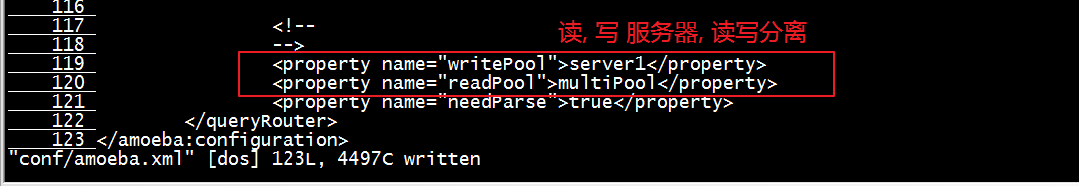
server1, 同时承担写服务器任务.

#### 配置amoeba本身

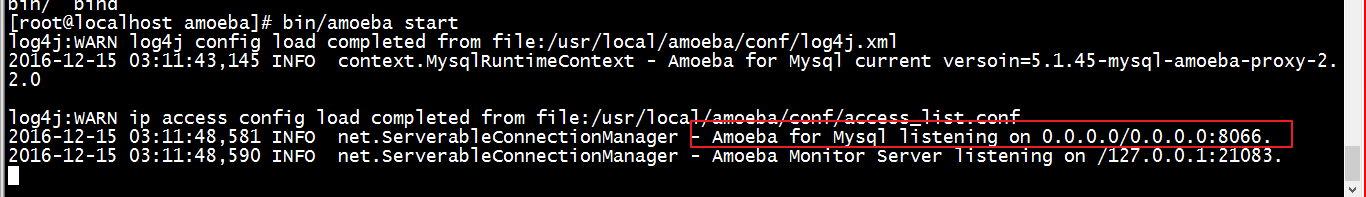


配置认证信息



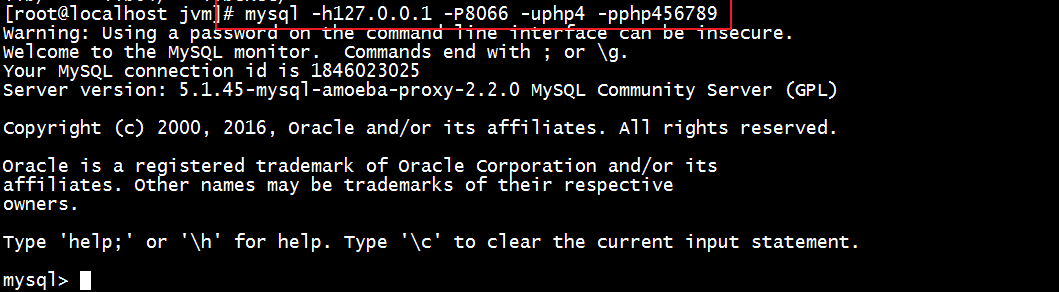


#### 启动amoeba

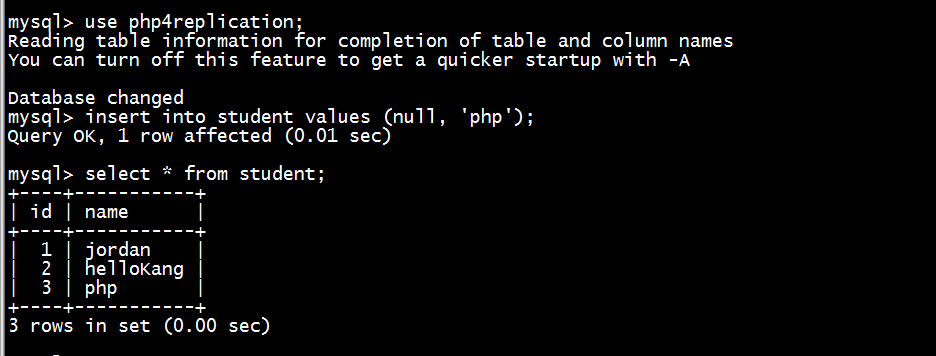


#### 测试

使用MySQL客户端, 连接Amoeba



常规操作即可



emoebo自动完成读写分离于负载均衡.

## 冷热数据分离

常用与不常用的数据分开.

常用的技术:

分表. 将冷数据, 存储在特殊的表中.

缓存+持久化(缓存预写+重做. 热数据, 先存缓存(redis), 定时执行将缓存中的数据同步到磁盘上.

## 高可用, HA, high availiable

服务器尽可能不宕机.

7x24 都在线.

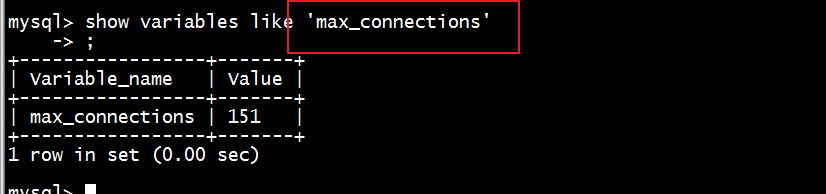
高可用的实现, 就是冗余.留一台备用.

单点服务器必须冗余高可用.



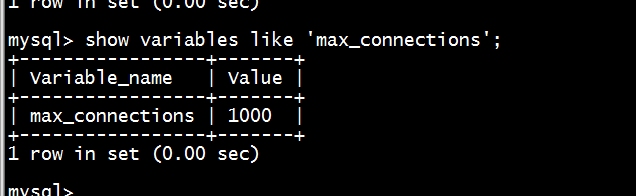
## 服务器配置

### max\_connections, 最大连接数

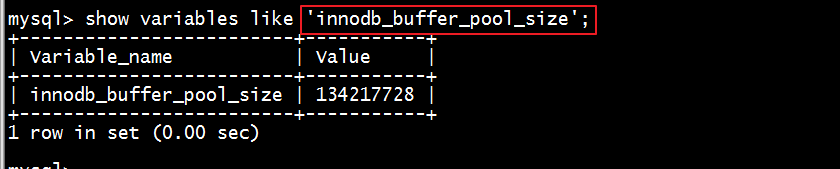


建议在配置文件中修改. 系统级别的配置





### innodb\_buffer\_pool\_size



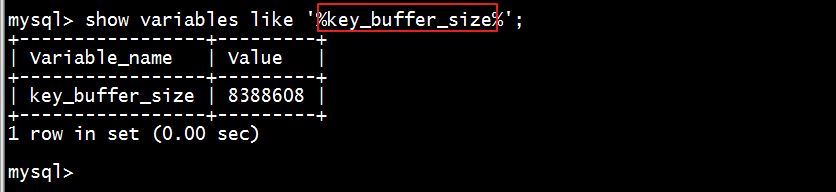
innodb缓冲池

innodb中 绝大多数的缓存, 都缓存于该缓冲池!

该配置, 可以尽可能的增加, 80%内存!

### key\_buffer\_size

索引缓存, 主要针对于myisam



# SQL层面

## explain, 分析SQL的执行计划

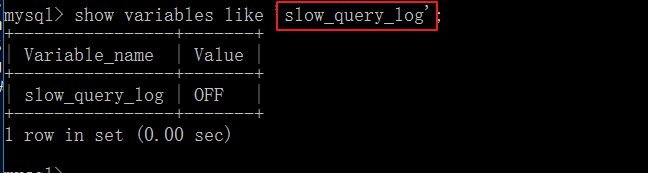
目前仅仅针对于select语句, 其他语句也被纳入计划.

## 慢SQL查询日志

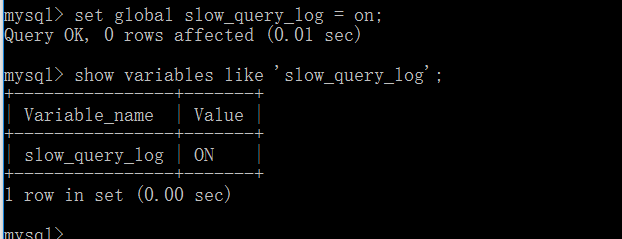
用于记录, 执行时间超过某个时间临界点的, 慢速的查询SQL.

快速定位, 哪条SQL需要被优先优化的工具.

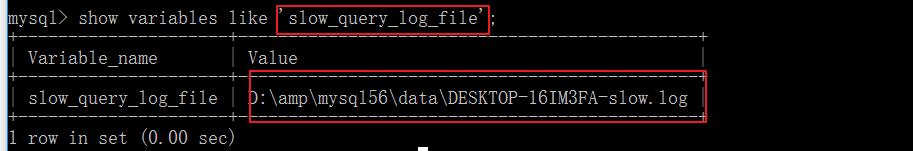
### 开启慢查日志



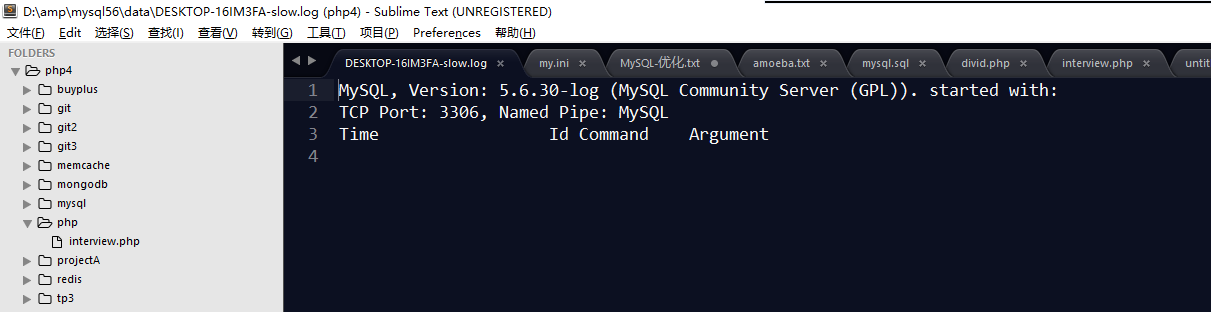
修改为on就是开启慢查询日志



一旦开启, 就会在一下位置, 生成慢查询日志文件:

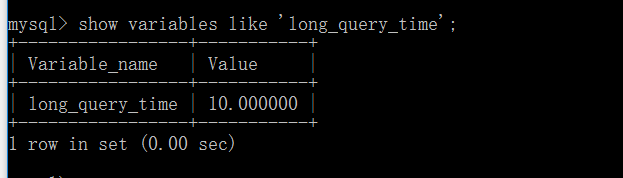


初始化好的, 空的慢查询日志

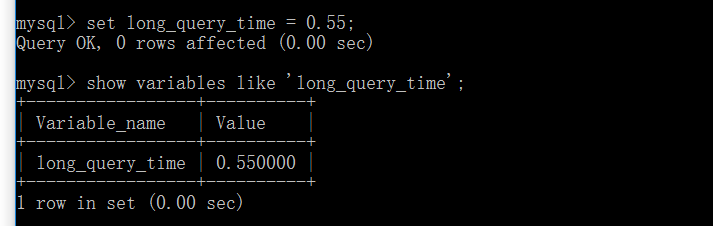


### 设置时间的临界值

默认的时间临界值为10s

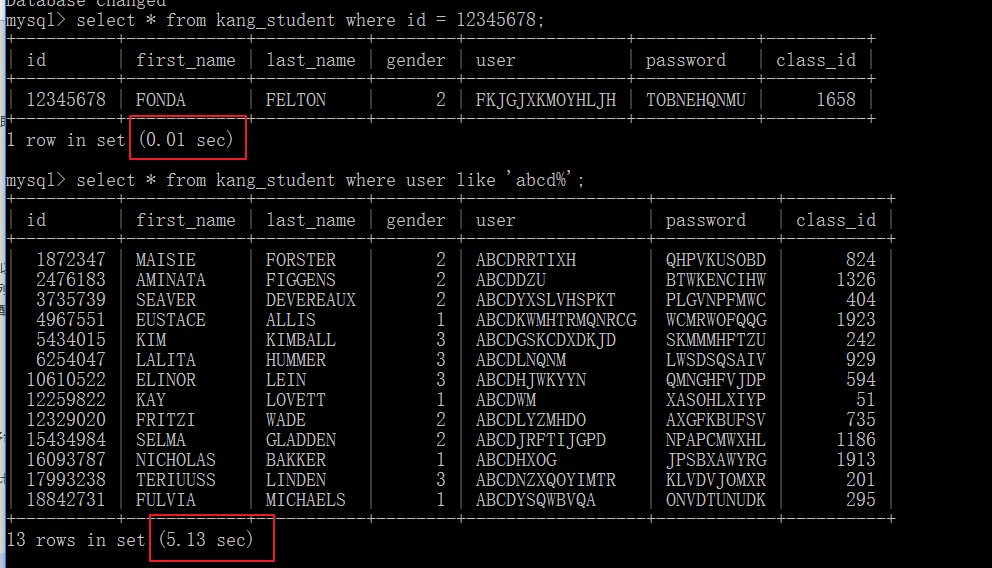


可以修改:

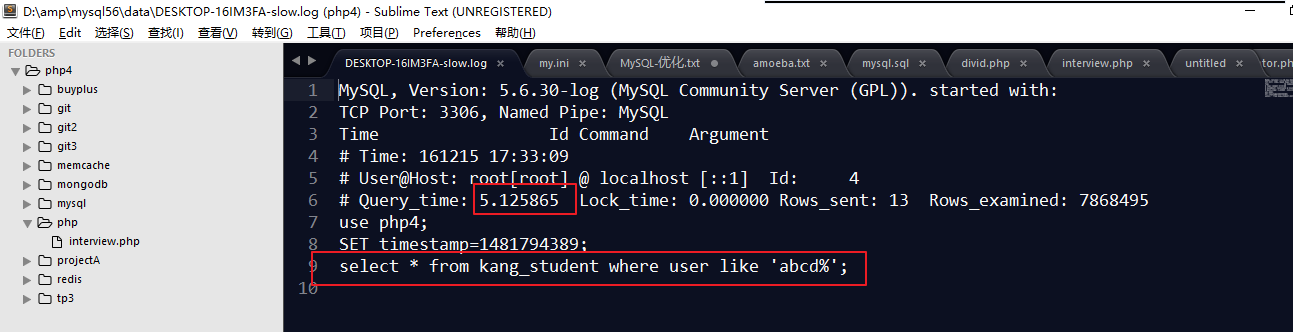


### 记录慢查询

范式执行时间超过临界值, 都认为是慢查询



查看, 慢查日志:

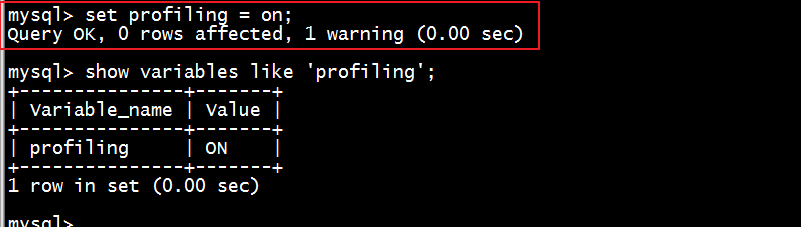


通常, 从大的时间临界, 一直测试到小的时间临界.

## profile机制

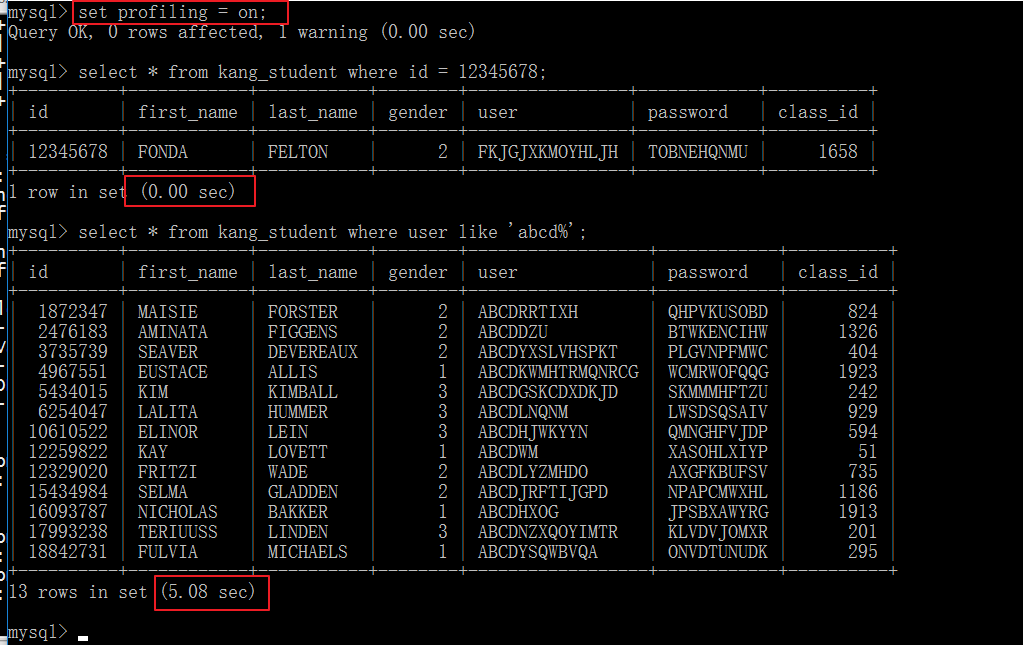
用于分析一条SQL的详细的执行时间

### 开启profile



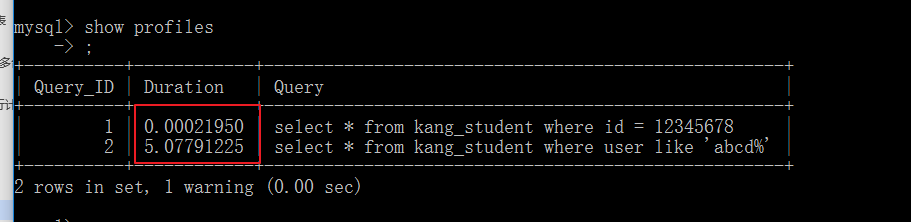
一旦开启, 详细记录, SQL的等执行时间

### 执行SQL



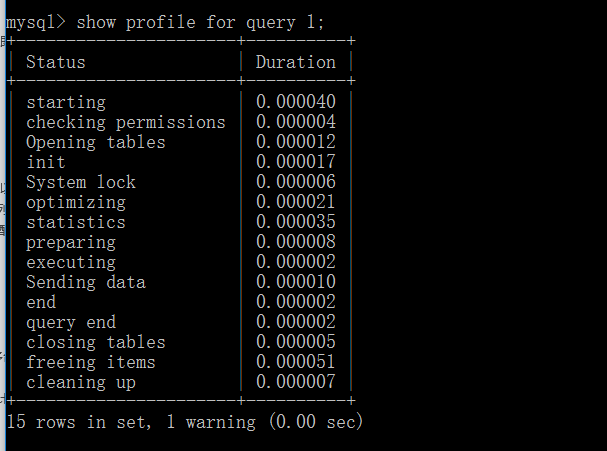
### 获取概要信息

show profiles 语法完成



### 查看某条SQL详细执行时间

show profile for query N; query\_id



## limit优化

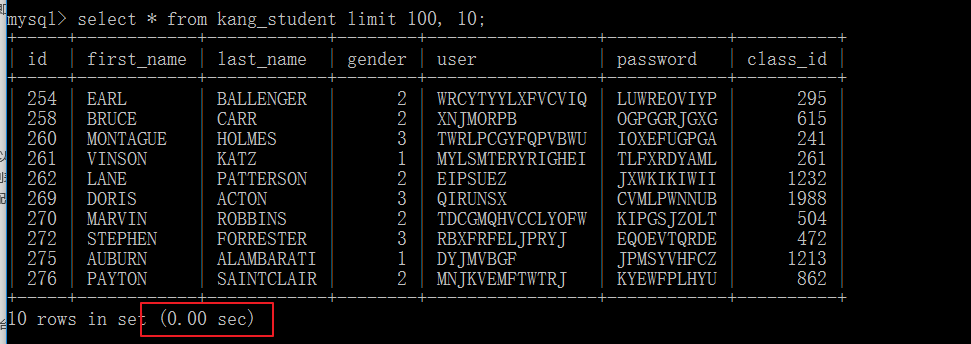
### limit 1

如果你确定, 仅仅需要获取一条记录, 请在SQL后加上limit 1. 包含select, update, delete操作.

select \* from table limit 1

### limit 分页

limit offset, pagesize;



当页码变大时: 变得很大时:



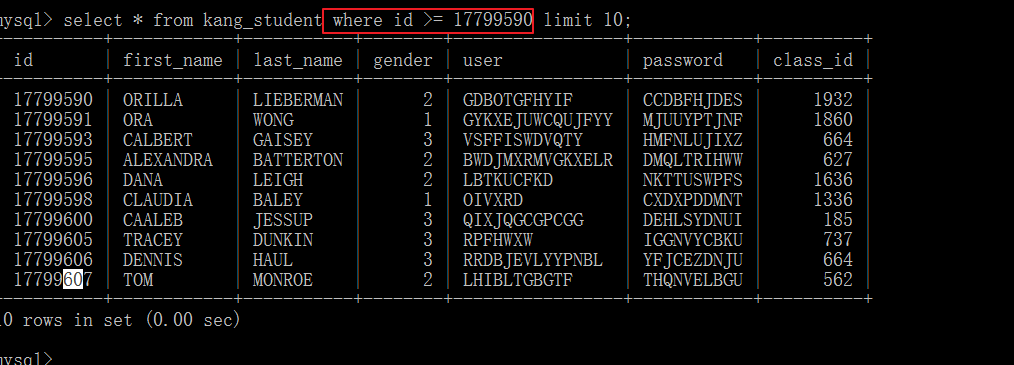
执行时间变长了.

慢在了offset上了.

limit的操作流程是, 先从数据表, 匹配到满足条件的记录, 通过offset, 偏移 ,跳过前面的N条记录, 获取后面的某个记录.

舍弃掉了, 匹配的记录. 做了大量的无用功,花费了大量的时间.

如果将以上的语句改为:



使用条件过滤, 在 SQL的检索阶段, 就放弃了大量的数据.

避免了 offset跳过已找到的无用数据问题.

当系统出现大页码时.

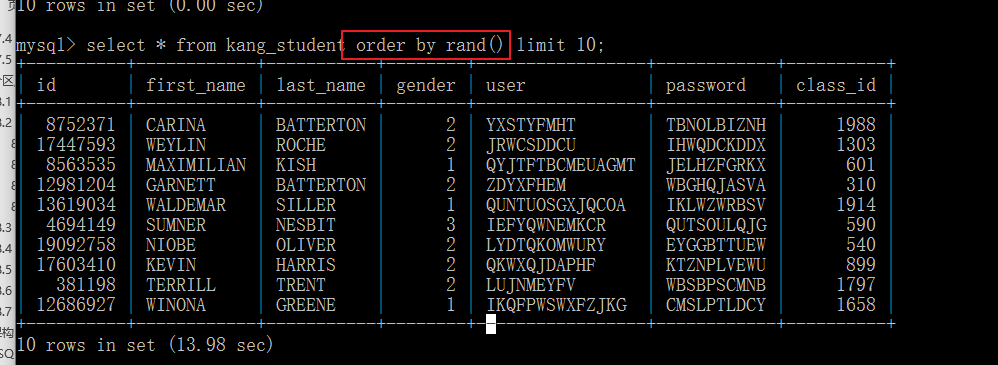
1, 尽可能使用条件过滤数据, 而不是offset, 利用条件上的index, 可以加快整体的速度.

2, 如果可能, 在业务逻辑层面,做一些额外的处理. 例如, 如果页码过界, 下界上届, 处理一下

## order

### order by rand(), 尽可能少用

通常用于随机获取记录的逻辑:



慢在 所有的记录都需要计算随机数!

再排序

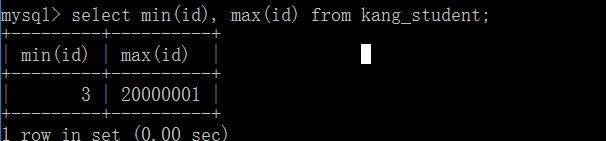
需要获取随机的10个人.

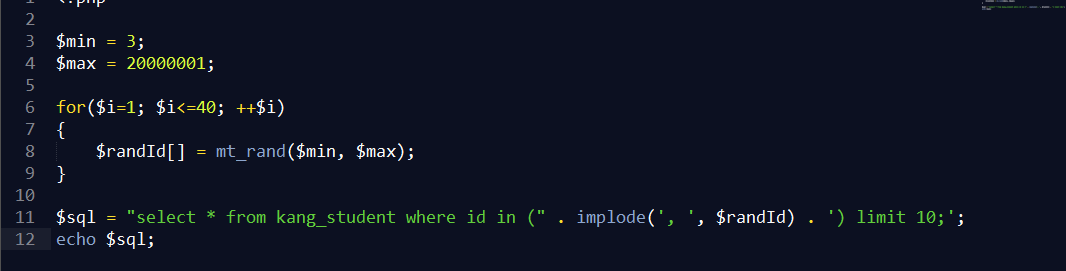
先找出随机的10个(或者多个)\_主键值, 再获取记录.

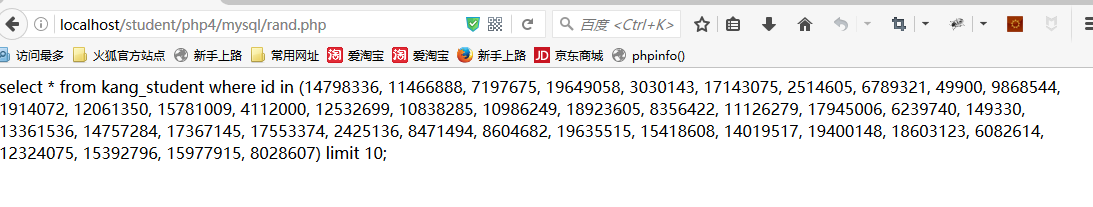
执行10次 = mt\_rand(minid, maxid).

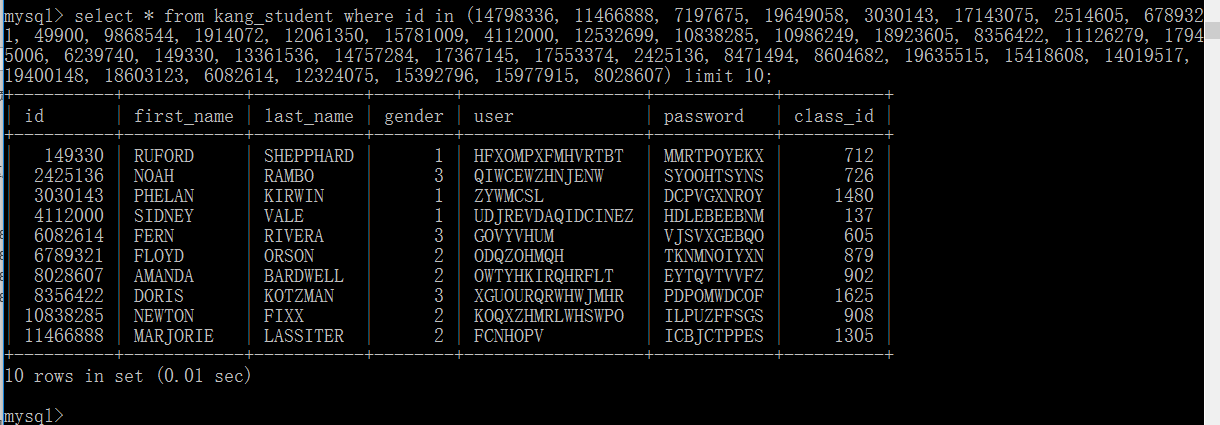
利用主键作为条件, 检索, 速度快很多

PHP代码演示:





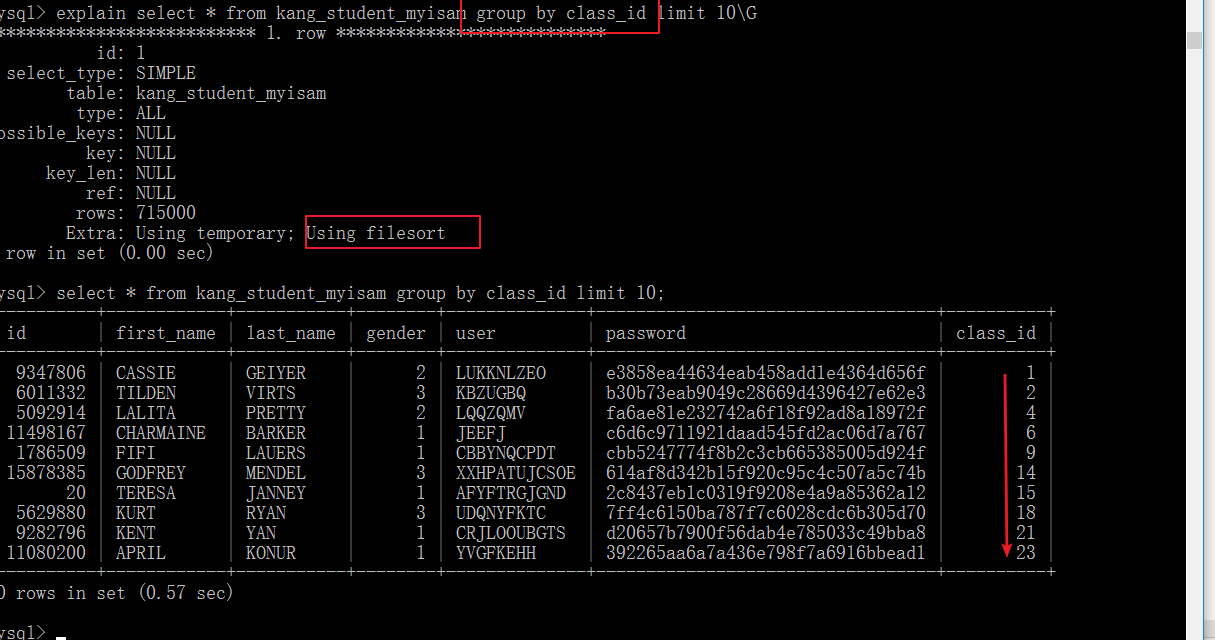




### order by null

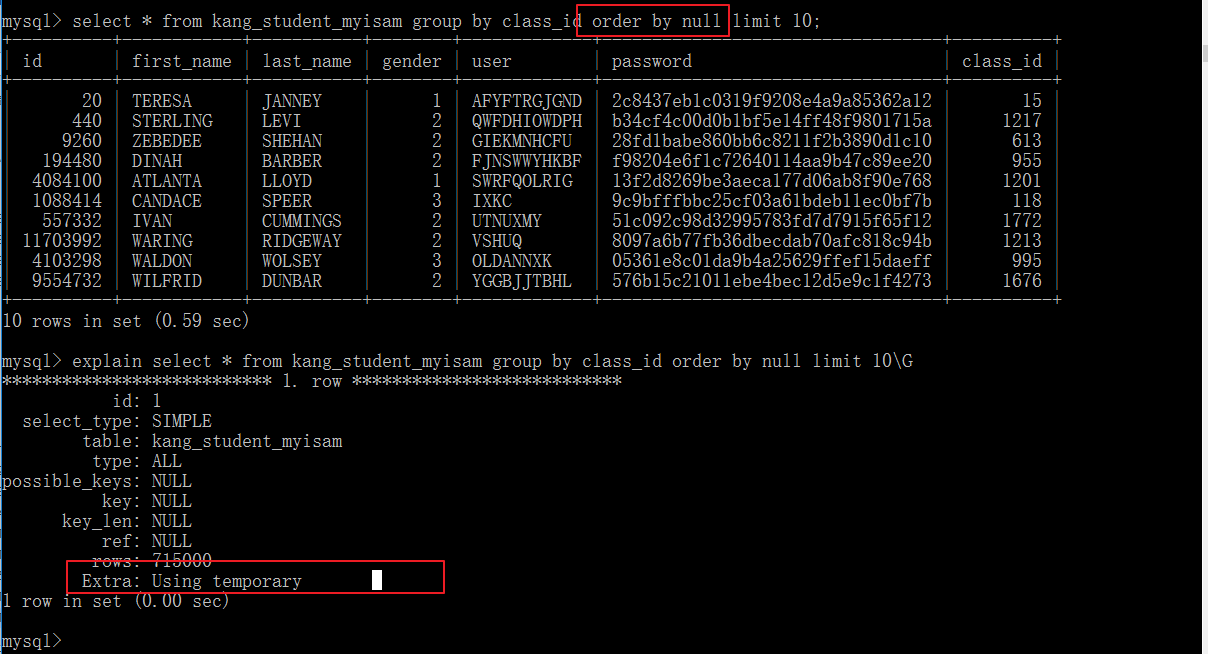
禁止排序.

group by, 默认会依据分组字段进行排序.



(如果需要这个排序, 在分组字段(排序字段)加索引)

如果不需要该排序, 可以使用order by null 禁用:



## 单表对多表查询

单表查询: 一条SQL仅仅涉及一张表. 拆分业务逻辑, 并发的提升.

多表查询: 一条SQL设计到多张表. 业务逻辑快速解决

涉及三张表:

一条条: 1, 2, 1

一条: 2.5

在典型的AR的关联模型中, 就是一条条的执行.

# Iterator告白

习题: 球队类, 遍历球队类对象, 获取每个球员:



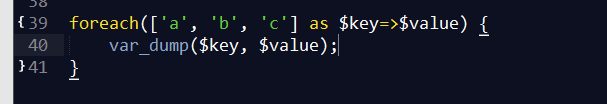
(认同, 对象是可能被遍历的)

默认的foreach遍历, 将对象当作集合型数据类型对待, 遍历的所有的可以访问的属性, 而不是按照我们的要求进行遍历.

此时,需要控制foreach的行为, 就需要给类对象, 提供更多的功能, 需要继承自Iterator的接口:

该接口, 实现了foreach需要的每个操作.

foreach的执行流程如下:





看图例中, foreach有几个关键步骤: 5个.

|  |
| --- |
| // 球队类  class Team implements Iterator  {  // private $name = 'HelloName';  // private $title = 'HelloTitle';  // private $subject = 'HelloSubject';  private $playerList = ['HelloKang', 'HelloTitle', 'HelloSubject'];  private $name = '北京首钢';  private $city = '北京';  public function rewind()  {  reset($this->playerList);// 重置数组指针  }  public function valid()  {  // null 没有元素 返回false  // 不是null 返回true  return ! is\_null(key($this->playerList));  }  public function current()  {  return current($this->playerList);  }  public function key()  {  return key($this->playerList);  }  public function next()  {  return next($this->playerList);  }  }  $object = new Team;  // 遍历球队对象时, 获取每个球员进行操作  foreach($object as $property => $value) {  var\_dump($property, $value); echo '<br>';  } |

而,Iterator迭代器, 中所要求实现的5个方法, 就是用来帮助foreach, 实现在遍历对象时的以上的5个关键步骤:

当foreach去遍历对象时, 如果发现对象实现了Ierator接口, 则执行以上5个步骤时, 不是foreach的默认行为, 而是调用对象的对应方法即可:



编码实现:

# 前++告白

# global关键字告白