MySQL

MySQL 5.7
---进阶---
MySQL 内核结构与原理
服务器优化
建模优化
索引优化
查询优化
主从复制
性能监控
容灾备份
SQL 编程

- **整体架构**

1. MySQL 客户端通过协议与 MySQL 服务端建立连接，发送查询语句，服务端先查查询缓存，命中则直接返回
2. 服务端进行语句解析，通过关键字将 SQL 解析成“解析树”，并做语法规则验证、合法性校验
3. 优化器将合法的解析树转化为执行计划

  - 参数
    - 配置参数

      /etc/my.cnf

      - datadir

        /var/lib/mysql/
        mysql 数据库文件的存放路径
      - basedir

        /usr/bin
        命令目录
        - mysql
        - mysqladmin
        - mysqldump
        - mysqlbinlog
        - mysqlcheck
      - plugin-dir

        /usr/lib64/mysql/plugin
        mysql 插件存放路径
      - log-error

        /var/log/mysqld.log
        错误日志路径
      - pid-file

        /var/run/mysqld/mysqld.pid
      - socket

        /var/lib/mysql/mysql.sock
        本地连接时用的 unix 套接字文件
      - 配置文件

        /usr/share/mysql
    - 字符集参数

      ---查看---
      show variables like '%char%';

      ---建库时设定---
      create database db1n default charset utf8mb4 collate utf8mb4_romanian_ci;

      ---修改库---
      alter database db1 character set 'utf8mb4';
      ---修改表---
      alter table t1 convert to character set 'utf8mb4';

      - character_set_client

        客户端使用的字符集
      - character_set_connection

        默认的连接数据库时的字符集，可有客户端发起连接时指定

- character_set_database

  创建数据库的默认字符集，可在建库时另行指定
- character_set_filesystem

  文件系统的编码格式
- character_set_results

  返回给客户端时使用的默认字符集
- character_set_server

  服务器安装时的默认字符集，系统自行管理，不需修改
- character_set_system

  数据库系统使用的编码格式,存储元数据的字符集，默认 utf8
- character_set_dir

  字符集安装的目录

- 大小写敏感

  show variables like '%lower_case_table_names%';

  0 - 敏感
  1 - 不敏感，读写都转化成小写，以小写存储库表
  2 - 写按语句执行，读都转化为小写

  windows 默认 1
  linux 默认 0
- sql_mode

  SQL 语句校验规则

  ---查看---
  select @@sql_mode;
  ---设置---
  set @@sql_mode = '';
  - ONLY_FULL_GROUP_BY

    group by 规范
    select 的列必须出现在 group by 中
  - STRICT_TRANS_TABLES

    事务中插值失败，中断
  - NO_ZERO_IN_DATE

    不允许日期和月份为 0
  - NO_ZERO_DATE

    不允许日期为 0
  - ERROR_FOR_DIVISION_BY_ZERO

    不允许有除零，非该模式时除零返回 NULL
  - NO_AUTO_CREATE_USER

    禁止 GRANT 创建密码为空的用户
  - NO_ENGINE_SUBSTITUTION

    需要的存储引擎被禁用时抛异常，非该模式下使用默认存储引擎并抛异常

- 用户与权限

  ---密码策略---
  show variables like '%validate%';
  set global validate_password_policy=0;
  set global validate_password_length=6;
  - mysql.user

---mysql.user---
host
- % 表示所有IP
- IP/hostname
- ::1/127.0.0.1/localhost

user - 用户名，同一用户不同方式连接的权限不一样

---创建---
create user aurelius indentified by '999999';

---密码---
set password=password('999999');
update mysql.user set password=password('999999') where user = 'aurelius'; -- 5.7 不支持

---修改---
update mysql.user set user = 'aurelius' where user = 'aurelius';

---删除---
drop user aurelius;
不可以用 delete from mysql.user 会有残留信息

---生效---
flush privileges;

- grant

---授权---
grant select, insert, ... on db.table to user@host identified by 'password';
grant all privileges on *.* to user@'%' identified by 'password';

---查看---
show grants;

---回收---
revoke select, insert, ... on db.table from uuser@host;
revoke all privileges on mysql.* from user@host;
用户重写登录后生效

- 逻辑架构
  - 连接层

    本地 sock 通信，tcp/ip 通信
    连接处理、授权认证、安全方案
    线程池 ssl 安全连接
  - 服务层
    - Management Services & Utilities
    - SQL Interface
    - Parser
    - Optimizer
    - Cache & Buffer

      cache - 读缓存
      buffer - 写缓存
  - 引擎层

    负责真正的数据存储和提取，服务层通过 API 与存储引擎通信
  - 存储层

    存储数据与文件系统上，并与存储引擎层交互
- SQL 执行顺序

  from
  on
  join
  where
  group by
  having
  select
  distinct
  order by
  limit
- 存储引擎

---MyISAM vs. InnoDB---
1. 外键：N - Y
2. 事务：N - Y
3. 并发：表锁 - 行锁（高并发）
4. 缓存：缓存索引 - 缓存索引和数据（内存较大）
5. 优势：读性能 - 并发写，事务，资源
6. 自带：Y - Y
7. 使用：N - Y
8. 系统表所用：Y - N
9. 备份：N - Y

- show engines

  查看所有存储引擎
- show variables like '%storage_engine%'

  查看默认存储引擎

- **索引&优化**

2

MySQL 默认使用 B+Tree 索引
每趟查询最多只用到一个索引?index_merge?

- 索引

  ---B-Tree---
  平衡树，查找树

  ---B+Tree---
  InnoDB
  顺序访问指针提高了区间查询性能

  ---Hash 索引---
  InnoDB
  O(1)查找，无有序性，无法用于排序与分值
  只支持精准查找，不能范围查找

  ---full-text---
  MyISAM，InnoDB(5.6)，全文索引，倒排索引，关键词查找
  MATCH AGAINST

  ---R-Tree---
  MyISAM，空间数据索引

  ---聚簇索引 vs. 非聚簇索引---
  1. 聚簇索引的data 域存放真实数据，非聚簇索引的data 域存放主键
  2. 聚簇索引的查询直接得到数据，范围查找依次完成，非聚簇索引的查询得到主键后需要根据所得主键值回表再查询真实数据，范围查找需要大量 IO 操作
  3. 聚簇索引只 InnoDB 支持
  4. 聚簇索引每张表只能有一个，建议主键
  5. 为重复利用聚簇索引的有序性，应尽量使用有序id，不建议 uuid

  - 优缺点

    ---优势---
    1. 提高数据检索效率，降低数据库 IO 成本
    2. 索引列对数据排序，降低数据排序成本，降低 CPU 消耗，避免创建临时表

    ---劣势---
    1. 降低表更新速度，因为要保存索引的修改
    2. 索引也占用存储空间，实际是一张保存了主键和索引字段的表，字段指向实体表的记录
    3. 需要更多人工时间设计和优化索引
  - 操作
    - create [unique] index [indexname] on table(column)

      穿件索引
    - drop index [indexname] on table

      删除索引
    - show index from table

      查看索引
    - alter table table add primary key(column ...)

      主键
    - alter table table add index indexname(column ...)

      普通索引
    - alter table table add fulltext indexname(column ...)

全文索引

- 分类
  - 单值

    一个索引只包含一个列，一个表可以有多个

    ```
    ---在建表时创建索引---
    create table customer(
    id int(10) unsigned auto_increament,
    customer_no varchar(200),
    customer_name varchar(200),
    primary key(id),
    key(customer_no) -- 普通索引
    );

    ---单独创建索引---
    create index idx_customer_name on customer(customer_name);
    ```

  - 唯一

    索引的值必须唯一，但允许有空值

    ```
    ---在建表时创建索引---
    create table customer(
    id int(10) unsigned auto_increament,
    customer_no varchar(200),
    customer_name varchar(200),
    primary key(id),
    unique(customer_no) -- 唯一索引
    );

    ---单独创建索引---
    create unique index idx_customer_name on customer(customer_no);
    ```

  - 主键

    主键索引，innodb 中是聚簇索引

    ```
    ---在建表时创建索引---
    create table customer(
    id int(10) unsigned auto_increament,
    customer_no varchar(200),
    customer_name varchar(200),
    primary key(id) -- 主键
    );

    ---单独创建索引---
    alter table customer add primary key customer(customer_no);

    ---删除索引---
    alter table drop primary key;
    ```

  - 复合

    一个索引包含多个列

    ```
    ---在建表时创建索引---
    create table customer(
    id int(10) unsigned auto_increament,
    customer_no varchar(200),
    customer_name varchar(200),
    primary key(id),
    key(customer_no, customer_name) -- 复合索引
    );

    ---单独创建索引---
    create index idx_customer_name on customer(customer_no, customer_name);
    ```

- 场景
  - 需要

    1. 主键自动创建唯一索引
    2. 频繁作为查询条件的字段
    3. 与其他表关联的字段
    4. 组合索引比单键索引性价比高
    5. 排序字段
    6. 分组或统计的字段

  - 不需要

1. 表记录太少
2. 经常增删改的表字段
3. where 用不到的字段
4. 过滤性不好的字段，重复行多的字段，选择性（值的种类/值的行数）低的字段

- explain/desc

  模拟优化器的 SQL 执行计划

  SELECT SQL_NO_CACHE column ... - 不使用缓存查询
  \G - 纵向显示

  - id

    子查询的顺序，每个不同的 id 表示一趟查询，查询趟数越少越好
    id 相同，执行顺序至上而下
    id 不同，越大的越先被执行

  - select_type

    查询类型，用于区分普通查询，联合查询，子查询等

    1. SIMPLE - 简单的 select 查询,查询中不包含子查询或者 UNION
    2. PRIMARY - 查询中若包含任何复杂的子部分，最外层查询则被标记为 Primary
    3. DERIVED - 在 FROM 列表中包含的子查询被标记为 DERIVED(衍生)， MySQL 会递归执行这些子查询, 把结果放在临时表里。
    4. SUBQUERY - 在 SELECT 或 WHERE 列表中包含了子查询
    5. DEPEDENT SUBQUERY - 在 SELECT 或 WHERE 列表中包含了子查询,子查询基于外层
    6. UNCACHEABLE SUBQUERY - 无法使用缓存的子查询
    7. UNION - 若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION；若 UNION 包含在 FROM 子句的子查询中,外层 SELECT 将被标记为：DERIVED
    8. UNION RESULT - 从 UNION 表获取结果的 SELECT

  - table

    查询基于哪张表

  - type

    查询的访问类型

    优劣从好到坏如下，通常至少保证 range 级别，最好到 ref：
    system - 表只有一行记录（相当于系统表），const 类型的特例
    const - 通过一次索引就能找到，只匹配一行数据，用于 primary key 或者 unique key，主键置于 where 列表，MySQL 能把查询转换成一个常量
    eq_ref - 唯一性索引扫描，表中只存在一行记录匹配，常用于 primary key 或者 unique key 的扫描
    ref - 非唯一性索引扫描，返回匹配的所有行，可能找到多个符合条件的行，属于查找和扫描的混合
    fulltext - 倒排索引
    ref_or_null - 其他条件 or null 值一起过滤的情况
    index_merge - 需要多个索引组合使用，通常因为有 or
    unique_subquery - 类似于 index_subquery，只是子查询的全索引扫描变成了唯一索引扫描
    index_subquery - 利用索引关联子查询，相当于子查询中全索引扫描，不是全表扫描
    range - 检索给定范围的行，一般因为 where 中出现 between、<、>、in 等范围扫描索引，比全表扫描好，不需要全索引扫描
    index - 使用了索引，但没有利用索引过滤数据，相当于全索引扫描，可能是使用了覆盖索引、或者索引排序分组
    all - 全表扫描

  - possiable_keys

    子句可能使用的索引，一个或多个，子句设计的字段若存在索引，就会被列出，不一定会被真正使用

  - key

    实际使用的索引，NULL 表示没有使用索引

  - key_len

    索引被使用的字节数，越长说明索引使用的越充分，可以通过使用的索引包含的列计算出来

    ---计算方式--
    1. 先看索引上字段的类型+长度，如 int=4; varchar(20) =20; char(20) =20
    2. 如果是 varchar 或者 char 这种字符串字段，视字符集要乘不同的值，比如 utf-8 要乘3, GBK 要乘2，
    3. varchar 这种动态字符串要加 2 个字节（用于标记动态长度）
    4. 允许为空的字段要加 1 个字节（NULL标记位）

  - ref

    用于索引列上查找的值，可能是个常量，或者其他列的引用

  - rows

    根据表统计信息和索引使用情况，估算的执行查询需要检索的行数，越少越好

  - Extra

额外重要信息

using filesort - 使用外部的索引排序，不是按照表内的索引顺序读取，无法更高效的利用索引完成排序操作
using temporary - 使用临时表保存中间结果，如对结果排序时使用临时表，常见于 order by/ group by
using index - 使用了覆盖索引，避免了数据行的访问，效率较好，若无 using where，说明只用来读取数据而非索引查找
using where - 使用了索引过滤数据
using join buffer - 使用了连接缓存
impossible where - where 子句的值总是 false，获取不到任何元组
select tables optimized away - 在没有 group by 的情况下，对 MIN/MAX 或 MyISAM 的 count(*) 优化，不必等到执行阶段进行计算
distinct - 对 distinct 优化，找到第一个元组后，后续不再查找相同值

- 优化
  - 全值匹配

    1

    过滤字段在索引中都可以匹配到，key_len 可最大化，using where
  - 最左前缀法则

    2

    过滤字段从索引的最左前列开始且不跳过索引中的列
    否则索引无法充分利用，甚至失效
  - 不在索引列做计算

    3

    等号左边无计算，等号右边无类型转换
    计算、函数、手动或自动类型转换 将导致索引失效，全表扫描
  - 不对索引列做范围检索

    4

    范围检索将导致查询类型从 ref 转为 range，导致 key_len 变短，所有利用率变低
    建议将可能做范围查找的字段放在组合索引的最后

    存储引擎不能使用组合索引中处在范围检索字段右侧的字段
  - 覆盖索引

    5

    触发 using index, 可避免回表

    varchar 长度超过 380 时覆盖索引失效
  - 谨慎使用 != 或 <>

    6

    可能导致失效，全表扫描
  - 谨慎使用 is [not] null

    7

    可能用不到索引
    where 子句对 null 的写法特异

    取决于成本：
    1. 读取二级索引记录的成本
    2. 将二级索引记录执行回表操作，也就是到聚簇索引中找到完整的用户记录的操作所付出的成本。
  - like 仅限后缀模糊匹配

    8

    后缀模糊匹配会触发 range 级别的查询，前缀模糊匹配无法用到索引

    like 'xx%' 相当于=常量；like '%xx' 相当于范围扫描
  - union [all] 替代 or

    9

    where 中 or 条件会导致索引失效，union [all] 拆成两个 ref 级别的查询执行，再将结果做全表 union
- 建议
  - 单键索引，使用过滤性较好的字段建索引
  - 组合索引，过滤性越好越放在前列，范围扫描字段放在最后
  - 组合索引，尽量包含 where 子句中包含的字段
  - 通过统计分析调整 query 写法达到使用合适索引的目的

- **查询分析**

  3

  - 查询优化

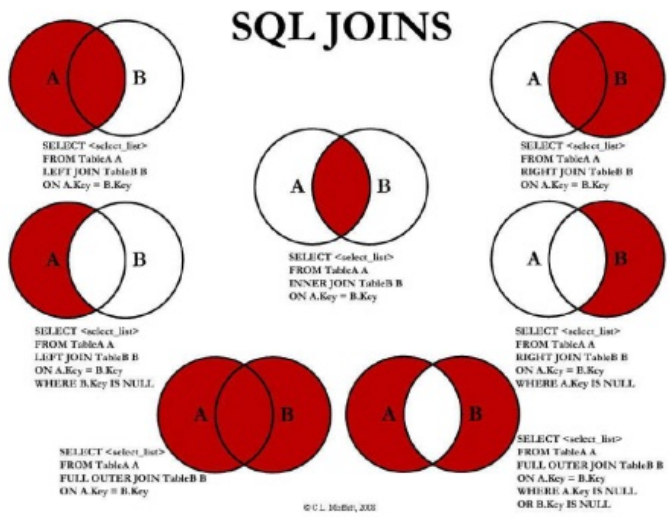
    1. 选择小表驱动大表，在被驱动的大表上建立索引
    2. 被驱动表尽量使用实体表，不要使用子查询（虚表），可能无法利用索引
    3. 尽量直接使用多表关联，不用子查询
    4. where 条件 on 判断作为过滤条件，在可以过滤足够多数据时应优先于 order by 和 group by 考虑，反之则相反

    - where

      过滤条件，无过滤不索引

- join on

  join 类型 nested loop
  on 判断作为过滤条件使用索引

  - left join

    索引建给被驱动表
    左表为驱动表，右表为被驱动表
  - inner join

    MySQL 自动选择小表为驱动表
    straight_join 会强制选择左表为驱动表
- in/exists

  A 表数据集比 B 表数据集小
  select * from A where exists select 1 from B

  A 表数据集比 B 表数据集大
  select * from A where id in select id from B

  在范围判断时，尽量不要使用 not in 和 not exists，使用 left join on xxx is null 代替
- order by

  MySQL 支持两种排序方式： index, filesort(低效，避免使用)

  order by 使用 index 排序的两个条件：
  1. order by 语句使用索引最左前列（顺序必须一直，排序方向必须一直）
  2. where 子句和 order by 子句条件列组合满足索引最左前列

  filesort 的两种算法：
  1. 双路排序：先取出 order by 列对其排序，再根据排好序的列回表读取数据
  2. 单路排序：取出所有所需列，在 buffer 对 order by 列排序，然后扫描排序后的列输出

  单路排序正常是单次 IO，一般会更高效，但 sort_buffer 不够时，可能造成多次IO，生成 tmp 文件用于合并，反而得不偿失

  ---filesort 调优---
  sort_buffer_size - 同时影响两种算法，1MB~8MB
  max_length_for_sort_data - 排序字段（query + order by）大小小于该值，且 order by 字段不是 text|blob 则使用单路排序，1MB~8MB
  减少 select 字段个数
- group by

  1. 实质是先排序后分组，索引使用原则与 order by 几乎一致
  2. where 先于 having 执行，能在 where 限定的条件尽量不要去 having 限定

- 慢 SQL 日志

  运行时间超过 long_query_time 的 SQL 会被记录到慢 SQL 日志

  slow_query_log - 启用慢 SQL 日志
  slow_query_log_file - 慢 SQL 日志文件地址
  long_query_time - 慢 SQL 记录阈值(s)
  log_output - 输出方式 FILE/TABLE(mysql.slow_log)

  ---慢 SQL 条数---
  show global status like '%Slow_queries%';
  - mysqldumpslow

日志分析工具

-s 何种方式排序
c 访问次数
l 锁定时间
r 返回记录
t 查询时间
al 平均锁定时间
ar 平均返回记录数
at 平均查询时间
-t 返回前面多少条的数据
-g 正则匹配模式，大小写不敏感的

- 全局查询日志

  general_log - 启停全局日志
  general_log_file - 全局日志文件地址
  log_output - 输出格式，FILE/TABLE(mysql.general_log)

  不可在生成环境启用该日志

- show processlist

  进程列表

  kill id; - 杀死指定进程

- show profiles

  查看 SQL 执行周期
  show profils;
  show profile cpu, block io for query query_id;

  ---查看---
  show variables like '%profiling%';

  ---开启---
  set profiling=1;

  ---参数---
  all - 显示所有开销
  block io - 显示块 io 开销
  context switches - 上下文切换相关开销
  cpu - 显示 cpu 相关开销
  ipc - 显示发送和接收相关信息
  memory - 显示内存相关开销信息
  page faults - 显示页面错误相关开销信息
  source - 显示和 source_function, source_file, source_line 相关的开销信息
  swaps - 显示交换次数相关开销的信息

  ---注意---
  convert HEAP to MyISAM - 查询结果太大，内存不够用，写入磁盘
  Creating tmp table - 创建临时表（拷贝数据到临时表 -> 用完删除临时表）
  Coping to tmp table on disk -把内存中临时表复制到磁盘
  locked

- **锁**

  4

  - 操作分类
    - 读锁（共享锁）

      多个读同时进行

    - 写锁（排它锁）

      写操作进行期间，阻塞其他写和读

  - 并发读/粒度
    - 表锁（偏读）

      偏向 MyISAM 引擎，MyISAM 读写锁调度是写优先，大量的写操作可能导致读操作永远阻塞，因此 MyISAM 不适合做写多场景的引擎
      无死锁，开销小，加锁快
      锁粒度大，发生锁冲突概率大，并发度低

      - lock table tablename read|write
      - unlock tables
      - show open tables
      - show status like 'table_locks%'

        状态变量

        table_locks_immediate - 产生表锁的次数，表示可以立即获取到锁的次数
        table_locks_waited - 产生表锁争用而发生等待的次数，不能立即获得锁+1，越高说明表锁争用情况越严重

- 行锁（偏写）

  偏向 InnoDB 引擎，对索引向加锁
  锁粒度小，发生锁冲突概率低，并发度高
  开销大，加锁慢，会出现死锁

  - Transation

    SQL 语句的逻辑处理单元

    - ACID

      Atomicity - 原子性，事务是一个原子操作，一个事务中的操作要么全部执行，要么全部回滚
      Consistent - 一致性，事务前后数据状态保持一致；所有相关数据保持完整性，内部数据结构（如索引）正确
      Isolation - 隔离性，事务执行过程中不受外部操作干扰，中间状态对外不可见
      Durable - 持久性，事务对数据的修改时永久性的，即时系统故障也能保持

    - 并发一致性问题

      更新丢失（Lost Update）- 多个更新事务同时进行，最后完成的更新覆盖所有前面的，通过隔离性避免这个问题
      胀读（Dirty Reads）- 一个事务读取了另一事务已修改未提交的数据（回滚）
      不可重复度（Non-Repeatable Reads）- 一个事务读取另一个事务已修改已提交的数据
      幻读（Phantom Reads）- 一个事务读取另一个事务已增删已提交的数据

    - 隔离级别

      未提交读（Read uncommitted），只能保证不读取物理上损坏的数据，存在胀读，不可重复度，幻读
      已提交度（Read committed），语句级，不可重复度，幻读
      可重复度（Repeatable read），事务级，幻读（默认）
      可序列化（Serializable），事务级

      ---查看---
      show variables like 'tx_isolation'

  - set autocommit=0/commit
  - select * from t where ... for update

    锁定指定行

  - show status like 'innodb_row_lock%'

    行锁相关状态变量

    innodb_row_lock_current_waits - 当前正在等待锁的数量
    innodb_row_lock_time - 从系统启动到现在锁定总时间长度
    innodb_row_lock_time_avg - 每次等待所花平均时间
    innodb_row_lock_time_max - 从系统启动到现在等待最长所花时间
    innodb_row_lock_waits - 系统启动到现在总共等待的次数

  - 升级表锁

    不走索引的查询，行锁会升级为表锁

  - 间隙锁

    利用索引对范围检索数据时，会锁定整个范围内所有所有键值，即使是在条件范围内但不存在的键值（间隙，GAP）

    会导致锁定期间无法对间隙插入数据，可能对性能造成较大危害

  - 优化

    1. 尽可能通过索引检索数据，避免无索引导致行锁升级表锁
    2. 合理设计索引，尽量减少锁定资源量
    3. 检索条件尽量少，避免间隙锁
    4. 事务尽量小，减小锁定资源量和时间长度
    5. 隔离级别尽量低

- 页锁

  锁粒度介于行锁和表锁之间，会出现死锁

- **主从复制**

  5

  读写分离
  slave 从 master 读取 binlog 进而同步数据(异步 + 串行化)

  1. （binlog线程）master 将修改记录到 binary log，这个过程叫 binary log event
  2. （I/O线程）slave 将 master 的 binary log event 拷贝到自己的 relay log（中继日志）
  3. （SQL线程）slave 重放 relay log 中的事件，将修改应用到自己的数据库

  复制基本原则
  1. 每个 slave 必须有一个 master
  2. 每个 slave 只能有一个唯一的服务器 ID
  3. 每个 master 可以有多个 slave

  复制最大的问题 - 延时

  - 性能提升

1. 主服务器负责写与及时性高要求的读，从服务器处理读，缓解锁争用
2. 从服务器可以使用 MyISAM，提升查询性能，节约系统开销
3. 增加荣誉，提高可用性

- 一主一从

注意事项：
1. 保证主从 MySQL 版本一致
2. 关闭防火墙或开放相应端口

- 主机配置

```
[mysqld]
server-id=1 # 【必须】主服务器唯一 ID
log-bin=/.../mysqlbin # 【必须】二进制日志路径
log-err=/.../mysqlerr # 错误日志路径
basedir=/.../ # 根目录
tmpdir=/.../ # 临时目录
datadir=/.../data 数据目录
read-only=0 # 可读可写
binlog-ignore-db=mysql # 不需要复制的数据库
binlog-do-db=db1 # 需要复制的数据库
binlog_format=STATEMENT # logbin 格式 默认是 STATEMENT(写语句)，还有 ROW(数据行) 和 MIXED(混合)
```

- 从机配置

```
server-id=2 # 【必须】从服务器 ID
log-bin=/.../mysqlbin # 二进制文件路径
relay-log=/.../relaylog # 中继日志文件路径
```

- 主机账号

```
grant replication slave on *.* to 'aurelius'@'slaveip' identified by 'xxxxxx';
flush privileges;
```

  - show master status

    查看主机状态（File/Position）

- 从机设置

```
change master to master_host='masterid', master_user='aurelius', master_password='xxxxxx', master_log_file='file',
master_log_pos='position';
```

主机 show master status; 获取 file 与 position 值

  - start slave

    启用从服务器复制功能

  - show slave status

    查看从服务器状态

    Slave_IO_Running:Yes
    Slave_SQL_Running:Yes

  - stop slave

    停用从服务器复制功能

  - reset master

    重置 master

- **切分**

6

数据分片：
垂直、水平、垂直+水平

- 策略

1. 哈希取模：hash(key) % N
2. 范围：定义 ID 或时间范围内数据的归宿
3. 映射表：使用单独的数据库来存储映射关系

- 问题

1. 事务问题：需要通过分布式事务解决
2. 连接：需要分解成多个单表查询，再在用户程序中进行连接
3. ID 唯一性：
1. 使用全局唯一 ID（GUID）
2. 指定每个分片的 ID 范围
3. 分布式 ID 生成算法（Twitter 的 Snowflake）

- **存储过程与函数**

7

1. 存储过程作用时批处理，函数用于实现特定功能
2. 存储过程有3种参数（in, out, inout），不需要有返回类型，函数只能有 in，明确返回值类型
3. 存储过程中可以使用非确定函数（rand），而用户自定函数中不允许使用非确定函数
4. 存储过程是单独调用的，函数可以在SQL 的 select、from 子句中被调用