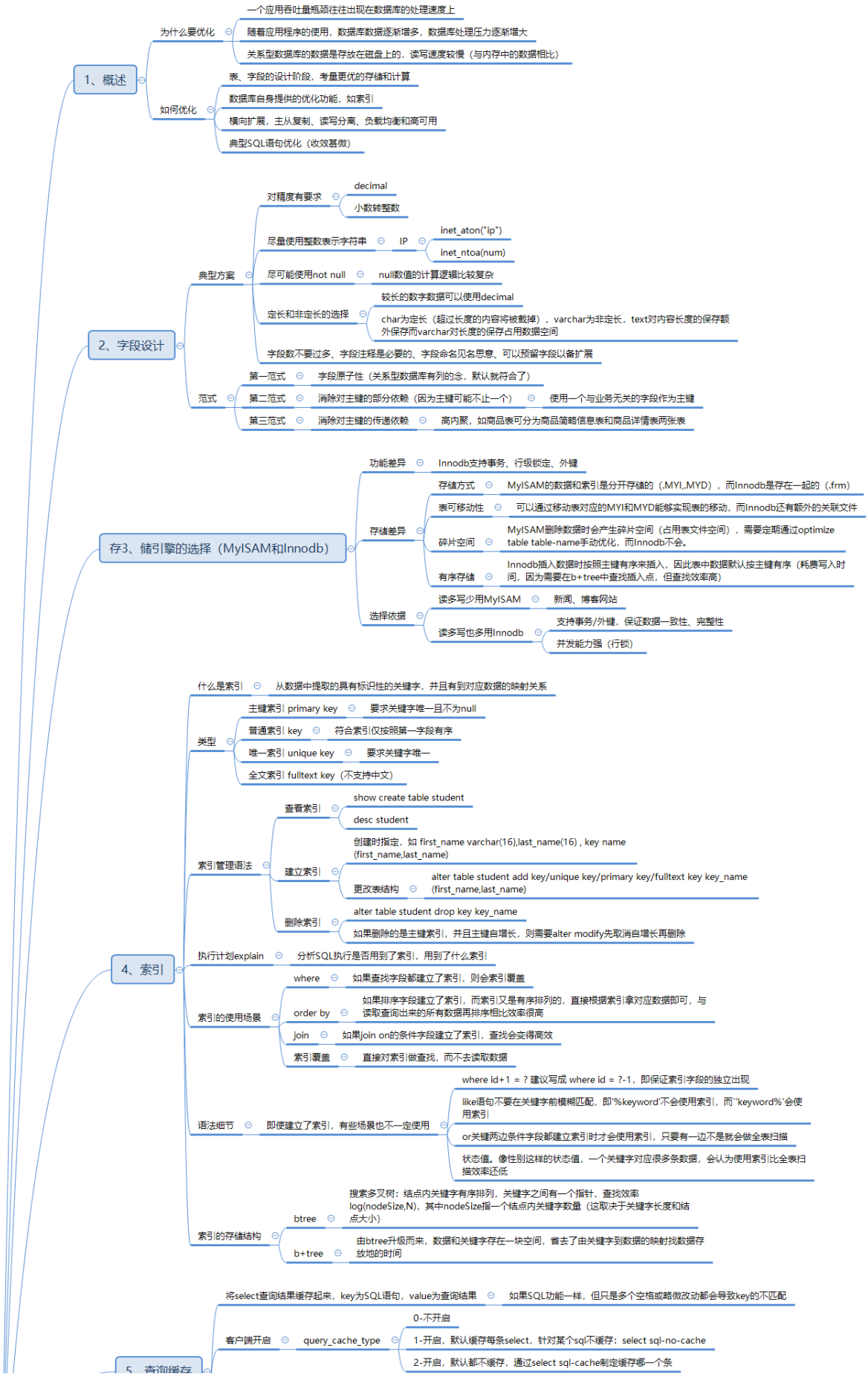
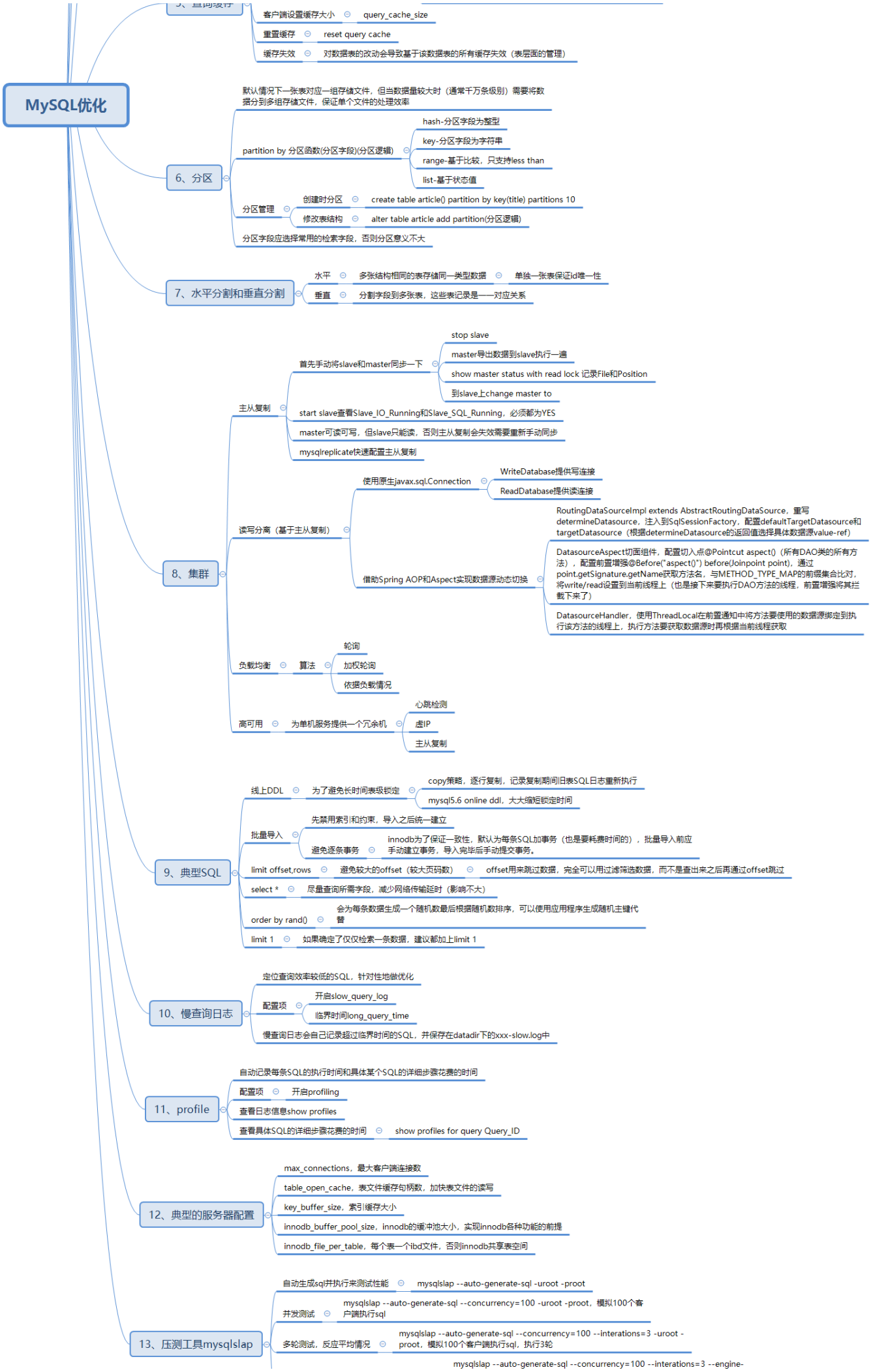
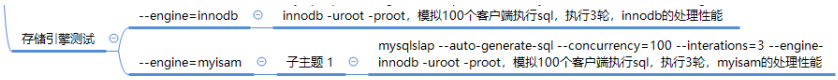
本文概要


1、概述
为什么要优化
一个应用吞吐量瓶颈往往出现在数据库的处理速度上
随着应用程序的使用，数据库数据逐渐增多，数据库处理压力逐渐增大
关系型数据库的数据是存放在磁盘上的，读写速度较慢（与内存中的数据相比）
如何优化
表、字段的设计阶段，考量更优的存储和计算
数据库自身提供的优化功能，如索引
横向扩展，主从复制、读写分离、负载均衡和高可用
典型SQL语句优化（收效甚微）
2、字段设计
典型方案
对精度有要求
decimal
小数转整数
尽量使用整数表示字符串
IP
inet_aton("ip")
inet_ntoa(num)
尽可能使用not null
null数值的计算逻辑比较复杂
定长和非定长的选择
较长的数字数据可以使用decimal
char为定长（超过长度的内容将被截掉），varchar为非定长，text对内容长度的保存额外保存而varchar对长度的保存占用数据空间
字段数不要过多、字段注释是必要的、字段命名见名思意、可以预留字段以备扩展
范式
第一范式
字段原子性（关系型数据库有列的念，默认就符合了）
第二范式
消除对主键的部分依赖（因为主键可能不止一个）
使用一个与业务无关的字段作为主键
第三范式
消除对主键的传递依赖
高内聚，如商品表可分为商品简略信息表和商品详情表两张表
存3、储引擎的选择（MyISAM和Innodb）
功能差异
Innodb支持事务、行级锁定、外键
存储差异
存储方式
MyISAM的数据和索引是分开存储的（.MYI,.MYD），而Innodb是存在一起的（.frm）
表可移动性
可以通过移动表对应的MYI和MYD能够实现表的移动，而Innodb还有额外的关联文件
碎片空间
MyISAM删除数据时会产生碎片空间（占用表文件空间），需要定期通过optimize table table-name手动优化，而Innodb不会。
有序存储
Innodb插入数据时按照主键有序来插入，因此表中数据默认按主键有序（耗费写入时间，因为需要在b+tree中查找插入点，但查找效率高）
选择依据
读多写少用MyISAM
新闻、博客网站
读多写也多用Innodb
支持事务/外键，保证数据一致性、完整性
并发能力强（行锁）
4、索引
什么是索引
从数据中提取的具有标识性的关键字，并且有到对应数据的映射关系
类型
主键索引 primary key
要求关键字唯一且不为null
普通索引 key
符合索引仅按照第一字段有序
唯一索引 unique key
要求关键字唯一
全文索引 fulltext key（不支持中文）
索引管理语法
查看索引
show create table student
desc student
建立索引
创建时指定，如 first_name varchar(16),last_name(16) , key name (first_name,last_name)
更改表结构
alter table student add key/unique key/primary key/fulltext key key_name (first_name,last_name)
删除索引
alter table student drop key key_name
如果删除的是主键索引，并且主键自增长，则需要alter modify先取消自增长再删除
执行计划explain
分析SQL执行是否用到了索引，用到了什么索引
索引的使用场景
where
如果查找字段都建立了索引，则会索引覆盖
order by
如果排序字段建立了索引，而索引又是有序排列的，直接根据索引拿对应数据即可，与读取查询出来的所有数据再排序相比效率很高
join
如果join on的条件字段建立了索引，查找会变得高效
索引覆盖
直接对索引做查找，而不去读取数据
语法细节
即使建立了索引，有些场景也不一定使用
where id+1 = ? 建议写成 where id = ?-1，即保证索引字段的独立出现
like语句不要在关键字前模糊匹配，即'%keyword'不会使用索引，而''keyword%'会使用索引
or关键两边条件字段都建立索引时才会使用索引，只要有一边不是就会做全表扫描
状态值。像性别这样的状态值，一个关键字对应很多条数据，会认为使用索引比全表扫描效率还低
索引的存储结构
btree
搜索多叉树：结点内关键字有序排列，关键字之间有一个指针，查找效率log(nodeSize,N)，其中nodeSize指一个结点内关键字数量（这取决于关键字长度和结点大小）
b+tree
由btree升级而来，数据和关键字存在一块空间，省去了由关键字到数据的映射找数据存放地的时间
5、查询缓存
将select查询结果缓存起来，key为SQL语句，value为查询结果
如果SQL功能一样，但只是多个空格或略微改动都会导致key的不匹配
客户端开启
query_cache_type
0-不开启
1-开启，默认缓存每条select，针对某个sql不缓存：select sql-no-cache
2-开启，默认都不缓存，通过select sql-cache制定缓存哪一个条


MySQL优化
客户端设置缓存大小
query_cache_size
重置缓存
reset query cache
缓存失效
对数据表的改动会导致基于该数据表的所有缓存失效（表层面的管理）
6、分区
默认情况下一张表对应一组存储文件，但当数据量较大时（通常千万条级别）需要将数据分到多组存储文件，保证单个文件的处理效率
partition by 分区函数(分区字段)(分区逻辑)
hash-分区字段为整型
key-分区字段为字符串
range-基于比较，只支持less than
list-基于状态值
分区管理
创建时分区
create table article() partition by key(title) partitions 10
修改表结构
alter table article add partition(分区逻辑)
分区字段应选择常用的检索字段，否则分区意义不大
7、水平分割和垂直分割
水平
多张结构相同的表存储同一类型数据
单独一张表保证id唯一性
垂直
分割字段到多张表，这些表记录是一一对应关系
8、集群
主从复制
首先手动将slave和master同步一下
stop slave
master导出数据到slave执行一遍
show master status with read lock 记录File和Position
到slave上change master to
start slave查看Slave_IO_Running和Slave_SQL_Running，必须都为YES
master可读可写，但slave只能读，否则主从复制会失效需要重新手动同步
mysqlreplicate快速配置主从复制
读写分离（基于主从复制）
使用原生javax.sql.Connection
WriteDatabase提供写连接
ReadDatabase提供读连接
借助Spring AOP和Aspect实现数据源动态切换
RoutingDataSourceImpl extends AbstractRoutingDataSource，重写determineDatasource，注入到SqlSessionFactory，配置defaultTargetDatasource和targetDatasource（根据determineDatasource的返回值选择具体数据源value-ref）
DatasourceAspect切面组件，配置切入点@Pointcut aspect()（所有DAO类的所有方法），配置前置增强@Before("aspect()") before(Joinpoint point)，通过point.getSignature.getName获取方法名，与METHOD_TYPE_MAP的前缀集合比对，将write/read设置到当前线程上（也是接下来要执行DAO方法的线程，前置增强将其拦截下来了）
DatasourceHandler，使用ThreadLocal在前置通知中将方法要使用的数据源绑定到执行该方法的线程上，执行方法要获取数据源时再根据当前线程获取
负载均衡
算法
轮询
加权轮询
依据负载情况
高可用
为单机服务提供一个冗余机
心跳检测
虚IP
主从复制
9、典型SQL
线上DDL
为了避免长时间表级锁定
copy策略，逐行复制，记录复制期间旧表SQL日志重新执行
mysql5.6 online ddl，大大缩短锁定时间
批量导入
先禁用索引和约束，导入之后统一建立
避免逐条事务
innodb为了保证一致性，默认为每条SQL加事务（也是要耗费时间的），批量导入前应手动建立事务，导入完毕后手动提交事务。
limit offset,rows
避免较大的offset（较大页码数）
offset用来跳过数据，完全可以用过滤筛选数据，而不是查出来之后再通过offset跳过
select *
尽量查询所需字段，减少网络传输延时（影响不大）
order by rand()
会为每条数据生成一个随机数最后根据随机数排序，可以使用应用程序生成随机主键代替
limit 1
如果确定了仅仅检索一条数据，建议都加上limit 1
10、慢查询日志
定位查询效率较低的SQL，针对性地做优化
配置项
开启slow_query_log
临界时间long_query_time
慢查询日志会自己记录超过临界时间的SQL，并保存在datadir下的xxx-slow.log中
11、profile
自动记录每条SQL的执行时间和具体某个SQL的详细步骤花费的时间
配置项
开启profiling
查看日志信息show profiles
查看具体SQL的详细步骤花费的时间
show profiles for query Query_ID
12、典型的服务器配置
max_connections，最大客户端连接数
table_open_cache，表文件缓存句柄数，加快表文件的读写
key_buffer_size，索引缓存大小
innodb_buffer_pool_size，innodb的缓冲池大小，实现innodb各种功能的前提
innodb_file_per_table，每个表一个ibd文件，否则innodb共享表空间
13、压测工具mysqlslap
自动生成sql并执行来测试性能
mysqlslap --auto-generate-sql -uroot -proot
并发测试
mysqlslap --auto-generate-sql --concurrency=100 -uroot -proot，模拟100个客户端执行sql
多轮测试，反应平均情况
mysqlslap --auto-generate-sql --concurrency=100 --interations=3 -uroot -proot，模拟100个客户端执行sql，执行3轮
mysqlslap --auto-generate-sql --concurrency=100 --interations=3 --engine-



概述

为什么要优化

- 系统的吞吐量瓶颈往往出现在数据库的访问速度上
- 随着应用程序的运行，数据库的中的数据会越来越多，处理时间会相应变慢
- 数据是存放在磁盘上的，读写速度无法和内存相比

如何优化

- 设计数据库时：数据库表、字段的设计，存储引擎
- 利用好MySQL自身提供的功能，如索引等
- 横向扩展：MySQL集群、负载均衡、读写分离
- SQL语句的优化（收效甚微）

字段设计

> 字段类型的选择，设计规范，范式，常见设计案例

原则：尽量使用整型表示字符串

存储IP

`INET_ATON(str)`，address to number

`INET_NTOA(number)`，number to address

MySQL内部的枚举类型（单选）和集合（多选）类型

但是因为维护成本较高因此不常使用，使用**关联表**的方式来替代`enum`

原则：定长和非定长数据类型的选择

> decimal不会损失精度，存储空间会随数据的增大而增大。double占用固定空间，较大数的存储会损失精度。非定长的还有varchar、text

金额

> 对数据的精度要求较高，小数的运算和存储存在精度问题（不能将所有小数转换成二进制）

定点数decimal

`price decimal(8,2)` 有2位小数的定点数，定点数支持很大的数（甚至是超过 `int,bigint` 存储范围的数）

小单位大数额避免出现小数

元->分

字符串存储

定长 `char` ，非定长 `varchar、text` （上限65535，其中 `varchar` 还会消耗1-3字节记录长度，而 `text` 使用额外空间记录长度）

原则：尽可能选择小的数据类型和指定短的长度

原则：尽可能使用 not null

非 `null` 字段的处理要比 `null` 字段的处理高效些！且不需要判断是否为 `null` 。

`null` 在MySQL中，不好处理，存储需要额外空间，运算也需要特殊的运算符。如 `select null = null` 和 `select null <> null` （ `<>` 为不等号）有着同样的结果，只能通过 `is null` 和 `is not null` 来判断字段是否为 `null` 。

如何存储？MySQL中每条记录都需要额外的存储空间，表示每个字段是否为 `null` 。因此通常使用特殊的数据进行占位，比如 `int not null default 0` 、 `string not null default ‘’`

原则：字段注释要完整，见名知意

原则：单表字段不宜过多

二三十个就极限了

原则：可以预留字段

> 在使用以上原则之前首先要满足业务需求

关联表的设计

> 外键 `foreign key` 只能实现一对一或一对多的映射

一对多

使用外键

多对多

单独新建一张表将多对多拆分成两个一对多

一对一

如商品的基本信息（`item`）和商品的详细信息（`item_intro`），通常使用相同的主键或者增加一个外键字段（`item_id`）

范式 Normal Format

> 数据表的设计规范，一套越来越严格的规范体系（如果需要满足N范式，首先要满足N-1范式）。N

第一范式1NF：字段原子性

字段原子性，字段不可再分割。

> 关系型数据库，默认满足第一范式

注意比较容易出错的一点，在一对多的设计中使用逗号分隔多个外键，这种方法虽然存储方便，但不利于维护和索引（比如查找带标签`java`的文章）

第二范式：消除对主键的部分依赖

> 即在表中加上一个与业务逻辑无关的字段作为主键

主键：可以唯一标识记录的字段或者字段集合。

course_name	course_class	weekday（周几）	course_teacher
MySQL	教育大楼1525	周一	张三
Java	教育大楼1521	周三	李四
MySQL	教育大楼1521	周五	张三

依赖：A字段可以确定B字段，则B字段依赖A字段。比如知道了下一节课是数学课，就能确定任课老师是谁。于是**周几**和**下一节课**和就能构成复合主键，能够确定去哪个教室上课，任课老师是谁等。但我们常常增加一个`id`作为主键，而消除对主键的部分依赖。

对主键的部分依赖：某个字段依赖复合主键中的一部分。

解决方案：新增一个独立字段作为主键。

第三范式：消除对主键的传递依赖

传递依赖：B字段依赖于A，C字段又依赖于B。比如上例中，任课老师是谁取决于是什么课，是什么课又取决于主键`id`。因此需要将此表拆分为两张表日程表和课程表（独立数据独立建表）：

id	weekday	course_class	course_id
1001	周一	教育大楼1521	3546

course_id	course_name	course_teacher
3546	Java	张三

这样就减少了数据的冗余（即使周一至周日每天都有Java课，也只是`course_id:3546`出现了7次）

存储引擎选择

> 早期问题：如何选择MyISAM和Innodb?
>
> 现在不存在这个问题了，Innodb不断完善，从各个方面赶超MyISAM，也是MySQL默认使用的。

存储引擎Storage engine：MySQL中的数据、索引以及其他对象是如何存储的，是一套文件系统的实现。

功能差异

`show engines`

Engine	Support	Comment
InnoDB	DEFAULT	**Supports transactions, row-level locking, and foreign keys**
MyISAM	YES	**MyISAM storage engine**

存储差异

	MyISAM	Innodb
文件格式	数据和索引是分别存储的，数据`.MYD`，索引`.MYI`	数据和索引是集中存储的，`.ibd`
文件能否移动	能，一张表就对应`.frm`、`MYD`、`MYI`3个文件	否，因为关联的还有`data`下的其它文件
记录存储顺序	按记录插入顺序保存	按主键大小有序插入

	MyISAM	Innodb
空间碎片（删除记录并`flush table 表名`之后，表文件大小不变）	产生。定时整理：使用命令`optimize table 表名`实现	不产生
事务	不支持	支持
外键	不支持	支持
锁支持（锁是避免资源争用的一个机制，MySQL锁对用户几乎是透明的）	表级锁定	行级锁定、表级锁定，锁定力度小并发能力高

> 锁扩展
>
> 表级锁（`table-level lock`）：`lock tables <table_name1>,<table_name2>... read/write`，`unlock tables <table_name1>,<table_name2>...`。其中`read`是共享锁，一旦锁定任何客户端都不可读；`write`是独占/写锁，只有加锁的客户端可读可写，其他客户端既不可读也不可写。锁定的是一张表或几张表。
>
> 行级锁（`row-level lock`）：锁定的是一行或几行记录。共享锁：`select * from <table_name> where <条件> LOCK IN SHARE MODE;`，对查询的记录增加共享锁；`select * from <table_name> where <条件> FOR UPDATE;`，对查询的记录增加排他锁。这里**值得注意**的是：`innodb`的行锁，其实是一个子范围锁，依据条件锁定部分范围，而不是就映射到具体的行上，因此还有一个学名：间隙锁。比如`select * from stu where id < 20 LOCK IN SHARE MODE`会锁定`id`在`20`左右以下的范围，你可能无法插入`id`为`18`或`22`的一条新纪录。

选择依据

如果没有特别的需求，使用默认的`Innodb`即可。

MyISAM：以读写插入为主的应用程序，比如博客系统、新闻门户网站。

Innodb：更新（删除）操作频率也高，或者要保证数据的完整性；并发量高，支持事务和外键保证数据完整性。比如OA自动化办公系统。

索引

> 关键字与数据的映射关系称为索引（包含关键字和对应的记录在磁盘中的地址）。关键字是从数据当中提取的用于标识、检索数据的特定内容。

索引检索为什么快?

- 关键字相对于数据本身，数据量小
- 关键字是有序的，二分查找可快速确定位置

图书馆为每本书都加了索引号（类别-楼层-书架）、字典为词语解释按字母顺序编写目录等都用到了索引。

MySQL中索引类型

> **普通索引**（`key`），**唯一索引**（`unique key`），**主键索引**（`primary key`），**全文索引**（`fulltext key`）

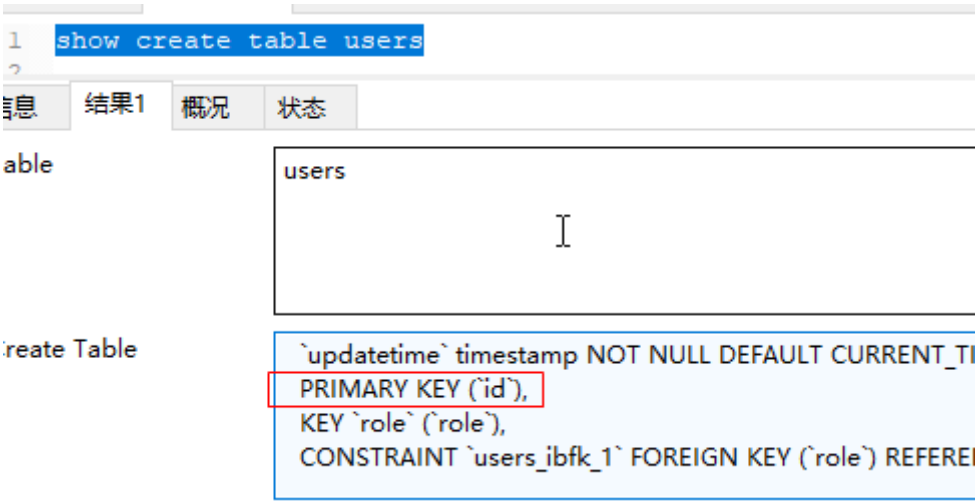
三种索引的索引方式是一样的，只不过对索引的关键字有不同的限制：

- 普通索引：对关键字没有限制
- 唯一索引：要求记录提供的关键字不能重复
- 主键索引：要求关键字唯一且不为null

索引管理语法

查看索引

`show create table 表名`：



`desc 表名`

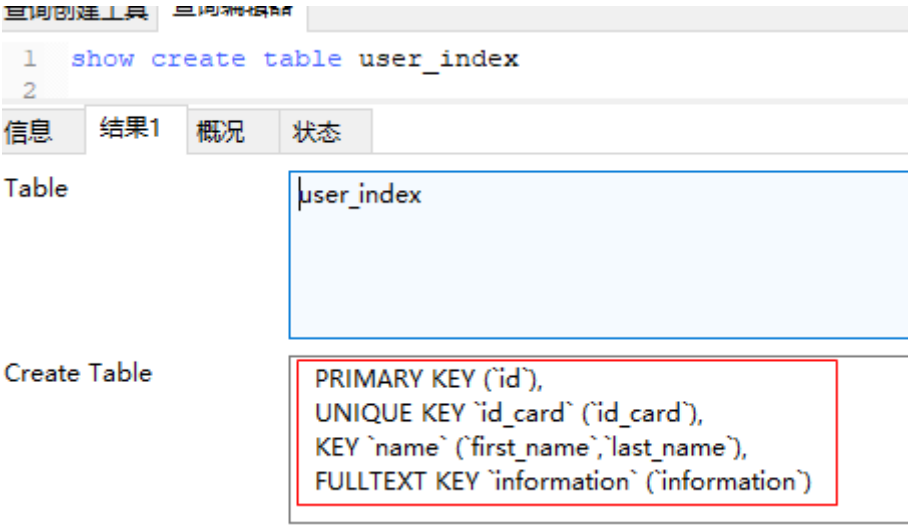
创建索引

创建表之后建立索引

```
create TABLE user_index(
    id int auto_increment primary key,
    first_name varchar(16),
    last_name VARCHAR(16),
    id_card VARCHAR(18),
    information text
);

-- 更改表结构
alter table user_index
-- 创建一个first_name和last_name的复合索引，并命名为name
add key name (first_name,last_name),
-- 创建一个id_card的唯一索引，默认以字段名作为索引名
add UNIQUE KEY (id_card),
-- 鸡肋，全文索引不支持中文
add FULLTEXT KEY (information);
```

`show create table user_index`：



创建表时指定索引

```
CREATE TABLE user_index2 (
    id INT auto_increment PRIMARY KEY,
    first_name VARCHAR (16),
    last_name VARCHAR (16),
    id_card VARCHAR (18),
    information text,
    KEY name (first_name, last_name),
    FULLTEXT KEY (information),
 UNIQUE KEY (id_card)
);
```

删除索引

根据索引名删除普通索引、唯一索引、全文索引：`alter table 表名 drop KEY 索引名`

```
alter table user_index drop KEY name;
alter table user_index drop KEY id_card;
alter table user_index drop KEY information;
```

删除主键索引：`alter table 表名 drop primary key`（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：

```
1  alter table user_index drop primary key
```

信息 | 概况 | 状态

[SQL]alter table user_index drop primary key

[Err] 1075 - Incorrect table definition; there can be only one auto column and it must be defined as a key

需要取消自增长再行删除：

```
alter table user_index
-- 重新定义字段
MODIFY id int,
drop PRIMARY KEY
```

但通常不会删除主键，因为设计主键一定与业务逻辑无关。

执行计划explain

```
CREATE TABLE innodb1 (
    id INT auto_increment PRIMARY KEY,
    first_name VARCHAR (16),
    last_name VARCHAR (16),
    id_card VARCHAR (18),
    information text,
    KEY name (first_name, last_name),
    FULLTEXT KEY (information),
 UNIQUE KEY (id_card)
);
insert into innodb1 (first_name,last_name,id_card,information) values ('张','三','1001','华山派');
```

我们可以通过`explain selelct`来分析SQL语句执行前的执行计划：

```
EXPLAIN select * from innodb1 where id<20
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	innodb1	(Null)	range	PRIMARY	PRIMARY	4	(Null)

由上图可看出此SQL语句是按照主键索引来检索的。

执行计划是：当执行SQL语句时，首先会分析、优化，形成执行计划，在按照执行计划执行。

索引使用场景（重点）

where

上图中，根据`id`查询记录，因为`id`字段仅建立了主键索引，因此此SQL执行可选的索引只有主键索引，如果有多个，最终会选一个较优的作为检索的依据。

```
-- 增加一个没有建立索引的字段
alter table innodb1 add sex char(1);
-- 按sex检索时可选的索引为null
EXPLAIN SELECT * from innodb1 where sex='男';
```

```
EXPLAIN SELECT * from innodb1 where sex='男'
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key
1	SIMPLE	innodb1	(Null)	ALL	(Null)	(Null)	(Nu

> 可以尝试在一个字段未建立索引时，根据该字段查询的效率，然后对该字段建立索引（`alter table 表名 add index(字段名)`），同样的SQL执行的效率，你会发现查询效率会有明显的提升（数据量越大越明显）。

order by

当我们使用`order by`将查询结果按照某个字段排序时，如果该字段没有建立索引，那么执行计划会将查询出的所有数据使用外部排序（将数据从硬盘分批读取到内存使用内部排序，最后合并排序结果），这个操作是很影

响性能的，因为需要将查询涉及到的所有数据从磁盘中读到内存（如果单条数据过大或者数据量过多都会降低效率），更无论读到内存之后的排序了。

但是如果我们对该字段建立索引`alter table 表名 add index(字段名)`，那么由于索引本身是有序的，因此直接按照索引的顺序和映射关系逐条取出数据即可。而且如果分页的，那么只用**取出索引表某个范围内的索引对应的数据**，而不用像上述那**取出所有数据**进行排序再返回某个范围内的数据。（从磁盘取数据是最影响性能的）

join

> 对`join`语句匹配关系（`on`）涉及的字段建立索引能够提高效率

索引覆盖

如果要查询的字段都建立过索引，那么引擎会直接在索引表中查询而不会访问原始数据（否则只要有一个字段没有建立索引就会做全表扫描），这叫索引覆盖。因此我们需要尽可能的在`select`后只写必要的查询字段，以增加索引覆盖的几率。

这里值得注意的是不要想着为每个字段建立索引，因为优先使用索引的优势就在于其体积小。

语法细节（要点）

> 在满足索引使用的场景下（`where/order by/join on`或索引覆盖），索引也不一定被使用

字段要独立出现

比如下面两条SQL语句在语义上相同，但是第一条会使用主键索引而第二条不会。

```
select * from user where id = 20-1;
select * from user where id+1 = 20;
```

`like`查询，不能以通配符开头

比如搜索标题包含`mysql`的文章：

```
select * from article where title like '%mysql%';
```

这种SQL的执行计划用不了索引（`like`语句匹配表达式以通配符开头），因此只能做全表扫描，效率极低，在实际工程中几乎不被采用。而一般会使用第三方提供的支持中文的全文索引来做。

但是 **关键字查询** 热搜提醒功能还是可以做的，比如键入`mysql`之后提醒`mysql 教程`、`mysql 下载`、`mysql 安装步骤`等。用到的语句是：

```
select * from article where title like 'mysql%';
```

这种 `like` 是可以利用索引的（当然前提是 `title` 字段建立过索引）。

复合索引只对第一个字段有效

建立复合索引：

```
alter table person add index(first_name,last_name);
```

其原理就是将索引先按照从 `first_name` 中提取的关键字排序，如果无法确定先后再按照从 `last_name` 提取的关键字排序，也就是说该索引表只是按照记录的 `first_name` 字段值有序。

因此 `select * from person where first_name = ?` 是可以利用索引的，而 `select * from person where last_name = ?` 无法利用索引。

> 那么该复合索引的应用场景是什么？组合查询

比如对于 `select * person from first_name = ? and last_name = ?`，复合索引就比对 `first_name` 和 `last_name` 单独建立索引要高效些。很好理解，复合索引首先二分查找与 `first_name = ?` 匹配的记录，再在这些记录中二分查找与 `last_name` 匹配的记录，只涉及到一张索引表。而分别单独建立索引则是在 `first_name` 索引表中二分找出与 `first_name = ?` 匹配的记录，再在 `last_name` 索引表中二分找出与 `last_name = ?` 的记录，两者取交集。

or，两边条件都有索引可用

> 一但有一边无索引可用就会导致整个SQL语句的全表扫描

状态值，不容易使用到索引

如性别、支付状态等状态值字段往往只有极少的几种取值可能，这种字段即使建立索引，也往往利用不上。这是因为，一个状态值可能匹配大量的记录，这种情况MySQL会认为利用索引比全表扫描的效率低，从而弃用索引。索引是随机访问磁盘，而全表扫描是顺序访问磁盘，这就好比有一栋20层楼的写字楼，楼底下的索引牌上写着某个公司对应不相邻的几层楼，你去公司找人，与其按照索引牌的提示去其中一层楼没找到再下来看索引牌再上楼，不如从1楼挨个往上找到顶楼。

如何创建索引

- 建立基础索引：在 `where、order by、join` 字段上建立索引。
- 优化，组合索引：基于业务逻辑
  - 如果条件经常性出现在一起，那么可以考虑将多字段索引升级为复合索引
  - 如果通过增加个别字段的索引，就可以出现索引覆盖，那么可以考虑为该字段建立索引
  - 查询时，不常用到的索引，应该删除掉

前缀索引

语法：`index(field(10))`，使用字段值的前10个字符建立索引，默认是使用字段的全部内容建立索引。

前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。

实操的难度：在于前缀截取的长度。

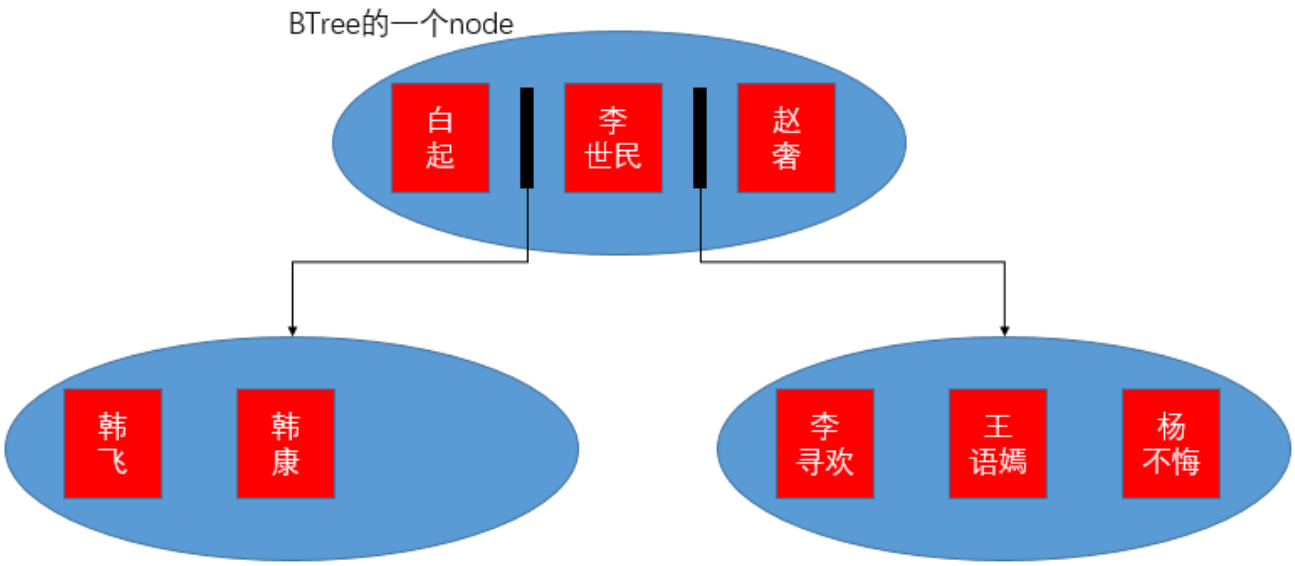
我们可以利用`select count(*)/count(distinct left(password,prefixLen));`，通过从调整`prefixLen`的值（从1自增）查看不同前缀长度的一个平均匹配度，接近1时就可以了（表示一个密码的前`prefixLen`个字符几乎能确定唯一一条记录）

索引的存储结构

BTree

btree（多路平衡查找树）是一种广泛应用于磁盘上实现索引功能的一种数据结构，也是大多数数据库索引表的实现。

以`add index(first_name,last_name)`为例：



BTree的一个node可以存储多个关键字，node的大小取决于计算机的文件系统，因此我们可以通过减小索引字段的长度使结点存储更多的关键字。如果node中的关键字已满，那么可以通过每个关键字之间的子节点指针来拓展索引表，但是不能破坏结构的有序性，比如按照`first_name`第一有序、`last_name`第二有序的规则，新添加的`韩香`就可以插到`韩康`之后。`白起 < 韩飞 < 韩康 < 李世民 < 赵奢 < 李寻欢 < 王语嫣 < 杨不悔`。这与二叉搜索树的思想是一样的，只不过二叉搜索树的查找效率是`log(2,N)`（以2为底N的对数），而BTree的查找效率是`log(x,N)`（其中x为node的关键字数量，可以达到1000以上）。

从`log(1000+,N)`可以看出，少量的磁盘读取即可做到大量数据的遍历，这也是btree的设计目的。

B+Tree聚簇结构

聚簇结构（也是在BTree上升级改造的）中，关键字和记录是存放在一起的。

在MySQL中，仅仅只有`Innodb`的主键索引为聚簇结构，其它的索引包括`Innodb`的非主键索引都是典型的BTree结构。

哈希索引

在索引被载入内存时，使用哈希结构来存储。

查询缓存

> 缓存`select`语句的查询结果

在配置文件中开启缓存

windows上是`my.ini`，linux上是`my.cnf`

在`[mysqld]`段中配置`query_cache_type`：

- 0：不开启
- 1：开启，默认缓存所有，需要在SQL语句中增加`select sql-no-cache`提示来放弃缓存
- 2：开启，默认都不缓存，需要在SQL语句中增加`select sql-cache`来主动缓存（常用）

更改配置后需要重启以使配置生效，重启后可通过`show variables like ‘query_cache_type’;`来查看：

```
show variables like 'query_cache_type';
query_cache_type    DEMAND
```

在客户端设置缓存大小

通过配置项`query_cache_size`来设置：

```
show variables like 'query_cache_size';
query_cache_size 0

set global query_cache_size=64*1024*1024;
show variables like 'query_cache_size';
query_cache_size 67108864
```

将查询结果缓存

`select sql_cache * from user;`

重置缓存

`reset query cache;`

缓存失效问题（大问题）

当数据表改动时，基于该数据表的任何缓存都会被删除。（表层面的管理，不是记录层面的管理，因此失效率较高）

注意事项

1. 应用程序，不应该关心 `query cache` 的使用情况。可以尝试使用，但不能由 `query cache` 决定业务逻辑，因为 `query cache` 由DBA来管理。
2. 缓存是以SQL语句为key存储的，因此即使SQL语句功能相同，但如果多了一个空格或者大小写有差异都会导致匹配不到缓存。

分区

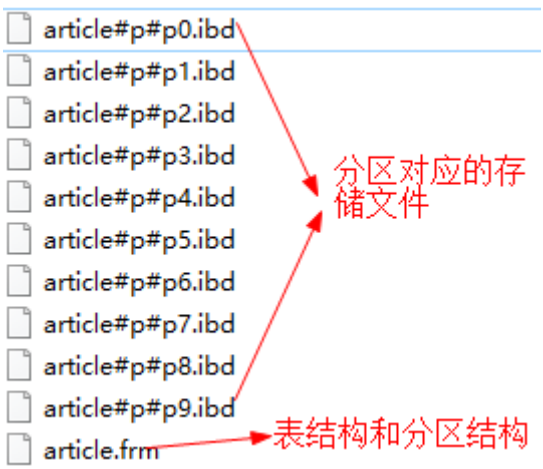
一般情况下我们创建的表对应一组存储文件，使用 `MyISAM` 存储引擎时是一个 `.MYI` 和 `.MYD` 文件，使用 `Innodb` 存储引擎时是一个 `.ibd` 和 `.frm` （表结构）文件。

当数据量较大时（一般千万条记录级别以上），MySQL的性能就会开始下降，这时我们就需要将数据分散到多组存储文件，保证其单个文件的执行效率。

最常见的分区方案是按 `id` 分区，如下将 `id` 的哈希值对10取模将数据均匀分散到10个 `.ibd` 存储文件中：

```
create table article(
    id int auto_increment PRIMARY KEY,
    title varchar(64),
    content text
)PARTITION by HASH(id) PARTITIONS 10
```

查看 `data` 目录：



> 服务端的表分区对于客户端是透明的，客户端还是照常插入数据，但服务端会按照分区算法分散存储数据。

MySQL提供的分区算法

> 分区依据的字段必须是主键的一部分，分区是为了快速定位数据，因此该字段的搜索频次较高应作为强检索字段，否则依照该字段分区毫无意义

hash(field)

相同的输入得到相同的输出。输出的结果跟输入是否具有规律无关。仅适用于整型字段

key(field)

和 `hash(field)` 的性质一样，只不过 `key` 是处理字符串的，比 `hash()` 多了一步从字符串中计算出一个整型在做取模操作。

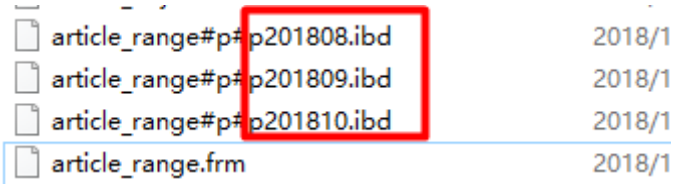
```
create table article_key(
    id int auto_increment,
    title varchar(64),
    content text,
 PRIMARY KEY (id,title) -- 要求分区依据字段必须是主键的一部分
)PARTITION by KEY(title) PARTITIONS 10
```

range算法

是一种条件分区算法，按照数据大小范围分区（将数据使用某种条件，分散到不同的分区中）。

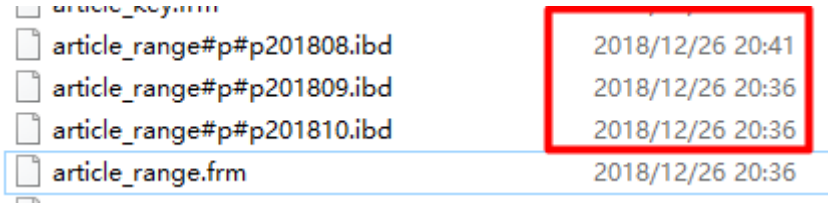
如下，按文章的发布时间将数据按照2018年8月、9月、10月分区存放：

```
create table article_range(
    id int auto_increment,
    title varchar(64),
    content text,
    created_time int, -- 发布时间到1970-1-1的毫秒数
 PRIMARY KEY (id,created_time) -- 要求分区依据字段必须是主键的一部分
)charset=utf8
PARTITION BY RANGE(created_time)(
 PARTITION p201808 VALUES less than (1535731199), -- select UNIX_TIMESTAMP('2018-8-31
23:59:59')
 PARTITION p201809 VALUES less than (1538323199), -- 2018-9-30 23:59:59
 PARTITION p201810 VALUES less than (1541001599) -- 2018-10-31 23:59:59
);
```



注意：条件运算符只能使用less than，这以为着较小的范围要放在前面，比如上述 `p201808,p201819,p201810` 分区的定义顺序依照 `created_time` 数值范围从小到大，不能颠倒。

```
insert into article_range values(null,'MySQL优化','内容示例',1535731180);
flush tables; -- 使操作立即刷新到磁盘文件
```



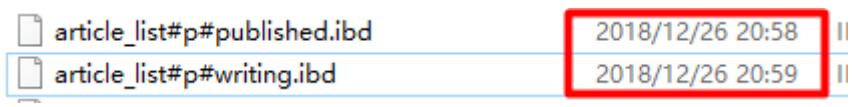
由于插入的文章的发布时间 `1535731180` 小于 `1535731199` （`2018-8-31 23:59:59`），因此被存储到 `p201808` 分区中，这种算法的存储到哪个分区取决于数据状况。

list算法

也是一种条件分区，按照列表值分区（`in (值列表)`）。

```
create table article_list(
    id int auto_increment,
    title varchar(64),
    content text,
    status TINYINT(1), -- 文章状态：0-草稿，1-完成但未发布，2-已发布
 PRIMARY KEY (id,status) -- 要求分区依据字段必须是主键的一部分
)charset=utf8
PARTITION BY list(status)(
 PARTITION writing values in(0,1), -- 未发布的放在一个分区
 PARTITION published values in (2) -- 已发布的放在一个分区
);
```

```
insert into article_list values(null,'mysql优化','内容示例',0);
flush tables;
```



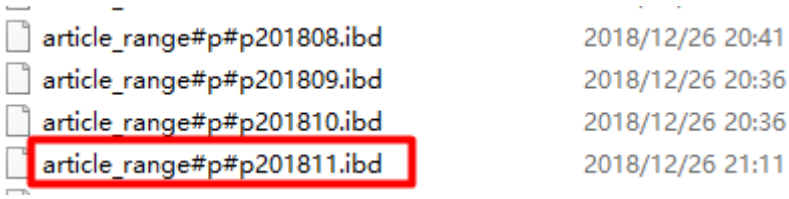
分区管理语法

range/list

增加分区

前文中我们尝试使用 `range` 对文章按照月份归档，随着时间的增加，我们需要增加一个月份：

```
alter table article_range add partition(
 partition p201811 values less than (1543593599) -- select UNIX_TIMESTAMP('2018-11-30
23:59:59')
 -- more
);
```



删除分区

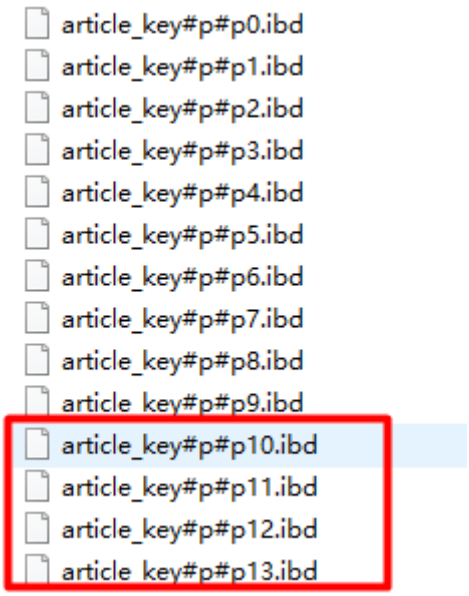
```
alter table article_range drop PARTITION p201808
```

注意：删除分区后，分区中原有的数据也会随之删除！

key/hash

新增分区

```
alter table article_key add partition partitions 4
```



销毁分区

```
alter table article_key coalesce partition 6
```

`key/hash` 分区的管理不会删除数据，但是每一次调整（新增或销毁分区）都会将所有的数据重写分配到新的分区上。效率极低，最好在设计阶段就考虑好分区策略。

分区的使用

当数据表中的数据量很大时，分区带来的效率提升才会显现出来。

只有检索字段为分区字段时，分区带来的效率提升才会比较明显。因此，分区字段的选择很重要，并且业务逻辑要尽可能地根据分区字段做相应调整（尽量使用分区字段作为查询条件）。

水平分割和垂直分割

> 水平分割：通过建立结构相同的几张表分别存储数据
>
> 垂直分割：将经常一起使用的字段放在一个单独的表中，分割后的表记录之间是一一对应关系。

分表原因

- 为数据库减压
- 分区算法局限
- 数据库支持不完善（`5.1` 之后 `mysql` 才支持分区操作）

id重复的解决方案

- 借用第三方应用如 `memcache、redis` 的 `id` 自增器
- 单独建一张只包含 `id` 一个字段的表，每次自增该字段作为数据记录的 `id`

集群

> 横向扩展：从根本上（单机的硬件处理能力有限）提升数据库性能 。由此而生的相关技术：读写分离、负载均衡

安装和配置主从复制

环境

- `Red Hat Enterprise Linux Server release 7.0 (Maipo)` （虚拟机）
- `mysql5.7` （下载地址）

安装和配置

解压到对外提供的服务的目录（我自己专门创建了一个 `/export/server` 来存放）

```
tar xzvf mysql-5.7.23-linux-glibc2.12-x86_64.tar.gz -C /export/server
cd /export/server
mv mysql-5.7.23-linux-glibc2.12-x86_64 mysql
```

添加 `mysql` 目录的所属组和所属者：

```
groupadd mysql
useradd -r -g mysql mysql
cd /export/server
chown -R mysql:mysql mysql/
chmod -R 755 mysql/
```

创建 `mysql` 数据存放目录（其中 `/export/data` 是我创建专门用来为各种服务存放数据的目录）

```
mkdir /export/data/mysql
```

初始化 `mysql` 服务

```
cd /export/server/mysql
./bin/mysqld --basedir=/export/server/mysql --datadir=/export/data/mysql --user=mysql --pid-
file=/export/data/mysql/mysql.pid --initialize
```

> 如果成功会显示 `mysql` 的 `root` 账户的初始密码，记下来以备后续登录。如果报错缺少依赖，则使用 `yum instally` 依次安装即可

配置 `my.cnf`

```
vim /etc/my.cnf

[mysqld]
basedir=/export/server/mysql
datadir=/export/data/mysql
socket=/tmp/mysql.sock
user=mysql
server-id=10 # 服务id，在集群时必须唯一，建议设置为IP的第四段
port=3306
# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
# Settings user and group are ignored when systemd is used.
# If you need to run mysqld under a different user or group,
# customize your systemd unit file for mariadb according to the
# instructions in http://fedoraproject.org/wiki/Systemd

[mysqld_safe]
log-error=/export/data/mysql/error.log
```

```
pid-file=/export/data/mysql/mysql.pid

#
# include all files from the config directory
#
!includedir /etc/my.cnf.d
```

将服务添加到开机自动启动

```
cp /export/server/mysql/support-files/mysql.server /etc/init.d/mysqld
```

启动服务

```
service mysqld start
```

配置环境变量，在`/etc/profile`中添加如下内容

```
# mysql env
MYSQL_HOME=/export/server/mysql
MYSQL_PATH=$MYSQL_HOME/bin
PATH=$PATH:$MYSQL_PATH
export PATH
```

使配置即可生效

```
source /etc/profile
```

使用`root`登录

```
mysql -uroot -p
# 这里填写之前初始化服务时提供的密码
```

登录上去之后，更改`root`账户密码（我为了方便将密码改为root），否则操作数据库会报错

```
set password=password('root');
flush privileges;
```

设置服务可被所有远程客户端访问

```
use mysql;
update user set host='%' where user='root';
flush privileges;
```

> 这样就可以在宿主机使用`navicat`远程连接虚拟机linux上的mysql了

配置主从节点

配置master

以`linux`（`192.168.10.10`）上的`mysql`为`master`，宿主机（`192.168.10.1`）上的`mysql`为`slave`配置主从复制。

修改`master`的`my.cnf`如下

```
[mysqld]
basedir=/export/server/mysql
datadir=/export/data/mysql
socket=/tmp/mysql.sock
user=mysql
server-id=10
port=3306
# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
# Settings user and group are ignored when systemd is used.
# If you need to run mysqld under a different user or group,
# customize your systemd unit file for mariadb according to the
# instructions in http://fedoraproject.org/wiki/Systemd

log-bin=mysql-bin   # 开启二进制日志
expire-logs-days=7  # 设置日志过期时间，避免占满磁盘
binlog-ignore-db=mysql  # 不使用主从复制的数据库
binlog-ignore-db=information_schema
binlog-ignore-db=performation_schema
binlog-ignore-db=sys
binlog-do-db=test   #使用主从复制的数据库

[mysqld_safe]
log-error=/export/data/mysql/error.log
pid-file=/export/data/mysql/mysql.pid

#
# include all files from the config directory
#
!includedir /etc/my.cnf.d
```

重启`master`

```
service mysqld restart
```

登录`master`查看配置是否生效（`ON`即为开启，默认为`OFF`）：

```
mysql> show variables like 'log_bin';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| log_bin       | ON    |
+---------------+-------+
```

在`master`的数据库中建立备份账号：`backup`为用户名，`%`表示任何远程地址，用户`back`可以使用密码`1234`通过任何远程客户端连接`master`

```
grant replication slave on *.* to 'backup'@'%' identified by '1234'
```

查看`user`表可以看到我们刚创建的用户：

```
mysql> use mysql
mysql> select user,authentication_string,host from user;
+---------------+-------------------------------------------+-----------+
| user          | authentication_string                     | host      |
+---------------+-------------------------------------------+-----------+
| root          | *81F5E21E35407D884A6CD4A731AEBFB6AF209E1B | %         |
| mysql.session | *THISISNOTAVALIDPASSWORDTHATCANBEUSEDHERE | localhost |
| mysql.sys     | *THISISNOTAVALIDPASSWORDTHATCANBEUSEDHERE | localhost |
| backup        | *A4B6157319038724E3560894F7F932C8886EBFCF | %         |
+---------------+-------------------------------------------+-----------+
```

新建`test`数据库，创建一个`article`表以备后续测试

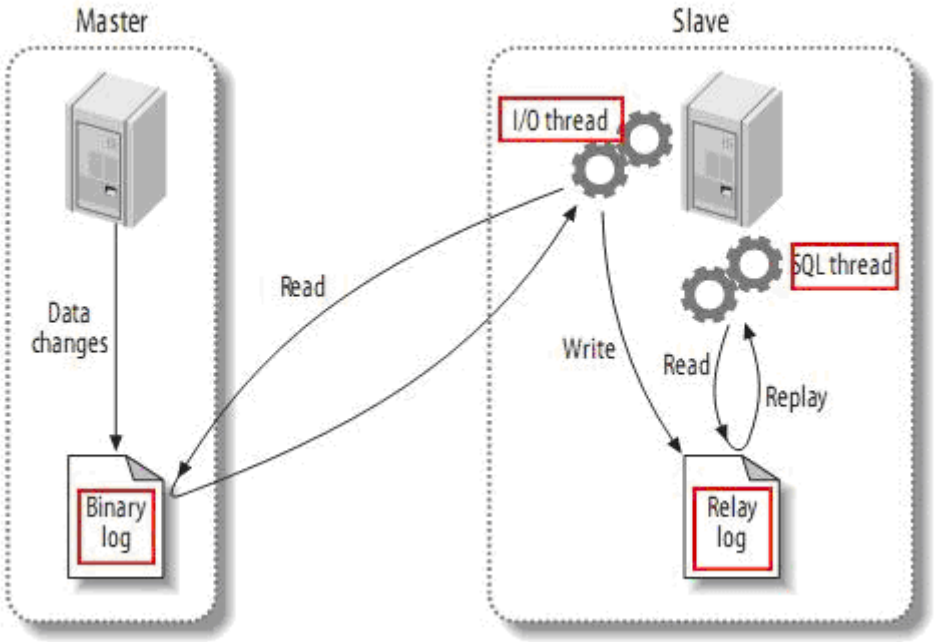
```
CREATE TABLE `article` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(64) DEFAULT NULL,
  `content` text,
  PRIMARY KEY (`id`)
) CHARSET=utf8;
```

重启服务并刷新数据库状态到存储文件中（`with read lock`表示在此过程中，客户端只能读数据，以便获得一个一致性的快照）

```
[root@zhenganwen ~]# service mysqld restart
Shutting down MySQL.... SUCCESS!
Starting MySQL. SUCCESS!
[root@zhenganwen mysql]# mysql -uroot -proot
mysql> flush tables with read lock;
Query OK, 0 rows affected (0.00 sec)
```

查看`master`上当前的二进制日志和偏移量（记一下其中的`File`和`Position`）

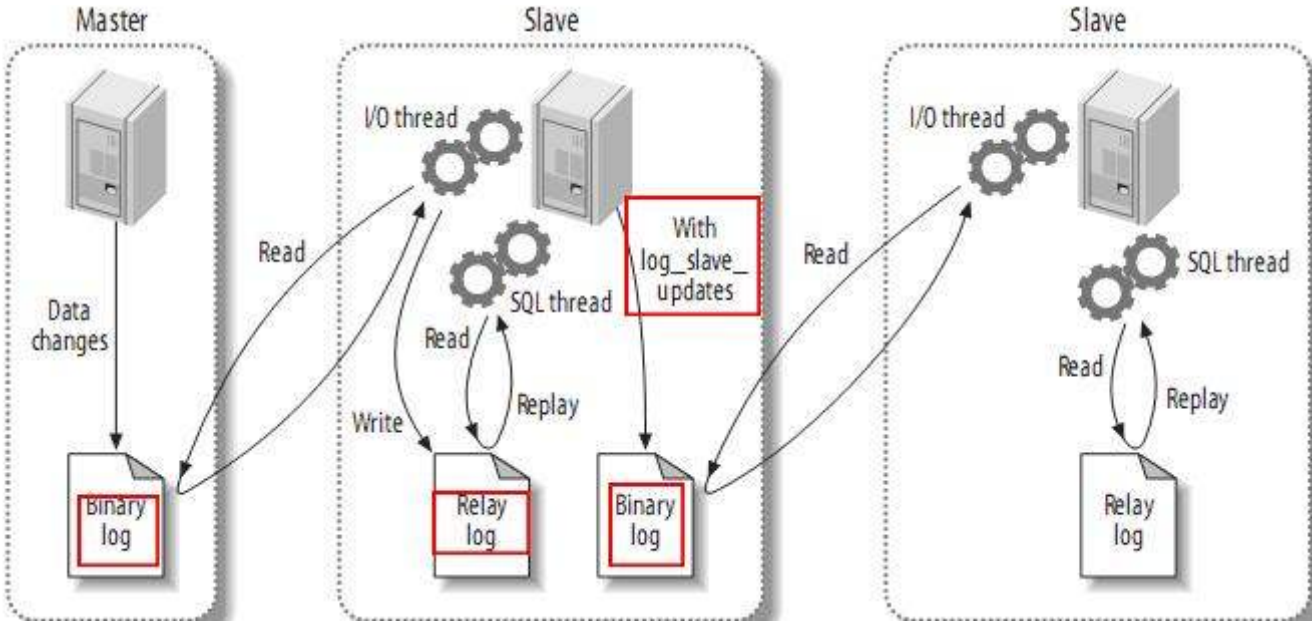
```
mysql> show master status \G
*************************** 1. row ***************************
             File: mysql-bin.000002
         Position: 154
     Binlog_Do_DB: test
 Binlog_Ignore_DB: mysql,information_schema,performation_schema,sys
Executed_Gtid_Set:
1 row in set (0.00 sec)
```



`File`表示实现复制功能的日志，即上图中的`Binary log`； `Position`则表示`Binary log`日志文件的偏移量之后的都会同步到`slave`中，那么在偏移量之前的则需要我们手动导入。

主服务器上面的任何修改都会保存在二进制日志Binary log里面，从服务器上面启动一个I/O thread（实际上就是一个主服务器的客户端进程），连接到主服务器上面请求读取二进制日志，然后把读取到的二进制日志写到本地的一个Realy log里面。从服务器上面开启一个SQL thread定时检查Realy log，如果发现有更改立即把更改的内容在本机上面执行一遍。

如果一主多从的话，这时主库既要负责写又要负责为几个从库提供二进制日志。此时可以稍做调整，将二进制日志只给某一从，这一从再开启二进制日志并将自己的二进制日志再发给其它从。或者是干脆这个从不记录只负责将二进制日志转发给其它从，这样架构起来性能可能要好得多，而且数据之间的延时应该也稍微要好一些



手动导入，从`master`中导出数据

```
mysqldump -uroot -proot -hlocalhost test > /export/data/test.sql
```

将`test.sql`中的内容在`slave`上执行一遍。

配置slave

修改`slave`的`my.ini`文件中的`[mysqld]`部分

```
log-bin=mysql
server-id=1 #192.168.10.1
```

保存修改后重启`slave`， `WIN+R` -> `services.msc` -> `MySQL5.7` ->重新启动

登录`slave`检查`log_bin`是否以被开启：

```
show VARIABLES like 'log_bin';
```

配置与`master`的同步复制：

```
stop slave;
change master to
    master_host='192.168.10.10', -- master的IP
    master_user='backup',      -- 之前在master上创建的用户
    master_password='1234',
    master_log_file='mysql-bin.000002', -- master上 show master status \G 提供的信息
    master_log_pos=154;
```

启用 `slave` 节点并查看状态

```
mysql> start slave;
mysql> show slave status \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.10.10
                  Master_User: backup
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000002
          Read_Master_Log_Pos: 154
               Relay_Log_File: DESKTOP-KUBSPE0-relay-bin.000002
                Relay_Log_Pos: 320
        Relay_Master_Log_File: mysql-bin.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 154
              Relay_Log_Space: 537
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids:
             Master_Server_Id: 10
                  Master_UUID: f68774b7-0b28-11e9-a925-000c290abe05
             Master_Info_File: C:\ProgramData\MySQL\MySQL Server 5.7\Data\master.info
                    SQL_Delay: 0
          SQL_Remaining_Delay: NULL
      Slave_SQL_Running_State: Slave has read all relay log; waiting for more updates
           Master_Retry_Count: 86400
                  Master_Bind:
      Last_IO_Error_Timestamp:
     Last_SQL_Error_Timestamp:
               Master_SSL_Crl:
           Master_SSL_Crlpath:
           Retrieved_Gtid_Set:
            Executed_Gtid_Set:
```

```
                  Auto_Position: 0
           Replicate_Rewrite_DB:
                   Channel_Name:
             Master_TLS_Version:
1 row in set (0.00 sec)
```

> 注意查看第4、14、15三行，若与我一致，表示`slave`配置成功

测试

关闭`master`的读取锁定

```
mysql> unlock tables;
Query OK, 0 rows affected (0.00 sec)
```

向`master`中插入一条数据

```
mysql> use test
mysql> insert into article (title,content) values ('mysql master and slave','record the cluster
building succeed!:)');
Query OK, 1 row affected (0.00 sec)
```

查看`slave`是否自动同步了数据

```
mysql> insert into article (title,content) values ('mysql master and slave','record the cluster
building succeed!:)');
Query OK, 1 row affected (0.00 sec)
```

至此，主从复制的配置成功！：）

使用mysqlreplicate命令快速搭建 Mysql 主从复制

读写分离

读写分离是依赖于主从复制，而主从复制又是为读写分离服务的。因为主从复制要求`slave`不能写只能读（如果对`slave`执行写操作，那么`show slave status`将会呈现`Slave_SQL_Running=NO`，此时你需要按照前面提到的手动同步一下`slave`）。

方案一、定义两种连接

就像我们在学JDBC时定义的`DataBase`一样，我们可以抽取出`ReadDataBase,WriteDataBase implements DataBase`，但是这种方式无法利用优秀的线程池技术如`DruidDataSource`帮我们管理连接，也无法利用`Spring AOP`让连接对`DAO`层透明。

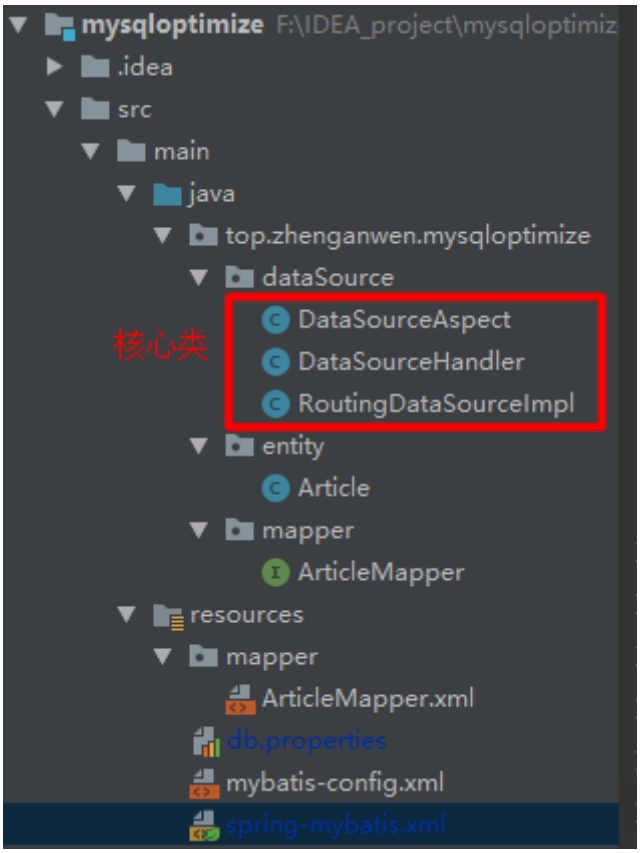
方案二、使用Spring AOP

如果能够使用`Spring AOP`解决数据源切换的问题，那么就可以和`Mybatis`、`Druid`整合到一起了。

我们在整合`Spring1`和`Mybatis`时，我们只需写DAO接口和对应的`SQL`语句，那么DAO实例是由谁创建的呢？实际上就是`Spring`帮我们创建的，它通过我们注入的数据源，帮我们完成从中获取数据库连接、使用连接执行`SQL`语句的过程以及最后归还连接给数据源的过程。

如果我们能在调用DAO接口时根据接口方法命名规范（增`addXXX/createXXX`、删`deleteXX/removeXXX`、改`updateXXXX`、查`selectXX/findXXX/getXX/queryXXX`）动态地选择数据源（读数据源对应连接`master`而写数据源对应连接`slave`），那么就可以做到读写分离了。

项目结构



引入依赖

其中，为了方便访问数据库引入了`mybatis`和`druid`，实现数据源动态切换主要依赖`spring-aop`和`spring-aspects`

```
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.2</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.6</version>
    </dependency>
```

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.6</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>6.0.2</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.22</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.0.8.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

    </dependencies>
```

数据类

```java
package top.zhenganwen.mysqloptimize.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Article {

    private int id;
    private String title;
    private String content;
}
```

spring配置文件

其中`RoutingDataSourceImpl`是实现动态切换功能的核心类，稍后介绍。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="db.properties"></context:property-placeholder>

    <context:component-scan base-package="top.zhenganwen.mysqloptimize"/>

    <bean id="slaveDataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${db.driverClass}"/>
        <property name="url" value="${master.db.url}"></property>
        <property name="username" value="${master.db.username}"></property>
        <property name="password" value="${master.db.password}"></property>
    </bean>

    <bean id="masterDataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${db.driverClass}"/>
        <property name="url" value="${slave.db.url}"></property>
        <property name="username" value="${slave.db.username}"></property>
        <property name="password" value="${slave.db.password}"></property>
    </bean>

    <bean id="dataSourceRouting" class="top.zhenganwen.mysqloptimize.dataSource.RoutingDataSour
ceImpl">
        <property name="defaultTargetDataSource" ref="masterDataSource"></property>
        <property name="targetDataSources">
            <map key-type="java.lang.String" value-type="javax.sql.DataSource">
                <entry key="read" value-ref="slaveDataSource"/>
                <entry key="write" value-ref="masterDataSource"/>
            </map>
```

```
            </property>
            <property name="methodType">
                <map key-type="java.lang.String" value-type="java.lang.String">
                    <entry key="read" value="query,find,select,get,load,"></entry>
                    <entry key="write" value="update,add,create,delete,remove,modify"/>
                </map>
            </property>
        </bean>

        <!-- Mybatis文件 -->
        <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
            <property name="configLocation" value="classpath:mybatis-config.xml" />
            <property name="dataSource" ref="dataSourceRouting" />
            <property name="mapperLocations" value="mapper/*.xml"/>
        </bean>

        <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
            <property name="basePackage" value="top.zhenganwen.mysqloptimize.mapper" />
            <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
        </bean>
    </beans>
```

`dp.properties`

```
    master.db.url=jdbc:mysql://localhost:3306/test?
    useUnicode=true&characterEncoding=utf8&serverTimezone=UTC
    master.db.username=root
    master.db.password=root

    slave.db.url=jdbc:mysql://192.168.10.10:3306/test?
    useUnicode=true&characterEncoding=utf8&serverTimezone=UTC
    slave.db.username=root
    slave.db.password=root

    db.driverClass=com.mysql.jdbc.Driver
```

`mybatis-config.xml`

```
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE configuration
            PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
            "http://mybatis.org/dtd/mybatis-3-config.dtd">
    <configuration>
        <typeAliases>
            <typeAlias type="top.zhenganwen.mysqloptimize.entity.Article" alias="Article"/>
        </typeAliases>
    </configuration>
```

mapper接口和配置文件

`ArticleMapper.java`

```java
package top.zhenganwen.mysqloptimize.mapper;

import org.springframework.stereotype.Repository;
import top.zhenganwen.mysqloptimize.entity.Article;

import java.util.List;

@Repository
public interface ArticleMapper {

    List<Article> findAll();

    void add(Article article);

    void delete(int id);

}
```

`ArticleMapper.xml`

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="top.zhenganwen.mysqloptimize.mapper.ArticleMapper">
    <select id="findAll" resultType="Article">
        select * from article
    </select>

    <insert id="add" parameterType="Article">
        insert into article (title,content) values (#{title},#{content})
    </insert>

    <delete id="delete" parameterType="int">
        delete from article where id=#{id}
    </delete>
</mapper>
```

核心类

RoutingDataSourceImpl

```java
package top.zhenganwen.mysqloptimize.dataSource;

import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

import java.util.*;

/**
 * RoutingDataSourceImpl class
 * 数据源路由
 *
 * @author zhenganwen, blog:zhenganwen.top
 * @date 2018/12/29
```

```
 */
public class RoutingDataSourceImpl extends AbstractRoutingDataSource {

    /**
     * key为read或write
     * value为DAO方法的前缀
     * 什么前缀开头的方法使用读数据员，什么开头的方法使用写数据源
     */
    public static final Map<String, List<String>> METHOD_TYPE_MAP = new HashMap<String,
List<String>>();

    /**
     * 由我们指定数据源的id，由Spring切换数据源
     *
     * @return
     */
    @Override
    protected Object determineCurrentLookupKey() {
        System.out.println("数据源为: "+DataSourceHandler.getDataSource());
        return DataSourceHandler.getDataSource();
    }

    public void setMethodType(Map<String, String> map) {
        for (String type : map.keySet()) {
            String methodPrefixList = map.get(type);
            if (methodPrefixList != null) {
                METHOD_TYPE_MAP.put(type, Arrays.asList(methodPrefixList.split(",")));
            }
        }
    }
}
```

它的主要功能是，本来我们只配置一个数据源，因此`Spring`动态代理DAO接口时直接使用该数据源，现在我们有了读、写两个数据源，我们需要加入一些自己的逻辑来告诉调用哪个接口使用哪个数据源（读数据的接口使用`slave`，写数据的接口使用`master`。这个告诉`Spring`该使用哪个数据源的类就是`AbstractRoutingDataSource`，必须重写的方法`determineCurrentLookupKey`返回数据源的标识，结合`spring`配置文件（下段代码的5，6两行）

```
<bean id="dataSourceRouting" class="top.zhenganwen.mysqloptimize.dataSource.RoutingDataSourceIm
pl">
    <property name="defaultTargetDataSource" ref="masterDataSource"></property>
    <property name="targetDataSources">
        <map key-type="java.lang.String" value-type="javax.sql.DataSource">
            <entry key="read" value-ref="slaveDataSource"/>
            <entry key="write" value-ref="masterDataSource"/>
        </map>
    </property>
    <property name="methodType">
        <map key-type="java.lang.String" value-type="java.lang.String">
            <entry key="read" value="query,find,select,get,load,"></entry>
            <entry key="write" value="update,add,create,delete,remove,modify"/>
        </map>
    </property>
</bean>
```

如果`determineCurrentLookupKey`返回`read`那么使用`slaveDataSource`，如果返回`write`就使用`masterDataSource`。

DataSourceHandler

```
package top.zhenganwen.mysqloptimize.dataSource;

/**
 * DataSourceHandler class
 * <p>
 * 将数据源与线程绑定，需要时根据线程获取
 *
 * @author zhenganwen, blog:zhenganwen.top
 * @date 2018/12/29
 */
public class DataSourceHandler {

    /**
     * 绑定的是read或write，表示使用读或写数据源
     */
    private static final ThreadLocal<String> holder = new ThreadLocal<String>();

    public static void setDataSource(String dataSource) {
        System.out.println(Thread.currentThread().getName()+"设置了数据源类型");
        holder.set(dataSource);
    }

    public static String getDataSource() {
        System.out.println(Thread.currentThread().getName()+"获取了数据源类型");
        return holder.get();
    }
}
```

DataSourceAspect

```
package top.zhenganwen.mysqloptimize.dataSource;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.Set;

import static top.zhenganwen.mysqloptimize.dataSource.RoutingDataSourceImpl.METHOD_TYPE_MAP;

/**
 * DataSourceAspect class
 *
 * 配置切面，根据方法前缀设置读、写数据源
 * 项目启动时会加载该bean，并按照配置的切面（哪些切入点、如何增强）确定动态代理逻辑
 * @author zhenganwen,blog:zhenganwen.top
```

```
 * @date 2018/12/29
 */
@Component
//声明这是一个切面，这样Spring才会做相应的配置，否则只会当做简单的bean注入
@Aspect
@EnableAspectJAutoProxy
public class DataSourceAspect {

    /**
     * 配置切入点：DAO包下的所有类的所有方法
     */
    @Pointcut("execution(* top.zhenganwen.mysqloptimize.mapper.*.*(..))")
    public void aspect() {

    }

    /**
     * 配置前置增强，对象是aspect()方法上配置的切入点
     */
    @Before("aspect()")
    public void before(JoinPoint point) {
        String className = point.getTarget().getClass().getName();
        String invokedMethod = point.getSignature().getName();
        System.out.println("对 "+className+"$"+invokedMethod+" 做了前置增强，确定了要使用的数据源类型");

        Set<String> dataSourceType = METHOD_TYPE_MAP.keySet();
        for (String type : dataSourceType) {
            List<String> prefixList = METHOD_TYPE_MAP.get(type);
            for (String prefix : prefixList) {
                if (invokedMethod.startsWith(prefix)) {
                    DataSourceHandler.setDataSource(type);
                    System.out.println("数据源为: "+type);
                    return;
                }
            }
        }
    }
}
```

测试读写分离

> 如何测试读是从`slave`中读的呢？可以将写后复制到`slave`中的数据更改，再读该数据就知道是从`slave`中读了。注意，一但对`slave`做了写操作就要重新手动将`slave`与`master`同步一下，否则主从复制就会失效。

```
package top.zhenganwen.mysqloptimize.dataSource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import top.zhenganwen.mysqloptimize.entity.Article;
import top.zhenganwen.mysqloptimize.mapper.ArticleMapper;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-mybatis.xml")
public class RoutingDataSourceTest {

    @Autowired
    ArticleMapper articleMapper;

    @Test
    public void testRead() {
        System.out.println(articleMapper.findAll());
    }

    @Test
    public void testAdd() {
        Article article = new Article(0, "我是新插入的文章", "测试是否能够写到master并且复制到
slave中");
        articleMapper.add(article);
    }

    @Test
    public void testDelete() {
        articleMapper.delete(2);
    }
}
```

负载均衡

负载均衡算法

- 轮询
- 加权轮询：按照处理能力来加权
- 负载分配：依据当前的空闲状态（但是测试每个节点的内存使用率、CPU利用率等，再做比较选出最闲的那个，效率太低）

高可用

在服务器架构时，为了保证服务器7x24不宕机在线状态，需要为每台单点服务器（由一台服务器提供服务的服务器，如写服务器、数据库中间件）提供冗余机。

对于写服务器来说，需要提供一台同样的写-冗余服务器，当写服务器健康时（写-冗余通过心跳检测），写-冗余作为一个从机的角色复制写服务器的内容与其做一个同步；当写服务器宕机时，写-冗余服务器便顶上来作为写服务器继续提供服务。对外界来说这个处理过程是透明的，即外界仅通过一个IP访问服务。

典型SQL

线上DDL

DDL(Database Definition Language)是指数据库表结构的定义（`create table`）和维护（`alter table`）的语言。在线上执行DDL，在低于`MySQL5.6`版本时会导致全表被独占锁定，此时表处于维护、不可操作状态，这会导致该期间对该表的所有访问无法响应。但是在`MySQL5.6`之后，支持`Online DDL`，大大缩短了锁定时间。

优化技巧是采用的维护表结构的DDL（比如增加一列，或者增加一个索引），是copy策略。思路：创建一个满足新结构的新表，将旧表数据逐条导入（复制）到新表中，以保证一次性锁定的内容少（锁定的是正在导入的数据），同时旧表上可以执行其他任务。导入的过程中，将对旧表的所有操作以日志的形式记录下来，导入完毕后，将更新日志在新表上再执行一遍（确保一致性）。最后，新表替换旧表（在应用程序中完成，或者是数据库的rename，视图完成）。

但随着MySQL的升级，这个问题几乎淡化了。

数据库导入语句

在恢复数据时，可能会导入大量的数据。此时为了快速导入，需要掌握一些技巧：

1. 导入时先禁用索引和约束：

```
alter table table-name disable keys
```

待数据导入完成之后，再开启索引和约束，一次性创建索引

```
alter table table-name enable keys
```

2. 数据库如果使用的引擎是`Innodb`，那么它默认会给每条写指令加上事务（这也会消耗一定的时间），因此建议先手动开启事务，再执行一定量的批量导入，最后手动提交事务。
3. 如果批量导入的SQL指令格式相同只是数据不同，那么你应该先`prepare`预编译一下，这样也能节省很多重复编译的时间。

limit offset,rows

尽量保证不要出现大的`offset`，比如`limit 10000,10`相当于对已查询出来的行数弃掉前`10000`行后再取`10`行，完全可以加一些条件过滤一下（完成筛选），而不应该使用`limit`跳过已查询到的数据。这是一个==`offset`做无用功==的问题。对应实际工程中，要避免出现大页码的情况，尽量引导用户做条件过滤。

select * 要少用

即尽量选择自己需要的字段`select`，但这个影响不是很大，因为网络传输多了几十上百字节也没多少延时，并且现在流行的ORM框架都是用的`select *`，只是我们在设计表的时候注意将大数据量的字段分离，比如商品详情可以单独抽离出一张商品详情表，这样在查看商品简略页面时的加载速度就不会有影响了。

order by rand()不要用

它的逻辑就是随机排序（为每条数据生成一个随机数，然后根据随机数大小进行排序）。如`select * from student order by rand() limit 5`的执行效率就很低，因为它为表中的每条数据都生成随机数并进行排序，而我们只要前5条。

解决思路：在应用程序中，将随机的主键生成好，去数据库中利用主键检索。

单表和多表查询

多表查询：`join`、子查询都是涉及到多表的查询。如果你使用`explain`分析执行计划你会发现多表查询也是一个表一个表的处理，最后合并结果。因此可以说单表查询将计算压力放在了应用程序上，而多表查询将计算压力放在了数据库上。

现在有ORM框架帮我们解决了单表查询带来的对象映射问题（查询单表时，如果发现有外键自动再去查询关联表，是一个表一个表查的）。

count(*)

在`MyISAM`存储引擎中，会自动记录表的行数，因此使用`count(*)`能够快速返回。而`Innodb`内部没有这样一个计数器，需要我们手动统计记录数量，解决思路就是单独使用一张表：

id	table	count
1	student	100

limit 1

如果可以确定仅仅检索一条，建议加上`limit 1`，其实ORM框架帮我们做到了这一点（查询单条的操作都会自动加上`limit 1`）。

慢查询日志

> 用于记录执行时间超过某个临界值的SQL日志，用于快速定位慢查询，为我们的优化做参考。

开启慢查询日志

配置项：`slow_query_log`

可以使用`show variables like 'slov_query_log'`查看是否开启，如果状态值为`OFF`，可以使用`set GLOBAL slow_query_log = on`来开启，它会在`datadir`下产生一个`xxx-slow.log`的文件。

设置临界时间

配置项：`long_query_time`

查看：`show VARIABLES like 'long_query_time'`，单位秒

设置：`set long_query_time=0.5`

实操时应该从长时间设置到短的时间，即将最慢的SQL优化掉

查看日志

一旦SQL超过了我们设置的临界时间就会被记录到`xxx-slow.log`中

profile信息

配置项：`profiling`

开启profile

`set profiling=on`

开启后，所有的SQL执行的详细信息都会被自动记录下来

```
mysql> show variables like 'profiling';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| profiling     | OFF   |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> set profiling=on;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

查看profile信息

`show profiles`

```
mysql> show variables like 'profiling';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| profiling     | ON    |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> insert into article values (null,'test profile',':)');
Query OK, 1 row affected (0.15 sec)

mysql> show profiles;
```

```
+----------+------------+-----------------------------------------------------------+
| Query_ID | Duration   | Query                                                     |
+----------+------------+-----------------------------------------------------------+
|        1 | 0.00086150 | show variables like 'profiling'                           |
|        2 | 0.15027550 | insert into article values (null,'test profile',':)') |
+----------+------------+-----------------------------------------------------------+
```

通过Query_ID查看某条SQL所有详细步骤的时间

`show profile for query Query_ID`

上面`show profiles`的结果中，每个SQL有一个`Query_ID`，可以通过它查看执行该SQL经过了哪些步骤，各消耗了多场时间

典型的服务器配置

> 以下的配置全都取决于实际的运行环境

- `max_connections`，最大客户端连接数

```
mysql> show variables like 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
```

- `table_open_cache`，表文件句柄缓存（表数据是存储在磁盘上的，缓存磁盘文件的句柄方便打开文件读取数据）

```
mysql> show variables like 'table_open_cache';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| table_open_cache | 2000  |
+------------------+-------+
```

- `key_buffer_size`，索引缓存大小（将从磁盘上读取的索引缓存到内存，可以设置大一些，有利于快速检索）

```
mysql> show variables like 'key_buffer_size';
+-----------------+---------+
| Variable_name   | Value   |
+-----------------+---------+
| key_buffer_size | 8388608 |
+-----------------+---------+
```

- `innodb_buffer_pool_size`， `Innodb`存储引擎缓存池大小（对于`Innodb`来说最重要的一个配置，如果所有的表用的都是`Innodb`，那么甚至建议将该值设置到物理内存的80%，`Innodb`的很多性能提升如索引都是依靠这个）

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+---------+
| Variable_name           | Value   |
+-------------------------+---------+
| innodb_buffer_pool_size | 8388608 |
+-------------------------+---------+
```

- `innodb_file_per_table`（`innodb`中，表数据存放在`.ibd`文件中，如果将该配置项设置为`ON`，那么一个表对应一个`ibd`文件，否则所有`innodb`共享表空间）

压测工具mysqlslap

安装MySQL时附带了一个压力测试工具`mysqlslap`（位于`bin`目录下）

自动生成sql测试

```
C:\Users\zaw>mysqlslap --auto-generate-sql -uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Average number of seconds to run all queries: 1.219 seconds
        Minimum number of seconds to run all queries: 1.219 seconds
        Maximum number of seconds to run all queries: 1.219 seconds
        Number of clients running queries: 1
        Average number of queries per client: 0
```

并发测试

```
C:\Users\zaw>mysqlslap --auto-generate-sql --concurrency=100 -uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Average number of seconds to run all queries: 3.578 seconds
```

```
        Minimum number of seconds to run all queries: 3.578 seconds
        Maximum number of seconds to run all queries: 3.578 seconds
        Number of clients running queries: 100
        Average number of queries per client: 0

C:\Users\zaw>mysqlslap --auto-generate-sql --concurrency=150 -uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Average number of seconds to run all queries: 5.718 seconds
        Minimum number of seconds to run all queries: 5.718 seconds
        Maximum number of seconds to run all queries: 5.718 seconds
        Number of clients running queries: 150
        Average number of queries per client: 0
```

多轮测试

```
C:\Users\zaw>mysqlslap --auto-generate-sql --concurrency=150 --iterations=10 -uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Average number of seconds to run all queries: 5.398 seconds
        Minimum number of seconds to run all queries: 4.313 seconds
        Maximum number of seconds to run all queries: 6.265 seconds
        Number of clients running queries: 150
        Average number of queries per client: 0
```

存储引擎测试

```
C:\Users\zaw>mysqlslap --auto-generate-sql --concurrency=150 --iterations=3 --engine=innodb -
uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Running for engine innodb
        Average number of seconds to run all queries: 5.911 seconds
        Minimum number of seconds to run all queries: 5.485 seconds
        Maximum number of seconds to run all queries: 6.703 seconds
        Number of clients running queries: 150
        Average number of queries per client: 0
```

```
C:\Users\zaw>mysqlslap --auto-generate-sql --concurrency=150 --iterations=3 --engine=myisam -
uroot -proot
mysqlslap: [Warning] Using a password on the command line interface can be insecure.
Benchmark
        Running for engine myisam
        Average number of seconds to run all queries: 53.104 seconds
        Minimum number of seconds to run all queries: 46.843 seconds
        Maximum number of seconds to run all queries: 60.781 seconds
        Number of clients running queries: 150
        Average number of queries per client: 0
```