# 基本概念

数据库本质是一种高效管理大量列表信息的办法,缩短记录归档时间,缩短记录检索时间,灵活的检索时间,灵活的输出格式,多用户同时访问记录,记录可以远程访问和电子传送

关系型数据库管理系统（RDNMS）

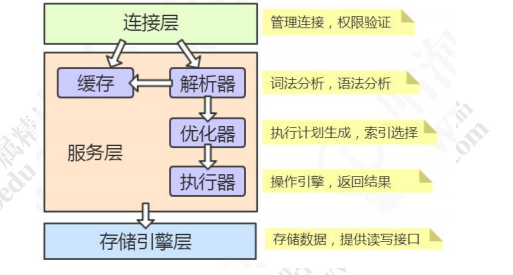
数据库（DataBase，DB）

管理系统（Management System）

关系（Relational）:通过两个表里的共同元素，将分别存放在两个表里的信息联系起来

Mysql是关系型数据库管理系统，mysql并不完美，但是却足够灵活，能够适应高要求的环境，可以支持数据仓库，内容索引，高可用的冗余系统等非常多的应用

# Mysql逻辑架构



最上层连接层：用来负者连接处理，授权认证，安全

每个客户端连接都会在服务器进程中拥有一个线程，这个连接的查询只会在这个单独的线程中执行；服务器会缓存这个线程，因此不需要为每个新的连接创建或者销毁线程

第二层架构服务层：大多数mysql的核心服务功能都在这一层，包括查询解析，分析，优化，缓存以及所有的内置函数（），所有跨存储引擎的功能都在这一层：存储过程，视图，触发器

第三次包含优化器：存储引擎赋值mysql中的数据提取和存储。服务器通过API和存储引擎进行网络通信；这些（API）接口屏蔽了不同存储引擎之间的差异，使得这些差异对上层的查询过程透明；存储引擎不会执行sql，不同存储引擎之间也不会进行通信，他们只是响应上层服务器的请求

## InnoDB的内存结构

InnoDB的内存结构

缓冲池（buffer pool）

buffer pool缓冲的是页面信息，包括数据页，索引页

**InnoDB的数据是放在磁盘上的，InnoDB操作数据有一个最小的逻辑单位，叫做页（数据页和索引页）；**

对数据的操作不是每次都操作磁盘中的，因为磁盘的速度太慢了；InnoDB使用了一种缓冲技术，也就是把磁盘读到的页放到

块内存区域中，这个区域也叫作buffer pool，缓冲页面信息；下一次读取相同的页时，首先判断是否在缓冲池里面，如果是，就直接读取，不

用在访问磁盘

修改数据的时候，先修改缓冲池中的数据，如果缓冲池的数据和磁盘的数据不一样的时候，这个就叫脏页；InnoDB中有专门的后天线程把buffer pool中

的数据写入到磁盘，每隔一段时间就一次性地把多个修改写入磁盘，这个动作就叫做刷脏；

change buffer写缓冲

当不存在数据重复时，不需要从磁盘加载索引页判断数据是不是重复的，那么就可以将修改记录在内存缓冲池中，从而提升更新语句

（update，delete，insert）的执行速度，这块记录修改记录的内存区域就叫做change buffer；把change buffer记录到数据页的操作叫做merge

如果数据库大部分索引都是非唯一索引，并且业务写多读少，不会立刻在写数据后读取，就可以使用change buffer；

写多读少的业务，调大这个值：SHOW VARIABLES LIKE 'innodb\_change\_buffer\_max\_size';

代表 Change Buffer 占 Buffer Pool 的比例，默认 25%；

Adaptive Hash Index

索引应该是放在磁盘的，为什么要专门把一种哈希的索引放到内存？哈希索引是通过hash函数寻找到需要的值的，放在内存中是为了提升访问的速度

（redo）Log buffer

当buffer pool中的脏页没有刷新到磁盘，数据库宕机或者重启，这些数据就丢失了，如果写操作执行到了一半，数据文件可能被破坏不可用；

为了避免这些会造成数据丢失的问题，InnoDB会把所有对页的修改记录到一个日志文件文件中，并且数据库启动时从这个文件中恢复数据---

用它来实现事务的持久性，这个文件就叫做磁盘的redo log（重做日志）；先写日志在写磁盘

刷盘是随机 I/O，而记录日志是顺序 I/O，顺序 I/O 效率更高。因此先把修改写入日志，可以延迟刷盘时机，进而提升系统吞吐；

redo log的内容主要用于崩溃恢复，磁盘的数据文件，数据来自buffer pool，redo log是写入磁盘，不是写入文件

那么，Log Buffer 什么时候写入 log file？

在我们写入数据到磁盘的时候，操作系统本身是有缓存的。flush 就是把操作系统缓冲区写入到磁盘。

Buffer pool缓冲的是页面信息（包括数据页和索引页）默认大小是128M

change buffer写缓冲：把修改记录到缓存池中，提升更新语句（delete update insert）的执行速度

Log buffer(redo)(当buffer pool里面的脏页还没有刷入磁盘，数据库宕机或者重启后的数据丢失；或者写操作进行一半之后，

数据文件被破坏导致数据库不可用)

InnoDB把所有对页面的修改操作都记录在一个日志文件中，并且在数据启动时从这个文件进行恢复操作--用它来实现对事务的持久性

（先写日志，在写入磁盘）

redo log内容主要用于崩溃恢复，磁盘的数据文件，数据来自buffer pool。redo log 写入磁盘，不是写入数据文件

redo log有什么特点

redo log是InnoDB存储引擎实现的，并不是所有存储引擎都有的

不是记录更新之后的状态，而是记录数据更新的操作，属于物理日志

redo log文件大小是固定的，前面的内容会被覆盖

redo log :innodb存储层的日志，重做日志文件， 记录操作的日志；如数据库掉电，InnoDB存储引擎会使用redo log恢复到掉电前的时刻，以此来保证数据的完整性。redo log日志的大小是固定的，即记录满了以后就从头循环写。

undo log ： 保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读

binlog：mysql service层的日志 记录操作的日志

redo log和binlog区别

* redo log是属于innoDB层面，binlog属于MySQL Server层面的，这样在数据库用别的存储引擎时可以达到一致性的要求。
* redo log是物理日志，记录该数据页更新的内容；binlog是逻辑日志，记录的是这个更新语句的原始逻辑
* redo log是循环写，日志空间大小固定；binlog是追加写，是指一份写到一定大小的时候会更换下一个文件，不会覆盖。
* binlog可以作为恢复数据使用，主从复制搭建，redo log作为异常宕机或者介质故障后的数据恢复使用。

update T set c=c+1 where ID=2;

* 执行器先找引擎取 ID=2 这一行。ID 是主键，引擎直接用树搜索找到这一行。如果 ID=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
* 执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。
* 引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。
* 执行器生成这个操作的 binlog，并把 binlog 写入磁盘。
* 执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成。

## 磁盘结构

表空间是InnoDB存储引擎逻辑结构的最高层，，所有数据都放在表空间中；

InnoDB的表空间分为5大类

1.系统表空间：在默认情况下InnoDB存储引擎有一个共享表空间，包含了InnoDB数据字典和双写缓冲区，change buffer和undo logs,

包含了用户创建的表和索引数据；

双写缓冲：在数据页本身已经损坏的情况下，需要一个页的副本，如果出现了写入失效，就用页的副本来还原这个页，然后在用redo；

这个副本就是double write，InnoDB的双写技术。通过它实现数据页的可靠性；

double write 由两部分组成，一部分是内存的 double write，

一个部分是磁盘上的 double write。因为 double write 是顺序写入的，不会带来很大的开销。

数据字典：内存系统表组成，存储表和索引的元数据

undo在后面介绍

2.独占表空间

每张表独占一个表空间，通过开关innodb\_file\_per\_table设置，默认是开启的；开启后，每张表会开辟一个表空间，这个文件就是数据目录下的

.ibd文件，存放表的索引和数据

回滚信息，插入缓冲索引页，系统事务信息，二次写缓冲等还是存放在共享表空间内

3.通用表空间:用来存储不同数据库的表，数据路径和文件自定义

4.临时表空间：存储临时表的数据，包括用户创建的临时表，和磁盘的内部临时表，当服务器表正常关闭时，该表空间被删除

undo log（撤销日志或回滚日志）记录了事务发生之前的数据状态（不包括 select）。

如果修改数据时出现异常，可以用 undo log 来实现回滚操作（保持原子性）。

在执行 undo 的时候，仅仅是将数据从逻辑上恢复至事务之前的状态，而不是从物

理页面上操作实现的，属于逻辑格式的日志。

redo Log 和 undo Log 与事务密切相关，统称为事务日志。

undo Log 的数据默认在系统表空间 ibdata1 文件中，因为共享表空间不会自动收

缩，也可以单独创建一个undo表空间。

# 数据库的建立

DML(data manipulation language)：数据操作语言，select insert delete update这几个sql,这4个命令用来对数据库里的数据进行操作的语言

DDL(data definition language)：数据库模式定义语言，用来描述数据库中要存储的现实世界实体的语言，主要的命令有CREATE、ALTER、DROP等，DDL主要是用在创建数据库，定义或改变表（TABLE）的结构，数据类型，表之间的链接和约束等初始化工作上

DCL(data control language):数据库控制功能，用来设置或者更改数据库用户或者权限的语句，包括（grant，deny，revoke）

TCL(transaction control language):事务控制语言，commit-提交事务，savepoint-在事务中设置保存点，可以回滚到此处，rollback-回滚，set transaction-改变事务选项

选择数据库

Use db\_name;

用客户端和数据库建立连接后可以使用user语句来选择一个默认的数据库；要选择数据库，用户必须具有数据库的访问权限，否则会保存

不一定要显示的选择数据库，在没有选择数据库的时候，可以用数据库名来引出表名：比如select \* from db\_name.table\_name;

选择数据库后，并不意味着只能使用该库的表，可以通过数据库名称来限定表名比如select \* from db\_name.table\_name;

创建数据库

使用语句Create database name;创建数据库，但是当前用户必须拥有建库的权限；

创建数据库的完整语法

Create database [if not exists] db\_name [charset set charset] [collate collation]

当创建数据库的时候，数据库可能已经存在了；如果继续执行创建，那么就会报错，那么可以在创建语句中加入if not exists来创建（create database if not exists name）

Charset是设置字符集，Collate是设置排序规则，这2个不设置就会使用数据库默认的字符集和排序规则

删除数据

删除必须要有drop的权限

Drop database db\_name

删除数据库会把所有的内容都删掉（包括表和存储过程）；一个数据库就是mysql的数据目录中的一个表，这个目录主要存放对象（表，视图，触发器）；如果删除失败通常是因为这个目录中包含了一些与数据库无关的文件，删除语句不会删除这些文件，需要手动删除后再来执行删除语句

更改数据库

更改数据库的全局属性，必须具有更改权限;目前只能修改字符集和排序规则

Alter database db\_name [character set charset] [collate collation]

# 表

创建表

Create table name

(

name varchar(10)

)

创建表时可以指定索引列；也可以在创建表后，在给这个表添加上索引，

设置表的存储引擎

在create语句的右括号之后加上一个或者多个可设置项；设置存储引擎engine=myisam

Create table name(…) engine=name;

没有设置存储引擎的话表默认是InnoDB

创建不存在的表

Create table if not exists name()

使用if not exists语句判断，如果存在就不创建，不存在表才创建表

创建临时表

临时表值对客户端可见，每个客户端只能看到自己创建的临时表，所以不同客户端创建的临时表名可以相同；临时表会在客户端会话结束后自动删除（最好是在结束使用后就删除，节约空间）；临时表的表名可以和永久表相同，不会对永久表造成任何影响，当和永久表同名的临时创建后，永久表就会隐藏，当临时表删除后隐藏的表就会重现;但是不能同时存在2个名字相同的临时表；

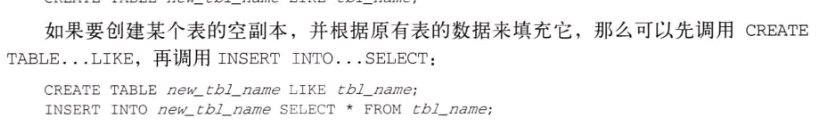
Create temporary table name()

更具查询结果或其他表来创建表

根据其他表创建新表

Create table … like 会根据原表创建一个新表，新表是原表的一个空副本；把原表结构复制过来，而且会保留列的所有属性，索引也会被复制过来，表是空的；

Create table new\_t\_name like tbl\_name;



根据查询结果创建表

Create table select：根据select语句的查询结果创建一个新表，默认情况下这个语句不会复制列属性，如auto\_increment列就不会被复制，不会复制索引；这个条语句可以完成表的创建和数据的填充

使用分区表

Mysql支持分区表，从而让表的内容分散存储在不同的物理存储位置

优点

1. 分区表对业务透明，只需要维护一个表的数据结构
2. DML操作加锁仅影响操作的分区，不会影响未访问的分区
3. 数据分区能有效的降低索引层数，提高查询效率

缺点

由于分区表在MySQL Server层为一个表，因此：

1、DDL操作需要锁定所有分区，导致所有分区上操作都被阻塞。

2、当表数据量较小时，分区表和非分区表性能相近，分区表效果有限。

3、当表数据量较大时，对分区表进行DDL或其他运维操作难度大风险高。

4、分区表使用较少，存在未知风险多，BUG多BUG多BUG多，MySQL社区版本免费，横向扩展成本低，分库分表实现简单且中间件完善。

5、当单台服务器性能无法满足时，对分区表进行分拆的成本较高，而分库分表能很容易实现横向分拆。

6、当分区表操作不当导致访问所有分区时，会导致严重的性能问题，而分库分表操作不当仅影响访问的表。

7、使用分库分表可以有效运维降低运维操作影响，对1亿数据量表做DDL操作需要谨慎评估，而对10万数据量表做DDL操作可以默认其很快完成。

8、使用分库分表可以有效减小宕机或其他故障影响，将数据分库分表到10套群集上，一套群集发生故障仅影响业务的一成。

创建分区表示例

Create table tbl

(

dt datetime not null,

info varchar(100) not null,

index(dt)

)

partition by rang(year(dt))

(

partition t1 less than (2011),

partition t2 less than(2012)

)

在createtable后面天机partition by 子句（定义一个将行分配到分区的分区函数），分区函数可以根据范围，值列表或者散列值来分配

删除表

要删除表首先要有删除权限

删除一张表 drop table name

删除多张表 drop table name1,name2…

删除不存在的表时，会报错，所以需要加if exists，当表存在时才执行删除，drop table if exists name;

删除临时表 drop temporary table name;

更改表结构

Alter table 语句可以用来修改表名，添加或删除列，列属性，更改存储引擎，创建或删除索引

Alter table的语法

Alter table name action[,action]..;

其中每个动作指的是对表的修改

修改示例

1. 修改列的数据类型

修改表的字段数据类型可以使用change或者modify，

Alter table tb modify c mediumint unsigned

Alter table tb change c c mediumint unsigned

修改表中字段c的类型为mediumint unsigned，change子句能够做到同时将该列重命名，如果不修改，那么也要重写名称2次

修改列字段长度，修改字符集

1. 修改表的存储引擎

Alter table tb engine=Myisam; 将存储引擎修改为myisam;

当打算用表改用另一种存储引擎时，能否实现，还要取决于两种存储引擎的功能是否兼容，比如InnoDB支持blob列，那么就不能将其转换为blob，memory不支持blob列

1. 重命名表

Alter table old\_name to newName;

Rename table on1 to n1,on2 to n2;

Rename可以同时修改多个table

# 查询

## 连接查询

笛卡尔集：

内连接：把一个表的行和另一个表之间通过某些列进行匹配，得到那些列值匹配的行的结果

inner join (inner关键字可以省略) [inner join比left join快]

从左表中取出每一条记录,与右表中的所有记录进行匹配

匹配必须是某个条件,在左表和右表中相同【公共部分】,才会保留结果.否则,不保留

外链接（左连接和有连接）

左外连接: 包含左边表的全部行（不管右边的表中是否存在与它们匹配的行），以及右边表中全部匹配的行。

对于左表查询出的每一行，不管他是否匹配右表，都会强制结果集包含一行记录，结果集里对于右表没有匹配的行，右表列都会为null

【通俗：就是2张表，查左表满足条件的所有以及右表中含有左表条件的数据，where (右表条件)..is not null显示不为null的数】

以某张表为主,取出里面的所有记录.每条与另外一张表.不管能不能匹配上条件.最终都会保留.如果不能匹配,那么其他表的字段都置空

left join (left join 是left outer join的简写)

基本语法

SELECT FROM [左表] LEFT JOIN 右表] ON [条件]

应用

当左表想找到那些在右表无法匹配的行时可以使用left join,实现功能只需使用where 子句把右表为null的筛选出来就可以了

Select t1.\*,t2.\* from table1 t1 left join table2 t2 on t1.c1=t2.c1

Where t2.c1 is null

## 子查询

子查询是指用括号括起来，并嵌入另一个sql的中select语句查询

子查询可以返回不同的类型

标量子查询返回一个值

列子查询返回由一个值或者多个值构成的列结果

行子查询返回由一个值或者多个值构成的行结果

表子查询返回由一个行或者多个行构成的表

子查询结果的操作

1. 带关系比较运算符的子查询（=，<,>…）

子查询前面有一个值和一个关系比较运算符，那么子查询就只能有一个查询结果，也就是说这是一个标量子查询，如果返回多个值，那么就会以失败告终

1. not in 和in子查询

当返回的多个行来和外层查询进行比较运算时，可以使用in 和not，测试某个给定的比较值是否存在于某一组值

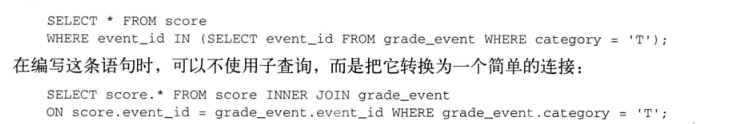
1. exist 和not exist

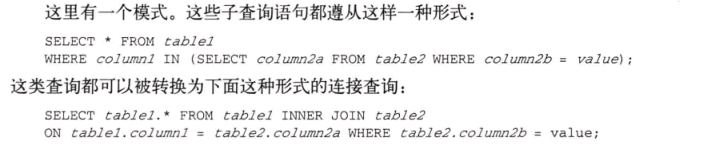
测试子查询是否返回了行，只是用来判断是否返回了行，所以返回的内容就不需要，那么可以在查询时直接写成select 1

子查询和关联查询

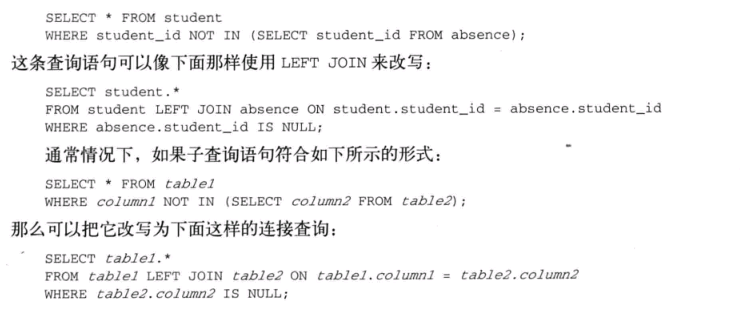
1. 将子查询修改成关联查询

改写用来查询匹配值的子查询





改写用来查询非匹配值的子查询



执行子查询时，MYSQL需要创建临时表，查询完毕后再删除这些临时表，所以，子查询的速度会受到一定的影响，这里多了一个创建和销毁临时表的过程。

In exists比较；

如果子查询和外部查询分别返回M和N行，那么该子查询被扫描为O(N+N\*M)，而不是O(N+M)；注意到慢的原因就是内部每次与外部比较时，都需要遍历一次表操作，可以采用另外一个方法，在嵌套一层子查询，避免多次遍历操作，语句如下：

select count(uid) from user where uid in (SELECT driver\_id FROM driver);

SELECT count(driver\_id) FROM driver where exists (SELECT uid FROM (SELECT uid from user) as b where b.uid = driver.driver\_id);

**通过预查询来减少遍历操作，而提高效率**

## Union

把多个查询结果合并为一个结果集，可以使用union。

编写多条select语句，将union放在它们之间，每条select语句检索的列数要相等，结果集的列名是由第一个sql决定的，各个sql对应的列名不一定相同，但是数据类型要相同；

在默认情况下union是会删除重复的行，如果要保留重复的行可以使用union all

使用order by 和 limit

如果要将union的结果集整体进行排序，那么就需要用括号把每个select语句扩起来，并在最后一个select语句后面加上order by语句

(select …)union (select..) union (select…) order by

要现在返回的行数类似的也可以在最后加上limit 语句

(select …)union (select..) union (select…) limit 2

# 删除更新

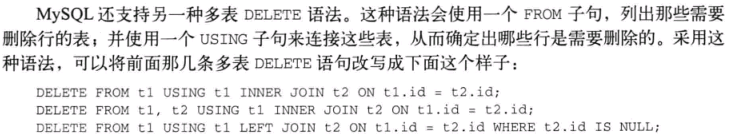
单表 操作

Delete from table [where …..],删除单表指定的数据

把关联的数据都删除（多表）

Delete t1,t2 from t1 inner join t1.c1=t2.c2;





更新操作类型

如果要对InnoDB的表进行多表删除或者更新，可以在表之间建立外键关系，使用外键约束条件on delete caseade或者 on update caseade

# Insert

一次性指定全部列值

Insert table\_name values(……);

指定添加的列

Insert table\_name (…)values(……);

一次添加多列

Insert table\_name values (…), (…), (…)

Insert table\_name () values (…), (…), (…)

主键，外键

# Mysql数据类型

Mysql支持的多种类别的数据值，有数值，字符串值，日期/时间，null值

数值：

mysql能够识别的数字包括整数，定点数或浮点数（可以有小数部分），已经位域值

整数类型：tinyint,smallint,mediuint,int,bigint,分别使用8,16,24,32,64，N是存储空间位数

整数类型有属性unsigned，表示不允许有负值，会让可以使用的正数上限扩大一倍

实数类型：实数是带有小数的数字；实数类型不止可以用来存储小数，也可以使用decimal来存储比bigint大的整数；mysql执行精确的实数类型，也支持不精确的实数类型

Float和double类型使用标准的浮点运算进行近似计算

Decimal类型用于存储精确的小数，支持精确计算

字符串值：字符串两端的引号可以是单引号也可以是双引号，在sql标准规范中规定使用单引号，使用单引号能够更好的移植到其他数据的数据库引擎

字符串值分为2种类型：二进制串和分二进制串

二进制串是一组字节序列，这些字节序列没有字符集，也没有排序的属性。比较操作是基于各个字节的数值逐个字节实现的，所有字节都有含义的

非二进制串是一个字符序列，每个字符串都和字符集相关，字符集决定了：哪些字符可用，字符集都有一种或多种排序规则，排序规则决定了字符在字符集里的先后顺序，会对比较产生影响

Varchar:用于存储可变长的字符串，是最常见的字符串类型。它比定长更节省空间，因为它仅使用必要的空间（越少的字符串使用越少的空间）；varchar需要1个或者2个额外的字节来记录字符串的长度，如果列的最大长度小于或等于255字节，则只需要1个字节表示，否则要使用2个字节；

Varchar其尾部的空格在存储和查询时都会被保留

下列这些情况适合用varchar:字符串列的最大长度比平均长度大很多;列的更新比较少，所以碎片不是问题。

Char:定长字符串类型，根据定义的长度分配空间；

从char列查询出来的值，其尾部空格会被移除，对于char列，如果存入的字符长度不到char设置的长度，那么在末尾会用空格补齐，查询取出数据会将空格移除；定义char时，最大长度为255，定义时长度是可选的，如果省略那么默认值就是1，char(0)也是和法的，这里只能为null或者空串，char(0)只占一个二进制位

Char适合存储很短的字符串，所有值都接近同一个长度；对于经常变更的数据，char也比varchar合适，定长的char不容易产生碎片，

Blob和text

Blob:对应的类型有tinyblob,smallblob.blob,mediublob,longblob,用来存储二进制数据，没有排序规则和字符集，可以存放4GB的数据

Text :对应的类型有tinytext,smallblob,text ,mediublob,longtext，用来存储字符串数据，有排序规则和字符集

这2个都是用来设计存储大数据的字符串数据类型，分别采用二进制和字符存储的方式；与其他类型不同的是mysql会将这2个数据类型当作单独的对象来处理，存储引擎在存储的时候会做特殊处理，

Blob和text能否使用索引，具体取决于所使用的存储引擎

InnoDB和myisam都支持对blob和text列进行索引，但是必须知道索引的一个前缀长度，避免创建出过于庞大的索引，

Memory不支持blob和text

Enum和set

Enum:枚举把不重复的字符串存储成一个预定义的集合。Mysql在存储枚举时非常紧凑，会根据列表值的数量压缩一个或者2个字节；mysql在内部会将每个值在列表中的位置保存为整数，并在。Frm文件中保存‘数字-字符串’映射关系的查询表（如果枚举存储整数，这种映射关系会导致查询结果混乱，尽量避免这样做）；enum的最长允许65535个成员，但是只能是单选的

Set:和enum类似，但是允许出现重复值，最多允许64个成员，允许选择多个选项

字符串属性

字符串数据类型的属性：character set(charset)和collate,分别用于设置字符集和排序规则

这些属性适合的字符类型有char varchar set enum,text,不是用于二进制类型

使用的字符集必须输mysql服务器支持的。Show character set 查看当前可用的字符集

日期时间类型

Mysql支持的时间类型：date time,datetime,timestamp和year，

Date用来保存日期值

Time用来保存时间值

Datetime用来保存日期和时间的组合

Datetime,从1001到9999，精度为秒，把日期和 时间封装到格式为YYYYMMDDHHMMSS的整数中，与时区无关，使用8个字节的存储空间

Timestamp:用于存储日期和时间的组合值，使用4个字节的存储空间来保存数据，范围你从1970到2038年，timestamp的值依赖于时区。Timestamp默认为not null,如果插入数据时没指定第一个timestamp类的值，mysql则设置当前的时间为这个字段的数据，更新也会更新这个列的值。尽量使用timestamp，因为它的空间效率比较高

Null值：用来表示无值，未知值，缺失值，在sql中写null的时候不用加引号

# 事务

什么时候需要事务

当业务流程中涉及到多表操作，希望它们要么全部操作成功要么全部失败；

事务：就是一组原子性的sql执行操作，它是一个独立执行单位；事务内的语句要么全部执行成功，要么全部执行失败;

事务是数据库管理系统（DBMS）执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。这里面有两个关键点，第一个，它是数据库最小的工作单元，是不可以再分的。第二个，它可能包含了一个或者一系列的 DML 语句，包括 insert delete update。（单条 DDL（create drop）和 DCL（grant revoke）也会有事务）

## 事务的特性

对数据库数据的一系列操作，要么成功要么失败，不会出现部分成功部分失败的情况

Atomic原子性:事务必须是一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，不可能只执行其中的一部分操作，这就是事务原子性操作

Consistency一致性：事务的执行不能破坏数据库数据的完整性和一致性，一个事务执行前后数据库必须处于一致性状态。数据库从一个一致性状态装换为另一个一致性状态

Isolate隔离性：一个事务所做的修改在最终提交之前，对其他事务是不可见的

Durability持久性：一旦事务提交所做的修改就会永久保存到数据库中

原子性 ，隔离性，持久性都是为了实现一致性；

隔离性：在sql标准规范中，定义了4个隔离级别，不同的隔离级别对事务的处理不同

1. read uncommitted:读未提交，（未授权读取），其隔离级别最低；该隔离级别允许脏读；脏读：在某个事务所做的修改尚未提交时，其他事务就能看到这些修改的数据；
2. read commited:读已提交，（授权读取），允许事务看到其他事务已提交的行修改；不可重复读：同一个事务使用同一个select语句读取时得到的结果都不一样，前后读取的数据不是重复的
3. repeatable read:重复读取。如果某个事务两次执行同一条select语句，其结果是可重复的，也就是说即时有其他事务在同时插入数据行，这个事务看到的结果都是一样的；幻读：一个事务突然看到之前没有见过的行，一个事务在刚执行select语句后，接着就有另一个事务插入新的行，当第一个事务再次执行相同select查询时，则可能会看到新增的行，而他其实就是一个幻影行
4. serializable:串行化执行，只允许在同一时刻只有一个事务在执行，也就是事务只能一个接着一个的执行，不能并发执行；隔离级别最高

事务隔离性带来的问题

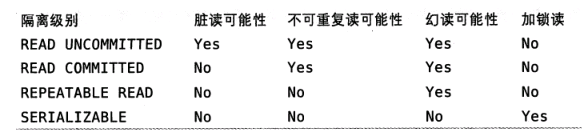
脏读：读取到其他事务还没有提交的数据

不可重复读：一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况

幻读：一个事务前后两次读取数据数据不一致，是由于其他事务插入数据造成的，这种情况我们把它叫做幻读。

不可重复读和幻读，的区别在那里呢？

不可重复读是修改或者删除，幻读是插入。



不可重复读：是发生修改或者删除

幻读：插入

## mysql中数据库事务处理事务方式

自动提交：

在mysql中有一个autocommit参数，默认值是turn/on,表示操作数据库时，会自动开启一个事务和自动提交事务

如果值设置为false/off时，那么就需要手动去设置事务手动的开启和结束

开启事务：1，begin,2,start transaction

提交事务：1，commit 提交事务；2，rollback 回滚事务

事务隔离带来的问题导致数据不一致

## mysql保证数据一致性的方案

1.基于锁的并发控制（LBCC）：锁定要修改的数据，不允许其他事务修改；使用锁来控制事务的隔离，那么就不能并发访问，在写比较多时，

会极大的影响数据库效率

1. 多版本并发控制(MVCC)：MVCC的实现，通过保存数据在某个时间点的快照来实现的。这意味着一个事务无论运行多长时间，在同一个事务里能够看到数据一致的视图。根据事务开始的时间不同，同时也意味着在同一个时刻不同事务看到的相同表里的数据可能是不同的。

基本特征

• 每行数据都存在一个版本，每次数据更新时都更新该版本。

• 修改时Copy出当前版本随意修改，各个事务之间无干扰。

• 保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）

InnoDB存储引擎MVCC的实现策略

在每一行数据中额外保存两个隐藏的列：当前行创建时的版本号和删除时的版本号（可能为空，其实还有一列称为回滚指针，用于事务回滚，不在本文范畴）。这里的版本号并不是实际的时间值，而是系统版本号。每开始新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询每行记录的版本号进行比较。每个事务又有自己的版本号，这样事务内执行CRUD操作时，就通过版本号的比较来达到数据版本控制的目的。

1、插入数据（insert）:记录的版本号即当前事务的版本号

2、在更新操作的时候，采用的是先标记旧的那行记录为已删除，并且删除版本号是事务版本号，然后插入一行新的记录的方式。

3、删除操作的时候，就把事务版本号作为删除版本号。

4、查询操作：

从上面的描述可以看到，在查询时要符合以下两个条件的记录才能被事务查询出来：

1) 删除版本号未指定或者大于当前事务版本号，即查询事务开启后确保读取的行未被删除。(即上述事务id为2的事务查询时，依然能读取到事务id为3所删除的数据行)

2) 创建版本号 小于或者等于 当前事务版本号 ，就是说记录创建是在当前事务中（等于的情况）或者在当前事务启动之前的其他事物进行的insert。

（即事务id为2的事务只能读取到create version<=2的已提交的事务的数据集）

MVCC只在read committed和repeatable committed两种模式下工作，read uncommitted读取的是最新的数据，不是符合当前版本的数据行，serializable

是加锁的串行访问

# 索引

## 索引基本概念

什么是索引

索引是存储引擎中的一种用于快速查找记录的数据结构，

索引的优点：

加快查询速度,减少服务器扫描的数据量

帮助服务器避免排序和临时表

索引将随机IO变为顺序IO

缺点：降低了索引列插入、删除和更新的速度，索引降低了与写相关的操作的速度，因为写一行数据还有更改索引；索引也会占用磁盘空间

Mysql中索引的分类

从逻辑角度

1. 唯一索引
2. 常规索引（普通索引）
3. Spatial索引（空间索引）
4. 主键索引
5. 复合索引

从数据结构角度

1. B+树索引
2. Hash索引
3. Full text索引
4. R-tree索引

从物理存储角度

1. 聚集索引
2. 非聚集索引

## 索引的数据结构（B+Tree）

1.二分查找：也可以叫折半查找，是一种高效的查找方法，折半要求数据必须是线性顺序存储的

假设表中元素按升序，将表中间记录的关键字与查找关键字比较，如果两者相等，则查找成功，如果不相等，则利用中间关键字

将表分为2份，判断查询关键字和中间关键字的大小来判断取左边表还是右边表，重复以上步骤知道找到结果

数据经过排序后来查找效率会比较高

选择索引的存储结构

数组：有序数组的等值查询和比较查询的效率非常高，但是更新时会非常麻烦，可能需要大量的移动数据，所有只适合做静态的数据；

链表：支持频繁的修改，比如插入数据，修改数据。链表的话单链表查询效率还是不够高，需要双向链表

所以就有了二叉查找树

2.二叉查找树

定义：1.如果左子树不为空，那么左子树节点上的值均小于它根节点上的值

2.若右子树不为空，那么右子树节点上的值均大于它根节点上的值

3.左右分为为二叉排序树

4.没有键值相等的节点

查找步骤：如果根节点的关键字等于查找关键字，那么就查找成功

如果根节点的关键字大于查找关键字，那么就遍历左子树

如果根节点的关键字小于查找关键字，那么就遍历右子树

若子树为空那么查找就不成功

二叉查找树能够实现快速查找，又能实现快速插入

但是二叉查找树存在一个问题：查找耗时和这个树的深度有关，最坏的情况下时间复杂度为o(n);

最坏情况就是整个结构变为了链表，那么就不能实现快速检索的目的，和顺序查找效率是没有区别

这种情况下树结构就是一个线性链表；造成这个的原因就是左右子树深度相差太大，也即是不平衡

那么就需要让左右子树平衡，那么就有了平衡二叉查找树（b tree）

3.平衡二叉查找树（B tree,或者叫avl树）

定义：左右子树深度不能超过1；比如左子树的深度为2，那么右子树的深度只能为1或者3

为了满足btree的定义，需要通过左旋和右旋来控制深度

所以为了保持平衡，在添加和删除b树的时候，需要进行一系列的计算和调整操作（判断深度，进行左旋和右旋）

在b tree一个节点的大小是固定的，它存储的内容

索引键值

数据的磁盘地址

左右子树的引用

4.B+树（多路平衡查找树）

使用b tree存在的问题

InnoDB存储引擎操作磁盘最小的单位是一页，大小是16k,那么树节点的大小就是16k的大小；如果使用一个键值，数据，引用，那么远远达不到16k容量，访问

一个树节点，进行一次IO的时候浪费大量的空间；每个节点存储的数据太少，从索引中找到需要的数据，就需要访问更多的节点，那么和磁盘交互的次数就会

过多

解决方案

让每个节点存放更多的数据

节点存放更多的指针，也就意味着可以有更多的分叉，分叉数越多树的深度就会越少

这样树就不是二叉了，而是多叉，也就是多路

mysql中使用b+tree来作为存储结构

1.节点关键字的数量和路数相等

2.B+tree的根节点和枝节点中都不会存储数据，只有叶子节点才存储数据，搜索到关键字不会直接返回，而是会到最后一层的叶子节点

3.B+tree的叶子节点增加了指向相邻叶子节点的指针，它的最后一个数据会指向下一个叶子节点的第一个数据，形成有序链表结构

InnoDB中b+tree的特点

1.每个节点存储更多的关键字，路数更多

2.扫库，扫表能力更强

3.读写能力更强

4.排序能力更强（叶子节点上有下一个数据区的指针，数据形成了链表）

5.效率更加稳定（B+tree是从叶子节点上拿到数据，所以IO次数是稳定的）

假设索引字段是 bigint 类型，长度为 8 字节。指针大小在 InnoDB 源码中设置为

6 字节，这样一共 14 字节。非叶子节点（一页）可以存储 16384/14=1170 个这样的

单元（键值+指针），代表有 1170个指针。树深度为2的时候,有1170^2个叶子节点，可以存储的数据为1170\*1170\*16=21902400;

在查找数据时一次页的查找代表一次 IO，也就是说，一张 2000 万左右的表，查询数据最多需要访问 3 次磁盘。

所以在 InnoDB 中 B+ 树深度一般为 1-3 层，它就能满足千万级的数据存储。

InnoDB存储结构分为5级：表空间，段，簇（区），页，行

表空间：存储引擎逻辑的最高层，所有数据都放在表空间，分为：系统表空间，独占表空间，通过表空间，临时表空间

段：表空间由各个段组成，常用的段有数据段，索引段，回滚段

簇（又叫区），一个短由很多个区组成，每个区的大小为1M(64个连续的页)

页：对物理空间继续划分，簇是由连续的页组成的空间，一个簇中有连续的64个页组成；

InnoDB也有也的概念，页是InnoDB存储引擎磁盘管理的最小单位

，通过Innodb\_page\_size设置，一个表空间最多拥有2-32个页，默认情况下一个页的大小为16kb,一个表空间最多存储64TB的数据

SHOW VARIABLES LIKE 'innodb\_page\_size';

行：存储引擎时面向行的，也就是说数据的按行进行存放

mysql的数据是以文件的形式存放在磁盘中，可以找到这个数据文件的目录地址，show variables like 'datadir';

InnoDB的表有2个文件:frm和idb

Myisam的表有3个文件:frm、myd、myi

frm定义了mysql里面表的结构，不管建表时选择的是什么存储引擎，都会创建这个文件

Myisam存储引擎

处理frm,那么还有另外2个文件

myd:存放数据，

myi:存放索引

在myisam中数据护额索引是放在2个文件中

myisam中的b+tree里面，叶子节点存储的是数据文件对应的磁盘地址，所以是先从索引文件中找到数据对于的磁盘地址

然后从数据文件中找到数据

主键索引和辅助索引都是一样的，都是叶子阶段存放数据的磁盘地址

InnoDB存储引擎

idb:是InnoDB存储引擎的存放数据和索引的文件

在InnoDB中，索引是以主键为索引来组织的数据存储，所以索引和数据都是放在同一个文件中，都是在idb中；

主键索引的叶子节点存放数据

辅助索引的叶子节点存放的是主键值

使用辅助索引时先从辅助索引中找到对应的主键，然后从主键索引中找到对应的数据

聚集索引（聚簇索引）

聚集索引并不是一个索引类型，而是数据的存储方式；数据行实际上是存放在索引的叶子节点上，叶子页上的（leaf page），叶子页包含了行的全部数据，节点页包含了索引列的值

聚簇表示的是数据行和相邻的键值紧凑地存储在一起；因为无法将数据行存放在不同的地方，所有一张表就只能有一个聚簇索引；因为存储引擎实现索引

因此不是所有的存储引擎都支持索引

InnoDB支持聚簇索引，主键索引是聚簇索引，非主键索引是非聚簇索引

辅助索引存储的是辅助索引和主键，如果使用辅助索引，会根据主键值在主键索引中查询取得最终值

为什么在辅助索引里面存储的是主键值而不是主键的磁盘地址呢？如果主键的数据

类型比较大，是不是比存地址更消耗空间呢？

我们前面说到 B Tree 是怎么实现一个节点存储多个关键字，还保持平衡的呢？

是因为有分叉和合并的操作，这个时候键值的地址会发生变化，所以在辅助索引里

面不能存储地址。

二级索引（非聚集索引）叶子节点保存的不是指向行的物理位置的指针，而是行的主键值；这就意味着二级索引查找行，存储引擎需要找到二级索引的叶子节点获得对应的主键值，然后去聚集索引中查找对应的行，这里就进行了2次索引扫描；对于InnoDB，自适应哈希索引能够减少这样的重复工作。

Myisam的数据分布：

Myisam按数据的插入顺序存储在磁盘上，在数据行的旁边显示行号，从0开始递增，因为是行定长，所以myisam可以从表的开头跳过所需要的字节找到需要的行

这种方式很容易创建索引，索引叶子节点包含行号

Myisam的主键索引和其他索引在结构上没有什么不同的，主键索引就是一个名为primary的唯一非空索引

InnoDB的数据分布：

聚集索引的每个叶子节点都包含了主键值，输完id,用于事务和MVCC的回滚指针以及所有的列

在InnoDB中二级索引和聚集索引很不同，二级索引的叶子页存储的不是指针，而是主键值，并以此作为指向行的指针，这种策略减少了当出现行移动或者数据页分裂时二级索引的维护；使用主键值作为指针会让二级索引占用更多的空间，换来的好处就是，InnoDB在移动行时无需更新二级索引中的指针

在InnoDB中聚簇索引选择主键为聚集索引，最简单的方法就是使用auto\_increment自增列，这样可以保证插入数据行是按数据顺序写入的，对于主键的关联操作的性能也很好，最好避免随机的（不连续且值的分布范围非常大）聚簇索引，特别是IO密集型的应用，

例如使用uuid来作为聚簇索引效果非常糟糕；它使得聚簇索引的插入变的完全随机，这是最坏的情况，使得数据没有任何的聚集特性，使用uuid不仅花费更长的时间，而且索引占用的空间也更大；一方面是因为主键字段更长，另一方面是由于页分裂和锁片造成的；InnoDB无法简单的总是把新行插入到索引的最后，而是需要找到合适的位置，通常是已有数据的中间

位置，并且分配空间。这会增加额外的工作，并且导致数据分布不够优化

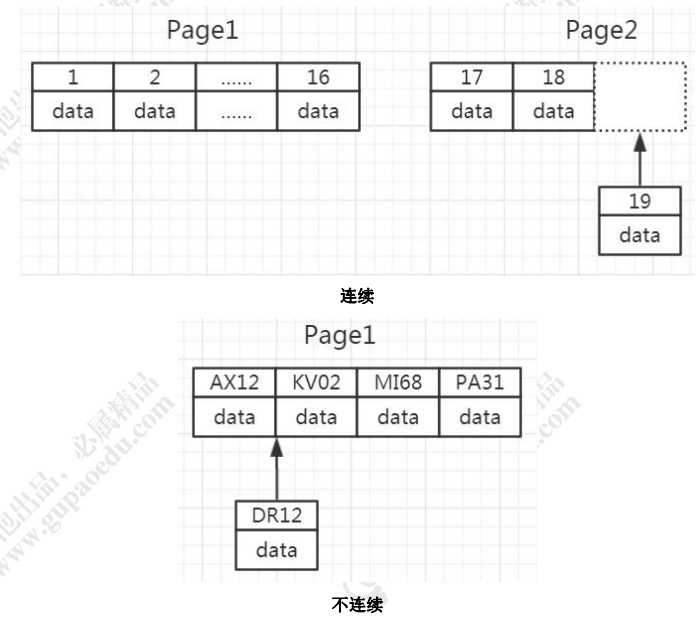
1.InnoDB不得不先找到目标也，并且读取目标页的数据到内存中，这将会导致大量的IO操作

2.因为写入是乱序的，InnoDB不得不频繁的左分页操作，以便分配新空间，页分裂会导致数据行大量移动，

3.由于也分裂，页会变得非常稀松，最终数据会有碎片

往表中插入数据时，如果一个页面已经写完，产生一个新的叶页面。如果一个簇的所有的页面都被用完，会从当前页面所在段新分配一个簇。

如果数据不是连续的，往已经写满的页中插入数据，会导致叶页面分裂：



当主键是顺序值时，InnoDB会把新添加的数据存储在上一条记录的后面，当达到页的最大因子时，下一个记录就会写入新的页中，数据按照这种方式加载，主键页就会近似于将顺序的记录填满，这也是期望的结果

另一个问题，如果一张表没有主键怎么办？

1、如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。

2、如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引

作为主键索引。

3、如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐

藏的聚集索引，它会随着行记录的写入而主键递增。

聚集索引的优点

数据访问更快

可以把相关数据保存在一起，根据主键来聚集数据，这样只需要从磁盘中读取少数据的数据页就可以获取全部数据

使用覆盖索引扫描的查询可以直接使用页节点中的主键

聚集缺点

插入速度严重依赖于插入顺序，

更新聚簇索引的表在插入新行，或者主键被更新导致需要移动行时可能面临页分裂的问题

HASH索引

## 索引的使用

正确的创建和使用索引是实现高性能查询的基础

独立的列指的是索引列不能是表达式的一部分，或者不是函数的参数；

例如

Select \* from actor where actor\_id+1=5;这个sql中无法使用actor\_id的索引，这个sql中where表达式等价于actor\_id=4,但是mysql无法自动解析这个方程式

另一种粗放方式就是使用函数

Select … where to\_days(date)<10

**前缀索引和索引选择**

当索引很长的数据列时，这会让索引变得很大且慢；

解决方法

索引数据开始的部分数据，这样可以大大节省索引空间，从而提高索引效率；同时这样也降低了索引的选择性

索引的选择性（又叫列的离散度，列重复值的多少）是指：不重复的索引值数量和数据总数的比值，索引的选择性越高索引效率越高，选择性高的索引可以让mysql在查找时过来掉更多的行；count(distinct(column\_name)) : count(\*)，列的全部不同值和所有数据行的比例。

简单来说，**如果列的重复值越多，离散度就越低，重复值越少，离散度就越高**。

唯一索引的选择性为1，这是最好的索引选择性，性能也是最好的（也就是说没有重复的行）；

当某个列前缀的选择性足够高的，足以满足查询性能

对应blob，text，或者很长的varchar类型的列， 必须使用前缀索引

选择做过长的前缀以保证较高的选择性，同时不能太长，前缀应该足够长，以使前缀索引的选择性更加接近整个列；前缀的基数应该接近于完整列的基数

基数：指一列中非重复值的个数（例如：1,3,4,7,7,3，那么基数就为4）

当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。我们可

以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。

前缀索引是一个使索引更小的，更有效的方法，但是也有缺点，无法使用前缀索引做order by和group by，也无法使用前缀索引做覆盖索引

**多列索引**

最左前缀

使用多个字段创建索引，在使用这个索引时，必须要按顺序从第一个字段开始使用，

单列索引

早期版本的mysql只能使用其中某一个单列索引，然而这种情况下没有哪一个独立的单列索引是非常有效的

比如film\_actor,有2个字段film\_id,film\_name，各有一个单列的索引，

Select \* from film\_actor where file\_id=2 or actor\_id =’tom’;

使用or时，老版本是不会使用索引的，而是会进行全盘扫描，除非改成union

Select \* from film\_actor where file\_id=2

Union

Select \* from film\_actor where actor\_id =’tom’ and file\_id<>2;

在mysql5.0之后查询能够使用两个单列索引进行扫描，并将其合并：or条件的联合，and条件的相交，组合前2中情况的联合及相交

**选择索引列顺序**

在一个多列的B tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列。。。索引可以按照升序或者降序进行扫描，以满足精确符号顺序的order by/group by/distinct，所以索引的列顺序至关重要

如何选择 索引列的顺序有一个经验：将选择性最高的列放到索引最前列；当不需要考虑排序和分组时，将选择性最高的列放在前面通常是最好的，这时索引的作用只是用来优化where条件查询

查询性能不只是依赖于所有索引列的选择性，也和查询条件的具体值有关，也和值的分布有关（当索引列的值具有顺序性，生成的索引效果是最好的）

**覆盖索引**

Mysql可以使用索引来直接获取列的数据，这样就不需要再回表查询；如果一个索引包含（覆盖）了查询返回的字段，那么这个索引就可以被称为覆盖索引；覆盖索引是非常有用的工具，能够极大的提升性能；考虑索引无需回表查询

覆盖索引的好处

1.索引列通常比数据行列少，所以只需要读取索引，那么mysql就会极大的减少数据的访问量

2.索引是按顺序存储的（至少在单个页中），对IO密集型的范围查询会比随机从磁盘读取每一行数据的IO要少的多（读取的数据更少）

3.在InnoDB中二级索引在叶子节点中保存了行的主键值，所以如果二级主键能够覆盖查询，则可以避免堆主键索引的二次查询

不是所有类的索引都是覆盖索引，覆盖索引必须要存储索引列的值；哈希索引，空间索引，全文索引等都不存储索引列的值，所以mysql中只能使用b+tree索引作为覆盖索引

大多数存储引擎中，覆盖索引只能覆盖那些只访问索引中部分列的查询

**使用索引扫描来做排序**

Mysql有两种排序：通过排序操作，或者按索引顺序扫描

Mysql可以使用索引来既满足排序，有用于查找行。所以设计索引时应尽可能的同时满足这两种任务，这是最好的

只有当索引的列顺序和order by子句的顺序完全一致，并所有列的排序方向都一样时，mysql才能够使用索引来对结果做排序，如果查询需要关联多种表，则只有当orderby的字段全部为第一个表时，才能够使用索引做排序；

**压缩（前缀压缩）索引**

Myisam使用前缀压缩来减少索引的大小，从而让更多的索引可以放入内存，极大的提示性能

Myisam压缩每个索引块的方法是：先保存索引 的第一个值，然后将其他的值和第一个值进行比较得到相同的字节数和剩余的不同部分，把这部分存储起来即可

压缩块使用空间更少，代价是某些操作可能很慢

**冗余和重复索引**

Mysql需要单独的维护重复的索引，而且优化器在优化查询时也需要逐个的进行考虑，这会影响性能

重复索引是指在相同列，相同顺序创建相同类型的索引；应避免创建重复索引；

冗余索引：当表创建了索引（A,B）,然后又创建了索引（A），这个索引（A）就是冗余索引，A只是一个前缀索引；此时索引（A,B）可以当做索引（A）和索引（A,B）来使用；如果创建了索引（B,A），那么这个就不是冗余索引

大多数情况下应该避免冗余索引，应该尽量扩展已有的索引而不是创建新索引；但是有时和使用冗余索引可以提升性能，因为扩展已有的索引会导致其变得太大，从而影响到其他的查询性能

Useinfo表有索引（state\_id）

查询数量



查询几个列的值



提升这个查询的办法是扩展索引（state\_id,city,address）,让覆盖索引查询，但是这样第一个查询就变满了

索引就需要一个单独列的所有（state\_id）

索引和锁

索引可以让查询锁定更少的行，那么就会锁定更少的行，减少锁定行带来的开销，锁定超过需要的行会增加锁竞争从而减少并发性

InnoDB只有访问行时才会对其锁定，而索引能够减少InnoDB访问的行数，从而减少锁的数量

因为索引对于改善查询性能的作用是巨大的，所以我们的目标是尽量使用索引。

## 索引的创建

1、在用于 where 判断 order 排序和 join 的（on）字段上创建索引

2、索引的个数不要过多

——浪费空间，更新变慢。

3、区分度低的字段，例如性别，不要建索引。

——离散度太低，导致扫描行数过多。

4、频繁更新的值，不要作为主键或者索引。

——页分裂

5、组合索引把散列性高（区分度高）的值放在前面。

6、创建复合索引，而不是修改单列索引。

7、过长的字段，怎么建立索引？

--fulltext全文索引 、前缀索引、hash索引等

8、为什么不建议用无序的值（例如身份证、UUID ）作为索引？

--无序会产生页分裂、递增的ID离散度较高，不会重复；

5.2. 什么时候用不到索引？

**1、索引列上使用函数（replace\SUBSTR\CONCAT\sum count avg）、表达式、**

**计算（+ - \* /）：**

explain SELECT \* FROM `t2` where id+1 = 4;

**2、字符串不加引号，出现隐式转换**

ALTER TABLE user\_innodb DROP INDEX comidx\_name\_phone;

ALTER TABLE user\_innodb add INDEX comidx\_name\_phone (name,phone);

explain SELECT \* FROM `user\_innodb` where name = 136;

explain SELECT \* FROM `user\_innodb` where name = '136';

**3、like 条件中前面带%**

where 条件中 like abc%，like %2673%，like %888 都用不到索引吗？为什么？

explain select \*from user\_innodb where name like 'wang%';

explain select \*from user\_innodb where name like '%wang';

过滤的开销太大，所以无法使用索引。这个时候可以用全文索引

**4、负向查询**

NOT LIKE 不能：

explain select \*from employees where last\_name not like 'wang'

!= （<>）和 NOT IN 在某些情况下可以：

explain select \*from employees where emp\_no not in (1)

explain select \*from employees where emp\_no <> 1

# Mysql中的锁

InnoDB存储引擎的锁粒度

行级：锁定数据行

表级：锁定整张表

表锁的粒度大于行锁

锁效率

表锁的加速效率更高，它锁定整个表就可以了，行锁还需要检索出数据行；表锁的冲突概率更大，并发效率更低

行级别锁

1.共享锁：获取一行数据的读锁（共享锁）后，用来读取数据，所以也叫作读锁；多个事务可以共享一把读锁；

在获取读锁的时候不要去获取写锁，这样会容易造成死锁

给一行数据加读锁：select ... lock in share mode;在查询后面加上lock in share mode

释放锁有两种方式：只要事务结束，锁就会自动释放，包括事务提交和回滚

2.排他锁：用来操作数据，又叫写锁，只要一个事务获取一行数据的排他锁，那么其他的事务就不能获取这会数据的共享锁和排他锁

排他锁有2种加锁方式：

自动加排他锁

手工加锁：在sql语句后面加上for update给一行数据加上一个排他锁

释放锁方式和行锁一样，

表锁

意向锁：这是由数据库自己维护的，意向共享锁和意向排他锁

在给数据加上共享锁之前，数据库会自动在这张表上面加上一个意向共享锁

给数据加上排他锁之前，数据库会自动在这张表上加上一个意向排他锁

表锁的作用

有了表级别的锁，在InnoDB里面支持更多粒度的锁

如果没有意向锁，那么在加锁的时候就需要去判断是否被某些事务加锁锁定了某些行，如果是不能加表锁的；

那么这个时候就需要去扫描整张表才能确定是否加表锁，如果数据量特别大（比如上千万级别的），那么加锁的效率就非常低；引入影响锁后

只需要判断表上面有没有意向锁，就能决定是否能够加锁（如果有就直接返回，如果没有就能够获取锁）；

所以InnoDB里面的表锁，可以看作是一个标志，用来决定是否加锁，从而提升效率

锁的作用：解决资源竞争；java中是对象，数据库中是数据表或者数据行；数据库锁使用来解决事务对数据的并发访问的问题

行锁的原理

锁定的内容（InnoDB的行锁，就是通过锁做索引来实现的）

mysql中表是一定会有索引的（及时没有显示的创建）

如果定义了主键，那么InnoDB会选择主键作为聚集索引

如果没有显示的定义主键，InnoDB会选择不包含有null的唯一索引作为主键索引

如果没有唯一索引，InnoDB会选择表内置的列rowid来作为隐藏的聚集索引，它会随着行记录的写入而递增

也就是说表一定会存在聚集索引

如果没有使用索引，那么会进行全表扫描，然后会把每个隐藏的聚集索引都锁住

为什么通过索引给数据行加锁，主键索引也会被锁住

辅助索引中，在叶节点中存储的是二级索引和主键值，主键索引叶节点存储的除了索引之外还存储了完整的数据；

通过辅助索引锁定一行数据的时候，它会根据检索到的主键值去寻找数据，然后也就锁定主键索引

记录record：数据库中存在的主键值

间隙Gap：record隔开的不存在的区间：左开右开的区间，

临建区间：间隙加上右边的记录，是一个左开右闭的区间

针对锁定的范围

记录锁：对唯一索引使用等值查询，精确匹配到一条记录的时候，这个时候就使用记录锁

间隙锁：查询的记录不存在，无论是等值查询还是范围查询，使用间隙锁

临建锁：在范围查询的时候，如果有查询结果，还包含了Gap间隙，这时就使用临建锁（相当于记录锁加上间隙锁）

临键锁，锁住最后一个 key 的下一个左开右闭的区间。

select \* from t2 where id >5 and id <=7 for update; -- 锁住(4,7]和(7,10]

select \* from t2 where id >8 and id <=10 for update; -- 锁住 (7,10]，(10,+∞)

为什么要锁住下一个左开右闭的区间？——就是为了解决幻读的问题。

事务隔离级别的实现

InnoDB能够解决幻读问题，就是用临建锁实现的，

read uncommitted不加锁，

serializable：所有的select语句都会被隐式的转换为select。。。in share mode,会和update delete互斥

repeatable read（重复读）普通的select使用快照（），底层使用mvcc来实现；

加锁的select以及更新操作update，delete等语句使用当前读，底层使用记录锁，或者间隙锁，临建锁

read committed：普通的select使用快照，使用mvcc实现；加锁的select都使用记录锁，因为没有gap lock(间隙锁)，

除了两种特殊情况——外键约束检查(foreign-key constraint checking)以及重复键检查(duplicate-key checking)时会使用间隙锁封锁区间。

所以 RC 会出现幻读的问题。

RC 和 RR 主要有几个区别：

1、 RR 的间隙锁会导致锁定范围的扩大。

2、 条件列未使用到索引，RR 锁表，RC 锁行。

3、 RC 的“半一致性”（semi-consistent）读可以增加 update 操作的并发性。

死锁的避免

1、 在程序中，操作多张表时，尽量以相同的顺序来访问（避免形成等待环路）；

2、 批量操作单张表数据的时候，先对数据进行排序（避免形成等待环路）；

3、 申请足够级别的锁，如果要操作数据，就申请排它锁；

4、 尽量使用索引访问数据，避免没有 where 条件的操作，避免锁表；

5、 如果可以，大事务化成小事务；

6、 使用等值查询而不是范围查询查询数据，命中记录，避免间隙锁对并发的影响。

在发生死锁时，InnoDB 一般都能通过算法（wait-for graph）自动检测到。

那么死锁需要满足什么条件？死锁的产生条件：

因为锁本身是互斥的，

（1）同一时刻只能有一个事务持有这把锁，

（2）其他的事务需要在这个事务释放锁之后才能获取锁，而不可以强行剥夺，

（3）当多个事务形成等待环路的时候，即发生死锁。

# Mysql分库分表

大数据存储时提升访问性能

超大容量

性能问题（查询耗时，cpu、磁盘io到达极限）

垂直切分

垂直分库：分不同的数据库，业务分布更加清晰，每个领域可以分布在不同的服务器上；解决的表过多的问题

垂直分布：解决的是单表列过多的问题；将列多的表拆分为小表

水平切分：大数据表拆分为小表；比如表数据上亿后性能就会非常差，将其拆分为多个小表（也就是将数据分布到不同的表中）

## 拆分策略

垂直拆分：er分片，（将关联表放到同一个数据库中）

水平拆分：一致性hash

拆分范围：根据日期，根据id

拆分带来的问题

* 1. 跨库join问题，关联查询
     1. 在服务层调用
     2. 全局表（数据变更较少的）
     3. 字段冗余（空间换时间），可以通过定时任务去更新冗余字段，可以通过消息通知更新

跨分片数据排序分页问题（不同数据库的数据进行分页排序）

在应用层拼接

主键唯一性问题（多表的主键唯一性）

解决方案:uuid（值比较大，索引性能低）

Zookeeper节点自增的id

Redis的incream

用一张专门用来存放唯一主键id的表

分布式事务问题

保证多个数据表之间的原子性（存在性能问题）（目前使用数据强一致性比较少）（可以使用最终一致性，软一致性）

如何权衡当前公司的存储需要优化

1. 提前规划（主键问题解决、 join问题）
2. 当前数据单表超过1000W、每天的增长量持续上升

Mysql主从

绝大多数应用是写少读多的

读写分离主从同步

Mysql自带主从复制（一主多从）

主从复制

性能优化

影响性能的因素

人为因素：需求（实时，准实时，有误差）

## 主从同步的原理

|  |  |
| --- | --- |
|  | 1. master记录二进制日志。在每个事务更新数据完成之前，master在二日志记录这些改变。MySQL将事务串行的写入二进制日志，即使事务中的语句都是交叉执行的。在事件写入二进制日志完成后，master通知存储引擎提交事务 2. slave将master的binary log拷贝到它自己的中继日志。首先，slave开始一个工作线程——I/O线程。I/O线程在master上打开一个普通的连接，然后开始binlog dump process。Binlog dump process从master的二进制日志中读取事件，如果已经跟上master，它会睡眠并等待master产生新的事件。I/O线程将这些事件写入中继日志 3. SQL线程从中继日志读取事件，并重放其中的事件而更新slave的数据，使其与master中的数据一致 |

binlog： 用来记录mysql的数据更新或者潜在更新（update xxx where id=x effect row 0）;

文件内容存储：/var/lib/mysql

mysqlbinlog --base64-output=decode-rows -v mysql-bin.000001查看binlog的内容

## binlog的格式

**statement： 基于sql语句的模式。update table set name =””; effect row 1000； uuid、now() other function**

row： 基于行模式;存在1000条数据变更； 记录修改以后每一条记录变化的值

mixed: 混合模式，由mysql自动判断处理

修改binlog\_formater,通过在mysql客户端输入如下命令可以修改

set global binlog\_format=’row/mixed/statement’;

或者在vim /etc/my.cnf的[mysqld]下增加binlog\_format=‘mixed’

主从同步的延时问题

### 主从同步延迟是怎么产生的

1. 当master库tps比较高的时候，产生的DDL数量超过slave一个sql线程所能承受的范围，或者slave的大型query语句产生锁等待
2. 网络传输： bin文件的传输延迟
3. 磁盘的读写耗时：文件通知更新、磁盘读取延迟、磁盘写入延迟

### 解决方案

1. 在数据库和应用层增加缓存处理，优先从缓存中读取数据
2. 减少slave同步延迟，可以修改slave库sync\_binlog属性；

sync\_binlog=0 文件系统来调度把binlog\_cache刷新到磁盘

sync\_binlog=n

1. 增加延时监控

Nagios做网络监控

mk-heartbeat

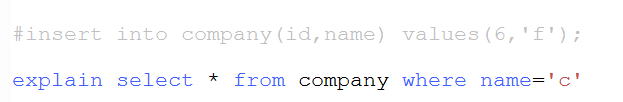
## 实战演示

## 单库大表拆分

配置都是在schema.xml文件中实现的



数据库操作

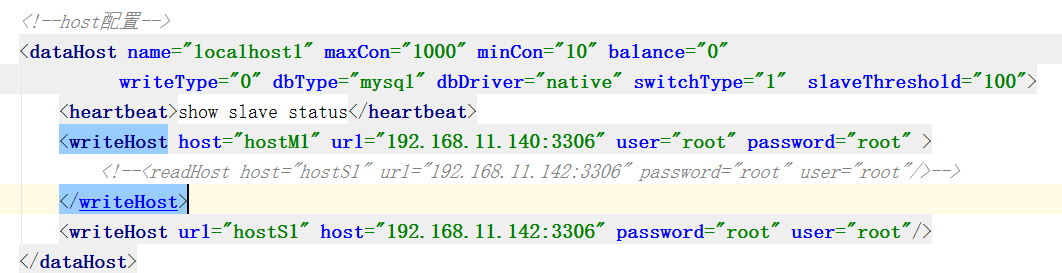


跨库分表



## 读写分离





# mycat

启动mycat服务端，使用navicate使用

核心概念

主要解决水平拆分的问题

2013年阿里的Cobar在社区使用过程中发现存在一些比较严重的问题，及其使用限制，经过Mycat发起人第一次改良，第一代改良版——Mycat诞生。 Mycat开源以后，一些Cobar的用户参与了Mycat的开发，最终Mycat发展成为一个由众多软件公司的实力派架构师和资深开发人员维护的社区型开源软件。

2014年Mycat首次在上海的《中华架构师》大会上对外宣讲，更多的人参与进来，随后越来越多的项目采用了Mycat。

2015年5月，由核心参与者们一起编写的第一本官方权威指南《Mycat权威指南》电子版发布，累计超过500本，成为开源项目中的首创。

2015年10月为止，Mycat项目总共有16个Committer。

截至2015年11月，超过300个项目采用Mycat，涵盖银行、电信、电子商务、物流、移动应用、O2O的众多领域和公司。

截至2015年12月，超过4000名用户加群或研究讨论或测试或使用Mycat。

Mycat是基于开源cobar演变而来，我们对cobar的代码进行了彻底的重构，使用NIO重构了网络模块，并且优化了Buffer内核，增强了聚合，Join等基本特性，同时兼容绝大多数数据库成为通用的数据库中间件。1.4 版本以后 完全的脱离基本cobar内核，结合Mycat集群管理、自动扩容、智能优化，成为高性能的中间件。我们致力于开发高性能数据库中间而努力。永不收费，永不闭源，持续推动开源社区的发展。

Mycat吸引和聚集了一大批业内大数据和云计算方面的资深工程师，Mycat的发展壮大基于开源社区志愿者的持续努力，感谢社区志愿者的努力让Mycat更加强大，同时我们也欢迎社区更多的志愿者，特别是公司能够参与进来，参与Mycat的开发，一起推动社区的发展，为社区提供更好的开源中间件。

Mycat还不够强大，Mycat还有很多不足，欢迎社区志愿者的持续优化改进。

# 关键特性

支持SQL92标准

遵守Mysql原生协议，跨语言，跨平台，跨数据库的通用中间件代理。

基于心跳的自动故障切换，支持读写分离，支持MySQL主从，以及galera cluster集群。

支持Galera for MySQL集群，Percona Cluster或者MariaDB cluster

基于Nio实现，有效管理线程，高并发问题。

支持数据的多片自动路由与聚合，支持sum,count,max等常用的聚合函数。

支持单库内部任意join，支持跨库2表join，甚至基于caltlet的多表join。

支持通过全局表，ER关系的分片策略，实现了高效的多表join查询。

支持多租户方案。

支持分布式事务（弱xa）。

支持全局序列号，解决分布式下的主键生成问题。

分片规则丰富，插件化开发，易于扩展。

强大的web，命令行监控。

支持前端作为mysq通用代理，后端JDBC方式支持Oracle、DB2、SQL Server 、 mongodb 、巨杉。

支持密码加密

支持服务降级

支持IP白名单

支持SQL黑名单、sql注入攻击拦截

支持分表（1.6）

集群基于ZooKeeper管理，在线升级，扩容，智能优化，大数据处理（2.0开发版）。

# Mycat安装与使用

## 下载：

<https://github.com/MyCATApache/Mycat-download> 具体下载哪个版本以发布为准，推荐1.4,1.5.

## 安装：

下载的文件直接解压即可。

## 运行：

### linux：

./mycat start 启动

./mycat stop 停止

./mycat console 前台运行

./mycat install 添加到系统自动启动（暂未实现）

./mycat remove 取消随系统自动启动（暂未实现）

./mycat restart 重启服务

./mycat pause 暂停

./mycat status 查看启动状态

### win：

直接运行startup\_nowrap.bat，如果出现闪退，在cmd 命令行运行，查看出错原因。

## 内存配置：

启动前，一般需要修改JVM配置参数，打开conf/wrapper.conf文件，如下行的内容为2G和2048，可根据本机配置情况修改为512M或其它值。 以下配置跟jvm参数完全一致，可以根据自己的jvm参数调整。

Java Additional Parameters

wrapper.java.additional.1=

wrapper.java.additional.1=-DMYCAT\_HOME=.

wrapper.java.additional.2=-server

#wrapper.java.additional.3=-XX:MaxPermSize=64M

wrapper.java.additional.4=-XX:+AggressiveOpts

wrapper.java.additional.5=-XX:MaxDirectMemorySize=100m

wrapper.java.additional.6=-Dcom.sun.management.jmxremote

wrapper.java.additional.7=-Dcom.sun.management.jmxremote.port=1984

wrapper.java.additional.8=-Dcom.sun.management.jmxremote.authenticate=false

wrapper.java.additional.9=-Dcom.sun.management.jmxremote.ssl=false

wrapper.java.additional.10=-Xmx100m

wrapper.java.additional.11=-Xms100m

wrapper.java.additional.12=-XX:+UseParNewGC

wrapper.java.additional.13=-XX:+UseConcMarkSweepGC

wrapper.java.additional.14=-XX:+UseCMSCompactAtFullCollection

wrapper.java.additional.15=-XX:CMSFullGCsBeforeCompaction=0

wrapper.java.additional.16=-XX:CMSInitiatingOccupancyFraction=70

以下配置作废：

wrapper.java.initmemory=3

wrapper.java.maxmemory=64

### Mycat连接测试：

测试mycat与测试mysql完全一致，mysql怎么连接，mycat就怎么连接。

推荐先采用命令行测试：

mysql -uroot -proot -P8066 -h127.0.0.1

如果采用工具连接，1.4,1.3目前部分工具无法连接，会提示database not selected，建议采用高版本，navicat测试。1.5已经修复了部分工具连接。

# Mycat配置入门

## 配置：

--bin 启动目录

--conf 配置目录存放配置文件：

--server.xml：是Mycat服务器参数调整和用户授权的配置文件。

--schema.xml：是逻辑库定义和表以及分片定义的配置文件。

--rule.xml： 是分片规则的配置文件，分片规则的具体一些参数信息单独存放为文件，也在这个目录下，配置文件修改需要重启MyCAT。

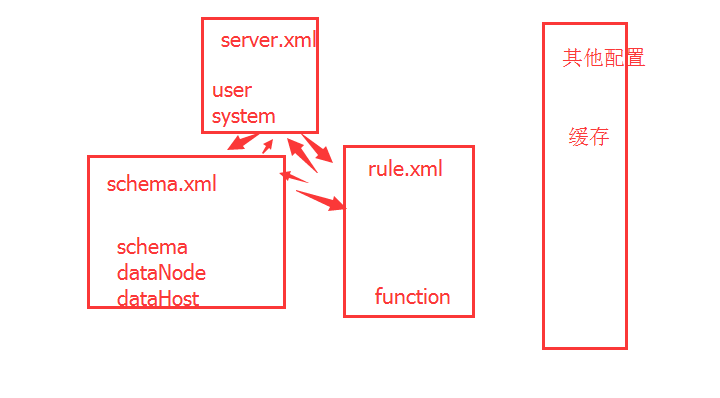
--log4j.xml： 日志存放在logs/log中，每天一个文件，日志的配置是在conf/log4j.xml中，根据自己的需要可以调整输出级别为debug debug级别下，会输出更多的信息，方便排查问题。

--autopartition-long.txt,partition-hash-int.txt,sequence\_conf.properties， sequence\_db\_conf.properties 分片相关的id分片规则配置文件

--lib MyCAT自身的jar包或依赖的jar包的存放目录。

--logs MyCAT日志的存放目录。日志存放在logs/log中，每天一个文件

下面图片描述了Mycat最重要的3大配置文件：

[](https://camo.githubusercontent.com/af0b8c0147e92f5e63e83f1558b39e178fb4d50c/687474703a2f2f736f6e677769652e636f6d2f61747461636865642f696d6167652f32303136303230352f32303136303230353136343535385f3135342e706e67)

## 逻辑库配置：

### 配置server.xml

添加两个mycat逻辑库：user,pay  
system 参数是所有的mycat参数配置，比如添加解析器：defaultSqlParser，其他类推  
user 是用户参数。

<system>

<property name="defaultSqlParser">druidparser</property>

</system>

<user name="mycat">

<property name="password">mycat</property>

<property name="schemas">user,pay</property>

</user>

### 编辑schema.xml

修改dataHost和schema对应的连接信息，user,pay 垂直切分后的配置如下所示：

schema 是实际逻辑库的配置，user，pay分别对应两个逻辑库，多个schema代表多个逻辑库。

dataNode是逻辑库对应的分片，如果配置多个分片只需要多个dataNode即可。

dataHost是实际的物理库配置地址，可以配置多主主从等其他配置，多个dataHost代表分片对应的物理库地址，下面的writeHost、readHost代表该分片是否配置多写，主从，读写分离等高级特性。

以下例子配置了两个writeHost为主从。

<schema name="user" checkSQLschema="false" sqlMaxLimit="100" dataNode="user" />

<schema name="pay" checkSQLschema="false" sqlMaxLimit="100" dataNode="pay" >

<table name="order" dataNode="pay1,pay2" rule="rule1"/>

</schema>

<dataNode name="user" dataHost="host" database="user" />

<dataNode name="pay1" dataHost="host" database="pay1" />

<dataNode name="pay2" dataHost="host" database="pay2" />

<dataHost name="host" maxCon="1000" minCon="10" balance="0"

writeType="0" dbType="mysql" dbDriver="native">

<heartbeat>select 1</heartbeat>

<!-- can have multi write hosts -->

<writeHost host="hostM1" url="192.168.0.2:3306" user="root" password="root" />

<writeHost host="hostM2" url="192.168.0.3:3306" user="root" password="root" />

</dataHost>

​

# Mycat逻辑库、系统参数配置

## 配置Mycat环境参数

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE mycat:server SYSTEM "server.dtd">

<mycat:server xmlns:mycat="http://org.opencloudb/">

<system>

<property name="defaultSqlParser">druidparser</property>

</system>

</mycat:server>

如例子中配置的所有的Mycat参数变量都是配置在server.xml 文件中，system标签下配置所有的参数，如果需要配置某个变量添加相应的配置即可，例如添加启动端口8066，默认为8066：

<property name="serverPort">8066</property>

其他所有变量类似。

## 配置Mycat逻辑库与用户

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE mycat:server SYSTEM "server.dtd">

<mycat:server xmlns:mycat="http://org.opencloudb/">

<user name="mycat">

<property name="password">mycat</property>

<property name="schemas">TESTDB</property>

</user>

</mycat:server>

如例子中配置的所有的Mycat连接的用户与逻辑库映射都是配置在server.xml 文件中，user标签下配置所有的参数，例如例子中配置了一个mycat用户供应用连接到mycat，同时mycat 在schema.xml中配置后了一个逻辑库TESTDB，配置好逻辑库与用户的映射关系。

# 逻辑库、表分片配置

## 配置逻辑库（schema）

Mycat作为一个中间件，实现mysql协议，那么对前端应用连接来说就是一个数据库，也就有数据库的配置，mycat的数据库配置是在schema.xml中配置，配置好后映射到server.xml里面的用户就可以了。

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mycat:schema SYSTEM "schema.dtd">

<mycat:schema xmlns:mycat="http://org.opencloudb/">

<schema name="TESTDB" checkSQLschema="true" sqlMaxLimit="100" dataNode="dn1">

<table name="t\_user" dataNode="dn1,dn2" rule="sharding-by-mod2"/>

<table name="ht\_jy\_login\_log" primaryKey="ID" dataNode="dn1,dn2" rule="sharding-by-date\_jylog"/>

</schema>

<dataNode name="dn1" dataHost="localhost1" database="mycat\_node1"/>

<dataNode name="dn2" dataHost="localhost1" database="mycat\_node2"/>

<dataHost name="localhost1" writeType="0" switchType="1" slaveThreshold="100" balance="1" dbType="mysql" maxCon="10" minCon="1" dbDriver="native">

<heartbeat>show status like 'wsrep%'</heartbeat>

<writeHost host="hostM1" url="127.0.0.1:3306" user="root" password="root" >

</writeHost>

</dataHost>

</mycat:schema >

上面例子配置了一个逻辑库TESTDB，同时配置了t\_user，ht\_jy\_login\_log两个分片表。

### 逻辑表配置

<table name="t\_user" dataNode="dn1,dn2" rule="sharding-by-mod2"/>

table 标签 是逻辑表的配置 其中

name代表表名，

dataNode代表表对应的分片，

Mycat默认采用分库方式，也就是一个表映射到不同的库上，

rule代表表要采用的数据切分方式，名称对应到rule.xml中的对应配置，如果要分片必须配置。

## 配置分片（dataNode）

<dataNode name="dn1" dataHost="localhost1" database="mycat\_node1"/>

<dataNode name="dn2" dataHost="localhost1" database="mycat\_node2"/>

表切分后需要配置映射到哪几个数据库中，Mycat的分片实际上就是库的别名，例如上面例子配置了两个分片dn1，dn2 分别对应到物理机映射dataHost localhost1 的两个库上。

## 配置物理库分片映射（dataHost）

<dataHost name="localhost1" writeType="0" switchType="1" slaveThreshold="100" balance="1" dbType="mysql" maxCon="10" minCon="1" dbDriver="native">

<heartbeat>show status like 'wsrep%'</heartbeat>

<writeHost host="hostM1" url="127.0.0.1:3306" user="root" password="root" >

</writeHost>

</dataHost>

Mycat作为数据库代理需要逻辑库，逻辑用户，表切分后需要配置分片，分片也就需要映射到真实的物理主机上，至于是映射到一台还是一台的多个实例上，Mycat并不关心，只需要配置好映射即可，例如例子中：

配置了一个名为localhost1的物理主机（dataHost）映射。

heartbeat 标签代表Mycat需要对物理库心跳检测的语句，正常情况下生产案例可能配置主从，或者多写 或者单库，无论哪种情况Mycat都需要维持到数据库的数据源连接，因此需要定时检查后端连接可以性，心跳语句就是来作为心跳检测。

writeHost 此标签代表 一个逻辑主机（dataHost）对应的后端的物理主机映射，例如例子中写库hostM1 映射到127.0.0.1:3306。如果后端需要做读写分离或者多写 或者主从则通过配置 多个writeHost 或者readHost即可。

dataHost 标签中的 writeType balance 等标签则是不同的策略，具体参考指南。

# Mycat 表切分规则配置

## 表切分规则

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mycat:rule SYSTEM "rule.dtd">

<mycat:rule xmlns:mycat="http://org.opencloudb/">

<tableRule name="sharding-by-hour">

<rule>

<columns>createTime</columns>

<algorithm>sharding-by-hour</algorithm>

</rule>

</tableRule>

<function name="sharding-by-hour" class="org.opencloudb.route.function.LatestMonthPartion">

<property name="splitOneDay">24</property>

</function>

</mycat:rule >

数据切分中作为表切分规则中最重要的配置，表的切分方式决定了数据切分后的性能好坏，因此也是最重要的配置。

如上面例子配置了一个切分规则，名为sharding-by-hour 对应的切分方式（function ）是按日期切分，该配置中：

### tableRule

name 为schema.xml 中table 标签中对应的 rule="sharding-by-hour" ,也就是配置表的分片规则，

columns 是表的切分字段： createTime 创建日期。

algorithm 是规则对应的切分规则：映射到function 的name。

### function

function 配置是分片规则的配置。

name 为切分规则的名称，名字任意取，但是需要与tableRule 中匹配。

class 是切分规则对应的切分类，写死，需要哪种规则则配置哪种，例如本例子是按小时分片：org.opencloudb.route.function.LatestMonthPartion

property 标签是切分规则对应的不同属性，不同的切分规则配置不同。