

# Mysql

---

基础语句

用户管理

权限控制

函数

约束

事务

为什么这些隔离级别可以隔离各种问题?

存储引擎

mysql体系结构

存储引擎的特点

索引

索引结构

为什么InnoDB存储引擎选择使用B+树?

索引分类

回表索引

索引语法

性能分析

索引的使用

单列索引和联合索引的选择

索引设计原则

SQL优化

插入数据

主键优化

Order By优化

group by优化

limit优化

count优化

update优化

视图

视图的更新

视图的作用

存储过程

介绍

特点

创建 调用 查看 删除

变量

存储过程关键字

if

存储过程的参数

case

while

repeat

loop

cursor游标

条件处理程序handler

存储函数

触发器

锁

概述

全局锁

表级锁

行级锁

InnoDB引擎

架构

后台线程

事务原理

redo log和undo log共同保证事务的一致性

redo log 解决事务的持久性

undo log 解决事务的原子性

MVCC 解决事务的隔离性

# 基础语句

查询所有数据库

```
SHOW DATABASES;
```

查询当前数据库

```
SELECT DATABASE();
```

创建

```
CREATE DATABASE [ IF NOT EXISTS ] 数据库名 [ DEFAULT CHARSET 字符集 ] [ COLLATE 排序规则];
```

删除

```
DROP DATABASE [ IF EXISTS ] 数据库名;
```

使用

```
USE 数据库名;
```

```
SHOW TABLES;
```

```
CREATE TABLE 表名 ( 字段 字段类型 , 字段 字段类型 );
```

```
DESC 表名;
```

```
SHOW CREATE TABLE 表名;
```

```
ALTER TABLE 表名 ADD/MODIFY/CHANGE/DROP/RENAME TO ...;
```

```
DROP TABLE 表名;
```

- DDL-表操作-创建

```
CREATE TABLE 表名(  
    字段1 字段1类型 [COMMENT 字段1注释],  
    字段2 字段2类型 [COMMENT 字段2注释],  
    字段3 字段3类型 [COMMENT 字段3注释],  
    .....  
    字段n 字段n类型 [COMMENT 字段n注释]  
) [COMMENT 表注释];
```

- DML-添加数据

1. 给指定字段添加数据

```
INSERT INTO 表名 (字段名1, 字段名2, ...) VALUES (值1, 值2, ...);
```

2. 给全部字段添加数据

```
INSERT INTO 表名 VALUES (值1, 值2, ...);
```

3. 批量添加数据

```
INSERT INTO 表名 (字段名1, 字段名2, ...) VALUES (值1, 值2, ...), (值1, 值2, ...), (值1, 值2, ...);
```

```
INSERT INTO 表名 VALUES (值1, 值2, ...), (值1, 值2, ...), (值1, 值2, ...);
```

SELECT

字段列表

FROM

表名列表

WHERE

条件列表

GROUP BY

分组字段列表

HAVING

分组后条件列表

ORDER BY

排序字段列表

LIMIT

分页参数

2. where与having区别

- 执行时机不同：where是分组之前进行过滤，不满足where条件，不参与分组；而having是分组之后对结果进行过滤。
- 判断条件不同：where不能对聚合函数进行判断，而having可以。



## 用户管理

### 1. 查询用户

```

USE mysql;
SELECT * FROM user;

```

### 2. 创建用户

```

CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';

```

### 3. 修改用户密码

```

ALTER USER '用户名'@'主机名' IDENTIFIED WITH mysql_native_password BY '新密码';

```

### 4. 删除用户

```

DROP USER '用户名'@'主机名';

```

#### 注意：

- 主机名可以使用 % 通配。
- 这类SQL开发人员操作的比较少，主要是DBA（Database Administrator 数据库管理员）使用。

## 权限控制

权限	说明
ALL, ALL PRIVILEGES	所有权限
SELECT	查询数据
INSERT	插入数据
UPDATE	修改数据
DELETE	删除数据
ALTER	修改表
DROP	删除数据库/表/视图
CREATE	创建数据库/表

### ● DCL-权限控制

#### 1. 查询权限

```
SHOW GRANTS FOR '用户名'@'主机名';
```

#### 2. 授予权限

```
GRANT 权限列表 ON 数据库名.表名 TO '用户名'@'主机名';
```

#### 3. 撤销权限

```
REVOKE 权限列表 ON 数据库名.表名 FROM '用户名'@'主机名';
```

注意：

- 多个权限之间，使用逗号分隔
- 授权时，数据库名和表名可以使用 \* 进行通配，代表所有。

# 函数

函数	功能
CONCAT(S1,S2,...Sn)	字符串拼接，将S1, S2, ... Sn拼接成一个字符串
LOWER(str)	将字符串str全部转为小写
UPPER(str)	将字符串str全部转为大写
LPAD(str,n,pad)	左填充，用字符串pad对str的左边进行填充，达到n个字符串长度
RPAD(str,n,pad)	右填充，用字符串pad对str的右边进行填充，达到n个字符串长度
TRIM(str)	去掉字符串头部和尾部的空格
SUBSTRING(str,start,len)	返回从字符串str从start位置起的len个长度的字符串

函数	功能
CEIL(x)	向上取整
FLOOR(x)	向下取整
MOD(x,y)	返回x/y的模
RAND()	返回0~1内的随机数
ROUND(x,y)	求参数x的四舍五入的值，保留y位小数

常见的日期函数如下：

函数	功能
CURDATE()	返回当前日期
CURTIME()	返回当前时间
NOW()	返回当前日期和时间
YEAR(date)	获取指定date的年份
MONTH(date)	获取指定date的月份
DAY(date)	获取指定date的日期
DATE_ADD(date, INTERVAL expr type)	返回一个日期/时间值加上一个时间间隔expr后的时间值
DATEDIFF(date1,date2)	返回起始时间date1 和 结束时间date2之间的天数

流程函数也是很常用的一类函数，可以在SQL语句中实现条件筛选，从而提高语句的效率。

函数	功能
IF(value , t , f)	如果value为true，则返回t，否则返回f
IFNULL(value1 , value2)	如果value1不为空，返回value1，否则返回value2
CASE WHEN [ val1 ] THEN [res1] ... ELSE [ default ] END	如果val1为true，返回res1，... 否则返回default默认值
CASE [ expr ] WHEN [ val1 ] THEN [res1] ... ELSE [ default ] END	如果expr的值等于val1，返回res1，... 否则返回default默认值

# 约束

约束	描述	关键字
非空约束	限制该字段的数据不能为null	NOT NULL
唯一约束	保证该字段的所有数据都是唯一、不重复的	UNIQUE
主键约束	主键是一行数据的唯一标识，要求非空且唯一	PRIMARY KEY
默认约束	保存数据时，如果未指定该字段的值，则采用默认值	DEFAULT
检查约束(8.0.16版本之后)	保证字段值满足某一个条件	CHECK
外键约束	用来让两张表的数据之间建立连接，保证数据的一致性和完整性	FOREIGN KEY

➤ 添加外键

```
CREATE TABLE 表名(
    字段名 数据类型,
    ...
    [CONSTRAINT] [外键名称] FOREIGN KEY(外键字段名) REFERENCES 主表(主表列名)
);
```

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY(外键字段名) REFERENCES 主表(主表列名);
```

行为	说明
NO ACTION	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 RESTRICT 一致)
RESTRICT	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 NO ACTION 一致)
CASCADE	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有，则也删除/更新外键在子表中的记录。
SET NULL	当在父表中删除对应记录时，首先检查该记录是否有对应外键，如果有则设置子表中该外键值为null(这就要求该外键允许取null)。
SET DEFAULT	父表有变更时，子表将外键列设置成一个默认的值(Innodb不支持)

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY(外键字段) REFERENCES 主表名(主表字段) ON UPDATE CASCADE ON DELETE CASCADE;
```

# 事务

- 查看/设置事务提交方式

```
SELECT @@autocommit;
SET @@autocommit = 0;
```

- 提交事务

```
COMMIT;
```

- 回滚事务

```
ROLLBACK;
```

autocommit=0 设置为手动提交

- 开启事务

```
START TRANSACTION 或 BEGIN;
```

- 提交事务

```
COMMIT;
```

- 回滚事务

```
ROLLBACK;
```

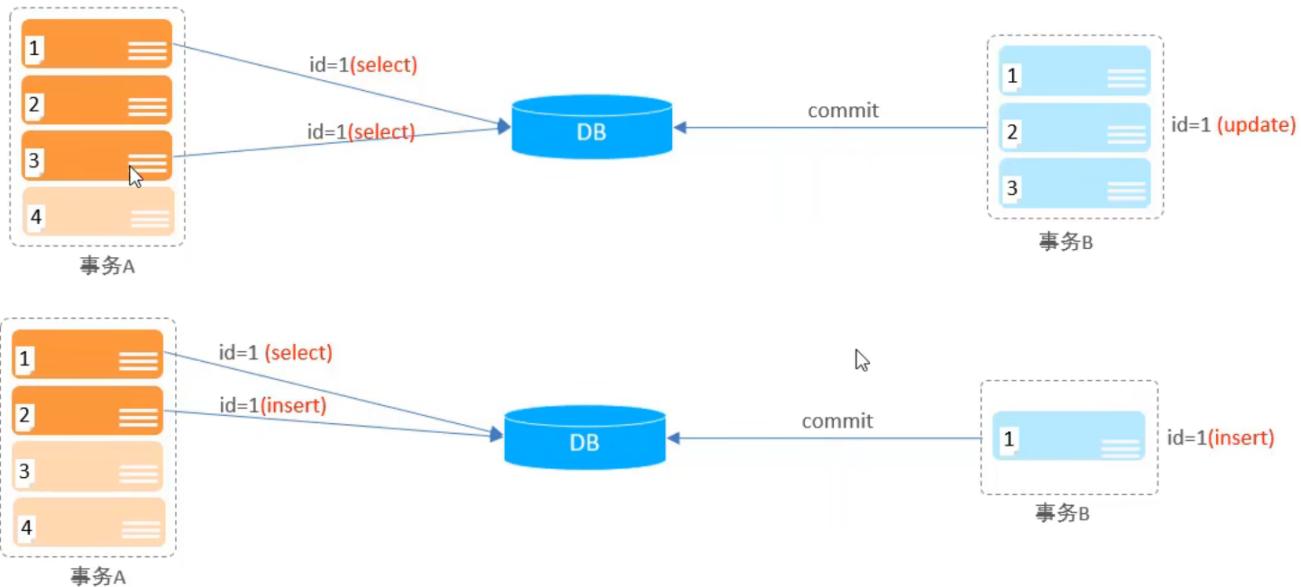
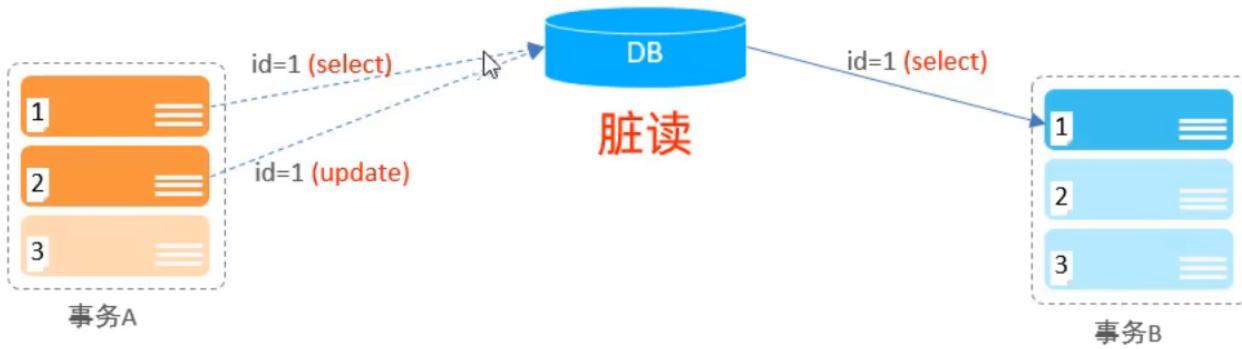
目前autocommit=1 为自动提交事务

## 事务四大特性

- 原子性 (Atomicity) : 事务是不可分割的最小操作单元，要么全部成功，要么全部失败。
- 一致性 (Consistency) : 事务完成时，必须使所有的数据都保持一致状态。
- 隔离性 (Isolation) : 数据库系统提供的隔离机制，保证事务在不受外部并发操作影响的独立环境下运行。
- 持久性 (Durability) : 事务一旦提交或回滚，它对数据库中的数据的改变就是永久的。

## 并发事务问题

问题	描述
脏读	一个事务读到另外一个事务还没有提交的数据。
不可重复读	一个事务先后读取同一条记录，但两次读取的数据不同，称之为不可重复读。
幻读	一个事务按照条件查询数据时，没有对应的数据行，但是在插入数据时，又发现这行数据已经存在，好像出现了“幻影”。



## 事务隔离级别

隔离级别	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	✗	√	√
Repeatable Read(默认)	✗	✗	√
Serializable	✗	✗	✗

-- 查看事务隔离级别

```
SELECT @@TRANSACTION_ISOLATION;
```

↳

-- 设置事务隔离级别

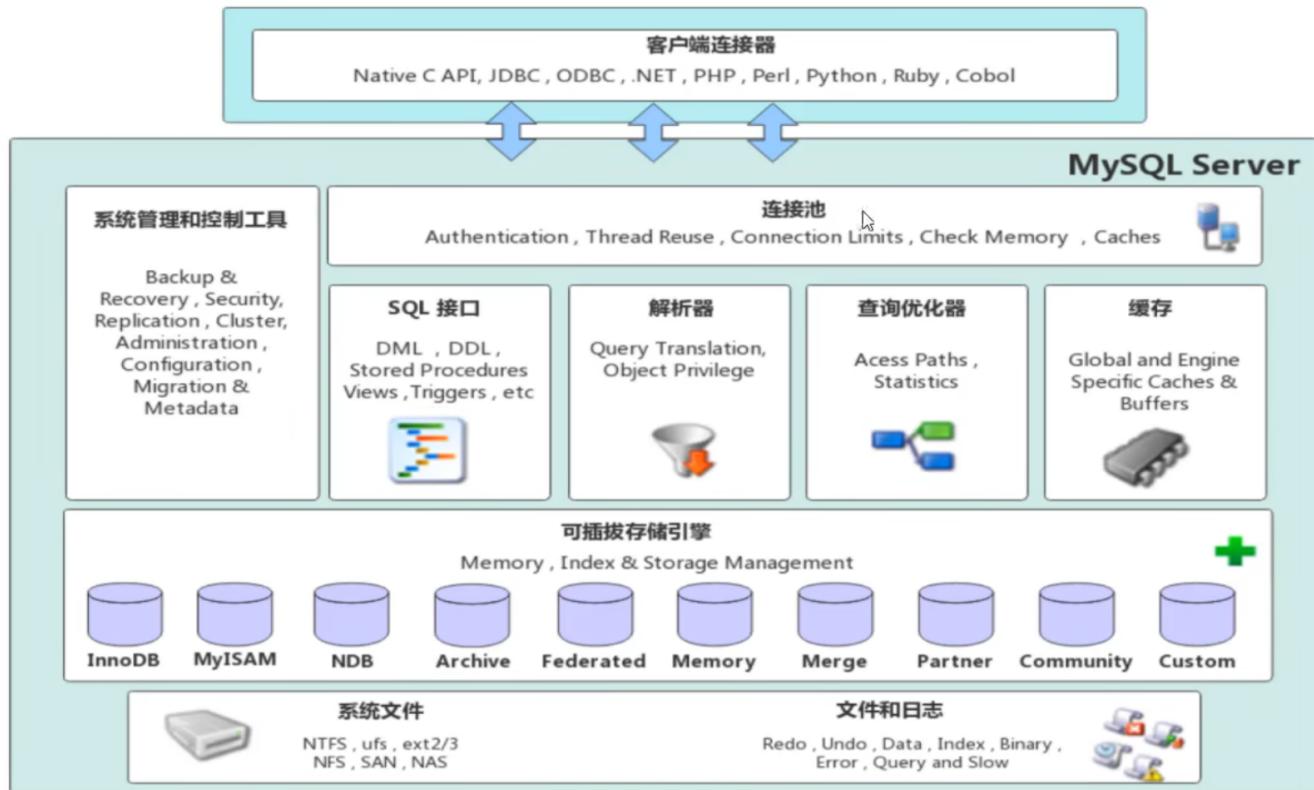
```
SET [ SESSION|GLOBAL ] TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

为什么这些隔离级别可以隔离各种问题？

# 存储引擎

## mysql体系结构

### MySQL体系结构



### MySQL体系结构

#### ● 连接层

最上层是一些客户端和链接服务，主要完成一些类似于连接处理、授权认证、及相关的安全方案。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

#### ● 服务层

第二层架构主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部分内置函数的执行。所有跨存储引擎的功能也在这一层实现，如过程、函数等。

#### ● 引擎层

存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API和存储引擎进行通信。不同的存储引擎具有不同的功能，这样我们可以根据自己的需要，来选取合适的存储引擎。

#### ● 存储层

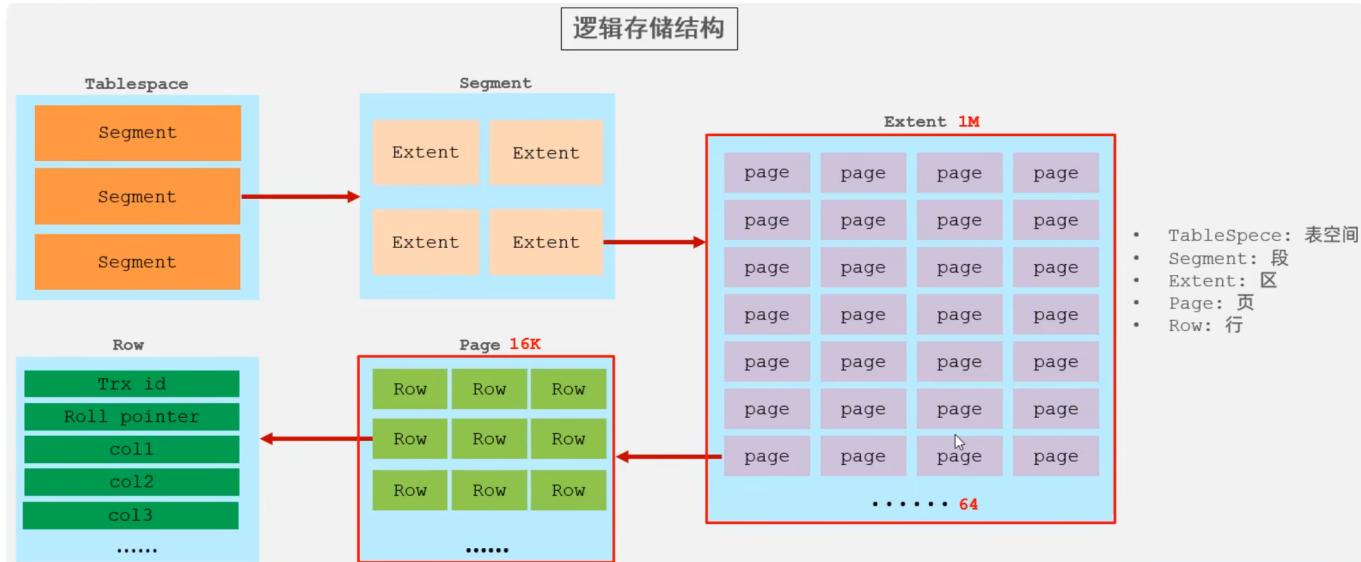
主要是将数据存储在文件系统之上，并完成与存储引擎的交互。

## 存储引擎的特点

- InnoDB 是一种兼顾高可靠性和高性能的通用存储引擎
  - DML操作遵循ACID模型 支持事务 行级锁，提高并发访问性能，支持外键FOREIGN KEY约束 保证数据的完整性和正确性
  - xxx.ibd xxx代表的是表名 innoDB引擎的每张表都会对应这样一个表空间文件 存储该表的表结构 数据和索引

## 存储引擎特点

### ● InnoDB



- MyISAM是MySQL早期的默认存储引擎
  - 不支持事务 不支持外键 支持表锁 不支持行锁 访问速度快
  - xxx.sdi 存储表结构信息 xxx.MYD 存储数据 xxx.MYI 存储索引
- Memory存储引擎的表数据是存储在内存中的 由于受到硬件影响 断电问题 只能将这些表作为临时表或者缓存使用
  - 内存存放 hash索引
  - xxx.sdi存储表结构信息

## 存储引擎特点

特点	InnoDB	MyISAM	Memory
存储限制	64TB	有	有
事务安全	支持	-	-
锁机制	行锁	表锁	表锁
B+tree索引	支持	支持	支持
Hash索引	-	-	支持
全文索引	支持(5.6版本之后)	支持	-
空间使用	高	低	N/A
内存使用	高	低	中等
批量插入速度	低	高	高
支持外键	支持	-	-

- InnoDB是Mysql的默认引擎，支持事务 外键 如果应用对事务的完整性有比较高的要求 在并发条件下要求数据的一致性 数据操作除了插入和查询之外 还包含很多的更新删除操作 那么InnoDB存储引擎是比较合适的选择
- MyISAM 如果应用是以读操作和插入操作为主 只有很少的更新和删除操作 并且对事务的完整性 并发性要求不是很高 那么选择这个存储引擎
- MEMORY 将所有的数据保存在内存中 访问速度快 通常用于临时表及缓存 MEMORY的缺陷是对表的大小有限制 太大的表无法缓存在内存中 并且无法保证数据的安全性

## 索引

索引是帮助MySQL高效获取数据的数据结构

- 优缺点

优势	劣势
提高数据检索的效率，降低数据库的IO成本	索引列也是要占用空间的。
通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。	索引大大提高了查询效率，同时却也降低更新表的速度，如对表进行INSERT、UPDATE、DELETE时，效率降低。

## 索引结构

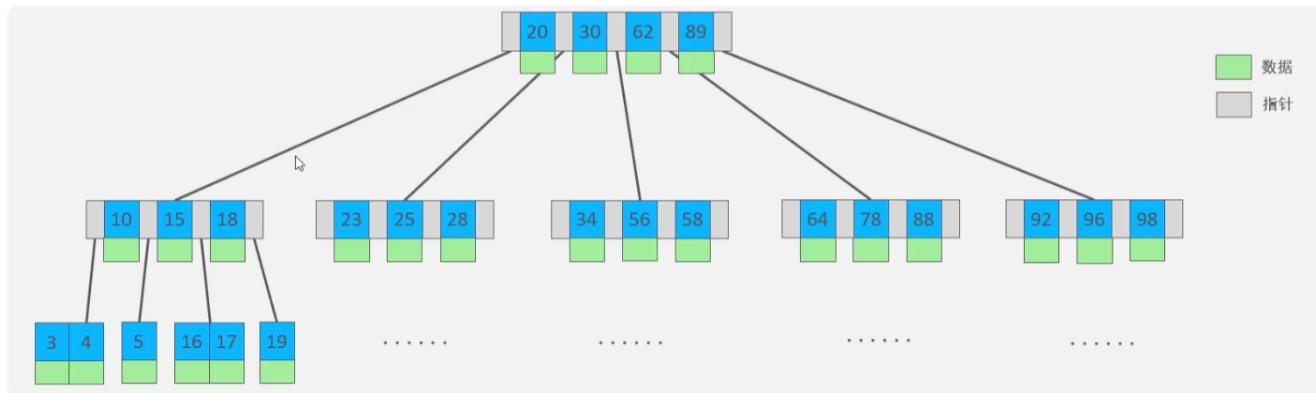
MySQL是在存储引擎层实现的 不同的存储引擎有不同的结构

索引结构	描述
B+Tree索引	最常见的索引类型，大部分引擎都支持 B+ 树索引
Hash索引	底层数据结构是用哈希表实现的，只有精确匹配索引列的查询才有效，不支持范围查询
R-tree(空间索引) 	空间索引是MyISAM引擎的一个特殊索引类型，主要用于地理空间数据类型，通常使用较少
Full-text(全文索引)	是一种通过建立倒排索引，快速匹配文档的方式。类似于Lucene,Solr,ES

索引	InnoDB	MyISAM	Memory
B+tree索引	支持	支持	支持
Hash 索引	不支持	不支持	支持
R-tree 索引	不支持	支持	不支持
Full-text	5.6版本之后支持 	支持	不支持

## ● B-Tree (多路平衡查找树)

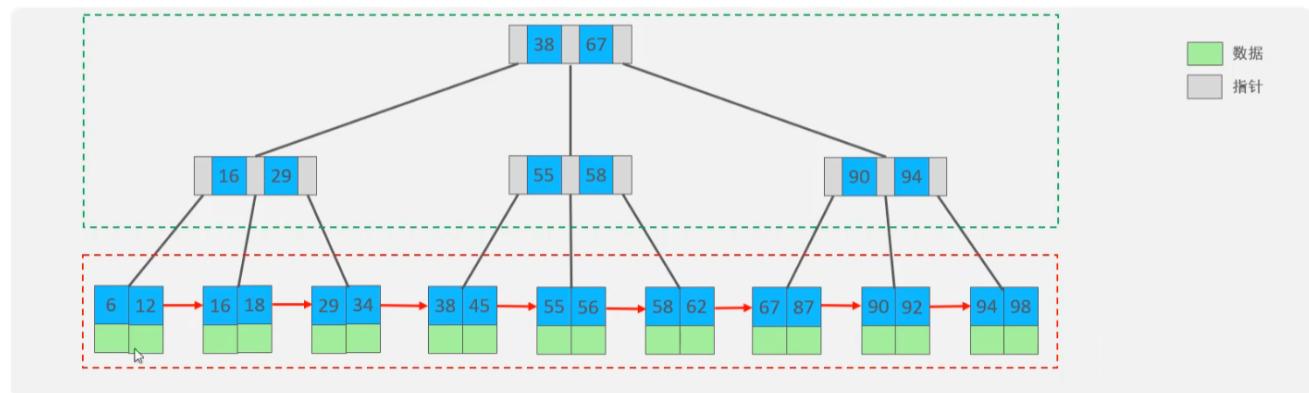
以一颗最大度数 (max-degree) 为5(5阶)的b-tree为例(每个节点最多存储4个key, 5个指针):



知识小贴士: 树的度数指的是一个节点的子节点个数。

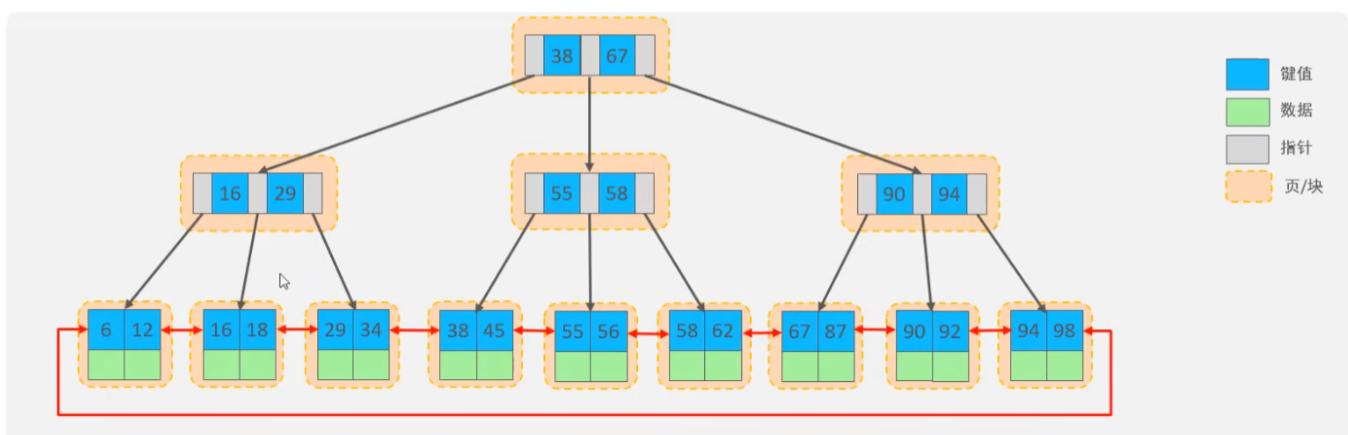
## ● B+Tree

以一颗最大度数 (max-degree) 为4 (4阶) 的b+tree为例:



和B树的区别 所有数据都会出现在叶子节点 叶子节点形成一个单向链表

MySQL索引数据结构对经典的B+树进行了优化 在原本B+树的基础上 增加了一个指向相邻叶子节点的链表指针 就形成了带有顺序指针的B+树 提高区间访问的性能



- Hash索引只支持对等比较 不支持范围查询 无法利用索引完成排序操作 查询效率高 通常只需要一次

检索就可以了 效率通常要高于B+树索引

- 在MySQL中 支持hash索引的是Memory引擎 InnoDB中具有自适应hash功能 hash索引是存储引擎根据B+树在指定条件下自动构建的

## 为什么InnoDB存储引擎选择使用B+树?

- 相对于二叉树 层级更少 搜索效率高
- 对于B树 无论是叶子节点还是非叶子节点 都会保存数据 这样导致一页中存储的键值减少 指针跟着减少 要同样保存大量数据 只能增加树的层数 导致性能降低
- 叶子节点变为双向链表 方便范围查询 和排序
- 相对于Hash索引 B+树支持范围匹配和排序操作

## 索引分类

分类	含义	特点	关键字
主键索引	针对于表中主键创建的索引	默认自动创建, 只能有一个	PRIMARY
唯一索引	避免同一个表中某数据列中的值重复	可以有多个	UNIQUE
常规索引	快速定位特定数据	可以有多个	
全文索引	全文索引查找的是文本中的关键词, 而不是比较索引中的值	可以有多个	FULLTEXT

分类	含义	特点
聚集索引(Clustered Index)	将数据存储与索引放到了一块, 索引结构的叶子节点保存了行数据	必须有,而且只有一个
二级索引(Secondary Index)	将数据与索引分开存储, 索引结构的叶子节点关联的是对应的主键	可以存在多个

- 如果存在主键 则主键索引就是聚集索引
- 如果不存在主键 将使用第一个唯一索引作为聚集索引
- 如果表没有主键 或没有合适的唯一索引 则InnoDB会自动生成一个rowid作为隐藏的聚集索引
  - 聚集索引叶子节点挂的是这一行的数据
  - 二级索引叶子节点挂的是这个字段对应这一行的id值

## 回表索引

先通过二级索引找到所需数据的id值 再通过聚集索引获得这一行的数据

# 索引语法

## 索引语法

- 创建索引

```
CREATE [UNIQUE|FULLTEXT] INDEX index_name ON table_name (index_col_name,...);
```

- 查看索引

```
SHOW INDEX FROM table_name;
```

- 删除索引

```
DROP INDEX index_name ON table_name;
```

# 性能分析

## 查询频次

MySQL 客户端连接成功后，通过 show [session|global] status 命令可以提供服务器状态信息。通过如下指令，可以查看当前数据库的 INSERT、UPDATE、DELETE、SELECT 的访问频次：

```
SHOW GLOBAL STATUS LIKE 'Com_____';
```

## 慢查询日志

- 慢查询日志

慢查询日志记录了所有执行时间超过指定参数 (long\_query\_time, 单位：秒，默认10秒) 的所有SQL语句的日志。

MySQL的慢查询日志默认没有开启，需要在MySQL的配置文件 (/etc/my.cnf) 中配置如下信息：

```
# 开启MySQL慢日志查询开关
slow_query_log=1
# 设置慢日志的时间为2秒，SQL语句执行时间超过2秒，就会视为慢查询，记录慢查询日志
long_query_time=2
```

## profile详情

- profile详情

执行一系列的业务SQL的操作，然后通过如下指令查看指令的执行耗时：

```
#查看每一条SQL的耗时基本情况
```

```
show profiles;
```

```
#查看指定query_id的SQL语句各个阶段的耗时情况
```

```
show profile for query query_id;
```

```
#查看指定query_id的SQL语句CPU的使用情况
```

```
show profile cpu for query query_id;
```

```

mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query
+-----+-----+-----+
| 2 | 0.00041925 | SELECT DATABASE()
| 3 | 0.01222200 | show databases
| 4 | 0.00726975 | show tables
| 5 | 0.00162125 | show tables
| 6 | 0.00062300 | select * from tb_user
| 7 | 0.00097975 | select count(*) from tb_user
| 8 | 0.00053200 | select * from tb_user
| 9 | 9.94508625 | select count(*) from tb_sku
| 10 | 0.00366750 | select @@have_profiling
| 11 | 0.00035200 | select @@profiling
| 12 | 0.00015650 | select @@profiling
| 13 | 0.00053075 | select * from tb_user
| 14 | 0.00062125 | select * from tb_user where id = 1
| 15 | 0.00741925 | select * from tb_user where name = '白起'
| 16 | 9.53712800 | select count(*) from tb_sku
+-----+-----+-----+
15 rows in set, 1 warning (0.00 sec)

```

```

mysql> show profile for query 16;
+-----+-----+
| Status | Duration |
+-----+-----+
| starting | 0.000137 |
| Executing hook on transaction | 0.000009 |
| starting | 0.000028 |
| checking permissions | 0.000027 |
| opening tables | 0.000033 |
| init | 0.000006 |
| System lock | 0.000010 |
| optimizing | 0.000022 |
| statistics | 0.000015 |
| preparing | 0.000013 |
| executing | 9.536456 |
| end | 0.000025 |
| query end | 0.000007 |
| waiting for handler commit | 0.000013 |
| closing tables | 0.000014 |
| freeing items | 0.000048 |
| logging slow query | 0.000244 |
| cleaning up | 0.000024 |
+-----+-----+
18 rows in set, 1 warning (0.01 sec)

```

## explain执行计划

获取MySQL如何执行select语句的信息 包括select语句执行过程中表如何连接和连接的顺序

explain select 字段列表 from 表明 where 条件

- explain执行计划

EXPLAIN 执行计划各字段含义：

➤ Id

select查询的序列号，表示查询中执行select子句或者是操作表的顺序(id相同，执行顺序从上到下；id不同，值越大，越先执行)。

➤ select\_type

表示 SELECT 的类型，常见的取值有 SIMPLE (简单表，即不使用表连接或者子查询)、PRIMARY (主查询，即外层的查询)、UNION (UNION 中的第二个或者后面的查询语句)、SUBQUERY (SELECT/WHERE之后包含了子查询) 等

➤ type

表示连接类型，性能由好到差的连接类型为NULL、system、const、eq\_ref、ref、range、index、all。

➤ possible\_key

显示可能应用在这张表上的索引，一个或多个。

➤ Key

实际使用的索引，如果为NULL，则没有使用索引。

➤ Key\_len

表示索引中使用的字节数，该值为索引字段最大可能长度，并非实际使用长度，在不损失精确性的前提下，长度越短越好。

➤ rows

MySQL认为必须要执行查询的行数，在innodb引擎的表中，是一个估计值，可能并不总是准确的。

➤ filtered

表示返回结果的行数占需读取行数的百分比，filtered 的值越大越好。

采用主键或者唯一索引进行索引是 type出现const 使用非唯一性索引进行查询时 出现ref

## 索引的使用

- 最左前缀法则

- 如果索引了多列(联合索引)，要遵循最左前缀法则 指的是查询从索引的最左列开始 且不跳过索引中的列
- 如果跳跃某一列 索引将部分失效(后面的字段索引失效)

```

mysql> explain select * from tb_user where profession = '软件工程' and age = 31 and status = '0';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
| 1 | SIMPLE     | tb_user | NULL      | ref  | idx_user_pro_age_sta | idx_user_pro_age_sta | 54 | const,const,const | 1 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+
mysql> explain select * from tb_user where age = 31 and status = '0';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | tb_user | NULL      | ALL  | NULL          | NULL | NULL    | NULL | 24 | 4.17 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

不满足最左前缀法则 索引失效

- 范围查询

- 联合索引中 如果出现范围查询 $><$  则 范围查询右侧的所有列索引失效
  - 修改策略 把 $><$ 改为 $\geq \leq$

- 索引列运算

- 不要在索引列上进行运算操作 否则索引会失效

- `explain select * from tb_user where substring(phone,10,2) = '15';`

- 字符串不加引号

- 字符串类型字段使用时 不加引号 则索引将失效

- 模糊查询

- 如果仅仅是尾部模糊匹配 索引不会失效 如果是头部模糊匹配 索引失效

- or连接的条件

- 用or分隔开的条件 如果or前的条件中的列有索引而后面的列中没有索引 则涉及的索引都不会被用到

- 解决方法 将or中的所有条件字段都添加索引

- 数据分布影响

- 如果MySQL评估 使用索引比全表更慢 则不使用索引

- SQL提示 是优化数据库的一种重要手段 就是在SQL语句中加入一些人为的提示来达到优化操作的目的

use index:

```
explain select * from tb_user use index(idx_user_pro) where profession = '软件工程';
```

ignore index:

```
explain select * from tb_user ignore index(idx_user_pro) where profession = '软件工程';
```

force index:

- ```
explain select * from tb_user force index(idx_user_pro) where profession = '软件工程';
```

- 覆盖索引

- 尽量使用覆盖索引(查询使用了索引 并且需要返回的列 在该索引中已经全部能够找到) 减少 select\*
- using index condition 查找使用了索引 但是回表查询了数据 效率较低
- using where using index 表示查找使用了索引 但是需要的数据在索引列中都能找到 所以不需要回表查询数据

- 前缀索引

- 当字段类型为字符串时 有时需要索引很长的字符串 这让索引变得很大 浪费大量的IO 影响查询效率 可以只将字符串的一部分前缀建立索引 可以大大节约索引空间 提高索引效率

➤ 语法

```
create index idx_xxxx on table_name(column(n));
```

- n表示截取多少个字符作为前缀索引
  - 前缀长度可以根据索引的选择性来确定 选择性是指不重复的索引值和数据表的记录总数的比值 索引选择性越高则查询效率越高
- ```
select count(distinct email) / count(*) from tb_user;
```
- ```
select count(distinct substring(email,1,5)) / count(*) from tb_user;
```

## 单列索引和联合索引的选择

- 单列索引 即一个索引只包含一个列
- 联合索引 即一个索引包含多个列
- 在业务场景中 如果存在多个查询条件 考虑针对查询字段建立索引时 建议建立联合索引而非单列索引

## 索引设计原则

- 针对数据量较大 查询比较频繁的表建立索引
- 针对常作为查询条件 排序 分组操作的字段建立索引

- 尽量选择区分度高的列作为索引 尽量建立唯一索引 区分度较高 使用索引的效率较高
- 如果是字符串类型的字段 字段的长度较长 可以针对字段的特点 建立前缀索引
- 尽量使用联合索引 减少单列索引 查询时 联合索引很多时候可以覆盖索引 节省存储空间 避免回表 提高查询效率
- 要控制索引的数量 索引并不是多多益善 索引越多 维护索引结构的代价也就越大 会影响增删改的效率
- 如果索引列不能存储null值 需要在创建表时使用not null约束它 当优化器知道每列是否包含null值时 可以更好地确定哪个索引最有效的用于查询

## SQL优化

### 插入数据

- insert优化
  - 批量插入

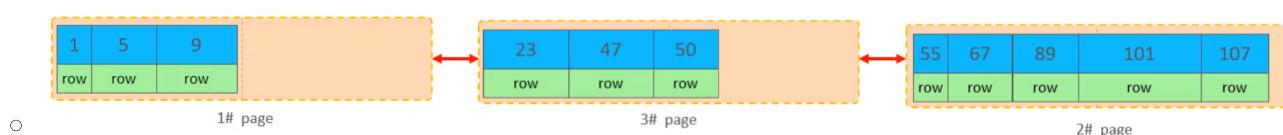
```
#客户端连接服务端时，加上参数 --local-infile
mysql --local-infile -u root -p
#设置全局参数local_infile为1，开启从本地加载文件导入数据的开关
set global local_infile=1;
#执行load指令将准备好的数据，加载到表结构中
■ load data local infile '/root/sql.log' into table `tb_user` fields terminated by ',' lines terminated by '\n';
```

- 手动提交事务
- 主键顺序插入

### 主键优化

- 数据组织方式
  - 在InnoDB存储引擎中 表数据都是根据主键顺序组织存放的 这种存储方式的表成为索引组织表
- 页分裂
  - 页可以为空 也可以存储一半 也可以填充满 每个页包含了2-N行数据 如果一行数据过大 就会行溢出 根据主键排列 如果乱序插入 就会出现页分裂的现象

主键乱序插入



- 页合并
  - 当删除一行记录时 实际上记录并没有被物理删除 只是纪录被标记为删除 并且它的空间变得允许被其他记录声明使用
  - 当页中删除的记录打到MERGE\_THRESHOLD 默认为页的50% InnoDB会开始寻找最靠近的页（前或后）看看是否可以将两个页合并达到优化空间的使用
- 主键设计原则
  - 满足业务需求的情况下 尽量减少主键的长度
  - 插入数据时 尽量顺序插入 选择使用AUTO\_INCREMENT自增主键
  - 尽量不要用UUID做主键或者其他自然主键 例如身份证号
  - 业务操作时 尽量避免对主键的修改

## Order By优化

- Using filesort 通过表的索引或者全表扫描 读取满足条件的数据行 然后在排序缓冲区sort buffer 中完成排序操作 所有不是通过索引直接返回排序结果的排序都叫做FileSort排序
- UsingIndex 通过有序索引顺序扫描直接返回有序数据 这种情况不需要额外排序 效率更好

```
#没有创建索引时，根据age, phone进行排序
explain select id,age,phone from tb_user order by age , phone;

#创建索引
create index idx_user_age_phone_aa on tb_user(age,phone);

#创建索引后，根据age, phone进行升序排序
explain select id,age,phone from tb_user order by age , phone; ⌂

#创建索引后，根据age, phone进行降序排序
explain select id,age,phone from tb_user order by age desc , phone desc ;
```

- 创建的索引默认是按照升序排列的 如果出现降序排列 则会出现UsingFileSort 前提是使用覆盖索引
- 根据排序字段建立合适的索引 多字段排序时 也遵循最左前缀法则
- 尽量使用覆盖索引
- 多字段排序 一个升序一个降序 此时需要注意联合索引在创建时的规则
- 如果不可避免的出现FileSort 大数据量排序时 可以适当增加排序缓冲区大小sort\_buffer\_size

## group by优化

- 在分组操作时 可以通过索引来提高效率
- 分组操作时 索引的使用也是满足最左前缀法则的

## limit优化

优化思路 一般分页查询时 通过创建覆盖索引 能够比较好的提升性能 可以通过覆盖索引加子查询的形式进行优化

## count优化

- MyISAM引擎把一个表的总行数放在了磁盘上 因此执行count的时候会直接返回这个数 效率很高
- InnoDB引擎执行count的时候 需要把数据一行一行的从引擎里面读出来 累积计数
- 优化思路 自己计数
- count是一种聚合函数 对于返回的数据集 一行行的判断 如果count函数的参数不是null 累计值就加1 否则不加 最后返回累计值
- count(主键) InnoDB引擎会遍历整张表 把每一行的主键id值都取出来 返回给服务层 服务层拿到主键之后 会直接按行进行累加 (主键不可能为null )
- count(字段)
  - 没有not null InnoDB 会遍历整张表把每一行的字段值都取出来 返回给服务层 服务层判断是否为null 不为null 计数累加
  - 有not null InnoDB会遍历整张表把每一行的字段值都取出来 返回给服务层 直接按行进行累加
- count(1) InnoDB遍历整张表 但不取值 服务层对于返回的每一行 放一个数字1进去 直接按行进行累加
- count(\*) InnoDB 并不会把所有字段都读取出来 而是专门做了优化 不取值 服务层直接按行进行累加
- count(字段 )<count(主键id)<count(1)<count(\*)

## update优化

InnoDB的行锁是针对索引加的锁 不是针对记录加的锁 并且该索引不能失效 否则会从行锁升级为表锁  
优化方案 where语句查询的时候按照索引规则进行查询 并且使用有索引的字段进行where

## 视图

视图是一种虚拟存在的表 视图中的数据并不在数据库中真实存在 行和列数据来自定义视图的查询中用到的表 并且是在使用视图时动态生成的

视图只保存了查询的SQL逻辑 不保存查询结果 所以在创建视图的时候 主要工作就在创建SQL查询上

```
CREATE [OR REPLACE] VIEW 视图名称[(列名列表)] AS SELECT语句 [WITH[ CASCDED | LOCAL ] CHECK OPTION ]
```

当使用with check option子句创建视图时 MySQL会通过检查视图检查正在更改的每一行 例如插入 更新 删除 使其符合视图的定义 MySQL允许基于另一个视图创建视图 还会检查依赖视图中的规则来保持一致性 为了确定检查的范围 mysql提供了两个选项 cascaded和local 默认值为cascaded

## 视图的更新

- 要使视图可以更新 视图中的行与基础表中的行之间必须存在一对一的关系 如果视图中包含任意一项 该视图不可以更新
  - 聚合函数或者窗口函数
  - DISTINCT
  - Group By
  - Having
  - Union或者Union All

## 视图的作用

- 简单
  - 视图不仅可以简化用户对于数据的理解 也可以简化操作 被经常使用的查询可以被定义为视图 使用用户不必为以后的操作每次指定全部的条件
- 安全
  - 数据库可以授权 但不可以授权到数据库特定行和特定列上 通过视图用户只能查询和修改他们所能见到的数据
- 数据独立
  - 视图可以帮助用户屏蔽真实表结构变化带来的影响

## 存储过程

### 介绍

存储过程是事先经过编译并且存储在数据库中的一段sql语句集合 调用存储过程可以简化操作 减少数据在数据库和应用服务器之间的传输 提高数据处理的效率

## 特点

- 封装 复用
- 可以接受参数 可以返回数据
- 减少网络交互 效率提升

## 创建 调用 查看 删除

```
CREATE PROCEDURE 存储过程名称 ([参数列表])
BEGIN
    -- SQL语句
    ↴
END ;
```

- 调用

```
CALL 名称 ([参数]);
```

- 查看

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES WHERE ROUTINE_SCHEMA = 'xxx' ; -- 查询指定数据库的存储过程及状态信息
SHOW CREATE PROCEDURE 存储过程名称 ; -- 查询某个存储过程的定义
```

- 删除

```
DROP PROCEDURE [IF EXISTS] 存储过程名称 ;
```

在命令行中 执行创建存储过程的sql时 需要通过关键字delimiter指定sql语句的结束符

## 变量

系统变量是mysql服务器提供 不是用户定义的 属于服务器层面 分为全局变量和会话变量

全局变量设置之后全局生效 但是在服务器重启之后 会恢复到初始值

如果没有指定session global 默认是session 会话变量

mysql服务重启之后 所设置的全局参数会取消 要想不失效 需要在/etc/my.cnf中设置

#### ➤ 查看系统变量

```
SHOW [ SESSION|GLOBAL ] VARIABLES ;          -- 查看所有系统变量  
SHOW [ SESSION|GLOBAL ] VARIABLES LIKE '.....'; -- 可以通过LIKE模糊匹配方式查找变量  
SELECT @@[SESSION|GLOBAL] 系统变量名;      -- 查看指定变量的值
```

#### ➤ 设置系统变量

```
SET [ SESSION|GLOBAL ] 系统变量名 = 值 ;  
SET @@[SESSION|GLOBAL]系统变量名 = 值 ;
```

用户自定义变量是用户根据需要自己定义的变量 用户变量不用提前声明 在用的时候直接用@变量名 作用域为当前连接

```
SET @var_name = expr [, @var_name = expr] ... ;  
SET @var_name := expr [, @var_name := expr] ... ;  
  
SELECT @var_name := expr [, @var_name := expr] ... ;  
SELECT 字段名 INTO @var_name FROM 表名;
```

#### ➤ 使用

```
SELECT @var_name ;
```

将查找结果赋值给变量

```
select @mycolor := 'red';  
  
select count(*) into @mycount from tb_user;
```

局部变量根据需要定义在局部生效的变量 访问之前 需要DECLARE声明 可用作存储过程内的局部变量和输入参数 局部变量的范围是在其内声明的BEGIN END块

#### ➤ 声明

```
DECLARE 变量名 变量类型 [DEFAULT ...];
```

变量类型就是数据库字段类型：INT、BIGINT、CHAR、VARCHAR、DATE、TIME等。

#### ➤ 赋值

```
SET 变量名 = 值 ;  
SET 变量名 := 值 ;  
SELECT 字段名 INTO 变量名 FROM 表名 ... ;
```

## 存储过程关键字

if

IF 条件1 THEN

.....

ELSEIF 条件2 THEN

-- 可选

.....

ELSE

-- 可选

.....

END IF;

## 存储过程的参数

| 类型    | 含义                     | 备注 |
|-------|------------------------|----|
| IN    | 该类参数作为输入，也就是需要调用时传入值   | 默认 |
| OUT   | 该类参数作为输出，也就是该参数可以作为返回值 |    |
| INOUT | 既可以作为输入参数，也可以作为输出参数    |    |

CREATE PROCEDURE 存储过程名称 ([ IN/OUT/INOUT 参数名 参数类型 ])

BEGIN

-- SQL语句

END ;

```
create procedure p4(in score int, out result varchar(10))
begin
    if score >= 85 then
        set result := '优秀';
    elseif score >= 60 then
        set result := '及格';
    else
        set result := '不及格';
    end if;
end;
```

💡  
call p4(score: 68, result: @result);

```
create procedure p5(inout score double)
begin
    set score := score * 0.5;
end;
```

```
set @score = 78;
call p5(score: @score);
select @score;
```

case

```
CASE case_value
    WHEN when_value1 THEN statement_list1
    [ WHEN when_value2 THEN statement_list2 ] ...
    [ ELSE statement_list ]
END CASE;
```

## while

有条件的循环控制语句 满足条件后再执行循环体中的sql

```
#先判定条件，如果条件为true，则执行逻辑，否则，不执行逻辑
WHILE 条件 DO
    SQL逻辑...
END WHILE;
```



## repeat

有条件的循环控制语句 当满足条件时退出循环

```
#先执行一次逻辑，然后判定逻辑是否满足，如果满足，则退出。如果不满足，则继续下一次循环
REPEAT
    SQL逻辑...
    UNTIL 条件
END REPEAT;
```

## loop

实现简单的循环 如果不在sql逻辑中添加退出循环的条件 可以用来实现简单的死循环 配合以下语句使用

- LEAVE 配合循环使用 退出循环
- ITERATE 必须用在循环中 跳过当前循环剩下的语句 直接进入下一次循环

```
[begin_label:] LOOP
```

SQL逻辑...

```
END LOOP [end_label];
```

LEAVE label; -- 退出指定标记的循环体

ITERATE label; -- 直接进入下一次循环

```
|create procedure p9(in n int)
|begin
|    declare total int default 0;
|
|    sum:loop
|        if n<=0 then
|            leave sum;
|        end if;
|
|        set total := total + n;
|        set n := n - 1;
|    end loop sum;
|
|    select total;
|
|end;
```

## cursor游标

用来存储查询结果集的数据类型 在存储过程和函数中可以使用游标对结果集进行循环的处理 游标的使用包括游标的声明 open fetch和close

➤ 声明游标

```
DECLARE 游标名称 CURSOR FOR 查询语句;
```



➤ 打开游标

```
OPEN 游标名称;
```

➤ 获取游标记录

```
FETCH 游标名称 INTO 变量 [ , 变量 ];
```

➤ 关闭游标

```
CLOSE 游标名称;
```

```
create procedure p11(in uage int)
begin
    declare uname varchar(100);
    declare upro varchar(100);
    declare u_cursor cursor for select name,profession from tb_user where age <= uage;

    drop table if exists tb_user_pro;
    create table if not exists tb_user_pro(
        id int primary key auto_increment,
        name varchar(100),
        profession varchar(100)
    );

```

```
open u_cursor;
while true do
    fetch u_cursor into uname,upro;
    insert into tb_user_pro values (null, uname, upro);
end while;
close u_cursor;
```

## 条件处理程序handler

用来定义在流程控制结构执行过程中遇到问题时相应的处理步骤

```
DECLARE handler_action HANDLER FOR condition_value [, condition_value] ... statement;
```

handler\_action

CONTINUE: 继续执行当前程序

EXIT: 终止执行当前程序

condition\_value

SQLSTATE sqlstate\_value: 状态码, 如 02000

SQLWARNING: 所有以01开头的SQLSTATE代码的简写

NOT FOUND: 所有以02开头的SQLSTATE代码的简写

SQLEXCEPTION: 所有没有被SQLWARNING 或 NOT FOUND捕获的SQLSTATE代码的简写

```
declare uname varchar(100);
declare upro varchar(100);
declare u_cursor cursor for select name,profession from tb_user where age <= uage;
declare exit handler for SQLSTATE '02000' close u_cursor;
```

## 存储函数

存储函数是有返回的存储过程 存储函数的参数只能是IN类型的

```
CREATE FUNCTION 存储函数名称 ([ 参数列表 ]) →  
RETURNS type [characteristic ...]  
BEGIN  
    -- SQL语句  
    RETURN ...;  
END ;
```

characteristic说明：

- DETERMINISTIC：相同的输入参数总是产生相同的结果
- NO SQL：不包含SQL语句。
- READS SQL DATA：包含读取数据的语句，但不包含写入数据的语句。

```
create function fun1(n int)
returns int deterministic
begin
    declare total int default 0;
    while n>0 do
        set total := total + n;
        set n := n - 1;
    end while;
    return total;
end;
```

## 触发器

触发器是与表有关的数据库对象 指在insert update delete之前或者之后 出发并执行触发器中定义的sql语句集合 触发器这种特性可以协助应用再数据端确保数据的完整性 日志记录 数据校验

使用old和new来引用触发器中发生变化的记录内容 只支持行级触发 不支持语句级触发

| 触发器类型       | NEW 和 OLD                         |
|-------------|-----------------------------------|
| INSERT 型触发器 | NEW 表示将要或者已经新增的数据                 |
| UPDATE 型触发器 | OLD 表示修改之前的数据 , NEW 表示将要或已经修改后的数据 |
| DELETE型触发器  | OLD 表示将要或者已经删除的数据                 |

➤ 创建

```
CREATE TRIGGER trigger_name  
BEFORE/AFTER INSERT/UPDATE/DELETE  
ON tbl_name FOR EACH ROW -- 行级触发器  
BEGIN  
    trigger_stmt;  
END;
```

➤ 查看

```
SHOW TRIGGERS;
```

➤ 删除

```
DROP TRIGGER [schema_name.]trigger_name; -- 如果没有指定 schema_name, 默认为当前数据库。
```

```
create trigger tb_user_insert_trigger  
    after insert on tb_user for each row  
begin  
    insert into user_logs(id, operation, operate_time, operate_id, operate_params) VALUES  
    (null, 'insert', now(), new.id, concat('插入的数据内容为: id=',new.id,',name=',new.name, ','))  
end;
```

## 锁

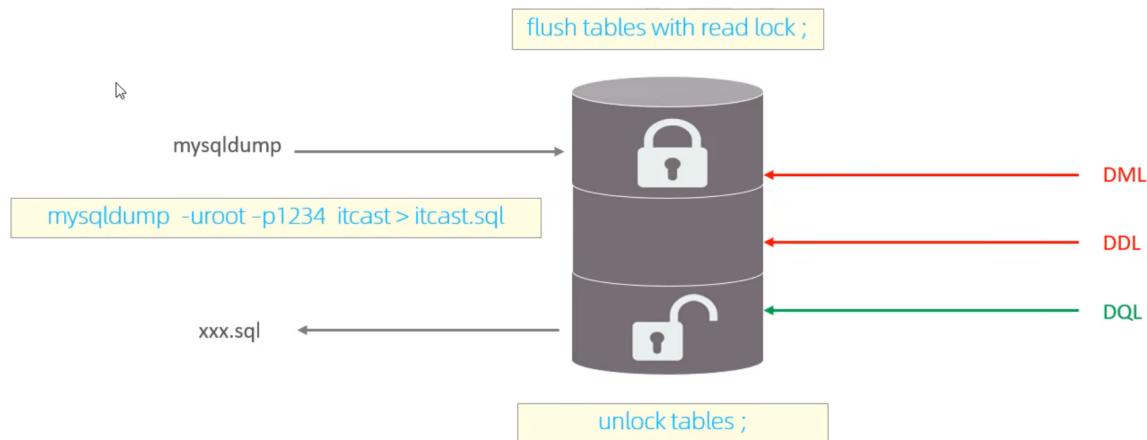
### 概述

- 全局锁 锁定数据库中所有表
- 表级锁 每次操作锁住整张表
- 行级锁 每次操作锁住对应的行数据

### 全局锁

全局锁就是对整个数据库实例加锁 加锁后整个实例就处于只读状态 后续的语句都将被阻塞

典型使用场景是对全库的逻辑备份 对所有的表锁定 从而获取一致性视图 保证数据的完整性



- 如果在主库上备份 那么在备份期间都不能执行更新 业务停摆
- 如果在从库上备份 那么备份期间从库不能执行主库同步过来的二进制日志 导致主从延迟
  - 在InnoDB中 可以在备份时加上参数--single-transaction参数来完成不加锁的一致性数据备份

## 表级锁

每次操作锁住整张表 锁级粒度大 发生锁冲突的概率最高 并发度最低

- 表锁
  - 表共享读锁
    - 不会阻塞其他客户端的读 但是会阻塞写
  - 表独占写锁
    - 既阻塞其他客户端的读 又阻塞其他客户端的写
- 元数据锁
  - 为避免增删改查和修改数据库结构语句的冲突
  - 系统自动控制 无需显式调用 在访问一张表的时候自动加上 MDL锁主要作用是维护元数据的数据一致性 在表上有活动事务得得时候 不可以对元数据进行写入操作
  - 在对一张表进行增删改查的时候 加MDL读锁 当对表结构进行变更的时候 加MDL写锁

| 对应SQL                                      | 锁类型                                     | 说明                                       |
|--------------------------------------------|-----------------------------------------|------------------------------------------|
| lock tables xxx read / write ↴             | SHARED_READ_ONLY / SHARED_NO_READ_WRITE |                                          |
| select、select ... lock in share mode       | SHARED_READ                             | 与SHARED_READ、SHARED_WRITE兼容，与EXCLUSIVE互斥 |
| insert、update、delete、select ... for update | SHARED_WRITE                            | 与SHARED_READ、SHARED_WRITE兼容，与EXCLUSIVE互斥 |
| alter table ...                            | EXCLUSIVE                               | 与其他的MDL都互斥                               |

- 意向锁

- 避免语句在执行时 加的表锁和行锁的冲突 意向锁使得表锁不用检查每行数据是否加锁 使用意向锁来减少表锁的检查
  - 意向共享锁 由语句select lock in share mode添加；‘
    - 与表锁共享锁兼容 与表锁排他锁互斥
  - 意向排他锁 由insert update delete select for update 添加
    - 与表锁共享锁及排他锁都互斥 意向锁之间不会互斥

- ```
mysql> select * from score where id = 1 lock in share mode;
```

```
mysql> select object_schema,object_name,index_name,lock_type,lock_mode,lock_data from performance_schema.data_locks;
+-----+-----+-----+-----+-----+-----+
| object_schema | object_name | index_name | lock_type | lock_mode | lock_data |
+-----+-----+-----+-----+-----+-----+
| db01          | score       | NULL        | TABLE     | IS          | NULL        |
| db01          | score       | PRIMARY     | RECORD    | S,REC_NOT_GAP | 1           |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

- select lock in share mode 增加了行锁和对表增加了意向锁

## 行级锁

InnoDB的数据是基于索引组织的 行锁是通过索引上的索引项加锁来实现的 而不是对记录加的锁

- 行锁 锁定单个行记录的锁 防止其他事务对此行进行update和delete 在RC RR隔离级别下都支持
- 间隙锁 锁定索引记录间隙 不包含该记录 确保索引记录间隙不变 防止其他事务在这个间隙进行 insert 产生幻读 在RR隔离级别下支持
- 临键锁 行锁和间隙锁组合 同时锁住数据 并锁住数据前面的间隙GAP 在RR下支持

InnoDB实现了两种类型的行锁

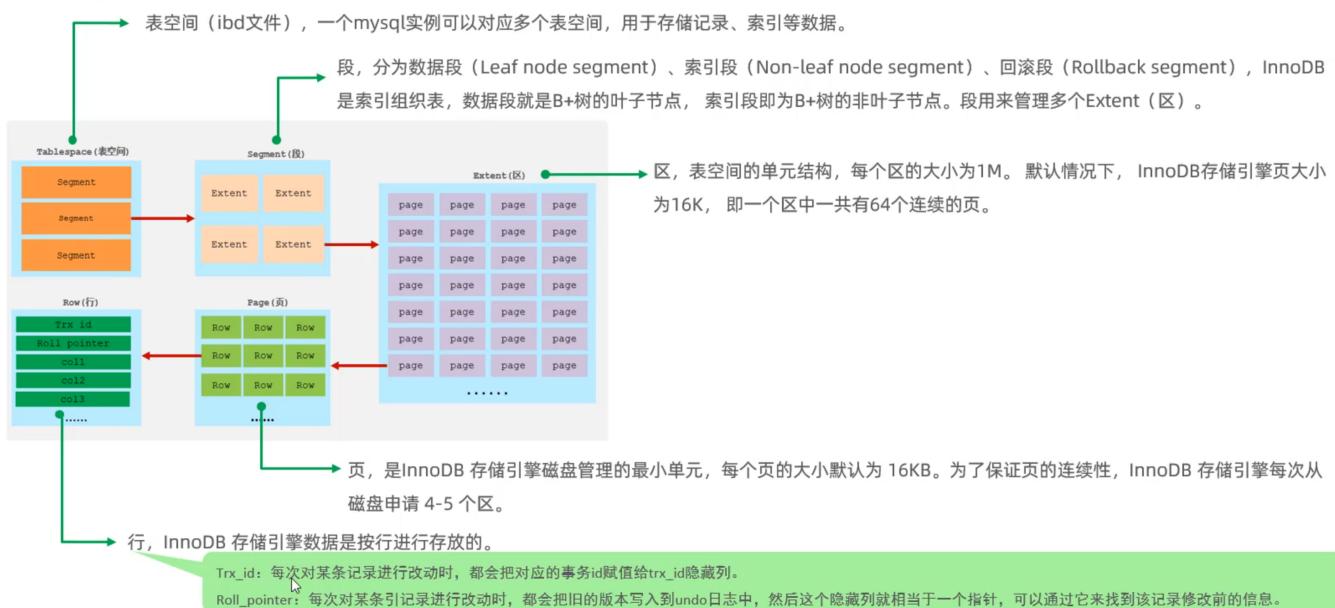
- 共享锁 允许一个事务去读一行 阻止其他事务获得相同数据的排他锁
- 排他锁 允许获取排他锁的事务更新数据 阻止其他事务获得相同数据集的共享锁和排他锁

请求锁类型 当前锁类型	S (共享锁)	X (排他锁)
S (共享锁)	兼容	冲突
X (排他锁)	冲突	冲突

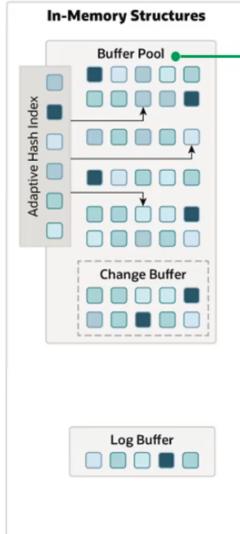
SQL	行锁类型	说明
INSERT ...	排他锁	自动加锁
UPDATE ...	排他锁	自动加锁
DELETE ...	排他锁	自动加锁
SELECT (正常)	不加任何锁	
SELECT ... LOCK IN SHARE MODE	共享锁	需要手动在SELECT之后加LOCK IN SHARE MODE
SELECT ... FOR UPDATE	排他锁	需要手动在SELECT之后加FOR UPDATE

- 默认情况下 InnoDB 使用临键锁进行搜索和索引扫描
  - 针对唯一索引进行检索的时候 对已经存在的记录进行等值匹配时 自动优化为行锁
  - InnoDB 的行锁是针对索引加的锁 不通过索引条件检索数据 那么InnoDB会对表中的所有数据加锁 升级为表锁
- 默认情况下InnoDB使用临键锁进行搜索和索引扫描
  - 索引上的等值查询 唯一索引 给不存在的记录加锁时 优化为间隙锁
  - 索引上的等值查询 普通索引 向右遍历时最后一个值不满足查询需求时 退化为间隙锁 会在记录前后数据都加锁 前加临键锁 后加间隙锁
  - 索引上的范围查询 唯一索引 会访问到不满足条件的第一个值为止 锁当前记录和之前一个数据的间隙 锁当前记录到正无穷之间的间隙
- 间隙锁锁的是间隙 不包含数据记录 临键锁既锁间隙也锁数据记录
- 间隙锁唯一的目的是防止其他事务插入间隙 间隙锁可以共存 一个事务采用的间隙锁不会组织另一个事务在同一间隙上采用间隙锁

## InnoDB引擎



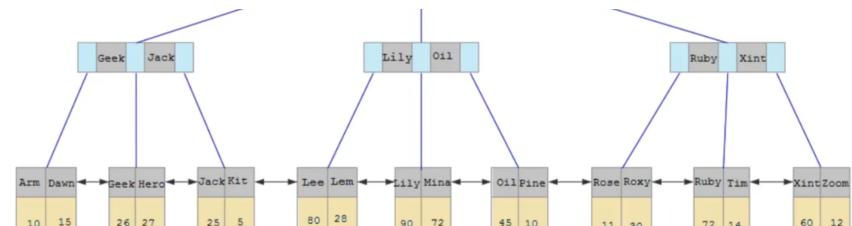
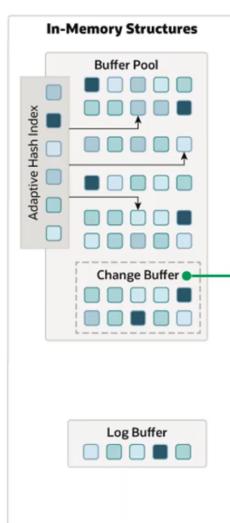
## 架构



Buffer Pool: 缓冲池是主内存中的一个区域，里面可以缓存磁盘上经常操作的真实数据，在执行增删改查操作时，先操作缓冲池中的数据（若缓冲池没有数据，则从磁盘加载并缓存），然后再以一定频率刷新到磁盘，从而减少磁盘IO，加快处理速度。

缓冲池以Page页为单位，底层采用链表数据结构管理Page。根据状态，将Page分为三种类型：

- free page: 空闲page，未被使用。
- clean page: 被使用page，数据没有被修改过。
- dirty page: 脏页，被使用page，数据被修改过，也中数据与磁盘的数据产生了不一致。



Change Buffer: 更改缓冲区（针对非唯一二级索引页），在执行DML语句时，如果这些数据Page没有在Buffer Pool中，不会直接操作磁盘，而会将数据变更存在更改缓冲区 Change Buffer 中，在未来数据被读取时，再将数据合并恢复到Buffer Pool中，再将合并后的数据刷新到磁盘中。

#### Change Buffer的意义是什么？

与聚集索引不同，二级索引通常是非唯一的，并且以相对随机的顺序插入二级索引。同样，删除和更新可能会影响索引树中不相邻的二级索引页，如果每一次都操作磁盘，会造成大量的磁盘IO。有了ChangeBuffer之后，我们可以在缓冲池中进行合并处理，减少磁盘IO。



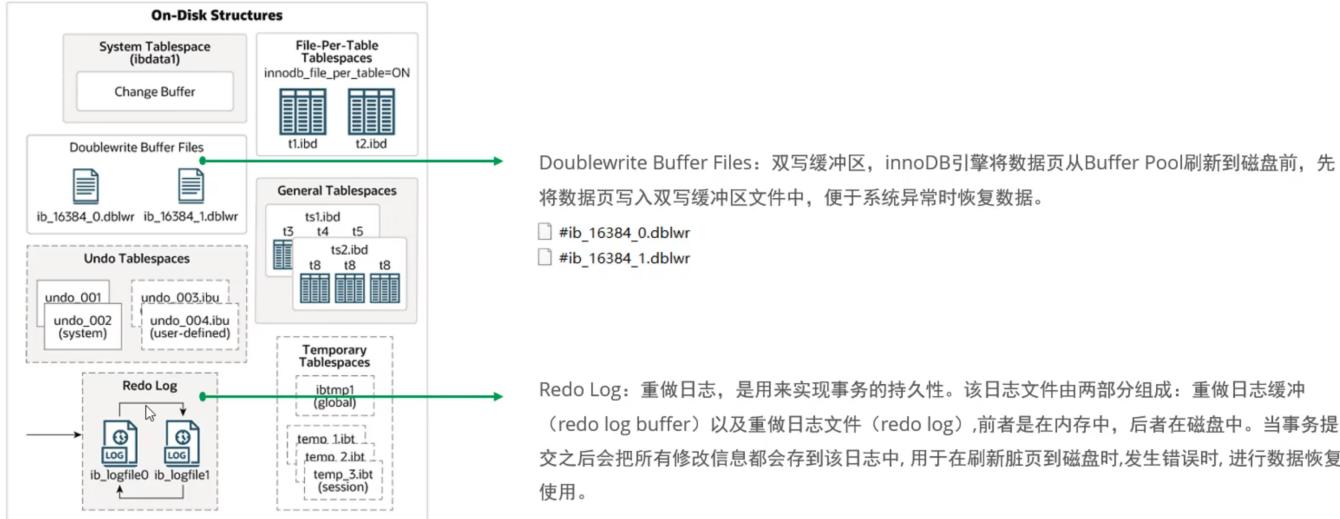
Adaptive Hash Index 自适应hash索引 用于优化对buffer pool数据的查询 InnoDB存储引擎会监控对表上各索引页的查询 如果观察到hash索引可以提升速度 则建立hash索引 称之为自适应hash索引

自适应hash索引 无需人工干预 是系统根据情况自动完成的

LogBuffer 日志缓冲区 用来保存需要写入到磁盘中的log日志数据 (redo log undo log) 默认大小为 16MB 日志缓冲区的日志会定期刷新到磁盘中 如果需要更新 插入 或者删除多行的事务 增加日志缓冲区的大小可以节省磁盘IO

innodb-flush-log-at-trx-commit 日志刷新到磁盘的时机

- 1 日志在每次事务提交时写入并刷新到磁盘
- 每秒将日志写入并刷新到磁盘一次
- 日志在每次事务提交后写入 并每秒刷新到磁盘一次



## 后台线程

- Master Thread 核心后台线程 负责调度其他线程 还负责将缓冲池中的数据异步刷新到磁盘中 保持数据的一致性 还包括脏页的刷新 合并插入缓存 undo页的回收
- IO Thread 在InnoDB存储引擎中大量使用了AIO（异步IO）来处理IO请求 提升数据库的性能 而IO Thread负责这些IO请求的回调

线程类型	默认个数	职责
Read thread	4	负责读操作
Write thread	4	负责写操作
Log thread	1	负责将日志缓冲区刷新到磁盘
Insert buffer thread	1	负责将写缓冲区内容刷新到磁盘

- Purge Thread 用于回收事务已经提交了的undo log 在事务提交之后 undo log可能不用了 就用它回收
- Page Cleaner Thread 协助Master Thread刷新脏页到磁盘的线程 可以减轻Master Thread的工作压力 减少阻塞

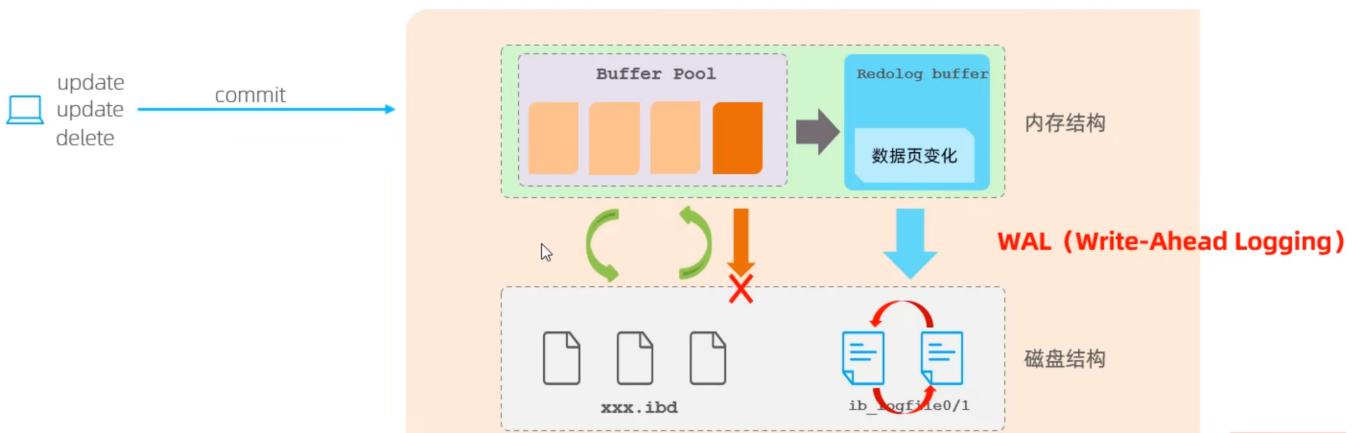
## 事务原理

redo log和undo log共同保证事务的一致性

### redo log 解决事务的持久性

重做日志 记录的是事务提交时数据页的物理修改 用来实现事务的持久性 日志文件分为两部分 重做日志缓冲以及重做日志文件 前者是在内存中 后者是在磁盘中 当事务提交之后会把所有修改信息都存到该日

志文件中 当刷新脏页到磁盘 发生错误时 进行数据恢复



## undo log 解决事务的原子性

回滚日志 用于记录数据被修改之前的信息 作用包括 提供回滚和MVCC (多版本并发控制)

undo log和redo log记录物理日志不一样 它是逻辑日志 可以认为 当delete一条记录时 undo log 会记录一条对应的insert记录 当update一条记录时 它记录一条对应相反的update记录 当执行roll back时 可以从undo log中的逻辑记录读取到相应的内容进行回滚

undo log销毁 undo log在事务执行时产生 事务提交时 并不会立即删除undo log 这些日志还可能用于 MVCC

undo log存储 undo log采用段的方式进行管理和记录 存放在rollback segment回滚段中 内部包含1024个undo log segment

## MVCC 解决事务的隔离性

- 当前读
  - 读取的是记录的最新版本 读取时还要保证其他并发事务不能修改当前记录 会对读取的记录进行加锁
    - select lock in share mode, select for update, update insert delete都是一种当前读
- 快照读
  - 简单的select就是快照读 快照读读取的是记录数据的可见版本 可能是历史数据 不加锁 是非阻塞读
    - Read Committed 每次select 都生成一个快照读
    - Repeatable Read 开启事务后 第一个select语句是快照读
    - Serializable 快照读会退化为当前读
- MVCC
  - 多版本并发控制 维护一个数据的多个版本 使得读写操作没有冲突 快照读为mysql实现MVCC提

供了一个非阻塞读功能 MVCC的实现需要依赖数据库记录中的三个隐式字段 undo log日志  
readView

- 记录中的隐藏字段

隐藏字段	含义
DB_TRX_ID	最近修改事务ID，记录插入这条记录或最后一次修改该记录的事务ID。
DB_ROLL_PTR	回滚指针，指向这条记录的上一个版本，用于配合undo log，指向上一个版本。
DB_ROW_ID	隐藏主键，如果表结构没有指定主键，将会生成该隐藏字段。

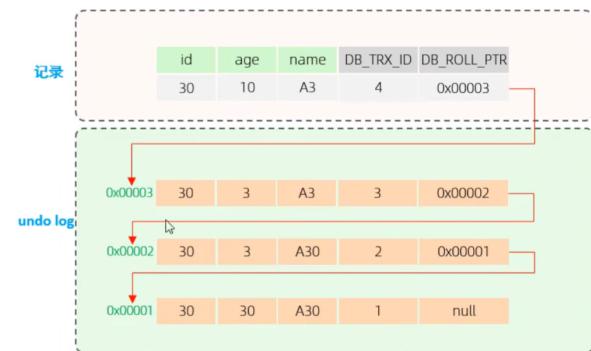
- undo log日志

- 当insert 的时候 产生的undolog 只在回滚时需要 在事务提交之后 可以直接删除
- 当update delete时 产生的undolog不仅在回滚时需要 在快照读时也需要 不会被立即删除

- undo log版本链

- undo log版本链

事务2	事务3	事务4	事务5
开始事务	开始事务	开始事务	开始事务
修改id为30记录 age改为3		查询id为30的记录	
提交事务	修改id为30记录 name改为A3		
		查询id为30的记录	
	提交事务	修改id为30记录 age改为10	
		查询id为30的记录	
			查询id为30的记录
			提交事务



- 不同事务或者相同事务对同一条记录进行修改 会导致该记录的undo log生成一条记录版本链表 链表的头部是最新的旧记录 链表的尾部是最早的旧记录

- readView 读视图

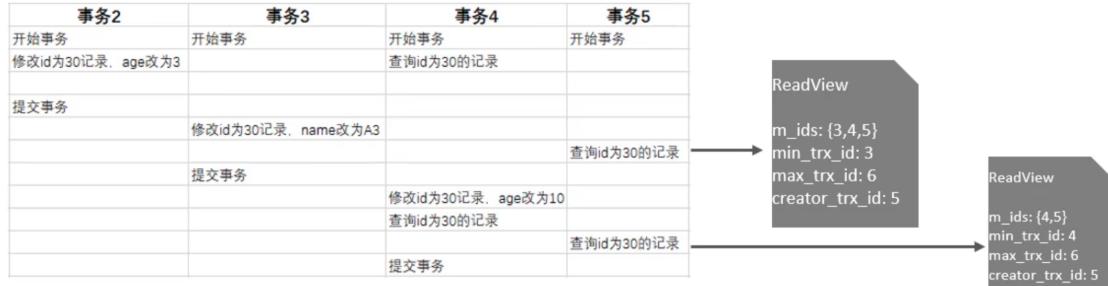
- 读视图是快照读SQL执行时MVCC提取数据的依据 记录并维护系统当前活跃的事务(未提交的)id
- 核心字段

字段	含义
m_ids	当前活跃的事务ID集合 ↓ 最小活跃事务ID
min_trx_id	
max_trx_id	预分配事务ID，当前最大事务ID+1 (因为事务ID是自增的)
creator_trx_id	ReadView创建者的事务ID

- readview



- READ COMMITTED 在事务中每一次执行快照读时生成ReadView



- REPEATABLE READ 仅在事务中第一次执行快照读时生成ReadView 后续复用该ReadView

