

SQL

SQL分类

- DDL 数据定义语言(Data Definition Language), 用来定义数据库对象 (数据库, 表, 字段)
- DML 数据操作语言(Data Manipulation Language), 用来对数据库表中的数据进行增删改
- DQL 数据查询语言(Data Query Language), 用来查询数据库中表的记录
- DCL 数据控制语言(Data Control Language), 用来创建数据库用户、控制数据库的访问权限

DDL

DDL-数据库操作

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
```

DDL-表操作

查询

查询当前数据库所有表

```
SHOW TABLES;
```

查询表结构

```
DESC 表名;
```

查询指定表的建表语句

```
SHOW CREATE TABLE 表名;
```

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];
```

修改

添加字段

```
ALTER TABLE 表名 ADD 字段名 类型 (长度) [COMMENT 注释] [约束];
```

修改数据类型

```
ALTER TABLE 表名 MODIFY 字段名 新数据类型(长度);
```

修改字段名和字段类型

```
ALTER TABLE 表名 CHANGE 旧字段名 新字段名 类型 (长度) [COMMENT 注释] [约束];
```

删除字段

```
ALTER TABLE 表名 DROP 字段名;
```

修改表名

```
ALTER TABLE 表名 RENAME TO 新表名;
```

删除

删除表

```
DROP TABLE [IF EXISTS] 表名;
```

删除指定表，并重新创建该表

```
TRUNCATE TABLE 表名;
```

DDL-数据类型

数值类型

类型	大小	有符号 (SIGNED) 范围	无符号 (UNSIGNED) 范围	描述
TINYINT	1byte	(-128,127)	(0,255)	小整数值
SMALLINT	2bytes	(-32768,32767)	(0,65535)	大整数值
MEDIUMINT	3bytes	(-8388608,8388607)	(0,16777215)	大整数值
INT或 INTEGER	4bytes	(-2147483648,2147483647)	(0,4294967295)	大整数值
BIGINT	8bytes	(-2 ⁶³ ,2 ⁶³ -1)	(0,2 ⁶⁴ -1)	极大整数值
FLOAT	4bytes	(-3.402823466E+38 - -1.175494351E-38)	0和(1.175494351E- 38 - 3.402823466E+38)	单精度浮 点数值
DOUBLE	8bytes			双精度浮 点数值
DECIMAL		依赖与M(精度)和(标度)的值	依赖与M(精度)和(标 度)的值	小数值 (精确定 点数)

字符串类型

类型	大小	描述
CHAR	0~255 bytes 0~2 ⁸ -1	定长字符串
VARCHAR	0~65535 bytes 0~2 ¹⁶ -1	变长字符串
TINYBLOB	0~255 bytes 0~2 ⁸ -1	不超过255个字符的二进制数据
TINYTEXT	0~255 bytes 0~2 ⁸ -1	短文本字符串
BLOB	0~65535 bytes 0~2 ¹⁶ -1	二进制形式的长文本数据
TEXT	0~65535 bytes 0~2 ¹⁶ -1	长文本数据
MEDIUMBLOB	0~16777215bytes 0~2 ²⁴ -1	二进制形式的中等长度文本数据
MEDIUMTEXT	0~16777215bytes 0~2 ²⁴ -1	中等长度文本数据
LOBLOB	0~4294967295 bytes 0~2 ³² -1	二进制形式的极大文本数据
LONGTEXT	0~4294967295 bytes 0~2 ³² -1	极大文本数据

日期类型

类型	大小 (byte)	范围	格式	描述
DATE	3	1000-01-01 至 9999-12-31	YYYY-MM-DD	日期值
TIME	3	-838:59:59 至 -838:59:59	HH:MM:SS	时间值或持续时间
YEAR	1	1901 至 2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00至 9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:01至 2038-01-19 03:14:07	YYYY-MM-DD HH:MM:SS	混合日期和时间值，时间戳

DML

DML-添加数据

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,...) ;
```

给全部字段添加数据

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

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,...);
```

注意:

- 插入数据时，指定的字段顺序需要与值的顺序是一一对应的
- 字符串和日期型数据应该包括在引号中
- 插入的数据大小，应该在字段的规定范围之内

DML-修改数据

```
UPDATE 表名 SET 字段名1=值1,字段名2=值2,...[WHERE 条件];
```

注意：修改语句的条件可以有，也可以没有，如果没有条件，则会修改整张表的所有数据

DML-删除数据

```
DELETE FROM 表名 [WHERE 条件]
```

注意:

- DELETE语句的条件可以有，也可以没有，如果没有条件，则会删除整张表的所有数据
- DELETE语句不能删除某一个字段的值(可以使用UPDATE)

DQL

- DQL语法

```
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DQL-基本查询
```

DQL-基本查询

查询多个字段

```
SELECT 字段1, 字段2, 字段3, ... FROM 表名;  
SELECT * FROM 表名;
```

设置别名

```
SELECT 字段1 [AS 别名1], 字段2 [AS 别名2], ... FROM 表名;
```

去除重复记录

```
SELECT DISTINCT 字段列表 FROM 表名;
```

DQL-条件查询

- 语法

```
SELECT  字段列表 FROM 表名 WHERE 条件列表；
```

- 条件

比较运算符	功能
>	大于
>=	大于等于
<	小于
<=	小于等于
=	等于
<>或!=	不等于
BETWEEN...AND...	在某个范围之内(含最小、最大值)
IN(...)	在in之后的列表中的值，多选一
LIKE 占位符	模糊匹配(_匹配单个字符，%匹配任意个字符)
IS NULL	是NULL

逻辑运算符	功能
AND 或 &&	并且(多个条件同时成立)
OR 或	或者(多个条件任意一个成立)
NOT 或 !	非 不是

DQL-聚合函数

- 将一列数据作为一个整体，进行纵向计算
- 常见聚合函数

函数	功能
count	统计数量
max	最大值
min	最小值
avg	平均值
sum	求和

DQL-分组查询

- 语法

```
SELECT 字段列表 FROM [WHERE 条件] GROUP BY 分组字段名 [HAVING 分组后过滤条件]
```

- where与having区别

1. 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。
2. 判断条件不同：where不能对聚合函数进行判断，而having可以

注意：

1. 执行顺序：where>聚合函数>having
2. 分组之后，查询的字段一般为聚合函数和分组字段，查询其他字段无任何意义

DQL-排序查询

- 语法

```
SELECT 字段列表 FROM 表名 ORDER BY 字段1 排序方式1, 字段2 排序方式2;
```

- 排序方式：ASC 升序（默认），DESC 降序

DQL-分页查询

- 语法

```
SELECT 字段列表 FROM 表名 LIMIT 起始索引, 查询记录数;
```

注意：

1. 起始索引从0开始，起始索引=(查询页面 - 1) * 每页显示记录数
2. 分页查询是数据库的方言，不同的数据库有不同的实现，MySQL中是LIMIT
3. 如果查询的是第一页数据，起始索引可以省略，直接简写为limit10

DQL-执行顺序

```
SELECT
    字段列表 ---4
FROM
    表名列表 ---1
WHERE
    条件列表 ---2
GROUP BY
    分组字段列表 ---3
HAVING
```

分组后条件列表

ORDER BY

排序字段列表 ---5

LIMIT

分页参数DQL-基本查询 ---6

DCL

DCL-管理用户

查询用户

```
USE mysql;  
SELECT * FROM user;
```

创建用户

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

修改用户密码

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

删除用户

```
DROP USER '用户名' @ '主机名'
```

注意:

- 主机名可以用%匹配
- 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

DCL-权限控制

权限	说明
ALL,ALL PRIVILEAGES	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

查询权限

```
SHOW GRANTS FOR '用户名' @ '主机名'
```

授予权限

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

撤销权限

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

函数

字符串函数

函数	功能
CONCATE(S1,S1,...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

数值函数

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

日期函数

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

- 流程函数是很常见的一类函数，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value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

概述

- 1. 概念：约束是作用于表中字段上的规则，用于限制存储在表中的数据
- 2. 目的：保证数据库中数据的正确、有效性和完整性
- 3. 分类：

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

注意：约束是作用于表中字段上的，可以在创建表/修改表的时候添加约束

外键约束

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)
```

删除外键

```
ALTER TABLE 表名 DROP FOREIGN KEY 外键名称
```

删除/更新行为

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(主表字段名) ON UPDATE CASCADE ON DELETE CASCADE;
```

多表查询

多表关系

1. 一对多（多对一）

实现：在多的的一方建立外键，指向一的一方的主键

2. 多对多

实现：建立第三张中间表，中间表至少包含两个外键，分别关联两方主键

3. 一对一

实现：在任意一方加入外键，关联另一方的主键，并且设置外键为唯一的（UNIQUE）

连接查询

内连接

隐式内连接

```
SELECT 字段列表 FROM 表1,表2 WHERE 条件...;
```

显式内连接

```
SELECT 字段列表 FROM 表1 [INNER] JOIN 表2 ON 连接条件...;
```

内连接查询的是两张表交集的部分

外连接

左外连接

```
SELECT 字段查询 FROM 表1 LEFT [OUTER] JOIN 表2 ON 条件...;
```

相当于查询表1(左表)的所有数据包含表1和表2交集部分的数据

右外连接

```
SELECT 字段列表 FROM 表1 RIGHT [OUTER] JOIN 表2 ON 条件...;
```

相当于查询表2(右表)的所有数据包含表1和表2交集部分的数据

自连接

自连接查询语法

```
SELECT 字段列表 FROM 表A 别名A JOIN 表A 别名B ON 条件...;
```

自连接查询，可以是内连接查询，也可以是外连接查询

联合查询

对于union查询，就是把多次查询的结果合并起来，形成一个新的查询结果集

```
SELECT 字段列表 FROM 表A...  
UNION [ALL]  
SELECT 字段列表 FROM 表B...;
```

对于联合查询的多张表的列数必须保持一致，字段类型也需要保持一致

union all会将全部的数据直接合并在一起，union会对合并之后的数据去重

子查询

概念：SQL语句中嵌套SELECT语句，称为**嵌套查询**，又称**子查询**

```
SELECT * FROM t1 WHERE column1=(SELECT column1 FROM t2)
```

子查询外部的语句可以是INSERT/UPDATE/DELETE/SELETC 的任何一个

根据子查询结果不同，分为：

- 标量子查询（子查询结果为单个值）

- 列子查询（子查询结果为一列）
- 行子查询（子查询结果为一行）
- 表子查询（子查询结果为多行多列）

根据子查询位置，分为：WHERE之后、FROM之后、SELECT之后

标量子查询

子查询返回的结果是单个值，（数字、字符串、日期等），最简单的形式，这种子查询称为**标量子查询**

常用的操作符：= <> > >= < <=

列子查询

子查询返回的结果是一列（可以是多行），这种子查询称为**列子查询**

常用的操作符：IN、NOT IN、ANY、SOME、ALL

操作符	描述
IN	在指定的集合范围之内，多选一
NOT IN	不在指定的集合范围之内
ANY	子查询返回列表中，有任意一个满足即可
SOME	与ANY等同，使用SOME的地方都可以用ANY
ALL	子查询返回列表的所有值都必须满足

行子查询

子查询返回的结果是一行（可以是多列），这种子查询称为**行子查询**

常用的操作符：=、<>、IN、NOT IN

表子查询

子查询返回的结果是多行多列，这种子查询称为**表子查询**

常用的操作符：IN

事务

简介

事务是一组操作的集合，它是一个不可分割的工作单位，事务会把所有的操作作为一个整体一起向系统提交或撤销操作请求，即**这些操作要么同时成功，要么同时失败。**

默认MySQL的事务是自动提交的，也就是说，当执行一条DML语句，MySQL会立即隐式的提交事务

事务操作

查看/设置事务提交方式

```
SELECT @@autocommit;
SELECT @@autocommit=1; #关闭事务自动提交
```

开启事务

```
START TRANSACTION 或 BEGIN ;
```

提交事务

```
COMMIT ;
```

回滚事务

```
ROLLBACK ;
```

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)：事务一旦提交或者回滚，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(读未提交)	允许	允许	允许
Read committed(读已提交)	禁止	允许	允许
Repeatable Read(可重复读)	禁止	禁止	允许
Serializable(可串行化)	禁止	禁止	禁止

```
-- 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 }
```

注意：事务隔离级别越高，数据越安全，但是性能越低

存储引擎

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
主要是将数据存储于文件系统之上，并完成与存储引擎的交互-

简介

存储引擎就是存储数据、建立索引、更新/查询数据等技术的实现方式。存储引擎是基于表的，而不是基于库的，所以存储引擎也可被称为表类型。

1. 在创建表时，指定存储引擎


```
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],
)ENGINE = INNODB [COMMENT 表注释];
```

2. 查看当前数据库支持的存储引擎

```
SHOW ENGINES;
```

存储引擎特点

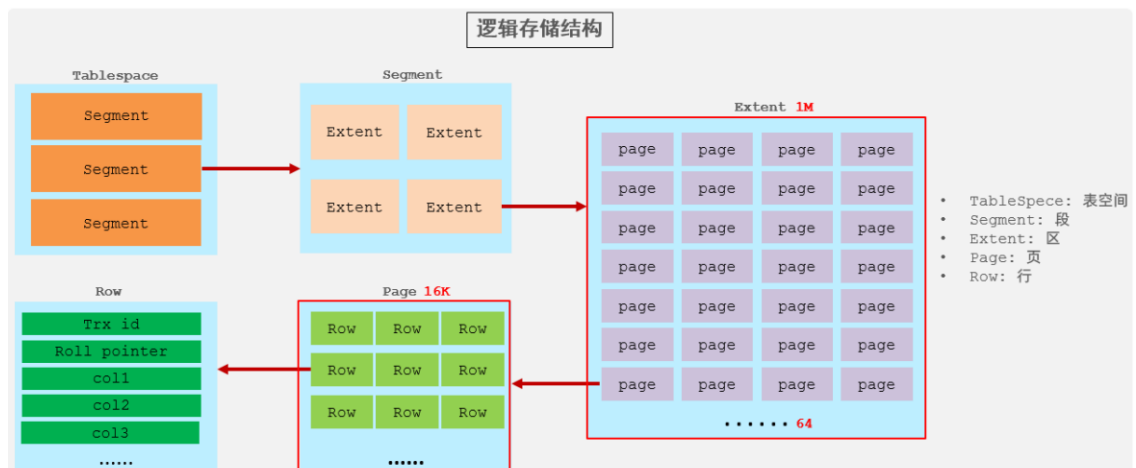
InnoDB

- 介绍：InnoDB是一种兼顾高可靠性和高性能的通用存储引擎，在MySQL5.5之后，InnoDB是默认的MySQL存储引擎。
- 特点
 1. DML操作遵循ACID模型，支持**事务**
 2. **行级锁**，提高并发访问性能
 3. 支持**外键** FOREIGN KEY约束，保证数据的完整性和正确性
- 文件

xxx.ibd: xxx代表的是表名，InnoDB引擎的每一张表都会对应这样一个表空间文件，存储该表的表结构(frm,sdi)。数据和索引

参数：innodb_file_per_table (默认打开，每一张表对应一个表空间文件)

- 逻辑存储结构



MyISAM

- 介绍：MyISAM是MySQL早期的默认存储引擎
- 特点
 1. 不支持事务，不支持外键

- 2. 支持**表锁**，不支持行锁
 - 3. 访问速度快
- 文件
 - xxx.sdi：存储表结构信息
 - xxx.MYD：存储数据
 - xxx.MYI：存储索引

Memory

- 介绍：Memory引擎的表数据是存储在内存中的，由于受到硬件问题，或断电问题的影响，只能将这些表作为临时表或缓存使用
- 特点
 - 1. 内存存放
 - 2. hash索引
- 文件
 - xxx.sdi：存储表结构信息

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	-	-

存储引擎选择

在选择存储引擎时，应该根据应用系统的特点选择合适的存储引擎。对于复杂的应用系统，还可以根据实际情况选择多种存储引擎进行组合。

- InnoDB: 是Mysql的默认存储引擎，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是非常合适的。
- MEMORY：将所有数据保存在内存中，访问速度快，通常用于临时表及缓存。MEMORY的缺陷就是对表的大小有限制，太大的表无法缓存在内存中，而且无法保障数据的安全性。

索引

概述

- 介绍
索引(index)是帮助MySQL**高效获取数据的数据结构（有序）**。在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法，这种数据结构就是索引。
- 优缺点

优势	劣势
提高数据检索的效率，降低数据库IO的成本	索引列也是要占用空间的
通过索引对数据进行排序，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

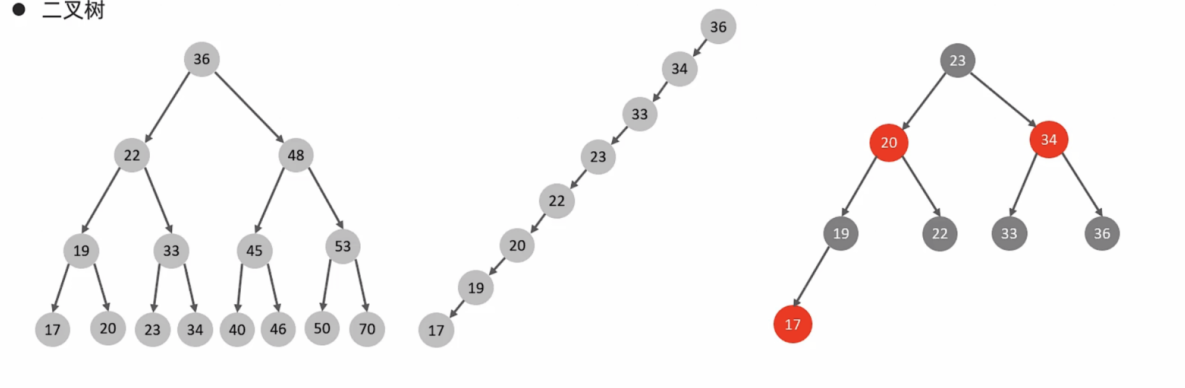
MySQL的索引是在存储引擎层实现的，不同的存储引擎有不同的结构，主要包含以下几种：

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+索引	支持	支持	支持
Hash索引	-	-	支持
R-tree索引	-	支持	-
Full-text	5.6版本之后支持	支持	-

二叉树

- 二叉树

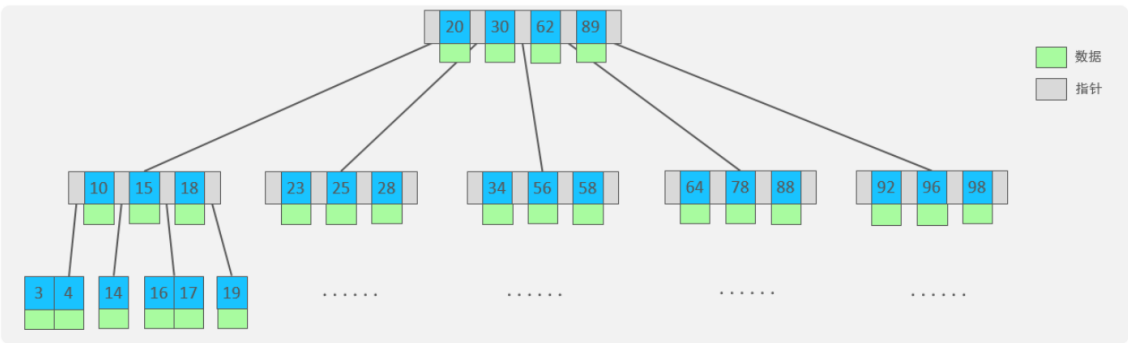


二叉树缺点：顺序插入时，会形成一个链表，查询性能大大降低。大数据量情况下，层级较深，检索速度慢。

红黑树：大数据量情况下，层级较深，检索速度慢

B-Tree

B-Tree，B树是一种多叉路衡查找树，相对于二叉树，B树每个节点可以有多个分支，即多叉。以一颗最大度数（max-degree）为5(5阶)的b-tree为例，那这个B树每个节点最多存储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。

插入一组数据： 100 65 169 368 900 556 780 35 215 1200 234 888 158 90 1000 88 120 268 250 。然后观察一些数据插入过程中，节点的变化情况。

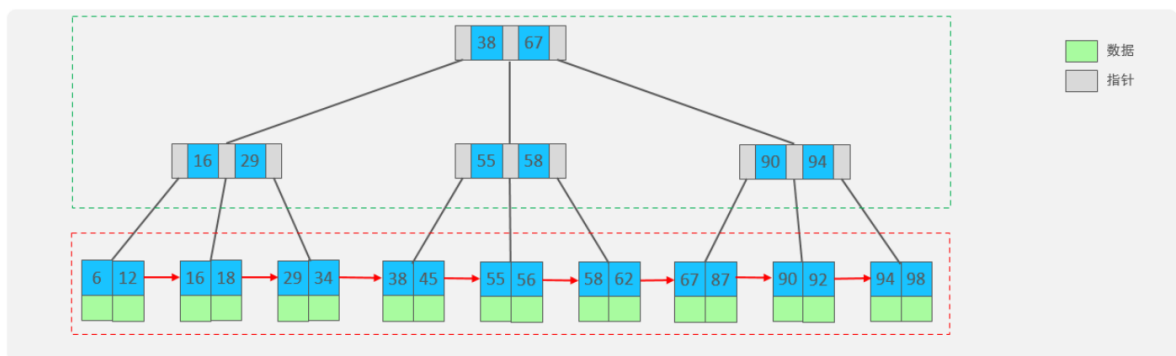


特点：

- 5阶的B树，每一个节点最多存储4个key，对应5个指针。
- 一旦节点存储的key数量到达5，就会裂变，中间元素向上分裂。
- 在B树中，非叶子节点和叶子节点都会存放数据

B+Tree

B+Tree是B-Tree的变种，我们以一颗最大度数（max-degree）为4（4阶）的b+tree为例，来看一下其结构示意图：



我们可以看到，两部分：

- 绿色框框起来的部分，是索引部分，仅仅起到索引数据的作用，不存储数据。
- 红色框框起来的部分，是数据存储部分，在其叶子节点中要存储具体的数据。

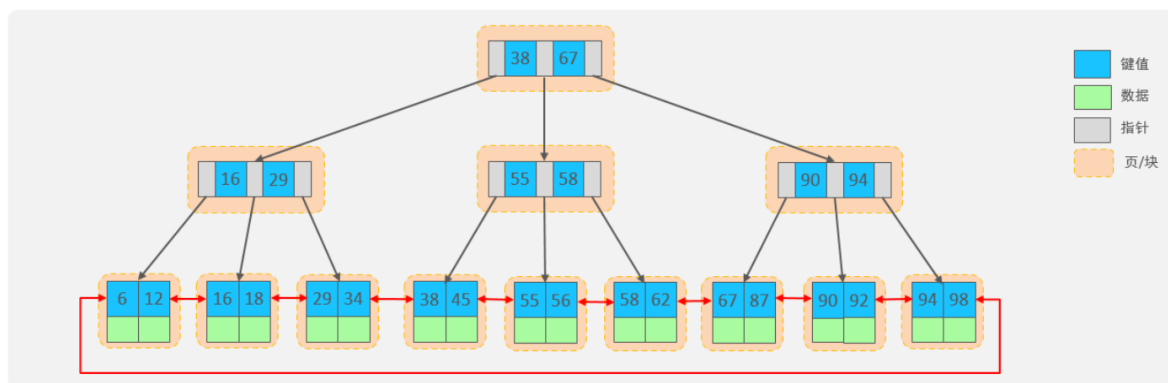
插入一组数据： 100 65 169 368 900 556 780 35 215 1200 234 888 158 90 1000 88 120 268 250 。然后观察一些数据插入过程中，节点的变化情况。



最终我们看到，B+Tree 与 B-Tree相比，主要有以下三点区别：

- 所有的数据都会出现在叶子节点。
- 叶子节点形成一个单向链表。
- 非叶子节点仅仅起到索引数据作用，具体的数据都是在叶子节点存放的。

MySQL索引数据结构对经典的B+Tree进行了优化。在原B+Tree的基础上，增加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+Tree，提高区间访问的性能，利于排序。

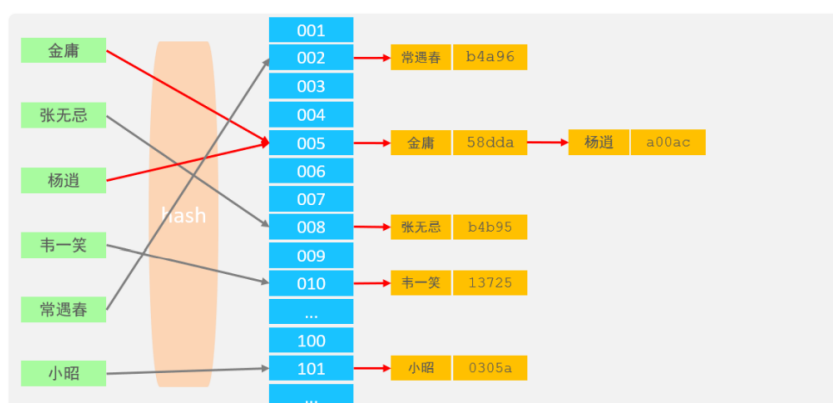


Hash

哈希索引就是采用一定的hash算法，将键值换算成新的hash值，映射到对应的槽位上，然后存储在hash表中。

如果两个（或多个）键值映射一个相同的槽位上，他们就产生了hash冲突（也称为hash碰撞），可以通过链表来解决。

	id	name	age
58dda	1	金庸	36
b4b95	2	张无忌	22
a00ac	3	杨逍	33
13725	4	韦一笑	48
b4a96	5	常遇春	53
0305a	6	小昭	19
48c00	7	灭绝	45
f2d22	8	周芷若	17
e29fd	9	丁敏君	23
a7b7d	10	赵敏	20
4af6d	11	鹿杖客	49
00a22	12	鹤笔翁	60



Hash索引特点：

1. Hash索引只能用于对等 (=, in)，不支持范围查询 (between, >, <, ...)
2. 无法利用索引完成排序操作
3. 查询效率高，通常只需要一次检索就可以了，效率通常高于B+tree索引

存储引擎支持：

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

思考题：为什么InnoDB存储引擎选择使用B+tree索引结构？

- 相对于二叉树，层级更少，搜索效率高；
- 对于B-Tree，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，只能增加树的高度，导致性能降低；
- 相对于Hash索引，B+Tree支持范围匹配及排序操作

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字，而不是比较索引中的值。	可以有多个	FULLTEXT

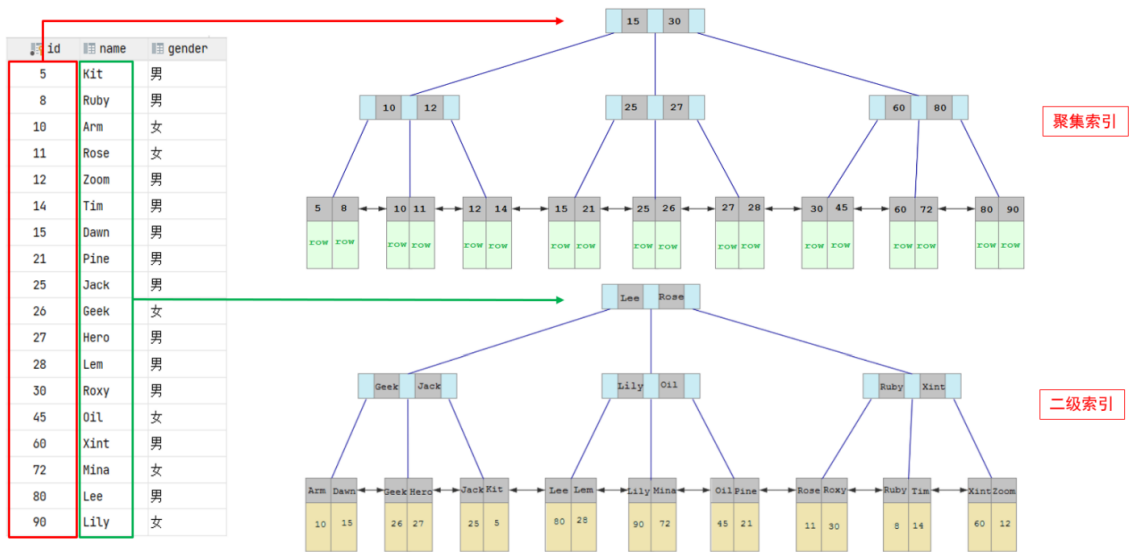
在InnoDB存储引擎中，根据索引的存储形式，又可以分为以下两种：

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，且只有一个
二级索引(Secondary Index)	将数据与索引	可以存在多个

聚集索引选取规则：

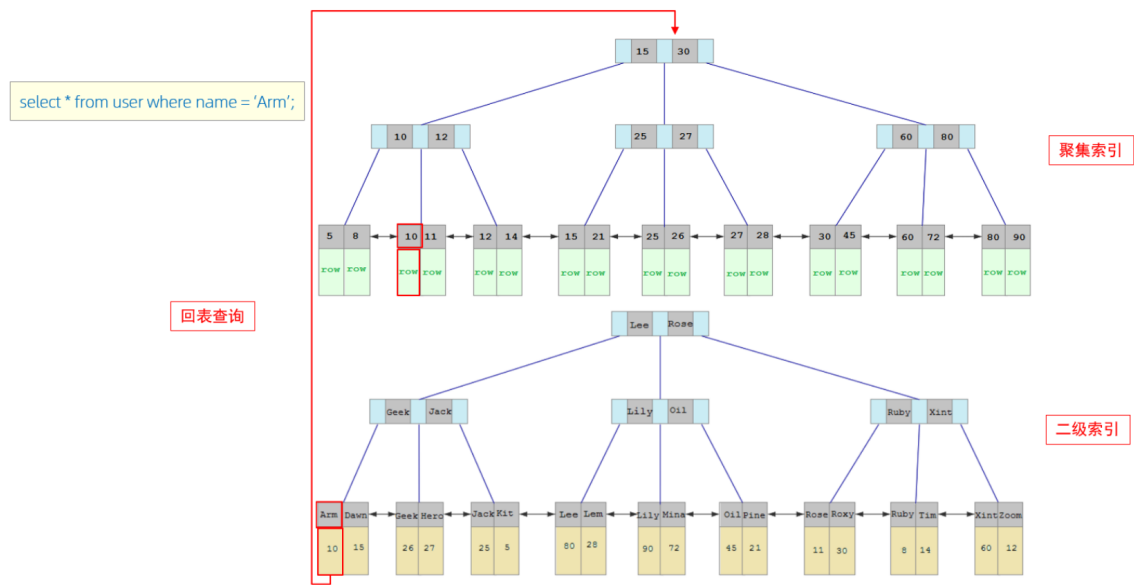
- 如果存在主键，主键索引就是聚集索引
- 如果不存在主键，将使用第一个唯一（UNIQUE）索引作为聚集索引
- 如果表没有主键，或没有合适的唯一索引，则InnoDB自动生成一个rowid作为隐藏的聚集索引

聚集索引和二级索引的具体结构如下：



- 聚集索引的叶子节点下挂的是这一行的数据。
- 二级索引的叶子节点下挂的是该字段值对应的主键值。

接下来，我们分析一下，当我们执行如下的SQL语句时，具体的查找过程是什么样子的。



回表查询： 这种先到二级索引中查找数据，找到主键值，然后再到聚集索引中根据主键值，获取数据的方式，就称之为回表查询。

思考题： 以下两条SQL语句，那个执行效率高? 为什么?

A. select * from user where id = 10 ; B. select * from user where name = 'Arm' ;

备注: id为主键， name字段创建的有索引；

解答： A 语句的执行性能要高于B 语句。 因为A语句直接走聚集索引，直接返回数据。 而B语句需要先查询name字段的二级索引， 然后再查询聚集索引， 也就是需要进行回表查询。

思考题： InnoDB主键索引的B+tree高度为多高呢?

假设: 一行数据大小为1k， 一页中可以存储16行这样的数据。 InnoDB的指针占用6个字节的空间， 主键即使为bigint， 占用字节数为8。

高度为2: $n * 8 + (n + 1) * 6 = 16 * 10 * 24$, 算出n约为 1171, $1171 * 16 = 18736$ 也就是说， 如果树的高度为2， 则可以存储 18000 多条记录。

高度为3: $1171 * 1171 * 16 = 21939856$ 也就是说， 如果树的高度为3， 则可以存储 2200w 左右的记录。

索引语法

创建索引

```
CREATE [UNIQUE | FULLTEXT] INDEX 索引名 ON 表名(列1, 列2, ...)
```

查看索引

```
SHOW INDEX FROM 表名;
```



```
DROP INDEX 索引名 ON 表名;
```

SQL性能分析

SQL执行频率

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|SESSION] STATUS LIKE 'Com_____';
```

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，我们可以查看一下系统变量 `slow_query_log`。如果要开启慢查询日志，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
```

配置完毕之后，通过以下指令重新启动MySQL服务器进行测试，查看慢日志文件中记录的信息
`/var/lib/mysql/localhost-slow.log`。

profile详情

`show profiles` 能够在做SQL优化时帮助我们了解时间都耗费到哪里去了。通过`have_profiling` 参数，能够看到当前MySQL是否支持profile操作：

```
SELECT @@have_profiling ;
```

可以通过set语句在 session/global级别开启profiling：

```
SET profiling = 1;
```

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;
```

explain执行计划

EXPLAIN 或者 DESC命令获取 MySQL 如何执行 SELECT 语句的信息，包括在 SELECT 语句执行 过程中表如何连接和连接的顺序。

语法

```
-- 直接在select语句之前加上关键字 explain / desc
EXPLAIN SELECT 字段列表 FROM 表名 WHERE 条件 ;
```

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。

字段	含义
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。

索引使用规则

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。

最左前缀法则指的是查询从索引的最左列开始，并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）。

注意：最左前缀法则中指的是最左边的列，是指在查询时，联合索引的最左边的字段(即是 第一个字段)必须存在，与我们编写SQL时，条件编写的先后顺序无关。

范围查询

联合索引中，出现范围查询（>,<），范围查询右侧的列索引失效

注意：当范围查询使用>= 或 <= 时，走联合索引，所以，在业务允许的情况下，尽可能的使用类似于 >= 或 <= 这类的范围查询，而避免使用 > 或 <。

索引列运算

不要在索引列上进行运算操作，索引将失效。

字符串不加引号

字符串类型字段使用时，不加引号，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，索引将失效。

or连接的条件

用or分割开的条件，如果or前的条件中的列有索引，而后面的列中没有索引，那么**涉及的索引都不会被用到**。

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

SQL提示，是优化数据库的一个重要手段，简单来说，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：建议MySQL使用哪一个索引完成此次查询（仅仅是建议，mysql内部还会再次进行评估）。

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

force index：强制使用索引。

```
explain select * from tb_user force index(idx_user_pro) where profession = '软件  
工  
程';
```

ignore index：忽略指定的索引

```
explain select * from tb_user ignore 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*。

Extra	含义
Using where; Using Index	查找使用了索引，但是需要的数据都在索引列中能找到，所以不需 要回表查询数据
Using index condition	查找使用了索引，但是需要回表查询数据

思考题：

一张表, 有四个字段(id, username, password, status), 由于数据量大, 需要对以下SQL语句进行优化, 该如何进行才是最优方案: select id,username,password from tb_user where username = 'itcast';

答案: 针对于 username, password建立联合索引, sql为: create index idx_user_name_pass on tb_user(username,password);

这样可以避免上述的SQL语句, 在查询的过程中, 出现回表查询。

前缀索引

当字段类型为字符串 (varchar, text, longtext等) 时, 有时候需要索引很长的字符串, 这会让索引变得很大, 查询时, 浪费大量的磁盘IO, 影响查询效率。此时可以只将字符串的一部分前缀, 建立索引, 这样可以大大节约索引空间, 从而提高索引效率。

语法:

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

-- 示例: 为tb_user表的email字段, 建立长度为5的前缀索引。

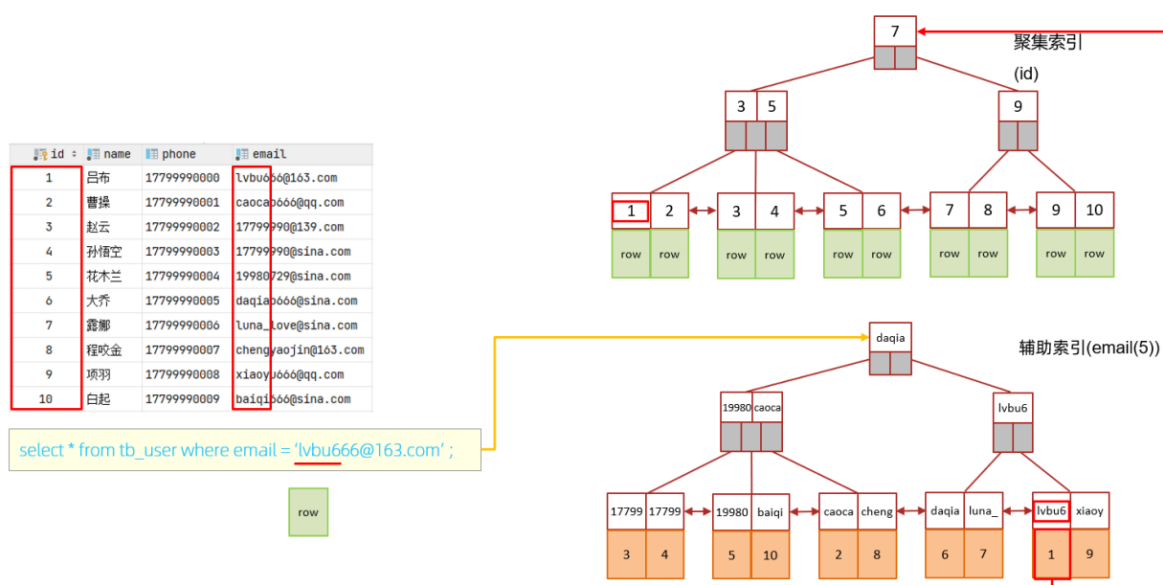
```
create index idx_email_5 on tb_user(email(5));
```

前缀长度:

可以根据索引的选择性来决定, 而选择性是指不重复的索引值 (基数) 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, 唯一索引的选择性是1, 这是最好的索引选择性, 性能也是最好的。

```
select count(distinct email) / count(*) from tb_user ;  
select count(distinct substring(email,1,5)) / count(*) from tb_user ;
```

前缀索引的查询流程:



单列索引&联合索引

单列索引：即一个索引只包含单个列。

联合索引：即一个索引包含了多个列。

在业务场景中，如果存在多个查询条件，考虑针对于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

1. 针对数据量较大，且查询比较频繁的表建立索引。
2. 针对于常作为查询条件（where）、排序（order by）、分组（group by）操作的字段建立索引。
3. 尽量选择区分度高的列作为索引，尽量建立唯一索引，区分度越高，使用索引的效率越高。
4. 如果是字符串类型的字段，字段的长度较长，可以针对于字段的特点，建立前缀索引。
5. 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。
6. 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。
7. 如果索引列不能存储NULL值，请在创建表时使用NOT NULL约束它。当优化器知道每列是否包含NULL值时，它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

- 批量插入
- 手动提交事务
- 主键顺序插入
- 大批量插入数据

如果一次性需要插入大批量数据，使用insert语句插入性能较低，此时可以使用MySQL数据库提供的load指令进行插入。

```
-- 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/sql1.log' into table tb_user fields
terminated by ',' lines terminated by '\n' ;
```

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，这种存储方式的表称为索引组织表(index organized table IOT)。

- 页分裂

页可以为空，也可以填充一半，也可以填充100%。每个页包含了2-N行数据(如果一行数据过大，会行溢出)，根据主键排列。

- 页合并

当删除一行记录时，实际上记录并没有被物理删除，只是记录被标记（flagged）为删除并且它的空间变得允许被其他记录声明使用。

当页中删除的记录达到 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以优化空间使用。

MERGE_THRESHOLD：合并页的阈值，可以自己设置，在创建表或者创建索引时指定。

- 主键设计原则：

1. 满足业务需求的情况下，尽量降低主键的长度
2. 插入数据时，尽量选择顺序插入，或者使用AUTO_INCREMENT自增主键
3. 尽量不要使用UUID做主键或者其他自然主键，如身份证号
4. 业务操作时，避免对主键的修改。

order by优化

- Using filesort : 通过表的索引或全表扫描，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 FileSort 排序。
- Using index : 通过有序索引顺序扫描直接返回有序数据，这种情况即为 using index，不需要 额外排序，操作效率高。

对于以上的两种排序方式，Using index的性能高，而Using filesort的性能低，我们在优化排序 操作时，尽量要优化为 Using index。

order by 优化原则：

- 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（ASC/DESC）。
- 如果不可避免的出现filesort，大数据量排序时，可以适当增大排序缓冲区大小sort_buffer_size（默认256k）

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

在数据量比较大时，如果进行limit分页查询，在查询时，越往后，分页查询效率越低。

当在进行分页查询时，如果执行 `limit 2000000,10`，此时需要MySQL排序前2000010记录，仅仅返回2000000 - 2000010的记录，其他记录丢弃，查询排序的代价非常大。

优化思路: 一般分页查询时，通过创建 **覆盖索引** 能够比较好地提高性能，可以通过**覆盖索引加子查询形式**进行优化。

```
explain select * from tb_sku t , (select id from tb_sku order by id
limit 2000000,10) a where t.id = a.id;
```

count优化

- MyISAM 引擎把一个表的总行数存在了磁盘上，因此执行 `count(*)` 的时候会直接返回这个数，效率很高；但是如果是带条件的count，MyISAM也慢。
- InnoDB 引擎就麻烦了，它执行 `count(*)` 的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

如果说要大幅度提升InnoDB表的count效率，**主要的优化思路：自己计数(可以借助于redis这样的数据库进行,但是如果是带条件的count又比较麻烦了)。**

count的几种用法：

`count()` 是一个聚合函数，对于返回的结果集，一行行地判断，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。

用法：`count (*)`、`count (主键)`、`count (字段)`、`count (数字)`

count用法	含义
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。服务层拿到主键后，直接按行进行累加(主键不可能为null)
count(字段)	没有not null 约束：InnoDB 引擎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 约束：InnoDB 引擎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(数字)	InnoDB 引擎遍历整张表，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引擎并不会把全部字段取出来，而是专门做了优化，不取值，服务层直接按行进行累加。

按照效率排序的话，`count(字段) < count(主键 id) < count(1) ≈ count(*)`，所以尽量使用 `count(*)`。

update优化

InnoDB的行锁是针对索引加的锁，不是说针对记录加的锁，并且该索引不能失效，否则会行锁升级为表锁。

我们主要需要注意一下update语句执行时的注意事项。

```
1  update  course  set  name = 'javaEE'  where  id  =  1  ;
```

当我们在执行删除的SQL语句时，会锁定id为1这一行的数据，然后事务提交之后，行锁释放。

但是当我们在执行如下SQL时。

```
1  update  course  set  name = 'SpringBoot'  where  name = 'PHP'  ;
```

当我们开启多个事务，在执行上述的SQL时，我们发现行锁升级为了表锁。 导致该update语句的性能大大降低。

视图

介绍

视图（view）是一种虚拟存在的表。视图中的数据并不在数据库中实际存在，行和列数据来自定义视图中查询中用的表，并且是在使用视图时动态生成的。

通俗的讲，视图只保存了查询的SQL逻辑，不保存查询结果。所以我们在创建视图的时候，主要的工作就落在创建这条SQL查询语句上。

基本语法

- 创建

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

- 查询

```
-- 查看创建视图语句  
SHOW CREATE VIEW 视图名称;  
-- 查看视图数据  
SELECT * FROM 视图名称...;
```

- 修改

```
-- 方式一  
CREATE OR REPLACE VIEW 视图名称[(列名列表)] AS SELECT 语句[WITH[CASCADE|LOCAL]]  
CHECK OPTION]  
-- 方式二  
ALTER VIEW 视图名称[(列名列表)] AS SELECT 语句[WITH[CASCADE|LOCAL]] CHECK  
OPTION]
```

- 删除

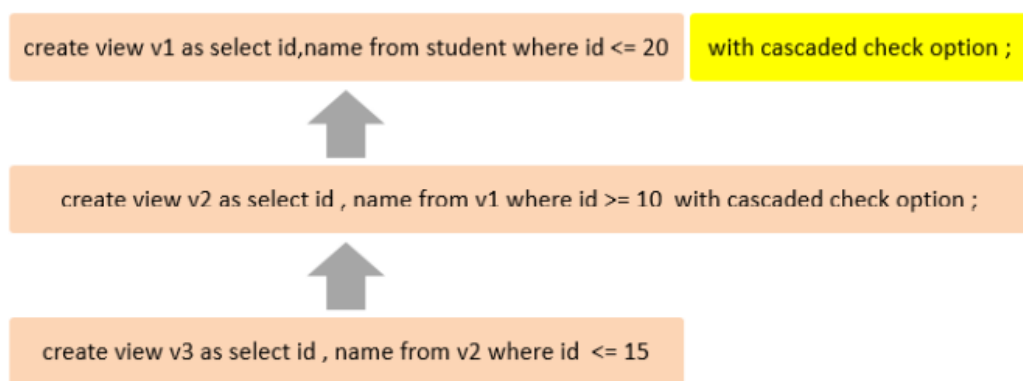
```
DROP VIEW [IF EXISTS] 视图名称 [,视图名称...]
```

视图的检查选项

当使用WITH CHECK OPTION子句创建视图时，MySQL会通过视图检查正在更改的每个行，例如 插入，更新，删除，以使其符合视图的定义。MySQL允许基于另一个视图创建视图，它还会检查依赖视图中的规则以保持一致性。为了确定检查的范围，mysql提供了两个选项：CASCADDED 和 LOCAL，默认值为CASCADDED。

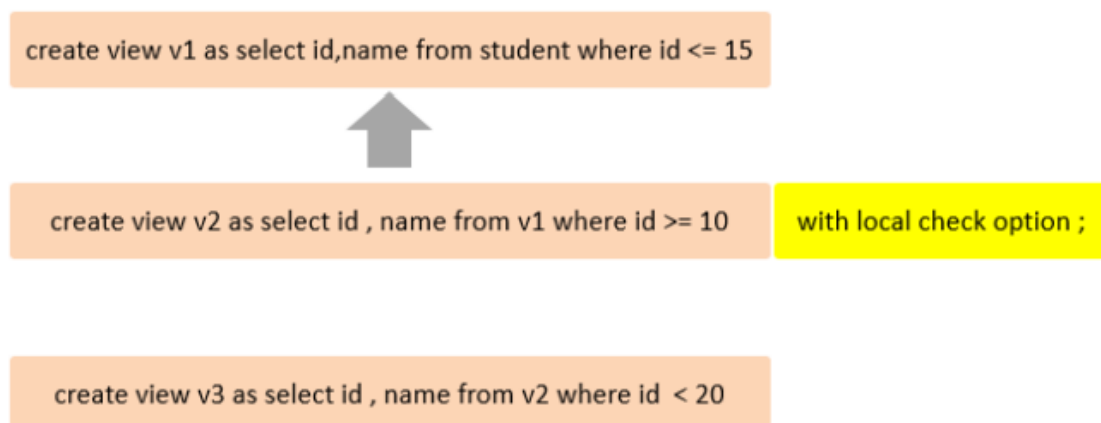
CASCADE

级联。比如，v2视图是基于v1视图的，如果在v2视图创建的时候指定了检查选项为 cascaded，但是v1视图创建时未指定检查选项。则在执行检查时，不仅会检查v2，还会级联检查v2的关联视图v1。



LOCAL

本地。比如，v2视图是基于v1视图的，如果在v2视图创建的时候指定了检查选项为 local，但是v1视图创建时未指定检查选项。则在执行检查时，只会检查v2，不会检查v2的关联视图v1。



视图的更新

要使视图可更新，视图中的行与基础表中的行之间必须存在一对一的关系。如果视图包含以下任何一项，则该视图不可更新：

1. 聚合函数或窗口函数 (SUM()、MIN()、MAX()、COUNT()等)
2. DISTINCT
3. GROUP BY
4. HAVING
5. UNION 或者 UNION ALL

视图作用

- 简单

视图不仅可以简化用户对数据的理解，也可以简化他们的操作。那些经常被使用的查询可以被定义为视图，从而使得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，但不能授权到数据库指定行和特定的列上。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存储在数据库中的一段SQL语句的集合，调用存储过程可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是有好处的。

存储过程思想上很简单，就是数据库SQL语言层面的代码封装与重用。

特点：

- 封装，复用(可以把某一业务SQL封装在存储过程中，需要用到 的时候直接调用即可。)
- 可以接收参数，也可以返回数据(在存储过程中，可以传递参数，也可以接收返回 值。)
- 减少网络交互，效率提升(如果涉及到多条SQL，每执行一次都是一次网络传 输。而如果封装在存储过程中，我们只需要网络交互一次可能就可以了。)

基本语法

- 创建

```
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 ;
```

注意：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

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（GLOBAL）、会话变量（SESSION）。

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 ;
```

注意：

- 如果没有指定SESSION/GLOBAL，默认是SESSION，会话变量。
- mysql服务重新启动之后，所设置的全局参数会失效，要想不失效，可以在 `/etc/my.cnf` 中配置。

用户定义变量

用户定义变量 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" 使用就可以。其作用域为当前连接。

赋值：

方式一

```
SET @var_name = expr [, @var_name = expr] ... ;
SET @var_name := expr [, @var_name := expr] ... ;
```

赋值时，可以使用 `=`，也可以使用 `:=`。

方式二:

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

使用:

```
SELECT @var_name ;
```

注意: 用户定义的变量无需对其进行声明或初始化, 只不过获取到的值为NULL。

局部变量

局部变量 是根据需要定义的在局部生效的变量, 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 ... ;
```

if判断

if 用于做条件判断, 具体的语法结构为:

```
IF 条件1 THEN  
.....  
ELSEIF 条件2 THEN -- 可选  
.....  
ELSE -- 可选  
.....  
END IF;
```

在if条件判断的结构中, ELSE IF 结构可以有多个, 也可以没有。ELSE结构可以有, 也可以没有。

参数

参数的类型, 主要分为以下三种: IN、OUT、INOUT。具体的含义如下:

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 ;
```

案例一：

根据传入参数score，判定当前分数对应的分数等级，并返回。

```
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;
-- 定义用户变量 @result来接收返回的数据，用户变量可以不用声明
call p4(18, @result);
select @result;
```

案例二：

将传入的200分制的分数，进行换算，换算成百分制，然后返回。

```
create procedure p5(inout score double)
begin
    set score := score * 0.5;
end;
set @score = 198;
call p5(@score);
select @score;
```

case

语法1：

```
-- 含义： 当case_value的值为 when_value1时，执行statement_list1，当值为 when_value2
时，
-- 执行statement_list2， 否则就执行 statement_list
CASE case_value
    WHEN when_value1 THEN statement_list1
    [ WHEN when_value2 THEN statement_list2 ] ...
    [ ELSE statement_list ]
END CASE;
```

语法2:

```
-- 含义： 当条件search_condition1成立时，执行statement_list1，当条件search_condition2
成
立时，执行statement_list2， 否则就执行 statement_list
CASE
    WHEN search_condition1 THEN statement_list1
    [WHEN search_condition2 THEN statement_list2] ...
    [ELSE statement_list]
END CASE;
```

注意：如果判定条件有多个，多个条件之间，可以使用 and 或 or 进行连接。

循环

while

while 循环是有条件的循环控制语句。满足条件后，再执行循环体中的SQL语句。具体语法为：

```
-- 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

repeat是有条件的循环控制语句, 当满足until声明的条件的时候，则退出循环。具体语法为：

```
-- 先执行一次逻辑，然后判定UNTIL条件是否满足，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

LOOP 实现简单的循环，如果不在SQL逻辑中增加退出循环的条件，可以用其来实现简单的死循环。
LOOP可以配合一下两个语句使用：

- LEAVE：配合循环使用，退出循环。
- ITERATE：必须用在循环中，作用是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
```

上述语法中出现的 begin_label, end_label, label 指的都是我们所自定义的标记。

游标cursor

游标（CURSOR）是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，其语法分别如下。

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 ；
```

案例:

根据传入的参数uage，来查询用户表tb_user中，所有的用户年龄小于等于uage的用户姓名（name）和专业（profession），并将用户的姓名和专业插入到所创建的一张新表(id,name,profession)中。

```
-- 逻辑：
-- A. 声明游标，存储查询结果集
-- B. 准备：创建表结构
-- C. 开启游标
-- D. 获取游标中的记录
-- E. 插入数据到新表中
-- F. 关闭游标
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;
end;
call p11(30);

```

上述的存储过程，最终我们在调用的过程中，会报错，之所以报错是因为上面的while循环中，并没有退出条件。当游标的数据集获取完毕之后，再次获取数据，就会报错，从而终止了程序的执行。

要想解决这个问题，就需要通过MySQL中提供的 **条件处理程序 Handler** 来解决。

条件处理程序handler

条件处理程序（Handler）可以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。具体语法为：

```

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

```

handler_action 的取值：

CONTINUE：继续执行当前程序

EXIT：终止执行当前程序

condition_value 的取值：

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

具体的错误状态码，可以参考官方文档：

<https://dev.mysql.com/doc/refman/8.0/en/declare-handler.html>

<https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html>

上一节案例补充:

```

-- 逻辑：
-- A. 声明游标，存储查询结果集
-- B. 准备：创建表结构
-- C. 开启游标
-- D. 获取游标中的记录
-- E. 插入数据到新表中
-- F. 关闭游标
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;

    -- 声明条件处理程序：当SQL语句执行抛出的状态码为02000时，将关闭游标u_cursor，并退出
    -- declare exit handler for SQLSTATE '02000' close u_cursor;

    -- 声明条件处理程序：当SQL语句执行抛出的状态码为02开头时，将关闭游标u_cursor，并退出
    declare exit handler for not found close u_cursor;

    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;
end;
call p11(30);

```

存储函数

存储函数是有返回值的存储过程，存储函数的参数只能是IN类型的。具体语法如下

```

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。

案例:计算从1累加到n的值, n为传入的参数值。

```
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;
select fun1(50);
```

触发器

介绍

触发器是与表有关的数据库对象, 指在insert/update/delete之前(BEFORE)或之后(AFTER), 触发并执行触发器中定义的SQL语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完整性, 日志记录, 数据校验等操作

使用别名OLD和NEW来引用触发器中发生变化的记录内容, 这与其他数据库是相似的。现在触发器还支持行级触发, 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

语法

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。
```

案例

通过触发器记录 tb_user 表的数据变更日志, 将变更日志插入到日志表user_logs中, 包含增加, 修改, 删除;

表结构准备:

```
-- 准备工作 : 日志表 user_logs
create table user_logs
(
    id            int(11)      not null auto_increment,
    operation      varchar(20) not null comment '操作类型, insert/update/delete',
    operate_time   datetime    not null comment '操作时间',
    operate_id     int(11)      not null comment '操作的ID',
    operate_params varchar(500) comment '操作参数',
    primary key (`id`)
) engine = innodb
default charset = utf8;
```

插入数据触发器

```
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, ', phone=', NEW.phone, ', email=', NEW.email, ',
profession=', NEW.profession));
end;
```

锁

分类

1. 全局锁: 锁定数据库中的所有表
2. 表级锁: 每次操作锁住整张表
3. 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DML的语句，DDL语句，已经更新操作的事务提交语句都将被阻塞。

其典型的使用场景是做全库的逻辑备份，对所有的表进行锁定，从而获取一致性视图，保证数据的完整性。

语法：

加全局锁

```
flush tables with read lock ;
```

数据备份

```
mysqldump -uroot -p1234 itcast > itcast.sql
```

释放锁

```
unlock tables ;
```

缺点：

数据库中加全局锁，是一个比较重的操作，存在以下问题：

- 如果在主库上备份，那么在备份期间都不能执行更新，业务基本上就得停摆
- 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（binlog），会导致主从延迟。

在InnoDB中，我们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。

```
mysqldump --single-transaction -uroot -p123456 itcast > itcast.sql
```

表级锁

表级锁，每次操作锁住整张表。锁定粒度大，发生锁冲突的概率最高，并发度最低。应用在MyISAM，InnoDB，BOB等存储引擎中。

对于表级锁，主要分为以下三类：

1. 表锁
2. 元数据锁（meta data lock，MDL）
3. 意向锁

表锁

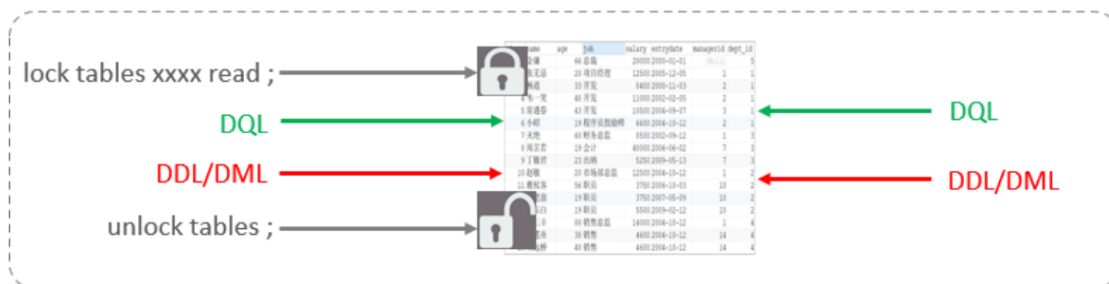
对于表锁，分为两类：

- 表共享读锁（read lock）
- 表独占写锁（write lock）

语法：

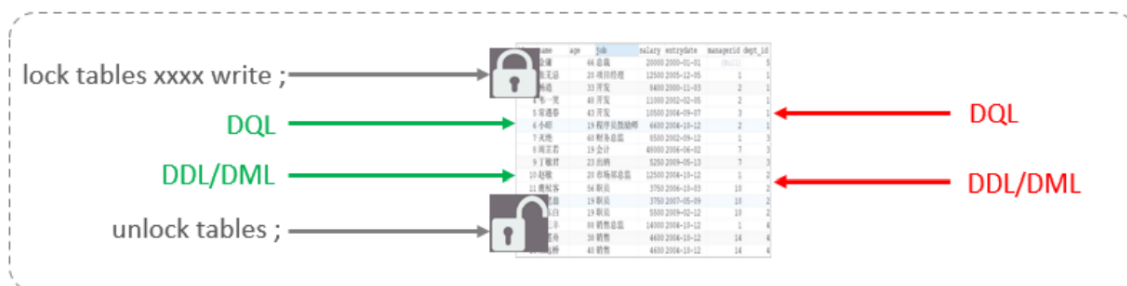
- 加锁：lock tables 表名... read/write。
- 释放锁：unlock tables / 客户端断开连接。

A. 读锁



左侧为客户端一，对指定表加了读锁，不会影响右侧客户端二的读，但是会阻塞右侧客户端的写。

B. 写锁



左侧为客户端一，对指定表加了写锁，会阻塞右侧客户端的读和写。

读锁不会阻塞其他客户端的读，但是会阻塞写。写锁既会阻塞其他客户端的读，又会阻塞其他客户端的写。

元数据锁 (meta data lock, MDL)

MDL加锁过程是系统自动控制，无需显式使用，在访问一张表的时候会自动加上。MDL锁主要作用是维护表元数据的数据一致性，在表上有活动事务的时候，不可以对元数据进行写入操作。**为了避免DML与DDL冲突，保证读写的正确性。**

这里的元数据，可以简单理解为就是一张表的表结构。也就是说，某一张表涉及到未提交的事务时，是不能够修改这张表的表结构的。

在MySQL5.5中引入了MDL，当对一张表进行增删改查的时候，加MDL读锁(共享)；当对表结构进行变更操作的时候，加MDL写锁(排他)。

常见的SQL操作时，所添加的元数据锁：

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

对应SQL	锁类型	说明
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

可以通过下面的SQL，来查看数据库中的元数据锁的情况：

```
select object_type,object_schema,object_name,lock_type,lock_duration from performance_schema.metadata_locks ;
```

意向锁

为了避免DML在执行时，加的行锁与表锁的冲突，在InnoDB中引入了意向锁，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。

分类：

- 意向共享锁(IS): 由语句select ... lock in share mode添加。与表锁共享锁 (read)兼容，与表锁排他锁(write)互斥。
- 意向排他锁(IX): 由insert、update、delete、select...for update添加。与表锁共享锁(read)及排他锁(write)都互斥，意向锁之间不会互斥。

一旦事务提交了，意向共享锁、意向排他锁，都会自动释放。

可以通过以下SQL，查看意向锁及行锁的加锁情况：

```
select object_schema,object_name,index_name,lock_type,lock_mode,lock_data from performance_schema.data_locks;
```

行级锁

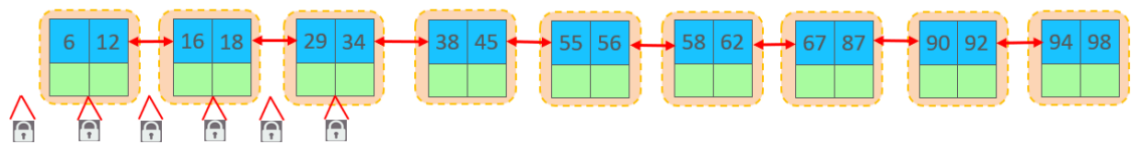
行级锁，每次操作锁住对应的行数据。锁定粒度最小，发生锁冲突的概率最低，并发度最高。应用在InnoDB存储引擎中。

InnoDB的数据是基于索引组织的，行锁是通过对索引上的索引项加锁来实现的，而不是对记录加的锁。对于行级锁，主要分为以下三类：

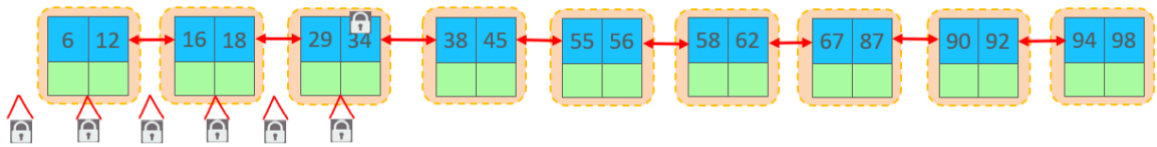
- 行锁（Record Lock）：锁定单个行记录的锁，防止其他事务对此行进行update和delete。在RC、RR隔离级别下都支持。



- 间隙锁（Gap Lock）：锁定索引记录间隙（不含该记录），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隔离级别下都支持。



- 临键锁（Next-Key Lock）：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隔离级别下支持。



InnoDB实现了以下两种类型的行锁：

- 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排它锁。
- 排他锁（X）：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	MODE SELECT ... FOR UPDATE 排他锁 需要手动在SELECT之后加FOR UPDATE

默认情况下，InnoDB在 REPEATABLE READ事务隔离级别运行，InnoDB使用 next-key 锁进行搜索和索引扫描，以防止幻读。

- 针对唯一索引进行检索时，对已存在的记录进行等值匹配时，将会自动优化为行锁。
- InnoDB的行锁是针对于索引加的锁，不通过索引条件检索数据，那么InnoDB将对表中的所有记录加锁，此时 就会升级为表锁。

间隙锁&临键锁

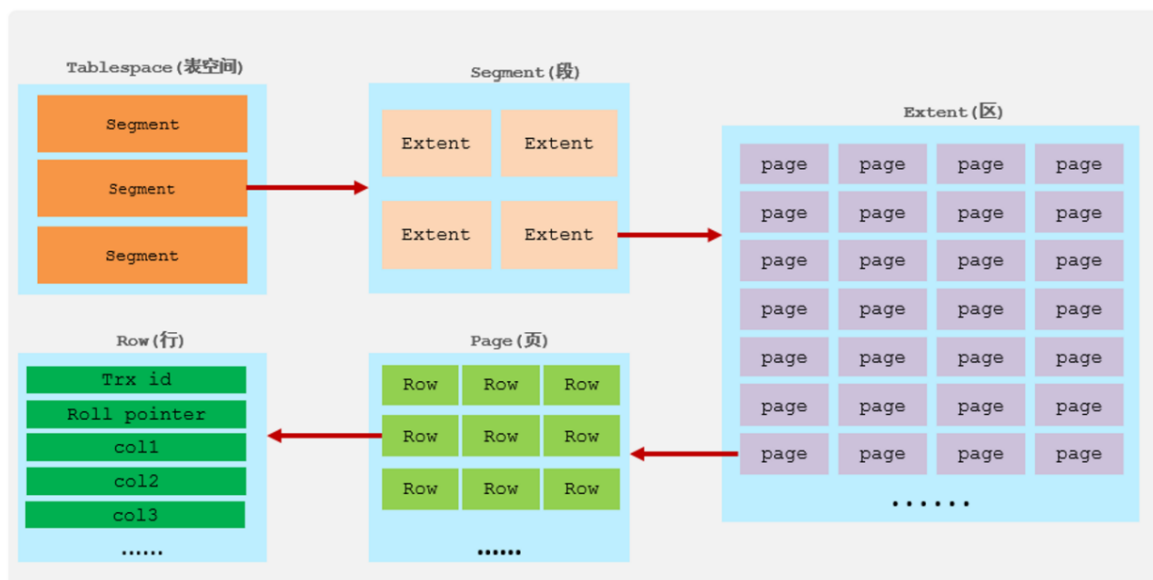
默认情况下，InnoDB在 REPEATABLE READ事务隔离级别运行，InnoDB使用 next-key 锁进行搜索和索引扫描，以防止幻读。

- 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(非唯一普通索引)，向右遍历时最后一个值不满足查询需求时，next-key lock 退化为间隙锁。
- 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。

注意：间隙锁唯一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阻止另一个事务在同一间隙上采用间隙锁。

InnoDB引擎

逻辑存储结构



- 表空间

表空间是InnoDB存储引擎逻辑结构的最高层，如果用户启用了参数 `innodb_file_per_table`(在 8.0 版本中默认开启)，则每张表都会有一个表空间 (`xxx.ibd`)，一个mysql实例可以对应多个表空间，用于存储记录、索引等数据。

- 段

段，分为数据段 (Leaf node segment)、索引段 (Non-leaf node segment)、回滚段 (Rollback segment)，InnoDB是索引组织表，数据段就是B+树的叶子节点，索引段即为B+树的非叶子节点。段用来管理多个Extent (区)。

- 区

区，表空间的单元结构，每个区的大小为1M。默认情况下，InnoDB存储引擎页大小为16K，即一个区中一共有64个连续的页。

- 页

页，是InnoDB 存储引擎磁盘管理的最小单元，每个页的大小默认为 16KB。为了保证页的连续性，InnoDB 存储引擎每次从磁盘申请 4-5 个区。

- 行

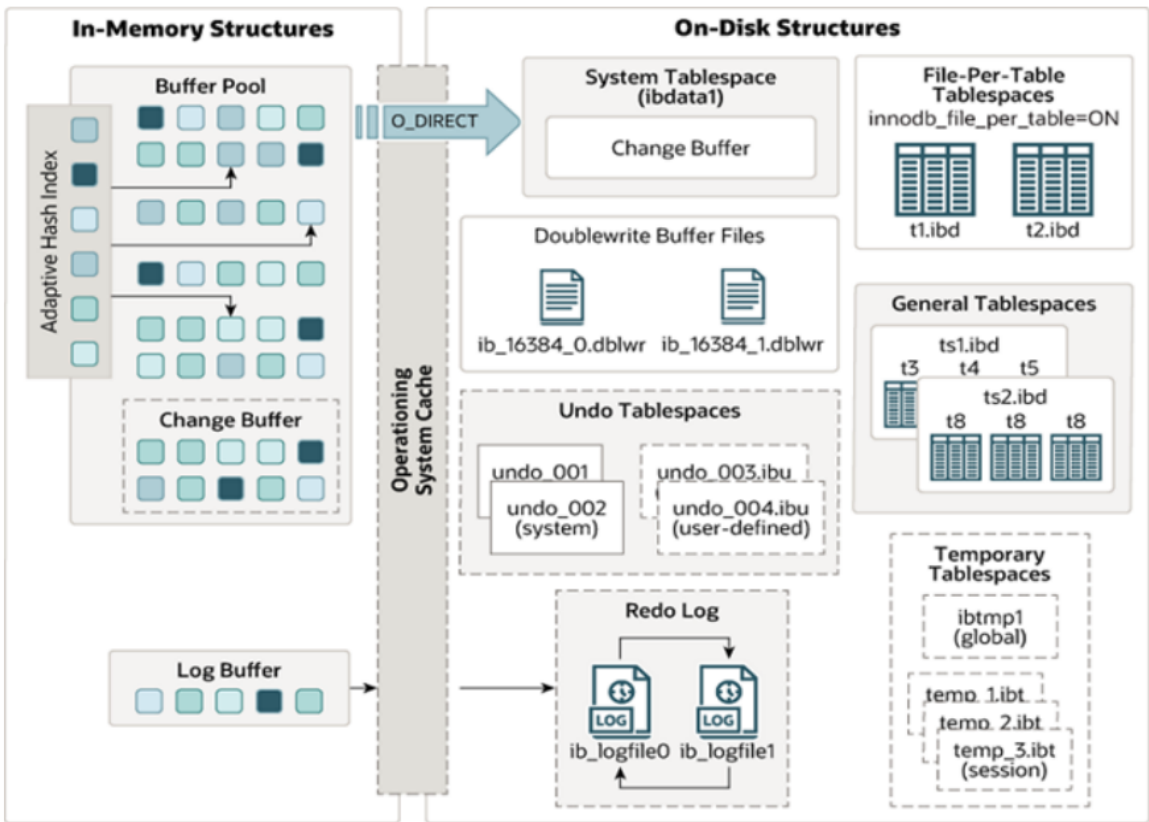
行，InnoDB 存储引擎数据是按行进行存放的。

在行中，默认有两个**隐藏字段**：

- Trx_id：每次对某条记录进行改动时，都会把对应的事务id赋值给trx_id隐藏列。
- Roll_pointer：每次对某条记录进行改动时，都会把旧的版本写入到undo日志中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

架构

MySQL5.5 版本开始，默认使用InnoDB存储引擎，它擅长事务处理，具有崩溃恢复特性，在日常开发中使用非常广泛。下面是InnoDB架构图，左侧为内存结构，右侧为磁盘结构



内存结构

在左侧的内存结构中，主要分为这么四大块儿：Buffer Pool、Change Buffer、Adaptive Hash Index、Log Buffer。接下来介绍一下这四个部分。

1. Buffer Pool：

InnoDB存储引擎基于磁盘文件存储，访问物理硬盘和在内存中进行访问，速度相差很大，为了尽可能弥补这两者之间的I/O效率的差值，就需要把经常使用的数据加载到缓冲池中，避免每次访问都进行磁盘I/O。

在InnoDB的缓冲池中不仅缓存了索引页和数据页，还包含了undo页、插入缓存、自适应哈希索引以及InnoDB的锁信息等等。

缓冲池 Buffer Pool，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。

在专用服务器上, 通常将多达80%的物理内存分配给缓冲池。参数设置: show variables like 'innodb_buffer_pool_size';

2. Change Buffer

Change Buffer, 更改缓冲区 (**针对于非唯一二级索引页**), 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的意义是什么呢?

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。

3. Adaptive Hash Index

自适应hash索引, 用于优化对Buffer Pool数据的查询。MySQL的InnoDB引擎中虽然没有直接支持hash索引, 但是给我们提供了一个功能就是这个自适应hash索引。因为前面我们讲到过, hash索引在 进行等值匹配时, 一般性能是要高于B+树的, 因为hash索引一般只需要一次IO即可, 而 B+树, 可能需 要几次匹配, 所以hash索引的效率要高, 但是hash索引又不适合做范围查询、模糊匹配等。

InnoDB存储引擎会监控对表上各索引页的查询, 如果观察到在特定的条件下hash索引可以提升速度, 则建立hash索引, 称之为自适应hash索引。

自适应哈希索引, 无需人工干预, 是系统根据情况自动完成。

参数: adaptive_hash_index

4. Log Buffer

Log Buffer: 日志缓冲区, 用来保存要写入到磁盘中的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、插入或删除许多行的事务, 增加日志缓冲区的大小可以节省磁盘 I/O。

参数:

innodb_log_buffer_size: 缓冲区大小

innodb_flush_log_at_trx_commit: 日志刷新到磁盘时机, 取值主要包含以下三个:

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

磁盘结构

1. System Tablespace

系统表空间是更改缓冲区的存储区域。如果表是在系统表空间而不是每个表文件或通用表空间中创建的, 它也可能包含表和索引数据。(在MySQL5.x版本中还包含InnoDB数据字典、undolog等)

参数: Innodb_data_file_path

系统表空间, 默认的文件名叫 ibdata1。

2. File-Per-Table Tablespaces

如果开启了innodb_file_per_table开关，则每个表的文件表空间包含单个InnoDB表的数据和索引，并存储在文件系统上的单个数据文件中。

开关参数：innodb_file_per_table，该参数默认开启。

3. General Tablespaces

通用表空间，需要通过 CREATE TABLESPACE 语法创建通用表空间，在创建表时，可以指定该表空间。

创建表空间：

```
CREATE TABLESPACE ts_name ADD DATAFILE 'file_name' ENGINE = engine_name;
```

创建表时指定表空间：

```
CREATE TABLE xxx ... TABLESPACE ts_name;
```

4. Undo Tablespaces

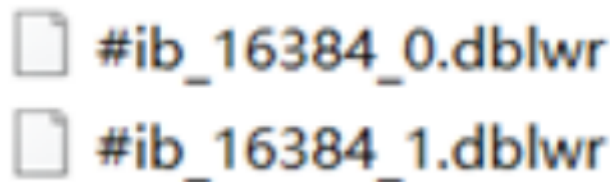
撤销表空间，MySQL实例在初始化时会自动创建两个默认的undo表空间（初始大小16M），用于存储 undo log日志。

5. Temporary Tablespaces

InnoDB 使用会话临时表空间和全局临时表空间。存储用户创建的临时表等数据。

6. Doublewrite Buffer Files

双写缓冲区，InnoDB引擎将数据页从Buffer Pool刷新到磁盘前，先将数据页写入双写缓冲区文件中，便于系统异常时恢复数据。

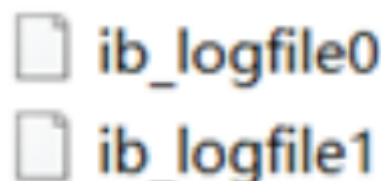


#ib_16384_0.dblwr
#ib_16384_1.dblwr

7. Redo Log

重做日志，是用来实现事务的持久性。该日志文件由两部分组成：重做日志缓冲（redo log buffer）以及重做日志文件（redo log），前者是在内存中，后者在磁盘中。当事务提交之后会把所有修改信息都会存到该日志中，用于在刷新脏页到磁盘时，发生错误时，进行数据恢复使用。

以循环方式写入重做日志文件，涉及两个文件：



ib_logfile0
ib_logfile1

后台线程

1. 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。

2. IO Thread

在InnoDB存储引擎中大量使用了AIO来处理IO请求, 这样可以极大地提高数据库的性能, 而IO Thread主要负责这些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

我们可以通过以下的这条指令，查看到InnoDB的状态信息，其中就包含IO Thread信息

```
show engine innodb status \G;
```

3. Purge Thread

主要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，就用它来回收。

4. Page Cleaner Thread

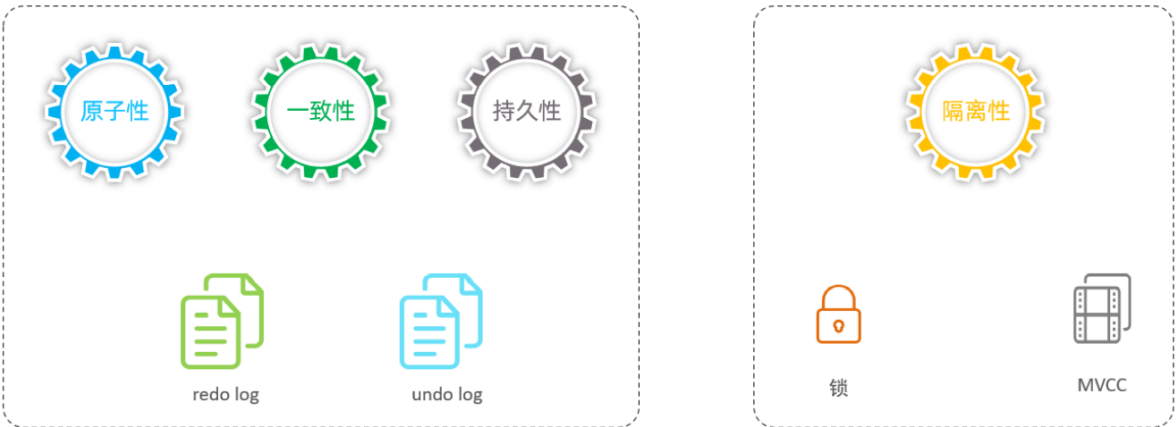
协助 Master Thread 刷新脏页到磁盘的线程，它可以减轻 Master Thread 的工作压力，减少阻塞。

事务原理

我们研究事务的原理，就是研究MySQL的InnoDB引擎是如何保证事务的这四大特性的。

而对于这四大特性，实际上分为两个部分：

- **原子性、一致性、持久化**，实际上是由InnoDB中的**两份日志**来保证的，一份是**redo log日志**，一份是**undo log日志**。
- **持久性**是通过数据库的**锁**，加上**MVCC**来保证的。



redo log

重做日志，记录的是事务提交时数据页的物理修改，是用来实现事务的持久性。

该日志文件由两部分组成：**重做日志缓冲 (redo log buffer)** 以及**重做日志文件 (redo log file)**，前者是在内存中，后者在磁盘中。当事务提交之后会把所有修改信息都存到该日志文件中，用于在刷新脏页到磁盘，发生错误时，进行数据恢复使用。

有了redolog之后，当对缓冲区的数据进行增删改之后，会首先将操作的数据页的变化，记录在redo log buffer中。在事务提交时，会将redo log buffer中的数据刷新到redo log磁盘文件中。过一段时间之后，如果刷新缓冲区的脏页到磁盘时，发生错误，此时就可以借助于redo log进行数据恢复，这样就保证了事务的持久性。而如果脏页成功刷新到磁盘或或者涉及到的数据已经落盘，此时redolog就没有作用了，就可以删除了，所以存在的两个redolog文件是循环写的。

那为什么每一次提交事务，要刷新redo log 到磁盘中呢，而不是直接将buffer pool中的脏页刷新到磁盘呢？

因为在业务操作中，我们操作数据一般都是随机读写磁盘的，而不是顺序读写磁盘。而redo log在往磁盘文件中写入数据，由于是日志文件，所以都是顺序写的。顺序写的效率，要远大于随机写。这种先写日志的方式，称之为WAL (Write-Ahead Logging)。

undo log

回滚日志，用于记录数据被修改前的信息，作用包含两个：提供回滚(保证事务的原子性)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中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。当执行rollback时，就可以从undo log中的逻辑记录读取到相应内容并进行回滚。

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，因为这些日志可能还用于MVCC。

Undo log存储：undo log采用段的方式进行管理和记录，存放在前面介绍的rollback segment回滚段中，内部包含1024个undo log segment。

MVCC

MVCC基本概念

- 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。对于我们日常的操作，如：select ... lock in share mode(共享锁)，select ... for update、update、insert、delete(排他锁)都是一种当前读。

- 快照读

简单的select（不加锁）就是快照读，快照读，读取的是记录数据的可见版本，有可能是历史数据，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语句才是快照读的地方。
- Serializable：快照读会退化为当前读。

- MVCC

全称 Multi-Version Concurrency Control，多版本并发控制。指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的具体实现，还需要依赖于数据库记录中的三个隐式字段、undo log日志、readView。

MVCC实现原理

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。

而上述的前两个字段是肯定会添加的，是否添加最后一个字段DB_ROW_ID，得看当前表有没有主键，如果有主键，则不会添加该隐藏字段。

undo log

回滚日志，在insert、update、delete的时候产生的便于数据回滚的日志。

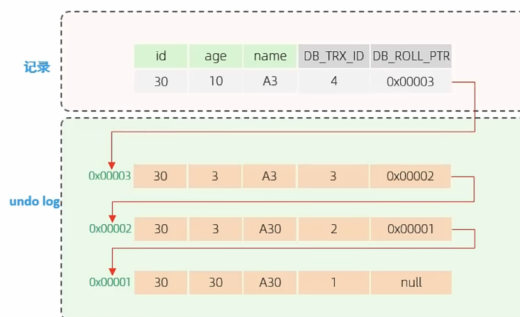
当insert的时候，产生的undo log日志只在回滚时需要，在事务提交后，可被立即删除。

而update、delete的时候，产生的undo log日志不仅在回滚时需要，在快照读时也需要，不会立即被删除。

- 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		
	提交事务		查询id为30的记录
		修改id为30记录，age改为10	
		查询id为30的记录	
		提交事务	查询id为30的记录



不同事务或相同事务对同一条记录进行修改，会导致该记录的undolog生成一条记录版本链表，链表的头部是最新的旧记录，链表尾部是最早的旧记录。

readview

ReadView（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。

ReadView中包含了四个核心字段：

m_ids	当前活跃的事务ID集合
min_trx_id	最小活跃事务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

而在readview中就规定了版本链数据的访问规则：trx_id 代表当前undolog版本链对应事务ID。

● readview

trx_id: 代表是当前事务ID。

版本链数据访问规则

①. $trx_id == creator_trx_id$? 可以访问该版本

成立，说明数据是当前这个事务更改的。

②. $trx_id < min_trx_id$? 可以访问该版本

成立，说明数据已经提交了。

③. $trx_id > max_trx_id$? 不可以访问该版本

成立，说明该事务是在ReadView生成后才开启。

④. $min_trx_id \leq trx_id \leq max_trx_id$? 如果trx_id不在m_ids中是可以访问该版本的

成立，说明数据已经提交。

条件	是否可以访问	说明
$trx_id == creator_trx_id$	可以访问该版本	成立，说明数据是当前这个事务更改的。
$trx_id < min_trx_id$	可以访问该版本	成立，说明数据已经提交了。
$trx_id > max_trx_id$	不可以访问该版本	成立，说明该事务是在 ReadView 生成后才开启。
$min_trx_id \leq trx_id \leq max_trx_id$	如果trx_id不在m_ids中，是可以访问该版本的	成立，说明数据已经提交。

不同的隔离级别，生成ReadView的时机不同：

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

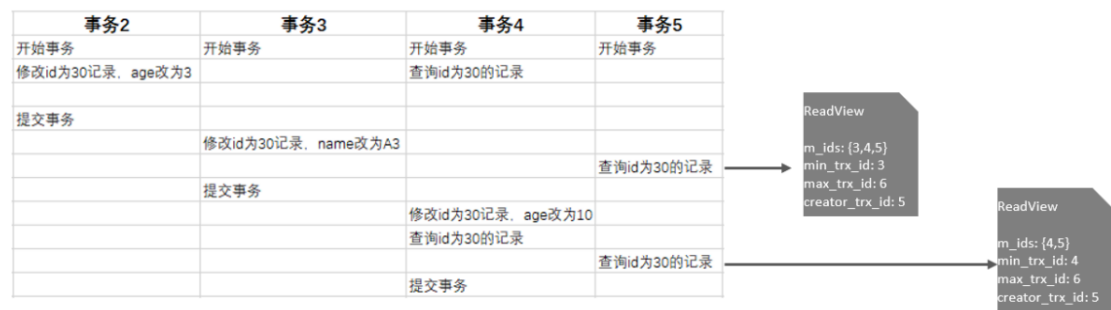
REPEATABLE READ：仅在事务中第一次执行快照读时生成ReadView，后续复用该ReadView。

原理分析

RC隔离级别

RC隔离级别下，在事务中每一次执行快照读时生成ReadView。

在事务5中，查询了两次id为30的记录，由于隔离级别为Read Committed，所以每一次进行快照读都会生成一个ReadView，那么两次生成的ReadView如下。



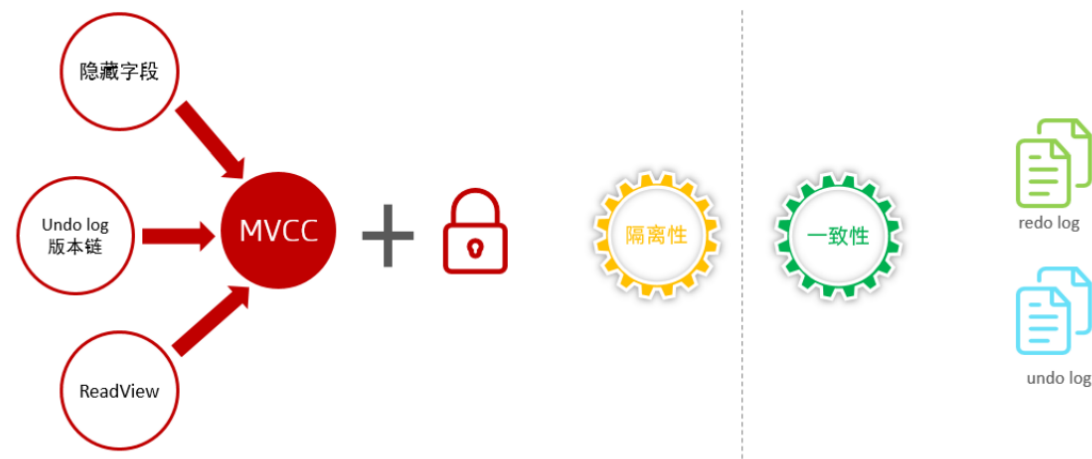
那么这两次快照读在获取数据时，就需要根据所生成的ReadView以及ReadView的版本链访问规则，到undolog版本链中匹配数据，最终决定此次快照读返回的数据。

RR隔离级别

RR隔离级别下，仅在事务中第一次执行快照读时生成ReadView，后续复用该ReadView。而RR 是可重复读，在一个事务中，执行两次相同的select语句，查询到的结果是一样的。

MVCC总结

MVCC的实现原理就是通过 InnoDB表的隐藏字段、UndoLog 版本链、ReadView来实现的。而MVCC + 锁，则实现了事务的隔离性。而一致性则是由redolog 与 undolog保证。



MySQL管理

系统数据库

Mysql数据库安装完成后，自带了一下四个数据库，具体作用如下：

数据库	含义
mysql	存储MySQL服务器正常运行所需要的各种信息（时区、主从、用户、权限等）
information_schema	提供了访问数据库元数据的各种表和视图，包含数据库、表、字段类型及访问权限等
performance_schema	为MySQL服务器运行时状态提供了一个底层监控功能，主要用于收集数据库服务器性能参数
sys	包含了一系列方便 DBA 和开发人员利用 performance_schema 性能数据库进行性能调优和诊断的视图

常用工具

mysql

该mysql不是指mysql服务，而是指mysql的客户端工具。

```
语法：  
mysql [options] [database]  
选项：  
-u, --user=name #指定用户名  
-p, --password[=name] #指定密码  
-h, --host=name #指定服务器IP或域名  
-P, --port=port #指定连接端口  
-e, --execute=name #执行SQL语句并退出
```

-e选项可以在Mysql客户端执行SQL语句，而不用连接到MySQL数据库再执行，对于一些批处理脚本，这种方式尤其方便。

示例：

```
mysql -uroot -p123456 db01 -e "select * from stu"
```

mysqladmin

mysqladmin 是一个执行管理操作的客户端程序。可以用它来检查服务器的配置和当前状态、创建并删除数据库等。

通过帮助文档查看选项：

```
mysqladmin --help
```

语法：

```
mysqladmin [options] command ...
```

选项：

```
-u, --user=name #指定用户名  
-p, --password[=name] #指定密码  
-h, --host=name #指定服务器IP或域名  
-P, --port=port #指定连接端口
```

示例：

```
mysqladmin -uroot -p1234 drop 'test01';  
mysqladmin -uroot -p1234 version;
```

mysqlbinlog

由于服务器生成的二进制日志文件以二进制格式保存，所以如果想要检查这些文本的文本格式，就会使用到mysqlbinlog 日志管理工具。

语法：

```
mysqlbinlog [options] log-files1 log-files2 ...
```

选项：

```
-d, --database=name 指定数据库名称，只列出指定的数据库相关操作。  
-o, --offset=# 忽略掉日志中的前n行命令。  
-r, --result-file=name 将输出的文本格式日志输出到指定文件。  
-s, --short-form 显示简单格式，省略掉一些信息。  
--start-datetime=date1 --stop-datetime=date2 指定日期间隔内的所有日志。  
--start-position=pos1 --stop-position=pos2 指定位置间隔内的所有日志。
```

mysqlshow

mysqlshow 客户端对象查找工具，用来很快地查找存在哪些数据库、数据库中的表、表中的列或者索引。

语法：

```
mysqlshow [options] [db_name [table_name [col_name]]]
```

选项：

```
--count 显示数据库及表的统计信息（数据库，表 均可以不指定）  
-i 显示指定数据库或者指定表的状态信息
```

示例：

```
#查询test库中每个表中的字段书，及行数
```

```
mysqlshow -uroot -p2143 test --count
```

```
#查询test库中book表的详细情况
```

```
mysqlshow -uroot -p2143 test book --count
```

mysqldump

mysqldump 客户端工具用来备份数据库或在不同数据库之间进行数据迁移。备份内容包含创建表，及插入表的SQL语句。

语法：

```
mysqldump [options] db_name [tables]
mysqldump [options] --database/-B db1 [db2 db3...]
mysqldump [options] --all-databases/-A
```

连接选项：

```
-u, --user=name 指定用户名
-p, --password[=name] 指定密码
-h, --host=name 指定服务器ip或域名
-P, --port=# 指定连接端口
```

输出选项：

```
--add-drop-database 在每个数据库创建语句前加上 drop database 语句
--add-drop-table 在每个表创建语句前加上 drop table 语句，默认开启；不
开启 (--skip-add-drop-table)
-n, --no-create-db 不包含数据库的创建语句
-t, --no-create-info 不包含数据表的创建语句
-d --no-data 不包含数据
-T, --tab=name 自动生成两个文件：一个.sql文件，创建表结构的语句；一个.txt文件，数据文件
```

mysqlimport/source

- mysqlimport

mysqlimport 是客户端数据导入工具，用来导入mysqldump 加 -T 参数后导出的文本文件。

语法：

```
mysqlimport [options] db_name textfile1 [textfile2...]
```

示例：

```
mysqlimport -uroot -p2143 test /tmp/city.txt
```

- source

如果需要导入sql文件,可以使用mysql中的source 指令：

语法：

```
source /root/xxxxx.sql
```