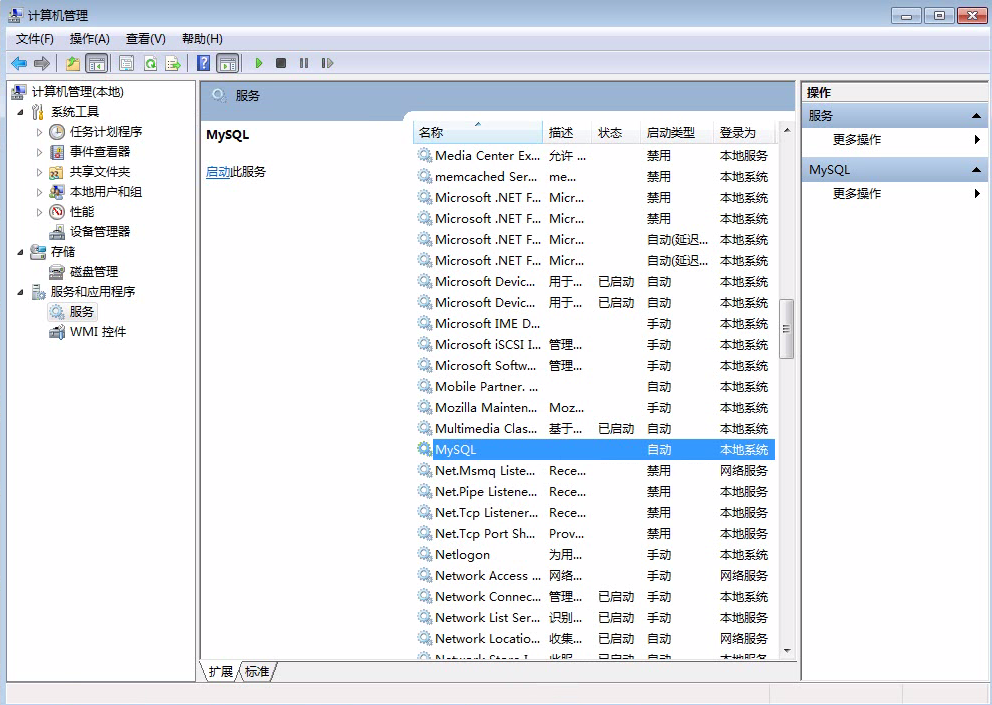
# 一、基本语法

## 1.MySQL的基本使用

### 1.1MySQL服务的启动

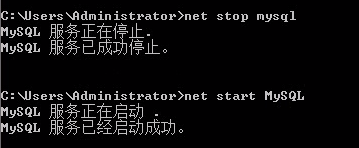
**方案1：**

通过计算机的服务管理界面可开启或MySQL服务



**方案2：**

通过命令行进行服务开启和关闭



### 1.2登陆到MySQL

打开cmd命令行，输入mysql –u root –p，回车即可，或者在wamp中打开mysql控制台

注意：这里的mysql –u root –p，完整的命令行语法应当是：

**mysql -h 主机名 –u 用户名 –p 密码**

## 2.MySQL数据库的创建

### 2.1数据库操作

**1.创建数据库语法（不限制大小写,但推荐大写）**

**CREATE DATABASE [IF NOT EXISTS] 数据库名字 [character set 字符集名][collate 校验规则]**

校验规则是数据库中数据默认的排序规则，一般使用utf8\_general\_ci规则,该属性会影响到order by查询，如果使用了不区分大小写的校验规则，那么排序就会忽略大小写

eg：

CREATE DATABASE db1;

CREATE DATABASE db2 character set utf8;

CREATE DATABASE db3 character set utf8 collate utf8\_general\_ci;

**2.如何查看当前数据库列表：**

**SHOW DATABASE;**

注意database后有s

**3.显示数据库创建语句：**

**SHOW CREATE DATABASE 数据库名;**

**4.删除数据库**

**DROP DATABASE [IF EXISTS] 数据库名;**

**5.如何查看当前数据库的表列表：**

**SHOW TABLES;**

**6.如何查看当前数据库的连接情况：**

**SHOW PROCESSLIST;**

**7.查看服务器对外处理的字符集**

**show variables like "character\_set%";**

### 2.2创建表

#### 2.2.1创建表

在创建表之前，应当先使用数据库

**语法:**

**use 数据库名;**

创建表

**语法：**

**CREATE TABLE 表名(**

**字段名1 数据类型[ unsigned],**

**字段名2 数据类型[ unsigned],**

**…**

**)[character set 字符集][collate 校对规则][engine 存储引擎方式]**

**注意： 1.如果表和数据库同时指定了字符集或校对规则，以表为主**

**2.常用的存储引擎有MYISAM、InnoDB、Memory,默认的存储引擎在my.ini文件中，可以修改**

**3.创建表时，字段不能使用mysql关键字(保留字)，如：name字段**

eg：

CREATE DATABASE db2 character set utf8;

use db2;

CREATE TABLE Student(

name varchar(10),

age int unsigned

)

**修改引擎**

alter table account engine=引擎名;

alter table oldTableName rename newTableName;

#### 2.2.2其他约束

注意;mysql没有check约束

##### 1.添加默认值

**语法：**

**not null default ‘’,**

**eg:**

CREATE TABLE Student(

name varchar(10) not null default ‘无名氏’,

age int not null

)

注意：因为空数据无法参与运算，应尽量避免出现null

##### 2.主键约束primary key

eg：

CREATE TABLE Student(

Snum smallint primary key comment ‘学号’,

name varchar(10),

age int not null

)

复合主键是指由两个及以上的字段组成的主键，如:

CREATE TABLE score(

Snum smallint comment '学号',

Cnum smallint comment '课程号',

name varchar(10),

score tinyint comment '成绩',

primary key(Snum,Cnum)

);

还可以通过修改的方式进行(前提是表未指定主键)

alter table score add primary key(Snum,Cnum);

删除主键

alter table score drop primary key;

##### 3.自动增长

启用自动增长后，当对应的字段不给值，或者给默认值或者为null，系统会从当前字段的已有最大值进行+1再填入该行记录

语法:

**字段名 字段类型 auto\_increment[=n]**

**注意： 1.自增长字段必须是整型，且一张表只能由一个自增长字段**

**2.可选的=n是指定自增长的初始值，可用alter table table\_name AUTO\_INCREMENT=n来改变**

**3.通过alter table table\_name modify 字段 类型;来增加和删除自增长**

eg:

create table class(

classNum int primary key auto\_increment,

stuNum tinyint not null

) auto\_increment=2;

insert into class(stuNum) value('50'),('43'),('56');

| 2 | 50 |

| 3 | 43 |

| 4 | 56 |

##### 4.注释comment

mysql创建表时的注释

eg： CREATE TABLE Student(

name varchar(10) not null comment ‘学生姓名’,

age int not null

)

##### 5.唯一键unique

很多时候，一张表中的某些字段不能重复，这时需要加上unique

eg：

name varchar(10) unique,

或unique key(name)

追加唯一键:

alter table 表名 add unique key(name);

删除唯一键（通过删除索引即可）:

alter table 表名 drop index 字段名

##### 6.外键foreign key

myisam引擎不支持外键

**语法：**

**foreign key(外键字段名)[constraint 外键约束名] references 外部表名(主键名)**

可以通过constraint 外键约束名指定该外键约束的名字

增加外键：

alter table 表名 add foreign key(外键字段名) references(主键名)

删除外键

alter table 表名drop foreign key 外键约束名

可通过

show create table 表名来查看外键约束名

#### 2.2.3索引

索引是建立在字段上的，他有两个意义：

1. 以牺牲一定存储空间为代价提升数据查询效率
2. 约束数据(唯一键等是通过索引进行的)

mysql提供了许多索引：

1. 主键索引
2. 唯一索引
3. 全文索引(针对文章内部的关键字进行索引，难度在于确定关键字)
4. 普通索引

### 2.3MySQL的数据类型

**1.数值类型**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 范围（有符号） | 用途 |
| tinyint | 1 字节 | (-128，127) | 存储小整数值 |
| smallint | 2 字节 | (-32 768，32 767) | 存储大整数值 |
| mediumint | 3 字节 | (-8 388 608，8 388 607) | 存储大整数值 |
| int | 4 字节 | (-2 147 483 648，2 147 483 647) | 存储大整数值 |
| bigint | 8 字节 | (-9 233 372 036 854 775 808，9 223 372 036 854 775 807) | 存储超大整数值 |
| float | 4 字节 | (-3.402 823 466 E+38，-1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38) | 存储单精度浮点数值 |
| double | 8 字节 | (-1.797 693 134 862 315 7 E+308，-2.225 073 858 507 201 4 E-308)，0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308) | 存储双精度浮点数值 |
| decimal | 视具体情况 |  | 存放浮点数(精度高) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

eg：

CREATE TABLE Student(

name varchar(10),

age int(3),

tuition float(4,2)//能存放的范围是-99.99~99.99，4是数字的位数，2是小数的位数

tuition decimal (4,2)// decimal的精度相较float更高

)

**注意： 1.显示宽度是在数据类型后面加上括号，如int(4)代表显示4位宽度，-122是四位，233是三位，显示宽度通常是为了补足位数，如宽度为3位，但值是2，即002，但要注意，zerofill会使字段变为无符号，eg：**

alter table student add newint tinyint(3) zerofill;

insert into student value('ad',1,1);

| ad | 1 | 001 |

**2.字符串类型**

|  |  |  |
| --- | --- | --- |
| 数据类型 | 大小 | 用途 |
| char | |  | | --- | |  |   0~255字符 | 存储定长字符串 |
| varchar | 0-65535字节 | 存储变长字符串 |
| text | 0-65535字节 | 存储长文本数据 |
|  |  |  |

**注意： 1.varchar最大是65535字节，因此，在定义varchar时：**

**字符类型若为gbk，每个字符最多占2个字节，定义的最大长度不能超过32766;**

**字符类型若为utf8，每个字符最多占3个字节，定义的最大长度不能超过21845。**

**varchar(n) 表示n个字符，无论汉字和英文，MySql都能存入 n 个字符(n<=32766或21845)**

**char(n) 也表示n个字符**

**2.char的最大长度是255，不限定字节**

**3. varchar占用空间小，但速度慢，char占用空间大但速度快**

**3.日期型**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 格式 | 用途 |
| date | 3字节 | YYYY-MM-DD | 存储日期值 |
| time | 3字节 | HH:MM:SS | 存储时间值 |
| datetime | 8字节 | YYYY-MM-DD HH:MM:SS | 混合日期和时间值 |
| year | 1字节 | YYYY | 存储年份值 |
| timestamp | 4字节 | YYYYMMDD HHMMSS | 混合日期和时间值，时间戳 |

**注意:timestamp时间戳会随着记录的更新自动更新为当前时间，而且该类型声明的字段有默认值且非空，php主要是以timestamp存储时间**

**eg：**

alter table students modify updated\_at TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP;//当修改现有记录的时候将updated\_at刷新为CURRENT\_TIMESTAMP

**4.枚举型enum和集合set**

**(1)枚举enum(单选)**

枚举型要求存储的数据必须是规定的数据中的一个(mysql没有check约束，通过enum和set可以达到相同效果)

**语法：**

**enum(可能出现的元素列表)**

eg：create table student(

snum smallint primary key,

name varchar(5),

sex enum('男','女') default '男',

class int unique

)character set utf8;

**注意：由于enum存储的是索引数值，因此它还可以节省空间，如记录(‘001’,’张政’,’男’,14)存储的就是(‘001’,’张政’,’1’,14)，因此也可以给枚举enum类型直接插入数值**

**(2)集合set(多选)**

集合和枚举类似，实际存储的是数值而不是字符串，但在存取数值时是存取多个元素

**语法：**

**set (可能出现的元素列表)**

**eg：**

alter table student add hobby set('足球','羽毛球','乒乓球','篮球');

| snum | smallint(6)

| name | varchar(5)

| sex | enum('男','女')

| class | int(11)

| hobby | set('足球','羽毛球','乒乓球','篮球')

insert into student value('001','张政','1','14','足球,羽毛球');

| 1 | 张政 | 男 | 14 | 足球,羽毛球 |

### 2.4插入语句insert

**语法：**

**INSERT INTO 表名(字段1, 字段2,…字段n) VALUES(值1, 值2,…值n) [, (值1, 值2,…值n)]**

注意： **1.控制台默认是GBK编码来管理输入的，所以插入的值也是GBK，但如果表示以UTF8格式存储，则会报错，解决方案是在表中输入指令set names gbk;将该字段的数据从GBK转成utf-8发送给数据库**

**2.字符和日期类型的数据应当包含在单引号中**

**3.当插入数据存在主键冲突时(插入数据的主键重复)，可以覆盖旧的记录，语法如下：**

**INSERT INTO 表名(主键, 字段1,…字段n) VALUES(值1,…值n) on duplicate key update 字段1=新值1, 字段2=新值2…**

eg:

insert into score values('001','001','张三','60');

insert into score values('001','001','李四','70') on duplicate key update name='李四',score='70';

INSERT INTO runoob\_tbl(runoob\_title, runoob\_author, submission\_date) VALUES ("学习 PHP", "菜鸟教程", NOW());

**4.蠕虫复制：通过查询进行插入，即将查询出的数据插入表中，语法如下：**

**NSERT INTO 表名(字段1,…字段n) select (字段1,…字段n)/\* from 数据表名…**

**蠕虫复制可以快速拷贝已有数据，还可以测试表的压力及效率**

### 2.5更新语句update

**语法：**

**UPDATE 表名 SET 字段名1=新值1[,字段名2=新值2…] where 条件 [limit 数量]**

eg:

update student set name='张政',age=20 where name='丁健';

update student set jiangli='已发奖' where score>90 limit 3;

### 2.5删除语句delete

**语法：**

**DELETE FROM 表名 where 条件 [limit 数量]**

**注意： 1.如果不添加where条件，则会删除表中所有数据，删除数据一定要慎重！**

**2.删除行还可以使用truncate语句，语法同上 TRUNCATE FROM 表名where 条件,二者的区别是：**

**（1）truncate速度快，但语句没有返回值，也不能带where条件，（2）delete速度慢，但有返回值，能带where条件**

**因此，一般当需要把表清空，才需要使用truncate**

eg:

delete from student where score<60 limit 3;

### 2.6查询语句select语句

**语法：**

**SELECT [DISTINCT] 字段名1,字段名2… from 表名 where 条件**

**[group by 列名 [HAVING 条件]]**

**[order by 列名 [ASC|DESC]]**

**[limit 数量]**

#### 2.6.1单表查询

insert into student values('丁健',18,'男','我叫丁健');

insert into student values('张政',18,'女','我叫张政');

insert into student values('张灿',28,'男','我叫张灿');

insert into student values('丁健',28,'男','我叫丁健');

##### 1.字段名可以是表达式

select语句中字段名还可以是表达式，如:

select name,age+100 from student;

##### 2.别名as

select语句中字段名还可以被替换，此处as可省略，如:

select name as NAME,jiehao,age+100 from student;

**注意：别名不能作为where条件的字段使用！**

##### 3.消除重复记录destinct

distinct是将重复的字段名去掉，是对查询结果进行过滤

eg：

select distinct name,jieshao from student

结果为：

| 丁健 | 我叫丁健 |

| 张政 | 我叫张政 |

| 张灿 | 我叫张灿 |

##### 4.where条件

比较条件 = , > , < , >= , <= , != , <>[不等于]

范围 between and[between后面一定要加较小值，and后加较大值] , not between and

集合 in , not in

模糊查询 like , not like

空值 is null , is not null

多重条件 AND , OR , NOT

**注意： 模糊查询的语法为 ：**

**[not ] like ‘匹配串’ [escape ‘自定义的转义字符’]**

**匹配串中，%代表任意长度(可为0)的字符串，\_代表一个任意的字符，如：a%b代表以a开头，以b结尾的字符串；a\_b代表a开头，b结尾的长度为3的字符串，escape ‘\’表示’\’是转义字符，’\\_’就意味着普通的’\_’符号**

eg:

select name as NAME,jiehao,age+100 from student where age+100 not between 119 and 230;

select name as NAME,jiehao,age+100 from student where sex in('女');

select \* from student where name LIKE '丁\_';

select \* from student where name LIKE '丁/\_\_' escape '/';(建议不要使用\)

select \* from student where name is not null;

##### 5.聚集函数

|  |  |
| --- | --- |
| COUNT (\*) | 统计记录个数 |
| AVG([distinct] 列名) | 返回某列的平均值 |
| COUNT([distinct] 列名) | 返回某列的行数 |
| MAX([distinct] 列名) | 返回某列的最大值 |
| MIN([distinct] 列名) | 返回某列的最小值 |
| SUM([distinct] 列名) | 返回某个列之和 |

若制定了distinct，则计算时忽略重复行

##### 6.分组group by

**group by 列名 [HAVING 条件]**

按照列名分组，值相等的为一组，分组后，聚集函数将作用于每一组，每一组都有一个函数值

eg：

select name,count(\*) from student group by name;

|丁\_健 | 1 |

| 丁健 | 2 |

| 张政 | 1 |

| 张灿 | 1 |

若分组后还要进行筛选，最终选出符合条件的组，即对组进行筛选，使用having

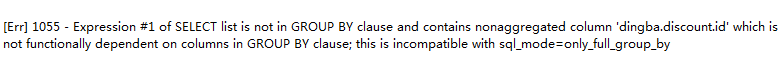
eg：

select name,count(\*) from student group by name having name in ('丁健','丁\_健');

|丁\_健 | 1 |

| 丁健 | 2 |

需要注意的是，如果出现以下错误：



即在only\_full\_group\_by模式下，先执行分组操作，然后再从分组的内容中取出select的字段，也就是说select的字段应当在分组中全部包含(**聚合函数中用到的字段可以不包括在group by中**)，eg：

SELECT

discountName,

title,

facilitator\_name,

count(title\_id) AS orderNum

FROM

discount

GROUP BY

title, facilitator\_name

报错，因为discountName不在分组的字段中

**方法一：**

只需将discountName加入到group by的字段中即可

**方法二(推荐)：**

方法一有许多局限性，最合适的方法就是利用聚合函数中用到的字段可以不包括在group by中这一特性，使用聚合函数处理一下查询的字段，eg：

SELECT

max(discountName) as discountName,

title,

max(facilitator\_name) as facilitator,

count(title\_id) AS orderNum

FROM

discount

GROUP BY

title

**方法三(不推荐)：**

或者关闭only\_full\_group\_by模式：

select @@global.sql\_mode 查询有哪些模式，然后去掉only\_full\_group\_by

如查询出查询出来的值为：

**ONLY\_FULL\_GROUP\_BY**,STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_AUTO\_CREATE\_USER,NO\_ENGINE\_SUBSTITUTION

那么只要去掉该模式即可：

set @@global.sql\_mode   
=`STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_AUTO\_CREATE\_USER,NO\_ENGINE\_SUBSTITUTION`;

参考：<http://blog.csdn.net/u010429286/article/details/64444271>

##### 7.排序order by

order by子句可以将结果属性的**升序(ASC)**[升序空值最大,放最后显示]或**降序(DESC)**排列，默认为升序

**order by 列名 [ASC|DESC]**

eg：select \* from student order by age desc;

**注意：order by在联合查询中需要加括号，同时必须搭配limit**

eg：

(select \* from student where class='1' order by snum desc limit 99999) union all (select \* from student where sex='男' order by snum desc limit 99999);

#### 2.6.2多表连接查询

##### 1.多表连接(内连接)

多表连接时，连接结果中有重复的列名需要加上对应的表名用以区分

eg：

select student.name,age,sex,score from student,score where student.name=score.name order by score desc;

或：

select student.name,age,sex,score from student inner join score on student.name=score.name order by name desc;

##### 2.自身连接

某个表也可以和自身进行连接，此时要给自身取两个别名

select distinct first.name,second.name as NAME,first.sex,second.age from student first,student second where first.name=second.name;

##### 3.外连接join

+student表+

| 丁健 | 18 | 男 | 我叫丁健 |

| 张政 | 18 | 女 | 我叫张政 |

| 张灿 | 28 | 男 | 我叫张灿 |

| 丁健 | 28 | 男 | 我叫丁健 |

| 丁\_健 | 18 | 男 | 我叫丁健 |

| 张彪 | 10 | 男 | NULL |

+score表+

| 丁健 | 100 |

| 丁\_健 | 90 |

| 张灿 | 80 |

| 张政 | 60 |

正常的连接中：

select student.name,sex,score from student,score where student.name=score.name;

| 丁健 | 男 | 100 |

| 张政 | 女 | 60 |

| 张灿 | 男 | 80 |

| 丁健 | 男 | 100 |

| 丁\_健 | 男 | 90 |

只会显示两表中都有的colum，不会将某一表中有而另一表中没有的显示出来，但join可以实现将所有的colum都显示出来的效果(没有的列值补为null)

外连接分为**左外连接**(列出左边的所有colum)和**右外连接**(列出右边的所有colum)

**语法：**

**select 列名 from 表名1 left|right outer join 表名2 on(条件)**

**eg：**

select student.name,sex,score from student left outer join score on(student.name=score.name);

#### 2.6.3联合查询

联合查询是指将两条或两条以上的select语句查询结果通过**并、交、差**运算得到新的查询结果

语法：

**select语句 UNION [distinct|all] select语句**

**union默认为distinct去除重复**

eg：

查询1班的学生和男同学的信息

select \* from student where class='1' union all select \* from student where sex='男';

**注意：order by在联合查询中需要加括号，同时必须搭配limit**

eg：

(select \* from student where class='1' order by snum desc limit 99999) union all (select \* from student where sex='男' order by snum desc limit 99999);

#### 2.6.4嵌套查询

**(1)不相关子查询**

子查询结果不依赖于父查询的嵌套查询称为不相关子查询，注意，order by子句只能用于父查询而不能用于子查询排序

**当子查询结果只有一行时可使用等号或in，子查询结果有多行时必须使用in**

eg：

select \* from student where class=(select classNum from class where stuNum=50);

select \* from student where class in(select classNum from class where stuNum=50);

**谓词all和any(some)，all是指当前记录中每一个结果，any是当前记录中的某一个结果，因此，>all意味着大于所有的结果，>any意味着大于结果集中最小的一个即可**

eg：

select \* from score where score > any(select score from score where Cnum=2) order by Cnum desc;//查找所有学生中比2号学科学生最低成绩高的学生成绩信息

select \* from score where select avg(score) from score group by Snum

**带有exists谓词的子查询只返回真值0(false)或1(true)，不返回数据，子查询有内容返回1，否则返回0，还有not exists谓词**

**(2)相关子查询**

相关子查询是指引用了父查询列的子查询，即子查询会对父查询的每行进行一次计算

### 2.7修改数据库alter

**1.修改数据库**

语法：

alter database/table [if exists] 数据库名/表名 修改语句

eg：

alert database mydb character set utf8;

alert database mydb collate utf8\_general\_ci;

alert table student engine innodb;

**注意： 1.对数据库的修改主要是修改字符集，校验规则**

**2.对表可以修改字符集，校验规则和引擎**

### 2.8字符集及数据乱码问题

**use 数据库名;show variables like "character\_set%";可以查看当前服务器各接口的默认字符集**

| character\_set\_client | utf8 |

| character\_set\_connection | utf8 |

| character\_set\_database | utf8 |

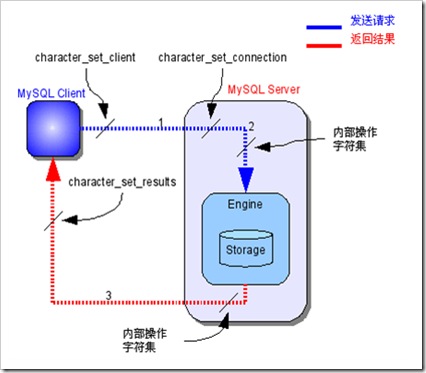
| character\_set\_filesystem | binary |

| character\_set\_results | utf8 |

| character\_set\_server | utf8 |

| character\_set\_system | utf8 |

| character\_sets\_dir | c:\wamp64\bin\mysql\mysql5.7.14\share\charsets\



**1.character\_set\_client是数据库默认的客户端来的数据字符集，即数据库接受读取数据使用的字符集**

**2. character\_set\_server是内部操作使用的字符集**

**3. character\_set\_database是当前所在数据库存储数据使用的字符集**

**4. character\_set\_results是返回结果使用的字符集**

**5.我们在使用控制台时，由于控制台的字符集只能是gbk，而我们读取使用utf8，因此存入数据出现乱码，若要解决问题，只需要：set character\_set\_client=gbk;即可，但此时通过控制台(gbk)查询仍然乱码，只需要set character\_set\_results=gbk;以上修改只是会话级别，重新连接数据库后修改失效，若要永久生效，需要set names 字符集**

## 3.php中的MySQL操作

### 3.1 $\_POST[]和$\_GET[]数组

**$\_POST[]**和**$\_GET[]**是php中预定义的全局数组，它包含了表单提交的所有数据，数组的key值是表单中input元素中对应的name属性值

### 3.2通过php使用数据

#### 3.2.1.mysql

mysql连接数据库是通过函数完成的，纯面向过程

步骤：

1. **接受表单传入的数据**

通过$\_POST[]和$\_GET[]数组接受传入的数据

**注意： from表单的action属性应设为php文件的url，method指定传输方法**

1. **连接数据库**

通过**$conn= new mysql\_connect (host , username , password , newLink);**函数连接数据库

**注意： (1)sever表示要连接的MySQL 服务器，可以包括端口号；**

**(2)在脚本结束前用同样的参数第二次调用 mysql\_connect()时，将不会建立新连接，而将返回已经打开的连接，可以通过修改bool类型的newLink来打开新连接，脚本一结束，到服务器的连接就被关闭，除非之前已经明确调用 mysql\_close() 关闭了，要创建一个持久连接，应使用 mysql\_pconnect() 函数**

**(3)该函数返回一个mysql连接资源，该连接可以转换成bool值以判断连接是否成功**

1. **选择数据库**

通过**mysql\_select\_db(dbName [,mysqlLink])**函数选择数据库，mysqlLink是数据库连接资源，如不指定连接，则使用由 mysql\_connect() 最近打开的连接,其返回值也能判断是否选择成功

eg：

$link = mysql\_connect('localhost', 'mysql\_user', 'mysql\_password',true);

if (!$link) {

die('Not connected : ' . mysql\_error());

}else{

$db\_selected = mysql\_select\_db(‘mydb’ , $link);

if($db\_selected){

echo “选择数据库成功”;

}

}

1. **执行sql语句**

通过**mysql\_query(query [ ,mysqlLink])**函数执行mysql语句

注意： (1)query是要执行的语句

(2)mysqlLink是数据库连接资源，如不指定连接，则使用，由 mysql\_connect() 最近打开的连接,其返回值也能判断是否选择成功

(3) 对 select，show，describe, explain 和其他语句 语句返回一个结果 resource，如果查询出现错误则返回 FALSE

对INSERT, UPDATE, DELETE, DROP等语句，执行成功时返回 **TRUE**，出错时返回 **FALSE**。

该方法在5.5版本已废弃！

eg：

$sql = SELECT \* from student WHERE name=’丁健’;

mysql\_query($sql , $link);

1. **查询数据**

在将查询语句执行以后，将返回一个结果resource，然后通过**mysql\_fetch\_assoc(结果resource)**函数一while循环方式将从数据库查询到的结果一条一条地取出，返回值是一个数据记录的数组，代表一条记录

该方法在5.5版本已废弃！

**mysql\_num\_rows(结果resource)获取结果集的记录数**

eg：if(mysql\_num\_rows($resultRes)>0){

while ($row = mysql\_fetch\_assoc($resultRes)) {

echo $row["name"];

echo $row["age"];

echo $row["sex"];

}

}

1. **错误处理**

通过mysql\_errno()错误代码函数和mysql\_error()错误信息函数输出错误信息,建议将其封装为函数

header(“content-type:text/html;charset=utf-8”);

$link = @mysql\_connect('localhost', 'mysql\_user', 'mysql\_password',true);//@用于抵制错误信息

if(!$link){

echo “错误代码：”. mysql\_errno().’<br>’;

echo “错误信息：”. iconv(‘gbk’ , ’utf-8’ , mysql\_error()).’<br>’;// iconv将控制台的gbk转码为utf-8

exit;

}

#### 3.2.2.mysqli扩展

mysqli扩展是一种面向过程＋面向对象的扩展

#### 3.2.3.PDO扩展

PDO是一种纯面向对象的扩展，推荐使用PDO

## 4. MySQL的备份和恢复

**(必须)在cmd命令行下**：

备份语法：

**mysqldump -u root -p 数据库名>数据存放路径/文件名.back**

**注意：这里后面一定不能加”;”!**

eg：

mysqldump -u root -p db2 > E:\myWeb\new.bak

**(必须)在mysql命令行下**：

恢复语法：

**source 数据存放路径.sql;**

注意：在恢复应在某个已创建好的数据库中使用，source命令会将.bak中的数据恢复到该数据库中,恢复时，数据库路径应使用正斜杠”/”。

eg：

create …..

use ….;

source E:\myWeb\new.bak;

注意：

**1.如果只希望备份某张表：**

**mysqldump -u root -p 数据库名 表名1 表名2…>数据存放路径/文件名.back**

**2.如果希望备份多个数据库：**

**mysqldump -u root -p -B 数据库名1数据库名2…>数据存放路径/文件名.back**

**3.如果备份时带上了-B，那么恢复时不需要创建空数据库并使用，直接source即可，因此建议以后备份数据库都带上-B**

## 5. 数据库的设计和优化

### 5.1 范式

**(1)第一范式(1NF)**

第一范式要求数据具有原子性，即字段的数据不可再分

eg：

下表中**时间**字段可分为上课时间和下课时间，因此不符合第一范式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程号 | 课程名 | 教职工编号 | 时间 | 学分 |
| 001 | 离散数学 | 001 | 09:35:00,11:00:00 | 2 |
| 002 | 编译原理 | 002 | 09:35:00,12:00:00 | 2 |
| 003 | 声乐艺术 | 003 | 07:35:00,09:00:00 | 1 |

**(2)第二范式(2NF)**

在第一范式的基础上，如果表中的主键是复合主键，且表中有不依赖于整个复合主键来确定的字段(如依赖于部分主键确定)，则该表不符合第二范式

第二范式有两种解决方案，**一种是分开复合主键，单独成表，另一种是在前面添加记录序号作为主键，推荐前一种**

eg：

下表中红色部分构成主键，因为同一种课可能有不同的老师来教，但是**性别**字段和**课程名**字段不依赖于全部主键

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 课程号 | 教职工编号 | 性别 | 课程名 | 时间 | 学分 |
| 001 | 001 | 男 | 离散数学 | 09:35:00,11:00:00 | 2 |
| 002 | 002 | 男 | 编译原理 | 09:35:00,12:00:00 | 2 |
| 003 | 003 | 女 | 声乐艺术 | 07:35:00,09:00:00 | 1 |

**(3)第三范式(3NF)**

在第二范式的基础上，如果表中存在字段不是直接依赖于主键，而是通过依赖某个非主键字段来传递依赖主键，则该表不符合第三范式

eg：

下表中**性别**字段依赖**教职工编号**字段，**课程名字**段依赖**课程号**字段，

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | 课程号 | 教职工编号 | 性别 | 课程名 | 时间 | 学分 |
| 1 | 001 | 001 | 男 | 离散数学 | 09:35:00,11:00:00 | 2 |
| 2 | 002 | 002 | 男 | 编译原理 | 09:35:00,12:00:00 | 2 |
| 3 | 003 | 003 | 女 | 声乐艺术 | 07:35:00,09:00:00 | 1 |

解决方案：将上表中传递依赖拆开，单独成表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | 课程号 | 教职工编号 | 时间 | 学分 |
| 1 | 001 | 001 | 09:35:00,11:00:00 | 2 |
| 2 | 002 | 002 | 09:35:00,12:00:00 | 2 |
| 3 | 003 | 003 | 07:35:00,09:00:00 | 1 |

|  |  |
| --- | --- |
| 教职工编号 | 性别 |
| 001 | 男 |
| 002 | 男 |
| 003 | 女 |

|  |  |
| --- | --- |
| 课程号 | 课程名 |
| 001 | 离散数学 |
| 002 | 编译原理 |
| 003 | 声乐艺术 |

**(4)逆规范化**

有时候为了加快查询速度，会违反范式将查询次数较多的数据放到一张表中，避免多表查询，但这样会增加数据冗余度

## 6. 视图

### 6.1 视图基本使用

视图是一种有结构但没有结果的虚拟表，它是将表的各字段拆分组合而来，他可以将一条复杂的sql语句使用视图进行保存，简化操作，由于视图是一张虚拟表，对其增删改是不会影响基表，因此相对安全，视图还可以隐藏重要信息，不必将基表提供给别人操作，只将视图提供给别人，防止不相关的信息泄露

**基本语法：**

**create [algorithm = 指定算法] view 视图名 as select语句 [with check option]**

**注意： 1. algorithm是指定使用的视图算法，默认为undefined**

**2. with check option是指在通过视图更新数据时，不能将数据更新为不符合select语句的where条件所限制的数据，eg：create view myview as select\* from student where age>30 with check option;意味着通过视图更新数据时，不能将age设为<=30的数**

eg：

create view my\_view\_1 as select score.Snum,score.Cnum,student.name,sex,class,score from student left outer join score on student.snum = score.Snum;

### 6.2 视图的基本操作

可以把视图当做表进行查看结构、查看数据等操作，此时的视图就相当于一张表，视图的本质就是讲select语句的查询结构封装成一个表，同时也可以通过视图进行增删改

**修改视图：**

**alter view 视图名 as 新的select语句;**

**删除视图：**

**drop view 视图名;**

**视图也可以进行增删改，语法同表**

**注意：多表视图不能插入、删除数据，单表视图插入数据必须符合基表的结构，单表和多表都能更新数据，但可能会有更新限制with check option**

### 6.3 视图的解析方式和算法

视图有两种解析方式：

**undefined：**未定义(默认的)，这不是一种实际使用的算法，是一种推卸责任的算法—-告诉系统，视图没有定义算法，你看着办。

**Temptable临时表算法：**先执行创建视图的select语句，再执行查询视图的select语句

**Merge合并算法：**将创建视图的select语句和查询视图的select语句合并执行

Merge算法的效率高，但有时可能会出现一些意料之外的情况，如:

create view my\_view\_1 as select \* from score order by score ;

select score from my\_view\_1 group by score;

将会先执行group by再执行order by，因为它默认算法是undefined，系统经常使用merge算法，若要使用Temptable算法，应在创建视图时给出

## 7. 变量

### 7.1 系统变量

系统变量是系统定义好的变量，诸如：autocommit，auto\_increment\_increment等

**1.查看系统变量**

我们可以通过

show variables like 'autocommit';

show variables like ' auto\_increment %';//查看auto\_increment\_increment 和auto\_increment\_offset变量

select @@autocommit;

等方式来查看系统变量

**2.修改系统变量**

修改系统变量分为**会话级(当次连接生效)**和**全局级(永久生效)**

会话级：

**set 变量名=值;**

全局级：

**set global 变量名=值;**

### 7.2 自定义变量

为了区分系统变量和自定义变量，自定义变量必须使用@符号定义，所有的自定义变量都是会话级

**语法：**

**set @变量名 := 值;**

**查看方式：**

**select @变量名**

mysql允许从表中获取数据并赋值给变量，这里有两种方法：

1. 赋值并查看

**select @变量名 := 字段名 from 表名**

select @name := name from account where ID='1001';

1. 只赋值不查询(一次只能用一行数据去赋值)

**select 字段名 from 表名 where 条件 into @变量名**

select name,sex from account where ID='1002' into @name,@sex;

## 8. 触发器

事先为某张表绑定一段代码，当表中某些内容发生改变(增删改)的时候，系统自动触发代码并执行

触发器事件类型有三种，分别为insert、delete、update，触发时间有两种：before和after，所以共有6种触发器，一张表也最多有6个不同的触发器

### 8.1 创建触发器

**语法：**

**delimiter 自定义符号//将后续代码的语句结束符改为自定义，防止触发器内容部分提前结束**

**create trigger 触发器名 触发时间 事件类型 on 表名 for each row**

**begin**

**触发器内容;**

**end**

**自定义符号**

**delimiter ;//将语句结束符改回’;’**

eg：

delimiter &&

create trigger my\_trigger after insert on my\_order for each row

begin

update goods set num=num-1 where id=1;

end

&&

delimiter ;

### 8.2 查看和删除触发器

**查看所有触发器语法：**

**show triggers\G;**

**查看指定触发器语法：**

**show create trigeer 触发器名;**

**删除触发器：**

**drop trigger 触发器名;**

### 8.3 触发器记录

不管触发器是否触发，系统都会将要操作的数据行触发执行的前后状态(数据)保留下来，供触发器使用，操作前的数据保留在old中，操作后的数据保留在new中，因此，删除操作没有new，插入操作没有old

**语法：**

**new/old.字段名**

eg:

delimiter $

create trigger my\_trigger after insert on my\_order for each row

begin

update goods set num=num-new.num where id=new.goodsId;

end

$

delimiter ;

### 8.4 if分支

**语法：**

**if 判断条件 then**

**满足条件所执行的代码;**

**[else**

**不满足条件所执行的代码;]**

**end if;**

eg：

delimiter $

create trigger my\_trigger before insert on my\_order for each row

begin

select num from goods where id=new.goodsId into @remainGoodsNum;

if @remainGoodsNum>new.num then

update goods set num=num-new.num where id=new.goodsId;

else

insert into xxx value(xxx); //暴力报错

end if;

end

$

delimiter ;

## 9. 函数

### 9.1 while循环

**语法：**

**[循环名字:]while 条件判断 do**

**循环体代码;**

**循环控制;**

**end while;**

在mysql中也有类似continue和break的语法，iterate是continue，leave是break,

**语法：**

**iterate/leave 循环名字;**

eg:

delimiter $

create function foo(a int , b int) returns int

begin

set @sum:=0;

while b>0 do

set @sum=@sum+a;

set b=b-1;

end while;

return @sum;

end

$

delimiter ;

### 9.2 系统函数

任何函数都有返回值，因此函数调用是通过select进行，这样才可以查看结果eg：

1. **substring(字符串,起始位置,长度)**

字符串截取函数，eg;

set @cn='你好！世界';

set @en='hello!world';

select substring(@cn,1,2);//结果为’你好’

1. **char\_length(字符串)**

字符长度函数，eg:

select char\_length(@cn); //结果为’5’

1. **length(字符串)**

字节长度函数，只与建表时选用的字符集有关，eg:

select length(@cn); //结果为’10’，因为是utf-8，一个中文3字节

1. **instr(原始字符串,目标字符串)**

判断原始字符串中是否含有目标字符串，eg:

select instr (@cn,'你好');//结果为’1’

1. **Lpad(原始字符串,填充后长度,用于填充的字符串)**

左填充，将填充字符串重复填充到原始字符的左边，最终长度为’ **填充后长度**’ ,eg:

select Lpad (@cn,10,'你好');//结果为’ 你好你好你你好！世界’

1. **insert(原始字符串,起始位置,替换长度,用于替换的字符串)**

替换字符串,eg:

select insert (@en,1,5,'hey');//结果为’ hey!world’

1. **strcmp(第一个字符串, 第二个字符串)**

字符串比较,判断第一个字符串是否大于第二个字符串，大写字母小于小写字母，字母表靠后的字母较大，eg:

set @f='hello';

set @s='hey';

select strcmp (@f, @s);//结果为’ -1’

### 9.3 自定义函数

**语法：**

**delimiter 自定义符号**

**create function 函数名([形参名 形参类型]) returns 数据类型**

**begin**

**函数体;**

**返回值;**

**end**

**自定义符号**

**delimiter ;**

eg：

delimiter $

create function foo(a int , b int) returns int

begin

return a+b;

end

$

delimiter ;

**查看自定义函数：**

**show function status [like'']\G;**

**查看函数创建语句：**

**show create function 函数名;**

**删除函数：**

**drop function 函数名;**

## 10. 存储过程

存储过程是一组为了完成特定功能的SQL 语句集，可以看做一种没有返回值的函数

### 10.1创建存储过程

**语法：**

**delimiter 自定义符号**

**create procedure 过程名字([参数限定 形参名 形参类型])**

**begin**

**过程体;**

**end**

**自定义符号**

**delimiter ;**

**查看存储过程：**

**show procedure status [like’’]\G;**

**查看存储过程创建语句：**

**show create procedure 存储过程名;**

**删除函数：**

**drop procedure 存储过程名;**

### 10.1调用存储过程

**语法：**

**call 存储过程名([实参列表]);**

### 10.2存储过程的参数限定

存储过程拥有比函数更严格的参数限定，分为三种：

**in(值传递)：**数据只是从外部传输给内部使用，可以是数值也可以是变量

**out(引用传递)：**被限定为out的形参，无论实参传递什么数值，在存储过程内部都会被初始化为null，out也是引用传递，即修改内部的值会影响外部

**inout(引用传递)：**

**注意： 1.out和inout限定的参数，实参只能为变量不能为数值，否则报错**

**2.存储过程修改全局变量(带@符号的变量)具有滞后性，会先执行过程，执行结束后再进行修改**



eg：

delimiter /

create procedure mypro(in int\_1 int, out int\_2 int, inout int\_3 int)

begin

select int\_1, int\_2, int\_3;//无论传入的值如何，int\_2的值一定为null

end

/

delimiter ;

set @int1=6;

set @int2=7;

call mypro (4,@int1,@int2);//结果为 4 | NULL | 7

select @int1,@int2;//结果为NULL | 7

delimiter /

create procedure mypro(in int\_1 int, out int\_2 int, inout int\_3 int)

begin

set int\_1=10;

set int\_2=100;

set int\_3=1000;

select @int\_1, @int\_2, @int\_3;-- 结果为 NULL | NULL | NULL

end

/

delimiter ;

set @int1=6;

set @int2=7;

set @int3=9;

call mypro (@int1,@int2,@int3);

select @int1,@int2,@int3;

//结果为 6 | 100 | 1000 |

set @int\_1= 'a';

set @int\_2='b';

set @int\_3='c';

select @int\_1, @int\_2, @int\_3; -- 结果为 a | b | c

# 二、高级特性

## 1. 锁

锁是编程技术中最常用的控制并发的一种强有力的手段，这里的锁不同于多线程中的锁，而是数据库中的锁，因为数据也是多个用户共享的资源。为了保证数据并发访问的一致性和有效性，我们需要锁机制的帮助，但是过多的锁会导致数据库性能下降。因此如何使用锁机制是一门真正的艺术

Mysql根据锁的粒度大小可分为：

·表级锁

·行级锁

·页面锁

MyISAM只支持表级锁

InnoDB 存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁

BDB 存储引擎采用的是页面锁（page-level locking），但也支持表级锁

默认情况下，表锁和行锁都是自动获得的， 不需要额外的命令。

但是在有的情况下， 用户需要明确地进行锁表或者进行事务的控制， 以便确保整个事务的完整性，这样就需要使用事务控制和锁定语句来完成。

根据锁的模式分为：

·共享锁(读锁，S锁)

·排他锁(写锁，X锁)

根据使用方式又可以分为：

·悲观锁

·乐观锁

根据写法又可以分为：

·隐式锁(自动加锁)

·显式锁(手动加锁)

共享锁、排它锁。表锁、行锁、页级锁。这些锁一般都是自动加锁。不用去管它，只需要知道在什么时候MYSQL会去加锁就行。当然也可以手动加锁，这就是显式锁

接下来我们将一一讲解这些锁的机制

### 1.1粒度锁

根据锁的粒度，可以将锁分为表级锁、行级锁和页面锁



由于BDB已经被InnoDB所取代，我们只讨论MyISAM和InnoDB的锁问题

#### 1.1.1.表级锁

##### 1.1.1.1表级锁概述

表级别的锁定是MySQL各存储引擎中最大颗粒度的锁定机制，它直接将整张表锁住，所以实现起来非常简单，所以**获取锁和释放锁的速度很快，但是其并发效率是最低的，不会发生死锁，锁冲突(锁争用)概率极高**。

**表级锁和行级锁不同，表级锁在mysql服务器层实现，与存储引擎无关**

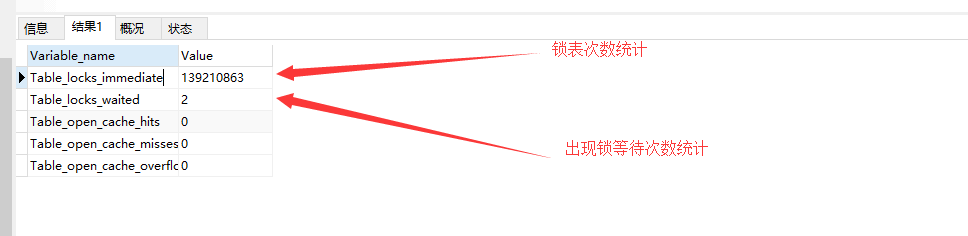
表级锁又分为：

**·表共享读锁(Table Read Lock)：**不会阻止其他语句对同一表的读请求，但会阻塞对同一表的写请求(允许其他语句读但不允许它们写，所以叫共享锁)；

**·表独占写锁(Table Write Lock)：**会阻止其他语句对同一表的读和写操作(自己独占)；

当一个线程获得对一个表的写锁后， 只有持有锁的线程可以对表进行读写操作。 其他线程的读写操作都会等待，直到锁被释放为止。但是默认情况下，写锁比读锁具有更高的优先级：当一个锁释放时，这个锁会优先给写锁队列中等候的获取锁请求，然后再给读锁队列中等候的获取锁请求。这也正是 **MyISAM 表不太适合于有大量更新操作应用**的原因，大量更新操作必然伴随着大量加锁，查询操作很难获得读锁从而一直被阻塞

使用show status like 'table%'指令可以查看数据库内部表级锁资源争用情况



##### 1.1.1.2如何加表级锁

MyISAM 在执行查询语句（SELECT）前，会自动给涉及的表加读锁，在执行更新操作(UPDATE、DELETE、INSERT)前，会自动给涉及的表加写锁，这个过程并不需要用户干预(默认)，在自动加锁的情况下，**MyISAM 总是一次获得 SQL 语句所需要的全部锁**，这也正是 MyISAM 表不会出现死锁（Deadlock Free）的原因。

MyISAM中表锁一般不需要手动加(也可以手动加)，InnoDB中默认使用行锁，如果希望使用表锁，就需要显式加锁

**语法：**

**加锁：lock tables 表名1 write|read[,表名2 write|read …];**

**释放当前会话持有的所有锁：unlock table;**

eg:

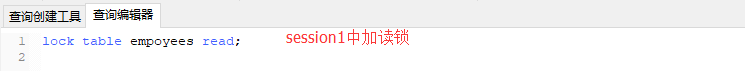
lock table haha READ;//加读锁(S锁)

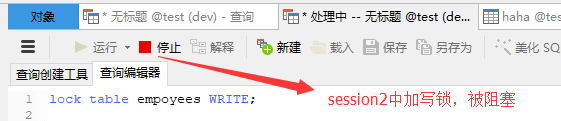
select \* from haha;//查询操作

unlock tables;//释放锁

lock table empoyees read; //加读锁(S锁)

INSERT INTO `empoyees` (`id`, `employee\_name`, `team\_id`) VALUES ('12', '辛吉德', '3'); //在本session执行插入操作，报错，在其他session中插入则等待本session释放S锁





**Session\_1中 ：** lock table haha WRITE; //加写锁(X锁)

select \* from haha;

select \* from haha; //两次查询都在session\_1中，该session持有X锁，因此都有结果

//重新新建查询，查询被阻塞



**Session\_1中 ：** lock table haha WRITE,empoyees WRITE; //给两张表加写锁(X锁)

select \* from haha; //正常查询

**Session\_2中：** select \* from haha; //被阻塞

**Session\_1中：** unlock table; //释放当前会话持有的所有锁

##### 1.1.1.3显式表锁和隐式表锁

一般来说，MyISAM不需要我们手动加表锁，一切由引擎自动加锁，但有时也需要手动加表锁来模拟事务操作。比如，现在有两张表orders和order\_detail，orders是存放所有订单的表(包括订单本身的属性如订单号，订单总金额，订单时间等信息)，而order\_detail是存放所有订单中商品信息的表(包括订单中商品的属性如商品名，所属订单，购买数量，金额小计等信息)，如果现有一需求，统计orders中订单总额的总计金额是否和order\_detail中的所有金额小计的总金额相等：

Select sum(total) from orders;

Select sum(subtotal) from order\_detail;

如果不显式加锁，在第一次查询后，order\_detail表可能已经发生了改变，导致双方数据不一致，这时候就需要通过加锁来模拟事务操作

InnoDB默认使用行锁，如果希望使用表锁，请显式加锁

**需要注意的是：**

**1．mysql不支持锁升级，也就是说，无论是MyISAM还是InnoDB，一旦显式加锁后，就只能访问已加锁的表，不能访问未加锁的表**，eg：

lock table haha read;

select \* from empoyees;//同一session中访问未显式加锁的表，报错

**2．显示加锁时，如果加锁后执行的语句中用到了表的别名，也必须对别名加锁，否则报错**，**另外，如果对表名加读锁，对表别名加了写锁，默认按写锁执行。**eg：

lock table empoyees WRITE;

select \* from empoyees as empoyees\_1//同一session中别名empoyees\_1未加锁，报错

lock table empoyees WRITE,empoyees as empoyees\_1 WRITE; //正确的加锁方式

lock table empoyees read,empoyees as empoyees\_1 WRITE; //先对表加读锁，再对表别名加写(X)锁，默认按X锁执行

select \* from empoyees as empoyees\_1

##### 1.1.1.4并发插入

在一般情况下，MyISAM表的读和写是串行的，即读操作与写操作、两次写操作之间串行执行，但这是就总体而言的。在一定条件下，MyISAM表也支持查询和插入操作的并发进行，MyISAM存储引擎有一个系统变量concurrent\_insert，专门用以控制其并发插入的行为，其值分别可以为0、1或2。

·当concurrent\_insert设置为0时，不允许并发插入。

·当concurrent\_insert设置为1时，如果MyISAM表中没有空洞（即表的中间没有被删除的行），MyISAM允许在一个进程读表的同时，另一个进程从表尾插入记录。这也是MySQL的默认设置。

·当concurrent\_insert设置为2时，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录。

**注意：session1获取表的读锁后，即使session2成功并发插入了一条新记录到表的末尾，但在session1释放读锁前，session1都无法访问新记录，并且，session只能并发插入到表的末尾，不能更新和删除(这两类操作依然会等待锁)**

+----+------+------+------+------+------+------+

| id | name | age |

| 1 | DJ | 18 |

| 2 | wJ | 8 |

+----+------+------+------+------+------+------+

|  |  |
| --- | --- |
| **Session1** | **Session2** |
| lock table test read local; |  |
|  | insert into test(name,age) values("ss",20); |
|  | select \* from test;  +----+------+------+------+------+------+------+  | id | name | age |  | 1 | DJ | 18 |  | 2 | wJ | 8 |  | 3 | ss | 20 |  +----+------+------+------+------+------+------+ |
| select \* from test;  +----+------+------+------+------+------+------+  | id | name | age |  | 1 | DJ | 18 |  | 2 | wJ | 8 |  +----+------+------+------+------+------+------+ |  |
| unlock table;  select \* from test;  +----+------+------+------+------+------+------+  | id | name | age |  | 1 | DJ | 18 |  | 2 | wJ | 8 |  | 3 | ss | 20 |  +----+------+------+------+------+------+------+ |  |

**想要使用并发插入，必须使用read local和write local来代替reade和write获取锁**

read/write local和read/write的区别在于：read local允许在表被加锁时，执行非冲突性INSERT语句(并发插入)，如上例，如果session1加的是read锁而不是read local锁，那么session2在执行insert into test(name,age) values("ss",20); 语句时就会被阻塞(session2必须等待session1释放共享锁)。当然了，**并发插入是MyISAM的特性，因此对于InnoDB来说read/write local和read/write没有区别**

可以利用MyISAM存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。例如，将concurrent\_insert系统变量设为2，总是允许并发插入，通过定期在系统空闲时段执行OPTIMIZE TABLE语句来整理文件碎片，收回因删除记录而产生的空洞。

#### 1.1.2.页面锁

页级锁是MySQL中比较独特的一种锁定级别，在其他数据库软件中也并不是太常见。页级锁的特点是锁定颗粒度介于行级锁与表级锁之间，所以**获取锁和释放锁的速度，锁冲突程度，以及所能提供的并发处理能力也同样是介于其它二者之间，会发生死锁。**

#### 1.1.3.行级锁

##### 1.1.3.1行级锁概述

**锁的粒度最小，锁冲突也最小，获取锁和释放锁的速度较长，并发处理能力最强，但极易发生死锁。**

行级锁不是mysql自己实现的锁定方式，而是由存储引擎提供的，典型的如InnoDB等。各个存储引擎的实现方式有所差异，由于InnoDB是目前最为广泛使用的引擎，因此我们这里只讨论InnoDB的行锁

InnoDB的行级锁同样分为共享锁(S)和排他锁(X)，在锁定机制的实现过程中为了让行级锁定和表级锁定共存，InnoDB使用了意向锁(表级锁定)的概念，因此有了意向共享锁(IS)和意向排他锁(IX)

可以通过show status like 'innodb\_row\_lock%';来查看Innodb的行锁争用情况



##### 1.1.3.2意向锁

**场景：**

因为表锁覆盖了行锁的数据，所以表锁和行锁也会产生冲突。假设事务A对表中某行加了行锁(读锁)。紧接着，事务B希望获取整张表的X锁，这当然是不允许的，因为一旦允许事务B获取表X锁，那么意味着事务B对表的每一行都拥有写的权力，但事务A的行S锁不允许其他事务对锁定行进行写操作，二者冲突，因此数据库是不允许事务B加表X锁的，那么问题来了，数据库是怎么判断事务B的锁请求会与事务A的锁冲突呢？

第一步：判断表是否已被其他事务用表锁锁表

第二步：判断表中的每一行是否已被行锁锁住。

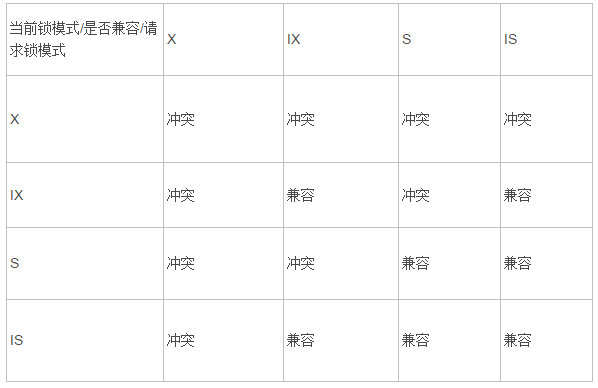
第一步没什么，但第二步的效率让人着实不敢恭维，如果表中数据量到达百万级甚至千万级，那么判断加锁的开销是极其恐怖的。所幸，意向锁帮我们解决了这个问题。

意向锁是一种表锁，它是一种行锁的标志，一张表被上了IX意味着表中存在被X上锁的行，同样的，一张表被上了IS意味着表中存在被S上锁的行。在InnoDB中，上行锁之前，必须要对表上对应的意向锁表明表的状态。因此，上面的步骤可以变为。

第一步：不变

第二步：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务B申请整张表的写锁会被阻塞。

需要注意的是申请意向锁的动作是数据库完成的，就是说，事务A申请一行的行锁的时候，数据库会自动开始先申请表的意向锁



需要注意的是：

**意向锁是为了解决表锁和行锁冲突的，是表级锁(是一种标记锁)，不会和行级的X，S锁发生冲突。只会和表级的X，S发生冲突，上表中的X和S是表级的X，S**

##### 1.1.3.3共享行锁和排他行锁

对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；对于普通SELECT语句，InnoDB不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。

**共享锁(S)：SELECT ... LOCK IN SHARE MODE**

**排他锁(X)：SELECT ... FOR UPDATE**

InnoDB会根据隔离级别在需要的时候自动加锁；锁只有在执行commit或者rollback的时候才会释放，并且所有持有的锁都是在同一时刻被释放。

为了不自动释放行锁，我们关闭自动提交

set GLOBAL autocommit=off; //关闭全局提交

**session\_1中执行：**

select \* from empoyees where id=1 for UPDATE; //为id=1的记录加X锁

**session\_2中执行：**

UPDATE empoyees set employee\_name="丝丝" where id=1 //试图更新id=1的记录，锁等待

select \* from empoyees where id=1//select不加任何锁，因此不会等待X锁，正常执行

select \* from empoyees where id=1 lock in share mode; //试图加S锁，等待session\_1释放id=1记录的X锁

##### 1.1.3.4显式行锁的应用场景

上一节我们提到，一般来说InnoDB会根据隔离级别在需要的时候自动加锁，锁只有在执行commit或者rollback的时候才会释放，并且所有的锁都是在同一时刻被释放。这是隐式行锁

显式行锁是：

**共享锁(S)：SELECT ... LOCK IN SHARE MODE**

**排他锁(X)：SELECT ... FOR UPDATE**

那么何时使用这两种显示行锁呢？

**select for update：**

为了让自己查到的数据确保是最新数据，不会在自己查询时修改数据，即使有，也是自己的修改，就需要用到select for update 子句。相当于一个update语句

**select lock in share mode：**

为了确保自己查到的数据没有被其他的事务正在修改，也就是说确保查到的数据是最新的数据，并且不允许其他人来修改数据。但是自己不一定能够修改数据，因为有可能其他的事务也对这些数据使用了in share mode 的方式上了 S 锁。如果没有其他事务上锁，自己就可以修改

##### 1.1.3.5行锁的实现方式

InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：**只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。**

由于MySQL的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。应用设计的时候要注意这一点。(需要注意的是主键是一种索引)

eg:

+---------+---------+---------+

| id | name | team\_id|

+---------+---------+---------+

| 1 | hahi | 5 |

| 2 | xix | 5 |

+---------+---------+---------+

假设team\_id为索引字段

**Session\_1中：**select \* from empoyees where team\_id = 5 and name = 'hahi' for update;

**Session\_2中：**select \* from empoyees where team\_id = 5 and name = 'xix' for update;

虽然访问的是不同行，但由于检索使用的索引一致(team\_id=5)，因此出现锁冲突等待

**Session\_1中：**select \* from empoyees where team\_id = 5 for update;

**Session\_2中：**select \* from empoyees where name = 'xix' for update;//等待锁

即使在使用行锁时使用了索引，但是否使用索引是由mysql决定的，大部分情况下都会使用索引，但如果表过小，可能mysql会认为全表扫描更快，就不会走索引，此时就是锁表而非锁行

##### 1.1.3.6间隙锁

当我们用范围条件而不是相等条件检索数据，并请求锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但实际并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁），如果使用相等条件请求给一个不存在的记录加锁，InnoDB也会使用间隙锁。。

说人话就是假设现有102条数据，1、2…100、101、102，我们要对id>100的记录加锁：select \* from empoyees where id> 100 for update; mysql不止会对101和102加锁，还会对id>102的记录(实际并不存在)加上间隙锁

那么InnoDB为什么要使用间隙锁呢？有如下两个目的：

**1.防止幻读，以满足相关隔离级别的要求。**对于上面的例子，如果不使用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；

**2.为了满足其恢复和复制的需要。**在数据库恢复和主从复制中作用很大

当然，间隙锁也会影响InnoDB的性能，造成在锁定的时候无法插入锁定键值范围内的任何数据。

  InnoDB存储引擎中不同SQL在不同隔离级别下锁比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 隔离级别          一致性读和锁  SQL | | Read Uncommited | Read Commited | Repeatable Read | Serializable |
| SQL | 条件 |  |  |  |  |
| select | 相等 | None locks | Consisten read/None lock | Consisten read/None lock | Share locks |
| 范围 | None locks | Consisten read/None lock | Consisten read/None lock | Share Next-Key |
| update | 相等 | exclusive locks | exclusive locks | exclusive locks | Exclusive locks |
| 范围 | exclusive next-key | exclusive next-key | exclusive next-key | exclusive next-key |
| Insert | N/A | exclusive locks | exclusive locks | exclusive locks | exclusive locks |
| delete | 相等 | exclusive locks | exclusive locks | exclusive locks | exclusive locks |
| 范围 | exclusive next-key | exclusive next-key | exclusive next-key | exclusive next-key |
| Select ... from ... Lock in share mode | 相等 | Share locks | Share locks | Share locks | Share locks |
| 范围 | Share locks | Share locks | Share Next-Key | Share Next-Key |
| Select \* from ... For update | 相等 | exclusive locks | exclusive locks | exclusive locks | exclusive locks |
| 范围 | exclusive locks | Share locks | exclusive next-key | exclusive next-key |
| Insert into ... Select ...  （指源表锁） | innodb\_locks\_unsafe\_for\_binlog=off | Share Next-Key | Share Next-Key | Share Next-Key | Share Next-Key |
| innodb\_locks\_unsafe\_for\_binlog=on | None locks | Consisten read/None lock | Consisten read/None lock | Share Next-Key |
| create table ... Select ...  （指源表锁） | innodb\_locks\_unsafe\_for\_binlog=off | Share Next-Key | Share Next-Key | Share Next-Key | Share Next-Key |
| innodb\_locks\_unsafe\_for\_binlog=on | None locks | Consisten read/None lock | Consisten read/None lock | Share Next-Key |

### 1.2乐观锁和悲观锁

乐观锁和悲观锁其实都是并发控制的机制，同时它们在原理上就有着本质的差别

#### 1.2.1.乐观锁

##### 1.2.1.1乐观锁概述

乐观锁是一种思想，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，它本质并不是一种真正的『锁』，它并没有使用数据库提供的锁机制，在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，就会进行重试，在整个的执行过程中其实都没有对数据库进行加锁

乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行重试，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

##### 1.2.1.1乐观锁实现

最常见的乐观锁实现方式有两种，第一种是使用版本号，第二种是使用时间戳。

1. **版本号**

使用版本号实现乐观锁，需要给每张表添加一个version字段记录版本号，表示数据被修改的次数，在记录初始化时给该字段指定一个初始版本号，当需要更新记录时，在读取数据的同时也读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才提交更新，否则不断重试，直到操作成功。Eg：

**1.查询出商品信息**

**select (id,status,version) from t\_goods where id=#{id}**

**2.根据商品信息生成订单**

**3.修改商品status为2**

**update t\_goods set status=2,version=version+1 where id=#{id} and version=#{version};**

1. **时间戳**

使用时间戳(TimeStamp)。做法类似于版本号。

#### 1.2.2.悲观锁

悲观锁顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会请求锁直到它拿到锁。悲观锁的实现，往往依靠数据库本身提供的锁机制，上面所说的表锁、行锁等都是悲观锁。悲观锁相对保守，适合大量写入，数据争用激烈的场景，数据安全性高但效率低下，容易产生死锁

悲观锁的实现依赖于数据库提供的锁机制

### 1.3 数据库中的死锁

在两个或两个以上的进s程在执行过程中，由于互相间竞争对方持有的资源导致的相持不下的局面，被称作死锁。在mysql中，由于MyISAM会一次获取所需的所有相关的表锁，要么全部满足，要么等待，因此不会出现死锁。但在InnoDB中，除单个SQL组成的事务外，锁是逐步获得的，这就决定了在InnoDB中发生死锁是可能的。下面就是一个典型的死锁，eg：

|  |  |
| --- | --- |
| **事务1** | **事务2** |
| set autocommit = 0; |  |
|  | set autocommit = 0; |
| select \* from table\_1 where id=1 for update;  做一些其他处理... |  |
|  | select \* from table\_2 where id=1 for update; |
| select \* from table\_2 where id =1 for update;  因事务2已取得table\_2中id=1的排他行锁，因此事务1等待获取锁 |  |
|  | select \* from table\_1 where where id=1 for update;  事务2试图获取事务1中id=1的锁，相互等待，死锁发生 |
| commit; |  |
|  | commit; |

发生死锁后，InnoDB一般都能自动检测到，并使一个事务释放锁并回退，另一个事务获得锁，继续完成事务。但在涉及外部锁，或涉及表锁的情况下，InnoDB并不能完全自动检测到死锁，这需要通过设置锁等待超时参数innodb\_lock\_wait\_timeout来解决

下面有几种常见的避免死锁的手段：

1. **如果不同的程序会并发存取多个表，应尽量约定以相同的顺序来访问表**，这样可以大大降低产生死锁的机会。如上例中，两个sessin都约定以同一顺序先获取某张表中的行锁，再去拿另一张表的行锁，就不会发生死锁
2. 设置锁等待超时参数innodb\_lock\_wait\_timeout

## 2. 事务

事务是指一些列要发生的连续操作，如银行转账，要A账户减少钱并且B账户增加钱，事务安全是一种连续操作实现的机制，他可以保证事务操作的完整性

create table account(ID int primary key auto\_increment,name varchar(5) not null,sex enum('男','女') not null , balance int not null) auto\_increment=1000;

| ID | name | sex | balance |

| 1000 | DJ | 男 | 10000 |

| 1001 | 李丽 | 女 | 10000 |

| 1002 | 张铭可 | 女 | 20000 |

| 1003 | 丁子全 | 男 | 30000 |

**注意：只有InnoDB和BDB引擎提供事务安全表，MyISAM不支持事务操作**

### 2.1事务安全

事务操作分为自动操作和手动操作

1. **手动事务**
2. 首先开启事务，告诉系统以下操作不要直接写入数据表，而是先存放到事务日志，语法如下：

**start transaction;**

1. 进行一系列的事务操作，如：账户A钱减少，账户B钱增加，但是这些事务操作没有影响到基本表

update account set balance = balance-1000 where ID='1003';

update account set balance = balance+1000 where ID='1002';

1. 关闭事务，将事务日志中的操作的结果同步到数据表，此时有两种操作：

**提交事务**：操作成功，同步数据表，此时相当于将事务日志中的操作全部执行

语法：**commit;**

**回滚事务：**直接清空日志表，意为着操作失败，此时相当于将事务日志中的操作全部清除

语法：**rollback;**

1. **自动事务**

mysql的任何增删改操作都属于自动事务，语句执行后会立即同步到数据表中，我们可以autocommit来关闭自动事务，关闭之后需手动提交事务操作才能生效，语法：

set session autocommit=off; //关闭会话级自动提交

set GLOBAL autocommit=off; //关闭全局自动提交

update account set sex='女' where ID=1000;//无效，另开一个控制台来select

commit;//生效

show session variables like 'autocommit';

show GLOBAL variables like 'autocommit'; //查看自动提交状态

如果关闭了自动提交，仍在自动提交，那你一定用的是MyISAM……MyISAM不支持事务….

### 2.2事务操作的原理

事务操作实际上是将所有操作所有造成的结果都保存到事务日志中，只有在得到commit命令时才会同步到数据表，其他诸如：rollback，断电，断开连接等都会清空事务日志

事务操作具有锁机制，innodb默认是行锁(只锁住一行数据)，如果在事务操作中没有使用到索引，系统会自动全表检索数据(一行一行地找)，从行锁上升到表锁(锁住整张表)

### 2.3事务回滚点

在某个成功的操作完成后，后续的操作可能成功也可能失败，但无论后续操作如何，该操作都已成功，在此设立回滚点，在后续操作失败时可返回该处，而不是返回所有操作，如银行转账，余额不足操作失败，应返回上一步而不是重新输入密码

**语法：**

**savapoint 回滚点名**

**rollback to 回滚点名**

eg：

start tranaction;

update account set balance=balance+1000 where ID='1000';//发工资

savepoint koushui;

update account set balance=balance-1000\*0.05 where ID='1001';//扣错税

rollback to koushui;

update account set balance=balance-1000\*0.05 where ID='1000';//扣税

commit;

### 2.4事务隔离级别

我们都知道，事务应当具有以下特性：

原子性:事务不可分割

一致性:事务执行的前后，数据完整性保持一致.

隔离性:一个事务执行的时候，不应该受到其他事务的打扰

持久性:一旦结束，数据就永久的保存到数据库.

#### 2.4.1事务并发的副作用

如果不考虑事务隔离性，则会产生以下现象：

1. **丢失更新：**两个事务同时更新一行数据，交叉执行，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失。**(没有加任何锁)**

**Eg:**

|  |  |
| --- | --- |
| **事务1** | **事务2** |
| start transaction; |  |
|  | start transaction; |
|  | update user set name='李四' where id='10'; |
| update user set name='张三' where id='10'; |  |
| commit; |  |
|  | commit; |

假如原来user表内id为10的行，是一条{id:10，name:"王五"，age:15} 的数据，经过**事务1**修改后变成{id:10，name:"张三"，age:15}。**事务2**修改后，该数据变成了{id:10，name:"李四"，age: 15}。由**事务1**所执行的操作在**事务2**的提交后，数据被覆盖了。这个现象就叫做丢失更新。

**读未提交(Read UnCommitted)及以上隔离级别**可以防止该现象

1. **脏读**:一个事务读到另一个事务未提交数据，后一个事务如果操作失败进行了回滚，前一个事务读取的就是错误数据，这样就造成了脏读。

**Eg:**

|  |  |
| --- | --- |
| **事务1** | **事务2** |
| start transaction; |  |
|  | start transaction; |
| update user set name='张三' where id='10'; |  |
|  | select \* from user where id='10'; |
| rollback; |  |
|  | commit; |

假如原来user表内id为10的行，是一条{id:10，name:"王五"，age:15} 的数据,经过**事务1**的修改，name变为张三，但这个修改后的数据并未提交，而**事务2**读取到了**事务1**未提交的name(张三)，这就是脏读。一旦**事务1**rollback，修改不生效，**事务2**读取的数据就成了脏数据

**读已提交(Read Committed)及以上隔离级别**可以防止该现象。

1. **不可重复读**: 事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。**关注点在于修改**。

**Eg:**

|  |  |
| --- | --- |
| **事务1** | **事务2** |
| start transaction; |  |
| select \* from user where id='10'; |  |
|  | start transaction; |
|  | update user set name='李四' where id='10'; |
|  | commit; |
| select \* from user where id='10'; |  |
| commit; |  |

假如原来user表内id为10的行，是一条{id:10，name:"王五"，age:15} 的数据,**事务1**对其进行两次查询，在两次查询的间隔期间，**事务2**对该行记录进行了修改并提交，修改后变为{id:10，name:" 李四"，age:15}，**事务1**对该记录进行第二次查询的时候，发现和第一次查询的结果不一致

**可重复读(Repeatable Read) 及以上隔离级别**可以防止该现象。

1. **虚读(幻读)**:一个事务多次读取一组数据，另一个事务在第一个事务的两次读取操作之间提交了数据（insert），导致一个事务两次查询结果条数不一致。**关注点在于新增和删除**，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

**Eg:**

|  |  |
| --- | --- |
| **事务1** | **事务2** |
| start transaction; |  |
| select count(\*) from user; |  |
|  | start transaction; |
|  | insert into user(id,name,age) values(11, "赵六",18); |
|  | commit; |
| select count(\*) from user; |  |
| commit; |  |

假如原来user表中有10条记录，**事务1**对其进行两次查询，在两次查询的间隔期间，**事务2**新增一条记录并提交，**事务1**对表中记录总条数进行第二次查询的时候，发现和第一次查询的结果不一致

**序列化(serializable)**可以防止该现象。

**请注意：mysql是很难复现幻读的，因为除了加锁，mysql提供了另一种方式来避免幻读，那就是多版本控制(MVCC)，上述案例，事务1第二次查询count(\*)时，发现并不能查到11条，而是10条，和第一次查询保持一致。但如果试图在事务1中插入id=11的记录(在事务2执行插入语句后)，则会提示主键重复。**

#### 2.4.2事务隔离级别

数据库提供了四种事务隔离级别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事务隔离级别 | 丢失更新 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 否 | **是** | **是** | **是** |
| 不可重复读（read-committed） | 否 | 否 | **是** | **是** |
| 可重复读（repeatable-read） | 否 | 否 | 否 | **是** |
| 串行化（serializable） | 否 | 否 | 否 | 否 |

1. **读未提交 Read UnCommitted**：

**实现原理：**通过**一级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)。**

**读取操作：不加任何锁。**

由于读取不加锁，读取操作不会与任何操作(包括修改)发生锁冲突，这可能导致**脏读、不可重复读、幻读**现象

**解决了丢失更新的问题。这可能会造成脏读、不可重复读、幻读现象**

1. **读已提交(sqlServer、Oracle默认级别)** **Read Committed**：

**实现原理：**通过**二级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)**

**读取操作：读取前必须加共享读(S)锁，读完立即释放，不必等到读取事务结束。**

因为X锁会和S锁冲突，执行修改操作的事务结束之前，其他事务不能进行读取，解决了**脏读**。

但由于**共享读(S)锁**读取完立即释放，事务A两次读取数据，加了两次S读锁，锁不连续，在两次加锁之间，事务B加了X写锁进行修改操作，可能导致**不可重复读**、**幻读**现象

**解决了丢失更新、脏读，但不可重复读和幻读仍可发生**

1. **可重复读(MySql默认级别) Repeatable Read**：

**实现原理：**通过三**级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)**

**读取操作：读取前必须加共享读(S)锁，直到事务结束(rollback或commit)。**

因为读取操作加S读锁直到事务结束，因此直到读取事务结束之前，修改事务不能进行修改，解决了**不可重复读。**

但由于读取操作没有对整张表加锁，因此在两次读取之间，虽然可能导致**幻读**现象

**解决了丢失更新、脏读、不可重复读，但幻读仍可发生**

1. **序列化 serializable**：

仅仅通过“行级锁”是无法实现事务序列化的，应该对整张表加锁(锁表)

事务隔离的最高级别，提供严格的事务隔离，它要求事务串行执行，事务只能一个接着一个地执行，但不能并发执行。会引发超时和锁竞争，耗资源多、损失并行度、性能低，一般不用

**解决了所有可能出现的事务并发问题**

**如果仅仅把事务级别设为最高，就会带来并发的性能问题**，数据库的事务隔离级别越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的，同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。因此，我们应当灵活应用事务隔离的特性来平衡＂隔离＂与＂并发＂的矛盾

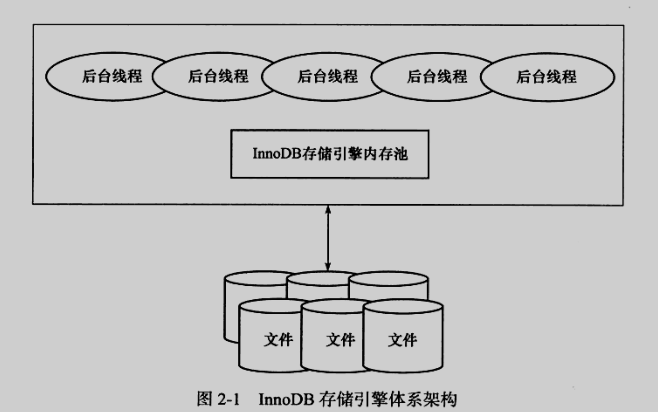
需要注意的是，如果在一次事务中，混合使用了事务型表和非事务型表，在正常commit时不会有什么问题，但如果需要rollback，非事务型表上的操作就难以撤销。因此，在设计表的结构时，使用何种存储引擎显得尤为重要

**请注意，上述实现原理是单纯通过锁机制实现了事务的隔离级别，在实际情况中，mysql还采取了另外一种方法来实现事务隔离**

### 2.5 事务日志

InnoDB的事务日志包括redo log(重做日志)和undo log(回滚日志)。

#### 2.5.1 InnoDB存储引擎事务操作原理



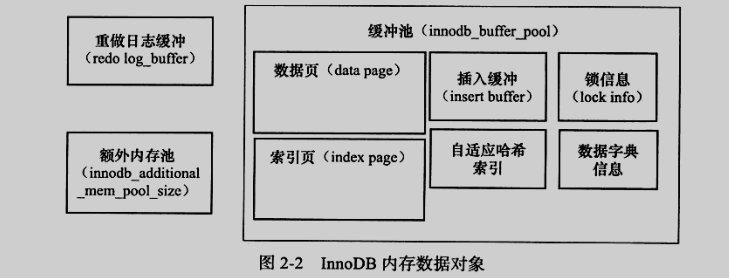
从上图我们可以看出，InnoDB存储引擎的体系架构，可以划分为三层

1. **内存池**
2. **后台线程**
3. **数据文件**

这里我们主讲内存池

##### 2.5.1.1 InnoDB的内存池

InnoDB的内存池有如下几部分构成



为了弥补CPU运算速度和磁盘IO读写之间的巨大鸿沟，InnoDB采用了缓存技术来提高数据库的读写性能。

当事务需要修改某条记录(row)时，InnoDB将该数据所在的页(page)从磁盘的数据文件(data file)读取到缓冲池(buffer pool)中(下一次再读取相同的页(page)时，首先会判断该页(page)是否已经存在于缓冲池(buffer pool)中，如果已存在，缓存命中，否则再从磁盘中读取)，事务提交以后，会修改缓冲池中的页(page)(而不是磁盘中的页(page))，此时缓冲池中的页(page)已经和磁盘中页(page)的不一样了，我们将缓冲池中的页(page)称为脏页(dirty page)，脏页(dirty page)会通过一种名为CheckPoint的机制以一定频率刷新到磁盘中(而不是立即刷新)。

缓冲池中缓存的数据页类型有:索引页、数据页、 undo 页、插入缓冲、自适应哈希索引、 InnoDB 的锁信息、数据字典信息等。索引页和数据页占缓冲池的很大一部分，具体缓冲池各项参数如何配置这里不在展开，只需要知道原理即可

既然InnoDB存在一个缓冲池，那么必然需要一个缓冲池管理算法进行管理，这里是通过LRU(Latest Recent Used最近最少使用算法)管理缓存的。传统的LRU算法是将最频繁使用的页在LRU列表的前面，最少使用的页在LRU列表的尾部，当缓冲池满时，优先释放未于LRU列表尾部的页。而InnoDB则对LRU进行了一定的优化，它提出了midpoint的概念。每次新读取的页(page)，并不是直接放在LRU列表的最前面，而是放在midpoint的位置，midpoint默认为63%处(值可修改)，在InnoDB中，把midpoint前面的部分(默认前63%的部分)称为new list(可以理解为热点数据)，midpoint后的部分(默认后37%的部分)称为old list。

那么这么做有什么好处呢？因为在sql操作中，往往一条sql语句会访问多个页(page)，这些通常只在本次查询中被需要，不属于活跃数据，如果直接放在LRU列表的最前面，一旦页(page)过多很容易将活跃数据从LRU列表中移除，降低缓冲命中率

##### 2.5.1.2 Write Ahead Log策略

如果我们在将脏页刷新到磁盘之前，数据库宕机，那么数据就会丢失，这是我们无法接受的。为了防止宕机导致的数据丢失问题，InnoDB采用了**Write Ahead Log策略**。

在InnoDB中，存在两种日志：重做日志(redo log)和回滚日志(undo log)

1. **redo log：当事务执行commit前，强制先将该事务的所有日志写入到redo日志文件中，用于实例故障恢复**
2. **undo log：记录事务开始前的状态，用于事务失败时的回滚操作和MVCC多版本控制**

那么一次事务的执行过程就应该是：

1. 事务开始
2. 如果需要加锁，就进行加锁操作
3. 判断缓冲池(buffer pool)中是否存在对应的页(page)，如果存在，缓存命中，进行下一步操作，如果不存在，将数据所在的页(page)从磁盘的数据文件(data file)读取到缓冲池(buffer pool)中
4. 记录undo log
5. 记录redo日志到redo log buffer中(何时从redo log buffer写入磁盘中的redo log file由参数innodb\_flush\_log\_at\_trx\_commit决定)
6. 在内存的缓冲池(buffer pool)中修改对应的页(page) (脏页通过CheckPoint机制定期写入磁盘的data file文件)
7. 事务提交commit

**Write Ahead Log策略**要求在提交前，先写redo log，再修改页数据，一旦发生宕机，我们可以通过redo log来完成数据恢复。**在数据库的世界里，数据从来都不重要，日志才是最重要的，有了日志就有了一切。**

**注意：**

1. **在第5步时，日志被记录到redo log buffer后，因为redo log buffer是存在于内存中的，要持久化到磁盘上的话(磁盘写入redo log是顺序读写，速度很快)，中间还要经过操作系统内核空间的os buffer，然后由os buffer的内存持久化到磁盘文件redo log file中。我们可以设置参数innodb\_flush\_log\_at\_trx\_commit的值决定redo log何时被持久化到磁盘文件redo log file中，它可以有3种值：**

**0: 由mysql的main\_thread每秒将存储引擎log buffer中的redo日志写入到log file，同时调用文件系统的sync操作，将日志刷新到磁盘。**

**1：每次事务提交时，将存储引擎log buffer中的redo日志写入到log file，同时调用文件系统的sync操作，将日志刷新到磁盘。**

**2：每次事务提交时，将存储引擎log buffer中的redo日志写入到log file，然后由存储引擎的main\_thread 每秒将日志刷新到磁盘。**

**只有设置innodb\_flush\_log\_at\_trx\_commit=1时，才完全不会因为宕机丢失数据，但同时，这种设置的性能也最差**

1. **第六步修改页(page)后，在适当时间会刷到磁盘的数据文件data file中**

##### 2.5.1.3 CheckPoint技术

当我们修改了记录时，实际上是修改了缓冲池中的页(page)，此时页是脏的，我们需要定时将页刷新到磁盘中，倘若每次发生变化我们都将页刷新到磁盘中，那么势必会产生大量的IO，不同于redo log的写入，数据文件的写入是随机读写，大量IO会极大地降低数据库的性能

假设以上数据库运行了很久，同时假设缓冲池可以缓冲数据库所有数据，并且重做日志可以无限大，此时突然发生了宕机，通过redo log来恢复数据，耗时过长，恢复的代价非常大，我们必须提供另一种机制来保证恢复时间不致过久

CheckPoint技术是为了解决如下问题：

1. 数据库恢复时间过长
2. 因内存大小有限，缓冲池不够用，需要将脏页刷新到磁盘
3. 重做日志已满，无法继续记录

当数据库发生宕机并重启后，只需要对checkpoint之后的redo log进行数据恢复，而checkpoint之前的数据都已经刷回磁盘这样就大大缩短了重做时间

#### 2.5.2 redo log重做日志

redo log通常是物理日志，记录的是数据页的物理修改，而不是某一行或某几行修改成怎样怎样，它被用来实现事务的持久性，redo log由两部分组成：

1. **redo log buffer 重做日志缓冲**
2. **redo log file 重做日志文件**

由于Write Ahead Log策略的存在，事务执行commit前，会强制先将该事务的所有日志写入重做日志缓冲(redo log buffer)中，接下来，根据参数innodb\_flush\_log\_at\_trx\_commit的值决定 缓冲日志(redo log buffer)持久化到磁盘重做日志文件(redo log file)的策略，该参数有三个值：

**0: 由mysql的main\_thread每秒将存储引擎log buffer中的redo日志写入到log file，同时调用操作系统中文件的fsync操作，将日志刷新到磁盘。**

**1：默认值，每次事务提交时，将存储引擎log buffer中的redo日志写入到log file，同时调用操作系统中文件的fsync操作，将日志刷新到磁盘。**

**2：每次事务提交时，将存储引擎log buffer中的redo日志写入到log file，然后由存储引擎的main\_thread 每秒将日志刷新到磁盘。**

将redo日志写入磁盘的file文件中，需要先写入磁盘file文件的文件系统缓存中，若想真正持久化到磁盘文件，还需要调用一次fsync操作将缓存刷入磁盘。虽然可以将参数innodb\_flush\_log\_at\_trx\_commit设为0或者2来提高事务执行效率，但这样做会丢失事务的持久化特性

#### 2.5.3 undo log回滚日志

undo log主要有两个作用：

1. **实现事务回滚**
2. **实现MVCC**

很多人都有这样的误解undo log会将数据库物理地恢复到事务开始之前的状态，实际上并不是这样的，因为当多个事务同时执行时，如果是物理地恢复，极有可能会影响其他事务的工作。Undo log不同于redo log，它是逻辑日志，比如当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。当某个事务进行回滚时，它所做的操作都会被逻辑地取消，逻辑取消可以通过事务隔离进行分隔，保证不会影响到其他事务的操作

除了回滚操作，undo log还可以被用来实现多版本并发控制(MVCC)

**最后，需要注意，由于undo log也需要持久性保护，所以undo log的记录过程也会产生redo log**

### 2.6 多版本并发控制(MVCC)

#### 2.6.1 MVCC概述

数据库实现事务隔离的方式，基本可以分为以下两种。

**1.在读取数据前，对其加锁，阻止其他事务对数据进行修改。**

**2.通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot），并用这个快照来提供一定级别（语句级或事务级）的一致性读取。**从用户的角度，好像是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据**多版本并发控制（ＭultiVersion Concurrency Control，简称MVCC或MCC）**，也经常称为多版本数据库。

MVCC在大多数情况下代替了行锁，**实现了对读的非阻塞，读不加锁，读写不冲突。**缺点是每行记录都需要额外的存储空间，需要做更多的行维护和检查工作

出于性能上的考虑，MySql的大多数事务型存储引擎实现的都不是简单地通过锁来实现事务隔离的，它们一般同时实现了多版本的并发控制(MVCC)。当然，不同的存储引擎的实现原理也不同，这里我们以InnoDB为例

#### 2.6.2 MVCC原理

##### 2.6.2.1 InnoDB的行结构

InnoDB中的数据行，除了我们用于存储数据的data部分，还隐藏了4个字段，他们分别是：

1. **db\_row\_id：**行ID，用来生成默认聚簇索引（聚簇索引，保存的数据在物理磁盘中按顺序保存，这样相关数据保存在一起，提高查询速度）
2. **db\_trx\_id：**表示最近修改该行数据的事务ID
3. **db\_roll\_ptr：**表示指向该行回滚段的指针。该行数据上的所有旧的版本，在undo中都通过链表的形式组织，即**事务链表**。而db\_roll\_ptr的值，指向undo中该数据行的历史记录链表
4. **deleted\_bit：**删除标记位，删除时设置



##### 2.6.2.2 MVCC的流程



#### 2.6.3 InnoDB中的MVCC

InnoDB中的mvcc不是表准MVCC，标准MVCC难以实现，InnoDB的MVCC结合了排他锁，它只在**Repeatable Read(可重复读)和Read Committed(读已提交)** 两个级别下工作，**Read UnCommitted(读未提交)总是读取最新的行，为了防止出现幻读现象，InnoDB采用了间隙锁策略，有效地防止了幻影行的插入**

# 三、性能优化

数据库设计上的优化

## 1.数据类型优化

MySQL支持多种数据类型，对于一个字段的数据类型，我们通常可以有多种选择，但不同的选择，往往导致不同的性能。不管存储哪种类型的数据，应当遵循以下几个原则。

### 数据类型选择原则

1. **尽量使用可正常存储数据的最小类型**

更小的数据类型意味着占用更少的磁盘，内存和缓存，对其进行处理的耗时也更短，如：tiny int unsigned比int更快

1. **使用简单数据类型**

简单数据类型所消耗的CPU周期明显小于复杂数据类型

如： int类型比varchar类型操作要快

使用datetime存储日期而不是字符串类型

使用整型存储ip地址

1. **尽量指定列为not null**

当列值可以为null时，MySQL将很难进行查询优化，因为null值将会使索引，索引统计和值比较都更为复杂。并且可为null的列将占用更多的存储空间

一般来说，将可为null的列改为not null所带来的性能提升较小，因此没必要将每列都设为not null，但如果计划在某列上建索引，就应当尽量设计为not null

在选择列的数据类型时，首先应选择范围比较宽广的大类型，比如：数字、字符串、时间等，接下来再选择具体的数据类型

Eg：创建时间，我们可以先确定是时间类型，然后再决定是datetime还是timestamp

接下来我们就将根据各个大类型来分析各具体数据类型的优劣

### 1.1数字类型

数字分为两种：整数和实数(小数)

#### 1.1.1.整数类型

存储整数可以使用如下几种类型：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **含义** | **有符号范围** | **无符号范围** | **占用存储空间** |
| tinyint | 最小的整数 | -128到127 | 0到255 | 1字节 |
| smallint | 小整数 | -32768到32767(万级) | 0到65535 | 2字节 |
| mediumint | 中等大小的整数 | -8388608到8388607(百万级) | 0到16777215 | 3字节 |
| int | 整数 | -2147483648到2147483647(十亿级) | 0到4294967295 | 4字节 |
| bigint | 大整数 | -9223372036854775808到9223372036854775807 | 0到18446744073709551615 | 8字节 |

注意：mysql允许为整数类型指定宽度，如int(11)，这对数据库来说没有实际意义，仅会影响mysql图形化客户端显示数据的长度，因此int(1)===int(11)===int(20)

#### 1.1.2.实数类型

实数类型往往被用于存储小数，但这也不是绝对的，有时也会使用decimal来存储比bigint还大的数据

浮点类型(float和double)在存储相同范围的数值时，往往占用比decimal更少的空间，通常为了减少计算的开销和避免浮点存储计算不精确，可以使用bigint来代替decimal，比如计算金额时，如果希望精确到万分之一，可以将所有金额都乘以一万来存储

### 1.2.字符串类型

Mysql的字符串类型主要有varchar和char类型，由于字符拥有多种编码方式，下面将先介绍不同的编码:

由于长度限制，varchar类型的列，最多存储长度不能超过65535个字节，因此

**字符类型若为gbk，每个字符最多占2个字节，最大长度不能超过32766个字符。**

**字符类型若为utf8，每个字符最多占3个字节，最大长度不能超过21845个字符。**

#### 1.2.1 varchar类型

varchar类型用于存储变长字符串，因此varchar类型通常比char类型更节省存储空间。为了存储列的长度，varchar需要多占用1~2个字节，varchar长度小于等于255的，需要1个额外字节，varchar长度大于255的，需要两个额外字节，因此，为了占用更少的空间，如果非必须，varchar长度最好不要大于255

但是在进行UPDATE操作时，被修改行的varchar类型的列中存储的数据可能会变长，比如5个字符变为10个字符，这会导致页内没有更多的存储空间来存储修改后变长的数据，这就需要做一些额外的处理，MyISAM会将行拆成不同片段进行存储，而InnoDB采用分裂页的方式使得变长后的数据可以放入页内

在以下情况中使用varchar是合适的：

1. **列的数据结尾可能出现空格的情况，比如”string ”，char类型保存时会在右边填充空格以达到指定的长度，因此，char类型在存储”string ”时最终只能读到”string”**
2. **对于UPDATE操作较少的列，不需要担心存储空间的碎片问题(比如varchar(20)本来存储了18个字符，update后，字符串被改成10个字符，这样就产生了8个字符的碎片)，这种情况下使用varchar更为合适，因为varchar不能很好地处理存储碎片的问题**

**需要注意的是：初学者可能会认为varchar(20)和varchar(200)完全相等，这种想法是错误的。诚然，varchar(20)和varchar(200)占用的磁盘空间大小一致，但二者对于内存的消耗是不同的，比如存入15个字符，二者都只占用16个字节，但是varchar(20)和varchar(200)分别占用21、201个，过多地占用内存会对排序或临时表等操作产生不利影响**

#### 1.2.2 char类型

Char类型用于存储定长字符串，取值可以在0～255 个字符之间波动，char类型在保存时，会在字符串的右边填充空格以达到指定长度，当检索/读取字符串时，右侧的空格又会被删除

在以下情况中使用char是合适的：

1. **长度固定的列。char的效率比varchar稍高，但通常来说，char往往会占用更多的空间，但是对于定长的列，我们就没有这么多忧虑了(对于定长列来说，varchar占用空间等同于char，比如手机号)**
2. **对于非常短的列，比如char(1)，如果采用utf8，只需要3个字节，而varchar(1)则需要4个字节**
3. **常被修改的列(每次修改后的列字符串长度都有变化)。varchar虽然比char节省空间，但是如果一个varchar列经常被修改，而且每次被修改的数据的长度不同，这会引起”行迁移”(Row Migration)现象，而这造成多余的I/O，是数据库设计和调整中要尽力避免的，在这种情况下用char代替varchar会更好一些。**

### 1.3.枚举类型

在一些场景下，可以使用枚举列来替代字符串类型，MySQL在存储枚举类型时非常紧凑，会根据枚举中值的数量将存储所需空间压缩到1~2个字节中。MySQL在内部会将每个枚举值在枚举列表中的位置保存为整数，eg：

创建表：column ENUM(”apple”,”orange”,”dog”)，新增记录column为dog，实际存储的是数字3

但是枚举也有缺点：**枚举列表固定，拓展性差，想要添加或删除枚举必须修改表结构(这对数据量大的表来说是种非常昂贵的操作)，因此，对于枚举类型未来可能会有改变的列，应尽量避免使用枚举**

### 1.4.日期类型

MySQL提供两种常用的日期类型，dateTime和timeStamp

#### 1.4.1 dateTime类型

DateTime类型可以保存大范围的日期，范围从1001年到9999年，精度为秒，与时区无关，将日期存储到YYYYMMDDHHMMSS格式的数字中，使用8个字节

#### 1.4.1 timeStamp类型

时间戳类型保存了从1970年1月1日以来的秒数，只使用4个存储空间，因此它的存储范围比dateTime小得多

通常，我们应该尽量使用timeStamp来代替dataTime，因为timeStamp拥有更高的空间效率。有时候，我们需要存储比秒更精确的时间，但MySQL目前并不支持，但是我们可以通过别的方式来解决，比如使用BigInt类型

### 1.5.主键id优化原则

为主键id选择一个合适的数据类型尤为关键，下面有几条合适的建议：

1. 保证主键id和关联表中的外键类型一致。主键id往往会作为另一张表的外键，因此二者的类型之间需要精确匹配，包括unsigned属性，否则可能导致性能问题甚至隐藏bug
2. 整数类型通常是id列的最好类型选择，尽量避免使用字符串类型作为主键id，字符串类型既消耗空间性能又差，MyISAM对字符串使用压缩索引，性能上会有好几倍的降低，并且由于字符串类型id往往通过随机算法产生，其插入时会随机读写，导致Insert和select操作性能降低(insert插入的值会随机写入索引的不同位置，造成磁盘随机写入，select时，逻辑上紧邻的记录在物理磁盘上可能分布在不同的地方，磁盘随机访问)

## 2.表结构设计优化原则

### 2.1不要设计过多的列

MySQL在服务层和存储引擎层是通过行缓冲格式来拷贝数据，Mysql会在服务层将行缓冲格式解码成各列，如果列过多，这种转换的代价就会很高

### 2.2避免过度使用枚举

Eg：

country enum(‘1’,’2’,’3’…..’40’)

枚举列表固定，拓展性差，想要添加或删除枚举必须修改表结构(这对数据量大的表来说是种非常昂贵的操作)，因此，对于枚举类型未来可能会有改变的列，应尽量避免使用枚举，上例中，如果希望添加第41个国家，则需要alter table，这是难以接受的

### 2.3在范式和反范式之间取得平衡点

首先回顾一下范式的定义，范式分为6种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式），但我们常用的是**三大范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）**

**(1)第一范式(1NF)：要求数据库的每一列都是不可分隔的数据项，即列应具有原子性**

eg：

下表中**时间**字段可分为上课时间和下课时间，因此不符合第一范式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程号 | 课程名 | 教职工编号 | 时间 | 学分 |
| 001 | 离散数学 | 001 | 09:35:00,11:00:00 | 2 |
| 002 | 编译原理 | 002 | 09:35:00,12:00:00 | 2 |
| 003 | 声乐艺术 | 003 | 07:35:00,09:00:00 | 1 |

**(2) 第二范式(2NF)：在第一范式的基础上，要求数据库的每列都完全依赖于全部主键(部分依赖不符合第二范式)**

**部分依赖：**当主键由两个或两个以上字段构成，而表中的某些信息通过主键的一个字段就能唯一确定，我们称这样的依赖关系为部分依赖

eg：

下表中红色部分构成主键，**性别**字段通过**教职工编号**就能确定，**课程名**字段和**学分**字段通过**课程号**就能确定，不依赖于全部主键，不符合第二范式，可以通过拆表的方式解决问题，下表可以拆成课程表，教职工表，课程-教职工关系表

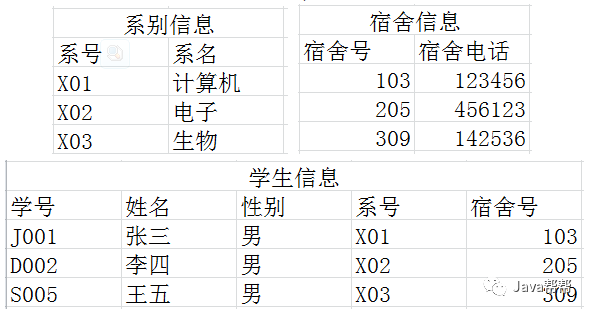
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程号 | 教职工编号 | 性别 | 课程名 | 学分 |
| 001 | 001 | 男 | 离散数学 | 2 |
| 002 | 002 | 男 | 编译原理 | 2 |
| 003 | 003 | 女 | 声乐艺术 | 1 |

**(3)第三范式(3NF)：在第二范式的基础上，要求数据库的每列都必须直接依赖主键，不能存在间接依赖的情况。**

下表中，主键是学号，姓名、性别、系号、宿舍号都是直接依赖主键学号的，而系名则是直接依赖系号，宿舍电话直接依赖宿舍号



可以通过拆表来解决以上问题：



**(4)反范式化**

范式化最大的优点在于可以大幅减少冗余，但是其缺点也很明显——查询时需要关联大量的表，这在表中数据量较大时，性能表现极差，可以适当增加冗余字段，虽然破坏了范式约束，但是提高了查询的效率，在实际操作中，范式需要与反范式结合使用

## 3.加快alter table的速度

在表中数据非常大时，alter table所付出的代价也是非常大的，这通常是由一下原因引起的：

**1. mysql中alter表的机制是：创建一个新的空表，然后把数据全部插入到新表中，然后删除老表。**

**2. 在数据迁移到新表时，每插入一行，需要对索引进行一次更新。效率低下且会产生索引碎片**

**3. 当没有足够的内存，或者表上的索引非常多时，效率十分的低下。**

下面将提供一些小建议来加快alter table的速度

1. **如果仅仅是修改表中某列的默认值的话，可以使用**

**alter table 表名 alter column 字段 set default 值;**

来代替

**alter table 表名 modify column 字段 set default 值;**

1. **在大批量导入数据到新表前，先禁用新表的索引，导入完毕后，再启用索引**

mysql>**ALTER TABLE table\_2 DISABLE KEYS;**

Query OK, 0 rows affected (0.00 sec)

mysql>**INSERT INTO table\_2 SELECT \* FROM table\_1;**

Query OK, 2000000 row affected (36.30 sec)

mysql>**ALTER TABLE table\_2 ENABLE KEYS;**

Query OK, 0 rows affected (44.55 sec)

**当然，在alter table操作之前千万记得进行备份！**

## 4. 索引

说到性能优化，不得不提到索引，它是我们实现数据库查询性能优化的关键点之一，尤其在数据量越来越大的时候，索引显得更加重要，下面我们就来详细了解一下索引

### 4.1 什么是索引

索引是一种可以帮助mysql快速查询数据的数据结构，他可以大幅提高mysql的检索速度。如果不使用索引，MySQL必须从第一条记录开始读完整个表(全表扫描)，直到找出相关的行，表越大，查询数据所花费的时间就越多。如果表中查询的列有索引，MySQL就能够快速到达一个位置去搜索数据文件，而不必查看所有数据，那么将会节省很大一部分时间。

没有添加索引的查询是线性的，而添加了索引，数据查询将会通过二分查询进行：

假设现有一张一百万条记录的表

设每条数据的长度：r = (4 + 44 + 2) = 50字节

使用MyISAM引擎存储，默认磁盘数据块大小：b = 1024字节

表的块因子：bfr = b / r = 1024 / 50 ≈ 20 （条数据/每磁盘数据块）

保存这张表所需的磁盘块为：n = s / bfr = 1,000,000 / 20 ≈ 50,000 块

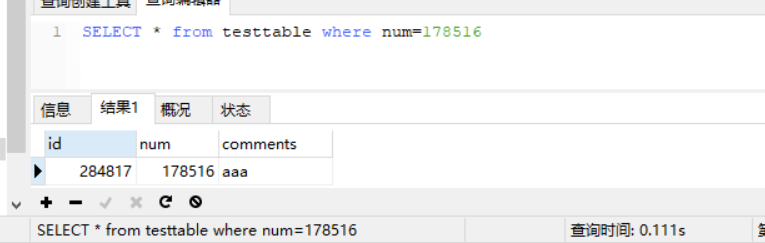
查询速度：

（1）假如id字段上面没有主键（id字段上面也就没有索引了，是无序的），则通过：select id from t4 where id = ?，查询数据时，是线性搜索，平均所需时间为：n / 2 = 25,000，即平均需要搜素25000块磁盘块才能查找到所需数据

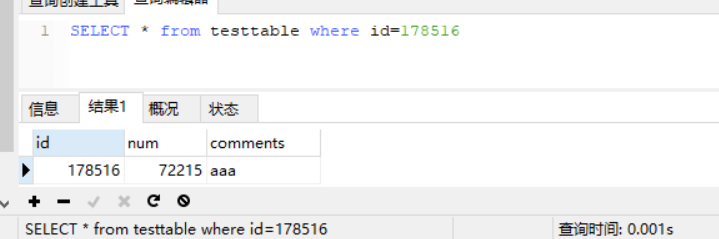
（2）既然id字段有主键（就是包含索引，是有序的），通过一个二分查询，所需平均时间为：log2 n = log2 (50,000) = 15.6，即平均需要搜素15.6块磁盘块就能查找到所需数据

所以，根据以上2个数据看来，有索引和没索引，检索速度差距是巨大的

实测：数据量单表35W条数据：



在没有索引的情况下，对于35W条数据全表扫描，耗时达到0.111s



对于添加了主键索引的字段进行查询，其查询耗时小于0.001s

**优点：索引可以大幅加快sql语句的查询速率**

**缺点：**

**1、创建索引和维护索引要耗费时间，并且随着数据量的增加所耗费的时间也会增加**

**2、索引也需要占空间，我们知道数据表中的数据也会有最大上线设置的，如果我们有大量的索引，索引文件可能会比数据文件更快达到上限值**

**3、当对表中的数据进行增加、删除、修改时，索引也需要动态的维护，降低了数据的增删改操作速度。**

索引不是越多越好，我们的使用原则是：

**1、对经常更新的表就避免对其创建过多的索引，对经常用于查询的字段应该创建索引**

**2、数据量小的表最好不要使用索引，因为由于数据较少，可能查询全部数据花费的时间比遍历索引的时间还要短，索引就可能不会产生优化效果。**

**3、在一个字段上不同值少的列上不要建立索引，比如在学生表的"性别"字段上只有男，女两个不同值，不适合索引。相反的，在一个字段上不同值较多可以建立索引。**

**4、频繁作为查询条件的字段可以考虑加上索引**

**5、查询中常用于排序的字段，排序的字段若通过索引去访问将大大提高排序速度，统计或者分组字段也是如此**

**6、避免对可能含有null值的列建立索引，如果建立了，应避免使用is null和is not null防止拖累查询速度**

### 4.2索引的分类(按结构)

MySQL目前主要有以下两种常用的索引方法：B+Tree，Hash。

#### 4.2.1 B+tree索引

B-tree是mysql最常用的索引，MyISAM和Innodb都支持该种索引，人们在谈论索引的时候如果没有特别说明，一般指的是B+Tree索引

**B+Tree适合用来查找某一范围内的数据，而且可以直接支持数据排序（ORDER BY）**

##### 4.2.1.1 为什么选用B+树作为索引的数据结构

所有索引原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件。那么我们可以尝试将数据分段进行查找，假设我们将1000条数据分成10段，如果我们希望查找第220条数据，我们只需要从第3段开始查找即可，这样一下就少了90%的无效数据，相比于全表扫描，查询量大大减少。

###### 1.为什么不选二叉查找树

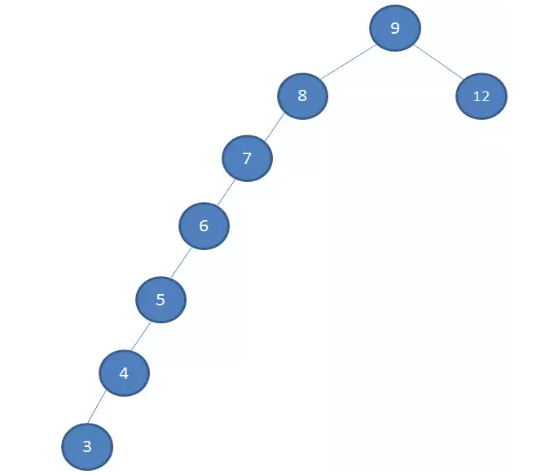
通过上述描述，我们首先想到的数据结构是**二叉查找树**，二叉查找树具有以下特征：

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉查找树

二叉查找树拥有理想的查找效率，其查找的平均时间复杂度为log2(n)，但是二叉查找树在极端情况下可能会退化成链表，如下图所示：



在最差情况下，二叉查找树的时间复杂度会退化到O(n)，显然，二叉搜索树并不适合作为索引

###### 2.为什么不选二叉平衡树

那么如何解决这个问题呢？由此我们想到了**二叉平衡树（AVL树）**，二叉平衡树具有以下特征：

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3. 树的左右两边的层级数相差不会大于1

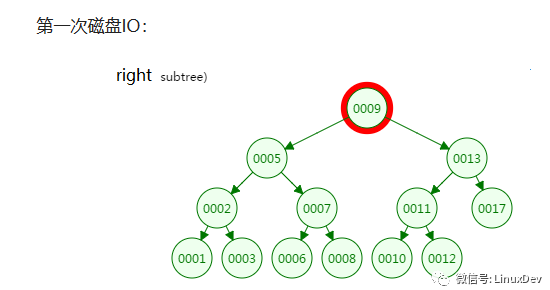
4. 左、右子树也分别为二叉平衡树

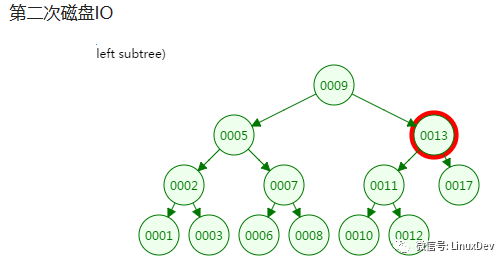
相比于二叉查找树，二叉平衡树(AVL)在左右子树的高度上做了限制，保证二者高度不会相差过大，这会使得二叉平衡树(AVL)的插入和删除操作变得更加复杂(新插入、删除节点都需要重新平衡二叉树)，但会将查询的最好情况和最差情况的时间复杂度都维持在log2(n)

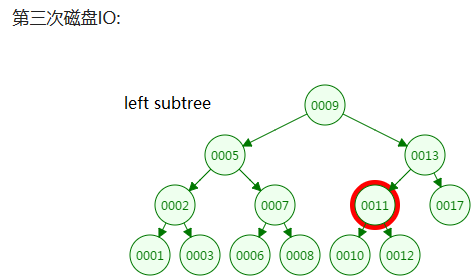
既然二叉搜索树的时间复杂度在最好和最差的情况下都能维持在log2(n)，那为什么二叉搜索树没有被作为索引的数据结构使用呢？

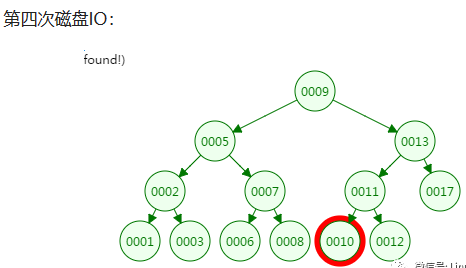
我们都知道，数据库索引是存储在硬盘中的，并且索引的大小是随着数据量增长而逐渐变大的，为了使用索引，我们必须进行磁盘IO，而磁盘IO是非常昂贵的操作，一次IO所花费的成本是一次内存访问的十万倍左右，B-Tree和二叉平衡树的磁盘访问次数是完全不同的。让我们看看二叉平衡树的一次查找需要多少次IO

假设我们需要查找值为10的节点：









我们可以看到，二叉平衡树的IO次数为4次，其最差查找次数是由二叉平衡树的高度决定的，为了减少IO次数，我们需要将原本”瘦高”的二叉树变得”矮胖”。

并且我们所说的平衡二叉树结构，指的是逻辑结构上的平衡二叉树，一个节点仅对应一个数据，而在逻辑结构上相近的节点在物理结构上可能会差很远。因此，每次读取的磁盘页的数据中有许多是用不上的。导致查找过程中要进行许多次的磁盘IO操作。

###### 3.磁盘上的数据结构

一般来说，虽然由于维护完全平衡的二叉树所需付出的代价可能非常大，限制了二叉平衡树的使用，但局部平衡的红黑树解决了这个问题，但是这里并不讨论红黑树的相关理论。

二叉平衡树在算法层级已经相当快了，但为何我们还是不使用其作为数据库的索引存储结构呢？这就需要了解一下二叉平衡树在实际情况中的效率

我们都知道，一台计算机一般有两大存储设备，内存和磁盘。而内存的存储效率远远高于磁盘，为了尽量提升数据的IO效率，磁盘在一次读取时，将读取物理地址上连续的多个数据，这被称为“预读”。

因为数据库索引是以树结构存储在磁盘上的，当数据库的数据总量较大时，其索引文件的大小也可能会随之增大，达到几个G甚至更多，而内存的大小是有限的，这样一来，我们就很难一次将整个数据库索引树全部加载到内存中去。因此，磁盘上的索引数据结构(B树系列的数据结构)将所需的页面从磁盘复制到内存，再将修改后的页面写回磁盘，任何时刻，内存中都只保存部分磁盘页面。并且，一个B树节点通常和一个完整磁盘页差不多大

###### 4.为什么不选B-树

一个m阶的B树具有如下几个特征：

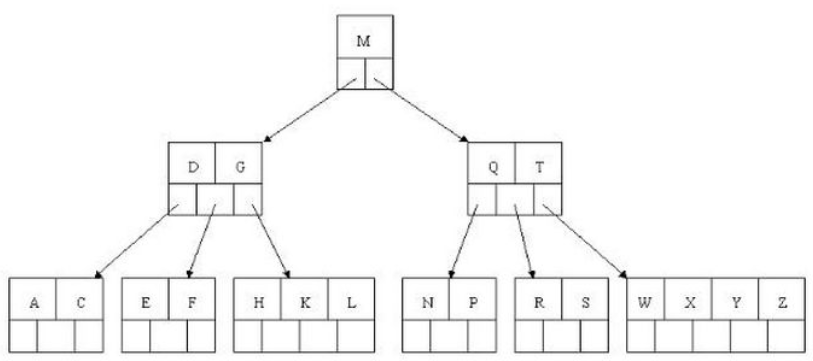
**1.根结点至少有两个子女。**

**2.每个中间结点都包含k-1个元素和k个孩子，其中 m/2 <= k <= m（中间结点是指除了根结点和叶子结点以外的结点，这句话的意思是孩子的数量必然比其父结点拥有的元素数多一个，因为B-树需要用k-1个父结点的元素作为键，来将K个孩子结点分开）**

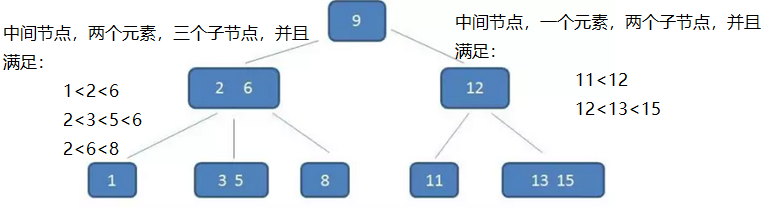
**3.每一个叶子结点都包含k-1个元素，其中 m/2 <= k <= m**

**4.所有的叶子结点都位于同一层。**

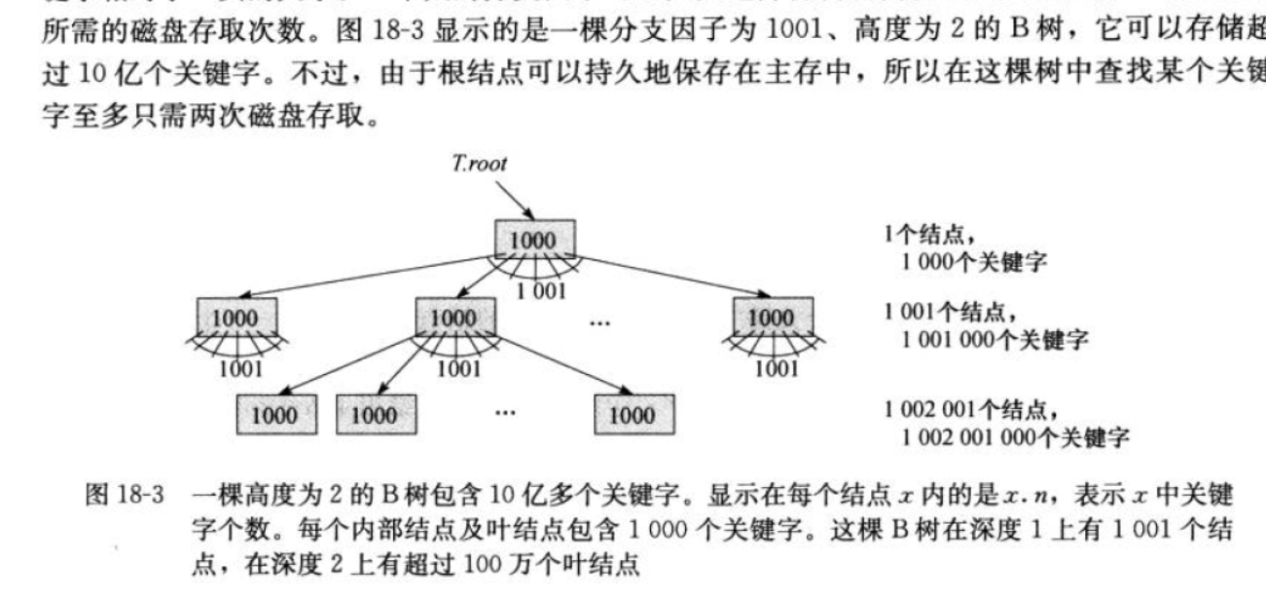
**5.每个结点中的元素从小到大排列，中间结点的k-1个元素正好是k个孩子包含的元素的值域分划。**



**如下图：**



通过上一节的分析我们可以知道，相比于二叉平衡（AVL）树，B-树的一个节点可以存储更多的数据，因此B-树显得更加”矮胖”，其磁盘定位次数更少，我们每次磁盘定位都读取一整个结点的数据，然后对其进行寻道、判断



**注意B-树就是B树，-只是一个符号**

既然B-树拥有这么多的优点，那为什么我们选用B+树作为MySql的索引而不是B-树呢？

###### 5.为什么选B+树

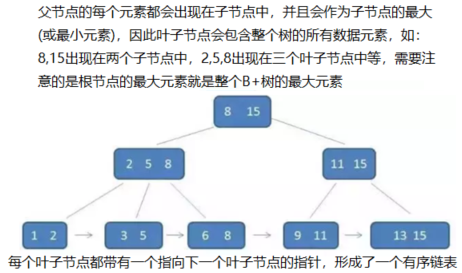
首先让我们来看看什么是B+树。B+树是B-树的一种变体，其拥有比B-树更高的查询性能。B+树的定义如下：

**1.有k个子树的中间节点包含有k个元素（B-树中是k-1个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。**

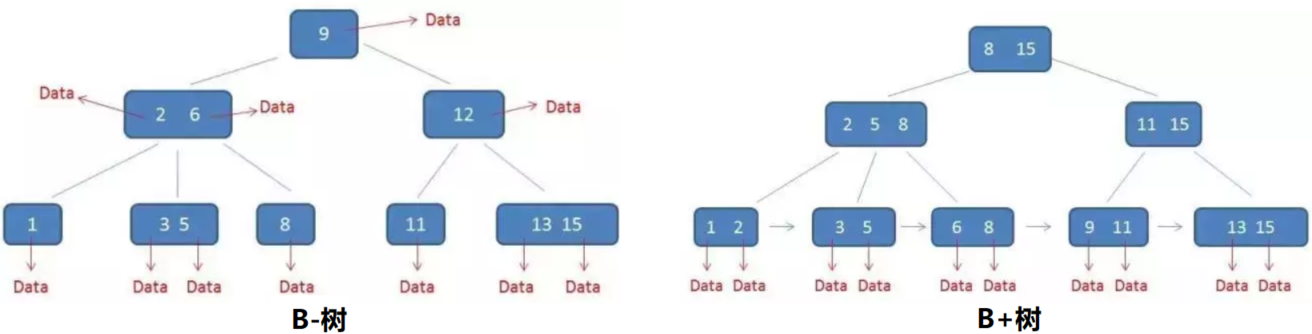
**2.所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小是按照自小而大顺序链接的。**

**3.所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。**

比如：



需要注意的是，与B-树不同，B+树的所有数据都存储在叶子节点中，B+树中间节点的数据仅用来帮助查询



那么相比于B-树，B+树到底有什么好处呢？

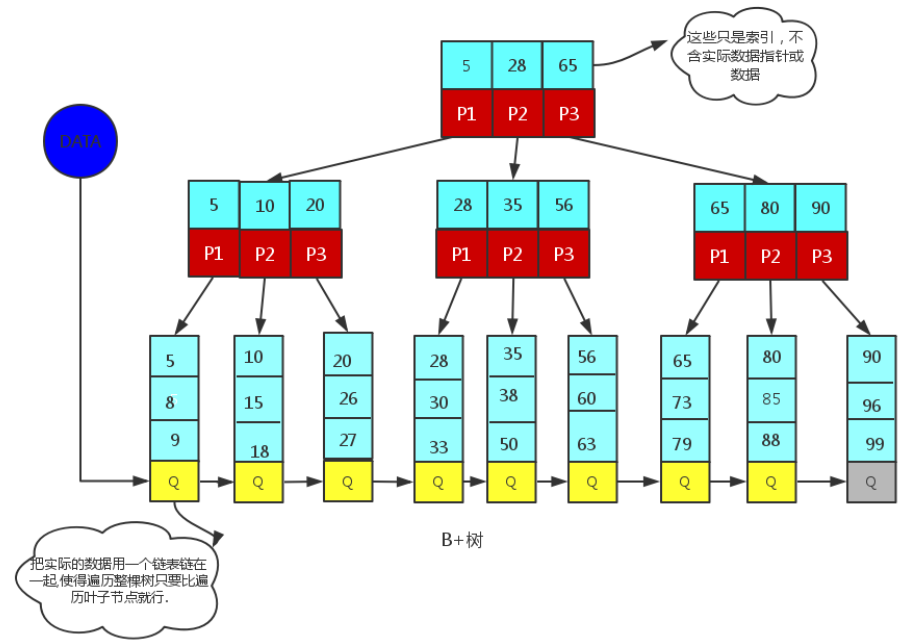
1. B+树的中间节点和根节点不含有卫星数据Data，因此，同样大小的磁盘页(B系列树的一个节点就是一个磁盘页，参见第3小节)可以容纳更多的元素，因此，相比于B-树，B+树更加矮胖，因此查询的IO页更少
2. B-树的查询相较于B+树更不稳定，B-树最好的情况是查到根节点，最糟糕的情况是查到叶子节点，而B+树必然会查询到叶子节点
3. B+树由于所有卫星存储在叶子节点中，并且所有的叶子节点之间还通过链表的方式进行连接，而B-树只能通过先(中、后)序遍历等方式遍历卫星数据，因此遍历B+树比B-树方便多了

##### 4.2.1.2 B+tree索引的原理

B+Tree在MyISAM里的实现和Innodb稍有不同：

MyISAM表数据文件和索引文件是分离的，索引文件仅保存数据记录的磁盘地址  
InnoDB表数据文件本身就是主索引，叶节点data域保存了完整的数据记录

下面是一棵B+Tree



#### 12.1.2.2 Hash索引

哈希索引检索效率非常高，索引的检索可以一次定位，不像B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引。

只有Memory引擎支持显式的Hash索引，但是它的Hash是non unique的，冲突太多时也会影响查找性能。Memory引擎默认的索引类型即是Hash索引，虽然它也支持B-Tree索引

但之所以B-Tree更常用，是因为Hash索引有诸多的缺陷：

**1.Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。**

由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。

**2.Hash 索引无法被用来避免数据的排序操作。**

由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；

**3.Hash 索引不能利用组合索引的一部分索引键进行查询。**

对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

**4.Hash 索引在任何时候都不能避免表扫描。**

前面已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

**5.Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。**

对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下

### 12.1.3索引的分类(按使用方式)

Mysql常见索引有：主键索引、唯一索引、普通索引、全文索引、组合索引

**普通索引**：最基本的索引，没有任何限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

**唯一索引**：与"普通索引"类似，不同的就是：索引列的值必须唯一，但允许有空值。

**主键索引**：它 是一种特殊的唯一索引，不允许有空值。

**全文索引**：仅可用于 MyISAM 表，只能在CHAR,VARCHAR,TEXT类型字段上使用全文索引，针对较大的数据，生成全文索引很耗时耗空间。

**组合索引**：为了更多的提高mysql效率可建立组合索引，遵循”最左前缀”原则。创建复合索引时应该将最常用（频率）作限制条件的列放在最左边，依次递减。