**SQL笔记**

1. SELECT语句的执行顺序 ★★★★★★★★★★★★★★★★★★★★ 5

2. 小知识 # 学生选课的表位置 6

3. 数值型运算 7

3.1 取模（%） 7

3.2 加法（+） 7

3.3 减法（-） 7

3.4 除法（\） 7

3.5 乘法（\*） 7

4. 比较运算 8

 Null与Unknow的区别： 8

 IS NULL与 = 的区别： 8

5. 字符匹配与连接 9

5.1 字符匹配(like) 9

5.2 连接(||) 9

6. 逻辑运算 10

6.1 AND 10

6.2 OR 10

6.3 NOT 10

7. 集合运算(多个查询语句的集合运算) 11

7.1 UNION (并运算) 11

7.2 INTERSECT (交运算) 11

7.3 MINUS(差运算) 11

7.4 (NOT) IN (确定集合) 11

7.5 (NOT) BETWEEN AND (确定范围) 11

8. 函数 13

8.1 汇总函数 (聚集函数) 13

 COUNT (计数) 13

 SUM (求和) 只针对数值类型的列 13

 AVG (平均值) 只针对数值类型的列 13

 MAX (最大值) 13

 MIN (最小值) 13

 ARIANCE (方差ariance) 只针对数值类型的列 14

 STDDEV (标准差stddev) 只针对数值类型的列 14

8.2 日期/时间函数 14

 ADD\_MONTHS(x,y) # 增加或减少指定个月 14

 LAST\_DAY(time) # 返回指定日期所在月份的最后一天 15

 MONTHS\_BETWEEN(x,y) # 返回两个日期之间的月份数 15

 NEW\_TIME(date,x,y) # 时区变换 15

 NEXT\_DAY(date,weekday) # 未来最近的指定星期对应的日期 15

 SYSDATE # 系统日期 16

 TO\_CHAR(date,x) # x 表示要转换成的格式 16

 TO\_DATE(date,x) # x 表示转换的标准日期格式。 16

8.3 数学函数 16

 ABS(x) (绝对值) 16

 CEIL(x) (向上取整) 16

 FLOOR(x) (向下取整) 17

 ROUND(x,[y]) (四舍五入) # y 表示保留位数 17

 MOD(x,y) (取模) 17

 POWER(x,y) # (x^y) 17

 EXP (x) # (e^x) 17

 LN(N)、LOG(a,N) 17

 SQRT(x) 求根 18

 SIGN(x) 符号函数 18

 TRUNC(x,[y]) 截取函数 18

 三角函数 19

8.4 字符函数 19

 CHR(x) # 将ASCII码转换为对应字符 19

 ASCII(x) # 将字符转换为ASCII码 19

 CONCAT(str1,str2) # 只能连接两个字符串 19

 INITCAP(str) <initcap> # 首字母大写，其余字母小写 20

 LOWER(str)、UPPER(str) # 全部小写and全部大写 20

 LPAD(str1,num,[str2])、RPAD(str1,num,[str2]) # 左填充/截取；右填充/截取 20

 LTRIM(str1,[ str2])、RTRIM(str1,[ str2]) # 从左/右切除(删除)指定字符，默认空格 20

 REPLACE(str1,str2[,str3]) # 替换字符串 21

 TRANSLATE(str1,str2,[str3]) # 替换字符(注：与REPLACE不同) 21

 SUBSTR(str1,num1,num2) # 截取（与Excel中mid()相似） 22

 INSTR(str1, str2,[num1],[num2]) # 字符查找（与Excel中search类似） 22

 LENGTH(str1) # 字符串长度 22

 LENGTHB(str1) # 字符串字节长度 23

 TO\_CHAR() # 将date/number转换为char 23

 TO\_NUMBER() # 将char转换为number 23

 TO\_DATE() # 将number/char转换为date 23

9. SQL字句 24

9.1 WHERE 24

 STARTING WITH字句 # 与like(str%)效果相同 24

9.2 GROUP BY （分组） 24

 单列表组查询 24

 多列分组查询 25

 使用HAVING限制返回组 25

9.3 ORDER BY （排序） 25

 多列排序 25

 按表达式排序 25

 使用别名排序 26

 用列位置编号排序 26

 自定义排序 26

10. 多表查询 27

10.1 连接(交叉连接) 27

10.2 内连接(INNER JOIN…ON…) 27

 等值连接 27

 非等值连接 27

10.3 自身连接(SELF JOIN) 27

10.4 自然连接(NATURAL JOIN…ON…) 28

10.5 外连接(OUTER JOIN) 28

 左外连接(LEFT JOIN…ON…) 28

 右外连接（RIGHT JOIN…ON） 29

 完全外连接(FULL JOIN…ON) 29

11. 嵌套查询(子查询) 31

11.1 带有IN谓词的子查询 32

11.2 带有ANY或ALL谓词的子查询 32

 ANY 32

 ALL 33

11.3 带有EXISTS谓词的子查询（可使用NOT EXISTS） 33

12. 数据操纵语言（DML） 34

12.1 插入数据（增：INSERT INTO … VALUES…） 34

 单条元组插入 34

 利用子查询插入 # 多值插入/表的复制 34

 多表插入 35

12.2 数据修改（改：UPDATE … SET … WHERE…） 36

 单条元组修改 36

 多条元组修改 36

 子查询修改 36

12.3 数据删除（删：DELETE FROM … WHERE…） 37

13. 数据定义语言（DDL） 38

13.1 定义基本表（create table） 38

 常见表参数 39

 利用子查询创建表 40

13.2 修改基本表（alter table） 40

 **添加列**  （ADD <列名> <类型> [约束]） 41

 **修改列类型**  （modify <列名> <类型>） 41

 **设置unused不可用状态** （set unused column <列名>） 41

 **删除不可用列** （drop unused culumns） 41

 **删除列** （drop column <列名>） 41

 **修改表名** （rename to <新表名>） 42

13.3 修改约束 （alter table <表名> …） 42

 **添加主键** （add constraint <约束名> primary key(字段名)） 43

 **添加唯一约束**  （add constraint <约束名> unique(字段名)） 43

 **添加外键约束** （add constraint… foreign key … references …） 43

 **添加条件(检查)约束** （add constraint <约束名> check(条件)） 43

 **添加非空约束**  （modify <字段名> not null） 43

 **添加默认值约束** （modify <字段名> default <默认值>） 44

 **删除约束**  （alter table … drop） 44

13.4 删除基本表 44

14. 创建视图与索引 45

* 1. **SELECT语句的执行顺序 ★★★★★★★★★★★★★★★★★★★★**

FROM：执行顺序为从后往前、从右往左。数据量较少的表尽量放在后面。

ON：

JOIN:

WHERE：执行顺序自下而上、从右到左。将过滤最大数量的条件表达式写在最后面。

GROUP BY：执行顺序从左往右。最好在分组前使用WHERE将不需要记录在组内的数据过滤掉。

HAVING：消耗资源，尽量不使用。

SELECT：少用\*，尽量使用字段名。Oracle在解析时，将\*依次转换为字段名。

ORDER BY：执行顺序从左到右排序，消耗资源。

例：SELECT 班级,avg(数学成绩) as 数学平均成绩

FROM 学生表

WHERE数学成绩 is not null

GROUP BY 班级

HAVING 数学平均成绩 > 75

ORDER BY 数学平均成绩 DESC;

执行解读：

① 首先执行from字句，从学生表中组装数据源的数据；

② 执行where字句，筛选学生表中数学成绩部位null的数据；

③ 执行group by字句，按照“班级”进行分组；

④ 计算avg聚合函数，按每个班级分组求出各班级的数学平均成绩；

⑤ 执行having字句，筛选出班级数学平均成绩大于75分的；

⑥ 执行select语句，选取将要返回的字段；

⑦ 执行order by字句，把最后的结果按照“数学平均成绩”进行排序。

注：第④步，执行group by后先执行了聚合函数。若将avg(数学成绩)>75放在where后，则 会出现报错，其根本原因是聚合函数在where后时还是未知的，没有对虚拟表完成分组或筛选。 这也就是聚合函数不能用在WHERE后面只能放在select与having后面的原因。

* 1. **小知识 #** [**学生选课的表位置**](#_嵌套查询(子查询))
* SQL语言不进行必须规范大小写，但数据表中字符型的元素区分大小写。

例：SELECT \* FROM student WHERE xing\_ming=’QLJ’;

SELECT \* FROM student WHERE xing\_ming=’qlj’;

两行代码中条件语句的qlj不同，区分大小写。

* **DISTINCT** (distinct) # 去掉查询结果中的重复行

例：SELECT DISTINCT xing\_ming FROM student

* 字段别名

1. 字段=别名

SELECT xing\_ming=name FROM student;

1. AS关键字指定别名

SELECT xing\_ming AS name FROM student;

1. 空格隔开

SELECT xing\_ming name FROM student

* = 的作用

① 对查找字段赋以别名；

② 在where 字句内做比较判断，可以理解为：在某字段中找出指定数据值的行。

* 查询语句中，当设置字段的字符长度小于数据长度，则直接被截取。

例：SELECT name FROM country WHERE name =’CH’;

原始数据为CHINA ，查找后的虚表数据值为CH。

* 字符的比较按照ASCII值

例：SELECT \* ROM student WHERE xing\_ming<’CH’;

* with as 用法

with as作用：定义一个或多个sql片段，该片段会被整个sql语句所用到，特别是union all语句。（依次构建，多次使用）

WITH

t1 AS

(SELECT \* FROM m\_gat.tb\_softwater\_1shift\_mid)**,**

t2 AS

(SELECT \* FROM m\_gat.tb\_softwater\_1shift\_mid)

SELECT \* FROM t1；

Limit用法

* 1. **数值型运算**

加、减、乘、除、取模。 (可以建立新列<虚表中>)

* + 1. **取模（%）**

返回一个除法结果中商的余数部分。 # 5%2=1

例：SELECT nian\_ling,nian\_ling%zhou\_qi FROM student;

注：小数不能进行取模运算

函数MOD()求模：SELECT nian\_ling,MOD(nian\_ling,zhou\_qi) FROM student;

两种结果输出一致。

* + 1. **加法（+）**

在SELECT中数值字段后面使用。

例：①SELECT nian\_ling,nian\_ling+1 FROM student;

②SELECT nian\_ling,(nian\_ling+1) age FROM student;

# 不同数据软件可能去掉括号，age为nian\_ling+1的别名

上面两例中①输出nian\_ling 与nian\_ling+1两列；②输出nian\_ling与age两列

* + 1. **减法（-）**

两种使用方法：在SELECT的数值字段后做负号和两个数值字段的减法。

例：①SELECT -nian\_ling FROM student; #添加负号

②SELECT （nian\_ling–nian\_ji） age FROM student; #入学年龄=年龄-年级

* + 1. **除法（/）**

例：SELECT nian\_ling,(nian\_ling/12) var\_1 FROM student;

* + 1. **乘法（\*）**

例：SELECT nian\_ling,(nian\_ling\*0.5) var\_1 FROM student;

* 1. **比较运算**

将两个表达式进行比较并返回三个数值（True，False，Unknow）中的一个。

* Null与Unknow的区别：

Null 表示字段中某单元为空值。 # 空值：数据为空，但可能被指定单元格类型

# 空白值：无任何数据类型的空值。

Unknow 表示作比较后不能确定是否正确或者错误（是一种不正常的状态，例如：Ture）

IS (NOT) NULL 判断数据表中是否含有空白值（无任何类型的数据）

* IS NULL与 = 的区别：

在条件语句WHERE 中，= 不能判断出空值。

例：假设年龄字段存在空值，查找空值元组。

SELECT nian\_ling FROM student WHERE nian\_ling = NULL;

# 返回的结果找不到空值所在的行。

IS (NOT) NULL 是专门用来查找数据库是否含有空值。

例：SELECT nian\_ling FROM student WHERE nian\_ling IS NULL;

# 返回结果为空值所在的行。

注：① 空值不能被索引；

② null在排序被默认为最大；

④ null与任何值做算术运算结果为null；

④ null 无法用比较运算符（=，<，>，!=）；

⑤ 在报表开发中通常使用函数NVL(p1,p2)忽略空值。

NVL(p1,p2) #p1为空时返回p2，否则返回p1。

* 1. **字符匹配与连接**
     1. **字符匹配(like)**

**LIKE**

查找指定的属性列值与相匹配的元组。匹配的字符串可以是完整的字符串也可以是一部分。

例：SELECT xing\_ming FROM student WHERE xing\_ming LIKE ‘qlj’;

* 通配符

|  |  |  |
| --- | --- | --- |
| **% (百分号)** | 表示任意长度的字符串 | **a%b** 表示a开头b结尾的任意长度字符串 |
| **\_ (下划线)** | 表示任意单个字符 | a\_b 表示a开头b结尾字符串长度为3的任意字符串 |

注：若字符串内包含% 或者\_ ，将字符使用\ 转义为普通字符。

例： LIKE ‘%qu\\_jie%’ # 查找包含qu\_jie的所有元组

* + 1. **连接(||)**

将两个字符串连接起来。

例：SELECT xing||ming xing\_ming FROM student;

SELECT xing||’-’||ming xing\_ming FROM student; #多连接

注：||将字段xing 与字段ming 连接并命名为xing\_ming字段。+ 不能连接字符串。

连接后两个字段的数据合并为一个字段，但数据中间可能会有大量空格，空格来源是由于字段设置的最大字符数大于实际字符数，合并时按最大字符数进行合并，因此会产生空格。

* 1. **逻辑运算**

WHERE 语句中将两个或多个字段组合在一起。

* + 1. **AND**

例：SELECT name FROM student WHERE age>10 AND gender=’man’;

#同真则真

* + 1. **OR**

例：SELECT name FROM student WHERE age>10 OR gender=’man’;

#同假则假

* + 1. **NOT**

例：SELECT name FROM student WHERE age IS NOT NULL;

* 1. **集合运算(多个查询语句的集合运算)**
     1. **UNION (并运算)**

将合并两个或多个SELECT语句，结果显示一个基本表（去除其中的重复部分）。

合并前提：UNION内部的SELECT语句必须有相同数量的列，列必须拥有相似的数据类 型，同时，每条SELECT语句中的列顺序必须相同。

例：SELECT name\_1 FROM student UNION SELECT name\_2 FROM worker;

注：1、合并的表列名始终为第一个SELECT语句的列名。

2、两个union的子查询不能同时写order by，只能在最后一个子查询写。

**UNION ALL** 合并后不去重复项

* + 1. **INTERSECT (交运算)**

将两个或多个SELECT语句中相同的行进行显示。

例; SELECT name\_1 FROM student INTERSECT SELECT name\_2 FROM worker;

# 不同的值只会被列出一次

* + 1. **MINUS(差运算)**

返回的记录是存在于第一个表中但不存在于第二个表中的记录

例：SELECT name\_1 FROM student MINUS SELECT name\_2 FROM worker;

* + 1. **(NOT) IN (确定集合)**

查找属性值(不)属于指定集合的元组。

例：SELECT \* FROM student WHERE name IN(’屈刘杰’,’王泽轩’,’许永琪’);

* + 1. **(NOT) BETWEEN AND (确定范围)**

查找属性值(不)在指定范围内的元组。 （between and 包含边界）

例：SELECT \* FROM student WHERE age NOT BETWEEN 10 AND 23;

# 查询学生表中年龄不在[10,23]之间的所有元组。

注：[10,23]为闭区间，not后，输出结果不包含10和23两端数据。

* 1. **函数**
     1. **汇总函数 (聚集函数)**

汇总函数作用于SELECT后的字段名上，WHERE 后不能使用。

* COUNT (计数)

例：SELECT COUNT(\*) FROM student WHERE age>20;

# 汇总结果表的列名为COUNT(\*)，且只有一个元组数据为整数值。

* SUM (求和) 只针对数值类型的列

例：SELECT SUM(age) FROM student WHERE age>20;

SELECT COUNT(name),SUM(grade), SUM(grade)/ COUNT(name) AS average FROM student WHERE age>20; # average全班平均成绩

* AVG (平均值) 只针对数值类型的列

例：① SELECT AVG(age) FROM student WHERE age>20;

② SELECT AVG(grade/name) FROM student WHERE age>20;

③ SELECT AVG(grade)/AVG(name) FROM student WHERE age>20;

注：②与③的计算方式不同，平均值结果也不同。

* MAX (最大值)

例：SELECT MAX(age) FROM student WHERE age>20;

SELECT MAX(name) FROM student WHERE age>20;

# 字段为姓名时，max返回字符串最大的元组。

* MIN (最小值)

例：SELECT MIN(age) FROM student WHERE age>20;

SELECT MIN(name) FROM student WHERE age>20;

# 字段为姓名时，min返回字符串最小的元组。

* ARIANCE (方差ariance) 只针对数值类型的列

例：SELECT ARIANCE(grage) FROM student; #成绩的方差

* STDDEV (标准差stddev) 只针对数值类型的列

例：SELECT STDDEV(grage) FROM student; #成绩的方差

* + 1. **日期/时间函数**

案例：学生表(student)：姓名(name)；年龄(age)；入学日期(date\_begin)；

毕业日期(date\_end)

日期格式：



* ADD\_MONTHS(x,y) # 增加或减少指定个月

x 表示被处理的日期字段； y 表示增加或减少的日期数

对日期格式的字段（含月份）增加或减少指定个月。

例：①SELECT ADD\_MONTHS(date\_begin,3) FROM student; # 增加3个月

②SELECT ADD\_MONTHS(date\_begin,-3) FROM student; # 减少3个月

③SELECT date\_begin，date\_end FROM student

WHERE ADD\_MONTHS(date\_begin,-3) < date\_end;

# ADD\_MONTHS 函数可以应用在WHERE语句中

* LAST\_DAY(time) # 返回指定日期所在月份的最后一天

返回指定日期所在月份的最后一天。 # time 为日期属性列

例：SELECT date\_end,LAST\_DAY(date\_end) day\_end FROM student;

注：闰年的月份最后一天有差异。

* MONTHS\_BETWEEN(x,y) # 返回两个日期之间的月份数

返回两个日期之间的月份数。 # 可用于WHERE语句中

① x > y (x的时间在y之后，x是y的未来时间):

SELECT MONTHS\_BETWEEN(date\_begin,date\_end) value FROM student;

# date\_begin = 2016-05-19,date\_end = 2016-03-23返回的值value=1.87。

② x < y (y是x的未来时间):

SELECT MONTHS\_BETWEEN(date\_begin,date\_end) value FROM student;

# date\_begin = 2016-05-19,date\_end = 2016-03-23返回的值value= -1.87。

③x = y，返回0。

注：x与y在函数参数的前后会影响返回的值正负，因此，通过WHERE 语句可以根据MONTHS\_BETWEEN 判断两个日期的前后关系，然后进行查询输出。

* NEW\_TIME(date,x,y) # 时区变换

给出时间date在x时区对应y时区的日期和时间。即，将x时区的时间对应到y时区上。

# x与y有特定的时区表示形式,可查询获得。

SELECE NEW\_TIME(date\_begin,’EDT’,’PDT’) new\_t FROM student;

* NEXT\_DAY(date,weekday) # 未来最近的指定星期对应的日期

返回输入日期开始，未来最近的指定星期对应的日期。

例：SELECT date\_begin NEXT\_DAY(date\_begin,’星期一’) FROM student;

# date\_begin = 2021-08-04，返回日期：2021-08-09。

# weekday 可以填写类型：星期几、1-7(1表示星期日)、MON-SUNDAY。

* SYSDATE # 系统日期

返回系统的日期与时间

例：SELECT SYSDATE FROM student;

* TO\_CHAR(date,x) # x 表示要转换成的格式

将日期型的数据转换为指定的字符型数据格式。(日期-->字符)

可用于单个年、月、日、时、分、秒。

例： date\_begin = 2017-08-30

① SELECT to\_char(date\_begin,’YYYY-MM-DD HH24:MI:SS’) FROM student;

# date\_begin 列中没有时-分-秒，此处只是为了说明可以添加此格式。

② SELECT to\_char(date\_begin,’YYYY-MM-DD’) FROM student;

# 输出：2017-08-30

③ SELECT to\_char(date\_begin,’MM’) FROM student;

# 输出：08

* TO\_DATE(date,x) # x 表示转换的标准日期格式。

将字符型日期转换为日期数据类型。(字符-->日期)

例：①SELECT to\_date(‘2021-08-09 09:58:29’,’YYYY-MM-DD HH24:MI:SS’) FROM student;

②SELECT to\_date(‘2021-08-09’,’YYYY-MM-DD’) FROM student;

注：date与x的格式要必须匹配，hive中只能转换YYYY-MM-DD格式的字符串。

* + 1. **数学函数**
* ABS(x) (绝对值)

返回指定值的绝对值

例：SELECT ABS(-1.2) FROM value; # 返回值：1.2

* CEIL(x) (向上取整)

返回大于或等于给出值的最小整数。

例：SELECT CEIL(-1.2) FROM value; # 返回值：-1

* FLOOR(x) (向下取整)

对给定的数字取整数（舍去小数位）。

例：SELECT FLOOR(-1.2) FROM value; # 返回值：-2

* ROUND(x,[y]) (四舍五入) # y 表示保留位数

对给定小数进行四舍五入。

例：① SELECT ROUND (1.278) FROM value; # 返回值：1

② SELECT ROUND (1.278,2) FROM value; # 返回值：1.28

③ SELECT ROUND (117.278，-1) FROM value; # 返回值：120

注：y 为可选项，忽略时截取所有小数部分，即y=0。

* MOD(x,y) (取模)

x除以y返回余数值

例：SELECT MOD(3,2) FROM value; # 返回值：1

SELECT MOD(2,3) FROM value; # 返回值：2

* POWER(x,y) # (x^y)

对x取y次方。

例：SELECT POWER(3,2) FROM value; # 返回值：9

* EXP (x) # (e^x)

以e为底数的幂值。

例：SELECT EXP(2) FROM value; # 返回值：7.389……

* LN(N)、LOG(a,N)

LN(N) 返回给定参数的自然对数，LOG(a,N) 以a为底数，N为真数做对数运算。

例; SELECT LN(1) FROM value; # 返回值：0

SELECT LOG(2,4) FROM value; # 返回值：2

注：对数函数中底数a={a|a>0,a!=1},真数N值有取值范围(0,+∞)

* SQRT(x) 求根

返回参数x值的算术平方根。

平方根：若一个数的平方等于a，则这个数叫做a的平方根。 # (+2)^2 = 4

算数平方根：若一个正数的平方等于a，则这个正数叫做a的算数平方根。

例：SELECT SQRT(4) FROM value; # 返回值：2

* SIGN(x) 符号函数

x > 0,sign = 1；

x = 0,sign = 0；

x < 0,sign = -1

例：SELECT SIGN(4) FROM value; # 返回值：1

* TRUNC(x,[y]) 截取函数

截取数字或日期

① 对数字TRUNC(x,[y]) # y表示保留小数位个数

例：SELECT TRUNC(17.17) FROM value; # 返回值：17

SELECT TRUNC(17.17,1) FROM value; # 返回值：17.1

SELECT TRUNC(17.17,-1) FROM value; # 返回值：10

SELECT TRUNC(17.17,-3) FROM value; # 返回值：0

② 对日期TRUNC(x,[y]) # y表示日期格式，可忽略。

例：① SELECT TRUNC(

to\_date(‘2018-02-01 1:00:00’,’YYYY-MM-DD HH24:MI:SS’),

’YYYY’) FROM value; # 返回值：2018/1/1

# 返回当前年的第一天。

② SELECT TRUNC(

to\_date(‘2018-02-02 1:00:00’,’YYYY-MM-DD HH24:MI:SS’)

’MM’) FROM value; # 返回值：2018/2/1

# 返回当月第一天

③ SELECT TRUNC(

to\_date(‘2018-02-02 1:14:23’,’YYYY-MM-DD HH24:MI:SS’)

’MM’) FROM value; # 返回值：2018/02/02 1:14:00

# 返回当前日期截取到分，秒位补0.

注：TRUNC()函数例②截取的返回值与TO\_CHAR()中例③按类型转化的返回值不同。

按日期类型截取后注意补0或为当前日期第一天

* 三角函数

SIN() 正弦函数、COS() 余弦函数、TAN() 正切函数；

SINH() 双曲正弦函数、COSH() 双曲余弦函数、TANH() 双曲正切函数；

ASIN() 反正弦函数、ACOS() 反余弦函数、ATAN()反正切函数

功能：返回给定参数的三角函数值，默认的参数为弧度制

注：1O=pi/180(弧度)=; 0.01745329251994 # 45 O = pi/180\*45

例：SELECT COS(45\* (pi/180)) FROM value;

返回值：0.70710678

* + 1. **字符函数**
* CHR(x) # 将ASCII码转换为对应字符

例：SELECT CHR(9) FROM value; # 制表符

* ASCII(x) # 将字符转换为ASCII码

例：SELECT ASCII(‘t’) FROM value; # 返回值：116

* CONCAT(str1,str2) # 只能连接两个字符串

例：SELECT CONCAT(‘qu’,’liujie’) FROM value;

连接多个字符串：使用concat()函数嵌套。

例：SELECT CONCAT(CONCAT(‘qu’,‘liu’),’jie’) “my name” FROM value;

注：在命名时使用多个字符串进行命名时，需要对属性名添加引号。

* INITCAP(str) <initcap> # 首字母大写，其余字母小写

例;SELECT INITCAP(‘HELLO WORLD’) FROM value;

# 返回值：Hello World

* LOWER(str)、UPPER(str) # 全部小写and全部大写

例;SELECT LOWER(‘Hello World’) FROM value;

# 返回值：hello world

SELECT UPPER(‘Hello World’) FROM value;

# 返回值：HELLO WORLD

* LPAD(str1,num,[str2])、RPAD(str1,num,[str2]) # 左填充/截取；右填充/截取

str1：字符串； num：设置返回的字符串长度； str2：可选，设置填充的字符，不选为空格

当num < str1 （字符长度）：作截取（保留），L左截取，R右截取，截取长度num

当num > str1 （字符长度）：作填充，L左填充，R右填充，填充后字符串长度num， str2为填充的字符，不选默认补充空格。

例：① SELECT RPAD(LPAD(‘quliujie’,10,’-\_’),12，’\_-’) FROM value;

返回值：-\_quliujie\_-

② SELECT RPAD(LPAD(‘quliujie’,5),3) FROM value;

返回值：liu

注：若字段原设置的字符数大于字符串长度（设置15，字符串8），则实际字符串长度 为15。也就是说，在函数LPAD()中做运算时，str1参数的字符长度为15。

* LTRIM(str1,[ str2])、RTRIM(str1,[ str2]) # 从左/右切除(删除)指定字符，默认空格

str1：字符串； str2：可选，设置切除的字符，默认为切除空格

例：① SELECT TRIM(‘ adwe ’) FROM value;

返回值：adwe # 删除两边空格

② SELECT LTRIM(‘acafacafaca’,’ac’) FROM value;

返回值：facafaca # 遇到f切除停止

③ SELECT RTRIM(‘acafacafaca’,’ac’) FROM value;

返回值：acafacaf # 遇到f切除停止

注：LTRID()在切除’ac’时，从字符串左边开始时搜索，若str1包含str2中的任一字 符，就进行切除。当在搜索过程中遇到不是str2的字符时，切除停止。

* REPLACE(str1,str2[,str3]) # 替换字符串

str1：所选字段名/字符串； str2：数据内需要替换的字符串；

str3：需要替换成的字符串。默认为空，即删除str1中包含str2的内容

例：① SELECT REPLACE(‘quliujie’,’jie’,’jia’) FROM student;

# 返回值：quliujia

② SELECT REPLACE(‘quliujie’,’qu’) FROM student;

# 省略参数str3，删除’qu’。 返回值：liujie

* TRANSLATE(str1,str2,[str3]) # 替换字符(注：与REPLACE不同)

将出现在str2中的每个字符替换为str3中相对应的字符，再用str2替换掉str1包含str2的字符，然后返回修改后的str1.

注：先将str2与str3每个字符进行对应，当字符数str2 > str3，则超出的部分即对应空值；

然后看str1与str2相交字符，将str1中相交部分进行替换（替换为str3的内容），其中str2起到的作用就是“桥梁”作用。 （该函数区分大小写）

str1：所选字段名/字符串； str2：数据内需要替换的字符串(可非数据内字符)；

str3：需要替换成的字符串。默认为空，即删除str1中包含str2的内容

例：① SELECT TRANSLATE(‘aqduofsadwi’,’siqo’,’xxcc’) FROM student;

# 返回值：acducfxadwx # s-x，i-x，q-c，o-c

② SELECT TRANSLATE(‘abcdefg’,’acdb’,’swj’) FROM student;

# 返回值：swjefg # a-s，c-w，d-j，o-空值（可用于删除）

③ SELECT TRANSLATE(‘abcdefg’,’acdb’,’’) FROM student;

返回值：NULL # str3 为空时返回null。

translate()函数与replace()的区别：translate()的替换是字符对应逐个替换，replace()函数是整个字符串的替换。

* SUBSTR(str1,num1,num2) # 截取（与Excel中mid()相似）

字符截取函数。 # num1 :开始位置 num2：截取长度

例:① SELECT SUBSTR(‘helloworld’,0,3) FROM student;

# 返回值：hel # 截取长度为1或0都是从第一位开始截取

② SELECT SUBSTR(‘helloworld’,-2,3) FROM student;

# 返回值：ld # 从倒数第二个字符开始向后截取三个字符

③ SELECT SUBSTR(‘helloworld’,-2) FROM student; # 省略第三个参数

# 返回值：ld # 截取从倒数第二个字符到最后

④ SELECT SUBSTR(‘helloworld’,2) FROM student;

# 返回值：elloworld # 截取从第二个字符到最后

注：一般主要与’||’和concat()混合使用。

* INSTR(str1, str2,[num1],[num2]) # 字符查找（与Excel中search类似）

返回指定的字符在原字符串里的位置。 （只检索1次）

str1：检索的字符串/字段； str2：指定的目标字符/字符串；

num1：检索的起始位置（默认从头1开始）；

num2：检索第num2次str2字符出现的位置。（默认为第1次出现）

例：① SELECT INSTR(‘helloworld’,’l’) FROM student;

返回值：3 # 参数num1与num2默认为1

② SELECT INSTR(‘helloworld’,’ld’) FROM student;

返回值：9 # 查找’ld’中l的位置

③ SELECT INSTR(‘helloworld’,’l’,-2,3) FROM student;

返回值：3 # 从倒数第2位置开始往回查找，第3个l位置。

* LENGTH(str1) # 字符串长度

例：SELECT LENGTH(‘helloworld’) FROM student;

返回值：10

注：字符串为空时LENGTH(‘’)，返回null。

* LENGTHB(str1) # 字符串字节长度

例：① SELECT LENGTHB(‘helloworld’) FROM student;

返回值：10

② SELECT LENGTHB(‘你好,世界’) FROM student;

返回值：13

注：在Oracle中，一个汉字占3个字节；

当LENGTHB()的参数全为单字节字符时，LENGTHB()与LENGTH()是一样的；

Oracle中常用LENGTHB(str1) = LENGTH(str1)判断字符串中是否包含汉字。

* TO\_CHAR() # 将date/number转换为char
* TO\_NUMBER() # 将char转换为number
* TO\_DATE() # 将number/char转换为date

注：① 对数字类型使用LENGTH()函数，将返回错误信息。

② to\_char()与chr()的区别：

chr()是返回指定的ASCII码的字符。

* rownum 关键字(伪列) # 构建行号字段

表示行号，实际构造了一个列，这个列是以行号为数据值的。

位置：在select后，当做字段名来使用，相当于oracle内本身就存在一个rownum字段。

例： SELECT ROWNUM,st.\* FROM student st;

# 第一列为行号字段

注：可通过新构建的rownum字段控制输出行数。

例：SELECT ROWNUM,st.\* FROM student st where rownum <= 10

# 输出前10行（不支持 > 号）

支持 > 方法：将上面查询结果集作为子查询。（将构建的rownum列重新命名，并作为子查询，此时行号为一个字段，外层select再通过where条件查询新的字段名即可）

* group\_concat # 多行转一行功能

将查询出的某一列值使用逗号进行隔开拼接，成为一条数据。

可联合group by一起使用，实现分组拼接

* + 1. **分析函数**
* 什么是分析函数？

分析函数是oracle专门用于解决复杂报表统计需求的函数。它可以在数据中进行分组然后计算基于组的某种统计值，并且每一组的每一行都可以返回一个统计值。

* 分析函数与聚合函数的不同之处？

普通的聚集函数用group by进行分组，每个分组只能返回一个统计值，而分析函数采用partition by分组，并且每组每行都可以返回一个统计值

* 分析函数的形式

分析函数带有一个开窗函数**over()**，包含三个分析字句（参数）：分组（partition by）、排序（order by）、窗口（rows）。

**over(partition by [字段名] order by [字段名] [rows between 参数])**

**between后参数**

1、< unbounded preceding and unbound following>

// 针对当前所有记录的前一条、后一条记录，也就是表中的所有记录。

2、< unbounded preceding and current row >

// 指第一行到当前行的记录

3、< 1 preceding and current row >

// 指当前行的上一行(rownum-1)到当前行的记录

4、< 1 preceding and 2 following >

// 指当前行的上一行(rownum-1)到当前行的下两行(rownum+2)的记录

Unbounded：不受控制的，无限的

Preceding：在……之前

Following：在……之后

注：窗口字句在这里只说rows方式的窗口，range方式和滑动窗口不做介绍。

* 常用分析函数

1. 聚集函数下的分析函数

Count()over() // 统计分区中各组的行数

Sum()over() // 统计分区中记录的总和

Avg()over() // 统计分区中记录的平均值

min()over() // 统计分区中记录的最小值

max()over() // 统计分区中记录的最大值

【例】下面以min()为例，其他聚集函数与min()类似

// 求总的最小值

Select 字段名,min(colName)over( ) from 表名；

// 递加求最小值

Select 字段名,min(colName)over(order by 字段名 ) from 表名；

// 分组求最小值

Select 字段名,min(colName)over(partition by 字段名 ) from 表名；

// 分组递加求最小值

Select 字段名,min(colName)over(partition by 字段1 order by 字段2) from 表名；

2、其他常用分析函数

* **rank()over()** // 跳跃排序

partition by 可选，order by 必选

（相同值排名相同，排名跳跃）

* **dense\_rank()over()**  // 连续排序

partition by 可选，order by 必选

（相同值排名相同，排名连续）

* **row\_number()over()**  // 排序

partition by 可选，order by 必选

（排名连续，无论值是否相同）

* ntile(n)over()

partition by 可选，order by 必选

（n表示将分区内记录平均分为n份，多出的按照依次分给前面的组 ）

* 1. **SQL字句**
     1. **WHERE**

返回符合搜索条件的行记录。

WHERE <条件表达式>

表 1 WHERE字句常用查询谓词

|  |  |
| --- | --- |
| **查询条件** | **谓词** |
| 比较 | =，>，<，>=，<=，!=，<>，!>，!<；NOT+前面的运算符 |
| [确定范围](#_(NOT)_BETWEEN_AND) | BETWEEN x AND y，NOT BETWEEN x AND y |
| [确定集合](#_(NOT)_IN_(确定集合)) | IN，NOT IN |
| [字符匹配](#_字符匹配(like)) | LIKE，NOT LIKE |
| [空值](#_比较运算) | IS NULL，IS NOT NULL |
| [多重条件(逻辑运算)](#_逻辑运算) | AND，OR，NOT |

* STARTING WITH字句 # 与like(str%)效果相同

附加在WHERE子句后面。

例：SELECT xing\_ming FROM student WHERE STARTING WITH(‘qu’)

# 查询xing\_ming字段中以’qu’开头的所有元组。

* + 1. **GROUP BY （分组）**

对一个或多个字段进行分组，一般与汇总函数一起使用做统计分析。

GROUP BY <列名1>[,<列名2>…] [HAVING<条件表达式>]

注：① 分组查询时，查询函数select后查询字段只能为：分组字段；min；分组表达式。

② HAVING 后的条件表达式作用于组，先用GROUP BY对字段进行分组，再根据having条 件决定返回哪些组；（即，having对分组后的组进行过滤。）

③ min用于对分组后的各组进行统计分析，若不进行分组，min作用于整个查询 结果。

* 单列表组查询

将查询出来的组按照某一特定的列进行分组，分组列的值相等的元组为一组。

例：SELECT gender,count(\*),avg(age) FROM student WHERE age>18

GROUP BY gender;

# 返回的表格以性别进行分组，并进行统计人数及平均年龄（大于18）。

注：若select后出现非分组列（如：添加在gender后xing\_ming列），则输出导致错误。

* 多列分组查询

指定两个或多个分组列，根据分组列组合的不同值进行查询分组并进行统计。

例：SELECT gender,department,count(\*),avg(age) FROM student WHERE age>18

GROUP BY gender,department;

# 将性别与院系组合在一起进行分组。

* 使用HAVING限制返回组

例：SELECT gender,count(\*),avg(age) FROM student

GROUP BY gender HAVING avg(age)>18;

注：having后的部分为分组后限制平均年龄大于18的组进行输出。与单列分组案例不同。

按照查询语句的逻辑顺序进行思考即可。

* + 1. **ORDER BY （排序）**

对查询结果按照一个或多个属性列的升序(ASC)或降序(DESC)排序。（默认升序）

例：SELECT \* FROM student ORDER BY name DESC;

# 对name字段进行降序排序。

* 多列排序

按多列或多个表达式排序时，先按照第一列或表达式进行排序，当第一列的数据相同时，再以第二列进行排序，以此类推。（也可以先升序再降序，对目标列1升序，对目标列2降序）

例：SELECT \* FROM student ORDER BY department,name DESC;

# 先用院系进行降序排序，再通过姓名进行降序排序。

* 按表达式排序

按特定的表达式进行排序

例：SELECT \* FROM student ORDER BY department,age+4 DESC;

# 先用院系进行降序排序，再通过年龄表达式进行降序排序。

* 使用别名排序

对目标列定义别名，排序时使用目标列的别名。

例：SELECT department,age+4 new\_age FROM student ORDER BY new\_age;

# 对年龄字段定义别名，再使用新的字段名进行升序排序。

* 用列位置编号排序

按照目标列的位置编号进行排序。

例：SELECT department,age+4 new\_age FROM student ORDER BY 2 ASC;

# 根据年龄字段进行升序排序。

* 自定义排序

根据目标列的数据进行自定义排序。

例; SELECT department,gender FROM student

ORDER BY decode(gender,’男’,1,’女’,2,0);

# 按照性别字段男、女依次排序

decode(条件，值1，返回值1，值2，返回值2，…，值n，返回值n，缺省值)

* 1. **多表查询**
     1. **连接(交叉连接)**

两个或多个表之间的无条件连接。一个表所有元组分别与其他表中所有元组进行连接。

笛卡尔积：设A和B是两个集合，存在一个集合，它的元素是用A中元素为第一元素，B中元 素为第二元素构成的有序元组。称这个集合为A和B的笛卡尔积。记为A×B。

笛卡尔积是一个集合。集合的元素都是序偶。 <一条记录为集合的一个元素>

例：SELECT \* FROM 学生,选课;

SELECT \* FROM 学生 CROSS JOIN选课; # 标准SQL

* + 1. **内连接(INNER JOIN…ON…)**

根据指定的连接条件进行连接查询，只有满足连接条件的数据才会被显示。

* 等值连接

WHERE连接条件为两个表的字段进行等号连接。

例：SELECT \* FROM 学生 st,选课 sc WHERE st.学号=sc.学号;

或SELECT \* FROM 学生 st INNER JOIN选课 sc ON st.学号=sc.学号;

# ON后面是连接条件。内连接inner可省略。

# 将学生表与选课表中的学号字段进行等值连接。

* 非等值连接

连接条件是除等号运算符以外还可以有其他的操作符限制连接结果。

将等值连接的运算符改为或添加 >，<，>=，<=，!=，<>，LIKE，IN，BETWEEN AND。

例：SELECT \* FROM 学生 st,选课 sc WHERE st.学号=sc.学号 AND age>18;

或SELECT \* FROM 学生 st INNER JOIN选课 sc ON st.学号=sc.学号 AND age>18;

* + 1. **自身连接(SELF JOIN)**

在同一个表或视图中进行连接，相当于同一个表作为两个或多个表使用。

例：SELECT \* FROM student st1,student st2 WHERE st1.sno=st2.sno;

* + 1. **自然连接(NATURAL JOIN…ON…)**

在两个表中寻找数据类型和列名都相同的字段，然后自动的将它们连接起来，并返回符合条件的结果。 (特殊的等值连接，将重复列去掉)

例：① SELECT \* FROM 学生 st,课程 sc WHERE st.sno=sc.sno;

或② SELECT \* FROM 学生 st NATURAL JOIN 课程 sc;

相当于：③ SELECT \* FROM 学生st NATURAL JOIN 课程 sc ON st.sno=sc.sno;

注：

★ 在②中未指定连接条件，是因为Oracle会自动识别表中是否有相同列。

★ 如果自然连接时两个表有多个字段都满足有相同名称与类型，那么它们都会被作为连接 条件。

★ 如果自然连接的两个表仅字段名相同，但类型不同，则会返回错误。

* + 1. **外连接(OUTER JOIN)**

外连接分为：左外连接、右外连接、完全外连接。

对应SQL语法：LEFT/RIGHT/FULL OUTER JOIN （使用语法时可省略outer）

悬浮元组：在下图的自然连接中，表str\_1的6,7元组被舍弃，表str\_2的4,8元组被舍弃。 这些舍弃的元组称为悬浮元组。

例：如下图中表str\_1与str\_2：

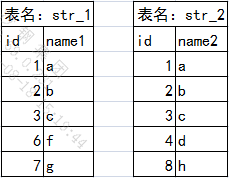
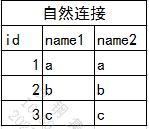


图 1作自然连接

图 2 定义表str\_1与表str\_2

* 左外连接(LEFT JOIN…ON…)

左外连接是以左表为基表，右表为匹配表，即左表会被全部显示，右表根据连接条件进行匹配，不能匹配的部分均为NULL。 （LEFT OUTER JOIN…ON） # 可省略OUTER

例：① SELECT \* FROM str\_1 LEFT JOIN str\_2 ON str\_1.id= str\_1.id;

**用（+）来实现。**+ 可理解为**补充**，加号所在表为匹配表。

① 操作符 + 只能出现在where字句中，且不能与outer join语法同时使用；

② 当 + 用于外连接时，如果在where中包含多个条件，则必须所有条件中都包含 + ；

③ + 只适用于列，不能用在表达式内；

④ + 不能与or与in一起使用；

⑤ + 只能用于实现左外连接和右外连接，而不能用于实现完全外连接。

例：② SELECT \* FROM str\_1,str\_2 WHERE str\_1.id= str\_1.id(+);



* 右外连接（RIGHT JOIN…ON）

右外连接是以右表为基表，左表为匹配表，即右表会被全部显示，左表根据连接条件进行匹配，不能匹配的部分均为NULL。

例：① SELECT \* FROM str\_1 RIGHT JOIN str\_2 ON str\_1.id= str\_1.id;

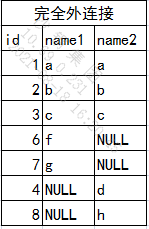
② SELECT \* FROM str\_1, str\_2 WHERE str\_1.id(+)= str\_1.id;



* 完全外连接(FULL JOIN…ON)

对左右表都不做限制，所有记录都会显示，两个表不足的地方用null补充。(不支持 + 方法)

例：SELECT \* FROM str\_1 FULL JOIN str\_2 ON str\_1.id= str\_1.id;



* 1. **嵌套查询(子查询)**

一个SELECT-FROM-WHERE语句称为一个查询块。讲一个查询块嵌套在另一个查询块中的查询语言称为嵌套查询。

在执行时，由里向外，先处理子查询，再将子查询的返回结果用于父查询（外部查询）执行。

子查询的作用：

① 在INSERT或CREATE TABLE语句中使用子查询，可将子查询结果写入到目标表内。

② 在UPDATE语句中使用子查询可以修改一个或多个记录的数据。

③ 在DELETE语句中使用子查询可以删除一个或多个记录。

④ 在WHERE和HAVING中使用子查询可以返回一个或多个值。

⑤ 在数据操纵（DML）与数据查询（DQL）语句中使用子查询时不能带有ORDER BY字句。在数 据定义（DDL）语句的子查询可以带有ORDER BY字句。

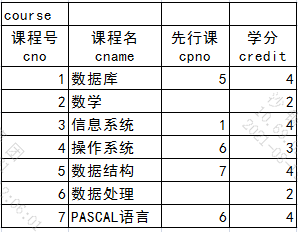




图 3课程表

图 4学生表

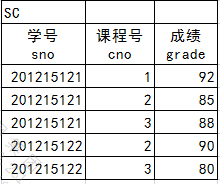


图 5 选课表

**不相关子查询**：子查询的查询条件不依赖于父查询，与父查询没有联系，父查询只使用了子查 询的返回结果。

**相关子查询**：子查询的查询条件依赖于父查询，与父查询有联系，子查询执行时需要引用外部 父查询的信息。

* + 1. **带有IN谓词的子查询**

与子查询返回结果中任何一个值相等。

例：① SELECT name FROM student

WHERE sno IN(SELECT sno FROM SC WHERE cno=’2’); # 不相关子查询

② SELECT sno,cno FROM SC x

WHERE grade>=(SELECT avg(grade) FROM SC y WHERE y.sno=x.sno);

# 相关子查询 # 子查询内使用了父查询的表x。

例②解析：首先在外层查询中执行from语句，取SC的一个元组，再将该元组sno值带入 内部查询；

执行内部查询，得到子查询的返回值88（近似），用该值代替子查询块；

执行外部查询，得到返回结果；

重复上述步骤直至外层SC表全部元组处理完成。

注：相关查询时，务必要根据嵌套查询的执行顺序（原理），逐步剖析。

* + 1. **带有ANY或ALL谓词的子查询**

子查询返回单个值时可以使用条件运算符，但返回值是多个值（集合）时，就需要ANY与ALL修饰符。

* ANY

|  |  |
| --- | --- |
| > ANY | 大于子查询结果的某个值 |
| < ANY | 小于子查询结果的某个值 |

例：SELECT name,age FROM student

WHERE age<ANY(SELECT age FROM student WHERE dept=’CS’)

AND dept<>’CS’;

相当于：SELECT name,age FROM student

WHERE age<(SELECT MAX(age) FROM student WHERE dept=’CS’)

AND dept<>’CS’;

解析：子查询块结果返回两个age值(22,19)，age<ANY(22,19)表示外部查询的age字段小于 22或小于19就可返回真值。相当于WHERE age<22 OR age>19

* ALL

|  |  |
| --- | --- |
| > ALL | 大于子查询结果的所有值 |
| < ALL | 小于子查询结果的所有值 |

例：SELECT name,age FROM student

WHERE age<ALL(SELECT age FROM student WHERE dept=’CS’)

AND dept<>’CS’;

解析：子查询块结果返回两个age值(22,19)，age<ALL(22,19)表示外部查询的age字段小于 22且小于19才可返回真值。相当于WHERE age<22 AND age>19

* + 1. **带有EXISTS谓词的子查询（可使用NOT EXISTS）**

EXISTS表示存在量词 **∃**。带有exists的子查询不返回任何数据，只产生逻辑真值“true”或逻辑假值“false”。 # 带有EXISTS的查询一般为相关子查询

当子查询是不相关子查询时，子查询有实际返回就为TRUE。（可以用该性质检查EXISTS内的子查询是否存在输出）

例：SELECT name FROM student

WHERE EXISTS(SELECT \* FROM SC WHERE sno=student.sno AND cno=’1’);

执行原理：① 首先取student表中的第一条元组，进入子查询；

② 依次判段进入到子查询的student.sno与SC.sno的元组是否相等且满足 cno=’1’。

③ 若找到与SC.sno中的某个元组满足条件，则认为存在一元组满足条件，所以 EXISTS下的子查询返回为真。

④ 将满足条件的student.name送入结果表。

SQL中没有全称量词 ∀，但全称量词可以对存在量词进行处理从而得到全称量词的效果。<具体参照《离散数学》>

* 1. **数据操纵语言（DML）**

增：INSERT INTO … VALUES…

删：DELETE FROM … WHERE…

改：UPDATE … SET … WHERE…

* + 1. **插入数据（增：INSERT INTO … VALUES…）**
* 单条元组插入

一个insert语句只能插入一行数据。

**INSERT INTO 表名[(字段名1，…)]** # into后可只跟表名，也可以指定具体字段

**VALUES(值1，值2，…)** # values对指定的字段对应插值

注;① 当into后只有表名时，表示对表中所有字段进行插入赋值；

② values中插入的数据类型、顺序、个数要与INTO中指定字段一致；

③ 不对某字段进行插入时（已指定该字段），需要明确该字段指定值为NULL；

④ 插入数据时注意表的完整性约束（如：主键字段不能插入与表中数据重复的值）；

⑤ 字符型与日期插入时要加单引号，日期类型按系统默认日期类型或用to\_date()转换。

例：① INSERT INTO student VALUES(‘202115126’,’屈刘杰’,’男’,20,’CS’);

② INSERT INTO SC(sno,cno) VALUES(‘202115126’,’1’)

# 可以指定部分字段进行插入。未指定字段位置自动赋空值。

③ INSERT INTO SC VALUES(‘202115126’,’1’,null)

* 利用子查询插入 # 多值插入/表的复制

将子查询得到的结果集插入指定的表中。（实现多值插入）

# 前提：子查询需要一个基本表再对其查询处理并符合待插入表的类型、个数、顺序。

注：insert内的表不能与from后的表相同。

**INSERT INTO表名[(字段名1，…)]**

**子查询; # 子查询中返回的字段与指定字段对应。**

例：假设存在一个表student\_new，其表内字段的顺序、个数、类型与student完全一致。

INSERT INTO student

SELECT \* FROM student\_new; # 将student\_new数据复制到student中。

* 多表插入

将一个子查询的结果集的数据分别插入不同表中。

all与first的区别：

INSERT **ALL** :在有条件插入中，每个when都会进行判断，若when满足则就会执行then。

INSERT **FIRST** :在有条件插入中，按顺序判断when字句，只执行第一个满足when的then字句。

**① 无条件多表插入**

INSERT [ALL]

INTO 表1 VALUES (字段1，[…])

INTO 表2 VALUES (字段1，[…])

……

子查询

例：假设存在student\_1与student\_2两个表，现要对两个表使用student进行插入操作。

INSERT ALL

INTO student\_1 VALUES(sno,name,sex,dept)

INTO student\_2 VALUES(sno,name,sex,age)

SELECT \* FROM student WHERE age>20;

# 子查询内要包括被插入表指定的字段即可。

**② 有条件多表查询**

INSERT [ALL/FIRST]

WHEN 条件1 THEN INTO 表1(字段1，[…])

WHEN 条件2 THEN INTO 表2(字段1，[…])

……

ELSE INTO 表n (字段1，[…])

子查询

例：INSERT ALL

WHEN sno=201215123 THEN INTO student\_1(sno,name,sex,age,dept)

WHEN name=’许永琪’ THEN INTO student\_2(sno,name,sex,age,dept)

ELSE INTO student\_3

SELECT \* FROM student;

* + 1. **数据修改（改：UPDATE … SET … WHERE…）**

修改一条元组数据或多条元组数据。

UPDATE<表名>

SET <列名>=<表达式> [,<列名>=<表达式>]…

[WHERE <条件>]; # where的作用：索引修改的元组

修改指定表中满足WHERE字句条件的元组。其中SET字句给出<表达式>的值用于取代相应的属性列值。省略WHERE，表示对所有元组进行修改。

* 单条元组修改

例：将学号为201215125的学生性别修改为女。

UPDATE student SET sex=’女’ WHERE sno=201215125;

* 多条元组修改

例：① 将性别为女且年龄大于等于20岁的学生性别改为男。

UPDATE student SET sex=’男’WHERE sex=’女’ AND age>=20;

② 将所有学生的所在系改为MA。

UPDATE student SET dept=’MA’; # 省略where

* 子查询修改

子查询块可以在set中使用，也可以在where内使用。需要注意的是在set与where中所代表的含义不同。

**① 在set中。**

例：查询学号为201215123学生性别，并用查询的性别替换掉学号为201215125的学生性别。

UPDATE student

SET sex=(SELECT sex FROM student WHERE sno=201215123)

WHERE sno=201215125; # 要对应查询块返回值与where指定的返回元组

**② 在where中**

例：将计算机科学系全体学生的成绩置0。

UPDATE SC

SET grade=0

WHERE sno IN (SELECT sno FROM student WHERE dept=’CS’);

* + 1. **数据删除（删：DELETE FROM … WHERE…）**

删除满足where条件的所有元组。省略where表示删除所有元组，但表中的字段定义依然存在于表中。

**注：delete删除的是表中的数据，而不是关于表中的定义。**

**DELETE FROM <表名> [WHERE <条件>];**

例：① 删除单个元组

DELETE FROM student WHERE sno=201215125;

② 删除多个元组

DELETE FROM student WHERE sex=’男’;

③ 子查询删除

DELETE FROM student WHERE sno IN (SELECT sno FROM SC);

# 删除学生表中没有成绩（对应于SC表）的学生。

* 1. **数据定义语言（DDL）**

表 2 Oracle常用数据类型

|  |  |  |
| --- | --- | --- |
| **类型** | **名称** | **说明** |
| char | 定长字符类型 | 未达到指定长度时，自动在末尾用空格补全，默认为1. |
| varchar2 | 变长字符类型 | 未达到指定长度时，不自动在末尾进行补全，默认为2. |
| number(p,s) | 数值类型 | number(p) 表示整数，长度为p；  number(p,s)表示一个小数，总长度为p，保留小数s位，整数为p-s位 |
| int/integer | 长整型 | 等同于number(38) |
| float(b) | 浮点数类型 | b表示二进制进度，b=[1,126],默认126. |
| date | 日期类型 | 世纪、年、月、日、时、分、秒 |
| clob | lob类型 | 存放大量字符数据,可存4G |
| blob | lob类型 | 存放较大可变长度的二进制对象，如：图形，视频，音频，可存4G |
| long | 字符类型 | 可以支持长达2G 的字符串，常用于添加备注 |
| raw | 二进制类型 | 可以存储长不过255 个字节的二进制代码 |

* + 1. **定义基本表（create table）**

CREATE TABLE <表名>

(<列名><数据类型>[列级完整性约束],

<列名><数据类型>[列级完整性约束],

……,

[表级完整性约束])

[参数设置];

例：① 建立一个学生表student.

CREATE TABLE student

(sno varchar2(9) PRIMARY KEY, # 列级完整性约束，主键。

name varchar2(20) UNIQUE, #列级完整性约束，唯一值。

sex char(2),

age number(2),

dept varchar2(20))

TABLESPACE users; # 表存储的表空间

② 建立课程表course

CREATE TABLE course

(cno varchar2(4) PRIMARY KEY, # 主键

cname varchar2(40) NOT NULL, # 非空字段

cpno varchar2(4), # 先行课，与cno存在联系

credit int,

FOREIGN KEY(cpno)REFERENCES course(cno))

TABLESPACE users;

# 表级完整性约束，cpno是外码，参照表是course，被参照列是cno。

注：参照表与被参照表可以是同一个表内联系，也可以是不同表之间联系。

③ 建立选课表SC

CREATE TABLE SC

(sno varchar2(9),

cno varchar2(4),

grade number(3),

PRIMARY KEY(sno,cno), # 主码由两个属性组成，必须定义表级完整性约束

FOREIGN KEY(sno) REFERENCES student(sno),

FOREIGN KEY(cno) REFERENCES course(cno))

TABLESPACE users;

* 常见表参数

表参数作用：通过参数指定表的存储位置、存储空间分配等。（表设计）

TABLESPACE：用于指定表存储的表空间。若指定，默认为当前用户默认表空间。

STORAGE：用于设置表的存储参数。若不指定，继承表空间的存储参数设置。

LOGGING、NOLOGGING：指明表的创建过程是否写入重做日志文件，默认LOGGING。

CACHE、NOCACHE：指明表中数据是否缓存，默认为CACHE。

NOPARALLEL(默认)、PARALLEL：指明是否允许并行创建表以及随后对表中数据进行并行操作。

并行执行：Oracle企业版才有的特性，指能够将一个大型串行任务(任何DML，或者一般DDL)物理的划分为多个较小的部分，这些较小的部分可以同时得到处理。

* 利用子查询创建表

CREATE TABLE <表名>

(<列名>[列级完整性约束], # 列名不指定类型，此列名相当于对子查询中的字段名重命名

……,

[表级完整性约束])

[参数设置]

**AS** <子查询块>;

例：用已有基本表student创建新的new\_student。

CREATE TABLE new\_student

(new\_sno PRIMARY KEY, new\_name UNIQUE, new\_sex, new\_age, new\_dept)

AS SELECT \* FROM student WHERE age>=20;

注：① 该方法可以修改表中列的名称，但是不能修改列类型和长度；

② 源表中的约束条件和列的默认值都不会复制到新表；

③ 子查询不能包含LOB类型和LONG类型；

④ 当子查询条件为真时，新表中包含查询到的数据；否则创建一个空表。

* + 1. **修改基本表（alter table）**

作用：主要用于维护数据表。

① 添加列；

② 修改列的数据类型与名称；

③ 将列设置为不可用；

④ 删除列；

⑤ 修改表名称。

ALTER TABLE <表名>

[ADD <新列名> <类型> [列级完整性约束]] # 添加新列

[MODIFY <列名> <类型>] # 修改列数据类型

[SET UNUSED COLUMN <列名>] # 标记为不可用列

[SET UNUSED COLUMNS (<列名1>,<列名2>,…)] # 标记多列为不可用列

[DROP COLUMN <列名>] # 删除列

[DROP (<列名1>,<列名2>,…)] # 删除多列

[DROP UNUSED COLUMNS] # 删除被标记的不可用列

[RENAME COLUMN <旧表名> TO <新表名>] # 修改表名

[RENAME TO <新表名>] # 修改表名

* **添加列**  （ADD <列名> <类型> [约束]）

ALTER TABLE student ADD class varchar2(10) not null;

* **修改列类型**  （modify <列名> <类型>）

ALTER TABLE student MODIFY age varchar2(2);

* **设置unused不可用状态** （set unused column <列名>）

ALTER TABLE student SET UNUSED COLUMN class; # 将class标记为不可用列

ALTER TABLE student SET UNUSED COLUMN (class,dept);

不可用列：表中1个或多个列无法再被程序利用的列。

场景：当删除多个列太过频繁或时间消耗大时可以将其标记为不可用状态。

原理：该操作并不是移除列数据或回收这些列占用的空间。一个不可用列不会在查询或数据字典视图中显示，其列名被删除以至于新增列可以重使用其列名。

* **删除不可用列** （drop unused culumns）

ALTER TABLE student DROP UNUSED COLUMNS;

* **删除列** （drop column <列名>）

ALTER TABLE student DROP COLUMN class;

ALTER TABLE student DROP (class,dept);

限制删除（RESTRICT）：restrict表示有限制删除。要删除的基本表不能被其他表的完整性约束所引用（如：foreign key、check等约束），不能有视图，不能有触发器，不能有存储过程或函数等。如果有此依赖关系，则该表不能被删除。

级联删除（CASCADE）：cascade表示没有限制的删除。再删除表的同时，相关的依赖对象也会被删除。

一般默认为限制删除(RESTRICT)。用于删除字句后。

例：ALTER TABLE student DROP COLUMN class CASCADE CONSTRAINTS;

* **修改表名** （rename to <新表名>）

ALTER TABLE student RENAME TO student\_new;

* + 1. **修改约束 （alter table <表名> …）**

**约束**：约束是强加在表上的规则或条件。确保数据库满足业务规则。保证数据的完整性。

注：① 对表进行DML或DDL操作时，如果此操作会造成表中的数据违反约束条件，系统就会拒 绝执行这个操作。

② 定义约束时没有给出约束的名称，Oracle系统将为该系统自动的生成一个名称，其格 式为SYS\_Cn（n为自然数）。

**约束分类**:

① not null（非空）

② unique（唯一）

③ primary key（主键）

④ foreign key（外键）

⑤ check（条件）

⑥ default（默认）

**约束命令规范**：

约束名称命名规则，自定义命名

表 3 约束命名

|  |  |  |
| --- | --- | --- |
| **约束** | **关键字** | **名称规则** |
| 非空约束 | not null | NN\_表名\_列名 |
| 唯一约束 | unique | UK\_表名\_列名 |
| 主键约束 | primary key | PK\_表名 |
| 外键约束 | foreign key | FK\_表名\_列名 |
| 条件约束 | check | CK\_表名\_列名 |
| 默认约束 | default | DF\_表名\_列名 |

约束名称一般在关键字之前使用。

例：CREATE TABLE student (id varchar2(11) CONSTRAINT PK\_id PRIMARY KEY);

ALTER TABLE student ADD CONSTRAINT NN\_student\_name UNIQUE(name);

* **添加主键** （add constraint <约束名> primary key(字段名)）

ALTER TABLE student ADD CONSTRAINT PK\_cno PRIMARY KEY(cno);

* **添加唯一约束**  （add constraint <约束名> unique(字段名)）

ALTER TABLE student ADD CONSTRAINT UK\_student\_name UNIQUE(name);

* **添加外键约束** （add constraint… foreign key … references …）

#（add constraint <约束名> foreign key(字段名) references 参照表名(字段名))

ALTER TABLE SC ADD CONSTRAINT FK\_SC\_sno FOREIGN KEY(sno) REFERENCES student(sno);

* **添加条件(检查)约束** （add constraint <约束名> check(条件)）

① CREATE TABLE student

(sex varchar2(2) CONSTRAINT CK\_student\_sex CHECK(sex=’男’orsex=’女’))；

② ALTER TABLE SC ADD CONSTRAINT CK\_SC\_grade CHECK(grade BETWEEN 0 AND 100);

# 添加条件约束“成绩”字段只能在0-100之间（包含0和100）。

* **添加非空约束**  （modify <字段名> not null）

ALTER TABLE student MODIFY name NOT NULL;

或： ALTER TABLE student MODIFY (name varchar2(10) NOT NULL);

* **添加默认值约束** （modify <字段名> default <默认值>）

① CREATE TABLE student

(age number(2) DEFAULT 20,

sex varchar2(2) DEFAULT ‘男’,

start\_time date DEFAULT SYSDATE); # 系统日期

② ALTER TABLE student MODIFY (dept varchar2(2) DEFAULT ‘MA’);

或：ALTER TABLE student MODIFY dept DEFAULT ‘MA’;

* **删除约束**  （alter table … drop）

指定约束的名称来删除约束，或指定约束的内容来删除内容。

① 指定类型约束

ALTER TABLE student DROP UNIQUE(name);

② 指定名称

ALTER TABLE student DROP CONSTRAINT PK\_sno; # 删除主键sno

③ 级联删除

ALTER TABLE SC DROP CONSTRAINT PK\_sno CASCADE;

* + 1. **删除基本表**

当一个基本表不再被需要时，可以使用drop table进行删除。

例：DROP TABLE student ;

DROP TABLE student CASCADE CONSTRAINTS; # 同时删除通过约束其他表中引用的列

闪回删除：用该方法删除的表，被放入“垃圾桶”。

若想重新找回可以通过PURGE关键字进行找回。

例：DROP TABLE student CASCADE CONSTRAINTS PURGE;

* 1. **创建视图与索引**
     1. **视图**

视图是封装了一条复杂查询的语句。

视图不是具体的表，而是一个“虚表”。视图中的引用表称作基表。

基表发生变化，视图中查询出的数据也会随之改变。

<用来存储常用的查询语句，以方便于下次再次查询同样的语句>

语法： CREATE VIEW <视图名> [(列名1,列名2,…)]

AS <子查询>

[WITH CHECK OPTION];

WITH CHECK OPTION ：表示创建后的视图，对其后续进行增、删、改操作时，需要满足视图中的条 件才能实现这些操作。

查询视图：直接from视图名

* OR REPLACE 创建或覆盖视图

例：CREATE OR REPLACE VIEW stu

AS select name from student;

* WITH CHECK OPTION 满足条件时才能更新

例：① 创建视图

CREAT VIEW stu

AS select name from student where name = ‘王泽轩’

WITH CHECK OPTION;

② 更新视图

update stu set name = ‘屈刘杰’where name = ‘王泽轩’;

* WITH READ ONLY 只读视图，不能修改

例：CREATE VIEW stu

AS select name from student

WITH READ ONLY;

* FORCE/NOFARCE 基表存在/不存在时才能创建视图

例：CREATE FORCE VIEW stu

AS select name from student;

* 删除视图

DROP VIEW stu;

* + 1. **索引**

索引是一种提高数据检索效率的数据库对象，能够为数据的查询提供快捷的存取路径，减少磁盘I/O。虽然索引是基于表建立的，但索引并不依赖于表。索引由系统自动维护和使用。

索引不能随意建立，会消耗系统性能。加快查询速度<一般数据量大时建立索引>

索引分类：

B树索引（平衡树索引）、位图索引。

B树索引分为：唯一索引、组合索引、反向键索引、基于函数索引

语法：

CREATE INDEX <索引名> ON <表名> (列名1[,列名2,…])

基于单个列建立索引时称单列索引，多个时为复合索引

例：① 单列索引

CREATE INDEX stu\_index ON student(name);

② 复合索引

CREATE INDEX stu\_index ON student(name,age);

注：复合索引是有顺序的，查询时需要按照建立索引时顺序查询

建立复合索引时，查询：

① select \* from student where name = ‘许永琪’and age = ‘23’;

# 查询时系统自动使用索引，条件中顺序正确。

② select \* from student where age = ‘23’and name = ‘许永琪’;

#查询时系统不会使用索引，条件没有按照创建索引的顺序。

索引不需要主动使用，在查询时自动使用索引加快查询

* 1. **PL/SQL语言基础**

过程化开发语言。

* + 1. **程序语法**

declare

声明部分 ;

begin

执行部分;

exception

异常处理;

end;

完整的pl/sql语句由4块组成；begin与end是必须存在的部分，其余两部分可选。

声明部分：用于声明变量、常量、数据类型、游标、异常处理名称和本地(局部)子程序定义等。

执行部分：通过变量赋值、流程控制、数据查询、数据操纵、数据定义等操作实现块的功能。

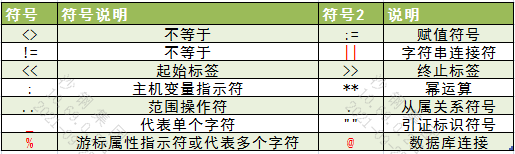
异常处理：用于处理该块执行过程中产生的异常。

**<输出函数：DBMS\_OUTPUT.PUT\_LINE( )>**

* + 1. **词法单元**
* 标识符

注意点：如果标识符区分大小写、使用关键字、包含空格等特殊符号，使用””将标识符括起来，此标识符称为引证标识符。例：”my book”，” exception”。

* 分隔符



补充：

&：替换变量符。 例：var\_1 := &num # 变量var\_1的值在执行程序时输入。

* + 1. **数据类型、常量、变量定义**
* 数据类型

数据类型就是oracle中建表时的数据类型如（char，varchar2，date，number，booleen，long）

* 定义变量

语法：

变量名 [constant] 数据类型 [not null] [default|:= 常量]

例： ① var\_1 varchar2(10);

② num\_1 number(2) not null :=12;

③ num\_3 number(9) := 11;

④ num\_3 constant number(4) default 100;

与创建表结构时基本一样。

注：① 每行只能定义一个变量；

② 如果加上关键字constant，则表示此定义为一个常量，必须为其赋值；

③ 使用not null关键字必须为变量赋初值；

④ 使用default或 := 为变量初始化。

* 引用变量

引用表中字段的类型。修改引用列的数据类型间接上修改了变量的类型。

变量名 表名.字段名%type # 只引用类型

例：定义引用型变量并对其赋值输出

declare

name\_1 student.name%type; # 变量name\_1类型引用student中name字段

begin

SELECT name INTO name\_1 FROM student WHERE sno = ‘17100305107’;

# 将查询出来的name赋值（into）name\_1

dbms\_output.put\_line(‘—‘||name\_1||’--’);

end;

输出结果：--屈刘杰—

* begin内也可以这样赋值：

SELECT ‘小张’INTO name\_1 FROM table\_1

# 将字符”小张”赋给name\_1。

# from的表可以是任意表名，只需要保证查询的完整性

* 记录型变量

记录型变量代表一行数据，可理解为数组，记录某个表的一行数据（一条元组/一条记录）

变量名 表名%rowtype

例：定义引用型变量并对其赋值输出

declare

var\_1 student%rowtype;

begin

SELECT \* INTO name\_1 FROM student WHERE sno = ‘17100305107’;

dbms\_output.put\_line(name\_1.sno||’，’||name\_1.name);

# 注意输出时用变量名.字段名

end;

输出结果： 17100305107，屈刘杰

* + 1. **pl/sql中的sql语句**
* select查询语句

主要用于对引用型变量与记录型变量进行赋值操作。

语法：

SELECT 字段名/\* INTO 变量名 FROM 表名 WHERE 条件表达式

注：① 当为单个字段名时，是对引用型变量赋值，否则对记录型变量赋值。

② select … into 语句只能查询一个数据或一条记录，如果没有查询到数据，会产生 NO\_DATA\_FOUND 异常；如果查询多条记录，则产生TOO\_MANY\_ROWS异常

③ into后的变量用于接收查询的结果，变量个数、顺序要与查询结果相匹配，记录型变 量例外。

实例可参照引用型变量与记录型变量标题

* 数据操纵语言

pl/sql中的DML语句对sql语句中的DML语句进行了扩展，允许使用变量

例：

declare

var\_1 student.sno%type := ‘17100305105’; # 默认

begin

INSERT INTO student(sno,name,sex,age,dept)

VALUES(var\_1,’刘振’,’男’,’23’,’应用数学’);

UPDATE student SET sex = ‘女’WHERE sno = var\_1;

DELETE FROM student WHERE sno = var\_1;

end;

* + 1. **控制结构**

选择结构，循环结构

* if语句

语法：

IF 条件表达式1 THEN 执行语句1;

[ELSIF 条件表达式2 THEN 执行语句2;]

……

[ELSE 执行语句n;]

END IF;

# 执行语句n ：当所有条件表达式都返回false时才执行。

* case 语句

语法：

CASE

WHEN 条件表达式1 THEN 执行语句1;

WHEN 条件表达式2 THEN 执行语句2;

……

[ELSE 执行语句n;]

END CASE;

与if语句意思一致。

* 简单循环

此循环是将循环跳出条件包含在循环体内的。

语法：

LOOP

循环语句;

EXIT [WHEN 条件]; # 退出条件

end loop;

注：在循环体内一定要包含exit语句，负责程序会进入死循环。

例：1~100累加

① exit when 退出循环

DECLARE

num\_1 NUMBER(10) := 0;

sum\_1 number(10) := 0;

BEGIN

LOOP

num\_1 := num\_1 + 1;

sum\_1 := sum\_1 + num\_1;

EXIT WHEN num\_1 >= 100; # 退出循环条件，也可以放在loop后面

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(sum\_1);

END;

输出结果：5050

② if 退出循环

DECLARE

num\_1 NUMBER(10) := 0;

sum\_1 number(10) := 0;

BEGIN

LOOP

num\_1 := num\_1 + 1;

sum\_1 := sum\_1 + num\_1;

IF num\_1 >= 100 THEN

EXIT;

END IF; # 退出循环条件

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(sum\_1);

END;

* while循环

先判断循环条件，只有满足条件才能进入循环体。

语法：

WHILE 条件表达式 LOOP

循环体;

END LOOP;

例：1~100累加

DECLARE

num\_1 NUMBER(10) := 0;

sum\_1 number(10) := 0;

BEGIN

WHILE num\_1 < 100 LOOP

num\_1 := num\_1 + 1;

sum\_1 := sum\_1 + num\_1;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(sum\_1);

END;

* for循环

在for循环中不需要定义循环变量，系统自动定义一个循环变量，每次循环该变量值自动增1或减1。

循环变量只能在循环体内使用，不能再循环外使用。

适合数值的遍历。

语法：

FOR 循环变量 IN [REVERSE] 下确界..上确界 LOOP

循环体;

END LOOP;

注：① 循环变量会自增自减，不需要进行循环体内部取增量；

② reverse表示循环变量从上确界递减到下确界；

例：1~100累加

DECLARE

sum\_1 NUMBER(10) := 0;

BEGIN

FOR i IN 1..100 LOOP

sum\_1 := sum\_1 + i;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(sum\_1);

end;

* + 1. **游标 cursor**

在pl/sql块内做增删改查操作时，oracle会在内存中分配一个缓冲区，缓冲区中包含了处理过程的必要信息，包括已经处理完的行数、指向被分析行的指针和查询情况下的活动集，即查询语句返回的数据行集。该缓冲区域称为上下文区。游标就是指向该缓冲区的句柄或指针。

简单来讲：

用来存储多条查询数据的一种数据结构（结果集），它有一个指针，从上往下移动(fetch)，从而能够遍历多条记录。

分类：

显式游标：有用户定义、操作，用于处理返回多行数据的select查询。

隐式游标：由系统自动进行操作，用于处理DML语句和返回单行数据的select查询。

* 定义游标

语法：

CURSOR 游标名 [(参数名 数据类型, 参数名 数据类型,)] IS 查询语句

注：① 定义位置：在声明部分进行定义

② 游标定义时可以引用pl/sql变量，但变量必须在定义游标前定义

③ 定义游标没有生成任何数据，只是将定义信息保存在数据字典中

④ 定义游标之后可以使用 游标名%rowtype定义记录型变量。

* 游标使用步骤
* 打开游标： OPEN 游标名
* 取一行游标值： FETCH 游标名 INTO 变量名
* 关闭游标： CLOSE 游标名
* 游标结束方式： exit when 游标名%notfound # 退出循环时使用

注：① 只有打开游标才能使用游标

② 游标定义中的变量值发生变化，则只能在下次打开游标时才能起到作用

③ 游标指针第一次fetch时，指向数据集的第一条元组(记录)，fetch后游标指针移动到 下一条记录

④ 游标指针只能向下移动，不能回退。如果想回退，关闭游标重新打开

⑤ into字句变量个数、顺序、数据类型必须与数据集中每个记录的字段数量、顺序、数 据类型一致。

* 显式游标属性

利用这些游标属性可以判断游标状态。（部分可用在循环遍历游标时的退出循环条件）

① %isopen：布尔型。用于检查游标是否打开，打开返回true。

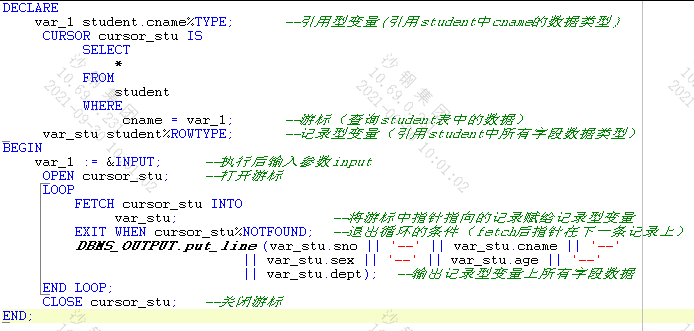
② %found：布尔型。判断最近一次使用fetch时是否从缓冲区（数据集）内检索到数据，检索 到数据返回true。 <可作while循环的退出条件>

③ %notfound：布尔型。判断最近一次使用fetch时是否从缓冲区内检索到数据，未检索到数据返回true。 <可作普通循环的退出条件>

④ %rowcount：数值型。返回到目前为止从游标缓冲区检索的记录个数（数据集行数）。

⑤ %bulk\_rowcount(i)：数值型。用于取得forall语句执行批绑定操作时第i个元素所影响的 行数。

例：将student表中的数据赋给一个记录型变量，并输出变量值。



* for循环遍历游标

利用for循环检索游标时，系统会自动打开、检索和关闭游标。

语法：

DECLARE

CURSOR 游标名 IS 查询语句;

BEGIN

FOR 循环变量 IN 游标名LOOP

循环体;

END LOOP;

END;

利用for循环检索游标时，系统首先会隐含的定义一个数据类型为记录型变量的循环变量，然后自动打开游标，从游标缓冲区中提取数据并放到循环变量中，同时进行%found属性检查以确定是否检索到数据。当遍历完所有数据记录时循环中断，系统自动关闭游标。

例：遍历student表数据

declare

cursor var\_cur IS

SELECT \* FROM student; --定义游标

BEGIN

FOR i IN var\_cur LOOP

dbms\_output.put\_line(i.sno || '--' || i.sname || '--' || i.sex || '--'

|| i.age || '--' || i.dept);

END LOOP;

END;

相当于： 利用子查询

BEGIN

FOR i IN (SELECT \* FROM student) LOOP --相当于将游标直接放在in后

dbms\_output.put\_line(i.cno || '--' || i.cname || '--' || i.sex || '--'

|| i.age || '--' || i.dept);

END LOOP;

END;

* 隐式游标

显式游标用于处理返回多行数据的查询语句，但所有的sql语句都有一个执行的缓冲区，隐式游标就是指向该缓冲区的指针，由系统自动打开、处理、关闭，隐式游标又称SQL游标。

隐式游标主要处理数据操纵语言（DML）以及单行的select … into语句，没有open、fetch、close等操作命令。

与显示游标类似，有：

① sql%isopen：布尔型。判断游标是否打开。对用户来说，该属性值始终是FALSE，因为在操 作时系统自动打开，操作完成后自动关闭。

②sql%found：布尔型。判断当前操作是否会对数据库产生影响。如果有数据的插入、删除、修 改或查询到数据，则返回true，否则返回false。

③ sql%notfound：布尔值。判断当前操作是否会对数据库产生影响。如果没有数据的插入、删 除、修改或查询到数据，则返回true，否则返回false。

④ sql%rowcount：数值型。返回当前操作所涉及数据库中的行数。

例：修改学号为17100305107的学生年龄，将年龄增加10岁。如果表内不存在该学生，则向表 内添加该学生信息。

BEGIN

UPDATE student SET

age = age + 10

WHERE

sno =‘17100305107’;

IF sql%notfound THEN --隐式属性用作if的条件表达式

INSERT INTO student(sno, sname, sex, age, dept)

values('17100305107'

, '屈刘杰'

, '男'

, 22

, '数学与应用数学');

END IF;

END;

* 游标变量（动态游标）

显示游标在定义时与特定的查询进行绑定，其数据集是固定不变的，因此被称为静态游标。游标变量是一个指向多行查询结果集指针，不与特定的查询进行绑定，可以在打开游标的时候定义查询，即可返回不同结构的数据集。

使用游标变量的步骤：

* **定义游标引用类型**

语法：

TYPE 游标类型名 IS REF CURSOR [RETURN 返回类型]

在oracle中系统预定义一个游标引用类型，称SYS\_REFCURSOR，可以直接使用它定义游标变量。

* **声明游标变量**

语法：

游标名 游标类型名

或：游标名 SYS\_REFCURSOR --系统预定义的变量类型

例：① 弱类型游标变量

DECLARE

TYPE cur\_type\_var1 IS REF CURSOR ; --弱类型

cur\_var1 cur\_type\_var1; --相当于实例化（未指定类型）

# cur\_var为游标名，cur\_type\_var1为游标类型名

② 强类型游标变量

DECLARE

TYPE cur\_type\_var2 IS REF CURSOR RETURN student%rowtype; --强类型

cur\_var2 cur\_type\_var2;

# 游标类型为记录型变量student字段类型

③ 系统游标变量

DECLARE

cur\_var3 SYS\_REFCURSOR;

* **打开游标变量**

定义游标变量后，需要在打开游标变量时指定游标变量所对应的查询语句(数据集)。打开游标后，系统将查询结果放在游标变量所指的内存空间内。

语法：

OPEN 变量名 FOR 查询语句

例：① 强类型对应的打开游标方法

OPEN cur\_var1 FOR SELECT \* FROM student;

# 强类型在定义时指定了游标变量的类型，因此查询语句的类型要与游标变量类型一致。

② 弱类型或系统游标变量

OPEN cur\_var2 FOR SELECT \* FROM sc;

* **检索游标变量**

检索方法与静态游标相似。使用fetch … into语句循环检索游标变量结果集的记录。

语法：

LOOP

FETCH 游标名 INTO 变量1[,变量2 …]

EXIT WHEN 游标名%notfound

注：① into后的变量可以是一个记录型变量（类型须一致），也可以是多个一般变量（只能存 放一个数据的变量，注意对应类型）

② 检索游标变量时只能使用简单循环或while循环，不能使用for循环

* **关闭游标变量**

语法：

CLOSE 游标名

例：使用游标变量。输入年龄，查询student表中，大于输入年龄的学生学号和姓名。

DECLARE

TYPE cur\_type IS REF CURSOR; --定义游标类型名

cur\_1 cur\_type; --游标变量

var\_1 number(3);

var\_2 student%ROWTYPE; --记录型变量

BEGIN

var\_1 := &input;

OPEN cur\_1 FOR

SELECT

\*

FROM

student; --打开游标（指定student数据集）

LOOP

FETCH cur\_1 INTO

var\_2; --记录型变量的数据类型与游标类型一致

IF var\_2.age >= var\_1 THEN

dbms\_output.put\_line('年龄大于' || var\_1 || '的学号和姓名' || var\_2.sno || '--'|| var\_2.name);

END IF;

EXIT WHEN cur\_1%NOTFOUND; --退出循环条件

END LOOP;

CLOSE cur\_1;

END;

* + 1. **存储过程**

存储过程通常用于执行特定的操作，不需要返回值。在调用时，存储过程可以作为一个独立的表达式被调用。

存储过程的优点：

① 提高sql执行效率：预编译后存储于内存中，故使用时无需再次编译；

② 减少“网络流量(I/O)”：减少sql与服务器之间的上下文交换

③ 提高系统的“安全性”：减少sql注入的危险、权限控制。

存储过程的缺点：

增加数据库服务器的“负荷”。

* 创建存储过程

CREATE [OR REPLACE]PROCEDURE 过程名[(参数名 IN/OUT 数据类型),…]

AS/IS # is与as等同

声明变量;

BEGIN

程序体;

EXCEPTION

异常程序体;

END [过程名];

* 参数说明

存储过程参数模式有三种，分别是：in、out、in out

① in：（默认参数模式）表示当存储过程被调用时，实参值被传递给形参；在存储过程中，形 参起常量作用，只能读不能修改该参数；当存储过程调用结束返回调用环境时，实参没有被改 变。in模式参数可以是常量或表达式。

调用后，一般在程序体内输出值会使用dbms\_output.put\_line进行输出。

② out：表示当存储过程被调用时，实参值被忽略；在存储过程内，形参起未初始化的pl/sql 变量的作用，初始值为null，可以进行读/写操作；当存储过程调用结束返回调用环境时，形 参被赋给实参。out模式参数只能是变量，不能是表达式。（只能在过程内部赋值）

③ in out：表示当存储过程被调用时，实参值被传递给形参；在存储过程内，形参起已初始化 的pl/sql变量的作用，可读可写；当存储过程调用结束返回调用环境时，形参值被赋给实参。 in out模式参数只能是变量，不能使常量或表达式。

* 调用存储过程

out、in out模式参数在存储过程调用结束时将形参的值赋给实参，因此实参必须是变量，而不能是常量，但是对应于in模式的实参可以是常量，也可以是变量。

语法：

CALL 存储名(实参值);

存储过程可以作为一个独立的表达式被调用。

在pl/sql程序中可以直接被调用:

DECLARE

变量名 变量类型;

BEGIN

存储过程名(实参值/变量名);

程序体;

END;

* + 1. **函数**

函数用于返回特定的数据。在调用时，函数只能作为表达式的一个组成部分被调用

函数的创建与存储过程的创建相似，不同之处是，函数有一个显式的返回值。

* 创建函数

基本语法：

CREATE [OR REPLACE]PROCEDURE 函数名[(参数名 IN/OUT 数据类型),…]

RETURN 返回值类型

AS/IS # is与as等同

声明变量;

BEGIN

程序体;

return 变量名;

EXCEPTION

异常程序体;

END [函数名];

注意：

① 定义函数时，在参数列表后，必须包含一个return语句来指明函数返回值的类型，但 不能约束返回值的长度、精度、刻度等。如果使用%type引用类型，则可以隐含的包括长 度、精度等约束信息

② 函数中可以有多个return，但最终只会有一个return被执行。

③ 定义返回值类型后，在begin中要返回具体哪个变量。

* 调用函数

可以在SQL语句中调用函数，也可以在PL/SQL程序内调用。

① SELECT语句的目标列

② where和having字句

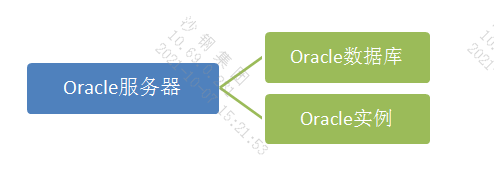
③ connect by，start with，order by，group by字句

④ insert语句的values字句中

⑤ update语句的set字句中

* 1. **oracle体系结构**

指数据库的组成、工作过程与原理，以及数据在数据库中的组织与管理机制。



**oracle数据库** 是一个数据的集合，该集合被视为一个逻辑单元。

**oracle实例** 是管理器的后台进程和内存结构的集合。

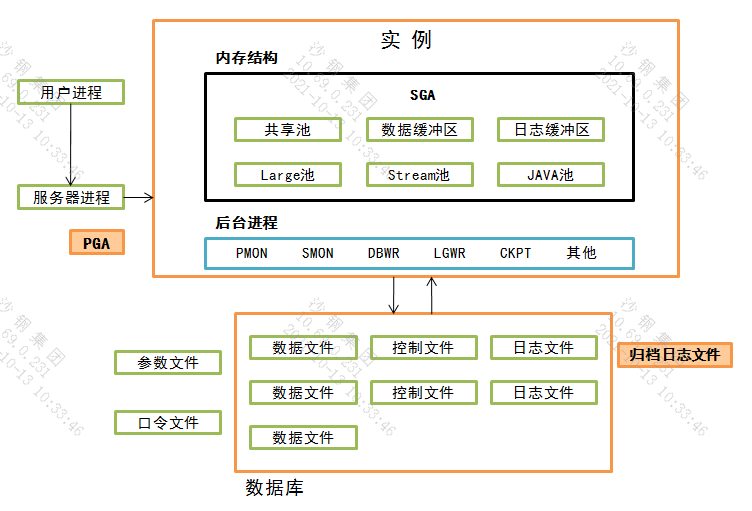


图 6 oracle主要组件

* + 1. **oracle实例**

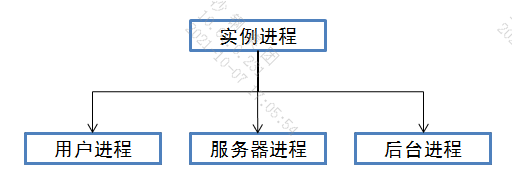
oracle实例是内存结构和后台进程的集合。

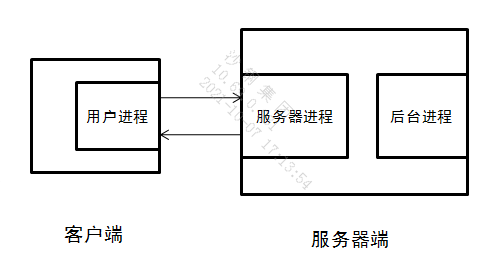
启动oracle实例的过程，即：分配内存、启动后台进程

oracle实例的内存包括SGA和PGA

oracle实例的后台进程包括SMON、PMON、DBWR、LGWR、ARC、CKPT等。

oracle实例进程有三种不同类型的进程。

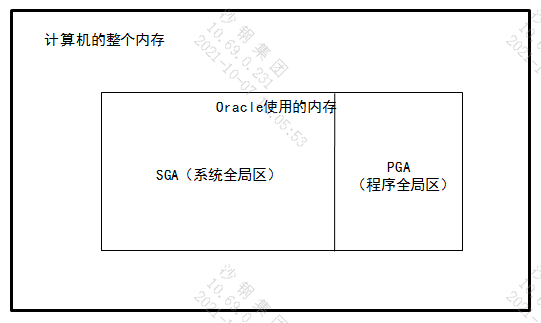




**用户进程**：是一个需要与oracle服务器进行交互的程序。当用户运行一个应用程序准备向数据库服务器发送请求时，即创建了用户进程。<位于客户端>

**服务器进程**：用于处理连接到该实例的用户进程的请求。当用户连接至oracle数据库实例创建会话时，即产生服务器进程。

SGA与PGA的关系：



实例启动时会分配系统全局区(SGA)。

服务器进程启动时分配程序全局区(PGA)

* **SGA**

数据库信息存储在SGA，由多个数据库进程共享。（所有用户共享）

**共享池**：对SQL、PL/SQL程序进行语法分析、编译、执行的内存区域（编译后生成一个执行计 划，最后执行该计划）。

注：重复执行完全相同的sql、pl/sql语句时会直接使用共享池中第一次执行的执行计划。 因此，共享池大小直接影响数据库性能（执行计划会储存的越多）。

**数据缓冲区**：用于存储从磁盘数据文件中读入的数据，所有用户共享。

用户发出命令时，会提取硬盘上的数据表文件，硬盘取出后先放到数据缓冲区，在从数据 缓冲区中返回给用户。优点：下次再读取时直接从数据缓冲区返回给用户（硬盘直接返回给用 户速度很慢）。<数据缓冲区大小直接影响数据库读取速度>

即： 用户 🡨-🡪 数据缓冲区 🡨-🡪 硬盘数据

**日志缓冲区**：日志纪录数据库修改的所有信息，这些信息直接产生与日志缓冲区。

当日志缓冲区的日志数据达到一定的数量时，后台进程将日志数据写入到在线日志文件中。

**Large池**：为了进行大的后台进程操作而分配的内存空间，主要指备份恢复、大型IO操作、并 行查询等。

**Stream池（流）**：为了stream应用而分配的内存空间。<高级复制技术>

**JAVA池**：为了Java应用而分配的内存空间。

* **PGA**

包含单个服务器进程所需的数据和控制信息，比如每个session(连接)传入的SQL绑定变 量、以及对该session进行控制的控制信息。如果该session进行了排序或hash连接操作，也 会使用PGA中的内存。

PGA是在用户进程连接到数据库并创建一个会话时自动分配的，保存每个与Oracle数据库 连接的用户进程所需的信息。（采用专用连接，多个用户进程（客户端）一一对应多个服务进程（服务端））

服务器进程是占用内存空间的，在PGA中，专用连接PGA不能共享给其他用户。

Oracle数据库采用的时专用连接，共享连接基本不采用。

<对于专用连接而言，用户在客户端启动了一个应用进程，例如sql&plus，就是在客户端启动一个用户进程：与Oracle服务器端连接成功后，会在服务器端生成一个服务器进程，该服务器进程作为用户进程的代理进程，代替客户端执行各种命令并把结果返回给客户端。

用户进程一旦中止，服务器进程立刻中止>

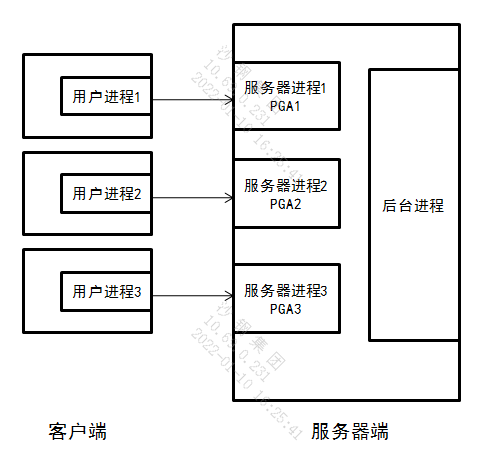


图 7 专用连接

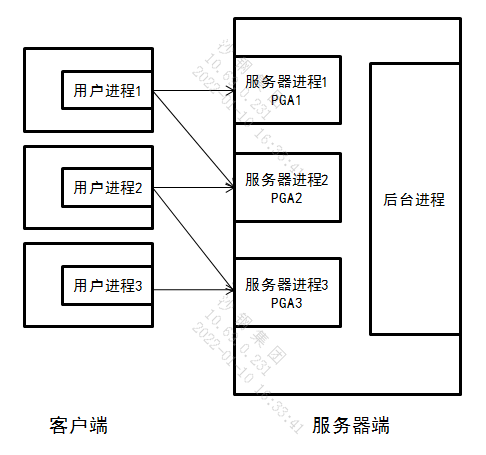


图 8 共享连接

* **后台进程**

后台进程是Oracle数据库为了保持最佳系统性能和协调多个用户请求而设置的。Oracle实例启动时即创建一系列后台进程

**PMOM：**进程监控进程

清理出现故障的进程；

释放所有当前挂起的锁定；

释放故障进程使用的资源。

<例：当用户进程中止，服务器进程则被清理，对应该服务器进程的PGA被回收>

**SMON：**系统监控进程

在实例失败之后，重新打开数据库时自动恢复实例；

整理数据文件的自由空间，将相邻区域结合起来；

释放不再使用的临时段。

<对整个Oracle的运行进行监控>

**DBWR：**数据写入进程

管理数据缓冲区，将最近使用过的块保留在内存中；

将修改后的缓冲区数据写入数据文件中。

<修改数据时，先将内存中数据块取出到数据缓冲区内，在缓冲区内进行修改，一段时间后 DBWR将修改后的数据块返回到内存中>

**LGWR：**日志写入进程

负责将日志缓冲区中的日志数据写入日志文件；

系统有多个日志文件，该进程以循环的方式将数据写入文件。

<把日志缓冲区中的日志写入到在线日志文件中>

**CKPT：**检查点进程

为了防止实例崩溃，为了尽快进行实例恢复的进程。

检查点包括两种：完全检查点、增量检查点

完全检查点有两种情况被触发：

① 用户发出alter system checkpoint；

② 除了shutdown abort以外的其他方式正常关闭数据库

增量检查点触发情况：

① 每隔三秒

② 在线日志切换

* **会话**

会话是用户与Oracle服务器的单个连接。

当用户与服务器建立连接时创建会话。

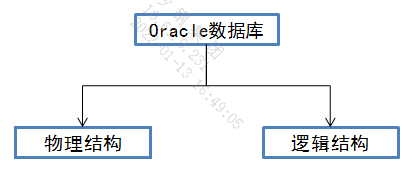
当用户与服务器断开连接时关闭会话。

从v$session中查询会话信息。

* + 1. **Oracle数据库**

oracle数据库由操作系统文件组成，这些文件为数据库信息提供实际物理存储区。

oracle数据库包括逻辑结构与物理结构。



物理结构包含数据库中的一组操作系统文件。

逻辑结构指数据库创建之后形成的逻辑概念之间的关系。

* 物理结构

主要物理文件：

① 数据文件：数据文件用于存储数据库数据，如表、索引数据等；

② 控制文件：控制文件是记录数据库物理结构的二进制文件；

③ 在线日志文件：记录对数据库的所有修改信息，用于故障恢复。

主要物理文件丢失，数据库不可以继续运行，需要进行介质恢复。

非主要物理文件：

参数文件：pfile静态参数文件和spfile动态参数文件；

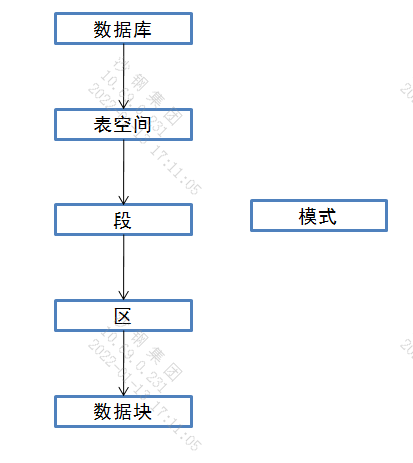
密码文件：sys用户的密码；

告警和跟踪文件、归档日志文件、备份文件

非主要物理文件丢失，数据库仍然可以运行，不需要介质恢复。

* 逻辑结构

从逻辑角度分析数据库的组成。逻辑组件包括：



表空间是数据库中最大的逻辑单位，一个Oracle数据库至少包含一个表空间，就是名为system的系统表空间；（system损坏，Oracle无法运行）

每个表空间是由一个或多个数据文件组成的，一个数据文件只能与一个表空间相关联；

表空间的大小等于构成该表空间的所有数据文件大小之和。

（temp是临时表空间，当排序不能在分配的空间中完成时，就会使用磁盘排序的方法，即Oracle实例中的临时表空间中进行；

users是数据库默认永久表空间）

* 1. **表分区**

把一个大的物理表分成若干个小的物理表，并使得这些小物理表在逻辑上可以被当成一张表来使用。

主表/父表：该表是创建子表的模板。它是一个正常的普通表，但正常情况下不存储任何数据。

子表/分区表：这些表继承并属于一个主表。子表中存储所有的数据。主表与分区表属于一对多的关系，而一个分区表只从属于一个主表。

分区方法：范围分区、散列分区、列表分区、复合分区

* + 1. **范围分区**

以表中的一个列或一组列的值的范围进行分区。

语法：

partition by range(列名)

(

Partition 分区表名1 value less than(值1),

Partition分区表名2 value less than(值2),

……

[Partition分区表名n value less than(maxvalue)]

);

注：”值1”的意思是：若数据小于值1则进入分区1。

maxvalue指分区条件是小于最大值且大于其余分区条件值的数据。

每个分区的范围不能重叠。

查询分区信息：

Select \* from sales partition(分区名);

添加分区：

Alter table sales add partition 分区名 values less than(值);

注：分区范围不能重叠。

* + 1. **散列分区**

允许用户对不具有逻辑范围的数据进行分区

通过在分区键上执行HASH函数决定存储的分区

将数据平均地分布到不同的分区

语法：

Partition by hash(列名)

(

Partition 分区名1,

partition 分区名2,

……

partition 分区名n

)

注：数据进入到不同分区是按照hash算法进入的。

* + 1. **列表分区**

允许用户将不相关的数据组织在一起

语法：

Partition by list(列名)

(

Partition 分区名1 values (值11,值12),

Partition 分区名2 values (值21，值22),

……

Partition 分区名n values (值)

)

注：values中可以有多个值作为条件分区。

* + 1. **复合分区**

范围分区与散列分区或列表分区的组合。即，(范围+散列)或(范围+列表)

11g新增表分区类型如下：

引用分区，

* + 1. **引用分区**

基于由外键引用的父表的分区的方法，它依赖已有的父表子表关系，子表通过外键关联到父表进而继承了父表的分区方式而不需自己创建，子表还继承了父表的维护操作。

注：父表必须分区，可以是范围分区、散列分区、列表分区。子表通过外键继承后直接引用父表。

* + 1. **间隔分区**

可以完全自动的根据间隔阈值创建范围分区，它是范围分区的扩展。

eg：例如通过日期/时间间隔扩展分区。定义初始分区日期，指定间隔长度(步长)，当插入数据时，按照数据日期字段进行增加分区。

* + 1. **基于虚拟列的分区**

把分区建立在某个虚拟列上，即建立在函数或表达式的结果上

同义词

* 1. **序列**

Oracle提供的用于产生唯一系列唯一数字的数据库对象。

* 作用：

自动提供唯一的数值；

共享对象；

主要用于提供主键值；

将序列值装入内存可以提高访问效率

* 语法：

create sequence 序列名

start with n // 从n开始

increment by n // 步长为n

maxvalue n // 序列最大值为n

minvalue n // 序列最小值为n

nocycle // 在达到最大值后停止生成下一个值（不循环）

cache n; // 内存中预先分配的序列数

注：创建序列时，用户必须拥有相应权限（create sequence 或create any sequence）

* 使用序列：

nextval：返回序列中下一个有效值，任何用户都可以使用

currval：存放序列的当前值。

注：nextval应该在currval之前指定，两者应该同时有效。

* 访问序列：

序列名.nextval // 在insert/select时实现主键字段自增。

* 序列不连续原因：

回滚；系统异常；多表同时使用同一序列

* 修改序列：

使用alter sequence语句修改序列，不能更改start with参数

eg：alter sequence 序列名 maxvalue 值 [其他参数];

* 删除序列：

使用drop sequence语句删除。

* 1. **事务**

用户定义的一个数据库操作序列，这些操作要么全做，要么不做，是一个不可分割的工作单元。

例如：在关系型数据库中，一个事务可以是一条语句、一组sql语句或整个程序。

事务的四个属性，简称ACID属性：

**原子性**：事务是一个完整的操作。事务的各步操作是不可分割的(原子的)；要么都执行，要么都不执行

**一致性**：当事务完成时，数据必须处于一致状态。

**隔离性**：对数据进行修改的所有并发事务是彼此隔离的，这表明事务必须是独立的，他不应以任何方式依赖于或影响其他事务。

**永久性**：事务完成后，它对数据库的修改被永久保持，事务日志能够保持事务的永久性。

**提交**：commit； **回滚**：rollback

**Sql\*puls自动提交**：set autocommit on/off // java中一般默认自动提交

Java中如何避免自动提交？

开启事务setAutoCommit(false)，这样可以避免java自动提交。

**PostgreSQL**

1. 子查询必须as别名。
2. 类型转换：

cast（值 as 数据类型）

3、update与delete时表名不能加别名。

4、pg数据库的查询语句执行顺序

from -> on -> join -> where -> group by -> with -> having -> select -> distinct

-> order by -> limit

5、hive的执行顺序

from -> where -> join -> on -> select -> group by -> select -> having -> distinct -> order by -> limit -> union

// group by是在两个select之间，hive默认开启map端的group by分组，所以是map端的select先执行，在reduce端是group by先执行

# MySql

## 索引的数据结构

### 为什么使用索引

学习网站：https://blog.csdn.net/LXYDSF/article/details/125873790

在没有索引的情况下，数据分布在硬盘不同的位置，在查找数据时，需要在不同的位置来回找数据，非常耗时，即使是顺序放置，也需要按顺序一个一个的读取并确认是否符合我们要查找的数据，也需要进行多次IO操作，依然很耗时。

为了解决这个问题，我们引入索引。

索引就是在磁盘上维护一个数据结构，这个数据结构一般是树，这里假设是二叉搜索树（其实默认索引以B+树的形式存在），以KV键值对的形式存储地址。

从时间复杂度上看，二叉树是O(logn)，顺序查找是O(n)，而O(logn)远小于O(n)，所以大大提高了效率。索引大大减少了磁盘IO次数，加快查询效率。

磁盘IO和算法层面上完全不在一个数量级，算法层面可能O(n^2)需要4ms，改进一下算法变成O(n)需要2ms，这个过程提高了2ms，而磁盘的读取需要的时间远远大于算法层面，读取一次磁盘可能需要200ms，在读取一次又是200ms,这个时间差完全冲散了算法层面的改进，所以减少磁盘IO次数、加快查询效率是非常重要的，这也侧面体现了索引的重要性。

### 索引概述

索引（Index）是帮助MySQL高效获取数据的数据结构

索引是数据结构。你可以简单理解为“排好序的快速查找数据结构”，满足特定查找算法。这些数据结构以某种方式指向数据，这样就可以在这些数据结构的基础上实现高级查找算法。

注意：

索引是在存储引擎中实现的，因此每种存储引擎的索引不一定完全相同，并且每种存储引擎不一定支持所有索引类型。同时，存储引擎可以定义每个表的最大索引数和最大索引长度。所有存储引擎支持每个表至少16个索引，总索引长度至少为256字符。有些存储引擎支持更多的索引数和更大的索引长度。

#### 优点

1、降低数据库IO成本，提高数据检索效率；(主要原因)

2、创建唯一索引，可以保证数据表中每行数据的唯一性；

3、加速表与表之间的连接。对于有依赖关系的子表和父表联合查询时，可以提高查询速度；

4、减少查询中分组和排序的时间 ，降低了CPU的消耗。

#### 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、索引需要占磁盘空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，存储在磁盘上，如果有大量的索引，索引文件就可能比数据文件更快达到最大文件尺寸。

3、虽然索引大大提高了查询速度，同时却会降低了更新表的速度。当对表中的数据进行增加、删除和修改的时候，索引也要动态地维护，这样就降低了数据的维护速度

<当然，如果想要插入记录，可以先将索引删除，然后插入数据，插入完成后再创建索引>

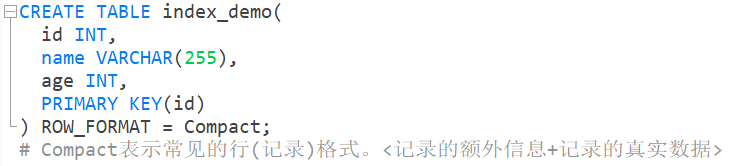
### innodb中的索引

#### 在一个页中的查找或者在很多页中查找

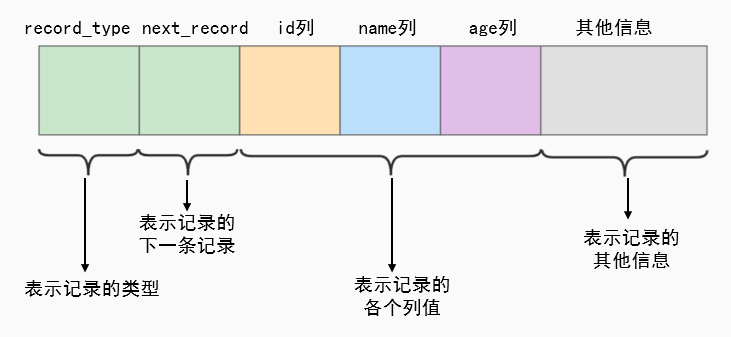
在没有索引的情况下，不论是根据主键列或者其他列的值进行查找，由于我们并不能快速的定位到记录所在的页，所以只能从第一个页沿着双向链表一直往下找，在每一个数据页中根据我们上面的查找方式去查找指定的记录。因为要遍历所有的数据页，所以这种方式显然是超级耗时的。如果一个表有一亿条记录呢？此时索引应运而生。

#### 设计索引

首先新建一个表



简化的index\_demo表的行格式示意图：



详解：

record\_type：记录头信息的一项属性，表示记录的类型，0表示普通记录、2表示最小记录、3表示最大记录、1暂时还没用过。

next\_record：记录头信息的一项属性，表示下一条地址相对于本条记录的地址偏移量，我们用箭头来表明下一条记录是谁。

各个列的值：这里表示id、name、age三个列。

其他信息：除了上述3种信息以外的所有信息，包括其他隐藏列的值以及记录的额外信息。

* 一个简单的索引设计方案

假设单个数据页可以存入3条记录(实际能存入很多)，向数据页10添加3条数据。



3条记录进入数据页后按照主键值的大小串联成一个单向链表。

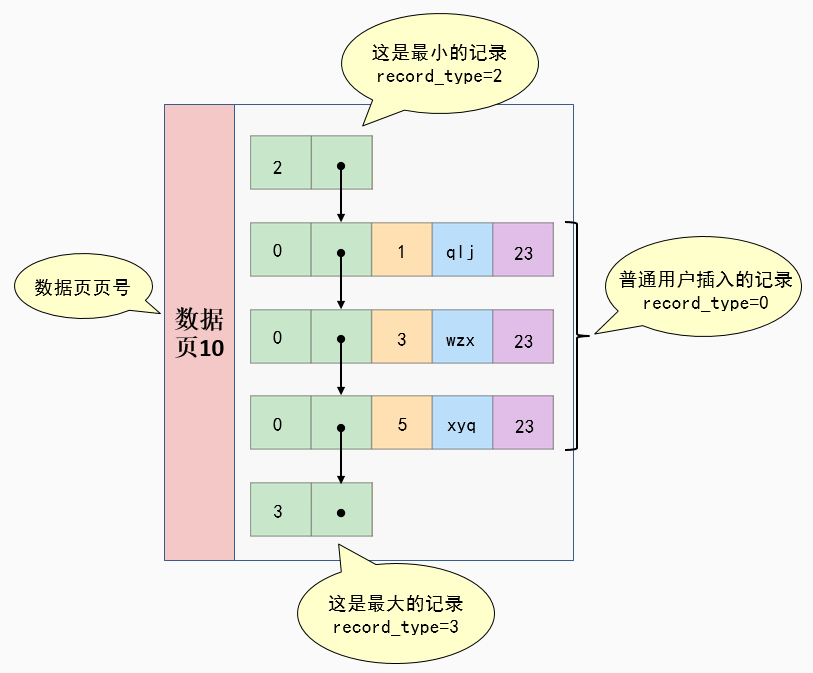


图 9 单数据页存储数据结构

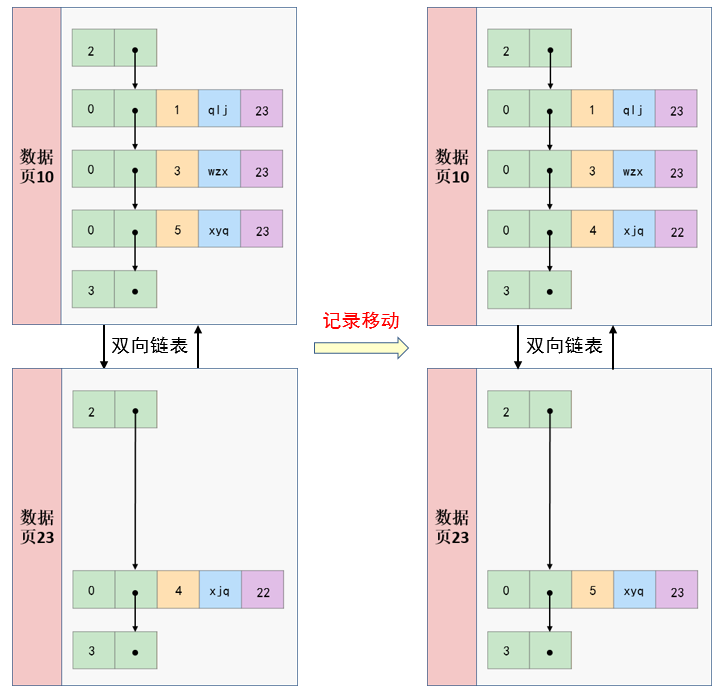
**①** 下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。

再次添加一条记录：INSERT INTO index\_demo VALUES (4, 'xjq', 22);

由于数据页只能存储3条记录，所以开辟新的数据页存储：

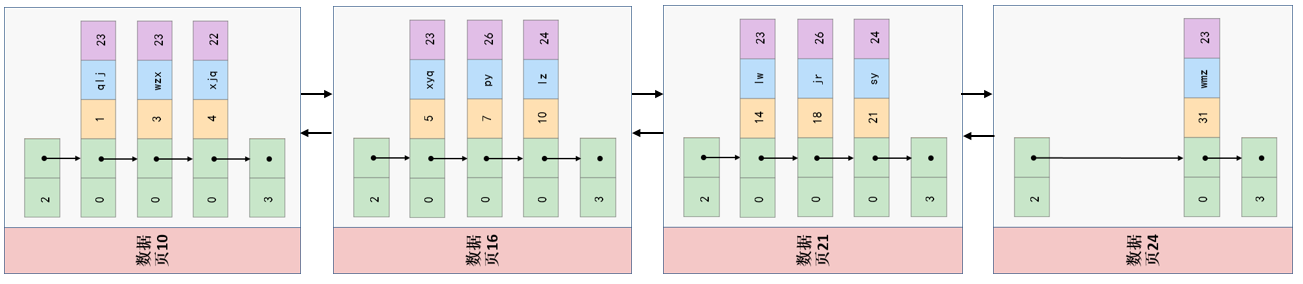
注意：

新分配的数据页编号可能不是连续的。同时可以看出新增的记录主键为4，并没有顺序存储，所以这就不符合下一个数据页的记录主键值大于上一页的记录主键值的要求。所以需要记录移动，也称页分裂。



**②** 给所有的页建立一个目录项

由于数据页的编号并不是连续的，所以向index\_demo表中插入许多条记录后，有下面的效果：



因为这些16kb的数据页在物理存储上是不连续的，所以如果想从这么多页中根据主键快速定位某些记录所在的数据页，我们需要给它们做个目录，每个数据页对应一个目录项，每个目录项包含两项：

-> 数据页记录集最小主键值，用key表示；

-> 页号，用page\_no表示；

建立好目录后，就可以分层/分页快速查找到对应主键的记录。

例：查找主键为18的记录。

直接遍历所有目录项或者二分法查找主键为18的目录区间，找到对应页区间后再进入数据页中遍历查找记录。<目录项组成一个数组，可以二分法查找>

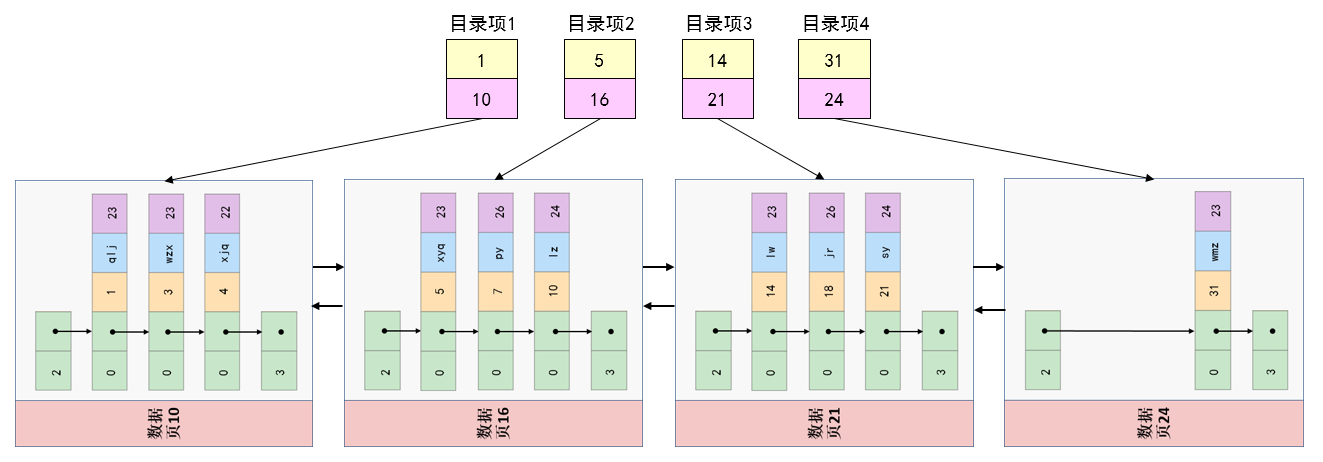


图 10 建立目录项

我们采用上面的方案就可以使用数据页目录项建立一个简单的索引。

* innodb中的索引方案

我们将上面的索引案例进行三次迭代。

第一次迭代：将目录项数组存入目录页。<第一次迭代只生成一个目录页>

目录页与存储数据的数据页异同：

不同点：目录页的记录类型为1；记录行存储的数据不同<目录页存储每个目录项>

相同点：实际都是用的是数据页，都会为主键值生成一个页目录，将主键提取出来组成一个数组，便于使用二分法查找<链表不能二分>

第二次迭代：多个目录项记录的页。<由于记录量较大，一个目录页存不下>

此时第二层多个目录页，每次访问目录页都会产生一次IO交互。依照案例来讲，如果迭代两次，生成两个目录页时，最多需要进行3次IO。分别是：

① 遍历第一个目录页，进行主键区间匹配；

② 遍历第二个目录页；

③ 找到目录索引，直接去数据页进行遍历查找数据。

第三次迭代：目录项记录页的目录页。

当数据量倍增时，目录页也会增多，导致IO次数增多，影响查询效率。再次进行迭代，与第二次迭代实际差不多。

经过多次迭代，生成了一个类似于树的数据结构。这个树形结构就是B+树。

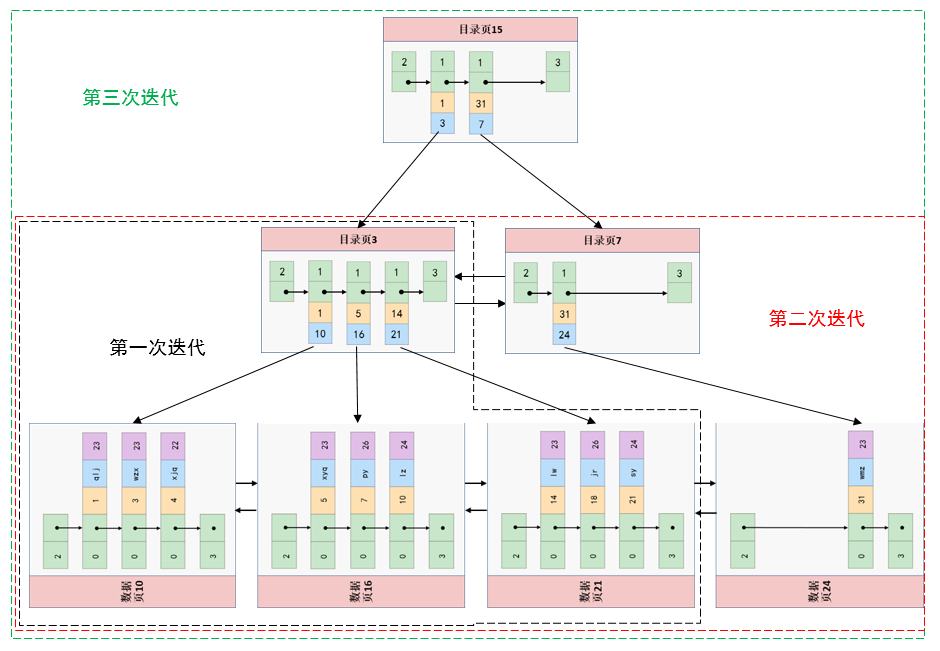


图 11 B+树数据结构

叶子结点存放实际的记录，非叶子结点用来存储目录项。

注意：

为什么在真实环境中我们B+树结构的深度并不是很高？

简单计算一下：

假设一个数据页16kb存放100条数据，目录页存放1000条数据<只有键值>

假设B+树有4层：最多可存放 = 100\*1000\*1000\*1000 = 1000亿数据量

**树的层次越低，IO次数就越少，从而效率越高**

#### 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式(所有的记录都存储在了叶子结点)，也就是所谓的‘索引即数据，数据即索引’。

聚簇表示数据行和相邻的键值聚簇的存储在一起。

特点：

1、使用记录主键值的大小进行记录和页的排序，包含下面三个含义：

① 页内的记录是按照主键的大小顺序排成一个单向链表。

② 各个存放存放用户记录的数据页，也是根据页中用户记录的主键大小顺序排成一个双向链表。

③ 存放目录项记录的目录页分为不同的层次，在同一层次中的目录页也是根据页中目录项记录的主键大小顺序排成一个双向链表。

2、B+树的叶子节点存储的是完整的用户记录

<完整的用户记录表示记录的所有列(包括隐藏列)>

我们把具有这两种特性的B+树称为聚簇索引，所有完整用户记录都存放在这个聚簇索引的叶子节点处。这种聚簇索引并不需要我们在MySQL中显示的使用INDEX语句去创建，innodb存储引擎会自动为我们创建聚簇索引。

优点：

数据访问速度快。<聚簇索引将索引和数据保存在同一个B+树中，所以访问速度快>。

聚簇索引对于主键的排序查找和范围查找速度非常快(因为数据本身就是有序的)。

节省了大量IO操作。

缺点：

① 插入速度严重依赖于插入顺序，按照主键的顺序插入最快，但是如果数据本身插入时主键并不是顺序排列插入的，这就导致构建B+树时需要大量的页分裂(记录移动)。

<所以对于innodb表一般会定义一个自增id为主键>

② 更新主键的代价很高，会导致被更新行进行记录移动。

<所以一般主键为不可更新字段>

③ 二级索引访问需要两次索引查找，第一次找主键，第二次根据主键找记录。

**数据表创建限制补充**：

① 对于MySQL数据库目前只有innodb数据引擎支持聚簇索引，而MyISAM并不支持聚簇索引。

② 由于数据物理存储排序方式只能有一种，所以每个数据表只能有一个聚簇索引。一般情况就是该表的唯一主键。<联合主键是一个主键>

③ 如果表没有定义主键，innodb会选择非空的唯一索引代替。如果没有这样的索引，innodb会隐式的定义一个主键来作为聚簇索引。

④ 为了补充利用聚簇索引的聚簇特性，所以innodb标的主键列尽量选用有序的顺序id，而不建议用无序的id，比如UUID、MD5、Hash、字符串列作为主键无法保证数据的顺序增长。

#### 二级索引(辅助索引、非聚簇索引)

上面的聚簇索引只能根据主键搜索查询时才能发挥索引的作用。因为B+树的索引是按照主键进行排序的。那如果我们想以别的字段为条件进行查询怎么办？

例如：

stu表中，主键是一个自增id，这个id在实际查询中没有查询作用，我们要想使用’学号’进行查找，此时并不能用到主键查找，索引失效。

解决方案：多建几颗B+树，不同的B+树中的数据采用不同的排序规则。

例如：

将主键和学号字段抽取出来，以这两个字段构建新的B+树，这个新的B+树以’学号’为索引，记录只保存学号和id，此时这个B+树相当于一个数据字典，用于查找对应的表级主键。

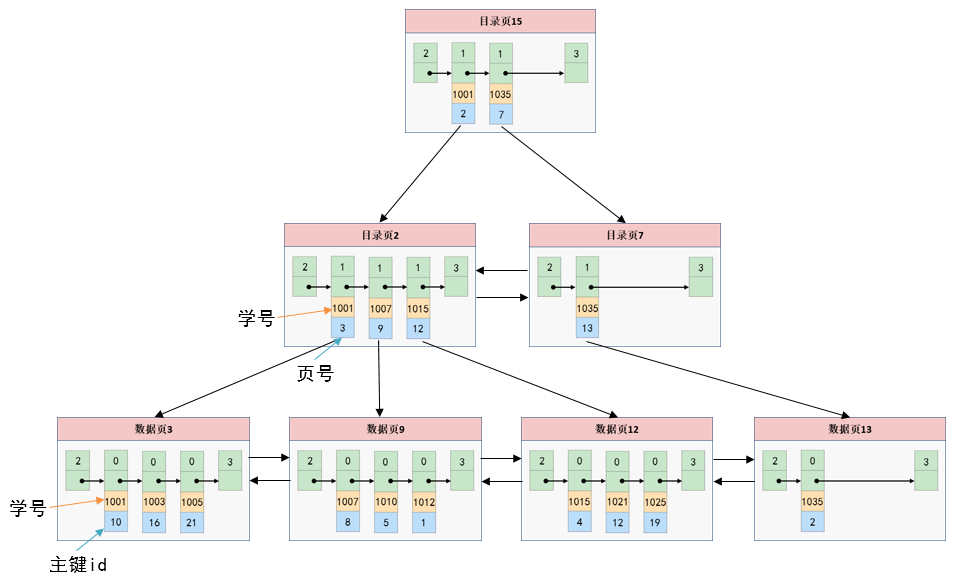


图 12 二级索引

这个B+树与上面聚簇索引的B+树不同之处：

1、使用字段’学号’的大小进行记录和页的排序。

① 页内的记录是按照学号的大小排列为单向链表。

② 存放记录的数据页之间根据页内学号大小排列为双向链表。

③ 存放目录项记录的目录页有多个层次，同层次根据页内学号大小排列为双向链表。

2、B+树的叶子结点存储的记录并不是完整的记录，只是主键+学号列。<数据字典>

3、目录项记录中不再是主键+页号，而是学号+页号。

在执行：select \* from stu where stno=1001;时，首先在二级索引中找到对应的主键值，找到主键值后再通过聚簇索引进行主键查找具体记录。

实际上这个再进入聚簇索引B+树查找称之为回表。

问题：为什么不直接将完整记录存放到这个二级索引中？

解答：如果把完整的记录放到二级索引的叶子结点中是可以不需要回表。但是这样太占地方了，相当于每建立一颗B+树都需要把所有的记录都拷贝一遍，浪费存储空间。

所以：

① 这种**按照非主键列建立**的**B+树**需要一次回表操作才可以定位完整的用户记录，所以被称为二级索引/辅助索引。

② 非聚簇索引的存在不会影响数据在聚簇索引中的组织，索引一张表可以有多个非聚簇索引。

小结：聚簇索引与非聚簇索引(二级索引)的区别

① 聚簇索引的叶子结点存储完整的记录内容，非聚簇索引的叶子结点存储的只是数据位置(这个数据位置指的是数据的主键)，非聚簇索引不会影响数据表的物理存储顺序。

② 一个表只能存在一个聚簇索引，因为只能有一种排序存储的方式。但表可以有多个非聚簇索引，也就是多个索引目录提供数据检索。

③ 使用聚簇索引的时候，数据的查询效率高，但如果进行插入、删除、更新等操作，效率比非聚簇索引低。<聚簇索引中存放的是记录的所有字段，一旦进行修改，聚簇索引上的数据是一定需要修改的。而非聚簇索引存放的是主键+单个字段，相对来说修改的少，效率也就高。>

#### 联合索引

联合索引实际就是一个二级索引(非聚簇索引)。

当查询时使用多条件查询时，此时就可以创建联合索引，同时对多个列进行排序。也就是对多个列建立索引。

例如：对’学号’和’姓名’进行联合索引(假设’学号’不唯一)。此时就要对两列数据进行排列。

先把各个记录和页按照’学号’进行排序；

在记录’学号’列相同的情况下，采用’学号’进行排序。

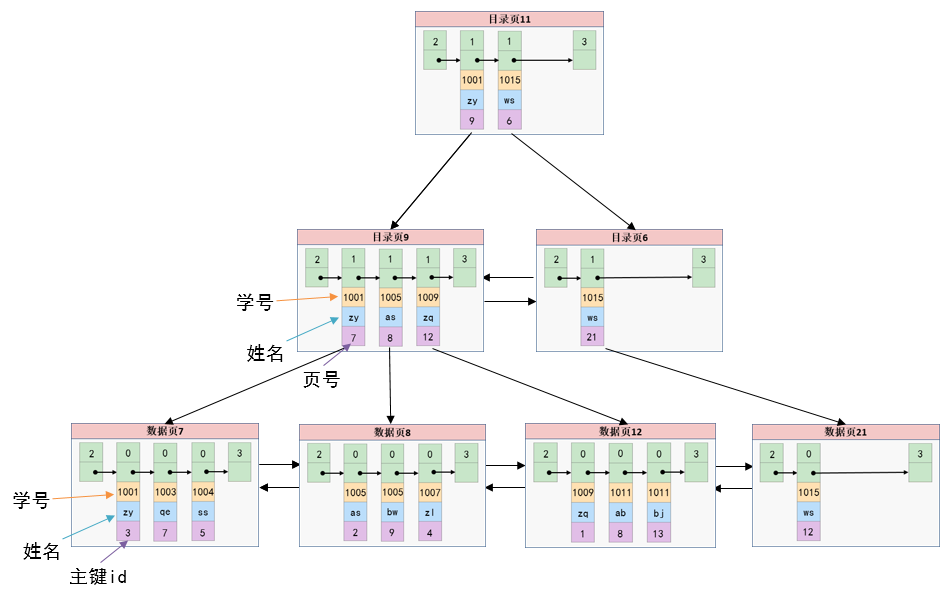


图 13 联合索引

注意：

数据页8与数据页12存在’学号’相同的情况，’学号’相同时会按照’姓名’进行排序。

可以看出本质上还是一个二级索引。

#### innodb中B+树索引的注意事项

**1、**根页面位置始终不动<树先有根>

在设计索引时，我们先将叶子结点的存储结构构建出来，然后才向上继续整合的。然而实际上B+树的构建过程并不是那样反向构建的，而是B+树的构建是自上往下构建的。

实现原理：

① 每当为某个表创建一个B+树索引(聚簇索引不是人为创建的，默认就有)的时候，都会为这个索引创建一个根节点页。最开始表中没有数据时，每个B+树索引对应的根节点中既没有用户记录，也没有目录项记录。<此时根节点页就是一个存储用户记录的数据页>

② 随后向表中插入用户记录时，先把用户记录存储到这个根节点中。

③ 当根节点中的可用空间用完时继续插入记录，此时会将根节点中的所有记录复制到一个新分配的页，比如页a中，然后对这个新页进行页分裂的操作，得到另一个新页，比如页b。这时新插入的记录根据键值(也就是聚簇索引中的主键值，二级索引中对应的索引列的值)的大小就会被分配到页a或者页b中，而根节点便升级为存储目录项记录的目录页。

注意：一个B+树索引的根节点自诞生之日起，便不会再移动。这样只要我们对某个表建立一个索引，那么它的根节点的页号就会被记录到某个地方，然后凡是innodb存储引擎需要用到这个索引的时候，都会从那个固定的地方取出根节点的页号，从而来访问这个索引。

**2、**内节点(目录页)中目录项记录的唯一性

为什么需要保证目录项记录的唯一性？

我们知道目录页的作用就是为了构建一个类似于数据字典的数据页。

对于聚簇索引来讲：目录页的目录项存储’主键+数据页页号’，既然存储了主键，则目录页中的目录项就是唯一的。

对于二级索引来讲：由于二级索引就是为了对非主键字段创建快捷索引，所以二级索引中数据页的数据项存储了’非主键字段+主键’ ，父节点目录页中的目录项则存储着’非主键字段+数据页页号’，此时我们就可以发现，目录页中存储的目录项索引可能不唯一。当目录项不唯一时，数据字典就不能起到作用，向下找不到数据在哪个子节点。

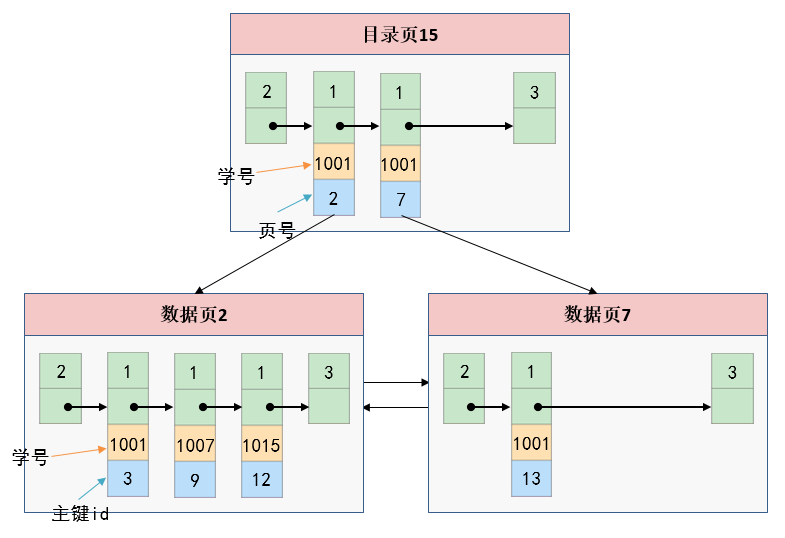
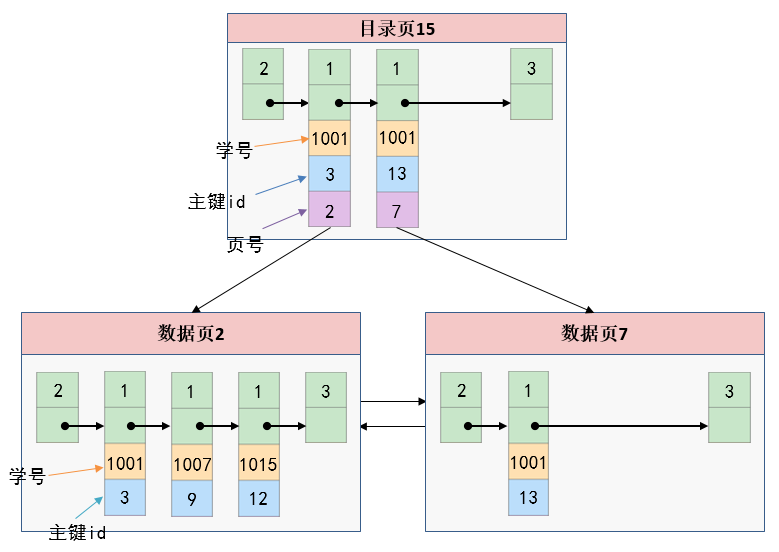


图 14 二级索引目录项不唯一

为了让新插入记录能找到自己在哪个二级索引的数据页中，我们必须保证在B+树的同一层内节点的目录项记录除页号字段以外是唯一的。所以对于二级索引的目录页中的目录项中实际由这三个部分构成：

索引列的值(非主键字段)、主键值、页号



添加了主键的目录页，此时保证了目录项的唯一性。

**3、**一个页面最少存储2条记录

B+树的本质是一个大的多层级目录，每经过一个目录时都会过滤掉许多无效的子目录，直到最后访问存储真实数据的目录。如果一个大目录中只存放一个子目录项，那么此时目录层就非常多，IO访问量就会增加。innodb的一个数据页可以至少存放两条记录。

### MyISAM中的索引

在MySQL中，Innodb和MyISAM默认的索引是B-tree索引；而Memory默认的索引是Hash索引。

<实际上Mysql中的B-Tree还是B+Tree。但是在选择MySQL的数据结构时，区别就很大>

MyISAM引擎使用B+Tree作为索引结构，叶子节点的data域存放的是数据记录的地址(所以MyISAM把索引和数据分开存储了)。

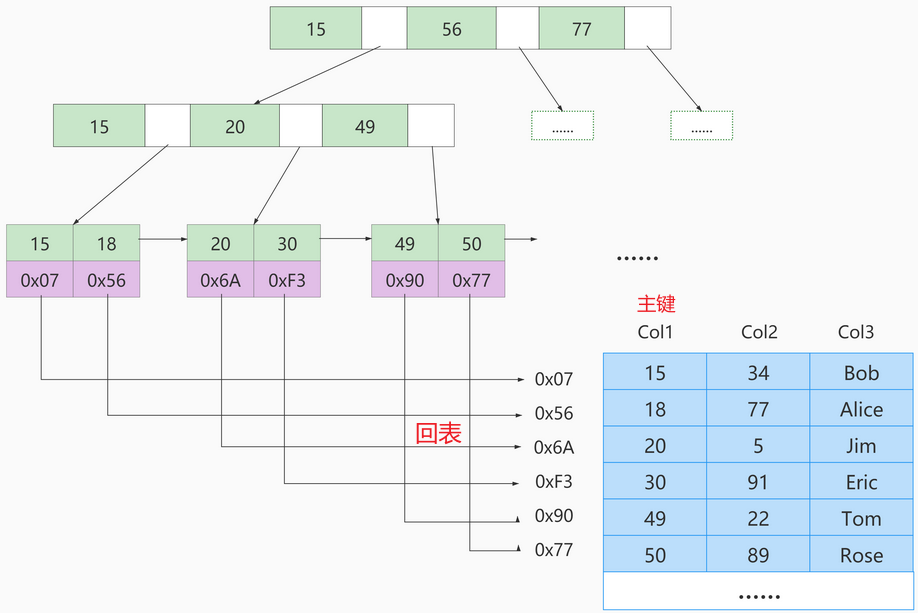
#### MyISAM 索引的原理

我们知道Innodb中索引即数据，也就是聚簇索引的叶子结点中本身就存储了记录。而MyISAM的索引方案虽然也使用了B+树结构，但却与聚簇索引不同，它将索引和数据分开存储。

数据文件+索引文件

将表中的记录按照记录的插入顺序单独存储在一个文件中，称之为数据文件。这个文件并不划分为若干个数据页，有多少记录就往这个文件中塞多少记录。由于在插入数据的时候并没有按照主键大小排序，所以我们并不能在这些数据上使用二分法进行查找。

使用MyISAM存储引擎的表会把索引信息另外存储到一个称为索引文件的另一个文件中。MyISAM会单独为表的主键创建一个索引，只不过在索引的叶子节点中存储的不是完整的用户记录，而是’主键/非主键+数据记录地址’。<可以理解为一种二级索引>



可以看出：索引树的叶子结点只存储了数据记录的地址和主键。当然非主键索引的B+树与主索引一样。

小结：

① MyISAM表可以不是必须存在主键的，因为它是非聚簇索引，索引与数据分离。

② 需要注意的是，叶子结点的父节点是空过索引值的间隙指向叶子节点的，同时在构建索引时会优先排序再构建索引的，所以不会存在索引值相同找不到记录地址的情况。

### innodb与myisam对比

**MyISAM的索引方式都是“非聚簇”的，与InnoDB包含1个聚簇索引是不同的**

区别：

1、在InnoDB存储引擎中，我们只需要根据主键值对聚簇索引进行一次查找就能找到对应的记录，而在MyISAM中却需要进行一次回表操作，意味着MyISAM中建立的索引相当于全部都是二级索引。（MyISAM的回表就是根据地址到data表中查找数据）

2、InnoDB的数据文件本身就是索引文件，而MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。

3、InnoDB的非聚簇索引data域存储相应记录主键的值，而MyISAM索引记录的是地址。换句话说，InnoDB的所有非聚簇索引都引用主键作为data域。

4、MyISAM的回表操作是十分快速的，因为是拿着地址偏移量直接到文件中取数据的，反观InnoDB是通过获取主键之后再去聚簇索引里找记录，虽然说也不慢，但还是比不上直接用地址去访问。

5、InnoDB要求表必须有主键(MyISAM可以没有)。如果没有显式指定，则MySQL系统会自动选择一个可以非空且唯一标识数据记录的列作为主键。如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整型。

InnoDB的主键是为了生成聚簇索引，所以必须有；MyISAM中的索引都是非聚簇的，数据和索引分离，所以对MyISAM没有要求。但是为了查询方便，还是建议指定某一字段为主键，从而方便存储引擎为其构建索引，提升查询效率。

补充：

* 1. Innodb不建议使用过长的字段作为主键。

<二级索引使用主键，过长的主键使得二级索引过大>

2、不建议在Innodb中用非单调的字段作为主键。

<Innodb数据文件本身是一个B+树，非单调的主键会造成在插入新纪录时，数据文件为了维持B+树的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键是最好的>

### 索引的代价

#### 空间上的代价

## innodb数据存储结构

## 索引的创建与设计原则

## 性能分析工具的使用

## 索引优化与查询优化

## 数据库设计规范

## 数据库其他调优策略

## 事务

数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

在一个业务流程中，通常需要多条DML语句共同联合才能完成，这多条DML语句必须同时完成，或者同时失败，这样才能保证数据的安全。

### 事务的四个特性

#### 原子性（Atomicity）

事务要么全部完成，要么全部取消。如果事务崩溃，数据库的状态要回滚到事务之前

#### 一致性（Consistency）

一致性是指在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。这是说数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。

#### 隔离性（Isolation）

指并发的事务是相互隔离的。即一个事务内部的操作及正在操作的数据必须封锁起来，不被企图进行修改的事务看到。

#### 持久性（Durability）

持久性是指在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。即使出现了任何事故比如断电等，事务一旦提交，则持久化保存在数据库中。

总结：

ACID是事务的四大特性，在这四个特性，原子性是基础、一致性是约束条件、隔离性是手段、持久性是我们的目的。

mysql默认的事务处理级别是'REPEATABLE-READ',也就是可重复读

oracle数据库支持READ COMMITTED 和 SERIALIZABLE这两种事务隔离级别。默认系统事务隔离级别是READ COMMITTED,也就是读已提交

### 事务的状态<事务的生命周期>

我们都知道事务是一个抽象的概念，它其实对应着一个或多个数据库操作，mysql根据这些操作所执行的不同阶段把事务大致划分为几个状态。

#### 活动的

事务对应的数据库操作正在执行过程中时，我们就说该事务处于活动的状态。

#### 部分提交的

当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并没有刷新到磁盘时，我们就说该事务处在部分提交的状态。

#### 提交的

当一个处在部分提交的状态的事务将修改过的数据都同步到磁盘上之后，我们就可以说该事务处在了提交的状态。

#### 失败的

当事务处在活动的或者部分提交的状态时，可能遇到了某些错误(数据库自身的错误、操作系统错误或者直接断电等)而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在失败的状态。

#### 中止的

如果事务执行了一部分而变为失败的状态，那么就需要把已经修改的事务中的操作还原到事务执行前的状态。

换句话说，就是要撤销失败事务对当前数据库造成的影响。我们把这个撤销的过程称之为回滚。当回滚操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了中止的状态。

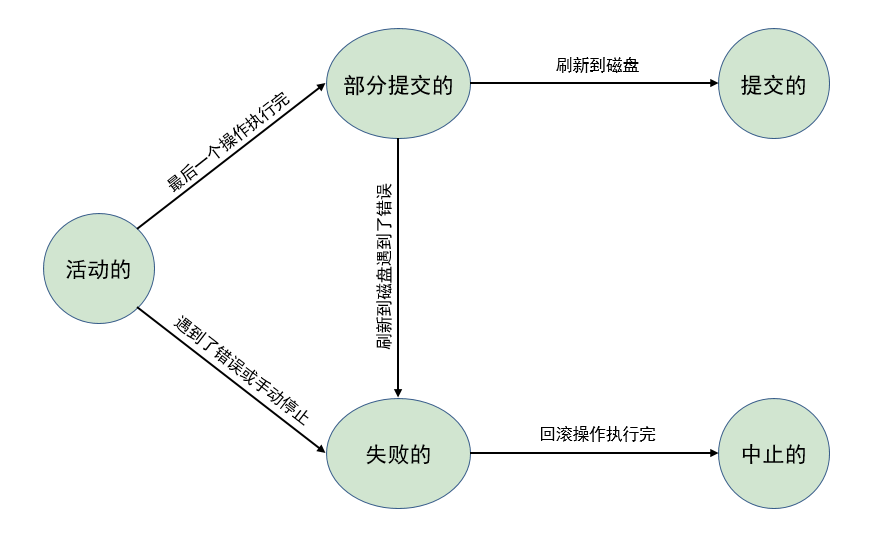


图 15 事务状态转换图

由图可知，只有当事务处于提交的或者中止的状态时，一个事务的生命周期才算是结束了。对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，对于处于中止状态的事务，该事务对数据库所做的所有修改都会被回滚到没执行该事务之前的状态。

### 事务四个处理过程

// 开启事务

start transaction;<或begin> // 开启事务后只能手动提交事务

// 执行核心业务代码

// 提交事务(提交的状态)<如果业务处理过程中没有异常>

commit;

// 回滚事务(中止的状态)<如果业务处理过程中出现异常>

rollback [to savepoint];

to savepoint：回滚到保存点。

保存点：

在事务执行过程中设置的标记点，如果只想回滚到指定的标记点上而不想回滚完，可以使用这种方式<不常用>

<创建保存点：savepoint 保存点名称>

<删除保存点：release savepoint 保存点名称>

#### 显示事务

start transaction [read only | read write | with consistent snapshot] // 开启事务

read only：只读事务。属于该事务的数据库操作只能读取数据，不能修改数据。

补充：

只读事务中只是不允许修改那些其他事务也能访问到的表中的数据，对于临时表来说(我们使用create temporary table创建的表)，由于它们只能在当前会话中可见，所以只读事务其实也是可以对临时表进行增、删、改操作的。

read write：读写事务。即可以查询也可以修改。

with consistent snapshot：启动一致性读

注意：

read only与read write可以和with consistent snapshot一起使用。

例：

start transaction read only // 开启一个只读事务

// 开启读写事务和一致性读

start transaction read write,with consistent snapshot

#### 隐式事务

关键字：autocommit // 自动提交

// 查看自动提交关键字状态

show variables like ‘autocommit’; // 默认 value=on

<mysql默认开启自动提交；Oracle默认关闭>

// 关闭自动提交

方式一：

set autocommit = false; <开启true>

注意：

这种关闭方式只是对数据操作DML有效，对数据定义DDL操作是无效的。

方式二：

在autocommit为true前提下，使用start transaction或begin开启事务，那么DML操作就不会自动提交数据。

注意：

如果在begin/start开启一个事务后没有执行提交操作，紧接着后面再次执行了begin/start开启事务，则上一个事务会自动提交。

例：

begin;

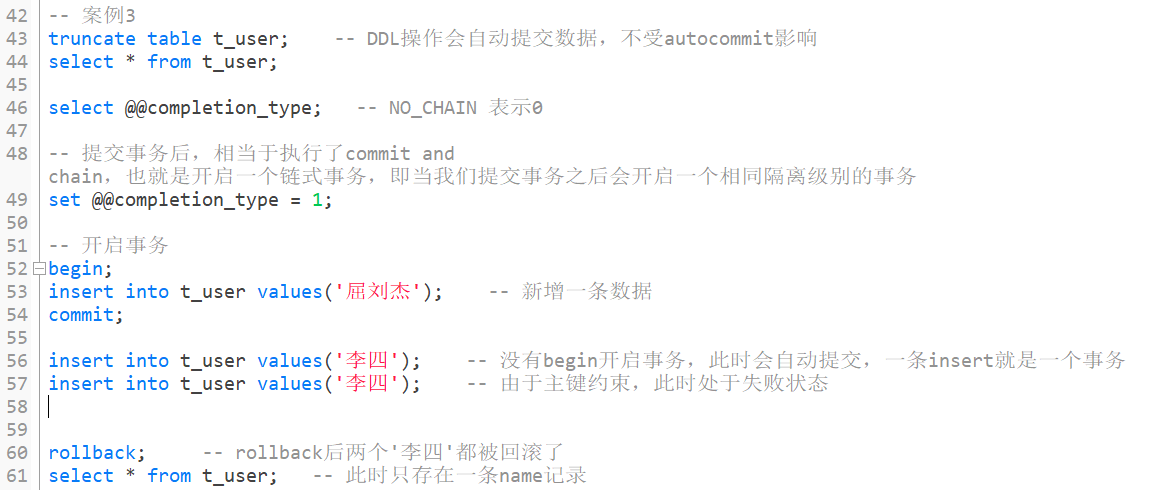
insert ……;update ……;select ……; // 此时自动提交

begin;

#### 案例：







**链式事务**：

@@completion\_type = 0

默认情况。当我们执行commit的时候会提交事务，在执行下一个事务时，还需要使用begin/start…来开启。

@@completion\_type = 1

提交事务后，相当于执行了commit and chain，也就是开启一个链式事务，即当我们提交事务之后会开启一个相同隔离级别的事务。<查看：案例3>

@@completion\_type = 2

这种情况下commit=commit and chain，也就是当我们提交后，会自动与服务器断开连接。

### 事务分类

学习地址：https://blog.csdn.net/wwj17647590781/article/details/119999298

#### 扁平事务(Flat Transactions)

是事务类型中最简单的一种，但在实际生产环境中，这可能是使用最为频繁的事务。 在扁平事务中，所有操作都处于同一层次，其由BEGIN WORK 开始，由COMMIT WORK 或ROLLBACK WORK 结束，其间的操作是原子的，要么都执行，要么都回滚。因此扁平事务是应用程序成为原子操作的基本组成模块。

注：扁平事务虽然简单，但在实际生产环境中使用最为频繁。正因为其简单，使用频繁，故每个数据库系统都实现了对扁平事务的支持。

扁平事务的主要限制是不能提交或者回滚事务的某一部分，或分几个步骤提交。

#### 带有保存点的扁平事务(Flat Transactions with Savepoints)

除了支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态。这是因为某些事务可能在执行过程中出现的错误并不会导致所有的操作都无效，放弃整个事务不合乎要求，开销也太大。

保存点（Savepoint)用来通知系统应该记住事务当前的状态，以便当之后发生错误时，事务能回到保存点当时的状态。<相当于虚拟机的快照/游戏存档>

注1：对于扁平的事务来说，其隐式地设置了一个保存点。然而在整个事务中，只有这一个保存点，因此，回滚只能回滚到事务开始时的状态。保存点用SAVEWORK函数来建立，通知系统记录当前的处理状态。当出现问题时，保存点能用作内部的重启动点，根据应用逻辑，决定是回到最近一个保存点还是其他更早的保存点。

注2：保存点在事务内部是递增的，这意味着ROLLBACK不影响保存点的计数，并且单调递增的编号能保持事务执行的整个历史过程，包括在执行过程中想法的改变。如果想要完全回滚事务，还需要再执行命令ROLLBACKWORK。

#### 链事务(Chained Transactions)

<上一小节做了简单的案例>

在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务。注意，提交事务操作和开始下一个事务操作将合并为一个原子操作。这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中进行的一样。

注1：可视为保存点模式的一种变种。带有保存点的扁平事务，当发生系统崩溃时，所有的保存点都将消失，因为其保存点是易失的(volatile),而非持久的(persistent)。这意味着当进行恢复时，事务需要从开始处重新执行，而不能从最近的一个保存点继续执行。

注2：链事务与带有保存点的扁平事务不同的是，带有保存点的扁平事务能回滚到任意正确的保存点。而链事务中的回滚仅限于当前事务，即只能恢复到最近一个的保存点。对于锁的处理，两者也不相同。



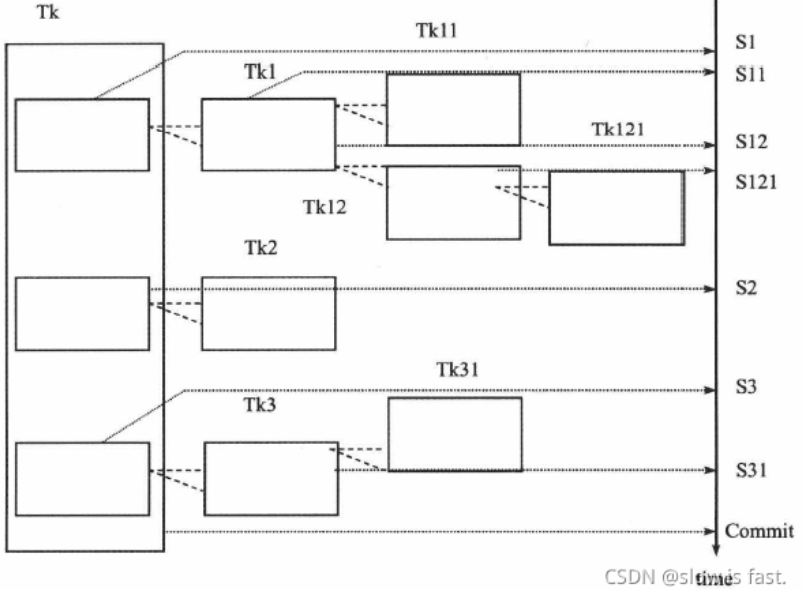
注3：链事务在执行COMMIT后即释放了当前事务所待有的锁，而带有保存点的扁平事务不影响迄今为止所持有的锁。

#### 嵌套事务(Nested Transactions)

是一个层次结构框架。由一个顶层事务(topleveltransaction)控制着各个层次的事务。顶层事务之下嵌套的事务被称为子事务(subtransaction),其控制每一个局部的变换。

注1：嵌套事务是由若干事务组成的一棵树，子树既可以是嵌套事务，也可以是扁平事务。处在叶子节点的事务是扁平事务。但是每个子事务从根到叶节点的距离可以是不同的。位于根节点的事务称为顶层事务(根)，其他事务称为子事务。事务的前驱称(predecessor)为父事务(parent),事务的下一层称为儿子事务(child)。子事务既可以提交也可以回滚。但是它的提交操作并不马上生效，除非其父事务巳经提交。因此可以推论出，任何子事物都在顶层事务提交后才真正的提交。树中的任意一个父事务的回滚会引起它的所有子事务一同回滚，故子事务仅保留A、C、I特性，不具有D的特性。

注2：实际的工作是交由叶子节点来完成的，即只有叶子节点的事务才能访问数据库、发送消息、获取其他类型的资源。而高层的事务仅负责逻辑控制，决定何时调用相关的子事务。即使一个系统不支持嵌套事务，用户也可以通过保存点技术来模拟嵌套事务。如下图：在恢复时采用保存点技术比嵌套查询有更大的灵活性，如在完成Tk3这事务时，可以回滚到保存点S2的状态。而在嵌套查询的层次结构中，却不允许。



嵌套事务的灵活性表现如下：

①、当通过保存点技术来模拟嵌套事务时，用户无法选择哪些锁需要被子事务继承，哪些需要被父事务保留。这就是说，无论有多少个保存点，所有被锁住的对象都可以被得到和访问。而在嵌套查询中，不同的子事务在数据库对象上持有的锁是不同的（说白了就是对于嵌套事务父事务的锁可以全部给它的子事务也可以就给其中的几个，如果子事务有父事务没有的锁，那么通过反向继承，也可以使父事务持有，而对于保存点技术模拟的嵌套事务是不能的选择哪些锁被优化的）。

②、如果系统支持在嵌套事务中并行地执行各个子事务，在这种情况下，采用保存点的扁平事务来模拟嵌套事务就不切实际了。这从另一个方面反映出，想要实现事务间的并行性，需要真正支持的嵌套事务。

#### 分布式事务(Distributed Transactions)

通常是一个在分布式环境下运行的扁平事务，因此需要根据数据所在位置访问网络中的不同节点（银行转账就是典型的分布式事务）。

注：对于InnoDB存储引擎来说，其支持扁平事务、带有保存点的事务、链事务、分布式事务。对于嵌套事务，其并不原生支持，因此，对有并行事务需求的用户来说，MySQL数据库或InnoDB存储引擎就显得无能为力了。然而用户仍可以通过带有保存点的事务来模拟串行的嵌套事务。

### 不隔离事务会出现哪些问题？

https://zhuanlan.zhihu.com/p/150107974

#### 脏写

所谓脏写，就是我刚才明明写了一个数据值，结果过了一会却没了。<修改丢失>

事务A和事务B同时在更新一条数据，事务A先把它更新为a值，事务B紧接着就把它更新为b值。

事务B去修改了事务A修改过的值，但是此时事务A还没提交，所以事务A随时会回滚，导致事务B修改的值也没了。



最终select查询name值时，发现既不是’wzx’，也不是’xyq’。

注意：

事务A回滚时，是通过自己修改时的日志进行回滚的，所以虽然事务B将数据已经修改成功，但是事务A的日志中存的还是两个事务未修改前的数据，即发生脏写。

<脏写这个问题很严重，所以所有的隔离级别都不允许脏写出现>

#### 脏读

A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。<脏数据产生原因是回滚>



解决方法：

如果在第一个事务提交前，任何其他事务不可读取其修改过的值，则可以避免该问题。

<无论是脏写还是脏读，都是因为一个事务去更新或者查询了另外一个还没提交的事务更新过的数据。因为另外一个事务还没提交，所以它随时可能会回滚，那么必然导致你更新的数据就没了，或者你之前查询到的数据就没了，这就是脏写和脏读>

#### 不可重复读

<更新事务未提交前，单个查询事务中不能对数据重复读取>

一个事务对同一行数据重复读取两次，但是却得到了不同的结果。事务A读取某一数据后，事务B对其做了修改，当事务A再次读该数据时得到与前一次不同的值。



解决方法：

如果只有在修改事务完全提交之后才可以读取数据，则可以避免该问题。

#### 幻读

指一个事务执行两次查询，但第二次查询的结果包含了第一次查询中未出现的数据(insert)。

<幻读强调的是多了数据，如果事务执行了delete语句可以归结到不可重复读>



### 四种隔离级别

#### 读未提交(READ UNCOMMITTED)

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读。

MySQL 事务隔离其实是依靠锁来实现的，加锁自然会带来性能的损失。而读未提交隔离级别是不加锁的，所以它的性能是最好的，没有加锁、解锁带来的性能开销。

依然可能发生：<脏读，不可重复读，幻读>

#### 读已提交(READ COMMITED)

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变<读已提交>。

这种隔离级别也支持所谓的不可重复读（NonrepeatableRead），因为同一事务的其他实例在该实例处理期间可能会有新的commit，所以同一select可能返回不同结果。

依然可能发生：<不可重复读，幻读>

#### 可重复读(REPEATABLE READ)

就是在开始读取数据（事务开启）时，不再允许修改操作。(实际更加复杂且可以避免幻读)

这是MySQL的默认事务隔离级别，同一事务的多个实例在并发读取数据时，会看到同样的数据。不过理论上，这会导致另一个棘手的问题：

幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。

依然可能发生：<幻读>

#### 可串行化(SERIALIZABLE)

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。

在这个级别，可能导致大量的超时现象和锁竞争。

### 事务隔离操作

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 隔离级别

例：

-- 设置当前会话事务隔离级别<当前会话就是当前查询页面>

set session transaction isolation level read uncommitted

< global表示全局范围设置>

## 事务日志

学习网址：https://blog.csdn.net/m0\_46845579/article/details/125777408

### 简介

事务有4种特性：原子性、一致性、隔离性、持久性。下面我们来介绍四种特性是基于什么机制实现的。

事务的隔离性由锁机制实现。

事务的原子性、一致性、持久性由事务的日志(redo、undo)来保证。

redo log：**重做日志**。提供再写入操作，恢复提交事务修改的页操作，用来保证事务的持久性。

undo log：**回滚日志**。回滚记录到某个特定版本，用来保证事务的原子性、一致性。

<两种日志都可以视作为一种恢复操作>

redo将内存数据恢复到磁盘；undo将失败的数据回滚到修改前

两种日志的区别：

redo log：是存储引擎层(innodb)生成的日志，记录的是‘物理级别’上的页修改操作，比如页号xxx、偏移量yyy和写入了zzz数据。主要为了保证数据的可靠性。

<redo日志实际就是保证部分提交状态到提交的状态(commit)>

undo log：是存储引擎层(innodb)生成的日志，记录的是‘逻辑操作’日志，比如对某一行数据进行了insert语句，那么undolog就记录一条与之相反的delete操作。主要用于事务的回滚(undolog记录的是每个修改操作的逆操作)和一致性非锁定读(undolog回滚行记录的某种特定的版本—MVCC，即多版本并发控制)。

<undo日志实际就是保证失败的状态到中止的状态(rollback)>

### redo日志

innodb存储引擎是以数据页为单位来管理存储空间的。在真正访问数据页之前，需要把在磁盘上的页缓存到内存中的Buffer Pool(缓冲池)之后才可以访问。所有的变更都必须先更新缓冲池中的数据，然后缓冲池中的脏页会以一定的频率被刷入磁盘(checkPoint机制)，通过缓冲池来优化CPU和磁盘之间的鸿沟，这样可以保证整体的性能不会降太快。

脏页：当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”

数据页：MySQL是按照数据页为最小单位进行存储的，b+树的特性，一页默认16k，数据属于页，页包含数据。当操作一条数据的时候，实际上就是在操作一个数据页，多条数据可能是在一个数据页也可能是在多个数据页，看数据具体分布。

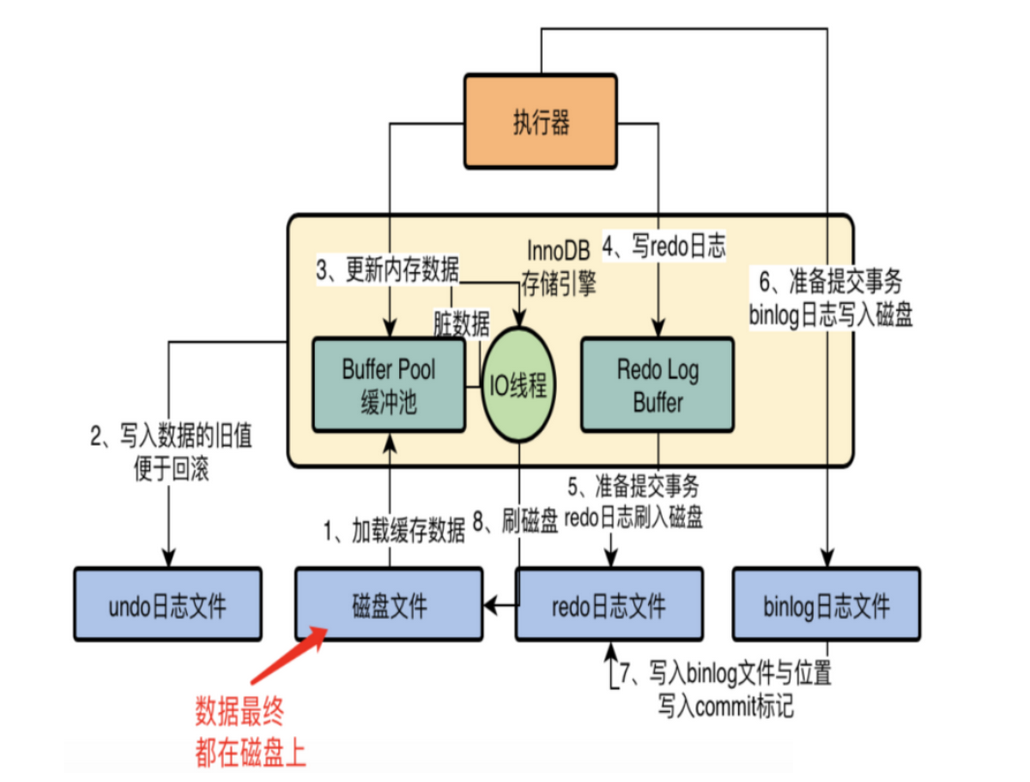


图 16 innodb存储引擎执行逻辑

#### 为什么需要redo日志<作用>

一方面，缓冲池可以帮助我们消除CPU和磁盘之间的鸿沟，checkpoint机制可以保证数据的最终落盘。然而由于checkpoint机制并不是每次变更的时候都会触发，而是master线程隔一段时间去处理的。所以最坏的情况就是事务提交之后，刚写完缓冲池，数据库宕机了，那么这段数据就是丢失的，无法恢复。<不能保证持久性>

另一方面，事务包含持久性的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。

如何解决checkpoint机制与持久性的矛盾？

简单的解决方法：

在事务提交完成之前把该事务所修改的所有数据页都刷新到磁盘。

这样粗暴的办法会存在一些问题：

① 修改量与刷新磁盘工作量严重不成比例

比如只修改一个数据，而底层要将整个数据页进行磁盘IO传输，小题大做。

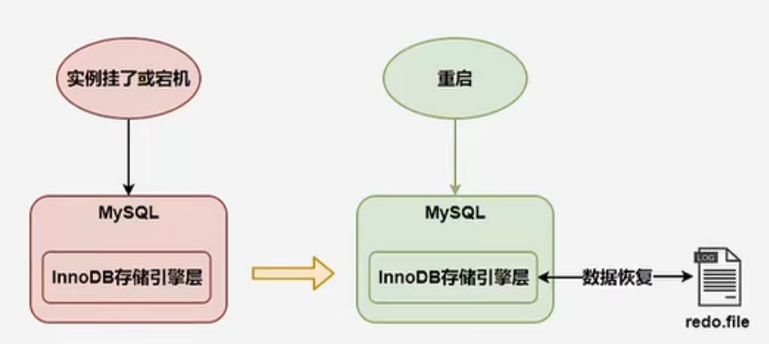
② 随机IO刷新较慢。

随机IO：读写操作时间连续，但访问地址不连续，随机分布在磁盘的地址空间中。

Mysql的解决办法：

每次对内存数据的刷新，都把操作记录下来就好了。就算数据库宕机，我们也可以重启数据库后，重新执行记录的操作，就可以将没有成功刷新的数据刷新到磁盘上。

Innodb采用了"日志先行"策略，每次修改，先写redo日志，再写磁盘，只有日志写入成功，才算事务提交成功。<保证了持久性>



总结作用：

redo log就是为了解决系统异常导致内存修改丢失的问题。<容灾>

#### redo日志特点、优点

优点：

redo日志降低了刷盘频率。

redo日志占用空间非常小。

特点：

redo日志是顺序写入磁盘的。

执行事务时，一条语句可能产生若干个日志语句，这些日志语句是按顺序写入磁盘的，是顺序IO，效率比随机IO快。

事务执行过程中，redolog不断记录。

redo日志记录不间断，区别于bin日志(数据库层产生)一个事务提交才整个记录一次。

#### redo组成

**缓冲区**和**日志文件区。**

**缓冲区(redo log buffer)**

① 缓冲区是内存层面的。

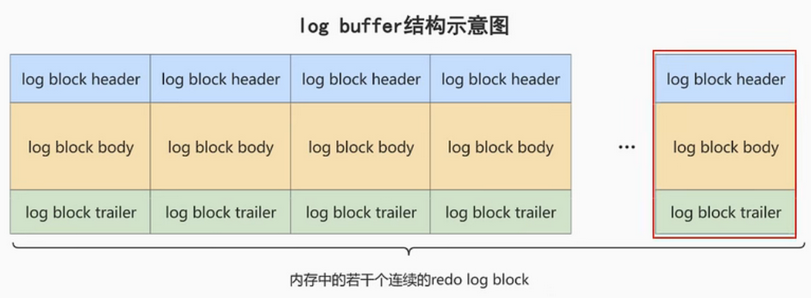
② 缓冲区默认大小16M，最小是1M，最大4096M。

查看log buffer内存大小：

show variables like '%innodb\_log\_buffer\_size%';



③ 缓冲区会将空间划分成一个个的缓冲块。



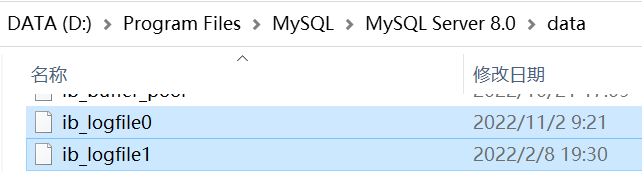
在服务器启动时就向操作系统申请了一大片称之为redo log buffer的连续内存空间，也称作redo日志缓冲区。这片内存空间被划分为若干个连续的redo log back。一个redo log back占用512字节大小。

**日志文件区(redo log file)**

日志文件是磁盘层面的，是持久的。

查看mysql目录，其中logfile0和logfile1就是redo日志的文件区(默认有两个)。

windows：<轮询写入>

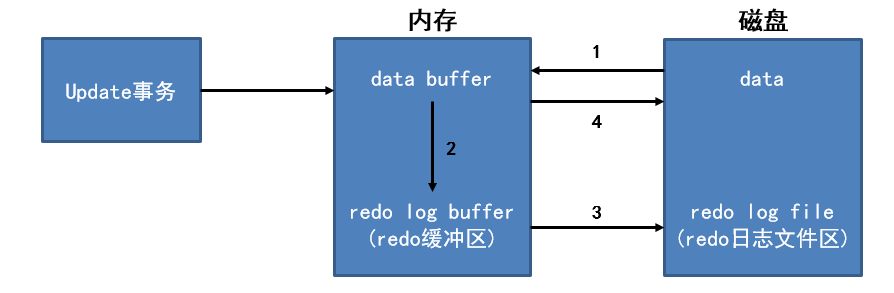


linux：命令查询：cd /var/lib/mysql -> ll/ls

#### redo的整体流程

以一个更新事务为例，redo log流转过程：

<实际上在图10 innodb存储逻辑已经介绍到了>



步骤：

① 先将原始数据从磁盘中读入内存中来，修改数据的内存拷贝。

② 生成一条重做日志(redo日志)并写入redo log buffer(缓冲区)，记录的是数据被修改后的值。

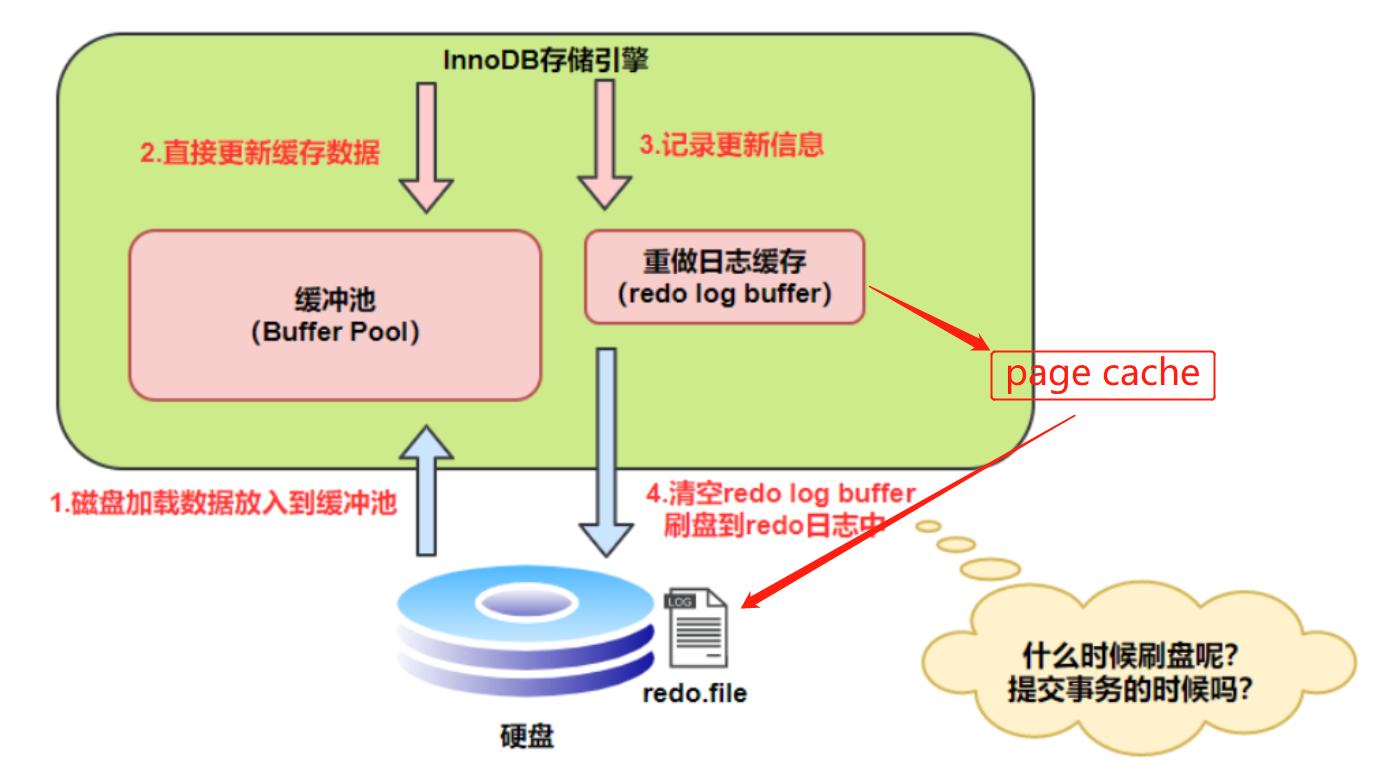
③ 当事务commit时，将redo log buffer(缓冲区)中的内容刷新到redo log file(日志文件区)，对redo log file采用追加写的方式。<重点步骤>

④ 定期将内存中修改的数据刷新到磁盘中。

#### redo log的刷盘策略

<解决redo整体流程中第三步写入操作问题>

redo的整体流程中提到redo log主要是由内存缓存区写入到磁盘日志文件区的，InnoDB引擎会在写redo log的时候先写redo log buffer，之后以一定的频率刷入到真正的redo log file中。这里的一定频率怎么看待呢？



【注意】

redo log buffer刷盘到redo log file的过程并不是真正的刷到磁盘中去，只是刷入到文件系统缓存（page cache）中去，真正的写入会交给操作系统自己来决定（比如page cache足够大了）。

新的问题：

对于innodb来说，如果交给系统来同步，同样如果系统宕机，那么数据也丢失了(容灾失败)。

针对这种新的情况，innodb给出了一个**innodb\_flush\_log\_at\_trx\_commit**参数。

参数作用：

控制commit提交事务时，如何将redo log buffer中的日志刷新到redo log file中。

参数设置：

设置为**0**：

表示每次事务提交时不进行刷盘操作。（系统默认master thread每隔1s进行一次重做日志的同步）

设置为**1**：

表示每次事务提交时都将进行同步、刷盘操作（默认值）

设置为**2**：

表示每次事务提交时都只把redo log buffer内容写入page cache，不进行同步。由os自己决定什么时候同步到磁盘文件。

**【redo log buffer到redo log file详解】**

innodb存储引擎有一个后台线程，每隔1s,就会把redo log buffer(日志缓存区)的内容写入page cache(文件系统缓存)，然后系统再调用刷盘操作写入redo log file。

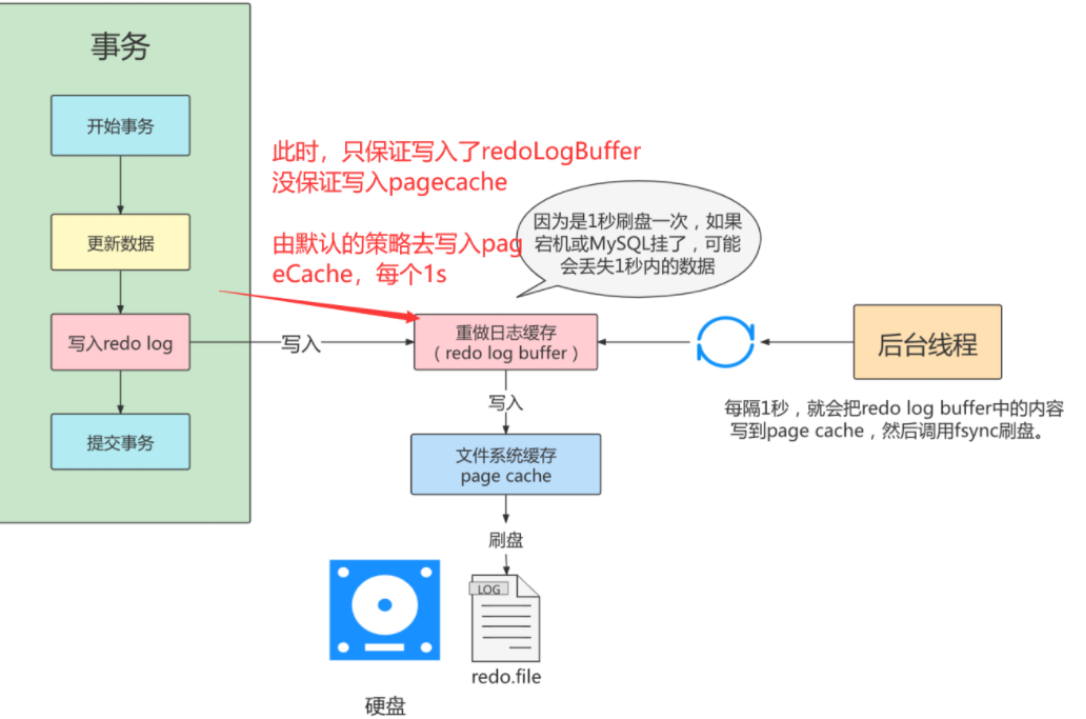
注意：这两个步是同步的，前一步执行完成后一步就会执行。

也就是说，一个没有提交事务的redo log记录，也可能会将redoLogBuffer中的日志刷盘到redoLogFile中。因为在事务执行过程redo log记录是会写入redo log buffer中，这些redo log记录会被后台线程刷盘。

【当参数为0】

每次事务提交时不主动进行刷盘操作。

此时只保证写入了redo log buffer，不能保证写入page cache，所以只能由innodb引擎的默认策略写入pageCache，然后刷盘写入redo log file中。这就导致redo log file不能保证一定存储日志。<最多丢失1秒钟内的事务日志>



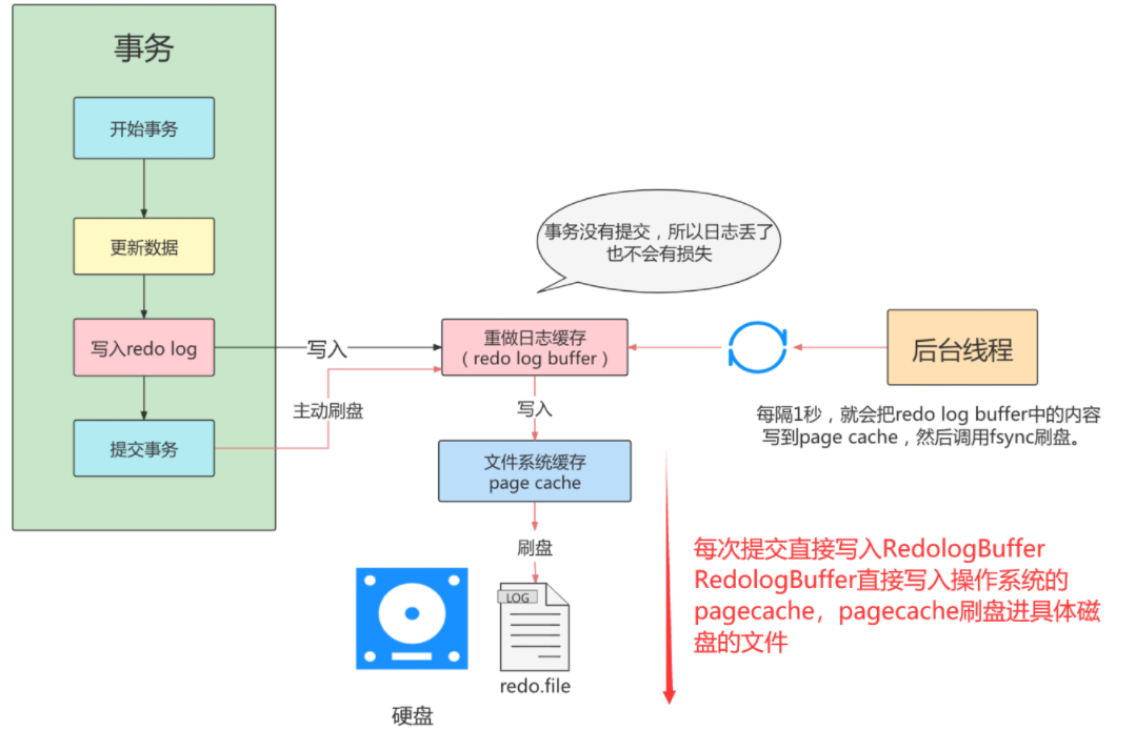
【当参数为1】

mysql默认值。

只要事务提交成功<主动刷盘>，redo log记录就一定在硬盘里，不会有任何数据丢失。

如果事务执行期间MySQL挂了或宕机，这部分日志丢了，但是事务并没有提交，所以日志丢了也不会有损失。可以保证ACID的D，数据绝对不会丢失，但是效率最差的。

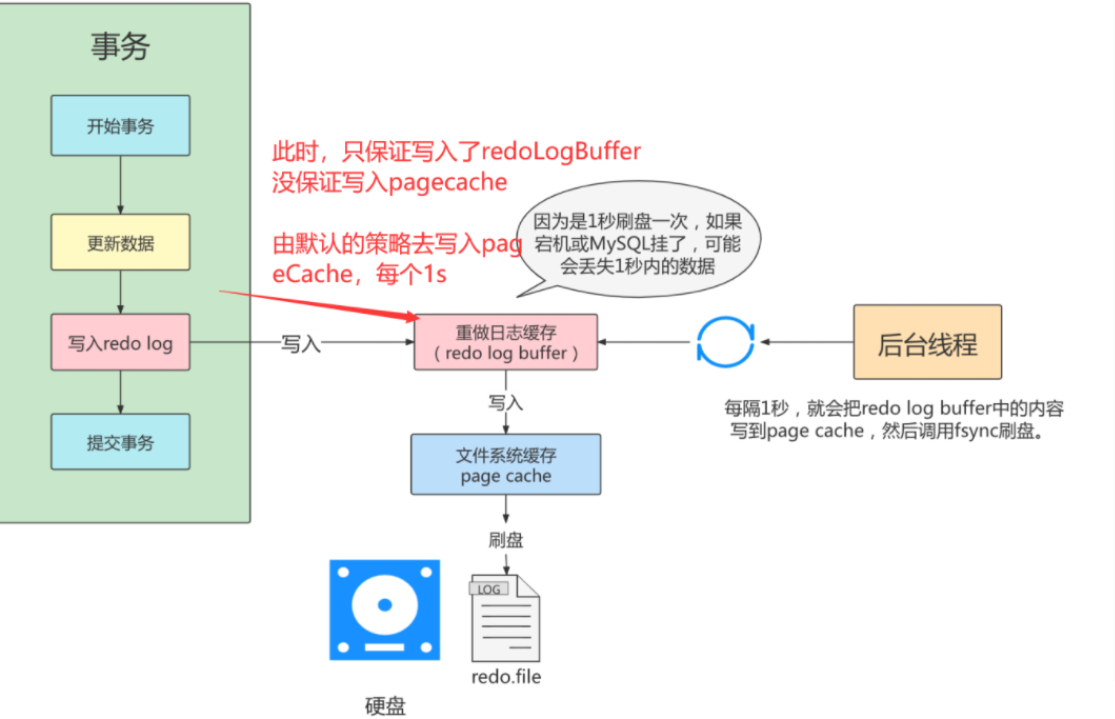
建议使用默认值，虽然操作系统宕机的概率理论小于数据库宕机的概率，但是一般既然使用了事务，那么数据的安全相对来说更重要些。



【当参数为2】

只要事务提交成功，redo log buffer中的内容只写入文件系统缓存（page cache)。

如果仅仅只是MysQL挂了不会有任何数据丢失，但是操作系统宕机可能会有1秒数据的丢失，这种情况下无法满足ACID中的D。但是数值2肯定是效率最高的。



总结：**0：延迟写，延迟刷，1：实时写，实时刷，2：实时写，延迟刷**

参数操作：

查看数据库参数值：

show variables like ‘innodb\_flush\_log\_at\_trx\_commit’;

设置参数值：

set [session | global] innodb\_flush\_log\_at\_trx\_commit = 1;

#### 详解写入redo log buffer

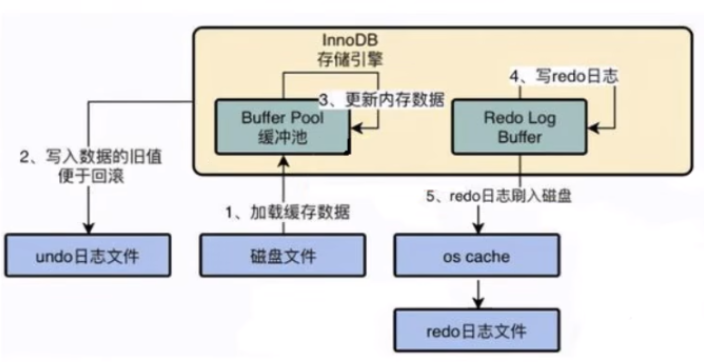
略，在上面学习网站中学习。<了解>

#### 详解redo log file

略，在上面学习网站中学习。<了解>

#### redo log小结

innodb的更新数据操作采用的是Write Ahead Log(预先日志持久化)策略，即先写redo log日志，再写入磁盘。



### undo日志

详解网址：https://zhuanlan.zhihu.com/p/449852704

redo日志是事务持久性的保证，undo日志是事务原子性的保证。在事务中更新数据的前置操作其实是要先写入一个undo log。

#### 理解undo日志

事务需要保证原子性，也就是事务中的操作要么全部完成，要么什么也不做。但有时候事务执行到一半会出现一些情况：

情况一：事务执行过程中遇到各种错误。比如：服务器本身错误、操作系统错误、甚至断点导致的错误。

情况二：程序员可以在事务执行过程中手动进行rollback语句回滚操作。

上面两种情况都会将数据恢复到事务开始的样子<回滚>。这个事务看起来就像是没有做什么事情<原子性>。

系统怎样回滚数据？

插入：对于每个insert操作，innodb存储引擎回滚时会完成一个delete操作。

修改：对于每个update操作，innodb存储引擎回滚时完成一个逻辑相反的update操作。

删除：对于每个insert操作，innodb存储引擎回滚时会完成一个insert操作。

这些被执行的反向操作都被innodb存储引擎事先存储起来。

MysQL把这些为了回滚而记录的这些内容称之为撤销日志/回滚日志(即undo log)。

注意：

由于查询操作(SELECT）并不会修改任何用户记录，所以在查询操作执行时，并不需要记录相应的undo日志。

undo log会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。

#### undo日志的作用

1. 回滚数据

undo并不是将数据库物理地恢复到执行语句或事务之前的样子。

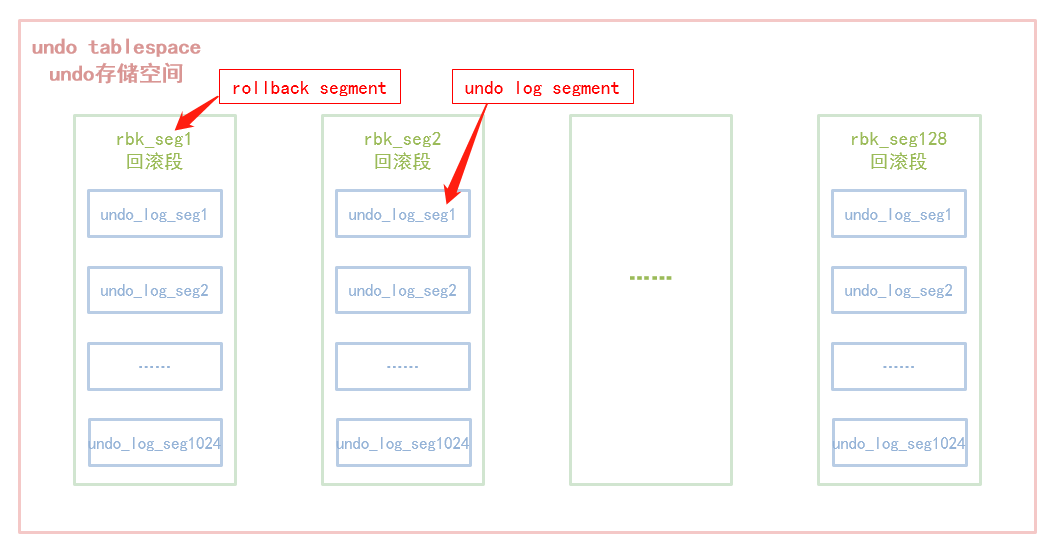
undo是逻辑日志，因此只是将数据库逻辑地(非物理地)恢复到原来的样子。所有的事务修改都被逻辑地取消了，但是数据结构和页本身在回滚之后可能大不相同。

2、MVCC

undo的另一个作用是MVCC,即在InnoDB存储引擎中MVCC的实现是通过undo来完成。当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过undo读取之前的行版本信息，以此实现非锁定读取。

#### undo的存储结构

InnoDB对undo log存储空间的管理采用段的方式，也就是回滚段(rollback segment)。每个回滚段记录了1024个undo log segment，而在每个undo log segment段中进行undo页的申请。



在InnoDB1.1版本之前，只有一个rollback segment，因此支持同时在线的事务限制为1024。虽然对绝大多数的应用来说都已经够用。

从1.1版本开始InnoDB支持最大128个rollback segment，故其支持同时在线的事务限制提高到了128\*1024。

查看回滚段数：show variables like ‘innodb\_undo\_logs’; // 128

#### undo页重用

当我们开启一个事务需要写undo log的时候，就得先去undo log segment中去找到一个空闲的位置，当有空位的时候，就去申请undo页，在这个申请到的undo页中进行undo log的写入。我们知道mysql默认一个数据页的大小是16k。

【为什么需要undo页重用】

为每一个事务分配一个undo页，是非常浪费的(除非你的事务非常长)，假设你的应用的TPS(每秒处理的事务数目）为1000，那么1s就需要1000个undo页，大概需要16M的存储，1分钟大概需要1G的存储。如果照这样下去除非MySQL清理的非常勤快，否则随着时间的推移，磁盘空间会增长的非常快，而且很多空间都是浪费的。

【如何实现undo页重用】

当事务提交时，并不会立刻删除undo页。因为重用，所以这个undo页可能混杂着其他事务的undo log。undo log在commit后，会被放到一个链表中，然后判断undo页的使用空间是否小于3/4，如果小于3/4的话，则表示当前的undo页可以被重用，那么它就不会被回收，其他事务的undo log可以记录在当前undo页的后面。由于undo log的存储是离散的，所以清理对应的磁盘空间时，效率不高。

#### 回滚段与事务

1、每个事务只会使用一个回滚段，一个回滚段在同一时刻可能会服务于多个事务。

2、当一个事务开始的时候，会制定一个回滚段，在事务进行的过程中，当数据被修改时，原始的数据会被复制到回滚段。

3、在回滚段中，事务会不断填充盘区，直到事务结束或所有的空间被用完。如果当前的盘区不够用，事务会在段中请求扩展下一个盘区，如果所有已分配的盘区都被用完，事务会覆盖最初的盘区或者在回滚段允许的情况下扩展新的盘区来使用。

4、回滚段存在于undo表空间中，在数据库中可以存在多个undo表空间(默认最小数量为2)，但同一时刻只能使用一个undo表空间。

show variables like ‘innodb\_undo\_tablespaces’;

undo log的truncate操作由purge协助线程发起。

5、当事务提交时，InnoDB存储引擎会做以下两件事情：

① 将undo log放入列表中，以供之后的purge操作；

② 判断undo log所在的undo页是否可以重用，若可以分配给下个事务使用。

#### 回滚段中的数据分类

1. 未提交的回滚数据(uncommitted undo information)：

该数据所关联的事务并未提交，用于实现读一致性，所以该undo回滚数据不能被其他事务的数据覆盖。

2、已经提交但未过期的回滚数据(committed undo information)：

该数据关联的事务已经提交，但是仍受到undo retention参数的保持时间的影响。

3、事务已经提交并过期的数据(expired undo information)：

事务已经提交，而且数据保存时间已经超过undo retention参数指定的时间，属于已经过期的数据。当回滚段满了之后，会优先覆盖"事务已经提交并过期的数据"。

事务提交后并不能马上删除undo log及undo log所在的页。这是因为可能还有其他事务需要通过undo log来得到行记录之前的版本。故事务提交时将undo log放入一个链表中，是否可以最终删除undo log及undo log所在页由purge线程来判断。

undo retention：

<有待商榷>

事务提交后undo log的保质期<默认900s>。超过期限就是过期undo数据。但是当undo数据满了之后，还是会优先将之前的undo数据覆盖掉。也就是说，一个undo log数据最大可以保存900s，之后就会过期被purge线程清理。

#### undo的类型

在InnoDB存储引擎中，undo log分为：

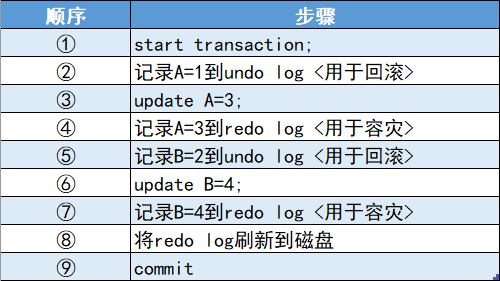
insert undo log是指在insert操作中产生的undo log。因为insert操作的记录，只对事务本身可见，对其他事务不可见（这是事务隔离性的要求)，故该undo log可以在事务提交后直接删除。不需要进行purge(truncate)操作。

update undo log记录的是对delete和update操作产生的undo log。该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表，等待purge线程进行最后的删除。

#### undo log的生命周期

案例：

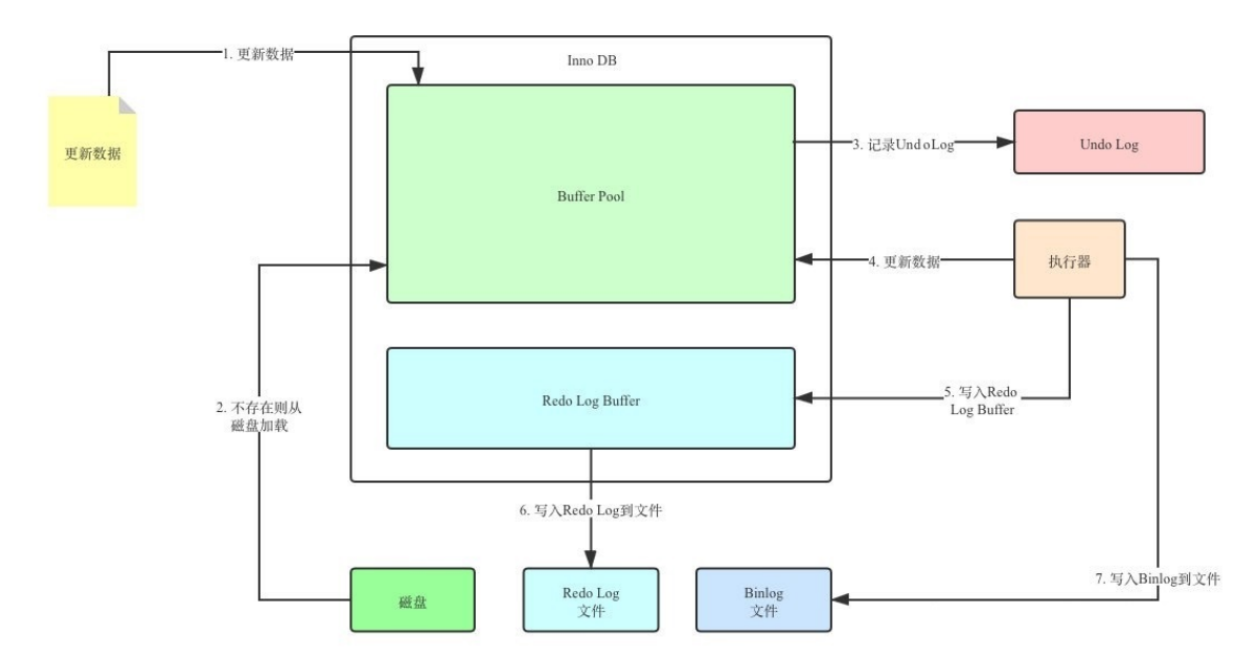
假设有2个数值，分别为A=1和B=2，然后将A修改为3，将B修改为4



在1-8步骤的任意步系统宕机，事务未提交，该事务就不会对磁盘上的数据做任何影响。

如果8-9之间宕机，恢复之后可以选择回滚，也可以选择继续完成事务提交，因为此时redo log已经持久化。

若在9之后系统宕机，内存映射中变更的数据还来不及刷回磁盘，那么系统恢复之后，可以根据redo log把数据刷回磁盘。



【undo log的删除】

针对insert undo log

因为insert操作的记录，只对事务本身可见，对其他事务不可见。故该undo log可以在事务提交后直接删 除，不需要进行purge操作。

针对update undo log

该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表，等 待purge线程进行最后的删除。

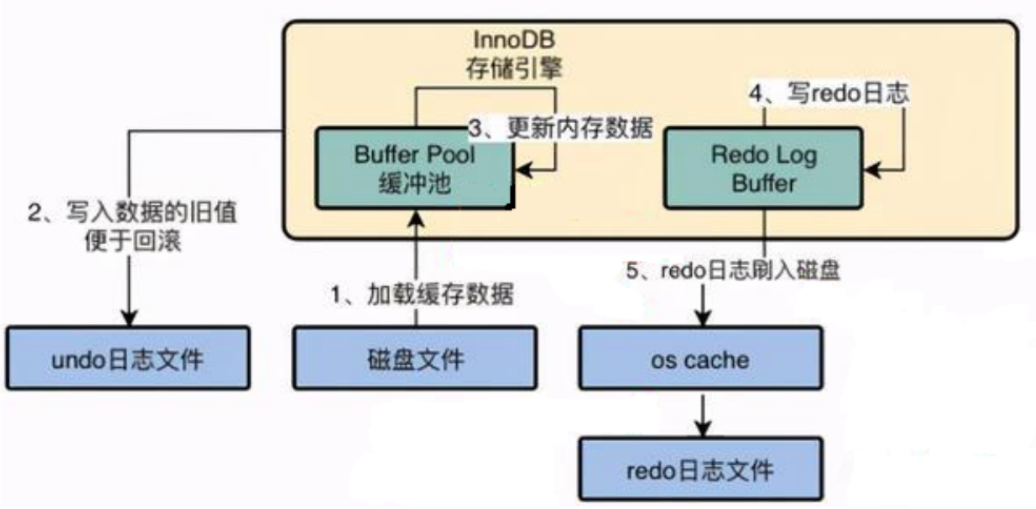
补充:

purge线程两个主要作用是:清理undo页和清除page里面带有Delete\_Bit标职的数据行。在InnoDB中，事务中的Delete操作实际上并不是真正的删除掉数据行，而是一种Delete Mark操作，在记录上标识Delete\_Bit，而不删除记录。是一种"假删除";只是做了个标记，真正的删除工作需要后台purge线程去完成。

#### 小结

undo log是逻辑日志，对事务回滚时，只是将数据库逻辑的恢复到原来的样子。

redo log是物理日志，记录的是数据页的物理变化，undo log不是redo log的逆过程。



## 锁

学习网址：https://blog.csdn.net/weixin\_46245201/article/details/123601014

锁是计算机协调多个进程或线程并发访问某一资源的机制。

在数据库中，除传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

### mysql并发事务访问相同记录

并发事务访问相同记录的情况大致可以划分为3种：

#### 读-读

并发事务相继读取相同的记录。读取操作本身不会对记录有任何影响，并不会引起什么问题，所以允许这种并发读这种情况发生。

#### 写-写

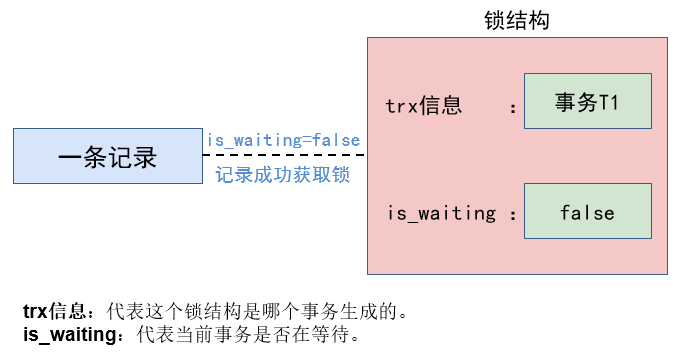
并发事务相继对相同的记录做出修改。

这种情况会发生[脏写](#_脏写)问题，任何隔离级别都不允许存在脏写情况发生。所以多个未提交事务相继对一条记录做修改时，需要让它们排队执行。这个排队执行就是通过锁来实现的。

这个所谓的锁其实是一个内存中的结构，在事务执行前本来是没有锁的，也就是说一开始是没有锁结构和记录进行关联的。<注意：锁由事务生成，然后关联到记录上>

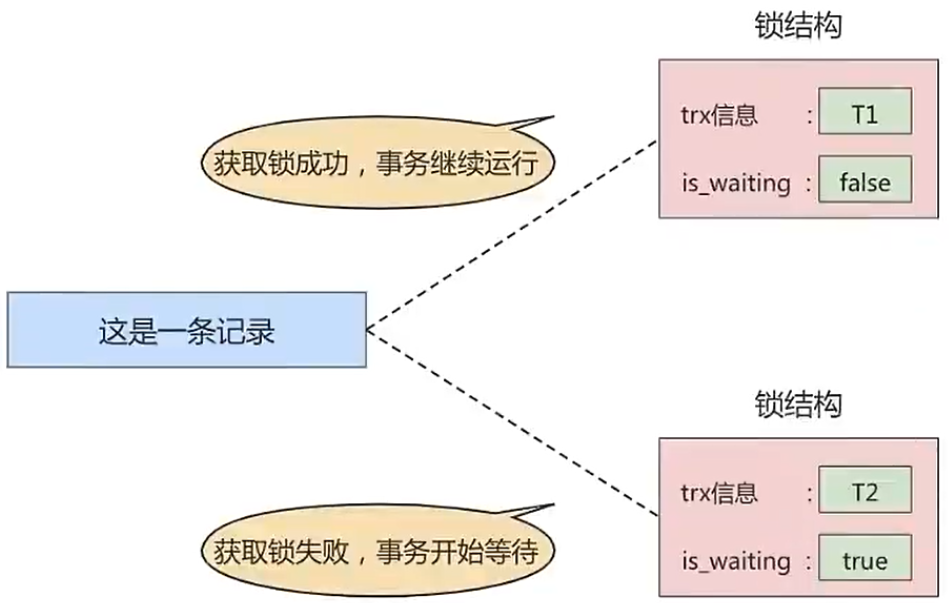
【如何加锁并获取锁】

1、当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的锁结构，当没有的时候就会在内存中生成一个锁结构与之关联。<对记录加锁>

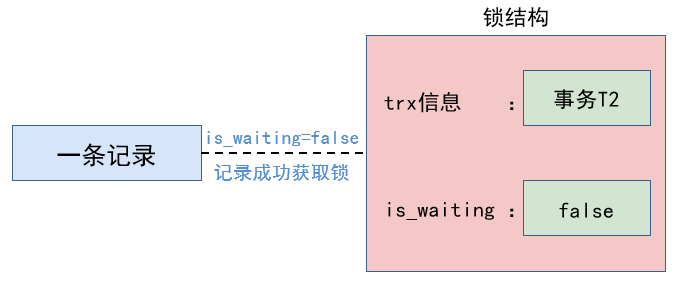


2、当事务T1改动了这条记录后，就生成了一个锁结构与该记录关联，因为之前没有别的事务为这条记录加锁，所以is\_waiting属性就是false，我们把这个场景就称之为获取锁成功，或者加锁成功，然后就可以继续执行操作了

3、在事务T1提交之前，另一个事务T2也想对该记录做改动，那么先看看有没有锁结构与这条记录关联，发现有一个锁结构与之关联后，然后也生成了一个锁结构与这条记录关联，不过锁结构的is\_waiting属性值为true，表示当前事务需要等待，我们把这个场景就称之为获取锁失败，或者加锁失败。



4、在事务T1提交之后，就会把该事务生成的锁结构释放掉(释放锁)，然后看看还有没有别的事务在等待获取锁，发现了事务T2还在等待获取锁，所以把事务T2对应的锁结构的is\_waiting属性设置为false，然后把该事务对应的线程唤醒，让它继续执行，此时事务T2就算获取到锁了。



#### 读-写或写-读

读-写或写-读，即一个事务进行读取操作，另一个进行改动操作。

这种情况下可能发生脏读、不可重复读、幻读的问题。

如何解决这些问题是关键。

#### 并发问题的解决方案

解决脏读、不可重复读、幻读这些问题。<实际就是隔离级别的实现>

* **方案一：MVCC**

读操作利用多版本并发控制（MVCC），写操作进行加锁

所谓的MVCC，就是生成一个ReadView(快照)，通过ReadView找到符合条件的记录版本（历史版本由undo日志构建）。查询语句只能读到在生成ReadView之前己提交事务所做的更改，在生成ReadView之前未提交的事务或者之后才开启的事务所做的更改是看不到的。

写操作直接针对最新版本的修改记录<加锁实现>。

读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用MVCC时，读-写操作并不冲突。

【SELECT语句在’读已提交’和’可重复读’隔离级别下会使用到MVCC读取记录。】

1、在读已提交隔离级别

一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView(快照)，ReadView的存在本身就保证了事务不可以读取到未提交的事务所做的更改，也就是避免了脏读现象；<结合[脏读](#_脏读)理解>

2、在可重复读隔离级别

一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都复用这个ReadView,这样也就避免了不可重复读和幻读的问题。

<[不可重复读](#_不可重复读)、[幻读](#_幻读)>

注意：不可重复读与幻读都是因为在一个事务中多次SELECT记录，而在这期间被查询的记录被其他事务修改了，所以只允许使用第一次SELECT生成的ReadView快照。

* **方案二：读、写操作都加锁**

读与写每次都必须读取记录的最新版本。采用读与写加锁操作，意味着读-写操作与写-写操作一样，排队执行。

对于脏读。

事务在执行写操作时(不commit/rollback不会释放锁)，对记录进行加锁，此时读取操作对应的事务不能读取该记录。也就是说记录是一个共享对象，不能被写与读同时操作，两个事务排队使用记录对象。

对于不可重复读、幻读。

如果读操作对应的事务读取记录时，写操作对应的事务不能使用记录对象，读操作的两次SELECT都会先一步执行完，再执行UPDATE/INSERT操作。

<幻读在此处加锁时会不知道给哪条记录添加锁，因为幻读时insert，与不可重复读不同(不可重复读update，可以定位到具体的记录)，幻读由于新增记录，如果对具体记录添加行级锁就不能起到避免幻读的效果。(mysql使用间隙锁解决这一问题，后面会讲到)>

* 小结

采用MVCC方式的话，读-写操作彼此并不冲突，性能更高。

采用加锁方式的话，读-写操作彼此需要排队执行，影响性能。

### 锁分类

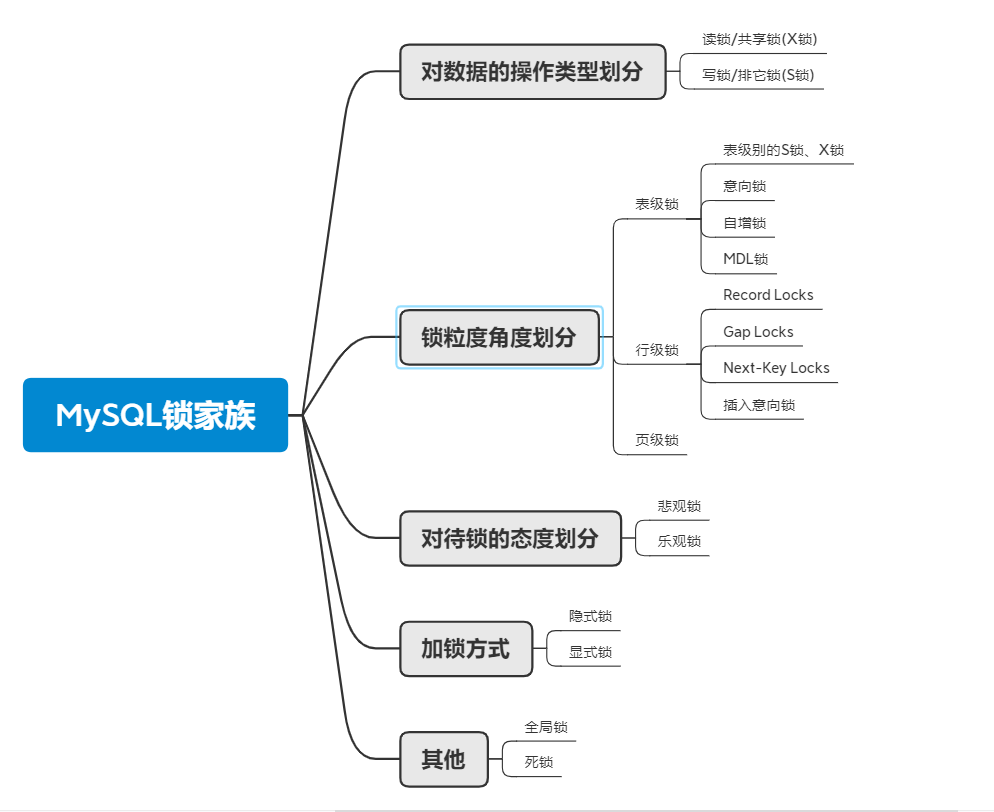


图 17 不同角度锁分类

#### 按数据操作<类型>划分

在使用加锁的方式解决问题时，由于既要允许读-读情况不受影响，又要使写-写、读-写或写-读情况中的操作相互阻塞，所以MySQL实现一个由两种类型的锁组成的锁系统来解决。这两种类型的锁通常被称为共享锁（Shared Lock,S Lock）和排他锁（Exclusive Lock,X Lock），也叫读锁(readlock)和写锁（write lock)。

**读锁**：也称为共享锁、英文用S表示。针对同一份数据，多个事务的读操作可以同时进行而不会互相影响，相互不阻塞的。

**写锁**：也称为排他锁、英文用X表示。当前写操作没有完成前，它会阻断其他写锁和读锁。这样就能确保在给定的时间里，只有一个事务能执行写入，并防止其他用户读取正在写入的同一资源。<注意：排他的意思是排除其他的读/写锁>

对于innodb而言，读锁与写锁可以加在表上(表级锁)，也可以加在行上(行级锁)。

实际上称为读锁与写锁不是很准确，正确来讲就是共享锁与排它锁。<因为读操作也可以有排他的行为，只是一般数据库不需要隔离这个级别(一个事务执行查询时不允许其他事务查询)>

**【读的锁定<select>】**

1. 添加共享锁(S)

SELECT …… LOCK IN SHARE MODE;

或：SELECT …… FOR SHARE; # 8.0新增语法

1. 添加排他锁(X)

SELECT ... FOR UPDATE;

注意：

不同事务之间不能在记录对象已经加锁的基础上添加排他锁，否则会发生发生阻塞。由于共享锁共享记录对象，所以可以多事务对记录加锁。

阻塞处理：

5.7版本存在有超时参数innodb\_lock\_wait\_timeout，超过指定时间就报错返回。

8.0新增了NOWAIT、SKIP LOCKED语法，在语句后添加后就会立即返回，不等待。

NOWAIT会立即报错返回。

SKIP LOCKED也会立即返回，只是返回的结果中不包含被锁定的行<表级锁不返回任何值>。

如果多个事务都是SELECT操作，那么就没有必要添加排它锁，毫无意义。

**【写的锁定DELETE、UPDATE、INSERT】**

直接添加排它锁即可。

**DELETE**：对一条记录做delete操作的过程其实是先在B+树中定位到这条记录的位置，然后获取这条记录的X锁，再执行delete mark操作。我们也可以把这个定位删除记录在B+树中位置的过程看成一个获取排它锁的读锁定。

**UPDATE**：在对一条记录做UPDATE操作分为三种情况。

情况1：未修改该记录的键值，并且被更新的列占用的存储空间在修改前后未发生变化。则先在B+树定位到这条记录的位置，然后再获取一下记录的X锁，最后在原纪录的位置进行修改操作。我们也可以把这个定位待修改记录在B+树中位置的过程看成是一个获取X锁的锁定读。

情况2：未修改该记录的键值，并且至少有一个被更新的列占用的存储空间在修改前后发生变化。则先在B+树中定位到这条记录的位置，然后获取一下记录的X锁，将该记录彻底删除掉(就是把记录彻底移入垃圾链表)，最后再插入一条新记录。这个定位待修改记录在B+树中位置的过程看成是一个获取X锁的锁定读，新插入的记录由insert操作提供的隐式锁进行保护。

情况3：修改了该记录的键值，则相当于在原纪录上做delete操作之后再来一次insert操作，加锁操作就需要按照delete和insert的规则进行了。

**INSERT**：一般情况下，新插入一条记录的操作并不加锁，通过一种称之为隐式锁的结构来保护这条新插入的记录在本事务提交前不被别的事务访问。

#### 按数据操作<粒度>划分

<表级锁、页级锁、行锁>

我们知道，锁定的对象范围越小，则数据库的并发度就越高。比如给一行添加锁与一个表添加锁不同，表级锁的并发度很小，效率就很小。然而锁是非常消耗资源的(获取、检查、释放等操作)，所以数据库系统需要在高并发响应与系统性能两个方面进行平衡。这样就产生了‘锁粒度’的概念。

**【表级锁】**

该锁会锁定整张表，它是MySQL中最基本的锁策略，并不依赖于存储引擎（不管你是MySQL的什么存储引擎，对于表锁的策略都是一样的），并且表锁是开销最小的策略（因为粒度比较大）。由于表级锁一次会将整个表锁定，所以可以很好的避免死锁问题。当然，锁的粒度大所带来最大的负面影响就是出现锁资源竞争的概率也会最高，导致并发率大打折扣。

* 表级的S锁、X锁

什么时候才会触发mysql的表级X锁？

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，InnoDB存储引擎是不会为这个表添加表级别的S锁或者X锁的。在对某个表执行一些诸如ALTER TABLE、DROP TABLE这类的DDL语句时，其他事务对这个表并发执行诸如SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞。同理，某个事务中对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，在其他会话中对这个表执行DDL语句也会发生阻塞。这个过程其实是通过在server层使用一种称之为元数据锁（英文名：Metadata Locks，简称MDL）结构来实现的。

一般情况下，不会使用InnoDB存储引擎提供的表级别的S锁和X锁。

什么情况下会用到表级锁？

在一些特殊情况下，比方说崩溃恢复过程中用到。

比如,在系统变量autocommit=0,innodb\_table\_locks=1时,手动获取InnoDB存储引擎提供的表级X/S锁。

表t1的S锁或者X锁可以这么写：

LOCK TABLES t1 READ：InnoDB存储引擎会对表t1加表级别的S锁。

LOCK TABLES t1 WRITE：InnoDB存储引擎会对表t1加表级别的X锁。

不过尽量避免在使用InnoDB存储引擎的表上使用LOCK TABLES这样的手动锁表语句，它们并不会提供什么额外的保护，只是会降低并发能力而已。InnoDB的厉害之处还是实现了更细粒度的行锁。



常用操作：

① 查看所有表的锁

show open tables [where in\_use = 1];

// 查看in\_use字段<1表示存在表锁>

② 释放表锁

unlock tables;

* 意向锁

InnoDB支持多粒度锁(特定场景下，允许行级锁与表级锁共存)<使用意向锁实现>。

意向锁是一种表锁。意向锁(表锁)不与行级锁冲突<与一般的表级锁区别>。

目的：为了解决并发事务判断是否存在行级锁的效率问题。

【意向锁解决了什么问题？】

在一般情况下，如果对一个表中添加行锁后，就不能对表添加表级锁。

那么出现一个问题，如果事务T1对某一行添加了一个行级排他锁，事务T2也想对该表添加一个行级排他锁，此时T2就会先遍历表中所有行判断是否存在排它锁，这样就导致性能很差。

而数据库系统使用意向锁解决这个问题。

如果我们给某一行数据加上了行级排它锁，数据库会自动给更大一级的空间，比如数据页或数据表加上意向锁，告诉其他人这个数据页或数据表已经有人上过排它锁了

这样当其他人想要获取数据表排它锁的时候，只需要了解是否有人已经获取了这个数据表的意向排他锁即可。

如果事务想要获得数据表中某些记录的共享锁，系统就在数据表上添加**意向共享锁**。

如果事务想要获得数据表中某些记录的排他锁，系统就在数据表上添加**意向排他锁**。

**意向共享锁(IS)**：事务有意向对表中的某些行加共享锁（S锁）

**意向排他锁(IX)**：事务有意向对表中的某些行加排他锁（X锁）

【注意】

意向锁是由存储引擎自己维护的，用户无法手动操作意向锁，在为数据行加共享/排他锁之前，InooDB会先获取该数据行所在数据表的对应意向锁。

【IS/IX意向锁与表级S/X冲突阻塞问题】

事务T1：

对某一行添加一个行级锁。<select \* form stu where id = 1001 for update;>

此时会自动对stu表添加一个意向排它锁(IX)。

事务T2：

对stu表添加排他表级锁X。< lock tables stu write;>

此时事务T2执行阻塞。

阻塞原因：

因为T1对某行添加了行级排他锁，也就是stu表被添加了意向排它锁。如果事务T2想添加一个排它锁或者共享锁(X/S)，都会发生锁冲突阻塞。

只有当T1添加的是行级共享锁时，T2添加标记共享锁是被允许的。



【IS/IX意向锁与IS/IX意向锁冲突阻塞问题】

只有一种情况会冲突。

事务T1：select \* form stu where id = 1001 for update;

事务T2：select \* form stu where id = 1001 for update;

事务T1、T2对同一行添加了行级排它锁，此时会发生阻塞。

如果是对不同行添加锁(X/S锁)，则自动创建的意向锁不会发生冲突阻塞。



* 自增锁<了解>

AUTO-INC锁是当向使用含有AUTO\_INCREMENT列的表中插入数据时需要获取的一种特殊的表级锁，在执行插入语句时就在表级别加一个AUTO-INC锁，然后为每条待插入记录的AUTO\_INCREMENT修饰的列分配递增的值，在该语句执行结束后，再把AUTO-INC锁释放掉。一个事务在持有AUTO-INC锁的过程中，其他事务的插入语句都要被阻塞，可以保证一个语句中分配的递增值是连续的。也正因为如此，其并发性显然并不高，当我们向一个有AUTO\_INCREMENT关键字的主键插入值的时候，每条语句都要对这个表锁进行竞争，这样的并发潜力其实是很低下的，所以innodb通过innodb\_autoinc\_lock\_mode的不同取值来提供不同的锁定机制，来显著提高SQL语句的可伸缩性和性能。

* 元数据锁(MDL锁)

MDL的作用是，保证读写的正确性。<数据库系统自动添加>

比如，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，增加了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，当对一个表做增删改查操作的时候，加**MDL读锁**；当要对表做结构变更操作的时候，加**MDL写锁**。

MDL读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性，解决了DML和DDL操作之间的一致性问题。不需要显式使用，在访问一个表的时候会被自动加上。

【行级锁】

行锁（Row Lock）也称为记录锁，顾名思义，就是锁住某一行（某条记录row）。需要注意的是，MySQL服务器层并没有实现行锁机制，行级锁只在存储引擎层实现。

优点：锁定力度小，发生所冲突的概率低，可以实现的并发度高。

缺点：对锁的开销比较大，加锁会比较慢，容易发生死锁。

* **记录锁**(Record locks)

记录锁的意思就是对单个记录进行加锁。官方的类型名称为：LOCK\_REC\_NOT\_GAP。

记录锁一样分为共享锁(S)和排它锁(X)。

当一个事务获取了一条记录的S型记录锁后，其他事务也可以继续获取该记录的S型记录锁，但不可继续获取X型记录锁。

当一个事务获取了一条记录的X型记录锁后，其他事务既不可以继续获取该记录的s型记录锁，也不可以继续获取X型记录锁。

例：<对student表加记录锁>

事务T1：

select \* from student where id = ‘1001’ for update; // 排它锁

事务T2：

update student set name=’zmq’ where id=’1001’ for share; // 阻塞

update student set name=’zmq’ where id=’1002’ for share; // 不阻塞

注意：

此处添加记录锁的方式实际上存在一些问题，临键锁时会提到。

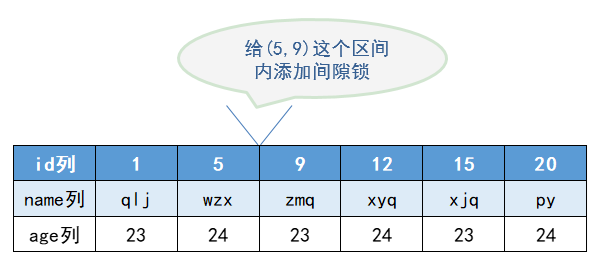
* **间隙锁**(Gap Locks)

在讲解‘[并发问题的解决办法](#_并发问题的解决方案)’时提到。

MySQL在可重复读隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用MVCC方案解决，地可以采用加锁方案解决。但是在使用加锁方案解决时有个大问题，就是事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些幻影记录加上记录锁。

InnoDB提出了一种称之为Gap Locks的间隙锁，官方的类型名称为：LOCK\_GAP，我们可以简称为gap锁。<可以将间隙锁理解为只影响其他事务进行insert操作。>

间隙锁可以完美的解决幻读的问题，就是为了防止插入幻读记录所提出的。



上图给你(5,9)添加了gap锁，意味着不允许别的事务在(5,9)区间内添加新的纪录，只有当锁被释放其他事务才可以插入。

注意：

区间是开区间，所以不影响已经存在的记录更新。

gap锁也可被区分为共享gap锁与独占gap锁，但是它们作用是相同的。也就是说，如果某条记录添加了间隙锁(无论是共享/独占)，并不会限制其他事务对这条记录继续加记录锁或gap锁。

如何添加gap锁？

① select \* from student where id = 5 for share; # 添加记录锁

② select \* from student where id = 6 for share; # 添加间隙锁

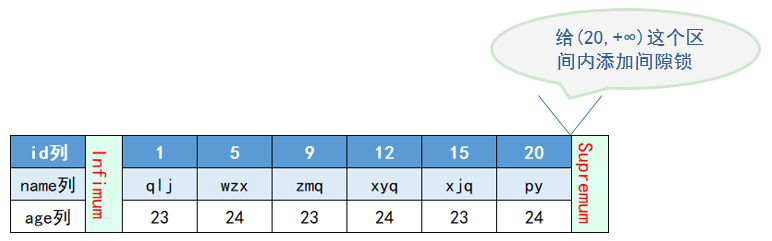
③ select \* from student where id = 7 for update; # 添加间隙锁

注意：②与③不冲突，并且作用相同。

问题一：存在于表头、表尾记录的前与后如何添加间隙锁。

infimum记录，表示该页面中最小的记录。

supremum记录，表示该页面中最大的记录。



例：

select \* from student where id=21 for share;

或 select \* from student where id>21 for share;

问题二：间隙锁容易发生死锁

例：



事务A与事务B互相等待锁的释放，导致死锁产生。

解决死锁：

策略一：直接进入等待，直到超时。超时通过innodb\_lock\_wait\_timeout控制。

策略二：发起死锁检测，主动回滚死锁链条的某个事务(将持有最少行级排它锁的事务回滚)，其他事务继续执行。(mysql默认使用该策略)

* **临键锁**(Next-Key Locks)

作用：既可以锁住某条记录，又可以阻止其他事务在该条记录之前的间隙insert新记录。

官方名称：LOCK\_ORDINARY。可以简称next-key锁

范围：在存储引擎innodb、事务级别时可重复读的情况下使用。

innodb默认的锁就是next-key锁。

next-key锁本质上就是一个记录锁和gap锁的合体。

注意：

我们知道innodb默认使用next-key锁，所以我们对某一记录添加记录锁时，实际是添加了一个临键锁。所以上面讲的记录锁在实现时对间隙也加了锁。

使用next-key锁：

select \* from student where id<=9 and id>5 for update; #不用

select \* from student where id=9 for update; # mysql默认使用临键锁

在id=9上添加了记录锁，在(5,9)间加了间隙锁。

注意：id=9是一个排他记录锁，所以其他事务不能对id=9的记录添加S/X锁。

* **插入意向锁**(Insert Intention Locks)

由于gap锁的原因，事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了gap锁（next-key锁也包含gap锁），如果有的话，插入操作需要等待，直到拥有gap锁的那个事务提交。

InnoDB规定事务在等待的时候也需要在内存中生成一个锁结构。

这个insert时等待的锁就是插入意向锁。

名称：LOCK\_INSERT\_INTENTION <插入意向锁>

本质：插入意向锁实际上是一个gap锁(间隙锁)。<并不是表锁中提到的意向锁>

作用：该锁用以表示插入意向，当多个事务在同一区间（gap）、插入位置不同的多条数据时，事务之间不需要互相等待。

注意：

1、插入意向锁是由insert操作阻塞所产生的一种间隙锁。

2、由于插入意向锁时gap锁，所以它们之间互不排斥，即使多个事务在同一区间内插入多条记录，只要记录本身不冲突(主键、唯一索引)，那么事务之间就不会发生阻塞等待。

实现：

事务T1：

select \* from student where id=7 for share; # 间隙锁(5,9)阻塞

事务T2：

insert into student values(8,’lb’,26); #insert发生阻塞，生成插入意向锁

事务T3：

insert into student values(6,’ll’,23); #insert发生阻塞，生成插入意向锁

rollback/commit事务T1，间隙锁被释放，T2与T3的insert就会立即执行，并且两个插入意向锁不会发生互相阻塞死锁的现象。

<两个insert主键、唯一索引必须不同>

故：

插入意向锁不会阻止别的事务继续获取该记录上任何类型的锁。

【页锁】

页锁就是在页的粒度上进行锁定，锁定的数据资源比行锁要多，因为一个页中可以有多个行记录。当我们使用页锁的时候，会出现数据浪费的现象，但这样的浪费最多也就是一个页上的数据行。页锁的开销介于表锁和行锁之间，会出现死锁。锁定粒度介于表锁和行锁之间，并发度一般。

【补充】

每个层级的锁数量是有限制的，因为锁会占用内存空间，锁空间的大小是有限的。当某个层级的锁数量超过了这个层级的阈值时，就会进行锁升级。锁升级就是用更大粒度的锁替代多个更小粒度的锁，比如InnoDB中行锁升级为表锁，这样做的好处是占用的锁空间降低了，但同时数据的并发度也下降了。

#### 按对待锁的<态度>划分

从对待锁的态度来看的话，可以将锁分为乐观锁与悲观锁，从名字中也可以看出这两种锁是两种看待数据并发的思维方式。需要注意的是，乐观锁与悲观锁不是实际的锁，而是锁的设计思想。

* 悲观锁

悲观锁总是将事情往最坏的情况考虑，每次拿数据的时候认为别人会修改，所以每次都会对数据上锁，这样别人用数据时就会发生阻塞挂起，直到释放锁后才会继续。(共享资源每次只给一个线程，其它线程阻塞，用完后再把资源转让给其它线程)。

java中synchronized和ReentranLock等独占锁就是悲观锁思想实现的。

注意：

select …… for update; (悲观锁)执行时会对所有扫描到的数据进行上锁，就算在最终显示数据的时候没有显示这些数据，但如果遍历表时，遍历到了就会加锁。所以，在使用这种方式加锁的时候一定要配合索引，加上索引后，遍历的数据量就会大大减少，所以添加的锁也就很少。

悲观锁在高并发的场景下效率很慢，并且长事务的情况下，开销也很大。

* 乐观锁

乐观锁认为在同一数据的并发操作不会总发生，这是一个小概率事件，不用每次都对数据上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，也就是不采用数据库自身的锁机制，而是通过程序来实现。在程序上，我们可以采用版本号机制或者CAS机制实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量。<因为读并发不影响>

java中java.util.concurrent.atomic包下的原子变量类就是使用了乐观锁的一种实现方式：CAS实现。

【乐观锁的版本号机制】

在表中设计一个版本字段version，第一次读的时候，会获取version字段的取值。然后对数据进行更新或删除操作时，会执行UPDATE … SET version=version+1 WHERE version=version。此时如果已经有事务对这条数据进行了更改，修改就不会成功。

这种方式类似我们熟悉的SVN、CVS版本管理系统，当我们修改了代码进行提交时，首先会检查当前版本号与服务器上的版本号是否一致，如果一致就可以直接提交，如果不一致就需要更新服务器上的最新代码，然后再进行提交。



事务B提交commit失败的原因：

# commit报错，在commit时会检测当前表中数据对应的version是否依然是当前事务修改前的值。

如果当前commit时，表中该记录的version+1等于当前事务修改version后的值，则就可以commit成功；

如果当前commit时表中该记录的version值等于或大于当前事务修改version后的值，则记录修改失败。

【乐观锁时间戳机制】

时间戳和版本号机制一样，也是在更新提交的时候，将当前数据的时间戳和更新之前取得的时间戳进行比较，如果两者一致则更新成功，否则就是版本冲突。

你能看到乐观锁就是程序员自己控制数据并发操作的权限，基本是通过给数据行增加一个戳（版本号或者时间戳），从而证明当前拿到的数据是否最新。

注意：

如果数据表是读写分离的表，当master(主)表中写入的数据没有及时同步到slave(从)表中时，会造成更新一直失效的问题。(一般主表时写操作，从表是读操作)

此时需要强制读取master表中的数据(即将select语句放到事务中即可，这时候查询的就是master主表了)

* 小结

1、乐观锁适合读操作多的场景，相对来说写的操作比较少。它的优点在于程序实现，不存在死锁问题，不过适用场景也会相对乐观，因为它阻止不了除了程序以外的数据库操作。

2、悲观锁适合写操作多的场景，因为写的操作具有排它性。采用悲观锁的方式，可以在数据库层面阻止其他事务对该数据的操作权限，防止读-写和写-写的冲突。

#### 按照加锁的<方式>划分

* 隐式锁

在我们前面学习的insert操作时。如果即将插入的记录所在的间隙已经被其他事务加了gap锁，那么本次INSERT操作会阻塞，并且当前事务会在该间隙上加一个插入意向锁，否则一般情况下INSERT操作是不加锁的。

如果一个事务首先插入了一条记录（此时并没有在内存生产与该记录关联的锁结构，并且事务没有结束），然后另一个事务：

# 立即使用SELECT … FOR SHARE语句读取这条记录，也就是要获取这条记录的S锁，或者使用SELECT … FOR UPDATE语句读取这条记录，也就是要获取这条记录的X锁，怎么办？如果允许这种情况的发生，那么可能产生脏读问题。

# 立即修改这条记录，也就是要获取这条记录的X锁，怎么办？如果允许这种情况的发生，那么可能产生脏写问题

这时候事务id又要起作用了(在锁结构中存有事务trx\_id与[is\_waiting](#_写-写))。把聚簇索引和二级索引中的记录分开看一下：

情景一：

对于聚簇索引记录来说，有一个trx\_id隐藏列，该隐藏列记录着最后改动该记录的事务id。那么如果在当前事务中新插入一条聚簇索引记录后，该记录的trx\_id隐藏列代表的就是当前事务的事务id，如果其他事务此时想对该记录添加S锁或者X锁时，首先会看一下该记录的trx\_id隐藏列代表的事务是否是当前的活跃事务，如果是的话，那么就帮助当前事务创建一个X锁（也就是为当前事务创建一个锁结构，is\_waiting属性是false)，然后自己进入等待状态(也就是为自己也创建一个锁构,is\_waiting属性是true）。<结合锁结构对记录加锁过程来看>

情景二：

对于二级索引记录来说，本身并没有trx\_id隐藏列，但是在二级索引页面的Page Header部分有一个PAGE\_MAX\_TRX\_ID属性，该属性代表对该页面做改动的最大的事务id,如果PAGE\_MAX\_TRX\_ID属性值小于当前最小的活跃事务id，那么说明对该页面做修改的事务都已经提交了，否则就需要在页面中定位到对应的二级索引记录，然后回表找到它对应的聚簇索引记录，然后再重复情景一的做法。

对隐式锁的理解：

一个事务对新插入的记录可以不显式的加锁（生成一个锁结构），但是由于事务id的存在，相当于加了一个隐式锁。也就是说，我们新的insert记录本身没有任何锁，但是由于insert操作本身存在与事务中，所以隐式的存在一个锁，这个隐式锁只代表当前新的insert记录还处于事务当中，是一个活跃状态。其他事务在对这条记录加S锁或者X锁时，由于隐式锁的存在，会先帮助当前事务生成一个锁结构(is\_ waiting=false)，然后自己再生成一个锁结构后进入等待状态(is\_ waiting=true)。隐式锁是一种延迟加锁的机制，从而来减少加锁的数量。

隐式锁在实际内存对象中并不含有这个锁信息。只有当产生锁等待时，隐式锁转化为显式锁。

如何查看是否存在锁：

SELECT \* FROM performance\_schema.data\_locks;

* 显式锁

能够显示查看的锁就是显式锁。

显示加共享锁：

select …… for share;

显示加排它锁：

select …… for update;

#### 其他锁-全局锁

全局锁就是对整个数据库实例加锁。当你需要让整个库处于只读状态的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。全局锁的典型使用场景是：做全库逻辑备份。

全局锁指令：

Flush tables with read lock;

#### 其他锁-死锁

指两个或多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环。

<间隙锁中有提到>



* 死锁产生的四大条件

① 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求该资源，则请求者只能等待，直至占有该资源的进程用毕释放。

② 请求与保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源又已被其它进程占有，此时请求进程阻塞，且对自己已获得的其它资源保持不放

③ 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

④ 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链。即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

* 如何避免死锁

① 破坏互斥条件

互斥条件是由设备系统的固有特性决定的，不能破坏，应保持。

② 破坏请求与保持条件

规定所有进程在开始运行之前，都必须一次性地申请其在整个运行过程所需的全部资源。此时，若系统有足够的资源分配给某进程，便可把其需要的所有资源分配给该进程，这样，该进程在整个运行期间便不会再提出资源要求，从而摒弃了请求条件。但在分配资源时，只要有一种资源不能满足某进程的要求，即使其它所需的各资源都空闲，也不分配给该进程，而让该进程等待。由于在该进程的等待期间，它并未占有任何资源，因而也摒弃了保持条件，从而可以避免发生死锁。

缺点：

资源严重浪费；进程延迟运行。

③ 破坏不剥夺条件

规定进程是逐个地提出对资源的要求的。当一个已经保持了某些资源的进程，再提出新的资源请求而不能立即得到满足时，必须释放它已经保持了的所有资源，待以后需要时再重新申请。这意味着某一进程已经占有的资源，在运行过程中会被暂时地释放掉，也可认为是被剥夺了，从而摒弃了“不剥夺”条件。

缺点：

释放已保持资源可能会导致前面的工作失效；

反复的申请和释放资源，延长进程运行时间，增加系统开销，降低系统吞吐量

④ 破坏环路等待条件

规定将资源按照一定规则进行排序，不管哪些进程并发时都按照顺序使用资源，这样就不会出现环路的情况。

相比前面两种策略，这种策略明显较好，但依然会有一些缺点。

* 其他解决死锁方案

① 方式一

等待超时策略。(innodb\_lock\_wait\_timeout=50s)

当两个事务相互等待时，如果一个事务等待时间超过设置的阈值，将将其回滚，另外一个事务就可以继续执行。

缺点：对于在线服务，不能让进程长时间等待。

② 方式二

使用死锁检测进行死锁处理。

innodb提供了wait-for graph算法，主动进行死锁检测，每当加锁请求无法立即满足需要并进入等待时，wait-for grap算法就会被触发。

死锁检测的原理是构建一个以事务为顶点，锁为边的有向图，判断有向图中是否存在圈(图论)，存在即为死锁。

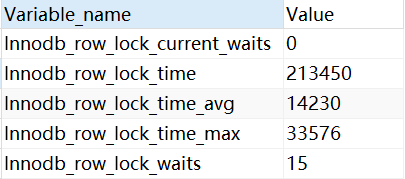
### 锁结构

了解即可，进入学习网站学习了解。

### 锁监控

mysql锁的监控一般通过检查InnoDB\_row\_lock等状态变量来分析系统上的行锁的争夺情况。

show status like 'innodb\_row\_lock%';



解释：

Innodb\_row\_lock\_current\_waits：当前正在等待锁定的数量；

Innodb\_row\_lock\_time：从系统启动到现在锁定总时间长度；(等待总时长)

Innodb\_row\_lock\_time\_avg：每次等待所花平均时间；(等待平均时长)

Innodb\_row\_lock\_time\_max：从系统启动到现在等待最长的一次所花时间；

Innodb\_row\_lock\_waits：系统启动后到现在总共等待的次数；(等待总次数)

查看事务和锁的信息记录：

mysql5.7及之前。使用INFORMATION\_SCHEMA.INNODB\_LOCKS查看事务的锁情况，但只能看到阻塞事务的锁；如果未被阻塞在INNODB\_LOCKS表中就看不到该事务锁的情况。

mysql8.0删除INFORMATION\_SCHEMA.INNODB\_LOCKS，添加performance\_schema.data\_locks，通过performance\_schema.data\_locks查看事务锁的情况，它与mysql5.7不同，既可以看阻塞的锁情况，也可以看未被阻塞的锁情况。

例：select performance\_schema.data\_locks;

## 多版本并发控制(MVCC)

### 概念