# 14.MySQL高级开发下(MySQL8.0新特性)√

## 1.MySQL8.0新特性–json支持

json在mysql数据库中是一种特殊数据类型，可以存储大对象。

### 1.1 创建有json字段的表

```bash
info列可以插入关于json格式的列信息
mysql> CREATE TABLE t_json(id INT PRIMARY KEY, sname VARCHAR(20) , info JSON);
```

### 1.2 插入json记录

```bash
0.函数调用
0.1插入含有json数组的记录
mysql> INSERT INTO t_json(id,sname,info) VALUES( 1, 'name1', JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME()));
0.2 插入含有json对象的记录
mysql> INSERT INTO t_json(id,sname,info) VALUES( 2, 'name2', JSON_OBJECT("age", 20, "time", now()));
1.json原格式
mysql> INSERT INTO t_json(id,sname,info) VALUES( 3, 'name3', '{"age":20, "time":"2020-02-14 10:52:00"}');

```

### 1.3 查询json记录

```
0.查看t_json全表信息。
mysql> select * from t_json;
+----+-------+-------------------------------------------------+
| id | sname | info                                            |
+----+-------+-------------------------------------------------+
|  1 | name1 | [1, "abc", null, true, "15:35:05.000000"]       |
|  2 | name2 | {"age": 20, "time": "2021-04-15 15:42:22.000000"} |
|  3 | name3 | {"age": 20, "time": "2020-02-14 10:52:00"}      |
+----+-------+-------------------------------------------------+
1.查询json格式中的列信息
mysql> SELECT sname,JSON_EXTRACT(info,'$.age') FROM t_json;
+-------+-----------------------------+
| sname | JSON_EXTRACT(info,'$.age') |
+-------+-----------------------------+
| name1 | NULL                        |
| name2 | 20                          |
| name3 | 20                          |
+-------+-----------------------------+
2.查询json格式中列的列信息
mysql> SELECT sname,info->'$.age' FROM t_json;
+-------+---------------+
| sname | info->'$.age' |
+-------+---------------+
| name1 | NULL          |
| name2 | 20            |
| name3 | 20            |
+-------+---------------+
3.查询json格式中的列
mysql> SELECT id,json_keys(info) FROM t_json;
+----+-----------------+
| id | json_keys(info) |
+----+-----------------+
|  1 | NULL            |
|  2 | ["age", "time"] |
|  3 | ["age", "time"] |
+----+-----------------+
```

## 1.4 修改json记录

```
-- 增加键值（如果有键值列改变键值，没有插入键值）
UPDATE t_json SET info = json_set(info,'$.ip','192.168.1.1') WHERE id = 2;
-- 变更键值
UPDATE t_json SET info = json_set(info,'$.ip','192.168.1.2') WHERE id = 2;
-- 删除键值
UPDATE t_json SET info = json_remove(info,'$.ip') WHERE id = 2;
```

# 2.MySQL8.0新特性–窗口函数

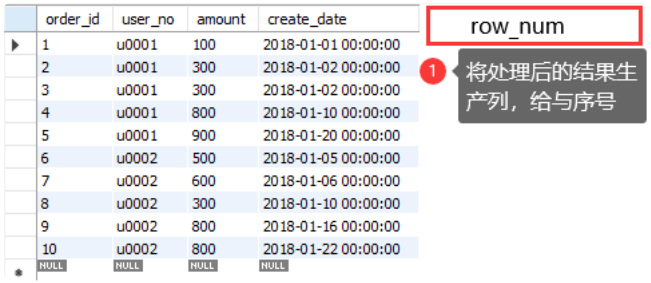
```
0.模拟环境创建表，插入数据。
create table order_info ( order_id int primary key, user_no varchar(10), amount int, create_date datetime );

insert into order_info values (1,'u0001',100,'2018-1-1'); insert into order_info values (2,'u0001',300,'2018-1-2'); insert into order_info values (3,'u0001',300,'2018-1-2'); insert into order_info values (4,'u0001',800,'2018-1-10'); insert into order_info values (5,'u0001',900,'2018-1-20'); insert into order_info values (6,'u0002',500,'2018-1-5'); insert into order_info values (7,'u0002',600,'2018-1-6'); insert into order_info values (8,'u0002',300,'2018-1-10'); insert into order_info values (9,'u0002',800,'2018-1-16'); insert into order_info values (10,'u0002',800,'2018-1-22');
1.查看创建的模拟环境表
mysql>  select * from order_info;
+----------+---------+--------+---------------------+
| order_id | user_no | amount | create_date         |
+----------+---------+--------+---------------------+
|        1 | u0001   |    100 | 2018-01-01 00:00:00 |
|        2 | u0001   |    300 | 2018-01-02 00:00:00 |
|        3 | u0001   |    300 | 2018-01-02 00:00:00 |
|        4 | u0001   |    800 | 2018-01-10 00:00:00 |
|        5 | u0001   |    900 | 2018-01-20 00:00:00 |
|        6 | u0002   |    500 | 2018-01-05 00:00:00 |
|        7 | u0002   |    600 | 2018-01-06 00:00:00 |
|        8 | u0002   |    300 | 2018-01-10 00:00:00 |
|        9 | u0002   |    800 | 2018-01-16 00:00:00 |
|       10 | u0002   |    800 | 2018-01-22 00:00:00 |
+----------+---------+--------+---------------------+
```

## 2.1 row_number()

### 2.1.1 语法

```
row_number()over(partition by user_no order by create_date desc) as row_num
窗口函数row_number()over
partition by（对那个列进行分组，没有指定就是作用于全表）
order by 对那个列进行排序
最后将结果生成新的别名列（标记序号）
```




```
mysql>  select row_number()over(partition by user_no order by create_date desc) as row_num, order_id,user_no,amount,create_date from order_info;
+---------+----------+---------+--------+---------------------+
| row_num | order_id | user_no | amount | create_date         |
+---------+----------+---------+--------+---------------------+
|       1 |        5 | u0001   |    900 | 2018-01-20 00:00:00 |
|       2 |        4 | u0001   |    800 | 2018-01-10 00:00:00 |
|       3 |        2 | u0001   |    300 | 2018-01-02 00:00:00 |
|       4 |        3 | u0001   |    300 | 2018-01-02 00:00:00 |
|       5 |        1 | u0001   |    100 | 2018-01-01 00:00:00 |
|       1 |       10 | u0002   |    800 | 2018-01-22 00:00:00 |
|       2 |        9 | u0002   |    800 | 2018-01-16 00:00:00 |
|       3 |        8 | u0002   |    300 | 2018-01-10 00:00:00 |
|       4 |        7 | u0002   |    600 | 2018-01-06 00:00:00 |
|       5 |        6 | u0002   |    500 | 2018-01-05 00:00:00 |
+---------+----------+---------+--------+---------------------+
```

### 2.1.2 案列

Bash | Copy

```
查询每个用户的第一笔订单 from+ 子查询（窗口函数）
select * from
( select row_number()over(partition by user_no order by create_date desc) as row_num, order_id,user_no,amount,create_date from order_info )t （每个派生出来的表都要有表名）
where row_num=1;
+---------+----------+---------+--------+---------------------+
| row_num | order_id | user_no | amount | create_date         |
+---------+----------+---------+--------+---------------------+
|       1 |        5 | u0001   |    900 | 2018-01-20 00:00:00 |
|       1 |       10 | u0002   |    800 | 2018-01-22 00:00:00 |
+---------+----------+---------+--------+---------------------+

```

## 2.2 rank()

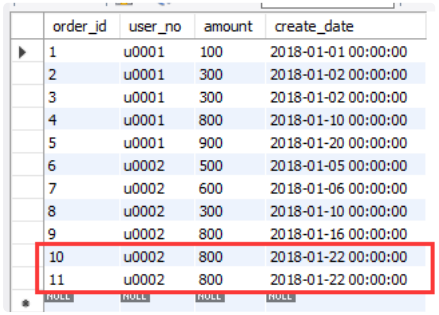
### 2.2.1 简介

> 类似于 row_number()，也是排序功能。
> 但是当插入于原表完全一样的数据时，row_number()编号，会将相同的数据编号不同的序号
> rank()编号会将会将相同的数据编号相同的序号，解决了row_number对于这种情况的问题。

### 2.2.2 案列

```
1.插入相同的数据
insert into order_info values (11,'u0002',800,'2018-1-22');
```

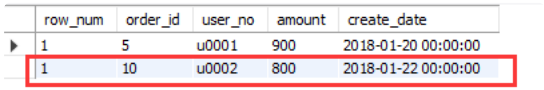
查看原表数据，同用户，同时间内有两笔订单。



| order_id | user_no | amount | create_date |
|---|---|---|---|
| 1 | u0001 | 100 | 2018-01-01 00:00:00 |
| 2 | u0001 | 300 | 2018-01-02 00:00:00 |
| 3 | u0001 | 300 | 2018-01-02 00:00:00 |
| 4 | u0001 | 800 | 2018-01-10 00:00:00 |
| 5 | u0001 | 900 | 2018-01-20 00:00:00 |
| 6 | u0002 | 500 | 2018-01-05 00:00:00 |
| 7 | u0002 | 600 | 2018-01-06 00:00:00 |
| 8 | u0002 | 300 | 2018-01-10 00:00:00 |
| 9 | u0002 | 800 | 2018-01-16 00:00:00 |
| 10 | u0002 | 800 | 2018-01-22 00:00:00 |
| 11 | u0002 | 800 | 2018-01-22 00:00:00 |

但是row_number()查询u002用户的第一笔订单只显示一个

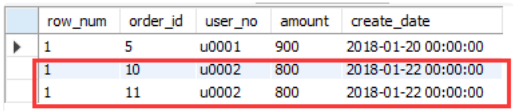
```
select * from
( select row_number()over(partition by user_no order by create_date desc) as row_num,
order_id,user_no,amount,create_date from order_info )t (每个派生出来的表都要有表名)
where row_num=1;
```



| row_num | order_id | user_no | amount | create_date |
|---|---|---|---|---|
| 1 | 5 | u0001 | 900 | 2018-01-20 00:00:00 |
| 1 | 10 | u0002 | 800 | 2018-01-22 00:00:00 |

我们用rank()就会把同一时间u002用户的所有订单显示出来

```
select * from
( select rank()over(partition by user_no order by create_date desc) as row_num,
order_id,user_no,amount,create_date from order_info )t (每个派生出来的表都要有表名)
where row_num=1;
```



| row_num | order_id | user_no | amount | create_date |
|---|---|---|---|---|
| 1 | 5 | u0001 | 900 | 2018-01-20 00:00:00 |
| 1 | 10 | u0002 | 800 | 2018-01-22 00:00:00 |
| 1 | 11 | u0002 | 800 | 2018-01-22 00:00:00 |

## 2.3 dense_rank()

### 2.3.1 简介

> dense_rank()的出现是为了解决rank()编号存在的问题的，
> rank()编号的时候存在跳号的问题，如果有两个并列第1，那么下一个名次的编号就是3，结果就是
> 没有编号为2的数据。
> 如果不想跳号，可以使用dense_rank()替代。

### 2.3.2 案列

**我们用rank()函数显示的是**

```
select rank()over(partition by user_no order by create_date desc) as row_num,
order_id,user_no,amount,create_date from order_info
```

| row_num | order_id | user_no | amount | create_date |
|---|---|---|---|---|
| 1 | 5 | u0001 | 900 | 2018-01-20 00:00:00 |
| 2 | 4 | u0001 | 800 | 2018-01-10 00:00:00 |
| 3 | 2 | u0001 | 300 | 2018-01-02 00:00:00 |
| 3 | 3 | u0001 | 300 | 2018-01-02 00:00:00 |
| 5 | 1 | u0001 | 100 | 2018-01-01 00:00:00 |
| 1 | 10 | u0002 | 800 | 2018-01-22 00:00:00 |
| 1 | 11 | u0002 | 800 | 2018-01-22 00:00:00 |
| 3 | 9 | u0002 | 800 | 2018-01-16 00:00:00 |
| 4 | 8 | u0002 | 300 | 2018-01-10 00:00:00 |
| 5 | 7 | u0002 | 600 | 2018-01-06 00:00:00 |
| 6 | 6 | u0002 | 500 | 2018-01-05 00:00:00 |

**我们用dense_rank()函数显示的是**

```
select dense_rank()over(partition by user_no order by create_date desc) as row_num, order_id,user_no,amount,cr
eate_date from order_info
```

| row_num | order_id | user_no | amount | create_date |
|---|---|---|---|---|
| 1 | 5 | u0001 | 900 | 2018-01-20 00:00:00 |
| 2 | 4 | u0001 | 800 | 2018-01-10 00:00:00 |
| 3 | 2 | u0001 | 300 | 2018-01-02 00:00:00 |
| 3 | 3 | u0001 | 300 | 2018-01-02 00:00:00 |
| 4 | 1 | u0001 | 100 | 2018-01-01 00:00:00 |
| 1 | 10 | u0002 | 800 | 2018-01-22 00:00:00 |
| 1 | 11 | u0002 | 800 | 2018-01-22 00:00:00 |
| 2 | 9 | u0002 | 800 | 2018-01-16 00:00:00 |
| 3 | 8 | u0002 | 300 | 2018-01-10 00:00:00 |
| 4 | 7 | u0002 | 600 | 2018-01-06 00:00:00 |
| 5 | 6 | u0002 | 500 | 2018-01-05 00:00:00 |

## 2.4 lag以及lead

### 2.4.1 简介

> lag(column,n)获取当前数据行按照某种排序规则的上n行数据的某个字段，
> lead(column,n)获 取当前数据行按照某种排序规则的下n行数据的某个字段.
> 举个实际例子，按照时间排序，获取当前订单的上一笔订单发生时间和下一笔订单发生时间，（可以计算订单的时间上的间隔度或者说买买买的频繁程度）

### 2.4.2 列子

Bash | Copy

```
select order_id, user_no, amount, create_date,
lag(create_date,1) over (partition by user_no order by create_date asc) 'last_transaction_time',
lead(create_date,1) over (partition by user_no order by create_date asc) 'next_transaction_time'
from order_info ;
```

| order_id | user_no | amount | create_date | last_transaction_time | next_transaction_time |
|---|---|---|---|---|---|
| 1 | u0001 | 100 | 2018-01-01 00:00:00 | NULL | 2018-01-02 00:00:00 |
| 2 | u0001 | 300 | 2018-01-02 00:00:00 | 2018-01-01 00:00:00 | 2018-01-02 00:00:00 |
| 3 | u0001 | 300 | 2018-01-02 00:00:00 | 2018-01-02 00:00:00 | 2018-01-10 00:00:00 |
| 4 | u0001 | 800 | 2018-01-10 00:00:00 | 2018-01-02 00:00:00 | 2018-01-20 00:00:00 |
| 5 | u0001 | 900 | 2018-01-20 00:00:00 | 2018-01-10 00:00:00 | NULL |
| 6 | u0002 | 500 | 2018-01-05 00:00:00 | NULL | 2018-01-06 00:00:00 |
| 7 | u0002 | 600 | 2018-01-06 00:00:00 | 2018-01-05 00:00:00 | 2018-01-10 00:00:00 |
| 8 | u0002 | 300 | 2018-01-10 00:00:00 | 2018-01-06 00:00:00 | 2018-01-16 00:00:00 |
| 9 | u0002 | 800 | 2018-01-16 00:00:00 | 2018-01-10 00:00:00 | 2018-01-22 00:00:00 |
| 10 | u0002 | 800 | 2018-01-22 00:00:00 | 2018-01-16 00:00:00 | 2018-01-22 00:00:00 |
| 11 | u0002 | 800 | 2018-01-22 00:00:00 | 2018-01-22 00:00:00 | NULL |

## 2.5 CTE（公用表表达式）

CTE有两种用法，非递归的CTE和递归的CTE。

### 2.5.1 非递归CTE

#### 简介

> 非递归的CTE可以用来增加代码的可读性，增加逻辑的结构化表达。
> 平时我们比较痛恨一句sql几十行甚至上上百行，根本不知道其要表达什么，难以理解，对于这种SQL，可以使用CTE分段解决.
> 比如逻辑块A做成一个CTE，逻辑块B做成一个CTE，然后在逻辑块A和逻辑块B的基础上继续进行查询，这样与直接一句代码实现整个查询，逻辑上就变得相对清晰直观。

#### 举例

```
要求：查询每个用户的最新一条订单.
第一步是对用户的订单按照时间排序编号，做成一个CTE,
第二步对上面的CTE查询，取行号等于1的数据。
mysql> with cte as
    ->  (
    -> select row_number()over(partition by user_no order by create_date desc) as row_num, order_id,user_no,ar
    -> )
    -> select * from cte where row_num=1;
+---------+----------+---------+--------+---------------------+
| row_num | order_id | user_no | amount | create_date         |
+---------+----------+---------+--------+---------------------+
|       1 |        5 | u0001   |    900 | 2018-01-20 00:00:00 |
|       1 |       10 | u0002   |    800 | 2018-01-22 00:00:00 |
+---------+----------+---------+--------+---------------------+
```

## 2.5.2 递归CTE

### 简介

> 递归处理每一行数据，再将处理的结果集于下一次处理的行数合并处理。

### 举例

Bash | Copy

```
要求：查询公司的组织架构数据，查询管理层级。
0.创建模拟环境表，插入数据。
CREATE TABLE emp(
id INT PRIMARY KEY NOT NULL,
name VARCHAR(100) NOT NULL,
manager_id INT NULL);

INSERT INTO emp VALUES (333, "总经理", NULL),
(198, "副总1", 333), (692, "副总2", 333),
(29, "主任1", 198),
(4610, "职员1", 29),
(72, "职员2", 29),
(123, "主任2", 692);
1.查看模拟表
mysql> SELECT * FROM world.emp;
+------+-----------+------------+
| id   | name      | manager_id |
+------+-----------+------------+
|   29 | 主任1     |        198 |
|   72 | 职员2     |         29 |
|  123 | 主任2     |        692 |
|  198 | 副总1     |        333 |
|  333 | 总经理    |       NULL |
|  692 | 副总2     |        333 |
| 4610 | 职员1     |         29 |
+------+-----------+------------+

2.递归ETC
WITH RECURSIVE test(id, name, path) AS (
SELECT id, name, CAST(id AS CHAR(200))
FROM emp WHERE manager_id IS NULL
UNION ALL
SELECT e.id, e.name, CONCAT(ep.path, ',', e.id)
FROM test AS ep JOIN emp AS e ON ep.id = e.manager_id
)
SELECT * FROM test ORDER BY path;
```

| id | name | path |
|---|---|---|
| 333 | 总经理 | 333 |
| 198 | 副总1 | 333,198 |
| 29 | 主任1 | 333,198,29 |
| 4610 | 职员1 | 333,198,29,4610 |
| 72 | 职员2 | 333,198,29,72 |
| 692 | 副总2 | 333,692 |
| 123 | 主任2 | 333,692,123 |

# 3.MySQL的分区表应用

实例：

```
CREATE TABLE part_tab
(id INT DEFAULT NULL,
remark VARCHAR(50) DEFAULT NULL,
d_date DATE DEFAULT NULL
)ENGINE=InnoDB
PARTITION BY RANGE(YEAR(d_date))(
PARTITION p0 VALUES LESS THAN(1995),
PARTITION p1 VALUES LESS THAN(1996),
PARTITION p2 VALUES LESS THAN(1997),
PARTITION p3 VALUES LESS THAN(1998),
PARTITION p4 VALUES LESS THAN(1999),
PARTITION p5 VALUES LESS THAN(2000),
PARTITION p6 VALUES LESS THAN(2001),
PARTITION p7 VALUES LESS THAN(2002),
PARTITION p8 VALUES LESS THAN(2003),
PARTITION p9 VALUES LESS THAN(2004),
PARTITION p10 VALUES LESS THAN maxvalue);
```

应用场景：

1. 对于那些已经失去保存意义的数据，通常可以通过删除与那些数据有关的分区，很容易地删除那些数据。相反地，在某些情况下，添加新数据的过程又可以通过为那些新数据专门增加一个新的分区，来很方便地实现。
2. 与单个磁盘或文件系统分区相比，可以存储更多的数据。
3. 一些查询可以得到极大的优化，这主要是借助于满足一个给定WHERE语句的数据可以只保存在一个或多个分区内，这样 在查找时就不用查找其他剩余的分区。因为分区可以在创建了分区表后进行修改，所以在第一次配置分区方案时还不曾这么做时，可以重新组织数据，来提高那些常用查询的效率。
4. 涉及到例如SUM()和COUNT()这样聚合函数的查询，可以很容易地进行并行处理。通过“并行”，这意味着该查询可以在每个分区上同时进行，最终结果只需通过总计所有分区得到的结果。
5. 通过跨多个磁盘来分散数据查询，来获得更大的查询吞吐量。

局限：

1. 适合开发周期短，成本低的项目，不建议大型项目使用。 /*许多大型项目基本采用按照实际业务逻辑，进行建立年库，分表*/
2. 分区表无法创建全文索引/*所以使用，就得结合自己的使用场景，是否涉要涉及到全文检索*/
3. 分区表无法使用外键约束
4. 一个表5.6之后支持8192分区

json%E6%94%AF%E6%8C%81json%E5%9C%A8mysql%E6%95%B0%E6%8D%AE%E5%BA%93%E4%B8%AD%E6%98%AF%E4%B8%80%E7%A7%8D%E7%89%B9%E6%AE%8A%E6%95%B0%E6%8D%AE%E