MySQL高手系列

第1篇：MySQL的一些基础知识

这是mysql系列第1篇。

本文主要内容

1. 背景介绍
2. 数据库基础知识介绍
3. mysql的安装
4. mysql常用的一些命令介绍
5. SQL分类

背景介绍

我们每天都在访问各种网站、APP，如微信、QQ、抖音、今日头条、腾讯新闻等，这些东西上面都存在大量的信息，这些信息都需要有地方存储，存储在哪呢？数据库。

所以如果我们需要开发一个网站、app，数据库我们必须掌握的技术，常用的数据库有mysql、oracle、sqlserver、db2等。

上面介绍的几个数据库，oracle性能排名第一，服务也是相当到位的，但是收费也是非常高的，金融公司对数据库稳定性要求比较高，一般会选择oracle。

mysql是免费的，其他几个目前暂时收费的，mysql在互联网公司使用率也是排名第一，资料也非常完善，社区也非常活跃，所以我们主要学习mysql。

mysql系列我们主要介绍

1. mysql的基本使用

2. mysql性能优化

3. 开发过程中mysql一些优秀的案例介绍

数据库常见的概念

DB：数据库，存储数据的容器。

DBMS：数据库管理系统，又称为数据库软件或数据库产品，用于创建或管理DB。

SQL：结构化查询语言，用于和数据库通信的语言，不是某个数据库软件持有的，而是几乎所有的主流数据库软件通用的语言。中国人之间交流需要说汉语，和美国人之间交流需要说英语，和数据库沟通需要说SQL语言。

数据库存储数据的一些特点

- 数据存放在表中，然后表存放在数据库中
- 一个库中可以有多张表，每张表具有唯一的名称（表名）来标识自己
- 表中有一个或多个列，列又称为“字段”，相当于java中的“属性”
- 表中每一行数据，相当于java中的“对象”

window中安装mysql

官网下载mysql5.7.25：`https://dev.mysql.com/downloads/mysql/5.7.html#downloads`

win10安装mysql5.7详细步骤可以看：`http://www.itsoku.com/article/192`

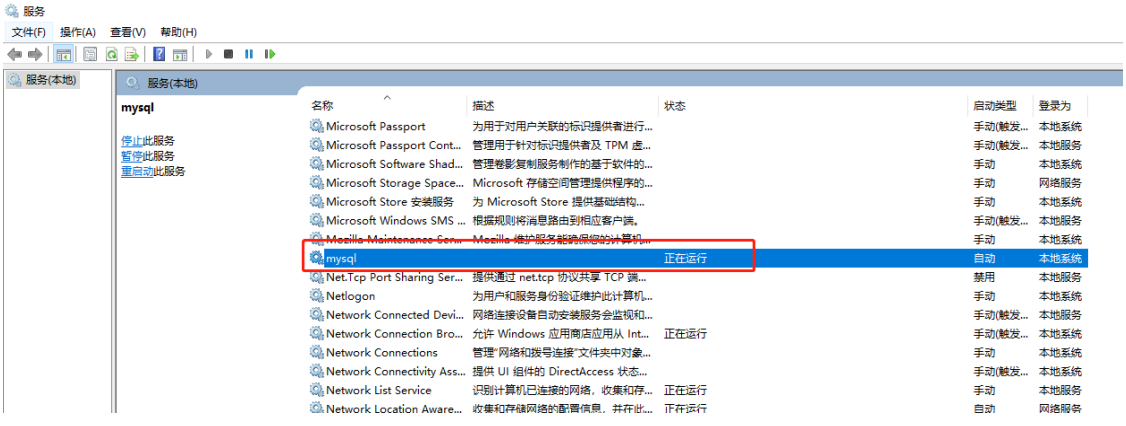
mysql常用的一些命令

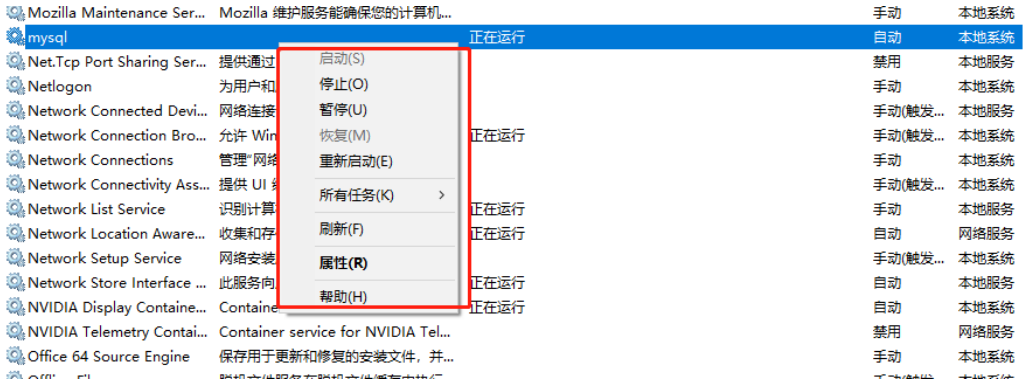
mysql启动2种方式

方式1：

cmd中运行services.msc

会打开服务窗口，在服务窗口中找到mysql服务，点击右键可以启动或者停止





方式2

以管理员身份运行cmd命令

停止命令：**net stop mysql**

启动命令：**net start mysql**

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止.
mysql 服务已成功停止。


C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

注意：命令后面没有结束符号

mysql登录命令

mysql -h ip -P 端口 -u 用户名 -p

```
C:\Windows\system32>mysql -h localhost -P 3306 -u root -p
Enter password: *******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 5.7.25-log MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

说明：

- -P 大写的P后面跟上端口
- 如果是登录本金ip和端口可以省略，如：

mysql -u 用户名 -p

- 可以通过上面的命令连接原创机器的mysql

查看数据库版本

mysql --version 或者mysql -V用于在未登录情况下，查看本机mysql版本：

```
C:\Windows\system32>mysql -V
mysql  Ver 14.14 Distrib 5.7.25, for Win64 (x86_64)

C:\Windows\system32>mysql --version
mysql  Ver 14.14 Distrib 5.7.25, for Win64 (x86_64)
```

select version();：登录情况下，查看链接的库版本：

```
mysql> select version();
+------------+
| version()  |
+------------+
| 5.7.25-log |
+------------+
1 row in set (0.00 sec)
```

显示所有数据库：show databases;

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| apolloconfigdb     |
| apolloportaldb     |
| config-server      |
| dblog              |
| diamond_devtest    |
| mysql              |
| nacos_config       |
| performance_schema |
| rs_elastic_job     |
| rs_master          |
```

```
| seata              |
| sys                |
+--------------------+
13 rows in set (0.00 sec)
```

进入指定的库：use 库名;

```
mysql> use seata;
Database changed
```

显示当前库中所有的表：show tables;

```
mysql> show tables;
+--------------------+
| Tables_in_dblog    |
+--------------------+
| biz_article        |
| biz_article_look   |
| biz_article_love   |
| biz_article_tags   |
| biz_comment        |
| biz_file           |
| biz_tags           |
| biz_type           |
| sys_config         |
| sys_link           |
| sys_log            |
| sys_notice         |
| sys_resources      |
| sys_role           |
| sys_role_resources |
| sys_template       |
| sys_update_recorde |
| sys_user           |
| sys_user_role      |
+--------------------+
19 rows in set (0.00 sec)
```

查看其他库中所有的表：show tables from 库名;

```
mysql> show tables from seata;
+-----------------+
| Tables_in_seata |
```

```
+-----------------+
| branch_table    |
| global_table    |
| lock_table      |
| t_account       |
| t_order         |
| t_storage       |
| undo_log        |
+-----------------+
7 rows in set (0.00 sec)
```

查看表的创建语句：show create table 表名;

```
mysql> show create table biz_tags;
+----------
+-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
----------------------------+
| Table    | Create Table
|
+----------
+-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
----------------------------+
| biz_tags | CREATE TABLE `biz_tags` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL COMMENT '书签名',
  `description` varchar(100) DEFAULT NULL COMMENT '描述',
  `create_time` datetime DEFAULT CURRENT_TIMESTAMP COMMENT '添加时间',
  `update_time` datetime DEFAULT CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=21 DEFAULT CHARSET=utf8
ROW_FORMAT=COMPACT              |
+----------
```

```
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
----------------------------+
1 row in set (0.00 sec)
```

查看表结构：desc 表名;

```
mysql> desc biz_tags;
+-------------+---------------------+------+-----+-------------------
+----------------+
| Field       | Type                | Null | Key | Default           |
Extra          |
+-------------+---------------------+------+-----+-------------------
+----------------+
| id          | bigint(20) unsigned | NO   | PRI | NULL              |
auto_increment |
| name        | varchar(50)         | NO   |     | NULL              |
|
| description | varchar(100)        | YES  |     | NULL              |
|
| create_time | datetime            | YES  |     | CURRENT_TIMESTAMP |
|
| update_time | datetime            | YES  |     | CURRENT_TIMESTAMP |
|
+-------------+---------------------+------+-----+-------------------
+----------------+
5 rows in set (0.00 sec)
```

查看当前所在库：select database();

```
C:\Windows\system32>mysql -h localhost -P 3306 -u root -p
Enter password: *******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.7.25-log MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights
reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> select database();
+------------+
| database() |
+------------+
| NULL       |
+------------+
1 row in set (0.00 sec)

mysql> use dblog;
Database changed
mysql> select database();
+------------+
| database() |
+------------+
| dblog      |
+------------+
1 row in set (0.00 sec)
```

查看当前mysql支持的存储引擎：SHOW ENGINES;

```
mysql> SHOW ENGINES;
+--------------------+---------
+-------------------------------------------------------------------
+--------------+------+------------+
| Engine             | Support | Comment
| Transactions | XA   | Savepoints |
+--------------------+---------
+-------------------------------------------------------------------
+--------------+------+------------+
| InnoDB             | DEFAULT | Supports transactions, row-level
locking, and foreign keys     | YES          | YES  | YES        |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables
| NO           | NO   | NO         |
| MEMORY             | YES     | Hash based, stored in memory, useful
for temporary tables      | NO           | NO   | NO         |
```

```
| BLACKHOLE          | YES     | /dev/null storage engine (anything
you write to it disappears) | NO           | NO   | NO         |
| MyISAM             | YES     | MyISAM storage engine
| NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine
| NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine
| NO           | NO   | NO         |
| PERFORMANCE_SCHEMA | YES     | Performance Schema
| NO           | NO   | NO         |
| FEDERATED          | NO      | Federated MySQL storage engine
| NULL         | NULL | NULL       |
+--------------------+---------
+-------------------------------------------------------------------
+--------------+------+------------+
9 rows in set (0.00 sec)
```

查看系统变量及其值：SHOW VARIABLES;

```
mysql> SHOW VARIABLES;
+-------------------------------------------------------------
+-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------+
| Variable_name                                                | Value
|
+-------------------------------------------------------------
+-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
-------------------------------------------------------+
| auto_increment_increment                                     | 1
|
| auto_increment_offset                                        | 1
|
| autocommit                                                   | ON
|
| automatic_sp_privileges                                      | ON
```

```
|
| avoid_temporal_upgrade                                  | OFF
|
| back_log                                                | 90
|
| basedir                                                 | D:
\installsoft\MySQL\mysql-5.7.25-winx64\
|
| big_tables                                              | OFF
|
| bind_address                                            | *
|
| binlog_cache_size                                       | 32768
|
| binlog_checksum                                         | CRC32
|
| binlog_direct_non_transactional_updates                 | OFF
|
| binlog_error_action                                     |
ABORT_SERVER
|
| binlog_format                                           | ROW
|
| binlog_group_commit_sync_delay                          | 0
|
| binlog_group_commit_sync_no_delay_count                 | 0
|
| binlog_gtid_simple_recovery                             | ON
|
| binlog_max_flush_queue_time                             | 0
|
| binlog_order_commits                                    | ON
|
| binlog_row_image                                        | FULL
|
| binlog_rows_query_log_events                            | OFF
|
| binlog_stmt_cache_size                                  | 32768
|
| binlog_transaction_dependency_history_size              | 25000
|
| binlog_transaction_dependency_tracking                  |
```

```
COMMIT_ORDER
|
| block_encryption_mode                                    | aes-128-
ecb
|
| bulk_insert_buffer_size                                  | 8388608
|
| character_set_client                                     | utf8
|
| character_set_connection                                 | utf8
|
| character_set_database                                   | utf8mb4
|
| character_set_filesystem                                 | binary
|
| character_set_results                                    | utf8
|
| character_set_server                                     | utf8
|
| character_set_system                                     | utf8
|
| character_sets_dir                                       | D:
\installsoft\MySQL\mysql-5.7.25-winx64\share\charsets\
|
| check_proxy_users                                        | OFF
|
| collation_connection                                     |
utf8_general_ci
|
| collation_database                                       |
utf8mb4_bin
|
| collation_server                                         |
utf8_general_ci
|
| completion_type                                          | NO_CHAIN
|
| concurrent_insert                                        | AUTO
|
| connect_timeout                                          | 10
|
| core_file                                                | OFF
```

```
|
| datadir                                                  | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\
|
| date_format                                              | %Y-%m-%d
|
| datetime_format                                          | %Y-%m-%d
%H:%i:%s
|
| default_authentication_plugin                            |
mysql_native_password
|
| default_password_lifetime                                | 0
|
| default_storage_engine                                   | InnoDB
|
| default_tmp_storage_engine                               | InnoDB
|
| default_week_format                                      | 0
|
| delay_key_write                                          | ON
|
| delayed_insert_limit                                     | 100
|
| delayed_insert_timeout                                   | 300
|
| delayed_queue_size                                       | 1000
|
| disabled_storage_engines                                 |
|
| disconnect_on_expired_password                           | ON
|
| div_precision_increment                                  | 4
|
| end_markers_in_json                                      | OFF
|
| enforce_gtid_consistency                                 | OFF
|
| eq_range_index_dive_limit                                | 200
|
| error_count                                              | 0
|
```

```
| event_scheduler                                          | OFF
|
| expire_logs_days                                         | 0
|
| explicit_defaults_for_timestamp                          | OFF
|
| external_user                                            |
|
| flush                                                    | OFF
|
| flush_time                                               | 0
|
| foreign_key_checks                                       | ON
|
| ft_boolean_syntax                                        | +
-><()~*:""&|
|
| ft_max_word_len                                          | 84
|
| ft_min_word_len                                          | 4
|
| ft_query_expansion_limit                                 | 20
|
| ft_stopword_file                                         | (built-
in)
|
| general_log                                              | OFF
|
| general_log_file                                         | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\DESKTOP-3OB6NA3.log
|
| group_concat_max_len                                     | 1024
|
| gtid_executed_compression_period                         | 1000
|
| gtid_mode                                                | OFF
|
| gtid_next                                                | AUTOMATIC
|
| gtid_owned                                               |
|
| gtid_purged                                              |
```

have_compress	YES
have_crypt	NO
have_dynamic_loading	YES
have_geometry	YES
have_openssl	DISABLED
have_profiling	YES
have_query_cache	YES
have_rtree_keys	YES
have_ssl	DISABLED
have_statement_timeout	YES
have_symlink	YES
host_cache_size	328
hostname	DESKTOP-3OB6NA3
identity	0
ignore_builtin_innodb	OFF
ignore_db_dirs	
init_connect	
init_file	
init_slave	
innodb_adaptive_flushing	ON

```
| innodb_adaptive_flushing_lwm                    | 10
|
| innodb_adaptive_hash_index                      | ON
|
| innodb_adaptive_hash_index_parts                | 8
|
| innodb_adaptive_max_sleep_delay                 | 150000
|
| innodb_api_bk_commit_interval                   | 5
|
| innodb_api_disable_rowlock                      | OFF
|
| innodb_api_enable_binlog                        | OFF
|
| innodb_api_enable_mdl                           | OFF
|
| innodb_api_trx_level                            | 0
|
| innodb_autoextend_increment                     | 64
|
| innodb_autoinc_lock_mode                        | 1
|
| innodb_buffer_pool_chunk_size                   | 134217728
|
| innodb_buffer_pool_dump_at_shutdown             | ON
|
| innodb_buffer_pool_dump_now                     | OFF
|
| innodb_buffer_pool_dump_pct                     | 25
|
| innodb_buffer_pool_filename                     |
ib_buffer_pool
|
| innodb_buffer_pool_instances                    | 1
|
| innodb_buffer_pool_load_abort                   | OFF
|
| innodb_buffer_pool_load_at_startup              | ON
|
| innodb_buffer_pool_load_now                     | OFF
|
| innodb_buffer_pool_size                         | 134217728
```

innodb_change_buffer_max_size	25
innodb_change_buffering	all
innodb_checksum_algorithm	crc32
innodb_checksums	ON
innodb_cmp_per_index_enabled	OFF
innodb_commit_concurrency	0
innodb_compression_failure_threshold_pct	5
innodb_compression_level	6
innodb_compression_pad_pct_max	50
innodb_concurrency_tickets	5000
innodb_data_file_path	ibdata1:12M:autoextend
innodb_data_home_dir	
innodb_deadlock_detect	ON
innodb_default_row_format	dynamic
innodb_disable_sort_file_cache	OFF
innodb_doublewrite	ON
innodb_fast_shutdown	1
innodb_file_format	Barracuda
innodb_file_format_check	ON
innodb_file_format_max	Barracuda

innodb_file_per_table	ON
innodb_fill_factor	100
innodb_flush_log_at_timeout	1
innodb_flush_log_at_trx_commit	1
innodb_flush_method	
innodb_flush_neighbors	1
innodb_flush_sync	ON
innodb_flushing_avg_loops	30
innodb_force_load_corrupted	OFF
innodb_force_recovery	0
innodb_ft_aux_table	
innodb_ft_cache_size	8000000
innodb_ft_enable_diag_print	OFF
innodb_ft_enable_stopword	ON
innodb_ft_max_token_size	84
innodb_ft_min_token_size	3
innodb_ft_num_word_optimize	2000
innodb_ft_result_cache_limit	2000000000
innodb_ft_server_stopword_table	
innodb_ft_sort_pll_degree	2
innodb_ft_total_cache_size	640000000

innodb_ft_user_stopword_table	
innodb_io_capacity	200
innodb_io_capacity_max	2000
innodb_large_prefix	ON
innodb_lock_wait_timeout	50
innodb_locks_unsafe_for_binlog	OFF
innodb_log_buffer_size	16777216
innodb_log_checksums	ON
innodb_log_compressed_pages	ON
innodb_log_file_size	50331648
innodb_log_files_in_group	2
innodb_log_group_home_dir	.\
innodb_log_write_ahead_size	8192
innodb_lru_scan_depth	1024
innodb_max_dirty_pages_pct	75.000000
innodb_max_dirty_pages_pct_lwm	0.000000
innodb_max_purge_lag	0
innodb_max_purge_lag_delay	0
innodb_max_undo_log_size	1073741824
innodb_monitor_disable	

innodb_monitor_enable	
innodb_monitor_reset	
innodb_monitor_reset_all	
innodb_old_blocks_pct	37
innodb_old_blocks_time	1000
innodb_online_alter_log_max_size	134217728
innodb_open_files	2000
innodb_optimize_fulltext_only	OFF
innodb_page_cleaners	1
innodb_page_size	16384
innodb_print_all_deadlocks	OFF
innodb_purge_batch_size	300
innodb_purge_rseg_truncate_frequency	128
innodb_purge_threads	4
innodb_random_read_ahead	OFF
innodb_read_ahead_threshold	56
innodb_read_io_threads	4
innodb_read_only	OFF
innodb_replication_delay	0
innodb_rollback_on_timeout	OFF
innodb_rollback_segments	128

```
| innodb_sort_buffer_size                                | 1048576
|
| innodb_spin_wait_delay                                 | 6
|
| innodb_stats_auto_recalc                               | ON
|
| innodb_stats_include_delete_marked                     | OFF
|
| innodb_stats_method                                    |
nulls_equal
|
| innodb_stats_on_metadata                               | OFF
|
| innodb_stats_persistent                                | ON
|
| innodb_stats_persistent_sample_pages                   | 20
|
| innodb_stats_sample_pages                              | 8
|
| innodb_stats_transient_sample_pages                    | 8
|
| innodb_status_output                                   | ON
|
| innodb_status_output_locks                             | ON
|
| innodb_strict_mode                                     | ON
|
| innodb_support_xa                                      | ON
|
| innodb_sync_array_size                                 | 1
|
| innodb_sync_spin_loops                                 | 30
|
| innodb_table_locks                                     | ON
|
| innodb_temp_data_file_path                             |
ibtmp1:12M:autoextend
|
| innodb_thread_concurrency                              | 0
|
| innodb_thread_sleep_delay                              | 10000
|
```

innodb_tmpdir	
innodb_undo_directory	.\
innodb_undo_log_truncate	OFF
innodb_undo_logs	128
innodb_undo_tablespaces	0
innodb_use_native_aio	ON
innodb_version	5.7.25
innodb_write_io_threads	4
insert_id	0
interactive_timeout	28800
internal_tmp_disk_storage_engine	InnoDB
join_buffer_size	262144
keep_files_on_create	OFF
key_buffer_size	8388608
key_cache_age_threshold	300
key_cache_block_size	1024
key_cache_division_limit	100
keyring_operations	ON
large_files_support	ON
large_page_size	0
large_pages	OFF

```
| last_insert_id                                          | 0
|
| lc_messages                                             | en_US
|
| lc_messages_dir                                         | D:
\installsoft\MySQL\mysql-5.7.25-winx64\share\
|
| lc_time_names                                           | en_US
|
| license                                                 | GPL
|
| local_infile                                            | ON
|
| lock_wait_timeout                                       | 31536000
|
| log_bin                                                 | ON
|
| log_bin_basename                                        | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\mysql_bin
|
| log_bin_index                                           | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\mysql_bin.index
|
| log_bin_trust_function_creators                         | OFF
|
| log_bin_use_v1_row_events                               | OFF
|
| log_builtin_as_identified_by_password                   | OFF
|
| log_error                                               | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\DESKTOP-3OB6NA3.err
|
| log_error_verbosity                                     | 3
|
| log_output                                              | FILE
|
| log_queries_not_using_indexes                           | OFF
|
| log_slave_updates                                       | OFF
|
| log_slow_admin_statements                               | OFF
|
```

log_slow_slave_statements	OFF
log_statements_unsafe_for_binlog	ON
log_syslog	ON
log_syslog_tag	
log_throttle_queries_not_using_indexes	0
log_timestamps	UTC
log_warnings	2
long_query_time	0.000000
low_priority_updates	OFF
lower_case_file_system	ON
lower_case_table_names	1
master_info_repository	FILE
master_verify_checksum	OFF
max_allowed_packet	4194304
max_binlog_cache_size	18446744073709547520
max_binlog_size	1073741824
max_binlog_stmt_cache_size	18446744073709547520
max_connect_errors	100
max_connections	200
max_delayed_threads	20

```
|
| max_digest_length                                    | 1024
|
| max_error_count                                      | 64
|
| max_execution_time                                   | 0
|
| max_heap_table_size                                  | 16777216
|
| max_insert_delayed_threads                           | 20
|
| max_join_size                                        |
18446744073709551615
|
| max_length_for_sort_data                             | 1024
|
| max_points_in_geometry                               | 65536
|
| max_prepared_stmt_count                              | 16382
|
| max_relay_log_size                                   | 0
|
| max_seeks_for_key                                    |
4294967295
|
| max_sort_length                                      | 1024
|
| max_sp_recursion_depth                               | 0
|
| max_tmp_tables                                       | 32
|
| max_user_connections                                 | 0
|
| max_write_lock_count                                 |
4294967295
|
| metadata_locks_cache_size                            | 1024
|
| metadata_locks_hash_instances                        | 8
|
| min_examined_row_limit                               | 0
|
```

multi_range_count	256
myisam_data_pointer_size	6
myisam_max_sort_file_size	2146435072
myisam_mmap_size	18446744073709551615
myisam_recover_options	OFF
myisam_repair_threads	1
myisam_sort_buffer_size	8388608
myisam_stats_method	nulls_unequal
myisam_use_mmap	OFF
mysql_native_password_proxy_users	OFF
named_pipe	OFF
named_pipe_full_access_group	*everyone*
net_buffer_length	16384
net_read_timeout	30
net_retry_count	10
net_write_timeout	60
new	OFF
ngram_token_size	2
offline_mode	OFF

```
| old                                                    | OFF
|
| old_alter_table                                        | OFF
|
| old_passwords                                          | 0
|
| open_files_limit                                       | 7048
|
| optimizer_prune_level                                  | 1
|
| optimizer_search_depth                                 | 62
|
| optimizer_switch                                       |
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_me
rge_intersection=on,engine_condition_pushdown=on,index_condition_pushd
own=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_acces
s=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplic
ateweedout=on,subquery_materialization_cost_based=on,use_index_extensi
ons=on,condition_fanout_filter=on,derived_merge=on |
| optimizer_trace                                        |
enabled=off,one_line=off
|
| optimizer_trace_features                               |
greedy_search=on,range_optimizer=on,dynamic_range=on,repeated_subselec
t=on
|
| optimizer_trace_limit                                  | 1
|
| optimizer_trace_max_mem_size                           | 16384
|
| optimizer_trace_offset                                 | -1
|
| parser_max_mem_size                                    |
18446744073709551615
|
| performance_schema                                     | ON
|
| performance_schema_accounts_size                       | -1
|
| performance_schema_digests_size                        | 10000
|
| performance_schema_events_stages_history_long_size     | 10000
```

performance_schema_events_stages_history_size	10
performance_schema_events_statements_history_long_size	10000
performance_schema_events_statements_history_size	10
performance_schema_events_transactions_history_long_size	10000
performance_schema_events_transactions_history_size	10
performance_schema_events_waits_history_long_size	10000
performance_schema_events_waits_history_size	10
performance_schema_hosts_size	-1
performance_schema_max_cond_classes	80
performance_schema_max_cond_instances	-1
performance_schema_max_digest_length	1024
performance_schema_max_file_classes	80
performance_schema_max_file_handles	32768
performance_schema_max_file_instances	-1
performance_schema_max_index_stat	-1
performance_schema_max_memory_classes	320
performance_schema_max_metadata_locks	-1
performance_schema_max_mutex_classes	210
performance_schema_max_mutex_instances	-1
performance_schema_max_prepared_statements_instances	-1
performance_schema_max_program_instances	-1

```
|
| performance_schema_max_rwlock_classes                      | 50
|
| performance_schema_max_rwlock_instances                    | -1
|
| performance_schema_max_socket_classes                      | 10
|
| performance_schema_max_socket_instances                    | -1
|
| performance_schema_max_sql_text_length                     | 1024
|
| performance_schema_max_stage_classes                       | 150
|
| performance_schema_max_statement_classes                   | 193
|
| performance_schema_max_statement_stack                     | 10
|
| performance_schema_max_table_handles                       | -1
|
| performance_schema_max_table_instances                     | -1
|
| performance_schema_max_table_lock_stat                     | -1
|
| performance_schema_max_thread_classes                      | 50
|
| performance_schema_max_thread_instances                    | -1
|
| performance_schema_session_connect_attrs_size              | 512
|
| performance_schema_setup_actors_size                       | -1
|
| performance_schema_setup_objects_size                      | -1
|
| performance_schema_users_size                              | -1
|
| pid_file                                                   | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\DESKTOP-3OB6NA3.pid
|
| plugin_dir                                                 | D:
\installsoft\MySQL\mysql-5.7.25-winx64\lib\plugin\
|
| port                                                       | 3306
```

preload_buffer_size	32768
profiling	OFF
profiling_history_size	15
protocol_version	10
proxy_user	
pseudo_slave_mode	OFF
pseudo_thread_id	2
query_alloc_block_size	8192
query_cache_limit	1048576
query_cache_min_res_unit	4096
query_cache_size	1048576
query_cache_type	OFF
query_cache_wlock_invalidate	OFF
query_prealloc_size	8192
rand_seed1	0
rand_seed2	0
range_alloc_block_size	4096
range_optimizer_max_mem_size	8388608
rbr_exec_mode	STRICT
read_buffer_size	131072
read_only	OFF

```
|
| read_rnd_buffer_size                                          | 262144
|
| relay_log                                                     |
|
| relay_log_basename                                            | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\DESKTOP-3OB6NA3-relay-bin
|
| relay_log_index                                               | D:
\installsoft\MySQL\mysql-5.7.25-winx64\data\DESKTOP-3OB6NA3-relay-
bin.index
|
| relay_log_info_file                                           | relay-
log.info
|
| relay_log_info_repository                                     | FILE
|
| relay_log_purge                                               | ON
|
| relay_log_recovery                                            | OFF
|
| relay_log_space_limit                                         | 0
|
| report_host                                                   |
|
| report_password                                               |
|
| report_port                                                   | 3306
|
| report_user                                                   |
|
| require_secure_transport                                      | OFF
|
| rpl_stop_slave_timeout                                        | 31536000
|
| secure_auth                                                   | ON
|
| secure_file_priv                                              | NULL
|
| server_id                                                     | 1
|
| server_id_bits                                                | 32
```

```
|
| server_uuid                                                 |
5535390b-40a4-11e9-af3e-e86a64887726
|
| session_track_gtids                                         | OFF
|
| session_track_schema                                        | ON
|
| session_track_state_change                                  | OFF
|
| session_track_system_variables                              |
time_zone,autocommit,character_set_client,character_set_results,charac
ter_set_connection
|
| session_track_transaction_info                              | OFF
|
| sha256_password_proxy_users                                 | OFF
|
| shared_memory                                               | OFF
|
| shared_memory_base_name                                     | MYSQL
|
| show_compatibility_56                                       | OFF
|
| show_create_table_verbosity                                 | OFF
|
| show_old_temporals                                          | OFF
|
| skip_external_locking                                       | ON
|
| skip_name_resolve                                           | OFF
|
| skip_networking                                             | OFF
|
| skip_show_database                                          | OFF
|
| slave_allow_batching                                        | OFF
|
| slave_checkpoint_group                                      | 512
|
| slave_checkpoint_period                                     | 300
|
```

slave_compressed_protocol	OFF
slave_exec_mode	STRICT
slave_load_tmpdir	C:\Windows\TEMP
slave_max_allowed_packet	1073741824
slave_net_timeout	60
slave_parallel_type	DATABASE
slave_parallel_workers	0
slave_pending_jobs_size_max	16777216
slave_preserve_commit_order	OFF
slave_rows_search_algorithms	TABLE_SCAN,INDEX_SCAN
slave_skip_errors	OFF
slave_sql_verify_checksum	ON
slave_transaction_retries	10
slave_type_conversions	
slow_launch_time	2
slow_query_log	ON
slow_query_log_file	D:/installsoft/MySQL/mysql-5.7.25-winx64/log/slow.log
socket	MySQL
sort_buffer_size	262144

```
| sql_auto_is_null                                              | OFF
|
| sql_big_selects                                               | ON
|
| sql_buffer_result                                             | OFF
|
| sql_log_bin                                                   | ON
|
| sql_log_off                                                   | OFF
|
| sql_mode                                                      |
ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ER
ROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
|
| sql_notes                                                     | ON
|
| sql_quote_show_create                                         | ON
|
| sql_safe_updates                                              | OFF
|
| sql_select_limit                                              |
18446744073709551615
|
| sql_slave_skip_counter                                        | 0
|
| sql_warnings                                                  | OFF
|
| ssl_ca                                                        |
|
| ssl_capath                                                    |
|
| ssl_cert                                                      |
|
| ssl_cipher                                                    |
|
| ssl_crl                                                       |
|
| ssl_crlpath                                                   |
|
| ssl_key                                                       |
|
| stored_program_cache                                          | 256
```

```
|
| super_read_only                                          | OFF
|
| sync_binlog                                              | 1
|
| sync_frm                                                 | ON
|
| sync_master_info                                         | 10000
|
| sync_relay_log                                           | 10000
|
| sync_relay_log_info                                      | 10000
|
| system_time_zone                                         |
|
| table_definition_cache                                   | 1400
|
| table_open_cache                                         | 2000
|
| table_open_cache_instances                               | 16
|
| thread_cache_size                                        | 10
|
| thread_handling                                          | one-
thread-per-connection
|
| thread_stack                                             | 262144
|
| time_format                                              | %H:%i:%s
|
| time_zone                                                | SYSTEM
|
| timestamp                                                |
1566971784.132916
|
| tls_version                                              |
TLSv1,TLSv1.1
|
| tmp_table_size                                           | 16777216
|
| tmpdir                                                   | C:
\Windows\TEMP
```

```
|
| transaction_alloc_block_size                                | 8192
|
| transaction_allow_batching                                  | OFF
|
| transaction_isolation                                       | READ-
COMMITTED
|
| transaction_prealloc_size                                   | 4096
|
| transaction_read_only                                       | OFF
|
| transaction_write_set_extraction                            | OFF
|
| tx_isolation                                                | READ-
COMMITTED
|
| tx_read_only                                                | OFF
|
| unique_checks                                               | ON
|
| updatable_views_with_limit                                  | YES
|
| version                                                     | 5.7.25-
log
|
| version_comment                                             | MySQL
Community Server (GPL)
|
| version_compile_machine                                     | x86_64
|
| version_compile_os                                          | Win64
|
| wait_timeout                                                | 28800
|
| warning_count                                               | 0
|
+-------------------------------------------------------------
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
```

```
-------------------------------------------------------------------------
-----------------------------------------------------+
513 rows in set, 1 warning (0.00 sec)
```

查看某个系统变量：SHOW VARIABLES like '变量名';

```
mysql> SHOW VARIABLES like 'wait_timeout';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| wait_timeout  | 28800 |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW VARIABLES like '%wait_timeou%t';
+--------------------------+----------+
| Variable_name            | Value    |
+--------------------------+----------+
| innodb_lock_wait_timeout | 50       |
| lock_wait_timeout        | 31536000 |
| wait_timeout             | 28800    |
+--------------------------+----------+
3 rows in set, 1 warning (0.00 sec)
```

mysql语法规范

1. 不区分大小写，但建议关键字大写，表名、列名小写
2. 每条命令最好用英文分号结尾
3. 每条命令根据需要，可以进行缩进或换行
4. 注释
    - 单行注释：#注释文字
    - 单行注释：-- 注释文字 ，注意， 这里需要加空格
    - 多行注释：/* 注释文字 */

SQL的语言分类

- **DQL（Data Query Language）**：数据查询语言 select 相关语句
- **DML（Data Manipulate Language）**：数据操作语言 insert 、update、delete 语句
- **DDL（Data Define Languge）**：数据定义语言 create、drop、alter 语句
- **TCL（Transaction Control Language）**：事务控制语言 set autocommit=0、start transaction、savepoint、commit、rollback

第2篇：MySQL中数据类型介绍

这是mysql系列第2篇文章。

环境：mysql5.7.25，cmd命令中进行演示。

主要内容

1. 介绍mysql中常用的数据类型
2. mysql类型和java类型对应关系
3. 数据类型选择的一些建议

MySQL的数据类型

主要包括以下五大类

- 整数类型：bit、bool、tinyint、smallint、mediumint、int、bigint
- 浮点数类型：float、double、decimal
- 字符串类型：char、varchar、tinyblob、blob、mediumblob、longblob、tinytext、text、mediumtext、longtext
- 日期类型：Date、DateTime、TimeStamp、Time、Year
- 其他数据类型：暂不介绍，用的比较少。

整数类型

类型	字节数	无符号	有符号
tiny int[(n)][unsigned]	1	[0, 2^7-1]	[-2^7, 2^8-1]

small int[(n)][unsigned]

2[[-0 2^{15}, , 2^{16} 2^{15}- - 1 1]]

medium int[(n)][unsigned] 3[[-0, , 2^24 2^23--1 1]]

int[(n)][unsigned] 4[[-0, , 2^32 2^31--1 1]]

b8[[
i - 0
g 2^{63},
i , 2^{64}
n 2^{63}-
t - 1
[1]
( ]
n
)
]
[
u
n
s
i
g
n
e
d
]

上面[]包含的内容是可选的，默认是有符号类型的，无符号的需要在类型后面跟上unsigned

示例1：有符号类型

```
mysql> create table demo1(
      c1 tinyint
     );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into demo1 values(-pow(2,7)),(pow(2,7)-1);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from demo1;
+------+
| c1   |
+------+
| -128 |
|  127 |
```

```
+------+
2 rows in set (0.00 sec)

mysql> insert into demo1 values(pow(2,7));
ERROR 1264 (22003): Out of range value for column 'c1' at row 1
```

demo1表中c1字段为tinyint有符号类型的，可以看一下上面的演示，有超出范围报错的。

关于数值对应的范围计算方式属于计算机基础的一些知识，可以去看一下计算机的二进制表示相关的文章。

示例2：无符号类型

```
mysql> create table demo2(
      c1 tinyint unsigned
     );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into demo2 values (-1);
ERROR 1264 (22003): Out of range value for column 'c1' at row 1
mysql> insert into demo2 values (pow(2,8)+1);
ERROR 1264 (22003): Out of range value for column 'c1' at row 1
mysql> insert into demo2 values (0),(pow(2,8));

mysql> insert into demo2 values (0),(pow(2,8)-1);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from demo2;
+------+
| c1   |
+------+
|    0 |
|  255 |
+------+
2 rows in set (0.00 sec)
```

c1是无符号的tinyint类型的，插入了负数会报错。

类型(n)说明

在开发中，我们会碰到有些定义整型的写法是int(11)，这种写法个人感觉在开发过程中没有什么用途，不过还是来说一下，int(N)我们只需要记住两点：

- 无论N等于多少，int永远占4个字节
- **N表示的是显示宽度，不足的用0补足，超过的无视长度而直接显示整个数字，但这要整型设置了unsigned zerofill才有效**

看一下示例，理解更方便：

```
mysql> CREATE TABLE test3 (
       `a` int,
       `b` int(5),
       `c` int(5) unsigned,
       `d` int(5) zerofill,
       `e` int(5) unsigned zerofill,
       `f` int    zerofill,
       `g` int    unsigned zerofill
     );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test3 values (1,1,1,1,1,1,1),
(11,11,11,11,11,11,11),(12345,12345,12345,12345,12345,12345,12345);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from test3;
+-------+-------+-------+-------+-------+------------+------------+
| a     | b     | c     | d     | e     | f          | g          |
+-------+-------+-------+-------+-------+------------+------------+
|     1 |     1 |     1 | 00001 | 00001 | 0000000001 | 0000000001 |
|    11 |    11 |    11 | 00011 | 00011 | 0000000011 | 0000000011 |
| 12345 | 12345 | 12345 | 12345 | 12345 | 0000012345 | 0000012345 |
+-------+-------+-------+-------+-------+------------+------------+
3 rows in set (0.00 sec)

mysql> show create table test3;
| Table | Create Table
```

```
| test3 | CREATE TABLE `test3` (
  `a` int(11) DEFAULT NULL,
  `b` int(5) DEFAULT NULL,
  `c` int(5) unsigned DEFAULT NULL,
  `d` int(5) unsigned zerofill DEFAULT NULL,
  `e` int(5) unsigned zerofill DEFAULT NULL,
  `f` int(10) unsigned zerofill DEFAULT NULL,
  `g` int(10) unsigned zerofill DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

show create table test3;输出了表test3的创建语句，和我们原始的创建语句不一致了，原始的d字段用的是无符号的，可以看出当使用了zerofill自动会将无符号提升为有符号。

说明：

int(5)输出宽度不满5时，前面用0来进行填充

int(n)中的n省略的时候，**宽度为对应类型无符号最大值的十进制的长度**，如bigint无符号最大值为2^{64}-1 = 18,446,744,073,709,551,615；长度是20位，来个bigint左边0填充的示例看一下

```
mysql> CREATE TABLE test4 (
       `a`  bigint    zerofill
     );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test4 values(1);
Query OK, 1 row affected (0.00 sec)

mysql> select *from test4;
+----------------------+
| a                    |
+----------------------+
| 00000000000000000001 |
+----------------------+
1 row in set (0.00 sec)
```

上面的结果中1前面补了19个0，和期望的结果一致。

浮点类型（容易懵，注意看）

茫茫

匪匪

宇宙无

节答答

类大号号用

型小））边

float[(m,d)]	4	(-3.402823466E+38,3.402823466351E+38)	[0,3.402823466E+38)	单精度 浮点数值

双精度浮点数值

```
d8([-01,.17.97796796391331434846826321351751E7+E3+0380,81).7976931348623
```

decimal[(m,d)]，如果M>D，为M+2否则为D+2

float数值类型用于表示单精度浮点数值，而double数值类型用于表示双精度浮点数值，float和double都是浮点型，而decimal是定点型。

浮点型和定点型可以用类型名称后加（M，D）来表示，M表示该值的总共长度，D表示小数点后面的长度，M和D又称为精度和标度。

float和double在不指定精度时，默认会按照实际的精度来显示，而DECIMAL在不指定精度时，默认整数为10，小数为0。

示例1(重点)

```
mysql> create table test5(a float(5,2),b double(5,2),c decimal(5,2));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test5 values (1,1,1),(2.1,2.1,2.1),
(3.123,3.123,3.123),(4.125,4.125,4.125),(5.115,5.115,5.115),
(6.126,6.126,6.126),(7.116,7.116,7.116),(8.1151,8.1151,8.1151),
(9.1251,9.1251,9.1251),(10.11501,10.11501,10.11501),
(11.12501,11.12501,11.12501);
Query OK, 7 rows affected, 5 warnings (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 5

mysql> select * from test5;
+-------+-------+-------+
| a     | b     | c     |
+-------+-------+-------+
|  1.00 |  1.00 |  1.00 |
|  2.10 |  2.10 |  2.10 |
|  3.12 |  3.12 |  3.12 |
|  4.12 |  4.12 |  4.13 |
|  5.12 |  5.12 |  5.12 |
|  6.13 |  6.13 |  6.13 |
|  7.12 |  7.12 |  7.12 |
|  8.12 |  8.12 |  8.12 |
|  9.13 |  9.13 |  9.13 |
| 10.12 | 10.12 | 10.12 |
| 11.13 | 11.13 | 11.13 |
+-------+-------+-------+
11 rows in set (0.00 sec)
```

结果说明（注意看）：

c是decimal类型，认真看一下输入和输出，发现**decimal采用的是四舍五入**

认真看一下a和b的输入和输出，尽然不是四舍五入，一脸闷逼，float和double采用的是**四舍六入五成双**

decimal插入的数据超过精度之后会触发警告。

什么是四舍六入五成双?

> 就是5以下舍弃5以上进位，如果需要处理数字为5的时候，需要看5后面是否还有不为0的任何数字，如果有，则直接进位，如果没有，需要看5前面的数字，若是奇数则进位，若是偶数则将5舍掉

示例2

我们将浮点类型的（M,D）精度和标度都去掉，看看效果：

```
mysql> create table test6(a float,b double,c decimal);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into test6 values (1,1,1),(1.234,1.234,1.4),
(1.234,0.01,1.5);
Query OK, 3 rows affected, 2 warnings (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 2

mysql> select * from test6;
+-------+-------+------+
| a     | b     | c    |
+-------+-------+------+
|     1 |     1 |    1 |
| 1.234 | 1.234 |    1 |
| 1.234 |  0.01 |    2 |
+-------+-------+------+
3 rows in set (0.00 sec)
```

说明：

> a和b的数据正确插入，而c被截断了
>
> 浮点数float、double如果不写精度和标度，则会按照实际显示
>
> decimal不写精度和标度，小数点后面的会进行四舍五入，并且插入时会有警告!

再看一下下面代码：

```
mysql> select sum(a),sum(b),sum(c) from test5;
+--------+--------+--------+
```

```
| sum(a) | sum(b) | sum(c) |
+--------+--------+--------+
|  67.21 |  67.21 |  67.22 |
+--------+--------+--------+
1 row in set (0.00 sec)

mysql> select sum(a),sum(b),sum(c) from test6;
+--------------------+--------------------+--------+
| sum(a)             | sum(b)             | sum(c) |
+--------------------+--------------------+--------+
| 3.4679999351501465 | 2.2439999999999998 |      4 |
+--------------------+--------------------+--------+
1 row in set (0.00 sec)
```

从上面sum的结果可以看出float、double会存在精度问题，decimal精度正常的，比如银行对统计结果要求比较精准的建议使用decimal。

日期类型

字节类型

大小

范围

格式

用途

DATE

1Y

0Y

0Y

--

0M

1M

--

0D

1D

/

9

9

9

9

-

1

2

-

3

1

日期值

TIME 3'-H8H:3M8M:5S9S:59'/'8838:59:59'

时间格式

YEAR 1Y9Y0Y1Y/2155

年份格式

混合日期和时间值

DATETIME

1Y
0Y
0Y
--
0M
1M
--
0D
1D
0H
0H
::
0M
0M
::
0S
0S
/
9
9
9
9
-
1
2
-
3
1
2
3
:
5
9
:
5
9

混合日期和时间值，时间戳

TIMESTAMP 1970-01-01 00:00:00/2038 结束时间是第 **2147483647**

字符串类型

类型	范围	存储所需字节	说明
char(M)	[0, m], m的范围[0, 2^8-1]	m	定长字符串

var[m0-65535字节var,m的范围[0,2^{16}-1]]

tiny blob(0L+255(2^8-1)字节)不超过255个字符的二进制字符串

blob 0L+ 二进制形式的长文本数据 (2^16 - 1) 字节

medium 0L+ 二进制形式的中等长度文本数据 (2^24 - 1) 字节

longblob	0L+4294967295(2^32-1)字节	二进制形式的极大文本数据
tinytext	0L+255(2^8-1)字节	短文本字符串

long_text	长文本数据
(0, 65535)	
(2^16 - 1) 字节	
medium_text	中等长度文本数据
(0, 16777215)	
(2^24 - 1) 字节	

longtext	L+4字节	极大文本数据
	0~4,294,967,295 (2^{32} - 1)字节	

char类型占用固定长度，如果存放的数据为固定长度的建议使用char类型，如：手机号码、身份证等固定长度的信息。

表格中的L表示存储的数据本身占用的字节，L 以外所需的额外字节为存放该值的长度所需的字节数。

MySQL 通过存储值的内容及其长度来处理可变长度的值，这些额外的字节是无符号整数。

请注意，可变长类型的最大长度、此类型所需的额外字节数以及占用相同字节数的无符号整数之间的对应关系：

> 例如，MEDIUMBLOB 值可能最多2^{24} - 1字节长并需要3 个字节记录其长度，3 个字节的整数类型MEDIUMINT 的最大无符号值为2^{24} - 1。

mysql类型和java类型对应关系

MySQLTypeName	Return value of GetColumnClassName	Returned as Java Class Names

BBIITT(1)(newinMySQL-5.0)

java.lang.Boolean

BBIITT(>1)(newinMySQL-5.0)

byte[]

```
TTjava.lang.Boolean if the configuration property t
IINNYYIINNTT
```

BOOL, BOOLEAN	TINYINT	See TINYINT, above as these are aliases for TINYI

```
SSMMAALLLIINNTT[[(MN)]][UNESIGNED]
java.lang.Integer(regardless if UNSIGNED or not
```

MMEEDDIIUUMMIINNTT[[(MN)SI]I[GUNNE,SDI]GNED]

java.lang.Integer, if UNSIGNED java.lang.Long

INT,INTEGER[(M)][UNSIGNED]	INTEGER[UNSIGNED]	java.lang.Integer, if UNSIGNED java.lang.Long

BBIIGGIINNTT[[(UMN)SI][GUNNE,SDif]GNED]

java.lang.Long

UNSIGNED

java.math.BigInteger

FLOAT[(M,D)] java.lang.Float

DOUBLE[(M,B)] java.lang.Double

```
DECIMAL[(M[,D])]   java.math.BigDecimal

DATE               java.sql.Date
```

DATETIME	DATETIME	java.sql.Timestamp
TIMESTAMP[(M)]	TIMESTAMP	java.sql.Timestamp

TIME

java.sql.Time

YYEEAARR[(2|4)] If yearIsDateType configuration property is set to

CCjHHaAAvRRa(M).lang.String(unlesstheCharactersetfor

VVAARRCCHHAARR(M)[BINARY]

java.lang.String (unless the character set for

BBb
IIy
NNt
AAe
RR[
YY]
(
M
)

VVb
AAy
RRt
BBe
II[
NN]
AA
RR
YY
(
M
)

TTb
IIy
NNt
YYe
BB[
LL]
OO
BB

TVIANRYCTHEAXRT java.lang.String

BBLLOOBB bytes[]

TVEAXRTC java.lang.String

MMbEEyDDtIIeUU[MM]BBLLOOBB

MVjEAaDRvICaUH.MAlaTRngEX.TString

LLbOOyNNtGGeBB[LL]OOBB

LVOANRGCTHEAXRT java.lang.String

ECNHAUAMR('value1', value2',..) java.lang.String

```
SCjava.lang.String
EHATA('Rvalue1','value2',...)
```

数据类型选择的一些建议

- **选小不选大**：一般情况下选择可以正确存储数据的最小数据类型，越小的数据类型通常更快，占用磁盘，内存和CPU缓存更小。
- **简单就好**：简单的数据类型的操作通常需要更少的CPU周期，例如：整型比字符操作代价要小得多，因为字符集和校对规则(排序规则)使字符比整型比较更加复杂。
- **尽量避免NULL**：尽量制定列为NOT NULL，除非真的需要NULL类型的值，有NULL的列值会使得索引、索引统计和值比较更加复杂。
- **浮点类型的建议统一选择decimal**
- **记录时间的建议使用int或者bigint类型，将时间转换为时间戳格式，如将时间转换为秒、毫秒，进行存储，方便走索引**

第3篇：MySQL管理员常用的一些命令

这是mysql系列第3篇文章。

环境：mysql5.7.25，cmd命令中进行演示。

在玩mysql的过程中，经常遇到有很多朋友在云上面玩mysql的时候，说我创建了一个用户为什么不能登录？为什么没有权限？等等各种问题，本文看完之后，这些都不是问题了。

本文主要内容

1. 介绍Mysql权限工作原理
2. 查看所有用户
3. 创建用户
4. 修改密码
5. 给用户授权
6. 查看用户权限
7. 撤销用户权限
8. 删除用户
9. 授权原则说明

10. 总结

Mysql权限工作原理

mysql是如何来识别一个用户的呢?

mysql为了安全性考虑，采用主机名+用户名来判断一个用户的身份，因为在互联网中很难通过用户名来判断一个用户的身份，但是我们可以通过ip或者主机名判断一台机器，某个用户通过这个机器过来的，我们可以识别为一个用户，所以**mysql中采用用户名+主机名来识别用户的身份**。当一个用户对mysql发送指令的时候，mysql就是通过用户名和来源（主机）来断定用户的权限。

Mysql权限验证分为2个阶段:

1. 阶段1：连接数据库，此时mysql会根据你的用户名及你的来源（ip或者主机名称）判断是否有权限连接
2. 阶段2：对mysql服务器发起请求操作，如create table、select、delete、update、create index等操作，此时mysql会判断你是否有权限操作这些指令

权限生效时间

用户及权限信息放在库名为mysql的库中，mysql启动时，这些内容被读进内存并且从此时生效，所以如果通过直接操作这些表来修改用户及权限信息的，需要重启`mysql`或者执行`flush privileges;`才可以生效。

用户登录之后，mysql会和当前用户之间创建一个连接，此时用户相关的权限信息都保存在这个连接中，存放在内存中，此时如果有其他地方修改了当前用户的权限，这些变更的权限会在下一次登录时才会生效。

查看mysql中所有用户

用户信息在`mysql.user`表中，如下:

```
mysql> use mysql;
Database changed
mysql> select user,host from user;
+---------------+--------------+
| user          | host         |
+---------------+--------------+
| test4         | 127.0.0.%    |
| test4         | 192.168.11.% |
| mysql.session | localhost    |
| mysql.sys     | localhost    |
| root          | localhost    |
| test2         | localhost    |
+---------------+--------------+
6 rows in set (0.00 sec)
```

创建用户

语法:

```
create user 用户名[@主机名] [identified by '密码'];
```

说明:

1. 主机名默认值为%，表示这个用户可以从任何主机连接mysql服务器
2. 密码可以省略，表示无密码登录

示例1：不指定主机名

不指定主机名时，表示这个用户可以从任何主机连接mysql服务器

```
mysql> use mysql;
Database changed

mysql> select user,host from user;
+---------------+-----------+
| user          | host      |
+---------------+-----------+
| mysql.session | localhost |
```

```
| mysql.sys     | localhost |
| root          | localhost |
+---------------+-----------+
3 rows in set (0.00 sec)

mysql> create user test1;
Query OK, 0 rows affected (0.00 sec)

mysql> select user,host from user;
+---------------+-----------+
| user          | host      |
+---------------+-----------+
| test1         | %         |
| mysql.session | localhost |
| mysql.sys     | localhost |
| root          | localhost |
+---------------+-----------+
4 rows in set (0.00 sec)

mysql> exit
Bye

C:\Users\Think>mysql -utest1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 49
Server version: 5.7.25-log MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

上面创建了用户名为test1无密码的用户，没有指定主机，可以看出host的默认值为%，表示test1可以从任何机器登录到mysql中。

用户创建之后可以在mysql库中通过 `select user,host from user;`查看到。

其他示例

```
create user 'test2'@'localhost' identified by '123';
```

说明：test2的主机为localhost表示本机，此用户只能登陆本机的mysql

```
create user 'test3'@% identified by '123';
```

说明：test3可以从任何机器连接到mysql服务器

```
create user 'test4'@'192.168.11.%' identified by '123';
```

说明：test4可以从192.168.11段的机器连接mysql

修改密码【3种方式】

方式1：通过管理员修改密码

```
SET PASSWORD FOR '用户名'@'主机' = PASSWORD('密码');
```

方式2：**create user 用户名[@主机名] [identified by '密码'];**

```
set password = password('密码');
```

方式3：通过修改**mysql.user**表修改密码

```
use mysql;
update user set authentication_string = password('321') where user =
'test1' and host = '%';
flush privileges;
```

注意：

通过表的方式修改之后，需要执行**`flush privileges;`**才能对用户生效。

5.7中user表中的authentication_string字段表示密码，老的一些版本中密码字段是password。

给用户授权

创建用户之后，需要给用户授权，才有意义。

语法：

```
grant privileges ON database.table TO 'username'[@'host'] [with grant option]
```

grant命令说明：

- priveleges (权限列表)，可以是`all`，表示所有权限，也可以是`select`、`update`等权限，多个权限之间用逗号分开。
- ON 用来指定权限针对哪些库和表，格式为`数据库.表名` ，点号前面用来指定数据库名，点号后面用来指定表名，`*.*` 表示所有数据库所有表。
- TO 表示将权限赋予某个用户, 格式为`username@host`，@前面为用户名，@后面接限制的主机，可以是IP、IP段、域名以及%，%表示任何地方。
- WITH GRANT OPTION 这个选项表示该用户可以将自己拥有的权限授权给别人。注意：经常有人在创建操作用户的时候不指定WITH GRANT OPTION选项导致后来该用户不能使用GRANT命令创建用户或者给其它用户授权。 备注：可以使用*GRANT*重复给用户添加权限，权限叠加，比如你先给用户添加一个*select*权限，然后又给用户添加一个*insert*权限，那么该用户就同时拥有了*select*和*insert*权限。

示例：

```
grant all on *.* to 'test1'@‘%’;
```

说明：给test1授权可以操作所有库所有权限，相当于dba

```
grant select on seata.* to 'test1'@'%';
```

说明：test1可以对seata库中所有的表执行select

```
grant select,update on seata.* to 'test1'@'%';
```

说明：test1可以对seata库中所有的表执行select、update

```
grant select(user,host) on mysql.user to 'test1'@'localhost';
```

说明：test1用户只能查询mysql.user表的user,host字段

查看用户有哪些权限

show grants for '用户名'[@'主机']

主机可以省略，默认值为%，示例：

```
mysql> show grants for 'test1'@'localhost';
+-----------------------------------------------------------------------+
| Grants for test1@localhost                                            |
+-----------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'test1'@'localhost'                             |
| GRANT SELECT (host, user) ON `mysql`.`user` TO 'test1'@'localhost' |
+-----------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

show grants;

查看当前用户的权限，如：

```
mysql> show grants;
+-----------------------------------------------------------------------
+
| Grants for root@localhost
|
+-----------------------------------------------------------------------
+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION
|
| GRANT ALL PRIVILEGES ON `test`.* TO 'root'@'localhost'
|
| GRANT DELETE ON `seata`.* TO 'root'@'localhost'
|
| GRANT PROXY ON ''@'' TO 'root'@'localhost' WITH GRANT OPTION
```

```
|
+-------------------------------------------------------------------
+
4 rows in set (0.00 sec)
```

撤销用户的权限

语法

```
revoke privileges ON database.table FROM '用户名'[@'主机'];
```

可以先通过show grants命令查询一下用户对于的权限，然后使用revoke命令撤销用户对应的权限，示例：

```
mysql> show grants for 'test1'@'localhost';
+-----------------------------------------------------------------------+
| Grants for test1@localhost                                            |
+-----------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'test1'@'localhost'                             |
| GRANT SELECT (host, user) ON `mysql`.`user` TO 'test1'@'localhost' |
+-----------------------------------------------------------------------+
2 rows in set (0.00 sec)

mysql> revoke select(host) on mysql.user from test1@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for 'test1'@'localhost';
+-------------------------------------------------------------+
| Grants for test1@localhost                                  |
+-------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'test1'@'localhost'                   |
| GRANT SELECT (user) ON `mysql`.`user` TO 'test1'@'localhost' |
+-------------------------------------------------------------+
2 rows in set (0.00 sec)
```

上面我们先通过grants命令查看test1的权限，然后调用revoke命令撤销对mysql.user表host字段的查询权限，最后又通过grants命令查看了test1的权限，和预期结果一致。

删除用户【2种方式】

方式1：

drop user '用户名'[@'主机']，示例：

```
mysql> drop user test1@localhost;
Query OK, 0 rows affected (0.00 sec)
```

drop的方式删除用户之后，用户下次登录就会起效。

方式2：

通过删除mysql.user表数据的方式删除，如下：

```
delete from user where user='用户名' and host='主机';
flush privileges;
```

注意通过表的方式删除的，需要调用`flush privileges;`刷新权限信息（权限启动的时候在内存中保存着，通过表的方式修改之后需要刷新一下）。

授权原则说明

- 只授予能满足需要的最小权限，防止用户干坏事，比如用户只是需要查询，那就只给select权限就可以了，不要给用户赋予update、insert或者delete权限
- 创建用户的时候限制用户的登录主机，一般是限制成指定IP或者内网IP段
- 初始化数据库的时候删除没有密码的用户，安装完数据库的时候会自动创建一些用户，这些用户默认没有密码
- 为每个用户设置满足密码复杂度的密码
- 定期清理不需要的用户，回收权限或者删除用户

总结

1. 通过命令的方式操作用户和权限不需要刷新，下次登录自动生效
2. 通过操作mysql库中表的方式修改、用户信息，需要调用`flush privileges;`刷新一下，下次登录自动生效
3. mysql识别用户身份的方式是：用户名+主机
4. 本文中讲到的一些指令中带主机的，主机都可以省略，默认值为%，表示所有机器
5. mysql中用户和权限的信息在库名为mysql的库中

Mysql系列目录

1. 第**1**天：**mysql**基础知识
2. 第**2**天：详解**mysql**数据类型（重点）

mysql系列大概有**20**多篇，喜欢的请关注一下！

第4篇：DDL常见操作汇总

这是Mysql系列第4篇。

环境：mysql5.7.25，cmd命令中进行演示。

DDL：Data Define Language数据定义语言，主要用来对数据库、表进行一些管理操作。

如：建库、删库、建表、修改表、删除表、对列的增删改等等。

文中涉及到的语法用[]包含的内容属于可选项，下面做详细说明。

库的管理

创建库

```
create database [if not exists] 库名;
```

删除库

```
drop databases [if exists] 库名;
```

建库通用的写法

```
drop database if exists 旧库名;
create database 新库名;
```

示例

```
mysql> show databases like 'javacode2018';
+--------------------------+
| Database (javacode2018) |
+--------------------------+
| javacode2018             |
+--------------------------+
1 row in set (0.00 sec)

mysql> drop database if exists javacode2018;
Query OK, 0 rows affected (0.00 sec)

mysql> show databases like 'javacode2018';
Empty set (0.00 sec)

mysql> create database javacode2018;
Query OK, 1 row affected (0.00 sec)
```

show databases like 'javacode2018';列出javacode2018库信息。

表管理

创建表

```
create table 表名(
  字段名1 类型[(宽度)] [约束条件] [comment '字段说明'],
  字段名2 类型[(宽度)] [约束条件] [comment '字段说明'],
  字段名3 类型[(宽度)] [约束条件] [comment '字段说明']
)[表的一些设置];
```

注意:

1. 在同一张表中，字段名不能相同
2. 宽度和约束条件为可选参数，字段名和类型是必须的
3. 最后一个字段后不能加逗号
4. 类型是用来限制 字段 必须以何种数据类型来存储记录
5. 类型其实也是对字段的约束(约束字段下的记录必须为XX类型)
6. 类型后写的 约束条件 是在类型之外的 额外添加的约束

约束说明

not null：标识该字段不能为空

```
mysql> create table test1(a int not null comment '字段a');
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test1 values (null);
ERROR 1048 (23000): Column 'a' cannot be null
mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test1;
+---+
| a |
+---+
| 1 |
```

```
+---+
1 row in set (0.00 sec)
```

default value：为该字段设置默认值，默认值为value

```
mysql> drop table IF EXISTS test2;
Query OK, 0 rows affected (0.01 sec)

mysql> create table test2(
    ->   a int not null comment '字段a',
    ->   b int not null default 0 comment '字段b'
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> insert into test2(a) values (1);
Query OK, 1 row affected (0.00 sec)

mysql> select *from test2;
+---+---+
| a | b |
+---+---+
| 1 | 0 |
+---+---+
1 row in set (0.00 sec)
```

上面插入时未设置b的值，自动取默认值0

primary key：标识该字段为该表的主键，可以唯一的标识记录，插入重复的会报错

两种写法，如下：

方式1：跟在列后，如下：

```
mysql> drop table IF EXISTS test3;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> create table test3(
    ->   a int not null comment '字段a' primary key
```

```
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test3 (a) values (1);
Query OK, 1 row affected (0.01 sec)

mysql> insert into test3 (a) values (1);
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

方式2：在所有列定义之后定义，如下：

```
mysql> drop table IF EXISTS test4;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> create table test4(
    ->   a int not null comment '字段a',
    ->   b int not null default 0 comment '字段b',
    ->   primary key(a)
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> insert into test4(a,b) values (1,1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test4(a,b) values (1,2);
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

插入重复的值，会报违法主键约束

方式2支持多字段作为主键，多个之间用逗号隔开，语法：primary key(字段1,字段2,字段n)，示例：

```
mysql> drop table IF EXISTS test7;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test7(
    ->    a int not null comment '字段a',
    ->    b int not null comment '字段b',
    ->   PRIMARY KEY (a,b)
```

```
    ->  );
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> insert into test7(a,b) VALUES (1,1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test7(a,b) VALUES (1,1);
ERROR 1062 (23000): Duplicate entry '1-1' for key 'PRIMARY'
```

foreign key：为表中的字段设置外键

语法：**foreign key(当前表的列名) references 引用的外键表(外键表中字段名称)**

```
mysql> drop table IF EXISTS test6;
Query OK, 0 rows affected (0.01 sec)

mysql> drop table IF EXISTS test5;
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> create table test5(
    ->   a int not null comment '字段a' primary key
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> create table test6(
    ->   b int not null comment '字段b',
    ->   ts5_a int not null,
    ->   foreign key(ts5_a) references test5(a)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test5 (a) values (1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test6 (b,test6.ts5_a) values (1,1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into test6 (b,test6.ts5_a) values (2,2);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`javacode2018`.`test6`, CONSTRAINT `test6_ibfk_1`
FOREIGN KEY (`ts5_a`) REFERENCES `test5` (`a`))
```

说明：表示test6中ts5_a字段的值来源于表test5中的字段a。

注意几点：

- 两张表中需要建立外键关系的字段类型需要一致
- 要设置外键的字段不能为主键
- 被引用的字段需要为主键
- 被插入的值在外键表必须存在，如上面向test6中插入ts5_a为2的时候报错了，原因：2的值在test5表中不存在

unique key(uq)：标识该字段的值是唯一的

支持一个到多个字段，插入重复的值会报违反唯一约束，会插入失败。

定义有2种方式。

方式1：跟在字段后，如下：

```
mysql> drop table IF EXISTS test8;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test8(
    ->     a int not null comment '字段a' unique key
    ->   );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> insert into test8(a) VALUES (1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into test8(a) VALUES (1);
ERROR 1062 (23000): Duplicate entry '1' for key 'a'
```

方式2：所有列定义之后定义，如下：

```
mysql> drop table IF EXISTS test9;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test9(
    ->    a int not null comment '字段a',
    ->   unique key(a)
    ->  );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> insert into test9(a) VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test9(a) VALUES (1);
ERROR 1062 (23000): Duplicate entry '1' for key 'a'
```

方式2支持多字段，多个之间用逗号隔开，语法：primary key(字段1,字段2,字段n)，示例：

```
mysql> drop table IF EXISTS test10;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test10(
    ->   a int not null comment '字段a',
    ->   b int not null comment '字段b',
    ->   unique key(a,b)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> insert into test10(a,b) VALUES (1,1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into test10(a,b) VALUES (1,1);
ERROR 1062 (23000): Duplicate entry '1-1' for key 'a'
```

auto_increment：标识该字段的值自动增长（整数类型，而且为主键）

```
mysql> drop table IF EXISTS test11;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test11(
    ->   a int not null AUTO_INCREMENT PRIMARY KEY comment '字段a',
    ->   b int not null comment '字段b'
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> insert into test11(b) VALUES (10);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test11(b) VALUES (20);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test11;
+---+----+
| a | b  |
+---+----+
| 1 | 10 |
| 2 | 20 |
+---+----+
2 rows in set (0.00 sec)
```

字段a为自动增长，默认值从1开始，每次+1

关于自动增长字段的初始值、步长可以在mysql中进行设置，比如设置初始值为1万，每次增长10

注意：

自增长列当前值存储在内存中，数据库每次重启之后，会查询当前表中自增列的最大值作为当前值，如果表数据被清空之后，数据库重启了，自增列的值将从初始值开始

我们来演示一下：

```
mysql> delete from test11;
Query OK, 2 rows affected (0.00 sec)

mysql> insert into test11(b) VALUES (10);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test11;
+---+----+
| a | b  |
+---+----+
| 3 | 10 |
+---+----+
1 row in set (0.00 sec)
```

上面删除了test11数据，然后插入了一条，a的值为3，执行下面操作：

> 删除test11数据，重启mysql，插入数据，然后看a的值是不是被初始化了？如下：

```
mysql> delete from test11;
Query OK, 1 row affected (0.00 sec)

mysql> select * from test11;
Empty set (0.00 sec)

mysql> exit
Bye

C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。


C:\Windows\system32>net start mysql
mysql 服务正在启动 .
```

mysql 服务已经启动成功。

```
C:\Windows\system32>mysql -uroot -p
Enter password: *******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.25-log MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> use javacode2018;
Database changed
mysql> select * from test11;
Empty set (0.01 sec)

mysql> insert into test11 (b) value (100);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test11;
+---+-----+
| a | b   |
+---+-----+
| 1 | 100 |
+---+-----+
1 row in set (0.00 sec)
```

删除表

```
drop table [if exists] 表名;
```

修改表名

```
alter table 表名 rename [to] 新表名;
```

表设置备注

```
alter table 表名 comment '备注信息';
```

复制表

只复制表结构

```
create table 表名 like 被复制的表名;
```

如：

```
mysql> create table test12 like test11;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from test12;
Empty set (0.00 sec)

mysql> show create table test12;
+--------+-------+
| Table  | Create Table
+--------+-------+
| test12 | CREATE TABLE `test12` (
  `a` int(11) NOT NULL AUTO_INCREMENT COMMENT '字段a',
  `b` int(11) NOT NULL COMMENT '字段b',
  PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8     |
+--------+-------+
1 row in set (0.00 sec)
```

复制表结构+数据

```
create table 表名 [as] select 字段,... from 被复制的表 [where 条件];
```

如：

```
mysql> create table test13 as select * from test11;
Query OK, 1 row affected (0.02 sec)
```

```
Records: 1  Duplicates: 0  Warnings: 0

mysql> select * from test13;
+---+-----+
| a | b   |
+---+-----+
| 1 | 100 |
+---+-----+
1 row in set (0.00 sec)
```

表结构和数据都过来了。

表中列的管理

添加列

```
alter table 表名 add column 列名 类型 [列约束];
```

示例:

```
mysql> drop table IF EXISTS test14;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql>
mysql> create table test14(
    ->   a int not null AUTO_INCREMENT PRIMARY KEY comment '字段a'
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> alter table test14 add column b int not null default 0 comment
'字段b';
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> alter table test14 add column c int not null default 0 comment
'字段c';
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> insert into test14(b) values (10);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test14;
c
+---+----+---+
| a | b  | c |
+---+----+---+
| 1 | 10 | 0 |
+---+----+---+
1 row in set (0.00 sec)
```

修改列

```
alter table 表名 modify column 列名 新类型 [约束];
或者
alter table 表名 change column 列名 新列名 新类型 [约束];
```

2种方式区别：modify不能修改列名，change可以修改列名

我们看一下test14的表结构：

```
mysql> show create table test14;
+--------+--------+
| Table  | Create Table |
+--------+--------+
| test14 | CREATE TABLE `test14` (
  `a` int(11) NOT NULL AUTO_INCREMENT COMMENT '字段a',
  `b` int(11) NOT NULL DEFAULT '0' COMMENT '字段b',
  `c` int(11) NOT NULL DEFAULT '0' COMMENT '字段c',
  PRIMARY KEY (`a`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8        |
+--------+--------+
1 row in set (0.00 sec)
```

我们将字段c名字及类型修改一下，如下：

```
mysql> alter table test14 change column c d varchar(10) not null
default '' comment '字段d';
Query OK, 0 rows affected (0.01 sec)
```

```
Records: 0  Duplicates: 0  Warnings: 0

mysql> show create table test14;
;;
+--------+--------+
| Table  | Create Table |
+--------+--------+
| test14 | CREATE TABLE `test14` (
  `a` int(11) NOT NULL AUTO_INCREMENT COMMENT '字段a',
  `b` int(11) NOT NULL DEFAULT '0' COMMENT '字段b',
  `d` varchar(10) NOT NULL DEFAULT '' COMMENT '字段d',
  PRIMARY KEY (`a`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8        |
+--------+--------+
1 row in set (0.00 sec)
```

删除列

```
alter table 表名drop column 列名;
```

示例:

```
mysql> alter table test14 drop column d;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show create table test14;
+--------+--------+
| Table  | Create Table |
+--------+--------+
| test14 | CREATE TABLE `test14` (
  `a` int(11) NOT NULL AUTO_INCREMENT COMMENT '字段a',
  `b` int(11) NOT NULL DEFAULT '0' COMMENT '字段b',
  PRIMARY KEY (`a`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8      |
+--------+--------+
1 row in set (0.00 sec)
```

Mysql系列目录

1. 第**1**天：**mysql**基础知识
2. 第**2**天：详解**mysql**数据类型（重点）
3. 第**3**天：管理员必备技能**(**必须掌握**)**

mysql系列大概有**20**多篇，喜欢的请关注一下!

第5篇：DML常见操作

这是Mysql系列第5篇。

环境：mysql5.7.25，cmd命令中进行演示。

DML(Data Manipulation Language)数据操作语言，以INSERT、UPDATE、DELETE三种指令为核心，分别代表插入、更新与删除，**是必须要掌握的指令**，DML和SQL中的select熟称CRUD（增删改查）。

文中涉及到的语法用[]包含的内容属于可选项，下面做详细说明。

插入操作

插入单行2种方式

方式1

```
insert into 表名[(字段,字段)] values (值,值);
```

说明:

值和字段需要一一对应

如果是字符型或日期类型，值需要用单引号引起来；如果是数值类型，不需要用单引号

字段和值的个数必须一致，位置对应

字段如果不能为空，则必须插入值

可以为空的字段可以不用插入值，但需要注意：字段和值都不写；或字段写上，值用null代替

表名后面的字段可以省略不写，此时表示所有字段，顺序和表中字段顺序一致。

方式2

```
insert into 表名 set 字段= 值,字段= 值;
```

方式2不常见，建议使用方式1

批量插入2种方式

方式1

```
insert into 表名[(字段,字段)] values (值,值),(值,值),(值,值);
```

方式2

```
insert into 表[(字段,字段)]
数据来源select语句;
```

说明:

数据来源select语句可以有很多种写法，需要注意：select返回的结果和插入数据的字段数量、顺序、类型需要一致。

关于select的写法后面文章会详细介绍。

如：

```
-- 删除test1
drop table if exists test1;
-- 创建test1
create table test1(a int,b int);
-- 删除test2
drop table if exists test2;
-- 创建test2
create table test2(c1 int,c2 int,c3 int);
-- 向test2中插入数据
insert into test2 values (100,101,102),(200,201,202),(300,301,302),
(400,401,402);
-- 向test1中插入数据
insert into test1 (a,b) select 1,1 union all select 2,2 union all
select 2,2;
-- 向test1插入数据，数据来源于test2表
insert into test1 (a,b) select c2,c3 from test2 where c1>=200;

select * from test1;

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    2 |
|  201 |  202 |
|  301 |  302 |
|  401 |  402 |

mysql> select * from test2;
+------+------+------+
| c1   | c2   | c3   |
```

```
+------+------+------+
|  100 |  101 |  102 |
|  200 |  201 |  202 |
|  300 |  301 |  302 |
|  400 |  401 |  402 |
+------+------+------+
4 rows in set (0.00 sec)
```

数据更新

单表更新

语法：

```
update 表名[[as] 别名] set [别名.]字段= 值,[别名.]字段= 值[where条件];
```

有些表名可能名称比较长，为了方便操作，可以给这个表名起个简单的别名，更方便操作一些。

如果无别名的时候，表名就是别名。

示例：

```
mysql> update test1 t set t.a = 2;
Query OK, 4 rows affected (0.00 sec)
Rows matched: 6  Changed: 4  Warnings: 0

mysql> update test1 as t set t.a = 3;
Query OK, 6 rows affected (0.00 sec)
Rows matched: 6  Changed: 6  Warnings: 0

mysql> update test1 set a = 1,b=2;
Query OK, 6 rows affected (0.00 sec)
Rows matched: 6  Changed: 6  Warnings: 0
```

多表更新

可以同时更新多个表中的数据

语法：

```
update 表1 [[as] 别名1],表名2 [[as] 别名2]
set [别名.]字段= 值,[别名.]字段= 值
[where条件]
```

示例：

```
-- 无别名方式
update test1,test2 set test1.a = 2 ,test1.b = 2, test2.c1 = 10;
-- 无别名方式
update test1,test2 set test1.a = 2 ,test1.b = 2, test2.c1 = 10 where
test1.a = test2.c1;
-- 别名方式更新
update test1 t1,test2 t2 set t1.a = 2 ,t1.b = 2, t2.c1 = 10 where t1.a
= t2.c1;
-- 别名的方式更新多个表的多个字段
update test1 as t1,test2 t2 set t1.a = 2 ,t1.b = 2, t2.c1 = 10 where
t1.a = t2.c1;
```

使用建议

建议采用单表方式更新，方便维护。

删除数据操作

使用delete删除

*delete*单表删除

```
delete [别名] from 表名[[as] 别名] [where条件];
```

注意：

如果无别名的时候，表名就是别名

如果有别名，delete后面必须写别名

如果没有别名，delete后面的别名可以省略不写。

示例

```
-- 删除test1表所有记录
delete from test1;
-- 删除test1表所有记录
delete test1 from test1;
-- 有别名的方式，删除test1表所有记录
delete t1 from test1 t1;
-- 有别名的方式删除满足条件的记录
delete t1 from test1 t1 where t1.a>100;
```

上面的4种写法，大家可以认真看一下。

多表删除

可以同时删除多个表中的记录，语法如下：

```
delete [别名1,别名2] from 表1 [[as] 别名1],表2 [[as] 别名2] [where条件];
```

说明：

别名可以省略不写，但是需要在delete后面跟上表名，多个表名之间用逗号隔开。

示例1

```
delete t1 from test1 t1,test2 t2 where t1.a=t2.c2;
```

删除test1表中的记录，条件是这些记录的字段a在test.c2中存在的记录

看一下运行效果：

```
-- 删除test1
drop table if exists test1;
-- 创建test1
create table test1(a int,b int);
-- 删除test2
drop table if exists test2;
-- 创建test2
create table test2(c1 int,c2 int,c3 int);
-- 向test2中插入数据
```

```
insert into test2 values (100,101,102),(200,201,202),(300,301,302),
(400,401,402);
-- 向test1中插入数据
insert into test1 (a,b) select 1,1 union all select 2,2 union all
select 2,2;
-- 向test1插入数据，数据来源于test2表
insert into test1 (a,b) select c2,c3 from test2 where c1>=200;

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    2 |
|  201 |  202 |
|  301 |  302 |
|  401 |  402 |

mysql> select * from test2;
+------+------+------+
| c1   | c2   | c3   |
+------+------+------+
|  100 |  101 |  102 |
|  200 |  201 |  202 |
|  300 |  301 |  302 |
|  400 |  401 |  402 |
+------+------+------+
4 rows in set (0.00 sec)

mysql> delete t1 from test1 t1,test2 t2 where t1.a=t2.c2;
Query OK, 3 rows affected (0.00 sec)

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    2 |
```

```
+------+------+
3 rows in set (0.00 sec)
```

从上面的输出中可以看到test1表中3条记录被删除了。

示例2

```
delete t2,t1 from test1 t1,test2 t2 where t1.a=t2.c2;
```

同时对2个表进行删除，条件是test.a=test.c2的记录

看一下运行效果：

```
-- 删除test1
drop table if exists test1;
-- 创建test1
create table test1(a int,b int);
-- 删除test2
drop table if exists test2;
-- 创建test2
create table test2(c1 int,c2 int,c3 int);
-- 向test2中插入数据
insert into test2 values (100,101,102),(200,201,202),(300,301,302),
(400,401,402);
-- 向test1中插入数据
insert into test1 (a,b) select 1,1 union all select 2,2 union all
select 2,2;
-- 向test1插入数据，数据来源于test2表
insert into test1 (a,b) select c2,c3 from test2 where c1>=200;

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    2 |
|  201 |  202 |
|  301 |  302 |
```

```
|  401 |  402 |
+------+------+
6 rows in set (0.00 sec)

mysql> select * from test2;
+------+------+------+
| c1   | c2   | c3   |
+------+------+------+
|  100 |  101 |  102 |
|  200 |  201 |  202 |
|  300 |  301 |  302 |
|  400 |  401 |  402 |
+------+------+------+
4 rows in set (0.00 sec)

mysql> delete t2,t1 from test1 t1,test2 t2 where t1.a=t2.c2;
Query OK, 6 rows affected (0.00 sec)

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    2 |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test2;
+------+------+------+
| c1   | c2   | c3   |
+------+------+------+
|  100 |  101 |  102 |
+------+------+------+
1 row in set (0.00 sec)
```

从输出中可以看出test1和test2总计6条记录被删除了。

平时我们用的比较多的方式是`delete from` 表名这种语法，上面我们介绍了再**delete**后面跟上表名的用法，大家可以在回顾一下，加深记忆。

使用truncate删除

语法

```
truncate 表名;
```

drop，truncate，delete区别

- drop (删除表)：删除内容和定义，释放空间，简单来说就是**把整个表去掉**，以后要新增数据是不可能的，除非新增一个表。

  drop语句将删除表的结构被依赖的约束（constrain），触发器（trigger）索引（index），依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

  如果要删除表定义及其数据，请使用 drop table 语句。

- truncate (清空表中的数据)：删除内容、释放空间但不删除定义(**保留表的数据结构**)，与drop不同的是，只是清空表数据而已。

  注意：truncate不能删除具体行数据，要删就要把整个表清空了。

- delete (删除表中的数据)：delete 语句用于**删除表中的行**。delete语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存，以便进行进行回滚操作。

  truncate与不带where的delete ： 只删除数据，而不删除表的结构（定义）

  truncate table 删除表中的所有行，但表结构及其列、约束、索引等保持不变。

  对于由foreign key约束引用的表，不能使用truncate table ，而应使用不带where子句的delete语句。由于truncate table 记录在日志中，所以它不能激活触发器。

  delete语句是数据库操作语言(dml)，这个操作会放到 rollback segement 中，**事务提交之后才生效**；如果有相应的 trigger，执行的时候将被触发。

truncate、drop 是数据库定义语言(ddl)，操作立即生效，原数据不放到 rollback segment 中，不能回滚，操作不触发 trigger。

如果有自增列，truncate方式删除之后，自增列的值会被初始化，delete方式要分情况（如果数据库被重启了，自增列值也会被初始化，数据库未被重启，则不变）

- **如果要删除表定义及其数据，请使用 drop table 语句**
- **安全性：小心使用 drop 和 truncate，尤其没有备份的时候，否则哭都来不及**
- **删除速度，一般来说: drop> truncate > delete**

t
r
ud
ne
dcl
rae
ott
pee

条不不支
件支支持
册持持
陈

册支不不
陈持支支
表 持持
纪
柊

事不支
务支持
的持
方
式
册
防

触否是
发
触
发
器

Mysql系列目录

1. 第1天：mysql基础知识
2. 第2天：详解mysql数据类型（重点）
3. 第3天：管理员必备技能(必须掌握)
4. 第4天：DDL常见操作

mysql系列大概有20多篇，喜欢的请关注一下！

第6篇：select查下基础篇

这是Mysql系列第6篇。

环境：mysql5.7.25，cmd命令中进行演示。

DQL(Data QueryLanguage)：数据查询语言，通俗点讲就是从数据库获取数据的，按照DQL的语法给数据库发送一条指令，数据库将按需求返回数据。

DQL分多篇来说，本文属于第1篇。

基本语法

```
select 查询的列 from 表名;
```

注意：

select语句中不区分大小写，SELECT和select、FROM和from效果一样。

查询的结果放在一个表格中，表格的第1行称为列头，第2行开始是数据，类属于一个二维数组。

查询常量

```
select 常量值1,常量值2,常量值3;
```

如：

```
mysql> select 1,'b';
+---+---+
| 1 | b |
+---+---+
| 1 | b |
```

```
+---+---+
1 row in set (0.00 sec)
```

查询表达式

select 表达式;

如:

```
mysql> select 1+2,3*10,10/3;
+-----+------+--------+
| 1+2 | 3*10 | 10/3   |
+-----+------+--------+
|   3 |   30 | 3.3333 |
+-----+------+--------+
1 row in set (0.00 sec)
```

查询函数

select 函数;

如:

```
mysql> select mod(10,4),isnull(null),ifnull(null,'第一个参数为空返回这个
值'),ifnull(1,'第一个参数为空返回这个值，否知返回第一个参数');
+-----------+--------------
+-----------------------------------------------------
+-------------------------------------------------------------------------
-----------+
| mod(10,4) | isnull(null) | ifnull(null,'第一个参数为空返回这个值')
| ifnull(1,'第一个参数为空返回这个值，否知返回第一个参数')
|
+-----------+--------------
+-----------------------------------------------------
+-------------------------------------------------------------------------
-----------+
|         2 |            1 | 第一个参数为空返回这个值            | 1
|
+-----------+--------------
```

```
+-----------------------------------------------------
+-------------------------------------------------------------------------
-----------+
1 row in set (0.00 sec)
```

说明一下：

mod函数，对两个参数取模运算。

isnull函数，判断参数是否为空，若为空返回1，否则返回0。

ifnull函数，2个参数，判断第一个参数是否为空，如果为空返回第2个参数的值，否则返回第1个参数的值。

查询指定的字段

```
select 字段1,字段2,字段3 from 表名;
```

如：

```
mysql> drop table if exists test1;
Query OK, 0 rows affected (0.01 sec)

mysql> create table test1(a int not null comment '字段a',b varchar(10)
not null default '' comment '字段b');
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test1 values(1,'a'),(2,'b'),(3,'c');
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select a,b from test1;
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | b |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)
```

说明：

test1表有两个字段a、b，`select a,b from test1;`用于查询test1中两个字段的数据。

查询所有列

```
select * from 表名
```

说明：

*表示返回表中所有字段。

如：

```
mysql> select * from test1;
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | b |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)
```

列别名

在创建数据表时，一般都会使用英文单词或英文单词缩写来设置字段名，在查询时列名都会以英文的形式显示，这样会给用户查看数据带来不便，这种情况可以使用别名来代替英文列名，增强阅读性。

语法：

```
select 列[as] 别名from 表;
```

使用双引号创建别名:

```
mysql> select a "列1",b "列2" from test1;
+------+------+
| 列1  | 列2  |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)
```

使用单引号创建别名:

```
mysql> select a '列1',b '列2' from test1;;
+------+------+
| 列1  | 列2  |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)
```

不用引号创建别名:

```
mysql> select a 列1,b 列2 from test1;
+------+------+
| 列1  | 列2  |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)
```

使用**as**创建别名:

```
mysql> select a as 列1,b as 列2 from test1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near '2 from test1' at line 1
mysql> select a as 列1,b as '列2' from test1;
```

```
+------+-------+
| 列1  | 列2   |
+------+-------+
|    1 | a     |
|    2 | b     |
|    3 | c     |
+------+-------+
3 rows in set (0.00 sec)
```

别名中有特殊符号的，比如空格，此时别名必须用引号引起来。

懵逼示例，看效果：

```
mysql> select 'a' 'b';
+----+
| a  |
+----+
| ab |
+----+
1 row in set (0.00 sec)

mysql> select 'a' b;
+---+
| b |
+---+
| a |
+---+
1 row in set (0.00 sec)

mysql> select 'a' "b";
+----+
| a  |
+----+
| ab |
+----+
1 row in set (0.00 sec)

mysql> select 'a' as "b";
+---+
```

```
| b |
+---+
| a |
+---+
1 row in set (0.00 sec)
```

认真看一下第1个和第3个返回的结果（列头和数据），是不是懵逼状态，建议这种的最好使用**as**，as后面跟上别名。

表别名

```
select 别名.字段,别名.* from 表名[as] 别名;
```

如：

```
mysql> select t.a,t.b from test1 as t;
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | b |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)

mysql> select t.a as '列1',t.b as 列2 from test1 as t;
+-------+------+
| 列1   | 列2  |
+-------+------+
|     1 | a    |
|     2 | b    |
|     3 | c    |
+-------+------+
3 rows in set (0.00 sec)

mysql> select t.* from test1 as t;                    ;;
+---+---+
| a | b |
+---+---+
```

```
| 1 | a |
| 2 | b |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)

mysql> select * from test1 as t;
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | b |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)
```

总结

- 建议别名前面跟上**as**关键字
- 查询数据的时候，避免使用**select ***，建议需要什么字段写什么字段

Mysql系列目录

1. 第**1**篇：**mysql**基础知识
2. 第**2**篇：详解**mysql**数据类型（重点）
3. 第**3**篇：管理员必备技能**(**必须掌握**)**
4. 第**4**篇：**DDL**常见操作
5. 第**5**篇：**DML**操作汇总（**insert,update,delete**）

mysql系列大概有**20**多篇，喜欢的请关注一下！

第7篇：select条件查询

这是Mysql系列第7篇。

环境：mysql5.7.25，cmd命令中进行演示。

电商中：我们想查看某个用户所有的订单，或者想查看某个用户在某个时间段内所有的订单，此时我们需要对订单表数据进行筛选，按照用户、时间进行过滤，得到我们期望的结果。

此时我们需要使用条件查询来对指定表进行操作，我们需要了解sql中的条件查询常见的玩法。

本篇内容

1. 条件查询语法
2. 条件查询运算符详解（=、<、>、>=、<=、<>、!=）
3. 逻辑查询运算符详解（and、or）
4. like模糊查询介绍
5. between and查询
6. in、not in查询

7. NULL值存在的坑
8. is null/is not null（NULL值专用查询）
9. <=>（安全等于）运算符
10. 经典面试题

条件查询

语法：

```
select 列名 from 表名 where 列 运算符 值
```

说明：

注意关键字where，where后面跟上一个或者多个条件，条件是对前面数据的过滤，只有满足where后面条件的数据才会被返回。

下面介绍常见的查询运算符。

条件查询运算符

操作符	描述
=	等于
<> 或者 !=	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于

等于（=）

```
select 列名 from 表名 where 列 = 值;
```

说明:

查询出指定的列和对应的值相等的记录。

值如果是字符串类型，需要用单引号或者双引号引起来。

示例：

```
mysql> create table test1 (a int,b varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test1 values (1,'abc'),(2,'bbb');
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 | abc  |
|    2 | bbb  |
+------+------+
2 rows in set (0.00 sec)

mysql> select * from test1 where a=2;
+------+------+
| a    | b    |
+------+------+
|    2 | bbb  |
+------+------+
1 row in set (0.00 sec)

mysql> select * from test1 where b = 'abc';
+------+------+
| a    | b    |
+------+------+
|    1 | abc  |
+------+------+
1 row in set (0.00 sec)

mysql> select * from test1 where b = "abc";
+------+------+
| a    | b    |
+------+------+
```

```
|    1 | abc  |
+------+------+
1 row in set (0.00 sec)
```

不等于（<>、!=）

不等于有两种写法：<>或者!=

```
select 列名 from 表名 where 列 <> 值;
或者
select 列名 from 表名 where 列 != 值;
```

示例：

```
mysql> select * from test1 where a<>1;
+------+------+
| a    | b    |
+------+------+
|    2 | bbb  |
+------+------+
1 row in set (0.00 sec)

mysql> select * from test1 where a!=1;
+------+------+
| a    | b    |
+------+------+
|    2 | bbb  |
+------+------+
1 row in set (0.00 sec)
```

> 注意：
>
> <> 这个是最早的用法。
>
> !=是后来才加上的。
>
> 两者意义相同，在可移植性上前者优于后者
>
> 故而sql语句中尽量使用<>来做不等判断

大于（>）

```
select 列名 from 表名 where 列 > 值;
```

示例：

```
mysql> select * from test1 where a>1;
+------+------+
| a    | b    |
+------+------+
|    2 | bbb  |
+------+------+
1 row in set (0.00 sec)

mysql> select * from test1 where b>'a';
+------+------+
| a    | b    |
+------+------+
|    1 | abc  |
|    2 | bbb  |
+------+------+
2 rows in set (0.00 sec)

mysql> select * from test1 where b>'ac';
+------+------+
| a    | b    |
+------+------+
|    2 | bbb  |
+------+------+
1 row in set (0.00 sec)
```

> **说明：**
>
> 数值按照大小比较。
>
> 字符按照*ASCII*码对应的值进行比较，比较时按照字符对应的位置一个字符一个字符的比较。

其他几个运算符（<、<=、>=）在此就不介绍了，用法和上面类似，大家可以自己练习一下。

逻辑查询运算符

当我们需要使用多个条件进行查询的时候，需要使用逻辑查询运算符。

逻辑运算符	描述
AND	多个条件都成立
OR	多个条件中满足一个

AND（并且）

```
select 列名 from 表名 where 条件1 and 条件2;
```

表示返回满足条件1和条件2的记录。

示例：

```
mysql> create table test3(a int not null,b varchar(10) not null);
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test3 (a,b) values (1,'a'),(2,'b'),(2,'c'),(3,'c');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from test3;
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | b |
| 2 | c |
| 3 | c |
+---+---+
4 rows in set (0.00 sec)

mysql> select * from test3 t where t.a=2 and t.b='c';
+---+---+
| a | b |
+---+---+
| 2 | c |
+---+---+
1 row in set (0.00 sec)
```

查询出了a=2 并且 b='c'的记录，返回了一条结果。

OR（或者）

```
select 列名 from 表名 where 条件1 or 条件2;
```

满足条件1或者满足条件2的记录都会被返回。

示例:

```
mysql> select * from test3;
+---+---+
| a | b |
+---+---+
```

```
| 1 | a |
| 2 | b |
| 2 | c |
| 3 | c |
+---+---+
4 rows in set (0.00 sec)

mysql> select * from test3 t where t.a=1 or t.b='c';
+---+---+
| a | b |
+---+---+
| 1 | a |
| 2 | c |
| 3 | c |
+---+---+
3 rows in set (0.00 sec)
```

查询出了a=1或者b='c'的记录，返回了3条记录。

like（模糊查询）

有个学生表，包含（学生id，年龄，姓名），当我们需要查询姓“张”的学生的时候，如何查询呢?

此时我们可以使用sql中的like关键字。语法：

```
select 列名 from 表名 where 列 like pattern;
```

pattern中可以包含通配符，有以下通配符：

%：表示匹配任意一个或多个字符

_：表示匹配任意一个字符。

学生表，查询名字姓“张”的学生，如下：

```
mysql> create table stu (id int not null comment '编号',age smallint
not null comment '年龄',name varchar(10) not null comment '姓名');
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> insert into stu values (1,22,'张三'),(2,25,'李四'),(3,26,'张学友'),(4,32,'刘德华'),(5,55,'张学良');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from stu;
+----+-----+-----------+
| id | age | name      |
+----+-----+-----------+
|  1 |  22 | 张三      |
|  2 |  25 | 李四      |
|  3 |  26 | 张学友    |
|  4 |  32 | 刘德华    |
|  5 |  55 | 张学良    |
+----+-----+-----------+
5 rows in set (0.00 sec)

mysql> select * from stu a where a.name like '张%';
+----+-----+-----------+
| id | age | name      |
+----+-----+-----------+
|  1 |  22 | 张三      |
|  3 |  26 | 张学友    |
|  5 |  55 | 张学良    |
+----+-----+-----------+
3 rows in set (0.00 sec)
```

查询名字中带有'学'的学生，'学'的位置不固定，可以这么查询，如下：

```
mysql> select * from stu a where a.name like '%学%'; ;
+----+-----+-----------+
| id | age | name      |
+----+-----+-----------+
|  3 |  26 | 张学友    |
|  5 |  55 | 张学良    |
+----+-----+-----------+
2 rows in set (0.00 sec)
```

查询姓'张'，名字2个字的学生：

```
mysql> select * from stu a where a.name like '张_';
+----+-----+--------+
| id | age | name   |
+----+-----+--------+
|  1 |  22 | 张三 |
+----+-----+--------+
1 row in set (0.00 sec)
```

上面的_代表任意一个字符，如果要查询姓'张'的3个字的学生，条件变为了'张__'，2个下划线符号。

BETWEEN AND(区间查询)

操作符 BETWEEN ... AND 会选取介于两个值之间的数据范围，这些值可以是数值、文本或者日期，属于一个闭区间查询。

```
selec 列名 from 表名 where 列名 between 值1 and 值2;
```

返回对应的列的值在[值1,值2]区间中的记录

使用between and可以提高语句的简洁度

两个临界值不要调换位置，只能是大于等于左边的值，并且小于等于右边的值。

示例：

查询年龄在[25,32]的，如下：

```
mysql> select * from stu;
+----+-----+-----------+
| id | age | name      |
+----+-----+-----------+
|  1 |  22 | 张三   |
|  2 |  25 | 李四   |
|  3 |  26 | 张学友  |
|  4 |  32 | 刘德华  |
|  5 |  55 | 张学良  |
+----+-----+-----------+
5 rows in set (0.00 sec)
```

```
mysql> select * from stu t where t.age between 25 and 32;
+----+-----+-----------+
| id | age | name      |
+----+-----+-----------+
|  2 |  25 | 李四   |
|  3 |  26 | 张学友  |
|  4 |  32 | 刘德华  |
+----+-----+-----------+
3 rows in set (0.00 sec)
```

下面两条sql效果一样

```
select * from stu t where t.age between 25 and 32;
select * from stu t where t.age >= 25 and t.age <= 32;
```

IN查询

我们需要查询年龄为10岁、15岁、20岁、30岁的人，怎么查询呢？可以用or查询，如下：

```
mysql> create table test6(id int,age smallint);
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test6 values(1,14),(2,15),(3,18),(4,20),(5,28),
(6,10),(7,10),(8,30);
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

mysql> select * from test6;
+------+------+
| id   | age  |
+------+------+
|    1 |   14 |
|    2 |   15 |
|    3 |   18 |
|    4 |   20 |
|    5 |   28 |
```

```
|     6 |    10 |
|     7 |    10 |
|     8 |    30 |
+------+------+
8 rows in set (0.00 sec)

mysql> select * from test6 t where t.age=10 or t.age=15 or t.age=20 or
t.age = 30;
+------+------+
| id   | age  |
+------+------+
|     2 |    15 |
|     4 |    20 |
|     6 |    10 |
|     7 |    10 |
|     8 |    30 |
+------+------+
5 rows in set (0.00 sec)
```

用了这么多or，有没有更简单的写法？有，用IN查询

IN 操作符允许我们在 WHERE 子句中规定多个值。

```
select 列名 from 表名 where 字段 in (值1,值2,值3,值4);
```

in 后面括号中可以包含多个值，对应记录的字段满足in中任意一个都会被返回

in列表的值类型必须一致或兼容

in列表中不支持通配符。

上面的示例用IN实现如下：

```
mysql> select * from test6 t where t.age in (10,15,20,30);
+------+------+
| id   | age  |
+------+------+
|     2 |    15 |
|     4 |    20 |
|     6 |    10 |
```

```
|     7 |    10 |
|     8 |    30 |
+-------+-------+
5 rows in set (0.00 sec)
```

相对于or简洁了很多。

NOT IN查询

not in和in刚好相反，in是列表中被匹配的都会被返回，NOT IN是和列表中都不匹配的会被返回。

```
select 列名 from 表名 where 字段 not in (值1,值2,值3,值4);
```

如查询年龄不在10、15、20、30之内的，如下：

```
mysql> select * from test6 t where t.age not in (10,15,20,30);
+------+------+
| id   | age  |
+------+------+
|    1 |   14 |
|    3 |   18 |
|    5 |   28 |
+------+------+
3 rows in set (0.00 sec)
```

NULL存在的坑

我们先看一下效果，然后在解释，示例如下：

```
mysql> create table test5 (a int not null,b int,c varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test5 values (1,2,'a'),(3,null,'b'),(4,5,null);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> select * from test5;
+---+------+------+
| a | b    | c    |
+---+------+------+
| 1 |    2 | a    |
| 3 | NULL | b    |
| 4 |    5 | NULL |
+---+------+------+
3 rows in set (0.00 sec)
```

上面我们创建了一个表test5，3个字段，a不能为空，b、c可以为空，插入了3条数据，睁大眼睛看效果了：

```
mysql> select * from test5 where b>0;
+---+------+------+
| a | b    | c    |
+---+------+------+
| 1 |    2 | a    |
| 4 |    5 | NULL |
+---+------+------+
2 rows in set (0.00 sec)

mysql> select * from test5 where b<=0;
Empty set (0.00 sec)

mysql> select * from test5 where b=NULL;
Empty set (0.00 sec)

mysql> select * from test5 t where t.b between 0 and 100;
+---+------+------+
| a | b    | c    |
+---+------+------+
| 1 |    2 | a    |
| 4 |    5 | NULL |
+---+------+------+
2 rows in set (0.00 sec)

mysql> select * from test5 where c like '%';
+---+------+------+
| a | b    | c    |
```

```
+---+------+------+
| 1 |    2 | a    |
| 3 | NULL | b    |
+---+------+------+
2 rows in set (0.00 sec)

mysql> select * from test5 where c in ('a','b',NULL);
+---+------+------+
| a | b    | c    |
+---+------+------+
| 1 |    2 | a    |
| 3 | NULL | b    |
+---+------+------+
2 rows in set (0.00 sec)

mysql> select * from test5 where c not in ('a','b',NULL);
Empty set (0.00 sec)
```

认真看一下上面的查询：

上面带有条件的查询，对字段b进行条件查询的，b的值为NULL的都没有出现。

对c字段进行like '%'查询、in、not查询，c中为NULL的记录始终没有查询出来。

between and查询，为空的记录也没有查询出来。

结论：查询运算符、like、between and、in、not in对NULL值查询不起效。

那NULL如何查询呢？继续向下看

IS NULL/IS NOT NULL（NULL值专用查询）

上面介绍的各种运算符对NULL值均不起效，mysql为我们提供了查询空值的语法：IS NULL、IS NOT NULL。

IS NULL（返回值为空的记录）

```
select 列名 from 表名 where 列 is null;
```

查询指定的列的值为NULL的记录。

如：

```
mysql> create table test7 (a int,b varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test7 (a,b) values (1,'a'),(null,'b'),(3,null),
(null,null),(4,'c');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from test7;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
| NULL | b    |
|    3 | NULL |
| NULL | NULL |
|    4 | c    |
+------+------+
5 rows in set (0.00 sec)

mysql> select * from test7 t where t.a is null;
+------+------+
| a    | b    |
+------+------+
| NULL | b    |
| NULL | NULL |
+------+------+
2 rows in set (0.00 sec)

mysql> select * from test7 t where t.a is null or t.b is null;
+------+------+
| a    | b    |
+------+------+
| NULL | b    |
|    3 | NULL |
| NULL | NULL |
+------+------+
3 rows in set (0.00 sec)
```

IS NULL（返回值不为空的记录）

```
select 列名 from 表名 where 列 is not null;
```

查询指定的列的值不为NULL的记录。

如：

```
mysql> select * from test7 t where t.a is not null;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    3 | NULL |
|    4 | c    |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test7 t where t.a is not null and t.b is not
null;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    4 | c    |
+------+------+
2 rows in set (0.00 sec)
```

<=>（安全等于）

<=>：既可以判断NULL值，又可以判断普通的数值，可读性较低，用得较少

示例：

```
mysql> create table test8 (a int,b varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test8 (a,b) values (1,'a'),(null,'b'),(3,null),
(null,null),(4,'c');
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> select * from test8;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
| NULL | b    |
|    3 | NULL |
| NULL | NULL |
|    4 | c    |
+------+------+
5 rows in set (0.00 sec)

mysql> select * from test8 t where t.a<=>null;
+------+------+
| a    | b    |
+------+------+
| NULL | b    |
| NULL | NULL |
+------+------+
2 rows in set (0.00 sec)

mysql> select * from test8 t where t.a<=>1;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
+------+------+
1 row in set (0.00 sec)
```

可以看到<=>可以将NULL查询出来。

经典面试题

下面的2个sql查询结果一样么?

```
select * from students;
select * from students where name like '%';
```

结果分2种情况:

当name没有NULL值时，返回的结果一样。

当name有NULL值时，第2个sql查询不出name为NULL的记录。

总结

- **like中的%可以匹配一个到多个任意的字符，_可以匹配任意一个字符**
- **空值查询需要使用IS NULL或者IS NOT NULL，其他查询运算符对NULL值无效**
- **建议创建表的时候，尽量设置表的字段不能为空，给字段设置一个默认值**
- **<=>（安全等于）玩玩可以，建议少使用**
- sql方面有问题的欢迎留言？或者加我微信**itsoku**交流。

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**

mysql系列大概有20多篇，喜欢的请关注一下！

第8篇：排序和分页（order by 、limit）

。 这是Mysql系列第8篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

本章内容

1. 详解排序查询
2. 详解limit
3. limit存在的坑
4. 分页查询中的坑

排序查询（order by）

电商中：我们想查看今天所有成交的订单，按照交易额从高到低排序，此时我们可以使用数据库中的排序功能来完成。

排序语法：

```
select 字段名 from 表名 order by 字段1 [asc|desc],字段2 [asc|desc];
```

需要排序的字段跟在order by之后；

asc|desc表示排序的规则，asc：升序，desc：降序，默认为asc；

支持多个字段进行排序，多字段排序之间用逗号隔开。

单字段排序

```
mysql> create table test2(a int,b varchar(10));
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test2 values (10,'jack'),(8,'tom'),(5,'ready'),
(100,'javacode');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from test2;
+------+----------+
| a    | b        |
+------+----------+
|   10 | jack     |
|    8 | tom      |
|    5 | ready    |
|  100 | javacode |
+------+----------+
4 rows in set (0.00 sec)

mysql> select * from test2 order by a asc;
+------+----------+
| a    | b        |
+------+----------+
|    5 | ready    |
|    8 | tom      |
|   10 | jack     |
|  100 | javacode |
+------+----------+
4 rows in set (0.00 sec)

mysql> select * from test2 order by a desc;
+------+----------+
| a    | b        |
+------+----------+
|  100 | javacode |
|   10 | jack     |
|    8 | tom      |
|    5 | ready    |
+------+----------+
```

```
4 rows in set (0.00 sec)

mysql> select * from test2 order by a;
+------+----------+
| a    | b        |
+------+----------+
|    5 | ready    |
|    8 | tom      |
|   10 | jack     |
|  100 | javacode |
+------+----------+
4 rows in set (0.00 sec)
```

多字段排序

比如学生表，先按学生年龄降序，年龄相同时，再按学号升序，如下：

```
mysql> create table stu(id int not null comment '学号' primary key,age
tinyint not null comment '年龄',name varchar(16) comment '姓名');
Query OK, 0 rows affected (0.01 sec)

mysql> insert into stu (id,age,name) values (1001,18,'路人甲Java'),
(1005,20,'刘德华'),(1003,18,'张学友'),(1004,20,'张国荣'),(1010,19,'梁朝
伟');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from stu;
+------+-----+---------------+
| id   | age | name          |
+------+-----+---------------+
| 1001 |  18 | 路人甲Java    |
| 1003 |  18 | 张学友   |
| 1004 |  20 | 张国荣   |
| 1005 |  20 | 刘德华   |
| 1010 |  19 | 梁朝伟   |
+------+-----+---------------+
5 rows in set (0.00 sec)
```

```
mysql> select * from stu order by age desc,id asc;
+------+-----+---------------+
| id   | age | name          |
+------+-----+---------------+
| 1004 |  20 | 张国荣   |
| 1005 |  20 | 刘德华   |
| 1010 |  19 | 梁朝伟   |
| 1001 |  18 | 路人甲Java    |
| 1003 |  18 | 张学友   |
+------+-----+---------------+
5 rows in set (0.00 sec)
```

按别名排序

```
mysql> select * from stu;
+------+-----+---------------+
| id   | age | name          |
+------+-----+---------------+
| 1001 |  18 | 路人甲Java    |
| 1003 |  18 | 张学友   |
| 1004 |  20 | 张国荣   |
| 1005 |  20 | 刘德华   |
| 1010 |  19 | 梁朝伟   |
+------+-----+---------------+
5 rows in set (0.00 sec)

mysql> select age '年龄',id as '学号' from stu order by 年龄 asc,学号
desc;
+--------+--------+
| 年龄 | 学号 |
+--------+--------+
|     18 |   1003 |
|     18 |   1001 |
|     19 |   1010 |
|     20 |   1005 |
|     20 |   1004 |
+--------+--------+
```

按函数排序

有学生表（id：编号，birth：出生日期，name：姓名），如下：

```
mysql> drop table if exists student;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE student (
    ->   id int(11) NOT NULL COMMENT '学号',
    ->   birth date NOT NULL COMMENT '出生日期',
    ->   name varchar(16) DEFAULT NULL COMMENT '姓名',
    ->   PRIMARY KEY (id)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> insert into student (id,birth,name) values
(1001,'1990-10-10','路人甲Java'),(1005,'1960-03-01','刘德华'),
(1003,'1960-08-16','张学友'),(1004,'1968-07-01','张国荣'),
(1010,'1962-05-16','梁朝伟');
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql>
mysql> SELECT * FROM student;
+------+------------+---------------+
| id   | birth      | name          |
+------+------------+---------------+
| 1001 | 1990-10-10 | 路人甲Java    |
| 1003 | 1960-08-16 | 张学友   |
| 1004 | 1968-07-01 | 张国荣   |
| 1005 | 1960-03-01 | 刘德华   |
| 1010 | 1962-05-16 | 梁朝伟   |
+------+------------+---------------+
5 rows in set (0.00 sec)
```

需求：按照出生年份升序、编号升序，查询出编号、出生日期、出生年份、姓名，2种写法如下：

```
mysql> SELECT id 编号,birth 出生日期,year(birth) 出生年份,name 姓名 from
student ORDER BY year(birth) asc,id asc;
+--------+-------------+-------------+--------------+
| 编号 | 出生日期  | 出生年份  | 姓名    |
+--------+-------------+-------------+--------------+
|   1003 | 1960-08-16   |         1960 | 张学友   |
|   1005 | 1960-03-01   |         1960 | 刘德华   |
|   1010 | 1962-05-16   |         1962 | 梁朝伟   |
|   1004 | 1968-07-01   |         1968 | 张国荣   |
|   1001 | 1990-10-10   |         1990 | 路人甲Java    |
+--------+-------------+-------------+--------------+
5 rows in set (0.00 sec)

mysql> SELECT id 编号,birth 出生日期,year(birth) 出生年份,name 姓名 from
student ORDER BY 出生年份 asc,id asc;
+--------+-------------+-------------+--------------+
| 编号 | 出生日期  | 出生年份  | 姓名    |
+--------+-------------+-------------+--------------+
|   1003 | 1960-08-16   |         1960 | 张学友   |
|   1005 | 1960-03-01   |         1960 | 刘德华   |
|   1010 | 1962-05-16   |         1962 | 梁朝伟   |
|   1004 | 1968-07-01   |         1968 | 张国荣   |
|   1001 | 1990-10-10   |         1990 | 路人甲Java    |
+--------+-------------+-------------+--------------+
5 rows in set (0.00 sec)
```

说明：

year函数：属于日期函数，可以获取对应日期中的年份。

上面使用了2种方式排序，第一种是在order by中使用了函数，第二种是使用了别名排序。

where之后进行排序

有订单数据如下：

```
mysql> drop table if exists t_order;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> create table t_order(
    ->   id int not null auto_increment comment '订单编号',
    ->   price decimal(10,2) not null default 0 comment '订单金额',
    ->   primary key(id)
    -> )comment '订单表';
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t_order (price) values (88.95),(100.68),(500),
(300),(20.88),(200.5);
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> select * from t_order;
+----+--------+
| id | price  |
+----+--------+
|  1 |  88.95 |
|  2 | 100.68 |
|  3 | 500.00 |
|  4 | 300.00 |
|  5 |  20.88 |
|  6 | 200.50 |
+----+--------+
6 rows in set (0.00 sec)
```

需求：查询订单金额>=100的，按照订单金额降序排序，显示2列数据，列头：订单编号、订单金额，如下：

```
mysql> select a.id订单编号,a.price订单金额 from t_order a where
a.price>=100 order by a.price desc;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
|            4 |       300.00 |
|            6 |       200.50 |
|            2 |       100.68 |
```

```
+--------------+--------------+
4 rows in set (0.00 sec)
```

limit介绍

limit用来限制select查询返回的行数，常用于分页等操作。

语法：

```
select 列 from 表 limit [offset,] count;
```

说明：

offset：表示偏移量，通俗点讲就是跳过多少行，offset可以省略，默认为0，表示跳过0行；范围：[0,+∞)。

count：跳过offset行之后开始取数据，取count行记录；范围：[0,+∞)。

limit中offset和count的值不能用表达式。

下面我们列一些常用的示例来加深理解。

获取前n行记录

```
select 列 from 表 limit 0,n;
或者
select 列 from 表 limit n;
```

示例，获取订单的前2条记录，如下：

```
mysql> create table t_order(
    ->   id int not null auto_increment comment '订单编号',
    ->   price decimal(10,2) not null default 0 comment '订单金额',
    ->   primary key(id)
    -> )comment '订单表';
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t_order (price) values (88.95),(100.68),(500),
(300),(20.88),(200.5);
```

```
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> select * from t_order;
+----+--------+
| id | price  |
+----+--------+
|  1 |  88.95 |
|  2 | 100.68 |
|  3 | 500.00 |
|  4 | 300.00 |
|  5 |  20.88 |
|  6 | 200.50 |
+----+--------+
6 rows in set (0.00 sec)

mysql> select a.id订单编号,a.price订单金额 from t_order a limit 2;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            1 |        88.95 |
|            2 |       100.68 |
+--------------+--------------+
2 rows in set (0.00 sec)

mysql> select a.id订单编号,a.price订单金额 from t_order a limit 0,2;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            1 |        88.95 |
|            2 |       100.68 |
+--------------+--------------+
2 rows in set (0.00 sec)
```

获取最大的一条记录

我们需要获取订单金额最大的一条记录，可以这么做：先按照金额降序，然后取第一条记录，如下：

```
mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
|            4 |       300.00 |
|            6 |       200.50 |
|            2 |       100.68 |
|            1 |        88.95 |
|            5 |        20.88 |
+--------------+--------------+
6 rows in set (0.00 sec)

mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc limit 1;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
+--------------+--------------+
1 row in set (0.00 sec)

mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc limit 0,1;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
+--------------+--------------+
1 row in set (0.00 sec)
```

获取排名第n到m的记录

我们需要先跳过n-1条记录，然后取m-n+1条记录，如下：

```
select 列 from 表 limit n-1,m-n+1;
```

如：我们想获取订单金额最高的3到5名的记录，我们需要跳过2条，然后获取3条记录，如下：

```
mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc;
+--------------+--------------+
| 订单编号   | 订单金额   |
+--------------+--------------+
|              3 |        500.00 |
|              4 |        300.00 |
|              6 |        200.50 |
|              2 |        100.68 |
|              1 |         88.95 |
|              5 |         20.88 |
+--------------+--------------+
6 rows in set (0.00 sec)

mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc limit 2,3;
+--------------+--------------+
| 订单编号   | 订单金额   |
+--------------+--------------+
|              6 |        200.50 |
|              2 |        100.68 |
|              1 |         88.95 |
+--------------+--------------+
3 rows in set (0.00 sec)
```

分页查询

开发过程中，分页我们经常使用，分页一般有2个参数：

page：表示第几页，从1开始，范围[1,+∞)

pageSize：每页显示多少条记录，范围[1,+∞)

如：page = 2，pageSize = 10，表示获取第2页10条数据。

我们使用limit实现分页，语法如下：

```
select 列 from 表名 limit (page - 1) * pageSize,pageSize;
```

需求：我们按照订单金额降序，每页显示2条，依次获取所有订单数据、第1页、第2页、第3页数据，如下：

```
mysql> select a.id 订单编号,a.price 订单金额 from t_order a order by
a.price desc;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
|            4 |       300.00 |
|            6 |       200.50 |
|            2 |       100.68 |
|            1 |        88.95 |
|            5 |        20.88 |
+--------------+--------------+
6 rows in set (0.00 sec)

mysql> select a.id 订单编号,a.price 订单金额 from t_order a order by
a.price desc limit 0,2;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            3 |       500.00 |
|            4 |       300.00 |
+--------------+--------------+
2 rows in set (0.00 sec)

mysql> select a.id 订单编号,a.price 订单金额 from t_order a order by
a.price desc limit 2,2;
+--------------+--------------+
| 订单编号  | 订单金额  |
+--------------+--------------+
|            6 |       200.50 |
|            2 |       100.68 |
+--------------+--------------+
2 rows in set (0.00 sec)
```

```
mysql> select a.id订单编号,a.price订单金额 from t_order a order by
a.price desc limit 4,2;
+--------------+--------------+
| 订单编号   | 订单金额   |
+--------------+--------------+
|            1 |        88.95 |
|            5 |        20.88 |
+--------------+--------------+
2 rows in set (0.00 sec)
```

避免踩坑

limit中不能使用表达式

```
mysql> select * from t_order where limit 1,4+1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'limit 1,4+1' at line 1
mysql> select * from t_order where limit 1+0;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'limit 1+0' at line 1
mysql>
```

结论：**limit后面只能够跟明确的数字。**

limit后面的2个数字不能为负数

```
mysql> select * from t_order where limit -1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'limit -1' at line 1
mysql> select * from t_order where limit 0,-1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'limit 0,-1' at line 1
mysql> select * from t_order where limit -1,-1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
```

```
manual that corresponds to your MySQL server version for the right
syntax to use near 'limit -1,-1' at line 1
```

排序分页存在的坑

准备数据:

```
mysql> insert into test1 (b) values (1),(2),(3),(4),(2),(2),(2),(2);
Query OK, 8 rows affected (0.01 sec)
Records: 8  Duplicates: 0  Warnings: 0

mysql> select * from test1;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |
| 8 | 2 |
+---+---+
8 rows in set (0.00 sec)

mysql> select * from test1 order by b asc;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |
| 8 | 2 |
| 3 | 3 |
| 4 | 4 |
```

```
+---+---+
8 rows in set (0.00 sec)
```

下面我们按照b升序，每页2条数据，来获取数据。

下面的sql依次为第1页、第2页、第3页、第4页、第5页的数据，如下：

```
mysql> select * from test1 order by b asc limit 0,2;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 2 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc limit 2,2;
+---+---+
| a | b |
+---+---+
| 8 | 2 |
| 6 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc limit 4,2;
+---+---+
| a | b |
+---+---+
| 6 | 2 |
| 7 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc limit 6,2;
+---+---+
| a | b |
+---+---+
| 3 | 3 |
| 4 | 4 |
+---+---+
```

```
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc limit 7,2;
+---+---+
| a | b |
+---+---+
| 4 | 4 |
+---+---+
1 row in set (0.00 sec)
```

上面有2个问题：

问题1：看一下第2个sql和第3个sql，分别是第2页和第3页的数据，结果出现了相同的数据，是不是懵逼了。

问题2：整个表只有8条记录，怎么会出现第5页的数据呢，又懵逼了。

我们来分析一下上面的原因：**主要是b字段存在相同的值，当排序过程中存在相同的值时，没有其他排序规则时，mysql懵逼了，不知道怎么排序了。**

就像我们上学站队一样，按照身高排序，那身高一样的时候如何排序呢？身高一样的就乱排了。

建议：排序中存在相同的值时，需要再指定一个排序规则，通过这种排序规则不存在二义性，比如上面可以再加上a降序，如下：

```
mysql> select * from test1 order by b asc,a desc;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 8 | 2 |
| 7 | 2 |
| 6 | 2 |
| 5 | 2 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+---+---+
```

```
8 rows in set (0.00 sec)

mysql> select * from test1 order by b asc,a desc limit 0,2;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 8 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc,a desc limit 2,2;
+---+---+
| a | b |
+---+---+
| 7 | 2 |
| 6 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc,a desc limit 4,2;
+---+---+
| a | b |
+---+---+
| 5 | 2 |
| 2 | 2 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc,a desc limit 6,2;
+---+---+
| a | b |
+---+---+
| 3 | 3 |
| 4 | 4 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from test1 order by b asc,a desc limit 8,2;
Empty set (0.00 sec)
```

看上面的结果，分页数据都正常了，第5页也没有数据了。

总结

- order by ... [asc|desc]用于对查询结果排序，asc：升序，desc：降序，asc|desc可以省略，默认为asc
- limit用来限制查询结果返回的行数，有2个参数（offset，count），offset：表示跳过多少行，count：表示跳过offset行之后取count行
- limit中offset可以省略，默认值为0
- limit中offset 和 count都必须大于等于0
- limit中offset和count的值不能用表达式
- 分页排序时，排序不要有二义性，二义性情况下可能会导致分页结果乱序，可以在后面追加一个主键排序

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**

mysql系列大概有20多篇，喜欢的请关注一下！提前祝大家中秋快乐！

第9篇：分组查询（group by、having）

这是Mysql系列第9篇。

环境：mysql5.7.25，cmd命令中进行演示。

本篇内容

1. 分组查询语法
2. 聚合函数
3. 单字段分组
4. 多字段分组
5. 分组前筛选数据
6. 分组后筛选数据
7. where和having的区别
8. 分组后排序
9. where & group by & having & order by & limit 一起协作
10. mysql分组中的坑
11. in多列查询的使用

分组查询

语法：

```
SELECT column, group_function,... FROM table
[WHERE condition]
GROUP BY group_by_expression
[HAVING group_condition];
```

说明：

group_function：聚合函数。

group*by*expression：分组表达式，多个之间用逗号隔开。

group_condition：分组之后对数据进行过滤。

分组中，select后面只能有两种类型的列：

1. 出现在group by后的列
2. 或者使用聚合函数的列

聚合函数

函数名称	作用
max	查询指定列的最大值
min	查询指定列的最小值
count	统计查询结果的行数

sum 求和，返回指定列的总和

avg 求平均值，返回指定列数据的平均值

分组时，可以使用使用上面的聚合函数。

准备数据

```
drop table if exists t_order;

-- 创建订单表
create table t_order(
  id int not null AUTO_INCREMENT COMMENT '订单id',
  user_id bigint not null comment '下单人id',
  user_name varchar(16) not null default '' comment '用户名',
  price decimal(10,2) not null default 0 comment '订单金额',
  the_year SMALLINT not null comment '订单创建年份',
  PRIMARY KEY (id)
) comment '订单表';

-- 插入数据
insert into t_order(user_id,user_name,price,the_year) values
  (1001,'路人甲Java',11.11,'2017'),
  (1001,'路人甲Java',22.22,'2018'),
  (1001,'路人甲Java',88.88,'2018'),
  (1002,'刘德华',33.33,'2018'),
  (1002,'刘德华',12.22,'2018'),
  (1002,'刘德华',16.66,'2018'),
  (1002,'刘德华',44.44,'2019'),
  (1003,'张学友',55.55,'2018'),
  (1003,'张学友',66.66,'2019');

mysql> select * from t_order;
+----+---------+---------------+-------+----------+
| id | user_id | user_name     | price | the_year |
+----+---------+---------------+-------+----------+
|  1 |    1001 | 路人甲Java    | 11.11 |     2017 |
|  2 |    1001 | 路人甲Java    | 22.22 |     2018 |
|  3 |    1001 | 路人甲Java    | 88.88 |     2018 |
|  4 |    1002 | 刘德华    | 33.33 |     2018 |
|  5 |    1002 | 刘德华    | 12.22 |     2018 |
|  6 |    1002 | 刘德华    | 16.66 |     2018 |
|  7 |    1002 | 刘德华    | 44.44 |     2019 |
```

```
|  8 |    1003 | 张学友    | 55.55 |     2018 |
|  9 |    1003 | 张学友    | 66.66 |     2019 |
+----+---------+---------------+-------+----------+
9 rows in set (0.00 sec)
```

单字段分组

需求：查询每个用户下单数量，输出：用户id、下单数量，如下：

```
mysql> SELECT
            user_id 用户id, COUNT(id) 下单数量
        FROM
            t_order
        GROUP BY user_id;
+----------+--------------+
| 用户id    | 下单数量  |
+----------+--------------+
|     1001 |            3 |
|     1002 |            4 |
|     1003 |            2 |
+----------+--------------+
3 rows in set (0.00 sec)
```

多字段分组

需求：查询每个用户每年下单数量，输出字段：用户id、年份、下单数量，如下：

```
mysql> SELECT
            user_id 用户id, the_year 年份, COUNT(id) 下单数量
        FROM
            t_order
        GROUP BY user_id , the_year;
+----------+--------+--------------+
| 用户id    | 年份 | 下单数量  |
+----------+--------+--------------+
|     1001 |   2017 |            1 |
|     1001 |   2018 |            2 |
```

```
|      1002 |    2018 |             3 |
|      1002 |    2019 |             1 |
|      1003 |    2018 |             1 |
|      1003 |    2019 |             1 |
+-----------+---------+---------------+
6 rows in set (0.00 sec)
```

分组前筛选数据

分组前对数据进行筛选，使用where关键字

需求：需要查询2018年每个用户下单数量，输出：用户id、下单数量，如下：

```
mysql> SELECT
            user_id 用户id, COUNT(id) 下单数量
        FROM
            t_order t
        WHERE
            t.the_year = 2018
        GROUP BY user_id;
+-----------+--------------+
| 用户id    | 下单数量   |
+-----------+--------------+
|      1001 |            2 |
|      1002 |            3 |
|      1003 |            1 |
+-----------+--------------+
3 rows in set (0.00 sec)
```

分组后筛选数据

分组后对数据筛选，使用having关键字

需求：查询2018年订单数量大于1的用户，输出：用户id，下单数量，如下：

方式1：

```
mysql> SELECT
          user_id 用户id, COUNT(id) 下单数量
        FROM
          t_order t
        WHERE
          t.the_year = 2018
        GROUP BY user_id
        HAVING count(id)>=2;
+----------+--------------+
| 用户id    | 下单数量   |
+----------+--------------+
|     1001 |            2 |
|     1002 |            3 |
+----------+--------------+
2 rows in set (0.00 sec)
```

方式2:

```
mysql> SELECT
          user_id 用户id, count(id) 下单数量
        FROM
          t_order t
        WHERE
          t.the_year = 2018
        GROUP BY user_id
        HAVING 下单数量>=2;
+----------+--------------+
| 用户id    | 下单数量   |
+----------+--------------+
|     1001 |            2 |
|     1002 |            3 |
+----------+--------------+
2 rows in set (0.00 sec)
```

where和having的区别

where是在分组（聚合）前对记录进行筛选，而having是在分组结束后的结果里筛选，最后返回整个sql的查询结果。

可以把having理解为两级查询，即含having的查询操作先获得不含having子句时的sql查询结果表，然后在这个结果表上使用having条件筛选出符合的记录，最后返回这些记录，因此，having后是可以跟聚合函数的，并且这个聚集函数不必与select后面的聚集函数相同。

分组后排序

需求：获取每个用户最大金额，然后按照最大金额倒序，输出：用户id，最大金额，如下：

```
mysql> SELECT
          user_id 用户id, max(price) 最大金额
        FROM
          t_order t
        GROUP BY user_id
        ORDER BY 最大金额 desc;
+----------+--------------+
| 用户id    | 最大金额  |
+----------+--------------+
|     1001 |        88.88 |
|     1003 |        66.66 |
|     1002 |        44.44 |
+----------+--------------+
3 rows in set (0.00 sec)
```

where & group by & having & order by & limit 一起协作

where、group by、having、order by、limit这些关键字一起使用时，先后顺序有明确的限制，语法如下：

```
select 列 from
表名
where [查询条件]
group by [分组表达式]
having [分组过滤条件]
```

```
order by [排序条件]
limit [offset,] count;
```

注意：

写法上面必须按照上面的顺序来写。

示例：

需求：查询出2018年，下单数量大于等于2的，按照下单数量降序排序，最后只输出第1条记录，显示：用户id，下单数量，如下：

```
mysql> SELECT
          user_id 用户id, COUNT(id) 下单数量
        FROM
          t_order t
        WHERE
          t.the_year = 2018
        GROUP BY user_id
        HAVING count(id)>=2
        ORDER BY 下单数量 DESC
        LIMIT 1;
+----------+--------------+
| 用户id   | 下单数量  |
+----------+--------------+
|     1002 |            3 |
+----------+--------------+
1 row in set (0.00 sec)
```

mysql分组中的坑

本文开头有介绍，分组中select后面的列只能有2种：

1. 出现在group by后面的列
2. 使用聚合函数的列

oracle、sqlserver、db2中也是按照这种规范来的。

文中使用的是5.7版本，默认是按照这种规范来的。

mysql早期的一些版本，没有上面这些要求，select后面可以跟任何合法的列。

示例

需求：获取每个用户下单的最大金额及下单的年份，输出：用户id，最大金额，年份，写法如下：

```
mysql> select
          user_id 用户id, max(price) 最大金额, the_year 年份
        FROM t_order t
        GROUP BY t.user_id;
ERROR 1055 (42000): Expression #3 of SELECT list is not in GROUP BY
clause and contains nonaggregated column 'javacode2018.t.the_year'
which is not functionally dependent on columns in GROUP BY clause;
this is incompatible with sql_mode=only_full_group_by
```

上面的sql报错了，原因因为the_year不符合上面说的2条规则（select后面的列必须出现在group by中或者使用聚合函数），而sql_mode限制了这种规则，我们看一下sql_mode的配置：

```
mysql> select @@sql_mode;
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
+
| @@sql_mode
|
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
+
|
ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ER
ROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
+
1 row in set (0.00 sec)
```

sql*mode*中包含了`*ONLY*FULL*GROUP*BY`，这个表示select后面的列必须符合上面的说的2点规范。

可以将ONLY_FULL_GROUP_BY去掉，select后面就可以加任意列了，我们来看一下效果。

修改mysql中的my.ini文件：

```
sql_mode=STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DI
VISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
```

重启mysql，再次运行，效果如下：

```
mysql> select
          user_id 用户id, max(price) 最大金额, the_year 年份
        FROM t_order t
        GROUP BY t.user_id;
+----------+--------------+--------+
| 用户id   | 最大金额  | 年份 |
+----------+--------------+--------+
|     1001 |        88.88 |   2017 |
|     1002 |        44.44 |   2018 |
|     1003 |        66.66 |   2018 |
+----------+--------------+--------+
3 rows in set (0.03 sec)
```

看一下上面的数据，第一条88.88的年份是2017年，我们再来看一下原始数据：

```
mysql> select * from t_order;
+----+---------+---------------+-------+----------+
| id | user_id | user_name     | price | the_year |
+----+---------+---------------+-------+----------+
|  1 |    1001 | 路人甲Java    | 11.11 |     2017 |
|  2 |    1001 | 路人甲Java    | 22.22 |     2018 |
|  3 |    1001 | 路人甲Java    | 88.88 |     2018 |
|  4 |    1002 | 刘德华    | 33.33 |     2018 |
|  5 |    1002 | 刘德华    | 12.22 |     2018 |
|  6 |    1002 | 刘德华    | 16.66 |     2018 |
|  7 |    1002 | 刘德华    | 44.44 |     2019 |
|  8 |    1003 | 张学友    | 55.55 |     2018 |
|  9 |    1003 | 张学友    | 66.66 |     2019 |
```

```
+----+---------+---------------+-------+----------+
9 rows in set (0.00 sec)
```

对比一下，user*id=1001*、*price=88.88*是第*3*条数据，即*the*year是2018年，但是上面的分组结果是2017年，结果和我们预期的不一致，此时mysql对这种未按照规范来的列，乱序了，mysql取的是第一条。

正确的写法，提供两种，如下：

```
mysql> SELECT
          user_id 用户id,
          price 最大金额,
          the_year 年份
        FROM
          t_order t1
        WHERE
          (t1.user_id , t1.price)
          IN
          (SELECT
             t.user_id, MAX(t.price)
           FROM
             t_order t
           GROUP BY t.user_id);
+----------+--------------+--------+
| 用户id    | 最大金额   | 年份 |
+----------+--------------+--------+
|     1001 |        88.88 |   2018 |
|     1002 |        44.44 |   2019 |
|     1003 |        66.66 |   2019 |
+----------+--------------+--------+
3 rows in set (0.00 sec)

mysql> SELECT
          user_id 用户id,
          price 最大金额,
          the_year 年份
        FROM
          t_order t1,(SELECT
                        t.user_id uid, MAX(t.price) pc
```

```
                    FROM
                      t_order t
                    GROUP BY t.user_id) t2
        WHERE
          t1.user_id = t2.uid
        AND  t1.price = t2.pc;
+----------+--------------+--------+
| 用户id    | 最大金额   | 年份 |
+----------+--------------+--------+
|     1001 |        88.88 |   2018 |
|     1002 |        44.44 |   2019 |
|     1003 |        66.66 |   2019 |
+----------+--------------+--------+
3 rows in set (0.00 sec)
```

上面第1种写法，比较少见，in中使用了多字段查询。

建议：在写分组查询的时候，最好按照标准的规范来写，select后面出现的列必须在group by中或者必须使用聚合函数。

总结

1. 在写分组查询的时候，最好按照标准的规范来写，**select后面出现的列必须在group by中或者必须使用聚合函数。**
2. select语法顺序：select、from、where、group by、having、order by、limit，顺序不能搞错了，否则报错。
3. **in多列查询的使用，下去可以试试**

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**

4. 第4篇：DDL常见操作
5. 第5篇：DML操作汇总（insert,update,delete）
6. 第6篇：select查询基础篇
7. 第7篇：玩转select条件查询，避免采坑
8. 第8篇：详解排序和分页(order by & limit)

mysql系列大概有20多篇，喜欢的请关注一下，欢迎大家加我微信itsoku或者留言交流mysql相关技术!

java高并发系列全集

1. 第1天:必须知道的几个概念
2. 第2天:并发级别
3. 第3天:有关并行的两个重要定律
4. 第4天:JMM相关的一些概念
5. 第5天:深入理解进程和线程
6. 第6天:线程的基本操作
7. 第7天:volatile与Java内存模型
8. 第8天:线程组
9. 第9天：用户线程和守护线程
10. 第10天:线程安全和synchronized关键字
11. 第11天:线程中断的几种方式
12. 第12天JUC:ReentrantLock重入锁
13. 第13天:JUC中的Condition对象

14. 第**14**天**:JUC**中的**LockSupport**工具类，必备技能
15. 第**15**天：**JUC**中的**Semaphore**（信号量）
16. 第**16**天：**JUC**中等待多线程完成的工具类**CountDownLatch**，必备技能
17. 第**17**天：**JUC**中的循环栅栏**CyclicBarrier**的**6**种使用场景
18. 第**18**天：**JAVA**线程池，这一篇就够了
19. 第**19**天：**JUC**中的**Executor**框架详解**1**
20. 第**20**天：**JUC**中的**Executor**框架详解**2**
21. 第**21**天：**java**中的**CAS**，你需要知道的东西
22. 第**22**天：**JUC**底层工具类**Unsafe**，高手必须要了解
23. 第**23**天：**JUC**中原子类，一篇就够了
24. 第**24**天：**ThreadLocal**、**InheritableThreadLocal**（通俗易懂）
25. 第**25**天：掌握**JUC**中的阻塞队列
26. 第**26**篇：学会使用**JUC**中常见的集合，常看看!
27. 第**27**天：实战篇，接口性能提升几倍原来这么简单
28. 第**28**天：实战篇，微服务日志的伤痛，一并帮你解决掉
29. 第**29**天：高并发中常见的限流方式
30. 第**30**天：**JUC**中工具类**CompletableFuture**，必备技能
31. 第**31**天：获取线程执行结果，这**6**种方法你都知道?
32. 第**32**天：高并发中计数器的实现方式有哪些?
33. 第**33**篇：怎么演示公平锁和非公平锁**?**
34. 第**34**篇：**google**提供的一些好用的并发工具类

第10篇：mysql常用函数汇总

这是Mysql系列第10篇。

环境：mysql5.7.25，cmd命令中进行演示。

MySQL 数值型函数

函数名称	作用
abs	求绝对值
sqrt	求二次方根
mod	求余数

两个函数功能相反，都是返回不小于参数的最小整数，即向上取整ceil和ceiling

向下取整，返回值转化为一个BIGINT

floor

rand 生成一个0～1之间的随机数，传入整数参数是，用来产生重复序列

round	对所传参数进行四舍五入
sign	返回参数的符号

两个函数的功能相同，都是对传参数的次方的结果值求正弦值

pow和power

sin

求反正弦值，与函数SIN互为反函数 求余弦值 acos asin

求反余弦值，与函数COS互为反函数 求正切值 acos tan

atan 求反正切值，与函数TAN互为反函数

cot 求余切值

abs:求绝对值

函数 ABS(x) 返回 x 的绝对值。正数的绝对值是其本身，负数的绝对值为其相反数，0 的绝对值是 0。

```
mysql> select abs(5),abs(-2.4),abs(-24),abs(0);
+--------+-----------+----------+--------+
| abs(5) | abs(-2.4) | abs(-24) | abs(0) |
+--------+-----------+----------+--------+
|      5 |       2.4 |       24 |      0 |
+--------+-----------+----------+--------+
1 row in set (0.00 sec)
```

sqrt:求二次方跟（开方）

函数 SQRT(x) 返回非负数 x 的二次方根。负数没有平方根，返回结果为 NULL。

```
mysql> select sqrt(25),sqrt(120),sqrt(-9);
+----------+--------------------+----------+
| sqrt(25) | sqrt(120)          | sqrt(-9) |
+----------+--------------------+----------+
|        5 | 10.954451150103322 |     NULL |
+----------+--------------------+----------+
1 row in set (0.00 sec)
```

mod:求余数

函数 MOD(x,y) 返回 x 被 y 除后的余数，MOD() 对于带有小数部分的数值也起作用，它返回除法运算后的余数。

```
mysql> select mod(63,8),mod(120,10),mod(15.5,3);
+-----------+-------------+-------------+
| mod(63,8) | mod(120,10) | mod(15.5,3) |
+-----------+-------------+-------------+
|         7 |           0 |         0.5 |
+-----------+-------------+-------------+
1 row in set (0.00 sec)
```

ceil和ceiling:向上取整

函数 CEIL(x) 和 CEILING(x) 的意义相同，返回不小于 x 的最小整数值，返回值转化为一个 BIGINT。

```
mysql> select ceil(-2.5),ceiling(2.5);
+------------+--------------+
| ceil(-2.5) | ceiling(2.5) |
+------------+--------------+
|         -2 |            3 |
+------------+--------------+
1 row in set (0.00 sec)
```

floor:向下取整

floor(x) 函数返回小于 x 的最大整数值。

```
mysql> select floor(5),floor(5.66),floor(-4),floor(-4.66);
+----------+-------------+-----------+--------------+
| floor(5) | floor(5.66) | floor(-4) | floor(-4.66) |
+----------+-------------+-----------+--------------+
|        5 |           5 |        -4 |           -5 |
+----------+-------------+-----------+--------------+
1 row in set (0.00 sec)
```

rand:生成一个随机数

生成一个0~1之间的随机数，传入整数参数是，用来产生重复序列

```
mysql> select rand(), rand(), rand();
+--------------------+--------------------+--------------------+
| rand()             | rand()             | rand()             |
+--------------------+--------------------+--------------------+
| 0.5224735778965741 | 0.3678060549942833 | 0.2716095720153391 |
+--------------------+--------------------+--------------------+
1 row in set (0.00 sec)

mysql> select rand(1),rand(2),rand(1);
+---------------------+--------------------+---------------------+
| rand(1)             | rand(2)            | rand(1)             |
+---------------------+--------------------+---------------------+
| 0.40540353712197724 | 0.6555866465490187 | 0.40540353712197724 |
+---------------------+--------------------+---------------------+
1 row in set (0.00 sec)

mysql> select rand(1),rand(2),rand(1);
+---------------------+--------------------+---------------------+
| rand(1)             | rand(2)            | rand(1)             |
+---------------------+--------------------+---------------------+
| 0.40540353712197724 | 0.6555866465490187 | 0.40540353712197724 |
+---------------------+--------------------+---------------------+
1 row in set (0.00 sec)
```

round:四舍五入函数

返回最接近于参数 x 的整数；ROUND(x,y) 函数对参数x进行四舍五入的操作，返回值保留小数点后面指定的y位。

```
mysql> select round(-6.6),round(-8.44),round(3.44);
+-------------+--------------+-------------+
| round(-6.6) | round(-8.44) | round(3.44) |
+-------------+--------------+-------------+
|          -7 |           -8 |           3 |
+-------------+--------------+-------------+
1 row in set (0.00 sec)

mysql> select
round(-6.66,1),round(3.33,3),round(88.66,-1),round(88.46,-2);
+----------------+---------------+-----------------+-----------------+
| round(-6.66,1) | round(3.33,3) | round(88.66,-1) | round(88.46,-2) |
+----------------+---------------+-----------------+-----------------+
|           -6.7 |         3.330 |              90 |             100 |
+----------------+---------------+-----------------+-----------------+
1 row in set (0.00 sec)
```

sign:返回参数的符号

返回参数的符号，x 的值为负、零和正时返回结果依次为 -1、0 和 1。

```
mysql> select sign(-6),sign(0),sign(34);
+----------+---------+----------+
| sign(-6) | sign(0) | sign(34) |
+----------+---------+----------+
|       -1 |       0 |        1 |
+----------+---------+----------+
1 row in set (0.00 sec)
```

pow 和 power:次方函数

POW(x,y) 函数和 POWER(x,y) 函数用于计算 x 的 y 次方。

```
mysql> select pow(5,-2),pow(10,3),pow(100,0),power(4,3),power(6,-3);
+-----------+-----------+------------+------------
+----------------------+
| pow(5,-2) | pow(10,3) | pow(100,0) | power(4,3) | power(6,-3)
|
+-----------+-----------+------------+------------
+----------------------+
|      0.04 |      1000 |          1 |         64 |
0.004629629629629629 |
```

```
+-----------+-----------+------------+------------
+----------------------+
1 row in set (0.00 sec)
```

sin:正弦函数

SIN(x) 返回 x 的正弦值，其中 x 为弧度值。

```
mysql> select sin(1),sin(0.5*pi()),pi();
+--------------------+---------------+----------+
| sin(1)             | sin(0.5*pi()) | pi()     |
+--------------------+---------------+----------+
| 0.8414709848078965 |             1 | 3.141593 |
+--------------------+---------------+----------+
1 row in set (0.00 sec)
```

注：PI() 函数返回圆周率（3.141593）

其他几个三角函数在此就不说了，有兴趣的可以自己去练习一下。

MySQL 字符串函数

函数名称用作 length 计算字符串长度函数，返回字符串的字节长度

concat 合并字符串函数，返回结果为连接参数产生的字符串，参数可以使一个或多

把换字符串函数将字符串中的字母转换为小写insertlower

将字符串中的字母转换为大写
upper

left 从左侧字符串截取若干个字符，返回字符串左边的若干个字符

right 从右侧字截取若干，返回字符串右边的若干个字符

删除字符串左右两侧的空格 字符串替换函数，返回替换后的新字符串 trim replace

截取字符串，返回从指定位置开始的指定长度的字符串substring

reverse 字符串反转（逆序）函数，返回与原始字符串顺序相反的字符串

length:返回字符串直接长度

返回值为字符串的字节长度，使用 uft8（UNICODE 的一种变长字符编码，又称万国码）编码字符集时，一个汉字是 3 个字节，一个数字或字母是一个字节。

```
mysql> select length('javacode2018'),length('路人甲Java'),length('路
人');
+-----------------------+-----------------------+-----------------
+
| length('javacode2018') | length('路人甲Java')    | length('路人')   |
+-----------------------+-----------------------+-----------------
+
|                    12 |                    13 |               6
|
+-----------------------+-----------------------+-----------------
+
1 row in set (0.00 sec)
```

concat:合并字符串

CONCAT(sl，s2，...) 函数返回结果为连接参数产生的字符串，或许有一个或多个参数。

若有任何一个参数为 NULL，则返回值为 NULL。若所有参数均为非二进制字符串，则结果为非二进制字符串。若自变量中含有任一二进制字符串，则结果为一个二进制字符串。

```
mysql> select concat('路人甲','java'),concat('路人甲',null,'java');
+---------------------------+---------------------------------+
| concat('路人甲','java')    | concat('路人甲',null,'java')    |
+---------------------------+---------------------------------+
| 路人甲java                 | NULL                            |
+---------------------------+---------------------------------+
1 row in set (0.00 sec)
```

insert:替换字符串

INSERT(s1，x，len，s2) 返回字符串 s1，子字符串起始于 x 位置，并且用 len 个字符长的字符串代替 s2。

x的值从1开始，第一个字符的x=1，若 x 超过字符串长度，则返回值为原始字符串。

假如 len 的长度大于其他字符串的长度，则从位置 x 开始替换。

若任何一个参数为 NULL，则返回值为 NULL。

```
mysql> select
    ->   insert('路人甲Java', 2, 4, '**') AS col1,
    ->   insert('路人甲Java', -1, 4,'**') AS col2,
    ->   insert('路人甲Java', 3, 20,'**') AS col3;
+---------+---------------+----------+
| col1    | col2          | col3     |
+---------+---------------+----------+
| 路**va  | 路人甲Java    | 路人**   |
+---------+---------------+----------+
1 row in set (0.00 sec)
```

lower:将字母转换成小写

LOWER(str) 可以将字符串 str 中的字母字符全部转换成小写。

```
mysql> select lower('路人甲JAVA');
+-----------------------+
| lower('路人甲JAVA')    |
+-----------------------+
| 路人甲java             |
+-----------------------+
1 row in set (0.00 sec)
```

upper:将字母转换成大写

UPPER(str) 可以将字符串 str 中的字母字符全部转换成大写。

```
mysql> select upper('路人甲java');
+-----------------------+
| upper('路人甲java')    |
+-----------------------+
| 路人甲JAVA             |
+-----------------------+
1 row in set (0.00 sec)
```

left:从左侧截取字符串

LEFT(s，n) 函数返回字符串 s 最左边的 n 个字符，s=1表示第一个字符。

```
mysql> select left('路人甲JAVA',2),left('路人甲JAVA',10),left('路人甲
JAVA',-1);
+-----------------------+-------------------------
```

```
+--------------------------+
| left('路人甲JAVA',2)     | left('路人甲JAVA',10)     | left('路人甲
JAVA',-1)     |
+--------------------------+-------------------------
+--------------------------+
| 路人         | 路人甲JAVA                |                              |
+--------------------------+-------------------------
+--------------------------+
1 row in set (0.00 sec)
```

right:从右侧截取字符串

RIGHT(s，n) 函数返回字符串 s 最右边的 n 个字符。

```
mysql> select right('路人甲JAVA',1),right('路人甲JAVA',10),right('路人甲
JAVA',-1);
+--------------------------+-------------------------
+--------------------------+
| right('路人甲JAVA',1)     | right('路人甲JAVA',10)     | right('路人甲
JAVA',-1)     |
+--------------------------+-------------------------
+--------------------------+
| A                         | 路人甲JAVA                |
|
+--------------------------+-------------------------
+--------------------------+
1 row in set (0.00 sec)
```

trim:删除字符串两侧空格

TRIM(s) 删除字符串 s 两侧的空格。

```
mysql> select '[    路人甲Java    ]',concat('[',trim('    路人甲Java
'),']');
+----------------------+-----------------------------------------------
+
| [    路人甲Java    ]     | concat('[',trim('    路人甲Java    '),']')     |
+----------------------+-----------------------------------------------
+
| [    路人甲Java    ]     | [路人甲Java]                                  |
+----------------------+-----------------------------------------------
```

```
+
1 row in set (0.00 sec)
```

replace:字符串替换

REPLACE(s，s1，s2) 使用字符串 s2 替换字符串 s 中所有的字符串 s1。

substr 和 substring:截取字符串

substr(str,pos)

substr(str from pos)

substr(str,pos,len)

substr(str from pos for len)

substr()是substring()的同义词。

没有len参数的形式是字符串str从位置pos开始返回一个子字符串。

带有len参数的形式是字符串str从位置pos开始返回长度为len的子字符串。

使用FROM的形式是标准的SQL语法。

也可以对pos使用负值，在这种情况下，子字符串的开头是字符串末尾的pos字符，而不是开头。 在这个函数的任何形式中pos可以使用负值。

对于所有形式的substring()，从中提取子串的字符串中第一个字符的位置被认为是1。

```
/** 第三个字符之后的子字符串：inese **/
SELECT substring('chinese', 3);
/** 倒数第三个字符之后的子字符串：ese **/
SELECT substring('chinese', -3);
/** 第三个字符之后的两个字符：in **/
SELECT substring('chinese', 3, 2);
/** 倒数第三个字符之后的两个字符：es **/
SELECT substring('chinese', -3, 2);
/** 第三个字符之后的子字符串：inese **/
SELECT substring('chinese' FROM 3);
/** 倒数第三个字符之后的子字符串：ese **/
SELECT substring('chinese' FROM -3);
/** 第三个字符之后的两个字符：in **/
SELECT substring('chinese' FROM 3 FOR 2);
```

```
/** 倒数第三个字符之后的两个字符：es **/
SELECT substring('chinese' FROM -3 FOR 2);
```

reverse:反转字符串

REVERSE(s) 可以将字符串 s 反转，返回的字符串的顺序和 s 字符串的顺序相反。

```
mysql> select reverse('路人甲Java');
+--------------------------+
| reverse('路人甲Java')    |
+--------------------------+
| avaJ甲人路      |
+--------------------------+
1 row in set (0.00 sec)
```

MySQL 日期和时间函数

作用：两个函数作用相同，返回当前系统的日期值

函数名称：current_date和curdate

两个函数作用相同，返回当前系统的时间值current_time和current_time

两个函数作用相同，返回当前系统的日期和时间值
now和sysdate

获取UNIX时间戳函数，返回一个以UNIX时间戳为基础的无符号整数

unix_timestamp

from_unixtime 将UNIX时间戳转换为时间格式，与UNIX_TIMESTAMP互为反函数

month 获取指定日期中的月份

monthname 获取指定日期中的月份英文名称

获取指定日期对应的星期几的英文名称

dayname

dayofweek 若取指定日期是一周中是第几天，返回值范围是1～7，1=周日

week 获取指定日期是一年中的第几周，返回值的范围是否为0～52或1～53

dayofyear 获取指定日期是一年中的第几天，返回值范围是1~366

获取指定日期是一个月中是第几天，返回值范围是1~31

dayofmonth

获取年份，返回值范围是1970～2069

time_to_sec 将时间参数转换为秒数

year

将秒数转换为时间，与TIME*TO*SEC互为反函数

sec*to*time

两个函数功能相同，都是向日期添加指定的时间间隔

date_add和adddate

两个函数功能相反，都是向日期减去指定的时间间隔

date_sub和subdate

时间加法运算，在原始时间上添加指定的时间

addtime

subtime 时间减法运算，在原始时间上减去指定的时间

datedif 获取两个日期之间间隔，返回参数1减去参数2的值

date_format，格式化指定的日期，根据参数返回指定格式的值

weekday 获取指定日期在一周内的对应的工作日索引

curtime 和 current_time:获取系统当前时间

CURTIME() 和 CURRENT_TIME() 函数的作用相同，将当前时间以“HH：MM：SS”或“HHMMSS”格式返回，具体格式根据函数用在字符串或数字语境中而定，返回time类型。

```
mysql> select curtime(),current_time(),current_time()+1;
+-----------+---------------+------------------+
| curtime() | current_time() | current_time()+1 |
+-----------+---------------+------------------+
| 16:11:25  | 16:11:25       |           161126 |
+-----------+---------------+------------------+
1 row in set (0.00 sec)
```

now 和 sysdate:获取当前时间日期

NOW() 和 SYSDATE() 函数的作用相同，都是返回当前日期和时间值，格式为“YYYY-MM-DD HH：MM：SS”或“YYYYMMDDHHMMSS”，具体格式根据函数用在字符串或数字语境中而定，返回datetime类型。

```
mysql> select now(),sysdate();
+---------------------+---------------------+
| now()               | sysdate()           |
+---------------------+---------------------+
| 2019-09-17 16:13:28 | 2019-09-17 16:13:28 |
+---------------------+---------------------+
1 row in set (0.00 sec)
```

unix_timestamp:获取UNIX时间戳

UNIX_TIMESTAMP(date) 若无参数调用，返回一个无符号整数类型的 UNIX 时间戳（'1970-01-01 00:00:00'GMT之后的秒数）。

```
mysql> select
unix_timestamp(),unix_timestamp(now()),now(),unix_timestamp('2019-09-1
7 12:00:00');
+------------------+----------------------+---------------------
+--------------------------------------+
| unix_timestamp() | unix_timestamp(now()) | now()               |
unix_timestamp('2019-09-17 12:00:00') |
+------------------+----------------------+---------------------
+--------------------------------------+
```

```
|       1568710893 |             1568710893 | 2019-09-17 17:01:33 |
1568692800 |
+------------------+------------------------+---------------------
+-------------------------------------+
1 row in set (0.00 sec)
```

from_unixtime:时间戳转日期

FROM*UNIXTIME(unix*timestamp[,format]) 函数把 UNIX 时间戳转换为普通格式的日期时间值，与 `UNIX_TIMESTAMP ()` 函数互为反函数。

有2个参数：

unix_timestamp：时间戳（秒）

format：要转化的格式 比如“”%Y-%m-%d“” 这样格式化之后的时间就是2017-11-30

可以有的形式：

栓
式

%M 月名字(January~December)

%W 星期名字(Sunday~Saturday)

%D 表英语前缀的月份的日期(1st,2nd,3rd,等等)

%Y,数字,4位年

%y，年，数字，2位
%a 缩写的星期名字（Sun~Sat）

%d 月份中的天数，数字(00~31)

%e 月份中的天数，数字(0~31)

%m, 数字(01~12)

%c, 数字(1~12)

%b 缩写的月份名字(Jan ~ Dec)

%j 一年中的天数(001 ~ 366)

%H 时(00~23)

%k 时(0~23)

%h 时(01~12)

%I 时(01~12)

%i 的大写)

%l 小时(1~12)

L的小写)

%i 分钟,数字(00~59)

%时间,12小时(hh:mm:ss[AP]M)

%T 时间，24小时(hh:mm:ss)

%S 秒(00~59)

%s 秒(00~59)

%AM或PM

%W一个星期中的天数英文名称（Sunday~Saturday）

%—w 个星期中的天数（0=Sunday~6=Saturday）

星期(0~52),这里星期天是星期的第一天%U

%u 星期(0~52),这里星期一是星期的第一天

%% 输出%

```
mysql> select from_unixtime(1568710866),from_unixtime(1568710866,'%Y-
%m-%d %H:%h:%s');
+---------------------------
+-------------------------------------------------+
| from_unixtime(1568710866) | from_unixtime(1568710866,'%Y-%m-%d %H:
%h:%s') |
+---------------------------
+-------------------------------------------------+
| 2019-09-17 17:01:06       | 2019-09-17 17:05:06
|
+---------------------------
```

```
+-----------------------------------------------+
1 row in set (0.00 sec)
```

month:获取指定日期的月份

MONTH(date) 函数返回指定 date 对应的月份，范围为 1～12。

```
mysql> select month('2017-12-15'),month(now());
+---------------------+--------------+
| month('2017-12-15') | month(now()) |
+---------------------+--------------+
|                  12 |            9 |
+---------------------+--------------+
1 row in set (0.00 sec)
```

monthname:获取指定日期月份的英文名称

MONTHNAME(date) 函数返回日期 date 对应月份的英文全名。

```
mysql> select monthname('2017-12-15'),monthname(now());
+-------------------------+-----------------+
| monthname('2017-12-15') | monthname(now()) |
+-------------------------+-----------------+
| December                | September       |
+-------------------------+-----------------+
1 row in set (0.00 sec)
```

dayname:获取指定日期的星期名称

DAYNAME(date) 函数返回 date 对应的工作日英文名称，例如 Sunday、Monday 等。

```
mysql> select now(),dayname(now());
+---------------------+----------------+
| now()               | dayname(now()) |
+---------------------+----------------+
| 2019-09-17 17:13:08 | Tuesday        |
+---------------------+----------------+
1 row in set (0.00 sec)
```

dayofweek:获取日期对应的周索引

DAYOFWEEK(d) 函数返回 d 对应的一周中的索引（位置）。1 表示周日，2 表示周一，……，7 表示周六。这些索引值对应于ODBC标准。

```
mysql> select now(),dayofweek(now());
+---------------------+------------------+
| now()               | dayofweek(now()) |
+---------------------+------------------+
| 2019-09-17 17:14:21 |                3 |
+---------------------+------------------+
1 row in set (0.00 sec)
```

week:获取指定日期是一年中的第几周

WEEK(date[,mode]) 函数计算日期 date 是一年中的第几周。WEEK(date,mode) 函数允许指定星期是否起始于周日或周一，以及返回值的范围是否为 0～52 或 1～53。

WEEK函数接受两个参数：

- date是要获取周数的日期。
- mode是一个可选参数，用于确定周数计算的逻辑。它允许您指定本周是从星期一还是星期日开始，返回的周数应在0到52之间或0到53之间。

  如果忽略mode参数，默认情况下WEEK函数将使用default_week_format系统变量的值。

  要获取default_week_format变量的当前值，请使用SHOW VARIABLES语句如下：

```
mysql> SHOW VARIABLES LIKE 'default_week_format';
+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| default_week_format | 0     |
+---------------------+-------+
1 row in set, 1 warning (0.00 sec)
```

在我们的服务器中，default_week_format的默认值为0，下表格说明了mode参数如何影响WEEK函数：

0	星期五	0	0	5	3
1	星期一	0	0	5	3
2	星期三	1	0	5	3
3	星期一	1	0	5	3
4	星期五	0	0	5	3
5	星期一	0	0	5	3
6	星期五	1	0	5	3

7 星1
期
一5
3

上表中“今年有4天以上”表示：

- 如果星期包含1月1日，并且在新的一年中有4天或更多天，那么这周是第1周。
- 否则，这一周的数字是前一年的最后一周，下周是第1周。

```
mysql> select now(),week(now());
+---------------------+-------------+
| now()               | week(now()) |
+---------------------+-------------+
| 2019-09-17 17:20:28 |          37 |
+---------------------+-------------+
1 row in set (0.00 sec)
```

dayofyear:获取指定日期在一年中的位置

DAYOFYEAR(d) 函数返回 d 是一年中的第几天，范围为 1 ~ 366。

```
mysql> select now(),dayofyear(now()),dayofyear('2019-01-01');
+---------------------+------------------+-------------------------+
| now()               | dayofyear(now()) | dayofyear('2019-01-01') |
+---------------------+------------------+-------------------------+
| 2019-09-17 17:22:00 |              260 |                       1 |
+---------------------+------------------+-------------------------+
1 row in set (0.00 sec)
```

dayofmonth:获取指定日期在一个月的位置

DAYOFMONTH(d) 函数返回 d 是一个月中的第几天，范围为 1 ~ 31。

```
mysql> select now(),dayofmonth(now()),dayofmonth('2019-01-01');
+---------------------+-------------------+--------------------------+
| now()               | dayofmonth(now()) | dayofmonth('2019-01-01') |
+---------------------+-------------------+--------------------------+
| 2019-09-17 17:23:09 |                17 |                        1 |
+---------------------+-------------------+--------------------------+
1 row in set (0.00 sec)
```

year:获取年份

YEAR() 函数可以从指定日期值中来获取年份值。

```
mysql> select now(),year(now()),year('2019-01-02');
+---------------------+-------------+--------------------+
| now()               | year(now()) | year('2019-01-02') |
+---------------------+-------------+--------------------+
| 2019-09-17 17:28:10 |        2019 |               2019 |
+---------------------+-------------+--------------------+
1 row in set (0.00 sec)
```

time*to*sec:将时间转换为秒值

TIME*TO*SEC(time) 函数返回将参数 time 转换为秒数的时间值，转换公式为“小时 ×3600+ 分钟 ×60+ 秒”。

```
mysql> select time_to_sec('15:15:15'),now(),time_to_sec(now());
+-------------------------+---------------------+--------------------+
| time_to_sec('15:15:15') | now()               | time_to_sec(now()) |
+-------------------------+---------------------+--------------------+
|                   54915 | 2019-09-17 17:30:44 |              63044 |
+-------------------------+---------------------+--------------------+
1 row in set (0.00 sec)
```

sec*to*time:将秒值转换为时间格式

SEC*TO*TIME(seconds) 函数返回将参数 seconds 转换为小时、分钟和秒数的时间值。

```
mysql> select sec_to_time(100),sec_to_time(10000);
+------------------+--------------------+
| sec_to_time(100) | sec_to_time(10000) |
+------------------+--------------------+
| 00:01:40         | 02:46:40           |
+------------------+--------------------+
1 row in set (0.00 sec)
```

date_add和adddate:向日期添加指定时间间隔

DATE_ADD(date,INTERVAL expr type)

date：参数是合法的日期表达式。*expr* 参数是您希望添加的时间间隔。

type：参数可以是下列值

Type值

MICROSECOND

SECOND

MINUTE

HOUR

DAY

WEEK

MONTH

QUARTER

YEAR

SECOND_MICROSECOND

MINUTE_MICROSECOND

MINUTE_SECOND

HOUR_MICROSECOND

HOUR_SECOND

HOUR_MINUTE

DAY_MICROSECOND

DAY_SECOND

DAY_MINUTE

DAY_HOUR

YEAR_MONTH

```
mysql> select date_add('2019-01-01',INTERVAL 10
day),adddate('2019-01-01 16:00:00',interval 100 SECOND);
+-----------------------------------------
+-----------------------------------------------------+
| date_add('2019-01-01',INTERVAL 10 day) | adddate('2019-01-01
16:00:00',interval 100 SECOND) |
+-----------------------------------------
+-----------------------------------------------------+
| 2019-01-11                              | 2019-01-01 16:01:40
|
+-----------------------------------------
+-----------------------------------------------------+
1 row in set (0.00 sec)

mysql> select date_add('2019-01-01',INTERVAL -10
day),adddate('2019-01-01 16:00:00',interval -100 SECOND);
+-----------------------------------------
+-----------------------------------------------------+
| date_add('2019-01-01',INTERVAL -10 day) | adddate('2019-01-01
16:00:00',interval -100 SECOND) |
+-----------------------------------------
+-----------------------------------------------------+
| 2018-12-22                               | 2019-01-01 15:58:20
|
+-----------------------------------------
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

date_sub和subdate:日期减法运算

DATE_SUB(date,INTERVAL expr type)

date：参数是合法的日期表达式。*expr* 参数是您希望添加的时间间隔。

type的类型和date_add中的type一样。

```
mysql> select date_sub('2019-01-01',INTERVAL 10
day),subdate('2019-01-01 16:00:00',interval 100 SECOND);
+-----------------------------------------
+-----------------------------------------------------+
| date_sub('2019-01-01',INTERVAL 10 day) | subdate('2019-01-01
16:00:00',interval 100 SECOND) |
+-----------------------------------------
```

```
+-----------------------------------------------------+
| 2018-12-22                                | 2019-01-01 15:58:20
|
+-------------------------------------
+-----------------------------------------------------+
1 row in set (0.00 sec)

mysql> select date_sub('2019-01-01',INTERVAL -10
day),subdate('2019-01-01 16:00:00',interval -100 SECOND);
+-------------------------------------
+-----------------------------------------------------+
| date_sub('2019-01-01',INTERVAL -10 day) | subdate('2019-01-01
16:00:00',interval -100 SECOND) |
+-------------------------------------
+-----------------------------------------------------+
| 2019-01-11                                 | 2019-01-01 16:01:40
|
+-------------------------------------
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

addtime:时间加法运算

ADDTIME(time,expr) 函数用于执行时间的加法运算。添加 expr 到 time 并返回结果。

其中：time 是一个时间或日期时间表达式，expr 是一个时间表达式。

```
mysql> select addtime('2019-09-18 23:59:59','0:1:1'),
addtime('10:30:59','5:10:37');
+-------------------------------------
+------------------------------+
| addtime('2019-09-18 23:59:59','0:1:1') |
addtime('10:30:59','5:10:37') |
+-------------------------------------
+------------------------------+
| 2019-09-19 00:01:00                    | 15:41:36
|
+-------------------------------------
+------------------------------+
1 row in set (0.00 sec)
```

subtime:时间减法运算

SUBTIME(time,expr) 函数用于执行时间的减法运算。

函数返回 time。expr 表示的值和格式 time 相同。time 是一个时间或日期时间表达式， expr 是一个时间。

```
mysql> select subtime('2019-09-18
23:59:59','0:1:1'),subtime('10:30:59','5:12:37');
+---------------------------------------
+------------------------------+
| subtime('2019-09-18 23:59:59','0:1:1') |
subtime('10:30:59','5:12:37') |
+---------------------------------------
+------------------------------+
| 2019-09-18 23:58:58                    | 05:18:22
|
+---------------------------------------
+------------------------------+
1 row in set (0.00 sec)
```

datediff:获取两个日期的时间间隔

DATEDIFF(date1，date2) 返回起始时间 date1 和结束时间 date2 之间的天数。date1 和 date2 为日期或 date-and-time 表达式。计算时只用到这些值的日期部分。

```
mysql> select datediff('2017-11-30','2017-11-29') as col1,
datediff('2017-11-30','2017-12-15') as col2;
+------+------+
| col1 | col2 |
+------+------+
|    1 |  -15 |
+------+------+
1 row in set (0.00 sec)
```

date_format:格式化指定的日期

DATE_FORMAT(date，format) 函数是根据 format 指定的格式显示 date 值。

DATE_FORMAT() 函数接受两个参数：

date：是要格式化的有效日期值format：是由预定义的说明符组成的格式字符串，每个说明符前面都有一个百分比字符(%)。

format：格式和上面的函数from_unixtime中的format一样，可以参考上面的。

```
mysql> select date_format('2017-11-30','%Y%m%d') as col0,now() as
col1, date_format(now(),'%Y%m%d%H%i%s') as col2;
+----------+---------------------+----------------+
| col0     | col1                | col2           |
+----------+---------------------+----------------+
| 20171130 | 2019-09-17 17:56:12 | 20190917175612 |
+----------+---------------------+----------------+
1 row in set (0.00 sec)
```

weekday:获取指定日期在一周内的索引位置

WEEKDAY(date) 返回date的星期索引(0=星期一，1=星期二, ……6= 星期天)。

```
mysql> select now(),weekday(now());
+---------------------+----------------+
| now()               | weekday(now()) |
+---------------------+----------------+
| 2019-09-17 18:01:34 |              1 |
+---------------------+----------------+
1 row in set (0.00 sec)

mysql> select now(),dayofweek(now());
+---------------------+------------------+
| now()               | dayofweek(now()) |
+---------------------+------------------+
| 2019-09-17 18:01:34 |                3 |
+---------------------+------------------+
1 row in set (0.00 sec)
```

MySQL 聚合函数

函数名称	作用
max	查询指定列的最大值
min	查询指定列的最小值
count	统计查询结果的行数

sum	求和，返回指定列的总和
avg	求平均值，返回指定列数据的平均值

MySQL 流程控制函数

函数名称用作

if 判断，流程控制

ifnull 判断是否为空

case 搜索语句

if:判断

IF(expr,v1,v2)

当 expr 为真是返回 v1 的值，否则返回 v2

```
mysql> select if(1<2,1,0) c1,if(1>5,'√','×')
c2,if(strcmp('abc','ab'),'yes','no') c3;
+----+----+-----+
| c1 | c2 | c3  |
+----+----+-----+
|  1 | ×  | yes |
```

```
+----+----+-----+
1 row in set (0.00 sec)
```

ifnull:判断是否为空

IFNULL(v1,v2)：v1为空返回v2，否则返回v1。

```
mysql> select ifnull(null,'路人甲Java'),ifnull('非空','为空');
+------------------------------+--------------------------+
| ifnull(null,'路人甲Java')    | ifnull('非空','为空')    |
+------------------------------+--------------------------+
| 路人甲Java                   | 非空                     |
+------------------------------+--------------------------+
1 row in set (0.00 sec)
```

case:搜索语句,类似于java中的if..else if..else

类似于java中的if..else if..else

有2种写法

方式1:

```
CASE  <表达式>
   WHEN <值1> THEN <操作>
   WHEN <值2> THEN <操作>
   ...
   ELSE <操作>
END CASE;
```

方式2:

```
CASE
    WHEN <条件1> THEN <命令>
    WHEN <条件2> THEN <命令>
    ...
    ELSE commands
END CASE;
```

示例:

准备数据:

```
CREATE TABLE t_stu (
  id   INT AUTO_INCREMENT  COMMENT '编号',
  name VARCHAR(10) COMMENT '姓名',
  sex  TINYINT COMMENT '性别,0:未知,1:男,2:女',
  PRIMARY KEY (id)
) COMMENT '学生表';

insert into t_stu (name,sex) VALUES
  ('张学友',1),
  ('刘德华',1),
  ('郭富城',1),
  ('蔡依林',2),
  ('xxx',0);

mysql> select * from t_stu;
+----+-----------+------+
| id | name      | sex  |
+----+-----------+------+
|  1 | 张学友  |    1 |
|  2 | 刘德华  |    1 |
|  3 | 郭富城  |    1 |
|  4 | 蔡依林  |    2 |
|  5 | xxx       |    0 |
+----+-----------+------+
5 rows in set (0.00 sec)
```

需求：查询所有学生信息，输出：姓名，性别（男、女、未知），如下：

```
mysql> SELECT
       t.name姓名,
       (CASE t.sex
        WHEN 1
          THEN '男'
        WHEN 2
          THEN '女'
        ELSE '未知' END) 性别
    FROM t_stu t;
+-----------+--------+
```

```
| 姓名   | 性别 |
+-----------+--------+
| 张学友  | 男   |
| 刘德华  | 男   |
| 郭富城  | 男   |
| 蔡依林  | 女   |
| xxx        | 未知 |
+-----------+--------+
5 rows in set (0.00 sec)

mysql> SELECT
    t.name    姓名,
    (CASE
    WHEN t.sex = 1
    THEN '男'
    WHEN t.sex = 2
    THEN '女'
    ELSE '未知' END) 性别
    FROM t_stu t;
+-----------+--------+
| 姓名   | 性别 |
+-----------+--------+
| 张学友  | 男   |
| 刘德华  | 男   |
| 郭富城  | 男   |
| 蔡依林  | 女   |
| xxx        | 未知 |
+-----------+--------+
5 rows in set (0.00 sec)
```

其他函数

函数名称 作用 数据库版本号

version 当前的数据库

database 当前连接用户

user

password 返回字符串密码形式

md5 返回字符串的md5数据

```
mysql> SELECT version();
+------------+
| version()  |
+------------+
| 5.7.25-log |
+------------+
1 row in set (0.00 sec)

mysql> SELECT database();
+--------------+
| database()   |
+--------------+
| javacode2018 |
+--------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT user();
+----------------+
| user()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)

mysql> SELECT  password('123456');
+-------------------------------------------+
| password('123456')                        |
+-------------------------------------------+
| *6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9 |
+-------------------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT md5('123456');
+----------------------------------+
| md5('123456')                    |
+----------------------------------+
| e10adc3949ba59abbe56e057f20f883e |
+----------------------------------+
1 row in set (0.00 sec)
```

今天列的函数比较多，大家搜藏一下，慢慢消化，喜欢的帮忙转发一下，谢谢。

Mysql系列目录

1. 第1篇：**mysql**基础知识
2. 第2篇：详解**mysql**数据类型（重点）
3. 第3篇：管理员必备技能**(**必须掌握**)**
4. 第4篇：**DDL**常见操作
5. 第5篇：**DML**操作汇总（**insert,update,delete**）

6. 第6篇：**select**查询基础篇
7. 第7篇：玩转**select**条件查询，避免采坑
8. 第8篇：详解排序和分页**(order by & limit)**
9. 第9篇：分组查询详解（**group by & having**）

mysql系列大概有**20**多篇，喜欢的请关注一下，欢迎大家加我微信**itsoku**或者留言交流**mysql**相关技术**!**

第11篇：深入了解连接查询及原理

这是Mysql系列第11篇。

环境：mysql5.7.25，cmd命令中进行演示。

当我们查询的数据来源于多张表的时候，我们需要用到连接查询，连接查询使用率非常高，希望大家都务必掌握。

本文内容

1. 笛卡尔积
2. 内连接

3. 外连接
4. 左连接
5. 右连接
6. 表连接的原理
7. 使用java实现连接查询，加深理解

准备数据

2张表：

t_team：组表。

t*employee*：员工表，内部有个*team*id引用组表的id。

```
drop table if exists t_team;
create table t_team(
  id int not null AUTO_INCREMENT PRIMARY KEY comment '组id',
  team_name varchar(32) not null default '' comment '名称'
) comment '组表';

drop table if exists t_employee;
create table t_employee(
  id int not null AUTO_INCREMENT PRIMARY KEY comment '部门id',
  emp_name varchar(32) not null default '' comment '员工名称',
  team_id int not null default 0 comment '员工所在组id'
) comment '员工表表';

insert into t_team values (1,'架构组'),(2,'测试组'),(3,'java组'),(4,'前端
组');
insert into t_employee values (1,'路人甲Java',1),(2,'张三',2),(3,'李
四',3),(4,'王五',0),(5,'赵六',0);
```

t_team表4条记录，如下：

```
mysql> select * from t_team;
+----+-----------+
| id | team_name |
+----+-----------+
|  1 | 架构组  |
|  2 | 测试组  |
|  3 | java组  |
|  4 | 前端组  |
+----+-----------+
4 rows in set (0.00 sec)
```

t_employee表5条记录，如下：

```
mysql> select * from t_employee;
+----+---------------+---------+
| id | emp_name      | team_id |
+----+---------------+---------+
|  1 | 路人甲Java    |       1 |
|  2 | 张三          |       2 |
|  3 | 李四          |       3 |
|  4 | 王五          |       0 |
|  5 | 赵六          |       0 |
+----+---------------+---------+
5 rows in set (0.00 sec)
```

笛卡尔积

介绍连接查询之前，我们需要先了解一下笛卡尔积。

笛卡尔积简单点理解：有两个集合A和B，笛卡尔积表示A集合中的元素和B集合中的元素任意相互关联产生的所有可能的结果。

假如A中有m个元素，B中有n个元素，A、B笛卡尔积产生的结果有m*n个结果，相当于循环遍历两个集合中的元素，任意组合。

java伪代码表示如下：

```
for(Object eleA : A){
      for(Object eleB : B){
            System.out.print(eleA+","+eleB);
      }
}
```

过程：拿A集合中的第1行，去匹配集合B中所有的行，然后再拿集合A中的第2行，去匹配集合B中所有的行，最后结果数量为m*n。

sql中笛卡尔积语法

```
select 字段 from 表1,表2[,表N];
```

或者

```
select 字段 from 表1 join 表2 [join 表N];
```

示例：

```
mysql> select * from t_team,t_employee;
+----+-----------+----+---------------+---------+
| id | team_name | id | emp_name      | team_id |
+----+-----------+----+---------------+---------+
|  1 | 架构组    |  1 | 路人甲Java    |       1 |
|  2 | 测试组    |  1 | 路人甲Java    |       1 |
|  3 | java组    |  1 | 路人甲Java    |       1 |
|  4 | 前端组    |  1 | 路人甲Java    |       1 |
|  1 | 架构组    |  2 | 张三          |       2 |
|  2 | 测试组    |  2 | 张三          |       2 |
|  3 | java组    |  2 | 张三          |       2 |
|  4 | 前端组    |  2 | 张三          |       2 |
|  1 | 架构组    |  3 | 李四          |       3 |
|  2 | 测试组    |  3 | 李四          |       3 |
|  3 | java组    |  3 | 李四          |       3 |
|  4 | 前端组    |  3 | 李四          |       3 |
|  1 | 架构组    |  4 | 王五          |       0 |
|  2 | 测试组    |  4 | 王五          |       0 |
|  3 | java组    |  4 | 王五          |       0 |
|  4 | 前端组    |  4 | 王五          |       0 |
|  1 | 架构组    |  5 | 赵六          |       0 |
```

```
|  2 | 测试组  |  5 | 赵六    |        0 |
|  3 | java组  |  5 | 赵六    |        0 |
|  4 | 前端组  |  5 | 赵六    |        0 |
+----+-----------+----+---------------+----------+
20 rows in set (0.00 sec)
```

t*team*表*4*条记录，*t*employee表5条记录，笛卡尔积结果输出了20行记录。

内连接

语法:

```
select 字段 from 表1 inner join 表2 on 连接条件;
或
select 字段 from 表1 join 表2 on 连接条件;
或
select 字段 from 表1, 表2 [where 关联条件];
```

内连接相当于在笛卡尔积的基础上加上了连接的条件。

当没有连接条件的时候，内连接上升为笛卡尔积。

过程用java伪代码如下:

```
for(Object eleA : A){
      for(Object eleB : B){
            if(连接条件是否为true){
                  System.out.print(eleA+","+eleB);
            }
      }
}
```

示例1：有连接条件

查询员工及所属部门

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join
t_team t2 on t1.team_id = t2.id;
```

```
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
| 张三          | 测试组    |
| 李四          | java组    |
+---------------+-----------+
3 rows in set (0.00 sec)

mysql> select t1.emp_name,t2.team_name from t_employee t1 join t_team
t2 on t1.team_id = t2.id;
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
| 张三          | 测试组    |
| 李四          | java组    |
+---------------+-----------+
3 rows in set (0.00 sec)

mysql> select t1.emp_name,t2.team_name from t_employee t1, t_team t2
where t1.team_id = t2.id;
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
| 张三          | 测试组    |
| 李四          | java组    |
+---------------+-----------+
3 rows in set (0.00 sec)
```

上面相当于获取了2个表的交集，查询出了两个表都有的数据。

示例2：无连接条件

无条件内连接，上升为笛卡尔积，如下：

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join
t_team t2;
+---------------+-----------+
```

```
| emp_name       | team_name |
+---------------+-----------+
| 路人甲Java     | 架构组  |
| 路人甲Java     | 测试组  |
| 路人甲Java     | java组  |
| 路人甲Java     | 前端组  |
| 张三     | 架构组  |
| 张三     | 测试组  |
| 张三     | java组  |
| 张三     | 前端组  |
| 李四     | 架构组  |
| 李四     | 测试组  |
| 李四     | java组  |
| 李四     | 前端组  |
| 王五     | 架构组  |
| 王五     | 测试组  |
| 王五     | java组  |
| 王五     | 前端组  |
| 赵六     | 架构组  |
| 赵六     | 测试组  |
| 赵六     | java组  |
| 赵六     | 前端组  |
+---------------+-----------+
20 rows in set (0.00 sec)
```

示例3：组合条件进行查询

查询架构组的员工，3种写法

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join
t_team t2 on t1.team_id = t2.id and t2.team_name = '架构组';
+---------------+-----------+
| emp_name       | team_name |
+---------------+-----------+
| 路人甲Java     | 架构组  |
+---------------+-----------+
1 row in set (0.00 sec)
```

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join
t_team t2 on t1.team_id = t2.id where t2.team_name = '架构组';
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
+---------------+-----------+
1 row in set (0.00 sec)

mysql> select t1.emp_name,t2.team_name from t_employee t1, t_team t2
where t1.team_id = t2.id and t2.team_name = '架构组';
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
+---------------+-----------+
1 row in set (0.00 sec)
```

上面3中方式解说。

方式1：on中使用了组合条件。

方式2：在连接的结果之后再进行过滤，相当于先获取连接的结果，然后使用where中的条件再对连接结果进行过滤。

方式3：直接在where后面进行过滤。

总结

内连接建议使用第3种语法，简洁：

```
select 字段 from 表1, 表2 [where 关联条件];
```

外连接

外连接涉及到2个表，分为：主表和从表，要查询的信息主要来自于哪个表，谁就是主表。

外连接查询结果为主表中所有记录。如果从表中有和它匹配的，则显示匹配的值，这部分相当于内连接查询出来的结果；如果从表中没有和它匹配的，则显示null。

最终：外连接查询结果 = 内连接的结果 + 主表中有的而内连接结果中没有的记录。

外连接分为2种：

左外链接：使用left join关键字，left join左边的是主表。

右外连接：使用right join关键字，right join右边的是主表。

左连接

语法

```
select 列 from 主表 left join 从表 on 连接条件;
```

示例1:

查询所有员工信息，并显示员工所在组，如下：

```
mysql> SELECT
        t1.emp_name,
        t2.team_name
    FROM
        t_employee t1
    LEFT JOIN
        t_team t2
    ON
        t1.team_id = t2.id;
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
| 张三          | 测试组    |
| 李四          | java组    |
| 王五          | NULL      |
| 赵六          | NULL      |
```

```
+---------------+-----------+
5 rows in set (0.00 sec)
```

上面查询出了所有员工，员工team*id=0*的，*team*name为NULL。

示例2：

查询员工姓名、组名，返回组名不为空的记录，如下：

```
mysql> SELECT
        t1.emp_name,
        t2.team_name
    FROM
        t_employee t1
    LEFT JOIN
        t_team t2
    ON
        t1.team_id = t2.id
    WHERE
        t2.team_name IS NOT NULL;
+---------------+-----------+
| emp_name      | team_name |
+---------------+-----------+
| 路人甲Java    | 架构组    |
| 张三          | 测试组    |
| 李四          | java组    |
+---------------+-----------+
3 rows in set (0.00 sec)
```

上面先使用内连接获取连接结果，然后再使用where对连接结果进行过滤。

右连接

语法

```
select 列 from 从表 right join 主表 on 连接条件;
```

示例

我们使用右连接来实现上面左连接实现的功能，如下：

```
mysql> SELECT
        t2.team_name,
        t1.emp_name
    FROM
        t_team t2
    RIGHT JOIN
        t_employee t1
    ON
        t1.team_id = t2.id;
+-----------+---------------+
| team_name | emp_name      |
+-----------+---------------+
| 架构组  | 路人甲Java    |
| 测试组  | 张三    |
| java组  | 李四    |
| NULL      | 王五    |
| NULL      | 赵六    |
+-----------+---------------+
5 rows in set (0.00 sec)

mysql> SELECT
        t2.team_name,
        t1.emp_name
    FROM
        t_team t2
    RIGHT JOIN
        t_employee t1
    ON
        t1.team_id = t2.id
    WHERE
        t2.team_name IS NOT NULL;
+-----------+---------------+
| team_name | emp_name      |
+-----------+---------------+
| 架构组  | 路人甲Java    |
```

```
| 测试组  | 张三     |
| java组  | 李四     |
+-----------+---------------+
3 rows in set (0.00 sec)
```

理解表连接原理

准备数据

```
drop table if exists test1;
create table test1(
  a int
);
drop table if exists test2;
create table test2(
  b int
);
insert into test1 values (1),(2),(3);
insert into test2 values (3),(4),(5);
```

```
mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)

mysql> select * from test2;
+------+
| b    |
+------+
|    3 |
|    4 |
|    5 |
+------+
3 rows in set (0.00 sec)
```

我们来写几个连接，看看效果。

示例1：内连接

```
mysql> select * from test1 t1,test2 t2;
+------+------+
| a    | b    |
+------+------+
|    1 |    3 |
|    2 |    3 |
|    3 |    3 |
|    1 |    4 |
|    2 |    4 |
|    3 |    4 |
|    1 |    5 |
|    2 |    5 |
|    3 |    5 |
+------+------+
9 rows in set (0.00 sec)

mysql> select * from test1 t1,test2 t2 where t1.a = t2.b;
+------+------+
| a    | b    |
+------+------+
|    3 |    3 |
+------+------+
1 row in set (0.00 sec)
```

9条数据正常。

示例2：左连接

```
mysql> select * from test1 t1 left join test2 t2 on t1.a = t2.b;
+------+------+
| a    | b    |
+------+------+
|    3 |    3 |
|    1 | NULL |
|    2 | NULL |
+------+------+
3 rows in set (0.00 sec)
```

```
mysql> select * from test1 t1 left join test2 t2 on t1.a>10;
+------+------+
| a    | b    |
+------+------+
|    1 | NULL |
|    2 | NULL |
|    3 | NULL |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test1 t1 left join test2 t2 on 1=1;
+------+------+
| a    | b    |
+------+------+
|    1 |    3 |
|    2 |    3 |
|    3 |    3 |
|    1 |    4 |
|    2 |    4 |
|    3 |    4 |
|    1 |    5 |
|    2 |    5 |
|    3 |    5 |
+------+------+
9 rows in set (0.00 sec)
```

上面的左连接第一个好理解。

第2个sql连接条件`t1.a>10`，这个条件只关联了test1表，再看看结果，是否可以理解？不理解的继续向下看，我们用java代码来实现连接查询。

第3个sql中的连接条件1=1值为true，返回结果为笛卡尔积。

java代码实现连接查询

下面是一个简略版的实现

```
package com.itsoku.sql;
```

```java
import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.stream.Collectors;

public class Test1 {
    public static class Table1 {
        int a;

        public int getA() {
            return a;
        }

        public void setA(int a) {
            this.a = a;
        }

        public Table1(int a) {
            this.a = a;
        }

        @Override
        public String toString() {
            return "Table1{" +
                    "a=" + a +
                    '}';
        }

        public static Table1 build(int a) {
            return new Table1(a);
        }
    }

    public static class Table2 {
        int b;
```

```
    public int getB() {
        return b;
    }

    public void setB(int b) {
        this.b = b;
    }

    public Table2(int b) {
        this.b = b;
    }

    public static Table2 build(int b) {
        return new Table2(b);
    }

    @Override
    public String toString() {
        return "Table2{" +
                "b=" + b +
                '}';
    }
}

public static class Record<R1, R2> {
    R1 r1;
    R2 r2;

    public R1 getR1() {
        return r1;
    }

    public void setR1(R1 r1) {
        this.r1 = r1;
    }

    public R2 getR2() {
        return r2;
    }
```

```
        public void setR2(R2 r2) {
            this.r2 = r2;
        }

        public Record(R1 r1, R2 r2) {
            this.r1 = r1;
            this.r2 = r2;
        }

        @Override
        public String toString() {
            return "Record{" +
                    "r1=" + r1 +
                    ", r2=" + r2 +
                    '}';
        }

        public static <R1, R2> Record<R1, R2> build(R1 r1, R2 r2) {
            return new Record(r1, r2);
        }
    }

    public static enum JoinType {
        innerJoin, leftJoin
    }


    public static interface Filter<R1, R2> {
        boolean accept(R1 r1, R2 r2);
    }

    public static <R1, R2> List<Record<R1, R2>> join(List<R1> table1,
List<R2> table2, JoinType joinType, Filter<R1, R2> onFilter,
Filter<R1, R2> whereFilter) {
        if (Objects.isNull(table1) || Objects.isNull(table2) ||
joinType == null) {
            return new ArrayList<>();
        }

        List<Record<R1, R2>> result = new CopyOnWriteArrayList<>();
```

```
        for (R1 r1 : table1) {
            List<Record<R1, R2>> onceJoinResult = joinOn(r1, table2,
onFilter);
            result.addAll(onceJoinResult);
        }

        if (joinType == JoinType.leftJoin) {
            List<R1> r1Record =
result.stream().map(Record::getR1).collect(Collectors.toList());
            List<Record<R1, R2>> leftAppendList = new ArrayList<>();
            for (R1 r1 : table1) {
                if (!r1Record.contains(r1)) {
                    leftAppendList.add(Record.build(r1, null));
                }
            }
            result.addAll(leftAppendList);
        }
        if (Objects.nonNull(whereFilter)) {
            for (Record<R1, R2> record : result) {
                if (!whereFilter.accept(record.r1, record.r2)) {
                    result.remove(record);
                }
            }
        }
        return result;
    }

    public static <R1, R2> List<Record<R1, R2>> joinOn(R1 r1, List<R2>
table2, Filter<R1, R2> onFilter) {
        List<Record<R1, R2>> result = new ArrayList<>();
        for (R2 r2 : table2) {
            if (Objects.nonNull(onFilter) ? onFilter.accept(r1, r2) :
true) {
                result.add(Record.build(r1, r2));
            }
        }
        return result;
    }
```

```
    @Test
    public void innerJoin() {
        List<Table1> table1 = Arrays.asList(Table1.build(1),
Table1.build(2), Table1.build(3));
        List<Table2> table2 = Arrays.asList(Table2.build(3),
Table2.build(4), Table2.build(5));

        join(table1, table2, JoinType.innerJoin, null,
null).forEach(System.out::println);
        System.out.println("-----------------");
        join(table1, table2, JoinType.innerJoin, (r1, r2) -> r1.a ==
r2.b, null).forEach(System.out::println);
    }

    @Test
    public void leftJoin() {
        List<Table1> table1 = Arrays.asList(Table1.build(1),
Table1.build(2), Table1.build(3));
        List<Table2> table2 = Arrays.asList(Table2.build(3),
Table2.build(4), Table2.build(5));

        join(table1, table2, JoinType.leftJoin, (r1, r2) -> r1.a ==
r2.b, null).forEach(System.out::println);
        System.out.println("-----------------");
        join(table1, table2, JoinType.leftJoin, (r1, r2) -> r1.a > 10,
null).forEach(System.out::println);
    }

}
```

代码中的**innerJoin()**方法模拟了下面的**sql**:

```
mysql> select * from test1 t1,test2 t2;
+------+------+
| a    | b    |
+------+------+
|    1 |    3 |
|    2 |    3 |
|    3 |    3 |
|    1 |    4 |
|    2 |    4 |
```

```
|     3 |     4 |
|     1 |     5 |
|     2 |     5 |
|     3 |     5 |
+------+------+
9 rows in set (0.00 sec)

mysql> select * from test1 t1,test2 t2 where t1.a = t2.b;
+------+------+
| a    | b    |
+------+------+
|     3 |     3 |
+------+------+
1 row in set (0.00 sec)
```

运行一下innerJoin()输出如下：

```
Record{r1=Table1{a=1}, r2=Table2{b=3}}
Record{r1=Table1{a=1}, r2=Table2{b=4}}
Record{r1=Table1{a=1}, r2=Table2{b=5}}
Record{r1=Table1{a=2}, r2=Table2{b=3}}
Record{r1=Table1{a=2}, r2=Table2{b=4}}
Record{r1=Table1{a=2}, r2=Table2{b=5}}
Record{r1=Table1{a=3}, r2=Table2{b=3}}
Record{r1=Table1{a=3}, r2=Table2{b=4}}
Record{r1=Table1{a=3}, r2=Table2{b=5}}
-----------------
Record{r1=Table1{a=3}, r2=Table2{b=3}}
```

对比一下sql和java的结果，输出的结果条数、数据基本上一致，唯一不同的是顺序上面不一样，顺序为何不一致，稍微介绍。

代码中的`leftJoin()`方法模拟了下面的sql：

```
mysql> select * from test1 t1 left join test2 t2 on t1.a = t2.b;
+------+------+
| a    | b    |
+------+------+
|     3 |     3 |
```

```
|    1 | NULL |
|    2 | NULL |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test1 t1 left join test2 t2 on t1.a>10;
+------+------+
| a    | b    |
+------+------+
|    1 | NULL |
|    2 | NULL |
|    3 | NULL |
+------+------+
3 rows in set (0.00 sec)
```

运行leftJoin()，结果如下：

```
Record{r1=Table1{a=3}, r2=Table2{b=3}}
Record{r1=Table1{a=1}, r2=null}
Record{r1=Table1{a=2}, r2=null}
-----------------
Record{r1=Table1{a=1}, r2=null}
Record{r1=Table1{a=2}, r2=null}
Record{r1=Table1{a=3}, r2=null}
```

效果和sql的效果完全一致，可以对上。

现在我们来讨论java输出的顺序为何和sql不一致?

> 上面java代码中两个表的连接查询使用了嵌套循环，外循环每执行一次，内循环的表都会全部遍历一次，如果放到mysql中，就相当于内表（被驱动表）全部扫描了一次（一次全表io读取操作），主表（外循环）如果有n条数据，那么从表就需要全表扫描n次，表的数据是存储在磁盘中，每次全表扫描都需要做io操作，io操作是最耗时间的，如果mysql按照上面的java方式实现，那效率肯定很低。

那mysql是如何优化的呢?

msql内部使用了一个内存缓存空间，就叫他join_buffer吧，先把外循环的数据放到join_buffer中，然后对从表进行遍历，从表中取一条数据和join_buffer的数据进行比较，然后从表中再取第2条和join_buffer数据进行比较，直到从表遍历完成，使用这方方式来减少从表的io扫描次数，当join_buffer足够大的时候，大到可以存放主表所有数据，那么从表只需要全表扫描一次（即只需要一次全表io读取操作）。

mysql中这种方式叫做Block Nested Loop。

java代码改进一下，来实现join_buffer的过程。

java代码改进版本

```
package com.itsoku.sql;

import org.junit.Test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.stream.Collectors;

import com.itsoku.sql.Test1.*;

public class Test2 {

    public static int joinBufferSize = 10000;
    public static List<?> joinBufferList = new ArrayList<>();

    public static <R1, R2> List<Record<R1, R2>> join(List<R1> table1,
List<R2> table2, JoinType joinType, Filter<R1, R2> onFilter,
Filter<R1, R2> whereFilter) {
        if (Objects.isNull(table1) || Objects.isNull(table2) ||
joinType == null) {
            return new ArrayList<>();
        }
```

```
        List<Test1.Record<R1, R2>> result = new
CopyOnWriteArrayList<>();

        int table1Size = table1.size();
        int fromIndex = 0, toIndex = joinBufferSize;
        toIndex = Integer.min(table1Size, toIndex);
        while (fromIndex < table1Size && toIndex <= table1Size) {
            joinBufferList = table1.subList(fromIndex, toIndex);
            fromIndex = toIndex;
            toIndex += joinBufferSize;
            toIndex = Integer.min(table1Size, toIndex);

            List<Record<R1, R2>> blockNestedLoopResult =
blockNestedLoop((List<R1>) joinBufferList, table2, onFilter);
            result.addAll(blockNestedLoopResult);
        }

        if (joinType == JoinType.leftJoin) {
            List<R1> r1Record =
result.stream().map(Record::getR1).collect(Collectors.toList());
            List<Record<R1, R2>> leftAppendList = new ArrayList<>();
            for (R1 r1 : table1) {
                if (!r1Record.contains(r1)) {
                    leftAppendList.add(Record.build(r1, null));
                }
            }
            result.addAll(leftAppendList);
        }
        if (Objects.nonNull(whereFilter)) {
            for (Record<R1, R2> record : result) {
                if (!whereFilter.accept(record.r1, record.r2)) {
                    result.remove(record);
                }
            }
        }
        return result;
    }

    public static <R1, R2> List<Record<R1, R2>>
blockNestedLoop(List<R1> joinBufferList, List<R2> table2, Filter<R1,
```

```
R2> onFilter) {
        List<Record<R1, R2>> result = new ArrayList<>();
        for (R2 r2 : table2) {
            for (R1 r1 : joinBufferList) {
                if (Objects.nonNull(onFilter) ? onFilter.accept(r1,
r2) : true) {
                    result.add(Record.build(r1, r2));
                }
            }
        }
        return result;
    }

    @Test
    public void innerJoin() {
        List<Table1> table1 = Arrays.asList(Table1.build(1),
Table1.build(2), Table1.build(3));
        List<Table2> table2 = Arrays.asList(Table2.build(3),
Table2.build(4), Table2.build(5));

        join(table1, table2, JoinType.innerJoin, null,
null).forEach(System.out::println);
        System.out.println("-----------------");
        join(table1, table2, JoinType.innerJoin, (r1, r2) -> r1.a ==
r2.b, null).forEach(System.out::println);
    }

    @Test
    public void leftJoin() {
        List<Table1> table1 = Arrays.asList(Table1.build(1),
Table1.build(2), Table1.build(3));
        List<Table2> table2 = Arrays.asList(Table2.build(3),
Table2.build(4), Table2.build(5));

        join(table1, table2, JoinType.leftJoin, (r1, r2) -> r1.a ==
r2.b, null).forEach(System.out::println);
        System.out.println("-----------------");
        join(table1, table2, JoinType.leftJoin, (r1, r2) -> r1.a > 10,
null).forEach(System.out::println);
    }
}
```

执行innerJoin()，输出：

```
Record{r1=Table1{a=1}, r2=Table2{b=3}}
Record{r1=Table1{a=2}, r2=Table2{b=3}}
Record{r1=Table1{a=3}, r2=Table2{b=3}}
Record{r1=Table1{a=1}, r2=Table2{b=4}}
Record{r1=Table1{a=2}, r2=Table2{b=4}}
Record{r1=Table1{a=3}, r2=Table2{b=4}}
Record{r1=Table1{a=1}, r2=Table2{b=5}}
Record{r1=Table1{a=2}, r2=Table2{b=5}}
Record{r1=Table1{a=3}, r2=Table2{b=5}}
-----------------
Record{r1=Table1{a=3}, r2=Table2{b=3}}
```

执行leftJoin()，输出：

```
Record{r1=Table1{a=3}, r2=Table2{b=3}}
Record{r1=Table1{a=1}, r2=null}
Record{r1=Table1{a=2}, r2=null}
-----------------
Record{r1=Table1{a=1}, r2=null}
Record{r1=Table1{a=2}, r2=null}
Record{r1=Table1{a=3}, r2=null}
```

结果和sql的结果完全一致。

扩展

表连接中还可以使用前面学过的group by、having、order by、limit。

这些关键字相当于在表连接的结果上在进行操作，大家下去可以练习一下，加深理解。

Mysql系列目录

1. 第1篇：mysql基础知识
2. 第2篇：详解mysql数据类型（重点）

3. 第3篇：管理员必备技能(必须掌握)
4. 第4篇：DDL常见操作
5. 第5篇：DML操作汇总（insert,update,delete）
6. 第6篇：select查询基础篇
7. 第7篇：玩转select条件查询，避免采坑
8. 第8篇：详解排序和分页(order by & limit)
9. 第9篇：分组查询详解（group by & having）
10. 第10篇：常用的几十个函数详解

mysql系列大概有20多篇，喜欢的请关注一下，欢迎大家加我微信itsoku或者留言交流mysql相关技术!

第12篇：子查询（本篇非常重要，高手必备）

。 这是Mysql系列第12篇。

环境：mysql5.7.25，cmd命令中进行演示。

本章节非常重要。

子查询

出现在select语句中的select语句，称为子查询或内查询。

外部的select查询语句，称为主查询或外查询。

子查询分类

按结果集的行列数不同分为4种

- 标量子查询（结果集只有一行一列）
- 列子查询（结果集只有一列多行）
- 行子查询（结果集有一行多列）
- 表子查询（结果集一般为多行多列）

按子查询出现在主查询中的不同位置分

- **select后面**：仅仅支持标量子查询。
- **from后面**：支持表子查询。
- **where或having后面**：支持标量子查询（单列单行）、列子查询（单列多行）、行子查询（多列多行）
- **exists后面（即相关子查询）**：表子查询（多行、多列）

准备测试数据

测试数据比较多，放在我的个人博客上了。

浏览器中打开链接：http://www.itsoku.com/article/209

mysql中执行里面的`javacode2018_employees`库部分的脚本。

成功创建javacode2018_employees库及5张表，如下：

表名	描述
departments	部门表
employees	员工信息表
jobs	职位信息表

locations 位置表（部门表中会用到）

job_grades 薪资等级表

select后面的子查询

子查询位于select后面的，仅仅支持标量子查询。

示例1

查询每个部门员工个数

```
SELECT
  a.*,
  (SELECT count(*)
   FROM employees b
   WHERE b.department_id = a.department_id) AS 员工个数
FROM departments a;
```

示例2

查询员工号=102的部门名称

```
SELECT (SELECT a.department_name
        FROM departments a, employees b
        WHERE a.department_id = b.department_id
              AND b.employee_id = 102) AS 部门名;
```

from后面的子查询

将子查询的结果集充当一张表，要求必须起别名，否者这个表找不到。

然后将真实的表和子查询结果表进行连接查询。

示例1

查询每个部门平均工资的工资等级

```
-- 查询每个部门平均工资
SELECT
  department_id,
  avg(a.salary)
FROM employees a
GROUP BY a.department_id;

-- 薪资等级表
SELECT *
FROM job_grades;

-- 将上面2个结果连接查询，筛选条件:平均工资 between lowest_sal and
highest_sal;
SELECT
  t1.department_id,
  sa AS '平均工资',
  t2.grade_level
FROM (SELECT
        department_id,
        avg(a.salary) sa
      FROM employees a
```

```
      GROUP BY a.department_id) t1, job_grades t2
WHERE
  t1.sa BETWEEN t2.lowest_sal AND t2.highest_sal;
```

运行最后一条结果如下：

```
mysql> SELECT
          t1.department_id,
          sa AS '平均工资',
          t2.grade_level
        FROM (SELECT
                department_id,
                avg(a.salary) sa
              FROM employees a
              GROUP BY a.department_id) t1, job_grades t2
        WHERE
          t1.sa BETWEEN t2.lowest_sal AND t2.highest_sal;
+---------------+--------------+-------------+
| department_id | 平均工资     | grade_level |
+---------------+--------------+-------------+
|          NULL |  7000.000000 | C           |
|            10 |  4400.000000 | B           |
|            20 |  9500.000000 | C           |
|            30 |  4150.000000 | B           |
|            40 |  6500.000000 | C           |
|            50 |  3475.555556 | B           |
|            60 |  5760.000000 | B           |
|            70 | 10000.000000 | D           |
|            80 |  8955.882353 | C           |
|            90 | 19333.333333 | E           |
|           100 |  8600.000000 | C           |
|           110 | 10150.000000 | D           |
+---------------+--------------+-------------+
12 rows in set (0.00 sec)
```

where和having后面的子查询

where或having后面，可以使用

1. 标量子查询（单行单列行子查询）
2. 列子查询（单列多行子查询）
3. 行子查询（一行多列）

特点

1. 子查询放在小括号内。
2. 子查询一般放在条件的右侧。
3. 标量子查询，一般搭配着单行单列操作符使用 **>、<、>=、<=、=、<>、!=**
4. 列子查询，一般搭配着多行操作符使用

   in(not in)：列表中的“任意一个”

   any或者some：和子查询返回的“某一个值”比较，比如a>some(10,20,30)，a大于子查询中任意一个即可，a大于子查询中最小值即可，等同于a>min(10,20,30)。

   all：和子查询返回的“所有值”比较，比如a>all(10,20,30)，a大于子查询中所有值，换句话说，a大于子查询中最大值即可满足查询条件，等同于a>max(10,20,30);
5. 子查询的执行优先于主查询执行，因为主查询的条件用到了子查询的结果。

mysql中的in、any、some、all

in，any，some，all分别是子查询关键词之一。

in：in常用于where表达式中，其作用是查询某个范围内的数据

any和some一样： 可以与=、>、>=、<、<=、<>结合起来使用，分别表示等于、大于、大于等于、小于、小于等于、不等于其中的任何一个数据。

all：可以与=、>、>=、<、<=、<>结合是来使用，分别表示等于、大于、大于等于、小于、小于等于、不等于其中的其中的所有数据。

下文中会经常用到这些关键字。

标量子查询

一般标量子查询，示例

查询谁的工资比Abel的高?

```
/*①查询abel的工资【改查询是标量子查询】*/
SELECT salary
FROM employees
WHERE last_name = 'Abel';

/*②查询员工信息，满足salary>①的结果*/
SELECT *
FROM employees a
WHERE a.salary > (SELECT salary
                  FROM employees
                  WHERE last_name = 'Abel');
```

多个标量子查询，示例

返回job*id*与*141*号员工相同，*salary*比*143*号员工多的员工、姓名、*job*id和工资

```
/*返回job_id与141号员工相同，salary比143号员工多的员工、姓名、job_id和工资*/
/*①查询141号员工的job_id*/
SELECT job_id
FROM employees
WHERE employee_id = 141;
/*②查询143好员工的salary*/
SELECT salary
FROM employees
WHERE employee_id = 143;
/*③查询员工的姓名、job_id、工资，要求job_id=① and salary>②*/
SELECT
  a.last_name 姓名,
  a.job_id,
```

```
  a.salary  工资
FROM employees a
WHERE a.job_id = (SELECT job_id
                  FROM employees
                  WHERE employee_id = 141)
      AND
      a.salary > (SELECT salary
                  FROM employees
                  WHERE employee_id = 143);
```

子查询+分组函数，示例

查询最低工资大于50号部门最低工资的部门id和其最低工资【having】

```
/*查询最低工资大于50号部门最低工资的部门id和其最低工资【having】*/
/*①查询50号部门的最低工资*/
SELECT min(salary)
FROM employees
WHERE department_id = 50;
/*②查询每个部门的最低工资*/
SELECT
  min(salary),
  department_id
FROM employees
GROUP BY department_id;
/*③在②的基础上筛选，满足min(salary)>①*/
SELECT
  min(a.salary) minsalary,
  department_id
FROM employees a
GROUP BY a.department_id
HAVING min(a.salary) > (SELECT min(salary)
                        FROM employees
                        WHERE department_id = 50);
```

错误的标量子查询，示例

将上面的示例③中子查询语句中的min(salary)改为salary，执行效果如下：

```
mysql> SELECT
          min(a.salary) minsalary,
          department_id
```

```
        FROM employees a
        GROUP BY a.department_id
        HAVING min(a.salary) > (SELECT salary
                                FROM employees
                                WHERE department_id = 500000);
ERROR 1242 (21000): Subquery returns more than 1 row
```

错误提示：子查询返回的结果超过了1行记录。

说明：上面的子查询只支持最多一列一行记录。

列子查询(子查询结果集一列多行)

列子查询需要搭配多行操作符使用：in(not in)、any/some、all。

为了提升效率，最好去重一下**distinct**关键字。

示例1

返回location_id是1400或1700的部门中的所有员工姓名

```
/*返回location_id是1400或1700的部门中的所有员工姓名*/
/*方式1*/
/*①查询location_id是1400或1700的部门编号*/
SELECT DISTINCT department_id
FROM departments
WHERE location_id IN (1400, 1700);

/*②查询员工姓名，要求部门是①列表中的某一个*/
SELECT a.last_name
FROM employees a
WHERE a.department_id IN (SELECT DISTINCT department_id
                          FROM departments
                          WHERE location_id IN (1400, 1700));

/*方式2：使用any实现*/
SELECT a.last_name
FROM employees a
WHERE a.department_id = ANY (SELECT DISTINCT department_id
                             FROM departments
```

```
                                WHERE location_id IN (1400, 1700));

/*拓展，下面与not in等价*/
SELECT a.last_name
FROM employees a
WHERE a.department_id <> ALL (SELECT DISTINCT department_id
                              FROM departments
                              WHERE location_id IN (1400, 1700));
```

示例2

返回其他工种中比job*id*为*'IT*PROG'工种任意工资低的员工的员工号、姓名、job_id、salary

```
/*返回其他工种中比job_id为'IT_PROG'工种任一工资低的员工的员工号、姓名、job_id、
salary*/
/*①查询job_id为'IT_PROG'部门任-工资*/
SELECT DISTINCT salary
FROM employees
WHERE job_id = 'IT_PROG';

/*②查询员工号、姓名、job_id、salary，slary<①的任意一个*/
SELECT
  last_name,
  employee_id,
  job_id,
  salary
FROM employees
WHERE salary < ANY (SELECT DISTINCT salary
                    FROM employees
                    WHERE job_id = 'IT_PROG') AND job_id != 'IT_PROG';

/*或者*/
SELECT
  last_name,
  employee_id,
  job_id,
  salary
```

```
FROM employees
WHERE salary < (SELECT max(salary)
                FROM employees
                WHERE job_id = 'IT_PROG') AND job_id != 'IT_PROG';
```

示例3

返回其他工种中比job*id*为*'IT*PROG'部门所有工资低的员工的员工号、姓名、job_id、salary

```
/*返回其他工种中比job_id为'IT_PROG'部门所有工资低的员工的员工号、姓名、job_id、salary*/
SELECT
  last_name,
  employee_id,
  job_id,
  salary
FROM employees
WHERE salary < ALL (SELECT DISTINCT salary
                    FROM employees
                    WHERE job_id = 'IT_PROG') AND job_id != 'IT_PROG';

/*或者*/
SELECT
  last_name,
  employee_id,
  job_id,
  salary
FROM employees
WHERE salary < (SELECT min(salary)
                FROM employees
                WHERE job_id = 'IT_PROG') AND job_id != 'IT_PROG';
```

行子查询（子查询结果集一行多列）

示例

查询员工编号最小并且工资最高的员工信息，3种方式。

```
/*查询员工编号最小并且工资最高的员工信息*/
/*①查询最小的员工编号*/
SELECT min(employee_id)
FROM employees;
/*②查询最高工资*/
SELECT max(salary)
FROM employees;
/*③方式1：查询员工信息*/
SELECT *
FROM employees a
WHERE a.employee_id = (SELECT min(employee_id)
                       FROM employees)
      AND salary = (SELECT max(salary)
                    FROM employees);

/*方式2*/
SELECT *
FROM employees a
WHERE (a.employee_id, a.salary) = (SELECT
                                     min(employee_id),
                                     max(salary)
                                   FROM employees);
/*方式3*/
SELECT *
FROM employees a
WHERE (a.employee_id, a.salary) in (SELECT
                                     min(employee_id),
                                     max(salary)
                                   FROM employees);
```

方式1比较常见，方式2、3更简洁。

exists后面（也叫做相关子查询）

1. 语法：exists(完整的查询语句)。
2. exists查询结果：1或0，exists查询的结果用来判断子查询的结果集中是否有值。

3. 一般来说，能用exists的子查询，绝对都能用in代替，所以exists用的少。

4. 和前面的查询不同，这先执行主查询，然后主查询查询的结果，在根据子查询进行过滤，子查询中涉及到主查询中用到的字段，所以叫相关子查询。

示例1

简单示例

```
mysql> SELECT exists(SELECT employee_id
                FROM employees
                WHERE salary = 300000) AS 'exists返回1或者0';
+----------------------+
| exists返回1或者0      |
+----------------------+
|                    0 |
+----------------------+
1 row in set (0.00 sec)
```

示例2

查询所有员工的部门名称

```
/*exists入门案例*/
SELECT exists(SELECT employee_id
                FROM employees
                WHERE salary = 300000) AS 'exists返回1或者0';

/*查询所有员工部门名*/
SELECT department_name
FROM departments a
WHERE exists(SELECT 1
              FROM employees b
              WHERE a.department_id = b.department_id);

/*使用in实现*/
SELECT department_name
FROM departments a
WHERE a.department_id IN (SELECT department_id
                          FROM employees);
```

示例3

查询没有员工的部门

```
/*查询没有员工的部门*/
/*exists实现*/
SELECT *
FROM departments a
WHERE NOT exists(SELECT 1
                 FROM employees b
                 WHERE a.department_id = b.department_id AND
b.department_id IS NOT NULL);
/*in的方式*/
SELECT *
FROM departments a
WHERE a.department_id NOT IN (SELECT department_id
                              FROM employees b
                              WHERE b.department_id IS NOT NULL);
```

上面脚本中有b.department_id IS NOT NULL，为什么，有大坑，向下看。

NULL的大坑

示例1

使用**not in**的方式查询没有员工的部门，如下：

```
SELECT *
FROM departments a
WHERE a.department_id NOT IN (SELECT department_id
                              FROM employees b);
```

运行结果：

```
mysql> SELECT *
    -> FROM departments a
    -> WHERE a.department_id NOT IN (SELECT department_id
    ->                               FROM employees b);
Empty set (0.00 sec)
```

not in的情况下，子查询中列的值为NULL的时候，外查询的结果为空。

建议：建表是，列不允许为空。

总结

1. 本文中讲解了常见的子查询，请大家务必多练习
2. 注意in、any、some、any的用法
3. **字段值为NULL的时候，not in查询有大坑，这个要注意**
4. 建议创建表的时候，列不允许为空

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**
11. **第11篇：深入了解连接查询及原理**

mysql系列大概有20多篇，喜欢的请关注一下，欢迎大家加我微信itsoku或者留言交流mysql相关技术!

第13篇：细说NULL导致的神坑，让人防不胜防

。 这是Mysql系列第13篇。

环境：mysql5.7.25，cmd命令中进行演示。

当数据的值为NULL的时候，可能出现各种意想不到的效果，让人防不胜防，我们来看看NULL导致的各种神坑，如何避免?

比较运算符中使用NULL

认真看下面的效果

```
mysql> select 1>NULL;
+--------+
| 1>NULL |
+--------+
|   NULL |
+--------+
1 row in set (0.00 sec)

mysql> select 1<NULL;
```

```
+--------+
| 1<NULL |
+--------+
|   NULL |
+--------+
1 row in set (0.00 sec)

mysql> select 1<>NULL;
+---------+
| 1<>NULL |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)

mysql> select 1>NULL;
+--------+
| 1>NULL |
+--------+
|   NULL |
+--------+
1 row in set (0.00 sec)

mysql> select 1<NULL;
+--------+
| 1<NULL |
+--------+
|   NULL |
+--------+
1 row in set (0.00 sec)

mysql> select 1>=NULL;
+---------+
| 1>=NULL |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)

mysql> select 1<=NULL;
```

```
+---------+
| 1<=NULL |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)

mysql> select 1!=NULL;
+---------+
| 1!=NULL |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)

mysql> select 1<>NULL;
+---------+
| 1<>NULL |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)

mysql> select NULL=NULL,NULL!=NULL;
+-----------+------------+
| NULL=NULL | NULL!=NULL |
+-----------+------------+
|      NULL |       NULL |
+-----------+------------+
1 row in set (0.00 sec)

mysql> select 1 in (null),1 not in (null),null in (null),null not in
(null);
+-------------+-----------------+---------------+--------------------
+
| 1 in (null) | 1 not in (null) | null in (null) | null not in (null)
|
+-------------+-----------------+---------------+--------------------
+
|        NULL |            NULL |           NULL |                NULL
```

```
|
+-------------+-----------------+---------------+--------------------
+
1 row in set (0.00 sec)

mysql> select 1=any(select null),null=any(select null);
+--------------------+----------------------+
| 1=any(select null) | null=any(select null) |
+--------------------+----------------------+
|               NULL |                 NULL |
+--------------------+----------------------+
1 row in set (0.00 sec)

mysql> select 1=all(select null),null=all(select null);
+--------------------+----------------------+
| 1=all(select null) | null=all(select null) |
+--------------------+----------------------+
|               NULL |                 NULL |
+--------------------+----------------------+
1 row in set (0.00 sec)
```

结论：任何值和NULL使用运算符（>、<、>=、<=、!=、<>）或者（in、not in、any/some、all）比较时，返回值都为NULL，NULL作为布尔值的时候，不为1也不为0。

准备数据

```
mysql> create table test1(a int,b int);
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test1 values (1,1),(1,null),(null,null);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
```

```
| NULL | NULL |
+------+------+
3 rows in set (0.00 sec)
```

上面3条数据，认真看一下，特别是注意上面NULL的记录。

IN、NOT IN和NULL比较

IN和NULL比较

```
mysql> select * from test1;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
| NULL | NULL |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test1 where a in (null);
Empty set (0.00 sec)

mysql> select * from test1 where a in (null,1);
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
+------+------+
2 rows in set (0.00 sec)
```

结论：当IN和NULL比较时，无法查询出为NULL的记录。

NOT IN 和NULL比较

```
mysql> select * from test1 where a not in (1);
Empty set (0.00 sec)
```

```
mysql> select * from test1 where a not in (null);
Empty set (0.00 sec)

mysql> select * from test1 where a not in (null,2);
Empty set (0.00 sec)

mysql> select * from test1 where a not in (2);
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
+------+------+
2 rows in set (0.00 sec)
```

结论：当**NOT IN** 后面有**NULL**值时，不论什么情况下，整个**sql**的查询结果都为空。

EXISTS、NOT EXISTS和NULL比较

```
mysql> select * from test2;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
| NULL | NULL |
+------+------+
3 rows in set (0.00 sec)

mysql> select * from test1 t1 where exists (select * from test2 t2
where t1.a = t2.a);
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    1 | NULL |
+------+------+
2 rows in set (0.00 sec)
```

```
mysql> select * from test1 t1 where not exists (select * from test2 t2
where t1.a = t2.a);
+------+------+
| a    | b    |
+------+------+
| NULL | NULL |
+------+------+
1 row in set (0.00 sec)
```

上面我们复制了表test1创建了表test2。

查询语句中使用exists、not exists对比test1.a=test2.a，因为=不能比较**NULL**，结果和预期一致。

判断NULL只能用IS NULL、IS NOT NULL

```
mysql> select 1 is not null;
+---------------+
| 1 is not null |
+---------------+
|             1 |
+---------------+
1 row in set (0.00 sec)

mysql> select 1 is null;
+-----------+
| 1 is null |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)

mysql> select null is null;
+--------------+
| null is null |
+--------------+
|            1 |
+--------------+
```

```
1 row in set (0.00 sec)

mysql> select null is not null;
+------------------+
| null is not null |
+------------------+
|                0 |
+------------------+
1 row in set (0.00 sec)
```

看上面的效果，返回的结果为1或者0。

结论：判断是否为空只能用**IS NULL、IS NOT NULL**。

聚合函数中NULL的坑

示例

```
mysql> select count(a),count(b),count(*) from test1;
+----------+----------+----------+
| count(a) | count(b) | count(*) |
+----------+----------+----------+
|        2 |        1 |        3 |
+----------+----------+----------+
1 row in set (0.00 sec)
```

count(a)返回了2行记录，a字段为NULL的没有统计出来。

count(b)返回了1行记录，为NULL的2行记录没有统计出来。

count(*)可以统计所有数据，不论字段的数据是否为NULL。

再继续看

```
mysql> select * from test1 where a is null;
+------+------+
| a    | b    |
+------+------+
| NULL | NULL |
+------+------+
1 row in set (0.00 sec)
```

```
mysql> select count(a) from test1 where a is null;
+----------+
| count(a) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)
```

上面第1个sql使用is null查询出了结果，第2个sql中count(a)返回的是0行。

结论：**count(字段)无法统计字段为NULL的值，count(*)可以统计值为null的行。**

NULL不能作为主键的值

```
mysql> create table test3(a int primary key,b int);
Query OK, 0 rows affected (0.01 sec)

mysql> insert into test3 values (null,1);
ERROR 1048 (23000): Column 'a' cannot be null
```

上面我们创建了一个表test3，字段a未指定不能为空，插入了一条NULL的数据，报错原因：a 字段的值不能为NULL，我们看一下表的创建语句：

```
mysql> show create table test3;
+-------+------------+
| Table | Create Table       |
+-------+------------+
| test3 | CREATE TABLE `test3` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
+-------+------------+
1 row in set (0.00 sec)
```

从上面的脚本可以看出，当字段为主键的时候，字段会自动设置为not null。

结论：**当字段为主键的时候，字段会自动设置为not null。**

看了上面这些还是比较晕，NULL的情况确实比较难以处理，容易出错，最有效的方法就是避免使用NULL。所以，强烈建议创建字段的时候字段不允许为NULL，设置一个默认值。

总结

- NULL作为布尔值的时候，不为1也不为0
- 任何值和NULL使用运算符（>、<、>=、<=、!=、<>）或者（in、not in、any/some、all），返回值都为NULL
- 当IN和NULL比较时，无法查询出为NULL的记录
- 当NOT IN 后面有NULL值时，不论什么情况下，整个sql的查询结果都为空
- 判断是否为空只能用IS NULL、IS NOT NULL
- count(字段)无法统计字段为NULL的值，count(*)可以统计值为null的行
- 当字段为主键的时候，字段会自动设置为not null
- NULL导致的坑让人防不胜防，强烈建议创建字段的时候字段不允许为NULL，给个默认值

Mysql系列目录

1. 第1篇：mysql基础知识
2. 第2篇：详解mysql数据类型（重点）
3. 第3篇：管理员必备技能(必须掌握)
4. 第4篇：DDL常见操作
5. 第5篇：DML操作汇总（insert,update,delete）

6. 第6篇：**select**查询基础篇
7. 第7篇：玩转**select**条件查询，避免采坑
8. 第8篇：详解排序和分页**(order by & limit)**
9. 第9篇：分组查询详解（**group by & having**）
10. 第**10**篇：常用的几十个函数详解
11. 第**11**篇：深入了解连接查询及原理
12. 第**12**篇：子查询（非常重要，高手必备）

mysql系列大概有**20**多篇，喜欢的请关注一下，欢迎大家加我微信**itsoku**或者留言交流**mysql**相关技术**!**

第14篇：事务详解

这是Mysql系列第14篇。

环境：mysql5.7.25，cmd命令中进行演示。

开发过程中，会经常用到数据库事务，所以本章非常重要。

本篇内容

1. 什么是事务，它有什么用?
2. 事务的几个特性
3. 事务常见操作指令详解
4. 事务的隔离级别详解
5. 脏读、不可重复读、可重复读、幻读详解
6. 演示各种隔离级别产生的现象
7. 关于隔离级别的选择

什么是事务?

数据库中的事务是指对数据库执行一批操作，这些操作最终要么全部执行成功，要么全部失败，不会存在部分成功的情况。

举个例子

比如A用户给B用户转账100操作，过程如下:

```
1.从A账户扣100
2.给B账户加100
```

如果在事务的支持下，上面最终只有2种结果:

1. 操作成功: A账户减少100; B账户增加100
2. 操作失败: A、B两个账户都没有发生变化

如果没有事务的支持，可能出现错: A账户减少了100，此时系统挂了，导致B账户没有加上100，而A账户凭空少了100。

事务的几个特性(ACID)

原子性(Atomicity)

事务的整个过程如原子操作一样，最终要么全部成功，或者全部失败，这个原子性是从最终结果来看的，从最终结果来看这个过程是不可分割的。

一致性(Consistency)

一个事务必须使数据库从一个一致性状态变换到另一个一致性状态。

首先回顾一下一致性的定义。所谓一致性，指的是数据处于一种有意义的状态，这种状态是**语义上的**而不是**语法上的**。最常见的例子是转帐。例如从帐户A转一笔钱到帐户B上，如果帐户A上的钱减少了，而帐户B上的钱却没有增加，那么我们认为此时数据处于不一致的状态。

从这段话的理解来看，所谓一致性，即，从实际的业务逻辑上来说，最终结果是对的、是跟程序员的所期望的结果完全符合的

隔离性(Isolation)

一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性(Durability)

一个事务一旦提交，他对数据库中数据的改变就应该是永久性的。当事务提交之后，数据会持久化到硬盘，修改是永久性的。

Mysql中事务操作

mysql中事务默认是隐式事务，执行insert、update、delete操作的时候，数据库自动开启事务、提交或回滚事务。

是否开启隐式事务是由变量`autocommit`控制的。

所以事务分为隐式事务和显式事务。

隐式事务

事务自动开启、提交或回滚，比如insert、update、delete语句，事务的开启、提交或回滚由mysql内部自动控制的。

查看变量autocommit是否开启了自动提交

```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)
```

autocommit为ON表示开启了自动提交。

显式事务

事务需要手动开启、提交或回滚，由开发者自己控制。

2种方式手动控制事务：

方式1：

语法：

```
//设置不自动提交事务
set autocommit=0;
//执行事务操作
commit|rollback;
```

示例1：提交事务操作，如下：

```
mysql> create table test1 (a int);
Query OK, 0 rows affected (0.01 sec)

mysql> select * from test1;
Empty set (0.00 sec)
```

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values(1);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

示例2：回滚事务操作，如下：

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values(2);
Query OK, 1 row affected (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

可以看到上面数据回滚了。

我们把autocommit还原回去：

```
mysql> set autocommit=1;
Query OK, 0 rows affected (0.00 sec)
```

方式2：

语法：

```
start transaction;//开启事务
//执行事务操作
commit|rollback;
```

示例1：提交事务操作，如下：

```
mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (2);
Query OK, 1 row affected (0.00 sec)

mysql> insert into test1 values (3);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
```

```
+------+
3 rows in set (0.00 sec)
```

上面成功插入了2条数据。

示例2：回滚事务操作，如下：

```
mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from test1;
Query OK, 3 rows affected (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

上面事务中我们删除了test1的数据，显示删除了3行，最后回滚了事务。

savepoint关键字

在事务中我们执行了一大批操作，可能我们只想回滚部分数据，怎么做呢?

我们可以将一大批操作分为几个部分，然后指定回滚某个部分。可以使用savepoin来实现，效果如下:

先清除test1表数据:

```
mysql> delete from test1;
Query OK, 3 rows affected (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)
```

演示savepoint效果，认真看:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)

mysql> savepoint part1;//设置一个保存点
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (2);
Query OK, 1 row affected (0.00 sec)

mysql> rollback to part1;//将savepint = part1的语句到当前语句之间所有的操作回滚
Query OK, 0 rows affected (0.00 sec)

mysql> commit;//提交事务
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
```

```
|    1 |
+------+
1 row in set (0.00 sec)
```

从上面可以看出，执行了2次插入操作，最后只插入了1条数据。

savepoint需要结合rollback to sp1一起使用，可以将保存点sp1到rollback to之间的操作回滚掉。

只读事务

表示在事务中执行的是一些只读操作，如查询，但是不会做insert、update、delete操作，数据库内部对只读事务可能会有一些性能上的优化。

用法如下：

```
start transaction read only;
```

示例：

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> start transaction read only;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)

mysql> delete from test1;
ERROR 1792 (25006): Cannot execute statement in a READ ONLY
transaction.
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)
```

只读事务中执行delete会报错。

事务中的一些问题

这些问题主要是基于数据在多个事务中的可见性来说的。

脏读

一个事务在执行的过程中读取到了其他事务还没有提交的数据。 这个还是比较好理解的。

读已提交

从字面上我们就可以理解，即一个事务操作过程中可以读取到其他事务已经提交的数据。

事务中的每次读取操作，读取到的都是数据库中其他事务已提交的最新的数据（相当于当前读）

可重复读

一个事务操作中对于一个读取操作不管多少次，读取到的结果都是一样的。

幻读

脏读、不可重复读、可重复读、幻读，其中最难理解的是幻读

以mysql为例：

幻读在可重复读的模式下才会出现，其他隔离级别中不会出现

幻读现象例子：

可重复读模式下，比如有个用户表，手机号码为主键，有两个事物进行如下操作

事务A操作如下： 1、打开事务 2、查询号码为X的记录，不存在 3、插入号码为X的数据，插入报错（为什么会报错，先向下看） 4、查询号码为X的记录，发现还是不存在（由于是可重复读，所以读取记录X还是不存在的）

事物B操作：在事务A第2步操作时插入了一条X的记录，所以会导致A中第3步插入报错（违反了唯一约束）

上面操作对A来说就像发生了幻觉一样，明明查询X（A中第二步、第四步）不存在，但却无法插入成功

幻读可以这么理解：事务中后面的操作（插入号码X）需要上面的读取操作（查询号码X的记录）提供支持，但读取操作却不能支持下面的操作时产生的错误，就像发生了幻觉一样。

如果还是理解不了的，继续向下看，后面后详细的演示。

事务的隔离级别

当多个事务同时进行的时候，如何确保当前事务中数据的正确性，比如A、B两个事物同时进行的时候，A是否可以看到B已提交的数据或者B未提交的数据，这个需要依靠事务的隔离级别来保证，不同的隔离级别中所产生的效果是不一样的。

事务隔离级别主要是解决了上面多个事务之间数据可见性及数据正确性的问题。

隔离级别分为4种：

1. 读未提交：**READ-UNCOMMITTED**
2. 读已提交：**READ-COMMITTED**
3. 可重复读：**REPEATABLE-READ**
4. 串行：**SERIALIZABLE**

上面4中隔离级别越来越强，会导致数据库的并发性也越来越低。

查看隔离级别

```
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

隔离级别的设置

分2步骤，修改文件、重启mysql，如下：

修改mysql中的my.init文件，我们将隔离级别设置为：READ-UNCOMMITTED，如下：

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=READ-UNCOMMITTED
```

以管理员身份打开cmd窗口，重启mysql，如下：

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

各种隔离级别中会出现的问题

不可重复读
隔离级别

READ-UNCOMMITTED

READ-COMMITTED

R无无有
E
P
E
A
T
A
B
L
E
-
R
E
A
D

S无无无
E
R
I
A
L
I
Z
A
B
L
E

表格中和网上有些不一样，主要是幻读这块，幻读只会在可重复读级别中才会出现，其他级别下不存在。

下面我们来演示一下，各种隔离级别中可见性的问题，开启两个窗口，叫做A、B窗口，两个窗口中登录mysql。

READ-UNCOMMITTED：读未提交

将隔离级别置为READ-UNCOMMITTED：

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=READ-UNCOMMITTED
```

重启mysql:

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

查看隔离级别:

```
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-UNCOMMITTED |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

先清空test1表数据:

```
delete from test1;
select * from test1;
```

按时间顺序在2个窗口中执行下面操作:

窗口A	窗口B
时间	

```
Ts
1t
 a
 r
 t
 t
 r
 a
 n
 s
 a
 c
 ti
 o
 n
 ;
Ts
2e
 l
 e
 c
 t
 *
 f
 r
 o
 m
 t
 e
 s
 t
 1
 ;
```

T3 start transaction;

T4 insert into otest1 values(1);

T5 select * from test1;

Ts6 elect * from test1;

T c
7 o
m
m
it
;

T c
8 o
m
m
it
;

A窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

B窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

看一下:

T2-A：无数据，T6-A：有数据，T6时刻B还未提交，此时A已经看到了B插入的数据，说明出现了脏读。

T2-A：无数据，T6-A：有数据，查询到的结果不一样，说明不可重复读。

结论：读未提交情况下，可以读取到其他事务还未提交的数据，多次读取结果不一样，出现了脏读、不可重复读

READ-COMMITTED：读已提交

将隔离级别置为READ-COMMITTED

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=READ-COMMITTED
```

重启mysql:

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

查看隔离级别：

```
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

先清空test1表数据：

```
delete from test1;
select * from test1;
```

按时间顺序在2个窗口中执行下面操作：

窗窗
时口口
间AB

Ts
1t
 a
 r
 t
 t
 r
 a
 n
 s
 a
 c
 ti
 o
 n
 ;

T2 select * from test1;

T3 start transaction;

T4 insert into otest1 values(1);

T5 select * from test1;

Ts6 elect * from test1;

T7 commit;

Ts8elect*fromtest1;

Tc9ommit;

A窗口如下:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)

mysql> select * from test1;
```

```
Empty set (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

B窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

看一下：

T5-B：有数据，T6-A窗口：无数据，A看不到B的数据，**说明没有脏读**。

T6-A窗口：无数据，T8-A：看到了B插入的数据，此时B已经提交了，A看到了B已提交的数据，**说明可以读取到已提交的数据**。

T2-A、T6-A：无数据，T8-A：有数据，多次读取结果不一样，**说明不可重复读**。

结论：读已提交情况下，无法读取到其他事务还未提交的数据，可以读取到其他事务已经提交的数据，多次读取结果不一样，未出现脏读，出现了读已提交、不可重复读。

REPEATABLE-READ：可重复读

将隔离级别置为REPEATABLE-READ

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=REPEATABLE-READ
```

重启mysql：

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

查看隔离级别：

```
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

先清空test1表数据：

```
delete from test1;
select * from test1;
```

按时间顺序在2个窗口中执行下面操作：

窗窗
口口
间AB

```
Ts
1t
 a
 r
 t
 t
 r
 a
 n
 s
 a
 c
 ti
 o
 n
 ;
Ts
2e
 l
 e
 c
 t
 *
 f
 r
 o
 m
 t
 e
 s
 t
 1
 ;
```

T3 start transaction;

T4 insert into otest1 values(1);

T5 select * from test1;

Ts6 elect * from test1;

```
T7 commit;

Ts8elect*fromtest1;

Tc9ommit;
```

Ts
1e
0l
 e
 c
 t
 *
 f
 r
 o
 m
 t
 e
 s
 t
 1
 ;

A窗口如下:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)

mysql> select * from test1;
Empty set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
```

```
|     1 |
+-------+
2 rows in set (0.00 sec)
```

B窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test1;
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

看一下：

T2-A、T6-A窗口：无数据，T5-B：有数据，A看不到B的数据，**说明没有脏读。**

T8-A：无数据，此时B已经提交了，A看不到B已提交的数据，A中3次读的结果一样都是没有数据的，**说明可重复读。**

结论：可重复读情况下，未出现脏读，未读取到其他事务已提交的数据，多次读取结果一致，即可重复读。

幻读演示

幻读只会在REPEATABLE-READ（可重复读）级别下出现，需要先把隔离级别改为可重复读。

将隔离级别置为REPEATABLE-READ

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=REPEATABLE-READ
```

重启mysql:

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

查看隔离级别:

```
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

准备数据:

```
mysql> create table t_user(id int primary key,name varchar(16) unique key);
Query OK, 0 rows affected (0.01 sec)

mysql> insert into t_user values (1,'路人甲Java'),(2,'路人甲Java');
ERROR 1062 (23000): Duplicate entry '路人甲Java' for key 'name'

mysql> select * from t_user;
Empty set (0.00 sec)
```

> 上面我们创建t_user表，name添加了唯一约束，表示name不能重复，否则报错。

按时间顺序在2个窗口中执行下面操作:

时间	窗口A	窗口B
T1	start transaction;	
T2		start transaction;

T3 --捐入跻人耳Java
insertintot_uservalues(1'

T4 select*fromt_user;

T5--查看路人甲Java是否有在
select*fromt_userwhe

T6 commit;

T7--捐入跻人耳Java
insertintot_uservalues(2'

T8--查看路人甲Java是否有在
select*fromt_userwhe

Tc
9o
m
m
it
;

A窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_user where name='路人甲Java';
Empty set (0.00 sec)

mysql> insert into t_user values (2,'路人甲Java');
ERROR 1062 (23000): Duplicate entry '路人甲Java' for key 'name'
mysql> select * from t_user where name='路人甲Java';
Empty set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

B窗口如下：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t_user values (1,'路人甲Java');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t_user;
+----+---------------+
| id | name          |
+----+---------------+
|  1 | 路人甲Java    |
+----+---------------+
1 row in set (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

看一下:

A想插入数据路人甲Java，插入之前先查询了一下（T5时刻）该用户是否存在，发现不存在，然后在T7时刻执行插入，报错了，报数据已经存在了，因为T6时刻B已经插入了路人甲Java。

然后A有点郁闷，刚才查的时候不存在的，然后A不相信自己的眼睛，又去查一次（T8时刻），发现路人甲Java还是不存在的。

此时A心里想：数据明明不存在啊，为什么无法插入呢？这不是懵逼了么，A觉得如同发生了幻觉一样。

SERIALIZABLE：串行

> **SERIALIZABLE会让并发的事务串行执行（多个事务之间读写、写读、写写会产生互斥，效果就是串行执行，多个事务之间的读读不会产生互斥）。**
>
> 读写互斥：事务A中先读取操作，事务B发起写入操作，事务A中的读取会导致事务B中的写入处于等待状态，直到A事务完成为止。
>
> 表示我开启一个事务，为了保证事务中不会出现上面说的问题（脏读、不可重复读、读已提交、幻读），那么我读取的时候，其他事务有修改数据的操作需要排队等待，等待我读取完成之后，他们才可以继续。
>
> 写读、写写也是互斥的，读写互斥类似。
>
> 这个类似于java中的
> java.util.concurrent.lock.ReentrantReadWriteLock类产生的效果。

下面演示读写互斥的效果。

将隔离级别置为SERIALIZABLE

```
# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行
transaction-isolation=SERIALIZABLE
```

重启mysql：

```
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。

C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。
```

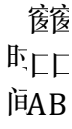
查看隔离级别：

```
mysql> show variables like 'transaction_isolation';
+-----------------------+--------------+
| Variable_name         | Value        |
+-----------------------+--------------+
| transaction_isolation | SERIALIZABLE |
+-----------------------+--------------+
1 row in set, 1 warning (0.00 sec)
```

先清空test1表数据：

```
delete from test1;
select * from test1;
```

按时间顺序在2个窗口中执行下面操作：



```
Ts
1t
 a
 r
 t
 t
 r
 a
 n
 s
 a
 c
 ti
 o
 n
 ;
Ts
2e
 l
 e
 c
 t
 *
 f
 r
 o
 m
 t
 e
 s
 t
 1
 ;
```

T3 start transaction;

T4 insert into test1 values(1);

T5 commit;

T6 commit;

按时间顺序运行上面的命令，会发现T4-B这样会被阻塞，直到T5-A执行完毕。

上面这个演示的是读写互斥产生的效果，大家可以自己去写一下写读、写写互斥的效果。

可以看出来，事务只能串行执行了。串行情况下不存在脏读、不可重复读、幻读的问题了。

关于隔离级别的选择

1. 需要对各种隔离级别产生的现象非常了解，然后选择的时候才能游刃有余
2. 隔离级别越高，并发性也低，比如最高级别`SERIALIZABLE`会让事物串行执行，并发操作变成串行了，会导致系统性能直接降低。
3. 具体选择哪种需要结合具体的业务来选择。
4. 读已提交（READ-COMMITTED）通常用的比较多。

总结

1. 理解事务的4个特性：原子性、一致性、隔离性、持久性
2. 掌握事务操作常见命令的介绍
3. `set autocommit`可以设置是否开启自动提交事务
4. start transaction：开启事务
5. start transaction read only：开启只读事物
6. commit：提交事务
7. rollback：回滚事务
8. savepoint：设置保存点
9. rollback to 保存点：可以回滚到某个保存点
10. 掌握4种隔离级别及了解其特点

11. 了解脏读、不可重复读、幻读

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**
11. **第11篇：深入了解连接查询及原理**
12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**

mysql系列大概有20多篇，喜欢的请关注一下，欢迎大家加我微信itsoku或者留言交流mysql相关技术!

第15篇：视图

这是Mysql系列第15篇。

环境：mysql5.7.25，cmd命令中进行演示。

需求背景

电商公司领导说：给我统计一下：当月订单总金额、订单量、男女订单占比等信息，我们啪啦啪啦写了一堆很复杂的sql，然后发给领导。

这样一大片sql，发给领导，你们觉得好么？

如果领导只想看其中某个数据，还需要修改你发来的sql，领导日后想新增其他的统计指标，你又会发送一大坨sql给领导，对于领导来说这个sql看起来很复杂，难以维护。

实际上领导并不关心你是怎么实现的，他关心的只是这些指标，并且方便查看、查询，而你却把复杂的实现都发给了领导。

那我们有什么办法隐藏这些细节，只暴露简洁的结果呢？

数据库已经帮我们想到了：使用视图来解决这个问题。

什么是视图

概念

视图是在mysql5之后出现的，是一种虚拟表，行和列的数据来自于定义视图时使用的一些表中，**视图的数据是在使用视图的时候动态生成的，视图只保存了sql的逻辑，不保存查询的结果**。

使用场景

多个地方使用到同样的查询结果，并且该查询结果比较复杂的时候，我们可以使用视图来隐藏复杂的实现细节。

视图和表的区别

实际中是否占用物理空间

使用create view语法创建视图

增删改查，实际上我们只是保存了sql的逻辑

使用查询

表create table 保增有册了改数查据

视图的好处

- 简化复杂的sql操作，不用知道他的实现细节
- 隔离了原始表，可以不让使用视图的人接触原始的表，从而保护原始数据，提高了安全性

准备测试数据

测试数据比较多，放在我的个人博客上了。

浏览器中打开链接：http://www.itsoku.com/article/209

mysql中执行里面的`javacode2018_employees`库部分的脚本。

成功创建`javacode2018_employees`库及5张表，如下：

表名 描述

部门表 departments

员工信息表 employees

职位信息表 jobs

locations 位置表（部门表中会用到）

job_grades 薪资等级表

创建视图

语法

```
create view 视图名
as
查询语句;
```

视图的使用步骤

- 创建视图
- 对视图执行查询操作

案例1

查询姓名中包含a字符的员工名、部门、工种信息

```
/*案例1：查询姓名中包含a字符的员工名、部门、工种信息*/
/*①创建视图myv1*/
CREATE VIEW myv1
AS
  SELECT
    t1.last_name,
    t2.department_name,
    t3.job_title
  FROM employees t1, departments t2, jobs t3
  WHERE t1.department_id = t2.department_id
        AND t1.job_id = t3.job_id;

/*②使用视图*/
SELECT * FROM myv1 a where a.last_name like 'a%';
```

效果如下：

```
mysql> SELECT * FROM myv1 a where a.last_name like 'a%';
+-----------+-----------------+----------------------+
| last_name | department_name | job_title            |
+-----------+-----------------+----------------------+
| Austin    | IT              | Programmer           |
| Atkinson  | Shi             | Stock Clerk          |
| Ande      | Sal             | Sales Representative |
| Abel      | Sal             | Sales Representative |
+-----------+-----------------+----------------------+
4 rows in set (0.00 sec)
```

上面我们创建了一个视图：myv1，我们需要看员工姓名、部门、工种信息的时候，不用关心这个视图内部是什么样的，只需要查询视图就可以了，sql简单多了。

案例2

案例2：查询各部门的平均工资级别

```
/*案例2：查询各部门的平均工资级别*/
/*①创建视图myv1*/
CREATE VIEW myv2
AS
  SELECT
    t1.department_id 部门id,
    t1.ag     平均工资,
    t2.grade_level  工资级别
  FROM (SELECT
          department_id,
          AVG(salary) ag
        FROM employees
        GROUP BY department_id)
       t1, job_grades t2
  WHERE t1.ag BETWEEN t2.lowest_sal AND t2.highest_sal;

/*②使用视图*/
SELECT * FROM myv2;
```

效果：

```
mysql> SELECT * FROM myv2;
+----------+--------------+--------------+
| 部门id    | 平均工资     | 工资级别     |
+----------+--------------+--------------+
|     NULL |  7000.000000 | C            |
|       10 |  4400.000000 | B            |
|       20 |  9500.000000 | C            |
|       30 |  4150.000000 | B            |
|       40 |  6500.000000 | C            |
|       50 |  3475.555556 | B            |
|       60 |  5760.000000 | B            |
|       70 | 10000.000000 | D            |
|       80 |  8955.882353 | C            |
|       90 | 19333.333333 | E            |
|      100 |  8600.000000 | C            |
|      110 | 10150.000000 | D            |
+----------+--------------+--------------+
12 rows in set (0.00 sec)
```

修改视图

2种方式。

方式1

如果该视图存在，就修改，如果不存在，就创建新的视图。

```
create or replace view 视图名
as
查询语句;
```

示例

```
CREATE OR REPLACE VIEW myv3
AS
  SELECT
    job_id,
    AVG(salary) javg
  FROM employees
  GROUP BY job_id;
```

方式2

```
alter view 视图名
as
查询语句;
```

示例

```
ALTER VIEW myv3
AS
SELECT *
FROM employees;
```

删除视图

语法

```
drop view 视图名1 [,视图名2] [,视图名n];
```

可以同时删除多个视图，多个视图名称之间用逗号隔开。

示例

```
mysql> drop view myv1,myv2,myv3;
Query OK, 0 rows affected (0.00 sec)
```

查询视图结构

```
/*方式1*/
desc 视图名称;
/*方式2*/
show create view 视图名称;
```

如：

```
mysql> desc myv1;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| last_name       | varchar(25) | YES  |     | NULL    |       |
| department_name | varchar(3)  | YES  |     | NULL    |       |
| job_title       | varchar(35) | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

```
mysql> show create view myv1;
+------
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
--------------+----------------------+----------------------+
| View | Create View
| character_set_client | collation_connection |
+------
+-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
--------------+----------------------+----------------------+
| myv1 | CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL
SECURITY DEFINER VIEW `myv1` AS select `t1`.`last_name` AS
`last_name`,`t2`.`department_name` AS
`department_name`,`t3`.`job_title` AS `job_title` from ((`employees`
`t1` join `departments` `t2`) join `jobs` `t3`) where
((`t1`.`department_id` = `t2`.`department_id`) and (`t1`.`job_id` =
`t3`.`job_id`)) | utf8                 | utf8_general_ci      |
+------
+--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

show create view显示了视图的创建语句。

更新视图【基本不用】

视图的更新是更改视图中的数据，而不是更改视图中的sql逻辑。

当对视图进行更新后，也会对原始表的数据进行更新。

为了防止对原始表的数据产生更新，可以为视图添加只读权限，只允许读视图，不允许对视图进行更新。

一般情况下，极少对视图进行更新操作。

示例

```
CREATE OR REPLACE VIEW myv4
  AS
  SELECT last_name,email
  from employees;

/*插入*/
insert into myv4 VALUES ('路人甲Java','javacode2018@163.com');
SELECT * from myv4 where email like 'javacode2018%';
```

```
/*修改*/
UPDATE myv4 SET last_name = '刘德华' WHERE last_name = '路人甲Java';
SELECT * from myv4 where email like 'javacode2018%';

/*删除*/
DELETE FROM myv4 where last_name = '刘德华';
SELECT * from myv4 where email like 'javacode2018%';
```

注意：视图的更新我们一般不使用，了解即可。

总结

1. 了解视图的用途及与表的区别。
2. 掌握视图的创建、使用、修改、删除。

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**

11. 第11篇：深入了解连接查询及原理
12. 第12篇：子查询
13. 第13篇：细说NULL导致的神坑，让人防不胜防
14. 第14篇：详解事务

java高并发系列全集

1. 第1天:必须知道的几个概念
2. 第2天:并发级别
3. 第3天:有关并行的两个重要定律
4. 第4天:JMM相关的一些概念
5. 第5天:深入理解进程和线程
6. 第6天:线程的基本操作
7. 第7天:volatile与Java内存模型
8. 第8天:线程组
9. 第9天：用户线程和守护线程
10. 第10天:线程安全和synchronized关键字
11. 第11天:线程中断的几种方式
12. 第12天JUC:ReentrantLock重入锁
13. 第13天:JUC中的Condition对象
14. 第14天:JUC中的LockSupport工具类，必备技能
15. 第15天：JUC中的Semaphore（信号量）

16. 第16天：JUC中等待多线程完成的工具类CountDownLatch，必备技能
17. 第17天：JUC中的循环栅栏CyclicBarrier的6种使用场景
18. 第18天：JAVA线程池，这一篇就够了
19. 第19天：JUC中的Executor框架详解1
20. 第20天：JUC中的Executor框架详解2
21. 第21天：java中的CAS，你需要知道的东西
22. 第22天：JUC底层工具类Unsafe，高手必须要了解
23. 第23天：JUC中原子类，一篇就够了
24. 第24天：ThreadLocal、InheritableThreadLocal（通俗易懂）
25. 第25天：掌握JUC中的阻塞队列
26. 第26篇：学会使用JUC中常见的集合，常看看!
27. 第27天：实战篇，接口性能提升几倍原来这么简单
28. 第28天：实战篇，微服务日志的伤痛，一并帮你解决掉
29. 第29天：高并发中常见的限流方式
30. 第30天：JUC中工具类CompletableFuture，必备技能
31. 第31天：获取线程执行结果，这6种方法你都知道?
32. 第32天：高并发中计数器的实现方式有哪些?
33. 第33篇：怎么演示公平锁和非公平锁?
34. 第34篇：google提供的一些好用的并发工具类

mysql系列大概有20多篇，喜欢的请关注一下，欢迎大家加我微信itsoku或者留言交流mysql相关技术!

第16篇：变量

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第16篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

我们在使用mysql的过程中，变量也会经常用到，比如查询系统的配置，可以通过查看系统变量来了解，当我们需要修改系统的一些配置的时候，也可以通过修改系统变量的值来进行。

我们需要做一些批处理脚本的时候，可以使用自定义变量，来做到数据的复用。所以变量这块也挺重要，希望大家能够掌握。

本文内容

- 详解系统变量的使用
- 详解自定义变量的使用

变量分类

- 系统变量
- 自定义变量

系统变量

概念

系统变量由系统定义的，不是用户定义的，属于mysql服务器层面的。

系统变量分类

- 全局变量
- 会话变量

使用步骤

查看系统变量

```
//1.查看系统所有变量
show [global | session] variables;
//查看全局变量
show global variables;
//查看会话变量
show session variables;
show variables;
```

上面使用了show关键字

查看满足条件的系统变量

通过like模糊匹配指定的变量

```
//查看满足条件的系统变量(like模糊匹配)
show [global|session] like '%变量名%';
```

上面使用了show和like关键字。

查看指定的系统变量

```
//查看指定的系统变量的值
select @@[global.|session.]系统变量名称;
```

注意select和@@关键字，global和session后面有个.符号。

赋值

```
//方式1
set [global|session] 系统变量名=值;

//方式2
set @@[global.|session.]系统变量名=值;
```

注意：

上面使用中介绍的，全局变量需要添加global关键字，会话变量需要添加session关键字，如果不写，默认为session级别。

全局变量的使用中用到了@@关键字，后面会介绍自定义变量，自定义变量中使用了一个@符号，这点需要和全局变量区分一下。

全局变量

作用域

mysql服务器每次启动都会为所有的系统变量设置初始值。

我们为系统变量赋值，针对所有会话（连接）有效，可以跨连接，但不能跨重启，重启之后，mysql服务器会再次为所有系统变量赋初始值。

示例

查看所有全局变量

```
/*查看所有全局变量*/
show global variables;
```

查看包含'tx'字符的变量

```
/*查看包含`tx`字符的变量*/
mysql> show global variables like '%tx%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
| tx_read_only  | OFF             |
+---------------+-----------------+
2 rows in set, 1 warning (0.00 sec)

/*查看指定名称的系统变量的值，如查看事务默认自动提交设置*/
mysql> select @@global.autocommit;
+---------------------+
| @@global.autocommit |
+---------------------+
|                   0 |
+---------------------+
1 row in set (0.00 sec)
```

为某个变量赋值

```
/*为某个系统变量赋值*/
set global autocommit=0;
set @@global.autocommit=1;

mysql> set global autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@global.autocommit;
+---------------------+
| @@global.autocommit |
+---------------------+
|                   0 |
+---------------------+
1 row in set (0.00 sec)

mysql> set @@global.autocommit=1;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@global.autocommit;
```

```
+----------------------+
| @@global.autocommit |
+----------------------+
|                    1 |
+----------------------+
1 row in set (0.00 sec)
```

会话变量

作用域

针对当前会话（连接）有效，不能跨连接。

会话变量是在连接创建时由mysql自动给当前会话设置的变量。

示例

查看所有会话变量

```
/*①查看所有会话变量*/
show session variables;
```

查看满足条件的会话变量

```
/*②查看满足条件的步伐会话变量*/
/*查看包含`char`字符变量名的会话变量*/
show session variables like '%char%';
```

查看指定的会话变量的值

```
/*③查看指定的会话变量的值*/
/*查看事务默认自动提交的设置*/
select @@autocommit;
select @@session.autocommit;
/*查看事务隔离级别*/
select @@tx_isolation;
select @@session.tx_isolation;
```

为某个会话变量赋值

```
/*④为某个会话变量赋值*/
set @@session.tx_isolation='read-uncommitted';
```

```
set @@tx_isolation='read-committed';
set session tx_isolation='read-committed';
set tx_isolation='read-committed';
```

效果:

```
mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation |
+-----------------+
| READ-COMMITTED |
+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql> set tx_isolation='read-committed';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation |
+-----------------+
| READ-COMMITTED |
+-----------------+
1 row in set, 1 warning (0.00 sec)
```

自定义变量

概念

变量由用户自定义的，而不是系统提供的。

使用

使用步骤:

1. 声明
2. 赋值
3. 使用（查看、比较、运算）

分类

- 用户变量
- 局部变量

用户变量

作用域

针对当前会话（连接）有效，作用域同会话变量。

用户变量可以在任何地方使用也就是既可以在begin end里面使用，也可以在他外面使用。

使用

声明并初始化(要求声明时必须初始化)

```
/*方式1*/
set @变量名=值;
/*方式2*/
set @变量名:=值;
/*方式3*/
select @变量名:=值;
```

> 注意：
>
> 上面使用了@符合，而上面介绍全局变量使用了2个@符号，这点注意区分一下。
>
> set中=号前面冒号是可选的，select方式=前面必须有冒号

赋值（更新变量的值）

```
/*方式1：这块和变量的声明一样*/
set @变量名=值;
set @变量名:=值;
select @变量名:=值;
```

```
/*方式2*/
select 字段 into @变量名 from 表;
```

注意上面select的两种方式。

使用

```
select @变量名;
```

综合示例

```
/*set方式创建变量并初始化*/
set @username='路人甲java';
/*select into方式创建变量*/
select 'javacode2018' into @gzh;
select count(*) into @empcount from employees;

/*select :=方式创建变量*/
select @first_name:='路人甲Java',@email:='javacode2018@163.com';
/*使用变量*/
insert into employees (first_name,email) values (@first_name,@email);
```

局部变量

作用域

declare用于定义局部变量变量，在存储过程和函数中通过declare定义变量在begin...end中，且在语句之前。并且可以通过重复定义多个变量

declare变量的作用范围同编程里面类似，在这里一般是在对应的begin和end之间。在end之后这个变量就没有作用了，不能使用了。这个同编程一样。

使用

声明

```
declare 变量名 变量类型;
declare 变量名 变量类型 [default 默认值];
```

赋值

```
/*方式1*/
set 局部变量名=值;
set 局部变量名:=值;
select 局部变量名:=值;

/*方式2*/
select 字段 into 局部变量名 from 表;
```

注意：局部变量前面没有@符号

使用（查看变量的值）

```
select 局部变量名;
```

示例

```
/*创建表test1*/
drop table IF EXISTS test1;
create table test1(a int PRIMARY KEY,b int);

/*声明脚本的结束符为$$*/
DELIMITER $$
DROP PROCEDURE IF EXISTS proc1;
CREATE PROCEDURE proc1()
BEGIN
  /*声明了一个局部变量*/
  DECLARE v_a int;

  select ifnull(max(a),0)+1 into v_a from test1;
  select @v_b:=v_a*2;
  insert into test1(a,b) select v_a,@v_b;
end $$

/*声明脚本的结束符为;*/
DELIMITER ;

/*调用存储过程*/
call proc1();
```

```
/*查看结果*/
select * from test1;
```

代码中使用到了存储过程，关于存储过程的详解下章节介绍。

*delimiter*关键字

我们写sql的时候，mysql怎么判断sql是否已经结束了，可以去执行了?

需要一个结束符，当mysql看到这个结束符的时候，表示可以执行前面的语句了，mysql默认以分号为结束符。

当我们创建存储过程或者自定义函数的时候，写了很大一片sql，里面包含了很多分号，整个创建语句是一个整体，需要一起执行，此时我们就不可用用分号作为结束符了。

那么我们可以通过delimiter关键字来自定义结束符。

用法:

```
delimiter 分隔符
```

上面示例的效果

```
mysql> /*创建表test1*/
mysql> drop table IF EXISTS test1;
Query OK, 0 rows affected (0.01 sec)

mysql> create table test1(a int PRIMARY KEY,b int);
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> /*声明脚本的结束符为$$*/
mysql> DELIMITER $$
mysql> DROP PROCEDURE IF EXISTS proc1;
    -> CREATE PROCEDURE proc1()
    -> BEGIN
    ->   /*声明了一个局部变量*/
    ->   DECLARE v_a int;
    ->
```

```
    ->   select ifnull(max(a),0)+1 into v_a from test1;
    ->   select @v_b:=v_a*2;
    ->   insert into test1(a,b) select v_a,@v_b;
    -> end $$
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> /*声明脚本的结束符为;*/
mysql> DELIMITER ;
mysql>
mysql> /*调用存储过程*/
mysql> call proc1();
+-------------+
| @v_b:=v_a*2 |
+-------------+
|           2 |
+-------------+
1 row in set (0.00 sec)

Query OK, 1 row affected (0.01 sec)

mysql> /*查看结果*/
mysql> select * from test1;
+---+------+
| a | b    |
+---+------+
| 1 |    2 |
+---+------+
1 row in set (0.00 sec)
```

用户变量和局部变量对比

定义位置	语法
用户变量	当前记事本的任何地方，加@符号，不用指定类型
局部变量	定义在begin和end之间，不加@符号，要指定类型

总结

- 本文对系统变量和自定义变量的使用做了详细的说明，知识点比较细，可以多看几遍，加深理解
- 系统变量可以设置系统的一些配置信息，数据库重启之后会被还原
- 会话变量可以设置当前会话的一些配置信息，对当前会话起效
- **declare**创建的局部变量常用于存储过程和函数的创建中
- 作用域：全局变量对整个系统有效、会话变量作用于当前会话、用户变量作用于当前会话、局部变量作用于**begin end**之间
- 注意全局变量中用到了`@@`，用户变量变量用到了`@`，而局部变量没有这个符号
- `delimiter`关键字用来声明脚本的结束符

Mysql系列目录

1. 第**1**篇：**mysql**基础知识
2. 第**2**篇：详解**mysql**数据类型（重点）
3. 第**3**篇：管理员必备技能**(**必须掌握**)**
4. 第**4**篇：**DDL**常见操作
5. 第**5**篇：**DML**操作汇总（**insert,update,delete**）
6. 第**6**篇：**select**查询基础篇
7. 第**7**篇：玩转**select**条件查询，避免采坑
8. 第**8**篇：详解排序和分页**(order by & limit)**
9. 第**9**篇：分组查询详解（**group by & having**）
10. 第**10**篇：常用的几十个函数详解

11. **第11篇：深入了解连接查询及原理**
12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**
14. **第14篇：详解事务**
15. **第15篇：详解视图**

第17篇：存储过程&自定义函数详解

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第17篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

需求背景介绍

线上程序有时候出现问题导致数据错误的时候，如果比较紧急，我们可以写一个存储来快速修复这块的数据，然后再去修复程序，这种方式我们用到过不少。

存储过程相对于java程序对于java开发来说，可能并不是太好维护以及阅读，所以不建议在程序中去调用存储过程做一些业务操作。

关于自定义函数这块，若mysql内部自带的一些函数无法满足我们的需求的时候，我们可以自己开发一些自定义函数来使用。

所以建议大家掌握mysql中存储过程和自定义函数这块的内容。

本文内容

- 详解存储过程的使用
- 详解自定义函数的使用

准备数据

```
/*建库javacode2018*/
drop database if exists javacode2018;
create database javacode2018;

/*切换到javacode2018库*/
use javacode2018;

/*建表test1*/
DROP TABLE IF EXISTS t_user;
CREATE TABLE t_user (
  id   INT NOT NULL PRIMARY KEY COMMENT '编号',
  age  SMALLINT UNSIGNED NOT NULL COMMENT '年龄',
  name VARCHAR(16) NOT NULL COMMENT '姓名'
) COMMENT '用户表';
```

存储过程

概念

一组预编译好的sql语句集合，理解成批处理语句。

好处：

- 提高代码的重用性
- 简化操作
- 减少编译次数并且减少和数据库服务器连接的次数，提高了效率。

创建存储过程

```
create procedure 存储过程名([参数模式] 参数名 参数类型)
begin
      存储过程体
end
```

参数模式有3种：

in：该参数可以作为输入，也就是该参数需要调用方传入值。

out：该参数可以作为输出，也就是说该参数可以作为返回值。

inout：该参数既可以作为输入也可以作为输出，也就是说该参数需要在调用的时候传入值，又可以作为返回值。

参数模式默认为IN。

一个存储过程可以有多个输入、多个输出、多个输入输出参数。

调用存储过程

```
call 存储过程名称(参数列表);
```

注意：调用存储过程关键字是call。

删除存储过程

```
drop procedure [if exists] 存储过程名称;
```

存储过程只能一个个删除，不能批量删除。

if exists：表示存储过程存在的情况下删除。

修改存储过程

存储过程不能修改，若涉及到修改的，可以先删除，然后重建。

查看存储过程

```
show create procedure 存储过程名称;
```

可以查看存储过程详细创建语句。

示例

示例1：空参列表

创建存储过程

```
/*设置结束符为$*/
DELIMITER $
/*如果存储过程存在则删除*/
DROP PROCEDURE IF EXISTS proc1;
/*创建存储过程proc1*/
CREATE PROCEDURE proc1()
  BEGIN
    INSERT INTO t_user VALUES (1,30,'路人甲Java');
    INSERT INTO t_user VALUES (2,50,'刘德华');
  END $

/*将结束符置为;*/
DELIMITER ;
```

delimiter用来设置结束符，当mysql执行脚本的时候，遇到结束符的时候，会把结束符前面的所有语句作为一个整体运行，存储过程中的脚本有多个sql，但是需要作为一个整体运行，所以此处用到了delimiter。

mysql默认结束符是分号。

上面存储过程中向t_user表中插入了2条数据。

调用存储过程：

```
CALL proc1();
```

验证效果：

```
mysql> select * from t_user;
+----+-----+---------------+
| id | age | name          |
+----+-----+---------------+
|  1 |  30 | 路人甲Java    |
|  2 |  50 | 刘德华    |
+----+-----+---------------+
2 rows in set (0.00 sec)
```

存储过程调用成功，test1表成功插入了2条数据。

示例2：带in参数的存储过程

创建存储过程：

```
/*设置结束符为$*/
DELIMITER $
/*如果存储过程存在则删除*/
DROP PROCEDURE IF EXISTS proc2;
/*创建存储过程proc2*/
CREATE PROCEDURE proc2(id int,age int,in name varchar(16))
  BEGIN
    INSERT INTO t_user VALUES (id,age,name);
  END $

/*将结束符置为;*/
DELIMITER ;
```

调用存储过程：

```
/*创建了3个自定义变量*/
SELECT @id:=3,@age:=56,@name:='张学友';
/*调用存储过程*/
CALL proc2(@id,@age,@name);
```

验证效果：

```
mysql> select * from t_user;
+----+-----+---------------+
| id | age | name          |
+----+-----+---------------+
|  1 |  30 | 路人甲Java    |
|  2 |  50 | 刘德华    |
|  3 |  56 | 张学友    |
+----+-----+---------------+
3 rows in set (0.00 sec)
```

张学友插入成功。

示例3：带out参数的存储过程

创建存储过程：

```
delete a from t_user a where a.id = 4;
/*如果存储过程存在则删除*/
DROP PROCEDURE IF EXISTS proc3;
/*设置结束符为$*/
DELIMITER $
/*创建存储过程proc3*/
CREATE PROCEDURE proc3(id int,age int,in name varchar(16),out
user_count int,out max_id INT)
  BEGIN
    INSERT INTO t_user VALUES (id,age,name);
    /*查询出t_user表的记录，放入user_count中,max_id用来存储t_user中最小的id*/
    SELECT COUNT(*),max(id) into user_count,max_id from t_user;
  END $

/*将结束符置为;*/
DELIMITER ;
```

proc3中前2个参数，没有指定参数模式，默认为in。

调用存储过程：

```
/*创建了3个自定义变量*/
SELECT @id:=4,@age:=55,@name:='郭富城';
/*调用存储过程*/
CALL proc3(@id,@age,@name,@user_count,@max_id);
```

验证效果：

```
mysql> select @user_count,@max_id;
+-------------+---------+
| @user_count | @max_id |
+-------------+---------+
|           4 |       4 |
+-------------+---------+
1 row in set (0.00 sec)
```

示例4：带inout参数的存储过程

创建存储过程：

```
/*如果存储过程存在则删除*/
DROP PROCEDURE IF EXISTS proc4;
/*设置结束符为$*/
DELIMITER $
/*创建存储过程proc4*/
CREATE PROCEDURE proc4(INOUT a int,INOUT b int)
  BEGIN
    SET a = a*2;
    select b*2 into b;
  END $

/*将结束符置为;*/
DELIMITER ;
```

调用存储过程：

```
/*创建了2个自定义变量*/
set @a=10,@b:=20;
/*调用存储过程*/
CALL proc4(@a,@b);
```

验证效果：

```
mysql> SELECT @a,@b;
+------+------+
| @a   | @b   |
+------+------+
|   20 |   40 |
+------+------+
1 row in set (0.00 sec)
```

上面的两个自定义变量@a、@b作为入参，然后在存储过程内部进行了修改，又作为了返回值。

示例5：查看存储过程

```
mysql> show create procedure proc4;
+-------+-------+-------+-------+-------+-------+
| Procedure | sql_mode | Create Procedure | character_set_client |
collation_connection | Database Collation |
+-------+-------+-------+-------+-------+-------+
| proc4     |
ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ER
ROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
CREATE DEFINER=`root`@`localhost` PROCEDURE `proc4`(INOUT a int,INOUT
b int)
BEGIN
    SET a = a*2;
    select b*2 into b;
  END | utf8                  | utf8_general_ci      | utf8_general_ci
|
+-------+-------+-------+-------+-------+-------+
1 row in set (0.00 sec)
```

函数

概念

一组预编译好的sql语句集合，理解成批处理语句。类似于java中的方法，但是必须有返回值。

创建函数

```
create function 函数名(参数名称 参数类型)
returns 返回值类型
begin
      函数体
end
```

参数是可选的。

返回值是必须的。

调用函数

```
select 函数名(实参列表);
```

删除函数

```
drop function [if exists] 函数名;
```

查看函数详细

```
show create function 函数名;
```

示例

示例1：无参函数

创建函数：

```
/*删除fun1*/
DROP FUNCTION IF EXISTS fun1;
/*设置结束符为$*/
DELIMITER $
/*创建函数*/
CREATE FUNCTION fun1()
  returns INT
  BEGIN
    DECLARE max_id int DEFAULT 0;
    SELECT max(id) INTO max_id FROM t_user;
    return max_id;
  END $
/*设置结束符为;*/
DELIMITER ;
```

调用看效果：

```
mysql> SELECT fun1();
+--------+
| fun1() |
+--------+
|      4 |
+--------+
1 row in set (0.00 sec)
```

示例2：有参函数

创建函数：

```
/*删除函数*/
DROP FUNCTION IF EXISTS get_user_id;
/*设置结束符为$*/
DELIMITER $
/*创建函数*/
CREATE FUNCTION get_user_id(v_name VARCHAR(16))
  returns INT
  BEGIN
    DECLARE r_id int;
    SELECT id INTO r_id FROM t_user WHERE name = v_name;
    return r_id;
  END $
/*设置结束符为;*/
DELIMITER ;
```

运行看效果：

```
mysql> SELECT get_user_id(name) from t_user;
+-------------------+
| get_user_id(name) |
+-------------------+
|                 1 |
|                 2 |
|                 3 |
|                 4 |
```

```
+--------------------+
4 rows in set (0.00 sec)
```

存储过程和函数的区别

存储过程的关键字为**procedure**，返回值可以有多个，调用时用**call**，一般用于执行比较复杂的的过程体、更新、创建等语句。

函数的关键字为**function**，返回值必须有一个，调用用**select**，一般用于查询单个值并返回。

	存储过程	函数
返回值	0—个或者多个	必须有一个
关键字	procedure	function

调用方式 call select

Mysql系列目录

1. 第1篇：mysql基础知识
2. 第2篇：详解mysql数据类型（重点）
3. 第3篇：管理员必备技能(必须掌握)
4. 第4篇：DDL常见操作
5. 第5篇：DML操作汇总（insert,update,delete）
6. 第6篇：select查询基础篇
7. 第7篇：玩转select条件查询，避免采坑
8. 第8篇：详解排序和分页(order by & limit)
9. 第9篇：分组查询详解（group by & having）
10. 第10篇：常用的几十个函数详解
11. 第11篇：深入了解连接查询及原理
12. 第12篇：子查询
13. 第13篇：细说NULL导致的神坑，让人防不胜防
14. 第14篇：详解事务
15. 第15篇：详解视图

16. **第16篇：变量详解**

第18篇：流程控制语句介绍

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第18篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

上一篇存储过程&自定义函数，对存储过程和自定义函数做了一个简单的介绍，但是如何能够写出复杂的存储过程和函数呢？

这需要我们熟练掌握流程控制语句才可以，本文主要介绍mysql中流程控制语句的使用，上干货。

本篇内容

- if函数
- case语句

- if结构
- while循环
- repeat循环
- loop循环
- 循环体控制语句

准备数据

```
/*建库javacode2018*/
drop database if exists javacode2018;
create database javacode2018;

/*切换到javacode2018库*/
use javacode2018;

/*创建表：t_user*/
DROP TABLE IF EXISTS t_user;
CREATE TABLE t_user(
  id int PRIMARY KEY COMMENT '编号',
  sex TINYINT not null DEFAULT 1 COMMENT '性别,1:男,2:女',
  name VARCHAR(16) not NULL DEFAULT '' COMMENT '姓名'
)COMMENT '用户表';

/*插入数据*/
INSERT INTO t_user VALUES
(1,1,'路人甲Java'),(2,1,'张学友'),(3,2,'王祖贤'),(4,1,'郭富城'),(5,2,'李嘉
欣');

SELECT * FROM t_user;

DROP TABLE IF EXISTS test1;
CREATE TABLE test1 (a int not null);

DROP TABLE IF EXISTS test2;
CREATE TABLE test2 (a int not null,b int NOT NULL );
```

if函数

语法

```
if(条件表达式,值1,值2);
```

if函数有3个参数。

当参数1为true的时候，返回值1，否则返回值2。

示例

需求：查询t_user表数据，返回：编号、性别（男、女）、姓名。

分析一下：数据库中性别用数字表示的，我们需要将其转换为（男、女），可以使用if函数。

```
mysql> SELECT id 编号,if(sex=1,'男','女') 性别,name 姓名 FROM t_user;
+--------+--------+---------------+
| 编号 | 性别 | 姓名     |
+--------+--------+---------------+
|      1 | 男   | 路人甲Java     |
|      2 | 男   | 张学友    |
|      3 | 女   | 王祖贤    |
|      4 | 男   | 郭富城    |
|      5 | 女   | 李嘉欣    |
+--------+--------+---------------+
5 rows in set (0.00 sec)
```

CASE结构

2种用法。

第1种用法

类似于java中的switch语句。

```
case 表达式
when 值1 then 结果1或者语句1（如果是语句需要加分号）
when 值2 then 结果2或者语句2
...
```

else 结果n或者语句n
end [**case**] （如果是放在begin end之间需要加case，如果在select后则不需要）

示例1：*select*中使用

查询t_user表数据，返回：编号、性别（男、女）、姓名。

```
/*写法1：类似于java中的if else*/
SELECT id 编号,(CASE sex WHEN 1 THEN '男' ELSE '女' END) 性别,name 姓名
FROM t_user;
/*写法2：类似于java中的if else if*/
SELECT id 编号,(CASE sex WHEN 1 then '男' WHEN 2 then '女' END) 性
别,name 姓名 FROM t_user;
```

示例2：*begin end*中使用

写一个存储过程，接受3个参数：id，性别（男、女），姓名，然后插入到t_user表

创建存储过程：

```
/*删除存储过程proc1*/
DROP PROCEDURE IF EXISTS proc1;
/*s删除id=6的记录*/
DELETE FROM t_user WHERE id=6;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程proc1*/
CREATE PROCEDURE proc1(id int,sex_str varchar(8),name varchar(16))
  BEGIN
    /*声明变量v_sex用于存放性别*/
    DECLARE v_sex TINYINT UNSIGNED;
    /*根据sex_str的值来设置性别*/
    CASE sex_str
      when '男' THEN
      SET v_sex = 1;
    WHEN '女' THEN
      SET v_sex = 2;
    END CASE ;
```

```
    /*插入数据*/
    INSERT INTO t_user VALUES (id,v_sex,name);
  END $
/*结束符置为;*/
DELIMITER ;
```

调用存储过程：

```
CALL proc1(6,'男','郭富城');
```

查看效果：

```
mysql> select * from t_user;
+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友        |
|  3 |   2 | 王祖贤        |
|  4 |   1 | 郭富城        |
|  5 |   2 | 李嘉欣        |
|  6 |   1 | 郭富城        |
+----+-----+---------------+
6 rows in set (0.00 sec)
```

示例3：函数中使用

需求：写一个函数，根据t_user表sex的值，返回男女

创建函数：

```
/*删除存储过程proc1*/
DROP FUNCTION IF EXISTS fun1;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程proc1*/
CREATE FUNCTION fun1(sex TINYINT UNSIGNED)
  RETURNS varchar(8)
  BEGIN
    /*声明变量v_sex用于存放性别*/
```

```
    DECLARE v_sex VARCHAR(8);
    CASE sex
    WHEN 1 THEN
      SET v_sex:='男';
    ELSE
      SET v_sex:='女';
    END CASE;
    RETURN v_sex;
  END $
/*结束符置为;*/
DELIMITER ;
```

看一下效果：

```
mysql> select sex, fun1(sex) 性别,name FROM t_user;
+-----+--------+---------------+
| sex | 性别 | name          |
+-----+--------+---------------+
|   1 | 男   | 路人甲Java    |
|   1 | 男   | 张学友    |
|   2 | 女   | 王祖贤    |
|   1 | 男   | 郭富城    |
|   2 | 女   | 李嘉欣    |
|   1 | 男   | 郭富城    |
+-----+--------+---------------+
6 rows in set (0.00 sec)
```

第2种用法

类似于java中多重if语句。

```
case
when 条件1 then 结果1或者语句1（如果是语句需要加分号）
when 条件2 then 结果2或者语句2
...
else 结果n或者语句n
```

end [case] （如果是放在begin end之间需要加case，如果是在select后面case可以省略）

这种写法和1中的类似，大家用上面这种语法实现第1中用法中的3个示例，贴在留言中。

if结构

if结构类似于java中的 if..else if...else的语法，如下：

```
if 条件语句1 then 语句1;
elseif 条件语句2 then 语句2;
...
else 语句n;
end if;
```

只能使用在begin end之间。

示例

写一个存储过程，实现用户数据的插入和新增，如果id存在，则修改，不存在则新增，并返回结果

```
/*删除id=7的记录*/
DELETE FROM t_user WHERE id=7;
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc2;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc2(v_id int,v_sex varchar(8),v_name
varchar(16),OUT result TINYINT)
  BEGIN
    DECLARE v_count TINYINT DEFAULT 0;/*用来保存user记录的数量*/
    /*根据v_id查询数据放入v_count中*/
    select count(id) into v_count from t_user where id = v_id;
    /*v_count>0表示数据存在，则修改，否则新增*/
    if v_count>0 THEN
      BEGIN
```

```
        DECLARE lsex TINYINT;
        select if(lsex='男',1,2) into lsex;
        update t_user set sex = lsex,name = v_name where id = v_id;
        /*获取update影响行数*/
        select ROW_COUNT() INTO result;
      END;
    else
      BEGIN
        DECLARE lsex TINYINT;
        select if(lsex='男',1,2) into lsex;
        insert into t_user VALUES (v_id,lsex,v_name);
        select 0 into result;
      END;
    END IF;
  END $
/*结束符置为;*/
DELIMITER ;
```

看效果:

```
mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友    |
|  3 |   2 | 王祖贤    |
|  4 |   1 | 郭富城    |
|  5 |   2 | 李嘉欣    |
|  6 |   1 | 郭富城    |
+----+-----+---------------+
6 rows in set (0.00 sec)

mysql> CALL proc2(7,'男','黎明',@result);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @result;
+---------+
| @result |
```

```
+---------+
|       0 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友    |
|  3 |   2 | 王祖贤    |
|  4 |   1 | 郭富城    |
|  5 |   2 | 李嘉欣    |
|  6 |   1 | 郭富城    |
|  7 |   2 | 黎明    |
+----+-----+---------------+
7 rows in set (0.00 sec)

mysql> CALL proc2(7,'男','梁朝伟',@result);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @result;
+---------+
| @result |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友    |
|  3 |   2 | 王祖贤    |
|  4 |   1 | 郭富城    |
```

```
|   5 |    2 | 李嘉欣    |
|   6 |    1 | 郭富城    |
|   7 |    2 | 梁朝伟    |
+-----+------+---------------+
7 rows in set (0.00 sec)
```

循环

mysql中循环有3种写法

1. while：类似于java中的while循环
2. repeat：类似于java中的do while循环
3. loop：类似于java中的while(true)死循环，需要在内部进行控制。

循环控制

对循环内部的流程进行控制，如：

结束本次循环

类似于java中的`continue`

```
iterate 循环标签;
```

退出循环

类似于java中的`break`

```
leave 循环标签;
```

下面我们分别介绍3种循环的使用。

while循环

类似于java中的while循环。

语法

```
[标签:]while 循环条件 do
循环体
end while [标签];
```

标签：是给while取个名字，标签和iterate、leave结合用于在循环内部对循环进行控制：如：跳出循环、结束本次循环。

注意：这个循环先判断条件，条件成立之后，才会执行循环体，每次执行都会先进行判断。

示例1：无循环控制语句

根据传入的参数v_count向test1表插入指定数量的数据。

```
/*删除test1表记录*/
DELETE FROM test1;
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc3;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc3(v_count int)
  BEGIN
    DECLARE i int DEFAULT 1;
    a:WHILE i<=v_count DO
      INSERT into test1 values (i);
      SET i=i+1;
    END WHILE;
  END $
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> CALL proc3(5);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * from test1;
+---+
```

```
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

示例2：添加leave控制语句

根据传入的参数v_count向test1表插入指定数量的数据，当插入超过10条，结束。

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc4;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc4(v_count int)
  BEGIN
    DECLARE i int DEFAULT 1;
    a:WHILE i<=v_count DO
      INSERT into test1 values (i);
      /*判断i=10，离开循环a*/
      IF i=10 THEN
        LEAVE a;
      END IF;

      SET i=i+1;
    END WHILE;
  END $
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> DELETE FROM test1;
Query OK, 20 rows affected (0.00 sec)
```

```
mysql> CALL proc4(20);
Query OK, 1 row affected (0.02 sec)

mysql> SELECT * from test1;
+----+
| a  |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
|  6 |
|  7 |
|  8 |
|  9 |
| 10 |
+----+
10 rows in set (0.00 sec)
```

示例3：添加iterate控制语句

根据传入的参数v_count向test1表插入指定数量的数据，只插入偶数数据。

```
/*删除test1表记录*/
DELETE FROM test1;
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc5;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc5(v_count int)
  BEGIN
    DECLARE i int DEFAULT 0;
    a:WHILE i<=v_count DO
      SET i=i+1;
      /*如果i不为偶数，跳过本次循环*/
      IF i%2!=0 THEN
        ITERATE a;
      END IF;
```

```
      /*插入数据*/
      INSERT into test1 values (i);
    END WHILE;
  END $
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> DELETE FROM test1;
Query OK, 5 rows affected (0.00 sec)

mysql> CALL proc5(10);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * from test1;
+----+
| a  |
+----+
|  2 |
|  4 |
|  6 |
|  8 |
| 10 |
+----+
5 rows in set (0.00 sec)
```

示例4：嵌套循环

test2表有2个字段（a,b），写一个存储过程（2个参数：v*a*count，v*b*count)，使用双重循环插入数据，数据条件：a的范围[1,v*a*count]、b的范围[1,v*b*count]所有偶数的组合。

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc8;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc8(v_a_count int,v_b_count int)
  BEGIN
```

```
    DECLARE v_a int DEFAULT 0;
    DECLARE v_b int DEFAULT 0;

    a:WHILE v_a<=v_a_count DO
      SET v_a=v_a+1;
      SET v_b=0;

      b:WHILE v_b<=v_b_count DO

        SET v_b=v_b+1;
        IF v_a%2!=0 THEN
          ITERATE a;
        END IF;

        IF v_b%2!=0 THEN
          ITERATE b;
        END IF;

        INSERT INTO test2 VALUES (v_a,v_b);

      END WHILE b;

    END WHILE a;
  END $
/*结束符置为;*/
DELIMITER ;
```

代码中故意将ITERATE a;放在内层循环中，主要让大家看一下效果。

见效果：

```
mysql> DELETE FROM test2;
Query OK, 6 rows affected (0.00 sec)

mysql> CALL proc8(4,6);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * from test2;
+---+---+
| a | b |
```

```
+---+---+
| 2 | 2 |
| 2 | 4 |
| 2 | 6 |
| 4 | 2 |
| 4 | 4 |
| 4 | 6 |
+---+---+
6 rows in set (0.00 sec)
```

repeat循环

语法

```
[标签:]repeat
循环体;
until 结束循环的条件 end repeat [标签];
```

repeat循环类似于java中的do...while循环，不管如何，循环都会先执行一次，然后再判断结束循环的条件，不满足结束条件，循环体继续执行。这块和while不同，while是先判断条件是否成立再执行循环体。

示例1：无循环控制语句

根据传入的参数v_count向test1表插入指定数量的数据。

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc6;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc6(v_count int)
  BEGIN
    DECLARE i int DEFAULT 1;
    a:REPEAT
      INSERT into test1 values (i);
      SET i=i+1;
    UNTIL i>v_count END REPEAT;
  END $
```

```
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> DELETE FROM test1;
Query OK, 1 row affected (0.00 sec)

mysql> CALL proc6(5);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * from test1;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

repeat中`iterate`和`leave`用法和while中类似，这块的示例算是给大家留的作业，写好的发在留言区，谢谢。

loop循环

语法

```
[标签:]loop
循环体;
end loop [标签];
```

> loop相当于一个死循环，需要在循环体中使用`iterate`或者`leave`来控制循环的执行。

示例1：无循环控制语句

根据传入的参数v_count向test1表插入指定数量的数据。

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc7;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc7(v_count int)
  BEGIN
    DECLARE i int DEFAULT 0;
    a:LOOP
      SET i=i+1;
      /*当i>v_count的时候退出循环*/
      IF i>v_count THEN
        LEAVE a;
      END IF;
      INSERT into test1 values (i);
    END LOOP a;
  END $
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> DELETE FROM test1;
Query OK, 5 rows affected (0.00 sec)

mysql> CALL proc7(5);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * from test1;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
```

```
| 5 |
+---+
5 rows in set (0.00 sec)
```

loop中`iterate`和`leave`用法和while中类似，这块的示例算是给大家留的作业，写好的发在留言区，谢谢。

总结

1. 本文主要介绍了mysql中控制流语句的使用，请大家下去了多练习，熟练掌握
2. if函数常用在select中
3. case语句有2种写法，主要用在select、begin end中，select中end后面可以省略case，begin end中使用不能省略case
4. if语句用在begin end中
5. 3种循环体的使用，while类似于java中的while循环，repeat类似于java中的do while循环，loop类似于java中的死循环，都用于begin end中
6. 循环中体中的控制依靠`leave`和`iterate`，`leave`类似于java中的`break`可以退出循环，`iterate`类似于java中的continue可以结束本次循环

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**

7. 第7篇：玩转select条件查询，避免采坑
8. 第8篇：详解排序和分页(order by & limit)
9. 第9篇：分组查询详解（group by & having）
10. 第10篇：常用的几十个函数详解
11. 第11篇：深入了解连接查询及原理
12. 第12篇：子查询
13. 第13篇：细说NULL导致的神坑，让人防不胜防
14. 第14篇：详解事务
15. 第15篇：详解视图
16. 第16篇：变量详解
17. 第17篇：存储过程&自定义函数详解

第19篇：游标详解

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第19篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

需求背景

当我们需要对一个select的查询结果进行遍历处理的时候，如何实现呢?

此时我们需要使用游标，通过游标的方式来遍历select查询的结果集，然后对每行数据进行处理。

本篇内容

- 游标定义
- 游标作用
- 游标使用步骤
- 游标执行过程详解
- 单游标示例
- 嵌套游标示例

准备数据

创建库：javacode2018

创建表：test1、test2、test3

```
/*建库javacode2018*/
drop database if exists javacode2018;
create database javacode2018;

/*切换到javacode2018库*/
```

```
use javacode2018;

DROP TABLE IF EXISTS test1;
CREATE TABLE test1(a int,b int);
INSERT INTO test1 VALUES (1,2),(3,4),(5,6);

DROP TABLE IF EXISTS test2;
CREATE TABLE test2(a int);
INSERT INTO test2 VALUES (100),(200),(300);

DROP TABLE IF EXISTS test3;
CREATE TABLE test3(b int);
INSERT INTO test3 VALUES (400),(500),(600);
```

游标定义

游标（Cursor）是处理数据的一种方法，为了查看或者处理结果集中的数据，游标提供了在结果集中一次一行遍历数据的能力。

游标只能在存储过程和函数中使用。

游标的作用

如sql：

```
select a,b from test1;
```

上面这个查询返回了test1中的数据，如果我们想对这些数据进行遍历处理，此时我们就可以使用游标来进行操作。

游标相当于一个指针，这个指针指向select的第一行数据，可以通过移动指针来遍历后面的数据。

游标的使用步骤

声明游标：这个过程只是创建了一个游标，需要指定这个游标需要遍历的select查询，声明游标时并不会去执行这个sql。

打开游标：打开游标的时候，会执行游标对应的select语句。

遍历数据：使用游标循环遍历select结果中每一行数据，然后进行处理。

关闭游标：游标使用完之后一定要关闭。

游标语法

声明游标

```
DECLARE 游标名称 CURSOR FOR 查询语句;
```

> 一个begin end中只能声明一个游标。

打开游标

```
open 游标名称;
```

遍历游标

```
fetch 游标名称 into 变量列表;
```

> 取出当前行的结果，将结果放在对应的变量中，并将游标指针指向下一行的数据。
>
> 当调用fetch的时候，会获取当前行的数据，如果当前行无数据，会引发mysql内部的`NOT FOUND`错误。

关闭游标

```
close 游标名称;
```

> 游标使用完毕之后一定要关闭。

单游标示例

写一个函数，计算test1表中a、b字段所有的和。

创建函数：

```
/*删除函数*/
DROP FUNCTION IF EXISTS fun1;
/*声明结束符为$*/
DELIMITER $
/*创建函数*/
CREATE FUNCTION fun1(v_max_a int)
  RETURNS int
  BEGIN
    /*用于保存结果*/
    DECLARE v_total int DEFAULT 0;
    /*创建一个变量，用来保存当前行中a的值*/
    DECLARE v_a int DEFAULT 0;
    /*创建一个变量，用来保存当前行中b的值*/
    DECLARE v_b int DEFAULT 0;
    /*创建游标结束标志变量*/
    DECLARE v_done int DEFAULT FALSE;
    /*创建游标*/
    DECLARE cur_test1 CURSOR FOR SELECT a,b from test1 where
a<=v_max_a;
    /*设置游标结束时v_done的值为true，可以v_done来判断游标是否结束了*/
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done=TRUE;
    /*设置v_total初始值*/
    SET v_total = 0;
    /*打开游标*/
    OPEN cur_test1;
    /*使用Loop循环遍历游标*/
    a:LOOP
      /*先获取当前行的数据，然后将当前行的数据放入v_a,v_b中，如果当前行无数据，
v_done会被置为true*/
      FETCH cur_test1 INTO v_a, v_b;
      /*通过v_done来判断游标是否结束了，退出循环*/
      if v_done THEN
```

```
        LEAVE a;
      END IF;
      /*对v_total值累加处理*/
      SET v_total = v_total + v_a + v_b;
    END LOOP;
    /*关闭游标*/
    CLOSE cur_test1;
    /*返回结果*/
    RETURN v_total;
  END $
/*结束符置为;*/
DELIMITER ;
```

上面语句执行过程中可能有问题，解决方式如下。

错误信息：**Mysql 创建函数出现This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA**

This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary

mysql的设置默认是不允许创建函数

解决办法1:

执行：

SET GLOBAL log*bin*trust*function*creators = 1;

不过 重启了 就失效了

注意： 有主从复制的时候 从机必须要设置 不然会导致主从同步失败

解决办法2：

在my.cnf里面设置

log-bin-trust-function-creators=1

不过这个需要重启服务

见效果：

```
mysql> SELECT a,b FROM test1;
+------+------+
| a    | b    |
```

```
+------+------+
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT fun1(1);
+---------+
| fun1(1) |
+---------+
|       3 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT fun1(2);
+---------+
| fun1(2) |
+---------+
|       3 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT fun1(3);
+---------+
| fun1(3) |
+---------+
|      10 |
+---------+
1 row in set (0.00 sec)
```

游标过程详解

以上面的示例代码为例，咱们来看一下游标的详细执行过程。

游标中有个指针，当打开游标的时候，**才会执行游标对应的select语句，这个指针会指向select结果中第一行记录。**

当调用fetch 游标名称时，会获取当前行的数据，如果当前行无数据，会触发NOT FOUND异常。

当触发NOT FOUND异常的时候，我们可以使用一个变量来标记一下，如下代码：

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done=TRUE;
```

当游标无数据触发NOT FOUND异常的时候，将变量v_down的值置为TURE，循环中就可以通过v_down的值控制循环的退出。

如果当前行有数据，则将当前行数据存到对应的变量中，并将游标指针指向下一行数据，如下语句：

```
fetch 游标名称 into 变量列表;
```

嵌套游标

> 写个存储过程，遍历test2、test3，将test2中的a字段和test3中的b字段任意组合，插入到test1表中。

创建存储过程：

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc1;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc1()
  BEGIN
    /*创建一个变量，用来保存当前行中a的值*/
    DECLARE v_a int DEFAULT 0;
    /*创建游标结束标志变量*/
    DECLARE v_done1 int DEFAULT FALSE;
    /*创建游标*/
    DECLARE cur_test1 CURSOR FOR SELECT a FROM test2;
    /*设置游标结束时v_done1的值为true，可以v_done1来判断游标cur_test1是否结束了*/
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done1=TRUE;
```

```
    /*打开游标*/
    OPEN cur_test1;
    /*使用Loop循环遍历游标*/
    a:LOOP
      FETCH cur_test1 INTO v_a;
      /*通过v_done1来判断游标是否结束了，退出循环*/
      if v_done1 THEN
        LEAVE a;
      END IF;

      BEGIN
        /*创建一个变量，用来保存当前行中b的值*/
        DECLARE v_b int DEFAULT 0;
        /*创建游标结束标志变量*/
        DECLARE v_done2 int DEFAULT FALSE;
        /*创建游标*/
        DECLARE cur_test2 CURSOR FOR SELECT b FROM test3;
        /*设置游标结束时v_done1的值为true，可以v_done1来判断游标cur_test2是否
结束了*/
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done2=TRUE;

        /*打开游标*/
        OPEN cur_test2;
        /*使用Loop循环遍历游标*/
        b:LOOP
          FETCH cur_test2 INTO v_b;
          /*通过v_done1来判断游标是否结束了，退出循环*/
          if v_done2 THEN
            LEAVE b;
          END IF;

          /*将v_a、v_b插入test1表中*/
          INSERT INTO test1 VALUES (v_a,v_b);
        END LOOP b;
        /*关闭cur_test2游标*/
        CLOSE cur_test2;
      END;
```

```
    END LOOP;
    /*关闭游标cur_test1*/
    CLOSE cur_test1;
  END $
/*结束符置为;*/
DELIMITER ;
```

见效果：

```
mysql> DELETE FROM test1;
Query OK, 9 rows affected (0.00 sec)

mysql> SELECT * FROM test1;
Empty set (0.00 sec)

mysql> CALL proc1();
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * from test1;
+------+------+
| a    | b    |
+------+------+
|  100 |  400 |
|  100 |  500 |
|  100 |  600 |
|  200 |  400 |
|  200 |  500 |
|  200 |  600 |
|  300 |  400 |
|  300 |  500 |
|  300 |  600 |
+------+------+
9 rows in set (0.00 sec)
```

成功插入了9条数据。

总结

1. 游标用来对查询结果进行遍历处理

2. 游标的使用过程：声明游标、打开游标、遍历游标、关闭游标
3. 游标只能在存储过程和函数中使用
4. 一个begin end中只能声明一个游标
5. 掌握单个游标及嵌套游标的使用
6. 大家下去了多练习一下，熟练掌握游标的使用

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**
11. **第11篇：深入了解连接查询及原理**
12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**
14. **第14篇：详解事务**

15. **第15篇：详解视图**
16. **第16篇：变量详解**
17. **第17篇：存储过程&自定义函数详解**
18. **第18篇：流程控制语句**

第20篇：异常捕获及处理详解

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第20篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

需求背景

我们在写存储过程的时候，可能会出现下列一些情况：

1. 插入的数据违反唯一约束，导致插入失败
2. 插入或者更新数据超过字段最大长度，导致操作失败

3. update影响行数和期望结果不一致

遇到上面各种异常情况的时，可能需要我们能够捕获，然后可能需要回滚当前事务。

本文主要围绕异常处理这块做详细的介绍。

此时我们需要使用游标，通过游标的方式来遍历select查询的结果集，然后对每行数据进行处理。

本篇内容

- 异常分类详解
- 内部异常详解
- 外部异常详解
- 掌握乐观锁解决并发修改数据出错的问题
- update影响行数和期望结果不一致时的处理

准备数据

创建库：javacode2018

创建表：test1，test1表中的a字段为主键。

```
/*建库javacode2018*/
drop database if exists javacode2018;
create database javacode2018;

/*切换到javacode2018库*/
use javacode2018;

DROP TABLE IF EXISTS test1;
CREATE TABLE test1(a int PRIMARY KEY);
```

异常分类

我们将异常分为mysql内部异常和外部异常

mysql内部异常

当我们执行一些sql的时候，可能违反了mysql的一些约束，导致mysql内部报错，如插入数据违反唯一约束，更新数据超时等，此时异常是由mysql内部抛出的，**我们将这些由mysql抛出的异常统称为内部异常**。

外部异常

当我们执行一个update的时候，可能我们期望影响1行，但是实际上影响的不是1行数据，这种情况：sql的执行结果和期望的结果不一致，这种情况也我们也把他作为外部异常处理，**我们将sql执行结果和期望结果不一致的情况统称为外部异常**。

Mysql内部异常

示例1

> test1表中的a字段为主键，我们向test1表同时插入2条数据，并且放在一个事务中执行，最终要么都插入成功，要么都失败。

创建存储过程：

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc1;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc1(a1 int,a2 int)
  BEGIN
    START TRANSACTION;
    INSERT INTO test1(a) VALUES (a1);
    INSERT INTO test1(a) VALUES (a2);
    COMMIT;
  END $
```

```
/*结束符置为;*/
DELIMITER ;
```

上面存储过程插入了两条数据，a的值都是1。

验证结果：

```
mysql> DELETE FROM test1;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL proc1(1,1);
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> SELECT * from test1;
+---+
| a |
+---+
| 1 |
+---+
1 row in set (0.00 sec)
```

上面先删除了test1表中的数据，然后调用存储过程proc1，由于test1表中的a字段是主键，插入第二条数据时违反了a字段的主键约束，mysql内部抛出了异常，导致第二条数据插入失败，最终只有第一条数据插入成功了。

上面的结果和我们期望的不一致，我们希望要么都插入成功，要么失败。

那我们怎么做呢？我们需要捕获上面的主键约束异常，然后发现有异常的时候执行rollback回滚操作，改进上面的代码，看下面示例2。

示例2

我们对上面示例进行改进，捕获上面主键约束异常，然后进行回滚处理，如下：

创建存储过程：

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc2;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc2(a1 int,a2 int)
  BEGIN
```

```
    /*声明一个变量，标识是否有sql异常*/
    DECLARE hasSqlError int DEFAULT FALSE;
    /*在执行过程中出任何异常设置hasSqlError为TRUE*/
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET hasSqlError=TRUE;

    /*开启事务*/
    START TRANSACTION;
    INSERT INTO test1(a) VALUES (a1);
    INSERT INTO test1(a) VALUES (a2);

    /*根据hasSqlError判断是否有异常，做回滚和提交操作*/
    IF hasSqlError THEN
      ROLLBACK;
    ELSE
      COMMIT;
    END IF;
  END $
/*结束符置为;*/
DELIMITER ;
```

上面重点是这句：

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET hasSqlError=TRUE;
```

当有sql异常的时候，会将变量hasSqlError的值置为TRUE。

模拟异常情况：

```
mysql> DELETE FROM test1;
Query OK, 2 rows affected (0.00 sec)

mysql> CALL proc2(1,1);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * from test1;
Empty set (0.00 sec)
```

上面插入了2条一样的数据，插入失败，可以看到上面test1表无数据，和期望结果一致，插入被回滚了。

模拟正常情况：

```
mysql> DELETE FROM test1;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL proc2(1,2);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * from test1;
+---+
| a |
+---+
| 1 |
| 2 |
+---+
2 rows in set (0.00 sec)
```

上面插入了2条不同的数据，最终插入成功。

外部异常

外部异常不是由mysql内部抛出的错误，而是由于sql的执行结果和我们期望的结果不一致的时候，我们需要对这种情况做一些处理，如回滚操作。

示例1

我们来模拟电商中下单操作，按照上面的步骤来更新账户余额。

电商中有个账户表和订单表，如下：

```
DROP TABLE IF EXISTS t_funds;
CREATE TABLE t_funds(
  user_id INT PRIMARY KEY COMMENT '用户id',
  available DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '账户余额'
) COMMENT '用户账户表';
DROP TABLE IF EXISTS t_order;
CREATE TABLE t_order(
  id int PRIMARY KEY AUTO_INCREMENT COMMENT '订单id',
  price DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '订单金额'
```

```
) COMMENT '订单表';
delete from t_funds;
/*插入一条数据，用户id为1001，余额为1000*/
INSERT INTO t_funds (user_id,available) VALUES (1001,1000);
```

下单操作涉及到操作上面的账户表，我们用存储过程来模拟实现：

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc3;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc3(v_user_id int,v_price decimal(10,2),OUT v_msg
varchar(64))
  a:BEGIN
    DECLARE v_available DECIMAL(10,2);

    /*1.查询余额，判断余额是否够*/
    select a.available into v_available from t_funds a where a.user_id
= v_user_id;
    if v_available<=v_price THEN
      SET v_msg='账户余额不足!';
      /*退出*/
      LEAVE a;
    END IF;

    /*模拟耗时5秒*/
    SELECT sleep(5);

    /*2.余额减去price*/
    SET v_available = v_available - v_price;

    /*3.更新余额*/
    START TRANSACTION;
    UPDATE t_funds SET available = v_available WHERE user_id =
v_user_id;

    /*插入订单明细*/
    INSERT INTO t_order (price) VALUES (v_price);
```

```
    /*提交事务*/
    COMMIT;
    SET v_msg='下单成功!';
  END $
/*结束符置为;*/
DELIMITER ;
```

上面过程主要分为3步骤：验证余额、修改余额变量、更新余额。

开启2个cmd窗口，连接mysql，同时执行下面操作：

```
USE javacode2018;
CALL proc3(1001,100,@v_msg);
select @v_msg;
```

然后执行：

```
mysql> SELECT * FROM t_funds;
+---------+-----------+
| user_id | available |
+---------+-----------+
|    1001 |    900.00 |
+---------+-----------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t_order;
+----+--------+
| id | price  |
+----+--------+
|  1 | 100.00 |
|  2 | 100.00 |
+----+--------+
2 rows in set (0.00 sec)
```

上面出现了非常严重的错误：下单成功了2次，但是账户只扣了100。

上面过程是由于2个操作并发导致的，2个窗口同时执行第一步的时候看到了一样的数据（看到的余额都是1000），然后继续向下执行，最终导致结果出问题了。

上面操作我们可以使用乐观锁来优化。

乐观锁的过程：用期望的值和目标值进行比较，如果相同，则更新目标值，否则什么也不做。

乐观锁类似于java中的cas操作，这块需要了解的可以点击：详解CAS

我们可以在资金表t_funds添加一个version字段，表示版本号，每次更新数据的时候+1，更新数据的时候将version作为条件去执行update，根据update影响行数来判断执行是否成功，优化上面的代码，见**示例2**。

示例2

对示例1进行优化。

创建表：

```
DROP TABLE IF EXISTS t_funds;
CREATE TABLE t_funds(
  user_id INT PRIMARY KEY COMMENT '用户id',
  available DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '账户余额',
  version INT DEFAULT 0 COMMENT '版本号，每次更新+1'
) COMMENT '用户账户表';

DROP TABLE IF EXISTS t_order;
CREATE TABLE t_order(
  id int PRIMARY KEY AUTO_INCREMENT COMMENT '订单id',
  price DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '订单金额'
)COMMENT '订单表';
delete from t_funds;
/*插入一条数据，用户id为1001，余额为1000*/
INSERT INTO t_funds (user_id,available) VALUES (1001,1000);
```

创建存储过程：

```
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc4;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc4(v_user_id int,v_price decimal(10,2),OUT v_msg
```

```
varchar(64))
    a:BEGIN
    /*保存当前余额*/
    DECLARE v_available DECIMAL(10,2);
    /*保存版本号*/
    DECLARE v_version INT DEFAULT 0;
    /*保存影响的行数*/
    DECLARE v_update_count INT DEFAULT 0;


    /*1.查询余额，判断余额是否够*/
    select a.available,a.version into v_available,v_version from
t_funds a where a.user_id = v_user_id;
    if v_available<=v_price THEN
      SET v_msg='账户余额不足!';
      /*退出*/
      LEAVE a;
    END IF;

    /*模拟耗时5秒*/
    SELECT sleep(5);

    /*2.余额减去price*/
    SET v_available = v_available - v_price;

    /*3.更新余额*/
    START TRANSACTION;
    UPDATE t_funds SET available = v_available WHERE user_id =
v_user_id AND version = v_version;
    /*获取上面update影响行数*/
    select ROW_COUNT() INTO v_update_count;

    IF v_update_count=1 THEN
      /*插入订单明细*/
      INSERT INTO t_order (price) VALUES (v_price);
      SET v_msg='下单成功!';
      /*提交事务*/
      COMMIT;
```

```
    ELSE
      SET v_msg='下单失败,请重试!';
      /*回滚事务*/
      ROLLBACK;
    END IF;
  END $
/*结束符置为;*/
DELIMITER ;
```

ROW_COUNT()可以获取更新或插入后获取受影响行数。将受影响行数放在v_update_count中。

然后根据v_update_count是否等于1判断更新是否成功，如果成功则记录订单信息并提交事务，否则回滚事务。

验证结果：开启2个cmd窗口，连接mysql，执行下面操作：

```
use javacode2018;
CALL proc4(1001,100,@v_msg);
select @v_msg;
```

窗口1结果：

```
mysql> CALL proc4(1001,100,@v_msg);
+----------+
| sleep(5) |
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)

Query OK, 0 rows affected (5.00 sec)

mysql> select @v_msg;
+---------------+
| @v_msg        |
+---------------+
| 下单成功!     |
+---------------+
1 row in set (0.00 sec)
```

窗口2结果：

```
mysql> CALL proc4(1001,100,@v_msg);
+----------+
| sleep(5) |
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)

Query OK, 0 rows affected (5.01 sec)

mysql> select @v_msg;
+-------------------------+
| @v_msg                  |
+-------------------------+
| 下单失败,请重试!        |
+-------------------------+
1 row in set (0.00 sec)
```

可以看到第一个窗口下单成功了，窗口2下单失败了。

再看一下2个表的数据：

```
mysql> SELECT * FROM t_funds;
+---------+-----------+---------+
| user_id | available | version |
+---------+-----------+---------+
|    1001 |    900.00 |       0 |
+---------+-----------+---------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t_order;
+----+--------+
| id | price  |
+----+--------+
|  1 | 100.00 |
+----+--------+
1 row in set (0.00 sec)
```

也正常。

总结

1. 异常分为Mysql内部异常和外部异常
2. 内部异常由mysql内部触发，外部异常是sql的执行结果和期望结果不一致导致的错误
3. sql内部异常捕获方式

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET hasSqlError=TRUE;
```

4. `ROW_COUNT()`可以获取mysql中insert或者update影响的行数
5. 掌握使用乐观锁（添加版本号）来解决并发修改数据可能出错的问题
6. `begin end`前面可以加标签，`LEAVE` 标签可以退出对应的begin end，可以使用这个来实现return的效果

Mysql系列目录

1. 第**1**篇：**mysql**基础知识
2. 第**2**篇：详解**mysql**数据类型（重点）
3. 第**3**篇：管理员必备技能**(**必须掌握**)**
4. 第**4**篇：**DDL**常见操作
5. 第**5**篇：**DML**操作汇总（**insert,update,delete**）
6. 第**6**篇：**select**查询基础篇
7. 第**7**篇：玩转**select**条件查询，避免采坑
8. 第**8**篇：详解排序和分页**(order by & limit)**
9. 第**9**篇：分组查询详解（**group by & having**）
10. 第**10**篇：常用的几十个函数详解

11. **第11篇：深入了解连接查询及原理**
12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**
14. **第14篇：详解事务**
15. **第15篇：详解视图**
16. **第16篇：变量详解**
17. **第17篇：存储过程&自定义函数详解**
18. **第18篇：流程控制语句**
19. **第19篇：游标详解**

第21篇：什么是索引?

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

。

这是Mysql系列第21篇。

本文开始连续**3**篇详解**mysql**索引：

1. 第1篇来说说什么是索引?
2. 第2篇详解Mysql中索引的原理
3. 第3篇结合索引详解关键字explain

本文为索引第一篇：我们来了解一下什么是索引?

来看一个问题

路人在搞计算机之前，是负责小区建设规划的，上级领导安排路人负责一个万人小区建设规划，并提了一个要求：可以快速通过户主姓名找到户主的房子；让路人出个好的解决方案。

方案1

刚开始路人没什么经验，实在想不到什么好办法。

路人告诉领导：你可以去敲每户的门，然后开门之后再去询问房主姓名，是否和需要找的人姓名一致。

领导一听郁闷了：我敲你的头，1万户，我一个个找，找到什么时候了？你明天不用来上班了。

> 这里面涉及到的时间有：走到每户的门口耗时、敲门等待开门耗时、询问户主获取户主姓名耗时、将户主姓名和需要查找的姓名对比是否一致耗时。
>
> 加入要找的人刚好在最后一户，领导岂不是要疯掉了，需要重复1万次上面的操作。

上面是最原始，最耗时的做法，可能要找的人根本不在这个小区，白费力的找了1万次，岂不是要疯掉。

方案2

路人灵机一动，想到了一个方案：

1. 给所有的户主制定一个编号，从1-10000，户主将户号贴在自家的门口
2. 路人自己制作了一个户主和户号对应的表格，我们叫做：户主目录表，共1万条记录，如下：

户主姓名	房屋编号
刘德华	00001
张学友	00002
路人	00888
路人甲java	10000

此时领导要查找路人甲Java时，过程如下：

1. 按照姓名在户主目录表查找路人甲Java，找到对应的编号：10000
2. 然后从第一户房子开始找，查看其门口户号是否是10000，直到找到为止

路人告诉领导，这个方案比方案1有以下好处：

1. 如果要找的人不在这个小区，通过户主目录表就确定，不需要第二步了
2. 步骤2中不需要再去敲每户的门以及询问户主的姓名了，只需对比一下门口的户号就可以了，比方案1省了不少时间。

领导笑着说，不错不错，有进步，不过我找路人甲Java还是需要挨家挨户看门牌号1万次啊！。。。。。你再去想想吧，看看是否还有更好的办法来加快查找速度。

路人下去了苦思冥想，想出了方案3。

方案3

方案2中第2步最坏的情况还是需要找1万次。

路人去上海走了一圈，看了那边小区搞的不错，很多小区都是搞成一栋一栋的，每栋楼里面有100户，路人也决定这么搞。

路人告诉领导：

1. 将1万户划分为100栋楼，每栋楼有25层，每层有4户人家，总共1万户
2. 给每栋楼一个编号，范围是[001,100]，将栋号贴在每栋楼最显眼的位置
3. 给每栋楼中的每层一个编号，编号范围是[01,25]，将层号贴在每层楼最显眼的位置
4. 户号变为：栋号-楼层-层中编号，如路人甲Java户号是：100-20-04，贴在每户门口

户主目录表还是有1万条记录，如下：

房屋编号	户主姓名	人口
001-08-04	刘德华	4
022-18-01	张学友	1
088-25-04	郭富城	3

路人甲java 100-25-04

此时领导要查找路人甲Java时，过程如下：

1. 按照姓名在户主目录表查找路人甲Java，找到对应的编号是100-25-04，将编号分解，得到：栋号（100）、楼层（25）、楼号（04）
2. 从第一栋开始找，看其栋号是否是100，直到找到编号为100为止，这个过程需要找100次，然后到了第100栋楼下
3. 从100栋的第一层开始向上走，走到每层看其编号是否为25，直到走到第25层，这个过程需要匹配25次
4. 在第25层依次看看户号是否为100-25-04，匹配了4次，找到了路人甲Java

此方案分析：

1. 查找户主目录表1万次，不过这个是在表格中，不用动身走路去找，只需要动动眼睛对比一下数字，速度还是比较快的
2. 将方案2中的第2步优化为上面的2/3/4步骤，上面最坏需要匹配129次（栋100+层25+楼号4次），相对于方案2的1万次好多了

领导拍拍路人的肩膀：小伙子，去过上海的人确实不一样啊，这次方案不错，不过第一步还是需要很多次，能否有更好的方案呢?

路人下去了又想了好几天，突然想到了我们常用的字典，可以按照字典的方式对方案3中第一步做优化，然后提出了方案4。

方案4

对户主表进行改造，按照姓的首字母(a-z)制作26个表格，叫做：姓氏户主表，每个表格中保存对应姓氏首字母及所有户号。如下：

姓首字母：A

姓名	户号
阿三	010-16-01
阿郎	017-11-04

路人057-111-04

路人甲Java048-08-02

现在查找户号步骤如下：

1. 通过姓名获取姓对应的首字母
2. 在26个表格中找到对应姓的表格，如路人甲Java，对应L表
3. 在L表中循环遍历，找到路人甲Java的户号
4. 根据户号按照方案3中的(2/3/4)步骤找对应的户主

理想情况：

1万户主的姓氏分配比较均衡，那么每个姓氏下面分配385户（10000/26），那么找到某个户主，最多需要:26次+385次 = 410次，相对于1万次少了很多。

最坏的情况：

1万个户主的姓氏都是一样的，导致这1万个户主信息都位于同一个姓氏户主表，此时查询又变为了1万多次。不过出现姓氏一样的情况比较低。

如果担心姓氏不足以均衡划分户主信息，那么也可以通过户主姓名的笔画数来划分，或者其他方法，主要是将用户信息划分为不同的区，可以快速过滤一些不相关的户主。

上面几个方案为了快速检索到户主，用到了一些数据结构，通过这些数据结构对户主的信息进行组织，从而可以快速过滤掉一些不相关的户主，减少查找次数，快速定位到户主的房子。

索引是什么?

通过上面的示例，我们可以概况一下索引的定义：索引是依靠某些数据结构和算法来组织数据，最终引导用户快速检索出所需要的数据。

索引有2个特点：

1. 通过数据结构和算法来对原始的数据进行一些有效的组织
2. 通过这些有效的组织，可以引导使用者对原始数据进行快速检索

mysql为了快速检索数据，也用到了一些好的数据结构和算法，来组织表中的数据，加快检索效率。

下篇文章将对mysql索引原理做详细介绍，敬请期待，喜欢的关注一下谢谢!

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**

6. 第**6**篇：**select**查询基础篇
7. 第**7**篇：玩转**select**条件查询，避免采坑
8. 第**8**篇：详解排序和分页**(order by & limit)**
9. 第**9**篇：分组查询详解（**group by & having**）
10. 第**10**篇：常用的几十个函数详解
11. 第**11**篇：深入了解连接查询及原理
12. 第**12**篇：子查询
13. 第**13**篇：细说**NULL**导致的神坑，让人防不胜防
14. 第**14**篇：详解事务
15. 第**15**篇：详解视图
16. 第**16**篇：变量详解
17. 第**17**篇：存储过程**&**自定义函数详解
18. 第**18**篇：流程控制语句
19. 第**19**篇：游标详解
20. 第**20**篇：异常捕获及处理详解

第22篇：MySQL索引原理详解

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第22篇。

背景

使用mysql最多的就是查询，我们迫切的希望mysql能查询的更快一些，我们经常用到的查询有：

1. 按照id查询唯一一条记录
2. 按照某些个字段查询对应的记录
3. 查找某个范围的所有记录（between and）
4. 对查询出来的结果排序

mysql的索引的目的是使上面的各种查询能够更快。

预备知识

什么是索引?

上一篇中有详细的介绍，可以过去看一下：什么是索引？

索引的本质：**通过不断地缩小想要获取数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是说，有了这种索引机制，我们可以总是用同一种查找方式来锁定数据。**

磁盘中数据的存取

以机械硬盘来说，先了解几个概念。

扇区：磁盘存储的最小单位，扇区一般大小为512Byte。

磁盘块：文件系统与磁盘交互的的最小单位（计算机系统读写磁盘的最小单位），一个磁盘块由连续几个（2^n）扇区组成，块一般大小一般为4KB。

磁盘读取数据：磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为**寻道时间、旋转延迟、传输时间**三个部分，寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下；旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms；传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。

mysql中的页

mysql中和磁盘交互的最小单位称为页，页是mysql内部定义的一种数据结构，默认为16kb，相当于4个磁盘块，也就是说mysql每次从磁盘中读取一次数据是16KB，要么不读取，要读取就是16KB，此值可以修改的。

数据检索过程

我们对数据存储方式不做任何优化，直接将数据库中表的记录存储在磁盘中，假如某个表只有一个字段，为int类型，int占用4个byte，每个磁盘块可以存储1000条记录，100万的记录需要1000个磁盘块，如果我们需要从这100万记录中检索所需要的记录，需要读取1000个磁盘块的数据（需要1000次io），每次io需要9ms，那么1000次需要9000ms=9s，100条数据随便一个查询就是9秒，这种情况我们是无法接受的，显然是不行的。

我们迫切的需求是什么?

我们迫切需要这样的数据结构和算法:

1. 需要一种数据存储结构:当从磁盘中检索数据的时候能,够减少磁盘的io次数,最好能够降低到一个稳定的常量值
2. 需要一种检索算法:当从磁盘中读取磁盘块的数据之后,这些块中可能包含多条记录,这些记录被加载到内存中,那么需要一种算法能够快速从内存多条记录中快速检索出目标数据

我们来找找,看是否能够找到这样的算法和数据结构。

我们看一下常见的检索算法和数据结构。

循环遍历查找

从一组无序的数据中查找目标数据,常见的方法是遍历查询,n条数据,时间复杂度为O(n),最快需要1次,最坏的情况需要n次,查询效率不稳定。

二分法查找

二分法查找也称为折半查找,用于在一个有序数组中快速定义某一个需要查找的数据。

原理是:

先将一组无序的数据排序(升序或者降序)之后放在数组中,此处用升序来举例说明:用数组中间位置的数据A和需要查找的数据F对比,如果A=F,则结束查找;如果A<F,则将查找的范围缩小至数组中A数据右边的部分;如果A>F,则将查找范围缩小至数组中A数据左边的部分,继续按照上面的方法直到找到F为止。

示例:

从下列有序数字中查找数字9，过程如下

[1,2,3,4,5,6,7,8,9]

第1次查找：[1,2,3,4,5,6,7,8,9]中间位置值为5，9>5，将查找范围缩小至5右边的部分：[6、7、8、9]

第2次查找：[6、7、8、9]中间值为8，9>8 ，将范围缩小至8右边部分：[9]

第3次查找：在[9]中查找9，找到了。

可以看到查找速度是相当快的，每次查找都会使范围减半，如果我们采用顺序查找，上面数据最快需要1次，最多需要9次，而二分法查找最多只需要3次，耗时时间也比较稳定。

二分法查找时间复杂度是:**O(logN)(N为数据量)**，100万数据查找最多只需要20次（2^{20} =1048576）

二分法查找数据的优点：定位数据非常快，前提是：目标数组是有序的。

有序数组

如果我们将mysql中表的数据以有序数组的方式存储在磁盘中，那么我们定位数据步骤是：

1. 取出目标表的所有数据，存放在一个有序数组中
2. 如果目标表的数据量非常大，从磁盘中加载到内存中需要的内存也非常大

步骤取出所有数据耗费的io次数太多，步骤2耗费的内存空间太大，还有新增数据的时候，为了保证数组有序，插入数据会涉及到数组内部数据的移动，也是比较耗时的，显然用这种方式存储数据是不可取的。

链表

链表相当于在每个节点上增加一些指针，可以和前面或者后面的节点连接起来，就像一列火车一样，每节车厢相当于一个节点，车厢内部可以存储数据，每个车厢和下一节车厢相连。

链表分为单链表和双向链表。

单链表

每个节点中有持有指向下一个节点的指针，只能按照一个方向遍历链表，结构如下：

```
//单项链表
class Node1{
    private Object data;//存储数据
    private Node1 nextNode;//指向下一个节点
}
```

双向链表

每个节点中两个指针，分别指向当前节点的上一个节点和下一个节点，结构如下：

```
//双向链表
class Node2{
    private Object data;//存储数据
    private Node1 prevNode;//指向上一个节点
    private Node1 nextNode;//指向下一个节点
}
```

链表的优点：

1. 可以快速定位到上一个或者下一个节点
2. 可以快速删除数据，只需改变指针的指向即可，这点比数组好

链表的缺点：

1. 无法向数组那样，通过下标随机访问数据

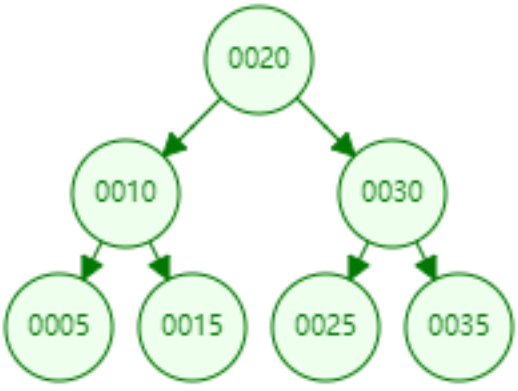
2. **查找数据需从第一个节点开始遍历，不利于数据的查找，查找时间和无需数据类似，需要全遍历，最差时间是O(N)**

二叉查找树

二叉树是每个结点最多有两个子树的树结构，通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。二叉树有如下特性：

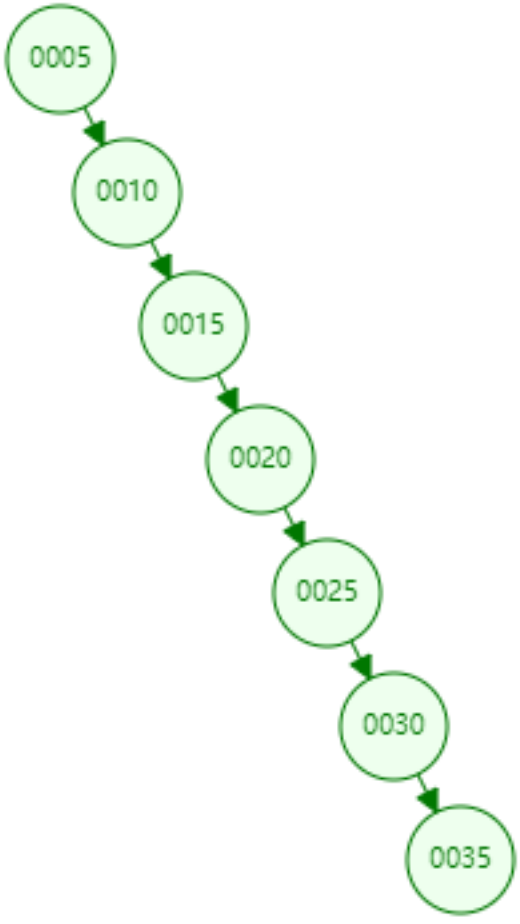
1、每个结点都包含一个元素以及n个子树，这里0≤n≤2。 2、左子树和右子树是有顺序的，次序不能任意颠倒，左子树的值要小于父结点，右子树的值要大于父结点。

数组[20,10,5,15,30,25,35]使用二叉查找树存储如下：



每个节点上面有两个指针（left,rigth），可以通过这2个指针快速访问左右子节点，检索任何一个数据最多只需要访问3个节点，相当于访问了3次数据，时间为O(logN)，和二分法查找效率一样，查询数据还是比较快的。

但是如果我们插入数据是有序的，如[5,10,15,20,30,25,35]，那么结构就变成下面这样：



二叉树退化为了一个链表结构，查询数据最差就变为了O(N)。

二叉树的优缺点：

1. **查询数据的效率不稳定，若树左右比较平衡的时，最差情况为O(logN)，如果插入数据是有序的，退化为了链表，查询时间变成了O(N)**

2. **数据量大的情况下，会导致树的高度变高，如果每个节点对应磁盘的一个块来存储一条数据，需io次数大幅增加，显然用此结构来存储数据是不可取的**

平衡二叉树（AVL树）

平衡二叉树是一种特殊的二叉树，所以他也满足前面说到的二叉查找树的两个特性，同时还有一个特性：

> 它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

平衡二叉树相对于二叉树来说，树的左右比较平衡，不会出现二叉树那样退化成链表的情况，不管怎么插入数据，最终通过一些调整，都能够保证树左右高度相差不大于1。

这样可以让查询速度比较稳定，查询中遍历节点控制在O(logN)范围内

如果数据都存储在内存中，采用AVL树来存储，还是可以的，查询效率非常高。不过我们的数据是存在磁盘中，用过采用这种结构，每个节点对应一个磁盘块，数据量大的时候，也会和二叉树一样，会导致树的高度变高，增加了io次数，显然用这种结构存储数据也是不可取的。

B-树

B杠树，千万不要读作B减树了，B-树在是平衡二叉树上进化来的，前面介绍的几种树，每个节点上面只有一个元素，而B-树节点中可以放多个元素，主要是为了降低树的高度。

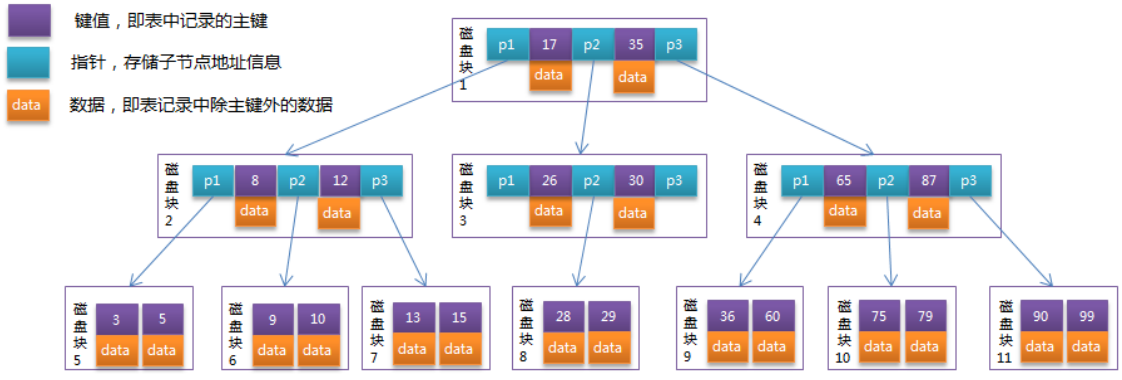
一棵m阶的B-Tree有如下特性【特征描述的有点绕，看不懂的可以跳过，看后面的图】：

1. 每个节点最多有m个孩子，m称为b树的阶
2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子
3. 若根节点不是叶子节点，则至少有2个孩子
4. 所有叶子节点都在同一层，且不包含其它关键字信息
5. 每个非终端节点包含n个关键字（健值）信息

6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1
7. ki(i=1,...n)为关键字，且关键字升序排序
8. Pi(i=1,...n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data]，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个键将数据划分成的三个范围域，对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2

3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】

4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2

5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】

6. 在磁盘块8中的关键字列表中找到关键字29

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作，由于内存中的关键字是一个有序表结构，可以利用二分法快速定位到目标数据，而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。

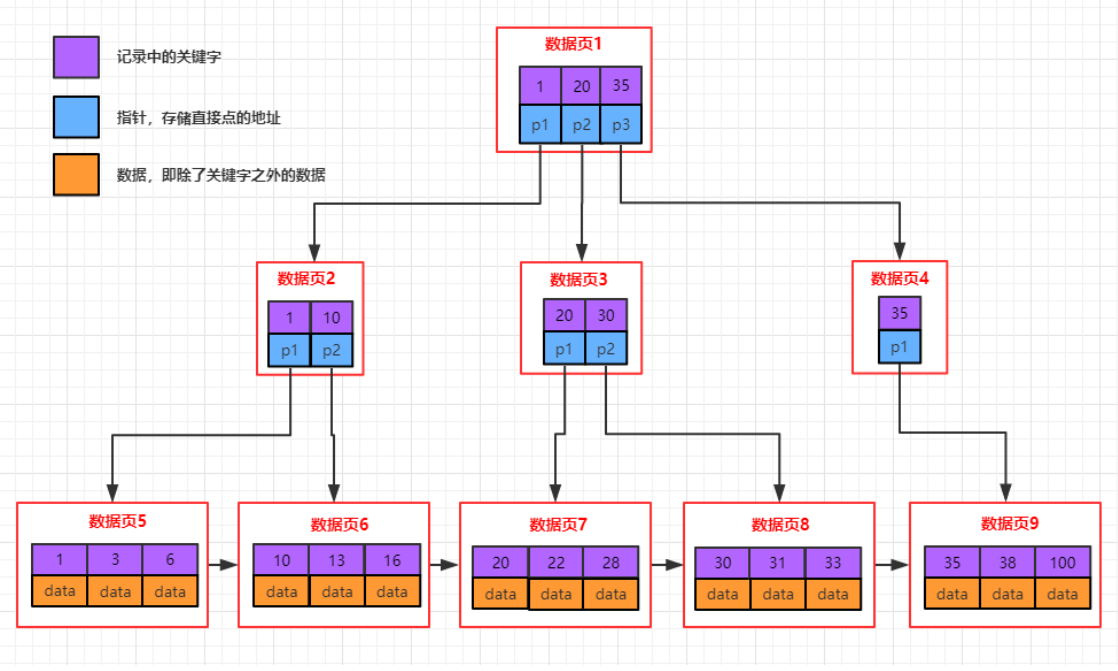
B-树相对于avl树，通过在节点中增加节点内部数据的个数来减少磁盘的io操作。

上面我们说过mysql是采用页方式来读写数据，每页是16KB，我们用B-树来存储mysql的记录，每个节点对应mysql中的一页（16KB），假如每行记录加上树节点中的1个指针占160Byte，那么每个节点可以存储1000（16KB/160byte）条数据，树的高度为3的节点大概可以存储（第一层1000+第二层1000^2+第三层1000^3）10亿条记录，是不是非常惊讶，一个高度为3个B-树大概可以存储10亿条记录，我们从10亿记录中查找数据只需要3次io操作可以定位到目标数据所在的页，而页内部的数据又是有序的，然后将其加载到内存中用二分法查找，是非常快的。

可以看出使用B-树定位某个值还是很快的(10亿数据中3次io操作+内存中二分法)，但是也是有缺点的：**B-不利于范围查找，比如上图中我们需要查找[15,36]区间的数据，需要访问7个磁盘块（1/2/7/3/8/4/9），io次数又上去了，范围查找也是我们经常用到的，所以b-树也不太适合在磁盘中存储需要检索的数据**。

b+树

先看个b+树结构图：



b+树的特征

1. 每个结点至多有m个子女
2. 除根结点外,每个结点至少有[m/2]个子女，根结点至少有两个子女
3. 有k个子女的结点必有k个关键字
4. 父节点中持有访问子节点的指针
5. 父节点的关键字在子节点中都存在（如上面的1/20/35在每层都存在），要么是最小值，要么是最大值，如果节点中关键字是升序的方式，父节点的关键字是子节点的最小值
6. 最底层的节点是叶子节点
7. 除叶子节点之外，其他节点不保存数据，只保存关键字和指针

8. 叶子节点包含了所有数据的关键字以及data，叶子节点之间用链表连接起来，可以非常方便的支持范围查找

b+树与b-树的几点不同

1. b+树中一个节点如果有k个关键字，最多可以包含k个子节点（k个关键字对应k个指针）；而b-树对应k+1个子节点（多了一个指向子节点的指针）

2. b+树除叶子节点之外其他节点值存储关键字和指向子节点的指针，而b-树还存储了数据，这样同样大小情况下，b+树可以存储更多的关键字

3. b+树叶子节点中存储了所有关键字及data，并且多个节点用链表连接，从上图中看子节点中数据从左向右是有序的，这样快速可以支撑范围查找（先定位范围的最大值和最小值，然后子节点中依靠链表遍历范围数据）

B-Tree和B+Tree该如何选择?

1. B-Tree因为非叶子结点也保存具体数据，所以在查找某个关键字的时候找到即可返回。而B+Tree所有的数据都在叶子结点，每次查找都得到叶子结点。所以在同样高度的B-Tree和B+Tree中，B-Tree查找某个关键字的效率更高。

2. 由于B+Tree所有的数据都在叶子结点，并且结点之间有指针连接，在找大于某个关键字或者小于某个关键字的数据的时候，B+Tree只需要找到该关键字然后沿着链表遍历就可以了，而B-Tree还需要遍历该关键字结点的根结点去搜索。

3. 由于B-Tree的每个结点（这里的结点可以理解为一个数据页）都存储主键+实际数据，而B+Tree非叶子结点只存储关键字信息，而每个页的大小有限是有限的，所以同一页能存储的B-Tree的数据会比B+Tree存储的更少。这样同样总量的数据，B-Tree的深度会更大，增大查询时的磁盘I/O次数，进而影响查询效率。

Mysql的存储引擎和索引

mysql内部索引是由不同的引擎实现的，主要说一下InnoDB和MyISAM这两种引擎中的索引，这两种引擎中的索引都是使用b+树的结构来存储的。

InnoDB中的索引

Innodb中有2种索引：**主键索引（聚集索引）**、**辅助索引（非聚集索引）**。

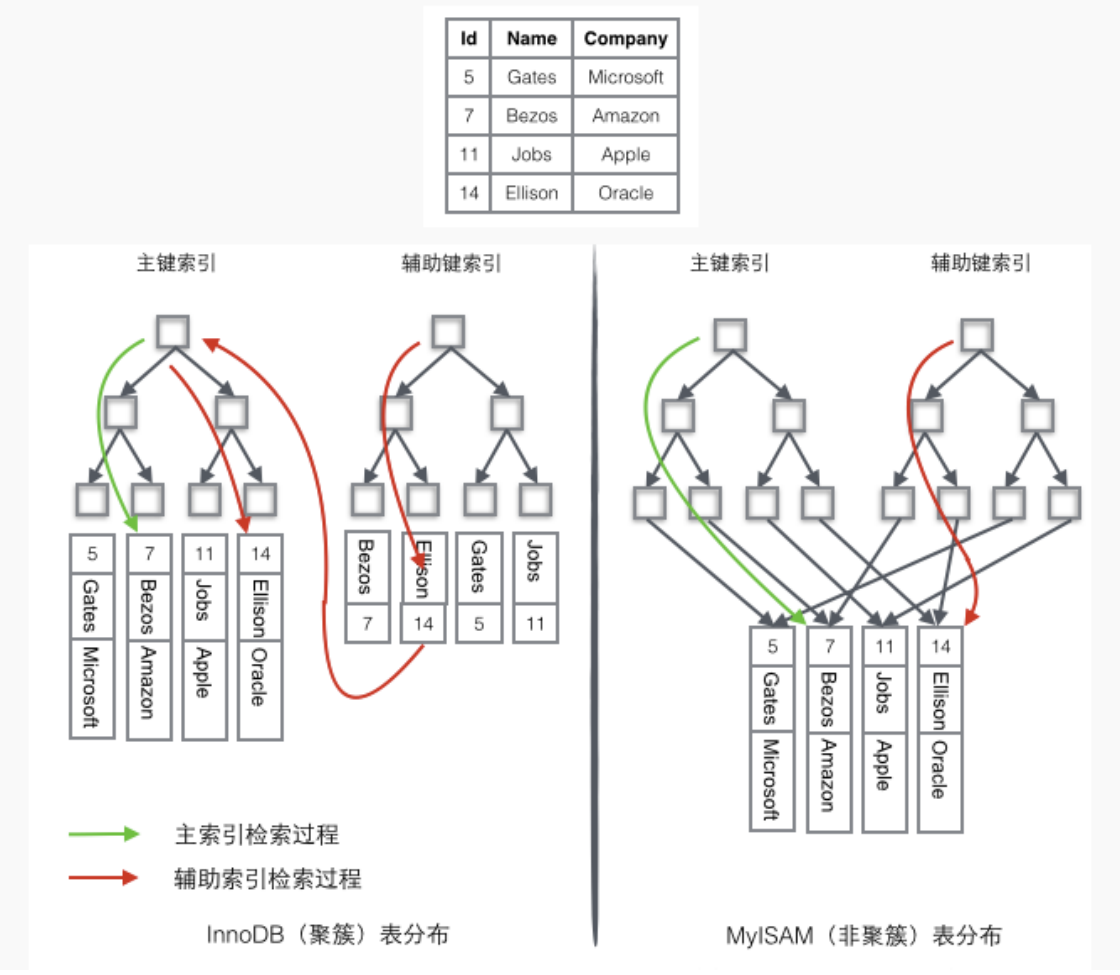
主键索引：每个表只有一个主键索引，b+树结构，叶子节点同时保存了主键的值也数据记录，其他节点只存储主键的值。

辅助索引：每个表可以有多个，b+树结构，叶子节点保存了索引字段的值以及主键的值，其他节点只存储索引指端的值。

MyISAM引擎中的索引

B+树结构，MyISM使用的是非聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

如下图：为了更形象说明这两种索引的区别，我们假想一个表存储了4行数据。其中Id作为主索引，Name作为辅助索引，图中清晰的显示了聚簇索引和非聚簇索引的差异。



我们看一下上图中数据检索过程。

InnoDB数据检索过程

如果需要查询id=14的数据，只需要在左边的主键索引中检索就可以了。

如果需要搜索name='Ellison'的数据，需要2步：

1. 先在辅助索引中检索到name='Ellison'的数据，获取id为14
2. 再到主键索引中检索id为14的记录

辅助索引这个查询过程在mysql中叫做回表。

MyISAM数据检索过程

1. 在索引中找到对应的关键字，获取关键字对应的记录的地址
2. 通过记录的地址查找到对应的数据记录

我们用的最多的是innodb存储引擎，所以此处主要说一下innodb索引的情况，innodb中最好是采用主键查询，这样只需要一次索引，如果使用辅助索引检索，涉及到回表操作，比主键查询要耗时一些。

innodb中辅助索引为什么不像myisam那样存储记录的地址?

表中的数据发生变更的时候，会影响其他记录地址的变化，如果辅助索引中记录数据的地址，此时会受影响，而主键的值一般是很少更新的，当页中的记录发生地址变更的时候，对辅助索引是没有影响的。

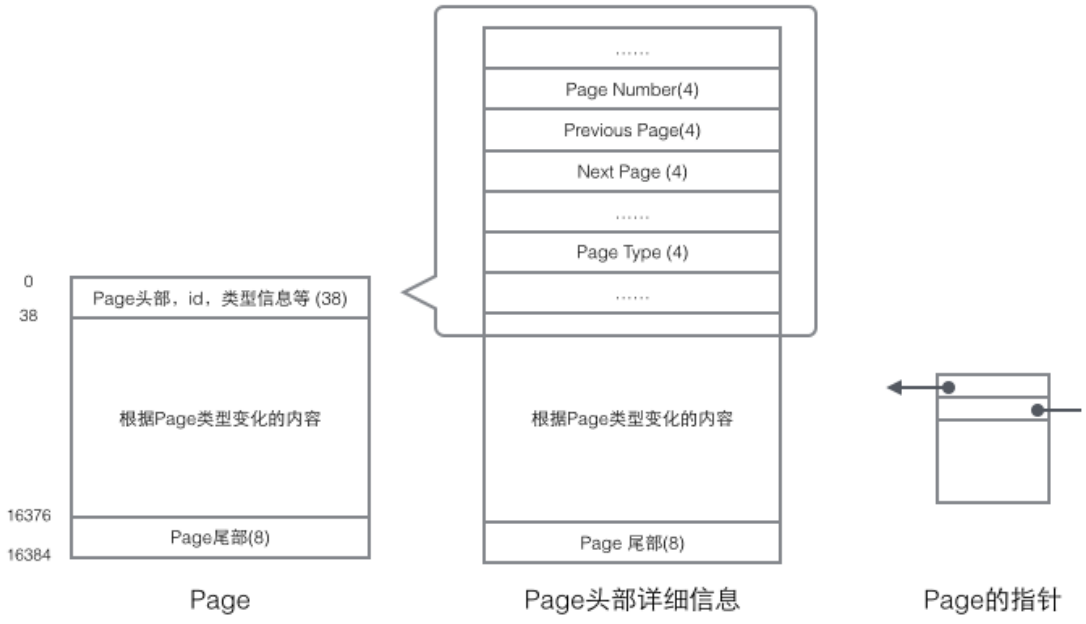
我们来看一下mysql中页的结构，页是真正存储记录的地方，对应B+树中的一个节点，也是mysql中读写数据的最小单位，页的结构设计也是相当有水平的，能够加快数据的查询。

页结构

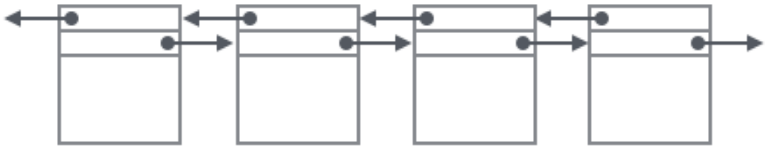
mysql中页是innodb中存储数据的基本单位，也是mysql中管理数据的最小单位，和磁盘交互的时候都是以页来进行的，默认是16kb，mysql中采用b+树存储数据，页相当于b+树中的一个节点。

页的结构如下图:

每个Page都有通用的头和尾，但是中部的内容根据Page的类型不同而发生变化。Page的头部里有我们关心的一些数据，下图把Page的头部详细信息显示出来：



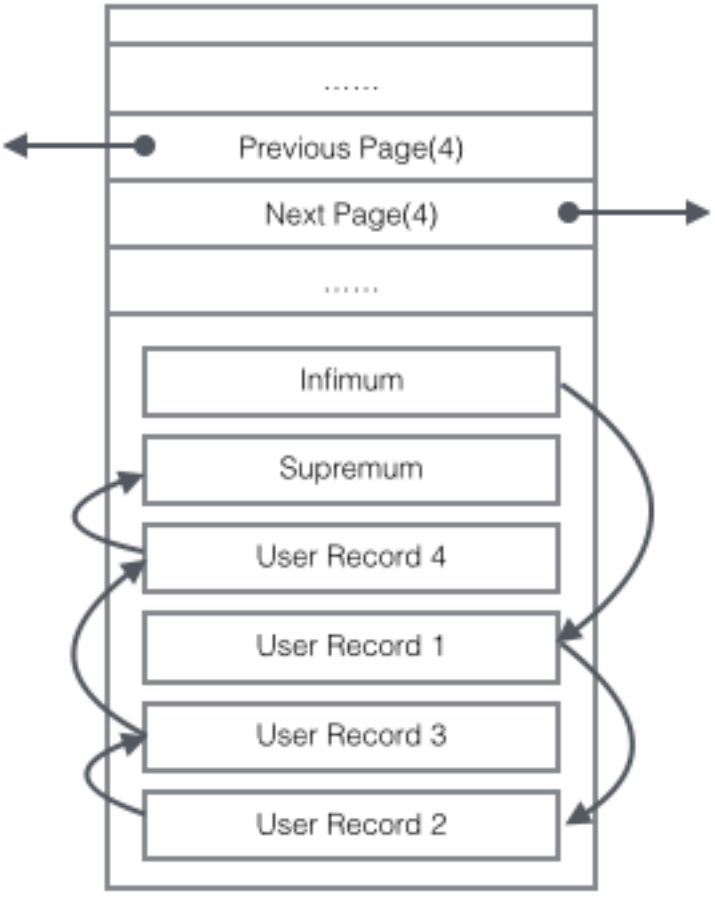
我们重点关注和数据组织结构相关的字段：Page的头部保存了两个指针，分别指向前一个Page和后一个Page，根据这两个指针我们很容易想象出Page链接起来就是一个双向链表的结构，如下图：



Page结构组成的双向链表

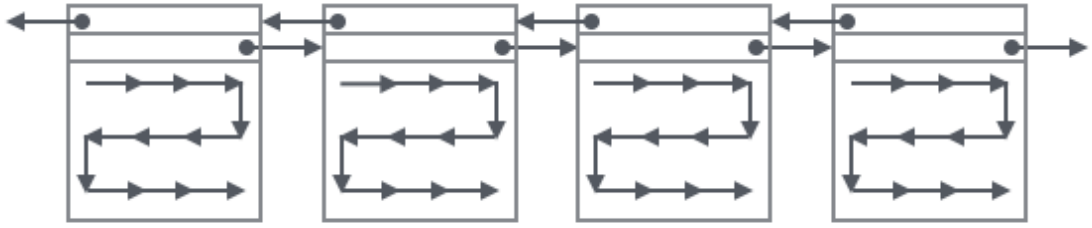
再看看Page的主体内容，我们主要关注行数据和索引的存储，他们都位于Page的User Records部分，User Records占据Page的大部分空间，User Records由一条一条的Record组成。在一个Page内部，单链表的头尾由固定内容的两条记录来表示，字符串形式的"Infimum"代表开头，"Supremum"代表结尾，这两个用来代表开头结尾的Record存储在System Records的，Infinum、Supremum和User Records组成了一个单向链表结构。最初

数据是按照插入的先后顺序排列的，但是随着新数据的插入和旧数据的删除，数据物理顺序会变得混乱，但他们依然通过链表的方式保持着逻辑上的先后顺序，如下图：

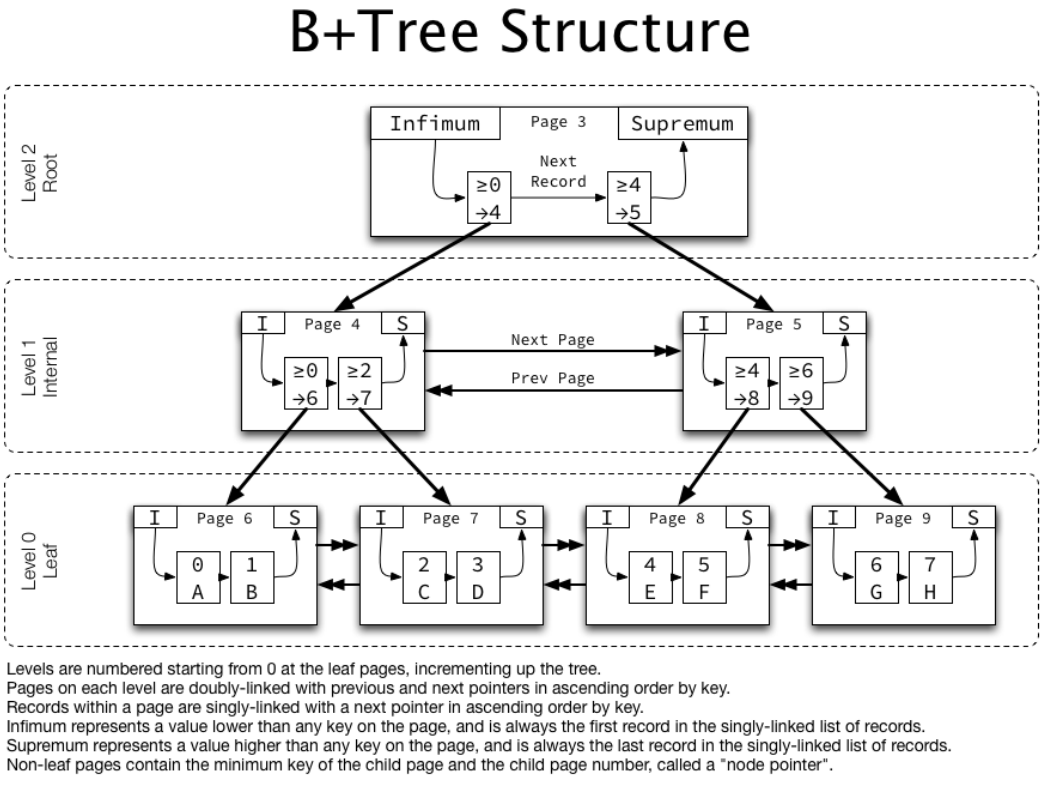


Page页内数据乱序分布

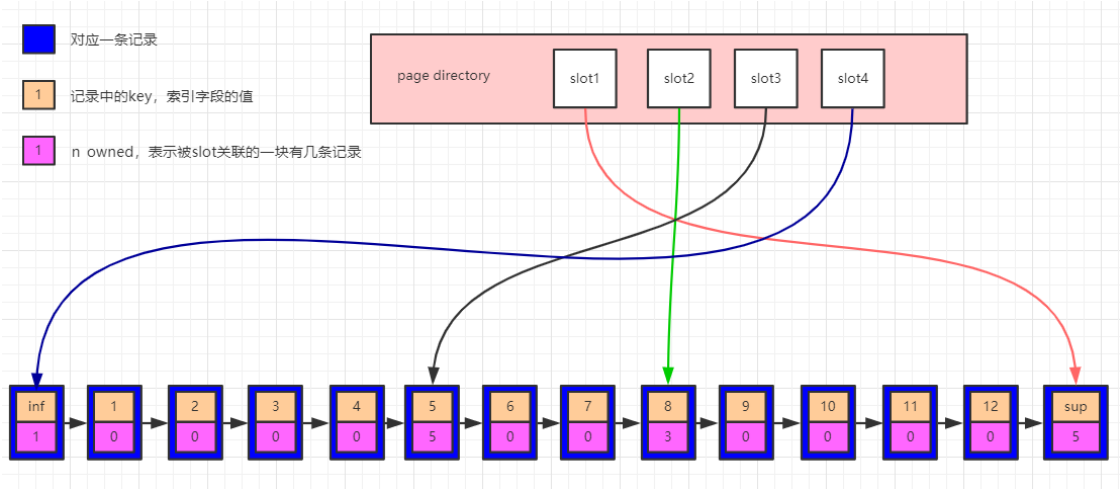
把User Record的组织形式和若干Page组合起来，就看到了稍微完整的形式。



双向链表和单向链表



innodb为了快速查找记录，在页中定义了一个称之为page directory的目录槽（slots）,每个槽位占用两个字节（用于保存指向记录的地址），page directory中的多个slot组成了一个有序数组（可用于二分法快速定位记录，向下看），行记录被Page Directory逻辑的分成了多个块，块与块之间是有序的，能够加速记录的查找，如下图：



看上图，每个行记录的都有一个n*owned*的区域（图中粉色区域），*n*owned标识所属的slot这个这个块有多少条数据，伪记录Infimum的n*owned*值总是*1*，记录*Supremum*的*n*owned的取值范围为[1,8]，其他用户记录n*owned*的取值范围*[4,8]*，并且只有每个块中最大的那条记录的*n*owned才会有值，其他的用户记录的n_owned为0。

数据检索过程

在page中查询数据的时候，先通过b+树中查询方法定位到数据所在的页，然后将页内整体加载到内存中，通过二分法在page directory中检索数据，缩小范围，比如需要检索7，通过二分法查找到7位于slot2和slot3所指向的记录中间，然后从slot3指向的记录5开始向后向后一个个找，可以找到记录7，如果里面没有7，走到slot2向的记录8结束。

n_owned范围控制在[4,8]内，能保证每个slot管辖的范围内数据量控制在[4,8]个，能够加速目标数据的查找，当有数据插入的时候，page directory为了控制每个slot对应块中记录的个数（[4,8]），此时page directory中会对slot的数量进行调整。

对page的结构总结一下

1. b+树中叶子页之间用双向链表连接的，能够实现范围查找
2. 页内部的记录之间是采用单向链表连接的，方便访问下一条记录

3. 为了加快页内部记录的查询，对页内记录上加了个有序的稀疏索引，叫页目录（page directory）

整体上来说mysql中的索引用到了b+树，链表，二分法查找，做到了快速定位目标数据，快速范围查找。

参考资料：
Jeremy Cole的一些文章
https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/
https://blog.jcole.us/innodb/

本篇到此，下一篇实战篇对mysql索引使用上面做详细介绍，喜欢的关注一下，谢谢！

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**
11. **第11篇：深入了解连接查询及原理**

12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**
14. **第14篇：详解事务**
15. **第15篇：详解视图**
16. **第16篇：变量详解**
17. **第17篇：存储过程&自定义函数详解**
18. **第18篇：流程控制语句**
19. **第19篇：游标详解**
20. **第20篇：异常捕获及处理详解**
21. **第21篇：什么是索引?**

第23篇：MySQL索引管理

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第23篇。

环境：mysql5.7.25，cmd命令中进行演示。

代码中被[]包含的表示可选，|符号分开的表示可选其一。

关于索引的，可以先看一下前2篇文章：

22. 什么是索引?

23. **mysql索引原理详解**

本文主要介绍mysql中索引常见的管理操作。

索引分类

分为聚集索引和非聚集索引。

聚集索引

每个表有且一定会有一个聚集索引，整个表的数据存储在聚集索引中，mysql索引是采用B+树结构保存在文件中，叶子节点存储主键的值以及对应记录的数据，非叶子节点不存储记录的数据，只存储主键的值。当表中未指定主键时，mysql内部会自动给每条记录添加一个隐藏的rowid字段（默认4个字节）作为主键，用rowid构建聚集索引。

聚集索引在mysql中又叫主键索引。

非聚集索引（辅助索引）

也是b+树结构，不过有一点和聚集索引不同，非聚集索引叶子节点存储字段（索引字段）的值以及对应记录主键的值，其他节点只存储字段的值（索引字段）。

每个表可以有多个非聚集索引。

*mysql*中非聚集索引分为

单列索引

即一个索引只包含一个列。

多列索引（又称复合索引）

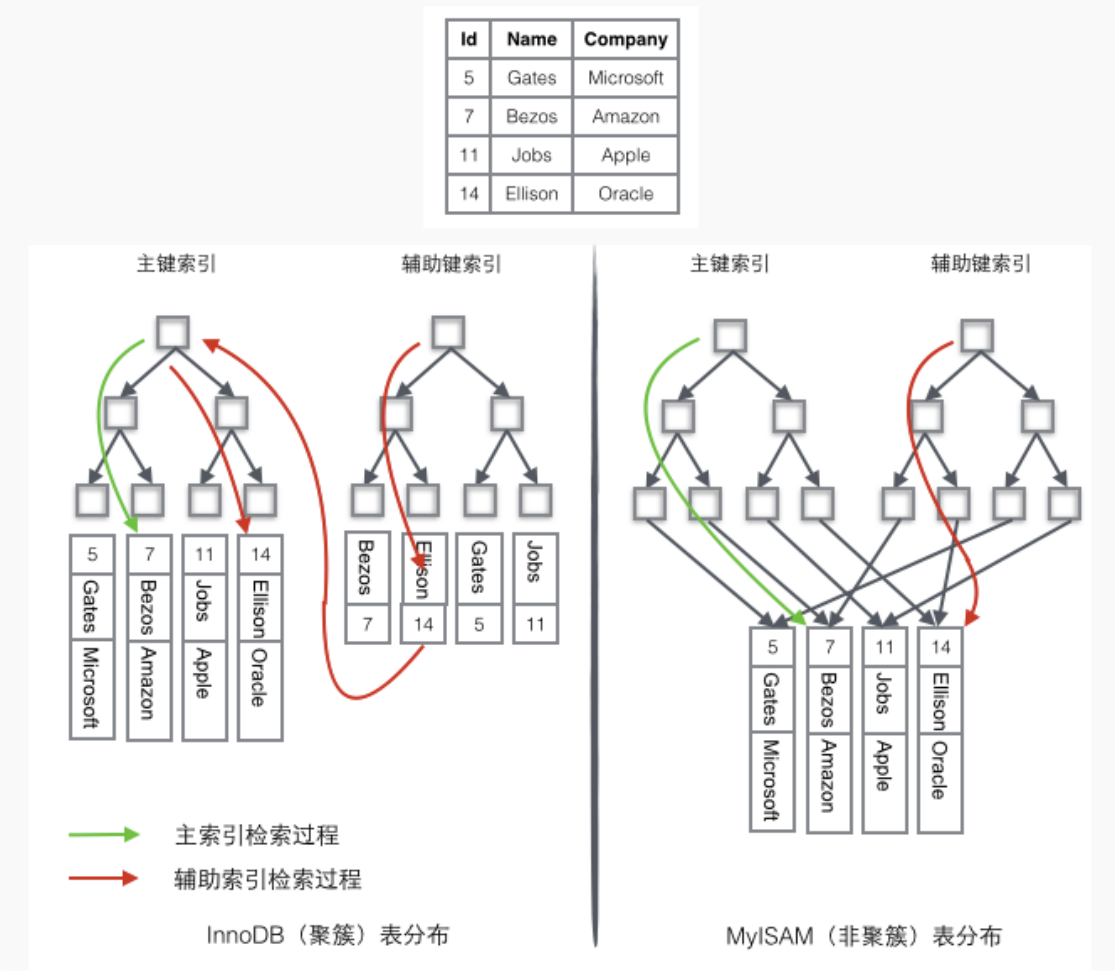
即一个索引包含多个列。

唯一索引

索引列的值必须唯一，允许有一个空值。

数据检索的过程

看一张图：



上面的表中有2个索引：id作为主键索引，name作为辅助索引。

innodb我们用的最多，我们只看图中左边的innodb中数据检索过程：

如果需要查询id=14的数据，只需要在左边的主键索引中检索就可以了。

如果需要搜索name='Ellison'的数据，需要2步：

1. 先在辅助索引中检索到name='Ellison'的数据，获取id为14
2. 再到主键索引中检索id为14的记录

辅助索引相对于主键索引多了第二步。

索引管理

创建索引

方式1：

```
create [unique] index 索引名称 on 表名(列名[(length)]);
```

方式2：

```
alter 表名 add [unique] index 索引名称 on (列名[(length)]);
```

如果字段是char、varchar类型，length可以小于字段实际长度，如果是blog、text等长文本类型，必须指定length。

[unique]：中括号代表可以省略，如果加上了unique，表示创建唯一索引。

如果table后面只写一个字段，就是单列索引，如果写多个字段，就是复合索引，多个字段之间用逗号隔开。

删除索引

```
drop index 索引名称 on 表名;
```

查看索引

查看某个表中所有的索引信息如下：

```
show index from 表名;
```

索引修改

可以先删除索引，再重建索引。

示例

准备200万数据

```
/*建库javacode2018*/
DROP DATABASE IF EXISTS javacode2018;
CREATE DATABASE javacode2018;
```

```
USE javacode2018;

/*建表test1*/
DROP TABLE IF EXISTS test1;
CREATE TABLE test1 (
  id     INT NOT NULL COMMENT '编号',
  name   VARCHAR(20) NOT NULL COMMENT '姓名',
  sex TINYINT NOT NULL COMMENT '性别,1：男，2：女',
  email  VARCHAR(50)
);

/*准备数据*/
DROP PROCEDURE IF EXISTS proc1;
DELIMITER $
CREATE PROCEDURE proc1()
  BEGIN
    DECLARE i INT DEFAULT 1;
    START TRANSACTION;
    WHILE i <= 2000000 DO
      INSERT INTO test1 (id, name, sex, email) VALUES
(i,concat('javacode',i),if(mod(i,2),1,2),concat('javacode',i,'@163.com
'));
      SET i = i + 1;
      if i%10000=0 THEN
        COMMIT;
        START TRANSACTION;
      END IF;
    END WHILE;
    COMMIT;
  END $

DELIMITER ;
CALL proc1();
SELECT count(*) FROM test1;
```

上图中使用存储过程循环插入了200万记录，表中有4个字段，除了sex列，其他列的值都是没有重复的，表中还未建索引。

插入的200万数据中，id，name，email的值都是没有重复的。

无索引我们体验一下查询速度

```
mysql> select * from test1 a where a.id = 1;
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (0.77 sec)
```

上面我们按id查询了一条记录耗时770毫秒，我们在id上面创建个索引感受一下速度。

创建索引

我们在id上面创建一个索引，感受一下：

```
mysql> create index idx1 on test1 (id);
Query OK, 0 rows affected (2.82 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from test1 a where a.id = 1;
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (0.00 sec)
```

上面的查询是不是非常快，耗时1毫秒都不到。

我们在name上也创建个索引，感受一下查询的神速，如下：

```
mysql> create unique index idx2 on test1(name);
Query OK, 0 rows affected (9.67 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> select * from test1 where name = 'javacode1';
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
```

```
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (0.00 sec)
```

查询快如闪电，有没有，索引是如此的神奇。

创建索引并指定长度

通过email检索一下数据

```
mysql> select * from test1 a where a.email =
'javacode1000085@163.com';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 1000085 | javacode1000085 |   1 | javacode1000085@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (1.28 sec)
```

耗时1秒多，回头去看一下插入数据的sql，我们可以看到所有的email记录，每条记录的前面15个字符是不一样的，结尾是一样的（都是@163.com），通过前面15个字符就可以定位一个email了，那么我们可以对email创建索引的时候指定一个长度为15，这样相对于整个email字段更短一些，查询效果是一样的，这样一个页中可以存储更多的索引记录，命令如下：

```
mysql> create index idx3 on test1 (email(15));
Query OK, 0 rows affected (7.67 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

然后看一下查询效果：

```
mysql> select * from test1 a where a.email =
'javacode1000085@163.com';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 1000085 | javacode1000085 |   1 | javacode1000085@163.com |
```

```
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

耗时不到1毫秒，神速。

查看表中的索引

我们看一下test1表中的所有索引，如下：

```
mysql> show index from test1;
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name |
Collation | Cardinality | Sub_part | Packed | Null | Index_type |
Comment | Index_comment |
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| test1 |          0 | idx2     |            1 | name        | A
|     1992727 |     NULL | NULL   |       | BTREE      |          |
|
| test1 |          1 | idx1     |            1 | id          | A
|     1992727 |     NULL | NULL   |       | BTREE      |          |
|
| test1 |          1 | idx3     |            1 | email       | A
|     1992727 |       15 | NULL   | YES   | BTREE      |          |
|
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
3 rows in set (0.00 sec)
```

可以看到test1表中3个索引的详细信息(索引名称、类型，字段)。

删除索引

我们删除idx1，然后再列出test1表所有索引，如下：

```
mysql> drop index idx1 on test1;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> show index from test1;
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name |
Collation | Cardinality | Sub_part | Packed | Null | Index_type |
Comment | Index_comment |
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| test1 |          0 | idx2     |            1 | name        | A
|     1992727 |     NULL | NULL   |       | BTREE      |          |
|
| test1 |          1 | idx3     |            1 | email       | A
|     1992727 |       15 | NULL   | YES  | BTREE      |          |
|
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
2 rows in set (0.00 sec)
```

本篇主要是mysql中索引管理相关一些操作，属于基础知识，希望大家掌握。

下篇文章介绍：

1. 一个表应该创建哪些索引？
2. 有索引时sql应该怎么写？
3. 我的sql为什么不走索引？需要知道内部原理
4. where条件涉及多个字段多个索引时怎么走？
5. 多表连接查询、子查询，怎么去利用索引，内部过程是什么样的？
6. like查询中前面有%的时候为何不走索引？
7. 字段中使用函数的时候为什么不走索引？
8. 字符串查询使用数字作为条件的时候为什么不走索引？、

9. 索引区分度、索引覆盖、最左匹配、索引排序又是什么？原理是什么？

关于上面各种索引选择的问题，我们会深入其原理，让大家知道为什么是这样？而不是只去记录一些优化规则，而不知道其原因，知道其原理用的时候跟得心应手一些。

Mysql系列目录

1. 第**1**篇：**mysql**基础知识
2. 第**2**篇：详解**mysql**数据类型（重点）
3. 第**3**篇：管理员必备技能**(**必须掌握**)**
4. 第**4**篇：**DDL**常见操作
5. 第**5**篇：**DML**操作汇总（**insert,update,delete**）
6. 第**6**篇：**select**查询基础篇
7. 第**7**篇：玩转**select**条件查询，避免采坑
8. 第**8**篇：详解排序和分页**(order by & limit)**
9. 第**9**篇：分组查询详解（**group by & having**）
10. 第**10**篇：常用的几十个函数详解
11. 第**11**篇：深入了解连接查询及原理
12. 第**12**篇：子查询
13. 第**13**篇：细说**NULL**导致的神坑，让人防不胜防
14. 第**14**篇：详解事务
15. 第**15**篇：详解视图
16. 第**16**篇：变量详解
17. 第**17**篇：存储过程**&**自定义函数详解

18. **第18篇：流程控制语句**
19. **第19篇：游标详解**
20. **第20篇：异常捕获及处理详解**
21. **第21篇：什么是索引?**
22. **第22篇：mysql索引原理详解**

第24篇：如何正确的使用索引?

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第24篇。

学习索引，主要是写出更快的sql，当我们写sql的时候，需要明确的知道sql为什么会走索引？为什么有些sql不走索引？sql会走那些索引，为什么会这么走？我们需要了解其原理，了解内部具体过程，这样使用起来才能更顺手，才可以写出更高效的sql。本篇我们就是搞懂这些问题。

读本篇文章之前，需要先了解一些知识：

23. 什么是索引?

24. mysql索引原理详解

25. mysql索引管理详解

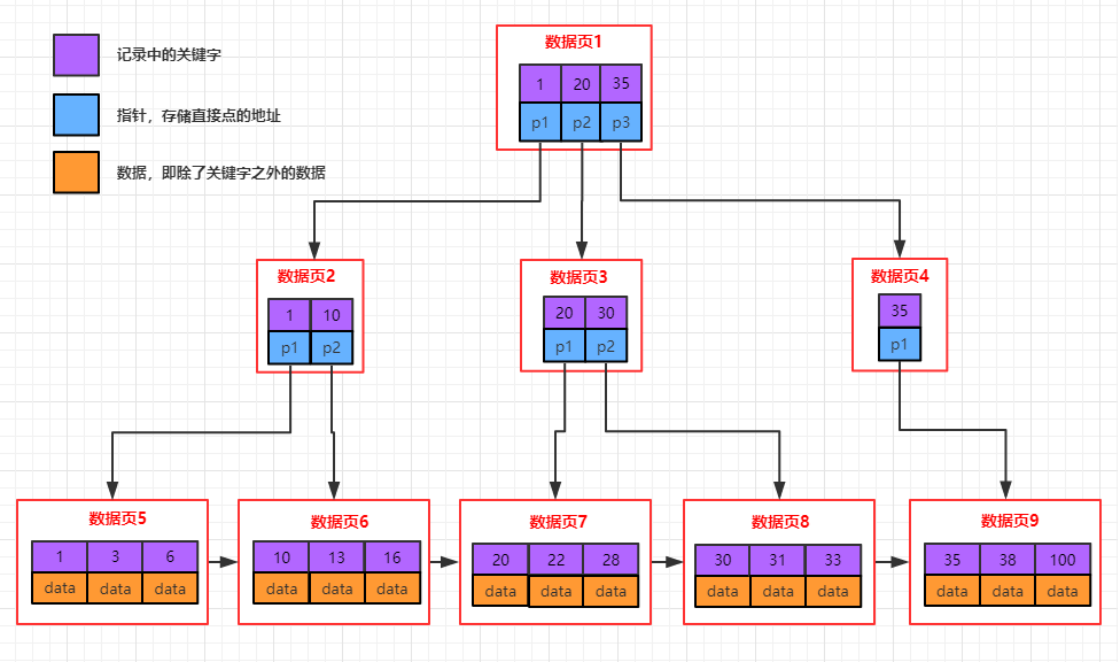
上面3篇文章没有读过的最好去读一下，不然后面的内容会难以理解。

先来回顾一些知识

本篇文章我们以innodb存储引擎为例来做说明。

mysql采用b+树的方式存储索引信息。

b+树结构如下：



说一下b+树的几个特点：

1. 叶子节点（最下面的一层）存储关键字（索引字段的值）信息及对应的data，叶子节点存储了所有记录的关键字信息
2. 其他非叶子节点只存储关键字的信息及子节点的指针

3. 每个叶子节点相当于mysql中的一页，同层级的叶子节点以双向链表的形式相连
4. 每个节点（页）中存储了多条记录，记录之间用单链表的形式连接组成了一条有序的链表，顺序是按照索引字段排序的
5. b+树中检索数据时：每次检索都是从根节点开始，一直需要搜索到叶子节点

InnoDB 的数据是按数据页为单位来读写的。也就是说，当需要读取一条记录的时候，并不是将这个记录本身从磁盘读取出来，而是以页为单位，将整个也加载到内存中，一个页中可能有很多记录，然后在内存中对页进行检索。在innodb中，每个页的大小默认是16kb。

Mysql中索引分为

聚集索引（主键索引）

每个表一定会有一个聚集索引，整个表的数据存储以b+树的方式存在文件中，b+树叶子节点中的key为主键值，data为完整记录的信息；非叶子节点存储主键的值。

通过聚集索引检索数据只需要按照b+树的搜索过程，即可以检索到对应的记录。

非聚集索引

每个表可以有多个非聚集索引，b+树结构，叶子节点的key为索引字段字段的值，data为主键的值；非叶子节点只存储索引字段的值。

通过非聚集索引检索记录的时候，需要2次操作，先在非聚集索引中检索出主键，然后再到聚集索引中检索出主键对应的记录，该过程比聚集索引多了一次操作。

索引怎么走，为什么有些查询不走索引？为什么使用函数了数据就不走索引了？

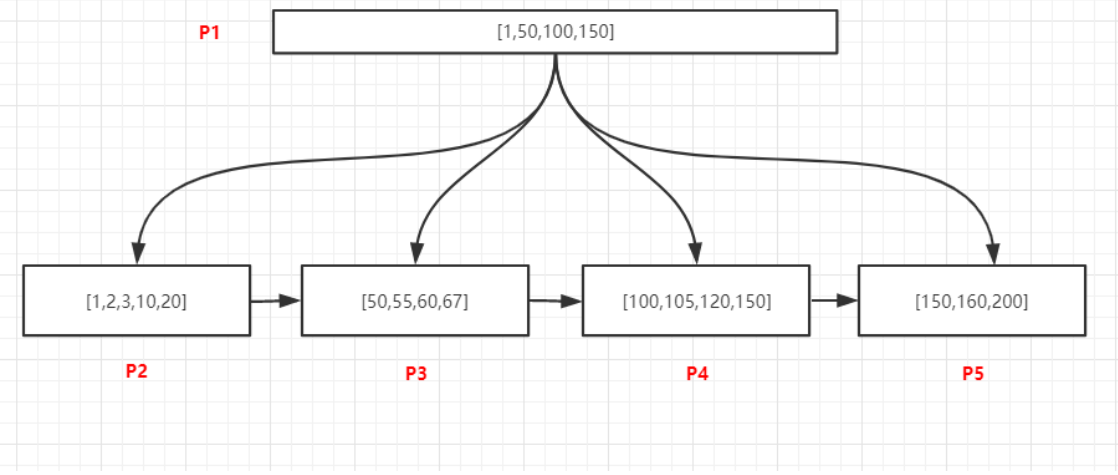
这些问题可以先放一下，我们先看一下b+树检索数据的过程，这个属于原理的部分，理解了b+树各种数据检索过程，上面的问题就都可以理解了。

通常说的这个查询走索引了是什么意思?

当我们对某个字段的值进行某种检索的时候，如果这个检索过程中，我们能够快速定位到目标数据所在的页，有效的降低页的io操作，而不需要去扫描所有的数据页的时候，我们认为这种情况能够有效的利用索引，也称这个检索可以走索引，如果这个过程中不能够确定数据在那些页中，我们认为这种情况下索引对这个查询是无效的，此查询不走索引。

b+树中数据检索过程

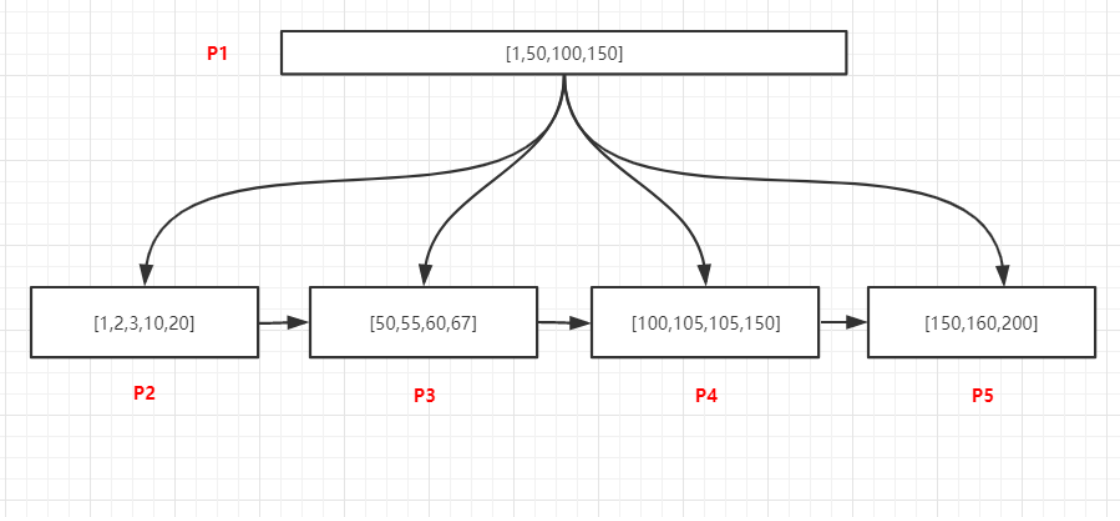
唯一记录检索



如上图，所有的数据都是唯一的，查询105的记录，过程如下：

1. 将P1页加载到内存
2. 在内存中采用二分法查找，可以确定105位于[100,150)中间，所以我们需要去加载100关联P4页
3. 将P4加载到内存中，采用二分法找到105的记录后退出

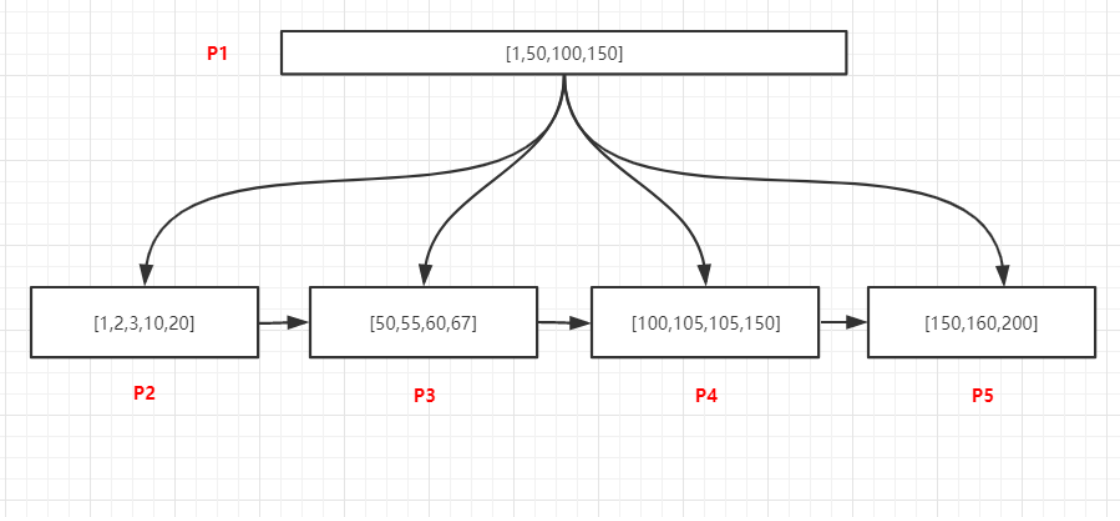
查询某个值的所有记录



如上图，查询105的所有记录，过程如下：

1. 将P1页加载到内存
2. 在内存中采用二分法查找，可以确定105位于[100,150)中间，100关联P4页
3. 将P4加载到内存中，采用二分法找到最有一个小于105的记录，即100，然后通过链表从100开始向后访问，找到所有的105记录，直到遇到第一个大于100的值为止

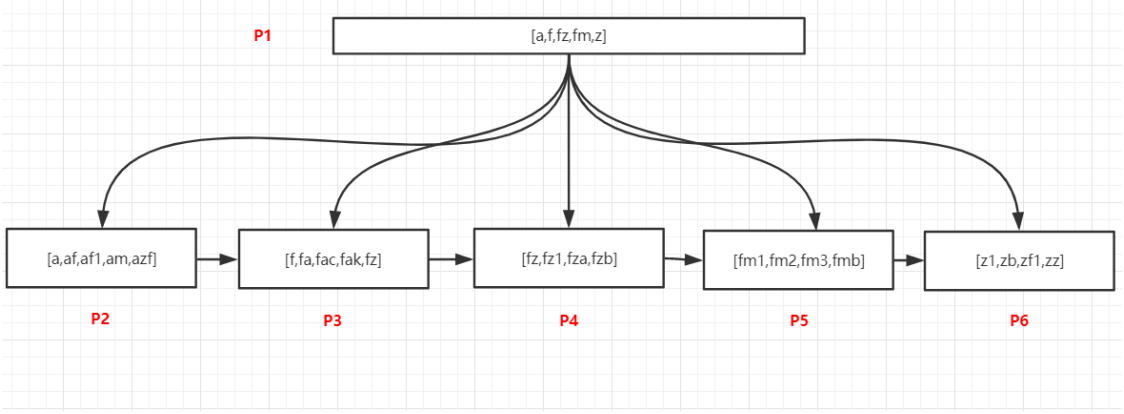
范围查找



数据如上图，查询[55,150]所有记录，由于页和页之间是双向链表升序结构，页内部的数据是单项升序链表结构，所以只用找到范围的起始值所在的位置，然后通过依靠链表访问两个位置之间所有的数据即可，过程如下：

1. 将P1页加载到内存
2. 内存中采用二分法找到55位于50关联的P3页中，150位于P5页中
3. 将P3加载到内存中，采用二分法找到第一个55的记录，然后通过链表结构继续向后访问P3中的60、67，当P3访问完毕之后，通过P3的nextpage指针访问下一页P4中所有记录，继续遍历P4中的所有记录，直到访问到P5中的150为止。

模糊匹配



数据如上图。

查询以*f*开头的所有记录

过程如下：

1. 将P1数据加载到内存中
2. 在P1页的记录中采用二分法找到最后一个小于等于f的值，这个值是f，以及第一个大于f的，这个值是z，f指向叶节点P3，z指向叶节点P6，此时可以断定以f开头的记录可能存在于[P3,P6)这个范围的页内，即P3、P4、P5这三个页中
3. 加载P3这个页，在内部以二分法找到第一条f开头的记录，然后以链表方式继续向后访问P4、P5中的记录，即可以找到所有已f开头的数据

查询包含*f*的记录

包含的查询在sql中的写法是`%f%`，通过索引我们还可以快速定位所在的页么？

可以看一下上面的数据，f在每个页中都存在，我们通过P1页中的记录是无法判断包含f的记录在那些页的，只能通过io的方式加载所有叶子节点，并且遍历所有记录进行过滤，才可以找到包含f的记录。

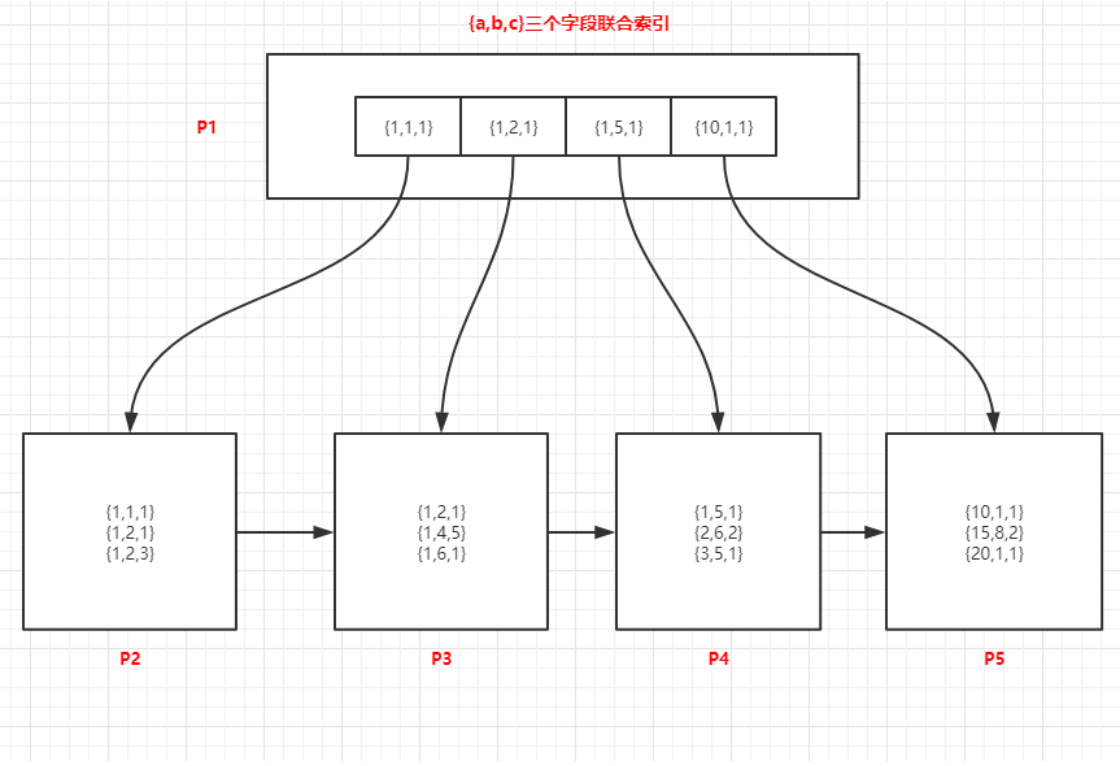
所以如果使用了`%值%`这种方式，索引对查询是无效的。

最左匹配原则

> 当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+树是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

来一些示例我们体验一下。

下图中是3个字段(a,b,c)的联合索引，索引中数据的顺序是以`a asc,b asc,c asc`这种排序方式存储在节点中的，索引先以a字段升序，如果a相同的时候，以b字段升序，b相同的时候，以c字段升序，节点中每个数据认真看一下。



查询*a=1*的记录

由于页中的记录是以`a asc,b asc,c asc`这种排序方式存储的，所以a字段是有序的，可以通过二分法快速检索到，过程如下：

1. 将P1加载到内存中
2. 在内存中对P1中的记录采用二分法找，可以确定a=1的记录位于{1,1,1}和{1,5,1}关联的范围内，这两个值子节点分别是P2、P4
3. 加载叶子节点P2，在P2中采用二分法快速找到第一条a=1的记录，然后通过链表向下一条及下一页开始检索，直到在P4中找到第一个不满足a=1的记录为止

查询*a=1 and b=5*的记录

方法和上面的一样，可以确定a=1 and b=5的记录位于{1,1,1}和{1,5,1}关联的范围内，查找过程和a=1查找步骤类似。

查询*b=1*的记录

这种情况通过P1页中的记录，是无法判断b=1的记录在那些页中的，只能加锁索引树所有叶子节点，对所有记录进行遍历，然后进行过滤，此时索引是无效的。

按照*c*的值查询

这种情况和查询b=1也一样，也只能扫描所有叶子节点，此时索引也无效了。

按照*b*和*c*一起查

这种也是无法利用索引的，也只能对所有数据进行扫描，一条条判断了，此时索引无效。

按照*[a,c]*两个字段查询

这种只能利用到索引中的a字段了，通过a确定索引范围，然后加载a关联的所有记录，再对c的值进行过滤。

查询*a=1 and b>=0 and c=1*的记录

这种情况只能先确定a=1 and b>=0所在页的范围，然后对这个范围的所有页进行遍历，c字段在这个查询的过程中，是无法确定c的数据在哪些页的，此时我们称c是不走索引的，只有a、b能够有效的确定索引页的范围。

类似这种的还有>、<、between and，多字段索引的情况下，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配。

上面说的各种情况，大家都多看一下图中数据，认真分析一下查询的过程，基本上都可以理解了。

上面这种查询叫做最左匹配原则。

索引区分度

我们看2个有序数组

[1,2,3,4,5,6,7,8,8,9,10]

[1,1,1,1,1,8,8,8,8,8]

上面2个数组是有序的，都是10条记录，如果我需要检索值为8的所有记录，那个更快一些?

咱们使用二分法查找包含8的所有记录过程如下：先使用二分法找到最后一个小于8的记录，然后沿着这条记录向后获取下一个记录，和8对比，知道遇到第一个大于8的数字结束，或者到达数组末尾结束。

采用上面这种方法找到8的记录，第一个数组中更快的一些。因为第二个数组中含有8的比例更多的，需要访问以及匹配的次数更多一些。

这里就涉及到数据的区分度问题:

索引区分度 = count(distint 记录) / count(记录)。

当索引区分度高的时候，检索数据更快一些，索引区分度太低，说明重复的数据比较多，检索的时候需要访问更多的记录才能够找到所有目标数据。

当索引区分度非常小的时候，基本上接近于全索引数据的扫描了，此时查询速度是比较慢的。

第一个数组索引区分度为1，第二个区分度为0.2，所以第一个检索更快的一些。

所以我们创建索引的时候，尽量选择区分度高的列作为索引。

正确使用索引

准备400万测试数据

```
/*建库javacode2018*/
DROP DATABASE IF EXISTS javacode2018;
CREATE DATABASE javacode2018;
```

```
USE javacode2018;
/*建表test1*/
DROP TABLE IF EXISTS test1;
CREATE TABLE test1 (
  id     INT NOT NULL COMMENT '编号',
  name   VARCHAR(20) NOT NULL COMMENT '姓名',
  sex TINYINT NOT NULL COMMENT '性别,1：男，2：女',
  email  VARCHAR(50)
);

/*准备数据*/
DROP PROCEDURE IF EXISTS proc1;
DELIMITER $
CREATE PROCEDURE proc1()
  BEGIN
    DECLARE i INT DEFAULT 1;
    START TRANSACTION;
    WHILE i <= 4000000 DO
      INSERT INTO test1 (id, name, sex, email) VALUES
(i,concat('javacode',i),if(mod(i,2),1,2),concat('javacode',i,'@163.com
'));
      SET i = i + 1;
      if i%10000=0 THEN
        COMMIT;
        START TRANSACTION;
      END IF;
    END WHILE;
    COMMIT;
  END $

DELIMITER ;
CALL proc1();
```

上面插入的400万数据，除了sex列，其他列的值都是没有重复的。

无索引检索效果

400万数据，我们随便查询几个记录看一下效果。

按照id查询记录

```
mysql> select * from test1 where id = 1;
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (1.91 sec)
```

id=1的数据，表中只有一行，耗时近2秒，由于id列无索引，只能对400万数据进行全表扫描。

主键检索

test1表中没有明确的指定主键，我们将id设置为主键：

```
mysql> alter table test1 modify id int not null primary key;
Query OK, 0 rows affected (10.93 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show index from test1;
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name |
Collation | Cardinality | Sub_part | Packed | Null | Index_type |
Comment | Index_comment |
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
| test1 |          0 | PRIMARY  |            1 | id          | A
|     3980477 |     NULL | NULL   |       | BTREE      |          |
|
+-------+------------+----------+--------------+-------------
+-----------+-------------+----------+--------+------+------------
+---------+---------------+
1 row in set (0.00 sec)
```

id被置为主键之后，会在id上建立聚集索引，随便检索一条我们看一下效果：

```
mysql> select * from test1 where id = 1000000;
+---------+-----------------+-----+------------------------+
```

```
| id      | name             | sex | email                    |
+---------+------------------+-----+--------------------------+
| 1000000 | javacode1000000 |   2 | javacode1000000@163.com |
+---------+------------------+-----+--------------------------+
1 row in set (0.00 sec)
```

这个速度很快，这个走的是上面介绍的唯一记录检索。

between and范围检索

```
mysql> select count(*) from test1 where id between 100 and 110;
+----------+
| count(*) |
+----------+
|       11 |
+----------+
1 row in set (0.00 sec)
```

速度也很快，id上有主键索引，这个采用的上面介绍的范围查找可以快速定位目标数据。

但是如果范围太大，跨度的page也太多，速度也会比较慢，如下：

```
mysql> select count(*) from test1 where id between 1 and 2000000;
+----------+
| count(*) |
+----------+
|  2000000 |
+----------+
1 row in set (1.17 sec)
```

上面id的值跨度太大，1所在的页和200万所在页中间有很多页需要读取，所以比较慢。

所以使用**between and的时候，区间跨度不要太大。**

in的检索

in方式检索数据，我们还是经常用的。

平时我们做项目的时候，建议少用表连接，比如电商中需要查询订单的信息和订单中商品的名称，可以先查询查询订单表，然后订单表中取出商品的id列表，采用in的方式到商品表检索商品信息，由于商品id是商品表的主键，所以检索速度还是比较快的。

通过id在400万数据中检索100条数据，看看效果：

```
mysql> select * from test1 a where a.id in (100000, 100001, 100002,
100003, 100004, 100005, 100006, 100007, 100008, 100009, 100010,
100011, 100012, 100013, 100014, 100015, 100016, 100017, 100018,
100019, 100020, 100021, 100022, 100023, 100024, 100025, 100026,
100027, 100028, 100029, 100030, 100031, 100032, 100033, 100034,
100035, 100036, 100037, 100038, 100039, 100040, 100041, 100042,
100043, 100044, 100045, 100046, 100047, 100048, 100049, 100050,
100051, 100052, 100053, 100054, 100055, 100056, 100057, 100058,
100059, 100060, 100061, 100062, 100063, 100064, 100065, 100066,
100067, 100068, 100069, 100070, 100071, 100072, 100073, 100074,
100075, 100076, 100077, 100078, 100079, 100080, 100081, 100082,
100083, 100084, 100085, 100086, 100087, 100088, 100089, 100090,
100091, 100092, 100093, 100094, 100095, 100096, 100097, 100098,
100099);
+--------+----------------+-----+------------------------+
| id     | name           | sex | email                  |
+--------+----------------+-----+------------------------+
| 100000 | javacode100000 |   2 | javacode100000@163.com |
| 100001 | javacode100001 |   1 | javacode100001@163.com |
| 100002 | javacode100002 |   2 | javacode100002@163.com |
.......
| 100099 | javacode100099 |   1 | javacode100099@163.com |
+--------+----------------+-----+------------------------+
100 rows in set (0.00 sec)
```

耗时不到1毫秒，还是相当快的。

这个相当于多个分解为多个唯一记录检索，然后将记录合并。

多个索引时查询如何走?

我们在name、sex两个字段上分别建个索引

```
mysql> create index idx1 on test1(name);
Query OK, 0 rows affected (13.50 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> create index idx2 on test1(sex);
Query OK, 0 rows affected (6.77 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

看一下查询：

```
mysql> select * from test1 where name='javacode3500000' and sex=2;
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 3500000 | javacode3500000 |   2 | javacode3500000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

上面查询速度很快，name和sex上各有一个索引，觉得上面走哪个索引？

有人说name位于where第一个，所以走的是name字段所在的索引，过程可以解释为这样：

1. 走name所在的索引找到javacode3500000对应的所有记录
2. 遍历记录过滤出sex=2的值

我们看一下name='javacode3500000'检索速度，确实很快，如下：

```
mysql> select * from test1 where name='javacode3500000';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 3500000 | javacode3500000 |   2 | javacode3500000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

走name索引，然后再过滤，确实可以，速度也很快，果真和where后字段顺序有关么？我们把name和sex的顺序对调一下，如下：

```
mysql> select * from test1 where sex=2 and name='javacode3500000';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
```

```
+---------+-----------------+-----+-------------------------+
| 3500000 | javacode3500000 |   2 | javacode3500000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

速度还是很快，这次是不是先走sex索引检索出数据，然后再过滤name呢？我们先来看一下sex=2查询速度：

```
mysql> select count(id) from test1 where sex=2;
+-----------+
| count(id) |
+-----------+
|   2000000 |
+-----------+
1 row in set (0.36 sec)
```

看上面，查询耗时360毫秒，200万数据，如果走sex肯定是不行的。

我们使用explain来看一下：

```
mysql> explain select * from test1 where sex=2 and
name='javacode3500000';
+----+-------------+-------+------------+------+---------------+------
+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key
| key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------
+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | test1 | NULL       | ref  | idx1,idx2     | idx1
| 62      | const |    1 |    50.00 | Using where |
+----+-------------+-------+------------+------+---------------+------
+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

possible_keys：列出了这个查询可能会走两个索引（idx1、idx2）

实际上走的却是idx1（key列：实际走的索引）。

当多个条件中有索引的时候，并且关系是and的时候，会走索引区分度高的，显然name字段重复度很低，走name查询会更快一些。

模糊查询

看两个查询

```
mysql> select count(*) from test1 a where a.name like 'javacode1000%';
+----------+
| count(*) |
+----------+
|     1111 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from test1 a where a.name like
'%javacode1000%';
+----------+
| count(*) |
+----------+
|     1111 |
+----------+
1 row in set (1.78 sec)
```

上面第一个查询可以利用到name字段上面的索引，下面的查询是无法确定需要查找的值所在的范围的，只能全表扫描，无法利用索引，所以速度比较慢，这个过程上面有说过。

回表

当需要查询的数据在索引树中不存在的时候，需要再次到聚集索引中去获取，这个过程叫做回表，如查询：

```
mysql> select * from test1 where name='javacode3500000';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 3500000 | javacode3500000 |   2 | javacode3500000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

上面查询是*，由于name列所在的索引中只有name、id两个列的值，不包含sex、email，所以上面过程如下：

1. 走name索引检索javacode3500000对应的记录，取出id为3500000
2. 在主键索引中检索出id=3500000的记录，获取所有字段的值

索引覆盖

查询中采用的索引树中包含了查询所需要的所有字段的值，不需要再去聚集索引检索数据，这种叫索引覆盖。

我们来看一个查询：

```
select id,name from test1 where name='javacode3500000';
```

name对应idx1索引，id为主键，所以idx1索引树叶子节点中包含了name、id的值，这个查询只用走idx1这一个索引就可以了，如果select后面使用*，还需要一次回表获取sex、email的值。

所以写sql的时候，尽量避免使用*，*可能会多一次回表操作，需要看一下是否可以使用索引覆盖来实现，效率更高一些。

索引下推

简称ICP，Index Condition Pushdown(ICP)是MySQL 5.6中新特性，是一种在存储引擎层使用索引过滤数据的一种优化方式，ICP可以减少存储引擎访问基表的次数以及MySQL服务器访问存储引擎的次数。

举个例子来说一下：

我们需要查询name以javacode35开头的，性别为1的记录数，sql如下：

```
mysql> select count(id) from test1 a where name like 'javacode35%' and
sex = 1;
+-----------+
| count(id) |
+-----------+
|     55556 |
+-----------+
1 row in set (0.19 sec)
```

过程：

1. 走name索引检索出以javacode35的第一条记录，得到记录的id
2. 利用id去主键索引中查询出这条记录R1
3. 判断R1中的sex是否为1，然后重复上面的操作，直到找到所有记录为止。

   上面的过程中需要走name索引以及需要回表操作。

如果采用ICP的方式，我们可以这么做，创建一个(name,sex)的组合索引，查询过程如下：

1. 走(name,sex)索引检索出以javacode35的第一条记录，可以得到(name,sex,id)，记做R1
2. 判断R1.sex是否为1，然后重复上面的操作，知道找到所有记录为止

   这个过程中不需要回表操作了，通过索引的数据就可以完成整个条件的过滤，速度比上面的更快一些。

数字使字符串类索引失效

```
mysql> insert into test1 (id,name,sex,email) values
(4000001,'1',1,'javacode2018@163.com');
Query OK, 1 row affected (0.00 sec)

mysql> select * from test1 where name = '1';
+---------+------+-----+----------------------+
| id      | name | sex | email                |
+---------+------+-----+----------------------+
| 4000001 | 1    |   1 | javacode2018@163.com |
+---------+------+-----+----------------------+
1 row in set (0.00 sec)

mysql> select * from test1 where name = 1;
+---------+------+-----+----------------------+
| id      | name | sex | email                |
+---------+------+-----+----------------------+
| 4000001 | 1    |   1 | javacode2018@163.com |
```

```
+---------+------+-----+----------------------+
1 row in set, 65535 warnings (3.30 sec)
```

上面3条sql，我们插入了一条记录。

第二条查询很快，第三条用name和1比较，name上有索引，name是字符串类型，字符串和数字比较的时候，会将字符串强制转换为数字，然后进行比较，所以第二个查询变成了全表扫描，只能取出每条数据，将name转换为数字和1进行比较。

数字字段和字符串比较什么效果呢？如下：

```
mysql> select * from test1 where id = '4000000';
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 4000000 | javacode4000000 |   2 | javacode4000000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)

mysql> select * from test1 where id = 4000000;
+---------+-----------------+-----+-------------------------+
| id      | name            | sex | email                   |
+---------+-----------------+-----+-------------------------+
| 4000000 | javacode4000000 |   2 | javacode4000000@163.com |
+---------+-----------------+-----+-------------------------+
1 row in set (0.00 sec)
```

id上面有主键索引，id是int类型的，可以看到，上面两个查询都非常快，都可以正常利用索引快速检索，所以如果字段是数组类型的，查询的值是字符串还是数组都会走索引。

函数使索引无效

```
mysql> select a.name+1 from test1 a where a.name = 'javacode1';
+----------+
| a.name+1 |
+----------+
|        1 |
+----------+
```

```
1 row in set, 1 warning (0.00 sec)

mysql> select * from test1 a where concat(a.name,'1') = 'javacode11';
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (2.88 sec)
```

name上有索引，上面查询，第一个走索引，第二个不走索引，第二个使用了函数之后，name所在的索引树是无法快速定位需要查找的数据所在的页的，只能将所有页的记录加载到内存中，然后对每条数据使用函数进行计算之后再进行条件判断，此时索引无效了，变成了全表数据扫描。

结论：索引字段使用函数查询使索引无效。

运算符使索引无效

```
mysql> select * from test1 a where id = 2 - 1;
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (0.00 sec)

mysql> select * from test1 a where id+1 = 2;
+----+-----------+-----+-------------------+
| id | name      | sex | email             |
+----+-----------+-----+-------------------+
|  1 | javacode1 |   1 | javacode1@163.com |
+----+-----------+-----+-------------------+
1 row in set (2.41 sec)
```

id上有主键索引，上面查询，第一个走索引，第二个不走索引，第二个使用运算符，id所在的索引树是无法快速定位需要查找的数据所在的页的，只能将所有页的记录加载到内存中，然后对每条数据的id进行计算之后再判断是否等于1，此时索引无效了，变成了全表数据扫描。

结论：索引字段使用了函数将使索引无效。

使用索引优化排序

我们有个订单表t*order(id,user*id,addtime,price)，经常会查询某个用户的订单，并且按照addtime升序排序，应该怎么创建索引呢？我们来分析一下。

在user_id上创建索引，我们分析一下这种情况，数据检索的过程：

1. 走user_id索引，找到记录的的id
2. 通过id在主键索引中回表检索出整条数据
3. 重复上面的操作，获取所有目标记录
4. 在内存中对目标记录按照addtime进行排序

我们要知道当数据量非常大的时候，排序还是比较慢的，可能会用到磁盘中的文件，有没有一种方式，查询出来的数据刚好是排好序的。

我们再回顾一下mysql中b+树数据的结构，记录是按照索引的值排序组成的链表，如果将user*id*和*addtime*放在一起组成联合索引*(user*id,addtime)，这样通过user_id检索出来的数据自然就是按照addtime排好序的，这样直接少了一步排序操作，效率更好，如果需addtime降序，只需要将结果翻转一下就可以了。

总结一下使用索引的一些建议

1. 在区分度高的字段上面建立索引可以有效的使用索引，区分度太低，无法有效的利用索引，可能需要扫描所有数据页，此时和不使用索引差不多
2. 联合索引注意最左匹配原则：必须按照从左到右的顺序匹配，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整
3. 查询记录的时候，少使用*，尽量去利用索引覆盖，可以减少回表操作，提升效率

4. 有些查询可以采用联合索引，进而使用到索引下推（IPC），也可以减少回表操作，提升效率
5. 禁止对索引字段使用函数、运算符操作，会使索引失效
6. 字符串字段和数字比较的时候会使索引无效
7. 模糊查询'%值%'会使索引无效，变为全表扫描，但是'值%'这种可以有效利用索引
8. 排序中尽量使用到索引字段，这样可以减少排序，提升查询效率

Mysql系列目录

1. **第1篇：mysql基础知识**
2. **第2篇：详解mysql数据类型（重点）**
3. **第3篇：管理员必备技能(必须掌握)**
4. **第4篇：DDL常见操作**
5. **第5篇：DML操作汇总（insert,update,delete）**
6. **第6篇：select查询基础篇**
7. **第7篇：玩转select条件查询，避免采坑**
8. **第8篇：详解排序和分页(order by & limit)**
9. **第9篇：分组查询详解（group by & having）**
10. **第10篇：常用的几十个函数详解**
11. **第11篇：深入了解连接查询及原理**
12. **第12篇：子查询**
13. **第13篇：细说NULL导致的神坑，让人防不胜防**
14. **第14篇：详解事务**

15. 第15篇：详解视图
16. 第16篇：变量详解
17. 第17篇：存储过程&自定义函数详解
18. 第18篇：流程控制语句
19. 第19篇：游标详解
20. 第20篇：异常捕获及处理详解
21. 第21篇：什么是索引?
22. 第22篇：mysql索引原理详解
23. 第23篇：mysql索引管理详解

关注公众号：大侠学JAVA获取更多高质量资料

第25篇：sql中的where条件在数据库中提取与应用浅析

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第25篇。

读本篇文章之前，需要先了解一些知识：

24. 什么是索引？
25. mysql索引原理详解
26. mysql索引管理详解
27. 如何正确的使用索引？

上面3篇文章没有读过的最好去读一下，不然后面的内容会难以理解。

问题描述

一条SQL，在数据库中是如何执行的呢？相信很多人都会对这个问题比较感兴趣。当然，要完整描述一条SQL在数据库中的生命周期，这是一个非常巨大的问题，涵盖了SQL的词法解析、语法解析、权限检查、查询优化、SQL执行等一系列的步骤，简短的篇幅是绝对无能为力的。因此，本文挑选了其中的部分内容，也是我一直都想写的一个内容，做重点介绍：

给定一条SQL，如何提取其中的where条件？where条件中的每个子条件，在SQL执行的过程中有分别起着什么样的作用？

通过本文的介绍，希望读者能够更好地理解查询条件对于SQL语句的影响；撰写出更为优质的SQL语句；更好地理解一些术语，例如：MySQL 5.6中一个重要的优化——Index Condition Pushdown，究竟push down了什么？

本文接下来的内容，安排如下：

1. 简单介绍关系型数据库中数据的组织形式
2. 给定一条SQL，如何提取其中的where条件
3. 最后做一个小的总结

关系型数据库中的数据组织

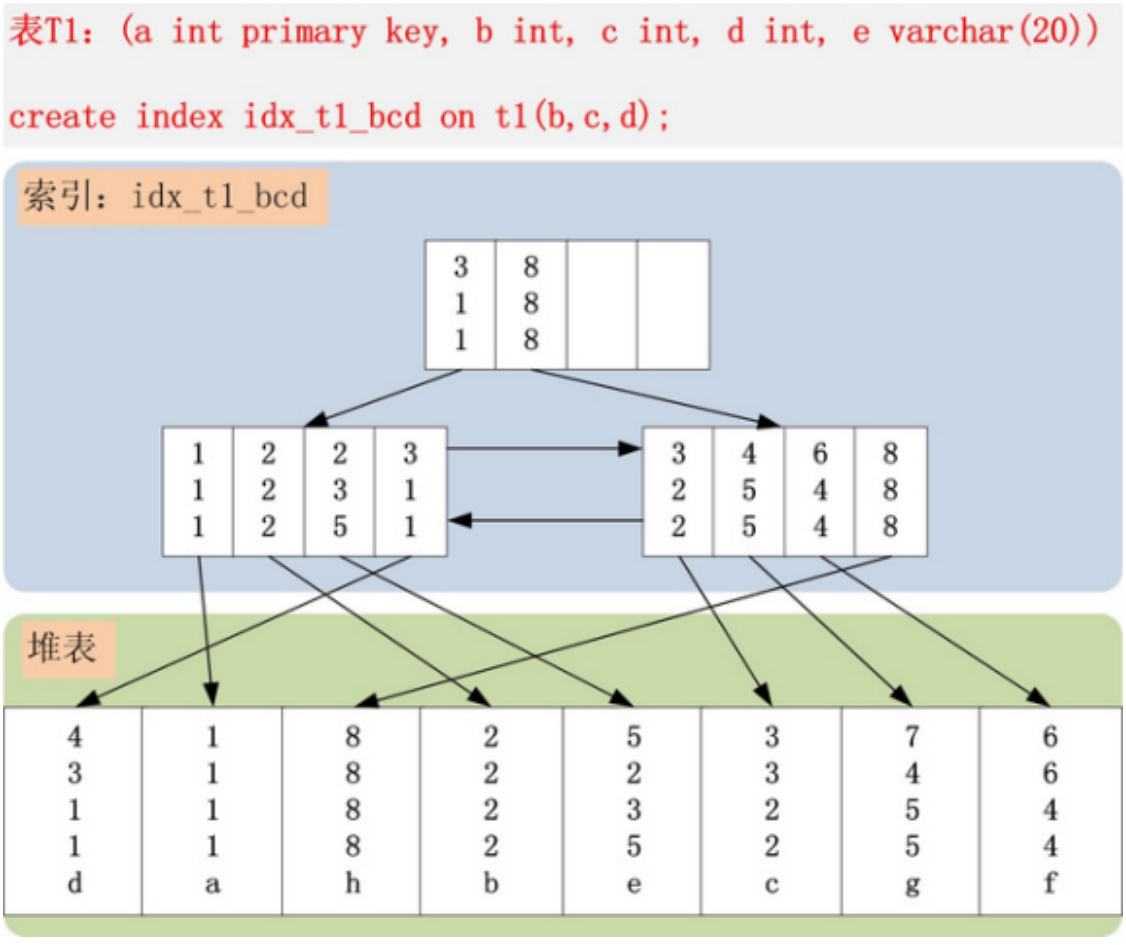
关系型数据库中，数据组织涉及到两个最基本的结构：表与索引。表中存储的是完整记录，一般有两种组织形式：堆表(所有的记录无序存储)，或者是聚簇索引表(所有的记录，按照记录主键进行排序存储)。索引中存储的是完整记录的一个子集，用于加速记录的查询速度，索引的组织形式，一般均为B+树结构。

有了这些基本知识之后，接下来让我们创建一张测试表，为表新增几个索引，然后插入几条记录，最后看看表的完整数据组织、存储结构式怎么样的。(注意：下面的实例，使用的表的结构为堆表形式，这也是Oracle/DB2/PostgreSQL等数据库采用的表组织形式，而不是InnoDB引擎所采用的聚簇索引表。其实，表结构采用何种形式并不重要，最重要的是理解下面章节的核心，在任何表结构中均适用)

```
create table t1 (a int primary key, b int, c int, d int, e
varchar(20));
create index idx_t1_bcd on t1(b, c, d);

insert into t1 values (4,3,1,1,'d');
insert into t1 values (1,1,1,1,'a');
insert into t1 values (8,8,8,8,'h'):
insert into t1 values (2,2,2,2,'b');
insert into t1 values (5,2,3,5,'e');
insert into t1 values (3,3,2,2,'c');
insert into t1 values (7,4,5,5,'g');
insert into t1 values (6,6,4,4,'f');
```

t1表的存储结构如下图所示(只画出了idx*t1*bcd索引与t1表结构，没有包括t1表的主键索引)：



简单分析一下上图，idx*t1*bcd索引上有[b,c,d]三个字段(注意：若是InnoDB类的聚簇索引表，idx*t1*bcd上还会包括主键a字段)，不包括[a,e]字段。idx*t1*bcd索引，首先按照b字段排序，b字段相同，则按照c字段排序，以此类推。记录在索引中按照[b,c,d]排序，但是在堆表上是乱序的，不按照任何字段排序。

SQL的where条件提取

在有了以上的t1表之后，接下来就可以在此表上进行SQL查询了，获取自己想要的数据。例如，考虑以下的一条SQL：

```
select * from t1 where b >= 2 and b < 8 and c > 1 and d != 4 and e !=
'a';
```

一条比较简单的SQL，一目了然就可以发现where条件使用到了[b,c,d,e]四个字段，而t1表的idx*t1*bcd索引，恰好使用了[b,c,d]这三个字段，那么走idx*t1*bcd索引进行条件过滤，应该是一个不错的选择。接下来，让我们抛弃数据库的思想，直接思考这条SQL的几个关键性问题：

此SQL，覆盖索引idx*t1*bcd上的哪个范围?

起始范围：记录[2,2,2]是第一个需要检查的索引项。索引起始查找范围由b >= 2，c > 1决定。

终止范围：记录[8,8,8]是第一个不需要检查的记录，而之前的记录均需要判断。索引的终止查找范围由b < 8决定；

在确定了查询的起始、终止范围之后，SQL中还有哪些条件可以使用索引idx*t1*bcd过滤?

根据SQL，固定了索引的查询范围[(2,2,2),(8,8,8))之后，此索引范围中并不是每条记录都是满足where查询条件的。例如：(3,1,1)不满足c > 1的约束；(6,4,4)不满足d != 4的约束。而c，d列，均可在索引idx*t1*bcd中过滤掉不满足条件的索引记录的。

因此，SQL中还可以使用c > 1 and d != 4条件进行索引记录的过滤。

在确定了索引中最终能够过滤掉的条件之后，还有哪些条件是索引无法过滤的?

此问题的答案显而易见，e != ‘a’这个查询条件，无法在索引idx*t1*bcd上进行过滤，因为索引并未包含e列。e列只在堆表上存在，为了过滤此查询条件，必须将已经满足索引查询条件的记录回表，取出表中的e列，然后使用e列的查询条件e != ‘a’进行最终的过滤。

在理解以上的问题解答的基础上，做一个抽象，可总结出一套放置于所有SQL语句而皆准的where查询条件的提取规则：

所有SQL的where条件，均可归纳为3大类

- Index Key (First Key & Last Key)
- Index Filter
- Table Filter

接下来，让我们来详细分析这3大类分别是如何定义，以及如何提取的。

1.Index Key

用于确定SQL查询在索引中的连续范围(起始范围+结束范围)的查询条件，被称之为Index Key。由于一个范围，至少包含一个起始与一个终止，因此Index Key也被拆分为Index First Key和Index Last Key，分别用于定位索引查找的起始，以及索引查询的终止条件。

Index First Key

用于确定索引查询的起始范围。提取规则：从索引的第一个键值开始，检查其在where条件中是否存在，若存在并且条件是=、>=，则将对应的条件加入Index First Key之中，继续读取索引的下一个键值，使用同样的提取规则；若存在并且条件是>，则将对应的条件加入Index First Key中，同时终止Index First Key的提取；若不存在，同样终止Index First Key的提取。

针对上面的SQL，应用这个提取规则，提取出来的Index First Key为(b >= 2, c > 1)。由于c的条件为 >，提取结束，不包括d。

Index Last Key

Index Last Key的功能与Index First Key正好相反，用于确定索引查询的终止范围。提取规则：从索引的第一个键值开始，检查其在where条件中是否存在，若存在并且条件是=、<=，则将对应条件加入到Index Last Key中，继续提取索引的下一个键值，使用同样的提取规则；若存在并且条件是 < ，则将条件加入到Index Last Key中，同时终止提取；若不存在，同样终止Index Last Key的提取。

针对上面的SQL，应用这个提取规则，提取出来的Index Last Key为(b < 8)，由于是 < 符号，因此提取b之后结束。

2.Index Filter

在完成Index Key的提取之后，我们根据where条件固定了索引的查询范围，但是此范围中的项，并不都是满足查询条件的项。在上面的SQL用例中，(3,1,1)，(6,4,4)均属于范围中，但是又均不满足SQL的查询条件。

Index Filter的提取规则：同样从索引列的第一列开始，检查其在where条件中是否存在：若存在并且where条件仅为 =，则跳过第一列继续检查索引下一列，下一索引列采取与索引第一列同样的提取规则；若where条件为 >=、>、<、<= 其中的几种，则跳过索引第一列，将其余where条件中索引相关列全部加入到Index Filter之中；若索引第一列的where条件包含 =、>=、>、<、<= 之外的条件，则将此条件以及其余where条件中索引相关列全部加入到Index Filter之中；若第一列不包含查询条件，则将所有索引相关条件均加入到Index Filter之中。

针对上面的用例SQL，索引第一列只包含 >=、< 两个条件，因此第一列可跳过，将余下的c、d两列加入到Index Filter中。因此获得的Index Filter为 c > 1 and d != 4 。

3.Table Filter

Table Filter是最简单，最易懂，也是提取最为方便的。提取规则：所有不属于索引列的查询条件，均归为Table Filter之中。

同样，针对上面的用例SQL，Table Filter就为 e != ‘a’。

Index Key/Index Filter/Table Filter小结

SQL语句中的where条件，使用以上的提取规则，最终都会被提取到Index Key (First Key & Last Key)，Index Filter与Table Filter之中。

Index First Key，只是用来定位索引的起始范围，因此只在索引第一次Search Path(沿着索引B+树的根节点一直遍历，到索引正确的叶节点位置)时使用，一次判断即可；

Index Last Key，用来定位索引的终止范围，因此对于起始范围之后读到的每一条索引记录，均需要判断是否已经超过了Index Last Key的范围，若超过，则当前查询结束；

Index Filter，用于过滤索引查询范围中不满足查询条件的记录，因此对于索引范围中的每一条记录，均需要与Index Filter进行对比，若不满足Index Filter则直接丢弃，继续读取索引下一条记录；

Table Filter，则是最后一道where条件的防线，用于过滤通过前面索引的层层考验的记录，此时的记录已经满足了Index First Key与Index Last Key构成的范围，并且满足Index Filter的条件，回表读取了完整的记录，判断完整记录是否满足Table Filter中的查询条件，同样的，若不满足，跳过当前记录，继续读取索引的下一条记录，若满足，则返回记录，此记录满足了where的所有条件，可以返回给前端用户。

结语

在读完、理解了以上内容之后，详细大家对于数据库如何提取where中的查询条件，如何将where中的查询条件提取为Index Key，Index Filter，Table Filter有了深刻的认识。以后在撰写SQL语句时，可以对照表的定义，尝试自己提取对应的where条件，与最终的SQL执行计划对比，逐步强化自己的理解。

同时，我们也可以回答文章开始提出的一个问题：MySQL 5.6中引入的Index Condition Pushdown，究竟是将什么Push Down到索引层面进行过滤呢？对了，答案是Index Filter。在MySQL 5.6之前，并不区分Index Filter与Table Filter，统统将Index First Key与Index Last Key范围内的索引记录，回表读取完整记录，然后返回给MySQL Server层进行过滤。而在MySQL 5.6之后，Index Filter与Table Filter分离，Index Filter下降到InnoDB的索引层面进行过滤，减少了回表与返回MySQL Server层的记录交互开销，提高了SQL的执行效率。

此篇文章属于转载的，对理解查询这块帮助比较大，所以拿过来用了。

来源于：`http://hedengcheng.com/?p=577`

Mysql系列目录

1. 第1篇：mysql基础知识
2. 第2篇：详解mysql数据类型（重点）
3. 第3篇：管理员必备技能(必须掌握)
4. 第4篇：DDL常见操作
5. 第5篇：DML操作汇总（insert,update,delete）
6. 第6篇：select查询基础篇
7. 第7篇：玩转select条件查询，避免采坑
8. 第8篇：详解排序和分页(order by & limit)
9. 第9篇：分组查询详解（group by & having）
10. 第10篇：常用的几十个函数详解
11. 第11篇：深入了解连接查询及原理
12. 第12篇：子查询
13. 第13篇：细说NULL导致的神坑，让人防不胜防
14. 第14篇：详解事务
15. 第15篇：详解视图
16. 第16篇：变量详解
17. 第17篇：存储过程&自定义函数详解
18. 第18篇：流程控制语句
19. 第19篇：游标详解

20. **第20篇：异常捕获及处理详解**
21. **第21篇：什么是索引?**
22. **第22篇：mysql索引原理详解**
23. **第23篇：mysql索引管理详解**
24. **第24篇：如何正确的使用索引?**

关注公众号：大侠学JAVA获取更多高质量资料

第26篇：聊聊如何使用MySQL实现分布式锁

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第26篇。

本篇我们使用mysql实现一个分布式锁。

分布式锁的功能

1. 分布式锁使用者位于不同的机器中，锁获取成功之后，才可以对共享资源进行操作
2. 锁具有重入的功能：即一个使用者可以多次获取某个锁
3. 获取锁有超时的功能：即在指定的时间内去尝试获取锁，超过了超时时间，如果还未获取成功，则返回获取失败
4. 能够自动容错，比如：A机器获取锁lock1之后，在释放锁lock1之前，A机器挂了，导致锁lock1未释放，结果会lock1一直被A机器占有着，遇到这种情况时，分布式锁要

能够自动解决，可以这么做：持有锁的时候可以加个持有超时时间，超过了这个时间还未释放的，其他机器将有机会获取锁

预备技能：乐观锁

通常我们修改表中一条数据过程如下：

t1：select获取记录R1
t2：对R1进行编辑
t3：update R1

我们来看一下上面的过程存在的问题：

如果A、B两个线程同时执行到t1，他们俩看到的R1的数据一样，然后都对R1进行编辑，然后去执行t3，最终2个线程都会更新成功，后面一个线程会把前面一个线程update的结果给覆盖掉，这就是并发修改数据存在的问题。

我们可以在表中新增一个版本号，每次更新数据时候将版本号作为条件，并且每次更新时候版本号+1，过程优化一下，如下：

t1：打开事务start transaction
t2：select获取记录R1,声明变量v=R1.version
t3：对R1进行编辑
t4：执行更新操作

```
update R1 set version = version + 1 where user_id=#user_id# and version = #v#;
```

t5：t4中的update会返回影响的行数，我们将其记录在count中，然后根据count来判断提交还是回滚

```
if(count==1){
    //提交事务
    commit;
}else{
    //回滚事务
    rollback;
}
```

上面重点在于步骤t4，当多个线程同时执行到t1，他们看到的R1是一样的，但是当他们执行到t4的时候，数据库会对update的这行记录加锁，确保并发情况下排队执行，所以只有第一个的update会返回1，其他的update结果会返回0，然后后面会判断count是否为1，进而对事务进行提交或者回滚。可以通过count的值知道修改数据是否成功了。

上面这种方式就乐观锁。**我们可以通过乐观锁的方式确保数据并发修改过程中的正确性。**

使用mysql实现分布式锁

建表

我们创建一个分布式锁表，如下

```
DROP DATABASE IF EXISTS javacode2018;
CREATE DATABASE javacode2018;
USE javacode2018;
DROP TABLE IF EXISTS t_lock;
create table t_lock(
  lock_key varchar(32) PRIMARY KEY NOT NULL COMMENT '锁唯一标志',
  request_id varchar(64) NOT NULL DEFAULT '' COMMENT '用来标识请求对象
的',
  lock_count INT NOT NULL DEFAULT 0 COMMENT '当前上锁次数',
  timeout BIGINT NOT NULL DEFAULT 0 COMMENT '锁超时时间',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
)COMMENT '锁信息表';
```

分布式锁工具类：

```
package com.itsoku.sql;

import lombok.Builder;
import lombok.Getter;
import lombok.Setter;
import lombok.extern.slf4j.Slf4j;
import org.junit.Test;

import java.sql.*;
import java.util.Objects;
import java.util.UUID;
```

```java
import java.util.concurrent.TimeUnit;


/**
 * 工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活！
 * 关注公众号：大侠学JAVA获取更多高质量资料
 */
@Slf4j
public class LockUtils {

    //将requestid保存在该变量中
    static ThreadLocal<String> requestIdTL = new ThreadLocal<>();

    /**
     * 获取当前线程requestid
     *
     * @return
     */
    public static String getRequestId() {
        String requestId = requestIdTL.get();
        if (requestId == null || "".equals(requestId)) {
            requestId = UUID.randomUUID().toString();
            requestIdTL.set(requestId);
        }
        log.info("requestId:{}", requestId);
        return requestId;
    }

    /**
     * 获取锁
     *
     * @param lock_key     锁key
     * @param locktimeout(毫秒) 持有锁的有效时间，防止死锁
     * @param gettimeout(毫秒)  获取锁的超时时间，这个时间内获取不到将重试
     * @return
     */
    public static boolean lock(String lock_key, long locktimeout, int gettimeout) throws Exception {
```

```
        log.info("start");
        boolean lockResult = false;
        String request_id = getRequestId();
        long starttime = System.currentTimeMillis();
        while (true) {
            LockModel lockModel = LockUtils.get(lock_key);
            if (Objects.isNull(lockModel)) {
                //插入一条记录,重新尝试获取锁

LockUtils.insert(LockModel.builder().lock_key(lock_key).request_id("")
.lock_count(0).timeout(0L).version(0).build());
            } else {
                String reqid = lockModel.getRequest_id();
                //如果reqid为空字符，表示锁未被占用
                if ("".equals(reqid)) {
                    lockModel.setRequest_id(request_id);
                    lockModel.setLock_count(1);
                    lockModel.setTimeout(System.currentTimeMillis() +
locktimeout);
                    if (LockUtils.update(lockModel) == 1) {
                        lockResult = true;
                        break;
                    }
                } else if (request_id.equals(reqid)) {
                    //如果request_id和表中request_id一样表示锁被当前线程持有
者，此时需要加重入锁
                    lockModel.setTimeout(System.currentTimeMillis() +
locktimeout);
                    lockModel.setLock_count(lockModel.getLock_count()
+ 1);
                    if (LockUtils.update(lockModel) == 1) {
                        lockResult = true;
                        break;
                    }
                } else {
                    //锁不是自己的，并且已经超时了，则重置锁，继续重试
                    if (lockModel.getTimeout() <
System.currentTimeMillis()) {
                        LockUtils.resetLock(lockModel);
                    } else {
```

```
                            //如果未超时，休眠100毫秒，继续重试
                            if (starttime + gettimeout >
System.currentTimeMillis()) {
                                TimeUnit.MILLISECONDS.sleep(100);
                            } else {
                                break;
                            }
                        }
                    }
                }
            }
            log.info("end");
            return lockResult;
        }

        /**
         *释放锁
         *
         * @param lock_key
         * @throws Exception
         */
        public static void unlock(String lock_key) throws Exception {
            //获取当前线程requestId
            String requestId = getRequestId();
            LockModel lockModel = LockUtils.get(lock_key);
            //当前线程requestId和库中request_id一致 && lock_count>0，表示可以释
放锁
            if (Objects.nonNull(lockModel) &&
requestId.equals(lockModel.getRequest_id()) &&
lockModel.getLock_count() > 0) {
                if (lockModel.getLock_count() == 1) {
                    //重置锁
                    resetLock(lockModel);
                } else {
                    lockModel.setLock_count(lockModel.getLock_count() -
1);
                    LockUtils.update(lockModel);
                }
            }
```

```
    }

    /**
     * 重置锁
     *
     * @param lockModel
     * @return
     * @throws Exception
     */
    public static int resetLock(LockModel lockModel) throws Exception
{
        lockModel.setRequest_id("");
        lockModel.setLock_count(0);
        lockModel.setTimeout(0L);
        return LockUtils.update(lockModel);
    }

    /**
     * 更新lockModel信息，内部采用乐观锁来更新
     *
     * @param lockModel
     * @return
     * @throws Exception
     */
    public static int update(LockModel lockModel) throws Exception {
        return exec(conn -> {
            String sql = "UPDATE t_lock SET request_id = ?,lock_count
= ?,timeout = ?,version = version + 1 WHERE lock_key = ? AND  version
= ?";
            PreparedStatement ps = conn.prepareStatement(sql);
            int colIndex = 1;
            ps.setString(colIndex++, lockModel.getRequest_id());
            ps.setInt(colIndex++, lockModel.getLock_count());
            ps.setLong(colIndex++, lockModel.getTimeout());
            ps.setString(colIndex++, lockModel.getLock_key());
            ps.setInt(colIndex++, lockModel.getVersion());
            return ps.executeUpdate();
        });
    }
```

```
    public static LockModel get(String lock_key) throws Exception {
        return exec(conn -> {
            String sql = "select * from t_lock t WHERE t.lock_key=?";
            PreparedStatement ps = conn.prepareStatement(sql);
            int colIndex = 1;
            ps.setString(colIndex++, lock_key);
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                return LockModel.builder().
                        lock_key(lock_key).
                        request_id(rs.getString("request_id")).
                        lock_count(rs.getInt("lock_count")).
                        timeout(rs.getLong("timeout")).
                        version(rs.getInt("version")).build();
            }
            return null;
        });
    }

    public static int insert(LockModel lockModel) throws Exception {
        return exec(conn -> {
            String sql = "insert into t_lock (lock_key, request_id,
lock_count, timeout, version) VALUES (?,?,?,?,?)";
            PreparedStatement ps = conn.prepareStatement(sql);
            int colIndex = 1;
            ps.setString(colIndex++, lockModel.getLock_key());
            ps.setString(colIndex++, lockModel.getRequest_id());
            ps.setInt(colIndex++, lockModel.getLock_count());
            ps.setLong(colIndex++, lockModel.getTimeout());
            ps.setInt(colIndex++, lockModel.getVersion());
            return ps.executeUpdate();
        });
    }

    public static <T> T exec(SqlExec<T> sqlExec) throws Exception {
        Connection conn = getConn();
        try {
            return sqlExec.exec(conn);
        } finally {
            closeConn(conn);
```

```
        }
    }

    @FunctionalInterface
    public interface SqlExec<T> {
        T exec(Connection conn) throws Exception;
    }

    @Getter
    @Setter
    @Builder
    public static class LockModel {
        private String lock_key;
        private String request_id;
        private Integer lock_count;
        private Long timeout;
        private Integer version;
    }

    private static final String url = "jdbc:mysql://localhost:3306/
javacode2018?useSSL=false";         //数据库地址
    private static final String username = "root";         //数据库用户名
    private static final String password = "root123";         //数据库密
码
    private static final String driver = "com.mysql.jdbc.Driver";
//mysql驱动

    /**
     *连接数据库
     *
     * @return
     */
    public static Connection getConn() {
        Connection conn = null;
        try {
            Class.forName(driver);  //加载数据库驱动
            try {
                conn = DriverManager.getConnection(url, username,
password);  //连接数据库
```

```
            } catch (SQLException e) {
                e.printStackTrace();
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return conn;
    }

    /**
     * 关闭数据库链接
     *
     * @return
     */
    public static void closeConn(Connection conn) {
        if (conn != null) {
            try {
                conn.close();  //关闭数据库链接
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

上面代码中实现了文章开头列的分布式锁的所有功能，大家可以认真研究下获取锁的方法：lock，释放锁的方法：unlock。

测试用例

```
package com.itsoku.sql;

import lombok.extern.slf4j.Slf4j;
import org.junit.Test;

import static com.itsoku.sql.LockUtils.lock;
import static com.itsoku.sql.LockUtils.unlock;

/**
 * 工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家
```

人过上更体面的生活！
 * 关注公众号：大侠学JAVA获取更多高质量资料
 */
```
@Slf4j
public class LockUtilsTest {

    //测试重复获取和重复释放
    @Test
    public void test1() throws Exception {
        String lock_key = "key1";
        for (int i = 0; i < 10; i++) {
            lock(lock_key, 10000L, 1000);
        }
        for (int i = 0; i < 9; i++) {
            unlock(lock_key);
        }
    }

    //获取之后不释放，超时之后被thread1获取
    @Test
    public void test2() throws Exception {
        String lock_key = "key2";
        lock(lock_key, 5000L, 1000);
        Thread thread1 = new Thread(() -> {
            try {
                try {
                    lock(lock_key, 5000L, 7000);
                } finally {
                    unlock(lock_key);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        thread1.setName("thread1");
        thread1.start();
        thread1.join();
    }
}
```

test1方法测试了重入锁的效果。

test2测试了主线程获取锁之后一直未释放，持有锁超时之后被thread1获取到了。

留给大家一个问题

上面分布式锁还需要考虑一个问题：比如A机会获取了key1的锁，并设置持有锁的超时时间为10秒，但是获取锁之后，执行了一段业务操作，业务操作耗时超过10秒了，此时机器B去获取锁时可以获取成功的，此时会导致A、B两个机器都获取锁成功了，都在执行业务操作，这种情况应该怎么处理？大家可以思考一下然后留言，我们一起讨论一下。

更多优质文章

1. **java高并发系列全集（34篇）**
2. **mysql高手系列（20多篇,高手必备）**
3. **聊聊db和缓存一致性常见的实现方式**

关注公众号：大侠学JAVA获取更多高质量资料

第27篇：MySQL如何确保数据不丢失的？有几点我们可以借鉴

Mysql系列的目标是：通过这个系列从入门到全面掌握一个高级开发所需要的全部技能。

这是Mysql系列第27篇。

本篇文章我们先来看一下**mysql**是如何确保数据不丢失的，通过本文我们可以了解**mysql**内部确保数据不丢失的原理，学习里面优秀的设计要点，然后我们再借鉴这些优秀的设计要点进行实践应用，加深理解。

预备知识

1. mysql内部是使用b+树的结构将数据存储在磁盘中，b+树中节点对应mysql中的页，mysql和磁盘交互的最小单位为页，页默认情况下为16kb，表中的数据记录存储在b+树的叶子节点中，当我们需要修改、删除、插入数据时，都需要按照页来对磁盘进行操作。

2. 磁盘顺序写比随机写效率要高很多，通常我们使用的是机械硬盘，机械硬盘写数据的时候涉及磁盘寻道、磁盘旋转寻址、数据写入的时间，耗时比较长，如果是顺序写，省去了寻道和磁盘旋转的时间，效率会高几个数量级。

3. 内存中数据读写操作比磁盘中数据读写操作速度高好多个数量级。

mysql确保数据不丢失原理分析

我们来思考一下，下面这条语句的执行过程是什么样的：

```
start transaction;
update t_user set name = '路人甲Java' where user_id = 666;
commit;
```

按照正常的思路，通常过程如下：

1. 找到user_id=666这条记录所在的页p1，将p1从磁盘加载到内存中

2. 在内存中对p1中user_id=666这条记录信息进行修改

3. mysql收到commit指令

4. 将p1页写入磁盘

5. 给客户端返回更新成功

上面过程可以确保数据被持久化到了磁盘中。

我们将需求改一下，如下：

```
start transaction;
update t_user set name = '路人甲Java' where user_id = 666;
update t_user set name = 'javacode2018' where user_id = 888;
commit;
```

来看一下处理过程：

1. 找到user_id=666这条记录所在的页p1，将p1从磁盘加载到内存中
2. 在内存中对p1中user_id=666这条记录信息进行修改
3. 找到user_id=888这条记录所在的页p2，将p2从磁盘加载到内存中
4. 在内存中对p2中user_id=888这条记录信息进行修改
5. mysql收到commit指令
6. 将p1页写入磁盘
7. 将p2页写入磁盘
8. 给客户端返回更新成功

上面过程我们看有什么问题

1. 假如6成功之后，mysql宕机了，此时p1修改已写入磁盘，但是p2的修改还未写入磁盘，最终导致user*id=666*的记录被修改成功了，*user*id=888的数据被修改失败了，数据是有问题的
2. 上面p1和p2可能位于磁盘的不同位置，涉及到磁盘随机写的问题，导致整个过程耗时也比较长

上面问题可以归纳为2点：无法确保数据可靠性、随机写导致耗时比较长。

关于上面问题，我们看一下mysql是如何优化的，mysql内部引入了一个redo log，这是一个文件，对于上面2条更新操作，mysql实现如下：

> mysql内部有个redo log buffer，是内存中一块区域，我们将其理解为数组结构，向redo log文件中写数据时，会先将内容写入redo log buffer中，后续会将这个buffer中的内容写入磁盘中的redo log文件，这个redo log buffer是整个mysql中所有连接共享的内存区域，可以被重复使用。

1. mysql收到start transaction后，生成一个全局的事务编号trx*id*，比如*trx*id=10
2. user*id=666*这个记录我们就叫*r1*，*user*id=888这个记录叫r2
3. 找到r1记录所在的数据页p1，将其从磁盘中加载到内存中
4. 在内存中找到r1在p1中的位置，然后对p1进行修改（这个过程可以描述为：将p1中的pos*start1*到*pos*start2位置的值改为v1），这个过程我们记为rb1(内部包含事务编号trx_id)，将rb1放入redo log buffer数组中，此时p1的信息在内存中被修改了，和磁盘中p1的数据不一样了
5. 找到r2记录所在的数据页p2，将其从磁盘中加载到内存中
6. 在内存中找到r2在p2中的位置，然后对p2进行修改（这个过程可以描述为：将p2中的pos*start1*到*pos*start2位置的值改为v2），这个过程我们记为rb2(内部包含事务编号trx_id)，将rb2放入redo log buffer数组中，此时p2的信息在内存中被修改了，和磁盘中p2的数据不一样了
7. 此时redo log buffer数组中有2条记录[rb1,rb2]
8. mysql收到commit指令
9. 将redo log buffer数组中内容写入到redo log文件中，写入的内容：

```
1.start trx=10;
2.写入rb1
3.写入rb2
4.end trx=10;
```

10. 返回给客户端更新成功。

上面过程执行完毕之后，数据是这样的：

1. 内存中p1、p2页被修改了，还未同步到磁盘中，此时内存中数据页和磁盘中数据页是不一致的，此时内存中数据页我们称为脏页
2. 对p1、p2页修改被持久到磁盘中的redolog文件中了，不会丢失

认真看一下上面过程中第9步骤，一个成功的事务记录在redo log中是有start和end的，redo log文件中如果一个trx_id对应start和end成对出现，说明这个事务执行成功了，如果只有start没有end说明是有问题的。

那么对p1、p2页的修改什么时候会同步到磁盘中呢？

redo log是mysql中所有连接共享的文件，对mysql执行insert、delete和上面update的过程类似，都是先在内存中修改页数据，然后将修改过程持久化到redo log所在的磁盘文件中，然后返回成功。redo log文件是有大小的，需要重复利用的（redo log有多个，多个之间采用环形结构结合几个变量来做到重复利用，这块知识不做说明，有兴趣的可以去网上找一下），当**redo log满了，或者系统比较闲的时候**，会对redo log文件中的内容进行处理，处理过程如下：

1. 读取redo log信息，读取一个完整的trx_id对应的信息，然后进行处理
2. 比如读取到了trx_id=10的完整内容，包含了start end，表示这个事务操作是成功的，然后继续向下
3. 判断p1在内存中是否存在，如果存在，则直接将p1信息写到p1所在的磁盘中；如果p1在内存中不存在，则将p1从磁盘加载到内存，通过redo log中的信息在内存中对p1进行修改，然后将其写到磁盘中

   上面的update之后，p1在内存中是存在的，并且p1是已经被修改过的，可以直接刷新到磁盘中。

如果上面的update之后，mysql宕机，然后重启了，p1在内存中是不存在的，此时系统会读取redo log文件中的内容进行恢复处理。

4. 将redo log文件中trx_id=10的占有的空间标记为已处理，这块空间会被释放出来可以重复利用了

5. 如果第2步读取到的trx_id对应的内容没有end，表示这个事务执行到一半失败了（可能是第9步骤写到一半宕机了），此时这个记录是无效的，可以直接跳过不用处理

上面的过程做到了：数据最后一定会被持久化到磁盘中的页中，不会丢失，做到了可靠性。

并且内部采用了先把页的修改操作先在内存中进行操作，然后再写入了redo log文件，此处redo log是按顺序写的，使用到了io的顺序写，效率会非常高，相对于用户来说响应会更快。

对于将数据页的变更持久化到磁盘中，此处又采用了异步的方式去读取redo log的内容，然后将页的变更刷到磁盘中，这块的设计也非常好，异步刷盘操作！

但是有一种情况，当一个事务commit的时候，刚好发现redo log不够了，此时会先停下来处理redo log中的内容，然后在进行后续的操作，遇到这种情况时，整个事物响应会稍微慢一些。

mysql中还有一个binlog，在事务操作过程中也会写binlog，先说一下binlog的作用，binlog中详细记录了对数据库做了什么操作，算是对数据库操作的一个流水，这个流水也是相当重要的，主从同步就是使用binlog来实现的，从库读取主库中binlog的信息，然后在从库中执行，最后，从库就和主库信息保持同步一致了。还有一些其他系统也可以使用binlog的功能，比如可以通过binlog来实现bi系统中etl的功能，将业务数据抽取到数据仓库，**阿里提供了一个java版本的项目：canal**，这个项目可以模拟从库从主库读取binlog的功能，也就是说可以通过java程序来监控数据库详细变化的流水，这个大家可以脑洞大开一下，可以做很多事情的，有兴趣的朋友可以去研究一下；所以binlog对mysql来说也是相当重要的，我们来看一下系统如何确保redo log 和binlog在一致性的，都写入成功的。

还是以update为例：

```
start transaction;
update t_user set name = '路人甲Java' where user_id = 666;
update t_user set name = 'javacode2018' where user_id = 888;
commit;
```

> 一个事务中可能有很多操作，这些操作会写很多binlog日志，为了加快写的速度，mysql先把整个过程中产生的binlog日志先写到内存中的binlog cache缓存中，后面再将binlog cache中内容一次性持久化到binlog文件中。一个事务的binlog是不能被拆开的，因此不论这个事务多大，也要确保一次性写入。这就涉及到了 binlog cache 的保存问题。系统给 binlog cache 分配了一片内存，每个线程一个，参数 binlog*cache*size 用于控制单个线程内 binlog cache 所占内存的大小。如果超过了这个参数规定的大小，就要暂存到磁盘。

过程如下：

1. mysql收到start transaction后，生成一个全局的事务编号trx*id*，比如*trx*id=10
2. user*id=666*这个记录我们就叫*r1*，*user*id=888这个记录叫r2
3. 找到r1记录所在的数据页p1，将其从磁盘中加载到内存中
4. 在内存中对p1进行修改
5. 将p1修改操作记录到redo log buffer中
6. 将p1修改记录流水记录到binlog cache中
7. 找到r2记录所在的数据页p2，将其从磁盘中加载到内存中
8. 在内存中对p2进行修改
9. 将p2修改操作记录到redo log buffer中
10. 将p2修改记录流水记录到binlog cache中
11. mysql收到commit指令

12. 将redo log buffer携带trx_id=10写入到redo log文件，持久化到磁盘，**这步操作叫做redo log prepare**，内容如下

    1.start trx=10; 2.写入rb1 3.写入rb2 4.prepare trx=10;

    注意上面是prepare了，不是之前说的end了。
13. 将binlog cache携带trx_id=10写入到binlog文件，持久化到磁盘
14. 向redo log中写入一条数据：`end trx=10;`表示redo log中这个事务完成了，**这步操作叫做redo log commit**
15. 返回给客户端更新成功

我们来分析一下上面过程可能出现的一些情况：

步骤10操作完成后，mysql宕机了

宕机之前，所有修改都位于内存中，mysql重启之后，内存修改还未同步到磁盘，对磁盘数据没有影响，所以无影响。

步骤12执行完毕之后，mysql宕机了

此时redo log prepare过程是写入redo log文件了，但是binlog写入失败了，此时mysql重启之后会读取redo log进行恢复处理，查询到trx*id=10*的记录是*prepare*状态，会去*binlog*中查找*trx*id=10的操作在binlog中是否存在，如果不存在，说明binlog写入失败了，此时可以将此操作回滚

步骤13执行完毕之后，mysql宕机

此时redo log prepare过程是写入redo log文件了，但是binlog写入失败了，此时mysql重启之后会读取redo log进行恢复处理，查询到trx*id=10*的记录是*prepare*状态，会去*binlog*中查找*trx*id=10的操作在binlog是存在的，然后接着执行上面的步骤14和15.

做一个总结

上面的过程设计比较好的地方，有2点

日志先行，io顺序写，异步操作，做到了高效操作

> 对数据页，先在内存中修改，然后使用io顺序写的方式持久化到redo log文件；然后异步去处理redo log，将数据页的修改持久化到磁盘中，效率非常高，整个过程，其实就是 MySQL 里经常说到的 WAL 技术，WAL 的全称是 Write-Ahead Logging，它的关键点就是先写日志，再写磁盘。

两阶段提交确保redo log和binlog一致性

> 为了确保redo log和binlog一致性，此处使用了二阶段提交技术，redo log 和binlog的写分了3步走：

1. 携带trx_id，redo log prepare到磁盘
2. 携带trx_id，binlog写入磁盘
3. 携带trx_id，redo log commit到磁盘

   上面3步骤，可以确保同一个trx_id关联的redo log 和binlog的可靠性。

关于上面2点优秀的设计，我们平时开发的过程中也可以借鉴，下面举2个常见的案例来学习一下。

案例：电商中资金账户高频变动解决方案

电商中有账户表和账户流水表，2个表结构如下：

```
drop table IF EXISTS t_acct;
create table t_acct(
  acct_id int primary key NOT NULL COMMENT '账户id',
  balance decimal(12,2) NOT NULL COMMENT '账户余额',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
)COMMENT '账户表';
```

```
drop table IF EXISTS t_acct_data;
create table t_acct_data(
  id int AUTO_INCREMENT PRIMARY KEY COMMENT '编号',
  acct_id int primary key NOT NULL COMMENT '账户id',
  price DECIMAL(12,2) NOT NULL COMMENT '交易额',
  open_balance decimal(12,2) NOT NULL COMMENT '期初余额',
  end_balance decimal(12,2) NOT NULL COMMENT '期末余额'
) COMMENT '账户流水表';

INSERT INTO t_acct(acct_id, balance, version) VALUES (1,10000,0);
```

上面向账户表t*acct*插入了一条数据，余额为*10000*，当我们下单成功或者充值的时候，会对上面*2*个表进行操作，会修改tacct的数据，顺便向t*acct*data表写一条流水，这个t*acct*data表有个期初和期末的流水，关系如下：

```
end_balance = open_balance + price;
```

open_balance为操作业务时，t_acct表的balance的值。

如给账户1充值100，过程如下：

```
t1：开启事务：start transaction;
t2：R1 = (select * from t_acct where acct_id = 1);
t3：创建几个变量
      v_balance = R1.balance;
t4：update t_acct set balnce = v_balance+100,version = version + 1
where acct_id = 1;
t5：insert into t_acct_data(acct_id,price,open_balnace,end_balance)
    values (1,100,#v_balance#,#v_balance+100#)
t6：提交事务：commit;
```

分析一下上面过程存在的问题：

我们开启2个线程【thread1、thread2】模拟分别充值100，正常情况下数据应该是这样的：

```
t_acct表记录：
(1,10200,1);
t_acct_data表产生2条数据：
```

```
(1,100,10000,10100);
(2,100,10100,10200);
```

但是当2个线程同时执行到t2的时候获取R1记录信息是一样的，变量v_balance的值也一样的，最后执行完成之后，数据变成了下面这样：

```
t_acct表：1，10200
t_acct_data表产生2条数据：
1,100,10000,10100;
2,100,10100,10100;
```

导致t_acct_data产生的2条数据是一样的，这种情况是有问题的，这就是并发导致的问题。

上篇文章中有说道乐观锁可以解决这种并发问题，有兴趣的可以去看一下，过程如下：

```
t1：打开事务start transaction
t2：R1 = (select * from t_acct where acct_id = 1);
t3：创建几个变量
    v_version = R1.version;
      v_balance = R1.balance;
      v_open_balance = v_balance;
      v_balance = R1.balance + 100;
      v_open_balance = v_balance;
t3：对R1进行编辑
t4：执行更新操作
      int count = (update t_acct set balance = #v_balance#,version =
version + 1 where acct_id = 1 and version = #v_version#);
t5：if(count==1){
          //向t_acct_data表写入数据
          insert into
t_acct_data(acct_id,price,open_balnace,end_balance) values
(1,100,#v_open_balance#,#v_open_balance#)
        //提交事务
        commit;
    }else{
        //回滚事务
        rollback;
    }
```

上面的过程中，如果2个线程同时执行到t2看到的R1数据是一样的，但是最后走到t4的时候会被数据库加锁，2个线程的update在mysql中会排队执行，最后只有一个update的结果返回的影响行数是1，然后根据t5，会有一个会被回滚，另外一个被提交，避免了并发导致的问题。

我们分析一下上面过程会有什么问题?

刚才上面也提到了，并发量大的时候，只有部分会成功，比如10个线程同时执行到t2的时候，其中只有1个会成功，其他9个都会失败，**并发量大的情况下失败的概率比较高**，这个大家可以并发测试一下，失败率很高，下面我们继续优化。

分析一下问题主要出现在写t*acct*data上面，如果没有这个表的操作，我们直接用一个update就完成了操作，速度是非常快的，上面我们学到的了mysql中先写日志，然后异步刷盘的方式，此处我们也可以采用这种思路，先记录一条交易日志，然后异步根据交易日志将交易流水写到t_acct_data表中。

那我们继续优化，新增一个账户操作日志表:

```
drop table IF EXISTS t_acct_log;
create table t_acct_log(
  id INT AUTO_INCREMENT PRIMARY KEY COMMENT '编号',
  acct_id int primary key NOT NULL COMMENT '账户id',
  price DECIMAL(12,2) NOT NULL COMMENT '交易额',
  status SMALLINT NOT NULL DEFAULT 0 COMMENT '状态,0:待处理，1：处理成功'
) COMMENT '账户操作日志表';
```

顺便对t*acct*标做一下改造，新增一个字段`*old*balance`，新结构如下:

```
drop table IF EXISTS t_acct;
create table t_acct(
  acct_id int primary key NOT NULL COMMENT '账户id',
  balance decimal(12,2) NOT NULL COMMENT '账户余额',
  old_balance decimal(12,2) NOT NULL COMMENT '账户余额(老的值)',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
)COMMENT '账户表';

INSERT INTO t_acct(acct_id, balance,old_balance,version) VALUES
(1,10000,10000,0);
```

新增了一个old_balance字段，这个字段的值刚开始的时候和balance的值是一致的，后面会在job中进行改变，可以先向下看，后面有解释

假设账户v*acct*id交易金额为v_price，过程如下：

```
t1.开启事务：start transaction;
t2.insert into t_acct_log(acct_id,price,status) values
(#v_acct_id#,#v_price#,0)
t3.int count = (update t_acct set balnce = v_balance+#v_price#,version
= version+1 where acct_id = #v_acct_id# and v_balance+#v_price#>=0);
t6.if(count==1){
        //提交事务
        commit;
    }else{
        //回滚事务
        rollback;
    }
```

可以看到上面没有记录流水了，变成插入了一条日志t_acct_log，后面我们异步根据t_acct_log的数据来生成t_acct_data记录。

上面这个操作支撑并发操作还是比较高的，测试了一下每秒500笔，并且都成功了，效率非常高。

新增一个job，查询t*acct*log中状态为0的记录，然后遍历进行一个个处理，处理过程如下：

```
假设t_acct_log中当前需要处理的记录为L1
t1：打开事务start transaction
t2：创建变量
     v_price = L1.price;
     v_acct_id = L1.acct_id;
t3：R1 = (select * from t_acct where acct_id = #v_acct_id#);
t4：创建几个变量
     v_old_balance = R1.old_balance;
     v_open_balance = v_old_balance;
     v_old_balance = R1.old_balance + v_price;
     v_open_balance = v_old_balance;
t5：int count = (update t_acct set old_balance =
```

```
#v_old_balance#,version = version + 1 where acct_id = #v_acct_id# and
version = #v_version#);
t6: if(count==1){
          //更新t_acct_log的status置为1
          count = (update t_acct_log set status=1 where status=0 and id
= #L1.id#);
      }

      if(count==1){
         //提交事务
         commit;
     }else{
         //回滚事务
         rollback;
     }
```

上面t5中update条件中加了version，t6中的update条件中加了status=0的操作，主要是为了防止并发操作修改可能会出错的问题。

上面t*acct*log中所有status=0的记录被处理完毕之后，t*acct*表中的*balance*和*old*balance会变为一致。

上面这种方式采用了先写账户操作日志，然后异步对日志进行操作，在生成流水，借鉴了mysql中的设计，大家也可以学习学习。

案例2：跨库转账问题

此处我们使用mysql上面介绍的二阶段提交来解决。

如从A库的T1表转100到B库的T1表。

我们创建一个C库，在C库新增一个转账订单表，如：

```
drop table IF EXISTS t_transfer_order;
create table t_transfer_order(
  id int NOT NULL AUTO_INCREMENT primary key COMMENT '账户id',
  from_acct_id int NOT NULL COMMENT '转出方账户',
  to_acct_id int NOT NULL COMMENT '转入方账户',
```

```
  price decimal(12,2) NOT NULL COMMENT '转账金额',
  addtime int COMMENT '入库时间（秒）',
  status SMALLINT NOT NULL DEFAULT 0 COMMENT '状态，0：待处理，1：转账成
功，2：转账失败',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
) COMMENT '转账订单表';
```

A、B库加3张表，如：

```
drop table IF EXISTS t_acct;
create table t_acct(
  acct_id int primary key NOT NULL COMMENT '账户id',
  balance decimal(12,2) NOT NULL COMMENT '账户余额',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
)COMMENT '账户表';

drop table IF EXISTS t_order;
create table t_order(
  transfer_order_id int primary key NOT NULL COMMENT '转账订单id',
  price decimal(12,2) NOT NULL COMMENT '转账金额',
  status SMALLINT NOT NULL DEFAULT 0 COMMENT '状态，1：转账成功，2：转账失
败',
  version INT NOT NULL DEFAULT 0 COMMENT '版本号，每次更新+1'
) COMMENT '转账订单表';

drop table IF EXISTS t_transfer_step_log;
create table t_transfer_step_log(
  id int primary key NOT NULL COMMENT '账户id',
  transfer_order_id int NOT NULL COMMENT '转账订单id',
  step SMALLINT NOT NULL COMMENT '转账步骤，0：正向操作，1：回滚操作',
  UNIQUE KEY (transfer_order_id,step)
) COMMENT '转账步骤日志表';
```

> t_transfer_step_log表用于记录转账日志操作步骤的，transfer_order_id,step上加了唯一约束，表示每个步骤只能执行一次，可以确保步骤的幂等性。

定义几个变量：

v*from*acct_id:转出方账户

v*to*acct_id:转入方账户

v_price:交易金额

整个转账流程如下：

每个步骤都有返回值，返回值是数组类型的，含义是：0：处理中（结果未知），1：成功，2：失败

```
step1:创建转账订单,订单状态为0，表示处理中
C1: start transaction;
C2: insert into
t_transfer_order(from_acct_id,to_acct_id,price,addtime,status,version)

values(#v_from_acct_id#,#v_to_acct_id#,#v_price#,0,unix_timestamp(now(
)));
C3: 获取刚才insert成功的订单id，放在变量v_transfer_order_id中
C4: commit;

step2:A库操作如下
A1: AR1 = (select * from t_order where transfer_order_id =
#v_transfer_order_id#);
A2: if(AR1!=null){
      return AR1.status==1?1:2;
      }
A3: start transaction;
A4: AR2 = (select 1 from t_acct where acct_id = #v_from_acct_id#);
A5: if(AR2.balance<v_price){
         //表示余额不足，那转账肯定是失败了，插入一个转账失败订单
      insert into t_order (transfer_order_id,price,status) values
(#transfer_order_id#,#v_price#,2);
      commit;
      //返回失败的状态2
      return 2;
      }else{
      //通过乐观锁 & balance - #v_price# >= 0更新账户资金，防止并发操作
```

```
      int count = (update t_acct set balance = balance - #v_price#,
version = version + 1 where acct_id = #v_from_acct_id# and balance -
#v_price# >= 0 and version = #AR2.version#);
         //count为1表示上面的更新成功
      if(count==1){
            //插入转账成功订单，状态为1
           insert into t_order (transfer_order_id,price,status) values
(#transfer_order_id#,#v_price#,1);
            //插入步骤日志
            insert into t_transfer_step_log (transfer_order_id,step)
values (#v_transfer_order_id#,1);
            commit;
            return 1;
         }else{
            //插入转账失败订单，状态为2
            insert into t_order (transfer_order_id,price,status)
values (#transfer_order_id#,#v_price#,2);
            commit;
            return 2;
         }
      }

step3:
    if(step2的结果==1){
        //表示A库中扣款成功了
   执行step4;
    }else if(step2的结果==2){
        //表示A库中扣款失败了
   执行step6;
    }

step4:对B库进行操作，如下:
B1: BR1 = (select * from t_order where transfer_order_id =
#v_transfer_order_id#);
B2: if(BR1!=null){
    return BR1.status==1?1:2;
}else{
      执行B3;
```

```
}
B3: start transaction;
B4: BR2 = (select 1 from t_acct where acct_id = #v_to_acct_id#);
B5: int count = (update t_acct set balance = balance + #v_price#,
version = version + 1 where acct_id = #v_to_acct_id# and version =
#BR2.version#);
if(count==1){
    //插入订单，状态为1
    insert into t_order (transfer_order_id,price,status) values
(#transfer_order_id#,#v_price#,1);
    //插入日志
    insert into t_transfer_step_log (transfer_order_id,step) values
(#v_transfer_order_id#,1);
    commit;
    return 1;
}else{
    //进入到此处说明有并发，返回0
    rollback;
    return 0;
}

step5:
    if(step4的结果==1){
        //表示B库中加钱成功了
   执行step7;
    }

step6:对C库操作（转账失败，将订单置为失败）
C1: AR1 = (select 1 from t_transfer_order where id =
#v_transfer_order_id#);
C2: if(AR1.status==1 || AR1.status=2){
     return AR1.status=1?"转账成功":"转账失败";
     }
C3: start transaction;
C4: int count = (udpate t_transfer_order set status = 2,version =
version+1 where id = #v_transfer_order_id# and version = version +
#AR1.version#)
C5: if(count==1){
```

```
      commit;
      return "转账失败";
      }else{
      rollback;
      return "处理中";
      }
```

step7:对C库操作（转账成功，将订单置为成功）

```
C1: AR1 = (select 1 from t_transfer_order where id =
#v_transfer_order_id#);
C2: if(AR1.status==1 || AR1.status=2){
      return AR1.status=1?"转账成功":"转账失败";
      }
C3: start transaction;
C4: int count = (udpate t_transfer_order set status = 1,version =
version+1 where id = #v_transfer_order_id# and version = version +
#AR1.version#)
C5: if(count==1){
      commit;
      return "转账成功";
      }else{
      rollback;
      return "处理中";
      }
```

还需要新增一个补偿的job，处理C库中状态为0的超过10分钟的转账订单订单，过程如下：

```
while(true){
      List list = select * from t_transfer_order where status = 0 and
addtime+10*60<unix_timestamp(now());
      if(list为空){
            //插叙无记录，退出循环
            break;
      }
      //循环遍历list进行处理
      for(Object r:list){
            //调用上面的steap2进行处理，最终订单状态会变为1或者2
```

```
        }
}
```

说一下：这个job的处理有不好的地方，可能会死循环，这个留给大家去思考一下，如何解决？欢迎留言

Mysql系列目录

1. 第1篇：**mysql基础知识**
2. 第2篇：详解**mysql数据类型（重点）**
3. 第3篇：管理员必备技能**(必须掌握)**
4. 第4篇：**DDL常见操作**
5. 第5篇：**DML操作汇总（insert,update,delete）**
6. 第6篇：**select查询基础篇**
7. 第7篇：玩转**select条件查询，避免采坑**
8. 第8篇：详解排序和分页**(order by & limit)**
9. 第9篇：分组查询详解（**group by & having**）
10. 第10篇：常用的几十个函数详解
11. 第11篇：深入了解连接查询及原理
12. 第12篇：子查询
13. 第13篇：细说**NULL**导致的神坑，让人防不胜防
14. 第14篇：详解事务
15. 第15篇：详解视图
16. 第16篇：变量详解

17. 第**17**篇：存储过程**&**自定义函数详解
18. 第**18**篇：流程控制语句
19. 第**19**篇：游标详解
20. 第**20**篇：异常捕获及处理详解
21. 第**21**篇：什么是索引?
22. 第**22**篇：**mysql**索引原理详解
23. 第**23**篇：**mysql**索引管理详解
24. 第**24**篇：如何正确的使用索引?
25. 第**25**篇：**sql**中**where**条件在数据库中提取与应用浅析
26. 第**26**篇：聊聊**mysql**如何实现分布式锁?