数据库学习

发布 *v1.0*

2020 年 12 月 13 日

Contents:

数据库相关学习笔记，琐碎知识点和坑

CHAPTER 1

数据库 01mysql 常用操作速查

1.1 启动停止服务和设置

net start MySQL 服务名

net stop MySQL 服务名

mysql -h 主机名 -u 用户名 [-P 端口] -p

quit; 或 exit;

set names utf8;

mysqldump -u 用户名 -p 数据库名 > 文件名 (备份)

mysql -u 用户名 -p 数据库名 < 文件名 (还原数据库)

查看最大连接数:

show variables like '%max_connections%';

修改最大连接数

方法一: 修改配置文件。打开 MySQL 配置文件 my.ini 或 my.cnf 查找 max_connections=100 , 重起 MySQL.

方法二: 命令行修改. 命令行登录 MySQL 后。set global max_connections=200; 只在 MySQL 当前服务进程有效，一旦重启，会恢复到初始

1.2 数据库

状态.

show databases;

use 数据库名;

create datebase 数据库名;

drop database 数据库名;

select database();(显示当前正在使用的数据库)

1.3 数据表

create table 数据表名称 (字段名 1 字段名 1 数据类型, 字段名 2 字段名 2 数据类型,..., 字段名 n 字段名 n 数据类型);

show tables;

desc 数据表名称;

drop table 数据表名称;

alter table 要修改的数据表名称 rename 修改后的数据表名称;

alter table 数据表名称 modify 字段名字段名数据类型;(修改字段类型)

alter table 数据表名称 change 要修改的字段名修改后的字段名修改后的字段名数据类型;(字段名称并且修改其类型)

alter table 数据表名称 drop 字段名;

alter table 数据表名称 add 字段名字段名数据类型;

insert into 数据表名称 values (字段 1 的值, 字段 2 的值,..., 字段 n 的值);

说明:auto_increment 输入 NULL 字段值序号自增. 字符类型, 枚举类型, 日期加单引号.

select * from 数据表名称;

select 字段名 from 数据表名称;

delete from 数据表名称;

delete from 数据表名称 where 字段名 = ' 关键字';

update 数据表名称字段名 1 = 修改值 where 字段名 2 = ' 关键字';

根据已有的表创建新表:

A：create table tab_new like tab_old (使用旧表 B 创建新表 A)

备注：此种方式在将表 B 复制到 A 时候会将表 B 完整的字段结构和索引复制到表 A 中来

B：create table tab_new as select col1,col2…from tab_old definition only

备注：此种方式只会将表 B 的字段结构复制到表 A 中来，但不会复制表 B 中的索引到表 A 中来。这种方式比较灵活可以在复制原表表结构的同时指定要复制哪些字段，并且自身复制表也可以根据需要增加字段结构。

结论：

create table as select 会将原表中的数据完整复制一份，但表结构中的索引会丢失。

create table like 只会完整复制原表的建表语句，但不会复制数据

两种方式在复制表的时候均不会复制权限对表的设置。比如说原本对表 B 做了权限设置，复制后，表 A 不具备类似于表 B 的权限。

添加主键：Alter table tabname add primary key(col)

说明：删除主键：Alter table tabname drop primary key

备注：一个数据表只可以有一个主键，所以不存在删除某一列的主键。

创建测试表

```
CREATE TABLE `test_number` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `number` int(11) unsigned NOT NULL DEFAULT '0' COMMENT ' 数字',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

视图

```
# 查看创建视图语句
SHOW CREATE VIEW v;
# 更改视图
CREATE OR REPLACE VIEW v AS SELECT name, age FROM n;
ALTER VIEW v AS SELECT name FROM n ;
# 删除视图
DROP VIEW IF EXISTS v;
```

其他

```
# 数据库备份
mysqldump -u root -p db_name > file.sql
mysqldump -u root -p db_name table_name > file.sql
# 数据库还原
mysql -u root -p < C:\file.sql
```

1.4 导入导出

```
select * from train\_user\_bt  into outfile 'd:/test.csv'
fields terminated by ','
lines terminated by '\\r\\n';
```

1.5 进阶查询

几个高级查询运算词

A：UNION 运算符

UNION 运算符通过组合其他两个结果表（例如 TABLE1 和 TABLE2）并消去表中任何重复行而派生出一个结果表。当 ALL 随 UNION 一起使用时（即 UNION ALL），不消除重复行。两种情况下，派生表的每一行不是来自 TABLE1 就是来自 TABLE2。B：EXCEPT 运算符

EXCEPT 运算符通过包括所有在 TABLE1 中但不在 TABLE2 中的行并消除所有重复行而派生出一个结果表。当 ALL 随 EXCEPT 一起使用时 (EXCEPT ALL)，不消除重复行。

C：INTERSECT 运算符

INTERSECT 运算符通过只包括 TABLE1 和 TABLE2 中都有的行并消除所有重复行而派生出一个结果表。当 ALL 随 INTERSECT 一起使用时 (INTERSECT ALL)，不消除重复行。

注：使用运算词的几个查询结果行必须是一致的。

1.6 truncate 与 delete 的区别

a. 事务：truncate 是不可以 rollback 的，但是 delete 是可以 rollback 的；

原因：truncate 删除整表数据 (ddl 语句, 隐式提交)，delete 是一行一行的删除，可以 rollback

b. 效果：truncate 删除后将重新水平线和索引 (id 从零开始) ,delete 不会删除索引

c.truncate 不能触发任何 Delete 触发器。

d.delete 删除可以返回行数

1.7 参考

MySQL SQL 命令速查：https://segmentfault.com/a/1190000007507732

MySQL 速查手册：http://blog.chinaunix.net/attachment/attach/26/22/61/0626226106fb9b53d3cb8bd8d6ad96f0d50618092e.pdf

mysql sql 常用语句大全：https://www.cnblogs.com/cy568searchx/p/3747993.html

row_number() over partition by 分组聚合：https://www.cnblogs.com/starzy/p/11146156.html

MYSQL 基础常见常用语句 200 条:https://blog.csdn.net/c361604199/article/details/79479398

MySQL 必备的常见知识点汇总整理 (实例, 事务 acid, 事务隔离讲解较好等):https://www.jb51.net/article/185982.htm

CHAPTER 2

数据库 02mongo 异常错误

2.1 mongo 报错 WiredTiger.wt, connection: /data/db/WiredTiger.wt: handle-open: open: Operation not permitted

表现：sudo mongod 可以成功启动 mongo，但是不加 sudo 则不行，自然不希望每次都加 sudo

完整报错：

```
 [initandlisten] MongoDB starting : pid=13900 port=27017 dbpath=/data/db 64-bit␣
↪host=john-P95-HP
 [initandlisten] db version v3.6.3
 [initandlisten] git version: 9586e557d54ef70f9ca4b43c26892cd55257e1a5
 [initandlisten] OpenSSL version: OpenSSL 1.1.1  11 Sep 2018
 [initandlisten] allocator: tcmalloc
 [initandlisten] modules: none
 [initandlisten] build environment:
 [initandlisten]     distarch: x86_64
 [initandlisten]     target_arch: x86_64
 [initandlisten] options: {}
[initandlisten] Detected data files in /data/db created by the 'wiredTiger' storage␣
↪engine, so setting the active storage engine to 'wiredTiger'.
[initandlisten]
[initandlisten] ** WARNING: Using the XFS filesystem is strongly recommended with the␣
↪WiredTiger storage engine
```

(下页继续)

(续上页)

```
 [initandlisten] **          See http://dochub.mongodb.org/core/prodnotes-filesystem
 [initandlisten] wiredtiger_open config: create,cache_size=7449M,session_max=20000,
↪eviction=(threads_min=4,threads_max=4),config_base=false,statistics=(fast),
↪log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_
↪idle_time=100000),statistics_log=(wait=0),verbose=(recovery_progress),
 [initandlisten] WiredTiger error (1) [1559653959:932022][13900:0x7f49416a70c0],␣
↪file:WiredTiger.wt, connection: /data/db/WiredTiger.wt: handle-open: open: Operation␣
↪not permitted
 [initandlisten] Assertion: 28595:1: Operation not permitted src/mongo/db/storage/
↪wiredtiger/wiredtiger_kv_engine.cpp 413
 [initandlisten] exception in initAndListen: Location28595: 1: Operation not permitted,␣
↪terminating
 [initandlisten] shutdown: going to close listening sockets...
 [initandlisten] removing socket file: /tmp/mongodb-27017.sock
 [initandlisten] now exiting
```

处理：

```
# storage.dbPath
sudo chown -R mongodb:mongodb /var/lib/mongodb

# systemLog.path
sudo chown -R mongodb:mongodb /var/log/mongodb
```

这里面的 mongodb:mongodb 分别是用户组和用户名，

可通过如下命令查询当前用户所属组，

```
groups xxxx：xxxx 是当前登录用户（一个用户可能属于多个组）
```

参　　考：https://stackoverflow.com/questions/43137250/mongodb-3-4-3-permission-denied-wiredtiger-kv-engine-cpp-267-error-with-ubuntu-1

CHAPTER 3

数据库 03mongo 占用磁盘空间过大

何为过大：mongodump 之前 2G，导入后变成 15G，大约 8 倍.

原因：如果 mongo 版本小于 3，则正常，mongo 请升级到 3.0 版本上，目前 3.6.7 较稳定版

3.1 错误安装方式 mongo2.x 版

如果您安装 mongodb 通过如下方式

```
sudo apt-get install mongodb
```

那么大概率是 2.7 版本的，这个版本就别用了，坑死人不偿命。

通过如下方式卸载，否则卸载不干净，会影响 3.x 版本的安装

```
sudo apt-get --purge remove mongodb mongodb-clients mongodb-server
```

执行:vim /etc/mongodb.conf

如果内容空（文件不存在）则说明卸载成功 (这个 2.x 版本配置文件典型位置)

3.2 安装 mongodb3.6 版

安装步骤:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv␣
↪2930ADAE8CAF5059EE73BB4B58712A2291FA4AD5
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.
↪6 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.6.list
sudo apt-get update
sudo apt-get install -y mongodb-org
```

启动 mongodb

```
1、创建配置文件:
sudo vim /etc/systemd/system/mongodb.service
2. 在里面追加文本
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
[Service]
User=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf
[Install]
WantedBy=multi-user.target
3. 启动服务，查看服务状态
sudo systemctl start mongodb

或常规启动方式:nohup mongod --bind_ip_all --config /etc/mongod.conf &
```

CHAPTER 4

数据库 04sqlite 转 mysql

基本思路:sqlite 导出 sql 保存到 mysql

1,sqlite 导出 sql, 手工替换

2,sqlite 导出 sql, 脚本替换

3,makemigrations 生成 mysql 表，数据:sqlite=>csv=>mysql

4,django 主从数据库机制，将 sqlite 同步到 mysql

4.1 sqlite 导出 sql

```
sqlite3 database.sqlite3
sqlite3> .output /path/to/dump.sql
sqlite3> .dump
sqlite3> .exit
```

or

```
sqlite3 database.sqlite3 .dump
```

4.2 01,sqlite=>mysql 语句修改

为了保证 SQL 语句的兼容，需要将 SQLite 的特有的格式，修改为 MySQL 的格式。下面为我总结的一般规则 (下面的方括号应被忽略)：

```
将 ["] 改为 [`]
也可以移除全部的 ["] ，但是如果有一些函数名作为字段名 (e.g. regexp) 时将会遇到错误
需要注意一些默认为 ["] ，其作用不在字段上的，不应被替换而应当被保留
移除所有的 [BEGIN TRANSACTION] [COMMIT] 以及 任何包含 [sqlite_sequence] 的 (整) 行
将所有 [autoincrement] 改为 [auto_increment]
将所有 ['f'] 改为 ['0'] 并将所有 ['t'] 改为 ['1']
或者 ['False'] 改为 ['0'] 及 ['True'] 改为 ['1']
text 字段不能设置 unipue，需改为 varchar(255)
注意 sqlite 是不区分类型的，所以有些整形字段和 text 字段要修改配合 mysql。
默认设为 CURRENT_TIMESTAMP 的字段类型一定为 TIMESTAMP。
```

将修改完的文件保存。

放弃，太麻烦且容易错，无法复用。

4.3 02,sqlite 转 mysql 格式脚本

01，https://github.com/vwbusguy/sqlite-to-mysql

02，（包含 2 不同脚本）Quick easy way to migrate SQLite3 to MySQL? [closed]:https://stackoverflow.com/questions/18671/quick-easy-way-to-migrate-sqlite3-to-mysql

以上脚本收集整合到 githubsqlite-to-mysql

可惜的是没有一个可以直接使用的，简单的表可以搞定，一旦涉及复杂的字段，比如 datetime 等，就会踩坑。

放弃!

4.4 03,makemigrations 方案

django makemigrations 生成 mysql 表，数据:sqlite=>csv=>mysql

前半部分可以:makemigrations 在 mysql 创建新表

后半部分不行:csv 数据 sqlite 导出导入 mysql，失败。django 支持利用 json 导出导入数据，也失败。

Django2 SQLite3 迁移到 MySQL 数据库:https://www.jianshu.com/p/6b28f7dbc1fa

django sqlite 数据库导出迁移到 mysql 数据库方法:https://blog.csdn.net/YPL_ZML/article/details/91892306

4.5 04,django 主从数据库机制

主要参考:

django sqlite3 迁移到 mysql:www.voidcn.com/article/p-hesvaooz-ru.html

数据表创建成功，但迁移数据时失败。

思路上查询出对象在保存到从数据库是没问题的，实际上由于数据存在主键依赖，导致创建表或记录的前后次序有特定要求，自动查询的化未必符合这种要求。

另一个问题就是这种方案会对 django 的默认数据表也进行迁移，也会产生主键依赖问题。

4.6 总结

没找到合适方案可以简单快速的迁移，最后采用 03 方法，django 在 mysql 生成新表。由于目前测试数据并不多，所以数据由各个模块开发人员自行初始化。

4.7 参考

将 sqlite3 中数据导入到 mysql 中的实战教程:https://www.jb51.net/article/118379.htm

sqlite 导入到 mysql:https://www.cnblogs.com/goldenstones/articles/9185950.html

CHAPTER 5

数据库 05redis 常用命令整理

5.1 redis 启动

```
redis-server            # 启动服务端
redis-cli redis.conf    # 启动客户端
redis-cli  -h  {host}  -p  {port}  -a  {password}     # 配置客户端启动的主机和端口
```

5.2 获取 redis 配置信息

| 序号 | 配置项 | 说明 | | —- | ———————————— | ———- | | 1 | **daemonize** no | 配置 Redis 是否以守护线程的方式启动, 默认为 NO | | 2 | **port** 6379 | Redis 监听端口，默认为 6379 | | 3 | **bind** 127.0.0.1 | 绑定的主机地址 | | 4 | **timeout** 300 | 客户端闲置多少秒后关闭连接，如果为 0，表示关闭该功能 | | 5 | **loglevel** notice | 日志记录级别，总共四个级别：debug、verbose、notice、warning，默认为 notice | | 6 | **databases** 16 | 设置数据库的数量, 默认 16 个库 (0-15)，默认初始使用的库为 0，可以使用 SELECT 命令在连接上指定数据库 id | | 7 | **save** | 在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合。Redis 配置文件默认 3 个条件:save 900 1 900 秒内有 1 个更改 save 300 10 300 秒内有 10 个更改 save 60 10000 60 秒内有 10000 个更改 | | 8 | **rdbcompression** yes | 存储至本地数据库时是否压缩数据，默认为 yes，Redis 采用 LZF 压缩，如果为了节省 CPU 时间，可以关闭该选项，但会导致数据库文件变的巨大 | | 9 | **dbfilename** dump.rdb | 本地数据库文件名，默认值为 dump.rdb | | 10 | **appendonly** no | 是否在每次更新操作后进行日志记录，默认为 no | | 11 | **appendfilename** appendonly.aof | 指定更新日志文件名，默认为 appendonly.aof | | 12 | **appendfsync** everysec | 异步更新日志条件：no：表示等操作系统

进行数据缓存同步到磁盘（快）always：表示每次更新操作后手动调用 fsync() 将数据写到磁盘（慢，安全）everysec：表示每秒同步一次（折中，默认值）| | 13 | **dir** ./ | 指定本地数据库存放目录 | | 14 | **slaveof** | 当本机为 slave 时，设置 master 的 IP 地址及端口 |

5.3 Redis 连接命令

下表列出了 redis 连接的基本命令：

```
序号        命令及描述
1         AUTH password 验证密码是否正确
2         ECHO message 打印字符串
3         PING 查看服务是否运行
4         QUIT 关闭当前连接
5         SELECT index 切换到指定的数据库
```

5.4 数据类型

```
String：可以包含任何数据，比如 jpg 图片或者序列化的对象，一个键最大能存储 512M
Hash：适合存储对象，并且可以像数据库中 update 一个属性一样只修改某一项属性值（Memcached 中需要取出整个字符串反序列化成对象修改完再序列化存回去）
List：增删快，提供了操作某一段元素的 API
Set:1，添加、删除，查找的复杂度都是 O(1).2、为集合提供了求交集、并集、差集等操作
Sorted Set：数据插入集合时，已经进行天然排序
```

5.5 Redis keys 命令

```
keys 命令
?    匹配一个字符
*    匹配任意个（包括 0 个）字符
[]    匹配括号间的任一个字符，可以使用 "-" 符号表示一个范围，如 a[b-d] 可以匹配 "ab","ac",
↪"ad"
\x    匹配字符 x，用于转义符号，如果要匹配 "?" 就需要使用 \?

keys * 查询当前库的所有键
exists 判断某个键是否存在
type 查看键的类型
```

（下页继续）

(续上页)

```
del 删除某个键
expire 为键值设置过期时间，单位秒。
```

ttl key

作用：以秒为单位返回 key 的剩余生存时间

返回值：-1：key 永不过期

-2: key 不存在

正整数：key 的剩余存在时间（秒）

5.6 字符串命令

一个 key 对应一个 value。一个键最大能存储 512MB。string 类型是二进制安全的。

```
incr key    # 递增数字
当存储的字符串是整数形式时，redis 提供了一个使用的命令 incr 作用是让当前的键值递增，并返回递增后的值
当要操作的键不存在时会默认键值为 0 ，所以第一次递增后的结果是 1 ，当键值不是整数时 redis 会提示错误

incrby key increment    # 增加指定的整数

decr key                #desc 命令与 incr 命令用法相同，只不过是让键值递减
decrby key increment    #decrby 命令与 incrby 命令用法相同

incrbyfloat key increment # 命令类似 incrby 命令，差别是前者可以递增一个双精度浮点数
```

(1) set key value [ex 秒数] [px 毫秒数] [nx/xx]

如果 ex 和 px 同时写，则以后面的有效期为准

nx：如果 key 不存在则建立

xx：如果 key 存在则修改其值

(2) get key：取值

(3) mset key1 value1 key2 value2 一次设置多个值

(4) mget key1 key2 ： 一次获取多个值

(5) setrange key offset value：把字符串的 offset 偏移字节改成 value

如果偏移量 > 字符串长度，该字符自动补 0x00

（6）append key value ： 把 value 追加到 key 的原值上

（7）getrange key start stop：获取字符串中 [start, stop] 范围的值

对于字符串的下标，左数从 0 开始，右数从-1 开始

注意：

当 start>length，则返回空字符串

当 stop>=length，则截取至字符串尾

如果 start 所处位置在 stop 右边，则返回空字符串

（8）getset key nrevalue：获取并返回旧值，在设置新值

5.7 hash（哈希）

Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。每个 hash 可以存储 232 - 1 键值对（40 多亿）。

（1）hset myhash field value：设置 myhash 的 field 为 value

（2）hsetnx myhash field value：不存在的情况下设置 myhash 的 field 为 value

（3）hmset myhash field1 value1 field2 value2：同时设置多个 field

（4）hget myhash field：获取指定的 hash field

（5）hmget myhash field1 field2：一次获取多个 field

（6）hincrby myhash field 5：指定的 hash field 加上给定的值

（7）hexists myhash field：测试指定的 field 是否存在

（8）hlen myhash：返回 hash 的 field 数量

（9）hdel myhash field：删除指定的 field

（10）hkeys myhash：返回 hash 所有的 field

（11）hvals myhash：返回 hash 所有的 value

（12）hgetall myhash：获取某个 hash 中全部的 field 及 value

5.8 list（列表）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储 40 多亿)。

（1）lpush key value：把值插入到链表头部

（2）rpush key value：把值插入到链表尾部

（3）lpop key ： 返回并删除链表头部元素

（4）rpop key：返回并删除链表尾部元素

（5）lrange key start stop：返回链表中 [start, stop] 中的元素

（6）lrem key count value：从链表中删除 value 值，删除 count 的绝对值个 value 后结束

count > 0 从表头删除

count < 0 从表尾删除

count=0 全部删除

（7）ltrim key start stop：剪切 key 对应的链接，切 [start, stop] 一段并把改制重新赋给 key

（8）lindex key index：返回 index 索引上的值

5.9 set（集合）

Redis 的 Set 是 string 类型的无序集合。值不重复。

（1）sadd key value1 value2：往集合里面添加元素

（2）smembers key：获取集合所有的元素

（3）srem key value：删除集合某个元素

（4）spop key：返回并删除集合中 1 个随机元素（可以坐抽奖，不会重复抽到某人）

（5）srandmember key：随机取一个元素

（6）sismember key value：判断集合是否有某个值

（7）scard key：返回集合元素的个数

（8）smove source dest value：把 source 的 value 移动到 dest 集合中

（9）sinter key1 key2 key3：求 key1 key2 key3 的交集

（10）sunion key1 key2：求 key1 key2 的并集

（11）sdiff key1 key2：求 key1 key2 的差集

（12）sinterstore res key1 key2：求 key1 key2 的交集并存在 res 里

5.10 zset（sorted set：有序集合）

Redis zset 和 set 一样也是 string 类型元素的集合。且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。zset 的成员是唯一的，但分数 (score) 却可以重复。

（1）zadd key score1 value1：添加元素

（2）zrange key start stop [withscore]：把集合排序后，返回名次 [start,stop] 的元素

默认是升续排列 withscores 是把 score 也打印出来

（3）zrank key member：查询 member 的排名（升序 0 名开始）

（4）zrangebyscore key min max [withscores] limit offset N：集合（升序）

排序后取 score 在 [min, max] 内的元素，并跳过 offset 个，取出 N 个

（5）zrevrank key member：查询 member 排名（降序 0 名开始）

（6）zremrangebyscore key min max：按照 score 来删除元素，删除 score 在 [min, max] 之间

（7）zrem key value1 value2：删除集合中的元素

（8）zremrangebyrank key start end：按排名删除元素，删除名次在 [start, end] 之间的

（9）zcard key：返回集合元素的个数

（10）zcount key min max：返回 [min, max] 区间内元素数量

5.11 服务器相关命令

（1）ping：测定连接是否存活

（2）echo：在命令行打印一些内容

（3）select：选择数据库

（4）quit：退出连接

（5）dbsize：返回当前数据库中 key 的数目

（6）info：获取服务器的信息和统计

（7）monitor：实时转储收到的请求

（8）config get 配置项：获取服务器配置的信息

config set 配置项值：设置配置项信息

（9）flushdb：删除当前选择数据库中所有的 key

（10）flushall：删除所有数据库中的所有的 key

（11）time：显示服务器时间，时间戳（秒），微秒数

（12）bgrewriteaof：后台保存 rdb 快照

（13）bgsave：后台保存 rdb 快照

（14）save：保存 rdb 快照

（15）lastsave：上次保存时间

（16）shutdown [save/nosave]

注意：如果不小心运行了 flushall，立即 shutdown nosave，关闭服务器，然后手工编辑 aof 文件，去掉文件中的 flushall 相关行，然后开启服务器，就可以倒回原来是数据。如果 flushall 之后，系统恰好 bgwriteaof 了，那么 aof 就清空了，数据丢失。

(17) showlog：显示慢查询

问：多慢才叫慢?

答：由 slowlog-log-slower-than 10000，来指定（单位为微秒）

问：服务器存储多少条慢查询记录

答：由 slowlog-max-len 128，来做限制

5.12 主从复制

master 可以有多个 slave，slave 还可以连接到其他 slave。

主从复制不会阻塞 master，在数据同步时，master 可以继续处理 client 请求

5.13 事务控制

注意：redis 部分事务失败，不会回滚全部事务

5.14 持久化机制

(1) 自动保存快照

配置信息：

```
save 900 1 # 如果 900 秒内超过 1 个 key 被修改，则发起快照保存
save 300 10 # 如果 300 秒内超过 1 个 key 被修改，则发起快照保存
save 60 10000 …
```

1. redis 调用 fork，为主进程（父进程）创建一个子进程
2. 父进程继续处理 client 请求，子进程将内存内容写入临时文件。子进程地址空间内的数据是 fork 时的整个数据库的快照。子进程不会影响父进程处理 client 请求。

3. 当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

(2)save / bgsave

手动保存快照，在主线程中完成快照保存，会阻塞所有的 client 请求。

每次都是将内存数据完整的写入到磁盘，如果数据量大，会引起大量的 IO 操作，影响性能。

Append-only file 方式

记录每次 write 的操作内容

比快照方式更好

5.15 发布及订阅消息 pub/sub

5.16 使用场景

1. 字符串

缓存功能、计数、共享 session、限速

2. 哈希

hash 特别适合用于存储对象。

存储部分变更的数据，如用户信息等。

3. 列表

lpush+lpop=stock(栈)

lpush+rpop=queue(队列)

lpush+ltrim=Capped Collection (有限集合)

lpush+brpop=Message Queue(消息队列)

4. 集合 sadd=Tagging(标签)

spop/srandmember=Random item (生成随机数，比如抽奖)

sadd+sinter=Social Graph(社交需求)

5. 有序集合

排行榜

5.17 参考

Redis 常用命令整理：https://blog.csdn.net/weixin_42389216/article/details/106792627

Redis 常用命令整理：https://blog.csdn.net/weixin_42714605/article/details/91413847

redis 常用命令整理：https://www.jb51.net/article/182067.htm

redis 常用命令总结：https://blog.csdn.net/qq_38174263/article/details/80009943

Redis 常用命令整理：https://www.cnblogs.com/seizedays/p/12493036.html

redis 常用命令及使用场景：https://www.cnblogs.com/alphathink/p/10749626.html

Redis 基本命令整理:https://blog.csdn.net/weixin_33834628/article/details/89442998

CHAPTER 6

数据库 06redis 事务

6.1 Redis 事务命令

下表列出了 redis 事务的相关命令：

序号命令及描述

1 DISCARD 取消事务，放弃执行事务块内的所有命令。

2 EXEC 执行所有事务块内的命令。

3 MULTI 标记一个事务块的开始。

4 UNWATCH 取消 WATCH 命令对所有 key 的监视。

5 WATCH key [key ...] 监视一个 (或多个) key ，如果在事务执行之前这个 (或这些) key 被其他命令所改动，那么事务将被打断。

事务可以一次执行多个命令，并且带有以下两个重要的保证：

事务是一个**单独的隔离操作**：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个**原子操作**：事务中的命令要么全部被执行，要么全部都不执行。

EXEC 命令负责触发并执行事务中的所有命令：

当使用 AOF 方式做持久化的时候，Redis 会使用单个 write(2) 命令将事务写入到磁盘中。

然而，如果 Redis 服务器因为某些原因被管理员杀死，或者遇上某种硬件故障，那么可能只有部分事务命令会被成功写入到磁盘中。

如果 Redis 在重新启动时发现 AOF 文件出了这样的问题，那么它会退出，并汇报一个错误。

使用 redis-check-aof 程序可以修复这一问题：它会移除 AOF 文件中不完整事务的信息，确保服务器可以顺利启动。

6.2 使用 check-and-set 操作实现乐观锁

WATCH 命令可以为 Redis 事务提供 check-and-set （CAS）行为。

被 WATCH 的键会被监视，并会发觉这些键是否被改动过了。如果有至少一个被监视的键在 EXEC 执行之前被修改了，那么整个事务都会被取消，EXEC 返回空多条批量回复（null multi-bulk reply）来表示事务已经失败。

举个例子，假设我们需要原子性地为某个值进行增 1 操作（假设 INCR 不存在）。

```
WATCH mykey



val = GET mykey

val = val + 1



MULTI

SET mykey $val

EXEC
```

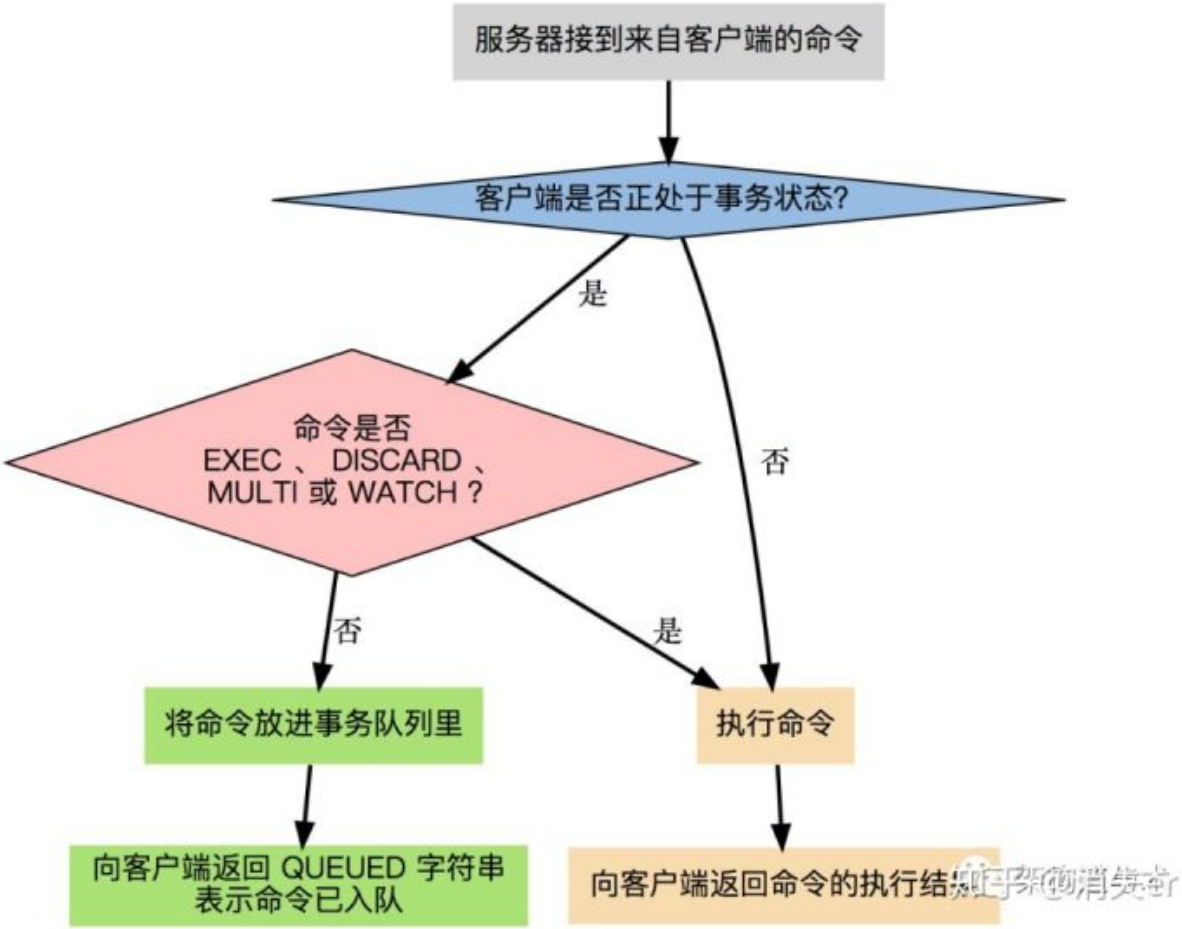
6.3 Redis 事务能回滚吗

如果一个事务中的命令出现了语法错误，大部分客户端驱动会返回错误，2.6.5 版本以上的 Redis 也会在执行 EXEC 时检查队列中的命令是否存在语法错误，如果存在，则会自动放弃事务 (提前标记 DISCARD) 并返回错误。

但如果一个事务中的命令有非语法类的错误 (比如对 String 执行 HSET 操作)，无论客户端驱动还是 Redis 都无法在真正执行这条命令之前发现，所以事务中的所有命令仍然会被依次执行。

在这种情况下，会出现一个事务中部分命令成功部分命令失败的情况，然而与 RDBMS 不同，Redis 不提供事务回滚的功能，所以只能通过其他方法进行数据的回滚。

6.4 在事务和非事务状态下执行命令



6.5 小结一：事务 3 阶段

开启：以 MULTI 开启一个事务

入队：将多个命令入队到事务中，接到这些命令不会立即执行，而是放到等待执行的事务队列里面

执行：由 EXEC 命令触发事务

6.6 小结二：事务 3 特性

单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

没有隔离级别的概念：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行，也就不存在”事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的

问题

不保证原子性：redis 同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

6.7 参考

Redis 的事务讲解：https://www.cnblogs.com/dwlovelife/p/10946868.html

你真的懂 Redis 事务吗?：https://cloud.tencent.com/developer/article/1189074

Redis 事务, 你真的了解吗：https://zhuanlan.zhihu.com/p/101902825?utm_source=wechat_session

https://www.redis.net.cn/tutorial/3502.html

CHAPTER 7

数据库 07redis 分布式锁

redis 本身不处理分布 bai 式事物或者说它的 du 事物非常弱，因为 zhiredis 本身是单线程的；

之所以很多时候出现 redis 的线程安全问题是因为应用本身是分布式的；这块处理办法基本都是 redis+lua 解决分布式安全问题

7.1 分布式锁

先说说“线程锁”，线程锁大家都很熟悉，其使用环境大致可以描述为：如果有多个线程要同时访问某个共享资源的时候，我们可以采用线程间加锁的机制，即当某个线程获取到这个资源后，就立即对这个资源进行加锁，当使用完资源之后，再解锁，其它线程就可以接着使用了。

而分布式事务锁和线程锁锁大致意思相同，只不过其作用范围是“进程”之间。系统可能会有多份服务进程并且部署在不同的机器上，许多资源已经不是在线程之间共享了，而是属于进程之间共享的资源。所以，分布式事务锁可以描述为：是指在分布式的部署环境下，通过锁机制来让多客户端互斥的对共享资源进行访问。

实现分布式事务锁的方式有很多，但是市面上常用的技术就是使用“数据库/redis”或 zookeeper 来实现。而核心的处理思路就是“获取锁”（写一条数据）、“删除锁”（删除这条数据）。其中使用“数据库“与”redis”的实现方式基本一样，咱们就只说如何使用 redis 实现分布式事务锁。

Redis 为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对 Redis 的连接并不存在竞争关系。

安全性：保证互斥，在任何时候，只有一个客户端可以持有锁

无死锁：即使当前持有锁的客户端崩溃或者从集群中被分开了，其它客户端最终总是能够获得锁。

容错性：只要大部分的 Redis 节点在线，那么客户端就能够获取和释放锁。

7.2 分布式锁的实现有哪些

1.Memcached 分布式锁

利用 Memcached 的 add 命令。此命令是原子性操作，只有在 key 不存在的情况下，才能 add 成功，也就意味着线程得到了锁。

2.Redis 分布式锁

和 Memcached 的方式类似，利用 Redis 的 setnx 命令。此命令同样是原子性操作，只有在 key 不存在的情况下，才能 set 成功。（setnx 命令并不完善，后续会介绍替代方案）

3.Zookeeper 分布式锁

利用 Zookeeper 的顺序临时节点，来实现分布式锁和等待队列。Zookeeper 设计的初衷，就是为了实现分布式锁服务的。

4.Chubby

Google 公司实现的粗粒度分布式锁服务，底层利用了 Paxos 一致性算法。

7.3 Redis 实现分布式锁

分布式锁实现的三个核心要素：1. 加锁

最简单的方法是使用 setnx 命令。key 是锁的唯一标识，按业务来决定命名。比如想要给一种商品的秒杀活动加锁，可以给 key 命名为 “lock_sale_ 商品 ID”。而 value 设置成什么呢？我们可以姑且设置成 1。加锁的伪代码如下：

setnx（key，1）

当一个线程执行 setnx 返回 1，说明 key 原本不存在，该线程成功得到了锁；当一个线程执行 setnx 返回 0，说明 key 已经存在，该线程抢锁失败。

2. 解锁

有加锁就得有解锁。当得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行 del 指令，伪代码如下：

del（key）

释放锁之后，其他线程就可以继续执行 setnx 命令来获得锁。

3. 锁超时

锁超时是什么意思呢？如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程再也别想进来。所以，setnx 的 key 必须设置一个超时时间，以保证即使没有被显式释放，这把锁也要在一定时间后自动释放。setnx 不支持超时参数，所以需要额外的指令，伪代码如下：

expire（key，30）

综合起来，我们分布式锁实现的第一版伪代码如下：

```
if (setnx (key, 1)  == 1) {
    expire (key, 30)
    try {
        do something ......
    } finally {
        del (key)
    }
}
```

存在着三个致命问题：

1. setnx 和 expire 的非原子性

设想一个极端场景，当某线程执行 setnx，成功得到了锁：

setnx 刚执行成功，还未来得及执行 expire 指令，节点 1 Duang 的一声挂掉了。

这样一来，这把锁就没有设置过期时间，变得“长生不老”，别的线程再也无法获得锁了。

怎么解决呢？setnx 指令本身是不支持传入超时时间的，幸好 Redis 2.6.12 以上版本为 set 指令增加了可选参数，伪代码如下：

set（key，1，30，NX）

1. del 导致误删

又是一个极端场景，假如某线程成功得到了锁，并且设置的超时时间是 30 秒。

如果某些原因导致线程 A 执行的很慢很慢，过了 30 秒都没执行完，这时候锁过期自动释放，线程 B 得到了锁。

随后，线程 A 执行完了任务，线程 A 接着执行 del 指令来释放锁。但这时候线程 B 还没执行完，**线程 A 实际上删除的是线程 B 加的锁**（A 自己的锁由于超时被自动删了）。

怎么避免这种情况呢？可以在 del 释放锁之前做一个判断，验证当前的锁是不是自己加的锁。

至于具体的实现，可以在加锁的时候把当前的线程 ID 当做 value，并在删除之前验证 key 对应的 value 是不是自己线程的 ID。

这样就可以取代 setnx 指令。

```
加锁：
String threadId = Thread.currentThread().getId()
set (key, threadId , 30, NX)
解锁：
if (threadId .equals(redisClient.get(key))) {
```

（下页继续）

(续上页)

```
    del(key)
}
```

但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性。

我们都是追求极致的程序员，所以这一块要用 Lua 脚本来实现：

```
String luaScript = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del
↪', KEYS[1]) else return 0 end";
redisClient.eval(luaScript , Collections.singletonList(key), Collections.
↪singletonList(threadId));
```

这样一来，验证和删除过程就是原子操作了。

1. 出现并发的可能性

还是刚才第二点所描述的场景，虽然我们避免了线程 A 误删掉 key 的情况，但是同一时间有 A，B 两个线程在访问代码块，仍然是不完美的。

怎么办呢？我们可以让获得锁的线程开启一个守护线程，用来给快要过期的锁“续航”。

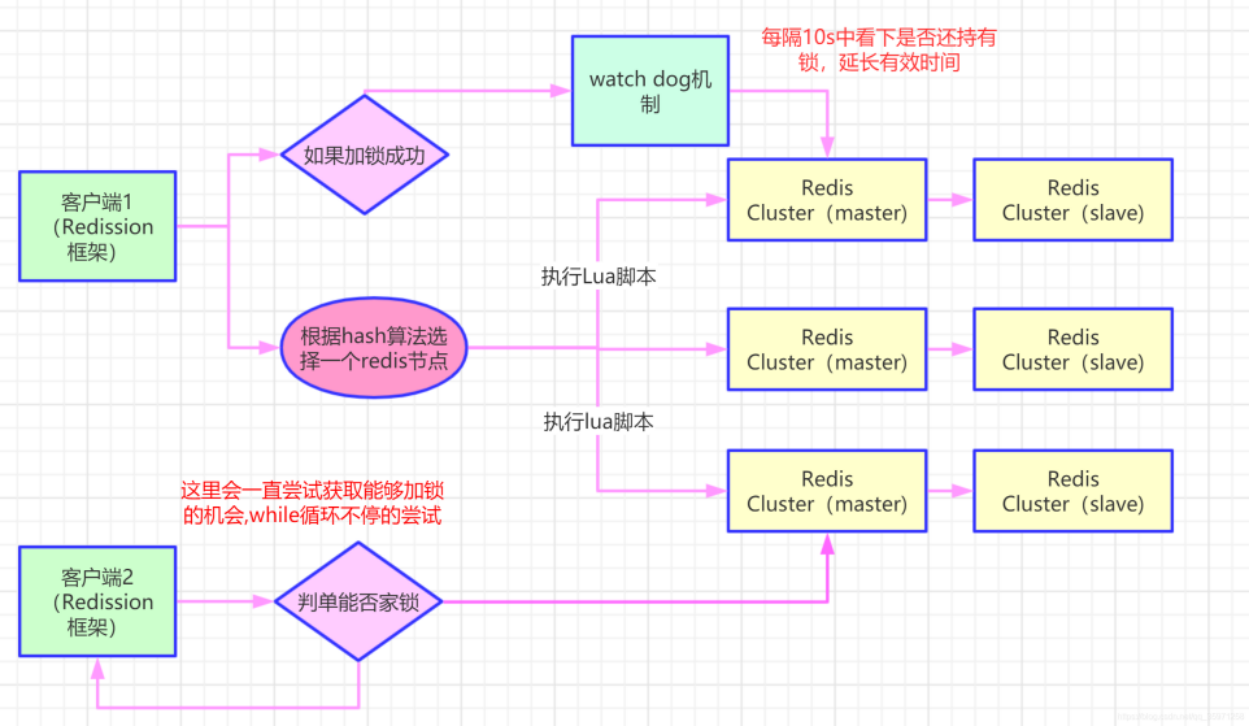
当过去了 29 秒，线程 A 还没执行完，这时候守护线程会执行 expire 指令，为这把锁“续命”20 秒。守护线程从第 29 秒开始执行，每 20 秒执行一次。

当线程 A 执行完任务，会显式关掉守护线程。

另一种情况，如果节点 1 忽然断电，由于线程 A 和守护线程在同一个进程，守护线程也会停下。这把锁到了超时的时候，没人给它续命，也就自动释放了。

7.4 Redisson 实现 Redis 分布式锁的底层原理

Redisson 这个开源框架对 Redis 分布式锁的实现原理。



（1）加锁机制

咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个 redis cluster 集群，他首先会根据 hash 节点选择一台机器。

这里注意，仅仅只是选择一台机器！这点很关键！

紧接着，就会发送一段 lua 脚本到 redis 上，那段 lua 脚本如下所示：

```
"if (redis.call('exists', KEYS[1]) == 0) then " +
    "redis.call('hset', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"return redis.call('pttl', KEYS[1]);"
```


（2）锁互斥机制

那么在这个时候，如果客户端 2 来尝试加锁，执行了同样的一段 lua 脚本，会咋样呢?

很简单，第一个 if 判断会执行 “exists myLock”，发现 myLock 这个锁 key 已经存在了。

接着第二个 if 判断，判断一下，myLock 锁 key 的 hash 数据结构中，是否包含客户端 2 的 ID，但是明显不是的，因为那里包含的是客户端 1 的 ID。

所以，客户端 2 会获取到 pttl myLock 返回的一个数字，这个数字代表了 myLock 这个锁 key 的剩余生存时间。比如还剩 15000 毫秒的生存时间。

此时客户端 2 会进入一个 while 循环，不停的尝试加锁。

(3) watch dog 自动延期机制

客户端 1 加锁的锁 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢?

简单！只要客户端 1 一旦加锁成功，就会启动一个 watch dog 看门狗，他是一个后台线程，会每隔 10 秒检查一下，如果客户端 1 还持有锁 key，那么就会不断的延长锁 key 的生存时间。

(4) 可重入加锁机制

那如果客户端 1 都已经持有了这把锁了，结果可重入的加锁会怎么样呢?

比如下面这种代码:

```
RLock lock = redisson.getLock("myLock");
lock.lock();

// 一大坨代码

lock.lock();
// 一大坨代码
lock.unlock();

lock.unlock();
```


那个 myLock 的 hash 数据结构中的那个客户端 ID，就对应着加锁的次数

(5) 释放锁机制

如果执行 lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对 myLock 数据结构中的那个加锁次数减 1。

如果发现加锁次数是 0 了，说明这个客户端已经不再持有锁了，此时就会用:

“del myLock” 命令，从 redis 里删除这个 key。

然后呢，另外的客户端 2 就可以尝试完成加锁了。

这就是所谓的分布式锁的开源 Redisson 框架的实现机制。

一般我们在生产系统中，可以用 Redisson 框架提供的这个类库来基于 redis 进行分布式锁的加锁与释放锁。

（6）上述 Redis 分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个 redis master 实例，写入了 myLock 这种锁 key 的 value，此时会异步复制给对应的 master slave 实例。但是这个过程中一旦发生 redis master 宕机，主备切换，redis slave 变为了 redis master。

接着就会导致，客户端 2 来尝试加锁的时候，在新的 redis master 上完成了加锁，而客户端 1 也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是 redis cluster，或者是 redis master-slave 架构的主从异步复制导致的 redis 分布式锁的最大缺陷：在 redis master 实例宕机的时候，可能导致多个客户端同时完成加锁。

原文链接：https://mp.weixin.qq.com/s?__biz=MzU0OTk3ODQ3Ng==&mid=2247483893&idx=1&sn=32e7051116ab60e41

7.5 错误加锁方式

错误方式一

保证互斥和防止死锁，首先想到的使用 redis 的 setnx 命令保证互斥，为了防止死锁，锁需要设置一个超时时间。

```
    public static void wrongLock(Jedis jedis, String key, String uniqueId, int␣
↪expireTime) {
        Long result = jedis.setnx(key, uniqueId);
        if (1 == result) {
            //如果该 redis 实例崩溃，那就无法设置过期时间了
            jedis.expire(key, expireTime);
        }
    }
```

在多线程并发环境下，任何非原子性的操作，都可能导致问题。这段代码中，如果设置过期时间时，redis 实例崩溃，就无法设置过期时间。如果客户端没有正确的释放锁，那么该锁 (永远不会过期)，就永远不会被释放。

错误方式二

比较容易想到的就是设置值和超时时间为原子原子操作就可以解决问题。那使用 setnx 命令，将 value 设置为过期时间不就 ok 了吗?

```
public static boolean wrongLock(Jedis jedis, String key, int expireTime) {
    long expireTs = System.currentTimeMillis() + expireTime;
    // 锁不存在，当前线程加锁成果
    if (jedis.setnx(key, String.valueOf(expireTs)) == 1) {
        return true;
    }

    String value = jedis.get(key);
    //如果当前锁存在，且锁已过期
    if (value != null && NumberUtils.toLong(value) < System.currentTimeMillis()) {
        //锁过期，设置新的过期时间
        String oldValue = jedis.getSet(key, String.valueOf(expireTs));
        if (oldValue != null && oldValue.equals(value)) {
            // 多线程并发下，只有一个线程会设置成功
            // 设置成功的这个线程，key 的旧值一定和设置之前的 key 的值一致
            return true;
        }
    }
    // 其他情况，加锁失败
    return true;
}
```

乍看之下，没有什么问题。但仔细分析，有如下问题：

value 设置为过期时间，就要求各个客户端严格的时钟同步，这就需要使用到同步时钟。即使有同步时钟，分布式的服务器一般来说时间肯定是存在少许误差的。

锁过期时，使用 jedis.getSet 虽然可以保证只有一个线程设置成功，但是不能保证加锁和解锁为同一个客户端，因为没有标志锁是哪个客户端设置的嘛。

7.6 错误解锁方式

解锁错误方式一

直接删除 key

```
public static void wrongReleaseLock(Jedis jedis, String key) {
    //不是自己加锁的 key，也会被释放
    jedis.del(key);
}
```

简单粗暴，直接解锁，但是不是自己加锁的，也会被删除，这好像有点太随意了吧！

解锁错误方式二

判断自己是不是锁的持有者，如果是，则只有持有者才可以释放锁。

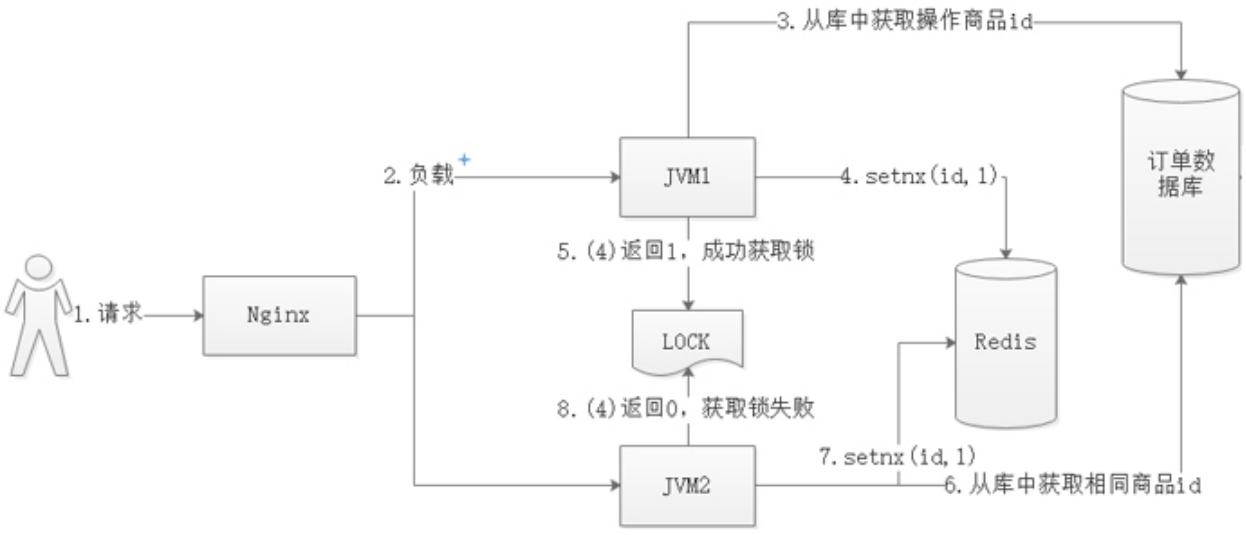
```
public static void wrongReleaseLock(Jedis jedis, String key, String uniqueId) {
    if (uniqueId.equals(jedis.get(key))) {
        // 如果这时锁过期自动释放，又被其他线程加锁，该线程就会释放不属于自己的锁
        jedis.del(key);
    }
}
```

看起来很完美啊，但是如果你判断的时候锁是自己持有的，这时锁超时自动释放了。然后又被其他客户端重新上锁，然后当前线程执行到 jedis.del(key)，这样这个线程不就删除了其他线程上的锁嘛，好像有点乱套了哦！

7.7 商品交易案例

7.7.1 分布式事务锁业务流程描述

一个系统中，有多个服务（jvm1、jvm2）同时对同一 id 的商品处理，其分布式事务锁流程如下所示：



7.7.2 实现方式

1、加锁

使用 setnx 命令，伪代码：setnx(id,value)。成功返回 1，说明 key 不存在，线程抢锁成功。失败返回 0，说明 key 已存在，线程抢锁失败。

注意：setnx(id,value) 中 key 为操作商品的 id，value 用于存储进程编号与线程编号，用于解锁的时候防止误删。

2、解锁

使用 del 命令，伪代码：del(id)

7.7.3 问题分析

1、锁超时不删问题

描述：如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，该资源将会被永远占用，其他线程将无法访问。

解决：为此，我们可以使用 expire 为 key 设置一个超时时间，与 setnx 命令一起执行（setnx 不支持超时参数），用以保证即使未被显式释放，该锁也可在一定时间后自动释放。伪代码：

```
expire (key,value, 30)。
```

2、误删问题

描述：根据第一个“超时问题”，我们引申一下，假如，时间到了，任务没有执行完，另一个新的进程获取了锁，咋办？结合上图，咱们描述一个场景：

```
a、JVM1 使用 set(001, 002, 30) 成功获取锁，并设置超时时间为 30s;
b、JVM1 开始数据处理，处理时间已经超过了 30s...
c、服务器检测到 (001, 002, 30) 数据超时，将自动执行 del 进行数据删除，此时 JVM1 还在数据处理.
↪..
d、此时，JVM2 使用 set(001, 002, 30) 成功获取锁，并设置超时时间为 30s;
e、JVM2 开始数据处理。与此同时，JVM1 处理完成，操作提交后，根据商品 id001，执行了 del;
f、到此，JVM1 成功误删了 JVM2 的锁。
```

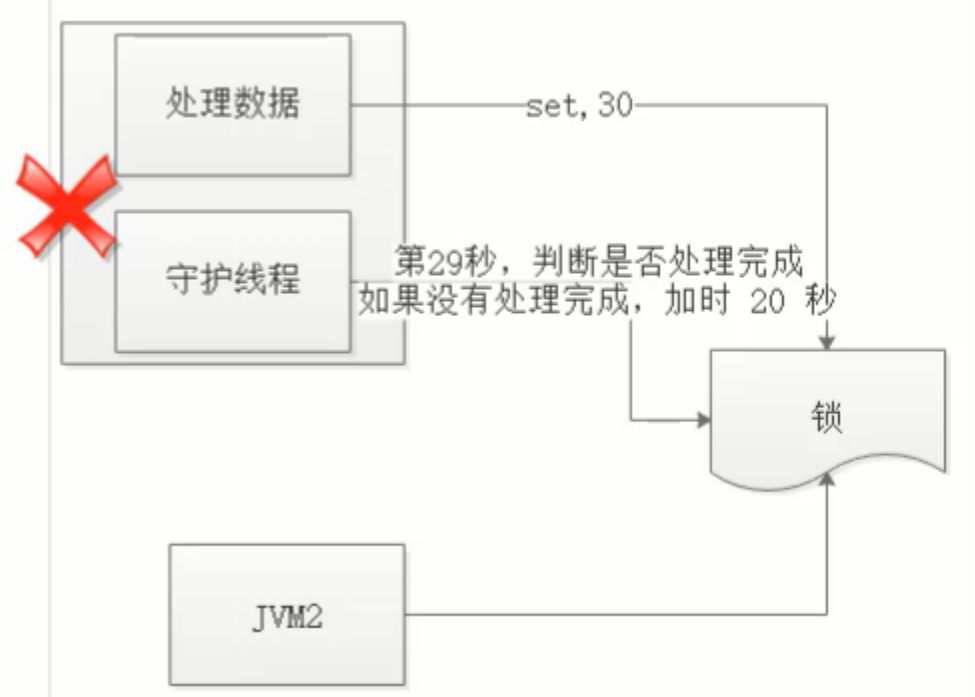
误删了锁，咋办？jvm1 线程明明还没有完成，结果，时间到期了，咋办？

解决：误删问题可以这样，加锁的时候，set(id, threadId,expire)，把 value 设为当前线程 ID 和进程的编号。执行 del 命令时，根据进程 id 和 threadId 数据判断该锁是否仍属于本线程。是，则删除。

遗留：jvm1 线程明明还没有完成，结果，时间到期了，咋办？

3、基于误删锁的前提下，由于我们无法确定程序成功处理完成数据的具体时间，这就为超时时间的设置提出了难题。设置时间过长、过短都将影响程序并发的效率。

解决：守护线程，即为获取锁的锁的线程开启一个守护线程。当 29 秒时（或更早），线程 A 还没执行完，守护线程会执行 expire 指令，为这把锁“续命”20 秒。守护线程从第 29 秒开始执行，每 20 秒执行一次。当线程 A 执行完任务，会显式关掉守护线程。如图：



4、守护线程挂了

这个问题是有上个问题引发的，如果守护现在挂了？上面的那些问题如何解决？虽然想法有点“极致”，但是，确实有可能发生，暂时还没有好的解决方案。

7.7.4 组件插件

在 nuget 上，有现成的 redis 分布式事务锁的包，咱们不用重复造轮子，可以直接下载下来用就行了：各种版本的基本上都有：各类语言实现的分布式事务锁 distlock。DontNET 基于 Redis 的分布式事务锁 RedLock.net 基于 sqlserver 实现的分布式事务锁 DistributedLock

7.7.5 尾语

最后的问题得不到完美的解决，恰巧说明了这中方式实现分布式事务锁有弊端，那么有没有没有一种完美的“分布式事务锁”的实现方案呢？有，参考我后面的文章：zookeeper 实现分布式事务锁。

7.8 参考

Redis 分布式事务锁实现:https://blog.csdn.net/wtopps/article/details/70768062

redis 分布式锁———redis 分布式事务:https://www.cnblogs.com/weigy/p/12560455.html

redis 怎么处理分布式事务的：https://zhidao.baidu.com/question/629220812958418764.html

Redis 分布式锁的实现原理看这篇就够了～:https://blog.csdn.net/weixin_34117211/article/details/92422665

redis 分布式锁，面试官请随便问，我都会：https://blog.csdn.net/chanllenge/article/details/102983597

分布式事务锁的实现-redis：zhifeiya.cn/post/分布式事务锁的实现-redis

漫画：什么是分布式锁?：https://mp.weixin.qq.com/s/8fdBKAyHZrfHmSajXT_dnA

CHAPTER 8

数据库 08redis 其他

8.1 Redis 中为什么引入 Lua 脚本？

Redis 是高性能的 key-value 内存数据库，在部分场景下，是对关系数据库的良好补充。

Redis 提供了非常丰富的指令集，官网上提供了 200 多个命令。但是某些特定领域，需要**扩充若干指令原子性执行时，仅使用原生命令便无法完成**。

Redis 为这样的用户场景提供了 lua 脚本支持，用户可以向服务器发送 lua 脚本来执行自定义动作，获取脚本的响应数据。**Redis 服务器会单线程原子性执行 lua 脚本**，保证 lua 脚本在处理的过程中不会被任意其它请求打断。

Redis 意识到上述问题后，在 2.6 版本推出了 lua 脚本功能，允许开发者使用 Lua 语言编写脚本传到 Redis 中执行。使用脚本的好处如下:

减少网络开销。可以将多个请求通过脚本的形式一次发送，减少网络时延。
原子操作。`Redis` 会将整个脚本作为一个整体执行，中间不会被其他请求插入。因此在脚本运行过程中无需担心会出现竞态条件，无需使用事务。
复用。客户端发送的脚本会永久存在 `redis` 中，这样其他客户端可以复用这一脚本，而不需要使用代码完成相同的逻辑。

8.2 什么是 Lua？

Lua 是一种轻量小巧的脚本语言，用标准 C 语言编写并以源代码形式开放。

其设计目的就是为了嵌入应用程序中，从而**为应用程序提供灵活的扩展和定制功能**。因为广泛的应用于：游戏开发、独立应用脚本、Web 应用脚本、扩展和数据库插件等。

比如：Lua 脚本用在很多游戏上，主要是 Lua 脚本可以嵌入到其他程序中运行，游戏升级的时候，可以直接升级脚本，而不用重新安装游戏。

8.3 redis 集群的三种模式（主从模式，哨兵模式,Cluster 集群）

8.4 参考

Redis 中使用 Lua 脚本（一）：https://zhuanlan.zhihu.com/p/77484377

redis 集群的三种模式：https://blog.csdn.net/zhangge3663/article/details/106617273

Redis 集群详解：https://blog.csdn.net/miss1181248983/article/details/90056960#t2

CHAPTER 9

数据库 09mysql 常用查询实例

9.1 查询统计结果中的前 n 条记录

```
SELECT * ,(yw+sx+wy) AS total FROM tb_score ORDER BY (yw+sx+wy) DESC LIMIT 0,$num
```

9.2 按月查询统计数据，区间查询 between and

```
SELECT * FROM tb_stu WHERE month(date) = between 1 and 3 ORDER BY date ;
```

注：SQL 语言中提供了如下函数，利用这些函数可以很方便地实现按年、月、日进行查询

year(data): 返回 data 表达式中的公元年分所对应的数值

month(data): 返回 data 表达式中的月分所对应的数值

day(data): 返回 data 表达式中的日期所对应的数值

9.3 NOT 与谓词进行组合条件的查询

```
(1)NOT BERWEEN … AND … 对介于起始值和终止值间的数据时行查询 可改成 < 起始值 AND > 终止值
(2)IS NOT NULL 对非空值进行查询
```

(下页继续)

(续上页)

(3)IS NULL 对空值进行查询
(4)NOT IN 该式根据使用的关键字是包含在列表内还是排除在列表外，指定表达式的搜索，搜索表达式可以是常量或列名，而列名可以是一组常量，但更多情况下是子查询

9.4 多列数据分组统计

多列数据分组统计与单列数据分组统计类似

```
SELECT *, SUM(字段 1* 字段 2) AS (新字段 1) FROM 表名 GROUP BY 字段 ORDER BY 新字段 1 DESC
SELECT id,name,SUM(price*num) AS sumprice  FROM tb_price GROUP BY pid ORDER BY sumprice␣
↪DESC
```

注：group by 语句后面一般为不是聚合函数的数列，即不是要分组的列

9.5 查询“c001”课程比“c002”课程成绩高的所有学生的学号；

```
select a.* from
 (select * from sc a where a.cno='c001') a,
 (select * from sc b where b.cno='c002') b
 where a.sno=b.sno and a.score > b.score;
```

9.6 查询平均成绩大于 60 分的同学的学号和平均成绩；

```
select sno,avg(score) from sc  group by sno having avg(score)>60;
```

9.7 查询没学过“谌燕”老师课的同学的学号、姓名；

```
select * from student st where st.sno not in
(
 select distinct sno from sc s
 join course c on s.cno=c.cno
 join teacher t on c.tno=t.tno where tname='谌燕'
)
```

9.8 查询没有学全所有课的同学的学号、姓名；

```
select stu.sno,stu.sname,count(sc.cno) from student stu
 left join sc on stu.sno=sc.sno
 group by stu.sno,stu.sname
 having count(sc.cno)<(select count(distinct cno)from course)
```

9.9 按各科平均成绩从低到高和及格率的百分数从高到低顺序

```
select cno,avg(score),sum(case when score>=60 then 1 else 0 end)/count(*) as 及格率
from sc group by cno
order by avg(score) , 及格率 desc
```

9.10 查询各科成绩前三名的记录:(不考虑成绩并列情况)

```
select * from
(select sno,cno,score,row_number()over(partition by cno order by score desc) rn from sc)
where rn<4
```

mysql 还有其他写法，通过求出极值再进行关联

复制代码

```
SELECT t.stuid,
        t.stuname,
        t.score,
        t.classid
FROM stugrade t
where t.score = (SELECT max(tmp.score) from stugrade tmp where tmp.classid=t.classid)
```

9.11 查询两门以上不及格课程的同学的学号及其平均成绩

```
语句：select stuId,avg(ifnull(stuScore,0)) from score where stuId in (select stuId from␣
↪score where stuScore <60 group by stuId having count(*) >2) group by stuId;
```

9.12 参考

23 个 MySQL 常用查询语句：https://bbs.csdn.net/topics/390407669 Mysql 常用 SQL 语句集锦:https://zhuanlan.zhihu.com/p/24327708

MySQL 学生表、老师表、课程表和成绩表查询语句，全部亲测：https://blog.csdn.net/wq12310613/article/details/100705492

MySQL 全方位练习（学生表教师表课程表分数表）：https://www.cnblogs.com/mzhaox/p/11280234.html

CHAPTER 10

数据库 10mysql 之坑

10.1 CONCAT 字符串拼接

```
select CONCAT('I',' love',' money');  -- I love money
```

10.2 INSERT（字符串，起始位置，长度，替换内容）字符串替换，可用作脱敏

```
select INSERT('123456789',4,4,'****'); # 123****89
select INSERT('123456789',2,2,'**');   # 1**456789
```

10.3 SUBSTRING（字符串，起始位置，长度）字符串截取

```
select SUBSTRING('123456789',1,4); #1234
select SUBSTRING('123456789',2,3); #234
```

10.4 BETWEEN AND

函数包括左右边界，相当于 >= 、<=。

10.5 COUNT 函数

```
SELECT COUNT(1) FROM t WHERE id<0; # 0
# count 函数与 group buy 组合使用，没有记录输出为 null
SELECT COUNT(1) FROM t WHERE id<0 GROUP BY id; # NULL
```

10.6 计算两个日期之间的天数

```
# DAY 两个日期之间时间戳的差值/86400 ，其它的自己实践下哈
# SECOND/MINUTE/HOUR/DAY/MONTH/YEAR 以 DAY 为例
SELECT TIMESTAMPDIFF(DAY ,'2018-08-23','2018-08-31');# 8
SELECT TIMESTAMPDIFF(DAY ,'2018-08-23','2018-08-32');# NULL
```

10.7 WHERE 条件

1、正则匹配

```
# . 匹配字符串中的任意一个字符，包括回车和换行
# * 匹配多个该符号之前的字符，包含 0 和 1 个
# + 匹配多个该字符之前的字符，包括 1 个
SELECT * FROM table WHERE column REGEXP '^a[bcd]e{2,3}f.z$';
```

2.、NOT IN 对 NULL 值的处理

```
SELECT * FROM `learn`; #[id,bro] => [1=>a,2=>b, 3=>null]
SELECT * FROM `learn` WHERE bro NOT IN ('a'); # [2=>b]
# 如果想要 [2=>b,3=>null]
select * FROM learn WHERE bro NOT IN ('a') OR bro is NULL;
## ps: unique 约束对 null 值无效，null 值还会降低索引效率，所以无特殊情况，字段应设置为 not␣
↪null
```

10.8 UNION / UNION ALL 数据合并时与单独子查询的字段名无关，与字段位置有关

```
# 对应下面第一个图
SELECT id as aid,money FROM test WHERE id<3
UNION ALL
SELECT id,money FROM test WHERE id<5;

# 对应下面第二个图
SELECT id as aid,money FROM test WHERE id<3
UNION ALL
SELECT money,id FROM test WHERE id<5;
```

aid	money
1	111
2	222
1	111
2	222
3	255
4	255

aid	money
1	111
2	222
111	1
222	2
255	3
255	4

10.9 行转列

大佬地址：https://www.cnblogs.com/ooo0/p/9085224.html

```
SELECT * FROM        property;
#### 结果如下####
id  name course score
1        张三         数学        3
2        张三         语文        4
3        张三         英语        5
4        李四         数学        6
5        李四         语文        7
6        李四         英语        8
7        王五         数学        9
8        王五         语文        10
9        王五         英语        11
```

行转列 sql

```
SELECT name,course,
MAX(CASE course WHEN ' 数学' THEN score ELSE 0 END) as ' 数学',
MAX(CASE course WHEN ' 语文' THEN score ELSE 0 END) as ' 语文',
MAX(CASE course WHEN ' 英语' THEN score ELSE 0 END) as ' 英语'
FROM `property` GROUP BY name;
#### 结果如下####
name course 数学 语文 英语
张三          数学         3          4          5
李四          数学         6          7          8
王五          数学         9          10          11
```

中间 sql，便于理解：

```
SELECT name,course,
CASE course WHEN ' 数学' THEN score ELSE 0 END as ' 数学',
CASE course WHEN ' 语文' THEN score ELSE 0 END as ' 语文',
CASE course WHEN ' 英语' THEN score ELSE 0 END as ' 英语'
FROM `property`;
#### 结果如下####
name course 数学 语文 英语
张三   数学         3         0         0
张三          语文         0         4         0
张三          英语         0         0         5
李四          数学         6         0         0
李四          语文         0         7         0
李四          英语         0         0         8
王五          数学         9         0         0
王五          语文         0         10         0
王五          英语         0         0         11
```

10.10 表中文乱码问题

在建表的时候额外执行

```
ALTER TABLE camera CONVERT TO CHARACTER SET utf8;
```

10.11 MySQL 的 utf8 编码坑

曾几何时，每次建库都选 utf8，觉得自己比那些用乱七八糟编码的人不知道酷到哪里去了。直到好多年前的某次课程设计做项目的时候，愉快的建了个用户表：

```
CREATE TABLE `test_user` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

然后愉快的新增用户：INSERT INTO test_user(name) VALUES(" 我是")，接着愉快的反思人生：

```
Incorrect string value: '\xF0\x9F\x98\x81' for column 'name' at row 1
```

我是谁？我来自哪里？我在干嘛？难道是我代码里面的字符集用错了？不对啊我所有地方都用的 utf8 啊……

MySQL 的 UTF8 编码是什么？

首先来看官方文档：

```
The character set named utf8 uses a maximum of three bytes per character and contains␣
↪only BMP characters. The utf8mb4 character set uses a maximum of four bytes per␣
↪character supports supplementary characters:
For a BMP character, utf8 and utf8mb4 have identical storage characteristics: same code␣
↪values, same encoding, same length.
For a supplementary character, utf8 cannot store the character at all, whereas utf8mb4␣
↪requires four bytes to store it. Because utf8 cannot store the character at all, you␣
↪have no supplementary characters in utf8 columns and need not worry about converting␣
↪characters or losing data when upgrading utf8 data from older versions of MySQL.
```

我们再看看维基百科对 UTF8 编码的解释：

```
UTF-8 is a variable width character encoding capable of encoding all 1,112,064 valid␣
↪code points in Unicode using one to four 8-bit bytes.
```

可以看出，**MySQL 中的 utf8 实质上不是标准的 UTF8**。MySQL 中，utf8 对每个字符最多使用三个字节来表示.

"utf8" 只支持每个字符最多三个字节，而真正的 UTF-8 是每个字符最多四个字节。

MySQL 一直没有修复这个 bug，他们在 2010 年发布了一个叫作 "utf8mb4" 的字符集，绕过了这个问题。

当然，他们并没有对新的字符集广而告之（可能是因为这个 bug 让他们觉得很尴尬），以致于现在网络上仍然在建议开发者使用“utf8”，但这些建议都是错误的。

简单概括如下：

1.MySQL 的“utf8mb4”是真正的“UTF-8”。

2.MySQL 的“utf8”是一种“专属的编码”，它能够编码的 Unicode 字符并不多。

我要在这里澄清一下：所有在使用“utf8”的 MySQL 和 MariaDB 用户都应该改用“utf8mb4”，永远都不要再使用“utf8”。

这里 https://mathiasbynens.be/notes/mysql-utf8mb4#utf8-to-utf8mb4 提供了一个指南用于将现有数据库的字符编码从“utf8”转成“utf8mb4”。

10.12 left-join 常见的坑 (数据不足和冗余)

```
mysql> select * from role;
+----+-----------+
| id | role_name |
+----+-----------+
| 1 | 管理员 |
| 2 | 总经理 |
| 3 | 科长 |
| 4 | 组长 |
+----+-----------+
4 rows in set (0.00 sec)

mysql> select * from user;
+----+---------+-----------+------+
| id | role_id | user_name | sex |
+----+---------+-----------+------+
| 1 | 1 | admin | 1 |
| 2 | 2 | 王经理 | 1 |
| 3 | 2 | 李经理 | 2 |
| 4 | 2 | 张经理 | 2 |
| 5 | 3 | 王科长 | 1 |
| 6 | 3 | 李科长 | 1 |
| 7 | 3 | 吕科长 | 2 |
| 8 | 3 | 邢科长 | 1 |
| 9 | 4 | 范组长 | 2 |
| 10 | 4 | 赵组长 | 2 |
| 11 | 4 | 姬组长 | 1 |
```

(下页继续)

(续上页)

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

基本业务

简单信息报表: 查询用户信息

```
mysql> SELECT
-> id,
-> user_name AS '姓名',
-> ( CASE WHEN sex = 1 THEN '男' WHEN sex = 2 THEN '女' ELSE '未知' END ) AS '性别'
-> FROM
-> USER;
+----+-----------+--------+
| id | 姓名 | 性别 |
+----+-----------+--------+
| 1 | admin | 男 |
| 2 | 王经理 | 男 |
| 3 | 李经理 | 女 |
| 4 | 张经理 | 女 |
| 5 | 王科长 | 男 |
| 6 | 李科长 | 男 |
| 7 | 吕科长 | 女 |
| 8 | 邢科长 | 男 |
| 9 | 范组长 | 女 |
| 10 | 赵组长 | 女 |
| 11 | 姬组长 | 男 |
+----+-----------+--------+
```

查询每个角色名称及对应人员中女性数量

```
mysql> SELECT
-> r.id,
-> r.role_name AS role,
-> count( u.sex ) AS sex
-> FROM
-> role r
-> LEFT JOIN USER u ON r.id = u.role_id
-> AND u.sex = 2
-> GROUP BY
-> r.role_name
```

(下页继续)

(续上页)

```
-> ORDER BY
-> r.id ASC;
+----+-----------+-----+
| id | role | sex |
+----+-----------+-----+
| 1 | 管理员 | 0 |
| 2 | 总经理 | 2 |
| 3 | 科长 | 1 |
| 4 | 组长 | 2 |
+----+-----------+-----+
4 rows in set (0.00 sec)
```

假如我们把性别过滤的条件改为 where 操作结果会怎么样呢?

```
mysql> SELECT
-> r.id,
-> r.role_name AS role,
-> count( u.sex ) AS sex
-> FROM
-> role r
-> LEFT JOIN USER u ON r.id = u.role_id
-> WHERE
-> u.sex = 2
-> GROUP BY
-> r.role_name
-> ORDER BY
-> r.id ASC;
+----+-----------+-----+
| id | role | sex |
+----+-----------+-----+
| 2 | 总经理 | 2 |
| 3 | 科长 | 1 |
| 4 | 组长 | 2 |
+----+-----------+-----+
3 rows in set (0.00 sec)
```

这里可以看到角色数据不完整了。(虽然数据并没错, 但是数据显然是残缺的, **虽然是 0 但却是有意义的, 不应该自动隐藏掉**)

找出角色为总经理的员工数量

```
mysql> SELECT
-> r.id,
-> r.role_name AS role,
-> count( u.sex ) AS sex
-> FROM
-> role r
-> LEFT JOIN USER u ON r.id = u.role_id
-> WHERE
-> r.role_name = '总经理'
-> GROUP BY
-> r.role_name
-> ORDER BY
-> r.id ASC;
+----+-----------+-----+
| id | role | sex |
+----+-----------+-----+
| 2 | 总经理 | 3 |
+----+-----------+-----+
1 row in set (0.00 sec)
```

同样将过滤条件由 where 改为 on

```
mysql> SELECT
-> r.id,
-> r.role_name AS role,
-> count( u.sex ) AS sex
-> FROM
-> role r
-> LEFT JOIN USER u ON r.id = u.role_id
-> AND r.role_name = '总经理'
-> GROUP BY
-> r.role_name
-> ORDER BY
-> r.id ASC;
+----+-----------+-----+
| id | role | sex |
+----+-----------+-----+
| 1 | 管理员 | 0 |
| 2 | 总经理 | 3 |
| 3 | 科长 | 0 |
| 4 | 组长 | 0 |
```

(下页继续)

(续上页)

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

这里可以看到数据多余了

总结: 在 left join 语句中，**左表过滤必须放 where 条件中，右表过滤必须放 on 条件中**，这样结果才能不多不少，刚刚好。

类似案例参考:left join 注意事项

10.13 任何字段与 null 比较，结果都是 false(即使是 null 与 null 比较)

多个字段关联查询时，如果其中一个字段为 null，关联结果就是 false，比如 null = null and 1 = 1。

在写多字段关联的 sql 时，需要结合业务场景，考虑当其中一个字段为 null 时，本次关联还生不生效。

比如 a 和 b 都是学生表，两个表都有 s_id（学生 id）和 c_id（班级 id）列，求两个表共同的学生（学生 id 和班级 id 同时相等时，判定为同一个学生），sql 为:

```
select * from a, b where a.s_id = b.s_id and a.c_id = c.c_id
```

假如有一个学生，在 a、b 表都存在，但是 c_id 为空（比如该学生未排到具体班级），那么上面的 sql 就会漏掉该学生，需要把 sql 调整为:

```
select * from a, b
where
((a.c_id is null and b.c_id is null) or a.c_id = b.c_id)
and a.s_id = c.s_id
```

10.14 联合索引的最左匹配原则

10.15 IN 子句逻辑问题

这个是在给同事调 BUG 时发现的, 展示之前先初始化一些数据.

```
create table mysql_pitfalls(
c1 int,
c2 varchar(128),
c3 datetime,
c4 timestamp
```

(下页继续)

(续上页)

```
);

-- 插入测试数据
insert into mysql_pitfalls(c1,c2,c3,c4) values(1,'1',now(),now());
insert into mysql_pitfalls(c1,c2,c3,c4) values(2,'2',now(),now());
insert into mysql_pitfalls(c1,c2,c3,c4) values(3,'3',now(),now());
insert into mysql_pitfalls(c1,c2,c3,c4) values(4,'4',now(),now());
```

下面我们分别执行以下两条 SQL

```
mysql> select * from mysql_pitfalls where c1 in (1,2,3);
+------+------+---------------------+---------------------+
| c1   | c2   | c3                  | c4                  |
+------+------+---------------------+---------------------+
|    1 | 1    | 2015-06-06 19:00:05 | 2015-06-06 19:00:05 |
|    2 | 2    | 2015-06-06 19:00:08 | 2015-06-06 19:00:08 |
|    3 | 3    | 2015-06-06 19:00:11 | 2015-06-06 19:00:11 |
+------+------+---------------------+---------------------+
3 rows in set (0.00 sec)
```

这条 SQL 很简单,C1 列是数值型的,IN 逻辑正确. 接下面再看一句有逻辑问题的查询，去 IN 一个字符串–瞬间就被玩坏了

```
mysql> select * from mysql_pitfalls where c1 in ('1,2,3');
+------+------+---------------------+---------------------+
| c1   | c2   | c3                  | c4                  |
+------+------+---------------------+---------------------+
|    1 | 1    | 2015-06-06 19:00:05 | 2015-06-06 19:00:05 |
+------+------+---------------------+---------------------+
1 row in set, 1 warning (0.00 sec)
```

同样是数值型 C1 列, 查询如果 IN 的条件是一个带逗号的字符串,IN 条件会错误命中字符串中第一个逗号之前的数字. 虽然这条 SQL 写错了, 但这本身算是一个逻辑错误: 明明不相等,IN 去处怎么能匹配成功呢. 再者, 由于错误返回了每一条数据, 有时候会麻痹开发和测试，误认为功能没有问题.PS: 可以试试执行 IN ('1,2,3','2,3,4'), 会发现 MySQL 会求每一个带逗号字符串的第一个值.

10.16 更新时，表无法做为条件嵌套引用

我们尝试从表中删除一行时间最大的列，其实可选的方法不少. 但如果采用下面的语句，会抛出异常

```
mysql> delete from mysql_pitfalls where c4 = ( select max(c4) d from mysql_pitfalls);
ERROR 1093 (HY000): You can't specify target table 'mysql_pitfalls' for update in FROM␣
↪clause
```

如果只是想删除最大一行的话，绕开的方法不少，也不算上让人郁闷. 但最让人郁闷的是，其它这条 SQL 稍微改造，再嵌套一层使子查询彻底变成内存表就可以使用了，而 MySQL 自身并没有做这样的优化– 再嵌套一层就可以正常使用了

```
mysql> delete from mysql_pitfalls where c4 =
(select max(d) from ( select max(c4) d from mysql_pitfalls) a );
Query OK, 1 row affected (0.04 sec)
```

当然，这只能算 " 体验 " 问题.

10.17 Group By: 选取非分组列

这个可能是我遇到第一个让人奇怪的功能，算起来，这可能不算做 " 坑 " ，而更像是 MySQL 的特色.

在其它关系型数据库中，在拥 Group By 子句的情况下，只能 select 出被分组的列，对于非分组的列，除非使用聚合函数，这列将无法选择，原理也很简单，画一个二维表就能明白. 但是在 MySQL 中，即使在 Select 中选择了非分组，MySQL 也不会抛出语法错误，而是会默认选择这一组中这列的最后一个值.

从一方面来讲，这个功能提供了很多的灵活性. 但另一方面，由于这个过程是静默，不会提示. 会对开发和测试造成一定程度的麻痹（这种 SQL 不符合严谨的逻辑），而且造成了程序的不确定性. 这个功能可以通过显式修改 MySQL 运行模式，变得和普通 RDBMS 一致: MySQL :: MySQL 5.5 Reference Manual :: 5.1.7 Server SQL Modes

10.18 MySQL 时间加减的正确打开方式

研发 sql：update table set time = time + 1 where id=1;

看似好像挺对的，但是偶尔会出现不是想要的结果。

正确方式

```
为日期加上一个时间间隔：date_add()
date_add(@dt, interval 1 microsecond); -加 1 毫秒
date_add(@dt, interval 1 second); -加 1 秒
date_add(@dt, interval 1 minute); -加 1 分钟
date_add(@dt, interval 1 hour); -加 1 小时
date_add(@dt, interval 1 day); -加 1 天
date_add(@dt, interval 1 week); -加 1 周
```

(下页继续)

(续上页)

```
date_add(@dt, interval 1 month); -加 1 月
date_add(@dt, interval 1 quarter); -加 1 季
date_add(@dt, interval 1 year); -加 1 年
```

为日期减去一个时间间隔：date_sub()，格式同 date_add()

10.19 INT 长度并不能指定

常见的 int(4) 并不是指最大存储 9999，而是低于 4 位的数字会使用空格或 0 在左侧补齐到 4 位。这个 4 是显示宽度。

int 实际上是定长字段，占用 4 字节。取值范围固定是-(231-1) ~ (231-1)，无符号时为 0 ~ (232-1)。

10.20 VARCHAR 存储的是字符而不是字节，但最大长度是另外的算法

VARCHAR 声明的是字符数，但存储的是字节。

理论上最大长度是 65535 bytes，但实际上往往达不到。因为有几个因素：

65535 是单行数据的最大值，实际上除了 varchar，表里应该还会有其他字段

varchar 存放的字符串，往往会有多字节字符、一个字符占多个字节，而我们前端展现计算长度往往用的是字符数，所以也肯定达不到 65535

varchar 字段有长度标识位，可能占用 1~2 个字节，与我们声明的字段长度、字符集单字符最大长度有关，换算关系比较复杂，一句话说不清楚。

10.21 自增不一定连续，还可能重复

首先传递一个逻辑，MySQL InnoDB 的自增，是使用了一个表级的计数器。

自增不一定连续：如果 insert 或 update 指定了比当前最大值更大的值，计数器会直接增加到新的最大值；如果 delete 已有的一行数据，计数器并不会减小。

自增可能重复：上面提到的计数器，是维护在内存中的，MySQL 一旦重启、又没有手工重新装载过计数器，新插入记录自增主键就会重新从最小值开始，就会出现重复。(**现实中好像不需要这么做吧，所以这个应该不可信**)

10.22 TIMESTAMP 只能表达 68 年

TIMESTAMP 以 4 字节整数（可看做 SIGNED INT）存储从 1970-01-01T00:00:00Z（UNIX 纪元）经过的秒数。UNIX 纪元看做 0 值，小于纪元的时间插入会报错。

有符号 4 字节可以表达最大绝对值为 2^31-1 的数字，所以 TIMESTAMP 的只能表达 1970-01-01T00:00:01Z ~ 2038-01-19T03:14:07Z 的范围。

10.23 参考

MySQL 各种套路统计/坑/小知识点:https://blog.csdn.net/qq_30923243/article/details/85002269

【MySQL】各种小坑-持续更新:https://www.cnblogs.com/Ryan16231112/p/11849448.html

MySQL 联表查询基本操作之 left-join 常见的坑:www.cppcns.com/shujuku/mysql/313570.html

经验总结：被 MySQL UTF8 编码坑的惨痛教训:https://www.eet-china.com/mp/a17039.html

说一个 MySQL 里可能人人都会遇到的坑:https://cloud.tencent.com/developer/article/1478057

mysql sql 语句常见踩坑点：https://www.jianshu.com/p/c72275728520

left join 注意事项 (where 条件和 on 条件差别)：https://www.jianshu.com/p/d0f9b3c75b0d

你遇到过哪些 MySQL 的坑?：https://www.zhihu.com/question/22108510

MySQL 时间加减的正确打开方式：https://cloud.tencent.com/developer/article/1056404

初学者必看的 MySQL 坑：www.suoniao.com/article/5ea77f1244558d4a860517d6

mysql 数据库里的一些坑 (读高性能 mysql 有感)：https://blog.csdn.net/weixin_33816300/article/details/88908604

CHAPTER 11

数据库 11mysql 之坑 null 专题

11.1 空值”和“NULL”的概念

首先，我们要搞清楚“空值”和“NULL”的概念：

1、空值是不占用空间的

2、mysql 中的 NULL 其实是占用空间的，下面是来自于 MYSQL 官方的解释

```
“NULL columns require additional space in the row to record whether their values are␣
↪NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the␣
↪nearest byte.”
```

11.2 数字 +null = null 被气成傻逼

where 必须使用 is null 使用 = null 直接无效，问题是还不报错

null 各种神奇的设定，只此一条就被气成傻逼

11.3 count(column) 不会统计所有行

count(column) 会忽略掉值为 NULL 的行，相比于 count(*) 或 count(1)，统计出来的数字可能会小一些。从性能角度也不建议使用列。

11.4 mysql not in 丢失数据

all_student 表是所有的学生信息，fail_student 表是不及格的学生信息

```
select * from all_student
```

	name	age
▶	甲	18
	乙	18
	丙	18
	丁	18

```
select * from fail_student
```

	name	age
▶	丙	18
	丁	18

查询及格的学生信息

```
select * from all_student
where name not in (
select name from fail_student
)
```

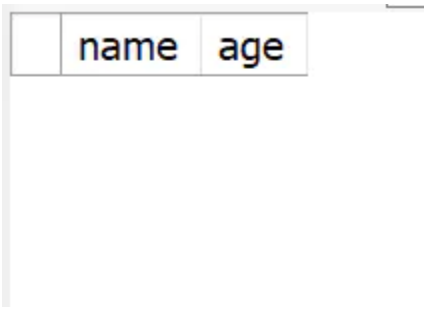
	name	age
▶	甲	18
	乙	18

假如 fail_student 有一条 name 为 null 的记录

	name	age
▶	丙	18
	丁	18
	NULL	18

再次查询及格的学生信息，结果集为空

```
select * from all_student
where name not in (
select name from fail_student
)
```



解决方案

修改表结构，设置 name 字段为 not null，并设置默认值

IFNULL 函数: SELECT IFNULL(a,0) FROM table WHERE 1;

11.5 MySQL 中 NOT IN 填坑之列为 null 的问题解决

SQL 中任意!=null 的运算结果都是 false，所以如果 t2 中存在一个 null，not in 的查询永远都会返回 false，即查询结果为空。

```
select COUNT(*) from t1 where t1.c1 not in (
select t2.c1 from t2 where t2.c1 is not null AND t2.c1 != ''
);
```

所以都是 null 引起的（为了避免错误我把空串也加上了），原因是 not in 的实现原理是，对每一个 t1.c1 和每一个 t2.c1 （括号内的查询结果）进行不相等比较（!=）。

```
foreach c1 in t2:
if t1.c1 != c1:
continue
else:
return false
return true
```

而 SQL 中任意!=null 的运算结果都是 false，所以如果 t2 中存在一个 null，not in 的查询永远都会返回 false，即查询结果为空。

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

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?

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。

11.7 MySQL 中 IS NULL、IS NOT NULL、!= 不能用索引？胡扯!

MySQL 中决定使不使用某个索引执行查询的依据很简单：就是成本够不够小。而不是是否在 WHERE 子句中用了 IS NULL、IS NOT NULL、!= 这些条件。大家以后也多多辟谣吧，没那么复杂，只是一个成本而已。

11.8 总结

1：NULL 作为布尔值的时候，不为 1 也不为 0

2：任何值和 NULL 使用运算符（>、<、>=、<=、!=、<>）或者（in、not in、any/some、all），返回值都为 NULL

3：当 IN 和 NULL 比较时，无法查询出为 NULL 的记录

4：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

5：判断是否为空只能用 IS NULL、IS NOT NULL

6：count(字段) 无法统计字段为 NULL 的值，count(*) 可以统计值为 null 的行

7：当字段为主键的时候，字段会自动设置为 not null

8：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

11.9 参考

mysql not in 丢失数据：https://www.jianshu.com/p/5898e54cdec9

MySQL 中 NOT IN 填坑之列为 null 的问题解决：www.111com.net/database/180180.htm

关于 mysql 的 null 相关查询的一些坑：https://www.cnblogs.com/mr-wuxiansheng/p/11578881.html

MySQL 中避免 NULL 的坑：https://blog.csdn.net/weixin_43894879/article/details/106306608

MySQL 中 IS NULL、IS NOT NULL、!= 不能用索引？胡扯！：https://blog.csdn.net/lonely_bin/article/details/99715968

CHAPTER 12

数据库 12 经验之谈

12.1 自增主键用完了怎么办?

在 mysql 中，Int 整型的范围（-2147483648~2147483648），约 20 亿！因此不用考虑自增 ID 达到最大值这个问题。而且数据达到千万级的时候就应该考虑分库分表了。

12.2 主键为什么不推荐有业务含义?

最好是主键是无意义的自增 ID，然后另外创建一个业务主键 ID，

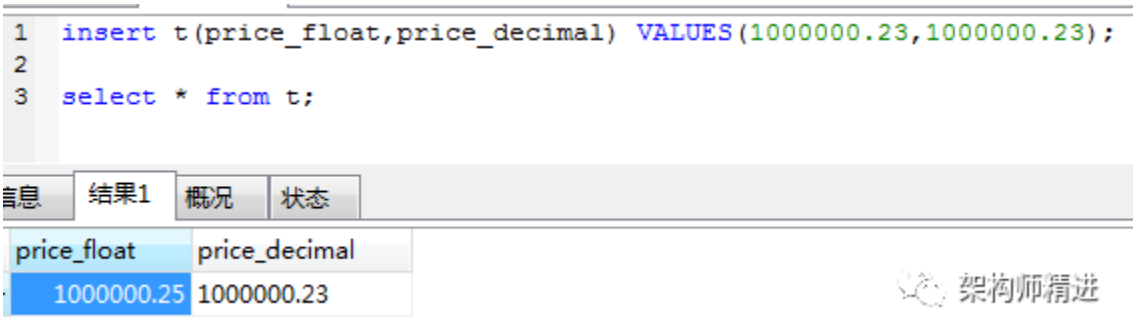
因为任何有业务含义的列都有改变的可能性，主键一旦带上了业务含义，那么主键就有可能发生变更。**主键一旦发生变更，该数据在磁盘上的存储位置就会发生变更，有可能会引发页分裂，产生空间碎片**。

还有就是，带有**业务含义的主键，不一定是顺序自增的。那么就会导致数据的插入顺序，并不能保证后面插入数据的主键一定比前面的数据大**。如果出现了，后面插入数据的主键比前面的小，就有可能引发页分裂，产生空间碎片。

12.3 货币字段用什么类型?

货币字段一般都用 Decimal 类型，

float 和 double 是以二进制存储的，数据大的时候，可能存在误差。看下面这个图就明白了:



12.4 时间字段用什么类型?

这个看具体情况和实际场景，timestamp ，datatime ，bigint 都行！把理由讲清楚就行！

timestamp，该类型是四个字节的整数，它能表示的时间范围为 1970-01-01 08:00:01 到 2038-01-19 11:14:07。2038 年以后的时间，是无法用 timestamp 类型存储的。

但是它有一个优势，timestamp 类型是带有时区信息的。一旦你系统中的时区发生改变，例如你修改了时区，该字段的值会自动变更。这个特性用来做一些国际化大项目，跨时区的应用时，特别注意！

datetime，占用 8 个字节，它存储的时间范围为 1000-01-01 00:00:00 ~ 9999-12-31 23:59:59。显然，存储时间范围更大。但是它坑的地方在于，它存储的是时间绝对值，不带有时区信息。如果你改变数据库的时区，该项的值不会自己发生变更！

bigint，也是 8 个字节，自己维护一个时间戳，查询效率高，不过数据写入，显示都需要做转换。

12.5 为什么不直接存储图片、音频、视频等大容量内容?

我们在实际应用中，都是文件形式存储的。mysql 中，只存文件的存放路径。虽然 mysql 中 blob 类型可以用来存放大容量文件，但是，我们在生产中，基本不用！

主要有如下几个原因：

```
1. Mysql 内存临时表不支持 TEXT、BLOB 这样的大数据类型，如果查询中包含这样的数据，查询效率会非常慢。
2. 数据库特别大，内存占用高，维护也比较麻烦。
3. binlog 太大，如果是主从同步的架构，会导致主从同步效率问题！
```

因此，不推荐使用 blob 等类型！

12.6 表中有大字段 X(例如：text 类型)，且字段 X 不会经常更新，以读为主，那么是拆成子表好？还是放一起好？

其实各有利弊，拆开带来的问题：连接消耗；不拆可能带来的问题：查询性能，所以要看你的实际情况，如果表数据量比较大，最好还是拆开为好。这样查询速度更快。

12.7 字段为什么要定义为 NOT NULL?

一般情况，都会设置一个默认值，不会出现字段里面有 null，又有空的情况。主要有以下几个原因：

1. 索引性能不好，Mysql 难以优化引用可空列查询，它会使索引、索引统计和值更加复杂。可空列需要更多的存储空间，还需要 mysql 内部进行特殊处理。可空列被索引后，每条记录都需要一个额外的字节，还能导致 MYisam 中固定大小的索引变成可变大小的索引。
2. 如果某列存在 null 的情况，可能导致 count() 等函数执行不对的情况。看一下 2 个图就明白了：
3. sql 语句写着也麻烦，既要判断是否为空，又要判断是否为 null 等。

12.8 where 执行顺序是怎样的？

where 条件**从左往右**执行的，在数据量小的时候不用考虑，但数据量多的时候要考虑条件的先后顺序，此时应遵守一个原则：**排除越多的条件放在第一个**。

12.9 应该在这些列上创建索引：

在经常需要**搜索的列**上，可以加快搜索的速度；在作为**主键的列**上，强制该列的唯一性和组织表中数据的排列结构；在经常用在**连接的列**上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据**范围进行搜索**的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要**排序的列**上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在 **WHERE 子句中的列**上面创建索引，加快条件的判断速度。

12.10 什么是最左前缀原则？

最左前缀原则指的是，如果查询的时候查询条件精确匹配索引的左边连续一列或几列，则此列就可以被用到。如下：

```
select * from user where name=xx and city=xx ; / / 可以命中索引
select * from user where name=xx ; // 可以命中索引
select * from user where city=xx ; // 无法命中索引
```

这里需要注意的是，查询的时候如果两个条件都用上了，但是顺序不同，如 city= xx and name = xx，那么现在的查询引擎会自动优化为匹配联合索引的顺序，这样是能够命中索引的。

由于最左前缀原则，在创建联合索引时，索引字段的顺序需要考虑**字段值去重之后的个数，较多的放前面**。ORDER BY 子句也遵循此规则。

12.11 什么情况下应不建或少建索引

表记录太少

经常插入、删除、修改的表

数据重复且分布平均的表字段，假如一个表有 10 万行记录，有一个字段 A 只有 T 和 F 两种值，且每个值的分布概率大约为 50%，那么对这种表 A 字段建索引一般不会提高数据库的查询速度。

经常和主字段一块查询但主字段索引值比较多的表字段

12.12 问了下 MySQL 数据库 cpu 飙升到 100% 的话他怎么处理？

1. 列出所有进程 show processlist 观察所有进程多秒没有状态变化的 (干掉)
2. 查看慢查询，找出执行时间长的 sql；explain 分析 sql 是否走索引，sql 优化；
3. 检查其他子系统是否正常，是否缓存失效引起，需要查看 buffer 命中率；

12.13 索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关, 在 MySQL 中使用较多的索引有 Hash 索引,B+ 树索引等, 而我们经常使用的 InnoDB 存储引擎的默认索引实现为 + 树索引.

12.14 Hash 索引和 B+ 树所有有什么区别或者说优劣呢?

首先要知道 Hash 索引和 B+ 树索引的底层实现原理:

hash 索引底层就是 hash 表, 进行查找时, 调用一次 hash 函数就可以获取到相应的键值, 之后进行回表查询获得实际数据.B+ 树底层实现是多路平衡查找树. 对于每一次的查询都是从根节点出发, 查找到叶子节点方可以获得所查键值, 然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

hash 索引进行等值查询更快 (一般情况下), 但是却无法进行范围查询.

因为在 hash 索引中经过 hash 函数建立索引之后, 索引的顺序与原顺序无法保持一致, 不能支持范围查询. 而 B+ 树的的所有节点皆遵循 (左节点小于父节点, 右节点大于父节点, 多叉树也类似), 天然支持范围.

hash 索引不支持使用索引进行排序, 原理同上.

hash 索引不支持模糊查询以及多列索引的最左前缀匹配. 原理也是因为 hash 函数的不可预测.AAAA 和 AAAAB 的索引没有相关性.

hash 索引任何时候都避免不了回表查询数据, 而 B+ 树在符合某些条件 (聚簇索引, 覆盖索引等) 的时候可以只通过索引完成查询.

hash 索引虽然在等值查询上较快, 但是不稳定. 性能不可预测, 当某个键值存在大量重复的时候, 发生 hash 碰撞, 此时效率可能极差. 而 B+ 树的查询效率比较稳定, 对于所有的查询都是从根节点到叶子节点, 且树的高度较低.

因此, 在大多数情况下, 直接选择 B+ 树索引可以获得稳定且较好的查询速度. 而不需要使用 hash 索引.

12.15 那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?

使用不等于查询,

列参与了数学运算或者函数

在字符串 like 时左边是通配符. 类似于'%aaa'.

当 mysql 分析全表扫描比使用索引快的时候不使用索引.

当使用联合索引, 前面一个条件为范围查询, 后面的即使符合最左前缀原则, 也无法使用索引.

以上情况,MySQL 无法使用索引.

12.16 为什么使用数据索引能提高效率

数据索引的存储是有序的

在有序的情况下，通过索引查询一个数据是无需遍历索引记录的

极端情况下，数据索引的查询效率为二分法查询效率，趋近于 log2(N)

12.17 B+ 树索引和哈希索引的区别

参考文献：mysql 常见面试题转载附录

12.18 哈希索引的优势

参考文献：mysql 常见面试题转载附录

12.19 B 树和 B+ 树的区别

参考文献：mysql 常见面试题转载附录

12.20 为什么说 B+ 比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

参考文献：mysql 常见面试题转载附录

12.21 其他

2. 主键自增 id 适合设置为无符号的 int 类型，这样最大值可以增加一倍：4294967295(2 的 32 次方减一)。

5.char 类型存储的数据长度小于最大长度时会用空格填充，检索时再剔除，因此如果存入的 string 最后有空格，查询出来是没有的。

7. 时间戳格式 timestamp 和 datetime，前者占用空间小（4bytes）且与时区相关，优先使用。除非是范围超过了 timestamp 的范围（1970~2038），不推荐使用 int 类型。

10.IPv4 地址可以保存为无符号 int 类型，因为它实际上是一个 32 位的无符号整数，使用 mysql 函数（INET_ATON 和 INET_NTOA）进行转换。

11. 范式与反范式的使用并不是绝对性的，需要根据自己的业务和数据量合理折中使用。数据量大查询频率高的时候适当的建立冗余字段减少关联，而数据少关联紧密的场合遵循范式化设计。

12.sql 中 limit 5 表示搜索前五条记录，limit 5,10 检索 6-10 条记录，limit 5,-1 表示 6-last 条记录。

12.22 MySQL 军规

基础规范

必须使用 InnoDB 存储引擎

解读：支持事务、行级锁、并发性能更好、CPU 及内存缓存页优化使得资源利用率更高。

表字符集默认使用 utf8，必要时候使用 utf8mb4

解读：万国码，无需转码，无乱码风险，节省空间，utf8mb4 是 utf8 的超集，有存储 4 字节例如表情符号时，使用它。

数据表、数据字段必须加入中文注释

禁止使用存储过程、视图、触发器、Event

解读：高并发大数据的互联网业务，架构设计思路是“解放数据库 CPU，将计算转移到服务层”，并发量大的情况下，这些功能很可能将数据库拖死，业务逻辑放到服务层具备更好的扩展性，能够轻易实现“增机器就加性能”。数据库擅长存储与索引，CPU 计算还是上移吧。

禁止存储大文件或者大照片

解读：为何要让数据库做它不擅长的事情？大文件和照片存储在文件系统，数据库里存 URI 多好。

控制单表数据量，单表记录控制在千万级

平衡范式与冗余，为提高效率可以牺牲范式设计，冗余数据

命名规范

线上环境、开发环境、测试环境数据库内网域名遵循命名规范

```
业务名称：xxx
线上环境：dj.xxx.db
开发环境：dj.xxx.rdb
测试环境：dj.xxx.tdb
```

库名、表名、字段名：小写，下划线风格，不超过 32 个字符，必须见名知意，禁止拼音英文混用

表名 t_xxx，非唯一索引名 idx_xxx，唯一索引名 uniq_xxx

表设计规范

单表列数目必须小于 30

表必须有主键，例如自增主键，推荐使用 UNSIGNED 整数为主键

解读：

主键递增，数据行写入可以提高插入性能，可以避免 page 分裂，减少表碎片，提升空间和内存的使用；

主键要选择较短的数据类型，InnoDB 引擎普通索引都会保存主键的值，较短的数据类型可以有效的减少索引的磁盘空间，提高索引的缓存效率；

无主键的表删除，在 row 模式的主从架构，会导致备库夯住；

禁止使用外键，如果有外键完整性约束，需要应用程序控制

解读：外键会导致表与表之间耦合，update 与 delete 操作都会涉及相关联的表，十分影响 SQL 的性能，甚至会造成死锁。高并发情况下容易造成数据库性能，大数据高并发业务场景数据库使用以性能优先。

建议将大字段，访问频度低的字段拆分到单独的表中存储，分离冷热数据

字段设计规范

必须把字段定义为 NOT NULL 并且提供默认值

解读：

• null 的列使索引/索引统计/值比较都更加复杂，对 MySQL 来说更难优化；• null 这种类型 MySQL 内部需要进行特殊处理，增加数据库处理记录的复杂性；同等条件下，表中有较多空字段的时候，数据库的处理

性能会降低很多；• null 值需要更多的存储空，无论是表还是索引中每行中的 null 的列都需要额外的空间来标识；• 对 null 的处理时候，只能采用 is null 或 is not null，而不能采用 =、in、<、<>、!=、not in 这些操作符号。如：where name != 'shenjian'，如果存在 name 为 null 值的记录，查询结果就不会包含 name 为 null 值的记录；

禁止使用 TEXT、BLOB 类型

解读：会浪费更多的磁盘和内存空间，非必要的大量的大字段查询会淘汰掉热数据，导致内存命中率急剧降低，影响数据库性能。

禁止使用小数存储货币

解读：使用整数吧，小数容易导致钱对不上。

必须使用 varchar(20) 存储手机号

解读：• 涉及到区号或者国家代号，可能出现 +-()；

- varchar 可以支持模糊查询，例如：like “138%”；

使用 INT UNSIGNED 存储 IPv4，不要用 char(15)

根据业务区分使用 char/varchar

解读：

- 字段长度固定，或者长度近似的业务场景，适合使用 char，能够减少碎片，查询性能高；
- 字段长度相差较大，或者更新较少的业务场景，适合使用 varchar，能够减少空间；

根据业务区分使用 datetime/timestamp

解读：前者占用 5 个字节，后者占用 4 个字节，存储年使用 YEAR，存储日期使用 DATE，存储时间使用 datetime

索引设计规范

单表索引建议控制在 5 个以内

解读：

- 互联网高并发业务，太多索引会影响写性能；
- 生成执行计划时，如果索引太多，会降低性能，并可能导致 MySQL 选择不到最优索引；
- 异常复杂的查询需求，可以选择 ES 等更为适合的方式存储；

单索引字段数不允许超过 5 个

解读：字段超过 5 个时，实际已经起不到有效过滤数据的作用了。

禁止在更新十分频繁、区分度不高的属性上建立索引解读：

- 更新会变更 B+ 树，更新频繁的字段建立索引会大大降低数据库性能；
- 【性别】这种区分度不大的属性，建立索引是没有什么意义的，不能有效过滤数据，性能与全表扫描类似；

建立组合索引，必须把区分度高的字段放在前面

解读：能够更加有效的过滤数据。

非必要不要进行 JOIN 查询，如果要进行 JOIN 查询，被 JOIN 的字段必须类型相同，并建立索引。

理解组合索引最左前缀原则，避免重复建设索引，如果建立了 (a,b,c)，相当于建立了 (a), (a,b), (a,b,c)

SQL 使用规范

禁止使用 SELECT *，只获取必要的字段，需要显示说明列属性

解读：

- 读取不需要的列会增加 CPU、IO、内存、网络带宽消耗；
- 不能有效的利用覆盖索引；
- 使用 SELECT * 容易在增加或者删除字段后出现程序 BUG；

禁止使用 INSERT INTO t_xxx VALUES(xxx)，必须显示指定插入的列属性解读：容易在增加或者删除字段后出现程序 BUG。

禁止使用属性隐式转换

解读：WHERE 子句中出现 COLUMN 字段的类型和传入的参数类型不一致的时候发生的类型转换，建议先确定 WHERE 中的参数类型。

SELECT uid FROM t_user WHERE phone=13812345678 会导致全表扫描，而不能命中 phone 索引。

禁止在 WHERE 条件的属性上使用函数或者表达式解读：SELECT uid FROM t_user WHERE from_unixtime(day)>='2017-02-15' 会导致全表扫描。

正确的写法是：SELECT uid FROM t_user WHERE day>= unix_timestamp('2017-02-15 00:00:00')。

** 禁止负向查询，以及% 开头的模糊查询 **

解读：

- 负向查询条件：NOT、!=、<>、!<、!>、NOT IN、NOT LIKE 等，会导致全表扫描；
- % 开头的模糊查询，会导致全表扫描；

禁止大表使用 JOIN 查询，禁止大表使用子查询

解读：会产生临时表，消耗较多内存与 CPU，极大影响数据库性能。

禁止使用 OR 条件，必须改为 IN 查询或者 UNION 查询，IN 的值必须少于 50 个

解读：旧版本 MySQL 的 OR 查询是不能命中索引的，即使能命中索引，为何要让数据库耗费更多的 CPU 帮助实施查询优化呢。

尽量使用 UNION ALL 替代 UNION，UNION 有去重开销

解读：UNION 和 UNION ALL 的差异主要是前者需要将结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的 CPU 运算，加大资源消耗及延迟。当然，使用 UNION ALL 的前提条件是两个结果集没有重复数据。

区分 IN 和 EXISTS 的使用场景

解读：

SELECT * FROM table_a WHERE id IN (SELECT id FROM table_b)

上面 SQL 语句相当于：

SELECT * FROM table_a WHERE EXISTS (SELECT * FROM table_b WHERE table_b.id = table_a.id)

区分 IN 和 EXISTS 的使用场景，主要参考两者的驱动顺序（这时性能变化的关键）。如果是 IN，会以内层表为驱动表，先执行子查询，**所以 IN 适合外表大而内表小的情况**；如果是 EXISTS，会以外层表为驱动表，先执行外表，所以 **EXISTS 适合外表小而内表大**的情况。

区分 NOT IN 和 NOT EXISTS 的使用场景

解读：关于 NOT IN 和 NOT EXISTS，推荐使用 NOT EXISTS，不仅仅是效率问题，NOT IN 可能存在逻辑问题。

使用左关联的写法代替 NOT EXISTS

解读：

原 SQL 语句：

```
SELECT * FROM table_a WHERE NOT EXISTS (SELECT * FROM table_b WHERE table_b.id = table_a.
↪id)
```

高效的 SQL 语句：

```
SELECT * FROM table_a LEFT JOIN table_b ON table_a.id = table_b.id WHERE table_b.id IS␣
↪NULL
```

避免在 WHERE 子句中对字段进行 NULL 值判断

解读：对于 NULL 的判断会导致引擎放弃使用索引而进行全表扫描。

关于 JOIN 的优化

解读：

- LEFT JOIN：左表是驱动表，右表是被驱动表
- RIGHT JOIN：右表是驱动表，左表是被驱动表
- INNER JOIN：MySQL 会选择数据量比较小的表作为驱动表，大表作为被驱动表

优化原则：

1. 尽量使用 INNER JOIN ，避免 LEFT JOIN & RIGHT JOIN
2. 被驱动表的索引字段作为 ON 的限制字段
3. 利用小表去驱动大表

12.23 参考

能避开很多坑的 mysql 面试题，你知道吗?：https://cloud.tencent.com/developer/article/1553999

100 道 MySQL 常见面试题总结：https://article.itxueyuan.com/eoJEMj

mysql 常见面试题转载附录 (B+ 树和 hash 区别)：https://www.cnblogs.com/williamjie/p/11081592.html

mysql 数据库里的一些坑 (读高性能 mysql 有感)：https://blog.csdn.net/weixin_33816300/article/details/88908604

MySQL 军规：https://www.yuque.com/yinjianwei/vyrvkf/mpu8gk

58 到家 MySQL 军规升级版：https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651961030&idx=1&sn=

赶集 mysql 军规：https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651960775&idx=1&sn=1a9c9f4b94

CHAPTER 13

数据库 13mysql 执行计划

13.1 EXPLAIN 用法详解

举例：EXPLAIN SELECT ……

变体：1. EXPLAIN EXTENDED SELECT ……将执行计划“反编译”成 SELECT 语句，运行 SHOW WARNINGS 可得到被 MySQL 优化器优化后的查询语句

```
mysql> explain SELECT t2.*
    -> FROM t2
    -> WHERE id = (SELECT id
    ->               FROM t1
    ->               WHERE id = (SELECT t3.id
    ->                           FROM t3
    ->                           WHERE t3.other_column = ''));
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
|  1 | PRIMARY     | t2    | const | PRIMARY       | PRIMARY | 4       | const |    1 |             |
|  2 | SUBQUERY    | t1    | const | PRIMARY       | PRIMARY | 4       |       |    1 | Using index |
|  3 | SUBQUERY    | t3    | ALL   | NULL          | NULL    | NULL    | NULL  |    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
3 rows in set (0.00 sec)
```

13.1.1 id

select 查询的序列号，标识执行的顺序

```
id 相同，执行顺序由上至下
id 不同，如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行
```

13.1.2 select_type

查询的类型，主要是用于区分普通查询、联合查询、子查询等。

```
SIMPLE: 简单的 select 查询，查询中不包含子查询或者 union
PRIMARY: 查询中包含子部分，最外层查询则被标记为 primary
SUBQUERY/MATERIALIZED: SUBQUERY 表示在 select 或 where 列表中包含了子查询，MATERIALIZED: 表示 where 后面 in 条件的子查询
UNION: 表示 union 中的第二个或后面的 select 语句
UNION RESULT: union 的结果
```

13.1.3 table

查询涉及到的表。

```
直接显示表名或者表的别名
<unionM,N> 由 ID 为 M，N 查询 union 产生的结果
<subqueryN> 由 ID 为 N 查询产生的结果
```

13.1.4 type

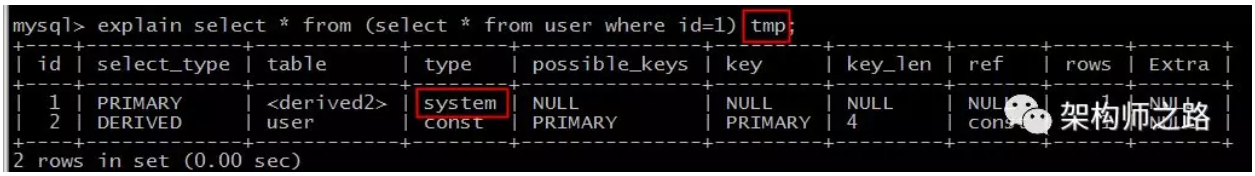
访问类型，SQL 查询优化中一个很重要的指标，结果值从好到坏依次是：system > const > eq_ref > ref > range > index > ALL。**查询至少达到 range 级别，最好能达到 ref**。

```
system: 系统表，少量数据，往往不需要进行磁盘 IO
const: 常量连接,PK 或者 unique 上的等值查询
eq_ref: 主键索引 (primary key) 或者非空唯一索引 (unique not null) 等值扫描
ref: 非唯一索引，等值匹配，可能有多行命中
range: 索引上的范围扫描，例如：between、in、>
index: 索引上的全集扫描，例如：InnoDB 的 count
ALL: 全表扫描 (full table scan) ，
```

举例:

```
system:explain select * from mysql.time_zone;
从系统库 MySQL 的系统表 time_zone 里查询数据，访问类型为 system，这些数据已经加载到内存里，不需要进行磁盘 IO，这类扫描是速度最快的


explain select * from (select * from user where id=1) tmp;
```



再举一个例子，内层嵌套（const）返回了一个临时表，外层嵌套从临时表查询，其扫描类型也是 system，也不需要走磁盘 IO，速度超快。

const 扫描的条件为：

```
命中主键（primary key）或者唯一（unique）索引
被连接的部分是一个常量（const）值
```

如上例，id 是主键索引，连接部分是常量 1。

eq_ref 扫描的条件为，对于前表的每一行（row），后表只有一行被扫描。

再细化一点：

```
join 查询
命中主键（primary key）或者非空唯一（unique not null）索引
等值连接；
```

ref 扫描，可能出现在 join 里，也可能出现在单表普通索引里，每一次匹配可能有多行数据返回，虽然它比 eq_ref 要慢，但它仍然是一个很快的 join 类型。

range 扫描就比较好理解了，它是索引上的范围查询，它会在索引上扫码特定范围内的值。

index: 该 count 查询需要通过扫描索引上的全部数据来计数，它仅比全表扫描快一点。

```
explain count (*) from user;
```

13.1.5 possible_keys

指出 MySQL 能使用哪个索引在表中找到行，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用

13.1.6 key

显示 MySQL 在查询中实际使用的索引，若没有使用索引，显示为 NULL

TIPS：查询中若使用了覆盖索引，则该索引仅出现在 key 列表中

13.1.7 key_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度

13.1.8 ref

表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

13.1.9 rows

表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

13.1.10 Extra

十分重要的额外信息。

```
Using filesort：MySQL 对数据使用一个外部的文件内容进行了排序，而不是按照表内的索引进行排序读取。
典型的，在一个没有建立索引的列上进行了 order by，就会触发 filesort，常见的优化方案是，在␣
↪order by 的列上添加索引，避免每次查询都全量排序。

Using temporary：使用临时表保存中间结果，也就是说 MySQL 在对查询结果排序时使用了临时表，常见于 order by 或 group by。
临时表存在两种引擎，一种是 Memory 引擎，一种是 MyISAM 引擎，如果返回的数据在 16M 以内（默认），且没有大字段的情况下，使用 Memory 引擎，否则使用 MyISAM 引擎。

Using index：表示 SQL 操作中使用了覆盖索引（Covering Index），避免了访问表的数据行，效率高。
Using index condition：表示 SQL 操作命中了索引，但不是所有的列数据都在索引树上，还需要访问实际的行记录。
Using where：表示 SQL 操作使用了 where 过滤条件。
Select tables optimized away：基于索引优化 MIN/MAX 操作或者 MyISAM 存储引擎优化 COUNT(*) 操作，不必等到执行阶段再进行计算，查询执行计划生成的阶段即可完成优化。
Using join buffer (Block Nested Loop)：表示 SQL 操作使用了关联查询或者子查询，且需要进行嵌套循环计算。
```

13.2 关于 MySQL 执行计划的局限总结如下：

1.EXPLAIN 不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况

2.EXPLAIN 不考虑各种 Cache

3.EXPLAIN 不能显示 MySQL 在执行查询时所作的优化工作

4. 部分统计信息是估算的，并非精确值

5.EXPALIN 只能解释 SELECT 操作，其他操作要重写为 SELECT 后查看执行计划

13.3 对于非 select 语句查看执行计划

在实际的工作中也经常需要查看一些诸如 update、delete 的执行计划，（mysql5.6 的版本已经支持直接查看）但是这时候并不能直接通过 explain 来进行查看，而需要通过改写语句进行查看执行计划；

13.4 参考

MySQL 执行计划分析工具 EXPLAIN 用法详解：blog.chinaunix.net/uid-25723371-id-5598143.html

MYSQL 查看执行计划：https://blog.51cto.com/xiaocao13140/2126580

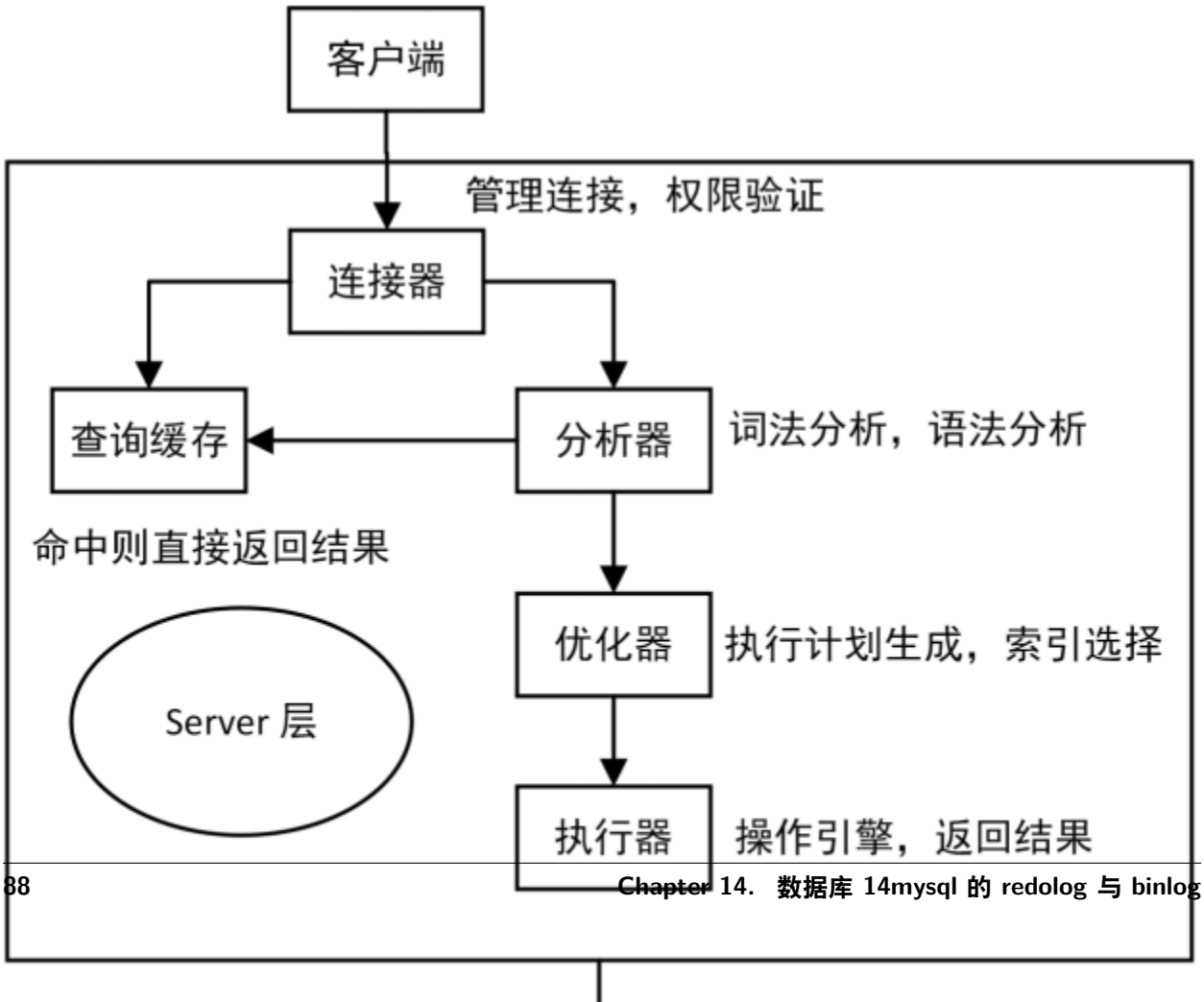
MySQL 执行计划详解：https://www.cnblogs.com/yinjw/p/11864477.html

MySQL——执行计划 (较好的案例列表,可作为练习手册)：https://www.cnblogs.com/sunjingwu/p/10755823.html

CHAPTER 14

数据库 14mysql 的 redolog 与 binlog

14.1 SQL 语句执行链路



14.2 什么是 redo log？

redo log 又称重做日志文件，用于记录事务操作的变化，记录的是数据修改之后的值，不管事务是否提交都会记录下来。在实例和介质失败（media failure）时，redo log 文件就能派上用场，如数据库掉电，InnoDB 存储引擎会使用 redo log 恢复到掉电前的时刻，以此来保证数据的完整性。

14.3 什么是 binlog

binlog 记录了对 MySQL 数据库执行更改的所有操作，但是不包括 SELECT 和 SHOW 这类操作，因为这类操作对数据本身并没有修改。然后，若操作本身并没有导致数据库发生变化 (比如 update 但条件筛选后为 0，实际没有真正 update)，那么该操作也会写入二进制日志。

14.4 redo log 与 binlog 的区别

第一:**redo log 是在 InnoDB 存储引擎**层产生，而 **binlog 是 MySQL 数据库的上层**产生的，并且二进制日志不仅仅针对 INNODB 存储引擎，MySQL 数据库中的**任何存储引擎对于数据库的更改都会产生二进制日志**。

第二：两种日志记录的内容形式不同。**MySQL 的 binlog 是逻辑日志**，其记录是对应的 SQL 语句。而**innodb 存储引擎层面的重做日志是物理日志**。第三：两种日志与记录写入磁盘的时间点不同，**二进制日志只在事务提交完成后进行一次写入**。而 innodb 存储引擎的重做日志**在事务进行中不断地被写入**，并日志不是随事务提交的顺序进行写入的。

二进制日志仅在事务提交时记录，并且对于每一个事务，仅在事务提交时记录，并且对于每一个事务，仅包含对应事务的一个日志。而对于 innodb 存储引擎的重做日志，**由于其记录是物理操作日志，因此每个事务对应多个日志条目，并且事务的重做日志写入是并发的，并非在事务提交时写入，其在文件中记录的顺序并非是事务开始的顺序**。

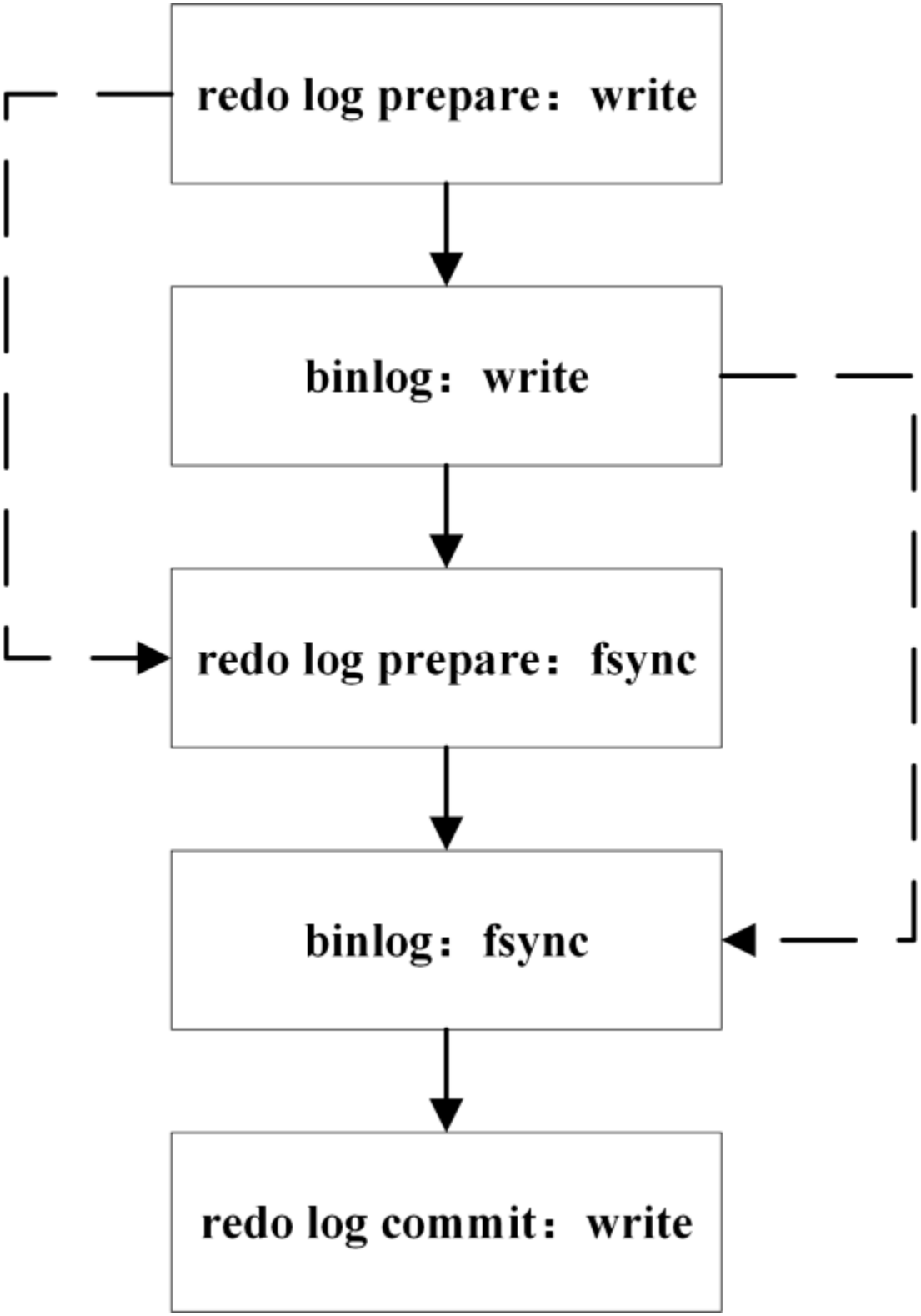
第四：**binlog 不是循环使用**，在写满或者重启之后，会生成新的 binlog 文件，**redo log 是循环使用**。

第五：**binlog 可以作为恢复数据使用，主从复制搭建**，**redo log 作为异常宕机或者介质故障后的数据恢复使用**。

14.5 两阶段提交（2PC）

MySQL 使用两阶段提交主要解决 binlog 和 redo log 的数据一致性的问题。

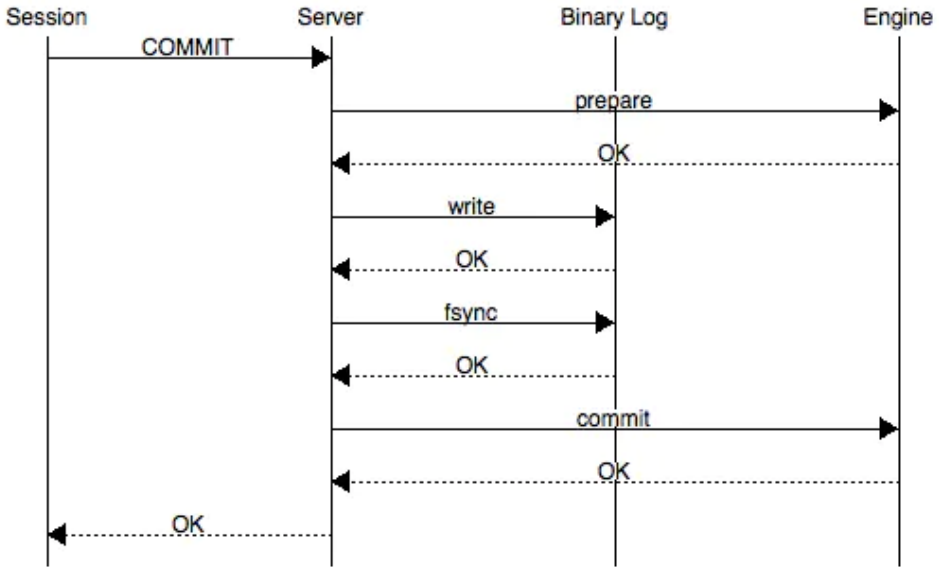
redo log 和 binlog 都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态保持逻辑上的一致。下图为 MySQL 二阶段提交简图：


redo log prepare：write
binlog：write
redo log prepare：fsync
binlog：fsync
redo log commit：write

两阶段提交原理描述:

```
InnoDB redo log 写盘，InnoDB 事务进入 prepare 状态。
如果前面 prepare 成功，binlog 写盘，那么再继续将事务日志持久化到 binlog，如果持久化成功，那么 InnoDB 事务则进入 commit 状态（在 redo log 里面写一个 commit 记录）
备注：每个事务 binlog 的末尾，会记录一个 XID event，标志着事务是否提交成功，也就是说，recovery 过程中，binlog 最后一个 XID event 之后的内容都应该被 purge。
```

Binlog 在 2PC 中充当了事务的协调者（Transaction Coordinator）。由 Binlog 来通知 InnoDB 引擎来执行 prepare，commit 或者 rollback 的步骤。事务提交的整个过程如下：



由上面的二阶段提交流程可以看出，通过**两阶段提交方式保证了无论在任何情况下，事务要么同时存在于存储引擎和 binlog 中，要么两个里面都不存在，可以保证事务的 binlog 和 redo log 顺序一致性**。一旦阶段 2 中持久化 Binlog 完成，就确保了事务的提交。此外需要注意的是，每个阶段都需要进行一次 fsync 操作才能保证上下两层数据的一致性。阶段 1 的 fsync 由参数 innodb_flush_log_at_trx_commit=1 控制，阶段 2 的 fsync 由参数 sync_binlog=1 控制，俗称“双 1”，是保证 crash-safe 的根本。

14.6 redo log 和 binlog 是怎么关联起来的?

redo log 和 binlog 有一个共同的数据字段，叫 XID。崩溃恢复的时候，会按顺序扫描 redo log：

```
如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；
如果碰到只有 parepare、而没有 commit 的 redo log，就拿着 XID 去 binlog 找对应的事务。
```

14.7 参考

MySQL redo log 与 binlog 的区别：https://blog.csdn.net/wanbin6470398/article/details/81941586

MySQL 日志系统之 redo log 和 binlog：https://www.cnblogs.com/yanglang/p/11758606.html

MySQL 的 Binlog 与 Redolog：https://www.jianshu.com/p/65eb0526bfc0

CHAPTER 15

Indices and tables

- genindex
- modindex
- search