

MySQL

简介

数据库

数据库： DataBase，简称 DB，存储和管理数据的仓库

数据库的优势：

- 可以持久化存储数据
- 方便存储和管理数据
- 使用了统一的方式操作数据库 SQL

数据库、数据表、数据的关系介绍：

- 数据库
 - 用于存储和管理数据的仓库
 - 一个库中可以包含多个数据表
- 数据表
 - 数据库最重要的组成部分之一
 - 由纵向的列和横向的行组成（类似 excel 表格）
 - 可以指定列名、数据类型、约束等
 - 一个表中可以存储多条数据
- 数据：想要永久化存储的数据

参考视频：<https://www.bilibili.com/video/BV1zJ411M7TB>

参考专栏：<https://time.geekbang.org/column/intro/139>

参考书籍：<https://book.douban.com/subject/35231266/>

MySQL

MySQL 数据库是一个最流行的关系型数据库管理系统之一，关系型数据库是将数据保存在不同的数据表中，而且表与表之间可以有关联关系，提高了灵活性

缺点：数据存储在磁盘中，导致读写性能差，而且数据关系复杂，扩展性差

MySQL 所使用的 SQL 语句是用于访问数据库最常用的标准化语言

MySQL 配置：

- MySQL 安装：<https://www.jianshu.com/p/ba48f1e386f0>
- MySQL 配置：
 - 修改 MySQL 默认字符集：安装 MySQL 之后第一件事就是修改字符集编码

```
vim /etc/mysql/my.cnf

添加如下内容:
[mysqld]
character-set-server=utf8
collation-server=utf8_general_ci

[client]
default-character-set=utf8
```

- 启动 MySQL 服务:

```
systemctl start/restart mysql
```

- 登录 MySQL:

```
mysql -u root -p 敲回车，输入密码
初始密码查看: cat /var/log/mysqld.log
在root@localhost: 后面的就是初始密码
```

- 查看默认字符集命令:

```
SHOW VARIABLES LIKE 'char%';
```

- 修改MySQL登录密码:

```
set global validate_password_policy=0;
set global validate_password_length=1;

set password=password('密码');
```

- 授予远程连接权限 (MySQL 内输入) :

```
-- 授权
grant all privileges on *.* to 'root' @'%' identified by '密码';
-- 刷新
flush privileges;
```

- 修改 MySQL 绑定 IP:

```
cd /etc/mysql/mysql.conf.d
sudo chmod 666 mysqld.cnf
vim mysqld.cnf
# bind-address = 127.0.0.1注释该行
```

- 关闭 Linux 防火墙

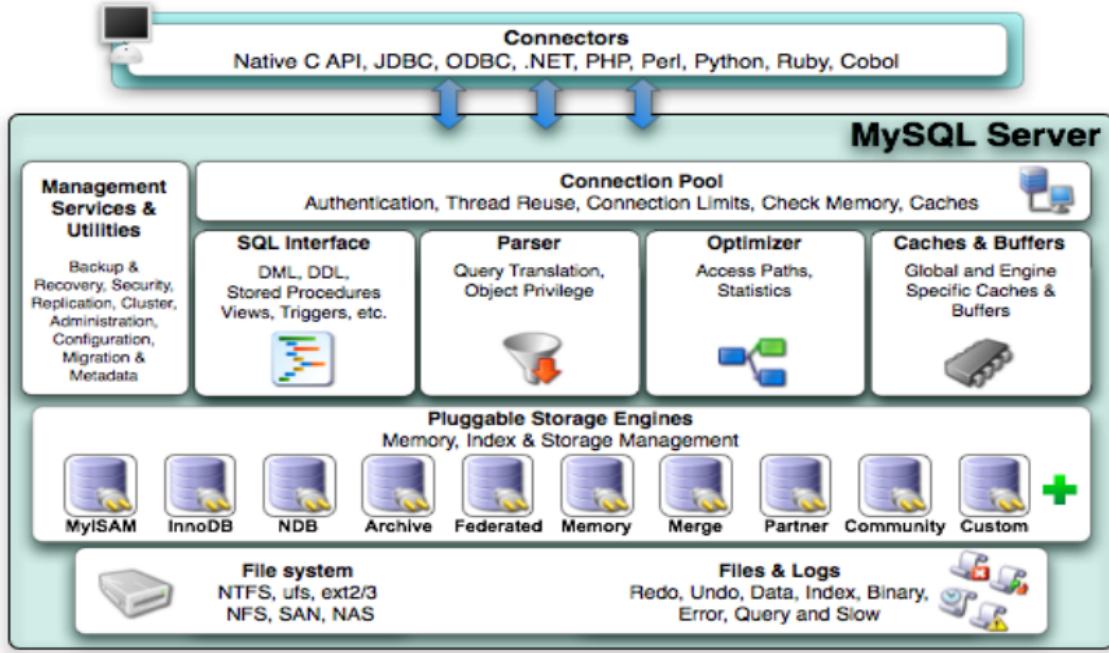
```
systemctl stop firewalld.service
# 放行3306端口
```

体系架构

整体架构

体系结构详解：

- 第一层：网络连接层
 - 一些客户端和链接服务，包含本地 Socket 通信和大多数基于客户端/服务端工具实现的 TCP/IP 通信，主要完成一些类似于连接处理、授权认证、及相关的安全方案
 - 在该层上引入了**连接池** Connection Pool 的概念，管理缓冲用户连接，线程处理等需要缓存的需求
 - 在该层上实现基于 SSL 的安全链接，服务器也会为安全接入的每个客户端验证它所具有的操作权限
- 第二层：核心服务层
 - 查询缓存、分析器、优化器、执行器等，涵盖 MySQL 的大多数核心服务功能，所有的内置函数（日期、数学、加密函数等）
 - Management Services & Utilities：系统管理和控制工具，备份、安全、复制、集群等
 - SQL Interface：接受用户的 SQL 命令，并且返回用户需要查询的结果
 - Parser：SQL 语句分析器
 - Optimizer：查询优化器
 - Caches & Buffers：查询缓存，服务器会查询内部的缓存，如果缓存空间足够大，可以在大量读操作的环境中提升系统性能
 - 所有**跨存储引擎的功能**在这一层实现，如存储过程、触发器、视图等
 - 在该层服务器会解析查询并创建相应的内部解析树，并对其进行相应的优化如确定表的查询顺序，是否利用索引等，最后生成相应的执行操作
 - MySQL 中服务器层不管理事务，**事务是由存储引擎实现的**
- 第三层：存储引擎层
 - Pluggable Storage Engines：存储引擎接口，MySQL 区别于其他数据库的重要特点就是其存储引擎的架构模式是插件式的（存储引擎是基于表的，而不是数据库）
 - 存储引擎**真正的负责了 MySQL 中数据的存储和提取**，服务器通过 API 和存储引擎进行通信
 - 不同的存储引擎具有不同的功能，共用一个 Server 层，可以根据开发的需要，来选取合适的存储引擎
- 第四层：系统文件层
 - 数据存储层，主要是将数据存储在文件系统之上，并完成与存储引擎的交互
 - File System：文件系统，保存配置文件、数据文件、日志文件、错误文件、二进制文件等



建立连接

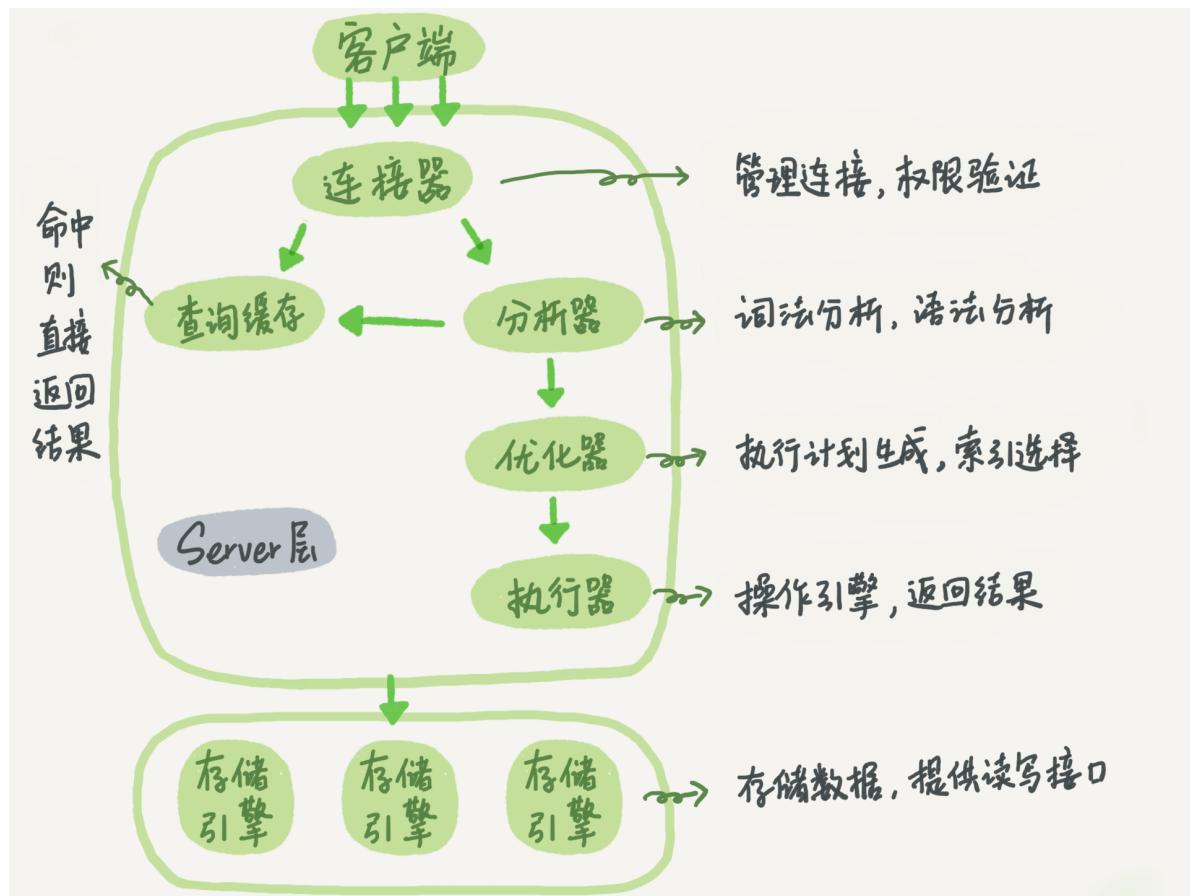
连接器

池化技术：对于访问数据库来说，建立连接的代价是比较昂贵的，因为每个连接对应一个用来交互的线程，频繁的创建关闭连接比较耗费资源，有必要建立数据库连接池，以提高访问的性能

连接建立 TCP 以后需要做**权限验证**，验证成功后可以进行执行 SQL。如果这时管理员账号对这个用户的权限做了修改，也不会影响已经存在连接的权限，只有再新建的连接才会使用新的权限设置

MySQL 服务器可以同时和多个客户端进行交互，所以要保证每个连接会话的隔离性（事务机制部分详解）

整体的执行流程：



权限信息

grant 语句会同时修改数据表和内存，判断权限的时候使用的是内存数据

flush privileges 语句本身会用数据表（磁盘）的数据重建一份内存权限数据，所以在权限数据可能存在不一致的情况下使用，这种不一致往往是由于直接用 DML 语句操作权限表导致的，所以尽量不要使用这类语句

| 权限 | 磁盘存储 | 内存存储 | 修改策略 | 作用范围 |
|------|----------------------|-----------------------------------|--------------------|------|
| 全局权限 | 表 mysql.user | 数组 acl_users | 已存在的连接不生效，新建连接立即生效 | 当前线程 |
| db权限 | 表 mysql.db | 数组 acl_dbs | 所有连接立即生效 | 全局 |
| 表权限 | 表 mysql.tables_priv | 和列权限组合的hash结构 column_priv_hash | 所有连接立即生效 | 全局 |
| 列权限 | 表 mysql.columns_priv | 和表权限组合的hash结构 column_priv_hash | 所有连接立即生效 | 全局 |

连接状态

客户端如果长时间没有操作，连接器就会自动断开，时间是由参数 `wait_timeout` 控制的，默认值是 8 小时。如果在连接被断开之后，客户端再次发送请求的话，就会收到一个错误提醒：`Lost connection to MySQL server during query`

数据库里面，长连接是指连接成功后，如果客户端持续有请求，则一直使用同一个连接；短连接则是指每次执行完很少的几次查询就断开连接，下次查询再重新建立一个

为了减少连接的创建，推荐使用长连接，但是过多的长连接会造成 OOM，解决方案：

- 定期断开长连接，使用一段时间，或者程序里面判断执行过一个占用内存的大查询后，断开连接，之后要查询再重连

KILL CONNECTION id

- MySQL 5.7 版本，可以在每次执行一个比较大的操作后，通过执行 `mysql_reset_connection` 来重新初始化连接资源，这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态

`SHOW PROCESSLIST`：查看当前 MySQL 在进行的线程，可以实时地查看 SQL 的执行情况，其中的 Command 列显示为 Sleep 的这一行，就表示现在系统里面有一个空闲连接

```
mysql> show processlist;
+----+-----+-----+-----+-----+-----+
| Id | User | Host      | db   | Command | Time | State    | Info          |
+----+-----+-----+-----+-----+-----+
| 35 | root | localhost | demo_02 | Query   | 0     | init     | show processlist |
| 38 | root | 192.168.192.1:53928 | demo_01 | Sleep   | 3278  |          | NULL           |
| 39 | root | 192.168.192.1:53929 | NULL   | Sleep   | 3287  |          | NULL           |
+----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

| 参数 | 含义 |
|---------|---|
| ID | 用户登录 mysql 时系统分配的 <code>connection_id</code> ，可以使用函数 <code>connection_id()</code> 查看 |
| User | 显示当前用户，如果不是 root，这个命令就只显示用户权限范围的 sql 语句 |
| Host | 显示这个语句是从哪个 ip 的哪个端口上发的，可以用来跟踪出现问题语句的用户 |
| db | 显示这个进程目前连接的是哪个数据库 |
| Command | 显示当前连接的执行的命令，一般取值为休眠 Sleep、查询 Query、连接 Connect 等 |
| Time | 显示这个状态持续的时间，单位是秒 |
| State | 显示使用当前连接的 sql 语句的状态，以查询为例，需要经过 <code>copying to tmp table</code> 、 <code>sorting result</code> 、 <code>sending data</code> 等状态才可以完成 |
| Info | 显示执行的 sql 语句，是判断问题语句的一个重要依据 |

Sending data 状态表示 MySQL 线程开始访问数据行并把结果返回给客户端，而不仅仅只是返回给客户端，是处于执行器过程中的任意阶段。由于在 Sending data 状态下，MySQL 线程需要做大量磁盘读取操作，所以是整个查询中耗时最长的状态

执行流程

查询缓存

工作流程

当执行完全相同的 SQL 语句的时候，服务器就会直接从缓存中读取结果，当数据被修改，之前的缓存会失效，修改比较频繁的表不适合做查询缓存

查询过程：

1. 客户端发送一条查询给服务器
2. 服务器先会检查查询缓存，如果命中了缓存，则立即返回存储在缓存中的结果（一般是 K-V 键值对），否则进入下一阶段
3. 分析器进行 SQL 分析，再由优化器生成对应的执行计划
4. 执行器根据优化器生成的执行计划，调用存储引擎的 API 来执行查询
5. 将结果返回给客户端

大多数情况下不建议使用查询缓存，因为查询缓存往往弊大于利

- 查询缓存的**失效非常频繁**，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能费力地把结果存起来，还没使用就被一个更新全清空了，对于更新压力大的数据库来说，查询缓存的命中率会非常低
- 除非业务就是有一张静态表，很长时间才会更新一次，比如一个系统配置表，那这张表上的查询才适合使用查询缓存

缓存配置

1. 查看当前 MySQL 数据库是否支持查询缓存：

```
SHOW VARIABLES LIKE 'have_query_cache'; -- YES
```

2. 查看当前 MySQL 是否开启了查询缓存：

```
SHOW VARIABLES LIKE 'query_cache_type'; -- OFF
```

参数说明：

- OFF 或 0：查询缓存功能关闭
- ON 或 1：查询缓存功能打开，查询结果符合缓存条件即会缓存，否则不予缓存；可以显式指定 SQL_NO_CACHE 不予缓存
- DEMAND 或 2：查询缓存功能按需进行，显式指定 SQL_CACHE 的 SELECT 语句才缓存，其它不予缓存

```
SELECT SQL_CACHE id, name FROM customer; -- SQL_CACHE: 查询结果可缓存  
SELECT SQL_NO_CACHE id, name FROM customer; -- SQL_NO_CACHE: 不使用查询缓存
```

3. 查看查询缓存的占用大小:

```
SHOW VARIABLES LIKE 'query_cache_size';-- 单位是字节 1048576 / 1024 = 1024 = 1KB
```

4. 查看查询缓存的状态变量:

```
SHOW STATUS LIKE 'Qcache%';
```

```
mysql> SHOW STATUS LIKE 'Qcache%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| Qcache_free_blocks | 1       |
| Qcache_free_memory | 1031832 |
| Qcache_hits        | 0       |
| Qcache_inserts     | 0       |
| Qcache_lowmem_prunes | 0       |
| Qcache_not_cached  | 6       |
| Qcache_queries_in_cache | 0       |
| Qcache_total_blocks | 1       |
+-----+-----+
8 rows in set (0.05 sec)
```

| 参数 | 含义 |
|-------------------------|--|
| Qcache_free_blocks | 查询缓存中的可用内存块数 |
| Qcache_free_memory | 查询缓存的可用内存量 |
| Qcache_hits | 查询缓存命中数 |
| Qcache_inserts | 添加到查询缓存的查询数 |
| Qcache_lowmem_prunes | 由于内存不足而从查询缓存中删除的查询数 |
| Qcache_not_cached | 非缓存查询的数量 (由于 query_cache_type 设置而无法缓存或未缓存) |
| Qcache_queries_in_cache | 查询缓存中注册的查询数 |
| Qcache_total_blocks | 查询缓存中的块总数 |

5. 配置 my.cnf:

```
sudo chmod 666 /etc/mysql/my.cnf
vim my.cnf
# mysql中配置缓存
query_cache_type=1
```

重启服务既可生效，执行 SQL 语句进行验证，执行一条比较耗时的 SQL 语句，然后多次执行几次，查看后面几次的执行时间；获取通过查看查询缓存的缓存命中数，来判定是否走查询缓存

缓存失效

查询缓存失效的情况：

- SQL 语句不一致，要想命中查询缓存，查询的 SQL 语句必须一致，因为缓存中 key 是查询的语句，value 是查询结构

```
select count(*) from tb_item;
select count(*) from tb_item; -- 不走缓存，首字母不一致
```

- 当查询语句中有一些不确定查询时，则不会缓存，比如：now()、current_date()、curdate()、curtime()、rand()、uuid()、user()、database()

```
SELECT * FROM tb_item WHERE updatetime < NOW() LIMIT 1;
SELECT USER();
SELECT DATABASE();
```

- 不使用任何表查询语句：

```
SELECT 'A';
```

- 查询 mysql、information_schema、performance_schema 等系统表时，不走查询缓存：

```
SELECT * FROM information_schema.engines;
```

- 在跨存储引擎的存储过程、触发器或存储函数的主体内执行的查询，缓存失效
- 如果表更改，则使用该表的所有高速缓存查询都将变为无效并从高速缓存中删除，包括使用 MERGE 映射到已更改表的表的查询，比如：INSERT、UPDATE、DELETE、ALTER TABLE、DROP TABLE、DROP DATABASE

分析器

没有命中查询缓存，就开始了 SQL 的真正执行，分析器会对 SQL 语句做解析

```
SELECT * FROM t WHERE id = 1;
```

解析器：处理语法和解析查询，生成一棵对应的解析树

- 先做词法分析，输入的是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串分别是什么代表什么。从输入的 select 这个关键字识别出来这是一个查询语句；把字符串 t 识别成表名 t，把字符串 id 识别成列 id
- 然后做语法分析，根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法。如果语句不对，就会收到 You have an error in your SQL syntax 的错误提醒

预处理器：进一步检查解析树的合法性，比如数据表和数据列是否存在、别名是否有歧义等

优化器

成本分析

优化器是在表里面都有多个索引的时候，决定使用哪个索引；或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序

- 根据搜索条件找出所有可能的使用的索引
- 成本分析，执行成本由 I/O 成本和 CPU 成本组成，计算全表扫描和使用不同索引执行 SQL 的代价
- 找到一个最优的执行方案，用最小的代价去执行语句

在数据库里面，扫描行数是影响执行代价的因素之一，扫描的行数越少意味着访问磁盘的次数越少，消耗的 CPU 资源越少，优化器还会结合是否使用临时表、是否排序等因素进行综合判断

统计数据

MySQL 中保存着两种统计数据：

- innodb_table_stats 存储了表的统计数据，每一条记录对应着一个表的统计数据
- innodb_index_stats 存储了索引的统计数据，每一条记录对应着一个索引的一个统计项的数据

MySQL 在真正执行语句之前，并不能精确地知道满足条件的记录有多少条，只能根据统计信息来估算记录，统计信息就是索引的区分度，一个索引上不同的值的个数（比如性别只能是男女，就是 2），称之为基数（cardinality），**基数越大说明区分度越好**

通过**采样统计**来获取基数，InnoDB 默认会选择 N 个数据页，统计这些页面上的不同值得到一个平均值，然后乘以这个索引的页面数，就得到了这个索引的基数

在 MySQL 中，有两种存储统计数据的方式，可以通过设置参数 `innodb_stats_persistent` 的值来选择：

- ON：表示统计信息会持久化存储（默认），采样页数 N 默认为 20，可以通过 `innodb_stats_persistent_sample_pages` 指定，页数越多统计的数据越准确，但消耗的资源更大
- OFF：表示统计信息只存储在内存，采样页数 N 默认为 8，也可以通过系统变量设置（不推荐，每次重新计算浪费资源）

数据表是会持续更新的，两种统计信息的更新方式：

- 设置 `innodb_stats_auto_recalc` 为 1，当发生变动的记录数量超过表大小的 10% 时，自动触发重新计算，不过是**异步进行**
- 调用 `ANALYZE TABLE t` 手动更新统计信息，只对信息做**重新统计**（不是重建表），没有修改数据，这个过程中加了 MDL 读锁并且是同步进行，所以会暂时阻塞系统

EXPLAIN 执行计划在优化器阶段生成，如果 `explain` 的结果预估的 `rows` 值跟实际情况差距比较大，可以执行 `analyze` 命令重新修正信息

错选索引

采样统计本身是估算数据，或者 SQL 语句中的字段选择有问题时，可能导致 MySQL 没有选择正确的执行索引

解决方法：

- 采用 force index 强行选择一个索引

```
SELECT * FROM user FORCE INDEX(name) WHERE NAME='seazean';
```

- 可以考虑修改 SQL 语句，引导 MySQL 使用期望的索引
- 新建一个更合适的索引，来提供给优化器做选择，或删掉误用的索引

执行器

开始执行的时候，要先判断一下当前连接对表有没有**执行查询的权限**，如果没有就会返回没有权限的错误，在工程实现上，如果命中查询缓存，会在查询缓存返回结果的时候，做权限验证。如果有权限，就打开表继续执行，执行器就会根据表的引擎定义，去使用这个引擎提供的接口

引擎层

Server 层和存储引擎层的交互是以记录为单位的，存储引擎会将单条记录返回给 Server 层做进一步处理，并不是直接返回所有的记录

工作流程：

- 首先根据二级索引选择扫描范围，获取第一条符合二级索引条件的记录，进行回表查询，将聚簇索引的记录返回 Server 层，由 Server 判断记录是否符合要求
- 然后在二级索引上继续扫描下一个符合条件的记录

推荐阅读：<https://mp.weixin.qq.com/s/YZ-LckObephrP1f15mzHpA>

终止流程

终止语句

终止线程中正在执行的语句：

```
KILL QUERY thread_id
```

KILL 不是马上终止的意思，而是告诉执行线程这条语句已经不需要继续执行，可以开始执行停止的逻辑（类似于打断）。因为对表做增删改查操作，会在表上加 MDL 读锁，如果线程被 KILL 时就直接终止，那这个 MDL 读锁就没机会被释放了

命令 `KILL QUERYthread_id_A` 的执行流程：

- 把 session A 的运行状态改成 THD::KILL_QUERY (将变量 killed 赋值为 THD::KILL_QUERY)
- 给 session A 的执行线程发一个信号，让 session A 来处理这个 THD::KILL_QUERY 状态

会话处于等待状态（锁阻塞），必须满足是一个可以被唤醒的等待，必须有机会去判断线程的状态，如果不满足就会造成 KILL 失败

典型场景：innodb_thread_concurrency 为 2，代表并发线程上限数设置为 2

- session A 执行事务，session B 执行事务，达到线程上限；此时 session C 执行事务会阻塞等待，session D 执行 kill query C 无效
- C 的逻辑是每 10 毫秒判断是否可以进入 InnoDB 执行，如果不行就调用 nanosleep 函数进入 sleep 状态，没有去判断线程状态

补充：执行 Ctrl+C 的时候，是 MySQL 客户端另外启动一个连接，然后发送一个 KILL QUERY 命令

终止连接

断开线程的连接：

```
KILL CONNECTION id
```

断开连接后执行 SHOW PROCESSLIST 命令，如果这条语句的 Command 列显示 Killed，代表线程的状态是 KILL_CONNECTION，说明这个线程有语句正在执行，当前状态是停止语句执行中，终止逻辑耗时较长

- 超大事务执行期间被 KILL，这时回滚操作需要对事务执行期间生成的所有新数据版本做回收操作，耗时很长
- 大查询回滚，如果查询过程中生成了比较大的临时文件，删除临时文件可能需要等待 IO 资源，导致耗时较长
- DDL 命令执行到最后阶段被 KILL，需要删除中间过程的临时文件，也可能受 IO 资源影响耗时较久

总结：KILL CONNECTION 本质上只是把客户端的 SQL 连接断开，后面的终止流程还是要走 KILL QUERY

一个事务被 KILL 之后，持续处于回滚状态，不应该强行重启整个 MySQL 进程，应该等待事务自己执行完成，因为重启后依然继续做回滚操作的逻辑

常用工具

mysql

mysql 不是指 mysql 服务，而是指 mysql 的客户端工具

```
mysql [options] [database]
```

- -u --user=name: 指定用户名
- -p --password[=name]: 指定密码
- -h --host=name: 指定服务器IP或域名
- -P --port=#: 指定连接端口
- -e --execute=name: 执行SQL语句并退出，在控制台执行SQL语句，而不用连接到数据库执行

示例：

```
mysql -h 127.0.0.1 -P 3306 -u root -p  
mysql -uroot -p2143 db01 -e "select * from tb_book";
```

admin

mysqladmin 是一个执行管理操作的客户端程序，用来检查服务器的配置和当前状态、创建并删除数据库等

通过 `mysqladmin --help` 指令查看帮助文档

```
mysqladmin -uroot -p2143 create 'test01';
```

binlog

服务器生成的日志文件以二进制格式保存，如果需要检查这些文本，就要使用 mysqlbinlog 日志管理工具

```
mysqlbinlog [options] log-files1 log-files2 ...
```

- -d --database=name: 指定数据库名称，只列出指定的数据库相关操作
- -o --offset=#: 忽略掉日志中的前 n 行命令。
- -r --result-file=name: 将输出的文本格式日志输出到指定文件。
- -s --short-form: 显示简单格式，省略掉一些信息。
- --start-datetime=date1 --stop-datetime=date2: 指定日期间隔内的所有日志

- --start-position=pos1 --stop-position=pos2：指定位置间隔内的所有日志
-

dump

命令介绍

mysqldump 客户端工具用来备份数据库或在不同数据库之间进行数据迁移，备份内容包含创建表，及插入表的 SQL 语句

```
mysqldump [options] db_name [tables]
mysqldump [options] --database/-B db1 [db2 db3...]
mysqldump [options] --all-databases/-A
```

连接选项：

- -u --user=name：指定用户名
- -p --password[=name]：指定密码
- -h --host=name：指定服务器 IP 或域名
- -P --port=#：指定连接端口

输出内容选项：

- --add-drop-database：在每个数据库创建语句前加上 Drop database 语句
- --add-drop-table：在每个表创建语句前加上 Drop table 语句，默认开启，不开启(--skip-add-drop-table)
- -n --no-create-db：不包含数据库的创建语句
- -t --no-create-info：不包含数据表的创建语句
- -d --no-data：不包含数据
- -T, --tab=name：自动生成两个文件：一个 .sql 文件，创建表结构的语句；一个 .txt 文件，数据文件，相当于 select into outfile

示例：

```
mysqldump -uroot -p2143 db01 tb_book --add-drop-database --add-drop-table > a
mysqldump -uroot -p2143 -T /tmp test city
```

数据备份

命令行方式：

- 备份命令：mysqldump -u root -p 数据库名称 > 文件保存路径
- 恢复
 1. 登录MySQL数据库：mysql -u root p
 2. 删除已经备份的数据库
 3. 重新创建与备份数据库名称相同的数据库

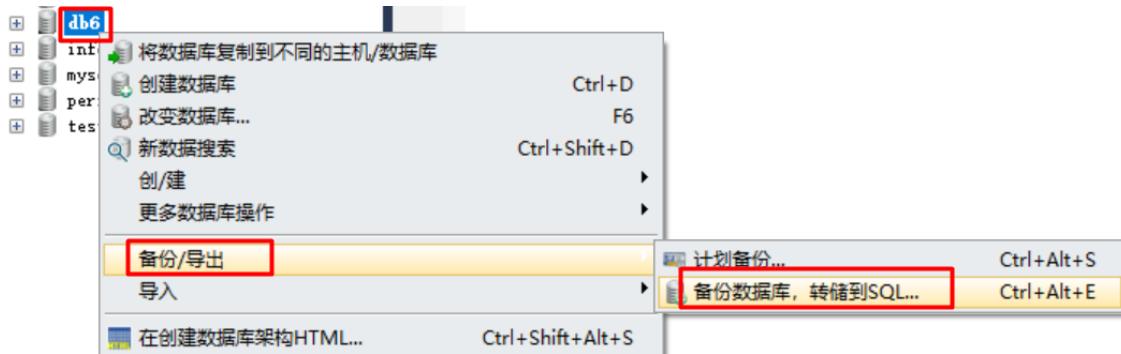
4. 使用该数据库

5. 导入文件执行: `source 备份文件全路径`

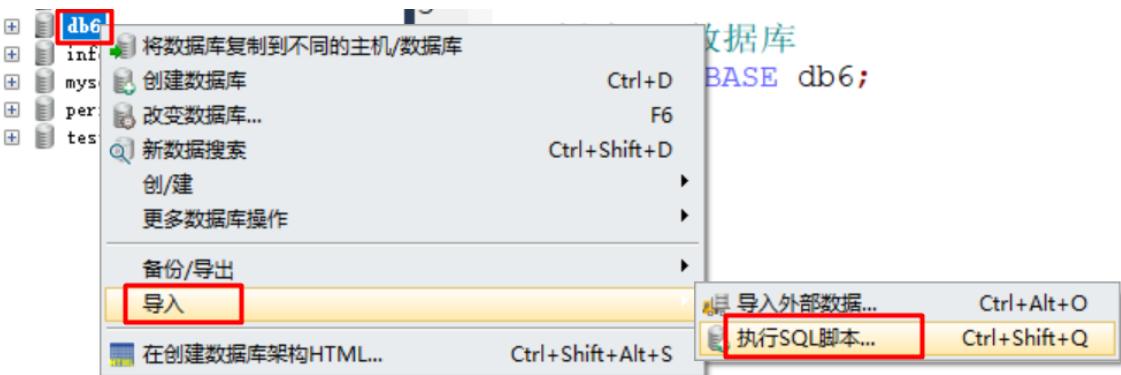
更多方式参考: <https://time.geekbang.org/column/article/81925>

图形化界面:

- 备份



- 恢复



import

`mysqlimport` 是客户端数据导入工具，用来导入`mysqldump`加`-T`参数后导出的文本文件

```
mysqlimport [options] db_name textfile1 [textfile2...]
```

示例:

```
mysqlimport -uroot -p2143 test /tmp/city.txt
```

导入 sql 文件，可以使用 MySQL 中的 `source` 指令：

```
source 文件全路径
```

show

mysqlshow 客户端对象查找工具，用来很快地查找存在哪些数据库、数据库中的表、表中的列或者索引

```
mysqlshow [options] [db_name [table_name [col_name]]]
```

- --count: 显示数据库及表的统计信息（数据库，表 均可以不指定）
- -i: 显示指定数据库或者指定表的状态信息

示例：

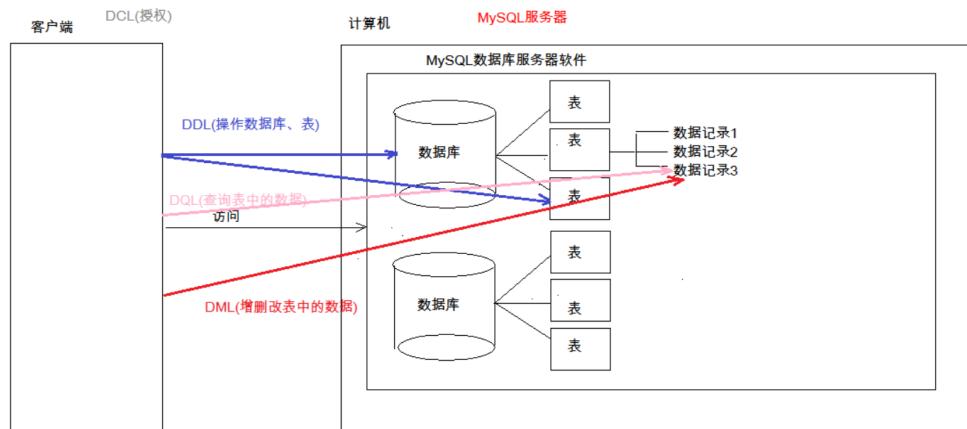
```
#查询每个数据库的表的数量及表中记录的数量
mysqlshow -uroot -p1234 --count
#查询test库中每个表中的字段数，及行数
mysqlshow -uroot -p1234 test --count
#查询test库中book表的详细情况
mysqlshow -uroot -p1234 test book --count
```

单表操作

SQL

- SQL
 - Structured Query Language：结构化查询语言
 - 定义了操作所有关系型数据库的规则，每种数据库操作的方式可能会存在不一样的地方，称为“方言”
- SQL 通用语法
 - SQL 语句可以单行或多行书写，以**分号结尾**。
 - 可使用空格和缩进来增强语句的可读性。
 - MySQL 数据库的 SQL 语句不区分大小写，**关键字建议使用大写**。
 - 数据库的注释：
 - 单行注释：-- 注释内容 #注释内容 (MySQL 特有)
 - 多行注释：/* 注释内容 */
- SQL 分类
 - DDL (Data Definition Language) 数据定义语言
 - 用来定义数据库对象：数据库，表，列等。关键字：create、drop,、alter 等
 - DML (Data Manipulation Language) 数据操作语言
 - 用来对数据库中表的数据进行增删改。关键字：insert、delete、update 等
 - DQL (Data Query Language) 数据查询语言

- 用来查询数据库中表的记录(数据)。关键字: select、where 等
- DCL (Data Control Language) 数据控制语言
 - 用来定义数据库的访问权限和安全级别, 及创建用户。关键字: grant, revoke等



DDL

数据库

- R(Retrieve): 查询
 - 查询所有数据库:

```
SHOW DATABASES;
```

- 查询某个数据库的创建语句

```
SHOW CREATE DATABASE 数据库名称; -- 标准语法
```

```
SHOW CREATE DATABASE mysql; -- 查看mysql数据库的创建格式
```

- C(Create): 创建
 - 创建数据库

```
CREATE DATABASE 数据库名称; -- 标准语法
```

```
CREATE DATABASE db1; -- 创建db1数据库
```

- 创建数据库 (判断, 如果不存在则创建)

```
CREATE DATABASE IF NOT EXISTS 数据库名称;
```

- 创建数据库, 并指定字符集

```
CREATE DATABASE 数据库名称 CHARACTER SET 字符集名称;
```

- 例如：创建db4数据库、如果不存在则创建，指定字符集为gbk

```
-- 创建db4数据库、如果不存在则创建，指定字符集为gbk  
CREATE DATABASE IF NOT EXISTS db4 CHARACTER SET gbk;  
  
-- 查看db4数据库的字符集  
SHOW CREATE DATABASE db4;
```

- U(Update): 修改

- 修改数据库的字符集

```
ALTER DATABASE 数据库名称 CHARACTER SET 字符集名称;
```

- 常用字符集：

```
-- 查询所有支持的字符集  
SHOW CHARSET;  
-- 查看所有支持的校对规则  
SHOW COLLATION;  
  
-- 字符集：utf8,latinI,GBK,,GBK是utf8的子集  
-- 校对规则：ci 大小定不敏感，cs或bin大小写敏感
```

- D>Delete): 删除

- 删除数据库：

```
DROP DATABASE 数据库名称;
```

- 删除数据库(判断，如果存在则删除)：

```
DROP DATABASE IF EXISTS 数据库名称;
```

- 使用数据库：

- 查询当前正在使用的数据库名称

```
SELECT DATABASE();
```

- 使用数据库

```
USE 数据库名称; -- 标准语法  
USE db4; -- 使用db4数据库
```

数据表

- R(Retrieve): 查询
 - 查询数据库中所有的数据表

```
USE mysql;-- 使用mysql数据库
```

```
SHOW TABLES;-- 查询库中所有的表
```

- 查询表结构

```
DESC 表名;
```

- 查询表字符集

```
SHOW TABLE STATUS FROM 库名 LIKE '表名';
```

- C(Create): 创建
 - 创建数据表

```
CREATE TABLE 表名(  
    列名1 数据类型1,  
    列名2 数据类型2,  
    ....  
    列名n 数据类型n  
)  
-- 注意: 最后一列, 不需要加逗号
```

- 复制表

```
CREATE TABLE 表名 LIKE 被复制的表名; -- 标准语法
```

```
CREATE TABLE product2 LIKE product; -- 复制product表到product2表
```

- 数据类型

| 数据类型 | 说明 |
|-----------|---|
| INT | 整数类型 |
| DOUBLE | 小数类型 |
| DATE | 日期, 只包含年月日: yyyy-MM-dd |
| DATETIME | 日期, 包含年月日时分秒: yyyy-MM-dd HH:mm:ss |
| TIMESTAMP | 时间戳类型, 包含年月日时分秒: yyyy-MM-dd HH:mm:ss 如果不给这个字段赋值或赋值为 NULL, 则默认使用当前的系统时间 |
| CHAR | 字符串, 定长类型 |
| VARCHAR | 字符串, 变长类型 name varchar(20) 代表姓名最大 20 个字符: zhangsan 8 个字符, 张三 2 个字符 |

`INT(n)` : n 代表位数

- 3: int (9) 显示结果为 000000010
- 3: int (3) 显示结果为 010

`varchar(n)` : n 表示的是字符数

- 例如:

```
-- 使用db3数据库
USE db3;

-- 创建一个product商品表
CREATE TABLE product(
    id INT,                      -- 商品编号
    NAME VARCHAR(30),            -- 商品名称
    price DOUBLE,                -- 商品价格
    stock INT,                   -- 商品库存
    insert_time DATE             -- 上架时间
);
```

- U(Update): 修改

- 修改表名

```
ALTER TABLE 表名 RENAME TO 新的表名;
```

- 修改表的字符集

```
ALTER TABLE 表名 CHARACTER SET 字符集名称;
```

- 添加一列

```
ALTER TABLE 表名 ADD 列名 数据类型;
```

- 修改列数据类型

```
ALTER TABLE 表名 MODIFY 列名 新数据类型;
```

- 修改列名称和数据类型

```
ALTER TABLE 表名 CHANGE 列名 新列名 新数据类型;
```

- 删除列

```
ALTER TABLE 表名 DROP 列名;
```

- D(Delete): 删除

- 删除数据表

```
DROP TABLE 表名;
```

- 删除数据表(判断, 如果存在则删除)

```
DROP TABLE IF EXISTS 表名;
```

DML

INSERT

- 新增表数据

- 新增格式 1: 给指定列添加数据

```
INSERT INTO 表名(列名1,列名2...) VALUES (值1,值2...);
```

- 新增格式 2: 默认给全部列添加数据

```
INSERT INTO 表名 VALUES (值1,值2,值3,...);
```

- 新增格式 3: 批量添加数据

```
-- 给指定列批量添加数据
```

```
INSERT INTO 表名(列名1,列名2,...) VALUES (值1,值2,...),(值1,值2,...)....;
```

```
-- 默认给所有列批量添加数据
```

```
INSERT INTO 表名 VALUES (值1,值2,值3,...),(值1,值2,值3,...)....;
```

- 字符串拼接

```
CONCAT(string1,string2,'',...)
```

- 注意事项
 - 列名和值的数量以及数据类型要对应
 - 除了数字类型，其他数据类型的数据都需要加引号(单引双引都可以，推荐单引)

UPDATE

- 修改表数据语法
 - 标准语法

```
UPDATE 表名 SET 列名1 = 值1,列名2 = 值2,... [WHERE 条件];
```

- 修改电视的价格为1800、库存为36

```
UPDATE product SET price=1800,stock=36 WHERE NAME='电视';  
SELECT * FROM product;-- 查看所有商品信息
```

- 注意事项
 - 修改语句中必须加条件
 - 如果不加条件，则将所有数据都修改

DELETE

- 删除表数据语法

```
DELETE FROM 表名 [WHERE 条件];
```

- 注意事项
 - 删除语句中必须加条件
 - 如果不加条件，则将所有数据删除

DQL

查询语法

数据库查询遵循条件在前的原则

```
SELECT DISTINCT
    <select list>
FROM
    <left_table> <join_type>
JOIN
    <right_table> ON <join_condition>      -- 连接查询在多表查询部分详解
WHERE
    <where_condition>
GROUP BY
    <group_by_list>
HAVING
    <having_condition>
ORDER BY
    <order_by_condition>
LIMIT
    <limit_params>
```

执行顺序：

```
FROM      <left_table>

ON        <join_condition>

<join_type>      JOIN      <right_table>

WHERE      <where_condition>

GROUP BY    <group_by_list>

HAVING      <having_condition>

SELECT DISTINCT    <select list>

ORDER BY    <order_by_condition>

LIMIT      <limit_params>
```

查询全部

- 查询全部的表数据

```
-- 标准语法  
SELECT * FROM 表名;  
  
-- 查询product表所有数据(常用)  
SELECT * FROM product;
```

- **查询指定字段的表数据**

```
SELECT 列名1,列名2,... FROM 表名;
```

- **去除重复查询**: 只有值全部重复的才可以去除, 需要创建临时表辅助查询

```
SELECT DISTINCT 列名1,列名2,... FROM 表名;
```

- **计算列的值 (四则运算)**

```
SELECT 列名1 运算符(+ - * /) 列名2 FROM 表名;  
  
/*如果某一列值为null, 可以进行替换  
  ifnull(表达式1,表达式2)  
    表达式1: 想替换的列  
    表达式2: 想替换的值*/
```

例如:

```
-- 查询商品名称和库存, 库存数量在原有基础上加10  
SELECT NAME,stock+10 FROM product;  
  
-- 查询商品名称和库存, 库存数量在原有基础上加10。进行null值判断  
SELECT NAME,IFNULL(stock,0)+10 FROM product;
```

- **起别名**

```
SELECT 列名1,列名2,... AS 别名 FROM 表名;
```

例如:

```
-- 查询商品名称和库存, 库存数量在原有基础上加10。进行null值判断, 起别名为getSum,AS可以省略。  
SELECT NAME,IFNULL(stock,0)+10 AS getsum FROM product;  
SELECT NAME,IFNULL(stock,0)+10 getsum FROM product;
```

条件查询

- 条件查询语法

```
SELECT 列名 FROM 表名 WHERE 条件;
```

- 条件分类

| 符号 | 功能 |
|---------------------|--|
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |
| = | 等于 |
| <> 或 != | 不等于 |
| BETWEEN ... AND ... | 在某个范围之内(都包含) |
| IN(...) | 多选一 |
| LIKE | 模糊查询：_单个任意字符、%任意个字符、[]匹配集合内的字符 LIKE '[^AB]%'：不以 A 和 B 开头的任意文本 |
| IS NULL | 是NULL |
| IS NOT NULL | 不是NULL |
| AND 或 && | 并且 |
| OR 或 | 或者 |
| NOT 或 ! | 非，不是 |
| UNION | 对两个结果集进行并集操作并进行去重，同时进行默认规则的排序 |
| UNION ALL | 对两个结果集进行并集操作不进行去重，不进行排序 |

- 例如：

```
-- 查询库存大于20的商品信息
```

```
SELECT * FROM product WHERE stock > 20;
```

```
-- 查询品牌为华为的商品信息
```

```
SELECT * FROM product WHERE brand='华为';
```

```
-- 查询金额在4000 ~ 6000之间的商品信息
```

```
SELECT * FROM product WHERE price >= 4000 AND price <= 6000;
```

```
SELECT * FROM product WHERE price BETWEEN 4000 AND 6000;
```

```
-- 查询库存为14、30、23的商品信息
```

```
SELECT * FROM product WHERE stock=14 OR stock=30 OR stock=23;
```

```

SELECT * FROM product WHERE stock IN(14,30,23);

-- 查询库存为null的商品信息
SELECT * FROM product WHERE stock IS NULL;

-- 查询库存不为null的商品信息
SELECT * FROM product WHERE stock IS NOT NULL;

-- 查询名称以'小米'开头的商品信息
SELECT * FROM product WHERE NAME LIKE '小米%';

-- 查询名称第二个字是'为'的商品信息
SELECT * FROM product WHERE NAME LIKE '_为%';

-- 查询名称为四个字符的商品信息 4个下划线
SELECT * FROM product WHERE NAME LIKE '____';

-- 查询名称中包含电脑的商品信息
SELECT * FROM product WHERE NAME LIKE '%电脑%';

```

```

mysql> SELECT * FROM product;
+----+-----+-----+-----+-----+
| id | NAME      | price | brand | stock | insert_time |
+----+-----+-----+-----+-----+
| 1  | 华为手机    | 3999 | 华为   | 23    | 2088-03-10 |
| 2  | 小米手机    | 2999 | 小米   | 30    | 2088-05-15 |
| 3  | 苹果手机    | 5999 | 苹果   | 18    | 2088-08-20 |
| 4  | 华为电脑    | 6999 | 华为   | 14    | 2088-06-16 |
| 5  | 小米电脑    | 4999 | 小米   | 26    | 2088-07-08 |
| 6  | 苹果电脑    | 8999 | 苹果   | 15    | 2088-10-25 |
| 7  | 联想电脑    | 7999 | 联想   | NULL  | 2088-11-11 |
+----+-----+-----+-----+-----+
7 rows in set (0.00 sec)

mysql> 

```

函数查询

聚合函数

聚合函数：将一列数据作为一个整体，进行纵向的计算

- 聚合函数语法

```

SELECT 函数名(列名) FROM 表名 [WHERE 条件]

```

- 聚合函数分类

| 函数名 | 功能 |
|-----------|-----------------------|
| COUNT(列名) | 统计数量 (一般选用不为 null 的列) |
| MAX(列名) | 最大值 |
| MIN(列名) | 最小值 |
| SUM(列名) | 求和 |
| AVG(列名) | 平均值 (会忽略 null 行) |

- 例如

```
-- 计算product表中总记录条数 7
SELECT COUNT(*) FROM product;

-- 获取最高价格
SELECT MAX(price) FROM product;
-- 获取最高价格的商品名称
SELECT NAME,price FROM product WHERE price = (SELECT MAX(price) FROM
product);

-- 获取最低库存
SELECT MIN(stock) FROM product;
-- 获取最低库存的商品名称
SELECT NAME,stock FROM product WHERE stock = (SELECT MIN(stock) FROM
product);

-- 获取总库存数量
SELECT SUM(stock) FROM product;
-- 获取品牌为小米的平均商品价格
SELECT AVG(price) FROM product WHERE brand='小米';
```

文本函数

CONCAT(): 用于连接两个字段

```
SELECT CONCAT(TRIM(col1), '(', TRIM(col2), ')') AS concat_col FROM mytable
-- 许多数据库会使用空格把一个值填充为列宽, 连接的结果出现一些不必要的空格, 使用TRIM()可以去除首尾空格
```

| 函数名称 | 作用 |
|-----------|------------------------------------|
| 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个 |
| INSERT | 替换字符串函数 |
| LOWER | 将字符串中的字母转换为小写 |
| UPPER | 将字符串中的字母转换为大写 |
| LEFT | 从左侧字截取字符串，返回字符串左边的若干个字符 |
| RIGHT | 从右侧字截取字符串，返回字符串右边的若干个字符 |
| TRIM | 删除字符串左右两侧的空格 |
| REPLACE | 字符串替换函数，返回替换后的新字符串 |
| SUBSTRING | 截取字符串，返回从指定位置开始的指定长度的字符串 |
| 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 |

数字函数

| 函数名称 | 作用 |
|----------------|--------------------------------|
| ABS | 求绝对值 |
| SQRT | 求二次方根 |
| MOD | 求余数 |
| CEIL 和 CEILING | 两个函数功能相同，都是返回不小于参数的最小整数，即向上取整 |
| FLOOR | 向下取整，返回值转化为一个BIGINT |
| 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 |
| POW 和 POWER | 两个函数的功能相同，都是所传参数的次方的结果值 |
| SIN | 求正弦值 |
| ASIN | 求反正弦值，与函数 SIN 互为反函数 |
| COS | 求余弦值 |
| ACOS | 求反余弦值，与函数 COS 互为反函数 |
| TAN | 求正切值 |
| ATAN | 求反正切值，与函数 TAN 互为反函数 |
| COT | 求余切值 |

日期函数

| 函数名称 | 作用 |
|------------------------|--------------------------------------|
| CURDATE 和 CURRENT_DATE | 两个函数作用相同，返回当前系统的日期值 |
| CURTIME 和 CURRENT_TIME | 两个函数作用相同，返回当前系统的时间值 |
| NOW 和 SYSDATE | 两个函数作用相同，返回当前系统的日期和时间值 |
| 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 |
| DAYNAME | 获取指定日期对应的星期几的英文名称 |
| DAYOFWEEK | 获取指定日期对应的一周的索引位置值 |
| 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 |
| DAYOFMONTH | 获取指定日期是一个月中是第几天，返回值范围是1~31 |
| YEAR | 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 |
| SEC_TO_TIME | 将秒数转换为时间，与TIME_TO_SEC 互为反函数 |
| DATE_ADD 和 ADDDATE | 两个函数功能相同，都是向日期添加指定的时间间隔 |
| DATE_SUB 和 SUBDATE | 两个函数功能相同，都是向日期减去指定的时间间隔 |
| ADDTIME | 时间加法运算，在原始时间上添加指定的时间 |
| SUBTIME | 时间减法运算，在原始时间上减去指定的时间 |
| DATEDIFF | 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 |

正则查询

正则表达式 (Regular Expression) 是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串

```

SELECT * FROM emp WHERE name REGEXP '^T';      -- 匹配以T开头的name值
SELECT * FROM emp WHERE name REGEXP '2$';      -- 匹配以2结尾的name值
SELECT * FROM emp WHERE name REGEXP '[uvw]';-- 匹配包含uvw的name值

```

| 符号 | 含义 |
|--------|------------------|
| ^ | 在字符串开始处进行匹配 |
| \$ | 在字符串末尾处进行匹配 |
| . | 匹配任意单个字符,包括换行符 |
| [...] | 匹配出括号内的任意字符 |
| [^...] | 匹配不出括号内的任意字符 |
| a* | 匹配零个或者多个a(包括空串) |
| a+ | 匹配一个或者多个a(不包括空串) |
| a? | 匹配零个或者一个a |
| a1 a2 | 匹配a1或a2 |
| a(m) | 匹配m个a |
| a(m,) | 至少匹配m个a |
| a(m,n) | 匹配m个a到n个a |
| a(,n) | 匹配0到n个a |
| (...) | 将模式元素组成单一元素 |

排序查询

- 排序查询语法

```

SELECT 列名 FROM 表名 [WHERE 条件] ORDER BY 列名1 排序方式1,列名2 排序方式2;

```

- 排序方式

ASC:升序
DESC:降序

注意：多个排序条件，当前级的条件值一样时，才会判断第二条件

- 例如

```
-- 按照库存升序排序
SELECT * FROM product ORDER BY stock ASC;

-- 查询名称中包含手机的商品信息。按照金额降序排序
SELECT * FROM product WHERE NAME LIKE '%手机%' ORDER BY price DESC;

-- 按照金额升序排序，如果金额相同，按照库存降序排列
SELECT * FROM product ORDER BY price ASC,stock DESC;
```

分组查询

分组查询会进行去重

- 分组查询语法

```
SELECT 列名 FROM 表名 [WHERE 条件] GROUP BY 分组列名 [HAVING 分组后条件过滤]
[ORDER BY 排序列名 排序方式];
```

WHERE 过滤行， HAVING 过滤分组，行过滤应当先于分组过滤

分组规定：

- GROUP BY 子句出现在 WHERE 子句之后， ORDER BY 子句之前
- NULL 的行会单独分为一组
- 大多数 SQL 实现不支持 GROUP BY 列具有可变长度的数据类型

- 例如

```
-- 按照品牌分组，获取每组商品的总金额
SELECT brand,SUM(price) FROM product GROUP BY brand;

-- 对金额大于4000元的商品，按照品牌分组，获取每组商品的总金额
SELECT brand,SUM(price) FROM product WHERE price > 4000 GROUP BY brand;

-- 对金额大于4000元的商品，按照品牌分组，获取每组商品的总金额，只显示总金额大于7000元的
SELECT brand,SUM(price) AS getSum FROM product WHERE price > 4000 GROUP BY
brand HAVING getSum > 7000;

-- 对金额大于4000元的商品，按照品牌分组，获取每组商品的总金额，只显示总金额大于7000元的、
并按照总金额的降序排列
SELECT brand,SUM(price) AS getSum FROM product WHERE price > 4000 GROUP BY
brand HAVING getSum > 7000 ORDER BY getSum DESC;
```

分页查询

- 分页查询语法

```
SELECT 列名 FROM 表名 [WHERE 条件] GROUP BY 分组列名 [HAVING 分组后条件过滤]  
[ORDER BY 排序列名 排序方式] LIMIT 开始索引, 查询条数;
```

- 公式: 开始索引 = (当前页码-1) * 每页显示的条数

- 例如

```
SELECT * FROM product LIMIT 0,2; -- 第一页 开始索引=(1-1) * 2  
SELECT * FROM product LIMIT 2,2; -- 第二页 开始索引=(2-1) * 2  
SELECT * FROM product LIMIT 4,2; -- 第三页 开始索引=(3-1) * 2  
SELECT * FROM product LIMIT 6,2; -- 第四页 开始索引=(4-1) * 2
```

| ID | Name | Price | Brand | Stock | Insert Time |
|----|------|-------|-------|--------|--------------|
| 1 | 华为手机 | 3999 | 华为 | 23 | 2088-03-10 0 |
| 2 | 小米手机 | 2999 | 小米 | 30 | 2088-05-15 1 |
| 3 | 苹果手机 | 5999 | 苹果 | 18 | 2088-08-20 2 |
| 4 | 华为电脑 | 6999 | 华为 | 14 | 2088-06-16 3 |
| 5 | 小米电脑 | 4999 | 小米 | 26 | 2088-07-08 4 |
| 6 | 苹果电脑 | 8999 | 苹果 | 15 | 2088-10-25 5 |
| 7 | 联想电脑 | 7999 | 联想 | (NULL) | 2088-11-11 6 |

```
SELECT * FROM product LIMIT 0,2; -- 第一页 开始索引=(1-1) * 2
```

```
1 华为手机 | 3999 华为 | 23 2088-03-10 0
```

```
2 小米手机 | 2999 小米 | 30 2088-05-15 1
```

```
SELECT * FROM product LIMIT 2,2; -- 第二页 开始索引=(2-1) * 2
```

```
3 苹果手机 | 5999 苹果 | 18 2088-08-20 2
```

```
4 华为电脑 | 6999 华为 | 14 2088-06-16 3
```

```
SELECT * FROM product LIMIT 4,2; -- 第三页 开始索引=(3-1) * 2
```

```
5 小米电脑 | 4999 小米 | 26 2088-07-08 4
```

```
6 苹果电脑 | 8999 苹果 | 15 2088-10-25 5
```

```
SELECT * FROM product LIMIT 6,2; -- 第四页 开始索引=(4-1) * 2
```

```
7 联想电脑 | 7999 联想 | (NULL) 2088-11-11 6
```

多表操作

约束分类

约束介绍

约束：对表中的数据进行限定，保证数据的正确性、有效性、完整性

约束的分类：

| 约束 | 说明 |
|-------------------------------|---------|
| PRIMARY KEY | 主键约束 |
| PRIMARY KEY AUTO_INCREMENT | 主键、自动增长 |
| UNIQUE | 唯一约束 |
| NOT NULL | 非空约束 |
| FOREIGN KEY | 外键约束 |
| FOREIGN KEY ON UPDATE CASCADE | 外键级联更新 |
| FOREIGN KEY ON DELETE CASCADE | 外键级联删除 |

主键约束

- 主键约束特点：
 - 主键约束默认包含**非空**和**唯一**两个功能
 - 一张表只能有一个主键
 - 主键一般用于表中数据的唯一标识
- 建表时添加主键约束

```
CREATE TABLE 表名(
    列名 数据类型 PRIMARY KEY,
    列名 数据类型,
    ...
);
```

- 删除主键约束

```
ALTER TABLE 表名 DROP PRIMARY KEY;
```

- 建表后单独添加主键约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型 PRIMARY KEY;
```

- 例如

```
-- 创建student表
CREATE TABLE student(
    id INT PRIMARY KEY -- 给id添加主键约束
);

-- 添加数据
INSERT INTO student VALUES (1),(2);
-- 主键默认唯一，添加重复数据，会报错
INSERT INTO student VALUES (2);
-- 主键默认非空，不能添加null的数据
INSERT INTO student VALUES (NULL);
```

主键自增

主键自增约束可以为空，并自动增长。删除某条数据不影响自增的下一个数值，依然按照前一个值自增

- 建表时添加主键自增约束

```
CREATE TABLE 表名 (
    列名 数据类型 PRIMARY KEY AUTO_INCREMENT,
    列名 数据类型,
    ...
);
```

- 删除主键自增约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型;
```

- 建表后单独添加主键自增约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型 AUTO_INCREMENT;
```

- 例如

```
-- 创建student2表
CREATE TABLE student2(
    id INT PRIMARY KEY AUTO_INCREMENT -- 给id添加主键自增约束
);

-- 添加数据
INSERT INTO student2 VALUES (1),(2);
-- 添加null值，会自动增长
INSERT INTO student2 VALUES (NULL),(NULL);-- 3, 4
```

唯一约束

唯一约束：约束不能有重复的数据

- 建表时添加唯一约束

```
CREATE TABLE 表名(
    列名 数据类型 UNIQUE,
    列名 数据类型,
    ...
);
```

- 删除唯一约束

```
ALTER TABLE 表名 DROP INDEX 列名;
```

- 建表后单独添加唯一约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型 UNIQUE;
```

非空约束

- 建表时添加非空约束

```
CREATE TABLE 表名(
    列名 数据类型 NOT NULL,
    列名 数据类型,
    ...
);
```

- 删除非空约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型;
```

- 建表后单独添加非空约束

```
ALTER TABLE 表名 MODIFY 列名 数据类型 NOT NULL;
```

外键约束

外键约束：让表和表之间产生关系，从而保证数据的准确性

- 建表时添加外键约束

```
CREATE TABLE 表名(
    列名 数据类型 约束,
    ...
    CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表名(主表主键列名)
);
```

- 删除外键约束

```
ALTER TABLE 表名 DROP FOREIGN KEY 外键名;
```

- 建表后单独添加外键约束

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表名(主表主键列名);
```

- 例如

```
-- 创建user用户表
CREATE TABLE USER(
    id INT PRIMARY KEY AUTO_INCREMENT,          -- id
    name VARCHAR(20) NOT NULL                  -- 姓名
);
-- 添加用户数据
INSERT INTO USER VALUES (NULL,'张三'),(NULL,'李四'),(NULL,'王五');

-- 创建orderlist订单表
CREATE TABLE orderlist(
    id INT PRIMARY KEY AUTO_INCREMENT,          -- id
    number VARCHAR(20) NOT NULL,                -- 订单编号
    uid INT,                                    -- 订单所属用户
    CONSTRAINT ou_fk1 FOREIGN KEY (uid) REFERENCES USER(id)   -- 添加外键约束
);
-- 添加订单数据
INSERT INTO orderlist VALUES (NULL,'hm001',1),(NULL,'hm002',1),
(NULL,'hm003',2),(NULL,'hm004',2),
(NULL,'hm005',3),(NULL,'hm006',3);

-- 添加一个订单，但是没有所属用户。无法添加
INSERT INTO orderlist VALUES (NULL,'hm007',8);
-- 删除王五这个用户，但是订单表中王五还有很多个订单呢。无法删除
DELETE FROM USER WHERE NAME='王五';
```

外键级联

级联操作：当把主表中的数据进行删除或更新时，从表中有关联的数据的相应操作，包括 RESTRICT、CASCADE、SET NULL 和 NO ACTION

- RESTRICT 和 NO ACTION 相同，是指限制在子表有关联记录的情况下，父表不能更新
- CASCADE 表示父表在更新或者删除时，更新或者删除子表对应的记录
- SET NULL 则表示父表在更新或者删除的时候，子表的对应字段被 SET NULL

级联操作：

- 添加级联更新

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表  
名(主表主键列名) ON UPDATE [CASCADE | RESTRICT | SET NULL];
```

- 添加级联删除

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表  
名(主表主键列名) ON DELETE CASCADE;
```

- 同时添加级联更新和级联删除

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表  
名(主表主键列名) ON UPDATE CASCADE ON DELETE CASCADE;
```

多表设计

一对一

多表：有多张数据表，而表与表之间有一定的关联关系，通过外键约束实现，分为一对一、一对多、多对多三类

举例：人和身份证件

实现原则：在任意一个表建立外键，去关联另外一个表的主键

```
-- 创建person表  
CREATE TABLE person(  
    id INT PRIMARY KEY AUTO_INCREMENT, -- 主键id  
    NAME VARCHAR(20) -- 姓名  
);  
-- 添加数据  
INSERT INTO person VALUES (NULL, '张三'), (NULL, '李四');  
  
-- 创建card表
```

```

CREATE TABLE card(
    id INT PRIMARY KEY AUTO_INCREMENT, -- 主键id
    number VARCHAR(20) UNIQUE NOT NULL, -- 身份证号
    pid INT UNIQUE, -- 外键列
    CONSTRAINT cp_fk1 FOREIGN KEY (pid) REFERENCES person(id)
);
-- 添加数据
INSERT INTO card VALUES (NULL, '12345', 1), (NULL, '56789', 2);

```

一对一

| card | | | person | |
|------|--------|-----|--------|------|
| id | number | pid | id | name |
| 1 | 12345 | 1 | 1 | 张三 |
| 2 | 56789 | 2 | 2 | 李四 |

实现原则：

在任意一个表建立外键，去关联另外一个表的主键

一对多

举例：用户和订单、商品分类和商品

实现原则：在多的一方，建立外键约束，来关联一的一方主键

```

-- 创建user表
CREATE TABLE USER(
    id INT PRIMARY KEY AUTO_INCREMENT, -- 主键id
    NAME VARCHAR(20) -- 姓名
);
-- 添加数据
INSERT INTO USER VALUES (NULL, '张三'), (NULL, '李四');

-- 创建orderlist表
CREATE TABLE orderlist(
    id INT PRIMARY KEY AUTO_INCREMENT, -- 主键id
    number VARCHAR(20), -- 订单编号
    uid INT, -- 外键列
    CONSTRAINT ou_fk1 FOREIGN KEY (uid) REFERENCES USER(id)
);
-- 添加数据
INSERT INTO orderlist VALUES (NULL, 'hm001', 1), (NULL, 'hm002', 1), (NULL, 'hm003', 2),
(NULL, 'hm004', 2);

```

一对多

orderlist

| id | number | uid |
|----|--------|-----|
| 1 | hm001 | 1 |
| 2 | hm002 | 1 |
| 3 | hm003 | 2 |
| 4 | hm004 | 2 |

user

| id | name |
|----|------|
| 1 | 张三 |
| 2 | 李四 |

实现原则：

在多的一方，建立外键约束，来关联一方主键

多对多

举例：学生和课程。一个学生可以选择多个课程，一个课程也可以被多个学生选择

实现原则：借助第三张表中间表，中间表至少包含两个列，这两个列作为中间表的外键，分别关联两张表的主键

```
-- 创建student表
CREATE TABLE student(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 主键id
    NAME VARCHAR(20)                      -- 学生姓名
);
-- 添加数据
INSERT INTO student VALUES (NULL, '张三'), (NULL, '李四');

-- 创建course表
CREATE TABLE course(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 主键id
    NAME VARCHAR(10)                       -- 课程名称
);
-- 添加数据
INSERT INTO course VALUES (NULL, '语文'), (NULL, '数学');

-- 创建中间表
CREATE TABLE stu_course(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 主键id
    sid INT,                                -- 用于和student表中的id进行外键关联
    cid INT,                                -- 用于和course表中的id进行外键关联
    CONSTRAINT sc_fk1 FOREIGN KEY (sid) REFERENCES student(id), -- 添加外键约束
    CONSTRAINT sc_fk2 FOREIGN KEY (cid) REFERENCES course(id)   -- 添加外键约束
);
-- 添加数据
INSERT INTO stu_course VALUES (NULL, 1, 1), (NULL, 1, 2), (NULL, 2, 1), (NULL, 2, 2);
```

多对多

student

| id | name |
|----|------|
| 1 | 张三 |
| 2 | 李四 |

course

| id | name |
|----|------|
| 1 | 语文 |
| 2 | 数学 |

stu_course

| id | sid | cid |
|----|-----|-----|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 1 |
| 4 | 2 | 2 |

实现原则：

需要借助第三张表中间表，中间表至少包含两个列，这两个列作为中间表的外键，分别关联两张表的主键

连接查询

内外连接

内连接

连接查询的是两张表有交集的部分数据，两张表分为**驱动表**和**被驱动表**，如果结果集中的每条记录都是两个表相互匹配的组合，则称这样的结果集为笛卡尔积

内连接查询，若驱动表中的记录在被驱动表中找不到匹配的记录时，则该记录不会加到最后的结果集

- 显式内连接：

```
SELECT 列名 FROM 表名1 [INNER] JOIN 表名2 ON 条件;
```

- 隐式内连接：内连接中 WHERE 子句和 ON 子句是等价的

```
SELECT 列名 FROM 表名1,表名2 WHERE 条件;
```

STRAIGHT_JOIN与JOIN类似，只不过左表始终在右表之前读取，只适用于内连接

外连接

外连接查询，若驱动表中的记录在被驱动表中找不到匹配的记录时，则该记录也会加到最后的结果集，只是对于被驱动表中**不匹配过滤条件**的记录，各个字段使用NULL填充

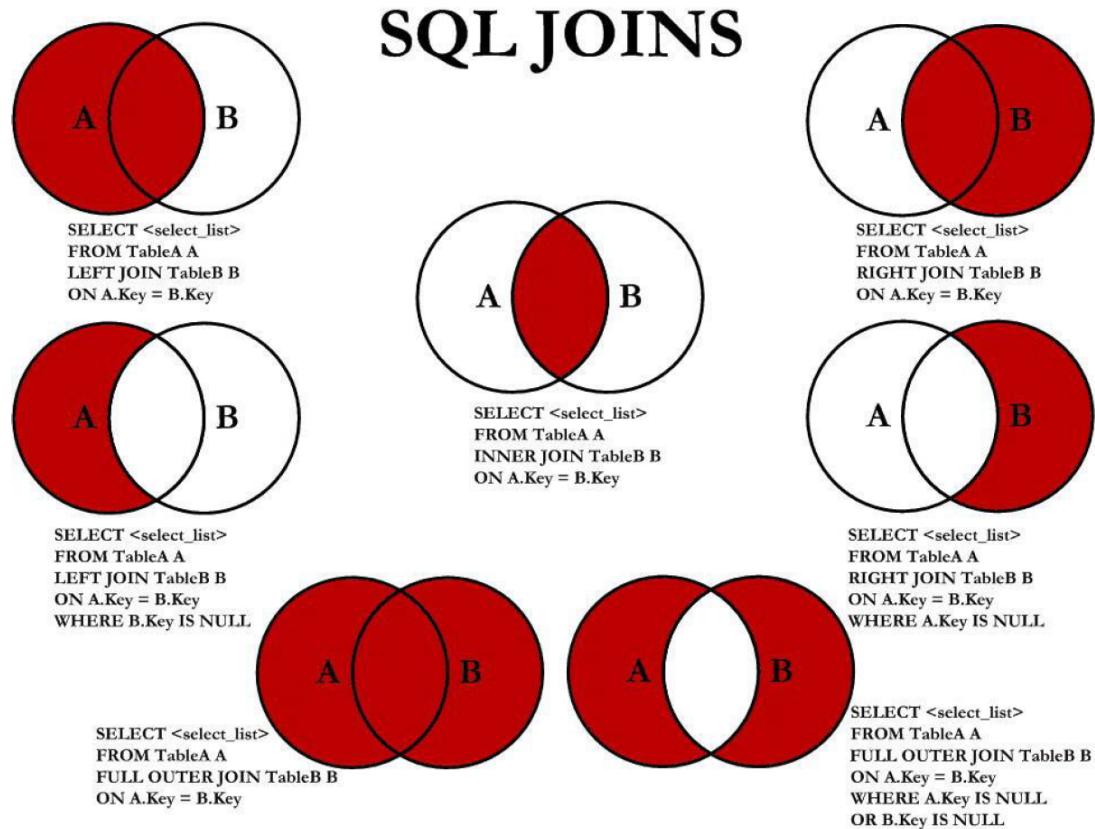
应用实例：查学生成绩，也想展示出缺考的人的成绩

- 左外连接：选择左侧的表为驱动表，查询左表的全部数据，和左右两张表有交集部分的数据

```
SELECT 列名 FROM 表名1 LEFT [OUTER] JOIN 表名2 ON 条件;
```

- 右外连接：选择右侧的表为驱动表，查询右表的全部数据，和左右两张表有交集部分的数据

```
SELECT 列名 FROM 表名1 RIGHT [OUTER] JOIN 表名2 ON 条件;
```



关联查询

自关联查询：同一张表中有数据关联，可以多次查询这同一个表

- 数据准备

```
-- 创建员工表
CREATE TABLE employee(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 员工编号
    NAME VARCHAR(20),                      -- 员工姓名
    mgr INT,                                -- 上级编号
    salary DOUBLE                          -- 员工工资
);
-- 添加数据
INSERT INTO employee VALUES (1001, '孙悟空', 1005, 9000.00), ..., (1009, '宋江', NULL, 16000.00);
```

| | id | NAME | mgr | salary |
|---|--------|--------|--------|--------|
| | 1001 | 孙悟空 | 1005 | 9000 |
| | 1002 | 猪八戒 | 1005 | 8000 |
| | 1003 | 沙和尚 | 1005 | 8500 |
| | 1004 | 小白龙 | 1005 | 7900 |
| | 1005 | 唐僧 | (NULL) | 15000 |
| | 1006 | 武松 | 1009 | 7600 |
| | 1007 | 李逵 | 1009 | 7400 |
| | 1008 | 林冲 | 1009 | 8100 |
| | 1009 | 宋江 | (NULL) | 16000 |
| * | (Auto) | (NULL) | (NULL) | (NULL) |

- 数据查询

```
-- 查询所有员工的姓名及其直接上级的姓名，没有上级的员工也需要查询
```

```
/*
```

```
分析
```

员工信息 employee表

条件: employee.mgr = employee.id

查询左表的全部数据，和左右两张表有交集部分数据，左外连接

```
*/
```

```
SELECT
```

```
    e1.id,
```

```
    e1.name,
```

```
    e1.mgr,
```

```
    e2.id,
```

```
    e2.name
```

```
FROM
```

```
    employee e1
```

```
LEFT OUTER JOIN
```

```
    employee e2
```

```
ON
```

```
    e1.mgr = e2.id;
```

- 查询结果

| id | name | mgr | id | name |
|------|------|------|------|------|
| 1001 | 孙悟空 | 1005 | 1005 | 唐僧 |
| 1002 | 猪八戒 | 1005 | 1005 | 唐僧 |
| 1003 | 沙和尚 | 1005 | 1005 | 唐僧 |
| 1004 | 小白龙 | 1005 | 1005 | 唐僧 |
| 1005 | 唐僧 | NULL | NULL | NULL |
| 1006 | 武松 | 1009 | 1009 | 宋江 |
| 1007 | 李逵 | 1009 | 1009 | 宋江 |
| 1008 | 林冲 | 1009 | 1009 | 宋江 |
| 1009 | 宋江 | NULL | NULL | NULL |

连接原理

Index Nested-Loop Join 算法：查询驱动表得到**数据集**，然后根据数据集中的每一条记录的**关联字段**再**分别到被驱动表中查找匹配（走索引）**，所以驱动表只需要访问一次，被驱动表要访问多次

MySQL 将查询驱动表后得到的记录成为驱动表的扇出，连接查询的成本：单次访问驱动表的成本 + 扇出值 * 单次访问被驱动表的成本，优化器会选择成本最小的表连接顺序（确定谁是驱动表，谁是被驱动表）生成执行计划，进行连接查询，优化方式：

- 减少驱动表的扇出（让数据量小的表来做驱动表）
- 降低访问被驱动表的成本

说明：STRAIGHT_JOIN 是查一条驱动表，然后根据关联字段去查被驱动表，要访问多次驱动表，所以需要优化为 INL 算法

Block Nested-Loop Join 算法：一种**空间换时间**的优化方式，基于块的循环连接，执行连接查询前申请一块固定大小的内存作为连接缓冲区 Join Buffer，先把若干条驱动表中的扇出暂存在缓冲区，每一条被驱动表中的记录一次性的与 Buffer 中多条记录进行匹配（扫描全部数据，一条一条的匹配），因为是在内存中完成，所以速度快，并且降低了 I/O 成本

Join Buffer 可以通过参数 `join_buffer_size` 进行配置，默认大小是 256 KB

在成本分析时，对于很多张表的连接查询，连接顺序有非常多，MySQL 如果挨着进行遍历计算成本，会消耗很多资源

- 提前结束某种连接顺序的成本评估：维护一个全局变量记录当前成本最小的连接方式，如果一种顺序只计算了一部分就已经超过了最小成本，可以提前结束计算
- 系统变量 `optimizer_search_depth`：如果连接表的个数小于该变量，就继续穷举分析每一种连接数量，反之只对数量与 depth 值相同的表进行分析，该值越大成本分析的越精确
- 系统变量 `optimizer_prune_level`：控制启发式规则的启用，这些规则就是根据以往经验指定的，不满足规则的连接顺序不分析成本

连接优化

BKA

Batched Key Access 算法是对 NLJ 算法的优化，在读取被驱动表的记录时使用顺序 IO，Extra 信息中会有 Batched Key Access 信息

使用 BKA 的表的 JOIN 过程如下：

- 连接驱动表将满足条件的记录放入 Join Buffer，并将两表连接的字段放入一个 DYNAMIC_ARRAY ranges 中
- 在进行表的过接过程中，会将 ranges 相关的信息传入 Buffer 中，进行被驱动表主键的查找及排序操作
- 调用步骤 2 中产生的有序主键，**顺序读取被驱动表的数据**
- 当缓冲区的数据被读完后，会重复进行步骤 2、3，直到记录被读取完

使用 BKA 优化需要设进行设置：

```
SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

说明：前两个参数的作用是启用 MRR，因为 BKA 算法的优化要依赖于 MRR（系统优化 → 内存优化 → Read 详解）

BNL

问题

BNL 即 Block Nested-Loop Join 算法，由于要访问多次被驱动表，会产生两个问题：

- Join 语句多次扫描一个冷表，并且语句执行时间小于 1 秒，就会在再次扫描冷表时，把冷表的数据页移到 LRU 链表头部，导致热数据被淘汰，影响业务的正常运行
这种情况冷表的数据量要小于整个 Buffer Pool 的 old 区域，能够完全放入 old 区，才会再次被读时加到 young，否则读取下一段时就已经把上一段淘汰
- Join 语句在循环读磁盘和淘汰内存页，进入 old 区域的数据页很可能在 1 秒之内就被淘汰，就会导致 MySQL 实例的 Buffer Pool 在这段时间内 young 区域的数据页没有被合理地淘汰

大表 Join 操作虽然对 IO 有影响，但是在语句执行结束后对 IO 的影响随之结束。但是对 Buffer Pool 的影响就是持续性的，需要依靠后续的查询请求慢慢恢复内存命中率

优化

将 BNL 算法转成 BKA 算法，优化方向：

- 在被驱动表上建索引，这样就可以根据索引进行顺序 IO
- 使用临时表，在临时表上建立索引，将被驱动表和临时表进行连接查询

驱动表 t1，被驱动表 t2，使用临时表的工作流程：

- 把表 t1 中满足条件的数据放在临时表 tmp_t 中
- 给临时表 tmp_t 的关联字段加上索引，使用 BKA 算法
- 让表 t2 和 tmp_t 做 Join 操作（临时表是被驱动表）

补充：MySQL 8.0 支持 hash join，join_buffer 维护的不再是一个无序数组，而是一个哈希表，查询效率更高，执行效率比临时表更高

嵌套查询

查询分类

查询语句中嵌套了查询语句，将嵌套查询称为子查询，FROM 子句后面的子查询的结果集称为派生表

根据结果分类：

- 结果是单行单列：可以将查询的结果作为另一条语句的查询条件，使用运算符判断

```
SELECT 列名 FROM 表名 WHERE 列名=(SELECT 列名/聚合函数(列名) FROM 表名 [WHERE 条件]);
```

- 结果是多行单列：可以作为条件，使用运算符 IN 或 NOT IN 进行判断

```
SELECT 列名 FROM 表名 WHERE 列名 [NOT] IN (SELECT 列名 FROM 表名 [WHERE 条件]);
```

- 结果是多行多列：查询的结果可以作为一张虚拟表参与查询

```
SELECT 列名 FROM 表名 [别名],(SELECT 列名 FROM 表名 [WHERE 条件]) [别名] [WHERE 条件];
```

```
-- 查询订单表orderlist中id大于4的订单信息和所属用户USER信息
SELECT
    *
FROM
    USER u,
    (SELECT * FROM orderlist WHERE id>4) o
WHERE
    u.id=o.uid;
```

相关性分类：

- 不相关子查询：子查询不依赖外层查询的值，可以单独运行出结果
- 相关子查询：子查询的执行需要依赖外层查询的值

查询优化

不相关子查询的结果集会被写入一个临时表，并且在写入时去重，该过程称为**物化**，存储结果集的临时表称为**物化表**

系统变量 tmp_table_size 或者 max_heap_table_size 为表的最值

- 小于系统变量时，内存中可以保存，会为建立**基于内存**的 MEMORY 存储引擎的临时表，并建立哈希索引
- 大于任意一个系统变量时，物化表会使用**基于磁盘**的 InnoDB 存储引擎来保存结果集中的记录，索引类型为 B+ 树

物化后，嵌套查询就相当于外层查询的表和物化表进行内连接查询，然后经过优化器选择成本最小的表连接顺序执行查询

子查询物化会产生建立临时表的成本，但是将子查询转化为连接查询可以充分发挥优化器的作用，所以引入：半连接

- t1 和 t2 表进行半连接，对于 t1 表中的某条记录，只需要关心在 t2 表中是否存在，而不需要关心有多少条记录与之匹配，最终结果集只保留 t1 的记录
- 半连接只是执行子查询的一种方式，MySQL 并没有提供面向用户的半连接语法

参考书籍：<https://book.douban.com/subject/35231266/>

联合查询

UNION 是取这两个子查询结果的并集，并进行去重，同时进行默认规则的排序（union 是行加起来，join 是列加起来）

UNION ALL 是对两个结果集进行并集操作不进行去重，不进行排序

```
(select 1000 as f) union (select id from t1 order by id desc limit 2); #t1表中包含  
id 为 1-1000 的数据
```

语句的执行流程：

- 创建一个内存临时表，这个临时表只有一个整型字段 f，并且 f 是主键字段
- 执行第一个子查询，得到 1000 这个值，并存入临时表中
- 执行第二个子查询，拿到第一行 id=1000，试图插入临时表中，但由于 1000 这个值已经存在于临时表了，违反了唯一性约束，所以插入失败，然后继续执行
- 取到第二行 id=999，插入临时表成功
- 从临时表中按行取出数据，返回结果并删除临时表，结果中包含两行数据分别是 1000 和 999

查询练习

数据准备：

```
-- 创建db4数据库  
CREATE DATABASE db4;  
-- 使用db4数据库  
USE db4;  
  
-- 创建user表  
CREATE TABLE USER(  
    id INT PRIMARY KEY AUTO_INCREMENT, -- 用户id  
    NAME VARCHAR(20), -- 用户姓名  
    age INT -- 用户年龄  
);  
  
-- 订单表
```

```

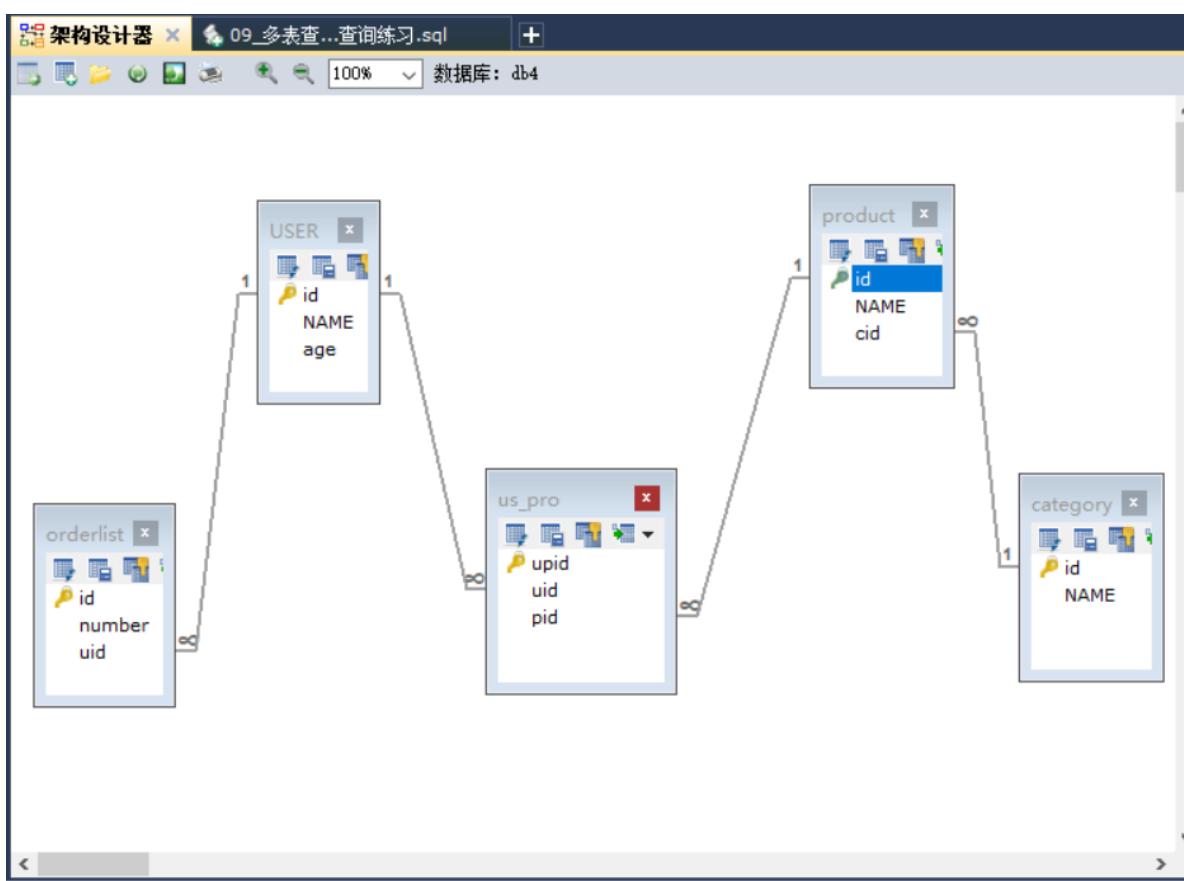
CREATE TABLE orderlist(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 订单id
    number VARCHAR(30),                   -- 订单编号
    uid INT,                            -- 外键字段
    CONSTRAINT ou_fk1 FOREIGN KEY (uid) REFERENCES USER(id)
);

-- 商品分类表
CREATE TABLE category(
    id INT PRIMARY KEY AUTO_INCREMENT,    -- 商品分类id
    NAME VARCHAR(10)                     -- 商品分类名称
);

-- 商品表
CREATE TABLE product(
    id INT PRIMARY KEY AUTO_INCREMENT,    -- 商品id
    NAME VARCHAR(30),                   -- 商品名称
    cid INT,                            -- 外键字段
    CONSTRAINT cp_fk1 FOREIGN KEY (cid) REFERENCES category(id)
);

-- 中间表
CREATE TABLE us_pro(
    upid INT PRIMARY KEY AUTO_INCREMENT,   -- 中间表id
    uid INT,                            -- 外键字段。需要和用户表的主键产生关联
    pid INT,                            -- 外键字段。需要和商品表的主键产生关联
    CONSTRAINT up_fk1 FOREIGN KEY (uid) REFERENCES USER(id),
    CONSTRAINT up_fk2 FOREIGN KEY (pid) REFERENCES product(id)
);

```



数据查询：

1. 查询用户的编号、姓名、年龄、订单编号

数据：用户的编号、姓名、年龄在 user 表，订单编号在 orderlist 表

条件：user.id = orderlist.uid

```
SELECT
    u.*,
    o.number
FROM
    USER u,
    orderlist o
WHERE
    u.id = o.uid;
```

2. 查询所有的用户，显示用户的编号、姓名、年龄、订单编号。

```
SELECT
    u.*,
    o.number
FROM
    USER u
LEFT OUTER JOIN
    orderlist o
ON
    u.id = o.uid;
```

3. 查询用户年龄大于 23 岁的信息，显示用户的编号、姓名、年龄、订单编号

```
SELECT
    u.*,
    o.number
FROM
    USER u,
    orderlist o
WHERE
    u.id = o.uid
    AND
    u.age > 23;
```

```
SELECT
    u.*,
    o.number
FROM
    (SELECT * FROM USER WHERE age > 23) u,-- 嵌套查询
    orderlist o
WHERE
    u.id = o.uid;
```

4. 查询张三和李四用户的信息，显示用户的编号、姓名、年龄、订单编号。

```
SELECT
    u.*,
    o.number
FROM
    USER u,
    orderlist o
WHERE
    u.id=o.uid
    AND
    u.name IN ('张三', '李四');
```

5. 查询所有的用户和该用户能查看的所有的商品，显示用户的编号、姓名、年龄、商品名称

数据：用户的编号、姓名、年龄在 user 表，商品名称在 product 表，中间表 us_pro

条件：us_pro.uid = user.id AND us_pro.pid = product.id

```
SELECT
    u.id,
    u.name,
    u.age,
    p.name
FROM
    USER u,
    product p,
    us_pro up
WHERE
    up.uid = u.id
    AND
    up.pid=p.id;
```

6. 查询张三和李四这两个用户可以看到的商品，显示用户的编号、姓名、年龄、商品名称。

```
SELECT
    u.id,
    u.name,
    u.age,
    p.name
FROM
    USER u,
    product p,
    us_pro up
WHERE
    up.uid=u.id
    AND
    up.pid=p.id
    AND
    u.name IN ('张三', '李四');
```

高级结构

视图

基本介绍

视图概念：视图是一种虚拟存在的数据表，这个虚拟的表并不在数据库中实际存在

本质：将一条 SELECT 查询语句的结果封装到了一个虚拟表中，所以在创建视图的时候，工作重心要放在这条 SELECT 查询语句上

作用：将一些比较复杂的查询语句的结果，封装到一个虚拟表中，再有相同查询需求时，直接查询该虚拟表

优点：

- 简单：使用视图的用户不需要关心表的结构、关联条件和筛选条件，因为虚拟表中已经是过滤好的结果集
- 安全：使用视图的用户只能访问查询的结果集，对表的权限管理并不能限制到某个行某个列
- 数据独立，一旦视图的结构确定，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响；源表修改列名，则可以通过修改视图来解决，不会造成对访问者的影响

视图创建

- 创建视图

```
CREATE [OR REPLACE]
VIEW 视图名称 [(列名列表)]
AS 查询语句
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

`WITH [CASCADED | LOCAL] CHECK OPTION` 决定了是否允许更新数据使记录不再满足视图的条件：

- LOCAL：只要满足本视图的条件就可以更新
 - CASCADED：必须满足所有针对该视图的所有视图的条件才可以更新， 默认值
- 例如

```
-- 数据准备 city
id  NAME      cid
1   深圳      1
2   上海      1
3   纽约      2
4   莫斯科    3

-- 数据准备 country
id  NAME
```

```
1 中国
2 美国
3 俄罗斯

-- 创建city_country视图，保存城市和国家的信息(使用指定列名)
CREATE
VIEW
    city_country (city_id,city_name,country_name)
AS
SELECT
    c1.id,
    c1.name,
    c2.name
FROM
    city c1,
    country c2
WHERE
    c1.cid=c2.id;
```

视图查询

- 查询所有数据表，视图也会查询出来

```
SHOW TABLES;
SHOW TABLE STATUS [\G];
```

- 查询视图

```
SELECT * FROM 视图名称;
```

- 查询某个视图创建

```
SHOW CREATE VIEW 视图名称;
```

视图修改

视图表数据修改，会**自动修改源表中的数据**，因为更新的是视图中的基表中的数据

- 修改视图表中的数据

```
UPDATE 视图名称 SET 列名 = 值 WHERE 条件;
```

- 修改视图的结构

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW 视图名称 [(列名列表)]
AS 查询语句
[WITH [CASCADED | LOCAL] CHECK OPTION]

-- 将视图中的country_name修改为name
ALTER
VIEW
    city_country (city_id,city_name,name)
AS
SELECT
    c1.id,
    c1.name,
    c2.name
FROM
    city c1,
    country c2
WHERE
    c1.cid=c2.id;
```

视图删除

- 删除视图

```
DROP VIEW 视图名称;
```

- 如果存在则删除

```
DROP VIEW IF EXISTS 视图名称;
```

存储过程

基本介绍

存储过程和函数：存储过程和函数是事先经过编译并存储在数据库中的一段 SQL 语句的集合

存储过程和函数的好处：

- 提高代码的复用性
- 减少数据在数据库和应用服务器之间的传输，提高传输效率
- 减少代码层面的业务处理
- 一次编译永久有效

存储过程和函数的区别：

- 存储函数必须有返回值
 - 存储过程可以没有返回值
-

基本操作

DELIMITER:

- DELIMITER 关键字用来声明 sql 语句的分隔符，告诉 MySQL 该段命令已经结束
- MySQL 语句默认的分隔符是分号，但是有时需要一条功能 sql 语句中包含分号，但是并不作为结束标识，这时使用 DELIMITER 来指定分隔符：

DELIMITER 分隔符

存储过程的创建调用查看和删除：

- 创建存储过程

```
-- 修改分隔符为$  
DELIMITER $  
  
-- 标准语法  
CREATE PROCEDURE 存储过程名称(参数...)  
BEGIN  
    sql语句;  
END$  
  
-- 修改分隔符为分号  
DELIMITER ;
```

- 调用存储过程

CALL 存储过程名称(实际参数);

- 查看存储过程

SELECT * FROM mysql.proc WHERE db='数据库名称';

- 删除存储过程

DROP PROCEDURE [IF EXISTS] 存储过程名称;

练习：

- 数据准备

| id | NAME | age | gender | score |
|----|------|-----|--------|-------|
| 1 | 张三 | 23 | 男 | 95 |
| 2 | 李四 | 24 | 男 | 98 |
| 3 | 王五 | 25 | 女 | 100 |
| 4 | 赵六 | 26 | 女 | 90 |

- 创建 stu_group() 存储过程，封装分组查询总成绩，并按照总成绩升序排序的功能

```

DELIMITER $

CREATE PROCEDURE stu_group()
BEGIN
    SELECT gender,SUM(score) getSum FROM student GROUP BY gender ORDER BY
getSum ASC;
END$

DELIMITER ;

-- 调用存储过程
CALL stu_group();
-- 删除存储过程
DROP PROCEDURE IF EXISTS stu_group;

```

存储语法

变量使用

存储过程是可以进行编程的，意味着可以使用变量、表达式、条件控制语句等，来完成比较复杂的功能

- 定义变量：DECLARE 定义的是局部变量，只能用在 BEGIN END 范围之内

```
DECLARE 变量名 数据类型 [DEFAULT 默认值];
```

- 变量的赋值

```

SET 变量名 = 变量值;
SELECT 列名 INTO 变量名 FROM 表名 [WHERE 条件];

```

- 数据准备：表 student

| id | NAME | age | gender | score |
|----|------|-----|--------|-------|
| 1 | 张三 | 23 | 男 | 95 |
| 2 | 李四 | 24 | 男 | 98 |
| 3 | 王五 | 25 | 女 | 100 |
| 4 | 赵六 | 26 | 女 | 90 |

- 定义两个 int 变量，用于存储男女同学的总分数

```

DELIMITER $
CREATE PROCEDURE pro_test3()
BEGIN
    -- 定义两个变量
    DECLARE men,women INT;
    -- 查询男同学的总分数, 为men赋值
    SELECT SUM(score) INTO men FROM student WHERE gender='男';
    -- 查询女同学的总分数, 为women赋值
    SELECT SUM(score) INTO women FROM student WHERE gender='女';
    -- 使用变量
    SELECT men,women;
END$
DELIMITER ;
-- 调用存储过程
CALL pro_test3();

```

IF语句

- if 语句标准语法

```

IF 判断条件1 THEN 执行的sql语句1;
[ELSEIF 判断条件2 THEN 执行的sql语句2;]
...
[ELSE 执行的sql语句n;]
END IF;

```

- 数据准备：表 student

| | id | NAME | age | gender | score |
|---|----|------|-----|--------|-------|
| 1 | 张三 | 23 | 男 | 95 | |
| 2 | 李四 | 24 | 男 | 98 | |
| 3 | 王五 | 25 | 女 | 100 | |
| 4 | 赵六 | 26 | 女 | 90 | |

- 根据总成绩判断：全班 380 分及以上学习优秀、320 ~ 380 学习良好、320 以下学习一般

```

DELIMITER $
CREATE PROCEDURE pro_test4()
BEGIN
    DECLARE total INT; -- 定义总分数变量
    DECLARE description VARCHAR(10); -- 定义分数描述变量
    SELECT SUM(score) INTO total FROM student; -- 为总分数变量赋值
    -- 判断总分数
    IF total >= 380 THEN
        SET description = '学习优秀';
    ELSEIF total >=320 AND total < 380 THEN
        SET description = '学习良好';
    ELSE
        SET description = '学习一般';
    END IF;
END$

```

```
    END IF;
END$  
DELIMITER ;  
-- 调用pro_test4存储过程  
CALL pro_test4();
```

参数传递

- 参数传递的语法

IN: 代表输入参数, 需要由调用者传递实际数据, 默认的

OUT: 代表输出参数, 该参数可以作为返回值

INOUT: 代表既可以作为输入参数, 也可以作为输出参数

```
DELIMITER $  
  
-- 标准语法  
CREATE PROCEDURE 存储过程名称([IN|OUT|INOUT] 参数名 数据类型)  
BEGIN  
    执行的sql语句;  
END$  
  
DELIMITER ;
```

- 输入总成绩变量, 代表学生总成绩, 输出分数描述变量, 代表学生总成绩的描述

```
DELIMITER $  
  
CREATE PROCEDURE pro_test6(IN total INT, OUT description VARCHAR(10))  
BEGIN  
    -- 判断总分数  
    IF total >= 380 THEN  
        SET description = '学习优秀';  
    ELSEIF total >= 320 AND total < 380 THEN  
        SET description = '学习不错';  
    ELSE  
        SET description = '学习一般';  
    END IF;  
END$  
  
DELIMITER ;  
-- 调用pro_test6存储过程  
CALL pro_test6(310,@description);  
CALL pro_test6((SELECT SUM(score) FROM student), @description);  
-- 查询总成绩描述  
SELECT @description;
```

- 查看参数方法

- @变量名: 用户会话变量, 代表整个会话过程他都是有作用的, 类似于全局变量

- @@变量名 : 系统变量

CASE

- 标准语法 1

```
CASE 表达式
    WHEN 值1 THEN 执行sql语句1;
    [WHEN 值2 THEN 执行sql语句2;]
    ...
    [ELSE 执行sql语句n;]
END CASE;
```

- 标准语法 2

```
SCASE
    WHEN 判断条件1 THEN 执行sql语句1;
    [WHEN 判断条件2 THEN 执行sql语句2;]
    ...
    [ELSE 执行sql语句n;]
END CASE;
```

- 演示

```
DELIMITER $
CREATE PROCEDURE pro_test7(IN total INT)
BEGIN
    -- 定义变量
    DECLARE description VARCHAR(10);
    -- 使用case判断
    CASE
        WHEN total >= 380 THEN
            SET description = '学习优秀';
        WHEN total >= 320 AND total < 380 THEN
            SET description = '学习不错';
        ELSE
            SET description = '学习一般';
    END CASE;

    -- 查询分数描述信息
    SELECT description;
END$
DELIMITER ;
-- 调用pro_test7存储过程
CALL pro_test7(390);
CALL pro_test7((SELECT SUM(score) FROM student));
```

WHILE

- while 循环语法

```
WHILE 条件判断语句 DO  
    循环体语句;  
    条件控制语句;  
END WHILE;
```

- 计算 1~100 之间的偶数和

```
DELIMITER $  
CREATE PROCEDURE pro_test6()  
BEGIN  
    -- 定义求和变量  
    DECLARE result INT DEFAULT 0;  
    -- 定义初始化变量  
    DECLARE num INT DEFAULT 1;  
    -- while循环  
    WHILE num <= 100 DO  
        IF num % 2 = 0 THEN  
            SET result = result + num;  
        END IF;  
        SET num = num + 1;  
    END WHILE;  
    -- 查询求和结果  
    SELECT result;  
END$  
DELIMITER ;  
  
-- 调用pro_test6存储过程  
CALL pro_test6();
```

REPEAT

- repeat 循环标准语法

```
初始化语句;  
REPEAT  
    循环体语句;  
    条件控制语句;  
    UNTIL 条件判断语句  
END REPEAT;
```

- 计算 1~10 之间的和

```
DELIMITER $
```

```

CREATE PROCEDURE pro_test9()
BEGIN
    -- 定义求和变量
    DECLARE result INT DEFAULT 0;
    -- 定义初始化变量
    DECLARE num INT DEFAULT 1;
    -- repeat循环
    REPEAT
        -- 累加
        SET result = result + num;
        -- 让num+1
        SET num = num + 1;
        -- 停止循环
        UNTIL num > 10
    END REPEAT;
    -- 查询求和结果
    SELECT result;
END$


DELIMITER ;
-- 调用pro_test9存储过程
CALL pro_test9();

```

LOOP

LOOP 实现简单的循环，退出循环的条件需要使用其他的语句定义，通常可以使用 LEAVE 语句实现，如果不加退出循环的语句，那么就变成了死循环

- loop 循环标准语法

```

[循环名称:] LOOP
    条件判断语句
        [LEAVE 循环名称;]
    循环体语句;
    条件控制语句;
END LOOP 循环名称;

```

- 计算 1~10 之间的和

```

DELIMITER $
CREATE PROCEDURE pro_test10()
BEGIN
    -- 定义求和变量
    DECLARE result INT DEFAULT 0;
    -- 定义初始化变量
    DECLARE num INT DEFAULT 1;
    -- Loop循环
    1:LOOP
        -- 条件成立，停止循环
        IF num > 10 THEN

```

```
        LEAVE l;
    END IF;
    -- 累加
    SET result = result + num;
    -- 让num+1
    SET num = num + 1;
END LOOP l;
-- 查询求和结果
SELECT result;
END$  
DELIMITER ;
-- 调用pro_test10存储过程
CALL pro_test10();
```

游标

游标是用来存储查询结果集的数据类型，在存储过程和函数中可以使用光标对结果集进行循环的处理

- 游标可以遍历返回的多行结果，每次拿到一行数据
- 简单来说游标就类似于集合的迭代器遍历
- MySQL 中的游标只能用在存储过程和函数中

游标的语法

- 创建游标

```
DECLARE 游标名称 CURSOR FOR 查询sql语句;
```

- 打开游标

```
OPEN 游标名称;
```

- 使用游标获取数据

```
FETCH 游标名称 INTO 变量名1,变量名2,...;
```

- 关闭游标

```
CLOSE 游标名称;
```

- Mysql 通过一个 Error handler 声明来判断指针是否到尾部，并且必须和创建游标的 SQL 语句声明在一起：

```
DECLARE EXIT HANDLER FOR NOT FOUND (do some action, 一般是设置标志变量)
```

游标的基本使用

- 数据准备：表 student

| | id | NAME | age | gender | score |
|---|----|------|-----|--------|-------|
| 1 | 张三 | 23 | 男 | 95 | |
| 2 | 李四 | 24 | 男 | 98 | |
| 3 | 王五 | 25 | 女 | 100 | |
| 4 | 赵六 | 26 | 女 | 90 | |

- 创建 stu_score 表

```
CREATE TABLE stu_score(
    id INT PRIMARY KEY AUTO_INCREMENT,
    score INT
);
```

- 将student表中所有的成绩保存到stu_score表中

```
DELIMITER $

CREATE PROCEDURE pro_test12()
BEGIN
    -- 定义成绩变量
    DECLARE s_score INT;
    -- 定义标记变量
    DECLARE flag INT DEFAULT 0;

    -- 创建游标，查询所有学生成绩数据
    DECLARE stu_result CURSOR FOR SELECT score FROM student;
    -- 游标结束后，将标记变量改为1 这两个必须声明在一起
    DECLARE EXIT HANDLER FOR NOT FOUND SET flag = 1;

    -- 开启游标
    OPEN stu_result;
    -- 循环使用游标
    REPEAT
        -- 使用游标，遍历结果，拿到数据
        FETCH stu_result INTO s_score;
        -- 将数据保存到stu_score表中
        INSERT INTO stu_score VALUES (NULL,s_score);
    UNTIL flag=1
    END REPEAT;
    -- 关闭游标
    CLOSE stu_result;
END$


DELIMITER ;

-- 调用pro_test12存储过程
CALL pro_test12();
-- 查询stu_score表
SELECT * FROM stu_score;
```

存储函数

存储函数和存储过程是非常相似的，存储函数可以做的事情，存储过程也可以做到

存储函数有返回值，存储过程没有返回值（参数的 out 其实也相当于是返回数据了）

- 创建存储函数

```
DELIMITER $  
-- 标准语法  
CREATE FUNCTION 函数名称(参数 数据类型)  
RETURNS 返回值类型  
BEGIN  
    执行的sql语句;  
    RETURN 结果;  
END$  
  
DELIMITER ;
```

- 调用存储函数，因为有返回值，所以使用 SELECT 调用

```
SELECT 函数名称(实际参数);
```

- 删除存储函数

```
DROP FUNCTION 函数名称;
```

- 定义存储函数，获取学生表中成绩大于95分的学生数量

```
DELIMITER $  
CREATE FUNCTION fun_test()  
RETURN INT  
BEGIN  
    -- 定义统计变量  
    DECLARE result INT;  
    -- 查询成绩大于95分的学生数量，给统计变量赋值  
    SELECT COUNT(score) INTO result FROM student WHERE score > 95;  
    -- 返回统计结果  
    SELECT result;  
END  
DELIMITER ;  
-- 调用fun_test存储函数  
SELECT fun_test();
```

触发器

基本介绍

触发器是与表有关的数据库对象，在 insert/update/delete 之前或之后触发并执行触发器中定义的 SQL 语句

- 触发器的这种特性可以协助应用在数据库端确保数据的完整性、日志记录、数据校验等操作
- 使用别名 NEW 和 OLD 来引用触发器中发生变化的记录内容，这与其他的数据库是相似的
- 现在触发器还只支持行级触发，不支持语句级触发

| 触发器类型 | OLD的含义 | NEW的含义 |
|-------------|-------------------|-------------------|
| INSERT 型触发器 | 无 (因为插入前状态无数据) | NEW 表示将要或者已经新增的数据 |
| UPDATE 型触发器 | OLD 表示修改之前的数据 | NEW 表示将要或已经修改后的数据 |
| DELETE 型触发器 | OLD 表示将要或者已经删除的数据 | 无 (因为删除后状态无数据) |

基本操作

- 创建触发器

```
DELIMITER $  
  
CREATE TRIGGER 触发器名称  
BEFORE|AFTER  INSERT|UPDATE|DELETE  
ON 表名  
[FOR EACH ROW]  -- 行级触发器  
BEGIN  
    触发器要执行的功能;  
END$  
  
DELIMITER ;
```

- 查看触发器的状态、语法等信息

```
SHOW TRIGGERS;
```

- 删除触发器，如果没有指定 schema_name，默認為当前数据库

```
DROP TRIGGER [schema_name.]trigger_name;
```

触发演示

通过触发器记录账户表的数据变更日志。包含：增加、修改、删除

- 数据准备

```
-- 创建db9数据库
CREATE DATABASE db9;
-- 使用db9数据库
USE db9;
```

```
-- 创建账户表account
CREATE TABLE account(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 账户id
    name VARCHAR(20),                      -- 姓名
    money DOUBLE                          -- 余额
);
-- 添加数据
INSERT INTO account VALUES (NULL, '张三', 1000), (NULL, '李四', 2000);
```

```
-- 创建日志表account_log
CREATE TABLE account_log(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 日志id
    operation VARCHAR(20),                  -- 操作类型 (insert update delete)
    operation_time DATETIME,                -- 操作时间
    operation_id INT,                      -- 操作表的id
    operation_params VARCHAR(200)           -- 操作参数
);
```

- 创建 INSERT 型触发器

```
DELIMITER $

CREATE TRIGGER account_insert
AFTER INSERT
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL, 'INSERT', NOW(), new.id, CONCAT('插入后
{id=' , new.id, ',name=' , new.name, ',money=' , new.money, '}'));
END$

DELIMITER ;
```

```
-- 向account表添加记录
INSERT INTO account VALUES (NULL, '王五', 3000);

-- 查询日志表
SELECT * FROM account_log;
/*
id  operation  operation_time      operation_id  operation_params
1   INSERT     2021-01-26 19:51:11    3           插入后{id=3, name=王五
money=2000}
*/

```

- 创建 UPDATE 型触发器

```
DELIMITER $

CREATE TRIGGER account_update
AFTER UPDATE
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL, 'UPDATE', NOW(), new.id, CONCAT('修改前
{id=' , old.id , ',name=' , old.name , ',money=' , old.money , '}'), '修改后
{id=' , new.id , ',name=' , new.name , ',money=' , new.money , '}'));
END$

DELIMITER ;
```

```
-- 修改account表
UPDATE account SET money=3500 WHERE id=3;

-- 查询日志表
SELECT * FROM account_log;
/*
id  operation  operation_time      operation_id  operation_params
2   UPDATE     2021-01-26 19:58:54    2           更新前{id=2, name=李四
money=1000}
                                         更新后{id=2, name=李四
money=200}
*/

```

- 创建 DELETE 型触发器

```

DELIMITER $

CREATE TRIGGER account_delete
AFTER DELETE
ON account
FOR EACH ROW
BEGIN
    INSERT INTO account_log VALUES (NULL, 'DELETE', NOW(), old.id, CONCAT('删除前
{id=' , old.id , ',name=' , old.name , ',money=' , old.money , '}'));
END$

DELIMITER ;

```

```

-- 删除account表数据
DELETE FROM account WHERE id=3;

-- 查询日志表
SELECT * FROM account_log;
/*
id operation   operation_time      operation_id   operation_params
3   DELETE      2021-01-26 20:02:48     3           删除前{id=3,name=王五
money=2000}
*/

```

存储引擎

基本介绍

对比其他数据库，MySQL 的架构可以在不同场景应用并发挥良好作用，主要体现在存储引擎，插件式的存储引擎架构将查询处理和其他的系统任务以及数据的存储提取分离，可以针对不同的存储需求可以选择最优的存储引擎

存储引擎的介绍：

- MySQL 数据库使用不同的机制存取表文件，机制的差别在于不同的存储方式、索引技巧、锁定水平等不同的功能和能力，在 MySQL 中，将这些不同的技术及配套的功能称为存储引擎
- Oracle、SqlServer 等数据库只有一种存储引擎，MySQL 提供了**插件式的存储引擎架构**，所以 MySQL 存在多种存储引擎，就会让数据库采取了不同的处理数据的方式和扩展功能
- 在关系型数据库中数据的存储是以表的形式存进行，所以存储引擎也称为表类型（存储和操作此表的类型）
- 通过选择不同的引擎，能够获取最佳的方案，也能够获得额外的速度或者功能，提高程序的整体效果。

MySQL 支持的存储引擎：

- MySQL 支持的引擎包括：InnoDB、MyISAM、MEMORY、Archive、Federate、CSV、BLACKHOLE 等
 - MySQL5.5 之前的默认存储引擎是 MyISAM，5.5 之后就改为了 InnoDB
-

引擎对比

MyISAM 存储引擎：

- 特点：不支持事务和外键，读取速度快，节约资源
- 应用场景：**适用于读多写少的场景**，对事务的完整性要求不高，比如一些数仓、离线数据、支付宝的年度总结之类的场景，业务进行只读操作，查询起来会更快
- 存储方式：
 - 每个 MyISAM 在磁盘上存储成 3 个文件，其文件名都和表名相同，拓展名不同
 - 表的定义保存在 .frm 文件，表数据保存在 .MYD (MYData) 文件中，索引保存在 .MYI (MYIndex) 文件中

InnoDB 存储引擎：(MySQL5.5 版本后默认的存储引擎)

- 特点：**支持事务和外键操作**，支持并发控制。对比 MyISAM 的存储引擎，InnoDB 写的处理效率差一些，并且会占用更多的磁盘空间以保留数据和索引
- 应用场景：对事务的完整性有比较高的要求，在并发条件下要求数据的一致性，读写频繁的操作
- 存储方式：
 - 使用共享表空间存储，这种方式创建的表的表结构保存在 .frm 文件中，数据和索引保存在 innodb_data_home_dir 和 innodb_data_file_path 定义的表空间中，可以是多个文件
 - 使用多表空间存储，创建的表的表结构存在 .frm 文件中，每个表的数据和索引单独保存在 .ibd 中

MEMORY 存储引擎：

- 特点：每个 MEMORY 表实际对应一个磁盘文件，该文件中只存储表的结构，表数据保存在内存中，且默认**使用 HASH 索引**，所以数据默认就是无序的，但是在需要快速定位记录可以提供更快的访问，**服务一旦关闭，表中的数据就会丢失**，存储不安全
- 应用场景：**缓存型存储引擎**，通常用于更新不太频繁的小表，用以快速得到访问结果
- 存储方式：表结构保存在 .frm 中

MERGE 存储引擎：

- 特点：
 - 是一组 MyISAM 表的组合，这些 MyISAM 表必须结构完全相同，通过将不同的表分布在多个磁盘上
 - MERGE 表本身并没有存储数据，对 MERGE 类型的表可以进行查询、更新、删除操作，这些操作实际上是对内部的 MyISAM 表进行的
- 应用场景：将一系列等同的 MyISAM 表以逻辑方式组合在一起，并作为一个对象引用他们，适合做数据仓库
- 操作方式：
 - 插入操作是通过 INSERT_METHOD 子句定义插入的表，使用 FIRST 或 LAST 值使得插入操作被相应地作用在第一个或者最后一个表上；不定义这个子句或者定义为 NO，表示不能对

MERGE 表执行插入操作

- 对 MERGE 表进行 DROP 操作，但是这个操作只是删除 MERGE 表的定义，对内部的表是没有任何影响的

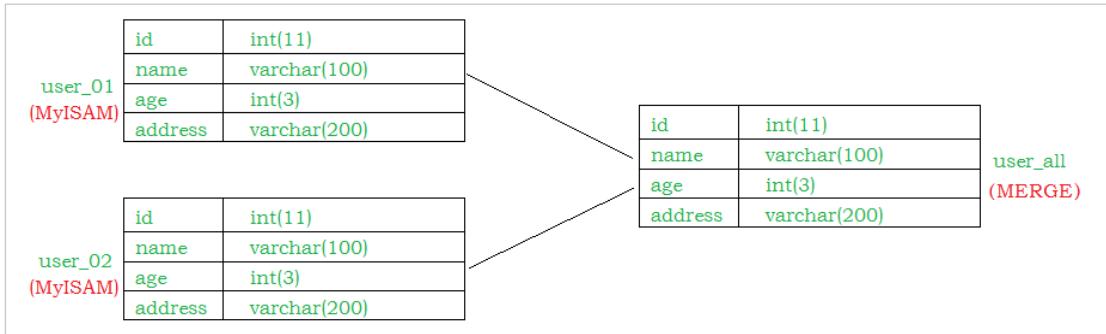
```

CREATE TABLE order_1(
)ENGINE = MyISAM DEFAULT CHARSET=utf8;

CREATE TABLE order_2(
)ENGINE = MyISAM DEFAULT CHARSET=utf8;

CREATE TABLE order_all(
-- 结构与MyISAM表相同
)ENGINE = MERGE UNION = (order_1,order_2) INSERT_METHOD=LAST DEFAULT
CHARSET=utf8;

```



| 特性 | MyISAM | InnoDB | MEMORY |
|-----------|------------------|--------|-------------|
| 存储限制 | 有 (平台对文件系统大小的限制) | 64TB | 有 (平台的内存限制) |
| 事务安全 | 不支持 | 支持 | 不支持 |
| 锁机制 | 表锁 | 表锁/行锁 | 表锁 |
| B+Tree 索引 | 支持 | 支持 | 支持 |
| 哈希索引 | 不支持 | 不支持 | 支持 |
| 全文索引 | 支持 | 支持 | 不支持 |
| 集群索引 | 不支持 | 支持 | 不支持 |
| 数据索引 | 不支持 | 支持 | 支持 |
| 数据缓存 | 不支持 | 支持 | N/A |
| 索引缓存 | 支持 | 支持 | N/A |
| 数据可压缩 | 支持 | 不支持 | 不支持 |
| 空间使用 | 低 | 高 | N/A |
| 内存使用 | 低 | 高 | 中等 |

| 特性 | MyISAM | InnoDB | MEMORY |
|--------|--------|--------|--------|
| 批量插入速度 | 高 | 低 | 高 |
| 外键 | 不支持 | 支持 | 不支持 |

只读场景 MyISAM 比 InnoDB 更快：

- 底层存储结构有差别，MyISAM 是非聚簇索引，叶子节点保存的是数据的具体地址，不用回表查询
- InnoDB 每次查询需要维护 MVCC 版本状态，保证并发状态下的读写冲突问题

引擎操作

- 查询数据库支持的存储引擎

```
SHOW ENGINES;  
SHOW VARIABLES LIKE '%storage_engine%'; -- 查看MySQL数据库默认的存储引擎
```

- 查询某个数据库中所有数据表的存储引擎

```
SHOW TABLE STATUS FROM 数据库名称;
```

- 查询某个数据库中某个数据表的存储引擎

```
SHOW TABLE STATUS FROM 数据库名称 WHERE NAME = '数据表名称';
```

- 创建数据表，指定存储引擎

```
CREATE TABLE 表名(  
    列名, 数据类型,  
    ...  
) ENGINE = 引擎名称;
```

- 修改数据表的存储引擎

```
ALTER TABLE 表名 ENGINE = 引擎名称;
```

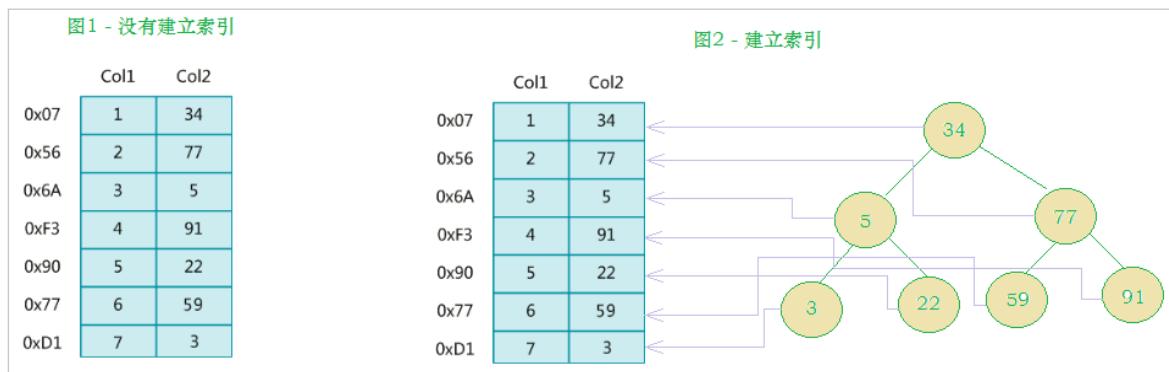
索引机制

索引介绍

基本介绍

MySQL 官方对索引的定义为：索引 (index) 是帮助 MySQL 高效获取数据的一种数据结构，**本质是排序好的快速查找数据结构**。在表数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构上实现高级查找算法，这种数据结构就是索引。**索引是在存储引擎层实现的**，所以并没有统一的索引标准，即不同存储引擎的索引的工作方式并不一样。

索引使用：一张数据表，用于保存数据；一个索引配置文件，用于保存索引；每个索引都指向了某一个数据



左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快 Col2 的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据的物理地址的指针，这样就可以运用二叉查找快速获取到相应数据

索引的优点：

- 类似于书籍的目录索引，提高数据检索的效率，降低数据库的 IO 成本
- 通过索引列对数据进行排序，降低数据排序的成本，降低 CPU 的消耗

索引的缺点：

- 一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式**存储在磁盘上**
- 虽然索引大大提高了查询效率，同时却也降低更新表的速度。对表进行 INSERT、UPDATE、DELETE 操作，MySQL 不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，还会调整因为更新所带来的键值变化后的索引信息，**但是更新数据也需要先从数据库中获取**，索引加快了获取速度，所以可以相互抵消一下。
- 索引会影响到 WHERE 的查询条件和排序 ORDER BY 两大功能

索引分类

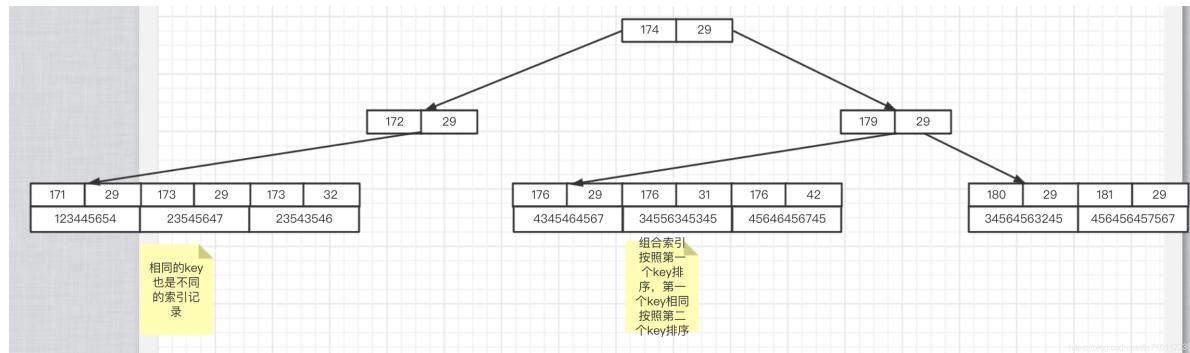
索引一般的分类如下：

- 功能分类
 - 主键索引：一种特殊的唯一索引，不允许有空值，一般在建表时同时创建主键索引
 - 单列索引：一个索引只包含单个列，一个表可以有多个单列索引（普通索引）

- 联合索引：顾名思义，就是将单列索引进行组合
- 唯一索引：索引列的值必须唯一，**允许有空值**，如果是联合索引，则列值组合必须唯一
 - NULL 值可以出现多次，因为两个 NULL 比较的结果既不相等，也不不等，结果仍然是未知
 - 可以声明不允许存储 NULL 值的非空唯一索引
- 外键索引：只有 InnoDB 引擎支持外键索引，用来保证数据的一致性、完整性和实现级联操作
- 结构分类
 - BTree 索引：MySQL 使用最频繁的一个索引数据结构，是 InnoDB 和 MyISAM 存储引擎默认的索引类型，底层基于 B+Tree
 - Hash 索引：MySQL 中 Memory 存储引擎默认支持的索引类型
 - R-tree 索引（空间索引）：空间索引是 MyISAM 引擎的一个特殊索引类型，主要用于地理空间数据类型
 - Full-text 索引（全文索引）：快速匹配全部文档的方式。MyISAM 支持，InnoDB 不支持 FULLTEXT 类型的索引，但是 InnoDB 可以使用 sphinx 插件支持全文索引，MEMORY 引擎不支持

| 索引 | InnoDB | MyISAM | Memory |
|-----------|------------|--------|--------|
| BTREE | 支持 | 支持 | 支持 |
| HASH | 不支持 | 不支持 | 支持 |
| R-tree | 不支持 | 支持 | 不支持 |
| Full-text | 5.6 版本之后支持 | 支持 | 不支持 |

联合索引图示：根据身高年龄建立的组合索引 (height、age)



索引操作

索引在创建表的时候可以同时创建，也可以随时增加新的索引

- 创建索引：如果一个表中有一列是主键，那么会**默认为其创建主键索引**（主键列不需要单独创建索引）

```
CREATE [UNIQUE|FULLTEXT] INDEX 索引名称 [USING 索引类型] ON 表名(列名...);  
-- 索引类型默认是 B+TREE
```

- 查看索引

```
SHOW INDEX FROM 表名;
```

- 添加索引

```
-- 单列索引  
ALTER TABLE 表名 ADD INDEX 索引名称(列名);  
  
-- 组合索引  
ALTER TABLE 表名 ADD INDEX 索引名称(列名1,列名2,...);  
  
-- 主键索引  
ALTER TABLE 表名 ADD PRIMARY KEY(主键列名);  
  
-- 外键索引(添加外键约束, 就是外键索引)  
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (本表外键列名) REFERENCES 主表名(主键列名);  
  
-- 唯一索引  
ALTER TABLE 表名 ADD UNIQUE 索引名称(列名);  
  
-- 全文索引(mysql只支持文本类型)  
ALTER TABLE 表名 ADD FULLTEXT 索引名称(列名);
```

- 删除索引

```
DROP INDEX 索引名称 ON 表名;
```

- 案例练习

数据准备: student

| id | NAME | age | score |
|----|------|-----|-------|
| 1 | 张三 | 23 | 99 |
| 2 | 李四 | 24 | 95 |
| 3 | 王五 | 25 | 98 |
| 4 | 赵六 | 26 | 97 |

索引操作:

```
-- 为student表中姓名列创建一个普通索引  
CREATE INDEX idx_name ON student(NAME);  
  
-- 为student表中年龄列创建一个唯一索引  
CREATE UNIQUE INDEX idx_age ON student(age);
```

聚簇索引

索引对比

聚簇索引是一种数据存储方式，并不是一种单独的索引类型

- 聚簇索引的叶子节点存放的是主键值和数据行，支持覆盖索引
- 非聚簇索引的叶子节点存放的是主键值或指向数据行的指针（由存储引擎决定）

在 Innodb 下主键索引是聚簇索引，在 MyISAM 下主键索引是非聚簇索引

Innodb

聚簇索引

在 Innodb 存储引擎，B+ 树索引可以分为聚簇索引（也称聚集索引、clustered index）和辅助索引（也称非聚簇索引或二级索引、secondary index、non-clustered index）

InnoDB 中，聚簇索引是按照每张表的主键构造一颗 B+ 树，叶子节点中存放的就是整张表的数据，将聚簇索引的叶子节点称为数据页

- 这个特性决定了**数据也是索引的一部分**，所以一张表只能有一个聚簇索引
- 辅助索引的存在不影响聚簇索引中数据的组织，所以一张表可以有多个辅助索引

聚簇索引的优点：

- 数据访问更快，聚簇索引将索引和数据保存在同一个 B+ 树中，因此从聚簇索引中获取数据比非聚簇索引更快
- 聚簇索引对于主键的排序查找和范围查找速度非常快

聚簇索引的缺点：

- 插入速度严重依赖于插入顺序，按照主键的顺序（递增）插入是最快的方式，否则将会出现页分裂，严重影响性能，所以对于 InnoDB 表，一般都会定义一个自增的 ID 列为主键
- 更新主键的代价很高，将会导致被更新的行移动，所以对于 InnoDB 表，一般定义主键为不可更新
- 二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据

辅助索引

在聚簇索引之上创建的索引称之为辅助索引，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引等

辅助索引叶子节点存储的是主键值，而不是数据的物理地址，所以访问数据需要二次查找，推荐使用覆盖索引，可以减少回表查询

检索过程: 辅助索引找到主键值，再通过聚簇索引（二分）找到数据页，最后通过数据页中的 Page Directory（二分）找到对应的数据分组，遍历组内所有的数据找到数据行

补充：无索引走全表查询，查到数据页后和上述步骤一致

索引实现

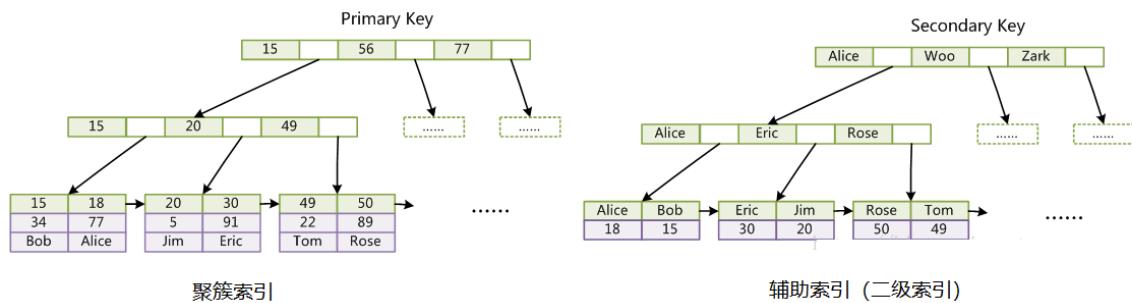
InnoDB 使用 B+Tree 作为索引结构，并且 InnoDB 一定有索引

主键索引：

- 在 InnoDB 中，表数据文件本身就是按 B+Tree 组织的一个索引结构，这个索引的 key 是数据表的主键，叶子节点 data 域保存了完整的数据记录
- InnoDB 的表数据文件通过主键聚集数据，如果没有定义主键，会选择非空唯一索引代替，如果没有这样的列，MySQL 会自动为 InnoDB 表生成一个隐含字段 **row_id** 作为主键，这个字段长度为 6 个字节，类型为长整形

辅助索引：

- InnoDB 的所有辅助索引（二级索引）都引用主键作为 data 域
- InnoDB 表是基于聚簇索引建立的，因此 InnoDB 的索引能提供一种非常快速的主键查找性能。不过辅助索引也会包含主键列，所以不建议使用过长的字段作为主键，**过长的主索引会令辅助索引变得过大**

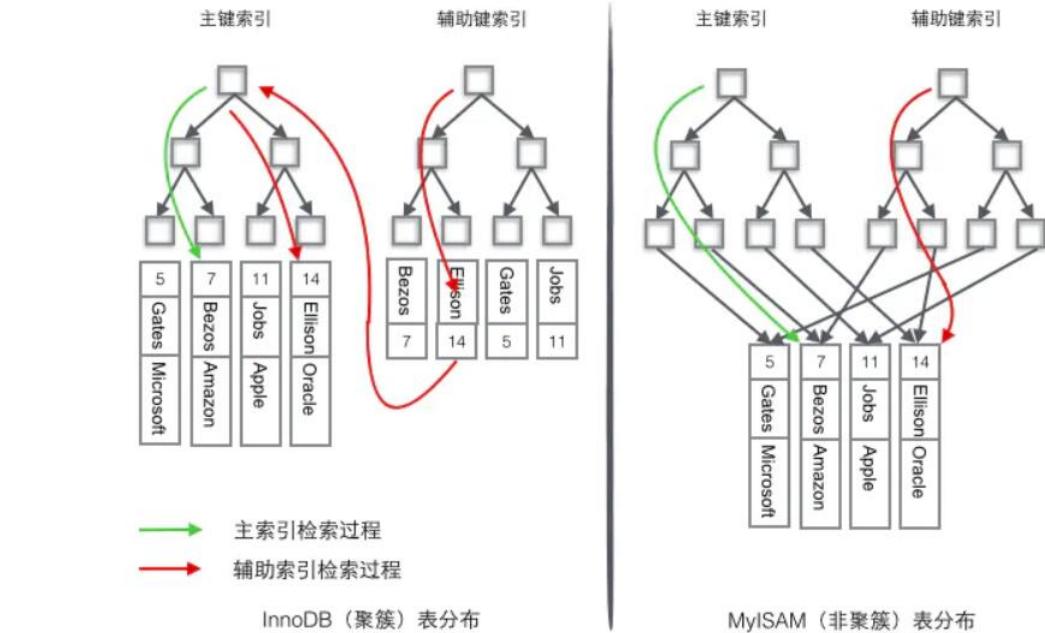


MyISAM

非聚簇

MyISAM 的主键索引使用的是非聚簇索引，索引文件和数据文件是分离的，**索引文件仅保存数据的地址**

- 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，通过辅助索引检索**无需回表查询**访问主键的索引树

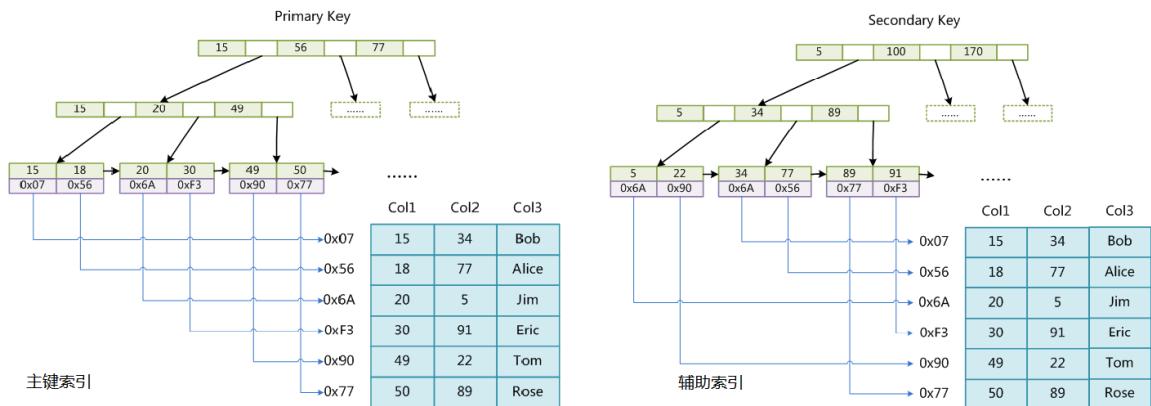


索引实现

MyISAM 的索引方式也叫做非聚集的，之所以这么称呼是为了与 InnoDB 的聚集索引区分

主键索引：MyISAM 引擎使用 B+Tree 作为索引结构，叶节点的 data 域存放的是数据记录的地址

辅助索引：MyISAM 中主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求 key 是唯一的，而辅助索引的 key 可以重复



参考文章：<https://blog.csdn.net/lm1060891265/article/details/81482136>

索引结构

数据页

文件系统的最小单元是块 (block) , 一个块的大小是 4K, 系统从磁盘读取数据到内存时是以磁盘块为基本单位的, 位于同一个磁盘块中的数据会被一次性读取出来, 而不是需要什么取什么

InnoDB 存储引擎中有页 (Page) 的概念, 页是 MySQL 磁盘管理的最小单位

- InnoDB 存储引擎中默认每个页的大小为 16KB, 索引中一个节点就是一个数据页, 所以会一次性读取 16KB 的数据到内存
- InnoDB 引擎将若干个地址连接磁盘块, 以此来达到页的大小 16KB
- 在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置, 这将会减少磁盘 I/O 次数, 提高查询效率

超过 16KB 的一条记录, 主键索引页只会存储部分数据和指向溢出页的指针, 剩余数据都会分散存储在溢出页中

数据页物理结构, 从上到下:

- File Header: 上一页和下一页的指针、该页的类型 (索引页、数据页、日志页等) 、校验和、LSN (最近一次修改当前页面时的系统 lsn 值, 事务持久性部分详解) 等信息
- Page Header: 记录状态信息
- Infimum + Supremum: 当前页的最小记录和最大记录 (头尾指针) , Infimum 所在分组只有一条记录, Supremum 所在分组可以有 1 ~ 8 条记录, 剩余的分组可以有 4 ~ 8 条记录
- User Records: 存储数据的记录
- Free Space: 尚未使用的存储空间
- Page Directory: 分组的目录, 可以通过目录快速定位 (二分法) 数据的分组
- File Trailer: 检验和字段, 在刷脏过程中, 页首和页尾的校验和一致才能说明页面刷新成功, 二者不同说明刷新期间发生了错误; LSN 字段, 也是用来校验页面的完整性

数据页中包含数据行, 数据的存储是基于数据行的, 数据行有 next_record 属性指向下一个行数据, 所以是可以遍历的, 但是一组数据至多 8 个行, 通过 Page Directory 先定位到组, 然后遍历获取所需的数据行即可

数据行中有三个隐藏字段: trx_id、roll_pointer、row_id (在事务章节会详细介绍它们的作用)

BTree

BTree 的索引类型是基于 B+Tree 树型数据结构的, B+Tree 又是 BTree 数据结构的变种, 用在数据库和操作系统中的文件系统, 特点是能够保持数据稳定有序

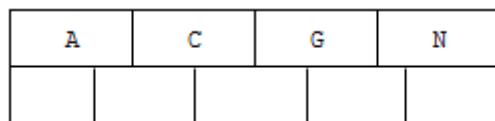
BTree 又叫多路平衡搜索树, 一颗 m 叉的 BTree 特性如下:

- 树中每个节点最多包含 m 个孩子
- 除根节点与叶子节点外, 每个节点至少有 $\lceil \text{ceil}(m/2) \rceil$ 个孩子
- 若根节点不是叶子节点, 则至少有两个孩子
- 所有的叶子节点都在同一层
- 每个非叶子节点由 n 个 key 与 $n+1$ 个指针组成, 其中 $\lceil \text{ceil}(m/2)-1 \rceil \leq n \leq m-1$

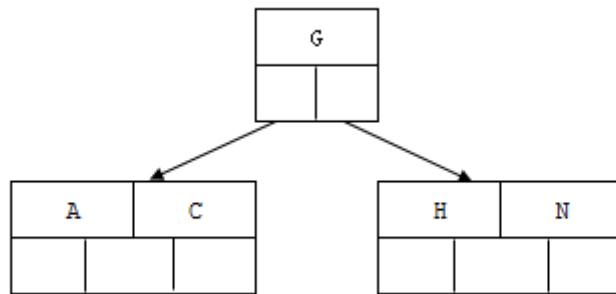
5 叉, key 的数量 $\lceil \text{ceil}(m/2)-1 \rceil \leq n \leq m-1$ 为 $2 \leq n \leq 4$, 当 $n > 4$ 时中间节点分裂到父节点, 两边节点分裂

插入 C N G A H E K Q M F W L T Z D P R X Y S 数据的工作流程:

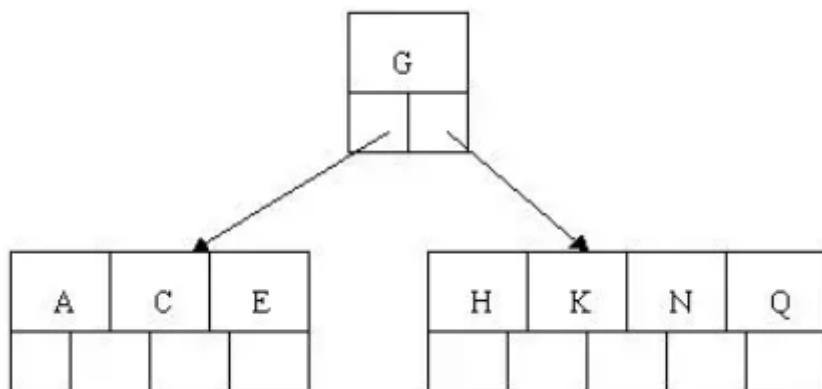
- 插入前 4 个字母 C N G A



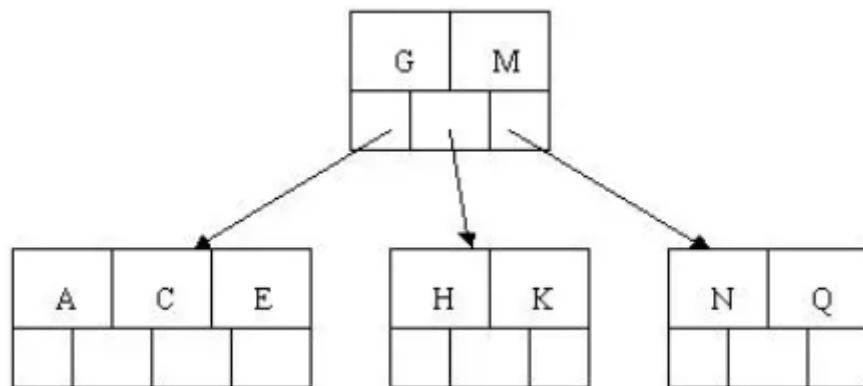
- 插入 H, n>4, 中间元素 G 字母向上分裂到新的节点



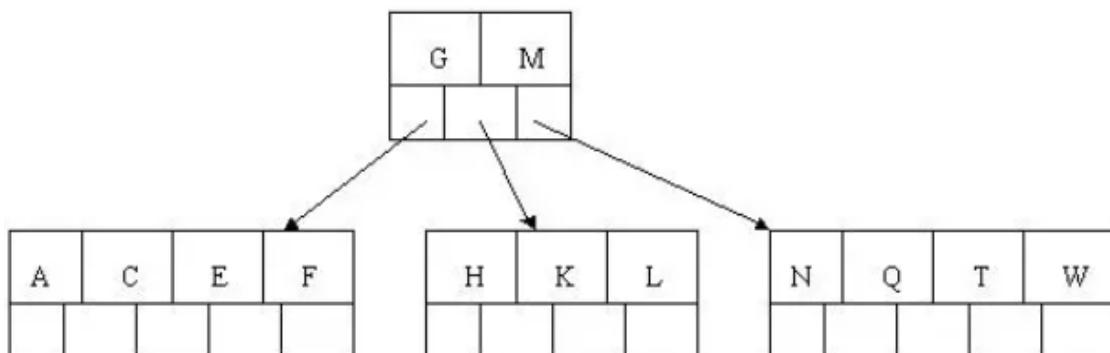
- 插入 E、K、Q 不需要分裂



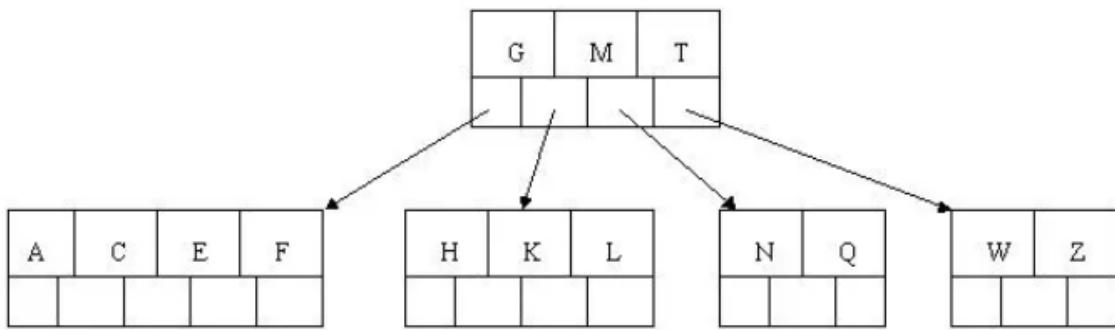
- 插入 M, 中间元素 M 字母向上分裂到父节点 G



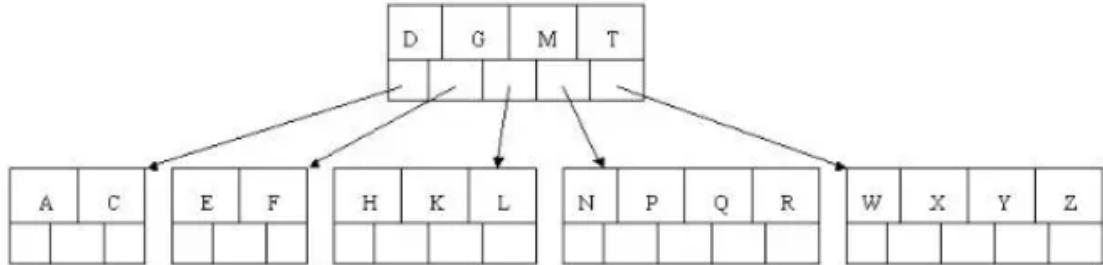
- 插入 F, W, L, T 不需要分裂



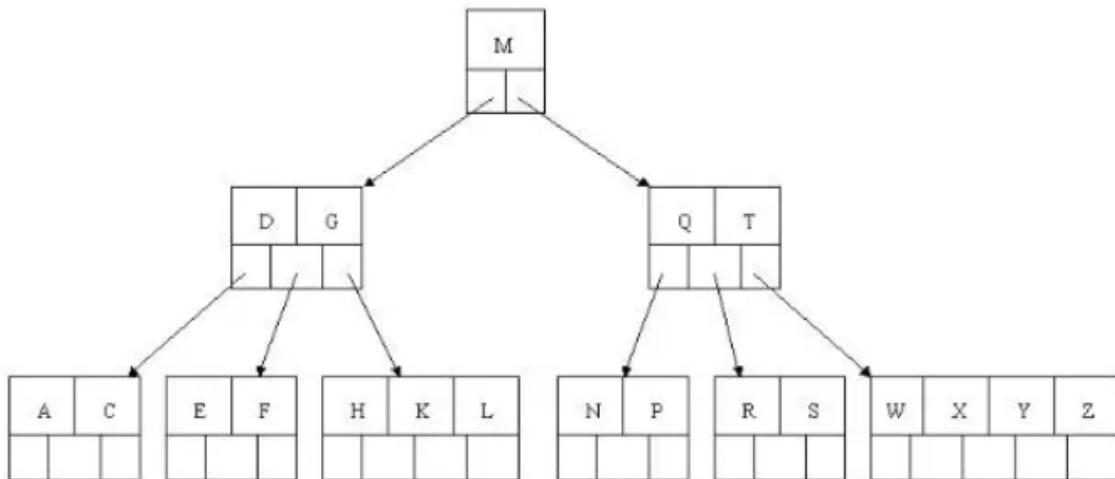
- 插入 Z, 中间元素 T 向上分裂到父节点中



- 插入 D, 中间元素 D 向上分裂到父节点中, 然后插入 P, R, X, Y 不需要分裂

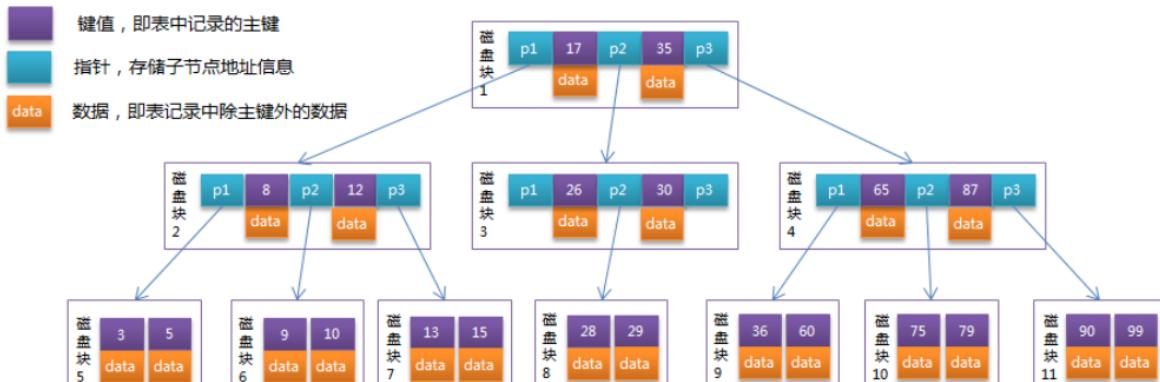


- 最后插入 S, NPQR 节点 n>5, 中间节点 Q 向上分裂, 但分裂后父节点 DGMT 的 n>5, 中间节点 M 向上分裂



BTree 树就已经构建完成了, BTree 树和二叉树相比, 查询数据的效率更高, 因为对于相同的数据量来说, **BTree 的层级结构比二叉树少**, 所以搜索速度快

BTree 结构的数据可以让系统高效的找到数据所在的磁盘块, 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, BTree 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



缺点: 当进行范围查找时会出现回旋查找

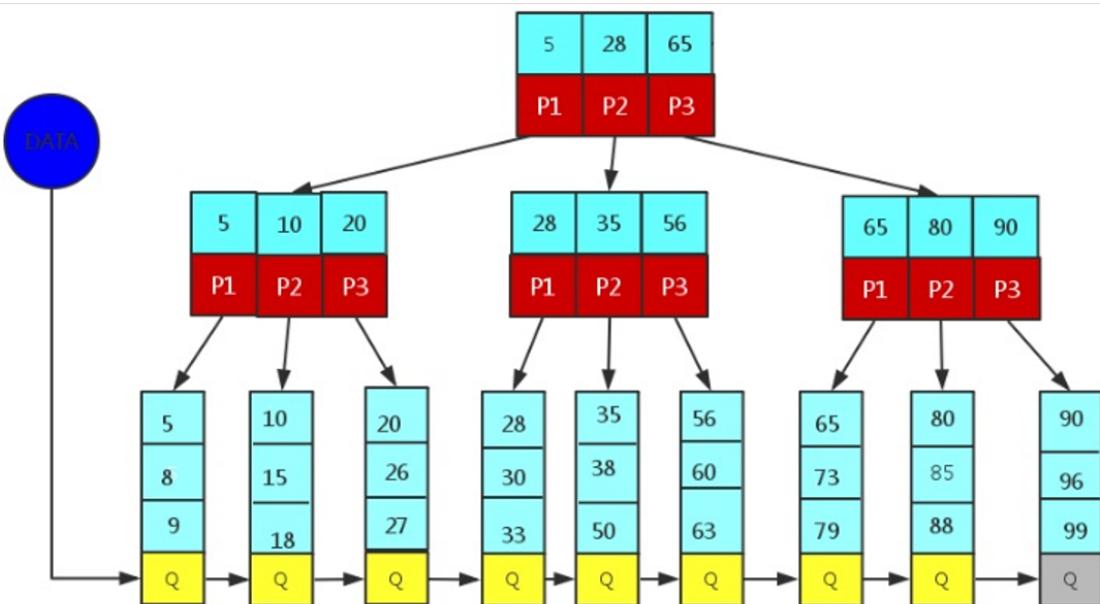
B+Tree

数据结构

BTree 数据结构中每个节点中不仅包含数据的 key 值，还有 data 值。磁盘中每一页的存储空间是有限的，如果 data 数据较大时将会导致每个节点（即一个页）能存储的 key 的数量很小，当存储的数据量很大时同样会导致 B-Tree 的深度较大，增大查询时的磁盘 I/O 次数，进而影响查询效率，所以引入 B+Tree

B+Tree 为 BTree 的变种，B+Tree 与 BTree 的区别为：

- n 叉 B+Tree 最多含有 n 个 key (哈希值)，而 BTree 最多含有 n-1 个 key
- 所有**非叶子节点只存储键值 key 信息**，只进行数据索引，使每个非叶子节点所能保存的关键字大大增加
- 所有**数据都存储在叶子节点**，所以每次数据查询的次数都一样
- **叶子节点按照 key 大小顺序排列，左边结尾数据都会保存右边节点开始数据的指针，形成一个链表**
- 所有节点中的 key 在叶子节点中也存在（比如 5），**key 允许重复**，B 树不同节点不存在重复的 key



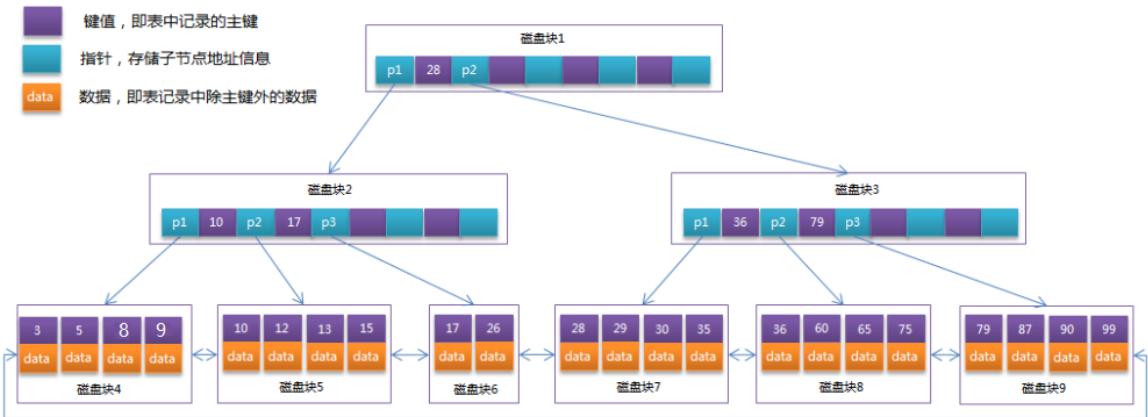
B* 树：是 B+ 树的变体，在 B+ 树的非根和非叶子结点再增加指向兄弟的指针

优化结构

MySQL 索引数据结构对经典的 B+Tree 进行了优化，在原 B+Tree 的基础上，增加一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的 B+Tree，提高区间访问的性能，防止回旋查找

区间访问的意思是访问索引为 5 - 15 的数据，可以直接根据相邻节点的指针遍历

B+ 树的**叶子节点是数据页** (page)，一个页里面可以存多个数据行



通常在 B+Tree 上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。可以对 B+Tree 进行两种查找运算：

- 有范围：对于主键的范围查找和分页查找
- 有顺序：从根节点开始，进行随机查找，顺序查找

InnoDB 中每个数据页的大小默认是 16KB，

- 索引行：一般表的主键类型为 INT (4 字节) 或 BIGINT (8 字节)，指针大小在 InnoDB 中设置为 6 字节，也就是说一个页大概存储 $16\text{KB}/(8B+6B)=1\text{K}$ 个键值 (估值)。则一个深度为 3 的 B+Tree 索引可以维护 $10^3 * 10^3 * 10^3 = 10\text{亿}$ 条记录
- 数据行：一行数据的大小可能是 1k，一个数据页可以存储 16 行

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree 的高度一般都在 2-4 层。MySQL 的 InnoDB 存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要 1~3 次磁盘 I/O 操作

B+Tree 优点：提高查询速度，减少磁盘的 IO 次数，树形结构较小

索引维护

B+ 树为了保持索引的有序性，在插入新值的时候需要做相应的维护

每个索引中每个块存储在磁盘页中，可能会出现以下两种情况：

- 如果所在的数据页已经满了，这时候需要申请一个新的数据页，然后挪动部分数据过去，这个过程称为**页分裂**，原本放在一个页的数据现在分到两个页中，降低了空间利用率
- 当相邻两个页由于删除了数据，利用率很低之后，会将数据页做**页合并**，合并的过程可以认为是分裂过程的逆过程
- 这两个情况都是由 B+ 树的结构决定的

一般选用数据小的字段做索引，字段长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小

自增主键的插入数据模式，可以让主键索引尽量地保持递增顺序插入，不涉及到挪动其他记录，**避免了页分裂**，页分裂的目的就是保证后一个数据页中的所有行主键值比前一个数据页中主键值大

设计原则

索引的设计可以遵循一些已有的原则，创建索引的时候请尽量考虑符合这些原则，便于提升索引的使用效率

创建索引时的原则：

- 对查询频次较高，且数据量比较大的表建立索引
- 使用唯一索引，区分度越高，使用索引的效率越高
- 索引字段的选择，最佳候选列应当从 where 子句的条件中提取，使用覆盖索引
- 使用短索引，索引创建之后也是使用硬盘来存储的，因此提升索引访问的 I/O 效率，也可以提升总体的访问效率。假如构成索引的字段总长度比较短，那么在给定大小的存储块内可以存储更多的索引值，相应的可以有效的提升 MySQL 访问索引的 I/O 效率
- 索引可以有效的提升查询数据的效率，但索引数量不是多多益善，索引越多，维护索引的代价越高。对于插入、更新、删除等 DML 操作比较频繁的表来说，索引过多，会引入相当高的维护代价，降低 DML 操作的效率，增加相应操作的时间消耗；另外索引过多的话，MySQL 也会犯选择困难病，虽然最终仍然会找到一个可用的索引，但提高了选择的代价
- MySQL 建立联合索引时会遵守**最左前缀匹配原则**，即最左优先，在检索数据时从联合索引的最左边开始匹配

N 个列组合而成的组合索引，相当于创建了 N 个索引，如果查询时 where 句中使用了组成该索引的前几个字段，那么这条查询 SQL 可以利用组合索引来提升查询效率

```
-- 对name、address、phone列建一个联合索引
ALTER TABLE user ADD INDEX index_three(name,address,phone);
-- 查询语句执行时会依照最左前缀匹配原则，检索时分别会使用索引进行数据匹配。
(name,address,phone)
(name,address)
(name,phone)    -- 只有name字段走了索引
(name)

-- 索引的字段可以是任意顺序的，优化器会帮助我们调整顺序，下面的SQL语句可以命中索引
SELECT * FROM user WHERE address = '北京' AND phone = '12345' AND name = '张
三';
```

```
-- 如果联合索引中最左边的列不包含在条件查询中，SQL语句就不会命中索引，比如：
SELECT * FROM user WHERE address = '北京' AND phone = '12345';
```

哪些情况不要建立索引：

- 记录太少的表
- 经常增删改的表
- 频繁更新的字段不适合创建索引
- where 条件里用不到的字段不创建索引

索引优化

覆盖索引

覆盖索引：包含所有满足查询需要的数据的索引（SELECT 后面的字段刚好是索引字段），可以利用该索引返回 SELECT 列表的字段，而不必根据索引去聚簇索引上读取数据文件

回表查询：要查找的字段不在非主键索引树上时，需要通过叶子节点的主键值去主键索引上获取对应的行数据

使用覆盖索引，防止回表查询：

- 表 user 主键为 id，普通索引为 age，查询语句：

```
SELECT * FROM user WHERE age = 30;
```

查询过程：先通过普通索引 age=30 定位到主键值 id=1，再通过聚集索引 id=1 定位到行记录数据，需要两次扫描 B+ 树

- 使用覆盖索引：

```
DROP INDEX idx_age ON user;
CREATE INDEX idx_age_name ON user(age,name);
SELECT id,age FROM user WHERE age = 30;
```

在一棵索引树上就能获取查询所需的数据，无需回表速度更快

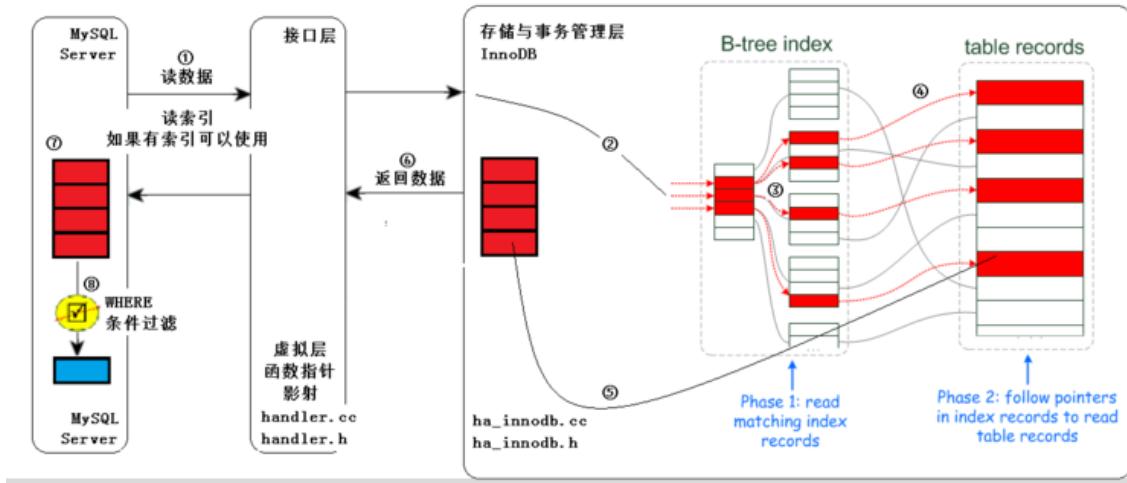
使用覆盖索引，要注意 SELECT 列表中只取出需要的列，不可用 SELECT *，所有字段一起做索引会导致索引文件过大，查询性能下降

索引下推

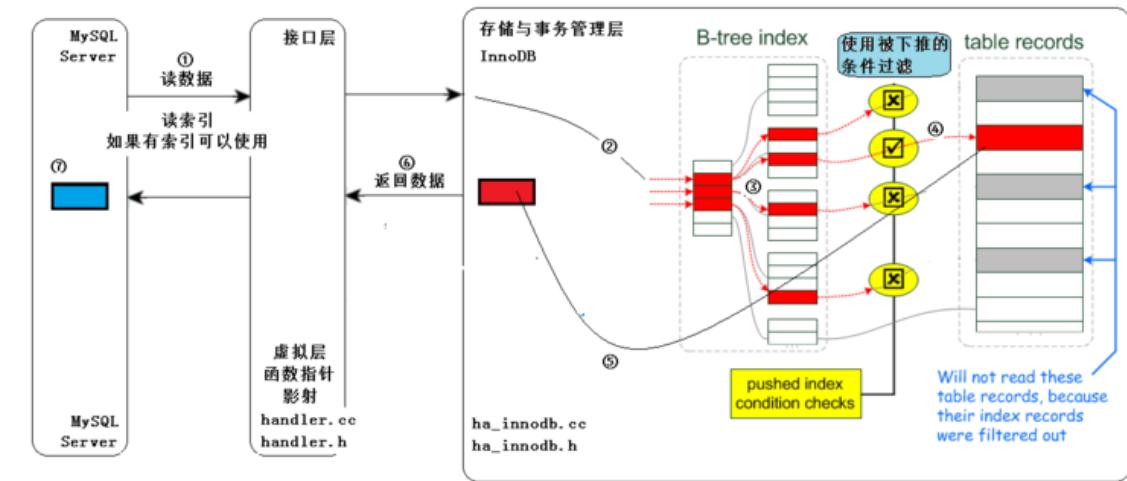
索引条件下推优化 (Index Condition Pushdown, ICP) 是 MySQL5.6 添加，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数

索引下推充分利用了索引中的数据，在查询出整行数据之前过滤掉无效的数据，再去主键索引树上查找

- 不使用索引下推优化时存储引擎通过索引检索到数据，然后回表查询记录返回给 Server 层，**服务器判断数据是否符合条件**



- 使用索引下推优化时，如果存在某些被索引的列的判断条件时，由存储引擎在索引遍历的过程中判断数据是否符合传递的条件，将符合条件的数据进行回表，检索出来返回给服务器，由此减少 IO 次数



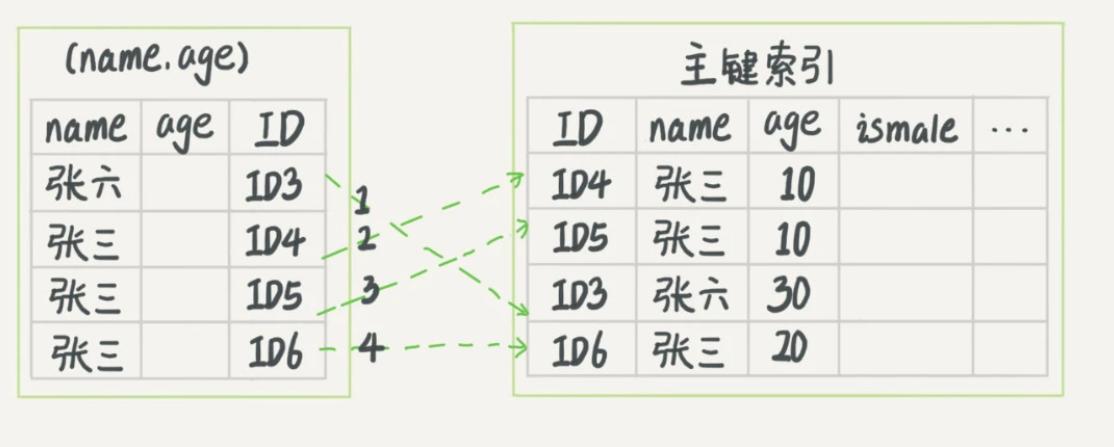
适用条件：

- 需要存储引擎将索引中的数据与条件进行判断（所以条件列必须都在同一个索引中），所以优化是基于存储引擎的，只有特定引擎可以使用，适用于 InnoDB 和 MyISAM
- 存储引擎没有调用跨存储引擎的能力，跨存储引擎的功能有存储过程、触发器、视图，所以调用这些功能的不可以进行索引下推优化
- 对于 InnoDB 引擎只适用于二级索引，InnoDB 的聚簇索引会将整行数据读到缓冲区，不再需要去回表查询了

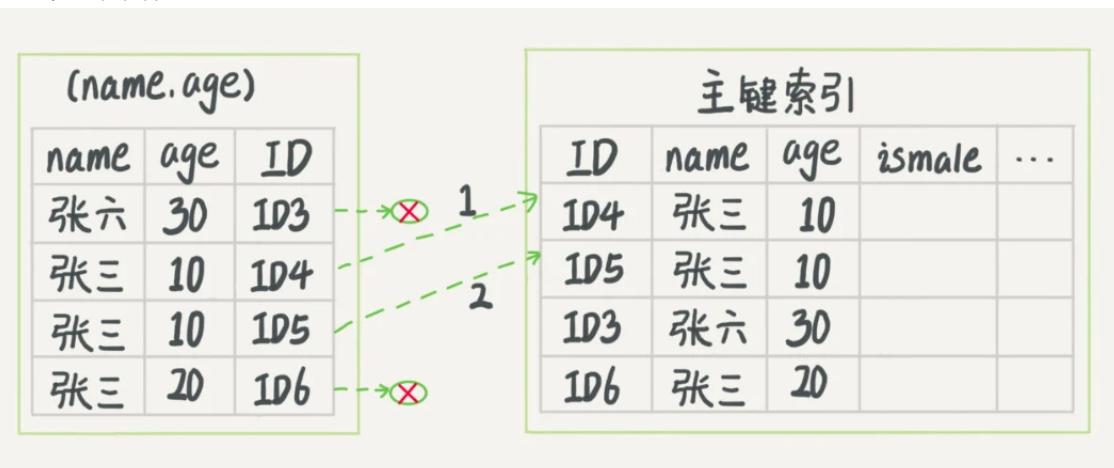
工作过程：用户表 user, (name, age) 是联合索引

```
SELECT * FROM user WHERE name LIKE '张%' AND age = 10; -- 头部模糊匹配会造成索引失效
```

- 优化前：在非主键索引树上找到满足第一个条件的行，然后通过叶子节点记录的主键值再回到主键索引树上查找到对应的行数据，再对比 AND 后的条件是否符合，符合返回数据，需要 4 次回表



- 优化后：检查索引中存储的列信息是否符合索引条件，然后交由存储引擎用剩余的判断条件判断此行数据是否符合要求，**不满足条件的不去读取表中的数据**，满足下推条件的就根据主键值进行回表查询，2次回表



当使用 EXPLAIN 进行分析时，如果使用了索引条件下推，Extra 会显示 Using index condition

参考文章：https://blog.csdn.net/sinat_29774479/article/details/103470244

参考文章：<https://time.geekbang.org/column/article/69636>

前缀索引

当要索引的列字符很多时，索引会变大变慢，可以只索引列开始的部分字符串，节约索引空间，提高索引效率

注意：使用前缀索引就系统就忽略覆盖索引对查询性能的优化了

优化原则：降低重复的索引值

比如地区表：

| area | gdp | code |
|---------------|-----|------|
| chinaShanghai | 100 | aaa |
| chinaDalian | 200 | bbb |
| usaNewYork | 300 | ccc |
| chinaFuxin | 400 | ddd |
| chinaBeijing | 500 | eee |

发现 area 字段很多都是以 china 开头的，那么如果以前 1-5 位字符做前缀索引就会出现大量索引值重复的情况，索引值重复性越低，查询效率也就越高，所以需要建立前 6 位字符的索引：

```
CREATE INDEX idx_area ON table_name(area(7));
```

场景：存储身份证件

- 直接创建完整索引，这样可能比较占用空间
- 创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引
- 倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不够的问题（前 6 位相同的很多）
- 创建 hash 字段索引，查询性能稳定，有额外的存储和计算消耗，跟第三种方式一样，都不支持范围扫描

索引合并

使用多个索引来完成一次查询的执行方法叫做索引合并 index merge

- Intersection 索引合并：

```
SELECT * FROM table_test WHERE key1 = 'a' AND key3 = 'b'; # key1 和 key3 列都是单列索引、二级索引
```

从不同索引中扫描到的记录的 id 值取交集（相同 id），然后执行回表操作，要求从每个二级索引获取到的记录都是按照主键值排序

- Union 索引合并：

```
SELECT * FROM table_test WHERE key1 = 'a' OR key3 = 'b';
```

从不同索引中扫描到的记录的 id 值取并集，然后执行回表操作，要求从每个二级索引获取到的记录都是按照主键值排序

- Sort-Union 索引合并

```
SELECT * FROM table_test WHERE key1 < 'a' OR key3 > 'b';
```

先将从不同索引中扫描到的记录的主键值进行排序，再按照 Union 索引合并的方式进行查询

索引合并算法的效率并不好，通过将其中的一个索引改成联合索引会优化效率

系统优化

表优化

分区表

基本介绍

分区表是将大表的数据按分区字段分成许多小的子集，建立一个以 ftime 年份为分区的表：

```
CREATE TABLE `t` (
    `ftime` datetime NOT NULL,
    `c` int(11) DEFAULT NULL,
    KEY (`ftime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB);
INSERT INTO t VALUES('2017-4-1',1),('2018-4-1',1);-- 这两行记录分别落在 p_2018 和
p_2019 这两个分区上
```

这个表包含了一个.frm 文件和 4 个.ibd 文件，每个分区对应一个.ibd 文件

- 对于引擎层来说，这是 4 个表，针对每个分区表的操作不会相互影响
- 对于 Server 层来说，这是 1 个表

分区策略

打开表行为：第一次访问一个分区表时，MySQL 需要把所有的分区都访问一遍，如果分区表的数量很多，超过了 open_files_limit 参数（默认值 1024），那么就会在访问这个表时打开所有的文件，导致打开表文件的个数超过了上限而报错

通用分区策略：MyISAM 分区表使用的分区策略，每次访问分区都由 Server 层控制，在文件管理、表管理的实现上很粗糙，因此有比较严重的性能问题

本地分区策略：从 MySQL 5.7.9 开始，InnoDB 引擎内部自己管理打开分区的行为，InnoDB 引擎打开文件超过 innodb_open_files 时就会关掉一些之前打开的文件，所以即使分区个数大于 open_files_limit，也不会报错

从 MySQL 8.0 版本开始，就不允许创建 MyISAM 分区表，只允许创建已经实现了本地分区策略的引擎，目前只有 InnoDB 和 NDB 这两个引擎支持了本地分区策略

Server 层

从 Server 层看一个分区表就只是一个表

- Session A:

```
SELECT * FROM t WHERE ftime = '2018-4-1';
```

- Session B:

```
ALTER TABLE t TRUNCATE PARTITION p_2017; -- blocked
```

现象: Session B 只操作 p_2017 分区, 但是由于 Session A 持有整个表 t 的 MDL 读锁, 就导致 B 的 ALTER 语句获取 MDL 写锁阻塞

分区表的特点:

- 第一次访问的时候需要访问所有分区
- 在 Server 层认为这是同一张表, 因此**所有分区共用同一个 MDL 锁**
- 在引擎层认为这是不同的表, 因此 MDL 锁之后的执行过程, 会根据分区表规则, 只访问需要的分区

应用场景

分区表的优点:

- 对业务透明, 相对于用户分表来说, 使用分区表的业务代码更简洁
- 分区表可以很方便的清理历史数据。按照时间分区的分区表, 就可以直接通过 `alter table t drop partition` 这个语法直接删除分区文件, 从而删掉过期的历史数据, 与使用 `drop` 语句删除数据相比, 优势是速度快、对系统影响小

使用分区表, 不建议创建太多的分区, 注意事项:

- 分区并不是越细越好, 单表或者单分区的数据一千万行, 只要没有特别大的索引, 对于现在的硬件能力来说都已经是小表
- 分区不要提前预留太多, 在使用之前预先创建即可。比如是按月分区, 每年年底时再把下一年度的 12 个新分区创建上即可, 并且对于没有数据的历史分区, 要及时的 `drop` 掉

参考文档: <https://time.geekbang.org/column/article/82560>

临时表

基本介绍

临时表分为内部临时表和用户临时表

- 内部临时表：系统执行 SQL 语句优化时产生的表，例如 Join 连接查询、去重查询等
- 用户临时表：用户主动创建的临时表

```
CREATE TEMPORARY TABLE temp_t like table_1;
```

临时表可以是内存表，也可以是磁盘表（多表操作 → 嵌套查询章节提及）

- 内存表指的是使用 Memory 引擎的表，建立哈希索引，建表语法是 `create table ... engine=memory`，这种表的数据都保存在内存里，系统重启时会被清空，但是表结构还在
- 磁盘表是使用 InnoDB 引擎或者 MyISAM 引擎的临时表，建立 B+ 树索引，写数据的时候是写到磁盘上的

临时表的特点：

- 一个临时表只能被创建它的 session 访问，对其他线程不可见，所以不同 session 的临时表是**可以重名的**
- 临时表可以与普通表同名，会话内有同名的临时表和普通表时，执行 show create 语句以及增删改查语句访问的都是临时表
- show tables 命令不显示临时表
- 数据库发生异常重启不需要担心数据删除问题，临时表会**自动回收**

重名原理

执行创建临时表的 SQL：

```
create temporary table temp_t(id int primary key)engine=innodb;
```

MySQL 给 InnoDB 表创建一个 frm 文件保存表结构定义，在 ibd 保存表数据。frm 文件放在临时文件目录下，文件名的后缀是 .frm，**前缀是 #sql{进程 id}_{线程 id}_ 序列号**，使用 `select @@tmpdir` 命令，来显示实例的临时文件目录

MySQL 维护数据表，除了物理磁盘上的文件外，内存里也有一套机制区别不同的表，每个表都对应一个 `table_def_key`

- 一个普通表的 `table_def_key` 的值是由 `库名 + 表名` 得到的，所以如果在同一个库下创建两个同名的普通表，创建第二个表的过程中就会发现 `table_def_key` 已经存在了
- 对于临时表，`table_def_key` 在 `库名 + 表名` 基础上，又加入了 `server_id + thread_id`，所以不同线程之间，临时表可以重名

实现原理：每个线程都维护了自己的临时表链表，每次 session 内操作表时，先遍历链表，检查是否有这个名字的临时表，如果有就**优先操作临时表**，如果没有再操作普通表；在 session 结束时对链表里的每个临时表，执行 `DROP TEMPORARY TABLE + 表名` 操作

执行 rename table 语句无法修改临时表，因为会按照 库名 / 表名.frm 的规则去磁盘找文件，但是临时表文件名的规则是 #sql{进程 id}_{线程 id}_ 序列号.frm，因此会报找不到文件名的错误

主备复制

创建临时表的语句会传到备库执行，因此备库的同步线程就会创建这个临时表。主库在线程退出时会自动删除临时表，但备库同步线程是持续在运行的并不会退出，所以这时就需要在主库上再写一个 DROP TEMPORARY TABLE 传给备库执行

binlog 日志写入规则：

- binlog_format=row，跟临时表有关的语句就不会记录到 binlog
- binlog_format=statement/mixed，binlog 中才会记录临时表的操作，也就会记录 DROP TEMPORARY TABLE 这条命令

主库上不同的线程创建同名的临时表是不冲突的，但是备库只有一个执行线程，所以 MySQL 在记录 binlog 时会把主库执行这个语句的线程 id 写到 binlog 中，在备库的应用线程就可以获取执行每个语句的主库线程 id，并利用这个线程 id 来构造临时表的 table_def_key

- session A 的临时表 t1，在备库的 table_def_key 就是：库名 + t1 +"M 的 serverid" + "session A 的 thread_id"
- session B 的临时表 t1，在备库的 table_def_key 就是：库名 + t1 +"M 的 serverid" + "session B 的 thread_id"

MySQL 在记录 binlog 的时不论是 create table 还是 alter table 语句都是原样记录，但是如果执行 drop table，系统记录 binlog 就会被服务端改写

```
DROP TABLE `t_normal` /* generated by server */
```

跨库查询

分库分表系统的跨库查询使用临时表不用担心线程之间的重名冲突，分库分表就是要把一个逻辑上的大表分散到不同的数据库实例上

比如将一个大表 ht，按照字段 f，拆分成 1024 个分表，分布到 32 个数据库实例上，一般情况下都有一个中间层 proxy 解析 SQL 语句，通过分库规则通过分表规则（比如 N%1024）确定将这条语句路由到哪个分表做查询

```
select v from ht where f=N;
```

如果这个表上还有另外一个索引 k，并且查询语句：

```
select v from ht where k >= M order by t_modified desc limit 100;
```

查询条件里面没有用到分区字段 f，只能到所有的分区中去查找满足条件的所有行，然后统一做 order by 操作，两种方式：

- 在 proxy 层的进程代码中实现排序，拿到分库的数据以后，直接在内存中参与计算，但是对 proxy 端的压力比较大，很容易出现内存不够用和 CPU 瓶颈问题
- 把各个分库拿到的数据，汇总到一个 MySQL 实例的一个表中，然后在这个汇总实例上做逻辑操作，执行流程：
 - 在汇总库上创建一个临时表 temp_ht，表里包含三个字段 v、k、t_modified
 - 在各个分库执行：`select v,k,t_modified from ht_x where k >= M order by t_modified desc limit 100`
 - 把分库执行的结果插入到 temp_ht 表中
 - 在临时表上执行：`select v from temp_ht order by t_modified desc limit 100`

优化步骤

执行频率

MySQL 客户端连接成功后，查询服务器状态信息：

```
SHOW [SESSION|GLOBAL] STATUS LIKE '';  
-- SESSION：显示当前会话连接的统计结果，默认参数  
-- GLOBAL：显示自数据库上次启动至今的统计结果
```

- 查看 SQL 执行频率：

```
SHOW STATUS LIKE 'Com_____';
```

Com_xxx 表示每种语句执行的次数

```
mysql> show status like 'Com_____';  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| Com_binlog | 0 |  
| Com_commit | 0 |  
| Com_delete | 0 |  
| Com_insert | 1 |  
| Com_repair | 0 |  
| Com_revoke | 0 |  
| Com_select | 6 |  
| Com_signal | 0 |  
| Com_update | 0 |  
| Com_xa_end | 0 |  
+-----+-----+  
10 rows in set (0.00 sec)
```

- 查询 SQL 语句影响的行数：

```
SHOW STATUS LIKE 'Innodb_rows_%';
```

```
mysql> show status like 'Innodb_rows_%';  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| Innodb_rows_deleted | 9 |  
| Innodb_rows_inserted | 9881021 |  
| Innodb_rows_read | 45633748 |  
| Innodb_rows_updated | 0 |  
+-----+-----+  
4 rows in set (0.00 sec)
```

Com_xxxx: 这些参数对于所有存储引擎的表操作都会进行累计

Innodb_xxxx: 这几个参数只是针对 InnoDB 存储引擎的，累加的算法也略有不同

| 参数 | 含义 |
|----------------------|---|
| Com_select | 执行 SELECT 操作的次数，一次查询只累加 1 |
| Com_insert | 执行 INSERT 操作的次数，对于批量插入的 INSERT 操作，只累加一次 |
| Com_update | 执行 UPDATE 操作的次数 |
| Com_delete | 执行 DELETE 操作的次数 |
| Innodb_rows_read | 执行 SELECT 查询返回的行数 |
| Innodb_rows_inserted | 执行 INSERT 操作插入的行数 |
| Innodb_rows_updated | 执行 UPDATE 操作更新的行数 |
| Innodb_rows_deleted | 执行 DELETE 操作删除的行数 |
| Connections | 试图连接 MySQL 服务器的次数 |
| Uptime | 服务器工作时间 |
| Slow_queries | 慢查询的次数 |

定位低效

SQL 执行慢有两种情况：

- 偶尔慢：DB 在刷新脏页（学完事务就懂了）
 - redo log 写满了
 - 内存不够用，要从 LRU 链表中淘汰
 - MySQL 认为系统空闲的时候
 - MySQL 关闭时
- 一直慢的原因：索引没有设计好、SQL 语句没写好、MySQL 选错了索引

通过以下两种方式定位执行效率较低的 SQL 语句

- 慢日志查询：慢查询日志在查询结束以后才记录，执行效率出现问题时查询日志并不能定位问题

配置文件修改：修改 .cnf 文件 `vim /etc/mysql/my.cnf`，重启 MySQL 服务器

```
slow_query_log=ON
slow_query_log_file=/usr/local/mysql/var/localhost-slow.log
long_query_time=1    #记录超过long_query_time秒的SQL语句的日志
log-queries-not-using-indexes = 1
```

使用命令配置：

```
mysql> SET slow_query_log=ON;
mysql> SET GLOBAL slow_query_log=ON;
```

查看是否配置成功：

```
SHOW VARIABLES LIKE '%query%'
```

- SHOW PROCESSLIST：实时查看当前 MySQL 在进行的连接线程，包括线程的状态、是否锁表、SQL 的执行情况，同时对一些锁表操作进行优化

```
mysql> show processlist;
+----+-----+-----+-----+-----+-----+
| Id | User | Host      | db     | Command | Time | State   | Info
+----+-----+-----+-----+-----+-----+
| 35 | root  | localhost | demo_02 | Query   | 0    | init    | show processlist
| 38 | root  | 192.168.192.1:53928 | demo_01 | Sleep   | 3278 |          | NULL
| 39 | root  | 192.168.192.1:53929 | NULL    | Sleep   | 3287 |          | NULL
+----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

EXPLAIN

执行计划

通过 EXPLAIN 命令获取执行 SQL 语句的信息，包括在 SELECT 语句执行过程中如何连接和连接的顺序，执行计划在优化器优化完成后、执行器之前生成，然后执行器会调用存储引擎检索数据

查询 SQL 语句的执行计划：

```
EXPLAIN SELECT * FROM table_1 WHERE id = 1;
```

```
mysql> explain select * from tb_item where id = 1;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE    | tb_item | const | PRIMARY       | PRIMARY | 4        | const | 1    | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

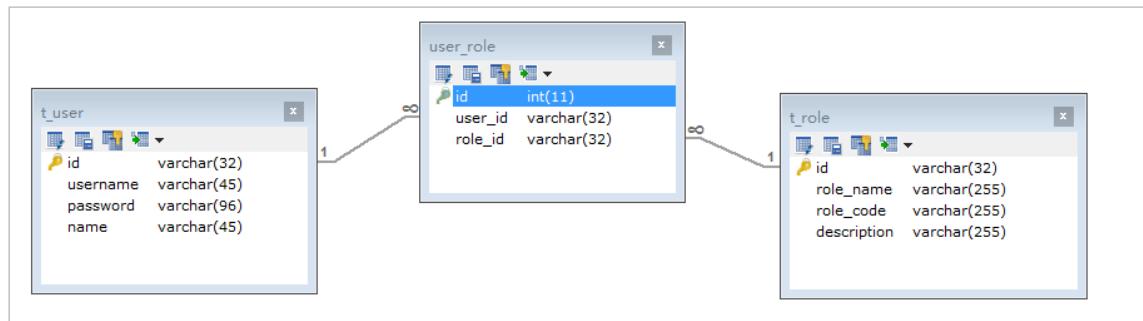
| 字段 | 含义 |
|---------------|--|
| id | SELECT 的序列号 |
| select_type | 表示 SELECT 的类型 |
| table | 访问数据库中表名称，有时可能是简称或者临时表名称 (<table_name>) |
| type | 表示表的连接类型 |
| possible_keys | 表示查询时，可能使用的索引 |
| key | 表示实际使用的索引 |
| key_len | 索引字段的长度 |
| ref | 表示与索引列进行等值匹配的对象，常数、某个列、函数等，type 必须在 (range, const] 之间，左闭右开 |
| rows | 扫描出的行数，表示 MySQL 根据表统计信息及索引选用情况， 估算 的找到所需的记录扫描的行数 |
| filtered | 条件过滤的行百分比，单表查询没意义，用于连接查询中对驱动表的扇出进行过滤，查询优化器预测所有扇出值满足剩余查询条件的百分比，相乘以后表示多表查询中还要对被驱动执行查询的次数 |
| extra | 执行情况的说明和描述 |

MySQL 执行计划的局限：

- 只是计划，不是执行 SQL 语句，可以随着底层优化器输入的更改而更改
- EXPLAIN 不会告诉显示关于触发器、存储过程的信息对查询的影响情况，不考虑各种 Cache
- EXPLAIN 不能显示 MySQL 在执行查询时的动态，因为执行计划在**执行查询之前生成**
- EXPALIN 只能解释 SELECT 操作，其他操作要重写为 SELECT 后查看执行计划
- EXPLAIN PLAN 显示的是在解释语句时数据库将如何运行 SQL 语句，由于执行环境和 EXPLAIN PLAN 环境的不同，此计划可能与 SQL 语句**实际的执行计划不同**，部分统计信息是估算的，并非精确值

SHOW WARINGS：在使用 EXPALIN 命令后执行该语句，可以查询与执行计划相关的拓展信息，展示出 Level、Code、Message 三个字段，当 Code 为 1003 时，Message 字段展示的信息类似于将查询语句重写后的信息，但是不是等价，不能执行复制过来运行

环境准备：



id

id 代表 SQL 执行的顺序的标识，每个 SELECT 关键字对应一个唯一 id，所以在同一个 SELECT 关键字中的表的 id 都是相同的。SELECT 后的 FROM 可以跟随多个表，每个表都会对应一条记录，这些记录的 id 都是相同的，

- id 相同时，执行顺序由上至下。连接查询的执行计划，记录的 id 值都是相同的，出现在前面的表为驱动表，后面为被驱动表

```
EXPLAIN SELECT * FROM t_role r, t_user u, user_role ur WHERE r.id = ur.role_id AND u.id = ur.user_id ;
```

```
mysql> explain select * from role r, user u, user_role ur where r.id = ur.role_id and u.id = ur.user_id ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | r | ALL | PRIMARY | NULL | NULL | NULL | 5 | NULL |
| 1 | SIMPLE | ur | ref | fk_ur_user_id,fk_ur_role_id | fk_ur_role_id | 99 | db03.r.id | 1 | Using where |
| 1 | SIMPLE | u | eq_ref | PRIMARY | PRIMARY | 98 | db03.ur.user_id | 1 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

- id 不同时，id 值越大优先级越高，越先被执行

```
EXPLAIN SELECT * FROM t_role WHERE id = (SELECT role_id FROM user_role WHERE user_id = (SELECT id FROM t_user WHERE username = 'stu1'))
```

```
mysql> EXPLAIN SELECT * FROM t_role WHERE id = (SELECT role_id FROM user_role WHERE user_id = (SELECT id FROM t_user WHERE username = 'stu1'));
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | t_role | const | PRIMARY | PRIMARY | 98 | const | 1 | NULL |
| 2 | SUBQUERY | user_role | ref | fk_ur_user_id | fk_ur_user_id | 99 | const | 1 | Using where |
| 3 | SUBQUERY | t_user | const | unique_user_username | unique_user_username | 137 | const | 1 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

- id 有相同也有不同时，id 相同的可以认为是一组，从上往下顺序执行；在所有的组中，id 的值越大的组，优先级越高，越先执行

```
EXPLAIN SELECT * FROM t_role r , (SELECT * FROM user_role ur WHERE ur.`user_id` = '2') a WHERE r.id = a.role_id ;
```

```
mysql> EXPLAIN SELECT * FROM t_role r , (SELECT * FROM user_role ur WHERE ur.`user_id` = '2') a WHERE r.id = a.role_id ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | <derived2> | ALL | NULL | NULL | NULL | NULL | 2 | Using where |
| 1 | PRIMARY | r | eq_ref | PRIMARY | PRIMARY | 98 | a.role_id | 1 | NULL |
| 2 | DERIVED | ur | ref | fk_ur_user_id | fk_ur_user_id | 99 | const | 1 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

- id 为 NULL 时代表的是临时表

select

表示查询中每个 select 子句的类型（简单 OR 复杂）

| select_type | 含义 |
|--------------------|--|
| SIMPLE | 简单的 SELECT 查询，查询中不包含子查询或者 UNION |
| PRIMARY | 查询中若包含任何复杂的子查询，最外层（也就是最左侧）查询标记为该标识 |
| UNION | 对于 UNION 或者 UNION ALL 的复杂查询，除了最左侧的查询，其余的小查询都是 UNION |
| UNION RESULT | UNION 需要使用临时表进行去重，临时表的是 UNION RESULT |
| DEPENDENT UNION | 对于 UNION 或者 UNION ALL 的复杂查询，如果各个小查询都依赖外层查询，是相关子查询，除了最左侧的小查询为 DEPENDENT SUBQUERY，其余都是 DEPENDENT UNION |
| SUBQUERY | 子查询不是相关子查询，该子查询第一个 SELECT 代表的查询就是这种类型，会进行物化（该子查询只需要执行一次） |
| DEPENDENT SUBQUERY | 子查询是相关子查询，该子查询第一个 SELECT 代表的查询就是这种类型，不会物化（该子查询需要执行多次） |
| DERIVED | 在 FROM 列表中包含的子查询，被标记为 DERIVED（衍生），也就是生成物化派生表的这个子查询 |
| MATERIALIZED | 将子查询物化后与与外层进行连接查询，生成物化表的子查询 |

子查询为 DERIVED: `SELECT * FROM (SELECT key1 FROM t1) AS derived_1 WHERE key1 > 10`

子查询为 MATERIALIZED: `SELECT * FROM t1 WHERE key1 IN (SELECT key1 FROM t2)`

type

对表的访问方式，表示 MySQL 在表中找到所需行的方式，又称访问类型

| type | 含义 |
|-----------------|---|
| ALL | 全表扫描，如果是 InnoDB 引擎是扫描聚簇索引 |
| index | 可以使用覆盖索引，但需要扫描全部索引 |
| range | 索引范围扫描，常见于 between、<、> 等的查询 |
| index_subquery | 子查询可以普通索引，则子查询的 type 为 index_subquery |
| unique_subquery | 子查询可以使用主键或唯一二级索引，则子查询的 type 为 index_subquery |
| index_merge | 索引合并 |
| ref_or_null | 非唯一性索引（普通二级索引）并且可以存储 NULL，进行等值匹配 |
| ref | 非唯一性索引与常量等值匹配 |
| eq_ref | 唯一性索引（主键或不存储 NULL 的唯一二级索引）进行等值匹配，如果二级索引是联合索引，那么所有联合的列都要进行等值匹配 |
| const | 通过主键或者唯一二级索引与常量进行等值匹配 |
| system | system 是 const 类型的特例，当查询的表只有一条记录的情况下，使用 system |
| NULL | MySQL 在优化过程中分解语句，执行时甚至不用访问表或索引 |

从上到下，性能从差到好，一般来说需要保证查询至少达到 range 级别，最好达到 ref

key

possible_keys:

- 指出 MySQL 能使用哪个索引在表中找到记录，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
- 如果该列是 NULL，则没有相关的索引

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
- 查询中若使用了覆盖索引，则该索引可能出现在 key 列表，不出现在 possible_keys

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
 - key_len 显示的值为索引字段的最大可能长度，并非实际使用长度，即 key_len 是根据表定义计算而得，不是通过表内检索出的
 - 在不损失精确性的前提下，长度越短越好
-

Extra

其他的额外的执行计划信息，在该列展示：

- No tables used: 查询语句中使用 FROM dual 或者没有 FROM 语句
- Impossible WHERE: 查询语句中的 WHERE 子句条件永远为 FALSE，会导致没有符合条件的行
- Using index: 该值表示相应的 SELECT 操作中使用了**覆盖索引** (Covering Index)
- Using index condition: 第一种情况是搜索条件中虽然出现了索引列，但是部分条件无法形成扫描区间（**索引失效**），会根据可用索引的条件先搜索一遍再匹配无法使用索引的条件，回表查询数据；第二种是使用了**索引条件下推优化**
- Using where: 搜索的数据需要在 Server 层判断，无法使用索引下推
- Using join buffer: 连接查询被驱动表无法利用索引，需要连接缓冲区来存储中间结果
- Using filesort: 无法利用索引完成排序（优化方向），需要对数据使用外部排序算法，将取得的数据在内存或磁盘中进行排序
- Using temporary: 表示 MySQL 需要使用临时表来存储结果集，常见于**排序、去重 (UNION)、分组等场景**
- Select tables optimized away: 说明仅通过使用索引，优化器可能仅从聚合函数结果中返回一行
- No tables used: Query 语句中使用 from dual 或不含任何 from 子句

参考文章：<https://www.cnblogs.com/ggjucheng/archive/2012/11/11/2765237.html>

PROFILES

SHOW PROFILES 能够在做 SQL 优化时分析当前会话中语句执行的**资源消耗**情况

- 通过 have_profiling 参数，能够看到当前 MySQL 是否支持 profile：

```
mysql> select @@have_profiling;
+-----+
| @have_profiling |
+-----+
| YES           |
+-----+
1 row in set, 1 warning (0.00 sec)
```

- 默认 profiling 是关闭的，可以通过 set 语句在 Session 级别开启 profiling：

```
mysql> select @@profiling;
+-----+
| @@profiling |
+-----+
|          0 |
+-----+
1 row in set, 1 warning (0.00 sec)
```

SET profiling=1; #开启profiling 开关;

- 执行 SHOW PROFILES 指令，来查看 SQL 语句执行的耗时：

SHOW PROFILES;

```
mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query
+-----+-----+-----+
|     1 | 0.00028600 | show databases
|     2 | 0.00011900 | SELECT DATABASE()
|     3 | 0.00031250 | show tables
|     4 | 0.00007575 | select * from tb_item where id = < 5
|     5 | 0.05521375 | select * from tb_item where id < 5
|     6 | 3.31852475 | select count(*) from tb_item
+-----+-----+-----+
6 rows in set, 1 warning (0.00 sec)
```

- 查看到该 SQL 执行过程中每个线程的状态和消耗的时间：

```
SHOW PROFILE FOR QUERY query_id;
```

```
mysql> show profile for query 6;
+-----+-----+
| Status          | Duration |
+-----+-----+
| starting        | 0.000057 |
| checking permissions | 0.000006 |
| Opening tables   | 0.000020 |
| init             | 0.000016 |
| System lock      | 0.000010 |
| optimizing       | 0.000007 |
| statistics       | 0.000014 |
| preparing         | 0.000015 |
| executing        | 0.000003 |
| Sending data     | 3.318290 |
| end              | 0.000020 |
| query end        | 0.000008 |
| closing tables    | 0.000018 |
| freeing items     | 0.000029 |
| cleaning up       | 0.000014 |
+-----+
15 rows in set, 1 warning (0.03 sec)
```

- 在获取到最消耗时间的线程状态后，MySQL 支持选择 all、cpu、block io、context switch、page faults 等类型查看 MySQL 在使用什么资源上耗费了过高的时间。例如，选择查看 CPU 的耗时：

```
mysql> show profile cpu for query 6;
+-----+-----+-----+-----+
| Status          | Duration | CPU_user | CPU_system |
+-----+-----+-----+-----+
| starting        | 0.000057 | 0.000000 | 0.000000 |
| checking permissions | 0.000006 | 0.000000 | 0.000000 |
| Opening tables   | 0.000020 | 0.000000 | 0.000000 |
| init             | 0.000016 | 0.000000 | 0.000000 |
| System lock      | 0.000010 | 0.000000 | 0.000000 |
| optimizing       | 0.000007 | 0.000000 | 0.000000 |
| statistics       | 0.000014 | 0.000000 | 0.000000 |
| preparing         | 0.000015 | 0.000000 | 0.000000 |
| executing        | 0.000003 | 0.000000 | 0.000000 |
| Sending data     | 3.318290 | 5.842112 | 0.226965 |
| end              | 0.000020 | 0.000000 | 0.000000 |
| query end        | 0.000008 | 0.000000 | 0.000000 |
| closing tables    | 0.000018 | 0.000000 | 0.000000 |
| freeing items     | 0.000029 | 0.000000 | 0.000000 |
| cleaning up       | 0.000014 | 0.000000 | 0.000000 |
+-----+
15 rows in set, 1 warning (0.00 sec)
```

- Status: SQL 语句执行的状态
- Durationsql: 执行过程中每一个步骤的耗时
- CPU_user: 当前用户占有的 CPU
- CPU_system: 系统占有的 CPU

TRACE

MySQL 提供了对 SQL 的跟踪，通过 trace 文件可以查看优化器生成执行计划的过程

- 打开 trace 功能，设置格式为 JSON，并设置 trace 的最大使用内存，避免解析过程中因默认内存过小而不能够完整展示

```
SET optimizer_trace="enabled=on",end_markers_in_json=ON;      -- 会话内有效
SET optimizer_trace_max_mem_size=1000000;
```

- 执行 SQL 语句：

```
SELECT * FROM tb_item WHERE id < 4;
```

- 检查 information_schema.optimizer_trace:

```
SELECT * FROM information_schema.optimizer_trace \G; -- \G代表竖列展示
```

执行信息主要有三个阶段：prepare 阶段、optimize 阶段（成本分析）、execute 阶段（执行）

索引优化

创建索引

索引是数据库优化最重要的手段之一，通过索引通常可以帮助用户解决大多数的 MySQL 的性能优化问题

```
CREATE TABLE `tb_seller` (
  `sellerid` varchar (100),
  `name` varchar (100),
  `nickname` varchar (50),
  `password` varchar (60),
  `status` varchar (1),
  `address` varchar (100),
  `createtime` datetime,
  PRIMARY KEY(`sellerid`)
)ENGINE=INNODB DEFAULT CHARSET=utf8mb4;
INSERT INTO `tb_seller` (`sellerid`, `name`, `nickname`, `password`, `status`,
`address`, `createtime`) values('xiaomi', '小米科技', '小米官方旗舰店',
'e10adc3949ba59abbe56e057f20f883e', '1', '西安市', '2088-01-01 12:00:00');
CREATE INDEX idx_seller_name_sta_addr ON tb_seller(name, status, address); # 联合索引
```

| sellerid | name | nickname | password | status | address | createtime |
|----------|------------|-----------|----------------------------------|--------|---------|---------------------|
| alibaba | 阿里巴巴 | 阿里小店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |
| baidu | 百度科技有限公司 | 百度小店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |
| huawei | 华为科技有限公司 | 华为小店 | e10adc3949ba59abbe56e057f20f883e | 0 | 北京市 | 2088-01-01 12:00:00 |
| luoji | 罗技科技有限公司 | 罗技小店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |
| oppo | OPPO科技有限公司 | OPPO官方旗舰店 | e10adc3949ba59abbe56e057f20f883e | 0 | 北京市 | 2088-01-01 12:00:00 |
| ourpalm | 掌趣科技股份有限公司 | 掌趣小店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |
| qiandu | 千度科技 | 千度小店 | e10adc3949ba59abbe56e057f20f883e | 2 | 北京市 | 2088-01-01 12:00:00 |
| sina | 新浪科技有限公司 | 新浪官方旗舰店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |
| xiaomi | 小米科技 | 小米官方旗舰店 | e10adc3949ba59abbe56e057f20f883e | 1 | 西安市 | 2088-01-01 12:00:00 |
| yijia | 宜家家居 | 宜家家居旗舰店 | e10adc3949ba59abbe56e057f20f883e | 1 | 北京市 | 2088-01-01 12:00:00 |

10 rows in set (0.00 sec)

避免失效

语句错误

- 全值匹配：对索引中所有列都指定具体值，这种情况索引生效，执行效率高

```
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
+----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr |
+----+-----+-----+-----+-----+-----+
key          | key_len | ref
idx_seller_name_sta_addr | 813   | const,const,const
+----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 最左前缀法则：**联合索引遵守最左前缀法则

匹配最左前缀法则，走索引：

```
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技';
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status='1';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status='1';
+----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr |
+----+-----+-----+-----+-----+-----+
key          | key_len | ref
idx_seller_name_sta_addr | 410   | const,const
+----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

违法最左前缀法则，索引失效：

```
EXPLAIN SELECT * FROM tb_seller WHERE status='1';
EXPLAIN SELECT * FROM tb_seller WHERE status='1' AND address='西安市';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE status='1';
+----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | NULL |
+----+-----+-----+-----+-----+-----+
key          | key_len | ref
NULL        | NULL    | NULL
+----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

如果符合最左法则，但是出现跳跃某一列，只有最左列索引生效：

```
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND address='西安市';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND address='西安市';
+----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr |
+----+-----+-----+-----+-----+-----+
key          | key_len | ref
idx_seller_name_sta_addr | 403   | const
+----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

虽然索引列失效，但是系统会**使用了索引下推进行了优化**

- 范围查询右边的列，不能使用索引：**

```
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status>'1' AND address='西安市';
```

根据前面的两个字段 name，status 查询是走索引的，但是最后一个条件 address 没有用到索引，使用了索引下推

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status>'1' AND address='西安市';
+----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | range | idx_seller_name_sta_addr |
+----+-----+-----+-----+-----+-----+
key          | key_len | ref
idx_seller_name_sta_addr | 410   | NULL
+----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 在索引列上函数或者运算 (+ - 数值) 操作，索引将失效：会破坏索引值的有序性**

```
EXPLAIN SELECT * FROM tb_seller WHERE SUBSTRING(name,3,2) = '科技';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE SUBSTRING(name,3,2) = '科技';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | NULL | NULL | NULL | NULL | 10 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 字符串不加单引号，造成索引失效：隐式类型转换，当字符串和数字比较时会把字符串转化为数字

没有对字符串加单引号，查询优化器会调用 CAST 函数将 status 转换为 int 进行比较，造成索引失效

```
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status = 1;
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' AND status=1;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr | idx_seller_name_sta_addr | 403 | const | 1 | 10.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 2 warnings (0.07 sec)
```

如果 status 是 int 类型，SQL 为 `SELECT * FROM tb_seller WHERE status = '1'` 并不会造成索引失效，因为会将 `'1'` 转换为 `1`，**并不会对索引列产生操作**

- 多表连接查询时，如果两张表的**字符集不同**，会造成索引失效，因为会进行类型转换

解决方法：CONVERT 函数是加在输入参数上、修改表的字符集

- 用 OR 分割条件，索引失效**，导致全表查询：

OR 前的条件中的列有索引而后面的列中没有索引或 OR 前后两个列是同一个复合索引，都造成索引失效

```
EXPLAIN SELECT * FROM tb_seller WHERE name='阿里巴巴' OR createtime = '2088-01-01 12:00:00';
EXPLAIN SELECT * FROM tb_seller WHERE name='小米科技' OR status='1';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE NAME='阿里巴巴' OR createtime = '2088-01-01 12:00:00';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | idx_seller_name_sta_addr | NULL | NULL | NULL | 10 | 19.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

AND 分割的条件不影响：

```
EXPLAIN SELECT * FROM tb_seller WHERE name='阿里巴巴' AND createtime = '2088-01-01 12:00:00';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name='阿里巴巴' AND createtime = '2088-01-01 12:00:00';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr | idx_seller_name_sta_addr | 403 | const | 1 | 10.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 以 % 开头的 LIKE 模糊查询，索引失效：

如果是尾部模糊匹配，索引不会失效；如果是头部模糊匹配，索引失效

```
EXPLAIN SELECT * FROM tb_seller WHERE name like '%科技%';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name like '%科技%';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | NULL | NULL | NULL | NULL | 10 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM tb_seller WHERE name like '科技%';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | range | idx_seller_name_sta_addr | idx_seller_name_sta_addr | 403 | NULL | 1 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.04 sec)
```

解决方案：通过覆盖索引来解决

```
EXPLAIN SELECT sellerid, name, status FROM tb_seller WHERE name like '%科技';
```

```
mysql> EXPLAIN SELECT sellerid, name, status FROM tb_seller WHERE name like '%科技';
+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | index | NULL |
+-----+-----+-----+-----+-----+
key | key_len | ref | rows | filtered | Extra |
idx_seller_name_sta_addr | 813 | NULL | 10 | 11.11 | Using where; Using index |
+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

原因：在覆盖索引的这棵 B+ 树上只需要进行 like 的匹配，或者是基于覆盖索引查询再进行 WHERE 的判断就可以获得结果

系统优化

系统优化为全表扫描：

- 如果 MySQL 评估使用索引比全表更慢，则不使用索引，索引失效：

```
CREATE INDEX idx_address ON tb_seller(address);
EXPLAIN SELECT * FROM tb_seller WHERE address='西安市';
EXPLAIN SELECT * FROM tb_seller WHERE address='北京市';
```

北京市的键值占 9/10 (区分度低)，所以优化为全表扫描，type = ALL

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE address='西安市';
+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_address |
+-----+-----+-----+-----+-----+
key | key_len | ref | rows | filtered | Extra |
idx_address | 403 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM tb_seller WHERE address='北京市';
+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | idx_address |
+-----+-----+-----+-----+-----+
key | key_len | ref | rows | filtered | Extra |
NULL | NULL | NULL | 10 | 90.00 | Using where |
+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- IS NULL、IS NOT NULL 有时索引失效：

```
EXPLAIN SELECT * FROM tb_seller WHERE name IS NULL;
EXPLAIN SELECT * FROM tb_seller WHERE name IS NOT NULL;
```

NOT NULL 失效的原因是 name 列全部不是 null，优化为全表扫描，当 NULL 过多时，IS NULL 失效

```
mysql> EXPLAIN SELECT * FROM tb_seller WHERE name IS NULL;
+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr |
+-----+-----+-----+-----+-----+
key | key_len | ref | rows | filtered | Extra |
idx_seller_name_sta_addr | 403 | const | 1 | 100.00 | Using index condition |
+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

ERROR:
No query specified

mysql> EXPLAIN SELECT * FROM tb_seller WHERE name IS NOT NULL;
+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys |
+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ALL | idx_seller_name_sta_addr |
+-----+-----+-----+-----+-----+
key | key_len | ref | rows | filtered | Extra |
NULL | NULL | NULL | 10 | 100.00 | Using where |
+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

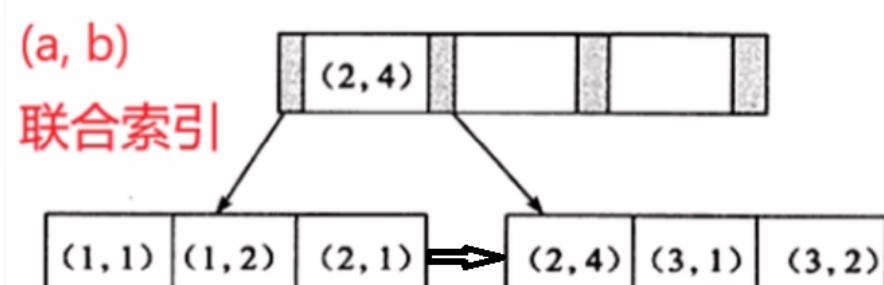
- IN 肯定会走索引，但是当 IN 的取值范围较大时会导致索引失效，走全表扫描：

```
EXPLAIN SELECT * FROM tb_seller WHERE sellerId IN ('alibaba', 'huawei'); -- 都走索引
EXPLAIN SELECT * FROM tb_seller WHERE sellerId NOT IN ('alibaba', 'huawei');
```

- [MySQL 实战 45 讲](#)该章节最后提出了一种慢查询场景，获取到数据以后 Server 层还会做判断

底层原理

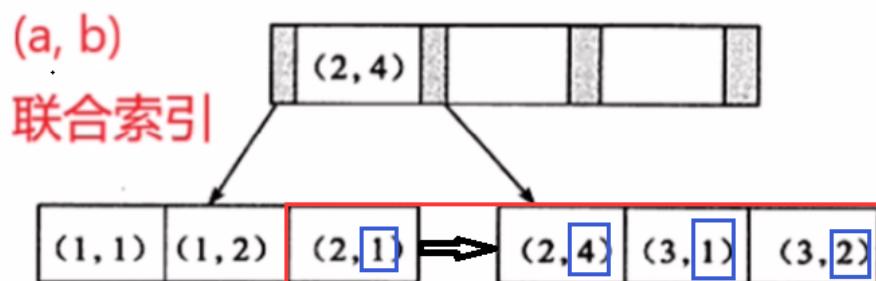
索引失效一般是针对联合索引，联合索引一般由几个字段组成，排序方式是先按照第一个字段进行排序，然后排序第二个，依此类推，图示 (a, b) 索引，**a 相等的情况下 b 是有序的**



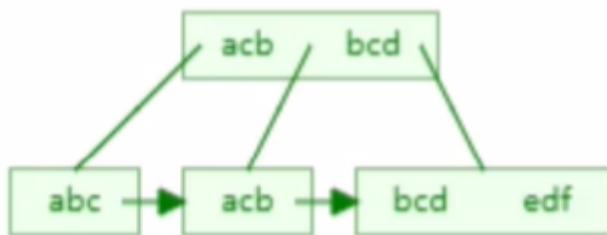
a 的值是有顺序的： 1, 1, 2, 2, 3, 3

b 的值是没有顺序的： 1, 2, 1, 4, 1, 2

- 最左前缀法则：当不匹配前面的字段的时候，后面的字段都是无序的。这种无序不仅体现在叶子节点，也会导致查询时扫描的非叶子节点也是无序的，因为索引树相当于忽略的第一个字段，就无法使用二分查找
- 范围查询右边的列，不能使用索引，比如语句：`WHERE a > 1 AND b = 1`，在 a 大于 1 的时候，b 是无序的，a > 1 是扫描时有序的，但是找到以后进行寻找 b 时，索引树就不是有序的了



- 以 % 开头的 LIKE 模糊查询，索引失效，比如语句：`WHERE a LIKE '%d'`，前面的不确定，导致不符合最左匹配，直接去索引中搜索以 d 结尾的节点，所以没有顺序



查看索引

```
SHOW STATUS LIKE 'Handler_read%';
SHOW GLOBAL STATUS LIKE 'Handler_read%';
```

```
mysql> SHOW STATUS LIKE 'Handler_read%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| Handler_read_first | 2      |
| Handler_read_key   | 14     |
| Handler_read_last  | 0      |
| Handler_read_next  | 10     |
| Handler_read_prev  | 0      |
| Handler_read_rnd   | 0      |
| Handler_read_rnd_next| 35    |
+-----+-----+
7 rows in set (0.42 sec)
```

- Handler_read_first: 索引中第一条被读的次数, 如果较高, 表示服务器正执行大量全索引扫描 (这个值越低越好)
- Handler_read_key: 如果索引正在工作, 这个值代表一个行被索引值读的次数, 值越低表示索引不经常使用 (这个值越高越好)
- Handler_read_next: 按照键顺序读下一行的请求数, 如果范围约束或执行索引扫描来查询索引列, 值增加
- Handler_read_prev: 按照键顺序读前一行的请求数, 该读方法主要用于优化 ORDER BY ... DESC
- Handler_read_rnd: 根据固定位置读一行的请求数, 如果执行大量查询并对结果进行排序则该值较高, 可能是使用了大量需要 MySQL 扫描整个表的查询或连接, 这个值较高意味着运行效率低, 应该建立索引来解决
- Handler_read_rnd_next: 在数据文件中读下一行的请求数, 如果正进行大量的表扫描, 该值较高, 说明表索引不正确或写入的查询没有利用索引

SQL 优化

自增主键

自增机制

自增主键可以让主键索引尽量地保持在数据页中递增顺序插入, 不自增需要寻找其他页插入, 导致随机 IO 和页分裂的情况

表的结构定义存放在后缀名为.frm 的文件中, 但是并不会保存自增值, 不同的引擎对于自增值的保存策略不同:

- MyISAM 引擎的自增值保存在数据文件中
- InnoDB 引擎的自增值保存在了内存里, 每次打开表都会去找自增值的最大值 max(id), 然后将 max(id)+1 作为当前的自增值; 8.0 版本后, 才有了自增值持久化的能力, 将自增值的变更记录在了 redo log 中, 重启的时候依靠 redo log 恢复重启之前的值

在插入一行数据的时候，自增值的行为如下：

- 如果插入数据时 id 字段指定为 0、null 或未指定值，那么把这个表当前的 AUTO_INCREMENT 值填到自增字段
- 如果插入数据时 id 字段指定了具体的值，比如某次要插入的值是 X，当前的自增值是 Y
 - 如果 $X < Y$ ，那么这个表的自增值不变
 - 如果 $X \geq Y$ ，就需要把当前自增值修改为新的自增值

参数说明：auto_increment_offset 和 auto_increment_increment 分别表示自增的初始值和步长，默认值都是 1

语句执行失败也不回退自增 id，所以保证了自增 id 是递增的，但不保证是连续的（不能回退，所以有些回滚事务的自增 id 就不会重新使用，导致出现不连续）

自增 ID

MySQL 不同的自增 id 在达到上限后的表现不同：

- 表的自增 id 如果是 int 类型，达到上限 $2^{32}-1$ 后，再申请时值就不会改变，进而导致继续插入数据时报主键冲突的错误
- row_id 长度为 6 个字节，达到上限后则会归 0 再重新递增，如果出现相同的 row_id，后写的数据会覆盖之前的数据，造成旧数据丢失，影响的是数据可靠性，所以应该在 InnoDB 表中主动创建自增主键报主键冲突，插入失败影响的是可用性，而一般情况下，**可靠性优先于可用性**
- Xid 长度 8 字节，由 Server 层维护，只需要不在同一个 binlog 文件中出现重复值即可，虽然理论上会出现重复值，但是概率极小
- InnoDB 的 max_trx_id 递增值每次 MySQL 重启都会被保存起来，重启也不会重置为 0，所以会导致一直增加到达上限，然后从 0 开始，这时原事务 0 修改的数据对当前事务就是可见的，产生脏读的现象

只读事务不分配 trx_id，所以 trx_id 的增加速度变慢了

- thread_id 长度 4 个字节，到达上限后就会重置为 0，MySQL 设计了一个唯一数组的逻辑，给新线程分配 thread_id 时做判断，保证不会出现两个相同的 thread_id：

```
do {
    new_id = thread_id_counter++;
} while (!thread_ids.insert_unique(new_id).second);
```

参考文章：<https://time.geekbang.org/column/article/83183>

覆盖索引

复合索引叶子节点不仅保存了复合索引的值，还有主键索引，所以使用覆盖索引的时候，加上主键也会用到索引

尽量使用覆盖索引，避免 SELECT *：

```
EXPLAIN SELECT name,status,address FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
```

```
mysql> EXPLAIN SELECT name,status,address FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
+----+-----+-----+-----+
| id | select_type | table | partitions |
+----+-----+-----+-----+
| 1 | SIMPLE    | tb_seller | NULL      |
+----+-----+-----+-----+
| ref | idx_seller_name_sta_addr | 813 | const,const,const |
+----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

如果查询列，超出索引列，也会降低性能：

```
EXPLAIN SELECT name,status,address,password FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
```

```
mysql> EXPLAIN SELECT name,status,address,password FROM tb_seller WHERE name='小米科技' AND status='1' AND address='西安市';
+----+-----+-----+-----+
| id | select_type | table | partitions |
+----+-----+-----+-----+
| 1 | SIMPLE    | tb_seller | NULL      |
+----+-----+-----+-----+
| ref | idx_seller_name_sta_addr,`idx_address` | 813 | const,const,const |
+----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

减少访问

避免对数据进行重复检索：能够一次连接就获取到结果的，就不用两次连接，这样可以大大减少对数据库无用的重复请求

- 查询数据：

```
SELECT id,name FROM tb_book;
SELECT id,status FROM tb_book; -- 向数据库提交两次请求，数据库就要做两次查询操作
-- > 优化为：
SELECT id,name,status FROM tb_book;
```

- 插入数据：

```
INSERT INTO tb_test VALUES(1,'Tom');
INSERT INTO tb_test VALUES(2,'Cat');
INSERT INTO tb_test VALUES(3,'Jerry'); -- 连接三次数据库
-- > 优化为
INSERT INTO tb_test VALUES(1,'Tom'),(2,'Cat'),(3,'Jerry'); -- 连接一次
```

- 在事务中进行数据插入：

```
start transaction;
INSERT INTO tb_test VALUES(1,'Tom');
INSERT INTO tb_test VALUES(2,'Cat');
INSERT INTO tb_test VALUES(3,'Jerry');
commit; -- 手动提交，分段提交
```

- 数据有序插入：

```
INSERT INTO tb_test VALUES(1, 'Tom');
INSERT INTO tb_test VALUES(2, 'Cat');
INSERT INTO tb_test VALUES(3, 'Jerry');
```

增加 cache 层：在应用中增加缓存层来达到减轻数据库负担的目的。可以部分数据从数据库中抽取出来放到应用端以文本方式存储，或者使用框架（Mybatis）提供的一级缓存 / 二级缓存，或者使用 Redis 数据库来缓存数据

数据插入

当使用 load 命令导入数据的时候，适当的设置可以提高导入的效率：

| | id | username | password | name | birthday |
|----|--------------|-----------------|-----------------|---------------------|-----------------|
| 1 | "username1" | "ZPAIFXVC" | "name1" | 0000-00-00 00:00:00 | |
| 2 | "username2" | "KPTMIQRZ" | "name2" | 0000-00-00 00:00:00 | |
| 3 | "username3" | "SUIPRGF1" | "name3" | 0000-00-00 00:00:00 | |
| 4 | "username4" | "WRXCPPDN" | "name4" | 0000-00-00 00:00:00 | |
| 5 | "username5" | "CHRJYQUT" | "name5" | 0000-00-00 00:00:00 | |
| 6 | "username6" | "XSOBMUKH" | "name6" | 0000-00-00 00:00:00 | |
| 7 | "username7" | "HIFUXXYH" | "name7" | 0000-00-00 00:00:00 | |
| 8 | "username8" | "DXRJCXZH" | "name8" | 0000-00-00 00:00:00 | |
| 9 | "username9" | "PHEAJVLH" | "name9" | 0000-00-00 00:00:00 | |
| 10 | "username10" | "HNUSCHZJ" | "name10" | 0000-00-00 00:00:00 | |

```
LOAD DATA LOCAL INFILE = '/home/seazean/sql1.log' INTO TABLE `tb_user_1` FIELD TERMINATED BY ',' LINES TERMINATED BY '\n'; -- 文件格式如上图
```

对于 InnoDB 类型的表，有以下几种方式可以提高导入的效率：

1. **主键顺序插入**：因为 InnoDB 类型的表是按照主键的顺序保存的，所以将导入的数据按照主键的顺序排列，可以有效的提高导入数据的效率，如果 InnoDB 表没有主键，那么系统会自动默认创建一个内部列作为主键

主键是否连续对性能影响不大，只要是递增的就可以，比如雪花算法产生的 ID 不是连续的，但是是递增的，因为递增可以让主键索引尽量地保持顺序插入，**避免了页分裂**，因此索引更紧凑

- 插入 ID 顺序排列数据：

```
mysql> load data local infile '/root/sql1.log' into table `tb_user` fields terminated by ',' lines terminated by '\n';
Query OK, 1000000 rows affected, 65535 warnings (20.58 sec)
Records: 1000000 Deleted: 0 Skipped: 0 Warnings: 4000000

mysql> select count(*) from tb_user;
+-----+
| count(*) |
+-----+
| 1000000 |
+-----+
1 row in set (0.40 sec)
```

- 插入 ID 无序排列数据：

```
mysql> load data local infile '/root/sql2.log' into table `tb_user` fields terminated by ',' lines terminated by '\n';
Query OK, 1000000 rows affected, 65535 warnings (1 min 59.29 sec)
Records: 1000000 Deleted: 0 Skipped: 0 Warnings: 4000000

mysql> select count(*) from tb_user;
+-----+
| count(*) |
+-----+
| 1000000 |
+-----+
1 row in set (0.27 sec)
```

2. **关闭唯一性校验**：在导入数据前执行 `SET UNIQUE_CHECKS=0`，关闭唯一性校验；导入结束后执行 `SET UNIQUE_CHECKS=1`，恢复唯一性校验，可以提高导入的效率。

```

mysql>
mysql>
mysql> SET UNIQUE_CHECKS=0;
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql>
mysql> load data local infile '/root/sql1.log' into table `tb_user` fields terminated by ',' lines terminated by '\n';
Query OK, 1000000 rows affected, 65535 warnings (19.39 sec)
Records: 1000000 Deleted: 0 Skipped: 0 Warnings: 4000000

mysql> SET UNIQUE_CHECKS=1;
Query OK, 0 rows affected (0.00 sec)

```

- 3. **手动提交事务**: 如果应用使用自动提交的方式, 建议在导入前执行 `SET AUTOCOMMIT=0`, 关闭自动提交; 导入结束后再打开自动提交, 可以提高导入的效率。

事务需要控制大小, 事务太大可能会影响执行的效率。MySQL 有 `innodb_log_buffer_size` 配置项, 超过这个值的日志会写入磁盘数据, 效率会下降, 所以在事务大小达到配置项数据级前进行事务提交可以提高效率

```

mysql> SET AUTOCOMMIT=0;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql>
mysql> load data local infile '/root/sql1.log' into table `tb_user` fields terminated by ',' lines terminated by '\n';
Query OK, 1000000 rows affected, 65535 warnings (19.58 sec)
Records: 1000000 Deleted: 0 Skipped: 0 Warnings: 4000000

mysql> select count(*) from tb_user;
+-----+
| count(*) |
+-----+
| 1000000 |
+-----+
1 row in set (0.30 sec)

```

分组排序

ORDER

数据准备:

```

CREATE TABLE `emp` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(100) NOT NULL,
  `age` INT(3) NOT NULL,
  `salary` INT(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8mb4;
INSERT INTO `emp`(`id`, `name`, `age`, `salary`)
VALUES('1','Tom','25','2300');-- ...
CREATE INDEX idx_emp_age_salary ON emp(age, salary);

```

- 第一种是通过对返回数据进行排序, 所有不通过索引直接返回结果的排序都叫 FileSort 排序, 会在内存中重新排序

```
EXPLAIN SELECT * FROM emp ORDER BY age DESC; -- 年龄降序
```

```

mysql> EXPLAIN SELECT * FROM emp ORDER BY age DESC;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | emp | NULL | ALL | NULL | NULL | NULL | NULL | 12 | 100.00 | Using filesort |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)

```

- 第二种通过有序索引顺序扫描直接返回**有序数据**, 这种情况为 `Using index`, 不需要额外排序, 操作效率高

```
EXPLAIN SELECT id, age, salary FROM emp ORDER BY age DESC;
```

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|----|-------------|-------|------------|-------|---------------|--------------------|---------|------|------|----------|-------------|
| 1 | SIMPLE | emp | NULL | index | NULL | idx_emp_age_salary | 9 | NULL | 12 | 100.00 | Using index |

- 多字段排序:

```
EXPLAIN SELECT id,age,salary FROM emp ORDER BY age DESC, salary DESC;  
EXPLAIN SELECT id,age,salary FROM emp ORDER BY salary DESC, age DESC;  
EXPLAIN SELECT id,age,salary FROM emp ORDER BY age DESC, salary ASC;
```

| mysql> EXPLAIN SELECT id,age,salary FROM emp ORDER BY age DESC, salary DESC; | | | | | | | | | | | |
|--|-------------|-------|------------|-------|---------------|--------------------|---------|------|------|----------|-------------|
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
| 1 | SIMPLE | emp | NULL | index | NULL | idx_emp_age_salary | 9 | NULL | 12 | 100.00 | Using index |

| mysql> EXPLAIN SELECT id,age,salary FROM emp ORDER BY salary DESC, age DESC; | | | | | | | | | | | |
|--|-------------|-------|------------|-------|---------------|--------------------|---------|------|------|----------|-----------------------------|
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
| 1 | SIMPLE | emp | NULL | index | NULL | idx_emp_age_salary | 9 | NULL | 12 | 100.00 | Using index; Using filesort |

| mysql> EXPLAIN SELECT id,age,salary FROM emp ORDER BY age DESC, salary ASC; | | | | | | | | | | | |
|---|-------------|-------|------------|-------|---------------|--------------------|---------|------|------|----------|-----------------------------|
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
| 1 | SIMPLE | emp | NULL | index | NULL | idx_emp_age_salary | 9 | NULL | 12 | 100.00 | Using index; Using filesort |

尽量减少额外的排序，通过索引直接返回有序数据。需要满足 Order by 使用相同的索引、Order By 的顺序和索引顺序相同、Order by 的字段都是升序或都是降序，否则需要额外的操作，就会出现 FileSort

- ORDER BY RAND() 命令用来进行随机排序，会使用了临时内存表，临时内存表排序的时使用 rowid 排序方法

优化方式：创建合适的索引能够减少 Filesort 的出现，但是某些情况下条件限制不能让 Filesort 消失，就要加快 Filesort 的排序操作

内存临时表，MySQL 有两种 Filesort 排序算法：

- rowid 排序：首先根据条件取出排序字段和信息，然后在排序区 sort buffer (Server 层) 中排序，如果 sort buffer 不够，则在临时表 temporary table 中存储排序结果。完成排序后再根据行指针回表读取记录，该操作可能会导致大量随机 I/O 操作

说明：对于临时内存表，回表过程只是简单地根据数据行的位置，直接访问内存得到数据，不会导致多访问磁盘，优先选择该方式

- 全字段排序：一次性取出满足条件的所有数据，需要回表，然后在排序区 sort buffer 中排序后直接输出结果集。排序时内存开销较大，但是排序效率比两次扫描算法高

具体的选择方式：

- MySQL 通过比较系统变量 max_length_for_sort_data 的大小和 Query 语句取出的字段的大小，来判定使用哪种排序算法。如果前者大，则说明 sort buffer 空间足够，使用第二种优化之后的算法，否则使用第一种。
- 可以适当提高 sort_buffer_size 和 max_length_for_sort_data 系统变量，来增大排序区的大小，提高排序的效率

```
SET @@max_length_for_sort_data = 10000;          -- 设置全局变量  
SET max_length_for_sort_data = 10240;            -- 设置会话变量  
SHOW VARIABLES LIKE 'max_length_for_sort_data';   -- 默认1024  
SHOW VARIABLES LIKE 'sort_buffer_size';           -- 默认262114
```

GROUP

GROUP BY 也会进行排序操作，与 ORDER BY 相比，GROUP BY 主要只是多了排序之后的分组操作，所以在 GROUP BY 的实现过程中，与 ORDER BY 一样也可以利用到索引

- 分组查询：

```
DROP INDEX idx_emp_age_salary ON emp;
EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age;
```

```
mysql> EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | emp   | NULL      | ALL  | NULL          | NULL | NULL    | NULL | 12 | 100.00 | Using temporary; Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.04 sec)
```

Using temporary：表示 MySQL 需要使用临时表（不是 sort buffer）来存储结果集，常见于排序和分组查询

- 查询包含 GROUP BY 但是用户想要避免排序结果的消耗，则可以执行 ORDER BY NULL 禁止排序：

```
EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age ORDER BY NULL;
```

```
mysql> EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age ORDER BY NULL;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | emp   | NULL      | ALL  | NULL          | NULL | NULL    | NULL | 12 | 100.00 | Using temporary |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 创建索引：索引本身有序，不需要临时表，也不需要再额外排序

```
CREATE INDEX idx_emp_age_salary ON emp(age, salary);
```

```
mysql> EXPLAIN SELECT age,COUNT(*) FROM emp GROUP BY age ORDER BY NULL;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | emp   | NULL      | index | idx_emp_age_salary | idx_emp_age_salary | 9 | NULL | 12 | 100.00 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- 数据量很大时，使用 SQL_BIG_RESULT 提示优化器直接使用直接用磁盘临时表

联合查询

对于包含 OR 的查询子句，如果要利用索引，则 OR 之间的每个条件列都必须用到索引，而且不能使用到条件之间的复合索引，如果没有索引，则应该考虑增加索引

- 执行查询语句：

```
EXPLAIN SELECT * FROM emp WHERE id = 1 OR age = 30; -- 两个索引，并且不是复合索引
```

```
mysql> mysql> EXPLAIN SELECT * FROM emp WHERE id = 1 OR age = 30;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type      | possible_keys | key           | key_len | ref   | rows  | filtered | Extra          |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE     | emp   | NULL       | index_merge | PRIMARY, idx_emp_age_salary | idx_emp_age_salary.PRIMARY | 4,4    | NULL  | 2     | 100.00  | Using sort_union(idx_emp_age_salary,PRIMARY); Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Extra: Using sort_union(idx_emp_age_salary, PRIMARY); Using where

- 使用 UNION 替换 OR, 求并集:

注意：该优化只针对多个索引列有效，如果有列没有被索引，查询效率可能会因为没有选择 OR 而降低

```
EXPLAIN SELECT * FROM emp WHERE id = 1 UNION SELECT * FROM emp WHERE age = 30;
```

- UNION 要优于 OR 的原因:

- UNION 语句的 type 值为 ref, OR 语句的 type 值为 range
 - UNION 语句的 ref 值为 const, OR 语句的 ref 值为 null, const 表示是常量值引用, 非常快

嵌套查询

MySQL 4.1 版本之后，开始支持 SQL 的子查询

- 可以使用 SELECT 语句来创建一个单列的查询结果，然后把结果作为过滤条件用在另一个查询中
 - 使用子查询可以一次性的完成逻辑上需要多个步骤才能完成的 SQL 操作，同时也可以避免事务或者表锁死
 - 在有些情况下，子查询是可以被更高效的连接（JOIN）替代

例如查找有角色的所有的用户信息：

- 执行计划：

```
EXPLAIN SELECT * FROM t_user WHERE id IN (SELECT user_id FROM user_role);
```

- 优化后:

```
EXPLAIN SELECT * FROM t_user u , user_role ur WHERE u.id = ur.user_id;
```

```
mysql> EXPLAIN SELECT * FROM t_user u , user_role ur WHERE u.id = ur.user_id;
+---+ | id | select_type | table | partitions | type | possible_keys | key | key_len | ref | | rows | filtered | Extra | +---+
| 1 | SIMPLE | ur | NULL | ALL | fk_ur_user_id | NULL | NULL | NULL | 6 | 100.00 | Using where |
| 1 | SIMPLE | u | NULL | eq_ref | PRIMARY | PRIMARY | 98 | db2.ur.user_id | 1 | 100.00 | NULL |
+---+
2 rows in set, 1 warning (0.00 sec)
```

连接查询之所以效率更高，是因为**不需要在内存中创建临时表**来完成逻辑上需要两个步骤的查询工作

分页查询

一般分页查询时，通过创建覆盖索引能够比较好地提高性能

一个常见的问题是 `LIMIT 200000,10`，此时需要 MySQL 扫描前 200010 记录，仅仅返回 200000 - 200010 之间的记录，其他记录丢弃，查询排序的代价非常大

- 分页查询:

```
EXPLAIN SELECT * FROM tb_user_1 LIMIT 200000,10;
```

- 优化方式一：内连接查询，在索引列 id 上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容

```
EXPLAIN SELECT * FROM tb_user_1 t,(SELECT id FROM tb_user_1 ORDER BY id
LIMIT 200000,10) a WHERE t.id = a.id;
```

- 优化方式二：方案适用于主键自增的表，可以把 LIMIT 查询转换成某个位置的查询

```
EXPLAIN SELECT * FROM tb_user_1 WHERE id > 200000 LIMIT 10; -- 写法 1  
EXPLAIN SELECT * FROM tb_user_1 WHERE id BETWEEN 200000 and 200010; -- 写法 2
```

使用提示

SQL 提示，是优化数据库的一个重要手段，就是在 SQL 语句中加入一些提示来达到优化操作的目的

- USE INDEX：在查询语句中表名的后面添加 USE INDEX 来提供 MySQL 去参考的索引列表，可以让 MySQL 不再考虑其他可用的索引

```
CREATE INDEX idx_seller_name ON tb_seller(name);
EXPLAIN SELECT * FROM tb_seller USE INDEX(idx_seller_name) WHERE name='小米科技';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller USE INDEX(idx_seller_name) WHERE name='小米科技';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name | idx_seller_name | 403 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

- IGNORE INDEX：让 MySQL 忽略一个或者多个索引，则可以使用 IGNORE INDEX 作为提示

```
EXPLAIN SELECT * FROM tb_seller IGNORE INDEX(idx_seller_name) WHERE name = '小米科技';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller IGNORE INDEX(idx_seller_name) WHERE name = '小米科技';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr | idx_seller_name_sta_addr | 403 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

- FORCE INDEX：强制 MySQL 使用一个特定的索引

```
EXPLAIN SELECT * FROM tb_seller FORCE INDEX(idx_seller_name_sta_addr) WHERE NAME='小米科技';
```

```
mysql> EXPLAIN SELECT * FROM tb_seller FORCE INDEX(idx_seller_name_sta_addr) WHERE NAME='小米科技';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_seller | NULL | ref | idx_seller_name_sta_addr | idx_seller_name_sta_addr | 403 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.04 sec)
```

统计计数

在不同的 MySQL 引擎中，count(*) 有不同的实现方式：

- MyISAM 引擎把一个表的总行数存在了磁盘上，因此执行 count(*) 的时候会直接返回这个数，效率很高，但不支持事务
- show table status 命令通过采样估算可以快速获取，但是不准确
- InnoDB 表执行 count(*) 会遍历全表，虽然结果准确，但会导致性能问题

解决方案：

- 计数保存在 Redis 中，但是更新 MySQL 和 Redis 的操作不是原子的，会存在数据一致性的问题
- 计数直接放到数据库里单独的一张计数表中，利用事务解决计数精确问题：

| 时刻 | 会话A | 会话B |
|----|---------------------|---|
| T1 | | |
| T2 | begin; 表C中计数值加1; | |
| T3 | | begin; 读表C计数值; 查询最近100条记录; commit; |
| T4 | 插入一行数据R commit; | |

会话 B 的读操作在 T3 执行的，这时更新事务还没有提交，所以计数值加 1 这个操作对会话 B 还不可见，因此会话 B 查询的计数值和最近 100 条记录，返回的结果逻辑上就是一致的

并发系统性能的角度考虑，应该先插入操作记录再更新计数表，因为更新计数表涉及到行锁的竞争，**先插入再更新能最大程度地减少事务之间的锁等待，提升并发度**

count 函数的按照效率排序：`count(字段) < count(主键id) < count(1) ≈ count(*)`，所以建议尽量使用 `count(*)`

- `count(主键 id)`: InnoDB 引擎会遍历整张表，把每一行的 id 值都取出来返回给 Server 层，Server 判断 id 不为空就按行累加
- `count(1)`: InnoDB 引擎遍历整张表但不取值，Server 层对于返回的每一行，放一个数字 1 进去，判断不为空就按行累加
- `count(字段)`: 如果这个字段是定义为 not null 的话，一行行地从记录里面读出这个字段，判断不能为 null，按行累加；如果这个字段定义允许为 null，那么执行的时候，判断到有可能是 null，还要把值取出来再判断一下，不是 null 才累加
- `count(*)`: 不取值，按行累加

参考文章：<https://time.geekbang.org/column/article/72775>

缓冲优化

优化原则

三个原则：

- 将尽量多的内存分配给 MySQL 做缓存，但也要给操作系统和其他程序预留足够内存
- MyISAM 存储引擎的数据文件读取依赖于操作系统自身的 IO 缓存，如果有 MyISAM 表，就要预留更多的内存给操作系统做 IO 缓存
- 排序区、连接区等缓存是分配给每个数据库会话（Session）专用的，值的设置要根据最大连接数合理分配，如果设置太大，不但浪费资源，而且在并发数较高时会导致物理内存耗尽

缓冲内存

Buffer Pool 本质上是 InnoDB 向操作系统申请的一段连续的内存空间。InnoDB 的数据是按数据页为单位来读写，每个数据页的大小默认是 16KB。数据是存放在磁盘中，每次读写数据都需要进行磁盘 IO 将数据读入内存进行操作，效率会很低，所以提供了 Buffer Pool 来暂存这些数据页，缓存中的这些页又叫缓冲页

工作原理：

- 从数据库读取数据时，会首先从缓存中读取，如果缓存中没有，则从磁盘读取后放入 Buffer Pool
- 向数据库写入数据时，会写入缓存，缓存中修改的数据会定期刷新到磁盘，这一过程称为刷脏

Buffer Pool 中每个缓冲页都有对应的控制信息，包括表空间编号、页号、偏移量、链表信息等，控制信息存放在占用的内存称为控制块，控制块与缓冲页是一一对应的，但并不是物理上相连的，都在缓冲池中

MySQL 提供了缓冲页的快速查找方式：**哈希表**，使用表空间号和页号作为 Key，缓冲页控制块的地址作为 Value 创建一个哈希表，获取数据页时根据 Key 进行哈希寻址：

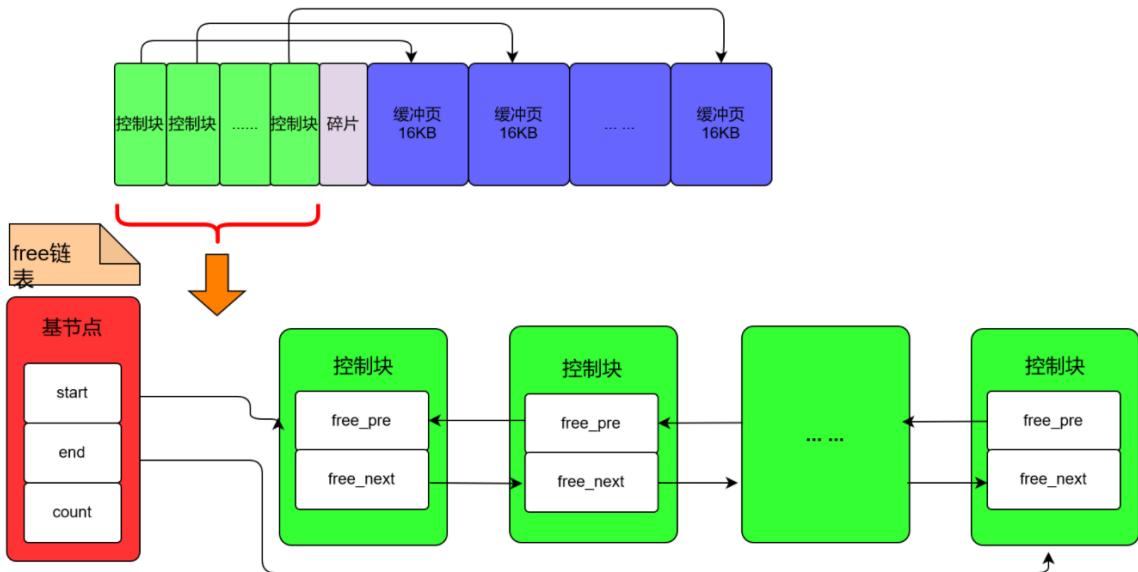
- 如果不存在对应的缓存页，就从 free 链表中选一个空闲缓冲页，把磁盘中的对应页加载到该位置
- 如果存在对应的缓存页，直接获取使用，提高查询数据的效率

当内存数据页跟磁盘数据页内容不一致时，称这个内存页为脏页；内存数据写入磁盘后，内存和磁盘上的数据页一致，称为干净页

内存管理

Free 链表

MySQL 启动时完成对 Buffer Pool 的初始化，先向操作系统申请连续的内存空间，然后将内存划分为若干对控制块和缓冲页。为了区分空闲和已占用的数据页，将所有空闲缓冲页对应的**控制块作为一个节点**放入一个链表中，就是 Free 链表（**空闲链表**）



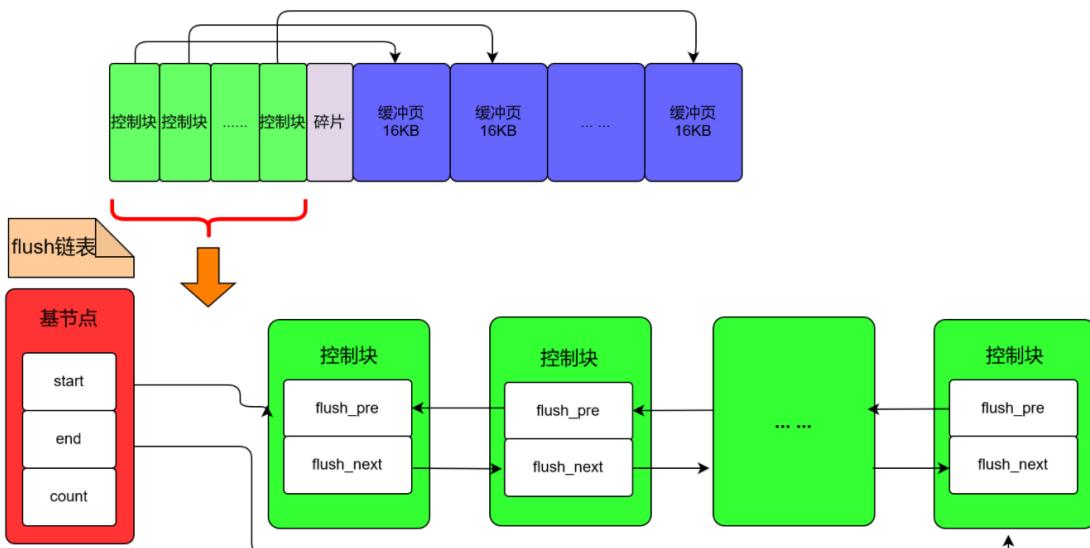
基节点：是一块单独申请的内存空间（占 40 字节），并不在 Buffer Pool 的那一大片连续内存空间里
磁盘加载页的流程：

- 从 Free 链表中取出一个空闲的缓冲页
- 把缓冲页对应的控制块的信息填上（页所在的表空间、页号之类的信息）
- 把缓冲页对应的 Free 链表节点（控制块）从链表中移除，表示该缓冲页已经被使用

参考文章：<https://blog.csdn.net/li1325169021/article/details/121124440>

Flush 链表

Flush 链表是一个用来存储脏页的链表，对于已经修改过的缓冲脏页，第一次修改后加入到链表头部，以后每次修改都不会重新加入，只修改部分控制信息，出于性能考虑并不是直接更新到磁盘，而是在未来的某个时间进行刷脏



后台有专门的线程每隔一段时间把脏页刷新到磁盘：

- 从 Flush 链表中刷新一部分页面到磁盘：
 - 后台线程定时从 Flush 链表刷脏，根据系统的繁忙程度来决定刷新速率，这种方式称为 BUF_FLUSH_LIST
 - 线程刷脏的比较慢，导致用户线程加载一个新的数据页时发现没有空闲缓冲页，此时会尝试从 LRU 链表尾部寻找缓冲页直接释放，如果该页面是已经修改过的脏页就同步刷新到磁盘，速度较慢，这种方式称为 BUF_FLUSH_SINGLE_PAGE
- 从 LRU 链表的冷数据中刷新一部分页面到磁盘，即：BUF_FLUSH_LRU
 - 后台线程会定时从 LRU 链表的尾部开始扫描一些页面，扫描的页面数量可以通过系统变量 `innodb_lru_scan_depth` 指定，如果在 LRU 链表中发现脏页，则把它们刷新到磁盘，这种方式称为 BUF_FLUSH_LRU
 - 控制块里会存储该缓冲页是否被修改的信息，所以可以很容易的获取到某个缓冲页是否是脏页

参考文章：<https://blog.csdn.net/li1325169021/article/details/121125765>

LRU 链表

Buffer Pool 需要保证缓存的命中率，所以 MySQL 创建了一个 LRU 链表，当访问某个页时：

- 如果该页不在 Buffer Pool 中，把该页从磁盘加载进来后会将该缓冲页对应的控制块作为节点放入 **LRU 链表的头部**，保证热点数据在链表头
- 如果该页在 Buffer Pool 中，则直接把该页对应的控制块移动到 LRU 链表的头部，所以 LRU 链表尾部就是最近最少使用的缓冲页

MySQL 基于局部性原理提供了预读功能：

- 线性预读：系统变量 `innodb_read_ahead_threshold`，如果顺序访问某个区（extent：16 KB 的页，连续 64 个形成一个区，一个区默认 1MB 大小）的页面数超过了该系统变量值，就会触发一次 **异步读取**下一个区中全部的页面到 Buffer Pool 中
- 随机预读：如果某个区 13 个连续的页面都被加载到 Buffer Pool，无论这些页面是否是顺序读取，都会触发一次 **异步读取**本区所有的其他页面到 Buffer Pool 中

预读会造成加载太多用不到的数据页，造成那些使用频率很高的数据页被挤到 LRU 链表尾部，所以 InnoDB 将 LRU 链表分成两段，**冷热数据隔离**：

- 一部分存储使用频率很高的数据页，这部分链表也叫热数据，young 区，靠近链表头部的区域
- 一部分存储使用频率不高的冷数据，old 区，靠近链表尾部，默认占 37%，可以通过系统变量 `innodb_old_blocks_pct` 指定

当磁盘上的某数据页被初次加载到 Buffer Pool 中会被放入 old 区，淘汰时优先淘汰 old 区

- 当对 old 区的数据进行访问时，会在控制块记录下访问时间，等待后续的访问时间与第一次访问的时间是否在某个时间间隔内，通过系统变量 `innodb_old_blocks_time` 指定时间间隔，默认 1000ms，成立就移动到 young 区的链表头部
- `innodb_old_blocks_time` 为 0 时，每次访问一个页面都会放入 young 区的头部

参数优化

InnoDB 用一块内存区做 IO 缓存池，该缓存池不仅用来缓存 InnoDB 的索引块，也用来缓存 InnoDB 的数据块，可以通过下面的指令查看 Buffer Pool 的状态信息：

```
SHOW ENGINE INNODB STATUS\G
```

`Buffer pool hit rate` 字段代表**内存命中率**，表示 Buffer Pool 对查询的加速效果

核心参数：

- `innodb_buffer_pool_size`：该变量决定了 Innodb 存储引擎表数据和索引数据的最大缓存区大小，默认 128M

```
SHOW VARIABLES LIKE 'innodb_buffer_pool_size';
```

在保证操作系统及其他程序有足够的内存可用的情况下，`innodb_buffer_pool_size` 的值越大，缓存命中率越高，建议设置成可用物理内存的 60%~80%

```
innodb_buffer_pool_size=512M
```

- `innodb_log_buffer_size`：该值决定了 Innodb 日志缓冲区的大小，保存要写入磁盘上的日志文件数据

对于可能产生大量更新记录的大事务，增加该值的大小，可以避免 Innodb 在事务提交前就执行不必要的日志写入磁盘操作，影响执行效率，通过配置文件修改：

```
innodb_log_buffer_size=10M
```

在多线程下，访问 Buffer Pool 中的各种链表都需要加锁，所以将 Buffer Pool 拆成若干个小实例，**每个线程对应一个实例**，独立管理内存空间和各种链表（类似 ThreadLocal），多线程访问各自实例互不影响，提高了并发能力

MySQL 5.7.5 之前 `innodb_buffer_pool_size` 只支持在系统启动时修改，现在已经支持运行时修改 Buffer Pool 的大小，但是每次调整参数都会重新向操作系统申请一块连续的内存空间，**将旧的缓冲池的内容拷贝到新空间**非常耗时，所以 MySQL 开始以一个 chunk 为单位向操作系统申请内存，所以一个 Buffer Pool 实例可以由多个 chunk 组成

- 在系统启动时设置系统变量 `innodb_buffer_pool_instance` 可以指定 Buffer Pool 实例的个数，但是当 Buffer Pool 小于 1GB 时，设置多个实例时无效的
- 指定系统变量 `innodb_buffer_pool_chunk_size` 来改变 chunk 的大小，只能在启动时修改，运行中不能修改，而且该变量并不包含缓冲页的控制块的内存大小
- `innodb_buffer_pool_size` 必须是 `innodb_buffer_pool_chunk_size × innodb_buffer_pool_instance` 的倍数，默认值是 $128M \times 16 = 2G$ ，Buffer Pool 必须是 2G 的整数倍，如果指定 5G，会自动调整成 6G
- 如果启动时 `chunk × instances > pool_size`，那么 chunk 的值会自动设置为 `pool_size ÷ instances`

内存优化

Change

InnoDB 管理的 Buffer Pool 中有一块内存叫 Change Buffer 用来对**增删改操作**提供缓存，可以通过参数来动态设置，设置为 50 时表示 Change Buffer 的大小最多占用 Buffer Pool 的 50%

- 唯一索引的更新不能使用 Change Buffer，需要将数据页读入内存，判断没有冲突在写入
- 普通索引可以使用 Change Buffer，**直接写入 Buffer 就结束**，不用校验唯一性

Change Buffer 并不是数据页，只是对操作的缓存，所以需要将 Change Buffer 中的操作应用到旧数据页，得到新的数据页（脏页）的过程称为 Merge

- 触发时机：访问数据页时会触发 Merge、后台有定时线程进行 Merge、在数据库正常关闭（shutdown）的过程中也会触发
- 工作流程：首先从磁盘读入数据页到内存（因为 Buffer Pool 中不一定存在对应的数据页），从 Change Buffer 中找到对应的操作应用到数据页，得到新的数据页即为脏页，然后写入 redo log，等待刷脏即可

说明：Change Buffer 中的记录，在事务提交时也会写入 redo log，所以是可以保证不丢失的

业务场景：

- 对于**写多读少**的业务来说，页面在写完以后马上被访问到的概率比较小，此时 Change Buffer 的使用效果最好，常见的就是账单类、日志类的系统
- 一个业务的更新模式是写入后马上做查询，那么即使满足了条件，将更新先记录在 Change Buffer，但之后由于马上要访问这个数据页，会立即触发 Merge 过程，这样随机访问 IO 的次数不会减少，并且增加了 Change Buffer 的维护代价

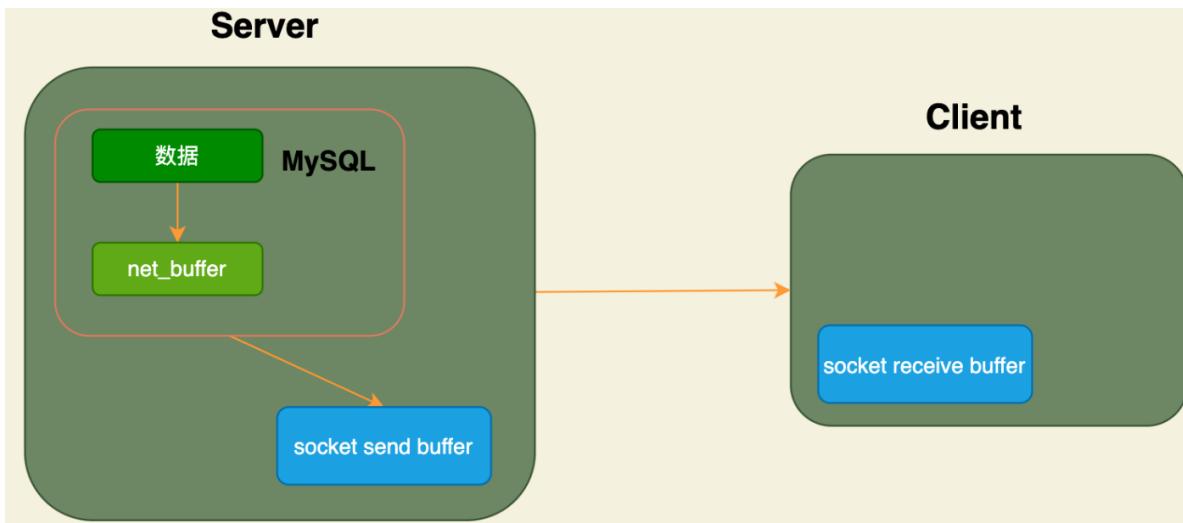
补充：Change Buffer 的前身是 Insert Buffer，只能对 Insert 操作优化，后来增加了 Update/Delete 的支持，改为 Change Buffer

Net

Server 层针对**优化查询**的内存为 Net Buffer，内存的大小是由参数 `net_buffer_length` 定义，默认 16k，实现流程：

- 获取一行数据写入 Net Buffer，重复获取直到 Net Buffer 写满，调用网络接口发出去
- 若发送成功就清空 Net Buffer，然后继续取下一行；若发送函数返回 EAGAIN 或 WSAEWOULDBLOCK，表示本地网络栈 `socket send buffer` 写满了，**进入等待**，直到网络栈重新可写再继续发送

MySQL 采用的是边读边发的逻辑，因此对于数据量很大的查询来说，不会在 Server 端保存完整的结果集，如果客户端读结果不及时，会堵住 MySQL 的查询过程，但是**不会把内存打爆导致 OOM**



SHOW PROCESSLIST 获取线程信息后，处于 Sending to client 状态代表服务器端的网络栈写满，等待客户端接收数据

假设有一个业务的逻辑比较复杂，每读一行数据以后要处理很久的逻辑，就会导致客户端要过很久才会去取下一行数据，导致 MySQL 的阻塞，一直处于 Sending to client 的状态

解决方法：如果一个查询的返回结果很是很多，建议使用 mysql_store_result 这个接口，直接把查询结果保存到本地内存

参考文章：https://blog.csdn.net/qq_33589510/article/details/117673449

Read

read_rnd_buffer 是 MySQL 的随机读缓冲区，当按任意顺序读取记录行时将分配一个随机读取缓冲区，进行排序查询时，MySQL 会首先扫描一遍该缓冲，以避免磁盘搜索，提高查询速度，大小是由 read_rnd_buffer_size 参数控制的

Multi-Range Read 优化，**将随机 IO 转化为顺序 IO** 以降低查询过程中 IO 开销，因为大多数的数据都是按照主键递增顺序插入得到，所以按照主键的递增顺序查询的话，对磁盘的读比较接近顺序读，能够提升读性能

二级索引为 a，聚簇索引为 id，优化回表流程：

- 根据索引 a，定位到满足条件的记录，将 id 值放入 read_rnd_buffer 中
- 将 read_rnd_buffer 中的 id 进行**递增排序**
- 排序后的 id 数组，依次回表到主键 id 索引中查记录，并作为结果返回

说明：如果步骤 1 中 read_rnd_buffer 放满了，就会先执行步骤 2 和 3，然后清空 read_rnd_buffer，之后继续找索引 a 的下个记录

使用 MRR 优化需要设进行设置：

```
SET optimizer_switch='mrr_cost_based=off'
```

Key

MyISAM 存储引擎使用 key_buffer 缓存索引块，加速 MyISAM 索引的读写速度。对于 MyISAM 表的数据块没有特别的缓存机制，完全依赖于操作系统的 IO 缓存

- key_buffer_size：该变量决定 MyISAM 索引块缓存区的大小，直接影响到 MyISAM 表的存取效率

```
SHOW VARIABLES LIKE 'key_buffer_size'; -- 单位是字节
```

在 MySQL 配置文件中设置该值，建议至少将1/4可用内存分配给 key_buffer_size：

```
vim /etc/mysql/my.cnf  
key_buffer_size=1024M
```

- read_buffer_size：如果需要经常顺序扫描 MyISAM 表，可以通过增大 read_buffer_size 的值来改善性能。但 read_buffer_size 是每个 Session 独占的，如果默认值设置太大，并发环境就会造成内存浪费
- read_rnd_buffer_size：对于需要做排序的 MyISAM 表的查询，如带有 ORDER BY 子句的语句，适当增加该的值，可以改善此类的 SQL 的性能，但是 read_rnd_buffer_size 是每个 Session 独占的，如果默认值设置太大，就会造成内存浪费

存储优化

数据存储

系统表空间是用来放系统信息的，比如数据字典什么的，对应的磁盘文件是 ibdata，数据表空间是一个个的表数据文件，对应的磁盘文件就是表名.ibd

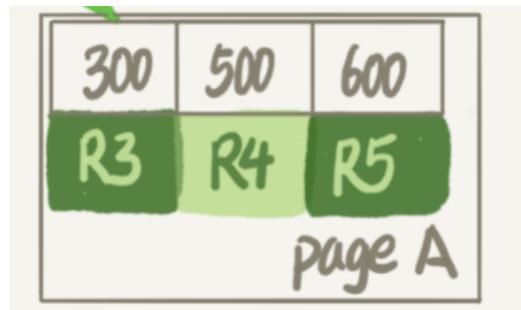
表数据既可以存在共享表空间里，也可以是单独的文件，这个行为是由参数 innodb_file_per_table 控制的：

- OFF：表示表的数据放在系统共享表空间，也就是跟数据字典放在一起
- ON：表示每个 InnoDB 表数据存储在一个以 .ibd 为后缀的文件中（默认）

一个表单独存储为一个文件更容易管理，在不需要这个表时通过 drop table 命令，系统就会直接删除这个文件；如果是放在共享表空间中，即使表删掉了，空间也是不会回收的

数据删除

MySQL 的数据删除就是移除掉某个记录后，该位置就被标记为**可复用**，如果有符合范围条件的数据可以插入到这里。符合范围条件的意思是假设删除记录 R4，之后要再插入一个 ID 在 300 和 600 之间的记录时，就会复用这个位置



InnoDB 的数据是按页存储的如果删掉了一个数据页上的所有记录，整个数据页就可以被复用了，如果相邻的两个数据页利用率都很小，系统就会把这两个页上的数据合到其中一个页上，另外一个数据页就被标记为可复用

删除命令其实只是把记录的位置，或者**数据页标记为了可复用，但磁盘文件的大小是不会变的**，这些可以复用还没有被使用的空间，看起来就像是空洞，造成数据库的稀疏，因此需要进行紧凑处理

重建数据

重建表就是按照主键 ID 递增的顺序，把数据一行一行地从旧表中读出来再插入到新表中，让数据更加紧凑。重建表时 MySQL 会自动完成转存数据、交换表名、删除旧表的操作，线上操作会阻塞大量的线程增删改查的操作

重建命令：

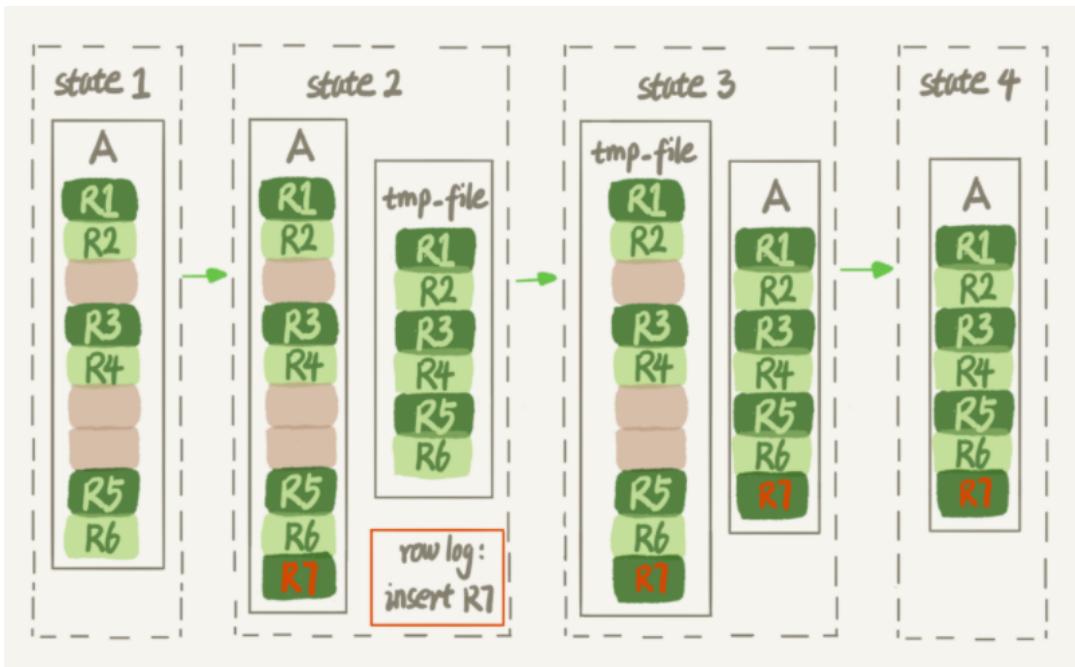
```
ALTER TABLE A ENGINE=InnoDB
```

工作流程：新建临时表 tmp_table B (在 Server 层创建的)，把表 A 中的数据导入到表 B 中，操作完成后用表 B 替换表 A，完成重建

重建表的步骤需要 DDL 不是 Online 的，因为在导入数据的过程有新的数据要写入到表 A 的话，就会造成数据丢失

MySQL 5.6 版本开始引入的 **Online DDL**，重建表的命令默认执行此步骤：

- 建立一个临时文件 tmp_file (InnoDB 创建)，扫描表 A 主键的所有数据页
- 用数据页中表 A 的记录生成 B+ 树，存储到临时文件中
- 生成临时文件的过程中，将所有对 A 的操作记录在一个日志文件 (row log) 中，对应的是图中 state2 的状态
- 临时文件生成后，将日志文件中的操作应用到临时文件，得到一个逻辑数据上与表 A 相同的数据文件，对应的就是图中 state3
- 用临时文件替换表 A 的数据文件



Online DDL 操作会先获取 MDL 写锁，再退化成 MDL 读锁。但 MDL 写锁持有时间比较短，所以可以称为 Online；而 MDL 读锁，不阻止数据增删查改，但会阻止其它线程修改表结构（可以对比 `ANALYZE TABLE t` 命令）

问题：重建表可以收缩表空间，但是执行指令后整体占用空间增大

原因：在重建表后 InnoDB 不会把整张表占满，每个页留了 1/16 给后续的更新使用。表在未整理之前页已经占用 15/16 以上，收缩之后需要保持数据占用空间在 15/16，所以文件占用空间更大才能保持

注意：临时文件也要占用空间，如果空间不足会重建失败

原地置换

DDL 中的临时表 tmp_table 是在 Server 层创建的，Online DDL 中的临时文件 tmp_file 是 InnoDB 在内部创建出来的，整个 DDL 过程都在 InnoDB 内部完成，对于 Server 层来说，没有把数据挪动到临时表，是一个原地操作，这就是 inplace

两者的关系：

- DDL 过程如果是 Online 的，就一定是 inplace 的
- inplace 的 DDL，有可能不是 Online 的，截止到 MySQL 8.0，全文索引 (FULLTEXT) 和空间索引 (SPATIAL) 属于这种情况

并发优化

MySQL Server 是多线程结构，包括后台线程和客户服务线程。多线程可以有效利用服务器资源，提高数据库的并发性能。在 MySQL 中，控制并发连接和线程的主要参数：

- max_connections：控制允许连接到 MySQL 数据库的最大连接数，默认值是 151

如果状态变量 connection_errors_max_connections 不为零，并且一直增长，则说明不断有连接请求因数据库连接数已达到允许最大值而失败，这时可以考虑增大 max_connections 的值

MySQL 最大可支持的连接数取决于很多因素，包括操作系统平台的线程库的质量、内存大小、每个连接的负荷、CPU 的处理速度、期望的响应时间等。在 Linux 平台下，性能好的服务器，可以支持 500-1000 个连接，需要根据服务器性能进行评估设定

- innodb_thread_concurrency：并发线程数，代表系统内同时运行的线程数量（已经被移除）
- back_log：控制 MySQL 监听 TCP 端口时的积压请求栈的大小

如果 MySQL 的连接数达到 max_connections 时，新来的请求将被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即 back_log。如果等待连接的数量超过 back_log，将不被授予连接资源直接报错

5.6.6 版本之前默认值为 50，之后的版本默认为 $50 + (\max_connections / 5)$ ，但最大不超过 900，如果需要数据库在较短的时间内处理大量连接请求，可以考虑适当增大 back_log 的值

- table_open_cache：控制所有 SQL 语句执行线程可打开表缓存的数量

在执行 SQL 语句时，每个执行线程至少要打开 1 个表缓存，该参数的值应该根据设置的最大连接数以及每个连接执行关联查询中涉及的表的最大数量来设定：`max_connections * N`

- thread_cache_size：可控制 MySQL 缓存客户服务线程的数量

为了加快连接数据库的速度，MySQL 会缓存一定数量的客户服务线程以备重用，池化思想

- innodb_lock_wait_timeout：设置 InnoDB 事务等待行锁的时间，默认值是 50ms

对于需要快速反馈的业务系统，可以将行锁的等待时间调小，以避免事务被长时间挂起；对于后台运行的批量处理程序来说，可以将行锁的等待时间调大，以避免发生大的回滚操作

事务机制

基本介绍

事务（Transaction）是访问和更新数据库的程序执行单元；事务中可能包含一个或多个 SQL 语句，这些语句要么都执行，要么都不执行，作为一个关系型数据库，MySQL 支持事务。

单元中的每条 SQL 语句都相互依赖，形成一个整体

- 如果某条 SQL 语句执行失败或者出现错误，那么整个单元就会回滚，撤回到事务最初的状态
- 如果单元中所有的 SQL 语句都执行成功，则事务就顺利执行

事务的四大特征：ACID

- 原子性 (atomicity)

- 一致性 (consistency)
- 隔离性 (isolation)
- 持久性 (durability)

事务的几种状态：

- 活动的 (active) : 事务对应的数据库操作正在执行中
- 部分提交的 (partially committed) : 事务的最后一个操作执行完，但是内存还没刷新至磁盘
- 失败的 (failed) : 当事务处于活动状态或部分提交状态时，如果数据库遇到了错误或刷脏失败，或者用户主动停止当前的事务
- 中止的 (aborted) : 失败状态的事务回滚完成后的状态
- 提交的 (committed) : 当处于部分提交状态的事务刷脏成功，就处于提交状态

事务管理

基本操作

事务管理的三个步骤

1. 开启事务：记录回滚点，并通知服务器，将要执行一组操作，要么同时成功、要么同时失败
2. 执行 SQL 语句：执行具体的一条或多条 SQL 语句
3. 结束事务（提交 | 回滚）
 - 提交：没出现问题，数据进行更新
 - 回滚：出现问题，数据恢复到开启事务时的状态

事务操作：

- 显式开启事务

```
START TRANSACTION [READ ONLY|READ WRITE|WITH CONSISTENT SNAPSHOT]; #可以跟一个  
或多个状态，最后的是一致性读  
BEGIN [WORK];
```

说明：不填状态默认是读写事务

- 回滚事务，用来手动中止事务

```
ROLLBACK;
```

- 提交事务，显示执行是手动提交，MySQL 默认为自动提交

```
COMMIT;
```

- 保存点：在事务的执行过程中设置的还原点，调用 ROLLBACK 时可以指定回滚到哪个点

```
SAVEPOINT point_name;          #设置保存点  
RELEASE point_name            #删除保存点  
ROLLBACK [WORK] TO [SAVEPOINT] point_name #回滚至某个保存点, 不填默认回滚到事务执行之前的状态
```

- 操作演示

```
-- 开启事务  
START TRANSACTION;  
  
-- 张三给李四转账500元  
-- 1.张三账户-500  
UPDATE account SET money=money-500 WHERE NAME='张三';  
-- 2.李四账户+500  
UPDATE account SET money=money+500 WHERE NAME='李四';  
  
-- 回滚事务(出现问题)  
ROLLBACK;  
  
-- 提交事务(没出现问题)  
COMMIT;
```

提交方式

提交方式的相关语法:

- 查看事务提交方式

```
SELECT @@AUTOCOMMIT;      -- 会话, 1 代表自动提交 0 代表手动提交  
SELECT @@GLOBAL.AUTOCommit; -- 系统
```

- 修改事务提交方式

```
SET @@AUTOCOMMIT=数字;      -- 系统  
SET AUTOCOMMIT=数字;       -- 会话
```

- 系统变量的操作:

```
SET [GLOBAL|SESSION] 变量名 = 值;           -- 默认是会话  
SET @@[GLOBAL|SESSION].]变量名 = 值;         -- 默认是系统
```

```
SHOW [GLOBAL|SESSION] VARIABLES [LIKE '变量%'];    -- 默认查看会话内系统变量值
```

工作原理:

- 自动提交: 如果没有 START TRANSACTION 显式地开始一个事务, 那么每条 SQL 语句都会被当做一个事务执行提交操作; 显式开启事务后, 会在本次事务结束(提交或回滚)前暂时关闭自动提交

- 手动提交：不需要显式的开启事务，所有的 SQL 语句都在一个事务中，直到执行了提交或回滚，然后进入下一个事务
 - 隐式提交：存在一些特殊的命令，在事务中执行了这些命令会马上**强制执行 COMMIT 提交事务**
 - **DDL 语句** (CREATE/DROP/ALTER)、LOCK TABLES 语句、LOAD DATA 导入数据语句、主从复制语句等
 - 当一个事务还没提交或回滚，显式的开启一个事务会隐式的提交上一个事务
-

事务 ID

事务在执行过程中对某个表执行了**增删改操作或者创建表**，就会为当前事务分配一个独一无二的事务 ID（对临时表并不会分配 ID），如果当前事务没有被分配 ID，默认是 0

说明：只读事务不能对普通的表进行增删改操作，但是可以对临时表增删改，读写事务可以对数据表执行增删改查操作

事务 ID 本质上就是一个数字，服务器在内存中维护一个全局变量：

- 每当需要为某个事务分配 ID，就会把全局变量的值赋值给事务 ID，然后变量自增 1
- 每当变量值为 256 的倍数时，就将该变量的值刷新到系统表空间的 Max Trx ID 属性中，该属性占 8 字节
- 系统再次启动后，会读取表空间的 Max Trx ID 属性到内存，加上 256 后赋值给全局变量，因为关机时的事务 ID 可能并不是 256 的倍数，会比 Max Trx ID 大，所以需要加上 256 保持事务 ID 是一个递增的数字

聚簇索引的行记录除了完整的数据，还会自动添加 trx_id、roll_pointer 隐藏列，如果表中没有主键并且没有非空唯一索引，也会添加一个 row_id 的隐藏列作为聚簇索引

隔离级别

四种级别

事务的隔离级别：多个客户端操作时，各个客户端的事务之间应该是隔离的，**不同的事务之间不该互相影响**，而如果多个事务操作同一批数据时，则需要设置不同的隔离级别，否则就会产生问题。

隔离级别分类：

| 隔离级别 | 名称 | 会引发的问题 | 数据库默认隔离级别 |
|------------------|------|-------------|---------------------|
| Read Uncommitted | 读未提交 | 脏读、不可重复读、幻读 | |
| Read Committed | 读已提交 | 不可重复读、幻读 | Oracle / SQL Server |
| Repeatable Read | 可重复读 | 幻读 | MySQL |
| Serializable | 可串行化 | 无 | |

一般来说，隔离级别越低，系统开销越低，可支持的并发越高，但隔离性也越差

- 脏写 (Dirty Write): 当两个或多个事务选择同一行，最初的事务修改的值被后面事务修改的值覆盖，所有的隔离级别都可以避免脏写（又叫丢失更新），因为有行锁
- 脏读 (Dirty Reads): 在一个事务处理过程中读取了另一个未提交的事务中修改过的数据
- 不可重复读 (Non-Repeatable Reads): 在一个事务处理过程中读取了另一个事务中修改并已提交的数据

可重复读的意思是不管读几次，结果都一样，可以重复的读，可以理解为快照读，要读的数据集不会发生变化

- 幻读 (Phantom Reads): 在事务中按某个条件先后两次查询数据库，后一次查询查到了前一次查询没有查到的行，**数据条目**发生了变化。比如查询某数据不存在，准备插入此记录，但执行插入时发现此记录已存在，无法插入

隔离级别操作语法：

- 查询数据库隔离级别

```
SELECT @@TX_ISOLATION;          -- 会话  
SELECT @@GLOBAL.TX_ISOLATION;   -- 系统
```

- 修改数据库隔离级别

```
SET GLOBAL TRANSACTION ISOLATION LEVEL 级别字符串;
```

加锁分析

InnoDB 存储引擎支持事务，所以加锁分析是基于该存储引擎

- Read Uncommitted 级别，任何操作都不会加锁
- Read Committed 级别，增删改操作会加写锁（行锁），读操作不加锁

在 Server 层过滤条件时发现不满足的记录会调用 `unlock_row` 方法释放该记录的行锁，保证最后只有满足条件的记录加锁，但是扫表过程中每条记录的**加锁操作不能省略**。所以对数据量很大的表做批量修改时，如果无法使用相应的索引（全表扫描），在 Server 过滤数据时就会特别慢，出现虽然没有修改某些行的数据，但是还是被锁住了的现象（锁表），这种情况同样适用于 RR

- Repeatable Read 级别，增删改操作会加写锁，读操作不加锁。因为读写锁不兼容，**加了读锁后其他事务就无法修改数据**，影响了并发性能，为了保证隔离性和并发性，MySQL 通过 MVCC 解决了读写冲突。RR 级别下的锁有很多种，锁机制章节详解
- Serializable 级别，读加共享锁，写加排他锁，读写互斥，使用的悲观锁的理论，实现简单，数据更加安全，但是并发能力非常差
 - 串行化：让所有事务按顺序单独执行，写操作会加写锁，读操作会加读锁
 - 可串行化：让所有操作相同数据的事务顺序执行，通过加锁实现

原子特性

实现方式

原子性是指事务是一个不可分割的工作单位，事务的操作如果成功就必须要完全应用到数据库，失败则不能对数据库有任何影响。比如事务中一个 SQL 语句执行失败，则已执行的语句也必须回滚，数据库退回到事务前的状态

InnoDB 存储引擎提供了两种事务日志：redo log（重做日志）和 undo log（回滚日志）

- redo log 用于保证事务持久性
- undo log 用于保证事务原子性和隔离性

undo log 属于**逻辑日志**，根据每行操作进行记录，记录了 SQL 执行相关的信息，用来回滚行记录到某个版本

当事务对数据库进行修改时，InnoDB 会先记录对应的 undo log，如果事务执行失败或调用了 rollback 导致事务回滚，InnoDB 会根据 undo log 的内容**做与之前相反的操作**：

- 对于每个 insert，回滚时会执行 delete
- 对于每个 delete，回滚时会执行 insert
- 对于每个 update，回滚时会执行一个相反的 update，把数据修改回去

参考文章：<https://www.cnblogs.com/kismetv/p/10331633.html>

DML 解析

INSERT

乐观插入：当前数据页的剩余空间充足，直接将数据进行插入

悲观插入：当前数据页的剩余空间不足，需要进行页分裂，申请一个新的页面来插入数据，会造成更多的 redo log，undo log 影响不大

当向某个表插入一条记录，实际上需要向聚簇索引和所有二级索引都插入一条记录，但是 undo log 只**针对聚簇索引记录**，在回滚时会根据聚簇索引去所有的二级索引进行回滚操作

roll_pointer 是一个指针，**指向记录对应的 undo log 日志**，一条记录就是一个数据行，行格式中的 roll_pointer 就指向 undo log

DELETE

插入到页面中的记录会根据 next_record 属性组成一个单向链表，这个链表称为正常链表，被删除的记录也会通过 next_record 组成一个垃圾链表，该链表中所占用的存储空间可以被重新利用，并不会直接清除数据

在页面 Page Header 中，PAGE_FREE 属性指向垃圾链表的头节点，删除的工作过程：

- 将要删除的记录的 delete_flag 位置为 1，其他不做修改，这个过程叫 **delete mark**
- 在事务提交前，delete_flag = 1 的记录一直都会处于中间状态
- 事务提交后，有专门的线程将 delete_flag = 1 的记录从正常链表移除并加入垃圾链表，这个过程叫 **purge**

purge 线程在执行删除操作时会创建一个 ReadView，根据事务的可见性移除数据（隔离特性部分详解）

当有新插入的记录时，首先判断 PAGE_FREE 指向的头节点是否足够容纳新纪录：

- 如果可以容纳新纪录，就会直接重用已删除的记录的存储空间，然后让 PAGE_FREE 指向垃圾链表的下一个节点
- 如果不能容纳新纪录，就直接向页面申请新的空间存储，并不会遍历垃圾链表

重用已删除的记录空间，可能会造成空间碎片，当数据页容纳不了一条记录时，会判断将碎片空间加起来是否可以容纳，判断为真就会重新组织页内的记录：

- 开辟一个临时页面，将页内记录一次插入到临时页面，此时临时页面时没有碎片的
- 把临时页面的内容复制到本页，这样就解放出了内存碎片，但是会耗费很大的性能资源

UPDATE

执行 UPDATE 语句，对于更新主键和不更新主键有两种不同的处理方式

不更新主键的情况：

- 就地更新 (in-place update)，如果更新后的列和更新前的列占用的存储空间一样大，就可以直接在原记录上修改
- 先删除旧纪录，再插入新纪录，这里的删除不是 delete mark，而是直接将记录加入垃圾链表，并且修改页面的相应的控制信息，执行删除的线程不是 purge，是执行更新的用户线程，插入新记录时可能造成页空间不足，从而导致页分裂

更新主键的情况：

- 将旧纪录进行 delete mark，在更新语句提交后由 purge 线程移入垃圾链表
- 根据更新的各列的值创建一条新纪录，插入到聚簇索引中

在对一条记录修改前会将记录的隐藏列 **trx_id** 和 **roll_pointer** 的旧值记录到当前 undo log 对应的属性中，这样当前记录的 roll_pointer 指向当前 undo log 记录，当前 undo log 记录的 roll_pointer 指向旧的 undo log 记录，形成一个版本链

UPDATE、DELETE 操作产生的 undo 日志会用于其他事务的 MVCC 操作，所以不能立即删除，INSERT 可以删除的原因是 MVCC 是对现有数据的快照

回滚日志

undo log 是采用段的方式来记录，Rollback Segement 称为回滚段，本质上就是一个类型是 Rollback Segement Header 的页面

每个回滚段中有 1024 个 undo slot，每个 slot 存放 undo 链表页面的头节点页号，每个链表对应一个叫 undo log segment 的段

- 在以前老版本，只支持 1 个 Rollback Segement，只能记录 1024 个 undo log segment
- MySQL5.5 开始支持 128 个 Rollback Segement，支持 128*1024 个 undo 操作

工作流程：

- 事务执行前需要到系统表空间第 5 号页面中分配一个回滚段（页），获取一个 Rollback Segement Header 页面的地址
- 回滚段页面有 1024 个 undo slot，首先去回滚段的两个 cached 链表获取缓存的 slot，缓存中没有就在回滚段页面中找一个可用的 undo slot 分配给当前事务
- 如果是缓存中获取的 slot，则该 slot 对应的 undo log segment 已经分配了，需要重新分配，然后从 undo log segment 中申请一个页面作为日志链表的头节点，并填入对应的 slot 中
- 每个事务 undo 日志在记录的时候**占用两个 undo 页面的组成链表**，分别为 insert undo 链表和 update undo 链表，链表的头节点页面为 first undo page 会包含一些管理信息，其他页面为 normal undo page

说明：事务执行过程的临时表也需要两个 undo 链表，不和普通表共用，这些链表并不是事务开始就分配，而是按需分配

隔离特性

实现方式

隔离性是指，事务内部的操作与其他事务是隔离的，多个并发事务之间要相互隔离，不能互相干扰

- 严格的隔离性，对应了事务隔离级别中的 serializable，实际应用中对性能考虑很少使用可串行化
- 与原子性、持久性侧重于研究事务本身不同，隔离性研究的是**不同事务**之间的相互影响

隔离性让并发情形下的事务之间互不干扰：

- 一个事务的写操作对另一个事务的写操作（写写）：锁机制保证隔离性
- 一个事务的写操作对另一个事务的读操作（读写）：MVCC 保证隔离性

锁机制：事务在修改数据之前，需要先获得相应的锁，获得锁之后，事务便可以修改数据；该事务操作期间，这部分数据是锁定的，其他事务如果需要修改数据，需要等待当前事务提交或回滚后释放锁（详见锁机制）

并发控制

MVCC 全称 Multi-Version Concurrency Control，即多版本并发控制，用来解决读写冲突的无锁并发控制，可以在发生读写请求冲突时不用加锁解决，这个读是指的快照读（也叫一致性读或一致性无锁读），而不是当前读：

- 快照读：实现基于 MVCC，因为是多版本并发，所以快照读读到的数据不一定是当前最新的数据，有可能是历史版本的数据
- 当前读：又叫加锁读，读取数据库记录是当前最新的版本（产生幻读、不可重复读），可以对读取的数据进行加锁，防止其他事务修改数据，是悲观锁的一种操作，读写操作加共享锁或者排他锁和串行化事务的隔离级别都是当前读

数据库并发场景：

- 读-读：不存在任何问题，也不需要并发控制
- 读-写：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读
- 写-写：有线程安全问题，可能会存在脏写（丢失更新）问题

MVCC 的优点：

- 在并发读写数据库时，做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了并发读写的性能
- 可以解决脏读，不可重复读等事务隔离问题（加锁也能解决），但不能解决更新丢失问题（写锁会解决）

提高读写和写写的并发性能：

- MVCC + 悲观锁：MVCC 解决读写冲突，悲观锁解决写写冲突
- MVCC + 乐观锁：MVCC 解决读写冲突，乐观锁解决写写冲突

参考文章：<https://www.jianshu.com/p/8845ddca3b23>

实现原理

隐藏字段

实现原理主要是隐藏字段，undo日志，Read View 来实现的

InnoDB 存储引擎，数据库中的聚簇索引每行数据，除了自定义的字段，还有数据库隐式定义的字段：

- DB_TRX_ID：最近修改事务 ID，记录创建该数据或最后一次修改该数据的事务 ID
- DB_ROLL_PTR：回滚指针，指向记录对应的 undo log 日志，undo log 中又指向上一个旧版本的 undo log
- DB_ROW_ID：隐含的自增 ID（**隐藏主键**），如果数据表没有主键，InnoDB 会自动以 DB_ROW_ID 作为聚簇索引

| name | age | DB_ROW_ID(隐式主键) | DB_TRX_ID(事务ID) | DB_ROLL_PTR(回滚指针) |
|-------|-----|-----------------|-----------------|-------------------|
| Jerry | 24 | 1 | 1 | 0x12446545 |

版本链

undo log 是逻辑日志，记录的是每个事务对数据执行的操作，而不是记录的全部数据，要根据 undo log 逆推出以往事务的数据

undo log 的作用：

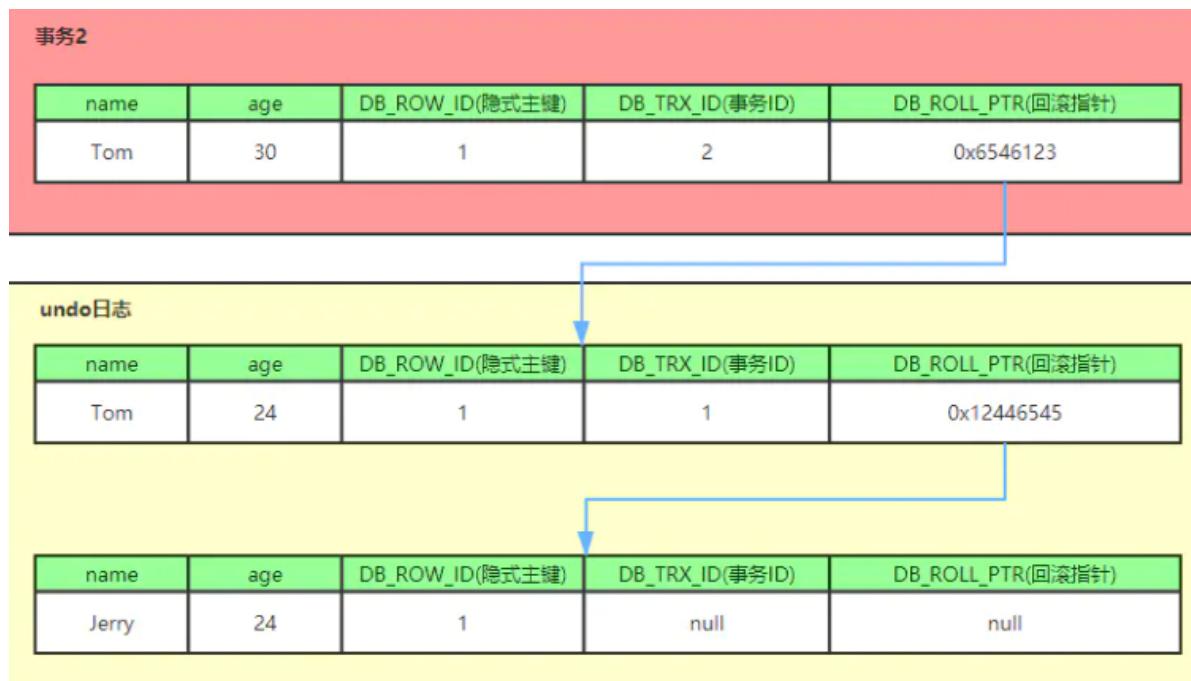
- 保证事务进行 rollback 时的原子性和一致性，当事务进行回滚的时候可以用 undo log 的数据进行恢复
- 用于 MVCC 快照读，通过读取 undo log 的历史版本数据可以实现不同事务版本号都拥有自己独立的快照数据

undo log 主要分为两种：

- insert undo log：事务在 insert 新记录时产生的 undo log，只在事务回滚时需要，并且在事务提交后可以被立即丢弃
- update undo log：事务在进行 update 或 delete 时产生的 undo log，在事务回滚时需要，在快照读时也需要。不能随意删除，只有在当前读或事务回滚不涉及该日志时，对应的日志才会被 purge 线程统一清除

每次对数据库记录进行改动，都会产生的新版本的 undo log，随着更新次数的增多，所有的版本都会被 roll_pointer 属性连接成一个链表，把这个链表称之为**版本链**，版本链的头节点就是当前的最新的 undo log，链尾就是最早的日 undo log

说明：因为 DELETE 删除记录，都是移动到垃圾链表中，不是真正的删除，所以才可以通过版本链访问原始数据



注意：undo 是逻辑日志，这里只是直观的展示出来

工作流程：

- 有个事务插入 person 表一条新记录，name 为 Jerry，age 为 24
- 事务 1 修改该行数据时，数据库会先对该行加排他锁，然后先记录 undo log，然后修改该行 name 为 Tom，并且修改隐藏字段的事务 ID 为当前事务 1 的 ID（默认为 1 之后递增），回滚指针

指向拷贝到 undo log 的副本记录，事务提交后，释放锁

- 以此类推
-

读视图

Read View 是事务进行读数据操作时产生的读视图，该事务执行快照读的那一刻会生成数据库系统当前的一个快照，记录并维护系统当前活跃事务的 ID，用来做可见性判断，根据视图判断当前事务能够看到哪个版本的数据

注意：这里的快照并不是把所有的数据拷贝一份副本，而是由 undo log 记录的逻辑日志，根据库中的数据进行计算出历史数据

工作流程：将版本链的头节点的事务 ID（最新数据事务 ID，大概率不是当前线程）DB_TRX_ID 取出来，与系统当前活跃事务的 ID 对比进行可见性分析，不可见就通过 DB_ROLL_PTR 回滚指针去取出 undo log 中的下一个 DB_TRX_ID 比较，直到找到最近的满足可见性的 DB_TRX_ID，该事务 ID 所在的旧记录就是当前事务能看见的最新的记录

Read View 几个属性：

- m_ids: 生成 Read View 时当前系统中活跃的事务 id 列表（未提交的事务集合，当前事务也在其中）
- min_trx_id: 生成 Read View 时当前系统中活跃的最小的事务 id，也就是 m_ids 中的最小值（已提交的事务集合）
- max_trx_id: 生成 Read View 时当前系统应该分配给下一个事务的 id 值，m_ids 中的最大值加 1（未开始事务）
- creator_trx_id: 生成该 Read View 的事务的事务 id，就是判断该 id 的事务能读到什么数据

creator 创建一个 Read View，进行可见性算法分析：（解决了读未提交）

- db_trx_id == creator_trx_id: 表示这个数据就是当前事务自己生成的，自己生成的数据自己肯定能看见，所以此数据对 creator 是可见的
 - db_trx_id < min_trx_id: 该版本对应的事务 ID 小于 Read view 中的最小活跃事务 ID，则这个事务在当前事务之前就已经被提交了，对 creator 可见（因为比已提交的最大事务 ID 小的并不一定已经提交，所以应该判断是否在活跃事务列表）
 - db_trx_id >= max_trx_id: 该版本对应的事务 ID 大于 Read view 中当前系统的最大事务 ID，则说明该数据是在当前 Read view 创建之后才产生的，对 creator 不可见
 - min_trx_id <= db_trx_id < max_trx_id: 判断 db_trx_id 是否在活跃事务列表 m_ids 中
 - 在列表中，说明该版本对应的事务正在运行，数据不能显示（**不能读到未提交的数据**）
 - 不在列表中，说明该版本对应的事务已经被提交，数据可以显示（**可以读到已经提交的数据**）
-

工作流程

表 user 数据

| id | name | age |
|----|------|-----|
| 1 | 张三 | 18 |

Transaction 20:

```
START TRANSACTION; -- 开启事务  
UPDATE user SET name = '李四' WHERE id = 1;  
UPDATE user SET name = '王五' WHERE id = 1;
```

Transaction 60:

```
START TRANSACTION; -- 开启事务  
-- 操作表的其他数据
```



ID 为 0 的事务创建 Read View:

- m_ids: 20、60
- min_trx_id: 20
- max_trx_id: 61
- creator_trx_id: 0

| 某版本 | trx_id == creator_trx_id | trx_id < min_trx_id | trx_id > max_trx_id | min_trx_id <= trx_id <= max_trx_id |
|-----|--------------------------|---------------------|---------------------|------------------------------------|
| 王五 | 20 == 0 ? | 20 < 20 ? | 20 > 61 ? | 20 < 20 < 60 ? |
| 李四 | 20 == 0 ? | 20 < 20 ? | 20 > 61 ? | 20 < 20 < 60 ? |
| 张三 | 10 == 0 ? | 10 < 20 ? | | |

只有红框部分才复合条件，所以只有张三对应的版本的数据可以被看到

参考视频: <https://www.bilibili.com/video/BV1t5411u7Fg>

二级索引

只有在聚簇索引中才有 trx_id 和 roll_pointer 的隐藏列，对于二级索引判断可见性的方式：

- 二级索引页面的 Page Header 中有一个 PAGE_MAX_TRX_ID 属性，代表修改当前页面的最大的事务 ID，SELECT 语句访问某个二级索引时会判断 ReadView 的 min_trx_id 是否大于该属性，大于说明该页面的所有属性对 ReadView 可见
 - 如果属性判断不可见，就需要利用二级索引获取主键值进行回表操作，得到聚簇索引后按照聚簇索引的可见性判断的方法操作
-

RC RR

Read View 用于支持 RC (Read Committed, 读已提交) 和 RR (Repeatable Read, 可重复读) 隔离级别的实现，所以 **SELECT 在 RC 和 RR 隔离级别使用 MVCC 读取记录**

RR、RC 生成时机：

- RC 隔离级别下，每次读取数据前都会生成最新的 Read View (当前读)
- RR 隔离级别下，在第一次数据读取时才会创建 Read View (快照读)

RC、RR 级别下的 InnoDB 快照读区别

- RC 级别下，事务中每次快照读都会新生成一个 Read View，这就是在 RC 级别下的事务中可以看到别的事务提交的更新的原因
- RR 级别下，某个事务的对某条记录的**第一次快照读**会创建一个 Read View，将当前系统活跃的其他事务记录起来，此后在调用快照读的时候，使用的是同一个 Read View，所以一个事务的查询结果每次都是相同的

RR 级别下，通过 `START TRANSACTION WITH CONSISTENT SNAPSHOT` 开启事务，会在执行该语句后立刻生成一个 Read View，不是在执行第一条 SELECT 语句时生成（所以说 `START TRANSACTION` 并不是事务的起点，执行第一条语句才算起点）

解决幻读问题：

- 快照读：通过 MVCC 来进行控制的，在可重复读隔离级别下，普通查询是快照读，是不会看到别的事务插入的数据的，但是**并不能完全避免幻读**

场景：RR 级别，T1 事务开启，创建 Read View，此时 T2 去 INSERT 新的一行然后提交，然后 T1 去 UPDATE 该行会发现更新成功，并且把这条新记录的 trx_id 变为当前的事务 id，所以对当前事务就是可见的。因为 **Read View 并不能阻止事务去更新数据，更新数据都是先读后写并且是当前读**，读取到的是最新版本的数据

- 当前读：通过 next-key 锁（行锁 + 间隙锁）来解决问题
-

持久特性

实现方式

持久性是指一个事务一旦被提交了，那么对数据库中数据的改变就是永久性的，接下来的其他操作或故障不应该对其有任何影响。

Buffer Pool 的使用提高了读写数据的效率，但是如果 MySQL 宕机，此时 Buffer Pool 中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证，所以引入了 redo log 日志：

- redo log **记录数据页的物理修改**，而不是某一行或某几行的修改，用来恢复提交后的数据页，只能**恢复到最后一次提交的位置**
- redo log 采用的是 WAL (Write-ahead logging, **预写式日志**)，所有修改要先写入日志，再更新到磁盘，保证了数据不会因 MySQL 宕机而丢失，从而满足了持久性要求
- 简单的 redo log 是纯粹的物理日志，复杂的 redo log 会存在物理日志和逻辑日志

工作过程：MySQL 发生了宕机，InnoDB 会判断一个数据页在崩溃恢复时丢失了更新，就会将它读到内存，然后根据 redo log 内容更新内存，更新完成后，内存页变成脏页，然后进行刷脏

缓冲池的刷脏策略：

- redo log 文件是固定大小的，如果写满了就要擦除以前的记录，在擦除之前需要把对应的更新持久化到磁盘中
- Buffer Pool 内存不足，需要淘汰部分数据页 (LRU 链表尾部)，如果淘汰的是脏页，就要先将脏页写到磁盘 (要避免大事务)
- 系统空闲时，后台线程会自动进行刷脏 (Flush 链表部分已经详解)
- MySQL 正常关闭时，会把内存的脏页都刷新到磁盘上

重做日志

日志缓冲

服务器启动时会向操作系统申请一片连续内存空间作为 redo log buffer (重做日志缓冲区)，可以通过 `innodb_log_buffer_size` 系统变量指定 redo log buffer 的大小，默认是 16MB

log buffer 被划分为若干 redo log block (块，类似数据页的概念)，每个默认大小 512 字节，每个 block 由 12 字节的 log block head、496 字节的 log block body、4 字节的 log block trailer 组成

- 当数据修改时，先修改 Change Buffer 中的数据，然后在 redo log buffer 记录这次操作，写入 log buffer 的过程是**顺序写入的** (先写入前面的 block，写满后继续写下一个)
- log buffer 中有一个指针 `buf_free`，来标识该位置之前都是填满的 block，该位置之后都是空闲区域

MySQL 规定对底层页面的一次原子访问称为一个 Mini-Transaction (MTR)，比如在 B+ 树上插入一条数据就算一个 MTR

- 一个事务包含若干个 MTR，一个 MTR 对应一组若干条 redo log，一组 redo log 是不可分割的，在进行数据恢复时也把一组 redo log 当作一个不可分割的整体处理
- 不是每生成一条 redo 日志就将其插入到 log buffer 中，而是一个 MTR 结束后**将一组 redo 日志写入**

InnoDB 的 redo log 是**固定大小的**, redo 日志在磁盘中以文件组的形式存储, 同一组中的每个文件大小一样格式一样

- `innodb_log_group_home_dir` 代表磁盘存储 redo log 的文件目录, 默认是当前数据目录
- `innodb_log_file_size` 代表文件大小, 默认 48M, `innodb_log_files_in_group` 代表文件个数, 默认 2 最大 100, 所以日志的文件大小为 `innodb_log_file_size * innodb_log_files_in_group`

redo 日志文件也是由若干个 512 字节的 block 组成, 日志文件的前 2048 个字节 (前 4 个 block) 用来存储一些管理信息, 以后的用来存储 log buffer 中的 block 镜像

注意: block 并不代表一组 redo log, 一组日志可能占用不到一个 block 或者几个 block, 依赖于 MTR 的大小

日志刷盘

redo log 需要在事务提交时将日志写入磁盘, 但是比 Buffer Pool 修改的数据写入磁盘的速度快, 原因:

- 刷脏是随机 IO, 因为每次修改的数据位置随机; redo log 和 binlog 都是**顺序写**, 磁盘的顺序 IO 比随机 IO 速度要快
- 刷脏是以数据页 (Page) 为单位的, 一个页上的一个小修改都要整页写入; redo log 中只包含真正需要写入的部分, 好几页的数据修改可能只记录在一个 redo log 页中, 减少无效 IO
- **组提交机制**, 可以大幅度降低磁盘的 IO 消耗

InnoDB 引擎会在适当的时候, 把内存中 redo log buffer 持久化 (fsync) 到磁盘, 具体的**刷盘策略**:

- 在事务提交时需要进行刷盘, 通过修改参数 `innodb_flush_log_at_trx_commit` 设置:
 - 0: 表示当提交事务时, 并不将缓冲区的 redo 日志写入磁盘, 而是等待**后台线程每秒刷新一次**
 - 1: 在事务提交时将缓冲区的 redo 日志**同步写入**到磁盘, 保证一定会写入成功 (默认值)
 - 2: 在事务提交时将缓冲区的 redo 日志异步写入到磁盘, 不能保证提交时肯定会写入, 只是有这个动作。日志已经在操作系统的缓存, 如果操作系统没有宕机而 MySQL 宕机, 也是可以恢复数据的
- 写入 redo log buffer 的日志超过了总容量的一半, 就会将日志刷入到磁盘文件, 这会影响执行效率, 所以开发中应**避免大事务**
- 服务器关闭时
- 并行的事务提交 (组提交) 时, 会将其他事务的 redo log 持久化到磁盘。假设事务 A 已经写入 redo log buffer 中, 这时另外一个线程的事务 B 提交, 如果 `innodb_flush_log_at_trx_commit` 设置的是 1, 那么事务 B 要把 redo log buffer 里的日志全部持久化到磁盘, **因为多个事务共用一个 redo log buffer**, 所以一次 fsync 可以刷盘多个事务的 redo log, 提升了并发量

服务器启动后 redo 磁盘空间不变, 所以 redo 磁盘中的日志文件是被**循环使用的**, 采用循环写数据的方式, 写完尾部重新写头部, 所以要确保头部 log 对应的修改已经持久化到磁盘

日志序号

lsn (log sequence number) 代表已经写入的 redo 日志量、flushed_to_disk_lsn 指刷新到磁盘中的 redo 日志量，两者都是全局变量，如果两者的值相同，说明 log buffer 中所有的 redo 日志都已经持久化到磁盘

工作过程：写入 log buffer 数据时，buf_free 会进行偏移，偏移量就会加到 lsn 上

MTR 的执行过程中修改过的页对应的控制块会加到 Buffer Pool 的 flush 链表中，链表中脏页是按照第一次修改的时间进行排序的（头插），控制块中有两个指针用来记录脏页被修改的时间：

- oldest_modification：第一次修改 Buffer Pool 中某个缓冲页时，将修改该页的 MTR 开始时对应的 lsn 值写入这个属性
- newest_modification：每次修改页面，都将 MTR 结束时全局的 lsn 值写入这个属性，所以该值是该页面最后一次修改后的 lsn 值

全局变量 checkpoint_lsn 表示当前系统可以被覆盖的 redo 日志总量，当 redo 日志对应的脏页已经被刷新到磁盘后，该文件空间就可以被覆盖重用，此时执行一次 checkpoint 来更新 checkpoint_lsn 的值存入管理信息（刷脏和执行一次 checkpoint 并不是同一个线程），该值的增量就代表磁盘文件中当前位置向后可以被覆盖的文件的量，所以该值是一直增大的

checkpoint：从 flush 链表尾部中找出还未刷脏的页面，该页面是当前系统中最早被修改的脏页，该页面之前产生的脏页都已经刷脏，然后将该页 oldest_modification 值赋值给 checkpoint_lsn，因为 lsn 小于该值时产生的 redo 日志都可以被覆盖了

但是在系统忙碌时，后台线程的刷脏操作不能将脏页快速刷出，导致系统无法及时执行 checkpoint，这时需要用户线程从 flush 链表中把最早修改的脏页刷新到磁盘中，然后执行 checkpoint

```
write pos ----- checkpoint_lsn // 两值之间的部分表示可以写入的日志量，当 pos 追赶上
lsn 时必须执行 checkpoint
```

使用命令可以查看当前 InnoDB 存储引擎各种 lsn 的值：

```
SHOW ENGINE INNODB STATUS\G
```

崩溃恢复

恢复的起点：在从 redo 日志文件组的管理信息中获取最近发生 checkpoint 的信息，从 checkpoint_lsn 对应的日志文件开始恢复

恢复的终点：扫描日志文件的 block，block 的头部记录着当前 block 使用了多少字节，填满的 block 总是 512 字节，如果某个 block 不是 512 字节，说明该 block 就是需要恢复的最后一个 block

恢复的过程：按照 redo log 依次执行恢复数据，优化方式

- 使用哈希表：根据 redo log 的 space id 和 page number 属性计算出哈希值，将对同一页面的修改放入同一个槽里，可以一次性完成对某页的恢复，避免了随机 IO
- 跳过已经刷新到磁盘中的页面：数据页的 File Header 中的 FILE_PAGE_LSN 属性（类似 newest_modification）表示最近一次修改页面时的 lsn 值，数据页被刷新到磁盘中，那么该页 lsn 属性肯定大于 checkpoint_lsn

工作流程

日志对比

MySQL 中还存在 binlog（二进制日志）也可以记录写操作并用于数据的恢复，**保证数据不丢失**，二者区别的区别是：

- 作用不同：redo log 是用于 crash recovery（故障恢复），保证 MySQL 容机也不会影响持久性；binlog 是用于 point-in-time recovery 的，保证服务器可以基于时间点恢复数据，此外 binlog 还用于主从复制
- 层次不同：redo log 是 InnoDB 存储引擎实现的，而 binlog 是 MySQL 的 Server 层实现的，同时支持 InnoDB 和其他存储引擎
- 内容不同：redo log 是物理日志，内容基于磁盘的 Page；binlog 的内容是二进制的，根据 binlog_format 参数的不同，可能基于 SQL 语句、基于数据本身或者二者的混合（日志部分详解）
- 写入时机不同：binlog 在事务提交时一次写入；redo log 的写入时机相对多元

binlog 为什么不支持崩溃恢复？

- binlog 记录的是语句，并不记录数据页级的数据（哪个页改了哪些地方），所以没有能力恢复数据页
 - binlog 是追加写，保存全量的日志，没有标志确定从哪个点开始的数据是已经刷盘了，而 redo log 只要在 checkpoint_lsn 后面的就是没有刷盘的
-

更新记录

更新一条记录的过程：写之前一定先读

- 在 B+ 树中定位到该记录，如果该记录所在的页面不在 Buffer Pool 里，先将其加载进内存
- 首先更新该记录对应的聚簇索引，更新聚簇索引记录时：
 - 更新记录前向 undo 页面写 undo 日志，由于这是更改页面，所以需要记录一下相应的 redo 日志
注意：修改 undo 页面也是在**修改页面**，事务只要修改页面就需要先记录相应的 redo 日志
 - 然后记录对应的 redo 日志（等待 MTR 提交后写入 redo log buffer），**最后进行真正的更新记录**
- 更新其他的二级索引记录，不会再记录 undo log，只记录 redo log 到 buffer 中
- 在一条更新语句执行完成后（也就是将所有待更新记录都更新完了），就会开始记录该语句对应的 binlog 日志，此时记录的 binlog 并没有刷新到硬盘上，还在内存中，在事务提交时才会统一将该事务运行过程中的所有 binlog 日志刷新到硬盘

假设表中有字段 id 和 a，存在一条 `id = 1, a = 2` 的记录，此时执行更新语句：

```
update table set a=2 where id=1;
```

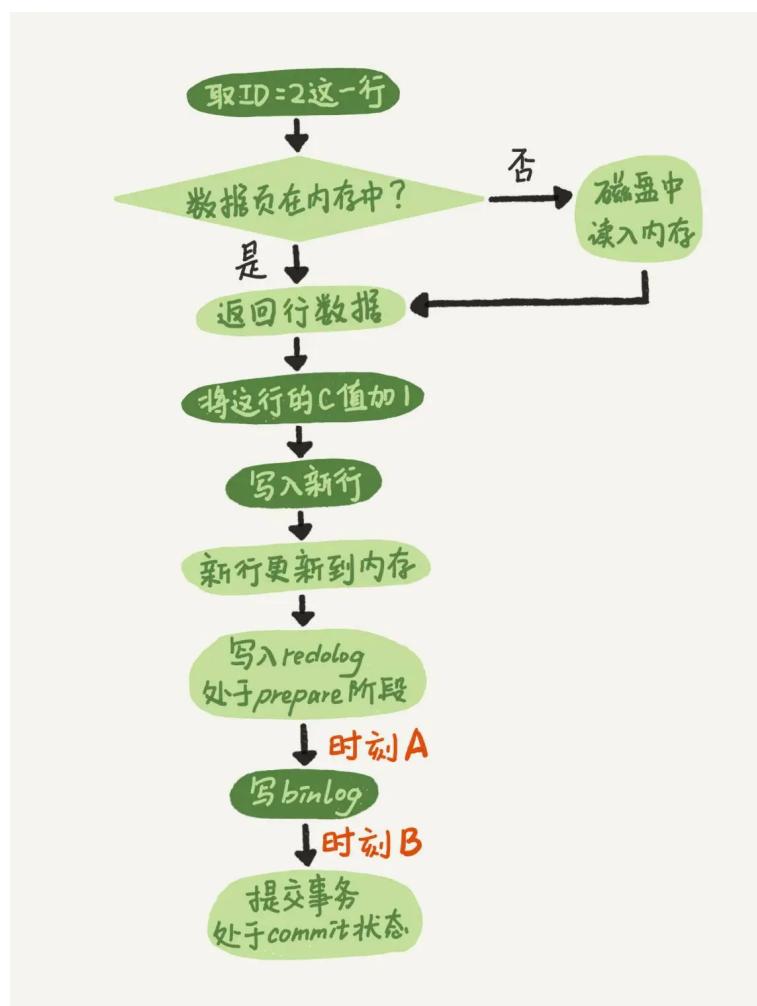
InnoDB 会真正的去执行把值修改成 (1,2) 这个操作，先加行锁，在去更新，并不会提前判断相同就不修改了

参考文章：<https://mp.weixin.qq.com/s/wcj2KisSaMnfP4nH5NYaQA>

两段提交

当客户端执行 COMMIT 语句或者在自动提交的情况下，MySQL 内部开启一个 XA 事务，分两阶段来完成 XA 事务的提交：

```
update T set c=c+1 where ID=2;
```



流程说明：执行引擎将这行新数据读入到内存中（Buffer Pool）后，先将此次更新操作记录到 redo log buffer 里，然后更新记录。最后将 redo log 刷盘后事务处于 prepare 状态，执行器会生成这个操作的 binlog，并把 binlog 写入磁盘，完成提交

两阶段：

- Prepare 阶段：存储引擎将该事务的 **redo 日志刷盘**，并且将本事务的状态设置为 PREPARE，代表执行完成随时可以提交事务
- Commit 阶段：先将事务执行过程中产生的 binlog 刷新到硬盘，再执行存储引擎的提交工作，引擎把 redo log 改成提交状态

存储引擎层的 redo log 和 server 层的 binlog 可以认为是一个分布式事务，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**，也有利于主从复制，更好的保持主从数据的一致性

数据恢复

系统崩溃前没有提交的事务的 redo log 可能已经刷盘（定时线程或者 checkpoint），怎么处理崩溃恢复？

工作流程：获取 undo 链表首节点页面的 undo segment header 中的 TRX_UNDO_STATE 属性，表示当前链表的事务属性，**事务状态是活跃（未提交）的就全部回滚**，如果是 PREPARE 状态，就需要根据 binlog 的状态进行判断：

- 如果在时刻 A 发生了崩溃（crash），由于此时 binlog 还没完成，所以需要进行回滚
- 如果在时刻 B 发生了崩溃，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 redo log 里面的事务是完整的，也就是已经有了 commit 标识，说明 binlog 也已经记录完整，直接从 redo log 恢复数据
 - 如果 redo log 里面的事务只有 prepare，就根据 XID 去 binlog 中判断对应的事务是否存在并完整，如果完整可以恢复数据

判断一个事务的 binlog 是否完整的方法：

- statement 格式的 binlog，最后会有 COMMIT
- row 格式的 binlog，最后会有一个 XID event
- MySQL 5.6.2 版本以后，引入了 binlog-checksum 参数用来验证 binlog 内容的正确性（可能日志中间出错）

参考文章：<https://time.geekbang.org/column/article/73161>

刷脏优化

系统在进行刷脏时会占用一部分系统资源，会影响系统的性能，**产生系统抖动**

- 一个查询要淘汰的脏页个数太多，会导致查询的响应时间明显变长
- 日志写满，更新全部堵住，写性能跌为 0，这种情况对敏感业务来说，是不能接受的

InnoDB 刷脏页的控制策略：

- `innodb_io_capacity` 参数代表磁盘的读写能力，建议设置成磁盘的 IOPS（每秒的 IO 次数）

- 刷脏速度参考两个因素：脏页比例和 redo log 写盘速度
 - 参数 `innodb_max_dirty_pages_pct` 是脏页比例上限，默认值是 75%，InnoDB 会根据当前的脏页比例，算出一个范围在 0 到 100 之间的数字
 - InnoDB 每次写入的日志都有一个序号，当前写入的序号跟 checkpoint 对应的序号之间的差值，InnoDB 根据差值算出一个范围在 0 到 100 之间的数字
 - 两者较大的值记为 R，执行引擎按照 `innodb_io_capacity` 定义的能力乘以 R% 来控制刷脏页的速度
 - `innodb_flush_neighbors` 参数置为 1 代表控制刷脏时检查相邻的数据页，如果也是脏页就一起刷脏，并检查邻居的邻居，这个行为会一直蔓延直到不是脏页，在 MySQL 8.0 中该值的默认值是 0，不建议开启此功能
-

一致特性

一致性是指事务执行前后，数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。

数据库的完整性约束包括但不限于：实体完整性（如行的主键存在且唯一）、列完整性（如字段的类型、大小、长度要符合要求）、外键约束、用户自定义完整性（如转账前后，两个账户余额的和应该不变）

实现一致性的措施：

- 保证原子性、持久性和隔离性，如果这些特性无法保证，事务的一致性也无法保证
 - 数据库本身提供保障，例如不允许向整形列插入字符串值、字符串长度不能超过列的限制等
 - 应用层面进行保障，例如如果转账操作只扣除转账者的余额，而没有增加接收者的余额，无论数据库实现的多么完美，也无法保证状态的一致
-

锁机制

基本介绍

锁机制：数据库为了保证数据的一致性，在共享的资源被并发访问时变得安全有序所设计的一种规则

利用 MVCC 性质进行读取的操作叫**一致性读**，读取数据前加锁的操作叫**锁定读**

锁的分类：

- 按操作分类：
 - 共享锁：也叫读锁。对同一份数据，多个事务读操作可以同时加锁而不互相影响，但不能修改数据
 - 排他锁：也叫写锁。当前的操作没有完成前，会阻断其他操作的读取和写入

- 按粒度分类：
 - 表级锁：会锁定整个表，开销小，加锁快；不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低，偏向 MyISAM
 - 行级锁：会锁定当前操作行，开销大，加锁慢；会出现死锁；锁定力度小，发生锁冲突概率低，并发度高，偏向 InnoDB
 - 页级锁：锁的力度、发生冲突的概率和加锁开销介于表锁和行锁之间，会出现死锁，并发性能一般
- 按使用方式分类：
 - 悲观锁：每次查询数据时都认为别人会修改，很悲观，所以查询时加锁
 - 乐观锁：每次查询数据时都认为别人不会修改，很乐观，但是更新时会判断一下在此期间别人有没有去更新这个数据
- 不同存储引擎支持的锁

| 存储引擎 | 表级锁 | 行级锁 | 页级锁 |
|--------|-----------|-----------|-----|
| MyISAM | 支持 | 不支持 | 不支持 |
| InnoDB | 支持 | 支持 | 不支持 |
| MEMORY | 支持 | 不支持 | 不支持 |
| BDB | 支持 | 不支持 | 支持 |

从锁的角度来说：表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如 Web 应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并查询的应用，如一些在线事务处理系统

内存结构

对一条记录加锁的本质就是在**内存中**创建一个锁结构与之关联，结构包括

- 事务信息：锁对应的事务信息，一个锁属于一个事务
- 索引信息：对于行级锁，需要记录加锁的记录属于哪个索引
- 表锁和行锁信息：表锁记录着锁定的表，行锁记录了 Space ID 所在表空间、Page Number 所在的页号、n_bits 使用了多少比特
- type_mode：一个 32 比特的数，被分成 lock_mode、lock_type、rec_lock_type 三个部分
 - lock_mode：锁模式，记录是共享锁、排他锁、意向锁之类
 - lock_type：代表表级锁还是行级锁
 - rec_lock_type：代表行锁的具体类型和 is_waiting 属性，is_waiting = true 时表示当前事务尚未获取到锁，处于等待状态。事务获取锁后的锁结构是 is_waiting 为 false，释放锁时会检查是否与当前记录关联的锁结构，如果有就唤醒对应事务的线程

一个事务可能操作多条记录，为了节省内存，满足下面条件的锁使用同一个锁结构：

- 在同一个事务中的加锁操作
- 被加锁的记录在同一个页面中
- 加锁的类型是一样的

- 加锁的状态是一样的
-

Server

MySQL 里面表级别的锁有两种：一种是表锁，一种是元数据锁（meta data lock，MDL）

MDL 叫元数据锁，主要用来保护 MySQL 内部对象的元数据，保证数据读写的正确性，**当对一个表做增删改查的时候，加 MDL 读锁；当要对表做结构变更操作 DDL 的时候，加 MDL 写锁**，两种锁不相互兼容，所以可以保证 DDL、DML、DQL 操作的安全

说明：DDL 操作执行前会隐式提交当前会话的事务，因为 DDL 一般会在若干个特殊事务中完成，开启特殊事务前需要提交到其他事务

MDL 锁的特性：

- MDL 锁不需要显式使用，在访问一个表的时候会被自动加上，在事务开始时申请，整个事务提交后释放（执行完单条语句不释放）
- MDL 锁是在 Server 中实现，不是 InnoDB 存储引擎层能直接实现的锁
- MDL 锁还能实现其他粒度级别的锁，比如全局锁、库级别的锁、表空间级别的锁

FLUSH TABLES WITH READ LOCK 简称（FTWRL），全局读锁，让整个库处于只读状态，DDL DML 都被阻塞，工作流程：

1. 上全局读锁（lock_global_read_lock）
2. 清理表缓存（close_cached_tables）
3. 上全局 COMMIT 锁（make_global_read_lock_block_commit）

该命令主要用于备份工具做**一致性备份**，由于 FTWRL 需要持有两把全局的 MDL 锁，并且还要关闭所有表对象，因此杀伤性很大

MyISAM

表级锁

MyISAM 存储引擎只支持表锁，这也是 MySQL 开始几个版本中唯一支持的锁类型

MyISAM 引擎在执行查询语句之前，会**自动**给涉及到的所有表加读锁，在执行增删改之前，会**自动**给涉及的表加写锁，这个过程并不需要用户干预，所以用户一般不需要直接用 LOCK TABLE 命令给 MyISAM 表显式加锁

- 加锁命令：（对 InnoDB 存储引擎也适用）

读锁：所有连接只能读取数据，不能修改

写锁：其他连接不能查询和修改数据

```
-- 读锁
LOCK TABLE table_name READ;

-- 写锁
LOCK TABLE table_name WRITE;
```

- 解锁命令：

```
-- 将当前会话所有的表进行解锁
UNLOCK TABLES;
```

锁的兼容性：

- 对 MyISAM 表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求
- 对 MyISAM 表的写操作，则会阻塞其他用户对同一表的读和写操作

| 请求锁模式 当前锁模式 | None | 读锁 | 写锁 |
|----------------|------|----|----|
| 读锁 | 是 | 是 | 否 |
| 写锁 | 是 | 否 | 否 |

锁调度：MyISAM 的读写锁调度是写优先，因为写锁后其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永远阻塞，所以 MyISAM 不适合做写为主的表的存储引擎

锁操作

读锁

两个客户端操作 Client 1 和 Client 2，简化为 C1、C2

- 数据准备：

```
CREATE TABLE `tb_book` (
  `id` INT(11) AUTO_INCREMENT,
  `name` VARCHAR(50) DEFAULT NULL,
  `publish_time` DATE DEFAULT NULL,
  `status` CHAR(1) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MYISAM DEFAULT CHARSET=utf8 ;

INSERT INTO tb_book (id, NAME, publish_time, STATUS) VALUES(NULL, 'java编程思想', '2088-08-01', '1');
INSERT INTO tb_book (id, NAME, publish_time, STATUS) VALUES(NULL, 'mysql编程思想', '2088-08-08', '0');
```

- C1、C2 加读锁，同时查询可以正常查询出数据

```
LOCK TABLE tb_book READ;      -- C1、C2
SELECT * FROM tb_book;        -- C1、C2
```

```
mysql> LOCK TABLE tb_book READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tb_book;
+----+-----+-----+
| id | name | publish_time | status |
+----+-----+-----+
| 1  | java编程思想 | 2088-08-01 | 1   |
| 2  | mysql编程思想 | 2088-08-08 | 0   |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql> LOCK TABLE tb_book READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tb_book;
+----+-----+-----+
| id | name | publish_time | status |
+----+-----+-----+
| 1  | java编程思想 | 2088-08-01 | 1   |
| 2  | mysql编程思想 | 2088-08-08 | 0   |
+----+-----+-----+
2 rows in set (0.00 sec)
```

- C1 加读锁，C1、C2 查询未锁定的表，C1 报错，C2 正常查询

```
LOCK TABLE tb_book READ;      -- C1
SELECT * FROM tb_user;        -- C1、C2
```

```
mysql> LOCK TABLE tb_book READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tb_user;
ERROR 1100 (HY000): Table 'tb_user' was not locked with LOCK TABLES
mysql>
mysql>
mysql>
```

```
mysql> SELECT * FROM tb_user;
+----+-----+
| id | name |
+----+-----+
| 1  | 令狐冲 |
| 2  | 田伯光 |
+----+-----+
2 rows in set (0.06 sec)
```

C1、C2 执行插入操作，C1 报错，C2 等待获取

```
INSERT INTO tb_book VALUES(NULL, 'Spring高级', '2088-01-01', '1');    -- C1、C2
```

```
mysql> INSERT INTO tb_book VALUES(NULL, 'Spring高级', '2088-01-01', '1');
ERROR 1099 (HY000): Table 'tb_book' was locked with a READ lock and
can't be updated
mysql>
```

```
mysql>
mysql>
mysql> INSERT INTO tb_book VALUES(NULL, 'Spring高级', '2088-01-01', '1');
mysql>
```

当在 C1 中释放锁指令 UNLOCK TABLES，C2 中的 INSERT 语句立即执行

写锁

两个客户端操作 Client 1 和 Client 2，简化为 C1、C2

- C1 加写锁，C1、C2 查询表，C1 正常查询，C2 需要等待

```
LOCK TABLE tb_book WRITE;      -- C1
SELECT * FROM tb_book;        -- C1、C2
```

```
mysql> LOCK TABLE tb_book WRITE;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM tb_book;
+----+-----+-----+
| id | name | publish_time | status |
+----+-----+-----+
| 1  | java编程思想 | 2088-08-01 | 1   |
| 2  | mysql编程思想 | 2088-08-08 | 0   |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

```
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> SELECT * FROM tb_book;
mysql>
```

当在 C1 中释放锁指令 UNLOCK TABLES，C2 中的 SELECT 语句立即执行

- C1、C2 同时加写锁

```
LOCK TABLE tb_book WRITE;
```

```

mysql> LOCK TABLE tb_book WRITE;
Query OK, 0 rows affected (0.00 sec)

mysql> 2 rows in set (0.00 sec)
mysql> LOCK TABLE tb_book WRITE;

```

- C1 加写锁，C1、C2查询未锁定的表，C1 报错，C2 正常查询

锁状态

- 查看锁竞争：

```
SHOW OPEN TABLES;
```

| | Database | Table | In_use | Name_locked |
|---|----------|---------------|--------|-------------|
| | mysql | servers | 0 | 0 |
| | mysql | gtid_executed | 0 | 0 |
| ✓ | db3 | tb_book | 0 | 0 |

In_use：表当前被查询使用的次数，如果该数为零，则表是打开的，但是当前没有被使用

Name_locked：表名称是否被锁定，名称锁定用于取消表或对表进行重命名等操作

```
LOCK TABLE tb_book READ; -- 执行命令
```

| | Database | Table | In_use | Name_locked |
|---|----------|---------------|--------|-------------|
| | mysql | servers | 0 | 0 |
| | mysql | gtid_executed | 0 | 0 |
| ✓ | db3 | tb_book | 1 | 0 |

- 查看锁状态：

```
SHOW STATUS LIKE 'Table_locks%';
```

| Variable_name | Value |
|-----------------------|-------|
| Table_locks_immediate | 268 |
| Table_locks_waited | 0 |

Table_locks_immediate：指的是能立即获得表级锁的次数，每立即获取锁，值加 1

Table_locks_waited：指的是不能立即获取表级锁而需要等待的次数，每等待一次，该值加 1，此值高说明存在着较为严重的表级锁争用情况

InnoDB

行级锁

记录锁

InnoDB 与 MyISAM 的最大不同有两点：一是支持事务；二是采用了行级锁，**InnoDB 同时支持表锁和行锁**

行级锁，也称为记录锁（Record Lock），InnoDB 实现了以下两种类型的行锁：

- 共享锁（S）：又称为读锁，简称 S 锁，多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改
- 排他锁（X）：又称为写锁，简称 X 锁，不能与其他锁并存，获取排他锁的事务是可以对数据读取和修改

RR 隔离级别下，对于 UPDATE、DELETE 和 INSERT 语句，InnoDB 会**自动给涉及数据集加排他锁**（行锁），在 commit 时自动释放；对于普通 SELECT 语句，不会加任何锁（只是针对 InnoDB 层来说的，因为在 Server 层会加 MDL 读锁），通过 MVCC 防止并发冲突

在事务中加的锁，并不是不需要了就释放，而是在事务中止或提交时自动释放，这个就是**两阶段锁协议**。所以一般将更新共享资源（并发高）的 SQL 放到事务的最后执行，可以让其他线程尽量的减少等待时间

锁的兼容性：

- 共享锁和共享锁 兼容
- 共享锁和排他锁 冲突
- 排他锁和排他锁 冲突
- 排他锁和共享锁 冲突

显式给数据集加共享锁或排他锁：**加锁读就是当前读，读取的是最新数据**

```
SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE -- 共享锁  
SELECT * FROM table_name WHERE ... FOR UPDATE -- 排他锁
```

注意：**锁默认会锁聚簇索引（锁就是加在索引上）**，但是当使用覆盖索引时，加共享锁只锁二级索引，不锁聚簇索引

锁操作

两个客户端操作 Client 1 和 Client 2，简化为 C1、C2

- 环境准备

```

CREATE TABLE test_innodb_lock(
    id INT(11),
    name VARCHAR(16),
    sex VARCHAR(1)
)ENGINE = INNODB DEFAULT CHARSET=utf8;

INSERT INTO test_innodb_lock VALUES(1, '100', '1');
-- ......

CREATE INDEX idx_test_innodb_lock_id ON test_innodb_lock(id);
CREATE INDEX idx_test_innodb_lock_name ON test_innodb_lock(name);

```

- 关闭自动提交功能:

```
SET AUTOCOMMIT=0; -- C1、C2
```

正常查询数据:

```
SELECT * FROM test_innodb_lock; -- C1、C2
```

- 查询 id 为 3 的数据，正常查询:

```
SELECT * FROM test_innodb_lock WHERE id=3; -- C1、C2
```

```
mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 3   | 1   |
+----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 3   | 1   |
+----+-----+-----+
1 row in set (0.05 sec)
```

- C1 更新 id 为 3 的数据，但不提交:

```
UPDATE test_innodb_lock SET name='300' WHERE id=3; -- C1
```

```
mysql> UPDATE test_innodb_lock SET name='300' WHERE id=3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
mysql>
```

```
mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 3   | 1   |
+----+-----+-----+
```

C2 查询不到 C1 修改的数据，因为隔离级别为 REPEATABLE READ，C1 提交事务，C2 查询:

```
COMMIT; -- C1
```

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
mysql>
```

```
mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 3   | 1   |
+----+-----+-----+
```

提交后仍然查询不到 C1 修改的数据，因为隔离级别可以防止脏读、不可重复读，所以 C2 需要提交才可以查询到其他事务对数据的修改:

```
COMMIT; -- C2
SELECT * FROM test_innodb_lock WHERE id=3; -- C2
```

```

mysql>
mysql>
mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 300  | 1   |
+----+-----+-----+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM test_innodb_lock WHERE id=3;
+----+-----+-----+
| id | name | sex |
+----+-----+-----+
| 3  | 300  | 1   |
+----+-----+-----+

```

- C1 更新 id 为 3 的数据，但不提交，C2 也更新 id 为 3 的数据：

```

UPDATE test_innodb_lock SET name='3' WHERE id=3;    -- C1
UPDATE test_innodb_lock SET name='30' WHERE id=3;    -- C2

```

```

mysql> UPDATE test_innodb_lock SET name='3' WHERE id=3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql>
mysql> UPDATE test_innodb_lock SET name='30' WHERE id=3;

```

当 C1 提交，C2 直接解除阻塞，直接更新

- 操作不同行的数据：

```

UPDATE test_innodb_lock SET name='10' WHERE id=1;    -- C1
UPDATE test_innodb_lock SET name='30' WHERE id=3;    -- C2

```

```

mysql> UPDATE test_innodb_lock SET name='10' WHERE id=1;
Query OK, 2 rows affected (0.01 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> UPDATE test_innodb_lock SET name='30' WHERE id=3;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql>

```

由于 C1、C2 操作的不同行，获取不同的行锁，所以都可以正常获取行锁

锁分类

间隙锁

InnoDB 会对间隙（GAP）进行加锁，就是间隙锁（RR 隔离级别下才有该锁）。间隙锁之间不存在冲突关系，**多个事务可以同时对一个间隙加锁**，但是间隙锁会阻止往这个间隙中插入一个记录的操作

InnoDB 加锁的基本单位是 next-key lock，该锁是行锁和 gap lock 的组合（X or S 锁），但是加锁过程是分为间隙锁和行锁两段执行

- 可以保护当前记录和前面的间隙**，遵循左开右闭原则，单纯的间隙锁是左开右开
- 假设 10、11、13，那么可能的间隙锁包括：(负无穷,10]、(10,11]、(11,13]、(13,正无穷)

几种索引的加锁情况：

- 唯一索引加锁在值存在时是行锁，next-key lock 会退化为行锁，值不存在会变成间隙锁
- 普通索引加锁会继续向右遍历到不满足条件的值为止，next-key lock 退化为间隙锁
- 范围查询无论是否是唯一索引，都需要访问到不满足条件的第一个值为止
- 对于联合索引且是唯一索引，如果 where 条件只包括联合索引的一部分，那么会加间隙锁

间隙锁优点：RR 级别下间隙锁可以**解决事务的一部分的幻读问题**，通过对间隙加锁，可以防止读取过程中数据条目发生变化。一部分的意思是不会对全部间隙加锁，只能加锁一部分的间隙

间隙锁危害：

- 当锁定一个范围的键值后，即使某些不存在的键值也会被无辜的锁定，造成在锁定的时候无法插入锁定键值范围内的任何数据，在某些场景下这可能会对性能造成很大的危害，影响并发度
- 事务 A B 同时锁住一个间隙后，A 往当前间隙插入数据时会被 B 的间隙锁阻塞，B 也执行插入间隙数据的操作时就会产生死锁

现场演示：

- 关闭自动提交功能：

```
SET AUTOCOMMIT=0; -- C1、C2
```

- 查询数据表：

```
SELECT * FROM test_innodb_lock;
```

```
mysql> SELECT * FROM test_innodb_lock;
+----+----+----+
| id | name | sex |
+----+----+----+
| 1  | 10  | 2   |
| 3  | 30  | 1   |
| 4  | 400 | 0   |
+----+----+----+
```

- C1 根据 id 范围更新数据，C2 插入数据：

```
UPDATE test_innodb_lock SET name='8888' WHERE id < 4; -- C1
INSERT INTO test_innodb_lock VALUES(2,'200','2'); -- C2
```

出现间隙锁，C2 被阻塞，等待 C1 提交事务后才能更新

意向锁

InnoDB 为了支持多粒度的加锁，允许行锁和表锁同时存在，支持在不同粒度上的加锁操作，InnoDB 增加了意向锁（Intention Lock）

意向锁是将锁定的对象分为多个层次，意向锁意味着事务希望在更细粒度上进行加锁，意向锁分为两种：

- 意向共享锁（IS）：事务有意向对表加共享锁
- 意向排他锁（IX）：事务有意向对表加排他锁

IX, IS 是表级锁，不会和行级的 X, S 锁发生冲突，意向锁是在加表级锁之前添加，为了在加表级锁时可以快速判断表中是否有记录被上锁，比如向一个表添加表级 X 锁的时：

- 没有意向锁，则需要遍历整个表判断是否有锁定的记录
- 有了意向锁，首先判断是否存在意向锁，然后判断该意向锁与即将添加的表级锁是否兼容即可，因为意向锁的存在代表有表级锁的存在或者即将有表级锁的存在

兼容性如下所示：

| 兼容性 | IS | IX | S | X |
|-----|-----|-----|-----|-----|
| IS | 兼容 | 兼容 | 兼容 | 不兼容 |
| IX | 兼容 | 兼容 | 不兼容 | 不兼容 |
| S | 兼容 | 不兼容 | 兼容 | 不兼容 |
| X | 不兼容 | 不兼容 | 不兼容 | 不兼容 |

插入意向锁 Insert Intention Lock 是在插入一行记录操作之前设置的一种间隙锁，是行级锁

插入意向锁释放了一种插入信号，即多个事务在相同的索引间隙插入时如果不是插入相同的间隙位置就不需要互相等待。假设某列有索引，只要两个事务插入位置不同，如事务 A 插入 3，事务 B 插入 4，那么就可以同时插入

自增锁

系统会自动给 AUTO_INCREMENT 修饰的列进行递增赋值，实现方式：

- AUTO_INC 锁：表级锁，执行插入语句时会自动添加，在该语句执行完成后释放，并不是事务结束
- 轻量级锁：为插入语句生成 AUTO_INCREMENT 修饰的列时获取该锁，生成以后释放掉，不需要等到插入语句执行完后释放

系统变量 `innodb_autoinc_lock_mode` 控制采取哪种方式：

- 0：全部采用 AUTO_INC 锁
 - 1：全部采用轻量级锁
 - 2：混合使用，在插入记录的数量确定时采用轻量级锁，不确定时采用 AUTO_INC 锁
-

隐式锁

一般情况下 INSERT 语句是不需要在内存中生成锁结构的，会进行隐式的加锁，保护的是插入后的安全

注意：如果插入的间隙被其他事务加了间隙锁，此次插入会被阻塞，并在该间隙插入一个插入意向锁

- 聚簇索引：索引记录有 `trx_id` 隐藏列，表示最后改动该记录的事务 id，插入数据后事务 id 就是当前事务。其他事务想获取该记录的锁时会判断当前记录的事务 id 是否是活跃的，如果不是就可以正常加锁；如果是就创建一个 X 的锁结构，该锁的 `is_waiting` 是 false，为自己的事务创建一个锁结构，`is_waiting` 是 true（类似 Java 中的锁升级）
- 二级索引：获取数据页 Page Header 中的 `PAGE_MAX_TRX_ID` 属性，代表修改当前页面的最大的事务 ID，如果小于当前活跃的最小微事务 id，就证明插入该数据的事务已经提交，否则就需要获取到主键值进行回表操作

隐式锁起到了延迟生成锁的效果，如果其他事务与隐式锁没有冲突，就可以避免锁结构的生成，节省了内存资源

INSERT 在两种情况下会生成锁结构：

- 重复键：在插入主键或唯一二级索引时遇到重复的键值会报错，在报错前需要对对应的聚簇索引进行加锁
 - 隔离级别 \leq Read Uncommitted，加 S 型 Record Lock
 - 隔离级别 \geq Repeatable Read，加 S 型 next_key 锁
- 外键检查：如果待插入的记录在父表中可以找到，会对父表的记录加 S 型 Record Lock。如果待插入的记录在父表中找不到
 - 隔离级别 \leq Read Committed，不加锁
 - 隔离级别 \geq Repeatable Read，加间隙锁

锁优化

优化锁

InnoDB 存储引擎实现了行级锁定，虽然在锁定机制的实现方面带来了性能损耗可能比表锁会更高，但是在整体并发处理能力方面要远远优于 MyISAM 的表锁，当系统并发量较高的时候，InnoDB 的整体性能能远远好于 MyISAM

但是使用不当可能会让 InnoDB 的整体性能表现不仅不能比 MyISAM 高，甚至可能会更差

优化建议：

- 尽可能让所有数据检索都能通过索引来完成，避免无索引行锁升级为表锁
- 合理设计索引，尽量缩小锁的范围
- 尽可能减少索引条件及索引范围，避免间隙锁
- 尽量控制事务大小，减少锁定资源量和时间长度
- 尽可使用低级别事务隔离（需要业务层面满足需求）

锁升级

索引失效造成**行锁升级为表锁**，不通过索引检索数据，全局扫描的过程中 InnoDB 会将对表中的所有记录加锁，实际效果和**表锁**一样，实际开发过程应避免出现索引失效的状况

- 查看当前表的索引：

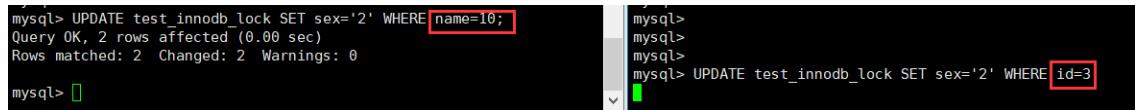
```
SHOW INDEX FROM test_innodb_lock;
```

- 关闭自动提交功能：

```
SET AUTOCOMMIT=0;    -- C1、C2
```

- 执行更新语句：

```
UPDATE test_innodb_lock SET sex='2' WHERE name=10; -- C1  
UPDATE test_innodb_lock SET sex='2' WHERE id=3; -- C2
```



```
mysql> UPDATE test_innodb_lock SET sex='2' WHERE name=10;  
Query OK, 2 rows affected (0.00 sec)  
Rows matched: 2 Changed: 2 Warnings: 0  
mysql>
```

```
mysql>  
mysql>  
mysql>  
mysql> UPDATE test_innodb_lock SET sex='2' WHERE id=3
```

索引失效：执行更新时 name 字段为 varchar 类型，造成索引失效，最终行锁变为表锁

死锁

不同事务由于互相持有对方需要的锁而导致事务都无法继续执行的情况称为死锁

死锁情况：线程 A 修改了 id = 1 的数据，请求修改 id = 2 的数据，线程 B 修改了 id = 2 的数据，请求修改 id = 1 的数据，产生死锁

解决策略：

- 直接进入等待直到超时，超时时间可以通过参数 `innodb_lock_wait_timeout` 来设置，默认 50 秒，但是时间的设置不好控制，超时可能不是因为死锁，而是因为事务处理比较慢，所以一般不采取该方式
- 主动死锁检测，发现死锁后主动回滚死锁链条中较小的一个事务，让其他事务得以继续执行，将参数 `innodb_deadlock_detect` 设置为 on，表示开启该功能（事务较小的意思就是事务执行过程中插入、删除、更新的记录条数）

死锁检测并不是每个语句都要检测，只有在加锁访问的行上已经有锁时，当前事务被阻塞了才会检测，也是从当前事务开始进行检测

通过执行 `SHOW ENGINE INNODB STATUS` 可以查看最近发生的一次死循环，全局系统变量 `innodb_print_all_deadlocks` 设置为 on，就可以将每个死锁信息都记录在 MySQL 错误日志中

死锁一般是行级锁，当表锁发生死锁时，会在事务中访问其他表时直接报错，破坏了持有并等待的死锁条件

锁状态

查看锁信息

```
SHOW STATUS LIKE 'innodb_row_lock%';
```

```

mysql> SHOW STATUS LIKE 'innodb_row_lock%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Innodb_row_lock_current_waits | 0 |
| Innodb_row_lock_time | 116865 |
| Innodb_row_lock_time_avg | 29216 |
| Innodb_row_lock_time_max | 51029 |
| Innodb_row_lock_waits | 4 |
+-----+-----+
2 rows in set (0.00 sec)

```

参数说明：

- Innodb_row_lock_current_waits：当前正在等待锁定的数量
- Innodb_row_lock_time：从系统启动到现在锁定总时间长度
- Innodb_row_lock_time_avg：每次等待所花平均时长
- Innodb_row_lock_time_max：从系统启动到现在等待最长的一次所花的时间
- Innodb_row_lock_waits：系统启动后到现在总共等待的次数

当等待的次数很高，而且每次等待的时长也不短的时候，就需要分析系统中为什么会有如此多的等待，然后根据分析结果制定优化计划

查看锁状态：

```

SELECT * FROM information_schema.innodb_locks; #锁的概况
SHOW ENGINE INNODB STATUS\G; #InnoDB整体状态，其中包括锁的情况

```

```

mysql> select * from information_schema.innodb_locks;
+-----+-----+-----+-----+-----+
| lock_id | lock trx_id | lock_mode | lock_type | lock_table |
+-----+-----+-----+-----+-----+
| 24053:99:3:2 | 24053 | X | RECORD | `test`.`account` |
| 24052:99:3:2 | 24052 | X | RECORD | `test`.`account` |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

lock_id 是锁 id；lock_trx_id 为事务 id；lock_mode 为 X 代表排它锁（写锁）；lock_type 为 RECORD 代表锁为行锁（记录锁）

乐观锁

悲观锁：在整个数据处理过程中，将数据处于锁定状态，为了保证事务的隔离性，就需要一致性锁定读。读取数据时给加锁，其它事务无法修改这些数据，修改删除数据时也加锁，其它事务同样无法读取这些数据

悲观锁和乐观锁使用前提：

- 对于读的操作远多于写的操作的时候，一个更新操作加锁会阻塞所有的读取操作，降低了吞吐量，最后需要释放锁，锁是需要一些开销的，这时候可以选择乐观锁
- 如果是读写比例差距不是非常大或者系统没有响应不及时，吞吐量瓶颈的问题，那就不要去使用乐观锁，它增加了复杂度，也带来了业务额外的风险，这时候可以选择悲观锁

乐观锁的实现方式：就是 CAS，比较并交换

- 版本号

1. 给数据表中添加一个 version 列，每次更新后都将这个列的值加 1
2. 读取数据时，将版本号读取出来，在执行更新的时候，比较版本号
3. 如果相同则执行更新，如果不相同，说明此条数据已经发生了变化
4. 用户自行根据这个通知来决定怎么处理，比如重新开始一遍，或者放弃本次更新

```
-- 创建city表
CREATE TABLE city(
    id INT PRIMARY KEY AUTO_INCREMENT,      -- 城市id
    NAME VARCHAR(20),                      -- 城市名称
    VERSION INT                           -- 版本号
);

-- 添加数据
INSERT INTO city VALUES (NULL, '北京', 1), (NULL, '上海', 1), (NULL, '广州', 1),
(NULL, '深圳', 1);

-- 修改北京为北京市
-- 1. 查询北京的version
SELECT VERSION FROM city WHERE NAME='北京';
-- 2. 修改北京为北京市，版本号+1。并对比版本号
UPDATE city SET NAME='北京市', VERSION=VERSION+1 WHERE NAME='北京' AND
VERSION=1;
```

- 时间戳

- 和版本号方式基本一样，给数据表中添加一个列，名称无所谓，数据类型需要是 timestamp
- 每次更新后都将最新时间插入到此列
- 读取数据时，将时间读取出来，在执行更新的时候，比较时间
- 如果相同则执行更新，如果不相同，说明此条数据已经发生了变化

乐观锁的异常情况：如果 version 被其他事务抢先更新，则在当前事务中更新失败，trx_id 没有变成当前事务的 ID，当前事务再次查询还是旧值，就会出现**值没变但是更新不了**的现象 (anomaly)

解决方案：每次 CAS 更新不管成功失败，就结束当前事务；如果失败则重新起一个事务进行查询更新

主从

基本介绍

主从复制是指将主数据库的 DDL 和 DML 操作通过二进制日志传到从库服务器中，然后在从库上对这些日志重新执行（也叫重做），从而使得从库和主库的数据保持同步

MySQL 支持一台主库同时向多台从库进行复制，从库同时也可作为其他从服务器的主库，实现链状复制

MySQL 复制的优点主要包含以下三个方面：

- 主库出现问题，可以快速切换到从库提供服务
- 可以在从库上执行查询操作，从主库中更新，实现读写分离
- 可以在从库中执行备份，以避免备份期间影响主库的服务（备份时会加全局读锁）

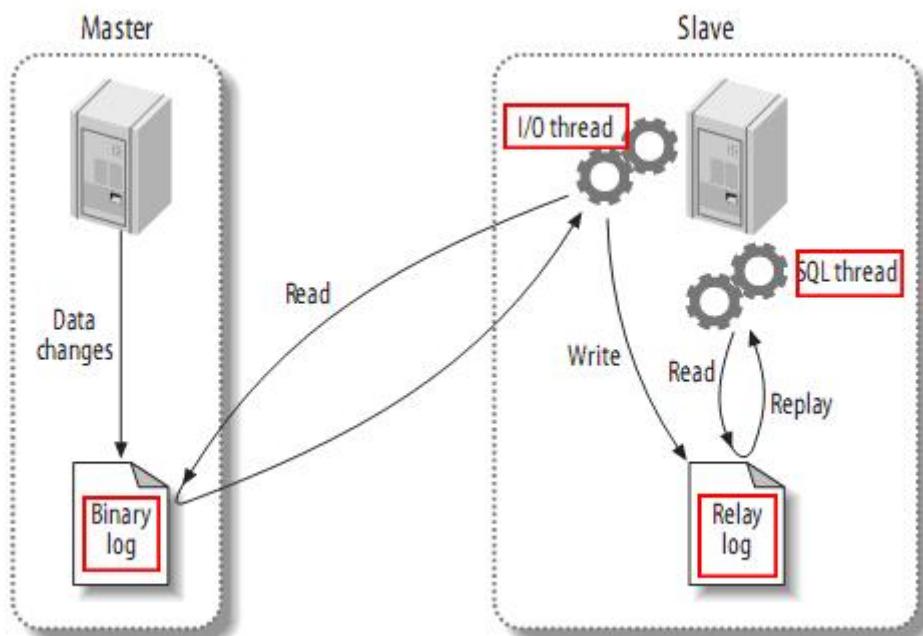
主从复制

主从结构

MySQL 的主从之间维持了一个长连接。主库内部有一个线程，专门用于服务从库的长连接，连接过程：

- 从库执行 change master 命令，设置主库的 IP、端口、用户名、密码以及要从哪个位置开始请求 binlog，这个位置包含文件名和日志偏移量
- 从库执行 start slave 命令，这时从库会启动两个线程，就是图中的 io_thread 和 sql_thread，其中 io_thread 负责与主库建立连接
- 主库校验完用户名、密码后，开始按照从传过来的位置，从本地读取 binlog 发给从库，开始主从复制

主从复制原理图：



主从复制主要依赖的是 binlog，MySQL 默认是异步复制，需要三个线程：

- binlog thread：在主库事务提交时，把数据变更记录在日志文件 binlog 中，并通知 slave 有数据更新

- I/O thread：负责从主服务器上**拉取二进制日志**，并将 binlog 日志内容依次写到 relay log 中转日志的最末端，并将新的 binlog 文件名和 offset 记录到 master-info 文件中，以便下一次读取日志时从指定 binlog 日志文件及位置开始读取新的 binlog 日志内容
- SQL thread：监测本地 relay log 中新增了日志内容，读取中继日志并重做其中的 SQL 语句，从库在 relay-log.info 中记录当前应用中继日志的文件名和位点以便下一次执行

同步与异步：

- 异步复制有数据丢失风险，例如数据还未同步到从库，主库就给客户端响应，然后主库挂了，此时从库晋升为主库的话数据是缺失的
 - 同步复制，主库需要将 binlog 复制到所有从库，等所有从库响应了之后主库才进行其他逻辑，这样的话性能很差，一般不会选择
 - MySQL 5.7 之后出现了半同步复制，有参数可以选择成功同步几个从库就返回响应
-

主主结构

主主结构就是两个数据库之间总是互为主从关系，这样在切换的时候就不用再修改主从关系

循环复制：在库 A 上更新了一条语句，然后把生成的 binlog 发给库 B，库 B 执行完这条更新语句后也会生成 binlog，会再发给 A

解决方法：

- 两个库的 server id 必须不同，如果相同则它们之间不能设定为主主关系
 - 一个库接到 binlog 并在重放的过程中，生成与原 binlog 的 server id 相同的新的 binlog
 - 每个库在收到从主库发过来的日志后，先判断 server id，如果跟自己的相同，表示这个日志是自己生成的，就直接丢弃这个日志
-

主从延迟

延迟原因

正常情况主库执行更新生成的所有 binlog，都可以传到从库并被正确地执行，从库就能达到跟主库一致的状态，这就是最终一致性

主从延迟是主从之间是存在一定时间的数据不一致，就是同一个事务在从库执行完成的时间和主库执行完成的时间的差值，即 T2-T1

- 主库 A 执行完成一个事务，写入 binlog，该时刻记为 T1
- 日志传给从库 B，从库 B 执行完这个事务，该时刻记为 T2

通过在从库执行 `show slave status` 命令，返回结果会显示 `seconds_behind_master` 表示当前从库延迟了多少秒

- 每一个事务的 binlog 都有一个时间字段，用于记录主库上写入的时间
- 从库取出当前正在执行的事务的时间字段，跟系统的时间进行相减，得到的就是 `seconds_behind_master`

主从延迟的原因：

- 从库的机器性能比主库的差，导致从库的复制能力弱
- 从库的查询压力大，建立一主多从的结构
- 大事务的执行，主库必须要等到事务完成之后才会写入 binlog，导致从节点出现应用 binlog 延迟
- 主库的 DDL，从库与主库的 DDL 同步是串行进行，DDL 在主库执行时间很长，那么从库也会消耗同样的时间
- 锁冲突问题也可能导致从节点的 SQL 线程执行慢

主从同步问题永远都是一致性和性能的权衡，需要根据实际的应用场景，可以采取下面的办法：

- 优化 SQL，避免慢 SQL，减少批量操作
- 降低多线程大事务并发的概率，优化业务逻辑
- 业务中大多数情况查询操作要比更新操作更多，搭建**一主多从**结构，让这些从库来分担读的压力
- 尽量采用短的链路，主库和从库服务器的距离尽量要短，提升端口带宽，减少 binlog 传输的网络延时
- 实时性要求高的业务读强制走主库，从库只做备份

并行复制

MySQL5.6

高并发情况下，主库的会产生大量的 binlog，在从库中有两个线程 IO Thread 和 SQL Thread 单线程执行，会导致主库延迟变大。为了改善复制延迟问题，MySQL 5.6 版本增加了并行复制功能，以采用多线程机制来促进执行

coordinator 就是原来的 SQL Thread，并行复制中它不再直接更新数据，**只负责读取中转日志和分发事务**：

- 线程分配完成并不是立即执行，为了防止造成更新覆盖，更新同一 DB 的两个事务必须被分发到同一个工作线程
- 同一个事务不能被拆开，必须放到同一个工作线程

MySQL 5.6 版本的策略：每个线程对应一个 hash 表，用于保存当前这个线程的执行队列里的事务所涉及的表，hash 表的 key 是数据库名，value 是一个数字，表示队列中有多少个事务修改这个库，适用于主库上有多个 DB 的情况

每个事务在分发的时候，跟线程的**冲突**（事务操作的是同一个库）关系包括以下三种情况：

- 如果跟所有线程都不冲突，coordinator 线程就会把这个事务分配给最空闲的线程
- 如果只跟一个线程冲突，coordinator 线程就会把这个事务分配给这个存在冲突关系的线程
- 如果跟多于一个线程冲突，coordinator 线程就进入等待状态，直到和这个事务存在冲突关系的线程只剩下 1 个

优缺点：

- 构造 hash 值的时候很快，只需要库名，而且一个实例上 DB 数也不会很多，不会出现需要构造很多项的情况
- 不要求 binlog 的格式，statement 格式的 binlog 也可以很容易拿到库名（日志章节詳解了 binlog）
- 主库上的表都放在同一个 DB 里面，这个策略就没有效果了；或者不同 DB 的热点不同，比如一个是业务逻辑库，一个是系统配置库，那也起不到并行的效果，需要**把相同热度的表均匀分到这些不**

MySQL5.7

MySQL 5.7 由参数 slave-parallel-type 来控制并行复制策略：

- 配置为 DATABASE，表示使用 MySQL 5.6 版本的**按库（DB）并行策略**
- 配置为 LOGICAL_CLOCK，表示的**按提交状态并行执行**

按提交状态并行复制策略的思想是：

- 所有处于 commit 状态的事务可以并行执行；同时处于 prepare 状态的事务，在从库执行时是可以并行的
- 处于 prepare 状态的事务，与处于 commit 状态的事务之间，在从库执行时也是可以并行的

MySQL 5.7.22 版本里，MySQL 增加了一个新的并行复制策略，基于 WRITESET 的并行复制，新增了一个参数 binlog-transaction-dependency-tracking，用来控制是否启用这个新策略：

- COMMIT_ORDER：表示根据同时进入 prepare 和 commit 来判断是否可以并行的策略
- WRITESET：表示的是对于每个事务涉及更新的每一行，计算出这一行的 hash 值，组成该事务的 writeset 集合，如果两个事务没有操作相同的行，也就是说它们的 writeset 没有交集，就可以并行（**按行并行**）

为了唯一标识，这个 hash 表的值是通过 库名 + 表名 + 索引名 + 值（表示的是某一行）计算出来的

- WRITESET_SESSION：是在 WRITESET 的基础上多了一个约束，即在主库上同一个线程先后执行的两个事务，在备库执行的时候，要保证相同的先后顺序

MySQL 5.7.22 按行并发的优势：

- writeset 是在主库生成后直接写入到 binlog 里面的，这样在备库执行的时候，不需要解析 binlog 内容，节省了计算量
- 不需要把整个事务的 binlog 都扫一遍才能决定分发到哪个线程，更省内存
- 从库的分发策略不依赖于 binlog 内容，所以 binlog 是 statement 格式也可以，更节约内存（因为 row 才记录更改的行）

MySQL 5.7.22 的并行复制策略在通用性上是有保证的，但是对于表上没主键、唯一和外键约束的场景，WRITESET 策略也是没法并行的，也会暂时退化为单线程模型

参考文章：<https://time.geekbang.org/column/article/77083>

读写分离

读写延迟

读写分离：可以降低主库的访问压力，提高系统的并发能力

- 主库不建查询的索引，从库建查询的索引。因为索引需要维护的，比如插入一条数据，不仅要在聚簇索引上面插入，对应的二级索引也得插入
- 将读操作分到从库了之后，可以在主库把查询要用的索引删了，减少写操作对主库的影响

读写分离产生了读写延迟，造成数据的不一致性。假如客户端执行完一个更新事务后马上发起查询，如果查询选择的是从库的话，可能读到的还是以前的数据，叫过期读

解决方案：

- 强制将写之后立刻读的操作转移到主库，比如刚注册的用户，直接登录从库查询可能查询不到，先走主库登录
 - 二次查询，如果从库查不到数据，则再去主库查一遍，由 API 封装，比较简单，但导致主库压力大
 - 更新主库后，读从库之前先 sleep 一下，类似于执行一条 `select sleep(1)` 命令，大多数情况下主备延迟在 1 秒之内
-

确保机制

无延迟

确保主备无延迟的方法：

- 每次从库执行查询请求前，先判断 `seconds_behind_master` 是否已经等于 0，如果不等于那就等到参数变为 0 执行查询请求
 - 对比位点，`Master_Log_File` 和 `Read_Master_Log_Pos` 表示的是读到的主库的最新位点，`Relay_Master_Log_File` 和 `Exec_Master_Log_Pos` 表示的是备库执行的最新位点，这两组值完全相同就说明接收到的日志已经同步完成
 - 对比 GTID 集合，`Retrieved_Gtid_Set` 是备库收到的所有日志的 GTID 集合，`Executed_Gtid_Set` 是备库所有已经执行完成的 GTID 集合，如果这两个集合相同也表示备库接收到的日志都已经同步完成
-

半同步

半同步复制就是 semi-sync replication，适用于一主一备的场景，工作流程：

- 事务提交的时候，主库把 binlog 发给从库
- 从库收到 binlog 以后，发回给主库一个 ack，表示收到了
- 主库收到这个 ack 以后，才能给客户端返回事务完成的确认

在一主多从场景中，主库只要等到一个从库的 ack，就开始给客户端返回确认，这时在从库上执行查询请求，有两种情况：

- 如果查询是落在这个响应了 ack 的从库上，是能够确保读到最新数据

- 如果查询落到其他从库上，它们可能还没有收到最新的日志，就会产生过期读的问题

在业务更新的高峰期，主库的位点或者 GTID 集合更新很快，导致从库来不及处理，那么两个位点等值判断就会一直不成立，很可能出现从库上迟迟无法响应查询请求的情况

等位点

在从库执行判断位点的命令，参数 file 和 pos 指的是主库上的文件名和位置，timeout 可选，设置为正整数 N 表示最多等待 N 秒

```
SELECT master_pos_wait(file, pos[, timeout]);
```

命令正常返回的结果是一个正整数 M，表示从命令开始执行，到应用完 file 和 pos 表示的 binlog 位置，执行了多少事务

- 如果执行期间，备库同步线程发生异常，则返回 NULL
- 如果等待超过 N 秒，就返回 -1
- 如果刚开始执行的时候，就已经执行过这个位置了，则返回 0

工作流程：先执行 trx1，再执行一个查询请求的逻辑，要保证能够查到正确的数据

- trx1 事务更新完成后，马上执行 `show master status` 得到当前主库执行到的 File 和 Position
- 选定一个从库执行判断位点语句，如果返回值是 $>= 0$ 的正整数，说明从库已经同步完事务，可以在这个从库执行查询语句
- 如果出现其他情况，需要到主库执行查询语句

注意：如果所有的从库都延迟超过 timeout 秒，查询压力就都跑到主库上，所以需要进行权衡

等GTID

数据库开启了 GTID 模式，MySQL 提供了判断 GTID 的命令

```
SELECT wait_for_executed_gtid_set(gtid_set [, timeout])
```

- 等待直到这个库执行的事务中包含传入的 gtid_set，返回 0
- 超时返回 1

工作流程：先执行 trx1，再执行一个查询请求的逻辑，要保证能够查到正确的数据

- trx1 事务更新完成后，从返回包直接获取这个事务的 GTID，记为 gtid
- 选定一个从库执行查询语句，如果返回值是 0，则在这个从库执行查询语句，否则到主库执行查询语句

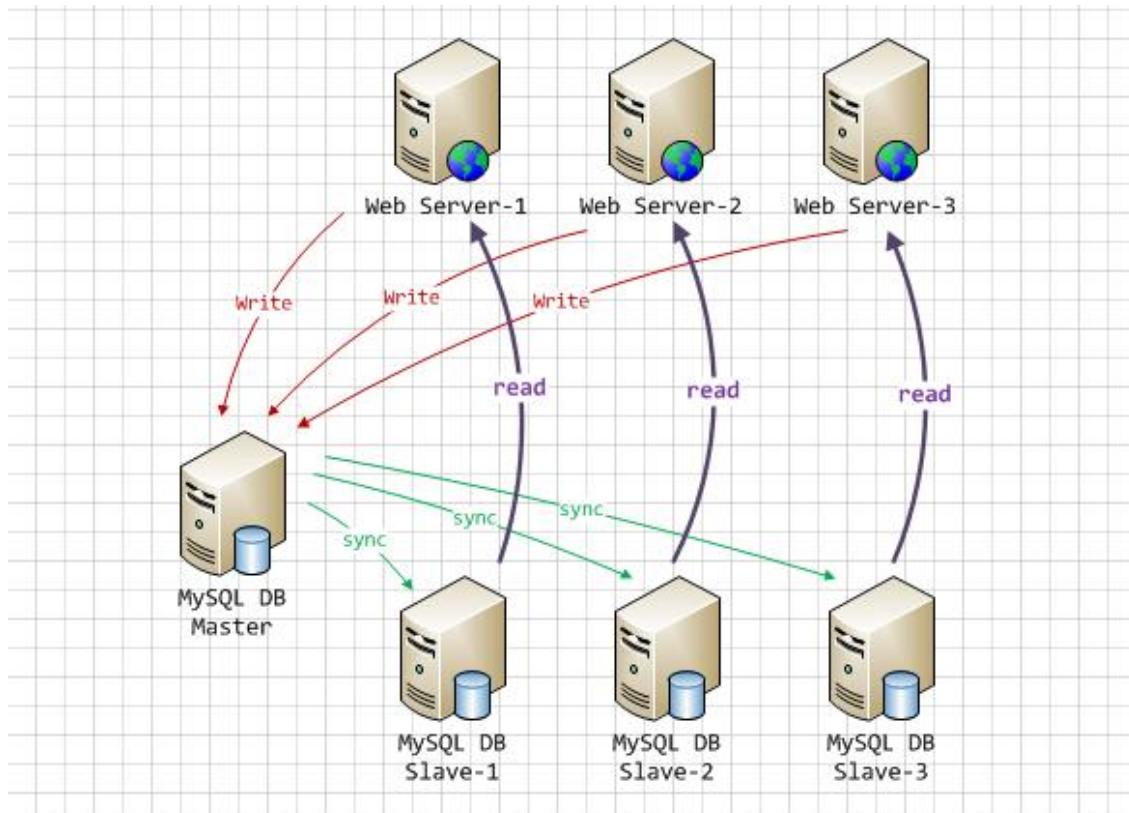
对比等待位点方法，减少了一次 `show master status` 的方法，将参数 `session_track_gtids` 设置为 `OWN_GTID`，然后通过 API 接口 `mysql_session_track_get_first` 从返回包解析出 GTID 的值即可

总结：所有的等待无延迟的方法，都需要根据具体的业务场景去判断实施

负载均衡

负载均衡是应用中使用非常普遍的一种优化方法，机制就是利用某种均衡算法，将固定的负载量分布到不同的服务器上，以此来降低单台服务器的负载，达到优化的效果

- 分流查询：通过 MySQL 的主从复制，实现读写分离，使增删改操作走主节点，查询操作走从节点，从而可以降低单台服务器的读写压力



- 分布式数据库架构：适合大数据量、负载高的情况，具有良好的拓展性和高可用性。通过在多台服务器之间分布数据，可以实现在多台服务器之间的负载均衡，提高访问效率
-

主从搭建

master

1. 在 master 的配置文件 (/etc/mysql/my.cnf) 中，配置如下内容：

```
#mysql 服务ID, 保证整个集群环境中唯一  
server-id=1
```

```
#mysql binlog 日志的存储路径和文件名
```

```
log-bin=/var/lib/mysql/mysqlbin
```

```
#错误日志，默认已经开启
```

```
#log-err
```

```
#mysql的安装目录
```

```
#basedir
```

```
#mysql的临时目录
```

```
#tmpdir
```

```
#mysql的数据存放目录
```

```
#datadir
```

```
#是否只读，1 代表只读，0 代表读写
```

```
read-only=0
```

```
#忽略的数据，指不需要同步的数据库
```

```
binlog-ignore-db=mysql
```

```
#指定同步的数据库
```

```
#binlog-do-db=db01
```

2. 执行完毕之后，需要重启 MySQL

3. 创建同步数据的账户，并且进行授权操作：

```
GRANT REPLICATION SLAVE ON *.* TO 'seazean'@'192.168.0.137' IDENTIFIED BY  
'123456';  
FLUSH PRIVILEGES;
```

4. 查看 master 状态：

```
SHOW MASTER STATUS;
```

```
mysql> show master status;  
+-----+-----+-----+-----+  
| File | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |  
+-----+-----+-----+-----+  
| mysqlbin.000001 | 413 | | mysql | |  
+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

- File：从哪个日志文件开始推送日志文件
- Position：从哪个位置开始推送日志
- Binlog_Ignore_DB：指定不需要同步的数据库

slave

1. 在 slave 端配置文件中，配置如下内容：

```
#mysql服务端ID,唯一  
server-id=2  
  
#指定binlog日志  
log-bin=/var/lib/mysql/mysqlbin
```

2. 执行完毕之后，需要重启 MySQL

3. 指定当前从库对应的主库的IP地址、用户名、密码，从哪个日志文件开始的那个位置开始同步推送日志

```
CHANGE MASTER TO MASTER_HOST= '192.168.0.138', MASTER_USER='seazeane',  
MASTER_PASSWORD='seazeane', MASTER_LOG_FILE='mysqlbin.000001',  
MASTER_LOG_POS=413;
```

4. 开启同步操作：

```
START SLAVE;  
SHOW SLAVE STATUS;
```

5. 停止同步操作：

```
STOP SLAVE;
```

验证

1. 在主库中创建数据库，创建表并插入数据：

```
CREATE DATABASE db01;  
USE db01;  
CREATE TABLE user(  
    id INT(11) NOT NULL AUTO_INCREMENT,  
    name VARCHAR(50) NOT NULL,  
    sex VARCHAR(1),  
    PRIMARY KEY (id)  
)ENGINE=INNODB DEFAULT CHARSET=utf8;  
  
INSERT INTO user(id,NAME,sex) VALUES(NULL, 'Tom', '1');  
INSERT INTO user(id,NAME,sex) VALUES(NULL, 'Trigger', '0');  
INSERT INTO user(id,NAME,sex) VALUES(NULL, 'Dawn', '1');
```

2. 在从库中查询数据，进行验证：

在从库中，可以查看到刚才创建的数据库：

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| db01          |
| mysql         |
| performance_schema |
| test          |
+-----+
5 rows in set (0.00 sec)
```

在该数据库中，查询表中的数据：

```
mysql>
mysql> select * from user;
+----+-----+-----+
| id | name  | sex  |
+----+-----+-----+
| 1  | Tom   | 1   |
| 2  | Trigger | 0   |
| 3  | Dawn  | 1   |
+----+-----+-----+
3 rows in set (0.00 sec)
```

主从切换

正常切换

正常切换步骤：

- 在开始切换之前先对主库进行锁表 `flush tables with read lock`，然后等待所有语句执行完成，切换完成后可以释放锁
- 检查 slave 同步状态，在 slave 执行 `show processlist`
- 停止 slave io 线程，执行命令 `STOP SLAVE IO_THREAD`
- 提升 slave 为 master

```
Stop slave;
Reset master;
Reset slave all;
set global read_only=off; -- 设置为可更新状态
```

- 将原来 master 变为 slave (参考搭建流程中的 slave 方法)

可靠性优先策略：

- 判断备库 B 现在的 `seconds_behind_master`，如果小于某个值（比如 5 秒）继续下一步，否则持续重试这一步
- 把主库 A 改成只读状态，即把 `readonly` 设置为 `true`
- 判断备库 B 的 `seconds_behind_master` 的值，直到这个值变成 0 为止（该步骤比较耗时，所以步骤 1 中要尽量等待该值变小）
- 把备库 B 改成可读写状态，也就是把 `readonly` 设置为 `false`
- 把业务请求切到备库 B

可用性优先策略：先做最后两步，会造成主备数据不一致的问题

参考文章：<https://time.geekbang.org/column/article/76795>

健康检测

主库发生故障后从库会上位，**其他从库指向新的主库**，所以需要一个健康检测的机制来判断主库是否宕机

- select 1 判断，但是高并发下检测不出线程的锁等待的阻塞问题
- 查表判断，在系统库（mysql 库）里创建一个表，比如命名为 health_check，里面只放一行数据，然后定期执行。但是当 binlog 所在磁盘的空间占用率达到 100%，所有的更新和事务提交语句都被阻塞，查询语句可以继续运行
- 更新判断，在健康检测表中放一个 timestamp 字段，用来表示最后一次执行检测的时间

```
UPDATE mysql.health_check SET t_modified=now();
```

节点可用性的检测都应该包含主库和备库，为了让主备之间的更新不产生冲突，可以在 mysql.health_check 表上存入多行数据，并用主备的 server_id 做主键，保证主、备库各自的检测命令不会发生冲突

基于位点

主库上位后，从库 B 执行 CHANGE MASTER TO 命令，指定 MASTER_LOG_FILE、MASTER_LOG_POS 表示从新主库 A 的哪个文件的哪个位点开始同步，这个位置就是**同步位点**，对应主库的文件名和日志偏移量

寻找位点需要找一个稍微往前的，然后再通过判断跳过那些在从库 B 上已经执行过的事务，获取位点方法：

- 等待新主库 A 把中转日志（relay log）全部同步完成
- 在 A 上执行 show master status 命令，得到当前 A 上最新的 File 和 Position
- 取原主库故障的时刻 T，用 mysqlbinlog 工具解析新主库 A 的 File，得到 T 时刻的位点

通常情况下该值并不准确，在切换的过程中会发生错误，所以要先主动跳过这些错误：

- 切换过程中，可能会重复执行一个事务，所以需要主动跳过所有重复的事务

```
SET GLOBAL sql_slave_skip_counter=1;
START SLAVE;
```

- 设置 slave_skip_errors 参数，直接设置跳过指定的错误，保证主从切换的正常进行
 - 1062 错误是插入数据时唯一键冲突
 - 1032 错误是删除数据时找不到行

该方法针对的是主备切换时，由于找不到精确的同步位点，只能采用这种方法来创建从库和新主库的主备关系。等到主备间的同步关系建立完成并稳定执行一段时间后，还需要把这个参数设置为空，以免真的出现了主从数据不一致也跳过了

基于GTID

GTID

GTID 的全称是 Global Transaction Identifier，全局事务 ID，是一个事务在提交时生成的，是这个事务的唯一标识，组成：

```
GTID=source_id:transaction_id
```

- source_id：是一个实例第一次启动时自动生成的，是一个全局唯一的值
- transaction_id：初始值是 1，每次提交事务的时候分配给这个事务，并加 1，是连续的（区分事务 ID，事务 ID 是在执行时生成）

启动 MySQL 实例时，加上参数 `gtid_mode=on` 和 `enforce_gtid_consistency=on` 就可以启动 GTID 模式，每个事务都会和一个 GTID 一一对应，每个 MySQL 实例都维护了一个 GTID 集合，用来存储当前实例执行过的所有事务

GTID 有两种生成方式，使用哪种方式取决于 session 变量 `gtid_next`：

- `gtid_next=automatic`：使用默认值，把 `source_id:transaction_id`（递增）分配给这个事务，然后加入本实例的 GTID 集合

```
@@SESSION.GTID_NEXT = 'source_id:transaction_id';
```

- `gtid_next=GTID`：指定的 GTID 的值，如果该值已经存在于实例的 GTID 集合中，接下来执行的事务会直接被系统忽略；反之就将该值分配给接下来要执行的事务，系统不需要给这个事务生成新的 GTID，也不用加 1

注意：一个 GTID 只能给一个事务使用，所以执行下一个事务，要把 `gtid_next` 设置成另外一个 GTID 或者 `automatic`

业务场景：

- 主库 X 和从库 Y 执行一条相同的指令后进行事务同步

```
INSERT INTO t VALUES(1,1);
```

- 当 Y 同步 X 时，会出现主键冲突，导致实例 X 的同步线程停止，解决方法：

```
SET gtid_next='(这里是主库 X 的 GTID 值)';
BEGIN;
COMMIT;
SET gtid_next=automatic;
START SLAVE;
```

前三条语句通过提交一个空事务，把 X 的 GTID 加到实例 Y 的 GTID 集合中，实例 Y 就会直接跳过这个事务

切换

在 GTID 模式下，CHANGE MASTER TO 不需要指定日志名和日志偏移量，指定 `master_auto_position=1` 代表使用 GTID 模式

新主库实例 A 的 GTID 集合记为 set_a，从库实例 B 的 GTID 集合记为 set_b，主备切换逻辑：

- 实例 B 指定主库 A，基于主备协议建立连接，实例 B 并把 set_b 发给主库 A
- 实例 A 算出 set_a 与 set_b 的差集，就是所有存在于 set_a 但不存在于 set_b 的 GTID 的集合，判断 A 本地是否包含了这个差集需要的所有 binlog 事务
 - 如果不包含，表示 A 已经把实例 B 需要的 binlog 给删掉了，直接返回错误
 - 如果确认全部包含，A 从自己的 binlog 文件里面，找出第一个不在 set_b 的事务，发给 B
- 实例 A 之后就从这个事务开始，往后读文件，按顺序取 binlog 发给 B 去执行

参考文章：<https://time.geekbang.org/column/article/77427>

日志

日志分类

在任何一种数据库中，都会有各种各样的日志，记录着数据库工作的过程，可以帮助数据库管理员追踪数据库曾经发生过的各种事件

MySQL 日志主要包括六种：

1. 重做日志 (redo log)
 2. 回滚日志 (undo log)
 3. 归档日志 (binlog) (二进制日志)
 4. 错误日志 (errorlog)
 5. 慢查询日志 (slow query log)
 6. 一般查询日志 (general log)
 7. 中继日志 (relay log)
-

错误日志

错误日志是 MySQL 中最重要的日志之一，记录了当 mysqld 启动和停止时，以及服务器在运行过程中发生任何严重错误时的相关信息。当数据库出现任何故障导致无法正常使用时，可以首先查看此日志

该日志是默认开启的，默认位置是：`/var/log/mysql/error.log`

查看指令：

```
SHOW VARIABLES LIKE 'log_error%';
```

查看日志内容：

```
tail -f /var/log/mysql/error.log
```

归档日志

基本介绍

归档日志 (BINLOG) 也叫二进制日志，是因为采用二进制进行存储，记录了所有的 DDL (数据定义语言) 语句和 DML (数据操作语言) 语句，但**不包括数据查询语句，在事务提交前的最后阶段写入**

作用：**灾难时的数据恢复和 MySQL 的主从复制**

归档日志默认情况下是没有开启的，需要在 MySQL 配置文件中开启，并配置 MySQL 日志的格式：

```
cd /etc/mysql  
vim my.cnf  
  
# 配置开启binlog日志， 日志的文件前缀为 mysqlbin ----> 生成的文件名如：mysqlbin.000001  
log_bin=mysqlbin  
# 配置二进制日志的格式  
binlog_format=STATEMENT
```

日志存放位置：配置时给定了文件名但是没有指定路径，日志默认写入MySQL 的数据目录

日志格式：

- STATEMENT：该日志格式在日志文件中记录的都是 **SQL 语句**，每一条对数据进行修改的 SQL 都会记录在日志文件中，通过 mysqlbinlog 工具，可以查看到每条语句的文本。主从复制时，从库会将日志解析为原语句，并在从库重新执行一遍

缺点：可能会导致主备不一致，因为记录的 SQL 在不同的环境中可能选择的索引不同，导致结果不同

- ROW：该日志格式在日志文件中记录的是每一行的**数据变更**，而不是记录 SQL 语句。比如执行 SQL 语句 `update tb_book set status='1'`，如果是 STATEMENT，在日志中会记录一行 SQL 语句；如果是 ROW，由于是对全表进行更新，就是每一行记录都会发生变更，ROW 格式的日志中会记录每一行的数据变更

缺点：记录的数据比较多，占用很多的存储空间

- MIXED：这是 MySQL 默认的日志格式，混合了STATEMENT 和 ROW 两种格式，MIXED 格式能尽量利用两种模式的优点，而避开它们的缺点

日志刷盘

事务执行过程中，先将日志写（write）到 binlog cache，事务提交时再把 binlog cache 写（fsync）到 binlog 文件中，一个事务的 binlog 是不能被拆开的，所以不论这个事务多大也要确保一次性写入
事务提交时执行器把 binlog cache 里的完整事务写入到 binlog 中，并清空 binlog cache

write 和 fsync 的时机由参数 sync_binlog 控制的：

- sync_binlog=0：表示每次提交事务都只 write，不 fsync
- sync_binlog=1：表示每次提交事务都会执行 fsync
- sync_binlog=N(N>1)：表示每次提交事务都 write，但累积 N 个事务后才 fsync，但是如果主机发生异常重启，会丢失最近 N 个事务的 binlog 日志

日志读取

日志文件存储位置：/var/lib/mysql

由于日志以二进制方式存储，不能直接读取，需要用 mysqlbinlog 工具来查看，语法如下：

```
mysqlbinlog log-file;
```

查看 STATEMENT 格式日志：

- 执行插入语句：

```
INSERT INTO tb_book VALUES(NULL, 'Lucene', '2088-05-01', '0');
```

- `cd /var/lib/mysql`：

```
-rw-r----- 1 mysql mysql      177 5月 23 21:08 mysqlbin.000001
-rw-r----- 1 mysql mysql       18 5月 23 21:04 mysqlbin.index
```

mysqlbin.index：该文件是日志索引文件，记录日志的文件名；

mysqlbing.000001：日志文件

- 查看日志内容：

```
mysqlbinlog mysqlbing.000001;
```

```

# at 120
#190401  8:48:11 server id 1  end_log_pos 199 CRC32 0xlaafdf97a  Query      thread_id=1      exec_time=0      error_code=0
SET TIMESTAMP=1554079691/*!*/;
SET @@session.pseudo_thread_id=1/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1075838976/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!*/C utf8 *//*!*/;
SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=8/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
BEGIN
/*!*/;
# at 199
# at 231
#190401  8:48:11 server id 1  end_log_pos 231 CRC32 0x7efb8148  Intvar
SET INSERT_ID=5/*!*/;
#190401  8:48:11 server id 1  end_log_pos 363 CRC32 0xafadfea2  Query      thread_id=1      exec_time=0      error_code=0
use `db01`/*!*/;
SET TIMESTAMP=1554079691/*!*/;
insert into tb_book values(null,'Lucene','2088-05-01','0')
/*!*/;
# at 363
#190401  8:48:11 server id 1  end_log_pos 443 CRC32 0x43719d16  Query      thread_id=1      exec_time=0      error_code=0
SET TIMESTAMP=1554079691/*!*/;
COMMIT
/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;

```

日志结尾有 COMMIT

查看 ROW 格式日志：

- 修改配置：

```

# 配置二进制日志的格式
binlog_format=ROW

```

- 插入数据：

```

INSERT INTO tb_book VALUES(NULL, 'SpringCloud实战', '2088-05-05', '0');

```

- 查看日志内容：日志格式 ROW，直接查看数据是乱码，可以在 mysqlbinlog 后面加上参数 -vv

```

mysqlbinlog -vv mysqlbin.000002

```

```

BINLOG '
+ZyhXMBAAAAAQAAAAPkAAAAAAEYAAAAAAEABGRiMDEAB3RiX2Jvb2sABAMPCv4ElgD+Aw4jrzA5
+ZyhXb4BAAAAPwAADgBAAAAAEYAAAAAAEAgAE//AGAAAAEVNwcmLuZ0Nsb3Vksa6e5oiYpVAQ
ATCvbaOV
'/*!*/;
### INSERT INTO `db01`.`tb_book`
### SET
###   @1=6 /* INT meta=0 nullable=0 is_null=0 */
###   @2='SpringCloud实战' /* VARSTRING(150) meta=150 nullable=1 is_null=0 */
###   @3='2088:05:05' /* DATE meta=0 nullable=1 is_null=0 */
###   @4='0' /* STRING(3) meta=65027 nullable=1 is_null=0 */
# at 312
#190401 13:09:13 server id 1  end_log_pos 385 CRC32 0xbd9d1911  Query      thread_id=1      exec_time=0      error_code=0
SET TIMESTAMP=1554095353/*!*/;
COMMIT
/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;

```

日志删除

对于比较繁忙的系统，生成日志量大，这些日志如果长时间不清除，将会占用大量的磁盘空间，需要删除日志

- Reset Master 指令删除全部 binlog 日志，删除之后，日志编号将从 xxxx.000001 重新开始

```
Reset Master      -- MySQL指令
```

- 执行指令 `PURGE MASTER LOGS TO 'mysqlbin.***'`，该命令将删除 *** 编号之前的所有日志
- 执行指令 `PURGE MASTER LOGS BEFORE 'yyyy-mm-dd hh:mm:ss'`，该命令将删除日志为 yyyy-mm-dd hh:mm:ss 之前产生的日志
- 设置参数 `--expire_logs_days=#`，此参数的含义是设置日志的过期天数，过了指定的天数后日志将会被自动删除，这样做有利于减少管理日志的工作量，配置 my.cnf 文件：

```
log_bin=mysqlbin
binlog_format=ROW
--expire_logs_days=3
```

数据恢复

误删库或者表时，需要根据 binlog 进行数据恢复

一般情况下数据库有定时的全量备份，假如每天 0 点定时备份，12 点误删了库，恢复流程：

- 取最近一次全量备份，用备份恢复出一个临时库
- 从日志文件中取出凌晨 0 点之后的日志
- 把除了误删除数据的语句外日志，全部应用到临时库

跳过误删除语句日志的方法：

- 如果原实例没有使用 GTID 模式，只能在应用到包含 12 点的 binlog 文件的时候，先用 `-stop-position` 参数执行到误操作之前的日志，然后再用 `-start-position` 从误操作之后的日志继续执行
- 如果实例使用了 GTID 模式，假设误操作命令的 GTID 是 gtid1，那么只需要提交一个空事务先将这个 GTID 加到临时实例的 GTID 集合，之后按顺序执行 binlog 的时就会自动跳过误操作的语句

查询日志

查询日志中记录了客户端的所有操作语句，而二进制日志不包含查询数据的 SQL 语句

默认情况下，查询日志是未开启的。如果需要开启查询日志，配置 my.cnf：

```
# 该选项用来开启查询日志，可选值0或者1，0代表关闭，1代表开启  
general_log=1  
# 设置日志的文件名，如果没有指定，默认的文件名为host_name.log，存放在/var/lib/mysql  
general_log_file=mysql_query.log
```

配置完毕之后，在数据库执行以下操作：

```
SELECT * FROM tb_book;  
SELECT * FROM tb_book WHERE id = 1;  
UPDATE tb_book SET name = 'Lucene入门指南' WHERE id = 5;  
SELECT * FROM tb_book WHERE id < 8
```

执行完毕之后，再次来查询日志文件：

```
[root@xaxh-server mysql]# cat mysql_query.log  
/usr/sbin/mysqld, Version: 5.6.25-log (MySQL Community Server (GPL)). started with:  
Tcp port: 0 Unix socket: (null)  
Time           Id Command     Argument  
190401 22:09:19    1 Connect    root@localhost on  
                   1 Init DB    db01  
                   1 Query      show databases  
                   1 Query      show tables  
                   1 Field List tb_book  
                   1 Field List tb_item  
                   1 Field List tb_item_cat  
                   1 Field List tb_seller  
190401 22:09:29    1 Query      select * from tb_book  
190401 22:09:37    1 Query      select * from tb_book where id = 1  
190401 22:10:54    1 Query      update tb_book set name = 'Lucene入门指南' where id = 5  
190401 22:12:26    1 Query      select * from tb_book where id < 8
```

慢日志

慢查询日志记录所有执行时间超过 long_query_time 并且扫描记录数不小于 min_examined_row_limit 的所有的 SQL 语句的日志 long_query_time 默认为 10 秒，最小为 0，精度到微秒

慢查询日志默认是关闭的，可以通过两个参数来控制慢查询日志，配置文件 /etc/mysql/my.cnf：

```
# 该参数用来控制慢查询日志是否开启，可选值0或者1，0代表关闭，1代表开启  
slow_query_log=1  
  
# 该参数用来指定慢查询日志的文件名，存放在 /var/lib/mysql  
slow_query_log_file=slow_query.log  
  
# 该选项用来配置查询的时间限制，超过这个时间将认为是慢查询，将需要进行日志记录，默认10s  
long_query_time=10
```

日志读取：

- 直接通过 cat 指令查询该日志文件：

```
cat slow_query.log
```

```
[root@xaxh-server mysql]# cat slow_query.log  
/usr/sbin/mysqld, Version: 5.6.25-log (MySQL Community Server (GPL)). started with:  
Tcp port: 0 Unix socket: (null)  
Time           Id Command     Argument  
# Time: 190401 22:49:26  
# User@Host: root[root] @ localhost []  Id: 1  
# Query_time: 26.769298  Lock_time: 0.000154 Rows_sent: 0  Rows_examined: 9880000  
use db01;
```

- 如果慢查询日志内容很多，直接查看文件比较繁琐，可以借助 mysql 自带的 mysqldumpslow 工具对慢查询日志进行分类汇总：

```
mysqldumpslow slow_query.log
```

```
[root@xaxh-server mysql]# mysqldumpslow slow_query.log
Reading mysql slow query log from slow_query.log
Count: 1  Time=26.77s (26s)  Lock=0.00s (0s)  Rows=0.0 (0), root[root]@localhost
      select * from tb_item where title like 'S'
```

范式

第一范式

建立科学的，**规范的数据表**就需要满足一些规则来优化数据的设计和存储，这些规则就称为范式

1NF：数据库表的每一列都是不可分割的原子数据项，不能是集合、数组等非原子数据项。即表中的某个列有多个值时，必须拆分为不同的列。简而言之，**第一范式每一列不可再拆分，称为原子性**

基本表：

| 学号 | 姓名 | 系 | | 课程名称 | 分数 |
|-------|-----|-----|-----|-------|----|
| | | 系名 | 系主任 | | |
| 10010 | 张无忌 | 经济系 | 张三丰 | 高等数学 | 95 |
| 10010 | 张无忌 | 经济系 | 张三丰 | 计算机基础 | 65 |
| 10011 | 令狐冲 | 法律系 | 任我行 | 法理学 | 77 |
| 10011 | 令狐冲 | 法律系 | 任我行 | 法律社会学 | 65 |
| 10012 | 杨过 | 法律系 | 任我行 | 法律社会学 | 95 |
| 10012 | 杨过 | 法律系 | 任我行 | 法理学 | 97 |

第一范式表：

| 学号 | 姓名 | 系名 | 系主任 | 课程名称 | 分数 | | | | |
|---|-----|-----|-----|-------|----|--|--|--|--|
| 10010 | 张无忌 | 经济系 | 张三丰 | 高等数学 | 95 | | | | |
| 10010 | 张无忌 | 经济系 | 张三丰 | 计算机基础 | 65 | | | | |
| 10011 | 令狐冲 | 法律系 | 任我行 | 法理学 | 77 | | | | |
| 10011 | 令狐冲 | 法律系 | 任我行 | 法律社会学 | 65 | | | | |
| 10012 | 杨过 | 法律系 | 任我行 | 法律社会学 | 95 | | | | |
| 10012 | 杨过 | 法律系 | 任我行 | 法理学 | 97 | | | | |
| 计算机系 | | | 殷天正 | | | | | | |
| 存在的问题： | | | | | | | | | |
| 1. 存在非常严重的数据冗余(重复): 姓名、系名、系主任 2. 数据添加存在问题: 添加新开设的系和系主任时, 数据不合法 3. 数据删除存在问题: 张无忌同学毕业了, 删除数据, 会将系的数据一起删除。 | | | | | | | | | |

第二范式

2NF: 在满足第一范式的基础上, 非主属性完全依赖于主码 (主关键字、主键), 消除非主属性对主码的部分函数依赖。简而言之, 表中的每一个字段 (所有列) 都完全依赖于主键, 记录的唯一性

作用: 遵守第二范式减少数据冗余, 通过主键区分相同数据。

1. 函数依赖: $A \rightarrow B$, 如果通过 A 属性(属性组)的值, 可以确定唯一 B 属性的值, 则称 B 依赖于 A
 - 学号 \rightarrow 姓名; (学号, 课程名称) \rightarrow 分数
2. 完全函数依赖: $A \rightarrow B$, 如果A是一个属性组, 则 B 属性值的确定需要依赖于 A 属性组的所有属性值
 - (学号, 课程名称) \rightarrow 分数
3. 部分函数依赖: $A \rightarrow B$, 如果 A 是一个属性组, 则 B 属性值的确定只需要依赖于 A 属性组的某些属性值
 - (学号, 课程名称) \rightarrow 姓名
4. 传递函数依赖: $A \rightarrow B$, $B \rightarrow C$, 如果通过A属性(属性组)的值, 可以确定唯一 B 属性的值, 在通过 B 属性(属性组)的值, 可以确定唯一 C 属性的值, 则称 C 传递函数依赖于 A
 - 学号 \rightarrow 系名, 系名 \rightarrow 系主任
5. 码: 如果在一张表中, 一个属性或属性组, 被其他所有属性所完全依赖, 则称这个属性(属性组)为该表的码
 - 该表中的码: (学号, 课程名称)
 - 主属性: 码属性组中的所有属性
 - 非主属性: 除码属性组以外的属性

| 学号 | 课程名称 | 分数 | 学号 | 姓名 | 系名 | 系主任 |
|-------|-------|----|-------|-----|------|-----|
| 10010 | 高等数学 | 95 | 10010 | 张无忌 | 经济系 | 张三丰 |
| 10010 | 计算机基础 | 65 | 10012 | 杨过 | 法律系 | 任我行 |
| 10011 | 法理学 | 77 | | | 计算机系 | 殷天正 |
| 10011 | 法律社会学 | 65 | | | | |
| 10012 | 法律社会学 | 95 | | | | |
| 10012 | 法理学 | 97 | | | | |

存在的问题：

- 2. 数据添加存在问题：添加新开设的系和系主任时，数据不合法
- 3. 数据删除存在问题：张无忌同学毕业了，删除数据，会将系的数据一起删除。

第三范式

3NF：在满足第二范式的基础上，表中的任何属性不依赖于其它非主属性，消除传递依赖。简而言之，**非主键都直接依赖于主键，而不是通过其它的键来间接依赖于主键。**

作用：可以通过主键 id 区分相同数据，修改数据的时候只需要修改一张表（方便修改），反之需要修改多表。

| 学号 | 课程名称 | 分数 | 学号 | 姓名 | 系名 |
|-------|-------|----|-------|-----|-----|
| 10010 | 高等数学 | 95 | 10010 | 张无忌 | 经济系 |
| 10010 | 计算机基础 | 65 | 10011 | 令狐冲 | 法律系 |
| 10011 | 法理学 | 77 | 10012 | 杨过 | 法律系 |
| 10011 | 法律社会学 | 65 | | | |
| 10012 | 法律社会学 | 95 | | | |
| 10012 | 法理学 | 97 | | | |

| 系名 | 系主任 |
|------|-----|
| 经济系 | 张三丰 |
| 法律系 | 任我行 |
| 计算机系 | 殷天正 |

总结

| 范式 | 特点 |
|-----|--|
| 1NF | 原子性：表中每列不可再拆分。 |
| 2NF | 不产生局部依赖，一张表只描述一件事情。 |
| 3NF | 不产生传递依赖，表中每一列都直接依赖于主键。而不是通过其它列间接依赖于主键。 |

Redis

NoSQL

概述

NoSQL (Not-Only SQL) : 泛指非关系型的数据库，作为关系型数据库的补充

MySQL 支持 ACID 特性，保证可靠性和持久性，读取性能不高，因此需要缓存的来减缓数据库的访问压力

作用：应对基于海量用户和海量数据前提下的数据处理问题

特征：

- 可扩容，可伸缩，SQL 数据关系过于复杂，NoSQL 不存关系，只存数据
- 大数据量下高性能，数据不存取在磁盘 IO，存取在内存
- 灵活的数据模型，设计了一些数据存储格式，能保证效率上的提高
- 高可用，集群

常见的 NoSQL：Redis、memcache、HBase、MongoDB

参考书籍：<https://book.douban.com/subject/25900156/>

参考视频：<https://www.bilibili.com/video/BV1CJ411m7Gc>

Redis

Redis (REmote DIctionary Server) : 用 C 语言开发的一个开源的高性能键值对 (key-value) 数据库

特征：

- 数据间没有必然的关联关系，**不存在关系，只存数据**

- 数据存储在内存，存取速度快，解决了磁盘 IO 速度慢的问题
 - 内部采用单线程机制进行工作
 - 高性能，官方测试数据，50 个并发执行 100000 个请求，读的速度是 11000 次/s，写的速度是 81000 次/s
 - 多数据类型支持
 - 字符串类型：string (String)
 - 列表类型：list (LinkedList)
 - 散列类型：hash (HashMap)
 - 集合类型：set (HashSet)
 - 有序集合类型：zset/sorted_set (TreeSet)
 - 支持持久化，可以进行数据灾难恢复
-

安装启动

安装：

- Redis 5.0 被包含在默认的 Ubuntu 20.04 软件源中

```
sudo apt update  
sudo apt install redis-server
```

- 检查 Redis 状态

```
sudo systemctl status redis-server
```

启动：

- 启动服务器——参数启动

```
redis-server [--port port]  
#redis-server --port 6379
```

- 启动服务器——配置文件启动

```
redis-server config_file_name  
#redis-server /etc/redis/conf/redis-6397.conf
```

- 启动客户端：

```
redis-cli [-h host] [-p port]  
#redis-cli -h 192.168.2.185 -p 6397
```

注意：服务器启动指定端口使用的是--port，客户端启动指定端口使用的是-p

基本配置

系统目录

1. 创建文件结构

创建配置文件存储目录

```
mkdir conf
```

创建服务器文件存储目录（包含日志、数据、临时配置文件等）

```
mkdir data
```

2. 创建配置文件副本放入 conf 目录，Ubuntu 系统配置文件 redis.conf 在目录 /etc/redis 中

```
cat redis.conf | grep -v "#" | grep -v "^$" -> /conf/redis-6379.conf
```

去除配置文件的注释和空格，输出到新的文件，命令方式采用 redis-port.conf

服务器

- 设置服务器以守护进程的方式运行，关闭后服务器控制台中将打印服务器运行信息（同日志内容相同）：

```
daemonize yes|no
```

- 绑定主机地址，绑定本地IP地址，否则SSH无法访问：

```
bind ip
```

- 设置服务器端口：

```
port port
```

- 设置服务器文件保存地址：

```
dir path
```

- 设置数据库的数量：

```
databases 16
```

- 多服务器快捷配置：

导入并加载指定配置文件信息，用于快速创建 redis 公共配置较多的 redis 实例配置文件，便于维护

```
include /path/conf_name.conf
```

客户端

- 服务器允许客户端连接最大数量，默认 0，表示无限制，当客户端连接到达上限后，Redis 会拒绝新的连接：

```
maxclients count
```

- 客户端闲置等待最大时长，达到最大值后关闭对应连接，如需关闭该功能，设置为 0：

```
timeout seconds
```

日志配置

设置日志记录

- 设置服务器以指定日志记录级别

```
loglevel debug|verbose|notice|warning
```

- 日志记录文件名

```
logfile filename
```

注意：日志级别开发期设置为 verbose 即可，生产环境中配置为 notice，简化日志输出量，降低写日志 IO 的频度

配置文件：

```
bind 192.168.2.185
port 6379
#timeout 0
daemonize no
logfile /etc/redis/data/redis-6379.log
dir /etc/redis/data
dbfilename "dump-6379.rdb"
```

基本指令

帮助信息：

- 获取命令帮助文档

```
help [command]  
#help set
```

- 获取组中所有命令信息名称

```
help [@group-name]  
#help @string
```

退出服务

- 退出客户端：

```
quit  
exit
```

- 退出客户端服务器快捷键：

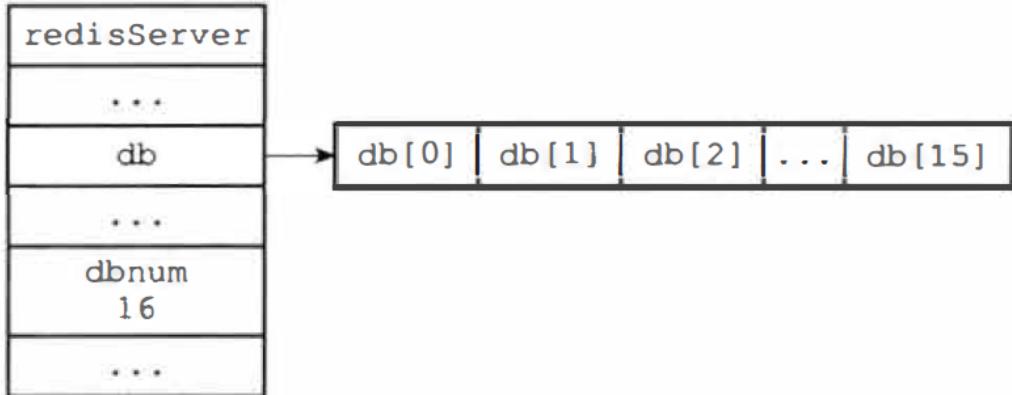
```
ctrl+c
```

数据库

服务器

Redis 服务器将所有数据库保存在**服务器状态 redisServer** 结构的 db 数组中，数组的每一项都是 redisDb 结构，代表一个数据库，每个数据库之间相互独立，**共用** Redis 内存，不区分大小。在初始化服务器时，根据 dbnum 属性决定创建数据库的数量，该属性由服务器配置的 database 选项决定，默认 16

```
struct redisServer {  
    // 保存服务器所有的数据库  
    redisDB *db;  
  
    // 服务器数据库的数量  
    int dbnum;  
};
```



在服务器内部, 客户端状态 redisClient 结构的 db 属性记录了目标数据库, 是一个指向 redisDb 结构的指针

```
struct redisClient {  
    // 记录客户端正在使用的数据库, 指向 redisServer.db 数组中的某一个 db  
    redisDB *db;  
};
```

每个 Redis 客户端都有目标数据库, 执行数据库读写命令时目标数据库就会成为这些命令的操作对象, 默认情况下 Redis 客户端的目标数据库为 0 号数据库, 客户端可以执行 SELECT 命令切换目标数据库, 原理是通过修改 redisClient.db 指针指向服务器中不同数据库

命令操作:

```
select index      #切换数据库, index从0-15取值  
move key db      #数据移动到指定数据库, db是数据库编号  
ping             #测试数据库是否连接正常, 返回PONG  
echo message     #控制台输出信息
```

Redis 没有可以返回客户端目标数据库的命令, 但是 redis-cli 客户端旁边会提示当前所使用的目标数据库

```
redis> SELECT 1  
OK  
redis[1]>
```

键空间

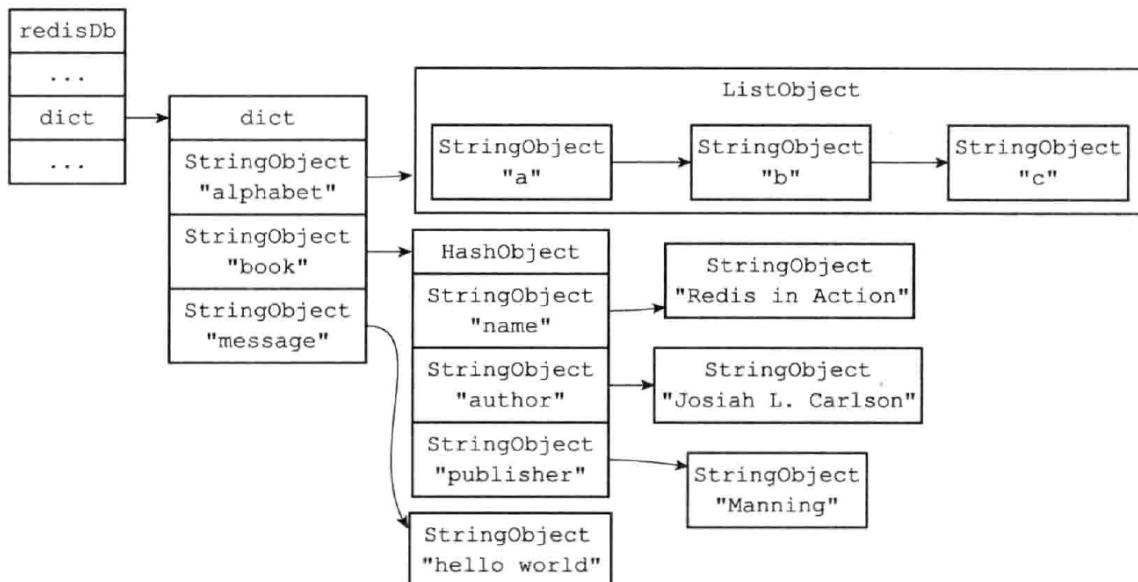
key space

Redis 是一个键值对 (key-value pair) 数据库服务器，每个数据库都由一个 redisDb 结构表示，redisDb.dict 字典中保存了数据库的所有键值对，将这个字典称为键空间 (key space)

```
typedef struct redisDB {
    // 数据库键空间，保存所有键值对
    dict *dict
} redisDB;
```

键空间和用户所见的数据库是直接对应的：

- 键空间的键就是数据库的键，每个键都是一个字符串对象
- 键空间的值就是数据库的值，每个值可以是任意一种 Redis 对象



当使用 Redis 命令对数据库进行读写时，服务器不仅会对键空间执行指定的读写操作，还会进行一些维护操作：

- 在读取一个键后（读操作和写操作都要对键进行读取），服务器会根据键是否存在来更新服务器的键空间命中 hit 次数或键空间不命中 miss 次数，这两个值可以在 `INFO stats` 命令的 `keyspace_hits` 属性和 `keyspace_misses` 属性中查看
- 更新键的 LRU (最后使用) 时间，该值可以用于计算键的闲置时间，使用 `OBJECT idletime key` 查看键 `key` 的闲置时间
- 如果在读取一个键时发现该键已经过期，服务器会先删除过期键，再执行其他操作
- 如果客户端使用 WATCH 命令监视了某个键，那么服务器在对被监视的键进行修改之后，会将这个键标记为脏 (dirty)，从而让事务注意到这个键已经被修改过
- 服务器每次修改一个键之后，都会对 dirty 键计数器的值增1，该计数器会触发服务器的持久化以及复制操作
- 如果服务器开启了数据库通知功能，那么在对键进行修改之后，服务器将按配置发送相应的数据库通知

读写指令

常见键操作指令：

- 增加指令

```
set key value      #添加一个字符串类型的键值对
```

- 删除指令

```
del key          #删除指定key  
unlink key       #非阻塞删除key, 真正的删除会在后续异步操作
```

- 更新指令

```
rename key newkey    #改名  
renamenx key newkey  #改名
```

值得更新需要参看具体得 Redis 对象得操作方式，比如字符串对象执行 `SET key value` 就可以完成修改

- 查询指令

```
exists key        #获取key是否存在  
randomkey         #随机返回一个键  
keys pattern      #查询key
```

`KEYS` 命令需要**遍历存储的键值对**，操作延时高，一般不被建议用于生产环境中

查询模式规则：* 匹配任意数量的任意符号、? 配合一个任意符号、[] 匹配一个指定符号

```
keys *            #查询所有key  
keys aa*          #查询所有以aa开头  
keys *bb          #查询所有以bb结尾  
keys ??cc          #查询所有前面两个字符任意，后面以cc结尾  
keys user:?        #查询所有以user:开头，最后一个字符任意  
keys u[st]er:1      #查询所有以u开头，以er:1结尾，中间包含一个字母，s或t
```

- 其他指令

```
type key          #获取key的类型  
dbsize             #获取当前数据库的数据总量，即key的个数  
flushdb            #清除当前数据库的所有数据(慎用)  
flushall           #清除所有数据(慎用)
```

在执行 `FLUSHDB` 这样的危险命令之前，最好先执行一个 `SELECT` 命令，保证当前所操作的数据库是目标数据库

时效设置

客户端可以以秒或毫秒的精度为数据库中的某个键设置生存时间 (Time To Live, TTL) , 在经过指定时间之后, 服务器就会自动删除生存时间为 0 的键; 也可以以 UNIX 时间戳的方式设置过期时间 (expire time) , 当键的过期时间到达, 服务器会自动删除这个键

```
expire key seconds          #为指定key设置生存时间, 单位为秒  
pexpire key milliseconds   #为指定key设置生存时间, 单位为毫秒  
expireat key timestamp     #为指定key设置过期时间, 单位为时间戳  
pexpireat key mil-timestamp #为指定key设置过期时间, 单位为毫秒时间戳
```

- 实际上 EXPIRE、EXPIRE、EXPIREAT 三个命令底层都是转换为 PEXPIREAT 命令来实现的
- SETEX 命令可以在设置一个字符串键的同时为键设置过期时间, 但是该命令是一个类型限定命令

redisDb 结构的 expires 字典保存了数据库中所有键的过期时间, 字典称为过期字典:

- 键是一个指针, 指向键空间中的某个键对象 (复用键空间的对象, 不会产生内存浪费)
- 值是一个 long long 类型的整数, 保存了键的过期时间, 是一个毫秒精度的 UNIX 时间戳

```
typedef struct redisDB {  
    // 过期字典, 保存所有键的过期时间  
    dict *expires  
} redisDB;
```

客户端执行 PEXPIREAT 命令, 服务器会在数据库的过期字典中关联给定的数据库键和过期时间:

```
def PEXPIREAT(key, expire_time_in_ms):  
    # 如果给定的键不存在于键空间, 那么不能设置过期时间  
    if key not in redisDb.dict:  
        return 0  
  
    # 在过期字典中关联键和过期时间  
    redisDB.expires[key] = expire_time_in_ms  
  
    # 过期时间设置成功  
    return 1
```

时效状态

TTL 和 PTTL 命令通过计算键的过期时间和当前时间之间的差, 返回这个键的剩余生存时间

- 返回正数代表该数据在内存中还能存活的时间
- 返回 -1 代表永久性, 返回 -2 代表键不存在

```
ttl key          #获取key的剩余时间, 每次获取会自动变化(减小), 类似于倒计时  
pttl key        #获取key的剩余时间, 单位是毫秒, 每次获取会自动变化(减小)
```

PERSIST 是 PEXPIREAT 命令的反操作, 在过期字典中查找给定的键, 并解除键和值 (过期时间) 在过期字典中的关联

```
persist key      #切换key从时效性转换为永久性
```

Redis 通过过期字典可以检查一个给定键是否过期：

- 检查给定键是否存在与过期字典：如果存在，那么取得键的过期时间
- 检查当前 UNIX 时间戳是否大于键的过期时间：如果是那么键已经过期，否则键未过期

补充：AOF、RDB 和复制功能对过期键的处理

- RDB：
 - 生成 RDB 文件，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的 RDB 文件中
 - 载入 RDB 文件，如果服务器以主服务器模式运行，那么在载入时会对键进行检查，过期键会被忽略；如果服务器以从服务器模式运行，会载入所有键，包括过期键，但是主从服务器进行数据同步时就会删除这些键
- AOF：
 - 写入 AOF 文件，如果数据库中的某个键已经过期，但还没有被删除，那么 AOF 文件不会因为这个过期键而产生任何影响；当该过期键被删除，程序会向 AOF 文件追加一条 DEL 命令，显式的删除该键
 - AOF 重写，会对数据库中的键进行检查，忽略已经过期的键
- 复制：当服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制
 - 主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个 DEL 命令，告知从服务器删除这个过期键
 - 从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将过期键删除，会当作未过期键处理，只有在接到主服务器发来的 DEL 命令之后，才会删除过期键（数据不一致）

过期删除

删除策略

删除策略就是**针对已过期数据的处理策略**，已过期的数据不一定被立即删除，在不同的场景下使用不同的删除方式会有不同效果，在内存占用与 CPU 占用之间寻找一种平衡，顾此失彼都会造成整体 Redis 性能的下降，甚至引发服务器宕机或内存泄露

针对过期数据有三种删除策略：

- 定时删除
- 惰性删除（被动删除）
- 定期删除

Redis 采用惰性删除和定期删除策略的结合使用

定时删除

在设置键的过期时间的同时，创建一个定时器（timer），让定时器在键的过期时间到达时，立即执行对键的删除操作

- 优点：节约内存，到时就删除，快速释放掉不必要的内存占用
- 缺点：对 CPU 不友好，无论 CPU 此时负载多高均占用 CPU，会影响 Redis 服务器响应时间和指令吞吐量
- 总结：用处理器性能换取存储空间（拿时间换空间）

创建一个定时器需要用到 Redis 服务器中的时间事件，而时间事件的实现方式是无序链表，查找一个事件的时间复杂度为 $O(N)$ ，并不能高效地处理大量时间事件，所以采用这种方式并不现实

惰性删除

数据到达过期时间不做处理，等下次访问到该数据时执行 `expireIfNeeded()` 判断：

- 如果输入键已经过期，那么 `expireIfNeeded` 函数将输入键从数据库中删除，接着访问就会返回空
- 如果输入键未过期，那么 `expireIfNeeded` 函数不做动作

所有的 Redis 读写命令在执行前都会调用 `expireIfNeeded` 函数进行检查，该函数就像一个过滤器，在命令真正执行之前过滤掉过期键

惰性删除的特点：

- 优点：节约 CPU 性能，删除的目标仅限于当前处理的键，不会在删除其他无关的过期键上花费任何 CPU 时间
 - 缺点：内存压力很大，出现长期占用内存的数据，如果过期键永远不被访问，这种情况相当于内存泄漏
 - 总结：用存储空间换取处理器性能（拿空间换时间）
-

定期删除

定期删除策略是每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对 CPU 时间的影响

- 如果删除操作执行得太频繁，或者执行时间太长，就会退化成定时删除策略，将 CPU 时间过多地消耗在删除过期键上
- 如果删除操作执行得太少，或者执行时间太短，定期删除策略又会和惰性删除策略一样，出现浪费内存的情况

定期删除是周期性轮询 Redis 库中的时效性数据，从过期字典中随机抽取一部分键检查，利用过期数据占比的方式控制删除频度

- Redis 启动服务器初始化时，读取配置 `server.hz` 的值，默认为 10，执行指令 `info server` 可以查看，每秒钟执行 `server.hz` 次 `serverCron() → activeExpireCycle()`
- `activeExpireCycle()` 对某个数据库中的每个 `expires` 进行检测，工作模式：

- 轮询每个数据库，从数据库中取出一定数量的随机键进行检查，并删除其中的过期键，如果过期 key 的比例超过了 25%，则继续重复此过程，直到过期 key 的比例下降到 25% 以下，或者这次任务的执行耗时超过了 25 毫秒
- 全局变量 current_db 用于记录 activeExpireCycle() 的检查进度（哪一个数据库），下一次调用时接着该进度处理
- 随着函数的不断执行，服务器中的所有数据库都会被检查一遍，这时将 current_db 重置为 0，然后再次开始新一轮的检查

定期删除特点：

- CPU 性能占用设置有峰值，检测频度可自定义设置
 - 内存压力不是很大，长期占用内存的冷数据会被持续清理
 - 周期性抽查存储空间（随机抽查，重点抽查）
-

数据淘汰

逐出算法

数据淘汰策略：当新数据进入 Redis 时，在执行每一个命令前，会调用 **freeMemoryIfNeeded()** 检测内存是否充足。如果内存不满足新加入数据的最低存储要求，Redis 要临时删除一些数据为当前指令清理存储空间，清理数据的策略称为**逐出算法**

逐出数据的过程不是 100% 能够清理出足够的可使用的内存空间，如果不成功则反复执行，当对所有数据尝试完毕，如不能达到内存清理的要求，**出现 Redis 内存打满异常**：

```
(error) OOM command not allowed when used memory > 'maxmemory'
```

策略配置

Redis 如果不设置最大内存大小或者设置最大内存大小为 0，在 64 位操作系统下不限制内存大小，在 32 位操作系统默认为 3GB 内存，一般推荐设置 Redis 内存为最大物理内存的四分之三

内存配置方式：

- 通过修改文件配置（永久生效）：修改配置文件 maxmemory 字段，单位为字节
- 通过命令修改（重启失效）：
 - `config set maxmemory 104857600`：设置 Redis 最大占用内存为 100MB
 - `config get maxmemory`：获取 Redis 最大占用内存
 - `info`：可以查看 Redis 内存使用情况，`used_memory_human` 字段表示实际已经占用的内存，`maxmemory` 表示最大占用内存

影响数据淘汰的相关配置如下，配置 conf 文件：

- 每次选取待删除数据的个数，采用随机获取数据的方式作为待检测删除数据，防止全库扫描，导致严重的性能消耗，降低读写性能

```
maxmemory-samples count
```

- 达到最大内存后的，对被挑选出来的数据进行删除的策略

```
maxmemory-policy policy
```

数据删除的策略 policy：3类8种

第一类：检测易失数据（可能会过期的数据集 server.db[i].expires）：

```
volatile-lru      # 对设置了过期时间的 key 选择最近最久未使用使用的数据淘汰  
volatile-lfu      # 对设置了过期时间的 key 选择最近使用次数最少的数据淘汰  
volatile-ttl      # 对设置了过期时间的 key 选择将要过期的数据淘汰  
volatile-random  # 对设置了过期时间的 key 选择任意数据淘汰
```

第二类：检测全库数据（所有数据集 server.db[i].dict）：

```
allkeys-lru      # 对所有 key 选择最近最少使用的数据淘汰  
allkeys-lfu      # 对所有 key 选择最近使用次数最少的数据淘汰  
allkeys-random  # 对所有 key 选择任意数据淘汰，相当于随机
```

第三类：放弃数据驱逐

```
no-eviction      #禁止驱逐数据(redis4.0中默认策略)，会引发OOM(out of Memory)
```

数据淘汰策略配置依据：使用 INFO 命令输出监控信息，查询缓存 hit 和 miss 的次数，根据需求调优 Redis 配置

排序机制

基本介绍

Redis 的 SORT 命令可以对列表键、集合键或者有序集合键的值进行排序，并不更改集合中的数据位置，只是查询

```
SORT key [ASC/DESC]          #对key中数据排序，默认对数字排序，并不更改集合中的数据位置，只是查询  
SORT key ALPHA                #对key中字母排序，按照字典序
```

SORT

`SORT <key>` 命令可以对一个包含数字值的键 key 进行排序

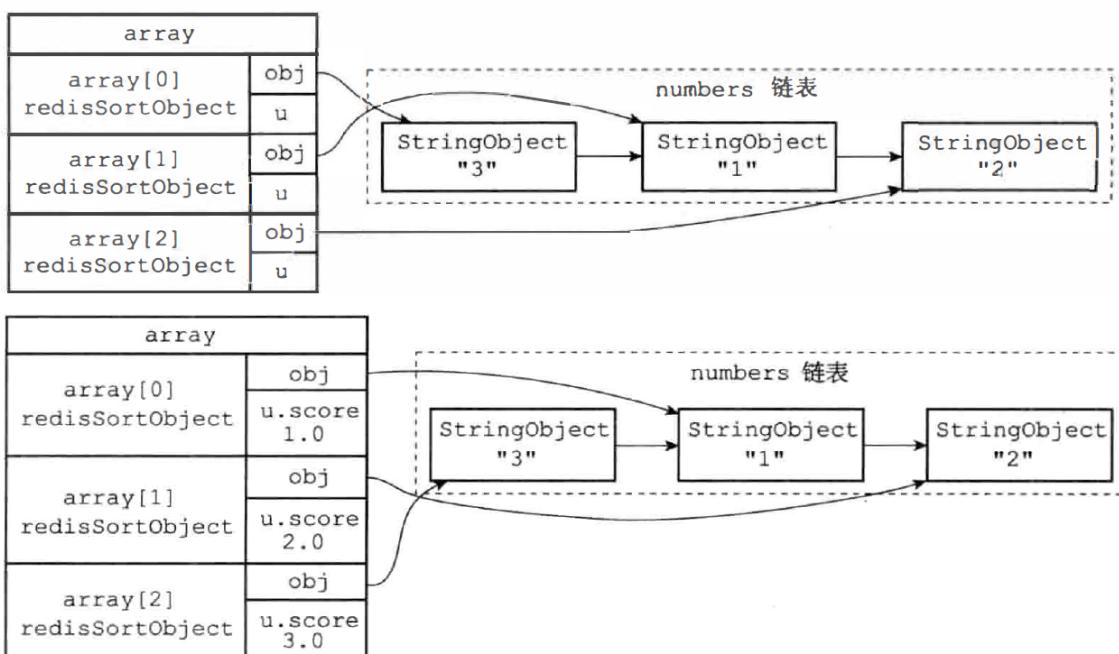
假设 `RPUSH numbers 3 1 2`, 执行 `SORT numbers` 的详细步骤:

- 创建一个和 key 列表长度相同的数组, 数组每项都是 redisSortObject 结构

```
typedef struct redisSortObject {
    // 被排序键的值
    robj *obj;

    // 权重
    union {
        // 排序数字值时使用
        double score;
        // 排序带有 BY 选项的字符串
        robj *cmpobj;
    } u;
}
```

- 遍历数组, 将各个数组项的 obj 指针分别指向 numbers 列表的各个项
- 遍历数组, 将 obj 指针所指向的列表项转换成一个 double 类型的浮点数, 并将浮点数保存在对应数组项的 u.score 属性里
- 根据数组项 u.score 属性的值, 对数组进行数字值排序, 排序后的数组项按 u.score 属性的值**从小到大排列**
- 遍历数组, 将各个数组项的 obj 指针所指向的值作为排序结果返回给客户端, 程序首先访问数组的索引 0, 依次向后访问



对于 `SORT key [ASC/DESC]` 函数:

- 在执行升序排序时, 排序算法使用的对比函数产生升序对比结果
- 在执行降序排序时, 排序算法使用的对比函数产生降序对比结果

BY

SORT 命令默认使用被排序键中包含的元素作为排序的权重，元素本身决定了元素在排序之后所处的位置，通过使用 BY 选项，SORT 命令可以指定某些字符串键，或者某个哈希键所包含的某些域（field）来作为元素的权重，对一个键进行排序

```
SORT <key> BY <pattern>      # 数值  
SORT <key> BY <pattern> ALPHA # 字符
```

```
redis> SADD fruits "apple" "banana" "cherry"  
(integer) 3  
redis> SORT fruits ALPHA  
1) "apple"  
2) "banana"  
3) "cherry"
```

```
redis> MSET apple-price 8 banana-price 5.5 cherry-price 7  
OK  
# 使用水果的价钱进行排序  
redis> SORT fruits BY *-price  
1) "banana"  
2) "cherry"  
3) "apple"
```

实现原理：排序时的 u.score 属性就会被设置为对应的权重

LIMIT

SORT 命令默认会将排序后的所有元素都返回给客户端，通过 LIMIT 选项可以让 SORT 命令只返回其中一部分已排序的元素

```
LIMIT <offset> <count>
```

- offset 参数表示要跳过的已排序元素数量
- count 参数表示跳过给定数量的元素后，要返回的已排序元素数量

```
# 对应 a b c d e f g  
redis> SORT alphabet ALPHA LIMIT 2 3  
1) "c"  
2) "d"  
3) "e"
```

实现原理：在排序后的 redisSortObject 结构数组中，将指针移动到数组的索引 2 上，依次访问 array[2]、array[3]、array[4] 这 3 个数组项，并将数组项的 obj 指针所指向的元素返回给客户端

GET

SORT 命令默认在对键进行排序后，返回被排序键本身所包含的元素，通过使用 GET 选项，可以在对键进行排序后，根据被排序的元素以及 GET 选项所指定的模式，查找并返回某些键的值

```
SORT <key> GET <pattern>
```

```
redis> SADD students "tom" "jack" "sea"
#设置全名
redis> SET tom-name "Tom Li"
OK
redis> SET jack-name "Jack Wang"
OK
redis> SET sea-name "Sea Zhang"
OK
```

```
redis> SORT students ALPHA GET *-name
1) "Jack Wang"
2) "Sea Zhang"
3) "Tom Li"
```

实现原理：对 students 进行排序后，对于 jack 元素和 *-name 模式，查找程序返回键 jack-name，然后获取 jack-name 键对应的值

STORE

SORT 命令默认只向客户端返回排序结果，而不保存排序结果，通过使用 STORE 选项可以将排序结果保存在指定的键里面

```
SORT <key> STORE <sort_key>
```

```
redis> SADD students "tom" "jack" "sea"
(integer) 3
redis> SORT students ALPHA STORE sorted_students
(integer) 3
```

实现原理：排序后，检查 sorted_students 键是否存在，如果存在就删除该键，设置 sorted_students 为空白的列表键，遍历排序数组将元素依次放入

执行顺序

调用 SORT 命令，除了 GET 选项之外，改变其他选项的摆放顺序并不会影响命令执行选项的顺序

```
SORT <key> ALPHA [ASC/DESC] BY <by-pattern> LIMIT <offset> <count> GET <get-pattern> STORE <store_key>
```

执行顺序：

- 排序：命令会使用 ALPHA、ASC 或 DESC、BY 这几个选项，对输入键进行排序，并得到一个排序结果集
 - 限制排序结果集的长度：使用 LIMIT 选项，对排序结果集的长度进行限制
 - 获取外部键：根据排序结果集中的元素以及 GET 选项指定的模式，查找并获取指定键的值，并用这些值来作为新的排序结果集
 - 保存排序结果集：使用 STORE 选项，将排序结果集保存到指定的键上面去
 - 向客户端返回排序结果集：最后一步命令遍历排序结果集，并依次向客户端返回排序结果集中的元素
-

通知机制

数据库通知是可以让客户端通过订阅给定的频道或者模式，来获知数据库中键的变化，以及数据库中命令的执行情况

- 关注某个键执行了什么命令的通知称为键空间通知（key-space notification）
- 关注某个命令被什么键执行的通知称为键事件通知（key-event notification）

图示订阅 0 号数据库 message 键：

```

127.0.0.1:6379> SUBSCRIBE __keyspace@0__:message
Reading messages... (press Ctrl-C to quit)

1) "subscribe"                                // 订阅信息
2) "__keyspace@0__:message"
3) (integer) 1

1) "message"                                    // 执行 SET 命令
2) "__keyspace@0__:message"
3) "set"

1) "message"                                    // 执行 EXPIRE 命令
2) "__keyspace@0__:message"
3) "expire"

1) "message"                                    // 执行 DEL 命令
2) "__keyspace@0__:message"
3) "del"

```

服务器配置的 `notify-keyspace-events` 选项决定了服务器所发送通知的类型

- AKE 代表服务器发送所有类型的键空间通知和键事件通知
- AK 代表服务器发送所有类型的键空间通知
- AE 代表服务器发送所有类型的键事件通知
- K\$ 代表服务器只发送和字符串键有关的键空间通知
- EL 代表服务器只发送和列表键有关的键事件通知
-

发送数据库通知的功能是由 `notifyKeyspaceEvent` 函数实现的：

- 如果给定的通知类型 `type` 不是服务器允许发送的通知类型，那么函数会直接返回
- 如果给定的通知是服务器允许发送的通知
 - 检测服务器是否允许发送键空间通知，允许就会构建并发送事件通知
 - 检测服务器是否允许发送键事件通知，允许就会构建并发送事件通知

体系架构

事件驱动

基本介绍

Redis 服务器是一个事件驱动程序，服务器需要处理两类事件

- 文件事件 (file event): 服务器通过套接字与客户端（或其他 Redis 服务器）进行连接，而文件事件就是服务器对套接字操作的抽象。服务器与客户端的通信会产生相应的文件事件，服务器通过监听并处理这些事件完成一系列网络通信操作
- 时间事件 (time event): Redis 服务器中的一些操作（比如 `serverCron` 函数）需要在指定时间执行，而时间事件就是服务器对这类定时操作的抽象

文件事件

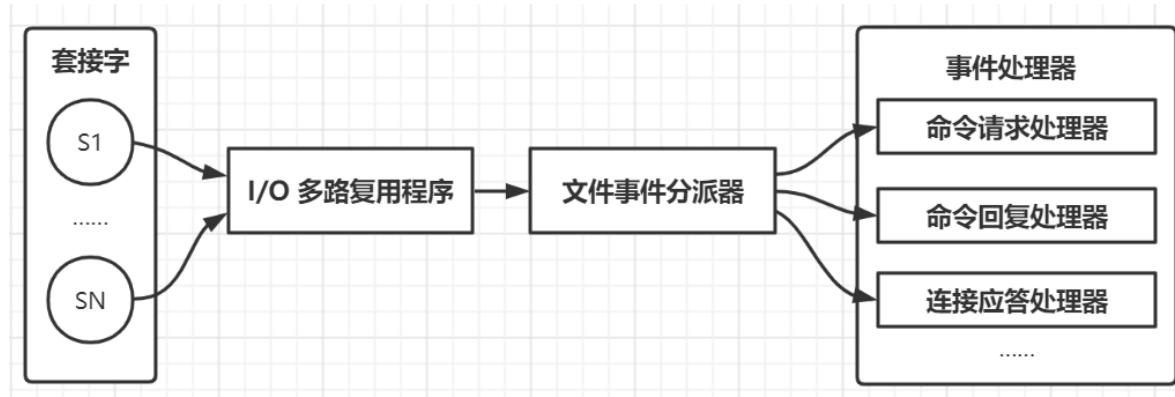
基本组成

Redis 基于 Reactor 模式开发了网络事件处理器，这个处理器被称为文件事件处理器 (file event handler)

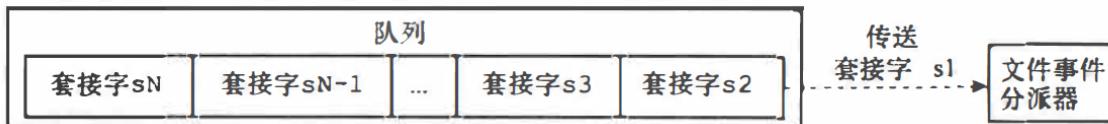
- 使用 I/O 多路复用 (multiplexing) 程序来同时监听多个套接字，并根据套接字执行的任务来为套接字关联不同的事件处理器
- 当被监听的套接字准备好执行连接应答 (accept)、读取 (read)、写入 (write)、关闭 (close) 等操作时，与操作相对应的文件事件就会产生，这时文件事件分派器会调用套接字关联好的事件处理器来处理事件

文件事件处理器以单线程方式运行，但通过使用 I/O 多路复用程序来监听多个套接字，既实现了高性能的网络通信模型，又可以很好地与 Redis 服务器中其他同样以单线程方式运行的模块进行对接，保持了 Redis 内部单线程设计的简单性

文件事件处理器的组成结构：



I/O 多路复用程序将所有产生事件的套接字处理请求放入一个单线程的执行队列中，通过队列有序、同步的向文件事件分派器传送套接字，上一个套接字产生的事件处理完后，才会继续向分派器传送下一个



Redis 单线程也能高效的原因：

- 纯内存操作
- 核心是基于非阻塞的 IO 多路复用机制，单线程可以高效处理多个请求
- 底层使用 C 语言实现，C 语言实现的程序距离操作系统更近，执行速度相对会更快
- 单线程同时也避免了多线程的上下文频繁切换问题，预防了多线程可能产生的竞争问题

多路复用

Redis 的 I/O 多路复用程序的所有功能都是通过包装常见的 select、epoll、evport 和 kqueue 这些函数库来实现的，Redis 在 I/O 多路复用程序的实现源码中用 #include 宏定义了相应的规则，编译时自动选择系统中性能最高的**多路复用函数**来作为底层实现

I/O 多路复用程序监听多个套接字的 AE_READABLE 事件和 AE_WRITABLE 事件，这两类事件和套接字操作之间的对应关系如下：

- 当套接字变得**可读**时（客户端对套接字执行 write 操作或者 close 操作），或者有新的**可应答**（acceptable）套接字出现时（客户端对服务器的监听套接字执行 connect 连接操作），套接字产生 AE_READABLE 事件
- 当套接字变得可写时（客户端对套接字执行 read 操作，对于服务器来说就是可以写了），套接字产生 AE_WRITABLE 事件

I/O 多路复用程序允许服务器同时监听套接字的 AE_READABLE 和 AE_WRITABLE 事件，如果一个套接字同时产生了这两种事件，那么文件事件分派器会优先处理 AE_READABLE 事件，等 AE_READABLE 事件处理完之后才处理 AE_WRITABLE 事件

处理器

Redis 为文件事件编写了多个处理器，这些事件处理器分别用于实现不同的网络通信需求：

- 连接应答处理器，用于对连接服务器的各个客户端进行应答，Redis 服务器初始化时将该处理器与 AE_READABLE 事件关联
- 命令请求处理器，用于接收客户端传来的命令请求，执行套接字的读入操作，与 AE_READABLE 事件关联
- 命令回复处理器，用于向客户端返回命令的执行结果，执行套接字的写入操作，与 AE_WRITABLE 事件关联
- 复制处理器，当主服务器和从服务器进行复制操作时，主从服务器都需要关联该处理器

Redis 客户端与服务器进行连接并发送命令的整个过程：

- Redis 服务器正在运作监听套接字的 AE_READABLE 事件，关联连接应答处理器
 - 当 Redis 客户端向服务器发起连接，监听套接字将产生 AE_READABLE 事件，触发连接应答处理器执行，对客户端的连接请求进行应答，创建客户端套接字以及客户端状态，并将客户端套接字的 **AE_READABLE 事件与命令请求处理器**进行关联
 - 客户端向服务器发送命令请求，客户端套接字产生 AE_READABLE 事件，引发命令请求处理器执行，读取客户端的命令内容传给相关程序去执行
 - 执行命令会产生相应的命令回复，为了将这些命令回复传回客户端，服务器会将客户端套接字的 **AE_WRITABLE 事件与命令回复处理器**进行关联
 - 当客户端尝试读取命令回复时，客户端套接字产生 AE_WRITABLE 事件，触发命令回复处理器执行，在命令回复全部写入套接字后，服务器就会解除客户端套接字的 AE_WRITABLE 事件与命令回复处理器之间的关联
-

时间事件

Redis 的时间事件分为以下两类：

- 定时事件：在指定的时间之后执行一次（Redis 中暂时未使用）
- 周期事件：每隔指定时间就执行一次

一个时间事件主要由以下三个属性组成：

- id：服务器为时间事件创建的全局唯一 ID（标识号），从小到大顺序递增，新事件的 ID 比旧事件的 ID 号要大
- when：毫秒精度的 UNIX 时间戳，记录了时间事件的到达（arrive）时间
- timeProc：时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理事件

时间事件是定时事件还是周期性事件取决于时间事件处理器的返回值：

- 定时事件：事件处理器返回 AE_NOMORE，该事件在到达一次后就会被删除
- 周期事件：事件处理器返回非 AE_NOMORE 的整数值，服务器根据该值对事件的 when 属性更新，让该事件在一段时间后再次交付

服务器将所有时间事件都放在一个无序链表中，新的时间事件插入到链表的表头：



无序链表指的是链表不按 when 属性的大小排序，每当时间事件执行器运行时就必须遍历整个链表，查找所有已到达的时间事件，并调用相应的事件处理器处理

无序链表并不影响时间事件处理器的性能，因为正常模式下的 Redis 服务器只使用 serverCron 一个时间事件，在 benchmark 模式下服务器也只使用两个时间事件，所以无序链表不会影响服务器的性能，几乎可以按照一个指针处理

事件调度

服务器中同时存在文件事件和时间事件两种事件类型，调度伪代码：

```
# 事件调度伪代码
def aeProcessEvents():
    # 获取到达时间离当前时间最接近的时间事件
    time_event = aeSearchNearestTime()

    # 计算最接近的时间事件距离到达还有多少毫秒
    remainind_ms = time_event.when - unix_ts_now()
    # 如果事件已到达，那么 remainind_ms 的值可能为负数，设置为 0
    if remainind_ms < 0:
```

```
remaind_ms = 0

# 根据 remaind_ms 的值，创建 timeval 结构
timeval = create_timeval_with_ms(remaind_ms)
# 【阻塞并等待文件事件】产生，最大阻塞时间由传入的timeval结构决定，remaind_ms的值为0时调用后马上返回，不阻塞
aeApiPoll(timeval)

# 处理所有已产生的文件事件
processFileEvents()
# 处理所有已到达的时间事件
processTimeEvents()
```

事件的调度和执行规则：

- aeApiPoll 函数的最大阻塞时间由到达时间最接近当前时间的时间事件决定，可以避免服务器对时间事件进行频繁的轮询（忙等待），也可以确保 aeApiPoll 函数不会阻塞过长时间
- 对文件事件和时间事件的处理都是**同步、有序、原子地执行**，服务器不会中断事件处理，也不会对事件进行抢占，所以两种处理器都要尽可能地减少程序的阻塞时间，并在有需要时**主动让出执行权**，从而降低事件饥饿的可能性
 - 命令回复处理器在写入字节数超过了某个预设常量，就会主动用 break 跳出写入循环，将余下的数据留到下次再写
 - 时间事件也会将非常耗时的持久化操作放到子线程或者子进程执行
- 时间事件在文件事件之后执行，并且事件之间不会出现抢占，所以时间事件的实际处理时间通常会比设定的到达时间稍晚

多线程

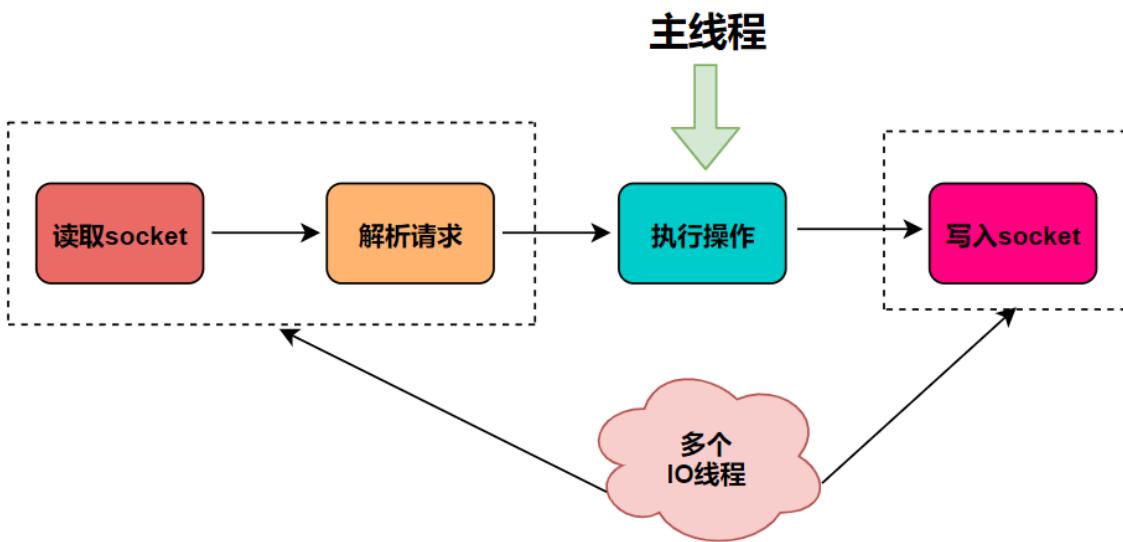
Redis6.0 引入多线程主要是为了提高网络 IO 读写性能，因为这是 Redis 的一个性能瓶颈（Redis 的瓶颈主要受限于内存和网络），多线程只是用来**处理网络数据的读写和协议解析**，执行命令仍然是单线程顺序执行，因此不需要担心线程安全问题。

Redis6.0 的多线程默认是禁用的，只使用主线程。如需开启需要修改 redis 配置文件 `redis.conf`：

```
io-threads-do-reads yesCopy to clipboardErrorCopied
```

开启多线程后，还需要设置线程数，否则是不生效的，同样需要修改 redis 配置文件：

```
io-threads 4 #官网建议4核的机器建议设置为2或3个线程，8核的建议设置为6个线程
```



参考文章：<https://mp.weixin.qq.com/s/dqmiR0ECf4lB6Y2OyK-dyA>

客户端

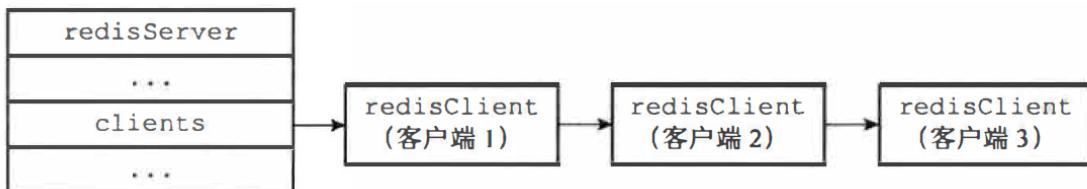
基本介绍

Redis 服务器是典型的一对多程序，一个服务器可以与多个客户端建立网络连接，服务器对每个连接的客户端建立了相应的 redisClient 结构（客户端状态，在服务器端的存储结构），保存了客户端当前的状态信息，以及执行相关功能时需要用到的数据结构

Redis 服务器状态结构的 clients 属性是一个链表，这个链表保存了所有与服务器连接的客户端的状态结构：

```
struct redisServer {
    // 一个链表，保存了所有客户端状态
    list *clients;

    //...
};
```



数据结构

redisClient

客户端的数据结构：

```
typedef struct redisClient {
    // ...

    // 套接字
    int fd;
    // 名字
    robj *name;
    // 标志
    int flags;

    // 输入缓冲区
    sds querybuf;
    // 输出缓冲区 buf 数组
    char buf[REDIS_REPLY_CHUNK_BYTES];
    // 记录了 buf 数组目前已使用的字节数量
    int bufpos;
    // 可变大小的输出缓冲区，链表 + 字符串对象
    list *reply;

    // 命令数组
    rboj **argv;
    // 命令数组的长度
    int argc;
    // 命令的信息
    struct redisCommand *cmd;

    // 是否通过身份验证
    int authenticated;

    // 创建客户端的时间
    time_t ctime;
    // 客户端与服务器最后一次进行交互的时间
    time_t lastinteraction;
    // 输出缓冲区第一次到达软性限制 (soft limit) 的时间
    time_t obuf_soft_limit_reached_time;
}
```

客户端状态包括两类属性

- 一类是比较通用的属性，这些属性很少与特定功能相关，无论客户端执行的是什么工作，都要用到这些属性
- 另一类是和特定功能相关的属性，比如操作数据库时用到的 db 属性和 dict id 属性，执行事务时用到的 mstate 属性，以及执行 WATCH 命令时用到的 watched_keys 属性等，代码中没有列出

套接字

客户端状态的 fd 属性记录了客户端正在使用的套接字描述符，根据客户端类型的不同，fd 属性的值可以是 -1 或者大于 -1 的整数：

- 伪客户端 (fake client) 的 fd 属性的值为 -1，命令请求来源于 AOF 文件或者 Lua 脚本，而不是网络，所以不需要套接字连接
- 普通客户端的 fd 属性的值为大于 -1 的整数，因为合法的套接字描述符不能是 -1

执行 `CLIENT list` 命令可以列出目前所有连接到服务器的普通客户端，不包括伪客户端

名字

在默认情况下，一个连接到服务器的客户端是没有名字的，使用 `CLIENT setname` 命令可以为客户端设置一个名字

标志

客户端的标志属性 flags 记录了客户端的角色以及客户端目前所处的状态，每个标志使用一个常量表示

- flags 的值可以是单个标志：`flags = <flag>`
- flags 的值可以是多个标志的二进制：`flags = <flag1> | <flag2> | ...`

一部分标志记录客户端的角色：

- REDIS_MASTER 表示客户端是一个从服务器，REDIS_SLAVE 表示客户端是一个从服务器，在主从复制时使用
- REDIS_PRE_PSYNC 表示客户端是一个版本低于 Redis2.8 的从服务器，主服务器不能使用 PSYNC 命令与该从服务器进行同步，这个标志只能在 REDIS_SLAVE 标志处于打开状态时使用
- REDIS LUA_CLIENT 表示客户端是专门用于处理 Lua 脚本里面包含的 Redis 命令的伪客户端

一部分标志记录目前客户端所处的状态：

- REDIS_UNIX_SOCKET 表示服务器使用 UNIX 套接字来连接客户端
- REDIS_BLOCKED 表示客户端正在被 BRPOP、BLPOP 等命令阻塞
- REDIS_UNBLOCKED 表示客户端已经从 REDIS_BLOCKED 所表示的阻塞状态脱离，在 REDIS_BLOCKED 标志打开的情况下使用
- REDIS_MULTI 标志表示客户端正在执行事务
- REDIS_DIRTY_CAS 表示事务使用 WATCH 命令监视的数据库键已经被修改
-

缓冲区

客户端状态的输入缓冲区用于保存客户端发送的命令请求，输入缓冲区的大小会根据输入内容动态地缩小或者扩大，但最大大小不能超过 1GB，否则服务器将关闭这个客户端，比如执行 `SET key value`，那么缓冲区 querybuf 的内容：

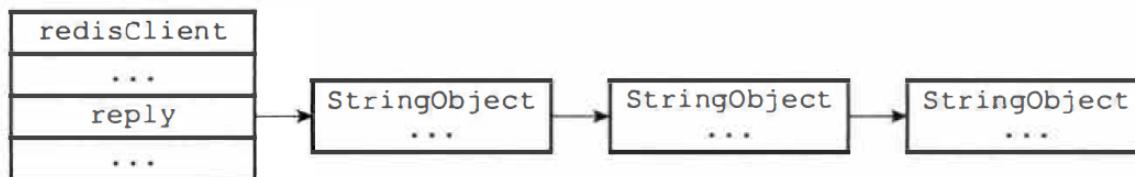
```
*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n#
```

输出缓冲区是服务器用于保存执行客户端命令所得的命令回复，每个客户端都有两个输出缓冲区可用：

- 一个是固定大小的缓冲区，保存长度比较小的回复，比如 OK、简短的字符串值、整数值、错误回复等
- 一个是可变大小的缓冲区，保存那些长度比较大的回复，比如一个非常长的字符串值或者一个包含了很多元素的集合等

buf 是一个大小为 `REDIS_REPLY_CHUNK_BYTES` (常量默认 $16 * 1024 = 16\text{KB}$) 字节的字节数组，bufpos 属性记录了 buf 数组目前已使用的字节数量，当 buf 数组的空间已经用完或者回复数据太大无法放进 buf 数组里，服务器就会开始使用可变大小的缓冲区

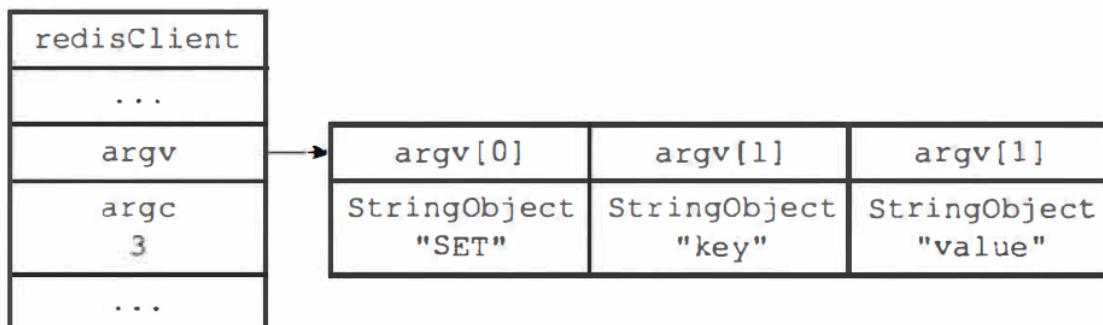
通过使用 reply 链表连接多个字符串对象，可以为客户端保存一个非常长的命令回复，而不必受到固定大小缓冲区 16KB 大小的限制



命令

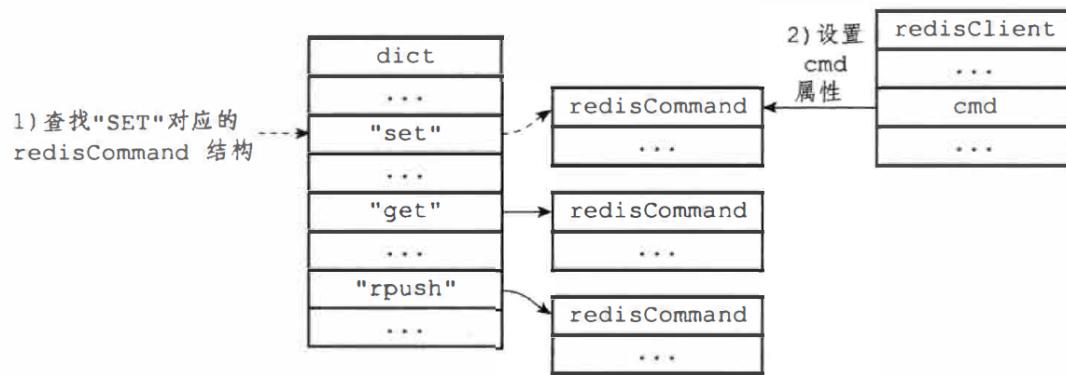
服务器对 querybuf 中的命令请求的内容进行分析，得出的命令参数以及参数的数量分别保存到客户端状态的 argv 和 argc 属性

- argv 属性是一个数组，数组中的每项都是字符串对象，其中 `argv[0]` 是要执行的命令，而之后的其他项则是命令的参数
- argc 属性负责记录 argv 数组的长度



服务器将根据项 `argv[0]` 的值，在命令表中查找命令所对应的命令的 `redisCommand`，将客户端状态的 cmd 指向该结构

命令表是一个字典结构，键是 SDS 结构保存命令的名字；值是命令所对应的 redisCommand 结构，保存了命令的实现函数、命令标志、命令应该给定的参数个数、命令的总执行次数和总消耗时长等统计信息



验证

客户端状态的 authenticated 属性用于记录客户端是否通过了身份验证

- authenticated 值为 0，表示客户端未通过身份验证
- authenticated 值为 1，表示客户端已通过身份验证

当客户端 authenticated = 0 时，除了 AUTH 命令之外，客户端发送的所有其他命令都会被服务器拒绝执行

```
redis> PING
(error) NOAUTH Authentication required.
redis> AUTH 123321
OK
redis> PING
PONG
```

时间

ctime 属性记录了创建客户端的时间，这个时间可以用来计算客户端与服务器已经连接了多少秒，`CLIENT list` 命令的 age 域记录了这个秒数

lastinteraction 属性记录了客户端与服务器最后一次进行互动 (interaction) 的时间，互动可以是客户端向服务器发送命令请求，也可以是服务器向客户端发送命令回复。该属性可以用来计算客户端的空转 (idle) 时长，就是距离客户端与服务器最后一次进行互动已经过去了多少秒，`CLIENT list` 命令的 idle 域记录了这个秒数

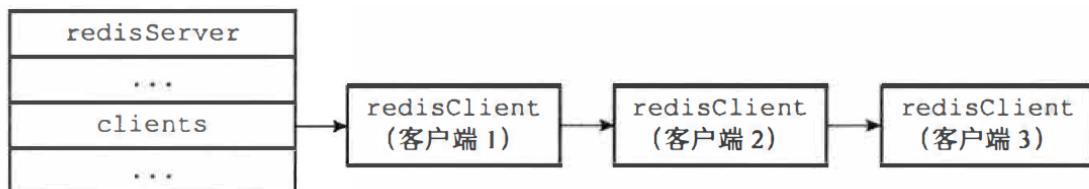
`obuf_soft_limit_reached_time` 属性记录了输出缓冲区第一次到达软性限制 (soft limit) 的时间

生命周期

创建

服务器使用不同的方式来创建和关闭不同类型的客户端

如果客户端是通过网络连接与服务器进行连接的普通客户端，那么在客户端使用 connect 函数连接到服务器时，服务器就会调用连接应答处理器为客户端创建相应的客户端状态，并将这个新的客户端状态添加到服务器状态结构 clients 链表的末尾



服务器会在初始化时创建负责执行 Lua 脚本中包含的 Redis 命令的伪客户端，并将伪客户端关联在服务器状态的 lua_client 属性

```
struct redisServer {  
    // 保存伪客户端  
    redisClient *lua_client;  
  
    //...  
};
```

lua_client 伪客户端在服务器运行的整个生命周期会一直存在，只有服务器被关闭时，这个客户端才会被关闭

载入 AOF 文件时，服务器会创建用于执行 AOF 文件包含的 Redis 命令的伪客户端，并在载入完成之后，关闭这个伪客户端

关闭

一个普通客户端可以因为多种原因而被关闭：

- 客户端进程退出或者被杀死，那么客户端与服务器之间的网络连接将被关闭，从而造成客户端被关闭
- 客户端向服务器发送了带有不符合协议格式的命令请求，那么这个客户端会被**服务器关闭**
- 客户端是 `CLIENT KILL` 命令的目标
- 如果用户为服务器设置了 `timeout` 配置选项，那么当客户端的空转时间超过该值时将被关闭，特殊情况不会被关闭：
 - 客户端是主服务器 (`REDIS_MASTER`) 或者从服务器 (打开了 `REDIS_SLAVE` 标志)
 - 正在被 `BLPOP` 等命令阻塞 (`REDIS_BLOCKED`)
 - 正在执行 `SUBSCRIBE`、`PSUBSCRIBE` 等订阅命令

- 客户端发送的命令请求的大小超过了输入缓冲区的限制大小（默认为 1GB）
- 发送给客户端的命令回复的大小超过了输出缓冲区的限制大小

理论上来说，可变缓冲区可以保存任意长的命令回复，但是为了回复过大占用过多的服务器资源，服务器会时刻检查客户端的输出缓冲区的大小，并在缓冲区的大小超出范围时，执行相应的限制操作：

- 硬性限制 (hard limit)：输出缓冲区的大小超过了硬性限制所设置的大小，那么服务器会关闭客户端（serverCron 函数中执行），积存在输出缓冲区中的所有内容会被直接释放，不会返回给客户端
- 软性限制 (soft limit)：输出缓冲区的大小超过了软性限制所设置的大小，小于硬性限制的大小，服务器的操作：
 - 用属性 obuf_soft_limit_reached_time 记录下客户端到达软性限制的起始时间，继续监视客户端
 - 如果输出缓冲区的大小一直超出软性限制，并且持续时间超过服务器设定的时长，那么服务器将关闭客户端
 - 如果在指定时间内不再超出软性限制，那么客户端就不会被关闭，并且 o_s_l_r_t 属性清零

使用 client-output-buffer-limit 选项可以为普通客户端、从服务器客户端、执行发布与订阅功能的客户端分别设置不同的软性限制和硬性限制，格式：

```
client-output-buffer-limit <class> <hard limit> <soft limit> <soft seconds>

client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
```

- 第一行：将普通客户端的硬性限制和软性限制都设置为 0，表示不限制客户端的输出缓冲区大小
- 第二行：将从服务器客户端的硬性限制设置为 256MB，软性限制设置为 64MB，软性限制的时长为 60 秒
- 第三行：将执行发布与订阅功能的客户端的硬性限制设置为 32MB，软性限制设置为 8MB，软性限制的时长为 60 秒

服务器

执行流程

Redis 服务器与多个客户端建立网络连接，处理客户端发送的命令请求，在数据库中保存客户端执行命令所产生的数据，并通过资源管理来维持服务器自身的运转，所以一个命令请求从发送到获得回复的过程中，客户端和服务器需要完成一系列操作

命令请求

Redis 服务器的命令请求来自 Redis 客户端，当用户在客户端中键入一个命令请求时，客户端会将这个命令请求转换成协议格式，通过连接到服务器的套接字，将协议格式的命令请求发送给服务器

```
SET KEY VALUE ->      # 命令
*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n# 协议格式
```

当客户端与服务器之间的连接套接字因为客户端的写入而变得可读，服务器调用**命令请求处理器**来执行以下操作：

- 读取套接字中协议格式的命令请求，并将其保存到客户端状态的输入缓冲区里面
- 对输入缓冲区中的命令请求进行分析，提取出命令请求中包含的命令参数以及命令参数的个数，然后分别将参数和参数个数保存到客户端状态的 argv 属性和 argc 属性里
- 调用命令执行器，执行客户端指定的命令

最后客户端接收到协议格式的命令回复之后，会将这些回复转换成用户可读的格式打印给用户观看，至此整体流程结束

命令执行

命令执行器开始对命令操作：

- 查找命令：首先根据客户端状态的 argv[0] 参数，在**命令表 (command table)** 中查找参数所指定的命令，并将找到的命令保存到客户端状态的 cmd 属性里面，是一个 redisCommand 结构
命令查找算法与字母的大小写无关，所以命令名字的大小写不影响命令表的查找结果
- 执行预备操作：
 - 检查客户端状态的 cmd 指针是否指向 NULL，根据 redisCommand 检查请求参数的数量是否正确
 - 检查客户端是否通过身份验证
 - 如果服务器打开了 maxmemory 功能，执行命令之前要先检查服务器的内存占用，在有需要时进行内存回收（**逐出算法**）
 - 如果服务器上一次执行 BGSAVE 命令出错，并且服务器打开了 stop-writes-on-bgsave-error 功能，那么如果本次执行的是写命令，服务会拒绝执行，并返回错误
 - 如果客户端当前正在用 SUBSCRIBE 或 PSUBSCRIBE 命令订阅频道，那么服务器会拒绝除了 SUBSCRIBE、SUBSCRIBE、UNSUBSCRIBE、PUNSUBSCRIBE 之外的其他命令
 - 如果服务器正在进行载入数据，只有 sflags 带有 1 标识（比如 INFO、SHUTDOWN、PUBLISH 等）的命令才会被执行
 - 如果服务器执行 Lua 脚本而超时并进入阻塞状态，那么只会执行客户端发来的 SHUTDOWN nosave 和 SCRIPT KILL 命令
 - 如果客户端正在执行事务，那么服务器只会执行客户端发来的 EXEC、DISCARD、MULTI、WATCH 四个命令，其他命令都会被**放进事务队列**中
 - 如果服务器打开了监视器功能，那么会将要执行的命令和参数等信息发送给监视器
- 调用命令的实现函数：被调用的函数会执行指定的操作并产生相应的命令回复，回复会被保存在客户端状态的输出缓冲区里面（buf 和 reply 属性），然后实现函数还会为**客户端的套接字关联命令回复处理器**，这个处理器负责将命令人回复返回给客户端
- 执行后续工作：

- 如果服务器开启了慢查询日志功能，那么慢查询日志模块会检查是否需要为刚刚执行完的命令请求添加一条新的慢查询日志
 - 根据执行命令所耗费的时长，更新命令的 redisCommand 结构的 milliseconds 属性，并将命令 calls 计数器的值增一
 - 如果服务器开启了 AOF 持久化功能，那么 AOF 持久化模块会将执行的命令请求写入到 AOF 缓冲区里面
 - 如果有其他从服务器正在复制当前这个服务器，那么服务器会将执行的命令传播给所有从服务器
- 将命且回复发送给客户端：客户端套接字变为可写状态时，服务器就会执行命且回复处理器，将客户端输出缓冲区中的命且回复发送给客户端，发送完毕之后回复处理器会清空客户端状态的输出缓冲区，为处理下一个命令请求做好准备
-

Command

每个 redisCommand 结构记录了一个Redis 命令的实现信息，主要属性

```
struct redisCommand {
    // 命令的名字，比如"set"
    char *name;

    // 函数指针，指向命令的实现函数，比如setCommand
    // redisCommandProc 类型的定义为 typedef void redisCommandProc(redisClient *c)
    redisCommandProc *proc;

    // 命令参数的个数，用于检查命令请求的格式是否正确。如果这个值为负数-N，那么表示参数的数量大于等于N。
    // 注意命令的名字本身也是一个参数，比如 SET msg "hello"，命令的参数是"SET"、"msg"、"hello" 三个
    int arity;

    // 字符串形式的标识值，这个值记录了命令的属性，，
    // 比如这个命令是写命令还是读命令，这个命令是否允许在载入数据时使用，是否允许在Lua脚本中使用等等
    char *sflags;

    // 对sflags标识进行分析得出的二进制标识，由程序自动生成。服务器对命令标识进行检查时使用的都是 flags 属性
    // 而不是sflags属性，因为对二进制标识的检查可以方便地通过& ^ ~ 等操作来完成
    int flags;

    // 服务器总共执行了多少次这个命令
    long long calls;

    // 服务器执行这个命令所耗费的总时长
    long long milliseconds;
};
```

serverCron

基本介绍

Redis 服务器以周期性事件的方式来运行 serverCron 函数，服务器初始化时读取配置 server.hz 的值，默认为 10，代表每秒钟执行 10 次，即**每隔 100 毫秒执行一次**，执行指令 info server 可以查看

serverCron 函数负责定期对自身的资源和状态进行检查和调整，从而确保服务器可以长期、稳定地运行

- 更新服务器的各类统计信息，比如时间、内存占用、数据库占用情况等
- 清理数据库中的过期键值对
- 关闭和清理连接失效的客户端
- 进行 AOF 或 RDB 持久化操作
- 如果服务器是主服务器，那么对从服务器进行定期同步
- 如果处于集群模式，对集群进行定期同步和连接测试

时间缓存

Redis 服务器中有很多功能需要获取系统的当前时间，而每次获取系统的当前时间都需要执行一次系统调用，为了减少系统调用的执行次数，服务器状态中的 unixtime 属性和 mstime 属性被用作当前时间的缓存

```
struct redisServer {
    // 保存了秒级精度的系统当前UNIX时间戳
    time_t unixtime;
    // 保存了毫秒级精度的系统当前UNIX时间戳
    long long mstime;

};
```

serverCron 函数默认以每 100 毫秒一次的频率更新两个属性，所以属性记录的时间的精确度并不高

- 服务器只会在打印日志、更新服务器的 LRU 时钟、决定是否执行持久化任务、计算服务器上线时间（uptime）这类对时间精确度要求不高的功能上
- 对于为键设置过期时间、添加慢查询日志这种需要高精确度时间的功能来说，服务器还是会再次执行系统调用，从而获得最准确的系统当前时间

LRU 时钟

服务器状态中的 lru_clock 属性保存了服务器的 LRU 时钟

```
struct redisServer {  
    // 默认每10秒更新一次的时钟缓存，用于计算键的空转(idle)时长。  
    unsigned lruClock:22;  
};
```

每个 Redis 对象都会有一个 lru 属性，这个 lru 属性保存了对象最后一次被命令访问的时间

```
typedef struct redisObject {  
    unsigned lru:22;  
} rObj;
```

当服务器要计算一个数据库键的空转时间（即数据库键对应的值对象的空转时间），程序会用服务器的 lrclock 属性记录的时间减去对象的 lru 属性记录的时间

serverCron 函数默认以每 100 毫秒一次的频率更新这个属性，所以得出的空转时间也是模糊的

命令次数

serverCron 中的 trackOperationsPerSecond 函数以每 100 毫秒一次的频率执行，函数功能是以**抽样计算**的方式，估算并记录服务器在最近一秒钟处理的命令请求数量，这个值可以通过 INFO status 命令的 instantaneous_ops_per_sec 域查看：

```
redis> INFO stats  
# Stats  
instantaneous_ops_per_sec:6
```

根据上一次抽样时间 ops_sec_last_sample_time 和当前系统时间，以及上一次已执行的命令数 ops_sec_last_sample_ops 和服务器当前已经执行的命令数，计算出两次函数调用期间，服务器平均每毫秒处理了多少个命令请求，该值乘以 1000 得到每秒内的执行命令的估计值，放入 ops_sec_samples 环形数组里

```
struct redisServer {  
    // 上一次进行抽样的时间  
    long long ops_sec_last_sample_time;  
    // 上一次抽样时，服务器已执行命令的数量  
    long long ops_sec_last_sample_ops;  
    // REDIS_OPS_SEC_SAMPLES 大小（默认值为16）的环形数组，数组的每一项记录一次的抽样结果  
    long long ops_sec_samples[REDIS_OPS_SEC_SAMPLES];  
    // ops_sec_samples 数组的索引值，每次抽样后将值自增一，值为16时重置为0，让数组成为一个环形数组  
    int ops_sec_idx;  
};
```

内存峰值

服务器状态里的 stat_peak_memory 属性记录了服务器内存峰值大小，循环函数每次执行时都会查看服务器当前使用的内存数量，并与 stat_peak_memory 保存的数值进行比较，设置为较大的值

```
struct redisServer {  
    // 已使用内存峰值  
    size_t stat_peak_memory;  
};
```

INFO memory 命令的 used_memory_peak 和 used_memory_peak_human 两个域分别以两种格式记录了服务器的内存峰值：

```
redis> INFO memory  
# Memory  
...  
used_memory_peak:501824  
used_memory_peak_human:490.06K
```

SIGTERM

服务器启动时，Redis 会为服务器进程的 SIGTERM 信号关联处理器 sigtermHandler 函数，该信号处理器负责在服务器接到 SIGTERM 信号时，打开服务器状态的 shutdown_asap 标识

```
struct redisServer {  
    // 关闭服务器的标识：值为1时关闭服务器，值为0时不操作  
    int shutdown_asap;  
};
```

每次 serverCron 函数运行时，程序都会对服务器状态的 shutdown_asap 属性进行检查，并根据属性的值决定是否关闭服务器

服务器在接到 SIGTERM 信号之后，关闭服务器并打印相关日志的过程：

```
[6794] signal handler] (1384435690) Received SIGTERM, scheduling shutdown ...  
[6794] 14 Nov 21:28:10.108 # User requested shutdown ...  
[6794] 14 Nov 21:28:10.108 * Saving the final RDB snapshot before exiting.  
[6794] 14 Nov 21:28:10.161 * DB saved on disk  
[6794] 14 Nov 21:28:10.161 # Redis is now ready to exit, bye bye ...
```

管理资源

serverCron 函数每次执行都会调用 clientsCron 和 databasesCron 函数，进行管理客户端资源和数据库资源

clientsCron 函数对一定数量的客户端进行以下两个检查：

- 如果客户端与服务器之间的连接已经超时（很长一段时间客户端和服务器都没有互动），那么程序释放这个客户端
- 如果客户端在上一次执行命令请求之后，输入缓冲区的大小超过了一定的长度，那么程序会释放客户端当前的输入缓冲区，并重新创建一个默认大小的输入缓冲区，从而防止客户端的输入缓冲区耗费了过多的内存

databasesCron 函数会对服务器中的一部分数据库进行检查，删除其中的过期键，并在有需要时对字典进行收缩操作

持久状态

服务器状态中记录执行 BGSAVE 命令和 BGREWRITEAOF 命令的子进程的 ID

```
struct redisServer {
    // 记录执行BGSAVE命令的子进程的ID，如果服务器没有在执行BGSAVE，那么这个属性的值为-1
    pid_t rdb_child_pid;
    // 记录执行BGREWRITEAOF命令的子进程的ID，如果服务器没有在执行那么这个属性的值为-1
    pid_t aof_child_pid
};
```

serverCron 函数执行时，会检查两个属性的值，只要其中一个属性的值不为 -1，程序就会执行一次 wait3 函数，检查子进程是否有信号发来服务器进程：

- 如果有信号到达，那么表示新的 RDB 文件已经生成或者 AOF 重写完毕，服务器需要进行相应命令的后续操作，比如用新的 RDB 文件替换现有的 RDB 文件，用重写后的 AOF 文件替换现有的 AOF 文件
- 如果没有信号到达，那么表示持久化操作未完成，程序不做动作

如果两个属性的值都为 -1，表示服务器没有进行持久化操作

- 查看是否有 BGREWRITEAOF 被延迟，然后执行 AOF 后台重写
- 查看服务器的自动保存条件是否已经被满足，并且服务器没有在进行持久化，就开始一次新的 BGSAVE 操作

因为条件 1 可能会引发一次 AOF，所以在这个检查中会再次确认服务器是否已经在执行持久化操作

- 检查服务器设置的 AOF 重写条件是否满足，条件满足并且服务器没有进行持久化，就进行一次 AOF 重写

如果服务器开启了 AOF 持久化功能，并且 AOF 缓冲区里还有待写入的数据，那么 serverCron 函数会调用相应的程序，将 AOF 缓冲区中的内容写入到 AOF 文件里

延迟执行

在服务器执行 BGSAVE 命令的期间，如果客户端发送 BGREWRITEAOF 命令，那么服务器会将 BGREWRITEAOF 命令的执行时间延迟到 BGSAVE 命令执行完毕之后，用服务器状态的 aof_rewrite_scheduled 属性标识延迟与否

```
struct redisServer {  
    // 如果值为1，那么表示有 BGREWRITEAOF命令被延迟了  
    int aof_rewrite_scheduled;  
};
```

serverCron 函数会检查 BGSAVE 或者 BGREWRITEAOF 命令是否正在执行，如果这两个命令都没在执行，并且 aof_rewrite_scheduled 属性的值为 1，那么服务器就会执行之前被推延的 BGREWRITEAOF 命令

执行次数

服务器状态的 cronloops 属性记录了 serverCron 函数执行的次数

```
struct redisServer {  
    // serverCron 函数每执行一次，这个属性的值就增 1  
    int cronloops;  
};
```

缓冲限制

服务器会关闭那些输入或者输出**缓冲区大小超出限制**的客户端

初始化

初始结构

一个 Redis 服务器从启动到能够接受客户端的命令请求，需要经过一系列的初始化和设置过程

第一步：创建一个 redisServer 类型的实例变量 server 作为服务器的状态，并为结构中的各个属性设置默认值，由 initServerConfig 函数进行初始化一般属性：

- 设置服务器的运行 ID、默认运行频率、默认配置文件路径、默认端口号、默认 RDB 持久化条件和 AOF 持久化条件
- 初始化服务器的 LRU 时钟，创建命令表

第二步：载入配置选项，用户可以通过给定配置参数或者指定配置文件，对 server 变量相关属性的默认值进行修改

第三步：初始化服务器数据结构（除了命令表之外），因为服务器**必须先载入用户指定的配置选项才能正确地对数据结构进行初始化**，所以载入配置完成后才进行数据结构的初始化，服务器将调用 initServer 函数：

- server.clients 链表，记录了的客户端的状态结构；server.db 数组，包含了服务器的所有数据库
- 用于保存频道订阅信息的 server.pubsub_channels 字典，以及保存模式订阅信息的 server.pubsub_patterns 链表
- 用于执行 Lua 脚本的 Lua 环境 server.lua
- 保存慢查询日志的 server.slowlog 属性

initServer 还进行了非常重要的设置操作：

- 为服务器设置进程信号处理器
- 创建共享对象，包含 OK、ERR、**整数 1 到 10000 的字符串对象等**
- **打开服务器的监听端口**
- **为 serverCron 函数创建时间事件**，等待服务器正式运行时执行 serverCron 函数
- 如果 AOF 持久化功能已经打开，那么打开现有的 AOF 文件，如果 AOF 文件不存在，那么创建并打开一个新的 AOF 文件，为 AOF 写入做好准备
- **初始化服务器的后台 I/O 模块 (BIO)**，为将来的 I/O 操作做好准备

当 initServer 函数执行完毕之后，服务器将用 ASCII 字符在日志中打印出 Redis 的图标，以及 Redis 的版本号信息

还原状态

在完成了对服务器状态的初始化之后，服务器需要载入RDB文件或者AOF文件，并根据文件记录的内容来还原服务器的数据库状态：

- 如果服务器启用了 AOF 持久化功能，那么服务器使用 AOF 文件来还原数据库状态
- 如果服务器没有启用 AOF 持久化功能，那么服务器使用 RDB 文件来还原数据库状态

当服务器完成数据库状态还原工作之后，服务器将在日志中打印出载入文件并还原数据库状态所耗费的时长

```
[7171] 22 Nov 22:43:49.084 * DB loaded from disk: 0.071 seconds
```

驱动循环

在初始化的最后一步，服务器将打印出以下日志，并开始**执行服务器的事件循环** (loop)

```
[7171] 22 Nov 22:43:49.084 * The server is now ready to accept connections on port 6379
```

服务器现在开始可以接受客户端的连接请求，并处理客户端发来的命令请求了

慢日志

基本介绍

Redis 的慢查询日志功能用于记录执行时间超过给定时长的命令请求，通过产生的日志来监视和优化查询速度

服务器配置有两个和慢查询日志相关的选项：

- slowlog-log-slower-than 选项指定执行时间超过多少微秒的命令请求会被记录到日志上
- slowlog-max-len 选项指定服务器最多保存多少条慢查询日志

服务器使用先进先出 FIFO 的方式保存多条慢查询日志，当服务器存储的慢查询日志数量等于 slowlog-max-len 选项的值时，在添加一条新的慢查询日志之前，会先将最旧的一条慢查询日志删除

配置选项可以通过 CONFIG SET option value 命令进行设置

常用命令：

```
SLOWLOG GET [n] # 查看 n 条服务器保存的慢日志
SLOWLOG LEN      # 查看日志数量
SLOWLOG RESET    # 清除所有慢查询日志
```

日志保存

服务器状态中包含了慢查询日志功能有关的属性：

```
struct redisServer {
    // 下一条慢查询日志的ID
    long long slowlog_entry_id;

    // 保存了所有慢查询日志的链表
    list *slowlog;

    // 服务器配置选项的值
    long long slowlog-log-slower-than;
    // 服务器配置选项的值
    unsigned long slowlog_max_len;
}
```

slowlog_entry_id 属性的初始值为 0，每当创建一条新的慢查询日志时，这个属性就会用作新日志的 id 值，之后该属性增一

slowlog 链表保存了服务器中的所有慢查询日志，链表中的每个节点是一个 slowlogEntry 结构，代表一条慢查询日志：

```
typedef struct slowlogEntry {
    // 唯一标识符
    long long id;
    // 命令执行时的时间，格式为UNIX时间戳
    time_t time;
    // 执行命令消耗的时间，以微秒为单位
    long long duration;
    // 命令与命令参数
    robj **argv;
    // 命令与命令参数的数量
    int argc;
}
```

添加日志

在每次执行命令的前后，程序都会记录微秒格式的当前 UNIX 时间戳，两个时间之差就是执行命令所耗费的时长，函数会检查命令的执行时长是否超过 slowlog-log-slower-than 选项所设置：

- 如果是的话，就为命令创建一个新的日志，并将新日志添加到 slowlog 链表的表头
- 检查慢查询日志的长度是否超过 slowlog-max-len 选项所设置的长度，如果是将多出来的日志从 slowlog 链表中删除掉
- 将 redisServer.slowlog_entry_id 的值增 1

数据结构

字符串

SDS

Redis 构建了简单动态字符串（SDS）的数据类型，作为 Redis 的默认字符串表示，包含字符串的键值对在底层都是由 SDS 实现

```

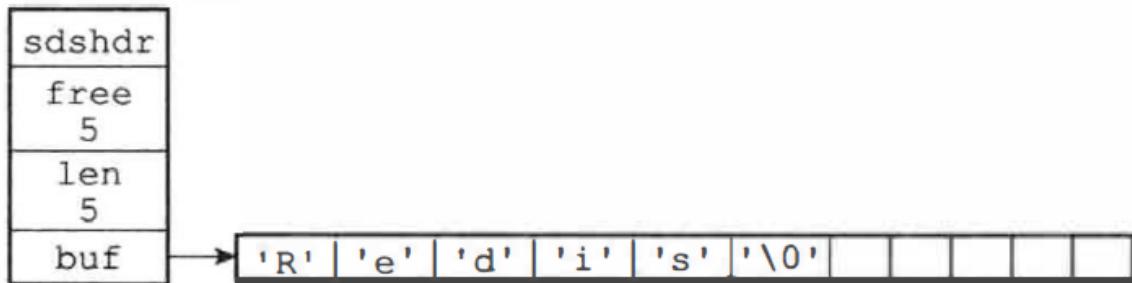
struct sdshdr {
    // 记录buf数组中已使用字节的数量，等于 SDS 所保存字符串的长度
    int len;

    // 记录buf数组中未使用字节的数量
    int free;

    // 【字节】数组，用于保存字符串（不是字符数组）
    char buf[];
};

```

SDS 遵循 C 字符串以空字符结尾的惯例，保存空字符的 1 字节不计算在 len 属性，SDS 会自动为空字符分配额外的 1 字节空间和添加空字符到字符串末尾，所以空字符对于 SDS 的使用者来说是完全透明的



对比

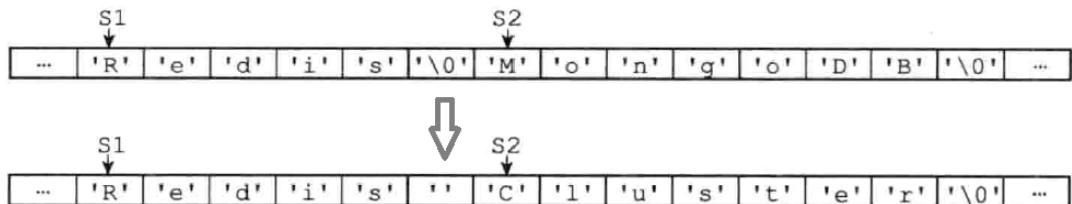
常数复杂度获取字符串长度：

- C 字符串不记录自身的长度，获取时需要遍历整个字符串，遇到空字符串为止，时间复杂度为 $O(N)$
- SDS 获取字符串长度的时间复杂度为 $O(1)$ ，设置和更新 SDS 长度由函数底层自动完成

杜绝缓冲区溢出：

- C 字符串调用 `strcat` 函数拼接字符串时，如果字符串内存不够容纳目标字符串，就会造成缓冲区溢出 (Buffer Overflow)

`s1` 和 `s2` 是内存中相邻的字符串，执行 `strcat(s1, " cluster")` (有空格)：



- SDS 空间分配策略：当对 SDS 进行修改时，首先检查 SDS 的空间是否满足修改所需的要求，如果不满足会自动将 SDS 的空间扩展至执行修改所需的大小，然后执行实际的修改操作，避免了缓冲区溢出的问题

二进制安全：

- C 字符串中的字符必须符合某种编码（比如 ASCII）方式，除了字符串末尾以外其他位置不能包含空字符，否则会被误认为是字符串的结尾，所以只能保存文本数据

- SDS 的 API 都是二进制安全的，使用字节数组 buf 保存一系列的二进制数据，**使用 len 属性来判断数据的结尾**，所以可以保存图片、视频、压缩文件等二进制数据

兼容 C 字符串的函数：SDS 会在为 buf 数组分配空间时多分配一个字节来保存空字符，所以可以重用一部分 C 字符串函数库的函数

内存

C 字符串每次增长或者缩短都会进行一次内存重分配，拼接操作通过重分配扩展底层数组空间，截断操作通过重分配释放不使用的内存空间，防止出现内存泄露

SDS 通过未使用空间解除了字符串长度和底层数组长度之间的关联，在 SDS 中 buf 数组的长度不一定就是字符数量加一，数组里面可以包含未使用的字节，字节的数量由 free 属性记录

内存重分配涉及复杂的算法，需要执行**系统调用**，是一个比较耗时的操作，SDS 的两种优化策略：

- 空间预分配：当 SDS 需要进行空间扩展时，程序不仅会为 SDS 分配修改所必需的空间，还会为 SDS 分配额外的未使用空间
 - 对 SDS 修改之后，SDS 的长度（len 属性）小于 1MB，程序分配和 len 属性同样大小的未使用空间，此时 len 和 free 相等
s 为 Redis，执行 `sds_cat(s, " cluster")` 后，len 变为 13 字节，所以也分配了 13 字节的 free 空间，总长度变为 27 字节（额外的一字节保存空字符， $13 + 13 + 1 = 27$ ）



- 对 SDS 修改之后，SDS 的长度大于等于 1MB，程序会分配 1MB 的未使用空间

在扩展 SDS 空间前，API 会先检查 free 空间是否足够，如果足够就无需执行内存重分配，所以通过预分配策略，SDS 将连续增长 N 次字符串所需内存的重分配次数从**必定 N 次降低为最多 N 次**

- 惰性空间释放：当 SDS 缩短字符串时，程序并不立即使用内存重分配来回收缩短后多出来的字节，而是使用 free 属性将这些字节的数量记录起来，并等待将来复用
SDS 提供了相应的 API 来真正释放 SDS 的未使用空间，所以不用担心空间惰性释放策略造成的内存浪费问题

链表

链表提供了高效的节点重排能力，C 语言并没有内置这种数据结构，所以 Redis 构建了链表数据类型

链表节点：

```

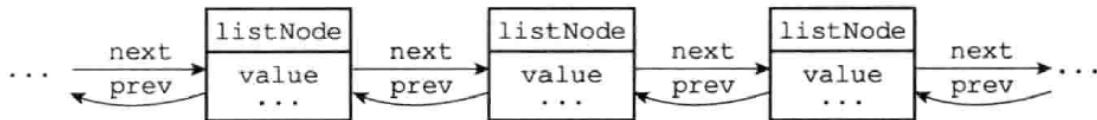
typedef struct listNode {
    // 前置节点
    struct listNode *prev;

    // 后置节点
    struct listNode *next;

    // 节点的值
    void *value
} listNode;

```

多个 listNode 通过 prev 和 next 指针组成**双端链表**:



list 链表结构: 提供了表头指针 head 、表尾指针 tail 以及链表长度计数器 len

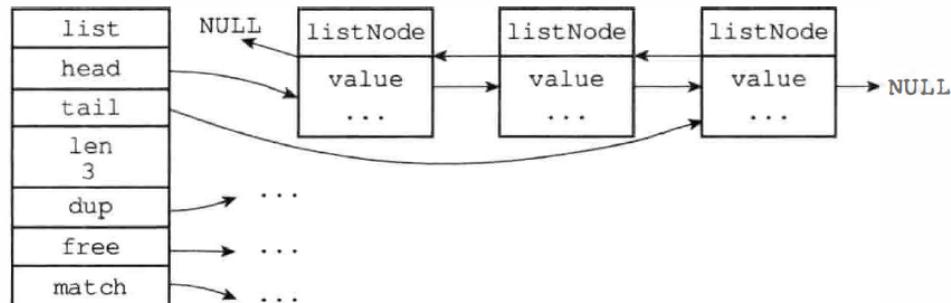
```

typedef struct list {
    // 表头节点
    listNode *head;
    // 表尾节点
    listNode *tail;

    // 链表所包含的节点数量
    unsigned long len;

    // 节点值复制函数, 用于复制链表节点所保存的值
    void *(*dup) (void *ptr);
    // 节点值释放函数, 用于释放链表节点所保存的值
    void (*free) (void *ptr);
    // 节点值对比函数, 用于对比链表节点所保存的值和另一个输入值是否相等
    int (*match) (void *ptr, void *key);
} list;

```



Redis 链表的特性:

- 双端: 链表节点带有 prev 和 next 指针, 获取某个节点的前置节点和后置节点的时间复杂度都是 O(1)
- 无环: 表头节点的 prev 指针和表尾节点的 next 指针都指向 NULL, 对链表的访问以 NULL 为终点
- 带表头指针和表尾指针: 通过 list 结构的 head 指针和 tail 指针, 获取链表的表头节点和表尾节点的时间复杂度为 O(1)
- 带链表长度计数器: 使用 len 属性来对 list 持有的链表节点进行计数, 获取链表中节点数量的时间复杂度为 O(1)

- 多态：链表节点使用 `void *` 指针来保存节点值，并且可以通过 `dup`、`free`、`match` 三个属性为节点值设置类型特定函数，所以链表可以保存各种**不同类型的值**

字典

哈希表

Redis 字典使用的哈希表结构：

```
typedef struct dictht {
    // 哈希表数组，数组中每个元素指向 dictEntry 结构
    dictEntry **table;

    // 哈希表大小，数组的长度
    unsigned long size;

    // 哈希表大小掩码，用于计算索引值，总是等于 【size-1】
    unsigned long sizemask;

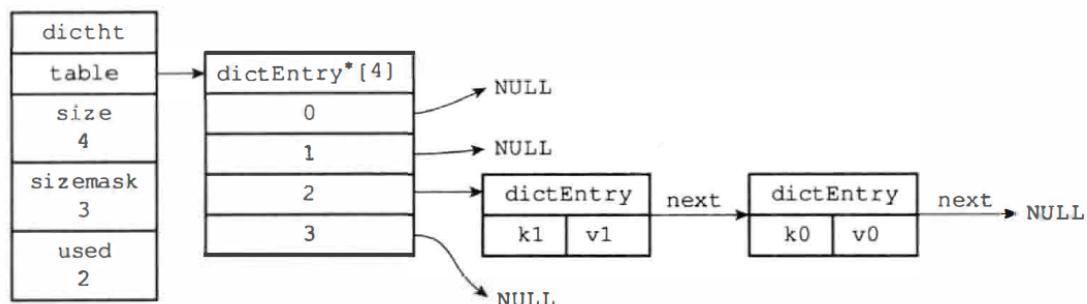
    // 该哈希表已有节点的数量
    unsigned long used;
} dictht;
```

哈希表节点结构：

```
typedef struct dictEntry {
    // 键
    void *key;

    // 值，可以是一个指针，或者整数
    union {
        void *val; // 指针
        uint64_t u64;
        int64_t s64;
    }

    // 指向下个哈希表节点，形成链表，用来解决冲突问题
    struct dictEntry *next;
} dictEntry;
```



字典结构

字典，又称为符号表、关联数组、映射（Map），用于保存键值对的数据结构，字典中的每个键都是独一无二的。底层采用哈希表实现，一个哈希表包含多个哈希表节点，每个节点保存一个键值对

```
typedef struct dict {
    // 类型特定函数
    dictType *type;

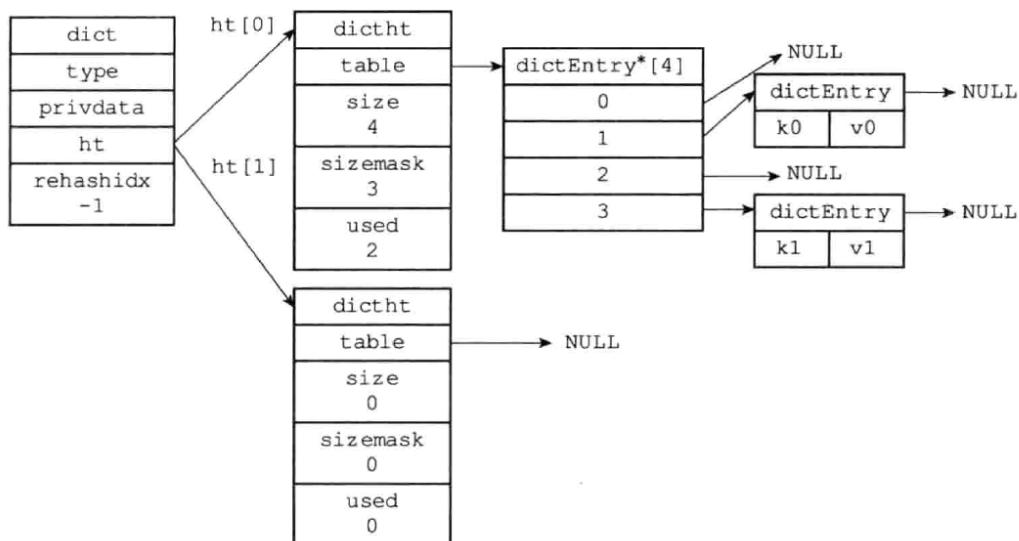
    // 私有数据
    void *privdata;

    // 哈希表，数组中的每个项都是一个dictht哈希表，
    // 一般情况下字典只使用 ht[0] 哈希表， ht[1] 哈希表只会在对 ht[0] 哈希表进行 rehash 时
    使用
    dictht ht[2];

    // rehash 索引，当 rehash 不在进行时，值为 -1
    int rehashidx;
} dict;
```

type 属性和 privdata 属性是针对不同类型的键值对，为创建多态字典而设置的：

- type 属性是指向 dictType 结构的指针，每个 dictType 结构保存了一簇用于操作特定类型键值对的函数，Redis 会为用途不同的字典设置不同的类型特定函数
- privdata 属性保存了需要传给那些类型特定函数的可选参数



哈希冲突

Redis 使用 MurmurHash 算法来计算键的哈希值，这种算法的优点在于，即使输入的键是有规律的，算法仍能给出一个很好的随机分布性，并且算法的计算速度也非常快

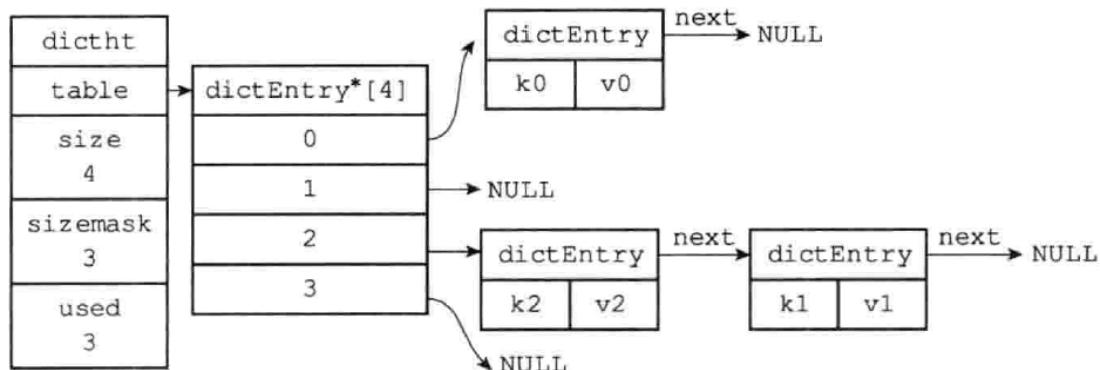
将一个新的键值对添加到字典里，需要先根据键 key 计算出哈希值，然后进行取模运算（取余）：

```
index = hash & dict->ht[x].sizemask
```

当有两个或以上数量的键被分配到了哈希表数组的同一个索引上时，就称这些键发生了哈希冲突 (collision)

Redis 的哈希表使用链地址法 (separate chaining) 来解决键哈希冲突，每个哈希表节点都有一个 next 指针，多个节点通过 next 指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来，这就解决了键冲突的问题

dictEntry 节点组成的链表没有指向链表表尾的指针，为了速度考虑，程序总是将新节点添加到链表的表头位置 (**头插法**)，时间复杂度为 O(1)



负载因子

负载因子的计算方式：哈希表中的**节点数量** / 哈希表的大小（**长度**）

```
load_factor = ht[0].used / ht[0].size
```

为了让哈希表的负载因子 (load factor) 维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序会自动对哈希表的大小进行相应的扩展或者收缩

哈希表执行扩容的条件：

- 服务器没有执行 BGSAVE 或者 BGREWRITEAOF 命令，哈希表的负载因子大于等于 1
- 服务器正在执行 BGSAVE 或者 BGREWRITEAOF 命令，哈希表的负载因子大于等于 5

原因：执行该命令的过程中，Redis 需要创建当前服务器进程的子进程，而大多数操作系统都采用写时复制 (copy-on-write) 技术来优化子进程的使用效率，通过提高执行扩展操作的负载因子，尽可能地避免在子进程存在期间进行哈希表扩展操作，可以避免不必要的内存写入操作，最大限度地节约内存

哈希表执行收缩的条件：负载因子小于 0.1 (自动执行，serveCron 中检测)

重新散列

扩展和收缩哈希表的操作通过 rehash（重新散列）来完成，步骤如下：

- 为字典的 ht[1] 哈希表分配空间，空间大小的分配情况：
 - 如果执行的是扩展操作，ht[1] 的大小为第一个大于等于 $\lfloor \text{ht}[0].used * 2 \rfloor$ 的 2^n
 - 如果执行的是收缩操作，ht[1] 的大小为第一个大于等于 $\lceil \text{ht}[0].used \rceil$ 的 2^n
- 将保存在 ht[0] 中所有的键值对重新计算哈希值和索引值，迁移到 ht[1] 上
- 当 ht[0] 包含的所有键值对都迁移到了 ht[1] 之后（ht[0] 变为空表），释放 ht[0]，将 ht[1] 设置为 ht[0]，并在 ht[1] 创建一个新的空白哈希表，为下一次 rehash 做准备

如果哈希表里保存的键值对数量很少，rehash 就可以在瞬间完成，但是如果哈希表里数据很多，那么要一次性将这些键值对全部 rehash 到 ht[1] 需要大量计算，可能会导致服务器在一段时间内停止服务

Redis 对 rehash 做了优化，使 rehash 的动作并不是一次性、集中式的完成，而是分多次，渐进式的完成，又叫**渐进式 rehash**

- 为 ht[1] 分配空间，此时字典同时持有 ht[0] 和 ht[1] 两个哈希表
- 在字典中维护了一个索引计数器变量 rehashidx，并将变量的值设为 0，表示 rehash 正式开始
- 在 rehash 进行期间，每次对字典执行增删改查操作时，程序除了执行指定的操作以外，还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1]，rehash 完成之后将 **rehashidx 属性的值增一**
- 随着字典操作的不断执行，最终在某个时间点 ht[0] 的所有键值对都被 rehash 至 ht[1]，将 rehashidx 属性的值设为 -1

渐进式 rehash 采用**分而治之**的方式，将 rehash 键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式 rehash 带来的庞大计算量

渐进式 rehash 期间的哈希表操作：

- 字典的查找、删除、更新操作会在两个哈希表上进行，比如查找一个键会先在 ht[0] 上查找，找不到就去 ht[1] 继续查找
- 字典的添加操作会直接在 ht[1] 上添加，不在 ht[0] 上进行任何添加

跳跃表

底层结构

跳跃表（skiplist）是一种有序（默认升序）的数据结构，在链表的基础上**增加了多级索引以提升查找的效率**，索引是占内存的，所以是一个**空间换时间**的方案，跳表平均 $O(\log N)$ 、最坏 $O(N)$ 复杂度的节点查找，效率与平衡树相当但是实现更简单

原始链表中存储的有可能是很大的对象，而索引结点只需要存储关键值和几个指针，并不需要存储对象，因此当节点本身比较大或者元素数量比较多的时候，其优势可以被放大，而缺点（占内存）则可以忽略

Redis 只在两个地方应用了跳跃表，一个是实现有序集合键，另一个是在集群节点中用作内部数据结构

```
typedef struct zskiplist {
    // 表头节点和表尾节点, O(1) 的时间复杂度定位头尾节点
    struct zskiplistNode *head, *tail;

    // 表的长度，也就是表内的节点数量（表头节点不计算在内）
    unsigned long length;

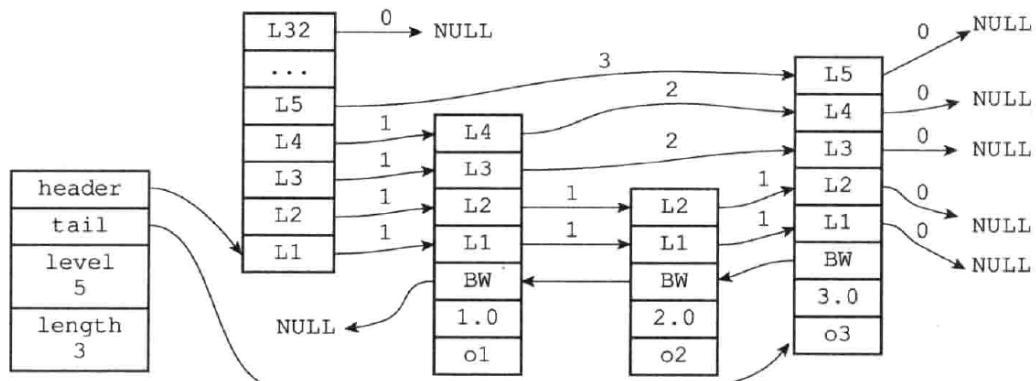
    // 表中层数最大的节点的层数（表头节点的层高不计算在内）
    int level
} zskiplist;
```

```
typedef struct zskiplistNode {
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 跨度
        unsigned int span;
    } level[];
}

// 后退指针
struct zskiplistNode *backward;

// 分值
double score;

// 成员对象
robj *obj;
} zskiplistNode;
```



属性分析

层：level 数组包含多个元素，每个元素包含指向其他节点的指针。根据幕次定律（power law，越大的数出现的概率越小）随机生成一个介于 1 和 32 之间的值（Redis5 之后最大为 64）作为 level 数组的大小，这个大小就是层的高度，节点的第一层是 level[0] = L1

前进指针：forward 用于从表头到表尾方向**正序（升序）遍历节点**，遇到 NULL 停止遍历

跨度：span 用于记录两个节点之间的距离，用来计算排位（rank）：

- 两个节点之间的跨度越大相距的就越远，指向 NULL 的所有前进指针的跨度都为 0
- 在查找某个节点的过程中，**将沿途访问过的所有层的跨度累计起来，结果就是目标节点在跳跃表中的排位**，按照上图所示：

查找分值为 3.0 的节点，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为 3，所以目标节点在跳跃表中的排位为 3

查找分值为 2.0 的节点，沿途经历的层：经过了两个跨度为 1 的节点，因此可以计算出目标节点在跳跃表中的排位为 2

后退指针：backward 用于从表尾到表头方向**逆序（降序）遍历节点**

分值：score 属性一个 double 类型的浮点数，跳跃表中的所有节点都按分值从小到大来排序

成员对象：obj 属性是一个指针，指向一个 SDS 字符串对象。同一个跳跃表中，各个节点保存的**成员对象必须是唯一的**，但是多个节点保存的分值可以是相同的，分值相同的节点将按照成员对象在字典序中的大小来进行排序（从小到大）

个人笔记：JUC → 并发包 → ConcurrentSkipListMap 详解跳跃表

整数集合

底层结构

整数集合（intset）是用于保存整数值的集合数据结构，是 Redis 集合键的底层实现之一

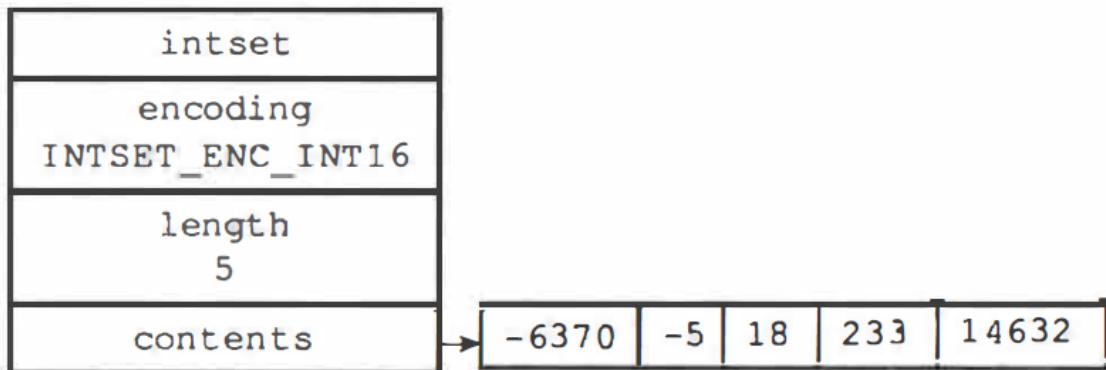
```
typedef struct intset {
    // 编码方式
    uint32_t encoding;

    // 集合包含的元素数量，也就是 contents 数组的长度
    uint32_t length;

    // 保存元素的数组
    int8_t contents[];
} intset;
```

encoding 取值为三种：INTSET_ENC_INT16、INTSET_ENC_INT32、INTSET_ENC_INT64

整数集合的每个元素都是 contents 数组的一个数组项 (item) , 在数组中按值的大小从小到大**有序排列** , 并且数组中**不包含任何重复项**。虽然 contents 属性声明为 int8_t 类型, 但实际上数组并不保存任何 int8_t 类型的值, 真正类型取决于 encoding 属性

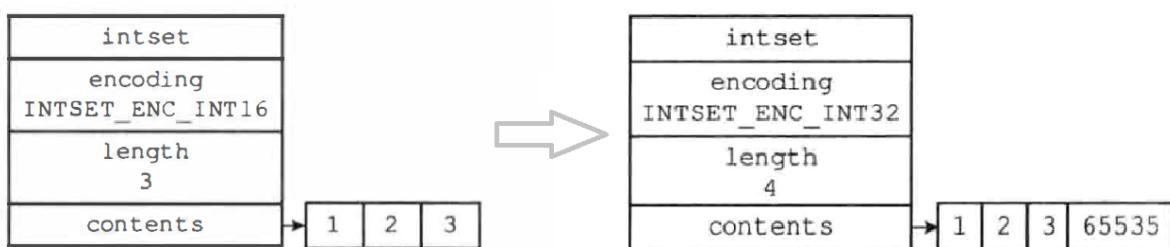


说明: 底层存储结构是数组, 所以为了保证有序性和不重复性, 每次添加一个元素的时间复杂度是 O(N)

类型升级

整数集合添加的新元素的类型比集合现所有所有元素的类型都要长时, 需要先进行升级 (upgrade) , 升级流程:

- 根据新元素的类型长度以及集合元素的数量 (包括新元素在内) , 扩展整数集合底层数组的空间大小
- 将底层数组现有的所有元素都转换成与新元素相同的类型, 并将转换后的元素放入正确的位置, 放置过程保证数组的有序性
- 将新元素添加到底层数组里



每次向整数集合添加新元素都可能会引起升级, 而每次升级都需要对底层数组中的所有元素进行类型转换, 所以向整数集合添加新元素的时间复杂度为 O(N)

引发升级的新元素的长度总是比整数集合并所有所有元素的长度都大, 所以这个新元素的值要么就大于所有现有元素, 要么就小于所有现有元素, 升级之后新元素的摆放位置:

- 在新元素小于所有现有元素的情况下, 新元素会被放置在底层数组的最开头 (索引 0)
- 在新元素大于所有现有元素的情况下, 新元素会被放置在底层数组的最末尾 (索引 length-1)

整数集合升级策略的优点:

- 提升整数集合的灵活性：C 语言是静态类型语言，为了避免类型错误通常不会将两种不同类型的值放在同一个数据结构里面，整数集合可以自动升级底层数组来适应新元素，所以可以随意的添加整数
- 节约内存：要让数组可以同时保存 int16、int32、int64 三种类型的值，可以直接使用 int64_t 类型的数组作为整数集合的底层实现，但是会造成内存浪费，整数集合可以确保升级操作只会在有需要的时候进行，尽量节省内存

整数集合**不支持降级操作**，一旦对数组进行了升级，编码就会一直保持升级后的状态

压缩列表

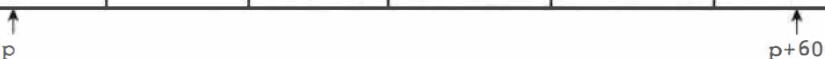
底层结构

压缩列表 (ziplist) 是 Redis 为了节约内存而开发的，是列表键和哈希键的底层实现之一。是由一系列特殊编码的连续内存块组成的顺序型 (sequential) 数据结构，一个压缩列表可以包含任意多个节点 (entry)，每个节点可以保存一个字节数组或者一个整数值

| | | | | | | | |
|--------|--------|-------|--------|--------|-----|--------|-------|
| zbytes | zltail | zllen | entry1 | entry2 | ... | entryN | zlend |
|--------|--------|-------|--------|--------|-----|--------|-------|

- zbytes: uint32_t 类型 4 字节，记录整个压缩列表占用的内存字节数，在对压缩列表进行内存重分配或者计算 zlend 的位置时使用
- zltail: uint32_t 类型 4 字节，记录压缩列表表尾节点距离起始地址有多少字节，通过这个偏移量程序无须遍历整个压缩列表就可以确定表尾节点的地址
- zllen: uint16_t 类型 2 字节，记录了压缩列表包含的节点数量，当该属性的值小于 UINT16_MAX (65535) 时，该值就是压缩列表中节点的数量；当这个值等于 UINT16_MAX 时节点的真实数量需要遍历整个压缩列表才能计算得出
- entryX: 列表节点，压缩列表中的各个节点，**节点的长度由节点保存的内容决定**
- zlend: uint8_t 类型 1 字节，是一个特殊值 0xFF (255)，用于标记压缩列表的末端

| | | | | | | |
|----------------|----------------|--------------|--------|--------|--------|---------------|
| zbytes 0x50 | zltail 0x3c | zllen 0x3 | entry1 | entry2 | entry3 | zlend 0xFF |
|----------------|----------------|--------------|--------|--------|--------|---------------|



列表 zbytes 属性的值为 0x50 (十进制 80)，表示压缩列表的总长为 80 字节，列表 zltail 属性的值为 0x3c (十进制 60)，假设表的起始地址为 p，计算得出表尾节点 entry3 的地址 p + 60

列表节点

列表节点 entry 的数据结构：

| | | |
|-----------------------|----------|---------|
| previous_entry_length | encoding | content |
|-----------------------|----------|---------|

`previous_entry_length`: 以字节为单位记录了压缩列表中前一个节点的长度, 程序可以通过指针运算, 根据当前节点的起始地址来计算出前一个节点的起始地址, 完成**从表尾向表头遍历**操作

- 如果前一节点的长度小于 254 字节, 该属性的长度为 1 字节, 前一节点的长度就保存在这一个字节里
- 如果前一节点的长度大于等于 254 字节, 该属性的长度为 5 字节, 其中第一字节会被设置为 0xFE (十进制 254), 之后的四个字节则用于保存前一节点的长度

`encoding`: 记录了节点的 `content` 属性所保存的数据类型和长度

- **长度为 1 字节、2 字节或者 5 字节**, 值的最高位为 00、01 或者 10 的是字节数组编码, 数组的长度由编码除去最高两位之后的其他位记录, 下划线 `_` 表示留空, 而 `b`、`x` 等变量则代表实际的二进制数据

| 编 码 | 编码长度 | content 属性保存的值 |
|--|------|----------------------------|
| 00bbbbbb | 1 字节 | 长度小于等于 63 字节的字节数组 |
| 01bbbbbb xxxxxxxx | 2 字节 | 长度小于等于 16 383 字节的字节数组 |
| 10 _____ aaaaaaaaa bbbbbbbb cccccccc dddddddd | 5 字节 | 长度小于等于 4 294 967 295 的字节数组 |

- 长度为 1 字节, 值的最高位为 11 的是整数编码, 整数值的类型和长度由编码除去最高两位之后的其他位记录

| 编码 | 编码长度 | content 属性保存的值 |
|----------|------|--|
| 11000000 | 1 字节 | <code>int16_t</code> 类型的整数 |
| 11010000 | 1 字节 | <code>int32_t</code> 类型的整数 |
| 11100000 | 1 字节 | <code>int64_t</code> 类型的整数 |
| 11110000 | 1 字节 | 24 位有符号整数 |
| 11111110 | 1 字节 | 8 位有符号整数 |
| 1111xxxx | 1 字节 | 使用这一编码的节点没有相应的 <code>content</code> 属性, 因为编码本身的 xxxx 四个位已经保存了一个介于 0 和 12 之间的值, 所以它无须 <code>content</code> 属性 |

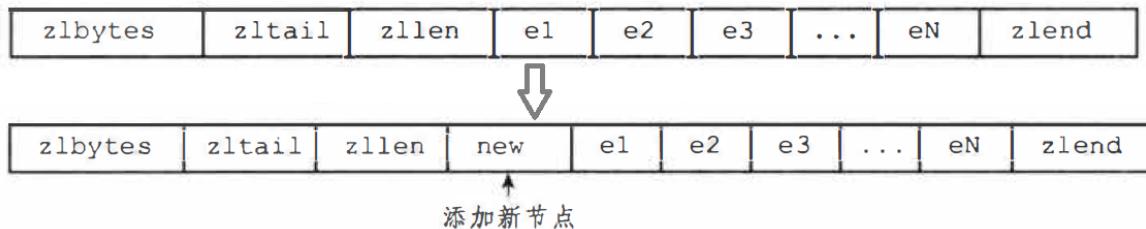
`content`: 每个压缩列表节点可以保存一个字节数组或者一个整数值

- 字节数组可以是以下三种长度的其中一种:
 - 长度小于等于 \$63 (2^6-1)\$ 字节的字节数组
 - 长度小于等于 \$16383(2^{14}-1)\$ 字节的字节数组
 - 长度小于等于 \$4294967295(2^{32}-1)\$ 字节的字节数组
- 整数值则可以是以下六种长度的其中一种:
 - 4 位长, 介于 0 至 12 之间的无符号整数
 - 1 字节长的有符号整数
 - 3 字节长的有符号整数
 - `int16_t` 类型整数
 - `int32_t` 类型整数
 - `int64_t` 类型整数

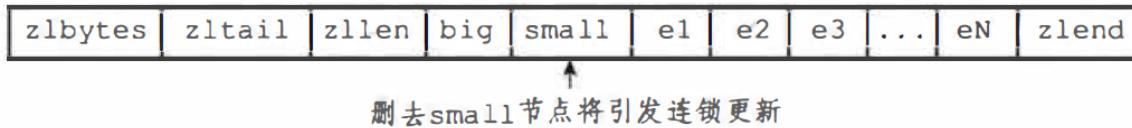
连锁更新

Redis 将在特殊情况下产生的连续多次空间扩展操作称之为连锁更新 (cascade update)

假设在一个压缩列表中，有多个连续的、长度介于 250 到 253 字节之间的节点 e1 至 eN。将一个长度大于等于 254 字节的新节点 new 设置为压缩列表的头节点，new 就成为 e1 的前置节点。e1 的 previous_entry_length 属性仅为 1 字节，无法保存新节点 new 的长度，所以要对压缩列表执行空间重分配操作，并将 e1 节点的 previous_entry_length 属性从 1 字节长扩展为 5 字节长。由于 e1 原本的长度介于 250 至 253 字节之间，所以扩展后 e1 的长度就变成了 254 至 257 字节之间，导致 e2 的 previous_entry_length 属性无法保存 e1 的长度，程序需要不断地对压缩列表执行空间重分配操作，直到 eN 为止



删除节点也可能会引发连锁更新， $\text{big.length} \geq 254$, $\text{small.length} < 254$, 删除 small 节点



连锁更新在最坏情况下需要对压缩列表执行 N 次空间重分配，每次重分配的最坏复杂度为 $O(N)$ ，所以连锁更新的最坏复杂度为 $O(N^2)$

说明：尽管连锁更新的复杂度较高，但出现的记录是非常低的，即使出现只要被更新的节点数量不多，就不会对性能造成影响

数据类型

redisObj

对象系统

Redis 使用对象来表示数据库中的键和值，当在 Redis 数据库中新创建一个键值对时至少会创建两个对象，一个对象用作键值对的键（键对象），另一个对象用作键值对的值（值对象）

Redis 中对象由一个 redisObject 结构表示，该结构中和保存数据有关的三个属性分别是 type、encoding、ptr：

```

typedef struct redisObject {
    // 类型
    unsigned type:4;
    // 编码
    unsigned encoding:4;
    // 指向底层数据结构的指针
    void *ptr;

    // ...
} robj;

```

Redis 并没有直接使用数据结构来实现键值对数据库，而是基于这些数据结构创建了一个对象系统，包含字符串对象、列表对象、哈希对象、集合对象和有序集合对象这五种类型的对象，而每种对象又通过不同的编码映射到不同的底层数据结构

Redis 是一个 Map 类型，其中所有的数据都是采用 key : value 的形式存储，**键对象都是字符串对象**，而值对象有五种基本类型和三种高级类型对象

| 类 型 | 编 码 | 对 象 |
|--------------|---------------------------|------------------------------|
| REDIS_STRING | REDIS_ENCODING_INT | 使用整数值实现的字符串对象 |
| REDIS_STRING | REDIS_ENCODING_EMBSTR | 使用 embstr 编码的简单动态字符串实现的字符串对象 |
| REDIS_STRING | REDIS_ENCODING_RAW | 使用简单动态字符串实现的字符串对象 |
| REDIS_LIST | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的列表对象 |
| REDIS_LIST | REDIS_ENCODING_LINKEDLIST | 使用双端链表实现的列表对象 |
| REDIS_HASH | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的哈希对象 |
| REDIS_HASH | REDIS_ENCODING_HT | 使用字典实现的哈希对象 |
| REDIS_SET | REDIS_ENCODING_INTSET | 使用整数集合实现的集合对象 |
| REDIS_SET | REDIS_ENCODING_HT | 使用字典实现的集合对象 |
| REDIS_ZSET | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的有序集合对象 |
| REDIS_ZSET | REDIS_ENCODING_SKIPLIST | 使用跳跃表和字典实现的有序集合对象 |

- 对一个数据库键执行 TYPE 命令，返回的结果为数据库键对应的值对象的类型，而不是键对象的类型
- 对一个数据库键执行 OBJECT ENCODING 命令，查看数据库键对应的值对象的编码

命令多态

Redis 中用于操作键的命令分为两种类型：

- 一种命令可以对任何类型的键执行，比如说 DEL、EXPIRE、RENAME、TYPE 等（基于类型的多态）
- 只能对特定类型的键执行，比如 SET 只能对字符串键执行、HSET 对哈希键执行、SADD 对集合键执行，如果类型不匹配会报类型错误：(error) WRONGTYPE Operation against a key holding the wrong kind of value

Redis 为了确保只有指定类型的键可以执行某些特定的命令，在执行类型特定的命令之前，先通过值对象 redisObject 结构 type 属性检查操作类型是否正确，然后再决定是否执行指定的命令

对于多态命令，比如列表对象有 ziplist 和 linkedlist 两种实现方式，通过 redisObject 结构 encoding 属性确定具体的编码类型，底层调用对应的 API 实现具体的操作（基于编码的多态）

内存回收

对象的整个生命周期可以划分为创建对象、操作对象、释放对象三个阶段

C 语言没有自动回收内存的功能，所以 Redis 在对象系统中构建了引用计数（reference counting）技术实现的内存回收机制，程序可以跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收

```
typedef struct redisObject {
    // 引用计数
    int refcount;
} robj;
```

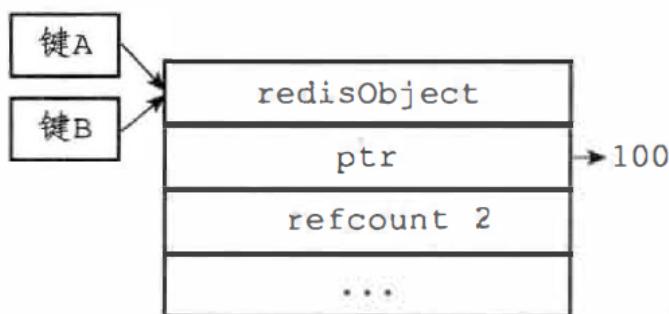
对象的引用计数信息会随着对象的使用状态而不断变化，创建时引用计数 refcount 初始化为 1，每次被一个新程序使用时引用计数加 1，当对象不再被一个程序使用时引用计数值会被减 1，当对象的引用计数值变为 0 时，对象所占用的内存会被释放

对象共享

对象的引用计数属性带有对象共享的作用，共享对象机制更节约内存，数据库中保存的相同值对象越多，节约的内存就越多

让多个键共享一个对象的步骤：

- 将数据库键的值指针指向一个现有的值对象
- 将被共享的值对象的引用计数增一



Redis 在初始化服务器时创建一万个（配置文件可以修改）字符串对象，包含了从 0 到 9999 的所有整数值，当服务器需要用到值为 0 到 9999 的字符串对象时，服务器就会使用这些共享对象，而不是新创建对象

比如创建一个值为 100 的键 A，并使用 OBJECT REFCOUNT 命令查看键 A 的值对象的引用计数，会发现值对象的引用计数为 2，引用这个值对象的两个程序分别是持有这个值对象的服务器程序，以及共享这个值对象的键 A

共享对象在嵌套了字符串对象的对象（linkedlist 编码的列表、hashtable 编码的哈希、zset 编码的有序集合）中也能使用

Redis 不共享包含字符串对象的原因：验证共享对象和目标对象是否相同的复杂度越高，消耗的 CPU 时间也会越多

- 整数值的字符串对象，验证操作的复杂度为 O(1)
- 字符串值的字符串对象，验证操作的复杂度为 O(N)
- 如果共享对象是包含了多个值（或者对象的）对象，比如列表对象或者哈希对象，验证操作的复杂度为 O(N^2)

空转时长

redisObject 结构包含一个 lru 属性，该属性记录了对象最后一次被命令程序访问的时间

```
typedef struct redisObject {
    unsigned lru:22;
} robj;
```

OBJECT IDLETIME 命令可以打印出给定键的空转时长，该值就是通过将当前时间减去键的值对象的 lru 时间计算得出的，这个命令在访问键的值对象时，不会修改值对象的 lru 属性

```
redis> OBJECT IDLETIME msg
(integer) 10
# 等待一分钟
redis> OBJECT IDLETIME msg
(integer) 70
# 访问 msg
redis> GET msg
"hello world"
# 键处于活跃状态，空转时长为 0
redis> OBJECT IDLETIME msg
(integer) 0
```

空转时长的作用：如果服务器开启 maxmemory 选项，并且回收内存的算法为 volatile-lru 或者 allkeys-lru，那么当服务器占用的内存数超过了 maxmemory 所设置的上限值时，空转时长较高的那部分键会优先被服务器释放，从而回收内存（LRU 算法）

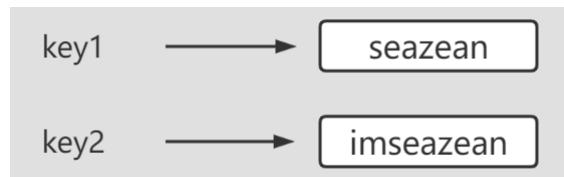
string

简介

存储的数据：单个数据，最简单的数据存储类型，也是最常用的数据存储类型，实质上是存一个字符串，string 类型是二进制安全的，可以包含任何数据，比如图片或者序列化的对象

存储数据的格式：一个存储空间保存一个数据，每一个空间中只能保存一个字符串信息

存储内容：通常使用字符串，如果字符串以整数的形式展示，可以作为数字操作使用



Redis 所有操作都是**原子性的**，采用**单线程**机制，命令是单个顺序执行，无需考虑并发带来的影响，原子性就是有一个失败则都失败

字符串对象可以是 int、raw、embstr 三种实现方式

操作

指令操作：

- 数据操作：

| | |
|----------------------------------|--------------------------------|
| <code>set key value</code> | #添加/修改数据 |
| <code>del key</code> | #删除数据 |
| <code>setnx key value</code> | #判定性添加数据，键值为空则设添加 |
| <code>mset k1 v1 k2 v2...</code> | #添加/修改多个数据，m: Multiple |
| <code>append key value</code> | #追加信息到原始信息后部（如果原始信息存在就追加，否则新建） |

- 查询操作

| | |
|--------------------------------|----------------------|
| <code>get key</code> | #获取数据，如果不存在，返回空（nil） |
| <code>mget key1 key2...</code> | #获取多个数据 |
| <code>strlen key</code> | #获取数据字符个数（字符串长度） |

- 设置数值数据增加/减少指定范围的值

| | |
|--|----------------|
| <code>incr key</code> | #key++ |
| <code>incrby key increment</code> | #key+increment |
| <code>incrbyfloat key increment</code> | #对小数操作 |
| <code>decr key</code> | #key-- |
| <code>decrby key increment</code> | #key-increment |

- 设置数据具有指定的生命周期

| | |
|--|------------------------------|
| <code>setex key seconds value</code> | #设置key-value存活时间，seconds单位是秒 |
| <code>psetex key milliseconds value</code> | #毫秒级 |

注意事项：

1. 数据操作不成功的反馈与数据正常操作之间的差异

- 表示运行结果是否成功

- (integer) 0 → false，失败
- (integer) 1 → true，成功

- 表示运行结果值
 - (integer) 3 → 3 个
 - (integer) 1 → 1 个
- 2. 数据未获取到时，对应的数据为 (nil)，等同于 null
- 3. **数据最大存储量：512MB**
- 4. string 在 Redis 内部存储默认就是一个字符串，当遇到增减类操作 incr, decr 时会转成数值型进行计算
- 5. 按数值进行操作的数据，如果原始数据不能转成数值，或超越了 Redis 数值上限范围，将报错 9223372036854775807 (java 中 Long 型数据最大值，Long.MAX_VALUE)
- 6. Redis 可用于控制数据库表主键 ID，为数据库表主键提供生成策略，保障数据库表的主键唯一性

单数据和多数据的选择：

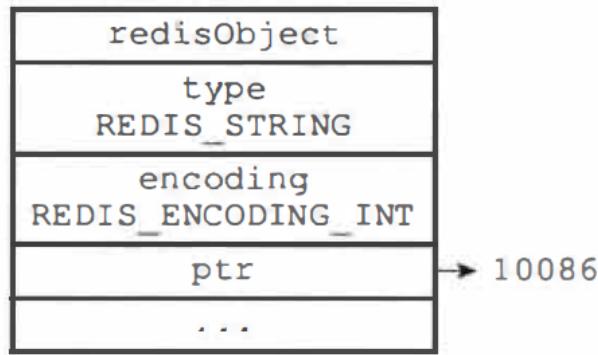
- 单数据执行 3 条指令的过程：3 次发送 + 3 次处理 + 3 次返回
- 多数据执行 1 条指令的过程：1 次发送 + 3 次处理 + 1 次返回（发送和返回的事件略高于单数据）



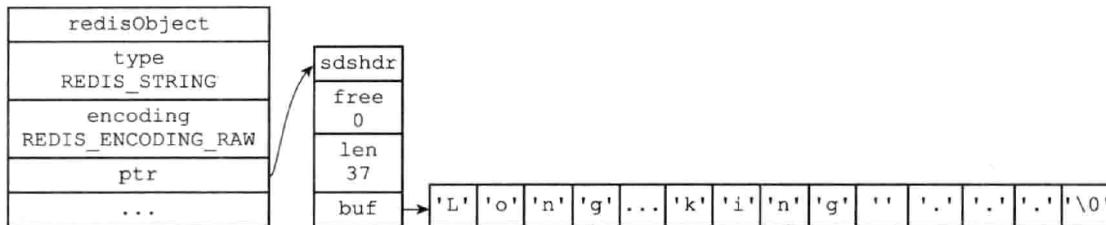
实现

字符串对象的编码可以是 int、raw、embstr 三种

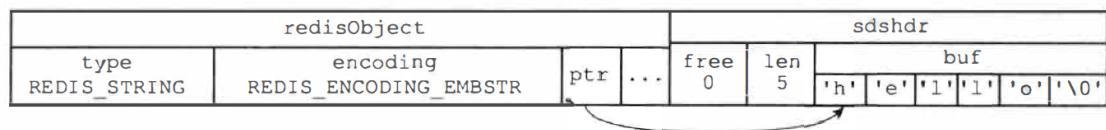
- int：字符串对象保存的是**整数值**，并且整数值可以用 long 类型来表示，那么对象会将整数值保存在字符串对象结构的 ptr 属性面（将 void * 转换成 long），并将字符串对象的编码设置为 int（浮点数用另外两种方式）



- raw: 字符串对象保存的是一个字符串值，并且值的长度大于 39 字节，那么对象将使用简单动态字符串 (SDS) 来保存该值，并将对象的编码设置为 raw



- embstr: 字符串对象保存的是一个字符串值，并且值的长度小于等于 39 字节，那么对象将使用 embstr 编码的方式来保存这个字符串值，并将对象的编码设置为 embstr



上图所示，embstr 与 raw 都使用了 redisObject 和 sdshdr 来表示字符串对象，但是 raw 需要调用两次内存分配函数分别创建两种结构，embstr 只需要一次内存分配来分配一块**连续的空间**

embstr 是用于保存短字符串的一种编码方式，对比 raw 的优点：

- 内存分配次数从两次降低为一次，同样释放内存的次数也从两次变为一次
- embstr 编码的字符串对象的数据都保存在同一块连续内存，所以比 raw 编码能够更好地利用缓存优势（局部性原理）

int 和 embstr 编码的字符串对象在条件满足的情况下，会被转换为 raw 编码的字符串对象：

- int 编码的整数值，执行 APPEND 命令追加一个字符串值，先将整数值转为字符串然后追加，最后得到一个 raw 编码的对象
- Redis 没有为 embstr 编码的字符串对象编写任何相应的修改程序，所以 embstr 对象实际上是**只读的**，执行修改命令会将对象的编码从 embstr 转换成 raw，操作完成后得到一个 raw 编码的对象

某些情况下，程序会将字符串对象里面的字符串值转换回浮点数值，执行某些操作后再将浮点数值转换回字符串值：

```

redis> SET pi 3.14
OK
redis> OBJECT ENCODING pi
"embstr"
redis> INCRBYFLOAT pi 2.0 # 转为浮点数执行增加的操作
"5.14"
redis> OBJECT ENCODING pi
"embstr"

```

应用

主页高频访问信息显示控制，例如新浪微博大 V 主页显示粉丝数与微博数量

- 在 Redis 中为大 V 用户设定用户信息，以用户主键和属性值作为 key，后台设定定时刷新策略

```
set user:id:3506728370:fans 12210947
set user:id:3506728370:blogs 6164
set user:id:3506728370:focuses 83
```

- 使用 JSON 格式保存数据

```
user:id:3506728370 → {"fans":12210947,"blogs":6164,"focuses":83}
```

- key的设置约定：表名 : 主键名 : 主键值 : 字段名

| 表名 | 主键名 | 主键值 | 字段名 |
|-------|-----|-----------|-------|
| order | id | 29437595 | name |
| equip | id | 390472345 | type |
| news | id | 202004150 | title |

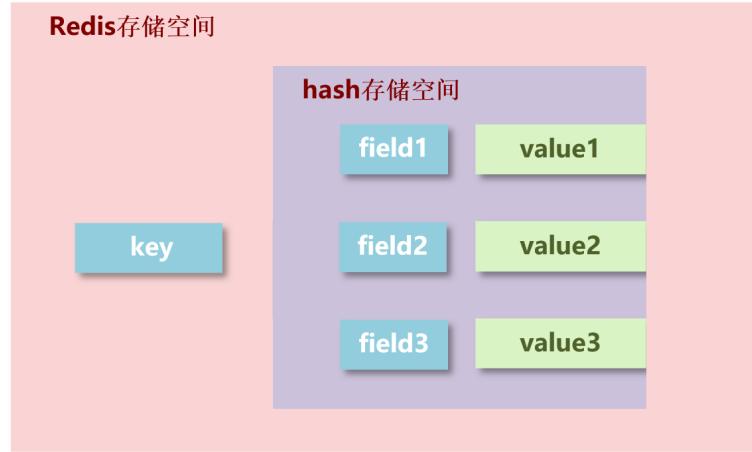
hash

简介

数据存储需求：对一系列存储的数据进行编组，方便管理，典型应用存储对象信息

数据存储结构：一个存储空间保存多个键值对数据

hash 类型：底层使用哈希表结构实现数据存储



Redis 中的 hash 类似于 Java 中的 `Map<String, Map<Object, Object>>`, 左边是 key, 右边是值, 中间叫 field 字段, 本质上 hash 存了一个 key-value 的存储空间

hash 是指的一个数据类型, 并不是一个数据

- 如果 field 数量较少, 存储结构优化为**压缩列表结构** (有序)
- 如果 field 数量较多, 存储结构使用 HashMap 结构 (无序)

操作

指令操作:

- 数据操作

| | |
|---------------------------------------|-------------------------------|
| <code>hset key field value</code> | #添加/修改数据 |
| <code>hdel key field1 [field2]</code> | #删除数据, []代表可选 |
| <code>hsetnx key field value</code> | #设置field的值, 如果该field存在则不做任何操作 |
| <code>hmset key f1 v1 f2 v2...</code> | #添加/修改多个数据 |

- 查询操作

| | |
|---|------------------|
| <code>hget key field</code> | #获取指定field对应数据 |
| <code>hgetall key</code> | #获取指定key所有数据 |
| <code>hmget key field1 field2...</code> | #获取多个数据 |
| <code>hexists key field</code> | #获取哈希表中是否存在指定的字段 |
| <code>hlen key</code> | #获取哈希表中字段的数量 |

- 获取哈希表中所有的字段名或字段值

| | |
|------------------------|-------------|
| <code>hkeys key</code> | #获取所有的field |
| <code>hvals key</code> | #获取所有的value |

- 设置指定字段的数值数据增加指定范围的值

```

hincrby key field increment      #指定字段的数值数据增加指定的值, increment为负数则减少
hincrbyfloat key field increment#操作小数

```

注意事项

1. hash 类型中 value 只能存储字符串，不允许存储其他数据类型，不存在嵌套现象，如果数据未获取到，对应的值为 (nil)
2. 每个 hash 可以存储 $2^{32} - 1$ 个键值对
3. hash 类型和对象的数据存储形式相似，并且可以灵活添加删除对象属性。但 hash 设计初衷不是为了存储大量对象而设计的，不可滥用，不可将 hash 作为对象列表使用
4. hgetall 操作可以获取全部属性，如果内部 field 过多，遍历整体数据效率就很会低，有可能成为数据访问瓶颈

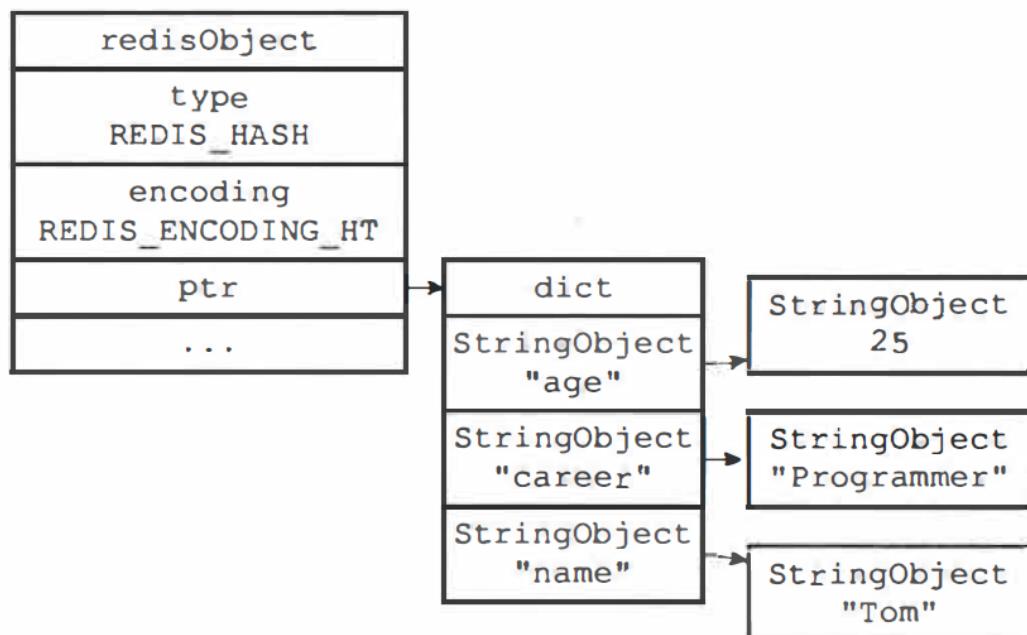
实现

哈希对象的内部编码有两种：ziplist（压缩列表）、hashtable（哈希表、字典）

- 压缩列表实现哈希对象：同一键值对的节点总是挨在一起，保存键的节点在前，保存值的节点在后



- 字典实现哈希对象：字典的每一个键都是一个字符串对象，每个值也是



当存储的数据量比较小的情况下，Redis 才使用压缩列表来实现字典类型，具体需要满足两个条件：

- 当键值对数量小于 hash-max-ziplist-entries 配置（默认 512 个）
- 所有键和值的长度都小于 hash-max-ziplist-value 配置（默认 64 字节）

以上两个条件的上限值是可以通过配置文件修改的，当两个条件的任意一个不能被满足时，对象的编码转换操作就会被执行

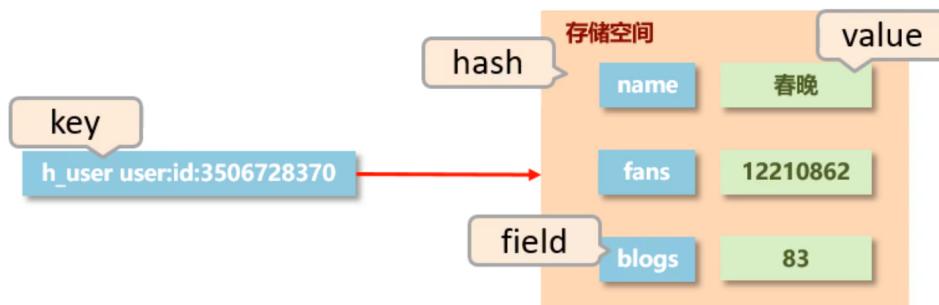
ziplist 使用更加紧凑的结构实现多个元素的连续存储，所以在节省内存方面比 hashtable 更加优秀，当 ziplist 无法满足哈希类型时，Redis 会使用 hashtable 作为哈希的内部实现，因为此时 ziplist 的读写效率会下降，而 hashtable 的读写时间复杂度为 O(1)

应用

```
user:id:3506728370 → {"name": "春晚", "fans": 12210862, "blogs": 83}
```

对于以上数据，使用单条去存的话，存的条数会很多。但如果用 json 格式，存一条数据就够了。

假如现在粉丝数量发生了变化，要把整个值都改变，但是用单条存就不存在这个问题，只需要改其中一个就可以



可以实现购物车的功能，key 对应着每个用户，存储空间存储购物车的信息

list

简介

数据存储需求：存储多个数据，并对数据进入存储空间的顺序进行区分

数据存储结构：一个存储空间保存多个数据，且通过数据可以体现进入顺序，允许重复元素

list 类型：保存多个数据，底层使用**双向链表**存储结构实现，类似于 LinkedList



如果两端都能存取数据的话，这就是双端队列，如果只能从一端进一端出，这个模型叫栈

操作

指令操作：

- 数据操作

```
lpush key value1 [value2]... #从左边添加/修改数据(表头)
rpush key value1 [value2]... #从右边添加/修改数据(表尾)
lpop key #从左边获取并移除第一个数据，类似于出栈/出队
rpop key #从右边获取并移除第一个数据
lrem key count value #删除指定数据，count=2删除2个，该value可能有多个(重复数据)
```

- 查询操作

```
lrange key start stop #从左边遍历数据并指定开始和结束索引，0是第一个索引，-1是终索引
lindex key index #获取指定索引数据，没有则为nil，没有索引越界
llen key #list中数据长度/个数
```

- 规定时间内获取并移除数据

```
b #代表阻塞
blpop key1 [key2] timeout #在指定时间内获取指定key(可以多个)的数据，超时则为(nil)
#可以从其他客户端写数据，当前客户端阻塞读取数据
brpop key1 [key2] timeout #从右边操作
```

- 复制操作

```
brpoplpush source destination timeout #从source获取数据放入destination，假如在指定时间内没有任何元素被弹出，则返回一个nil和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长
```

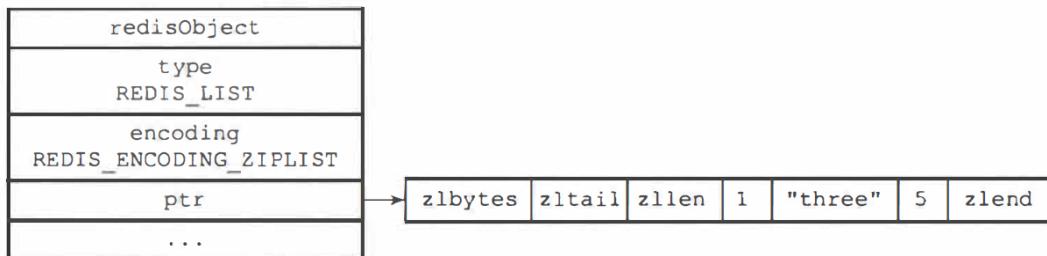
注意事项

- list 中保存的数据都是 string 类型的，数据总容量是有限的，最多 $2^{32} - 1$ 个元素
(4294967295)
 - list 具有索引的概念，但操作数据时通常以队列的形式进行入队出队，或以栈的形式进行入栈出栈
 - 获取全部数据操作结束索引设置为 -1
 - list 可以对数据进行分页操作，通常第一页的信息来自于 list，第 2 页及更多的信息通过数据库的形式加载
-

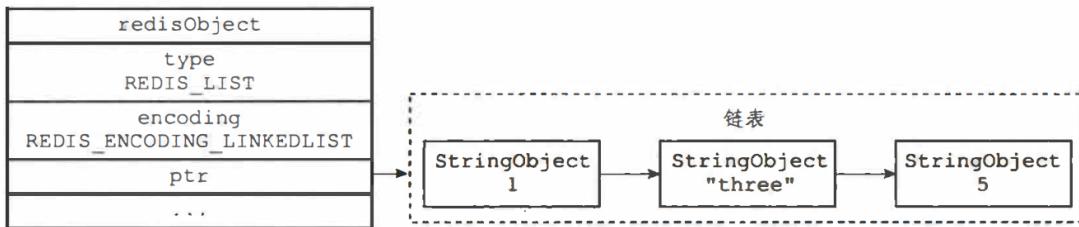
实现

在 Redis3.2 版本以前列表对象的内部编码有两种：ziplist（压缩列表）和 linkedlist（链表）

- 压缩列表实现的列表对象：PUSH 1、three、5 三个元素



- 链表实现的列表对象：为了简化字符串对象的表示，使用了 StringObject 的结构，底层其实是 sdshdr 结构

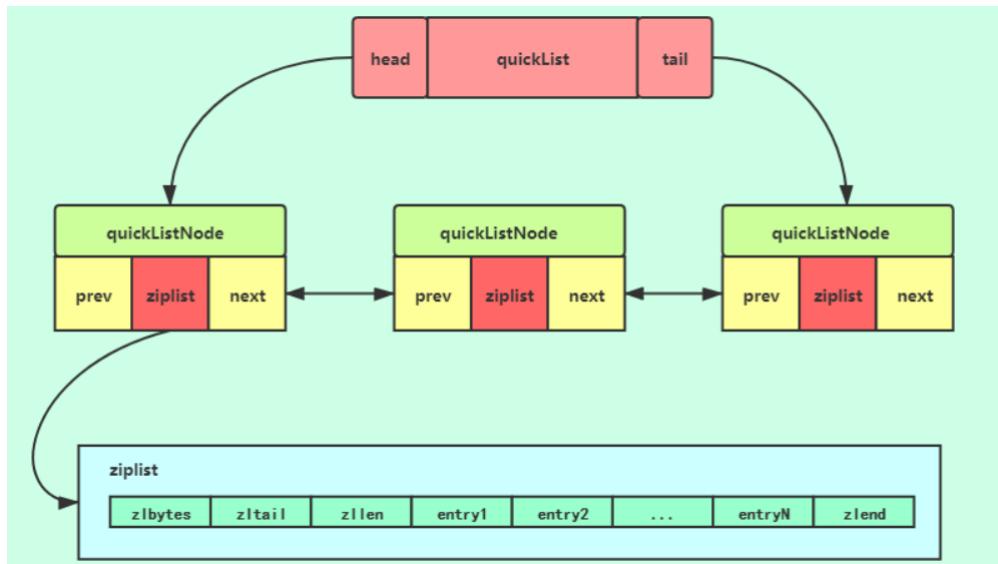


列表中存储的数据量比较小的时候，列表就会使用一块连续的内存存储，采用压缩列表的方式实现的条件：

- 列表对象保存的所有字符串元素的长度都小于 64 字节
- 列表对象保存的元素数量小于 512 个

以上两个条件的上限值是可以通过配置文件修改的，当两个条件的任意一个不能被满足时，对象的编码转换操作就会被执行

在 Redis3.2 版本以后对列表数据结构进行了改造，使用 **quicklist (快速列表)** 代替了 linkedlist，quicklist 实际上是 ziplist 和 linkedlist 的混合体，将 linkedlist 按段切分，每一段使用 ziplist 来紧凑存储，多个 ziplist 之间使用双向指针串接起来，既满足了快速的插入删除性能，又不会出现太大的空间冗余



应用

企业运营过程中，系统将产生出大量的运营数据，如何保障多台服务器操作日志的统一顺序输出？

- 依赖 list 的数据具有顺序的特征对信息进行管理，右进左查或者左进左查
- 使用队列模型解决多路信息汇总合并的问题
- 使用栈模型解决最新消息的问题

微信文章订阅公众号：

- 比如订阅了两个公众号，它们发布了两篇文章，文章 ID 分别为 666 和 888，可以通过执行 `Lpush key 666 888` 命令推送给我的

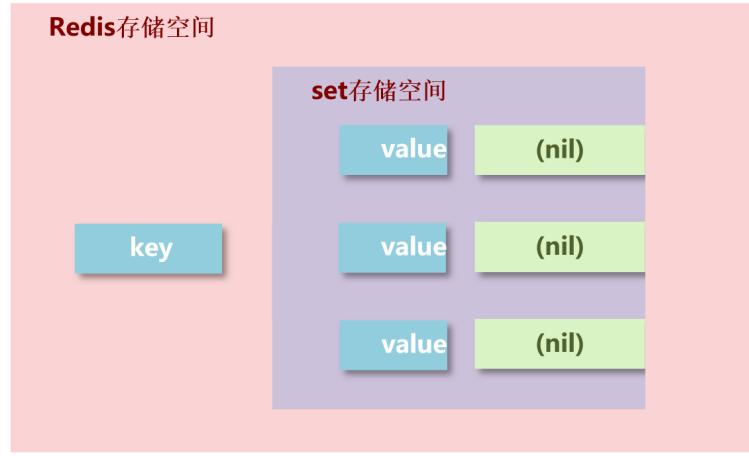
set

简介

数据存储需求：存储大量的数据，在查询方面提供更高的效率

数据存储结构：能够保存大量的数据，高效的内部存储机制，便于查询

set 类型：与 hash 存储结构哈希表完全相同，只是仅存储键不存储值（nil），所以添加，删除，查找的复杂度都是 O(1)，并且值是不允许重复且无序的



操作

指令操作:

- 数据操作

```
sadd key member1 [member2] #添加数据
srem key member1 [member2] #删除数据
```

- 查询操作

```
smembers key #获取全部数据
scard key #获取集合数据总量
sismember key member #判断集合中是否包含指定数据
```

- 随机操作

```
spop key [count] #随机获取集中的某个数据并将该数据移除集合
 srandmember key [count] #随机获取集合中指定(数量)的数据
```

- 集合的交、并、差

```
sinter key1 [key2...] #两个集合的交集, 不存在为(empty list or
set)
sunion key1 [key2...] #两个集合的并集
sdiff key1 [key2...] #两个集合的差集

sinterstore destination key1 [key2...] #两个集合的交集并存储到指定集合中
sunionstore destination key1 [key2...] #两个集合的并集并存储到指定集合中
sdiffstore destination key1 [key2...] #两个集合的差集并存储到指定集合中
```

- 复制

```
smove source destination member #将指定数据从原始集合中移动到目标集合中
```

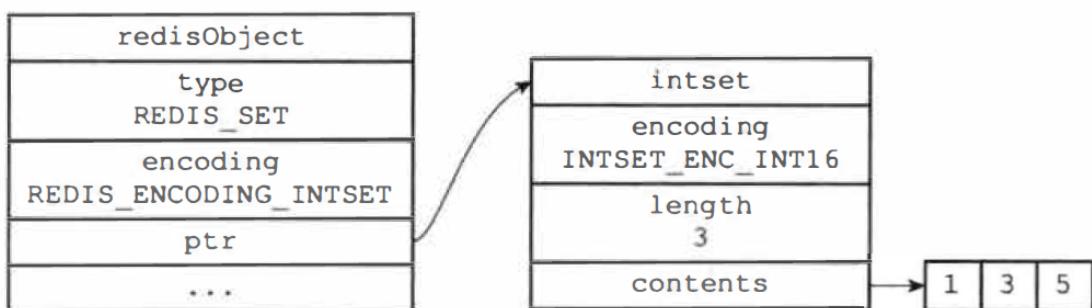
注意事项

1. set 类型不允许数据重复，如果添加的数据在 set 中已经存在，将只保留一份
 2. set 虽然与 hash 的存储结构相同，但是无法启用 hash 中存储值的空间
-

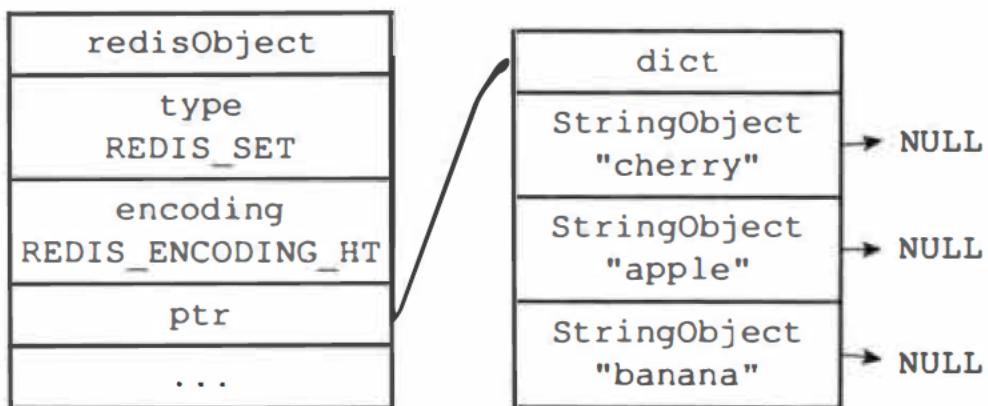
实现

集合对象的内部编码有两种：intset（整数集合）、hashtable（哈希表、字典）

- 整数集合实现的集合对象：



- 字典实现的集合对象：键值对的值为 NULL



当集合对象可以同时满足以下两个条件时，对象使用 intset 编码：

- 集合中的元素都是整数值
- 集合中的元素数量小于 set-maxintset-entries 配置（默认 512 个）

以上两个条件的上限值是可以通过配置文件修改的

应用

应用场景：

1. 黑名单：资讯类信息类网站追求高访问量，但是由于其信息的价值，往往容易被不法分子利用，通过爬虫技术，快速获取信息，个别特种行业网站信息通过爬虫获取分析后，可以转换成商业机密。

注意：爬虫不一定做摧毁性的工作，有些小型网站需要爬虫为其带来一些流量。

2. 白名单：对于安全性更高的应用访问，仅仅靠黑名单是不能解决安全问题的，此时需要设定可访问的用户群体，依赖白名单做更为苛刻的访问验证
 3. 随机操作可以实现抽奖功能
 4. 集合的交并补可以实现微博共同关注的查看，可以根据共同关注或者共同喜欢推荐相关内容
-

zset

简介

数据存储需求：数据排序有利于数据的有效展示，需要提供一种可以根据自身特征进行排序的方式

数据存储结构：新的存储模型，可以保存可排序的数据

操作

指令操作：

- 数据操作

```
zadd key score1 member1 [score2 member2]      #添加数据
zrem key member [member ...]                  #删除数据
zremrangebyrank key start stop               #删除指定索引范围的数据
zremrangebyscore key min max                #删除指定分数区间内的数据
zscore key member                            #获取指定值的分数
zincrby key increment member               #指定值的分数增加increment
```

- 查询操作

```
zrange key start stop [WITHSCORES]          #获取指定范围的数据，升序，WITHSCORES 代
表显示分数
zrevrange key start stop [WITHSCORES]       #获取指定范围的数据，降序

zrangebyscore key min max [WITHSCORES] [LIMIT offset count] #按条件获取数据，从
小到大
zrevrangebyscore key max min [WITHSCORES] [...]           #从大到小

zcard key                                     #获取集合数据的总量
zcount key min max                          #获取指定分数区间内的数据总量
zrank key member                           #获取数据对应的索引（排名）升序
zrevrank key member                         #获取数据对应的索引（排名）降序
```

- min 与 max 用于限定搜索查询的条件
- start 与 stop 用于限定查询范围，作用于索引，表示开始和结束索引

- offset 与 count 用于限定查询范围，作用于查询结果，表示开始位置和数据总量
- 集合的交、并操作

```
zinterstore destination numkeys key [key ...]      #两个集合的交集并存储到指定集合中
zunionstore destination numkeys key [key ...]      #两个集合的并集并存储到指定集合中
```

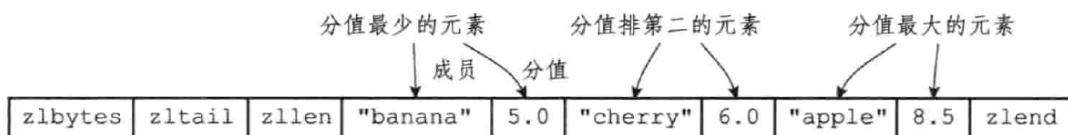
注意事项：

1. score 保存的数据存储空间是 64 位，如果是整数范围是 -9007199254740992~9007199254740992
2. score 保存的数据也可以是一个双精度的 double 值，基于双精度浮点数的特征可能会丢失精度，慎重使用
3. sorted_set 底层存储还是基于 set 结构的，因此数据不能重复，如果重复添加相同的数据，score 值将被反复覆盖，保留最后一次修改的结果

实现

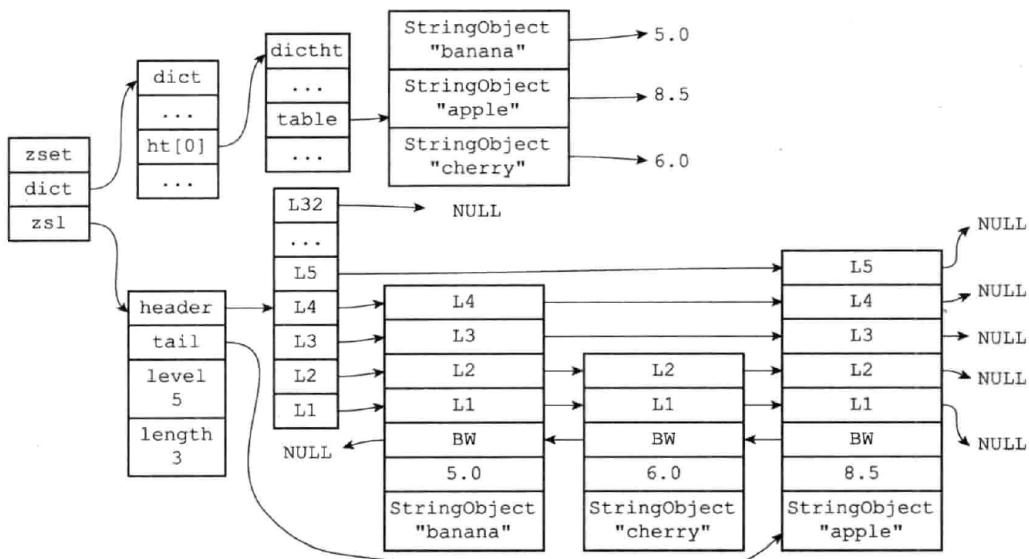
有序集合对象的内部编码有两种：ziplist（压缩列表）和 skipiplist（跳跃表）

- 压缩列表实现有序集合对象：ziplist 本身是有序、不可重复的，符合有序集合的特性



- 跳跃表实现有序集合对象：底层是 zset 结构，zset 同时包含字典和跳跃表的结构，图示字典和跳跃表中重复展示了各个元素的成员和分值，但实际上两者会通过指针来共享相同元素的成员和分值，不会产生空间浪费

```
typedef struct zset {
    zskipiplist *zsl;
    dict *dict;
} zset;
```



使用字典加跳跃表的优势：

- 字典为有序集合创建了一个**从成员到分值的映射**，用 $O(1)$ 复杂度查找给定成员的分值
- **排序操作使用跳跃表完成**，节省每次重新排序带来的时间成本和空间成本

使用 ziplist 格式存储需要满足以下两个条件：

- 有序集合保存的元素个数要小于 128 个；
- 有序集合保存的所有元素大小都小于 64 字节

当元素比较多时，此时 ziplist 的读写效率会下降，时间复杂度是 $O(n)$ ，跳表的时间复杂度是 $O(\log n)$

为什么用跳表而不用平衡树？

- 在做范围查找的时候，跳表操作简单（前进指针或后退指针），平衡树需要回旋查找
- 跳表比平衡树实现简单，平衡树的插入和删除操作可能引发子树的旋转调整，而跳表的插入和删除只需要修改相邻节点的指针

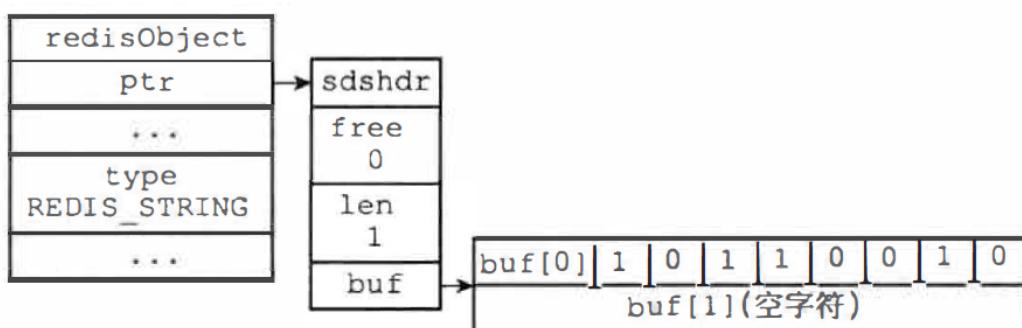
应用

- 排行榜
- 对于基于时间线限定的任务处理，将处理时间记录为 score 值，利用排序功能区分处理的先后顺序
- 当任务或者消息待处理，形成了任务队列或消息队列时，对于高优先级的任务要保障对其优先处理，采用 score 记录权重

Bitmaps

基本操作

Bitmaps 是二进制位数组 (bit array)，底层使用 SDS 字符串表示，因为 SDS 是二进制安全的



buf 数组的每个字节用一行表示，buf[1] 是 '`\0`'，保存位数组的顺序和书写位数组的顺序是完全相反的，图示的位数组 0100 1101

数据结构的详解查看 Java → Algorithm → 位图

命令实现

GETBIT

GETBIT 命令获取位数组 bitarray 在 offset 偏移量上的二进制位的值

```
GETBIT <bitarray> <offset>
```

执行过程：

- 计算 `byte = offset/8` (向下取整) , byte 值记录数据保存在位数组中的索引
- 计算 `bit = (offset mod 8) + 1`, bit 值记录数据在位数组中的第几个二进制位
- 根据 byte 和 bit 值, 在位数组 bitarray 中定位 offset 偏移量指定的二进制位, 并返回这个位的值

GETBIT 命令执行的所有操作都可以在常数时间内完成, 所以时间复杂度为 O(1)

SETBIT

SETBIT 将位数组 bitarray 在 offset 偏移量上的二进制位的值设置为 value, 并向客户端返回二进制位的旧值

```
SETBIT <bitarray> <offset> <value>
```

执行过程：

- 计算 `len = offset/8 + 1`, len 值记录了保存该数据至少需要多少个字节
 - 检查 bitarray 键保存的位数组的长度是否小于 len, 成立就会将 SDS 扩展为 len 字节 (注意空间预分配机制) , 所有新扩展空间的二进制位的值置为 0
 - 计算 `byte = offset/8` (向下取整) , byte 值记录数据保存在位数组中的索引
 - 计算 `bit = (offset mod 8) + 1`, bit 值记录数据在位数组中的第几个二进制位
 - 根据 byte 和 bit 值, 在位数组 bitarray 中定位 offset 偏移量指定的二进制位, 首先将指定位现存的值保存在 oldvalue 变量, 然后将新值 value 设置为这个二进制位的值
 - 向客户端返回 oldvalue 变量的值
-

BITCOUNT

BITCOUNT 命令用于统计给定位数组中，值为 1 的二进制位的数量

```
BITCOUNT <bitarray> [start end]
```

二进制位统计算法：

- 遍历法：遍历位数组中的每个二进制位
- 查表算法：读取每个字节（8 位）的数据，查表获取数值对应的二进制中有几个 1
- variable-precision SWAR 算法：计算汉明距离
- Redis 实现：
 - 如果二进制位的数量大于等于 128 位，那么使用 variable-precision SWAR 算法来计算二进制位的汉明重量
 - 如果二进制位的数量小于 128 位，那么使用查表算法来计算二进制位的汉明重量

BITOP

BITOP 命令对指定 key 按位进行交、并、非、异或操作，并将结果保存到指定的键中

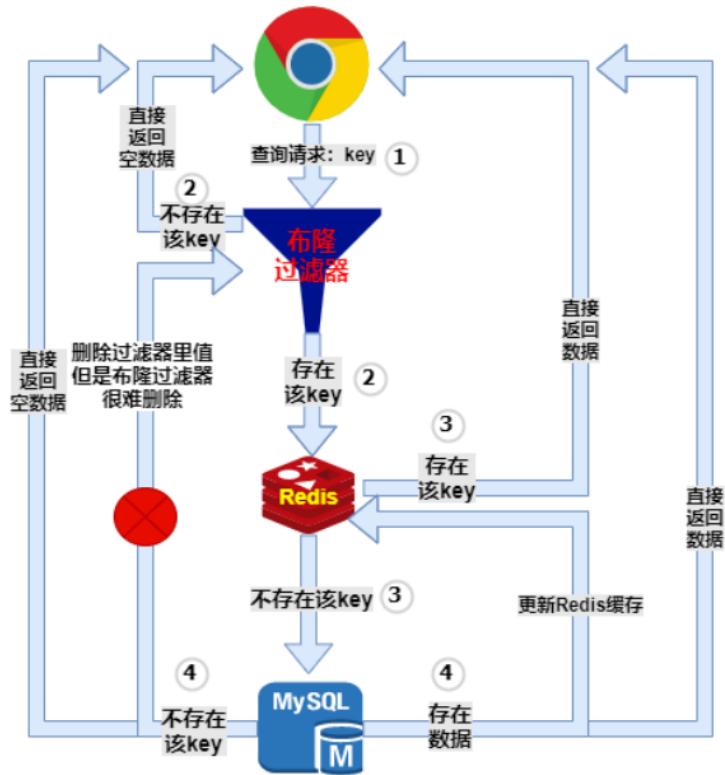
```
BITOP OPTION destKey key1 [key2...]
```

OPTION 有 AND（与）、OR（或）、XOR（异或）和 NOT（非）四个选项

AND、OR、XOR 三个命令可以接受多个位数组作为输入，需要遍历输入的每个位数组的每个字节来进行计算，所以命令的复杂度为 $O(n^2)$ ；与此相反，NOT 命令只接受一个位数组输入，所以时间复杂度为 $O(n)$

应用场景

- 解决 Redis 缓存穿透，判断给定数据是否存在，防止缓存穿透



白名单

存在的问题:

布隆过滤器里的数据，存在误判，如果没在白名单里的数据被误判存在于过滤器里的话，会穿透到数据库，不过误判的几率本来就很小，所以穿透问题不大

注意的问题:

必须将所有的key都放到布隆过滤器和Redis里，否则请求会被直接返回空数据

- 垃圾邮件过滤，对每一个发送邮件的地址进行判断是否在布隆的黑名单中，如果在就判断为垃圾邮件
- 爬虫去重，爬给定网址的时候对已经爬取过的 URL 去重
- 信息状态统计

Hyper

基数是数据集去重后元素个数，HyperLogLog 是用来做基数统计的，运用了 LogLog 的算法

| | | |
|-----------------------|----------------------|-------|
| {1, 3, 5, 7, 5, 7, 8} | 基数集: {1, 3, 5, 7, 8} | 基数: 5 |
| {1, 1, 1, 1, 1, 7, 1} | 基数集: {1, 7} | 基数: 2 |

相关指令:

- 添加数据

```
pfadd key element [element ...]
```

- 统计数据

```
pfcount key [key ...]
```

- 合并数据

```
pfmerge destkey sourcekey [sourcekey...]
```

应用场景：

- 用于进行基数统计，不是集合不保存数据，只记录数量而不是具体数据，比如网站的访问量
 - 核心是基数估算算法，最终数值存在一定误差
 - 误差范围：基数估计的结果是一个带有 0.81% 标准错误的近似值
 - 耗空间极小，每个 hyperloglog key 占用了12K的内存用于标记基数
 - pfadd 命令不是一次性分配12K内存使用，会随着基数的增加内存逐渐增大
 - Pfmerge 命令合并后占用的存储空间为12K，无论合并之前数据量多少
-

GEO

GeoHash 是一种地址编码方法，把二维的空间经纬度数据编码成一个字符串

- 添加坐标点

```
geoadd key longitude latitude member [longitude latitude member ...]  
georadius key longitude latitude radius m|km|ft|mi [withcoord] [withdist]  
[withhash] [count count]
```

- 获取坐标点

```
geopos key member [member ...]  
georadiusbymember key member radius m|km|ft|mi [withcoord] [withdist]  
[withhash] [count count]
```

- 计算距离

```
geodist key member1 member2 [unit] #计算坐标点距离  
geohash key member [member ...] #计算经纬度
```

Redis 应用于地理位置计算

持久机制

概述

持久化：利用永久性存储介质将数据进行保存，在特定的时间将保存的数据进行恢复的工作机制称为持久化

作用：持久化用于防止数据的意外丢失，确保数据安全性，因为 Redis 是内存级，所以需要持久化到磁盘

计算机中的数据全部都是二进制，保存一组数据有两种方式

```
10011001110000001  
00101001011010110  
10110011001110000  
00100101001011011
```

```
删除第3行  
第4行末位添加字符x  
删除第2到第4行  
复制第3行粘贴到第5行
```

数据(快照)

RDB

过程(日志)

AOF

RDB：将当前数据状态进行保存，快照形式，存储数据结果，存储格式简单

AOF：将数据的操作过程进行保存，日志形式，存储操作过程，存储格式复杂

RDB

文件创建

RDB 持久化功能所生成的 RDB 文件是一个经过压缩的紧凑二进制文件，通过该文件可以还原生成 RDB 文件时的数据库状态，有两个 Redis 命令可以生成 RDB 文件，一个是 SAVE，另一个是 BGSAVE

SAVE

SAVE 指令：手动执行一次保存操作，该指令的执行会阻塞当前 Redis 服务器，客户端发送的所有命令请求都会被拒绝，直到当前 RDB 过程完成为止，有可能会造成长时间阻塞，线上环境不建议使用

工作原理：Redis 是个**单线程的工作模式**，会创建一个任务队列，所有的命令都会进到这个队列排队执行。当某个指令在执行的时候，队列后面的指令都要等待，所以这种执行方式会非常耗时

配置 redis.conf：

```

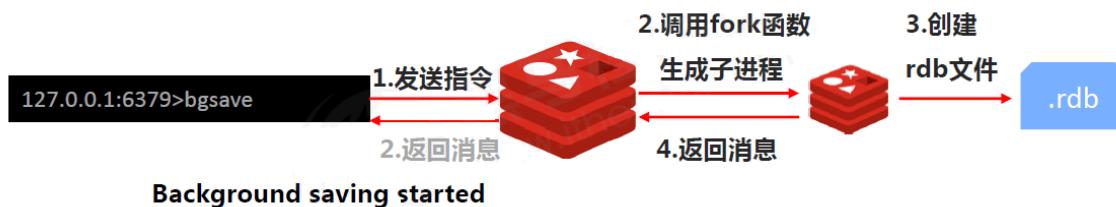
dir path          #设置存储.rdb文件的路径，通常设置成存储空间较大的目录中，目录名称
data
dbfilename "x.rdb"      #设置本地数据库文件名，默认值为dump.rdb，通常设置为dump-端口号.rdb
rdbcompression yes|no    #设置存储至本地数据库时是否压缩数据，默认yes，设置为no节省CPU运行时间
rdbchecksum yes|no       #设置读写文件过程是否进行RDB格式校验，默认yes

```

BGSAVE

BGSAVE: bg 是 background，代表后台执行，命令的完成需要两个进程，**进程之间不相互影响**，所以持久化期间 Redis 正常工作

工作原理：



流程：客户端发出 BGSAVE 指令，Redis 服务器使用 fork 函数创建一个子进程，然后响应后台已经开始执行的信息给客户端。子进程会异步执行持久化的操作，持久化过程是先将数据写入到一个临时文件中，持久化操作结束再用这个临时文件替换上次持久化的文件

```

# 创建子进程
pid = fork()
if pid == 0:
    # 子进程负责创建 RDB 文件
    rdbSave()
    # 完成之后向父进程发送信号
    signal_parent()
elif pid > 0:
    # 父进程继续处理命令请求，并通过轮询等待子进程的信号
    handle_request_and_wait_signal()
else:
    # 处理出错情况
    handle_fork_error()

```

配置 redis.conf

```

stop-writes-on-bgsave-error yes|no  #后台存储过程中如果出现错误，是否停止保存操作，默认yes
dbfilename filename
dir path
rdbcompression yes|no
rdbchecksum yes|no

```

注意：BGSAVE 命令是针对 SAVE 阻塞问题做的优化，Redis 内部所有涉及到 RDB 操作都采用 BGSAVE 的方式，SAVE 命令放弃使用

在 BGSAVE 命令执行期间，服务器处理 SAVE、BGSAVE、BGREWRITEAOF 三个命令的方式会和平时有所不同

- SAVE 命令会被服务器拒绝，服务器禁止 SAVE 和 BGSAVE 命令同时执行是为了避免父进程（服务器进程）和子进程同时执行两个 `rbSave` 调用，产生竞争条件
- BGSAVE 命令也会被服务器拒绝，也会产生竞争条件
- BGREWRITEAOF 和 BGSAVE 两个命令不能同时执行
 - 如果 BGSAVE 命令正在执行，那么 BGREWRITEAOF 命令会被延迟到 BGSAVE 命令执行完毕之后执行
 - 如果 BGREWRITEAOF 命令正在执行，那么 BGSAVE 命令会被服务器拒绝

特殊指令

RDB 特殊启动形式的指令（客户端输入）

- 服务器运行过程中重启

```
debug reload
```

- 关闭服务器时指定保存数据

```
shutdown save
```

默认情况下执行 `shutdown` 命令时，自动执行 `bgsave`（如果没有开启 AOF 持久化功能）

- 全量复制：主从复制部分详解

文件载入

RDB 文件的载入工作是在服务器启动时自动执行，期间 Redis 会一直处于阻塞状态，直到载入完成

Redis 并没有专门用于载入 RDB 文件的命令，只要服务器在启动时检测到 RDB 文件存在，就会自动载入 RDB 文件

```
[7379] 30 Aug 21:07:01.289 * DB loaded from disk: 0.018 seconds # 服务器在成功载入  
RDB 文件之后打印
```

AOF 文件的更新频率通常比 RDB 文件的更新频率高：

- 如果服务器开启了 AOF 持久化功能，那么会优先使用 AOF 文件来还原数据库状态
- 只有在 AOF 持久化功能处于关闭状态时，服务器才会使用 RDB 文件来还原数据库状态

自动保存

配置文件

Redis 支持通过配置服务器的 save 选项，让服务器每隔一段时间自动执行一次 BGSAVE 命令

配置 redis.conf:

```
save second changes #设置自动持久化条件，满足限定时间范围内key的变化数量就进行持久化  
(bgsave)
```

- second: 监控时间范围
- changes: 监控 key 的变化量

默认三个条件:

```
save 900 1      # 900s内1个key发生变化就进行持久化  
save 300 10  
save 60 10000
```

判定 key 变化的依据:

- 对数据产生了影响，不包括查询
- 不进行数据比对，比如 name 键存在，重新 set name seazean 也算一次变化

save 配置要根据实际业务情况进行设置，频度过高或过低都会出现性能问题，结果可能是灾难性的

自动原理

服务器状态相关的属性:

```
struct redisServer {  
    // 记录了保存条件的数组  
    struct saveparam *saveparams;  
  
    // 修改计数器  
    long long dirty;  
  
    // 上一次执行保存的时间  
    time_t lastsave;  
};
```

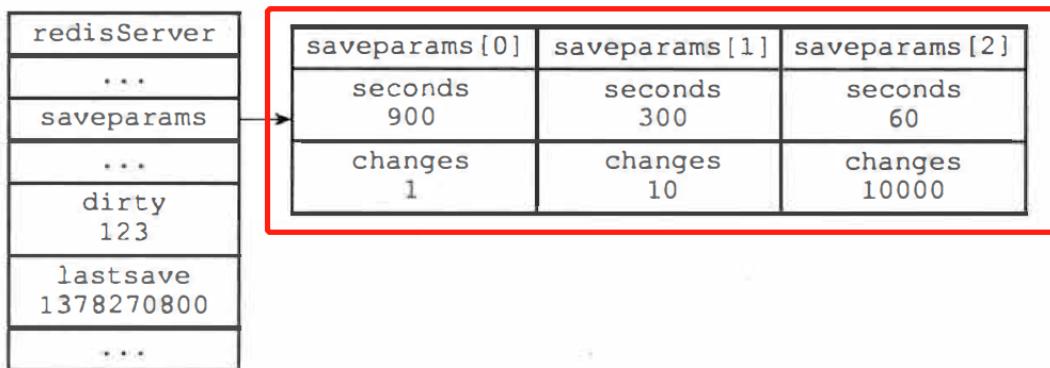
- Redis 服务器启动时，可以通过指定配置文件或者传入启动参数的方式设置 save 选项，如果没有自定义就设置为三个默认值（上节提及），设置服务器状态 redisServer.saveparams 属性，该数组每一项为一个 saveparam 结构，代表 save 的选项设置

```
struct saveparam {
    // 秒数
    time_t seconds
    // 修改数
    int changes;
};
```

- dirty 计数器记录距离上一次成功执行 SAVE 或者 BGSAVE 命令之后，服务器中的所有数据库进行了多少次修改（包括写入、删除、更新等操作），当服务器成功执行一个修改指令，该命令修改了多少次数据库，dirty 的值就增加多少
- lastsave 属性是一个 UNIX 时间戳，记录了服务器上一次成功执行 SAVE 或者 BGSAVE 命令的时间

Redis 的服务器周期性操作函数 serverCron 默认每隔 100 毫秒就会执行一次，该函数用于对正在运行的服务器进行维护

serverCron 函数的其中一项工作是检查 save 选项所设置的保存条件是否满足，会遍历 saveparams 数组中的**所有保存条件**，只要有任意一个条件被满足服务器就会执行 BGSAVE 命令



文件结构

RDB 的存储结构：图示全大写单词标示常量，用全小写单词标示变量和数据

| | | | | |
|-------|------------|-----------|-----|-----------|
| REDIS | db_version | databases | EOF | check_sum |
|-------|------------|-----------|-----|-----------|

- REDIS：长度为 5 字节，保存着 REDIS 五个字符，是 RDB 文件的开头，在载入文件时可以快速检查所载入的文件是否 RDB 文件
- db_version：长度为 4 字节，是一个用字符串表示的整数，记录 RDB 的版本号
- database：包含着零个或任意多个数据库，以及各个数据库中的键值对数据
- EOF：长度为 1 字节的常量，标志着 RDB 文件正文内容的结束，当读入遇到这个值时，代表所有数据库的键值对都已经载入完毕

- check_sum: 长度为 8 字节的无符号整数，保存着一个校验和，该值是通过 REDIS、db_version、databases、EOF 四个部分的内容进行计算得出。服务器在载入 RDB 文件时，会将载入数据所计算出的校验和与 check_sum 所记录的校验和进行对比，来检查 RDB 文件是否有出错或者损坏

Redis 本身带有 RDB 文件检查工具 redis-check-dump

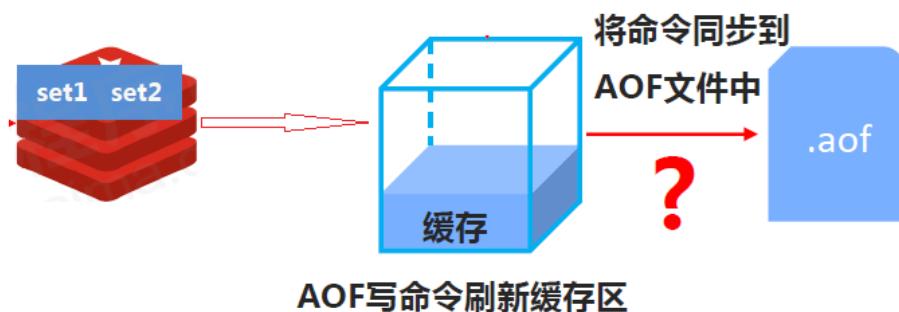
AOF

基本概述

AOF (append only file) 持久化：以独立日志的方式记录每次写命令（不记录读）来记录数据库状态，**增量保存**只许追加文件但不可以改写文件，与 RDB 相比可以理解为由记录数据改为记录数据的变化

AOF 主要作用是解决了**数据持久化的实时性**，目前已经是 Redis 持久化的主流方式

AOF 写数据过程：



Redis 只会将对数据库进行了修改的命令写入到 AOF 文件，并复制到各个从服务器，但是 PUBSUB 和 SCRIPT LOAD 命令例外：

- PUBSUB 命令虽然没有修改数据库，但 PUBSUB 命令向频道的所有订阅者发送消息这一行为带有副作用，接收到消息的所有客户端的状态都会因为这个命令而改变，所以服务器需要使用 REDIS_FORCE_AOF 标志强制将这个命令写入 AOF 文件。这样在将来载入 AOF 文件时，服务器就可以再次执行相同的 PUBSUB 命令，并产生相同的副作用
- SCRIPT LOAD 命令虽然没有修改数据库，但它修改了服务器状态，所以也是一个带有副作用的命令，需要使用 REDIS_FORCE_AOF

持久实现

AOF 持久化功能的实现可以分为命令追加 (append)、文件写入、文件同步 (sync) 三个步骤

命令追加

启动 AOF 的基本配置：

```
appendonly yes|no          #开启AOF持久化功能，默认no，即不开启状态  
appendfilename filename    #AOF持久化文件名，默认appendonly.aof，建议设置  
appendonly-端口号.aof  
dir                         #AOF持久化文件保存路径，与RDB持久化文件路径保持一致即可
```

当 AOF 持久化功能处于打开状态时，服务器在执行完一个写命令之后，会以协议格式将被执行的写命令 **追加到** 服务器状态的 aof_buf 缓冲区的末尾

```
struct redisServer {  
    // AOF 缓冲区  
    sds aof_buf;  
};
```

文件写入

服务器在处理文件事件时会执行**写命令**，**追加一些内容到 aof_buf 缓冲区里**，所以服务器每次结束一个事件循环之前，就会执行 flushAppendOnlyFile 函数，判断是否需要**将 aof_buf 缓冲区中的内容写入和保存到 AOF 文件里**

flushAppendOnlyFile 函数的行为由服务器配置的 appendfsync 选项的值来决定

```
appendfsync always|everysec|no  #AOF写数据策略：默认为everysec
```

- always：每次写入操作都将 aof_buf 缓冲区中的所有内容**写入并同步**到 AOF 文件
特点：安全性最高，数据零误差，但是性能较低，不建议使用
- everysec：先将 aof_buf 缓冲区中的内容写入到操作系统缓存，判断上次同步 AOF 文件的时间距离现在超过一秒钟，再次进行同步 fsync，这个同步操作是由一个（子）线程专门负责执行的
特点：在系统突然宕机的情况下丢失 1 秒内的数据，准确性较高，性能较高，建议使用，也是默认配置
- no：将 aof_buf 缓冲区中的内容写入到操作系统缓存，但并不进行同步，何时同步由操作系统来决定
特点：整体不可控，服务器宕机会丢失上次同步 AOF 后的所有写指令

文件同步

在现代操作系统中，当用户调用 write 函数将数据写入文件时，操作系统通常会将写入数据暂时保存在一个内存缓冲区空间，等到缓冲区写满或者到达特定时间周期，才真正地将缓冲区中的数据写入到磁盘里面（刷脏）

- 优点：提高文件的写入效率
- 缺点：为写入数据带来了安全问题，如果计算机发生停机，那么保存在内存缓冲区里面的写入数据将会丢失

系统提供了 fsync 和 fdatasync 两个同步函数做**强制硬盘同步**，可以让操作系统立即将缓冲区中的数据写入到硬盘里面，函数会阻塞到写入硬盘完成后返回，保证了数据持久化

异常恢复：AOF 文件损坏，通过 redis-check-aof--fix appendonly.aof 进行恢复，重启 Redis，然后重新加载

文件载入

AOF 文件里包含了重建数据库状态所需的所有写命令，所以服务器只要读入并重新执行一遍 AOF 文件里的命令，就还原服务器关闭之前的数据状态，服务器在启动时，还原数据库状态打印的日志：

```
[8321] 05 Sep 11:58:50.449 * DB loaded from append only file: 0.000 seconds
```

AOF 文件里面除了用于指定数据库的 SELECT 命令是服务器自动添加的，其他都是通过客户端发送的命令

```
* 2\r\n$6\r\nSELECT\r\n$n$1\r\nn0\r\n# 服务器自动添加
* 3\r\n$3\r\nnSET\r\nr\n$3\r\nr\nmsg\r\nr\n$n$5\r\nnhello\r\nr\n
*
5\r\n$4\r\nnSADD\r\nr\n$6\r\nnfruits\r\nn$5\r\nnapple\r\nr\n$n$6\r\nnbanana\r\nn$6\r\nncherry\r
\r\n
```

Redis 读取 AOF 文件并还原数据库状态的步骤：

- 创建一个**不带网络连接的伪客户端**（fake client）执行命令，因为 Redis 的命令只能在客户端上下文中执行，而载入 AOF 文件时所使用的命令来源于本地 AOF 文件而不是网络连接
 - 从 AOF 文件分析并读取一条写命令
 - 使用伪客户端执行被读出的写命令，然后重复上述步骤
-

重写实现

重写策略

AOF 重写：读取服务器当前的数据库状态，**生成新 AOF 文件来替换旧 AOF 文件**，不会对现有的 AOF 文件进行任何读取、分析或者写入操作，而是直接原子替换。新 AOF 文件不会包含任何浪费空间的冗余命令，所以体积通常会比旧 AOF 文件小得多

AOF 重写规则：

- 进程内具有时效性的数据，并且数据已超时将不再写入文件
- 对同一数据的多条写命令合并为一条命令，因为会读取当前的状态，所以直接将当前状态转换为一条命令即可。为防止数据量过大造成客户端缓冲区溢出，对 list、set、hash、zset 等集合类型，**单条指令最多写入 64 个元素**
如 `lpush list1 a`、`lpush list1 b`、`lpush list1 c` 可以转化为：`lpush list1 a b c`
- 非写入类的无效指令将被忽略，只保留最终数据的写入命令，但是 select 指令虽然不更改数据，但是更改了数据的存储位置，此类命令同样需要记录

AOF 重写作用：

- 降低磁盘占用量，提高磁盘利用率
- 提高持久化效率，降低持久化写时间，提高 IO 性能
- 降低数据恢复的用时，提高数据恢复效率

重写原理

AOF 重写程序 `aof_rewrite` 函数可以创建一个新 AOF 文件，但是该函数会进行大量的写入操作，调用这个函数的线程将被长时间阻塞，所以 Redis 将 AOF 重写程序放到 `fork` 的子进程里执行，不会阻塞父进程，重写命令：

```
bgrewriteaof
```

- 子进程进行 AOF 重写期间，服务器进程（父进程）可以继续处理命令请求
- 子进程带有服务器进程的数据副本，使用子进程而不是线程，可以在避免使用锁的情况下，保证数据安全性



Background append only file rewriting started

子进程在进行 AOF 重写期间，服务器进程还需要继续处理命令请求，而新命令可能会对现有的数据库状态进行修改，从而使得服务器当前的数据库状态和重写后的 AOF 文件所保存的数据库状态不一致，所以 Redis 设置了 AOF 重写缓冲区

工作流程：

- Redis 服务器执行完一个写命令，会同时将该命令追加到 AOF 缓冲区和 AOF 重写缓冲区（从创建子进程后才开始写入）

- 当子进程完成 AOF 重写工作之后，会向父进程发送一个信号，父进程在接到该信号之后，会调用一个信号处理函数，该函数执行时会对服务器进程（父进程）造成阻塞（影响很小，类似 JVM STW），主要工作：
 - 将 AOF 重写缓冲区中的所有内容写入到新 AOF 文件中，这时新 AOF 文件所保存的状态将和服务器当前的数据库状态一致
 - 对新的 AOF 文件进行改名，**原子地 (atomic) 覆盖**现有的 AOF 文件，完成新旧两个 AOF 文件的替换

自动重写

触发时机：Redis 会记录上次重写时的 AOF 大小，默认配置是当 AOF 文件大小是上次重写后大小的一倍且文件大于 64M 时触发

```
auto-aof-rewrite-min-size size      #设置重写的基准值，最小文件 64MB，达到这个值开始重写  
auto-aof-rewrite-percentage percent #触发AOF文件执行重写的增长率，当前AOF文件大小超过上一次重写的AOF文件大小的百分之多少才会重写，比如文件达到 100% 时开始重写就是两倍时触发
```

自动重写触发比对参数（运行指令 `info Persistence` 获取具体信息）：

| | |
|------------------|----------------------------|
| aof_current_size | #AOF文件当前尺寸大小（单位：字节） |
| aof_base_size | #AOF文件上次启动和重写时的尺寸大小（单位：字节） |

自动重写触发条件公式：

- $aof_current_size > auto-aof-rewrite-min-size$
- $(aof_current_size - aof_base_size) / aof_base_size \geq auto-aof-rewrite-percentage$

对比

RDB 的特点

- RDB 优点：
 - RDB 是一个紧凑压缩的二进制文件，存储效率较高，但存储数据量较大时，存储效率较低
 - RDB 内部存储的是 Redis 在某个时间点的数据快照，非常适合用于数据备份，全量复制、灾难恢复
 - RDB 恢复数据的速度要比 AOF 快很多，因为是快照，直接恢复
- RDB 缺点：
 - BGSAVE 指令每次运行要执行 fork 操作创建子进程，会牺牲一些性能
 - RDB 方式无论是执行指令还是利用配置，无法做到实时持久化，具有丢失数据的可能性，最后一次持久化后的数据可能丢失

- Redis 的众多版本中未进行 RDB 文件格式的版本统一，可能出现各版本之间数据格式无法兼容

AOF 特点：

- AOF 的优点：数据持久化有**较好的实时性**，通过 AOF 重写可以降低文件的体积
- AOF 的缺点：文件较大时恢复较慢

AOF 和 RDB 同时开启，系统默认取 AOF 的数据（数据不会存在丢失）

应用场景：

- 对数据**非常敏感**，建议使用默认的 AOF 持久化方案，AOF 持久化策略使用 everysecond，每秒钟 fsync 一次，该策略 Redis 仍可以保持很好的处理性能

注意：AOF 文件存储体积较大，恢复速度较慢，因为要执行每条指令

- 数据呈现**阶段有效性**，建议使用 RDB 持久化方案，可以做到阶段内无丢失，且恢复速度较快

注意：利用 RDB 实现紧凑的数据持久化，存储数据量较大时，存储效率较低

综合对比：

- RDB 与 AOF 的选择实际上是在做一种权衡，每种都有利有弊
- 灾难恢复选用 RDB
- 如不能承受数分钟以内的数据丢失，对业务数据非常敏感，选用 AOF；如能承受数分钟以内的数据丢失，且追求大数据集的恢复速度，选用 RDB
- 双保险策略，同时开启 RDB 和 AOF，重启后 Redis 优先使用 AOF 来恢复数据，降低丢失数据的量
- 不建议单独用 AOF，因为可能会出现 Bug，如果只是做纯内存缓存，可以都不用

fork

介绍

fork() 函数创建一个子进程，子进程与父进程几乎是完全相同的进程，系统先给子进程分配资源，然后把父进程的所有数据都复制到子进程中，只有少数值与父进程的值不同，相当于克隆了一个进程

在完成对其调用之后，会产生 2 个进程，且每个进程都会从 **fork() 的返回处开始执行**，这两个进程将执行相同的程序段，但是拥有各自不同的堆段，栈段，数据段，每个子进程都可修改各自的数据段，堆段，和栈段

```
#include<unistd.h>
pid_t fork(void);
// 父进程返回子进程的pid，子进程返回0，错误返回负值，根据返回值的不同进行对应的逻辑处理
```

fork 调用一次，却能够**返回两次**，可能有三种不同的返回值：

- 在父进程中，fork 返回新创建子进程的进程 ID
- 在子进程中，fork 返回 0
- 如果出现错误，fork 返回一个负值，错误原因：
 - 当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN
 - 系统内存不足，这时 errno 的值被设置为 ENOMEM

fpid 的值在父子进程中不同：进程形成了链表，父进程的 fpid 指向子进程的进程 id，因为子进程没有子进程，所以其 fpid 为0

创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的调度策略

每个进程都有一个独特（互不相同）的进程标识符 process ID，可以通过 getpid() 函数获得；还有一个记录父进程 pid 的变量，可以通过 getppid() 函数获得变量的值

使用

基本使用：

```
#include <unistd.h>
#include <stdio.h>
int main () {
    pid_t fpid; // fpid表示fork函数返回的值
    int count = 0;
    fpid = fork();
    if (fpid < 0)
        printf("error in fork!");
    else if (fpid == 0) {
        printf("i am the child process, my process id is %d/n", getpid());
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d/n", getpid());
        count++;
    }
    printf("count: %d/n", count); // 1
    return 0;
}
/* 输出内容：
   i am the child process, my process id is 5574
   count: 1
   i am the parent process, my process id is 5573
   count: 1
*/
```

进阶使用：

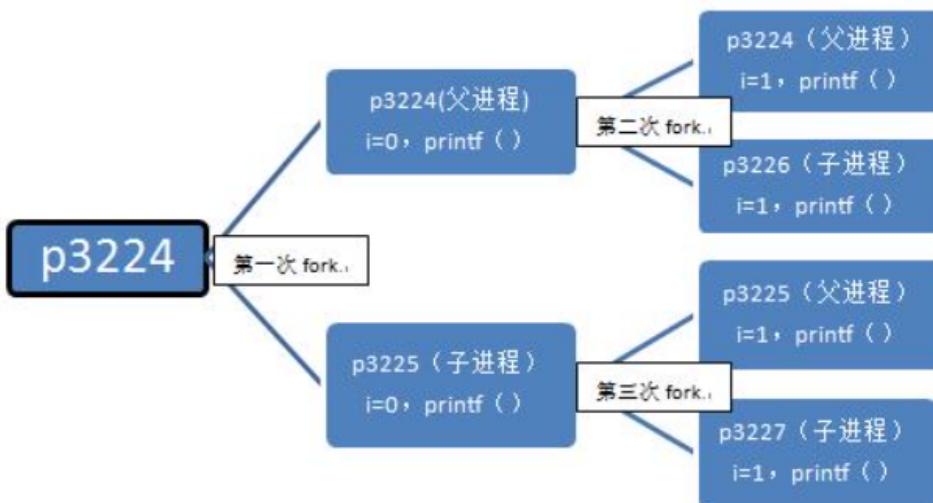
```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
    int i = 0;
    // ppid 指当前进程的父进程pid
    // pid 指当前进程的pid,
    // fpid 指fork返回给当前进程的值，在这可以表示子进程
    for(i = 0; i < 2; i++) {
```

```

pid_t fpid = fork();
if(fpid == 0)
    printf("%d child %4d %4d %4d/n", i, getppid(), getpid(), fpid);
else
    printf("%d parent %4d %4d %4d/n", i, getppid(), getpid(), fpid);
}
return 0;
}

/*输出内容:
   i      父id  id  子id
   0 parent 2043 3224 3225
   0 child  3224 3225    0
   1 parent 2043 3224 3226
   1 parent 3224 3225 3227
   1 child    1 3227    0
   1 child    1 3226    0
*/

```



在 p3224 和 p3225 执行完第二个循环后，main 函数退出，进程死亡。所以 p3226, p3227 就没有父进程了，成为孤儿进程，所以 p3226 和 p3227 的父进程就被置为 ID 为 1 的 init 进程（笔记 Tool → Linux → 进程管理详解）

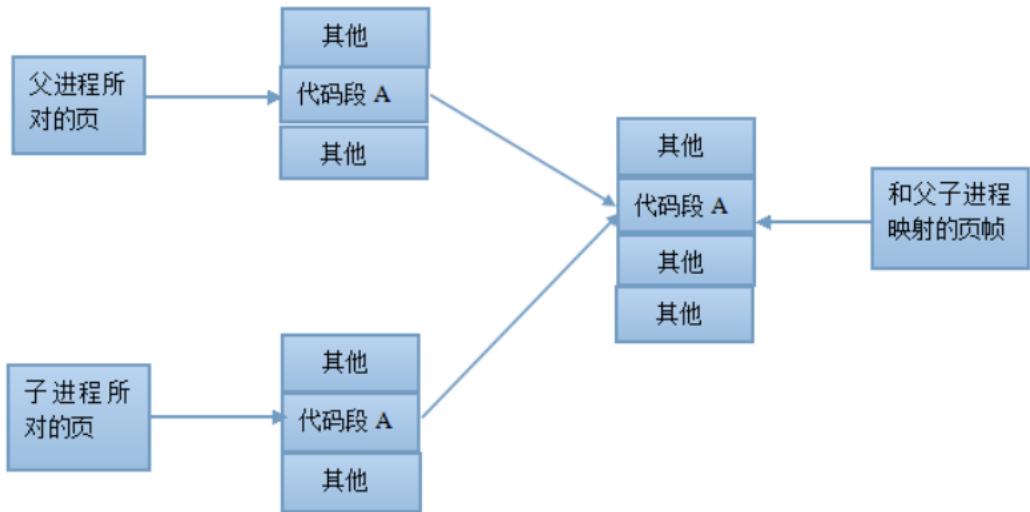
参考文章：https://blog.csdn.net/love_gaohz/article/details/41727415

内存

fork() 调用之后父子进程的内存关系

早期 Linux 的 fork() 实现时，就是全部复制，这种方法效率太低，而且造成了很大的内存浪费，现在 Linux 实现采用了两种方法：

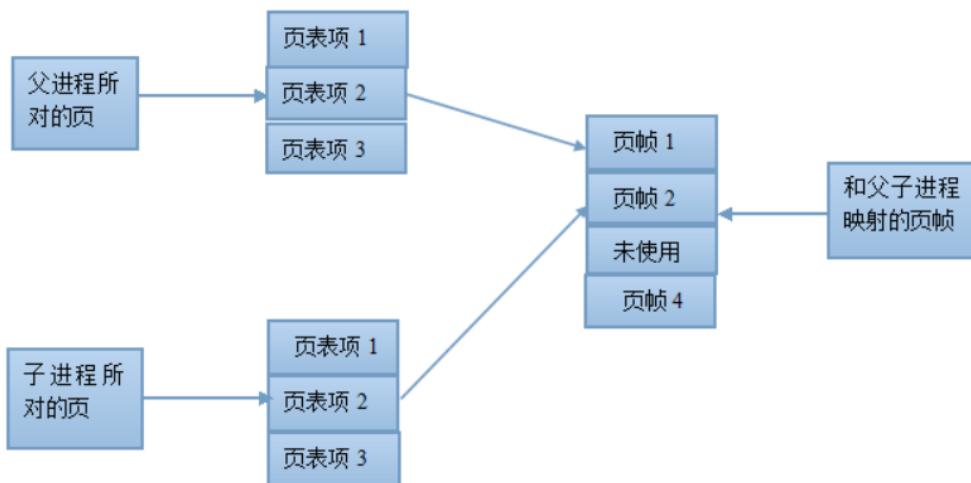
- 父子进程的代码段是相同的，所以代码段是没必要复制的，只需内核将代码段标记为只读，父子进程就共享此代码段。fork() 之后在进程创建代码段时，子进程的进程级页表项都指向和父进程相同的物理页帧



- 对于父进程的数据段，堆段，栈段中的各页，由于父子进程相互独立，采用**写时复制 COW** 的技术，来提高内存以及内核的利用率

在 fork 之后两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，**两者的虚拟空间不同，但其对应的物理空间是同一个**，当父子进程中有关更改相应段的行为发生时，再为子进程相应的段分配物理空间。如果两者的代码完全相同，代码段继续共享父进程的物理空间；而如果两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

fork 之后内核会将子进程放在队列的前面，让子进程先执行，以免父进程执行导致写时复制，而后子进程再执行，因无意义的复制而造成效率的下降



补充知识：

`vfork` (虚拟内存 fork virtual memory fork) : 调用 `vfork()` 父进程被挂起，子进程使用父进程的地址空间。不采用写时复制，如果子进程修改父地址空间的任何页面，这些修改过的页面对于恢复的父进程是可见的

参考文章：<https://blog.csdn.net/Shreck66/article/details/47039937>

事务机制

事务特征

Redis 事务就是将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务去执行其他的命令请求，会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求，Redis 事务的特性：

- Redis 事务没有隔离级别的概念，队列中的命令在事务没有提交之前都不会实际被执行
- Redis 单条命令式保存原子性的，但是事务不保证原子性，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

工作流程

事务的执行流程分为三个阶段：

- 事务开始：MULTI 命令的执行标志着事务的开始，通过在客户端状态的 flags 属性中打开 REDIS_MULTI 标识，将执行该命令的客户端从非事务状态切换至事务状态

```
MULTI # 设定事务的开启位置，此指令执行后，后续的所有指令均加入到事务中
```

- 命令入队：事务队列以先进先出（FIFO）的方式保存入队的命令，每个 Redis 客户端都有事务状态，包含着事务队列：

```
typedef struct redisClient {
    // 事务状态
    multistate mstate; /* MULTI/EXEC state */
}

typedef struct multistate {
    // 事务队列，FIFO顺序
    multicmd *commands;

    // 已入队命令计数
    int count;
}
```

- 如果命令为 EXEC、DISCARD、WATCH、MULTI 四个命中的一个，那么服务器立即执行这个命令
- 其他命令服务器不执行，而是将命令放入一个事务队列里面，然后向客户端返回 QUEUED 回复
- 事务执行：EXEC 提交事务给服务器执行，服务器会遍历这个客户端的事务队列，执行队列中的命令并将执行结果返回

```
EXEC # Commit 提交，执行事务，与multi成对出现，成对使用
```

事务取消的方法：

- 取消事务：

```
DISCARD # 终止当前事务的定义，发生在multi之后，exec之前
```

一般用于事务执行过程中输入了错误的指令，直接取消这次事务，类似于回滚

WATCH

监视机制

WATCH 命令是一个乐观锁（optimistic locking），可以在 EXEC 命令执行之前，监视任意数量的数据键，并在 EXEC 命令执行时，检查被监视的键是否至少有一个已经被修改过了，如果是服务器将拒绝执行事务，并向客户端返回代表事务执行失败的空回复

- 添加监控锁

```
WATCH key1 [key2.....] #可以监控一个或者多个key
```

- 取消对所有 key 的监视

```
UNWATCH
```

实现原理

每个 Redis 数据库都保存着一个 watched_keys 字典，键是某个被 WATCH 监视的数据库键，值则是一个链表，记录了所有监视相应数据库键的客户端：

```
typedef struct redisdb {
    // 正在被 WATCH 命令监视的键
    dict *watched_keys;
}
```

所有对数据库进行修改的命令，在执行后都会调用 multi.c/touchwatchKey 函数对 watched_keys 字典进行检查，是否有客户端正在监视刚被命令修改过的数据库键，如果有的话函数会将监视被修改键的客户端的 REDIS_DIRTY_CAS 标识打开，表示该客户端的事务安全性已经被破坏

服务器接收到个客户端 EXEC 命令时，会根据这个客户端是否打开了 REDIS_DIRTY_CAS 标识，如果打开了说明客户端提交事务不安全，服务器会拒绝执行

ACID

原子性

事务具有原子性 (Atomicity) 、一致性 (Consistency) 、隔离性 (Isolation) 、持久性 (Durability)

原子性指事务队列中的命令要么就全部都执行，要么一个都不执行，但是在命令执行出错时，不会保证原子性（下一节详解）

Redis 不支持事务回滚机制 (rollback)，即使事务队列中的某个命令在执行期间出现了错误，整个事务也会继续执行下去，直到将事务队列中的所有命令都执行完毕为止

回滚需要程序员在代码中实现，应该尽可能避免：

- 事务操作之前记录数据的状态
 - 单数据：string
 - 多数据：hash、list、set、zset
- 设置指令恢复所有的被修改的项
 - 单数据：直接 set（注意周边属性，例如时效）
 - 多数据：修改对应值或整体克隆复制

一致性

事务具有一致性指的是，数据库在执行事务之前是一致的，那么在事务执行之后，无论事务是否执行成功，数据库也应该仍然是一致的

一致是数据符合数据库的定义和要求，没有包含非法或者无效的错误数据，Redis 通过错误检测和简单的设计来保证事务的一致性：

- 入队错误：命令格式输入错误，出现语法错误造成，**整体事务中所有命令均不会执行**，包括那些语法正确的命令

```
192.168.0.137:6379> set name seazean
QUEUED
192.168.0.137:6379> jdhsdj sdh
(error) ERR unknown command `jdhsdj` , with args beginning with: `sdh` ,
192.168.0.137:6379> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
192.168.0.137:6379>
```

- 执行错误：命令执行出现错误，例如对字符串进行 incr 操作，事务中正确的命令会被执行，运行错误的命令不会被执行

```
192.168.0.137:6379> MULTI
OK
192.168.0.137:6379> set name qq
QUEUED
192.168.0.137:6379> INCR name
QUEUED
192.168.0.137:6379> get name
QUEUED
192.168.0.137:6379> EXEC
1) OK
2) (error) ERR value is not an integer or out of range
3) "qq"
```

- 服务器停机：
 - 如果服务器运行在无持久化的内存模式下，那么重启之后的数据库将是空白的，因此数据库是一致的
 - 如果服务器运行在持久化模式下，重启之后将数据库还原到一致的状态
-

隔离性

Redis 是一个单线程的执行原理，所以对于隔离性，分以下两种情况：

- 并发操作在 EXEC 命令前执行，隔离性的保证要使用 WATCH 机制来实现，否则隔离性无法保证
 - 并发操作在 EXEC 命令后执行，隔离性可以保证
-

持久性

Redis 并没有为事务提供任何额外的持久化功能，事务的持久性由 Redis 所使用的持久化模式决定

配置选项 `no-appendfsync-on-rewrite` 可以配合 appendfsync 选项在 AOF 持久化模式使用：

- 选项打开时在执行 BGSAVE 或者 BGREWRITEAOF 期间，服务器会暂时停止对 AOF 文件进行同步，从而尽可能地减少 I/O 阻塞
- 选项打开时运行在 always 模式的 AOF 持久化，事务也不具有持久性，所以该选项默认关闭

在一个事务的最后加上 SAVE 命令总可以保证事务的耐久性

Lua 脚本

环境创建

基本介绍

Redis 对 Lua 脚本支持，通过在服务器中嵌入 Lua 环境，客户端可以使用 Lua 脚本直接在服务器端原子地执行多个命令

```
EVAL <script> <numkeys> [key ...] [arg ...]  
EVALSHA <sha1> <numkeys> [key ...] [arg ...]
```

EVAL 命令可以直接对输入的脚本计算：

```
redis> EVAL "return 1 + 1" 0      # 0代表需要的参数  
(integer) 2
```

EVALSHA 命令根据脚本的 SHA1 校验和来对脚本计算：

```
redis> EVALSHA "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" 0  
(integer) 2
```

应用场景：Redis 只保证单条命令的原子性，所以为了实现原子操作，将多条的对 Redis 的操作整合到一个脚本里，但是避免把不需要做并发控制的操作写入脚本中

Lua 语法特点：

- 声明变量的时候无需指定数据类型，而是用 local 来声明变量为局部变量
- 数组下标是从 1 开始

创建过程

Redis 服务器创建并修改 Lua 环境的整个过程：

- 创建一个基础的 Lua 环境，调用 Lua 的 API 函数 `lua_open`
- 载入多个函数库到 Lua 环境里面，让 Lua 脚本可以使用这些函数库来进行数据操作，包括基础核心函数
- 创建全局变量 `redis` 表格，表格包含以下函数：
 - 执行 Redis 命令的 `redis.call` 和 `redis.pcall` 函数
 - 记录 Redis 日志的 `redis.log` 函数，以及相应的日志级别 (level) 常量 `redis.LOG_DEBUG` 等
 - 计算 SHA1 校验和的 `redis.sha1hex` 函数
 - 返回错误信息的 `redis.error_reply` 函数和 `redis.status_reply` 函数
- 使用 Redis 自制的随机函数来替换 Lua 原有的带有副作用的随机函数，从而避免在脚本中引入副作用
 - `math.random`
 - `math.randomseed`

Redis 要求所有传入服务器的 Lua 脚本，以及 Lua 环境中的所有函数，都必须是无副作用 (side effect) 的纯函数 (pure function)，所以对有副作用的随机函数 `math.random` 和 `math.randomseed` 进行替换

- 创建排序辅助函数 `_redis_compare_helper`，使用辅助函数来对一部分 Redis 命令的结果进行排序，从而消除命令的不确定性

比如集合元素的排列是无序的，所以即使两个集合的元素完全相同，输出结果也不一定相同，Redis 将 S_MEMBERS 这类在相同数据集上产生不同输出的命令称为带有不确定性的命令

- 创建 `redis.pcall` 函数的错误报告辅助函数 `_redis_err_handler`，这个函数可以打印出错代码的来源和发生错误的行数
- 对 Lua 环境中的全局环境进行保护，确保传入服务器的脚本不会因忘记使用 `local` 关键字，而将额外的全局变量添加到 Lua 环境
- 将完成修改的 Lua 环境保存到服务器状态的 `lua` 属性中，等待执行服务器传来的 Lua 脚本

```
struct redisServer {
    Lua *lua;
};
```

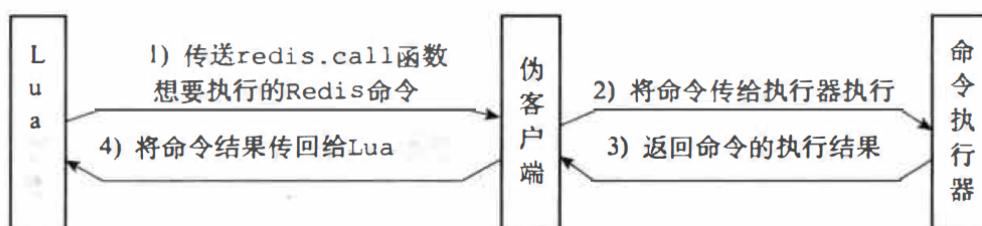
Redis 使用串行化的方式来执行 Redis 命令，所以在任何时间里最多都只会有一个脚本能够被放进 Lua 环境里面运行，因此整个 Redis 服务器只需要创建一个 Lua 环境即可

协作组件

伪客户端

Redis 服务器为 Lua 环境创建了一个伪客户端负责处理 Lua 脚本中包含的所有 Redis 命令，工作流程：

- Lua 环境将 `redis.call` 或者 `redis.pcall` 函数想要执行的命令传给伪客户端
- 伪客户端将命令传给命令执行器
- 命令执行器执行命令并将命令的执行结果返回给伪客户端
- 伪客户端接收命令执行器返回的命令结果，并将结果返回给 Lua 环境
- Lua 将命令结果返回给 `redis.call` 函数或者 `redis.pcall` 函数
- `redis.call` 函数或者 `redis.pcall` 函数会将命令结果作为返回值返回给脚本的调用者



脚本字典

Redis 服务器为 Lua 环境创建 `lua_scripts` 字典，键为某个 Lua 脚本的 SHA1 校验和（checksum），值则是校验和对应的 Lua 脚本

```
struct redisServer {
    dict *lua_scripts;
};
```

服务器会将所有被 `EVAL` 命令执行过的 Lua 脚本，以及所有被 `SCRIPT LOAD` 命令载入过的 Lua 脚本都保存到 `lua_scripts` 字典

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" # 字典的键, SHA1 校验和
```

命令实现

脚本函数

`EVAL` 命令的第一步是为传入的脚本定义一个相对应的 Lua 函数，Lua 函数的名字由 `f_` 前缀加上脚本的 SHA1 校验和（四十个字符长）组成，而函数的体（body）则是脚本本身

```
EVAL "return 'hello world'" 0
# 命令将会定义以下的函数
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91() {
    return 'hello world'
}
```

使用函数来保存客户端传入的脚本有以下优点：

- 通过函数的局部性来让 Lua 环境保持清洁，减少了垃圾回收的工作量，并且避免了使用全局变量
- 如果某个脚本在 Lua 环境中被定义过至少一次，那么只需要 SHA1 校验和，服务器就可以在不知道脚本本身的情况下，直接通过调用 Lua 函数来执行脚本

`EVAL` 命令第二步是将客户端传入的脚本保存到服务器的 `lua_scripts` 字典里，在字典中新添加一个键值对

执行函数

EVAL 命令第三步是执行脚本函数

- 将 EVAL 命令中传入的键名参数和脚本参数分别保存到 KEYS 数组和 ARGV 数组，将这两个数组作为全局变量传入到 Lua 环境
- 为 Lua 环境装载超时处理钩子 (hook)，这个钩子可以在脚本出现超时运行情况时，让客户端通过 `SCRIPT KILL` 命令停止脚本，或者通过 `SHUTDOWN` 命令直接关闭服务器

因为 Redis 是单线程的执行命令，当 Lua 脚本阻塞时需要兜底策略，可以中断执行

- 执行脚本函数
- 移除之前装载的超时钩子
- 将执行脚本函数的结果保存到客户端状态的输出缓冲区里，等待服务器将结果返回给客户端

EVALSHA

EVALSHA 命令的实现原理就是根据脚本的 SHA1 校验和来调用脚本对应的函数，如果函数在 Lua 环境中不存在，找不到 `f_` 开头的函数，就会返回 `SCRIPT NOT FOUND`

管理命令

Redis 中与 Lua 脚本有关的管理命令有四个：

- `SCRIPT FLUSH`: 用于清除服务器中所有和 Lua 脚本有关的信息，会释放并重建 `lua_scripts` 字典，关闭现有的 Lua 环境并重新创建一个新的 Lua 环境
- `SCRIPT EXISTS`: 根据输入的 SHA1 校验和（允许一次传入多个校验和），检查校验和对应的脚本是否存在于服务器中，通过检查 `lua_scripts` 字典实现
- `SCRIPT LOAD`: 在 Lua 环境中为脚本创建相对应的函数，然后将脚本保存到 `lua_scripts` 字典里

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
```

- `SCRIPT KILL`: 停止脚本

如果服务器配置了 `lua-time-limit` 选项，那么在每次执行 Lua 脚本之前，都会设置一个超时处理的钩子。钩子会在脚本运行期间会定期检查运行时间是否超过配置时间，如果超时钩子将定期在脚本运行的间隙中，查看是否有 `SCRIPT KILL` 或者 `SHUTDOWN` 到达：

- 如果超时运行的脚本没有执行过写入操作，客户端可以通过 `SCRIPT KILL` 来停止这个脚本
- 如果执行过写入操作，客户端只能用 `SHUTDOWN nosave` 命令来停止服务器，防止不合法的数据被写入数据库中

脚本复制

命令复制

当服务器运行在复制模式时，具有写性质的脚本命令也会被复制到从服务器，包括 EVAL、EVALSHA、SCRIPT FLUSH，以及 SCRIPT LOAD 命令

Redis 复制 EVAL、SCRIPT FLUSH、SCRIPT LOAD 三个命令的方法和复制普通 Redis 命令的方法一样，当主服务器执行完以上三个命令的其中一个时，会直接将被执行的命令传播（propagate）给所有从服务器，在从服务器中产生相同的效果

EVALSHA

EVALSHA 命令的复制操作相对复杂，因为多个从服务器之间载入 Lua 脚本的清况各有不同，一个在主服务器被成功执行的 EVALSHA 命令，在从服务器执行时可能会出现脚本未找到（not found）错误

Redis 要求主服务器在传播 EVALSHA 命令时，必须确保 EVALSHA 命令要执行的脚本已经被所有从服务器载入过，如果不能确保主服务器会将 EVALSHA 命令转换成一个等价的 EVAL 命令，然后通过传播 EVAL 命令来代替 EVALSHA 命令

主服务器使用服务器状态的 repl_scriptcache_dict 字典记录已经将哪些脚本传播给了所有从服务器，当一个校验和出现在字典时，说明校验和对应的 Lua 脚本已经传播给了所有从服务器，主服务器可以直接传播 EVALSHA 命令

```
struct redisServer {
    // 键是一个个 Lua 脚本的 SHA1 校验和，值则全部都是 NULL
    dict *repl_scriptcache_dict;
}
```

注意：每当主服务器添加一个新的从服务器时，都会清空 repl_scriptcache_dict 字典，因为字典里面记录的脚本已经不再被所有从服务器载入过，所以服务器以清空字典的方式，强制重新向所有从服务器传播脚本

通过使用 EVALSHA 命令指定的 SHA1 校验和，以及 lua_scripts 字典保存的 Lua 脚本，可以将一个 EVALSHA 命令转化为 EVAL 命令

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
# -> 转换
EVAL "return'hello world'" 0
```

脚本内容 "return'hello world'" 来源于 lua_scripts 字典
5332031c6b470dc5a0dd9b4bf2030dea6d65de91 键的值

分布式锁

基本操作

在分布式场景下，锁变量需要由一个共享存储系统来维护，多个客户端才可以通过访问共享存储系统来访问锁变量，加锁和释放锁的操作就变成了读取、判断和设置共享存储系统中的锁变量值多步操作

Redis 分布式锁的基本使用，悲观锁

- 使用 SETNX 设置一个公共锁

```
SETNX lock-key value      # value任意数，返回为1设置成功，返回为0设置失败
```

NX：只在键不存在时，才对键进行设置操作，`SET key value NX` 效果等同于 `SETNX key`

value

XX：只在键已经存在时，才对键进行设置操作

EX：设置键 key 的过期时间，单位时秒

PX：设置键 key 的过期时间，单位时毫秒

说明：由于 `SET` 命令加上选项已经可以完全取代 `SETNX`、`SETEX`、`PSETEX` 的功能，Redis 不推荐使用这几个命令

- 操作完毕通过 DEL 操作释放锁

```
DEL lock-key
```

- 使用 EXPIRE 为锁 key 添加存活（持有）时间，过期自动删除（放弃）锁，防止线程出现异常，无法释放锁

```
EXPIRE lock-key second  
PEXPIRE lock-key milliseconds
```

通过 EXPIRE 设置过期时间缺乏原子性，如果在 `SETNX` 和 `EXPIRE` 之间出现异常，锁也无法释放

- 在 SET 时指定过期时间，保证原子性

```
SET key value NX [EX seconds | PX milliseconds]
```

防误删

场景描述：线程 A 正在执行，但是业务阻塞，在锁的过期时间内未执行完成，过期删除后线程 B 重新获取到锁，此时线程 A 执行完成，删除锁，导致线程 B 的锁被线程 A 误删

SETNX 获取锁时，设置一个指定的唯一值（UUID），释放前获取这个值，判断是否是自己的锁，防止出现线程之间误删了其他线程的锁

```
// 加锁，unique_value 作为客户端唯一性的标识，  
// PX 10000 则表示 lock_key 会在 10s 后过期，以免客户端在这期间发生异常而无法释放锁  
SET lock_key unique_value NX PX 10000
```

Lua 脚本（unlock.script）实现的释放锁操作的伪代码：key 类型参数会放入 KEYS 数组，其它参数会放入 ARGV 数组，在脚本中通过 KEYS 和 ARGV 传递参数，保证判断标识和释放锁这两个操作的原子性

```
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 lock_key unique_value # 1 代  
表需要一个参数
```

```
// 释放锁，KEYS[1] 就是锁的 key，ARGV[1] 就是标识值，避免误释放  
// 获取标识值，判断是否与当前线程标识一致  
if redis.call("get", KEYS[1]) == ARGV[1] then  
    return redis.call("del", KEYS[1])  
else  
    return 0  
end
```

优化锁

不可重入

不可重入：同一个线程无法多次获取同一把锁

使用 hash 键，filed 是加锁的线程标识，value 是锁重入次数

| key | value |
|----------|---------|
| filed | value |
| lock_key | thread1 |

锁重入：

- 加锁时判断锁的 filed 属性是否是当前线程，如果是将 value 加 1
- 解锁时判断锁的 filed 属性是否是当前线程，首先将 value 减一，如果 value 为 0 直接释放锁

使用 Lua 脚本保证多条命令的原子性

不可重试

不可重试：获取锁只尝试一次就返回 false，没有重试机制

- 利用 Lua 脚本尝试获取锁，获取失败获取锁的剩余超时时间 ttl，或者通过参数传入线程抢锁允许等待的时间
 - 利用订阅功能订阅锁释放的信息，然后线程挂起等待 ttl 时间
 - 利用 Lua 脚本在释放锁时，发布一条锁释放的消息
-

超时释放

超时释放：锁超时释放可以避免死锁，但如果是业务执行耗时较长，需要进行锁续时，防止业务未执行完提前释放锁

看门狗 Watch Dog 机制：

- 获取锁成功后，提交周期任务，每隔一段时间（Redisson 中默认为过期时间 / 3），重置一次超时时间
 - 如果服务宕机，Watch Dog 机制线程就停止，就不会再延长 key 的过期时间
 - 释放锁后，终止周期任务
-

主从一致

主从一致性：集群模式下，主从同步存在延迟，当加锁后主服务器宕机时，从服务器还没同步主服务器中的锁数据，此时从服务器升级为主服务器，其他线程又可以获取到锁

将服务器升级为多主多从：

- 获取锁需要从所有主服务器 SET 成功才算获取成功
 - 某个 master 宕机，slave 还没有同步锁数据就升级为 master，其他线程尝试加锁会加锁失败，因为其他 master 上已经存在该锁
-

主从复制

基本操作

主从介绍

主从复制：一个服务器去复制另一个服务器，被复制的服务器为主服务器 master，复制的服务器为从服务器 slave

- master 用来写数据，执行写操作时，将出现变化的数据自动同步到 slave，很少会进行读取操作
- slave 用来读数据，禁止在 slave 服务器上进行读操作

进行复制中的主从服务器双方的数据库将保存相同的数据，将这种现象称作**数据库状态一致**

主从复制的特点：

- **薪火相传**：一个 slave 可以是下一个 slave 的 master，slave 同样可以接收其他 slave 的连接和同步请求，那么该 slave 作为链条中下一个的 master，可以有效减轻 master 的写压力，去中心化降低风险

注意：主机挂了，从机还是从机，无法写数据了

- **反客为主**：当一个 master 宕机后，后面的 slave 可以立刻升为 master，其后面的 slave 不做任何修改

主从复制的作用：

- **读写分离**：master 写、slave 读，提高服务器的读写负载能力
- **负载均衡**：基于主从结构，配合读写分离，由 slave 分担 master 负载，并根据需求的变化，改变 slave 的数量，通过多个从节点分担数据读取负载，大大提高 Redis 服务器并发量与数据吞吐量
- 故障恢复：当 master 出现问题时，由 slave 提供服务，实现快速的故障恢复
- 数据冗余：实现数据热备份，是持久化之外的一种数据冗余方式
- 高可用基石：基于主从复制，构建哨兵模式与集群，实现 Redis 的高可用方案

三高架构：

- 高并发：应用提供某一业务要能支持很多客户端同时访问的能力，称为并发
- 高性能：性能最直观的感受就是速度快，时间短
- 高可用：
 - 可用性：应用服务在全年宕机的时间加在一起就是全年应用服务不可用的时间
 - 业界可用性目标 5 个 9，即 99.999%，即服务器年宕机时长低于 315 秒，约 5.25 分钟

操作指令

系统状态指令：

```
INFO replication
```

master 和 slave 互连：

- 方式一：客户端发送命令，设置 slaveof 选项，产生主从结构

```
slaveof masterip masterport
```

- 方式二：服务器带参启动

```
redis-server --slaveof masterip masterport
```

- 方式三：服务器配置（主流方式）

```
slaveof masterip masterport
```

主从断开连接：

- slave 断开连接后，不会删除已有数据，只是不再接受 master 发送的数据，可以作为**从服务器升级为主服务器的指令**

```
slaveof no one
```

授权访问：master 有服务端和客户端，slave 也有服务端和客户端，不仅服务端之间可以发命令，客户端也可以

- master 客户端发送命令设置密码：

```
requirepass password
```

master 配置文件设置密码：

```
config set requirepass password  
config get requirepass
```

- slave 客户端发送命令设置密码：

```
auth password
```

slave 配置文件设置密码：

```
masterauth password
```

slave 启动服务器设置密码：

```
redis-server -a password
```

复制流程

旧版复制

Redis 的复制功能分为同步 (sync) 和命令传播 (command propagate) 两个操作，主从库间的复制是异步进行的

同步操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态，该过程又叫全量复制：

- 从服务器向主服务器发送 SYNC 命令来进行同步
- 收到 SYNC 的主服务器执行 BGSAVE 命令，在后台生成一个 RDB 文件，并使用一个**缓冲区**记录从现在开始执行的所有**写命令**
- 当 BGSAVE 命令执行完毕时，主服务器会将 RDB 文件发送给从服务器
- 从服务接收并载入 RDB 文件（从服务器会**清空原有数据**）
- 缓冲区记录了 RDB 文件所在状态后的所有写命令，主服务器将在缓冲区的所有命令发送给从服务器，从服务器执行这些写命令
- 至此从服务器的数据库状态和主服务器一致

命令传播用于在主服务器的数据库状态被修改，导致主从数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态

- 主服务器会将自己执行的写命令，也即是造成主从服务器不一致的那条写命令，发送给从服务器
 - 从服务器接受命令并执行，主从服务器将再次回到一致状态
-

功能缺陷

SYNC 本身就是一个非常消耗资源的操作，每次执行 SYNC 命令，都需要执行以下动作：

- 生成 RDB 文件，耗费主服务器大量 CPU、内存和磁盘 I/O 资源
- RDB 文件发送给从服务器，耗费主从服务器大量的网络资源（带宽和流量），并对主服务器响应命令请求的时间产生影响
- 从服务器载入 RDB 文件，期间会因为阻塞而没办法处理命令请求

SYNC 命令下的从服务器对主服务器的复制分为两种情况：

- 初次复制：从服务器没有复制过任何主服务器，或者从服务器当前要复制的主服务器和上一次复制的主服务器不同
- 断线后重复制：处于命令传播阶段的主从服务器因为网络原因而中断了复制，自动重连后并继续复制主服务器

旧版复制在断线后重复制时，也会创建 RDB 文件进行**全量复制**，但是从服务器只需要断线时间内的这部分数据，所以旧版复制的实现方式非常浪费资源

新版复制

Redis 从 2.8 版本开始，使用 PSYNC 命令代替 SYNC 命令来执行复制时的**同步操作**（命令传播阶段相同），解决了旧版复制在处理断线重复制情况的低效问题

PSYNC 命令具有完整重同步 (full resynchronization) 和**部分重同步** (partial resynchronization) 两种模式：

- 完整重同步：处理初次复制情况，执行步骤和 SYNC 命令基本一样
 - 部分重同步：处理断线后重复制情况，主服务器可以将主从连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态，该过程又叫**部分复制**
-

部分同步

部分重同步功能由以下三个部分构成：

- 主服务器的复制偏移量 (replication offset) 和从服务器的复制偏移量
- 主服务器的复制积压缓冲区 (replication backlog)
- 服务器的运行 ID (run ID)

偏移量

主服务器和从服务器会分别维护一个复制偏移量：

- 主服务器每次向从服务器传播 N 个字节的数据时，就将自己的复制偏移量的值加上 N
- 从服务器每次收到主服务器传播来的 N 个字节的数据时，就将自己的复制偏移量的值加上 N

通过对比主从服务器的复制偏移量，可以判断主从服务器是否处于一致状态

- 主从服务器的偏移量是相同的，说明主从服务器处于一致状态
 - 主从服务器的偏移量是不同的，说明主从服务器处于不一致状态
-

缓冲区

复制积压缓冲区是由主服务器维护的一个固定长度 (fixed-size) 先进先出 (FIFO) 队列，默认大小为 1MB

- 出队规则跟普通的先进先出队列一样
- 入队规则是当入队元素的数量大于队列长度时，最先入队的元素会被弹出，然后新元素才会被放入队列

当主服务器进行命令传播时，不仅会将写命令发送给所有从服务器，还会将写命令入队到复制积压缓冲区，缓冲区会保存着一部分最近传播的写命令，并且缓冲区会为队列中的每个字节记录相应的复制偏移量

| | | | | | | | | | | | | | |
|-----|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|
| 偏移量 | ... | 10087 | 10088 | 10089 | 10090 | 10091 | 10092 | 10093 | 10094 | 10095 | 10096 | 10097 | ... |
| 字节值 | ... | '*' | 3 | '\r' | '\n' | '\$' | 3 | '\r' | '\n' | 'S' | 'E' | 'T' | ... |

从服务器会通过 PSYNC 命令将自己的复制偏移量 offset 发送给主服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：

- offset 之后的数据（即 offset+1）仍然存在于复制积压缓冲区里，那么主服务器将对从服务器执行部分重同步操作
- offset 之后的数据已经不在复制积压缓冲区，说明部分数据已经丢失，那么主服务器将对从服务器执行完整重同步操作

复制缓冲区大小设定不合理，会导致**数据溢出**。比如主服务器需要执行大量写命令，又或者主从服务器断线后重连接所需的时间较长，导致缓冲区中的数据已经丢失，则必须进行完整重同步

```
rep1-backlog-size ?mb
```

建议设置如下，这样可以保证绝大部分断线情况都能用部分重同步来处理：

- 从服务器断线后重新连接上主服务器所需的平均时间 second
- 获取 master 平均每秒产生写命令数据总量 write_size_per_second
- 最优复制缓冲区空间 = 2 * second * write_size_per_second

运行ID

服务器运行 ID (run ID)：是每一台服务器每次运行的身份识别码，在服务器启动时自动生成，由 40 位随机的十六进制字符组成，一台服务器多次运行可以生成多个运行 ID

作用：服务器间进行传输识别身份，如果想两次操作均对同一台服务器进行，**每次必须操作携带对应的运行 ID**，用于对方识别

从服务器对主服务器进行初次复制时，主服务器将自己的运行 ID 传送给从服务器，然后从服务器会将该运行 ID 保存。当从服务器断线并重新连上一个主服务器时，会向当前连接的主服务器发送之前保存的运行 ID：

- 如果运行 ID 和当前连接的主服务器的运行 ID 相同，说明从服务器断线之前复制的就是当前连接的这个主服务器，执行部分重同步
- 如果不同，需要执行完整重同步操作

PSYNC

PSYNC 命令的调用方法有两种

- 如果从服务器之前没有复制过任何主服务器，或者执行了 `SLAVEOF no one`，开始一次新的复制时将向主服务器发送 `PSYNC ? -1` 命令，主动请求主服务器进行完整重同步
- 如果从服务器已经复制过某个主服务器，那么从服务器在开始一次新的复制时将向主服务器发送 `PSYNC <runid> <offset>` 命令，`runid` 是上一次复制的主服务器的运行 ID，`offset` 是复制的偏移量

接收到 PSYNC 命令的主服务器会向从服务器返回以下三种回复的其中一种：

- 执行完整重同步操作：返回 `+FULLRESYNC <runid> <offset>`，`runid` 是主服务器的运行 ID，`offset` 是主服务器的复制偏移量
- 执行部分重同步操作：返回 `+CONTINUE`，从服务器收到该回复说明只需要等待主服务器发送缺失的部分数据即可
- 主服务器的版本低于 Redis2.8：返回 `-ERR`，版本过低识别不了 PSYNC，从服务器将向主服务器发送 SYNC 命令

复制实现

实现流程

通过向从服务器发送 SLAVEOF 命令，可以让从服务器去复制一个主服务器

- 设置主服务器的地址和端口：将 SLAVEOF 命令指定的 ip 和 port 保存到服务器状态 redisServer

```
struct redisServer {
    // 主服务器的地址
    char *masterhost;
    // 主服务器的端口
    int masterport;
};
```

SLAVEOF 命令是一个**异步命令**，在完成属性的设置后服务器直接返回 OK，而实际的复制工作将在 OK 返回之后才真正开始执行

- 建立套接字连接：
 - 从服务器 connect 主服务器建立套接字连接，成功后从服务器将为这个套接字关联一个用于复制工作的文件事件处理器，负责执行后续的复制工作，如接收 RDB 文件、接收主服务器传播来的写命令等
 - 主服务器在接受 accept 从服务器的套接字连接后，将为该套接字创建相应的客户端状态，将从服务器看作一个客户端，从服务器将同时具有 server 和 client（可以发命令）两个身份
- 发送 PING 命令：从服务器向主服务器发送一个 PING 命令，检查主从之间的通信是否正常、主服务器处理命令的能力是否正常
 - 返回错误，表示主服务器无法处理从服务器的命令请求（忙碌），从服务器断开并重新创建连向主服务器的套接字

- 返回命令回复，但从服务器不能在规定的时间内读取出命令人回复的内容，表示主从之间的网络状态不佳，需要断开重连
- 读取到 PONG，表示一切状态正常，可以执行复制
- 身份验证：如果从服务器设置了 masterauth 选项就进行身份验证，将向主服务器发送一条 AUTH 命令，命令参数为从服务器 masterauth 选项的值，如果主从设置的密码不相同，那么主将返回一个 invalid password 错误
- 发送端口信息：身份验证后
 - 从服务器执行命令 `REPLCONF listening-port <portnumber>`，向主服务器发送从服务器的监听端口号
 - 主服务器在接收到这个命令后，会将端口号记录在对应的客户端状态 `redisClient.slave_listening_port` 属性中：
- 同步：从服务器将向主服务器发送 PSYNC 命令，在同步操作执行之后，**主从服务器双方都是对方的客户端**，可以相互发送命令
 - 完整重同步：主服务器需要成为从服务器的客户端，才能将保存在缓冲区里面的写命令发送给从服务器执行
 - 部分重同步：主服务器需要成为从服务器的客户端，才能向从服务器发送保存在复制积压缓冲区里面的写命令
- 命令传播：主服务器将写命令发送给从服务器，保持数据库的状态一致

复制图示



心跳检测

心跳机制

心跳机制：进入命令传播阶段，从服务器默认会以每秒一次的频率，向主服务器发送命令：REPLCONF ACK <replication_offset>，replication_offset 是从服务器当前的复制偏移量

心跳的作用：

- 检测主从服务器的网络连接状态
- 辅助实现 min-slaves 选项
- 检测命令丢失

网络状态

如果主服务器超过一秒钟没有收到从服务器发来的 REPLCONF ACK 命令，主服务就认为主从服务器之间的连接出现问题

向主服务器发送 INFO replication 命令，lag 一栏表示从服务器最后一次向主服务器发送 ACK 命令距离现在多少秒：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0: ip=127.0.0.1,port=11111,state=online,offset=123,lag=0 # 刚刚发送过
REPLCONF ACK
slave1: ip=127.0.0.1,port=22222,state=online,offset=456,lag=3 # 3秒之前发送过
REPLCONF ACK
```

在一般情况下，lag 的值应该在 0 或者 1 秒之间跳动，如果超过 1 秒说明主从服务器之间的连接出现了故障

配置选项

Redis 的 min-slaves-to-write 和 min-slaves-max-lag 两个选项可以防止主服务器在**不安全的情况下**拒绝执行写命令

比如向主服务器设置：

- min-slaves-to-write：主库最少有 N 个健康的从库存活才能执行写命令，没有足够的从库直接拒绝写入
- min-slaves-max-lag：从库和主库进行数据复制时的 ACK 消息延迟的最大时间

```
min-slaves-to-write 5
min-slaves-max-lag 10
```

那么在从服务器的数少于 5 个，或者 5 个从服务器的延迟 (lag) 值都大于或等于10 秒时，主服务器将拒绝执行写命令

命令丢失

检测命令丢失：由于网络或者其他原因，主服务器传播给从服务器的写命令丢失，那么当从服务器向主服务器发送 REPLICATION ACK 命令时，主服务器会检查从服务器的复制偏移量是否小于自己的，然后在复制积压缓冲区里找到从服务器缺少的数据，并将这些数据重新发送给从服务器

说明：REPLICATION ACK 命令和复制积压缓冲区都是 Redis 2.8 版本新增的，在 Redis 2.8 版本以前，即使命令在传播过程中丢失，主从服务器都不会注意到，也不会向从服务器补发丢失的数据，所以为了保证**主从复制的数据一致性**，最好使用 2.8 或以上版本的 Redis

常见问题

重启恢复

系统不断运行，master 的数据量会越来越大，一旦 **master 重启**，runid 将发生变化，会导致全部 slave 的全量复制操作

解决方法：本机保存上次 runid，重启后恢复该值，使所有 slave 认为还是之前的 master

优化方案：

- master 内部创建 master_replid 变量，使用 runid 相同的策略生成，并发送给所有 slave
- 在 master 关闭时执行命令 `shutdown save`，进行 RDB 持久化，将 runid 与 offset 保存到 RDB 文件中

`redis-check-rdb dump.rdb` 命令可以查看该信息，保存为 repl-id 和 repl-offset

- master 重启后加载 RDB 文件，恢复数据，将 RDB 文件中保存的 repl-id 与 repl-offset 加载到内存中，`master_repl_id = repl-id`, `master_repl_offset = repl-offset`
 - 通过 info 命令可以查看该信息
-

网络中断

master 的 CPU 占用过高或 slave 频繁断开连接

- 出现的原因：
 - slave 每 1 秒发送 REPLICATION ACK 命令到 master
 - 当 slave 接到了慢查询时 (`keys *`, `hgetall` 等)，会大量占用 CPU 性能

- master 每 1 秒调用复制定时函数 replicationCron()，比对 slave 发现长时间没有进行响应
最终导致 master 各种资源（输出缓冲区、带宽、连接等）被严重占用
- 解决方法：通过设置合理的超时时间，确认是否释放 slave

```
repl-timeout      # 该参数定义了超时时间的阈值（默认60秒），超过该值，释放slave
```

slave 与 master 连接断开

- 出现的原因：
 - master 发送 ping 指令频度较低
 - master 设定超时时间较短
 - ping 指令在网络中存在丢包
- 解决方法：提高 ping 指令发送的频度

```
repl-ping-slave-period
```

超时时间 repl-time 的时间至少是 ping 指令频度的5到10倍，否则 slave 很容易判定超时

一致性

网络信息不同步，数据发送有延迟，导致多个 slave 获取相同数据不同步

解决方案：

- **优化主从间的网络环境**，通常放置在同一个机房部署，如使用阿里云等云服务器时要注意此现象
- 监控主从节点延迟（通过offset）判断，如果 slave 延迟过大，**暂时屏蔽程序对该 slave 的数据访问**

```
slave-serve-stale-data yes|no
```

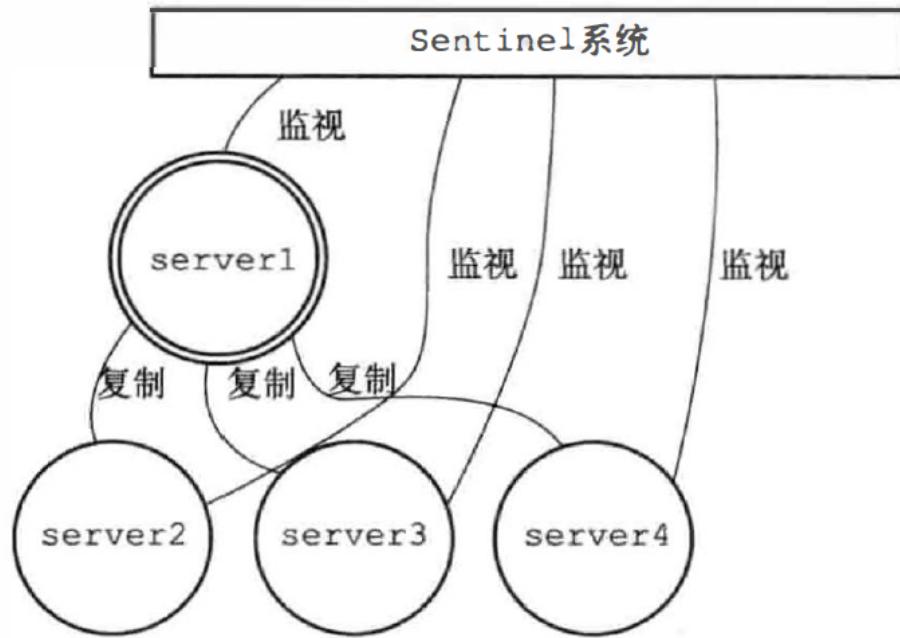
开启后仅响应 info、slaveof 等少数命令（慎用，除非对数据一致性要求很高）

- 多个 slave 同时对 master 请求数据同步，master 发送的 RDB 文件增多，会对带宽造成巨大冲击，造成 master 带宽不足，因此数据同步需要根据业务需求，适量错峰

哨兵模式

哨兵概述

Sentinel (哨兵) 是 Redis 的高可用性 (high availability) 解决方案，由一个或多个 Sentinel 实例 instance 组成的 Sentinel 系统可以监视任意多个主服务器，以及这些主服务器的所有从服务器，并在被监视的主服务器下线时进行故障转移



- 双环图案表示主服务器
- 单环图案表示三个从服务器

哨兵的作用：

- 监控：监控 master 和 slave，不断的检查 master 和 slave 是否正常运行，master 存活检测、master 与 slave 运行情况检测
- 通知：当被监控的服务器出现问题时，向其他哨兵发送通知
- 自动故障转移：断开 master 与 slave 连接，选取一个 slave 作为 master，将其他 slave 连接新的 master，并告知客户端新的服务器地址

启用哨兵

配置方式

配置三个哨兵 sentinel.conf：一般多个哨兵配置相同、端口不同，特殊需求可以配置不同的属性

```
port 26401
dir "/redis/data"
sentinel monitor mymaster 127.0.0.1 6401 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 20000
sentinel parallel-sync mymaster 1
sentinel deny-scripts-reconfig yes
```

配置说明：

- 设置哨兵监听的主服务器信息，判断主观下线所需要的票数

```
sentinel monitor <master-name> <master_ip> <master_port> <quorum>
```

- 指定哨兵在监控 Redis 服务时，设置判定服务器宕机的时长，该设置控制是否进行主从切换

```
sentinel down-after-milliseconds <master-name> <million_seconds>
```

- 出现故障后，故障切换的最大超时时间，超过该值，认定切换失败，默认 3 分钟

```
sentinel failover-timeout <master_name> <million_seconds>
```

- 故障转移时，同时进行主从同步的 slave 数量，数值越大，要求网络资源越高

```
sentinel parallel-syncs <master_name> <sync_slave_number>
```

启动哨兵：服务端命令（Linux 命令）

```
redis-sentinel filename
```

初始化

Sentinel 本质上只是一个运行在特殊模式下的 Redis 服务器，当一个 Sentinel 启动时，首先初始化 Redis 服务器，但是初始化过程和普通 Redis 服务器的初始化过程并不完全相同，哨兵**不提供数据相关服务**，所以不会载入 RDB、AOF 文件

整体流程：

- 初始化服务器
- 将普通 Redis 服务器使用的代码替换成 Sentinel 专用代码
- 初始化 Sentinel 状态
- 根据给定的配置文件，初始化 Sentinel 的监视主服务器列表
- 创建连向主服务器的网络连接

代码替换

将一部分普通 Redis 服务器使用的代码替换成 Sentinel 专用代码

Redis 服务器端口：

```
# define REDIS_SERVERPORT 6379      // 普通服务器端口
# define REDIS_SENTINEL_PORT 26379    // 哨兵端口
```

服务器的命令表：

```
// 普通 Redis 服务器
struct redisCommand redisCommandTable[] = {
    {"get", getCommand, 2, "r", 0, NULL, 1, 1, 1, 0, 0},
    {"set", setCommand, -3, "wm", 0, noPreloadGetKeys, 1, 1, 1, 0, 0},
    //...
}

// 哨兵
struct redisCommand sentinelcmds[] = {
    {"ping", pingCommand, 1, "", 0, NULL, 0, 0, 0, 0, 0},
    {"sentinel", sentinelCommand, -2, "", 0, NULL, 0, 0, 0, 0, 0},
    {"subscribe", ...}, {"unsubscribe", ...}, {"psubscribe", ...},
    {"punsubscribe", ...},
    {"info", ...}
};
```

上述表是哨兵模式下客户端可以执行的命令，所以对于 GET、SET 等命令，服务器根本就没有载入

哨兵状态

服务器会初始化一个 sentinelState 结构，又叫 Sentinel 状态，结构保存了服务器中所有和 Sentinel 功能有关的状态（服务器的一般状态仍然由 redisServer 结构保存）

```
struct sentinelState {
    // 当前纪元，用于实现故障转移
    uint64_t current_epoch;

    // 【保存了所有被这个sentinel监视的主服务器】
    dict *masters;

    // 是否进入了 TILT 模式
    int tilt;
    // 进入 TILT 模式的时间
    mstime_t tilt_start_time;

    // 最后一次执行时间处理的事件
    mstime_t previous_time;

    // 目前正在执行的脚本数量
    int running_scripts;
    // 一个FIFO队列，包含了所有需要执行的用户脚本
    list *scripts_queue;

} sentinel;
```

监控列表

Sentinel 状态的初始化将 masters 字典的初始化，根据被载入的 Sentinel 配置文件 conf 来进行属性赋值

Sentinel 状态中的 masters 字典记录了所有被 Sentinel 监视的**主服务器的相关信息**，字典的键是被监视主服务器的名字，值是主服务器对应的实例结构

实例结构是一个 sentinelRedisinstance 数据类型，代表被 Sentinel 监视的实例，这个实例可以是主、从服务器，或者其他 Sentinel

```
typedef struct sentinelRedisinstance {
    // 标识值，记录了实例的类型，以及该实例的当前状态
    int flags;

    // 实例的名字，主服务器的名字由用户在配置文件中设置,
    // 从服务器和哨兵的名字由 sentinel 自动设置，格式为 ip:port，例如 127.0.0.1:6379
    char *name;

    // 实例运行的 ID
    char *runid;

    // 配置纪元，用于实现故障转移
    uint64_t config_epoch;

    // 实例地址
    sentinelAddr *addr;

    // 如果当前实例时主服务器，该字段保存从服务器信息，键是名字格式为 ip:port，值是实例结构
    dict *slaves;

    // 所有监视当前服务器的 sentinel 实例，键是名字格式为 ip:port，值是实例结构
    dict *sentinels;

    // sentinel down-after-milliseconds 的值，表示实例无响应多少毫秒后会被判断为主观下线
    // (subjectively down)
    mstime_t down_after_period;

    // sentinel monitor 选项中的quorum参数，判断这个实例为客观下线(objectively down)所
    // 需的支持投票数量
    int quorum;

    // sentinel parallel-syncs 的值，在执行故障转移操作时，可以同时对新的主服务器进行同步的
    // 从服务器数量
    int parallel-syncs;

    // sentinel failover-timeout的值，刷新故障迁移状态的最大时限
    mstime_t failover_timeout;
}
```

addr 属性是一个指向 sentinelAddr 的指针：

```
typedef struct sentinelAddr {  
    char *ip;  
    int port;  
}
```

网络连接

初始化 Sentinel 的最后一步是创建连向被监视主服务器的网络连接，Sentinel 将成为主服务器的客户端，可以向主服务器发送命令，并从命裔回复中获取相关的信息

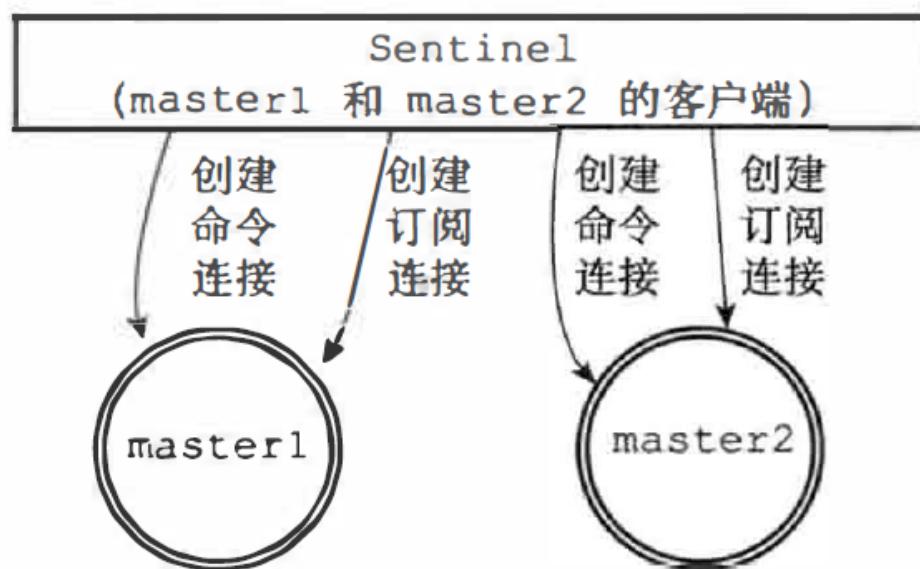
每个被 Sentinel 监视的主服务器，Sentinel 会创建两个连向主服务器的**异步网络连接**：

- 命令连接：用于向主服务器发送命令，并接收命裔回复
- 订阅连接：用于订阅主服务器的 `_sentinel_:hello` 频道

建立两个连接的原因：

- 在 Redis 目前的发布与订阅功能中，被发送的信息都不会保存在 Redis 服务器里，如果在信息发送时接收信息的客户端离线或断线，那么这个客户端就会丢失这条信息，为了不丢失 hello 频道的任何信息，Sentinel 必须用一个订阅连接来接收该频道的信息
- Sentinel 还必须向主服务器发送命令，以此来与主服务器进行通信，所以 Sentinel 还必须向主服务器创建命令连接

说明：断线的意思就是网络连接断开



信息交互

获取信息

主服务器

Sentinel 默认会以每十秒一次的频率，通过命令连接向被监视的主服务器发送 INFO 命令，来获取主服务器的信息

- 一部分是主服务器本身的信息，包括 runid 域记录的服务器运行 ID，以及 role 域记录的服务器角色
- 另一部分是服务器属下所有从服务器的信息，每个从服务器都由一个 slave 字符串开头的行记录，根据这些 IP 地址和端口号，Sentinel 无须用户提供从服务器的地址信息，就可以**自动发现从服务器**

```
# Server
run_id:7611c59dc3a29aa6fa0609f841bb6a1019008a9c
...
# Replication
role:master
...
slave0: ip=127.0.0.1, port=11111, state=online, offset=22, lag=0
slave1: ip=127.0.0.1, port=22222, state=online, offset=22, lag=0
...
```

根据 run_id 和 role 记录的信息 Sentinel 将对主服务器的实例结构进行更新，比如主服务器重启之后，运行 ID 就会和实例结构之前保存的运行 ID 不同，哨兵检测到这一情况之后就会对实例结构的运行 ID 进行更新

对于主服务器返回的从服务器信息，用实例结构的 slaves 字典记录了从服务器的信息：

- 如果从服务器对应的实例结构已经存在，那么 Sentinel 对从服务器的实例结构进行更新
- 如果不存在，为这个从服务器新创建一个实例结构加入字典，字典键为 `ip:port`

从服务器

当 Sentinel 发现主服务器有新的从服务器出现时，会为这个新的从服务器创建相应的实例结构，还会**创建到从服务器的命令连接和订阅连接**，所以 Sentinel 对所有的从服务器之间都可以进行命令操作

Sentinel 默认会以每十秒一次的频率，向从服务器发送 INFO 命令：

```
# Server
run_id:7611c59dc3a29aa6fa0609f841bb6a1019008a9c #从服务器的运行 id
...
# Replication
role:slave          # 从服务器角色
...
master_host:127.0.0.1    # 主服务器的 ip
master_port:6379        # 主服务器的 port
master_link_status:up   # 主从服务器的连接状态
slave_repl_offset:11111 # 从服务器的复制偏移量
slave_priority:100      # 从服务器的优先级
...
```

- **优先级属性**在故障转移时会用到

根据这些信息，Sentinel 会对从服务器的实例结构进行更新

发送信息

Sentinel 在默认情况下，会以每两秒一次的频率，通过命令连接向所有被监视的主服务器和从服务器发送以下格式的命令：

```
PUBLISH _sentinel_:hello "<s_ip>, <s_port>, <s_runid>, <s_epoch>, <m_name>,
<m_ip>, <m_port>, <m_epoch>"
```

这条命令向服务器的 `_sentinel_:hello` 频道发送了一条信息，信息的内容由多个参数组成：

- 以 s_ 开头的参数记录的是 Sentinel 本身的信息
- 以 m_ 开头的参数记录的则是主服务器的信息

说明：通过命令连接发送的频道信息

接受信息

订阅频道

Sentinel 与一个主或从服务器建立起订阅连接之后，就会通过订阅连接向服务器发送订阅命令，频道的订阅会一直持续到 Sentinel 与服务器的连接断开为止

```
SUBSCRIBE _sentinel_:hello
```

订阅成功后，Sentinel 就可以通过订阅连接从服务器的 `_sentinel_:hello` 频道接收信息，对消息分析：

- 如果信息中记录的 Sentinel 运行 ID 与自己的相同，不做进一步处理
- 如果不同，将根据信息中的各个参数，对相应主服务器的实例结构进行更新

Sentinel 为主服务器创建的实例结构的 sentinels 字典保存所有同样监视这个**主服务器的 Sentinel 信息**（包括 Sentinel 自己），字典的键是 Sentinel 的名字，格式为 `ip:port`，值是键所对应 Sentinel 的实例结构

监视同一个服务器的 Sentinel 订阅的频道相同，Sentinel 发送的信息会被其他 Sentinel 接收到（发送信息的为源 Sentinel，接收信息的为目标 Sentinel），目标 Sentinel 在自己的 sentinelState.masters 中查找源 Sentinel 服务器的实例结构进行添加或更新

因为 Sentinel 可以接收到的频道信息来感知其他 Sentinel 的存在，并通过发送频道信息来让其他 Sentinel 知道自己的存在，所以用户在使用 Sentinel 时并不需要提供各个 Sentinel 的地址信息，**监视同一个主服务器的多个 Sentinel 可以相互发现对方**

哨兵实例之间可以相互发现，要归功于 Redis 提供发布订阅机制

命令连接

Sentinel 通过频道信息发现新的 Sentinel，除了创建实例结构，还会创建一个连向新 Sentinel 的命令连接，而新 Sentinel 也同样会创建连向这个 Sentinel 的命令连接，最终监视同一主服务器的多个 Sentinel 将形成相互连接的网络

作用：**通过命令连接相连的各个 Sentinel** 可以向其他 Sentinel 发送命令请求来进行信息交换

Sentinel 之间不会创建订阅连接：

- Sentinel 需要通过接收主服务器或者从服务器发来的频道信息来发现未知的新 Sentinel，所以才创建订阅连接
 - 相互已知的 Sentinel 只要使用命令连接来进行通信就足够了
-

下线检测

主观下线

Sentinel 在默认情况下会以每秒一次的频率向所有与它创建了命令连接的实例（包括主从服务器、其他 Sentinel）发送 PING 命令，通过实例返回的 PING 命令回复来判断实例是否在线

- 有效回复：实例返回 +PONG、-LOADING、-MASTERDOWN 三种回复的其中一种
- 无效回复：实例返回除上述三种以外的任何数据

Sentinel 配置文件中 down-after-milliseconds 选项指定了判断实例进入主观下线所需的时长，如果主服务器在该时间内一直向 Sentinel 返回无效回复，Sentinel 就会在该服务器对应实例结构的 flags 属性打开 SRI_S_DOWN 标识，表示该主服务器进入主观下线状态

配置的 down-after-milliseconds 值不仅适用于主服务器，还会被用于当前 Sentinel 判断主服务器属下的所有从服务器，以及所有同样监视这个主服务器的其他 Sentinel 的主观下线状态

注意：对于监视同一个主服务器的多个 Sentinel 来说，设置的 down-after-milliseconds 选项的值可能不同，所以当一个 Sentinel 将主服务器判断为主观下线时，其他 Sentinel 可能仍然会认为主服务器处于在线状态

客观下线

当 Sentinel 将一个主服务器判断为主观下线之后，会向同样监视这一主服务器的其他 Sentinel 进行询问。Sentinel 使用命令询问其他 Sentinel 是否同意主服务器已下线：

```
SENTINEL is-master-down-by-addr <ip> <port> <current_epoch> <runid>
```

- ip：被 Sentinel 判断为主观下线的主服务器的 IP 地址
- port：被 Sentinel 判断为主观下线的主服务器的端口号
- current_epoch：Sentinel 当前的配置纪元，用于选举领头 Sentinel
- runid：取值为 * 符号代表命令仅仅用于检测主服务器的客观下线状态；取值为 Sentinel 的运行 ID，则用于选举领头 Sentinel

目标 Sentinel 接收到源 Sentinel 的命令时，会根据参数的 IP 和端口号，检查主服务器是否已下线，然后返回一条包含三个参数的 Multi Bulk 回复：

- down_state：返回目标 Sentinel 对服务器的检查结果，1 代表主服务器已下线，0 代表未下线
- leader_runid：取值为 * 符号代表命令仅用于检测服务器的下线状态；而局部领头 Sentinel 的运行 ID 则用于选举领头 Sentinel
- leader_epoch：目标 Sentinel 的局部领头 Sentinel 的配置纪元

源 Sentinel 将统计其他 Sentinel 同意主服务器已下线的数量，当这一数量达到配置指定的判断客观下线所需的数量（quorum）时，Sentinel 会将主服务器对应实例结构 flags 属性的 SRI_O_DOWN 标识打开，代表客观下线，并对主服务器执行故障转移操作。

注意：不同 Sentinel 判断客观下线的条件可能不同，因为载入的配置文件中的属性 quorum 可能不同

领头选举

主服务器被判断为客观下线时，**监视该主服务器的各个 Sentinel 会进行协商**，选举出一个领头 Sentinel 对下线服务器执行故障转移。

Redis 选举领头 Sentinel 的规则：

- 所有在线的 Sentinel 都有被选为领头 Sentinel 的资格
- 每个发现主服务器进入客观下线的 Sentinel 都会要求其他 Sentinel 将自己设置为局部领头 Sentinel
- 在一个配置纪元里，所有 Sentinel 都只有一次将某个 Sentinel 设置为局部领头 Sentinel 的机会，并且局部领头一旦设置，在这个配置纪元里就不能再更改
- Sentinel 设置局部领头 Sentinel 的规则是先到先得，最先向目标 Sentinel 发送设置要求的源 Sentinel 将成为目标 Sentinel 的局部领头 Sentinel，之后接收到的所有设置要求都会被目标

Sentinel 拒绝

- 领头 Sentinel 的产生需要半数以上 Sentinel 的支持，并且每个 Sentinel 只有一票，所以一个配置纪元只会出现一个领头 Sentinel，比如 10 个 Sentinel 的系统中，至少需要 $10/2 + 1 = 6$ 票

选举过程：

- 一个 Sentinel 向目标 Sentinel 发送 `SENTINEL is-master-down-by-addr` 命令，命令中的 runid 参数不是 * 符号而是源 Sentinel 的运行 ID，表示源 Sentinel 要求目标 Sentinel 将自己设置为它的局部领头 Sentinel
- 目标 Sentinel 接受命令处理完成后，将返回一条命令回复，回复中的 leader_runid 和 leader_epoch 参数分别记录了目标 Sentinel 的局部领头 Sentinel 的运行 ID 和配置纪元
- 源 Sentinel 接收目标 Sentinel 命令回复之后，会判断 leader_epoch 是否和自己的相同，相同就继续判断 leader_runid 是否和自己的运行 ID 一致，成立表示目标 Sentinel 将源 Sentinel 设置成了局部领头 Sentinel，即获得一票
- 如果某个 Sentinel 被半数以上的 Sentinel 设置成了局部领头 Sentinel，那么这个 Sentinel 成为领头 Sentinel
- 如果在给定时限内，没有一个 Sentinel 被选举为领头 Sentinel，那么各个 Sentinel 将在一段时间后再次选举，直到选出领头
- 每次进行领头 Sentinel 选举之后，不论选举是否成功，所有 Sentinel 的配置纪元（configuration epoch）都要自增一次

Sentinel 集群至少 3 个节点的原因：

- 如果 Sentinel 集群只有 2 个 Sentinel 节点，则领头选举需要 $2/2 + 1 = 2$ 票，如果一个节点挂了，那就永远选不出领头
- Sentinel 集群允许 1 个 Sentinel 节点故障则需要 3 个节点的集群，允许 2 个节点故障则需要 5 个节点集群

如何获取哨兵节点的半数数量？

- 客观下线是通过配置文件获取的数量，达到 quorum 就客观下线
- 哨兵数量是通过主节点实例结构中，保存着监视该主节点的所有哨兵信息，从而获取得到

故障转移

执行流程

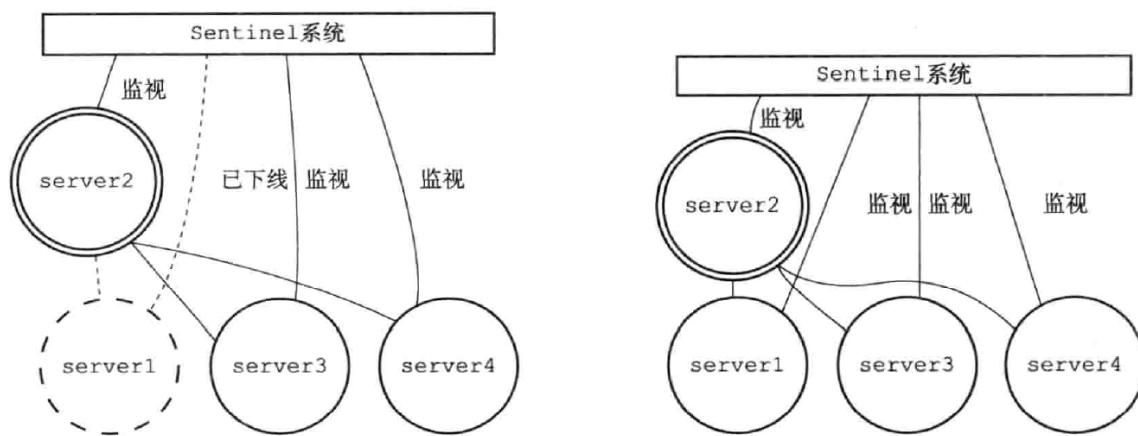
领头 Sentinel 将对已下线的主服务器执行故障转移操作，该操作包含以下三个步骤

- 从下线主服务器属下的所有从服务器里面，挑选出一个从服务器，执行 `SLAVEOF no one`，将从服务器升级为主服务器

在发送 `SLAVEOF no one` 命令后，领头 Sentinel 会以每秒一次的频率（一般是 10s/次）向被升级的从服务器发送 `INFO` 命令，观察命令回复中的角色信息，当被升级服务器的 `role` 从 `slave` 变为 `master` 时，说明从服务器已经顺利升级为主服务器

- 将已下线的主服务器的所有从服务器改为复制新的主服务器，通过向从服务器发送 `SLAVEOF` 命令实现
- 将已经下线的主服务器设置为新的主服务器的从服务器，设置是保存在服务器对应的实例结构中，当旧的主服务器重新上线时，Sentinel 就会向它发送 `SLAVEOF` 命令，成为新的主服务器的从服务器

示例：server1 是主，server2、server3、server4 是从服务器，server1 故障后选中 server2 升级



选择算法

领头 Sentinel 会将已下线主服务器的所有从服务器保存到一个列表里，然后按照以下规则对列表进行过滤，最后挑选出一个**状态良好、数据完整**的从服务器

- 删除列表中所有处于下线或者断线状态的从服务器，保证列表中的从服务器都是正常在线的
- 删除列表中所有最近五秒内没有回复过领头 Sentinel 的 INFO 命令的从服务器，保证列表中的从服务器最近成功进行过通信
- 删除所有与已下线主服务器连接断开超过 `down-after-milliseconds * 10` 毫秒的从服务器，保证列表中剩余的从服务器都没有过早地与主服务器断开连接，保存的数据都是比较新的
`down-after-milliseconds` 时间用来判断是否主观下线，其余的时间完全可以完成客观下线和领头选举
- 根据从服务器的优先级，对列表中剩余的从服务器进行排序，并选出其中**优先级最高**的从服务器
- 如果有多个具有相同最高优先级的从服务器，领头 Sentinel 将对这些相同优先级的服务器按照复制偏移量进行排序，选出其中偏移量最大的从服务器，也就是保存着最新数据的从服务器
- 如果还没选出来，就按照运行 ID 对这些从服务器进行排序，并选出其中运行 ID 最小的从服务器

集群模式

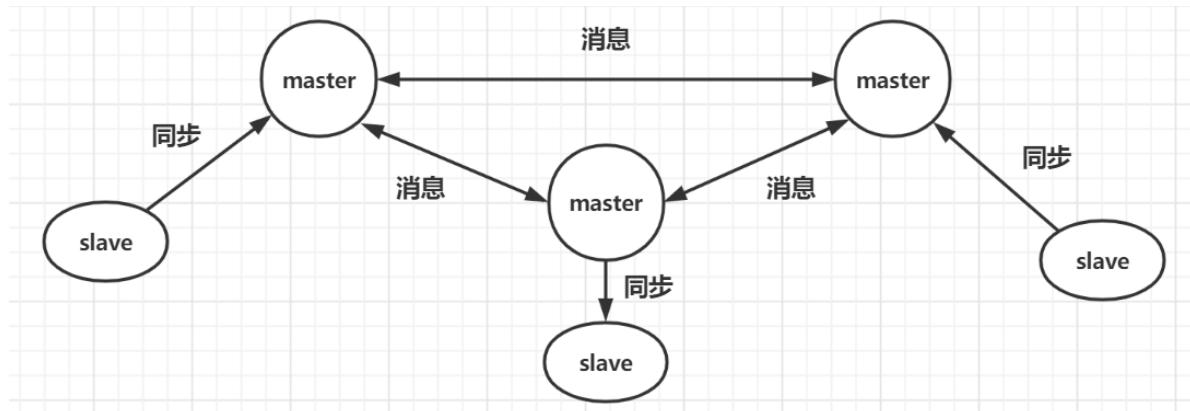
集群节点

节点概述

Redis 集群是 Redis 提供的分布式数据库方案，集群通过分片（sharding）来进行数据共享，并提供复制和故障转移功能，一个 Redis 集群通常由多个节点（node）组成，将各个独立的节点连接起来，构成一个包含多节点的集群。

一个节点就是一个运行在集群模式下的 Redis 服务器，Redis 在启动时会根据配置文件中的 `cluster-enabled` 配置选项是否为 yes 来决定是否开启服务器的集群模式。

节点会继续使用所有在单机模式中使用的服务器组件，使用 `redisServer` 结构来保存服务器的状态，使用 `redisClient` 结构来保存客户端的状态，也有集群特有的数据结构。



数据结构

每个节点都保存着一个集群状态 `clusterState` 结构，这个结构记录了在当前节点的视角下，集群目前所处的状态。

```
typedef struct clusterState {
    // 指向当前节点的指针
    clusterNode *myself;

    // 集群当前的配置纪元，用于实现故障转移
    uint64_t currentEpoch;

    // 集群当前的状态，是在线还是下线
    int state;

    // 集群中至少处理着一个槽的（主）节点的数量，为0表示集群目前没有任何节点在处理槽
    // 【选举时投票数量超过半数，从这里获取的】
    int size;

    // 集群节点名单（包括 myself 节点），字典的键为节点的名字，字典的值为节点对应的
    // clusterNode结构
    dict *nodes;
}
```

每个节点都会使用 clusterNode 结构记录当前状态，并为集群中的所有其他节点（包括主节点和从节点）都创建一个相应的 clusterNode 结构，以此来记录其他节点的状态

```
struct clusterNode {
    // 创建节点的时间
    mstime_t ctime;

    // 节点的名字，由 40 个十六进制字符组成
    char name[REDIS_CLUSTER_NAMELEN];

    // 节点标识，使用各种不同的标识值记录节点的角色（比如主节点或者从节点）以及节点目前所处的状态（比如在线或者下线）
    int flags;

    // 节点当前的配置纪元，用于实现故障转移
    uint64_t configEpoch;

    // 节点的IP地址
    char ip[REDIS_IP_STR_LEN];

    // 节点的端口号
    int port;

    // 保存连接节点所需的有关信息
    clusterLink *link;
}
```

clusterNode 结构的 link 属性是一个 clusterLink 结构，该结构保存了连接节点所需的有关信息

```
typedef struct clusterLink {
    // 连接的创建时间
    mstime_t ctime;

    // TCP套接字描述符
    int fd;

    // 输出缓冲区，保存着等待发送给其他节点的消息(message)。
    sds sndbuf;

    // 输入缓冲区，保存着从其他节点接收到的消息。
    sds rcvbuf;

    // 与这个连接相关联的节点，如果没有的话就为NULL
    struct clusterNode *node;
}
```

- redisClient 结构中的套接字和缓冲区是用于连接客户端的
- clusterLink 结构中的套接字和缓冲区则是用于连接节点的

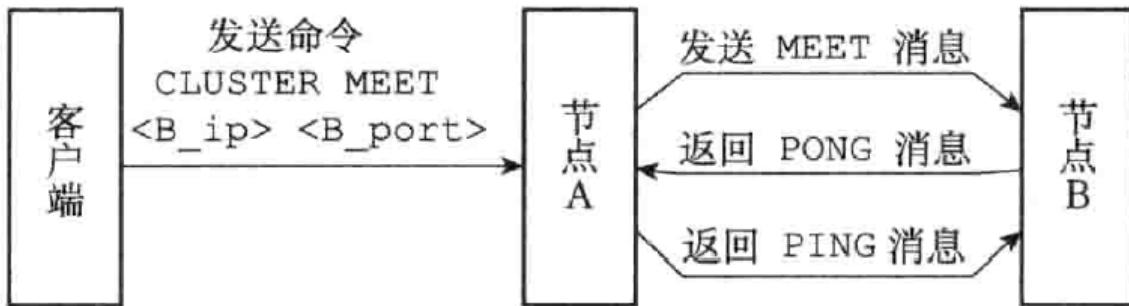
MEET

CLUSTER MEET 命令用来将 ip 和 port 所指定的节点添加到接受命令的节点所在的集群中

```
CLUSTER MEET <ip> <port>
```

假设向节点 A 发送 CLUSTER MEET 命令，让节点 A 将另一个节点 B 添加到节点 A 当前所在的集群里，收到命令的节点 A 将与根据 ip 和 port 向节点 B 进行握手（handshake）：

- 节点 A 会为节点 B 创建一个 clusterNode 结构，并将该结构添加到自己的 clusterState.nodes 字典里，然后节点 A 向节点 B **发送 MEET 消息** (message)
- 节点 B 收到 MEET 消息后，节点 B 会为节点 A 创建一个 clusterNode 结构，并将该结构添加到自己的 clusterState.nodes 字典里，之后节点 B 将向节点 A **返回一条 PONG 消息**
- 节点 A 收到 PONG 消息后，代表节点 A 可以知道节点 B 已经成功地接收到了自己发送的 MEET 消息，此时节点 A 将向节点 B **返回一条 PING 消息**
- 节点 B 收到 PING 消息后，代表节点 B 可以知道节点 A 已经成功地接收到了自己返回的 PONG 消息，握手完成



节点 A 会将节点 B 的信息通过 Gossip 协议传播给集群中的其他节点，让其他节点也与节点 B 进行握手，最终经过一段时间之后，节点 B 会被集群中的所有节点认识

槽指派

基本操作

Redis 集群通过分片的方式来保存数据库中的键值对，集群的整个数据库被分为 16384 个槽（slot），数据库中的每个键都属于 16384 个槽中的一个，集群中的每个节点可以处理 0 个或最多 16384 个槽（**每个主节点存储的数据并不一样**）

- 当数据库中的 16384 个槽都有节点在处理时，集群处于上线状态（ok）
- 如果数据库中有任何一个槽得到处理，那么集群处于下线状态（fail）

通过向节点发送 CLUSTER ADDSLOTS 命令，可以将一个或多个槽指派（assign）给节点负责

```
CLUSTER ADDSLOTS <slot> [slot ... ]
```

```
127.0.0.1:7000> CLUSTER ADDSLOTS 0 1 2 3 4 ... 5000 # 将槽0至槽5000指派给节点7000负责
OK
```

命令执行细节：

- 如果命令参数中有一个槽已经被指派给了某个节点，那么会向客户端返回错误，并终止命令执行
- 将 slots 数组中的索引 i 上的二进制位设置为 1，就代表指派成功

节点指派

clusterNode 结构的 slots 属性和 numslot 属性记录了节点负责处理哪些槽：

```
struct clusterNode {
    // 处理信息，一字节等于 8 位
    unsigned char slots[16384/8];
    // 记录节点负责处理的槽的数量，就是 slots 数组中值为 1 的二进制位数量
    int numslots;
}
```

slots 是一个二进制位数组 (bit array)，长度为 $16384/8 = 2048$ 个字节，包含 16384 个二进制位，Redis 以 0 为起始索引，16383 为终止索引，对 slots 数组的 16384 个二进制位进行编号，并根据索引 i 上的二进制位的值来判断节点是否负责处理槽 i：

- 在索引 i 上的二进制位的值为 1，那么表示节点负责处理槽 i
- 在索引 i 上的二进制位的值为 0，那么表示节点不负责处理槽 i

| 字节 | slots[0] | | | | | | | | slots[1] | | | | | | | | ... | slots[2047] | | |
|----|----------|---|---|---|---|---|---|---|----------|---|----|----|----|----|----|----|-----|-------------|-------|-------|
| 索引 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | ... | ... | 16382 | 16383 |
| 值 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ... | ... | 0 | 0 |

取出和设置 slots 数组中的任意一个二进制位的值的复杂度仅为 O(1)，所以对于一个给定节点的 slots 数组来说，检查节点是否负责处理某个槽或者将某个槽指派给节点负责，这两个动作的复杂度都是 O(1)

传播节点的槽指派信息：一个节点除了会将自己负责处理的槽记录在 clusterNode 中，还会将自己的 slots 数组通过消息发送给集群中的其他节点，每个接收到 slots 数组的节点都会将数组保存到相应节点的 clusterNode 结构里面，因此集群中的每个节点都会知道数据库中的 16384 个槽分别被指派给了集群中的哪些节点

集群指派

集群状态 clusterState 结构中的 slots 数组记录了集群中所有 16384 个槽的指派信息，数组每一项都是一个指向 clusterNode 的指针

```
typedef struct clusterState {  
    // ...  
    clusterNode *slots[16384];  
}
```

- 如果 slots[i] 指针指向 NULL，那么表示槽 i 尚未指派给任何节点
- 如果 slots[i] 指针指向一个 clusterNode 结构，那么表示槽 i 已经指派给该节点所代表的节点

通过该节点，程序检查槽 i 是否已经被指派或者取得负责处理槽 i 的节点，只需要访问 clusterState.slots[i] 即可，时间复杂度仅为 O(1)

集群数据

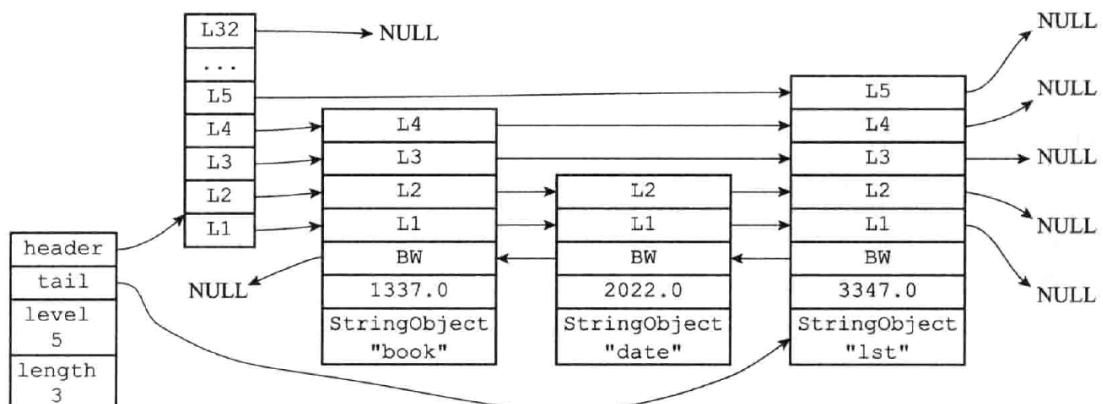
集群节点保存键值对以及键值对过期时间的方式，与单机 Redis 服务器保存键值对以及键值对过期时间的方式完全相同，但是**集群节点只能使用 0 号数据库**，单机服务器可以任意使用

除了将键值对保存在数据库里面之外，节点还会用 clusterState 结构中的 slots_to_keys 跳跃表来**保存槽和键之间的关系**

```
typedef struct clusterState {  
    // ...  
    zskiplist *slots_to_keys;  
}
```

slots_to_keys 跳跃表每个节点的分值 (score) 都是一个槽号，而每个节点的成员 (member) 都是一个数据库键 (按槽号升序)

- 当节点往数据库中添加一个新的键值对时，节点就会将这个键以及键的槽号关联到 slots_to_keys 跳跃表
- 当节点删除数据库中的某个键值对时，节点就会在 slots_to_keys 跳跃表解除被删除键与槽号的关联



通过在 slots_to_keys 跳跃表中记录各个数据库键所属的槽，可以很方便地对属于某个或某些槽的所有数据库键进行批量操作，比如 `CLUSTER GETKEYSINSLOT <slot> <count>` 命令返回最多 count 个属于槽 slot 的数据库键，就是通过该跳表实现

集群命令

执行命令

集群处于上线状态，客户端就可以向集群中的节点发送命令（16384 个槽全部指派就进入上线状态）

当客户端向节点发送与数据库键有关的命令时，接收命令的节点会计算出命令该键属于哪个槽，并检查这个槽是否指派给了自己

- 如果键所在的槽正好就指派给了当前节点，那么节点直接执行这个命令
- 反之，节点会向客户端返回一个 MOVED 错误，指引客户端转向（redirect）至正确的节点，再次发送该命令

计算键归属哪个槽的寻址算法：

```
def slot_number(key):          // CRC16(key) 语句计算键 key 的 CRC-16 校验和
    return CRC16(key) & 16383;  // 取模，十进制对16384的取余
```

使用 `CLUSTER KEYSLOT <key>` 命令可以查看一个给定键属于哪个槽，底层实现：

```
def CLUSTER_KEYSLot(key):
    // 计算槽号
    slot = slot_number(key);
    // 将槽号返回给客户端
    reply_client(slot);
```

判断槽是否由当前节点负责处理：如果 clusterState.slots[i] 不等于 clusterState.myself，那么说明槽 i 并非由当前节点负责，节点会根据 clusterState.slots[i] 指向的 clusterNode 结构所记录的节点 IP 和端口号，向客户端返回 MOVED 错误

MOVED

MOVED 错误的格式为：

```
MOVED <slot> <ip>:<port>
```

参数 slot 为键所在的槽，ip 和 port 是负责处理槽 slot 的节点的 ip 地址和端口号

```
MOVED 12345 127.0.0.1:6380 # 表示槽 12345 正由 IP地址为 127.0.0.1，端口号为 6380 的节点负责
```

当客户端接收到节点返回的 MOVED 错误时，客户端会根据 MOVED 错误中提供的 IP 地址和端口号，转至负责处理槽 slot 的节点重新发送执行的命令

- 一个集群客户端通常会与集群中的多个节点创建套接字连接，节点转向实际上就是换一个套接字来发送命令
- 如果客户端尚未与转向的节点创建套接字连接，那么客户端会先根据 IP 地址和端口号来连接节点，然后再进行转向

集群模式的 redis-cli 在接收到 MOVED 错误时，并不会打印出 MOVED 错误，而是根据错误**自动进行节点转向**，并打印出转向信息：

```
$ redis-cli -c -p 6379 #集群模式
127.0.0.1:6379> SET msg "happy"
-> Redirected to slot [6257] located at 127.0.0.1:6380
OK

127.0.0.1:6379>
```

使用单机 (stand alone) 模式 redis-cli 会打印错误，因为单机模式客户端不清楚 MOVED 错误的作用，不会进行自动转向：

```
$ redis-cli -c -p 6379 #集群模式
127.0.0.1:6379> SET msg "happy"
(error) MOVED 6257 127.0.0.1:6380

127.0.0.1:6379>
```

重新分片

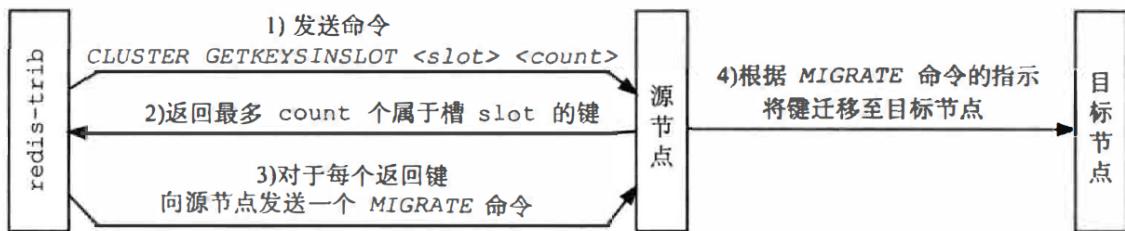
实现原理

Redis 集群的重新分片操作可以将任意数量已经指派给某个节点（源节点）的槽改为指派给另一个节点（目标节点），并且相关槽的键值对也会从源节点被移动到目标节点，该操作是可以在线 (online) 进行，在重新分片的过程中源节点和目标节点都可以处理命令请求

Redis 的集群管理软件 redis-trib 负责执行重新分片操作，redis-trib 通过向源节点和目标节点发送命令来进行重新分片操作

- 向目标节点发送 `CLUSTER SETSLOT <slot> IMPORTING <source_id>` 命令，准备好从源节点导入属于槽 slot 的键值对
- 向源节点发送 `CLUSTER SETSLOT <slot> MIGRATING <target_id>` 命令，让源节点准备好将属于槽 slot 的键值对迁移
- redis-trib 向源节点发送 `CLUSTER GETKEYSINSLOT <slot> <count>` 命令，获得最多 count 个属于槽 slot 的键值对的键名
- 对于每个 key，redis-trib 都向源节点发送一个 `MIGRATE <target_ip> <target_port> <key_name> 0 <timeout>` 命令，将被选中的键原地从源节点迁移至目标节点
- 重复上述步骤，直到源节点保存的所有槽 slot 的键值对都被迁移至目标节点为止

- redis-trib 向集群中的任意一个节点发送 `CLUSTER SETSLOT <slot> NODE <target _id>` 命令，将槽 slot 指派给目标节点，这一指派信息会通过消息传播至整个集群，最终集群中的所有节点都直到槽 slot 已经指派给了目标节点



如果重新分片涉及多个槽，那么 redis-trib 将对每个给定的槽分别执行上面给出的步骤

命令原理

`clusterState` 结构的 `importing_slots_from` 数组记录了当前节点正在从其他节点导入的槽，`migrating_slots_to` 数组记录了当前节点正在迁移至其他节点的槽：

```

typedef struct clusterState {
    // 如果 importing_slots_from[i] 的值不为 NULL, 而是指向一个 clusterNode 结构,
    // 那么表示当前节点正在从 clusterNode 所代表的节点导入槽 i
    clusterNode *importing_slots_from[16384];

    // 表示当前节点正在将槽 i 迁移至 clusterNode 所代表的节点
    clusterNode *migrating_slots_to[16384];
}

```

`CLUSTER SETSLOT <slot> IMPORTING <source_id>` 命令：将目标节点

`clusterState.importing_slots_from[slot]` 的值设置为 `source_id` 所代表节点的 `clusterNode` 结构

`CLUSTER SETSLOT <slot> MIGRATING <target_id>` 命令：将源节点

`clusterState.migrating_slots_to[slot]` 的值设置为 `target_id` 所代表节点的 `clusterNode` 结构

ASK 错误

重新分片期间，源节点向目标节点迁移一个槽的过程中，可能出现被迁移槽的一部分键值对保存在源节点，另一部分保存在目标节点

客户端向源节点发送命令请求，并且命令要处理的数据库键属于被迁移的槽：

- 源节点会先在数据库里面查找指定键，如果找到的话，就直接执行客户端发送的命令
- 未找到会检查 `clusterState.migrating_slots_to[slot]`，看键 `key` 所属的槽 `slot` 是否正在进行迁移

- 槽 slot 正在迁移则源节点将向客户端返回一个 ASK 错误，指引客户端转向正在导入槽的目标节点

```
ASK <slot> <ip:port>
```

- 接到 ASK 错误的客户端，会根据错误提供的 IP 地址和端口号转向目标节点，首先向目标节点发送一个 ASKING 命令，再重新发送原本想要执行的命令

和 MOVED 错误情况类似，集群模式的 redis-cli 在接到 ASK 错误时不会打印错误进行自动转向；单机模式的 redis-cli 会打印错误

对比 MOVED 错误：

- MOVED 错误代表槽的负责权已经从一个节点转移到了另一个节点，转向是一种持久性的转向
- ASK 错误只是两个节点在迁移槽的过程中使用的一种临时措施，ASK 的转向不会对客户端今后发送关于槽 slot 的命令请求产生任何影响，客户端仍然会将槽 slot 的命令请求发送至目前负责处理槽 slot 的节点，除非 ASK 错误再次出现

ASKING

客户端不发送 ASKING 命令，而是直接发送执行的命令，那么客户端发送的命令将被节点拒绝执行，并返回 MOVED 错误

ASKING 命令作用是打开发送该命令的客户端的 REDIS_ASKING 标识，该命令的伪代码实现：

```
def ASKING():
    // 打开标识
    client.flags |= REDIS_ASKING
    // 向客户端返回OK回复
    reply("OK")
```

当前节点正在导入槽 slot，并且发送命令的客户端带有 REDIS_ASKING 标识，那么节点将破例执行这个关于槽 slot 的命令一次

客户端的 REDIS_ASKING 标识是一次性标识，当节点执行了一个带有 REDIS_ASKING 标识的客户端发送的命令之后，该客户端的 REDIS_ASKING 标识就会被移除

高可用

节点复制

Redis 集群中的节点分为主节点（master）和从节点（slave），其中主节点用于处理槽，而从节点则用于复制主节点，并在被复制的主节点下线时，代替下线主节点继续处理命令请求

```
CLUSTER REPLICATE <node_id>
```

向一个节点发送命令可以让接收命令的节点成为 node_id 所指定节点的从节点，并开始对主节点进行复制

- 接受命令的节点首先会在的 clusterState.nodes 字典中找到 node_id 所对应节点的 clusterNode 结构，并将自己的节点中的 clusterState.myself.slaveof 指针指向这个结构，记录这个节点正在复制的主节点
- 节点会修改 clusterState.myself.flags 中的属性，关闭 REDIS_NODE_MASTER 标识，打开 REDIS_NODE_SLAVE 标识
- 节点会调用复制代码，对主节点进行复制（节点的复制功能和单机 Redis 服务器的使用了相同的代码）

一个节点成为从节点，并开始复制某个主节点这一信息会通过消息发送给集群中的其他节点，最终集群中的所有节点都会知道某个从节点正在复制某个主节点

主节点的 clusterNode 结构的 slaves 属性和 numslaves 属性中记录正在复制这个主节点的从节点名单：

```
struct clusterNode {  
    // 正在复制这个主节点的从节点数量  
    int numslaves;  
  
    // 数组项指向一个正在复制这个主节点的从节点的clusterNode结构  
    struct clusterNode **slaves;  
};
```

故障检测

集群中的每个节点都会定期地向集群中的其他节点发送 PING 消息，来检测对方是否在线，如果接收 PING 的节点没有在规定的时间内返回 PONG 消息，那么发送消息节点就会将接收节点标记为**疑似下线** (probable fail)

集群中的节点会互相发送消息，来**交换集群中各个节点的状态信息**，当一个主节点 A 通过消息得知主节点 B 认为主节点 C 进入了疑似下线状态时，主节点 A 会在 clusterState.nodes 字典中找到主节点 C 所对应的节点，并将主节点 B 的下线报告 (failure report) 添加到 clusterNode.fail_reports 链表里面

```
struct clusterNode {  
    // 一个链表，记录了所有其他节点对该节点的下线报告  
    list *fail_reports;  
}  
// 每个下线报告由一个 clusterNodeFailReport 结构表示  
struct clusterNodeFailReport {  
    // 报告目标节点已经下线的节点  
    struct clusterNode *node;  
  
    // 最后一次从node节点收到下线报告的时间  
    // 程序使用这个时间戳来检查下线报告是否过期，与当前时间相差太久的下线报告会被删除  
    mstime_t time;  
};
```

集群里半数以上负责处理槽的主节点都将某个主节点 X 报告为疑似下线，那么 X 将被标记为已下线 (FAIL)，将 X 标记为已下线的节点会向集群广播一条关于主节点 X 的 FAIL 消息，所有收到消息的节点都会将 X 标记为已下线

故障转移

当一个从节点发现所属的主节点进入了已下线状态，从节点将开始对下线主节点进行故障转移，执行步骤：

- 下属的从节点通过选举产生一个节点
 - 被选中的从节点会执行 `SLAVEOF no one` 命令，成为新的主节点
 - 新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己
 - 新的主节点向集群广播一条 PONG 消息，让集群中的其他节点知道当前节点变成了主节点，并且接管了下线节点负责处理的槽
 - 新的主节点开始接收有关的命令请求，故障转移完成
-

选举算法

集群选举新的主节点的规则：

- 集群的配置纪元是一个自增的计数器，初始值为 0
- 当集群里某个节点开始一次故障转移，集群的配置纪元就是增加一
- 每个配置纪元里，集群中每个主节点都有一次投票的机会，而第一个向主节点要求投票的从节点将获得该主节点的投票
- 具有投票权的主节点是必须具有正在处理的槽
- 集群里有 N 个具有投票权的主节点，那么当一个从节点收集到大于等于 $N/2+1$ 张支持票时，从节点就会当选
- 每个配置纪元里，具有投票权的主节点只能投一次票，所以获得一半以上票的节点只会有一个

选举流程：

- 当某个从节点发现正在复制的主节点进入已下线状态时，会向集群广播一条 `CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST` 消息，要求所有收到这条消息、并且具有投票权的主节点向这个从节点投票
- 如果主节点尚未投票给其他从节点，将向要求投票的从节点返回一条 `CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK` 消息，表示这个主节点支持从节点成为新的主节点
- 如果从节点获取到了半数以上的选票，则会当选新的主节点
- 如果一个配置纪元里没有从节点能收集到足够多的支持票，那么集群进入一个新的配置纪元，并再次进行选举，直到选出新的主节点

选举新主节点的方法和选举领头 Sentinel 的方法非常相似，两者都是基于 Raft 算法的领头选举 (leader election) 方法实现的

消息机制

消息结构

集群中的各个节点通过发送和接收消息 (message) 来进行通信，将发送消息的节点称为发送者 (sender)，接收消息的节点称为接收者 (receiver)

节点发送的消息主要有：

- MEET 消息：当发送者接到客户端发送的 CLUSTER MEET 命令时，会向接收者发送 MEET 消息，请求接收者加入到发送者当前所处的集群里
- PING 消息：集群里的每个节点默认每隔一秒钟就会从已知节点列表中随机选出五个，然后对这五个节点中最长时间没有发送过 PING 消息的节点发送 PING，以此来随机检测被选中的节点是否在线

如果节点 A 最后一次收到节点 B 发送的 PONG 消息的时间，距离当前已经超过了节点 A 的 cluster-node-timeout 设置时长的一半，那么 A 也会向 B 发送 PING 消息，防止 A 因为长时间没有随机选中 B 发送 PING，而导致对节点 B 的信息更新滞后

- PONG 消息：当接收者收到 MEET 消息或者 PING 消息时，为了让发送者确认已经成功接收消息，会向发送者返回一条 PONG；节点也可以通过向集群广播 PONG 消息来让集群中的其他节点立即刷新关于这个节点的认识（从升级为主）
- FAIL 消息：当一个主节点 A 判断另一个主节点 B 已经进入 FAIL 状态时，节点 A 会向集群广播一条 B 节点的 FAIL 信息
- PUBLISH 消息：当节点接收到一个 PUBLISH 命令时，节点会执行这个命令并向集群广播一条 PUBLISH 消息，接收到 PUBLISH 消息的节点都会执行相同的 PUBLISH 命令

消息头

节点发送的所有消息都由一个消息头包裹，消息头除了包含消息正文之外，还记录了消息发送者自身的一些信息

消息头：

```
typedef struct clusterMsg {
    // 消息的长度（包括这个消息头的长度和消息正文的长度）
    uint32_t totlen;
    // 消息的类型
    uint16_t type;
    // 消息正文包含的节点信息数量，只在发送MEET、PING、PONG这三种Gossip协议消息时使用
    uint16_t count;

    // 发送者所处的配置纪元
    uint64_t currentEpoch;
```

```

// 如果发送者是一个主节点，那么这里记录的是发送者的配置纪元
// 如果发送者是一个从节点，那么这里记录的是发送者正在复制的主节点的配置纪元
uint64_t configEpoch;

// 发送者的名字(ID)
char sender[REDIS_CLUSTER_NAMELEN];
// 发送者目前的槽指派信息
unsigned char myslots[REDIS_CLUSTER_SLOTS/8];

// 如果发送者是一个从节点，那么这里记录的是发送者正在复制的主节点的名字
// 如果发送者是一个主节点，那么这里记录的是 REDIS_NODE_NULL_NAME，一个 40 字节长值全为
0 的字节数组
char slaveof[REDIS_CLUSTER_NAMELEN];

// 发送者的端口号
uint16_t port;
// 发送者的标识值
uint16_t flags;
// 发送者所处集群的状态
unsigned char state;
// 消息的正文（或者说，内容）
union clusterMsgData data;
}

```

clusterMsg 结构的 currentEpoch、sender、myslots 等属性记录了发送者的节点信息，接收者会根据这些信息在 clusterState.nodes 字典里找到发送者对应的 clusterNode 结构，并对结构进行更新，比如 **传播节点的槽指派信息**

消息正文：

```

union clusterMsgData {
    // MEET、PING、PONG 消息的正文
    struct {
        // 每条 MEET、PING、PONG 消息都包含两个 clusterMsgDataGossip 结构
        clusterMsgDataGossip gossip[1];
    } ping;

    // FAIL 消息的正文
    struct {
        clusterMsgDataFail about;
    } fail;

    // PUBLISH 消息的正文
    struct {
        clusterMsgDataPublish msg;
    } publish;

    // 其他消息正文...
}

```

Gossip

Redis 集群中的各个节点通过 Gossip 协议来交换各自关于不同节点的状态信息，其中 Gossip 协议由 MEET、PING、PONG 消息实现，三种消息使用相同的消息正文，所以节点通过消息头的 type 属性来判断消息的具体类型

发送者发送这三种消息时，会从已知节点列表中**随机选出两个节点**（主从都可以），将两个被选中节点信息保存到两个 Gossip 结构

```
typedef struct clusterMsgDataGossip {
    // 节点的名字
    char nodename[REDIS_CLUSTER_NAMELEN];

    // 最后一次向该节点发送PING消息的时间戳
    uint32_t ping_sent;
    // 最后一次从该节点接收到PONG消息的时间戳
    uint32_t pong_received;

    // 节点的IP地址
    char ip[16];
    // 节点的端口号
    uint16_t port;
    // 节点的标识值
    uint16_t flags;
}
```

当接收者收到消息时，会访问消息正文中的两个数据结构，来进行相关操作

- 如果被选中节点不存在于接收者的已知节点列表，接收者将根据结构中记录的 IP 地址和端口号，与节点进行握手
- 如果存在，根据 Gossip 结构记录的信息对节点所对应的 clusterNode 结构进行更新

FAIL

在集群的节点数量比较大的情况下，使用 Gossip 协议来传播节点的已下线信息会带来一定延迟，因为 Gossip 协议消息通常需要一段时间才能传播至整个集群，所以通过发送 FAIL 消息可以让集群里的所有节点立即知道某个主节点已下线，从而尽快进行其他操作

FAIL 消息的正文由 clusterMsgDataFail 结构表示，该结构只有一个属性，记录了已下线节点的名字

```
typedef struct clusterMsgDataFail {
    char nodename[REDIS_CLUSTER_NAMELEN];
}
```

因为传播下线信息不需要其他属性，所以节省了传播的资源

PUBLISH

当客户端向集群中的某个节点发送命令，接收到 PUBLISH 命令的节点不仅会向 channel 频道发送消息 message，还会向集群广播一条 PUBLISH 消息，所有接收到这条 PUBLISH 消息的节点都会向 channel 频道发送 message 消息，最终集群中所有节点都发了

```
PUBLISH <channel> <message>
```

PUBLISH 消息的正文由 clusterMsgDataPublish 结构表示：

```
typedef struct clusterMsgDataPublish {
    // channel参数的长度
    uint32_t channel_len;
    // message参数的长度
    uint32_t message_len;

    // 定义为8字节只是为了对齐其他消息结构，实际的长度由保存的内容决定
    // bulk_data 的 0 至 channel_len-1 字节保存的是channel参数
    // bulk_data的 channel_len 字节至 channel_len + message_len-1 字节保存的则是
    // message参数
    unsigned char bulk_data[8];
}
```

让集群的所有节点执行相同的 PUBLISH 命令，最简单的方法就是向所有节点广播相同的 PUBLISH 命令，这也是 Redis 复制 PUBLISH 命令时所使用的，但是这种做法并不符合 Redis 集群的各个节点通过发送和接收消息来进行通信的规则

脑裂问题

脑裂指在主从集群中，同时有两个相同的主节点能接收写请求，导致客户端不知道应该往哪个主节点写入数据，最后不同客户端往不同的主节点上写入数据

- 原主节点并没有真的发生故障，由于某些原因无法处理请求（CPU 利用率很高、自身阻塞），无法按时响应心跳请求，被哨兵/集群主节点错误的判断为下线
- 在被判断下线之后，原主库又重新开始处理请求了，哨兵/集群主节点还没有完成主从切换，客户端仍然可以和原主库通信，客户端发送的写操作就会在原主库上写入数据，造成脑裂问题

数据丢失问题：从库一旦升级为新主库，哨兵就会让原主库执行 slave of 命令，和新主库重新进行全量同步，原主库需要清空本地的数据，加载新主库发送的 RDB 文件，所以原主库在主从切换期间保存的新写数据就丢失了

预防脑裂：在主从集群部署时，合理地配置参数 min-slaves-to-write 和 min-slaves-max-lag

- 假设从库有 K 个，可以将 min-slaves-to-write 设置为 K/2+1（如果 K 等于 1，就设为 1）
- 将 min-slaves-max-lag 设置为十几秒（例如 10 ~ 20s）
- 在假故障期间无法响应哨兵发出的心跳测试，无法和从库进行 ACK 确认，并且没有足够的从库，**拒绝客户端的写入**

结构搭建

整体框架：

- 配置服务器（3 主 3 从）
- 建立通信（Meet）
- 分槽（Slot）
- 搭建主从（master-slave）

创建集群 conf 配置文件：

- redis-6501.conf

```
port 6501
dir "/redis/data"
dbfilename "dump-6501.rdb"
cluster-enabled yes
cluster-config-file "cluster-6501.conf"
cluster-node-timeout 5000
```

#其他配置文件参照上面的修改端口即可，内容完全一样

- 服务端启动：

```
redis-server config_file_name
```

- 客户端启动：

```
redis-cli -p 6504 -c
```

cluster 配置：

- 是否启用 cluster，加入 cluster 节点

```
cluster-enabled yes|no
```

- cluster 配置文件名，该文件属于自动生成，仅用于快速查找文件并查询文件内容

```
cluster-config-file filename
```

- 节点服务响应超时时间，用于判定该节点是否下线或切换为从节点

```
cluster-node-timeout milliseconds
```

- master 连接的 slave 最小数量

```
cluster-migration-barrier min_slave_number
```

客户端启动命令：

cluster 节点操作命令 (客户端命令) :

- 查看集群节点信息

```
cluster nodes
```

- 更改 slave 指向新的 master

```
cluster replicate master-id
```

- 发现一个新节点，新增 master

```
cluster meet ip:port
```

- 忽略一个没有 slot 的节点

```
cluster forget server_id
```

- 手动故障转移

```
cluster failover
```

集群操作命令 (Linux) :

- 创建集群

```
redis-cli --cluster create masterhost1:masterport1 masterhost2:masterport2  
masterhost3:masterport3 [masterhostn:masterportn ...] slavehost1:slaveport1  
slavehost2:slaveport2 slavehost3:slaveport3 --cluster-replicas n
```

注意：master 与 slave 的数量要匹配，一个 master 对应 n 个 slave，由最后的参数 n 决定。
master 与 slave 的匹配顺序为第一个 master 与前 n 个 slave 分为一组，形成主从结构

- 添加 master 到当前集群中，连接时可以指定任意现有节点地址与端口

```
redis-cli --cluster add-node new-master-host:new-master-port now-host:now-  
port
```

- 添加 slave

```
redis-cli --cluster add-node new-slave-host:new-slave-port master-  
host:master-port --cluster-slave --cluster-master-id masterid
```

- 删除节点，如果删除的节点是 master，必须保障其中没有槽 slot

```
redis-cli --cluster del-node del-slave-host:del-slave-port del-slave-id
```

- 重新分槽，分槽是从具有槽的 master 中划分一部分给其他 master，过程中不创建新的槽

```
redis-cli --cluster reshard new-master-host:new-master:port --cluster-from
src- master-id1, src-master-id2, src-master-idn --cluster-to target-master-
id -- cluster-slots slots
```

注意：将需要参与分槽的所有 masterid 不分先后顺序添加到参数中，使用 , 分隔，指定目标得到的槽的数量，所有的槽将平均从每个来源的 master 处获取

- 重新分配槽，从具有槽的 master 中分配指定数量的槽到另一个 master 中，常用于清空指定 master 中的槽

```
redis-cli --cluster reshard src-master-host:src-master-port --cluster-from
src- master-id --cluster-to target-master-id --cluster-slots slots --
cluster-yes
```

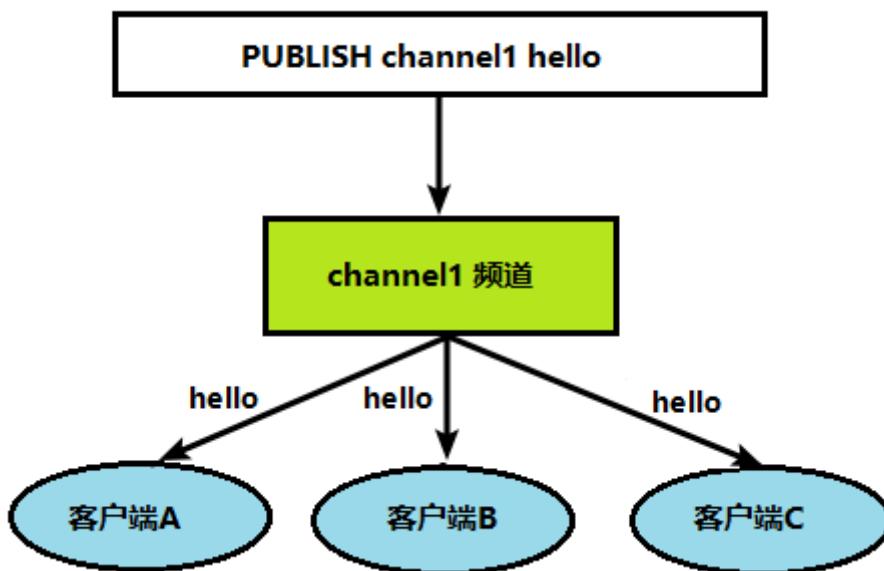
其他操作

发布订阅

基本指令

Redis 发布订阅 (pub/sub) 是一种消息通信模式：发送者 (pub) 发送消息，订阅者 (sub) 接收消息

Redis 客户端可以订阅任意数量的频道，每当有客户端向被订阅的频道发送消息 (message) 时，频道的所有订阅者都会收到消息



操作过程：

- 打开一个客户端订阅 channel1：`SUBSCRIBE channel1`

- 打开另一个客户端，给 channel1 发布消息 hello: `PUBLISH channel1 hello`
- 第一个客户端可以看到发送的消息

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "hello"
```

客户端还可以通过 PSUBSCRIBE 命令订阅一个或多个模式，每当有其他客户端向某个频道发送消息时，消息不仅会被发送给这个频道的所有订阅者，还会被发送给所有与这个频道相匹配的模式的订阅者，比如 `PSUBSCRIBE channel*` 订阅模式，与 channel1 匹配

注意：发布的消息没有持久化，所以订阅的客户端只能收到订阅后发布的消息

频道操作

Redis 将所有频道的订阅关系都保存在服务器状态的 pubsub_channels 字典里，键是某个被订阅的频道，值是一个记录所有订阅这个频道的客户端链表

```
struct redisServer {
    // 保存所有频道的订阅关系,
    dict *pubsub_channels;
}
```

客户端执行 SUBSCRIBE 命令订阅某个或某些频道，服务器会将客户端与频道进行关联：

- 频道已经存在，直接将客户端添加到链表末尾
- 频道还未有任何订阅者，在字典中为频道创建一个键值对，再将客户端添加到链表

UNSUBSCRIBE 命令用来退订某个频道，服务器将从 pubsub_channels 中解除客户端与被退订频道之间的关联

模式操作

Redis 服务器将所有模式的订阅关系都保存在服务器状态的 pubsub_patterns 属性里

```

struct redisServer {
    // 保存所有模式订阅关系，链表中每个节点是一个 pubsubPattern
    list *pubsub_patterns;
}

typedef struct pubsubPattern {
    // 订阅的客户端
    redisClient *client;
    // 被订阅的模式，比如 channel
    rObj *pattern;
}

```

客户端执行 PSUBSCRIBE 命令订阅某个模式，服务器会新建一个 pubsubPattern 结构并赋值，放入 pubsub_patterns 链表结尾

模式的退订命令 PUNSUBSCRIBE 是订阅命令的反操作，服务器在 pubsub_patterns 链表中查找并删除对应的结构

发送消息

Redis 客户端执行 `PUBLISH <channel> <message>` 命令将消息 message 发送给频道 channel，服务器会执行：

- 在 pubsub_channels 字典里找到频道 channel 的订阅者名单，将消息 message 发送给所有订阅者
- 遍历整个 pubsub_patterns 链表，查找与 channel 频道相**匹配的模式**，并将消息发送给所有订阅了这些模式的客户端

```

// 如果频道和模式相匹配
if match(channel, pubsubPattern.pattern) {
    // 将消息发送给订阅该模式的客户端
    send_message(pubsubPattern.client, message);
}

```

查看信息

PUBSUB 命令用来查看频道或者模式的相关信息

`PUBSUB CHANNELS [pattern]` 返回服务器当前被订阅的频道，其中 pattern 参数是可选的

- 如果不给定 pattern 参数，那么命令返回服务器当前被订阅的所有频道
- 如果给定 pattern 参数，那么命令返回服务器当前被订阅的频道中与 pattern 模式相匹配的频道

`PUBSUB NUMSUB [channel1-1 channel1-2 ... channel1-n]` 命令接受任意多个频道作为输入参数，并返回这些频道的订阅者数量

`PUBSUB NUMPAT` 命令用于返回服务器当前被订阅模式的数量

ACL 指令

Redis ACL 是 Access Control List (访问控制列表) 的缩写，该功能允许根据可以执行的命令和可以访问的键来限制某些连接

```
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
    ↓      ↓      ↓      ↓      ↓
用户名  是否启用 (on/off) 密码 (没密码 nopass) 可操作的key 可执行的命令
```

- `acl cat`: 查看添加权限指令类别
- `acl whoami`: 查看当前用户
- `acl setuser username on >password ~cached:*` +get: 设置有用户名、密码、ACL 权限 (只能 get)

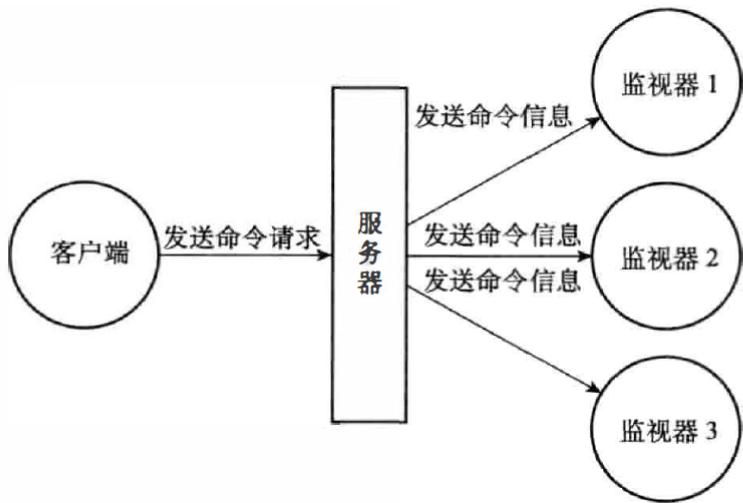
监视器

MONITOR 命令，可以将客户端变为一个监视器，实时地接收并打印出服务器当前处理的命令请求的相关信息

```
// 实现原理
def MONITOR():
    // 打开客户端的监视器标志
    client.flags |= REDIS_MONITOR

    // 将客户端添加到服务器状态的 redisServer.monitors 链表的末尾
    server.monitors.append(client)
    // 向客户端返回 ok
    send_reply("OK")
```

服务器每次处理命令请求都会调用 `replicationFeedMonitors` 函数，函数将被处理的命令请求的相关信息发送给各个监视器



```

redis> MONITOR
OK
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Banana" "Cherry"
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"

```

批处理

Redis 的管道 Pipeline 机制可以一次处理多条指令

- Pipeline 中的多条命令非原子性，因为在向管道内添加命令时，其他客户端的发送的命令仍然在执行
- 原生批命令 (MSET 等) 是服务端实现，而 Pipeline 需要服务端与客户端共同完成

使用 Pipeline 封装的命令数量不能太多，数据量过大将增加客户端的等待时间，造成网络阻塞，Jedis 中的 Pipeline 使用方式：

```

// 创建管道
Pipeline pipeline = jedis.pipelined();
for (int i = 1; i <= 100000; i++) {
    // 放入命令到管道
    pipeline.set("key_" + i, "value_" + i);
    if (i % 1000 == 0) {
        // 每放入1000条命令，批量执行
        pipeline.sync();
    }
}

```

集群下模式下，批处理命令的多个 key 必须落在一个插槽中，否则就会导致执行失败，N 条批处理命令的优化方式：

- 串行命令：for 循环遍历，依次执行每个命令
- 串行 slot：在客户端计算每个 key 的 slot，将 slot 一致的分为一组，每组都利用 Pipeline 批处理，串行执行各组命令
- 并行 slot：在客户端计算每个 key 的 slot，将 slot 一致的分为一组，每组都利用 Pipeline 批处理，**并行执行各组命令**
- hash_tag：将所有 key 设置相同的 hash_tag，则所有 key 的 slot 一定相同

| | 耗时 | 优点 | 缺点 |
|----------|---|------------|------------------|
| 串行命令 | N 次网络耗时 + N 次命令耗时 | 实现简单 | 耗时久 |
| 串行 slot | m 次网络耗时 + N 次命令耗时， $m = \text{key}$ 的 slot 个数 | 耗时较短 | 实现稍复杂 |
| 并行 slot | 1 次网络耗时 + N 次命令耗时 | 耗时非常短 | 实现复杂 |
| hash_tag | 1 次网络耗时 + N 次命令耗时 | 耗时非常短、实现简单 | 容易出现 数据倾斜 |

解决方案

缓存方案

缓存模式

旁路缓存

缓存本质：弥补 CPU 的高算力和 IO 的慢读写之间巨大的鸿沟

旁路缓存模式 Cache Aside Pattern 是平时使用比较多的一个缓存读写模式，比较适合读请求比较多的场景

Cache Aside Pattern 中服务端需要同时维系 DB 和 cache，并且是以 DB 的结果为准

- 写操作：先更新 DB，然后直接删除 cache
- 读操作：从 cache 中读取数据，读取到就直接返回；读取不到就从 DB 中读取数据返回，并放到 cache

时序导致的不一致问题：

- 在写数据的过程中，不能先删除 cache 再更新 DB，因为会造成缓存的不一致。比如请求 1 先写数据 A，请求 2 随后读数据 A，当请求 1 删除 cache 后，请求 2 直接读取了 DB，此时请求 1 还没写入 DB（延迟双删）

- 在写数据的过程中，先更新 DB 再删除 cache 也会出现问题，但是概率很小，因为缓存的写入速度非常快

旁路缓存的缺点：

- 首次请求数据一定不在 cache 的问题，一般采用缓存预热的方法，将热点数据可以提前放入 cache 中
- 写操作比较频繁的话导致 cache 中的数据会被频繁被删除，影响缓存命中率

删除缓存而不是更新缓存的原因：每次更新数据库都更新缓存，造成无效写操作较多（懒惰加载，需要的时候再放入缓存）

读写穿透

读写穿透模式 Read/Write Through Pattern：服务端把 cache 视为主要数据存储，从中读取数据并将数据写入其中，cache 负责将此数据同步写入 DB，从而减轻了应用程序的职责

- 写操作：先查 cache，cache 中不存在，直接更新 DB；cache 中存在则先更新 cache，然后 cache 服务更新 DB（同步更新 cache 和 DB）
- 读操作：从 cache 中读取数据，读取到就直接返回；读取不到先从 DB 加载，写入到 cache 后返回响应

Read-Through Pattern 实际只是在 Cache-Aside Pattern 之上进行了封装。在 Cache-Aside Pattern 下，发生读请求的时候，如果 cache 中不存在对应的数据，是由客户端负责把数据写入 cache，而 Read Through Pattern 则是 cache 服务自己来写入缓存的，对客户端是透明的

Read-Through Pattern 也存在首次不命中的问题，采用缓存预热解决

异步缓存

异步缓存写入 Write Behind Pattern 由 cache 服务来负责 cache 和 DB 的读写，对比读写穿透不同的是 Write Behind Caching 是只更新缓存，不直接更新 DB，改为**异步批量**的方式来更新 DB，可以减小写的成本

缺点：这种模式对数据一致性没有高要求，可能出现 cache 还没异步更新 DB，服务就挂掉了

应用：

- DB 的写性能非常高，适合一些数据经常变化又对数据一致性要求不高的场景，比如浏览量、点赞量
 - MySQL 的 InnoDB Buffer Pool 机制用到了这种策略
-

缓存一致

使用缓存代表不需要强一致性，只需要最终一致性

缓存不一致的方法：

- 数据库和缓存数据强一致场景：
 - 同步双写：更新 DB 时同样更新 cache，保证在一个事务中，通过加锁来保证更新 cache 时不存在线程安全问题
 - 延迟双删：先淘汰缓存再写数据库，休眠 1 秒再次淘汰缓存，可以将 1 秒内造成的缓存脏数据再次删除
 - 异步通知：
 - 基于 MQ 的异步通知：对数据的修改后，代码需要发送一条消息到 MQ 中，缓存服务监听 MQ 消息
 - Canal 订阅 MySQL binlog 的变更上报给 Kafka，系统监听 Kafka 消息触发缓存失效，或者直接将变更发送到处理服务，**没有任何代码侵入**
- 低耦合，可以同时通知多个缓存服务，但是时效性一般，可能存在中间不一致状态
- 低一致性场景：
 - 更新 DB 的时候同样更新 cache，但是给缓存加一个比较短的过期时间，这样就可以保证即使数据不一致影响也比较小
 - 使用 Redis 自带的内存淘汰机制

缓存问题

缓存预热

场景：宕机，服务器启动后迅速宕机

问题排查：

1. 请求数量较高，大量的请求过来之后都需要去从缓存中获取数据，但是缓存中又没有，此时从数据库中查找数据然后将数据再存入缓存，造成了短期内对 redis 的高强度操作从而导致问题
2. 主从之间数据吞吐量较大，数据同步操作频度较高

解决方案：

- 前置准备工作：
 1. 日常例行统计数据访问记录，统计访问频度较高的热点数据
 2. 利用 LRU 数据删除策略，构建数据留存队列例如：storm 与 kafka 配合
- 准备工作：
 1. 将统计结果中的数据分类，根据级别，redis 优先加载级别较高的热点数据
 2. 利用分布式多服务器同时进行数据读取，提速数据加载过程
 3. 热点数据主从同时预热
- 实施：
 4. 使用脚本程序固定触发数据预热过程

5. 如果条件允许，使用了 CDN（内容分发网络），效果会更好

总的来说：缓存预热就是系统启动前，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据

缓存雪崩

场景：数据库服务器崩溃，一连串的问题会随之而来

问题排查：在一个较短的时间内，**缓存中较多的 key 集中过期**，此周期内请求访问过期的数据 Redis 未命中，Redis 向数据库获取数据，数据库同时收到大量的请求无法及时处理。

解决方案：

1. 加锁，慎用
2. 设置热点数据永远不过期，如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中
3. 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生
4. 构建**多级缓存**架构，Nginx 缓存 + Redis 缓存 + ehcache 缓存
5. 灾难预警机制，监控 Redis 服务器性能指标，CPU 使用率、内存容量、平均响应时间、线程数
6. **限流、降级**：短时间范围内牺牲一些客户体验，限制一部分请求访问，降低应用服务器压力，待业务低速运转后再逐步放开访问

总的来说：缓存雪崩就是瞬间过期数据量太大，导致对数据库服务器造成压力。如能够有效避免过期时间集中，可以有效解决雪崩现象的出现（约 40%），配合其他策略一起使用，并监控服务器的运行数据，根据运行记录做快速调整。

缓存击穿

缓存击穿也叫热点 Key 问题

1. **Redis 中某个 key 过期，该 key 访问量巨大**
2. 多个数据请求从服务器直接压到 Redis 后，均未命中
3. Redis 在短时间内发起了大量对数据库中同一数据的访问

解决方案：

1. 预先设定：以电商为例，每个商家根据店铺等级，指定若干款主打商品，在购物节期间，加大此类信息 key 的过期时长 注意：购物节不仅仅指当天，以及后续若干天，访问峰值呈现逐渐降低的趋势
2. 现场调整：监控访问量，对自然流量激增的数据**延长过期时间或设置为永久性 key**
3. 后台刷新数据：启动定时任务，高峰期来临之前，刷新数据有效期，确保不丢失
4. **二级缓存**：设置不同的失效时间，保障不会被同时淘汰就行
5. 加锁：分布式锁，防止被击穿，但是要注意也是性能瓶颈，慎重

总的来说：缓存击穿就是单个高热数据过期的瞬间，数据访问量较大，未命中 Redis 后，发起了大量对同一数据的数据库访问，导致对数据库服务器造成压力。应对策略应该在业务数据分析与预防方面进行，配合运行监控测试与即时调整策略，毕竟单个 key 的过期监控难度较高，配合雪崩处理策略即可

缓存穿透

场景：系统平稳运行过程中，应用服务器流量随时间增量较大，Redis 服务器命中率随时间逐步降低，Redis 内存平稳，内存无压力，Redis 服务器 CPU 占用激增，数据库服务器压力激增，数据库崩溃

问题排查：

1. Redis 中大面积出现未命中
2. 出现非正常 URL 访问

问题分析：

- 访问了不存在的数据，跳过了 Redis 缓存，数据库页查询不到对应数据
- Redis 获取到 null 数据未进行持久化，直接返回
- 出现黑客攻击服务器

解决方案：

1. 缓存 null：对查询结果为 null 的数据进行缓存，设定短时限，例如 30-60 秒，最高 5 分钟
2. 白名单策略：提前预热各种分类数据 id 对应的 bitmaps，id 作为 bitmaps 的 offset，相当于设置了数据白名单。当加载正常数据时放行，加载异常数据时直接拦截（效率偏低），也可以使用布隆过滤器（有关布隆过滤器的命中问题对当前状况可以忽略）
3. 实时监控：实时监控 Redis 命中率（业务正常范围时，通常会有一个波动值）与 null 数据的占比
 - 非活动时段波动：通常检测 3-5 倍，超过 5 倍纳入重点排查对象
 - 活动时段波动：通常检测 10-50 倍，超过 50 倍纳入重点排查对象

根据倍数不同，启动不同的排查流程。然后使用黑名单进行防控

4. key 加密：临时启动防灾业务 key，对 key 进行业务层传输加密服务，设定校验程序，过来的 key 校验；例如每天随机分配 60 个加密串，挑选 2 到 3 个，混淆到页面数据 id 中，发现访问 key 不满足规则，驳回数据访问

总的来说：缓存击穿是指访问了不存在的数据，跳过了合法数据的 Redis 数据缓存阶段，**每次访问数据库**，导致对数据库服务器造成压力。通常此类数据的出现量是一个较低的值，当出现此类情况以毒攻毒，并及时报警。无论是黑名单还是白名单，都是对整体系统的压力，警报解除后尽快移除

参考视频：<https://www.bilibili.com/video/BV15y4y1r7X3>

Key 设计

大 Key：通常以 Key 的大小和 Key 中成员的数量来综合判定，引发的问题：

- 客户端执行命令的时长变慢
- Redis 内存达到 maxmemory 定义的上限引发操作阻塞或重要的 Key 被逐出，甚至引发内存溢出 (OOM)
- 集群架构下，某个数据分片的内存使用率远超其他数据分片，使**数据分片的内存资源不均衡**
- 对大 Key 执行读请求，会使 Redis 实例的带宽使用率被占满，导致自身服务变慢，同时易波及相关的服务
- 对大 Key 执行删除操作，会造成主库较长时间的阻塞，进而可能引发同步中断或主从切换

热 Key：通常以其接收到的 Key 被请求频率来判定，引发的问题：

- 占用大量的 CPU 资源，影响其他请求并导致整体性能降低
- 分布式集群架构下，产生**访问倾斜**，即某个数据分片被大量访问，而其他数据分片处于空闲状态，可能引起该数据分片的连接数被耗尽，新的连接建立请求被拒绝等问题
- 在抢购或秒杀场景下，可能因商品对应库存 Key 的请求量过大，超出 Redis 处理能力造成超卖
- 热 Key 的请求压力数量超出 Redis 的承受能力易造成缓存击穿，即大量请求将被直接指向后端的存储层，导致存储访问量激增甚至宕机，从而影响其他业务

热 Key 分类两种，治理方式如下：

- 一种是单一数据，比如秒杀场景，假设总量 10000 可以拆为多个 Key 进行访问，每次对请求进行路由到不同的 Key 访问，保证最终一致性，但是会出现访问不同 Key 产生的剩余量是不同的，这时可以通过前端进行 Mock 假数据
- 一种是多数据集合，比如进行 ID 过滤，这时可以添加本地 LRU 缓存，减少对热 Key 的访问

参考文档：https://help.aliyun.com/document_detail/353223.html

慢查询

确认服务和 Redis 之间的链路是否正常，排除网络原因后进行 Redis 的排查：

- 使用复杂度过高的命令
- 操作大 key，分配内存和释放内存会比较耗时
- key 集中过期，导致定时任务需要更长的时间去清理
- 实例内存达到上限，每次写入新的数据之前，Redis 必须先从实例中踢出一部分数据

参考文章：<https://www.cnblogs.com/traditional/p/15633919.html> (非常好)

Java

JDBC

概述

JDBC (Java DataBase Connectivity, Java 数据库连接) 是一种用于执行 SQL 语句的 Java API，可以为多种关系型数据库提供统一访问，是由一组用 Java 语言编写的类和接口组成的。

JDBC 是 Java 官方提供的一套规范（接口），用于帮助开发人员快速实现不同关系型数据库的连接

功能类

DriverManager

DriverManager：驱动管理对象

- 注册驱动：

- 注册给定的驱动：`public static void registerDriver(Driver driver)`
- 代码实现语法：`Class.forName("com.mysql.jdbc.Driver")`
- `com.mysql.jdbc.Driver` 中存在静态代码块

```
static {
    try {
        DriverManager.registerDriver(new Driver());
    } catch (SQLException var1) {
        throw new RuntimeException("Can't register driver!");
    }
}
```

- 不需要通过 `DriverManager` 调用静态方法 `registerDriver`，因为 `Driver` 类被使用，则自动执行静态代码块完成注册驱动
- jar 包中 `META-INF` 目录下存在一个 `java.sql.Driver` 配置文件，文件中指定了 `com.mysql.jdbc.Driver`
- 获取数据库连接并返回连接对象：
方法：`public static Connection getConnection(String url, String user, String password)`
 - `url`: 指定连接的路径，语法为 `jdbc:mysql://ip地址(域名):端口号/数据库名称`
 - `user`: 用户名
 - `password`: 密码

Connection

Connection: 数据库连接对象

- 获取执行者对象
 - 获得普通执行者对象: `Statement createStatement()`
 - 获得预编译执行者对象: `PreparedStatement prepareStatement(String sql)`
 - 管理事务
 - 开启事务: `setAutoCommit(boolean autoCommit)`, false 开启事务, true 自动提交模式 (默认)
 - 提交事务: `void commit()`
 - 回滚事务: `void rollback()`
 - 释放资源
 - 释放此 Connection 对象的数据库和 JDBC 资源: `void close()`
-

Statement

Statement: 执行 sql 语句的对象

- 执行 DML 语句: `int executeUpdate(String sql)`
 - 返回值 int: 返回影响的行数
 - 参数 sql: 可以执行 insert、update、delete 语句
 - 执行 DQL 语句: `ResultSet executeQuery(String sql)`
 - 返回值 ResultSet: 封装查询的结果
 - 参数 sql: 可以执行 select 语句
 - 释放资源
 - 释放此 Statement 对象的数据库和 JDBC 资源: `void close()`
-

ResultSet

ResultSet: 结果集对象, ResultSet 对象维护了一个游标, 指向当前的数据行, 初始在第一行

- 判断结果集中是否有数据: `boolean next()`
 - 有数据返回 true, 并将索引向下移动一行
 - 没有数据返回 false
- 获取结果集中当前行的数据: `xxx getXxx("列名")`
 - XXX 代表数据类型 (要获取某列数据, 这一列的数据类型)
 - 例如: `String getString("name"); int getInt("age");`
- 释放资源
 - 释放 ResultSet 对象的数据库和 JDBC 资源: `void close()`

代码实现

数据准备

```
-- 创建db14数据库
CREATE DATABASE db14;

-- 使用db14数据库
USE db14;

-- 创建student表
CREATE TABLE student(
    sid INT PRIMARY KEY AUTO_INCREMENT, -- 学生id
    NAME VARCHAR(20), -- 学生姓名
    age INT, -- 学生年龄
    birthday DATE, -- 学生生日
);

-- 添加数据
INSERT INTO student VALUES (NULL, '张三', 23, '1999-09-23'), (NULL, '李四', 24, '1998-08-10'),
    (NULL, '王五', 25, '1996-06-06'), (NULL, '赵六', 26, '1994-10-20');
```

JDBC 连接代码：

```
public class JDBCdemo01 {
    public static void main(String[] args) throws Exception{
        //1.导入jar包
        //2.注册驱动
        Class.forName("com.mysql.jdbc.Driver");

        //3.获取连接
        Connection con =
        DriverManager.getConnection("jdbc:mysql://192.168.2.184:3306/db2", "root", "123456");
        
        //4.获取执行者对象
        Statement stat = con.createStatement();

        //5.执行sql语句，并且接收结果
        String sql = "SELECT * FROM user";
        ResultSet rs = stat.executeQuery(sql);

        //6.处理结果
        while(rs.next()) {
            System.out.println(rs.getInt("id") + "\t" + rs.getString("name"));
        }

        //7.释放资源
        con.close();
    }
}
```

```
    stat.close();
    con.close();
}
}
```

注入攻击

攻击演示

SQL 注入攻击演示

- 在登录界面，输入一个错误的用户名或密码，也可以登录成功



- 原理：我们在密码处输入的所有内容，都应该认为是密码的组成，但是 Statement 对象在执行 SQL 语句时，将一部分内容当做查询条件来执行

```
SELECT * FROM user WHERE loginname='aaa' AND password='aaa' OR '1'='1';
```

攻击解决

PreparedStatement：预编译 sql 语句的执行者对象，继承 `PreparedStatement extends Statement`

- 在执行 sql 语句之前，将 sql 语句进行提前编译，**明确 sql 语句的格式**，剩余的内容都会认为是参数
- sql 语句中的参数使用 ? 作为**占位符**

为 ? 占位符赋值的方法：`setXXX(int parameterIndex, xxx data)`

- 参数1：? 的位置编号（编号从 1 开始）
- 参数2：? 的实际参数

```
String sql = "SELECT * FROM user WHERE loginname=? AND password=?";
pst = con.prepareStatement(sql);
pst.setString(1, loginName);
pst.setString(2, password);
```

执行 sql 语句的方法

- 执行 insert、update、delete 语句：`int executeUpdate()`
- 执行 select 语句：`ResultSet executeQuery()`

连接池

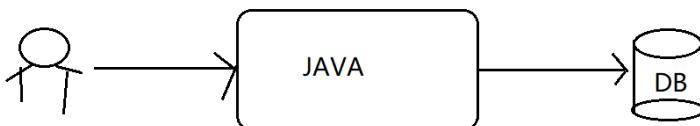
概念

数据库连接背景：数据库连接是一种关键的、有限的、昂贵的资源，这一点在多用户的网页应用程序中体现得尤为突出。对数据库连接的管理能显著影响到整个应用程序的伸缩性和健壮性，影响到程序的性能指标。

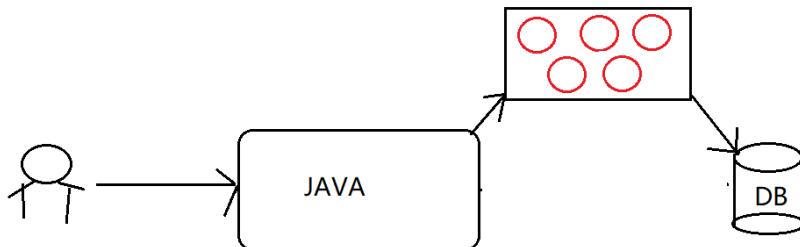
数据库连接池：**数据库连接池负责分配、管理和释放数据库连接**，它允许应用程序**重复使用**一个现有的数据库连接，而不是再重新建立一个，这项技术能明显提高对数据库操作的性能。

数据库连接池原理

之前的程序：来一个访问就会创建一个连接，使用完了关闭连接。频繁的创建连接和关闭连接是非常耗时的



优化后的程序：提前准备一些数据库连接，使用的时候从池中获取，用完后重新归还给池中



归还连接

使用动态代理的方式来改进

自定义数据库连接池类：

```
public class MyDataSource implements DataSource {
    //1.准备一个容器。用于保存多个数据库连接对象
    private static List<Connection> pool = Collections.synchronizedList(new
    ArrayList<>());

    //2.定义静态代码块，获取多个连接对象保存到容器中
    static{
        for(int i = 1; i <= 10; i++) {
            Connection con = JDBCUtils.getConnection();
            pool.add(con);
        }
    }

    //3.提供一个获取连接池大小的方法
    public int getSize() {
        return pool.size();
    }

    //动态代理方式
    @Override
    public Connection getConnection() throws SQLException {
        if(pool.size() > 0) {
            Connection con = pool.remove(0);
            return con;
        } else {
            return JDBCUtils.getConnection();
        }
    }
}
```

```

Connection proxyCon = (Connection) Proxy.newProxyInstance(
    con.getClass().getClassLoader(), new Class[]{Connection.class},
    new InvocationHandler() {
        /*
         * 执行Connection实现类连接对象所有的方法都会经过invoke
         * 如果是close方法，归还连接
         * 如果不是，直接执行连接对象原有的功能即可
         */
        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
            if(method.getName().equals("close")) {
                //归还连接
                pool.add(con);
                return null;
            }else {
                return method.invoke(con,args);
            }
        }
    );
    return proxyCon;
}else {
    throw new RuntimeException("连接数量已用尽");
}
}
}
}

```

开源项目

C3P0

使用 C3P0 连接池：

- 配置文件名称：c3p0-config.xml，必须放在 src 目录下

```

<c3p0-config>
    <!-- 使用默认的配置读取连接池对象 -->
    <default-config>
        <!-- 连接参数 -->
        <property name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="jdbcUrl">jdbc:mysql://192.168.2.184:3306/db14</property>
        <property name="user">root</property>
        <property name="password">123456</property>

        <!-- 连接池参数 -->
        <!-- 初始化数量-->
        <property name="initialPoolSize">5</property>
        <!--最大连接数量-->
        <property name="maxPoolSize">10</property>
        <!--超时时间 3000ms-->
        <property name="checkoutTimeout">3000</property>
    </default-config>
</c3p0-config>

```

```
</default-config>

<named-config name="otherc3p0">
    <!-- 连接参数 -->
    <!-- 连接池参数 -->
</named-config>
</c3p0-config>
```

- 代码演示

```
public class C3P0Test1 {
    public static void main(String[] args) throws Exception{
        //1.创建c3p0的数据库连接池对象
        DataSource dataSource = new ComboPooledDataSource();

        //2.通过连接池对象获取数据库连接
        Connection con = dataSource.getConnection();

        //3.执行操作
        String sql = "SELECT * FROM student";
        PreparedStatement pst = con.prepareStatement(sql);

        //4.执行sql语句，接收结果集
        ResultSet rs = pst.executeQuery();

        //5.处理结果集
        while(rs.next()) {
            System.out.println(rs.getInt("sid") + "\t" +
rs.getString("name") + "\t" + rs.getInt("age") + "\t" +
rs.getDate("birthday"));
        }

        //6.释放资源
        rs.close();    pst.close();    con.close();
    }
}
```

Druid

Druid 连接池：

- 配置文件：druid.properties，必须放在 src 目录下

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://192.168.2.184:3306/db14
username=root
password=123456
initialSize=5
maxActive=10
maxWait=3000
```

- 代码演示

```
public class DruidTest1 {  
    public static void main(String[] args) throws Exception{  
        //获取配置文件的流对象  
        InputStream is =  
DruidTest1.class.getClassLoader().getResourceAsStream("druid.properties");  
  
        //1.通过Properties集合，加载配置文件  
        Properties prop = new Properties();  
        prop.load(is);  
  
        //2.通过Druid连接池工厂类获取数据库连接池对象  
        DataSource dataSource =  
DruidDataSourceFactory.createDataSource(prop);  
  
        //3.通过连接池对象获取数据库连接进行使用  
        Connection con = dataSource.getConnection();  
  
        //4.执行sql语句，接收结果集  
        String sql = "SELECT * FROM student";  
        PreparedStatement pst = con.prepareStatement(sql);  
        ResultSet rs = pst.executeQuery();  
  
        //5.处理结果集  
        while(rs.next()) {  
            System.out.println(rs.getInt("sid") + "\t" +  
rs.getString("name") + "\t" + rs.getInt("age") + "\t" +  
rs.getDate("birthday"));  
        }  
  
        //6.释放资源  
        rs.close();    pst.close();    con.close();  
    }  
}
```

基本使用

Jedis 用于 Java 语言连接 Redis 服务，并提供对应的操作 API

- jar 包导入

下载地址: <https://mvnrepository.com/artifact/redis.clients/jedis>

基于 maven:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

- 客户端连接 Redis: API 文档 <http://xetorthio.github.io/jedis/>

连接 redis: `Jedis jedis = new Jedis("192.168.0.185", 6379)`

操作 redis: `jedis.set("name", "seazean"); jedis.get("name")`

关闭 redis: `jedis.close()`

代码实现:

```
public class JedisTest {
    public static void main(String[] args) {
        //1. 获取连接对象
        Jedis jedis = new Jedis("192.168.2.185", 6379);
        //2. 执行操作
        jedis.set("age", "39");
        String hello = jedis.get("hello");
        System.out.println(hello);
        jedis.lpush("list1", "a", "b", "c", "d");
        List<String> list1 = jedis.lrange("list1", 0, -1);
        for (String s:list1) {
            System.out.println(s);
        }
        jedis.sadd("set1", "abc", "abc", "def", "poi", "cba");
        Long len = jedis.scard("set1");
        System.out.println(len);
        //3. 关闭连接
        jedis.close();
    }
}
```

工具类

连接池对象：

- JedisPool: Jedis 提供的连接池技术
- poolConfig: 连接池配置对象
- host: Redis 服务地址
- port: Redis 服务端口号

JedisPool 的构造器如下：

```
public JedisPool(GenericObjectPoolConfig poolConfig, String host, int port) {  
    this(poolConfig, host, port, 2000, (String)null, 0, (String)null);  
}
```

- 创建配置文件 redis.properties

```
redis.maxTotal=50  
redis.maxIdle=10  
redis.host=192.168.2.185  
redis.port=6379
```

- 工具类：

```
public class JedisUtils {  
    private static int maxTotal;  
    private static int maxIdle;  
    private static String host;  
    private static int port;  
    private static JedisPoolConfig jpc;  
    private static JedisPool jp;  
  
    static {  
        ResourceBundle bundle = ResourceBundle.getBundle("redis");  
        //最大连接数  
        maxTotal = Integer.parseInt(bundle.getString("redis.maxTotal"));  
        //活动连接数  
        maxIdle = Integer.parseInt(bundle.getString("redis.maxIdle"));  
        host = bundle.getString("redis.host");  
        port = Integer.parseInt(bundle.getString("redis.port"));  
  
        //Jedis连接配置  
        jpc = new JedisPoolConfig();  
        jpc.setMaxTotal(maxTotal);  
        jpc.setMaxIdle(maxIdle);  
        //连接池对象  
        jp = new JedisPool(jpc, host, port);  
    }  
  
    //对外访问接口，提供jedis连接对象，连接从连接池获取  
    public static Jedis getJedis() {  
        return jp.getResource();  
    }  
}
```

