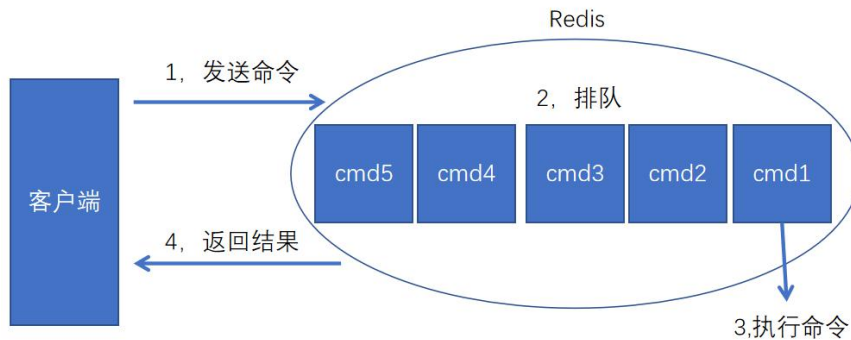


Redis 第 4 节课笔记

1, 什么是 Redis 慢查询?

与 mysql 一样:当执行时间超过极大值时, 会将发生时间 耗时 命令记录.

通信流程如下:



Redis 的所有指令全部会存放到队列, 由单线程按顺序获取并执行指令, 如果某个指令执行很慢, 会出现阻塞, 以上图可以得出: Redis 的慢查询指的是在执行第 3 个操作的时候发生的.

2, Redis 如何设置?

- 动态设置 6379:> `config set slowlog-log-slower-than 10000` //10 毫秒
使用 `config set` 完后,若想将配置持久化保存到 `redis.conf`, 要执行 `config rewrite`
- `redis.conf` 修改: 找到 `slowlog-log-slower-than 10000`, 修改保存即可
注意: `slowlog-log-slower-than =0` 记录所有命令 -1 命令都不记录

3, Redis 慢查询原理

慢查询记录也是存在队列里的, `slow-max-len` 存放的记录最大条数, 比如设置的 `slow-max-len=10`, 当有第 11 条慢查询命令插入时, 队列的第一条命令就会出列, 第 11 条入列到慢查询队列中, 可以 `config set` 动态设置, 也可以修改 `redis.conf` 完成配置

4, Redis 慢查询的命令

获取队列里慢查询的命令: `slowlog get`

获取慢查询列表当前的长度: `slowlog len` //以上只有 1 条慢查询, 返回 1;

- 对慢查询列表清理 (重置): `slowlog reset` //再查 `slowlog len` 此时返回 0 清空;
- 对于线上 `slow-max-len` 配置的建议: 线上可加大 `slow-max-len` 的值, 记录慢查询存长命令时 `redis` 会做截断, 不会占用大量内存, 线上可设置 1000 以上
- 对于线上 `slowlog-log-slower-than` 配置的建议: 默认为 10 毫秒, 根据 `redis` 并发量来调整, 对于高并发比建议为 1 毫秒
- 慢查询是先进先出的队列, 访问日志记录出列丢失, 需定期执行 `slowlog get`, 将结果存储到其它设备中 (如 `mysql`)

5, Redis 性能测试工具如何使用?

- A、`redis-benchmark -h 192.168.42.111 -p 6379 -c 100 -n 10000`
//100 个并发连接, 10000 个请求, 检测服务器性能

B、redis-benchmark -h 192.168.42.111 -p 6379 -q -d 100

//测试存取大小为 100 字节的数据包的性能

C、redis-benchmark -h 192.168.42.111 -p 6379 -t set,get -n 100000 -q

//只测试 set,lpush 操作的性能

D 、 redis-benchmark -h 192.168.42.111 -p 6379 -n 100000 -q script load
"redis.call('set','foo','bar')"

//只测试某些数值存取的性能

6. 什么是 Resp 协议?

Redis 服务器与客户端通过 RESP (REdis Serialization Protocol) 协议通信。

主要以下特点: 容易实现,解析快,人类可读.

RESP 底层采用的是 TCP 的连接方式,通过 tcp 进行数据传输,然后根据解析规则解析相应信息,完成交互。

我们可以测试下,首先运行一个 serverSocket 监听 6379,来接收 redis 客户端的请求信息,实现如下

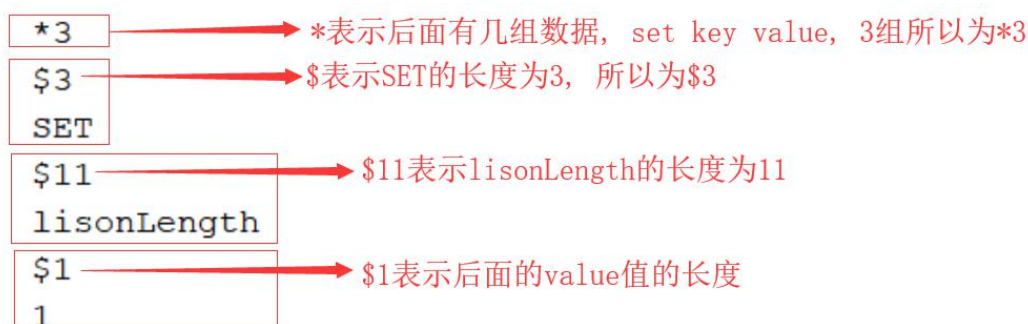
服务端程序如下:

```
//假的Redis
public class ServerRedis {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(port: 6379);
            Socket rec = server.accept();
            byte[] result = new byte[2048];
            rec.getInputStream().read(result);
            System.out.println(new String(result));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

客户端程序如下:

```
public class JedisTest {
    public static void main(String[] args) {
        Jedis jedis = new Jedis(host: "127.0.0.1", port: 6379);
        jedis.set("lisonLength", "1");
        jedis.close();
    }
}
```

测试发现, 服务端打印的信息如下:



这就是 Resp 协议的结构.

7. 将你现有表数据快速存放到 Redis

流程如下:

- A), 使用用户名和密码登陆连接数据库
- B), 登陆成功后执行 order.sql 的 select 语句得到查询结果集 result
- C), 使用密码登陆 Redis
- D), Redis 登陆成功后, 使用 PIPE 管道将 result 导入 Redis.

操作指令如下:

```
mysql -utest -ptest stress --default-character-set=utf8 --skip-column-names --raw < order.sql |  
redis-cli -h 192.168.42.111 -p 6379 -a 12345678 --pipe
```

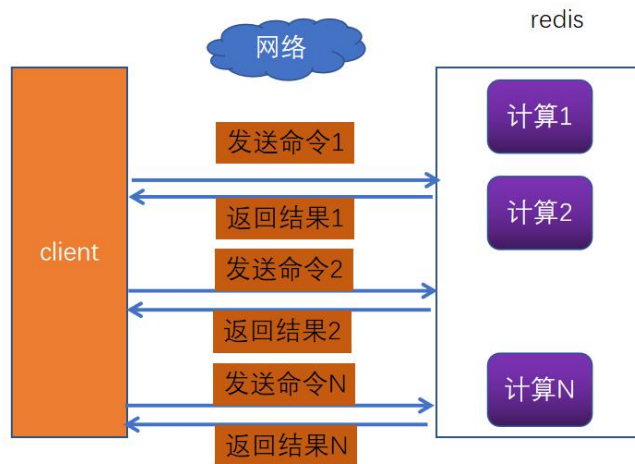
8. PIPELINE 操作流程

大多数情况下, 我们都会通过请求-相应机制去操作 redis。只用这种模式的一般的步骤是, 先获得 jedis 实例, 然后通过 jedis 的 get/put 方法与 redis 交互。由于 redis 是单线程的, 下一次请求必须等待上一次请求执行完成后才能继续执行。然而使用 Pipeline 模式, 客户端可以一次性的发送多个命令, 无需等待服务端返回。这样就大大的减少了网络往返时间, 提高了系统性能。

A>批量操作时使用如下代码网络开销非常大

```
public static void delByStu(String...keys){  
    Jedis jedis = new Jedis(RedisTools.ip, RedisTools.port);  
    for(String key: keys){  
        jedis.del(key);  
    }  
    jedis.close();  
}
```

每一次请求都会建立网络连接, 非常耗时, 特别是跨机房的场景下



B>使用 PIPELINE 可以解决网络开销的问题,代码如下:

```

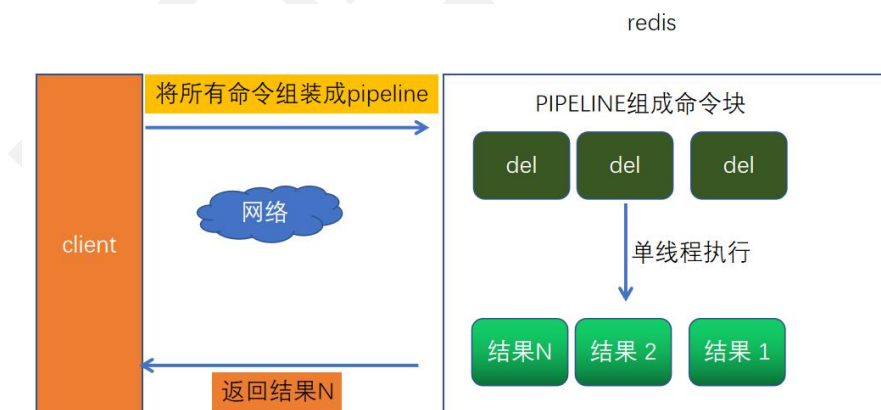
public static void delByPipeline(String...keys) {
    Jedis jedis = new Jedis(RedisTools.ip, RedisTools.port);

    Pipeline pipelined = jedis.pipelined();

    for(String key: keys){
        pipelined.del(key); //提交redis? 1 2 封装 没提交
    }
    pipelined.sync(); //提交
    jedis.close();
}

```

原理也非常简单,流程如下, 将多个指令打包后,一次性提交到 Redis, 网络通信只有一次



9, Redis 弱事务

刚大家知道, pipeline 是多条命令的组合, 为了保证它的原子性, redis 提供了简单

的事务，什么是事务？事务是指一组动作的执行，这一组动作要么成功，要么失败。

A> redis 的简单事务，将一组需要一起执行的命令放到 multi 和 exec 两个命令之间，其中 multi 代表事务开始，exec 代表事务结束，

注：在 multi 前 set user:age 4 //请提前初始化该值

```
127.0.0.1:6379> multi 事务开始
OK
127.0.0.1:6379> sadd user:name james 业务操作1
QUEUED
127.0.0.1:6379> sadd user:age 24 业务操作2
QUEUED
127.0.0.1:6379> get user:age 若在另一客户端取user:age，返回0，事务还没结束
QUEUED
127.0.0.1:6379> exec 事务结束
1) (integer) 1
2) (integer) 1
3) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get age 此时才能查到数据
"4"
```

B> 停止事务 discard

```
127.0.0.1:6379> multi 事务开始
OK
127.0.0.1:6379> sadd tt 1 业务操作
QUEUED
127.0.0.1:6379> discard 停止事务
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI
127.0.0.1:6379> get tt 查不到数据
(nil)
```

C> 命令错误，语法不正确，导致事务不能正常结束

```
127.0.0.1:6379> multi 事务开始
OK
127.0.0.1:6379> set aa 123 正确业务操作
QUEUED
127.0.0.1:6379> sett bb 234 错误的命令
(error) ERR unknown command 'sett'
127.0.0.1:6379> exec 事务提交无效
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get aa 查不到数据
(nil)
127.0.0.1:6379>
```

D> 运行错误，语法正确，但类型错误，事务可以正常结束


```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> set t 1
QUEUED
127.0.0.1:6379> sadd t 1
QUEUED
127.0.0.1:6379> set t 2
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
3) OK
127.0.0.1:6379> get t
"2"

```

可以看到 redis 不支持回滚功能

E> Watch 让事务失效，操作命令

客户端 1: set book java->watch book ->multi
 客户端 2: append book c
 使用 watch 后, multi 失效, 事务失效

10, redis 主要提供发布消息、订阅频道、取消订阅以及按照模式订阅和取消订阅

A) 发布消息

publish channel:test "hello world"

```

127.0.0.1:6379> publish channel:test "hello world"
(integer) 0 此时还没订阅, 所以返回0
127.0.0.1:6379>

```

B) 订阅消息

subscribe channel:test

此时另一个客户端发布一个消息:publish channel:test "james test"

当前订阅者客户端会收到如下消息:

```

127.0.0.1:6379> subscribe channel:test
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel:test"
3) (integer) 1
1) "message"
2) "channel:test"
3) "james test"

```

和很多专业的消息队列 (kafka rabbitmq) ,redis 的发布订阅显得很 lower, 比如无法实现消息规程和回溯, 但就是简单, 如果能满足应用场景, 用这个也可以

C) 查看订阅数:

pubsub numsub channel:test // 频道 channel:test 的订阅数

D) 取消订阅

```
unsubscribe channel:test
```

客户端可以通过 `unsubscribe` 命令取消对指定频道的订阅，取消后，不会再收到该频道的消息

E) 按模式订阅和取消订阅

`psubscribe ch*` //订阅以 `ch` 开头的所有频道

```
127.0.0.1:6379> psubscribe ch*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "ch*"
3) (integer) 1
1) "pmessage"
2) "ch*"
3) "channel:test"
4) "james test"
```

`punsubscribe ch*` //取消以 `ch` 开头的所有频道

```
127.0.0.1:6379> punsubscribe ch*
1) "punsubscribe"
2) "ch*"
3) (integer) 0
```

F) 应用场景:

- 1、今日头条订阅号、微信订阅公众号、新浪微博关注、邮件订阅系统
- 2、即时通信系统
- 3、群聊部落系统（微信群）

测试实践：微信班级群 `class: 20170101`

- 1、学生 C 订阅一个主题叫：`class:20170101`

```
>subscribe class:20170101
```

- 2、学生 A 针对 `class:20170101` 主体发送消息，那么所有订阅该主题的用户都能够接收到该数据。

```
>publish class:20170101 "hello world! I am A"
```

- 3、学生 B 针对 `class:20170101` 主体发送消息，那么所有订阅该主题的用户都能够接收到该数据。

```
>publish class:20170101 "hello world! I am B"
```

展示学生 C 接受到的 A\B 同学发送过来的消息信息

```
1) "subscribe"
2) "class:20170101"
3) (integer) 1
1) "message"
2) "class:20170101"
3) "hello world! I am A"
1) "message"
2) "class:20170101"
3) "hello word! I am B"
```

11, 键的迁移: 把部分数据迁移到另一台 redis 服务器

1, move key db //reids 有 16 个库, 编号为 0—15
set name james1; move name 5 //迁移到第 6 个库
select 5; //数据库切换到第 6 个库, get name 可以取到 james1
这种模式不建议在生产环境使用, 在同一个 reids 里可以玩

2, dump key;

restore key ttl value //实现不同 redis 实例的键迁移, ttl=0 代表没有过期时间

例子: 在 A 服务器上 192.168.42.111

set name james;

dump name; // 得到"\x00\x05james\b\x001\x82;\f"DhJ"

在 B 服务器上: 192.168.1.118

restore name 0 "\x00\x05james\b\x001\x82;\f"DhJ"

get name //返回 james

3, migrate 用于在 Redis 实例间进行数据迁移, 实际上 migrate 命令是将 dump、restore、del 三个命令进行组合, 从而简化了操作流程。

migrate 命令具有原子性, 从 Redis 3.0.6 版本后已经支持迁移多个键的功能。

migrate 命令的数据传输直接在源 Redis 和目标 Redis 上完成, 目标 Redis 完成 restore 后会发送 OK 给源 Redis。

migrate 实例操作:

migrate 指令迁移到其它实例 redis, 在 4222.111 服务器上将 name 移到 112

migrate	192.168.42.112	6379	name	0	1000	copy	replace	keys
指令	要迁移的目标 IP	端口	迁移键值	目标库	超时时间	迁移后不删除原键	不管目标库是不存在 test 键都迁移成功	迁移多个键

指令如下: 把 111 上的 name 键值迁移到 112 上的 redis

192.168.42.111:6379> migrate 192.168.42.112 6379 name 0 1000 copy

12, 键的遍历

redis 提供了两个命令来遍历所有的键

1, 键全量遍历:

mset country china city bj name james //设置 3 个字符串键值对

keys * //返回所有的键, * 匹配任意字符多个字符

keys *y //以结尾的键,

keys n*e //以 n 开头以 e 结尾, 返回 name

keys n?me // ? 问号代表只匹配一个字符 返回 name, 全局匹配

keys n?m* //返回 name

keys [j,l]* //返回以 j,l 开头的键 keys [j]ames 全量匹配 james

考虑到是单线程，在生产环境不建议使用，如果键多可能会阻塞
如果键少，可以

2，渐进式遍历

初始化 13 组 KEY-VALUE

渐进式遍历

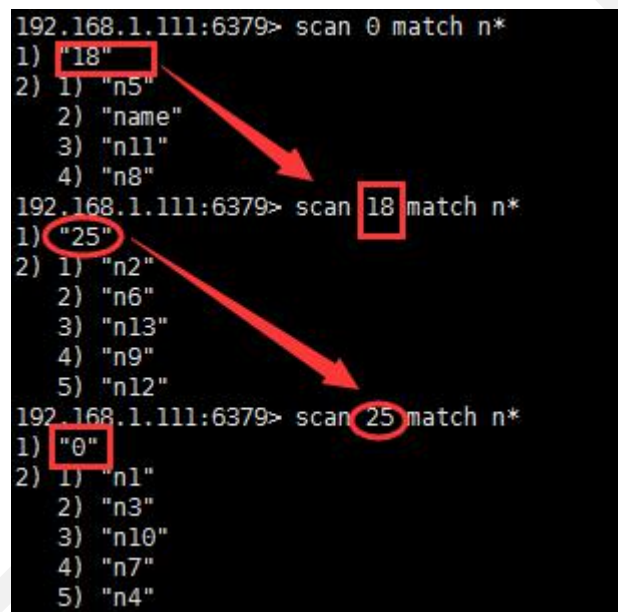
1，初始化数据

```
mset n1 1 n2 2 n3 3 n4 4 n5 5 n6 6 n7 7 n8 8 n9 9 n10 10 n11 11 n12 12 n13 13
```

2，遍历匹配:匹配以 n 开头的键，最大是取 5 条，第一次 scan 0 开始

```
scan 0 match n* count 5
```

第二次从游标 4096 开始取 20 个以 n 开头的键，相当于一页一页的取当最后
返回 0 时，键被取完,但 count 不准,一般用来代替 keys *操作，可避免阻塞



```
192.168.1.111:6379> scan 0 match n*
1) "18"
2) 1) "n5"
   2) "name"
   3) "n11"
   4) "n8"
192.168.1.111:6379> scan 18 match n*
1) "25"
2) 1) "n2"
   2) "n6"
   3) "n13"
   4) "n9"
   5) "n12"
192.168.1.111:6379> scan 25 match n*
1) "0"
2) 1) "n1"
   2) "n3"
   3) "n10"
   4) "n7"
   5) "n4"
```

第二次从游标 18 开始取 20 个以 n 开头的键，相当于一页一页的取
当最后返回 0 时，键被取完

3，两种遍历对比

scan 相比 keys 具备有以下特点:

1，通过游标分布进行的，不会阻塞线程（可能这个特点我们靠谱点，下面了解即可）；

2，提供 limit 参数，可以控制每次返回结果的最大条数，limit 不准，返回的结果可多可少；

3，同 keys 一样，Scan 也提供模式匹配功能；

4，服务器不需要为游标保存状态，游标的唯一状态就是 scan 返回给客户端的游标整数；

5，scan 返回的结果可能会有重复，需要客户端去重复；

6，scan 遍历的过程中如果有数据修改，改动后的数据能不能遍历到是不确定的；

7，单次返回的结果是空的并不意味着遍历结束，而要看返回的游标值是否为零；

注：可有效地解决 keys 命令可能产生的阻塞问题

- 除 scan 字符串外：还有以下
- **SCAN** 命令用于迭代当前数据库中的数据库键。
- **SSCAN** 命令用于迭代集合键中的元素。
- **HSCAN** 命令用于迭代哈希键中的键值对。
- **ZSCAN** 命令用于迭代有序集合中的元素（包括元素成员和分值）。

用法和 scan 一样