redis 学习笔记整理

一、 redis 环境搭建.....2
二、 redis 学习笔记之数据类型.....3
三、 redis 学习笔记之排序.....11
四、 redis 学习笔记之事务.....16
五、 redis 学习笔记之 pipeline.....20
六、 redis 学习笔记之发布订阅.....23
七、 redis 学习笔记之持久化.....28
八、 redis 学习笔记之主从复制.....30
九、 redis 学习笔记之虚拟内存.....31

redis 环境搭建

1.简介

redis 是一个开源的 key-value 数据库。它又经常被认为是一个数据结构服务器。因为它的 value 不仅包括基本的 string 类型还有 list,set ,sorted set 和 hash 类型。当然这些类型的元素也都是 string 类型。也就是说 list,set 这些集合类型也只能包含
string 类型。你可以在这些类型上做很多原子性的操作。比如对一个字符 value 追加字符串（APPEND 命令）。加加或者减减一个数字字符串(INCR 命令，当 然是按整数处理的).可以对 list 类型进行 push,或者 pop 元素操作（可以模拟栈和队列）。对于 set 类型可以进行一些集合相关操作 (intersection union difference)。memcache 也有类似与++,--的命令。
不过 memcache 的 value 只包括 string 类型。远没有 redis 的 value 类型丰富。和 memcahe 一样为了性能。redis 的数据通常都是放到内存中的。当然 redis 可以每间隔一定时间将内存中数据写入到磁盘以防止数据丢失。redis 也支持主从复制机制（master-slave replication）。redis 的其他特性包括简单的事务支持和 发布订阅(pub/sub)通道功能,而且 redis 配置管理非常简单。还有各种语言版本的开源客户端类库。

2.安装

下载地址：http://redis.googlecode.com/files/redis-2.0.4.tar.gz
2.0 目前是最新稳定版
可以在 linux 下运行如下命令进行安装

```
$ tar xzf redis-2.0.4.tar.gz
$ cd redis-2.0.4
$ make
```

make 完后 redis-2.0.4 目录下会出现编译后的 redis 服务程序 redis-server,还有用于测试的客户端程序 redis-cli

下面启动 redis 服务.

```
$./redis-server
```

注意这种方式启动 redis 使用的是默认配置。也可以通过启动参数告诉 redis 使用指定配置文件使用下面命令启动.

```
$ ./redis-server redis.conf
```

redis.conf 是一个默认的配置文件。我们可以根据需要使用自己的配置文件。

启动 redis 服务进程后，就可以使用测试客户端程序 redis-cli 和 redis 服务交互了.

比如

```
$ ./redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

这里演示了 get 和 set 命令操作简单类型 value 的例子。foo 是 key ,bar 是个 string 类型的 value

没 linux 的可以通过这个在线的来练习，当然在线版的很多管理相关的命令是不支持的。

http://try.redis-db.com/

3.java 客户端 hello,world

客户端 jar 包地址 http://cloud.github.com/downloads/alphazero/jredis/jredis-1.0-rc2.jar 。版本目前有点老，支持到 Redis 1.2.6。最新版 2.0 的还没 release

在 eclipse 中新建一个 java 项目，然后添加 jredis 包引用。下面是个 hello,world 程序

```
package jredisStudy;
import org.jredis.*;
import org.jredis.ri.alphazero.JRedisClient;
public class App {
public static void main(String[] args) {
try {
            JRedis  jr = new JRedisClient("192.168.56.55",6379); //redis 服务地址和端口号
            String key = "mKey";
            jr.set(key, "hello,redis!");
            String v = new String(jr.get(key));
            String k2 = "count";
            jr.incr(k2);
            jr.incr(k2);
            System.out.println(v);
            System.out.println(new String(jr.get(k2)));
      } catch (Exception e) {
// TODO: handle exception
      }
```

```
    }
}
```

好了 redis 环境已经搭建好了。后面会写写 redis 的各种类型和类型相关的命令和一些具体的应用场景

redis 学习笔记之数据类型

本文介绍下 redis 支持的各种数据类型包括 string,list ,set ,sorted set 和 hash

1. keys

redis 本质上一个 key-value db，所以我们首先来看看他的 key.首先 key 也是字符串类型，但是 key 中不能包括边界字符

由于 key 不是 binary safe 的字符串，所以像"my key"和"mykey\n"这样包含空格和换行的 key 是不允许的

顺便说一下在 redis 内部并不限制使用 binary 字符，这是 redis 协议限制的。"\r\n"在协议格式中会作为特殊字符。

redis 1.2 以后的协议中部分命令已经开始使用新的协议格式了(比如 MSET)。总之目前还是把包含边界字符当成非法的 key

吧，

免得被 bug 纠缠。

另外关于 key 的一个格式约定介绍下，object-type:id:field。比如 user:1000:password，blog:xxidxx:title

还有 key 的长度最好不要太长。道理很明显占内存啊，而且查找时候相对短 key 也更慢。不过也推荐过短的 key，

比如 u:1000:pwd,这样的。显然没上面的 user:1000:password 可读性好。

下面介绍下 key 相关的命令

exits key 测试指定 key 是否存在，返回 1 表示存在，0 不存在

del key1 key2 ....keyN 删除给定 key,返回删除 key 的数目，0 表示给定 key 都不存在

type key 返回给定 key 的 value 类型。返回 none 表示不存在 key,string 字符类型，list 链表类型 set 无序集合类型...

keys pattern 返回匹配指定模式的所有 key,下面给个例子

redis> set test dsf

OK

redis> set tast dsaf

OK

redis> set tist adff

OK

redis> keys t*

1. "tist"

2. "tast"

3. "test"

redis> keys t[ia]st

1. "tist"

2. "tast"

redis> keys t?st

1. "tist"

2. "tast"

3. "test"

randomkey 返回从当前数据库中随机选择的一个 key,如果当前数据库是空的，返回空串

rename oldkey newkey 原子的重命名一个 key,如果 newkey 存在，将会被覆盖，返回 1 表示成功，0 失败。可能是 oldk

ey 不存在或者和 newkey 相同

renamenx oldkey newkey 同上，但是如果 newkey 存在返回失败

dbsize 返回当前数据库的 key 数量

expire key seconds 为 key 指定过期时间，单位是秒。返回 1 成功，0 表示 key 已经设置过过期时间或者不存在

ttl key 返回设置过过期时间的 key 的剩余过期秒数 -1 表示 key 不存在或者没有设置过过期时间

select db-index 通过索引选择数据库，默认连接的数据库所有是 0,默认数据库数是 16 个。返回 1 表示成功，0 失败

move key db-index 将 key 从当前数据库移动到指定数据库。返回 1 成功。0 如果 key 不存在，或者已经在指定数据库中

flushdb 删除当前数据库中所有 key,此方法不会失败。慎用

flushall 删除所有数据库中的所有 key，此方法不会失败。更加慎用

2. string 类型

string 是 redis 最基本的类型，而且 string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据。比如 jpg 图片或

者序列化的对象

。从内部实现来看其实 string 可以看作 byte 数组，最大上限是 1G 字节。下面是 string 类型的定义。

```
struct sdshdr {
    long len;
    long free;
    char buf[];
};
```

buf 是个 char 数组用于存贮实际的字符串内容。其实 char 和 c#中的 byte 是等价的，都是一个字节

len 是 buf 数组的长度，free 是数组中剩余可用字节数。由此可以理解为什么 string 类型是二进制安全的了。因为它本质上就是个 byte 数组。

当然可以包含任何数据了。另外 string 类型可以被部分命令按 int 处理.比如 incr 等命令，下面详细介绍。还有 redis 的其他类型像 list,set,sorted set ， hash

它们包含的元素与都只能是 string 类型。

如果只用 string 类型，redis 就可以被看作加上持久化特性的 memcached.当然 redis 对 string 类型的操作比 memcached 多很多啊。如下：

set key value 设置 key 对应的值为 string 类型的 value,返回 1 表示成功，0 失败

setnx key value 同上，如果 key 已经存在，返回 0 。nx 是 not exist 的意思

get key 获取 key 对应的 string 值,如果 key 不存在返回 nil

getset key value 原子的设置 key 的值，并返回 key 的旧值。如果 key 不存在返回 nil

mget key1 key2 ... keyN 一次获取多个 key 的值，如果对应 key 不存在，则对应返回 nil。下面是个实验,首先清空当前数据库，然后

设置 k1,k2.获取时 k3 对应返回 nil

```
redis> flushdb
OK
redis> dbsize
(integer) 0
redis> set k1 a
```

OK

redis> set k2 b

OK

redis> mget k1 k2 k3

1. "a"

2. "b"

3. (nil)

mset key1 value1 ... keyN valueN 一次设置多个 key 的值，成功返回 1 表示所有的值都设置了，失败返回 0 表示没有任何值被设置

msetnx key1 value1 ... keyN valueN 同上，但是不会覆盖已经存在的 key

incr key 对 key 的值做加加操作,并返回新的值。注意 incr 一个不是 int 的 value 会返回错误，incr 一个不存在的 key，则设置 key 为 1

decr key 同上，但是做的是减减操作，decr 一个不存在 key，则设置 key 为-1

incrby key integer 同 incr，加指定值 ，key 不存在时候会设置 key，并认为原来的 value 是 0

decrby key integer 同 decr，减指定值。decrby 完全是为了可读性，我们完全可以通过 incrby 一个负值来实现同样效果，反之一样。

append key value 给指定 key 的字符串值追加 value,返回新字符串值的长度。下面给个例子

redis> set k hello

OK

redis> append k ,world

(integer) 11

redis> get k

"hello,world"

substr key start end 返回截取过的 key 的字符串值,注意并不修改 key 的值。下标是从 0 开始的，接着上面例子

redis> substr k 0 8

"hello,wor"

redis> get k

"hello,world"

3. list

redis 的 list 类型其实就是一个每个子元素都是 string 类型的双向链表。所以[lr]push 和[lr]pop 命令的算法时间复杂度都是 O(1)

另外 list 会记录链表的长度。所以 llen 操作也是 O(1).链表的最大长度是(2 的 32 次方-1)。我们可以通过 push,pop 操作从链表的头部

或者尾部添加删除元素。这使得 list 既可以用作栈，也可以用作队列。有意思的是 list 的 pop 操作还有阻塞版本的。当我们[lr]pop 一个

list 对象是，如果 list 是空，或者不存在，会立即返回 nil。但是阻塞版本的 b[lr]pop 可以则可以阻塞，当然可以加超时时间，超时后也会返回 nil

。为什么要阻塞版本的 pop 呢，主要是为了避免轮询。举个简单的例子如果我们用 list 来实现一个工作队列。执行任务的 thread 可以调用阻塞版本的 pop 去

获取任务这样就可以避免轮询去检查是否有任务存在。当任务来时候工作线程可以立即返回，也可以避免轮询带来的延迟。ok 下面介绍 list 相关命令

lpush key string 在 key 对应 list 的头部添加字符串元素，返回 1 表示成功，0 表示 key 存在且不是 list 类型

rpush key string 同上，在尾部添加

llen key 返回 key 对应 list 的长度，key 不存在返回 0,如果 key 对应类型不是 list 返回错误

lrange key start end 返回指定区间内的元素，下标从 0 开始，负值表示从后面计算，-1 表示倒数第一个元素 ，key 不存在返回空列表

ltrim key start end 截取 list，保留指定区间内元素，成功返回 1，key 不存在返回错误

lset key index value 设置 list 中指定下标的元素值，成功返回 1，key 或者下标不存在返回错误

lrem key count value 从 key 对应 list 中删除 count 个和 value 相同的元素。count 为 0 时候删除全部

lpop key 从 list 的头部删除元素，并返回删除元素。如果 key 对应 list 不存在或者是空返回 nil，如果 key 对应值不是 list

返回错误

rpop 同上，但是从尾部删除

blpop key1...keyN timeout 从左到右扫描返回对第一个非空 list 进行 lpop 操作并返回，比如 blpop list1 list2 list3 0 ，如果 list 不存在

list2,list3 都是非空则对 list2 做 lpop 并返回从 list2 中删除的元素。如果所有的 list 都是空或不存在，则会阻塞 timeout 秒，timeout 为 0 表示一直阻塞。

当阻塞时，如果有 client 对 key1...keyN 中的任意 key 进行 push 操作，则第一在这个 key 上被阻塞的 client 会立即返回。如果超时发生，则返回 nil。有点像 unix 的 select 或者 poll

brpop 同 blpop，一个是从头部删除一个是从尾部删除

rpoplpush srckey destkey 从 srckey 对应 list 的尾部移除元素并添加到 destkey 对应 list 的头部,最后返回被移除的元素值，整个操作是原子的.如果 srckey 是空

或者不存在返回 nil

4. set

redis 的 set 是 string 类型的无序集合。set 元素最大可以包含(2 的 32 次方-1)个元素。set 的是通过 hash table 实现的，所以添加，删除，查找的复杂度都是 O(1)。hash table 会随着添加或者删除自动的调整大小。需要注意的是调整 hash table 大小时候需要同步（获取写锁）会阻塞其他读写操作。可能不久后就会改用跳表（skip list）来实现

跳表已经在 sorted set 中使用了。关于 set 集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)，

差集(difference)。通过这些操作可以很容易的实现 sns 中的好友推荐和 blog 的 tag 功能。下面详细介绍 set 相关命令

sadd key member 添加一个 string 元素到,key 对应的 set 集合中，成功返回 1,如果元素以及在集合中返回 0,key 对应的 set 不存在返回错误

srem key member 从 key 对应 set 中移除给定元素，成功返回 1，如果 member 在集合中不存在或者 key 不存在返回 0，如果 key 对应的不是 set 类型的值返回错误

spop key 删除并返回 key 对应 set 中随机的一个元素,如果 set 是空或者 key 不存在返回 nil

srandmember key 同 spop，随机取 set 中的一个元素，但是不删除元素

smove srckey dstkey member 从 srckey 对应 set 中移除 member 并添加到 dstkey 对应 set 中，整个操作是原子的。

成功返回 1,如果 member 在 srckey 中不存在返回 0，如果

key 不是 set 类型返回错误

scard key 返回 set 的元素个数，如果 set 是空或者 key 不存在返回 0

sismember key member 判断 member 是否在 set 中，存在返回 1，0 表示不存在或者 key 不存在

sinter key1 key2...keyN 返回所有给定 key 的交集

sinterstore dstkey key1...keyN 同 sinter，但是会同时将交集存到 dstkey 下

sunion key1 key2...keyN 返回所有给定 key 的并集

sunionstore dstkey key1...keyN 同 sunion，并同时保存并集到 dstkey 下

sdiff key1 key2...keyN 返回所有给定 key 的差集

sdiffstore dstkey key1...keyN 同 sdiff，并同时保存差集到 dstkey 下

smembers key 返回 key 对应 set 的所有元素，结果是无序的

5 sorted set

和 set 一样 sorted set 也是 string 类型元素的集合，不同的是每个元素都会关联一个 double 类型的 score。sorted set 的实现是 skip list 和 hash table 的混合体

当元素被添加到集合中时，一个元素到 score 的映射被添加到 hash table 中，所以给定一个元素获取 score 的开销是 O(1)，另一个 score 到元素的映射被添加到 skip list

并按照 score 排序，所以就可以有序的获取集合中的元素。添加，删除操作开销都是 O(log(N))和 skip list 的开销一致,redis 的 skip list 实现用的是双向链表,这样就

可以逆序从尾部取元素。sorted set 最经常的使用方式应该是作为索引来使用.我们可以把要排序的字段作为 score 存储，对象的 id 当元素存储。下面是 sorted set 相关命令

zadd key score member 添加元素到集合，元素在集合中存在则更新对应 score

zrem key member 删除指定元素，1 表示成功，如果元素不存在返回 0

zincrby key incr member 增加对应 member 的 score 值，然后移动元素并保持 skip list 保持有序。返回更新后的 score 值

zrank key member 返回指定元素在集合中的排名（下标）,集合中元素是按 score 从小到大排序的

zrevrank key member 同上,但是集合中元素是按 score 从大到小排序

zrange key start end 类似 lrange 操作从集合中去指定区间的元素。返回的是有序结果

zrevrange key start end 同上，返回结果是按 score 逆序的

zrangebyscore key min max 返回集合中 score 在给定区间的元素

zcount key min max 返回集合中 score 在给定区间的数量

zcard key 返回集合中元素个数

zscore key element 返回给定元素对应的 score

zremrangebyrank key min max 删除集合中排名在给定区间的元素

zremrangebyscore key min max 删除集合中 score 在给定区间的元素

6. hash

redis hash 是一个 string 类型的 field 和 value 的映射表.它的添加，删除操作都是 O(1)（**平均**）.hash 特别适合用于存储对象。相较于将对象的每个字段存成

单个 string 类型。将一个对象存储在 hash 类型中会占用更少的内存，并且可以更方便的存取整个对象。省内存的原因是新建一个 hash 对象时开始是用 zipmap（又称为 small hash）来存储的。这个 zipmap 其实并不是 hash table，但是 zipmap 相比正常的 hash 实现可以节省不少 hash 本身需要的一些元数据存储开销。尽管 zipmap 的添加，删除，查找都是 O(n)，但是由于一般对象的 field 数量都不太多。所以使用 zipmap 也是很快的,也就是说添加删除平均还是 O(1)。如果 field 或者 value 的大小超出一定限制后，redis 会在内部自动将 zipmap 替换成正常的 hash 实现. 这个限制可以在配置文件中指定

hash-max-zipmap-entries 64 #配置字段最多 64 个

hash-max-zipmap-value 512 #配置 value 最大为 512 字节

下面介绍 hash 相关命令

hset key field value 设置 hash field 为指定值，如果 key 不存在，则先创建

hget key field 获取指定的 hash field

hmget key filed1....fieldN 获取全部指定的 hash filed

hmset key filed1 value1 ... filedN valueN 同时设置 hash 的多个 field

hincrby key field integer 将指定的 hash filed 加上给定值

hexists key field 测试指定 field 是否存在

hdel key field 删除指定的 hash field

hlen key 返回指定 hash 的 field 数量

hkeys key 返回 hash 的所有 field

hvals key 返回 hash 的所有 value

hgetall 返回 hash 的所有 filed 和 value

redis 学习笔记之排序

在了解完各种 redis 类型后，这次介绍下 redis 排序命令.redis 支持对 list，set 和 sorted set 元素的排序。排序命令是 sort

完整的命令格式如下：

SORT key [BY pattern] [LIMIT start count] [GET pattern] [ASC|DESC] [ALPHA] [STORE dstkey]

下面我们一一说明各种命令选项

（1）sort key

这个是最简单的情况，没有任何选项就是简单的对集合自身元素排序并返回排序结果.下面给个例子

redis> lpush ml 12

(integer) 1

redis> lpush ml 11

(integer) 2

redis> lpush ml 23

(integer) 3

redis> lpush ml 13

(integer) 4

redis> sort ml

1. "11"

2. "12"

3. "13"

4. "23"

(2)[ASC|DESC] [ALPHA]

sort 默认的排序方式（asc）是从小到大排的,当然也可以按照逆序或者按字符顺序排。逆序可以加上 desc 选项，想按字母顺序排可以加 alpha 选项，当然 alpha 可以和 desc 一起用。下面是个按字母顺序排的例子

redis> lpush mylist baidu

(integer) 1

redis> lpush mylist hello

(integer) 2

redis> lpush mylist xhan

(integer) 3

redis> lpush mylist soso

(integer) 4

redis> sort mylist

1. "soso"

2. "xhan"

3. "hello"

4. "baidu"

redis> sort mylist alpha

1. "baidu"

2. "hello"

3. "soso"

4. "xhan"

redis> sort mylist desc alpha

1. "xhan"

2. "soso"

3. "hello"

4. "baidu"

（3）[BY pattern]

除了可以按集合元素自身值排序外，还可以将集合元素内容按照给定 pattern 组合成新的 key，并按照新 key 中对应的内容进行排序。下面的例子接着使用第一个例子中的 ml 集合做演示：

redis> set name11 nihao

OK

redis> set name12 wo

OK

redis> set name13 shi

OK

redis> set name23 lala

OK

redis> sort ml by name*

1. "13"

2. "23"

3. "11"

4. "12"

*代表了 ml 中的元素值，所以这个排序是按照 name12 name13 name23 name23 这四个 key 对应值排序的,当然返回的还是排序后 ml 集合中的元素

(4)[GET pattern]

上面的例子都是返回的 ml 集合中的元素。我们也可以通过 get 选项去获取指定 pattern 作为新 key 对应的值。看个组合起来的例子

redis> sort ml by name* get name* alpha

1. "lala"

2. "nihao"

3. "shi"

4. "wo"

这 次返回的就不在是 ml 中的元素了，而是 name12 name13 name23 name23 对应的值。当然排序是按照 name12 na

me13 name23 name23值并根据字母顺序排的。另外 get选项可以有多个。看例子（#特殊符号引用的是原始集合也就是ml）

```
redis> sort ml by name* get name* get #  alpha
1. "lala"
2. "23"
3. "nihao"
4. "11"
5. "shi"
6. "13"
7. "wo"
8. "12"
```

最后在还有一个引用hash类型字段的特殊字符->，下面是例子

```
redis> hset user1 name hanjie
(integer) 1
redis> hset user11 name hanjie
(integer) 1
redis> hset user12 name 86
(integer) 1
redis> hset user13 name lxl
(integer) 1
redis> sort ml get user*->name
1. "hanjie"
2. "86"
3. "lxl"
4. (nil)
```

很容易理解，注意当对应的user23不存在时候返回的是 nil

(5) [LIMIT start count]

上面例子返回结果都是全部。limit 选项可以限定返回结果的数量。例子

redis> sort ml get name* limit 1 2

1. "wo"

2. "shi"

start 下标是从 0 开始的，这里的 limit 选项意思是从第二个元素开始获取 2 个

(6)[STORE dstkey]

如果对集合经常按照固定的模式去排序，那么把排序结果缓存起来会减少不少 cpu 开销.使用 store 选项可以将排序内容保存到指定 key 中。保存的类型是 list

redis> sort ml get name* limit 1 2 store cl

(integer) 2

redis> type cl

list

redis> lrange cl 0 -1

1. "wo"

2. "shi"

这个例子我们将排序结果保存到了 cl 中

功能介绍完后，再讨论下关于排序的一些问题。如果我们有多个 redis server 的话，不同的 key 可能存在于不同的 server 上。比如 name12 name13 name23 name23，很有可能分别在四个不同的 server 上存贮着。这种情况会对排序性能造成很大的影响。redis 作者在他的 blog 上提到了这个问题的解 决办法，就是通过 key tag 将需要排序的 key 都放到同一个 server 上 。由于具体决定哪个 key 存在哪个服务器上一般都是在 client 端 hash 的办法来做的。我们可以通过只对 key 的部分进行 hash.举个例子假如我们 的 client 如果发现 key 中包含[]。那么只对 key 中[]包含的内容进行 hash。我们将四个 n

ame相关的key，都这样命名[name]12 [name]13 [name]23 [name]23，于是client 程序就会把他们都放到同一 server 上。不知道jredis实现了没。

还有一个问题也比较严重。如果要sort的集合非常大的话排序就会消耗很长时间。由于redis单线程的，所以长时间的排序操作会阻塞其他client的 请求。解决办法是通过主从复制机制将数据复制到多个slave上。然后我们只在slave上做排序操作。并进可能的对排序结果缓存。另外就是一个方案是就 是采用sorted set对需要按某个顺序访问的集合建立索引。

redis学习笔记之事务

redis对事务的支持目前还比较简单。redis只能保证一个client发起的事务中的命令可以连续的执行，而中间不会插入其他client的命令。 由于redis是单线程来处理所有client的请求的所以做到这点是很容易的。一般情况下redis在接受到一个client发来的命令后会立即处理并 返回处理结果，但是当一个client在一个连接中发出multi命令有，这个连接会进入一个事务上下文，该连接后续的命令并不是立即执行，而是先放到一 个队列中。当从此连接受到exec命令后，redis会顺序的执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就 结束事务上下文。下面可以看一个例子

```
redis> multi
OK
redis> incr a
QUEUED
redis> incr b
QUEUED
redis> exec
1. (integer) 1
2. (integer) 1
```

从这个例子我们可以看到incr a ,incr b命令发出后并没执行而是被放到了队列中。调用exec后俩个命令被连续的执行，最后返回的是两条命令执行后的结果

我们可以调用discard命令来取消一个事务。接着上面例子

```
redis> multi
OK
```

```
redis> incr a
QUEUED
redis> incr b
QUEUED
redis> discard
OK
redis> get a
"1"
redis> get b
"1"
```

可以发现这次 incr a incr b 都没被执行。discard 命令其实就是清空事务的命令队列并退出事务上下文。

虽说 redis 事务在本质上也相当于序列化隔离级别的了。但是由于事务上下文的命令只排队并不立即执行，所以事务中的写操作不能依赖事务中的读操作结果。看下面例子

```
redis> multi
OK
redis> get a
QUEUED
redis> get b
QUEUED
redis> exec
1. "1"
2. "1"
```

发现问题了吧。假如我们想用事务实现 incr 操作怎么办？可以这样做吗？

```
redis> get a
```

"1"

redis> multi

OK

redis> set a 2

QUEUED

redis> exec

1. OK

redis> get a,

"2"

结 论很明显这样是不行的。这样和 get a 然后直接 set a 是没区别的。很明显由于 get a 和 set a 并不能保证两个命令是连续执行的(get 操作不在事务上下文中)。很可能有两个 client 同时做这个操作。结果我们期望是加两次 a 从原来的 1 变成 3. 但是很有可能两个 client 的 get a，取到都是 1，造成最终加两次结果却是 2。主要问题我们没有对共享资源 a 的访问进行任何的同步

也就是说 redis 没提供任何的加锁机制来同步对 a 的访问。

还好 redis 2.1 后添加了 watch 命令，可以用来实现乐观锁。看个正确实现 incr 命令的例子,只是在前面加了 watch a

redis> watch a

OK

redis> get a

"1"

redis> multi

OK

redis> set a 2

QUEUED

redis> exec

1. OK

redis> get a,

"2"

watch 命令会监视给定的 key,当 exec 时候如果监视的 key 从调用 watch 后发生过变化，则整个事务会失败。也可以调用 watch 多次监视多个 key.这 样就可以对指定的 key 加乐观锁了。注意 watch 的 key 是对整个连接有效的，事务也一样。如果连接断开，监视和事务都会被自动清除。当然了 exec,discard,unwatch 命令都会清除连接中的所有监视.

redis 的事务实现是如此简单，当然会存在一些问题。第一个问题是 redis 只能保证事务的每个命令连续执行，但是如果事务中的一个命令失败了，并不回滚其他命令，比如使用的命令类型不匹配。

```
redis> set a 5
OK
redis> lpush b 5
(integer) 1
redis> set c 5
OK
redis> multi
OK
redis> incr a
QUEUED
redis> incr b
QUEUED
redis> incr c
QUEUED
redis> exec
1. (integer) 6
2. (error) ERR Operation against a key holding the wrong kind of value
3. (integer) 6
```

可以看到虽然 incr b 失败了，但是其他两个命令还是执行了。

最 后一个十分罕见的问题是 当事务的执行过程中，如果 redis 意外的挂了。很遗憾只有部分命令执行了，后面的也就被丢弃了。当然如果我们使用的 append-only file 方式持久化，redis 会用单个 write 操作写入整个事务内容。即是是这种方式还是有可能只部分写入了事务到磁盘。发生部分写入事务的情况 下，redis 重启时会检测到这种情况，然后失败退出。可以使用 redis-check-aof 工具进行修复，修复会删除部分写入的事务内容。修复完后就 能够重新启动了。

redis 学习笔记之 pipeline

redis 是一个 cs 模式的 tcp server，使用和 http 类似的请求响应协议。一个 client 可以通过一个 socket 连接发起多个请求命令。每个请求命令发出后 client 通常 会阻塞并等待 redis 服务处理，redis 处理完后请求命令后会将结果通过响应报文返回给 client。基本的通信过程如下

Client: INCR X

Server: 1

Client: INCR X

Server: 2

Client: INCR X

Server: 3

Client: INCR X

Server: 4

基 本上四个命令需要 8 个 tcp 报文才能完成。由于通信会有网络延迟,假如从 client 和 server 之间的包传输时间需要 0.125 秒。那么上面的四个命 令 8 个报文至少会需要 1 秒才能完成。这样即使 redis 每秒能处理 100 个命令，而我们的 client 也只能一秒钟发出四个命令。这显示没有充分利用 redis 的处理能力。除了可以利用 mget,mset 之类的单条命令处理多个 key 的命令外

我们还可以利用 pipeline 的方式从 client 打包多条命令一起发出，不需要等待单条命令的响应返回，而 redis 服务端会处理完多条命令后会将多条命令的处理结果打包到一起返回给客户端。通信过程如下

Client: INCR X

Client: INCR X

Client: INCR X

Client: INCR X

Server: 1

Server: 2

Server: 3

Server: 4

假 设不会因为tcp 报文过长而被拆分。可能两个 tcp 报文就能完成四条命令,client 可以将四个 incr 命令放到一个 tcp 报文一起发送，server 则可以将四条命令 的处理结果放到一个 tcp 报文返回。通过 pipeline 方式当有大批量的操作时候。我们可以节省很多原来浪费在网络延迟的时间。需要注意到是用 pipeline 方式打包命令发送，redis 必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并是不是打 包的命令越多越好。具体多少合适需要根据具体情况测试。下面是个 jredis 客户端使用 pipeline 的测试

```
package jredisStudy;

import org.jredis.JRedis;

import org.jredis.connector.ConnectionSpec;

import org.jredis.ri.alphazero.JRedisClient;

import org.jredis.ri.alphazero.JRedisPipelineService;

import org.jredis.ri.alphazero.connection.DefaultConnectionSpec;

public class PipeLineTest {

    public static void main(String[] args) {

        long start = System.currentTimeMillis();

        usePipeline();

        long end = System.currentTimeMillis();

        System.out.println(end-start);
```

```
        start =  System.currentTimeMillis();
        withoutPipeline();
        end  = System.currentTimeMillis();
        System.out.println(end-start);

}

private static void withoutPipeline()
{
    try {
        JRedis  jredis = new JRedisClient("192.168.56.55",6379);
           for(int i =0 ; i < 100000 ; i++)
           {
               jredis.incr("test2");
           }
           jredis.quit();
   } catch (Exception e) {
   }
}

private static void usePipeline() {
   try {
       ConnectionSpec spec = DefaultConnectionSpec.newSpec("192.168.56.55", 6379, 0, null);
       JRedis jredis = new JRedisPipelineService(spec);
       for(int i =0 ; i < 100000 ; i++)
       {
           jredis.incr("test2");
       }
```

```
            jredis.quit();
        } catch (Exception e) {
        }
    }
}
```

输出

```
103408 //使用了 pipeline
104598 //没有使用
```

测试结果不是很明显，这应该是跟我的测试环境有关。我是在自己 win 连接虚拟机的 linux。网络延迟比较小。所以 pipeline 优势不明显。如果网络延迟小的话，最好还是不用 pipeline。除了增加复杂外，带来的性能提升不明显。

redis 学习笔记之发布订阅

发布订阅(pub/sub)是一种消息通信模式，主要的目的是解耦消息发布者和消息订阅者之间的耦合，这点和设计模式中的观察者模式比较相似。pub /sub 不仅仅解决发布者和订阅者直接代码级别耦合也解决两者在物理部署上的耦合。redis 作为一个 pub/sub server，在订阅者和发布者之间起到了消息路由的功能。订阅者可以通过 subscribe 和 psubscribe 命令向 redis server 订阅自己感兴趣的消息类型，redis 将消息类型称为通道(channel)。当发布者通过 publish 命令向 redis server 发送特定类型的消息时。订阅该消息类型的全部 client 都会收到此消息。这里消息的传递是多对多的。一个 client 可以订阅多个 channel,也可以向多个 channel 发送消息。

下面做个实验。这里使用两个不同的 client 一个是 redis 自带的 redis-cli 另一个是用 java 写的简单的 client。代码如下

```
import java.net.*;
import java.io.*;
public class PubSubTest {
    public static void main(String[] args) {
        String cmd = args[0]+"\r\n";
        try {
```

```
            Socket socket = new Socket("192.168.56.55",6379);
            InputStream in = socket.getInputStream();
            OutputStream out = socket.getOutputStream();
            out.write(cmd.getBytes());  //发送订阅命令
            byte[] buffer = new byte[1024];
            while (true) {
               int readCount =    in.read(buffer);
               System.out.write(buffer, 0, readCount);
               System.out.println("-------------------------------------");
            }
        } catch (Exception e) {
        }
    }
}
```

代码就是简单的从命令行读取传过来的订阅命令，然后通过一个 socket 连接发送给 redis server,然后进入 while 循环一直读取 redis server 传过来订阅的消息。并打印到控制台

1 首先编译并运行此 java 程序(我是 win7 下面运行的)

```
D:\>javac PubSubTest.java
D:\>java PubSubTest "subscribe news.share news.blog"
*3
$9
subscribe
$10
news.share
:1
```

*3

$9

subscribe

$9

news.blog

:2

---

2 启动 redis-cli

redis> **psubscribe news.***

Reading messages... (press Ctrl-c to quit)

1. "psubscribe"

2. "news.*"

3. (integer) 1

3 再启动一个 redis-cli 用来发布两条消息

redis> **publish news.share "share a link http://www.google.com"**

(integer) 2

redis> **publish news.blog "I post a blog"**

(integer) 2

4.查看两个订阅 client 的输出

此时 java client 打印如下内容

*3

$7

message

$10

news.share

$34

share a link http://www.google.com

---

*3

$7

message

$9

news.blog

$13

I post a blog

另一个 redis-cli 输出如下

1. "pmessage"

2. "news.*"

3. "news.share"

4. "share a link http://www.google.com"

1. "pmessage"

2. "news.*"

3. "news.blog"

4. "I post a blog"

分析下

java client 使用 subscribe 命令订阅 news.share 和 news.blog 两个通道，然后立即收到 server 返回的订阅成功消息，可以看出 redis 的协议是文本类型的，这里不解释具体协议内容了，可以参考 http://redis.io/topics/protocol 或者 http://terrylee.me/blog/post/2011/01/26/redis-internal-part3.aspx。这个报文内容有两部分,第一部分表示该 socket 连接上使用 subscribe 订阅 news.share 成功后，此连接订阅通道数为 1，后一部分表示使用 subscribe 订阅 news.blog 成功后，该连接订 阅通道总数为 2。

redis client 使用 psubscribe 订阅了一个使用通配符的通道(*表示任意字符串)，此订阅会收到所有与 news.*匹配的通道消息。redis- cli 打印到控制台的订阅成功消息表示使用 psubscribe 命令订阅 news.*成功后，连接订阅通道总数为 1。

当我们在一个 client 使用 publish 向 news.share 和 news.blog 通道发出两个消息后。redis 返回的(integer) 2 表示有两个连接收到了此消息。通过观察两个订阅者的输出可以验证。具体格式不解释了，都比较简单。

看 完一个小例子后应该对 pub/sub 功能有了一个感性的认识。需要注意的是当一个连接通过 subscribe 或者 psubscribe 订阅通道后就进入订 阅模式。在这种模式除了再订阅额外的通道或者用 unsubscribe 或者 punsubscribe 命令退出订阅模式，就不能再发送其他命令。另外使用 psubscribe 命令订阅多个通配符通道，如果一个消息匹配上了多个通道模式的话，会多次收到同一个消息。

jredis 目前版本没提供 pub/sub 支持，不过自己实现一个应该也挺简单的。整个应用程序可以共享同一个连接。因为 redis 返回的消息报文中除了消息内容本身外还包括消息相关的通道信息，当收到消息后可以根据不同的通道信息去调用不同的 callback 来处理。

另外个人觉得 redis 的 pub/sub 还是有点太单薄（实现才用 150 行代码）。在安全，认证，可靠性这方便都没有太多支持。

redis 学习笔记之持久化

redis 是一个支持持久化的内存数据库，也就是说 redis 需要经常将内存中的数据同步到磁盘来保证持久化。redis 支持两种持久化方式，一种是 Snapshotting（快照）也是默认方式，另一种是 Append-only file（缩写 aof）的方式。下面分别介绍

Snapshotting

快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为 dump.rdb。

可以通过配置设置自动做快照持久 化的方式。我们可以配置 redis 在 n 秒内如果超过 m 个 key 被修改就自动做快照，下面是默认的快照保存配置

save 900 1 #900 秒内如果超过 1 个 key 被修改，则发起快照保存

save 300 10 #300 秒内容如超过 10 个 key 被修改，则发起快照保存

save 60 10000

下面介绍详细的快照保存过程

1.redis 调用 fork,现在有了子进程和父进程。

2. 父进程继续处理 client 请求，子进程负责将内存内容写入到临时文件。由于 os 的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时 os 会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数 据是 fork 时刻整个数据库的一个快照。

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

client 也可以使用 save 或者 bgsave 命令通知 redis 做一次快照持久化。save 操作是在主线程中保存快照的，由于 redis 是用一个主线程来处理所有 client 的请求，这种方式会阻塞所有 client 请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘 io 操作，可能会严重影响性能。

另外由于快照方式是在一定间隔时间做一次的，所以如果 redis 意外 down 掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用 aof 持久化方式。下面介绍

Append-only file

aof 比快照方式有更好的持久化性，是由于在使用 aof 持久化方式时,redis 会将每一个收到的写命令都通过 write 函数追加到

文件中(默认是 appendonly.aof)。当 redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于 os 会在内核中缓存 write 做的修改，所以可能不是立即写到磁盘上。这样 aof 方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉 redis 我们想要 通过 fsync 函数强制 os 写入到磁盘的时机。有三种方式如下（默认是：每秒 fsync 一次）

```
appendonly yes              //启用 aof 持久化方式
# appendfsync always     //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用
appendfsync everysec     //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐
# appendfsync no    //完全依赖 os，性能最好,持久化没保证
```

aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用 incr test 命令 100 次，文件中必须保存全部的 100 条命令，其实有 99 条都是多余的。因为要恢复数据库的状态其实文件中保存一条 set test 100 就够了。为了压缩 aof 的持久化文件。redis 提供了 bgrewriteaof 命令。收到此命令 redis 将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下

1. redis 调用 fork ，现在有父子两个进程

2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

3.父进程继续处理 client 请求，除了把写命令写入到原来的 aof 文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。

4.当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。

5.现在父进程可以使用临时文件替换老的 aof 文件，并重命名，后面收到的写命令也开始往新的 aof 文件中追加。

需要注意到是重写 aof 文件的操作，并没有读取旧的 aof 文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的 aof 文件,这点和快照有点类似。

redis 学习笔记之主从复制

redis 主从复制配置和使用都非常简单。通过主从复制可以允许多个 slave server 拥有和 master server 相同的数据库副本。下面是关于 redis 主从复制的一些特点

1.master 可以有多个 slave

2.除了多个 slave 连到相同的 master 外，slave 也可以连接其他 slave 形成图状结构

3.主从复制不会阻塞 master。也就是说当一个或多个 slave 与 master 进行初次同步数据时，master 可以继续处理 client 发来的请求。相反 slave 在初次同步数据时则会阻塞不能处理 client 的请求。

4.主从复制可以用来提高系统的可伸缩性,我们可以用多个 slave 专门用于 client 的读请求，比如 sort 操作可以使用 slave 来处理。也可以用来做简单的数据冗余

5.可以在 master 禁用数据持久化，只需要注释掉 master 配置文件中的所有 save 配置，然后只在 slave 上配置数据持久化。

下面介绍下主从复制的过程

当设置好 slave 服务器后，slave 会建立和 master 的连接，然后发送 sync 命令。无论是第一次同步建立的连接还是连接断开后的重新连 接，master 都会启动一个后台进程，将数据库快照保存到文件中，同时 master 主进程会开始收集新的写命令并缓存起来。后台进程完成写文件 后，master 就发送文件给 slave，slave 将文件保存到磁盘上，然后加载到内存恢复数据库快照到 slave 上。接着 master 就会把缓存的命 令转发给 slave。而且后续 master 收到的写命令都会通过开始建立的连接发送给 slave。从 master 到 slave 的同步数据的命令和从 client 发送的命令使用相同的协议格式。当 master 和 slave 的连接断开时 slave 可以自动重新建立连接。如果 master 同时收到多个 slave 发来的同步连接命令，只会使用启动一个进程来写数据库镜像，然后发送给所有 slave。

配置 slave 服务器很简单，只需要在配置文件中加入如下配置

slaveof 192.168.1.1 6379 #指定 master 的 ip 和端口

redis 学习笔记之虚拟内存

首先说明下 redis 的虚拟内存与 os 的虚拟内存不是一码事，但是思路和目的都是相同的。就是暂时把不经常访问的数据从内存交换到磁盘中，从而腾出宝贵的 内存空间用于其他需要访问的数据。尤其是对于 redis 这样的内存数据库，内存总是不够用的。除了可以将数据分割到多个 redis server 外。另外的能够提高数据库容量的办法就是使用 vm 把那些不经常访问的数据交换的磁盘上。如果我们的存储的数据总是有少部分数据被经常访问，大 部分数据很少被访问，对于网站来说确实总是只有少量用户经常活跃。当少量数据被经常访问时，使用 vm 不但能提高单台 redis server 数据库的容量，而且也不会对性能造成太多影响。

redis 没有使用 os 提供的虚拟内存机制而是自己在用户态实现了自己的虚拟内存机制,作者在自己的 blog 专门解释了其中原因。http://antirez.com/post/redis-virtual-memory-story.html

主要的理由有两点

1.os 的虚拟内存是已 4k 页面为最小单位进行交换的。而 redis 的大多数对象都远小于 4k，所以一个 os 页面上可能有多个 redis 对象。另外 redis 的集 合对象类型如 list,set 可能存在与多个 os 页面上。最终可能造成只有 10%key 被经常访问，但是所有 os 页面都会被 os 认为是活跃的，这样只有内 存真正耗尽时 os 才会交换页面。

2.相比于 os 的交换方式。redis 可以将被交换到磁盘的对象进行压缩,保存到磁盘的对象可以去除指针和对象元数据信息。一般压缩后的对象会比内存中的对象小 10 倍。这样 redis 的 vm 会比 os vm 能少做很多 io 操作。

下面是 vm 相关配置

```
vm-enabled yes                    #开启 vm 功能
vm-swap-file /tmp/redis.swap      #交换出来的 value 保存的文件路径/tmp/redis.swap
vm-max-memory 1000000             #redis 使用的最大内存上限,超过上限后 redis 开始交换 value 到磁盘文件中。

vm-page-size 32                   #每个页面的大小 32 个字节
vm-pages 134217728                #最多使用在文件中使用多少页面,交换文件的大小 = vm-page-size * vm-pages
vm-max-threads 4                  #用于执行 value 对象换入换出的工作线程数量。0 表示不使用工作线程（后面介绍)
```

redis 的 vm 在设计上为了保证 key 的查找速度，只会将 value 交换到 swap 文件中。所以如果是内存问题是由于太多 v

alue 很小的 key 造成 的，那么 vm 并不能解决。和 os 一样 redis 也是按页面来交换对象的。redis 规定同一个页面只能保存一个对象。但是一个对象可以保存在多个页面中。 在 redis 使用的内存没超过 vm-max-memory 之前是不会交换任何 value 的。当超过最大内存限制后，redis 会选择较老的对象。如果两个 对象一样老会优先交换比较大的对象，精确的公式 swappability = age*log(size_in_memory)。 对于 vm-page-size 的设置应该根据自己的应用将页面的大小设置为可以容纳大多数对象的大小。太大了会浪费磁盘空间，太小了会造成交换文件出现碎 片。对于交换文件中的每个页面，redis 会在内存中对应一个 1bit 值来记录页面的空闲状态。所以像上面配置中页面数量(vm-pages 134217728)会占用 16M 内存用来记录页面空闲状态。vm-max-threads 表示用做交换任务的线程数量。如果大于 0 推荐设为服务器的 cpu core 的数量。如果是 0 则交换过程在主线程进行。

参数配置讨论完后，在来简单介绍下 vm 是如何工作的，

当 vm-max-threads 设为 0 时(Blocking VM)

换出

主线程定期检查发现内存超出最大上限后，会直接已阻塞的方式,将选中的对象保存到 swap 文件中，并释放对象占用的内存,此过程会一直重复直到下面条件满足

1.内存使用降到最大限制以下

2.swap 文件满了

3.几乎全部的对象都被交换到磁盘了

换入

当有 client 请求 value 被换出的 key 时。主线程会以阻塞的方式从文件中加载对应的 value 对象，加载时此时会阻塞所以 client。然后处理 client 的请求

当 vm-max-threads 大于 0(Threaded VM)

换出

当主线程检测到使用内存超过最大上限，会将选中的要交换的对象信息放到一个队列中交由工作线程后台处理，主线程会继续处理 client 请求。

换入

如果有 client 请求的 key 被换出了，主线程先阻塞发出命令的 client,然后将加载对象的信息放到一个队列中，让工作线程去加载。加载完毕后工作线程通知主线程。主线程再执行 client 的命令。这种方式只阻塞请求 value 被换出 key 的 client

总 的来说 blocking vm 的方式总的性能会好一些，因为不需要线程同步，创建线程和恢复被阻塞的 client 等开销。但是也相应的牺牲了响应性。threaded vm 的方式主线程不会阻塞在磁盘 io 上，所以响应性更好。如果我们的应用不太经常发生换入换出，而且也不太在意有点延迟的话则推荐使用 blocking vm 的方式。关于 redis vm 的更详细介绍可以参考下面链接

http://antirez.com/post/redis-virtual-memory-story.html

http://redis.io/topics/internals-vm