Redis 教程

REmote DIctionary Server(Redis) 是一个由 Salvatore Sanfilippo 写的 key-value 存储系统。Redis 是一个开源的使用 ANSI C 语言编写、遵守 BSD 协议、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。它通常被称为数据结构服务器，因为值（value）可以是 字符串(String), 哈希(Hash), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型。

一. Redis 安装

从 redis 的官网 http://download.redis.io/ 下载源代码 redis-stable.tar.gz

```
$ tar xzvf redis-stable.tar.gz
$ cd redis-stable
$ make
```

make 之后就编译完成了。

```
$ cd src/
$ ./redis-server ../redis.conf # 启动完成
```

二. 基本数据结构

Key: 键

Redis 键命令用于管理 redis 的键。基本语法：

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

实例：

```
redis 127.0.0.1:6379> SET 0voice redis
OK
redis 127.0.0.1:6379> DEL 0voice
(integer) 1
```

Redis Keys 命令：

序号	命令及描述

1	DEL key 该命令用于在 key 存在时删除 key。
2	DUMP key 序列化给定 key ，并返回被序列化的值。
3	EXISTS key 检查给定 key 是否存在。
4	EXPIRE key seconds 为给定 key 设置过期时间，以秒计。
5	EXPIREAT key timestamp EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置过期时间。 不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。
6	PEXPIRE key milliseconds 设置 key 的过期时间以毫秒计。
7	PEXPIREAT key milliseconds-timestamp 设置 key 过期时间的时间戳(unix timestamp) 以毫秒计
8	KEYS pattern 查找所有符合给定模式(pattern)的 key 。
9	MOVE key db 将当前数据库的 key 移动到给定的数据库 db 当中。
10	PERSIST key 移除 key 的过期时间，key 将持久保持。
11	PTTL key 以毫秒为单位返回 key 的剩余的过期时间。

12	TTL key 以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。
13	RANDOMKEY 从当前数据库中随机返回一个 key 。
14	RENAME key newkey 修改 key 的名称
15	RENAMENX key newkey 仅当 newkey 不存在时，将 key 改名为 newkey 。
16	TYPE key 返回 key 所储存的值的类型。

String: 字符串

Redis 字符串数据类型的相关命令用于管理 redis 字符串值，基本语法如下：

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

实例：

```
redis 127.0.0.1:6379> SET zerovoicekey redis
OK
redis 127.0.0.1:6379> GET zerovoicekey
"redis"
```

Redis 字符串命令

序号	命令及描述
1	SET key value 设置指定 key 的值
2	GET key 获取指定 key 的值。

3	GETRANGE key start end 返回 key 中字符串值的子字符
4	GETSET key value 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。
5	GETBIT key offset 对 key 所储存的字符串值，获取指定偏移量上的位(bit)。
6	MGET key1 [key2..] 获取所有(一个或多个)给定 key 的值。
7	SETBIT key offset value 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。
8	SETEX key seconds value 将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。
9	SETNX key value 只有在 key 不存在时设置 key 的值。
10	SETRANGE key offset value 用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。
11	STRLEN key 返回 key 所储存的字符串值的长度。
12	MSET key value [key value ...] 同时设置一个或多个 key-value 对。
13	MSETNX key value [key value ...] 同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。

14	PSETEX key milliseconds value 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。
15	INCR key 将 key 中储存的数字值增一。
16	INCRBY key increment 将 key 所储存的值加上给定的增量值（increment） 。
17	INCRBYFLOAT key increment 将 key 所储存的值加上给定的浮点增量值（increment） 。
18	DECR key 将 key 中储存的数字值减一。
19	DECRBY key decrement key 所储存的值减去给定的减量值（decrement） 。
20	APPEND key value 如果 key 已经存在并且是一个字符串， APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。

Hash：散列表

Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。Redis 中每个 hash 可以存储 2^{32} - 1 键值对（40 多亿）。案例如下：

```
127.0.0.1:6379>  hmset kingvoice name "redis tutorial" likes 20 visitors 23000
OK
127.0.0.1:6379> hgetall kingvoice
1) "name"
```

```
2) "redis tutorial"
3) "likes"
4) "20"
5) "visitors"
6) "23000"
```

Redis Hash 命令

序号	命令及描述
1	HDEL key field1 [field2] 删除一个或多个哈希表字段
2	HEXISTS key field 查看哈希表 key 中，指定的字段是否存在。
3	HGET key field 获取存储在哈希表中指定字段的值。
4	HGETALL key 获取在哈希表中指定 key 的所有字段和值
5	HINCRBY key field increment 为哈希表 key 中的指定字段的整数值加上增量 increment 。
6	HINCRBYFLOAT key field increment 为哈希表 key 中的指定字段的浮点数值加上增量 increment 。
7	HKEYS key 获取所有哈希表中的字段
8	HLEN key 获取哈希表中字段的数量
9	HMGET key field1 [field2] 获取所有给定字段的值

10	HMSET key field1 value1 [field2 value2] 同时将多个 field-value (域-值)对设置到哈希表 key 中。
11	HSET key field value 将哈希表 key 中的字段 field 的值设为 value 。
12	HSETNX key field value 只有在字段 field 不存在时，设置哈希表字段的值。
13	HVALS key 获取哈希表中所有值
14	HSCAN key cursor [MATCH pattern] [COUNT count] 迭代哈希表中的键值对。

List: 列表

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）一个列表最多可以包含 $2^{32} - 1$ 个元素 (4294967295, 每个列表超过 40 亿个元素)。

实例：

```
127.0.0.1:6379> lpush kingkey redis
(integer) 1
127.0.0.1:6379> lpush kingkey memcached
(integer) 2
127.0.0.1:6379> lpush kingkey mongodb
(integer) 3
127.0.0.1:6379> lpush kingkey mysql
(integer) 4
127.0.0.1:6379> lrange kingkey 0 10
1) "mysql"
2) "mongodb"
3) "memcached"
4) "redis"
```

Redis List 列表命令

序号	命令及描述
1	BLPOP key1 [key2] timeout 移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
2	BRPOP key1 [key2] timeout 移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
3	BRPOPLPUSH source destination timeout 从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它；如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
4	LINDEX key index 通过索引获取列表中的元素
5	LINSERT key BEFORE|AFTER pivot value 在列表的元素前或者后插入元素
6	LLEN key 获取列表长度
7	LPOP key 移出并获取列表的第一个元素
8	LPUSH key value1 [value2] 将一个或多个值插入到列表头部
9	LPUSHX key value 将一个值插入到已存在的列表头部

10	LRANGE key start stop 获取列表指定范围内的元素
11	LREM key count value 移除列表元素
12	LSET key index value 通过索引设置列表元素的值
13	LTRIM key start stop 对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。
14	RPOP key 移除列表的最后一个元素，返回值为移除的元素。
15	RPOPLPUSH source destination 移除列表的最后一个元素，并将该元素添加到另一个列表并返回
16	RPUSH key value1 [value2] 在列表中添加一个或多个值
17	RPUSHX key value 为已存在的列表添加值

Set：集合

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。
集合中最大的成员数为 2^{32} - 1 (4294967295, 每个集合可存储 40 多亿个成员)。
实例：

```
127.0.0.1:6379> sadd 0voicekey redis
```

```
(integer) 1
127.0.0.1:6379> sadd 0voicekey mongodb
(integer) 1
127.0.0.1:6379> sadd 0voicekey mysql
(integer) 1
127.0.0.1:6379> sadd 0voicekey memcached
(integer) 1
127.0.0.1:6379> sadd 0voicekey memcached
(integer) 0
127.0.0.1:6379> smembers 0voicekey
1) "memcached"
2) "mysql"
3) "mongodb"
4) "redis"
```

Redis Set 集合命令

序号	命令及描述
1	SADD key member1 [member2] 向集合添加一个或多个成员
2	SCARD key 获取集合的成员数
3	SDIFF key1 [key2] 返回给定所有集合的差集
4	SDIFFSTORE destination key1 [key2] 返回给定所有集合的差集并存储在 destination 中
5	SINTER key1 [key2] 返回给定所有集合的交集
6	SINTERSTORE destination key1 [key2] 返回给定所有集合的交集并存储在 destination 中

7	SISMEMBER key member 判断 member 元素是否是集合 key 的成员
8	SMEMBERS key 返回集合中的所有成员
9	SMOVE source destination member 将 member 元素从 source 集合移动到 destination 集合
10	SPOP key 移除并返回集合中的一个随机元素
11	SRANDMEMBER key [count] 返回集合中一个或多个随机数
12	SREM key member1 [member2] 移除集合中一个或多个成员
13	SUNION key1 [key2] 返回所有给定集合的并集
14	SUNIONSTORE destination key1 [key2] 所有给定集合的并集存储在 destination 集合中
15	SSCAN key cursor [MATCH pattern] [COUNT count] 迭代集合中的元素

Sorted Set: 有序集合

Redis 有序集合和集合一样也是 string 类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。有序集合的成员是唯一的,但分数(score)却可以重复。集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。 集合中最大的成员数为 2^{32} - 1 (4294967295, 每个集合可存储 40 多亿个成员)。

实例：

```
127.0.0.1:6379> zadd zerokey 1 redis
(integer) 1
127.0.0.1:6379> zadd zerokey 2 mongodb
(integer) 1
127.0.0.1:6379> zadd zerokey 3 memcached
(integer) 1
127.0.0.1:6379> zrange zerokey 0 10
1) "redis"
2) "mongodb"
3) "memcached"
127.0.0.1:6379> zadd zerokey 4 mysql
(integer) 1
127.0.0.1:6379> zrange zerokey 0 10 withscores
1) "redis"
2) "1"
3) "mongodb"
4) "2"
5) "memcached"
6) "3"
7) "mysql"
8) "4"
```

序号	命令及描述
1	ZADD key score1 member1 [score2 member2] 向有序集合添加一个或多个成员，或者更新已存在成员的分数
2	ZCARD key 获取有序集合的成员数
3	ZCOUNT key min max 计算在有序集合中指定区间分数的成员数
4	ZINCRBY key increment member 有序集合中对指定成员的分数加上增量 increment

5	ZINTERSTORE destination numkeys key [key ...] 计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中
6	ZLEXCOUNT key min max 在有序集合中计算指定字典区间内成员数量
7	ZRANGE key start stop [WITHSCORES] 通过索引区间返回有序集合成指定区间内的成员
8	ZRANGEBYLEX key min max [LIMIT offset count] 通过字典区间返回有序集合的成员
9	ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] 通过分数返回有序集合指定区间内的成员
10	ZRANK key member 返回有序集合中指定成员的索引
11	ZREM key member [member ...] 移除有序集合中的一个或多个成员
12	ZREMRANGEBYLEX key min max 移除有序集合中给定的字典区间的所有成员
13	ZREMRANGEBYRANK key start stop 移除有序集合中给定的排名区间的所有成员
14	ZREMRANGEBYSCORE key min max 移除有序集合中给定的分数区间的所有成员
15	ZREVRANGE key start stop [WITHSCORES] 返回有序集中指定区间内的成员，通过索引，分数从高到底

16	ZREVRANGEBYSCORE key max min [WITHSCORES] 返回有序集中指定分数区间内的成员，分数从高到低排序
17	ZREVRANK key member 返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序
18	ZSCORE key member 返回有序集中，成员的分数值
19	ZUNIONSTORE destination numkeys key [key ...] 计算给定的一个或多个有序集的并集，并存储在新的 key 中
20	ZSCAN key cursor [MATCH pattern] [COUNT count] 迭代有序集合中的元素（包括元素成员和元素分值）

三. 操作命令

Redis HyperLogLog

Redis 在 2.8.9 版本添加了 HyperLogLog 结构。Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的。在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

实例：

```
127.0.0.1:6379> pfadd kingpfkey "redis"
(integer) 1
127.0.0.1:6379> pfadd kingpfkey "mongodb"
(integer) 1
127.0.0.1:6379> pfadd kingpfkey "mysql"
(integer) 1
127.0.0.1:6379> pfcount kingpfkey
```

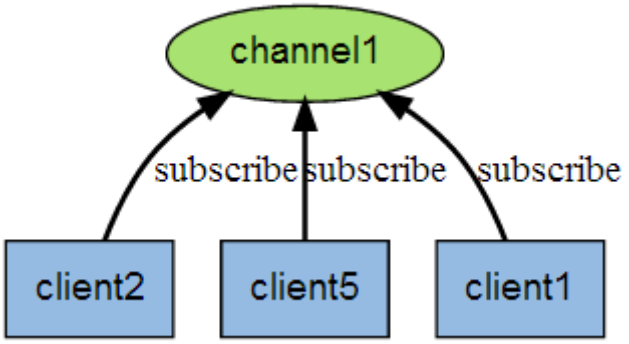
```
(integer) 3
```

HyperLogLog 命令：

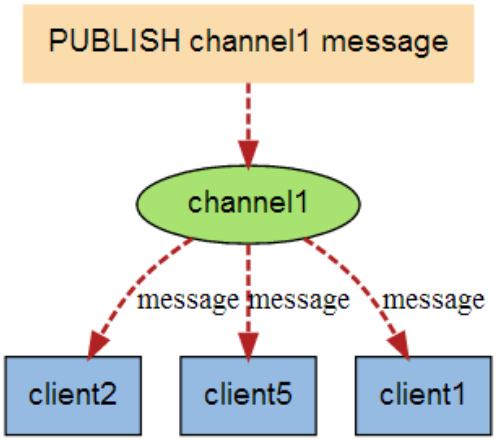
序号	命令及描述
1	PFADD key element [element ...] 添加指定元素到 HyperLogLog 中。
2	PFCOUNT key [key ...] 返回给定 HyperLogLog 的基数估算值。
3	PFMERGE destkey sourcekey [sourcekey ...] 将多个 HyperLogLog 合并为一个 HyperLogLog

Redis 发布订阅

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。下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



实例：在我们实例中我们创建了订阅频道名为 redisChat:

Subscribe 客户端 1

```
127.0.0.1:6379> subscribe redisChat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
1) "message"
2) "redisChat"
3) "Redis is a great caching technique"
1) "message"
2) "redisChat"
3) "Think you"
```

另外开启一个 redis 客户端，Subscribe 客户端 2

```
127.0.0.1:6379> subscribe redisChat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
1) "message"
2) "redisChat"
3) "Think you"
```

开启 Publish 客户端

```
127.0.0.1:6379> publish redisChat "Redis is a great caching technique"
(integer) 1
127.0.0.1:6379> publish redisChat "Think you"
(integer) 2
127.0.0.1:6379>
```

Redis 发布订阅命令：

序号	命令及描述
1	PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	SUBSCRIBE channel [channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

Redis 事务

Redis 事务可以一次执行多个命令， 并且带有以下三个重要的保证：

1> 批量操作在发送 EXEC 命令前被放入队列缓存。

2> 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。

3> 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

一个事务从开始到执行会经历以下三个阶段：

1> 开始事务。

2> 命令入队。

3> 执行事务。

实例：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set bookname "mastering c++"
QUEUED
127.0.0.1:6379> set bookname "0voice class"
QUEUED
127.0.0.1:6379> get bookname
QUEUED
127.0.0.1:6379> sadd tag "c++" "Programming" "Mastering Serires"
QUEUED
127.0.0.1:6379> smembers tag
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
3) "0voice class"
4) (integer) 3
5) 1) "c++"
   2) "Mastering Serires"
   3) "Programming"
127.0.0.1:6379>
```

redis 事务的相关命令：

序号	命令及描述
1	DISCARD 取消事务，放弃执行事务块内的所有命令。
2	EXEC 执行所有事务块内的命令。
3	MULTI 标记一个事务块的开始。
4	UNWATCH 取消 WATCH 命令对所有 key 的监视。

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

Redis 脚本

Redis 脚本使用 Lua 解释器来执行脚本。 Redis 2.6 版本通过内嵌支持 Lua 环境。执行脚本的常用命令为 EVAL。

Eval 命令的基本语法如下：

```
redis 127.0.0.1:6379> EVAL script numkeys key [key ...] arg [arg ...]
```

以下实例演示了 redis 脚本工作过程：

```
redis 127.0.0.1:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 fi
rst second

1) "key1"
2) "key2"
3) "first"
4) "second"
```

Redis 脚本命令

下表列出了 redis 脚本常用命令：

序号	命令及描述
1	EVAL script numkeys key [key ...] arg [arg ...] 执行 Lua 脚本。
2	EVALSHA sha1 numkeys key [key ...] arg [arg ...] 执行 Lua 脚本。

3	SCRIPT EXISTS script [script ...] 查看指定的脚本是否已经被保存在缓存当中。
4	SCRIPT FLUSH 从脚本缓存中移除所有脚本。
5	SCRIPT KILL 杀死当前正在运行的 Lua 脚本。
6	SCRIPT LOAD script 将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本。

Redis 安全

我们可以通过 redis 的配置文件设置密码参数，这样客户端连接到 redis 服务就需要密码验证，这样可以让你的 redis 服务更安全。

我们可以通过以下命令查看是否设置了密码验证：

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
```

默认情况下 requirepass 参数是空的，这就意味着你无需通过密码验证就可以连接到 redis 服务。

你可以通过以下命令来修改该参数：

```
127.0.0.1:6379> CONFIG set requirepass "0voice"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "0voice"
```

设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令。

AUTH 命令基本语法格式如下：

```
127.0.0.1:6379> AUTH password
```

```
127.0.0.1:6379> AUTH "0voice"
OK
127.0.0.1:6379> SET mykey "Test value"
OK
127.0.0.1:6379> GET mykey
"Test value"
```

Redis 数据备份与恢复

Redis SAVE 命令用于创建当前数据库的备份。

redis Save 命令基本语法如下：

```
127.0.0.1:6379> SAVE
```

实例

```
127.0.0.1:6379> SAVE
OK
```

该命令将在 redis 安装目录中创建 dump.rdb 文件。

恢复数据

如果需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可。获取 redis 目录可以使用 CONFIG 命令，如下所示：

```
 127.0.0.1:6379> CONFIG GET dir
1) "dir"
2) "/usr/local/redis/bin"
```

以上命令 CONFIG GET dir 输出的 redis 安装目录为 /usr/local/redis/bin。

Bgsave

创建 redis 备份文件也可以使用命令 BGSAVE，该命令在后台执行。

实例

```
127.0.0.1:6379> BGSAVE

Background saving started
```

Redis 性能测试

Redis 性能测试是通过同时执行多个命令实现的。

redis 性能测试的基本命令如下：

```
redis-benchmark [option] [option value]
```

注意：该命令是在 redis 的目录下执行的，而不是 redis 客户端的内部指令。

以下实例同时执行 10000 个请求来检测性能：

```
$ ./redis-benchmark -n 10000 -q
PING_INLINE: 78740.16 requests per second
PING_BULK: 76335.88 requests per second
SET: 77519.38 requests per second
GET: 83333.34 requests per second
INCR: 84033.61 requests per second
LPUSH: 70422.53 requests per second
RPUSH: 80000.00 requests per second
LPOP: 76923.08 requests per second
RPOP: 85470.09 requests per second
SADD: 78125.00 requests per second
HSET: 86206.90 requests per second
SPOP: 80000.00 requests per second
LPUSH (needed to benchmark LRANGE): 78740.16 requests per second
LRANGE_100 (first 100 elements): 71942.45 requests per second
LRANGE_300 (first 300 elements): 81967.21 requests per second
LRANGE_500 (first 450 elements): 83333.34 requests per second
LRANGE_600 (first 600 elements): 83333.34 requests per second
MSET (10 keys): 64102.56 requests per second
```

redis 性能测试工具可选参数如下所示：

序号	选项	描述	默认值
1	**-h**	指定服务器主机名	127.0.0.1
2	**-p**	指定服务器端口	6379

3	**-s**	指定服务器 socket	
4	**-c**	指定并发连接数	50
5	**-n**	指定请求数	10000
6	**-d**	以字节的形式指定 SET/GET 值的数据大小	2
7	**-k**	1=keep alive 0=reconnect	1
8	**-r**	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	**-P**	通过管道传输 <numreq> 请求	1
10	**-q**	强制退出 redis。仅显示 query/sec 值	
11	**--csv**	以 CSV 格式输出	
12	**-l**	生成循环，永久执行测试	
13	**-t**	仅运行以逗号分隔的测试命令列表。	
14	**-I**	Idle 模式。仅打开 N 个 idle 连接并等待。	

以下实例我们使用了多个参数来测试 redis 性能：

```
$ redis-benchmark -h 127.0.0.1 -p 6379 -t set,lpush -n 10000 -q

SET: 146198.83 requests per second
LPUSH: 145560.41 requests per second
```

以上实例中主机为 127.0.0.1，端口号为 6379，执行的命令为 set,lpush，请求数为 10000，通过 -q 参数让结果只显示每秒执行的请求数。

Redis 客户端连接

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

1. 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。

2. 然后为这个 socket 设置 TCP_NODELAY 属性，禁用 Nagle 算法
3. 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

最大连接数

在 Redis2.4 中，最大连接数是被直接硬编码在代码里面的，而在 2.6 版本中这个值变成可配置的。maxclients 的默认值是 10000，你也可以在 redis.conf 中对这个值进行修改。

```
config get maxclients

1) "maxclients"
2) "10000"
```

以下实例我们在服务启动时设置最大连接数为 100000：

```
redis-server --maxclients 100000
```

客户端命令

S.N.	命令	描述
1	**CLIENT LIST**	返回连接到 redis 服务的客户端列表
2	**CLIENT SETNAME**	设置当前连接的名称
3	**CLIENT GETNAME**	获取通过 CLIENT SETNAME 命令设置的服务名称
4	**CLIENT PAUSE**	挂起客户端连接，指定挂起的时间以毫秒计
5	**CLIENT KILL**	关闭客户端连接

Redis 管道技术

Redis 是一种基于客户端-服务端模型以及请求/响应协议的 TCP 服务。这意味着通常情况下一个请求会遵循以下步骤：

1> 客户端向服务端发送一个查询请求，并监听 Socket 返回，通常是以阻塞模式，等待服务端响应。

2> 服务端处理命令，并将结果返回给客户端。

Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

实例

查看 redis 管道，只需要启动 redis 实例并输入以下命令：

```
$(echo -en "PING\r\n SET kingkey redis\r\nGET kingkey\r\nINCR visitor\r\nINCR visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379

+PONG
+OK
redis
:1
:2
:3
```

以上实例中我们通过使用 PING 命令查看 redis 服务是否可用， 之后我们设置了 kingkey 的值为 redis，然后我们获取 kingkey 的值并使得 visitor 自增 3 次。在返回的结果中我们可以看到这些命令一次性向 redis 服务提交，并最终一次性读取所有服务端的响应

管道技术的优势

管道技术最显著的优势是提高了 redis 服务的性能。

一些测试数据

在下面的测试中，我们将使用 Redis 的 Ruby 客户端，支持管道技术特性，测试管道技术对速度的提升效果。

```
require 'rubygems'
require 'redis'
def bench(descr)
start = Time.now
yield
```

```
puts "#{descr} #{Time.now-start} seconds"
end
def without_pipelining
r = Redis.new
10000.times {
    r.ping
}
end
def with_pipelining
r = Redis.new
r.pipelined {
    10000.times {
        r.ping
    }
}
end
bench("without pipelining") {
    without_pipelining
}
bench("with pipelining") {
    with_pipelining
}
```

从处于局域网中的 Mac OS X 系统上执行上面这个简单脚本的数据表明，开启了管道操作后，往返延时已经被改善得相当低了。

```
without pipelining 1.185238 seconds
with pipelining 0.250783 seconds
```

如你所见，开启管道后，我们的速度效率提升了 5 倍。

Redis 分区

分区是分割数据到多个 Redis 实例的处理过程，因此每个实例只保存 key 的一个子集。

分区的优势

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

分区的不足

redis 的一些特性在分区方面表现的不是很好：

1> 涉及多个 key 的操作通常是不被支持的。举例来说，当两个 set 映射到不同的 redis 实例上时，你就不能对这两个 set 执行交集操作。

2> 涉及多个 key 的 redis 事务不能使用。

3> 当使用分区时，数据处理较为复杂，比如你需要处理多个 rdb/aof 文件，并且从多个实例和主机备份持久化文件。

4> 增加或删除容量也比较复杂。redis 集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做 presharding 的技术对此是有帮助的。

分区类型

Redis 有两种类型分区。 假设有 4 个 Redis 实例 R0，R1，R2，R3，和类似 user:1，user:2 这样的表示用户的多个 key，对既定的 key 有多种不同方式来选择这个 key 存放在哪个实例中。也就是说，有不同的系统来映射某个 key 到某个 Redis 服务。

范围分区

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的 Redis 实例。

比如，ID 从 0 到 10000 的用户会保存到实例 R0，ID 从 10001 到 20000 的用户会保存到 R1，以此类推。

这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对 Redis 来说并非是好的方法。

哈希分区

另外一种分区方法是 hash 分区。这对任何 key 都适用，也无需是 object_name:这种形式，像下面描述的一样简单：

- 用一个 hash 函数将 key 转换为一个数字，比如使用 crc32 hash 函数。对 key foobar 执行 crc32(foobar)会输出类似 93024922 的整数。

- 对这个整数取模，将其转化为 0-3 之间的数字，就可以将这个整数映射到 4 个 Redis 实例中的一个了。93024922 % 4 = 2，就是说 key foobar 应该被存到 R2 实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

四．C 语言使用 Redis

连接 Redis 服务

```
1. // 连接 Redis 服务
2.     redisContext *context = redisConnect("127.0.0.1", 6379);
3.     if (context == NULL || context->err) {
4.         if (context) {
5.             printf("%s\n", context->errstr);
6.         } else {
7.             printf("redisConnect error\n");
8.         }
9.         exit(EXIT_FAILURE);
10.    }
```

授权 Auth

```
redisReply *reply = redisCommand(context, "auth 0voice");
    printf("type : %d\n", reply->type);
    if (reply->type == REDIS_REPLY_STATUS) {
        /*SET str Hello World*/
        printf("auth ok\n");
    }
    freeReplyObject(reply);
```

Set Key Value

```
char *key = "str";
char *val = "Hello World";
/*SET key value */
reply = redisCommand(context, "SET %s %s", key, val);
printf("type : %d\n", reply->type);
if (reply->type == REDIS_REPLY_STATUS) {
    /*SET str Hello World*/
    printf("SET %s %s\n", key, val);
}
freeReplyObject(reply);
```

Get Key

```
// GET Key
    reply = redisCommand(context, "GET %s", key);
    if (reply->type == REDIS_REPLY_STRING) {
        /*GET str Hello World*/
        printf("GET str %s\n", reply->str);
        /*GET len 11*/
        printf("GET len %ld\n", reply->len);
    }
    freeReplyObject(reply);
```

APPEND Key Value

```
// APPEND key value
    char *append = " I am your GOD";
    reply = redisCommand(context, "APPEND %s %s", key, append);
    if (reply->type == REDIS_REPLY_INTEGER) {
        printf("APPEND %s %s \n", key, append);
    }
    freeReplyObject(reply);
    /*GET key*/
    reply = redisCommand(context, "GET %s", key);
    if (reply->type == REDIS_REPLY_STRING) {
        //GET Hello World I am your GOD
        printf("GET %s\n", reply->str);
    }
    freeReplyObject(reply);
```