# 目　录

# 关于版块

## 关于**redis**

简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。

优势

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。
- 支持异机复制

应用场景

- 存储 缓存 用的数据
  使用 Redis 进行存储的时候，我们需要从以下几个方面来考虑：
  1、业务数据常用吗？命中率如何？如果命中率很低，就没有必要写入缓存；
  2、该业务数据是读操作多，还是写操作多？如果写操作多，频繁需要写入数据库，也没有必要使用缓存；
  3、业务数据大小如何？如果要存储几百兆字节的文件，会给缓存带来很大的压力，这样也没有必要；
- 需要高速读/写的场合使用它快速读/写
  1、当一个请求到达服务器时，只是把业务数据在 Redis 上进行读写，而没有对数据库进行任何的操作，这样就能大大提高读写的速度，从而满足高速响应的需求；
  2、但是这些缓存的数据仍然需要持久化，也就是存入数据库之中，所以在一个请求操作完 Redis 的读/写之后，会去判断该高速读/写的业务是否结束，比方天猫双11、抢红包等，这个判断通常会在秒杀商品为0，红包金额为0时成立，如果不成立，则不会操作数据库；如果成立，则触发事件将 Redis 的缓存的数据以批量的形式一次性写入数据库，从而完成持久化的工作。

## 本版块维护人员

版主：子木

QQ：1242119478

交流Q群：526749756

# 安装部署

## 单点部署

服务器：test11 & ip：192.168.37.11
系统版本：centos 7.6
redis版本：redis 5.0.5

- 下载安装包：`wget http://download.redis.io/releases/redis-5.0.5.tar.gz`
- 解压安装包：`tar zxf redis-5.0.5.tar.gz -C /usr/local/`
- 进入解压后目录编译

```
[root@test11 ~]# cd /usr/local/redis-5.0.5/
[root@test11 ~]# make
```

- 启动redis服务：`nohup ./src/redis-server >/dev/null 2>&1 &`
- 简单测试redis

```
[root@test11 redis-5.0.5]# ./src/redis-cli
127.0.0.1:6379> set key01 HelloWord          #设置一个key
OK
127.0.0.1:6379>keys *                        #列出所有key
1) "key01"
127.0.0.1:6379> get key01                    #获取指定key的值
"HelloWord"
192.168.37.11:7006> del foo                  #删除key值
-> Redirected to slot [12182] located at 192.168.37.11:7003
(integer) 1
```

## 集群部署

简介

redis有三种集群方式：主从复制，哨兵模式和Redis-Cluster集群。Redis-Cluster集群模式是从redis 3.0开始提供了，目前redis最新版本已经到了5.0了，建议最新安装redis集群的直接采用Redis-Cluster集群模式。主从模式和哨兵模式缺点相对明显，这里就不介绍了，下面将着重介绍Redis-Cluster集群模式。

Redis-Cluster采用无中心结构,它的特点如下：

- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。
- 节点的fail是通过集群中超过半数的节点检测失效时才生效。

- 客户端与redis节点直连,不需要中间代理层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

工作方式

- 在redis的每一个节点上，都有这么两个东西，一个是插槽（slot），它的的取值范围是：0-16383。还有一个就是cluster，可以理解为是一个集群管理的插件。当我们的存取的key到达的时候，redis会根据crc16的算法得出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作。

- 为了保证高可用，redis-cluster集群引入了主从模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点。当其它主节点ping一个主节点A时，如果半数以上的主节点与A通信超时，那么认为主节点A宕机了。如果主节点A和它的从节点A1都宕机了，那么该集群就无法再提供服务了。

部署

要创建集群，我们首先要做的是在集群模式下运行一些空的Redis实例，本实验中，我们在一台test11服务器上搭建6个节点的Redis集群(实际生产环境，需要3台Linux服务器分布存放3个Master)，如下：

- 批量创建6个节点的文件夹，以端口号命名：`mkdir cluster-test/700{1,2,3,4,5,6} -p`
- 拷贝redis-server redis-cli到cluster-test文件夹下(上面单点部署生成的服务端与客户端)

```
[root@test11 redis-5.0.5]# cd src/
[root@test11 src]# cp redis-server redis-cli ../cluster-test/
```

- 拷贝redis.conf文件到cluster-test/7001下，并修改配置文件如下：

```
bind 192.168.37.11                          #连入主机的ip地址，不修改外部无法连入你的redis缓存服务器中
port 7001
dir ./7001/                                 #数据存放路径
pidfile /usr/local/redis-5.0.5/cluster-test/7001/redis_7001.pid
appendonly yes                              #指定是否在每次更新操作后进行日志记录
cluster-enabled yes                         #开启集群模式
cluster-config-file 7001/nodes-7001.conf    #存储此节点配置的文件路径，只是在Redis Cluster实例启动时生成，并在每次需要时更新
cluster-node-timeout 5000                   #集群超时时间，节点超过这个时间没反应就断定是宕机
daemonize yes                               #开启守护进程模式
```

- 把7001目录下的redis.conf文件复制到其余700x的目录下，并更改配置文件里的端口号

- 写一个启动集群实例redis的脚本

```
[root@test11 cluster-test]# cat start-cluster.sh
#! /bin/bash

cd /usr/local/redis-5.0.5/cluster-test/

for ((i=1;i<=6;i++))
do
    ./redis-server 700$i/redis.conf
done
```

- 给脚本赋执行权限，并启动脚本

```
[root@test11 cluster-test]# chmod +x start-cluster.sh
[root@test11 cluster-test]# ./start-cluster.sh
[root@test11 cluster-test]# ps aux|grep redis
root     19645  0.1  0.0 153892  2832 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7001 [cluster]
root     19647  0.1  0.0 153892  2832 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7002 [cluster]
root     19649  0.1  0.0 156964  2856 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7003 [cluster]
root     19654  0.1  0.0 156964  2856 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7004 [cluster]
root     19665  0.1  0.0 153892  2836 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7005 [cluster]
root     19667  0.1  0.0 153892  2836 ?        Ssl  16:48   0:00 ./redis
-server 192.168.37.11:7006 [cluster]
```

- 创建集群，在redis 5版本后创建很简单，因为可以用redis-cli直接创建集群，对于Redis版本3或4，有一个名为redis-trib.rb非常相似的工具，具体安装办法请另行搜索文档，集群创建如下：

```
[root@test11 cluster-test]# ./redis-cli --cluster create 192.168.37.11:7
001 192.168.37.11:7002 192.168.37.11:7003 192.168.37.11:7004 192.168.37.
11:7005 192.168.37.11:7006 --cluster-replicas 1

#--cluster-replicas 1 意味着我们希望每个创建的主服务器都有一个从服 其他参数是我要
用于创建新集群的实例的地址列表
#按照提示输入yes,等一会后看到屏幕输出 “[OK] All 16384 slots covered.”，即证明集
群创建成功
```

- 简单测试一下Redis Cluster,以下是使用redis-cli命令行的交互示例

```
[root@test11 cluster-test]# ./redis-cli -h 192.168.37.11 -c -p 7001
192.168.37.11:7001> set foo bar
-> Redirected to slot [12182] located at 192.168.37.11:7003
OK
```

```
192.168.37.11:7003> set hello world
-> Redirected to slot [866] located at 192.168.37.11:7001
OK
192.168.37.11:7001> get foo
-> Redirected to slot [12182] located at 192.168.37.11:7003
"bar"
192.168.37.11:7003> get hello
-> Redirected to slot [866] located at 192.168.37.11:7001
"world"
192.168.37.11:7001>

#退出redis-cli命令行，并把7001的redis关掉，再进去看下原来存放在7001的节点hello的key值，如下：

192.168.37.11:7001> quit
[root@test11 cluster-test]# ./redis-cli -h 192.168.37.11 -c -p 7001 shut down
[root@test11 cluster-test]# ps aux|grep redis            #7001已经被关掉了
root     22600  0.1  0.0 156964  3344 ?        Ssl  7月10   1:15 ./redis-server 192.168.37.11:7002 [cluster]
root     22605  0.1  0.0 159012  3344 ?        Ssl  7月10   1:19 ./redis-server 192.168.37.11:7003 [cluster]
root     22610  0.1  0.0 161060  3384 ?        Ssl  7月10   1:22 ./redis-server 192.168.37.11:7004 [cluster]
root     22615  0.1  0.0 161060  3372 ?        Ssl  7月10   1:23 ./redis-server 192.168.37.11:7005 [cluster]
root     22620  0.1  0.0 161060  3384 ?        Ssl  7月10   1:23 ./redis-server 192.168.37.11:7006 [cluster]
root     29738  0.0  0.0 112728   988 pts/0    R+   10:14   0:00 grep --color=auto redis
[root@test11 cluster-test]# ./redis-cli -h 192.168.37.11 -c -p 7002
192.168.37.11:7002> get hello
-> Redirected to slot [866] located at 192.168.37.11:7004      #原来存放在7001的key值存放到7004上了
"world"

#重新启动一下7001节点
[root@test11 cluster-test]# ./redis-server 7001/redis.conf

#再查看一下节点状态
192.168.37.11:7004> cluster nodes
bcba724b6e97da7cd7c650ace2b02ad68d0cb75a 192.168.37.11:7002@17002 master - 0 1562811393000 2 connected 5461-10922
fc48836459091a72854e9334b2cb946ff63a0e9b 192.168.37.11:7003@17003 master - 0 1562811393000 3 connected 10923-16383
e3d3ec5d1b4b0616ffdcee93d0c5ad4114f84efe 192.168.37.11:7001@17001 slave
fb9a846bb1f82622c4a728bfe4e2ad6749ce8d19 0 1562811392002 7 connected
a015b4fb0c20ad986abaa91d7d6a803171eacf75 192.168.37.11:7005@17005 slave
```

```
bcba724b6e97da7cd7c650ace2b02ad68d0cb75a 0 1562811392505 5 connected
6ce44274e68c47446040ead84b8e077f7e3890ea 192.168.37.11:7006@17006 slave
fc48836459091a72854e9334b2cb946ff63a0e9b 0 1562811393005 6 connected
fb9a846bb1f82622c4a728bfe4e2ad6749ce8d19 192.168.37.11:7004@17004 myself
,master - 0 1562811391000 7 connected 0-5460    #原来7001是master节点的，
由于shutdown后，7004节点由slave节点变为master节点
```

管理集群

- 添加新节点
  创建一个7007的空白实例节点，创建步骤参考上面，启动节点后，把节点加入集群

```
#使用add-node命令将新节点的地址指定为第一个参数，并将集群中***随机***存在节点的地址
指定为第二个参数
[root@test11 cluster-test]# ./redis-cli --cluster add-node 192.168.37.11
:7007 192.168.37.11:7002

#进入集群，查看节点状态：
[root@test11 cluster-test]# ./redis-cli -h 192.168.37.11 -c -p 7007
192.168.37.11:7007> cluster nodes
fb9a846bb1f82622c4a728bfe4e2ad6749ce8d19 192.168.37.11:7004@17004 master
 - 0 1562815745275 7 connected 0-5460
fc48836459091a72854e9334b2cb946ff63a0e9b 192.168.37.11:7003@17003 master
 - 0 1562815746585 3 connected 10923-16383
bcba724b6e97da7cd7c650ace2b02ad68d0cb75a 192.168.37.11:7002@17002 master
 - 0 1562815746281 2 connected 5461-10922
a015b4fb0c20ad986abaa91d7d6a803171eacf75 192.168.37.11:7005@17005 slave
bcba724b6e97da7cd7c650ace2b02ad68d0cb75a 0 1562815744572 2 connected
e3d3ec5d1b4b0616ffdcee93d0c5ad4114f84efe 192.168.37.11:7001@17001 slave
fb9a846bb1f82622c4a728bfe4e2ad6749ce8d19 0 1562815745000 7 connected
6ce44274e68c47446040ead84b8e077f7e3890ea 192.168.37.11:7006@17006 slave
fc48836459091a72854e9334b2cb946ff63a0e9b 0 1562815746080 3 connected
335c44207c90db737d2848d0b302e751b10c8fab 192.168.37.11:7007@17007 myself
,master - 0 1562815746000 0 connected
```

- 将新副本分配给特定的主副本，如下将7007节点设置为7002节点的从节点

```
./redis-cli --cluster add-node 192.168.37.11:7007 192.168.37.11:7002 --c
luster-slave --cluster-master-id bcba724b6e97da7cd7c650ace2b02ad68d0cb75
a
```

- 删除节点,例下面将删除新添加的7007节点
  命令：redis-cli - -cluster del-node 127.0.0.1:7001 “node-id”

```
[root@test11 cluster-test]# ./redis-cli --cluster del-node 192.168.37.11
:7001 335c44207c90db737d2848d0b302e751b10c8fab
```

# redis配置

**redis5.0**配置文件参数说明,转自：**Super_RD**

```
#是否在后台执行，yes：后台运行；no：不是后台运行
daemonize yes

#是否开启保护模式，默认开启。要是配置里没有指定bind和密码。开启该参数后，redis只会本地进行访问，拒绝外部访问。
protected-mode yes

#redis的进程文件
pidfile /var/run/redis/redis-server.pid

#redis监听的端口号。
port 6379

#此参数确定了TCP连接中已完成队列(完成三次握手之后)的长度， 当然此值必须不大于Linux系统定义的/proc/sys/net/core/somaxconn值，默认是511，而Linux的默认参数值是128。当系统并发量大并且客户端速度缓慢的时候，可以将这二个参数一起参考设定。该内核参数默认值一般是128，对于负载很大的服务程序来说大大的不够。一般会将它修改为2048或者更大。在/etc/sysctl.conf中添加:net.core.somaxconn = 2048，然后在终端中执行sysctl -p。
tcp-backlog 511

#指定 redis 只接收来自于该 IP 地址的请求，如果不进行设置，那么将处理所有请求
bind 127.0.0.1

#配置unix socket来让redis支持监听本地连接。
# unixsocket /var/run/redis/redis.sock

#配置unix socket使用文件的权限
# unixsocketperm 700

# 此参数为设置客户端空闲超过timeout，服务端会断开连接，为0则服务端不会主动断开连接，不能小于0。
timeout 0

#tcp keepalive参数。如果设置不为0，就使用配置tcp的SO_KEEPALIVE值，使用keepalive有两个好处:检测挂掉的对端。降低中间设备出问题而导致网络看似连接却已经与对端端口的问题。在Linux内核中，设置了keepalive，redis会定时给对端发送ack。检测到对端关闭需要两倍的设置值。
tcp-keepalive 0

#指定了服务端日志的级别。级别包括：debug（很多信息，方便开发、测试），verbose（许多有用的信息，但是没有debug级别信息多），notice（适当的日志级别，适合生产环境），warn（只有非常重要的信息）
```

```
loglevel notice

#指定了记录日志的文件。空字符串的话，日志会打印到标准输出设备。后台运行的redis标准输出是/dev/null。
logfile /var/log/redis/redis-server.log

#是否打开记录syslog功能
# syslog-enabled no

#syslog的标识符。
# syslog-ident redis

#日志的来源、设备
# syslog-facility local0

#数据库的数量，默认使用的数据库是DB 0。可以通过SELECT命令选择一个db
databases 16

# redis是基于内存的数据库，可以通过设置该值定期写入磁盘。
# 注释所有save行则停止rdb持久化
# 900秒（15分钟）内至少1个key值改变（则进行数据库保存--持久化）
# 300秒（5分钟）内至少10个key值改变（则进行数据库保存--持久化）
# 60秒（1分钟）内至少10000个key值改变（则进行数据库保存--持久化）
save 900 1
save 300 10
save 60 10000

#当RDB持久化出现错误后，是否依然进行继续进行工作，yes：不能进行工作，no：可以继续进行工作，可以通过info中的rdb_last_bgsave_status了解RDB持久化是否有错误
stop-writes-on-bgsave-error yes

#使用压缩rdb文件，rdb文件压缩使用LZF压缩算法，yes：压缩，但是需要一些cpu的消耗。no：不压缩，需要更多的磁盘空间
rdbcompression yes

#是否校验rdb文件。从rdb格式的第五个版本开始，在rdb文件的末尾会带上CRC64的校验和。这跟有利于文件的容错性，但是在保存rdb文件的时候，会有大概10%的性能损耗，所以如果你追求高性能，可以关闭该配置。
rdbchecksum yes

#rdb文件的名称
dbfilename dump.rdb

#数据目录，数据库的写入会在这个目录。rdb、aof文件也会写在这个目录
dir /var/lib/redis



################ 主从复制 ################
```

```
#复制选项，slave复制对应的master。
# slaveof <masterip> <masterport>

#如果master设置了requirepass，那么slave要连上master，需要有master的密码才行。masterauth就是用来配置master的密码，这样可以在连上master后进行认证。
# masterauth <master-password>

#当从库同主机失去连接或者复制正在进行，从机库有两种运行方式：1) 如果slave-serve-stale-data设置为yes(默认设置)，从库会继续响应客户端的请求。2) 如果slave-serve-stale-data设置为no，除去INFO和SLAVOF命令之外的任何请求都会返回一个错误”SYNC with master in progress”。
slave-serve-stale-data yes

#作为从服务器，默认情况下是只读的（yes），可以修改成NO，用于写（不建议）。
slave-read-only yes

#是否使用socket方式复制数据。目前redis复制提供两种方式，disk和socket。如果新的slave连上来或者重连的slave无法部分同步，就会执行全量同步，master会生成rdb文件。有2种方式：disk方式是master创建一个新的进程把rdb文件保存到磁盘，再把磁盘上的rdb文件传递给slave。socket是master创建一个新的进程，直接把rdb文件以socket的方式发给slave。disk方式的时候，当一个rdb保存的过程中，多个slave都能共享这个rdb文件。socket的方式就的一个个slave顺序复制。在磁盘速度缓慢，网速快的情况下推荐用socket方式。
repl-diskless-sync no

#diskless复制的延迟时间，防止设置为0。一旦复制开始，节点不会再接收新slave的复制请求直到下一个rdb传输。所以最好等待一段时间，等更多的slave连上来。
repl-diskless-sync-delay 5

#slave根据指定的时间间隔向服务器发送ping请求。时间间隔可以通过 repl_ping_slave_period 来设置，默认10秒。
# repl-ping-slave-period 10

#复制连接超时时间。master和slave都有超时时间的设置。master检测到slave上次发送的时间超过repl-timeout，即认为slave离线，清除该slave信息。slave检测到上次和master交互的时间超过repl-timeout，则认为master离线。需要注意的是repl-timeout需要设置一个比repl-ping-slave-period更大的值，不然会经常检测到超时。
# repl-timeout 60

#是否禁止复制tcp链接的tcp nodelay参数，可传递yes或者no。默认是no，即使用tcp nodelay。如果master设置了yes来禁止tcp nodelay设置，在把数据复制给slave的时候，会减少包的数量和更小的网络带宽。但是这也可能带来数据的延迟。默认我们推荐更小的延迟，但是在数据量传输很大的场景下，建议选择yes。
repl-disable-tcp-nodelay no

#复制缓冲区大小，这是一个环形复制缓冲区，用来保存最新复制的命令。这样在slave离线的时候，不需要完全复制master的数据，如果可以执行部分同步，只需要把缓冲区的部分数据复制给slave，就能恢复正常复制状态。缓冲区的大小越大，slave离线的时间可以更长，复制缓冲区只
```

```
有在有slave连接的时候才分配内存。没有slave的一段时间，内存会被释放出来，默认1m。
# repl-backlog-size 5mb

#master没有slave一段时间会释放复制缓冲区的内存，repl-backlog-ttl用来设置该时间长度。单位为秒。
# repl-backlog-ttl 3600

#当master不可用，Sentinel会根据slave的优先级选举一个master。最低的优先级的slave，当选master。而配置成0，永远不会被选举。
slave-priority 100

#redis提供了可以让master停止写入的方式，如果配置了min-slaves-to-write，健康的slave的个数小于N，mater就禁止写入。master最少得有多少个健康的slave存活才能执行写命令。这个配置虽然不能保证N个slave都一定能接收到master的写操作，但是能避免没有足够健康的slave的时候，master不能写入来避免数据丢失。设置为0是关闭该功能。
# min-slaves-to-write 3

#延迟小于min-slaves-max-lag秒的slave才认为是健康的slave。
# min-slaves-max-lag 10

# 设置1或另一个设置为0禁用这个特性。
# Setting one or the other to 0 disables the feature.
# By default min-slaves-to-write is set to 0 (feature disabled) and
# min-slaves-max-lag is set to 10.


################ 安全相关 ###############

#requirepass配置可以让用户使用AUTH命令来认证密码，才能使用其他命令。这让redis可以使用在不受信任的网络中。为了保持向后的兼容性，可以注释该命令，因为大部分用户也不需要认证。使用requirepass的时候需要注意，因为redis太快了，每秒可以认证15w次密码，简单的密码很容易被攻破，所以最好使用一个更复杂的密码。注意只有密码没有用户名。
# requirepass foobared

#把危险的命令给修改成其他名称。比如CONFIG命令可以重命名为一个很难被猜到的命令，这样用户不能使用，而内部工具还能接着使用。
# rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52

#设置成一个空的值，可以禁止一个命令
# rename-command CONFIG ""


################ 进程限制相关 ###############

# 设置能连上redis的最大客户端连接数量。默认是10000个客户端连接。由于redis不区分连接是客户端连接还是内部打开文件或者和slave连接等，所以maxclients最小建议设置到32。如果超过了maxclients，redis会给新的连接发送'max number of clients reached'，并关闭连接。
```

```
# maxclients 10000

#redis配置的最大内存容量。当内存满了，需要配合maxmemory-policy策略进行处理。注意slave的输出缓冲区是不计算在maxmemory内的。所以为了防止主机内存使用完，建议设置的maxmemory需要更小一些。
# maxmemory <bytes>

#内存容量超过maxmemory后的处理策略。
#volatile-lru：利用LRU算法移除设置过过期时间的key。
#volatile-random：随机移除设置过过期时间的key。
#volatile-ttl：移除即将过期的key，根据最近过期时间来删除（辅以TTL）
#allkeys-lru：利用LRU算法移除任何key。
#allkeys-random：随机移除任何key。
#noeviction：不移除任何key，只是返回一个写错误。
#上面的这些驱逐策略，如果redis没有合适的key驱逐，对于写命令，还是会返回错误。redis将不再接收写请求，只接收get请求。写命令包括：set setnx setex append incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby getset mset msetnx exec sort。
# maxmemory-policy noeviction

#lru检测的样本数。使用lru或者ttl淘汰算法，从需要淘汰的列表中随机选择sample个key，选出闲置时间最长的key移除。
# maxmemory-samples 5


################ APPEND ONLY 持久化方式 ################

#默认redis使用的是rdb方式持久化，这种方式在许多应用中已经足够用了。但是redis如果中途宕机，会导致可能有几分钟的数据丢失，根据save来策略进行持久化，Append Only File是另一种持久化方式，可以提供更好的持久化特性。Redis会把每次写入的数据在接收后都写入 appendonly.aof 文件，每次启动时Redis都会先把这个文件的数据读入内存里，先忽略RDB文件。
appendonly no

#aof文件名
appendfilename "appendonly.aof"

#aof持久化策略的配置
#no表示不执行fsync，由操作系统保证数据同步到磁盘，速度最快。
#always表示每次写入都执行fsync，以保证数据同步到磁盘。
#everysec表示每秒执行一次fsync，可能会导致丢失这1s数据。
appendfsync everysec

# 在aof重写或者写入rdb文件的时候，会执行大量IO，此时对于everysec和always的aof模式来说，执行fsync会造成阻塞过长时间，no-appendfsync-on-rewrite字段设置为默认设置为no。如果对延迟要求很高的应用，这个字段可以设置为yes，否则还是设置为no，这样对持久化特性来说这是更安全的选择。设置为yes表示rewrite期间对新写操作不fsync,暂时存在内存中,等
```

```
rewrite完成后再写入，默认为no，建议yes。Linux的默认fsync策略是30秒。可能丢失30秒
数据。
no-appendfsync-on-rewrite no

#aof自动重写配置。当目前aof文件大小超过上一次重写的aof文件大小的百分之多少进行重写，
即当aof文件增长到一定大小的时候Redis能够调用bgrewriteaof对日志文件进行重写。当前AO
F文件大小是上次日志重写得到AOF文件大小的二倍（设置为100）时，自动启动新的日志重写过程
。
auto-aof-rewrite-percentage 100
#设置允许重写的最小aof文件大小，避免了达到约定百分比但尺寸仍然很小的情况还要重写
auto-aof-rewrite-min-size 64mb

#aof文件可能在尾部是不完整的，当redis启动的时候，aof文件的数据被载入内存。重启可能发
生在redis所在的主机操作系统宕机后，尤其在ext4文件系统没有加上data=ordered选项（re
dis宕机或者异常终止不会造成尾部不完整现象。）出现这种现象，可以选择让redis退出，或者
导入尽可能多的数据。如果选择的是yes，当截断的aof文件被导入的时候，会自动发布一个log
给客户端然后load。如果是no，用户必须手动redis-check-aof修复AOF文件才可以。
aof-load-truncated yes


################ LUA SCRIPTING ###############

# 如果达到最大时间限制（毫秒），redis会记个log，然后返回error。当一个脚本超过了最大
时限。只有SCRIPT KILL和SHUTDOWN NOSAVE可以用。第一个可以杀没有调write命令的东西
。要是已经调用了write，只能用第二个命令杀。
lua-time-limit 5000


################ 集群相关 ###############

#集群开关，默认是不开启集群模式。
# cluster-enabled yes

#集群配置文件的名称，每个节点都有一个集群相关的配置文件，持久化保存集群的信息。这个文
件并不需要手动配置，这个配置文件有Redis生成并更新，每个Redis集群节点需要一个单独的配
置文件，请确保与实例运行的系统中配置文件名称不冲突
# cluster-config-file nodes-6379.conf

#节点互连超时的阀值。集群节点超时毫秒数
# cluster-node-timeout 15000

#在进行故障转移的时候，全部slave都会请求申请为master，但是有些slave可能与master断
开连接一段时间了，导致数据过于陈旧，这样的slave不应该被提升为master。该参数就是用来
判断slave节点与master断线的时间是否过长。判断方法是：
#比较slave断开连接的时间和(node-timeout * slave-validity-factor) + repl-pin
g-slave-period
#如果节点超时时间为三十秒, 并且slave-validity-factor为10,假设默认的repl-ping-s
lave-period是10秒，即如果超过310秒slave将不会尝试进行故障转移
```

```
# cluster-slave-validity-factor 10

#master的slave数量大于该值，slave才能迁移到其他孤立master上，如这个参数若被设为2，那么只有当一个主节点拥有2 个可工作的从节点时，它的一个从节点会尝试迁移。
# cluster-migration-barrier 1

#默认情况下，集群全部的slot有节点负责，集群状态才为ok，才能提供服务。设置为no，可以在slot没有全部分配的时候提供服务。不建议打开该配置，这样会造成分区的时候，小分区的master一直在接受写请求，而造成很长时间数据不一致。
# cluster-require-full-coverage yes


################ SLOW LOG 慢查询日志 ################

###slog log是用来记录redis运行中执行比较慢的命令耗时。当命令的执行超过了指定时间，就记录在slow log中，slog log保存在内存中，所以没有IO操作。
#执行时间比slowlog-log-slower-than大的请求记录到slowlog里面，单位是微秒，所以10000000就是1秒。注意，负数时间会禁用慢查询日志，而0则会强制记录所有命令。
slowlog-log-slower-than 10000

#慢查询日志长度。当一个新的命令被写进日志的时候，最老的那个记录会被删掉。这个长度没有限制。只要有足够的内存就行。你可以通过 SLOWLOG RESET 来释放内存。
slowlog-max-len 128

################ 延迟监控 ################
#延迟监控功能是用来监控redis中执行比较缓慢的一些操作，用LATENCY打印redis实例在跑命令时的耗时图表。只记录大于等于下边设置的值的操作。0的话，就是关闭监视。默认延迟监控功能是关闭的，如果你需要打开，也可以通过CONFIG SET命令动态设置。
latency-monitor-threshold 0

################ EVENT NOTIFICATION 订阅通知 ################
#键空间通知使得客户端可以通过订阅频道或模式，来接收那些以某种方式改动了 Redis 数据集的事件。因为开启键空间通知功能需要消耗一些 CPU ，所以在默认配置下，该功能处于关闭状态。
#notify-keyspace-events 的参数可以是以下字符的任意组合，它指定了服务器该发送哪些类型的通知：
##K 键空间通知，所有通知以 __keyspace@__ 为前缀
##E 键事件通知，所有通知以 __keyevent@__ 为前缀
##g DEL 、 EXPIRE 、 RENAME 等类型无关的通用命令的通知
##$ 字符串命令的通知
##l 列表命令的通知
##s 集合命令的通知
##h 哈希命令的通知
##z 有序集合命令的通知
##x 过期事件：每当有过期键被删除时发送
##e 驱逐(evict)事件：每当有键因为 maxmemory 政策而被删除时发送
##A 参数 g$lshzxe 的别名
#输入的参数中至少要有一个 K 或者 E，否则的话，不管其余的参数是什么，都不会有任何 通知
```

```
被分发。详细使用可以参考http://redis.io/topics/notifications

notify-keyspace-events ""

############### ADVANCED CONFIG 高级配置 ###############
#数据量小于等于hash-max-ziplist-entries的用ziplist，大于hash-max-ziplist-ent
ries用hash
hash-max-ziplist-entries 512
#value大小小于等于hash-max-ziplist-value的用ziplist，大于hash-max-ziplist-v
alue用hash。
hash-max-ziplist-value 64

#数据量小于等于list-max-ziplist-entries用ziplist，大于list-max-ziplist-entr
ies用list。
list-max-ziplist-entries 512
#value大小小于等于list-max-ziplist-value的用ziplist，大于list-max-ziplist-v
alue用list。
list-max-ziplist-value 64

#数据量小于等于set-max-intset-entries用iniset，大于set-max-intset-entries用s
et。
set-max-intset-entries 512

#数据量小于等于zset-max-ziplist-entries用ziplist，大于zset-max-ziplist-entr
ies用zset。
zset-max-ziplist-entries 128
#value大小小于等于zset-max-ziplist-value用ziplist，大于zset-max-ziplist-val
ue用zset。
zset-max-ziplist-value 64

#value大小小于等于hll-sparse-max-bytes使用稀疏数据结构（sparse），大于hll-spar
se-max-bytes使用稠密的数据结构（dense）。一个比16000大的value是几乎没用的，建议的
value大概为3000。如果对CPU要求不高，对空间要求较高的，建议设置到10000左右。
hll-sparse-max-bytes 3000

#Redis将在每100毫秒时使用1毫秒的CPU时间来对redis的hash表进行重新hash，可以降低内
存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受Redis时不时的对请求有2
毫秒的延迟的话，把这项配置为no。如果没有这么严格的实时性要求，可以设置为yes，以便能够
尽可能快的释放内存。
activerehashing yes

##对客户端输出缓冲进行限制可以强迫那些不从服务器读取数据的客户端断开连接，用来强制关闭
传输缓慢的客户端。
#对于normal client，第一个0表示取消hard limit，第二个0和第三个0表示取消soft lim
it，normal client默认取消限制，因为如果没有寻问，他们是不会接收数据的。
client-output-buffer-limit normal 0 0 0
#对于slave client和MONITER client，如果client-output-buffer一旦超过256mb，又
或者超过64mb持续60秒，那么服务器就会立即断开客户端连接。
```

```
client-output-buffer-limit slave 256mb 64mb 60
#对于pubsub client，如果client-output-buffer一旦超过32mb，又或者超过8mb持续60秒，那么服务器就会立即断开客户端连接。
client-output-buffer-limit pubsub 32mb 8mb 60

#redis执行任务的频率为1s除以hz。
hz 10

#在aof重写的时候，如果打开了aof-rewrite-incremental-fsync开关，系统会每32MB执行一次fsync。这对于把文件写入磁盘是有帮助的，可以避免过大的延迟峰值。
aof-rewrite-incremental-fsync yes
```

**Redis**动态设置

- 获取所有的配置项，如下

```
192.168.37.11:7006> config get *
  1) "dbfilename"
  2) "dump.rdb"
  3) "requirepass"
  4) ""
  5) "masterauth"
  6) ""
  7) "cluster-announce-ip"
  8) ""
   .
   .
   .
```

- 获取指定的配置项，如下

```
192.168.37.11:7006> config get loglevel
1) "loglevel"
2) "notice
```

- 修改配置项

```
192.168.37.11:7006> config set loglevel "warning"
OK
192.168.37.11:7006> config get loglevel
1) "loglevel"
2) "warning"
#如果需要把更改后的配置项写到redis.conf文件下，需要执行下面命令：
192.168.37.11:7006> config rewrite
OK
```

**Redis**安全

以下只是简单的设置一下redis密码，如果redis需要设置密码的，请设置复杂度高的密码：

```
127.0.0.1:6379> config set requirepass "12345"
OK
127.0.0.1:6379> config get requirepass
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth "12345"
OK
127.0.0.1:6379> config get requirepass
1) "requirepass"
2) "12345"
```

**Redis**慢查询

- 由上面配置文件可知，默认参数 slowlog-log-slower-than 10000 （微秒），即超过10ms(毫秒)的查询才会被记录到slowlog
- slowlog-max-len表示慢查询最大的条数，当slowlog超过设定的最大值后，会将最早的slowlog删除，是个FIFO队列
- 因为redis的查询太快了，我们需要把慢查询的时间再调小一下，下面是模拟慢查询的例子

```
192.168.37.11:7003> config set slowlog-log-slower-than 10
OK
192.168.37.11:7003> slowlog get                 #查询所有条目
1) 1) (integer) 1                               #慢查询日志的标识id
   2) (integer) 1563248180                      #发生时间戳
   3) (integer) 16                              #命令耗时
   4) 1) "slowlog"                              #执行命令和参数
      2) "get"
   5) "192.168.37.11:59140"
   6) ""
2) 1) (integer) 0
   2) (integer) 1563248170
   3) (integer) 11
   4) 1) "config"
      2) "set"
      3) "slowlog-log-slower-than"
      4) "10"
   5) "192.168.37.11:59140"
   6) ""
192.168.37.11:7003> slowlog get 1               #查询指定条目数
1) 1) (integer) 4
   2) (integer) 1563248488
   3) (integer) 24
   4) 1) "REPLCONF"
      2) "ACK"
      3) "691025"
```

```
   5) "192.168.37.11:34695"
   6) ""
192.168.37.11:7003> slowlog len                    #查询条目总数
(integer) 2
192.168.37.11:7003> slowlog reset                  #清理所有条目
OK
192.168.37.11:7003> slowlog len
(integer) 0
192.168.37.11:7003> slowlog get
(empty list or set
```

持久化详解，转自：**http://redisdoc.com/topic/persistence.html**

**Redis** 提供了多种不同级别的持久化方式：

- RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。
- AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。
- Redis 还可以同时使用 AOF 持久化和 RDB 持久化。 在这种情况下， 当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。
- 你甚至可以关闭持久化功能，让数据只在服务器运行时存在。

**RDB** 的优点

- RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。
- RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

**RDB** 的缺点

- 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。
- 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

### AOF 的优点

- 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
- AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。
- AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

### AOF 的缺点

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
- AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH source destination timeout 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

### RDB 和 AOF ，应该用哪一个？

- 一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。
- 如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。
- 有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快， 除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug 。

### RDB和AOF如何开启

- redis是默认开启RDB持久化的，注释所有save行则停止rdb持久化，参考上面配置文件说明
- 修改redis配置文件，开启如下：appendonly yes，具体参考上面配置文件说明

# redis优化

**redis**优化

- 尽量使用ssd或者高性能磁盘，提高持久化到磁盘的效率
- 控制Redis实例最大可用内存，建议设置每实例内存为2G
- 不要让你的redis所在的机器物理内存使用超过实际内存总量的3/5
- 优先使用物理机或者高效支持fork操作的虚拟化技术
- 大数据量尽量按业务使用多个redis实例或集群把数据分散开
- 不要和其他CPU密集型服务部署在一起， 造成CPU过度竞争
- 不要和其他高硬盘负载的服务部署在一起
- 根据业务需要选择合适的数据类型，并为不同的应用场景设置相应的紧凑存储参数
- 当业务场景不需要数据持久化时，关闭所有的持久化方式可以获得最佳的性能以及最在的内存使用量
- key的命名尽量简短，键在存储中也消耗内存
- 如果有redis集群的，Master最好不要做任何持久化工作。如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
- Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内

# redis命令

连接

- redis客户端连接，如果是本地的直接执行：redis-cli
- 如果要连接到远程服务器的即按照以下格式连接
  端口号不是默认的6379需要加-p指定端口：
  `redis-cli -h host -p port`
  有设置密码的需要加-a指定默认：
  `redis-cli -h host -p port -a “password”`
  如果是集群的，还需要加上-c参数：
  `redis-cli -h host -p port -c -a “password”`
  例：
  `./redis-cli -h 192.168.37.11 -c -p 7006`

统计redis内拥有的key的数量：
`[root@test11 redis-5.0.5]# ./src/redis-cli keys "*"|wc -l`

字符

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：
`192.168.37.11:7003> command key_name`
例：

```
192.168.37.11:7003> set test11 HelloWord
OK
192.168.37.11:7003> getset test11 HelloWorld
"HelloWord"
192.168.37.11:7003> get test11
"HelloWorld"
```

还有以下常用的字符串命令：

```
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 value 追加到 key 原来的值的末尾
```

哈希

Redis hash Redis hash 是一个键值对集合。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
我们设置了redis的一些描述信息(name, description, node, replicas) 到哈希表的 hashkey01 中,例：

```
192.168.37.11:7003> hmset hashkey01 name "redis cluster" description "ve
rsion 5.0" node 3 replicas 2
-> Redirected to slot [8386] located at 192.168.37.11:7002
OK
192.168.37.11:7002> hgetall hashkey01
1) "name"
2) "redis cluster"
3) "description"
4) "version 5.0"
5) "node"
6) "3"
7) "replicas"
8) "2"
```

下表列出了 redis hash 基本的相关命令：

```
序号     命令及描述
1      HDEL key field2 [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
```

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导列表的头部（左边）或者尾部（右边）
- 一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)

以下实例中我们使用了 LPUSH 将三个值插入了名为 listkey 的列表当中

```
192.168.37.11:7002> lpush listkey redis
-> Redirected to slot [15776] located at 192.168.37.11:7003
(integer) 1
192.168.37.11:7003> lpush listkey mongodb
(integer) 2
192.168.37.11:7003> lpush listkey mysql
(integer) 3
192.168.37.11:7003> lrange listkey 0 3
1) "mysql"
2) "mongodb"
```

```
3) "redis"
```

下表列出了redis list基本的相关命令

```
序号     命令及描述
01     LINSERT key BEFORE|AFTER pivot value    #在列表的元素前或者后插入元素
02     LLEN key                                #获取列表长度
03     LPOP key                                #移出并获取列表的第一个元素
04     LPUSH key value1 [value2]               #将一个或多个值插入到列表头部
05     LPUSHX key value                        #将一个或多个值插入到已存在的列
表头部
06     LRANGE key start stop                   #获取列表指定范围内的元素，例如
：lrange listkey 0 4就是从列表下标为0开始到4结束的元素
07     LREM key count value                    #移除列表元素,count > 0即从表
头开始向表尾搜索，<0从表尾开始向表头搜索,count = 0移除表中所有与 VALUE 相等的值
08     LINDEX key index                        #通过索引获取列表中的元素，这里
的index就是代表列表元素的下标0，1，2。。。等
09     LSET key index value                    #通过索引设置列表元素的值
10     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。
11     RPOP key                                #移除并获取列表最后一个元素
12     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
13     RPUSH key value1 [value2]               #在列表中添加一个或多个值
14     RPUSHX key value                        #为已存在的列表添加值
```

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 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

以下实例中我们通过 SADD 命令向名为 setkey 的集合插入的三个元素：

```
192.168.37.11:7002> sadd setkey redis
-> Redirected to slot [2440] located at 192.168.37.11:7004
(integer) 1
192.168.37.11:7004> sadd setkey mongodb
(integer) 1
192.168.37.11:7004> sadd setkey mysql
(integer) 1
192.168.37.11:7004> sadd setkey mysql
(integer) 0
192.168.37.11:7004> smembers setkey
1) "mongodb"
2) "redis"
3) "mysql"
```

下表列出了redis set基本的相关命令:

```
序号     命令及描述
01     SADD key member1 [member2]                          #向集合添加一个或多个成员
02     SCARD key                                           #获取集合的成员数
03     SDIFF key1 [key2]                                   #返回给定所有集合的差集
04     SDIFFSTORE destination key1 [key2]                  #返回给定所有集合的差集并存储在 destination 中
05     SINTER key1 [key2]                                  #返回给定所有集合的交集
06     SINTERSTORE destination key1 [key2]                 #返回给定所有集合的交集并存储在 destination 中
07     SISMEMBER key member                                #判断 member 元素是否是集合 key 的成员
08     SMEMBERS key                                        #返回集合中的所有成员
09     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
```

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 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

以下实例中我们通过命令 ZADD 向 sortedkey 的有序集合中添加了三个值并关联上分数:

```
192.168.37.11:7004> zadd sortedkey 1 redis
-> Redirected to slot [11273] located at 192.168.37.11:7003
(integer) 1
192.168.37.11:7003> zadd sortedkey 2 mongodb
(integer) 1
```

```
192.168.37.11:7003> zadd sortedkey 2 mysql
(integer) 1
192.168.37.11:7003> zrange sortedkey 0 5 withscores
1) "redis"
2) "1"
3) "mongodb"
4) "2"
5) "mysql"
6) "2"
192.168.37.11:7003> zadd sortedkey 3 mysql
(integer) 0
192.168.37.11:7003> zrange sortedkey 0 5 withscores
1) "redis"
2) "1"
3) "mongodb"
4) "2"
5) "mysql"
6) "3"
```

下表列出了redis sorted set基本的相关命令:

```
序号     命令及描述
01    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
02    ZCARD key                                         #获取有序集合的成员数
03    ZCOUNT key min max                                #计算在有序集合中指定区间分数的成员数
04    ZINCRBY key increment member                      #有序集合中对指定成员的分数加上增量 increment
05    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
06    ZLEXCOUNT key min max                             #在有序集合中计算指定字典区间内成员数量
07    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
08    ZRANGEBYLEX key min max [LIMIT offset count]      #通过字典区间返回有序集合的成员
09    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）
```

数据过期

如下实例：

```
127.0.0.1:6379> set timeover 1122
OK
127.0.0.1:6379> ttl timeover                    #查看timeover的过期时间
(integer) -1                                    #-1代表永不过期
127.0.0.1:6379> help expire

  EXPIRE key seconds
  summary: Set a key's time to live in seconds
  since: 1.0.0
  group: generic

127.0.0.1:6379> expire timeover 5               #设置过期时间为5秒
(integer) 1
127.0.0.1:6379> ttl timeover                    #5秒过后再查看timeover的过期时间
(integer) -2                                    #-2代表这个key过期了
127.0.0.1:6379> get timeover
(nil)
```

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。

以下实例演示了发布订阅是如何工作的。在我们实例中我们创建了订阅频道名为 redisChat:

```
192.168.37.11:7003> subscribe redischat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
```

```
2) "redischat"
3) (integer) 1
```

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息:

```
[root@test11 cluster-test]# ./redis-cli -h 192.168.37.11 -c -p 7001
192.168.37.11:7001> publish redischat "redis is quickly"
(integer) 0
192.168.37.11:7001> publish redischat "redis is good"
(integer) 0
# 刚才创建订阅者的客户端会显示如下消息：
1) "message"
2) "redischat"
3) "redis is quickly"
1) "message"
2) "redischat"
3) "redis is good"
```

下表列出了 redis 发布订阅常用命令：

```
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,例：psubscribe redis*
2    PUNSUBSCRIBE [pattern [pattern ...]]                #退订所有给定模式的频
道。
3    PUBSUB subcommand [argument [argument ...]]         #查看订阅与发布系统状
态。
- pubsub是一个查看pub/sub状态的内省命令，有以下几个用法：
- - pubsub channels[pattern]：列出当前的活跃频道；
- - pubsub numsub[channel_1....channel_n]：返回给定频道订阅者的数量；
- - pubsub numpat：返回订阅者的数量，所有客户端之和。
4    PUBLISH channel message                             #将信息发送到指定的频
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， 并且带有以下两个重要的保证：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务可以理解为一个打包的批量执行脚本,事务中任意命令执行失败，其余的命令仍会被执行。

一个事务从开始到执行会经历以下三个阶段：

- 开始事务。
- 命令入队。
- 执行事务。

以下实例说明redis事务是如何工作的：

```
127.0.0.1:6379> set k2 v2
OK
127.0.0.1:6379> multi                                   #标记一个事务的开始
OK
127.0.0.1:6379> set k1 11
QUEUED
127.0.0.1:6379> incr k2                                 #对k2值为v2进行+1
QUEUED
127.0.0.1:6379> set k3 33
QUEUED
127.0.0.1:6379> exec                                    #执行所有事务块内的命令
1) OK
2) (error) ERR value is not an integer or out of range      #这个说明了事务中存在任意命令执行失败，不影响其余的命令执行
3) OK
127.0.0.1:6379> watch k3                                #开启对k3的监视，这种形式的锁被称作乐观锁，它是一种非常强大的锁机制
OK
127.0.0.1:6379> multi                                   #标注1
OK
127.0.0.1:6379> incrby k1 10                            #对k1 +10
QUEUED
127.0.0.1:6379> decrby k3 10                            #对k3 -10
QUEUED
127.0.0.1:6379> exec                                    #事务失败，因为在 WATCH 执行之后，EXEC 执行之前，有其他客户端修改了事务中的key的值，那么当前客户端的事务就会失败
(nil)
127.0.0.1:6379> get k3
"30"
127.0.0.1:6379> get k1
"11"

#这里是在开启事务后（标注1处），在新窗口执行标注2中的操作，更改balance的值
127.0.0.1:6379> get k3
"33"
127.0.0.1:6379> set k3 30                                #标注2
OK
```

除上面用到的几个事务命令，还有下面两务相关命令：

- discard 取消事务，放弃执行事务块内的所有命令
- unwatch 取消 WATCH 命令对所有 key 的监视

# redis工具

## 性能测试

- redis-benchmark是Redis自带的基准性能测试工具， 它提供了很多选项帮助开发和运维人员测试Redis的相关性能。
- 以下测试实例指定实例中主机为 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：

```
[root@test11 redis-5.0.5]# ./src/redis-benchmark -h 127.0.0.1 -p 6379 -t
 set,lpush -n 100000 -q
SET: 75528.70 requests per second
LPUSH: 103305.79 requests per second
```

- 当然也可以不指定测试命令，这样就是测试所有的redis命令的性能，如下：

```
[root@test11 redis-5.0.5]# ./src/redis-benchmark -n 100000
====== SET ======
  100000 requests completed in 1.01 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.35% <= 1 milliseconds
99.75% <= 2 milliseconds
99.84% <= 3 milliseconds
99.90% <= 4 milliseconds
100.00% <= 4 milliseconds
99304.87 requests per second

====== GET ======
.
.
.
====== INCR ======
.
.
.
```

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。 | |

## 系统监控

info可以查看redis的状态，如下：
具体参数含义请参考：https://redis.io/commands/info

```
127.0.0.1:6379> info
# Server
redis_version:5.0.5
redis_git_sha1:00000000
redis_git_dirty:0
.
# Clients
.
# Memory
.
# Persistence
.
# Stats
.
# Replication
.
# CPU
.
# Cluster
cluster_enabled:0
# Keyspace
db0:keys=8,expires=0,avg_ttl=0
```

可以具体的查看某一项的状态信息，语法：info [option] 下面将着重介绍一些比较重要的性能指标：

### MEMORY

```
127.0.0.1:6379> info memory
# Memory
used_memory:2885952                     #由Redis分配器分配的内存总量，以字节（byte）为单位，不包含内存碎片浪费掉的内存
used_memory_human:2.75M                 #与used_memory一样，只能单位转为M为单位，易于阅读
used_memory_rss_human:6.90M             #从操作系统上显示已经分配的内存总量，包含内存碎片
used_memory_lua_human:37.00K            #Lua脚本引擎所使用的内存大小
mem_fragmentation_ratio:2.54            #内存碎片率
mem_allocator:jemalloc-5.1.0            #在编译时指定的Redis使用的内存分配器
```

- 内存使用率是Redis服务最关键的一部分。如果一个Redis实例的内存使用率超过可用最大内存(used_memory > 可用最大内存)，那么操作系统开始进行内存与swap空间交换，Redis和依赖Redis上数据的应用会受到严重的性能影响。
- 若是在使用Redis期间没有开启rdb快照或aof持久化策略，那么缓存数据在Redis崩溃时就有丢失的危险。因为当Redis内存使用率超过可用内存的95%时，部分数据开始在内存与swap空间来回交换，这时就可能有丢失数据的危险。
  - 可以按以下方案来避免这种问题
    1、通过减少Redis的内存占用率
    2、尽可能的使用Hash数据结构
    3、设置key的过期时间
    4、回收key
- mem_fragmentation_ratio=used_memory_rss_human/used_memory_human；内存碎片率稍大于1是合理的，这个值表示内存碎片率比较低，也说明redis没有发生内存交换。但如果内存碎片率超过1.5，那就说明Redis消耗了实际需要物理内存的150%，其中50%是内存碎片率。若是内存碎片率低于1的话，说明Redis内存分配超出了物理内存，操作系统正在进行内存交换。内存交换会引起非常明显的响应延迟。
  - 倘若内存碎片率超过了1.5，那可能是操作系统或Redis实例中内存管理变差的表现。下面有3种方法解决内存管理变差的问题，并提高Redis性能：
    1、重启Redis服务器
    2、限制内存交换
    3、修改内存分配器

### STATS

```
# Stats
total_connections_received:1309         #Redis自启动以来处理的客户端连接数总数
total_commands_processed:2240765        #Redis自启动以来命令处理数
rejected_connections:0                  #Redis自启动以来拒绝的客户端连接数
latest_fork_usec:252                    #获取最近一个fork操作的耗时，  单位为微秒
```

- 可以写个脚本，定期记录total_commands_processed的值。当客户端明显发现响应时间过慢时，可以通过记录的total_commands_processed历史数据值来判断命理处理总数是上升趋势还是下降趋势，以便排查问题。
  - 命令队列里的命令数量过多，后面命令一直在等待中，解决办法：
    1、使用多参数命令，通过单命令多参数的形式取代多命令单参数的形式
    2、管道命令：减少多命令的方法是使用管道(pipeline)，把几个命令合并一起执行，从而减少因网络开销引起的延迟问题
    3、避免操作大集合的慢命令：如果命令处理频率过低导致延迟时间增加，这可能是因为使用了高时间复杂度的命令操作导致
  - 慢命令阻塞Redis，解决办法：
    1、使用slowlog查出引发延迟的慢命令

CLIENTS

```
127.0.0.1:6379> info clients
# Clients
connected_clients:2                     #当前Redis节点的客户端连接数，默认最大值为10000,一旦超过新的客户端连接将被拒绝。
client_recent_max_input_buffer:2        #最大输入缓冲区，可以设置超过10M报警
client_recent_max_output_buffer:0       #最大输出缓冲区
blocked_clients:0                       #正在执行阻塞命令的客户端个数
```

## Redis监控

基础环境：服务器：192.168.37.11
系统版本：centos7.6
go版本：1.12
由grafana+prometheus+redis_exporter组成监控套件，部署如下：

### grafana

- 执行如下命令安装： `yum install https://dl.grafana.com/oss/release/grafana-5.4.2-1.x86_64.rpm`
- 启动： `systemctl start grafana-server`
- 访问地址：http://192.168.37.11:3000/login 默认管理员帐号密码：admin admin

### node_exporter

github地址：https://github.com/prometheus/node_exporter/
作用：用于机器系统数据收集，配合Prometheus使用，每个redis服务器上安装，此实例是单服务器，所以只安装一个。

- 安装node_exporter

```
[root@test11 ~]# go get github.com/prometheus/node_exporter
```

```
[root@test11 ~]# cd go/src/github.com/prometheus/node_exporter
[root@test11 redis_exporter]# make
```

- 启动： `[root@test11 node_exporter]# ./node_exporter &`
- 安装成功界面如下：


**redis_exporter**

github地址：https://github.com/oliver006/redis_exporter
作用：抓取redis的指标，每个redis服务器上安装，此实例是单服务器，所以只安装一个。

- 安装redis_exporter：

```
[root@test11 ~]# go get github.com/oliver006/redis_exporter
[root@test11 ~]# cd go/src/github.com/oliver006/redis_exporter/
[root@test11 redis_exporter]# go build
```

- 启动： `[root@test11 redis_exporter]# ./redis_exporter &`
- 地址：http://192.168.37.11:9121/
- 安装成功界面如下：


**prometheus**

github地址：https://github.com/prometheus/prometheus
定义：标准的数据搬运系统

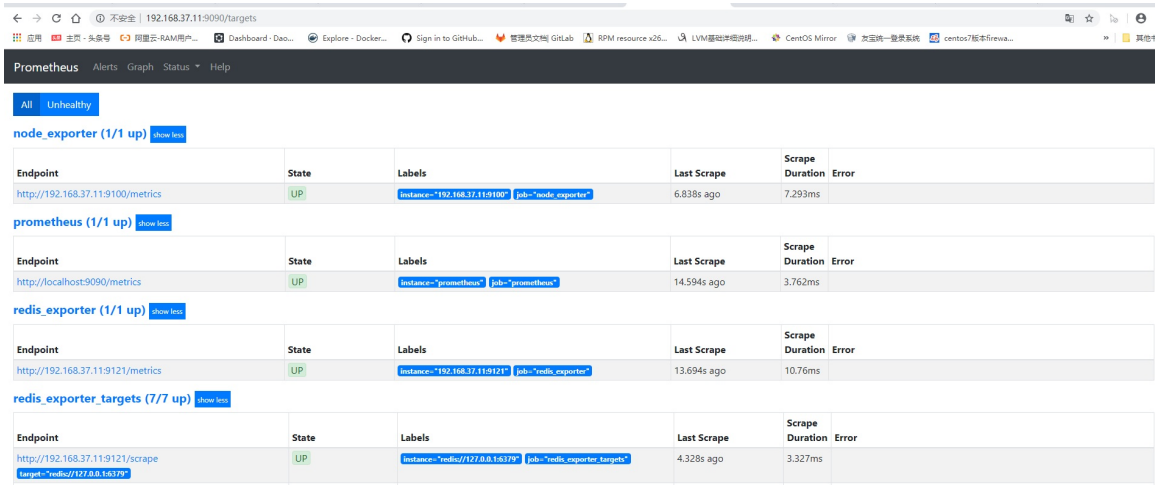
- 下载： `wget https://github.com/prometheus/prometheus/archive/master.zip`
- 解压到并移动到指定目录下：

```
unzip master.zip
mv prometheus-master/ /usr/local/
cd /usr/local/prometheus-master
```

- 构建prometheus-master `make build`
- 编辑prometheus.yml文件,监控原来创建的redis节点，如下：

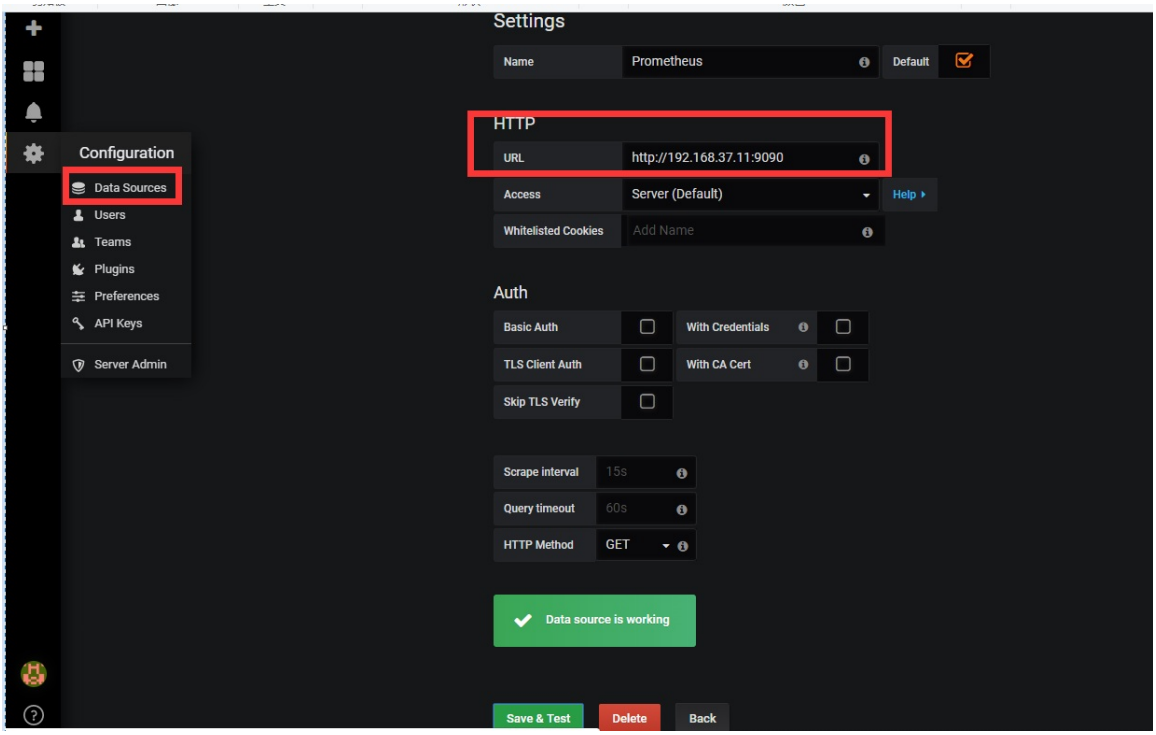
```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
        labels:
          instance: prometheus

  - job_name: 'node_exporter'
    static_configs:
      - targets: ['192.168.37.11:9100']

  - job_name: 'redis_exporter_targets'
    static_configs:
      - targets:
        - redis://127.0.0.1:6379
        - redis://192.168.37.11:7001
        - redis://192.168.37.11:7002
        - redis://192.168.37.11:7003
        - redis://192.168.37.11:7004
        - redis://192.168.37.11:7005
        - redis://192.168.37.11:7006
    metrics_path: /scrape
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 192.168.37.11:9121

  - job_name: 'redis_exporter'
    static_configs:
      - targets:
        - 192.168.37.11:9121
```

- 启动： `[root@test11 prometheus-master]# ./prometheus --config.file=prometheus.yml &`
- 访问地址：http://192.168.37.11:9090
- 安装成功界面如下：

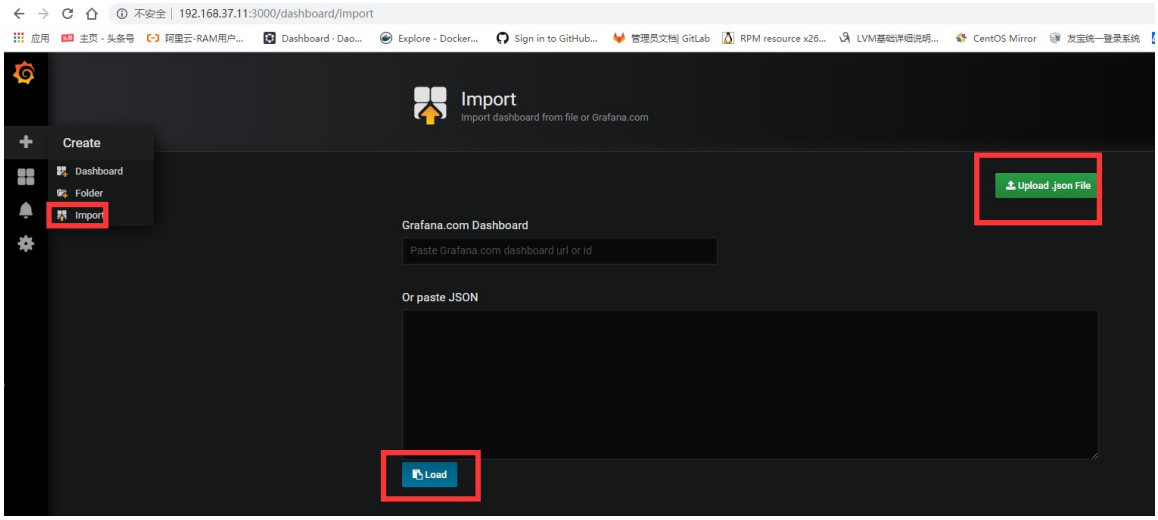



集成到**Grafana**

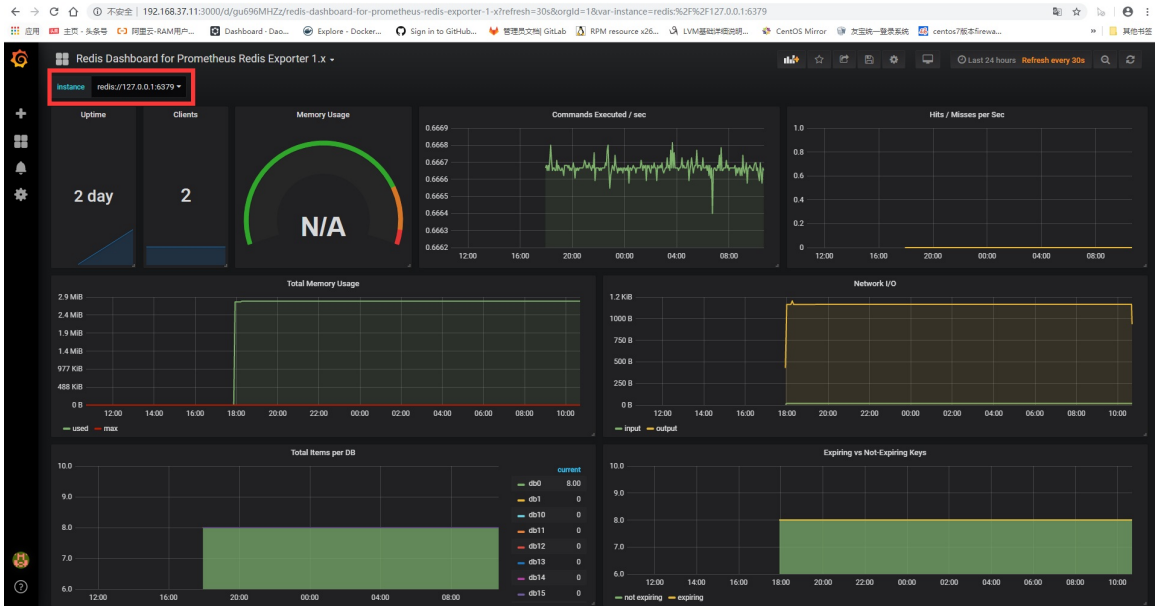
- 下载grafana的redis的prometheus-redis_rev1.json模板，下载地址：https://grafana.com/api/dashboards/763/revisions/3/download
- 在grafana中新建prometheus数据源，如下：




- 导入刚才下载的json模板，如下：




- 集成到grafana的模板界面如下：



# 备份迁移

## 备份

自动备份

### RDB

redis配置文件里默认是开启了RDB持久化的，符合下面的条件的会触发持久化到磁盘上（可以自定义）：

```
# 900秒（15分钟）内至少1个key值改变（则进行数据库保存--持久化）
# 300秒（5分钟）内至少10个key值改变（则进行数据库保存--持久化）
# 60秒（1分钟）内至少10000个key值改变（则进行数据库保存--持久化）
save 900 1
save 300 10
save 60 10000
```

### AOF

redis配置文件里默认是开启了AOF持久化,如下：
`appendonly yes`

手动备份

### RDB

命令：

- save：redis save 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘
- bgsave：bgsave命令执行之后立即返回Background saving started，然后 redis fork 出一个新子进程，原来的 redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。返回成功执行的最后一次DB保存的UNIX TIME。客户端可以检查bgsave命令是否成功读取LASTSAVE值，然后发出bgsave命令，并且如果lastsave改变则每隔N秒定期检查一次。

区别：

- save保存是阻塞主进程，客户端无法连接redis，等SAVE完成后，主进程才开始工作，客户端可以连接
- bgsave是fork一个save的子进程，在执行save过程中，不影响主进程，客户端可以正常链接redis，等子进程fork执行save完成后，通知主进程，子进程关闭。很明显BGSAVE方式比较适合线上的维护操作。

### AOF

命令：

- bgrewriteaof

## 恢复

- 如果是redis机器或进程挂掉，假如已经开启了AOF持久化，那么重启机器/进程，redis会直接基于AOF日志文件恢复数据
- 如果redis当前最新的AOF和RDB文件出现了丢失/损坏，那么可以尝试基于该机器上当前的某个最新的RDB数据副本进行数据恢复
  1、停止redis(命令是redis-cli shutdown),
  2、在配置文件中关闭aof: appendonly no
  3、拷贝rdb日志备份到redis的数据目录下
  4、启动redis，在没有AOF的文件下，redis会自动读取RDB文件恢复数据
  5、尝试get一个key,确认数据恢复

## 迁移工具

**redis-migrate-tool**

github地址：https://github.com/vipshop/redis-migrate-tool

简介：redis-migrate-tool是一种方便有用的工具，用于在redis之间迁移数据。

特征：

- 快速。
- 多线程。
- 基于redis复制。
- 实时迁移。
- 在迁移数据的过程中，源redis还可以为用户提供服务。
- 异构迁移。
- Twemproxy和redis集群支持。
- 当目标是twemproxy时，键被直接导入到twemproxy后面的redis中。
- 迁移状态视图。
- 数据验证机制。

基础环境：服务器：192.168.37.11
系统版本：centos7.6

安装依赖：`yum -y install automake libtool autoconf bzip2 unzip`

安装软件：

```
[root@test11 redis-5.0.5]# wget https://github.com/vipshop/redis-migrate
-tool/archive/master.zip
[root@test11 redis-5.0.5]# unzip master.zip
[root@test11 redis-5.0.5]# cd redis-migrate-tool-master/
[root@test11 redis-migrate-tool-master]# autoreconf -fvi
[root@test11 redis-migrate-tool-master]# ./configure
```

```
[root@test11 redis-migrate-tool-master]# make
[root@test11 redis-migrate-tool-master]# ./src/redis-migrate-tool -h
```

实例：从集群到集群迁移，编辑rmt.conf进行迁移，如下：

```
[root@test11 redis-migrate-tool-master]# cat rmt.conf
[source]
type: redis cluster
servers :
  - 192.168.37.11:7001

[target]
type: redis cluster
servers:
  - 192.168.37.11:7007

[common]
listen: 0.0.0.0:8888
```

执行同步：`[root@test11 redis-migrate-tool-master]# ./src/redis-migrate-tool -c rmt.conf -o log -d`

一致性效验：`[root@test11 redis-migrate-tool-master]# ./src/redis-migrate-tool -c rmt.conf -o log -C redis_check`

插入数据效验：`[root@test11 redis-migrate-tool-master]# ./src/redis-migrate-tool -c rmt.conf -o log -C redis_testinsert`

注意了：redis5.0版本会报错，查看生成的日志：`cat log`，报错如下：

```
[2019-07-19 14:31:53.548] rmt_redis.c:6446 ERROR: Can't handle RDB forma
t version -905891832
[2019-07-19 14:31:53.548] rmt_redis.c:6715 ERROR: Rdb file for node[192.
168.37.11:7004@17004] parsed failed
```

原因是redis-migrate-tool-master这个工具最后一次更新是2017年，应该是不支持redis5.0的新格式，期待作者更新！所以redis5.0建议还是采用冷迁移。redis3.x是可以迁移成功的。

# redis面试

请参考：https://www.w3cschool.cn/redis/redis-ydwp2ozz.html