尚观数据库研究室

www.updba.com

UOA 云数据库集群架构师

redis 管理维护高可用与集群及企业项目

(注：本项目完全需要学生自主实施，讲师不进行传统授课，只进行辅导或解说)

尚观数据库研究室团队

项目总监： Shrek

项目研发： Foway

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

目 录

前言

REDIS 是键值内存存储的 NoSQL 典型代表。本课程就 redis 管理维护方面进行系统的学习。为从事 redis 相关工作打下坚实的基础，同时也让自己具备相应的企业应用技巧及实战经验。

新浪微博大量使用 Redis 作为自己的 NoSQL 数据库，Redis 是一个基于内存的 Key-Value 存储的 NoSQL 引擎。与其他 Key-Value 引擎不同，Redis 的 Value 可以支持多种数据结构，如哈希、List、Set 等。和 Memcached 类似，它支持存储的 value 类型相对更多，与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了

master-slave(主从)同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得 Redis 可执行单层树复制。从盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。Redis 存储使用了两种文件格式：全量数据和增量请求。全量数据格式是把内存中的数据写入磁盘。

版本

软件版本为2.8

章节安排

Redis 管理		
课程模块	**知识架构**	**讲课目的**
了解 Redis	键值数据库概述 Redis 特性 Redis 体系概述 Redis 应用场景与企业案例分析	让学员熟悉 redis 的基本功能与组件。能清楚在什么场合可以使用 redis。
安装 Redis	配置 yum 进行安装 认识 redis 目录结构与文件 熟悉/etc/redis.conf 启动与关闭 redis 服务 启动故障分析与排除	让学员熟悉 redis 安装与启动关闭操作

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

	升级 redis 版本	
管理	配置文件详解 主从复制架构及部署 数据持久性及备份恢复技术 高可用性架构与部署 数据分区架构与部署 安全设计与实施 如何连接与操作 redis redis 常用命令使用 事务与大量数据快速插入 程序调用 redis 企业案例	让学员熟悉 redis 配置文件语法并进行配置。熟练如 string,hash,list,set,zset,push,pop,add,remove 等常用命令。熟悉事务处理与大数据快速加载。熟悉持久性操作，虚拟内存等优化技巧

第一章 了解 redis

Redis 体系

我们知道 NoSQL 具有如下特点：
处理超大规模的数据
运行在 pc 服务机群上
轻松解决性能瓶颈

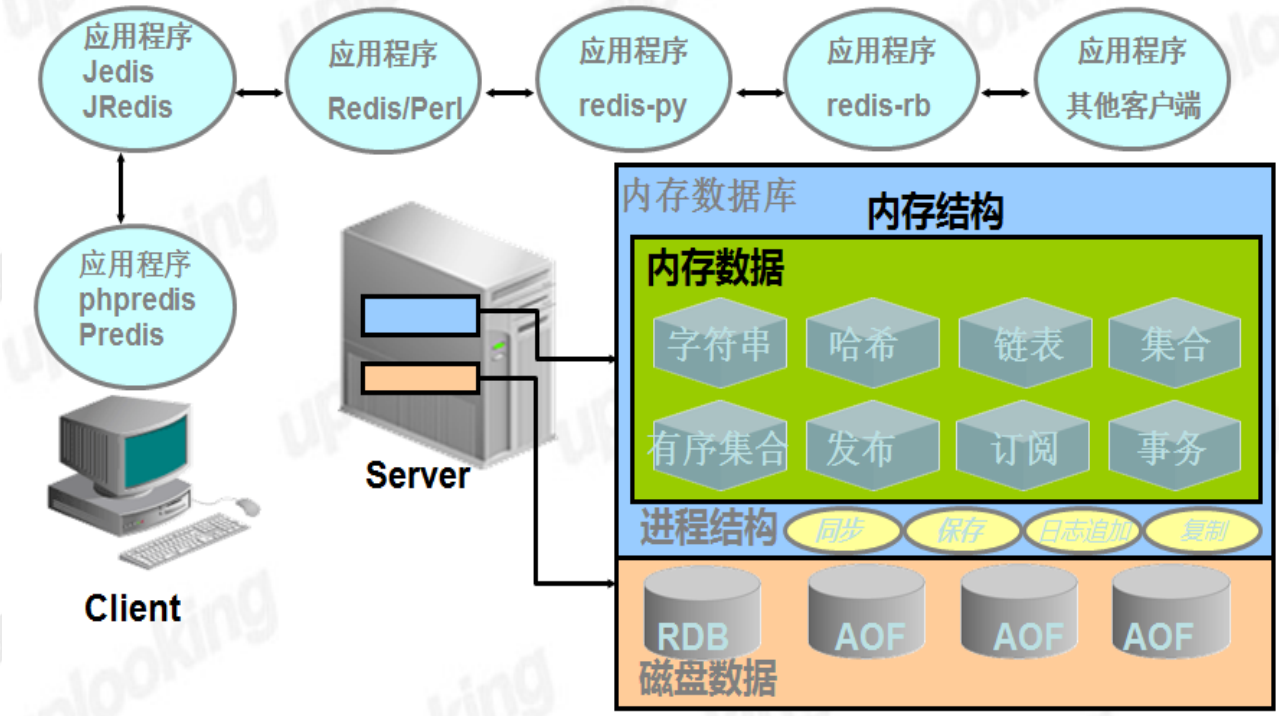
我们也明白 NoSQL 适合如下场景：
对数据高并发处理
对大数据高效率存储和访问

对数据高可用及高扩展性

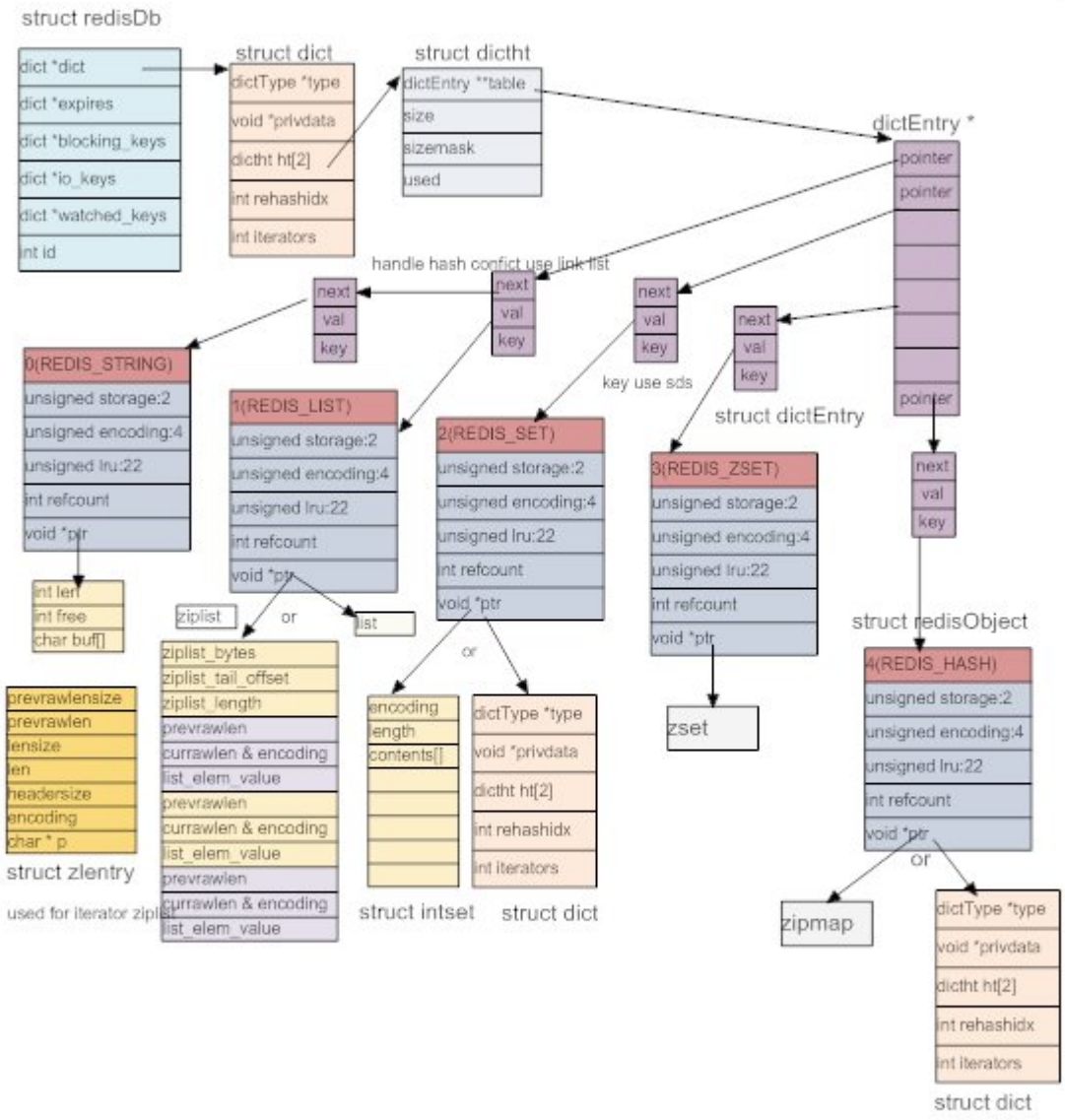
Redis 是 NoSQL 数据库的一种。自然也具备上面的特点及使用场景。
Redis 是以 kv 方式存储数据库，只要特点：非关系，分布，开源，可扩展，高速内存操作。
适合运行在廉价的 pc 服务器上分布式处理海量数据。

Redis 是一个开源的，先进的 kv 存储方式的数据库，通常叫数据结构服务器。键可以包含字符串 strings，哈希 hashes，lists 链表，集合 sets，有序集合 sorted sets/zsets. 这些数据类型支持 push、pop，add，move 及集合等操作，支持各种不同方式排序。 为了提供性能，数据都在内存中。为了数据可靠根据需要周期更新数据到磁盘或日志记录文件。

Redis 体系结构:



Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.



redis 的内存存储结构示意图

Redis 是服务器与客户端结构模式。 而 redis 只提供了服务器端功能，具体的客户端功能是根据实际需求采用符合项目要求的客户端程序。比如通过 php 程序使用 redis 则推荐 Predis 或 phpredis。

Strings

Strings 数据结构是简单的 key-value 类型，value 其实不仅是 String，也可以是数字。使用 Strings 类型，你可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受 Redis 的定时持久化，操作日志及 Replication 等功能。除了提供与 Memcached 一样的 get、set、incr、decr 等操作外，Redis 还提供了下面一些操作：

- 获取字符串长度
- 往字符串 append 内容
- 设置和获取字符串的某一段内容

- 设置及获取字符串的某一位（bit）
- 批量设置一系列字符串的内容

Hashs

在 Memcached 中，我们经常将一些结构化的信息打包成 hashmap，在客户端序列化后存储为一个字符串的值，比如用户的昵称、年龄、性别、积分等，这时候在需要修改其中某一项时，通常需要将所有值取出反序列化后，修改某一项的值，再序列化存储回去。这样不仅增大了开销，也不适用于一些可能并发操作的场合（比如两个并发的操作都需要修改积分）。而 Redis 的 Hash 结构可以使你像在数据库中 Update 一个属性一样只修改某一项属性值。

Lists

Lists 就是链表，相信略有数据结构知识的人都应该能理解其结构。使用 Lists 结构，我们可以轻松地实现最新消息排行等功能。Lists 的另一个应用就是消息队列，可以利用 Lists 的 PUSH 操作，将任务存在 Lists 中，然后工作线程再用 POP 操作将任务取出进行执行。Redis 还提供了操作 Lists 中某一段的 api，你可以直接查询，删除 Lists 中某一段的元素。

Sets

Sets 就是一个集合，集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Sets 数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis 还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

Sorted Sets

和 Sets 相比，Sorted Sets 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，比如一个存储全班同学成绩的 Sorted Sets，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用 Sorted Sets 来做带权重的队列，比如普通消息的 score 为 1，重要消息的 score 为 2，然后工作线程可以选择按 score 的倒序来获取工作任务。让重要的任务优先执行。

Pub/Sub

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

Transactions

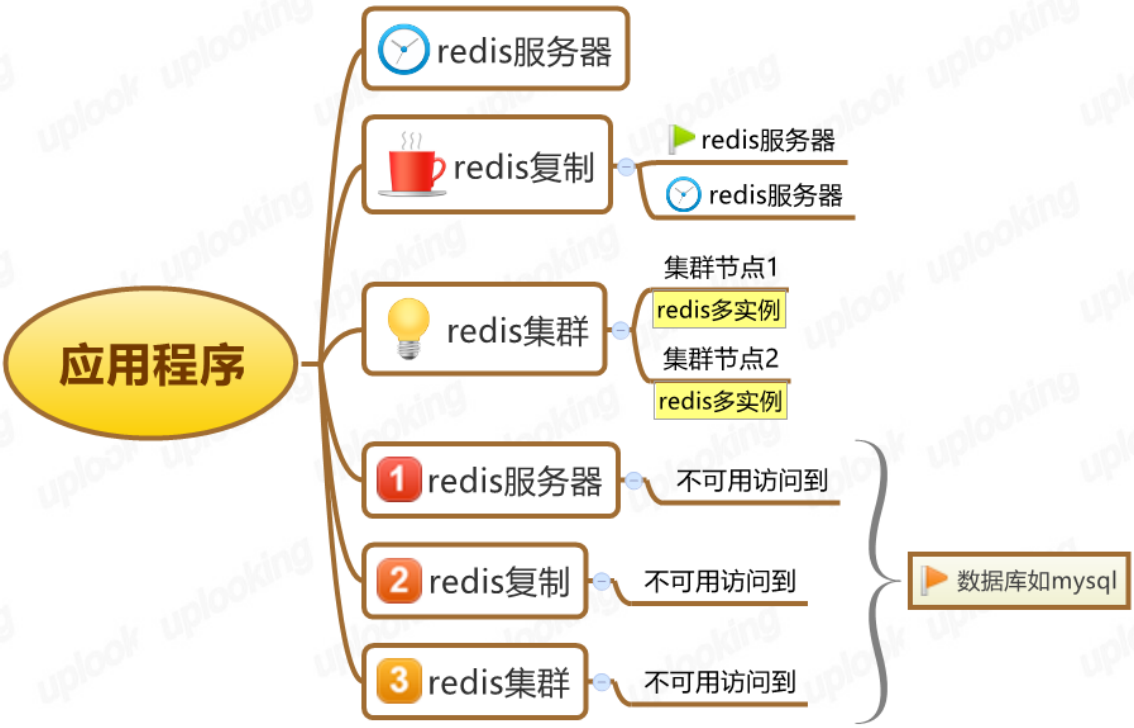
谁说 NoSQL 都不支持事务，虽然 Redis 的 Transactions 提供的并不是严格的 ACID 的事务（比如一串用 EXEC 提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个 Transactions 还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间有会有其它客户端命令插进来执行）。Redis 还提供了一个 Watch 功能，你可以对一个 key 进行 Watch，然后再执行 Transactions，在这过程中，如果这个 Watched 的值进行了修改，那么这个 Transactions 会发现并拒绝执行。

Redis 典型架构

Redis 适用场合典型架构：
应用程序直接访问 redis 数据库

应用程序直接访问 redis，当 redis 访问失败访问磁盘数据库比如 mysql 关系数据库

Redis 架构汇总：



Redis 典型用途

Redis 典型企业应用:

下面列出 11 种 Web 应用场景，在这些场景下可以充分的利用 Redis 的特性，大大提高效率。

1．在主页中显示最新的项目列表。

Redis 使用的是常驻内存的缓存，速度非常快。LPUSH 用来插入一个内容 ID，作为关键字存储在列表头部。LTRIM 用来限制列表中的项目数最多为 5000。如果用户需要的检索的数据量超越这个缓存容量，这时才需要把请求发送到数据库。

2．删除和过滤。

如果一篇文章被删除，可以使用 LREM 从缓存中彻底清除掉。

3．排行榜及相关问题。

排行榜（leader board）按照得分进行排序。ZADD 命令可以直接实现这个功能，而 ZREVRANGE 命令可以用来按照得分来获取前 100 名的用户，ZRANK 可以用来获取用户排名，非常直接而且操作容易。

4．按照用户投票和时间排序。

这就像 Reddit 的排行榜，得分会随着时间变化。LPUSH 和 LTRIM 命令结合运用，把文章添加到一个列表中。一项后台任务用来获取列表，并重新计算列表的排序，ZADD 命令用来按照新的顺序填充生成列表。列表可以实现非常快速的检索，即使是负载很重的站点。

5．过期项目处理。

使用 unix 时间作为关键字，用来保持列表能够按时间排序。对 current_time 和 time_to_live 进行检索，完成查找过期项目的艰巨任务。另一项后台任务使用 ZRANGE...WITHSCORES 进行查询，删除过期的条目。

6．计数。

进行各种数据统计的用途是非常广泛的，比如想知道什么时候封锁一个 IP 地址。INCRBY 命令让这些变得很容易，通过原子递增保持计数；GETSET 用来重置计数器；过期属性用来确认一个关键字什么时候应该删除。

7．特定时间内的特定项目。

这是特定访问者的问题，可以通过给每次页面浏览使用 SADD 命令来解决。SADD 不会将已经存在的成员添加到一个集合。

8．实时分析正在发生的情况，用于数据统计与防止垃圾邮件等。

使用 Redis 原语命令，更容易实施垃圾邮件过滤系统或其他实时跟踪系统。

9．Pub/Sub。

在更新中保持用户对数据的映射是系统中的一个普遍任务。Redis 的 pub/sub 功能使用了 SUBSCRIBE、UNSUBSCRIBE 和 PUBLISH 命令，让这个变得更加容易。

10．队列。

在当前的编程中队列随处可见。除了 push 和 pop 类型的命令之外，Redis 还有阻塞队列的命令，能够让一个程序在执行时被另一个程序添加到队列。你也可以做些更有趣的事情，比如一个旋转更新的 RSS feed 队列。

11．缓存。

Redis 缓存使用的方式与 memcache 相同。

网络应用不能无休止地进行模型的战争，看看这些 Redis 的原语命令，尽管简单但功能强大，把它们加以组合，所能完成的就更无法想象。当然，你可以专门编写代码来完成所有这些操作，但 Redis 实现起来显然更为轻松。

第二章 redis 安装与配置

服务器端程序安装

redis 是服务端与客户端组成的。本部分主要是服务端的安装与配置。

安装 redis 方式主要源码编译与 rpm 打包方式安装。

源码编译：

wget http://download.redis.io/releases/redis-2.8.13.tar.gz

tar xzvf redis-2.8.13.tar.gz

cd redis-2.8.13

make

make test & make install

cd ../utils

./install_server.sh

[root@wyzc ~]# wget http://download.redis.io/releases/redis-2.8.13.tar.gz

--2014-08-29 10:39:15-- http://download.redis.io/releases/redis-2.8.13.tar.gz

Resolving download.redis.io... 109.74.203.151

Connecting to download.redis.io|109.74.203.151|:80... connected.

HTTP request sent, awaiting response... 200 OK

Length: 1227538 (1.2M) [application/x-gzip]

Saving to: “redis-2.8.13.tar.gz”

```
100%[===========================>] 1,227,538      208K/s     in 5.8s


2014-08-29 10:39:22 (208 KB/s) -  "redis-2.8.13.tar.gz"  saved [1227538/1227538]



[root@wyzc ~]# tar xzf   redis-2.8.13.tar.gz

[root@wyzc ~]# ls redis-2.8.13

00-RELEASENOTES   deps          README               sentinel.conf

BUGS              INSTALL       redis.conf           src

CONTRIBUTING      Makefile      runtest              tests

COPYING           MANIFESTO     runtest-sentinel     utils

[root@wyzc ~]#

[root@wyzc ~]# cd redis-2.8.13

[root@wyzc redis-2.8.13]# make

。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

o/ All tests passed without errors!

Cleanup: may take some time... OK

make[1]: Leaving directory `/root/redis-2.8.13/src'

[root@wyzc redis-2.8.13]#

[root@wyzc redis-2.8.13]# make   install

cd src && make install

make[1]: Entering directory `/root/redis-2.8.13/src'
```

```
Hint: To run 'make test' is a good idea ;)


    INSTALL install

    INSTALL install

    INSTALL install

    INSTALL install

    INSTALL install

make[1]: Leaving directory `/root/redis-2.8.13/src'

[root@wyzc redis-2.8.13]#

[root@wyzc redis-2.8.13]# cd utils/

[root@wyzc utils]# ./install_server.sh

Welcome to the redis service installer

This script will help you easily set up a running redis server


Please select the redis port for this instance: [6379]

Selecting default: 6379

Please select the redis config file name [/etc/redis/6379.conf]

Selected default - /etc/redis/6379.conf

Please select the redis log file name [/var/log/redis_6379.log]

Selected default - /var/log/redis_6379.log

Please select the data directory for this instance [/var/lib/redis/6379]

Selected default - /var/lib/redis/6379
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
Please select the redis executable path [/usr/local/bin/redis-server]

Selected config:

Port               : 6379

Config file      : /etc/redis/6379.conf

Log file         : /var/log/redis_6379.log

Data dir           : /var/lib/redis/6379

Executable       : /usr/local/bin/redis-server

Cli Executable : /usr/local/bin/redis-cli

Is this ok? Then press ENTER to go on or Ctrl-C to abort.

Copied /tmp/6379.conf => /etc/init.d/redis_6379

Installing service...

Successfully added to chkconfig!

Successfully added to runlevels 345!

Starting Redis server...

Installation successful!

[root@wyzc utils]#

[root@wyzc utils]# head -15   /etc/init.d/redis_6379

#/bin/sh

#Configurations injected by install_server below....


EXEC=/usr/local/bin/redis-server

CLIEXEC=/usr/local/bin/redis-cli
```

```
PIDFILE=/var/run/redis_6379.pid

CONF="/etc/redis/6379.conf"

REDISPORT="6379"

###############

# SysV Init Information

# chkconfig: - 58 74

# description: redis_6379 is the redis daemon.

### BEGIN INIT INFO

# Provides: redis_6379

# Required-Start: $network $local_fs $remote_fs
```

```
[root@wyzc utils]# vi   /etc/init.d/redis_6379

[root@wyzc utils]# head -10   /etc/init.d/redis_6379

#/bin/sh

#

# SysV Init Information

# chkconfig: - 58 74

# description: redis_6379 is the redis daemon.

# Configurations injected by install_server below....


EXEC=/usr/local/bin/redis-server

CLIEXEC=/usr/local/bin/redis-cli

PIDFILE=/var/run/redis_6379.pid
```

```
[root@wyzc utils]#
[root@wyzc utils]# chkconfig   redis_6379    on --level 2345
[root@wyzc utils]# chkconfig   redis_6379    --list
redis_6379        0:off    1:off    2:on     3:on     4:on     5:on     6:off
[root@wyzc utils]#
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    status
Redis is not running
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    start
Starting Redis server...
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    status
Redis is running (8255)
[root@wyzc redis-2.8.13]# /usr/local/bin/redis-cli    shutdown
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    status
Redis is not running
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    start
Starting Redis server...
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    status
Redis is running (8268)
[root@wyzc redis-2.8.13]# /etc/init.d/redis_6379    stop
Stopping ...
Redis stopped
[root@wyzc redis-2.8.13]#
```

[root@wyzc redis-2.8.13]# ls /etc/redis/

6379.conf

[root@wyzc redis-2.8.13]# ls /etc/init.d/redis_6379

/etc/init.d/redis_6379

[root@wyzc redis-2.8.13]# ls /usr/local/bin/redis-*

/usr/local/bin/redis-benchmark

/usr/local/bin/redis-cli

/usr/local/bin/redis-check-aof

/usr/local/bin/redis-server

/usr/local/bin/redis-check-dump

[root@wyzc redis-2.8.13]#

rpm 包安装：

配置 yum 进行安装

rhel 系统直接配置 EPEL 扩展源

yum install redis

[root@wyzc ~]# yum install redis

[root@wyzc ~]# rpm -ql redis

/etc/logrotate.d/redis

/etc/rc.d/init.d/redis

/etc/redis.conf

/usr/bin/redis-benchmark

/usr/bin/redis-check-aof

/usr/bin/redis-check-dump

/usr/bin/redis-cli

/usr/sbin/redis-server

/usr/share/doc/redis-2.4.10

/usr/share/doc/redis-2.4.10/00-RELEASENOTES

/usr/share/doc/redis-2.4.10/BUGS

/usr/share/doc/redis-2.4.10/CONTRIBUTING

/usr/share/doc/redis-2.4.10/COPYING

/usr/share/doc/redis-2.4.10/README

/usr/share/doc/redis-2.4.10/TODO

/var/lib/redis

/var/log/redis

/var/run/redis

[root@wyzc ~]#

版本 2.8 的

[root@wyzc ~]# rpm -e redis

[root@wyzc ~]# rpm -ivh redis-2.8.13-3.el6.x86_64.rpm

Preparing... ### [100%]

1:redis ### [100%]

[root@wyzc ~]# rpm -ql redis

/etc/logrotate.d/redis

/etc/rc.d/init.d/redis

/etc/rc.d/init.d/redis-sentinel

/etc/redis-sentinel.conf

/etc/redis.conf

/usr/bin/redis-benchmark

/usr/bin/redis-check-aof

/usr/bin/redis-check-dump

/usr/bin/redis-cli

/usr/sbin/redis-sentinel

/usr/sbin/redis-server

/usr/share/doc/redis-2.8.13

/usr/share/doc/redis-2.8.13/00-RELEASENOTES

/usr/share/doc/redis-2.8.13/BUGS

/usr/share/doc/redis-2.8.13/CONTRIBUTING

/usr/share/doc/redis-2.8.13/COPYING

/usr/share/doc/redis-2.8.13/README

/var/lib/redis

/var/log/redis

/var/run/redis

[root@wyzc ~]#

认识 redis 目录结构与文件

熟悉/etc/redis.conf

echo “vm.overcommit_memory=1” >>/etc/sysctl.conf

#此参数可用的值为 0,1,2

#0 表示当用户空间请求更多的内存时，内核尝试估算出可用的内存

#1 表示内核允许超量使用内存直到内存用完为止

#2 表示整个内存地址空间不能超过 swap+(vm.overcommit_ratio)%的 RAM 值

启动与关闭 redis 服务

redis-server 服务启动命令：

版本 2.4:

```
[root@wyzc ~]# /usr/sbin/redis-server   --help
Usage: ./redis-server [/path/to/redis.conf]
        ./redis-server - (read config from stdin)
        ./redis-server --test-memory <megabytes>
[root@wyzc ~]#
```

版本 2.8:

```
[root@wyzc ~]# redis-server    -h
Usage: ./redis-server [/path/to/redis.conf] [options]
        ./redis-server - (read config from stdin)
```

./redis-server -v or --version

./redis-server -h or --help

./redis-server --test-memory <megabytes>

Examples:

./redis-server (run the server with default conf)

./redis-server /etc/redis/6379.conf

./redis-server --port 7777

./redis-server --port 7777 --slaveof 127.0.0.1 8888

./redis-server /etc/myredis.conf --loglevel verbose

Sentinel mode:

./redis-server /etc/sentinel.conf --sentinel

[root@wyzc ~]#

服务脚本启动命令：

[root@wyzc ~]# /etc/init.d/redis start

如是 rhel7

[root@wyzc ~]# systemctl start redis.service

停止服务：

[root@wyzc ~]#redis-cli shutdown

或

[root@wyzc ~]# /etc/init.d/redis start

如是 rhel7

[root@wyzc ~]# systemctl stop redis.service

状态：

[root@wyzc ~]# /etc/init.d/redis status

Redis is running (9433)

或

[root@wyzc ~]# ps -ef|grep redis

root 9433 1 0 11:57 ? 00:00:00 /usr/sbin/redis-server *:6379

root 9441 6427 0 11:58 pts/1 00:00:00 grep redis

如是 rhel7

[root@wyzc ~]# systemctl status redis.service

启动故障分析与排除

看错误日志提示，读 redis 日志文件 。/var/log/message

升级 redis 版本

平滑升级的步骤

1.1 新建一个新版的 redis。
1.2 配置文件修改，作为 slave 进行配置。
1.3 启动 slave，并做好数据测试。
1.4 将原来程序的 redis 读写迁移到新的 slave 上。
1.5 升级原来的版本到新版本

监控服务

-Sentinel

Sentinel 是 Redis 自带的工具，它可以对 Redis 主从复制进行监控，并实现主挂掉之后的自动故障转移。在转移的过程中，它还可以被配置去执行一个用户自定义的脚本，在脚本中我们就能够实现报警通知等功能。

-Redis Live

Redis Live 是一个更通用的 Redis 监控方案，它的原理是定时在 Redis 上执行 MONITOR 命令，来获取当前 Redis 当前正在执行的命令，并通过统计分析，生成 web 页面的可视化分析报表。

-Redis Faina

Redis Faina 是由著名的图片分享应用 instagram 开发的 Redis 监控服务，其原理和 Redis Live 类似，都是对通过 MONITOR 来做的。

数据分布

弄清 Redis 中数据存储分布是一件很难的是，比如你想知道哪类型的 key 值占用内存最多。下面是一些工具，可以帮助你对 Redis 的数据集进行分析。

-Redis-audit

Redis-audit 是一个脚本，通过它，我们可以知道每一类 key 对内存的使用量。它可以提供的数据有：某一类 key 值的访问频率如何，有多少值设置了过期时间，某一类 key 值使用内存的大小，这很方便让我们能排查哪些 key 不常用或者压根不用。

-Redis-rdb-tools

Redis-rdb-tools 跟 Redis-audit 功能类似，不同的是它是通过对 rdb 文件进行分析来取得统计数据的。

Redmon 安装：

redmon 监控：

```
curl -L get.rvm.io | bash -s stable
```

```
/usr/local/rvm/bin/rvm    install    1.9.3
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/gem update --system
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/gem    install    bundler
git clone https://github.com/steelThread/redmon.git
cd redmon
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/bundler install
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/gem install    redmon
cd /usr/local/redmon/
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/bundle show redmon
/usr/local/redmon
/usr/local/rvm/rubies/ruby-1.9.3-p547/bin/bundle exec    bin/redmon
```

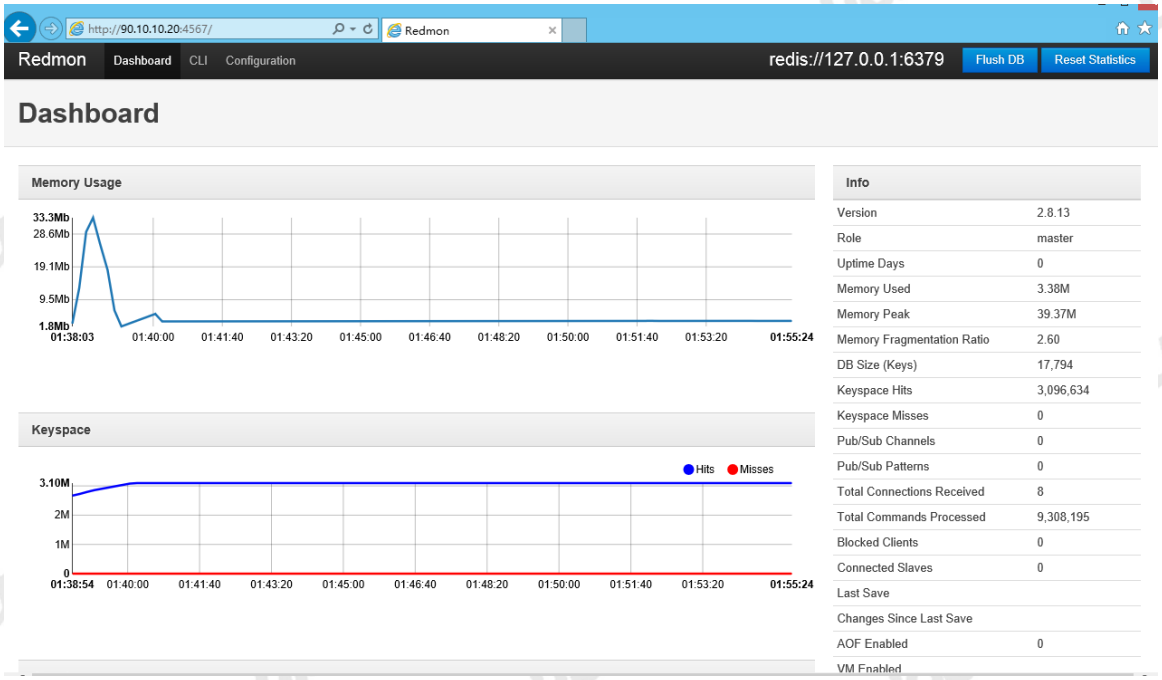
注意配置文件:

```
[root@wyzc redmon]# cat    lib/redmon/config.rb
module Redmon
  def self.configure
    yield config
  end

  def self.config
    @config ||= Config.new
  end

  class Config
    DEFAULTS = {
      :namespace        => 'redmon',
      :redis_url        => 'redis://127.0.0.1:6379',
      :app              => true,
      :worker           => true,
      :web_interface => ['0.0.0.0', 4567],
      :poll_interval => 10,
      :data_lifespan => 30,
      :secure           => false
    }

    attr_accessor(*DEFAULTS.keys)

    def initialize
      apply DEFAULTS
    end

    def apply(opts)
      opts.each { |k,v| send("#{k}=", v) if respond_to? k }
    end
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
    end
end
[root@wyzc redmon]#
```

打开浏览器即可访问到：



Redis-stat 监控：

git clone https://github.com/junegunn/redis-stat.git

cd redis-stat

```
[root@wyzc redis-stat]# ls
bin      Gemfile  lib  LICENSE  Rakefile  README.md  redis-stat.gemspec
screenshots  test
[root@wyzc redis-stat]# /usr/local/rvm/rubies/ruby-1.9.3-p547/bin/gem  install redis-stat
ERROR:  Could not find a valid gem 'redis-stat' (>= 0), here is why:
          Unable to download data from https://rubygems.org/ - Errno::ECONNREFUSED:
Connection refused - connect(2) (https://api.rubygems.org/specs.4.8.gz)
[root@wyzc redis-stat]# /usr/local/rvm/rubies/ruby-1.9.3-p547/bin/bundle  install
Don't run Bundler as root. Bundler can ask for sudo if it is needed, and installing your
bundle as root will break this application for all non-root users on this machine.
fatal: Not a git repository (or any of the parent directories): .git
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
Installing ansi256 0.2.5
Using daemons 1.1.9
Using multi_json 1.10.1
Installing elasticsearch-api 1.0.5
Installing multipart-post 2.0.0
```

```
Installing faraday 0.9.0
Installing elasticsearch-transport 1.0.5
Installing elasticsearch 1.0.5
Using eventmachine 1.0.3
Installing insensitive_hash 0.3.3
Installing json 1.7.7
Installing option_initializer 1.5.1
Installing lps 0.2.1
Installing parallelize 0.4.1
Using rack 1.5.2
Using rack-protection 1.5.3
Installing redis 3.0.7
Installing si 0.1.4
Using tilt 1.4.1
Using sinatra 1.3.6
Installing unicode-display_width 0.1.1
Installing tabularize 0.2.9
Using thin 1.5.1
Using redis-stat 0.4.6 from source at .
Using bundler 1.7.2
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
[root@wyzc redis-stat]# /usr/local/rvm/rubies/ruby-1.9.3-p547/bin/bundle show    redis-stat
fatal: Not a git repository (or any of the parent directories): .git
/usr/local/redis-stat
[root@wyzc redis-stat]# /usr/local/rvm/rubies/ruby-1.9.3-p547/bin/gem install    redis-stat
Fetching: redis-stat-0.4.6.gem (100%)
Successfully installed redis-stat-0.4.6
Installing ri documentation for redis-stat-0.4.6
1 gem installed
[root@wyzc redis-stat]#
[root@wyzc redis-stat]# ./bin/redis-stat    --help
usage: redis-stat [HOST[:PORT] ...] [INTERVAL [COUNT]]

    -a, --auth=PASSWORD              Password
    -v, --verbose                    Show more info
        --style=STYLE                Output style: unicode|ascii
        --no-color                   Suppress ANSI color codes
        --csv=OUTPUT_CSV_FILE_PATH   Save the result in CSV format
        --es=ELASTICSEARCH_URL       Send results to ElasticSearch:
[http://]HOST[:PORT][/INDEX]

        --server[=PORT]              Launch redis-stat web server (default port:
63790)
        --daemon                     Daemonize redis-stat. Must be used with
```

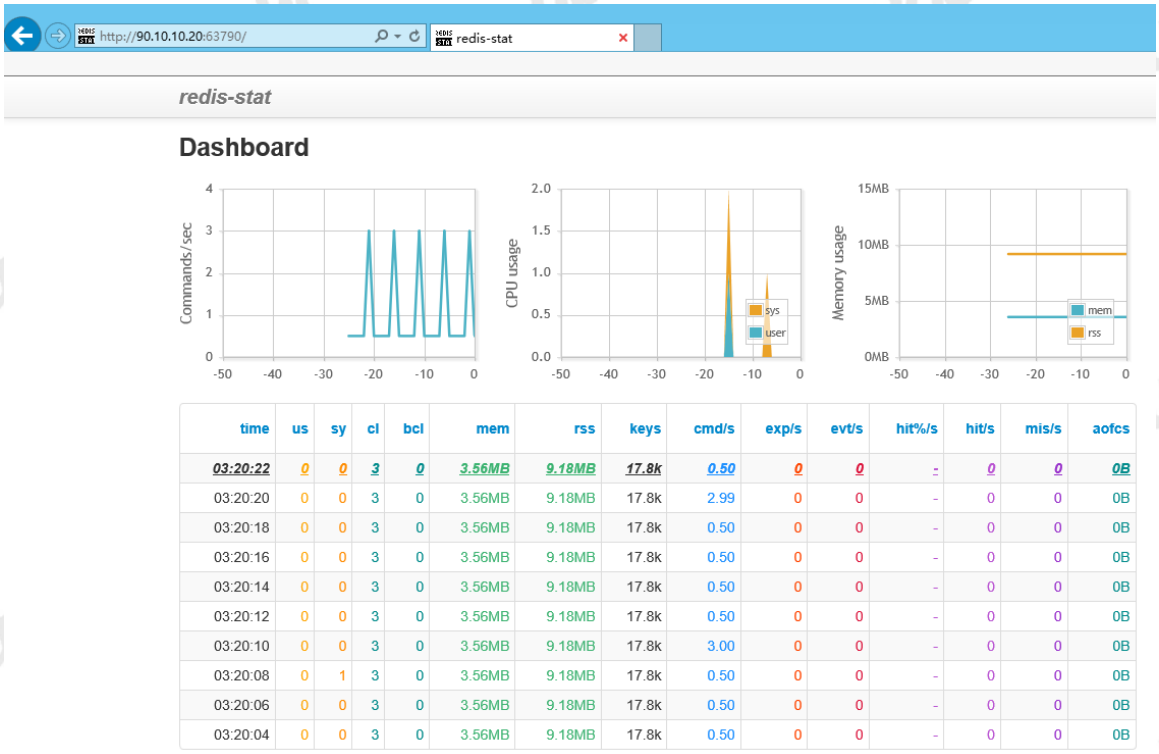
```
--server option.

        --version                   Show version
        --help                      Show this message
[root@wyzc redis-stat]#
[root@wyzc redis-stat]# ./bin/redis-stat   2   5
┌──────────────────────────────────────────────────┬──────────────────────┐
│                                  127.0.0.1:6379  │
├──────────────────────────────────────────────────┼──────────────────────┤
│                   redis_version  │               2.8.13  │
│                      redis_mode  │         standalone    │
│                      process_id  │               3027    │
│              uptime_in_seconds   │               8844    │
│                 uptime_in_days   │                  0    │
│                           role   │             master    │
│              connected_slaves    │                  0    │
│                     aof_enabled  │                  0    │
│     rdb_bgsave_in_progress       │                  0    │
│            rdb_last_save_time    │         1409340074    │
└──────────────────────────────────────────────────┴──────────────────────┘

┌──────────┬──┬──┬──┬──┬─────────┬─────────┬─────────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│     time │us│sy│cl│bcl│      mem│      rss│     keys│cmd/s │exp/s │evt/s │hit%/s│hit/s │mis/s │aofcs │
├──────────┼──┼──┼──┼──┼─────────┼─────────┼─────────┼──────┼──────┼──────┼──────┼──────┼──────┼──────┤
│ 03:23:09 │- │- │3 │0 │3.56MB   │9.18MB   │17.8k    │-     │-     │-     │-     │-     │-     │-     │
0B
│ 03:23:11 │0 │0 │3 │0 │3.56MB   │9.18MB   │17.8k    │2.99  │0     │0     │-     │0     │0     │0     │
0B
│ 03:23:13 │0 │0 │3 │0 │3.56MB   │9.18MB   │17.8k    │0.50  │0     │0     │-     │0     │0     │0     │
0B
│ 03:23:15 │0 │0 │3 │0 │3.56MB   │9.18MB   │17.8k    │0.50  │0     │0     │-     │0     │0     │0     │
0B
│ 03:23:17 │0 │0 │3 │0 │3.56MB   │9.18MB   │17.8k    │0.50  │0     │0     │-     │0     │0     │0     │
0B
└──────────┴──┴──┴──┴──┴─────────┴─────────┴─────────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘
Elapsed: 8.04 sec.
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

[root@wyzc redis-stat]#

[root@wyzc redis-stat]# ./bin/redis-stat --server

打开浏览器访问：



time	us	sy	cl	bcl	mem	rss	keys	cmd/s	exp/s	evt/s	hit%/s	hit/s	mis/s	aofcs
03:20:22	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:20	0	0	3	0	3.56MB	9.18MB	17.8k	2.99	0	0	-	0	0	0B
03:20:18	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:16	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:14	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:12	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:10	0	0	3	0	3.56MB	9.18MB	17.8k	3.00	0	0	-	0	0	0B
03:20:08	0	1	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:06	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B
03:20:04	0	0	3	0	3.56MB	9.18MB	17.8k	0.50	0	0	-	0	0	0B

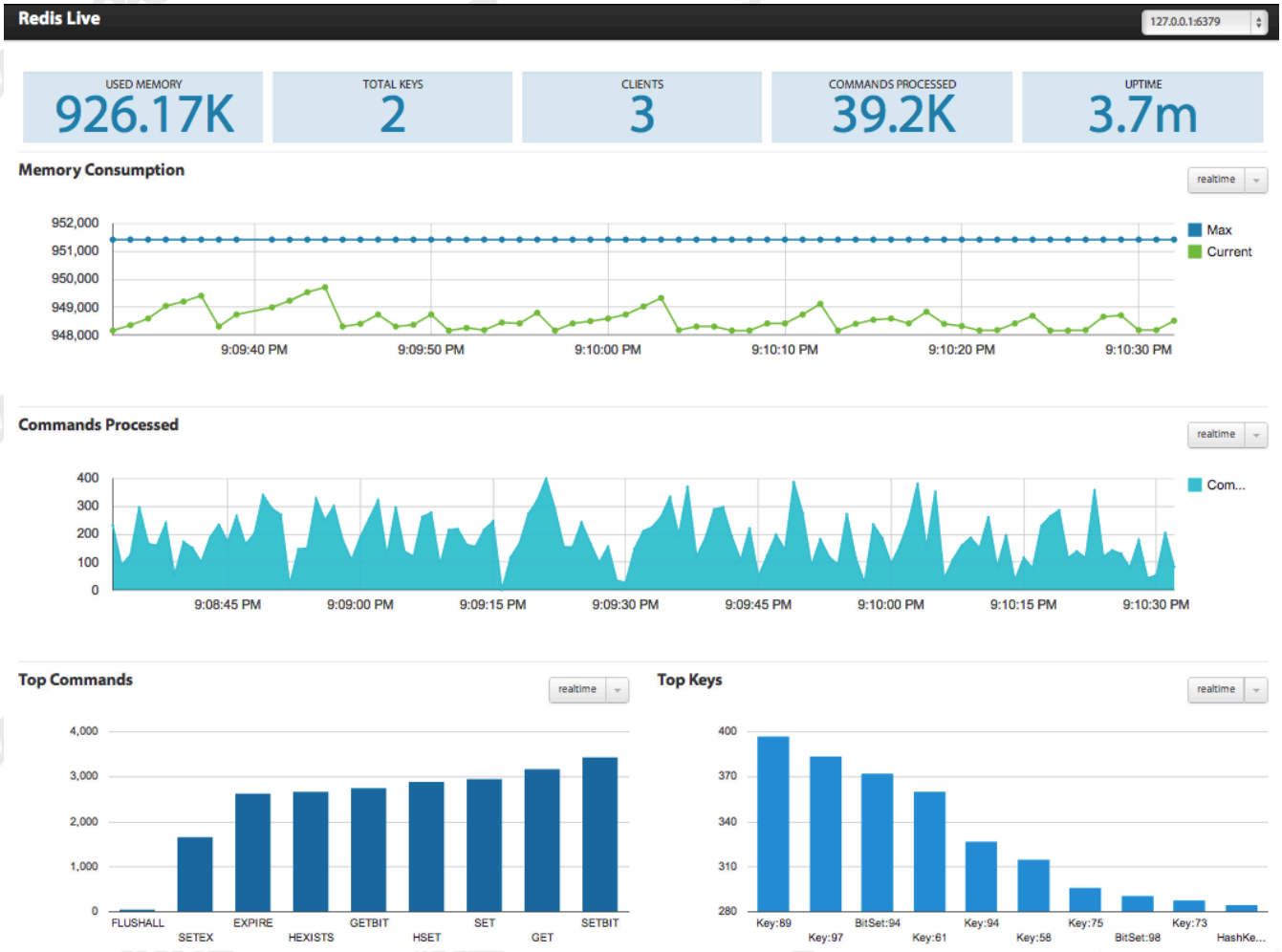
Redis-live 监控：

```
wget http://dl.fedoraproject.org/pub/epel/6/x86_64/python-pip-0.8-1.el6.noarch.rpm
   rpm -ivh python-pip-0.8-1.el6.noarch.rpm
   pip-python install tornado
   pip-python install redis
     pip-python install python-dateutil
   pip-python install argparse
   git clone https://github.com/kumarnitin/RedisLive.git
   cd RedisLive/src
   vim redis-live.conf

{
            "RedisServers":
            [
                        {
                            "server": "192.168.1.161",
                            "port" : 6378
                        },
                        {
                            "server" : "192.168.1.78",
```

```
                        "port"    : 6380
                    },
        ],

        "DataStoreType" : "sqlite",

        "RedisStatsServer":
        {
                    "server" : "127.0.0.1",
                    "port" : 6381
        }
}
```

```
./redis-monitor.py --duration   120   &
./redis-live.py   &
```

打开浏览器访问 ip:8888 即可。但前提是你的服务器可以访问到 google 业务。因为绘图使用 google 的 chart 在线绘制功能。不管是无法访问外网还是被墙，最后你看不到你需要的监控效果的。



配置文件详解

```
# Redis 示例配置文件

# 注意单位问题：当需要设置内存大小的时候，可以使用类似 1k、5GB、4M 这样的常见格式：
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# 单位是大小写不敏感的，所以 1GB 1Gb 1gB 的写法都是完全一样的。

# Redis 默认是不作为守护进程来运行的。你可以把这个设置为"yes"让它作为守护进程来运行。
# 注意，当作为守护进程的时候，Redis 会把进程 ID 写到 /var/run/redis.pid
daemonize no

# 当以守护进程方式运行的时候，Redis 会把进程 ID 默认写到 /var/run/redis.pid。你可以在这里修改路径。
pidfile /var/run/redis.pid

# 接受连接的特定端口，默认是 6379。
# 如果端口设置为 0，Redis 就不会监听 TCP 套接字。
port 6379

# 如果你想的话，你可以绑定单一接口；如果这里没单独设置，那么所有接口的连接都会被监听。
#
# bind 127.0.0.1

# 指定用来监听连接的 unxi 套接字的路径。这个没有默认值，所以如果你不指定的话，Redis 就不会通过 unix
套接字来监听。
#
# unixsocket /tmp/redis.sock
# unixsocketperm 755

#一个客户端空闲多少秒后关闭连接。(0 代表禁用，永不关闭)
timeout 0

# 设置服务器调试等级。
# 可能值：
# debug （很多信息，对开发/测试有用）
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
# verbose（很多精简的有用信息，但是不像 debug 等级那么多）
# notice（适量的信息，基本上是你生产环境中需要的程度）
# warning（只有很重要/严重的信息会记录下来）
loglevel verbose

# 指明日志文件名。也可以使用"stdout"来强制让 Redis 把日志信息写到标准输出上。
# 注意：如果 Redis 以守护进程方式运行，而你设置日志显示到标准输出的话，那么日志会发送到 /dev/null
logfile stdout

# 要使用系统日志记录器很简单，只要设置 "syslog-enabled" 为 "yes" 就可以了。
# 然后根据需要设置其他一些 syslog 参数就可以了。
# syslog-enabled no

# 指明 syslog 身份
# syslog-ident redis

# 指明 syslog 的设备。必须是一个用户或者是 LOCAL0 ~ LOCAL7 之一。
# syslog-facility local0

# 设置数据库个数。默认数据库是 DB 0，你可以通过 SELECT <dbid> WHERE dbid（0～'databases' - 1）来为每个连接使用不同的数据库。
databases 16

################################## 快照 ###################################

#
# 把数据库存到磁盘上:
#
#   save <seconds> <changes>
#
#   会在指定秒数和数据变化次数之后把数据库写到磁盘上。
#
#   下面的例子将会进行把数据写入磁盘的操作:
#   900 秒（15 分钟）之后，且至少 1 次变更
#   300 秒（5 分钟）之后，且至少 10 次变更
#   60 秒之后，且至少 10000 次变更
#
#   注意：你要想不写磁盘的话就把所有 "save" 设置注释掉就行了。

save 900 1
save 300 10
save 60 10000

# 当导出到 .rdb 数据库时是否用 LZF 压缩字符串对象。
# 默认设置为 "yes"，所以几乎总是生效的。
```

```
# 如果你想节省 CPU 的话你可以把这个设置为 "no"，但是如果你有可压缩的 key 的话，那数据文件就会更大了。
rdbcompression yes

# 数据库的文件名
dbfilename dump.rdb

# 工作目录
#
# 数据库会写到这个目录下，文件名就是上面的 "dbfilename" 的值。
#
# 累加文件也放这里。
#
# 注意你这里指定的必须是目录，不是文件名。
dir ./

################################## 同步 #################################

#
# 主从同步。通过 slaveof 配置来实现 Redis 实例的备份。
# 注意，这里是本地从远端复制数据。也就是说，本地可以有不同的数据库文件、绑定不同的 IP、监听不同的端口。
#
# slaveof <masterip> <masterport>

# 如果 master 设置了密码（通过下面的 "requirepass" 选项来配置），那么 slave 在开始同步之前必须进行身份验证，否则它的同步请求会被拒绝。
#
# masterauth <master-password>

# 当一个 slave 失去和 master 的连接，或者同步正在进行中，slave 的行为有两种可能：
#
# 1) 如果 slave-serve-stale-data 设置为 "yes" (默认值)，slave 会继续响应客户端请求，可能是正常数据，也可能是还没获得值的空数据。
# 2) 如果 slave-serve-stale-data 设置为 "no"，slave 会回复"正在从 master 同步（SYNC with master in progress）"来处理各种请求，除了 INFO 和 SLAVEOF 命令。
#
slave-serve-stale-data yes

# 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

# 下面的选项设置了大块数据 I/O、向 master 请求数据和 ping 响应的过期时间。
```

```
# 默认值 60 秒。
#
# 一个很重要的事情是：确保这个值比 repl-ping-slave-period 大，否则 master 和 slave 之间的传输过期时间比预想的要短。
#
# repl-timeout 60

################################## 安全 ###################################

# 要求客户端在处理任何命令时都要验证身份和密码。
# 这在你信不过来访者时很有用。
#
# 为了向后兼容的话，这段应该注释掉。而且大多数人不需要身份验证（例如：它们运行在自己的服务器上。）
#
# 警告：因为 Redis 太快了，所以居心不良的人可以每秒尝试 150k 的密码来试图破解密码。
# 这意味着你需要一个高强度的密码，否则破解太容易了。
#
# requirepass foobared

# 命令重命名
#
# 在共享环境下，可以为危险命令改变名字。比如，你可以为 CONFIG 改个其他不太容易猜到的名字，这样你自己仍然可以使用，而别人却没法做坏事了。
#
# 例如:
#
# rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52
#
# 甚至也可以通过给命令赋值一个空字符串来完全禁用这条命令：
#
# rename-command CONFIG ""

################################## 限制 ###################################

#
# 设置最多同时连接客户端数量。
# 默认没有限制，这个关系到 Redis 进程能够打开的文件描述符数量。
# 特殊值"0"表示没有限制。
# 一旦达到这个限制，Redis 会关闭所有新连接并发送错误"达到最大用户数上限（max number of clients reached）"
#
# maxclients 128

# 不要用比设置的上限更多的内存。一旦内存使用达到上限，Redis 会根据选定的回收策略（参见：maxmemmory-policy）删除 key。
```

```
#
# 如果因为删除策略问题 Redis 无法删除 key，或者策略设置为 "noeviction"，Redis 会回复需要更多内存的错误信息给命令。
# 例如，SET,LPUSH 等等。但是会继续合理响应只读命令，比如：GET。
#
# 在使用 Redis 作为 LRU 缓存，或者为实例设置了硬性内存限制的时候（使用 "noeviction" 策略）的时候，这个选项还是满有用的。
#
# 警告：当一堆 slave 连上达到内存上限的实例的时候，响应 slave 需要的输出缓存所需内存不计算在使用内存当中。
# 这样当请求一个删除掉的 key 的时候就不会触发网络问题 / 重新同步的事件，然后 slave 就会收到一堆删除指令，直到数据库空了为止。
#
# 简而言之，如果你有 slave 连上一个 master 的话，那建议你把 master 内存限制设小点儿，确保有足够的系统内存用作输出缓存。
# （如果策略设置为"noeviction"的话就不无所谓了）
#
# maxmemory <bytes>

# 内存策略：如果达到内存限制了，Redis 如何删除 key。你可以在下面五个策略里面选：
#
# volatile-lru -> 根据 LRU 算法生成的过期时间来删除。
# allkeys-lru -> 根据 LRU 算法删除任何 key。
# volatile-random -> 根据过期设置来随机删除 key。
# allkeys->random -> 无差别随机删。
# volatile-ttl -> 根据最近过期时间来删除（辅以 TTL）
# noeviction -> 谁也不删，直接在写操作时返回错误。
#
# 注意：对所有策略来说，如果 Redis 找不到合适的可以删除的 key 都会在写操作时返回一个错误。
#
#     这里涉及的命令：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
#
# 默认值如下：
#
# maxmemory-policy volatile-lru

# LRU 和最小 TTL 算法的实现都不是很精确，但是很接近（为了省内存），所以你可以用样例做测试。
# 例如：默认 Redis 会检查三个 key 然后取最旧的那个，你可以通过下面的配置项来设置样本的个数。
#
# maxmemory-samples 3
```

```
################################ 纯累加模式 ################################

# 默认情况下，Redis 是异步的把数据导出到磁盘上。这种情况下，当 Redis 挂掉的时候，最新的数据就丢了。
# 如果不希望丢掉任何一条数据的话就该用纯累加模式：一旦开启这个模式，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 都会把这个文件的数据读入内存里。
#
# 注意，异步导出的数据库文件和纯累加文件可以并存（你得把上面所有"save"设置都注释掉，关掉导出机制）。
# 如果纯累加模式开启了，那么 Redis 会在启动时载入日志文件而忽略导出的 dump.rdb 文件。
#
# 重要：查看 BGREWRITEAOF 来了解当累加日志文件太大了之后，怎么在后台重新处理这个日志文件。

appendonly no

# 纯累加文件名字（默认："appendonly.aof"）
# appendfilename appendonly.aof

# fsync() 请求操作系统马上把数据写到磁盘上，不要再等了。
# 有些操作系统会真的把数据马上刷到磁盘上；有些则要磨蹭一下，但是会尽快去做。
#
# Redis 支持三种不同的模式：
#
# no：不要立刻刷，只有在操作系统需要刷的时候再刷。比较快。
# always：每次写操作都立刻写入到 aof 文件。慢，但是最安全。
# everysec：每秒写一次。折衷方案。
#
# 默认的 "everysec" 通常来说能在速度和数据安全性之间取得比较好的平衡。
# 如果你真的理解了这个意味着什么，那么设置"no"可以获得更好的性能表现（如果丢数据的话，则只能拿到一个不是很新的快照）；
# 或者相反的，你选择 "always" 来牺牲速度确保数据安全、完整。
#
# 如果拿不准，就用 "everysec"

# appendfsync always
appendfsync everysec
# appendfsync no

# 如果 AOF 的同步策略设置成 "always" 或者 "everysec"，那么后台的存储进程（后台存储或写入 AOF 日志）会产生很多磁盘 I/O 开销。
# 某些 Linux 的配置下会使 Redis 因为 fsync() 而阻塞很久。
# 注意，目前对这个情况还没有完美修正，甚至不同线程的 fsync() 会阻塞我们的 write(2) 请求。
#
# 为了缓解这个问题，可以用下面这个选项。它可以在 BGSAVE 或 BGREWRITEAOF 处理时阻止 fsync()。
#
# 这就意味着如果有子进程在进行保存操作，那么 Redis 就处于"不可同步"的状态。
```

```
# 这实际上是说，在最差的情况下可能会丢掉 30 秒钟的日志数据。（默认 Linux 设定）
#
# 如果你有延迟的问题那就把这个设为 "yes"，否则就保持 "no"，这是保存持久数据的最安全的方式。
no-appendfsync-on-rewrite no

# 自动重写 AOF 文件
#
# 如果 AOF 日志文件大到指定百分比，Redis 能够通过 BGREWRITEAOF 自动重写 AOF 日志文件。
#
# 工作原理：Redis 记住上次重写时 AOF 日志的大小（或者重启后没有写操作的话，那就直接用此时的 AOF 文件），
#       基准尺寸和当前尺寸做比较。如果当前尺寸超过指定比例，就会触发重写操作。
#
# 你还需要指定被重写日志的最小尺寸，这样避免了达到约定百分比但尺寸仍然很小的情况还要重写。
#
# 指定百分比为 0 会禁用 AOF 自动重写特性。

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

################################## 慢查询日志
#####################################

# Redis 慢查询日志可以记录超过指定时间的查询。运行时间不包括各种 I/O 时间。
# 例如：连接客户端，发送响应数据等。只计算命令运行的实际时间（这是唯一一种命令运行线程阻塞而无法同时为其他请求服务的场景）
#
# 你可以为慢查询日志配置两个参数：一个是超标时间，单位为微妙，记录超过个时间的命令。
# 另一个是慢查询日志长度。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。
#
# 下面的时间单位是微秒，所以 1000000 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

# 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。（译者注：日志居然是在内存里的 Orz）
slowlog-max-len 128

################################ 虚拟内存 ###############################

### 警告！虚拟内存在 Redis 2.4 是反对的。
### 非常不鼓励使用虚拟内存!!

# 虚拟内存可以使 Redis 在内存不够的情况下仍然可以将所有数据序列保存在内存里。
# 为了做到这一点，高频 key 会调到内存里，而低频 key 会转到交换文件里，就像操作系统使用内存页一样。
```

```
#
# 要使用虚拟内存，只要把 "vm-enabled" 设置为 "yes"，并根据需要设置下面三个虚拟内存参数就可以了。

vm-enabled no
# vm-enabled yes

# 这是交换文件的路径。估计你猜到了，交换文件不能在多个 Redis 实例之间共享，所以确保每个 Redis 实例使用一个独立交换文件。
#
# 最好的保存交换文件（被随机访问）的介质是固态硬盘（SSD）。
#
# *** 警告 *** 如果你使用共享主机，那么默认的交换文件放到 /tmp 下是不安全的。
# 创建一个 Redis 用户可写的目录，并配置 Redis 在这里创建交换文件。
vm-swap-file /tmp/redis.swap

# "vm-max-memory" 配置虚拟内存可用的最大内存容量。
# 如果交换文件还有空间的话，所有超标部分都会放到交换文件里。
#
# "vm-max-memory" 设置为 0 表示系统会用掉所有可用内存。
# 这默认值不咋地，只是把你能用的内存全用掉了，留点余量会更好。
# 例如，设置为剩余内存的 60%-80%。
vm-max-memory 0

# Redis 交换文件是分成多个数据页的。
# 一个可存储对象可以被保存在多个连续页里，但是一个数据页无法被多个对象共享。
# 所以，如果你的数据页太大，那么小对象就会浪费掉很多空间。
# 如果数据页太小，那用于存储的交换空间就会更少（假定你设置相同的数据页数量）
#
# 如果你使用很多小对象，建议分页尺寸为 64 或 32 个字节。
# 如果你使用很多大对象，那就用大一些的尺寸。
# 如果不确定，那就用默认值 :)
vm-page-size 32

# 交换文件里数据页总数。
# 根据内存中分页表（已用/未用的数据页分布情况），磁盘上每 8 个数据页会消耗内存里 1 个字节。
#
# 交换区容量 = vm-page-size * vm-pages
#
# 根据默认的 32 字节的数据页尺寸和 134217728 的数据页数来算，Redis 的数据页文件会占 4GB，而内存里的分页表会消耗 16MB 内存。
#
# 为你的应验程序设置最小且够用的数字比较好，下面这个默认值在大多数情况下都是偏大的。
vm-pages 134217728

# 同时可运行的虚拟内存 I/O 线程数。
```

```
# 这些线程可以完成从交换文件进行数据读写的操作，也可以处理数据在内存与磁盘间的交互和编码/解码处理。
# 多一些线程可以一定程度上提高处理效率，虽然 I/O 操作本身依赖于物理设备的限制，不会因为更多的线程而提高单次读写操作的效率。
#
# 特殊值 0 会关闭线程级 I/O，并会开启阻塞虚拟内存机制。
vm-max-threads 4

################################ 高级配置 ################################

# 当有大量数据时，适合用哈希编码（需要更多的内存），元素数量上限不能超过给定限制。
# 你可以通过下面的选项来设定这些限制：
hash-max-zipmap-entries 512
hash-max-zipmap-value 64

# 与哈希相类似，数据元素较少的情况下，可以用另一种方式来编码从而节省大量空间。
# 这种方式只有在符合下面限制的时候才可以用：
list-max-ziplist-entries 512
list-max-ziplist-value 64

# 还有这样一种特殊编码的情况：数据全是 64 位无符号整型数字构成的字符串。
# 下面这个配置项就是用来限制这种情况下使用这种编码的最大上限的。
set-max-intset-entries 512

# 与第一、第二种情况相似，有序序列也可以用一种特别的编码方式来处理，可节省大量空间。
# 这种编码只适合长度和元素都符合下面限制的有序序列：
zset-max-ziplist-entries 128
zset-max-ziplist-value 64

# 哈希刷新，每 100 个 CPU 毫秒会拿出 1 个毫秒来刷新 Redis 的主哈希表（顶级键值映射表）。
# redis 所用的哈希表实现（见 dict.c）采用延迟哈希刷新机制：你对一个哈希表操作越多，哈希刷新操作就越频繁；
# 反之，如果服务器非常不活跃那么也就是用点内存保存哈希表而已。
#
# 默认是每秒钟进行 10 次哈希表刷新，用来刷新字典，然后尽快释放内存。
#
# 建议：
# 如果你对延迟比较在意的话就用 "activerehashing no"，每个请求延迟 2 毫秒不太好嘛。
# 如果你不太在意延迟而希望尽快释放内存的话就设置 "activerehashing yes"。
activerehashing yes

################################## 包含 ###################################

# 包含一个或多个其他配置文件。
# 这在你有标准配置模板但是每个 redis 服务器又需要个性设置的时候很有用。
```

```
# 包含文件特性允许你引人其他配置文件，所以好好利用吧。
#
# include /path/to/local.conf
# include /path/to/other.conf
```

Redis 可以在没有配置文件的情况下通过内置的配置来启动，但是这种启动方式只适用于开发和测试。

合理的配置 Redis 的方式是提供一个 Redis 配置文件，这个文件通常叫做 `redis.conf`。

`redis.conf` 文件中包含了很多格式简单的指令如下：

```
关键字 参数 1 参数 2 ...参数 N
```

如下是一个配置指令的示例：

```
slaveof 127.0.0.1 6380
```

如果参数中含有空格，那么可以用双引号括起来，如下：

```
requirepass "hello world"
```

这些指令的配置，意义以及深入使用方法都能在每个 Redis 发布版本自带的的 redis.conf 文档中找到。

- 自描述文档 redis.conf for Redis 2.6.
- 自描述文档 redis.conf for Redis 2.4.

通过命令行传参

自 Redis2.6 起就可以直接通过命令行传递 Redis 配置参数。这种方法可以用于测试。 以下是一个例子：这个例子配置一个新运行并以 6380 为端口的 Redis 实例，使配置它为 127.0.0.1:6379Redis 实例的 slave。

```
./redis-server --port 6380 --slaveof 127.0.0.1 6379
```

通过命令行传递的配置参数的格式和在 redis.conf 中设置的配置参数的格式完全一样， 唯一不同的是需要在关键字之前加上 前缀`--`。

需要注意的是通过命令行传递参数的过程会在内存中生成一个临时的配置文件(也许会直接追加在 命令指定的配置文件后面)，这些传递的参数也会转化为跟 Redis 配置文件一样的形式。

运行时配置更改

Redis 允许在运行的过程中，在不重启服务器的情况下更改服务器配置，同时也支持 使用特殊的 CONFIG SET

和 CONFIG GET 命令用编程方式查询并设置配置。

并非所有的配置指令都支持这种使用方式，但是大部分是支持的。更多相关的信息请查阅 CONFIG SET and CONFIG GET 页面。

需要确保的是在通过CONFIG SET 命令进行的设置的同时，也需在 `redis.conf` 文件中进行了相应的更改。未来 Redis 有计划提供一个 `CONFIG REWRITE` 命令在不更改现有配置文件的同时， 根据当下的服务器配置对 `redis.conf` 文件进行重写。

配置 Redis 成为一个缓存

如果你想把 Redis 当做一个缓存来用，所有的 key 都有过期时间，那么你可以考虑 使用以下设置（假设最大内存使用量为 2M）：

```
maxmemory 2mb
```

```
maxmemory-policy allkeys-lru
```

以上设置并不需要我们的应用使用 EXPIRE(或相似的命令)命令去设置每个 key 的过期时间，因为 只要内存使用量到达 2M，Redis 就会使用类 LRU 算法自动删除某些 key。

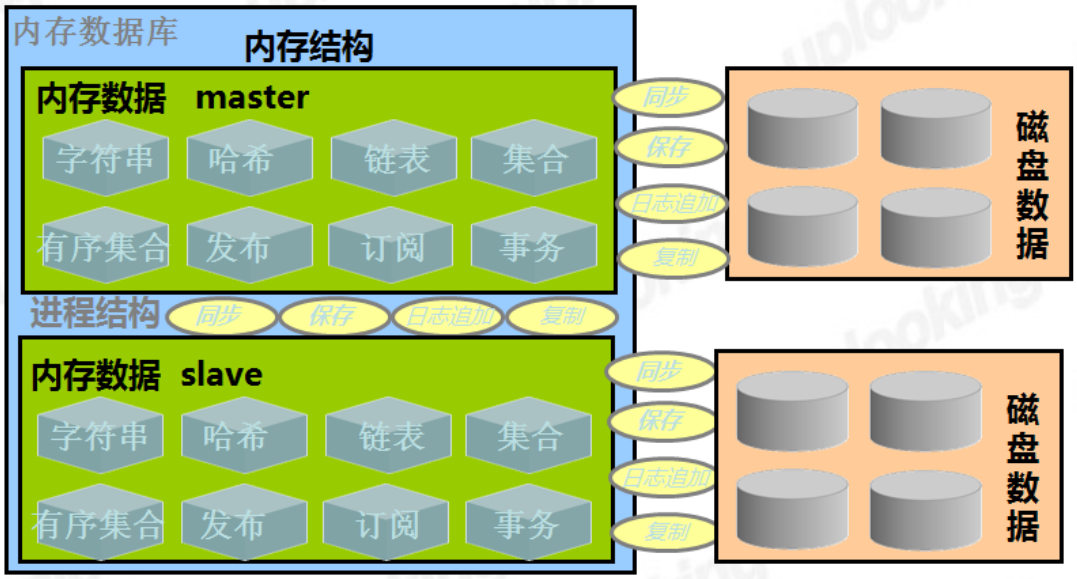
相比使用额外内存空间存储多个键的过期时间，使用缓存设置是一种更加有效利用内存的方式。而且相比每个键固定的 过期时间，使用 LRU 也是一种更加推荐的方式，因为这样能使应用的*热数据*(更频繁使用的键) 在内存中停留时间更久。

基本上这么配置下的 Redis 可以当成 memcached 使用。

当我们把 Redis 当成缓存来使用的时候，如果应用程序同时也需要把 Redis 当成存储系统来使用，那么强烈建议 使用两个 Redis 实例。一个是缓存，使用上述方法进行配置，另一个是存储，根据应用的持久化需求进行配置，并且 只存储那些不需要被缓存的数据。

*请注意:*用户需要详细阅读示例 redis.conf 文件来决定使用什么内存上限处理策略。

主从复制架构及部署



主从架构可以本机多实例数据库之间实现，也可以异机多实例之间实现。

主可读可写，备只读，这样就可以实现读写分离的架构。

redis 主从复制的特点：

1.一台 master 可以拥有多个 slave（1 对多的关系）

2.多个 slave 可以连接同一个 master 外，还可以连接到其他 slave（这样做的原因是如果 masterdown 掉之后其中的一台 slave 立马可以充当 master 的角色，这样整个服务流程不受影响）

3.中从复制不会阻塞 master，在同步数据的同时，master 可以继续处理 client 请求。

4.提高系统的伸缩性

redis 主从复制的过程：

1.slave 与 master 建立连接，发送 sync 同步命令。

2.Master 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得写命令并缓存。

3.后台完成保存后，将文件发送给 slave

4.slave 将文件保存到硬盘上

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。

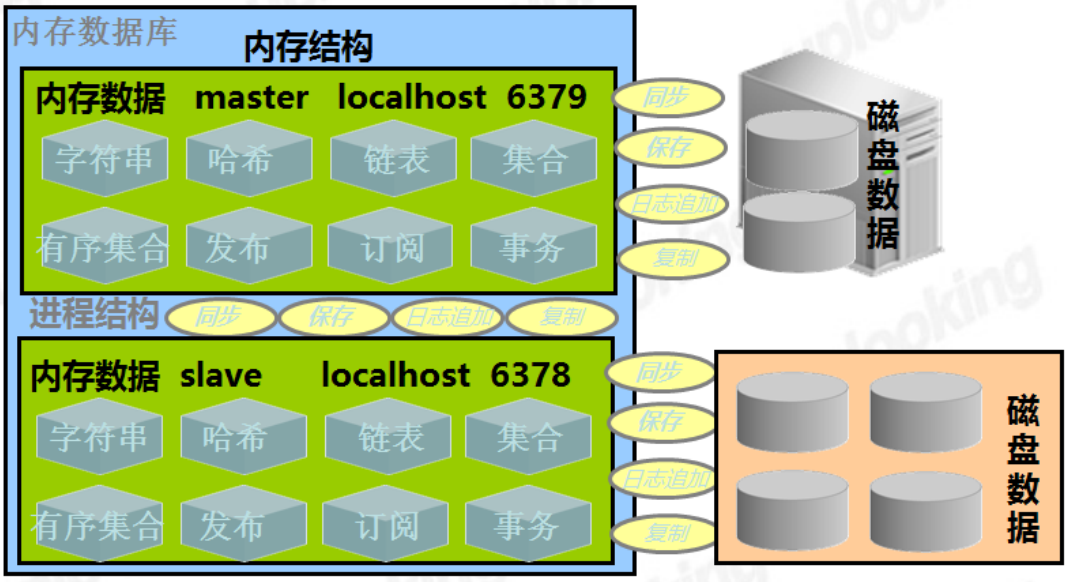
下面介绍下主从复制的过程

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

下面是本机主从架构及实现



主配置文件默认即可。从配置文件修改：

```
[root@wyzc utils]# grep -B 2 -A 2 ^slaveof /etc/redis/6378.conf
#
# slaveof <masterip> <masterport>
slaveof localhost 6379


# If the master is password protected (using the "requirepass" configuration
[root@wyzc utils]#
```

分别启动主从数据库

```
[root@wyzc utils]# /etc/init.d/redis_6378    start
Starting Redis server...
[root@wyzc utils]# /etc/init.d/redis_6379    start
Redis is running (16404)
[root@wyzc utils]#
[root@wyzc utils]# /usr/local/bin/redis-cli    -h localhost    -p 6379 info
# Server
redis_version:2.8.13
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:d6a0ac9b1eb64e85
redis_mode:standalone
os:Linux 2.6.39-400.17.1.el6uek.x86_64 x86_64
arch_bits:64
```

multiplexing_api:epoll

gcc_version:4.4.7

process_id:16404

run_id:c2291f428af0324f1ab1d1ccdc6c3ccce3dcbb61

tcp_port:6379

uptime_in_seconds:943

uptime_in_days:0

hz:10

lru_clock:69782

config_file:/etc/redis/6379.conf

Clients

connected_clients:1

client_longest_output_list:0

client_biggest_input_buf:0

blocked_clients:0

Memory

used_memory:1880672

used_memory_human:1.79M

used_memory_rss:7671808

used_memory_peak:1880672

used_memory_peak_human:1.79M

used_memory_lua:33792

mem_fragmentation_ratio:4.08

mem_allocator:jemalloc-3.6.0

Persistence

loading:0

rdb_changes_since_last_save:0

rdb_bgsave_in_progress:0

rdb_last_save_time:1409355879

rdb_last_bgsave_status:ok

rdb_last_bgsave_time_sec:0

rdb_current_bgsave_time_sec:-1

aof_enabled:0

aof_rewrite_in_progress:0

aof_rewrite_scheduled:0

aof_last_rewrite_time_sec:-1

aof_current_rewrite_time_sec:-1

aof_last_bgrewrite_status:ok

aof_last_write_status:ok

Stats

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

total_connections_received:2

total_commands_processed:48

instantaneous_ops_per_sec:1

rejected_connections:0

sync_full:1

sync_partial_ok:0

sync_partial_err:0

expired_keys:0

evicted_keys:0

keyspace_hits:0

keyspace_misses:0

pubsub_channels:0

pubsub_patterns:0

latest_fork_usec:14339

Replication

role:master

connected_slaves:1

slave0:ip=::1,port=6378,state=online,offset=71,lag=1

master_repl_offset:71

repl_backlog_active:1

repl_backlog_size:1048576

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
repl_backlog_first_byte_offset:2

repl_backlog_histlen:70


# CPU

used_cpu_sys:1.13

used_cpu_user:0.43

used_cpu_sys_children:0.01

used_cpu_user_children:0.00


# Keyspace

db0:keys=2,expires=0,avg_ttl=0

[root@wyzc utils]# /usr/local/bin/redis-cli    -h localhost    -p 6378 info

# Server

redis_version:2.8.13

redis_git_sha1:00000000

redis_git_dirty:0

redis_build_id:d6a0ac9b1eb64e85

redis_mode:standalone

os:Linux 2.6.39-400.17.1.el6uek.x86_64 x86_64

arch_bits:64

multiplexing_api:epoll

gcc_version:4.4.7
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

process_id:16615

run_id:06cdd39fbd1f4a2e417ab8085ced897880c36460

tcp_port:6378

uptime_in_seconds:82

uptime_in_days:0

hz:10

lru_clock:69816

config_file:/etc/redis/6378.conf

Clients

connected_clients:2

client_longest_output_list:0

client_biggest_input_buf:0

blocked_clients:0

Memory

used_memory:831864

used_memory_human:812.37K

used_memory_rss:7806976

used_memory_peak:831864

used_memory_peak_human:812.37K

used_memory_lua:33792

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

mem_fragmentation_ratio:9.38

mem_allocator:jemalloc-3.6.0

Persistence

loading:0

rdb_changes_since_last_save:0

rdb_bgsave_in_progress:0

rdb_last_save_time:1409355878

rdb_last_bgsave_status:ok

rdb_last_bgsave_time_sec:-1

rdb_current_bgsave_time_sec:-1

aof_enabled:0

aof_rewrite_in_progress:0

aof_rewrite_scheduled:0

aof_last_rewrite_time_sec:-1

aof_current_rewrite_time_sec:-1

aof_last_bgrewrite_status:ok

aof_last_write_status:ok

Stats

total_connections_received:1

total_commands_processed:8

instantaneous_ops_per_sec:0

rejected_connections:0

sync_full:0

sync_partial_ok:0

sync_partial_err:0

expired_keys:0

evicted_keys:0

keyspace_hits:0

keyspace_misses:0

pubsub_channels:0

pubsub_patterns:0

latest_fork_usec:0

Replication

role:slave

master_host:localhost

master_port:6379

master_link_status:up

master_last_io_seconds_ago:9

master_sync_in_progress:0

slave_repl_offset:113

slave_priority:100

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
slave_read_only:1

connected_slaves:0

master_repl_offset:0

repl_backlog_active:0

repl_backlog_size:1048576

repl_backlog_first_byte_offset:0

repl_backlog_histlen:0


# CPU

used_cpu_sys:0.12

used_cpu_user:0.04

used_cpu_sys_children:0.00

used_cpu_user_children:0.00


# Keyspace

db0:keys=2,expires=0,avg_ttl=0

[root@wyzc utils]#

[root@wyzc Desktop]# /usr/local/bin/redis-cli   -h localhost -p 6379

localhost:6379> set a 1

OK

localhost:6379> set b 2

OK
```

```
localhost:6379> set c 3

OK

localhost:6379> get a

"1"

localhost:6379> save

OK

localhost:6379> set   d 4

OK

localhost:6379> exit
```

```
[root@wyzc Desktop]# /usr/local/bin/redis-cli   -h localhost -p 6378

localhost:6378> get a

"1"

localhost:6378> get b

"2"

localhost:6378> get c

"3"

localhost:6378> get d

"4"

localhost:6378> exit
```

可以看到主从复制是正常的。

这种方式搭建主从。任意一个其他从都可以与主或与从进行主从架构。也就是说安全性不好。

解决办法可以通过防火墙控制，可以设置主从使用密码验证。

主设置密码：

```
[root@wyzc utils]# grep requirepass /etc/redis/6379.conf
# If the master is password protected (using the "requirepass" configuration
# requirepass foobared
requirepass redis
[root@wyzc utils]#
[root@wyzc utils]# /etc/init.d/redis_6379    stop
Stopping ...
Redis stopped
[root@wyzc utils]# /etc/init.d/redis_6379    start
Starting Redis server...
[root@wyzc utils]#
[root@wyzc utils]# grep    -A 10 stop\)    /etc/init.d/redis_6379
      stop)
          if [ ! -f $PIDFILE ]
          then
                echo "$PIDFILE does not exist, process is not running"
          else
                PID=$(cat $PIDFILE)
                echo "Stopping ..."
```

```
        $CLIEXEC -a redis -p $REDISPORT shutdown
        while [ -x /proc/${PID} ]
        do
            echo "Waiting for Redis to shutdown ..."
[root@wyzc utils]#
```

注意红色部分增加的内容

目前从没有设置密码，应该是无法复制的。

```
[root@wyzc utils]# /usr/local/bin/redis-cli     -a redis -h localhost    -p 6379 info
# Server
redis_version:2.8.13
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:d6a0ac9b1eb64e85
redis_mode:standalone
os:Linux 2.6.39-400.17.1.el6uek.x86_64 x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.4.7
process_id:17826
run_id:05511bb9216d71d1c4f90610281acdf21abfe27b
tcp_port:6379
uptime_in_seconds:260
```

uptime_in_days:0

hz:10

lru_clock:74522

config_file:/etc/redis/6379.conf

Clients

connected_clients:1

client_longest_output_list:0

client_biggest_input_buf:0

blocked_clients:0

Memory

used_memory:810040

used_memory_human:791.05K

used_memory_rss:7565312

used_memory_peak:810040

used_memory_peak_human:791.05K

used_memory_lua:33792

mem_fragmentation_ratio:9.34

mem_allocator:jemalloc-3.6.0

Persistence

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

loading:0

rdb_changes_since_last_save:0

rdb_bgsave_in_progress:0

rdb_last_save_time:1409360406

rdb_last_bgsave_status:ok

rdb_last_bgsave_time_sec:-1

rdb_current_bgsave_time_sec:-1

aof_enabled:0

aof_rewrite_in_progress:0

aof_rewrite_scheduled:0

aof_last_rewrite_time_sec:-1

aof_current_rewrite_time_sec:-1

aof_last_bgrewrite_status:ok

aof_last_write_status:ok

Stats

total_connections_received:251

total_commands_processed:1

instantaneous_ops_per_sec:0

rejected_connections:0

sync_full:0

sync_partial_ok:0

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
sync_partial_err:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

# CPU
used_cpu_sys:0.52
used_cpu_user:0.23
used_cpu_sys_children:0.00
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

used_cpu_user_children:0.00

Keyspace

db0:keys=4,expires=0,avg_ttl=0

[root@wyzc utils]# /usr/local/bin/redis-cli -h localhost -p 6378 info

Server

redis_version:2.8.13

redis_git_sha1:00000000

redis_git_dirty:0

redis_build_id:d6a0ac9b1eb64e85

redis_mode:standalone

os:Linux 2.6.39-400.17.1.el6uek.x86_64 x86_64

arch_bits:64

multiplexing_api:epoll

gcc_version:4.4.7

process_id:17734

run_id:65f7215bd1ab622f981b5222b25e3e179c88fd14

tcp_port:6378

uptime_in_seconds:517

uptime_in_days:0

hz:10

lru_clock:74530

```
config_file:/etc/redis/6378.conf

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:811088
used_memory_human:792.08K
used_memory_rss:7852032
used_memory_peak:811088
used_memory_peak_human:792.08K
used_memory_lua:33792
mem_fragmentation_ratio:9.68
mem_allocator:jemalloc-3.6.0

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
```

```
rdb_last_save_time:1409360157
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok
aof_last_write_status:ok

# Stats
total_connections_received:2
total_commands_processed:1
instantaneous_ops_per_sec:0
rejected_connections:0
sync_full:0
sync_partial_ok:0
sync_partial_err:0
expired_keys:0
evicted_keys:0
```

keyspace_hits:0

keyspace_misses:0

pubsub_channels:0

pubsub_patterns:0

latest_fork_usec:0

Replication

role:slave

master_host:localhost

master_port:6379

master_link_status:down

master_last_io_seconds_ago:-1

master_sync_in_progress:0

slave_repl_offset:1

master_link_down_since_seconds:1409360674

slave_priority:100

slave_read_only:1

connected_slaves:0

master_repl_offset:0

repl_backlog_active:0

repl_backlog_size:1048576

repl_backlog_first_byte_offset:0

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
repl_backlog_histlen:0


# CPU

used_cpu_sys:2.36

used_cpu_user:0.44

used_cpu_sys_children:0.00

used_cpu_user_children:0.00


# Keyspace

db0:keys=4,expires=0,avg_ttl=0

[root@wyzc utils]#
```

可以看到从无法同步的。

```
[root@wyzc utils]# grep masterauth /etc/redis/6378.conf

# masterauth <master-password>

masterauth redis

[root@wyzc utils]# /etc/init.d/redis_6378    stop

Stopping ...

Redis stopped

[root@wyzc utils]# /etc/init.d/redis_6378    start

Starting Redis server...

[root@wyzc utils]# /usr/local/bin/redis-cli    -h localhost    -p 6378 info

# Server
```

redis_version:2.8.13

redis_git_sha1:00000000

redis_git_dirty:0

redis_build_id:d6a0ac9b1eb64e85

redis_mode:standalone

os:Linux 2.6.39-400.17.1.el6uek.x86_64 x86_64

arch_bits:64

multiplexing_api:epoll

gcc_version:4.4.7

process_id:17944

run_id:5bd399dbe11fa143fa6b0a99c6c098e320678641

tcp_port:6378

uptime_in_seconds:17

uptime_in_days:0

hz:10

lru_clock:74646

config_file:/etc/redis/6378.conf

Clients

connected_clients:2

client_longest_output_list:0

client_biggest_input_buf:0

```
blocked_clients:0

# Memory
used_memory:831968
used_memory_human:812.47K
used_memory_rss:7806976
used_memory_peak:831968
used_memory_peak_human:812.47K
used_memory_lua:33792
mem_fragmentation_ratio:9.38
mem_allocator:jemalloc-3.6.0

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1409360773
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
```

aof_rewrite_scheduled:0

aof_last_rewrite_time_sec:-1

aof_current_rewrite_time_sec:-1

aof_last_bgrewrite_status:ok

aof_last_write_status:ok

Stats

total_connections_received:1

total_commands_processed:1

instantaneous_ops_per_sec:0

rejected_connections:0

sync_full:0

sync_partial_ok:0

sync_partial_err:0

expired_keys:0

evicted_keys:0

keyspace_hits:0

keyspace_misses:0

pubsub_channels:0

pubsub_patterns:0

latest_fork_usec:0

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
# Replication
role:slave
master_host:localhost
master_port:6379
master_link_status:up
master_last_io_seconds_ago:10
master_sync_in_progress:0
slave_repl_offset:15
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

# CPU
used_cpu_sys:0.04
used_cpu_user:0.00
used_cpu_sys_children:0.00
used_cpu_user_children:0.00
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

Keyspace

db0:keys=4,expires=0,avg_ttl=0

[root@wyzc utils]#

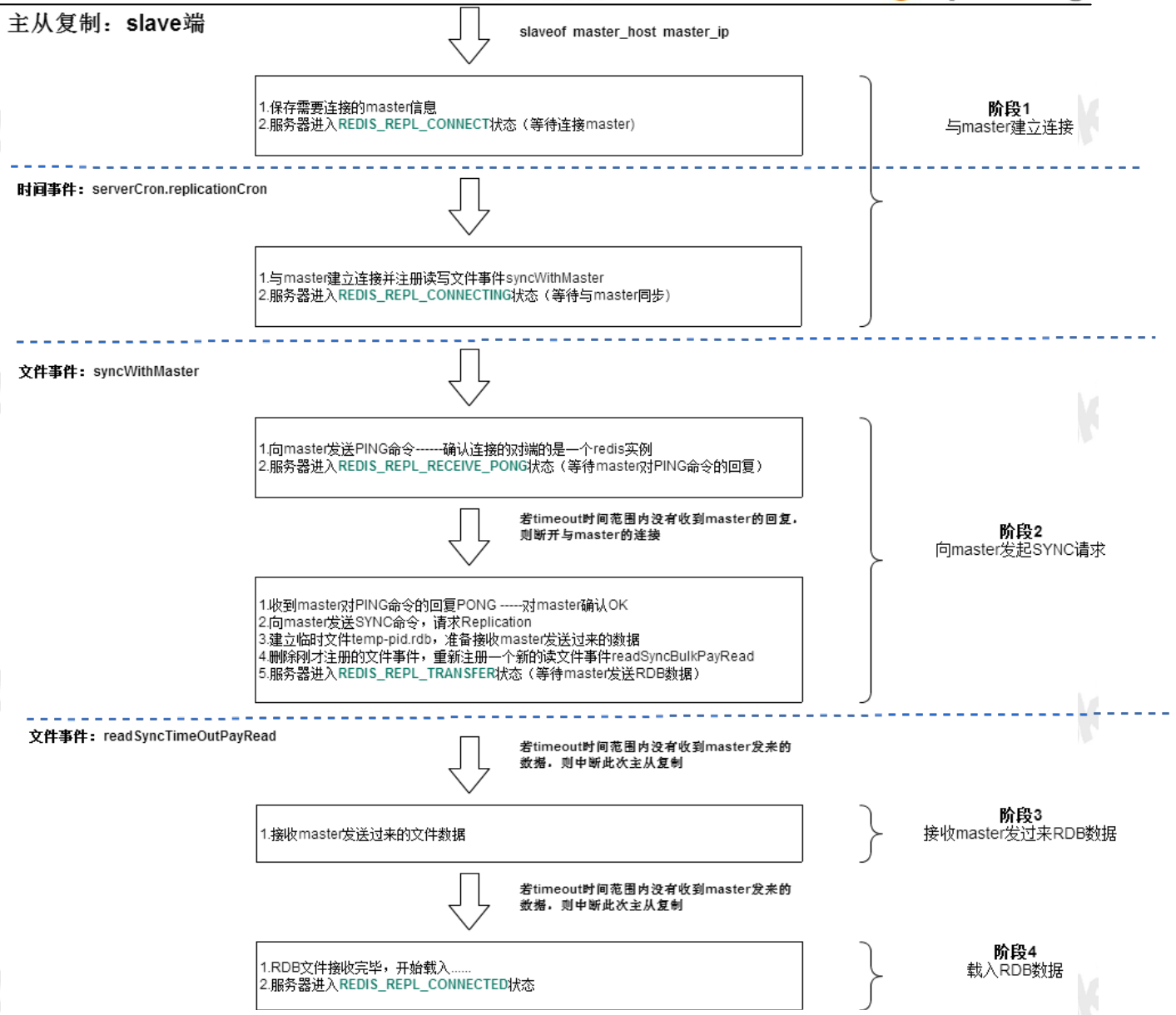
给从设置密码验证后可以看到能复制了。主从状态正常了。

主从复制简单来说就是把一台 redis 数据库中的数据同步到另一台 redis 数据库，并且按照数据流向，数据的发送者我们称作 master，数据的接受者我们称作 slave（master/slave 的划分并不是那么一定的，譬如 B 可以作为 A 的 slave，但同时也可以作为 C 的 master），下面就从 slave 和 master 的角度分别说明主从复制流程。

首先是 slave 端，对于 slave 端来说，主从复制主要经历四个阶段：

第一阶段：与 master 建立连接
第二阶段：向 master 发起同步请求（SYNC）
第三阶段：接受 master 发来的 RDB 数据
第四阶段：载入 RDB 文件

下面我们就通过一个图来概述在每一个阶段中，slave 究竟做了些什么：

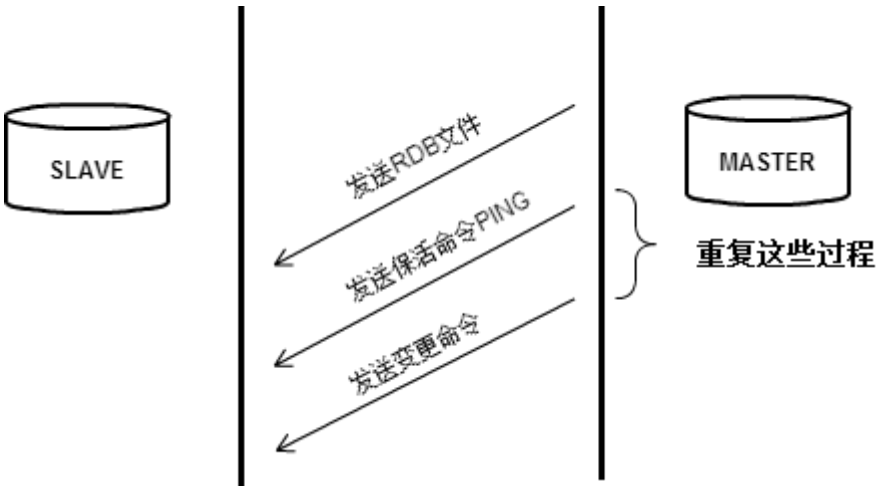


关于上图，有一点说明下：redis 接收到 slaveof master_host master_port 命令后并没有马上与 master 建立连接，而是当执行服务器例行任务 serverCron，发现自己正处于 REDIS_REPL_CONNECT 状态，这时才真正的向 maser 发起连接，伪代码：

Python 代码

```
1. def serverCron():
2.     # 服务器处于 REDIS_REPL_CONNECT 状态
3.     if redisServer.repl_state == REDIS_REPL_CONNECT:
4.         # 向 master 发起连接
5.         connectWithMaster()
6.     # 其他例行任务（省略）...
```

接着我们来看下主从复制过程中，master 这边的流程是如何，在具体看细节之前，我们先综合来看 master 这边主要做的几件事情：



看完这个图，你也许会有以下几个疑问：

1. 为什么在 master 发送完 RDB 文件后，还要定期的向 slave 发送 PING 命令？
2. 在发送完 RDB 文件之后，master 发送的“变更”命令又是什么，有什么用？

在回答问题之前 1，我们先回答问题 2：

master 保存 RDB 文件是通过一个子进程进行的，所以 master 依然可以处理客户端请求而不被阻塞，但这也导致了在保存 RDB 文件期间，“键空间”可能发生变化（譬如接收到一个客户端请求，执行"set name diaocow"命令），因此为了保证数据同步的一致性，master 会在保存 RDB 文件期间，把接受到的这些可能变更数据库“键空间”的命令保存下来，然后放到每个 slave 的回复列表中，当 RDB 文件发送完 master 会发送这些回复列表中的内容，并且在这之后，如果数据库发生变更，master 依然会把变更的命令追加到回复列表发送给 slave，这样就可以保证 master 和 slave 数据的一致性！相关伪代码：

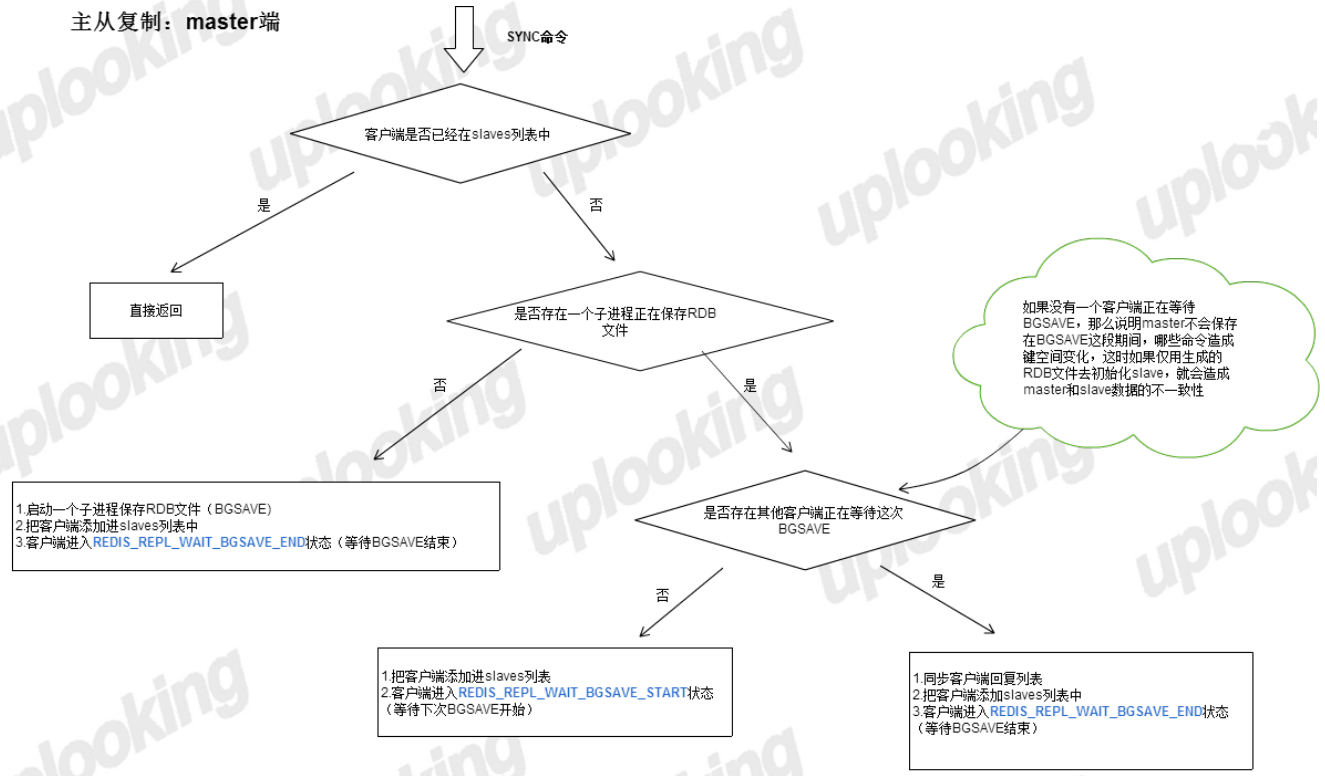
Python 代码

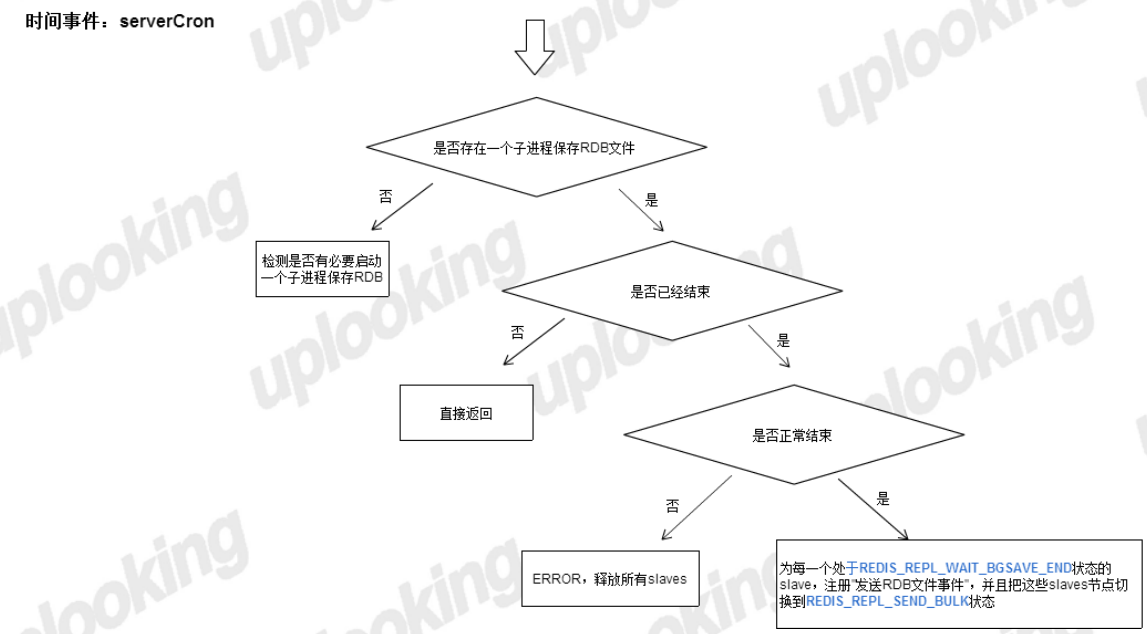
```
1.def processCommand(cmd, argc, argv):
2.    # 处理命令
3.    call(cmd, argc, argv)
4.    # 如果该命令造成数据库键空间变化 and 当前 redis 是一个 master，则同步变更命令
5.    if redisServer.update_key_space and len(redisServer.slaves) > 0:
6.        replicationFeedSlaves(cmd, argc, argv)
```

```
7.
8.def replicationFeedSlaves(cmd, argc, argv):
9.    # 把变更命令发送给每一个处于：REDIS_REPL_WAIT_BGSAVE_END 状态的 slave 节点
10.    for slave in redisServer.slaves:
11.        if slave.replstate == REDIS_REPL_WAIT_BGSAVE_START:
12.            continue
13.        slave.updateNotify(cmd, argc, argv)
```

由于在发送完 RDB 文件之后，master 会不定时的给 slave 发送“变更”命令，可能过 1s，也可能过 1 小时，所以为了防止 slave 无意义等待（譬如 master 已经挂掉的情况），master 需要定时发送“保活”命令 PING，以此告诉 slave：我还活着，不要中断与我的连接

现在我们就看下，当 master 接受到 slave 发送的 sync 同步命令后究竟发生了哪些事：

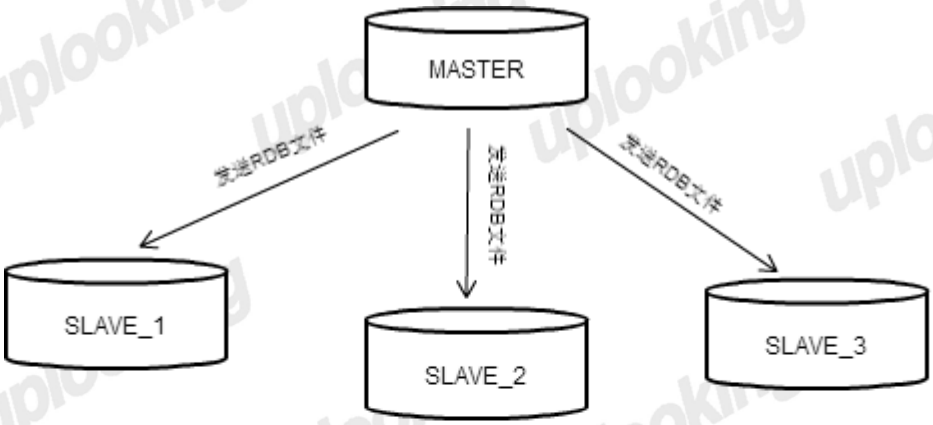




上图看似分支复杂，但我们抓住以下几点即可：

1.保存 RDB 文件是在一个子进程中进行的；
2.如果 master 已经在保存 RDB 文件，但是没有客户端正在等待这次 BGSAVE，新添加的 slave 需要等到下次 BGSAVE，而不能直接使用这次生成的 RDB 文件（原因图中已经说明）
3.master 会定期检查 RDB 文件是否保存完毕（时间事件 serverCron）；

接下来我们看下，master 是如何给每一个 slave 发送 RDB 文件的：

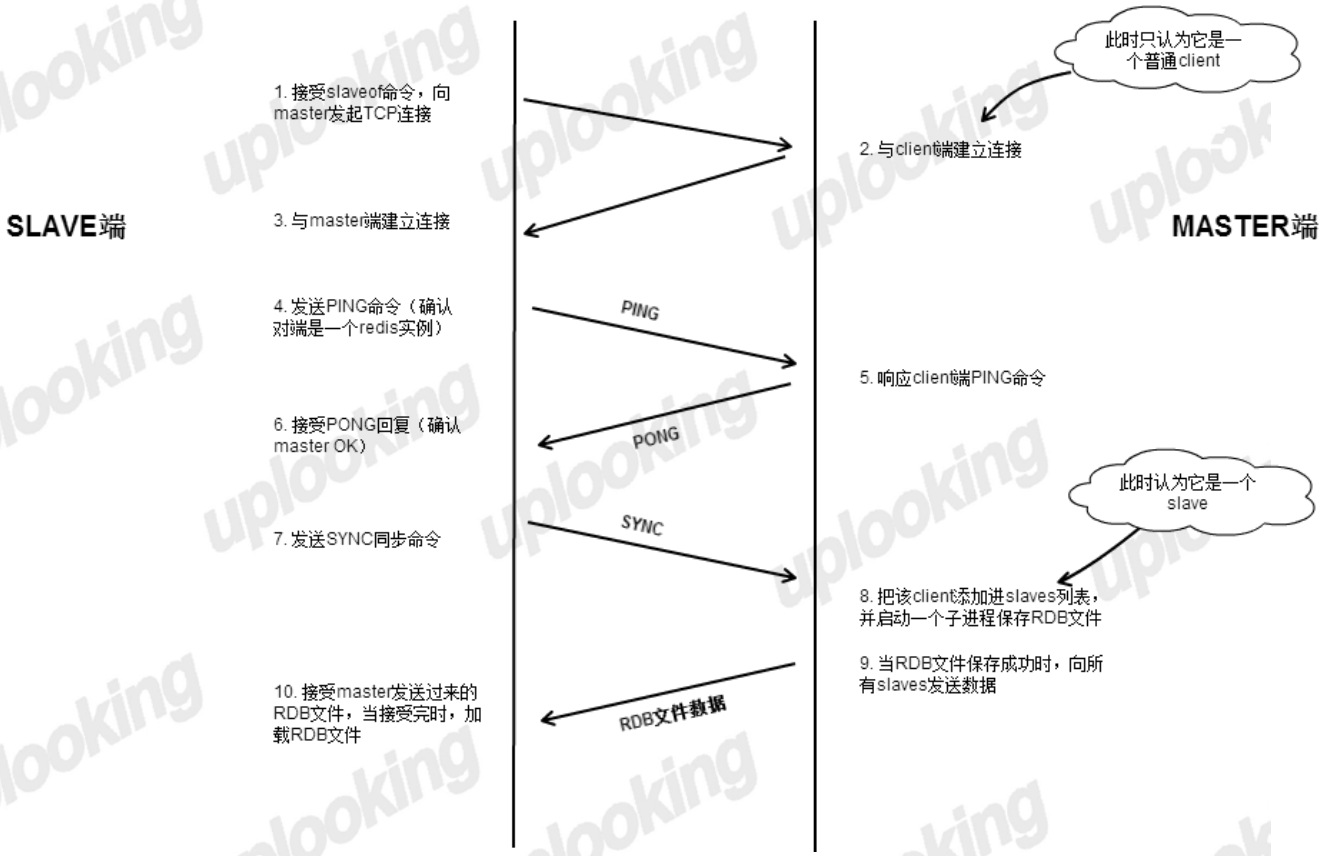


1. master会为每个slave维护下列信息：
a. repldboff： 当前发送RDB文件的偏移
b. repldbsize：需要发送的RDB文件大小
c. replstate：当前slave状态（等待发送数据？正在发送数据？已经发送完数据？）

2. RDB文件分两部分发送：
a. 文件头（用来告诉slave客户端，你应该接受多少数据）
b. 文件体（RDB数据）

3. 当向某个客户端发送完RDB文件后，该客户端状态从REDIS_REPL_SEND_BULK状态变为REDIS_REPL_ONLINE

好了，至此我们已经分析完在主从复制过程中，master 和 slave 两边分别是怎么一个处理流程;最后，我绘制了一个图，综述了主从复制这一过程（我们可以边看图，边回忆其中的具体细节）：



PS：在主从复制过程中，任何一步发生错误，都会导致整个过程重头开始，所以若 RDB 文件很大又或是此时正处在业务高峰期，对系统性能将会有非常大的影响！

总结：

1. 了解主从复制 master 和 slave 的概念；
2. 了解主从复制执行过程，特别是其中关键的几步；
3. 了解目前主从复制过程中尚存的不足之处；

数据持久性及备份恢复技术

Redis 持久化

Redis 提供了不同级别的持久化方式:

- RDB 持久化方式能够在指定的时间间隔能对你的数据进行快照存储.
- AOF 持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF 命令以 redis 协议追加保存每次写的操作到文件末尾.Redis 还能对 AOF 文件进行后台重写,使得 AOF 文件的体积不至于过大.
- 如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.

- 你也可以同时开启两种持久化方式，在这种情况下，当 redis 重启的时候会优先载入 AOF 文件来恢复原始的数据,因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整.

最重要的事情是了解 RDB 和 AOF 持久化方式的不同,让我们以 RDB 持久化方式开始:

RDB 的优点

- RDB 是一个非常紧凑的文件,它保存了某个时间点得数据集,非常适用于数据集的备份,比如你可以在每个小时报保存一下过去 24 小时内的数据,同时每天保存过去 30 天的数据,这样即使出了问题你也可以根据需求恢复到不同版本的数据集.
- RDB 是一个紧凑的单一文件,很方便传送到另一个远端数据中心或者亚马逊的 S3（可能加密），非常适用于灾难恢复.
- RDB 在保存 RDB 文件时父进程唯一需要做的就是 fork 出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他 IO 操作，所以 RDB 持久化方式可以最大化 redis 的性能.
- 与 AOF 相比,在恢复大的数据集的时候，RDB 方式会更快一些.

RDB 的缺点

- 如果你希望在 redis 意外停止工作（例如电源中断）的情况下丢失的数据最少的话，那么 RDB 不适合你.虽然你可以配置不同的 save 时间点(例如每隔 5 分钟并且对数据集有 100 个写的操作),是 Redis 要完整的保存整个数据集是一个比较繁重的工作,你通常会每隔 5 分钟或者更久做一次完整的保存,万一在 Redis 意外宕机,你可能会丢失几分钟的数据.
- RDB 需要经常 fork 子进程来保存数据集到硬盘上,当数据集比较大的时候,fork 的过程是非常耗时的,可能会导致 Redis 在一些毫秒级内不能响应客户端的请求.如果数据集巨大并且 CPU 性能不是很好的情况下,这种情况会持续 1 秒,AOF 也需要 fork,但是你可以调节重写日志文件的频率来提高数据集的耐久度.

AOF 优点

- 使用 AOF 会让你的 Redis 更加耐久: 你可以使用不同的 fsync 策略：无 fsync,每秒 fsync,每次写的时候 fsync.使用默认的每秒 fsync 策略,Redis 的性能依然很好(fsync 是由后台线程进行处理的,主线程会尽力处理客户端请求),一旦出现故障，你最多丢失 1 秒的数据.
- AOF 文件是一个只进行追加的日志文件,所以不需要写入 seek,即使由于某些原因(磁盘空间已满，写的过程中宕机等等)未执行完整的写入命令,你也也可使用 redis-check-aof 工具修复这些问题.
- 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 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）。

<=”” bug=”” 几乎是不可能出现这种=”” rdb=”” 但是对比来说，=”” 文件中并不常见，=”” aof=”” 在=”” 虽然这种=”” 并通过重新载入这些数据来确保一切正常。=”” 它们会自动生成随机的、复杂的数据集，=”” 测试套件里为这种情况添加了测试：=”” 。）=”” 就曾经引起过这样的=”” brpoplpush=”” （举个例子，阻塞命令=”” 文件在重新载入时，无法将数据集恢复成保存时的原样。=”” 因为个别命令的原因，导致=”” ：=”” 在过去曾经发生过这样的=”">

如何选择使用哪种持久化方式？

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 。

Note: 因为以上提到的种种原因， 未来我们可能会将 AOF 和 RDB 整合成单个持久化模型。 （这是一个长期计划。）

接下来的几个小节将介绍 RDB 和 AOF 的更多细节。

快照

在默认情况下，Redis 将数据库快照保存在名字为 `dump.rdb` 的二进制文件中。你可以对 Redis 进行设置，让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。你也可以通过调用 SAVE 或者 BGSAVE ， 手动让 Redis 进行数据集保存操作。

比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集:

```
save 60 1000
```

这种持久化方式被称为快照 *snapshotting*.

工作方式

当 Redis 需要保存 dump.rdb 文件时， 服务器执行以下操作:

- Redis 调用 forks. 同时拥有父进程和子进程。
- 子进程将数据集写入到一个临时 RDB 文件中。

- 当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件。

这种工作方式使得 Redis 可以从写时复制（copy-on-write）机制中获益。

只追加操作的文件（Append-only file，AOF）

快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。

从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化。

你可以在配置文件中打开 AOF 方式:

```
appendonly yes
```

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 SET）， 这个命令就会被追加到 AOF 文件的末尾。这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

日志重写

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。

为了处理这种情况， Redis 支持一种有趣的特性： 可以在不打断服务客户端的情况下， 对 AOF 文件进行重建（rebuild）。执行 BGREWRITEAOF 命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。Redis 2.2 需要自己手动执行 BGREWRITEAOF 命令； Redis 2.4 则可以自动触发 AOF 重写， 具体信息请查看 2.4 的示例配置文件。

AOF 有多耐用?

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。有三种方式：

- 每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全
- 每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。
- 从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

如果 AOF 文件损坏了怎么办?

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么

Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：

- 为现有的 AOF 文件创建一个备份。
- 使用 Redis 附带的 redis-check-aof 程序，对原来的 AOF 文件进行修复:

```
$ redis-check-aof --fix <filename>
```

- （可选）使用 diff -u 对比修复后的 AOF 文件和原始 AOF 文件的备份，查看两个文件之间的不同之处。
- 重启 Redis 服务器，等待服务器载入修复后的 AOF 文件，并进行数据恢复。

工作原理

AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制:

- Redis 执行 fork() ，现在同时拥有父进程和子进程。
- 子进程开始将新 AOF 文件的内容写入到临时文件。
- 对于所有新执行的写入命令，父进程一边将它们累积到一个内存缓存中，一边将这些改动追加到现有 AOF 文件的末尾,这样样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。
- 当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。
- 搞定！现在 Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

怎样从 RDB 方式切换为 AOF 方式

在 Redis 2.2 或以上版本，可以在不重启的情况下，从 RDB 切换到 AOF ：

- 为最新的 dump.rdb 文件创建一个备份。
- 将备份放到一个安全的地方。
- 执行以下两条命令:
- redis-cli config set appendonly yes
- redis-cli config set save ""

<li 确保命令执行之后，数据库的键的数量没有改变。< li="">

- 确保写命令会被正确地追加到 AOF 文件的末尾。

</li 确保命令执行之后，数据库的键的数量没有改变。<>

执行的第一条命令开启了 AOF 功能： Redis 会阻塞直到初始 AOF 文件创建完成为止， 之后 Redis 会继续处理命令请求， 并开始将写入命令追加到 AOF 文件末尾。

执行的第二条命令用于关闭 RDB 功能。这一步是可选的， 如果你愿意的话， 也可以同时使用 RDB 和 AOF 这两种持久化功能。

重要:别忘了在 redis.conf 中打开 AOF 功能！ 否则的话， 服务器重启之后， 之前通过 CONFIG SET 设置的配置就会被遗忘， 程序会按原来的配置来启动服务器。

AOF 和 RDB 之间的相互作用

在版本号大于等于 2.4 的 Redis 中， BGSAVE 执行的过程中， 不可以执行 BGREWRITEAOF 。 反过来说， 在 BGREWRITEAOF 执行的过程中， 也不可以执行 BGSAVE。这可以防止两个 Redis 后台进程同时对磁盘进行大量的 I/O 操作。

如果 BGSAVE 正在执行，并且用户显示地调用 BGREWRITEAOF 命令，那么服务器将向用户回复一个 OK 状态， 并告知用户， BGREWRITEAOF 已经被预定执行： 一旦 BGSAVE 执行完毕， BGREWRITEAOF 就会正式开始。

当 Redis 启动时， 如果 RDB 持久化和 AOF 持久化都被打开了， 那么程序会优先使用 AOF 文件来恢复数据集， 因为 AOF 文件所保存的数据通常是最完整的。

备份 redis 数据

在阅读这个小节前， 请牢记下面这句话: **确保你的数据由完整的备份**. 磁盘故障， 节点失效， 诸如此类的问题都可能让你的数据消失不见， 不进行备份是非常危险的。

Redis 对于数据备份是非常友好的， 因为你可以在服务器运行的时候对 RDB 文件进行复制： RDB 文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的 RDB 文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用 rename(2) 原子地用临时文件替换原来的 RDB 文件。

这也就是说， 无论何时， 复制 RDB 文件都是绝对安全的。

- 创建一个定期任务（cron job）， 每小时将一个 RDB 文件备份到一个文件夹， 并且每天将一个 RDB 文件备份到另一个文件夹。
- 确保快照的备份都带有相应的日期和时间信息， 每次执行定期任务脚本时， 使用 find 命令来删除过期的快照： 比如说， 你可以保留最近 48 小时内的每小时快照， 还可以保留最近一两个月的每日快照。
- 至少每天一次， 将 RDB 备份到你的数据中心之外， 或者至少是备份到你运行 Redis 服务器的物理机器之外。

容灾备份

Redis 的容灾备份基本上就是对数据进行备份， 并将这些备份传送到多个不同的外部数据中心。容灾备份可以在 Redis 运行并产生快照的主数据中心发生严重的问题时， 仍然让数据处于安全状态。

因为很多 Redis 用户都是创业者， 他们没有大把大把的钱可以浪费， 所以下面介绍的都是一些实用又便宜的容债备份方法：

- Amazon S3 ，以及其他类似 S3 的服务，是一个构建灾难备份系统的好地方。 最简单的方法就是将你的每小时或者每日 RDB 备份加密并传送到 S3 。 对数据的加密可以通过 gpg -c 命令来完成（对称加密模式）。 记得把你的密码放到几个不同的、安全的地方去（比如你可以把密码复制给你组织里最重要的人物）。 同时使用多个储存服务来保存数据文件，可以提升数据的安全性。

- 传送快照可以使用 SCP 来完成（SSH 的组件）。 以下是简单并且安全的传送方法： 买一个离你的数据中心非常远的 VPS ， 装上 SSH ， 创建一个无口令的 SSH 客户端 key ， 并将这个 key 添加到 VPS 的 authorized_keys 文件中， 这样就可以向这个 VPS 传送快照备份文件了。 为了达到最好的数据安全性，至少要从两个不同的提供商那里各购买一个 VPS 来进行数据容灾备份。

需要注意的是， 这类容灾系统如果没有小心地进行处理的话， 是很容易失效的。最低限度下， 你应该在文件传送完毕之后， 检查所传送备份文件的体积和原始快照文件的体积是否相同。 如果你使用的是 VPS ， 那么还可以通过比对文件的 SHA1 校验和来确认文件是否传送完整。

另外， 你还需要一个独立的警报系统， 让它在负责传送备份文件的传送器（transfer）失灵时通知你。

高可用性架构与部署

引言

大概是因为 Redis 是个人开发的产品，所以 Redis 的高可用方案是被分成了几块来实现：主从复制、主从切换以及虚拟 IP 或客户端方案。

从 Redis 2.8 开始加入对 Sentinel 机制从而实现了服务器端的主从切换，但目前尚未发现实现虚拟 IP 或客户端切换方案。

主从复制研究

```
wget http://download.redis.io/releases/redis-2.8.2.tar.gz
tar xzf redis-2.8.2.tar.gz

mv redis-2.8.2 /opt/
cp redis.conf redis-master.conf
cp redis.conf redis-slave.conf

cd /opt/redis-2.8.2
make
```

以下是关于 Redis 复制功能的几个重要方面：
1. 一个 Master 可以有多个 Slave；
2. Redis 使用异步复制。从 2.8 开始，Slave 会周期性（每秒一次）发起一个 Ack 确认复制流（replication stream）被处理进度；
3. 不仅主服务器可以有从服务器， 从服务器也可以有自己的从服务器， 多个从服务器之间可以构成一个图状结构；
4. 复制在 Master 端是非阻塞模式的，这意味着即便是多个 Slave 执行首次同步时，Master 依然可以提供查询服务；
5. 复制在 Slave 端也是非阻塞模式的：如果你在 redis.conf 做了设置，Slave 在执行首次同步的时候仍可以使用旧数据集提供查询；你也可以配置为当 Master 与 Slave 失去联系时，让 Slave 返回客户端一个错误提示；
6. 当 Slave 要删掉旧的数据集，并重新加载新版数据时，Slave 会阻塞连接请求（一般发生在与 Master 断开

重连后的恢复阶段）；
7. 复制功能可以单纯地用于数据冗余（data redundancy），也可以通过让多个从服务器处理只读命令请求来提升扩展性（scalability）： 比如说， 繁重的 SORT 命令可以交给附属节点去运行。
8. 可以通过修改 Master 端的 redis.config 来避免在 Master 端执行持久化操作（Save），由 Slave 端来执行持久化。

分别修改 redis-master.conf 和 redis-slave.conf,
daemonize 项，改为 yes(缺省为 no)：daemonize yes
maxmemory 项，设最大占用内存为 50MB：maxmemory 50mb
而有 6 种内存过期策略，通过 maxmemory-policy 修改，一般使用默认值或 allkeys-lru：
volatile-lru：只对设置了过期时间的 key 进行 LRU（默认值）
allkeys-lru ： 是从所有 key 里 删除 不经常使用的 key
volatile-random：随机删除即将过期 key
allkeys-random：随机删除
volatile-ttl ： 删除即将过期的
noeviction ： 永不过期，返回错误

修改 redis-slave.conf 中的端口，避免和 Master 的相同：port 7379

Redis 复制工作原理：
1. 如果设置了一个 Slave，无论是第一次连接还是重连到 Master，它都会发出一个 SYNC 命令；
2. 当 Master 收到 SYNC 命令之后，会做两件事：
a) Master 执行 BGSAVE，即在后台保存数据到磁盘（rdb 快照文件）；
b) Master 同时将新收到的写入和修改数据集的命令存入缓冲区（非查询类）；
3. 当 Master 在后台把数据保存到快照文件完成之后，Master 会把这个快照文件传送给 Slave，而 Slave 则把内存清空后，加载该文件到内存中；
4. 而 Master 也会把此前收集到缓冲区中的命令，通过 Reids 命令协议形式转发给 Slave，Slave 执行这些命令，实现和 Master 的同步；
5. Master/Slave 此后会不断通过异步方式进行命令的同步，达到最终数据的同步一致；
6. 需要注意的是 Master 和 Slave 之间一旦发生重连都会引发全量同步操作。但在 2.8 之后版本，也可能是部分同步操作。

部分复制
2.8 开始，当 Master 和 Slave 之间的连接断开之后，他们之间可以采用持续复制处理方式代替采用全量同步。Master 端为复制流维护一个内存缓冲区（in-memory backlog），记录最近发送的复制流命令；同时，Master 和 Slave 之间都维护一个复制偏移量(replication offset)和当前 Master 服务器 ID（Master run id）。当网络断开，Slave 尝试重连时：
a. 如果 MasterID 相同（即仍是断网前的 Master 服务器），并且从断开时到当前时刻的历史命令依然在 Master 的内存缓冲区中存在，则 Master 会将缺失的这段时间的所有命令发送给 Slave 执行，然后复制工作就可以继续执行了；
b. 否则，依然需要全量复制操作；

Redis 2.8 的这个部分重同步特性会用到一个新增的 PSYNC 内部命令， 而 Redis 2.8 以前的旧版本只有 SYNC 命令， 不过， 只要从服务器是 Redis 2.8 或以上的版本， 它就会根据主服务器的版本来决定到底是使用 PSYNC 还是 SYNC ：

如果主服务器是 Redis 2.8 或以上版本，那么从服务器使用 PSYNC 命令来进行同步。
如果主服务器是 Redis 2.8 之前的版本，那么从服务器使用 SYNC 命令来进行同步。

配置 Slave

只需要将 redis-slave.conf 中 REPLICATION 段中的 slaveof <masterip> <masterport>行的注释去掉，并修改为：

slaveof 127.0.0.1 6379

即完成该 Slave 的配置，并指向本地端口为 6379 的 Master 端。

masterauth

如果 Master 端通过 requirepass 设置了密码，Slave 需要对应的通过 masterauth <password>设置密码；

slave-serve-stale-data

当 Slave 和 Master 断开连接时，Slave 是直接返回错误提示还是利用历史数据响应客户端（或是直接返回空数据，当全量复制进行时）。yes 是缺省值，即利用历史数据响应。

slave-read-only

缺省模式下，Slave 服务器是只读的。

repl-ping-slave-peroid

即心跳检测间隔时间，缺省值为 10 秒。

repl-timeout 60

复制超时

启动 Redis

#redis-server redis-master.conf

#redis-server redis-slave.conf

使用 ps 查看进程

#ps -ef | grep redis

还可以用 netstat 查看端口

#netstat -tpln

验证主从复制

#redis-cli set test 1000

#redis-cli get test #只能说明目前 Master 工作正常，不能说明 Slave 已经复制数据

根据刚刚查看到的端口号，把端口号是 6379 的 Master 进程杀掉

#kill -9 XXXX(PID 号)

#redis-cli -p 7379 get test #连接到 Slave 上，读取 test

#redis-cli -p 7379 set test 1000 #会提示 Slave 只读错误，不能写入

Redis 2.8 版开始正式提供名为 Sentinel 的主从切换方案，Sentinel 用于管理多个 Redis 服务器实例，主要负责三个方面的任务：

1. **监控（Monitoring）**： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
2. **提醒（Notification）**： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
3. **自动故障迁移（Automatic failover）**： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

Redis Sentinel 是一个分布式系统， 你可以在一个架构中运行多个 Sentinel 进程（progress）， 这些进程使用流言协议（gossip protocols)来接收关于主服务器是否下线的信息， 并使用投票协议（agreement protocols）来决定是否执行自动故障迁移， 以及选择哪个从服务器作为新的主服务器。

启动 Sentinel

使用--sentinel 参数启动，并必须指定一个对应的配置文件，系统会使用配置文件来保存 Sentinel 的当前状态， 并在 Sentinel 重启时通过载入配置文件来进行状态还原。

```
redis-server /path/to/sentinel.conf --sentinel
```

使用 TCP 端口 26379，可以使用 redis-cli 或其他任何客户端与其通讯。

如果启动 Sentinel 时没有指定相应的配置文件， 或者指定的配置文件不可写（not writable）， 那么 Sentinel 会拒绝启动。

配置 Sentinel

以下是一段配置文件的示例：

```
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 60000
sentinel failover-timeout mymaster 180000
sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4
sentinel down-after-milliseconds resque 10000
sentinel failover-timeout resque 180000
sentinel parallel-syncs resque 5
```

第一行配置指示 Sentinel 去监视一个名为 mymaster 的主服务器， 这个主服务器的 IP 地址为 127.0.0.1 ， 端口号为 6379 ， 而将这个主服务器判断为失效至少需要 2 个 Sentinel 同意 （只要同意 Sentinel 的数量不达标，自动故障迁移就不会执行）。

不过需要注意的是，无论你设置要多少个 Sentinel 同意才能判断一个服务器失效，一个 Sentinel 都需要获得系统中多数（majority） Sentinel 的支持，才能发起一次自动故障迁移，并预留一个给定的配置纪元（Configuration Epoch ，一个配置纪元就是一个新主服务器配置的版本号）。也就是说，如果只有少数（minority）Sentinel 进程正常运作的情况下，是不能执行自动故障迁移的。

down-after-milliseconds 选项指定了 Sentinel 认为服务器已经断线所需的毫秒数（判定为主观下线 SDOWN）。

parallel-syncs 选项指定了在执行故障转移时， 最多可以有多少个从服务器同时对新的主服务器进行同步， 这个数字越小， 完成故障转移所需的时间就越长，但越大就意味着越多的从服务器因为复制而不可用。可以通过将这个值设为 1 来保证每次只有一个从服务器处于不能处理命令请求的状态。

主观下线和客观下线

1. **主观下线（Subjectively Down， 简称 SDOWN）**指的是单个 Sentinel 实例对服务器做出的下线判断。

2. **客观下线（Objectively Down， 简称 ODOWN）**指的是多个 Sentinel 实例在对同一个服务器做

出 SDOWN 判断， 并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后， 得出的服务器下线判断。

客观下线条件只适用于主服务器： 对于任何其他类型的 Redis 实例， Sentinel 在将它们判断为下线前不需要进行协商， 所以从服务器或者其他 Sentinel 永远不会达到客观下线条件。
只要一个 Sentinel 发现某个主服务器进入了客观下线状态， 这个 Sentinel 就可能会被其他 Sentinel 推选出， 并对失效的主服务器执行自动故障迁移操作。

每个 Sentinel 实例都执行的定时任务

1. 每个 Sentinel 以每秒钟一次的频率向它所知的主服务器、从服务器以及其他 Sentinel 实例发送一个 PING 命令。

2. 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 那么这个实例会被 Sentinel 标记为主观下线。 一个有效回复可以是： +PONG 、-LOADING 或者 -MASTERDOWN 。

3. 如果一个主服务器被标记为主观下线， 那么正在监视这个主服务器的所有 Sentinel 要以每秒一次的频率确认主服务器的确进入了主观下线状态。

4. 如果一个主服务器被标记为主观下线， 并且有足够数量的 Sentinel （至少要达到配置文件指定的数量）在指定的时间范围内同意这一判断， 那么这个主服务器被标记为客观下线。

5. 在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令。 当一个主服务器被 Sentinel 标记为客观下线时， Sentinel 向下线主服务器的所有从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。

6. 当没有足够数量的 Sentinel 同意主服务器已经下线， 主服务器的客观下线状态就会被移除。 当主服务器重新向 Sentinel 的 PING 命令返回有效回复时， 主服务器的主管下线状态就会被移除。

Sentinel API

有两种方式可以与 Sentinel 进行通讯：指令、发布与订阅。

Sentinel 命令

PING ：返回 PONG 。

SENTINEL masters ：列出所有被监视的主服务器，以及这些主服务器的当前状态；

SENTINEL slaves <master name> ：列出给定主服务器的所有从服务器，以及这些从服务器的当前状态；

SENTINEL get-master-addr-by-name <master name> ： 返回给定名字的主服务器的 IP 地址和端口号。 如果这个主服务器正在执行故障转移操作， 或者针对这个主服务器的故障转移操作已经完成， 那么这个 命令返回新的主服务器的 IP 地址和端口号；

SENTINEL reset <pattern> ： 重置所有名字和给定模式 pattern 相匹配的主服务器。 pattern 参数是一个 Glob 风格的模式。 重置操作清楚主服务器目前的所有状态， 包括正在执行中的故障转移， 并移除目前已经发现和关联的， 主服务器的所有从服务器和 Sentinel ；

SENTINEL failover <master name> ： 当主服务器失效时， 在不询问其他 Sentinel 意见的情况下， 强制开始一次自动故障迁移。

客户端可以通过 SENTINEL get-master-addr-by-name <master name>获取当前的主服务器 IP 地址和端口号，以及 SENTINEL slaves <master name>获取所有的 Slaves 信息

发布与订阅信息

客户端可以将 Sentinel 看作是一个只提供了订阅功能的 Redis 服务器：你不可以使用 PUBLISH 命令向这个服务器发送信息，但你可以用 SUBSCRIBE 命令或者 PSUBSCRIBE 命令，通过订阅给定的频道来获取相应的事件提醒。

一个频道能够接收和这个频道的名字相同的事件。比如说，名为 +sdown 的频道就可以接收所有实例进入主观下线（SDOWN）状态的事件。

通过执行 PSUBSCRIBE * 命令可以接收所有事件信息。

+switch-master <master name> <oldip> <oldport> <newip> <newport> ：配置变更，主服务器的 IP 和地址已经改变。这是绝大多数外部用户都关心的信息。

可以看出，我们使用 Sentinel 命令和发布订阅两种机制就能很好的实现和客户端的集成整合：

使用 get-master-addr-by-name 和 slaves 指令可以获取当前的 Master 和 Slaves 的地址和信息；而当发生故障转移时，即 Master 发生切换，可以通过订阅的+switch-master 事件获得最新的 Master 信息。

*PS:更多 Sentinel 的可订阅事件参见官方文档。

sentinel.conf 中的 notification-script

在 sentinel.conf 中可以配置多个 sentinel notification-script <master name> <shell script-path>，如 sentinel notification-script mymaster ./check.sh

这个是在群集 `failover` 时会触发执行指定的脚本。脚本的执行结果若为 1，即稍后重试（最大重试次数为 10）；若为 2，则执行结束。并且脚本最大执行时间为 60 秒，超时会被终止执行。

PS：目前会存在该脚本被执行多次的问题，查找资料有人解释是：

脚本分为两个级别，SENTINEL_LEADER 和 SENTINEL_OBSERVER，前者仅由领头 Sentinel 执行（一个 Sentinel），而后者由监视同一个 master 的所有 Sentinel 执行（多个 Sentinel）。

Redis sentinel(哨兵)模块已经被集成在 redis2.4+的版本中,尚管目前不是 release,不过可以尝试去使用和了解,事实上 sentinel 还是有点复杂的.

sentinel 主要功能就是为 Redis M-S(master,slaves)集群提供了 1)master 存活检测 2)集群中 M-S 服务监控 3) 自动故障转移,M-S 角色转换等能力,从一个方面说是提高了 redis 集群的可用性.

一般情况下,最小 M-S 单元各有一个 maste 和 slave 组成,当 master 失效后,sentinel 可以帮助我们自动将 slave 提升为 master;有了 sentinel 组件,可以减少系统管理员的人工切换 slave 的操作过程.

sentinel 的一些设计思路和 zookeeper 非常类似,事实上,你可以不使用 sentinel,而是自己开发一个监控 redis 的 zk 客户端也能够完成相应的设计要求.

一.环境部署

准备 3 个 redis 服务,简单构建一个小的 M-S 环境；它们各自的 redis.conf 配置项中,除了 port 不同外,要求其他的配置完全一样(包括 aof/snap,memory,rename 以及授权密码等);原因是基于 sentinel 做故障转移,所有的 server 运行机制都必须一样,它们只不过在运行时"角色"不同,而且它们的角色可能在故障时会被转换;slave 在某些时刻也会成为 master,尽管在一般情况下,slave 的数据持久方式经常采取 snapshot,而

master 为 aof,不过基于 sentinel 之后,slave 和 master 均要采取 aof(通过 bgsave,手动触发 snapshot 备份).

1) redis.conf:

```
##redis.conf
##redis-0,默认为 master
port 6379
##授权密码，请各个配置保持一致
requirepass 012_345^678-90
masterauth 012_345^678-90
##暂且禁用指令重命名
##rename-command
##开启 AOF，禁用 snapshot
appendonly yes
save “”
##slaveof no one
slave-read-only yes
```

```
##redis.conf
##redis-1,通过启动参数配置为 slave，配置文件保持独立
port 6479
slaveof 127.0.0.1 6379
##-----------其他配置和 master 保持一致-----------##
```

```
##redis.conf
##redis-1,通过启动参数配置为 slave，配置文件保持独立
port 6579
slaveof 127.0.0.1 6379
##-----------其他配置和 master 保持一致-----------##
```

2) sentinel.conf

请首先在各个 redis 服务中 sentinel.conf 同目录下新建 local-sentinel.conf,并将复制如下配置信息.

```
##redis-0
##sentinel 实例之间的通讯端口
port 26379
sentinel monitor def_master 127.0.0.1 6379 2
sentinel auth-pass def_master 012_345^678-90
sentinel down-after-milliseconds def_master 30000
```


Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

```
7.    sentinel can-failover def_master yes
8.    sentinel parallel-syncs def_master 1
9.    sentinel failover-timeout def_master 900000
```

```
1.    ##redis-1
2.    port 26479
3.    ##--------其他配置同上-------##
```

```
1.    ##redis-2
2.    port 26579
3.    ##--------其他配置同上-------#
```

3）启动与检测

```
1.    ##redis-0(默认为 master)
2.    > ./redis-server --include ../redis.conf
3.    ##启动 sentinel 组件
4.    > ./redis-sentinel ../local-sentinel.conf
```

按照上述指令,依次启动 redis-0,redis-1,redis-2;在启动 redis-1 和 redis-2 的时候,你会发现在 redis-0 的 sentinel 控制台会输出"+sentinel ..."字样,表示有新的 sentinel 实例加入到监控.不过此处需要提醒,首次构建 sentinel 环境时,必须首先启动 master 机器.

此后你可以使用任意一个"redis-cli"窗口,输入"INFO"命令,可以查看当前 server 的状态:

```
1.    > ./redis-cli -h 127.0.0.1 -p 6379
2.    ##如下为打印信息摘要:
3.    #Replication
4.    role:master
5.    connected_salves:2
6.    slave0:127.0.0.1,6479,online
7.    slave1:127.0.0.1.6579,online
```

"INFO"指令将会打印完整的服务信息,包括集群,我们只需要关注"replication"部分,这部分信息将会告诉我们"当前 server 的角色"以及指向它的所有的 slave 信息.可以通过在任何一个 slave 上,使用"INFO"指令获得当前 slave 所指向的 master 信息.

"INFO"指令不仅可以帮助我们获得集群的情况,当然 sentinel 组件也是使用"INFO"做同样的事情.

当上述部署环境稳定后,我们直接关闭 redis-0,在等待"down-after-milliseconds"秒之后(30 秒),redis-0/redis-1/redis-2 的 sentinel 窗口会立即打印 "+sdown""+odown""+failover""+selected-slave""+promoted-slave""+slave-reconf"等等一系列指令,这些指令标明当 master 失效后,sentinel 组件进行 failover 的过程.

当环境再次稳定后,我们发现,redis-1 被提升("promoted")为 master,且 redis-2 也通过"slave-reconf"过程之后跟随了 redis-1.

如果此后想再次让 redis-0 加入集群,你需要首先通过"INFO"指令找到当前的 masterip + port,并在启动指令中明确指明 slaveof 参数:

```
1.    > ./redis-server --include ../redis.conf --slaveof 127.0.0.1 6479
```

sentinel 实例需要全程处于启动状态,如果只启动 server 而不启动相应的 sentinel,仍然不能确保 server 能够正确的被监控和管理.

二. sentinel 原理

首先解释 2 个名词:SDOWN 和 ODOWN.

- SDOWN:subjectively down,直接翻译的为"主观"失效,即当前 sentinel 实例认为某个 redis 服务为"不可用"状态.
- ODOWN:objectively down,直接翻译为"客观"失效,即多个 sentinel 实例都认为 master 处于"SDOWN"状态,那么此时 master 将处于 ODOWN,ODOWN 可以简单理解为 master 已经被集群确定为"不可用",将会开启 failover.

SDOWN 适合于 master 和 slave,但是 ODOWN 只会使用于 master;当 slave 失效超过"down-after-milliseconds"后,那么所有 sentinel 实例都会将其标记为"SDOWN".

1) SDOWN 与 ODOWN 转换过程:

- 每个 sentinel 实例在启动后,都会和已知的 slaves/master 以及其他 sentinels 建立 TCP 连接,并周期性发送 PING(默认为 1 秒)
- 在交互中,如果 redis-server 无法在"down-after-milliseconds"时间内响应或者响应错误信息,都会被认为此 redis-server 处于 SDOWN 状态.
- 如果 2)中 SDOWN 的 server 为 master,那么此时 sentinel 实例将会向其他 sentinel 间歇性(一秒)发送"is-master-down-by-addr <ip> <port>"指令并获取响应信息,如果足够多的 sentinel 实例检测到 master 处于 SDOWN,那么此时当前 sentinel 实例标记 master 为 ODOWN...其他 sentinel 实例做同样的交互操作.配置项"sentinel monitor <masername> <masterip> <masterport> <quorum>",如果检测到 master 处于 SDOWN 状态的 slave 个数达到<quorum>,那么此时此 sentinel 实例将会认为 master 处于 ODOWN.
- 每个 sentinel 实例将会间歇性(10 秒)向 master 和 slaves 发送"INFO"指令,如果 master 失效且没有新 master 选出时,每 1 秒发送一次"INFO";"INFO"的主要目的就是获取并确认当前集群环境中 slaves 和 master 的存活情况.
- 经过上述过程后,所有的 sentinel 对 master 失效达成一致后,开始 failover.

2) Sentinel 与 slaves"自动发现"机制:

在 sentinel 的配置文件中(local-sentinel.conf),都指定了 port,此 port 就是 sentinel 实例侦听其他 sentinel 实例建立链接的端口.在集群稳定后,最终会每个 sentinel 实例之间都会建立一个 tcp 链接,此链接中发送"PING"以及类似于"is-master-down-by-addr"指令集,可用用来检测其他 sentinel 实例的有效性以及

"ODOWN"和"failover"过程中信息的交互.

在 sentinel 之间建立连接之前,sentinel 将会尽力和配置文件中指定的 master 建立连接.sentinel 与 master 的连接中的通信主要是基于 pub/sub 来发布和接收信息,发布的信息内容包括当前 sentinel 实例的侦听端口:

```
1.    +sentinel sentinel 127.0.0.1:26579 127.0.0.1 26579 ....
```

发布的主题名称为"__sentinel__:hello";同时 sentinel 实例也是"订阅"此主题,以获得其他 sentinel 实例的信息.由此可见,环境首次构建时,在默认 master 存活的情况下,所有的 sentinel 实例可以通过 pub/sub 即可获得所有的 sentinel 信息,此后每个 sentinel 实例即可以根据+sentinel 信息中的"ip+port"和其他 sentinel 逐个建立 tcp 连接即可.不过需要提醒的是,每个 sentinel 实例均会间歇性(5 秒)向"__sentinel__:hello"主题中发布自己的 ip+port,目的就是让后续加入集群的 sentinel 实例也能或得到自己的信息.

根据上文,我们知道在 master 有效的情况下,即可通过"INFO"指令获得当前 master 中已有的 slave 列表;此后任何 slave 加入集群,master 都会向"主题中"发布"+slave 127.0.0.1:6579 ..",那么所有的 sentinel 也将立即获得 slave 信息,并和 slave 建立链接并通过 PING 检测其存活性.

补充一下,每个 sentinel 实例都会保存其他 sentinel 实例的列表以及现存的 master/slaves 列表,各自的列表中不会有重复的信息(不可能出现多个 tcp 连接),对于 sentinel 将使用 ip+port 做唯一性标记,对于 master/slaver 将使用 runid 做唯一性标记,其中 redis-server 的 runid 在每次启动时都不同.

3) Leader 选举:

其实在 sentinels 故障转移中，仍然需要一个“Leader”来调度整个过程：master 的选举以及 slave 的重配置和同步。当集群中有多个 sentinel 实例时，如何选举其中一个 sentinel 为 leader 呢？

在配置文件中“can-failover”“quorum”参数，以及“is-master-down-by-addr”指令配合来完成整个过程。

A) “can-failover”用来表明当前 sentinel 是否可以参与“failover”过程，如果为“YES”则表明它将有能力参与“Leader”的选举，否则它将作为“Observer”，observer 参与 leader 选举投票但不能被选举；

B) “quorum”不仅用来控制 master ODOWN 状态确认，同时还用来选举 leader 时最小“赞同票”数；

C) “is-master-down-by-addr”，在上文中以及提到，它可以用来检测“ip + port”的 master 是否已经处于 SDOWN 状态，不过此指令不仅能够获得 master 是否处于 SDOWN，同时它还额外的返回当前 sentinel 本地“投票选举”的 Leader 信息(runid);

每个 sentinel 实例都持有其他的 sentinels 信息，在 Leader 选举过程中(当为 leader 的 sentinel 实例失效时，有可能 master server 并没失效，注意分开理解)，sentinel 实例将从所有的 sentinels 集合中去除“can-failover = no”和状态为 SDOWN 的 sentinels，在剩余的 sentinels 列表中按照 runid 按照“字典”顺序排序后，取出 runid 最小的 sentinel 实例，并将它“投票选举”为 Leader，并在其他 sentinel 发送的“is-master-down-by-addr”指令时将推选的 runid 追加到响应中。每个 sentinel 实例都会检测“is-master-down-by-addr”的响应结果,如果“投票选举”的 leader 为自己,且状态正常的 sentinels 实例中,“赞同者”的自己的 sentinel 个数不小于(>=) 50% + 1,且不小与<quorum>，那么此 sentinel 就会认为选举成功且 leader 为自己。

在 sentinel.conf 文件中，我们期望有足够多的 sentinel 实例配置“can-failover yes”，这样能够确保当 leader 失效时，能够选举某个 sentinel 为 leader，以便进行 failover。如果 leader 无法产生，比如较少的 sentinels 实例有效，那么 failover 过程将无法继续.

4) failover 过程:

在 Leader 触发 failover 之前，首先 wait 数秒(随即 0~5)，以便让其他 sentinel 实例准备和调整(有可能多个 leader??),如果一切正常，那么 leader 就需要开始将一个 salve 提升为 master，此 slave 必须为状态良好(不能处于 SDOWN/ODOWN 状态)且权重值最低(redis.conf 中)的，当 master 身份被确认后，开始 failover

A）“+failover-triggered”: Leader 开始进行 failover，此后紧跟着“+failover-state-wait-start”，wait 数秒。

B）“+failover-state-select-slave”: Leader 开始查找合适的 slave

C）“+selected-slave”: 已经找到合适的 slave

D） “+failover-state-sen-slaveof-noone”: Leader 向 slave 发送“slaveof no one”指令，此时 slave 已经完成角色转换，此 slave 即为 master

E） “+failover-state-wait-promotition”: 等待其他 sentinel 确认 slave

F）“+promoted-slave”：确认成功

G）“+failover-state-reconf-slaves”: 开始对 slaves 进行 reconfig 操作。

H）“+slave-reconf-sent”:向指定的 slave 发送“slaveof”指令，告知此 slave 跟随新的 master

I）“+slave-reconf-inprog”: 此 slave 正在执行 slaveof + SYNC 过程，如过 slave 收到“+slave-reconf-sent”之后将会执行 slaveof 操作。

J）“+slave-reconf-done”: 此 slave 同步完成，此后 leader 可以继续下一个 slave 的 reconfig 操作。循环 G）

K）“+failover-end”: 故障转移结束

L）“+switch-master”：故障转移成功后，各个 sentinel 实例开始监控新的 master。

三.Sentinel.conf 详解

```
1.    ##sentinel 实例之间的通讯端口
2.    ##redis-0
3.    port 26379
4.    ##sentinel 需要监控的 master 信息：<mastername> <masterIP> <masterPort> <quorum>
5.    ##<quorum>应该小于集群中 slave 的个数,只有当至少<quorum>个 sentinel 实例提交"master 失效
      "
6.    ##才会认为 master 为 O_DWON("客观"失效)
7.    sentinel monitor def_master 127.0.0.1 6379 2
8.
9.    sentinel auth-pass def_master 012_345^678-90
10.
11.   ##master 被当前 sentinel 实例认定为“失效”的间隔时间
12.   ##如果当前 sentinel 与 master 直接的通讯中，在指定时间内没有响应或者响应错误代码，那么
```

```
##当前 sentinel 就认为 master 失效(SDOWN，“主观”失效)
##<mastername> <millseconds>
##默认为 30 秒
sentinel down-after-milliseconds def_master 30000

##当前 sentinel 实例是否允许实施“failover”(故障转移)
##no 表示当前 sentinel 为“观察者”(只参与"投票".不参与实施 failover)，
##全局中至少有一个为 yes
sentinel can-failover def_master yes

##当新 master 产生时，同时进行“slaveof”到新 master 并进行“SYNC”的 slave 个数。
##默认为 1,建议保持默认值
##在 salve 执行 salveof 与同步时，将会终止客户端请求。
##此值较大，意味着“集群”终止客户端请求的时间总和和较大。
##此值较小,意味着“集群”在故障转移期间，多个 salve 向客户端提供服务时仍然使用旧数据。
sentinel parallel-syncs def_master 1

##failover 过期时间，当 failover 开始后，在此时间内仍然没有触发任何 failover 操作，
##当前 sentinel 将会认为此次 failoer 失败。
sentinel failover-timeout def_master 900000

##当 failover 时，可以指定一个“通知”脚本用来告知系统管理员，当前集群的情况。
##脚本被允许执行的最大时间为 60 秒，如果超时，脚本将会被终止(KILL)
##脚本执行的结果：
## 1     -> 稍后重试，最大重试次数为 10;
## 2     -> 执行结束，无需重试
##sentinel notification-script mymaster /var/redis/notify.sh

##failover 之后重配置客户端，执行脚本时会传递大量参数，请参考相关文档
# sentinel client-reconfig-script <master-name> <script-path>
```

采用 keepalived 实现 redis 高可用

1. 相关软件

redis： 开源高性能键值存储 nosql

keepalived： 开源高可用软件

2. 测试环境

redis 服务器两台，采用虚拟机来测试

操作系统是: centos5.4

主(master)： 192.168.136.128

备份(backup)： 192.168.136.129

实现目标： 主服务宕机或服务停止，系统可以继续对外提供服务，实现高可用，同时数据不丢失

为了实现高可用，采用高可用软件 keepalived 实现，所以 对外服务的 ip 地址使用 keepalived 服务产生的虚拟 IP

假设虚拟 ip 为： 192.168.136.100

就是说对外服务的 ip 地址是 192.168.136.100

3. 搭建步骤：

1. 安装 redis

wget http://redis.googlecode.com/files/redis-2.0.0-rc4.tar.gz

tar zxvf redis-2.0.0-rc4.tar.gz

cd redis-2.0.0-rc4

make

cp redis.conf /etc/ 这个文件是 redis 启动的配置文件

cp redis-benchmark redis-cli redis-server /usr/bin/

redis-server /etc/redis.conf

redis-cli

注意修改配置：daemonize yes

2. 安装高可用软件 keepalived

wget http://www.keepalived.org/software/keepalived-1.1.20.tar.gz

tar zxvf keepalived-1.1.20.tar.gz

cd keepalived-1.1.20

./configure --with-kernel-dir=/usr/src/kernels/2.6.18-164.el5-x86_64/

make && make install

cp /usr/local/etc/rc.d/init.d/keepalived /etc/rc.d/init.d/

cp /usr/local/etc/sysconfig/keepalived /etc/sysconfig/
mkdir /etc/keepalived
cp /usr/local/etc/keepalived/keepalived.conf /etc/keepalived/
cp /usr/local/sbin/keepalived /usr/sbin
/etc/init.d/keepalived start

4. 关于配置文件：

redis: 注意配置主备服务器都为 master.

配置文件路径：/etc/redis.conf

keepalived:

配置文件路径： /etc/keepalived/keepalived.conf

5. 相关的命令：

keepalived 相关命令
/etc/init.d/keepalived start

service keepalived status
service keepalived stop
service keepalived start
redis-cli shutdown
quit：关闭连接（connection）

redis-cli -h 192.168.136.100 info
redis-cli -h 192.168.136.128 info
redis-cli -h 192.168.136.129 info

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

chmod +x /etc/keepalived/scripts/*.sh

redis-server /etc/redis.conf

redis-cli

dos2unix test.sh

dos2unix *.sh

ps -ef | grep redis

ip add

tailf /var/log/keepalived-redis-state.log

6. 辅助 shell 脚本

/etc/keepalived/scripts/redis_master.sh

/etc/keepalived/scripts/redis_backup.sh

/etc/keepalived/scripts/redis_fault.sh

/etc/keepalived/scripts/redis_stop.sh

7. 测试流程

脚本创建完成以后，我们开始按照如下流程进行测试：

1.启动 Master 上的 Redis

/etc/init.d/redis start

2.启动 Slave 上的 Redis

/etc/init.d/redis start

3.启动 Master 上的 Keepalived

/etc/init.d/keepalived start

4.启动 Slave 上的 Keepalived

/etc/init.d/keepalived start

5.尝试通过 VIP 连接 Redis:

redis-cli -h 192.168.132.100 INFO

连接成功，Slave 也连接上来了。

role:master

slave0:192.168.132.129,6379,online

6.尝试插入一些数据：

redis-cli -h 192.168.132.100 SET Hello Redis

OK

从 VIP 读取数据

redis-cli -h 192.168.132.100 GET Hello

"Redis"

从 Master 读取数据

redis-cli -h 192.168.132.128 GET Hello

"Redis"

从 Slave 读取数据

redis-cli -h 192.168.132.129 GET Hello

"Redis"

下面，模拟故障产生：

将 Master 上的 Redis 进程杀死：

killall -9 redis-server

查看 Master 上的 Keepalived 日志

tailf /var/log/keepalived-redis-state.log

[fault]

Thu Sep 27 08:29:01 CST 2012

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

同时 Slave 上的日志显示：

tailf /var/log/keepalived-redis-state.log

[master]

Fri Sep 28 14:14:09 CST 2012

Being master....

Run SLAVEOF cmd ...

OK

Run SLAVEOF NO ONE cmd ...

OK

然后我们可以发现，Slave 已经接管服务，并且担任 Master 的角色了。

redis-cli -h 192.168.132.100 INFO

redis-cli -h 192.168.132.128 INFO

role:master

然后我们恢复 Master 的 Redis 进程

/etc/init.d/redis start

查看 Master 上的 Keepalived 日志

tailf /var/log/keepalived-redis-state.log

[master]

Thu Sep 27 08:31:33 CST 2012

Being master....

Run SLAVEOF cmd ...

OK

Run SLAVEOF NO ONE cmd ...

OK

同时 Slave 上的日志显示：

tailf /var/log/keepalived-redis-state.log

[backup]

Fri Sep 28 14:16:37 CST 2012

Being slave....

Run SLAVEOF cmd ...

OK

可以发现目前的 Master 已经再次恢复了 Master 的角色，故障切换以及自动恢复都成功了。

9. java 版本简单测试代码

```
package com.yanek.redis;

import redis.clients.jedis.Jedis;

public class Test {

 /**
  * @param args
  */
 public static void main(String[] args) {


    Jedis jedis = new Jedis("192.168.132.100");
    jedis.set("id", "10000");
    String id = jedis.get("id");
    jedis.set("name", "javaboy2012");
    String name = jedis.get("name");
    System.out.println("id="+id);
    System.out.println("name="+name);
    System.out.println("name="+jedis.get("ray"));
 }

}
```

10. 可能遇到相关问题和处理办法：

1. 如果两机器不能连通，请注意关掉防火墙： service iptables stop

2. windows 上编写的 shell 脚本，要注意通过 dos2unix 命令转换，否则 shell 脚本可能有不可见符号不能正常执行。

基于 keepalived、redis sentinel 的多实例 redis 集群

硬件

机器名	IP	作用
master	192.168.0.2	redis 的 master 服务器,两个主实例
slave1	192.168.0.3	redis 的 slave 服务器，两个从实例
slave2	192.168.0.4	redis 的 slave 服务器，两个从实例
route1	192.168.0.5【虚拟 IP：192.168.0.7】	keepalived 和 redis sentinel 服务器，承载写 redis 的 VIP【虚拟 ip】，做写的双机热备的主 master 指定，redis 哨兵的安装节点 1
route2	192.168.0.6【虚拟 IP：192.168.0.8】	keepalived 和 redis sentinel 服务器，承载读 redis 的 VIP，做读的负载均衡和写的双机热备的 master 备份路由指定，redis 哨兵的安装节点 2

route1

1.安装 redis 在 route1 上，安装路径/usr/local/redis/

2.在 redis 安装路径下创建 scripts 目录，将需要的脚本复制到此处：

1	RunCmd.py	基础功能模块，提供 redis 服务超时检查
2	master_config_set.py	将 master 的 save 参数配置为空
3	redischeck.py	检查 master 的 redis 服务状态
4	slave_config_set.py	将 slave 的 save 参数配置为特定值
5	weightchange.py	调整读的 redis 服务在 keepalived 的权重

详细的 keepalived 配置，

```
! Configuration File for keepalived

global_defs {
   notification_email {
     接收邮箱
   }
   notification_email_from 发送邮箱
   smtp_server 邮件服务器
   smtp_connect_timeout 30
   router_id LVS_DEVEL
}
vrrp_instance VI_1 {
    state MASTER
    interface eth1
    lvs_sync_daemon_inteface eth1
```

```
    virtual_router_id 100
    priority 160
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.0.7
    }
}
vrrp_instance VI_2 {
    state BACKUP
    interface eth1
    lvs_sync_daemon_inteface eth1
    virtual_router_id 101
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.0.
    }
}
virtual_server 192.168.0.7 6379 {
        delay_loop 3
        lb_algo rr
        lb_kind DR
        #nat_mask 255.255.255.0
        persistence_timeout 15
        protocol TCP
        real_server 192.168.0.2 6379 {
                weight 8
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.2 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.2
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6379 {
```

```
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.3 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.3
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6379 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.4 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.4
6379"
                misc_timeout 5
                msic_dynamic
                }
        }
}
virtual_server 192.168.0.7 6380 {
        delay_loop 3
        lb_algo rr
        lb_kind DR
        #nat_mask 255.255.255.0
        persistence_timeout 15
        protocol TCP
        real_server 192.168.0.2 6380 {
                weight 8
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.2 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.2
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6380 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.3 6380"
                MISC_CHECK {
```

```
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.3
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6380 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.4 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.4
6380"
                misc_timeout 5
                msic_dynamic
                }
        }
}
virtual_server 192.168.0.8 6379 {
        delay_loop 3
        lb_algo wrr
        lb_kind DR
        persistence_timeout 30
        protocol TCP
        real_server 192.168.0.2 6379 {
                weight 6
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.2 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.2
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6379 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.3 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.3
6379"
                misc_timeout 5
                misc_dynamic
                }
```

```
        }
        real_server 192.168.0.4 6379 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.4 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.4
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
}
virtual_server 192.168.0.8 6380 {
        delay_loop 3
        lb_algo wrr
        lb_kind DR
        persistence_timeout 30
        protocol TCP
        real_server 192.168.0.2 6380 {
                weight 6
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.2 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.2
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6380 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.3 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.3
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6380 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.4 6380"
```

```
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.4
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
}

route2 的 keepalived 配置文件

! Configuration File for keepalived

global_defs {
   notification_email {
     接受邮箱
   }
   notification_email_from 发送邮箱
   smtp_server 邮件服务器
   smtp_connect_timeout 30
   router_id LVS_DEVEL
}
vrrp_instance VI_1 {
    state BACKUP
    interface eth1
    lvs_sync_daemon_inteface eth1
    virtual_router_id 100
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.0.7
    }
}
vrrp_instance VI_2 {
    state MASTER
    interface eth1
    lvs_sync_daemon_inteface eth1
    virtual_router_id 101
    priority 151
    advert_int 1
    authentication {
```

```
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
         192.168.0.8
    }
}
virtual_server 192.168.0.7 6379 {
        delay_loop 3
        lb_algo rr
        lb_kind DR
        persistence_timeout 15
        protocol TCP
        real_server 192.168.0.2 6379 {
                weight 8
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.2 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.2
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6379 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.3 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.3
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6379 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.4 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.4
6379"
                misc_timeout 5
                misc_dynamic
                }
```

```
        }
}
virtual_server 192.168.0.7 6380 {
        delay_loop 3
        lb_algo rr
        lb_kind DR
        persistence_timeout 15
        protocol TCP
        real_server 192.168.0.2 6380 {
                weight 8
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.2 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.2
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6380 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.3 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.3
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6380 {
                weight 3
                notify_up "/usr/local/redis/scripts/master_config_set.py
192.168.0.4 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/redischeck.py 192.168.0.4
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
}
virtual_server 192.168.0.8 6379 {
        delay_loop 3
        lb_algo wrr
```

```
        lb_kind DR
        persistence_timeout 30
        protocol TCP
        real_server 192.168.0.2 6379 {
                weight 1
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.2 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.2
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6379 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.3 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.3
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6379 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.4 6379"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.4
6379"
                misc_timeout 5
                misc_dynamic
                }
        }
}
virtual_server 192.168.0.8 6380 {
        delay_loop 3
        lb_algo wrr
        lb_kind DR
        persistence_timeout 30
        protocol TCP
        real_server 192.168.0.2 6380 {
                weight 1
```

```
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.2 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.2
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.3 6380 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.3 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.3
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
        real_server 192.168.0.4 6380 {
                weight 2
                notify_up "/usr/local/redis/scripts/slave_config_set.py
192.168.0.4 6380"
                MISC_CHECK {
                misc_path "/usr/local/redis/scripts/weightchange.py 192.168.0.4
6380"
                misc_timeout 5
                misc_dynamic
                }
        }
}
```

在 keepalived 使用的脚本

RunCmd.py

```
#!/usr/bin/python
import os;
import sys;
import time;
import fcntl;
import select;
import signal;
import commands;
```

```
import subprocess;
class RunCmd:
    def __init__(self):
        pass;

    def Run(self,ip,port, nTimeOut = 0, nIntervalTime = 0.1):
        lsCmd=['/usr/local/redis/bin/redis-cli','-h',ip,'-p',port,'ping']
        oProc = subprocess.Popen(lsCmd, stdout =subprocess.PIPE, stderr =
subprocess.PIPE)
        istimeout=False
        nStartTime = time.time()
        while True:
            time.sleep(nIntervalTime)
            print("1:")
            print(oProc.poll())
            if None != oProc.poll():
                break;
            if (nTimeOut > 0) and (time.time() - nStartTime) > nTimeOut:
                istimeout=True
                break;
        print("2:")
        print(istimeout)
        if istimeout:
            print(oProc.poll())
            if None == oProc.poll():
                self.KillAll(oProc.pid)
        print("3:")
        print(istimeout)
        return istimeout

    def KillAll(self, nKillPid, nKillSignal = signal.SIGKILL):
        nRet, strOutput = commands.getstatusoutput("kill "+str(nKillPid));#as root
run
        return (True, strOutput)
```

master_config_set.py 脚本

```
#!/usr/bin/python
from RunCmd import RunCmd
import sys,commands
oCmd = RunCmd();
istimeout = oCmd.Run(sys.argv[1],sys.argv[2], 0.1)
if not istimeout:
    cmd="/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+" -p "+sys.argv[2]+"
info"
```

```
    str=commands.getoutput(cmd)
    ismaster=str.count("role:master")
    zero=0
    if ismaster>zero:
        t=commands.getoutput("/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+"
-p "+sys.argv[2]+" config set save \"\"")
        print t


slave_config_set.py

#!/usr/bin/python
from RunCmd import RunCmd
import sys,commands
oCmd = RunCmd();
istimeout = oCmd.Run(sys.argv[1],sys.argv[2], 0.1)
if not istimeout:
    cmd="/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+" -p "+sys.argv[2]+"
info"
    str=commands.getoutput(cmd)
    isslave=str.count("role:slave")
    zero=0
    if isslave>zero:
        t=commands.getoutput("/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+"
-p "+sys.argv[2]+" config set save \"90 1 300 10 60 1000\"")
        print t

redischeck.py

#!/usr/bin/python
from RunCmd import RunCmd
import sys,commands
oCmd = RunCmd();
istimeout = oCmd.Run(sys.argv[1],sys.argv[2], 0.1)
if not istimeout:
        cmd="/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+" -p "+sys.argv[2]+"
info"
        str=commands.getoutput(cmd)
        ismaster=str.count("role:master")
        zero=0
        if ismaster>zero:
          sys.exit(0)
        else:
          sys.exit(1)
else:
```

```
        sys.exit(1)
```

weightchange.py

```
#!/usr/bin/python
from RunCmd import RunCmd
import sys,commands
oCmd = RunCmd();
istimeout = oCmd.Run(sys.argv[1],sys.argv[2], 0.1)
if not istimeout:
        result=1
        cmd="/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+" -p "+sys.argv[2]+"
ping"
        strping=commands.getoutput(cmd)
        zero=0
        ispong=-100
        ispong=strping.count("PONG")
        if ispong>zero:
                result=0
        if result>zero:
                sys.exit(1)
        else:
                cmdmaster="/usr/local/redis/bin/redis-cli -h "+sys.argv[1]+" -p
"+sys.argv[2]+" info"
                str=commands.getoutput(cmdmaster)
                ismaster=-100
                ismaster=str.count("role:master")
                if ismaster>zero:
                  sys.exit(3)
                else:
                  sys.exit(10)
else:
        sys.exit(1)
```

redis 的哨兵的配置文件 sentinel.conf

```
# Example sentinel.conf

# port <sentinel-port>
# The port that this sentinel instance will run on
port 26379

# sentinel monitor <master-name> <ip> <redis-port> <quorum>
#
```

```
# Tells Sentinel to monitor this slave, and to consider it in O_DOWN
# (Objectively Down) state only if at least <quorum> sentinels agree.
#
# Note: master name should not include special characters or spaces.
# The valid charset is A-z 0-9 and the three characters ".-_".
sentinel monitor mymaster 192.168.0.2 6379 2
sentinel monitor mymaster6380 192.168.0.2 6380 2
# sentinel auth-pass <master-name> <password>
#
# Set the password to use to authenticate with the master and slaves.
# Useful if there is a password set in the Redis instances to monitor.
#
# Note that the master password is also used for slaves, so it is not
# possible to set a different password in masters and slaves instances
# if you want to be able to monitor these instances with Sentinel.
#
# However you can have Redis instances without the authentication enabled
# mixed with Redis instances requiring the authentication (as long as the
# password set is the same for all the instances requiring the password) as
# the AUTH command will have no effect in Redis instances with authentication
# switched off.
#
# Example:
#
# sentinel auth-pass mymaster MySUPER--secret-0123passw0rd

# sentinel down-after-milliseconds <master-name> <milliseconds>
#
# Number of milliseconds the master (or any attached slave or sentinel) should
# be unreachable (as in, not acceptable reply to PING, continuously, for the
# specified period) in order to consider it in S_DOWN state (Subjectively
# Down).
#
# Default is 30 seconds.
sentinel down-after-milliseconds mymaster 3800
sentinel down-after-milliseconds mymaster6380 3800
# sentinel can-failover <master-name> <yes|no>
#
# Specify if this Sentinel can start the failover for this master.
sentinel can-failover mymaster yes
sentinel can-failover mymaster6380 yes
# sentinel parallel-syncs <master-name> <numslaves>
#
# How many slaves we can reconfigure to point to the new slave simultaneously
# during the failover. Use a low number if you use the slaves to serve query
```

```
# to avoid that all the slaves will be unreachable at about the same
# time while performing the synchronization with the master.
sentinel parallel-syncs mymaster 1
sentinel parallel-syncs mymaster6380 1
# sentinel failover-timeout <master-name> <milliseconds>
#
# Specifies the failover timeout in milliseconds. When this time has elapsed
# without any progress in the failover process, it is considered concluded by
# the sentinel even if not all the attached slaves were correctly configured
# to replicate with the new master (however a "best effort" SLAVEOF command
# is sent to all the slaves before).
#
# Also when 25% of this time has elapsed without any advancement, and there
# is a leader switch (the sentinel did not started the failover but is now
# elected as leader), the sentinel will continue the failover doing a
# "takeover".
#
# Default is 15 minutes.
sentinel failover-timeout mymaster 90000
sentinel failover-timeout mymaster6380 90000
# SCRIPTS EXECUTION
#
# sentinel notification-script and sentinel reconfig-script are used in order
# to configure scripts that are called to notify the system administrator
# or to reconfigure clients after a failover. The scripts are executed
# with the following rules for error handling:
#
# If script exists with "1" the execution is retried later (up to a maximum
# number of times currently set to 10).
#
# If script exists with "2" (or an higher value) the script execution is
# not retried.
#
# If script terminates because it receives a signal the behavior is the same
# as exit code 1.
#
# A script has a maximum running time of 60 seconds. After this limit is
# reached the script is terminated with a SIGKILL and the execution retried.

# NOTIFICATION SCRIPT
#
# sentinel notification-script <master-name> <script-path>
#
# Call the specified notification script for any sentienl event that is
# generated in the WARNING level (for instance -sdown, -odown, and so forth).
```

```
# This script should notify the system administrator via email, SMS, or any
# other messaging system, that there is something wrong with the monitored
# Redis systems.
#
# The script is called with just two arguments: the first is the event type
# and the second the event description.
#
# The script must exist and be executable in order for sentinel to start if
# this option is provided.
#
# Example:
#
# sentinel notification-script mymaster /var/redis/notify.sh

# CLIENTS RECONFIGURATION SCRIPT
#
# sentinel client-reconfig-script <master-name> <script-path>
#
# When the failover starts, ends, or is aborted, a script can be called in
# order to perform application-specific tasks to notify the clients that the
# configuration has changed and the master is at a different address.
#
# The script is called in the following cases:
#
# Failover started (a slave is already promoted)
# Failover finished (all the additional slaves already reconfigured)
# Failover aborted (in that case the script was previously called when the
#                   failover started, and now gets called again with swapped
#                   addresses).
#
# The following arguments are passed to the script:
#
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
#
# <state> is "start", "end" or "abort"
# <role> is either "leader" or "observer"
#
# The arguments from-ip, from-port, to-ip, to-port are used to communicate
# the old address of the master and the new address of the elected slave
# (now a master) in the case state is "start" or "end".
#
# For abort instead the "from" is the address of the promoted slave and
# "to" is the address of the original master address, since the failover
# was aborted.
#
```

```
# This script should be resistant to multiple invocations.
#
# Example:
#
# sentinel client-reconfig-script mymaster /var/redis/reconfig.sh
```

在两个 route 上修改/etc/sysctl.conf 文件

net.ipv4.ip_forward=1#转发开启

执行 sysctl -p 让文件起效

有防火墙需要设置防火墙转发

配置 realserver

vim /etc/sysctl.conf,添加内容如下:

```
    net.ipv4.conf.lo.arp_ignore = 1
    net.ipv4.conf.lo.arp_announce = 2
    net.ipv4.conf.all.arp_ignore = 1
    net.ipv4.conf.all.arp_announce = 2
```

在 realserver 的 lo 上指定虚拟 ip

有两种方法

1.命令:

```
ip addr add 192.168.0.7/32 dev lo
ip  addr  add  192.168.0.8/32  dev  lo
```

2.修改/etc/sysconfig/network-scripts/下的配置文件，添加两个配置文件 ifcfg-lo:1 和 ifcfg-lo:2

lo:1

```
DEVICE=lo:1
IPADDR=192.168.0.7
NETMASK=255.255.255.255
# If you're having problems with gated making 127.0.0.0/8 a martian,
# you can change this to something else (255.255.255.255, for example)
ONBOOT=yes
```

lo:2

```
DEVICE=lo:2
IPADDR=192.168.0.8
NETMASK=255.255.255.255
# If you're having problems with gated making 127.0.0.0/8 a martian,
# you can change this to something else (255.255.255.255, for example)
ONBOOT=yes
```

配置完成后执行 service network restart

使用 ip addr 查看是否应绑定 ip 成功

完成所有配置后启动不同端口的 redis

master，启动实例，并且将日志存放到/data/redis 路径下

```
nohup redis-server --port 6379 >/data/redis/redis6379.log &
nohup redis-server --port 6380 >/data/redis/redis6380.log &
```

slave

```
nohup redis-server --port --slaveof 192.168.0.2
6379 >/data/redis/redisslave6379.log &
nohup redis-server --port --slaveof 192.168.0.2
6380 >/data/redis/redisslave6380.log &
```

slave 如果有多个实例配置文件则为

```
nohup redis-server
/etc/redis/redis_slave_6379.conf >/data/redis/redisslave6379.log &
nohup redis-server
/etc/redis/redis_slave_6380.conf >/data/redis/redisslave6380.log &
```

数据分区架构与部署

redis cluster

redis cluster 的现状

reids-cluster 计划在 redis3.0 中推出，可以看作者 antirez 的声明:http://antirez.com/news/49 (ps:跳票了好久，今年貌似加快速度了),目前的最新版本是 redis3 beta2(2.9.51).

作者的目标:Redis Cluster will support up to ~1000 nodes. 赞...

目前 redis 支持的 cluster 特性(已亲测):

1):节点自动发现

2):slave->master 选举,集群容错

3):Hot resharding:在线分片

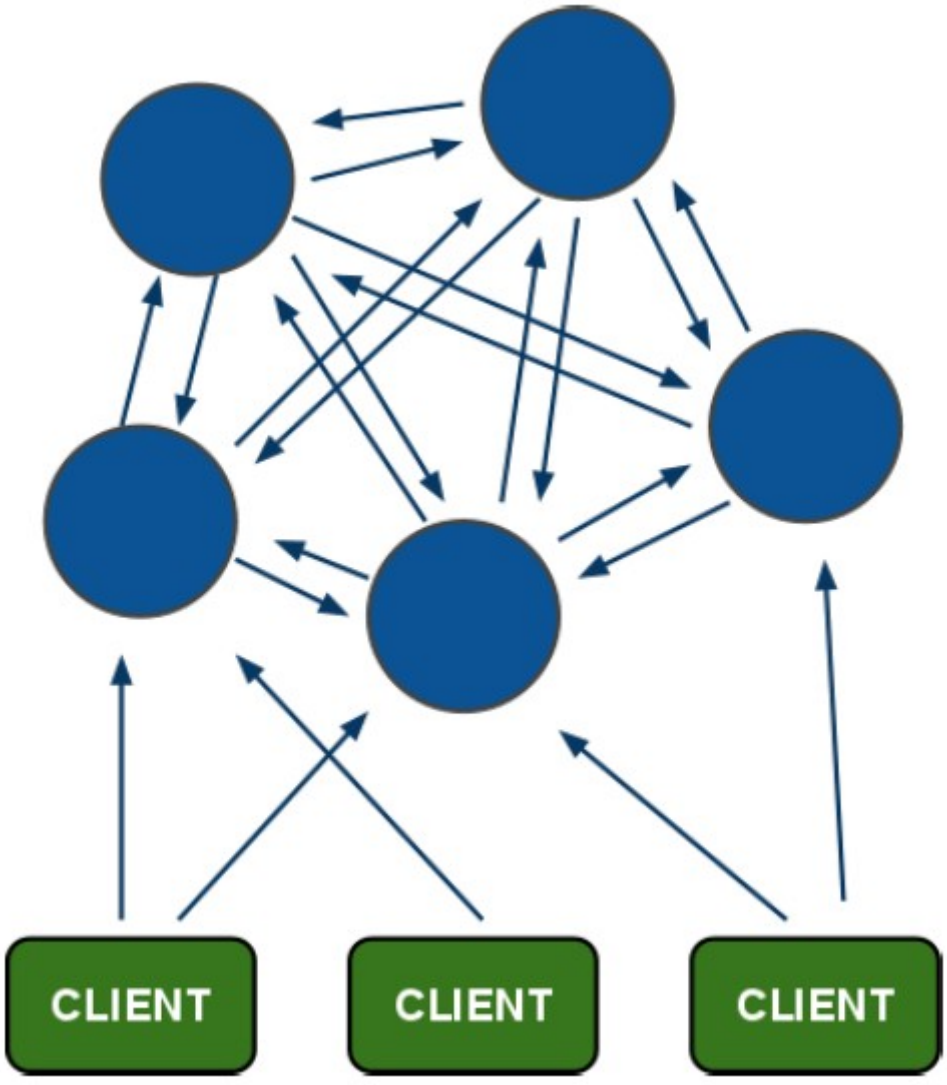
4):进群管理:cluster xxx

5):基于配置(nodes-port.conf)的集群管理

6):ASK 转向/MOVED 转向机制.

redis cluster 架构

1)redis-cluster 架构图



架构细节:

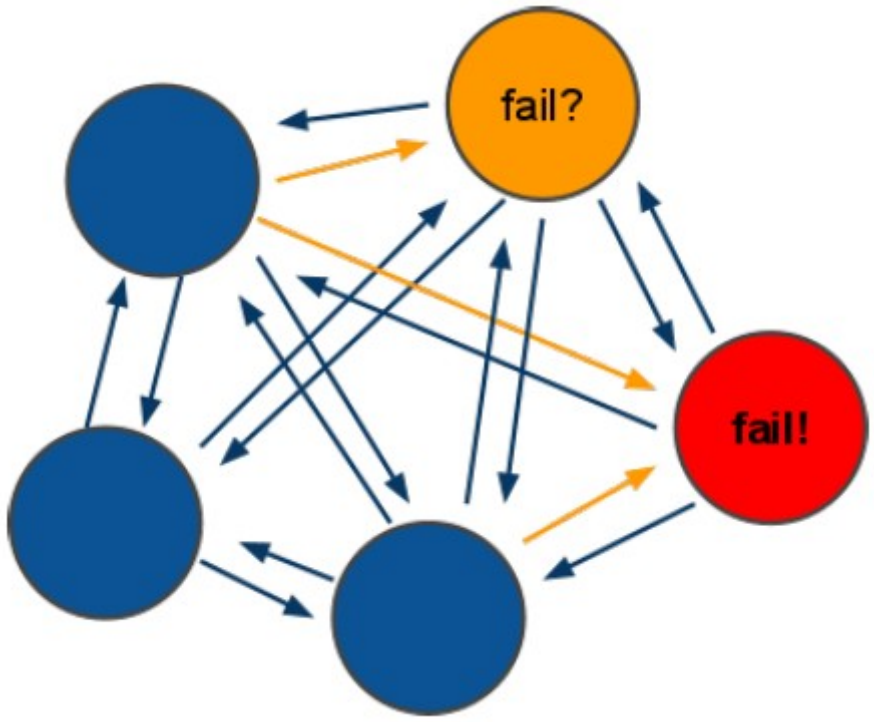
(1)所有的 redis 节点彼此互联(PING-PONG 机制),内部使用二进制协议优化传输速度和带宽.

(2)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.

(3)客户端与 redis 节点直连,不需要中间 proxy 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

(4)redis-cluster 把所有的物理节点映射到[0-16383]slot 上,cluster 负责维护 node<->slot<->value

2) redis-cluster 选举:容错



(1)领着选举过程是集群中所有 master 参与,如果半数以上 master 节点与 master 节点通信超过(cluster-node-timeout),认为当前 master 节点挂掉.

(2):什么时候整个集群不可用(cluster_state:fail),当集群不可用时,所有对集群的操作做都不可用，收到((error) CLUSTERDOWN The cluster is down)错误

a:如果集群任意 master 挂掉,且当前 master 没有 slave.集群进入 fail 状态,也可以理解成进群的 slot 映射[0-16383]不完成时进入 fail 状态.

b:如果进群超过半数以上 master 挂掉，无论是否有 slave 集群进入 fail 状态.

redis cluster 实现

安装 redis cluster

1):安装 redis-cluster 依赖:redis-cluster 的依赖库在使用时有兼容问题,在 reshard 时会遇到各种错误,请按指定版本安装.

(1)确保系统安装 zlib,否则 gem install 会报(no such file to load -- zlib)

Java 代码

```
#download:zlib-1.2.6.tar
./configure
make
make install
```

(1)安装 ruby:version(1.9.2)

Java 代码

```
# ruby1.9.2
cd /path/ruby
./configure -prefix=/usr/local/ruby
make
make install
sudo cp ruby /usr/local/bin
```

(2)安装 rubygem:version(1.8.16)

Java 代码

```
# rubygems-1.8.16.tgz
cd /path/gem
sudo ruby setup.rb
sudo cp bin/gem /usr/local/bin
```

(3)安装 gem-redis:version(3.0.0)

Java 代码

```
1. gem install redis --version 3.0.0
2. #由于源的原因，可能下载失败，就手动下载下来安装
3. #download 地址:http://rubygems.org/gems/redis/versions/3.0.0
4. gem install -l /data/soft/redis-3.0.0.gem
```

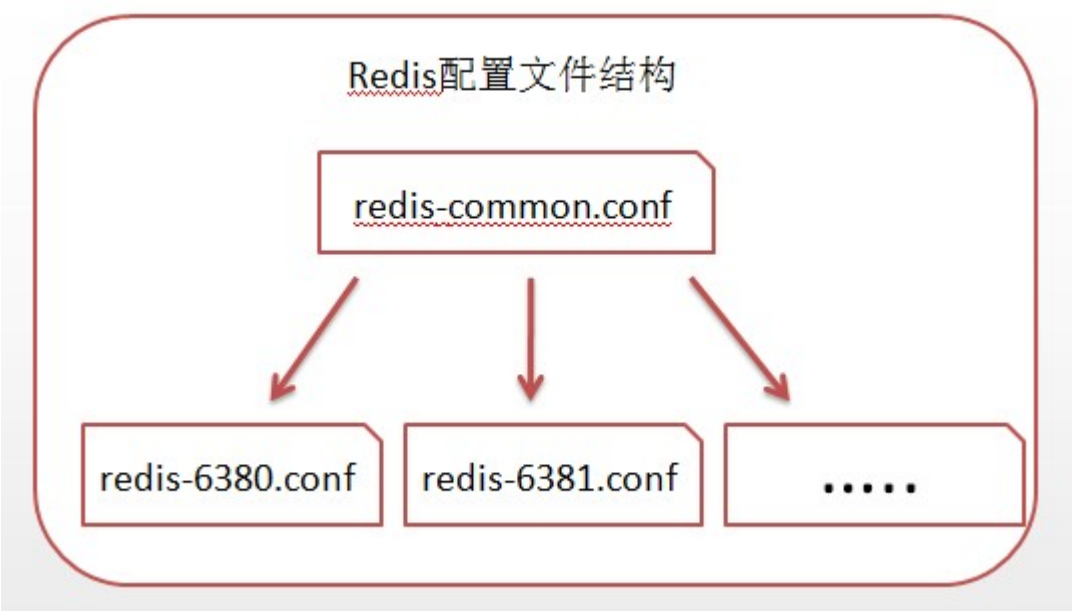
2)安装 redis-cluster

Java 代码

```
1. cd /path/redis
2. make
3. sudo cp /opt/redis/src/redis-server /usr/local/bin
4. sudo cp /opt/redis/src/redis-cli /usr/local/bin
5. sudo cp /opt/redis/src/redis-trib.rb /usr/local/bin
```

2:配置 redis cluster

1)redis 配置文件结构:



使用包含(include)把通用配置和特殊配置分离,方便维护.

2)redis 通用配置.

Java 代码

```
1. #GENERAL
2. daemonize no
3. tcp-backlog 511
4. timeout 0
5. tcp-keepalive 0
```

```
6.  loglevel notice
7.  databases 16
8.  dir /opt/redis/data
9.  slave-serve-stale-data yes
10. #slave 只读
11. slave-read-only yes
12. #not use default
13. repl-disable-tcp-nodelay yes
14. slave-priority 100
15. #打开 aof 持久化
16. appendonly yes
17. #每秒一次 aof 写
18. appendfsync everysec
19. #关闭在 aof rewrite 的时候对新的写操作进行 fsync
20. no-appendfsync-on-rewrite yes
21. auto-aof-rewrite-min-size 64mb
22. lua-time-limit 5000
23. #打开 redis 集群
24. cluster-enabled yes
25. #节点互连超时的阀值
26. cluster-node-timeout 15000
27. cluster-migration-barrier 1
28. slowlog-log-slower-than 10000
29. slowlog-max-len 128
30. notify-keyspace-events ""
31. hash-max-ziplist-entries 512
32. hash-max-ziplist-value 64
33. list-max-ziplist-entries 512
34. list-max-ziplist-value 64
35. set-max-intset-entries 512
36. zset-max-ziplist-entries 128
37. zset-max-ziplist-value 64
38. activerehashing yes
39. client-output-buffer-limit normal 0 0 0
40. client-output-buffer-limit slave 256mb 64mb 60
41. client-output-buffer-limit pubsub 32mb 8mb 60
42. hz 10
43. aof-rewrite-incremental-fsync yes
```

3)redis 特殊配置.

Java 代码

```
1.  #包含通用配置
```

```
include /opt/redis/redis-common.conf
#监听 tcp 端口
port 6379
#最大可用内存
maxmemory 100m
#内存耗尽时采用的淘汰策略:
# volatile-lru -> remove the key with an expire set using an LRU algorithm
# allkeys-lru -> remove any key accordingly to the LRU algorithm
# volatile-random -> remove a random key with an expire set
# allkeys-random -> remove a random key, any key
# volatile-ttl -> remove the key with the nearest expire time (minor TTL)
# noeviction -> don't expire at all, just return an error on write operations
maxmemory-policy allkeys-lru
#aof 存储文件
appendfilename "appendonly-6379.aof"
#rdb 文件,只用于动态添加 slave 过程
dbfilename dump-6379.rdb
#cluster 配置文件(启动自动生成)
cluster-config-file nodes-6379.conf
#部署在同一机器的 redis 实例，把
<span style="font-size: 1em; line-height: 1.5;">auto-aof-rewrite 搓开，防止瞬间 fork 所有 redis 进程做 rewrite,占用大量内存</span>
auto-aof-rewrite-percentage 80-100
```

3:cluster 操作

cluster 集群相关命令,更多 redis 相关命令见文档:http://redis.readthedocs.org/en/latest/

Java 代码

```
集群
CLUSTER INFO 打印集群的信息
CLUSTER NODES 列出集群当前已知的所有节点（node），以及这些节点的相关信息。
节点
CLUSTER MEET <ip> <port> 将 ip 和 port 所指定的节点添加到集群当中，让它成为集群的一份子。
CLUSTER FORGET <node_id> 从集群中移除 node_id 指定的节点。
CLUSTER REPLICATE <node_id> 将当前节点设置为 node_id 指定的节点的从节点。
CLUSTER SAVECONFIG 将节点的配置文件保存到硬盘里面。
槽(slot)
CLUSTER ADDSLOTS <slot> [slot ...] 将一个或多个槽（slot）指派（assign）给当前节点。
CLUSTER DELSLOTS <slot> [slot ...] 移除一个或多个槽对当前节点的指派。
CLUSTER FLUSHSLOTS 移除指派给当前节点的所有槽，让当前节点变成一个没有指派任何槽的节点。
CLUSTER SETSLOT <slot> NODE <node_id> 将槽 slot 指派给 node_id 指定的节点，如果槽已经指派给另一个节点，那么先让另一个节点删除该槽>，然后再进行指派。
CLUSTER SETSLOT <slot> MIGRATING <node_id> 将本节点的槽 slot 迁移到 node_id 指定的节点中。
```

```
15. CLUSTER SETSLOT <slot> IMPORTING <node_id> 从 node_id 指定的节点中导入槽 slot 到本节点。
16. CLUSTER SETSLOT <slot> STABLE 取消对槽 slot 的导入（import）或者迁移（migrate）。
17. 键
18. CLUSTER KEYSLOT <key> 计算键 key 应该被放置在哪个槽上。
19. CLUSTER COUNTKEYSINSLOT <slot> 返回槽 slot 目前包含的键值对数量。
20. CLUSTER GETKEYSINSLOT <slot> <count> 返回 count 个 slot 槽中的键。
```

4:redis cluster 运维操作

1)初始化并构建集群

(1)#启动集群相关节点（必须是空节点）,指定配置文件和输出日志

Java 代码

```
1. redis-server /opt/redis/conf/redis-6380.conf > /opt/redis/logs/redis-6380.log 2>&1 &
2. redis-server /opt/redis/conf/redis-6381.conf > /opt/redis/logs/redis-6381.log 2>&1 &
3. redis-server /opt/redis/conf/redis-6382.conf > /opt/redis/logs/redis-6382.log 2>&1 &
4. redis-server /opt/redis/conf/redis-7380.conf > /opt/redis/logs/redis-7380.log 2>&1 &
5. redis-server /opt/redis/conf/redis-7381.conf > /opt/redis/logs/redis-7381.log 2>&1 &
6. redis-server /opt/redis/conf/redis-7382.conf > /opt/redis/logs/redis-7382.log 2>&1 &
```

(2):使用自带的 ruby 工具(redis-trib.rb)构建集群

Java 代码

```
1. #redis-trib.rb 的 create 子命令构建
2. #--replicas 则指定了为 Redis Cluster 中的每个 Master 节点配备几个 Slave 节点
3. #节点角色由顺序决定,先 master 之后是 slave(为方便辨认,slave 的端口比 master 大 1000)
4. redis-trib.rb create --replicas 1 10.10.34.14:6380 10.10.34.14:6381 10.10.34.14:6382 10.10
   .34.14:7380 10.10.34.14:7381 10.10.34.14:7382
```

(3):检查集群状态,

Java 代码

```
1. #redis-trib.rb 的 check 子命令构建
2. #ip:port 可以是集群的任意节点
3. redis-trib.rb check 1 10.10.34.14:6380
```

最后输出如下信息,没有任何警告或错误，表示集群启动成功并处于 ok 状态

```
Java 代码
1.  [OK] All nodes agree about slots configuration.
2.  >>> Check for open slots...
3.  >>> Check slots coverage...
4.  [OK] All 16384 slots covered.
```

2):添加新 master 节点

(1)添加一个 master 节点:创建一个空节点（empty node），然后将某些 slot 移动到这个空节点上,这个过程目前需要人工干预

a):根据端口生成配置文件(ps:establish_config.sh 是我自己写的输出配置脚本)

```
Java 代码
1.  sh establish_config.sh 6386 > conf/redis-6386.conf
```

b):启动节点

```
Java 代码
1.  nohup redis-server /opt/redis/conf/redis-6386.conf > /opt/redis/logs/redis-6386.log 2>&1 &
```

c):加入空节点到集群
add-node 将一个节点添加到集群里面， 第一个是新节点 ip:port, 第二个是任意一个已存在节点 ip:port

```
Java 代码
1.  redis-trib.rb add-node 10.10.34.14:6386 10.10.34.14:6381
```

node:新节点没有包含任何数据， 因为它没有包含任何 slot。新加入的加点是一个主节点， 当集群需要将某个从节点升级为新的主节点时， 这个新节点不会被选中

d):为新节点分配 slot

```
Java 代码
1.  redis-trib.rb reshard 10.10.34.14:6386
2.  #根据提示选择要迁移的 slot 数量(ps:这里选择 500)
3.  How many slots do you want to move (from 1 to 16384)? 500
4.  #选择要接受这些 slot 的 node-id
5.  What is the receiving node ID? f51e26b5d5ff74f85341f06f28f125b7254e61bf
```

```
#选择 slot 来源:
#all 表示从所有的 master 重新分配,
#或者数据要提取 slot 的 master 节点 id,最后用 done 结束
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node #1:all
#打印被移动的 slot 后，输入 yes 开始移动 slot 以及对应的数据.
#Do you want to proceed with the proposed reshard plan (yes/no)? yes
#结束
```

3):添加新的 slave 节点

a):前三步操作同添加 master 一样

b)第四步:redis-cli 连接上新节点 shell,输入命令:cluster replicate 对应 master 的 node-id

Java 代码

```
cluster replicate 2b9ebcbd627ff0fd7a7bbcc5332fb09e72788835
```

note:在线添加 slave 时，需要 dump 整个 master 进程，并传递到 slave，再由 slave 加载 rdb 文件到内存，rdb 传输过程中 Master 可能无法提供服务,整个过程消耗大量 io,小心操作.

例如本次添加 slave 操作产生的 rdb 文件

Java 代码

```
-rw-r--r-- 1 root root  34946 Apr 17 18:23 dump-6386.rdb
-rw-r--r-- 1 root root  34946 Apr 17 18:23 dump-7386.rdb
```

4):在线 reshard 数据:

对于负载/数据均匀的情况，可以在线 reshard slot 来解决,方法与添加新 master 的 reshard 一样，只是需要 reshard 的 master 节点是老节点.

5):删除一个 slave 节点

Java 代码

```
#redis-trib del-node ip:port '<node-id>'
redis-trib.rb del-node 10.10.34.14:7386 'c7ee2fca17cb79fe3c9822ced1d4f6c5e169e378'
```

6):删除一个 master 节点

a):删除 master 节点之前首先要使用 reshard 移除 master 的全部 slot,然后再删除当前节点(目前只能把被删除

master 的 slot 迁移到一个节点上)

Java 代码

```
#把 10.10.34.14:6386 当前 master 迁移到 10.10.34.14:6380 上
redis-trib.rb reshard 10.10.34.14:6380
#根据提示选择要迁移的 slot 数量(ps:这里选择 500)
How many slots do you want to move (from 1 to 16384)? 500(被删除 master 的所有 slot 数量)
#选择要接受这些 slot 的 node-id(10.10.34.14:6380)
What is the receiving node ID? c4a31c852f81686f6ed8bcd6d1b13accdc947fd2 (ps:10.10.34.14:6380 的 node-id)
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node #1:f51e26b5d5ff74f85341f06f28f125b7254e61bf(被删除 master 的 node-id)
Source node #2:done
#打印被移动的 slot 后，输入 yes 开始移动 slot 以及对应的数据.
#Do you want to proceed with the proposed reshard plan (yes/no)? yes
```

b):删除空 master 节点

Java 代码

```
redis-trib.rb del-node 10.10.34.14:6386 'f51e26b5d5ff74f85341f06f28f125b7254e61bf'
```

三:redis cluster 客户端(Jedis)

客户端基本操作使用

Java 代码

```
<span style="color: #333333; font-family: Arial, sans-serif;"><span style="color: #333333; font-family: Arial, sans-serif;"> private static BinaryJedisCluster jc;
  static {
        //只给集群里一个实例就可以
         Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();
         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 6380));
         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 6381));
```

```
7.          jedisClusterNodes.add(new HostAndPort("10.10.34.14", 6382));
8.          jedisClusterNodes.add(new HostAndPort("10.10.34.14", 6383));
9.          jedisClusterNodes.add(new HostAndPort("10.10.34.14", 6384));
10.         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 7380));
11.         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 7381));
12.         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 7382));
13.         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 7383));
14.         jedisClusterNodes.add(new HostAndPort("10.10.34.14", 7384));
15.         jc = new BinaryJedisCluster(jedisClusterNodes);
16.     }
17. @Test
18.     public void testBenchRedisSet() throws Exception {
19.         final Stopwatch stopwatch = new Stopwatch();
20.         List list = buildBlogVideos();
21.         for (int i = 0; i < 1000; i++) {
22.             String key = "key:" + i;
23.             stopwatch.start();
24.             byte[] bytes1 = protostuffSerializer.serialize(list);
25.             jc.setex(key, 60 * 60, bytes1);
26.             stopwatch.stop();
27.         }
28.         System.out.println("time=" + stopwatch.toString());
29.     }</span></span>
```

jedis 客户端的坑.

1)cluster 环境下 redis 的 slave 不接受任何读写操作，

2)client 端不支持 keys 批量操作,不支持 select dbNum 操作，只有一个 db:select 0

3)JedisCluster 的 info()等单机函数无法调用,返回(No way to dispatch this command to Redis Cluster)错误，.

4)JedisCluster 没有针对 byte[]的 API，需要自己扩展(附件是我加的基于 byte[]的 BinaryJedisCluster api)

安全设计与实施

安全功能相对比较简单，目前只能通过登录需要口令实现。

配置文件中 redis.conf 中设置

requirepass <password> --主密码

masterauth <master-password> --复制时从的密码

单节点安全

[root@wyzc ~]# grep requirepass /etc/redis/redis.conf

If the master is password protected (using the "requirepass" configuration

requirepass foobared

[root@wyzc ~]#

[root@wyzc ~]# redis-cli

127.0.0.1:6379> get s1

"9"

127.0.0.1:6379> exit

[root@wyzc ~]#

[root@wyzc ~]# grep requirepass /etc/redis/redis.conf

If the master is password protected (using the "requirepass" configuration

requirepass foobared

requirepass redis

[root@wyzc ~]# /etc/init.d/redis stop

[root@wyzc ~]# /etc/init.d/redis start

[root@wyzc ~]#

可以在 redis-cli 登录后使用 auth 进行密码验证

[root@wyzc ~]# redis-cli

127.0.0.1:6379> get s1

(error) NOAUTH Authentication required.

127.0.0.1:6379> auth redis

OK

127.0.0.1:6379> get s1

"9"

127.0.0.1:6379> exit

密码验证可以 –a 指定密码

[root@wyzc ~]# redis-cli -a redis

127.0.0.1:6379> get s1

"9"

127.0.0.1:6379> exit

[root@wyzc ~]#

[root@wyzc ~]# /etc/init.d/redis_6379 stop

Stopping ...

(error) NOAUTH Authentication required.

Waiting for Redis to shutdown ...

Waiting for Redis to shutdown ...

发现无法关闭的，因为需要密码验证

[root@wyzc ~]# vi /etc/init.d/redis

//

```
stop)
        if [ ! -f $PIDFILE ]
        then
                echo "$PIDFILE does not exist, process is not running"
        else
                PID=$(cat $PIDFILE)
                echo "Stopping ..."
                $CLIEXEC -a redis -p $REDISPORT shutdown
                while [ -x /proc/${PID} ]
                do
                        echo "Waiting for Redis to shutdown ..."
                        sleep 1
                done
                echo "Redis stopped"
        fi
        ;;
```

//

加入 -a redis 内容 这样停止才能通过验证，否则错误。

主从复制安全

主从复制的密码验证

主: redis.conf 包含

requirepass redis

从: redis.conf 包含

masterauth redis

这样就可以通过密码验证的节点才能与主同步，在一定程度上保护了数据安全。

当然设置防火墙限制哪些 ip 可以访问服务器也是可以达到安全效果的。

如何连接与操作 redis

Redis 信号处理

本文档提供的信息是有关 Redis 是如何应对不同 POSIX 系统下产生的信号异常，比如 SIGTERM,SIGSEGV 等等。

本文档中的信息**只适用于 Redis2.6 或更高版本**。

SIGTERM 信号的处理

SIGTERM 信号会让 Redis 安全的关闭。Redis 收到信号时并不立即退出，而是开启一个定时任务，这个任务就类似执行一次 SHUTDOWN 命令的。 这个定时关闭任务会在当前执行命令终止后立即施行，因此通常有有 0.1 秒或更少时间延迟。

万一 Server 被一个耗时的 LUA 脚本阻塞，如果这个脚本可以被 SCRIPT KILL 命令终止，那么定时执行任务就会在脚本被终止后立即执行，否则直接执行。

这种情况下的 Shutdown 过程也会同时包含以下的操作：

- 如果存在正在执行 RDB 文件保存或者 AOF 重写的子进程，子进程被终止。
- 如果 AOF 功能是开启的，Redis 会通过系统调用 fsync 将 AOF 缓冲区数据强制输出到硬盘。
- 如果 Redis 配置了使用 RDB 文件进行持久化，那么此时就会进行同步保存。由于保存时同步的，那也就不需要额外的内存。
- 如果 Server 是守护进程，PID 文件会被移除。
- 如果 Unix 域的 Socket 是可用的，它也会被移除。
- Server 退出，退出码为 0.

万一 RDB 文件保存失败，Shutdown 失败，Server 则会继续运行以保证没有数据丢失。自从 Redis2.6.11 之后，Redis 不会再次主动 Shutdown，除非它接收到了另一个 SIGTERM 信号或者另外一个 SHUTDOWN 命令

SIGSEGV,SIGBUS,SIGFPF 和 SIGILL 信号的处理

Redis 接收到以下几种信号时会崩溃：

- SIGSEGV
- SIGBUS
- SIGFPE
- SIGILL

如果以上信号被捕获，Redis 会终止所有正在进行的操作，并进行以下操作：

- 包括调用栈信息，寄存器信息，以及 clients 信息会以 bug 报告的形式写入日志文件。
- 自从 Redis2.8（当前为开发版本）之后，Redis 会在系统崩溃时进行一个快速的内存检测以保证系统的可靠性。
- 如果 Server 是守护进程，PID 文件会被移除。
- 最后 server 会取消自己对当前所接收信号的信号处理器的注册，并重新把这个信号发给自己，这是为了保证一些默认的操作被执行，比如把 Redis 的核心 Dump 到文件系统。

子进程被终止时会发生什么

当一个正在进行 AOF 重写的子进程被信号终止，Redis 会把它当成一个错误并丢弃这个 AOF 文件(可能是部分或者完全损坏)。AOF 重写过程会在以后被重新触发。

当一个正在执行 RDB 文件保存的子进程被终止 Redis 会把它当做一个严重的错误，因为 AOF 重写只会导致 AOF 文件冗余，但是 RDB 文件保存失败会导致 Redis 不可用。

如果一个正在保存 RDB 文件的子进程被信号终止或者自身出现了错误(非 0 退出码)，Redis 会进入一种特殊的错误状态，不允许任何写操作。

- Redis 会继续回复所有的读请求。
- Redis 会回复给所有的写请求一个 MISCONFIG 错误。

此错误状态只需被清楚一次就可以进行成功创建数据库文件。

不触发错误状态的情况下终止 RDB 文件的保存

但是有时用户希望可以在不触发错误的情况下终止保存 RDB 文件的子进程。自从 Redis2.6.10 之后就可以使用信号 SIGUSR1，这个信号会被特殊处理：它会像其他信号一样终止子进程，但是父进程不会检测到这个严重的错误，照常接收所有的用户写请求。

Redis 如何处理客户端连接

本文档提供有关 Redis 如何处理来自客户端的信息，从网络层来看包含以下几点：连接、超时、缓冲区以及一些其它类似的主题。

这篇文档所包含的信息 **仅仅适用于 Redis 的 2.6 或者更高版本。**.

客户端的连接的建立

Redis 通过在 TCP 端口上进行监听，或者 Unix socket（如果启用）的方式来接受客户端的连接。当一个新的客户端连接被接受执行以下操作：

- 当 Redis 使用非阻塞 I/O 复用，客户端 socket 设置为非阻塞状态。
- socket TCP_NODELAY 属性被设置确保在连接中我们不会延迟。
- 一个 *可读*的文件事件被创建，因而当新的数据可以被访问时，Reids 可以更快接收客户端在 socket 上的查询

当客户端初始化后，Redis 检查我们是否还在它可以同时处理的客户端的数量限制范围内(这个是使用 `maxclients` 配置指令配置的，请参阅本文档的下一节获取更多的信息。

如果它因为当前已经接受了最大数量的客户端，无法接受当前的客户端，Redis 将尝试发送一个错误给客户端以便让其意识到这种情况，并且立即关闭连接。即使连接被 Redis 立即关闭，错误信息也会返回给客户端，因为新的 socket 输出缓冲区一般情况下都足够放下错误信息，因而客户端内核将处理连接错误。

客户端按照什么顺序被处理

该顺序是由客户端 socket 文件描述符的数字大小及核心报告客户端事件的顺序决定的，因此顺序可以看成不确定的。

不过 Redis 给客户端提供服务时会做以下两件事:

- 每次它从客户端 socket 读取新东西的时候它只执行一次 `read()` 系统调用，以确保当我们有多台客户端连接时，并且有一些要求高客户端以非常快的速率发送查询时，其它客户端不会因此而受到惩罚和经历一个糟糕的延时。（译者注：意思就是不读取完整个 socket 的消息，而是每个 socket 轮流读一次）
- 当系统调用执行完，当前缓冲中的命令不管有多少都会被顺序处理。

最大数量的客户端

在 Redis 2.4 中，同时处理的最大客户端数量的限制是硬编码的。

在 Redis 2.6 中这个限制是动态的：默认情况下为 10000 个客户端，除非在 redis.conf 中配置了 `maxmemory` 配置项。

Redis 通过检查内核中我们可以打开的最多的文件描述符数量，（*soft limit* 被检查），如果限制小于最大连接客户端连接数，则加上 32（这是 Redis 储备给内部使用的文件描述符数量）， 接着这个最大连接客户端的数量将被 Redis 修改为系统要求的值，以便符合在当前操作系统限制下的真正能够处理的客户端数量

当配置的最大客户端数目不起作用时，则日志将在启动时显示，如下面这个例子：

```
$ ./redis-server --maxclients 100000
```

```
[41422] 23 Jan 11:28:33.179 # Unable to set the max number of files
```

```
limit to 100032 (Invalid argument), setting the max clients
```

```
configuration to 10112.
```

当 Redis 配置处理客户的具体数量时，确认操作系统中每个进程文件描述符的限制也相应地设置成最大值是个好主意。

在 Linux 下这些限制可以在当前的会话设置，用下面的命令在系统范围内进行设置：

- ulimit -Sn 100000 # 这个将只在硬限制足够大的情况下生效。
- sysctl -w fs.file-max=100000

输出缓冲限制

Redis 需要为每个客户端处理可变长度的输出，因为简单的命令也可能产生一个需要传送给客户端的巨大的数据量。

也可能只是客户端以较快的速度发送多个的命令产生的更多的输出，当客户端处理新消息的速度比服务端发给给它的速度还慢时，特别是 Pub/Sub 客户端更是如此。

这两个原因将导致客户端输出缓冲增长及内存消耗增多。因为这个原因在默认情况下 Redis 为不同类型的客户端设置了输出缓冲限制。当限制到达后客户端的连接将被关闭，同时事件日志记录在 Redis 的日志文件中。

Redis 使用两种类型的限制：

- **硬限制**是个固定的限制，当大小达到它 Redis 会以最快的速度关闭掉客户端的连接。
- **软限制**依赖于时间，例如每 10 秒 32 兆字节意味着加入客户端拥有比 32 兆字节还大的输出缓冲，持续的在 10 秒内超过的话连接将被关闭。

不同类型的客户端有着不同的默认限制：

- **普通客户端**有着默认为 0 的限制，这意味着没有限制，因为大部分的普通客户端使用阻塞实现发送单个命令，并且在发送下一个命令前等待答复以完全读取，因此去关闭普通客户端的连接始终是没必要的。
- **Pub/Sub 客户端**有默认的 32 兆字节的硬限制及每 60 秒 8 兆字节的软限制。
- **从机**有默认的 256 兆字节的硬限制及每 60 秒 64 兆字节的软限制。

可以在运行时改变这些限制，使用 `CONFIG SET` 命令或者修改 `redis.conf` 以永久地改变它。见

redis.conf 中更多的关于如何设置限制的介绍。

搜索缓冲硬限制

每一个客户端也受到搜索缓冲限制。这是个不可配置的硬限制，当客户端搜索缓冲（这是个我们用来积累客户端的命令的缓冲）达到 1GB 的时候它将关闭连接，这只是个极限限制，用来避免当客户端或者服务端软件出错导致服务器崩溃的情况。

客户端超时

最近版本的 Reids 在默认情况下不会在客户端空闲很久后关闭连接；连接将永久保留。

不过假如你不喜欢这种行为，你可以设置一个超时时间，这样当客户端空闲超过设置的几秒后，客户端连接就会被关闭。

你可以在 redis.conf 中配置这个限制或者简单的使用 CONFIG SET timeout <value>。

记住这个超时时间只适用于多个客户端并且它**不支持 Pub/Sub 客户端**, Pub/Sub 连接是推送类型的连接，因而客户端空闲是正常的。

即使在默认情况下连接是不受超时时间限制的，但是有两种情况设置超时是有意义的：

- 关键任务应用，客户端软件可能因为 Redis 连接饱和而造成出错，造成服务中断。
- 如果一个客户端出错使得服务器因为空闲连接而饱和，使得无法与服务器交互，此时可以作为一个检错机制去连接服务器。

超时并非非常准确：Redis 避免设置计时器或者运行 **O(N)** 算法去轮询检测客户端是否超时，所以检查是渐近的一部分一部分完成的。这意味着有可能当超时时间设置为 10 秒，客户端的连接将在稍晚的时候被关闭，例如当很多客户端在同一时间连接的话，可能 12 秒才被关闭。

客户端命令

Redis 客户端命令允许检查所有连接的客户端的状态、关掉指定的客户端的连接、设置连接的名称。假如你使用一定规模的 Redis 的话这是个很强大的排错工具

CLIENT LIST 命令用来获得连接的客户端列表及它们的状态：

```
redis 127.0.0.1:6379> client list
```

```
addr=127.0.0.1:52555 fd=5 name= age=855 idle=0 flags=N db=0 sub=0
psub=0 multi=-1 qbuf=0 qbuf-free=32768 obl=0 oll=0 omem=0 events=r
cmd=client
addr=127.0.0.1:52787 fd=6 name= age=6 idle=5 flags=N db=0 sub=0
psub=0 multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0 omem=0 events=r
cmd=ping
```

在上面的示例中两台客户端都连接到了 Redis 服务器。一部分有趣的字段的含义如下表所示：

- **addr**：客户端地址，也就是客户端用来连接 Redis 服务器的 IP 和远程端口号
- **fd**：客户端 socket 文件描述符数目。
- **name**：客户端名称，由 `CLIENT SETNAME` 命令设置。
- **age**：连接已经存在了多少秒。
- **idle**：连接已经空闲了几秒。

flags：客户端的类型(N 代表普通客户端，其他参数如下：

O: the client is a slave in MONITOR mode

S: the client is a normal slave server

M: the client is a master

x: the client is in a MULTI/EXEC context

b: the client is waiting in a blocking operation

i: the client is waiting for a VM I/O (deprecated)

d: a watched keys has been modified - EXEC will fail

c: connection to be closed after writing entire reply

u: the client is unblocked

A: connection to be closed ASAP

N: no specific flag set)。

- **omem**：客户端输出缓冲占用的内存量。
- **cmd**：最后执行的命令。

参考 CLIENT LIST 文档，查看完整的字段列表及它们的说明。

当你有了客户端的列表，你可以很简单的使用 `CLIENT KILL` 命令带上客户端 IP 作为参数来关掉连接。

命令 `CLIENT SETNAME` 和 `CLIENT GETNAME` 可以用于设置和取得连接的名称。

redis 常用命令使用

服务命令：

- BGREWRITEAOF 异步重写追加文件
- BGSAVE 异步保存数据集到磁盘上
- CLIENT KILL ip:port 关闭客户端连接
- CLIENT LIST 获得客户端连接列表
- CLIENT PAUSE timeout 暂停处理客户端命令
- CLIENT GETNAME 获得当前连接名称
- CLIENT SETNAME connection-name 设置当前连接的名字
- CONFIG GET parameter 获取配置参数的值
- CONFIG REWRITE 从写内存中的配置文件
- CONFIG SET parameter value 获取配置参数的值
- CONFIG RESETSTAT 复位再分配使用 info 命令报告的统计
- DBSIZE 返回当前数据库里面的 keys 数量
- DEBUG OBJECT key 获取一个 key 的 debug 信息
- DEBUG SEGFAULT 使服务器崩溃
- FLUSHALL 清空所有数据库
- FLUSHDB 清空当前的数据库
- INFO [section]获得服务器的详细信息
- LASTSAVE 获得最后一次同步磁盘的时间
- MONITOR 实时监控服务器
- SAVE 同步数据到磁盘上
- SHUTDOWN [NOSAVE] [SAVE]关闭服务
- SLAVEOF host port 指定当前服务器的主服务器
- SLOWLOG subcommand [argument]管理再分配的慢查询日志
- SYNC 用于复制的内部命令
- TIME 返回当前服务器时间

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

连接命令：

- AUTH password 验证服务器
- ECHO message 回显输入的字符串
- PING Ping 服务器
- QUIT 关闭连接，退出
- SELECT index 选择数据库

键值命令：

- DEL key [key ...]删除一个 key
- DUMP key 导出 key 的值
- EXISTS key 查询一个 key 是否存在
- EXPIRE key seconds 设置一个 key 的过期的秒数
- EXPIREAT key timestamp 设置一个 UNIX 时间戳的过期时间
- KEYS pattern 查找所有匹配给定的模式的键
- MIGRATE host port key destination-db timeout 原子性的将 key 从 redis 的一个实例移到另一个实例
- MOVE key db 移动一个 key 到另一个数据库
- OBJECT subcommand [arguments [arguments ...]]检查内部的再分配对象
- PERSIST key 移除 key 的过期时间
- PEXPIRE key milliseconds 设置一个 key 的过期的毫秒数
- PEXPIREAT key milliseconds-timestamp 设置一个带毫秒的 UNIX 时间戳的过期时间
- PTTL key 获取 key 的有效毫秒数
- RANDOMKEY 返回一个随机的 key
- RENAME key newkey 将一个 key 重命名
- RENAMENX key newkey 重命名一个 key,新的 key 必须是不存在的 key
- RESTORE key ttl serialized-valueCreate a key using the provided serialized value, previously obtained using DUMP.
- SCAN cursor [MATCH pattern] [COUNT count]增量迭代 key
- SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination]对队列、集合、有序集合排序
- TTL key 获取 key 的有效时间（单位：秒）
- TYPE key 获取 key 的存储类型

脚本命令：

- EVAL script numkeys key [key ...] arg [arg ...]在服务器端执行 LUA 脚本
- EVALSHA sha1 numkeys key [key ...] arg [arg ...]在服务器端执行 LUA 脚本
- SCRIPT EXISTS script [script ...]Check existence of scripts in the script cache.
- SCRIPT FLUSH 删除服务器缓存中所有 Lua 脚本。
- SCRIPT KILL 杀死当前正在运行的 Lua 脚本。
- SCRIPT LOAD script 从服务器缓存中装载一个 Lua 脚本。

事务命令：

- DISCARD 丢弃所有 MULTI 之后发的命令
- EXEC 执行所有 MULTI 之后发的命令
- MULTI 标记一个事务块开始
- UNWATCH 取消事务
- WATCH key [key ...]锁定 key 直到执行了 MULTI/EXEC 命令

发布订阅命令：

- PSUBSCRIBE pattern [pattern ...]听出版匹配给定模式的渠道的消息
- PUBLISH channel message 发布一条消息到频道
- PUBSUB subcommand [argument [argument ...]]检查的 Pub/Sub 子系统的状态
- PUNSUBSCRIBE [pattern [pattern ...]]停止发布到匹配给定模式的渠道的消息
- SUBSCRIBE channel [channel ...]聆听发布途径的消息
- UNSUBSCRIBE [channel [channel ...]]停止发布途径的消息

数据类型

字符串（Strings）

字符串是一种最基本的 Redis 值类型。Redis 字符串是二进制安全的，这意味着一个 Redis 字符串能包含任意类型的数据，例如： 一张 JPEG 格式的图片或者一个序列化的 Ruby 对象。

一个字符串类型的值最多能存储 512M 字节的内容。

你可以用 Redis 字符串做许多有趣的事，例如你可以：

- 利用 INCR 命令簇（INCR, DECR, INCRBY）来把字符串当作原子计数器使用。
- 使用 APPEND 命令在字符串后添加内容。
- 将字符串作为 GETRANGE 和 SETRANGE 的随机访问向量。
- 在小空间里编码大量数据，或者使用 GETBIT 和 SETBIT 创建一个 Redis 支持的 Bloom 过滤器。

- APPEND key value 追加一个值到 key 上
- BITCOUNT key [start] [end]统计字符串指定起始位置的字节数
- BITOP operation destkey key [key ...]Perform bitwise operations between strings
- DECR key 整数原子减 1
- DECRBY key decrement 原子减指定的整数
- GET key 获取 key 的值

- GETBIT key offset 返回位的值存储在关键的字符串值的偏移量。
- GETRANGE key start end 获取存储在 key 上的值的一个子字符串
- GETSET key value 设置一个 key 的 value，并获取设置前的值
- INCR key 执行原子加 1 操作
- INCRBY key increment 执行原子增加一个整数
- INCRBYFLOAT key increment 执行原子增加一个浮点数
- MGET key [key ...]获得所有 key 的值
- MSET key value [key value ...]设置多个 key value
- MSETNX key value [key value ...]设置多个 key value,仅当 key 存在时
- PSETEX key milliseconds valueSet the value and expiration in milliseconds of a key
- SET key value 设置一个 key 的 value 值
- SETBIT key offset valueSets or clears the bit at offset in the string value stored at key
- SETEX key seconds value 设置 key-value 并设置过期时间（单位：秒）
- SETNX key value 设置的一个关键的价值，只有当该键不存在
- SETRANGE key offset valueOverwrite part of a string at key starting at the specified offset
- STRLEN key 获取指定 key 值的长度

哈希（Hashes）

Redis Hashes 是字符串字段和字符串值之间的映射，所以它们是完美的表示对象（eg:一个有名，姓，年龄等属性的用户）的数据类型。

```
redis>  HMSET user:1000 username antirez password P1pp0 age 34
OK
redis>  HGETALL user:1000
1) "username"
2) "antirez"
3) "password"
4) "P1pp0"
5) "age"
6) "34"
redis>  HSET user:1000 password 12345
(integer) 0
redis>  HGETALL user:1000
1) "username"
2) "antirez"
3) "password"
4) "12345"
5) "age"
6) "34"
```

一个拥有少量（100 个左右）字段的 hash 需要 很少的空间来存储，所有你可以在一个小型的 Redis 实例中存储上百万的对象。

尽管 Hashes 主要用来表示对象，但它们也能够存储许多元素，所以你也可以用 Hashes 来完成许多其他的任务。

一个 hash 最多可以包含 2^{32-1} 字段-值对（超过 40 亿）。

- HDEL key field [field ...]删除一个或多个哈希域
- HEXISTS key field 判断给定域是否存在于哈希集中
- HGET key field 读取哈希域的的值
- HGETALL key 从哈希集中读取全部的域和值
- HINCRBY key field increment 将哈希集中指定域的值增加给定的数字
- HINCRBYFLOAT key field increment 将哈希集中指定域的值增加给定的浮点数
- HKEYS key 获取 hash 的所有字段
- HLEN key 获取 hash 里所有字段的数量
- HMGET key field [field ...]获取 hash 里面指定字段的值
- HMSET key field value [field value ...]设置 hash 字段值
- HSCAN key cursor [MATCH pattern] [COUNT count]迭代 hash 里面的元素
- HSET key field value 设置 hash 里面一个字段的值
- HSETNX key field value 设置 hash 的一个字段，只有当这个字段不存在时有效
- HVALS key 获得 hash 的所有值

列表（Lists）

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）。

LPUSH 命令插入一个新元素到列表头部，而 RPUSH 命令 插入一个新元素到列表的尾部。当 对一个空 key 执行其中某个命令时，将会创建一个新表。 类似的，如果一个操作要清空列表，那么 key 会从对应的 key 空间删除。这是个非常便利的语义， 因为如果使用一个不存在的 key 作为参数，所有的列表命令都会像在对一个空表操作一样。

一些列表操作及其结果：

```
LPUSH mylist a   # now the list is "a"
```

```
LPUSH mylist b   # now the list is "b","a"
```

```
RPUSH mylist c   # now the list is "b","a","c" (RPUSH was used this
time)
```

一个列表最多可以包含 2^{32-1} 个元素（4294967295，每个表超过 40 亿个元素）。

从时间复杂度的角度来看，Redis 列表主要的特性就是支持时间常数的 插入和靠近头尾部元素的删除，即使是需要插入上百万的条目。 访问列表两端的元素是非常快的，但如果你试着访问一个非常大 的列表的中间元素仍然是十分慢的，因为那是一个时间复杂度为 **O(N)** 的操作。

你可以用 Redis 列表做许多有趣的事，例如你可以：

- 在社交网络中建立一个时间线模型，使用 LPUSH 去添加新的元素到用户时间线中，使用 LRANGE 去检索一些最近插入的条目。
- 你可以同时使用 LPUSH 和 LTRIM 去创建一个永远不会超过指定元素数目的列表并同时记住最后的 N 个元素。
- 列表可以用来当作消息传递的基元（primitive），例如，众所周知的用来创建后台任务的 Resque Ruby 库。
- 你可以使用列表做更多事，这个数据类型支持许多命令，包括像 BLPOP 这样的阻塞命令。

- BLPOP key [key ...] timeout 删除，并获得该列表中的第一元素，或阻塞，直到有一个可用
- BRPOP key [key ...] timeout 删除，并获得该列表中的最后一个元素，或阻塞，直到有一个可用
- BRPOPLPUSH source destination timeout 弹出一个列表的值，将它推到另一个列表，并返回它;或阻塞，直到有一个可用
- LINDEX key index 获取一个元素，通过其索引列表
- LINSERT key BEFORE|AFTER pivot value 在列表中的另一个元素之前或之后插入一个元素
- LLEN key 获得队列(List)的长度
- LPOP key 从队列的左边出队一个元素
- LPUSH key value [value ...]从队列的左边入队一个或多个元素
- LPUSHX key value 当队列存在时，从队到左边入队一个元素
- LRANGE key start stop 从列表中获取指定返回的元素
- LREM key count value 从列表中删除元素
- LSET key index value 设置队列里面一个元素的值
- LTRIM key start stop 修剪到指定范围内的清单
- RPOP key 从队列的右边出队一个元素
- RPOPLPUSH source destination 删除列表中的最后一个元素，将其追加到另一个列表
- RPUSH key value [value ...]从队列的右边入队一个元素
- RPUSHX key value 从队列的右边入队一个元素，仅队列存在时有效

```
lpush list1 a

lpush list1 b

lpush list1 c

lrange list1 0 -1

c
```

```
b
a



rpush list2   a
rpush list2   b
rpush list2   c
lrange list2 0 -1
a
b
c


lpop list1
c


rpop list1
a


lrange list1 0 -1
b


linsert list1 before b   a
lrange list1 0 -1
linsert list1 after   b   c
```

```
lrange list1 0 -1


llen list1

lindex list1 0

lindex list1 1

lindex list1 -1

lindex list1 -2

lset list1 0 aa

lindex list1 0


lpush list3    wyzc

lpush list3    wyzc

lpush list3    wyzc

lrange list3 0 -1

lrem    list3    2 wyzc

lrange list3 0 -1


lpush list4    wyzc1

lpush list4    wyzc2

lpush list4    wyzc3

ltrim list4    1 -1

lrange list4 0 -1


lrange list1 0 -1
```

```
c

b

a

lrange list2 0 -1

a

b

c


rpoplpush list1 list2

lrange list1 0 -1

c

b

lrange list2 0 -1

a

a

b

c
```

集合（Sets）

Redis 集合是一个无序的字符串合集。你可以以 **O(1)** 的时间复杂度（无论集合中有多少元素时间复杂度都为常量）完成 添加，删除以及测试元素是否存在的操作。

Redis 集合有着不允许相同成员存在的优秀特性。向集合中多次添加同一元素，在集合中最终只会存在一个此元素。实际上这就意味着，在添加元素前，你并不需要事先进行*检验此元素是否已经存在*的操作。

一个 Redis 列表十分有趣的事是，它们支持一些服务端的命令从现有的集合出发去进行集合运算。 所以你可以在很短的时间内完成合并（union）,求交(intersection)，找出不同元素的操作。

一个集合最多可以包含 2^{32-1} 个元素（4294967295，每个集合超过 40 亿个元素）。

你可以用 Redis 集合做很多有趣的事，例如你可以：

- 用集合跟踪一个独特的事。想要知道所有访问某个博客文章的独立 IP？只要每次都用 SADD 来处理一个页面访问。那么你可以肯定重复的 IP 是不会插入的。
- Redis 集合能很好的表示关系。你可以创建一个 tagging 系统，然后用集合来代表单个 tag。接下来你可以用 SADD 命令把所有拥有 tag 的对象的所有 ID 添加进集合，这样来表示这个特定的 tag。如果你想要同时有 3 个不同 tag 的所有对象的所有 ID，那么你需要使用 SINTER.
- 使用 SPOP 或者 SRANDMEMBER 命令随机地获取元素。

- SADD key member [member ...]添加一个或者多个元素到集合(set)里
- SCARD key 获取集合里面的元素数量
- SDIFF key [key ...]获得队列不存在的元素
- SDIFFSTORE destination key [key ...]获得队列不存在的元素，并存储在一个关键的结果集
- SINTER key [key ...]获得两个集合的交集
- SINTERSTORE destination key [key ...]获得两个集合的交集，并存储在一个关键的结果集
- SISMEMBER key member 确定一个给定的值是一个集合的成员
- SMEMBERS key 获取集合里面的所有 key
- SMOVE source destination member 移动集合里面的一个 key 到另一个集合
- SPOP key 删除并获取一个集合里面的元素
- SRANDMEMBER key [count]从集合里面随机获取一个 key
- SREM key member [member ...]从集合里删除一个或多个 key
- SSCAN key cursor [MATCH pattern] [COUNT count]迭代 set 里面的元素
- SUNION key [key ...]添加多个 set 元素
- SUNIONSTORE destination key [key ...]合并 set 元素，并将结果存入新的 set 里面

```
sadd   s1   a

sadd   s1   b

sadd   s1   b

smembers   s1

b

a

srem s1   b


sadd   s2   a
```

```
sadd    s2    c

spop s2



smembers s1

a

b


smembers s2

a

b

c


sdiff s1    s2

c

sdiffstore s1    s2    s3


sinter    s1 s2

a

b

sinterstore    s1 s2 s3


sunion s1 s2

sunionstore s1 s2 s3
```

smove s2 s1 c

scard s1

sismember s1 d

sismember s1 a

srandmember s1

srandmember s1

有序集合（Sorted sets）

Redis 有序集合和 Redis 集合类似，是不包含 相同字符串的合集。它们的差别是，每个有序集合 的成员都关联着一个评分，这个评分用于把有序集 合中的成员按最低分到最高分排列。

使用有序集合，你可以非常快地（**O(log(N))**）完成添加，删除和更新元素的操作。 因为元素是在*插入时就排好序*的，所以很快地通过评分(score)或者 位次(position)获得一个范围的元素。 访问有序集合的中间元素同样也是非常快的，因此你可以使用有序集合作为一个没用重复成员的智能列表。 在这个列表中， 你可以轻易地访问任何你需要的东西: 有序的元素，快速的存在性测试，快速访问集合中间元素！

简而言之，使用有序集合你可以很好地完成 很多在其他数据库中难以实现的任务。

使用有序集合你可以：

- 在一个巨型在线游戏中建立一个排行榜，每当有新的记录产生时，使用 ZADD 来更新它。你可以用 ZRANGE 轻松地获取排名靠前的用户， 你也可以提供一个用户名，然后用 ZRANK 获取他在排行榜中的名次。 同时使用 ZRANK 和 ZRANGE 你可以获得与指定用户有相同分数的用户名单。 所有这些操作都非常*迅速*。
- 有序集合通常用来索引存储在 Redis 中的数据。 例如：如果你有很多的 hash 来表示用户，那么你可以使用一个有序集合，这个集合的年龄字段用来当作评分，用户 ID 当作值。用 ZRANGEBYSCORE 可以简单快速地检索到给定年龄段的所有用户。
- 有序集合或许是最高级的 Redis 数据类型

- ZADD key score member [score member ...]添加到有序 set 的一个或多个成员，或更新的分数，如果它已经存在
- ZCARD key 获取一个排序的集合中的成员数量
- ZCOUNT key min max 给定值范围内的成员数与分数排序
- ZINCRBY key increment member 增量的一名成员在排序设置的评分
- ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]相交多个排序集，导致排序的设置存储在一个新的关键

- ZRANGE key start stop [WITHSCORES]返回的成员在排序设置的范围，由指数
- ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]返回的成员在排序设置的范围，由得分
- ZRANK key member 确定在排序集合成员的索引
- ZREM key member [member ...]从排序的集合中删除一个或多个成员
- ZREMRANGEBYRANK key start stop 在排序设置的所有成员在给定的索引中删除
- ZREMRANGEBYSCORE key min max 删除一个排序的设置在给定的分数所有成员
- ZREVRANGE key start stop [WITHSCORES]在排序的设置返回的成员范围，通过索引，下令从分数高到低
- ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]返回的成员在排序设置的范围，由得分，下令从分数高到低
- ZREVRANK key member 确定指数在排序集的成员，下令从分数高到低
- ZSCAN key cursor [MATCH pattern] [COUNT count]迭代 sorted sets 里面的元素
- ZSCORE key member 获取成员在排序设置相关的比分
- ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]添加多个排序集和导致排序的设置存储在一个新的关键

```
zadd   z1   1   a

zadd   z1   2   b

zadd   z1   3   b

zrange z1 0   -1   withscores

a

1

b

3


zrem z1 b

zrange z1 0   -1   withscores

a

1
```

zadd z2 1 a

zadd z2 2 b

zincrby z2 2 b

zrange z2 0 -1 withscores

zrank z2 b

zrevrank z2 b

zrangebyscore z2 2 3 withscores

zcount z2 1 3

zcard z2

zremrangebyrank z2 1 1

zremrangebyscore z2 1 2

事务与大量数据快速插入

事物的使用

1.Redis 事物通过 MULTI 命令开始。 这条命令总是返回 OK。

2.然后用户可以执行多条指令，redis 不会马上执行这些指令，还只是放入到队列中。

3.当执行 exec 指令时，所有的指令执行。

4.调用 discard 指令，将会 flush 事物队列，并且退出事物。

如下:

redis 127.0.0.1:6379> multi

OK

```
redis 127.0.0.1:6379> set foo 1

QUEUED

redis 127.0.0.1:6379> incr foo

QUEUED

redis 127.0.0.1:6379> incr foo

QUEUED

redis 127.0.0.1:6379> exec

1) OK

2) (integer) 2

3) (integer) 3
```

从以上会话中能看到 multi 命令返回的回复是一个数组，每个元素即是事物中每条指令的回复，并且跟指令发布的顺序一样。当 redis 连接在 multi 请求下，所有的命令回复都是 queued，除非这条指令的语句法不正确。而一些指令语法正确，但执行阶段出错也是允许的。

如以下

```
redis 127.0.0.1:6379> multi

OK

redis 127.0.0.1:6379> set t 13

QUEUED

redis 127.0.0.1:6379> lpop t

QUEUED

redis 127.0.0.1:6379> exec
```

1) OK

2) (error) ERR Operation against a keyholding the wrong kind of value

对于这种 err，需要客户端给予合理的提示。

需要注意的是，所有在队列中的指令都会被执行，redis 不会终止指令的执行。

取消队列指令

Discard 为取消命令队列。可以终断一个事物。不会有命令会被执行，并且连接的状态是正常的。

如:

> SET foo 1

OK

> MULTI

OK

> INCR foo

QUEUED

> DISCARD

OK

> GET foo

"1"

Optimistic locking using check-and-set(乐观锁)

watch 指令在 redis 事物中提供了 CAS 的行为。为了检测被 watch 的 keys 在是否有多个 clients 改变时引起冲突，这些 keys 将会被监控。如果至少有一个 watch 的 key 在执行 exec 命令前被修改，整个事物将会被终止，并且执行 exec 会得到 null 的回复。

例如:一个 key 自增长(假设 redis 不提供 incr 的功能)

val = GET mykey

val = val + 1

SET mykey $val

以上指令执行，如果是单一的 client，整个操作是没问题的。如果多个 client 在同一时间操作。如 client A 与 client B 读取了老的值，假如是 10，这个值在两个 client 将会被增长到 11，最后 set 这个 key 值时，这个 key 最终是 11 还不是 12.

watch 能够很好的处理这种问题:

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

使用以上代码，如果在执行 watch 与 exec 指令这段时间里有其它客户端修改此 key 值，此事物将执行失败。以上形式的锁被称为乐观锁。在大多数使用场合中，多并发将会处理不能的 keys，因为冲突不太可能。(通常没有必要重复操作)

Watch 指令说明

watch 指令是 exec 指令的执行条件:保证在执行 redis 事物操作时没有任何 client 修改被 watched 的 keys.否则事物不会执行。(注意:如果 watch 一个不稳定的 key 并且 key 过期，exec 仍然会执行这条指令),当 exec 指令被调用，所有的 keys 将是 unwatched,无论事物是否被终止。当 client 连接关闭后，所有的 keys 也会变成 unwatched.为了 flush 所有的 watched keys,也可以使用 unwatch 指令.有时间使用乐观锁锁住一些

keys 是很有用的，因为可能需要选择的 keys 需要事物操作，但是在执行读取现有的 keys 的内容后发现不需要继续执行，这时只要使用 unwatch 指令，使此连接能够继续被使用做其它新的事物操作。

ZPOP(取 sorted set 中 socre 最低的元素)指令就是通过 watch 指令实现的

WATCH zset

element = ZRANGE zset 0 0

MULTI

ZREM zset element

EXEC

事务总结

1.redis 事物实现，multi 开始，所有指令会被放入到队列中。当调用 exec 后，队列中所有指令会依次被执行。

2.multi-exec 中指令执行时，所有指令只要语法合理都会被写入队列中。队列执行时，指令有可能会执行失败，但不影响其它指令执行。

3.redis 事物提供了乐观锁，通过 watch 指令可以实现 CAS 操作。watch--multi--exec 操作

在给 key 加上乐观锁后，当在执行 exec 指令前，有其它 client 修改此 key，此事物将执行失败。从而保证原子操作。

说明：对于 redis 事物的应用其实需要灵活使用，上面介绍的例子是从官网翻译而来。其实在实际中可以通过 watch 一些标记位来保证多线程下缓存与数据库数据库的一致性。（我们的系统是分布式缓存与数据库的结合使用，缓存需要跟据数据库的一致性很重要，下面举例我们应用中的一个场景：）

如一个 service 方法，serviceA,执行 DAO 方法（1），然后更新缓存(2),两个并发线程，线程一执行了方法 1，此刻他需要把 DAO 相关的数据更新到缓存 2 中，多线程情况下，线程二在线程一执行 1 后，也同样执行 1，2 相关的操作，并且比线程一优先完成，这样将导致线程一在执行 2 时，将出现缓存数据与数

据库不一致的现象。（以上是针对单帐号的多并发操作，发生的概率还是存在），对于以上问题我们的解决方案是：

1. 帐号为 acc,为每个 acc 在缓存中增加一个 tag 标识.

2. 当线程一执行方法 1 前，设置标记位 tag.

3. 当执行方法 2 时，将会 watch tag,并且比较 tag 是否发生了修改，如果一旦发生修改，则此次缓存操作不将更新，并清空此 acc 缓存。

4. 如果 tag 值达到预期，则提交缓存更新，在提交缓存这段时间，如果 tag 发生变换，则 redisexec 提交时，会返回 null ,这样，虽然缓存内容更新成功，但跟据返回结果，可以即时清除此 acc 的缓存，从而清空了缓存的脏数据。

5. 通过以上事物保证了缓存数据与数据库数据不一致性的时间很短，甚至可以忽略，因为基本上在 MS 级别上。

6. (我们的应用在缓存数据不存在 acc 的情况下，会尝试从数据库读取，而缓存的作用只是缓解我们系统数据库的压力，这样实现，很好的达到了我们的预期效果).

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。

大量数据快速插入：

```
(cat data.txt; sleep 10) | nc localhost 6379 > /dev/null
```

```
cat data.txt | redis-cli --pipe
```

程序调用

Redis 是 c/s 架构的，如果要使用程序连接 redis，需要根据不同程序去选择相对客户端程序进行安装并配置，这样才能访问到 redis。比如 php 访问 redis 需要安装 phpredis 或 predis 客户端程序，并给 php 配置上模块，然后就可以通过 php-redis 语法去调用 redis 数据库了。

★表示该语言的推荐客户端。

ActionScript

as3redis Repository cwahlers

C

hiredis ★	Repository antirez pnoordhuis	这是官方的 C 语言客户端。支持全部的 set 命令，管道，事件驱动编程
credis	Repository	
libredis	Repository	通过 poll，ketama 哈希支持在多服务器上并行执行命令

C#

ServiceStack.Redis ★	Homepage demisbellot	这是 Miguel De Icaza 写的 C#客户端的一个增强版分支。
Booksleeve ★	Homepage marcgravell	通过堆交换实现的高性能客户端。
Sider	Homepage chakrit	.NET4.0 提供的简约客户端
TeamDev Redis Client	Repository TeamDevPerugia	基于 redis-sharp，提供基本通信功能的 redis 客户端，但是有一些不同的地方。
redis-sharp	Repository migueldeicaza	

C++

C++ Client Repository

Clojure

redis-clojure Repository ragge

Common Lisp

CL-Redis Repository Homepage BigThingist

Erlang

Erldis Repository dialtone_ japerk

Eredis Repository wooga 注重性能的 redis 客户端

Fancy

redis.fy Repository bakkdoor A Fancy Redis client library

Go

Go-Redis Repository SunOf27

Tideland RDC Repository themue

godis Repository simonz05

redis.go Repository hoisie

Haskell

redis Homepage

haskell-redis Repository Homepage old_sound 不积极维护，支持 2.0 以下版本

haXe

hxneko-redis Repository Homepage

Io

iodis Repository ichverstehe

Java

Jedis ★ Repository xetorthio

JRedis Repository Homepage SunOf27

JDBC-Redis Repository Homepage mavcunha

RJC Repository e_mzungu

Lua

redis-lua ★ Repository JoL1hAHN

lua-hiredis Repository agladysh Lua bindings for the hiredis library

Node.js

node_redis ★ Repository mranney Recommended client for node.

redis-node-client Repository fictorial 不再维护，不支持 0.3 以上版本

Objective-C

ObjCHiredis Repository loopole iOS4 设备和模拟器的静态库，为 MacOS10.5 及以上版本增加 Objective-C 框架。

Perl

Redis ★ Repository Homepage pedromelo Redis 数据库的 Perl 绑定。

Redis::hiredis Homepage neophenix C 客户端的 Perl 绑定

AnyEvent::Redis Repository Homepage miyagawa 非阻塞的 Redis 客户端

MojoX::Redis Repository Homepage und3f Mojolicious 的异步 Redis 客户端

Danga::Socket::Redis Homepage martinredmond 用 Danga::Spcket 库实现的异步 Redis 客户端。

PHP

Predis ★ Repository JoL1hAHN 成熟有支持。

phpredis ★ Repository yowgi nicolasff C 语言实现的作为 PHP 模块的客户端。

Rediska Repository Homepage shumkov

RedisServer Repository OZ PHP 中为 redis 提供独立的全功能的类。

Redisent Repository justinpoliey

Pure Data

Puredis Repository loopole 同步，异步和订阅客户端。

Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.

Python

redis-py ★ Repository andymccurdy 成熟有支持，现在看来适用于 python。

txredis Homepage dio_rian

desir Repository aallamaa

Ruby

redis-rb Repository ezmobius soveran 非常稳定和成熟的客户端。为保证最大性能安装 ★ Homepage djanowski pnoordhuis redis-rb 之前需要安装 hiredis gem。

em-redis Repository madsimian

Scala

scala-redis Repository alejandrocrosa

scala-redis ★ Repository debasishg 显然是@alejandrocrosa 之前提供的客户端的一个分支。

redis-client-scala-netty Repository

sedis Repository pk11 a thin scala wrapper for the popular Redis Java client, Jedis

Smalltalk

Smalltalk Redis Client Repository

Tcl

Tcl Client Repository antirez 在 Redis 测试套件中使用的客户端

php 调用 redis：

[root@wyzc html]# yum install httpd php php-cli php-common php-gd php-devel

Or

[root@wyzc html]# yum install nginx php php-cli php-common php-gd php-devel php-fpm

[root@wyzc html]# wget https://github.com/owlient/phpredis/archive/master.zip

[root@wyzc html]# unzip master

[root@wyzc html]# cd phpredis-master/

[root@wyzc phpredis-master]# phpize

[root@wyzc phpredis-master]# ./configure

[root@wyzc phpredis-master]# make

[root@wyzc phpredis-master]# make install

```
[root@wyzc ~]# cat   /etc/php.d/redis.ini
; Enable redis extension module
extension = redis.so


; phpredis can be used to store PHP sessions.
; To do this, uncomment and configure below
;session.save_handler = redis
;session.save_path   =   "tcp://host1:6379?weight=1,   tcp://host2:6379?weight=2&timeout=2.5,
tcp://host3:6379?weight=2"
[root@wyzc ~]#
[root@wyzc ~]# /etc/init.d/httpd   start
[root@wyzc ~]# cat /var/www/html/tt.php
<?php
$redis=new Redis();
$redis->connect('127.0.0.1',6379);
$redis->set('test','Hello www.wyzc.com !! ');
echo $redis->get('test');
?>
[root@wyzc ~]#
```

Or

Nginx 修改支持 php 然后启动 nginx php-fpm 后 写个测试页面到/usr/share/nginx/html/下

```
[root@wyzc ~]# /etc/init.d/httpd   stop
```

```
[root@wyzc ~]# /etc/init.d/nginx start

[root@wyzc ~]# /etc/init.d/php-fpm start


[root@wyzc ~]# links   -dump   http://127.0.0.1/tt.php

 Hello www.wyzc.com !!

[root@wyzc ~]#
```

redis 企业案例

- craigslist
- guardian.co.uk
- mercado Libre.com

- 
- 
- 
- 

还有很多其他的:

- Superfeedr
- Vidiowiki
- Wish Internet Consulting
- Ruby Minds
- Boxcar
- Zoombu
- Dark Curse
- OKNOtizie
- Moodstocks uses Redis as its main database by means of Ohm.
- Favstar
- Heywatch
- Sharpcloud
- Wooga for the games *"Happy Hospital"* and *"Monster World"*.
- Sina Weibo
- Engage
- PoraOra
- Leatherbound
- AuthorityLabs
- Fotolog
- TheMatchFixer
- Check-Host describes their architecture here.
- ShopSquad
- localshow.tv
- PennyAce
- Nasza Klasa
- Forrst

- Surfingbird

新浪微博：国内曾经最大的 Redis 集群

Tape is Dead，Disk is Tape，Flash is Disk，RAM Locality is King. — Jim Gray

Redis 不是比较成熟的 memcache 或者 Mysql 的替代品，是对于大型互联网类应用在架构上很好的补充。现在有越来越多的应用也在纷纷基于 Redis 做架构的改造。首先简单公布一下 Redis 平台实际情况：

- **2200+亿 commands/day 5000 亿 Read/day 500 亿 Write/day**
- **18TB+ Memory**
- **500+ Servers in 6 IDC 2000+instances**

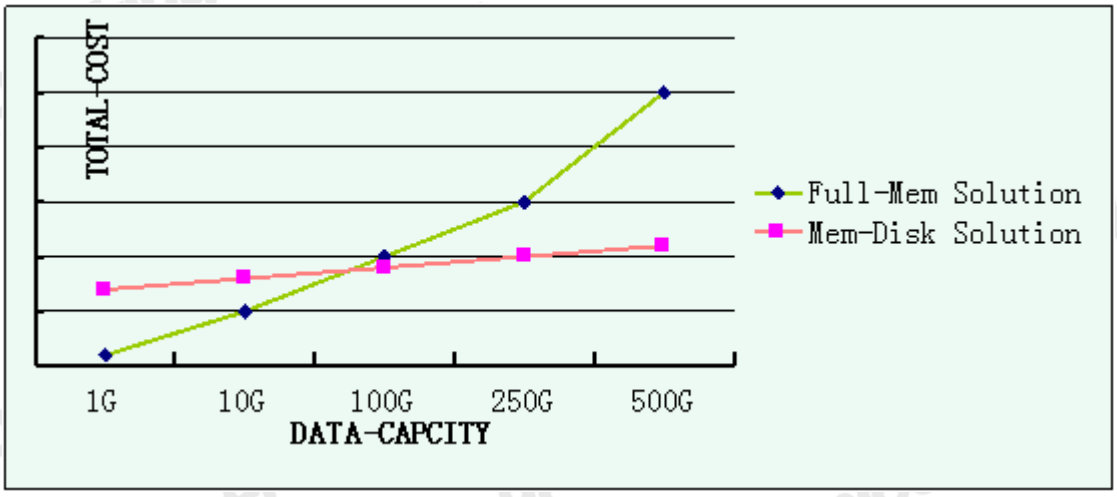
应该是国内外比较大的 Redis 使用平台，今天主要从应用角度谈谈 Redis 服务平台。

Redis 使用场景

1.Counting（计数）

计数的应用在另外一篇文章里较详细的描述，计数场景的优化 ***http://www.xdata.me/?p=262*** 这里就不多加描述了。

可以预见的是，有很多同学认为把计数全部存在内存中成本非常高，我在这里用个图表来表达下我的观点：



很多情况大家都会设想纯使用内存的方案会很有很高成本，但实际情况往往会有一些不一样：

- COST，对于有一定吞吐需求的应用来说，肯定会单独申请 DB、Cache 资源，很多担心 DB 写入性能的同学还会主动将 DB 更新记入异步队列，而这三块的资源的利用率一般都不会太高。资源算下来，你惊异的发现：反而纯内存的方案会更精简！

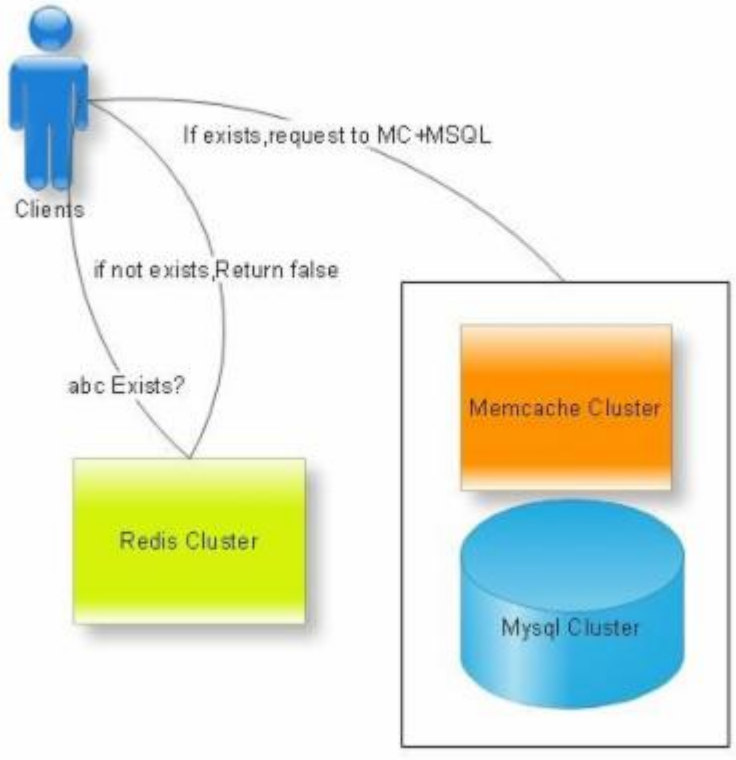
- KISS 原则，这对于开发是非常友好的，我只需要建立一套连接池，不用担心数据一致性的维护，不用维护异步队列。
- Cache 穿透风险，如果后端使用 DB，肯定不会提供很高的吞吐能力，cache 宕机如果没有妥善处理，那就悲剧了。
- 大多数的起始存储需求，容量较小。

2.Reverse cache（反向 cache）

面对微博常常出现的热点，如最近出现了较为火爆的短链，短时间有数以万计的人点击、跳转，而这里会常常涌现一些需求，比如我们向快速在跳转时判定用户等级，是否有一些账号绑定，性别爱好什么的，已给其展示不同的内容或者信息。

普通采用 memcache+Mysql 的解决方案，当调用 id 合法的情况下，可支撑较大的吞吐。但当调用 id 不可控，有较多垃圾用户调用时，由于 memcache 未有命中，会大量的穿透至 Mysql 服务器，瞬间造成连接数疯长，整体吞吐量降低，响应时间变慢。

这里我们可以用 redis 记录全量的用户判定信息，如 string key:uid int:type，做一次反向的 cache，当用户在 redis 快速获取自己等级等信息后，再去 Mc+Mysql 层去获取全量信息。如图：



当然这也不是最优化的场景，如用 Redis 做 bloomfilter，可能更加省用内存。

3.Top 10 list

产品运营总会让你展示最近、最热、点击率最高、活跃度最高等等条件的 top list。很多更新较频繁的列表如果使用 MC+MySQL 维护的话缓存失效的可能性会比较大，鉴于占用内存较小的情况，使用 Redis 做存储也是相当不错的。

4.Last Index

用户最近访问记录也是 redis list 的很好应用场景，lpush lpop 自动过期老的登陆记录，对于开发来说还是非常友好的。

5.Relation List/Message Queue

这里把两个功能放在最后，因为这两个功能在现实问题当中遇到了一些困难，但在一定阶段也确实解决了我们很多的问题，故在这里只做说明。

Message Queue 就是通过 list 的 lpop 及 lpush 接口进行队列的写入和消费，由于本身性能较好也能解决大部分问题。

6.Fast transaction with Lua

Redis 的 Lua 的功能扩展实际给 Redis 带来了更多的应用场景，你可以编写若干 command 组合作为一个小型的非阻塞事务或者更新逻辑，如：在收到 message 推送时，同时 1.给自己的增加一个未读的对话 2.给自己的私信增加一个未读消息 3.最后给发送人回执一个完成推送消息，这一层逻辑完全可以在 Redis Server 端实现。

但是，需要注意的是 Redis 会将 lua script 的全部内容记录在 aof 和传送给 slave，这也将是对磁盘，网卡一个不小的开销。

7.Instead of Memcache

1. 很多测试和应用均已证明，
2. 在性能方面 Redis 并没有落后 memcache 多少，而单线程的模型给 Redis 反而带来了很强的扩展性。
3. 在很多场景下，Redis 对同一份数据的内存开销是小于 memcache 的 slab 分配的。
4. Redis 提供的数据同步功能，其实是对 cache 的一个强有力功能扩展。

Redis 使用的重要点

1.rdb/aof Backup!

我们线上的 Redis 95%以上是承担后端存储功能的，我们不仅用作 cache，而更为一种 k-v 存储，他完全替代了后端的存储服务（MySQL），故其数据是非常重要的，如 果出现数据污染和丢失，误操作等情况，将是难以恢复的。所以备份是非常必要的！为此，我们有共享的 hdfs 资源作为我们的备份池，希望能随时可以还原业务 所需数据。

2.Small item & Small instance!

由于 Redis 单线程（严格意义上不是单线程，但认为对 request 的处理是单线程的）的模型，大的数据结构 list,sorted set,hash set 的批量处理就意味着其他请求的等待，故使用 Redis 的复杂数据结构一定要控制其单 key-struct 的大小。

另外，Redis 单实例的内存容量也应该有严格的限制。单实例内存容量较大后，直接带来的问题就是故障恢复或者 Rebuild 从库的时候时间较长，而更糟糕的是，Redis rewrite aof 和 save rdb 时，将会带来非常大且长的系统压力，并占用额外内存，很可能导致系统内存不足等严重影响性能的线上故障。我们线上 96G/128G 内存服务器不建议单实例容量大于 20/30G。

3.Been Available!

业界资料和使用比较多的是 Redis sentinel（哨兵）

http://www.huangz.me/en/latest/storage/redis_code_analysis/sentinel.html

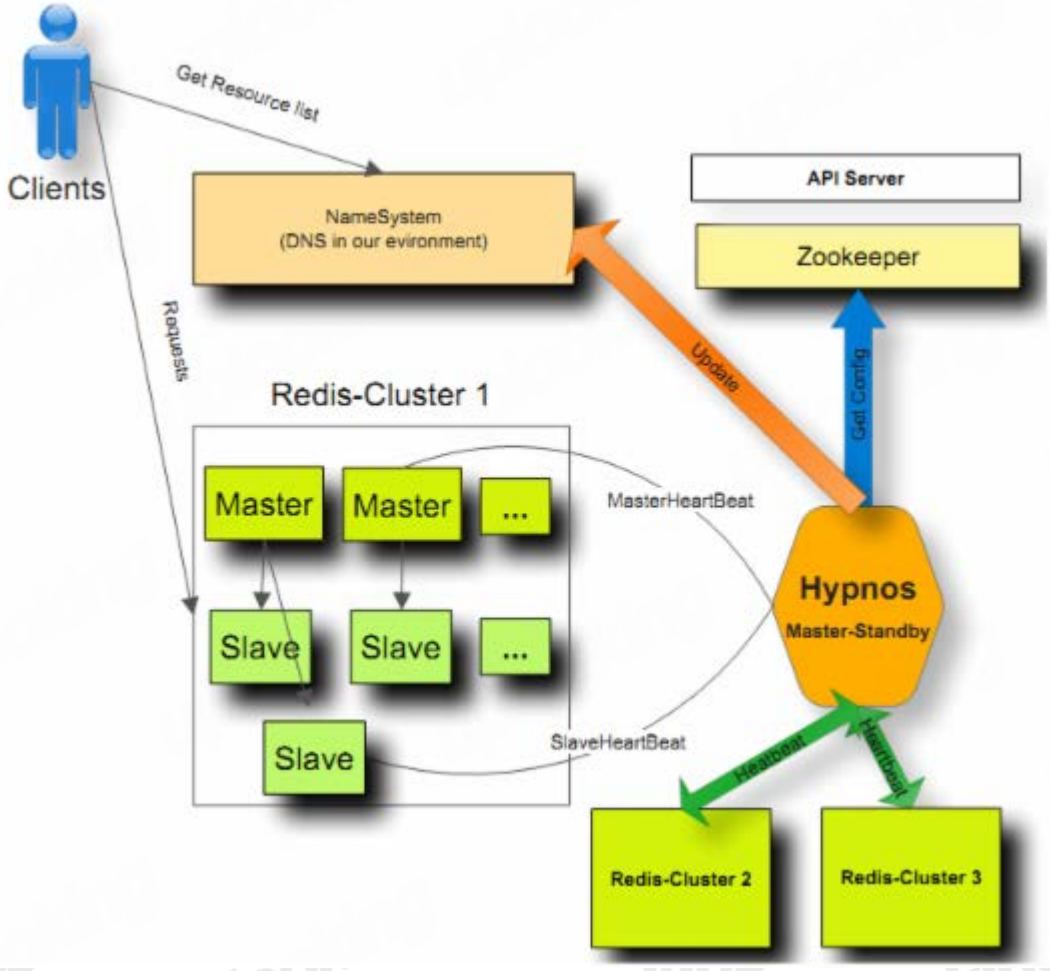
http://qiita.com/wellflat/items/8935016fdee25d4866d9

2000 行 C 实现了服务器状态检测，自动故障转移等功能。

但由于自身实际架构往往会复杂，或者考虑的角度比较多，为此 *@许琦 eryk* 和我一同做了 hypnos 项目。

hypnos 是神话中的睡神，字面意思也是希望我们工程师无需在休息时间处理任何故障。:-)

其工作原理示意如下：



Talk is cheap, show me your code! 稍后将单独写篇博客细致讲下 Hypnos 的实现。

4.In Memory or not?

发现一种情况，开发在沟通后端资源设计的时候，常常因为习惯使用和错误了解产品定位等原因，而忽视了对真实使用用户的评估。也许这是一份历史数据，只有最近一天的数据才有人进行访问，而把历史数据的容量和最近一天请求量都抛给内存类的存储现实是非常不合理的。

所以当你在究竟使用什么样的数据结构存储的时候，请务必先进行成本衡量，有多少数据是需要存储在内存中的？有多少数据是对用户真正有意义的。因为这其实对后端资源的设计是至关重要的，1G 的数据容量和 1T 的数据容量对于设计思路是完全不一样的

Plans in future?

1.slave sync 改造

全部改造线上 master-slave 数据同步机制，这一点我们借鉴了 MySQL Replication 的思路，使用 rdb+aof+pos 作为数据同步的依据，这里简要说明为什么官方提供的 psync 没有很好的满足我们的需求：

假设 A 有两个从库 B 及 C，及 A `— B&C，这时我们发现 master A 服务器有宕机隐患需要重启或者 A 节点直接宕机，需要切换 B 为新的主库，如果 A、B、C 不共享 rdb 及 aof 信息，C 在作为 B 的从库时，仍会清除自身数 据，因为 C 节点只记录了和 A 节点的同步状况。

故我们需要有一种将 A`–B&C 结构切换切换为 A`–B`–C 结构的同步机制，psync 虽然支持断点续传，但仍无法支持 master 故障的平滑切换。

实际上我们已经在我们定制的 Redis 计数服务上使用了如上功能的同步，效果非常好，解决了运维负担，但仍需向所有 Redis 服务推广，如果可能我们也会向官方 Redis 提出相关 sync slave 的改进。

2.更适合 redis 的 name-system Or proxy

细心的同学发现我们除了使用 DNS 作为命名系统，也在 zookeeper 中有一份记录，为什么不让用户直接访问一个系统，zk 或者 DNS 选择其一呢？

其实还是很简单，命名系统是个非常重要的组件，而 dns 是一套比较完善的命名系统，我们为此做了很多改进和试错，zk 的实现还是相对复杂，我们还没有较强的把控粒度。我们也在思考用什么做命名系统更符合我们需求。

3.后端数据存储

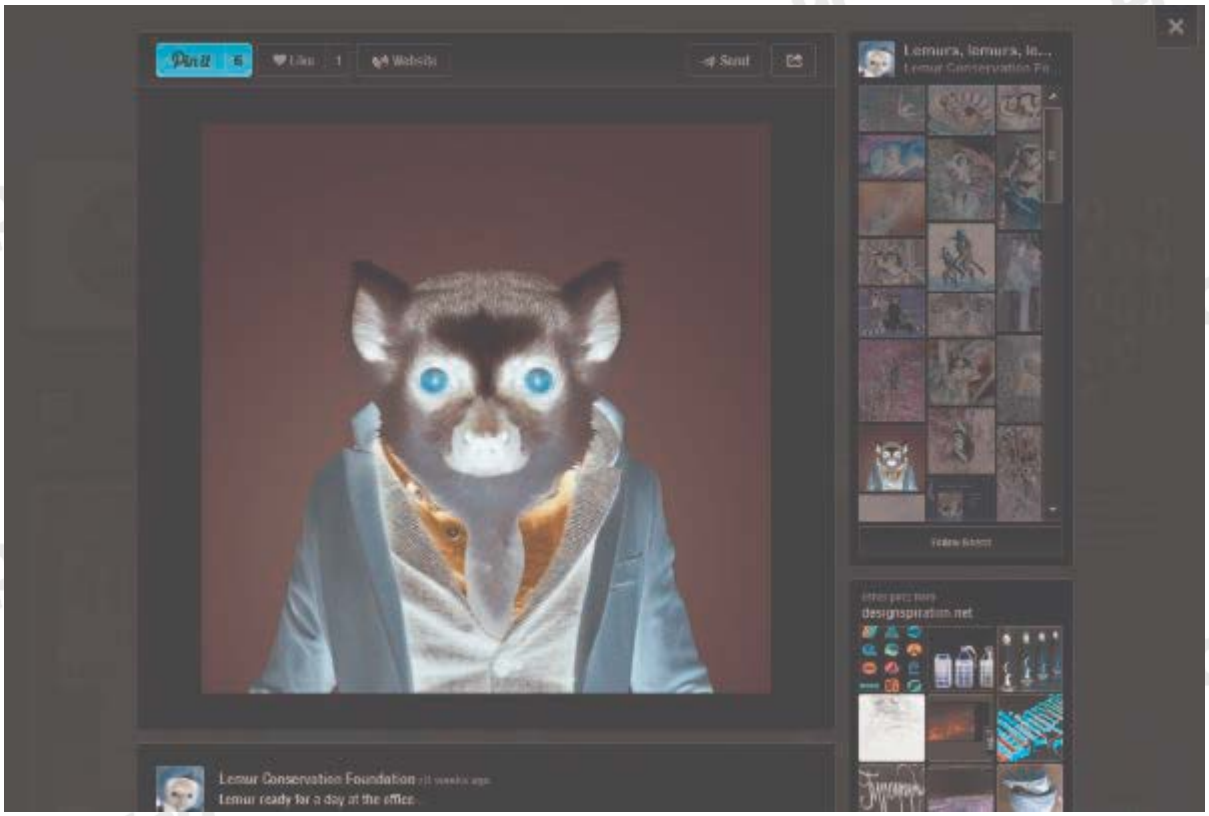
大内存的使用肯定是一个重要的成本优化方向，flash 盘及分布式的存储也在我们未来计划之中。（原文链接： *Largest Redis Clusters Ever* ）

Pinterest：Reids 维护上百亿的相关性

Pinterest 已经成为硅谷最疯故事之一，在 2012 年，他们基于 PC 的业务增加 1047%，移动端采用增加 1698%， *该年 3 月其独立访问数量更飙升至 533 亿* 。在 Pinterest，人们关注的事物以百亿记——每个用户界面都会查询某个 board 或者是用户是否关注的行为促成了异常复杂的工程问题。这也让 Redis 获得了用武之地。经过数年的发展，Pinterest 已经成为媒体、社交等多个领域的佼佼者，其辉煌战绩如下：

- 获得的推荐流量高于 Google+、YouTube 及 LinkedIn 三者的总和
- 与 Facebook 及 Twitter 一起成为最流行的三大社交网络
- 参考 Pinterest 进行购买的用户比其它网站更高（ *更多详情* ）

如您所想，基于其独立访问数，Pinterest 的高规模促成了一个非常高的 IT 基础设施需求。



通过缓存来优化用户体验

近日，Pinterest 工程经理 Abhi Khune 对其公司的用户体验需求及 Redis 的使用经验 *进行了分享* 。即使是滋生的应用程序打造者，在分析网站的细节之前也不会理解这些特性，因此先大致的理解一下使用场景：首先，为每个粉丝进行提及到的预检查；其次，UI 将准确的显示用户的粉丝及关注列表分页。高效的执行这些操作，每次点击都需要非常高的性能架构。

不能免俗，Pinterest 的软件工程师及架构师已经使用了 MySQL 及 memcache，但是缓存解决方案仍然达到了他们的瓶颈；因此 为了拥有更好的用户体验，缓存必须被扩充。而在实际操作过程中，工程团队已然发现缓存只有当用户 sub-graph 已经在缓存中时才会起到作用。因此。任 何使用这个系统的人都需要被缓存，这就导致了整个图的缓存。同时，最常见的查询“用户 A 是否关注了用户 B”的答案经常是否定的，然而这却被作为了缓存丢 失，从而促成一个数据库查询，因此他们需要一个新的方法来扩展缓存。最终，他们团队决定使用 Redis 来存储整个图，用以服务众多的列表。

使用 Redis 存储大量的 Pinterest 列表

Pinterest 使用了 Redis 作为解决方案，并将性能推至了内存数据库等级，为用户保存多种类型列表：

- 关注者列表

- 你所关注的 board 列表
- 粉丝列表
- 关注你 board 的用户列表
- 某个用户中 board 中你没有关注的列表
- 每个 board 的关注者及非关注者

Redis 为其 7000 万用户存储了以上的所有列表，本质上讲可以说是储存了所有粉丝图，通过用户 ID 分片。鉴于你可以通过类型来查看以上 列表的数据，分析概要信息被用看起来更像事务的系统储存及访问。Pinterest 当下的用户 like 被限制为 10 万，初略进行统计：如果每个用户关注 25 个 board，将会在用户及 board 间产生 17.5 亿的关系。同时更加重要的是，这些关系随着系统的使用每天都会增加。

Pinterest 的 Reids 架构及运营

通过 Pinterest 的一个创始人了解到，Pinterest 开始使用 Python 及订制的 Django 编写应用程序，并一直持续到其拥 有 1800 万用户级日 410TB 用户数据的时候。虽然使用了多个存储对数据进行储存，工程师根据用户 id 使用了 8192 个虚拟分片，每个分片都运行在一个 Redis DB 之上，同时 1 个 Redis 实例将运行多个 Redis DB。为了对 CPU 核心的充分使用，同一台主机上同时使用多线程和单线程 Redis 实例。

鉴于整个数据集运行在内存当中，Redis 在 Amazon EBS 上对每秒传输进来的写入都会进行持久化。扩展主要通过两个方面进行：第 一，保持 50%的利用率，通过主从转换，机器上运行的 Redis 实例一半会转译到一个新机器上；第二，扩展节点和分片。整个 Redis 集群都会使用一个主 从配置，从部分将被当做一个热备份。一旦主节点失败，从部分会立刻完成主的转换，同时一个新的从部分将会被添加，ZooKeeper 将完成整个过程。同时 他们每个小时都会在 Amazon S3 上运行 BGsave 做更持久的储存——这项 Reids 操作会在后端进行，之后 Pinterest 会使用这些数据做 MapReduce 和分析作业。（更多内容见原文）

Viacom：Redis 在系统中的用例盘点

Viacom 是全球最大的传媒集体之一，同时也遭遇了当下最大的数据难题之一：如何处理日益剧增的动态视频内容。

着眼这一挑战的上升趋势，我们会发现： *2010 年世界上所有数据体积达到 ZB 级* ，而单单 2012 这一年，互联网产生的数据就增加了 2.8 个 ZB，其中大部分的数据都是非结构化的，包括了视频和图片。

覆盖 MVN（以前称为 MTV Networks、Paramount 及 BET），Viacom 是个名副其实的传媒巨头，支持众多人气站点， 其中包括 The Daily Show、osh.0、South Park Studios、GameTrailers.com 等。作为媒体公司，这些网 站上的文档、图片、视频短片都在无时无刻的更新。长话短说，下面就进入 Viacom 高级架构师 Michael Venezia 分享的 Redis 实践：

Viacom 的网站架构背景

对于 Viacom，横跨多个站点传播内容让必须专注于规模的需求，同时为了将内容竟可能快的传播到相应用户，他们还必须聚焦内容之间的关 系。然而即使 The Daily Show、Nickelodeon、Spike 或者是 VH1 这些单独的网站上，日平均 PV 都可以达到千万，峰值时流量 更会达到平均值的 20-30 倍。同时基于对实时的需求，动态的规模及速度已成为架构的基础之一。

除去动态规模之外，服务还必须基于用户正在浏览的视频或者是地理位置来推测用户的喜好。比如说，某个页面可能会将一个独立的视频片段与本地 的促销，视频系列的额外部分，甚至是相关视频联系起来。为了能让用户能在网站上停留更长的时间，他们建立了一个能基于详细元数据自动建立页面的软件引擎， 这个引擎可以根据用户当下兴趣推荐额外的内容。鉴于用于兴趣的随时改变，数据的类型非常广泛——类似 graph-like，实际上做的是大量的 join。

这样做有利于减少类似视频的大体积文件副本数，比如数据存储中一个独立的记录是 Southpark 片段 “Cartman gets an Anal Probe”，这个片段可能也会出现在德语的网站上。虽然视频是一样的，但是英语用户搜索的可能就是另一个 不同的词语。元数据的副本转换成搜索结果，并指向相同的视频。因此在美国用户搜索真实标题的情况下，德国浏览者可能会使用转译的标题——德国网站上的 “Cartman und die Analsonde”。

这些元数据覆盖了其它记录或者是对象，同时还可以根据使用环境来改变内容，通过不同的规则集来限制不同地理位置或者是设备请求的内容。

Viacom 的实现方法

尽管许多机构通过使用 ORM 及传统关系型数据库来解决这个问题，Viacom 却使用了一个迥然不同的方法。

本质上，他们完全承担不了对数据库的直接访问。首先，他们处理的大部分都是流数据，他们偏向于使用 Akamai 从地理上来分配内容。其次， 基于页面的复杂性可能会取上万个对象。取如此多的数据显然会影响到性能，因此 JSON 在 1 个数据服务中投入了使用。当然，这些 JSON 对象的缓存将直接影 响到网站性能。同时，当内容或者是内容之间的关系发生改变时，缓存还需要动态的进行更新。

Viacom 依靠对象基元和超类解决这个问题，继续以 South Park 为例：一个私有的“episode”类包含了所有该片段相关信 息，一个“super object”将有助于发现实际的视频对象。超类这个思想确实非常有益于建设低延迟页面的自动建设，这些超类可以帮助到基元对象到 缓存的映射及保存。

Viacom 为什么要使用 Redis

每当 Viacom 上传一个视频片段，系统将建立一个私有的对象，并于 1 个超类关联。每一次修改，他们都需要重估私有对象的每个改变，并更新 所有复合对象。同时，系统还需要无

效 Akamail 中的 URL 请求。系统现有架构的组合及更敏捷的管理方法需求将 Viacom 推向了 Redis。

基于 Viacom 主要基于 PHP，所以这个解决方案必须支持 PHP。他们首先选择了 memcached 做对象存储，但是它并不能很好的支持 hashmap；同时他们还需要一个更有效的进行无效步骤的重估，即更好的理解内容的依赖性。本质上说，他们需要时刻跟进无效步骤中的依赖性改变。因此他 们选择了 Redis 及 Predis 的组合来解决这个问题。

他们团队使用 Redis 给 southparkstudios.com 和 thedailyshow.com 两个网站建设依赖性图，在取得了很大的成功后他们开始着眼 Redis 其它适合场景。

Redis 的其它使用场景

显而易见，如果有人使用 Redis 来建设依赖性图，那么使用它来做对象处理也是说得通的。同样，这也成了架构团队为 Redis 选择的第二使 用场景。Redis 的复制及持久化特性同时也征服了 Viacom 的运营团队，因此在几个开发周期后，Redis 成为他们网站的主要数据及依赖性储存。

后两个用例则是行为追踪及浏览计数的缓冲，改变后的架构是 Redis 每几分钟向 MySQL 中储存一次，而浏览计数则通过 Redis 进行存储 及计数。同时 Redis 还被用来做人气的计算，一个基于访问数及访问时间的得分系统——如果某个视频最近被访问的次数越多，它的人气就越高。在如此多内容 上每隔 10-15 分钟做一次计算绝对不是类似 MySQL 这样传统关系型数据库的强项，Viacom 使用 Redis 的理由也非常简单——在 1 个存储浏览信息 的 Redis 实例上运行 Lua 批处理作业，计算出所有的得分表。信息被拷贝到另一个 Redis 实例上，用以支持相关的产品查询。同时还在 MySQL 上做了 另一个备份，用以以后的分析，这种组合会将这个过程耗费的时间降低 60 倍。

Viacom 还使用 Redis 存储一步作业信息，这些信息被插入一个列表中，工作人员则使用 ***BLPOP*** 命令行在队列中抓取顶端的任务。同时 zsets 被用于从众多社交网络（比如 Twitter 及 Tumblr）上综合内容，Viacom 通过 Brightcove 视频播放器来同步多个内容管理系统。

横跨这些用例，几乎所有的 Redis 命令都被使用——sets、lists、zlists、hashmaps、scripts、counters 等。同时，Redis 也成为 Viacom 可扩展架构中不可或缺的一环。

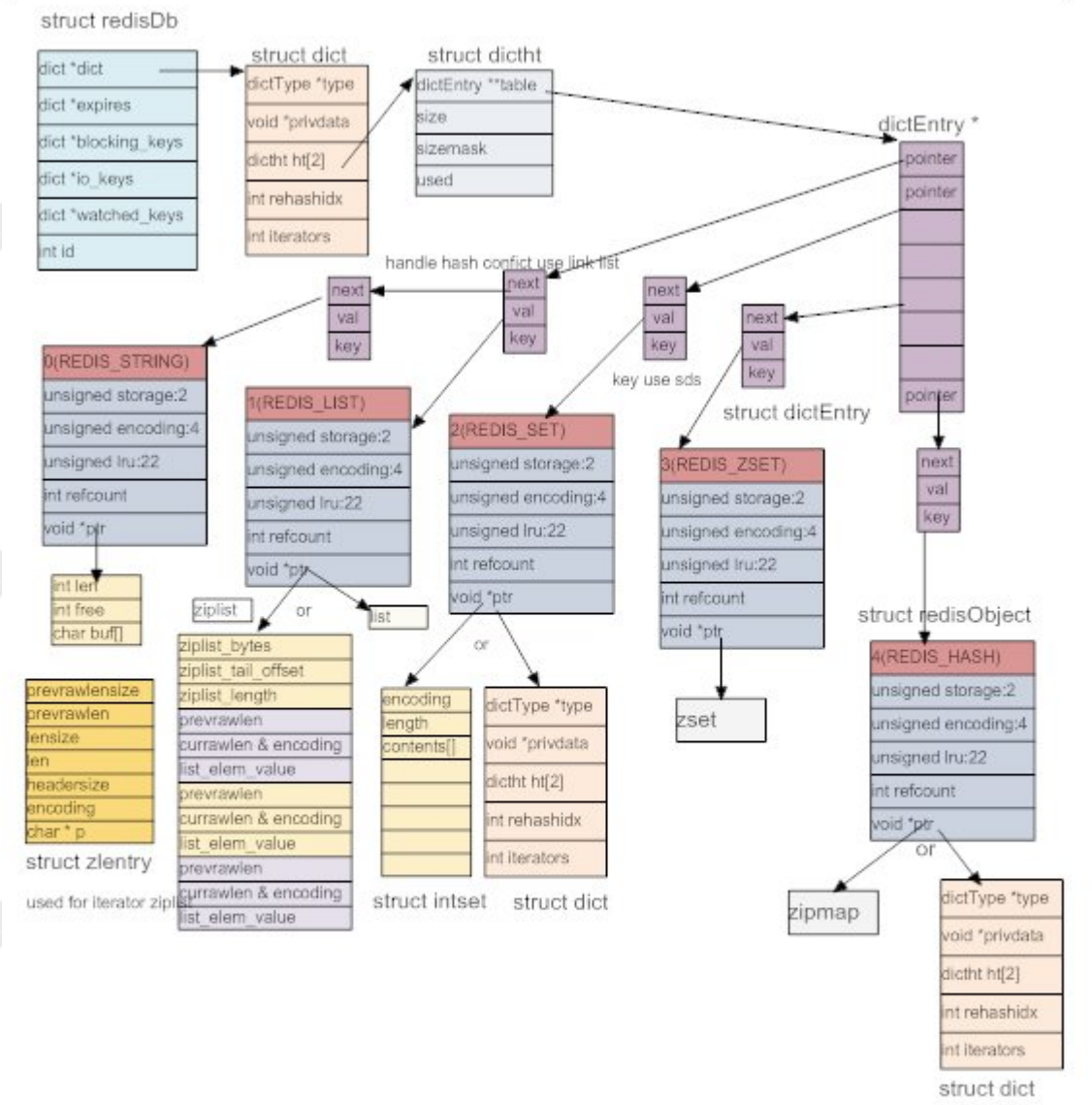
附录：内存结构分析：

Redis 内存存储结构

本文是基于 Redis-v2.2.4 版本进行分析.

1.1 Redis 内存存储总体结构

Redis 是支持多 key-value 数据库(表)的,并用 RedisDb 来表示一个 key-value 数据库(表). redisServer 中有一个 redisDb *db; 成员变量, RedisServer 在初始化时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是 redisDb 数组的第 1 个(即下标是 0) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的. 下面会详细介绍一下 redisDb 的内存结构.



redis 的内存存储结构示意图

redis 的内存存储结构示意图

redisDb 的定义:

```
1 typedef struct redisDb
2
3 {
4
5 dict *dict;                    /* The keyspace for this DB */
6
7 dict *expires;                 /* Timeout of keys with a timeout set */
```

```
8
9 dict *blocking_keys;    /* Keys with clients waiting for data (BLPOP)
10*/
11
12dict *io_keys;              /* Keys with clients waiting for VM I/O
13*/
14
15dict *watched_keys;         /* WATCHED keys for MULTI/EXEC CAS */
16
17int id;
18
19} redisDb;
```

struct redisDb 中 ,dict 成员是与实际存储数据相关的. dict 的定义如下:

```
1 typedef struct dictEntry
2
3 {
4
5 void *key;
6
7 void *val;
8
9 struct dictEntry *next;
10
11} dictEntry;
12
13typedef struct dictType
14
15{
16
17unsigned int (*hashFunction)(const void *key);
18
19void *(*keyDup)(void *privdata, const void *key);
20
21void *(*valDup)(void *privdata, const void *obj);
22
23int (*keyCompare)(void *privdata, const void *key1, const void *key2);
24
25void (*keyDestructor)(void *privdata, void *key);
26
27void (*valDestructor)(void *privdata, void *obj);
28
29} dictType;
30
```

```
/* This is our hash table structure. Every dictionary has two of this as we
* implement incremental rehashing, for the old to the new table. */
typedef struct dictht
{
dictEntry **table;
unsigned long size;
unsigned long sizemask;
unsigned long used;
} dictht;
typedef struct dict
{
dictType *type;
void *privdata;
dictht ht[2];
int rehashidx; /* rehashing not in progress if rehashidx == -1 */
int iterators; /* number of iterators currently running */
 } dict;
```

dict 是主要是由 struct dictht 的 哈唏表构成的, 之所以定义成长度为 2 的(dictht ht[2]) 哈唏表数组,是因为 redis 采用渐进的 rehash,即当需要 rehash 时,每次像 hset,hget 等操作前,先执行 N 步 rehash. 这样就把原来一次性的 rehash 过程拆散到进行, 防止一次性 rehash 期间 redis 服务能力大幅下降. 这种渐进的 rehash 需要一个额外的 struct dictht 结构来保存.

struct dictht 主要是由一个 struct dictEntry 指针数组组成的, hash 表的冲突是通过链表法来解决的.

struct dictEntry 中的 key 指针指向用 sds 类型表示的 key 字符串, val 指针指向一个 struct redisObject 结构体, 其定义如下:

```
typedef struct redisObject
{
```

```
5 unsigned type:4;
6
7 unsigned storage:2;   /* REDIS_VM_MEMORY or REDIS_VM_SWAPPING */
8
9 unsigned encoding:4;
10
11unsigned lru:22;        /* lru time (relative to server.lruclock) */
12
13int refcount;
14
15void *ptr;
16
17/* VM fields are only allocated if VM is active, otherwise the
18
19* object allocation function will just allocate
20
21* sizeof(redisObjct) minus sizeof(redisObjectVM), so using
22
23* Redis without VM active will not have any overhead. */
24
25} robj;
26
27
28
29//type 占 4 bit,用来表示 key-value 中 value 值的类型,目前 redis 支持:
30
31string, list, set,zset,hash 5 种类型的值.
32
33/* Object types */
34
35#define REDIS_STRING 0
36
37#define REDIS_LIST 1
38
39#define REDIS_SET 2
40
41#define REDIS_ZSET 3
42
43#define REDIS_HASH 4
44
45#define REDIS_VMPOINTER 8
46
47// storage 占 2 bit ,表示 此值是在 内存中,还是 swap 在硬盘上.
48
49// encoding 占 4 bit ,表示值的编码类型,目前有 8 种类型:
```

```
50
51/* Objects encoding. Some kind of objects like Strings and Hashes can
52be
53
54* internally represented in multiple ways. The 'encoding' field of
55the object
56
57* is set to one of this fields for this object. */
58
59#define REDIS_ENCODING_RAW 0     /* Raw representation */
60
61#define REDIS_ENCODING_INT 1     /* Encoded as integer */
62
63#define REDIS_ENCODING_HT 2      /* Encoded as hash table */
64
65#define REDIS_ENCODING_ZIPMAP 3  /* Encoded as zipmap */
66
67#define REDIS_ENCODING_LINKEDLIST 4 /* Encoded as regular linked list
68*/
69
70#define REDIS_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
71
72#define REDIS_ENCODING_INTSET 6  /* Encoded as intset */
73
74#define REDIS_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
75
76
77/* 如 type 是 REDIS_STRING 类型的,则其值如果是数字,就可以编码成
78
79REDIS_ENCODING_INT,以节约内存.
80
81* 如 type 是 REDIS_HASH 类型的,如果其 entry 小于配置值:
82
83hash-max-zipmap-entries 或 value 字符串的长度小于
  hash-max-zipmap-value, 则可以编码成 REDIS_ENCODING_ZIPMAP 类型存储,以
  节约内存. 否则采用 Dict 来存储.

  * 如 type 是 REDIS_LIST 类型的,如果其 entry 小于配置值:
  list-max-ziplist-entries 或 value 字符串的长度小于
```

list-max-ziplist-value, 则可以编码成 REDIS_ENCODING_ZIPLIST 类型存储, 以节约内存; 否则采用 REDIS_ENCODING_LINKEDLIST 来存储.

* 如 type 是 REDIS_SET 类型的,如果其值可以表示成数字类型且 entry 小于配置值 set-max-intset-entries, 则可以编码成 REDIS_ENCODING_INTSET 类型存储,以节约内存; 否则采用 Dict 类型来存储.

* lru: 是时间戳

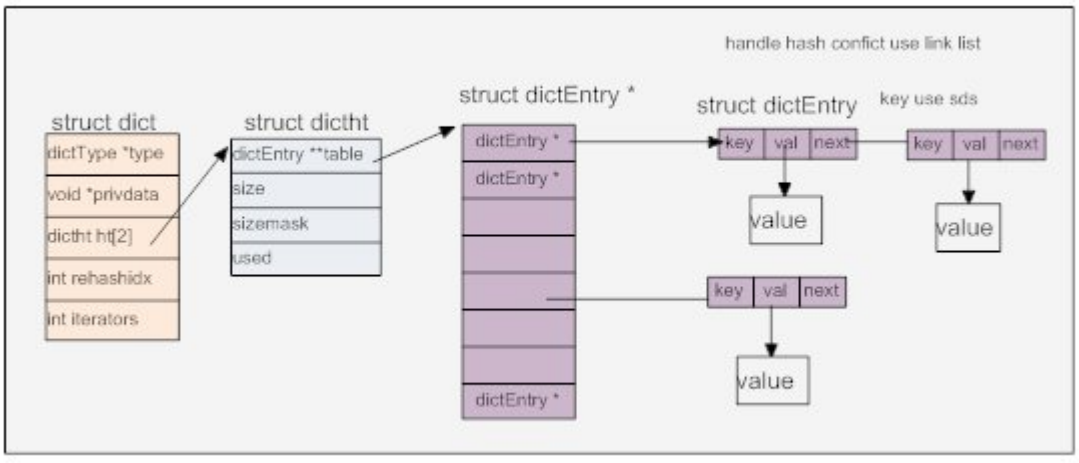
* refcount: 引用次数

* void * ptr : 指向实际存储的 value 值内存块,其类型可以是 string, set, zset,list,hash ,编码方式可以是上述 encoding 表示的一种.

* 至于一个 key 的 value 采用哪种类型来保存,完全是由客户端的指令来决定的,如 hset ,则值是采用 REDIS_HASH 类型表示的,至于那种编码(encoding),则由 redis 根据配置自动决定.

*/

1.2 Dict 结构

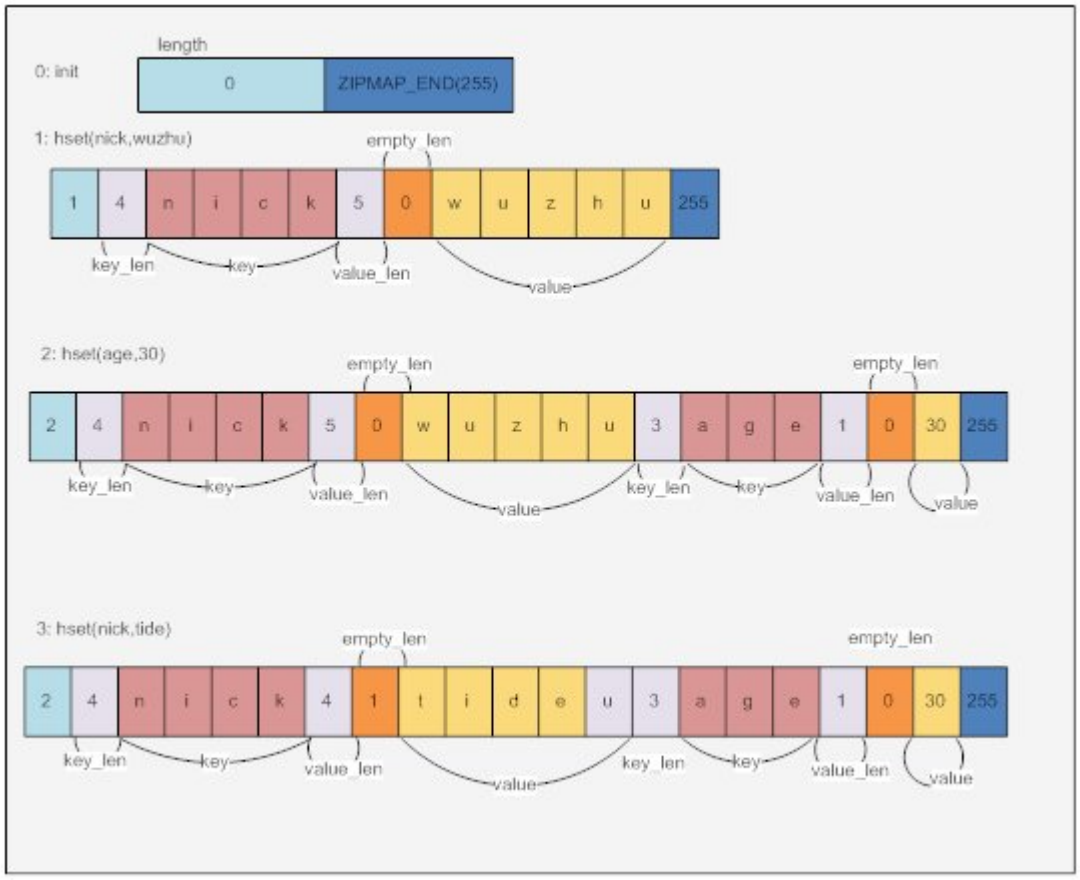


Dict 结构在<1.1Redis 内存存储结构>; 已经描述过了,这里不再赘述.

1.3 zipmap 结构

如果 redisObject 的 type 成员值是 REDIS_HASH 类型的,则当该 hash 的 entry 小于配置值: hash-max-zipmap-entries 或者 value 字符串的长度小于 hash-max-zipmap-value, 则可以编码成 REDIS_ENCODING_ZIPMAP 类型存储,以节约内存. 否则采用 Dict 来存储.

zipmap 其实质是用一个字符串数组来依次保存key和value,查询时是依次遍列每个 key-value 对,直到查到为止. 其结构示意图如下:



为了节约内存,这里使用了一些小技巧来保存 key 和 value 的长度. 如果 key 或 value 的长度小于 ZIPMAP_BIGLEN(254),则用一个字节来表示,如果大于 ZIPMAP_BIGLEN(254),则用 5 个字节保存,第一个字节为保存 ZIPMAP_BIGLEN(254),后面 4 个字节保存 key 或 value 的长度.

初始化时只有 2 个字节,第 1 个字节表示 zipmap 保存的 key-value 对的个数(如果 key-value 对的个数超过 254,则一直用 254 来表示, zipmap 中实际保存的 key-value 对个数可以通过 zipmapLen() 函数计算得到).

hset(nick,wuzhu) 后,

第 1 个字节保存 key-value 对(即 zipmap 的 entry 数量)的数量 1

第 2 个字节保存 key_len 值 4

第 3~6 保存 key “nick”

第 7 字节保存 value_len 值 5

第 8 字节保存空闭的字节数 0 (当 该 key 的值被重置时,其新值的长度与旧值的长度不一定相等,如果新值长度比旧值的长度大,则 realloc 扩大内存; 如果新值长度比旧值的长度小,且相差大于 4 bytes ,则 realloc 缩小内存,如果相差小于 4,则将值往前移,并用 empty_len 保存空闲的 byte 数)

第 9~13 字节保存 value 值 “wuzhu”

hset(age,30)

插入 key-value 对 (“age”,30)

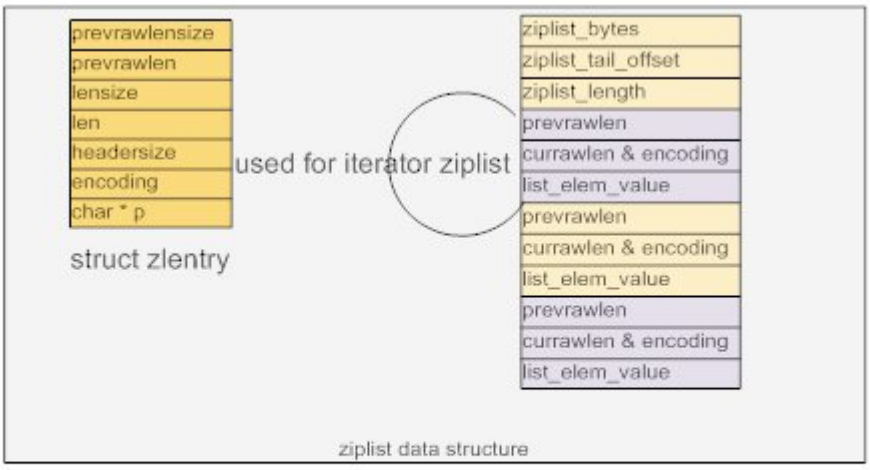
hset(nick,tide)

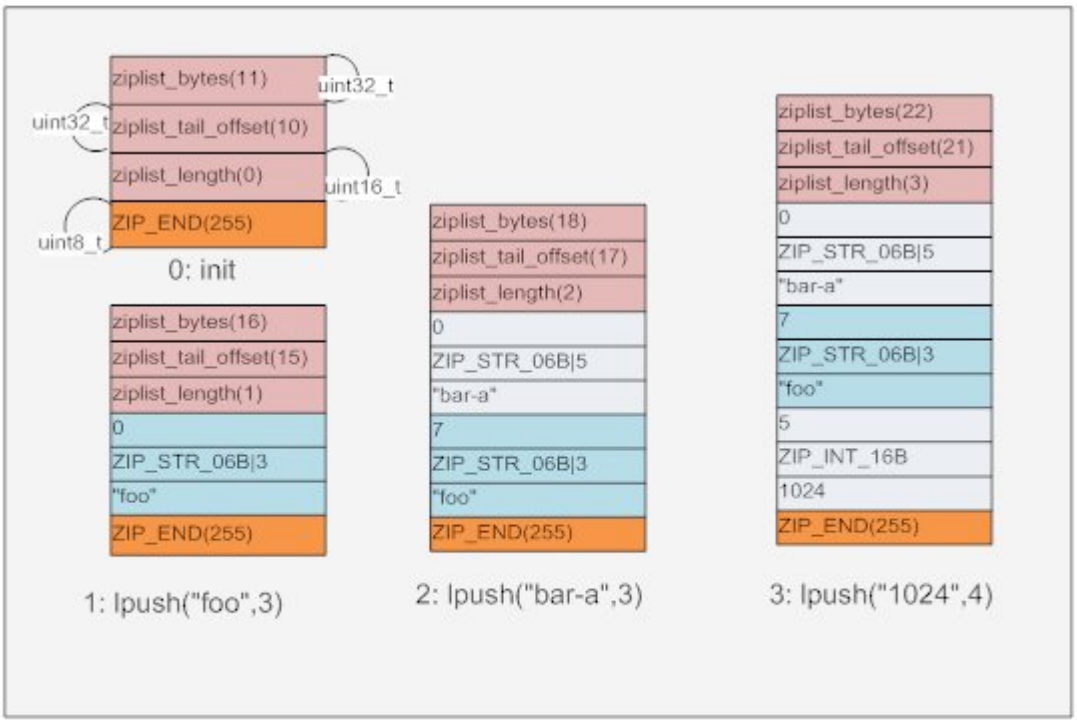
插入 key-value 对 (“nick”,”tide”), 后可以看到 empty_len 为 1,

1.4 ziplist 结构

如果 redisObject 的 type 成员值是 REDIS_LIST 类型的,则当该 list 的 elem 数小于配置值: hash-max-ziplist-entries 或者 elem_value 字符串的长度小于 hash-max-ziplist-value, 则可以编码成 REDIS_ENCODING_ZIPLIST 类型存储,以节约内存. 否则采用 list 来存储.

ziplist 其实质是用一个字符串数组形式的双向链表. 其结构示意图如下:





ziplist header 由 3 个字段组成:

ziplist_bytes: 用一个 uint32_t 来保存, 构成 ziplist 的字符串数组的总长度,包括 ziplist header,

ziplist_tail_offset: 用一个 uint32_t 来保存,记录 ziplist 的尾部偏移位置.

ziplist_length: 用一个 uint16_t 来保存,记录 ziplist 中 elem 的个数

ziplist node 也由 3 部分组成:

prevrawlen: 保存上一个 ziplist node 的占用的字节数,包括: 保存 prevarwlen,currawlen 的字节数和 elem value 的字节数.

currawlen&encoding: 当前 elem value 的 raw 形式存款所需的字节数及在 ziplist 中保存时的编码方式(例如,值可以转换成整数,如示意图中的”1024”, raw_len 是 4 字节,但在 ziplist 保存时转换成 uint16_t 来保存,占 2 个字节).

(编码后的)value

可以通过 prevrawlen 和 currawlen&encoding 来遍列 ziplist.

ziplist 还能到一些小技巧来节约内存.

len 的存储: 如果 len 小于 ZIP_BIGLEN(254),则用一个字节来保存; 否则需要 5 个字节来保存,第 1 个字节存 ZIP_BIGLEN,作为标识符.

value 的存储: 如果 value 是数字类型的,则根据其值的范围转换成 ZIP_INT_16B, ZIP_INT_32B 或 ZIP_INT_64B 来保存,否则用 raw 形式保存.

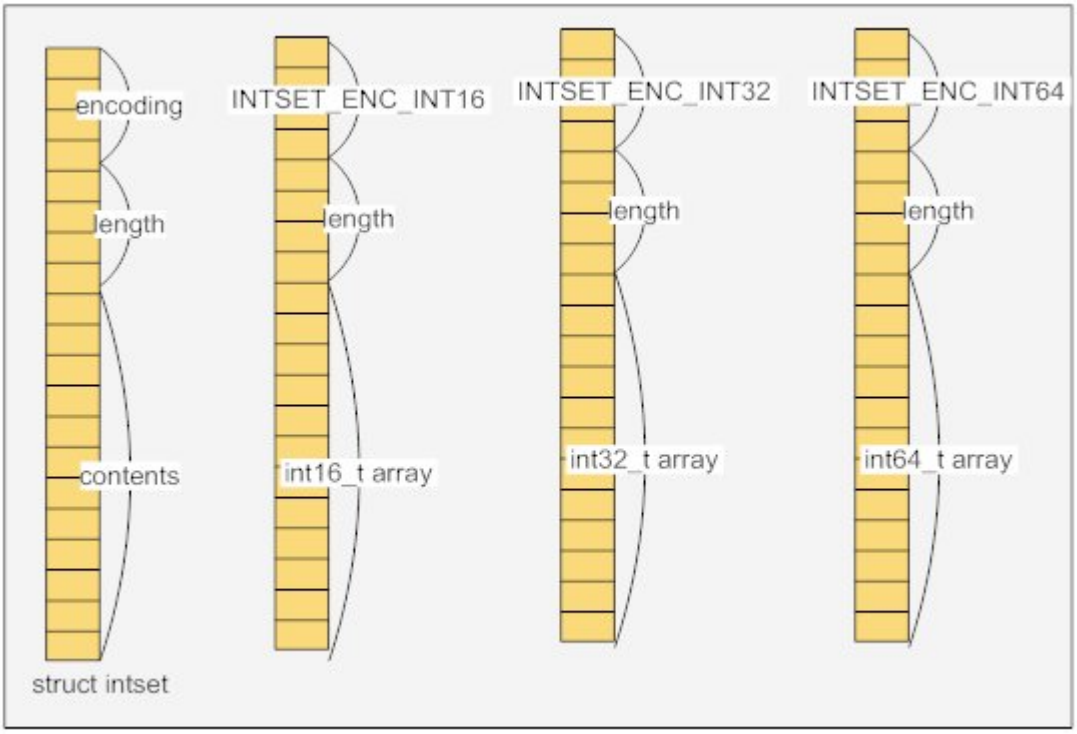
1.5 adlist 结构

```
1 typedef struct listNode
2 {
3
4 struct listNode *prev;
5
6 struct listNode *next;
7
8 void *value;
9
10} listNode;
11
12typedef struct listIter
13
14{
15
16listNode *next;
17
18int direction;
19
20} listIter;
21
22typedef struct list
23
24{
25
26listNode *head;
27
28listNode *tail;
29
30void *(*dup)(void *ptr);
31
32void (*free)(void *ptr);
33
34int (*match)(void *ptr, void *key);
```

```
35
36unsigned int len;
37
38} list;
```

常见的双向链表,不作分析.

1.6 intset 结构



intset 是用一个有序的整数数组来实现集合(set). struct intset 的定义如下:

```
1 typedef struct intset
2
3 {
4
5 uint32_t encoding;
6
7 uint32_t length;
8
9 int8_t contents[];
10
11} intset;
```

encoding: 来标识数组是 int16_t 类型, int32_t 类型还是 int64_t 类型的数组. 至于怎么先择是那种类型的数组,是根据其保存的值的取值范围来决定的,初始化时是 int16_t, 根据 set 中的最大值在 [INT16_MIN, INT16_MAX] , [INT32_MIN, INT32_MAX], [INT64_MIN, INT64_MAX]的那个取值范围来动态确定整个数组的类型. 例如 set 一开始是 int16_t 类型,当一个取值范围在 [INT32_MIN, INT32_MAX]的值加入到 set 时,则将保存 set 的数组升级成 int32_t 的数组.

length: 表示 set 中值的个数

contents: 指向整数数组的指针

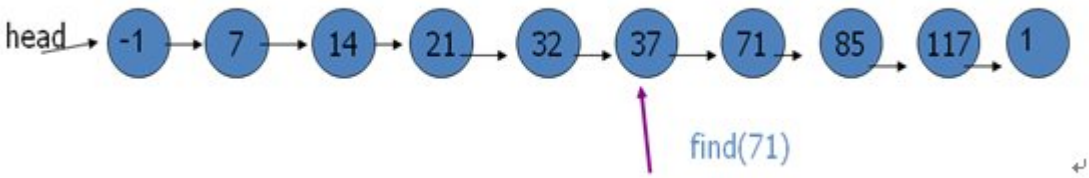
1.7 zset 结构

首先，介绍一下 skip list 的概念，然后再分析 zset 的实现.

1.7.1 Skip List 介绍

1.7.1.1 有序链表

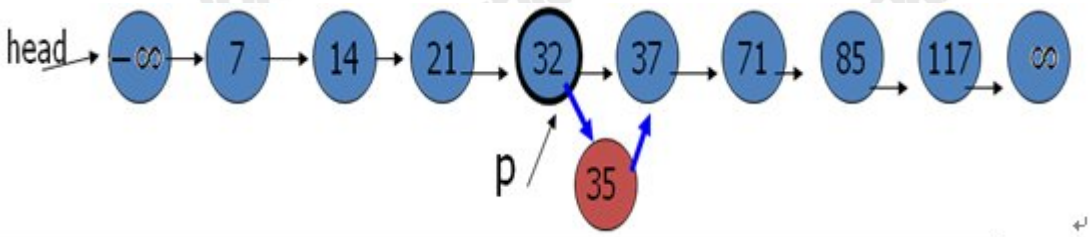
1) Searching a key in a Sorted linked list




```
1//Searching an element <em>x</em>
2
3cell *p =head ;
4
5while(p->next->key < x )  p=p->next ;
6
7return p ;
```

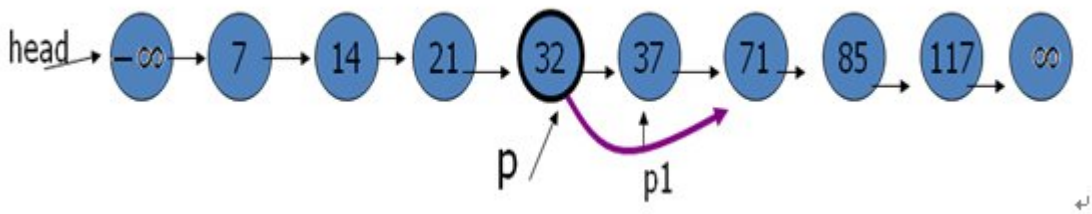
Note: we return the element proceeding either the element containing *x*, or the smallest element with a key larger than *x* (if *x* does not exists)

2) inserting a key into a Sorted linked list




```
1 //To insert 35 -
2
3 p=find(35);
4
5 CELL *p1 = (CELL *) malloc(sizeof(CELL));
6
7 p1->key=35;
8
9 p1->next = p->next ;
10
11p->next  = p1 ;
```

3) deleteing a key from a sorted list



```
//To delete 37 -

p=find(37);

CELL *p1 =p->next;

p->next = p1->next ;

free(p1);
```

1.7.1.2 SkipList(跳跃表)定义

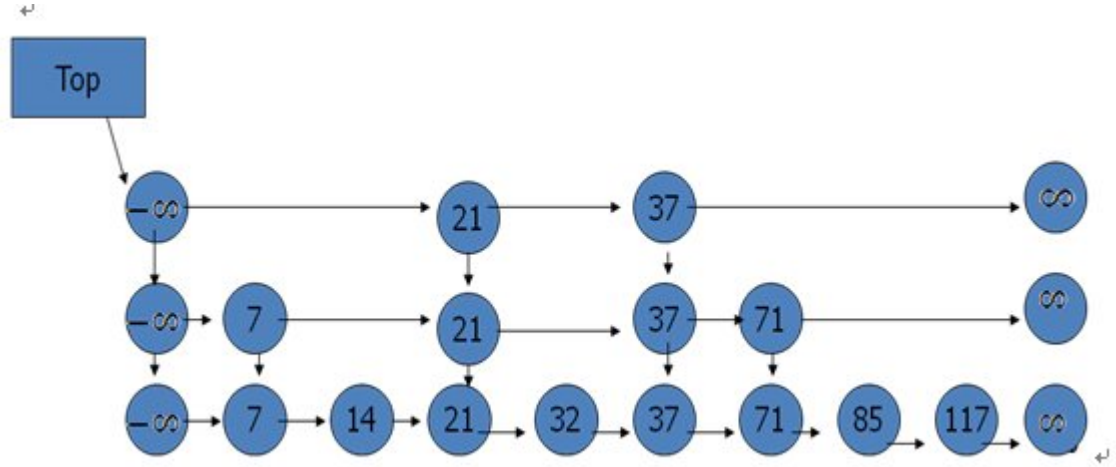
SKIP LIST : A data structure for maintaing a set of keys in a sorted order.

Consists of several **levels.**

All keys appear in level 1

Each level is a sorted list.

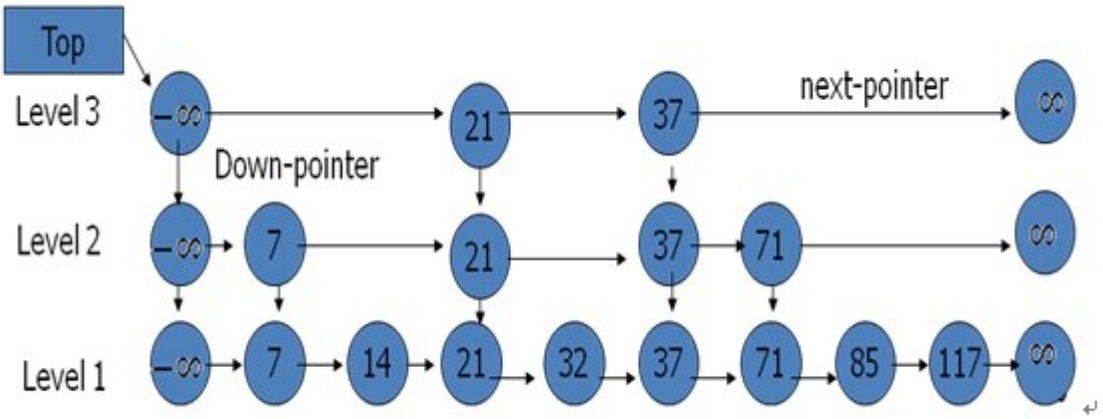
If key *x* appears in level *i*, then it also appears in all levels below *i*



An element in level *i* points (via down pointer) to the element with the same key in the level below.

In each level the keys and appear. (In our implementation, INT_MIN and INT_MAX

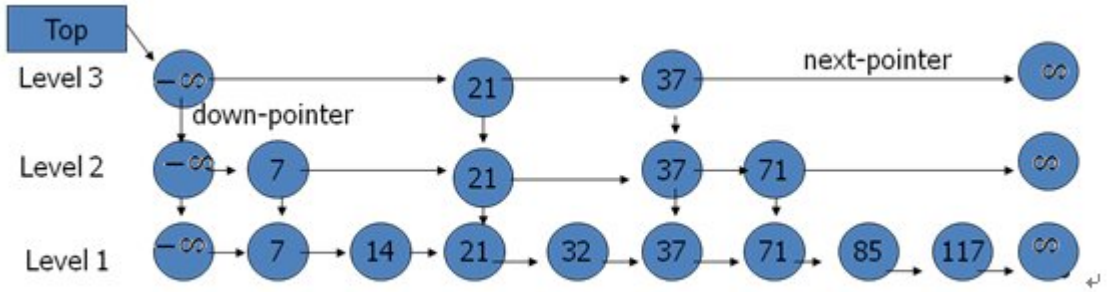
Top points to the smallest element in the highest level.



1.7.1.3 SkipList(跳跃表)操作

1) An empty SkipList

2) Finding an element with key *x*




```
1 p=top
2
3 While(1)
4
5 {
6
7 while(p->next->key < x ) p=p->next;
8
9 If (p->down == NULL ) return p->next
10
11p=p->down ;
12
13}
```

Observe that we return *x*, if exists, or *succ(x)* if *x* is not in the SkipList

3) Inserting new element *X*

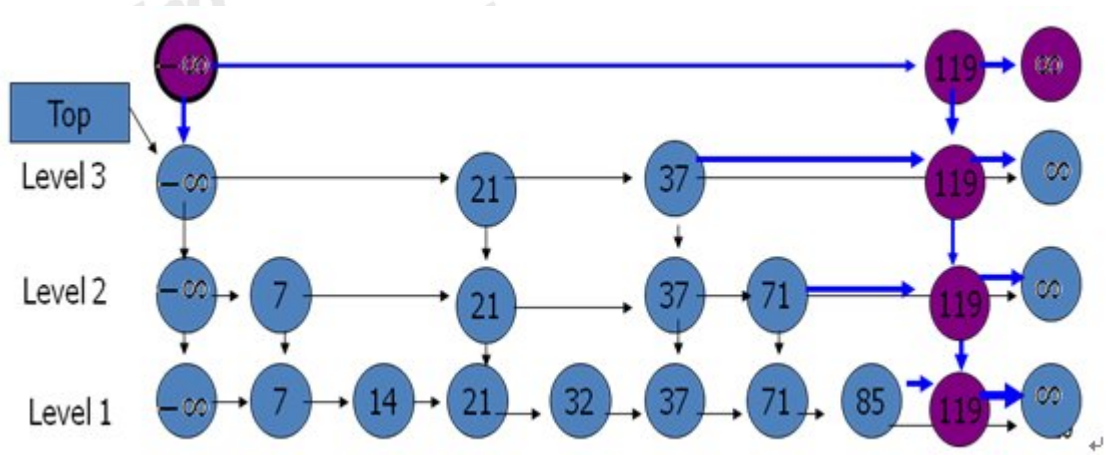
Determine ***k*** the number of levels in which *x* participates (explained later)

Do find(x), and insert x to the appropriate places in the lowest ***k*** levels. (after the elements at which the search path turns down or terminates)

Example – inserting 119. ***k***=2

If ***k*** is larger than the current number of levels, add new levels (and update top)

Example – inser(119) when k=4



Determining *k*

k – the number of levels at which an element x participate.

Use a random function OurRnd() — returns 1 or 0 (True/False) with equal probability.

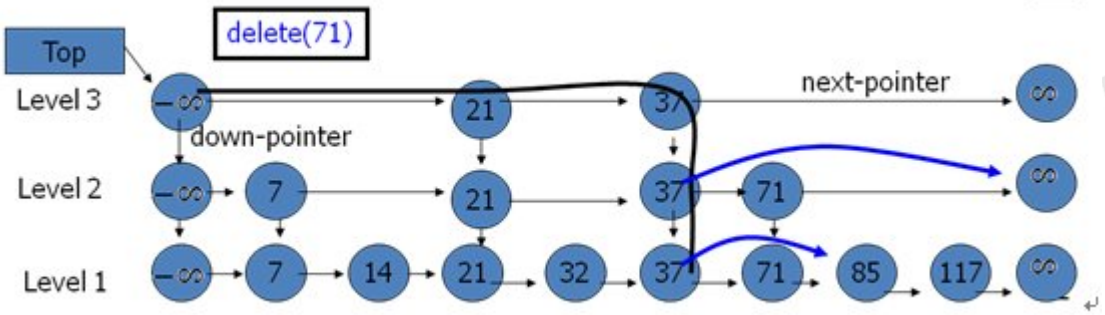
k=1 ;

While(OurRnd()) k++ ;

Deleteing a key *x*

Find x in all the levels it participates, and delete it using the standard ‘delete from a linked list’ method.

If one or more of the upper levels are empty, remove them (and update top)



Facts about SkipList

The expected number of levels is O(log *n*)

(here *n* is the numer of elements)

The expected time for insert/delete/find is O(log *n*)

The expected size (number of cells) is O(*n*)

1.7.2 redis SkipList 实现

/* ZSETs use a specialized version of Skiplists */

```
typedef struct zskiplistNode

{

robj *obj;

double score;

struct zskiplistNode *backward;
```

```
10
11struct zskiplistLevel
12
13{
14
15struct zskiplistNode *forward;
16
17unsigned int span;
18
19} level[];
20
21} zskiplistNode;
22
23typedef struct zskiplist
24
25{
26
27struct zskiplistNode *header, *tail;
28
29unsigned long length;
30
31int level;
32
33} zskiplist;
34
35typedef struct zset
36
37{
38
39dict *dict;
40
41zskiplist *zsl;
42
43} zset;
```

zset 的实现用到了 2 个数据结构: hash_table 和 skip list (跳跃表),其中 hash table 是使用 redis 的 dict 来实现的,主要是为了保证查询效率为 O(1),而 skip list (跳跃表) 是用来保证元素有序并能够保证 INSERT 和 REMOVE 操作是 O(logn)的复杂度。

1) zset 初始化状态

createZsetObject 函数来创建并初始化一个 zset

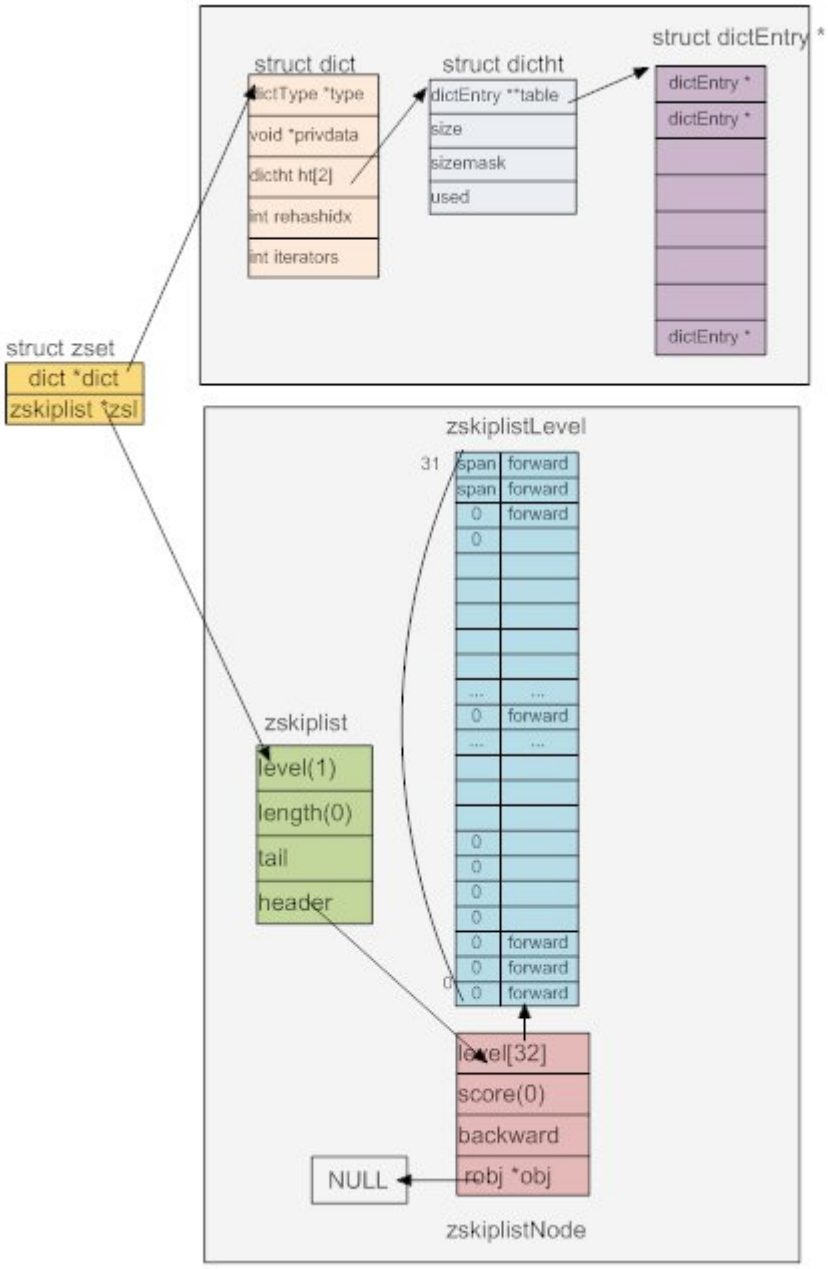
```
1 robj *createZsetObject(void)
2
3 {
4
5 zset *zs = zmalloc(sizeof(*zs));
6
```

```
7 robj *o;
8
9 zs->dict = dictCreate(&zsetDictType,NULL);
10
11zs->zsl = zslCreate();
12
13o = createObject(REDIS_ZSET,zs);
14
15o->encoding = REDIS_ENCODING_SKIPLIST;
16
17return o;
18
19}
```

zslCreate() 函数用来创建并初如化一个 skiplist。 其中，skiplist 的 level 最大值为 ZSKIPLIST_MAXLEVEL=32 层。

```
1 zskiplist *zslCreate(void)
2
3 {
4
5 int j;
6
7 zskiplist *zsl;
8
9 zsl = zmalloc(sizeof(*zsl));
10
11zsl->level = 1;
12
13zsl->length = 0;
14
15zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL,0,NULL);
16
17for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
18
19zsl->header->level[j].forward = NULL;
20
21zsl->header->level[j].span = 0;
22
23}
24
25zsl->header->backward = NULL;
26
27zsl->tail = NULL;
28
29return zsl;
30
```

31}



2) ZADD myzset 1 “one”

ZADD 命令格式:

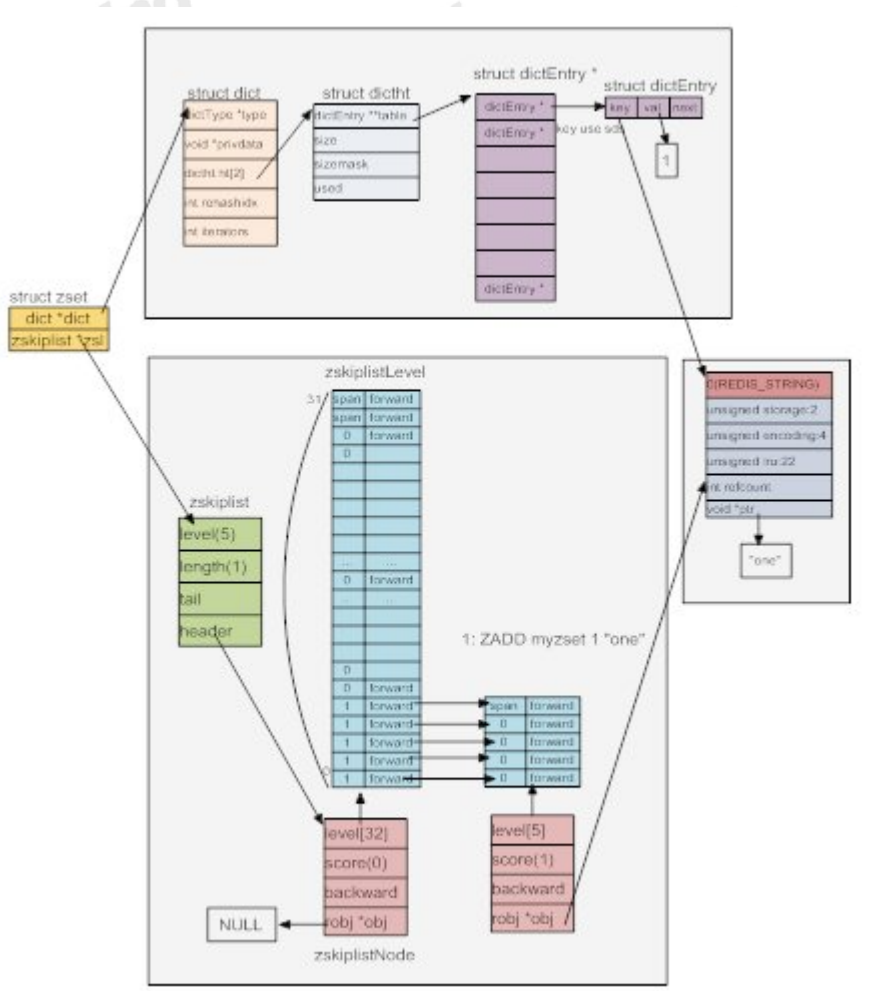
ZADD key score member

根据 key 从 redisDb 进行查询，返回 zset 对象。

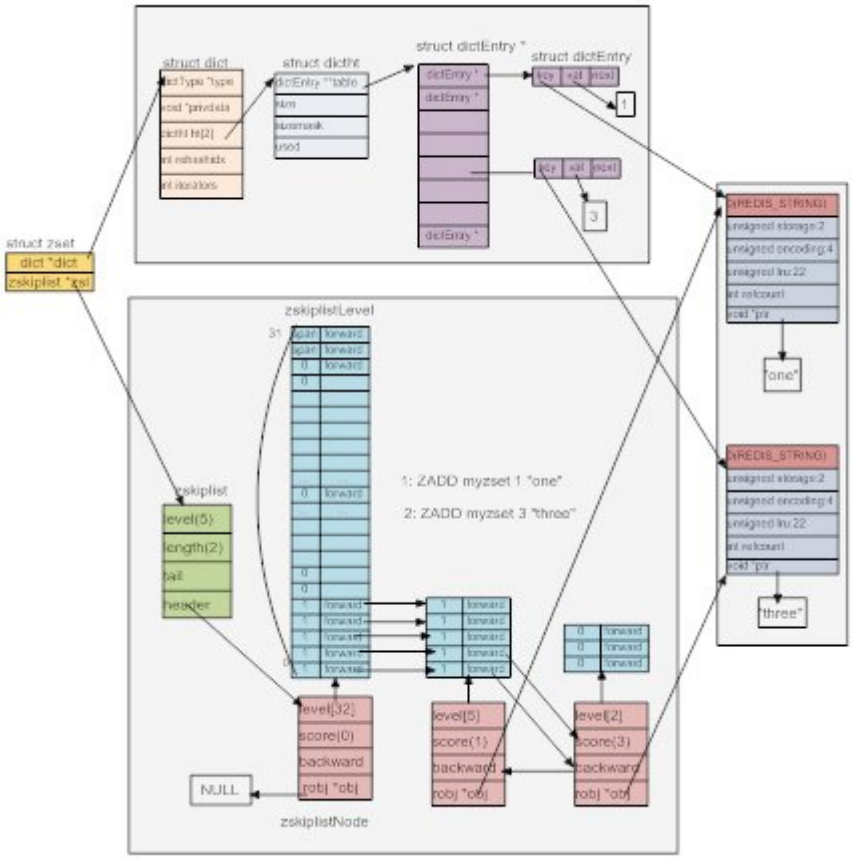
以 member 作为 key,score 作为 value ，向 zset 的 dict 进行中插入;

如果返回成功，表明 member 没有在 dict 中出现过，直接向 skiplist 进行插入。

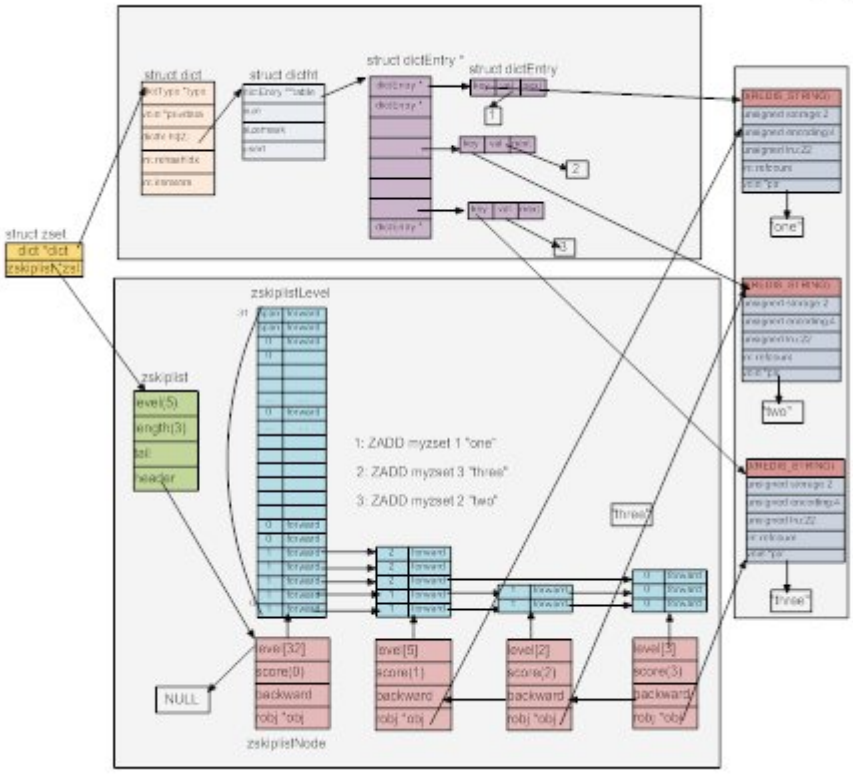
如果步骤返回失败，表明以 member 已经在 dict 中出现过，则需要先从 skiplist 中删除，然后以现在的 score 值重新插入。



3) ZADD myzset 3 “three”



4) ZADD myzset 2 “two”



Copyright © 2005-2012 UPLOOKINGTechnology Co., Ltd.