

Redis详解

回顾：

今天任务

- 1、Redis介绍
- 2、Redis安装
- 3、Redis客户端
- 4、Redis数据类型
- 5、Redis持久化
- 6、Redis主从复制
- 7、Redis集群
- 8、Redis集群的连接
- 9、Redis集群的jedis连接

教学目标

- 1、深入理解Redis的应用场景及概念
- 2、熟练掌握Redis的数据类型
- 3、熟练掌握Redis持久化
- 4、熟悉Redis集群搭建
- 5、熟练掌握Redis集群的Jedis连接

第一节：NoSQL介绍

<http://nosql-database.org/>了解NoSql的概念以及现有的NoSql数据库。

1.1 什么是NoSQL

NoSQL DEFINITION: Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable(分库、分表、分库分表).

为了解决高并发、高可扩展、高可用、大数据存储问题而产生的数据库解决方案，就是NoSql数据库。NoSQL，泛指非关系型的数据库，NoSQL即Not-Only SQL，它可以作为关系型数据库的良好补充。但是它不能替代关系型数据库，而且它是存储在内存中，所以它的访问速度很快。

适用关系型数据库的时候就是用关系型数据库，不适用的时候也没有必要非使用关系型数据库，也可以考虑更加合适的数据存储。

1.2 为什么使用NoSQL

传统关系数据库的优势

- 数据的一致性（事务处理）
- 比较标准规范，数据更新开销小
- 可以进行JOIN进行复杂查询
- 有比较成熟的使用和解决方案

传统关系数据库的瓶颈

- 大量数据的写入更新处理，很难规模化；
- 为有数据更新的表做索引或表结构变更，需要对数据锁定，长时间无法更新等；
- 字段不固定的应用；
- 对简单查询需要快速反应；
- 扩展困难：由于存在类似Join这样多表查询机制，使得数据库在扩展方面很艰难；
- 读写慢：这种情况主要发生在数据量达到一定规模时由于关系型数据库的系统逻辑非常复杂，使得其非常容易发生死锁等的并发问题，所以导致其读写速度下滑非常严重；

1.3 NoSQL优缺点

优点：

易扩展 NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

大数据量，高性能，快速读写 NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

灵活的数据模型 NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的web2.0时代尤其明显。

高可用 NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

低廉的成本

这是大多数分布式数据库共有的特点，因为主要都是开源软件，没有昂贵的License成本；

缺点：但瑕不掩瑜，NoSQL数据库还存在着很多的不足，常见主要有下面这几个：

- 不提供对SQL的支持：如果不支持SQL这样的工业标准，将会对用户产生一定的学习和应用迁移成本；
- 支持的特性不够丰富：现有产品所提供的功能都比较有限，大多数NoSQL数据库都不支持事务，也不像MS SQL Server和Oracle那样能提供各种附加功能，比如BI和报表等；
- 现有产品的不够成熟：大多数产品都还处于初创期，和关系型数据库几十年的完善不可同日而语；

1.4 NoSQL适用的场景

1) 数据库表schema经常变化 比如在线商城，维护产品的属性经常要增加字段，这就意味着ORMapping层的代码和配置要改，如果该表的数据量过百万，新增字段会带来额外开销（重建索引等）。NoSQL应用在这种场景，可以极大提升DB的可伸缩性，开发人员可以将更多的精力放在业务层。

2)数据库表字段是复杂数据类型

对于复杂数据类型，比如SQL Sever提供了可扩展性的支持，像xml类型的字段。很多用过的同学应该知道，该字段不管是查询还是更改，效率非常一般。主要原因是DB层对xml字段很难建高效索引，应用层又要做从字符流到dom的解析转换。NoSQL以json方式存储，提供了原生态的支持，在效率方便远远高于传统关系型数据库。

3)高并发数据库请求

此类应用常见于web2.0的网站，很多应用对于数据一致性要求很低，而关系型数据库的事务以及大表join反而成了“性能杀手”。

4)海量数据的分布式存储

海量数据的存储如果选用大型商用数据，如Oracle，那么整个解决方案的成本是非常高的，要花很多钱在硬件上。NoSQL分布式存储，可以部署在廉价的硬件上，是一个性价比非常高的解决方案。

并不是说NoSQL可以解决一切问题，像ERP系统、BI系统，在大部分情况还是推荐使用传统关系型数据库。主要的原因是此类系统的业务模型复杂，使用NoSQL将导致系统的维护成本增加。

选择合适的NoSQL 如此多类型的NoSQL，而每种类型的NoSQL又有很多，到底选择什么类型的NoSQL来作为我们的存储呢？这并不是一个很好回答的问题，影响我们选择的因素有很多，而选择也可能有多种，随着业务场景，需求的变更可能选择又会变化。我们常常需要根据如下情况考虑：

- 1.数据结构特点。包括结构化、半结构化、字段是否可能变更、是否有大文本字段、数据字段是否可能变化。
- 2.写入特点。包括insert比例、update比例、是否经常更新数据的某一个小字段、原子更新需求。
- 3.查询特点。包括查询的条件、查询热点的范围。比如用户信息的查询，可能就是随机的，而新闻的查询就是按照时间，越新的越频繁。

NoSQL和关系数据库结合 其实NoSQL数据库仅仅是关系数据库在某些方面（性能，扩展）的一个弥补，单从功能上讲，NoSQL的几乎所有的功能，在关系数据库上都能够满足，所以选择NoSQL的原因并不在功能上。

所以，我们一般会把NoSQL和关系数据库进行结合使用，各取所长，需要使用关系特性的时候我们使用关系数据库，需要使用NoSQL特性的时候我们使用NoSQL数据库，各得其所。

举个简单的例子吧，比如用户评论的存储，评论大概有主键id、评论的对象aid、评论内容content、用户uid等字段。我们能确定的是评论内容content肯定不会在数据库中用where content="查询，评论内容也是一个大文本字段。那么我们可以把主键id、评论对象aid、用户id存储在数据库，评论内容存储在NoSQL，这样数据库就节省了存储content占用的磁盘空间，从而节省大量IO，对content也更容易做Cache。

```
//从MySQL中查询出评论主键id列表 commentIds=DB.query("SELECT id FROM comments where aid='评论对象id' LIMIT 0,20"); //根据主键id列表，从NoSQL取回评论实体数据
CommentsList=NoSQL.get(commentIds);
```

NoSQL代替MySQL 在某些应用场合，比如一些配置的关系键值映射存储、用户名和密码的存储、Session会话存储等等，用NoSQL完全可以替代MySQL存储。不但具有更高的性能，而且开发也更加方便。

NoSQL作为缓存服务器 MySQL+Memcached的架构中，我们处处都要精心设计我们的缓存，包括过期时间的设计、缓存的实时性设计、缓存内存大小评估、缓存命中率等等。

NoSQL数据库一般都具有非常高的性能，在大多数场景下面，你不必再考虑在代码层为NoSQL构建一层Memcached缓存。NoSQL数据本身在Cache上已经做了相当多的优化工作。

Memcached这类内存缓存服务器缓存的数据大小受限于内存大小，如果用NoSQL来代替Memcached来缓存数据库的话，就可以不再受限于内存大小。虽然可能有少量的磁盘IO读写，可能比Memcached慢一点，但是完全可以用来缓存数据库的查询操作。

规避风险 由于NoSQL是一个比较新的东西，特别是我们选择的NoSQL数据库还不是非常成熟的产品，所以我们可能会遇到未知的风险。为了得到NoSQL的好处，又要考虑规避风险，鱼与熊掌如何兼得？

现在业内很多公司的做法就是数据的备份。在往NoSQL里面存储数据的时候还会往MySQL里面存储一份。NoSQL数据库本身也需要进行备份（冷备和热备）。或者可以考虑使用两种NoSQL数据库，出现问题后可以切换（避免出现digg使用Cassandra的悲剧）。

1.5 Nosql的数据库分类

- 键值(Key-Value)存储数据库

相关产品：Tokyo Cabinet/Tyrant、**Redis**、Voldemort、Berkeley DB

典型应用：内容缓存，主要用于处理大量数据的高访问负载。

数据模型：一系列键值对

优势：快速查询

劣势：存储的数据缺少结构化

- **列存储数据库**

相关产品：Cassandra, **HBase**, Riak

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在文件系统中

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

- **文档型数据库**

相关产品：CouchDB、MongoDB

典型应用：Web应用（与Key-Value类似，Value是结构化的）

数据模型：一系列键值对

优势：数据结构要求不严格

劣势：查询性能不高，而且缺乏统一的查询语法

- **图形(Graph)数据库**

相关数据库：Neo4j、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

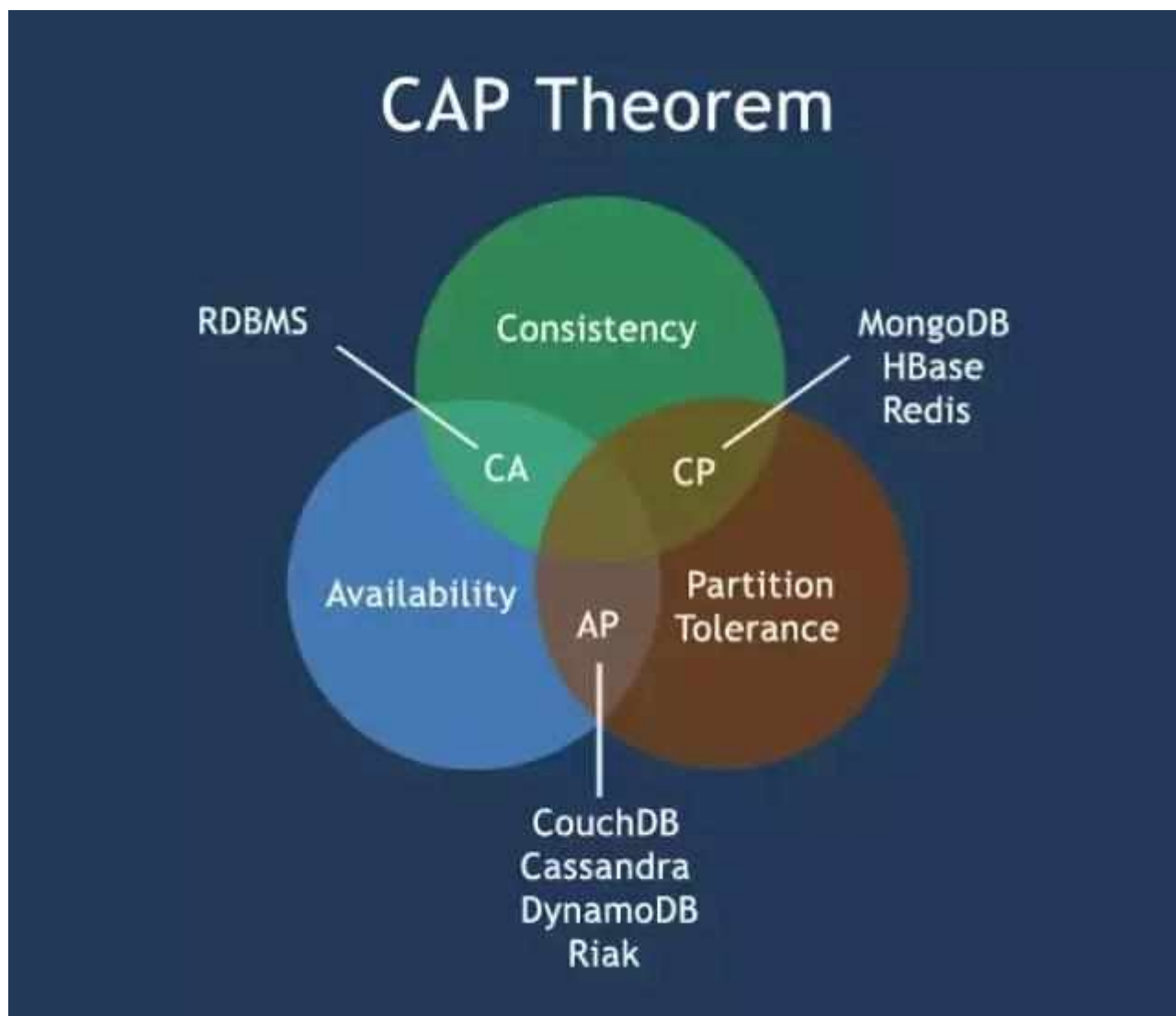
如何选择NoSQL

1.6 如何选择合适的NoSQL(扩展)

CAP理论告诉我们：一个分布式系统不可能同时满足一致性(C:Consistency)、可用性(A:Availability)、分区容错性(P:Partitiontolerance)这三个基本需求，并且最多只能满足其中的两项。

- C：一致性被称为原子对象，任何的读写都应该看起来是“原子”的，或串行的。写后面的读一定能读到前面写的内容。所有的读写请求都好像被全局排序。
- A：对任何非失败节点都应该在有限时间内给出请求的回应。“可用性”，意思是只要收到用户的请求，服务器就必须给出回应，（在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求）（请求的可终止性）
- P：分区容错，区间通信可能失败。允许节点之间丢失任意多的消息，当网络分区发生时，节点之间的消息可能会完全丢失

(大多数分布式系统都分布在多个子网络。每个子网络就叫做一个区 (partition)。分区容错的意思是，区间通信可能失败。比如，一台服务器放在中国，另一台服务器放在美国，这就是两个区，它们之间可能无法通信。G1 和 G2 是两台跨区的服务器。G1 向 G2 发送一条消息，G2 可能无法收到。系统设计的时候，必须考虑到这种情况。一般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。)



第二节 Redis入门

2.1 Redis介绍

Redis是用C语言开发的一个开源的高性能键值对 (key-value) 数据库。它通过提供多种键值数据类型来适应不同场景下的存储需求，目前为止Redis支持的键值数据类型如下：

- 字符串类型
- 散列类型
- 列表类型
- 集合类型
- 有序集合类型。

2.2 Redis发展简史

2008年，意大利的一家创业公司Merzia推出了一款基于MySQL的网站实时统计系统LLOOGG，然而没过多久该公司的创始人 Salvatore Sanfilippo便对MySQL的性能感到失望，于是他决定亲自为LLOOGG量身定做一个数据库，并于2009年开发完成，这个数据库就是Redis。不过Salvatore Sanfilippo并不满足只将Redis用于LLOOGG这一款产品，而是希望更多的人使用它，于是在同一年Salvatore Sanfilippo将Redis开源发布，并开始和Redis的另一名主要的代码贡献者Pieter Noordhuis一起继续着Redis的开发，直到今天。

SalvatoreSanfilippo自己也没有想到，短短的几年时间，Redis就拥有了庞大的用户群体。Hacker News在2012年发布了一份数据库的使用情况调查，结果显示有近12%的公司在使用Redis。国内如新浪微博、街旁网、知乎网，国外如GitHub、Stack Overflow、Flickr等都是Redis的用户。

VMware公司从2010年开始赞助Redis的开发，Salvatore Sanfilippo和Pieter Noordhuis也分别在3月和5月加入VMware，全职开发Redis。

2.3 国内使用情况

2.4 Redis应用场景

- 缓存（数据查询、短连接、新闻内容、商品内容等等）。（最多使用）
- 分布式集群架构中的session分离。
- 聊天室的在线好友列表。
- 任务队列。（秒杀、抢购、12306等等）
- 应用排行榜。
- 网站访问统计。
- 数据过期处理（可以精确到毫秒）

2.5 Redis的特性

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

2.6 下载Redis

官网地址：<http://redis.io/>

下载地址：<http://download.redis.io/releases/redis-3.0.0.tar.gz>

2.7 Redis安装和启动

Redis安装一般会在Linux系统下进行安装，又因为redis是使用c语言开发，所以需要c语言环境。

- Linux：centOS
- VMware:10
- C语言环境：

2.7.1 Redis安装

第一步：在VMware中安装CentOS系统（Linux）。

第二步：在Linux系统中安装c语言环境

```
[root@redis01~]# yum install gcc-c++
```

离线安装，下载依赖的所有文件，在文件所在的文件夹内，执行

`rpm -Uvh *.rpm --nodeps --force` 安装过程走完了过后，前往 `/usr/bin` 目录查看是否有 `gcc`和`g++`两个文件夹，如果有，说明安装成功。

或者使用`gcc -v`

第三步：将redis的源码包上传到Linux系统。

```
[root@redis01 ~]# ll
总用量 1524
-rw-r--r--. 1 root root 36582 10月 8 23:25 08.jpg
-rw-----. 1 root root 1639 10月 19 19:56 anaconda-ks.cfg
-rw-r--r--. 1 root root 48546 10月 19 19:56 install.log
-rw-r--r--. 1 root root 10726 10月 19 19:55 install.log.syslog
-rw-r--r--. 1 root root 57856 10月 8 21:50 redis-3.0.0.gem
-rw-r--r--. 1 root root 1358081 10月 8 21:50 redis-3.0.0.tar.gz
drwxr-xr-x. 2 root root 4096 10月 21 04:55 公共的
drwxr-xr-x. 2 root root 4096 10月 21 04:55 模板
drwxr-xr-x. 2 root root 4096 10月 21 04:55 视频
drwxr-xr-x. 2 root root 4096 10月 21 04:55 图片
```

第四步：解压源码包

```
[root@redis01 ~]# ll
总用量 1528
-rw-r--r--. 1 root root 36582 10月 8 23:25 08.jpg
-rw-----. 1 root root 1639 10月 19 19:56 anaconda-ks.cfg
-rw-r--r--. 1 root root 48546 10月 19 19:56 install.log
-rw-r--r--. 1 root root 10726 10月 19 19:55 install.log.syslog
drwxrwxr-x. 6 root root 4096 4月 1 2015 redis-3.0.0
-rw-r--r--. 1 root root 57856 10月 8 21:50 redis-3.0.0.gem
-rw-r--r--. 1 root root 1358081 10月 8 21:50 redis-3.0.0.tar.gz
drwxr-xr-x. 2 root root 4096 10月 21 04:55 公共的
drwxr-xr-x. 2 root root 4096 10月 21 04:55 模板
drwxr-xr-x. 2 root root 4096 10月 21 04:55 视频
drwxr-xr-x. 2 root root 4096 10月 21 04:55 图片
drwxr-xr-x. 2 root root 4096 10月 21 04:55 文档
drwxr-xr-x. 2 root root 4096 10月 21 04:55 下载
drwxr-xr-x. 2 root root 4096 10月 21 04:55 音乐
```

第五步：进入redis-3.0.0包，然后执行make命令，编译redis的源码

```
[root@redis01 redis-3.0.0]# make
```

有可能报如下错误：

```
zmalloc.h:50:31: error: jemalloc/jemalloc.h: No such file or directory
zmalloc.h:55:2: error:
#error "Newer version of jemalloc required"
make[1]: *** [adlist.o] Error 1
make[1]: Leaving directory `/usr/local/src/redis-3.2.9/src'
make: *** [all] Error 2
```

解决方式（README）：

```
make MALLOC=libc
```

第六步：安装

安装并指定安装目录.

```
[root@redis01redis-3.0.0]# make install PREFIX=/usr/local/redis
```

2.7.2 Redis启动

三种启动方式

1. 前端启动

前端启动，如果客户端关掉或者执行ctrl+c命令。则整个redis服务也停掉。

前端启动，即在客户端中执行以下命令：

```
[root@redis01 bin]# ./redis-server
```

```
-rwxr-xr-x. 1 root root 5686505 10月 24 17:51 redis-server
[root@redis01 bin]# ./redis-server
5187:C 24 Oct 18:05:40.132 # warning: no config file specified, using the default
t config. In order to specify a config file use ./redis-server /path/to/redis.co
nf
5187:M 24 Oct 18:05:40.133 * Increased maximum number of open files to 10032 (it
was originally set to 1024).
5187:M 24 Oct 18:05:40.166 # Warning: 32 bit instance detected but no memory lim
it set. Setting 3 GB maxmemory limit with 'noeviction' policy now.

  _____
 /  _  _  _  \
|  _ \| | | | | |
| |_) | | | |
|  _ <| | | |
|_| \_|_|_|_|

Redis 3.0.0 (00000000/0) 32 bit
Running in standalone mode
Port: 6379
PID: 5187

http://redis.io
```

关闭：ctrl+c

2. 后端启动

第一步：执行cp命令将redis解压缩包中的redis.conf文件拷贝到bin目录下


```

[root@redis01 redis-3.0.0]# cp redis.conf /usr/local/redis0707/bin/
[root@redis01 redis-3.0.0]# cd /usr/local/redis0707/bin/
[root@redis01 bin]# ll
总用量 13896
-rw-r--r--. 1 root root      18 10月 24 18:07 dump.rdb
-rwxr-xr-x. 1 root root 4167902 10月 24 17:51 redis-benchmark
-rwxr-xr-x. 1 root root  16459 10月 24 17:51 redis-check-aof
-rwxr-xr-x. 1 root root  37691 10月 24 17:51 redis-check-dump
-rwxr-xr-x. 1 root root 4256668 10月 24 17:51 redis-cli
-rw-r--r--. 1 root root   41403 10月 24 18:11 redis.conf
lrwxrwxrwx. 1 root root      12 10月 24 17:51 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 5686505 10月 24 17:51 redis-server

```

第二步：修改redis.conf文件：

```

#
# include /path/to/local.conf
# include /path/to/other.conf

##### GENERAL #####

# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when
daemonize yes

# When running daemonized, Redis writes a pid file in /var/run/
# default. You can specify a custom pid file location here.
pidfile /var/run/redis.pid

# Accept connections on the specified port, default is 6379
# If port 0 is specified Redis will not listen on a TCP socket

```

将no 改为 yes 即可后端启动

第三步：启动redis服务

```

-rwxr-xr-x. 1 root root 5686505 10月 24 17:51 redis-server
[root@redis01 bin]# ./redis-server redis.conf
[root@redis01 bin]# ps -ef|grep red
root      5261      1   0 18:19 ?        00:00:00 ./redis-server *:6379
root      5267  5191   0 18:19 pts/2    00:00:00 grep red
[root@redis01 bin]#

```

关闭：正常关闭

```

[root@redis01 bin]# ./redis-cli shutdown
[root@redis01 bin]# ps -ef|grep red
root      5275  5191   0 18:20 pts/2    00:00:00 grep red
[root@redis01 bin]#

```

非正常关闭

```
bash: kill: (5292) 没有那个进程
[root@redis01 bin]# ./redis-server redis.conf
[root@redis01 bin]# ps -ef|grep red
root      5292      1   0 18:22 ?          00:00:00 ./redis-server *:6379
root      5296  5191   0 18:22 pts/2    00:00:00 grep red
[root@redis01 bin]# kill 5292
[root@redis01 bin]# ps -ef|grep red
root      5298  5191   0 18:22 pts/2    00:00:00 grep red
[root@redis01 bin]#
```

第三节 Redis客户端

3.1 Redis自带的客户端

```
not connected> quit
[root@redis01 bin]# ./redis-server redis.conf
[root@redis01 bin]# ./redis-cli -h 127.0.0.1 -p 6379
127.0.0.1:6379> set s1 redis1
OK
127.0.0.1:6379> get s1
"redis1"
127.0.0.1:6379>
```

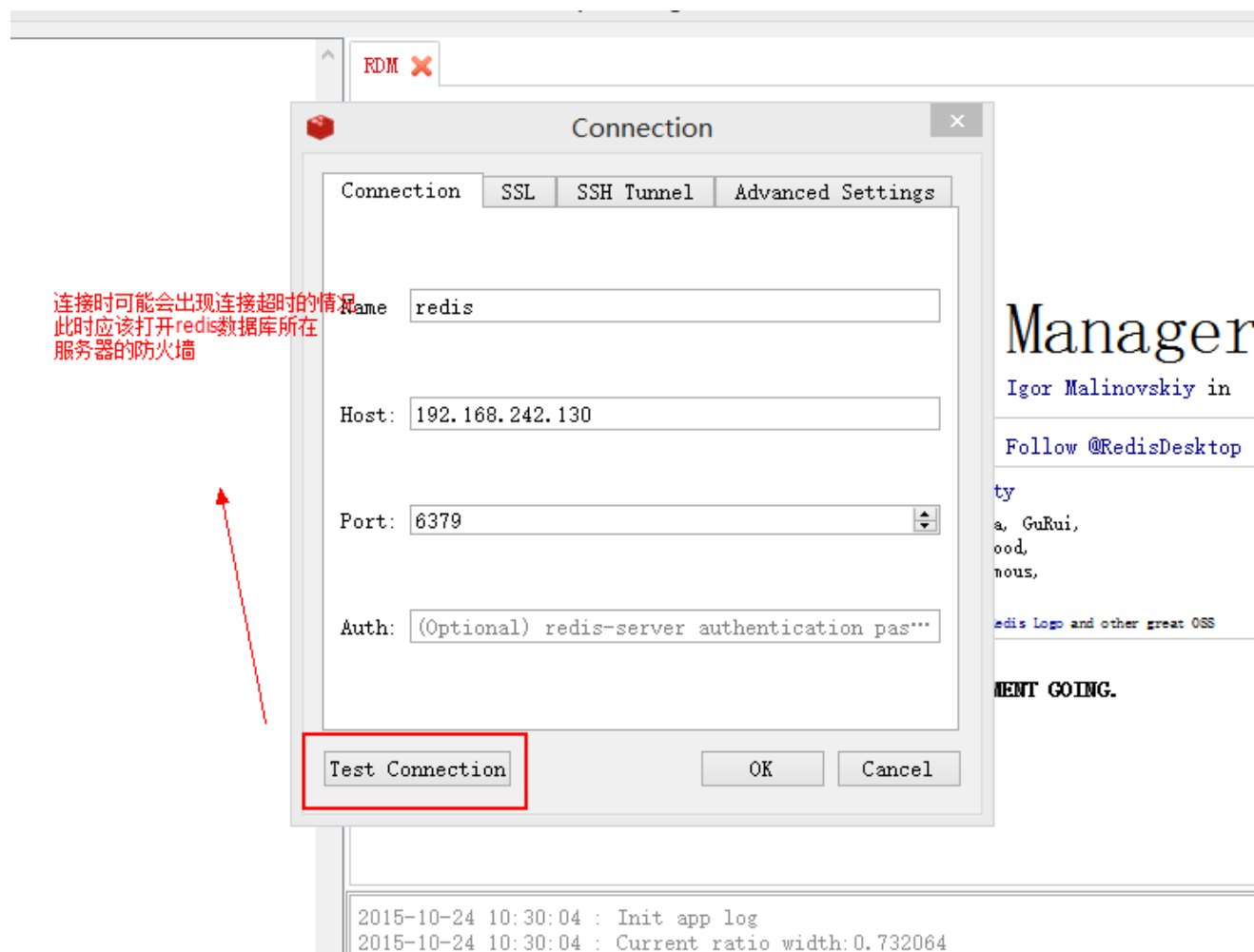
./redis-cli -h redis数据库的ip -p 端口号

默认可以执行 ./redis-cli 此时使用默认的ip为127.0.0.1 默认的端口为 6379

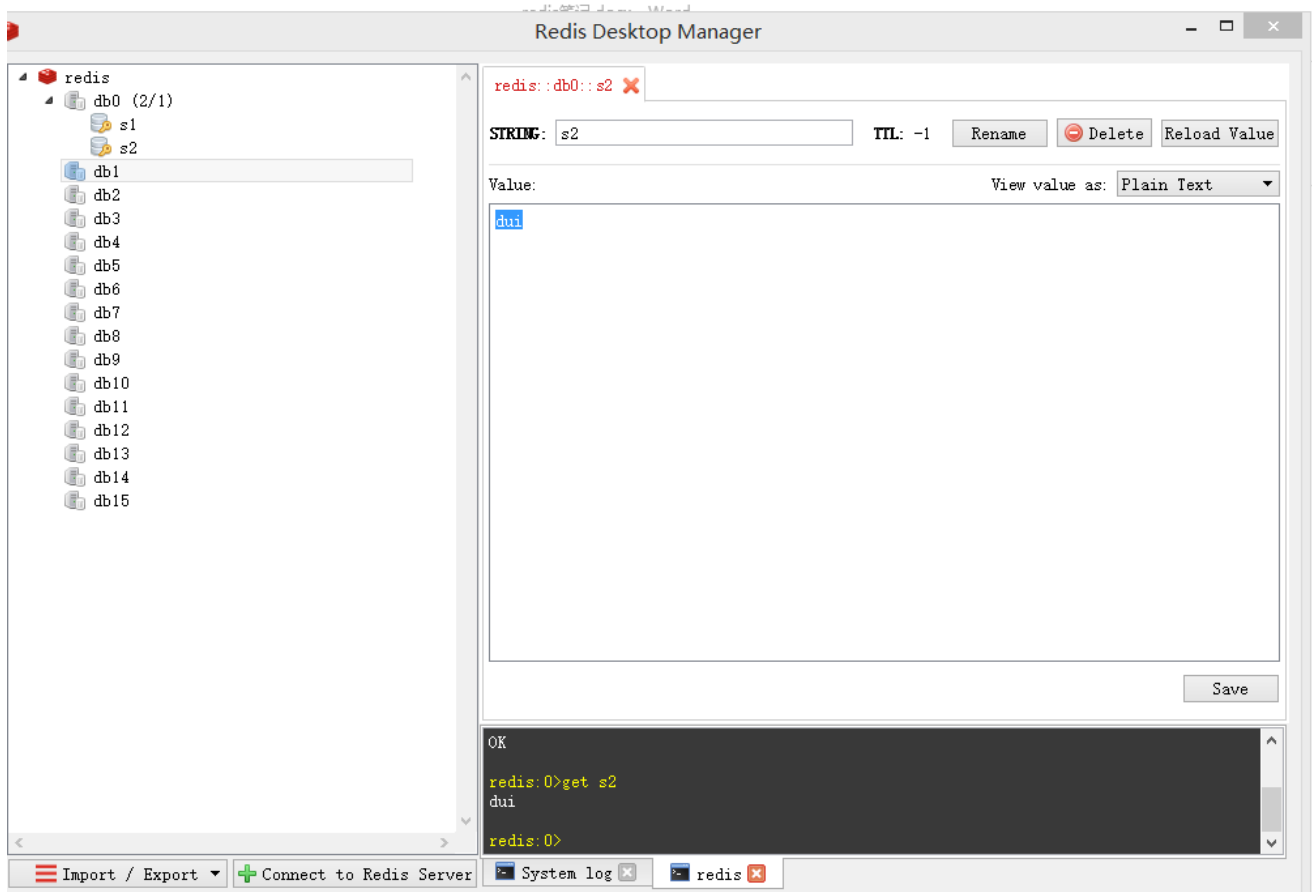
3.2 Redis桌面管理工具

第一步：安装redis桌面管理工具

第二步：创建连接



界面如下：



Redis默认有16个库，这个数字可以修改。

切换库使用如下命令：

```
127.0.0.1:6379> quit
[root@redis01 bin]# ./redis-cli
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> set s3 haha
OK
```

3.2.1 访问超时

打开防火墙

```
[root@redis01 ~]# vi /etc/sysconfig/iptables

# Firewall configuration written by system-config-firewall

# Manual customization of this file is not recommended.

*filter

:INPUT ACCEPT [0:0]

:FORWARD ACCEPT [0:0]

:OUTPUT ACCEPT [0:0]
```

```

-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

-A INPUT -p icmp -j ACCEPT

-A INPUT -i lo -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT


-A INPUT -j REJECT --reject-with icmp-host-prohibited

-A FORWARD -j REJECT --reject-with icmp-host-prohibited

COMMIT

"/etc/sysconfig/iptables" 15L,544C 已写入

[root@redis01 ~]# service iptables restart

iptables : 清除防火墙规则 : [确定]

iptables : 将链设置为政策 ACCEPT : filter [确定]

iptables : 正在卸载模块 : [确定]

iptables : 应用防火墙规则 : [确定]

[root@redis01 ~]#

```

3.3 Java客户端

3.3.1 Jedis介绍

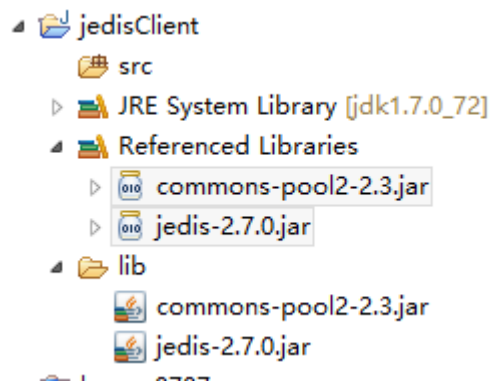
Redis不仅是使用命令来操作，现在基本上主流的语言都有客户端支持，比如java、C、C#、C++、php、Node.js、Go等。

在官方网站里列一些Java的客户端，有Jedis、Redisson、Jredis、JDBC-Redis、等其中官方推荐使用Jedis和Redisson。在企业中用的最多的就是Jedis，下面我们就重点学习下Jedis。

Jedis同样也是托管在github上，地址：<https://github.com/xetorthio/jedis>

3.3.2 环境准备及工程搭建

Jedis依赖：



3.3.3 单机连接Redis

@Test

```
public void jedisClientTest() {  
    // 创建Jedis  
    // host : redis数据库的ip地址  
    // port: redis数据库的端口号  
    Jedis jedis = new Jedis("192.168.242.130", 6379);  
  
    // 通过jedis存储值  
    jedis.set("s4", "jedis test");  
  
    // 通过jedis获取值  
    String s2 = jedis.get("s2");  
  
    System.out.println(s2);  
  
    // 关闭jedis  
    jedis.close();  
}
```

3.3.4 连接池连接

```

@Test
public void jedisPoolTest() {
    // 创建JedisPool
    JedisPool pool = new JedisPool("192.168.242.130", 6379);

    // 通过pool获取jedis实例
    Jedis jedis = pool.getResource();

    jedis.set("s5", "jedis pool test");

    // 关闭jedis
    jedis.close();

    // 关闭jedisPool
    pool.close();
}

```

第四节 Redis数据类型

4.0 键和值类型

键类型

Redis keys are binary safe, this means that you can use any binary sequence as a key, from a string like "foo" to the content of a JPEG file. The empty string is also a valid key. A few other rules about keys:

Redis的键是二进制安全的，你可以使用任意的二进制序列作为键，比如简单的“foo”，或者复杂的JPEG文件，空字符串也是一个有效的键值。

键的一些创建建议：

Very long keys are not a good idea. For instance a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons. Even when the task at hand is to match the existence of a large value, hashing it (for example with SHA1) is a better idea, especially from the perspective of memory and bandwidth. Very short keys are often not a good idea. There is little point in writing "u1000flw" as a key if you can instead write "user:1000:followers". The latter is more readable and the added space is minor compared to the space used by the key object itself and the value object. While short keys will obviously consume a bit less memory, your job is to find the right balance. Try to stick with a schema. For instance "object-type:id" is a good idea, as in "user:1000". Dots or dashes are often used for multi-word fields, as in "comment%reply.to" or "comment%reply-to". The maximum allowed key size is 512 MB.

值类型

String类型

Hash类型

List类型

Set类型

SortedSet类型

4.1 String类型

4.1.1 二进制安全的字符串

string是二进制安全的：

```
struct sdshdr {  
  
    int len; //记录buf数组大小  
  
    int free; //记录buf数组还有多少可用空间  
  
    char buf[]; //字符串实体，保存字符串的内容  
  
};
```

因为有了对字符串长度定义len, 所以在处理字符串时候不会以零值字节(\0)为字符串结尾标志. 二进制安全就是输入任何字节都能正确处理, 即使包含零值字节.

Note – A string value can be at max 512 megabytes in length.

4.1.2 常用命令

```
#SET key value 设置指定 key 的值  
127.0.0.1:6379> set str1 angelababy  
OK  
  
#GET key 获取指定 key 的值。  
127.0.0.1:6379> get str1  
" angelababy"  
  
#DEL key 删除指定 key  
127.0.0.1:6379> del str1  
(integer) 1  
  
#自增  
#INCR key 将 key 中储存的数字值增一。  
必须value为数字类型  
127.0.0.1:6379> set s1 1  
OK  
127.0.0.1:6379> incr s1  
(integer) 2  
127.0.0.1:6379> incr s1  
(integer) 3  
127.0.0.1:6379> incr s1  
(integer) 4  
127.0.0.1:6379> incr s1  
(integer) 5  
  
#自减  
#DECR key 将 key 中储存的数字值减一。  
127.0.0.1:6379> decr s1  
(integer) 4  
127.0.0.1:6379> decr s1
```



```

(integer) 3
127.0.0.1:6379> decr s1
(integer) 2
127.0.0.1:6379> decr s1

#自增自减指定数值
#INCRBY key increment
#DECRBY key decrement key
127.0.0.1:6379> incrby s1 3
(integer) 4
127.0.0.1:6379> decrby s1 3
(integer) 1

#设置或者获取多个key/value
#MSET key value [key value ...]
#MGET key1 [key2...]
127.0.0.1:6379> mset s1 v1 s2 v2
OK
127.0.0.1:6379> mget s1 s2
1) "v1"
2) "v2"

```

4.1.3 其他命令（自学）

GETRANGE key start end 返回 key 中字符串值的子字符串。 GETSET key value 将给定 key 的值设为 value，并返回 key 的旧值(old value)。 SETEX key seconds value 将值 value 关联到 key，并将 key 的过期时间设为 seconds (以秒为单位)。 SETNX key value 只有在 key 不存在时设置 key 的值。 STRLEN key 返回 key 所储存的字符串值的长度。 MSETNX key value [key value ...] 同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。 PSETEX key milliseconds value 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。 INCRBYFLOAT key increment 将 key 所储存的值加上给定的浮点增量值 (increment)。。 APPEND key value 如果 key 已经存在并且是一个字符串， APPEND 命令将 value 追加到 key 原来的值的末尾。

4.2 Hash类型

4.2.1 使用String的问题

假设有User对象以JSON序列化的形式存储到Redis中，User对象有id，username、password、age、name等属性，存储的过程如下：

保存、更新：

User对象 → json(string) → redis

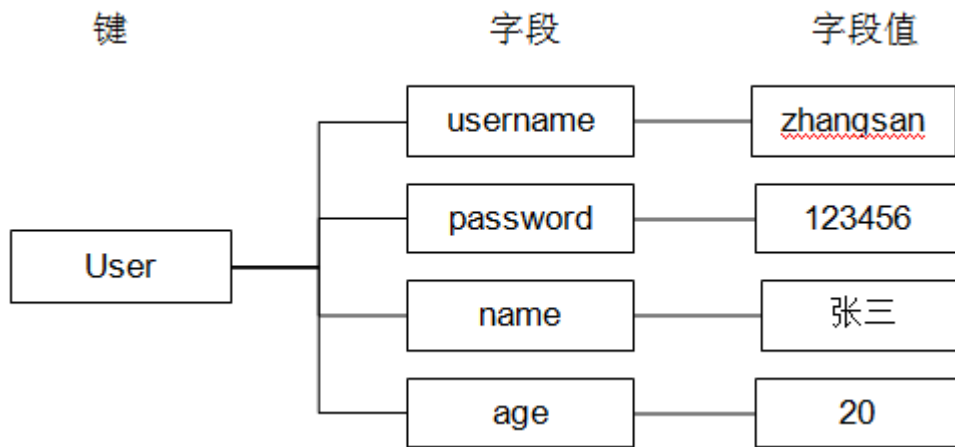
如果在业务上只是更新age属性，其他的属性并不做更新我应该怎么做呢？如果仍然采用上边的方法在传输、处理时会造成资源浪费，下边讲的hash可以很好的解决这个问题。

4.2.2 Redis Hash介绍

redis散列是键值对的集合，是字符串字段和字符串字段值的map，经常用来存储对象。

每个 hash 可以存储 $2^{32} - 1$ 键值对（40多亿）。

注：字段值只能是字符串类型，不支持散列类型、集合类型等其它类型。如下：



对象里含有其他对象怎么处理？

4.2.3 命令

- hset

在redis中，命令是不区分大小写，但是key区分大小写

```
127.0.0.1:6379> hset m1 k1 v1
(integer) 1
127.0.0.1:6379> HSET m1 k1 v1
(integer) 0
```

- hget

```
127.0.0.1:6379> hget m1 k1
"v1"
127.0.0.1:6379> hget m1
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379>
```

- hdel

```
127.0.0.1:6379> hdel m1
(error) ERR wrong number of arguments for 'hdel' command
127.0.0.1:6379> hdel m1 k1
(integer) 1
```

- 批量操作

```
127.0.0.1:6379> hmset m1 k1 v1 k2 v2
OK
127.0.0.1:6379> hmget m1 k1 k2
1) "v1"
2) "v2"
127.0.0.1:6379> hmget m1 k1 k2 k3
1) "v1"
2) "v2"
3) (nil)
```

- 增加数值

```
127.0.0.1:6379> hincrby m2 k1
(error) ERR wrong number of arguments for 'hincrby' command
127.0.0.1:6379> hincrby m2 k1 2
(integer) 3
```

4.2.4 其他命令（自学）

- 判断字段是否存在

HEXISTS key field

```
127.0.0.1:6379> hexists user age      查看user中是否有age字段
(integer) 1
127.0.0.1:6379> hexists user name    查看user中是否有name字段
(integer) 0
```

HSETNX key field value

当**字段**不存在时赋值，类似HSET，区别在于如果字段已经存在，该命令不执行任何操作。

```
127.0.0.1:6379> hsetnx user age 30  如果user中没有age字段则设置age值为30，否则不做任何操作
(integer) 0
```

- 只获取字段名或字段值

HKEYS key

HVALS key

```
127.0.0.1:6379> hmset user age 20 name lisi
OK
127.0.0.1:6379> hkeys user
1) "age"
2) "name"
127.0.0.1:6379> hvals user
1) "20"
2) "lisi"
```

- 获取字段数量

HLEN key

```
127.0.0.1:6379> hlen user
(integer) 2
```

4.2.5 应用

商品id、商品名称、商品描述、商品库存、商品好评

定义商品信息的key：

商品1001的信息在 redis中的key为：items:1001

存储商品信息

```
192.168.101.3:7003> HMSET items:1001 id 3 name apple price 999.9
OK
```

获取商品信息

```
192.168.101.3:7003> HGET items:1001 id
"3"
192.168.101.3:7003> HGETALL items:1001
1) "id"
2) "3"
3) "name"
4) "apple"
5) "price"
6) "999.9"
```

4.3 List类型

List是有序可重复的集合。

Redis Lists 是字符串构成的list，排序按照插入的顺序，可以在Redis Lists的头部和尾部追加元素。

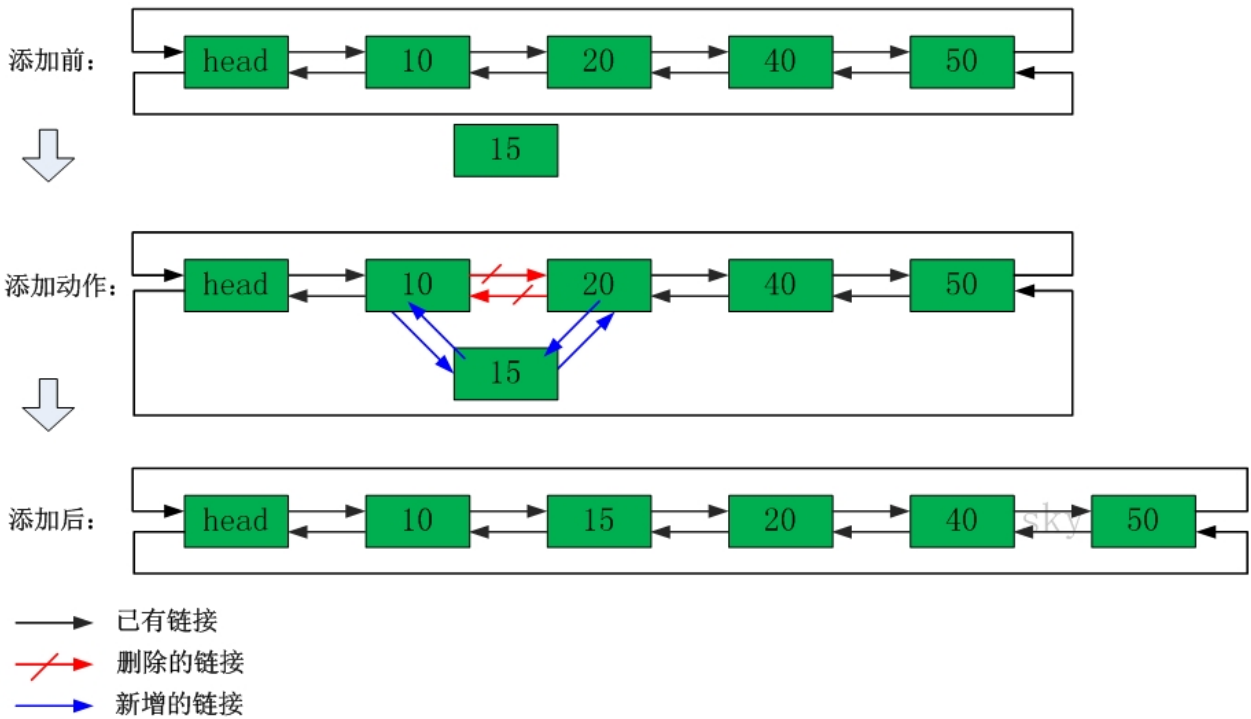
一个列表最多可以包含 $2^{32} - 1$ 个元素 (4294967295, 每个列表超过40亿个元素)。

4.3.1 ArrayList与LinkedList的区别

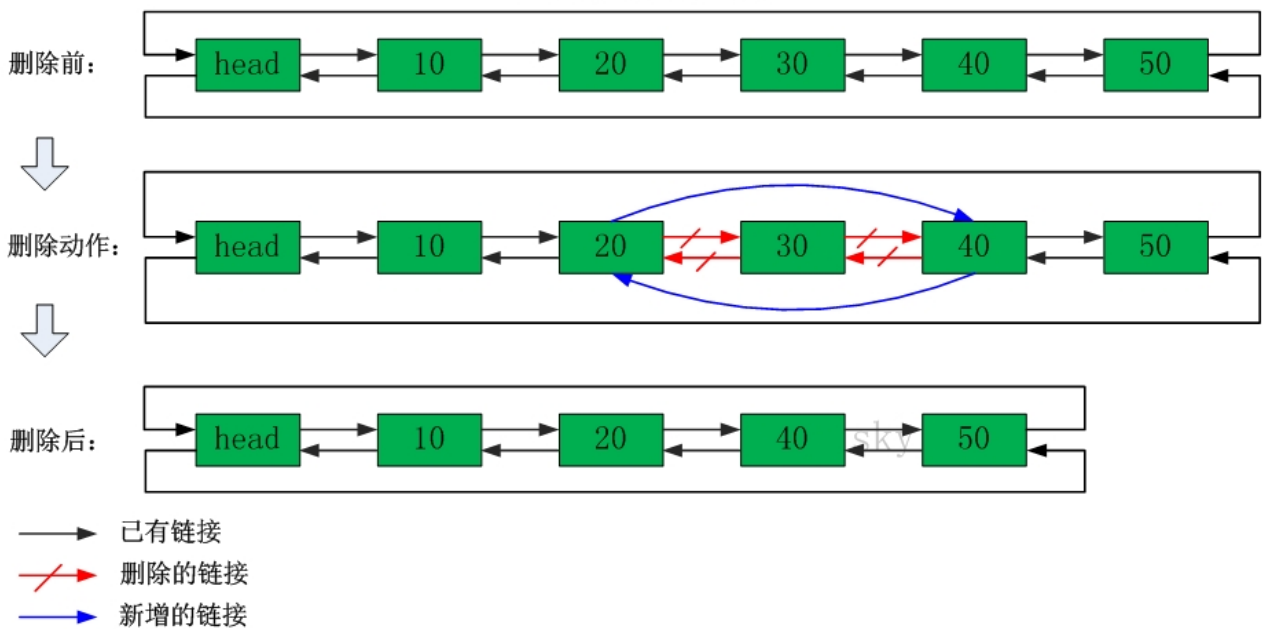
ArrayList使用数组方式存储数据，所以根据索引查询数据速度快，而新增或者删除元素时需要设计到位移操作，所以比较慢。

LinkedList使用双向链接方式存储数据，每个元素都记录前后元素的指针，所以插入、删除数据时只是更改前后元素的指针指向即可，速度非常快，然后通过下标查询元素时需要从头开始索引，所以比较慢，但是如果查询前几个元素或后几个元素速度比较快。

双链表添加节点示意图



双链表删除节点示意图



总结

arrayList在进行增删改时很麻烦

4.3.2 命令

- 从左边存值

```
127.0.0.1:6379> lpush list1 1 2 3 4 5 6
(integer) 6
```

- 从右边存值

```
127.0.0.1:6379> rpush list1 a b c d
(integer) 10
```

- 查看List值

```
127.0.0.1:6379> lrange list1 0 3
1) "6"
2) "5"
3) "4"
4) "3"
```

如果查看全部，使用以下命令：

```
127.0.0.1:6379> lrange list1 0 -1
1) "6"
2) "5"
3) "4"
4) "3"
5) "2"
6) "1"
7) "a"
8) "b"
9) "c"
10) "d"
```

- 从两端弹出值

```
# LPUSH key value1 [value2] 将一个或多个值插入到列表头部
# LPUSHX key value 将一个或多个值插入到已存在的列表头部
# LPUSH与LPUSHX区别？？
127.0.0.1:6379> lpush list1 1 2 3 4 5 6
(integer) 6
127.0.0.1:6379> lrange list1 0 -1
1) "6"
2) "5"
3) "4"
4) "3"
5) "2"
6) "1"
# LPOP key 移出并获取列表的第一个元素
127.0.0.1:6379> lpop list1
"6"
127.0.0.1:6379> lrange list1 0 -1
1) "5"
2) "4"
3) "3"
4) "2"
5) "1"
```

```
#RPOP key 移除并获取列表最后一个元素
127.0.0.1:6379> rpop list1
"1"
127.0.0.1:6379> lrange list1 0 -1
1) "5"
2) "4"
3) "3"
4) "2"
```

- 获取列表的长度

```
127.0.0.1:6379> llen list1
(integer) 4
```

4.3.3 其他命令（自学）

- 删除列表中指定的值

LREM key count value

LREM命令会删除列表中前count个值为value的元素，返回实际删除的元素个数。根据count值的不同，该命令的执行方式会有所不同：

当count>0时，LREM会从列表左边开始删除。

当count<0时，LREM会从列表后边开始删除。

当count=0时，LREM删除所有值为value的元素。

- 获得/设置指定索引的元素值

LINDEX key index

LSET key index value

```
# LINDEX key index 通过索引获取列表中的元素
127.0.0.1:6379> lindex l:list 2
"1"
# LSET key index value 通过索引设置列表元素的值
127.0.0.1:6379> lset l:list 2 2
OK
127.0.0.1:6379> lrange l:list 0 -1
1) "6"
2) "5"
3) "2"
4) "2"
```

- 只保留列表指定片段，指定范围和LRANGE一致

LTRIM key start stop

```
127.0.0.1:6379> lrange l:list 0 -1
1) "6"
2) "5"
3) "0"
4) "2"
127.0.0.1:6379> ltrim l:list 0 2
OK
127.0.0.1:6379> lrange l:list 0 -1
1) "6"
2) "5"
3) "0"
```

- 向列表中插入元素

LINSERT key BEFORE|AFTER pivot value

该命令首先会在列表中从左到右查找值为pivot的元素，然后根据第二个参数是BEFORE还是AFTER来决定将value插入到该元素的前面还是后面。

```
127.0.0.1:6379> lrange list 0 -1
1) "3"
2) "2"
3) "1"
127.0.0.1:6379> linsert list after 3 4
(integer) 4
127.0.0.1:6379> lrange list 0 -1
1) "3"
2) "4"
3) "2"
4) "1"
```

- 将元素从一个列表转移到另一个列表中

RPOPLPUSH source destination

```
127.0.0.1:6379> rpoplpush list newlist
"1"
# LRANGE key start stop 获取列表指定范围内的元素
127.0.0.1:6379> lrange newlist 0 -1
1) "1"
127.0.0.1:6379> lrange list 0 -1
1) "3"
2) "4"
3) "2"
```

4.4 Set类型

Set类型的数据是有序且不可重复。

Redis Sets 是无序的无重复的字符串集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合中最大的成员数为 $2^{32} - 1$ (4294967295, 每个集合可存储40多亿个成员)。

4.4.1 命令

- 添加元素
- SADD key member1 [member2] 向集合添加一个或多个成员

```
127.0.0.1:6379> sadd set1 1 2 3 3 4 5 5
(integer) 5
```

- 删除元素
- SREM key member1 [member2] 移除集合中一个或多个成员

```
127.0.0.1:6379> sadd set1 1 2 3 3 4 5 5
(integer) 5
127.0.0.1:6379> srem set1 3
(integer) 1
127.0.0.1:6379> smembers set1
1) "1"
2) "2"
3) "4"
4) "5"
```

- 查看元素

```
127.0.0.1:6379> smembers set1
1) "1"
2) "2"
3) "4"
4) "5"
```

- 判断元素是否存在
- SISMEMBER key member 判断 member 元素是否是集合 key 的成员

```
127.0.0.1:6379> sismember set1 6
(integer) 0
```

4.4.2 运算命令

- 差集运算

```
127.0.0.1:6379> sadd set3 2 3 4
(integer) 3
127.0.0.1:6379> sadd set4 1 2 3
(integer) 3
#SDIFF key1 [key2] 返回给定所有集合的差集
#SDIFFSTORE destination key1 [key2] 返回给定所有集合的差集并存储在 destination 中
127.0.0.1:6379> sdiff set4 set3
1) "1"
127.0.0.1:6379> sdiff set3 set4
1) "4"
```

- 交集运算
- SINTER key1 [key2] 返回给定所有集合的交集
- SINTERSTORE destination key1 [key2] 返回给定所有集合的交集并存储在 destination 中

```
127.0.0.1:6379> sinter set3 set4
1) "2"
2) "3"
```

- 并集运算
- SUNION key1 [key2] 返回所有给定集合的并集
- SUNIONSTORE destination key1 [key2] 所有给定集合的并集存储在 destination 集合中

```
127.0.0.1:6379> sunion set3 set4
1) "1"
2) "2"
3) "3"
4) "4"
```

4.4.3 其他命令（自学）

- 获得集合中元素的个数

SCARD key

```
# SMEMBERS key 返回集合中的所有成员
127.0.0.1:6379> smembers setA
1) "1"
2) "2"
3) "3"
127.0.0.1:6379> scard setA
(integer) 3
```

- 从集合中弹出一个元素
- SPOP key 移除并返回集合中的一个随机元素

```
127.0.0.1:6379> spop setA
"1"
```

注意：由于集合是无序的，所有SPOP命令会从集合中随机选择一个元素弹出

SMOVE source destination member 将 member 元素从 source 集合移动到 destination 集合

4.5 SortedSet类型zset

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

Zset在设置时，会给设置一个分数，通过分数，可以进行排序。

4.5.1 命令

- 添加元素

```
127.0.0.1:6379> zadd zset1 1 haha 2 hehe 0 heihei
(integer) 3
```

- 删除元素

```
127.0.0.1:6379> zrem zset1 haha
(integer) 1
```

- 获得排名在某个范围的元素列表

```
127.0.0.1:6379> zrange zset1 0 3
1) "heihei"
2) "hehe"
127.0.0.1:6379> zrevrange zset1 0 3
1) "hehe"
2) "heihei"
127.0.0.1:6379> zrevrange zset1 0 3 withscores
1) "hehe"
2) "2"
3) "heihei"
4) "0"
```

4.5.2 其他命令（自学）

- 获得指定分数范围的元素

```
ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]
```

```
127.0.0.1:6379> ZRANGEBYSCORE scoreboard 90 97 WITHSCORES
1) "wangwu"
2) "94"
3) "lisi"
4) "97"
127.0.0.1:6379> ZRANGEBYSCORE scoreboard 70 100 limit 1 2
1) "wangwu"
2) "lisi"
```

- 增加某个元素的分数，返回值是更改后的分数。

`ZINCRBY key increment member`

```
127.0.0.1:6379> ZINCRBY scoreboard 4 lisi
"101"
```

- 获得集合中元素的数量

`ZCARD key`

```
127.0.0.1:6379> ZCARD scoreboard
(integer) 3
```

- 获得指定分数范围内的元素个数

`ZCOUNT key min max`

```
127.0.0.1:6379> ZCOUNT scoreboard 80 90
(integer) 1
```

- 按照排名范围删除元素

`ZREMRANGEBYRANK key start stop`

```
127.0.0.1:6379> ZREMRANGEBYRANK scoreboard 0 1
(integer) 2
127.0.0.1:6379> ZRANGE scoreboard 0 -1
1) "lisi"
```

- 按照分数范围删除元素

`ZREMRANGEBYSCORE key min max`

```
127.0.0.1:6379> zadd scoreboard 84 zhangsan
(integer) 1
127.0.0.1:6379> ZREMRANGEBYSCORE scoreboard 80 100
(integer) 1
```

- 获取元素的排名

从小到大

ZRANK key member

```
127.0.0.1:6379> ZRANK scoreboard lisi  
(integer) 0
```

从大到小

ZREVRANK key member

```
127.0.0.1:6379> ZREVRANK scoreboard zhangsan  
(integer) 1
```

4.5.3 应用：商品销售排行榜

根据商品销售量对商品进行排行显示，定义sorted set集合，商品销售量为元素的分数。

定义商品销售排行榜key：items:sellsort

写入商品销售量：

商品编号1001的销量是9，商品编号1002的销量是10

```
192.168.101.3:7007> ZADD items:sellsort9 1001 10 1002
```

商品编号1001的销量加1

```
192.168.101.3:7001> ZINCRBYitems:sellsort 1 1001
```

商品销量前10名：

```
192.168.101.3:7001> ZRANGEitems:sellsort 0 9 withscores
```

第五节 Keys 命令

- Keys * 查看所有的key

5.1 设置Key的生存时间

Redis在实际使用过程中更多的用作缓存，然而缓存的数据一般都是需要设置生存时间的，即：到期后数据销毁。

EXPIRE key seconds 设置key的生存时间（单位：秒）key在多少秒后会自动删除

*TTL key 查看key**剩余的生存时间*

PERSIST key 清除生存时间

PEXPIRE key milliseconds 生存时间设置单位为：毫秒

例子：

设置test的值为1

```
192.168.101.3:7002> set test 1
```

OK

获取test的值

```
192.168.101.3:7002> get test
```

"1"

设置test的生存时间为5秒

```
192.168.101.3:7002> EXPIRE test 5
```

(integer) 1

查看test的生成时间,还有1秒删除

```
192.168.101.3:7002> TTL test
```

(integer) 1

```
192.168.101.3:7002> TTL test
```

(integer) -2

获取test的值,已经删除

```
192.168.101.3:7002> get test
```

(nil)

5.2 其他命令 (自学)

- keys

返回满足给定pattern 的所有key

```
redis 127.0.0.1:6379> keys mylist*
```

1) "mylist"

2) "mylist5"

3) "mylist6"

4) "mylist7"

5) "mylist8"

- exists

确认一个key 是否存在

```
redis 127.0.0.1:6379> exists HongWan
```

(integer) 0

```
redis 127.0.0.1:6379> exists age
```

(integer) 1

从结果来看数据库中不存在HongWan 这个key , 但是age 这个key 是存在的

- del

删除一个key

```
redis 127.0.0.1:6379> del age
(integer) 1
redis 127.0.0.1:6379> exists age
(integer) 0
```

- rename

重命名key

```
redis 127.0.0.1:6379[1]> keys *
1) "age"
redis 127.0.0.1:6379[1]> rename age age_new
OK
redis 127.0.0.1:6379[1]> keys *
1) "age_new"
```

age 成功的被我们改名为age_new 了

- type

返回值的类型

```
redis 127.0.0.1:6379> type addr
string
redis 127.0.0.1:6379> type myzset2
zset
redis 127.0.0.1:6379> type mylist
list
```

这个方法可以非常简单的判断出值的类型

第六节 Redis持久化

Redis 持久化

<http://doc.redisfans.com/topic/persistence.html>Redis

提供了多种不同级别的持久化方式：

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

RDB 的优点

- RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。这种文件非常适合用于进行备份：比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也

备份一个 RDB 文件。这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。

- RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

RDB 的缺点

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

AOF 的优点

1. 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync，每秒钟一次 fsync，或者每次执行写入命令时 fsync。AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
2. AOF 文件是一个只进行追加操作的日志文件（append only log），因此对 AOF 文件的写入不需要进行 seek，即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等），redis-check-aof 工具也可以轻易地修复这种问题。

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写：重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作，这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被别人读懂，对文件进行分析（parse）也很轻松。导出（export）AOF 文件也非常简单：举个例子，如果你不小心执行了 FLUSHALL 命令，但只要 AOF 文件未被重写，那么只要停止服务器，移除 AOF 文件末尾的 FLUSHALL 命令，并重启 Redis，就可以将数据集恢复到 FLUSHALL 执行之前的状态。

AOF 的缺点

对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。

根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB。在一般情况下，每秒 fsync 的性能依然非常高，而关闭 fsync 可以让 AOF 的速度和 RDB 一样快，即使在高负荷之下也是如此。不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。

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

6.1 rdb方式

RDB方式的持久化是通过快照（snapshotting）完成的，当符合一定条件时Redis会自动将内存中的数据进行快照并持久化到硬盘。

RDB是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

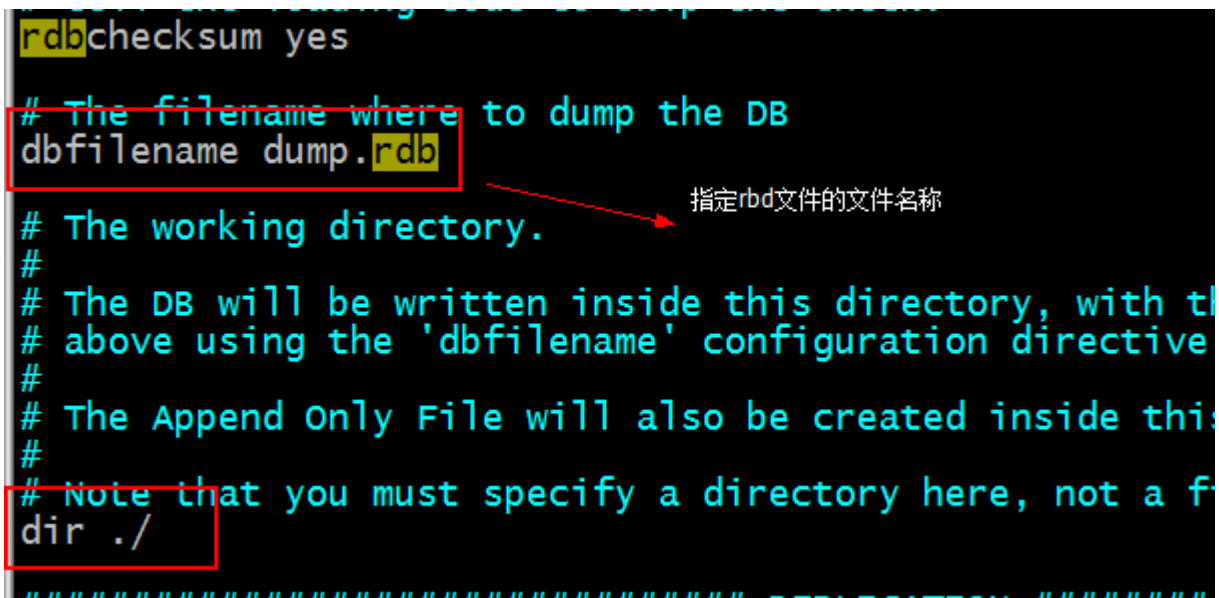
save 900 1

save 300 10

save 60 10000

save开头的一行就是持久化配置，可以配置多个条件（每行配置一个条件），每个条件之间是“或”的关系，“save 900 1”表示15分钟（900秒钟）内至少1个键被更改则进行快照，“save 300 10”表示5分钟（300秒）内至少10个键被更改则进行快照。

默认Redis会把快照文件存储为当前目录下一个名为dump.rdb的文件。要修改文件的存储路径和名称，可以通过修改配置文件redis.conf实现：



```

rdbchecksum yes
# The filename where to dump the DB
dbfilename dump.rdb
# The working directory.
# The DB will be written inside this directory, with the
# above using the 'dbfilename' configuration directive
# The Append Only File will also be created inside this
# Note that you must specify a directory here, not a file
dir ./

```

指定rdb文件的文件名称

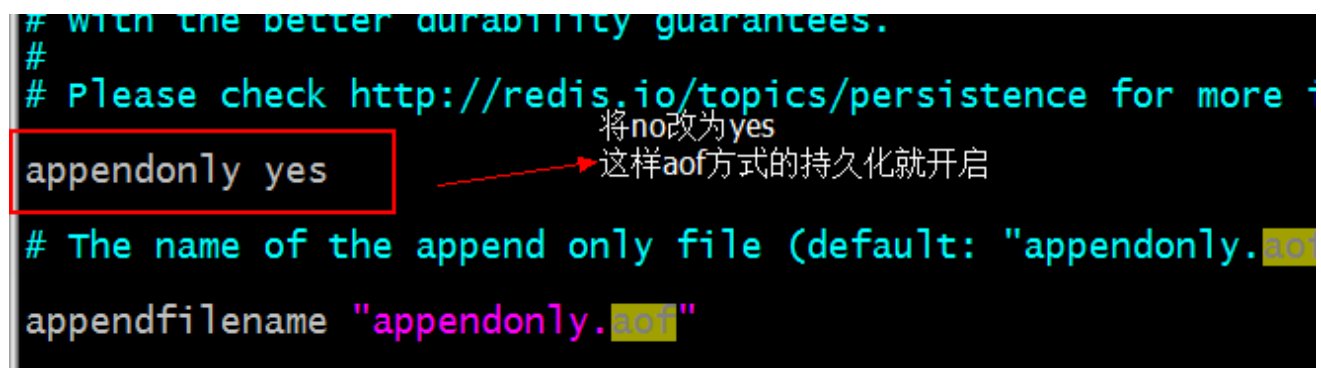
Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存。根据数据量大小与结构和服务性能不同，这个时间也不同。通常将记录一千万个字符串类型键、大小为1GB的快照文件载入到内存中需要花费20~30秒钟。

问题总结：

通过RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这就需要开发者根据具体的应用场合，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受的范围。如果数据很重要以至于无法承受任何损失，则可以考虑使用AOF方式进行持久化。

6.2 AOF方式

aof是默认不开启的，需要手动设置。



```

# With the better durability guarantees.
# Please check http://redis.io/topics/persistence for more
appendonly yes
# The name of the append only file (default: "appendonly.aof")
appendfilename "appendonly.aof"

```

将no改为yes
这样aof方式的持久化就开启

如果rdb方式和aof方式同时使用的话，那么默认从aof文件中加载数据。



```
192.168.242.130 | 192.168.242.130 (2) | 192.168.242.130 (1) x | S
$6
SELECT
$1
1
*3
$3
set
$2
s1
$3
111
~
```

第七节 事务

Redis 事务可以一次执行多个命令，并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。一个事务从开始到执行会经历以下三个阶段：

开始事务。命令入队。执行事务。

7.1 Redis 事务命令

DISCARD 取消事务，放弃执行事务块内的所有命令。EXEC 执行所有事务块内的命令。MULTI 标记一个事务块的开始。UNWATCH 取消 WATCH 命令对所有 key 的监视。WATCH key [key ...] 监视一个(或多个) key，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

7.2 实例

事务的例子，它先以 MULTI 开始一个事务，然后将多个命令入队到事务中，最后由 EXEC 命令触发事务，一并执行事务中的所有命令：

```
redis 127.0.0.1:6379> MULTI
OK

redis 127.0.0.1:6379> SET book-name "Mastering C++ in 21 days"
QUEUED

redis 127.0.0.1:6379> GET book-name
QUEUED

redis 127.0.0.1:6379> SADD tag "C++" "Programming" "Mastering Series"
QUEUED

redis 127.0.0.1:6379> SMEMBERS tag
QUEUED

redis 127.0.0.1:6379> EXEC
```

```
1) OK
2) "Mastering C++ in 21 days"
3) (integer) 3
4) 1) "Mastering Series"
   2) "C++"
   3) "Programming"
```

第八节 Redis 数据备份与恢复

8.1 备份

Redis SAVE 命令用于创建当前数据库的备份。

语法 redis Save 命令基本语法如下：

redis 127.0.0.1:6379> SAVE 实例 redis 127.0.0.1:6379> SAVE OK 该命令将在 redis 安装目录中创建dump.rdb文件。

Bgsave 创建 redis 备份文件也可以使用命令 BGSAVE，该命令在后台执行。

实例 127.0.0.1:6379> BGSAVE

Background saving started

8.2 恢复数据

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

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

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

第九节 Redis主从复制

为了高可用，引入Redis的主从复制的概念。

9.1 准备工作

完成主从复制，最少需要两台服务器，讲学方便，在一台服务器中演示即可。

但是一台服务器中需要启动两个Redis

第一步：复制一个Redis

```
[root@redis01 redis0707]# cp bin/ bin2 -r
```

第二步：修改端口

将bin2目录下的redis.conf文件中的端口修改为6380

```
# Accept connections on
# If port 0 is specific
port 6380
# TCP listen() backlog
```

9.2 主机配置

无需配置

9.3 从机配置

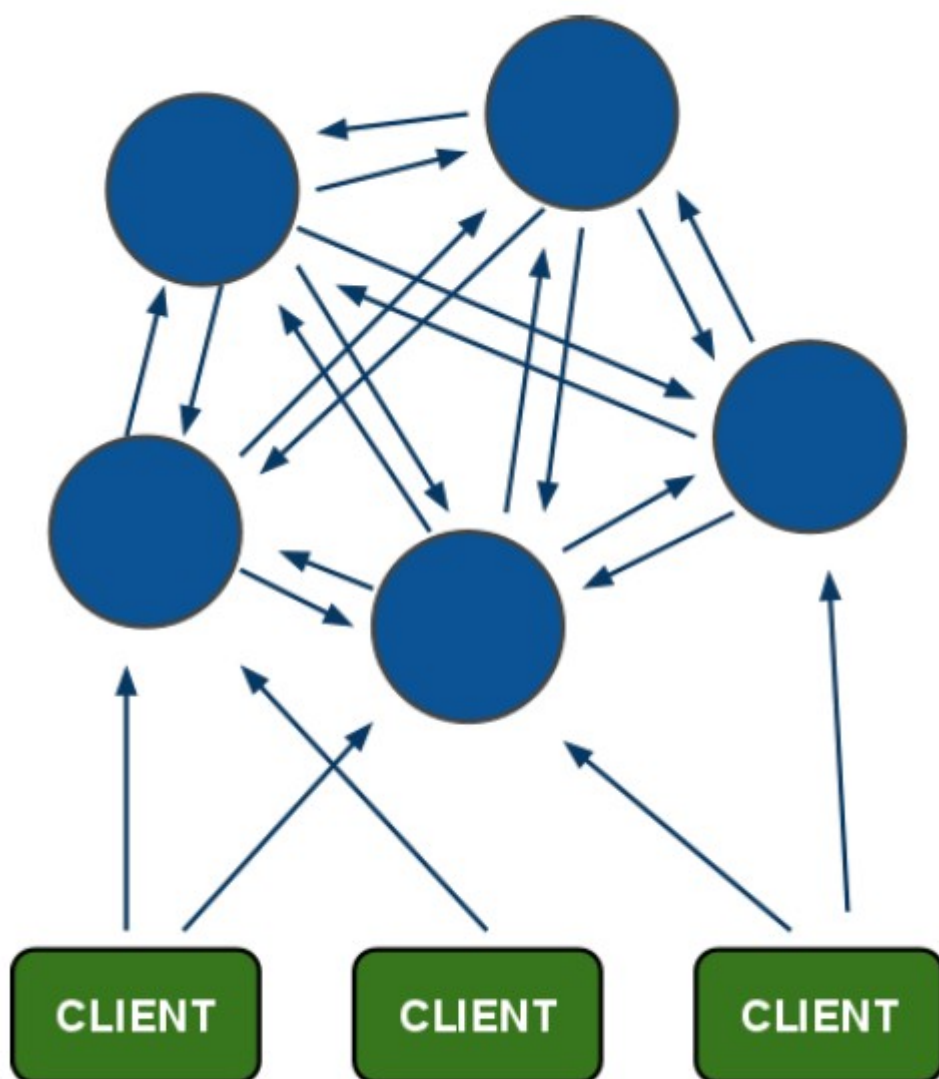
```
# sections of this file) with
# 3) Replication is automatic and
# network partition slaves aut
# and resynchronize with them.
#
slaveof 127.0.0.1 6379
# If the master is password prote
# directive below) it is possible
# starting the replication synchr
```

打开注释并
将主机的ip和
端口配置

从机是只读的。

第十节 Redis集群

10.1 Redis-Cluster架构图

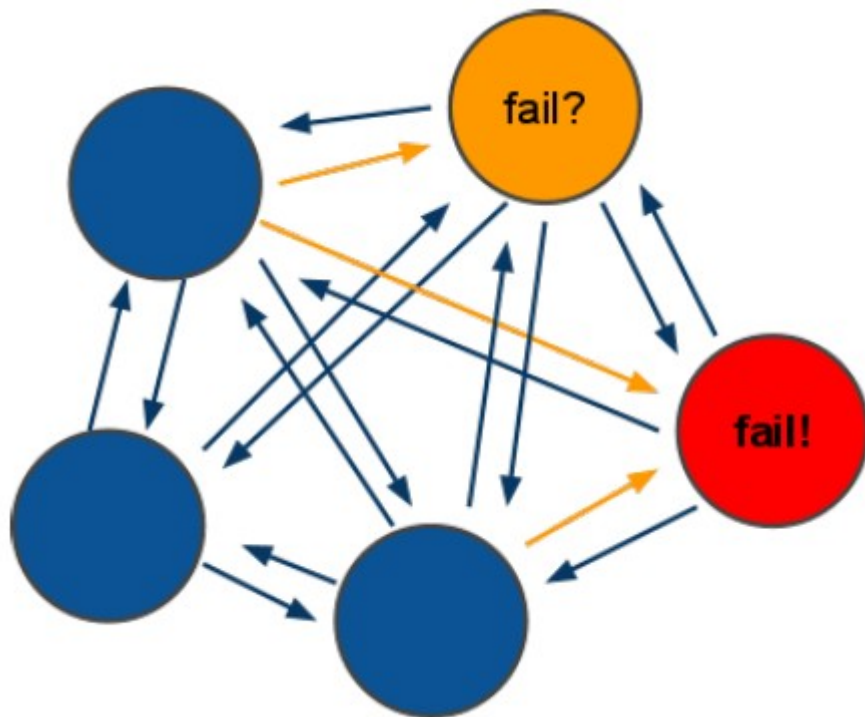


- 1、 集群通信是通过“ping-pong”机制进行通信；
- 2、 客户端不需要将所有的节点都连接上，只需要连接其中一个节点即可。
- 3、 集群中存储数据是存储到一个个的槽中，集群中槽的个数是固定的：16384，槽的编号是【0-16383】。在集群中存储数据时，会根据key进行计算，计算出一个结果，然后将这个结果和16384取余，余数就是这个key将要存储的槽的编号。

注意：槽的编号之间不能断开。

槽的计算会将数据保存的很平均，不会产生一个槽满一个槽空的情况。

10.2 Redis-Cluster投票：容错



什么时候整个集群不可用(cluster_state:fail)?

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

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

10.3 集群搭建

由于集群的脚本是用ruby语言编写的，所以需要准备ruby的环境

10.3.1 Ruby环境准备

需要ruby环境。搭建集群的脚本是ruby实现的。

redis集群管理工具redis-trib.rb依赖ruby环境，首先需要安装ruby环境：

安装ruby

```
yum install -y ruby
yum install -y rubygems
```

安装ruby和redis的接口程序

拷贝redis-3.0.0.gem至/u!查看源码文件](F:\文档\Redis详解\查看源码文件.png)sr/local下

执行：

```
gem install /usr/local/redis-3.0.0.gem
```

```
-rw-r--r--. 1 root root 20460 10月 24 17:49 zmalloc.o
[root@redis01 src]# ll *.rb
-rwxrwxr-x. 1 root root 48141 4月 1 2015 redis-trib.rb
[root@redis01 src]#
```

10.3.2 机器准备

集群环境最少要三台机器（master），每个主机都需要配置一个从机。即总共需要6台机器。

6台机器的端口号如下：

7001

7002

7003

7004

7005

7006

第一步：拷贝出6个目录

```
[root@redis01 redis-cluster]# cp redis01/redis02 -r
[root@redis01 redis-cluster]# cp redis01/redis03 -r
[root@redis01 redis-cluster]# cp redis01/redis04 -r
[root@redis01 redis-cluster]# cp redis01/redis05 -r
[root@redis01 redis-cluster]# cp redis01/redis06 -r
[root@redis01 redis-cluster]# ll
drwxr-xr-x. 2 root root 4096 10月 25 00:28 redis01
drwxr-xr-x. 2 root root 4096 10月 25 00:29 redis02
drwxr-xr-x. 2 root root 4096 10月 25 00:29 redis03
```

```
drwxr-xr-x. 2 root root 4096 10月 25 00:29 redis04
drwxr-xr-x. 2 root root 4096 10月 25 00:29 redis05
drwxr-xr-x. 2 root root 4096 10月 25 00:29 redis06
```

第二步：修改端口

```
# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket
port 7006
```

第三步：修改配置，配置允许集群的配置

将cluster-enabled 的值改为yes

```
# started as cluster nodes can.
# cluster node enable the cluster
#
cluster-enabled yes
```

第四步：启动6台redis

```
[root@redis01 redis-cluster]# vi start-all.sh
1 cd redis01
2 ./redis-server redis.conf
3 cd ..
4 cd redis02
5 ./redis-server redis.conf
6 cd ..
7 cd redis03
8 ./redis-server redis.conf
9 cd ..
10 cd redis04
11 ./redis-server redis.conf
12 cd ..
13 cd redis05
14 ./redis-server redis.conf
15 cd ..
16 cd redis06
17 ./redis-server redis.conf
"start-all.sh" [新] 35L, 270C 已写入

[root@redis01 redis-cluster]# ll
drwxr-xr-x. 2 root root 4096 10月 25 00:33 redis01
drwxr-xr-x. 2 root root 4096 10月 25 00:35 redis02
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis03
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis04
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis05
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis06

-rw-r--r--. 1 root root 270 10月 25 00:37 start-all.sh
```

```
[root@redis01 redis-cluster]# ./start-all.sh
-bash: ./start-all.sh: 权限不够
[root@redis01 redis-cluster]# chmod 777 start-all.sh
[root@redis01 redis-cluster]# ll
drwxr-xr-x. 2 root root 4096 10月 25 00:33 redis01
drwxr-xr-x. 2 root root 4096 10月 25 00:35 redis02
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis03
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis04
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis05
drwxr-xr-x. 2 root root 4096 10月 25 00:34 redis06
-rwxrwxrwx. 1 root root 270 10月 25 00:37 start-all.sh
[root@redis01 redis-cluster]# ./start-all.sh
```

看到以下信息，则说明启动成功

```
root      8033      1  0 00:24 ?        00:00:04 ./redis-server *:6379
root      8149      1  0 00:01 ?        00:00:03 ./redis-server *:6380
root      8430      1  0 00:38 ?        00:00:00 ./redis-server *:7001 [cluster]
root      8432      1  0 00:38 ?        00:00:00 ./redis-server *:7002 [cluster]
root      8434      1  0 00:38 ?        00:00:00 ./redis-server *:7003 [cluster]
root      8436      1  0 00:38 ?        00:00:00 ./redis-server *:7004 [cluster]
root      8438      1  0 00:38 ?        00:00:00 ./redis-server *:7005 [cluster]
root      8440      1  0 00:38 ?        00:00:00 ./redis-server *:7006 [cluster]
root      8454    7960  0 00:39 pts/2    00:00:00 grep redis
```

第五步：集群

将redis-trib.rb文件复制到redis0707目录下

```
[root@redis01 redis0707]# ll
总用量 60
drwxr-xr-x. 2 root root 4096 10月 25 00:16 bin
drwxr-xr-x. 2 root root 4096 10月 25 00:16 bin2
drwxr-xr-x. 8 root root 4096 10月 25 00:37 redis-cluster
-rwxr-xr-x. 1 root root 48141 10月 25 00:41 redis-trib.rb
```

执行命令

```
[root@redis01 redis0707]# ./redis-trib.rb create --replicas 1 127.0.0.1:7001 127.0.0.1:7002
127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006
>>> Creating cluster
Connecting to node 127.0.0.1:7001: OK
Connecting to node 127.0.0.1:7002: OK
Connecting to node 127.0.0.1:7003: OK
Connecting to node 127.0.0.1:7004: OK
Connecting to node 127.0.0.1:7005: OK
Connecting to node 127.0.0.1:7006: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
127.0.0.1:7001
127.0.0.1:7002
127.0.0.1:7003
Adding replica 127.0.0.1:7004 to 127.0.0.1:7001
Adding replica 127.0.0.1:7005 to 127.0.0.1:7002
Adding replica 127.0.0.1:7006 to 127.0.0.1:7003
```



```

M: e2669f9cef230acfe90f01e207a0d410a6dbb489 127.0.0.1:7001
  slots:0-5460 (5461 slots) master
M: f8cf8ced81e5a111181d13ee8206dd39b3f46db4 127.0.0.1:7002
  slots:5461-10922 (5462 slots) master
M: c4e8a6615f4e8b2ba408207ac9a16de9af848420 127.0.0.1:7003
  slots:10923-16383 (5461 slots) master
S: 95a979b999f9cb7071763370f0c2a275abeabca9 127.0.0.1:7004
  replicates e2669f9cef230acfe90f01e207a0d410a6dbb489
S: 2e40daf6cc502ca175115f92393ebc258818efe8 127.0.0.1:7005
  replicates f8cf8ced81e5a111181d13ee8206dd39b3f46db4
S: d92e6a23ffadc2aa0f8d8b34ddc61f4c0ae29412 127.0.0.1:7006
  replicates c4e8a6615f4e8b2ba408207ac9a16de9af848420
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join.....
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: e2669f9cef230acfe90f01e207a0d410a6dbb489 127.0.0.1:7001
  slots:0-5460 (5461 slots) master
M: f8cf8ced81e5a111181d13ee8206dd39b3f46db4 127.0.0.1:7002
  slots:5461-10922 (5462 slots) master
M: c4e8a6615f4e8b2ba408207ac9a16de9af848420 127.0.0.1:7003
  slots:10923-16383 (5461 slots) master
M: 95a979b999f9cb7071763370f0c2a275abeabca9 127.0.0.1:7004
  slots: (0 slots) master
  replicates e2669f9cef230acfe90f01e207a0d410a6dbb489
M: 2e40daf6cc502ca175115f92393ebc258818efe8 127.0.0.1:7005
  slots: (0 slots) master
  replicates f8cf8ced81e5a111181d13ee8206dd39b3f46db4
M: d92e6a23ffadc2aa0f8d8b34ddc61f4c0ae29412 127.0.0.1:7006
  slots: (0 slots) master
  replicates c4e8a6615f4e8b2ba408207ac9a16de9af848420
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.

```

10.4 连接集群

```

[root@redis01 bin]# ./redis-cli -p 7001 -c
127.0.0.1:7001> set s1 111
-> Redirected to slot [15224] located at 127.0.0.1:7003
OK
127.0.0.1:7003> get s1
"111"
127.0.0.1:7003> set s2 222
-> Redirected to slot [2843] located at 127.0.0.1:7001
OK

```

查看集群信息

```
OK
127.0.0.1:7001> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_sent:3027
cluster_stats_messages_received:3027
127.0.0.1:7001> cluster nodes
d92e6a23ffadc2a0f8d8b34ddc61f4c0ae29412 127.0.0.1:7006 slave c4e8a6615f4e8b2ba408207ac9a16de9af848420 0 1445706456891 6
connected
95a979b999f9cb7071763370f0c2a275abeabca9 127.0.0.1:7004 slave e2669f9cef230acfef90f01e207a0d410a6dbb489 0 1445706454870 4
connected
c4e8a6615f4e8b2ba408207ac9a16de9af848420 127.0.0.1:7003 master - 0 1445706452853 3 connected 10923-16383
2e40daf6cc502ca175115f92393ebc258818efe8 127.0.0.1:7005 slave f8cf8ced81e5a111181d13ee8206dd39b3f46db4 0 1445706454365 5
connected
e2669f9cef230acfef90f01e207a0d410a6dbb489 127.0.0.1:7001 myself,master - 0 0 1 connected 0-5460
f8cf8ced81e5a111181d13ee8206dd39b3f46db4 127.0.0.1:7002 master - 0 1445706455879 2 connected 5461-10922
127.0.0.1:7001>
```

10.5 节点扩展

10.5.1 添加主节点

集群创建成功后可以向集群中添加节点，下面是添加一个master主节点

添加7007结点，参考集群结点规划章节添加一个7007目录作为新节点。

执行下边命令：

```
./redis-trib.rb add-node 192.168.101.3:7007 192.168.101.3:7001
```

```
[root@server01 7007]# ./redis-trib.rb add-node 192.168.101.3:7007 192.168.101.3:7001
>>> Adding node 192.168.101.3:7007 to cluster 192.168.101.3:7001
Connecting to node 192.168.101.3:7001: OK
Connecting to node 192.168.101.3:7003: OK
Connecting to node 192.168.101.3:7006: OK
Connecting to node 192.168.101.3:7002: OK
Connecting to node 192.168.101.3:7005: OK
Connecting to node 192.168.101.3:7004: OK
>>> Performing cluster check (using node 192.168.101.3:7001)
M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001
slots:0-5460 (5461 slots) master
1 additional replica(s)
M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003
slots:10923-16383 (5461 slots) master
1 additional replica(s)
S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006
slots: (0 slots) slave
replicates 1a8420896c3ff60b70c716e8480de8e50749ee65
M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002
slots:5461-10922 (5462 slots) master
1 additional replica(s)
S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005
slots: (0 slots) slave
replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841
S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004
slots: (0 slots) slave
replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
[OK] All nodes agree about slots configuration.
>>> check for open slots...
>>> check slots coverage...
[OK] All 16384 slots covered.
Connecting to node 192.168.101.3:7007: OK
>>> Send CLUSTER MEET to node 192.168.101.3:7007 to make it join the cluster.
[OK] New node added correctly.
```

查看集群结点发现7007已添加到集群中：

```
192.168.101.3:7005> cluster nodes
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430155626174 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430155621629 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430155627185 2 connected 5461-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430155625164 9 connected 10923-16383
15b809eadae8895e36bcd8b8144f61bbbf38fb 192.168.101.3:7007 master - 0 1430155628700 0 connected
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430155628197 1 connected 0-5460
```

10.5.2 hash槽重新分配

添加完主节点需要对主节点进行hash槽分配这样该主节点才可以存储数据。

redis集群有16384个槽，集群中的每个节点分配自己槽，通过查看集群节点可以看到槽占用情况。

```
192.168.101.3:7005> cluster nodes
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430155241550 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430155240540 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430155239532 2 connected 5461-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430155243568 9 connected 10923-16383
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430155242560 1 connected 0-5460
192.168.101.3:7005>
```

每个master节点都分配了一定数量的槽

给刚添加的7007节点分配槽：

第一步：连接上集群

```
./redis-trib.rb reshard 192.168.101.3:7001
```

第二步：输入要分配的槽数量

```
[root@server01 redis~cluster]# ./redis-trib.rb reshard 192.168.101.3:7001
Connecting to node 192.168.101.3:7001: OK
Connecting to node 192.168.101.3:7003: OK
Connecting to node 192.168.101.3:7006: OK
Connecting to node 192.168.101.3:7002: OK
Connecting to node 192.168.101.3:7005: OK
Connecting to node 192.168.101.3:7007: OK
Connecting to node 192.168.101.3:7004: OK
>>> Performing Cluster Check (using node 192.168.101.3:7001)
M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001
slots:999-5460 (4462 slots) master
1 additional replica(s)
M: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003
slots:11922-16383 (4462 slots) master
1 additional replica(s)
S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006
slots: (0 slots) slave
replicates 1a8420896c3ff60b70c716e8480de8e50749ee65
M: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002
slots:6462-10922 (4461 slots) master
1 additional replica(s)
S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005
slots: (0 slots) slave
replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841
M: 15b809eadae88955e36bcd8b8144f61bbbf38fb 192.168.101.3:7007
slots:0-998,5461-6461,10923-11921 (2999 slots) master
0 additional replica(s)
S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004
slots: (0 slots) slave
replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
How many slots do you want to move (from 1 to 16384)?
```

这里输入要分配的槽数量

输入 500表示要分配500个槽

第三步：输入接收槽的节点id

```
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
How many slots do you want to move (from 1 to 16384)? 500
what is the receiving node ID?
```

输入接收槽的结点id

这里准备给7007分配槽，通过cluster nodes查看7007结点id为15b809eadae88955e36bcd8b8144f61bbaf38fb

输入：

15b809eadae88955e36bcd8b8144f61bbaf38fb

第四步：输入源结点id

```
How many slots do you want to move (from 1 to 16384)? 500
what is the receiving node ID? 15b809eadae88955e36bcd8b8144f61bbaf38fb
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:
```

输入源结点id，槽将从源结点中拿，分配后的槽在源结点中就不存在了，输入all从所有源结点中获取槽，输入done取消分配

这里输入all

第五步：输入yes开始移动槽到目标节点id

```
Do you want to proceed with the proposed reshard plan (yes/no)? █
```

10.5.3 添加从节点

集群创建成功后可以向集群中添加节点，下面是添加一个slave从节点。

添加7008从结点，将7008作为7007的从结点。

```
./redis-trib.rb add-node --slave --master-id 主节点id 添加节点的ip和端口 集群中已存在节点ip和端口
```

执行如下命令：

```
./redis-trib.rb add-node --slave --master-id cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
192.168.101.3:7008 192.168.101.3:7001
```

cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 是7007结点的id，可通过cluster nodes查看。

```
[root@server01 redis-cluster]# ./redis-trib.rb add-node --slave --master-id cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7008 192.168.101.3:7001
>>> Adding node 192.168.101.3:7008 to cluster 192.168.101.3:7001
connecting to node 192.168.101.3:7001: OK
connecting to node 192.168.101.3:7003: OK
connecting to node 192.168.101.3:7006: OK
connecting to node 192.168.101.3:7002: OK
connecting to node 192.168.101.3:7005: OK
connecting to node 192.168.101.3:7007: OK
connecting to node 192.168.101.3:7004: OK
>>> Performing cluster check (using node 192.168.101.3:7001)
M: cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001
slots:1166-5460 (4295 slots) master
1 additional replica(s)
W: 1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003
slots:12088-16383 (4296 slots) master
1 additional replica(s)
S: 444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006
slots: (0 slots) slave
replicates 1a8420896c3ff60b70c716e8480de8e50749ee65
W: 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002
slots:6628-10922 (4295 slots) master
1 additional replica(s)
S: d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005
slots: (0 slots) slave
replicates 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841
W: 15b809eadae88955e36bcd8b8144f61bbbf38fb 192.168.101.3:7007
slots:0-1165,5461-6627,10923-12087 (3498 slots) master
0 additional replica(s)
S: 69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004
slots: (0 slots) slave
replicates cad9f7413ec6842c971dbcc2c48b4ca959eb5db4
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
Connecting to node 192.168.101.3:7008: OK
>>> Send CLUSTER MEET to node 192.168.101.3:7008 to make it join the cluster.
waiting for the cluster to join.
>>> Configure node as replica of 192.168.101.3:7001.
[OK] New node added correctly.
```

注意：如果原来该结点在集群中的配置信息已经生成cluster-config-file指定的配置文件中（如果cluster-config-file没有指定则默认为nodes.conf），这时可能会报错：

[ERR]Node XXXXXX is not empty. Either the node already knows other nodes (check withCLUSTER NODES) or contains some key in database 0

解决方法是删除生成的配置文件nodes.conf，删除后再执行./redis-trib.rb add-node指令

查看集群中的结点，刚添加的7008为7007的从节点：

```
192.168.101.3:7005> cluster nodes
05dbaf8059630157c245dfe5441dbe9c26b9016d 192.168.101.3:7008 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430157051979 1 connected
69d94b4963fd94f315fba2b9f12fae1278184fe8 192.168.101.3:7004 slave cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 0 1430157046926 4 connected
444e7bedbdfa40714ee55cd3086b8f0d5511fe54 192.168.101.3:7006 slave 1a8420896c3ff60b70c716e8480de8e50749ee65 0 1430157051878 9 connected
4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 192.168.101.3:7002 master - 0 1430157049956 2 connected 6628-10922
d2421a820cc23e17a01b597866fd0f750b698ac5 192.168.101.3:7005 myself,slave 4e7c2b02f0c4f4cfe306d6ad13e0cfee90bf5841 0 0 5 connected
1a8420896c3ff60b70c716e8480de8e50749ee65 192.168.101.3:7003 master - 0 1430157050968 9 connected 12088-16383
15b809eadae88955e36bcd8b8144f61bbbf38fb 192.168.101.3:7007 master - 0 1430157052990 10 connected 0-1165 5461-6627 10923-12087
cad9f7413ec6842c971dbcc2c48b4ca959eb5db4 192.168.101.3:7001 master - 0 1430157048946 1 connected 1166-5460
```

10.5.4 删除节点

```
./redis-trib.rb del-node 127.0.0.1:70054b45eb75c8b428fbd77ab979b85080146a9bc017
```

删除已经占有hash槽的结点会失败，报错如下：

[ERR] Node 127.0.0.1:7005 is not empty!Reshard data away and try again.

需要将该结点占用的hash槽分配出去（参考hash槽重新分配章节）。

第十一节 Jedis连接集群

```
public static void main(String[] args) {
    //节点设置
    HashSet<HostAndPort> nodes = new HashSet<>();
    nodes.add(new HostAndPort("node01", 7001));
    nodes.add(new HostAndPort("node02", 7002));
    nodes.add(new HostAndPort("node03", 7003));
    nodes.add(new HostAndPort("node04", 7004));
    nodes.add(new HostAndPort("node05", 7005));
    nodes.add(new HostAndPort("node06", 7006));
```



```
//创建JedisCluster实例
JedisCluster cluster = new JedisCluster(nodes);
cluster.set("ruhua", "rightgirl");
String s = cluster.get("ruhua");
System.out.println(s);

cluster.close();
}
```

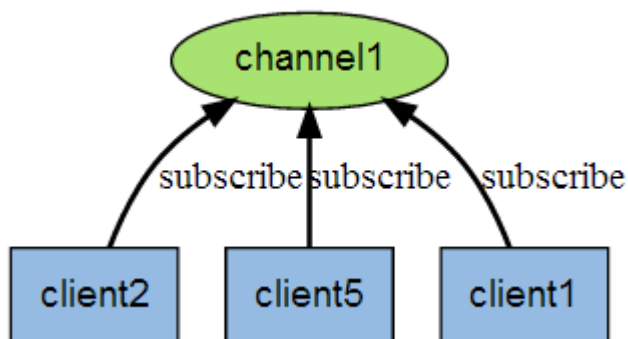
第十二节 其他(扩展)

12.1 发布与订阅

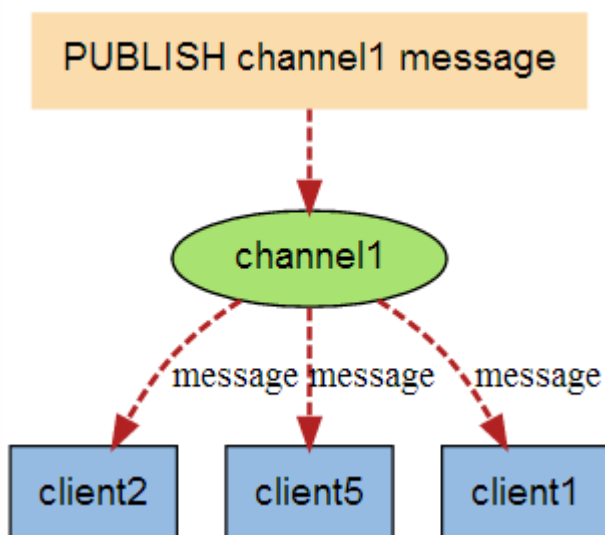
Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

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



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



12.2 管道

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

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

客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应。服务端处理命令，并将结果返回给客户端。Redis 管道技术 Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

实例 查看 redis 管道，只需要启动 redis 实例并输入以下命令：

```
$(echo -en "PING\r\n SET w3ckey redis\r\nGET w3ckey\r\nINCR visitor\r\nINCR visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379
```

+PONG +OK redis :1 :2 :3 以上实例中我们通过使用 PING 命令查看redis服务是否可用，之后我们设置了w3ckey 的值为 redis，然后我们获取 w3ckey 的值并使得 visitor 自增 3 次。

在返回的结果中我们可以看到这些命令一次性向 redis 服务提交，并最终一次性读取所有服务端的响应

管道技术的优势 管道技术最显著的优势是提高了 redis 服务的性能。

一些测试数据 在下面的测试中，我们将使用Redis的Ruby客户端，支持管道技术特性，测试管道技术对速度的提升效果。

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

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

without pipelining 1.185238 seconds with pipelining 0.250783 seconds 如你所见，开启管道后，我们的速度效率提升了5倍。

12.3 分区

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

分区的优势 通过利用多台计算机内存的和值，允许我们构造更大的数据库。通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。分区的不足 redis的一些特性在分区方面表现的不是很好：

涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。涉及多个key的redis事务不能使用。当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。分区类型 Redis 有两种类型分区。假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

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

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

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

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

用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。 $93024922 \% 4 = 2$ ，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

12.4 安全

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

实例 我们可以通过以下命令查看是否设置了密码验证：

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

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

你可以通过以下命令来修改该参数：

```
127.0.0.1:6379> CONFIG set requirepass "w3cschool.cc" OK 127.0.0.1:6379> CONFIG get requirepass 1) "requirepass" 2) "w3cschool.cc"
```

设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令。

语法 AUTH 命令基本语法格式如下：

```
127.0.0.1:6379> AUTH password 实例 127.0.0.1:6379> AUTH "w3cschool.cc" OK 127.0.0.1:6379> SET mykey "Test value" OK 127.0.0.1:6379> GET mykey "Test value"
```